Texas Instruments

TMS34020

User's Guide

1990

Computer Video Products

TMS34020 User's Guide

2564006-9721 revision *
August 1990

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

Apollo is a trademark of Apollo Computer Incorporated.
DEC, Ultrix, VAX, and *VMS* are trademarks of Digital Equipment Corporation.
IBM-PC is a registered trademark of International Business Machines Corporation.
Macintosh is a trademark of Apple Computer Incorporated.
MS-DOS is a trademark of Microsoft Corporation.
SUN-3 is a trademark of Sun Microsystems Incorporated.
UNIX is a registered trademark of AT&T.
XDS is a trademark of Texas Instruments Incorporated.

Preface

Read This First

This document describes the TMS34020 Graphics System Processor. It focuses on the TMS34020's role in applications that involve CRT-based bit-mapped graphics systems.

If You Need Assistance...

If you want to . . .	Do this...
Receive more information about Texas Instruments graphics products	Call the CRC † hotline: **(800) 232–3200** Or write to: Market Communications Manager P.O. Box 1443, MS 736 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the CRC † hotline: **(800) 232–3200**
Ask questions about product operation or report suspected problems	Call the graphics hotline: **(713) 274–2340**
Report mistakes in this document or in any other TI documentation	Send your comments to: Technical Publications Manager Texas Instruments Incorporated P.O. Box 1443, MS640 Houston, Texas 77251–9879

† Texas Instruments Customer Response Center

Related Documentation from Texas Instruments

The following TMS34010 and TMS34020 documents are available from Texas Instruments. To obtain a copy of any of these TI documents, please call the Texas Instruments Customer Response Center (CRC) at (800) 232–3200. When ordering, please identify the book by its title and its literature number.

Pixel Perspectives is a quarterly newsletter, published by the Graphics Products group of Texas Instruments Incorporated. This newsletter describes new products, discusses support for existing products, and identifies new documentation releases.

The ***TMS34020 Data Sheet*** (lit. number SPVS004) contains electrical specifications, timing information, and mechanical data for the TMS34020.

The ***TMS340 Family Code-Generation Tools User's Guide*** (lit. number SPVU004) describes the C compiler, assembler, linker, archiver, and auxiliary tools that are available for developing TMS34010 or TMS34020 code.

The ***TIGA-340 Interface User's Guide*** (lit. number SPVU015) describes the Texas Instruments Graphics Architecture (TIGA), a software interface that standardizes communication between application software and TMS340-based hardware for IBM-compatible PCs.

The ***TMS34010 Software Development Board User's Guide*** (lit. number SPVU002) describes a high-performance graphics card that aids in understanding TI graphics products. Read Pixel Perspectives for discussions of a TMS34020 version of this product and its documentation.

The ***TMS34010 User's Guide*** (lit. number SPVU001) describes the TMS34010, which is the first-generation graphics system processor in the TMS340 family of graphics products.

Notational Conventions

The following terms and conventions are used throughout this manual.

Term/Convention	Description
Rs, Rd	source register, destination register.
Rs.**XY**, Rd.**XY**	source or destination register in XY form.
Rs.**X**, Rd.**X**	X half of source or destination register.
Rs.**Y**, Rd.**Y**	Y half of source or destination register.
R	is a bit in an instruction opcode that identifies which register file the register operands are in. R=0 identifies file A; R=1 identifies file B.
PC'	is the address of the next instruction (current PC plus the length of the current instruction).
Rp	register pair.
cc	condition codes for a jump instruction.
IW, IL	16-bit immediate value (short), 32-bit immediate value (long).
SAddress, DAddress	source address, destination address.
SOffset, DOffset	source offset, destination offset.
LSB, MSB	least significant bit, most significant bit.
LSbyte, MSbyte	least significant byte, most significant byte.
LSW, MSW	least significant word, most significant word.

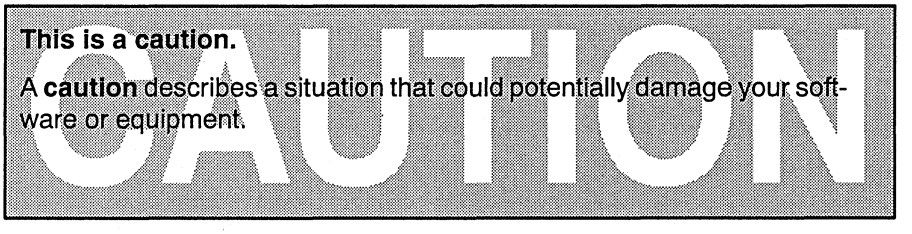
Term/Convention	Description
n	identifies a number that may have several values.
A*n*, B*n*	identifies register *n* in register file A or file B.
TOS	top of stack.
F	optional field select parameter for MOVE-field instructions. F=0 selects the field size & extension of field 0 for the move; F=1 selects the field size & extension of field 1 for the move.
xxx[[REGISTER]]	identifies a bit within a register. For example, CBP[[CONFIG]] refers to the CBP bit in the CONFIG register.

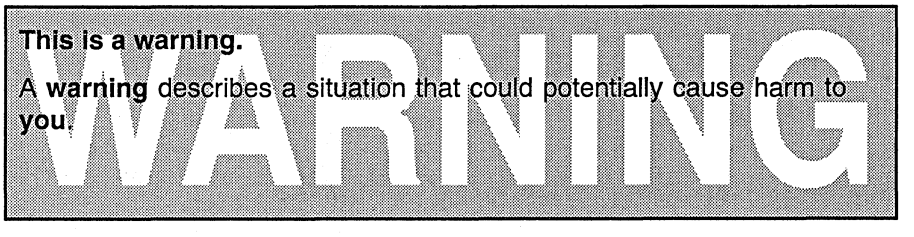
`special font`	identifies program listings, coding examples, filenames, and symbol names. For example,
	`0011 00000210 0001 .field 1, 2`
	`0012 00000212 0003 .field 3, 4`
	`0013 00000215 0006 .field 6, 3`
	`0014 00000220 .even`

boldface text	serves two purposes. In text, **boldface** identifies a key term that is being defined. In instruction syntax, **boldface** identifies the part of the instruction that you must enter as shown. For example, enter **PIXBLT B,XY** exactly as shown (`PIXBLT B,XY`).
italic text	serves two purposes. In text, *italics* emphasize important explanations. In instruction syntax, *italics* identify "placeholders" that identify the type of information you should enter for a parameter. For example, **CVXYL** *Rs, Rd* **CVXYL** is an instruction that has two parameters, *Rs* and *Rd*—you must replace *Rs* and *Rd* with actual source and destination registers (`CVXYL A0,A3`).
[ ]	identify an optional parameter. Here's an example of an instruction with an optional parameter: **CMPI** *IW, Rd* **[, W]** CMPI has 3 parameters; the first two are required, the third is optional. Note that the **, W** is bold—so if you use the optional parameter, you must type it as shown.
→	means *becomes the contents of*. In an instruction execution description, for example, *Rs* → *PC* the contents of Rs become the contents of (or replace the contents of) the program counter.
|*value*|	means take the absolute value of the item between the parallel bars.

Term/Convention	Description
{*choice1*|*choice 2*}	identifies a list; you can enter choice 1 or choice 2.
:	indicates concatenation. For example, Rd:Rd+1 forms a 64-bit register area of Rd and the next register in the same file.
*value*b , *value*B	identifies a binary integer. For example, 01b 1111B
*value*h , *value*H	identifies a hexadecimal integer. A hex number can't start with a letter—start it with a 0 instead. For example, 0FFFFh 123H

Information About Cautions and Warnings





The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Suggested References

The following books and articles provide further background information about graphics and system concepts associated with graphics:

Artwick, Bruce A. *Applied Concepts in Microcomputer Graphics*. Englewood Cliffs, New Jersey: Prentice-Hall, 1984.

Asal, Short, Preston, Simpson, Roskell, and Guttag. "The Texas Instruments 34010 Graphics System Processor." *IEEE Computer Graphics and Applications* vol.6, no.10, pp. 24—39.

Bresenham, J.E. "Algorithm for Computer Control of a Digital Plotter." *IBM Systems Journal* 4, No.1 (1965): 25—30.

Bresenham, J.E. "A Linear Algorithm for Incremental Display of Digital Arcs." *Communications of the ACM* 20 (Feb. 1977): 100—106.

Cody, William J. Jr., and William Waite. *Software Manual for the Elementary Functions*. Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

Foley, James, and Andries van Dam. *Fundamentals of Interactive Computer Graphics*. Reading, Massachusets: Addison-Wesley, 1982.

Gupta, Satish. "Architectures and Algorithms for Parallel Updates of Raster Scan Displays." Tech. Report CMU-CS-82-111, Computer Science Dept., Carnegie Mellon University, 1981.

Ingalls, D.H. "The Smalltalk Graphics Kernel." Special Issue on Smalltalk, *Byte*, August 1981, pp. 168—194.

Kernighan, B., and D. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

Killebrew, C.R. Jr., "The TMS34010 Graphics System Processor." *Byte*, December 1986, pp. 193—204.

Kochan, Stephen G. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, 1983.

Newman, W.M., and R.F. Sproull. *Principles of Interactive Computer Graphics*. 2nd ed. New York: McGraw-Hill, 1979.

Pike, Rob. "Graphics in Overlapping Bitmap Layers." *ACM Transactions On Graphics* 2 (April 1983): 135—160.

Pinkham, R., M. Novak, and K. Guttag. "Video RAM Excels at Fast Graphics." *Electronic Design*, August 18, 1983, pp. 161—168.

Pitteway, M.L.V. "Algorithm for Drawing Ellipses or Hyperbolae with a Digital Plotter." *Computer Journal* 10 (November 1967): 24—35.

Porter, T. and T. Duff. "Composing Digital Images." *Computer Graphics*, July 1984, pp. 253—259.

Sproull, R.F. and I.E. Sutherland. "A Clipping Divider." *Fall Joint Computer Conference*. Washington, DC: Thompson Books, 1968.

Van Aken, Jerry R. "An Efficient Ellipse-Drawing Algorithm." *IEEE Computer Graphics & Applications* 4 (Sept. 1984): 24—35.

Wientjes, Guttag, and Roskell. "First Graphics Processor Takes Complex Orders to Run Bit-Mapped Displays." *Electronic Design* Vol. 34, No.2 (January 23, 1986): 73—80.

Contents

Figures

Tables

Examples

Chapter 1

Overview of the TMS34020

The TMS34020 Graphics System Processor (GSP) is an advanced, 32-bit microprocessor, optimized for graphic display systems. The TMS34020 is the second generation of the TMS340 family of computer graphics products from Texas Instruments.

The TMS34020 provides a high-performance, cost-effective solution for applications that require efficient data manipulation in a graphics environment. The TMS34020 can be configured to serve in a host-based, stand-alone, or multiprocessing system. The TMS34020 has host and multiprocessor interfaces to facilitate implementation of multiple TMS34020 systems.

The TMS34020 is well supported by a full set of hardware and software development tools, including an optimizing C compiler, an assembler, software libraries, a PC-based development board, and an emulator. In addition, the TMS34020 is fully compatible with and supported by the Texas Instruments Graphics Architecture (TIGA-340).

Topics covered in this introductory section include

1.1 Key Features of the TMS34020

- Fully programmable 32-bit general-purpose processor with 512-Mbyte linear address range (bit addressable)
- Second-generation graphics system processor
  - Object code compatible with the TMS34010
  - Enhanced instruction set
  - Optimized graphics instructions
  - Direct coprocessor interface to TMS34082 floating-point processor
- Instruction cycle times:
  - TMS34020-40 ..... 100 ns
  - TMS34020-32 ..... 125 ns
- On-chip peripheral functions include
  - Programmable CRT control
  - Direct DRAM/VRAM interface
  - Direct communication with an external (host) processor
  - Communication with multiple TMS34020s
  - Functional expansion with the coprocessor interface
  - Automatic CRT display refresh
- Instruction set supports special graphics functions such as pixel processing, XY addressing, and window checking
- Programmable 1-, 2-, 4-, 8-, 16-, or 32-bit pixel size
- 16 Boolean and 6 arithmetic pixel processing options (raster-ops)
- 30 general-purpose 32-bit registers
- 512-byte LRU on-chip instruction cache
- Optimized DRAM/VRAM interface
  - Page mode for burst memory operations up to 40 Mbytes per second
  - Dynamic bus sizing (16-bit and 32-bit transfers)
  - Byte-oriented $\overline{\text{CAS}}$ strobes
  - Automatic CRT display refresh
- Flexible host processor interface
  - Supports host transfers at up to 20 Mbytes per second
  - Direct access to all of the TMS34020 address space
  - Implicit addressing (autoincrementing)
  - Prefetching for enhanced read access
- Flexible multiprocessor interface

- Programmable CRT control
  - Composite sync mode
  - Separate sync mode
  - Synchronization to external sync
- Direct support for special features of 1M VRAMs
  - Load write mask
  - Load color mask
  - Block write
  - Write using the write mask

1.2 Typical Applications of the TMS34020

The TMS34020's 32-bit processing power and its ability to handle complex data structures make it well suited for a variety of applications. Typical applications that take advantage of the TMS34020's features include

Computers
- Terminals and CRTs
- Windowing systems
- Electronic publishing
- Laser printers
- Personal computers
- Printers and plotters
- Engineering workstations
- Copiers
- Document readers
- FAX
- Imaging
- Data processing

Industrial Control
- Robotics
- Process control
- Instrumentation
- Motor control
- Navigation

Consumer Electronics
- Automotive displays
- Information terminals
- Cable TV
- Home control
- Video games

Telecommunications
- Video phones
- PBX

1.3 Major Components of the TMS34020 Architecture

The TMS340 family of processors from Texas Instruments combines the best features of general-purpose processors and graphics controllers to create a range of cost-effective, flexible, powerful graphics systems. The key features of the TMS340 family are speed, a high degree of programmability, and efficient manipulation of hardware-supported data types such as pixels and 2-dimensional pixel arrays.

With a built-in instruction cache, the ability to simultaneously access memory and registers, and an instruction set that enhances raster graphics operations, the TMS34020 provides programmable control of the CRT interface as well as the memory interface (both standard DRAM and multiport VRAM). The TMS34020's 4-gigabit (512 Mbyte) physical address space is completely bit-addressable on bit boundaries using variable-width data fields. Graphics addressing modes support 1-, 2-, 4-, 8-, 16-, and 32-bit pixels.

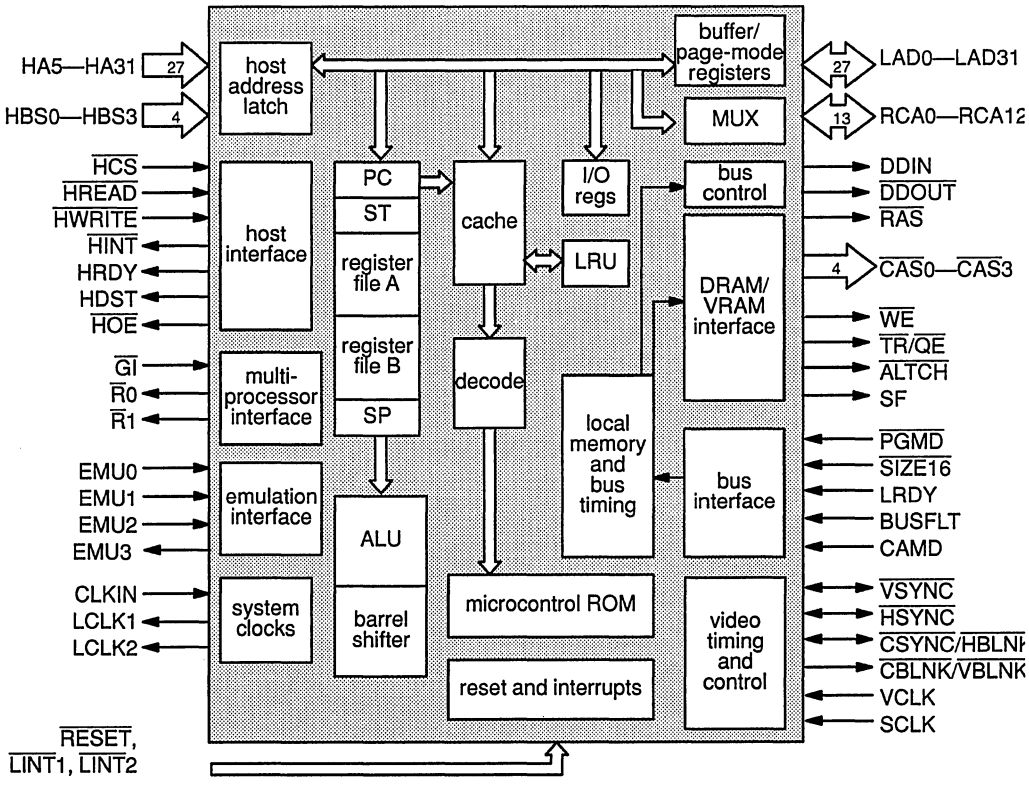
The TMS34020's unique memory interface reduces the time needed to perform tasks such as bit alignment and masking while supporting advanced DRAM access modes. The 32-bit architecture supplies the large blocks of continuously addressable memory that are necessary in graphics applications. Systems designed with the TMS34020 can take advantage of VRAM technology to facilitate applications such as high-bandwidth frame buffers; this circumvents the bottleneck often encountered when using conventional DRAMs in graphics systems.

The TMS34020 instruction set includes a full complement of general- purpose instructions, as well as graphics functions, that you can use to construct efficient high-level functions. The instructions support arithmetic and Boolean operations, data moves, conditional jumps, and subroutine calls and returns. The TMS34020 instruction set also supports the TMS34082 as a coprocessor.

The TMS34020 architecture supports a variety of pixel sizes, frame buffer sizes, and screen sizes. On-chip functions have been carefully selected so that no functions tie the TMS34020 to a particular display resolution. This enhances the portability of graphics software and allows the TMS34020 to adapt to graphics standards such as TIGA, MIT's X, CGI/CGM, GKS, NAPLPS, PHIGS, and evolving industry and display-management standards.

Figure 1–1 illustrates the TMS34020's internal architecture.

Figure 1–1. TMS34020 Block Diagram



1.3.1 Internal Functions

The center portion of Figure 1–1 highlights the main internal functions of the TMS34020 CPU.

- The 32-bit status register (ST) contains several bits that indicate the CPU status. Section 4.1 (page 4-2) discusses the status register.
- The 32-bit program counter (PC) points to the next instruction word to be fetched. The PC's four LSBs are always 0. Section 4.2 (page 4-4) discusses the program counter.
- Register files A and B each contain fifteen 32-bit general-purpose registers. The B-file registers are also used as implied operands for the graphics instructions. Section 4.4 (page 4-6) discusses the register files.
- The 32-bit stack pointer (SP) contains the bit address of the top of the system stack. The SP is also available to instructions that operate on either register file. For more information, refer to Section 4.3 (page 4-5).

- The 32-bit barrel shifter shifts or rotates 32-bit operands from 1 to 32 bit positions in a single machine cycle. This user's guide does not discuss barrel-shifter operation because the operation is transparent.
- The 32-bit ALU allows the TMS34020 to perform most register-to-register operations in a single machine state. (Accessing external memory requires a minimum of two states.) The following actions can occur in parallel during a single machine state:
  - Two operands are transferred from the selected general-purpose register file to the ALU.
  - The ALU performs the specified operation on the operands.
  - The result is routed back to the general-purpose register file.

Instruction cache

The TMS34020 contains a 512-byte instruction cache that can contain up to 256 instruction words (an instruction word may be an entire single-word instruction or 16 bits of a multiple-word instruction). When the cache is enabled, the TMS34020 provides single-cycle execution of general-purpose instructions and of most integer arithmetic and Boolean operations.

Chapter 5 discusses cache operation.

I/O registers

Fifty-four 16-bit, on-chip registers are dedicated to peripheral control functions. The I/O registers are divided into five categories:

- *Local-memory registers* are dedicated to controlling functions such as big-endian/little-endian addressing, refresh rate, row/column mode, plane masking, refresh address, and recovery from bus faults.
- *Video timing and screen-refresh registers* generate the sync and blanking signals used to drive a CRT, schedule screen refreshes, and allow external synchronization.
- *Host-interface registers* help the TMS34020 to communicate with a host processor.
- *Interrupt-control registers* provide status information about interrupt requests.
- *CPU-control registers* configure the TMS34020 to operate with specific characteristics.

Section 4.6 (page 4-14) provides individual descriptions of each I/O register.

Microcontrol ROM

The TMS34020 transfers decoded instructions to the microcontrol ROM for interpretation.

Clock timing logic

The clock timing logic converts the clock-input signal (CLKIN) to internal timing signals and generates the clock-output signals, LCLK1 and LCLK2, used by external devices. The machine state is a fundamental time unit of the TMS34020's graphics processor; it is the time interval during which the processor is in a particular microinstruction state. The instruction timing for each assembly-language instruction is specified in multiples of machine states. The TMS34020's machine state is a single local clock period in duration (the time from one LCLK1 low-to-high transition to the next). The local clock period is four times the period of CLKIN.

Host control logic

The host control logic allows a host processor to communicate with the TMS34020 and allows access to TMS34020 local memory. Commands, data, and status information are communicated through this logic.

Page-mode registers

The page-mode registers buffer data to and from the local-memory interface so that data may be temporarily stored during processing. This enhances data flow to memory.

Other special processing hardware

The TMS34020 CPU also supports the following special processing functions in hardware:

- Detecting whether a pixel lies within a specified display window
- Detecting the leftmost or rightmost 1 within a 32-bit register
- Expanding a black-and-white pattern to a variable pixel-depth pattern
- Rotating and merging variable-width fields
- Individual byte strobes for partial word writes to memory
- Dynamic bus sizing
- Data bus swizzling for special VRAM block modes
- Big-endian and little-endian addressing modes

1.3.2 Major Interfaces

Local-memory and DRAM/VRAM interfaces

The TMS34020's local-memory interface consists of a 32-bit, bidirectional address/data bus, various control signals, and row/column address control. During a local-memory cycle, address and status information are output on the local address/data (LAD) bus; then, data is transferred over the same LAD lines. The TMS34020 can transfer data over 16-bit or 32-bit buses.

The TMS34020 interfaces directly to DRAMs and VRAMs, providing address multiplexing for 64K×*n*, 256K×*n*, 1M×*n*, and 4M×*n* devices. The row and column addresses necessary for accessing DRAMs and VRAMs are available directly from the TMS34020's RCA bus, eliminating the need for external multiplexing hardware.

For more information, refer to Chapter 8.

Video interface

The TMS34020's video interface is extremely flexible and programmable, allowing you to choose between

- Separate sync and blanking or composite sync and blanking
- Synchronization to externally or internally generated video signals
- Interlaced or noninterlaced video

The video interface directly supports VRAMs by generating the serial-register transfers necessary for refreshing a display.

For more information, refer to Chapter 9.

Host interface

The host interface allows you to map the TMS34020's local memory into a host's memory address space. This allows you to transfer data, commands, and status information between the TMS34020 and the host processor.

For more information, refer to Chapter 7.

Coprocessor interface

The coprocessor interface allows you to extend the TMS34020's basic architecture. Most coprocessor interfaces require a memory-mapped approach, so that a processor treats a coprocessor as a peripheral device. The TMS34020, however, allows direct connection to a coprocessor and provides special

instructions that allow you to send instructions and data between the TMS34020 and a coprocessor. The TMS34020 provides extended coprocessor support for the TMS34082 Floating-Point Processor, which is specially designed to serve in a TMS34020 system.

For more information, refer to Chapter 10.

Multiprocessor interface

The multiprocessor interface allows multiple TMS34020s (as well as other processors) to share the same local memory. The TMS34020's grant-in and request-priority signals provide a flexible method of passing control from one processor to another. The multiprocessor interface requires external arbitration logic to

- inform a TMS34020 when it can take control of the bus, and
- decode the priorities of requests from the multiple processors.

This scheme allows back-to-back memory cycles even when control passes from one TMS34020 to another.

Any number of devices can be configured together within a single system. However, system performance is not increased significantly when a system contains more than three TMS34020s.

For more information, refer to Chapter 11.

Emulation interface

The TMS34020 supports a 4-wire interface that simplifies connections between a debugger and a target system. For details about emulation, refer to Appendix A.

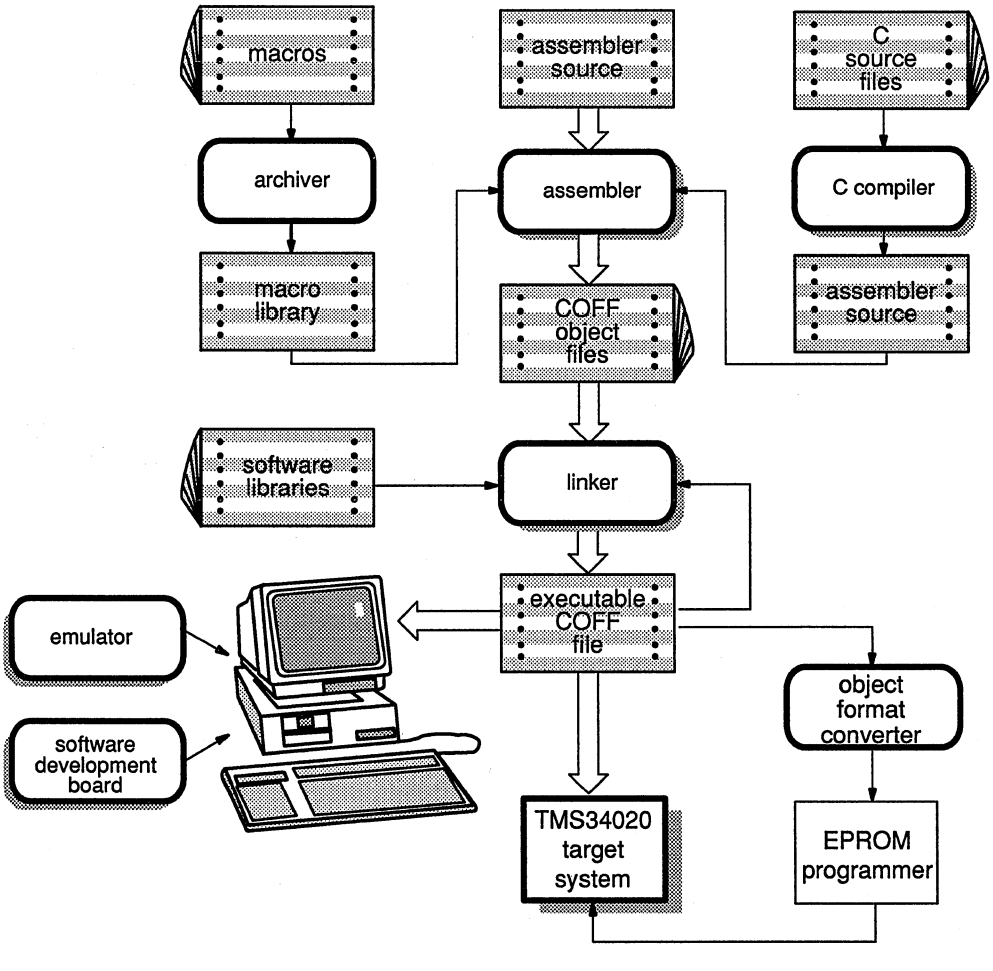
1.4 System Development Tools

The TMS34020 is well supported by a complete set of hardware and software development tools, including a C compiler, an assembler/linker, software libraries, and a PC-based development board. In addition, the TMS34020 is fully compatible with and supported by the Texas Instruments Graphics Architecture (TIGA-340).

1.4.1 Code-Generation Tools

Figure 1–2 illustrates the TMS34020 code development flow. The figure highlights the most common paths of software development; the other portions are optional.

Figure 1–2. TMS34020 Software Development Flow



These tools use common object files format (COFF), which encourages modular programming. COFF allows you to divide your code into logical blocks, define your system's memory map, and then link code into specific memory areas. COFF also provides rich support for source-level debugging.

The following list describes the tools shown in Figure 1–2.

C compiler

The TMS34020 **C compiler** is a full-featured optimizing compiler that translates standard Kernighan-and-Ritchie C programs into TMS34020 assembly-language source. Key characteristics include

- *Standard Kernighan-and-Ritchie C with extensions.* The compiler compiles standard C programs as defined by Kernighan and Ritchie's ***The C Programming Language*** (first edition). The compiler supports these standard extensions: enumeration types, structure assignments, passing structures to functions, and returning structures from functions. A future release of the compiler will support the full ANSI standard.
- *Big-endian or little-endian code.*
- *Optimization.* The compiler uses several advanced techniques for generating efficient, compact code from C source.
- *Assembly-language output.* The compiler generates assembly- language source that is easily inspected, enabling you to see the code generated from the C source files.
- *ANSI standard runtime support.* The compiler package comes with a complete runtime library that conforms to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, exponential, and hyperbolic functions. Functions for I/O and signal handling are not included because they are application-specific.
- *Flexible assembly-language interface.* The compiler has straight-forward calling conventions, allowing you to easily write assembly and C functions that call each other.
- *Shell program.* The compiler package includes a shell program that enables you to compile, assemble, and link programs in a single step.
- *Source interlist utility.* The compiler package includes a utility that interlists your original C source statements into the assembly-language output of the compiler. This utility provides you with an easy method for inspecting the assembly code generated for each C statement.

assembler

The **assembler** translates assembly-language source files into machine language object files.

archiver

The **archiver** allows you to collect a group of files into a library. It also allows you to modify a library by deleting, replacing, extracting, or adding members.

One of the most useful applications of the archiver is to build a library of object modules. Several object libraries and a source library are included with the C compiler.

You can also use application-specific object libraries, available as separate products:

- The **math/graphics function library** contains math functions for performing algebraic, trigonometric, and transcendental operations as well as graphics functions for performing viewport management, bitmapped text, graphics output, color-palette control, 3-dimensional transformations, and graphics initialization.
- The **font library** contains a variety of proportionally spaced and monospaced fonts. You can use the functions in the graphics library to display the fonts.
- The **CCITT data compression function library** contains CCITT-compatible routines for compressing and decompressing monochrome image data.
- The **8514 adaptor emulation function library** contains routines for emulating IBM PS/2 high-resolution display.

These functions and routines can be called from C programs. You can also create your own object libraries.

linker

The **linker** combines object files into a single, executable object module. As the linker creates the executable module, it performs relocation and resolves external references. The linker is a tool that allows you to define your system's memory map and associate blocks of code with defined memory areas.

debugging tools

The main purpose of the development process is to produce a module that can be executed in a **TMS34020 target system**. You can use one of several debugging tools to refine and correct your code. Available products include a PC-based **software development board** (SDB) and a realtime in-circuit **emulator**.

object format converter

An **object format converter** is also available; it converts a COFF object file into an Intel, Tektronix, or TI-tagged object-format file that can be downloaded to an EPROM programmer.

1.4.2 Supported Systems

The TMS34020 C compiler and assembly language tools are available for these systems:

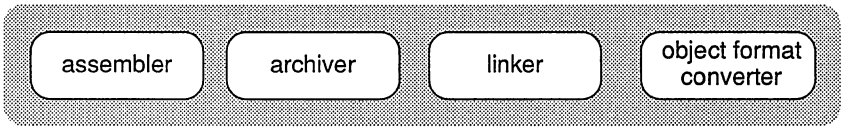
- IBM-PC with PC-DOS
- VAX:
  - VMS
  - Ultrix

- Apollo workstations:
  - Domain/IX
  - AEGIS
- Sun-3 workstations with UNIX
- Macintosh with MPW

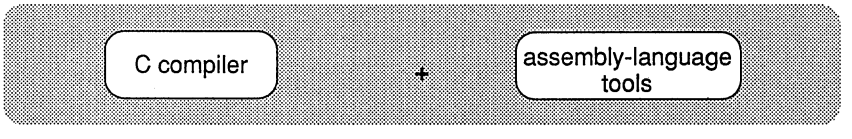
1.4.3 Packages

Texas Instruments supplies development tools in several packages.

- **Assembly language tools package**



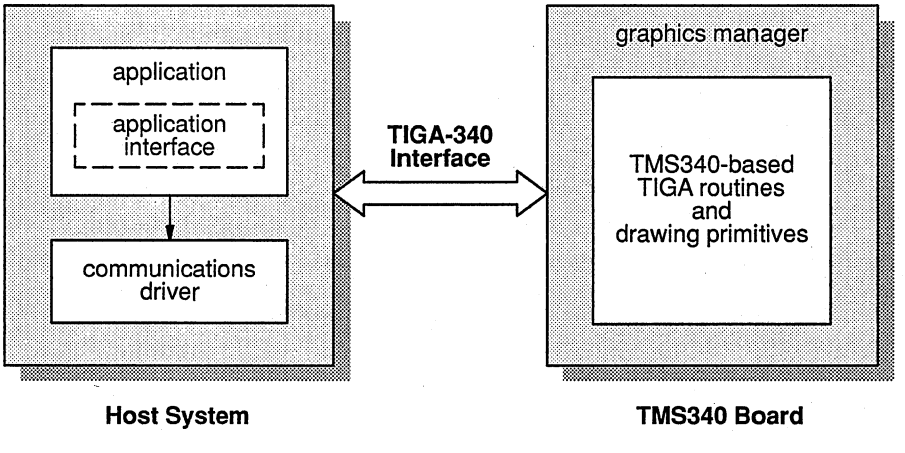
- **C compiler package**



1.4.4 TIGA-340 Graphics Interface

The Texas Instruments Graphics Architecture (TIGA-340) is a software interface standard for the TMS340 family of graphics system processors. TIGA enhances the performance of MS-DOS-based PCs that contain a TMS34010 or TMS34020 and an 8088/86 or 80286/80386 host microprocessor by optimizing communications between the graphics processor and the host processor. The TIGA interface allows the host and graphics processors to share execution of the application.

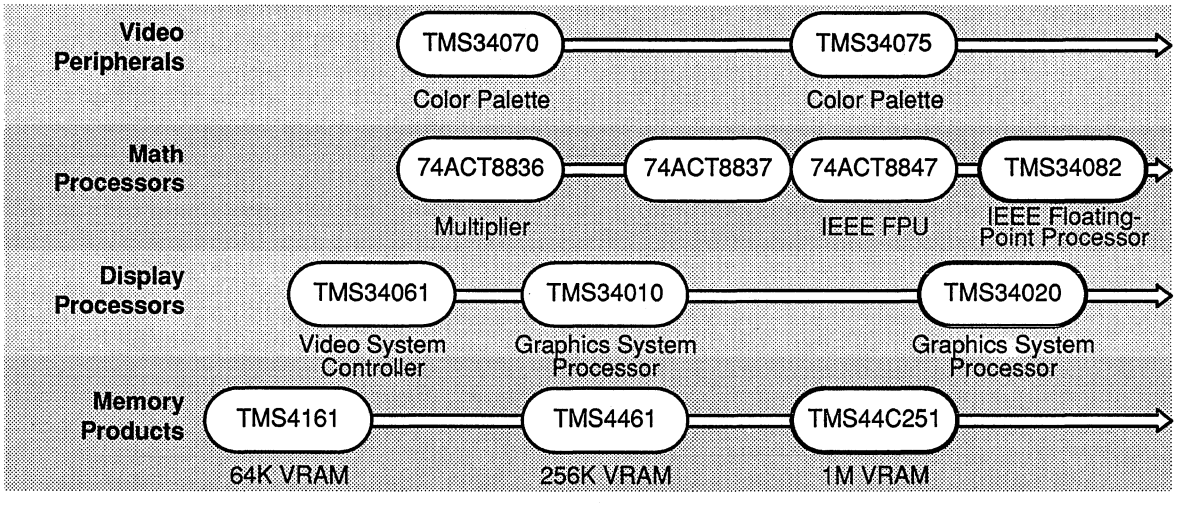
Figure 1–3. Graphics Processing Shared Between TMS340 and Host Processors



1.5 Processors for a Graphics System

Texas Instruments offers a broad line of graphics and video products. The TMS34020 Graphics System Processor, TMS34082 Floating-Point Processor, and TMS44C251 1-Mbit Video RAM bring workstation performance to the PC and other small systems. Figure 1–4 shows all of the Texas Instruments Graphics products.

Figure 1–4. Graphics Products Roadmap



The following paragraphs describe the TMS34082 and TMS44C251, which are included in the TMS34020 sample system shown in Figure 1–5.

TMS34082

Many TMS34020 applications require floating-point operations. The TMS34082 floating-point processor is designed to interface directly with the TMS34020, allowing the TMS34020 to perform computation-intensive functions more than 100 times faster than a software implementation. The TMS34082 performs single- and double-precision floating-point operations, conforming fully to the IEEE 754 standard.

In addition to normal floating-point operations, the TMS34082 performs complex 2- and 3-dimensional operations such as 3×3 convolution, 4×4 matrix, and cubic spline operations.

Additional TMS34082 features include

- 32-bit data path
- 32-bit integer and logical operations
- 40-MFLOPS sustained operation
- Single-instruction divide/square-root operations
- External microcode memory interface for defining custom instructions

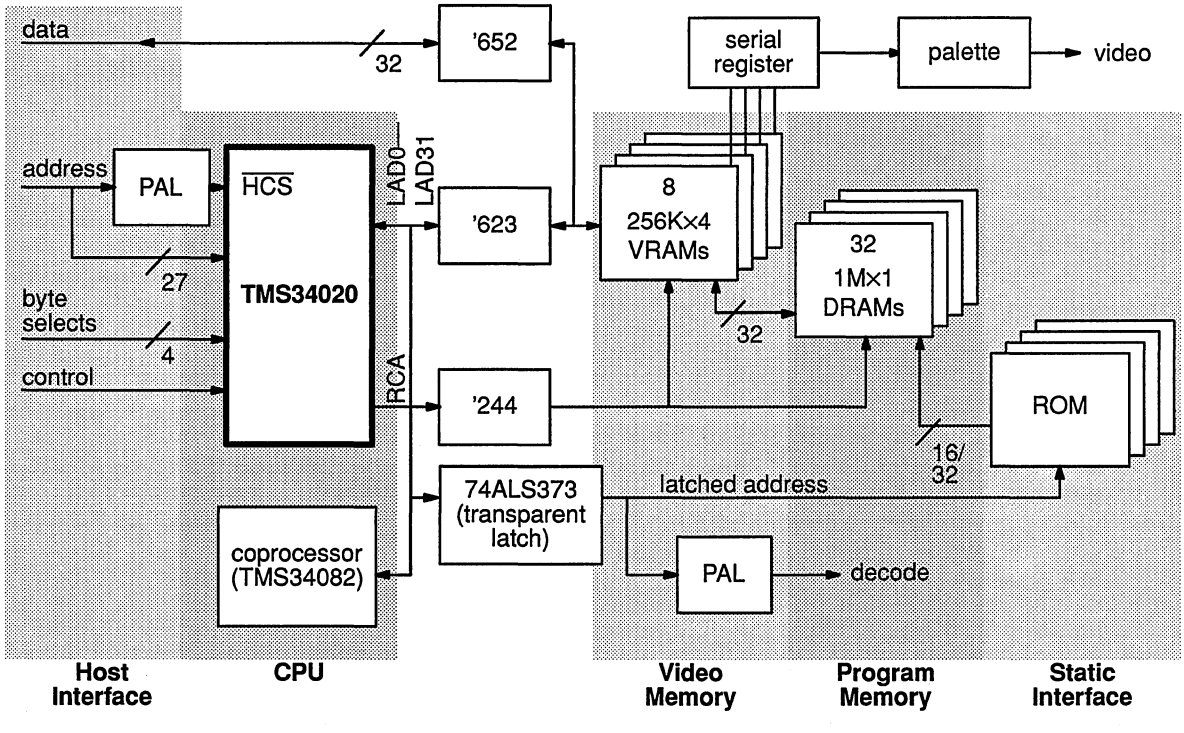
TMS44C251

A video RAM (VRAM) is a special memory device, optimized for use in graphics systems. The TMS44C251 multiport VRAM is a high-speed, dual-ported memory. It consists of DRAM organized as 262,144 4-bit words, interfaced to a serial data register.

Sample system

A typical graphics system designed with the TMS34020 uses several types of memory, as well as external latches, buffers, and transceivers to connect the TMS34020 to the memories, a coprocessor, or a host processor. Figure 1–5 shows a representative TMS34020 design for a PC display system. Note that this system uses the TMS34082 as a coprocessor, a palette, and VRAMs, DRAMs, and ROM memories.

Figure 1–5. TMS34020 1K×1K×8 PC Display System



Key:

'652	Bidirectional, latching transceivers (such as the 74ALS652)	*Required*
'623	Bidirectional bus transceivers (such as the 74ALS623)	*Optional*
'244	Buffer (such as the 74ALS244)	*Optional*

1.6 Compatibility Between the TMS34020 and TMS34010

> The information in this section is for readers who are familiar with the TMS34010 graphics system processor. If you are not familiar with the TMS34010, you may want to skip this section. Note that this user's guide *does not* require you to be familiar with the TMS34010.

The TMS34020 is the second generation of the TMS340 family of graphics system processors; the TMS34010 is the first generation. The TMS34010 was the building block for the TMS34020; however, the TMS34020 greatly extends the TMS34010's capabilities by adding new features and enhancing existing features. Table 1–1 shows a sample comparison of the TMS34010 and TMS34020 features.

Note that the TMS34010 and TMS34020 **are not** pin-for-pin compatible.

Table 1–1. Quick Comparison of TMS34010 and TMS34020 Features

Feature	TMS34010	TMS34020
External bus size	❑ 16 bits	❑ 32 bits
Cycle time	❑ 130, 160, or 200 ns	❑ 100 or 125 ns
Cache size	❑ 256 bytes	❑ 512 bytes
Horizontal pitch	❑ Power of 2	❑ Unlimited
Word addressing	❑ Little endian	❑ Little or big endian
VRAM support	❑ Serial registers	❑ Serial registers ❑ Block writes ❑ Split serial registers ❑ Enhanced page mode
Interfaces	❑ Host ❑ Hold	❑ Host ❑ Coprocessor ❑ Multiprocessor
Coprocessor support	❑ Memory mapped	❑ Direct connection

Throughout this user's guide you'll find descriptions of compatibility between the TMS34010 and TMS34020. Such passages are marked with this symbol in the margin:

TMS34010 object code is upward compatible with the TMS34020. If new TMS34020 features would prevent TMS34010 code from running, the TMS34020 provides you with a method of switching these features off. At reset, these features are off to provide compatibility with the TMS34010.

In general, if you followed the compatibility notes in the *TMS34010 User's Guide*, your TMS34010 code should be object-code compatible with the TMS34020.

The following list describes restrictions that TMS34010 code must adhere to in order to be compatible with the TMS34020.

- **Color information.** The TMS34020 uses all 32 bits of the COLOR0 (B8) and COLOR1 (B9) values. The TMS34010 used only the 16 LSBs of these values. Although the TMS34010 will ignore the 16 MSBs of these values, TMS34010 code should replicate the color information throughout all 32 bits of these registers.

- **Plane mask.** All 32 bits of the PMASK register, at addresses C000 0160h (16 LSBs) and C000 0170h (16 MSBs), are valid for the TMS34020. TMS34010 could should copy the 16-bit PMASK value at address C000 0160h to address C000 0170h.

- **Reserved bits.** TMS34010 code should not use any reserved bits in the status register or the I/O registers.

- **Register B13.** The TMS34020 uses register B13 as a pattern register. TMS34010 code should load B13 with all 1s, causing the code to draw a solid line instead of an unexpected patterned line.

- **CONVSP & SPTCH, CONVDP & DPTCH.** The TMS34020 uses SPTCH and DPTCH to determine the values of CONVSP and CONVDP, respectively. TMS34010 code should be sure that SPTCH and DPTCH agree with CONVSP and CONVDP. That is, the 5 LSBs of CONVxP must equal the 1s complement of $\log_2$(xPTCH), which is given by the LMO of xPTCH. Set the 11 MSBs of CONVxP to 0.

  If an instruction uses CONVSP or CONVDP, then the MSB of CONVxP should be 0 and xPTCH should contain 2^{CONVxP} before instruction execution.

- **Timing loops.** TMS34010 code should avoid timing loops; obtain timing via the video logic (using DPYINT) or via external interrupt 1 or 2.

- **Data alignment.** For optimum TMS34020 performance, TMS34010 code should align to 32-bit boundaries (instead of 16-bit boundaries).

- **Cache.** TMS34010 code should not depend on cache-load order.

- **Saving the graphics context.** If TMS34010 code requires saving/restoring of the graphics context, the code should store the I/O registers at addresses C000 00B0h and C000 0130—C000 01A0h (inclusive).

- **Reset vector.** At reset, the TMS34020 loads the 4 LSBs of the reset vector into the 4 LSBs of the CONFIG register. TMS34010 code should not depend on values in the 4 LSBs of the reset vector.

- **Video registers.** All accesses to video timing registers should be separate from other code. Particularly, HESYNC, HEBLNK, HSBLNK, HTOTAL, HCOUNT, VESYNC, VEBLNK, VSBLNK, VTOTAL, and VCOUNT should be manipulated through symbolic names (not by addresses) because their addresses have changed.

- **Interrupt routines.** Interrupt service routines for the TMS34010 should make no assumptions about the state of the stack, except that the PC and ST are stacked after any extra words. The interrupt routine must return with an RETI instruction, which will pop any extra words to the correct internal registers.

- **Illegal opcodes.** TMS34010 code should not depend on any of the TMS34010's illegal opcodes (except 0000h) to cause a TRAP 30.

- **Traps.** Trap FFFF FBC0h is the TMS34020's bus-fault trap. Trap FFFF FBE0h is the TMS34020's single-step trap.

- **Host interface.** To TMS34010 code, the TMS34020's host interface appears the same as the TMS34010's host interface. It is desirable for data shared with the host to be aligned on 32-bit boundaries. In general, code written for the TMS34010 host interface will need to be changed because the TMS34020's host interface is different from the TMS34010's.

Chapter 2

Pinouts and Signal Descriptions

This chapter illustrates the TMS34020 pinouts and provides detailed descriptions of the TMS34020's signals. For mechanical dimensions of TMS34020 packages, refer to the *TMS34020 Data Sheet.*

2.1 Pinouts

The TMS34020 is offered in two packages:

- a 145-pin grid array (PGA) package and
- a 132-pin quad flat package (QFP).

Figure 2–1 shows the pinout of the 145-pin PGA, and Figure 2–2 shows the pinout for the 132-pin QFP.

Figure 2–1. TMS34020 Pinout, 145-Pin PGA Package (Bottom View)

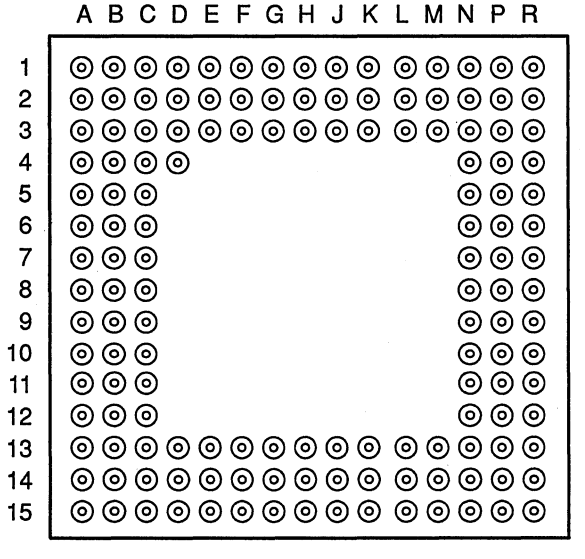


Table 2–1. Numerical List of TMS34020 Pin Assignments (145-Pin PGA)

Pin #	Signal	Pin #	Signal	Pin #	Signal	Pin #	Signal	Pin #	Signal
A1	V_{SS}	B15	LAD12	F1	HRDY	K15	LAD20	P2	$\overline{\text{HWRITE}}$
A2	$\overline{\text{ALTCH}}$	C1	$\overline{\text{CAS0}}$	F2	$\overline{\text{R0}}$	L1	$\overline{\text{LINT1}}$	P3	$\overline{\text{HCS}}$
A3	$\overline{\text{CBLNK}}$/ $\overline{\text{VBLNK}}$	C2	V_{CC}	F3	V_{SS}	L2	CAMD	P4	HA30
A4	$\overline{\text{HSYNC}}$	C3	$\overline{\text{DDOUT}}$	F13	LAD24	L3	LRDY	P5	HA27
A5	$\overline{\text{TR/QE}}$	C4	DDIN	F14	LAD8	L13	LAD1	P6	HA24
A6	RCA2	C5	V_{SS}	F15	V_{SS}	L14	LAD2	P7	HA22
A7	RCA3	C6	SF	G1	$\overline{\text{HINT}}$	L15	LAD19	P8	HA18
A8	V_{CC}	C7	RCA4	G2	$\overline{\text{HOE}}$	M1	BUSFLT	P9	HA14
A9	RCA6	C8	V_{SS}	G3	HDST	M2	$\overline{\text{PGMD}}$	P10	HA13
A10	RCA7	C9	RCA8	G13	LAD7	M3	VCLK	P11	HA10
A11	RCA10	C10	RCA12	G14	V_{SS}	M13	V_{SS}	P12	HA7
A12	SCLK	C11	LAD30	G15	LAD23	M14	LAD16	P13	HA5
A13	LAD15	C12	V_{SS}	H1	LCLK1	M15	LAD18	P14	HBS0
A14	LAD29	C13	V_{SS}	H2	EMU3	N1	$\overline{\text{SIZE16}}$	P15	LAD0
A15	V_{SS}	C14	V_{CC}	H3	LCLK2	N2	V_{CC}	R1	$\overline{\text{HREAD}}$
B1	$\overline{\text{CAS3}}$	C15	LAD26	H13	LAD22	N3	CLKIN	R2	HA31
B2	$\overline{\text{WE}}$	D1	$\overline{\text{RAS}}$	H14	LAD21	N4	V_{SS}	R3	HA28
B3	V_{SS}	D2	$\overline{\text{CAS2}}$	H15	LAD6	N5	HA29	R4	HA26
B4	$\overline{\text{CSYNC}}$/ $\overline{\text{HBLNK}}$	D3	V_{SS}	J1	EMU0	N6	HA25	R5	HA23
B5	$\overline{\text{VSYNC}}$	D4	NC	J2	$\overline{\text{GI}}$	N7	HA21	R6	HA20
B6	RCA0	D13	LAD28	J3	EMU1	N8	V_{SS}	R7	HA19
B7	RCA1	D14	LAD11	J13	LAD4	N9	V_{SS}	R8	HA17
B8	RCA5	D15	LAD10	J14	V_{CC}	N10	HA12	R9	HA16
B9	RCA9	E1	$\overline{\text{R1}}$	J15	LAD5	N11	HA6	R10	HA15
B10	RCA11	E2	V_{CC}	K1	EMU2	N12	HBS2	R11	HA11
B11	LAD31	E3	$\overline{\text{CAS1}}$	K2	$\overline{\text{RESET}}$	N13	HBS1	R12	HA9
B12	LAD14	E13	LAD27	K3	$\overline{\text{LINT2}}$	N14	V_{CC}	R13	HA8
B13	V_{CC}	E14	LAD25	K13	V_{SS}	N15	LAD17	R14	HBS3
B14	LAD13	E15	LAD9	K14	LAD3	P1	V_{CC}	R15	V_{SS}

Note: Pin D4 is NC (not internally connected). You may use this pin for package alignment, but do not connect it.

Table 2–2. Alphabetical List of TMS34020 Pin Assignments (145-Pin PGA)

Signal	Pin #	Signal	Pin #	Signal	Pin #	Signal	Pin #	Signal	Pin #
$\overline{\text{ALTCH}}$	A2	HA17	R8	LAD2	L14	LAD31	B11	V_{CC}	A8
BUSFLT	M1	HA18	P8	LAD3	K14	LCLK1	H1	V_{CC}	B13
CAMD	L2	HA19	R7	LAD4	J13	LCLK2	H3	V_{CC}	C2
$\overline{\text{CAS0}}$	C1	HA20	R6	LAD5	J15	$\overline{\text{LINT1}}$	L1	V_{CC}	C14
$\overline{\text{CAS1}}$	E3	HA21	N7	LAD6	H15	$\overline{\text{LINT2}}$	K3	V_{CC}	E2
$\overline{\text{CAS2}}$	D2	HA22	P7	LAD7	G13	LRDY	L3	V_{CC}	J14
$\overline{\text{CAS3}}$	B1	HA23	R5	LAD8	F14	NC	D4	V_{CC}	N2
$\overline{\text{CBLNK}}$/ $\overline{\text{VBLNK}}$	A3	HA24	P6	LAD9	E15	$\overline{\text{PGMD}}$	M2	V_{CC}	N14
CLKIN	N3	HA25	N6	LAD10	D15	$\overline{\text{R0}}$	F2	V_{CC}	P1
$\overline{\text{CSYNC}}$/ $\overline{\text{HBLNK}}$	B4	HA26	R4	LAD11	D14	$\overline{\text{R1}}$	E1	VCLK	M3
DDIN	C4	HA27	P5	LAD12	B15	$\overline{\text{RAS}}$	D1	V_{SS}	A1
$\overline{\text{DDOUT}}$	C3	HA28	R3	LAD13	B14	RCA0	B6	V_{SS}	A15
EMU0	J1	HA29	N5	LAD14	B12	RCA1	B7	V_{SS}	B3
EMU1	J3	HA30	P4	LAD15	A13	RCA2	A6	V_{SS}	C5
EMU2	K1	HA31	R2	LAD16	M14	RCA3	A7	V_{SS}	C8
EMU3	H2	HBS0	P14	LAD17	N15	RCA4	C7	V_{SS}	C12
$\overline{\text{GI}}$	J2	HBS1	N13	LAD18	M15	RCA5	B8	V_{SS}	C13
HA5	P13	HBS2	N12	LAD19	L15	RCA6	A9	V_{SS}	D3
HA6	N11	HBS3	R14	LAD20	K15	RCA7	A10	V_{SS}	F3
HA7	P12	$\overline{\text{HCS}}$	P3	LAD21	H14	RCA8	C9	V_{SS}	F15
HA8	R13	HDST	G3	LAD22	H13	RCA9	B9	V_{SS}	G14
HA9	R12	$\overline{\text{HINT}}$	G1	LAD23	G15	RCA10	A11	V_{SS}	K13
HA10	P11	$\overline{\text{HOE}}$	G2	LAD24	F13	RCA11	B10	V_{SS}	M13
HA11	R11	HRDY	F1	LAD25	E14	RCA12	C10	V_{SS}	N4
HA12	N10	$\overline{\text{HREAD}}$	R1	LAD26	C15	$\overline{\text{RESET}}$	K2	V_{SS}	N8
HA13	P10	$\overline{\text{HSYNC}}$	A4	LAD27	E13	SCLK	A12	V_{SS}	N9
HA14	P9	$\overline{\text{HWRITE}}$	P2	LAD28	D13	SF	C6	V_{SS}	R15
HA15	R10	LAD0	P15	LAD29	A14	$\overline{\text{SIZE16}}$	N1	$\overline{\text{VSYNC}}$	B5
HA16	R9	LAD1	L13	LAD30	C11	$\overline{\text{TR/QE}}$	A5	$\overline{\text{WE}}$	B2

Note: Pin D4 is NC (not internally connected). You may use this pin for package alignment, but do not connect it.

Figure 2–2. TMS34020 Pinout, 132-Pin QFP Package

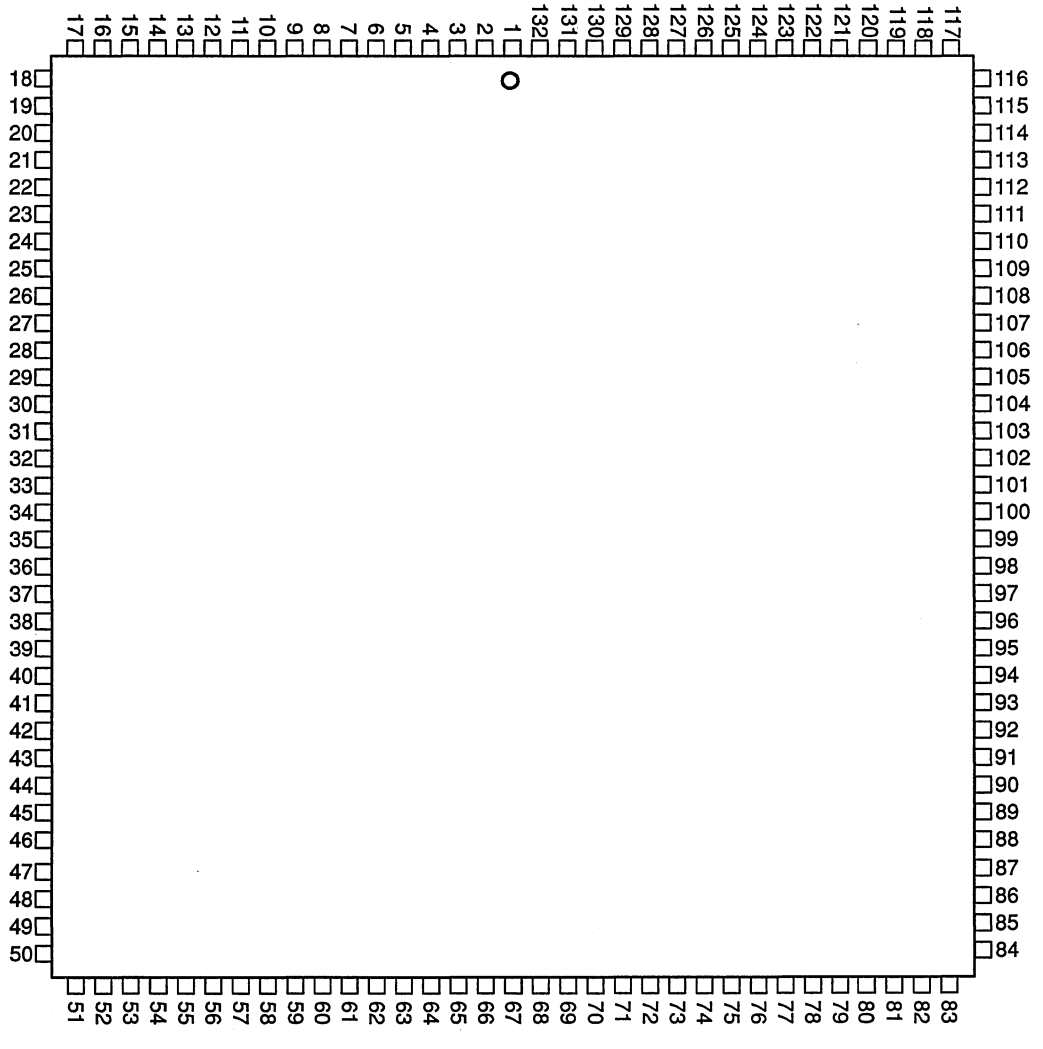


Table 2–3. Numerical List of TMS34020 Pin Assignments (132-Pin QFP)

Pin #	Signal	Pin #	Signal	Pin #	Signal	Pin #	Signal
1	EMU1	34	V_{SS}	67	LAD7	100	RCA4
2	EMU0	35	HA16	68	LAD23	101	RCA3
3	EMU2	36	HA15	69	V_{SS}	102	RCA2
4	$\overline{GI}$	37	HA14	70	V_{SS}	103	RCA1
5	$\overline{RESET}$	38	HA13	71	LAD8	104	RCA0
6	$\overline{LINT2}$	39	HA12	72	LAD24	105	SF
7	$\overline{LINT1}$	40	HA11	73	LAD9	106	$\overline{TR/QE}$
8	CAMD	41	HA10	74	LAD25	107	$\overline{VSYNC}$
9	BUSERR	42	HA9	75	LAD10	108	$\overline{HSYNC}$
10	$\overline{SIZE16}$	43	HA8	76	LAD26	109	$\overline{CBLNK/VBLNK}$
11	$\overline{PGMD}$	44	HA7	77	LAD11	110	$\overline{CSYNC/HBLNK}$
12	$\overline{LRDY}$	45	HA6	78	LAD27	111	V_{SS}
13	V_{CC}	46	HA5	79	V_{CC}	112	V_{SS}
14	VCLK	47	HBS3	80	LAD12	113	$\overline{ALTCH}$
15	CLKIN	48	HBS2	81	LAD28	114	DDIN
16	$\overline{HWRITE}$	49	HBS1	82	V_{SS}	115	$\overline{DDOUT}$
17	$\overline{HREAD}$	50	HBS0	83	LAD13	116	$\overline{WE}$
18	$\overline{HCS}$	51	LAD0	84	LAD29	117	$\overline{CAS3}$
19	HA31	52	LAD16	85	LAD14	118	$\overline{CAS2}$
20	HA30	53	LAD1	86	LAD30	119	$\overline{CAS1}$
21	HA29	54	LAD17	87	LAD15	120	$\overline{CAS0}$
22	HA28	55	LAD2	88	LAD31	121	V_{CC}
23	HA27	56	LAD18	89	SCLK	122	$\overline{RAS}$
24	HA26	57	V_{SS}	90	RCA12	123	V_{SS}
25	HA25	58	LAD3	91	RCA11	124	$\overline{R0}$
26	HA24	59	LAD19	92	RCA10	125	$\overline{R1}$
27	HA23	60	V_{CC}	93	RCA9	126	$\overline{HOE}$
28	HA22	61	LAD4	94	RCA8	127	HDST
29	HA21	62	LAD20	95	RCA7	128	HRDY
30	HA20	63	LAD5	96	RCA6	129	$\overline{HINT}$
31	HA19	64	LAD21	97	RCA5	130	EMU3
32	HA18	65	LAD6	98	V_{CC}	131	LCLK1
33	HA17	66	LAD22	99	V_{SS}	132	LCLK2

Table 2–4. Alphabetical List of TMS34020 Pin Assignments (132-Pin QFP)

Signal	Pin #	Signal	Pin #	Signal	Pin #	Signal	Pin #
$\overline{\text{ALTCH}}$	113	HA21	29	LAD10	75	RCA2	102
BUSERR	9	HA22	28	LAD11	77	RCA3	101
CAMD	8	HA23	27	LAD12	80	RCA4	100
$\overline{\text{CAS0}}$	120	HA24	26	LAD13	83	RCA5	97
$\overline{\text{CAS1}}$	119	HA25	25	LAD14	85	RCA6	96
$\overline{\text{CAS2}}$	118	HA26	24	LAD15	87	RCA7	95
$\overline{\text{CAS3}}$	117	HA27	23	LAD16	52	RCA8	94
$\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$	109	HA28	22	LAD17	54	RCA9	93
CLKIN	15	HA29	21	LAD18	56	RCA10	92
$\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$	110	HA30	20	LAD19	59	RCA11	91
DDIN	114	HA31	19	LAD20	62	RCA12	90
$\overline{\text{DDOUT}}$	115	HBS0	50	LAD21	64	$\overline{\text{RESET}}$	5
EMU0	2	HBS1	49	LAD22	66	SCLK	89
EMU1	1	HBS2	48	LAD23	68	SF	105
EMU2	3	HBS3	47	LAD24	72	$\overline{\text{SIZE16}}$	10
EMU3	130	$\overline{\text{HCS}}$	18	LAD25	74	$\overline{\text{TR/QE}}$	106
$\overline{\text{GI}}$	4	HDST	127	LAD26	76	V_{CC}	13
HA5	46	$\overline{\text{HOE}}$	126	LAD27	78	V_{CC}	60
HA6	45	$\overline{\text{HINT}}$	129	LAD28	81	V_{CC}	79
HA7	44	HRDY	128	LAD29	84	V_{CC}	98
HA8	43	$\overline{\text{HREAD}}$	17	LAD30	86	V_{CC}	121
HA9	42	$\overline{\text{HSYNC}}$	108	LAD31	88	VCLK	14
HA10	41	$\overline{\text{HWRITE}}$	16	LCLK1	131	V_{SS}	34
HA11	40	LAD0	51	LCLK2	132	V_{SS}	57
HA12	39	LAD1	53	$\overline{\text{LINT1}}$	7	V_{SS}	69
HA13	38	LAD2	55	$\overline{\text{LINT2}}$	6	V_{SS}	70
HA14	37	LAD3	58	LRDY	12	V_{SS}	82
HA15	36	LAD4	61	$\overline{\text{PGMD}}$	11	V_{SS}	99
HA16	35	LAD5	63	$\overline{\text{R0}}$	124	V_{SS}	111
HA17	33	LAD6	65	$\overline{\text{R1}}$	125	V_{SS}	112
HA18	32	LAD7	67	$\overline{\text{RAS}}$	122	V_{SS}	123
HA19	31	LAD8	71	RCA0	104	$\overline{\text{VSYNC}}$	107
HA20	30	LAD9	73	RCA1	103	$\overline{\text{WE}}$	116

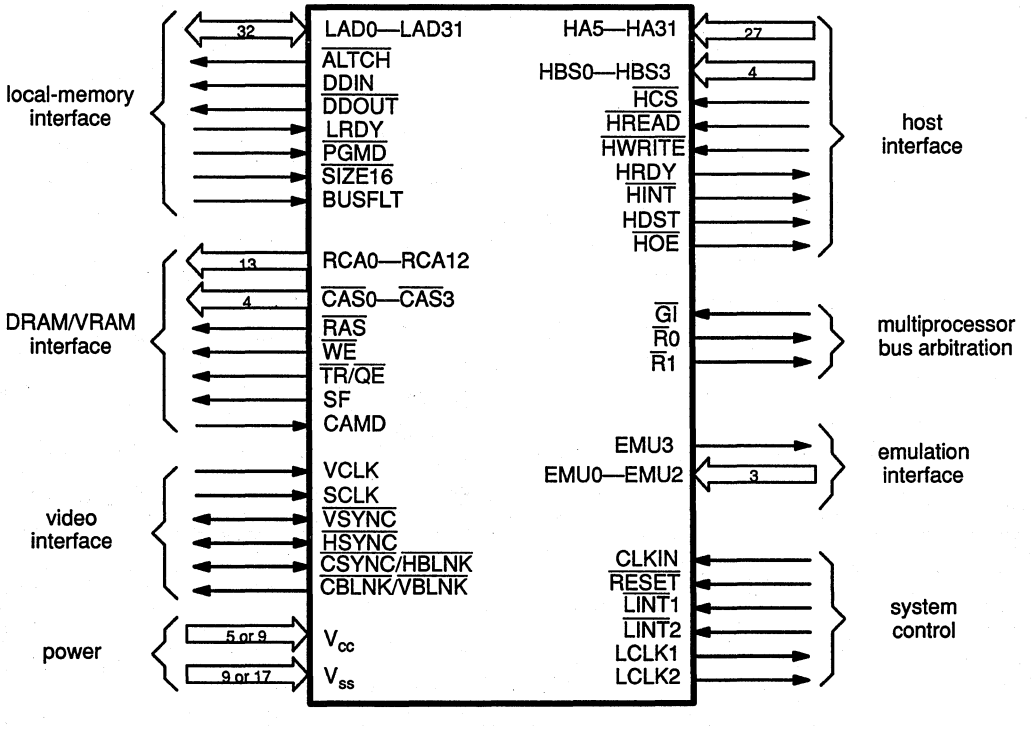
2.2 The TMS34020's Major Interfaces

The TMS34020's pins are divided among several interfaces:

Name	Pins
Local-memory interface	39 pins
DRAM/VRAM control interface	22 pins
Multiprocessor interface	3 pins
Host interface	38 pins
Video interface	6 pins
Emulation interface	4 pins
System control	6 pins
Power and ground	26 pins (PGA) 14 pins (QFP)

Figure 2–3 associates the TMS34020's pins with its major interfaces.

Figure 2–3. The TMS34020's Major Interfaces



2.3 Signal Descriptions

This section describes the TMS34020 signals. Table 2–5 associates the signals with the proper interfaces, provides brief descriptions, and references page numbers on which you can find detailed signal descriptions. Subsections 2.3.1 through 2.3.7 provide details concerning the individual groups.

Table 2–5. TMS34020 Pin Descriptions

	Signal Name	I/O	Description	Refer to Page. . .
Local-Memory Interface	$\overline{\text{ALTCH}}$	O	Address latch	2-11
	BUSFLT	I	Bus fault	2-11
	DDIN	O	Data bus direction input enable	2-11
	$\overline{\text{DDOUT}}$	O	Data bus direction output enable	2-11
	LAD0—LAD31	I/O	Local address/data multiplexed signals	2-11
	LRDY	I	Local ready	2-11
	$\overline{\text{PGMD}}$	I	Page mode	2-11
	$\overline{\text{SIZE16}}$	I	Bus size	2-11
DRAM/VRAM Control	CAMD	I	Column-address mode	2-12
	$\overline{\text{CAS0}}$—$\overline{\text{CAS3}}$	O	Column-address strobes	2-12
	$\overline{\text{RAS}}$	O	Row-address strobe	2-12
	RCA0—RCA12	O	Thirteen multiplexed row-/column-address signals	2-12
	SF	O	Special function pin	2-12
	$\overline{\text{TR/QE}}$	O	Transfer/output enable	2-12
	$\overline{\text{WE}}$	O	Write enable	2-12
Multiprocessor Interface	$\overline{\text{GI}}$	I	Bus-grant input	2-13
	$\overline{\text{R0}}$, $\overline{\text{R1}}$	O	Bus-request code	2-13

Table 2–5. TMS34020 Pin Descriptions (Continued)

	Signal Name	I/O	Description	Refer to Page. . .
Host Interface	HA5—HA31	I	Host-address input signals	2-13
	HBS0—HBS3	I	Host byte selects	2-13
	$\overline{\text{HCS}}$	I	Host chip select	2-13
	HDST	O	Host data-latch strobe	2-13
	$\overline{\text{HINT}}$	O	Host interrupt	2-13
	$\overline{\text{HOE}}$	O	Host data-latch output enable	2-13
	HRDY	O	Host ready	2-13
	$\overline{\text{HREAD}}$	I	Host read strobe	2-13
	$\overline{\text{HWRITE}}$	I	Host write strobe	2-13
Video Interface	$\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$	O	Composite blanking or vertical blanking	2-15
	$\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$	I/O	Composite sync or horizontal blanking	2-15
	$\overline{\text{HSYNC}}$	I/O	Horizontal sync	2-15
	SCLK	I	Serial data clock	2-15
	VCLK	I	Video clock	2-15
	$\overline{\text{VSYNC}}$	I/O	Vertical sync	2-15
Emulation Interface	EM0—EMU2	I	Emulation pins 0—2	Appendix A
	EMU3	O	Emulation pin 3	Appendix A
System Control	CLKIN	I	Clock input	2-16
	LCLK1, LCLK2	O	Local output clocks	2-16
	$\overline{\text{LINT1}}$, $\overline{\text{LINT2}}$	I	Local interrupt requests	2-16
	$\overline{\text{RESET}}$	I	System reset	2-16
Power Control	V_{CC}	I	Nominal 5-volt power supply inputs (5 pins on QFP, 9 pins on PGA)	2-16
	V_{SS}	I	Electrical ground inputs (9 pins pins on QFP, 17 pins on PGA)	2-16

2.3.1 Local-Memory Interface Signals

The TMS34020 communicates with external memory and with external memory-mapped I/O devices through its local-memory interface. This interface's signals are also used in conjunction with the DRAM and VRAM interface.

Signal Name	I/O	Description
$\overline{\text{ALTCH}}$	O	**Address latch.** The high-to-low transition of $\overline{\text{ALTCH}}$ can be used to capture the address and status present on the LAD bus. A transparent latch (such as a 74ALS373) will maintain the current address and status as long as $\overline{\text{ALTCH}}$ remains low.
BUSFLT	I	**Bus fault.** External logic asserts BUSFLT high to the TMS34020 to indicate that an error or fault has occurred on the current bus cycle. BUSFLT is also used with LRDY to generate bus-cycle retries so that the entire memory address is presented again on the LAD pins.
DDIN	O	**Data bus direction, input enable.** This active-high output is used to drive the active-high input enables on bidirectional transceivers (such as the 74ALS623). The transceivers buffer data input and output on the LAD0—LAD31 pins when the TMS34020 is interfaced to several memories.
$\overline{\text{DDOUT}}$	O	**Data bus direction, output enable**. This active-low signal drives the active-low output enables on bidirectional transceivers (such as the 74ALS623). The transceivers buffer data input and output on the LAD0—LAD31 pins.
LAD0—LAD31	I/O	**Multiplexed local address/data bus.** At the beginning of a memory cycle, the word address is output on LAD4—LAD31, and the cycle status is output on LAD0—LAD3. After the address is presented, LAD0—LAD31 are used for transferring data within the TMS34020 system. LAD0 is the LSB and LAD31 is the MSB.
LRDY	I	**Local ready.** External circuitry drives this signal low to stop the TMS34020 from completing a local-memory cycle it has initiated. While LRDY remains low, the TMS34020 will wait, unless the TMS34020 is given a retry request (through the BUSFLT signal). Wait states are generated in increments of one full LCLK1 cycle. LRDY can be driven low to extend local-memory read and write cycles, VRAM serial-data-register transfer cycles, and DRAM-refresh cycles. During internal cycles, the TMS34020 ignores LRDY.
$\overline{\text{PGMD}}$	I	**Page mode.** The memory decode logic asserts this signal low if the currently addressed memory supports burst (page mode) accesses. Burst accesses occur as a series of CAS cycles for a single RAS cycle to memory.
$\overline{\text{SIZE16}}$	I	**Bus size.** The memory decode logic may pull this signal low if the currently addressed memory or port supports only 16-bit transfers. $\overline{\text{SIZE16}}$ can also be used to determine which 16 bits of the data bus are used for a data transfer.

Table 2–6 lists the bus cycle completion conditions controlled by LRDY and BUSFLT.

Table 2–6. Bus-Cycle Completion Conditions

Completion Condition	BUSFLT	LRDY
Wait	0	0
Successful transfer	0	1
Retry	1	0
Bus fault	1	1

2.3.2 DRAM and VRAM Control Signals

Signal Name	I/O	Description
CAMD	I	**Column-address mode.** This input dynamically shifts the column address on the RCA0—RCA12 bus to allow the mixing of DRAM and VRAM address matrices using the same multiplexed address RCA0—RCA12 signals.
$\overline{\text{CAS0}}$—$\overline{\text{CAS3}}$	O	**Column-address strobes.** The $\overline{\text{CAS}}$ outputs drive the $\overline{\text{CAS}}$ inputs of DRAMs and VRAMs. These signals strobe the column address on RCA0—RCA12 to the memory. The four $\overline{\text{CAS}}$ strobes provide byte write access to the memory.
$\overline{\text{RAS}}$	O	**Row-address strobe.** The $\overline{\text{RAS}}$ output drives the $\overline{\text{RAS}}$ inputs of DRAMs and VRAMs. The high-to-low transition on this signal strobes the row address on RCA0—RCA12 to memory.
RCA0—RCA12	O	**Multiplexed row-address/column-address signals.** At the beginning of a memory access cycle, the row address for DRAMs is present on RCA0—RCA12. The row address contains the most significant address bits for the memory. As the cycle progresses, the memory column address is placed on RCA0—RCA12. The addresses that are actually output during row and column times depend on the memory configuration (set by RCM0 and RCM1 in the CONFIG register) and the state of CAMD during the access. RCA0 is the LSB and RCA12 is the MSB.
SF	O	**Special-function pin.** This is the special-function signal to 1M VRAMs. This signal allows the use of block write, load write mask, load color mask, and write using write mask. This signal is also used to differentiate instructions and addresses for the coprocessor as part of the coprocessor interface.
$\overline{\text{TR/QE}}$	O	**Transfer/output enable.** This signal drives the $\overline{\text{TR/QE}}$ input of VRAMs. During a local-memory read cycle, $\overline{\text{TR/QE}}$ functions as an active-low output enable to gate data from memory to LAD0—LAD31. During special VRAM function cycles, $\overline{\text{TR/QE}}$ controls the type of cycle that is performed.
$\overline{\text{WE}}$	O	**Write enable.** The active low $\overline{\text{WE}}$ output drives the $\overline{\text{WE}}$ inputs of DRAMs and VRAMs. $\overline{\text{WE}}$ can also be used as the active-low write enable to static memories and other devices connected to the TMS34020 local interface. During a local-memory read cycle, $\overline{\text{WE}}$ remains inactive high while $\overline{\text{CAS}}$ is strobed active low. During a local-memory write cycle, $\overline{\text{WE}}$ is strobed active low before $\overline{\text{CAS}}$ is. During VRAM serial-data-register transfer cycles, the state of $\overline{\text{WE}}$ at the falling edge of $\overline{\text{RAS}}$ controls the direction of the transfer.

2.3.3 Multiprocessor Interface Signals

The multiprocessor interface allows multiple TMS34020s to share the same local memory by providing a request/grant protocol for devices that want to access shared memory.

Signal Name	I/O	Description
$\overline{GI}$	I	**Bus grant input.** External bus arbitration logic drives $\overline{GI}$ low to enable the TMS34020 to gain access to the local-memory bus. The TMS34020 must release the bus if $\overline{GI}$ is high so that another device can access the bus.
$\overline{R1}$, $\overline{R0}$	O	**Bus request and control.** These two signals indicate a request for use of the bus in a multiprocessor system; they are decoded as shown below.

$\overline{R1}$	$\overline{R0}$	Bus Request Type
0	0	High-priority bus request
0	1	Bus cycle termination
1	0	Low-priority bus request
1	1	No bus request pending

- A high-priority bus request provides for VRAM serial-data-register transfer cycles, DRAM refresh (when 12 or more refresh cycles are pending), or a host-initiated access. The external arbitration logic should grant this request as soon as possible by asserting $\overline{GI}$ low.
- A low-priority bus request is used to provide for CPU-requested access and DRAM refresh (when less than 12 refresh cycles are pending).

Bus cycle termination status is provided so that the arbitration logic can determine that the device currently accessing the bus is completing an access and other devices may compete for the next bus cycle. A *no bus request pending* status is output when the currently active device does not require the bus on subsequent cycles.

2.3.4 Host Interface Signals

The host interface signals are used for communication between the TMS34020 and a host processor. Signals input on these pins are assumed to be asynchronous with respect to the local clocks (LCLK1 and LCLK2). Signals output on these pins are synchronized only when responses are dependent on memory cycles that must be generated by the TMS34020.

The host interface allows the TMS34020's memory to be mapped into a host processor's address space. The TMS34020 can act as a DRAM controller for a host processor. The address of the required access is input to the TMS34020, and data is transferred through external transceivers.

Signal Name	I/O	Description
HA5—HA31	I	**27 host-address input signals.** A host can access a long-word by placing the address on these lines. HA5—HA31 correspond to the LAD5—LAD31 signals that output the address to the local memory.
HBS0—HBS4	I	**4 host byte selects.** The byte selects identify which bytes within the long-word are being selected.
$\overline{\text{HCS}}$	I	**Host chip select.** A host drives this signal low to latch the current host address present on HA5—HA31 and the host byte selects on HBS0—HBS3. This signal also enables host access cycles to the TMS34020 I/O registers or local memory. During the low-to-high transition of $\overline{\text{RESET}}$, the level on the $\overline{\text{HCS}}$ input determines whether the TMS34020 is halted ($\overline{\text{HCS}}$ is high for host-present mode) or whether it begins executing its reset service routine ($\overline{\text{HCS}}$ is low for self-bootstrap mode).
HDST	O	**Host data strobe.** The rising edge of this signal latches data from the TMS34020 local address space to the external transceivers on host read accesses. It can be used in conjunction with HRDY to indicate that data is valid in the external transceivers.
$\overline{\text{HINT}}$	O	**Host interrupt.** This signal allows the TMS34020 to interrupt a host by setting the INTOUT bit in the HSTCTLL I/O register. This signal can also be used to interrupt the host if a BUSFLT or RETRY occurs due to a host access cycle.
$\overline{\text{HOE}}$	O	**Host-data output enable.** This signal enables data from the external transceivers to the TMS34020 local address space on host write cycles. $\overline{\text{HOE}}$ can be used in conjunction with HRDY to indicate data has been written to memory from the external transceivers.
HRDY	O	**Host ready.** This signal is normally low and goes high to indicate that the TMS34020 is ready to complete a host-initiated read or write cycle. A host can use HRDY logically combined with HDST and $\overline{\text{HOE}}$ to determine when the local bus access cycles have completed.
$\overline{\text{HREAD}}$	I	**Host read strobe.** This signal is driven low during a read request from a host processor. This notifies the TMS34020 that the host is requesting access to local memory or to the I/O registers. $\overline{\text{HREAD}}$ should not be asserted at the same time that $\overline{\text{HWRITE}}$ is asserted.
$\overline{\text{HWRITE}}$	I	**Host write strobe.** This signal is driven low to indicate a write request by a host processor. This notifies the TMS34020 that a write request is pending. The rising edge of $\overline{\text{HWRITE}}$ is used to indicate that the data provided by the host in the external data transceivers can be written. $\overline{\text{HWRITE}}$ should not be asserted at the same time $\overline{\text{HREAD}}$ is asserted.

2.3.5 Video Interface Signals

Signal Name	I/O	Description
$\overline{\text{CBLNK}}/\overline{\text{VBLNK}}$	O	**Composite blanking/vertical blanking.** You can program this signal to select one of two blanking functions: ❑ **Composite blanking** for blanking the display during both horizontal- and vertical- retrace periods in *composite-sync video mode.* ❑ **Vertical blanking** for blanking the display during vertical retrace in *separate-sync video mode.* Immediately following reset, this signal is configured as $\overline{\text{CBLNK}}$ output.
$\overline{\text{CSYNC}}/\overline{\text{HBLNK}}$	I/O	**Composite sync/horizontal blanking.** You can program this signal to select one of two functions: ❑ **Composite sync** (either input or output as set by a control bit in the DPYCTL register) in *composite-sync video mode*: ■ As an **input**, $\overline{\text{CSYNC}}$ synchronizes the TMS34020 video-control registers to externally generated horizontal-sync pulses. The actual synchronization can be programmed to begin at any VCLK cycle; this allows for any external pipelining of signals. $\overline{\text{CSYNC}}$ extracts $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ from externally generated horizontal-sync pulses. ■ As an **output**, $\overline{\text{CSYNC}}$ is the active-low composite-sync pulse generated by the TMS34020's on-chip video timers. ❑ **Horizontal blank** (output only) for blanking the display during horizontal retrace in *separate-sync video mode.* Immediately following reset, this signal is configured as a $\overline{\text{CSYNC}}$ input.
$\overline{\text{HSYNC}}$	I/O	**Horizontal sync.** $\overline{\text{HSYNC}}$ is the horizontal-sync signal that controls external video circuitry. You can program this signal to be either an input or an output by modifying a control bit in the DPYCTL register. ❑ As an **output**, $\overline{\text{HSYNC}}$ is the active-low horizontal-sync signal generated by the TMS34020's on-chip video timers. ❑ As an **input**, $\overline{\text{HSYNC}}$ synchronizes the TMS34020 video-control registers to externally generated horizontal-sync pulses. The actual synchronization can be programmed to begin at any VCLK cycle; this allows for any external pipelining of signals. Immediately following reset, $\overline{\text{HSYNC}}$ is configured as an input.
SCLK	I	**Serial data clock.** This signal is the same as the signal that drives VRAM serial-data registers. This allows the TMS34020 to track the VRAM serial-data-register count, providing serial-register-transfer midline-reload cycles. (SCLK may be asynchronous to VCLK; however, it typically has a frequency that is a multiple of the VCLK frequency.)
VCLK	I	**Video clock.** This clock is a derivative of the video system's dotclock and is used internally to drive the video timing logic.

Signal Name	I/O	Description
$\overline{\text{VSYNC}}$	I/O	**Vertical sync.** $\overline{\text{VSYNC}}$ is the vertical-sync signal that controls external video circuitry. You can program this signal to be either an input or an output by modifying a control bit in the DPYCTL register. ❑ As an **output**, $\overline{\text{VSYNC}}$ is the active-low vertical-sync signal generated by the TMS34020's on-chip video timers. ❑ As an **input**, $\overline{\text{VSYNC}}$ synchronizes the TMS34020 video-control registers to externally generated vertical-sync pulses. The actual synchronization can be programmed to begin at any horizontal line; this allows for any external pipelining of signals. Immediately following reset, $\overline{\text{VSYNC}}$ is configured as an input.

2.3.6 System Control Signals

Signal Name	I/O	Description
CLKIN	I	**Clock input.** This system input clock is used to generate the LCLK1 and LCLK2 outputs, to which all processor functions in the TMS34020 are synchronous. A separate asynchronous input clock (VCLK) controls the video timing and video registers.
LCLK1, LCLK2	O	**Local output clocks.** These two clocks are 90 degrees out of phase with each other. They provide convenient synchronous control of external circuitry to the internal timing. All signals output from the TMS34020 (except the CRT timing signals) are synchronous to these clocks.
$\overline{\text{LINT1}}$, $\overline{\text{LINT2}}$	I	**Local interrupt requests.** Interrupts from external devices are transmitted to the TMS34020 on $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$. Each local interrupt signal activates the request for one of two interrupt request levels. An external device generates an interrupt request by driving the appropriate interrupt request pin to its active-low state. These signals can be applied asynchronously to the TMS34020 as they are synchronized internally before use. The signal should remain low until it is recognized by the TMS34020.
$\overline{\text{RESET}}$	I	**System reset.** $\overline{\text{RESET}}$ is normally high. During normal operation, $\overline{\text{RESET}}$ is driven low to reset the TMS34020. When $\overline{\text{RESET}}$ is asserted low, the TMS34020's internal registers are set to an initial known state, all output pins are driven to inactive levels, and all bidirectional pins are driven to a high-impedance state. The TMS34020's behavior following reset depends on the level of the $\overline{\text{HCS}}$ input just before the low-to-high transition of $\overline{\text{RESET}}$. If $\overline{\text{HCS}}$ is low, the TMS34020 begins executing the instructions pointed to by the reset vector. If $\overline{\text{HCS}}$ is high, the TMS34020 is halted until a host processor writes a 0 to the HLT bit in the HSTCTLL register.

2.3.7 Power Signals

Signal Name	I/O	Description
V_{CC}	I	Nominal 5-volt power supply inputs (5 pins for the QFP, 9 pins for the PGA)
V_{SS}	I	Electrical ground inputs (9 pins for the QFP, 17 pins for the PGA)

Note: For proper TMS34020 operation, all these signals must be connected externally.

Chapter 3

Memory Organization and Data Structures

Much of the TMS34020's power derives from its flexible memory access. Several memory organization features are tailored specifically for graphics applications:

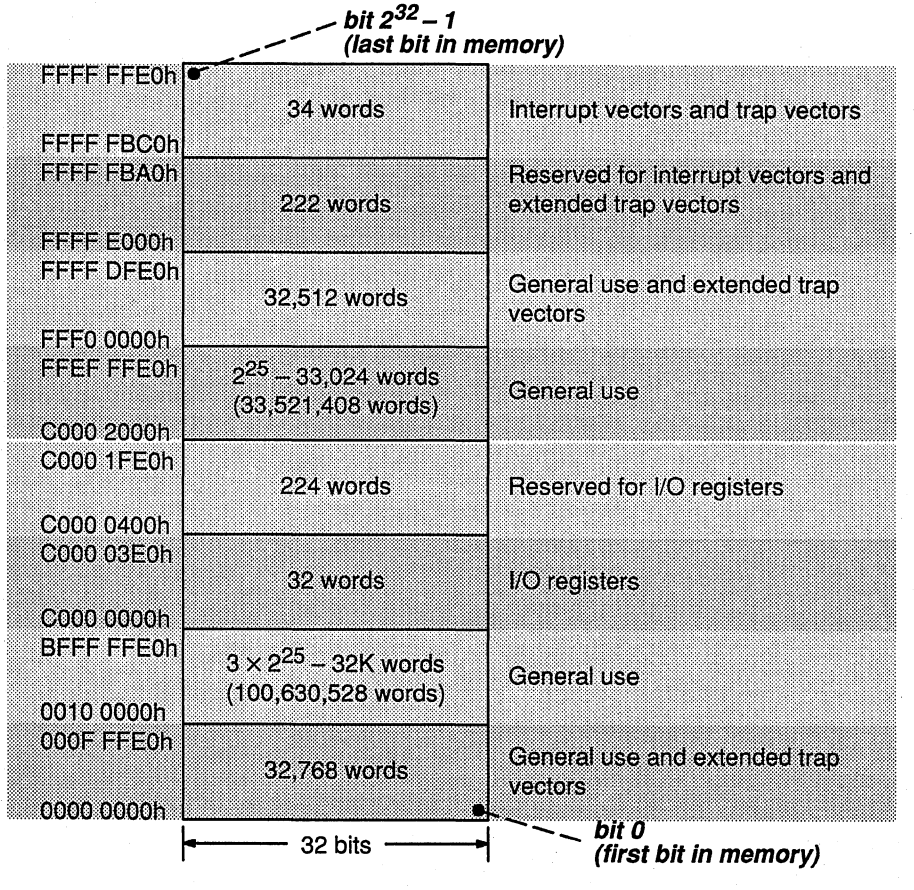
- A large memory space supports a variety of display resolutions.
- You can access memory locations with linear or XY addresses.
- The TMS34020 provides hardware support for several data structures:
  - **Fields** are configurable data structures. A field can begin and end at any bit address and can be 1 to 32 bits long.
  - As used by the TMS34020, **bytes** are a special type of field; byte length is fixed at 8 bits.
  - **Pixels** are configurable data structures; pixel length can be any power of 2 in the range of 1 to 32 bits.
  - **Pixel arrays** are 2-dimensional, rectangular blocks of pixels.

Additionally, the TMS34020 can be addressed in little-endian or big-endian mode, and provides a system stack. Unless explicitly stated otherwise, all discussions refer to little-endian addressing.

3.1 Memory Map

Figure 3–1 illustrates the TMS34020 memory map.

Figure 3–1. TMS34020 Memory Map



Memory is logically organized as 4 gigabits, but is physically accessed 32 bits at a time. Figure 3–1 shows locations as long (32-bit) words, identified by 32-bit addresses. Word addresses range from 0000 0000h to FFFF FFE0h (bit address 0000 0000h is the rightmost bit in the word at the bottom of Figure 3–1, and bit address FFFF FFFFh is the leftmost bit in the word at the top.) Reading or writing to an address in the range C000 0000h to C000 03E0h accesses an internal I/O register. (An external memory cycle is also generated on accesses to these locations, allowing the I/O registers to be shadow mapped in external memory.) Reading or writing to any address outside this range accesses external memory (or a memory-mapped device).

As Figure 3–1 shows, memory is divided into several regions:

- **General use**

  Address ranges 0h—BFFF FFE0h and C000 2000h—FFFF DFE0h are for general use (executable code, data tables, etc.).

- **I/O registers**

  Addresses C000 0000h—C000 03E0h are reserved for the internal I/O registers. Chapter 4 discusses the I/O registers; it contains a map of this memory area that associates each I/O register with the appropriate address.

- **Interrupt, reset, and trap vectors**

  Addresses FFFF FBC0h—FFFF FFE0h are reserved for 34 interrupt, reset, and trap vectors. A vector is a 32-bit address that points to the starting location in memory of the appropriate interrupt, reset, or trap service routine. Chapter 6 contains more information about interrupts and traps.

- **Reserved memory**

  Addresses C000 0400h—C000 1FE0h are reserved for future expansion of the I/O registers. Addresses FFFF E000h—FFFF FBA0h are reserved for future expansion of the interrupt vectors.

3.2 Memory Addressing

The TMS34020 is a bit-addressable machine with a 32-bit memory address. The total memory capacity is 4 gigabits (512 Mbytes). Memory is accessed as a continuously addressable string of bits; each 32-bit address points to an individual bit within memory. Bit addresses range from 0000 0000h to FFFF FFFFh.

Figure 3–2 illustrates the TMS34020's logical memory structure.

Figure 3–2. Logical Memory Address Space

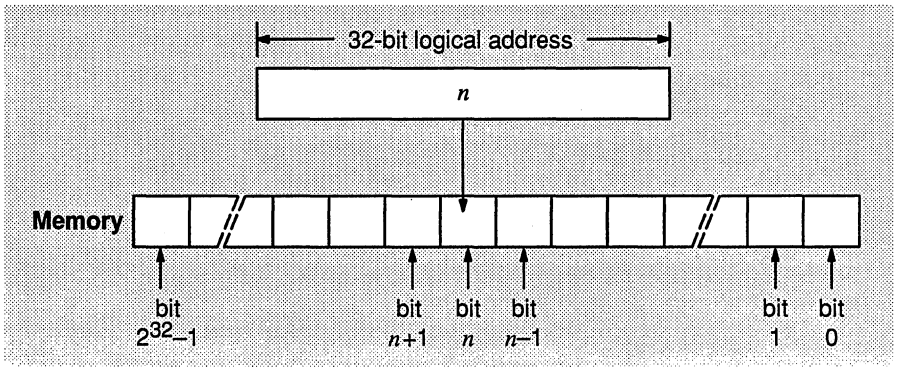
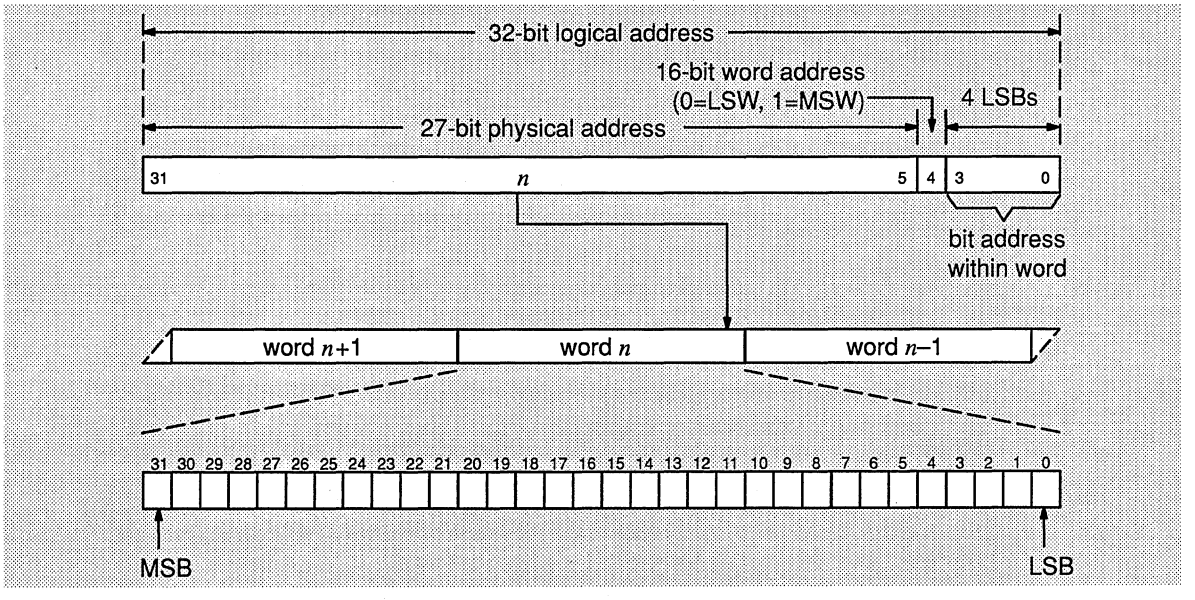


Figure 3–3 illustrates physical memory organization.

Figure 3–3. Physical Memory Addressing



The TMS34020 communicates with memory over a 32-bit address/data bus (LAD0—LAD31) and always reads a complete long (32-bit) word from memory. Writes to memory may be 8-, 16-, 24-, or 32-bit values through the use of the TMS34020's $\overline{\text{CAS}}$ (byte) strobes.

A long-word accessed during a memory cycle always begins on a 32-bit boundary; thus, the 5 LSBs of the 32-bit starting address of the word are always 0s. Bits within a word are numbered from 0 to 31; bit 31 is the MSB and bit 0 is the LSB. A word is identified by the address of its LSB.The LSB of a memory word is depicted as the rightmost bit in the word.

The 4 LSBs of the 32-bit logical address in Figure 3–3 do not appear on the LAD bus. Bit 4 is output for use with 16-bit memory devices only. When the TMS34020 accesses a field that does not begin and end on long-word boundaries, these 5 LSBs are used internally to identify a bit boundary within an accessed long-word.

Internal logic automatically performs the bit alignment and masking necessary to extract a field from physical memory; this is completely transparent to software. Similarly, inserting a field into memory may require a series of read and write cycles, accompanied by internal masking and shifting of data to properly align the data structure within memory. The memory control logic performs these tasks automatically.

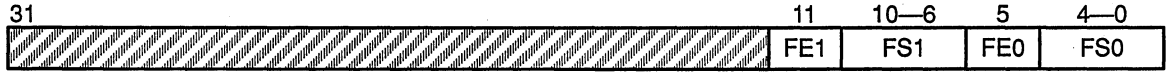
3.3 Fields

The TMS34020 supports 2 software-configurable field types, **field 0** and **field 1**. A field is defined by 2 parameters:

- **Starting address.** A field's starting address is the address of the field's LSB. A field can begin at an arbitrary bit address in memory. When a field is moved from memory to a general-purpose register, the field is right-justified within the register; that is, the field's LSB coincides with the register's rightmost bit (bit 0). The register bits to the left of the field are all 1s or all 0s, depending on the values of both the appropriate FE (field extension) status bit and the field's sign bit (MSB). If FE=1 the field is sign-extended; if FE=0, the field is zero-extended.
- **Field size.** Field size can range from 1 to 32 bits. The lengths of fields 0 and 1 are defined by two 5-bit fields in the status register, FS0 and FS1.

Figure 3–4 identifies the status bits that control the size and extension of field 0 and field 1. Table 3–1 shows how the field size is encoded in FS0 and FS1.

Figure 3–4. Status Bits That Control Field 0 and Field 1



Note: For a complete description and illustration of the ST, see Chapter 4.

Table 3–1. Decoding the Field-Size Bits in the Status Register

5 FS Bits	Field Size	5 FS Bits	Field Size	5 FS Bits	Field Size	5 FS Bits	Field Size
00001	1	01001	9	10001	17	11001	25
00010	2	01010	10	10010	18	11010	26
00011	3	01011	11	10011	19	11011	27
00100	4	01100	12	10100	20	11100	28
00101	5	01101	13	10101	21	11101	29
00110	6	01110	14	10110	22	11110	30
00111	7	01111	15	10111	23	11111	31
01000	8	10000	16	11000	24	00000	32

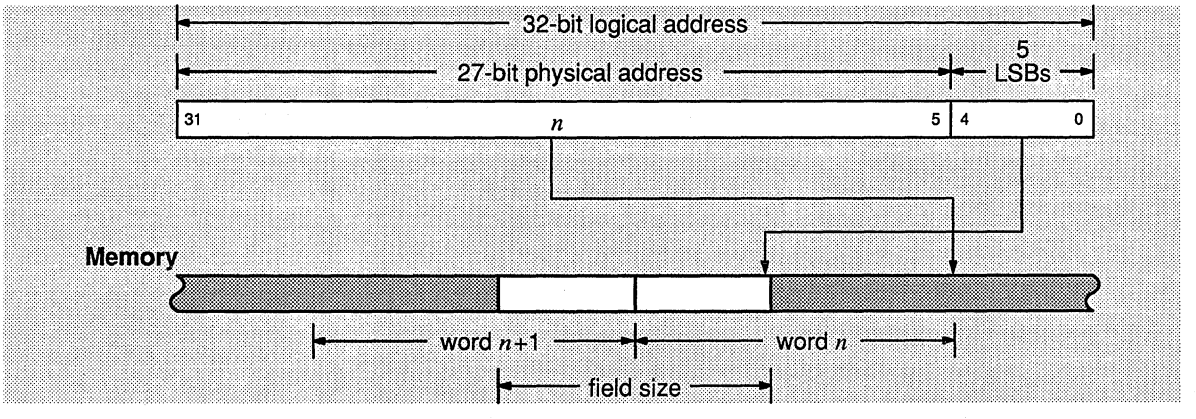
Figure 3–5 illustrates a field in memory. In this example, the field straddles the boundary between words n and $n+1$ in memory. Field extraction and insertion is performed by on-chip hardware:

- To move the field **to** a general-purpose register, the TMS34020 extracts the field from memory by reading word n and word $n+1$ in separate cycles.

- To move the field **from** a general-purpose register, the TMS34020 inserts the field into memory by reading and writing word *n* and reading and writing word *n*+1.

The memory operations necessary to insert or extract a field are performed automatically by special hardware and are transparent to software.

Figure 3–5. Field Storage in External Memory



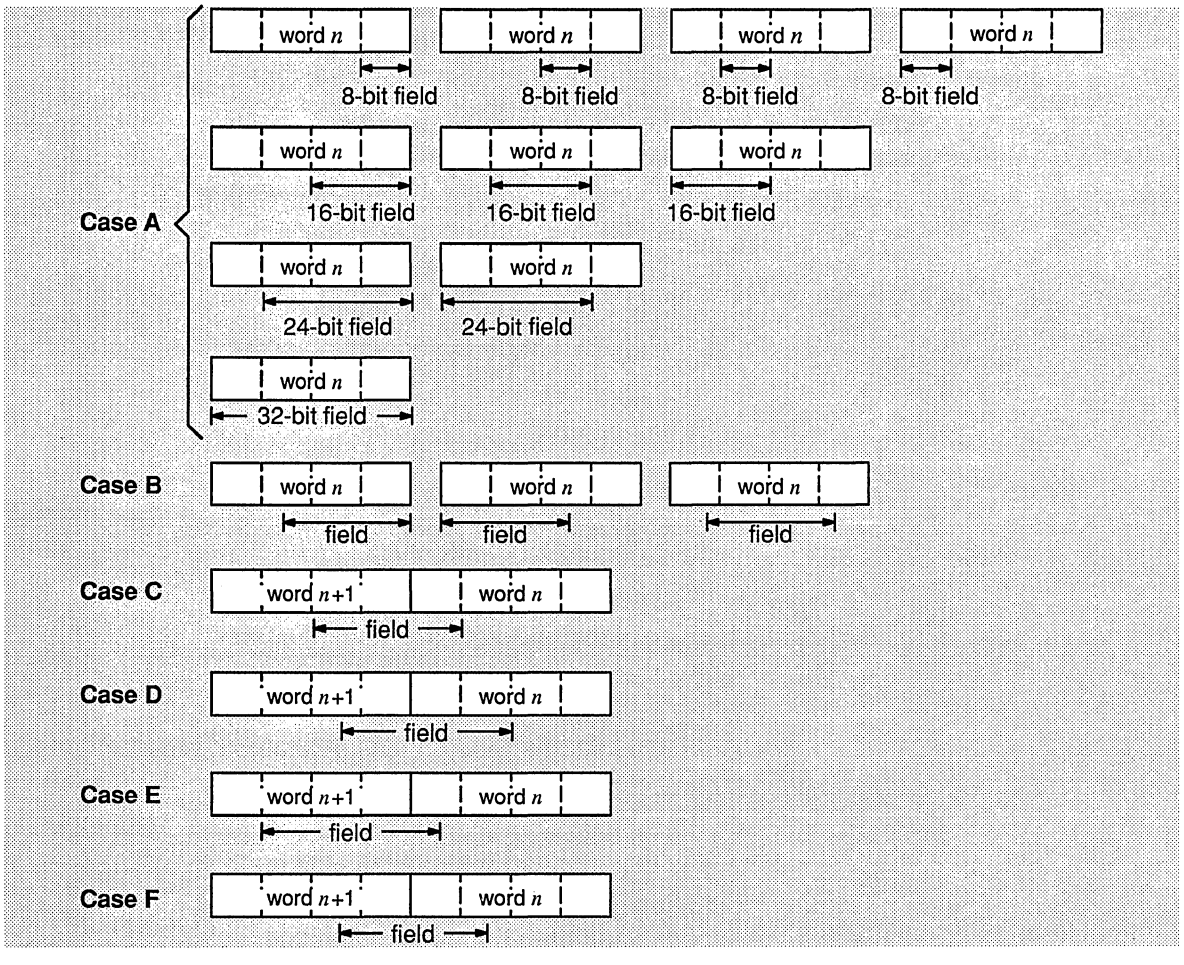
In Figure 3–5, word *n* is pointed to by a 27-bit physical address, output by the TMS34020 to memory. This 27-bit address corresponds to bits 5—31 of the field's 32-bit logical address. The 5 LSBs of the logical address point to the beginning of the field within word *n*.

The number of memory cycles required to extract or insert a field depends on how the field is aligned within memory. Field manipulation is more rapid when fields are stored in memory so that they do not cross word boundaries. Figure 3–6 illustrates various cases of alignment and nonalignment of fields to word boundaries in memory. Given a field starting address and field length, the memory controller will recognize the specified field alignment as one of the 6 cases in Figure 3–6. Field extraction and field insertion are performed in a manner that requires the minimum number of memory cycles.

- **Cases A1—A4.** The field begins and ends on byte boundaries within a single word.
  - In Case A1, the field is 8 bits wide and the starting address is aligned to a byte boundary within a word.
  - In Case A2, the field is 16 bits wide and the starting address is aligned to the first, second, or third byte boundary within a word.
  - In Case A3, the field is 24 bits wide and is aligned to the first or second byte boundary in a word.
  - In Case A4, the field is 32 bits wide and is word-aligned.

  For Cases A1—A4, a field extraction requires a single read cycle, and a field insertion requires a single write cycle.

Figure 3–6. Field Alignment in Memory



- **Case B.** The field does not straddle a word boundary and does not begin and end on byte boundaries (that is, either it is not aligned on a byte boundary, or it is aligned on a byte boundary but is not a multiple of 8 bits). A field extraction requires a single read cycle. A field insertion requires the following sequence of memory cycles:
  - Read word *n*
  - Write word *n*
- **Case C.** The field straddles the boundary between 2 words and begins and ends on byte boundaries. A field extraction requires the following sequence of memory cycles:
  - Read word *n*
  - Read word *n*+1

A field insertion requires the following sequence or memory cycles:

- Write word n
- Write word n+1

❑ **Case D.** The field straddles the boundary between 2 words. The field address is byte aligned, but the end of the field does not coincide with the end of a byte. A field extraction requires the following sequence of memory cycles:

- Read word n
- Read word n+1

A field insertion requires the following sequence of memory cycles:

- Write word n
- Read word n+1
- Write word n+1

❑ **Case E.** The field straddles the boundary between 2 words. The end of the field is byte aligned, but the start is not. A field extraction requires the following sequence of memory cycles:

- Read word n
- Read word n+1

A field insertion requires the following sequence of memory cycles:

- Read word n
- Write word n
- Write word n+1

❑ **Case F.** The field straddles the boundary between 2 words and neither the start nor the end of the field is aligned to a byte boundary. A field extraction requires the following sequence of memory cycles:

- Read word n
- Read word n+1

A field insertion requires the following sequence of memory cycles:

- Read word n
- Write word n
- Read word n+1
- Write word n+1

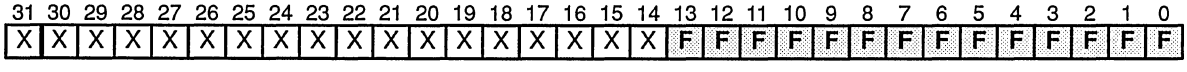
A field insertion modifies only the portion of a word that lies within a field. The TMS34020 memory controller must perform a read-modify-write operation when a field that does not begin and end on byte boundaries is written to memory. The memory controller uses these 2 parameters (address LSBs and

field size) to produce a mask that identifies the bits in the word corresponding to the field. Hardware uses the mask to perform the read-modify-write cycle. The TMS34020's local memory control logic automatically generates the mask and executes the read-modify-write operation; this is transparent to the software.

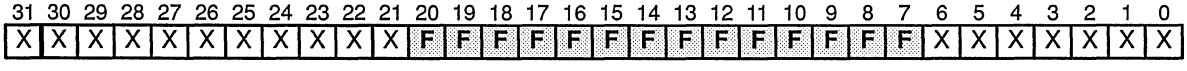
Figure 3–7 shows an example of inserting a 14-bit field stored in a register to logical address 0000 0007h.

Figure 3–7. Field Insertion

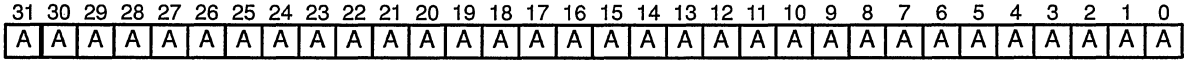
(a) Field to be inserted



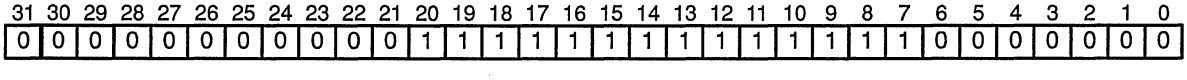
(b) Rotate to align to bit 7



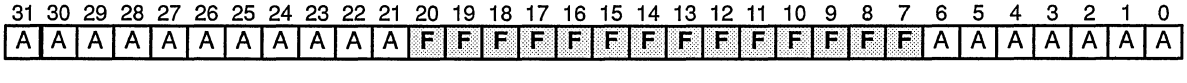
(c) Initial destination data



(d) Mask generated



(e) Field destination data



(a) The field to be inserted is shown right-justified in the designated general-purpose register.

(b) The CPU has rotated the field to align it with the destination in memory.

(c) The TMS34020 reads the original word from the destination in memory.

(d) The mask is generated to designate the bits to be modified.

(e) The field is inserted into the word from memory, and the result is written back to the destination address in memory.

In the more complex case in which a field straddles a word boundary in memory, the portion of the field lying within each word is inserted into that word using the methods described above.

3.4 Pixels

The term **pixel** has two meanings in the context of a TMS34020-based graphics system. Outside the TMS34020, a physical pixel is a picture element on a display surface. Inside the TMS34020, a logical pixel is a software configurable data structure supported by the TMS34020 instruction set. The logical pixel data structure in TMS34020 memory contains the information needed to specify the attributes of a picture element visible on a screen. The information for a horizontal line of pixels on a screen is usually stored in consecutive words in memory.

3.4.1 Pixels in Memory

Within TMS34020 memory, the pixel data structure is defined by 2 parameters:

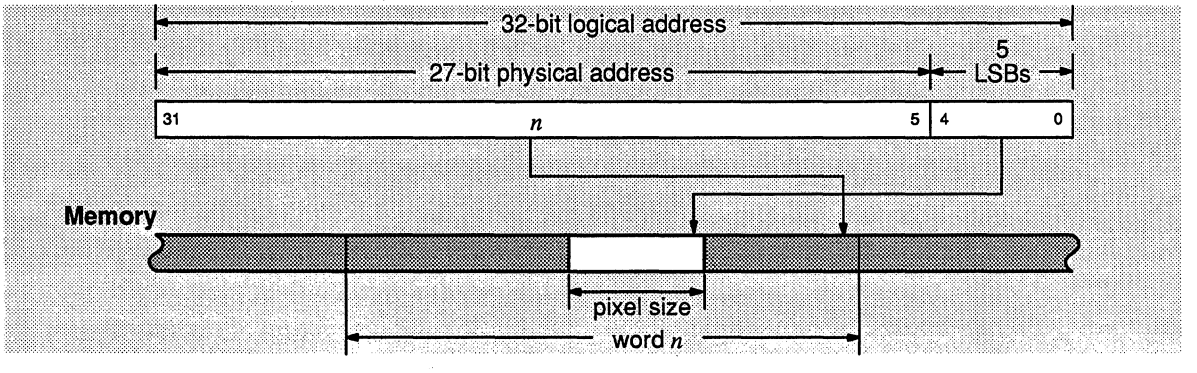
- its starting address (the address of the pixel's LSB) and
- the pixel size (the number of bits per pixel).

The PSIZE register defines the current pixel size. A pixel can be 1, 2, 4, 8, 16, or 32 bits long. The TMS34020 treats pixels as a special case of a field in which the field size is constrained to be a power of 2. Unlike other memory fields, pixels do not cross long-word boundaries within memory; they are aligned within memory so that a memory word contains an integral number of pixels. For example, a 2-bit pixel should begin at a bit address whose LSB is 0, a 4-bit pixel should begin at a bit address whose 2 LSBs are 0s, and so forth.

When a pixel is moved from memory to a general-purpose register, the pixel is right-justified within the register. That is, the pixel's LSB coincides with the rightmost bit (bit 0) of the register. Register bits to the left of the pixel are loaded with 0s.

Figure 3–8 illustrates pixel storage in memory. The pixel is located within the word pointed to by the 27-bit physical address corresponding to bits 5—31 of the pixel's 32-bit logical address. The 5 LSBs of the logical address specify the displacement of the pixel within the word. When the pixel length is less than 32, each word contains 2 or more pixels.

Figure 3–8. Pixel Storage in External Memory

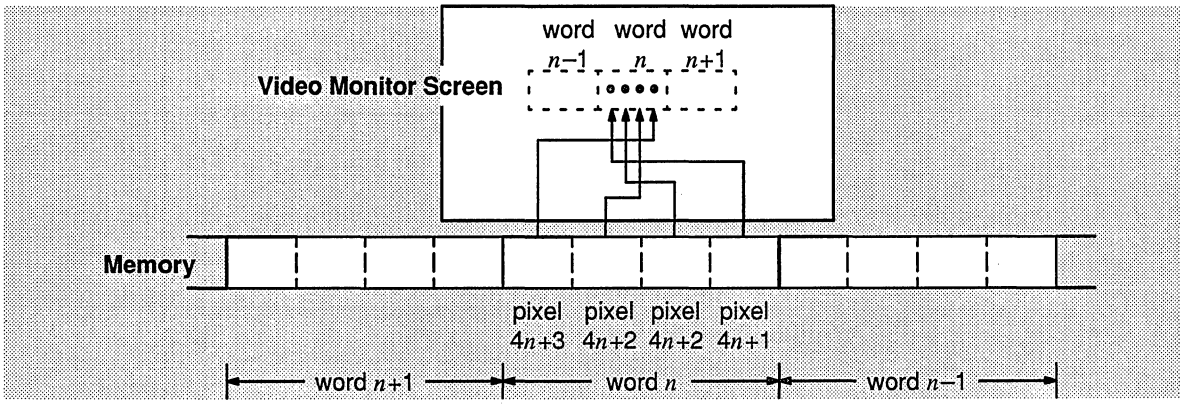


On-chip hardware performs pixel extraction and insertion in a manner that requires the minimum number of memory cycles. (The operations are transparent to software.) Two memory cycles (a read followed by a write) are always required to insert a pixel of less than 8 bits. Inserting an 8-, 16-, or 32-bit pixel requires a single write cycle (unless plane masking is enabled). Extracting a pixel (1 to 32 bits) requires a single read cycle.

3.4.2 Pixels on the Screen

Figure 3–9 illustrates the mapping of pixels from memory to a display screen. The screen-refresh function outputs pixels in the sequence of ascending pixel addresses. However, the electron beam sweeps from the left edge of the screen to the right edge during each horizontal scan interval, so pixels appear on the screen in the opposite order of their representation in memory. That is, the least significant pixel (in terms of bit address) appears on the left, and the most significant pixel appears on the right.

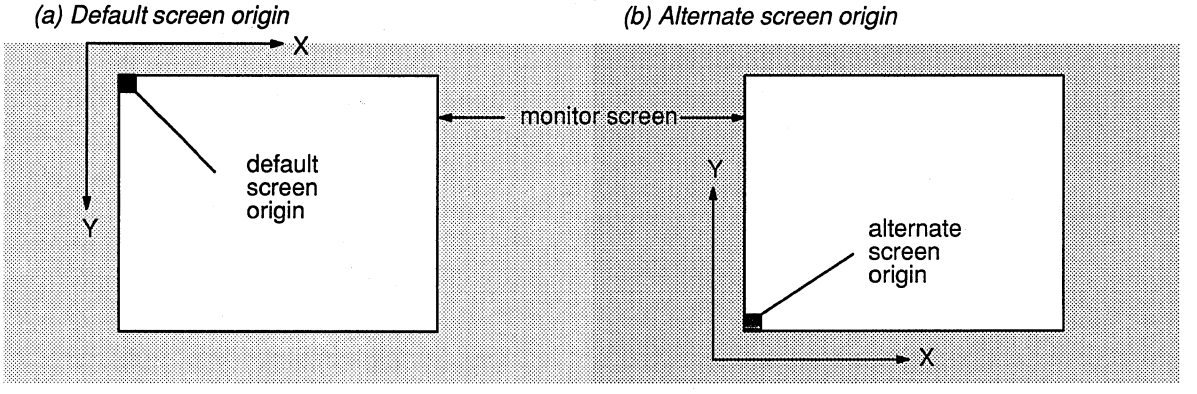
Figure 3–9. Mapping of Pixels to a Monitor Screen



The TMS34020 allows you to identify a pixel in terms of its XY coordinates on the screen or in terms of the address of the logical pixel in memory. These 2 methods are called **XY addressing** and **linear addressing**, respectively.

When you use XY addressing, you can select the origin to lie in either the upper left or lower left corner of the screen. The DPYST and DINC registers control the origin's position. Figure 3–10 *(a)* illustrates the default coordinate system in which the origin of the 2 coordinate axes is located in the upper left corner of the screen. In this system, DPYST contains the address of the pixel at the upper left of the screen, and DINC contains the display pitch. Figure 3–10 *(b)* shows the alternate coordinate system in which the origin is located in the lower left corner of the screen. In this case, DPYST contains the address of the pixel at the lower left of the screen, and DINC contains the 2s compliment of the display pitch.

Figure 3–10. Configurable Screen Origin



Using the default screen origin, Figure 3–11 illustrates the mapping of pixels from the memory to the screen. In Figure 3–11, horizontal movement represents travel in the X direction on the screen. Vertical movement represents travel in the Y direction. The depth of the buffer represents the pixel size. The on-screen memory contains the pixels that appear on the screen.

In Figure 3–11, the display memory is shown in terms of a *screen format*, rather than the memory format used in the memory map in Figure 3–1 (page 3-2). The screen format places the lowest pixel address at the upper left corner of the memory map. This is the same relative orientation in which the pixels appear on the screen. Compare this to the memory format shown in Figure 3–1, which places the lowest bit address at the lower right corner of the memory map. This convention is frequently used in industry to represent the relative location of addresses in memory. In this user's guide, assume the standard format is used unless the screen format is explicitly indicated.

Figure 3–11. Display Memory Dimensions

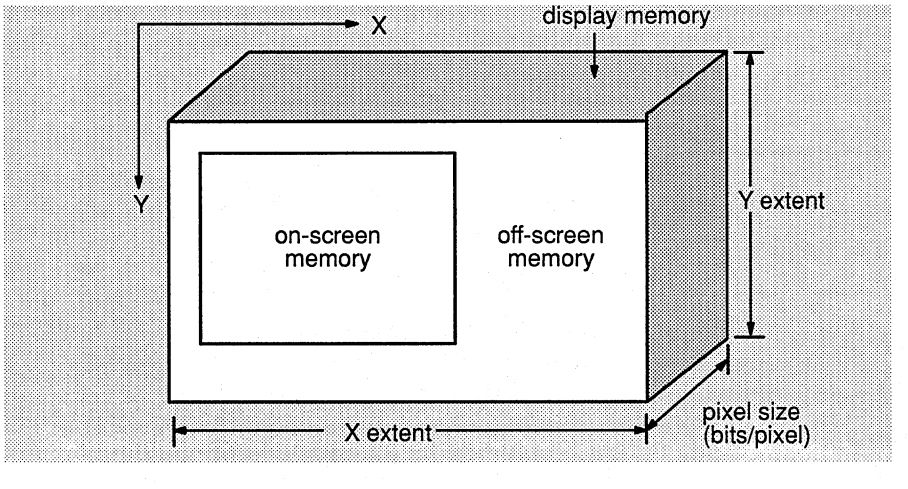
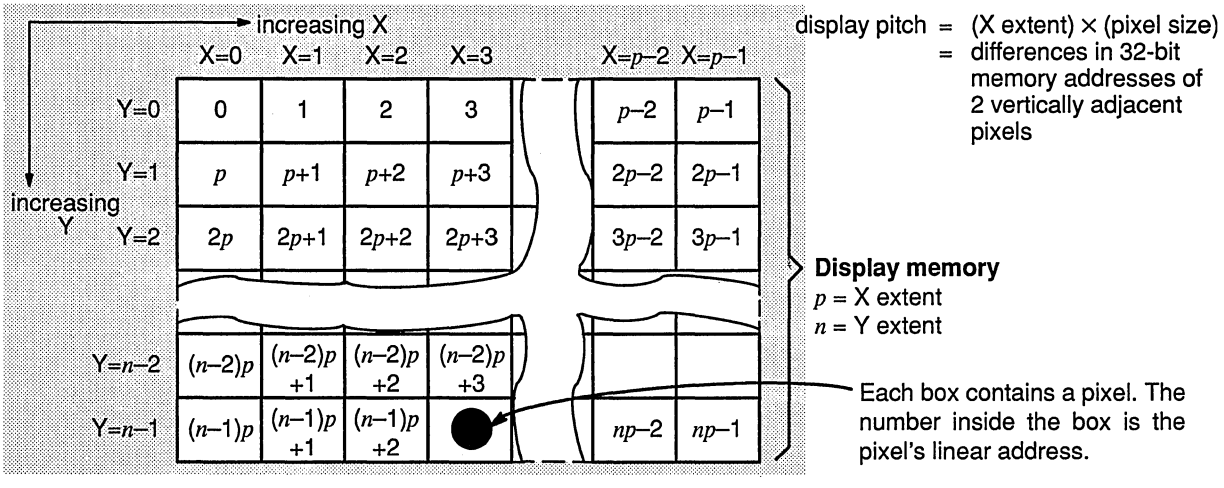


Figure 3–12 illustrates the mapping of XY coordinates to on-screen memory. For simplicity, assume that the screen origin coincides with the upper left corner of the display memory. p represents the X extent of the display memory; n represents the Y extent. Each box represents a pixel within the memory; the number in the box represents the pixel's memory location, relative to the beginning of the on-screen memory.

Figure 3–12. Display Memory Coordinates



3.4.3 Display Pitch

Display pitch is the difference in memory addresses between 2 pixels that are vertically adjacent on the screen (one is directly above the other). In Figure 3–12, the pitch is calculated as p times the pixel size, where p is the X extent of the display memory. The pixel size is constrained to be a power of 2, so the multiply can be replaced by a shift operation. **Array pitch** is the difference in memory addresses of 2 vertically adjacent pixels in the array. If the array occupies a rectangular area on the screen, the array pitch is the same as the display pitch.

During a pixel operation such as a PIXBLT, the source array pitch, destination array pitch, and (if it is a masked PIXBLT) mask array pitch are defined in separate, dedicated hardware registers. This eases the transfer of pixel arrays between on-screen and off-screen memory, which may have different pitches.

As an example, here's how you would calculate the display pitch if the pixel size = 4 bits and the X extent of the pixel display = 1024 pixels:

display pitch = (1024 pixels per line) × (4 bits per pixel)
= 4096 (which is 2^{12})

Note that the TMS34020 does not require the display pitch to be a power of 2, as was the case for the TMS34010.

3.5 XY Addressing

The TMS34020 allows you to define pixel addresses in terms of 2-dimensional XY coordinates that correspond to screen locations. This is referred to as **XY addressing**. XY addressing has several benefits:

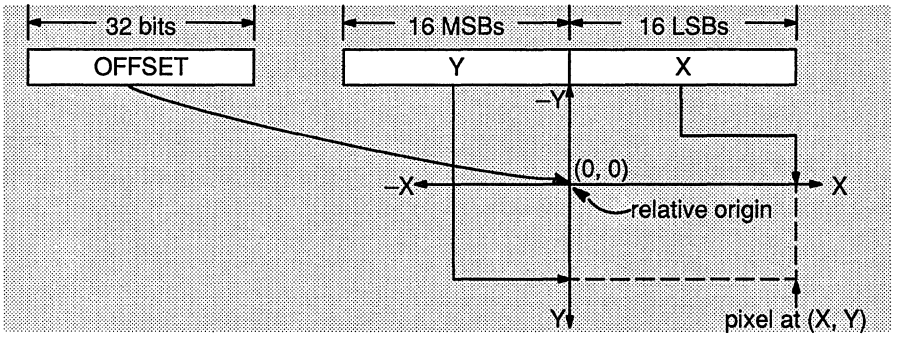
- TMS34020 software can be easily ported from one display configuration to another. System-dependent details, such as the number of bits per pixel and the X extent of the display memory, are transparent to the software. However, these are used by the machine to automatically convert the XY coordinates to the address of a pixel in memory.
- XY addressing allows you to think in terms of the high-level concept of XY coordinates rather than in terms of the machine-level mapping of pixels into memory.
- XY addressing facilitates operations such as window checking.

The TMS34020 supports XY coordinates in the range (–32768,–32768) to (+32767,+32767).

The TMS34010 did not support signed XY values, as the TMS34020 does.

Figure 3–13 illustrates the XY addressing format.

Figure 3–13. Pixel Addressing in Terms of XY Coordinates



In Figure 3–13, a 32-bit general-purpose register contains an XY address. The X and Y components are treated as separate16-bit signed integers. The X component is right-justified within the 16 LSBs of the register. The Y component is right-justified within the 16 MSBs of the register.

3.6 Converting an XY Address to a Linear Address

For all instructions that use XY addressing, the TMS34020 automatically converts a pixel's XY address to a 32-bit logical address (linear address). The TMS34020 uses four parameters to perform XY-to-linear conversion:

logical pixel size	defined in the PSIZE register
pitch conversion factor	defined in the CONVSP, CONVDP, or CONVMP register
actual pitch	defined in the SPTCH, DPTCH, or MPTCH register if the conversion involves a pitch that is not a power of 2 or a sum two of powers of 2
offset	specifies the XY origin, defined in the OFFSET register

The TMS34020 uses the following formula to calculate the physical address associated with the XY address:

$$\textbf{address} = [(Y \times \textit{display pitch}) + (X \times \textit{pixel size})] + \textit{offset}$$

Because the pixel size must always be a power of 2, the multiplication of the X component is performed using a shift operation. The method of calculating the Y component depends on the pitch value.

If the pitch is. . .	This is how the Y value is calculated
a power of 2	The TMS34020 performs a left shift. The amount that the component is shifted is contained in the lower half of the appropriate CONVxP register.
two powers of 2	The operation is performed by summing 2 shifts of the Y value. The number of bits to be shifted during the first and second shifts are contained in the lower and upper halves of the appropriate CONVxP register, respectively. This adds a cycle to each conversion.
an arbitrary pitch (not a power of 2 and not two powers of 2)	The TMS34020 must perform a full 16-bit×32-bit multiply. In this case, the appropriate xPTCH register is used directly as the multiplier of the Y value. This adds about 12 cycles to each conversion.

The TMS34020 must perform one or more XY-to-linear conversions for the following instructions:

CVDXYL	FILL XY	PIXBLT L, XY
CVMXYL	FLINE	PIXBLT XY, L
CVSXYL	LINE	PIXBLT XY, XY
CVXYL	PIXBLT B, XY	PIXTs
DRAV		

The TMS34020 uses the pitch conversion factors in the CONVSP, CONVDP, and CONVMP registers to calculate the Y component of an address.

CONVSP (source pitch) is used if the XY address points to a source pixel or pixel array.

CONVDP (destination pitch) is used if the XY address points to a destination pixel or pixel array.

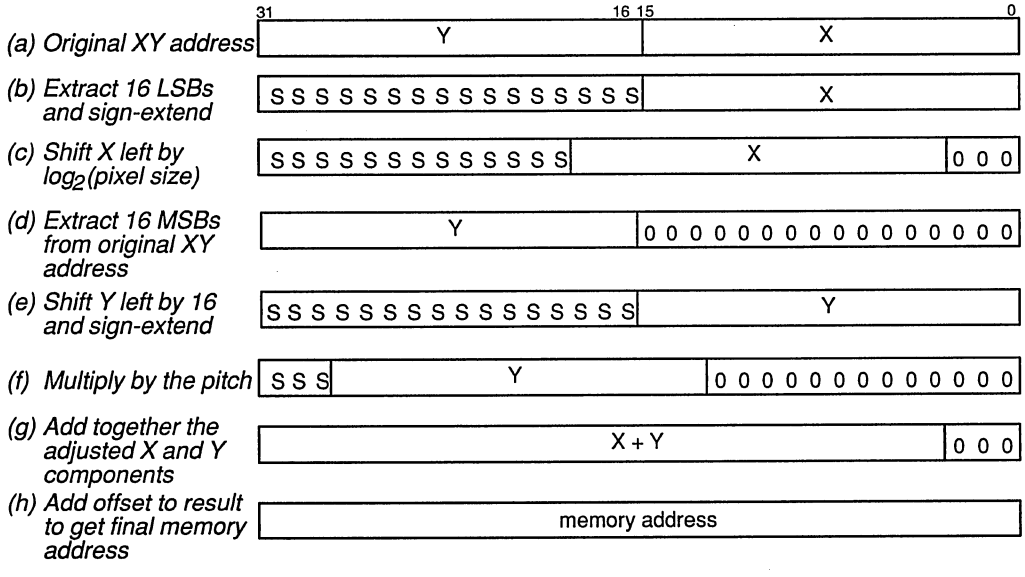
CONVMP (mask pitch) is used for calculating the correct value when using a binary mask array.

Before executing an instruction that uses XY addressing, use the SETCSP, SETCDP, or SETCMP instruction to load the value for the appropriate register.

The TMS34020 uses the PSIZE value to determine the displacement of the X component.

The OFFSET register contains the linear memory address of the pixel located at coordinates (0,0). The TMS34020 uses the OFFSET register when translating XY coordinates into linear addresses. (Note that OFFSET does not control which region of the display memory is output to refresh the video screen—it is a virtual screen origin.) This allows the coordinate axes of the XY address to be translated to an arbitrary position in memory. The OFFSET register supports the use of window-relative addressing in which the XY coordinates are specified relative to coordinate offsets in the display memory. The window's position and size can be specified arbitrarily. You can use the CVXYL instruction to convert a new XY offset to a linear address. CVXYL converts an XY address to a linear address for the purpose of absolute memory addressing, or for using special features available to instructions that use linear addressing. Figure 3–14 illustrates the XY-to- linear conversion process.

Figure 3–14. Conversion from XY Coordinates to Memory Address



Key: **S** represents the sign bit.

The example in Figure 3–14 corresponds to a pixel size of 8 bits and a pitch of 8,192.

(a) shows the original XY address.

(b) extracts the X component.

(c) shifts the X component left by $\log_2$(pixel size). The result represents the product of the X component and the pixel size.

(d) extracts the Y component.

(e) left rotates the Y component by 16, sign-extends the Y component.

(f) multiplies the shifted Y component by the pitch. (This may be a shift, the sum of 2 shifts, or an actual multiply depending on the pitch value. In this example, Y is shifted 12 bits to the left.)

(g) adds the results of step *(c)* and *(f)* to form the displacement in memory of the pixel at (X,Y) from the pixel at the origin.

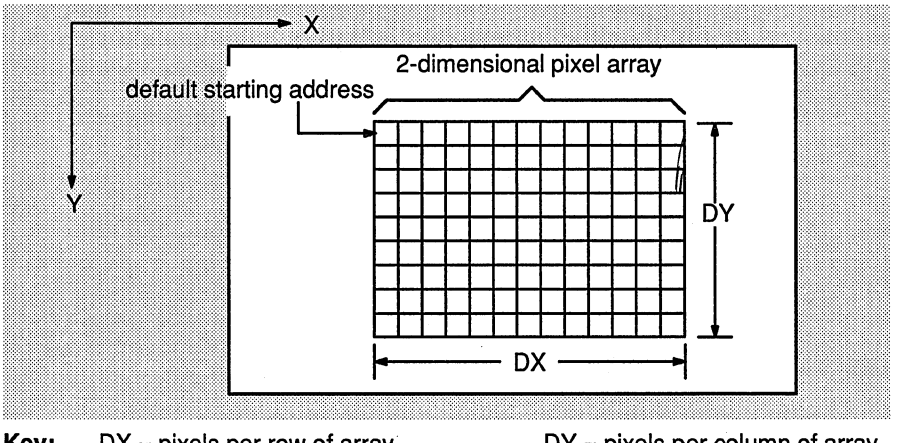
(h) adds the offset to the result of *(g)*, producing the the final memory address.

3.7 Pixel Arrays

A rectangular area of the screen that is DX pixels wide and DY pixels high is an example of a data structure called a 2-dimensional **pixel array**. An array may contain many pixels, but the TMS34020 can manipulate an array as a single structure. The TMS34020's instruction set includes a powerful set of raster operations, called PIXBLTs (pixel-block transfers), that manipulate pixel arrays on the screen and elsewhere in memory.

Figure 3–15 shows a pixel array that occupies a rectangular area in display memory. The pixels in each row are packed together into adjacent cells in the display memory. Rows don't usually occupy adjacent areas of memory; they're separated from each other by a constant displacement (the **array pitch**). The array pitch is the difference in memory addresses between 2 vertically adjacent pixels. In Figure 3–15, the array pitch equals the display pitch. The product of the array width (DX) and the pixel size must be less than or equal to the pitch.

Figure 3–15. Pixel Array



Key: DX = pixels per row of array DY = pixels per column of array

A pixel array is specified in terms of its *width*, *height*, *pitch*, and *starting address*. The starting address is usually the address of the first pixel to be moved during a PIXBLT. The default starting address is simply the base address in the array—that is, the address of the pixel with the lowest address in the array.

In Figure 3–15, the XY origin is located in its default position at the upper left corner of the screen. The default starting address is the address of the pixel located in the upper left corner of the array. When a PIXBLT operation moves the pixels from a source pixel array to a destination array, the pixels in each row are moved in sequence from left to right, and the rows are moved in sequence from top to bottom.

Certain PIXBLT operations allow the starting pixel to be specified as the pixel in one of the other three corners of the array. This feature is provided so that when the source and destination arrays overlap, the appropriate starting corner can be selected to insure that no data is lost by being overwritten during

PIXBLT execution. The order in which pixels in the array are moved can be altered to be from right to left and from bottom to top, as appropriate, to accommodate the change in the starting corner.

The starting address of a pixel array can be specified in terms of either the XY coordinates of the starting pixel (XY address), or the memory address of the starting pixel (linear address):

- An array whose starting location is specified as an XY address is referred to as an XY array. In this format, the starting location of the array is identified by the XY coordinates of the first pixel in the array.
- A pixel array whose starting location is specified as a memory address is referred to as a linear array. In this format, the location of the array is identified by the memory address of the first pixel in the array.

The XY array format has 2 advantages. First, the starting location of the array is given in system-independent Cartesian coordinates, rather than as a system-dependent memory address. Second, using XY addressing allows you to take advantage of the TMS34020's window checking facilities (which allows it to automatically detect an attempt to write a pixel inside or outside a defined area).

The linear format's main advantage is that it allows PIXBLTs to execute more quickly because it eliminates the need to translate from XY to linear format before accessing memory.

The general rules governing array pitch are

- When an array is specified in XY format, the pitch can be any multiple of the pixel size. However, PIXBLT operations performed on XY-format arrays are most efficient if the pitch is a power of 2.
- When an array is specified in linear format, the pitch must be a multiple of the pixel size. For the special case of a PIXBLT B,XY or PIXBLT B,L instruction, the source pitch may be any value. (Note that this corresponds to a pitch that is a multiple of the pixel size where the pixel size is 1.) This feature supports efficient use of memory by allowing adjacent rows of the source array to be packed together with no intervening gaps.

PIXBLTs are useful for moving arrays from one area of the screen to another; they can also be used to move arrays to the screen from other parts of memory, and vice versa. The pitch for the off-screen pixel array can be specified independently of the pitch for the on-screen array. This allows you to store off-screen data efficiently, regardless of the display pitch. On-screen objects can be defined as XY arrays but may be more efficiently stored as linear arrays in off-screen memory. The PIXBLT instructions support the transfer of a linear array to an XY array, and vice versa. PIXBLT instructions can also be used to rapidly move blocks of nonpixel data (for example, ASCII characters) from one memory location to another.

3.8 Big-Endian and Little-Endian Addressing

The TMS34020 allows you to address fields within memory in one of two ways—in **little-endian** or **big-endian mode.**

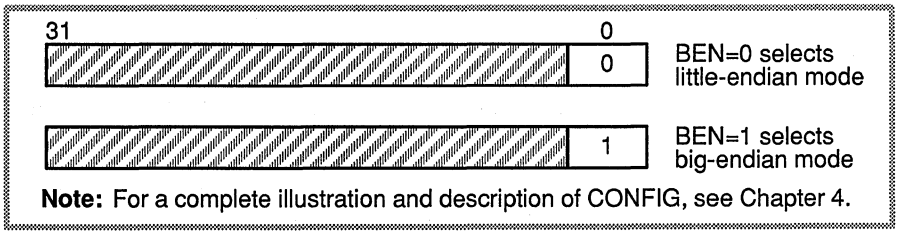
> **Note:**
>
> Unless specifically stated otherwise, all illustrations and discussions in this user's guide refer to little-endian mode.

The TMS34010 uses little-endian addressing only.

3.8.1 Selecting Big-Endian or Little-Endian Mode

The value of BEN[[CONFIG]] determines which endian mode the TMS34020 will use for addressing.

Figure 3–16. How BEN [[CONFIG]] Determines the Endian Mode



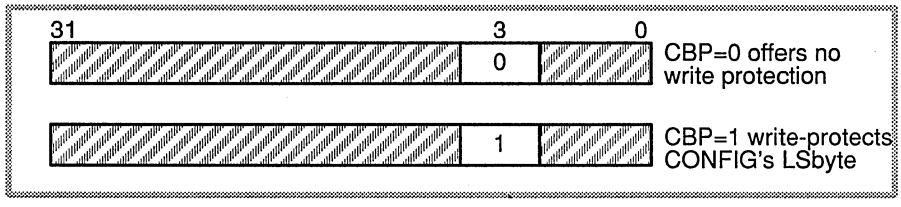
Note: For a complete illustration and description of CONFIG, see Chapter 4.

You can set BEN in one of two ways:

- By writing to CONFIG via the host interface. In this case, the TMS34020 should be halted.
- By resetting the TMS34020. At the end of the reset routine, the TMS34020 copies the 4 LSBs of the reset vector into the 4 LSBs of the CONFIG register.

Program code should not change bits 0—2 of the CONFIG register; this could cause unpredictable behavior. To ensure that code doesn't accidentally change these bits, you can set CBP[[CONFIG]] to write-protect the LSbyte of CONFIG.

Figure 3–17. How CBP [[CONFIG]] Write-Protects CONFIG's LSbyte



You can set CBP in one of three ways:

- By writing to CONFIG via the host interface.
- By resetting the TMS34020. At the end of the reset routine, the TMS34020 copies the 4 LSBs of the reset vector into the 4 LSBs of the CONFIG register.
- By allowing TMS34020 program code to write to this bit. If you do this, be sure that you don't alter bits 0—2 of CONFIG.

The only way to clear CBP is by resetting the TMS34020. CBP will remain cleared only if bit 3 of the reset vector is also 0 (if it's 1, then a reset will write a 1 back to CBP).

3.8.2 How the TMS34020 Accesses Memory in These Modes

The following descriptions uses several terms and conventions:

- The terms *least significant bit* (LSB) and *most significant bit* (MSB) define specific bits within a 32-bit long-word, or specific bits within a field. These terms refer to the *arithmetic significance* of these bits.
- The following illustrations show the MSB of a field or long-word on the left side. The bits at the ends of the word are numbered 0 or 31, implying that the bits within a long-word are numbered 0 through 31. The 0 and 31 are positioned so that the implied number associated with each bit is its appropriate bit address within the long-word. The manner in which the TMS34020 addresses these bits differs in the two modes.
- Note also that there are two frames of reference for data; illustrations show data in
  - memory or an I/O register or
  - a general-purpose register.

The TMS34020 is primarily a little-endian processor; its method for placing data in a general-purpose register reflects this.

Little-endian mode

This is the TMS34020's default mode. Figure 3–18 shows the same 32-bit hexadecimal value (01234 ABCDh) in a register and in a long-word in memory. Note that this illustration shows the MSB (bit 31) on the left side.

Figure 3–18. How Data Is Represented in Little-Endian Mode

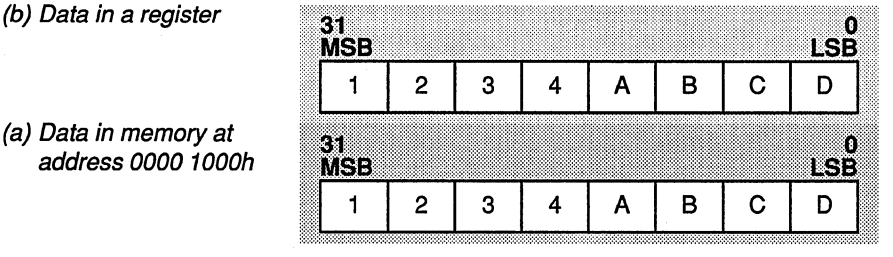
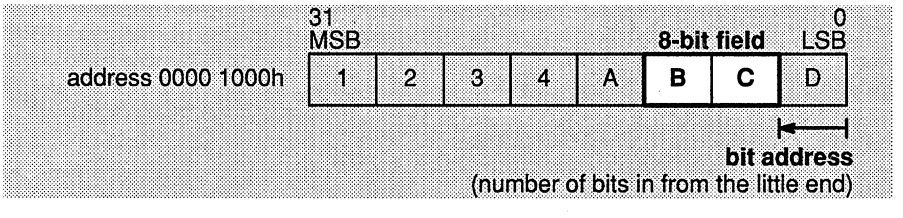


Figure 3–19 shows how the TMS34020 uses little-endian mode to access an 8-bit field that starts at bit 4 within the long-word. Notice that for little-endian mode, the field's bit address is determined by counting in from the LS (little) end of the long-word.

Figure 3–19. Addressing a Field in a Long-Word (Little-Endian)

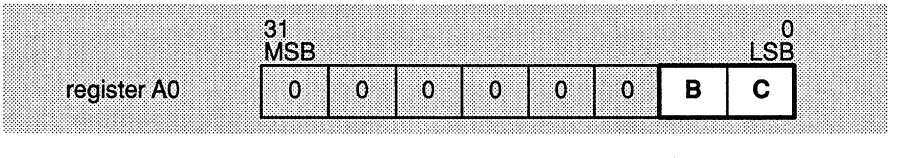


If you want to move this 8-bit data field into a general-purpose register, you might execute the following instructions:

```
SETF   8,0,0
MOVE   @00001004,A0,0
```

Figure 3–20 shows how the TMS34020 places this data into A0.

Figure 3–20. Moving a Field into a General-Purpose Register (Little-Endian)



Note that the data is right-aligned within the register, so that the LSB of the field coincides with the register's LSB.

Big-endian mode

Figure 3–21 shows the same 32-bit hexadecimal value (01234ABCDh) in a register and in a long-word in memory. This illustration shows the MSB on the left side. The register's MSB is bit 31; the long-word's MSB is bit 0.

Figure 3–21. How Data Is Represented in Big-Endian Mode

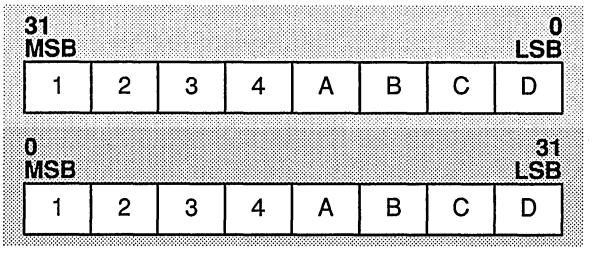
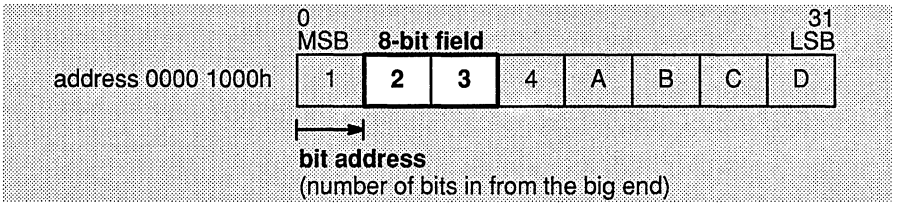


Figure 3–22 shows how the TMS34020 uses big-endian mode to access an 8-bit field that starts at bit 4 within the long-word. Notice that for big-endian mode, the field's bit address is determined by counting in from the MS (big) end of the long-word.

Figure 3–22. Addressing a Field in a Long-Word (Big-Endian)

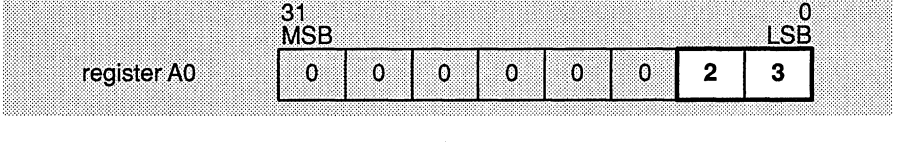


If you want to move this 8-bit data field into a general-purpose register, you might execute the following instructions:

```
SETF   8,0,0
MOVE   @00001004,A0,0
```

Figure 3–23 shows how the TMS34020 places this data into A0.

Figure 3–23. Moving a Field into a General-Purpose Register (Big-Endian)



Note that the data is right-aligned within the register, so that the LSB of the field coincides with the register's LSB.

Differences between big- and little-endian modes

- The TMS34020 accesses 32-bit-wide, 32-bit-aligned fields in the same manner for both modes. Differences between the two modes are apparent only when data is not 32 bits long-or when it is not aligned to a 32-bit boundary within memory.
- In both modes, data is right-aligned when it is moved into a general-purpose register.
- In big-endian mode, bits within a field or long-word are renumbered, not reordered.

3.8.3 Assembling Code for Big-Endian or Little-Endian Addressing

The TMS34020 assembler can produce object code for little-endian or big-endian mode. By default, the assembler produces little-endian code; if you want it to produce big-endian code, be sure to use the –e assembler option.

To assemble little-endian code	To assemble big-endian code
gspa *filename*	gspa **–e** *filename*

(For more information about the assembler, refer to the *TMS340 Family Code-Generation Tools User's Guide.*)

If you compare the listing files of big- and little-endian code, you'll find no differences in the listed object code. Figure 3–24 shows a listing file with object code. For ease of reading, the assembler lists object code in the same manner for both little- and big-endian code.

Figure 3–24. Sample Listing File (Assembler Output) for Little-Endian and Big-Endian Code

```
0001   00000000  aabbccdd        .long  0AABBCCDDh
0002   00000020      09e0        MOVI   11223344h, A0
       00000030  11223344
0003   00000050      0380        ABS    A0
0004   00000060      09C1        MOVI   6677h, A1
       00000070      6677
                      ↑
                 object code
```

Although the object code in the listing file looks the same for both modes, the assembler actually creates different object code for the two modes. If you're writing a loader, it's important to know how to load the object code into memory. Figure 3–25 demonstrates this.

Figure 3–25. Loading Object Code into Memory

(a) Little-endian mode

31			0
AA	BB	CC	DD
33	44	09	E0
03	80	11	22
66	77	09	C1

(b) Big-endian mode

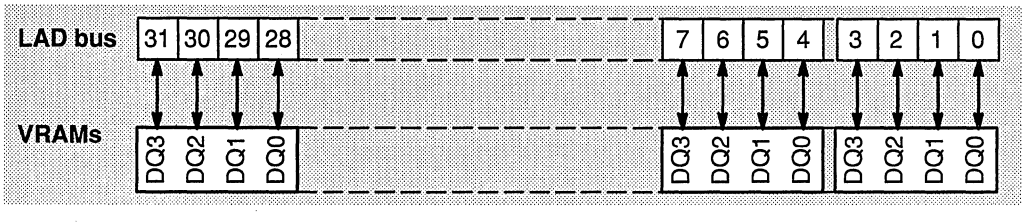
31			0
AA	BB	CC	DD
09	E0	33	44
11	22	03	80
09	C1	66	77

3.8.4 Wiring VRAMs to the LAD Bus

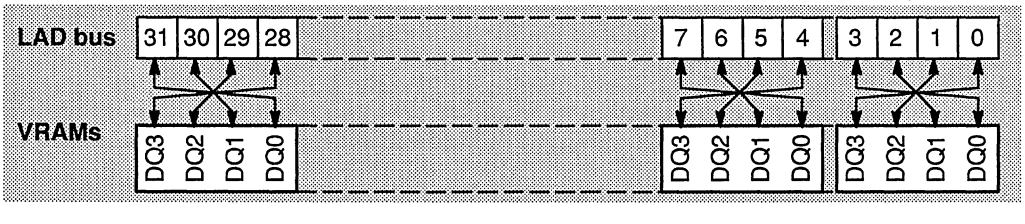
Figure 3–26 shows examples of how you might use TMS44C251 VRAMs in your system. A TMS44C251 has 4 bidirectional data pins, DQ0—DQ3; each data pin is connected to an LAD pin. Figure 3–26 *(a)* shows wiring for little-endian mode; as *(b)* shows, you must wire the VRAMs backwards for big-endian mode.

Figure 3–26. Connecting VRAMs to the LAD Bus

(a) Little-endian mode



(b) Big-endian mode



3.8.5 Big-Endian Effects on Instruction Timing

The instruction timings listed in this document are for little-endian code. Timings for big-endian code are essentially the same as timings for little-endian code; however, the setup for graphics instructions may consume extra machine states (instructions' inner loops consume no additional states). The effect on timing is slight.

3.9 Stacks

The TMS34020's system stack is implemented in local memory and managed in hardware. The stack is used to store return addresses and processor status information during interrupts, traps, and subroutine calls. The contents of general-purpose registers can be pushed onto the stack and popped off the stack. The system stack can also be used for dynamically allocated data storage.

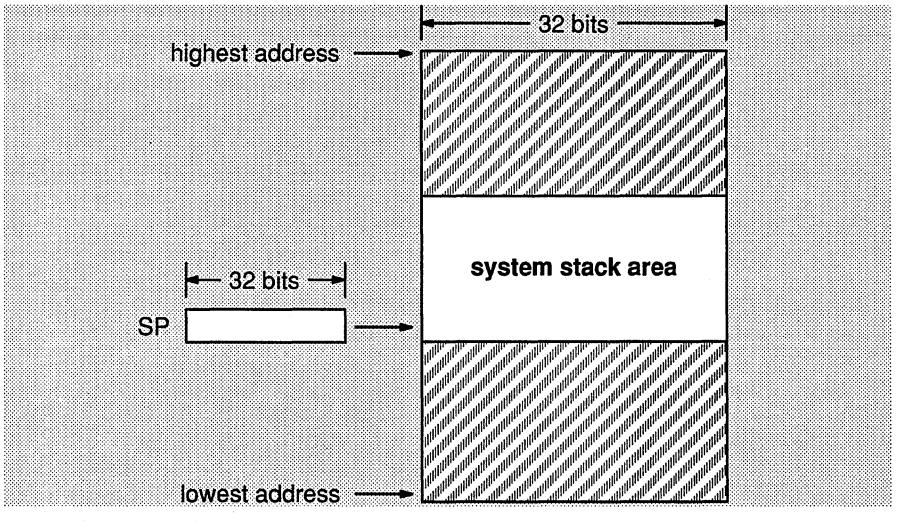
The stack is accessed through a dedicated 32-bit internal register, called the stack pointer, or **SP**. The SP points to the top of the system stack; it can be accessed as register 15 in either of the general-purpose register files, A or B.

In addition to the system stack, you can define your own auxiliary stacks. The system stack always grows toward lower memory addresses; an auxiliary stack can be defined to grow toward either lower or higher addresses. The MOVE instructions, combined with the predecrement and postincrement addressing modes, facilitate pushing and popping of auxiliary stack data. You can use one or more general-purpose registers as auxiliary stack pointers and frame pointers. The indexed addressing modes can be used in conjunction with a frame pointer to access variables embedded within the stack.

3.9.1 System Stack

Figure 3–27 shows the structure of the system stack, which grows in the direction of lower memory addresses.

Figure 3–27. System Stack



The SP points to the top of the stack; it contains the 32-bit address of the LSB (bit 0) of the value on top of the stack. The SP can contain any 32-bit address; however, stack operations execute more efficiently when the 5 LSBs of the SP

are 0s. This aligns the SP to long-word boundaries in memory, reducing the number of memory cycles needed to push or pop values.

Any instruction that manipulates general-purpose registers can also be used to manipulate the SP. The SP can be specified as the source or destination operand in any instruction that operates on the general-purpose registers. Instructions that manipulate the SP include:

Instructions That Push Values on the Stack	Instructions That Pop Values from the Stack
MMTM SP, *register list*	MMFM SP, *register list*
CALL *Rs*	RETI
CALLA *absolute address*	RETS
CALLR *relative address*	POPST
TRAP *number*	MOVE *SP+, *Rd*
PUSHST	
MOVE *Rs, –*SP*	

3.9.1.1 Saving Registers on the System Stack

Register information can be stored on the stack during an interrupt or a subroutine call. This frees up the register for use by an interrupt routine or a subroutine and allows you to restore the original register values from the stack when the routine completes.

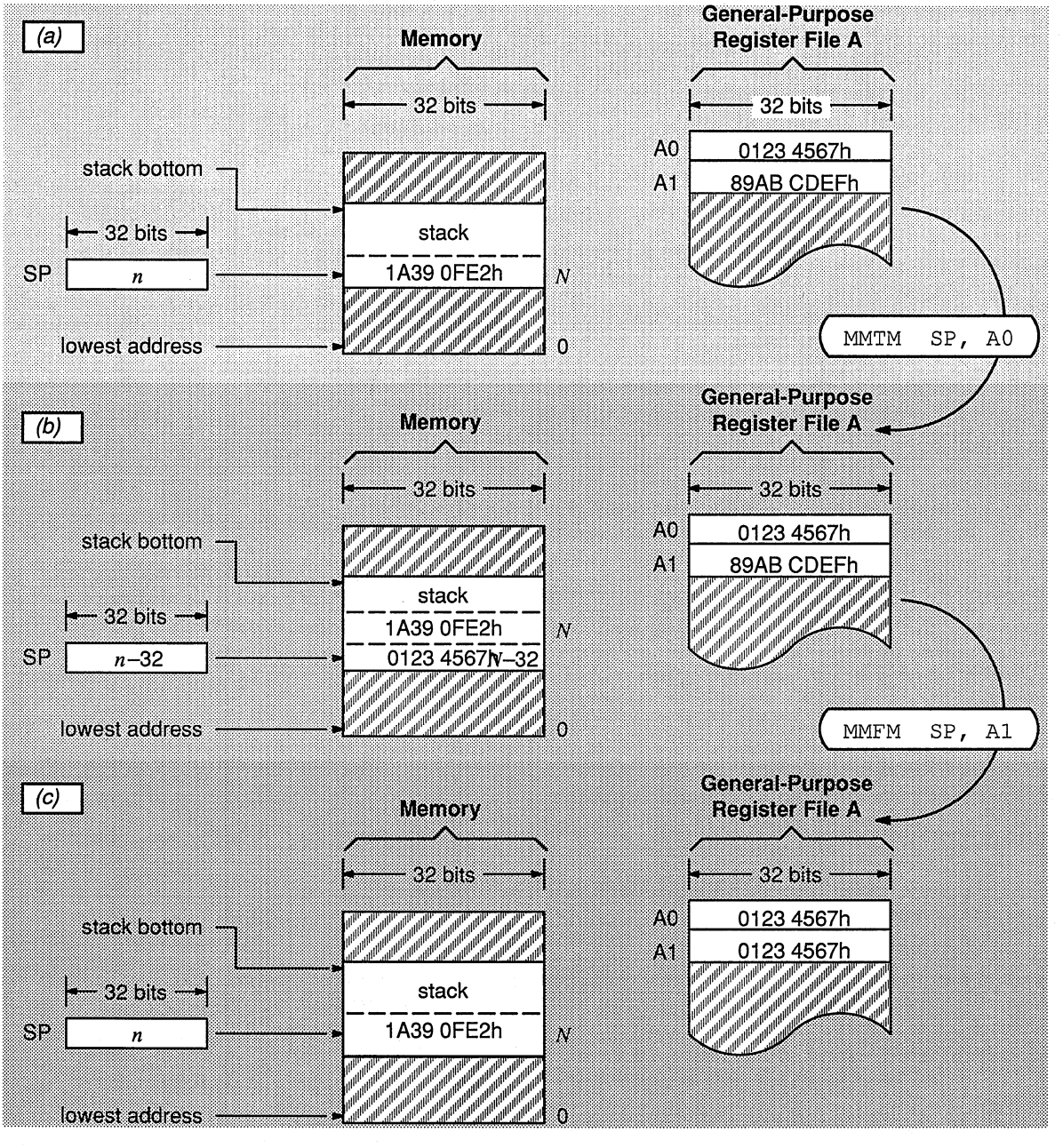
During an interrupt, the contents of the PC and ST are automatically saved on the stack; if you want to save values that are in general-purpose registers, you can use the MMTM and MMFM instructions. MMTM pushes multiple general-purpose registers onto the stack, and MMFM pops multiple general-purpose registers from the stack.

When the contents of a 32-bit register are pushed onto the stack, they are stored in the 32-bit word below the word whose address is contained in the SP. This is shown in Figure 3–28, which demonstrates the effects of the following instruction sequence:

```
MMTM  SP,A0 ; Push register A0 onto stack
MMFM  SP,A1 ; Pop stack into register A1
```

- Figure 3–28 *(a)* shows the original state of the stack and registers.
- Figure 3–28 *(b)* illustrates the state after A0 is pushed onto the stack.
- Figure 3–28 *(c)* shows the results of popping the top of the stack into A1.

Figure 3–28. Stack Operations



The TMS34020 performs 2 steps to push the contents of a 32-bit register onto the top of the stack:

1) Decrements the SP by 32.
2) Pushes the register contents onto the stack.

The TMS34020 performs 2 steps to pop the top of the stack into a 32-bit register:

1) Pops the 32 bits at the top of the stack into the register.
2) Increments the SP by 32.

3.9.1.2 Saving Information on the System Stack During an Interrupt

During an interrupt, the TMS34020 pushes the PC and ST onto the stack; this allows the interrupted routine to resume execution when the interrupt processing is completed. An interrupt routine performs the following actions:

1) Decrements the SP by 32.
2) Pushes the PC onto the stack.
3) Decrements the SP again by 32.
4) Pushes the ST onto the stack.

During a return from an interrupt

1) Pops the 32 bits at the top of the stack into the ST.
2) Increments the SP by 32.
3) Pops the 32 bits at the top of the stack into the PC.
4) Increments the SP again by 32.

3.9.1.3 Saving Information on the System Stack During a Subroutine Call

A subroutine call saves the state of the calling routine on the stack; this allows the routine to resume execution when the subroutine completes. A subroutine call performs the following actions:

1) Decrements the SP by 32.
2) Pushes the PC onto the stack.

During a return from a subroutine

3) Pops the 32 bits at the top of the stack into the PC.
4) Increments the SP by 32.

3.9.2 Auxiliary Stacks

Auxiliary stacks, which are typically used to contain dynamically allocated data storage, can be managed in software. You can use any A- or B-file register (except the SP) as the auxiliary stack pointer. For the purposes of discussion,

the symbol STK represents the auxiliary stack pointer. STK is a symbol that must be equated to one of the general-purpose registers; for example:

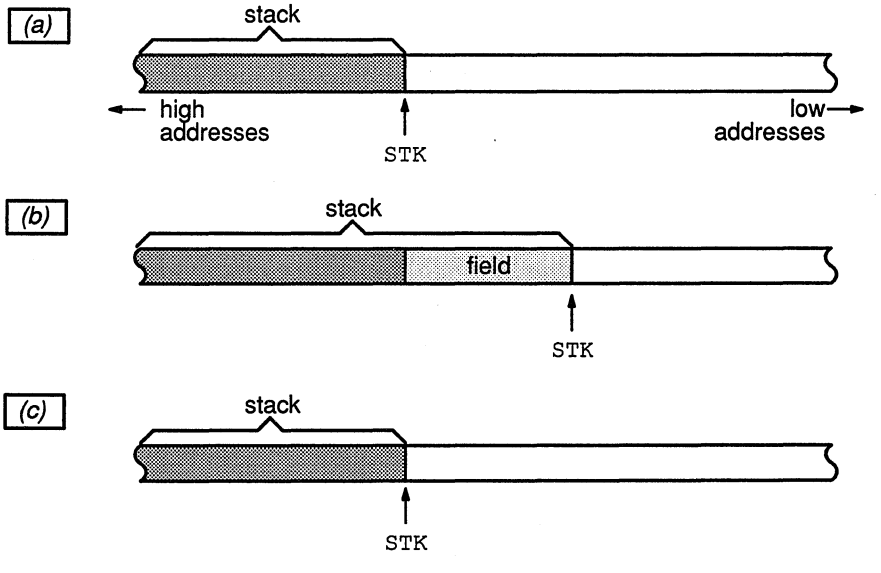
```
STK       .set      A0
```

STK can contain any 32-bit value; however, stack operations execute more efficiently when the 5 LSBs of the STK are 0s. This aligns the STK to long-word boundaries in memory, reducing the number of memory cycles needed to push or pop values.

As Figure 3–29 and Figure 3–30 show, an auxiliary stack can grow in either direction in memory. These figures represent memory as a string of continuously addressable bits.

Figure 3–29 shows a stack that grows toward lower memory addresses.

Figure 3–29. An Auxiliary Stack That Grows Toward Lower Addresses



- Figure 3–29 *(a)* shows the original stack.
- In Figure 3–29 *(b)*, a field of arbitrary size is pushed onto the stack with this instruction:

  MOVE *Rs*,*–STK

  (*Rs* and STK represent general-purpose registers and must be in the same register file.)
- In Figure 3–29 *(c)*, the field is popped off the stack with this instruction:

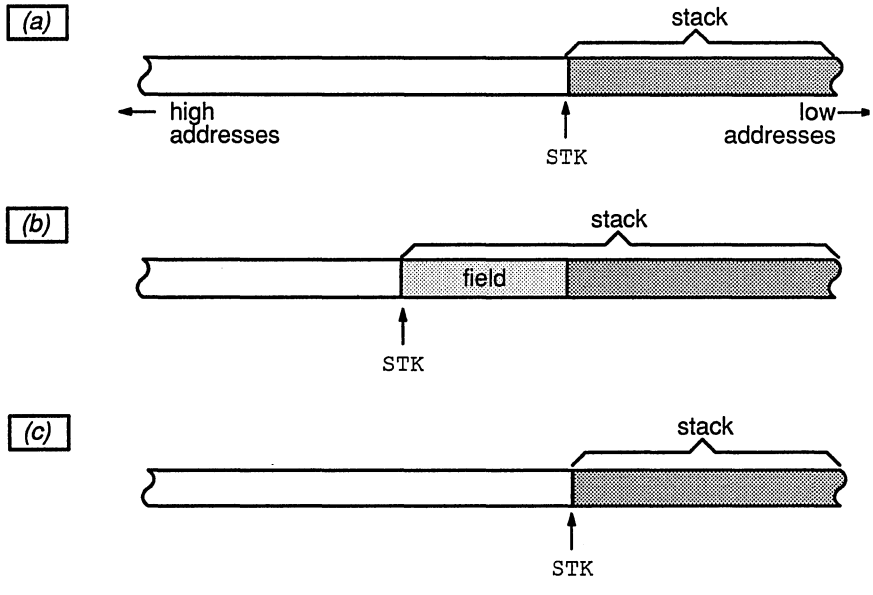
  MOVE *STK+,*Rd*

  (*Rd* and STK represent general-purpose registers and must be in the same register file.)

Between instructions, STK always points to the lowest bit address in the stack—this corresponds to the very top of the stack. You can use the MMTM STK,*register list* instruction to save multiple registers on the stack in Figure 3–29. Later, you can restore the registers to their former values with an MMFM STK,*register list* instruction.

Figure 3–30 shows a stack that grows toward higher memory addresses:

Figure 3–30. An Auxiliary Stack That Grows Toward Higher Addresses



- Figure 3–30 *(a)* shows the original stack.
- In Figure 3–30 *(b)*, a field of arbitrary size is pushed onto the stack using the following instruction:

  MOVE *Rs*,*STK+

- In Figure 3–30 *(c)*, the field is popped off the stack with this instruction:

  MOVE *–STK,Rd

Between instructions, the STK always points to one plus the highest bit address in the stack—this location is one bit beyond the very top of the stack.

Chapter 4

TMS34020 Registers

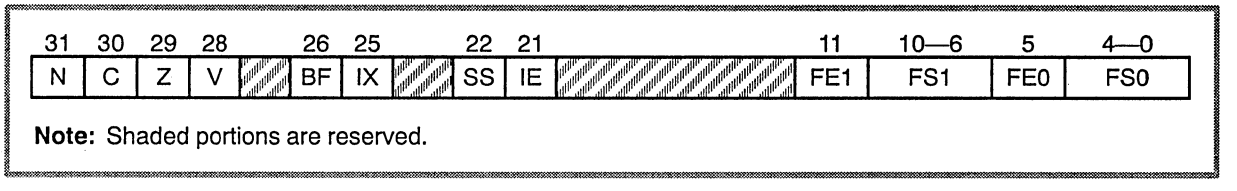
The TMS34020 has two on-chip general-purpose register files, file A and file B. Each register file contains fifteen 32-bit registers. The register files share a 32-bit hardware stack pointer (SP) that automatically manages the system stack during interrupts and subroutine calls. The TMS34020 also has 2 dedicated 32-bit registers—a program counter and a status register.

In addition to the CPU registers, the TMS34020 has 54 memory-mapped registers that are dedicated to I/O functions.

4.1 The Status Register (ST)

The status register (ST) is a special-purpose, 32-bit register that reflects the processor status. The ST also contains several parameters that define the characteristics of two programmable data types, fields 0 and 1. At reset, the TMS34020 initializes the ST to 0000 0010h. Figure 4–1 illustrates the status register. Table 4–1 lists the functions associated with the status bits.

Figure 4–1. Status Register

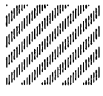


Note: Shaded portions are reserved.

Table 4–1. Definitions of Bits in the Status Register

Bit Number	Field Name	Function
0—4	FS0	**Field size 0:** Length in bits of the first memory data field. **FS0 = 00001_2—11111_2** defines a field size of 1—31 **FS0 = 00000_2** defines a field size of 32
5	FE0	**Field extension 0:** Determines whether the memory field is extended with 0s or with the sign bit when loaded into a 32-bit general-purpose register. **FE0 = 0** selects zero extension for field 0 **FE0 = 1** selects sign extension for field 0
6—10	FS1	**Field size 1:** Length in bits of the second memory data field. **FS1 = 00001_2—11111_2** defines a field size of 1—31 **FS1 = 00000_2** defines a field size of 32
11	FE1	**Field extension 1:** Determines whether the memory field is extended with 0s or with the sign bit when loaded into a 32-bit general-purpose register. **FE1 = 0** selects zero extension for field 1 **FE1 = 1** selects sign extension for field 1
21	IE	**Interrupt enable:** Master interrupt enable/disable bit. **IE = 0** disables all maskable interrupts **IE = 1** enables all maskable interrupts
22	SS	**Single step:** Setting the SS bit to 1 causes the TMS34020 to interrupt program execution following execution of each instruction. This is useful for debugging purposes.
25	IX	**Interruptible instruction executing:** When an interrupt occurs during instruction execution, the TMS34020 sets or clears the IX bit before saving the ST on the stack. **IX=0** indicates that an interrupt occurred at an instruction boundary **IX = 1** indicates that an interrupt occurred in the middle of an interruptible instruction

Table 4–1. Definitions of Bits in the Status Register (continued)

Bit Number	Field Name	Function
26	BF	**Bus fault:** Set when a bus fault occurs on a local-memory cycle.
28	V	**Overflow:** Set according to instruction execution.
29	Z	**Zero:** Set according to instruction execution.
30	C	**Carry:** Set according to instruction execution.
31	N	**Negative:** Set according to instruction execution.
12—20 23—24 27		**Reserved:** These bits are reserved; the TMS34020 does not use them. At reset, the TMS34020 clears these reserved bits to 0. **Note:** To maintain compatibility, you should write only 0s to these bits.

All instructions can potentially change the status register; during instruction execution, the TMS34020 may set the V, Z, C, and N bits. If you want to directly affect the ST, you can use the following instructions.

PUTST writes the contents of a specified general-purpose register into the status register. Here's an example:

```
MOVI  00000010h, A0
PUTST A0
```

GETST copies the contents of the ST into a specified general-purpose register.

SETC sets the C bit without altering any other status bits.

CLRC clears the C bit without altering any other status bits.

SETF writes values to the FS0 and FE0 or FS1 and FE1 bits without altering any other status bits.

EXGF exchanges the 6 LSBs of a specified general-purpose register with the FS0 and FE0 bits or with the FS1 and FE1 bits.

EINT sets the IE bit.

DINT clears the IE bit.

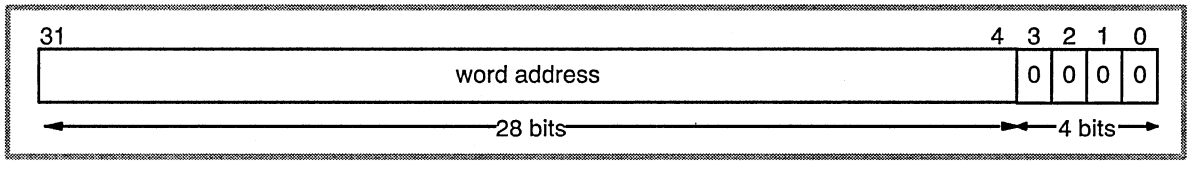
PUSHST pushes the contents of the ST onto the stack.

POPST pops the value at the top of the stack into the ST.

4.2 The Program Counter (PC)

The program counter (PC) is a special-purpose, 32-bit register that points to the next instruction word to be executed. Instructions are always aligned on 16-bit boundaries; thus, as Figure 4–2 shows, the PC's 4 LSBs always contain 0s.

Figure 4–2. Program Counter



An instruction consists of one or more 16-bit instruction words. The first word contains the opcode for the instruction; additional words may contain immediate data, displacements, or absolute addresses. As the TMS34020 fetches each 16-bit instruction word, it increments the PC to point to the next instruction word.

The PC contents are replaced during a branch instruction, subroutine call instruction, return instruction, or interrupt. As Table 4–2 shows, instructions can be categorized according to their effects on the PC.

Table 4–2. How Instruction Execution Affects the PC

Instruction Type	Effect on PC
No branch	The PC is incremented at the end of the instruction, allowing execution to proceed sequentially to the next instruction.
Absolute branch (TRAP, CALLA, JA*cc*)	The PC is loaded with an absolute address; the address' 4 LSBs are set to 0s.
Relative branch (CALLR, JR*cc*, DSJ*cc*)	The signed displacement (8 or 16 bits) is added to the PC's current contents. The signed displacement is treated as a word displacement; that is, it is shifted left 4 bit positions before it is added to the PC.
Indirect branch (JUMP, CALL)	The PC is loaded with the register contents. The 4 LSBs are set to 0s.

Two additional instructions provide you with direct control of the PC.

GETPC copies the contents of the PC into a specified general-purpose register.

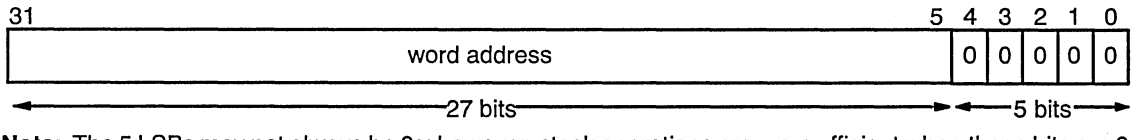
EXGPC exchanges the contents of the PC with the contents of a specified general-purpose register.

4.3 The Stack Pointer (SP)

The stack pointer (SP) is a special-purpose, 32-bit register that contains the bit address of the top of the system stack. *The TMS34020 contains only a single SP*; however, this SP can be addressed as a member of *either* register file, as register A15 or register B15. Any instruction that uses a general-purpose register as an operand can also use the SP as an operand.

Figure 4–3 illustrates the stack pointer; Section 3.9, Stacks, (page 3-26) describes stack operation in detail.

Figure 4–3. The Stack-Pointer Register



Note: The 5 LSBs may not always be 0s; however, stack operations are more efficient when these bits are 0s.

The system stack grows toward smaller addresses. The stack pointer always points to the value at the top of the stack. Specifically, the SP contains the 32-bit address of the LSB of that value. Although the SP's 5 LSBs can have any arbitrary value, stack operations execute more efficiently when the 5 LSBs are 0. Clearing these bits to 0s aligns the stack pointer on a 32-bit word boundary; thus, only a single memory access (two cycles) is necessary to push or pop the contents of a 32-bit register.

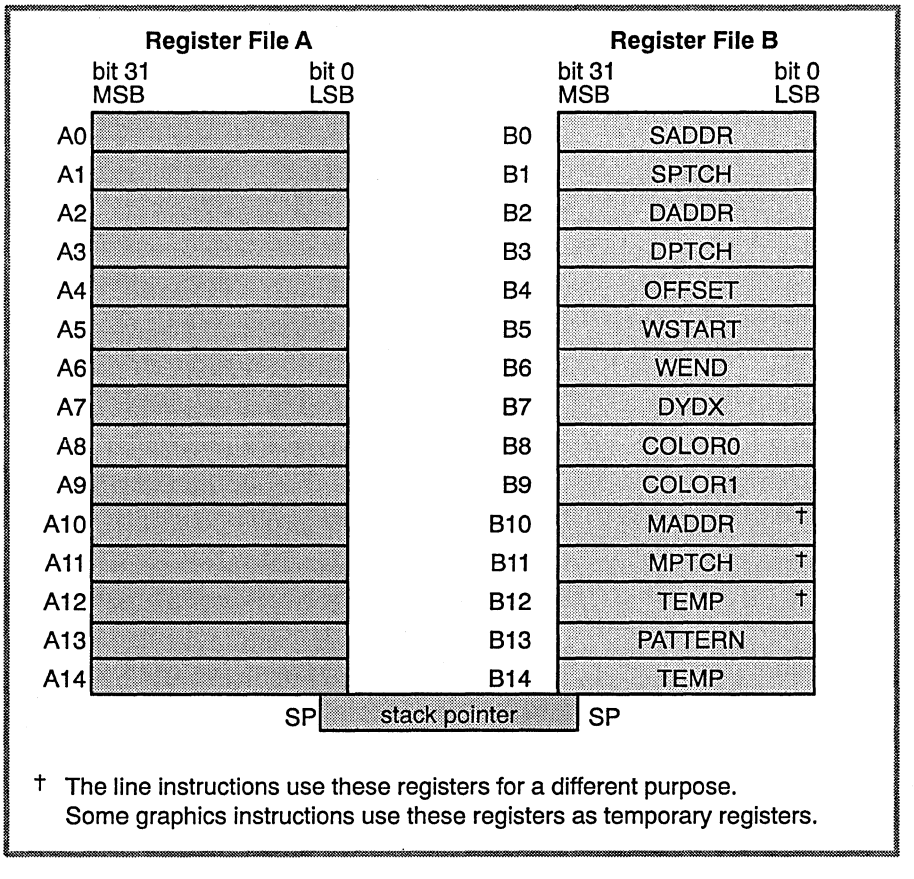
During subroutine calls and interrupts, the PC and ST are pushed onto the stack. These are both 32-bit registers. (If the SP is not long-word aligned when instruction execution is interrupted, the TMS34020 aligns the stack before saving the PC and ST.) The MMTM and MMFM instructions push/pop multiple 32-bit registers from the A or B file.

For the most efficient execution, you should ensure that the SP is always aligned to a long word and that it is incremented or decremented in multiples of 32 bits.

4.4 General-Purpose Registers (Register Files A and B)

The TMS34020 has thirty 32-bit general-purpose registers, divided into register files A and B. The register files share a single stack pointer (SP). Figure 4–4 illustrates the register files (note the shared SP).

Figure 4–4. The Register Files



As Figure 4–4 shows, 15 of the general-purpose registers, A0—A14, form register file A. Register file B also consists of 15 general-purpose registers, B0—B14. Many of the TMS34020 instructions use these registers for storing and manipulating data.

The TMS34020's register files have several advantages:

- The general-purpose registers are dual-ported. This allows the TMS34020 to read from or write to two separate registers at the same time.
- Several instructions use general-purpose registers to contain source and destination operands; these are called **register-to-register instructions.**

Multiple internal data paths link the ALU to the general-purpose registers, allowing the TMS34020 to execute most register-to-register instructions in a single machine state. Single-state instructions include add, subtract, Boolean operations, and shifts (1 to 32 bits).

During a single-state instruction, the following actions may occur:

- The TMS34020 reads, in parallel, two 32-bit operands from the general-purpose registers.
- The ALU performs the specified operation.
- The 32-bit result is stored in the specified general-purpose register.

All register-to-register instructions (except MOVE *Rs*, *Rd*) require both registers to be in the same file. Instructions that can use registers A0—A14 and B0—B14 as operands can also use the SP as an operand.

Note:

For some graphics operations, the B-file registers have hardware-dedicated functions. When their special functions are used, the contents of the B-file registers are referred to as **implied graphics operands**. Several I/O registers also contain implied operands.

No hardware-dedicated functions are associated with the A-file registers; generally, instructions do not use the A-file registers as implied operands.

Table 4–3 (page 4-8) summarizes the names and functions associated with the B-file registers when they are used as implied operands.

Table 4–3. Summary of B-File Registers' Implied-Operand Functions

Register	Function	Description
B0	SADDR	**Source address.** Address (linear or XY) of a source pixel array; usually the address of the array's upper left corner (the lowest pixel address in the array).
B1	SPTCH	**Source pitch.** Difference in start addresses (linear) between adjacent rows of a source pixel array.
B2	DADDR	**Destination address.** Address (linear or XY) of a destination pixel array; usually the address of the array's upper left corner (the lowest pixel address in the array).
B3	DPTCH	**Destination pitch.** Difference in start addresses (linear) between adjacent rows of a destination pixel array.
B4	OFFSET	**Offset.** Linear bit address, corresponds to the XY origin (X=0, Y=0).
B5	WSTART	**Window start address.** XY address of the upper left corner of the window (smallest X and Y coordinate values in the window).
B6	WEND	**Window end address.** XY address of the lower right corner of the window (largest X and Y coordinate values in the window).
B7	DYDX	**Delta Y/delta X.** The 16 LSBs of DYDX define the width (X dimension) of a pixel array. The 16 MSBs define the height (Y dimension) of a pixel array.
B8	COLOR0	**Background pixel color.** COLOR0 contains the background color for graphics operations.
B9	COLOR1	**Foreground pixel color.** COLOR1 contains the foreground color for graphics operations.
B10	MADDR	**Mask address.** Address of the upper left corner of a mask pixel array (lowest pixel address in the array).
	COUNT	**Loop counter.** LINE & FLINE instructions use B10 to count the number of pixels drawn within the line.
	TEMP	**Temporary register.**
B11	MPTCH	**Mask pitch.** Difference in start addresses (linear) between adjacent rows of a mask array.
	INC1	**Diagonal increment.** LINE & FLINE use INC1 to identify the amount by which a pixel address is incremented in the diagonal direction.
B12	INC2	**Dominant increment.** LINE & FLINE use INC2 to identify the amount by which a pixel address is incremented in the dominant direction.
	TEMP	**Temporary register.**
B13	PATTERN	**Array or line pattern.** The 1s and 0s within PATTERN identify a pixel pattern for an array or a line.
B14	TEMP	**Temporary register.**

Note: Some graphics instructions use the TEMP (temporary) registers to store temporary values and context information during instruction execution.

4.5 I/O Registers

The TMS34020 supports a set of I/O registers that control and monitor

- communications between the TMS34020 and a host processor,
- the TMS34020's interface to local memory,
- interrupts,
- video timing and screen refreshing, and
- graphics-drawing operations.

The I/O registers reside in the TMS34020's on-chip memory, occupying addresses C000 0000h—C000 03FFh. Figure 4–5 shows this.

Figure 4–5. I/O Register Memory Map

	Most Significant Half	Least Significant Half	
HESYNC	C000 0010	C000 0000	VESYNC
HEBLNK	C000 0030	C000 0020	VEBLNK
HSBLNK	C000 0050	C000 0040	VSBLNK
HTOTAL	C000 0070	C000 0060	VTOTAL
DPYSTRT	C000 0090	C000 0080	DPYCTL
CONTROL	C000 00B0	C000 00A0	DPYINT
HSTADRL	C000 00D0	C000 00C0	HSTDATA
HSTCTLL	C000 00F0	C000 00E0	HSTADRH
INTENB	C000 0110	C000 0100	HSTCTLH
CONVSP	C000 0130	C000 0120	INTPEND
PSIZE	C000 0150	C000 0140	CONVDP
PMASKH	C000 0170	C000 0160	PMASKL
CONTROL	C000 0190	C000 0180	CONVMP
DPYTAP	C000 01B0	C000 01A0	CONFIG
HCOUNT	C000 01D0	C000 01C0	VCOUNT
REFADR	C000 01F0	C000 01E0	DPYADR
DPYSTH	C000 0210	C000 0200	DPYSTL
DPYNXH	C000 0230	C000 0220	DPYNXL
DINCH	C000 0250	C000 0240	DINCL
HESERR	C000 0270	C000 0260	reserved
reserved	C000 0290	C000 0280	reserved
reserved	C000 02B0	C000 02A0	reserved
BSFLTST	C000 02D0	C000 02C0	SCOUNT
reserved	C000 02F0	C000 02E0	DPYMSK
SETHCNT	C000 0310	C000 0300	SETVCNT
BSFLTDH	C000 0330	C000 0320	BSFLTDL
reserved	C000 0350	C000 0340	reserved
reserved	C000 0370	C000 0360	reserved
IHOST1H	C000 0390	C000 0380	IHOST1L
IHOST2H	C000 03B0	C000 03A0	IHOST2L
IHOST3H	C000 03D0	C000 03C0	IHOST3L
IHOST4H	C000 03F0	C000 03E0	IHOST4L

The TMS34020 can access these registers directly; a host processor can access them through the TMS34020's host interface. I/O registers are accessed like any other memory location. Table 4–4 summarizes the I/O registers and their functions.

Table 4–4. Summary of I/O Registers

Register	Address	Description
BSFLTDL BSFLTDH	C000 0320h C000 0330h	**Bus-fault data.** When a bus fault occurs, the TMS34020 stores the current LAD data in the BSFLTD registers.
BSFLTST	C000 2D0h	**Bus-fault status.** When a bus fault occurs, the TMS34020's memory controller saves its current state into BSFLTST.
CONFIG	C000 01A0h	**System configuration.** Contains several parameters that enable VRAM register loads and control little-/big-endian addressing, row-column address configuration, and refresh rates.
CONTROL	C000 00B0h C000 0190h	**Memory control.** Controls transparency, window checking, PIXBLT direction, and cache operation.
CONVDP	C000 0140h	**Destination pitch conversion factor.** Contains the XY-to-linear factor for converting a destination array address.
CONVMP	C000 0180h	**Mask pitch conversion factor.** Contains the XY-to-linear factor for converting a mask array address.
CONVSP	C000 0130h	**Source pitch conversion factor.** Contains the XY-to-linear factor for converting a source array address.
DINCL DINCH	C000 0240h C000 0250h	**Display increment.** Contains the difference in addresses between vertically adjacent pixels (the *display pitch*).
DPYADR	C000 01E0h	**Display address.** Provides TMS34010 compatibility.
DPYCTL	C000 0080h	**Display control.** Controls video timing parameters.
DPYINT	C000 00A0h	**Display interrupt.** Identifies the next scan line at which a display interrupt will be requested.
DPYNXL DPYNXH	C000 0220h C000 0230h	**Display next address.** The DPYNX registers contain a 32-bit address.
DPYMSK	C000 02E0h	**Display mask.** When screen refreshes are enabled, DPYMSK defines which bits of the address in the DPYNX and DPYST registers correspond to the tap-point portion of the address output during screen-refresh cycles.
DPYSTL DPYSTH	C000 0200h C000 0210h	**Display start address.** Points to the pixel at the left of the 1st line displayed on the screen.
DPYSTRT	C000 0090h	**Display start address.** Provides TMS34010 compatibility.
DPYTAP	C000 01B0h	**Display tap point address.** Provides TMS34010 compatibility.
HCOUNT	C000 01D0h	**Horizontal count.** Tracks the number of VCLKs per horizontal scan line.
HEBLNK	C000 0030h	**Horizontal end blank.** Defines the point at which the horizontal blanking interval ends.
HESERR	C000 0270h	**Horizontal end serration.** Defines the point at which the composite sync pulse ends during the serration region of vertical blanking.
HESYNC	C000 0010h	**Horizontal end sync.** Defines the point at which the horizontal sync pulse ends.
HSTADRH HSTADRL	C000 00E0h C000 00D0h	**Host interface address.** Provides TMS34010 compatibility.

Table 4–4. Summary of I/O Registers (continued)

Register	Address	Description
HSTCTLH	C000 0100h	**Host interface control, high word.** Controls host-interface functions such as halt acknowledge, software reset, the nonmaskable interrupt, host autoincrements and prefetches, and halting TMS34020 execution.
HSTCTLL	C000 00F0h	**Host interface control, low word.** Controls host-interface functions such as messages, emulator control, and bus-fault and retry information.
HSTDATA	C000 00C0h	**Host interface data.** Provides TMS34010 compatibility.
HSBLNK	C000 0050h	**Horizontal start blank.** Defines the point at which the horizontal blanking interval begins.
HTOTAL	C000 0070h	**Horizontal total.** Defines the duration of each horizontal scan line (in terms of VCLK periods).
IHOST	C000 0380h through C000 03F0h	**Internal host interface address.** The host interface uses these 32-bit locations.
INTENB	C000 0110h	**Interrupt enable.** Assuming that the status IE bit = 1, setting specific bits to 1 enables external interrupts 1 & 2, the host interrupt, the display interrupt, or the window-violation interrupt.
INTPEND	C000 0120h	**Interrupt pending.** The values of specific bits indicate whether an external interrupt, host interrupt, display interrupt, or window-violation interrupt has been requested but not yet serviced.
PMASKL PMASKH	C000 0160h C000 0170h	**Plane mask.** The PMASK registers form a 32-bit value that selectively enables/disables individual planes in a multiple-bit-per-pixel display system.
PSIZE	C000 0150h	**Pixel size.** Defines the pixel size (in bits). Valid pixel sizes include 1, 2, 4, 8, 16, and 32.
REFADR	C000 01F0h	**Refresh pseudo-address.** Contains the address output during DRAM-refresh cycles.
SCOUNT	C000 02C0h	**Shift clock counter.** Incremented during the active display time so that it always contains the tap point of the bit most recently shifted out of the VRAM serial registers.
SETHCNT	C000 0310h	**Set horizontal count.** When external video is enabled, SETHCNT contains the value that is loaded into HCOUNT.
SETVCNT	C000 0300h	**Set vertical count.** When external video is enabled, SETVCNT contains the value that is loaded into VCOUNT.
VCOUNT	C000 01C0h	**Vertical count.** Counts the horizontal lines in the video display, incrementing on the same clock edge that resets HCOUNT to 0.
VEBLNK	C000 0020h	**Vertical end blank.** Defines the time at which the vertical blanking interval ends.
VESYNC	C000 0000h	**Vertical end sync.** Defines the time at which the vertical sync pulse ends.
VSBLNK	C000 0040h	**Vertical start blank.** Defines the time at which the vertical blanking interval begins.
VTOTAL	C000 0060h	**Vertical total.** Defines the time at which the vertical sync pulse begins.

4.5.1 CPU Control Registers

CONTROL	PSIZE	CONVSP
CONVDP	CONVMP	

These 5 registers provide CPU control. They allow you to select those TMS34020 characteristics that meet your specific system needs, such as the pitches for pixel transfers, window-checking modes, transparency modes, Boolean or arithmetic pixel-processing options, PIXBLT direction, and pixel size.

4.5.2 Host Communications Registers

HSTCTLH	HSTCTLL

These registers provide a host processor with the ability to interrupt or halt the TMS34020, flush the instruction cache, communicate with an emulator, and select modes for accessing TMS34020 local memory.

4.5.3 Local-Memory and DRAM/VRAM Interface Registers

CONFIG	PMASK	REFADR
BSFLTD	BSFLTST	

The memory controller manages the TMS34020's interface to the local memory, automatically performing the bit alignment and the masking necessary to access data located at arbitrary bit boundaries within memory.

4.5.4 Interrupt Registers

INTENB	INTPEND

These registers control and monitor interrupt requests to the TMS34020, including 2 externally generated interrupts and 3 internally generated interrupts, including

- External interrupts 1 and 2
- Window-violation interrupt
- Host interrupt
- Display interrupt

If the IE status bit (global interrupt enable) = 1, you can set a bit in the INTENB register to enable any of these interrupts. You can check bits in the INTPEND register to see if any of these interrupts are pending.

4.5.5 Video Timing and Screen-Refresh Registers

Twenty-eight registers are dedicated to video timing and screen-refresh functions. The TMS34020 can drive composite sync or separate sync displays. Parameters in the DPYCTL register allow you to select the direction (input/output) of the sync signals:

Composite Sync Mode		Separate Sync Mode	
Signal	**Direction**	**Signal**	**Direction**
VSYNC	I/O	VSYNC	I/O
HSYNC	I/O	HSYNC	I/O
CSYNC	I/O	HBLNK	O
CBLNK	O	VBLNK	O

In composite mode, the TMS34020 can extract $\overline{\text{VSYNC}}$ and $\overline{\text{HSYNC}}$ from an external composite sync, or it can generate $\overline{\text{CSYNC}}$ from separate $\overline{\text{VSYNC}}$ and HSYNC inputs. Internally, you can set the TMS34020 to preset the horizontal and vertical counts upon receiving an external sync signal. This allows compensation for any combination of internal and external delays that occur in the video synchronization process.

- An external $\overline{\text{HSYNC}}$ loads HCOUNT from SETHCNT.
- An external $\overline{\text{VSYNC}}$ loads VCOUNT.
- An external $\overline{\text{CSYNC}}$ loads both HCOUNT and VCOUNT from SETHCNT and SETVCNT, respectively.

The TMS34020 directly supports multiport VRAMs by generating the serial-register transfer cycles that are necessary for refreshing a display. The memory locations that contain the display information, as well as the number of horizontal scan lines displayed between serial-register transfer cycles, are programmable.

4.5.6 Latency of Writes to I/O Registers

The TMS34020 has a high degree of internal parallelism; for example, it can fetch instructions and data while still executing the current instruction. Normally this is beneficial.This could cause problems, however, if the current instruction alters an I/O register and the next instruction uses that register as an implied operand. In this situation, the second instruction may not execute properly. This could occur, for example, if a PIXBLT followed a MOVE instruction that modified the CONTROL register.

You can easily avoid this situation by ensuring that the write to the I/O register completes before any subsequent instructions use the modified register value. To do this, follow the write to the register with an MWAIT instruction.

4.6 Alphabetical Summary of I/O Registers and B-File Registers

The remainder of this chapter contains an alphabetical reference of the I/O and B-file registers. Some I/O registers contain implied operands for graphics instructions; the B-file registers also contain implied graphics operands. Therefore, the B-file registers and I/O registers are summarized together in this section.

Here's an important point: Although you'll use both B-file and I/O registers as implied operands, you must access them differently. Because the I/O registers are memory mapped, they are accessed similarly to external memory locations.

The code segment below shows a sample implied-operand setup for a FILL L instruction. It shows that you must use a different MOVE instruction for loading an I/O register than you would use for loading a B-file register. Note that most programs refer to registers by their symbolic names (such as DADDR or PSIZE, assuming you've equated these names to the actual register name or location).

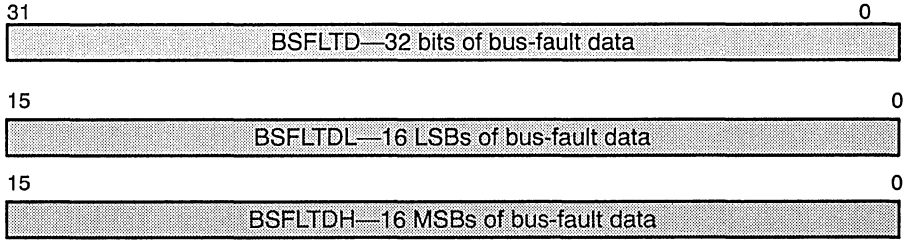
```
* Set up the B-file registers
      MOVI  0050h, B2              ; DADDR
      MOVI  0100h, B3              ; DPTCH
      MOVI  000050008h, B7         ; DYDX

* Set up the I/O registers
      MOVK  4, A0
      MOVE  A0, @0C0000150h, 0     ; PSIZE
      CLR      A0
      MOVE  A0, @0C0000160h, 1     ; PMASK
      MOVE  A0, @0C00000B0h, 0     ; CONTROL
      MWAIT    ; wait until data has been written

      FILL  L
```

B-file register?		register number:
I/O register?	√	*BSFLTD* **(32-bit address): C000 0320h** *BSFLTDL* **(16-bit address): C000 0320h** *BSFLTDH* **(16-bit address): C000 0330h**

Format



Note:

You can access the bus-fault registers separately or together by using different addresses and different field sizes.

- To access BSFLTD as a single 32-bit register, access the 32-bit field at address C000 0320h.
- To access BSFLTDL as a 16-bit register, access the 16-bit field at address C000 0320h.
- To access BSFLTDH as a 16-bit register, access the 16-bit field at address C000 0330h.

Description

When a bus fault occurs, the TMS34020's memory controller uses BSFLTD to store the data on the LAD bus. When any CPU-initiated memory access returns a bus-fault completion code on the LRDY and BUSFLT pins, the memory controller

Step 1: Saves the data currently stored on LAD0—LAD31 into BSFLTD.

Step 2: Signals the CPU that a bus fault occurred. The CPU

- pushes the current machine state onto the stack,
- executes a bus fault interrupt routine to clear the cause of the bus fault, and
- pops the machine state off the system stack to restore the CPU to its previous state.

Step 3: Uses the data in the BSFLTD registers to restore LAD0—LAD31 and re-executes the memory access that caused the bus fault. If the faulted memory access was a read, the data saved in and restored

from the BSFLTD registers has no significance. However, the memory controller saves and restores the LAD data, regardless of whether the faulted memory access was a read or a write.

Do not write to these registers. When a bus fault occurs, the saved LAD data writes over any data in the BSFLTD registers. If necessary, you can read the contents of the BSFLTD registers during your bus fault interrupt routine.

Note:

Although BSFLTDL and BSFLTDH are I/O registers, they are not loaded by a memory write when a bus fault occurs. If external memory shadows these locations, the BSFLTD registers are not copied to external memory.

B-file register?		register number:
I/O register?	√	**address: C000 02D0h**

Format

Description

When a bus-fault occurs, the TMS34020's memory controller saves its current state into the BSFLTST register. The status information tells the memory controller what type of access triggered the bus fault and marks the point within the access where execution can resume.

When any CPU-initiated memory access returns a bus-fault completion code on the LRDY and BUSFLT pins, the memory controller

Step 1: Saves its current state into BSFLTST.

Step 2: Signals to the main processor that a bus fault occurred.

Step 3: After the CPU clears the cause of the bus fault and restores its internal state, the memory controller restores its pre-bus-fault state from the BSFLTST register and re-executes the memory access that caused the bus fault.

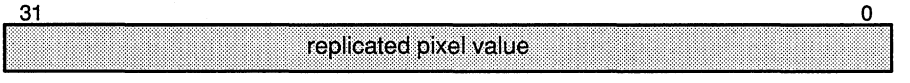
Usually, you should not write to the BSFLTST register. When a bus fault occurs, the saved memory controller state writes over any data in the BSFLTST register. If you do not want the TMS34020 to re-execute the faulted memory access, your bus-fault interrupt routine should write the value $FFFF_{16}$ to the BSFLTST register. This causes the memory controller to return from the bus fault in an idle state.

Note:

Although BSFLTST is an I/O register, it is not loaded by a memory write when a bus fault occurs. If external memory shadows this location, the BSFLTST register is not copied to external memory.

B-file register?	√	register number: B8
I/O register?		address:

Format



Description

COLOR0 provides a background color, defining the replacement color for 0 bits in a binary source array or in the PATTERN register. Pixel alignment within COLOR0 corresponds directly to alterable pixels within memory; individual pixels within COLOR0 are used as they align with pixels in the destination word.

Binary PIXBLTs use color information in COLOR0 and COLOR1 to transform a binary pixel array into a multiple-bits-per-pixel array.

Note:

You must replicate the color information throughout all 32 bits of COLOR0.

Execution of graphics instructions does not modify COLOR0.

Which instructions use this register?

Instruction	COLOR0's function
FLINE, LINE	Replaces 0s in the PATTERN value
FPIXEQ, FPIXNE	Comparison value
PFILL XY	Replaces 0s in the PATTERN value
PIXBLT B, L	Background pixel color for color-expanded array
PIXBLT B, XY	Background pixel color for color-expanded array

Example

This example is for 4-bit pixels. A pixel value of 5 is replicated throughout the COLOR0 register.

```
COLOR0    .set  B8

   MOVI  55555555h, COLOR0 ; store uniform pixel
         ; value in COLOR0
```

B-file register?	√	**register number: B9**	
I/O register?		address:	

Format

Description

COLOR1 provides a foreground color, defining the replacement color for 1 bits in a binary source array. Pixel alignment within COLOR1 corresponds directly to alterable pixels within memory; individual pixels within COLOR1 are used as they align with pixels in the destination word.

Binary PIXBLTs use color information in COLOR0 and COLOR1 to transform a binary pixel array into a multiple-bits-per-pixel array. Other graphics instructions use COLOR1 as the replacement color for an alterable destination pixel or for alterable pixels within a pixel block.

Note:

You must replicate the color information throughout all 32 bits of COLOR1.

Execution of graphics instructions does not modify COLOR1.

Which instructions use this register?

Instruction	COLOR1's function
DRAV	Pixel color for pixel draw
FILLs (both)	Pixel color for filled array
FLINE, LINE	Replaces 1s in the PATTERN value
PFILL XY	Replaces 1s in the PATTERN value
PIXBLT B, L	Foreground pixel color for color-expanded array
PIXBLT B, XY	Foreground pixel color for color-expanded array
TFILL	Pixel color for drawing
VLCOL	Color-fill data value for VRAM color registers

Example

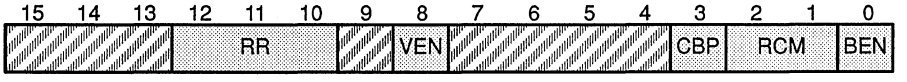
This example is for 4-bit pixels. A pixel value of 3 is replicated throughout the COLOR1 register.

```
COLOR1    .set  B9

   MOVI   33333333h, COLOR1 ; Store uniform pixel
          ; value in COLOR1
```

B-file register?		register number:
I/O register?	√	**address: C000 01A0h**

Format



Bits

Bits	Name	Function
0	BEN	Enables big-endian memory addressing
1—2	RCM	Configures RCA bus address
3	CBP	Enables configuration byte protect
8	VEN	Enables VRAM internal register load
12—10	RR	Selects refresh rate
4—7, 9, 13—15		Reserved; do not use

Description

CONFIG controls several system parameters: it selects the memory addressing configuration, informs the TMS34020 that the system contains VRAMs with color-latch and write-mask registers, and selects the DRAM-refresh rate.

Note:

Future pin-compatible TMS340x0 devices may use bit 4, providing you with the ability to extend the Q4 phase of certain memory subcycles. This will ease interfacing to DRAMs if the TMS340x0's LCLK frequency is increased above 10 MHz. To ensure compatibility with your existing TMS34020 system, set bit 4 to 1. Setting this bit will not affect the TMS34020.

Before almost any system activity can take place, you must select appropriate values for CONFIG's 3 LSBs. BEN and RCM affect memory addressing; until BEN and RCM have appropriate values, the TMS34020 can successfully access only 32-bit words at memory addresses that have row addresses of all 1s or all 0s.

- If the TMS34020 is not powered up in host-present mode, it reads the reset vector from address FFFF FFE0h. Then, before fetching any instructions, the TMS34020 writes the 4 LSBs of the reset vector to the 4 LSBs of CONFIG; this defines the system's memory addressing configuration. You should program the BEN and RCM values into the 3 LSBs of the reset vector; program bit 3 of the reset vector to set the CBP bit. Because the reset vector's row address is all 1s, the TMS34020 can successfully read the

reset vector, regardless of the BEN and RCM values. The TMS34020 assumes that the reset vector is aligned to a 16-bit word, so the values in the reset vector's 4 LSBs do not affect the location from which the TMS34020 starts fetching instructions.

- If the TMS34020 is powered up in host-present mode, the host must set BEN and RCM before accessing the TMS34020's local memory.

BEN
bit 0

Big-endian memory addressing enable

BEN	Effect
0	Selects little-endian addressing (default)
1	Selects big-endian addressing

The TMS34020 can use either little- or big-endian addressing conventions. Little-endian is the default (BEN=0). To use big-endian memory addressing, set BEN to 1. For more information about these addressing modes, refer to Section 3.8, Big-Endian and Little-Endian Addressing, on page 3-20.

RCM0—RCM1
bits 1 & 2

RCA bus configuration mode

RCM1	RCM0	Base Array Size (CAMD=0)	Logical Address Bits Output on RCA0—RCA12 at Row-Address Time
0	0	64K×*n*	24 to 12
0	1	256K×*n*	25 to 13
1	0	1M×*n*	26 to 14
1	1	4M×*n*	27 to 15

The RCM bits determine which bits of the logical address are output on RCA0—RCA12 at row-address time. Additionally, the CAMD pin allows the address output at column-address time to be modified on a cycle-by-cycle basis. These capabilities allow you to directly wire DRAMs and VRAMs of more than 1 of the above sizes to the RCA bus in the same system, without using external multiplexing logic. If CAMD is set high during a cycle, most of the bits in the column address are shifted left by 1 bit. However, the logical address bits output on RCA0, RCA11, and RCA12 are not determined by a shift, and vary according to the value of the RCM bits.

RCM1	RCM0	Logical Address Bits Output on RCA0—RCA12 at Column-Address Time with CAMD=1
0	0	23, 22, 13, 12, 11, 10, 9, 8, 7, 6, 5, S, S
0	1	26, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, S, S
1	0	15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, S, S
1	1	28, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, S, 16

Key: **S** is the 16-bit word select.

For more information about RCA0—RCA12 address multiplexing, refer to Section 8.16.2 on page 8-53.

CBP
bit 3

Configuration byte protect

CBP	Effect
0	LSbyte of CONFIG is not write-protected
1	Write-protects the LSbyte of CONFIG

Setting CBP to 1 write-protects the LSbyte of CONFIG (bits 0 to 7). You can set CBP by writing to it or by placing a 1 in bit 3 of the reset vector. During a reset, the TMS34020 automatically copies the 4 LSBs of the reset vector to the 4 LSBs of CONFIG. To clear CBP, reset the TMS34020 with a hardware reset or write a 1 to RST[[HSTCTLH]].

The BEN and RCM bits are extremely important system configuration bits; accidentally writing to them could cause your system to malfunction. When CBP=1, bits 0 to 7 of CONFIG remain write-protected until a reset occurs. Reset is the only operation that can clear CBP. When CBP is set to 1, the byte is write-protected from the next machine state.

VEN
bit 8

VRAM internal register load enable

VEN	Effect
0	Enables VRAM write-mask load and write with mask
1	Disables VRAM write-mask load and write with mask

The TMS34020 instructions and memory interface support VRAMs with internal write-mask and color registers (such as the TMS44251). Use VEN to inform the TMS34020 that your system's VRAMs support these features.

The VEN bit does not enable or disable execution of VBLT or VFILL instructions. Don't use these instructions if your system's VRAMs do not support the block-write feature.

If VEN=1 and any bit in the PMASK is written, the TMS34020 automatically executes a special load-write-mask memory cycle to load the 1s complement of the 32-bit plane mask into the VRAMs' write masks. This cycle is performed in the next available memory cycle. No further CPU-initiated memory cycles are executed until after the write mask is loaded.

If the TMS34020 subsequently performs a VFILL, VBLT, or pixel write, the plane mask≠ 0, and VEN=1, the TMS34020 automatically generates special block-write-with-mask and write-with-mask cycles. This allows selected planes within each pixel to be written without the need for read-modify-write cycles.

RR0—RR2
bits 10—12

Refresh rate

RR2	RR1	RR0	Refreshes scheduled every . . .
0	0	0	8 machine states
0	0	1	16 machine states
0	1	0	32 machine states
0	1	1	64 machine states
1	0	0	128 machine states
1	0	1	256 machine states
1	1	0	undefined
1	1	1	DRAM refresh disabled

The RR bits determine the frequency of DRAM refreshes. An internal counter schedules a DRAM-refresh request at the frequency determined by RR. Each time a DRAM refresh is scheduled, the TMS34020 increments another internal counter to track the number of pending refreshes. *(Pending DRAM refreshes* are refreshes that are requested but not yet performed.) Each time a DRAM refresh is performed, the refresh pending counter is decremented. Note that if a retry terminates a DRAM-refresh cycle, the pending count is not decremented, and the refresh is retried.

A maximum of 15 DRAM refreshes can be pending. If more refreshes are requested, the pending counter overflows and the 16 refreshes are lost (15 pending, plus the refresh that caused the overflow). However, 12 or more pending DRAM refreshes cause DRAM-refresh memory cycles to become one of the highest priority memory cycles, so losing the refreshes should never happen. Realistically, even 12 DRAM refreshes pending are unlikely, because 4 or more DRAM refreshes pending are a higher priority than CPU-initiated memory accesses.

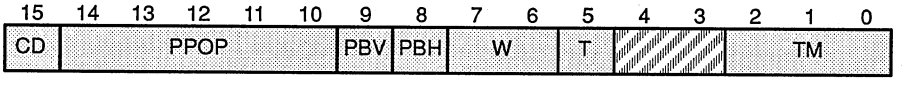
Which instructions use this register?

Any write to the PMASK register while VEN=1 causes the TMS34020 to load the VRAM write-mask registers with the 1s complement of PMASK.

B-file register? ☐ register number:

I/O register? √ **address: C000 00B0h** *and* **C000 0190h**

Format



Bits

Bits	Name	Function
0—2	TM	Selects a transparency mode
5	T	Enables transparency
6—7	W	Selects a window-checking mode
8	PBH	Selects PIXBLT horizontal direction
9	PBV	Selects PIXBLT vertical direction
10—14	PPOP	Selects a pixel-processing operation
15	CD	Disables cache
3—4		Reserved; do not use

Description

The CONTROL register controls several aspects of CPU instruction execution and of the memory interface. You can access this register at two addresses: at address C000 00B0h for compatibility with the TMS34010, and at C000 0190h so that all I/O registers used as implied operands are in a contiguous block (C000 0130h to C000 0190h). If you write to one location, both are affected.

TM
bits 0—2

Transparency mode select

TM2	TM1	TM0	Description
0	0	0	Transparency on result equal 0
0	0	1	Transparency on source equal COLOR0
1	0	0	Transparency on result equal 0
1	0	1	Transparency on destination equal COLOR0
0	1	0	Reserved
0	1	1	
1	1	0	
1	1	1	

When transparency is enabled, the TM bits select the transparency mode. You can enable transparency by setting the T bit to 1.

T
bit 5

Pixel transparency enable

T	Description
0	Disables transparency
1	Enables transparency

The T bit enables or disables pixel transparency. When transparency is enabled, the TMS34020 inhibits overwriting of a transparent pixel (as determined by the current transparency mode).

W
bits 6&7

Window checking

W1	W0	Description
0	0	No pixel writes are inhibited, and no interrupt requests are generated
0	1	Generate interrupt request on attempt to write to pixel lying inside window and inhibit all pixel writes
1	0	Generate interrupt request on attempt to write to pixel lying outside window
1	1	Inhibit pixel writes outside window, but do not request interrupt

The W bits select the action the TMS34020 takes when a pixel operation would cause the TMS34020 to write a pixel to a location lying either inside or outside specified window limits. Window checking applies to attempts to write to pixel locations defined by XY addresses only. Window checking affects neither non-pixel data writes nor writes to pixel locations defined by linear memory addresses.

A request for a window violation interrupt can occur when $W=01_2$ or $W=10_2$. WVP[[INTPEND]] is set to 1 to indicate that a window violation occurred. This in turn interrupts the TMS34020, if both WVE[[INTENB]] and IE[[ST]] equal 1.

PBH
bit 8

PIXBLT horizontal direction

PBH	Description
0	Increment in the X direction (move from left to right)
1	Decrement in the X direction (move from right to left)

The PBH bit determines the horizontal direction (increasing or decreasing X) of pixel processing for these instructions:

PIXBLT XY,XY
PIXBLT XY, L
PIXBLT L, M, L
PIXBLT L, XY
PIXBLT L, L

PBV
bit 9

PIXBLT vertical direction

PBV	Description (assuming default screen origin)
0	Increment in the Y direction (move from top to bottom)
1	Decrement in the Y direction (move from bottom to top)

The PBV bit determines the vertical direction (increasing or decreasing Y) of pixel processing for these instructions:

PIXBLT XY,XY
PIXBLT XY, L
PIXBLT L, M, L
PIXBLT L, XY
PIXBLT L, L

PPOP
bits 10—14

Pixel processing operation

The PPOP bits define the manner in which a source pixel is combined with a destination pixel during a pixel operation. The following 16 PPOP codes perform Boolean operations on pixels of 1, 2, 4, 8, 16, and 32 bits.

PPOP: 4	3	2	1	0	Operation	Description
0	0	0	0	0	S→D	Source replaces destination
0	0	0	0	1	S AND D→D	AND source with destination
0	0	0	1	0	S AND $\overline{D}$→D	AND source with NOT(destination)
0	0	0	1	1	0→D	0s replace destination
0	0	1	0	0	S OR $\overline{D}$→D	OR source with NOT(destination)
0	0	1	0	1	S XNOR D→D	XNOR source with destination
0	0	1	1	0	$\overline{D}$→D	Invert destination
0	0	1	1	1	S NOR D→D	NOR source with destination
0	1	0	0	0	S OR D→D	OR source with destination
0	1	0	0	1	D→D	Do not change destination (note, however, that memory cycles still occur)
0	1	0	1	0	S XOR D→D	XOR source with destination
0	1	0	1	1	$\overline{S}$ AND D→D	AND NOT(source) with destination
0	1	1	0	0	1→D	1s replace destination
0	1	1	0	1	$\overline{S}$ OR D→D	OR NOT(source) with destination
0	1	1	1	0	S NAND D→D	NAND the source and destination
0	1	1	1	1	$\overline{S}$→D	NOT (source) replaces destination

These PPOP codes perform arithmetic operations on 2, 4, 8, 16, and 32-bit pixels (but not on 1-bit pixels).

PPOP:	4	3	2	1	0	Operation	Description
	1	0	0	0	0	S + D→D	Add source to destination
	1	0	0	0	1	S ADDS D→D	Add source to destination with saturation
	1	0	0	1	0	D – S→D	Subtract source from destination
	1	0	0	1	1	D SUBS S→D	Subtract source from destination with saturation
	1	0	1	0	0	S MAX D→D	Replace destination with maximum of source and destination
	1	0	1	0	1	S MIN D→D	Replace destination with minimum of source and destination

Note: PPOP codes 10110_2 through 11111_2 are reserved.

CD

bit 15

Cache disable

CD	Description
0	Enables instruction cache
1	Disables instruction cache

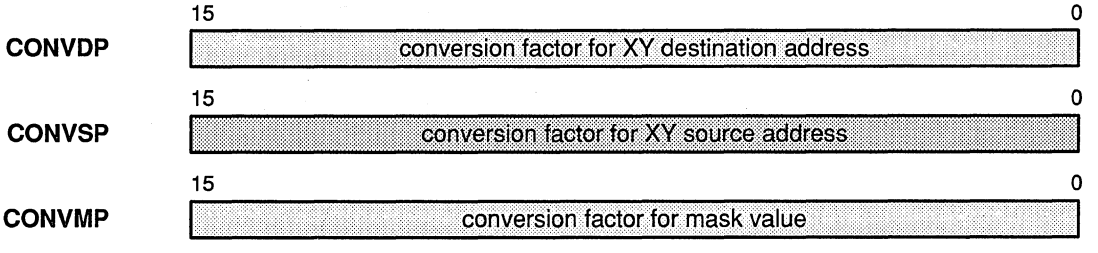
CD enables or disables the instruction cache. When the cache is disabled, cache contents (including data, P flags, SSA registers, etc.) are not disturbed, and all instructions are fetched from memory, not the cache. When the cache is re-enabled, its previous state (before it was disabled) is restored, and the instructions retained within the cache are once again available for execution.

Which instructions use this register?

Instruction	Bits used			
all instructions	CD			
DRAV	PPOP	T & TM	W	
FILL L	PPOP	T & TM		
FILL XY	PPOP	T & TM	W	
FLINE	PPOP	T & TM		
LINE	PPOP	T & TM	W	
PIXBLT B, L	PPOP	T & TM		
PIXBLT B, XY	PPOP	T & TM	W	
PIXBLT L, L	PPOP	T & TM		PBH & PBV
PIXBLT L, XY	PPOP	T & TM	W	PBH & PBV
PIXBLT XY, XY	PPOP	T & TM	W	PBH & PBV
PIXBLT XY, L	PPOP	T & TM		PBH & PBV
PIXT *Rs*, **Rd*	PPOP	T & TM		
PIXT **Rs*, **Rd*	PPOP	T & TM		
PIXT *Rs*, **Rd*.**XY**	PPOP	T & TM	W	
TFILL	PPOP	T & TM	W	

B-file register?	register number:
I/O register? √	*CONVDP* **address: C000 0140h** *CONVSP* **address: C000 0130h** *CONVMP* **address: C000 0180h**

Format



Description

CONVDP, CONVSP, and CONVMP are 16-bit registers that contain control parameters used during execution of a pixel operation. The TMS34020 uses CONVDP and CONVSP with

- XY addressing,
- window clipping, and
- FILLs or PIXBLTs (except for PIXBLT L,L) that process pixels from the bottom of the array to the top (PBV=1).

The TMS34020 uses CONVMP for XY addressing (CVMXYL).

Each conversion factor register is associated with an appropriate pitch register; each CONVxP register associated with an instruction that loads the conversion factor into CONVxP according to the pitch value in xPTCH.

Conversion Factor Register (CONVxP)	Associated Pitch Register (xPTCH)	Associated Instruction
CONVDP	DPTCH	SETCDP
CONVSP	SPTCH	SETCSP
CONVMP	MPTCH	SETCMP

TMS34020 internal hardware uses the CONVDP and CONVSP values when converts an XY destination or source address, respectively, to a linear address.

- PIXBLT and FILL instructions with an XY destination use DPTCH and CONVDP to convert the XY coordinates to a linear address before the pixel transfer begins.
- PIXBLT instructions with an XY source address use the SPTCH and CONVSP values to convert the XY coordinates to a linear memory address before beginning the pixel transfer.

If a PIXBLT or FILL requires preclipping of the destination array in the Y direction, the TMS34020 uses CONVDP to calculate the effect of the clipped starting Y coordinate on the destination array's starting linear address. For PIXBLTs, the starting source address is modified to accommodate the resulting changes to the starting destination address. When a PIXBLT instruction's starting Y coordinate lies in either of the 2 lower corners of the destination array (PBV=1), the TMS34020 uses CONVDP and CONVSP to calculate the linear addresses corresponding to the specified starting coordinates.

CONVxP contains 1 of 3 types of values, depending on the value in the associated pitch register:

If the xPTCH register =	**then. . .**
a power of 2	The 5 LSBs of CONVxP contain the 1s complement of $\log_2$(xPTCH). During XY-to-linear conversion, the product of the Y value and the pitch is calculated by shifting Y left by $\log_2$(xPTCH).
two powers of 2	CONVxP contains 2 conversion values. The 5 LSBs of CONVxP should contain the 1s complement of $\log_2$ of the greater of the powers of 2, and the 5 LSBs of the upper byte contain the 1s complement of $\log_2$ of the lesser of the powers of 2. During conversion, the product of the Y value and the pitch is calculated by adding Y shifted left by each of the 2 conversion factors.
arbitrary pitch	The LSbyte of CONVxP contains 0s. The TMS34020 must multiply the address by xPTCH. This is a 16-by-32-bit signed multiply in which only the 32 LSBs of the result are retained.

Which instructions use these registers?

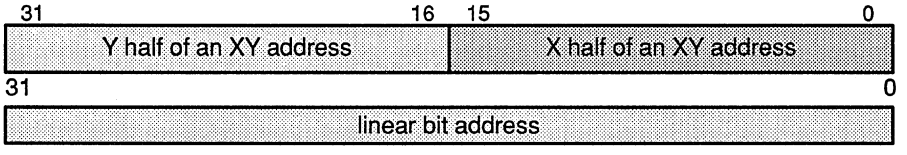
CONVDP	**CONVSP**	**CONVMP**
CVXYL	CVSXYL	CVMXYL
CVDXYL	PIXBLT L, XY	SETCMP
DRAV	PIXBLT XY, L	
FILL XY	PIXBLT XY, XY	
FLINE, LINE	PIXT **Rs*.XY*, *Rd*	
PIXBLT B, XY	PIXT **Rs*.XY*, **Rd*.XY**	
PIXBLT L, XY	SETCSP	
PIXBLT XY, L		
PIXBLT XY, XY		
PIXT *Rs*, **Rd*.XY**		
PIXT **Rs*.XY*, **Rd*.XY**		
SETCDP		
TFILL		

For more information about array pitches and XY-to-linear conversion, refer to Section 12.12, Converting an XY Address to a Linear Address, on page 12-47.

B-file register? √ **register number: B2**

I/O register? ☐ address:

Format



Description

DADDR contains the destination array address for PIXBLTs, FILLs, LINE, and FLINE. DADDR usually points to the pixel with the lowest address in the destination array. When the selected starting corner is not the upper left corner, the TMS34020 automatically adjusts DADDR to point to the selected starting corner of the destination array. (For PIXBLT L,L, however, you must manually adjust DADDR to point to the starting corner.)

Some instructions use DADDR with DYDX to perform a common rectangle function (FILL XY, PFILL XY, PIXBLT B,XY, PIXBLT L,XY, and PIXBLT XY,XY, with window option 1). In these cases, the TMS34020 sets DADDR to the starting XY address of the rectangle that represents the intersection of the original destination array and the clipping window. No drawing is performed. If the array and the window do not intersect, the V bit is not set and the contents of DADDR are undefined.

The TMS34020 treats the address in DADDR as an XY address or a linear address, depending on the instruction you use.

If DADDR contains an XY address, the instruction converts it to the corresponding linear address before beginning the pixel transfer. During a PIXBLT or FILL, DADDR is maintained in linear format. When the instruction completes, DADDR points to the linear starting address of the row following the last row in the array (for LINE, FLINE, and VFILL, DADDR contains the address of the next point on the line). If a PIXBLT is interrupted, DADDR points to the next word of pixels to be read.

Which instructions use this register?

Instruction	DADDR's format and function
BLMOVE	Linear; points to the beginning of the destination array
CLIP	XY; points to the beginning of the destination array
FILL L	Linear; points to the beginning of the destination array
FILL XY	XY; points to the beginning of the destination array
FLINE	Linear; starting point for the line
LINE	XY; starting point for the line
PFILL XY	XY; points to the beginning of the destination array
PIXBLT B, L	Linear; points to the beginning of the destination array
PIXBLT B, XY	XY; points to the beginning of the destination array

Instruction	DADDR's format and function
PIXBLT L, L	Linear with special requirements when PBH=1 or PBV=1; refer to PIXBLT L,L for a description of its unique requirements
PIXBLT L, XY	XY; points to the beginning of the destination array
PIXBLT XY, L	Linear; points to the beginning of the destination array
PIXBLT XY, XY	XY; points to the beginning of the destination array
PIXBLT L,M,L	Linear; points to the beginning of the destination array
TFILL	XY; instruction uses this to hold temporary values
VBLT	Linear; points to the beginning of the destination array
VFILL	Linear; points to the beginning of the destination array

Example

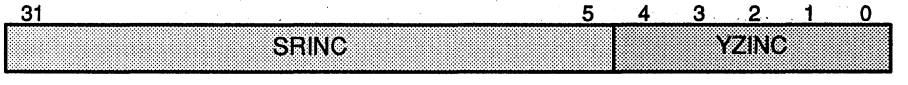
```
DADDR   .set   B2

        MOVI   [0008h, 0015h], DADDR  ; Move XY value
                                      ; 15h,8h into B2
        MOVI   00010AFCh, DADDR       ; Move linear
                                      ; value 10AFCh
                                      ; into B2
```

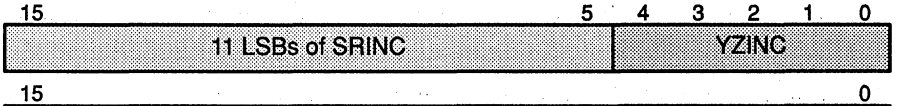
B-file register?		register number:
I/O register?	√	*DINC* **(32-bit address): C000 0240h** *DINCL* **(16-bit address): C000 0240h** *DINCH* **(16-bit address): C000 0250h**

Format

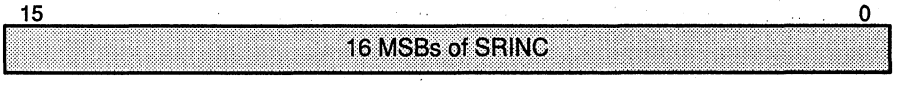
DINC



or DINCL



DINCH



Bits

Bits	Name	Function
0—4	YZINC	Y-zoom increment value
5—31	SRINC	Screen-refresh address increment value

Note:

You can access the display increment registers separately or together by using different addresses and different field sizes.

- To access DINC as a single 32-bit register, access the 32-bit field at address C000 0240h.
- To access DINCL as a 16-bit register, access the 16-bit field at address C000 0240h.
- To access DINCH as a 16-bit register, access the 16-bit field at address C000 0250h.

Description

The DINC registers contain two increment values. One controls the TMS34020's Y-zoom feature; the other is used for screen refreshes.

YZINC
bits 0—4

Y-zoom increment value

YZINC					Zoom	
4	**3**	**2**	**1**	**0**	**Factor**	**Description**
0	0	0	0	0	1	No repetition of scan lines
1	0	0	0	0	2	Repeat scan line 2 times
0	1	0	0	0	4	Repeat scan line 4 times
0	0	1	0	0	8	Repeat scan line 8 times
0	0	0	1	0	16	Repeat scan line 16 times
0	0	0	0	1	32	Repeat scan line 32 times

If you want to change the value in YZINC when video is enabled (ENV[DPYCTL]=1), you should also clear YZCNT[DPYNX] to 0.

SRINC
bits
5—31

Display increment value

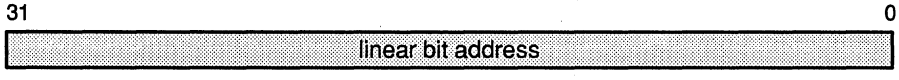
The 27-bit SRINC value specifies the amount by which the address stored in SRNX[DPYNX] should be incremented following completion of each horizontal-blanking screen-refresh cycle. This value corresponds to the display pitch. If you are using the Y-zoom feature, the TMS34020 will not increment SRNX after each horizontal-blanking screen refresh. Instead, it will increment SRNX after every *n*th screen refresh if zoom×*n* is selected.

For both interlaced and noninterlaced video, load SRINC with the display pitch. In interlaced video, the SRNX[DPYNX] registers are automatically incremented by 2xSRINC to account for the fact that in any field (odd or even), only alternate lines are displayed.

The bits of DPYNX and DPYST that correspond to the column and row addresses actually latched into the VRAMs vary from system to system. The bits of SRINC that contain the display pitch depend on the alignment of the address in SRNX[DPYNX] and SRST[DPYST]. SRINC is usually a power of 2 or a sum of two powers of 2, but can be any arbitrary value required.

B-file register?	√	register number: B3
I/O register?		address:

Format



Description

DPTCH defines the linear difference in the starting memory addresses of adjacent rows of a destination array. The TMS34020 uses the value in DPTCH to move from row to row through the destination array. DPTCH can have any value that is a multiple of the current pixel size. Note that XY-to-linear conversion is most efficient when DPTCH is a power of 2.

If you're manually converting an XY address to a linear address, you can use the SETCDP instruction. SETCDP uses the DPTCH value to calculate the destination pitch conversion factor and loads the correct value into CONVDP. The contents of CONVDP are then available for use by the CVXYL or CVDXYL instructions; these instructions perform the conversion.

Which instructions use this register?

Instruction	DPTCH's format and valid values
CVXYL	Linear; any value
CVDXYL	Linear; any value
DRAV	Linear; any value
FILLs (both)	Linear; any value
FLINE, LINE	Linear; any value
PFILL XY	Linear; any value
PIXBLTs (all)	Linear; any value
PIXT *Rs*, **Rd*.**XY**	Linear; any value
PIXT **Rs*.**XY**, **Rd*.**XY**	Linear; any value
SETCDP	Linear; any value
TFILL	Linear; any value
VBLT	Linear; any value
VFILL	Linear; any value

Example

```
DPTCH   .set   B3

        MOVI   00001000h, DPTCH  ; Power of 2
        MOVI   00010AFCh, DPTCH  ; Arbitrary value
        MOVI   00000180h, DPTCH  ; 2 powers of 2
                                 ; (128 + 256)
```

B-file register?		register number:
I/O register?	√	**address: C000 01E0h**

Format

15	0
no defined function for TMS34020	

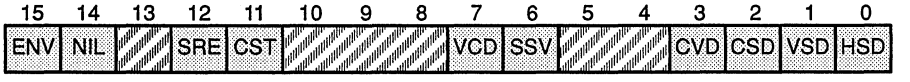
Description

DPYADR is a simple 16-bit read/write location that is included for compatibility with the TMS34010. The TMS34010 used DPYADR as the source of the row and column addresses output during screen-refresh cycles. The TMS34020 uses a different register for this purpose and assigns no function to DPYADR.

B-file register? ☐ register number:

I/O register? √ **address: C000 0080h**

Format



Bits

Bits	Name	Function
0	HSD	Selects the horizontal-sync direction
1	VSD	Selects the vertical-sync direction
2	CSD	Selects the composite-sync direction
3	CVD	Disables composite video
6	SSV	Enables split-serial-register midline reload
7	VCE	Enables video capture
11	CST	Enables CPU serial-register transfers
12	SRE	Enables screen refreshes
14	NIL	Enables noninterlaced video
15	ENV	Enables video
4—5 8—10		Reserved; do not use

Description

DPYCTL contains several parameters that control video timing.

HSD
bit 0

Horizontal-sync direction

When HSD =	$\overline{\text{HSYNC}}$ is an . .
0	Input
1	Output

The HSD bit controls the direction (input or output) of the $\overline{\text{HSYNC}}$ signal.

- When HSD=0, $\overline{\text{HSYNC}}$ is an input. The TMS34020's internal video timing logic synchronizes to external pulses applied to $\overline{\text{HSYNC}}$. Whenever the TMS34020 detects the start of an external horizontal-sync pulse on $\overline{\text{HSYNC}}$, it loads HCOUNT from SETHCNT. The internal horizontal- and composite-sync intervals begin if they were not already started.

- When HSD=1, $\overline{\text{HSYNC}}$ is an output and is controlled according to the values in the video timing registers.

VSD
bit 1

Vertical-sync direction

When VSD =	$\overline{\text{VSYNC}}$ is an . .
0	Input
1	Output

VSD controls the direction (input or output) of the $\overline{\text{VSYNC}}$ signal.

- When VSD=0, $\overline{\text{VSYNC}}$ is an input. The TMS34020's internal video timing logic synchronizes to pulses that an external source applies to $\overline{\text{VSYNC}}$. Whenever the TMS34020 detects the start of an external vertical-sync pulse input on $\overline{\text{VSYNC}}$, the TMS34020 loads VCOUNT from SETVCNT. The internal vertical-sync interval begins if it was not already started by the internal video timing logic. Enabling noninterlaced video (NIL=1) also loads HCOUNT from SETHCNT. The internal horizontal- and composite-sync intervals begin if they were not already started by the internal video timing logic.
- When VSD=1, $\overline{\text{VSYNC}}$ is an output and is controlled according to the values in the video timing registers.

CSD
bit 2

Composite-sync direction

CVD	CSD	Status of $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$
0	0	$\overline{\text{CSYNC}}$ is an input
0	1	$\overline{\text{CSYNC}}$ is an output
1	0	undefined
1	1	$\overline{\text{HBLNK}}$ is an output

CSD controls the direction (input or output) of the $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ pin when it is configured as $\overline{\text{CSYNC}}$ (CVD=0).

- If CVD=1, the pin is configured as $\overline{\text{HBLNK}}$, and CSD **must** be 1. When CSD=1, $\overline{\text{CSYNC}}$ (CVD=0) or $\overline{\text{HBLNK}}$ (CVD=1) is an output and is controlled according to values in the video timing registers.
- If CSD=0, $\overline{\text{CSYNC}}$ is an input and the TMS34020's internal video timing logic synchronizes to external pulses applied to $\overline{\text{CSYNC}}$. Whenever the TMS34020 detects the start of an external composite-sync pulse input on $\overline{\text{CSYNC}}$, the TMS34020 loads HCOUNT from SETHCNT. The internal composite-sync interval begins if it was not already started by the internal video timing logic. Normally, the internal horizontal-sync interval also begins if it has not already started. However, in interlaced video (NIL=0), external composite-sync pulses occur every half horizontal scan line during the equalization and serration regions of vertical blanking, so the internal horizontal-sync interval is started by alternate external composite-sync pulses at these times. The first serration pulse input on $\overline{\text{CSYNC}}$ also loads VCOUNT from SETVCNT, and the internal vertical-sync interval begins if it was not already started by the internal video timing logic.

CVD
bit 3

Composite video disable

	Status of $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$	Status of $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$
CVD=0	Selects $\overline{\text{CSYNC}}$	Selects $\overline{\text{CBLNK}}$
CVD=1	Selects $\overline{\text{HBLNK}}$	Selects $\overline{\text{VBLNK}}$

CVD controls the functions of the $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ and $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$ pins. Because both composite and separate synchronization and blanking signals are internal, CVD simply selects which of these functions is visible at the pins.

SSV
bit 6

Split-serial-register midline reload enable

SRE	SSV	Effect
0	0	Disables split-serial-register midline reload
0	1	Disables split-serial-register midline reload
1	0	Disables split-serial-register midline reload
1	1	Enables split-serial-register midline reload

SSV determines whether or not the TMS34020 performs screen-refresh cycles for VRAMs with split-serial registers during the active display time. SSV works in conjunction with the screen-refresh enable bit (SRE). If SSV=1 and screen refreshes are enabled (SRE=1), the TMS34020 performs an ordinary screen-refresh (memory-to-register) cycle during horizontal blanking, which

Step 1: reloads an entire row of VRAM memory into the VRAM serial registers,

Step 2: updates the address in SRNX[DPYNX] for the next screen refresh, and

Step 3: reloads the SCOUNT register with the tap point of the current screen-refresh address, using the mask in the DPYMSK register.

This is immediately followed by a split register-to-memory cycle, which

Step 4: reloads the half serial registers that do not contain the current tap point so that they contain data for the next half line to be displayed,

Step 5: and initializes the VRAM for split-serial-register operation.

After blanking ends, SCLK starts shifting data from the VRAMs and increments SCOUNT (which tracks the VRAM tap point). When SCOUNT overflows from a tap point of all 1s to all 0s, this indicates that the VRAMs have switched from one half serial register to the other. A split register-to-memory cycle (midline-reload cycle) is executed, performing Step 4.

Note:

You must provide an SCLK pulse to the VRAMs between these two screen-refresh cycles to ensure that the tap-point address is latched correctly.

VCE
bit 7

Video capture enable

VCE	Effect
0	Selects memory-to-register screen-refresh cycles
1	Selects register-to-memory screen-refresh cycles

VCE determines whether TMS34020 screen-refresh cycles are memory-to-register cycles or register-to-memory cycles. VCE affects only those memory cycles that are initiated by the TMS34020's video timing logic.

When VCE=1, screen-refresh cycles initiated by the video timing logic are performed as screen-capture cycles; data shifted into the VRAM serial registers is transferred to the specified row of VRAM ready for the next line.

Do not use midline reload in systems with video capture; clear SSV to 0. VRAMs support only the transfer of an *entire* serial register's contents into the specified memory row. If you used midline reload to condense the display memory into a contiguous region of VRAM, it would be necessary to transfer only some of the bits of the serial register into the memory array in order to not overwrite previously captured data.This is not possible.

CST
bit 11

CPU serial-register transfer enable

CST	Effect
0	Pixel-access cycles occur normally
1	Converts pixel-access cycles into VRAM serial-register-transfer cycles

CST converts an ordinary pixel access into a VRAM serial-register transfer cycle. Several of the TMS34020's graphic instructions treat data as pixels.

By default, CST=0 and accesses of pixel data are normal read and write cycles. When CST=1, however, pixel accesses are converted to serial-register-transfer cycles:

- A pixel read cycle becomes a memory-to-register cycle.
- A pixel write cycle becomes a register-to-memory cycle.

This register-transfer cycle is performed under explicit program control, as opposed to the screen-refresh cycles enabled by the SRE bit, which are automatically generated at regular intervals.

CST is useful for bulk initialization of an entire VRAM array. You can clear the entire screen to a specified background color in only 256 memory cycles for 64K×*n* VRAMs, or 512 memory cycles for 256K×*n* VRAMs (where *n* is the number of planes within the VRAM). (Note that the TMS4461 and TMS44251 have this capability, but not all VRAMs support this function.) The CST bit affects only *pixel* accesses; it does not affect instruction fetches or nonpixel accesses.

SRE
bit 12

Screen-refresh enable

ENV	SRE	Effect	ENV	SRE	Effect
0	0	Disables screen refresh	1	0	Disables screen refresh, but tracks the address
0	1	Disables screen refresh	1	1	Enables screen refresh

SRE enables automatic screen refreshing. Screen refreshes are performed by means of the VRAM memory-to-register cycles, which the TMS34020 performs automatically during each horizontal-blanking interval. DPYST, DINC, and DPYCTL control generation of addresses output during these cycles. If ENV=1, the TMS34020 continues to generate the screen-refresh address internally, even if SRE=0. This allows an external source to insert images into the display; during each horizontal-blanking period, the TMS34020 continues to track the address of the image hidden beneath the external image. Thus, the TMS34020 can restart screen refreshes after inserting the image without adjusting the address to account for the undisplayed lines.

Changing SRE's value affects screen refreshes, starting with the next horizontal-blanking period—or, if SSV=1 and SCLK is running, starting with the next time the VRAMs change active half serial registers, whichever comes first. Normally, however, SCLK does not shift data to the screen when screen refreshes are disabled.

NIL
bit 14

Noninterlaced video enable

NIL	Effect
0	Selects interlaced video timing
1	Selects noninterlaced video timing

NIL selects between an interlaced or a noninterlaced display. The TMS34020 modifies its video timing output signals according to NIL's value. Chapter 9 describes the timing differences between interlaced and noninterlaced video.

ENV
bit 15

Enable video

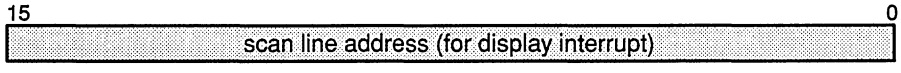
ENV	Effect
0	Blanks the entire video screen
1	Enables the video display

ENV enables or disables the video display.

- When ENV=0, the display remains blanked. The signal output at $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$ (and at $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ if CVD=1) is forced to remain at active low throughout the frame, inhibiting the display interrupt. (DIP[[INTPEND]] can't be set. If DIP is already set when ENV changes from 1 to 0, it remains set until you explicitly clear it.)
- When ENV=1, the video display is enabled. The output signals are controlled according to the parameters in the video timing registers, and DIP is set when VCOUNT becomes equal to DPYINT.

B-file register?		register number:
I/O register?	√	**address: C000 00A0h**

Format



Description

DPYINT identifies the next scan line (in some cases, the next half scan line) at which a display interrupt will be requested. DPYINT helps to coordinate software activity with the refreshing of a selected horizontal scan line on the screen.

The video timing logic compares the contents of DPYINT to VCOUNT. This usually coincides with the start of the horizontal-blanking interval that marks the end of the line designated by the value in DPYINT. If interlaced video is enabled (NIL=0), then during the part of the vertical-blanking interval when VCOUNT is incremented every half line, DPYINT is compared to VCOUNT just before VCOUNT is incremented, at the end and in the center of each horizontal scan line. When VCOUNT=DPYINT, a display interrupt is requested and DIP[[INTPEND]] is set to 1.

For split-screen applications, you can load a new value into the SRNX[[DPYNX]] bits, immediately following detection of the 0-to-1 transition of DIP. The new SRNX value does not affect the line that follows the current horizontal-blanking interval, but affects the next line. A screen-refresh cycle will be scheduled to occur at the start of the same horizontal-blanking period in which DIP is set. At the end of the screen-refresh memory cycle, the screen-refresh address in SRNX is automatically incremented. Requests for screen-refresh cycles have a higher priority than CPU requests. Thus, if the CPU loads a new value into SRNX immediately after setting the DIP bit, SRNX will not actually be modified until after the screen-refresh cycle completes and the existing contents are incremented. This new address becomes the address used in the next screen refresh. SRNX can change only during the scan line under explicit program control. The display interrupt is disabled when ENV[[DPYCTL]] is 0.

B-file register? ☐ register number:

I/O register? √

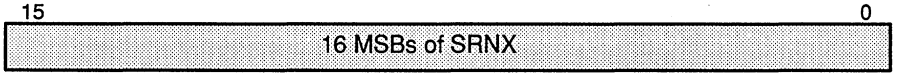
DPYNX **(32-bit address): C000 0220h**
DPYNXL **(16-bit address): C000 0220h**
DPYNXH **(16-bit address): C000 0230h**

Format

DPYNX

or DPYNXL

DPYNXH



Bits

Bits	Name	Function
0—4	YZCNT	Y-zoom count
5—31	SRNX	Next screen-refresh address

Note:

You can access the display next-address registers separately or together by using different addresses and different field sizes.

- To access DPYNX as a single 32-bit register, access the 32-bit field at address C000 0220h.
- To access DPYNXL as a 16-bit register, access the 16-bit field at address C000 0220h.
- To access DPYNXH as a 16-bit register, access the 16-bit field at address C000 0230h.

Description

The DPYNX registers contain two values. One is used for the Y-zoom feature; the other is an address that is output during a screen-refresh cycle.

YZCNT
bits 0—4

Y-zoom increment value

The 5-bit YZCNT value determines when the SRNX address can be incremented by SRINC[[DINC]]. After every local-memory screen-refresh cycle, the TMS34020 increments YZCNT by the value of YZINC[[DINC]]. If YZCNT=0 before it is incremented, SRNX is incremented at the same time. If YZCNT≠0, SRNX is not incremented, so the next scan line contains the same pixels as the current scan line. This allows the image on the screen to be magnified (or zoomed) in the Y direction. The value of YZINC determines how many times the scan line is output, and thus determines the zoom factor.

YZCNT will equal 0 when

$$n \times YZINC \text{ modulo } 32 = 0$$

(n is the Y-zoom factor). This occurs once every n scan lines. During each vertical-blanking interval, YZCNT is reset to YZINC.

SRNX
bits
5—31

Next screen-refresh address

The 27-bit SRNX value represents the long-word address that is output during a screen-refresh cycle. When YZCNT=0, the TMS34020 increments SRNX (by SRINC[[DINC]]) after each screen-refresh cycle.

SRNX consists of a row-address portion and a column-address portion, corresponding to the bits of the address connected to the VRAMs at row- and column-address times on RCA0—RCA12. The column- and row-address fields should be contiguous to one another within SRNX. However, you can choose where the two fields are placed within the 27 bits of SRNX, provided that all of the row-address bits are output on RCA0—RCA12 at row-address time, and all the column-address bits are output on RCA0—RCA12 at column-address time. Section 8.16.2 (page 8-53) details which bits of the logical address are output on RCA0—RCA12 at row- and column-address times.

B-file register?		register number:
I/O register?	√	**address: C000 02E0h**

Format

Description

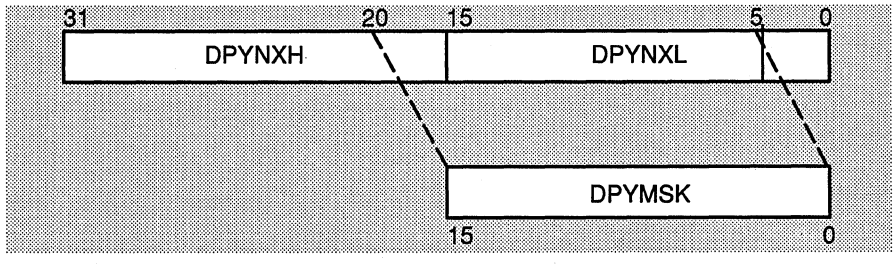
DPYMSK is used when midline-reload screen refreshes are enabled (SSV[[DPYCTL]]=1). DPYMSK defines which bits of the address in SRST[[DPYST]] and SRNX[[DPYNX]] correspond to the tap-point portion of the address output during screen-refresh cycles. DPYMSK is loaded with a field of contiguous 1s to indicate where the tap point is within SRST and SRNX. This information is then used to perform these functions:

- Isolate the tap point from the 27-bit, long-word screen-refresh address so that it can be loaded into SCOUNT (the counter register that tracks the VRAM tap point and schedules midline-reload split-serial-register screen-refresh cycles during the active portion of the display).
- Determine which bit of the 27-bit, long-word address should be incremented so that the address output during a midline-reload memory cycle is the address of the next half-row of VRAM.

DPYMSK maps to the 16 LSBs of SRST and SRNX, which correspond to bits 5 to 20 of DPYST and DPYNX, respectively. There are two reasons for this skewed mapping:

- Bit 5 of DPYNX is the least significant address bit output during screen-refresh memory cycles.
- If the mapping were not skewed, a 32-bit DPYMSK register would be required to determine which bits in DPYNXH and DPYSTH were part of the tap point.

Figure 4–6. How DPYMSK Maps to the Logical Screen-Refresh Address



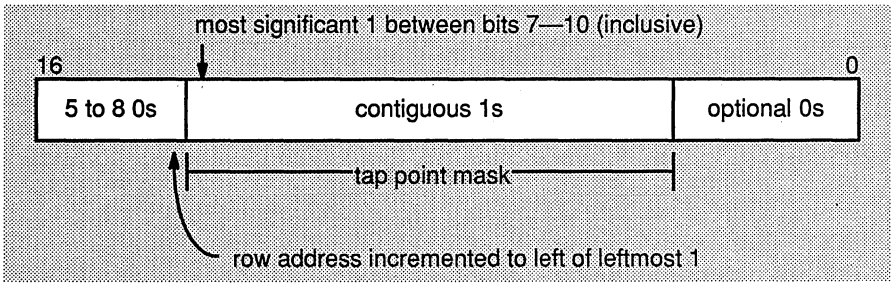
The tap point's LSB does not have to be in bit 5 of the logical address. This allows for some bank selection bits at the least significant end of the address.

The bits from the logical address identified by DPYMSK to be the tap point are automatically shifted right by the number of 0s at the least significant end of DPYMSK before being loaded into SCOUNT.

The number of contiguous 1s in DPYMSK depends on the number of address bits needed to specify the tap-point address for the VRAMs. This field is either right-justified in DPYMSK, or it is preceded by several 0s if there are interleaved banks addressed with bits of the logical address less significant than the tap point. The remaining bits at the most significant end of DPYMSK should be 0s.

During a midline-reload screen-refresh cycle, the address output by the TMS34020 is that of the next half-row of VRAM to be displayed. This address has a 0 tap point and is stored in an inaccessible register. This address must then be incremented to point to the next half-row after that, as there may be multiple midline reloads on any one horizontal scan line. The address is incremented at the next bit position to the left of the leftmost 1 in DPYMSK. The most significant 1 in DPYMSK must not be any higher than bit 10 or lower than bit 7 of DPYMSK.

Figure 4–7. The Functions of the Different Fields of DPYMSK



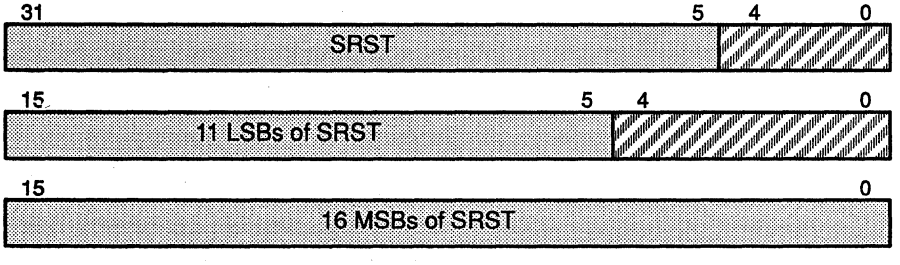
Because the serial registers in the VRAMs are split into two parts, the tap point for each VRAM needs to address only enough bits for half a row. This means, for instance, that for a 256K×4 VRAM (the smallest available with the split-serial-register), the tap point mask should be 8 contiguous 1s despite the fact that a 256K×4 VRAM or DRAM requires 9 row- and column-address bits; when considering the split-serial register, the MSB of the column address is not part of the tap point, but selects between the upper and lower serial register halves. Similarly, 1M×*n* VRAMs should have a 9-bit tap point mask.

B-file register? ☐ register number:

I/O register? √

DPYST **(32-bit address): C000 0200h**
DPYSTL **(16-bit address): C000 0200h**
DPYSTH **(16-bit address): C000 0210h**

Format



Bits

Bits	Name	Function
0—4	▨	Reserved; do not use
5—31	SRST	Screen-refresh start address

Note:

You can access the display start address registers separately or together by using different addresses and different field sizes.

- To access DPYST as a single 32-bit register, access the 32-bit field at address C000 0200h.
- To access DPYSTL as a 16-bit register, access the 16-bit field at address C000 0200h.
- To access DPYSTH as a 16-bit register, access the 16-bit field at address C000 0210h.

Description

The 27-bit SRST value represents the address that points to the pixel at the left of the first line displayed on the screen. This address is used in calculating the screen-refresh address output just before the start of each frame (or field in interlaced video). There are a number of cases to consider.

- In noninterlaced video, the address output at the beginning of each field is simply that contained in SRST, so SRST is copied into SRNX[DPYNX] at the beginning of each vertical-blanking period.
- In interlaced video,
  - At the beginning of the even field, the address of the first pixel displayed is that of the pixel half way across the first line. So,

SRINC[[DINC]]/2 is added to SRST before the address is loaded into SRNX[[DPYNX]] at the start of the vertical-blanking interval.

- At the beginning of the odd field, the address of the first pixel displayed is that of the first pixel on the second line. So, SRST is added to SRINC[[DINC]] before being loaded into SRNX[[DPYNX]] at the start of the vertical-blanking interval.

- The address output at the beginning of the second line of the even field in interlaced video is that of the first pixel on the third line of the display. Normally, the address is generated by adding SRINC[[DINC]] or, in interlaced video, 2×SRINC to the value in SRNX[[DPYNX]]. However, in this particular instance ,SRNX contains the address of the pixel half way across the first line of the display, and so would need 1.5×SRINC added to it to arrive at the correct address. Because of this, the value of SRNX generated after the first screen refresh in the even field is generated by adding 2×SRINC to SRST.

The address consists of a row-address portion and a column-address portion, corresponding to the bits of the address connected to the VRAMs at row- and column-address times, respectively, on the RCA0—RCA12 bus. The column- and row-address fields should be contiguous to one another within the 27 bits of SRNX. However, you can choose where the two fields are placed within the 32-bit register, provided that all of the row-address bits are output on RCA0—RCA12 at row-address time, and all the column- address bits are output on RCA0—RCA12 at column-address time. Section 8.16.2 (page 8-53) details which bits of the logical address are output on RCA0—RCA12 at row- and column-address times.

B-file register? ☐ register number:

I/O register? ☑ **address: C000 0090h**

Format

15	0
no defined function for TMS34020	

Description

DPYSTRT is a simple 16-bit read/write location that is included for compatibility with the TMS34010. The TMS34010 used DPYSTRT to indicate the address of the first pixel to be displayed in each frame. The TMS34020 uses a different register for this purpose and assigns no function to DPYSTRT.

B-file register?		register number:
I/O register?	√	**address: C000 01B0h**

Format

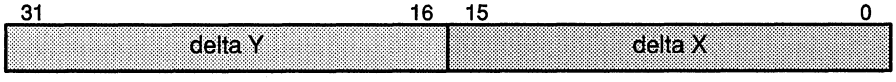
15	0
no defined function for TMS34020	

Description

DPYTAP is a simple 16-bit read/write location that is included for compatibility with the TMS34010. The TMS34010 used DPYTAP to indicate the VRAM tap point used during screen-refresh cycles. The TMS34020 uses a different register for this purpose and assigns no function to DPYTAP.

B-file register?	√	**register number: B7**	
I/O register?		address:	

Format



Description

DYDX defines the X and Y dimensions of the rectangular destination array for PIXBLT and FILL instructions. Both the X and Y dimensions are in pixels; the DX value is the number of pixels in the width of the array, and DY is the number of rows of pixels in the array.

When window clipping is selected, the pixel block dimensions for the transfer are determined by the relationships between WSTART, WEND, DADDR, and DYDX. If either the X or Y dimension is 0, then the entire block is interpreted as having a dimension of 0; no transfer is performed.

The values for DY and DX can range up to the coordinate extent of the display (up to 65,535, depending on the display pitch and pixel size selected). For window operations, the relationship between DYDX, WSTART (at location [X_{start}, Y_{start}]), and WEND (at location [X_{end}, Y_{end}]) is such that DY ≤ ($Y_{end} - Y_{start} + 1$) and DX ≤ ($X_{end} - X_{start} + 1$). The value in DYDX is used with WSTART, WEND, and DADDR to preclip pixels, lines, and pixel arrays.

Most graphics instructions do not modify the contents of DYDX. For FILL XY, PIXBLT B,XY, PIXBLT L,XY, and PIXBLT XY,XY, with window option 1, however, DYDX is used with DADDR to perform a common rectangle function. In this case, the instruction sets DYDX to the dimensions of the common pixel block represented by the intersection of the original destination array and the window. No drawing is performed. If there is no common rectangle, the V bit is not set, and the value of DYDX is indeterminate.

Which instructions use this register?

Instruction	DYDX's format and function
FILL L	Array dimensions in XY format
FILL XY	Array dimensions in XY format; special results when W=1 is selected, as previously noted
FLINE, LINE	Dimensions of the rectangle described by the line to be drawn
PIXBLT B, L	Array dimensions in XY format
PIXBLT B, XY	Array dimensions in XY format; special results when pick is selected, as previously noted
PIXBLT L, L	Array dimensions in XY format
PIXBLT L, XY	Array dimensions in XY format; special results when pick is selected, as previously noted
PIXBLT XY, L	Array dimensions in XY format

Instruction	DYDX's format and function
PIXBLT XY, XY	Array dimensions in XY format; special results when pick is selected, as previously noted
PIXBLT L, L	Array dimensions in XY format
PIXBLT L, XY	Array dimensions in XY format; special results when pick is selected, as previously noted
PFILL XY	XY; dimensions of the fill area
VBLT	XY; dimensions of the pixel block
VFILL	XY; dimensions of the fill area

Example

This example illustrates the relationship of DYDX to WSTART and WEND by setting DYDX to the width and height of the clipping window.

```
WSTART  .set    B5
WEND    .set    B6
DYDX    .set    B7

        MOVE    WEND, DYDX      ; Put WEND into DYDX
        SUBXY   WSTART, DYDX    ; Generate (WEND–WSTART)
        ADDXYI  [1, 1], DYDX    ; Increment by 1 in each
                                ; dimension
```

B-file register?		register number:
I/O register?	√	**address: C000 01D0h**

Format

Description

HCOUNT is a 16-bit counter used for generating horizontal and composite video signals. HCOUNT is incremented on the falling edge of the VCLK, thus counting the number of VCLK periods per horizontal scan line.

- To generate horizontal sync and blanking signals, the value of HCOUNT is compared to the values of HESYNC, HEBLNK, HSBLNK, and HTOTAL.
- To generate composite serration and equalization pulses, HCOUNT is compared to the value of HESERR and half the value of HESYNC, respectively.

HCOUNT is reset to 0 on the next VCLK falling edge after HCOUNT= HTOTAL, and the $\overline{\text{HSYNC}}$ output is driven low. If $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ is configured to $\overline{\text{CSYNC}}$, this pin is also driven low.

If interlaced composite video is enabled, HCOUNT is also reset to 0 on the next VCLK falling edge after HCOUNT=HTOTAL/2 during the equalization and serration regions of vertical blanking.

In external horizontal or composite-sync video, HCOUNT is reloaded from the SETHCNT register on the rising edge of the video input clock. This is 4 VCLK cycles after the $\overline{\text{HSYNC}}$ or $\overline{\text{CSYNC}}$ input signals, respectively, are driven low.

Two separate, asynchronous elements of the TMS34020 logic can access the HCOUNT register.

- The video timing control logic (which runs synchronously to the VCLK) increments, clears, and reloads HCOUNT (from SETHCNT) in generating the sync and blanking signals.
- The internal processor (which runs synchronously to LCLK1 and LCLK2) can access HCOUNT as an I/O register.

No synchronization between these subsystems is provided. HCOUNT can be reliably read from or written to only while VCLK is held at the logic-high level. HCOUNT is typically not read from or written to except during chip test.

B-file register?		register number:
I/O register?	√	**address: C000 0030h**

Format

Description

HEBLNK is used for generating the $\overline{\text{HBLNK}}$ or $\overline{\text{CBLNK}}$ signals output to a video monitor. The 16-bit HEBLNK value is compared to HCOUNT and defines the point at which the horizontal-blanking interval ends.

For composite video, select the $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$ pin as $\overline{\text{CBLNK}}$. $\overline{\text{CBLNK}}$ outputs the logical-OR of the internal horizontal- and vertical-blanking signals; it is low if horizontal- or vertical-blanking is active internally.

Most video monitors require HEBLNK to contain a value that is less than HSBLNK but greater than HESYNC.

B-file register?		register number:
I/O register?	√	**address: C000 0270h**

Format

15	0
end of composite sync during serration region	

Description

HESERR is used for generating the composite-serration pulses output to the video monitor. You need to program this register only when the $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ pin is selected as $\overline{\text{CSYNC}}$. The 16-bit HESERR value defines the point at which the composite-sync pulse ends during the serration region of vertical blanking (this coincides with the vertical-sync region). When the value in HCOUNT=HESERR during this region, the signal output from the $\overline{\text{CSYNC}}$ pin is driven inactive high to signal the end of the serration pulse.

NTSC and similar composite-video standards require HESERR to contain the value (HTOTAL/2) – HESYNC – 1. (Serration pulses occur every half line, and in each cycle, the $\overline{\text{CSYNC}}$ signal is inactive high for the same duration as horizontal sync is active low.)

When external composite sync is enabled, load HESERR with a value that ensures that the HCOUNT does not become equal to HESERR before the external composite-sync signal goes inactive high, but before $\overline{\text{CSYNC}}$ goes active low again.

B-file register?		register number:
I/O register?	√	**address: C000 0010h**

Format

Description

HESYNC is used for generating the horizontal- and composite-sync signals output to a video monitor. The 16-bit HESYNC value defines the point at which the horizontal-sync pulse ends. If the $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ pin is selected as $\overline{\text{CSYNC}}$, HESYNC also determines the point at which the composite- sync pulse ends (except during the serration region of vertical blanking). When the value in HCOUNT=HESYNC, the signal output from the $\overline{\text{HSYNC}}$ and $\overline{\text{CSYNC}}$ pins is driven inactive high to signal the end of the horizontal-sync interval. During the equalization regions of vertical sync, the $\overline{\text{CSYNC}}$ pin is driven inactive high when HCOUNT= HESYNC/2, indicating the end of the composite-equalization interval.

Monitors typically require HESYNC to contain a value less than HEBLNK; however, the TMS34020 does not require this. The minimum value of HESYNC is 0.

When external horizontal or composite sync is enabled, you should load HESYNC with a value that ensures two things:

- that the value in HCOUNT does not reach HESYNC before the external horizontal or composite-sync signal goes inactive high
- that the value in HCOUNT reaches HESYNC before $\overline{\text{HSYNC}}$ or $\overline{\text{CSYNC}}$ goes active low again

For external composite sync, HCOUNT must not become equal to HESYNC/2 before the composite-sync equalization pulse goes inactive.

B-file register?		register number:
I/O register?	√	*HSTADRH address:* **C000 00E0h** *HSTADRL address:* **C000 00D0h**

Format

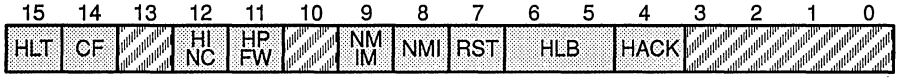
15	0
no defined function for TMS34020	

Format

HSTADRH and HSTADRL are simple 16-bit read/write locations that can be used to store information passed between the host and the TMS34020.

B-file register?		register number:
I/O register?	√	**address: C000 0100h**

Format



Bits

Bits	Name	Function
4	HACK	Acknowledges TMS34020 halt
5—6	HLB0, HLB1	Selects the host byte order
7	RST	Resets the TMS34020
8	NMI	Enables the nonmaskable interrupt
9	NMIM	Selects the mode for the nonmaskable interrupt
11	HPFW	Enables host prefetches after writes
12	HINC	Enables host increment
14	CF	Flushes the cache
15	HLT	Halts TMS34020 processing
0—3 10, 13		Reserved; do not use

Description

HSTCTLH contains10 programmable bits for controlling host interface communications. Both the TMS34020 and a host processor can read from or write to HSTCTLH; typically, however, only the host alters HSTCTLH.

HACK
bit 4

Halt acknowledge

HACK	Description
0	The TMS34020 is running
1	The TMS34020 is halted

The TMS34020 sets HACK when the TMS34020 is halted by setting HLT. (Both the host processor and the TMS34020 can write to HLT; in either case, the TMS34020 sets HACK.) By polling the value of HACK, the host can determine when the TMS30420 actually halts. The TMS34020 automatically clears HACK when HLT is cleared to release the TMS34020 from halt.

HLB0, HLB1
bits 5&6

Host last byte

HLB1	HLB0	Last Byte Accessed	HLB1	HLB0	Last Byte Accessed
0	0	byte 3	1	0	byte 1
0	1	byte 2	1	1	byte 0

The host processor sets the HLB bits to inform the TMS34020 which byte of a 32-bit word the host will access last. The TMS34020's host interface uses the HLB value to determine when to prefetch the next 32-bit word in memory. If prefetches are enabled, the TMS34020 prefetches the next location from memory after the host accesses the last byte. The HPFW bit determines whether this access must be a read or a write. The 2-bit HLB code allows for all host byte-ordering conventions.

For an 8-bit host, the value of both bits determines after which byte access of the appropriate type (read or write) the TMS34020 will prefetch the next 32-bit location. For a 16-bit host, the value of HLB1 alone is sufficient to determine after which 16-bit word access the TMS34020 will prefetch the next 32-bit location. For a 32-bit host, any combination of HLB1—HLB0 causes the TMS34020 to prefetch the next 32-bit location after each host access of the appropriate type.

RST
bit 7

Reset

RST	Effect
0	Allows normal operation
1	Executes reset

Setting RST to 1 has the same effect as asserting the $\overline{\text{RESET}}$ pin low—the TMS34020 executes a reset. However, when RST is set, *only* the TMS34020 is reset; typically, the $\overline{\text{RESET}}$ signal is connected to all devices in the system, and asserting it low affects the entire system.

While the TMS34020 is executing reset internally and the $\overline{\text{RESET}}$ pin is high, DRAM $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh cycles are performed, thus preserving the contents of the DRAMs in the system. During a hardware reset, the TMS34020 uses the value of $\overline{\text{HCS}}$ just before the rising edge of $\overline{\text{RESET}}$ to determine whether to come up in self-bootstrap or host-present mode. When reset is caused by setting RST, however, there is no rising edge on the $\overline{\text{RESET}}$ pin. Because of this, the TMS34020 remembers which mode it was brought up in the last time a reset was caused by asserting $\overline{\text{RESET}}$, and it configures itself in that mode.

It is not necessary to clear RST; reset clears it automatically.

NMI
bit 8

Nonmaskable interrupt, host to TMS34020

NMI	Effect
0	No NMI is requested
1	The host is requesting an NMI

The nonmaskable interrupt allows a host processor to redirect the CPU's execution flow to an NMI routine, regardless of the current state of the interrupt mask flags. The host writes a 1 to the NMI bit to send a nonmaskable interrupt

request to the TMS34020. The interrupt request cannot be disabled, and is always executed (unless the TMS34020 is reset before it can complete interrupt execution). The interrupt is initiated immediately when NMI is set (at the time the current instruction completes execution, or at the next interruptible point in an instruction). Once the interrupt is taken, internal logic automatically clears the NMI bit to 0.

You can use NMI to generate a soft reset after the host downloads new code into TMS34020 memory. After execution of a nonmaskable interrupt, screen-refresh and DRAM-refresh functions continue unaffected. The interrupt does not alter the contents of internal registers except for HSTCTLH (the NMI bit), although the NMI service routine may alter them.

NMIM
bit 9

Nonmaskable interrupt mode

NMI	NMIM	Effect	NMI	NMIM	Effect
0	0	No effect	1	0	NMI, save context on current stack
0	1	undefined	1	1	NMI, discard context

The NMI mode bit determines whether or not the context (PC and ST) of the interrupted program is saved when a nonmaskable interrupt occurs.

- When NMIM=0, the TMS34020 saves the context on the system stack before executing the NMI service routine. This is useful for applications (single-step instruction execution, for example) that must preserve the PC's status between consecutive nonmaskable interrupts. Note that saving the context may be of no benefit if either
  - control will never be returned to the interrupted program, or
  - the integrity of the stack pointer is suspect.
- When NMIM=1, the TMS34020 discards the context when it executes the NMI service routine. You can use a nonmaskable interrupt to simulate a hardware reset in software (using the NMI vector). The NMI does not reset the I/O registers; if you simulate a hardware reset with an NMI, the NMI service routine should reset the I/O registers.

HPFW
bit 11

Host prefetch-after-write enable

HINC	HPFW	Effect
0	0	Disables prefetching and incrementing of internal address
0	1	Disables prefetching and incrementing of internal address
1	0	Enables prefetching after last byte read and incrementing of internal address after any last byte access
1	1	Enables prefetching after last byte write and incrementing of internal address after any last byte writes

HPFW works with the HINC bit to enhance a host processor's access to blocks of TMS34020 memory. When host prefetches are enabled (HINC=1), the value

of HPFW determines whether prefetches are executed after a read to or a write from the last byte of a word (identified by the HLB bits).

- Selecting **HINC=1** and **HPFW=0** enhances the host processor's ability to read contiguous blocks of TMS34020 memory. This tells the TMS34020 to prepare for the host's next read request by prefetching the next 32-bit location in memory after completing the read of the current long word. If the host uses implicit addressing to access TMS34020 memory (that is, the host provides only the first address of a contiguous block of memory), the TMS34020 automatically generates subsequent addresses by incrementing the address after each access (regardless of whether the access is a read or a write).
- Selecting **HINC=1** and **HPFW=1** enhances the host processor's ability to modify contiguous blocks of TMS34020 memory. This tells the TMS34020 to prepare for the host's next read request by prefetching the next 32-bit location in memory after the write to the current word is complete. This provides an efficient read-modify-write mechanism. If the host uses implicit addressing to access a block of TMS34020 memory, the TMS34020 generates subsequent addresses by incrementing the address after each write.

If the host is not using implicit addressing, prefetching could yield an unwanted address; however, the TMS34020 has a built-in mechanism that compares the fetched address to the requested address. If the TMS34020 prefetches an unwanted location, it makes an additional access to the requested location. This ensures that the host always accesses the correct location.

HINC
bit 12

Host increment

HINC	Effect
0	Disables prefetching, incrementing, and comparison of addresses
1	Enables prefetching, incrementing, and comparison of addresses

Setting HINC enhances the TMS34020's host interface performance by providing these features.

- **Address comparison.** The TMS34020 compares the address most recently read or prefetched by the host with the address currently requested by the host on read accesses. This allows prefetching while ensuring that the correct location is always accessed. If the host requests access to a location different from a prefetched location, the TMS34020 detects this and initiates another access to the explicitly requested location.

  Address comparison is also useful if the host is not a 32-bit machine. In this case, the host must perform multiple reads to fully read a 32-bit word. The address comparison ensures that once data from the address is latched into the external host data transceivers, accesses to other bytes of the same word do not cause the data to be fetched repeatedly from the TMS34020's local memory.

- **Address prefetch.** The TMS34020 supports prefetching, providing the host with an efficient method for accessing contiguous blocks of TMS34020 memory.
- **Address increment.** A host can use implicit addressing, supplying only the first address in the block of words that it will read, write, or modify. The TMS34020 automatically increments the address after each last-byte host read/write (HPFW=0), or after each last-byte write (HPFW=1).

CF
bit 14

Cache flush

CF	Effect
0	No effect
1	Flushes and disables the cache

Setting CF to 1 disables the instruction cache and flushes the cache contents. While CF=1, all 4 of the cache's P flags are forced to 0. The TMS34020 must fetch instructions one-at-a-time from local memory.

Normal cache operation resumes when CF is cleared to 0 (assuming that CD[[CONTROL]] also = 0). When the value of CF changes from 1 to 0, the cache begins operation in the same initial state as that which immediately follows reset.

Flushing the cache is useful when the host processor downloads new code to TMS34020 local memory. By setting CF to 1 and then to 0, the host forces the TMS34020 to load new instructions into the cache from memory rather than to continue executing the stale instructions already in the cache.

HLT
bit 15

Halt TMS34020 program execution

HLT	Effect
0	Allows TMS34020 to run
1	Halts TMS34020 instruction execution

When HLT=1, the TMS34020 suspends instruction processing at the next instruction boundary. Once halted, the TMS34020 **does not respond** to interrupt requests, including NMI. Local-memory-refresh and video-timing functions continue unaffected while the TMS34020 is halted. When HLT is cleared to 0, the TMS34020 continues execution.

The state of HLT immediately after reset is determined by the state of the $\overline{\text{HCS}}$ pin at the low-to-high transition of $\overline{\text{RESET}}$:

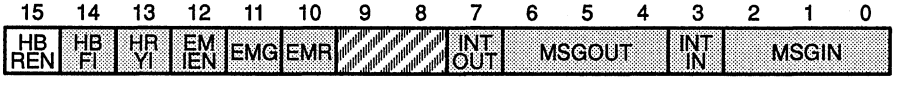
- If $\overline{\text{HCS}}$ is low, HLT is set to 0 and the TMS34020 can begin executing its reset routine.
- If $\overline{\text{HCS}}$ is high, HLT is set to 1 and the TMS34020 is halted.

Both the host processor and the TMS34020 can write to the HLT bit; this means the TMS34020 can halt itself by setting HLT to 1.

B-file register? ☐ register number:

I/O register? √ **address: C000 00F0h**

Format



Bits

Bits	Name	Function
0—2	MSGIN	Buffers an input message
3	INTIN	Sends input interrupt from host to TMS34020
4—6	MSGOUT	Buffers an output message
7	INTOUT	Sends output interrupt from TMS34020 to host
10	EMR	Emulator handshake (request to host)
11	EMG	Emulator handshake (grant from host)
12	EMIEN	Enables emulator inhibit host port interrupt
13	HRYI	Indicates a retry on a host access
14	HBFI	Indicates a bus fault on a host access
15	HBREN	Enables host retry or bus-fault interrupt
8—9		Reserved; do not use

Description

HSTCTLL controls host interface communications. Both the TMS34020 and the host can *read* all of HSTCTLL's bits, but these restrictions apply to *writes*:

	TMS34020	Host Processor
MSGOUT	Can modify	Can't modify
MSGIN	Can't modify	Can modify
INTIN	Can write a 0; writing a 1 has no effect	Can write a 1; writing a 0 has no effect
INTOUT	Can write a 1; writing a 0 has no effect	Can write a 0; writing a 1 has no effect

MSGIN
bits 0—2

Message in—host to TMS34020

The message-in bits buffer a 3-bit interrupt message to the TMS34020 from the host. The host can read from and write to MSGIN, but the TMS34020 can only read MSGIN. MSGIN typically contains a command or status code from the host, which the TMS34020 reads in response to a host-generated interrupt (INTIN=1). The code's meaning depends on your application.

INTIN
bit 3

Interrupt in—host to TMS34020

INTIN	Effect
0	No interrupt request to TMS34020
1	Host requests a TMS34020 interrupt

INTIN controls the host's message interrupt to the TMS34020. To generate this request, the host sets INTIN to 1. The TMS34020 can deactivate the request by clearing INTIN. The host cannot clear INTIN; similarly, the TMS34020 cannot set INTIN. The HIP[[INTPEND]] bit reflects the status of INTIN.

MSGOUT
bits 4—6

Message out—TMS34020 to host

MSGOUT buffers a 3-bit interrupt message to the host from the TMS34020. The TMS34020 can read from and write to MSGOUT, but the host can only read MSGOUT. MSGOUT typically contains a command or status code from the TMS34020, which the host reads in response to a TMS34020-generated interrupt (INTOUT=1). The code's meaning depends on your application.

INTOUT
bit 7

Interrupt out—TMS34020 to host

INTOUT	Effect
0	No interrupt request to host
1	TMS34020 requests a host interrupt

The INTOUT bit controls the TMS34020's message interrupt to the host. The TMS34020 transmits an interrupt request to the host by driving $\overline{\text{HINT}}$ active low. When INTOUT=1, $\overline{\text{HINT}}$ is driven active low; when INTOUT=0, $\overline{\text{HINT}}$ is driven inactive high. The TMS34020 activates the interrupt request by setting INTOUT to 1; the host deactivates the request by clearing INTOUT. The TMS34020 cannot clear INTOUT; similarly, the host cannot set INTOUT.

EMR, EMG
bits 10&11

Emulator (or debugger) handshake—request to/grant from host

EMG	EMR	Interpretation
0	0	No request, no interrupt
0	1	Host request from EMU, interrupt if enabled
1	0	Host released by EMU, interrupt if enabled
1	1	Host grant to EMU, no interrupt

An in-circuit emulator or software debugger may use EMR and EMG for exchanging information and coordinating activity with a host processor. The precise meaning of these bits depends on your application, the emulator or debugger software, and the host processor.

If a debugger or an in-circuit emulator needs to start emulation activity with the TMS34020, the debugger or emulator may set EMR to make this request to the host. If EMIEN=1, setting EMR causes the host to be interrupted via $\overline{\text{HINT}}$. The host then sets EMG to acknowledge this, causing $\overline{\text{HINT}}$ to return to its inac-

tive level. The emulator or debugger then clears EMR, signalling the end of the activity to the host. Again, if EMIEN=1, clearing EMR causes the host to be interrupted via $\overline{\text{HINT}}$. The host then clears EMG, completing the transaction and causing $\overline{\text{HINT}}$ to return to its inactive level.

Only an emulator or debugger should modify EMR, and only the host should modify EMG. If you are not using this protocol, clear these bits to 0.

EMIEN
bit 12

Emulator inhibit host port interrupt enable

EMIEN	Effect
0	EMR XOR EMG causes no host interrupt via $\overline{\text{HINT}}$
1	EMR XOR EMG causes an interrupt to the host via $\overline{\text{HINT}}$

EMIEN controls whether the exclusive-OR of the EMR and EMG bits causes the $\overline{\text{HINT}}$ pin to be driven active low, thus interrupting the host.

HRYI
bit 13

Retry on host access interrupt

HRYI	Effect
0	Host access was not retried
1	Host access was retried ($\overline{\text{HINT}}$ was set active, if enabled)

The TMS34020's host interface sets HRYI if a host access returns a retry memory cycle completion code. The TMS34020 automatically attempts to retry the memory access. If enabled (HBREN=1), the TMS34020 interrupts the host via the $\overline{\text{HINT}}$ pin. The host must ensure that the appropriate action (if any) is taken to clear the cause of the retry, and then the host must clear HRYI.

HBFI
bit 14

Bus fault on host access interrupt

HBFI	Effect
0	Host access was not faulted
1	Host access was faulted ($\overline{\text{HINT}}$ was set active, if enabled)

The TMS34020's host interface sets HBFI if a host access returns a bus-fault memory cycle completion code. The TMS34020 performs no further error recovery, but terminates the host request and drives HRDY high as if the cycle completed successfully. If enabled (HBREN=1), the TMS34020 interrupts the host via the $\overline{\text{HINT}}$ pin. The host must ensure that the appropriate action (if any) is taken to clear the cause of the bus fault, and then the host must clear HBFI.

HBREN
bit 15

Host bus-fault or retry interrupt enable

HBREN	Effect
0	No interrupt sent to the host via $\overline{\text{HINT}}$ if HRYI or HBFI is set
1	An interrupt is sent to the host via $\overline{\text{HINT}}$ if HRYI or HBFI is set

HBREN enables or inhibits the TMS34020 from interrupting the host when a retry or bus fault occurs on a host access.

B-file register?		register number:
I/O register?	√	**address: C000 00C0h**

Format

Description

HSTDATA is a simple 16-bit read/write location that can be used to store information passed between the host and the TMS34020.

B-file register?		register number:
I/O register?	√	**address: C000 0050h**

Format

Description

HSBLNK is used for generating the $\overline{\text{HBLNK}}$ or $\overline{\text{CBLNK}}$ signal output to the video monitor. The 16-bit HSBLNK value is compared to HCOUNT and defines the point at which the horizontal-blanking interval begins.

For composite video, select the $\overline{\text{CBLNK}}/\overline{\text{VBLNK}}$ pin as $\overline{\text{CBLNK}}$. $\overline{\text{CBLNK}}$ outputs the logical-OR of the internal horizontal- and vertical-blanking signals; it is low if either horizontal- or vertical-blanking is active internally.

Several internal events coincide with the start of horizontal blanking:

- A request for a screen-refresh memory cycle is sent to the TMS34020's memory controller.
- If a display interrupt is programmed to occur at a particular horizontal scan line, the actual interrupt request is generated at this point.

Monitors typically require that HSBLNK contain a value less than HTOTAL, but greater than HEBLNK.

B-file register?		register number:
I/O register?	√	**address: C000 0070h**

Format

Description

HTOTAL is used for generating the horizontal- and composite-sync signals output to the video monitor. The 16-bit HTOTAL value is compared to HCOUNT and defines the duration of each horizontal scan line on the screen in terms of VCLK periods.

HTOTAL is compared with the horizontal count in HCOUNT to determine the point at which the horizontal- and composite-sync pulses begin. Usually, HCOUNT counts from 0 to the value in HTOTAL. When HCOUNT = HTOTAL, the $\overline{\text{HSYNC}}$ output is driven active low on the next falling edge of VCLK, and HCOUNT is reset to 0 on the same clock edge. If the $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ pin is selected as $\overline{\text{CSYNC}}$, then $\overline{\text{CSYNC}}$ is also driven active low.

In addition, for interlaced composite video, HCOUNT is reset to 0 when HCOUNT=HTOTAL/2 during the equalization and serration regions. This condition triggers the equalization and serration pulses on the $\overline{\text{CSYNC}}$ pin (which occur every half horizontal scan line). During this time, the beginning of horizontal-sync pulses on the $\overline{\text{HSYNC}}$ pin are caused by alternating occurrences of HCOUNT=HTOTAL/2.

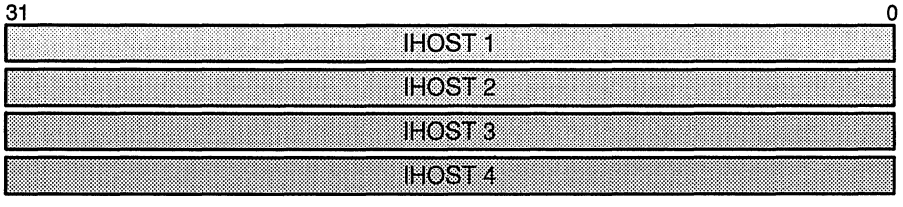
In interlaced video, HTOTAL should contain an odd number (LSB=1) to achieve equal spacing between lines. Equalization and serration pulses are then evenly separated by half a scan line. The total number of VCLKs per horizontal scan line is calculated as HTOTAL + 1.

When external horizontal or composite video is enabled, HTOTAL should contain a value not less than HCOUNT's value at the point at which the external sync pulse is expected. If you use SETHCNT to exactly align the internal video timing with the external sync, set HTOTAL to exactly match the number of VCLKs between external syncs.

HTOTAL should contain a 16-bit value greater than HSBLNK but less than or equal to 65,535 ($FFFF_{16}$).

B-file register?		register number:
I/O register?	√	**addresses: C000 0380h** **C000 03A0h** **C000 03C0h** **C000 03E0h**

Format



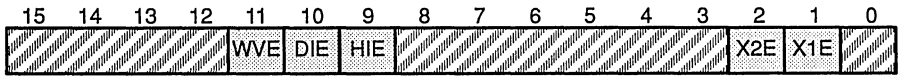
Description

The TMS34020's host interface uses these 32-bit locations for storing the addresses used in making host-requested reads, writes, and prefetches to TMS34020 local memory. These locations are included in the I/O register memory space for purposes of chip test only. You cannot write to these locations. The data read from these locations is generally not 0.

B-file register? ☐ register number:

I/O register? √ **address: C000 0110h**

Format



Bits

Bits	Name	Function
1	X1E	Enables external interrupt 1
2	X2E	Enables external interrupt 2
9	HIE	Enables the host interrupt
10	DIE	Enables the display interrupt
11	WVE	Enables the window violation interrupt
0 3—8 12—15		Reserved; do not use

Description

INTENB contains an interrupt mask that selectively enables 3 internally generated and 2 externally generated interrupt requests.

X1 & X2 *external interrupts 1 and 2.* Generated by active-low signals on the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ input pins, respectively.

HI *host interrupt.* Generated when the host processor sets INTIN[[HSTCTLL]] to 1.

DI *display interrupt.* Generated when the vertical count in the VCOUNT register reaches the value of DPYINT.

WV *window-violation interrupt.* Caused by an attempt to write a pixel inside or outside the current window limits (depending on the selected window-checking mode).

The status register contains a global interrupt enable bit (IE). The INTENB register contains individual interrupt enable bits associated with each of the 5 interrupts described above. Interrupts are enabled through a 2-step process:

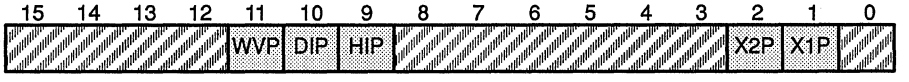
Step 1: Set the IE bit to 1.

Step 2: Set the appropriate bits in INTENB to 1.

Setting IE to 0 disables all of these interrupts, regardless of the value in INTENB. When IE=1, each interrupt is enabled according to the appropriate value in INTENB (1 enables the interrupt, 0 disables it).

B-file register?		register number:
I/O register?	√	**address: C000 0120h**

Format



Bits

Bits	Name	Function
1	X1P	Identifies a pending external interrupt 1
2	X2P	Identifies a pending external interrupt 2
9	HIP	Identifies a pending host interrupt
10	DIP	Identifies a pending display interrupt
11	WVP	Identifies a pending window violation interrupt
0, 3—8 12—15		Reserved; do not use

Description

INTPEND indicates which interrupt requests are currently pending (for a description of these interrupts, refer to the discussion of the INTENB register). The individual pending bits in the INTPEND register reflect the status of interrupt requests. The interrupt is requested if the corresponding pending bit is 1; there is no request if the pending bit is 0. INTPEND reflects the status of each interrupt request, regardless of whether the interrupt is enabled or not; this allows the TMS34020 to poll interrupts.

X1P and X2P are read-only bits that reflect the input levels on $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$; they are not affected when INTPEND is written to. $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ are asynchronous inputs, but signals to these pins are synchronized internally so that you can always reliably read X1P and X2P. If an external interrupt is disabled, the TMS34020 ignores the interrupt request, even if the corresponding pending flag is set. The TMS34020 takes the interrupt only if the external request is maintained at the request pin, until the interrupt is again enabled.

DIP and WVP reflect the status of interrupt requests generated internally. These 2 bits are implemented as latches. Once set, DIP or WVP remain set until a 0 is written to the appropriate bit (or until the TMS34020 is reset). Writing a 1 to either of these bits has no effect at any time.While an internal interrupt is disabled, the interrupt request is ignored, even if the corresponding pending flag is set. If the interrupt is then enabled while the interrupt-pending flag is set (because of a prior interrupt request), the TMS34020 takes the interrupt.

HIP is a read-only bit that always displays the current contents of INTIN. Writing to the INTPEND register does not affect HIP. A host interrupt request is generated when the host processor writes a 1 to INTIN. The TMS34020 clears the interrupt request by writing a 0 to INTIN[HSTCTLL].

B-file register?	√	**register number: B10**
I/O register?		address:

Format

Description

MADDR contains the mask array address for PIXBLT L,M,L. MADDR usually points to the mask bit with the lowest address in the mask array. When the selected starting corner is not the upper left corner, you must manually adjust MADDR to point to the mask array's starting corner.

MADDR always contains a linear address. When the PIXBLT L,M,L completes, MADDR points to the starting location of the row that follows the last row in the array. If PIXBLT L,M,L is interrupted, MADDR points to the next word of pixels to be read.

Which instructions use this register?

Instruction	MADDR's format
SETCMP	Linear; any value
PIXBLT L, M, L	Linear; any value

Example

```
MADDR   .set   B10

        MOVI   00010AFCh, MADDR        ; Move linear
                                       ; value 10AFCh
                                       ; into B10
```

B-file register?	√	**register number: B11**
I/O register?		address:

Format

Description

MPTCH defines the linear difference in the starting memory addresses of adjacent rows of the mask array for PIXBLT L,M,L. The TMS34020 uses the value in MPTCH to move from row to row through the mask array. MPTCH can have any value.

If you're manually converting an XY address to a linear address, you can use the SETCMP instruction; SETCMP uses the MPTCH value to calculate the mask pitch conversion factor and loads the correct value into CONVMP. You can then use CVMXYL to perform the conversion.

Which instructions use this register?

Instruction	MPTCH's format
CVMXYL	Linear; any value
FLINE, LINE	Linear; any value
FPIXEQ, FPIXNE	Linear; any value
PIXBLT L, M, L	Linear; any value
SETCMP	Linear; any value

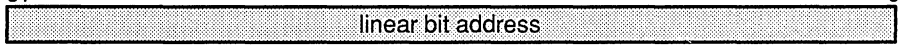
Example

```
MPTCH   .set   B11

        MOVI   00000100h, MPTCH   ; Power of 2
        MOVI   000A03D0h, MPTCH   ; Arbitrary value
        MOVI   00000220h, MPTCH   ; 2 powers of 2
                                  ; (512 + 32)
```

B-file register?	√	**register number: B4**	
I/O register?		address:	

Format



Description

OFFSET contains the linear address of the first pixel in the XY coordinate space for instructions using XY addressing. This address corresponds to the linear address of the XY origin (X=0,Y=0). The TMS34020 uses this value as the memory base for performing XY-to-linear address conversions.

OFFSET always contains a *linear* address. The offset address may be at any position in the TMS34020 linear address space. For proper XY address conversions, transparency, pixel processing, and plane masking, OFFSET should contain a pixel-aligned value. Instructions that use OFFSET as an implied operand do not modify the register's contents.

Which instructions use this register?

Instruction	OFFSET's function
CVXYL	Linear address of XY origin
DRAV	Linear address of XY origin
FILL XY	Linear address of XY origin
LINE	Linear address of XY origin
PFILL XY	Linear address of XY origin
PIXBLT B, XY	Linear address of XY origin
PIXBLT L, XY	Linear address of XY origin
PIXBLT XY, L	Linear address of XY origin
PIXBLT XY, XY	Linear address of XY origin
PIXBLT L, M, L	Linear address of XY origin
PIXT *Rs*, **Rd*.**XY**	Linear address of XY origin
PIXT **Rs*.**XY**, *Rd*	Linear address of XY origin
PIXT **Rs*.**XY**, **Rd*.**XY**	Linear address of XY origin
TFILL	Linear address of XY origin

Example

```
OFFSET .set  B4

       MOVI  00042000h, OFFSET ; Linear value on
                               ; pixel boundary
```

B-file register? √	**register number: B13**
I/O register?	address:

Format

Description

PATTERN uses the information in COLOR0 and COLOR1 to define a pixel pattern. COLOR0 defines the replacement color for 0 bits in the pattern; COLOR1 provides the replacement color for 0 bits in the pattern.

Note:

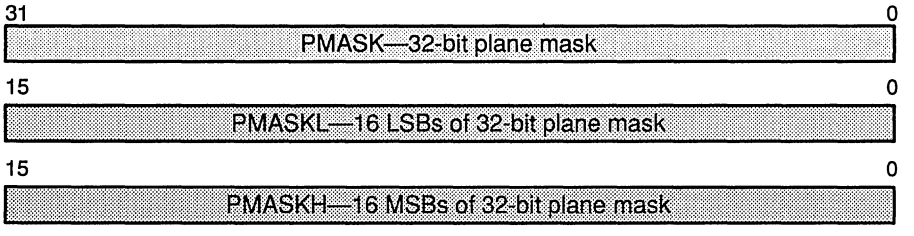
If the PATTERN value is less than 32 bits, you must replicate the pattern throughout all 32 bits of the PATTERN register.

Which instructions use this register?

Instruction	PATTERN's function
FLINE, LINE	Line pattern
PFILL XY	Array pattern. If PATTERN contains all 1s, PFILL uses COLOR1 to produce a solid fill. If PATTERN contains all 0s, PFILL uses the COLOR0 value to produce a solid fill.

B-file register?		register number:
I/O register?	√	*PMASK* **(32-bit address): C000 0160h** *PMASKL* **(16-bit address): C000 0160h** *PMASKH* **(16-bit address): C000 0170h**

Format



or

Note:

You can access the plane mask registers separately or together by using different addresses and different field sizes.

- To access PMASK as a single 32-bit register, access the 32-bit field at address C000 0160h.
- To access PMASKL as a 16-bit register, access the 16-bit field at address C000 0160h.
- To access PMASKH as a 16-bit register, access the 16-bit field at address C000 0170h.

Description

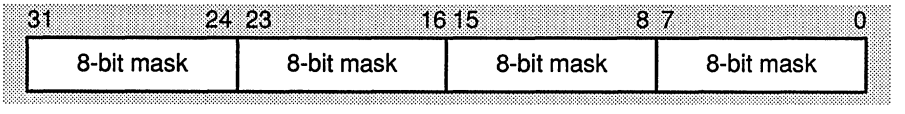
The PMASK registers selectively enable or disable various planes in the bitmap of a multiple-bit-per-pixel display system. Together, PMASKL and PMASKH (referred to as PMASK) contain a 32-bit value that determines which bits of each pixel can be modified during execution of a graphics instruction. The PMASK registers enable you to identify which bits in each pixel are protected (mask bit=1) or not protected (mask bit=0) from modification.

- During a pixel write, 0s in the plane mask identify destination bit positions that can be modified. The 1s in the plane mask represent bit positions within the destination that are protected from modification.
- During a pixel read, 0s in the plane mask identify readable bits within a pixel; bits corresponding to 1s in the mask are always read as 0s.

Display memory organization can be described in terms of bit planes. If the pixel size is 4 bits, for example, and the bits in each pixel are numbered from 0 to 3, the display memory is composed of 4 bit planes, numbered 0 to 3. Plane 0 contains all the bits numbered 0 from all the pixels, plane 1 contains all the bits numbered 1 from all the pixels, and so on. A 4-bit mask is constructed so that bit 0 of the mask enables or disables writes to the bits in plane 0, mask bit 1 enables or disables writes to plane 1, etc.

The plane-mask value for a 4-bit pixel is a 4-bit value; the plane mask for an 8-bit pixel is an 8-bit value, etc. You must replicate the plane mask throughout the 32 bits of the PMASK registers. For example, when PSIZE=8, you must load the PMASK registers with 4 identical copies of the 8-bit plane-mask value, as Figure 4–8 shows. In general, all 32 bits of the registers are used, and a mask for a pixel size of less than 32 bits must be duplicated n times (where n is 32 divided by the pixel size).

Figure 4–8. Replicating the Mask Value for an 8-Bit Pixel



The individual bits of the PMASK registers are associated with corresponding bits of the 32-bit LAD bus (data are multiplexed over the same LAD0—LAD31 pins as the address). PMASK register bit 0 corresponds to bit 0 of the data bus (the bit transferred on LAD0), PMASK bit 1 is associated with bit 1 of the data bus, etc. In general, if PMASK bit n is a 0, the mask enables bit n of the data bus; if PMASK bit n is a 1, the mask disables bit n.

You can effectively disable plane masking by loading all 0s into the PMASK registers; this allows all bits of each pixel to be modified. This is the default state of the PMASK registers following reset.

If your system's VRAMs can store a copy of the plane mask internally (the TMS44251 can do this), then you should set VEN[[CONFIG]]. The TMS34020 automatically detects when the PMASK registers are modified. It subsequently performs a special load-write-mask memory cycle to copy the 1s complement of the PMASK contents into the VRAMs' internal write-mask. (The PMASK is inverted because the meaning of the bits in a VRAM's write mask is opposite to the meaning of the bits in the PMASK.) The TMS34020 can use the VRAM copy of the plane mask to perform plane-masked writes without performing read-modify-write cycles.

Which instructions use this register?

Instruction	PMASK's function
DRAV	Plane-mask value for graphics operations
FILLs (both)	Plane-mask value for graphics operations
FLINE, LINE	Plane-mask value for graphics operations
FPIXEQ, FPIXNE	Plane-mask value for graphics operations
PIXBLTs (all)	Plane-mask value for graphics operations
PIXT *Rs*, **Rd* and **Rs*, **Rd*.**XY**	Plane-mask value for graphics operations
TFILL	Plane-mask value for graphics operations
VBLT	Plane-mask value for graphics operations
VFILL	Plane-mask value for graphics operations

B-file register?		register number:
I/O register?	√	**address: C000 0150h**

Format

Description

PSIZE defines the pixel size in bits. If the pixel size is 4, load PSIZE with the value 4; if the pixel size is 8, load PSIZE with 8, etc. All 16 bits of the PSIZE register can be written to or read. Legal pixel sizes are 1, 2, 4, 8, 16, and 32 bits; any other value of PSIZE is undefined.

PSIZE = **0001h**	Pixel size =	**1**	bit per pixel
PSIZE = **0002h**	Pixel size =	**2**	bits per pixel
PSIZE = **0004h**	Pixel size =	**4**	bits per pixel
PSIZE = **0008h**	Pixel size =	**8**	bits per pixel
PSIZE = **0010h**	Pixel size =	**16**	bits per pixel
PSIZE = **0020h**	Pixel size =	**32**	bits per pixel

Which instructions use this register?

Instruction	PSIZE's function
CVXYL	X shift amount for XY-to-linear conversion
CVDXYL	X shift amount for XY-to-linear conversion
CVMXYL	X shift amount for XY-to-linear conversion
CVSXYL	X shift amount for XY-to-linear conversion
DRAV	Pixel size for graphics operations
FILLs (both)	Pixel size for graphics operations
FLINE, LINE	Pixel size for graphics operations
FPIXEQ, FPIXNE	Pixel size for graphics operations
PIXBLTs (all)	Pixel size for graphics operations
PIXT *Rs*, **Rd*	Pixel size for graphics operations
PIXT **Rs*, **Rd*	Pixel size for graphics operations
PIXT *Rs*, **Rd*.**XY**	Pixel size for graphics operations
PIXT **Rs*.**XY**, **Rd*	Pixel size for graphics operations
PIXT **Rs*.**XY**, **Rd*.**XY**	Pixel size for graphics operations
RPIX	Field size for replication
TFILL	Pixel size for graphics operations
VBLT	Pixel size for graphics operations
VFILL	Pixel size for graphics operations

B-file register?		register number:
I/O register?	√	**address: C000 01F0h**

Format

Description

REFADR contains the address output during DRAM-refresh cycles. DRAMs require periodic refreshing to retain their data. The TMS34020 automatically generates $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ cycles that refresh the DRAMs at regular intervals. You can select the interval between refresh cycles by loading an appropriate value into RR0—RR2[[CONFIG]]. This determines how often, if at all, DRAM refreshes should be performed.

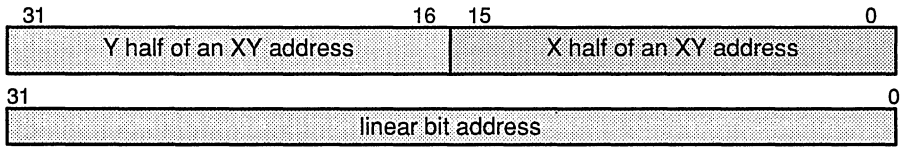
REFADR forms a contiguous binary counter. Each time a DRAM refresh is performed, the address in REFADR is output on both LAD16—LAD31 and RCA0—RCA12. RCM0—RCM1[[CONFIG]] determines which bits of the logical address appear on RCA0—RCA12 at row-address time. During a DRAM-refresh cycle, the address is valid on both LAD16—LAD31 and RCA0—RCA12 throughout the memory cycle. This memory cycle is 3 machine states long, allowing plenty of time for the external decode logic to detect the refresh (from the status code output on LAD3—LAD0) and then to enable the appropriate banks of memory for refresh. The refresh pseudo-address is incremented after each DRAM-refresh cycle that completes normally (that is, does not return the retry completion code on the LRDY and BUSFLT pins). If a refresh cycle does return a retry condition, the refresh cycle is automatically rescheduled and the same address is output.

You can use the refresh pseudo-address to determine which banks of memory will be refreshed. Or, you can use it as the refresh address required by DRAMs that support $\overline{\text{RAS}}$-only refresh. The TMS34020 does not directly support $\overline{\text{RAS}}$-only refresh; if you use $\overline{\text{RAS}}$-only refresh, you must use external hardware to prevent activation of the $\overline{\text{CAS}}$ strobes.

Reset clears the REFADR register to 0; no refreshes are performed while the $\overline{\text{RESET}}$ pin is held active low. However, if the $\overline{\text{RESET}}$ pin is held high while the TMS34020 is still executing reset internally, DRAM refreshes are performed. After $\overline{\text{RESET}}$ is taken high, no CPU-initiated memory cycles occur until 8 DRAM-refresh cycles are completed. This ensures that the DRAMs and VRAMs in the system are initialized correctly.

B-file register? √	**register number: B0**
I/O register?	address:

Format



or

Description

SADDR contains the source array address for PIXBLTs. SADDR usually points to the pixel with the lowest address in the source array. When the selected starting corner is not the upper left corner, the TMS34020 automatically adjusts SADDR to point to the selected starting corner of the source array. This feature allows you to handle overlapping arrays. (For PIXBLT L,L and PIXBLT L,M,L, however, you must manually adjust SADDR to point to the starting corner.)

The TMS34020 treats the address in SADDR as an XY address or a linear address, depending on the instruction using it.

During PIXBLT operations, SADDR is maintained in linear format. When the PIXBLT completes, SADDR points to the starting location of the row that follows the last row in the array. If a PIXBLT is interrupted, SADDR points to the next word of pixels to be read.

Which instructions use this register?

Instruction	SADDR's format and function
BLMOVE	Linear; points to the beginning of the source array
FLINE, LINE	Decision variable $d = 2b - a$, used for the line draw
PIXBLT B, L PIXBLT B, XY	Linear; points to the beginning of the binary source array
PIXBLT L, L PIXBLT L,M,L	Linear with special requirements when PBH = 1 or PBV=1; refer to the PIXBLT L,L discussion for a description of its unique requirements
PIXBLT L, XY	Linear; points to the beginning of the source array
PIXBLT XY, L PIXBLT XY,XY	XY; points to the beginning of the source array
TFILL	XY; points to the first pixel in the line
VBLT	Linear; points to the beginning of the source array

Example

```
SADDR   .set   B0

        MOVI   [08h, 015h], SADDR      ; Move XY value
                                       ; 15h,8h into B0
        MOVI   0000AAAAh, SADDR        ; Move linear value
                                       ; AAAAh into B0
```

B-file register?		register number:
I/O register?	√	**address: C000 02C0h**

Format

Description

During horizontal-blanking screen-refresh cycles, the video timing logic automatically loads SCOUNT with the VRAM tap point (determined using the value in DPYMSK). The tap point is automatically right-justified before it is loaded into SCOUNT. SCOUNT is incremented by the rising edge of a pulse on the SCLK pin. You should connect SCLK to the VRAM's serial clock signal so that SCOUNT is incremented each time a bit of data is shifted out of the VRAM's serial register. This means that SCOUNT always contains the tap point of the bit most recently shifted. When the VRAMs shift the last bit out of one half serial register and start shifting bits from the other half serial register, SCOUNT overflows from all 1s to 0 and schedules a midline-reload screen refresh to transfer the next half-row of VRAM into the half serial register not being shifted out.

Hold the SCLK pin at the inactive-low level throughout horizontal and vertical blanking, when the VRAM serial registers are not shifting data.

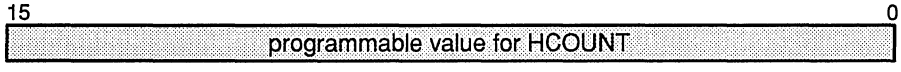
Two separate asynchronous elements of the TMS34020 internal logic can access SCOUNT:

- The midline-reload timing control logic, which runs synchronously to SCLK, increments SCOUNT during the active regions of the screen (when neither vertical nor horizontal blanking is active).
- The internal processor, which runs synchronously to the local clocks LCLK1 and LCLK2, can access SCOUNT as an I/O register and can load it with the VRAM tap point during horizontal blanking.

No synchronization between these subsystems is provided. SCOUNT can be reliably read or written only while SCLK is held at the logic-low level. SCOUNT is not typically read or written except during chip test.

B-file register?		register number:
I/O register?	√	**address: C000 0310h**

Format



Description

If external horizontal or composite video is enabled (by clearing the HSD or CSD bit, respectively, in DPYCTL), the video timing logic loads the value of SETHCNT into HCOUNT when

- the logic detects an external horizontal-sync pulse on $\overline{\text{HSYNC}}$, or
- the logic detects an external composite-sync pulse on $\overline{\text{CSYNC}}$.

Setting HCOUNT to a programmable value rather than clearing it counteracts delays inherent in the synchronization of external sync pulses and other external system delays.

It takes 4 VCLK cycles from the time an external sync is detected at the appropriate sync input pin until its effects propagate to the video output pins. If SETHCNT contains the value 4, then HCOUNT is set to 4 at the beginning of the fifth VCLK cycle (that is, 4 whole VCLK cycles) after the external sync pulse is detected at the pins. This has the same effect as if HCOUNT were cleared to 0 in the same VCLK cycle that the external sync signal went active low. If the HTOTAL value matches the parameters of the external video source, then HCOUNT=HTOTAL coincides with the beginning of the next external sync pulse. This condition causes HCOUNT to be loaded with 0, and the internal horizontal- and composite-sync pulses begin. As a result, any sync pins configured as outputs go active low on the same VCLK cycle as the external sync signal, and internally generated video signals are synchronous to and aligned with the external video signals.

This is especially useful if the TMS34020 is performing sync conversion. By programming all the video timing registers to match the parameters of the external video source, an external composite sync can be used to generate horizontal- and vertical-sync outputs, or external horizontal and composite syncs can be used to generate a composite-sync output.

Programming SETHCNT to a value greater than 4 causes the sync pins configured as outputs to change *SETHCNT*–4 VCLK cycles before the external sync (or syncs) changes. This is useful if the system contains external clock skews. Similarly, programming SETHCNT to a value less than 4 causes the sync pins configured as outputs to change after the external sync (or syncs). Do not set SETHCNT to a value less than 0.

B-file register?	☐	register number:
I/O register?	√	**address: C000 0300h**

Format

Description

If external vertical or composite video is enabled (by clearing the VSD or CSD bit, respectively, in DPYCTL), the video timing logic loads the value of SETVCNT into VCOUNT when

- the logic detects an external vertical-sync pulse on $\overline{\text{VSYNC}}$, or
- the logic detects the first serration pulse in the external composite-sync signal on $\overline{\text{CSYNC}}$.

SETVCNT provides symmetry with the SETHCNT register. If you are using interlaced video, program SETVCNT to 0. If you do not, the TMS34020 will not be able to distinguish between external odd and even fields and may not correctly synchronize to the external source.

For noninterlaced video, you can set SETVCNT to nonzero values. This causes the internal vertical-blanking signal (visible externally on $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$) and the internal vertical-sync signal (visible externally on $\overline{\text{VSYNC}}$ if it is an output) to start and end an integral number of scan lines ahead of the external vertical sync and blanking signals. Programming SETVCNT to *n* causes the internal signals to be *n* scan lines in advance of the external signals.

B-file register? √	**register number: B1**
I/O register?	address:

Format

Description

SPTCH defines the linear difference between the starting addresses of adjacent rows of a source array. The TMS34020 uses the value in SPTCH to move from row to row through the source array. SPTCH can have any value that is a multiple of the current pixel size. Note that XY-to-linear conversion is most efficient when SPTCH is a power of 2.

If you're manually converting an XY address to a linear address, you can use the SETCSP instruction; SETCSP uses the SPTCH value to calculate the source pitch conversion factor and loads the correct value into CONVSP. You can then use CVSXYL to perform the conversion.

Instruction	SPTCH's format
CVSXYL	Linear; any value
PIXBLTs (all)	Linear; any value
PIXT **Rs.XY*, *Rd*	Linear; any value
PIXT **Rs.XY*, **Rd.XY*	Linear; any value
SETCSP	Linear; any value

Example

```
SPTCH   .set   B1

        MOVI   00001000h, SPTCH    ; Power of 2
        MOVI   00000900h, SPTCH    ; 2 powers of 2
        MOVI   00010AFCh, SPTCH    ; Arbitrary value
```

B-file register?		register number:
I/O register?	√	**address: C000 01C0h**

Format

Description

VCOUNT counts the horizontal lines in the display, and is used for generating vertical and composite sync and blanking signals. VCOUNT increments on the clock edge that resets HCOUNT; in interlaced video, VCOUNT also increments on the clock edge after HCOUNT=HTOTAL/2. This causes $\overline{\text{VSYNC}}$'s falling and rising edges to coincide with the falling edge of $\overline{\text{CSYNC}}$ or $\overline{\text{HSYNC}}$ (in interlaced video, some $\overline{\text{VSYNC}}$ transitions occur halfway between two $\overline{\text{HSYNC}}$ pulses). If $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$ is selected as $\overline{\text{VBLNK}}$, this also applies to $\overline{\text{VBLNK}}$.

To generate vertical sync and blanking signals, the video timing logic compares VCOUNT to the values of VEBLNK, VSBLNK, VTOTAL, and VESYNC/2. In interlaced composite video (when $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ is selected as $\overline{\text{CSYNC}}$), the full value of VESYNC is used to determine the end of the second equalization region. When HCOUNT=HTOTAL (or HCOUNT= HTOTAL/2 in interlaced video) and VCOUNT=VTOTAL simultaneously, VCOUNT is reset to 0 on the next falling edge and $\overline{\text{VSYNC}}$ is driven active low. In external vertical- or composite-sync video, VCOUNT is reloaded from SETVCNT when a falling edge is detected on $\overline{\text{VSYNC}}$, or when the first serration pulse is detected on $\overline{\text{CSYNC}}$.

When $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ is selected as $\overline{\text{CSYNC}}$, VCOUNT determines the type of pulse output on this pin. If VESYNC<VCOUNT≤ VSBLNK, then $\overline{\text{CSYNC}}$ outputs ordinary horizontal-sync pulses that coincide with $\overline{\text{HSYNC}}$. In interlaced video, if VSBLNK<VCOUNT≤VTOTAL or VESYNC/2<VCOUNT≤ VESYNC, equalization pulses are generated on $\overline{\text{CSYNC}}$. Equalization pulses appear every half line and are half the width of the pulses on $\overline{\text{HSYNC}}$. Every other pulse begins coincident with a pulse on $\overline{\text{HSYNC}}$.

If VTOTAL≤VESYNC/2, $\overline{\text{CSYNC}}$ generates serration pulses. HESERR defines the length of these pulses. In noninterlaced video, they always begin coincident with a pulse on $\overline{\text{HESYNC}}$. In interlaced video, they occur every half line, so every other pulse begins coincident with a pulse on $\overline{\text{HSYNC}}$.

A display interrupt is generated when VCOUNT=DPYINT. You can use this to coordinate software activity with the refreshing of selected lines on the screen.

Two separate, asynchronous elements of the TMS34020 internal logic can access VCOUNT:

- The video timing control logic (which runs synchronously to VCLK) increments and clears or reloads VCOUNT while generating sync and blanking signals.

- The internal processor (which runs synchronously to LCLK1 and LCLK2) can access VCOUNT as an I/O register.

The TMS34020 provides no synchronization between these subsystems. VCOUNT can be reliably read or written only while VCLK is held at the logic-low level. VCOUNT is not typically read or written except during chip test.

B-file register?		register number:
I/O register?	√	**address: C000 0020h**

Format

Description

VEBLNK is compared to VCOUNT to determine when the vertical-blanking interval ends. The vertical-blanking interval ends when VCOUNT=VEBLNK and either of these conditions is satisfied:

- HCOUNT=HTOTAL or
- HCOUNT=HTOTAL/2 in the interlaced even field.

In separate sync, the vertical-blanking signal is output on the $\overline{\text{CBLNK}}/\overline{\text{VBLNK}}$ pin, which is selected as $\overline{\text{VBLNK}}$. In composite sync, $\overline{\text{CBLNK}}/\overline{\text{VBLNK}}$ is selected as $\overline{\text{CBLNK}}$; in this case, $\overline{\text{CBLNK}}$ outputs the logical-OR of the internal horizontal- and vertical-blanking signals. $\overline{\text{CBLNK}}$ is low if either horizontal- or vertical-blanking is active internally.

Monitors typically require VEBLNK to contain a value less than VSBLNK and greater than VESYNC/2.

B-file register?		register number:
I/O register?	√	**address: C000 0000h**

Format

Description

VESYNC is compared to VCOUNT to determine when the vertical-sync pulse ends and, in interlaced composite video, when the second equalization region ends. The sync pulse ends when VCOUNT= VESYNC/2 and either of these conditions is satisfied:

- HCOUNT=HTOTAL or
- HCOUNT=HTOTAL/2 in interlaced video.

The $\overline{\text{VSYNC}}$ output is driven inactive high to signal the end of the vertical-sync interval.

In interlaced video, the second serration region ends if both of these conditions are satisfied:

- VCOUNT=VESYNC and
- HCOUNT=HTOTAL/2.

Monitors typically require VESYNC/2 to contain a value less than VEBLNK; VESYNC's minimum value is 0.

In external vertical or composite video, you should load VESYNC with a value such that the internal vertical-sync pulse ends before or at the same time as the end of the external vertical-sync pulse or the end of the external serration region. If the external vertical-sync pulse or the external serration region is still active when the internal vertical-sync pulse ends, it causes VCOUNT to be reloaded from SETVCNT, and the internal vertical-sync interval starts again.

B-file register?		register number:
I/O register?	√	**address: C000 0040h**

Format

Description

VSBLNK is compared to VCOUNT to determine when the vertical-blanking interval starts. Vertical blanking starts when VCOUNT=VSBLNK and either of these conditions is satisfied:

- HCOUNT=HTOTAL or
- HCOUNT=HTOTAL/2 in the interlaced odd field.

Additionally, vertical blanking will start if the video timing logic detects an external composite-sync pulse (when $\overline{\text{CSYNC}}$ is an input) or a horizontal-sync pulse (when $\overline{\text{HSYNC}}$ is an input).

In separate sync, the vertical-blanking signal is output on the $\overline{\text{CBLNK}}/\overline{\text{VBLNK}}$ pin, which is selected as $\overline{\text{VBLNK}}$. In composite sync, $\overline{\text{CBLNK}}/\overline{\text{VBLNK}}$ is selected as $\overline{\text{CBLNK}}$; in this case, $\overline{\text{CBLNK}}$ outputs the logical-OR of the internal horizontal- and vertical-blanking signals. $\overline{\text{CBLNK}}$ is low if either horizontal- or vertical-blanking is active internally.

Monitors typically require VSBLNK to contain a value less than VTOTAL and greater than VEBLNK.

B-file register?		register number:
I/O register?	√	**address: C000 0060h**

Format

Description

VTOTAL defines the time at which the vertical-sync pulse begins. The video timing logic compares VTOTAL to VCOUNT to determine when to start the vertical-sync pulse. The vertical-sync pulse starts and VCOUNT is reset to 0 when VCOUNT=VTOTAL and either of these conditions is satisfied:

- HCOUNT=HTOTAL or
- HCOUNT=HTOTAL/2 in the interlaced video.

The internal vertical-sync pulse begins (if it was not already caused to do so by either of the conditions above) when

- the video timing logic detects an external vertical-sync pulse (when $\overline{\text{VSYNC}}$ is an input) or
- the video timing logic detects the first external composite serration pulse (when $\overline{\text{CSYNC}}$ is an input).

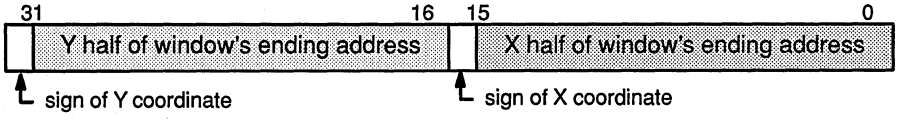
When this happens, VCOUNT is reloaded from the SETVCNT register.

The $\overline{\text{VSYNC}}$ output is driven low to signal the start of the vertical-sync interval. The falling and rising edges of $\overline{\text{VSYNC}}$ coincide with the falling edge of $\overline{\text{CSYNC}}$ or $\overline{\text{HSYNC}}$ (in interlaced video, some transitions of $\overline{\text{VSYNC}}$ occur halfway between 2 $\overline{\text{HSYNC}}$ pulses).

Set VTOTAL to a value greater than VSBLNK. VTOTAL's maximum value is 65,535.

B-file register?	√	**register number: B6**	
I/O register?		address:	

Format



Description

WEND defines the XY address of the most significant pixel within the rectangular destination clipping window. WEND must be valid for instructions that use an XY destination address and a nonzero window option. The most significant pixel is the pixel with the highest address within the window. For a screen with the origin in the top left corner of the pixel array, this address corresponds to the pixel in the lower right corner.

The X and Y portions of the address are signed values; WEND can be at any position in any quadrant of the XY address space. It describes an inclusive pixel; that is, the pixel at the XY location in WEND is included in the window.The value in WEND is used with WSTART, DADDR, and DYDX to preclip pixels, lines, and pixel arrays. WEND is not modified by instruction execution.

Which instructions use this register?

Instruction	WEND's function
CPW	XY address of most significant window corner
DRAV	XY address of most significant window corner
FILL XY	XY address of most significant window corner
FLINE, LINE	XY address of most significant window corner
PIXBLT B, XY	XY address of most significant window corner
PIXBLT L, XY	XY address of most significant window corner
PIXBLT XY, XY	XY address of most significant window corner
PIXT *Rs*, *Rd*.XY	XY address of most significant window corner
PIXT *Rs*.XY, *Rd*.XY	XY address of most significant window corner
TFILL	XY address of most significant window corner

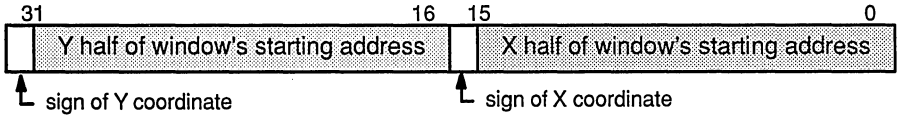
Example

```
WEND    .set   B6

        MOVI   [040h, 0100h], WEND  ; XY value (256,64)
                                    ; stored in WEND
```

B-file register? √	**register number: B7**
I/O register?	address:

Format



Description

WSTART defines the XY address of the least significant pixel within the rectangular destination clipping window. WSTART must be valid for instructions that use an XY destination address and a nonzero window option. The least significant pixel is the pixel with the lowest address in the array. For a screen with the origin in the top left corner of the pixel array, this address corresponds to the pixel in the upper left corner.

The X and Y portions of the address are signed values; WSTART can be at any position in any quadrant of the XY address space. It describes an inclusive pixel; that is, the pixel at the XY location in WSTART is included in the window. The value in WSTART is used with WEND, DADDR, and DYDX to preclip pixels, lines, and pixel arrays. WSTART is not modified by instruction execution.

Which instructions use this register?

Instruction	WSTART's function
CPW	XY address of least significant window corner
DRAV	XY address of least significant window corner
FILL XY	XY address of least significant window corner
FLINE, LINE	XY address of least significant window corner
PIXBLT B, XY	XY address of least significant window corner
PIXBLT L, XY	XY address of least significant window corner
PIXBLT XY, XY	XY address of least significant window corner
PIXT *Rs*, *Rd*.XY	XY address of least significant window corner
PIXT *Rs*.XY, *Rd*.XY	XY address of least significant window corner
TFILL	XY address of least significant window corner

Example

```
WSTART .set  B5

       MOVI  [040h, 0100h], WSTART ; XY value (256,64)
                                   ; stored in WSTART
```

Chapter 5

Instruction Cache and Internal Parallelism

Most program execution time is spent on repeatedly executing a few main procedures or loops. Program execution can be speeded up by placing these often-used code segments into a fast memory. The TMS34020 uses a 512-byte, on-chip **instruction cache** for this purpose.

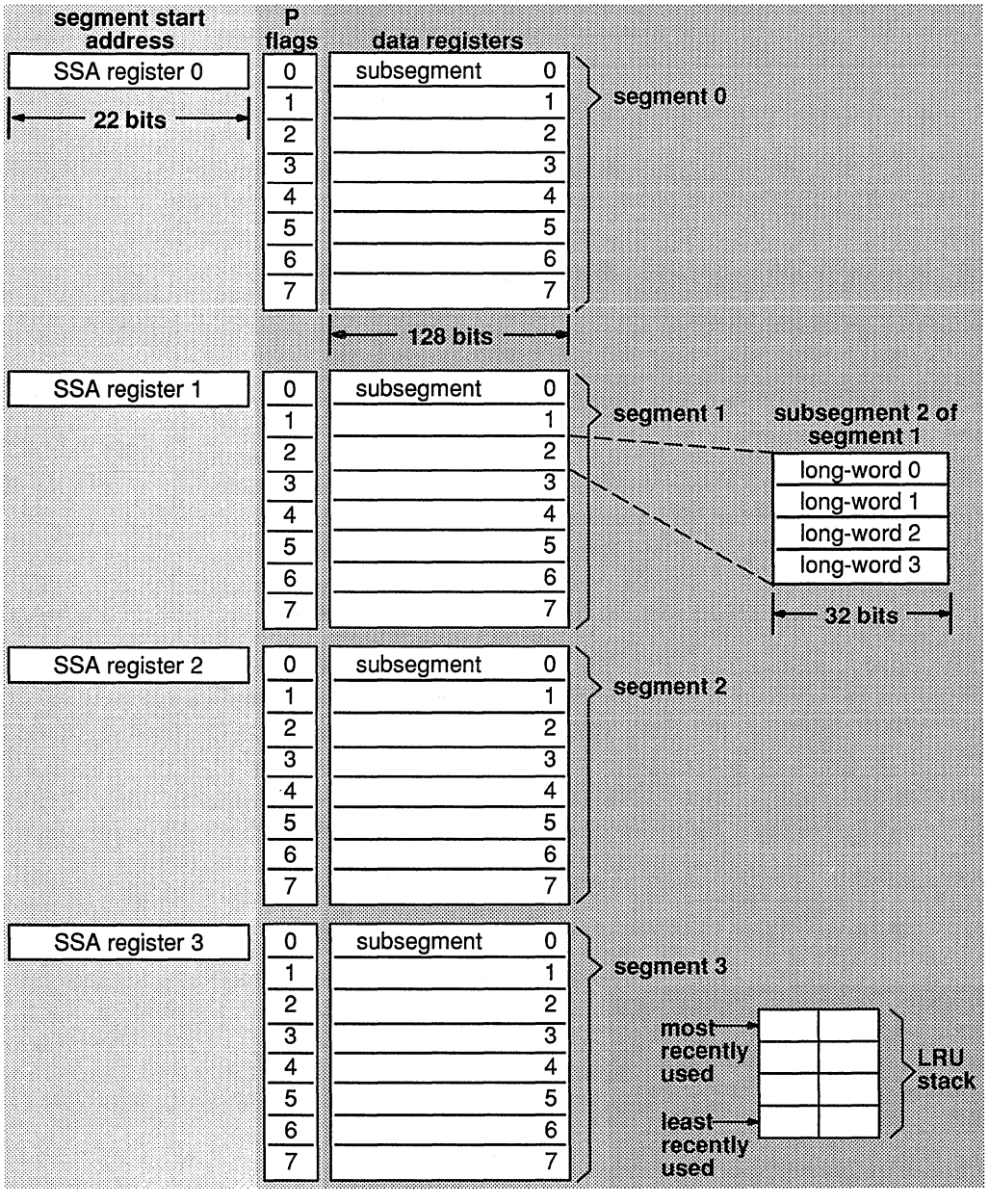
To further enhance execution speed, the TMS34020 can access several areas of memory, including the cache, in parallel. Although the TMS34020 stores code and data in a single memory space, the TMS34020's internal parallelism provides benefits that are often found in processors that use separate code and data spaces.

This chapter includes the following topics:

5.1 Cache Architecture

Figure 5–1 illustrates cache organization.

Figure 5–1. TMS34020 Instruction Cache



Only instruction words (memory words that the PC points to) can be accessed from the cache. This includes

- Opcodes
- Immediate operands
- Displacements
- Absolute addresses

Instructions and data can reside in the same area of memory; therefore, data may occasionally be copied into the instruction cache along with instruction words. However, the TMS34020 cannot access data from the cache; all reads and writes of data in memory bypass the cache.

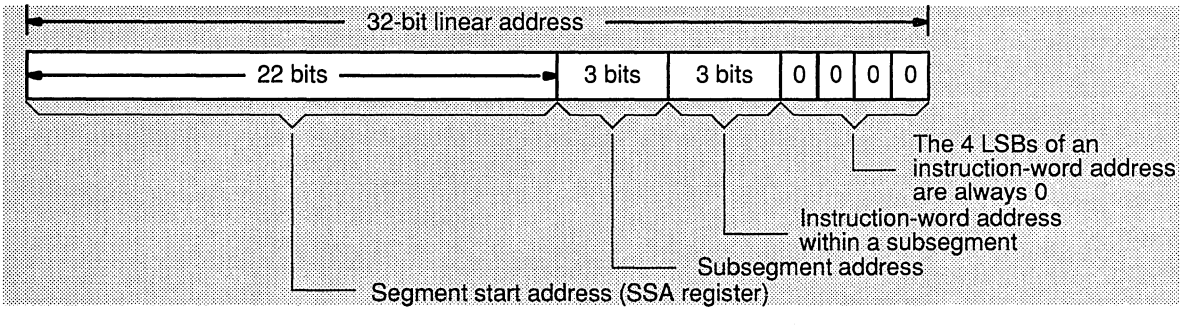
The instruction cache contains 512 bytes of RAM for storing up to 128 long (32-bit) words (this is equivalent to 256 6-bit instruction words). Each instruction word in cache is aligned on an even long-word boundary.

As Figure 5–1 shows, the cache is divided into four 64-word segments. Each cache segment may contain up to 64 instruction words of a 64-word segment from memory. This memory segment is a block of 64 contiguous instruction words, beginning at an even 64-word boundary in memory.

Each cache segment is further divided into 8 subsegments; each subsegment contains 4 long-words (up to 8 instruction words). Dividing each segment into subsegments reduces the number of word fetches required from memory when fewer than 64 words of a memory segment are used.

Each of the 4 cache segments is associated with a segment start address (**SSA**) register. Figure 5–2 shows how a long-word is partitioned into the components used by the cache-control algorithm.

Figure 5–2. Segment Start Address



The 22 bits of the SSA register correspond to the 22 MSBs of the segment's memory address. These 22 MSBs are common to all 8 subsegments within a segment. The next three bits (bits 6—8) identify one of the 8 subsegments. Bits 4, 5, and 6 identify one of the 8 instruction words within a subsegment. The 4 LSBs are always 0s because instructions are aligned on word boundaries.

5.2 Cache Replacement Algorithm

When the TMS34020 requests an instruction word from a segment that is not in the cache, the contents of one of the 4 cache-resident segments must be discarded to make room for the segment that contains the requested word. A modified form of the least-recently-used (**LRU**) replacement algorithm is used to select the segment to be discarded.

The LRU segment manager (part of the cache control logic) maintains an LRU stack to track use of the 4 segments. The LRU stack contains a queue of segment numbers, 0 through 3. Each time a segment is accessed, its segment number is moved to the top of the stack, pushing the other segment numbers down as necessary to make room at the top. Thus, the number at the top of the LRU stack identifies the most-recently-used segment and the number at the bottom identifies the least-recently-used segment.

When a new segment must be loaded into cache, the least-recently-used segment is discarded. The 8 P flags (described in Section 5.3) of the selected segment are set to 0s, and the segment's SSA register is loaded with the new segment address. After the requested subsegment is loaded from memory, its P flag is set to 1, and the requested instruction fetch is allowed to complete.

Following reset, all P flags in the cache are set to 0.

5.3 Cache Operation

When the TMS34020 requests an instruction word, it checks to see if the cache contains the word. First, the processor compares the 22 MSBs of the instruction address to the 4 SSA registers. If the TMS34020 fins a match, it searches for the appropriate subsegment. A present (**P**) flag, associated with each subsegment, indicates the presence of a particular subsegment within a cache segment:

- P=1 indicates that the requested instruction word is in cache. This is called a **cache hit**.
- If there is no match, or if there is a match and P=0, the instruction word is not in cache. This is called a **cache miss**.

5.3.1 Cache Hits

A **cache hit** means that the cache contains the requested instruction word. In this case, the TMS34020 performs the following actions:

1) Performs a short (one machine state) access cycle to read the instruction word from cache.
2) Moves the segment number to the top of the LRU stack, pushing the other three segment numbers toward the bottom of the stack (assuming that this segment was not the most recently used segment).

Because of pipelining, instruction fetches from the cache overlap completion of preceding instructions. Thus, the overhead due to instruction fetches is effectively 0.

5.3.2 Cache Misses

A **cache miss** means that the cache does not contain the instruction word. There are two types of cache miss—subsegment miss and segment miss.

- **Subsegment miss**

  The 22 MSBs of the instruction-word address match one of the 4 SSA registers' 22 MSBs; that is, the appropriate segment is in the cache. However, the P flag for the requested subsegment is not set. The TMS34020 performs these actions:

  1) Reads into cache the 8-instruction-word subsegment that contains the requested instruction word.
  2) Moves the segment number to the top of the LRU stack, pushing the other three segment numbers toward the bottom of the stack (assuming that this segment was not the most recently used segment).

3) Sets the subsegment's P flag to 1.

4) Reads the instruction word from the cache.

❑ **Segment miss**

The instruction word address does not match any of the SSA registers. The TMS34020 performs the following actions:

1) Selects the least-recently-used segment for replacement; clears the P flags of all 8 subsegments.

2) Loads the SSA register for the selected segment with the 22 MSBs of the address of the requested instruction word.

3) Reads into cache the 8-instruction-word subsegment in memory that contains the requested instruction word. This word is placed in the appropriate subsegment of the least-recently-used segment. The TMS34020 sets the subsegment's P flag to 1.

4) Adjusts the LRU stack by moving the number of the new segment from the bottom (indicating that it is least recently used) to the top (indicating that it is most recently used). This pushes the other three segment numbers in the stack down one position.

5) Reads the instruction word from the cache.

5.3.3 Fetching Data into the Cache Following a Cache Miss

Following either type of cache miss, the TMS34020 loads 4 long-words into a cache subsegment. The order in which the TMS34020 fetches these long-words is determined by the position, within the 4 long-words, of the opcode or immediate data that caused the cache miss.

Example 5–1. Code Without Branches or Immediate Data

Consider code that starts at address 0 and continues to a high address (such as 010000h). Assume that this code contains no loops or immediate data. When the TMS34020 begins to execute this code, it jumps to the first opcode (at address 0) and finds that the opcode is not in cache. So, it fills the first subsegment with the 8 opcodes that are in the first 4 long-words in memory. The TMS34020 reads the data in this order:

1st read: 32 bits at address 020h (opcodes 2&3) **2nd read:** 32 bits at address 040h (ops 4&5)
3rd read: 32 bits at address 060h (opcodes 6&7) **4th read:** 32 bits at address 000h (ops 0&1)

Note that the TMS34020 does not read the words in the expected cyclic order—it reads the long-word with the first opcode *last*, not first. This is the general case.

Example 5–2. Code with Branches

As another example, consider the following code segment.

```
0001 00000000     5600                clr     a0
0002 00000010     5684                clr     a4
0003 00000020     56a5                clr     a5
0004 00000030     56c6                clr     a6
0005 00000040     c000                jruc    next_subseg+48
     00000050     0005
0006 00000060     0300                nop
0007 00000070     0300                nop
0008 00000080           next_subseg:
0009 00000080     0300                nop
0010 00000090     0300                nop
0011 000000a0     0300                nop
0012 000000b0     5673                clr     b3
0013 000000c0     5694                clr     b4
0014 000000d0     56b5                clr     b5
0015 000000e0     56d6                clr     b6
0016 000000f0     56f7                clr     b7
0017 00000100           loop:
0018 00000100     c0ff                jruc    loop
```

This example jumps from the middle of the first subsegment to the middle of the second subsegment. The first subsegment is loaded into cache as described in Example 5–1. The code executes until the TMS34020 encounters the jump on line 5. At this point, control passes to the opcode at address 0B0h. This opcode is not in cache, so the TMS34020 loads the next subsegment. The 4 long-words are loaded in cyclic order; the long-word containing the opcode at address 0B0h is read last. The order of memory accesses is

1st read: 32 bits at address 0C0h **2nd read:** 32 bits at address 0E0h
3rd read: 32 bits at address 080h **4th read:** 32 bits at address 0A0h

Even though the code jumps over the long-word at address 080h, this word is loaded into cache.

Example 5–3. Code with Immediate Data

Some instructions have immediate data; for example,

```
movi   0ABCDABCDh,A0                ; (32 bits of
                                    ; immediate data)
move   @0FFFFFF20h,@0EEEEEE00,0     ; (64 bits of
                                    ; immediate data)
```

Immediate data follows the opcode in the object code. If an opcode with immediate data is near the end of a subsegment, the TMS34020 may encounter a cache miss when it attempts to access the immediate data. The next subsegment is loaded with 4 long-words, in cyclic order, so that the long-word containing the immediate data that caused the first cache miss is loaded in last.

5.3.4 Self-Modifying Code

Avoid using self-modifying code; it can cause unpredictable results. When a program modifies its own instructions, only the copy of the instruction that resides in external memory is affected. Copies of the instructions that reside in cache are not modified, and the TMS34020 doesn't attempt to detect this.

5.3.5 Flushing the Cache

Flushing the cache sets it to an initial state, identical to the state of the cache following reset: The cache is empty and all 32 P flags are cleared to 0.

You can flush the cache by setting the CF (cache flush) bit in the HSTCTLH register to 1. The CF bit retains the last value loaded until a new value is loaded or until the TMS34020 is reset. The contents of the cache remain flushed as long as the CF bit equals 1. All instruction fetches bypass the cache and are accessed directly from memory.

Unless the cache is disabled, normal cache operation resumes when the CF bit is cleared to 0.

One use for flushing the cache is to facilitate downloading new code from a host processor to TMS34020 local memory. The host typically halts the TMS34020 during downloading by writing a 1 to HLT[[HSTCTLH]]. Before allowing the TMS34020 to execute downloaded code, the host should flush the cache to purge it of stale instructions.

For performance reasons, CF[[CONTROL]] should not remain set to 1 for long periods. While CF=1, only 1 word is fetched at a time.

5.3.6 Disabling the Cache

Disabling the cache facilitates program debugging and emulation. The cache is disabled by setting CD[[CONTROL]] to 1. While the cache is disabled, the TMS34020 bypasses the cache and fetches all instructions from external memory.

Setting CD to1 is similar to setting CF to 1. However, when CD=1 and CF=0, data already in the cache is protected from change. When the CD bit is cleared to 0, the prior state of the cache (before CD was set to 1) is restored. The instructions in the cache are once again available for execution. If the cache contents become invalid while CD=1, they can be flushed by setting CF to 1.

For faster execution in some time-critical applications, you may wish to manipulate the CD bit to preserve code in the cache . For example, if an inner loop just exceeds 512 bytes, most of the loop, but not all of it, can fit in the cache. During execution of the few instructions that are not in the cache, you can set the CD bit to 1 to prevent the TMS34020 from replacing the code in the cache. In this instance, the loop's execution speed is improved by eliminating the thrashing of cache contents. Use this technique carefully; in some cases, it can negatively affect execution speed.

5.4 Performance with Cache Enabled vs. Cache Disabled

When the instruction cache is disabled, the TMS34020 fetches instruction words from external memory. Assuming no wait states are necessary, each instruction fetch from external memory adds 3 machine cycles to the access time. This is considerably slower than a program that uses the cache efficiently (when each word in cache is used several times before it is replaced).

A less efficient use of cache occurs when words in cache are used only once before they are replaced. This produces a cache miss every eighth word (even in this case, operation is usually much better than operation when the cache is disabled). With the cache enabled, the time penalty due to cache misses in this case is .75 machine states per single-word instruction (compare this to 3 states when the cache is disabled), which is calculated as follows:

- 5 machine cycles are required to load 4 long-words (8 instruction words) into cache from memory (in page mode).
- 1 additional machine state is required to begin processing the instruction.
- Dividing the total of 6 machine states by 8 instruction words yields an average of .75 machine states per instruction word.

Performance with the cache enabled is nearly always better than performance with the cache disabled. There are two exceptions:

- If the code contains many jumps, only a portion of each subsegment may be executed before control is transferred to another subsegment.
- If an inner loop is larger than the cache, only a portion of the instructions in the inner loop can be contained in the cache at any time. In this situation, you can improve performance by manipulating the CD bit as described in Section 5.3.6.

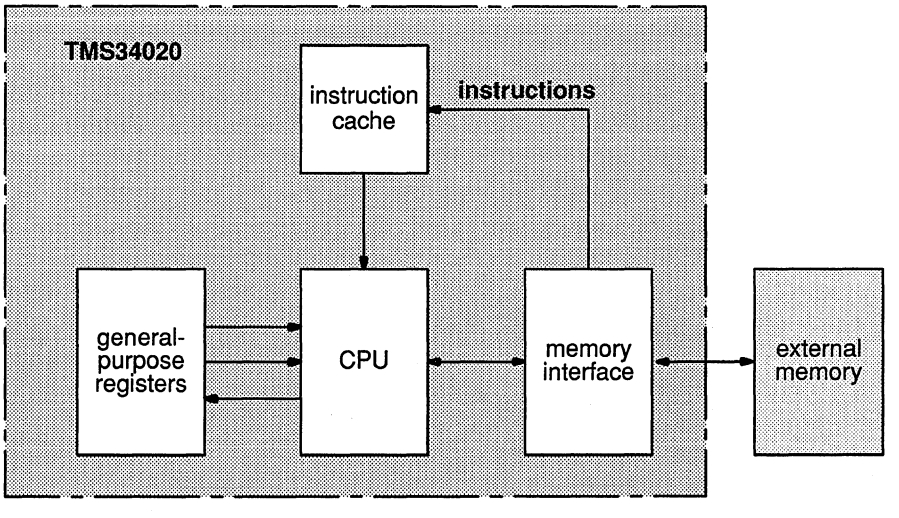
While the cache is disabled, the TMS34020's internal memory controller fetches each instruction word from memory only as it is requested by the CPU. This differs from operation with the cache enabled, in which case a cache miss causes the entire 8-instruction-word subsegment containing the requested instruction word to be loaded into the cache at once.

5.5 Internal Parallelism

Figure 5–3 illustrates the internal data paths associated with TMS34020 processor functions. The TMS34020 uses a single, logical memory space for storing both data and instructions. However, internal parallelism provides the TMS34020 with the benefits found in architectures that use separate data and instruction storage (sometimes referred to as *Harvard architectures*). The ability to fetch instructions from cache in parallel with accessing data from memory greatly enhances execution speed. Hardware parallelism allows the TMS34020 to access these three storage areas simultaneously:

- Instruction cache
- Dual-ported, general-purpose register files A and B
- External memory

Figure 5–3. Internal Data Paths



The TMS34020 can access each storage area independently of the other two. This allows the TMS34020 to perform the following actions in parallel during a pair of machine states:

- 1 external memory cycle
- 2 instruction fetches from cache
- 4 reads and 2 writes to the general-purpose register files

The need to schedule conflicting internal operations can limit the TMS34020's ability to perform these actions in parallel. For example, an instruction that requires the memory controller to perform a read must finish executing before the next instruction can be executed. Figure 5–4 illustrates an example of internal parallelism.

Figure 5–4. Parallel Operation of Cache, Execution Unit, and Memory Interface

(a) Code

```
A   L1:   MOVE   *B1+, B10, 0    ; Get DELTAX
B         ADD    B10, B8         ; Adjust pixel pointer
C         PIXT   *B1, *B8        ; Draw next pixel
D         ADD    B0, B1          ; Add field size
E         DSJS   B11, L1         ; Loop N times
```

(b) Result

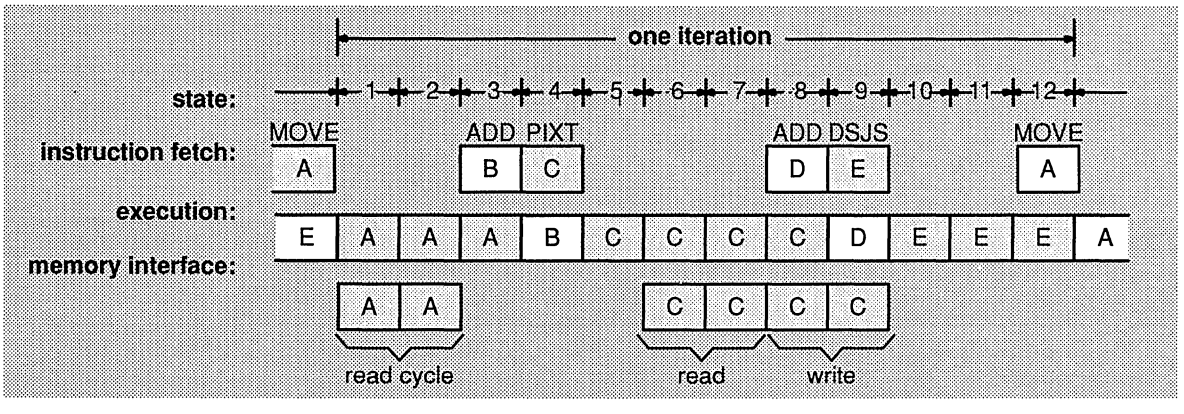


Figure 5–4 *(a)* shows the inner loop of a graphics routine; Figure 5–4 *(b)* represents execution of the code in *(a)*. Figure 5–4 *(b)* shows three activities occurring in parallel:

- Instructions are fetched from cache.
- Instructions are executed through the general-purpose registers and the ALU.
- The local memory interface controller performs memory accesses.

The memory controller accesses pixels while the ALU fetches instructions from cache. The memory controller completes a write cycle while execution begins on the next instruction.

Chapter 6

Interrupts, Traps, and Reset

The TMS34020 supports 10 interrupts, including reset, and up to 65,536 software traps. These interrupts and traps use a set of 32-bit vector addresses that point to appropriate service routines. The TMS34020 also supports bus-fault conditions and single-step execution through these vectors.

You'll find these topics on the following pages:

6.1 Related Signals

Several of the TMS34020's pins request interrupts. Chapter 2 describes the interrupt signals in detail; they are summarized below for your convenience.

Signals	Descriptions	I/O
BUSFLT	is a bus-fault signal that tells the local-memory controller that an error (or *fault*) occurred on the current bus cycle. BUSFLT operates in conjunction with the LRDY signal; if both BUSFLT and LRDY are sampled high during a local-memory cycle, a bus-fault interrupt is generated.	I
$\overline{\text{HINT}}$	is the interrupt signal that allows the TMS34020 to send an interrupt request to a host processor. This interrupt is activated by setting bits in the HSTCTLL register.	O
$\overline{\text{LINT1}}$, $\overline{\text{LINT2}}$	are level-sensitive, active-low inputs. They allow external devices to interrupt the TMS34020.	I
$\overline{\text{RESET}}$	is the system reset signal. During normal operation, $\overline{\text{RESET}}$ is driven low to reset the TMS34020.	I

6.2 Related Registers

Several of the TMS34020's I/O registers provide you with control over interrupts. (Chapter 4 provides detailed descriptions of all the I/O registers.) Some registers contain bits that you must set to enable certain interrupts; others contain bits that the TMS34020 or another device sets to identify an interrupt request.

These registers control interrupt functions:

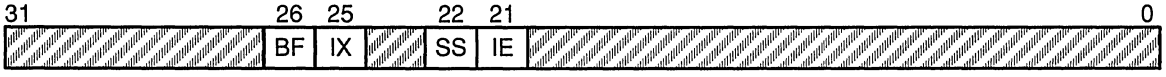
- The **status register** contains a bit that globally controls interrupts; it also reflects the status of certain interrupts.
- The **INTENB** register is the interrupt-enable register.
- The **INTPEND** register is the interrupt-pending register.
- The **HSTCTLL** register is a host-interface register that provides control over general TMS34020-to-host and host-to-TMS34020 interrupts.
- The **HSTCTLH** register is a host-interface register that provides control over the nonmaskable interrupt, halt, and reset.

Note:

You can access I/O registers in the same manner as any other TMS34020 memory locations. You can access the status register with the GETST and PUTST instructions.

The remainder of this section describes these registers and tells you which bits are associated with the interrupts. In the pictures of the registers, shaded areas indicate bits that have no interrupt functions.

Status Register (ST)



Note: For a complete illustration and description of ST, see Section 4.1, page 4-2.

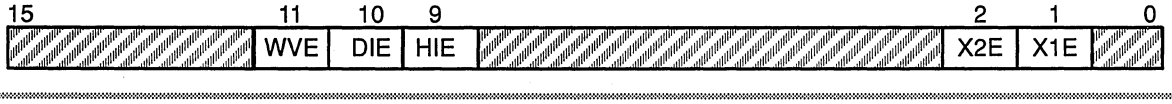
IE bit 21 — Setting IE (global interrupt enable) to 1 allows you to globally enable the interrupts that are controlled by the INTENB register. If IE = 0, then interrupts are globally disabled; in this case, the values in the INTENB register have no effect.

SS bit 22 — Setting SS (single-step enable) to 1 causes a special interrupt to be generated after each instruction is executed. This allows you to single-step through a program.

IX bit 25 — The TMS34020 sets IX (interruptible instruction executing) when it takes an interrupt at an interruptible point in an instruction. The TMS34020 uses IX to ensure that instruction execution resumes correctly after returning from the interrupt.

BF bit 26 — The TMS34020 sets BF (bus fault) when it takes a bus-fault interrupt. The TMS34020 uses this bit to ensure that instruction execution resumes correctly after returning from a bus fault.

INTENB register **address: C000 0110h**



The INTENB register allows you to selectively enable or disable interrupts (when IE=1).

WVE bit 11 — Setting WVE (window-violation interrupt enable) to 1 enables the window-violation interrupt.

DIE bit 10 — Setting DIE (display interrupt enable) to 1 enables the display interrupt.

HIE bit 9 — Setting HIE (host interrupt enable) to 1 enables the host interrupt.

X2E bit 2 — Setting X2E (external interrupt 2 enable) to 1 enables external interrupt 2.

X1E bit 1 — Setting X1E (external interrupt 1 enable) to 1 enables external interrupt 1.

INTPEND register **address: C000 0120h**

The INTPEND register identifies pending interrupts. A *pending interrupt* is an interrupt that was requested but has not yet been serviced.

WVP bit 11 — When the WVP (window-violation interrupt pending) bit equals 1, a window-violation interrupt is pending.

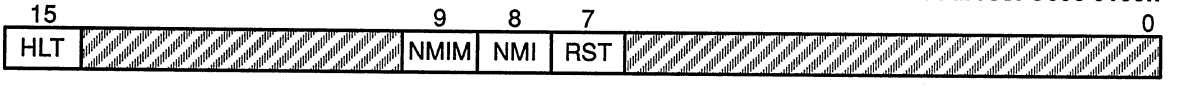
DIP bit 10 — When the DIP (display interrupt pending) bit equals 1, a display interrupt is pending.

HIP bit 9 — When the HIP (host interrupt pending) bit equals 1, a host interrupt is pending.

X2E bit 2 — When the X2P (external interrupt 2 pending) bit equals 1, an external interrupt 2 is pending.

X1E bit 1 — When the X1P (external interrupt 1 pending) bit equals 1, an external interrupt 1 is pending.

HSTCTLH register **address: C000 0100h**



Note: For a complete illustration and description of HSTCTLH, see Chapter 4.

The primary function of these bits is to allow the host processor to interrupt the TMS34020; however, the TMS34020 is also able to write to these bits and may therefore set them itself.

RST bit 7 — Setting the RST (reset) bit causes the TMS34020 to execute a reset.

NMI bit 8 — The NMI (nonmaskable interrupt) bit allows the host processor to interrupt TMS34020 execution.

NMIM bit 9 — The NMIM (NMI mode) bit specifies if the context of an interrupted program is saved when a nonmaskable interrupt occurs.

HLT bit 15 — A host processor can halt the TMS34020's on-chip processor by setting the HLT (halt TMS34020 program execution) bit.

HSTCTLL register **address: C000 00F0h**

Note: For a complete illustration and description of HSTCTLL, see Chapter 4.

MSGIN Bits 0—2 — The MSGIN (message in) field buffers a 3-bit interrupt message from the host processor to the TMS34020.

INTIN Bit 3 — The host processor can set the INTIN (input interrupt) bit to 1 to generate an interrupt request to the TMS34020.

MSGOUT Bits 4—6 — The MSGOUT (message out) field buffers a 3-bit interrupt message from the TMS34020 to the host.

INTOUT Bit 7 — The TMS34020 can set the INTOUT (output interrupt) field to send an interrupt request to the host processor.

HRYI Bit 13 — The TMS34020 sets the HRYI (host retry interrupt) bit if a retry occurs on a host access. If HBREN=1, setting HRYI sends an interrupt request (via $\overline{\text{HINT}}$) to the host processor.

HBFI Bit 14 — The TMS34020 sets the HBFI (host bus-fault interrupt) bit if a bus fault occurs on a host access. If HBREN=1, setting HBFI sends an interrupt request (via $\overline{\text{HINT}}$) to the host processor.

HBREN Bit 15 — When the HBREN (host bus-fault/retry interrupt enable) bit is set, setting HRYI or HBFI causes an interrupt request to be sent to the host processor.

6.3 Enabling and Disabling Interrupts

The TMS34020 supports 10 interrupts; 6 of the interrupts must be enabled before the TMS34020 can recognize them. These interrupts include

- Single-step interrupt
- External interrupt one
- External interrupt two
- Host interrupt
- Display interrupt
- Window-violation interrupt

Note that only these 6 interrupts can be enabled or disabled; reset, bus fault, NMI, and ILLOP cannot be disabled.

Enabling an interrupt

To enable the single-step interrupt, set the SS status bit to 1. The single-step interrupt operates independently of the IE status bit.

To enable any of the other interrupts listed above (excluding the single-step interrupt), follow these steps:

Step 1: Set the IE status bit to 1 (you can do this by executing an EINT instruction).

Step 2: Set the appropriate bit in the INTENB register to 1:

To enable this interrupt:	Set this INTENB bit:
external interrupt 1	X1E (bit 1)
external interrupt 2	X2E (bit 2)
host interrupt	HIE (bit 9)
display interrupt	DIE (bit 10)
window-violation interrupt	WVE (bit 11)

Disabling an interrupt

To disable the single-step interrupt, clear the SS status bit to 0. The IE status bit does not affect this interrupt.

To disable any of these other interrupts (excluding the single-step interrupt), you can do one of two things:

- If the IE bit = 1, clear the appropriate bit in the INTENB register.
- Clear the IE bit to 0 (You can do this by executing a DINT instruction). This disables all five of these interrupts, regardless of the values in the INTENB register.

6.4 Interrupt Priorities and Vector Addresses

Table 6–1 lists the TMS34020 interrupts by priority. Figure 6–1 shows the interrupts' vector addresses.

Table 6–1. Interrupt Priorities

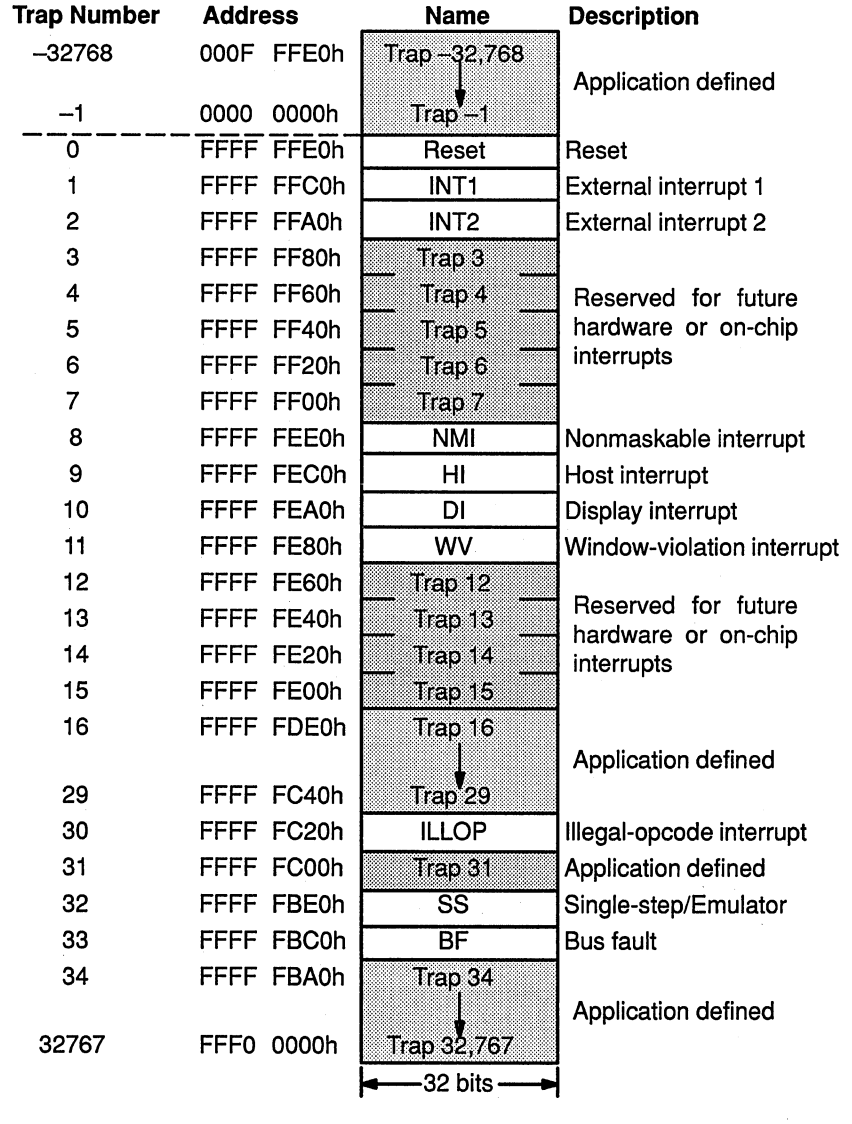
Interrupt	Priority	Source	Description
$\overline{\text{RESET}}$	1	external/ internal	**Device reset.** Taken when the $\overline{\text{RESET}}$ input signal is asserted low or when the RST[HSTCTLH] bit is set.
BF	2	external	**Bus-fault interrupt.** External logic generates a bus-fault interrupt by asserting the BUSFLT signal high; LRDY must also be high.
NMI	3	internal	**Nonmaskable interrupt.** Setting NMI[HSTCTLH]generates a nonmaskable interrupt.
HI	4	internal	**Host interrupt.** The host generates this interrupt by setting INTIN [HSTCTLL].
DI	5	internal	**Display interrupt.** The TMS34020's video timing hardware generates the display interrupt.
WV	6	internal	**Window-violation interrupt.** The TMS34020's CPU generates a window-violation interrupt when a pixel lies either inside a window (windowing mode 1) or outside a window (windowing mode 2).
INT1	7	external	**External interrupt 1.** Asserting $\overline{\text{LINT1}}$ low generates this interrupt.
INT2	8	external	**External interrupt 2.** Asserting $\overline{\text{LINT2}}$ low generates this interrupt.
SS	9	internal	**Single-step interrupt.** When the SS status bit is set, this interrupt is generated after each instruction execution.
ILLOP	10	internal	**Illegal-opcode interrupt.** The TMS34020 generates this interrupt when it encounters an illegal opcode.

Notes: 1) In order for the TMS34020 to recognize interrupts with priorities 4 through 8, you must set the IE status bit.

2) Because only the host can set INTIN[HSTCTLL], HI could be considered an external interrupt. However, HI is not generated directly from an input pin; in keeping with the other interrupts, it is listed as internal.

As Table 6–1 shows, $\overline{\text{RESET}}$ has the highest priority. If 2 interrupts are requested at the same time, the highest priority interrupt is serviced first (assuming it is enabled). The bus-fault condition is considered to be in the interrupt priority chain, although it behaves differently from other interrupts.

Figure 6–1. Vector Address Map



Notes: 1) Traps –1 through –32,768 use the memory at the bottom of the address space for vector addresses. Traps 0 through 32,767 use the memory at the top of the address space.

2) Traps 0 through 31 may be accessed by either a TRAP or TRAPL instruction.

3) Traps –1 through –32,768 and 32 through 32,767 are accessed only by TRAPL.

4) Traps 3 through 7 and 12 through 15 are reserved for future interrupts.

6.5 Interrupt Processing

When an interrupt has been requested but has not yet been processed, it is called a ***pending* interrupt**. If a pending interrupt is enabled (and no interrupt with a higher priority is also pending), the TMS34020 accepts the interrupt at the end of the current instruction cycle (or at the next interruptible point within instruction execution). Figure 6–2 lists the actions that the TMS34020 takes when an interrupt occurs.

Figure 6–2. Actions Performed When the TMS34020 Takes an Interrupt

1) If necessary, the TMS34020 pushes onto the stack any temporary registers that the current instruction is using. This allows the instruction to resume execution correctly upon return from the interrupt.
   a) If the interrupt is taken part-way through an interruptible instruction, the TMS34020 pushes twenty-four 32-bit words. (If the SP is not word aligned when the interrupt is taken, the TMS34020 may push another long word on as padding.)
   b) If the interrupt is caused by a bus fault, the TMS34020 pushes thirty-one 32-bit words.
2) The TMS34020 pushes the PC onto the stack.
3) If necessary, the TMS34020 modifies the ST so that instruction execution can resume correctly after returning from the interrupt.
   a) If the interrupt is taken part-way through an interruptible instruction, the TMS34020 sets the IX bit.
   b) If the interrupt is caused by a bus fault, the TMS34020 sets the BF bit.
4) The TMS34020 pushes the ST onto the stack.
5) The TMS34020 modifies the contents of the ST as follows:

N	C	Z	V		BF	IX		SS	IE		FE1	FS0	FE0	FS1
0	0	0	0	0	0	0	0 0	0	0	0s	0	0 0 0 0 0	0	1 0 0 0 0

6) The TMS34020 fetches the interrupt vector from external memory and places it into the PC.
7) The TMS34020 begins executing the instruction pointed to by the new PC value.

When the first instruction of the service routine begins execution, the new status register contents imply the following conditions:

- All interrupts (except BF, NMI, and reset) are disabled.
- Field 0 is 16 bits long and zero-extended.
- Field 1 is 32 bits long and zero-extended.
- Single-stepping is disabled.

If a graphics instruction is interrupted, the TMS34020 *does not save* the B-file registers used as implied operands (it doesn't push them onto the stack). If your interrupt service routine needs to use these registers, the routine should first push them onto the stack, then pop them from the stack before returning.

You may usually want an interrupt service routine to complete before allowing any more interrupts. However, if you want to be able to interrupt a service routine, the routine should

Step 1: Set the IE status bit to 1.

Step 2: Set the appropriate bits in the INTENB register.

The service routine can also load new field sizes, if required.

If you want to single-step through the interrupt service routine, you can do so by setting the SS status bit.

6.5.1 Returning from an Interrupt Service Routine

Interrupt service routines should not assume anything about the state of the stack except that the stack contains the PC, the ST, and possibly some extra words (as outlined in Figure 6–2, item 1). The interrupt service routine *must* return using a RETI or RETM instruction. Only these instructions pop the PC, ST, and any extra words from the stack to their correct internal locations, thereby enabling instruction execution to proceed from the point at which the interrupt occurred. Note that RETS cannot be used, because it pops only the PC.

Figure 6–3. Actions Performed When the TMS34020 Executes a RETI or RETM Instruction

1) The TMS34020 pops the value of the ST from the stack.
2) The TMS34020 pops the value of the PC from the stack.
3) If necessary (as indicated by the IX and BF bits of the restored ST value), any extra words that were pushed onto the stack are popped from it.
4) If the restored IE bit is 1, the TMS34020 takes one of the following actions, depending on the instruction used:
   - RETI does not alter the restored value of IE. If another interrupt is pending, it is taken as soon as RETI completes, before the TMS34020 can resume execution of the interrupted program.
   - RETM masks IE during the last machine state of the return. This has the effect of not enabling interrupts until one machine state after RETM completes, which means that even if another interrupt is pending, the TMS34020 resumes execution of the interrupted program. The interrupt is then taken at the next interruptible point within the program. RETM is used primarily with the single-step interrupt (see Section 6.13, page 6-28).

The RETI and RETM instructions perform the actions described in Figure 6–3. Provided the interrupt routine has not changed any of the values on the stack, this restores the CPU to its state immediately prior to taking the interrupt. Under no circumstances should you change the value of any of the additional words that may have been pushed onto the stack.

If the cause of an interrupt remains when the TMS34020 completes execution of RETI or RETM, the interrupt is taken again. When necessary, the interrupt service routine should take the appropriate steps to clear the cause of the interrupt. Consideration is given to this in Sections 6.7 to 6.9, which discuss each of the interrupts in detail.

6.5.2 Interrupt Latency

The delay between when an interrupt request is made and when the TMS34020 begins servicing the interrupt depends primarily on what activity the TMS34020 is performing at the time. The delay can be broken down into a number of smaller delays, which fall into four categories. These are listed below.

- **Delay 1: Interrupt request recognition.** This is the period between the time the interrupt is requested and the time the interrupt is recognized. This is one machine state for interrupts generated synchronous to the TMS34020 (such as HI, NMI, and WV), and one to two machine states for interrupts generated asynchronously to the TMS34020 (such as INT1, INT2, and DI).

- **Delay 2: CPU response time.** This is the time required for an instruction that was already executing when the interrupt was recognized either to complete or to reach the next interruptible point. This depends on the instruction. The instruction timings in Chapter 15 provides details of the delay possible for each instruction.

- **Delay 3: Interrupt context switch.** This is the time required to push the PC, ST, and any extra words required onto the stack (as detailed in Figure 6–2), and read the appropriate interrupt vector from memory.

- **Delay 4: Local memory traffic.** Any other memory controller activity that occurs while the CPU is completing the current instruction or performing the context switch has an adverse effect on Delay 3, and affects Delay 2 if the executing instruction (such as a PIXBLT or FILL) performs many local-memory cycles. You should determine what percentage of local-memory bus bandwidth is taken up with screen refresh, DRAM refresh, and host local-memory cycles, then increase the delay produced by Delay 2 + Delay 3 by this amount.

Table 6–2 summarizes these delays and gives some best- and worst-case figures. Because the local-memory interface is typically the limiting factor in the calculations, two worst-case conditions are described; one with 32-bit page-mode memory, and the other with 16-bit non-page-mode memory. These figures are intended only to give you some idea of the delay involved in servicing an interrupt. Obviously, other factors not discussed here could further delay the interrupt: for instance, if another interrupt, which maintained IE=0 throughout its service routine (disabling other interrupts), was being serviced at the time the interrupt being considered here occurred. Inserting wait states into the local-memory cycles also increase the delays. The precise effect of this is difficult to estimate, because not all the delays are determined by local-memory interface performance.

Table 6–2. Sources of Interrupt Delay

	Latency (in states)		
Delay Type	**Minimum**	**Maximum A**	**Maximum B**
Interrupt recognition	1	2	2
CPU response time (note 3)	0	32	80
Interrupt context switch	12		
❑ Interruptible instruction		48	129
❑ Bus-fault interrupt		55	157
Local memory traffic	0		
❑ Per screen refresh		2	2
❑ Per DRAM refresh		3	3
❑ Per host access		2	4

Notes: 1) Maximum A assumes 32-bit wide memory, which supports page mode.

2) Maximum B assumes 16-bit wide memory, which does not support page mode, with no wait states.

3) This is for the worst-case instruction (PIXBLT XY,XY). Other instructions are less. See Chapter 13, *Assembly-Language Instruction Set*, for more details.

4) If both the host and the CPU request accesses as frequently as they can, the memory controller priorities are arranged so that they each receive alternative local-memory cycles.

6.6 Interrupting Instruction Execution

When an instruction is interrupted, instruction execution is suspended until the interrupt routine completes. At this point, the instruction resumes an finishes executing.

While an instruction is executing, the state of the instruction is stored in inaccessible internal registers. When an interrupt occurs, the contents of these registers must be moved to memory before the TMS34020 begins executing the interrupt routine, so that the instruction state can be restored when the interrupt routine completes.

The following events take place when an instruction is interrupted.

Step 1: The IX[[ST]] (interruptible instruction executing) bit is set to 1. This indicates that the interrupt occurred while an instruction was executing.

Step 2: The contents of any internal temporary registers are pushed onto the stack.

Step 3: The PC and ST are also pushed onto the stack.

Step 4: Control branches to your TRAP routine, which should use the MMTM instruction to stack any register values that need to be preserved for later use outside the trap routine.

Step 5: At the end of the TRAP routine, you should use the MMFM instruction to restore the stacked register file values and execute a RETI instruction. (RETI marks the end of the TRAP routine.) Executing RETI returns control to the interrupted program, popping the ST and PC from the stack. When the IX bit is detected, the internal register values are also popped from the stack, and the interrupted instruction resumes execution. RETI clears IX.

Note that the graphics instructions described in Chapter 12 may take several thousand machine cycles to execute, depending on the size of the lines and arrays involved. These instructions check for interrupts at regular intervals, preventing delays to high-priority interrupts from becoming prohibitively long.

Note:

1) IX is not set to 1 when a PIXBLT or FILL instruction is *aborted* as a result of a window violation. In this case, returning from an interrupt routine causes the TMS34020 to execute the instruction that *follows* the interrupted instruction.

2) If a graphics instruction is executing when a *bus fault* occurs, the steps described on page 6-13 take place. In addition, BF[[ST]] is set to 1. In this case, executing RETI at the end of the bus-fault TRAP routine also clears BF.

3) If the SP is not long-word aligned when an instruction is interrupted, then the TMS34020 aligns the stack to expedite the interrupt sequence. RETI always restores the SP to its original alignment.

4) The FPIXEQ and FPIXNE instructions are exceptions. These two instructions follow the same basic operation when interrupted, but they do not preserve the contents of temporary registers (skipping step 2, listed on page 6-13). Instead, these instructions reset their operands so that they can resume execution when control returns from the interrupt.

6.7 External Interrupts 1 and 2

The TMS34020 supports two general-purpose interrupts that allow external devices to interrupt the TMS34020. This is achieved by driving $\overline{\text{LINT1}}$ or $\overline{\text{LINT2}}$ low. The interrupts generated by requests to $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ are called INT1 and INT2. Table 6–3 lists these interrupts, the signals that generate them, and the interrupt trap vectors.

Table 6–3. External Interrupt Vectors

Name	Input Pin	Vector Address
INT1	$\overline{\text{LINT1}}$	FFFF FFC0h
INT2	$\overline{\text{LINT2}}$	FFFF FFA0h

Each signal is dedicated to an individual interrupt, allowing 2 separate and distinct external-interrupt requests. INT1 has a higher priority than INT2; if $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ become active at the same time and both external interrupts are enabled, INT1 is serviced first.

X1P[[INTPEND]] and X2P[[INTPEND]] reflect the current state of the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ inputs. A bit equals 1 if the corresponding request is active, 0 if it is not. You can poll these bits to detect transitions at the interrupt inputs.

Once an external devices request an interrupt, the device should continue to drive the interrupt signal low until the TMS34020 has started to execute the interrupt service routine.

- If the device permits the interrupt pin to go inactive high before the routine recognizes the interrupt, the request may be missed.
- If the active level is maintained after the interrupt service routine completes, the interrupt is taken again.

How you ensure that the interrupt pin is held active until after the beginning of the service routine depends on the application. However, two possibilities are

- The interrupt service routine writes to an external location to cause the appropriate interrupt pin to be deactivated.
- External hardware decodes the vector fetch for the interrupt from the status code and vector address output on the LAD bus.

The TMS34020 assumes that signals input to $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ are asynchronous to the TMS34020 local clocks; the TMS34020 synchronizes the signals before it processes them. The TMS34020 samples the state of the $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ inputs at each high-to-low transition of LCLK1 and updates the X1P and X2P bits accordingly. The delay from the transition at the input to the corresponding change in the X1P or X2P bit is from 1 to 2 states, depending on the transition's phase relationship to LCLK1.

6.8 Internal Interrupts

Table 6–4 summarizes the internal interrupts.

Table 6–4. Interrupts That are Associated with Internal Events

Name	Function	Trap #	Vector Location
NMI	Nonmaskable interrupt	8	FFFF FEE0h
HI	Host interrupt	9	FFFF FEC0h
DI	Display interrupt	10	FFFF FEA0h
WV	Window-violation interrupt	11	FFFF FE80h
SS	Single-step interrupt	32	FFFF FBE0h
ILLOP	Illegal-opcode interrupt	30	FFFF FC20h

If more than one interrupt is pending, the interrupts are serviced according to the priorities listed in Table 6–1 (page 6-7).

6.8.1 The Nonmaskable Interrupt (NMI)

The nonmaskable interrupt occurs when a 1 is written to NMI[[HSTCTLH]] (this is normally done by a host processor). This interrupt cannot be disabled and always occurs as soon as possible following the request. The NMI is delayed only for completion of an instruction already in progress, or until the next interruptible point of an interruptible instruction (such as a PIXBLT) is reached.

NMIM[[HSTCTLH]] (NMI mode bit) determines whether context information is saved on the stack when a nonmaskable interrupt occurs:

- If NMIM = 0, the PC and ST are pushed on the stack before servicing the interrupt.
- If NMIM = 1, nothing is saved on the stack before servicing the interrupt.

The TMS34020 automatically clears the NMI bit when it takes the interrupt, so there is no need for the interrupt service routine to do this.

6.8.2 The Host Interrupt (HI)

The host interrupt occurs when a 1 is written to INTIN[[HSTCTLL]], providing that HI is enabled (HIE=1). Only the host processor can do this; the CPU cannot write a 1 to INTIN. The host interrupt is serviced as soon as possible following the request.

The MSGIN[[HSTCTLL]] bits provide a mechanism for specifying the action taken by the host interrupt; by checking the value of MSGIN at the beginning of the interrupt routine, you can branch into one of up to eight different procedures. The host can write to MSGIN when it sets INTIN. Only the host can write to these bits; a write by the CPU has no effect.

Before returning from the interrupt, the service routine must clear INTIN. This is the only way INTIN can be cleared, because the host cannot write a 0 to INTIN. Clearing HIP[[INTPEND]] *does not clear* the interrupt.

6.8.3 The Display Interrupt (DI)

The display interrupt coordinates processing activity with display refreshes. The display-interrupt request becomes active when a particular display line, specified in the DPYINT register, has been output to the monitor screen. At the start of each horizontal blanking period, the VCOUNT register is compared to the DPYINT register. When the vertical-count value in VCOUNT has reached the value in DPYINT, the TMS34020 generates a display interrupt request. If enabled, the interrupt is taken.

Before returning, the display interrupt service routine should clear DIP[[INTPEND]] so that the interrupt isn't taken again.

6.8.4 Window-Violation Interrupt (WV)

The window-violation interrupt may occur when the TMS34020 is executing a graphics operation and windowing option 1 or 2 is selected. W[[CONTROL]] defines the window-checking option. WVP[[INTPEND]] is set if

W=1 and an attempt is made to write a pixel *inside* the specified window, or

W=2 and an attempt is made to write a pixel *outside* the specified window.

Before returning, the window-violation interrupt service routine should clear WVP[[INTPEND]] so that the interrupt is not taken again.

6.8.5 The Single-Step Interrupt

The single-step interrupt provides a mechanism for executing instructions one at a time. This is useful when developing and debugging new programs. While SS=1, an interrupt is generated after each instruction is executed. Unlike other interrupts, the single-step interrupt is not taken at interruptible points within an instruction, only on instruction boundaries.

The single-step interrupt service routine should use the RETM instruction to return. This allows the next instruction to be executed before the interrupt is taken again. If RETI is used, the single-step interrupt is taken again as soon as the RETI completes, and the program is not executed at all; only the interrupt service routine is executed.

Because SS is contained in ST, and ST is saved on the stack before servicing the interrupt routine, and then restored afterwards, the only methods for clearing the SS bit are

- To modify the value of the ST on the stack during the single-step service routine, so that when the ST is restored by RETM, SS is cleared.
- To single-step a PUTST instruction within the program which clears SS.

Note:

All interrupts clear the SS bit so that interrupt service routines execute normally (the instructions in the routine aren't single-stepped). If you want to single-step through an interrupt service routine, one of the routine's first instructions should set the SS bit.

6.8.6 Illegal-Opcode Interrupts

The TMS34020 recognizes several reserved opcodes as illegal. If the TMS34020 encounters one of these opcodes, it traps to vector number 30 (located at memory address FFFF FC20h). An illegal opcode is similar in effect to a TRAP 30 instruction. The illegal-opcode interrupt cannot be disabled.

For testing purposes, opcodes 0000h and FFFFh are reserved as illegal opcodes on all TMS340 family devices. Other currently illegal opcodes may be used for special functions on future TMS340 devices.

A typical cause of an illegal-opcode interrupt is that the program being executed is corrupted (perhaps because insufficient stack space was allocated). If you wish to resume execution of the program, the interrupt service routine should take whatever steps are necessary to remove the illegal opcode from the program, and then flush the cache before resuming. If the cache is not flushed, the illegal opcode is executed again, because it is still present in the cache.

6.9 The Bus-Fault Interrupt

The bus-fault interrupt provides a mechanism by which the TMS34020 can be interrupted by its local memory system. This allows correction of an error that occurred during an access to a particular location. The precise locations (or groups of locations) within the local-memory cycle that can generate a bus fault depend entirely on the application. Here are some examples of the use of the bus-fault mechanism:

- It can indicate that the TMS34020 is attempting to access invalid areas of memory.
- It can indicate that the TMS34020 is attempting to access protected devices.
- It can indicate that the TMS34020 is attempting to access an area of memory implemented as virtual memory space that is not currently mapped into the physical memory.

A bus fault may be generated as one of the options for ending a local-memory cycle. Asserting the BUSFLT and LRDY pins high at the rising edge of LCLK2 during the data subcycle causes a bus fault to occur. This is discussed in detail in Section 8.6, Ending a Local-Memory Cycle (page 8-12). Bus faults can be generated on virtually all types of local-memory cycle. However, a bus-fault interrupt is generated only if the local-memory cycle was initiated by the CPU.

6.9.1 Activity During a Bus-Fault Interrupt

Unlike any other interrupt, a bus fault does not occur at an instruction boundary or at an interruptible point within an instruction. By definition, it occurs *during* an instruction and must be serviced before program execution can continue. However, the pipelining of data between the CPU and the memory controller means that when a bus fault occurs on a memory cycle, the CPU may already be requesting the *next* memory operation. If this is the case, the CPU will have discarded the information relevant to the bus-faulted memory cycle. Because of this, the CPU cannot stack information relating to the bus-faulted access, and so the memory controller must save its own state at the time of the bus-faulted access. This means that the sequence of events when a bus fault occurs is slightly different from other interrupts:

1) The memory controller saves the value of the LAD bus from the bus-faulted cycle in the 32-bit BSFLTD register.
2) The memory controller saves its own state in BSFLTST.
3) The memory controller generates a bus-fault interrupt to the CPU.
4) The CPU responds to this interrupt *immediately* (within one machine state) and proceeds to push its state onto the stack as outlined in Figure 6–2.

The bus-fault service routine should start from the address stored at the bus-fault vector (address FFFF FBC0h). The interrupt service routine should take the action necessary to clear the cause of the bus fault. Upon returning from the bus-fault routine, the following actions occur:

1) The CPU restores its state from the stack as outlined in Figure 6–3.

2) The CPU signals the memory controller to resume normal execution.

3) The memory controller restores its state to that of the bus-faulted access from the BSFLTST register.

4) The memory controller restarts the faulted access. If the access was a write, the data stored in the BSFLTD register is driven out on the LAD bus during the cycle.

If you do not want to restart the memory cycle that was originally bus-faulted, your interrupt service routine should write FFFFh to the BSFLTST register. This causes the memory controller to restore its inert state (no local-memory accesses pending). Do not, under any circumstances, modify the BSFLTST register to any other value; doing so causes unreliable operation.

6.9.2 Bus Fault System Considerations

Because the memory controller saves the state of the bus-faulted memory access in the BSFLTD and BSFLTST I/O registers, and not on a stack, you should ensure that bus faults cannot be generated when accessing the system stack. If a bus fault occurs while the CPU is pushing its state onto the stack before servicing the bus-fault interrupt (or popping its state off the stack after returning from the bus-fault interrupt), the state of the original bus-faulted memory cycle is lost.

If it is possible for a bus fault to occur while you are executing the bus-fault service routine, you should ensure that one of the first operations performed by the service routine is to push the BSFLTD and BSFLTST registers onto the stack. These should be restored before returning from the bus fault. Care should be taken to ensure that any other interrupts which could occur before the service routine has stacked these registers (such as an NMI) cannot access locations that could generate a bus fault.

6.10 Interrupting a Host Processor

The TMS34020 has an output pin dedicated to interrupting a host processor. The TMS34020's CPU can interrupt the host by writing a 1 to INTOUT [HSTCTLL]. When this occurs, the $\overline{\text{HINT}}$ pin is asserted active low. It remains at this level until INTOUT is cleared. Only the host can clear INTOUT; only the TMS34020 can set it.

The MSGOUT[HSTCTLL] bits provide a mechanism for specifying the action taken by the host when it takes the interrupt; by reading the value of MSGOUT at the beginning of the interrupt routine, the host can branch into one of up to eight different procedures. The CPU can write to MSGOUT when it sets INTOUT. Only the CPU can write to these bits; a write by the host has no effect.

In addition, provided that HBREN[HSTCTLL] (host bus-fault/retry interrupt enable) is set, $\overline{\text{HINT}}$ is driven low whenever a retry or bus fault occurs during a host-initiated local-memory cycle. When one of these situations arises, the TMS34020's memory controller sets either HRYI[HSTCTLL] (host retry interrupt) or HBFI[HSTCTLL] (host bus-fault interrupt). $\overline{\text{HINT}}$ remains active until the host clears the appropriate bit (HRYI or HBFI). Clearing HBREN also causes $\overline{\text{HINT}}$ to be driven inactive.

6.11 Traps

The TMS34020 supports 65,536 software traps, numbered –32,768 through 32,767. Software traps behave similarly to interrupts, except that they are initiated when the TMS34020 executes a TRAP or TRAPL instruction. The TRAP instruction provides access to traps 0—31 by using a single 16-bit instruction word. The TRAPL instruction allows access to all 65,536 traps. Unlike an interrupt, a software trap cannot be disabled.

When the TMS34020 executes a TRAP or TRAPL instruction, it performs the same sequence of actions that it performs for interrupts. All traps except trap 0 push the status register and the PC onto the stack. Trap 0 is similar to a hardware reset because it does not push the status register or PC onto the stack; it differs from a hardware reset because it does not set the TMS34020's internal registers to a known initial state. Trap 8 is similar to an NMI interrupt, except that NMIM[HSTCTLH] does not affect instruction execution. The status register and PC are stacked unconditionally when trap 8 is executed.

A 32-bit vector address is associated with each software trap. Here's how you determine the vector address for a trap number n:

- If n = 0 through 32,767, subtract 32n from FFFF FFE0h.
- If n = –1 through –32,767, subtract 32 from –32n.

Figure 6–1 on page 6-8 shows the vector addresses for the software traps.

6.12 Reset

Reset is the highest priority interrupt; it puts the TMS34020 into a known initial state. There are two ways to invoke reset.

- **Assert an active-low level on the $\overline{\text{RESET}}$ pin.** At power-up, $\overline{\text{RESET}}$ should always be asserted for a minimum of forty local clock periods after power levels have become stable. At other times, you may reset the TMS34020 by asserting $\overline{\text{RESET}}$ for a minimum of four local clock periods.
- **Write a 1 to RST[HSTCTLH].** This achieves the same result as asserting the $\overline{\text{RESET}}$ pin, but without affecting any other devices in the system to which the $\overline{\text{RESET}}$ pin may be wired. Reset should not be invoked in this way at power-up.

Unlike other interrupts and software traps, reset does not save the previous ST and PC values on the stack. The value of the stack pointer just before a reset may be invalid. Saving these values on the stack could corrupt valid memory locations.

6.12.1 Activity During Reset

When reset is initiated, the TMS34020 takes 34 local clock periods to completely initialize itself (40 cycles must be allowed at power-up because the TMS34020 may be in an illegal state not achievable during normal operation). Most of this time is spent clearing the I/O registers, which are cleared at the rate of two 16-bit registers per machine state. While the $\overline{\text{RESET}}$ pin is asserted, the local-memory control signals are in the states shown in Table 6–5.

Table 6–5. Initial State of Output Pins while $\overline{\text{RESET}}$ and $\overline{\text{GI}}$ are Low

Outputs Driven to High Level		Outputs Driven to Low Level	Bidirectional Pins Driven to High-Impedance
$\overline{\text{RAS}}$	$\overline{\text{HINT}}$	HRDY	$\overline{\text{VSYNC}}$
$\overline{\text{CAS3}}$—$\overline{\text{CAS0}}$	$\overline{\text{R1}}$	$\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$	$\overline{\text{HSYNC}}$
$\overline{\text{WE}}$	$\overline{\text{HOE}}$	DDIN	$\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$
$\overline{\text{TR}}$/$\overline{\text{QE}}$	HDST	$\overline{\text{R0}}$	LAD31—LAD0
$\overline{\text{DDOUT}}$	RCA12—RCA0		
$\overline{\text{ALTCH}}$	SF		

Note: When $\overline{\text{GI}}$ is high, all $\overline{\text{GI}}$-controlled pins are driven to high impedance. $\overline{\text{GI}}$-controlled pins include $\overline{\text{RAS}}$, $\overline{\text{CAS0}}$—$\overline{\text{CAS3}}$, $\overline{\text{WE}}$, $\overline{\text{TR}}$/$\overline{\text{QE}}$, $\overline{\text{DDOUT}}$, DDIN, $\overline{\text{ALTCH}}$, $\overline{\text{HOE}}$, HDST, RCA0—RCA12, LAD0—LAD31, and SF.

The specifications for certain DRAM and VRAM devices require that at power-up the $\overline{\text{RAS}}$ signal be driven inactive-high for 1 millisecond after power becomes stable. As long as $\overline{\text{RESET}}$ is maintained active, the TMS34020 drives its $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals inactive-high. In general, holding $\overline{\text{RESET}}$ low for t microseconds ensures that $\overline{\text{RAS}}$ remains high initially for t–(10–t_q) microseconds, where t_q is one quarter of the local clock period.

At times other than power-up, the TMS34020 may be in the process of resetting itself while $\overline{\text{RESET}}$ is high. This is the case if reset was initiated by setting the RST bit or by asserting $\overline{\text{RESET}}$ for less than 34 LCLK periods. If $\overline{\text{RESET}}$ is high and the TMS34020 is internally resetting itself, the memory controller performs consecutive DRAM-refresh cycles. This ensures that the DRAM contents are maintained while the TMS34020 is reset.

The value of the REFADR register is output as a pseudo-address during each DRAM-refresh cycle. REFADR is incremented after each DRAM refresh cycle. However, if the DRAM refreshes start before REFADR is cleared, the address output reverts to zero when this occurs, and then starts incrementing again.

6.12.2 Initial State Following Reset

The TMS34020 completes its reset procedure when $\overline{\text{RESET}}$ is deactivated, or thirty-four local clock periods after the high-to-low transition of the $\overline{\text{RESET}}$ pin, which ever occurs last. Immediately following reset, the TMS34020 is in the following state:

- **Registers**
  - All I/O registers are cleared to 0000h. The only possible exceptions to this are HLT[[HSTCTLH]] (see Section 6.12.4), REFADR (which will have incremented if DRAM refreshes were performed during reset), and SCOUNT (if SCLK is oscillating during reset).
  - The general-purpose register files A and B are uninitialized.
  - The ST is set to 0000 0010h.
  - The PC is uninitialized.
- **Cache**
  - The cache SSA (segment start address) registers are uninitialized.
  - The cache LRU (least recently used) stack is set to the sequence 0, 1, 2, 3. This indicates that segment 0 is the most recently used, and segment 3 is the least recently used.
  - All cache P (present) flags are cleared. This indicates that the cache is empty.
- The DRAM refresh-pending counter is set to 9.

6.12.3 Activity Following Reset

Immediately following reset, the memory controller begins normal operation. At this time, the refresh-pending counter is set to 9. Four or more DRAM refreshes take priority over any CPU memory request, so the memory controller performs DRAM refreshes continuously until the pending counter counts down to 3. At this point, any pending CPU memory request (in this case the reset-vector fetch) is performed. Because reset sets the TMS34020 to increment the refresh-pending counter every 8 machine states, additional DRAM refreshes are requested before the 9 counts down to 3. This results in a total of 9 consecutive DRAM refreshes, which occur before any CPU-initiated memory requests are performed. The remaining refreshes are performed using the normal memory controller priority scheme.

This fulfills most DRAM/VRAM requirements that the DRAM/VRAM's $\overline{\text{RAS}}$ pins (after being held inactive for 1 ms) are cycled a minimum of 8 times after power-up before any memory accesses are made. This ensures that the DRAMs/VRAMs are initialized for correct operation.

Note, however, that if a host requests access before the 8 DRAM refreshes complete, the host request are performed. Thus, at power-up, the host should not make any requests to DRAM memory until the 8 initialization cycles have had time to complete.

If at other times, reset is initiated by setting RST[HSTCTLH] or by asserting $\overline{\text{RESET}}$ in a manner that maintains the data in the DRAMs. There is no need for the host to delay making a memory request, because the DRAMs will already be initialized.

If you initiate reset by asserting $\overline{\text{RESET}}$, the memory will be maintained if $\overline{\text{RESET}}$ is not held low for longer than the maximum refresh interval less the time taken for the TMS34020 to refresh the memories.

After reset completes and the 8 DRAM-refresh cycles are performed, the TMS34020 either

- begins executing instructions (self-bootstrap mode), or
- halts until the host clears HLT[HSTCTLH].

The level on the $\overline{\text{HCS}}$ pin just before $\overline{\text{RESET}}$'s low-to-high transition selects between these two modes. The TMS34020 remembers this information, so that if reset is initiated via RST[HSTCTLH], the CPU is configured in the mode indicated at the most recent rising edge of $\overline{\text{RESET}}$.

6.12.3.1 Self-Bootstrap Mode

In self-bootstrap mode, the TMS34020 begins executing instructions immediately following reset. This mode is typically used in a system in which the reset vector and reset service routine are contained in nonvolatile memory (such as a bootstrap ROM). This type of system does not necessarily require a host processor, and the TMS34020 may be responsible for performing host-processor functions for the system.

The TMS34020 is configured in self-bootstrap mode when the $\overline{\text{HCS}}$ pin is low just before $\overline{\text{RESET}}$'s low-to-high transition. The low level on $\overline{\text{HCS}}$ does not alter the HLT bit, which was cleared to 0 during reset. Immediately following the end of reset and the 9 DRAM-refresh cycles, the TMS34020 fetches the level-0 vector address (from address FFFF FFE0h) and begins executing the reset interrupt routine.

Transitions of the $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ signals are assumed to be asynchronous with respect to the TMS34020 local clock. $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ are internally synchronized to the local clock by being held in latches for at least 1 clock period before the TMS34020 uses them. The delay through the synchronizer latch is from 1 to 2 local clock periods, depending on the phase of the signal transitions relative to the clock. TMS34020 on-chip logic delays the $\overline{\text{HCS}}$ low-to-high transition to ensure that it is detected **after** $\overline{\text{RESET}}$'s low-to-high transition. The level of the delayed $\overline{\text{HCS}}$ signal at this time determines the value of the HLT bit. In systems without a host processor, this allows $\overline{\text{HCS}}$ and $\overline{\text{RESET}}$ to be wired together without the need for any external logic to delay the transition on the $\overline{\text{HCS}}$ pin.

6.12.3.2 Host-Present Mode

Host-present mode assumes that a host processor is connected to the TMS34020's host-interface pins. In this mode, the TMS34020 local memory can be composed entirely of RAM. Following reset, the host processor may download the initial program code, interrupt vectors, etc., before allowing the TMS34020 to begin executing instructions.

Here's how the TMS34020 is configured in host-present mode. The $\overline{\text{HCS}}$ input is sampled on the trailing edge of $\overline{\text{RESET}}$. If $\overline{\text{HCS}}$ is inactive high, internal logic forces the HLT bit to a 1. In this fashion, the TMS34020 is automatically halted following reset, and does not begin executing its reset service routine until the host processor clears HLT to 0. In the meantime, the host processor can load the memory and I/O registers with the appropriate initial values before the TMS34020 begins executing instructions. This may, for example, include writing the reset vector and reset service routine into the TMS34020's memory.

No additional external logic is required to force $\overline{\text{HCS}}$ high before $\overline{\text{RESET}}$'s low-to-high transition. External decode logic is typically used to drive the $\overline{\text{HCS}}$ input active low only when the TMS34020 is addressed by the host processor. Assuming that the host processor is not actively chip-selecting the TMS34020 at the end of reset, $\overline{\text{HCS}}$ is high.

6.12.4 System Configuration Following Reset

Before any memory locations can be accessed, or instructions executed, the TMS34020 must be configured in the correct addressing mode. Two different aspects must be considered:

- Little-endian or big-endian addressing. This determines which bit within a word is addressed as the least significant (see Chapter 3 more details).
- The base array size of the DRAMs and VRAMs in the system. This determines how logical address bits are mapped to the RCA bus to form the row and column addresses used by DRAMs and VRAMs in the system (see Section 8.16.2, page 8-51, for details).

The mode bits that determine these configurations are BEN[[CONFIG]] (big-endian enable) and RCM0—RCM1[[CONFIG]] (RCA mode).

The TMS34020 provides a mechanism for ensuring that these bits are set correctly following reset.

Reset normally clears BEN and RCM0—RCM1, configuring the TMS34020 to little-endian operation with a DRAM base array size of 64K×*n*. Before these bits are set correctly, the only memory locations that can be reliably accessed are those with a row address of all 1s or all 0s and are long-word-aligned, 32-bit words (locations with a bit address of zero, and a field size of 32). The reset vector fulfills these characteristics. This is the first location accessed after reset completes (assuming that the TMS34020 is configured in self-bootstrap mode).

The reset vector contains the address of the first instruction to be executed. The reset vector's 4 LSBs are not required to specify this address because instructions must be aligned to 16-bit word boundaries in memory. The TMS34020 takes advantage of this fact. You should write the values you choose for BEN and RCM0—RCM1 in bits 0—3 of the reset vector. When the reset vector is fetched, the TMS34020 automatically copies these bits into CONFIG and then sets the 4 LSBs of the PC to zero.

Bit 3 of CONFIG is the CBP (configuration byte protect) bit. When a 1 is written to this bit, CONFIG's LSbyte is write-protected until the next time the TMS34020 is reset. You can write the value of this bit to bit 3 of the reset vector; it is also copied into CONFIG with the BEN and RCM0—RCM1 bits. By setting bit 3 of the reset vector to 1, you can ensure that this is the only time BEN and RCM0—RCM1 are modified.

If the TMS34020 completes reset in host-present mode, the host *must* write to BEN and RCM0—RCM1 *directly* before accessing the TMS34020's local memory (unless the system configuration requires these bits to remain 0). After the host writes to these bits, it must not access the local memory during the machine state immediately following the write, because of the latency required

to set the bits. There must be a single machine state between the write to CONFIG and an access to local memory. If the host is capable of requesting an access *immediately* following the write to CONFIG, you should ensure that the host makes a dummy request (such as reading CONFIG) before attempting to write to memory.

If the TMS34020's I/O registers are shadowed in external memory, the initial write to CONFIG to set BEN, RCM0—RCM1, and CBP (either by the TMS34020 or the host) may not be duplicated in external memory because the logical address bitmapping may be incorrect.

6.12.5 $\overline{\text{RESET}}$ and Multiprocessor Synchronization

You can use $\overline{\text{RESET}}$ to synchronize multiple TMS34020s that share a local memory. (Systems that use the TMS34020's multiprocessor interface to control memory access must synchronize the processors.) Synchronization is achieved by taking $\overline{\text{RESET}}$ high within a specific interval relative to CLKIN. TMS34020s to be synchronized should use the same CLKIN and $\overline{\text{RESET}}$ inputs; use CLKIN to clock the $\overline{\text{RESET}}$. All of the local-memory and bus-control signals should be connected in parallel (without buffers) between the processors. After power-up, the processors may not all be executing the same machine state quarter cycle at the same time, because each machine state consists of 4 CLKIN cycles. When the low-to-high transition of $\overline{\text{RESET}}$ occurs, the TMS34020 could be in any one of the 4 quarter cycles. The TMS34020 stretches the first quarter phase (signified by LCLK1 high and LCLK2 low), until the eleventh CLKIN cycle after the rising edge of $\overline{\text{RESET}}$. This ensures that all TMS34020s in the system are exactly synchronized. Section 11.3.2, page 11-3, discusses this in more detail.

The setup and hold times of $\overline{\text{RESET}}$ relative to CLKIN's low-to-high transition (specified in the *TMS34020 Data Sheet)* must be met only to guarantee that a $\overline{\text{RESET}}$ transition is detected at a particular clock edge. In a system with a single TMS34020, the $\overline{\text{RESET}}$ input signal can be asynchronous.

6.12.6 State of VCLK During Reset

In many systems, the VCLK pin continues to be clocked during reset. However, a system in which VCLK is not clocked during reset should maintain VCLK at the logic-high level. This is necessary to ensure that the video counters are reset properly. (In fact, VCLK should be held at the logic-high level when it is not being clocked, regardless of whether the device is being reset.) While VCLK is low, storage nodes in the VCOUNT and HCOUNT registers rely on their internal capacitance to maintain their state. If VCLK remains low for a sufficiently long period, charge leakage may cause bit errors in these registers.

6.13 An Application for Interrupts: Debugging Code

The single-step interrupt causes the TMS34020 to interrupt execution after each instruction; this can be especially useful if you're developing and debugging new programs. This section describes this application.

6.13.1 How a Debugger Works

A debugger typically runs as code on the TMS34020 and code on a host system. If you decide to inspect a TMS34020 register or memory location, one of two situations may arise:

- If you're inspecting an I/O register or a local-memory location, the host-system portion of the debugger can perform a host read to directly access the information.
- If you're inspecting an internal register (general-purpose, ST, or PC), the host-system portion of the debugger must send a request to the TMS34020 portion of the debugger. The TMS34020 can put the required information into an agreed place in memory where the host software can access it through the host interface.

6.13.2 Using a Debugger

A debugger is usually used for loading and running development software under controlled conditions. Such conditions allow you to stop the software at any point and examine or change the state of the TMS34020 before continuing program execution.

If you want to examine the state of the TMS34020 after each instruction, you can do one of two things:

- Insert breakpoints into your code; that is, insert instructions that force the TMS34020 to take a software interrupt. Any of these instructions can act as a breakpoint:

TRAP	TRAPL
EMU	any illegal opcode

  Inserting breakpoints requires the debugger to replace opcodes in the development software. This can be difficult because the debugger must be able to spot potential branches within the code and place breakpoints at all the addresses that could be branched to.
- Use single-step mode. Single-step mode requires no changes to the development software and is much simpler to manage.

6.13.3 Entering Single-Step Mode

In single-step mode, the TMS34020 executes one instruction at a time. This happens regardless of how many machine cycles the instruction consumes and regardless of the amount of immediate data required. After executing an instruction, the TMS34020 saves the PC and ST values on the stack and

passes program control to the single-step routine (the routine's address is at vector address FFFF FBE0h). The single-step routine usually clears the single-step bit is cleared so the routine itself is not single-stepped. The routine then returns control to the debugger so that you to examine the state of the TMS34020.

When you are ready to continue program execution, the TMS34020 portion of the debugger should execute a RETM instruction, marking the end of the single-step routine in much the same way that RETI marks the end of other interrupt code.

RETM tells the TMS34020 to pop the PC and ST values and resume execution of the development code. By popping the ST, the single-step bit is restored to 1; development code resumes execution in single-step mode. After executing the next instruction, the TMS34020 again stacks the PC and ST and passes control to the single-step routine.

6.13.4 Clearing the Single-Step Bit

Note that a single-step routine may alter the ST value on the stack. Clearing the single-step bit in the stacked ST value switches off single-step mode. The RETM that terminates the routine pops the new ST value and returns to the interrupted code. The code now executes normally because the SS bit is cleared.

6.13.5 A Few Things to Keep in Mind

Instructions that change the ST

Instructions that directly change the contents of the status register could cause problems if the development code does not deal correctly with the changes to the ST. For example, assume the following code is being single-stepped:

```
clr      a0
putst a0
movi  012345678h,a9
```

After executing CLR A0, the TMS34020 takes the single-step trap. When the next RETM executes, control passes back to the development code. When the TMS34020 executes PUTST A0, it puts the value 0h (from A0) into the ST, thus clearing the single-step bit. From this point, single-step mode is no longer enabled; the debugger loses control of the development code.

To avoid these problems, the debugger should check the next instruction that the stacked PC points to, before executing it, to see if executing the instruction would affect the SS bit.

Only the PUTST and POPST instructions can directly alter the SS bit. If the debugger finds that the next instruction is PUTST, the debugger can alter the

register value that will be put into the ST so that SS is set to 1. Similarly, if the next instruction is a POPST, the debugger can alter the stack value that will be put into the ST so that SS is set to 1.

Interrupts during execution of development code

When the TMS34020 takes an interrupt, it stacks the PC and ST[1], then clears the ST to 00000010h. The TMS34020 then reads the appropriate trap vector and branches to an interrupt routine.

If single-step mode is enabled when an interrupt occurs, the SS bit is cleared (because the interrupt sets the ST to 00000010h); thus, the interrupt code will not execute in single-step mode. Interrupts that may cause this include

reset	nonmaskable interrupt
host interrupt	display interrupt
window-violation interrupt	external interrupts 1 & 2

Executing one of the following instructions may also cause this:

TRAP	TRAPL
TRAP 0	TRAPL 0
EMU (with a single-step code)	any illegal opcode

As an example, assume that the TMS34020 is single-stepping through this development code:

```
clr     a0
xor     a7
movi  012345678h,a9
```

Assume that an interrupt occurs while the TMS34020 is executing the XOR instruction. The following events occur while the TMS34020 is executing this development code:

1) The CLR instruction executes.
2) A single-step trap returns control to the debugger (single-step routine).
3) The RETM instruction at the end of the single-step routine returns control to the development software.
4) The XOR executes.
5) The interrupt occurs. The TMS34020 stacks the current PC and ST values, sets the ST to 00000010h, and passes control to the interrupt routine. The interrupt routine executes normally (it does not execute in single-step mode).

[1] The following interrupts do not stack the PC or ST: reset, NMI (when NMIM = 1), TRAP 0, and TRAPL 0.

6) The RETI that terminates the interrupt routine restores the old PC and ST values (with SS set).

7) Control immediately passes to the single-step routine.

8) The RETM instruction that terminates the single-step routine returns control to the development code.

9) The MOVI executes.

10) The development code continues execution, taking the single-step routine after each instruction.

11) An RETM instruction terminates the single-step routine.

Here's another look at this sequence:

Code that's single-stepped	Code that's not single-stepped
1) `clr a0`	
	2) *execute single-step routine*
	3) `RETM`
4) `xor a7`	
	5) *execute interrupt routine*
	6) `RETI`
	7) *execute single-step routine)*
	8) `RETM`
9) `movi 012345678h,a9`	
	10) *execute single-step routine*
	11) `RETM`

Notice that the flow of development code and single-step routines is unbroken by the interrupt. The non-single-step interrupt is essentially transparent to the development code.

Single-stepping interrupts

If you want to single-step through an interrupt, then you should place a breakpoint at the start of all relevant interrupt routines. This breakpoint should be

- an EMU instruction (with the associated single-step code),
- a trap instruction, or
- any illegal opcode.

When the development code single-steps to the interrupt point, it takes the interrupt. The first instruction in the interrupt routine causes yet another interrupt, this time set up by the debugger to take the actions necessary to ensure that SS is set when control returns to the initial interrupt routine. As an example, this sequence of events might be

1) Single-step through the development code .

2) Encounter interrupt#1 (this is a non-single-step interrupt such as the display interrupt). The TMS34020 then automatically

   a) Stacks PC#1 and ST#1.
   b) Sets ST to 000000010h (the SS bit is now cleared).
   c) Branches to the first instruction of the interrupt routine (IR#1).

3) The debugger should have changed IR#1 so that its first instruction causes interrupt#2 (that is, IR#1's first instruction could be an illegal opcode). Now, the TMS34020 automatically

   a) Stacks PC#2 and ST#2.
   b) Sets ST to 000000010h (the SS bit is still clear).
   c) Branches to the first instruction of the illegal-opcode trap routine (IR#2).

4) IR#2 is installed by the debugger and could change the stacked value ST#2, setting ST#2's SS bit to 1. Finally, IR#2 should execute a RETM, after which the TMS34020 automatically pops the new ST#2 and PC#2 values before returning to IR#1. In IR#1, the TMS34020 passes control to the single-step trap routine before executing the instruction following the breakpoint.

5) The RETM that terminates the single-step causes control to pass back to IR#1 ,where the instruction following the breakpoint can execute. This and subsequent instructions in IR#1 execute in single-step mode.

6) Finally, at the end of IR#1, the terminating RETI executes, causing the TMS34020 to

   a) Pop ST#1 (which already has the SS bit set) and PC#1, and
   b) Resume single-stepping through the development code.

❑ **Differences between RETI and RETM.** The main difference between these two instructions is that RETM allows the TMS34020 to execute one instruction before returning to single-step instruction.

Chapter 7

Communicating with a Host Processor

A host processor can communicate with a TMS34020 through the TMS34020's **host interface**. The host interface allows you to map the TMS34020's local memory into a host's memory address space so that you can transfer data, commands, and status information between the host processor and the TMS34020's local memory.

This chapter describes host-interface operation.

7.1 Related Signals

Many of the TMS34020's pins are devoted to host-interface functions; the TMS34020 provides a 27-bit address bus, 4 byte-select signals, and 7 control lines. Chapter 2 describes the host-interface signals in detail; the signals are summarized below for your convenience.

Signals	Descriptions	I/O
HA5—HA31	is the 27-bit host address input bus. The host uses this bus to identify the address of a word in local memory that it requires access to.	I
HBS0—HBS3	are the 4 host byte-select inputs. The host uses the HBS signals in conjunction with HA5—HA31 to access specific bytes in a selected word.	I
$\overline{\text{HCS}}$	is the host chip-select signal. Driving this signal active low allows the host processor to access the TMS34020's local memory or I/O registers.	I
HDST	is the host data-strobe signal, driven low by the TMS34020 during host read accesses. Data is strobed into the transceivers on HDST's rising (trailing) edge.	O
$\overline{\text{HINT}}$	is the interrupt signal that allows the TMS34020 to send interrupt requests to the host processor.	O
$\overline{\text{HOE}}$	is the host output-enable signal. The TMS34020 drives this signal active low to allow the transceivers to output data onto the TMS34020's local address/data bus (LAD0—LAD31) during host writes.	O
HRDY	is the host ready signal that informs the host processor when the TMS34020 is ready to complete a host-initiated access cycle.	O
$\overline{\text{HREAD}}$	is the host read signal. Driving this signal active low allows the host processor to read the contents of a selected location in the TMS34020's local memory or I/O registers. The TMS34020 latches the requested data into the transceivers.	I
$\overline{\text{HWRITE}}$	is the host write signal. Driving this signal active low allows the host processor to write the contents of the transceivers to the selected location in the TMS34020's local memory or I/O registers.	I

7.2 Related Registers

Several of the TMS34020's I/O registers provide you with control over various aspects of the host interface. (Chapter 4 provides detailed descriptions of all the I/O registers.)

Note:

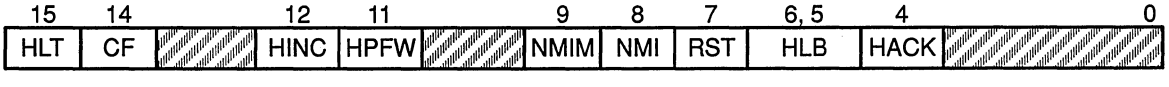
The *TMS34010* interface accessed certain I/O registers over its HAD bus. The TMS34020 does not access I/O registers this way. You must access an I/O register by its defined memory address within the TMS34020's local-memory space. For details, refer to Section 7.4 on page 7-7.

Five I/O registers are associated with the host interface:

- **HSTCTLL** and **HSTCTLH** control host-interface functions.
- **HSTADRH, HSTADRL,** and **HSTDATA** provide compatibility with the TMS34010.

The remainder of this section describes these registers, identifying the bits that are associated with host-interface functions. In the pictures of the registers, shaded areas identify bits that have no host-interface functions.

HSTCTLH register **address: C000 0100h**



Note: For a complete illustration and description of HSTCTLH, see Chapter 4.

HACK bit 4 — When the TMS34020 is halted, it sets the HACK (halt acknowledge) bit.

HLB bit 5 & 6 — The HLB (host last byte) bits form a 2-bit code. This code informs the TMS34020 which byte of a word the host accesses last.

RST bit 7 — Setting the RST (reset) bit causes the TMS34020 to execute a reset.

NMI bit 8 — The host processor can set the NMI (nonmaskable interrupt) bit to interrupt TMS34020 execution, causing the TMS34020 to execute an NMI routine.

NMIM bit 9 — When the NMIM (NMI mode) bit equals 0 and an NMI occurs, the TMS34020 saves the context of the interrupted program. If NMIM=1, the TMS34020 does not save the context when an NMI occurs.

HPFW bit 10 — When HINC=1, the HPFW (host prefetch after write) bit controls whether the TMS34020 prefetches (prereads) the next word in memory after a host read or write.

HINC bit 12 — Setting the HINC (host increment) bit enhances the host's ability to access blocks of TMS34020 memory; when HINC=1, the TMS34020 increments a host-supplied address and prefetches the contents from this new location.

CF bit 14 — Setting the CF (cache flush) bit flushes the contents of the TMS34020 instruction cache. The host processor can force the TMS34020 to execute new, downloaded code by flushing old instructions out of the cache.

HLT bit 15 — The host processor can halt the TMS34020's CPU by setting the HLT (halt TMS34020 program execution) bit.

HSTCTLL register **address: C000 00F0h**

Note: For a complete illustration and description of HSTCTLL, see Chapter 4.

MSGIN bits 0—2 — The MSGIN (message in) bits buffer a 3-bit interrupt message from the host processor to the TMS34020.

INTIN bit 3 — The host processor can set the INTIN (input interrupt) bit to 1 to generate an interrupt request to the TMS34020.

MSGOUT bits 4—6 — The MSGOUT (message out) bits buffer a 3-bit interrupt message from the TMS34020 to the host.

INTOUT bit 7 — The TMS34020 can set the INTOUT (output interrupt) bit to send an interrupt request to the host processor.

EMR & EMG bits 10 & 11 — An in-circuit emulator can use the EMR (emulator request) and EMG (emulator grant) bits as a mechanism for changing information and coordinating activity with the host processor. The information that these bits provide depends on the application, the host processor, and the software executed by the emulator. The emulator can set EMR to signal the host that an activity is beginning. The host can then set the EMG bit to acknowledge this. When the activity ends, the emulator signals the host by clearing EMR; the host then clears EMG. Only the emulator should modify EMR, and only the host should modify EMG. If you are not using this protocol with your emulator, these bits should always = 0.

EMIEN bit 12 — If you are using an in-circuit emulator, setting the EMIEN (emulator host interrupt enable) causes the exclusive-OR of EMR and EMG to interrupt the host by asserting a low level on the $\overline{\text{HINT}}$ pin.

Note:

If you are not using an in-circuit emulator, clear the EMG, EMR, and EMIEN bits to 0.

HRYI bit 13 — The TMS34020 sets the HRYI (host retry interrupt) bit if a retry occurs on a host access. If HBREN=1, setting HRYI sends an interrupt request (via $\overline{\text{HINT}}$) to the host processor.

HBFI bit 14 — The TMS34020 sets the HBFI (host bus-fault interrupt) bit if a bus fault occurs on a host access. If HBREN=1, setting HBFI sends an interrupt request (via $\overline{\text{HINT}}$) to the host processor.

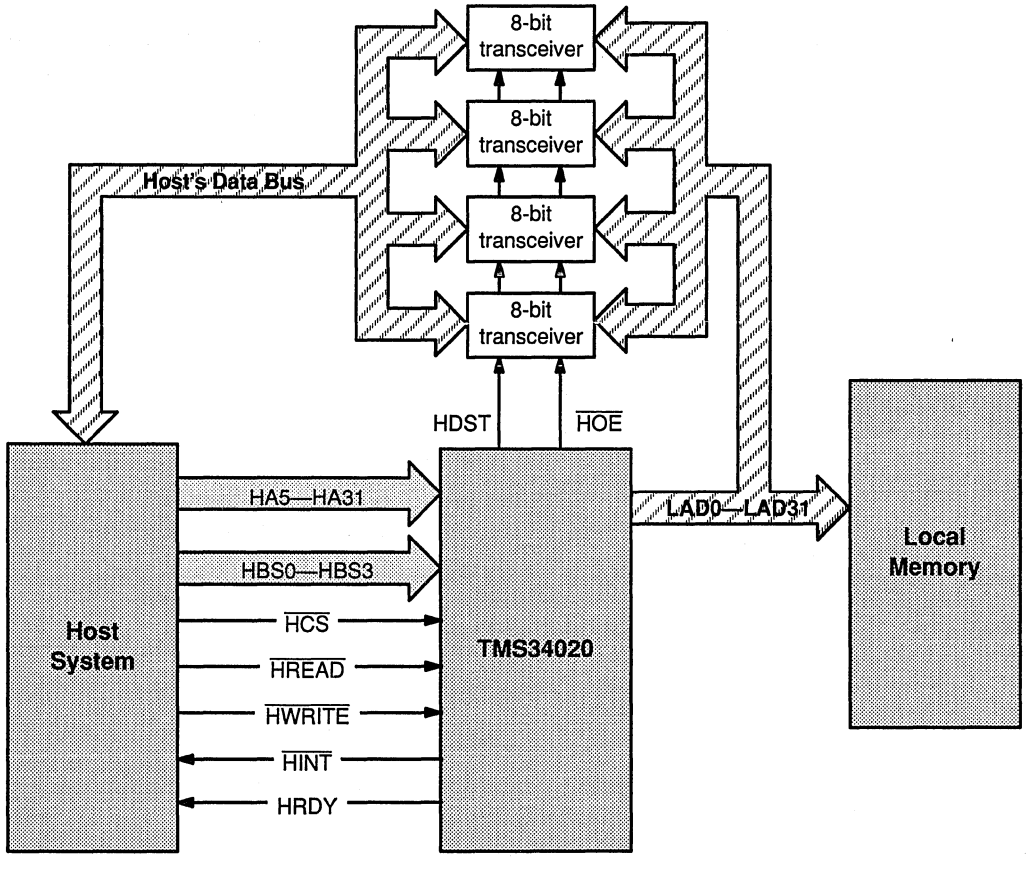
HBREN bit 15 — When the HBREN (host bus-fault/retry interrupt enable) bit is set, setting HRYI or HBFI causes an interrupt request to be sent to the host processor.

The TMS34010's host interface used the HSTADRH, HSTADRL, and HSTDATA registers for transferring address and data information. **The TMS34020's host interface does not use these registers.** The TMS34020 maintains these registers in the I/O register map to provide compatibility with TMS34010 software that uses these registers for passing information to the host interface. Because the TMS34020's host interface does not use HSTADRH, HSTADRL, or HSTDATA, any additional characteristics (such as autoincrementing) present for the TMS34010 are not implemented for the TMS34020. For more information about the TMS34020's treatment of these registers, refer to Section 1.6, Compatibility Between the TMS34020 and the TMS34010, on page 1-16.

7.3 A Basic Block Diagram for the Host Interface

In order for the TMS34020 to share data with a host processor, you must place bidirectional latching transceivers (such as the 74ALS652) between the TMS34020's local address/data bus and the host's data bus.

Figure 7–1. Block Diagram with a Host System, a TMS34020, and External Transceivers



Note:

Throughout this chapter, all references to *transceivers* refer to the external, bidirectional, latching, bus transceivers shown in Figure 7–1.

7.4 Basic Communication: How a Host Processor Reads from and Writes to TMS34020 Local Memory

The host processor initiates reads and writes through the TMS34020's host interface. The host controls the read and write cycles through a 2-step process:

Step 1: Provide address and byte information.

- Identify a long (32-bit) word address on the HA5—HA31 bus.
- Activate the appropriate HBS signal(s) to identify a specific byte (or bytes) within the selected word. A host processor accesses data in groups of 1 to 4 bytes. The host byte-select signals tell the TMS34020 two things:
  - *How many* bytes to access in the selected word **and**
  - *Which* bytes to access.

Figure 7–2 (page 7-8) shows several examples of how HBS0—HBS3 determine which bytes of a long-word are accessed.

Step 2: Set the $\overline{\text{HCS}}$, $\overline{\text{HREAD}}$, and $\overline{\text{HWRITE}}$ control signals.

- If the host wants to **read from** an address, it asserts both $\overline{\text{HCS}}$ and $\overline{\text{HREAD}}$ active low.
- If the host wants to **write to** an address, it asserts both $\overline{\text{HCS}}$ and $\overline{\text{HWRITE}}$ active low.

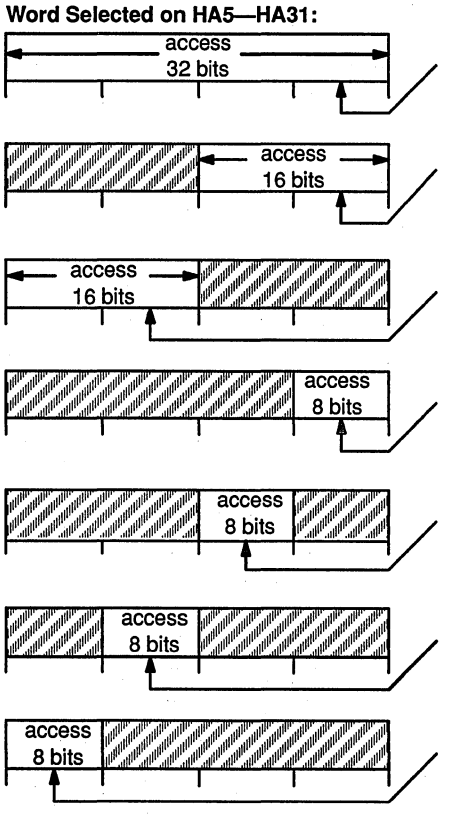
The host can assert or deassert the $\overline{\text{HCS}}$, $\overline{\text{HREAD}}$, and $\overline{\text{HWRITE}}$ control signals in any order. The last control signal that becomes active begins the access; the first control signal that becomes inactive ends the access. *The control signal that begins or ends an access is referred to as the **strobe** for the access.*

The address and byte selection information becomes valid on the falling edge of $\overline{\text{HCS}}$.

Note:

$\overline{\text{HREAD}}$ and $\overline{\text{HWRITE}}$ should not be active low simultaneously if $\overline{\text{HCS}}$ is also active low or if all 4 HBS signals are active. Having both signals low under these conditions could cause unpredictable host-interface behavior.

Figure 7–2. How a Host Processor Uses the Host Byte-Select Signals to Access Data in TMS34020 Memory



Select the 32 bits starting at byte 0 by driving these HBS lines:

Select the 16 bits starting at byte 0 by driving these HBS lines:

Select the 16 bits starting at byte 2 by driving these HBS lines:

Select the 8 bits starting at byte 0 by driving these HBS lines:

Select the 8 bits starting at byte 1 by driving these HBS lines:

Select the 8 bits starting at byte 2 by driving these HBS lines:

Select the 8 bits starting at byte 3 by driving these HBS lines:

Byte address on HBS3—HBS0

HBS3	HBS2	HBS1	HBS0
1	1	1	1
0	0	1	1
1	1	0	0
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

Note: All other combinations of HBS0—HBS3 are valid as well.

7.4.1 How a Host Processor Requests a Read Cycle

When a host processor wants to read data from a TMS34020 memory location, the host must

1) Provide address and byte-select information over HA5—HA31 and HBS0—HBS3, **then**

2) Assert $\overline{\text{HCS}}$ and $\overline{\text{HREAD}}$ (in either order).

The host must keep $\overline{\text{HREAD}}$ and $\overline{\text{HCS}}$ active until the HRDY signal becomes active; when HRDY becomes active, the host can terminate the access and read the data from the transceivers.

The host must provide the transceivers with an output-enable signal that tells the transceivers to transfer their information to the host's data bus. No matter what size bus the host processor has, the transceivers must be capable of handling 32-bit information (for example, you could use four 8-bit transceivers).

Regardless of how many bytes the host requests, the TMS34020 latches the entire 32 bits of data into the transceivers; the host's controls must enable the appropriate transceivers to select the desired bytes.

7.4.2 How a Host Processor Requests a Write Cycle

When a host processor wants to write data to a TMS34020 memory location, the host must

1) Provide address and byte-select information over HA5—HA31 and HBS0—HBS3, **then**

2) Assert $\overline{\text{HCS}}$ and $\overline{\text{HWRITE}}$ (in either order).

The host must keep $\overline{\text{HWRITE}}$ and $\overline{\text{HCS}}$ active until the HRDY signal becomes active. When HRDY becomes active, the host can terminate the request and latch the data-to-be-written into the transceivers.

The TMS34020 uses a byte-write feature that ensures that the host modifies only the selected bytes. The host must correctly align the data in the transceivers. The TMS34020 ignores the transceiver contents of unselected bytes.

7.4.3 Local-Memory Faults and Retries

It is possible, in some applications, that the host may request access to locations in the TMS34020's local memory, which may cause a retry or bus fault. During the local-memory access initiated in response to a host request, a retry or bus-fault completion code could occur on the LRDY or BUSFLT pins, respectively.

- If a host access causes the local memory to generate a retry, the TMS34020 automatically reschedules the access. The HRDY signal is not asserted until the TMS34020 successfully completes the access. In addition, the TMS34020 sets HRYI[HSTCTLH] to indicate to the host that the retry occurred. No further host accesses can be made until the retried access completes successfully.

- If a host access causes the local memory to generate a bus fault, the TMS34020 sets HBFI[HSTCTLH] to indicate to the host that the bus fault occurred; the TMS34020 takes no further action. HRDY is asserted as if the access terminated normally. It is then the host's responsibility to take the appropriate action (if any) to clear the cause of the bus fault. (The appropriate action depends on your application.)

Once HRYI or HBFI is set, it remains set until the host or the TMS34020 explicitly clears it.

Setting HBREN[HSTCTLH] to 1 causes the TMS34020 to assert the $\overline{\text{HINT}}$ pin active low if either HRYI or HBFI is set. This allows a retry or bus fault on a host access to directly interrupt the host.

7.5 Features That Improve Performance of the Host Interface

This section describes several features that increase the performance and efficiency of the host interface.

7.5.1 Prefetching Data from the TMS34020's Local Memory

Prefetching (or prereading) information is an optional feature that can speed up the transfer of information. Instead of waiting for the host's next request, the TMS34020 fetches data from the next consecutive long-word address. The new data is fetched as soon as the current access completes and is placed in the transceivers in anticipation of the host's next request. Prefetching is beneficial for both reads and read(modify)writes:

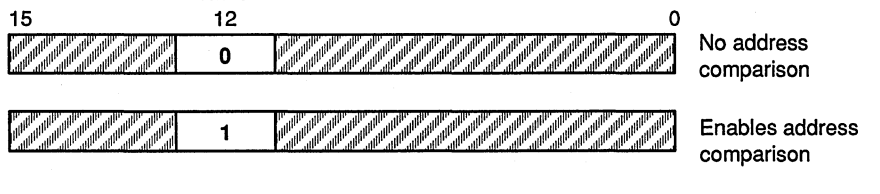
- After **reads**, prefetching enables efficient reads of contiguous memory locations.
- After **writes**, prefetching enables efficient read(modify)writes of contiguous memory locations.

You can enable the prefetching feature by setting HINC[HSTCTLH] to 1. Once this is done, you must choose to prefetch information after reads or after writes. You control this by setting or clearing HPFW[HSTCTLH]. Figure 7–3 shows how different combinations of HINC and HPFW values affect prefetching.

Figure 7–3. How the Values of HINC[HSTCTLH] and HPFW[HSTCTLH] Affect Prefetching

The TMS34020 has an address-comparison feature that ensures the host interface always accesses the correct location. Figure 7–4 shows how you can enable the address-comparison feature.

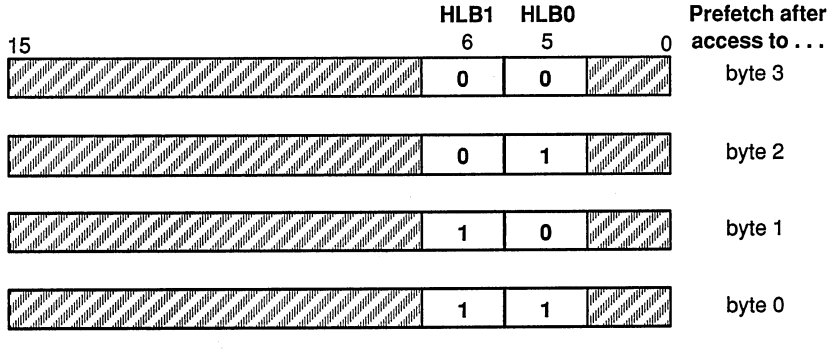
Figure 7–4. How the Value of HINC[HSTCTLH] Affects Address Comparison



- **Prefetching enabled.** On each read access, the host interface compares the address provided by the host with the address of the prefetched data. If the two addresses differ, the host interface automatically reads the new location requested by the host. This ensures that the host always reads from the address it requests, even when the TMS34020 prefetches data from a different location. This allows the host to access one contiguous block of words and then access another contiguous block, discontiguous from the first block, without disabling prefetching.
- **Prefetching disabled.** The TMS34020 performs no address comparison. Each time the host makes a read request, the TMS34020 explicitly copies the data from the requested location into the transceivers.

The size of a host's data bus can affect prefetching. If the host does not have a 32-bit data bus, then the host needs more than one access to fully read or write a 32-bit word. In this case, the TMS34020 must know which byte of the word the host will access last. This information is conveyed by the value of the HLB[[HSTCTLH]] bits. Figure 7–5 shows how this value identifies which byte the host will access last.

Figure 7–5. How the Value of HLB[[HSTCTLH]] Affects Prefetching



The HLB bits allow the host interface to accommodate all possible byte-ordering conventions and several different data-bus sizes.

- A host processor with an 8-bit data bus can select any of the 4 bytes as the last byte. For example, assume that such a host processor is reading a contiguous block of TMS34020 local memory; assume also that the host sets HLB to 00_2 (designating byte 3 as the last byte that is prefetched). The host interface does not prefetch the next word until after the host reads byte 3 of the current word. All 32 bits of data are placed into the transceivers when the first byte is read; no memory accesses are required to read the subsequent 3 bytes because the transceivers already have the data.
- For a 16-bit host bus, only the value of HLB1 (bit 6) is needed to determine which 16-bit word the host accesses last.
- A 32-bit host need not program the HLB bits.

Although a non-32-bit host must make multiple read requests, the TMS34020 reads the location only once—on the first access (or after the designated last byte of the previous access). From this single read, the TMS34020 stores all 32 bits of data from the requested long-word address into the transceivers. Subsequent host accesses to read the remaining bytes within the word do not generate local-memory accesses because the requested data is already in the transceivers. Only after the designated last byte is accessed is a local-memory access initiated; this copies the data from the *next* 32-bit location, preparing for the next host access. The TMS34020's single-read feature minimizes the extra time required by a non-32-bit host to read an entire long-word.

Note:

The TMS34020 ensures that the host always reads the current contents of a memory location. Each read that follows a host-initiated write requests a local-memory cycle to explicitly copy the location's contents into the transceivers. This happens even if a previous read copied this address into the transceivers. Thus, while using prefetch-after-write (HINC=1, HPFW=1) with a non-32-bit host, each write causes a prefetch, regardless of whether or not the write was to a designated last byte. If the write was not to the designated last byte, the same location is prefetched again. Only after the last byte is written is the address incremented and the next location prefetched.

If prefetches and address-comparison are disabled, each host read request initiates a local-memory read cycle. This cycle copies the contents of the requested address into the transceivers, even if the data from that location is already stored in the transceivers.

7.5.2 Autoincrementing (Implicit Addressing)

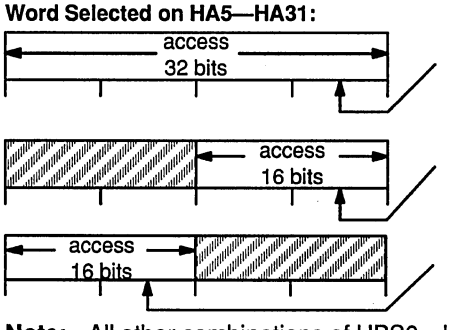
The TMS34020's autoincrementing feature allows a host processor to address a contiguous block of local memory by specifying the address of only the first word in a block. The TMS34020 automatically generates all subsequent addresses and byte selects. A host can access contiguous blocks of words 16 bits or 32 bits at a time.

To use autoincrementing, the host asserts $\overline{\text{HCS}}$ active low at the beginning of the first access, providing the address and byte selects just as it does for a regular access. The host must maintain $\overline{\text{HCS}}$ active low throughout; the TMS34020 enables autoincrementing when it detects that $\overline{\text{HCS}}$ remains low between the end of the previous access and the beginning of the current access. The host then uses $\overline{\text{HREAD}}$ or $\overline{\text{HWRITE}}$ as a strobe to request access to subsequent addresses.

When autoincrementing is detected, the host interface uses the value of HBS0—HBS3 to determine whether the host is accessing the block 16 bits or 32 bits at time. If all 4 byte selects are active, the TMS34020 assumes that the host needs only 1 access to completely read or write a 32-bit location. If only

2 byte selects are active (HBS0 and HBS1, or HBS2 and HBS3), the TMS34020 assumes that the host requires 2 accesses to completely read or write a 32-bit location. Figure 7–6 shows valid byte-select combinations.

Figure 7–6. Legal Host Byte-Select Combinations for Autoincrementing



Select 32-bit autoincrementing by driving these HBS lines:

Select 16-bit autoincrementing, starting at byte 0, by driving these HBS lines:

Select 16-bit autoincrementing, starting at byte 2, by driving these HBS lines:

Byte address on HBS3—HBS0

HBS3	HBS2	HBS1	HBS0
1	1	1	1

HBS3	HBS2	HBS1	HBS0
0	0	1	1

HBS3	HBS2	HBS1	HBS0
1	1	0	0

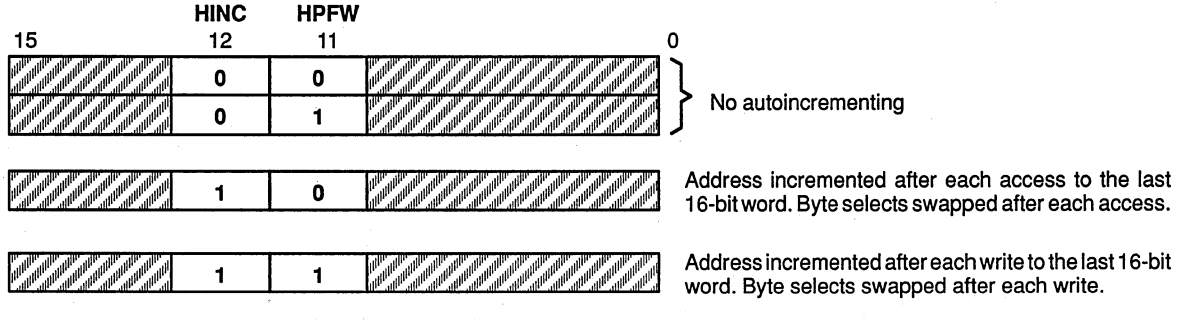
Note: All other combinations of HBS0—HBS3 are invalid for autoincrementing

The TMS34020 latches the value of HBS0—HBS3 at the falling edge of $\overline{HCS}$. If the next access will be made to the subsequent 16-bit or 32-bit location, the TMS34020 automatically generates the byte selects for the access by swapping its internal copy of the byte selects after each access. Thus, if the latched value of HBS0—HBS3 is 0011_2, then the internal copy is swapped to 1100_2 at the beginning of the next access. This in turn is swapped back to 0011_2 at the beginning of the following access, and so on. Similarly, if the value of HBS0—HBS3 latched at the falling edge of $\overline{HCS}$ is 1100_2, it is swapped to 0011_2 at the beginning of the next access. In this way, the internal copy of the byte selects alternates between selecting one 16-bit word and the other. Obviously, if HBS0—HBS3 is 1111_2, swapping produces the same value.

Just as for regular accesses, the TMS34020 uses the HLB code to determine when the host has completely accessed a 32-bit word. It does this by comparing the internal copy of the byte selects with the HLB bits. The TMS34020 increments the address only after the host accesses the designated last 16-bit word.

Figure 7–7 shows how HINC[[HSTCTLH]] and HPFW[[HSTCTLH]] control how the address is incremented and when the byte selects are swapped.

Figure 7–7. How the Values of HINC[[HSTCTLH]] and HPFW[[HSTCTLH]] Affect Autoincrementing



The following list describes the byte-swapping operation for different cases.

- **Autoincrementing for reads *and* writes.**

HINC=1	HPFW=0	Host can read & write contiguous locations

The TMS34020 swaps the internal copy of the byte selects after each access. After each access to the designated last 16-bit word, the TMS34020 swaps the the byte selects, then increments the address. If the access is a read, the next 32-bit location is prefetched into the transceivers. In this mode of operation, all autoincrementing accesses to a contiguous block of addresses should be of the same type—either reads or writes, but not a mixture of the two.

- **Autoincrementing after writes *only*.**

HINC=1	HPFW=1	Host can read-modify-write contiguous locations

The TMS34020 swaps the internal copy of the byte selects only after writes. After each write to the designated last 16-bit word, the address is incremented, and the next 32-bit location is prefetched into the transceivers. No modifications to the address or the byte selects are made after reads, so the host writes back to the same location. In this mode of operation, the host should perform read and write requests alternately; the first access can be either a read or a write. Note that in 16-bit autoincrement mode, a location is read only once. Because the sequence of reads and writes is predefined, there is no need to reread the 32-bit location after the first write.

- **No autoincrementing.**

HINC=0	HPFW=don't care	Host can't autoincrement

It is not anticipated that this mode of operation will be very useful. However, for completeness, the response to autoincrement accesses while HINC=0 is described below.

- **For writes**, the TMS34020 accesses the same 32-bit address on **each** host request. If 16-bit autoincrementing is selected, the byte selects are still swapped, so successive accesses oscillate between the two 16-bit words within the specified 32-bit location.
- **For reads**, the TMS34020 copies all 32 bits of data from the specified location into the transceivers during the first access. On subsequent reads in autoincrement mode, no new data is transferred into the transceivers.

In this mode, all accesses should be of the same type—either reads or writes, but not a mixture of the two.

If the host wants to access an address that is not contiguous with the block, it must deassert $\overline{HCS}$ and provide a new address just as it would for a regular access.

When autoincrementing, the actual local-memory cycles performed by the TMS34020 in response to host requests do not differ from those performed when not autoincrementing. Autoincrement accesses offer no throughput advantage compared to similar non-autoincrement accesses.

7.5.3 The TMS34020's Default Memory Cycle

When no other memory requests are pending, the TMS34020's memory controller executes a state similar to the address subcycle of a host-initiated local-memory cycle. This is known as the **host-default state**. The only difference between the host-default state and the address subcycle is that $\overline{ALTCH}$ and $\overline{RAS}$ do not go active. If a host request is synchronized while a host-default state is being executed, the host-default state is converted into a regular host access by internal control logic, which causes $\overline{ALTCH}$ and $\overline{RAS}$ to go active. This action reduces the cycle time required for host-initiated memory cycles by one LCLK cycle because it allows the first half of the host access to occur in parallel with synchronization of the request.

The address output on LAD0—LAD31 and RCA0—RCA12 during a host-default cycle is the contents of the internal register used for storing the address presented by the host. If the host-default cycle is converted into a real host-initiated memory cycle, then the synchronization delays inherent in the host-interface logic ensure that the address output is the correct one for the access. However, at other times (when the host-default memory cycle is not converted into a real host access) the address output on LAD0—LAD31 and RCA0—RCA12 is not valid. Because $\overline{ALTCH}$ and $\overline{RAS}$ do not go active in this case, this should not matter.

7.6 Completing Host Accesses

When the TMS34020 is ready to complete a host-initiated access, it asserts HRDY active high. *The host must not deassert its request strobe until HRDY becomes active.* By default, the TMS34020 maintains HRDY in an inactive-low state. After the host terminates a request, HRDY becomes inactive low again.

Various conditions govern when HRDY can become active-high. These are described in the following sections.

7.6.1 Activating HRDY for Host Reads

When prefetches are disabled and the host requests a read to any address, or when prefetches are enabled and the host requests a read to a location other than the one currently copied into the transceivers, the TMS34020 fetches the data and places it in the transceivers.

The leading edge of this type of host request triggers the TMS34020 to access the requested address. During the last machine state of the memory access, HRDY becomes active. This happens while the data is being transferred into the transceivers from the local memory and HDST is active low. The data is not guaranteed valid in the transceivers until HDST becomes high. The host can deassert the read-request strobe as soon as HRDY goes active, but must not read the data from the transceivers while HDST is low. HRDY is asserted before the data is valid so that it is possible for the host to make requests sufficiently close together that the maximum throughput of the TMS34020's local-memory interface can be used. This is discussed in more detail in Section 7.9.2.

7.6.2 Activating HRDY for Host Writes

When the host requests a write, the TMS34020 asserts HRDY when it is ready for the host to deassert its write-request strobe. This trailing edge of the host write-request triggers the local-memory cycle required to perform the write, which therefore occurs after the host terminates its request. The data for the write must be latched into the transceivers and valid before $\overline{\text{HOE}}$ becomes active low. This could be as few as 1.25 LCLK cycles after the host write-request strobe is deasserted.

If the host requests another write, the actions taken depend on whether any access previously initiated by the host is still pending:

- If no incomplete memory accesses are pending or in progress from the previous host write, HRDY immediately becomes active and the host can latch new data into the transceivers.
- If the previous host write is not finished, HRDY remains low until the last machine state of the memory access that writes the contents of the transceivers to the requested memory location. HRDY becomes active while

$\overline{HOE}$ is still active low, enabling the contents of the transceivers onto LAD0—LAD31. The host can deassert the write-request strobe as soon as HRDY goes active, but must not latch new data into the transceivers while $\overline{HOE}$ is low. HRDY is asserted while the previous data is still being written so that it is possible for the host to make requests sufficiently close together that the maximum throughput of the TMS34020's local-memory interface can be used. This is discussed in more detail in Section 7.9.2.

7.6.3 Activating HRDY for Host Reads and Writes after Prefetches

When prefetching is enabled (HINC=1), the trailing edge of the host's read-request strobe (if HPFW=0) or write-request strobe (if HPFW=1) on an access to the designated last byte of the current location automatically initiates a prefetch of the next location. If the host makes a request while a local-memory cycle to prefetch data into the transceivers is in progress, HRDY cannot become active high until that memory cycle is almost complete. It is likely, if you are trying to use the maximum throughput of the TMS34020's local-memory interface, that the prefetch memory cycle will not have completed when the host makes its next read or write request.

Here are the rules that govern when HRDY becomes active.

- **Writes.** If a prefetch is pending or in progress when the host requests an access, HRDY remains inactive until the last machine state of the prefetch.
- **Reads.** The address to be read is compared with the address of the location that is currently copied into the transceivers. The action taken depends on whether the addresses match and whether the prefetch has completed:
  - **Addresses match, prefetch complete.** HRDY immediately becomes active. Because the requested data is already in the transceivers and HDST is already inactive-high, the data is valid immediately.
  - **Addresses match, prefetch in progress.** HRDY becomes active in the last machine state of the prefetch while HDST is still low. The data is valid in the transceivers as soon as HDST becomes high.
  - **Addresses do not match, prefetch complete.** The host initiates another read to the required location before HRDY becomes active.
  - **Addresses do not match, prefetch in progress.** The host interface waits for the prefetch access to complete. Then it initiates another read to the required location before HRDY becomes active.

  In the second, third, and fourth cases described above, HRDY becomes active while HDST is low. HRDY is asserted before the data is valid. This allows the host to make requests quickly enough to use the TMS34020's maximum local-memory bandwidth. This is discussed in more detail in Section 7.9.2 (page 7-35).

7.7 Timing Examples

This section contains timing diagrams that illustrate the host interface's response to various host requests. Each host-access request to a new 32-bit location in the TMS34020's local memory or I/O registers generates a local-memory cycle. The prefetch mechanisms described in Section 7.5.1 (page 7-10) also generate local-memory accesses.

Each diagram shows the host interface signals at the top. The buses marked *DATA (in)* and *DATA(out)* indicate the validity of the data in the transceivers for host writes and reads, respectively.

Below the host signals, a scale marks each quarter phase of a TMS34020 machine state (Q1—Q4). Below this scale are the local-memory buses and control signals. This illustrates the relationship between the host requests and the resulting local-memory cycles.

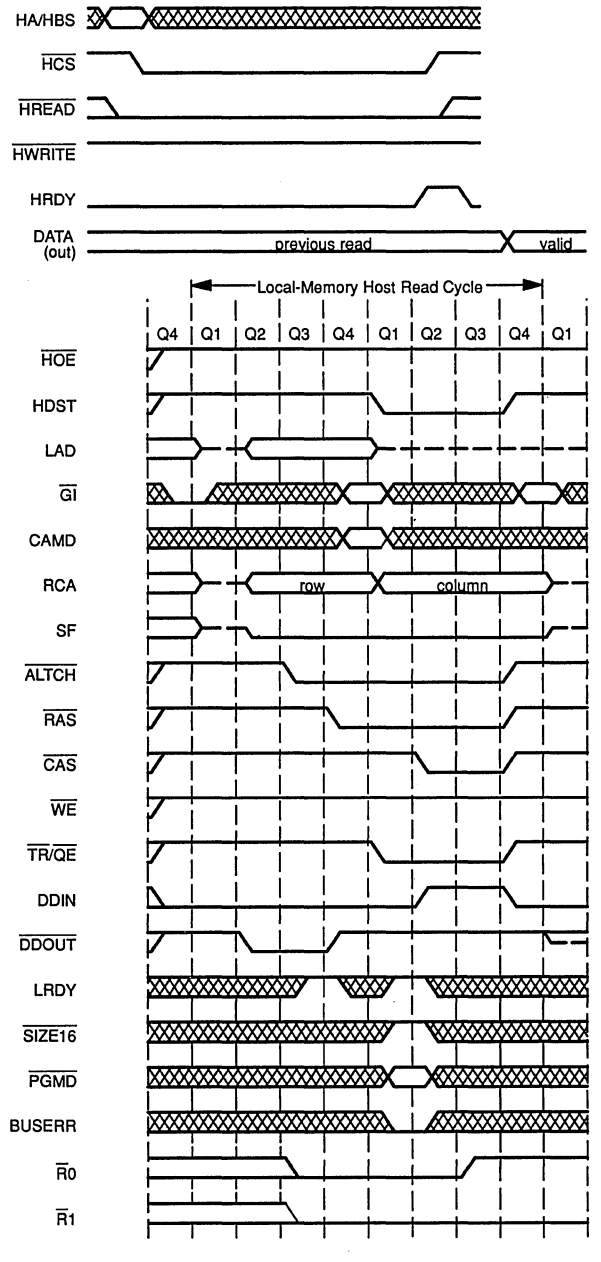
All the diagrams assume that:

- The local-memory interface is always immediately available to the host.
- There are no wait states (unless otherwise noted).

With the exception of HRDY, no timing relationships are implied between the host-interface signals above the scale and the local-memory signals below it. However, in order to achieve optimum performance, the synchronization time between the host interface and the local-memory interface should be minimized. The local-memory access can be initiated in the next machine state only if the host request strobe that initiates the local-memory access occurs before the fourth quarter phase (Q4) of the current machine state. If the strobe occurs after the beginning of Q4 of the current machine state, the earliest that the access can begin is two machine states later. The timing diagrams illustrate this synchronization (Section 7.9, page 7-34, discusses this in more detail).

Some of the timing diagrams show $\overline{\text{HCS}}$ being asserted (or deasserted) before $\overline{\text{HREAD}}$ or $\overline{\text{HWRITE}}$; some show $\overline{\text{HREAD}}$ or $\overline{\text{HWRITE}}$ being asserted (or deasserted) before $\overline{\text{HCS}}$. This emphasizes that any assertion order is allowed and implies no relationship between the assertion order required for the request and the nature of the request. The signal that begins (or ends) an access and that initializes a local-memory cycle for the host is referred to as the **host request strobe** for the access.

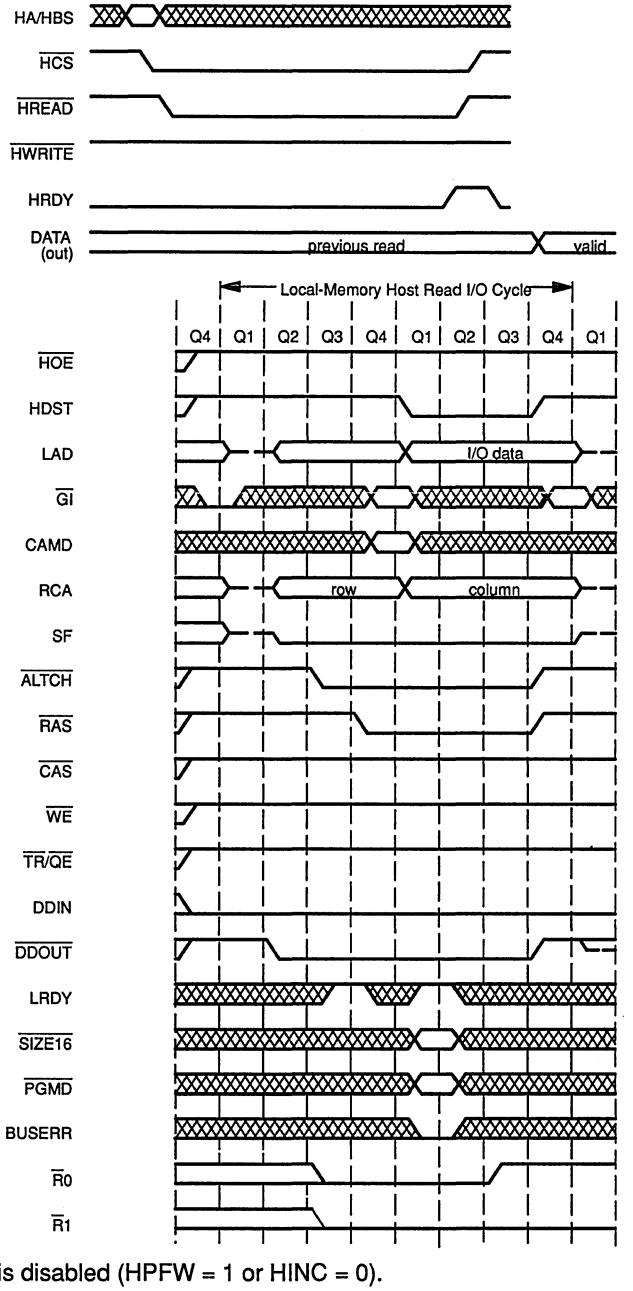
Figure 7–8. Single Host Read Cycle; $\overline{HCS}$ Used as Strobe



Notes: 1) Prefetch after read is disabled (HPFW = 1 or HINC = 0).
2) This figure refers to host reads of TMS34020 local memory and does not include host reads of TMS34020 I/O registers.

Figure 7–8 shows a simple read cycle; the host is accessing a location in local memory. The leading edge (high-to-low transition) of the host request strobe initializes the local-memory cycle.

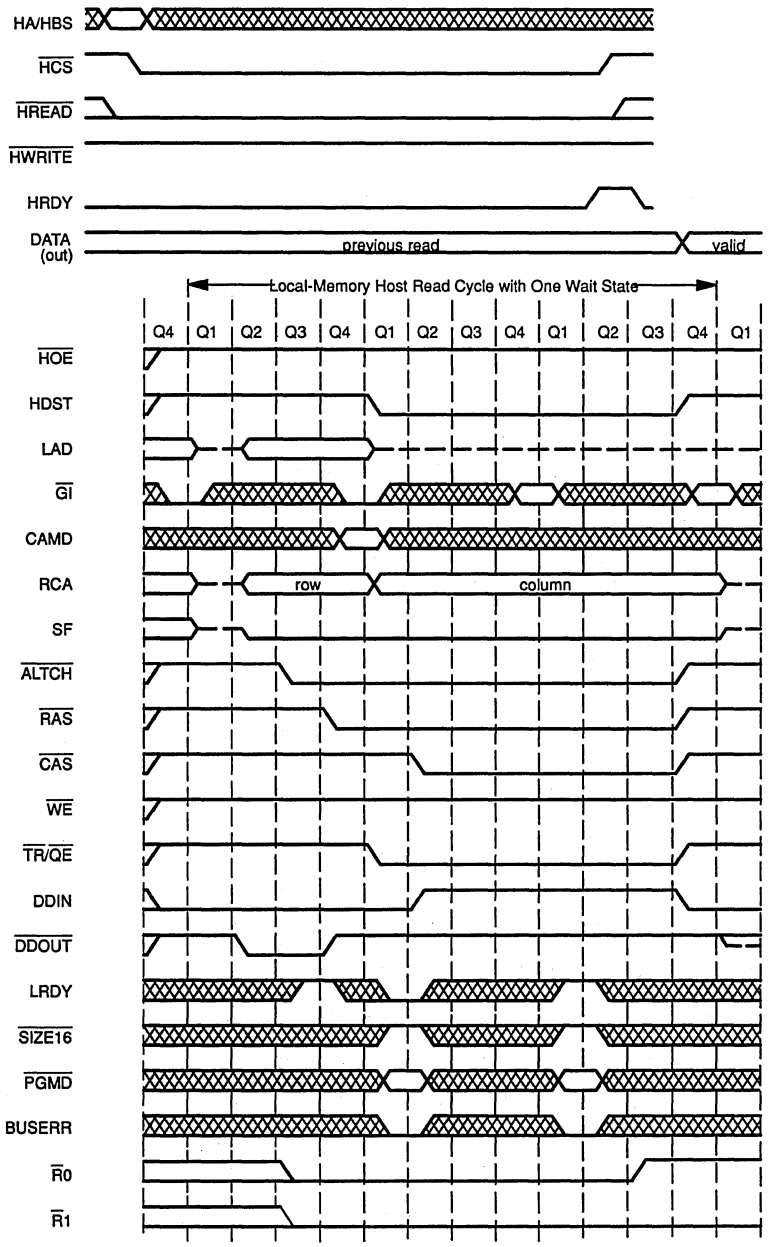
Figure 7–9. Single Host Read from I/O Registers; $\overline{\text{HREAD}}$ Used as Strobe



Note: Prefetch after read is disabled (HPFW = 1 or HINC = 0).

Figure 7–9 shows a simple read cycle; the host is accessing an I/O register. The leading edge of the strobe initializes the local-memory cycle. Comparing this figure to the previous figure, you'll see that to the host, there's little difference between accessing an I/O register and accessing the rest of the local memory. However, some of the local-memory strobes are different; $\overline{\text{CAS0}}$—$\overline{\text{CAS3}}$, DDIN, and $\overline{\text{TR/QE}}$ are not activated because the data is transferred from within the TMS34020, not from the local memory.

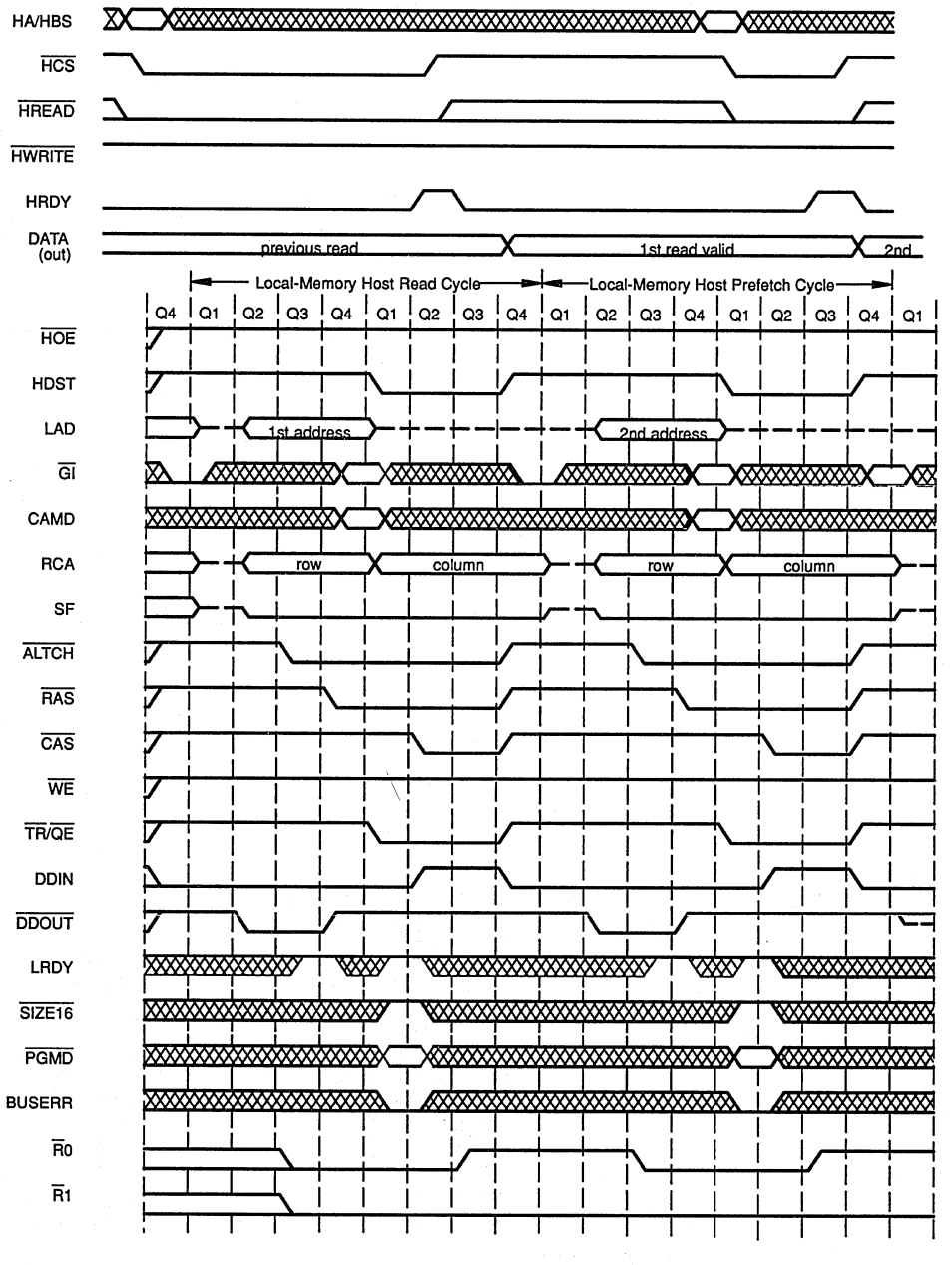
Figure 7–10. Single Host Read with One Wait State; $\overline{HCS}$ Used as Strobe



Notes: 1) Prefetch after read is disabled (HPFW = 1 or HINC = 0).
2) This figure refers to host reads of TMS34020 local memory and does not include host reads of TMS34020 I/O registers.

Figure 7–10 is identical to Figure 7–8 except that the memory cycle includes one wait state. Both figures show a simple read cycle in which a host processor is accessing a location in the TMS34020's local memory. The leading edge of the host request strobe initializes the local-memory cycle.

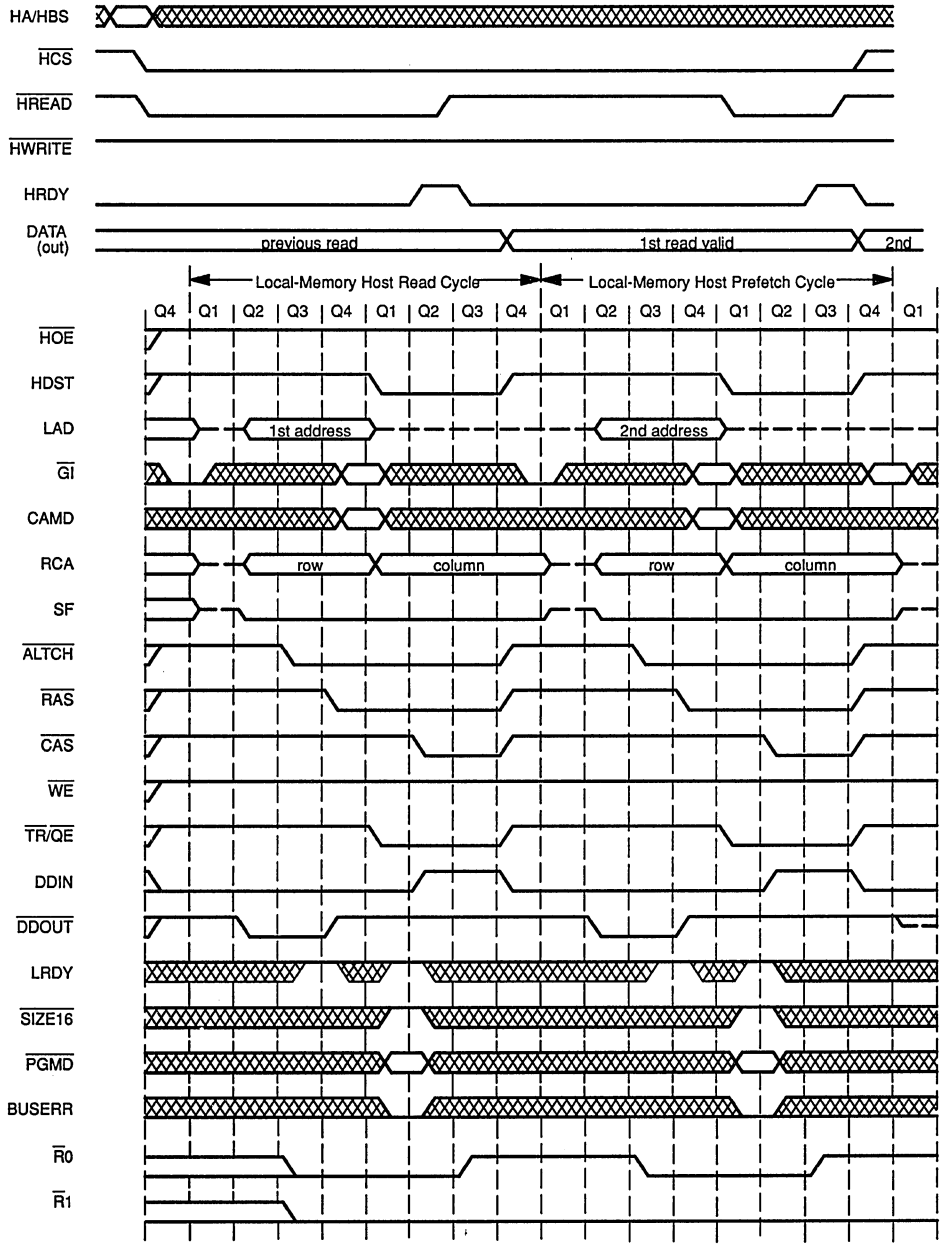
Figure 7–11. Host Read Back-to-Back with Prefetch of Next Word; $\overline{HCS}$ Used as Strobe



Notes: 1) Prefetch after read is enabled (HINC = 1 and HPFW = 0).
2) This figure refers to host reads of TMS34020 local memory and does not include host reads of TMS34020 I/O registers.

Figure 7–11 shows 2 back-to-back read cycles; the second location is prefetched after the first location is read. The trailing edge (low-to-high transition) of the host request strobe initiates the prefetch local-memory cycle.

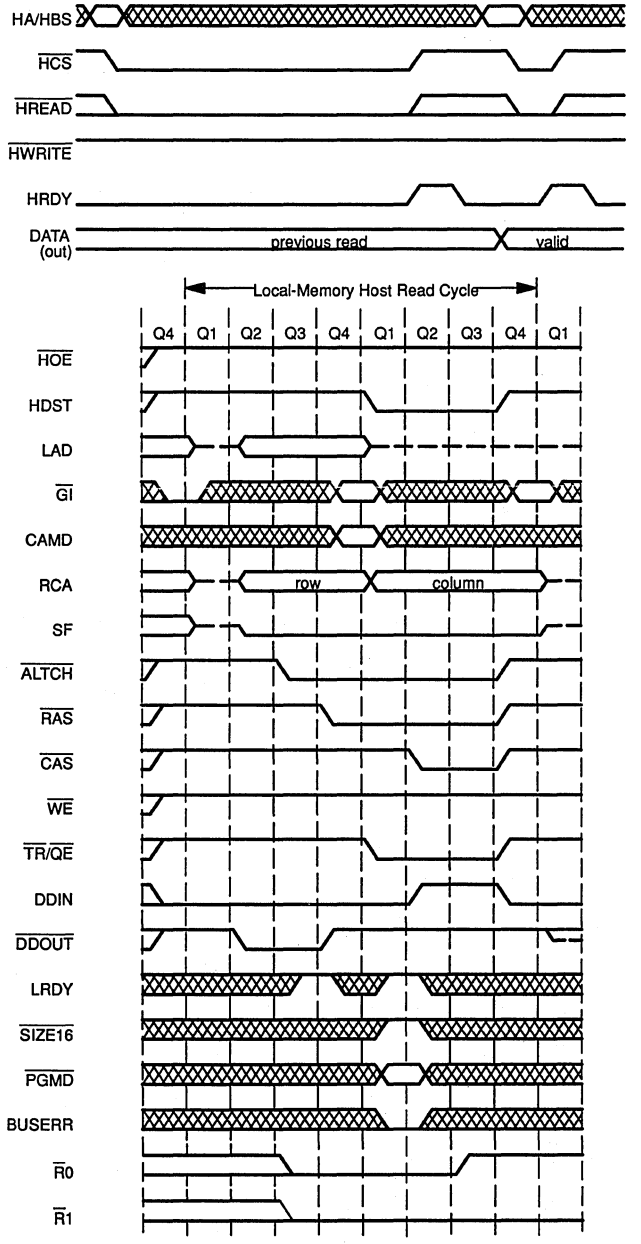
Figure 7–12. Back-to-Back Host Read Cycles with Implicit Addressing; $\overline{\text{HREAD}}$ as Strobe



Notes: 1) Prefetch after read is enabled (HINC = 1 and HPFW = 0).
2) This figure refers to host reads of local memory and does not include host reads of I/O registers.

Figure 7–12 shows 2 back-to-back read cycles; the second location is prefetched after the first location is read. The trailing edge of the host request strobe initiates the prefetch. Just as in Figure 7–11, the prefetch relaxes the timing of the request strobe. Figure 7–12 also illustrates implicit addressing. Although not explicitly shown, the value on HBS0—HBS3 must be 1111_2, 0011_2, or 1100_2. The initial address is the one supplied at the falling edge of $\overline{\text{HCS}}$.

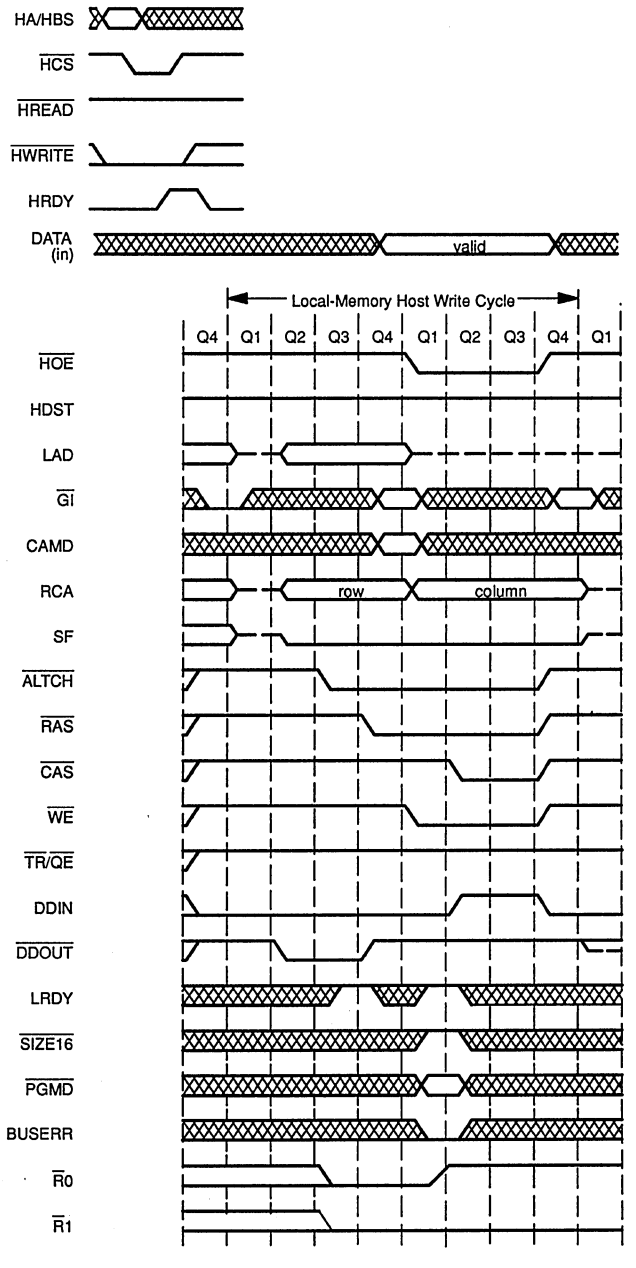
Figure 7–13. Successive Reads to Same 32-Bit Location; $\overline{HCS}$ and $\overline{HREAD}$ Strobed Together



Notes: 1) Prefetch after read or write is enabled (HINC = 1 and HPFW = don't care).
2) This figure refers to host reads of TMS34020 local memory and does not include host reads of TMS34020 I/O registers.

Figure 7–13 shows successive reads to the different bytes at the same location. Notice how repeatedly accessing the same word carries little overhead because the TMS34020 accesses the data only once. The HINC bit must equal 1 (enabling prefetching) to support the necessary address comparison.

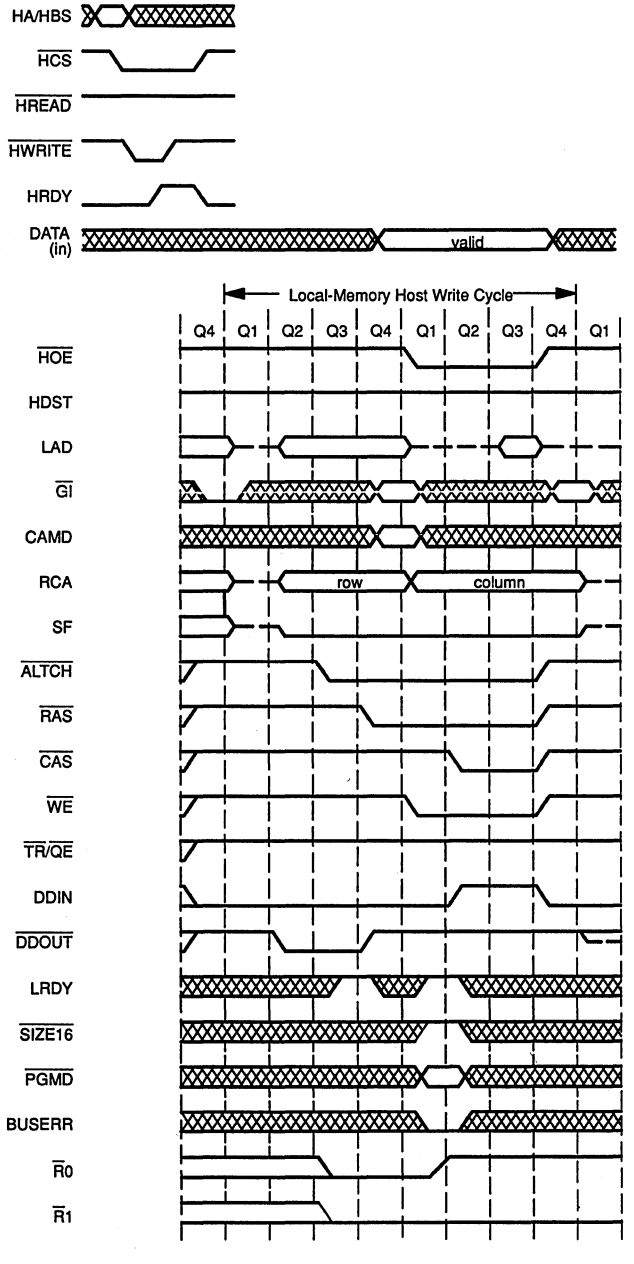
Figure 7–14. Single Host Write Cycle; $\overline{HCS}$ Used as Strobe



Notes: 1) Prefetch after write is disabled (HPFW = 1 or HINC = 0).
2) This figure refers to host writes to TMS34020 local memory and does not include host writes to TMS34020 I/O registers.

Figure 7–14 shows a simple write cycle; the host is writing to a location in local memory. The trailing edge of the host request strobe initializes the local-memory cycle.

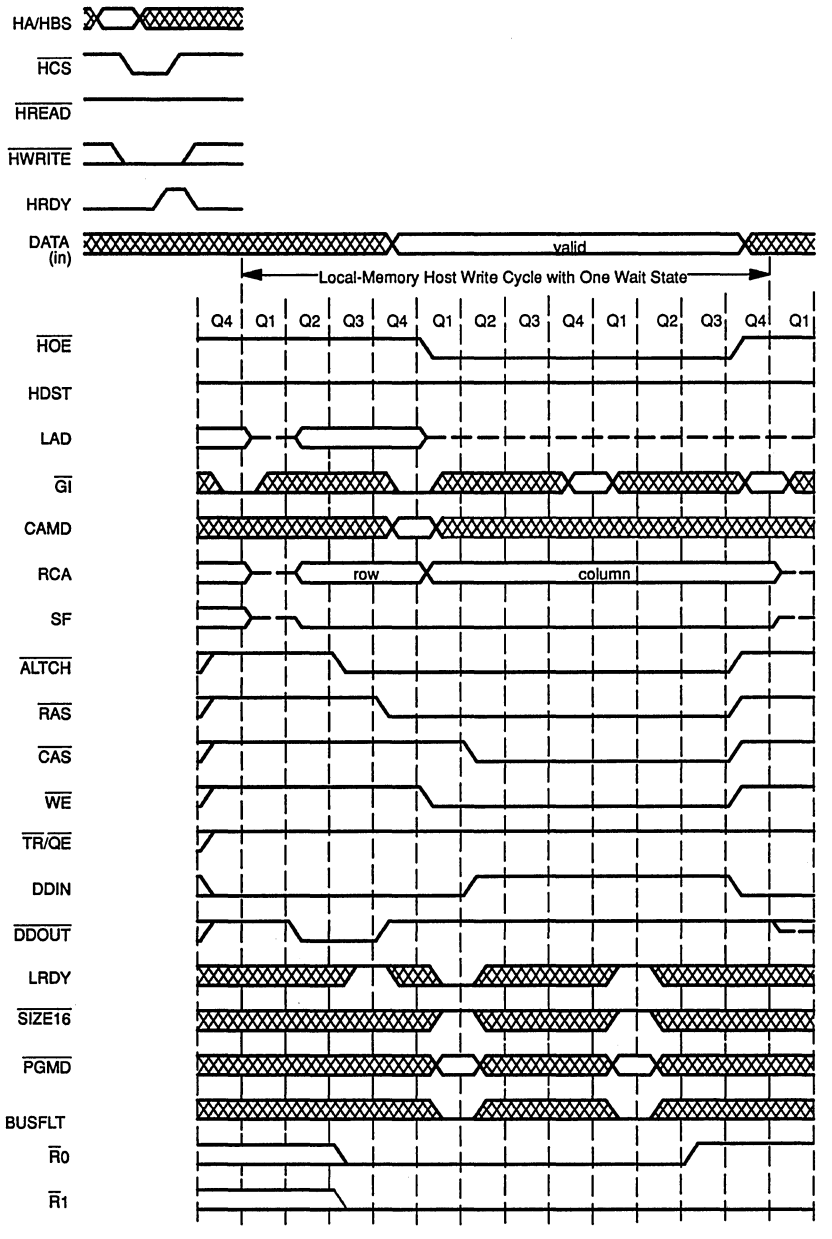
Figure 7–15. Single Host Write Cycle to I/O Registers; $\overline{\text{HWRITE}}$ Used as Strobe



Note: Prefetch after write is disabled (HPFW = 0 or HINC = 0).

Figure 7–15 shows a simple write cycle; the host is writing to an I/O register. The trailing edge of the host request strobe initializes the local-memory cycle. Comparing Figure 7–15 to Figure 7–14, you'll see that to the host, there's little difference between writing to an I/O register appears and accessing any other memory location. However, data is latched into the TMS34020 on LAD31—LAD0.

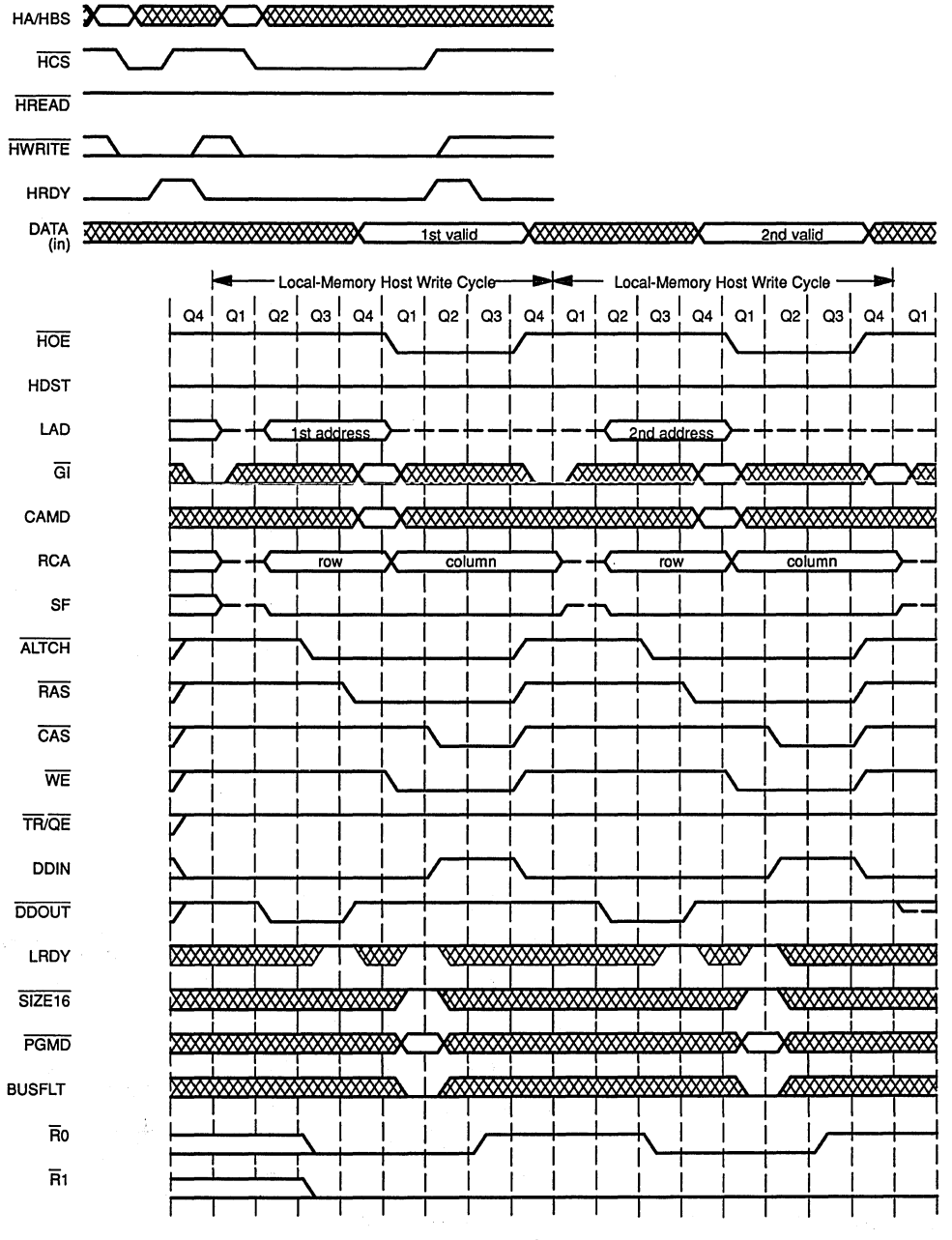
Figure 7–16. Single Host Write Cycle with One Wait State; $\overline{HCS}$ Used as Strobe



Notes: 1) Prefetch after write is disabled (HPFW = 0 or HINC = 0).
2) This figure refers to host writes to TMS34020 local memory and does not include host writes to TMS34020 I/O registers.

Figure 7–16 shows a simple write cycle; the host is writing to a location in local memory. The trailing edge of the host request strobe initializes the local-memory cycle. Figure 7–16 is identical to Figure 7–14 except that the memory cycle includes one wait state.

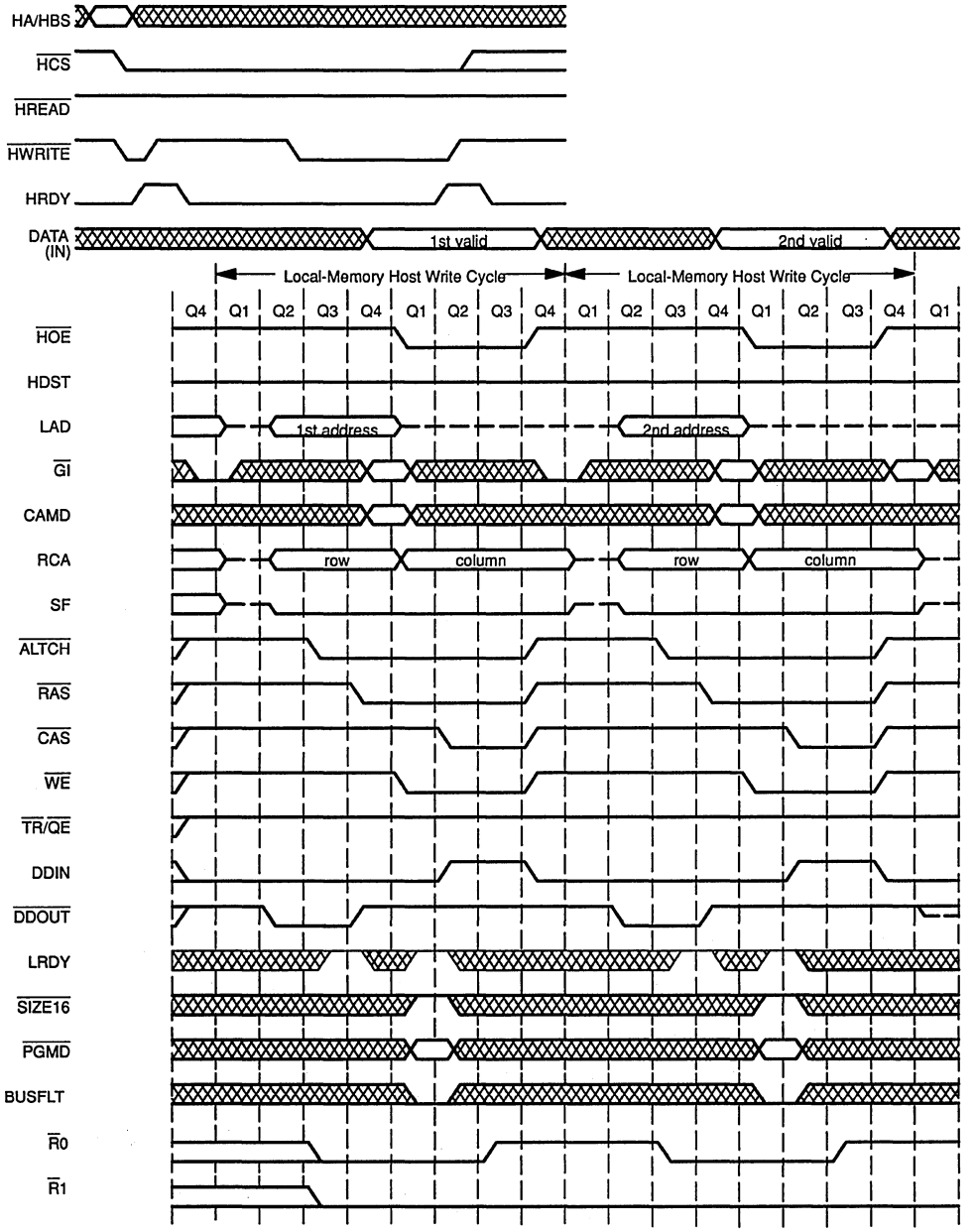
Figure 7–17. Back-to-Back Host Write Cycles; $\overline{HCS}$ Used as Strobe



Notes: 1) Prefetch after write is disabled (HPFW = 0 or HINC = 0).
2) This figure refers to host writes to local memory and does not include host writes to I/O registers.

Figure 7–17 shows 2 consecutive writes. The trailing edge of the host request strobe initializes the local-memory cycle. On the second write, notice that HRDY remains low until just before the data in the transceivers (from the first write) is written to memory.

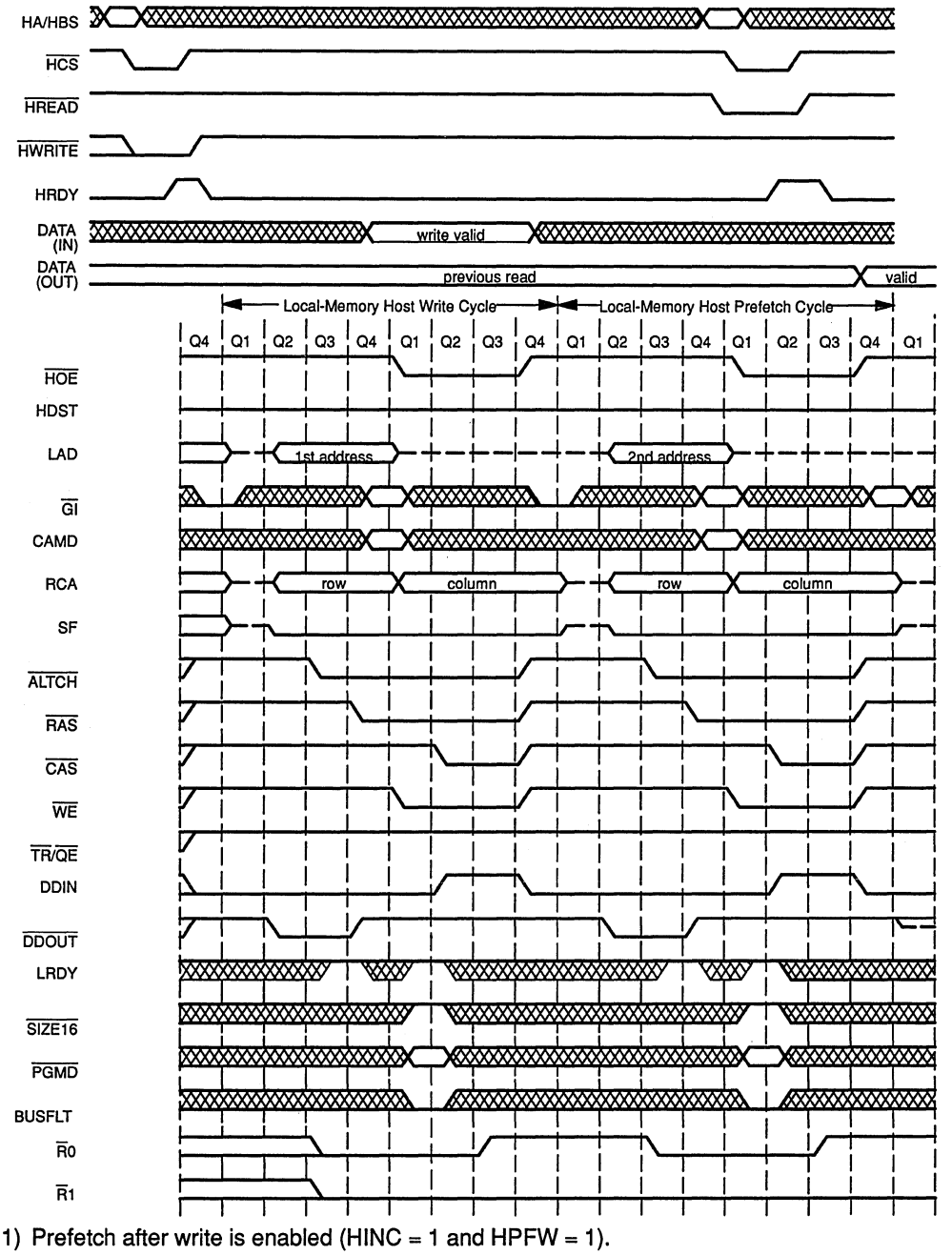
Figure 7–18. Back-to-Back Host Write Cycles with Implicit Addressing; $\overline{HWRITE}$ as Strobe



Notes: 1) Prefetch after read is enabled (HINC = 1 and HPFW = 0).
2) This figure refers to host writes to local memory and does not include host writes to I/O registers.

Figure 7–18 shows 2 consecutive writes. The trailing edge of the host request strobe initializes the local-memory cycle. Note that on the second write, HRDY remains low until just before the data in the transceivers (from the first write) is written to memory. Figure 7–18 also illustrates implicit addressing. Although not explicitly shown, the value on HBS0—HBS3 must be 1111_2, 0011_2, or 1100_2. The initial address is the one supplied at the falling edge of $\overline{HCS}$.

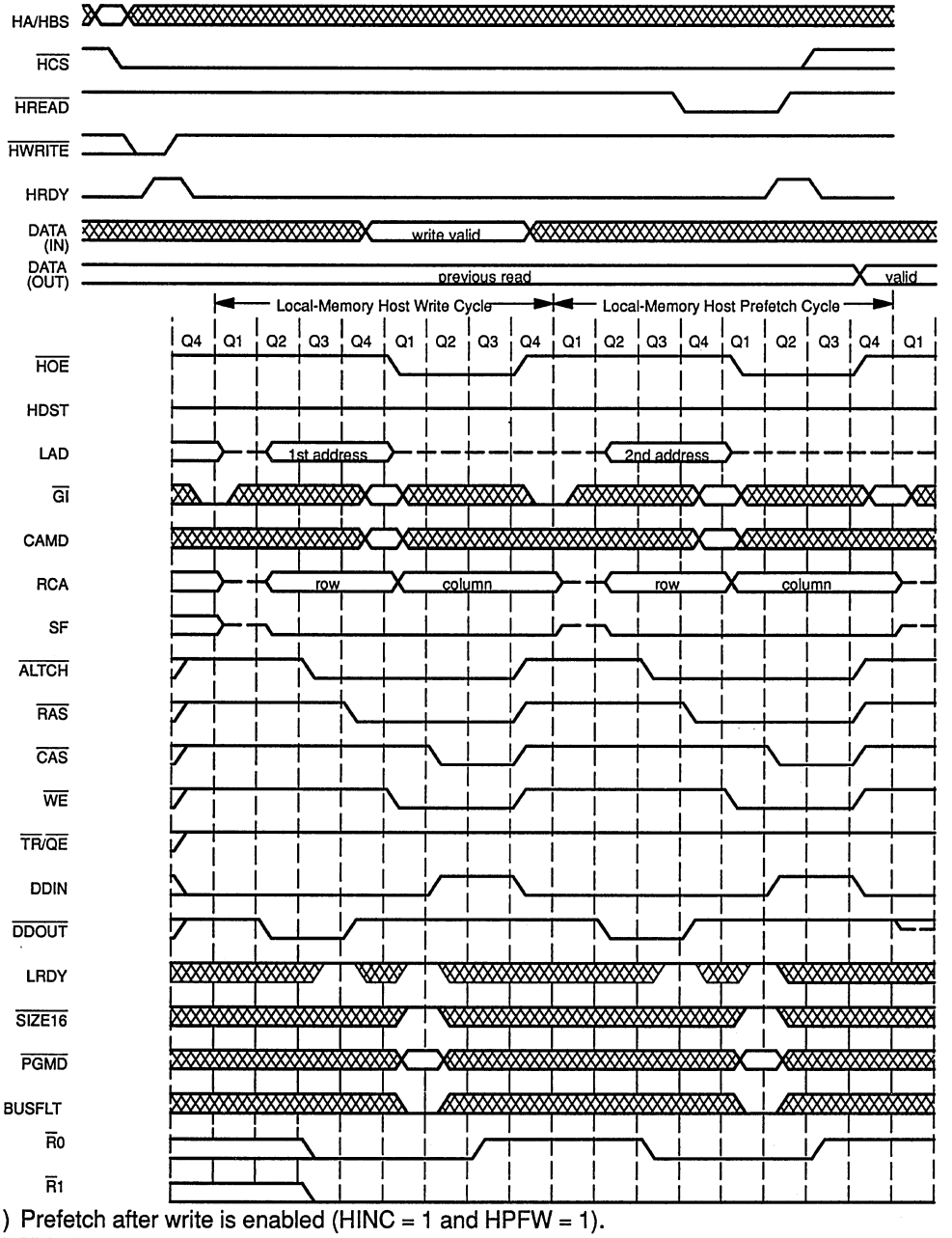
Figure 7–19. Host Write Cycle Back-to-Back with Prefetch of Next Word; $\overline{HCS}$ Used as Strobe



Notes: 1) Prefetch after write is enabled (HINC = 1 and HPFW = 1).
2) This figure refers to host writes to local memory and does not include host writes to I/O registers.

Figure 7–19 shows a write cycle followed by a prefetch of the next location. The TMS34020 automatically initiates the prefetch after it completes the write. Notice that HRDY remains low until just before the prefetched data is latched into the transceivers.

Figure 7–20. Host Write Cycle Back-to-Back with Prefetch of Next Word and Implicit Addressing; $\overline{\text{HREAD}}$ and $\overline{\text{HWRITE}}$ Used as Strobes



Notes: 1) Prefetch after write is enabled (HINC = 1 and HPFW = 1).
2) This figure refers to host writes to local memory and does not include host writes to I/O registers.

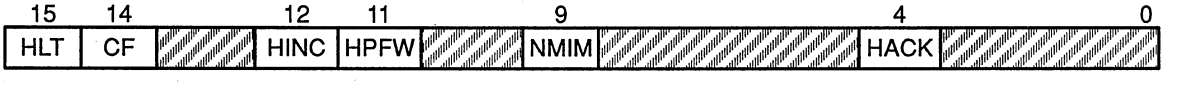
Figure 7–20 shows a write cycle followed by a prefetch of the next location. Notice that HRDY remains low until just before the prefetched data is latched into the transceivers. Figure 7–20 also illustrates implicit addressing. Although not explicitly shown, the value on HBS0—HBS3 must be 1111_2, 0011_2, or 1100_2.

7.8 Halting TMS34020 Execution and Downloading New Code

A host processor can halt TMS34020 execution by setting HLT[[HSTCTLH]]. While halted, the TMS34020 cannot access its local memory. This removes the major source of delay to host-initiated memory cycles and reduces (by half) the time taken for the host to access local memory.

Halting the TMS34020 involves several bits in the HSTCTLH register. The picture below identifies the pertinent bits:

HSTCTLH register **address: C000 0100h**



Note: For a complete illustration and description of HSTCTLH, see Chapter 4.

The TMS34020 may not immediately recognize the halt (reasons for this are described in Section 7.10.2, Halt Latency, on page 7-39). If the host is transferring large amounts of data, however, the increased throughput gained by halting the TMS34020 more than compensates for any delay in waiting for the TMS34020 to recognize the halt.

In order for a host processor to determine when the TMS34020 actually halts, it can

- Wait for a period equal to the TMS34020's maximum halt latency, **or**
- Repeatedly read the value of the HACK bit until HACK=1. When the TMS34020 halts, it sets HACK. When the TMS34020 is released from halt, it automatically clears HACK.

The TMS34020 recognizes a halt request on an instruction boundary only. Unlike an interrupt, a halt is not recognized during an interruptible instruction such as a PIXBLT. A PIXBLT of a very large area of memory could take a significant time to complete. If the host needs a prompter response to the halt request—that is, if it cannot wait for the current instruction to complete—the host should set the NMI bit when it sets HLT. The NMI (nonmaskable interrupt) is taken at the first interruptible point, which may be midway through an interruptible instruction. As soon as the NMI is taken, the halt is recognized (when both NMI and HLT are set, the TMS34020 halts before executing the first instruction in the NMI service routine).

One of the most useful applications for halting the TMS34020 is to provide a host processor with an efficient method for downloading new code to TMS34020 local memory. The host can download code by following these steps:

Step 1: Set the NMI, HLT, and CF bits to 1. This interrupts and halts the TMS34020 and flushes the cache.

Step 2: Download the code through the host interface. You can use the auto-incrementing (implicit addressing) feature by setting HINC to 1, clearing HPFW to 0, and providing the starting address of the code's destination.

Step 3: Write the starting address of the new code to the NMI vector so that the new code begins execution at the vectored address. (The NMI vector address is FFFF FEE0h.)

Step 4: Set the NMI bit to 1 to initiate a nonmaskable interrupt (the bit was cleared automatically if the previous nonmaskable interrupt was taken before the halt). At the same time, set the NMIM bit to 1 if the host does not need the current context to be stored on the stack or if the nonmaskable interrupt was taken in the first step; otherwise, clear NMIM to 0.

Step 5: Restart the TMS34020 by writing a 0 to HLT; at the same time, clear the CF bit.

7.9 Host-Interface Data Throughput (Bandwidth)

The host-interface **bandwidth** is the number of bits per second that can be transferred through the host interface during a block data transfer to or from TMS34020 local memory.

7.9.1 Achieving Maximum Bandwidth

The maximum host-interface bandwidth approaches the bandwidth of the TMS34020's local-memory interface. For a 40MHz TMS34020 with no memory wait states, and allowing for the fact that successive host-initiated local-memory cycles cannot be performed using page mode, the local-memory interface can perform one memory cycle every 200 nanoseconds. This is a bandwidth of 160 megabits per second.

Memory requests from other sources (such as screen refreshes, DRAM refreshes, or the TMS34020's CPU) tend to reduce the rate of data transfer. However, refreshes of either type occur infrequently (and therefore have little effect), and halting the TMS34020 prevents the CPU from making memory requests.

If the CPU is not halted, it could reduce the bandwidth through the host interface by up to 50%. If both host and CPU memory requests are pending, the host request is performed first, but the CPU request is performed before the next host request. Thus, if both the host and the CPU continue to make memory requests, host and CPU memory cycles occur alternately. Table 7–1 summarizes the different host interface bandwidths that can be expected.

Table 7–1. Host Interface Estimated Maximum Bandwidth

Assumptions	Approximate Throughput
❑ TMS34020 is halted ❑ 40 MHz TMS34020 ❑ No wait states	$\frac{\text{32 bits per transfer}}{\text{200 ns per transfer}}$ = 160 Mbits per second
❑ TMS34020 is running ❑ 40 MHz TMS34020 ❑ No wait states	$\frac{\text{32 bits per transfer}}{\text{400 ns per transfer}}$ = 80 Mbits per second
❑ TMS34020 is halted ❑ 40 MHz TMS34020 ❑ *N* wait states	$\frac{\text{32 bits per transfer}}{\text{(200+}N\text{) 100 ns per transfer}}$ Mbits per second

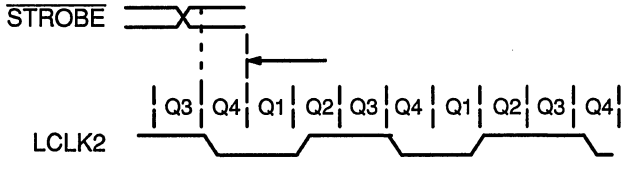
7.9.2 Timing Considerations for Optimizing Host-Interface Bandwidth

To use the local memory's full bandwidth, the host must be able to initiate a memory cycle every 200 nanoseconds. However, because there is an inherent synchronization and recognition delay between when the host request strobe occurs and when the local-memory cycle it initiates can begin, the request strobe that initiates the next memory cycle must occur before the current memory cycle has completed.

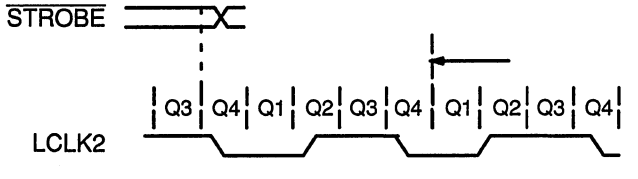
To enable the host to do this, HRDY is asserted active-high *early* during host-initiated memory cycles, shortly after LCLK2's low-to-high transition. The host request strobe that initiates the next memory cycle must occur before LCLK2's high-to-low transition in the current machine state, so that the memory cycle can begin in the next machine state. If the host request strobe occurs after this, it is not recognized until LCLK2's next high-to-low transition, so the memory cycle it initiates cannot begin until the machine state after that. This is illustrated in Figure 7–21.

Figure 7–21. Host Request Synchronization

(a) Case 1: host request strobe occurs before Q4



(b) Case 2: host request strobe occurs after Q4



Note: This illustration uses the hypothetical signal $\overline{\text{STROBE}}$, which is the logical-OR of $\overline{\text{HCS}}$ with either $\overline{\text{HREAD}}$ or $\overline{\text{HWRITE}}$; it is low when $\overline{\text{HCS}}$ and either $\overline{\text{HREAD}}$ or $\overline{\text{HWRITE}}$ are low. Host read cycles are initiated by $\overline{\text{STROBE}}$'s high-to-low transition (the leading edge of the host request). Write and prefetch cycles are initiated by $\overline{\text{STROBE}}$'s low-to-high transition (the trailing edge of the host request). The diagrams illustrate both possibilities. In both cases, the arrow (←——) identifies the earliest point at which the host-initiated local-memory cycle can begin. Obviously, this may be delayed if another local-memory cycle is being performed at this time.

For write requests or prefetch-initiating read requests, a low-to-high transition of the host request strobe is all that is necessary to initiate the next local-memory cycle. However, for host read requests when prefetching is disabled, the strobe's high-to-low transition enables the next local-memory

cycle for the host. Because of this, it is not possible to achieve maximum bandwidth when performing multiple host read requests where prefetches are disabled. There is not enough time, after HRDY goes active shortly following LCLK2's low-to-high transition, to deassert the host request strobes and then reassert them before LCLK2's high-to-low transition. In most applications, this should not matter because consecutive host accesses are typically to contiguous blocks where prefetching can be used.

Although HRDY is asserted just after LCLK2's low-to-high transition, the host cannot access the transceivers until both HDST and $\overline{\text{HOE}}$ are high. One of these signals is always low when HRDY goes high during a host-initiated cycle. This means that the host must gate HRDY with HDST and $\overline{\text{HOE}}$ to determine when it can access the transceivers.

The manner in which HRDY is gated with HDST and $\overline{\text{HOE}}$ depends primarily on the rate at which the host can consecutively access the TMS34020:

- **Host can make requests fast enough to use the maximum available throughput.** As described above, the host request strobe that initiates the next memory cycle must occur before the end of the current memory cycle, *before* the host can access the transceivers. This requires additional data buffering (or pipelining) into and out of the transceivers. HRDY will have returned to its inactive-low state before HDST or $\overline{\text{HOE}}$ goes high at the end of the current memory cycle. Because of this, the rising edge of HRDY must be used to set an edge-triggered latch, so that the output of the latch becomes a 1. This must be ANDed with HDST and $\overline{\text{HOE}}$, so that when HDST is high and the HRDY latch is set, the transceivers can be read from; when $\overline{\text{HOE}}$ is high and the HRDY latch is set, the transceivers can be written to. A mechanism for clearing the latch must be provided, although this depends on the precise requirements of the application. However, the latch must be cleared before the next low-to-high transition of HRDY, which can occur while either HDST or $\overline{\text{HOE}}$ is low. One possibility would be to use an edge-cleared latch cleared on the falling edge or either HDST or $\overline{\text{HOE}}$.

- **Host cannot make requests fast enough to use the maximum available throughput.** There is no need for the host request strobe that initiates the next local-memory access to occur until after the current host-initiated memory cycle completes. This can be achieved by ANDing HRDY with HDST and $\overline{\text{HOE}}$, so that the host does not respond to HRDY until HRDY, HDST, and $\overline{\text{HOE}}$ are all high.

7.10 Delays to Host Accesses

The longest (worst-case) delay that can occur on a host access is the accumulation of a number of smaller delays from a variety of sources. Worst-case delay depends on the application; all of these delays do not exist for all systems. It is unlikely that all possible delay sources exist for a single application.

- In some applications, you must determine not only the effective bandwidth of the host interface but also the **data transfer delays** that can occur under worst-case conditions. These conditions rarely occur, but the fact that they can occur makes them worthy of consideration. These types of delays affect the time it takes for the host to access local memory; see Section 7.10.1.
- It is sometimes advantageous for a host processor to halt TMS34020 execution by setting the HLT bit. However, there can be a delay from the time that a host write request sets HLT until the time that the TMS34020 actually halts. This delay is referred to as **halt latency**; see Section 7.10.2.

7.10.1 Worst-Case Delay

Worst-case delays are unusual; they occur too infrequently to significantly affect overall performance. The following paragraphs list all possible sources of delay. Some of these sources may produce different delays, depending on the application. Each delay is characterized by two parameters:

n is the **number of wait states** per TMS34020 memory cycle.

t is the **local clock period** (nominally 100 nS for a 40-MHz TMS34020).

Table 7–2 summarizes the various delay sources.

Table 7–2. Sources of Delay

	Delay (in local clock periods, t)	
Operation	**Minimum**	**Maximum**
Request synchronization	0.25	2.25
Screen refresh	0	
with midline reload		$2(2 + n)$
without midline reload		$2 + n$
DRAM refresh	0	
15 pending		$4 (3 + n)$
12 or fewer pending		$3 + n$
Bus-master arbitration	0	system dependent
Previous host cycle	0	
with prefetch after write		$2 (2 + n)$
without prefetch after write		$2 + n$
CPU cycle	0	$2 + n$

To calculate your application's worst case, determine which delays apply, and sum these individual delay times. This figure is the worst-case delay for a host access. It does not include the time required to actually perform the host access. The following paragraphs describe the delay sources.

- **Delay 1: Host request synchronization.** This is the time required to internally synchronize a host request to the TMS34020's local clock. Typically, host signals are not synchronous to the TMS34020's local clocks. Before using a signal from the host, the TMS34020 must pass it through a synchronizing latch (part of the TMS34020's on-chip host-interface logic). The delay through the synchronizer is 1.25 to 2.25 local clock cycles. However, the host-default cycle performed by the local-memory interface when no other requests are pending allows this synchronization to occur in parallel with the address subcycle. Thus, synchronization delay can be as small as 0.25 local clock cycles.

- **Delay 2: Screen-refresh cycles.** Any screen refresh has a higher priority than a host request. Thus, if a screen refresh and a host access are requested simultaneously, the TMS34020 performs the screen refresh. The longest delay that can be caused by screen refreshes occurs during horizontal blanking in split-serial register VRAM mode; this delay takes 4 machine states plus wait states. If this mode is not used, the longest delay is 2 machine states plus wait states (see Chapter 9, *Video Timing and Screen Refresh*).

- **Delay 3: DRAM-refresh cycles.** The TMS34020 can have up to 15 DRAM refreshes pending. If 12 or more DRAM refreshes are pending, they have higher priority than host requests. In this situation, if a DRAM refresh and a host access are requested simultaneously, the TMS34020 performs the DRAM refresh first. Thus, the TMS34020 may perform up to 4 DRAM refresh cycles before servicing a host request. In most systems, however, no more than 12 refreshes will ever be pending. The only way more than 12 DRAM refreshes could be pending is if a memory cycle is wait-stated for an extremely long time, or if the $\overline{GI}$ pin is deasserted for long enough. If 12 or fewer DRAM refreshes are pending, a host access can be delayed by one DRAM refresh only.

- **Delay 4: Bus-master arbitration.** This occurs only in systems where multiple TMS34020s and/or other devices share local memory and arbitrate for bus control through the multiprocessor interface (see Chapter 11). Typically, the host is connected to the only TMS34020 in the system that can make high-priority bus-master requests. Host requests generate the high-priority request code on $\overline{R0}$ and $\overline{R1}$. The time taken to reclaim bus ownership after a high-priority request code is output is entirely application dependent.

- **Delay 5: Previous host cycle.** The worst form of this delay occurs when the previous host request was a write with prefetch, and the host then

attempts to read from or write to a location other than the prefetched location. In this case, the TMS34020 performs the write and the prefetch before performing the current request. This is a delay of 4 machine states plus wait states. If prefetch-after-write is not enabled, the worst-case delay is 2 machine states plus wait states.

- **Delay 6: CPU cycle.** The TMS34020 CPU might be using the local-memory interface when the host requests a write. This causes a delay before the memory controller can perform the requested write. Note that a CPU cycle cannot delay a host access if
  - The TMS34020 encounters a bus-master arbitration delay.
  - The TMS34020 is halted.

In any system, worst-case delays occur too infrequently to affect overall performance. For example, DRAM-refresh cycles typically consume approximately 2% of the available memory bandwidth, and screen-refresh cycles (using VRAMs) consume approximately 1.5%. Do not use the worst-case delay to determine host-interface throughput.

The worst-case delay assumes that

- A screen-refresh cycle is generated within the TMS34020 on the same clock edge at which the host request arrives (after synchronization).
- 12 or more DRAM refreshes are requested during the next 1 + n clock edges. This is the number pending when the host and screen-refresh requests were made, plus additional requests (if any) made in the ensuing 1 + n machine states while the screen refresh is executed.
- An equivalent delay occurs when 12 or more DRAM-refresh requests and the host request occur on the same clock edge and a screen refresh is requested on a later clock edge before the host access begins. Host, DRAM-refresh, and screen-refresh requests all assert the high-priority request code through the multiprocessor interface (see Chapter 11).

7.10.2 Halt Latency

Section 7.8 (page 7-32) describes a method for halting the TMS34020 by setting the HLT bit. If you use this halt function, you may find it useful to know how long it can take the TMS34020 to enter the halt condition. This time, called **halt latency,** *always* includes

1 or 2 machine states	caused by the host-request synchronization (see delay 1 on page 7-38)
+ 2 machine states	for the host to actually write to the HLT bit
+ 1 machine state	for the CPU to halt after recognizing the request
4 or 5 machine states	is the minimum amount of time required to halt the TMS34020

In addition to the delays listed in Section 7.10.1, the TMS34020 may be executing an instruction when it receives the halt request. In this case, the TMS34020 does not halt until it completes the instruction execution. This could take several machine states (depending on the instruction), and applies to noninterruptible instructions as well. The TMS34020 recognizes a halt request only on an instruction boundary.

Section 7.10.1 shows the worst-case delay for accessing local memory (including I/O registers). Delays 2 through 6 can affect the time taken to halt the TMS34020, but their effect on halt latency is usually much less than the worst-case delay. In fact, they usually do not affect halt latency at all; even a single DRAM refresh is unlikely to delay halt recognition.

7.11 Systems with Multiple TMS34020s

Multiple TMS34020s running within a single system can share the same local memory. This can improve system performance significantly; however, it may complicate host-interface communications. For a general description of multiple TMS34020 systems, see Chapter 11, *Multiprocessing and System Architecture.*

If a multi-TMS34020 system contains ROM, the TMS34020s will probably be initialized from the ROM at power-up. However, if the system contains no ROM, the host processor must be able to download code to all of the TMS34020s. To do this, a host typically halts the TMS34020s; as an alternative, the host can prevent the processors from fetching code by setting their $\overline{GI}$ pins inactive, which prevents them from using the local-memory bus.

If all the TMS34020s are halted at power-up, the host must be able to restart them. To restart a halted TMS34020, the host must write a 0 to that TMS34020's HLT bit. The host can access the I/O registers of only those TMS34020s to which it is directly attached; one TMS34020 cannot access another's I/O registers. This means that the only method for clearing the HLT bit of a TMS34020 to which the host is not connected is to assert its $\overline{RESET}$ pin. *Only those TMS34020s to which the host has direct access can be powered-up halted.*

The host can access multiple TMS34020s by having a different $\overline{HCS}$ signal for each one. Because host requests are considered high priority, your multiprocessor arbitration scheme must prioritize simultaneous high-priority bus requests from different TMS34020s. To accomplish this, you can use one of two possible transceiver configurations:

- **All TMS34020s share one set of transceivers.** Wire together all the HDST pins, and wire together all the $\overline{HOE}$ pins. All data is transferred through the one set of transceivers to whichever TMS34020 is being accessed. This solution requires less hardware to implement. However, be very cautious when using prefetch modes; the location prefetched by one TMS34020 will no longer be stored in the transceivers if another

TMS34020 performed a host read. This could result in the host reading incorrect data. There are a number of solutions to this problem; here are two:

- Do not use the prefetch modes at all, or
- Ensure that the each time the host requests an access from a different TMS34020, the first read request performed is to a dummy location (this ensures that any previously fetched data is flushed out of the transceivers).

❑ **Each TMS34020 uses its own set of transceivers.** Connect the HDST and $\overline{\text{HOE}}$ pins from each TMS34020 to their respective sets of transceivers. Each set of transceivers transfers data between the host and one TMS34020. Although this implementation requires more hardware, it does allow you to use the prefetch modes. It also allows each TMS34020 to have a different host, if you desire.

When a TMS34020 loses bus mastership, the HDST and $\overline{\text{HOE}}$ pins are set to high impedance just like the local-memory control signals. However, unlike the local-memory signals, HDST and $\overline{\text{HOE}}$ both have internal pull-up resistors. This allows either of the above configurations to be implemented without requiring any additional external control circuitry. If HDST and $\overline{\text{HOE}}$ are not driven by another TMS34020 (the second case described above), the resistors hold them at the logic-high level. However, the resistors are sufficiently large that they do not prevent another TMS34020 from subsequently driving HDST and $\overline{\text{HOE}}$ (the first case described above).

If you want to connect the host to only one TMS34020, but want the host to download code to the other TMS34020s (to which it is not directly attached), you can use indirect-access methods. These other methods take advantage of the fact that even if not halted, a TMS34020 cannot access memory while its $\overline{\text{GI}}$ pin is held inactive-high. By explicitly controlling the multiprocessor-arbitration logic, the host can prevent TMS34020s to which it is not attached from fetching and executing new code until after the host downloads code through the TMS34020 to which it is attached.

If you adopt this procedure, a device can inform the TMS34020 of where to execute code from by asserting an external interrupt (via $\overline{\text{LINT1}}$ or $\overline{\text{LINT2}}$) while $\overline{\text{GI}}$ is held in its inactive state, then writing the address of the first word of code into the appropriate interrupt vector before asserting $\overline{\text{GI}}$ active low. This may be most useful at power-up (when the TMS34020 fetches code from the address specified in the reset vector). At other times, the TMS34020 may not start executing the new code immediately, because it may not have been at an instruction boundary or an interruptible point within an interruptible instruction when $\overline{\text{GI}}$ was deasserted.

In addition, the host can access other TMS34020s by using interrupt routines and/or flags (semaphores) to access information at a known location in memory, accessible to the host through the TMS34020 to which it is attached. The TMS34020's SWAPF instruction is helpful for this type of operation.

7.12 Systems with 16-Bit Memory Devices

If some or all of the TMS34020's local memory is made up of 16-bit memory devices, the TMS34020's memory controller uses its dynamic bus-sizing capability to access this memory.

If the host wants to access 16-bit local memory, you must add transceivers to route the data to and from the correct halves of the TMS34020's local address/data bus (LAD0—LAD31).

If the host uses a 32-bit bus and at least part of the TMS34020's local memory is 16 bits wide, then you'll need six 8-bit transceivers:

- The 4 regular transceivers for transferring data between the host and the 32-bit local memory.
- 2 additional transceivers to multiplex data between whichever half of the local bus the 16-bit memory is attached to and the opposite half of the host's data bus.

For example, if the 16-bit memory is configured on LAD0—LAD15, then the extra transceivers are required to multiplex data between LAD0—LAD15 and bits 16—31 of the host's data bus (when accessing the second of the two 16-bit words contained in the 32-bit location specified by HA5—HA31).

If 16-bit memory is configured to both halves of the TMS34020's local-memory address/data bus (LAD0—LAD15 and LAD16—LAD31), then you need eight 8-bit transceivers because it is necessary to swap both halves of the local address/data and host data buses.

Even if the host is configured to a 16-bit bus and all of the TMS34020's local memory is 16 bits wide, four 8-bit transceivers are still required because the I/O registers are 32 bits wide. Only in the unlikely situation in which the host would never access any I/O registers could you use two 8-bit transceivers to connect the host's 16-bit data bus to the appropriate half of the TMS34020's local address/data bus.

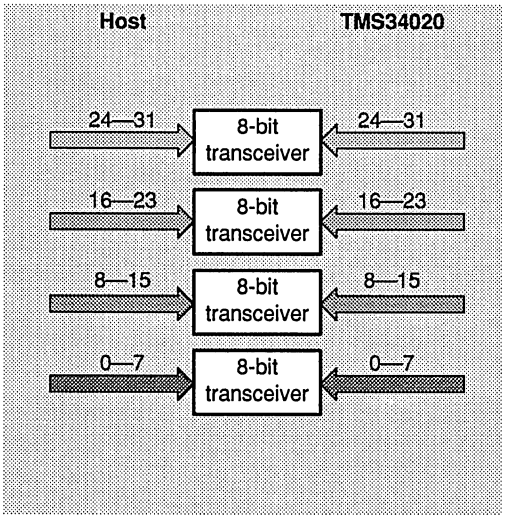
Data should be routed according to the $\overline{\text{SIZE16}}$ signal. For more details about dynamic bus sizing, refer to Section 8.6, Dynamic Bus Sizing, on page 8-12.

If the host accesses 16-bit-wide memory, the TMS34020's memory controller automatically performs both cycles necessary to access the full 32 bits of the long-word address present on HA5—HA31, regardless of which HBS byte strobes are active.

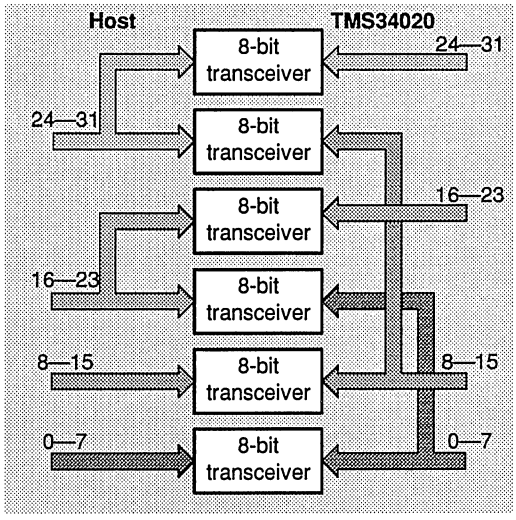
Figure 7–22 shows the transceiver combinations required for the various system configurations.

Figure 7–22. Host-to-TMS34020 Transceiver Wiring with 16-Bit Memory

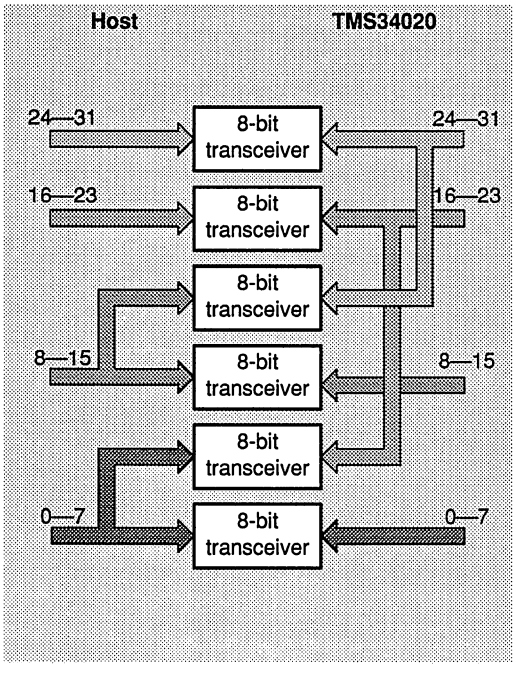
(a) Without 16-bit memory



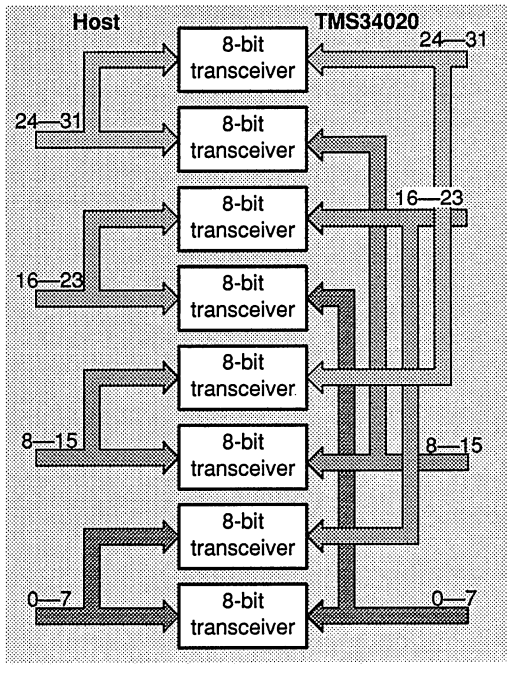
(b) With16-bit memory connected to LAD0—LAD15



(c) With 16-bit memory connected to LAD16—LAD31



(d) With 16-bit memory connected to both LAD0—LAD15 and LAD16—LAD31



7.13 Systems with Big-Endian Addressing

If the your system uses the TMS34020 in big-endian mode, then the bytes addressed by HBS0—HBS3 are swapped. See Chapter 3, *Memory Organization and Hardware-Supported Data Structures*, for more detail on the differences between big-endian and little-endian operation. Figure 7–23 shows the byte addressing for both.

Figure 7–23. Big-Endian and Little-Endian Byte Addressing Modes

(a) Little-endian byte addressing

	Byte 3	Byte 2	Byte 1	Byte 0
Bits:	31—24	23—16	15—8	7—0
Byte Strobes:	HBS3	HBS2	HBS1	HBS0

(b) Big-endian byte addressing

	Byte 3	Byte 2	Byte 1	Byte 0
Bits:	7—0	15—8	23—16	31—24
Byte Strobes:	HBS3	HBS2	HBS1	HBS0

For example, in order to access bits 7—0 of a word in the TMS34020's local memory in big-endian mode, the host must assert HBS3 instead of HBS0.

Chapter 8

Local-Memory and DRAM/VRAM Interfaces

This chapter describes the local-memory interface and the DRAM/VRAM interface.

8.1 Related Signals

Many of the TMS34020's pins provide access to and control over the local-memory and DRAM/VRAM interfaces. Chapter 2 describes these signals in detail; they are summarized below for your convenience.

Signals	Descriptions	I/O
$\overline{\text{ALTCH}}$	is the address latch signal. An external latch (such as a 74ALS373) can use a high-to-low transition on $\overline{\text{ALTCH}}$ to maintain the current address and status from LAD0—LAD31.	O
BUSFLT	is the bus-fault signal. If external logic detects an error or fault in the current cycle, the logic asserts BUSFLT high. BUSFLT is also used in conjunction with LRDY to generate bus-retry cycles.	I
CAMD	is the column-address mode input that allows mixing of DRAM address matrices using the same multiplexed RCA0—RCA12 address signals.	I
$\overline{\text{CAS0}}$—$\overline{\text{CAS3}}$	are the column-address strobe signals that drive the $\overline{\text{CAS}}$ inputs of DRAMs and VRAMs. They also provide access to the individual bytes in each long-word in memory.	O
CLKIN	is the clock input signal. The TMS34020's processor functions are synchronous to CLKIN.	I
DDIN	is the local data-bus-direction input-enable signal, driven high to enable data transceivers (such as a 74ALS623) on LAD0—LAD31 to input data to the TMS34020.	O
$\overline{\text{DDOUT}}$	is the local data-bus-direction output-enable signal. It is driven low to enable data transceivers (such as a 74ALS623) on LAD0—LAD31 to output data from the TMS34020.	O
LAD0—LAD31	form the local address/data bus.	I/O
LCLK1, LCLK2	are the local output clocks. These signals are available to the system for synchronous control of external logic.	O
$\overline{\text{LINT1}}$, $\overline{\text{LINT2}}$	are the TMS34020's local interrupt-request signals. External devices can use $\overline{\text{LINT1}}$ and $\overline{\text{LINT2}}$ to interrupt TMS34020 operation.	I
LRDY	is the local ready signal. External circuitry drives LRDY low to inhibit the TMS34020 from completing a local-memory cycle.	I

Signals	Descriptions	I/O
$\overline{\text{PGMD}}$	is the page-mode signal. External memory decode logic asserts $\overline{\text{PGMD}}$ low if the currently addressed memory supports burst (page mode) accesses.	I
$\overline{\text{RAS}}$	is the row-address strobe signal that drives the $\overline{\text{RAS}}$ inputs of DRAMs and VRAMs.	O
RCA0—RCA12	form the multiplexed row/column address bus, used for the row and column addresses for DRAMs and VRAMs.	O
SF	is the special-function signal that connects to the SF pin of 1M VRAMs.	O
$\overline{\text{SIZE16}}$	is the bus-size signal. External memory decode logic asserts $\overline{\text{SIZE16}}$ low if the currently addressed memory supports 16-bit accesses only. $\overline{\text{SIZE16}}$ is also used to determine to which half of the LAD bus the accesses are made.	I
$\overline{\text{TR/QE}}$	is the transfer/output-enable signal that connects to the $\overline{\text{TR/QE}}$ pins of VRAMs.	O
$\overline{\text{WE}}$	is the write-enable signal that drives the $\overline{\text{WE}}$ inputs of DRAMs and VRAMs.	O

8.2 Related Registers

The local-memory interface registers summarized below are a subset of the I/O registers discussed in Chapter 4.

Four registers are associated with the local-memory and DRAM/VRAM interfaces:

- **CONFIG** contains several bits that control individual aspects of the local-memory interface.
- **DPYCTL** controls aspects of the DRAM/VRAM interface.
- The **PMASK** and **REFADR** registers contain 16-bit values that are necessary for proper operation of the local-memory or DRAM/VRAM interface.

CONFIG register **address: C000 01A0h**

Note: For a complete illustration and description of CONFIG, see Chapter 4.

BEN bit 0 — Setting the BEN (big-endian enable) bit to 1 tells the TMS34020 to access memory in big-endian mode. Clearing BEN to 0 tells the TMS34020 to access memory in little-endian mode. Little-endian is the default.

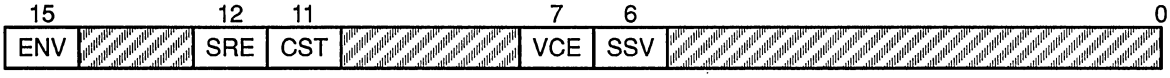
RCM bits 1—2 — The RCM (RCA bus mode configuration) bits determine the basic row- and column-address mode for the RCA bus.

CBP bit 3 — Setting the CBP (configuration byte protect) to 1 write-protects bits 0—7 of the CONFIG register. This prevents modification of the selected configuration.

VEN bit 8 — Setting the VEN (VRAM enable) bit to 1 informs the TMS34020 that the system contains VRAMs with special features such as write-mask and color-latch registers.

RR bits10—12 — The RR (DRAM refresh rate) bits control the frequency of DRAM refresh cycles.

DPYCTL register **address: C000 0080h**



Note: For a complete illustration and description of DPYCTL, see Chapter 4.

SSV bit 6 — Setting the SSV (split-serial-register midline-reload enable) bit to 1 enables the TMS34020 to perform screen refreshes for VRAMs with split serial registers.

VCE bit 7 — The VCE (video capture enable) bit selects the type of screen-refresh memory cycle:

VCE=0 selects memory-to-register screen-refresh cycles.
VCE=1 selects register-to-memory screen-refresh cycles.

CST bit 11 — Setting the CST (CPU serial register transfer) bit to 1 tells the TMS34020 to convert ordinary pixel accesses into VRAM serial register transfers.

SRE bit 12 — Setting the SRE (screen-refresh enable) bit to 1 enables automatic screen refreshes when video is enabled (ENV=1).

ENV bit 15 — The ENV (enable video) bit **must be 1** to allow the video timing logic to operate. When ENV=0, the TMS34020's blanking outputs are permanently at the active-low level, and all display control is disabled.

PMASKL register **address: C000 0160h**
PMASKH register **address: C000 0170h**
REFADR register **address: C000 01F0h**

PMASKL, PMASKH work together to form a 32-bit plane mask. The 0 and 1 values in the plane mask selectively enable or disable various planes in a multiple-bit-per-pixel bitmapped display.

REFADR is the refresh-address register. It contains a pseudo-address that is output on each refresh cycle.

8.3 Priorities of Memory Bus Requests

The TMS34020's local-memory controller assigns priorities to local-memory requests from various on- and off-chip sources. The memory controller can generate VRAM serial-data-register transfer cycles, refresh cycles, host accesses to memory, CPU cycles, and emulator accesses, and it can release the local bus for other processors. Table 8–1 shows the order in which the TMS34020 responds to these access requests; priority 1 is the highest priority (the TMS34020 responds to it first).

Table 8–1. Priorities for Memory Cycle Requests

Priority	Memory Cycle Request
1	Loss of bus grant, indicated by $\overline{GI}$ (grant input) high
2	Any video-generated VRAM serial-data-register transfer cycle
3	12 or more DRAM-refresh cycles are pending
4	Emulator (if emulator priority requested)
5	Host access
6	4 or more DRAM-refresh cycles are pending
7	TMS34020 CPU access (single cycle, page mode, or CPU-initiated serial-data-register transfer cycles)
8	Less than 4 DRAM-refresh cycles are due
9	Host-default cycle

Note: If enabled, the TMS34020 waits for a host request when no other cycle is requested.

Here are more detailed descriptions of these memory cycle requests.

1) The highest priority is assigned to the loss of bus grant, which takes priority over all other types of memory requests. However, it is recommended that the bus grant is not removed when a high-priority memory cycle code is output on the $\overline{R0}$ and $\overline{R1}$ signals. Requests of priorities 2—5 generate this code.

2) Any video-generated, VRAM serial-data-register transfer requests have the second highest priority. This is the highest priority of any internally generated request.

3) DRAM-refresh requests are programmed to occur at regular intervals. The TMS34020 has an internal counter that tracks how many refreshes are due. The TMS34020 increments this counter by 1 when the refresh counter requests a refresh; the TMS34020 decrements the counter by 1 when a refresh takes place. When 12 refreshes are due, refreshing becomes a higher priority to ensure that a refresh cycle occurs.

4) An in-circuit emulator (such as the Texas Instruments TMS34020 Emulator) can raise the priority of the emulator's and the CPU's local-memory requests above the priority of the host's local-memory requests. If you do not do this, the TMS34020 treats emulator requests with the same priority as regular CPU requests.

5) The host interface has the next highest priority. If a CPU-initiated cycle is in progress, it is interrupted on the next 32-bit boundary. (If the CPU is inserting a field, then both the read and write cycles complete. Any operation required as a result of dynamic bus sizing also completes.)

6) If the DRAM-refresh counter indicates that 4 or more cycles are pending and no page-mode accesses are in progress, then refresh cycles assume a higher priority than CPU-initiated requests. This prevents the refresh counter from progressing too far without interrupting page-mode blocks.

7) CPU requests are serviced if no higher priority requests are present. Once a CPU cycle begins, request with priorities 1 through 5 can interrupt a series of CPU-initiated page-mode accesses. The interrupted sequence will restart as with any ordinary access.

8) If the refresh counter is not 0 and nothing else is happening, then DRAM refreshes occur.

9) If no definite request is pending, then the TMS34020 performs a host-default cycle. This is a special type of idle cycle. (For more information, see Section 8.15 on page 8-49.)

Even if host requests are being made at the full bandwidth rate, there will always be one cycle after a host access when the host interface request to the memory controller is not active. If no other accesses are pending, the host-default state will enable host accesses to use the full bus bandwidth. However, if another type of access is pending, this other access will be executed in preference to the host default.

Thus, if both the host interface and the CPU are requesting access at the maximum rate possible, the memory controller services the requests alternately. This prevents the host from completely dominating the local-memory interface; the host cannot prevent servicing of other requests.

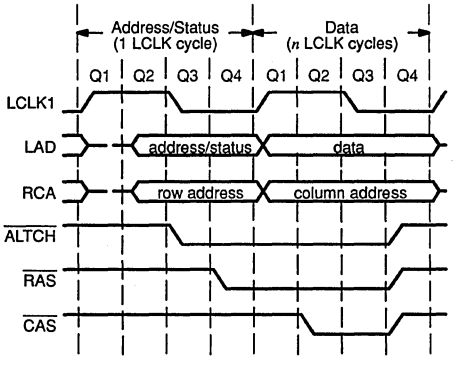
8.4 General Form of a Local-Memory Cycle

Most local-memory cycles initiated by the TMS34020's memory controller are at least 2 machine states in duration. (An exception to this is page-mode cycles, discussed in Section 8.7, page 8-15.) Each machine state is one LCLK period long and begins at the rising edge of LCLK1. As Figure 8–1 shows, each local-memory cycle has two parts:

- the **address/status subcycle** and
- the **data subcycle**.

Figure 8–1 also shows the signals required to latch the addresses that are on the LAD and RCA buses. Page-mode cycles are an extension of this form.

Figure 8–1. The Two Parts of a Local-Memory Cycle



Notes: 1) The data subcycle is an integral number of LCLK cycles long. The diagram shows the minimum length of 1 LCLK cycle.

2) This illustration does not apply to page-mode memory cycles.

3) For complete timing diagrams, refer to Sections 8.8—8.13 (pages 8-18—8-44).

8.4.1 The Address/Status Subcycle

The address/status subcycle coincides with the first machine state of the local-memory cycle. The address, status code, and row address for the access are output at this time. This portion of the cycle is often called the **row-address time** because the row address is output during this subcycle.

Section 8.5 (page 8-10) discusses the status codes output during this subcycle.

The LAD bus outputs an address that can be latched (using either $\overline{\text{ALTCH}}$ or $\overline{\text{RAS}}$) and used to decode the TMS34020's address space. This address points to the currently accessed long-word. The RCA bus outputs a subset of this address to serve as the *row address*. The RCA bus is usually connected

directly to DRAMs and VRAMs, which require the row and column parts of the address to be time-multiplexed over their address input pins. The address can also be latched using either $\overline{\text{ALTCH}}$ or $\overline{\text{RAS}}$. Section 8.16 (page 8-50) provides a detailed discussion of the addressing modes.

LAD0—LAD3 convey a status code that indicates the type of local-memory cycle being performed. This information can be latched, along with the address on LAD. Section 8.5 (page 8-10) contains a detailed description of the status codes.

8.4.2 The Data Subcycle

This subcycle is at least one machine state long and immediately follows the address/status subcycle. The column address for the cycle is output at this time, and data is transferred between the TMS34020 and the local memory (or vice versa). This portion of the cycle is often called the **column-address time**.

The LAD bus transfers data between the TMS34020 and local memory. Data is either driven out (for a write cycle) or latched in (for a read cycle). The $\overline{\text{RCA}}$ bus outputs a column address that can be latched using any of the $\overline{\text{CAS}}$ signals. The RCA bus is usually connected directly to DRAMs and VRAMs, which require the row and column parts of the address to be time-multiplexed over their input pins. Section 8.16 (page 8-50) discusses available addressing modes.

If a 1-LCLK data subcycle is not sufficient for a data transfer to complete, the data subcycle can be extended, as required, by inserting wait states. Section 8.6 (page 8-12) deals with this in more detail.

8.5 Local-Memory Cycle Status Codes

In addition to the address output over the LAD lines, the TMS34020 outputs a status code on LAD0—LAD3. This status code identifies the type of local-memory cycle being performed. The code can be latched by the falling edge of $\overline{\text{ALTCH}}$, along with the address output on the remaining LAD pins. Table 8–2 lists the status codes and their meanings.

Table 8–2. LAD-Bus Status Codes

LAD PIN					
3	2	1	0	Bus Status	Type
0	0	0	0	Coprocessor cycle	miscella-
0	0	0	1	Emulator operation	neous
0	0	1	0	Host cycle	(00xx)
0	0	1	1	DRAM refresh	
0	1	0	0	Video-generated VRAM serial-register transfer	VRAM
0	1	0	1	CPU-generated VRAM serial-register transfer	(01xx)
0	1	1	0	Write-mask load	
0	1	1	1	Color-register load	
1	0	0	0	Data access	CPU
1	0	0	1	Cache fill	(1xxx)
1	0	1	0	Instruction fetch	
1	0	1	1	Interrupt-vector fetch	
1	1	0	0	Bus-locked operation	
1	1	0	1	Pixel operation	
1	1	1	0	Block write	
1	1	1	1	Reserved	

Here are more detailed descriptions of these status codes.

- **Coprocessor cycle.** Output for any memory cycle that transfers instruction information or data between a coprocessor and either the TMS34020 or the local memory. These memory cycles are beyond the scope of this chapter but are discussed in detail in Chapter 10, *Communicating with a Coprocessor.*

- **Emulator operation.** Output during a special memory cycle that only an in-circuit emulator can initiate. This cycle is intended for loading external mapping registers implemented within the emulator using SRAM.

- **Host cycle.** Output on any memory cycle that is generated in response to the host interface.

- **DRAM refresh.** Output when the memory controller performs a DRAM-refresh cycle.

- **Video-generated VRAM serial-register transfer.** Output when the TMS34020 memory controller performs a midline-reload cycle, or a serial-register-transfer cycle initiated by horizontal blanking. (VCE[[DPYCTL]] controls the transfer direction.)

- **CPU-generated VRAM serial-register transfer.** Output when the TMS34020 memory controller performs a serial-register-transfer cycle due to a pixel operation instruction with CST[[DPYCTL]] = 1. A pixel-read operation generates a read transfer (VRAM memory to serial register). A pixel write generates a write transfer (serial register to VRAM memory).

- **Write-mask load.** Generated during the special 1-Mbit VRAM cycle to load the VRAM's on-chip write-mask register with the 1s complement of the TMS34020's PMASK registers.

- **Color-register load.** Generated during the special 1-Mbit VRAM cycle initiated by the VLCOL instruction, which loads the VRAM's on-chip color register.

- **Data access.** Output when the current memory access (read or write) involves transferring data to or from the TMS34020's CPU.

- **Cache fill.** Output when the TMS34020 is reading a subsegment (4 long-words) into the on-chip instruction cache.

- **Instruction fetch.** Output when the instruction cache is disabled (CD[[CONTROL]] = 1) and the TMS34020 is reading instructions or immediate data from memory.

- **Interrupt-vector fetch.** Output any time a TRAP, interrupt, or reset occurs. The code indicates that the TMS34020 is reading the appropriate vector address from memory.

- **Bus-locked operation.** Executed when the TMS34020 executes a SWAPF instruction (indicating a locked read-modify-write operation).

- **Pixel operation.** Output when the TMS34020 performs any pixel operation and CST[[DPYCTL]] = 0.

- **Block write.** Generated during one of the special 1-Mbit VRAM block-write cycles performed when executing the VBLT or VFILL instruction.

8.6 Ending a Local-Memory Cycle

A data subcycle begins in the machine state immediately following the address/status subcycle. The data subcycle lasts for a minimum of 1 machine state. During this subcycle, the TMS34020 samples the LRDY and BUSFLT pins to determine when and how the local-memory cycle will end. The TMS34020 also samples the $\overline{\text{PGMD}}$ and $\overline{\text{SIZE16}}$ pins at this time. (Section 8.7, page 8-15, discusses $\overline{\text{PGMD}}$; Section 8.9, page 8-25, discusses $\overline{\text{SIZE16}}$.)

Immediately following the address subcycle, the TMS34020 samples the LRDY and BUSFLT pins on LCLK2's low-to-high transition in *each* machine cycle of the data subcycle. The pins are *not* sampled in subsequent page-mode data subcycles (see Section 8.7). External hardware must decode the address output during the address/status subcycle and determine the appropriate levels for LRDY and BUSFLT. Table 8–3 lists the logical combinations of these signals.

Table 8–3. Bus Cycle Completion Conditions

Completion Condition	BUSFLT	LRDY
Insert a wait state	0	0
Successful transfer	0	1
Retry	1	0
Bus fault	1	1

8.6.1 Extending a Local-Memory Cycle with Wait States

BUSFLT low	LRDY low

Under these conditions, the TMS34020 inserts a wait state to extend the data subcycle. All local-memory control signals activated during the cycle remain active for an additional LCLK cycle. The TMS34020 samples LRDY and BUSFLT again on LCLK2's next rising edge. If the TMS34020 detects a wait condition again, it extends the data subcycle for another LCLK cycle, and so on.

After the first data subcycle completes successfully, subsequent accesses may occur as page-mode cycles (see Section 8.7 on page 8-15). During page-mode cycles, the TMS34020 does not sample LRDY and BUSFLT. This means that wait states can be inserted during the first data subcycle, but not during subsequent page-mode cycles. This can be useful, for example, when using external memory management hardware that must translate an address; although the memory management hardware may allow for high-speed page-mode accesses, it may not be able to quickly translate the address during the first access.

8.6.2 Completing a Successful Local-Memory Cycle

BUSFLT low	LRDY high

Under these conditions, all the local-memory control signals active during this cycle are driven inactive on LCLK2's next high-to-low transition, completing the cycle. A successful transfer should be indicated only if

- for **read cycles**, valid data will be available on the LAD bus at the next falling edge of LCLK2.
- for **write cycles**, the devices being written to will be able to latch the data on the LAD bus at the next falling edge of LCLK2.

If these criteria cannot be met, the TMS34020 must extend the data subcycle by inserting wait states.

8.6.3 Retrying a Local-Memory Cycle

BUSFLT high	LRDY low

These conditions terminate the local-memory cycle (in the same way as a successful transfer) and subsequently restart the cycle from the beginning. The original address and status code are output again during the address/status subcycle of the retried access.

When a retry occurs, the cycle may not restart immediately because the memory controller will first perform any higher priority requests that might be pending when the retry occurs.

Some memory operations require multiple local-memory cycles. (For example, modifying a field within a word requires a read and a write. If a field crosses a word boundary, 2 reads and writes are required. Also, accessing a 16-bit memory device requires twice the number of local-memory cycles.) Except for bus-locked local-memory cycles, a retry restarts only the memory cycle on which the retry occurred. If a retry occurs on the write portion of a read-modify-write sequence, it restarts from the write, not the read. Similarly, if a retry occurs on the second of the 2 accesses required to read or write a location implemented as 16-bit memory, the first of the 2 accesses is not performed again; the address output on the LAD and RCA buses during the restarted access will have the S bit set to 1.

The SWAPF instruction initiates bus-locked local-memory cycles used for exchanging semaphores. The read and write portions of a bus-locked operation must occur one immediately after the other. Because of this, if a retry occurs on the write part of a bus-locked operation, it is restarted from the read cycle.

If a retry occurs during a host-initiated local-memory cycle, the cycle restarts normally. However, the memory controller also sets HRYI[[HSTCTLL]], indicating that a retry occurred. If HBREN[[HSTCTLL]] is also set, the $\overline{\text{HINT}}$ pin is driven low to interrupt the host. Once set by the memory controller, HRYI remains set until the host clears the bit.

8.6.4 Bus Faulting a Local-Memory Cycle

BUSFLT high	LRDY high

These conditions inform the TMS34020 that the access was unsuccessful. The local-memory cycle is terminated (in the same way as a successful transfer); subsequent actions depend on the type of access that caused the bus fault.

- **If the CPU initiated the access**, the memory controller first saves its current state in the BSFLTD and BSFLTST registers, then signals a bus-fault interrupt to the CPU. (This applies to all memory cycles except those with host, DRAM refresh, or video-initiated VRAM serial-register transfer status codes.) The CPU saves on the stack all the data necessary to allow it to correctly restart execution after returning from the bus fault, then reads the bus-fault vector address from FFFFFBC0h.

  The bus-fault routine should clear the cause of the bus fault. Upon returning from the bus-fault routine, the CPU pops all the data it pushed before taking the interrupt. The memory controller restores the state of the faulted memory cycle from BSFLTD and BSFLTST and restarts the cycle (in the same way as for a retry). Section 6.9 (page 6-19) discusses the bus-fault interrupt in detail.

- **If the host initiated the access**, the memory controller terminates the local-memory cycle (in the same way as for a successful transfer) and sets HBFI[[HSTCTLL]] to indicate that a bus fault occurred. No other action is taken. If HBREN[[HSTCTLL]] is also set, the $\overline{\text{HINT}}$ pin is driven low (while HBFI is set), to interrupt the host. The host must then clear the source of the bus fault before attempting to restart the access. Once set by the memory controller, HBFI remains set until the host clears it.

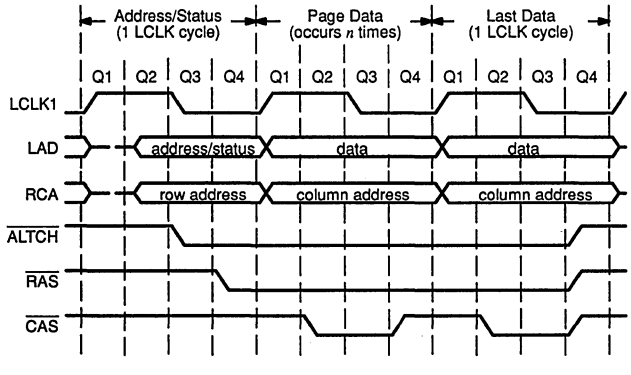
- **If the access was a DRAM-refresh or video-initiated screen-refresh cycle**, no action is taken. The local-memory cycle is terminated in the same way as a successful transfer.

8.7 Performing Local-Memory Cycles in Page Mode

Many DRAMs and VRAMs can operate in **page mode**. This mode can greatly reduce the time required to access a memory location when performing multiple memory cycles, provided that the row address for each access is the same. The TMS34020 directly supports page-mode accesses and uses page mode whenever possible.

Figure 8–2 shows the general form of page-mode accesses. It also shows the signals required to latch the addresses present on the LAD and RCA buses. The first access of any sequence always consists of an address/status subcycle and a data subcycle, as outlined in Section 8.4 (page 8-8). However, if multiple accesses are to be made using page mode, the $\overline{\text{ALTCH}}$ and $\overline{\text{RAS}}$ signals do not go inactive at the end of the first data subcycle. They remain active until the end of the last access in the sequence, thereby ensuring that the original row address stays latched in the DRAMs throughout the access. After the first word is accessed, all subsequent accesses are made using single machine-state page-mode cycles, during which the column address for the access is output and data is transferred.

Figure 8–2. Multiple Local-Memory Cycles Using Page Mode



Notes: 1) Except for the first data subcycle (which may be extended by inserting wait states), each page-mode data cycle is only 1-LCLK cycle long.

2) n is an integer in the range of 0—63.

3) For complete timing diagrams for specific local-memory cycles, refer to Sections 8.8—8.13 (pages 8-18—8-44).

8.7.1 Selecting Page-Mode Operation

Each time the TMS3020 performs a standard memory access, it has the opportunity to perform a page-mode access. The TMS34020 samples the $\overline{\text{PGMD}}$ pin to determine whether the DRAM/VRAM supports page mode. If subsequent memory accesses are required, these are (whenever possible) performed in page mode unless the local memory indicates that page-mode accesses are not supported at the current address.

External hardware must decode the address output during the address/status subcycle and determine the appropriate level for the $\overline{\text{PGMD}}$ pin, which the TMS34020 samples at the same time as LRDY and BUSFLT. Sampling a 0 on the $\overline{\text{PGMD}}$ pin indicates that page-mode accesses are supported. If wait states are inserted into the initial memory cycle, $\overline{\text{PGMD}}$ must be kept at a valid level on LCLK2's low-to-high transition during each machine state of the data subcycle.

The TMS34020 supports a page size of sixty-four 32-bit words. All memory locations for which logical address bits 10 through 31 are the same are considered to be within the same page. If, while performing a sequence of page-mode accesses, the TMS34020 wishes to access a location that is not on the same page as the previous access (that is, the value on bits 10 through 31 differ), the page-mode sequence ends. The access is made using an standard local-memory cycle, with an address/status subcycle and a data subcycle, so that a new row address can be output. LRDY, BUSFLT, and $\overline{\text{PGMD}}$ can be resampled.

Once $\overline{\text{PGMD}}$ is detected low and a page-mode sequence begins, the TMS34020 does not sample LRDY, BUSFLT, and $\overline{\text{PGMD}}$ again. This means that

- All 64 locations within the page must be capable of supporting page-mode operation.
- All locations within a page must support 0 wait-state accesses.
- No bus faults or retries can be generated within the page, once a page-mode sequence has started. If you are using bus faults to implement a virtual memory system, this means that the entire page must be mapped into the physical address space.

8.7.2 How the TMS34020 Uses Page Mode

The TMS34020 uses page mode any time it must make multiple accesses to contiguous addresses. Page-mode cycles can originate from two sources: the CPU and the memory controller. Assuming that the addressed memory supports page mode, a page-mode cycle occurs:

- Any time the CPU wishes to read or write multiple consecutive words in memory. This can occur during almost all graphics instructions (PIXBLT, FILL, VBLT, VFILL, etc.) and the multiple move instructions (MMTM, MMFM, and BLMOVE). It can also occur anytime a cache fill is performed, or if a context switch is made before servicing or when returning from an interrupt.
- Any time the memory controller must perform multiple memory accesses to complete a single memory operation. This occurs when the CPU wishes

to write a non-byte-aligned field within a word (in which case a read-modify-write must be performed), or if the CPU accesses a field that straddles a word boundary. It can also occur when dynamic bus sizing is used to access a 16-bit memory (because 2 memory accesses are required to completely read or write one word). This occurs on both host- and CPU-initiated accesses. Section 8.9 (page 8-25) discusses dynamic bus sizing.

Note:

The TMS34020's I/O registers cannot be accessed using page mode.

To allow instructions that move data from one area of memory to another to use page-mode, the TMS34020 contains a temporary register file capable of storing eight 32-bit words. This allows up to 9 words to be read from the source using page mode, and then written to the destination. If the original destination data is not to be combined with the source data in any way, these writes can also be performed using page mode.

8.8 Local-Memory Read and Write Cycles

This section describes all the basic, local-memory, read and write cycles. Figure 8–3 (page 8-19) illustrates general timing of the local-memory interface for reads and writes. Use these figures to observe signal relationships for the various cycles (do not use these figures to determine specific signal timings).

All of the TMS34020's local-memory output signals are synchronous to local clocks LCLK1 and LCLK2. The 90° phase shift between the two local clocks is used to divide each machine cycle into quarter cycles or **Q phases**. All local-memory output signals start transitions at the beginning of a Q phase.

All subsequent timing diagrams in this chapter indicate the following:

- $\overline{GI}$ is sampled low at the beginning of each memory cycle. After the memory cycle, $\overline{GI}$ may be at either level because the TMS34020 may or may not retain control of the local-memory bus after the cycle completes. This is shown by the dual level for $\overline{DDOUT}$ during Q1 after memory cycle completion. If bus control is regained, $\overline{DDOUT}$ is driven to high impedance at this time; otherwise, it remains high. Chapter 11, *Multiprocessing and System Architecture*, discusses $\overline{GI}$ in detail.

- $\overline{R0}$ and $\overline{R1}$ are shown in all the possible states that could occur during a particular cycle. For instance, in Figure 8–3, they initially indicate either a no-request or low-priority code (either may be appropriate if no memory cycle was performed in the previous machine state), or an access-termination code (identifying the end of the previous access). During the memory access, the low-priority code is output, followed by the access-termination code during the access' final machine state. Chapter 11 also discusses $\overline{R0}$ and $\overline{R1}$.

- Wherever appropriate, a page-mode sequence (consisting of a standard memory cycle followed by one page-mode cycle) is shown. Obviously, the number of page-mode cycles is typically greater than 1.

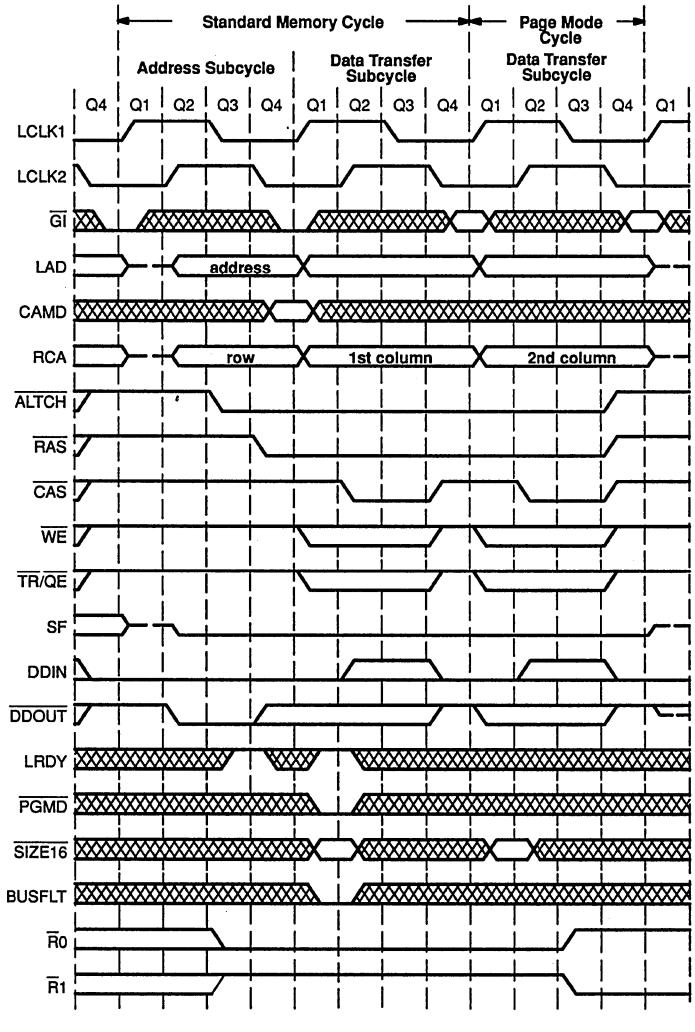
Note also that

- CAMD is sampled on LCLK1's low-to-high transition, immediately following the address/status subcycle. This selects the logical address bits that are output on the RCA bus at column-address time. (See Section 8.16, page 8-50.)

- LRDY, BUSFLT, and $\overline{PGMD}$ are sampled on LCLK2's low-to-high transition in each machine state of the sequence's first data subcycle (as described in Sections 8.6 and 8.7, pages 8-12 and 8-15). $\overline{SIZE16}$ is sampled at this time in each data subcycle (during page-mode cycles as well). This is discussed in Section 8.9 (page 8-25).

LRDY is shown being sampled high on LCLK2's high-to-low transition during an address/status subcycle. This is to maintain compatibility with future pin-compatible TMS340x0 devices. Current devices ignore the state of LRDY. However, future devices may not; the value of LRDY sampled at this time may be used to prolong the address/status subcycle. This eases interfacing to DRAMs if the TMS34020's LCLK frequency is increased above 10MHz. Maintaining LRDY high at this time in your current designs ensures that the address/status subcycle timing shown in the diagrams will not be affected when you use future TMS3402x devices.

Figure 8–3. General Timing of the Local-Memory Read and Write Cycles



8.8.1 Local-Memory Read Cycle Timing (with Page Mode)

A local-memory read cycle (with page mode) transfers data and instructions to the TMS34020 from memory that supports page-mode accesses, as Figure 8–4 shows. The status code on LAD0—LAD4 identifies a data transfer, cache fill, instruction fetch, interrupt-vector fetch, or pixel operation.

$\overline{\text{DDOUT}}$ goes low during the address subcycle to enable the address through external bus transceivers to the memory. $\overline{\text{DDOUT}}$ returns high after the address is latched, indicating that a memory read cycle is about to take place. The LAD bus goes into a high-impedance state following the address subcycle, allowing data from the addressed memory to be placed on the bus.

External decode logic asserts $\overline{\text{PGMD}}$ low at the beginning of Q2 after $\overline{\text{RAS}}$ low (the first data subcycle), indicating that the memory supports page-mode operation. External logic also asserts LRDY high and BUSFLT low at this time, indicating that the cycle may continue without a retry, bus fault, or wait-state insertion. $\overline{\text{SIZE16}}$ must be valid at the beginning of Q2 during all data transfer cycles, indicating whether the access is to a 32-bit or a 16-bit memory device. In Figure 8–4, the external logic asserts $\overline{\text{SIZE16}}$ high to allow access to 32-bit memory.

The TMS34020 outputs DDIN high during the data transfer cycle, allowing the external bus transceivers to enable data to the processor. Data from the accessed memory must be valid at the beginning of Q4 in the data transfer cycle (with some data setup time). The data from the memory should be disabled with the low-to-high transition of $\overline{\text{ALTCH}}$, $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, or $\overline{\text{TR/QE}}$, or with DDIN's high-to-low transition. These transitions occur well before the time at which the LAD bus turns on to output the address for the next cycle.

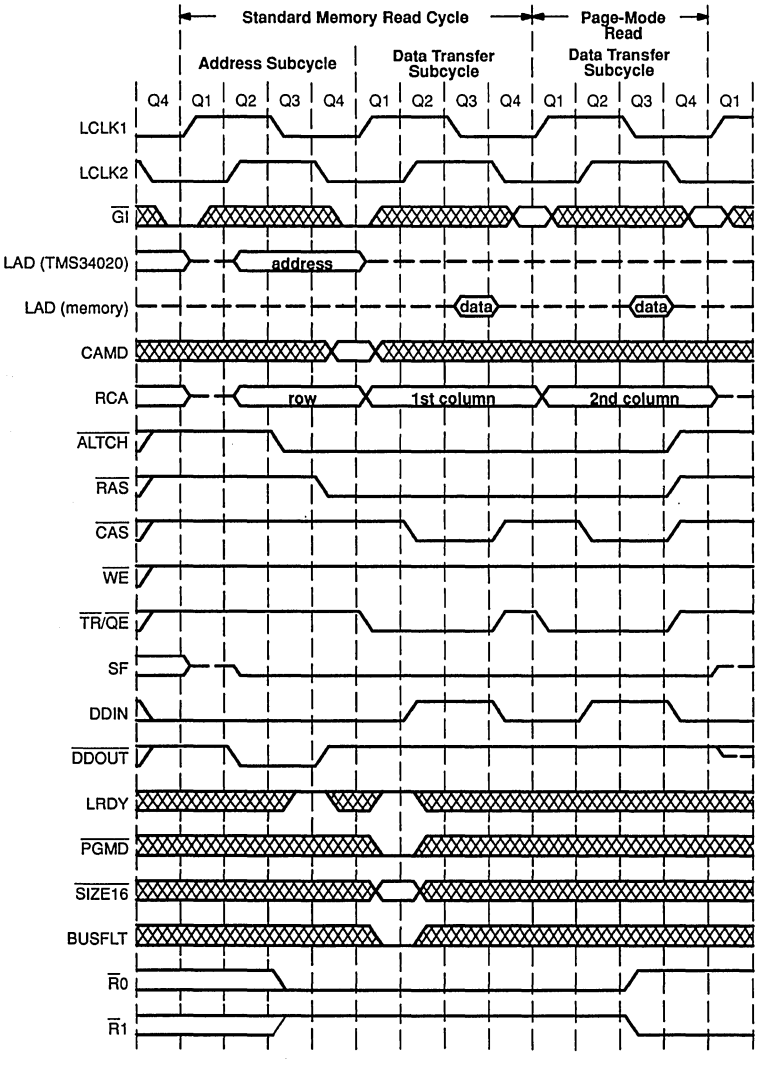
Note that the TMS34020 does not sample LRDY, $\overline{\text{PGMD}}$, and BUSFLT during subsequent page-mode cycles when accessing 32-bit memory. The TMS34020 does not sample $\overline{\text{SIZE16}}$ during the page-mode cycle, because the memory has responded that it supports 32-bit accesses. Section 8.9 (page 8-25) discusses dynamic bus sizing and $\overline{\text{SIZE16}}$ in detail.

8.8.2 Local-Memory Write-Cycle Timing (with Page Mode)

As Figure 8–5 (page 8-22) shows, this page-mode write cycle transfers data from the TMS34020 to memory that supports page-mode accesses. LAD0—LAD31 output the data during the data transfer subcycle. The status code output on LAD0—LAD3 identifies a data transfer or a pixel operation.

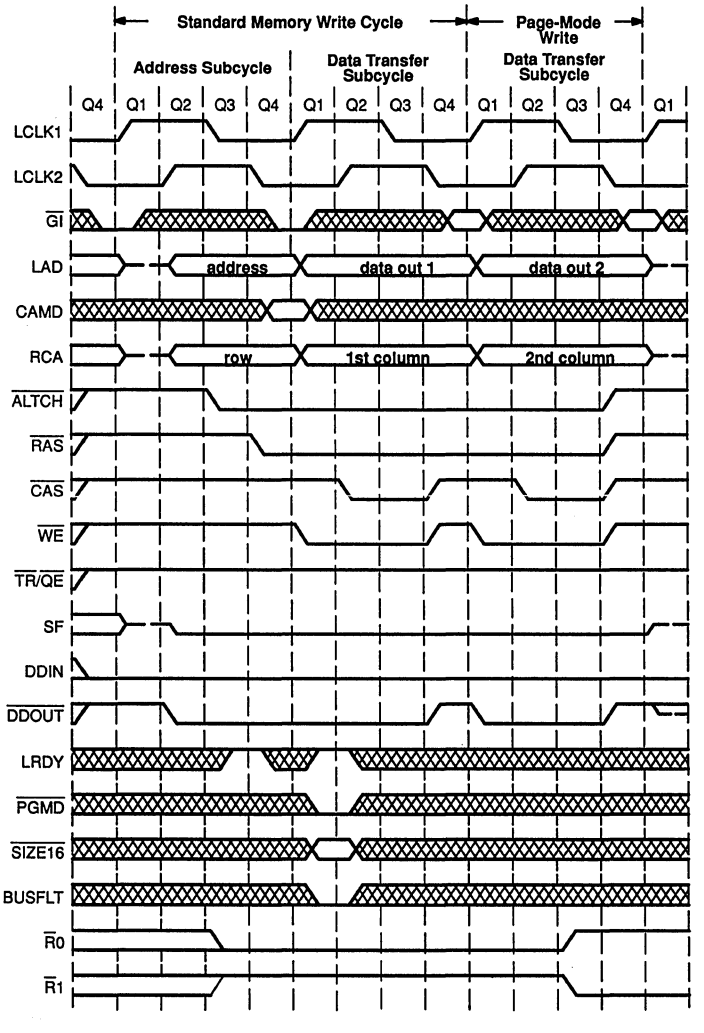
$\overline{\text{DDOUT}}$ remains low after the initial address output on LAD (during Q4 after $\overline{\text{RAS}}$ goes low). This indicates that a memory write cycle is about to take place, allowing the data to be output through external data transceivers. $\overline{\text{WE}}$ is output low before the $\overline{\text{CAS}}$ signals go low to implement an early write to the DRAMs and VRAMs. Because data is valid both before and after the $\overline{\text{CAS}}$ signals are asserted low, external devices can latch the data on either the high-to-low or low-to-high edge of a signal that is the logical-OR of $\overline{\text{CAS}}$ and $\overline{\text{WE}}$.

Figure 8–4. Local-Memory Read-Cycle Timing (with Page Mode)



Key: **LAD (TMS34020):** The TMS34020 outputs this to the LAD bus.
LAD (memory): Memory outputs this to the LAD bus.

Figure 8–5. Local-Memory Write-Cycle Timing (with Page Mode)



Note that when accessing a 32-bit memory, the TMS34020 does not sample LRDY, $\overline{\text{PGMD}}$, and BUSFLT during subsequent page-mode cycles.The TMS34020 does not sample $\overline{\text{SIZE16}}$ during the page-mode cycle, because the memory has responded that it supports 32-bit accesses. Section 8.9 (page 8-25) discusses dynamic bus sizing and $\overline{\text{SIZE16}}$ in detail.

8.8.3 Local-Memory Read/Write or Read-Modify-Write Cycle Timing

The TMS34020 performs a read-modify-write cycle when inserting a field that crosses byte boundaries. This is not the same as the read-modify-write cycle specified for some DRAMs, because the $\overline{\text{CAS}}$ signals do not remain active low between the read and write. The read-modify-write operation consists of:

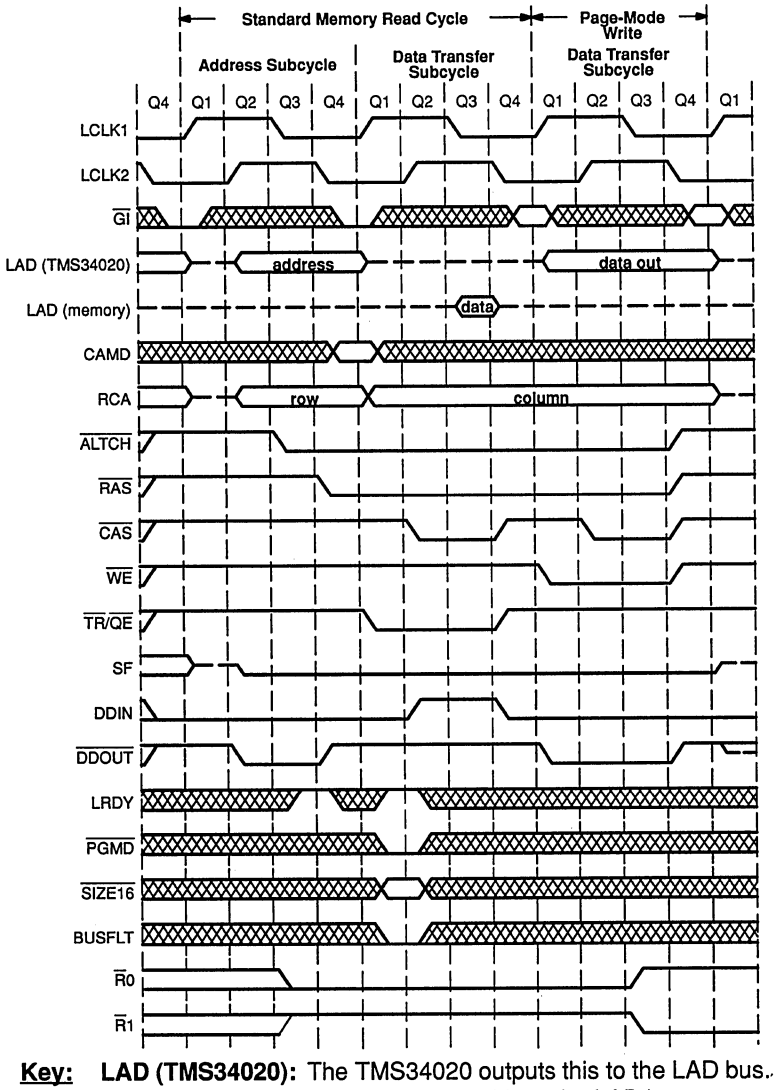
Step 1: A word is read from memory.

Step 2: The portion of that word corresponding to the field being inserted is loaded with the new value.

Step 3: The modified word is written back to memory.

If the accessed memory supports page-mode accesses, this operation occurs as a read followed by a page-mode write. Figure 8–6 shows the timing for the cycle. The status code output on LAD0—LAD3 identifies a data transfer, pixel operation, or bus-locked operation. Note that the column address presented during the write data transfer cycle is the same as that present during the read data transfer cycle.

Figure 8–6. Local-Memory Read/Write or Read-Modify-Write-Cycle Timing



Key: **LAD (TMS34020):** The TMS34020 outputs this to the LAD bus.
LAD (memory): Memory outputs this to the LAD bus.

If a bus fault or retry aborts this cycle, the cycle restarts from the aborted access. If the cycle is interrupted by a higher priority access between the read and the write, the memory controller restarts it from the write (unless the access was bus locked, in which case it restarts from the read).

Note that when accessing 32-bit memory, the TMS34020 does not sample LRDY, $\overline{\text{PGMD}}$, and BUSFLT during subsequent page-mode cycles. The TMS34020 does not sample $\overline{\text{SIZE16}}$ during the page-mode cycle, because the memory has responded that it supports 32-bit accesses. Section 8.9 (page 8-25) discusses dynamic bus sizing and $\overline{\text{SIZE16}}$ in detail.

8.8.4 Host-Initiated Local-Memory Read and Write Cycles

When the host requests access to the TMS34020's local memory, the host interface schedules a host request to the memory controller. The memory controller then performs the appropriate local-memory cycle. The host can request a read or a write; there are two types of each request.

- **Host read from . . .**
  - **local memory.** This cycle transfers data from the local memory to the bidirectional data transceivers required to interface the LAD bus to the host's data bus.
  - **TMS34020 I/O register.** This cycle transfers data from one of the TMS34020's I/O registers to the bidirectional data transceivers required to interface the LAD bus to the host's data bus.
- **Host write to . . .**
  - **local memory.** This cycle transfers data from the bidirectional data transceivers to the local memory.
  - **TMS34020 I/O register.** This cycle transfers data from the bidirectional data transceivers to one of the TMS34020's I/O registers.

Chapter 7, *Communicating with a Host Processor*, describes these cycles in detail. Cycle timings for the four host local-memory cycles are in Figure 7–8 (page 7-19), Figure 7–9 (page 7-20), Figure 7–15 (page 7-26), and Figure 7–16 (page 7-27).

Host-initiated cycles use page mode only when accessing 16-bit memory.

8.9 Accessing 16-Bit or 32-Bit Memory Devices (Dynamic Bus Sizing)

The TMS34020's dynamic bus sizing capability allows the local memory to be organized as 32 bits wide, 16 bits wide, or a combination of the two. This can be useful for minimizing system cost if only a small amount of a particular memory type (such as ROM or EPROM) is required.

Initially, when the TMS34020 accesses a new location, the 16-bit word-select bit (S) is 0. (S is output on LAD4 during the address/status subcycle and on RCA0 during the data subcycle.) S=0 means that the address output is aligned to a long-word. If the memory is 16 bits wide, only 16 bits of data are transferred during this access. Another memory access with S=1 must be performed to transfer the remaining 16 bits. The $\overline{\text{SIZE16}}$ pin provides a mechanism for doing this.

The TMS34020 samples $\overline{\text{SIZE16}}$ on LCLK2's low-to-high transition during the data subcycle (at the same time as LRDY, BUSFLT, and $\overline{\text{PGMD}}$). $\overline{\text{SIZE16}}$ conveys two different but related pieces of information to the TMS34020, depending on the value of S output during the memory cycle.

- During the initial access to a location (when S=0), $\overline{\text{SIZE16}}$ determines whether the memory is 32 or 16 bits wide. Asserting $\overline{\text{SIZE16}}$ low selects 16-bit wide memory.
- If $\overline{\text{SIZE16}}$ was asserted low during the initial access to a location, the TMS34020 performs a second access (with S=1). During this memory cycle, $\overline{\text{SIZE16}}$ is used to determine which half of the LAD bus the memory is attached to (LAD0—LAD15 or LAD16—LAD31).

Table 8–4 summarizes the interpretation of the different combinations of $\overline{\text{SIZE16}}$ and S.

Table 8–4. Interpretation of $\overline{\text{SIZE16}}$

$\overline{\text{SIZE16}}$ (S=0)	$\overline{\text{SIZE16}}$ (S=1)	Interpretation
1	—	32-bit access
0	0	16-bit access via LAD0—LAD15
0	1	16-bit access via LAD16—LAD31

Note: If $\overline{\text{SIZE16}}$ is sampled as a 1 when S=0, no access is made with S=1.

Determining which level to assert for the $\overline{\text{SIZE16}}$ pin depends on which half of the LAD bus the memories are wired to:

- When data is transferred over LAD0—LAD15, the external decode logic should simply assert $\overline{\text{SIZE16}}$ low during all accesses to 16-bit memory.
- When data will be transferred over LAD16—LAD31, the external decode logic can use the S bit's value to determine which level to assert on $\overline{\text{SIZE16}}$ during accesses to 16-bit memory.

8.9.1 Data Transfer Using Dynamic Bus Sizing

With dynamic bus sizing, the initial access (with S=0) is no different from a regular 32-bit access. The TMS34020 asserts the appropriate $\overline{CAS}$ strobes for the access and transfers data over the LAD bus in the normal way. During the second access (with S=1), the local-memory control signals behave differently:

- The 2 least significant $\overline{CAS}$ strobes are swapped with the most significant $\overline{CAS}$ strobes; the levels initially output on $\overline{CAS0}$ and $\overline{CAS1}$ (when S=0) are output on $\overline{CAS2}$ and $\overline{CAS3}$, respectively, and vice versa.
- During a local-memory write, the LAD bus halves are swapped; the data initially output on LAD0—LAD15 (when S=0) is output on LAD16—LAD31, and vice versa.
- During a local-memory read, the data latched from LAD0—LAD15 is swapped onto bits 16—31 of the internal data bus, and vice versa.

This mechanism allows 16-bit memories to be wired directly to one half of the LAD bus and to the corresponding pair of $\overline{CAS}$ strobes, without the need for any external multiplexing logic.

Figure 8–7 shows a read operation from 16-bit memory. All $\overline{CAS}$ strobes are active during both read cycles. Figure 8–8 shows a write operation to a 16-bit memory device. Note how the $\overline{CAS}$ strobes have been swapped during the second access. Both diagrams show page-mode operation (discussed in Section 8.9.2).

Note:

During local-memory read cycles to 16-bit memory, the TMS34020 must merge the 16 bits of valid data latched during the second (S=1) cycle with the 16 bits of valid data latched during the initial (S=0) cycle, forming the full 32 bits for the access. During the initial cycle, all 32 bits of data on the LAD bus are latched, but only half is valid. During the second cycle, the level sampled on $\overline{SIZE16}$ determines which half of this data is overwritten by the second 16 bits of valid data. Because the halves of the LAD bus can be swapped only during the second access, memories connected to LAD16— LAD31 must actually transfer the 16 MSBs of data during the initial cycle, and the 16 LSBs during the second cycle. This can be achieved by inverting the S bit before connecting it to the memory's least significant address pin.

When performing a read-modify-write operation to 16-bit memory, the TMS34020 performs the 2 read cycles first, followed by the 2 write cycles.

If the local-memory system indicates that the location being accessed is arranged as 16-bit memory, the TMS34020 always accesses both 16-bit words, even if it was only attempting to insert a field in the first of the two 16-bit words.

Figure 8–7. Dynamic Bus Sizing for a Read Cycle (Connection to LAD0—LAD15, Indicated by $\overline{SIZE16}$ Low During 2nd Data Cycle)

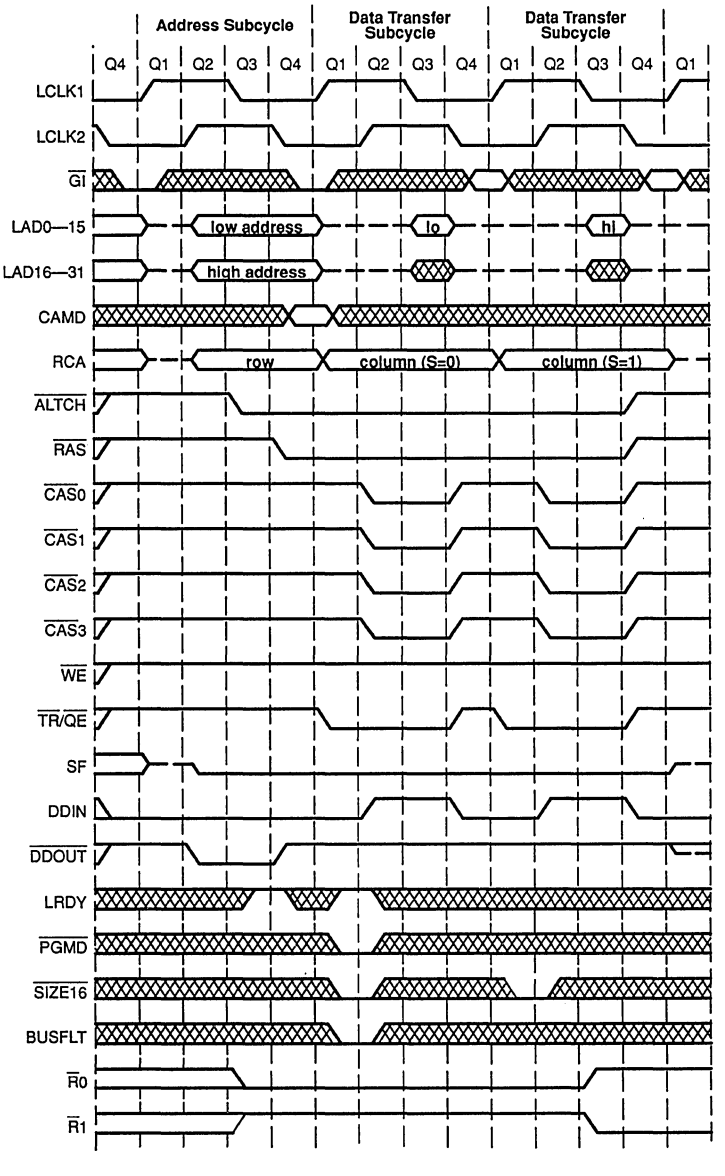
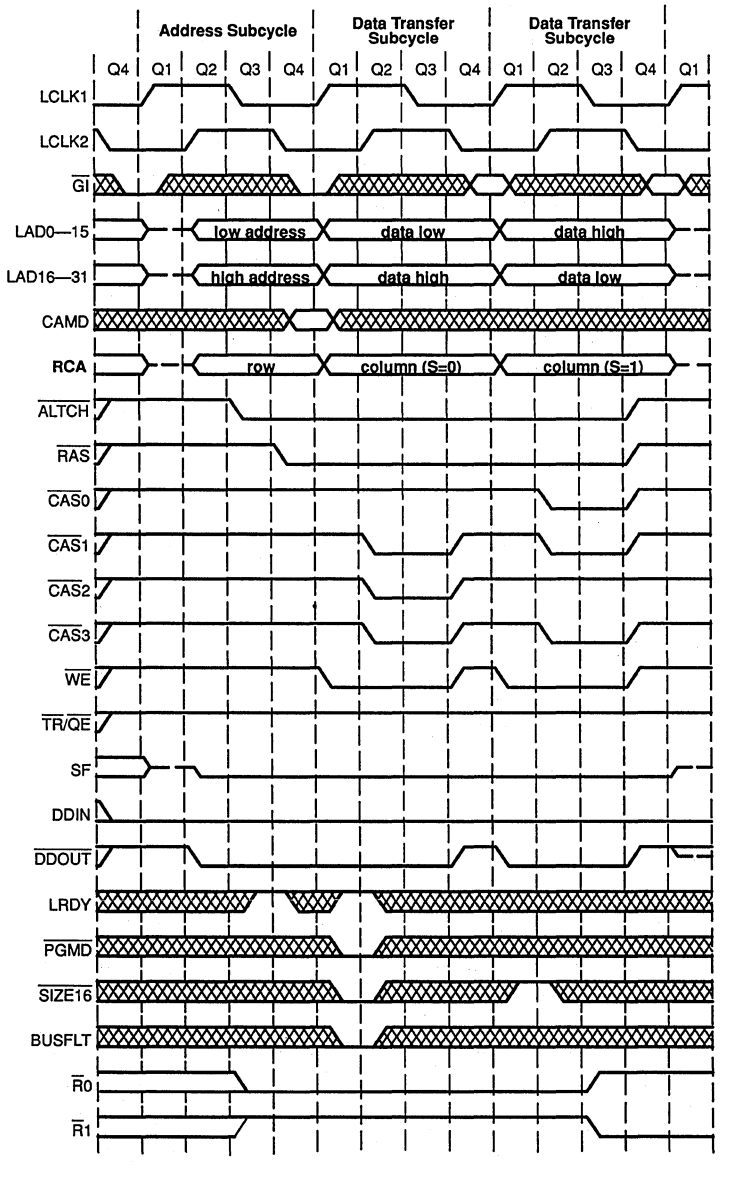


Figure 8–8. Dynamic Bus Sizing for a Write Cycle (Connection to LAD16—LAD31, Indicated by $\overline{\text{SIZE16}}$ High During 2nd Data Cycle)



8.9.2 Page Mode and Dynamic Bus Sizing

Whenever possible, the TMS34020 uses page mode for accessing 16-bit memory. The TMS34020 samples $\overline{\text{SIZE16}}$ at the same time as LRDY, $\overline{\text{BUSERR}}$, and $\overline{\text{PGMD}}$ (on LCLK2's low-to-high transition immediately following the address/status subcycle). If the TMS34020 samples $\overline{\text{PGMD}}$ low and

$\overline{\text{SIZE16}}$ high (indicating that page-mode accesses can be made to 32-bit memory), it does not sample $\overline{\text{SIZE16}}$ again during the page-mode sequence that may follow. If, however, the TMS34020 samples $\overline{\text{SIZE16}}$ low (indicating 16-bit memory), it samples the signal during all subsequent page-mode accesses to determine which half of the LAD bus the memory is connected to.

- If the memory is connected to LAD0—LAD15, $\overline{\text{SIZE16}}$ is asserted low during all subsequent page-mode cycles.
- If the memory is connected to LAD16—LAD31, the value of S output on the RCA bus can be used to determine the level of $\overline{\text{SIZE16}}$ required for each page-mode cycle, as Table 8–4 (page 8-25) illustrates.

Page mode greatly increases the rate of access to 16-bit memory. For example, reading or writing a long-word in 16-bit memory requires 2 standard memory cycles without page mode (at least 4 machine states), or a standard memory cycle followed by a page-mode cycle if page mode is supported (3 machine states). If subsequent words are accessed using page mode, only 2 machine states are required for each long-word. Similarly, a read-modify-write operation takes only 5 machine states in page mode, compared to 8 without.

8.9.3 Bus-Locked Operation and Dynamic Bus Sizing

The SWAPF instruction initiates the bus-locked read-modify-write operation. This operation is used for passing semaphores to and from a memory location that both the host and the TMS34020 can access. It is uninterruptible; if the cycle *is* interrupted between the read and the write, it starts again from the read. This ensures that the write always immediately follows the read. However, if 2 reads and 2 writes are required, uninterruptibility cannot be guaranteed, and because of this, the bus-locked local-memory cycle does not sample the $\overline{\text{SIZE16}}$ pin and does not support 16-bit memory.

A single read and write are performed. Each cycle outputs only S=0. Any semaphores contained in 16-bit memory locations that require S=1 in order to access them are not accessed during a bus-locked operation.

8.10 VRAM Serial-Register Transfers

The TMS34020 provides control for data transfers between the serial registers and the memory array of VRAMs. This is achieved by the appropriate timing of the SF, $\overline{\text{CAS}}$, $\overline{\text{TR/QE}}$, and $\overline{\text{WE}}$ pins of the VRAMs at the falling edges of $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$. The cycles include

- Memory-to-serial-register cycle (VRAM read transfer)
- Memory-to-split-serial-register cycle (VRAM split-serial-register-read transfer)
- Serial-register-to-memory cycle (VRAM write, pseudo-write transfers)
- Serial-register-to-memory cycle (VRAM alternate-write transfer)

8.10.1 Memory-to-Serial-Data-Register Cycle (VRAM Read Transfer)

Performed when
- executing a pixel-read instruction and
- CST[[DPYCTL]] =1

or
- video timing logic requests a horizontal- blank-reload cycle and
- VCE[[DPYCTL]] = 0

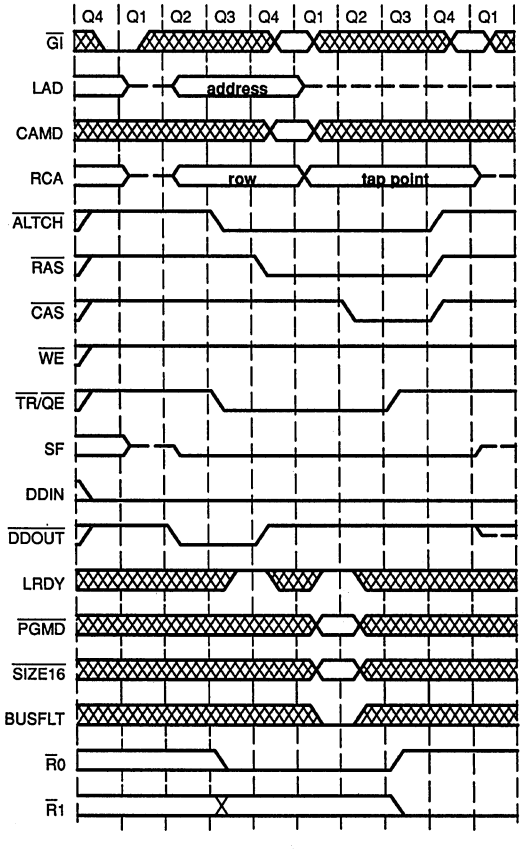
Indicated by
- $\overline{TR}/\overline{QE}$ and SF low and
- $\overline{CAS}$ and $\overline{WE}$ high when $\overline{RAS}$ goes low

Status code identifies a memory-to-register transfer
- 0100_2 (initiated by the video timing logic) or
- 0101_2 (initiated by the CPU)

Although the TMS34020 ignores $\overline{PGMD}$ and $\overline{SIZE16}$ during this cycle, you should hold them at a valid level (as the figures show them).

As Figure 8–9 shows, this cycle causes $\overline{TR}/\overline{QE}$ to make its low-to-high transition at the beginning of Q3.

Figure 8–9. Memory-to-Serial-Data-Register Cycle (VRAM Read Transfer)



8.10.2 Memory-to-Split-Serial-Data-Register Cycle (VRAM Split-Register Midline-Reload Transfer)

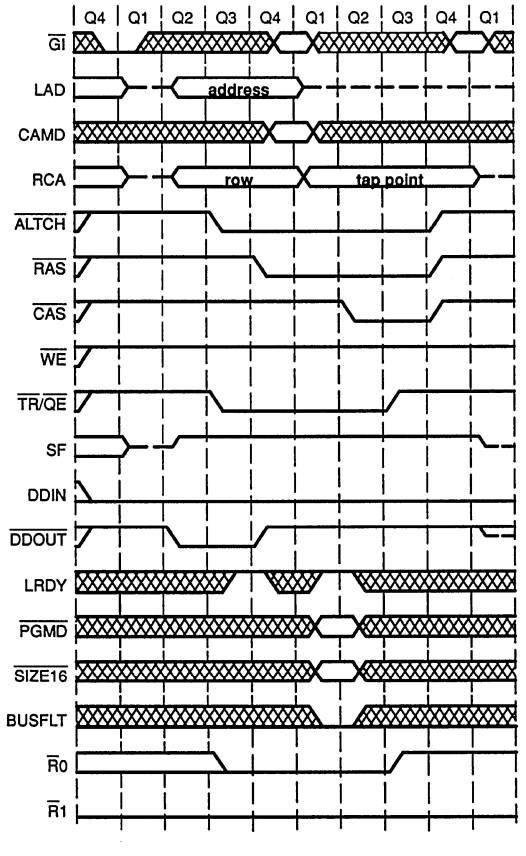
Performed when
- SCOUNT overflows from all 1s to all 0s,
- VCE[DPYCTL] is cleared to 0, and
- SSV[DPYCTL] and SRE[DPYCTL] are set to 1

Indicated by
- $\overline{TR}/\overline{QE}$ low and
- $\overline{CAS}$, SF, and $\overline{WE}$ high when $\overline{RAS}$ goes low

Status code 0100_2 (video-initiated VRAM-memory-to-register cycle)

Although the TMS34020 ignores $\overline{PGMD}$ and $\overline{SIZE16}$ during this cycle, they should be held at valid levels.

Figure 8–10. Memory-to-Split-Serial-Data-Register Cycle (VRAM Split-Register Midline-Reload Transfer)



8.10.3 Serial-Data-Register-to-Memory Cycle (VRAM Write Transfer and Pseudo-Write Transfer)

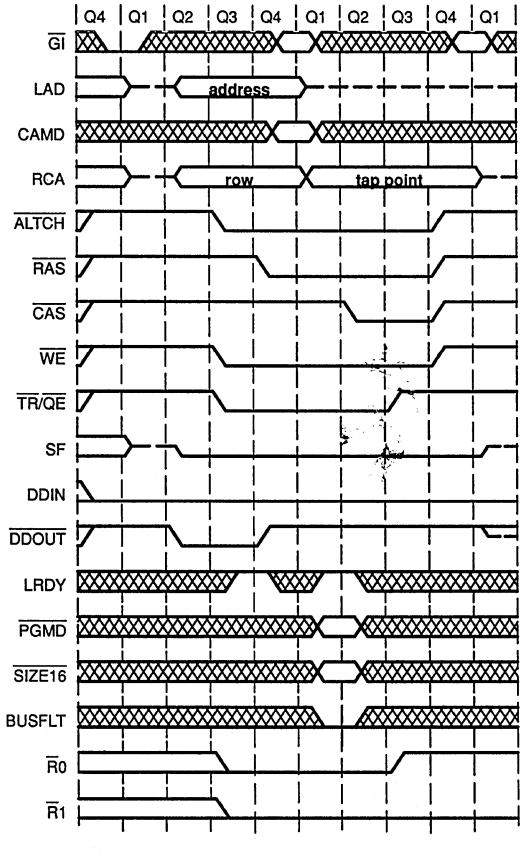
Performed when
- video timing logic requests a horizontal-blank reload and
- VCE[DPYCTL] and SRE[DPYCTL] are set to 1

Indicated by
- $\overline{TR/QE}$, SF, and $\overline{WE}$ low and
- $\overline{CAS}$ high when $\overline{RAS}$ goes low

Status code 0100_2 (video-initiated VRAM-memory-to-register cycle)

The level of the VRAM $\overline{SOE}$ pin at the falling edge of $\overline{RAS}$ selects between write transfer ($\overline{SOE}$ low) and pseudo-write transfer ($\overline{SOE}$ high) cycles. Section 9.13, Video RAM Control (page 9-42), describes application of this cycle.

Figure 8–11. VRAM Write Transfer and Pseudo-Write Transfer



Although the TMS34020 ignores $\overline{PGMD}$ and $\overline{SIZE16}$ during this cycle, they should be held at valid levels as shown. Note that external logic must provide control of the VRAM $\overline{SOE}$ pin.

8.10.4 Serial-Data-Register-to-Memory Cycle (VRAM Alternate-Write Transfer)

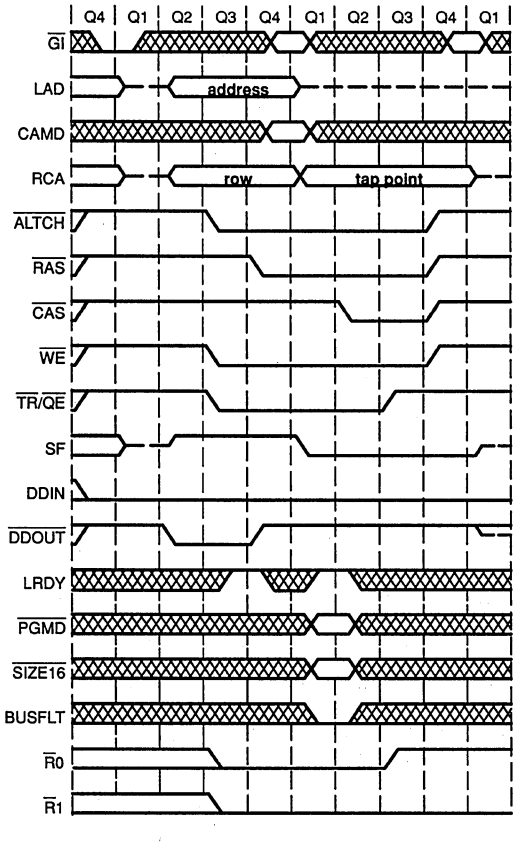
Performed when
- pixel-write instruction is executed and
- CST[DPYCTL] is set to 1

Indicated by
- $\overline{CAS}$, $\overline{WE}$, and SF high when $\overline{RAS}$ goes low and
- $\overline{TR/QE}$ low when $\overline{RAS}$ goes low

Status code 0101_2 (CPU-initiated VRAM-memory-to-register cycle)

This cycle does not require use of the VRAM's $\overline{SOE}$ pin and does not affect the status of the serial I/O pins. Although the TMS34020 ignores $\overline{PGMD}$ and $\overline{SIZE16}$ during this cycle, they should be held at valid levels, as shown.

Figure 8–12. VRAM Alternate-Write Transfer



Note:

VRAMs without an SF pin do not directly support the VRAM alternate-write transfer. However, the only difference between this cycle and the VRAM write transfer (see Section 8.10.3) is the state of SF at the falling edge of $\overline{RAS}$, so both cycles appear the same to the VRAMs. In this case, the external hardware used to control the VRAM $\overline{SOE}$ pin must operate for both types of cycle.

8.11 VRAM Write-Mask Local-Memory Cycles

Some VRAMs (such as the TMS44251) contain special logic to enhance the performance of writing certain types of data to the memory. The TMS34020 provides direct support for the VRAM's on-chip write-mask and block-write capability.

A write-mask register within the VRAM allows each 1-bit plane within the memory array to be selectively writable or write protected. This is equivalent to the TMS34020's plane mask (described in Section 12.10, page 12-39). When plane masking is used, the memory controller must normally perform a read-modify-write operation, so that only the required bits of each pixel are modified. However, when a copy of the plane mask is stored within the VRAM, the memory controller need only perform a special write cycle that uses the VRAM's write mask. The plane masking is then carried out *within the VRAM*, as data is written. Obviously, the ability to perform a write instead of a read-modify-write significantly increases the rate at which data within the memory can be modified when using plane masking.

Writing a 1 to VEN[[CONFIG]] indicates that the VRAMs in the system contain a write-mask register. The TMS34020 provides the following local-memory cycles to enable the VRAM's write mask to be used:

- Load-write-mask cycle
- Write cycle (with mask)
- Block-write cycle (with mask)

Section 8.12 (page 8-37) discusses block-write cycles, including block-write with mask.

8.11.1 Load-Write-Mask Cycle

Performed when
- PMASKL or PMASKH is written to and
- VEN[[CONFIG]] is set

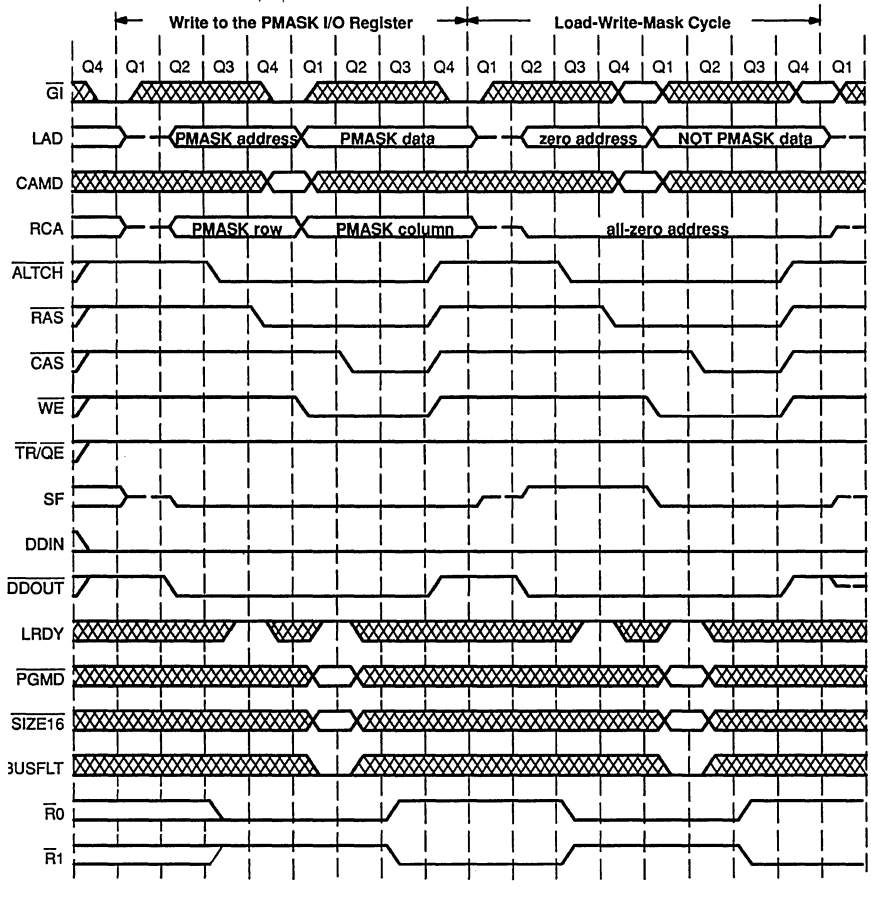
Indicated by
- $\overline{CAS}$, $\overline{WE}$, $\overline{TR}/\overline{QE}$, and SF high at the falling edge of $\overline{RAS}$ and
- SF low at the falling edge of $\overline{CAS}$

Status code 0110_2 (write-mask load)

After the plane mask is copied into the TMS34020's PMASK registers, the 1s complement of PMASK is written to a special register on the VRAM that is used in subsequent cycles requiring a write mask. (The 1s complement of PMASK is output on the load-write-mask cycle because the TMS34020 *masks* the bits that are set high in the PMASK registers and the VRAM write mask *enables* the bits that are set high in the write-mask register.)

Although the TMS34020 ignores CAMD, $\overline{\text{PGMD}}$, and $\overline{\text{SIZE16}}$ during this cycle, they should be at a valid level at the time they would normally be sampled.

Figure 8–13. Load-Write-Mask Cycle



8.11.2 Write Cycle (with Mask)

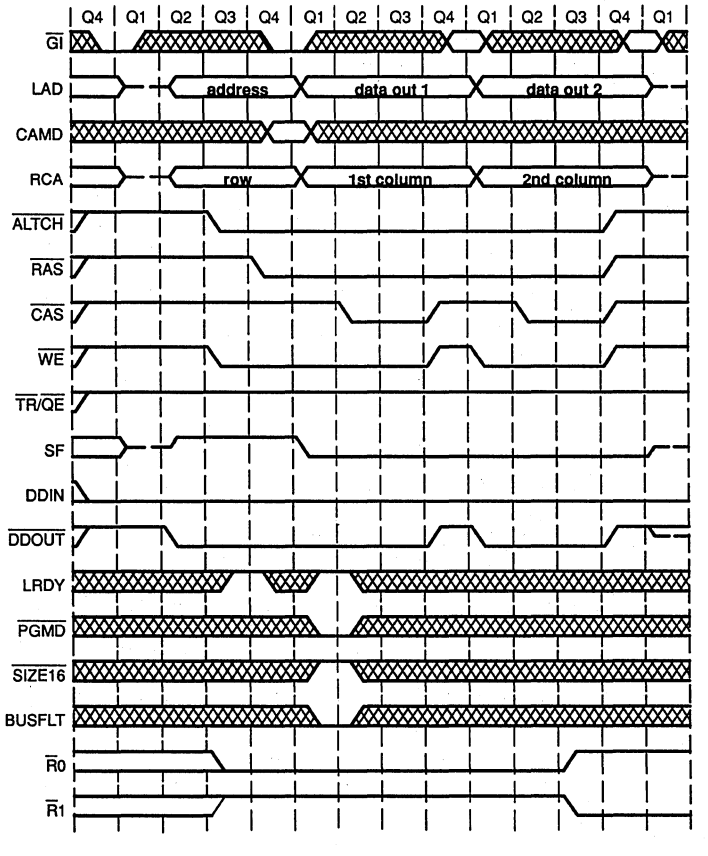
Performed when
- executing a byte-aligned pixel-write instruction,
- the PMASK registers ≠ 0,
- CST[[DPYCTL]] = 0, and
- VEN[[CONFIG]] = 1

Indicated by
- $\overline{CAS}$, $\overline{TR/QE}$, and SF high at the falling edge of $\overline{RAS}$,
- $\overline{WE}$ low at the falling edge of $\overline{RAS}$, and
- SF low at the falling edge of $\overline{CAS}$

Status code 1101_2 (pixel operation)

The data on LAD is written to memory just as a normal DRAM write, except that data in the VRAMs' write mask enables the data bits that are written to memory.

Figure 8–14. Write Cycle (with Mask)



8.12 VRAM Block-Write Local-Memory Cycles

Some VRAMs (like the TMS44251) contain special logic to enhance the performance of writing certain types of data to the VRAM. The TMS34020 directly supports the VRAM's on-chip write-mask and block-write capabilities.

The block-write feature provides a method for writing to up to 128 bits of data during a single TMS34020 local-memory cycle (when writing a specific data pattern). The value latched on each of the VRAM's data pins enables/disables a write of a multi-bit value into the memory. This multi-bit value (typically 4 or 8 bits) is stored in a color register within each VRAM. Block-write may be used to implement several functions; however, these are typically variations of two applications:

- **Fast fills.** A fast fill uses block write to replicate the same pixel value or pattern in an area of memory. The pixel value or pattern is stored in the VRAM's color register.
- **Data expansion.** An example of data expansion is a bitmap stored in compressed form (with 1s representing the presence of a pixel and 0s representing the absence of a pixel). Typically, this sort of bitmap expansion is applied to character fonts, which may be stored in compressed form to save memory; data expansion allows the characters to be displayed in color (multi-bit pixels), but stored in black and white (1-bit pixels).

8.12.1 VRAM Support of Block-Write Cycles

During a block-write cycle, VRAMs ignore the 2 LSBs of the column address. This means that the 4 locations specified by the remainder of the column address are all selected. Which of these 4 locations is written to is determined by the value on the VRAMs' data pins at the falling edge of $\overline{CAS}$; a 1 on a data pin enables a write, a 0 prevents a write. Each data pin is associated with the appropriate location; the least significant data pin controls writes to the least significant location, the most significant data pin controls writes to the most significant location. If all 4 data pins are 1s, all 4 locations are written to.

When a write to a particular location is enabled, the value stored in the VRAM color register is copied into that location. If required, the VRAM write mask (described in Section 8.11.1, page 8-34) can be used to selectively enable or disable the write of each of the bits within the color register.

8.12.2 TMS34020 Support of VRAM Block-Write Cycles

The TMS34020 provides the following local-memory cycles to enable the VRAM block write to be used:

- Load-color-register cycle
- Block-write cycle (without mask)
- Block-write cycle (with mask)

All of these cycles are initiated by specific instructions. Refer to the VLCOL, VBLT, and VFILL instructions in Chapter 13, and Sections 12.5.4 (page 12-14,

VRAM Block Mode PIXBLT) and 12.5.7 (page 12-16, VRAM Block Mode Fill) for more information on restrictions and use of these cycles.

8.12.3 Load-Color-Register Cycle

Performed when
- executing a VLCOL instruction

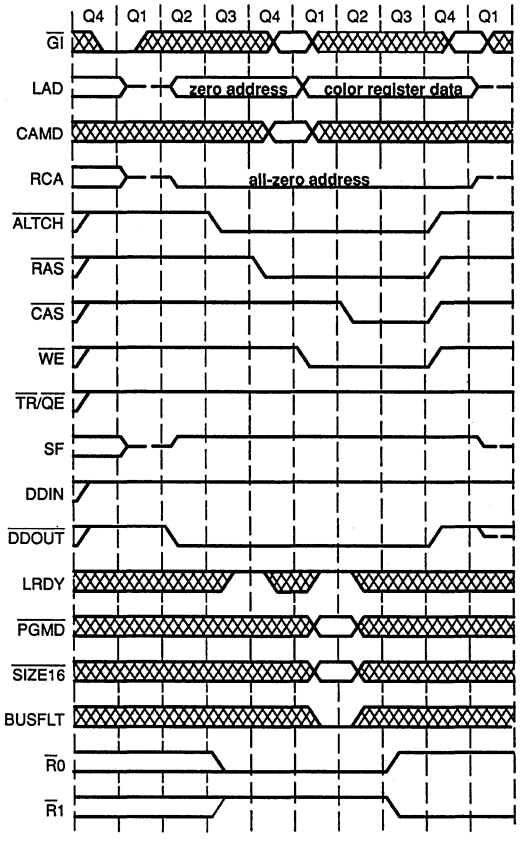
Indicated by
- $\overline{CAS}$, $\overline{WE}$, $\overline{TR/QE}$, and SF high at the falling edge of $\overline{RAS}$ and
- SF high at the falling edge of $\overline{CAS}$

Status code 0111_2 (color-latch load)

The data in COLOR1 is output on the LAD bus and is written to a special VRAM register which is used during subsequent block-write cycles. The VRAMs ignore the address output during this cycle, so the TMS34020 outputs all 0s on LAD0—LAD31.

Although the TMS34020 ignores CAMD, $\overline{PGMD}$, and $\overline{SIZE16}$ during this cycle, these signals should be at a valid level when they would normally be sampled.

Figure 8–15. Load-Color-Register Cycle



8.12.4 Block-Write Cycle (Without Mask)

Performed when
- executing a VBLT or VFILL instruction and
- PMASK = 0

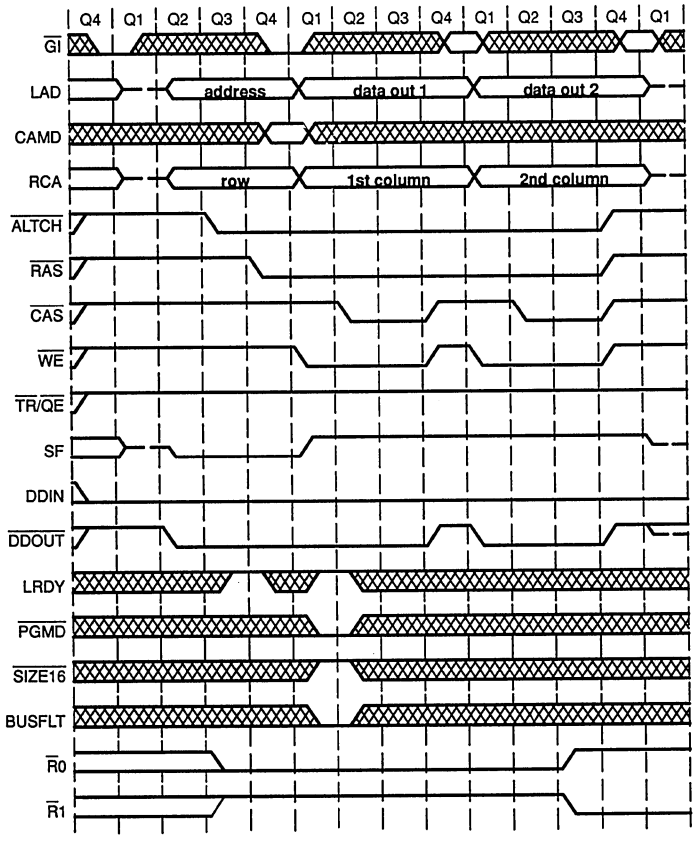
Indicated by
- $\overline{CAS}$, $\overline{WE}$, and $\overline{TR}/\overline{QE}$ high at the falling edge of $\overline{RAS}$,
- SF low at the falling edge of $\overline{RAS}$, and
- SF high at the falling edge of $\overline{CAS}$

Status code 1110_2 (block write)

The data stored in the VRAM's color register is written to the memory locations enabled by the appropriate data bits output on the LAD bus. This cycle allows up to a total of 128 bits to be written at once.

Although 16-bit transfers are allowed on this cycle, external multiplexing logic is required to map the data correctly to the appropriate memories.

Figure 8–16. Block-Write Cycle (Without Mask)



8.12.5 Block-Write Cycle (with Mask)

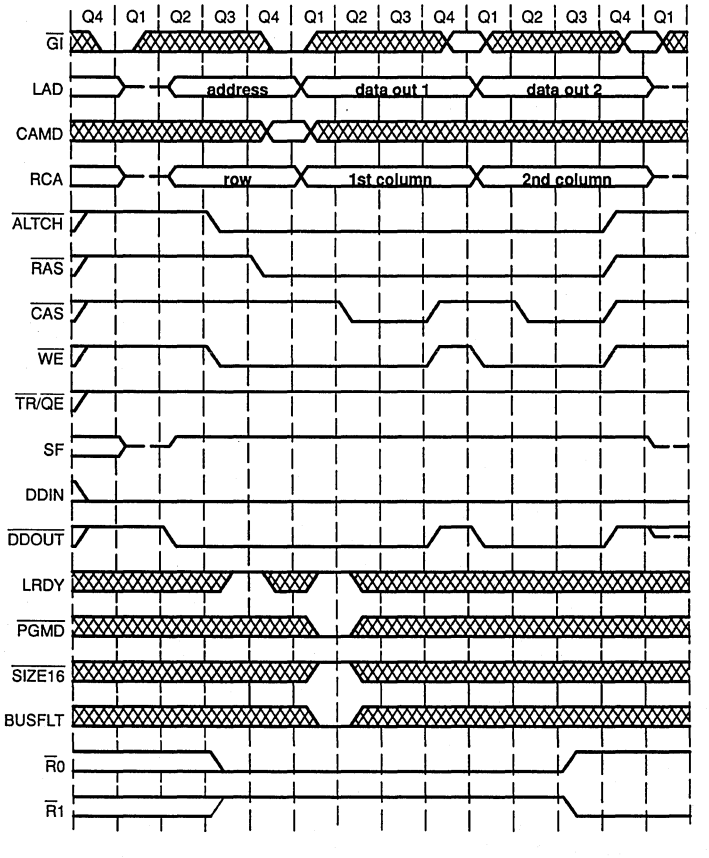
Performed when
- executing a VBLT or VFILL instruction and
- PMASK ≠ 0

Indicated by
- $\overline{CAS}$, $\overline{TR/QE}$, and SF high at the falling edge of $\overline{RAS}$,
- $\overline{WE}$ low at the falling edge of $\overline{RAS}$, and
- SF high at the falling edge of $\overline{CAS}$

Status code 1110_2 (block write)

The data stored in the VRAM's color register is written to the memory locations enabled by the appropriate data bits output on the LAD bus. The value in the VRAM's write mask determines which bits of the VRAM's color register are written. This cycle allows up to a total of 128 bits to be written at once. Although this cycle allows16-bit transfers, external multiplexing logic would be required to correctly map the data to the appropriate memories.

Figure 8–17. Block-Write Cycle (with Mask)



8.12.6 Data Mapping During Block-Write Cycles

Instructions that use block write (VBLT and VFILL) operate on data that is essentially a 1-bit-per-pixel representation of multi-bit pixels in the VRAM. However, because of the data expansion performed inside the VRAMs, the 1-bit data must be remapped (reordered) before being output on the LAD bus.

For example, consider a system that uses 4 bits per pixel and eight 4-bit VRAMs (such as 1-Mbit VRAMs constructed as 256Kx4). For reference, number the VRAMs from 0 to 7, with VRAM 0 connected to the 4 LSBs of the LAD bus, and VRAM 7 connected to the 4 MSBs of the LAD bus. Because the pixel size is 4, each regular memory access to a long-word accesses 8 pixels at a time. Because there are 8 VRAMs, pixels with adjacent addresses are not physically stored in the same VRAM. VRAM 0 holds pixels 0, 8, 16, 24, etc. In general, the VRAM containing pixel n also contains pixels n+8, n+16, n+24, etc. Table 8–5 shows this.

Table 8–5. Connections of 4-Bit VRAMs to the TMS34020 LAD Bus for 4 Bits per Pixel

	VRAM 7				VRAM 6				VRAM 5				VRAM 4				VRAM 3				VRAM 2				VRAM 1				VRAM 0			
VRAM Bit #	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
LAD Bus Connection	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0 Pixel #	7				6				5				4				3				2				1				0			
D1 Pixel #	15				14				13				12				11				10				9				8			
D2 Pixel #	23				22				21				20				19				18				17				16			
D3 Pixel #	31				30				29				28				27				26				25				24			

Note: **Dn Pixel #** refers to the pixels controlled by VRAM data pin Dn during a block-write cycle (n is 0, 1, 2, or 3).

Normally, when the TMS34020 addresses long-word 0, the 8 VRAMs access pixels 0—7; when the TMS34020 addresses long-word 1, the VRAMs access pixels 8—15. Now consider a block-write cycle; the VRAMs ignore the 2 LSBs of the address, selecting 4 contiguous 4-bit locations within each VRAM for access. The 4 data lines act as write enables; a 1 on a data line enables a write of the color register to the corresponding 4-bit location in the VRAM. The data from the TMS34020 is organized so that each bit enables (or disables) the writing of a pixel, allowing 32 pixels to be written for each long-word of data passed to the VRAMs. As Table 8–5 shows, the data on LAD0—3, for example, controls pixels 0, 8, 16, and 24 in the VRAM. It is evident that the order of the bits within the CPU is not the same as the order of the bits expected by the VRAM.

If you want to write pixel 8 only, then the D1 data input of VRAM 0 should be a 1 (that is, the long-word on the LAD bus must be 0000 0002h). However, because the TMS34020 operates on 32 contiguous bits that represent 32 contiguous pixel locations in the memory, enabling pixel 8 is represented by setting bit 8 (so the long-word in the CPU is 0000 0100h).

The TMS34020 logic performs the remapping necessary to reorder the bits within each 32-bit word, so the bits are written out over the LAD bus in the order expected by the VRAMs. Table 8–6 shows the remapping for 4 bits per pixel.

Table 8–6. Data Remapping for Block Write at 4 Bits per Pixel

Internal Data Bit	LAD Pin Number	VRAM Connection	Internal Data Bit	LAD Pin Number	VRAM Connection
0	0	VRAM 0, data pin 0	16	2	VRAM 0, data pin 2
1	4	VRAM 1, data pin 0	17	6	VRAM 1, data pin 2
2	8	VRAM 2, data pin 0	18	10	VRAM 2, data pin 2
3	12	VRAM 3, data pin 0	19	14	VRAM 3, data pin 2
4	16	VRAM 4, data pin 0	20	18	VRAM 4, data pin 2
5	20	VRAM 5, data pin 0	21	22	VRAM 5, data pin 2
6	24	VRAM 6, data pin 0	22	26	VRAM 6, data pin 2
7	28	VRAM 7, data pin 0	23	30	VRAM 7, data pin 2
8	1	VRAM 0, data pin 1	24	3	VRAM 0, data pin 3
9	5	VRAM 1, data pin 1	25	7	VRAM 1, data pin 3
10	9	VRAM 2, data pin 1	26	11	VRAM 2, data pin 3
11	13	VRAM 3, data pin 1	27	15	VRAM 3, data pin 3
12	17	VRAM 4, data pin 1	28	19	VRAM 4, data pin 3
13	21	VRAM 5, data pin 1	29	23	VRAM 5, data pin 3
14	25	VRAM 6, data pin 1	30	27	VRAM 6, data pin 3
15	29	VRAM 7, data pin 2	31	31	VRAM 7, data pin 3

Due to the organization of the VRAM, which accesses 4 bits at a time, the data expansion of the block-write feature is possible only when PSIZE ≥4.

Table 8–7. Block-Write Data Expansion

Bits per pixel	Pixels per 32-bit write	Pixels per block write
1	32	—
2	16	—
4	8	32
8	4	16
16	2	8
32	1	4

The TMS34020 supports the data bus remapping required for 4, 8, 16, and 32 bits per pixel. The TMS34020 uses the value contained within the PSIZE register to determine the correct data remapping to be used. Table 8–8 and Table 8–9 show the data connections and required remapping for 8 bits per pixel. Because two 4-bit VRAMs are needed to hold an 8-bit pixel, the VRAM data lines must be controlled in pairs; thus, the 32-bit bus can write only 16 pixels at a time.

Table 8–8. Connections of 4-Bit VRAMs to the TMS34020 LAD Bus for 8 Bits per Pixel

	RAM 7				RAM 6				RAM 5				RAM 4				RAM 3				RAM 2				RAM 1				RAM 0			
VRAM Bit #	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**	**3**	**2**	**1**	**0**
LAD Bus Connection	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0 Pixel #	3								2								1								0							
D1 Pixel #	7								6								5								4							
D2 Pixel #	11								10								9								8							
D3 Pixel #	15								14								13								12							

Note: **D*n* Pixel #** refers to the pixels controlled by VRAM data pin D*n* during a block-write cycle (*n* is 0, 1, 2, or 3).

The VBLT and VFILL instructions always operate on 32 1-bit pixels at a time. As Table 8–7 shows, the number of pixels that can be written in each block-write cycle is less than 32 when the pixel size is 8 or more. The instructions automatically compensate for this by generating the appropriate number of block-write cycles to ensure that all the data is written. Similarly, 4 block-write cycles are required at 16 bits per pixel, and 8 at 32 bits per pixel.

Table 8–9. Data Remapping for Block Write at 8 Bits per Pixel

Internal Data Bit	Internal Data Bit	LAD Pin Number	VRAM Connection
0	16	0 and 4	VRAM 0, data 0 and VRAM 1, data 0
1	17	8 and 12	VRAM 2, data 0 and VRAM 1, data 0
2	18	16 and 20	VRAM 4, data 0 and VRAM 1, data 0
3	19	24 and 28	VRAM 6, data 0 and VRAM 1, data 0
4	20	1 and 5	VRAM 0, data 1 and VRAM 1, data 1
5	21	9 and 13	VRAM 2, data 1 and VRAM 1, data 1
6	22	17 and 21	VRAM 4, data 1 and VRAM 1, data 1
7	23	25 and 29	VRAM 6, data 1 and VRAM 1, data 1
8	24	2 and 6	VRAM 0, data 2 and VRAM 1, data 2
9	25	10 and 14	VRAM 2, data 2 and VRAM 1, data 2
10	26	18 and 22	VRAM 4, data 2 and VRAM 1, data 2
11	27	26 and 30	VRAM 6, data 2 and VRAM 1, data 2
12	28	3 and 7	VRAM 0, data 3 and VRAM 1, data 3
13	29	11 and 15	VRAM 2, data 3 and VRAM 1, data 3
14	30	19 and 23	VRAM 4, data 3 and VRAM 1, data 3
15	31	27 and 31	VRAM 6, data 3 and VRAM 1, data 3

Note: Data bits 0 & 16 use the same LAD pin numbering and VRAM connection information, as do data bits 1 & 17, 2 & 18, etc.

8.13 DRAM-Refresh Local-Memory Cycles

The TMS34020 supports DRAM and VRAM memory refresh with $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh cycles. Each time the internal refresh counter counts the number of machine states indicated by RR[[CONFIG]], a DRAM-refresh request is scheduled. DRAM-refresh cycles are performed according to the local-memory cycle priorities outlined in Section 8.3 (page 8-6). The TMS34020 can keep track of up to 15 pending DRAM-refresh requests (although it is unlikely that this many refreshes could ever be scheduled but not performed).

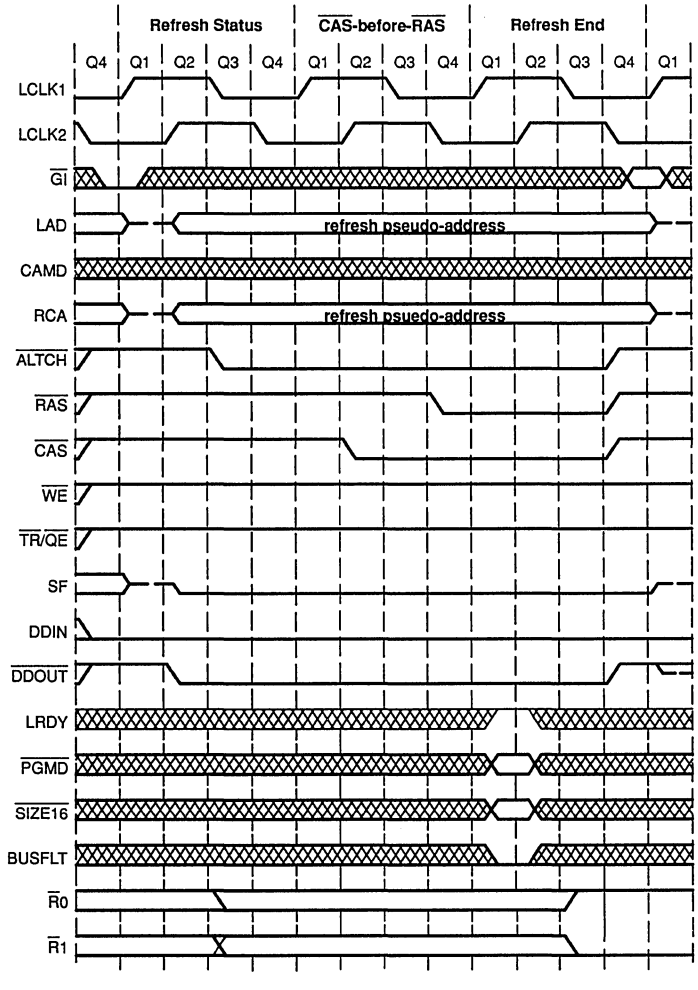
The refresh cycle has been implemented on the TMS34020 to use 3 machine cycles (12 quarter cycles), so the refresh status may be used to modify the generation of the $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ signals to the DRAMs.

The refresh pseudo-address output to RCA0—RCA12 and LAD0—LAD31 comes from the 16-bit REFADR register, which is incremented after each refresh cycle. The 16 bits of the address are placed on LAD16—LAD31 and LAD0—LAD3 contain the refresh status code. All other LAD bus lines are 0s. The logical addresses on RCA0—RCA12 corresponding to LAD16—LAD31 also output the address from REFADR. The LAD and RCA buses are held constant through the refresh cycle.

Note that CAMD, $\overline{\text{PGMD}}$, and $\overline{\text{SIZE16}}$ are not sampled during a refresh cycle, although $\overline{\text{PGMD}}$ and $\overline{\text{SIZE16}}$ must be held at a valid level as indicated in Figure 8–18. Once the refresh cycle has begun, $\overline{\text{GI}}$ is not sampled until it completes. LRDY and BUSFLT are not sampled until LCLK2's low-to-high transition after $\overline{\text{RAS}}$ has gone low.

If a refresh cycle is aborted because of a retry, then the count of refreshes pending is not decremented and the same pseudo-address is reissued when the refresh is restarted.

Figure 8–18. Refresh Cycle Timing



8.14 Local-Memory Cycles with Wait States

All other local-memory cycle timing diagrams throughout this chapter assume that LRDY is sampled high during the cycle and that there are no wait states. A slower memory that requires a longer cycle time may pull the LRDY pin low. The TMS34020 samples the LRDY input on the low-to-high transition of LCLK2 after $\overline{RAS}$ goes low, as indicated in the illustrations. If LRDY is low, the TMS34020 inserts an additional state, called a **wait state**, into the cycle by maintaining all of the signals output to memory at the same level. Wait states continue to be inserted until LRDY is sampled at a high level.

8.14.1 Adding Wait States in Read and Write Cycles

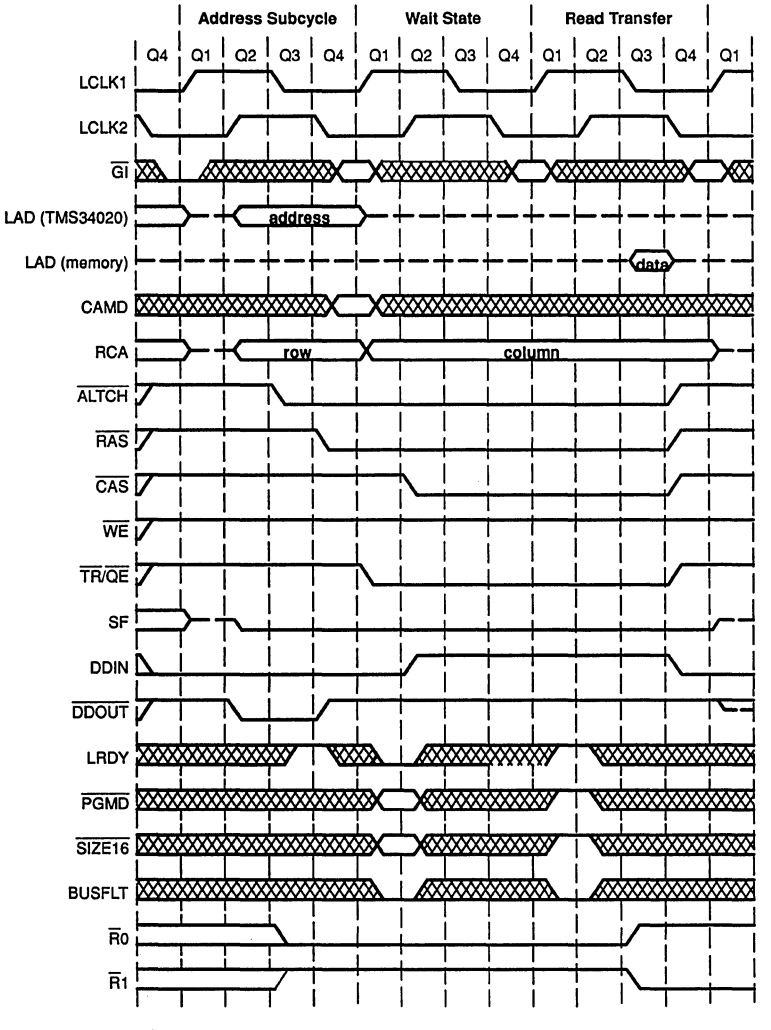
Figure 8–19 shows an example of a local-memory read cycle extended by 1 wait state. The first time LRDY is sampled, the TMS34020 detects a low level, causing the cycle to be extended by 1 wait state. When LRDY is sampled one LCLK period later, the TMS34020 detects a high level, permitting the cycle to complete.

All local-memory read and write cycles (including the special write-mask and block-write cycles for VRAMs) are extended in the same way as Figure 8–19. VRAM serial-register transfers are extended slightly differently.

Note:

$\overline{PGMD}$, $\overline{SIZE16}$, and BUSFLT must be held at a valid level at the beginning of each Q2 until LRDY is sampled high. The levels of $\overline{PGMD}$ and $\overline{SIZE16}$ at the time LRDY is sampled high will affect the subsequent cycle (allowing page mode or dynamic bus sizing).

Figure 8–19. Local-Memory Read Cycle with 1 Wait State

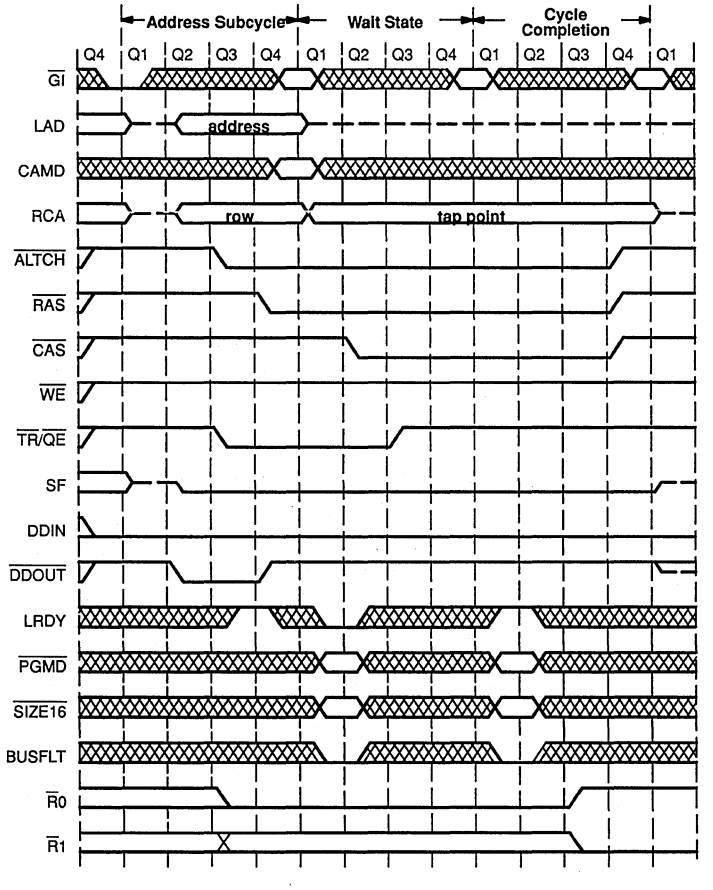


Key: **LAD (TMS34020):** The TMS34020 outputs this to the LAD bus.
LAD (memory): Memory outputs this to the LAD bus.

8.14.2 Adding Wait States in VRAM Serial-Register Transfers

Figure 8–20 shows an example of a memory-to-serial-data-register transfer cycle extended by 1 wait state. The wait state is inserted in exactly the same way as for other local-memory cycles. During a serial register transfer, $\overline{TR}/\overline{QE}$ normally makes a low-to-high transition 1/4 cycle earlier than the other local-memory control signals. When a wait state is inserted, $\overline{TR}/\overline{QE}$ is not kept low during the wait state. All serial-register transfers exhibit this behavior.

Figure 8–20. Memory-to-Serial-Data-Register Cycle with Wait State (VRAM Read Transfer)

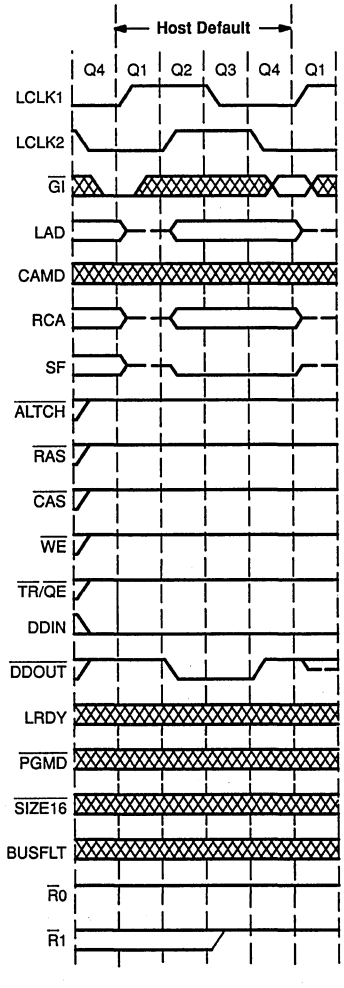


8.15 The Host-Default Local-Memory Cycle

When no other local-memory requests are pending, the memory controller executes a special idle cycle known as the **host-default cycle**. Th memory controller repeats the host-default cycle until another memory request occurs.

The host-default cycle is similar to the address subcycle of a host-initiated access, except that $\overline{\text{ALTCH}}$ and $\overline{\text{RAS}}$ are not activated. Figure 8–21 shows a host-default cycle. During this cycle, the host access status code is output on LAD0—LAD3. The address output on the remainder of the LAD bus and the RCA bus comes from the internal register used to store the address provided by the host for a host access.

Figure 8–21. The Host-Default Cycle



The address and status code output are normally of no significance, because $\overline{\text{ALTCH}}$ and $\overline{\text{RAS}}$ are not activated. The reason they are output at all is because

the host-default cycle provides a mechanism for reducing the time taken to respond to a host request when the local-memory interface is idle.

If the TMS34020 synchronizes a host request while performing the host-default cycle, internal logic converts the cycle into an ordinary host access (by asserting $\overline{\text{ALTCH}}$ and $\overline{\text{RAS}}$ at the appropriate times). This is similar to any other host access. If this happens, the address output at the beginning of the host-default cycle will be the valid address for the access.

8.16 Addressing Mechanisms

The TMS34020 uses a 32-bit logical address that points to a bit in memory. The logical address bits are numbered from 0 to 31, where bit 0 is the LSB and bit 31 is the MSB. The TMS34020 uses logical address bits 0—4 to specify the bit address for a particular access but does not output these bits. Thus, each of the TMS34020's memory accesses are initially aligned to a 32-bit boundary.

A 16-bit word address pointer (S) is output in place of logical address bit 4. This bit is a 1 only when you are using the dynamic bus sizing capability of the TMS34020 to access the second of the two 16-bit memory locations needed to contain all the data specified by the logical long-word (32-bit) address. Section 8.9 (page 8-25) describes dynamic bus sizing in detail.

During each local-memory cycle, the TMS34020 outputs the long-word address and 16-bit word select in two different formats: nonmultiplexed and multiplexed.

8.16.1 Nonmultiplexed Addressing

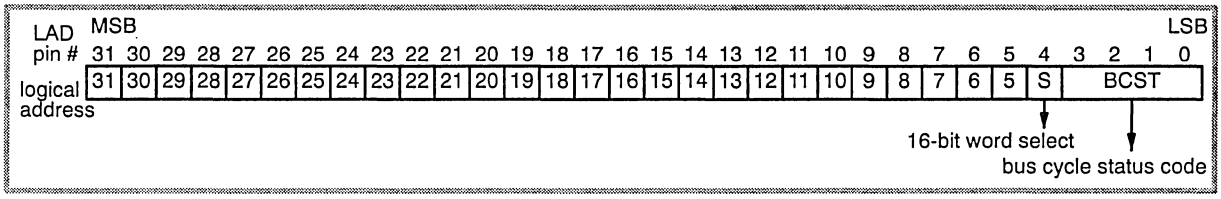
During an address/status subcycle, the following information is output on the LAD bus.

- full logical long-word address on LAD5—LAD31
- 16-bit word select (S) on LAD4
- local-memory cycle status code on LAD0—LAD3

Figure 8–22 shows this. The high-to-low transition of the address latch signal ($\overline{\text{ALTCH}}$) can be used to hold this information in external latches (such as the 74ALS573) during the entire memory cycle for memory devices or decode logic that require a static address.

For accesses to 32-bit memories, the S bit on LAD4 is always 0. S=1 only if the TMS34020 accesses a 16-bit memory that does not support page mode. In this case, separate local-memory cycles (consisting of both address/status and data subcycles) must be performed for each 16-bit location within the long-word being accessed. S=1 for the second of these accesses. Sections 8.7 and 8.9 (pages 8-15 and 8-25) discuss page mode and dynamic bus sizing, respectively.

Figure 8–22. Logical Address Output on LAD



8.16.2 Multiplexed Addressing

The TMS34020 provides direct support for DRAMs and VRAMs, which expect an address' row and column parts to be provided sequentially, time-multiplexed over the same wires. The 13-bit RCA bus is used for this purpose. For the sake of brevity, this section uses the word *DRAM* to refer to DRAM, VRAM, or any other device that expects multiplexed row and column addresses.

During an address/status subcycle, a subset of the logical long-word address output on the LAD bus is output simultaneously on the RCA bus. This serves as the row address for the DRAMs. During a data subcycle, a different subset of the logical address is output on the RCA bus. This serves as the column address for the DRAMs. The 16-bit word select (S) is also output on the RCA bus during the data subcycle.

The RCA bus is designed so that DRAMs from 64K×*n* to 16M×*n* can be connected directly, without external multiplexing hardware. (*n* represents the number of 1-bit memories of the base size integrated onto a single chip. For example, 1-Mbit VRAMs are implemented as four 256K memories within a single chip; 256K×4.) To do this, the following requirements must be met:

- All the row address bits for the DRAM must be available during the address/status subcycle, and then all the column address bits must be available during the data subcycle *on exactly the same RCA pins*.
- All the logical address bits that comprise the row and column addresses must form a continuous field. That is, all the logical address bits between the lowest and the highest connected to the DRAMs must also be connected. If this is not the case, the memory will not be fully decoded.
- Logical address bits that are connected to the DRAMs at row-address time should not be connected to the DRAMs at column-address time.

Because the different DRAM sizes require different numbers of address lines (from 8 for 64K×*n* up to 12 for 16M×*n*), this can be achieved only by providing a number of programmable choices for the subsets of the logical address output at row-address time and at column-address time. Two mechanisms are provided to do this:

- RCM[[CONFIG]] determines the address subset output at row-address time. The RCM bits are loaded from bits 1 and 2 of the reset vector (address FFFF FFE0h; Section 6.12.4, page 6-26, discusses this).

- The address subset output at column-address time is determined dynamically on a cycle-by-cycle basis by the CAMD pin. This allows DRAMs of different array sizes to coexist within the same system, still without the need for external multiplexing hardware.

RCM[[CONFIG]] determines the DRAM base array size for the application. This is the DRAM array size supported when CAMD=0. It is determined by the displacement between the logical address bit output on any given RCA pin at row-address time and the logical address bit output on the same pin at column-address time. For instance, when the base array size=64K×*n*, if the logical address bit output at row-address time on a given RCA pin is bit *m*, then the logical address bit output on the same pin at column-address time is bit *m*–8. For 256K×*n* base array size, it is bit *m*–9, and so on. Base DRAM array sizes supported are 64K×*n*, 256K×*n*, 1M×*n*, and 4M×*n*.

The TMS34020 samples the data on the CAMD pin on LCLK1's low-to-high transition at the end of the address/status subcycle. This allows you to decode the full logical address output on the LAD bus at this time (or the row address output on the RCA bus) with external logic to determine what the array size of the memory being addressed is, and thus what the column-address mode should be for the particular access. Table 8–10 shows all the possible DRAM array sizes that are supported by the various combinations of the RCM bits and the CAMD pin.

Table 8–10. DRAM Array Sizes

RCM		Base Array	Additional Arrays
1	0	(CAMD=0)	Supported by CAMD=1
0	0	64K×*n*	256K×*n*, 1M×*n*
0	1	256K×*n*	1M×*n*, 4M×*n*
1	0	1M×*n*	4M×*n*
1	1	4M×*n*	16M×*n* (32 bits wide only)

Note: 16M×*n* memory can be addressed only as 32-bit-wide memory; dynamic bus sizing cannot be used, because there is no way to incorporate the S bit into the address connected to DRAMs of this size.

- **When CAMD=0,** the logical address bits output on the RCA bus at column-address time allow DRAMs of the base array size to be connected.
- **When CAMD=1,** a different logical address mapping is generated at column-address time. For most of the RCA pins, the logical address bits output at column-address time when CAMD=0 are output 1 RCA pin higher when CAMD=1. For example, the logical address bit output on RCA6 when CAMD=0 is output on RCA7 when CAMD=1. This increases the displacement between the logical address bit output on a given RCA pin at row-address time and the logical address bit output on the same pin at column time by 1, thus allowing DRAMs with an array size 1 larger than the base array size specified by the RCM bits to be directly connected.

For some base array modes, RCA4, RCA11, and RCA12 may output logical address bits that are not determined by this CAMD=1 mapping:

64K×*n* The logical address bits output on RCA11 and RCA12 are not contiguous with the logical address bits output on RCA1—RCA10. This allows for 1M×*n* DRAMs to be connected without the same logical address bit appearing in both row and column addresses.

256K×*n* The logical address bit output on RCA12 is discontiguous with the logical address bits output on RCA1—RCA11. This allows 4M×*n* DRAMs to be connected without the same logical address bit appearing in both the row and column addresses.

4M×*n* The logical address bits output on RCA12 and RCA4 allow 16M×*n* DRAMs to be connected without the same logical address bit appearing in both the row and column addresses.

All except 4M×*n*
The S bit is always mapped to RCA4.

Table 8–11 lists the actual logical address bits output on the RCA bus for each of the base DRAM array sizes, with both states of CAMD.

Table 8–11. Logical Addresses Output on the RCA Bus

RCM		Base	Address		RCA Bus												
1	0	Array	Time	CAMD	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	64K×*n*	Row	—	24	23	22	21	20	19	18	17	16	15	14	13	12
		Δ=8	Column	0	16	15	14	13	12	11	10	9	8	7	6	5	S
			Column	1	23	22	13	12	11	10	9	8	7	6	5	S	S
0	1	256K×*n*	Row	—	25	24	23	22	21	20	19	18	17	16	15	14	13
		Δ=9	Column	0	16	15	14	13	12	11	10	9	8	7	6	5	S
			Column	1	26	14	13	12	11	10	9	8	7	6	5	S	S
1	0	1M×*n*	Row	—	26	25	24	23	22	21	20	19	18	17	16	15	14
		Δ=10	Column	0	16	15	14	13	12	11	10	9	8	7	6	5	S
			Column	1	15	14	13	12	11	10	9	8	7	6	5	S	S
1	1	4M×*n*	Row	—	27	26	25	24	23	22	21	20	19	18	17	16	15
		Δ=11	Column	0	16	15	14	13	12	11	10	9	8	7	6	5	S
			Column	1	28	14	13	12	11	10	9	8	7	6	5	S	16

Key: Δ is the displacement between the logical address bits output at row-address time and column-address time on a given pin, for CAMD=0.
n is the number of 1-bit memories for each of the base array sizes, integrated within a single chip.
S is the 16-bit word select.

During page-mode memory cycles (described in detail in Section 8.7, page 8-15), the data subcycle is repeated. Each time access to a new location is required, the address output on the RCA bus at this time is incremented or decremented.

8.16.3 Display Memory Requirements for Multiplexed Addressing

The TMS34020's RCA bus makes provision for many different multiplexed addressing schemes. However, in a graphics based system, there are some additional considerations that should be taken into account regarding the display memory.

When addressing the display memory (the memory area that stores the graphics image output to the screen), the logical address bits connected to the VRAMs must be sequential. That is, the least significant logical address bit is the LSB of the address presented to the VRAMs, the next least significant logical address bit is the next LSB of the address presented to the VRAMs, and so on up to the most significant logical address bit, which is the MSB of the address presented to the VRAMs.

If this is not the case, pixels with sequential addresses are not stored in sequential locations within the VRAM. This means pixels do not appear on the screen in the order they appear in the address map. Because of this requirement, the display memory should not be implemented with

- 1M×*n* VRAMs when the base DRAM array size is 64K×*n*
- 4M×*n* VRAMs when the base DRAM array size is 256K×*n*
- 16M×*n* VRAMs when the base DRAM array size is 4M×*n*

When using DRAMs to store data that is not to be output to the screen, the order in which the data is actually stored in the memory is unimportant; thus, these considerations do not apply.

8.16.4 Example Connections for Multiplexed Addressing

Table 8–12 provides example wiring configurations for DRAMs of different array sizes. This table assumes that any bank selecting is done with high-order logical-address bits so that the banks are not interleaved. This means that the DRAMs are wired to the lowest RCA pins possible.

In all modes except 4M×*n* with CAMD=1, connections are shown for memory arranged as 32 bits wide and 16 bits wide. The latter requires the use of the TMS34020's dynamic bus sizing capability. You can't connect16-bit-wide 16M×*n* memory; if the S bit is connected to one of the memory's address pins, logical address bit 15 cannot be connected. This means that the data within the DRAM appears at 2 logical addresses (one for each of the values of logical address bit 15).

Table 8–12. Example Connections to the RCA Bus

Base Array	CAMD	DRAM Array	RCA Bus Wiring												
			12	11	10	9	8	7	6	5	4	3	2	1	0
64K×*n*	0	64K (16)						a7	a6	a5	a4	a3	a2	a1	a0
		64K (32)					a7	a6	a5	a4	a3	a2	a1	a0	
	1	256K (16)				a8	a7	a6	a5	a4	a3	a2	a1	a0	
		256K (32)			a8	a7	a6	a5	a4	a3	a2	a1	a0		
		1M (16)		a9		a8	a7	a6	a5	a4	a3	a2	a1	a0	
		1M (32)	a9		a8	a7	a6	a5	a4	a3	a2	a1	a0		
256K×*n*	0	256K (16)					a8	a7	a6	a5	a4	a3	a2	a1	a0
		256K (32)				a8	a7	a6	a5	a4	a3	a2	a1	a9	
	1	1M (16)			a9	a8	a7	a6	a5	a4	a3	a2	a1	a0	
		1M (32)		a9	a8	a7	a6	a5	a4	a3	a2	a1	a0		
		4M (32)	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0		
1M×*n*	0	1M (16)				a9	a8	a7	a6	a5	a4	a3	a2	a1	a0
		1M (32)			a9	a8	a7	a6	a5	a4	a3	a2	a1	a0	
	1	4M (16)		a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0	
		4M (32)	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0		
4M×*n*	0	4M (16)			a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0
		4M (32)		a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0	
	1	16M (32)	a11	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1		a0

Note: a0—a11 are the DRAM address pins (a0 is the least significant).

8.16.5 Memory Organization and Bank Selecting

System memory is typically partitioned into several banks. Each bank contains the number of memory devices that can be accessed in a single memory cycle. Thus, the number of memory devices per bank is determined by dividing the data-bus width by the memory-device width. The TMS34020's data bus can access 16 or 32 bits per cycle (depending on dynamic bus sizing). Therefore, a bank composed of 1M×1 RAMs contains 16 or 32 RAM devices. Using 256K×4 RAMs requires either 4 or 8 RAM devices.

Logical address bits not used to form the row and column addresses of the DRAMs can be used to select between banks. These bits can be decoded with external logic, so that only the $\overline{\text{RAS}}$ or $\overline{\text{CAS}}$ strobe to the appropriate memory bank is enabled. This can be achieved in two ways:

- *The DRAM's address pins can be connected to the RCA bus' lower end.* Use higher order logical address bits to select between different banks; prevent activation of the unselected banks' $\overline{\text{RAS}}$ pins. These can be decoded from either the LAD or RCA bus during the address/status sub-cycle. In this case, the long-words within a given bank are contiguous. For instance, if each bank contains m words, then one bank may contain words 0 to m–1, the next bank words m to $2m$–1, and so on.

- *The DRAM's address pins can be connected to the RCA bus' upper end.* Use lower order logical address bits to select between different banks; prevent activation of the unselected banks' $\overline{CAS}$ pins. These can be decoded from the RCA bus during the data subcycle. In this case, the long-words within each bank are interleaved—if there are 2 banks, all odd words are in one bank, and all even words in another; each bank contains alternating words. If there are 4 banks, each bank contains every fourth word, etc.

Although you can use either form of bank selecting (or a combination of the two), be aware that interleaved banks do not easily support the use of the special block-write features available on some VRAMs and supported by the VBLT and VFILL instructions. This is because the VRAM block write allows 4 long-words to be written to simultaneously. If you use interleaved banking, these 4 words will not all be contained within the same memory bank. However, because the VRAMs ignore the 2 LSBs of the column address when a block-write cycle is performed, it is not possible to write less than 4 long-words at a time to a given bank.

In a typical graphics system, the local memory is divided into two parts:

- *display memory*
- *system memory* (additional DRAMs needed to store programs and data)

A high-order address bit is typically used to select between the two. Within each part, other address bits can be used to select particular banks.

The decode logic must be capable of more than just selecting a particular bank of the display memory or system memory during a memory read or write cycle. It must also be able to enable all DRAMs and VRAMs during a DRAM-refresh cycle and to enable the appropriate VRAMs during a serial-register transfer cycle. The decode logic must then distinguish DRAM-refresh and serial-register transfer cycles from memory-access cycles. This is done by decoding the 4 LSBs of the address output on the LAD bus to determine the current bus status.

8.16.6 Display Memory Hardware Requirements

The minimum number of memory bits required to implement the display memory is the product of the total number of pixels (on-screen and off-screen areas combined) and the number of bits per pixel. The minimum number of VRAMs required to contain the display memory is calculated as follows:

$$\textit{number of VRAMS} = \frac{(\textit{pixels per line}) \times (\textit{lines per frame}) \times (\textit{bits per pixel})}{\textit{number of bits per VRAM}}$$

This calculation yields the minimum number of VRAMs needed; however, some applications may require additional VRAMs. For example, the TMS34020 supports XY addressing most efficiently when the number of pixels per line of the display memory is a power of 2. Achieving this may require more than the minimum number of VRAMs needed to contain the display.

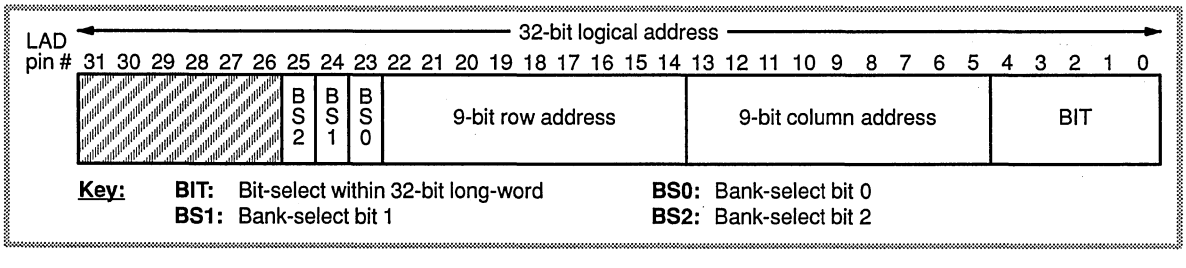
8.17 Double-Buffered Display Example (2×1280×1024)

As an example, consider a display system that implements a 1280×1024 double-buffered screen at 8 bits per pixel, with an additional 4 Mbytes of system memory.

The TMS34020's 32-bit data path allows four 8-bit pixels to be accessed simultaneously. This also means that 4 pixels are clocked out of the VRAMs' serial registers simultaneously. If the display is refreshed 60 times per second, pixels must be displayed at a rate of one every 12.7 ns. The VRAMs' serial registers must therefore be clocked once every 50.8 ns (that is, at a frequency of 19.6 MHz). This is well within the operating range of VRAMs.

The VRAM address decoding scheme shown in Figure 8–23 provides for these configuration requirements. Three logical address bits (23, 24, and 25) are used as bank-select bits. Logical address bits 5—13 are used as the 9-bit column address, and bits 14—22 are used as the 9-bit row address. The base VRAM array size should be set (via RCM[CONFIG]) to 256K×*n*. Referring to Table 8–11 (page 8-53), the row and column addresses are multiplexed out on the same 9 pins, RCA1—RCA10. The total number of address bits used to address external memory is 26, for a total address reach of 8 megabytes. The remaining 6 address bits output by the TMS34020 are not used for this example, but could be used for more system memory or any peripherals required for the system.

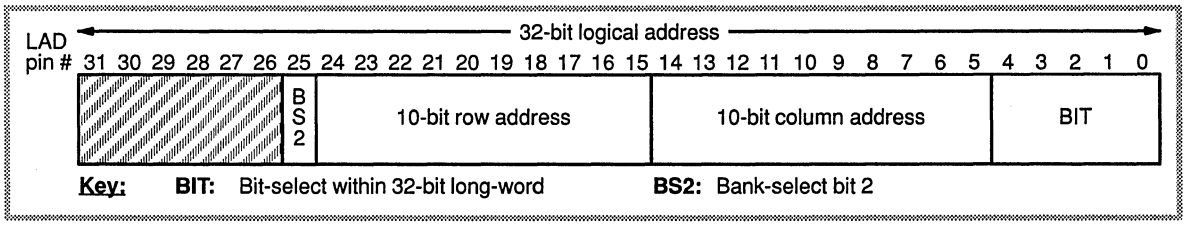
Figure 8–23. VRAM Address Decode for Example System



BS0 & BS1 (bank-select bits 0 and 1) select between the 2 buffers in the display memory.

BS2 (bank-select bit 2) selects between the display memory and the system memory. BS2=0 selects the display memory and BS2=1 selects the system memory. If the system memory is implemented with 1Mx1 DRAMs, the decode logic should assert CAMD=1 when BS2=1 to configure the RCA bus correctly. Figure 8–24 shows the address decode configuration for the system memory. Logical address bits 5—14 are used as the 10-bit column address, and bits 15—24 are used as the 10-bit row address. Referring to Table 8–11, the row and column addresses are multiplexed out over the same 10 pins, RCA2—RCA11.

Figure 8–24. DRAM Address Decode for Example System



The amount of VRAM required to implement the display memory in this system depends on whether midline reload is used to pack the data for each scan line into contiguous locations within the VRAM. The serial register within each VRAM is 512 bits long. Between them, the 32 serial registers hold 2048 pixels at any one time. However, each scan line requires only 1280 pixels. If midline reload is not used, each scan line must begin with a new row of VRAM, and 768 pixels from each row of memory cannot be displayed.

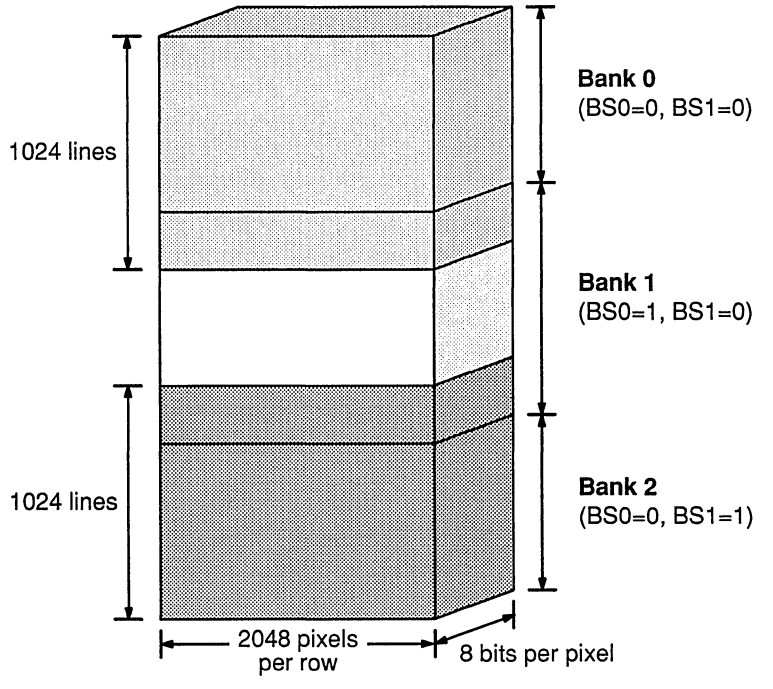
8.17.1 Display Memory Implementation Using Midline Reload

Midline reload allows all 2048 pixels in each row of VRAM to be displayed, because the VRAM row being clocked out can change in the middle of a scan line. Sections 9.13 to 9.15 (pages 9-42—9-51) discuss midline reload in detail. It is enabled by setting SSV[DPYCTL] to a 1. For this example, DPYMSK should be loaded with FFh (to indicate no bank-selects at the least significant end of the address, and a VRAM half serial register of 256 bits).

Figure 8–25 shows the structure of the display memory when midline reload is used. Note that only 3 banks are required to implement both display buffers. The total amount of display memory required is 24 Mbytes. BS0 and BS1 select between the banks. Display buffer 1 is spread across banks 0 and 1; display buffer 2 is spread across banks 1 and 2.

Because each buffer is spread across 2 banks, the bank from which pixels are being shifted must change part way down the screen. The changeover between banks must occur during a horizontal blanking period, when the VRAMs are not clocking pixels out of their serial registers. This means that the last pixel in a bank must also be the last pixel on a scan line. Each bank contains enough pixels for 4/5 of one display buffer (819.2 lines). The address of the first pixel in each display buffer must be such that the last pixel in banks 0 and 1 is also the last pixel on a line.

Figure 8–25. Example Display Memory Dimensions (with Midline Reload)



Key: Display buffer 1
Display buffer 2

Notes: 1) The pixels contained in each row are spread across the 2 scan lines.
2) The white area (~ 0.5 Mbytes) is not displayed.

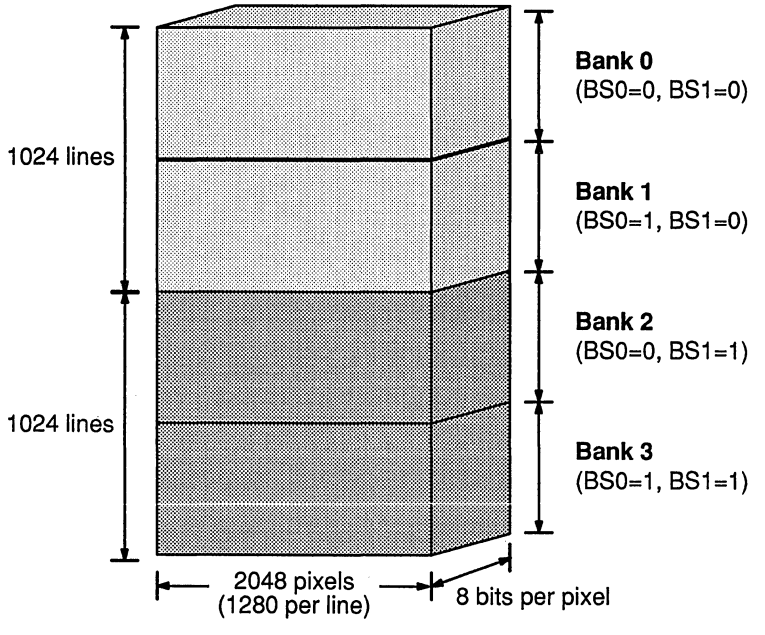
8.17.2 Display Memory Implementation Without Midline Reload

When midline reload is not used, all the pixel data for any single scan line must be contained within a single row of VRAM. Each VRAM row contains 2048 pixels, but only 1280 of these are required for a scan line. This means that 40% of the display memory is not output to the display (unless you wish to be able to pan the display around within the display buffer).

Figure 8–26 shows the structure of the display memory when midline reload is not used. In this case, 4 banks are required to implement both display buffers because each display buffer requires 1 row of VRAM for each of its 1024 lines. The total amount of display memory required is 32 Mbytes. BS0 and BS1 select between the banks. BS1 selects between the display buffers.

Unfortunately, the 12.8 Mbytes of memory not used for the display is not addressable as a contiguous block. This makes it very difficult to use for program storage, etc.

Figure 8–26. Example Display Memory Dimensions (Without Midline Reload)



Key: Display buffer 1

Display buffer 2

Note: Each display buffer can contain 2048 pixels per line, but only 1280 are output to the VRAMs. The undisplayed 2/5 of all rows constitutes a total of 12.8 Mbytes of undisplayed memory.

Chapter 9

Video Timing and Screen Refresh

The TMS34020's video interface provides these features:

- Separate or composite sync and blanking
- Synchronization to internal or external signals
- Interlaced or noninterlaced video
- A variety of screen resolutions

Additionally, the TMS34020 directly supports the use of VRAMs by generating the memory-to-register cycles needed to refresh a screen. The TMS34020 can also control video capture by performing register-to-memory (instead of memory-to-register) cycles.

This chapter includes the following topics:

9.1 Related Signals

The TMS34020's video timing logic is driven by an external video clock; the TMS34020 generates sync and blanking signals on chip. These signals control the horizontal and vertical sweep rates of the screen and synchronize the screen display to data output by the VRAMs. The video-timing and screen-refresh signals are summarized below for your convenience; for detailed descriptions, refer to Chapter 2. Note that these signals depend on information that is provided in the DPYCTL register.

Signals	Descriptions	I/O
$\overline{\text{CBLNK}}$/ $\overline{\text{VBLNK}}$	can be either a composite-blanking signal or a vertical-blanking signal, depending on the value of CVD:	
	CVD=0 selects $\overline{\text{CBLNK}}$, which is the composite-blanking signal that turns off a CRT's electron beam during both horizontal and vertical retrace intervals. You can also use $\overline{\text{CBLNK}}$ to control stopping and starting of the VRAM serial registers.	O
	CVD=1 selects $\overline{\text{VBLNK}}$, which is the vertical-blanking signal that turns off a CRT's electron beam during vertical retrace intervals. You can also use $\overline{\text{VBLNK}}$ with $\overline{\text{HBLNK}}$ to control stopping and starting of the VRAM serial registers.	O
	Both signals are always outputs.	
$\overline{\text{CSYNC}}$/ $\overline{\text{HBLNK}}$	can be either a composite-sync signal or a horizontal-blanking signal, depending on the value of CVD:	
	CVD=0 selects $\overline{\text{CSYNC}}$, which is the composite-sync signal that controls external video circuitry. $\overline{\text{CSYNC}}$ can be an input or an output, depending on the value of CSD:	
	CSD=0 selects external composite sync; $\overline{\text{CSYNC}}$ is an input	I
	CSD=1 $\overline{\text{CSYNC}}$ is an output	O
	CVD=1 selects $\overline{\text{HBLNK}}$, which is the horizontal-blanking signal that turns off a CRT's electron beam during horizontal retrace intervals. $\overline{\text{HBLNK}}$ is always an output, regardless of the value of CSD (however, to ensure correct operation, you should always set CSD to 1 when CVD=1).	O
	$\overline{\text{HBLNK}}$ is always an output, but $\overline{\text{CSYNC}}$ can be an input or an output.	

Signals	Descriptions	I/O
$\overline{\text{HSYNC}}$	is the horizontal-sync signal that controls external video circuitry. $\overline{\text{HSYNC}}$ can be an input or an output, depending on the value of HSD:	
	HSD=0 selects external horizontal sync; $\overline{\text{HSYNC}}$ is an input	I
	HSD=1 $\overline{\text{HSYNC}}$ is an output	O
SCLK	is the VRAM serial-register clock. When using midline reload, this input is used to track the VRAM tap point. SCLK should be low during horizontal and vertical blanking. If you are not using midline reload, SCLK should be low.	I
VCLK	is the video clock signal that drives the TMS34020's internal video timing logic. VCLK is derived from the dot clock of the external video system.	I
$\overline{\text{VSYNC}}$	is the vertical-sync signal that controls external video circuitry. $\overline{\text{VSYNC}}$ can be an input or an output, depending on the value of VSD:	
	VSD=0 selects external vertical sync; $\overline{\text{VSYNC}}$ is an input	I
	VSD=1 $\overline{\text{VSYNC}}$ is an output	O

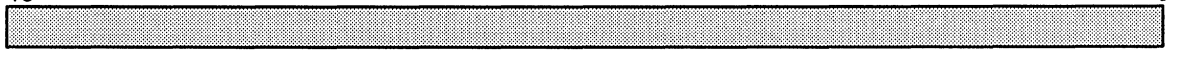
Holding VCLK low for long periods may cause video counter errors. When VCLK is not being clocked for long periods, hold it at the logic-high level. While VCLK is low, the storage nodes within the device rely on their internal capacitance to maintain state information; if VCLK is held low for a sufficiently long time, charge leakage may cause bit errors.

9.2 Related Registers

The video timing and screen-refresh registers are a subset of the I/O registers described in Chapter 4. These registers are divided into 3 groups:

- **Horizontal timing registers** control the timing of the $\overline{\text{HSYNC}}$ signal, the $\overline{\text{HBLNK}}$ signal, and the horizontal components of the $\overline{\text{CSYNC}}$ and $\overline{\text{CBLNK}}$ composite signals.
- **Vertical timing registers** control the timing of the $\overline{\text{VSYNC}}$ signal, the $\overline{\text{VBLNK}}$ signal, and the vertical components of the $\overline{\text{CSYNC}}$ and $\overline{\text{CBLNK}}$ composite signals.
- **Screen-refresh registers** control the addresses generated during local-memory screen-refresh (memory-to-register) cycles.

HCOUNT register	**address: C000 01D0h**
HESYNC register	**address: C000 0010h**
HESERR register	**address: C000 0270h**
HEBLNK register	**address: C000 0030h**
HSBLNK register	**address: C000 0050h**
HTOTAL register	**address: C000 0070h**
SETHCNT register	**address: C000 0310h**



HCOUNT tracks the number of VCLK periods that occur per horizontal scan line.

HESYNC defines the point in a horizontal scan line where horizontal sync ends and the $\overline{\text{HSYNC}}$ signal is driven inactive.

HESERR is used in composite mode only. It identifies the point in the horizontal scan line where serration ends. Serration may occur on the $\overline{\text{CSYNC}}$ pin during the vertical-sync portion of the display, and is similar in form to horizontal sync, though of different duration.

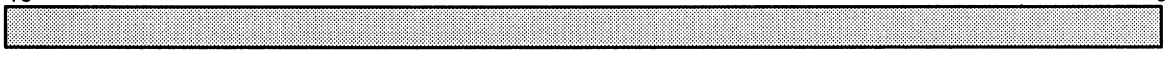
HEBLNK defines the endpoint of the horizontal-blanking period; this is the point when the $\overline{\text{HBLNK}}$ signal is driven inactive.

HSBLNK defines the startpoint of horizontal blanking, when $\overline{\text{HBLNK}}$ is driven active.

HTOTAL defines the number of VCLK periods allowed per horizontal line. It also specifies when the horizontal-sync period begins and $\overline{\text{HSYNC}}$ is activated.

SETHCNT is used in external video modes to reload HCOUNT when an external composite or horizontal sync begins. This allows you to start the horizontal count from an arbitrary value, counteracting synchronization delays and/or external signal skew.

VCOUNT register **address: C000 01C0h**
VESYNC register **address: C000 0000h**
VEBLNK register **address: C000 0020h**
VSBLNK register **address: C000 0040h**
VTOTAL register **address: C000 0060h**
SETVCNT register **address: C000 0300h**



VCOUNT counts the horizontal scan lines in the display.

VESYNC defines the number of horizontal scan lines where vertical sync ends and the $\overline{\text{VSYNC}}$ signal is driven inactive.

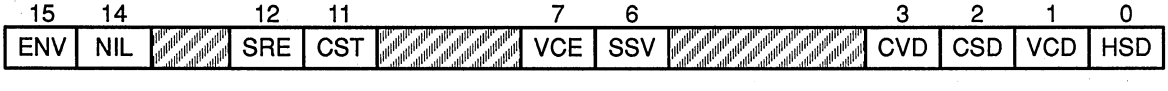
VEBLNK defines the endpoint of the vertical-blanking period; this is the point when the $\overline{\text{VBLNK}}$ signal is driven inactive.

VSBLNK defines the startpoint of vertical blanking, when $\overline{\text{VBLNK}}$ is driven active.

VTOTAL defines the number of horizontal scan lines allowed for the entire display. It also specifies when the vertical-sync period begins and $\overline{\text{VSYNC}}$ is activated.

SETVCNT is used in external noninterlaced video modes to reload VCOUNT when the first external composite serration pulse or vertical sync begins. This allows you to start the vertical count from an arbitrary value, counteracting external signal skew. For external interlaced video this register must be programmed to 0 so that the video timing logic can distinguish the external odd and even fields and synchronize accordingly.

DPYCTL register **address: C000 0080h**



Note: For a complete illustration and description of DPYCTL, see Chapter 4.

The DPYCTL register does not affect display *timing*, but it contains several bits that control various display functions:

HSD
bit 0

The HSD (horizontal-sync direction) bit determines whether $\overline{\text{HSYNC}}$ is an input or an output:

HSD=0 selects $\overline{\text{HSYNC}}$ as an input.

HSD=1 selects $\overline{\text{HSYNC}}$ as an output.

VSD bit 1

The VSD (vertical-sync direction) bit determines whether $\overline{\text{VSYNC}}$ is an input or an output:

VSD=0 selects $\overline{\text{VSYNC}}$ as an input.

VSD=1 selects $\overline{\text{VSYNC}}$ as an output.

CSD bit 2

The CSD (composite-sync direction) bit helps to determine whether $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ is an input or an output:

CSD=0 selects $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ as an output.

CSD=1 selects $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ as an input, provided that CVD=0 (if CVD=1, this pin is always an output).

CVD bit 3

The CVD (composite-video disable) bit determines the functions of $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ and $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$:

CVD=0 selects $\overline{\text{CSYNC}}$ and $\overline{\text{CBLNK}}$.

CVD=1 disables composite video and selects $\overline{\text{HBLNK}}$ and $\overline{\text{VBLNK}}$.

SSV bit 6

Setting the SSV (split-serial-register midline-reload enable) bit to 1 enables the TMS34020 to perform screen refreshes for VRAMs with split serial registers.

VCE bit 7

The VCE (video-capture enable) bit selects the type of screen-refresh memory cycle:

VCE=0 selects memory-to-register screen-refresh cycles.

VCE=1 selects register-to-memory screen-refresh cycles.

CST bit 11

Setting the CST (CPU serial-register transfer) bit to 1 tells the TMS34020 to convert ordinary pixel accesses into VRAM serial-register transfers.

SRE bit 12

Setting the SRE (screen-refresh enable) bit to 1 enables automatic screen refreshes when video is enabled (ENV=1).

NIL bit 14

The NIL (noninterlaced video) bit chooses interlaced or noninterlaced video.

NIL=0 enables interlaced video.

NIL=1 enables noninterlaced video.

ENV bit 15

The ENV (enable video) bit **must be 1** to allow the video timing logic to operate. When ENV=0, the TMS34020's blanking outputs are permanently at the active-low level, and all display control is disabled.

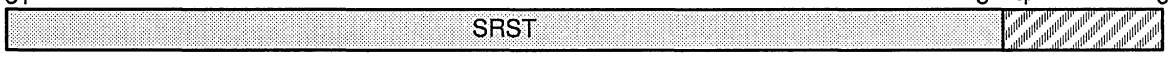
DXV bit 13

The TMS34010 used bit 13 as the DXV (disable external video) bit. The TMS34020 does not need this bit; the TMS34020 relies entirely on HSD, VSD, and CSD to individually select which video timing signals are external. The TMS34020 ignores bit 13.

ORG bit 10

The TMS34010 used bit 10 as the ORG (origin) bit to determine whether the display origin was in the top left or bottom left corner of the screen, and therefore whether to increment or decrement the screen-refresh address during horizontal blanking. The TMS34020 does not need this bit; the TMS34020 allows you to load SRINC[[DINC]] with the 2s complement of the display pitch if you wish to decrement the screen-refresh address. The TMS34020 ignores bit 10.

DPYST registers **address: C000 0200h**



Note: For a complete illustration and description of DPYST, see Chapter 4.

DPYSTL and **DPYSTH** are two 16-bit registers that work together to form a single 32-bit register. Throughout this chapter, this register pair is referred to as **DPYST**. DPYST contains the 27-bit SRST value, which represents the long-word address of the first pixel to be displayed on the screen.

DPYNX registers **address: C000 0220h**

Note: For a complete illustration and description of DPYNX, see Chapter 4.

DPYNXL and **DPYNXL** are two 16-bit registers that work together to form a single 32-bit register. Throughout this chapter, this register pair is referred to as **DPYNX**. The DPYNX registers contain two values.

SRNX
bits 5—31

The 27-bit SRNX value represents the long-word address of the next-screen refresh. This is the address of the first pixel on the next scan line of the display.

YZCNT
bits 0—4

The 5-bit YZCNT value is used to determine whether or not SRNX can be incremented between screen refreshes. This allows the image on the screen to be magnified (zoomed) in the Y direction by repeating scan lines multiple times.

DINC registers **address: C000 0240h**

Note: For a complete illustration and description of DINC, see Chapter 4.

DINCL and **DINCH** are two 16-bit registers that work together to form a single 32-bit register. Throughout this chapter, this register pair is referred to as **DINC**. The DINC registers contain two values.

SRINC
bits 5—31

The 27-bit SRINC value is the amount by which the address in SRNX is incremented between screen refreshes; this value is equal to the display pitch.

YZINC
bits 0—4

The 5-bit YZINC value is used to determine the number of screen refreshes that must occur between each increment of SRNX. YZINC is added to YZCNT

between every screen refresh, but SRNX is incremented only when YZCNT=0. This allows the screen image to be magnified (zoomed) in the Y direction by repeating scan lines multiple times.

DPYMSK register **address: C000 02E0h**

DPYMSK identifies the portion of the screen-refresh address that represents the column or tap-point address for midline reload. DPYMSK's LSB maps to the LSB (bit 5) of SRST[[DPYST]] and SRNX[[DPYNX]]. Allowing for this 5-bit offset, DPYMSK must be loaded with a string of contiguous 1s at the position of the column address in the logical address.

The TMS34020's screen-refresh registers differ from those of the TMS34010, although the registers are similar in function.

- SRST[[DPYST]] is equivalent to the SRSTRT field of the TMS34010's DPYSTRT register.
- SRNX[[DPYNX]] is equivalent to DPYADR.
- SRINC[[DINC]] is equivalent to the DUDATE field in the TMS34010's DPYCTL register.
- The TMS34020 does not need the TAPPNT register.

The TMS34020 supports the DPYSTRT, DPYADR, and TAPPNT registers for compatibility purposes only. They are simple read/write registers; they have no additional functionality and have no effect on the TMS34020's screen-refresh mechanisms.

There is no equivalent of the LCSTRT field in the TMS34010's DPYSTRT register. Screen-refresh cycles are generated during the horizontal-blanking interval at the beginning of every unblanked line of the display.

9.3 Relationship Between Horizontal and Vertical Timing Signals

Figure 9–1 illustrates the relationship between the horizontal and vertical timing signals for constructing a 2-dimensional raster display pattern.

- The **horizontal** sync and blanking signals span a single horizontal scan line within the frame and are repeated for each line.
- The **vertical** sync and blanking signals span an entire frame (1 complete pass of the display).

Figure 9–1. Horizontal and Vertical Timing Relationship

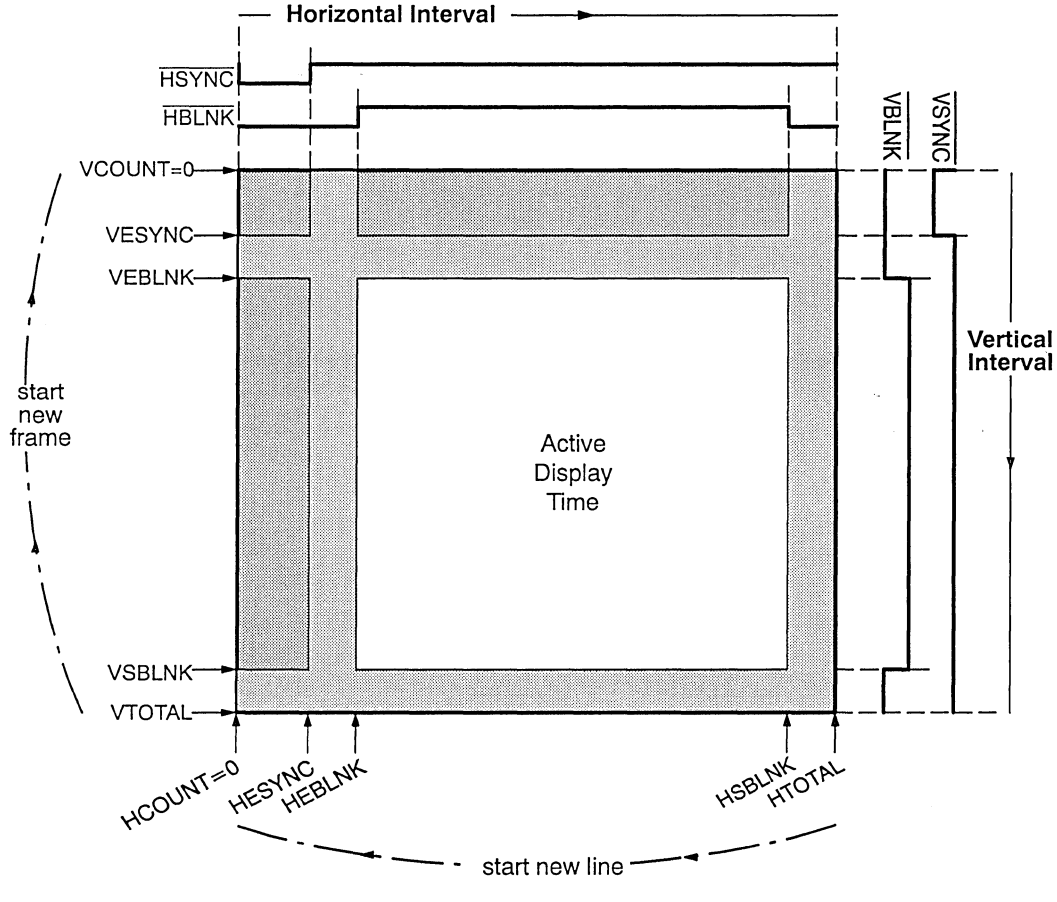
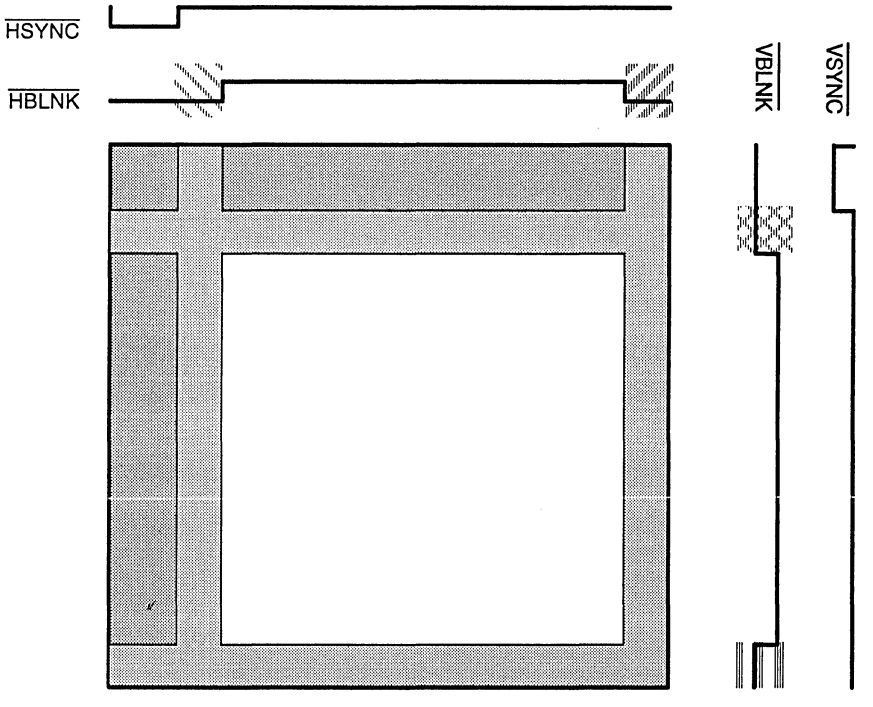


Figure 9–2, a simplified version of Figure 9–1, illustrates several terms and phrases used throughout this chapter.

Figure 9–2. The Porches

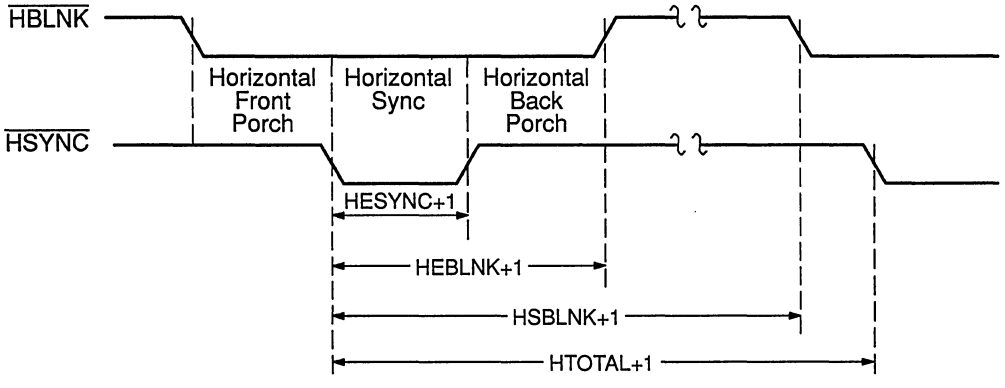


horizontal front porch	is the interval between the beginning of horizontal blanking and the beginning of horizontal sync.
horizontal back porch	is the interval between the end of horizontal sync and the end of horizontal blanking.
vertical front porch	is the interval between the beginning of vertical blanking and the beginning of vertical sync.
vertical back porch	is the interval between the end of vertical sync and the end of vertical blanking.

9.4 Horizontal Video Timing (Internal)

This discussion applies to video timing signals that the TMS34020 generates internally (it does not apply to external signals). Horizontal timing signals are the same for interlaced and noninterlaced video displays. Figure 9–3 shows how the HESYNC, HEBLNK, HSBLNK, and HTOTAL registers control horizontal signal timing.

Figure 9–3. Horizontal Timing



- All horizontal timing parameters are multiples of VCLK, which are counted by HCOUNT.
- The time between the start of 2 successive $\overline{\text{HSYNC}}$ pulses is specified by HTOTAL. Because HCOUNT counts 0 as its first value, the value in HTOTAL represents 1 less than the number of VCLK periods per horizontal scan line. For example, to specify a horizontal interval that is some even number $2n$ of VCLK periods in duration, the HTOTAL register must be set to an odd value, $2n–1$.
- Similarly, the value in HESYNC represents the duration of the $\overline{\text{HSYNC}}$ pulse, minus 1, and the values in $\overline{\text{HSBLNK}}$ and $\overline{\text{HEBLNK}}$ represent the startpoints and endpoints of the horizontal-blanking interval.

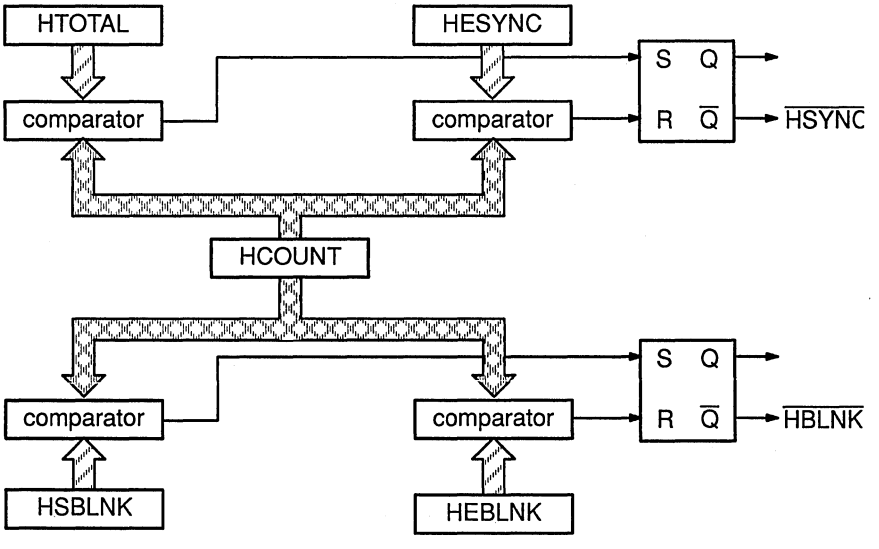
Figure 9–4 shows a simplified diagram of the internal logic that generates the horizontal timing signals. HCOUNT is incremented once every VCLK period (on the high-to-low transition) until it equals the value in HTOTAL. During the VCLK period following HCOUNT = HTOTAL, HCOUNT is reset to 0, and begins counting again.

The limits of the horizontal-sync pulse are defined by the values in HESYNC and HTOTAL. $\overline{\text{HSYNC}}$ is driven active low after HCOUNT = HTOTAL; it is then driven inactive high after HCOUNT = HESYNC. After HCOUNT becomes equal to HTOTAL or HESYNC, a 2-VCLK delay occurs before the appropriate transition takes place at the $\overline{\text{HSYNC}}$ pin.

The $\overline{\text{HBLNK}}$ signal is driven active low after HCOUNT = HSBLNK; it is then driven inactive high after HCOUNT = HEBLNK. After HCOUNT becomes

equal to HSBLNK or HEBLNK, there is a 2-VCLK delay before the appropriate transition takes place at the $\overline{\text{HBLNK}}$ or $\overline{\text{CBLNK}}$ pin (depending on the value of CSD[DPYCTL]).

Figure 9–4. Horizontal Timing Logic—Equivalent Circuit

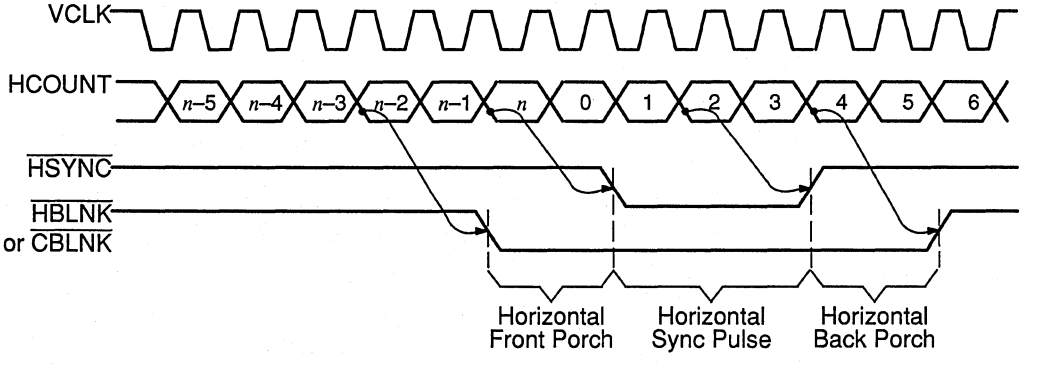


When HCOUNT= HSBLNK (shortly before the end of the horizontal scan), horizontal blanking begins. At this time, the video timing logic automatically schedules a screen-refresh memory cycle, preparing for the next displayed line.

Note:

For interlaced video, HSBLNK, HTOTAL/2, and HESYNC/2 must be separated from each other by a time of at least 2 VCLK periods, plus 2 LCLK periods, in order that the correct screen-refresh requests are detected by the TMS34020's memory controller.

Figure 9–5. Example of Horizontal Signal Generation

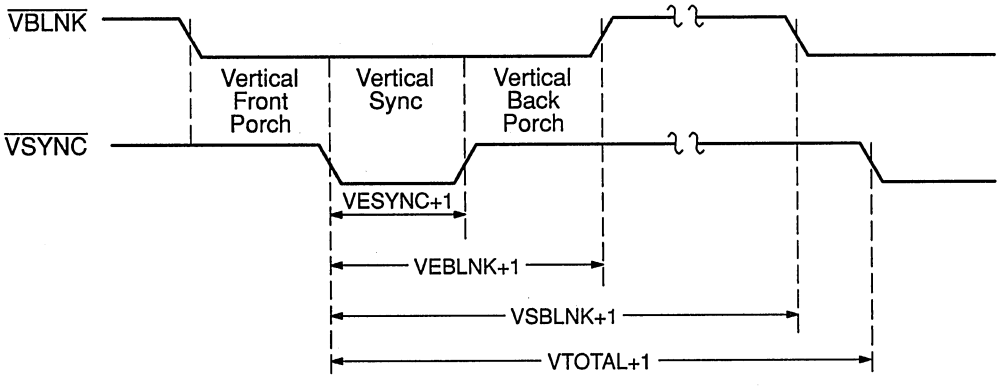


9.5 Vertical Video Timing (Internal)

This discussion applies to video timing signals that the TMS34020 generates internally (it does not apply to external signals). Some additional vertical timing characteristics that are unique to interlaced or noninterlaced video are described in Sections 9.7 (Noninterlaced Video Timing, page 9-18) and 9.8 (Interlaced Video Timing, page 9-21).

Figure 9–6 shows how the VESYNC, VEBLNK, VSBLNK, and VTOTAL registers control vertical signal timing.

Figure 9–6. Vertical Timing for Noninterlaced Display



For most of the display, VCOUNT increments once every horizontal scan line (when HCOUNT = HTOTAL). For interlaced displays, it also increments when HCOUNT = HTOTAL/2 during the equalization and serration regions of vertical blanking because these pulses occur twice per line. For this reason, the values programmed into the vertical video timing registers do not necessarily have a 1-to-1 relationship with the number of horizontal scan lines. Sections 9.7 and 9.8 discuss programming these registers.

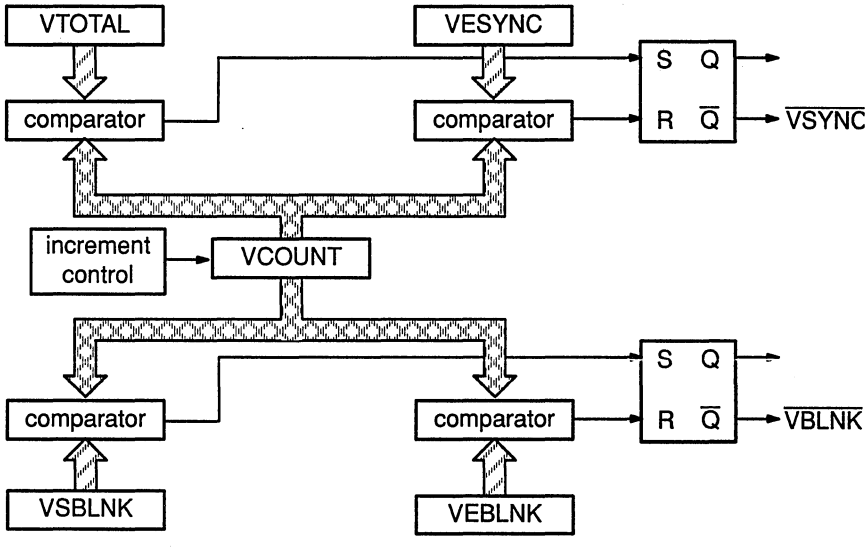
VTOTAL defines the time between the start of two successive $\overline{\text{VSYNC}}$ pulses. VESYNC represents the duration of the $\overline{\text{VSYNC}}$ pulse. $\overline{\text{VSBLNK}}$ and $\overline{\text{VEBLNK}}$ represent the startpoints and endpoints of the vertical-blanking interval.

Figure 9–7 shows a simplified schematic of the internal logic that generates the vertical timing signals. VCOUNT is incremented once every scan line (at the beginning of each $\overline{\text{HSYNC}}$ pulse, when HCOUNT = HTOTAL) and also at the midpoint of each scan line (when HCOUNT = HTOTAL/2) during the equalization and serration portions of an interlaced display, until it equals the value in VTOTAL. After VCOUNT = VTOTAL, VCOUNT is reset to 0, and begins counting again.

The limits of the vertical-sync pulse are defined by the values in VESYNC and VTOTAL. $\overline{\text{VSYNC}}$ is driven active low after VCOUNT = VTOTAL; it is then driven inactive high after VCOUNT = VESYNC. Transitions on the $\overline{\text{VSYNC}}$ pin coincide with transitions on the $\overline{\text{HSYNC}}$ or $\overline{\text{CSYNC}}$ pins.

The $\overline{\text{VBLNK}}$ signal is driven active low after VCOUNT = VSBLNK; it is then driven inactive high after VCOUNT = VEBLNK. Transitions on the $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$ pin coincide with transitions on the $\overline{\text{HSYNC}}$ or $\overline{\text{CSYNC}}$ pins.

Figure 9–7. Vertical Timing Logic—Equivalent Circuit



9.6 Composite Video Timing

This discussion applies to video timing signals that the TMS34020 generates internally (it does not apply to external signals).

Composite video signals combine the horizontal and vertical video timing into a single blanking signal, $\overline{\text{CBLNK}}$, and a single sync signal, $\overline{\text{CSYNC}}$.

- $\overline{\text{CBLNK}}$ is simply the logical-OR (negative logic) of the $\overline{\text{HBLNK}}$ and $\overline{\text{VBLNK}}$ signals. Thus, when either $\overline{\text{HBLNK}}$ or $\overline{\text{VBLNK}}$ is active low, $\overline{\text{CBLNK}}$ is active low.

- $\overline{\text{CSYNC}}$ is essentially the same as the $\overline{\text{HSYNC}}$ signal, except during portions of the vertical-blanking interval. During these portions, special sync pulses (known as *serration* and *equalization* pulses) are generated. These pulses allow a monitor to detect the vertical-sync interval, while at the same time ensuring that it remains in horizontal sync.

Some video monitors are designed to use composite rather than separate sync and blanking signals. Video systems used in television and broadcasting use composite video signals.

9.6.1 Theory Behind Serration and Equalization Pulses

Monitors pass the composite-sync signal through integrating (low pass) filters to extract the vertical-sync information from it. Serration pulses occur during the vertical-sync interval and are used by the monitor to determine when the vertical retrace occurs.

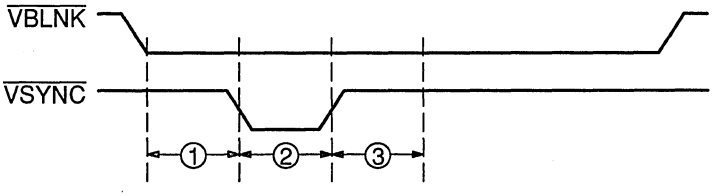
In noninterlaced video mode, serration pulses begin at the end of each scan line at the same time as the horizontal-sync pulses. In interlaced video mode, they also begin at the midpoint of each horizontal scan line. This is because in the even field, the vertical-sync interval begins and ends midway through a scan line. If the pulses occurred only once per scan line, the vertical-sync pulse extracted by the monitor would not start or end at the right time in the even field.

In interlaced video, equalization pulses occur in the regions immediately before and after the vertical-sync interval. They also begin at the end and the midpoint of each scan line, and because of this each pulse is only half the width of a regular horizontal-sync pulse. They are necessary to ensure that the monitor extracts a vertical-sync interval of exactly the same width in both the even and odd fields, because the vertical-sync interval begins at the midpoint of a scan line in the even field.

Because vertical sync always begins and ends at the end of a scan line in noninterlaced video mode, equalization pulses are not required.

Figure 9–8 shows the regions of vertical blanking in which the serration and equalization pulses occur. Each region contains a number of scan lines specified by the vertical video timing registers. Outside these regions, $\overline{\text{CSYNC}}$ is identical to $\overline{\text{HSYNC}}$.

Figure 9–8. Regions of Vertical Blanking Where Equalization and Serration Pulses Occur on $\overline{\text{CSYNC}}$



① The **first equalization region** coincides with the vertical front porch. In broadcast-quality composite-video standards such as NTSC and PAL, this region has the same duration as the vertical-sync pulse. However, you can reprogram the duration by varying the duration of the vertical front porch.

② The **serration region** coincides with vertical sync and immediately follows the first equalization region.

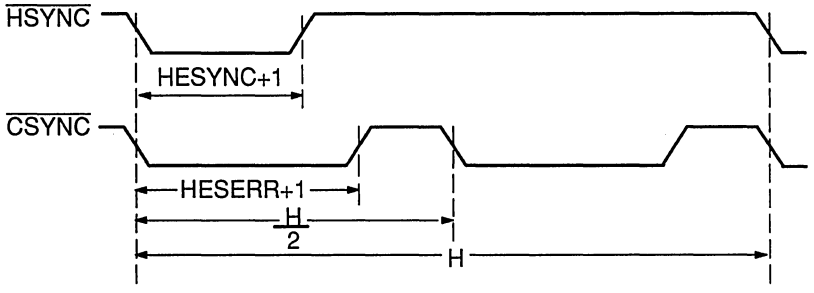
③ The **second equalization region** immediately follows the serration region. Hardware defines this region to have duration equal to that of vertical sync.

9.6.2 Serration Pulses on $\overline{\text{CSYNC}}$

Serration pulses are produced during the serration region (see Figure 9–8). Serration pulses begin at the end of each horizontal scan line (coinciding with the beginning of horizontal sync) after the condition HCOUNT = HTOTAL is reached. If the display is interlaced, they also begin at the midpoint of each scan line, after the condition HCOUNT = HTOTAL/2 is reached. The duration of these pulses is determined by the value of the HESERR register. This value is 1 less than the number of VCLK periods in the serration pulse, in the same way that the value in HESYNC is 1 less than the number of VCLK periods in the horizontal-sync pulse.

Serration pulses are generally longer in duration than regular horizontal-sync pulses. This is shown in Figure 9–9. Broadcast-quality composite-video standards such as NTSC and PAL require that serration pulses are of such a duration that $\overline{\text{CSYNC}}$ is inactive high for a period equal to the active-low time of $\overline{\text{HSYNC}}$. This can be achieved by programming HESERR to HTOTAL minus the number of VCLK periods in the horizontal-sync pulse.

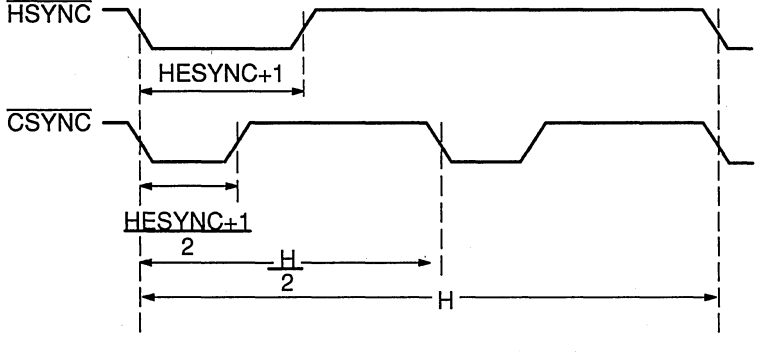
Figure 9–9. Composite Sync During Serration Region (Interlaced)



9.6.3 Equalization Pulses on CSYNC

If the display is interlaced, equalization pulses are produced during the equalization regions (see Figure 9–8, page 9-16). These pulses begin at both the end and the midpoint of each horizontal scan line, after HCOUNT = HTOTAL, and after HCOUNT = HTOTAL/2. Each pulse ends after HCOUNT = HESYNC/2. The beginning of every other pulse coincides with horizontal sync (see Figure 9–10).

Figure 9–10. Composite Sync During Equalization Regions (Interlaced)



Broadcast-quality composite-video standards such as NTSC and PAL require that equalization pulses are exactly half the duration of horizontal-sync pulses. To achieve this, set HESYNC to an odd value. HCOUNT starts counting from 0, so an odd value in HESYNC means that horizontal sync is an even number of VCLK cycles in length, and therefore exactly divisible by 2. Similarly, set HTOTAL to an odd value so that HTOTAL/2 is exactly the midpoint of each scan line.

If the display is noninterlaced, no special pulses are produced during the equalization regions. The CSYNC output continues to appear like HSYNC.

9.7 Noninterlaced Video Timing

You can select noninterlaced scan mode by setting NIL[[DPYCTL]] to 1.

DPYCTL register **address: C000 0080h**

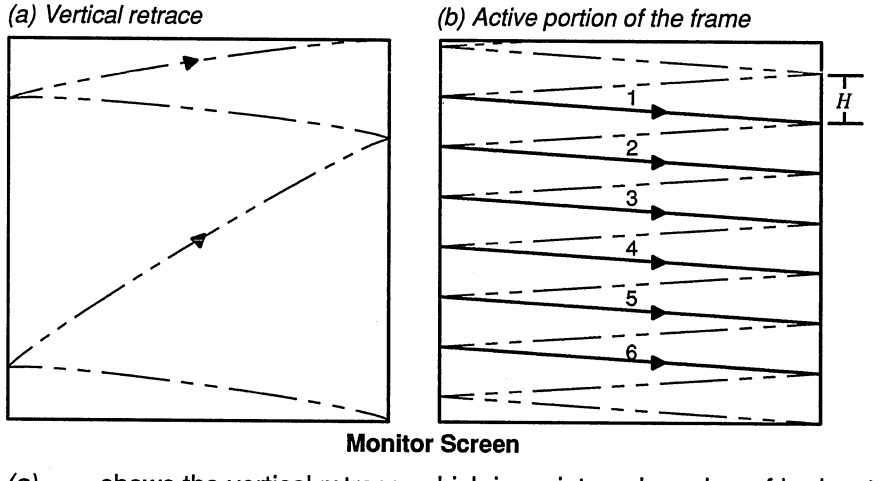
15	14	0
	NIL	
	1	

Note: For a complete illustration and description of DPYCTL, see Chapter 4.

9.7.1 Activity in Noninterlaced Mode

In noninterlaced mode, each video frame consists of a single vertical field. Figure 9–11 shows the path traced by the electron beam on the screen.

Figure 9–11. Electron Beam Pattern for Noninterlaced Video



(a) shows the vertical retrace, which is an integral number of horizontal scan lines in duration.

(b) shows the active portion of the frame. Solid lines represent lines that are displayed; dashed lines are blanked. Each line on the display is separated by the horizontal sweep time, H, where

$$H = (HTOTAL + 1) \times (VCLK\ period)$$

Figure 9–12 illustrates the video timing signals that generate the display shown in Figure 9–11. The display line numbers and regions from Figure 9–11 are shown for reference. For completeness, all 6 possible output signals are shown, although you can choose only 4 in either of the 2 allowed combinations:

Combination 1 (CVD=0):	Combination 2 (CVD=1):
$\overline{\text{HSYNC}}$	$\overline{\text{HSYNC}}$
$\overline{\text{VSYNC}}$	$\overline{\text{VSYNC}}$
$\overline{\text{CSYNC}}$	$\overline{\text{HBLNK}}$
$\overline{\text{CBLNK}}$	$\overline{\text{VBLNK}}$

The example in Figure 9–12 uses the following register values:

VSBLNK = 8	VTOTAL = 9
VESYNC = 3	VEBLNK = 2

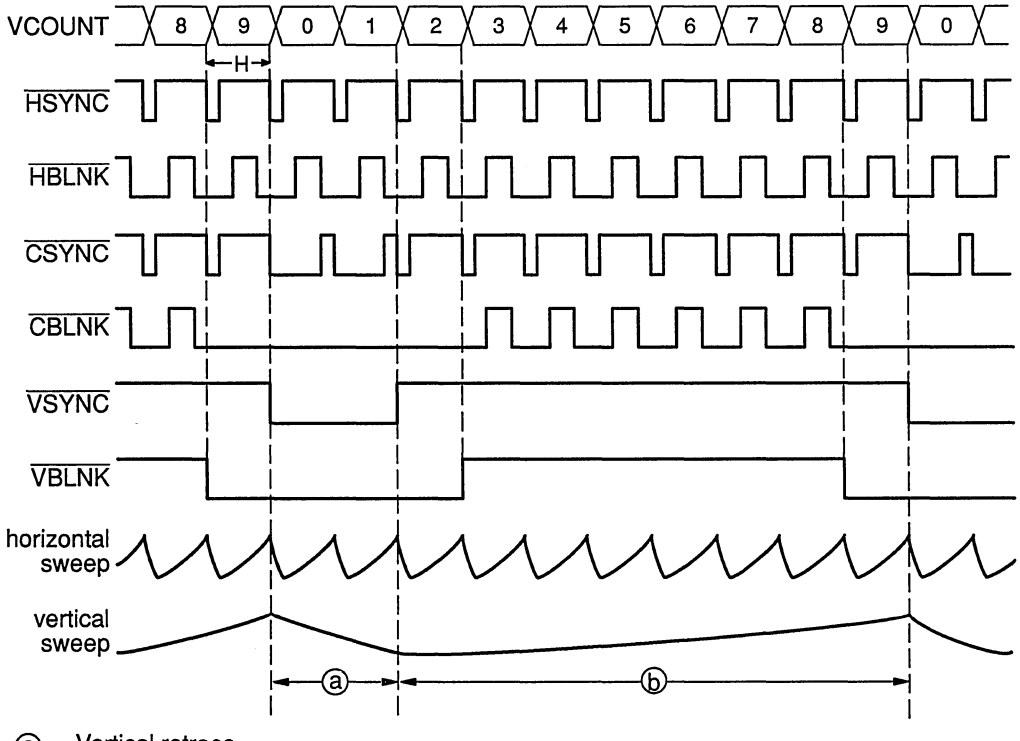
Actual applications use much larger values; these values are for illustration purposes only. Note that the VESYNC is not selected as you might expect (you would expect it to be 1). Section 9.7.2 describes the reason for this, along with formal equations for selecting proper values for all the vertical timing registers.

Each horizontal scan line is preceded by a horizontal retrace, triggered by the horizontal-sync pulse. The horizontal scan pattern repeats until VCOUNT = VTOTAL. Then, VCOUNT is reset to 0, and the vertical retrace (triggered by vertical-sync pulse) returns the beam to the top of the screen.

VCOUNT is incremented each time HCOUNT is reset to 0 (HCOUNT is reset to 0 at the end of each scan line). The $\overline{\text{VSYNC}}$ output goes active when VCOUNT = VTOTAL, coinciding with the beginning of the horizontal-sync pulse. The $\overline{\text{VSYNC}}$ signal goes inactive after VCOUNT = VESYNC; this transition also coincides with the beginning of the horizontal-sync pulse.

Horizontal blanking is typically active during vertical-blanking transitions, which coincide with the leading edges of horizontal-sync pulses.

Figure 9–12. Noninterlaced Video Timing Waveform Example



Key: ⓐ Vertical retrace
ⓑ Active portion of the frame

9.7.2 Programming the Vertical Timing Registers for Noninterlaced Video

For noninterlaced video, VCOUNT increments only after HCOUNT = HTOTAL, because there are no equalization or serration pulses with a period of half a scan line. Thus, with one exception, all the vertical timing registers are programmed in terms of the number of integral horizontal scan lines. VTOTAL is 1 less than the number of lines in the frame because VCOUNT counts 0 as its first value. VSBLNK and VEBLNK contain the line number, minus 1, where vertical blanking starts and ends, respectively. For reasons associated with composite interlaced video, the VESYNC register detects the end of vertical sync at **half** the value it is programmed to, and so should be programmed to **twice** the number of lines in the vertical-sync interval, minus 1. Figure 9–13 summarizes this.

Figure 9–13. Programming the Video Timing Registers for Noninterlaced Video

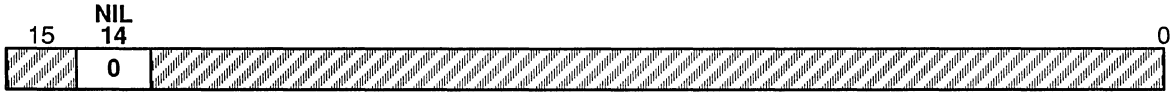
HESYNC	=	(the number of VCLKs in horizontal sync) – 1
HEBLNK	=	(the number of VCLKs from the start of horizontal sync to the end of horizontal blanking) – 1
HSBLNK	=	(the number of VCLKs from the start of horizontal sync to the start of horizontal blanking) – 1
HTOTAL	=	(the number of VCLKs in the line) – 1
HESERR	=	(the number of VCLKs in horizontal serration) – 1
VESYNC	=	(twice the number of lines in vertical sync) – 1
VEBLNK	=	(the number of lines from the start of vertical sync to the end of vertical blanking) – 1
VSBLNK	=	(the number of lines from the start of vertical sync to the start of vertical blanking – 1)
VTOTAL	=	(the number of lines in the frame) – 1

Note: You don't have to program HESERR if you're not using composite video.

9.8 Interlaced Video Timing

You can select interlaced scan mode by clearing NIL[[DPYCTL]] to 0.

DPYCTL register **address: C000 0080h**



Note: For a complete illustration and description of DPYCTL, see Chapter 4.

In interlaced mode, each video frame consists of two vertical fields of horizontal scan lines. The display consists of alternate lines from the two fields. This doubles the display resolution while only slightly increasing the frequency with which data is supplied to the screen.

The TMS34020 can produce interlaced video signals compatible with NTSC (RS-170), PAL, SECAM, and similar broadcast-quality video standards. The TMS34020 also supports higher resolution standards such as RS-330 and RS-343.

Note:

For simplicity, the illustrations in this section show American (NTSC) waveforms. European waveforms differ slightly; Section 9.8.3 (page 9-27) describes these differences.

9.8.1 Activity in Interlaced Mode

Figure 9–14 shows the path traced by the electron beam on the screen for a typical interlaced-video display. In interlaced mode, 2 separate vertical scans are performed for each frame—1 for the even lines (even field) and 1 for the odd lines (odd field). The even field is scanned first.

(a) shows the vertical retrace at the beginning of the even field, which coincides with the vertical-sync pulse. In this example, the vertical retrace is an integral number of horizontal scan lines in duration but, in fact, can be programmed to any number of *half* horizontal scan lines.

(b) shows the active portion of the even field. Solid lines represent displayed lines; dashed lines are blanked.

(c) & *(d)* show the vertical retrace and the active portion of the display, respectively, for the odd field.

(e) shows how the 2 fields form the complete display. Note that active scan lines 0—7 are partially blanked.

Figure 9–14. Electron Beam Pattern for a Typical Interlaced Display

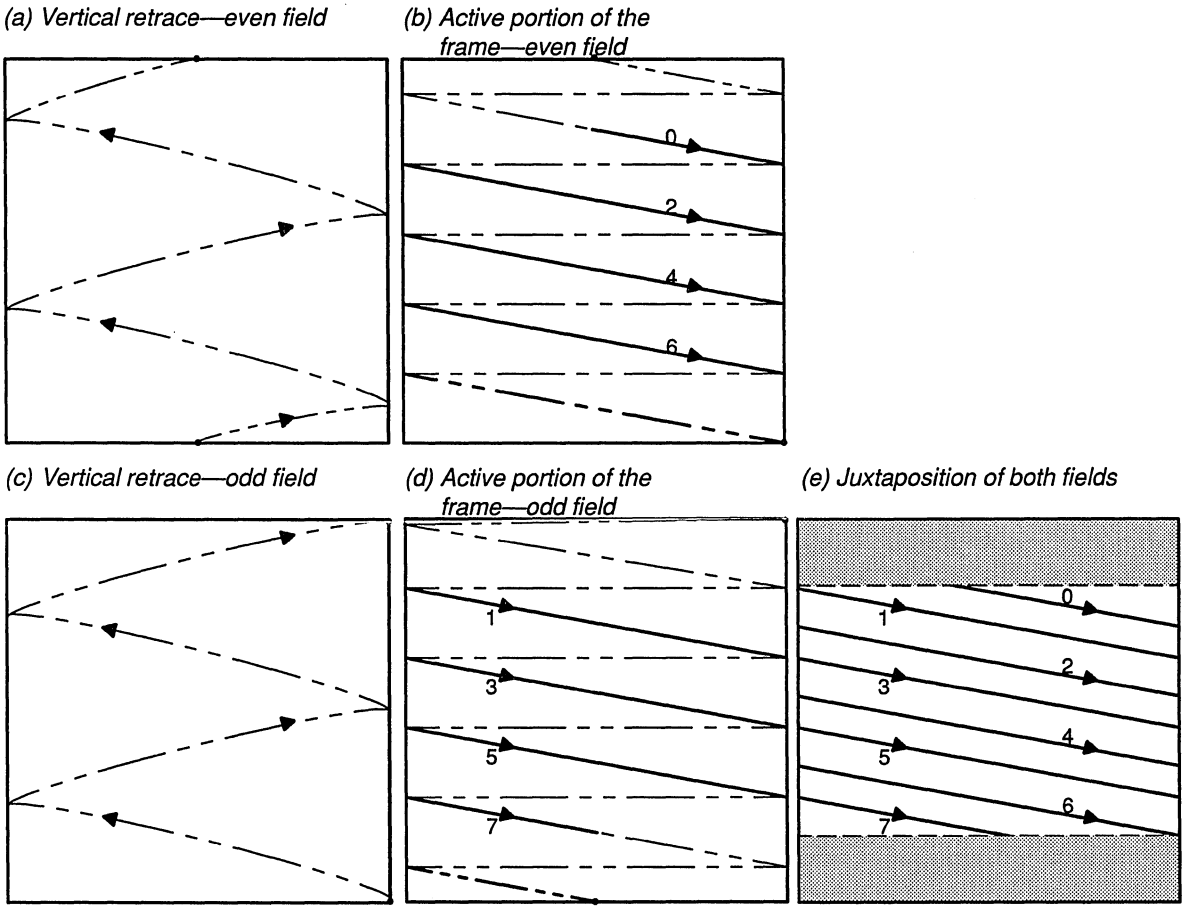


Figure 9–15 illustrates the video timing signals that generate a display similar to the one shown in Figure 9–14. However, Figure 9–15 has more blanked lines, to properly illustrate the composite-sync signal. The display line numbers and regions from Figure 9–14 are shown for reference. For completeness, all 6 possible output signals are shown, although you can choose only 4 in either of the 2 allowed combinations:

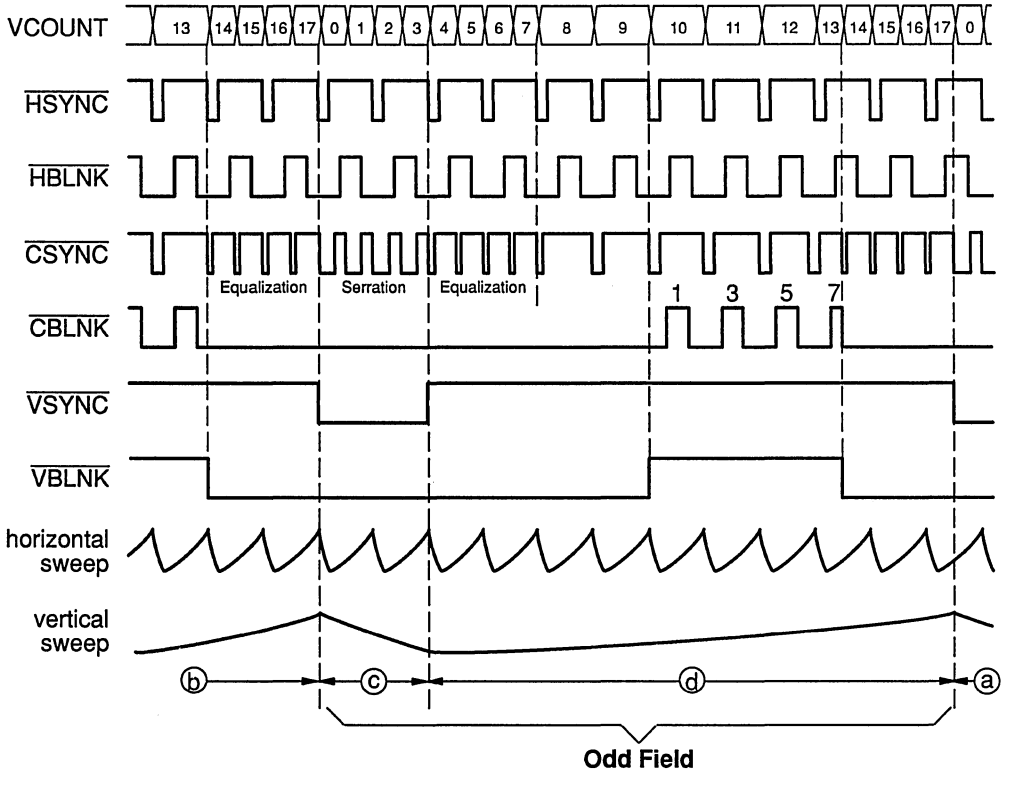
Combination 1 (CVD=0):	Combination 2 (CVD=1):
$\overline{\text{HSYNC}}$	$\overline{\text{HSYNC}}$
$\overline{\text{VSYNC}}$	$\overline{\text{VSYNC}}$
$\overline{\text{CSYNC}}$	$\overline{\text{HBLNK}}$
$\overline{\text{CBLNK}}$	$\overline{\text{VBLNK}}$

The example in Figure 9–15 uses the following register values:

VSBLNK = 13
VESYNC = 7
VTOTAL = 17
VEBLNK = 9

VCOUNT increments every half horizontal scan line for parts of the display; you must allow for this when programming the vertical timing registers. The formal programming equations and their derivations are given in Section 9.8.2.

Figure 9–15. Interlaced Video Timing Waveform Example



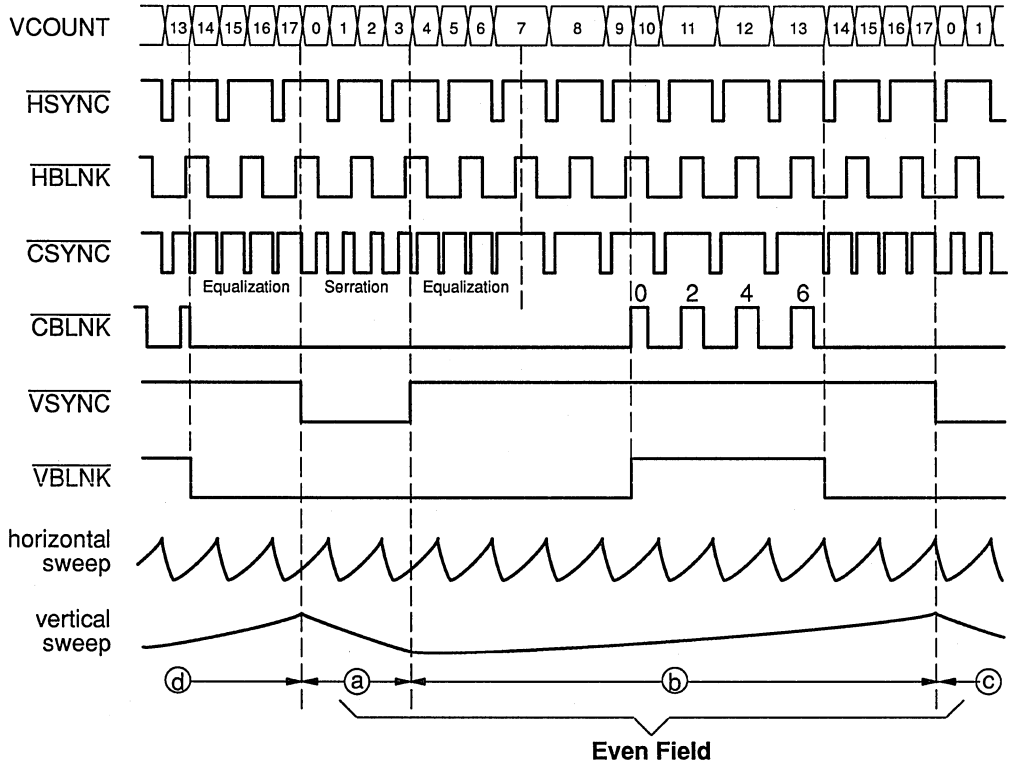
Note: This figure is continued on the next page.

- Vertical blanking **begins** at the end of a horizontal scan line near the end of the even field, and in the center of a horizontal scan line (after HCOUNT = HTOTAL/2) near the end of the odd field.

- Similarly, vertical blanking **ends** in the center of a horizontal scan line (after HCOUNT = HTOTAL/2) at the beginning of the even field, and ends at the end of a horizontal scan line at the beginning of the odd field.

The vertical-blanking interval is of the same duration in both the odd and even fields. In this way, the beam is positioned so that the horizontal scan lines in the odd field fall between the horizontal scan lines of the even field. Because each field contains an odd half line (not an integral number of lines), the total number of lines in the entire frame must be **odd**.

Figure 9–15. Interlaced Video Timing Waveform Example (Continued)



In Figure 9–15, VCOUNT is incremented twice per horizontal scan line during serration and equalization periods. This is the frequency at which equalization and serration pulses occur. Because VCOUNT increments every half line between the beginning of the first equalization period to the end of the second equalization period, you can specify the number of equalization and serration pulses required without being restricted to an integral number of lines.

Outside of the equalization and serration regions in Figure 9–15, VCOUNT increments only once per horizontal scan line, except there is an extra increment at the end of vertical blanking in the even field. This is necessary to ensure that the two fields are of exactly the same duration.

9.8.2 Programming the Vertical Timing Registers for Interlaced Video

As Figure 9–15 shows, VCOUNT is incremented twice per scan line during serration and equalization regions, and once per scan line outside of these regions. VTOTAL is not merely the total number of scan lines minus one, as it is for noninterlaced video. VTOTAL must account for those regions where VCOUNT is incremented twice per line. This applies to all interlaced video, independent of the value of CVD[[DPYCTL]]. VCOUNT always increments on half horizontal scan lines during equalization and serration regions of

interlaced displays, even if these pulses are not visible at the pins. For this reason, you must comply with the programming procedure and equations (described in this section and shown in Figure 9–17) for all interlaced displays.

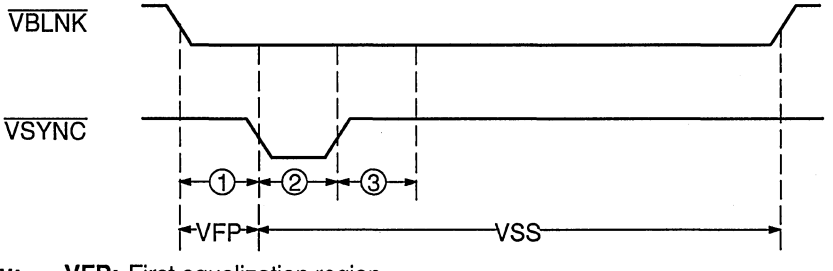
To program the vertical video timing registers for interlaced video, divide vertical blanking into two parts for calculation purposes:

VFP *Vertical front porch.* This is also the first equalization region (the region between the beginning of vertical blanking and the beginning of vertical sync).

VSS *Vertical sync to setup.* This is the region between the beginning of vertical sync and the end of vertical blanking. Note that this can also be thought of as comprising the serration region, the second equalization region plus the remainder of vertical blanking, or the vertical-sync region plus the vertical back porch.

Figure 9–8 (page 9-16) first described these regions; Figure 9–16 shows them again, illustrating the two compound regions used for calculation.

Figure 9–16. The Two Regions of Vertical Blanking Used for Programming Calculations



Key: **VFP:** First equalization region
VSS: Vertical sync to setup

Both VFP and VSS should be determined as a number of horizontal scan lines. The number of horizontal scan lines in vertical sync is also used in the calculations. If any one of these values is not an integral number of lines, they should **not** be rounded down. Rounding down should take place only at the end of each calculation.

After determining the duration of both VFP and VSS, you can consider the values that you must program into the registers:

- **VESYNC** detects the end of vertical sync at **half** its stored value. This is because the second equalization region is of equal duration to vertical sync, and the end of this region is detected at the full value of VESYNC. In addition, because VCOUNT increments every half horizontal scan line during equalization and serration, VESYNC should be programmed to 4 times the number of lines in vertical sync, less 1 because VCOUNT starts from 0:

  VESYNC = (4 x the number of lines in vertical sync) – 1

- **VTOTAL** contains the maximum VCOUNT value achieved for every field (odd and even). Because the counter increments every half line during equalization and serration, this is not simply the number of lines in the field, as is the case with noninterlaced video. Allowance must be made for the extra increments. VTOTAL is therefore 1/2 the total number of lines in the entire frame, plus 1 for the odd half line at the beginning or end of the field, plus the number of lines throughout which VCOUNT is incrementing every half line (to account for the half line increments), minus 1 because VCOUNT starts from 0. The *plus 1* and the *minus 1* cancel, and VCOUNT increments every half scan line during VFP, vertical sync, and the second equalization region (equal in duration to vertical sync). This reduces to:

  VTOTAL = (the number of lines in the frame / 2) + (2 x the number of lines in vertical sync) + VFP

- **VEBLNK** schedules the end of vertical blanking, which occurs VSS scan lines after VCOUNT is set to 0 at the beginning of vertical sync. However, VCOUNT increments every half scan line during vertical sync and the second equalization region (equal in duration to vertical sync). Allowing for the fact that VCOUNT starts from 0, this gives rise to the following expression:

  VEBLNK = VSS + (2 x the number of lines in vertical sync) – 1

- **VSBLNK** schedules the start of vertical blanking, which occurs VFP scan lines before the start of vertical sync (when VCOUNT = VTOTAL). Because VCOUNT increments twice every scan line during VFP, the value of VSBLNK is:

  VSBLNK = VTOTAL – (VFP x 2)

  In this case, there is no need to subtract 1 to account for VCOUNT starting from 0, because this is already accounted for in determining VTOTAL.

Figure 9–17. Programming the Video Timing Registers for Interlaced Video

HESYNC	=	(the number of VCLKs in horizontal sync) – 1
HEBLNK	=	(the number of VCLKs from the start of horizontal sync to the end of horizontal blanking) – 1
HSBLNK	=	(the number of VCLKs from the start of horizontal sync to the start of horizontal blanking) – 1
HTOTAL	=	(the number of VCLKs in the line) – 1
HESERR	=	(the number of VCLKs in horizontal serration) – 1
VESYNC	=	4 x (he number of lines in vertical sync) – 1
VTOTAL	=	(the number of lines in the frame) / 2 + (twice the number of lines in vertical sync) + VFP
VEBLNK	=	VSS + (twice the number of lines in vertical sync) – 1
VSBLNK	=	VTOTAL – (VFP x 2)

Note: You don't have to program HESERR if you're not using composite video.

9.8.3 American and European Video Standards

- **RS-170** is the **NTSC** broadcasting standard used for all *American* television.
- **PAL** is the *British* standard.
- **SECAM** is the *French* standard.

The programming procedures and equations described in Section 9.8.2 are the same for these and similar standards. However, as far as the production of sync and blanking signals are concerned, there is a fundamental difference between the American and European standards. For the American standard, the vertical retrace (specified by vertical sync) is an integral number of horizontal scan lines. For the European standards, however, the vertical retrace is not a whole number of lines; it contains an odd half line. As a result, the composite-sync waveform is different for American and European standards at the end of the second equalization region.

The diagrams and discussions in this chapter apply to the American format. However, the TMS34020 automatically adjusts to the European format by detecting whether or not VESYNC is programmed to indicate an even or odd number of half lines in vertical sync (if vertical sync is not an integral number of horizontal scan lines, it contains an odd number of half lines). Figure 9–18 shows how the composite-sync waveforms differ for the two types of standard.

Figure 9–18. Vertical Blanking for NTSC and PAL Standards

(a) NTSC

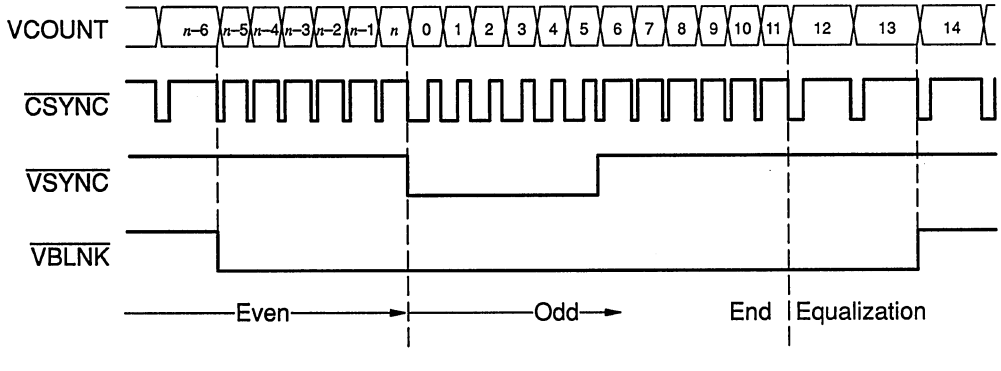
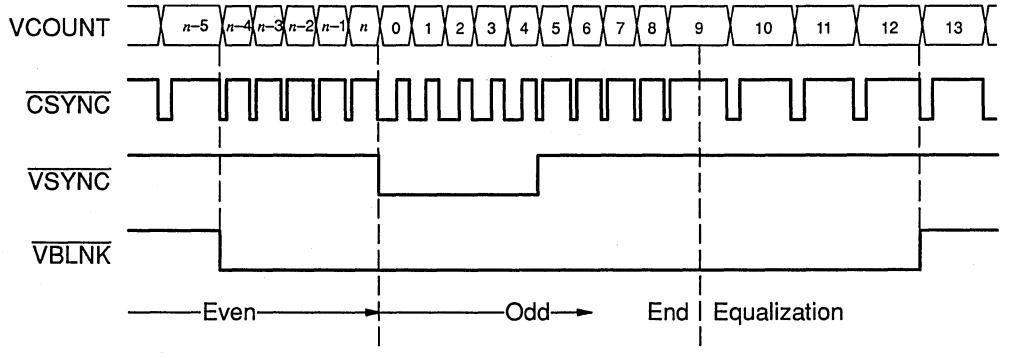
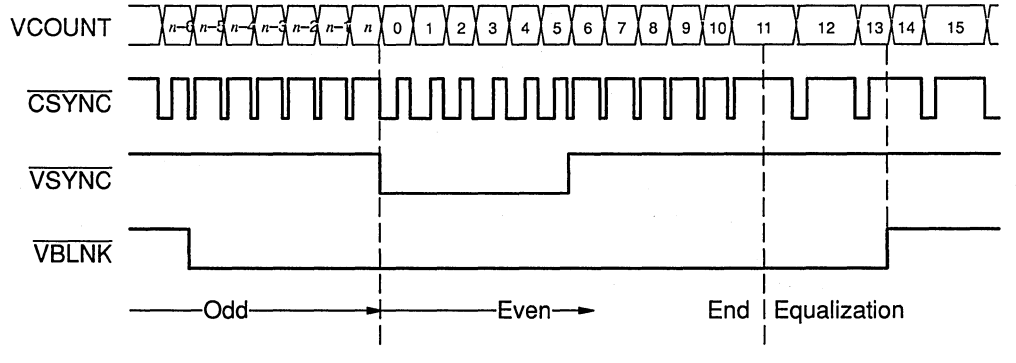


Figure 9–18. Vertical Blanking for NTSC and PAL Standards (Continued)

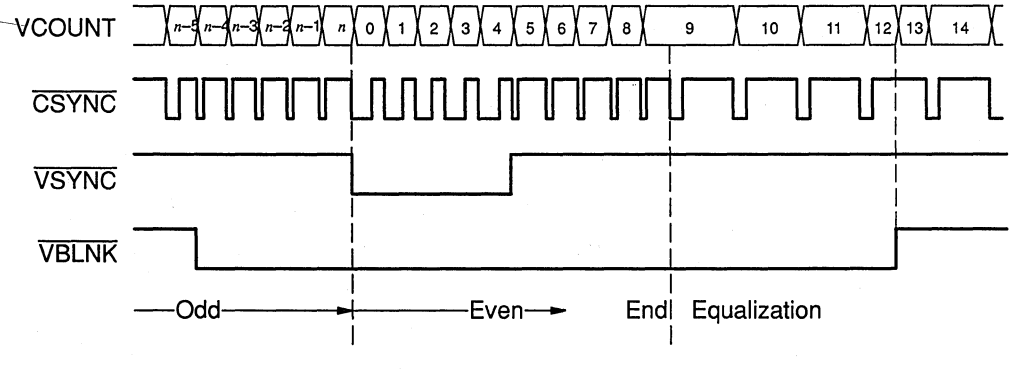
(b) PAL



(c) NTSC



(d) PAL



9.9 External Synchronization Modes

External synchronization modes enable the TMS34020 to use horizontal-, vertical-, and composite-sync signals from an external source. This allows you to superimpose or mix TMS34020-generated graphics with images from external sources.

Depending on the video pin configuration (separate or composite sync, selectable via CVD[[DPYCTL]]), either 2 or 3 of the 4 video pins are configured as sync pins. External synchronization mode is selected by configuring the appropriate sync pins as inputs by way of the CSD, VSD, or HSD bits in DPYCTL. Each pin is configured as an input when its associated bit is a 0. In separate-sync mode, the CSD bit must be a 1 because the $\overline{\text{CSYNC}}/\overline{\text{HBLNK}}$ pin is configured as horizontal blanking and, as such, is an output only.

When all the sync pins are configured as outputs, the horizontal and vertical counters are cleared to 0s after HCOUNT = HTOTAL and after VCOUNT = VTOTAL, respectively. This also initiates the corresponding sync pulse. However, when one or more of the sync pins is configured as an input, this behavior is altered somewhat; a high-to-low transition on an input sync waveform (the beginning of an external sync pulse) causes one or both of the video counters to be loaded from SETHCNT and SETVCNT. By loading the video counters with a programmable value instead of 0, you can compensate for delays inherent in the process of synchronizing the external sync signals to VCLK within the TMS34020, for the time required for the TMS34020 to respond to the external sync signals, and also for external signal skews. HCOUNT and VCOUNT are reloaded as follows:

- HCOUNT is reloaded by a falling edge on either the $\overline{\text{HSYNC}}$ or the $\overline{\text{CSYNC}}$ pin. In noninterlaced video mode, a falling edge on $\overline{\text{VSYNC}}$ also reloads the horizontal counter.
- VCOUNT is reloaded by a falling edge on the $\overline{\text{VSYNC}}$ pin or the first serration pulse on the $\overline{\text{CSYNC}}$ pin (which occurs at the beginning of vertical sync). For this reason, the serration pulse width must be at least 2 VCLK periods greater than the internal horizontal-sync pulse width. In programming terms, this means HESERR ≥ HESYNC + 2. If this is not the case, the TMS34020 will not be able to detect the beginning of vertical sync from the composite input waveform. The HESERR and HESYNC registers **must** be programmed to accurately match the parameters of the external video source.

By causing VCOUNT and HCOUNT to follow the external synchronization signals in this way, the blanking intervals and screen-refresh cycles are also forced to follow the external video source, and the TMS34020 is therefore synchronized to the external video.

While the sync pins are independently configurable as inputs or outputs, not all of the possible combinations of inputs are useful. Typically, you would use one of the following 3 combinations of inputs:

- **External vertical-sync signal only.**

$\overline{\text{VSYNC}}$: input	$\overline{\text{HSYNC}}$: output	$\overline{\text{CSYNC}}/\overline{\text{HBLNK}}$: output
Used with noninterlaced video		

HCOUNT and VCOUNT are loaded simultaneously at the beginning of the external vertical-sync pulse, thus completely synchronizing the TMS34020 to the external source. Because HCOUNT is synchronized only once per frame, it is important that it be programmed to the correct value for the external source. For interlaced video, vertical sync coincides with a horizontal-sync pulse in every other frame only. Also, vertical sync alone is not sufficient to determine the field parity (odd/even) of the external video source, and therefore is insufficient to determine the field in which the horizontal and vertical sync coincide. For these reasons, external vertical sync has no effect on the horizontal timing registers in interlaced video.

- **External vertical- and horizontal-sync signals.**

$\overline{\text{VSYNC}}$: input	$\overline{\text{HSYNC}}$: input	$\overline{\text{CSYNC}}/\overline{\text{HBLNK}}$: output
Used in both composite- and separate-sync modes, interlaced or noninterlaced video		

The beginning of the external vertical-sync signal reloads VCOUNT; the beginning of the external horizontal-sync signal reloads HCOUNT.

- **External composite-sync signal only.**

$\overline{\text{VSYNC}}$: output	$\overline{\text{HSYNC}}$: output	$\overline{\text{CSYNC}}$: input
Used with interlaced video only		

The TMS34020 must be configured in composite video mode (CVD=0) so that the $\overline{\text{CSYNC}}/\overline{\text{HBLNK}}$ pin is selected as $\overline{\text{CSYNC}}$. The beginning of each external composite-sync signal reloads HCOUNT, and the beginning of the first serration pulse reloads VCOUNT. For noninterlaced video, the TMS34020 does not recognize equalization or serration pulses, and so cannot detect the beginning of vertical sync from a composite signal. For this reason, external composite sync has no effect on the vertical timing registers in noninterlaced mode.

To avoid any potential signal conflicts, all three sync pins are configured as inputs when the TMS34020 is reset. However, sync pins not actually being driven by external signals should subsequently be configured as outputs.

9.9.1 Odd and Even Field Alignment in Interlaced Mode

In interlaced mode, the TMS34020 synchronizes with the same field parity as the external source, *provided that* VTOTAL is programmed to exactly match the external source. This is necessary because of the way that the video timing logic adjusts to the correct field.

If the internal video timing has field parity opposite to the external source, the horizontal-sync pulse generated by the internal video timing logic is initially displaced from those of the external system by half a scan line. However, as soon as an external horizontal-sync pulse occurs, HCOUNT is reset, and VCOUNT is incremented. This realigns the internally generated horizontal-sync pulses with those of the external system. Because this occurs while the internal video timing logic is midway through a scan line (where VCOUNT would not normally be incremented), it causes the internal field to be shortened by half a scan line. The TMS34020 takes advantage of this at the end of the field to ensure that it becomes aligned to the same field as the external source.

After reaching VTOTAL, the internal video timing logic starts a new field with the opposite field parity. However, because the previous field was shortened by a half a scan line, the external vertical sync does not occur for another half scan line. When it does occur, it causes the video logic to resynchronize to the external source and start yet another new field, and the field parity changes again. Field parity changes twice, whereas the external source changes field parity only once. Thus, the internal video timing logic has the same field parity as the external source due to the extra (half scan line long) field.

If the internal and external vertical-sync pulses are separated by less than the duration of the horizontal-sync pulse (if CVD=1) or the composite-serration pulse (if CVD=0), the internal field parity does not change twice. This allows for normal differences between when VCOUNT = VTOTAL and detection of the beginning of the external vertical-sync interval. This will be the case when the internal and external field parities match correctly, *provided that* VTOTAL matches the external source. If this is not the case, one of the following situations applies.

- If VTOTAL gives an internal field duration less than that of the external source, the internal video timing logic changes field parity twice per field: once when VCOUNT = VTOTAL, and once at the beginning of the external vertical-sync interval.

- If VTOTAL gives an internal field duration greater than that of the external source, the internal video timing logic changes field parity only once per field: At the beginning of the external vertical-sync interval, even if the internal video timing logic does not have the same field parity as the external source.

9.9.2 Synchronizing External Syncs to VCLK

The TMS34020 synchronizes input signals to VCLK before passing them to the internal video logic. This means that inputs can be asynchronous to VCLK. The delay from the high-to-low transition of an external sync input to an occurrence of transition-induced changes at the video output pins will be from 4 to 5 VCLK periods, depending on the phase relationship between the transition and VCLK. If you do not require the TMS34020 to follow the external video source to an accuracy of 1 VCLK period (if, for example, the TMS34020 uses the external synchronization simply to perform screen refreshes or display interrupts), asynchronous operation may be acceptable.

If you use any of the video output signals (either blanking or sync) to control any other part of the system, then the TMS34020 may need to synchronize precisely to the external video source. In this case, you can present the external input sync signals synchronous to VCLK. If the setup and hold times for a valid level on one of the external video input pins with respect to the low-to-high transition of VCLK (described in the *TMS34020 Data Sheet*, Appendix A) are met, then the timing relationship between input signals and output signals is a constant, integral number of VCLK cycles. However, the exact number of cycles will appear to vary according to the value programmed in the SETHCNT register.

9.9.3 Loading the Video Counters

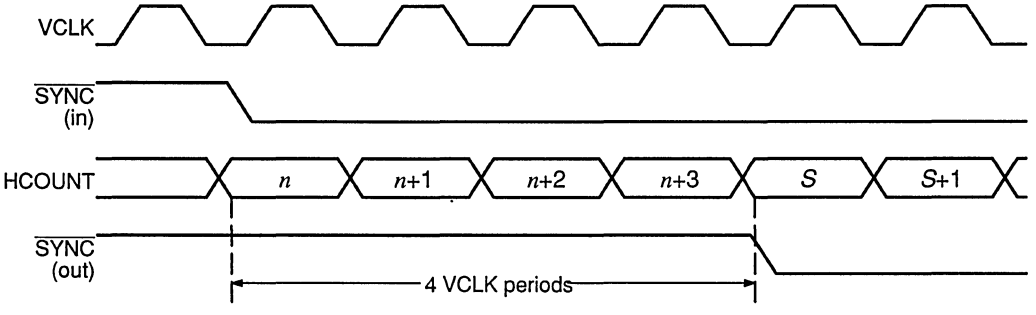
When the external video input signals are presented synchronous to VCLK, there is a 4-VCLK delay between the VCLK cycle when a falling edge is sampled on an external sync pin, and when the changes induced by this edge are visible at the video output pins. If you require the TMS34020 to accurately follow the external source, the effect of this delay can be eliminated by using the SETHCNT register.

SETHCNT is loaded into HCOUNT at the beginning of the fifth VCLK period after the appropriate external sync signal is detected at the input pin. This is shown in Figure 9–19 *(a)*. By programming SETHCNT to 4, the TMS34020's video timing registers are aligned exactly to the external source. Loading HCOUNT to four 4-VCLK cycles, after the transition on the external sync pin, is equivalent to loading HCOUNT to 0 as soon as the transition on the external sync pin is detected.

Once the internal video timing logic is synchronized to the external system (by the first high-to-low transition of an external sync pin detected by the TMS34020), the video timing outputs are aligned with the external system on subsequent scan lines, provided that HCOUNT is programmed to match the external system. This is shown in Figure 9–19 *(b)*; the internal HTOTAL occurs simultaneously with the end of the external system's scan line, causing HCOUNT to be reset to 0, activating the appropriate output sync signals at the same time as the input sync from the external system.

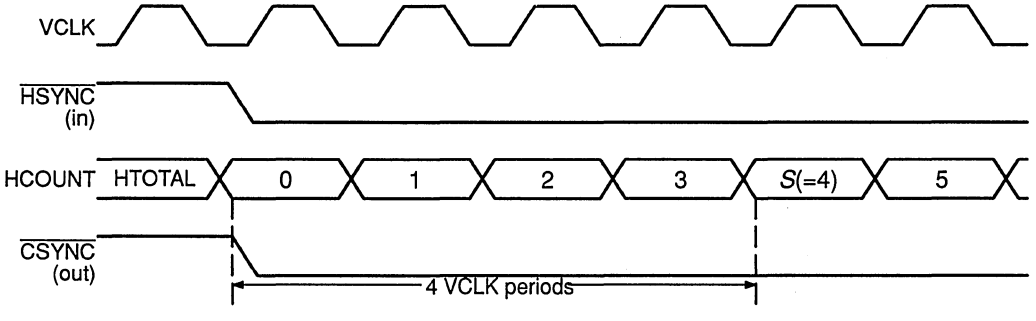
Figure 9–19. Synchronization Delay Compensation

(a) Initial synchronization to external sync



Note: *S* is the value contained in SETHCNT. 4-VCLK cycles after the falling edge on the input $\overline{\text{SYNC}}$ pin occurs, HCOUNT is loaded with *S*, and the transition on the output $\overline{\text{SYNC}}$ is caused to occur.

(b) Subsequent synchronization to external sync



Note: If HTOTAL matches the external system and *S*=4, HTOTAL causes the output sync pin to transition in the same VCLK cycle as the input sync.

Programming SETHCNT to values greater than 4 causes the TMS34020's video outputs to be in advance of the external video source. This can be useful for eliminating additional signal skews in the system.

There is a similar register for loading the vertical counter, SETVCNT. This must be loaded with **0** for interlaced video mode; otherwise, the TMS34020 will not be able to synchronize to the external field parity, for the reasons discussed in Section 9.9.1. If desired, you may load it with nonzero values in noninterlaced video mode; this displaces the internal video from the external video by a number of scan lines.

9.9.4 Synchronization Conversion

When you load SETHCNT with 4, the TMS34020's video output signals are synchronized with the input signals, provided that the input signals meet the setup and hold times for synchronous operation (see the *TMS34020 Data Sheet*). Because of this feature, the TMS34020 can be used to convert one form of synchronization waveforms into another.

- If the $\overline{\text{VSYNC}}$ and $\overline{\text{HSYNC}}$ pins are selected as inputs and the video timing registers accurately match the parameters of the external source, the $\overline{\text{CSYNC}}$ output is an exactly synchronized composite waveform with (in interlaced mode) the correct equalization and serration pulses.

- If the $\overline{\text{CSYNC}}$ pin is selected as an input and the video timing registers accurately match the parameters of the external source, the $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$ outputs are exactly synchronized separate waveforms equivalent to the input.

9.9.5 Programming Flexibility and Limitations

For asynchronous operation in which the TMS34020 is not required to exactly follow the external source, you can program HTOTAL to a very large value (such as FFFFh). This prevents the condition HCOUNT = HTOTAL from occurring, and HCOUNT is reset only when a high-to-low transition on an external sync occurs. When using external synchronization modes there are, depending on the application, a number of different approaches that can be taken in programming the video timing registers.

For noninterlaced video, the same is also true for VTOTAL; however, for interlaced video, VTOTAL **must** be programmed to the correct value for the TMS34020 to be able to resolve the field parity of the external source.

Anytime SETHCNT is not 0 (typically for synchronous operation), HTOTAL should be loaded to match the external source. If this is not adhered to, then the TMS34020 will not be able to start horizontal- or composite-sync pulses coincident with the external-sync pulse, because the condition HCOUNT = HTOTAL will not occur. If you are not concerned about starting horizontal- or composite-sync pulses exactly aligned, but still wish to use SETHCNT, HTOTAL can be programmed to a large value as detailed above. If you take this approach, there are some pitfalls to avoid: Because HCOUNT is reloaded with the value in SETHCNT but HTOTAL is never reached, the counter never counts through those values from 0 up to SETHCNT – 1. You must ensure that the values of HESYNC, HESYNC/2, HESERR, HSBLNK, HEBLNK, or HTOTAL/2 do not coincide with any of these values; if they do coincide, the output waveforms could be corrupted. Again, for interlaced video, VTOTAL must be accurately programmed.

To allow the TMS34020 to detect the beginning of vertical sync from a composite-sync input, HESERR must be equal to at least HESYNC + 2. The TMS34020 samples the state of the composite input waveform when HCOUNT = HESYNC to determine whether serration has begun; if this is not adhered to, the TMS34020 may mistake a regular horizontal-sync pulse for a serration pulse.

For the TMS34010, it was important that HTOTAL and VTOTAL contained values large enough not to cause HCOUNT and VCOUNT to be cleared before the leading edges of the external sync pulses could clear them. This is not the case for the TMS34020. Instead, it is more important to program the registers accurately; it is essential to program VTOTAL correctly for interlaced field parity alignment, and HTOTAL correctly for synchronization conversion.

9.9.6 External Synchronization Pulse Widths

The external synchronization pulses input to the TMS34020 should have the following dimensions:

- **Minimum width of $\overline{\text{HSYNC}}$ and $\overline{\text{VSYNC}}$**: Not less than 1 VCLK cycle.
- **Minimum width of $\overline{\text{CSYNC}}$**: Same as for $\overline{\text{HSYNC}}$/$\overline{\text{VSYNC}}$ except the serration pulses input during the vertical-sync portion of the signal must be at least 2 VCLK periods longer than the ordinary horizontal-sync pulse.
- **Maximum width, all pins**: Not greater than the internally generated pulse (HESYNC + 1, etc.).
- As for internal interlaced video, HSBLNK, HTOTAL/2, and HESYNC/2 must be separated from each other by a time of at least 2 VCLK periods plus 2 LCLK periods, ensuring that the memory controller can detect the correct requests.

9.10 Screen Sizes and Dot Rate

The TMS34020's 512-Mbyte address reach supports very high-resolution displays. For example, a large TMS34020-based system could use the lower half of the address space for display memory and use the upper half for storing programs and data. The 256-Mbyte display memory in this example could support the following display sizes:

- 8,192 by 8,192 pixels at 32 bits per pixel
- 16,384 by 8,192 pixels at 16 bits per pixel
- 16,384 by 16,384 pixels at 8 bits per pixel
- 32,768 by 16,384 pixels at 4 bits per pixel
- 32,768 by 32,768 pixels at 2 bits per pixel
- 65,536 by 32,768 pixels at 1 bits per pixel

At most, all of the TMS34020's memory space below that allocated to the I/O registers could be allocated to the display.

The video timing registers also support high-resolution displays. The 16-bit vertical counter register, VCOUNT, directly supports screen lengths of up to 65,536 lines. The 16-bit horizontal counter register, HCOUNT, does not directly limit horizontal resolution. Each horizontal line can be up to 65,536 VCLK (video clock) periods in length. The VCLK period, however, is an arbitrary number of dot-clock periods long, depending on the external divide-down logic that derives VCLK from the dot clock. Therefore, the number of pixels per line supported by the TMS34020's horizontal timing registers is limited only by the amount of video memory present.

A typical screen must be refreshed 60 times per second for a noninterlaced display, or 30 times per second for an interlaced display. For a noninterlaced display, the dot period (or time to refresh 1 pixel) is estimated as

$$\textit{dot period} = \frac{\textit{DBR} \times \textit{(1/60 second)}}{\textit{(pixels/line) (lines/frame)}}$$

For an interlaced display, the dot period is estimated as

$$\textit{dot period} = \frac{\textit{DBR} \times \textit{(1/30 second)}}{\textit{(pixels/line) (lines/frame)}}$$

DBR is the **display-to-blanking ratio**, equal to the unblanked fraction of each frame. This is typically about 0.8, although this factor varies from monitor to monitor. During each dot period, the complete information for 1 pixel must be obtained from the display memory, or frame buffer. Thus, the rate at which video data must be supplied from the display memory (usually the limiting factor for large systems) is a function of pixel size as well as screen dimensions.

9.11 Display Interrupts and Applications

You can program the TMS34020 to interrupt the CPU when a specified line is displayed on the screen. This is called the **display interrupt**. Enabling the display interrupt is a 2-step process:

Step 1: Set DIE[[INTENB]] to 1.

Step 2: Load the DPYINT register with the number of the desired horizontal scan line. When VCOUNT = DPYINT, the interrupt request is generated to coincide with the start of horizontal blanking at the end of the specified line.

DIP[[INTPEND]] is set every time the interrupt request is generated. You can poll the display interrupt by disabling the interrupt (setting DIE to a 0) and checking the value of DIP. Writing a 0 to DIP clears the interrupt request.

The display interrupt has several applications:

- Coordinating modifications to a bitmap with displaying the bitmap's contents. For example, while the bottom half of the screen is being displayed, the TMS34020 can modify the bitmap used for the top half of the display, and vice versa.

- Maintaining a cursor on the monitor screen. The cursor image resides in the on-screen memory only during the time the electron beam is scanning the lines containing the cursor, and remains free from flicker even during periods when the TMS34020 is busy drawing to the screen. The technique is to load the DPYINT register with the VCOUNT value of the scan line just above where the top of the cursor is to appear. When the display interrupt occurs, the interrupt service routine performs the following tasks:

  - Sets DPYINT to the scan line just below the cursor,
  - Saves the portion of the screen where the cursor is to appear, and
  - PIXBLTs the cursor onto the screen.

  The cursor then remains on the screen until the electron beam next scans the lines it is on. In the mean time, as soon as the electron beam reaches the bottom of the cursor, a second display interrupt occurs. This causes the original screen to be restored in preparation for the next frame, and the TMS34020 can resume drawing to the screen.

- Split-screen applications. By modifying the contents of SRNX[[DPYNX]] part way through a frame, you can display different parts of the bitmap in different horizontal bands of the screen. Sections 9.15.2 and 9.15.3 (beginning on page 9-53) discuss the use of SRNX as part of the screen-refresh mechanism. No special steps are necessary to ensure that loading a new value to SRNX does not interfere with the ongoing screen- refresh cycle; the display interrupt is requested at the beginning of the horizontal-

blanking interval coincident with a screen-refresh request. However, the TMS34020 cannot respond to the interrupt request until the screen refresh and subsequent updating of SRNX is complete (this is true whether the interrupt was taken or the TMS34020 polls the DIP bit for the 0-to-1 transition). After DIP is set to 1, SRNX can be loaded with a new value to achieve the split screen any time before the next screen-refresh cycle.

In interlaced mode, VCOUNT increments every half scan line during the equalization and serration regions of vertical blanking. When this is occurring, DPYINT is compared with VCOUNT twice per scan line; the display interrupt may occur at the start of horizontal blanking and also when HCOUNT = HTOTAL/2 (in the center of each line).

9.12 Video Timing Programming Examples

This section illustrates procedures for determining the values to be programmed into the TMS34020's video timing logic.

9.12.1 Noninterlaced 1024 × 768 Display

This example assumes that the NIL, CVD, and CSD bits in DPYCTL are set to 1, so that the TMS34020 is in noninterlaced video mode with separate horizontal- and vertical-blanking outputs. The values of other control bits in DPYCTL depend on other system specifications not covered in this example (such as external synchronization modes).

Specifications

The monitor in this example uses these parameters; you could easily modify the example to fit the specifications of another monitor.

- **Horizontal** (refer to Figure 9–3 on page 9-11):

Scan line duration (LD):	20.625 μS
Sync duration (HS):	1 μS
Back porch (HBP):	2.875 μS
Front porch (HFP):	0.75 μS

- **Vertical** (refer to Figure 9–6 on page 9-13):

Frame duration (FD):	16.665 mS
Sync duration (VS):	83 μS
Back porch (VBP):	660 μS
Front porch (VFP):	82 μS

- **Screen dimensions:** 1024 pixels by 768 lines

Procedure

Figure 9–13 (page 9-20) lists the timing register programming equations for noninterlaced video. Before you can calculate proper values for the video timing registers, you must determine the active display time and the VCLK period:

- Use the horizontal information to calculate the **active display time**:

  active time (AT) = LD – HS – HBP – HFP = *16.000* µS

  At 1024 pixels per line, this is 15.625 nS per pixel.

- Determine the **VCLK period**. The minimum VCLK period VCLK is 50 nS, which means that the pixel dot clock must be divided by at least 4 to generate VCLK. Assuming this to be the case,

  VCLK period (T_{vclk}) = *62.5 nS* and *VCLK frequency* = *16 MHz*

Step	Calculate...	Formula	Yields a register value of...
1	Number of VCLK cycles per scan line	$\frac{LD}{T_{vclk}} = 330$	HTOTAL = 329_{10} (149h)
2	Number of VCLK cycles in horizontal sync	$\frac{HS}{T_{vclk}} = 16$	HESYNC = 15_{10} (0Fh)
3	Number of VCLK cycles in the horizontal back porch	$\frac{HBP}{T_{vclk}} = 46$	
4	Number of VCLK cycles between the beginning of horizontal sync and the end of horizontal blanking	$\frac{HS + HBP}{T_{vclk}} = 62$	HEBLNK = 61_{10} (3Dh)
5	Number of VCLK cycles in the horizontal front porch	$\frac{HFP}{T_{vclk}} = 12$	
6	Number of VCLK cycles between the beginning of horizontal sync and the beginning of horizontal blanking	$\frac{LD - HBP}{T_{vclk}} = 218$	HSBLNK = 217_{10} (0D9h)
7	Number of lines per frame	$\frac{FD}{LD} = 808$	VTOTAL = 807_{10} (327h)
8	Number of lines in vertical sync	$\frac{VS}{LD} = 4$	VESYNC = 7_{10} (7h)
9	Number of lines in the vertical back porch	$\frac{VBP}{LD} = 32$	
10	Number of lines between the beginning of vertical sync and the end of vertical blanking	$\frac{VS + VBP}{LD} = 36$	VEBLNK = 35_{10} (23h)
11	Number of lines in the vertical front porch	$\frac{VFP}{LD} = 4$	
12	Number of lines between the beginning of vertical sync and the beginning of vertical blanking	FD – VFP = *804*	VSBLNK = 803_{10} (323h)

9.12.2 Composite Interlaced NTSC Display Example

This example assumes that the NIL and CVD bits in DPYCTL are set to 0 to configure the TMS34020 for interlaced video, with $\overline{\text{CSYNC}}/\overline{\text{HBLNK}}$ and $\overline{\text{CBLNK}}/\overline{\text{VBLNK}}$ configured as $\overline{\text{CSYNC}}$ and $\overline{\text{CBLNK}}$, respectively. The values of other control bits in DPYCTL depend on other system specifications not covered in this example (such as external synchronization modes).

Specifications

This example demonstrates how to program the TMS34020's video timing registers for fully NTSC-compatible composite video (that is, broadcast-standard TV). The required specifications are

- **Horizontal** (refer to Figure 9–3 on page 9-11):

Scan line duration (LD):	63.556 µS
Sync duration (HS):	4.7 (±0.1) µS
Sync to setup (HSS)†:	9.4 (±0.1) µS
Front porch (HFP):	1.5 (±0.1) µS
Subcarrier period (SP):	(2 / 455) × LD

- **Vertical** (refer to Figure 9–6 on page 9-13):

Frame duration (FD):	525 lines
Sync duration (VS):	3 lines
Sync to setup (VSS)†:	17 lines
Front porch (VFP):	3 lines

- **Screen dimensions:** A typical TV display has 376 pixels per line. This yields screen dimensions of 376 pixels × 525 lines.

† *Sync to setup* is an NTSC-specified parameter, and is equal to sync plus the back porch.

Procedure

Figure 9–17 (page 9-26) lists the timing register programming equations for interlaced video. Before you can calculate proper values for the video timing registers, you must determine the minimum VCLK period and frequency.

In calculating the **minimum VCLK frequency**, the most important consideration for NTSC is that the NTSC subcarrier waveform (which transmits color information) and VCLK must be harmonically related. The number of subcarrier cycles per scan line is

$$LD / SP = 227.5$$

There must be an integral number of VCLKs per line, which means that VCLK must be at least twice the frequency of the subcarrier, resulting in 455 VCLKs per line. However, for composite video, the equalization and serration pulses that occur in vertical blanking occur at exactly half scan line intervals, and this

means that the number of VCLKs per line must also be even. This results in VCLK being 4 times the frequency of the subcarrier, with 910 VCLKs per line. Therefore,

VCLK period (T_{vclk}) = 69.48 nS and *VCLK frequency = 14.318 MHz*

Step	Calculate...	Formula	Yields a register value of...
1	Number of VCLK cycles per scan line	$\frac{LD}{T_{vclk}} = 910$	HTOTAL = 909_{10} (148h)
2	Number of VCLK cycles in horizontal sync	$\frac{HS}{T_{vclk}} = 67.29$	HESYNC = 67_{10} (42h)
		This is not an integer, but if 68 is chosen, the value of HS derived is still within the ±0.1 µS tolerance specified: 4.749 µS. Remember that horizontal sync should be an even number of VCLKs long to ensure that the equalization pulses are exactly half that duration.	
3	Number of VCLK cycles in horizontal serration	$\frac{\frac{LD}{2} - HS}{T_{vclk}} = 387.7$	HESERR = 387_{10} (183h)
		388 gives the inactive time between serration pulses, which is specified identical to HS, as 4.679 µS, which is within the tolerance	
4	Number of VCLK cycles between the beginning of horizontal sync and the end of horizontal blanking	$\frac{HSS}{T_{vclk}} = 134.58$	HEBLNK = 134_{10} (86h)
		135 is within the tolerance specified: HSS=9.428 µS	
5	Number of VCLK cycles in the horizontal front porch	$\frac{HFP}{T_{vclk}} = 21.47$	
		22 is within the tolerance specified: HFP=1.536 µS	
6	Number of VCLK cycles between the beginning of horizontal sync and the beginning of horizontal blanking	$\frac{LD - HFP}{T_{vclk}} = 888$	HSBLNK = 887_{10} (337h)

Because the vertical timing parameters are already specified in lines, it is necessary only to work through the formulas given in Figure 9–17 to calculate the vertical timing register programming:

Step	Calculate...	Formula	Yields a register value of...
7	VESYNC	$VS \times 4 - 1$	VESYNC = 11_{10} (0Bh)
8	VTOTAL	$\frac{FD}{2} + (2 \times VS) + VFP = 271.5$	VESYNC = 271_{10} (10Fh)
9	VSBLNK	$VTOTAL - VFP \times 2$	VSBLNK = 265_{10} (109h)
10	VEBLNK	$VSS + 2 \times VS - 1$	VEBLNK = 22_{10} (16h)

9.13 Video RAM Control

Display memory is the area of memory that holds the graphics image output to the video monitor. The display memory is typically implemented with VRAMs (video RAMs). The TMS34020 automatically schedules VRAM memory-to-register cycles, called **screen-refresh cycles**, needed to refresh a video screen. A screen-refresh cycle typically affects all VRAMs in a system. During a screen-refresh cycle, a selected row of the display memory is transferred to the internal serial register of each VRAM. The data is then shifted out to refresh the display.

The TMS34020 supports 2 distinct types of screen refresh:

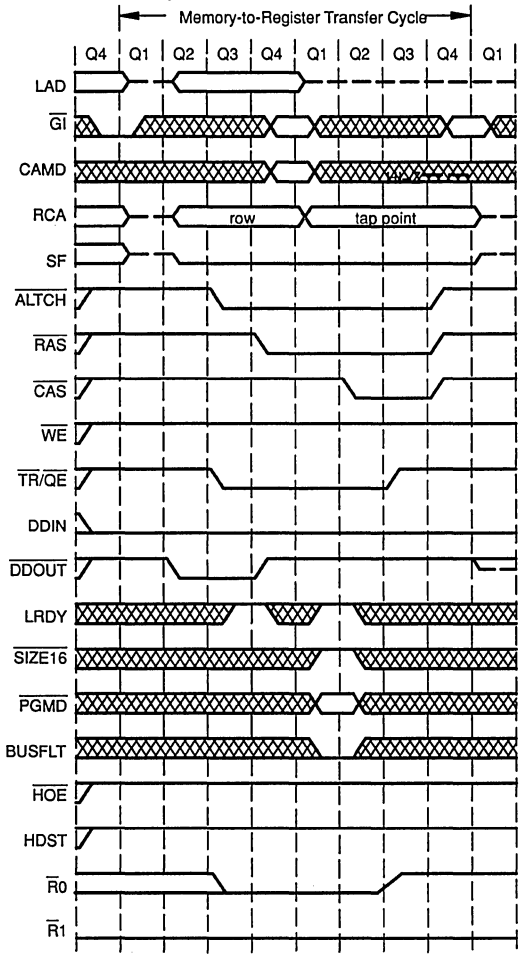
- Screen refreshes that occur during horizontal blanking and
- Screen refreshes that occur during the active display time.

The addresses output during both types of screen-refresh cycles are generated by a separate mechanism discussed in Section 9.15 (page 9-51); this section also discusses other VRAM-control options programmed via the DPYCTL register.

9.13.1 Screen Refreshes During Horizontal Blanking

The video timing logic schedules a screen-refresh cycle at the beginning of each horizontal-blanking interval. During blanking, the VRAM serial registers should not be shifting data (SCLK should be off). During the vertical-blanking interval, no screen refreshes take place until the horizontal-blanking interval at the beginning of the first displayed line of the new frame. Figure 9–20 shows the local-memory screen-refresh memory cycle.

Figure 9–20. Local-Memory Memory-to-Register Transfer Cycle



9.13.2 Screen Refreshes During the Active Display Time (Midline Reload)

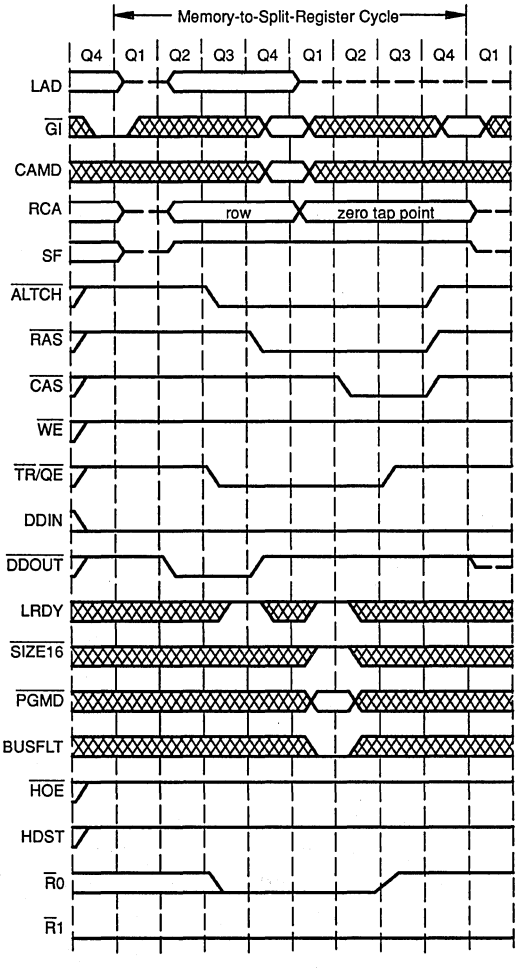
The TMS34020 contains dedicated circuitry that operates synchronously to the VRAM's shift clock (SCLK). This enables the TMS34020 to perform screen refreshes during the active display time in systems with VRAMs that have split serial registers (such as the TMS44C251, a 1-Mbit, 256K×*n* VRAM). These occur without interrupting the flow of data to the screen. If enabled, midline-reload screen-refresh cycles occur as well as (not instead of) horizontal-blanking screen-refresh cycles.

The SCOUNT register is loaded automatically during the horizontal-blanking screen refresh with the column address (or tap-point) portion of the logical address output to the VRAMs. When blanking ends and SCLK starts again, SCOUNT is incremented each time the VRAM serial registers shift out a bit of data. In this way, SCOUNT always contains the column address of the bit of

data currently being shifted out. When SCOUNT increments from all 1s (the column address of the last bit to be shifted out of one-half of the serial register) to all 0s (the column address of the first bit to be shifted out of the other half of the serial register), a midline-reload screen-refresh cycle is scheduled.

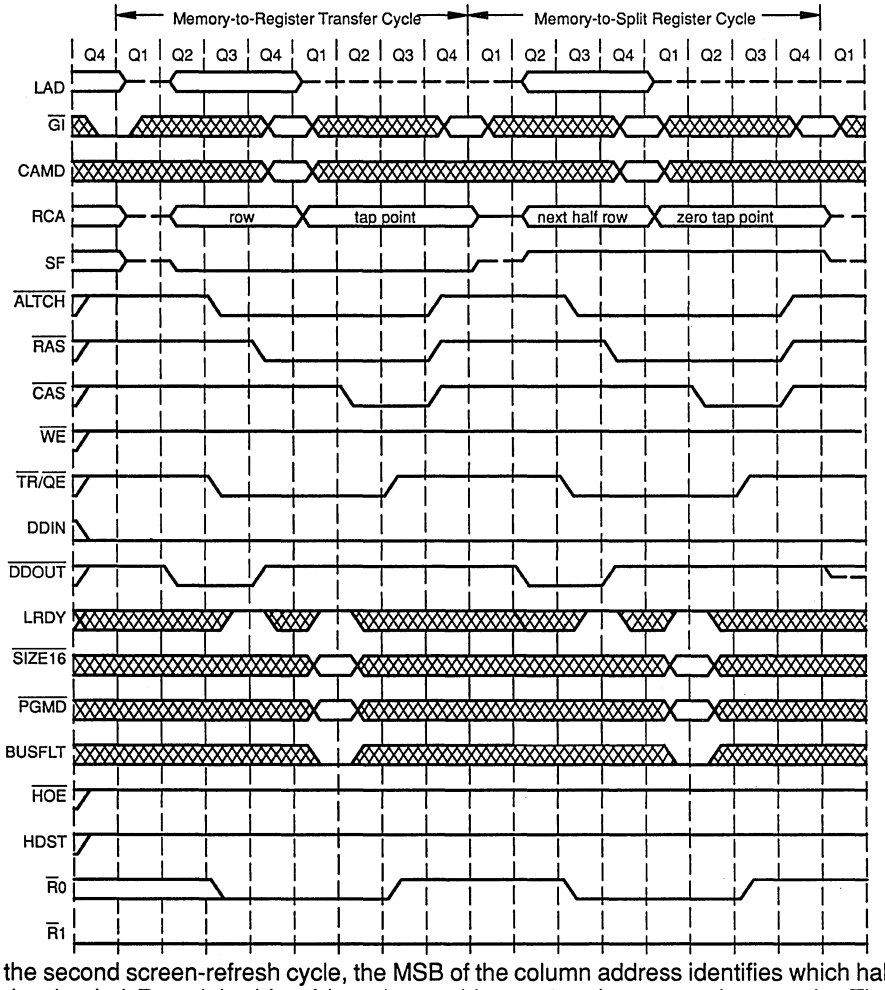
You must set SSV[[DPYCTL]] to a 1 to allow the TMS34020 to perform split-serial-register screen refreshes when they are scheduled by SCOUNT. The DPYMSK register must also be programmed to ensure that SCOUNT is loaded with the correct portion of the full logical address (Section 9.15, page 9-51, describes this). Figure 9–21 shows the split-serial-register VRAM mid-line-reload screen-refresh memory cycle. The generation of the addresses output during the register-to-memory cycles described in this section is also covered in Section 9.15.

Figure 9–21. Local-Memory Split-Serial-Register VRAM Memory-to-Register Cycle



When midline reload is enabled, an extra screen refresh is scheduled during horizontal blanking. As for any horizontal-blanking period, an ordinary memory-to-register cycle loads the specified row of VRAM into each of the VRAM serial registers. This loads **both** halves of the split serial register. However, if the tap point is more than halfway along the row, then the idle half of each serial register (loaded from the lower half of the VRAM row) will not contain data from the next row of VRAM to be displayed. To ensure that the idle half contains the **next** half row, and not the **previous**, a split-serial-register screen refresh with the address of the **next** half-row is generated immediately after the ordinary horizontal-blanking screen refresh. This is the same type of cycle used for the midline-reload screen-refresh cycles, and loads only **half** of each of the VRAM serial registers. Figure 9–22 shows these memory cycles.

Figure 9–22. Local-Memory Split-Serial-Register VRAM Horizontal-Blanking Memory-to-Register Cycles



Note: During the second screen-refresh cycle, the MSB of the column address identifies which half of the serial register is reloaded. Remaining bits of the column address = 0 and represent the tap point. The row address output now addresses the memory row containing the next half row to be transferred to the split serial reg.

Note:

Normally, SCLK should be low during blanking. However, VRAMs require an SCLK pulse to latch the tap-point address into an internal counter. As Figure 9–22 shows, 2 tap-point addresses are provided during horizontal blanking. To ensure correct operation, you must provide a single SCLK pulse to the VRAMs between $\overline{TR}/\overline{QE}$'s low-to-high transition at the end of the first screen-refresh cycle, and $\overline{RAS}$'s high-to low transition at the beginning of the second screen-refresh cycle. (This is in accordance with the timing specifications for VRAMs.) Failure to do this causes the TMS34020 to ignore the tap-point address provided during the screen-refresh cycle.

9.13.3 Why Use Midline Reload?

If serial-register transfers during horizontal blanking are the only type of transfers available in a system, the display memory must be arranged in a way that allows the VRAM serial registers to be loaded with enough information for an entire scan line during each horizontal-blanking interval.Each row of VRAM must contain at least as many bits as the number of pixels on one screen line. Depending on the number of pixels, this means that there can potentially be a significant amount of the display memory that cannot be used.

If, however, the VRAM serial registers can be reloaded during the active display time as well as during blanking, there is no need for any of the display memory to be wasted. Also, because each row of the display memory no longer needs to contain enough information for a whole screen line. The number of memory rows required may allow the display memory to be contained in fewer banks. Section 8.17 (Double-Buffered Display Example, page 8-57) illustrates how using midline reload can reduce the amount of VRAM required for display memory.

Although midline reload can reduce the amount of VRAM required, there is an inherent trade-off: The screen pitch may not be a power of 2. If XY addressing is being used for graphics instructions (as it typically is), this increases the time taken to convert an XY address to a linear address:

- If the screen pitch is a power of 2, each conversion takes 3 machine cycles.
- If the screen pitch is the sum of two powers of 2, each conversion takes 4 machine cycles.
- If the screen pitch is an arbitrary value, each conversion takes 15 cycles.

In addition to the CVXYL instruction, which performs one conversion, graphics instructions that have XY operands (either explicit or implicit) automatically perform XY-to-linear conversions. The following instructions are affected:

- DRAV performs one conversion.
- FILL XY and PFILL XY perform one conversion.

- LINE performs one conversion.
- PIXBLT instructions perform one conversion for each XY operand.
- PIXT instructions perform one conversion for each XY operand.

Chapter 12, *Graphics Instructions and Operations*, provides precise details of how the execution time for each instruction is affected by using a screen pitch that is not a power of 2.

9.13.4 VRAM Bulk Initialization

VRAMs may be rapidly loaded with an initial value using a special feature that converts pixel accesses into register transfers. This rapid loading method is referred to as **bulk initialization**. When CST[[DPYCTL]] is set to 1, the TMS34020 converts all reads and writes of pixel data into register-transfer cycles. When CST = 0, pixel accesses are performed in the normal way.

When CST = 1, the TMS34020 can initiate register-transfer cycles under explicit program control. By performing a series of such cycles, some or all of the display memory can be set to an initial background color or pattern very rapidly (in a small fraction of a frame time). First, the VRAM serial registers are loaded with an initial value. The video memory is then set to the initial color or pattern one row at a time by writing the serial register contents to the memory.

The row address output during the register-transfer cycle determines which row of memory is involved in the transfer. The direction of the transfer is determined by whether the cycle is a read or write. A write cycle (such as a PIXT from a general-purpose register to memory) is converted to a VRAM register-to-memory cycle. Similarly, a read cycle (such as a PIXT from memory to a general-purpose register) is converted to a VRAM memory- to-register cycle.

The value of the CST bit affects only pixel transfers. Other data accesses and instruction fetches are unaffected.

Before bulk initialization, the VRAM serial registers must be loaded with the solid color or pattern with which the display memory will be loaded. This can be accomplished by

- serially shifting bits into the serial registers, or
- loading a row of display memory with the color or pattern using a series of normal pixel writes (CST = 0), then loading the contents of this row into the VRAM serial registers by means of a PIXT memory-to-register instruction (CST = 1).

To further speed bulk initialization, you can make a series of transfers more rapidly by using a single FILL instruction in place of a series of PIXTs. Select the fill region so that each pixel write cycle generates a new row address. The fill region should be 1, 2, 3, or 4 bytes wide. Plane masking and transparency

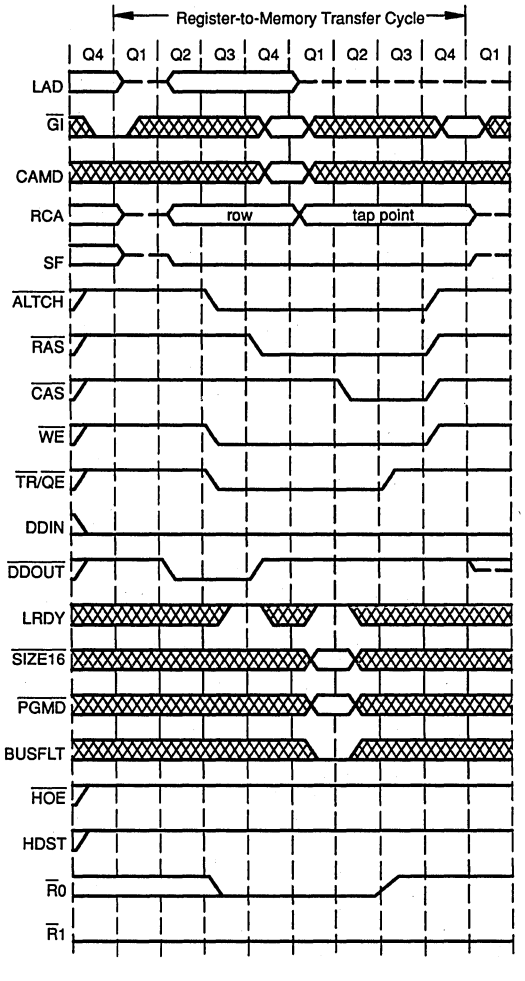
are disabled, and the pixel-processing replace option is selected. This ensures that the TMS34020 need not perform any read-modify-write operations, and each row is addressed only once during the course of the fill operation.

The number of bits of the display memory that are altered by a single register-to-memory transfer cycle is calculated by multiplying the number of VRAM devices by the number of serial register bits in each device.

9.13.5 Video Capture

If VCE[[DPYCTL]] is set to 1, all screen-refresh cycles are executed as register-to-memory cycles, rather than memory-to-register cycles. Figure 9–23 shows a memory-to-register cycle. This feature is useful if the VRAMs are used to capture video data by shifting data bits *into* them, rather than out.

Figure 9–23. Local-Memory Register-to-Memory Transfer Cycle



Do not use the video-capture feature with midline reloads. The register-to-memory cycle overwrites an entire VRAM row from the VRAM serial register. In the case of the first midline reload that follows the end of horizontal blanking, not all the bits in the serial register would need transferring—only those shifted in since the end of blanking.

9.13.6 Disabling Screen Refreshes

In order for the TMS34020 to perform screen-refresh memory cycles initiated by the video-timing logic or the midline-reload logic, SRE[[DPYCTL]] (screen refresh enable) must be set to a 1. While it is 0, the TMS34020 still increments the screen-refresh address during horizontal blanking, but no actual screen-refresh memory cycles occur.

This can be useful for inserting a video image from another source onto the image generated by the TMS34020. If a section of the screen image is to be provided by another source, SRE is set to 0 during the line immediately preceding the inserted image. The TMS34020 then stops refreshing the screen, allowing another device to perform this function.

To revert to the TMS34020-generated image, set SRE to 1 again during the last line of the inserted image. The TMS34020 then restarts screen refreshes, beginning in the horizontal-blanking interval immediately following the last line of the inserted image. Because the screen-refresh address is incremented every line, regardless of the value of SRE, the first line displayed will be the same line that would have been displayed at this time if the screen refreshes had never been switched off.

If midline-reload screen refreshes are also used, the SSV bit should be set to 1 at the same time as the SRE bit. However, you should note that if the TMS34020's SCLK input is clocking during the last line of the inserted image after the SSV bit has been set to 1, a midline-reload screen refresh could occur before the end of the line. Holding the SCLK input low while the inserted image is being output to the screen eliminates this problem.

9.14 Scheduling Screen-Refresh Cycles

The horizontal-blanking period should be long enough to ensure that the TMS34020 has completed the serial-register transfer by the time blanking ends and data starts shifting out to the screen again. The delay from the start of horizontal blanking until the start of the screen-refresh memory cycle is called the **screen-refresh latency** and is determined by the TMS34020's memory controller. It consists of 2 elements:

- The time necessary to synchronize and recognize the screen-refresh request from the video timing logic, and
- The time to complete the memory cycle currently in progress when the request is recognized.

The synchronization and recognition time is, at worst, 3.5 machine states and, at best, 2.5 machine states. Table 9–1 shows the maximum and minimum screen-refresh latency.

Table 9–1. Screen-Refresh Latency

Measurement	Value
Minimum screen-refresh latency	$2.5T$
Maximum screen-refresh latency:	
❑ DRAM refreshes *enabled*	$(5.5 + N)T$
❑ DRAM refreshes *disabled*	$(4.5 + N)T$

Key: T local clock period
N maximum number of wait states per memory cycle

Page mode bursts are terminated by the screen-refresh request and restart after the screen-refresh cycle from the next address in the sequence. In this way, they do not delay the start of the screen-refresh cycle.

A new memory access (in which the full address is output on the LAD bus) that starts, or is in a wait state, in the machine state just before the screen refresh is recognized cannot be aborted until the first word of data has been transferred. This, therefore, delays the start of the screen refresh by $1+N$ machine states. However, a DRAM refresh (which takes three machine states minimum) will delay the start of the screen refresh by $2+N$ machine states.

The length of time taken for the screen refresh to complete, once started, depends on the number of wait states used. The minimum time is 2 machine states. For systems using split-serial-register VRAM midline reloads, the horizontal-blanking screen-refresh cycle is actually 2 distinct memory cycles (see Section 8.13.2). In this case, the minimum duration of the horizontal-blanking screen refresh is 4 machine states, and both of the memory cycles can have wait states inserted. The maximum possible time from the beginning of horizontal blanking to the end of the screen-refresh cycle defines the minimum horizontal-blanking duration. Table 9–2 summarizes this.

Table 9–2. Minimum Horizontal-Blanking Duration

Measurement	Value
Minimum horizontal-blanking duration (*without* split-serial-register VRAM reloads):	
❑ DRAM refreshes *enabled*	$(7.5 + N)T$
❑ DRAM refreshes *disabled*	$(6.5 + N)T$
Minimum horizontal-blanking duration (*with* split-serial-register VRAM reloads)	
❑ DRAM refreshes *enabled*	$(9.5 + 2N)T$
❑ DRAM refreshes *disabled*	$(8.5 + 2N)T$

Key: T local clock period
N maximum number of wait states per memory cycle

9.15 Generating Screen-Refresh Addresses

This section describes methods for programming the TMS34020 to generate the addresses for all types of screen refreshes.

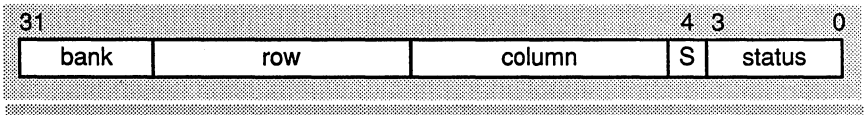
The address output during a screen-refresh cycle identifies the first of a series of pixels to be output to the monitor. In the case of a horizontal-blanking screen refresh, this is the first pixel on the next scan line. For a midline reload, it is the first pixel of the next row of VRAM. The TMS34020 outputs a 28-bit logical address on LAD31—LAD4. This address can be broken down into several fields:

- the 16-bit word select on bit 4,
- the column-address and row-address fields, and
- a bank-select field or fields (optional). The bank-select field(s), if required, can be at either end of the address (less significant than the column address, or more significant than the row address, or both).

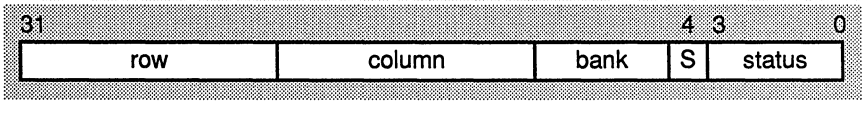
The precise positioning of all these fields (except for the 16-bit word-select bit), must be determined by system requirements. Figure 9–24 shows this.

Figure 9–24. Screen-Refresh Address Fields

(a) Bank select at the most significant end of the address



(b) Bank select at the least significant end of the address



Notes: 1) The screen refresh status code is output on LAD3—LAD0.
2) The S bit is always 0 unless dynamic bus sizing is employed to access 16-bit wide VRAMs, and is always 0 during screen-refresh cycles.
3) The address may not extend all the way to address bit 31 as shown. In a typical system, the full address range is unlikely to be required.

The row address output during the screen-refresh cycle specifies the row in memory to be loaded into the serial register internal to the VRAM. The column address determines which bit of that row is shifted out of the VRAM serial register first.

9.15.1 Horizontal-Blanking Screen-Refresh Addresses

The portion of the display memory actually output to the monitor is referred to as **on-screen memory**. To generate the appropriate address for each horizontal-blanking screen refresh, it is necessary to know

- The **starting location** of the **on-screen** memory.

  Typically, the starting location of the on-screen memory is the address of the pixel appearing in the top left-hand corner of the display. Load this address into SRST[[DPYST]].

- The difference in 32-bit memory addresses between 2 vertically adjacent pixels on the screen; this is called the **screen pitch**.

  Screen pitch is the difference between the memory addresses of the first pixels of 2 consecutive scan lines. For systems not using midline reload, this is also the width of the display memory because each row of VRAM must contain all the information for a whole scan line. Load the screen pitch into SRINC[[DINC]].

  In some systems, the screen may be refreshed starting from the bottom of the screen. If this is the case, load SRINC with the 2s complement of the screen pitch so that the screen-refresh address is decremented between each horizontal-blanking screen refresh.

- The **vertical magnification** (Y-zoom) of the display. This is the number of times each scan line in the display memory will be displayed on the screen. Typically, each line is displayed only once; the remainder of this discussion assumes this is the case. For more information about the Y-zoom feature, refer to Section 9.15.5 on page 9-56.

- Whether or not the display is **interlaced** (NIL[[DPYCTL]] = 0).

The TMS34020 automatically calculates the address required for each screen refresh and stores the address in SRNX[[DPYNX]].

The DPYMSK register is required for midline-reload screen refreshes (see Section 9.15.4 on page 9-55). However, if you are not using midline reload, there is no need to program DPYMSK.

9.15.2 Screen-Refresh Addressing Sequence for Noninterlaced Displays

If NIL[[DPYCTL]] = 1, the TMS34020 generates screen-refresh addresses in the following sequence:

1) At the beginning of the vertical-blanking period, the address specified by SRST[[DPYST]] is loaded into SRNX[[DPYNX]].

2) In the horizontal-blanking period just before the first line of the frame, the addresses contained in SRNX is output to the VRAMs during the screen-refresh cycle. SRINC[[DINC]] is then added to this address, and the result is loaded back into SRNX.

3) During the next horizontal-blanking period, the contents of SRNX are output during the screen-refresh cycle. SRINC is then added to this, and the result is loaded back into SRNX. This process then repeats during all subsequent horizontal-blanking periods until the end of the frame.

Figure 9–25 illustrates this sequence in the form of a flow chart.

9.15.3 Screen-Refresh Addressing Sequence for Interlaced Displays

If NIL[[DPYCTL]] = 0, the TMS34020 generates screen-refresh addresses in 2 sequences—one for the odd field and one for the even field. These sequences are described below, and illustrated in the form of a flow chart in Figure 9–25.

1) At the beginning of the vertical-blanking interval between the odd and even fields, the address specified in SRST[[DPYST]] is added to *half* SRINC[[DINC]], and the result is loaded into SRNX[[DPYNX]]. This is because vertical blanking ends halfway across the first line of the even field, and the first pixel displayed is half a line away from the on-screen memory start address (refer back to Figure 9–14).

2) In the last horizontal-blanking period before the first line of the even field, the address contained in SRNX is output to the VRAMs during the screen-refresh cycle. Then, *twice* SRINC is added to SRST, and the result is loaded into SRNX. SRNX then points to the first pixel on the third scan line because alternate lines are scanned in interlaced mode.

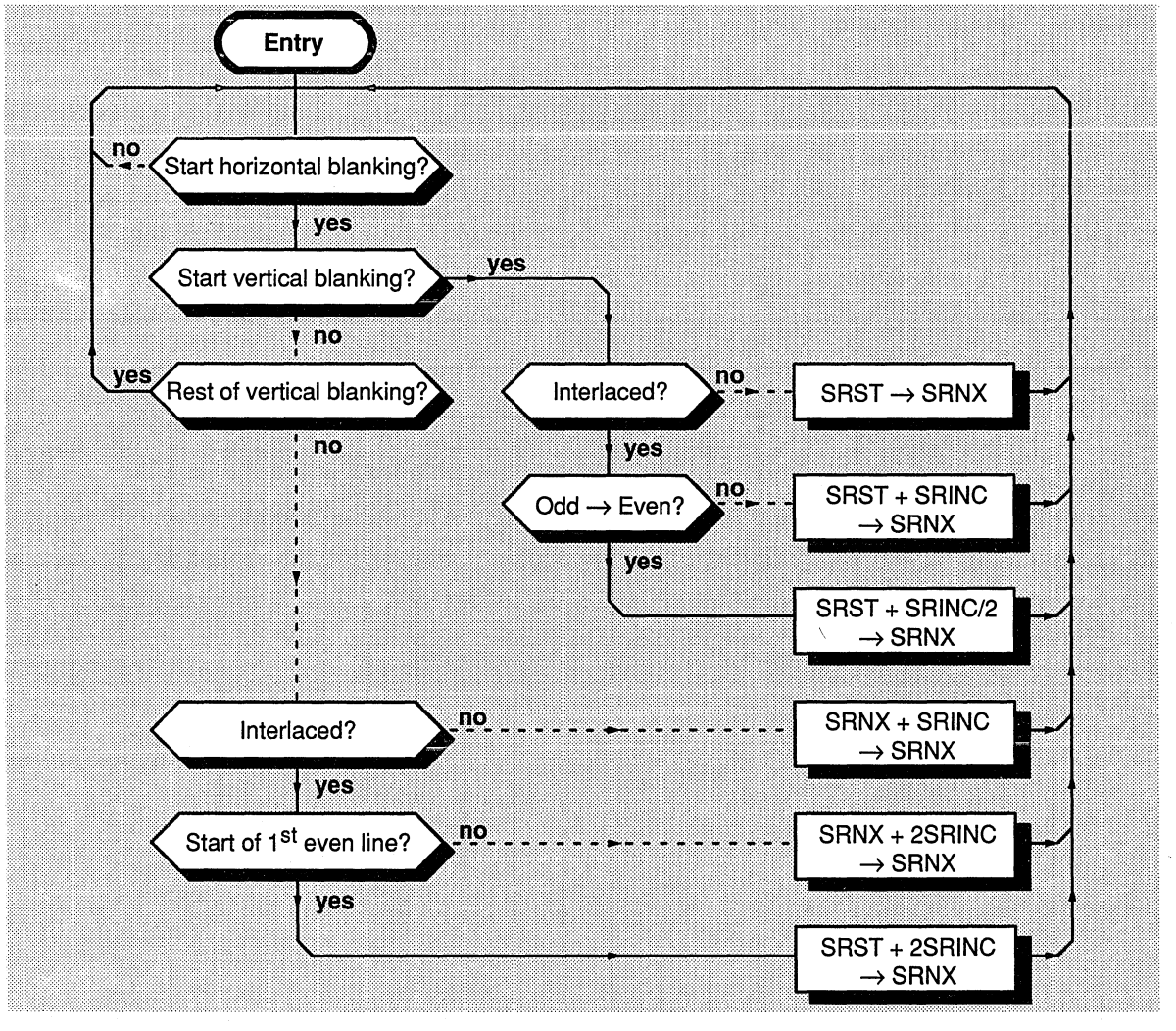
3) During the next horizontal-blanking period, the contents of SRNX are output during the screen-refresh cycle. Twice SRINC is then added to this, and the result loaded back into SRNX. This process then repeats during all subsequent horizontal-blanking periods until the end of the field.

4) At the beginning of the vertical-blanking interval between the even and odd fields, the address specified in SRST[[DPYST]] is added to SRINC, and the result is loaded into SRNX. SRNX then points to the first pixel on the second scan line (the first line scanned in the odd field).

5) In the horizontal-blanking period just before the first line of the odd field, the address contained in SRINC is output to the VRAMs during the screen-refresh cycle. Then, *twice* SRINC is added to SRNX, and the result is loaded back into SRNX. SRNX then points to the first pixel on the fourth scan line.

6) During the next horizontal-blanking period, the contents of SRNX are output during the screen-refresh cycle. Twice SRINC is then added to this, and the result is loaded back into SRNX. This process then repeats during all subsequent horizontal-blanking periods until the end of the field.

Figure 9–25. Screen-Refresh Address Generation Flow



9.15.4 Midline-Reload Screen-Refresh Addresses

If midline-reload screen refreshes are enabled, the column address part of the logical address must be identified, so that

- SCOUNT can be loaded with the VRAM tap-point to enable it to correctly schedule the midline-reload screen-refresh cycles.
- The correct row address can be isolated and output during the midline-reload screen refreshes.

DPYMSK stores a mask of contiguous 1s ,which correspond to the column address portion of SRST[[DPYST]] or SRNX[[DPYNX]] .

This mask determines which bits of the logical address are loaded into SCOUNT. SCOUNT is loaded with the tap-point portion of the address loaded into SRNX whenever SRNX is loaded during horizontal blanking.

256K×4 (1 Mbit) VRAMs have 9 column address bits and 9 row address bits (512 rows and 512 columns), but because they also have split serial registers, each row of VRAM is conceptually split into 2; each midline-reload screen-refresh cycle loads only half a VRAM row into the relevant half serial register. Thus, the address must be incremented at the most significant column-address bit, rather than the least significant row bit (to select between the 2 halves of the VRAM). Therefore, when considering midline reload and DPYMSK, the most significant column-address bit of the VRAM is not considered as being part of the tap point, but as the least significant half-row address bit.

When a midline-reload screen refresh occurs, the address output to the VRAMs is the address of the first pixel in the next half-row of VRAM. Therefore, the column portion of the address is 0.

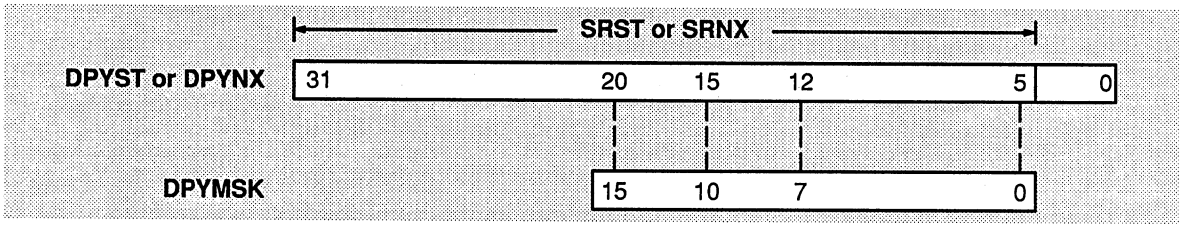
The mask stored in DPYMSK is also used to isolate the half-row address from the full logical address (all bits more significant than that mapped to by the most significant 1 in DPYMSK are considered to be the half-row address). A non user-accessible register is loaded with the row address portion of the address loaded into SRNX whenever SRNX is updated during horizontal-blanking screen refreshes. This address is then incremented to point to the next VRAM half-row at the bit position mapped to the rightmost 0 at the most significant end of DPYMSK.

If a midline-reload screen refresh occurs, this address is output to the VRAMs and then incremented to point to the next half row. All address bits less significant than the half-row address are masked by DPYMSK, so that they are output as 0s. This corresponds to a tap-point address of 0, thus pointing to the first pixel in the next half row. This procedure then repeats every time a midline-reload screen refresh is scheduled on the current scan line.

There is a 5-bit offset between DPYMSK and the logical address. Thus, if the column address is 8 bits long, starts at bit 7 of the logical address, and ends

at bit 14, then DPYMSK should be loaded with a contiguous field of 1s between bits 2 and 9 (inclusive). All other bits of DPYMSK should be 0. Figure 9–26 shows the mapping.

Figure 9–26. Mapping Relationship Between DPYMSK, DPYST, and DPYNX



Note: Bit 0 of DPYMSK maps to bit 5 of DPYST, and so on.

Bit 13 is the lowest address bit that can be incremented to change the half-row address for midline-reload screen refreshes. The most significant 1 in DPYMSK must be at bit 7 or higher of DPYMSK.

Bit 16 is the highest address bit that can be incremented for midline-reload screen refreshes. The most significant 1 in DPYMSK must not be any higher than bit 10 of DPYMSK.

9.15.5 Display Magnification and Y-Zoom

It is often desirable to magnify part of an image on the display. To do this, you can make the pixels appear larger on the display by increasing their physical X and Y dimensions.

- The X dimensions of pixels can be increased by reducing the rate at which they are fed to the monitor. Fewer pixels are displayed on a scan line, so each pixel is wider.
- The Y dimension of pixels can be increased by repeating a scan line multiple times. If the information for one scan line is repeated on consecutive lines, the pixels appear taller.

The X dimension must be controlled by external hardware that shifts serial pixel data to the monitor. The Y dimension, however, can be controlled by the TMS34020, which provides direct support for changing the Y dimension.

To repeat a scan line multiple times, simply add 0 to SRNX (instead of adding SRINC). Control for this is provided via YZINC[[DINC]] and YZCNT[[DPYNX]]. After each horizontal-blanking screen-refresh cycle, YZINC is added to YZCNT. As Table 9–3 shows, YZINC is normally a power of 2 between 0 and 16. This means that periodically (when YZCNT + YZINC = 32), YZINC overflows to 0 because it is only a 5-bit field.

- If YZCNT=0 during the horizontal-blanking screen-refresh cycle, SRNX is incremented in the normal way (as described in Sections 9.15.2 and 9.15.3, and summarized in Figure 9–25).

- If YZCNT≠0 during the horizontal-blanking screen-refresh cycle, 0 is added to SRNX or SRST instead of SRINC, 2×SRINC, or SRINC/2 (as outlined in Figure 9–25) after the horizontal-blanking screen refresh cycle. During each vertical-blanking interval, YZCNT is reset to YZINC.

Table 9–3. Y-Zoom Control

YZINC					Zoom	
4	3	2	1	0	Factor	Description
0	0	0	0	0	0	No repetitions of scan lines
1	0	0	0	0	2	Repeat scan line 2 times
0	1	0	0	0	4	Repeat scan line 4 times
0	0	1	0	0	8	Repeat scan line 8 times
0	0	0	1	0	16	Repeat scan line 16 times
0	0	0	0	1	32	Repeat scan line 32 times

Clearing YZINC to 0 causes no Y-zoom because YZCNT is always 0. Conversely, when YZINC=1, YZCNT equals 0 only once every 32 horizontal-blanking intervals.

If YZINC is set to an odd value (such as 31), YZCNT will never be 0, and the scan line will be repeated indefinitely. This could be useful if you wish to clear the screen temporarily while transferring a new image to the display memory. A particular line could be transferred to the entire screen without the need for bulk erasing the VRAMs of the display memory.

To ensure that YZCNT + n×YZINC (where n is the zoom factor) always overflows to 0, YZCNT should always be cleared to 0 when YZINC is changed. This is the only time it should be necessary for you to write to YZCNT.

9.15.6 Panning the Display

By changing the value of the column-address portion of SRST[[DPYST]] between frames, you can pan the on-screen image horizontally across the display memory. By changing the value of the row-address portion of SRST between frames, you can pan the on-screen image vertically up and down the display memory.

SRST is loaded into SRNX[[DPYNX]] at the beginning of vertical blanking. To affect the position of the screen image in the display memory for the next frame, SRST should be changed before the beginning of vertical blanking at the end of the current frame.

Chapter 10

Communicating with a Coprocessor

The TMS34020 can communicate with a coprocessor through the **coprocessor interface**, which consists of special instructions and bus cycles. A coprocessor extends TMS34020 capabilities, providing hardware and software enhancements without incurring significant control overhead.

A coprocessor differs from a typical peripheral device, which is usually mapped to reside within the TMS34020's address space. When using peripherals, you must write code to provide the communications protocol between the TMS34020 and the peripheral device. However, the TMS34020's coprocessor interface extends the base architecture of the TMS34020, adding instructions that implement the communication protocol in hardware. Thus, the TMS34020 can use the coprocessor's data types and registers without treating the coprocessor as a separate device.

This chapter provides the following details about the coprocessor interface:

Note:

Before reading this chapter, you should be familiar with the TMS34020's local-memory interface. If you are not, please read Chapter 8.

10.1 Related Signals

The coprocessor interface uses a subset of the local-memory interface signals. Although these are primarily local-memory signals, their functions may differ when used for the coprocessor interface. Chapter 2 describes these signals in detail; they are summarized below for your convenience.

Signals	Descriptions	I/O
$\overline{\text{ALTCH}}$	is the address latch signal. $\overline{\text{ALTCH}}$'s high-to-low transition latches the current coprocessor instruction (and status code) or address. During data transfers from the coprocessor to TMS34020 memory, a high level on $\overline{\text{ALTCH}}$ enables data to the LAD bus.	O
BUSFLT	is the bus-fault signal.If external logic detects an error or fault in the current cycle, it asserts BUSFLT high. BUSFLT is used with LRDY to generate bus-retry cycles. A coprocessor must monitor this signal to determine the status of cycle termination.	I
$\overline{\text{CAS0}}$—$\overline{\text{CAS3}}$	are the column-address strobe signals. They can be connected to a coprocessor to control data transfers.	O
LAD0—LAD31	form the multiplexed local address/data bus. The coprocessor interface uses this bus to transfer coprocessor instructions, data, and the memory addresses that are used for the transfers.	I/O
LCLK1, LCLK2	are the local output clocks. These signals drive the coprocessor logic, providing control signals that are synchronous to the TMS34020.	O
$\overline{\text{LINT1}}$, $\overline{\text{LINT2}}$	are the TMS34020's local interrupt requests that a coprocessor can use to interrupt the TMS34020.	I
LRDY	is the local ready signal. External circuitry (which may be controlled by a coprocessor) can drive LRDY low to prevent the TMS34020 from completing a local-memory cycle. LRDY is used with BUSFLT to indicate retries and bus faults. A coprocessor must monitor this signal to determine the status of cycle termination.	I
SF	is the special function signal that coprocessors monitor to distinguish between instruction cycles and address cycles.	O
$\overline{\text{WE}}$	is the write-enable signal that identifies the direction of a data transfer.	O

10.2 Overview of the Coprocessor Interface

The coprocessor interface provides both software and hardware features that help the TMS34020 communicate with a coprocessor. These features take the form of extensions to the TMS34020's instruction set and local-memory interface. Special coprocessor instructions may be used, which, in turn, invoke special local-memory interface cycles to pass commands and data between the TMS34020, local memory, and a coprocessor.

The instructions fall into two categories:

- **Generalized instructions** for use with any coprocessor.

  The TMS34020 supports several instructions that allow you to pass commands and data (when appropriate) to a coprocessor, in order that the coprocessor can execute the command. You pass commands and data as parameters to the TMS34020 instruction. Available coprocessor commands depend upon your coprocessor's requirements and upon the operation you wish it to perform. Table 10–1 lists these TMS34020 instructions; Chapter 13 describes them in detail.

- **Specialized instructions** for use with the TMS34082 floating-point processor.

  These are special cases of the generalized instructions, where the TMS34082 command is explicitly coded into the instruction opcode. Chapter 14 describes the TMS34082 instruction set.

Note:

The term *instruction* refers to an assembly-language instruction that the TMS34020 executes. The terms *command* and *coprocessor command* refer to assembly-language instructions that a coprocessor supports and executes.

Table 10–1. TMS34020 General Coprocessor Instructions

Mnemonic	Description
CEXEC	Execute coprocessor internal operation
CMOVCG	Move, coprocessor to TMS34020
CMOVCM	Move, coprocessor to local memory
CMOVCS	Move, coprocessor to status register
CMOVGC	Move, TMS34020 to coprocessor
CMOVMC	Move, local memory to coprocessor

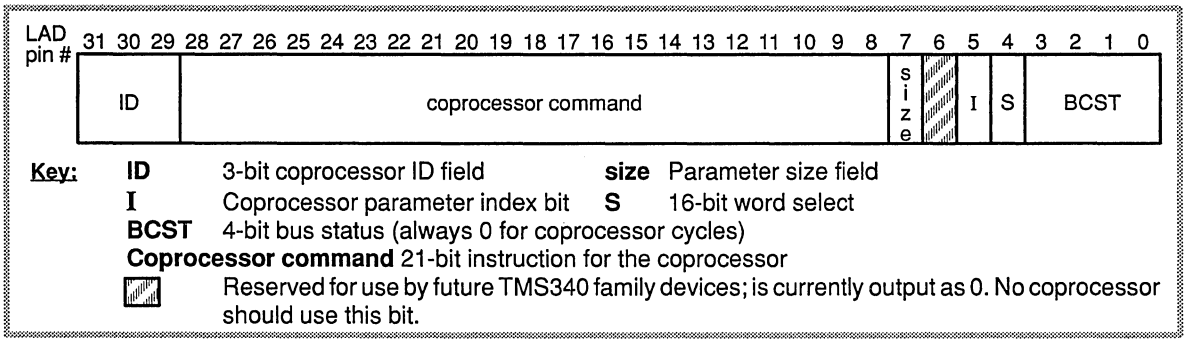
The coprocessor instructions can invoke five types of local-memory coprocessor cycles. Each cycle passes a command to the coprocessor; four of these cycles also transfer data. With a single instruction, the TMS34020 can pass a command to a coprocessor, and any associated data transferred to or from the coprocessor.

- The **coprocessor internal command** cycle passes a command to a coprocessor.
- The **TMS34020 to coprocessor transfer** cycle passes a command to a coprocessor, then performs one of the following data transfers between the TMS34020 and the coprocessor:
  - Move one 32-bit parameter
  - Move two 32-bit parameters
  - Move one 64-bit parameter
- The **coprocessor to TMS34020 transfer** cycle passes a command to a coprocessor, then performs one of the following data transfers between the coprocessor and the TMS34020:
  - Move one 32-bit parameter
  - Move two 32-bit parameters
  - Move one 64-bit parameter
  - Move 4 bits to the N, C, Z, and V bits of the TMS34020's status register
- The **memory to coprocessor transfer** cycle passes a command to a coprocessor, then performs one of the following data transfers of up to 32 32-bit words between the local memory and the coprocessor:
  - Move the number of 32-bit words specified in the coprocessor instruction using postincrement
  - Move the number of 32-bit words specified in the coprocessor instruction using predecrement
  - Move the number of 32-bit words specified in a register using postincrement
- The **coprocessor to memory transfer** cycle passes a command to a coprocessor, then performs one of the following data transfers of up to 32 32-bit words between the coprocessor and the local memory:
  - Move the number of 32-bit words specified in the coprocessor instruction using postincrement
  - Move the number of 32-bit words specified in the coprocessor instruction using predecrement

10.3 Format of Commands Passed to a Coprocessor

The TMS34020 passes a command to a coprocessor over the LAD bus. The command replaces the address usually output during a local-memory cycle's address/status subcycle. In addition to the actual coprocessor command, some auxiliary control bits and the coprocessor operation status code are also output at this time, as Figure 10–1 shows.

Figure 10–1. Coprocessor Instruction Format



The information for the ID, the command, and the size bit are provided as arguments to the coprocessor instruction. The precise form of this information is determined by your processor's requirements and the operation you wish it to perform.

10.3.1 Coprocessor ID

The ID bits allow you to address multiple coprocessors within a system. Each coprocessor should respond to commands only when the ID output matches the coprocessor's assigned ID. If you do not supply an ID argument with the coprocessor instruction, the default ID is 000_2. You can change the default ID with the .coproc assembler directive (refer to the *TMS340 Family Code Generation Tools User's Guide* for information about .coproc).

Table 10–2 lists suggested ID code assignments. Using these IDs promotes compatibility across different hardware configurations.

000_2—011_2 Use the first 4 coprocessor ID numbers to identify up to 4 TMS34082 floating-point processors.

100_2 Use as a **broadcast ID** to which all coprocessors should respond.

101_2 & 110_2 Reserve for compatibility with future devices.

111_2 Use for addressing your own coprocessor (in a manner that conforms to the TMS34020 coprocessor interface). This allows the TMS34020 to provide control through custom instructions.

Table 10–2. Suggested Coprocessor ID Assignments

ID	Description	ID	Description
000_2	TMS34082$_0$ is to execute a command	100_2	All coprocessors are to execute command
001_2	TMS34082$_1$ is to execute a command	101_2	Reserved for future devices
010_2	TMS34082$_2$ is to execute a command	110_2	Reserved for future devices
011_2	TMS34082$_3$ is to execute a command	111_2	User-defined coprocessor to execute a command

Note:

You can adopt other assignments as required; however, all coprocessors should respond to the broadcast ID (100_2).

10.3.2 Coprocessor Command

The identified coprocessor uses the 21-bit coprocessor command to determine what operation it should perform. The command itself is specific to the coprocessor that must execute the command.

Some coprocessors may not require all 21 bits to fully specify a command. If this is the case, other information (such as data, status, or other coprocessor control) may also be coded into this area.

At least part of the command should inform the coprocessor of what type of data transfer is to follow the command (if any) and the number of 32-bit words to be transferred.

10.3.3 Coprocessor Parameter Size (size)

During direct data transfers between the TMS34020 and a coprocessor, data is transferred during the data subcycle (immediately following the command). Section 10.4.2 (page 10-8) discusses direct and indirect data transfers in more detail. You can transfer 0, 1, or 2 32-bit words. The size bit identifies the size of the parameters passed to or from the coprocessor:

size=0 Transfer 1 or 2 32-bit parameters.

size=1 Transfer 1 64-bit parameter.

If no data is transferred following the command, size=0.

In either case, the data can be an integer or floating-point value. You can use size in conjunction with other bits of the coprocessor command to specify which of these formats is appropriate for the data.

10.3.4 Coprocessor Parameter Index (I)

During direct data transfers between the TMS34020 and a coprocessor, data is transferred during a data subcycle (immediately following the command). Section 10.4.2 (page 10-8) discusses direct and indirect data transfers in more detail. You can transfer 0, 1, or 2 · 32-bit words. The I bit identifies which of the words are transferred:

I=0 The first (and possibly only) 32-bit word are transferred immediately following the command.

I=1 The second 32-bit word are transferred immediately following the re-issued command.

If no data is transferred following the command, I=0.

Initially, I is output as a 0. It is output as a 1 only when two 32-bit words are transferred. If the second transfer cannot be made using page mode. In this case, a complete memory cycle (consisting of address/status and data subcycles) must be generated for the second 32-bit word to be transferred. During the address/status subcycle for this transfer, I is 1. The rest of the command is the same at this time.

The second transfer cannot be performed using page mode if

- The coprocessor does not support page-mode operation and asserts $\overline{\text{PGMD}}$ high during the first data transfers.
- A high-priority local-memory request (such as a VRAM serial-register transfer) is requested while the first 32-bit word is being transferred; it is performed before the second 32-bit word is transferred.

10.3.5 16-Bit Word Select (S)

The S bit output on LAD4 at the beginning of the coprocessor cycle is always 0 (low). The S bit is 1 during cycles in which the TMS34020 accesses 16-bit memory devices; however, the TMS34020 does not support coprocessor accesses to 16-bit-wide memory devices. TMS34020-controlled coprocessor cycles are restricted to memory accesses of 32-bit memory devices.

10.3.6 Coprocessor Status Code (BCST)

This code is 0000_2. It is output during the address/status subcycle of all local-memory coprocessor cycles.

10.4 Local-Memory Coprocessor Cycles

This section describes the local-memory coprocessor cycles for interfacing to a coprocessor. A coprocessor must be able to recognize these cycles in order to properly receive and transmit data through the coprocessor interface. During the address/status subcycle of these cycles, the TMS34020 outputs the coprocessor operation status code (0000_2) on LAD0—LAD3.

In these examples, the TMS34020 controls the local-memory interface and provides all controls necessary for transferring commands and data to and from a coprocessor. If you want the coprocessor to control the local-memory interface through the TMS34020's multiprocessor interface, refer to Chapter 11.

10.4.1 Passing Commands to a Coprocessor

As Section 10.3 describes, the TMS34020 passes a command to a coprocessor over the LAD bus during the address/status subcycle. The coprocessor must distinguish a command from an ordinary memory address by

- the status code of 0000_2 output on LAD0—LAD3 and
- a high level on the SF pin.

The command should be latched on $\overline{\text{ALTCH}}$'s high-to-low transition when the these conditions occur. Because a coprocessor command is output on the LAD bus in essentially the same manner as a normal memory address, parts of the command are also output on the RCA bus in the same manner as the row and column portions of a regular address.

10.4.2 Transferring Data to or from a Coprocessor

After passing a command to a coprocessor, data can be transferred using one of these methods:

- **Direct transfer** between the TMS34020 and a coprocessor. This transfers data (via LAD) in the data subcycle, immediately following the command.
- **Indirect transfer** between local memory and a coprocessor. The TMS34020 provides the addresses and control signals for the memory. No data is transferred immediately after the command; the TMS34020 generates another memory cycle sequence, beginning with an address/status cycle in which the memory address for the transfer is output.

Note:

All local-memory coprocessor cycles are implemented as 32-bit data transfer operations. The TMS34020 does not support data transfers between 16-bit memory and a coprocessor (using the TMS34020's dynamic bus-sizing feature). The TMS34020 ignores the state of the $\overline{\text{SIZE16}}$ pin during local-memory coprocessor cycles. However, $\overline{\text{SIZE16}}$ should still be asserted at a valid level (high or low) at the time it is sampled by the TMS34020.

10.4.3 Data Transfer Sequences to or from a Coprocessor

When more than one 32-bit word of data is to be transferred, the TMS34020 uses page mode unless either the coprocessor or the local memory indicates that they do not support page mode (this is accomplished by asserting the $\overline{\text{PGMD}}$ pin high at the appropriate time).

If page mode is not supported, the TMS34020 outputs an address/status subcycle before each data transfer. During direct data transfers, the command is output again (with I=1 to indicate that the second 32-bit word is about to be transferred).

Even if page mode is supported, high-priority local-memory requests (such as a VRAM serial-register transfer) can interrupt a page-mode sequence; the cycle restarts from the next word after the high-priority request is serviced. The sequence restarts with an address/status subcycle, outputting either the address of the appropriate location in memory (during an indirect transfer), or the command with I=1 (during a direct transfer).

Because of this, a coprocessor must be able to tolerate an interruption of a data-transfer sequence, and must be informed in advance of the number of words of data to be transferred. This information could be included as part of the coprocessor command that precedes the data transfer.

10.4.4 Ending a Local-Memory Coprocessor Cycle

During local-memory coprocessor cycles, the TMS34020 samples the LRDY and BUSFLT pins (just as it samples them for other local-memory cycles). This is how the TMS34020 determines when and how the memory cycle ends. For a detailed discussion of how the TMS34020 responds to these inputs, see Section 8.6, Ending a Local-Memory Cycle (page 8-12).

- **Inserting wait states.** You can extend a coprocessor data transfer, as required, by inserting wait states. If the coprocessor is not ready to perform a data transfer when the local-memory cycle is initiated, it can control LRDY and BUSFLT to insert wait states.
- **Retrying local-memory coprocessor cycles.** A coprocessor data transfer may be retried. If this occurs, the memory cycle is performed again in the same manner as a regular access. Any data on the LAD bus during a retried memory cycle should be considered invalid and should not be latched or used by the coprocessor.
- **Bus faults on local-memory coprocessor cycles.** A coprocessor data transfer may be bus faulted. As the CPU initiates local-memory coprocessor cycles, the bus-fault interrupt is taken when a bus fault occurs. Any data on the LAD bus during a bus-faulted memory cycle should be considered invalid and should not be latched or used by the coprocessor. After the bus-fault interrupt is serviced, the memory cycle is performed again in the same manner as for a retry.

10.4.5 Coprocessor Command Cycle

Performed when Passing a command to a coprocessor (see Figure 10–2). No other data is transferred.

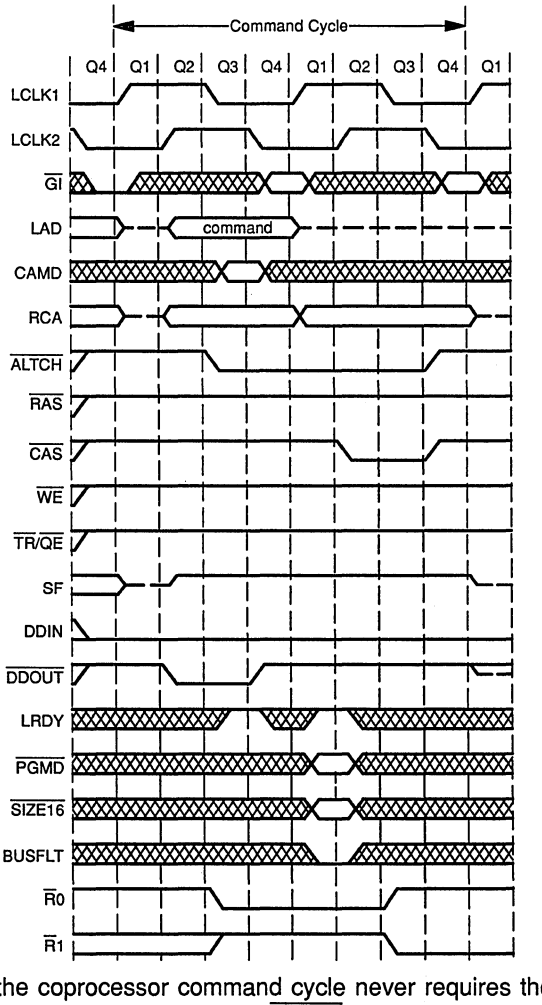
Indicated by
- DDIN is low and
- $\overline{RAS}$, $\overline{TR/QE}$, and SF are all high while $\overline{ALTCH}$ is low

Caused by CEXEC instruction

Status code 0000_2

The command typically causes the coprocessor to execute some internal function. However, the coprocessor command could also be used to pass values to the coprocessor.

Figure 10–2. Coprocessor Command Cycle

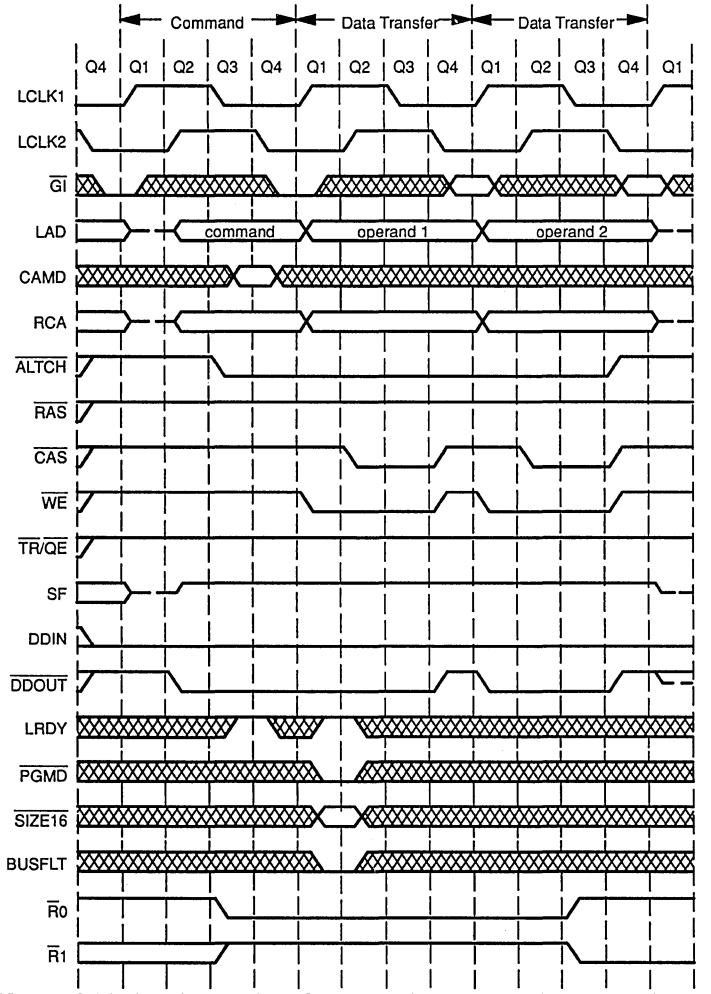


Note: Although the coprocessor command cycle never requires the use of page-mode cycles, you should still assert $\overline{PGMD}$ at a valid level (high or low) at the time it is sampled by the TMS34020.

10.4.6 Transferring Values from TMS34020 Registers to a Coprocessor

Performed when	Passing one or two parameters from TMS34020 registers to a coprocessor. Figure 10–3 shows two words being transferred using page mode.
Indicated by	$\overline{RAS}$ and SF are high while $\overline{ALTCH}$ is low
Caused by	CMOVGC instruction
Status code	0000_2

Figure 10–3. Transferring a TMS34020 Register to a Coprocessor



Key: **LAD (TMS34020)** Output to the LAD bus by the TMS34020
operands 1 & 2 Data to the coprocessor

One or two 32-bit words may be transferred. If two words are transferred, the size bit determines whether the coprocessor should treat the words as two

32-bit parameters or one 64-bit parameter. If two words are transferred but page mode cannot be used, the I bit equals 1 when the command is reissued before transferring the second word.

A coprocessor should latch the data from the LAD bus on the low-to-high transition of $\overline{CAS}$.

10.4.7 Transferring Values from a Coprocessor to TMS34020 Registers

Performed when Passing one or two parameters from a coprocessor to TMS34020 registers. Figure 10–4 shows two words being transferred using page mode.

Indicated by
- $\overline{RAS}$, $\overline{TR}/\overline{QE}$, and SF are high while $\overline{ALTCH}$ is low
- DDIN is low

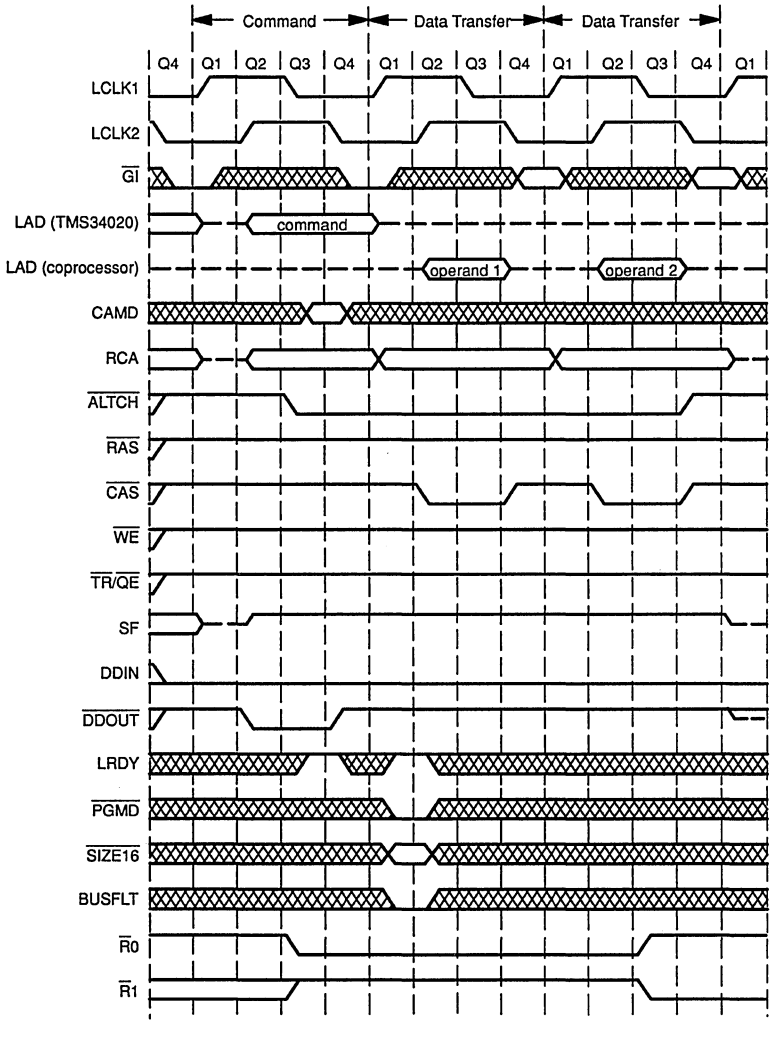
Caused by
- CMOVCG, which transfers one or two 32-bit words.
- CMOVCS, which transfers one 32-bit word. The TMS34020 ignores the 28 LSBs of this word; however, the 4 bits of data output on LAD28—LAD31 are transferred to the N, C, Z and V status bits.

Status code 0000_2

One or two 32-bit words may be transferred. If two words are transferred, the value of the size bit (output as part of the command during the address/status subcycle) depends on whether the TMS34020 treats the words as two 32-bit parameters or one 64-bit parameter. If two words are transferred but page mode cannot be used, the I bit equals 1 when the command is reissued before transferring the second word.

During this type of cycle, a coprocessor should not assert data onto the LAD bus until the high-to-low transition of $\overline{CAS}$.

Figure 10–4. Transferring from a Coprocessor to a TMS34020 Register

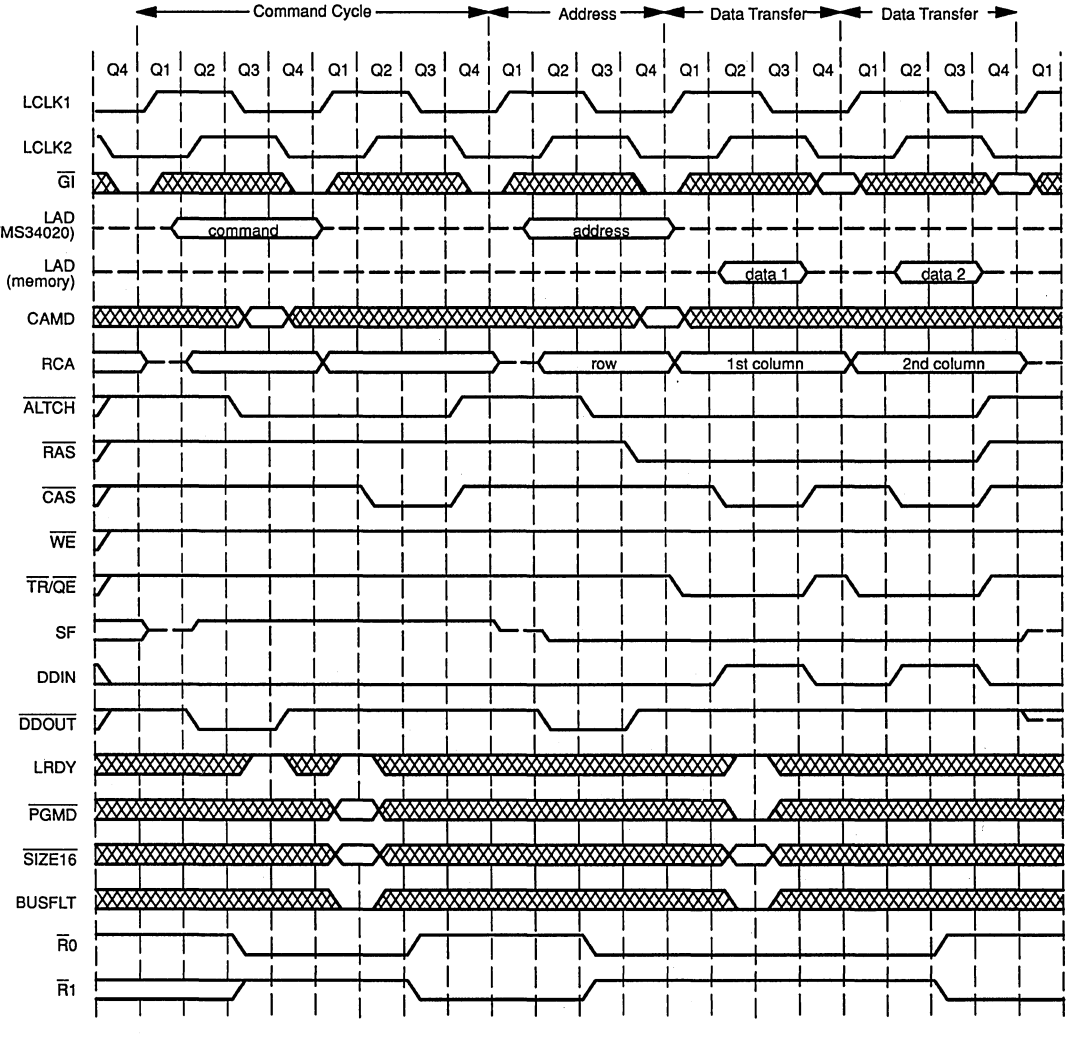


Key: **LAD (TMS34020)** Output to the LAD bus by the TMS34020
LAD (coprocessor) Output to the LAD bus by the coprocessor
operands 1 & 2 Data from the coprocessor

10.4.8 Transferring Values from Local Memory to a Coprocessor

Performed when	Passing up to thirty-two 32-bit words from the local memory to a coprocessor. Figure 10–5 shows two words being transferred using page mode.
Caused by	CMOVMC instruction
Status code	0000_2

Figure 10–5. Transfer Memory to Coprocessor



Key: **LAD (TMS34020)** Output to the LAD bus by the TMS34020
LAD (memory) Output to the LAD bus by the memory
operands 1 & 2 Data to the coprocessor

Transferring data from local memory to a coprocessor requires explicit addressing of the memory. Because of this, no data is transferred immediately after the command. After the command cycle completes, the data-transfer sequence begins with another address/status subcycle. During this subcycle, the address of the memory location to be transferred to the coprocessor is output.

This data-transfer sequence is identical to an ordinary data read from memory, except that the status code is 0000_2. A coprocessor should latch the data from the LAD bus on the low-to-high transition of $\overline{CAS}$.

If page-mode operation is not supported (by either the coprocessor or the local memory), a complete memory cycle is generated for each word transferred.

A page-mode sequence may also be interrupted by a high-priority memory request. In this case, the data transfer resumes with the next word in memory following the high-priority request. The address of that location is output, along with the coprocessor operation status code; the command is not reissued.

10.4.9 Transferring Values from a Coprocessor to Local Memory

This cycle is used to pass up to thirty-two 32-bit words from a coprocessor to local memory. Figure 10–6 shows 2 words being transferred using page mode.

This cycle requires the memory to be explicitly addressed. No data is transferred immediately following the command. After the command cycle completes, the data transfer sequence begins with another address/status subcycle. During this subcycle, the address of the location in memory to be written to by the coprocessor is output. In this case, however, data is not transferred immediately after the address/status subcycle of the data-transfer sequence. An extra machine state is inserted to separate the address/status and data subcycles of the transfer cycle. This **spacer** subcycle is inserted to ensure that the TMS34020 can set its LAD bus drivers to the high- impedance state before the coprocessor starts to drive data onto the LAD bus. This is necessary because the coprocessor must assert data onto the LAD bus following the high-to-low transition of $\overline{WE}$ during this type of cycle, so that the LAD bus contains valid data for the memories on the high-to-low transition of $\overline{CAS}$.

Note that whereas data is asserted onto the LAD bus after the falling edge of $\overline{WE}$ during this type of cycle, it **must not** be asserted onto the bus before the falling edge of $\overline{CAS}$ during a direct coprocessor-to-TMS34020 transfer (see Section 10.4.6). To enable the coprocessor to distinguish between the two and assert data onto the LAD bus at the appropriate time, $\overline{ALTCH}$ is driven high at the end of the spacer subcycle.

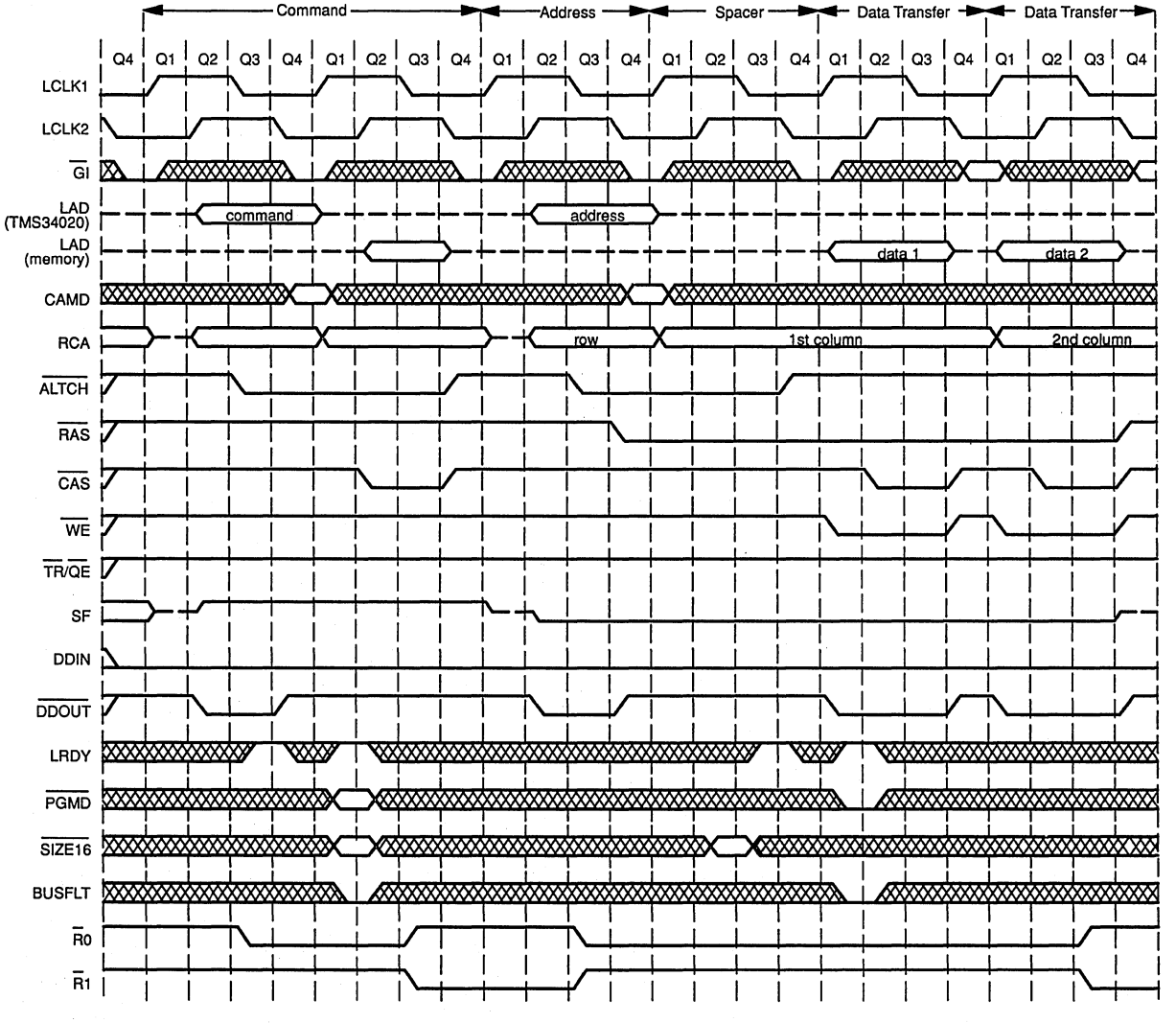
This data-transfer sequence is identical to an ordinary data write to memory, except that the status code is 0000_2, the spacer subcycle is inserted, and $\overline{ALTCH}$ is high while data is transferred.

If page-mode operation is not supported (by either the coprocessor or the local memory), a complete memory cycle (including spacer) is generated for each word transferred.

A page-mode sequence may also be interrupted by a high-priority memory request. In this case, the data transfer resumes from the next word in memory following the high-priority request. The address of that location is output, along with the coprocessor operation status code; the command is not reissued.

The CMOVCM instruction causes this type of cycle.

Figure 10–6. Transferring from Coprocessor to Memory



Key:

LAD (TMS34020)	Output to the LAD bus by the TMS34020
LAD (memory)	Output to the LAD bus to the memory
operands 1 & 2	Data to the coprocessor

10.5 Coprocessor Aborts and Status Checks

All coprocessors should recognize at least two coprocessor command cycles and provide the appropriate response, even if busy. These cycles would be recognized as:

abort The abort should terminate all coprocessor activity, restoring the coprocessor to a known state so that it is available for further commands from the TMS34020. The abort allows the TMS34020 to regain control of the coprocessor in the event of a system fault that may involve the coprocessor.

status check The response to the status check command should be to provide information regarding the coprocessor's operating condition. The MSB output in response to the status check should be set high if the coprocessor is busy. Other bits in the response word may be used for other information concerning the coprocessor status. The TMS34020 can use the status check command to determine if a coprocessor is currently busy when multiple tasks are competing for the coprocessor. Thus, the TMS34020 does not have to enter an extended wait state to obtain access to the coprocessor, but may continue with another task not requiring the coprocessor.

The actual encoding of the coprocessor command field for the abort and status check cycles depends on the requirements of the specific coprocessor.

10.6 System Configuration

Some applications execute the same software on two systems—one with a coprocessor and one without. The TMS34020 does not support this directly. However, you can accomplish this in a number of ways by building configuration information into the system hardware, such as a wire jumper or switch, and setting it one way or the other. depending on whether a coprocessor is present. A number of possible approaches can then be taken.

- Depending on the state of the jumper of switch, different values can be returned based on the value at a system-defined memory location. This would allow the software to configure itself to execute code that either did or did not use the coprocessor instructions.

- If the jumper or switch indicated that a coprocessor was not present in the system, external logic could be used to detect memory cycles that output the coprocessor operation status code and to generate a bus fault accordingly. The bus-fault service routine could then determine what type of operation the coprocessor instruction was trying to perform and emulate it in software.

By using more switches or jumpers, you could easily extend these mechanisms for use in systems in which the number of coprocessors varies.

Chapter 11

Multiprocessing and System Architecture

The TMS34020 can share its local-memory resources with other processors by means of a 3-wire interface. This **multiprocessor interface** allows multiple TMS34020s (as well as other processors) to share the same local-memory space. In order to accomplish this, these processors must support a conventional hold/hold-acknowledge protocol. This chapter includes these topics:

Advantages of a 3-wire interface

The TMS34020's 3-wire multiprocessor interface provides more flexibility than the conventional 2-wire hold/hold-acknowledge interface supported by many other processors. The $\overline{R}0$ and $\overline{R}1$ request pins provide information about how urgently a TMS34020 requires access to the local-memory bus. They continue to provide this information, regardless of whether the TMS34020 is currently controlling the bus or not. This allows external arbitration logic to allocate control of the local-memory bus to the processor most in need.

This chapter uses the following terms:

- **Bus master** refers to the device that currently controls the local-memory interface. If the bus-master TMS34020 generates a high-priority request code, the arbitration logic can ensure that the TMS34020 maintains control of the bus.
- **Bus requestor** refers to a device that does not currently control the local-memory interface, but is requesting control. If a bus-requestor TMS34020 generates a high-priority request code, the arbitration logic can take the appropriate steps to ensure that the requestor obtains control of the local-memory bus as soon as possible.

11.1 Related Signals

Chapter 2 describes the multiprocessor-interface signals in detail; these signals are summarized below for your convenience.

Signals	Descriptions	I/O
$\overline{GI}$	is the grant-input signal. When active low, $\overline{GI}$ informs the TMS34020 that it is the bus master, which means that it must drive the local-memory bus and that it controls the next memory cycle. When driven inactive high, $\overline{GI}$ informs the TMS34020 that it should terminate the current memory access as soon as possible and relinquish bus mastership.	I
$\overline{R0}$, $\overline{R1}$	form the bus-request code output by the TMS34020. External arbitration logic uses this encoded information when determining which processor should receive an active-low signal on its $\overline{GI}$ pin.	O

11.2 Overview

The multiprocessor interface supports a general protocol that can be used with external arbitration logic to form a system of multiple processors sharing a common local-memory space. The TMS34020 provides a mechanism for synchronizing multiple TMS34020s to the same local clock (LCLK) phase. The interface signals are timed to allow multiple, synchronized TMS3020s to share the local-memory bus without wasting memory cycles when passing control from one TMS34020 to another.

When multiple TMS34020s share local memory, performance should increase because of the large internal program cache that can allow long time intervals during program execution where no external memory accesses are required. This memory bandwidth can then be used by another TMS34020.

In theory, there is no limit to the number of devices that can be configured together. However, it is doubtful that performance would increase significantly if a system included more than three TMS34020s. It is likely that the performance difference between one processor and two processors will be greater than the performance difference between two processors and three processors, and so on. Once the local-memory bandwidth is completely used, no performance benefit arises from adding extra processors.

11.3 Basic Multiprocessor System Configuration

This section contains general information about connecting and synchronizing multiple processors within a single system.

11.3.1 Connecting Multiple Processors Together

All the local-memory control pins, address pins, and data pins should be wired in parallel between the processors. (This includes LAD0—LAD31, RCA0—RCA12, SF, ALTCH, RAS, CAS0—CAS3, WE, TR/QE, DDIN, and DDOUT.) If your system contains a non-TMS34020 device, wire the equivalent signals common. If you wish, you can wire the local-memory input pins (CAMD, LRDY, BUSERR, PGMD, and SIZE16) common. How you choose to wire HDST and HOE depends on your system architecture and on how you wish to connect a host processor. Section 11.7 describes this in detail.

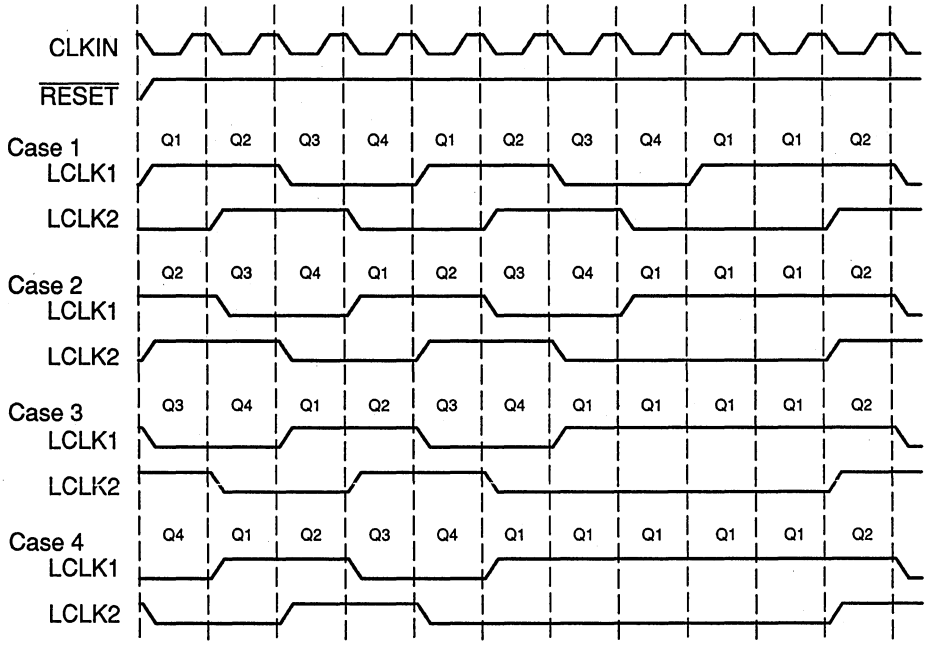
All TMS34020s in the system should share the same RESET and CLKIN inputs. *Do not* wire the local clock outputs (LCLK1 and LCLK2) together. Use one TMS34020's outputs as the local clocks for the entire system.

11.3.2 Synchronizing Multiple TMS34020s at Reset

To allow multiple TMS34020s to share control of the local-memory interface without losing any memory bandwidth, the TMS34020s must be synchronized to the same local clock phase. The TMS34020 achieves this with special internal logic that detects the rising edge of the RESET pin at the end of reset. In a typical single-TMS34020 system, RESET is not required to be synchronous to CLKIN. However, to allow synchronization of multiple TMS34020s in a system, the rising edge of RESET must meet the setup and hold requirements with respect to CLKIN. This ensures that all TMS34020s respond to RESET on the same quarter phase (refer to the *TMS34020 Data Sheet* for details).

Figure 11–1 shows the four possible conditions for the state of the TMS34020 at the time RESET goes high. Within 10 CLKIN cycles after RESET goes high, all TMS34020s will be synchronized to the same quarter phase through the extension of the Q1 phase.

Figure 11–1. Synchronization of Multiple TMS34020s



Note: This figure does not imply timing dependences of LCLK1 and LCLK2 relative to CLKIN or $\overline{\text{RESET}}$.

Key: **Case 1** Extension of Q1 for 1 quarter phase.
Case 2 Extension of Q1 for 2 quarter phases.
Case 3 Extension of Q1 for 3 quarter phases.
Case 4 Extension of Q1 for 4 quarter phases.

11.4 Protocols for Communicating in a Multiprocessor System

This section provides details of how the TMS34020 requests, is granted, and relinquishes control of the local-memory bus and control signals.

11.4.1 How a Processor Requests Control of the Local-Memory Bus

The multiprocessor interface provides encoded information on $\overline{R0}$ and $\overline{R1}$. These codes indicate whether or not the TMS34020 wishes to use the local memory and, if so, how urgently. Table 11–1 describes this coding.

Table 11–1. Bus Request Codes for the Multiprocessor Interface

$\overline{R0}$	$\overline{R1}$	Code and Description
0	0	**High-priority request.** The TMS34020 is requesting or using the bus to perform a high-priority memory access. High-priority memory requests are issued for accesses that must be performed urgently. They include ❑ VRAM serial-register transfer initiated by the video control logic. ❑ 12 or more DRAM refreshes pending. ❑ Accesses initiated by a host processor.
0	1	**Low-priority request.** The TMS34020 is requesting or using the bus to perform a low-priority memory access. A low-priority request is an access requested by the TMS34020's CPU or a DRAM refresh (when less than 12 refreshes are pending).
1	0	**Access termination.** The TMS34020 will terminate the current memory access in this machine state. When performing multiple accesses using page mode, the access-termination code is output during the last data access of the sequence only.
1	1	**No request.** The TMS34020 does not require access to the local memory in the next machine state. This code is output during host-default states when the next machine state will be another host-default state.

Section 8.5, Local-Memory Cycle Status Codes (page 8-10), details the different types of local-memory cycles and their relative priorities; the request code depends on the cycle's priority.

- ❑ Cycles with priorities **2** through **5** generate a **high-priority** request code.
- ❑ Cycles with priorities **6** through **8** generate a **low-priority** request code.
- ❑ The host-default cycle can generate either a **no-request** or **low-priority** request code.

11.4.2 How a Processor Releases Control of the Local-Memory Bus

If external bus-arbitration logic drives the bus-master TMS34020's $\overline{GI}$ pin inactive high, the TMS34020 releases the local-memory bus as soon as possible:

- ❑ **No-request or access-terminate code.** If the TMS34020 is currently outputting either of these codes, it relinquishes control of the bus and sets all its local bus outputs to high impedance during the second quarter phase (Q2) of the next LCLK cycle. ($\overline{DDOUT}$ is set to high impedance during Q1.)

- **Low-priority or high-priority request code.** If the TMS34020 is currently outputting either of these codes, it terminates the current memory access as soon as possible. This occurs after the current access completes.
  - If the TMS34020 is partially through a sequence of page-mode accesses, the sequence is broken. When bus mastership is regained, the TMS34020 continues from the next address in the sequence.
  - If the TMS34020 is partially through a memory operation that requires multiple memory accesses (such as a read-modify-write or an access to 16-bit memory), the current access completes. When bus mastership is regained, the operation resumes with the next memory access in the sequence.

The current access is allowed to complete in the manner determined by the LRDY and BUSFLT pins. A detailed discussion of the function of these pins is given in Section 8.6, Ending a Local-Memory Cycle (page 8-12).

During the last machine state of any memory access, the TMS34020 issues the access-termination code. If $\overline{GI}$ is driven inactive at this time, the TMS34020 *will* relinquish control of the local-memory bus at the beginning of the next machine state.

11.4.3 Passing Control of the Local-Memory Bus

When bus control is passed from one TMS34020 to another (or to/from another type of device), it is necessary to avoid conflicting driven outputs. However, it is also desirable that the shared local-memory bus and control signals are not left undriven; floating signals are prone to noise and can make the system unreliable. You could use external pull-up resistors to hold the undriven signals at a good voltage level, but this is undesirable because the resulting circuit board is larger and more expensive to produce. The TMS34020's multiprocessor interface allows signals to be driven to the same level by both processors involved in the exchange, avoiding all of these problems. When different TMS34020s pass control of the local-memory bus, the following steps happen automatically:

Step 1: The TMS34020 that becomes bus master drives the local bus and control pins to their inactive levels from the beginning of the LCLK cycle after its $\overline{GI}$ pin is asserted.

Step 2: The TMS34020 that releases control of the bus sets its local bus and control pins to high impedance during Q2 of the LCLK cycle after its $\overline{GI}$ pin is driven inactive. This provides one quarter phase of overlap where both TMS34020s are driving the signals inactive, and therefore no pull-up resistors are necessary. The only exception to this is $\overline{DDOUT}$, which is not driven inactive by the releasing device during Q1, as the acquiring device may drive it active low during Q2.

Depending on inter-TMS34020 clock skew, this may result in a *short* period when $\overline{\text{DDOUT}}$ is undriven. This is preferable to having the pin driven to conflicting levels, however. $\overline{\text{DDOUT}}$ will never be undriven for more than a few nanoseconds, and so external resistors should not be necessary.

If your system contains a non-TMS34020 processor, you must ensure that this processor drives *all* the local-memory control and bus pins from the beginning of the memory cycle when it assumes control of the bus and that it stops driving them at the end of the last memory cycle when it releases control of the bus.

CAUTION

Undriven signals may produce unpredictable or unreliable system operation. Signals driven to different levels simultaneously by more than one device may damage the driving devices as well as cause unreliable system operation.

11.4.4 Functional Timing Examples

The following diagrams show various examples of the TMS34020 obtaining and releasing control of the local-memory bus and control signals. The TMS34020 samples the $\overline{\text{GI}}$ pin on the rising edge of LCLK1 (at the end of Q4). $\overline{\text{R0}}$ and $\overline{\text{R1}}$ change on the falling edge of LCLK1 (at the beginning of the Q3 phase of the machine state). Refer to the *TMS34020 Data Sheet* for the precise timing relationships of these pins.

Figure 11–2 shows the TMS34020 outputting the no-request code and releasing the bus during Q2 of the LCLK cycle, immediately after $\overline{\text{GI}}$ is driven high.

Figure 11–2. Releasing Control of the Local-Memory Bus and Control Signals ($\overline{GI}$ Driven High During a Host-Default Idle Cycle)

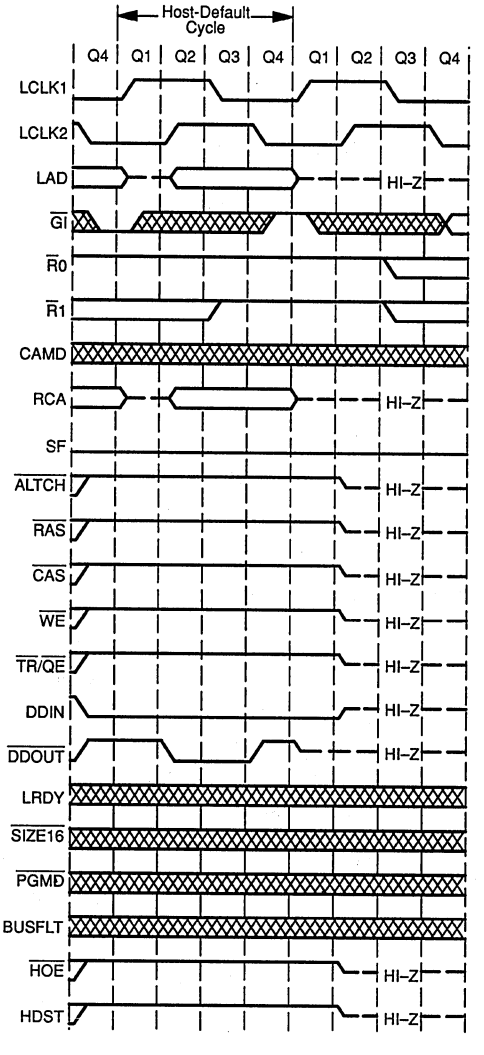


Figure 11–3 is identical to Figure 11–2 except that the TMS34020 is outputting the access-termination code at the end of a low-priority write cycle. Note that $\overline{GI}$ may be sampled at either level while the low-priority request code is being output. The TMS34020 cannot relinquish control of the local-memory interface until it completes the memory access (at which time it issues the access-termination code).

Figure 11–3. Releasing Control of the Local-Memory Bus and Control Signals ($\overline{GI}$ Driven High at the End of a Write Cycle)

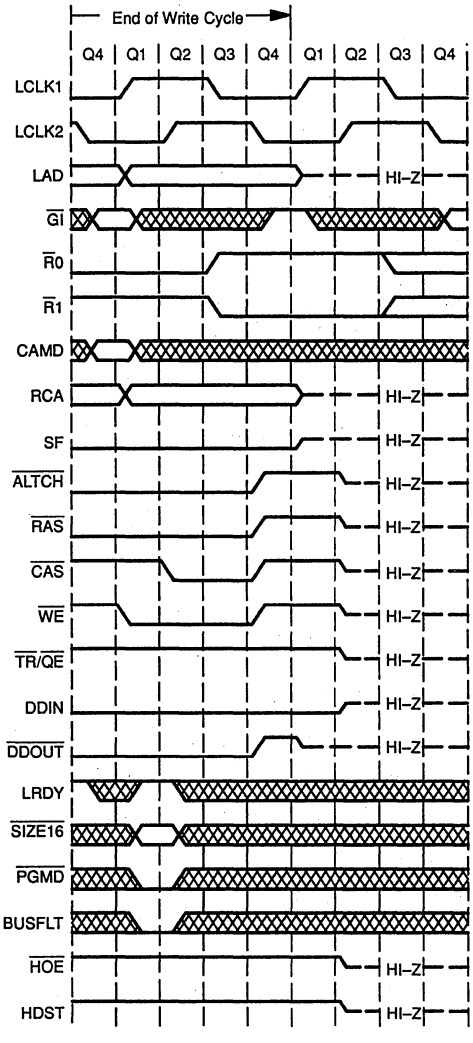


Figure 11–4 shows the TMS34020 outputting the access-termination code at the end of a (high priority) host read cycle. Note that $\overline{GI}$ may be sampled at either level while the high-priority request code is being output. The TMS34020 cannot relinquish control of the local-memory interface until it completes the memory access (at which time it issues the access-termination code).

Figure 11–4. Releasing Control of the Local-Memory Bus and Control Signals ($\overline{GI}$ Driven High at the End of a Host Read Cycle)

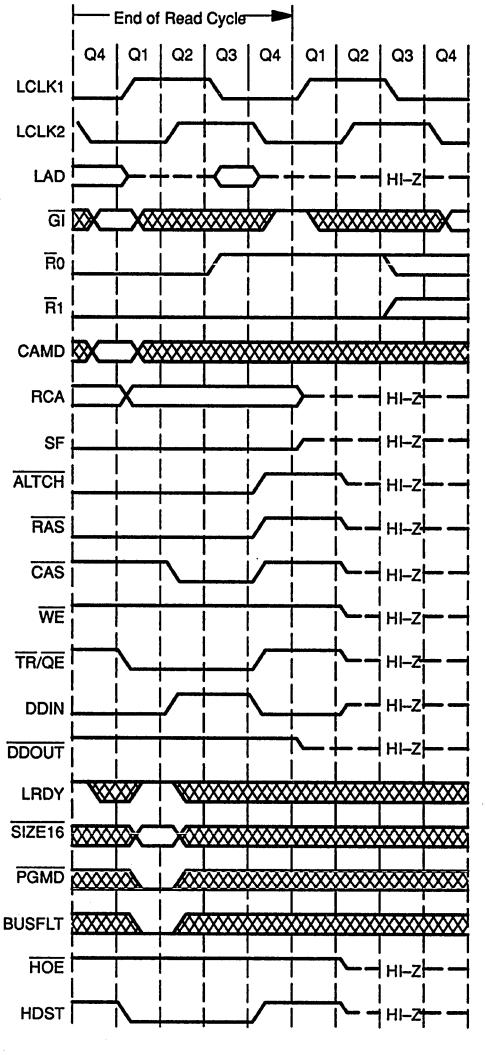
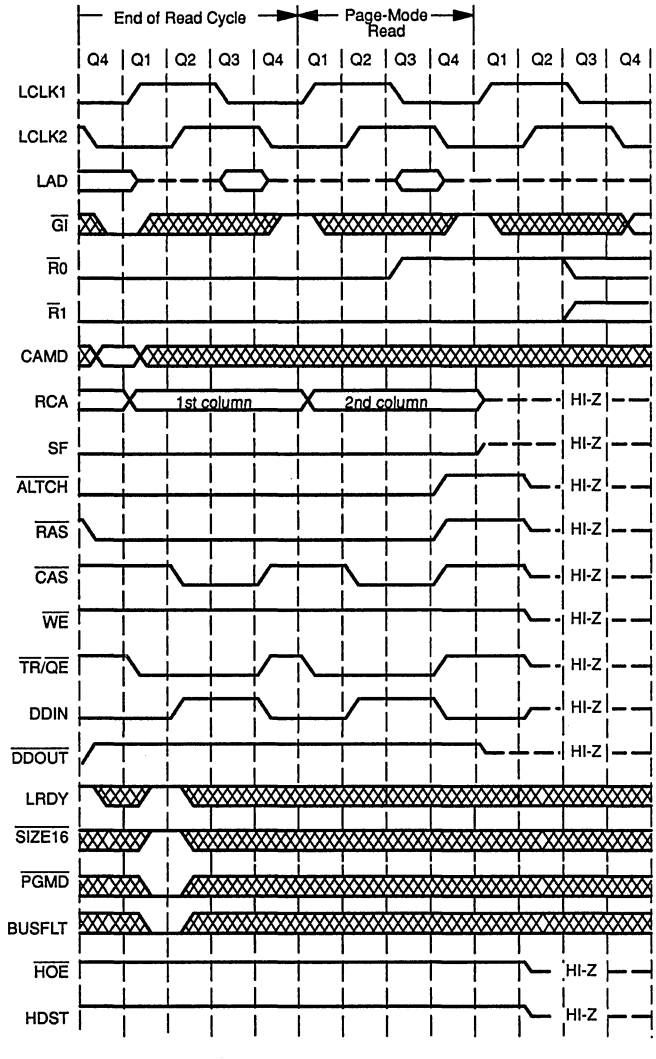


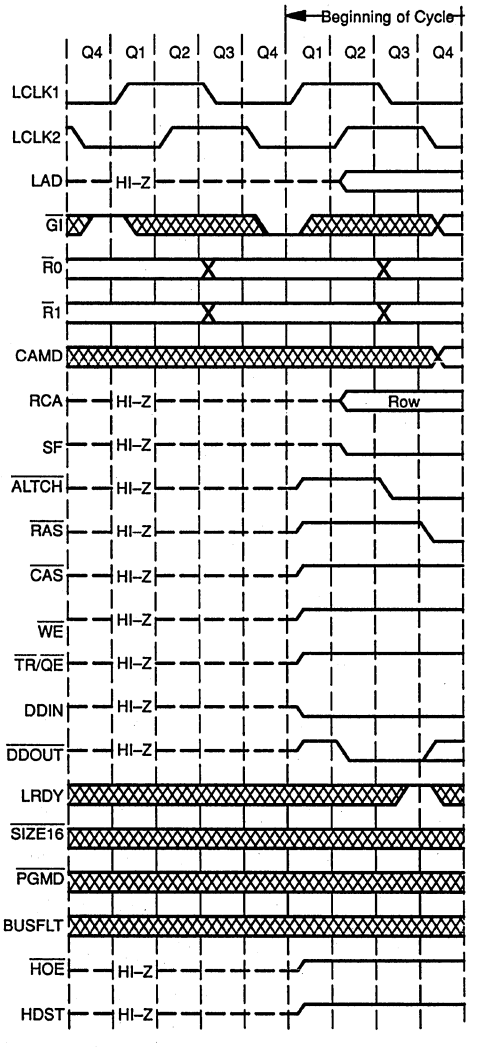
Figure 11–5 shows $\overline{GI}$ being driven inactive just after the TMS34020 has started a sequence of page mode accesses. The TMS34020 breaks the sequence and issues the access-termination code. If $\overline{GI}$ had remained active, this code would not have been generated until the sequence completed.

Figure 11–5. Releasing Control of the Local-Memory Interface ($\overline{GI}$ Driven High During a Page-Mode Sequence)



In Figure 11–6, the TMS34020 regains bus mastership as soon as its $\overline{GI}$ pin is driven active low. $\overline{R0}$ and $\overline{R1}$ could be outputting any of the codes with the exception of the access-termination code.

Figure 11–6. Regaining Control of the Local-Memory Bus and Control Signals



11.5 Arbitration Logic Requirements

A multiprocessor system's external arbitration logic must generate the $\overline{GI}$ outputs to each processor for the next cycle. To do this, it uses each processor's $\overline{R0}$ and $\overline{R1}$ outputs in conjunction with the current value of each processors' $\overline{GI}$ input. The arbitration logic must **never** assert the $\overline{GI}$ pins of more than one TMS34020 simultaneously, though there may be times when none of the processors has an active $\overline{GI}$.

11.5.1 Passing Control of the Local-Memory Bus

When passing control of the local-memory bus from one processor to another, the arbitration logic must not assert the acquiring processor's $\overline{GI}$ pin until the current master processor indicates that it will relinquish control of the local-memory bus at the end of the current machine state. This is the case when the current bus master outputs either the no-request or access-termination code. These are easily identified by looking for an inactive high level on $\overline{R0}$.

Figure 11–7 shows a typical example for two TMS34020s.

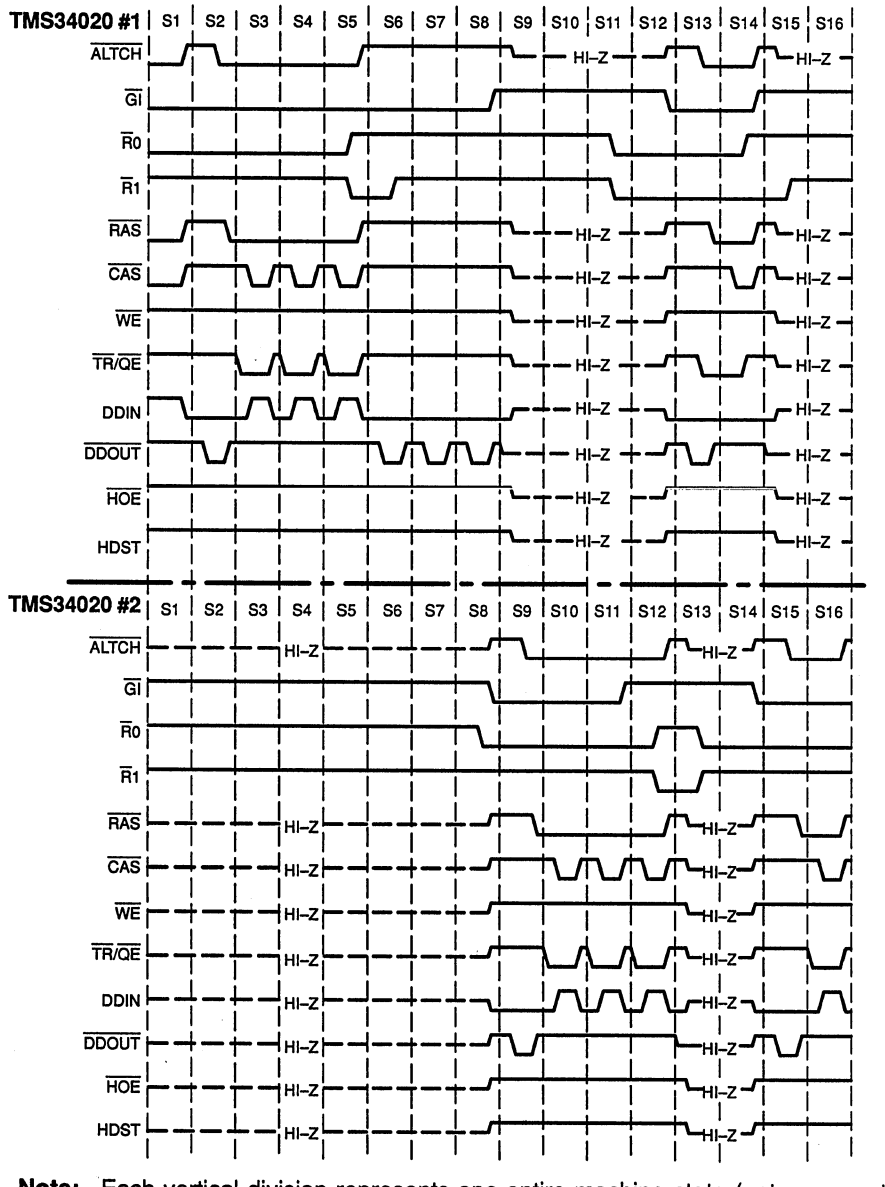
- TMS34020#1 starts off as the bus master. It finishes using the bus during the fifth machine state (S5), but continues to drive the bus inactive until after TMS34020#2 requests the bus in the eighth machine state (S8).

- As TMS34020#1 outputs the no-request code during S8, the $\overline{GI}$ pins of both TMS34020s can be changed simultaneously. TMS34020#2 then becomes bus master in S9. However, a high-priority request (in this case, for a screen-refresh cycle) made by TMS34020#1 during S11 causes TMS34020#2 to relinquish control of the bus at the end of S12. Note that because TMS34020#2 was busy when TMS34020#1 first generated the high-priority request, its $\overline{GI}$ is not asserted until TMS34020#2 outputs the access-termination code.

- TMS34020#2 continues to output a low-priority request and resumes control of the bus (at the beginning of S15) as soon as TMS34020#1 finishes and asserts the access-termination code.

  Note that although TMS34020 #2 is making a request, TMS34020 #1's grant-in is not removed, because it is performing a high-priority request. Also note that no bus cycles are lost. $\overline{R0}$ and $\overline{R1}$ change sufficiently early for the $\overline{GI}$ signals to change and allow back-to-back local-memory cycles from different TMS34020s.

Figure 11–7. Passing Control of the Local-Memory Between Two TMS34020s



Note: Each vertical division represents one entire machine state (not one quarter phase, as other diagrams show).

11.5.2 Wait States, Retries, and High-Priority Bus Requests

As previously indicated, if the TMS34020 is performing a memory cycle extended with wait states when $\overline{GI}$ is driven inactive, the TMS34020 will not relinquish control of the local-memory interface until the access completes. This could consume many machine states. (For instance, the TMS34020 could be accessing a slow peripheral or possibly a location in the host's memory via the system bus.)

This could cause a problem if one of the TMS34020s has a limited time in which to perform high-priority accesses. For example, if the horizontal-blanking interval is very short, the screen-refresh cycle scheduled at this time must be serviced promptly (before the end of the blanking interval).

To circumvent this sort of problem, the arbitration logic can control LRDY and BUSFLT to cause a retry of the slow, wait-stated access. If a retry code is detected (LRDY low, BUSFLT high), the memory cycle will terminate (issuing the access-termination code) and subsequently restart. Once the access-termination code is issued, however, the arbitration logic can assign control of the local-memory interface to another TMS34020.

If you wish to do this, you should ensure that the accessed memory location can respond correctly to a retry. For example, a peripheral that increments an internal address pointer when a cycle completes would need to be able to distinguish a successful transfer from a retry, preventing the internal pointer from being incremented after the retried access.

11.6 Multiprocessor Arbitration Examples

This section provides two examples of arbitration schemes; these examples should help you to understand the multiprocessor interface.

- The first example shows a typical scheme for two TMS34020s.
- The second example shows a scheme for a TMS34020 and another (non-TMS34020) device that uses a hold/hold-acknowledge protocol.

11.6.1 Arbitration Scheme for Two TMS34020s

This example describes a protocol between two TMS34020s, *TMS34020*$_A$ and *TMS34020*$_B$. Assume that only TMS34020$_A$ can make high-priority requests; only these requests can cause TMS34020$_B$'s $\overline{GI}$ signal ($\overline{GI}_B$) to be driven inactive when TMS34020$_B$ is not outputting the no-request or access-termination code. In this way, only high-priority requests interrupt page-mode bursts.

Table 11–2 shows how the arbitration logic allocates active or inactive levels for the next machine state on the $\overline{GI}$ pins of the two TMS34020s ($\overline{GI}_A$ and $\overline{GI}_B$, respectively). The values of the $\overline{GI}$ pins for the next machine state are sampled on the rising edge of LCLK1, at the beginning of the next machine state.

The decision is based on the present state of the bus request outputs from the two TMS34020s ($\overline{R0}_A$, $\overline{R1}_A$, $\overline{R0}_B$, and $\overline{R1}_B$) and the present state of the $\overline{GI}$ outputs ($p\overline{GI}_A$ and $p\overline{GI}_B$). The bus-request pins become valid shortly after the falling edge of LCLK1, at the beginning of the Q3 phase of the TMS34020s' machine states. Assume that the $p\overline{GI}_A$ and $p\overline{GI}_B$ values are latched externally on the rising edge of LCLK1.

The top and left side of Table 11–2 show the possible combinations of $\overline{R0}_A$, $\overline{R1}_A$, $p\overline{GI}_A$, and $\overline{R0}_B$, $\overline{R1}_B$, $p\overline{GI}_B$. The resulting grid shows which $\overline{GI}$ (if any) is asserted for the next machine state, given the current state.

Table 11–2. Arbitration Scheme for Two TMS3402020s

				TMS34020$_A$							
			$p\overline{GI}_A$	1	1	1	1	0	0	0	0
			$\overline{R0}_A$	1	0	0	1	1	0	0	1
	$p\overline{GI}_B$	$\overline{R0}_B$	$\overline{R1}_A$ / $\overline{R1}_B$	1	1	0	0	1	1	0	0
TMS34020$_B$	1	1	1		—	—	—	—	A	A	A	A
	1	0	1		—	—	X	—	B	A	A	B
	1	1	0		—	—	—	—	—	—	—	—
	0	1	1		B	A	A	—	—	—	—	—
	0	0	1		B	B	X	—	—	—	—	—
	0	1	0		B	A	A	—	—	—	—	—

Key: **A** Bus allocated to TMS34020$_A$ (assert $\overline{GI}_A$)
B Bus allocated to TMS34020$_B$ (assert $\overline{GI}_B$)
— Illegal combination
X No allocation (wait for cycle termination)

Table 11–2 can be summarized as follows:

- Active $\overline{GI}$ is made to TMS34020$_A$ ($\overline{GI}_A$) if
  - Active $\overline{GI}$ was already allocated to TMS34020$_A$ and there is no request from TMS34020$_B$.
  - Active $\overline{GI}$ was already allocated to TMS34020$_A$ and both devices are requesting access (high or low priority).
  - TMS34020$_A$ is requesting an access (high or low priority) and TMS34020$_B$ is not requesting access or is terminating an access.
- Active $\overline{GI}$ is made to TMS34020$_B$ ($\overline{GI}_B$) if
  - Active $\overline{GI}$ was already allocated to TMS34020$_B$ and there is no request from TMS34020$_A$.
  - Active $\overline{GI}$ was already allocated to TMS34020$_B$ and both devices are requesting a low-priority access.
  - TMS34020$_B$ is requesting a low-priority access and TMS34020$_A$ is not requesting access or is terminating an access.
- Active $\overline{GI}$ is made to *neither* TMS34020$_A$ or TMS34020$_B$ ($\overline{GI}_X$) if TMS34020$_A$ requests a high-priority access but TMS34020$_B$ has not yet terminated its access.

The following equations express these conditions.

$$\overline{GI}_A = (p\overline{GI}_A \cdot R0_B \cdot R1_B) + (p\overline{GI}_A \cdot \overline{R0}_A) + (p\overline{GI}_B \cdot R0_B \cdot \overline{R0}_A)$$
$$\overline{GI}_B = (p\overline{GI}_B \cdot R0_A \cdot R1_A) + (p\overline{GI}_B \cdot \overline{R0}_B \cdot R1_A) + (p\overline{GI}_A \cdot \overline{R0}_B \cdot R0_A)$$
$$\overline{GI}_X = (pGI_A \cdot \overline{R0}_A \cdot \overline{R1}_A \cdot \overline{R0}_B)$$

You could easily expand this protocol to include additional TMS34020s by adding more slave devices, each with a successively lower priority.

11.6.2 Arbitration Scheme for One TMS34020 and a Hold Device

If a multiprocessor system contains a non-TMS34020 device that supports a hold/hold-acknowledge protocol (known as a **hold device**), you can use a modified version of the arbitration scheme described in Section 11.6.1. In the modified version, the hold device replaces the slave TMS34020; however, the non-TMS34020 processor's priority is raised so that hold has a higher priority than a low-priority request from the bus-master TMS34020.

Table 11–3 shows which $\overline{GI}$ is activated for a given set of inputs in this situation. The single TMS34020's signals have the suffix $_T$. $\overline{R0}_B$ and $\overline{R1}_B$ are replaced by a single signal, HOLD. Bus mastership is allocated to the hold device through the $\overline{GI}_H$ signal.

Table 11–3. Arbitration Scheme for One TMS3402020 and a Hold Device

			TMS34020							
		$p\overline{GI}_T$	1	1	1	1	0	0	0	0
		$\overline{R0}_T$	1	0	0	1	1	0	0	1
Hold Device										
$p\overline{GI}_H$	Hold	$\overline{R1}_T$	1	1	0	0	1	1	0	0
1	0		—	—	—	—	T	T	T	T
1	1		—	X	X	—	H	X	T	H
0	0		T	T	T	—	—	—	—	—
0	1		H	H	X	—	—	—	—	—

Key: **T** Bus allocated to TMS34020 (assert $\overline{GI}_T$)
H Bus allocated to the hold device (assert $\overline{GI}_H$)
— Illegal combination
X No allocation (wait for cycle termination)

Table 11–3 can be summarized as follows:

- Active $\overline{GI}$ is made to the TMS34020 ($\overline{GI}_T$) if
  - Active $\overline{GI}$ was already allocated to the TMS34020 and there is no request from the hold device.
  - Active $\overline{GI}$ was already allocated to the TMS34020 and the TMS34020 is requesting a high-priority access.
  - The TMS34020 is requesting an access (high or low priority) and the hold device is not requesting an access.

- Active $\overline{GI}$ is made to the hold device ($\overline{GI}_H$) if
  - Active $\overline{GI}$ was already allocated to the hold device and the TMS34020 is not requesting an access.
  - Active $\overline{GI}$ was already allocated to the hold device and the devices are requesting hold and low-priority accesses, respectively.
  - The hold device is requesting an access, and the TMS34020 is not making an access or is terminating an access.
- Active $\overline{GI}$ is made to *neither* device ($\overline{GI}_X$) if
  - The TMS34020 is requesting a high-priority access but the hold device has not yet terminated its access.
  - The hold device is requesting an access but the TMS34020 has not yet terminated its low-priority access.

The following equations express these conditions:

$$\overline{GI}_T = (p\overline{GI}_T \cdot \overline{HOLD}) + (p\overline{GI}_T \cdot \overline{R0}_T \cdot \overline{R1}_T) + (p\overline{GI}_H \cdot \overline{HOLD})$$
$$\overline{GI}_H = (p\overline{GI}_H \cdot HOLD \cdot R1_T) + (p\overline{GI}_T \cdot HOLD \cdot R0_T)$$
$$\overline{GI}_X = (pGI_T \cdot \overline{R0}_T \cdot \overline{R1}_T \cdot HOLD) + (p\overline{GI}_H \cdot \overline{R0}_T \cdot \overline{R1}_T \cdot HOLD)$$

Note that $\overline{GI}_H$ is equivalent to hold-acknowledge in this scheme. HOLD is effectively treated like an $\overline{R0}$. If back-to-back memory cycles between master and hold device are required, then HOLD going inactive should be an early warning that the hold device has finished with the bus. If this is not possible, then there will be a dead cycle when neither processor is using the bus when passing control.

Again, this protocol can be easily expanded to include more processors: more hold devices, more TMS34020s, or more of both.

11.7 Initializing Multiple TMS34020s

In a single-TMS34020 system, power-up initialization of the device can be performed in one of two ways.

Method 1: Store an initialization routine in ROM and bring the TMS34020 out of reset in the self-bootstrap mode. The TMS34020 will immediately start executing the code from ROM.

Method 2: Bring the TMS34020 out of reset in the host-present mode. In this situation, the TMS34020 is halted so that a host can download code to local memory before clearing the HLT[[HSTCTLH]] bit and allowing the TMS34020 to execute the code.

In a multiple-TMS34020 system, initialization is more complex. You can use method 1, method 2, or a combination of both, with some modification to the single-TMS34020 approach. Here are three alternatives:

- When operating in self-bootstrap mode, all the TMS34020s will fetch the same code from the same reset vector unless you provide additional mapping to point each processor to a different code. A mapper at the reset vector address can, when read, output a different address, depending on which $\overline{GI}$ pin is active at the time. In this way, each processor reads a different value from the reset vector and therefore fetches code from different locations.

- Alternatively, you can prevent each TMS34020 from fetching the reset vector and executing code by using the multiprocessor interface to inhibit assertion of each TMS34020's $\overline{GI}$ pin. If each TMS34020 becomes bus master in a predefined order at power-up, each can modify the reset vector after loading its own code so that the vector points to the next processor's code.

- You can use a host processor to download code to all the TMS34020s. If you do this, be careful when using host-present mode; the host cannot access the I/O registers of a TMS34020 to which it is not directly attached. This means that if a TMS34020 is powered up halted in host-present mode but is not directly connected to a host, there is no way to clear the HLT bit except through $\overline{RESET}$. The next section outlines the ways in which a multiple-TMS34020 system can be connected to a host.

11.8 Configuration with a Host Processor

If your multiple-TMS34020 system contains a host processor, there are a number of ways to configure the system. The configuration you choose depends on the type of accesses the host needs to make and on the cost of the system. For a general description of host communications, refer to Chapter 7; Section 7.11, Systems with Multiple TMS34020s (page 7-40), discusses the implications of the different configurations in detail.

The $\overline{\text{HOE}}$ and HDST pins support several different configurations.

- Host connected to **one TMS34020 only.** In this case, the $\overline{\text{HOE}}$ and HDST pins of the host-connected TMS34020 should be wired to the external bidirectional data transceivers. There is no need to wire the $\overline{\text{HOE}}$ and HDST pins of the other TMS34020s in parallel.

- Host connected to **more than one TMS34020** via one set of **shared data transceivers.** In this case, the $\overline{\text{HOE}}$ pins from all TMS34020s sharing the data transceivers should be wired together, as should the HDST pins.

- Host connected to **more than one TMS34020** via **individual sets of data transceivers** for each TMS34020. In this case, each TMS34020 is connected to the host via its own set of data transceivers. The HDST and $\overline{\text{HOE}}$ pins should be connected between each TMS34020 and its associated transceivers only.

When bus mastership is regained, HDST and $\overline{\text{HOE}}$ are set to high impedance just like the local-memory control signals. However, unlike the local-memory control signals, they both are attached to internal pull-up resistors. This allows any of the above configurations to be implemented without requiring additional external control circuitry; if HDST and $\overline{\text{HOE}}$ are not driven by another TMS34020 (the first and third cases described above) they are held in the logic-high state by the resistors. Yet, the resistors are sufficiently large that they will not prevent another TMS34020 from driving HDST and $\overline{\text{HOE}}$ subsequently (the second case described above).

Chapter 12

Graphics Instructions and Operations

The TMS34020 is especially useful in graphics systems because it has a large set of graphics-specific instructions. In addition, the TMS34020 supports a variety of special operations (such as windowing, transparency, and options for combining pixels) that you can apply to graphics instructions. This chapter contains four types of information:

12.1 An Overview of Graphics Instructions

The TMS34020 instruction set supports several fundamental graphics drawing instructions. Table 12–1 lists these instructions by categories.

Table 12–1. Summary of Graphics Instructions

	Instructions		Refer to...
Single-Pixel Instructions	*These instructions draw single pixels.*		page 12-6
	DRAV	(draw-and-advance) draws a single pixel at a specified address and then increments that address. This is useful for implementing incremental algorithms that draw circles, ellipses, arcs, and other curves.	
	PIXT	(pixel transfer) transfers individual pixels from one location to another. There are several different versions of the PIXT instruction that use different addressing modes.	
Line Instructions	*These instructions draw single lines or set up the implied operands for drawing a line.*		page 12-7
	LINE	implements the inner loop of Bresenham's algorithm for drawing lines. For this instruction, a line's endpoints are identified by XY addresses.	
	FLINE	is a faster implementation of the LINE instruction. For this instruction, a line's endpoints are identified by linear addresses.	
	LINIT	initializes implied operands required by the LINE and FLINE instructions.	
Pixel-Array Instructions	*These instructions draw 2-dimensional pixel arrays.*		page 12-8
	PIXBLT	(pixel block transfer) copies a 2-dimensional block of pixels from one area in memory to another. There are several forms of the PIXBLT instruction.	
	FILL	sets all the pixels in a pixel array to a known value.	
	PFILL	fills a block with the pattern from the PATTERN register.	
	VFILL	special fill that takes advantage of VRAM features.	
	VBLT	special block transfer that takes advantage of VRAM features.	
Auxiliary Instructions	The TMS34020 supports several auxiliary instructions that extend the functions of the base set of graphics instructions		page 12-17
	FPIXEQ	searches for a pixel that is equal to the COLOR0 pixel.	
	FPIXNE	searches for a pixel that is not equal to COLOR0.	
		FPIXEQ and FPIXNE are useful for seed fills and data compression.	
	EXGPS	exchanges value in PSIZE with the value in a general-purpose register.	
	GETPS	moves the value in PSIZE into a general-purpose register.	
	RPIX	replicates a pixel value within a general-purpose register.	
	CLIP	clips an array to fit within specified window dimensions.	
	CPW	compares a point to the window limits in WSTART and WEND.	
	TFILL	fills a trapezoidal block by filling a series of horizontal lines.	

12.2 An Overview of Graphics Operations

Table 12–2 summarizes the operations that you can apply to graphics instructions. These operations extend the functionality of the graphics instructions, providing you with explicit control over where a pixel is drawn and how it is combined with other pixels. Note that these operations affect only pixel transfers—they do not affect data transfers by nongraphics instructions (such as MOVE).

Table 12–2. Summary of Graphics Operations

	Description	Refer to...
Window Checking	The TMS34020's special window-checking hardware compares a pixel's XY address to the XY addresses within a defined rectangular window. The TMS34020 supports four window-checking modes: ❑ *No window checking*—you can ignore window-checking information. ❑ *Window hit detection* draws no pixels, but generates an interrupt if a pixel lies inside a window. This supports selection of screen objects that a cursor is pointing to. ❑ *Window miss detection* draws pixels that lie within a window and generates an interrupt if a pixel lies outside the window. This is useful for line clipping. ❑ *Window clipping mode* draws only pixels that lie within a window; it generates no interrupts.	page 12-19
Pixel Processing	Most of the graphics instructions move pixels to locations in memory; some instructions combine source pixels with pixels that are already at the specified destination address. The pixel-processing options provide you with control over how the source and destination pixels are combined. You can choose from 16 Boolean and 6 arithmetic pixel-processing options. The default option is simply to replace the destination pixels with the source pixels.	page 12-27
Transparency	Sometimes you may want a background or an existing object to show through an object that you're drawing to the screen. The TMS34020 supports three modes for choosing which of the pixels in an object are transparent: ❑ *Transparency on result = 0* inhibits the pixel write if zero will be written to the destination address. ❑ *Transparency on source = COLOR0* inhibits the pixel write if the source pixel is the same as the COLOR0 pixel. ❑ *Transparency on destination = COLOR0* inhibits the pixel write if the original destination pixel is the same as the COLOR0 pixel.	page 12-36
Plane Masking	The plane mask is a register that you can load with a mask value. The 1s and 0s in the mask tell the TMS34020 which bits in a source or destination pixel can be modified. Transparency and plane masking are similar but work at different levels. Transparency works at the pixel level, preventing an instruction from writing entire pixels. Plane masking, however, works at the bit level, preventing an instruction from writing individual bits within a pixel.	page 12-39

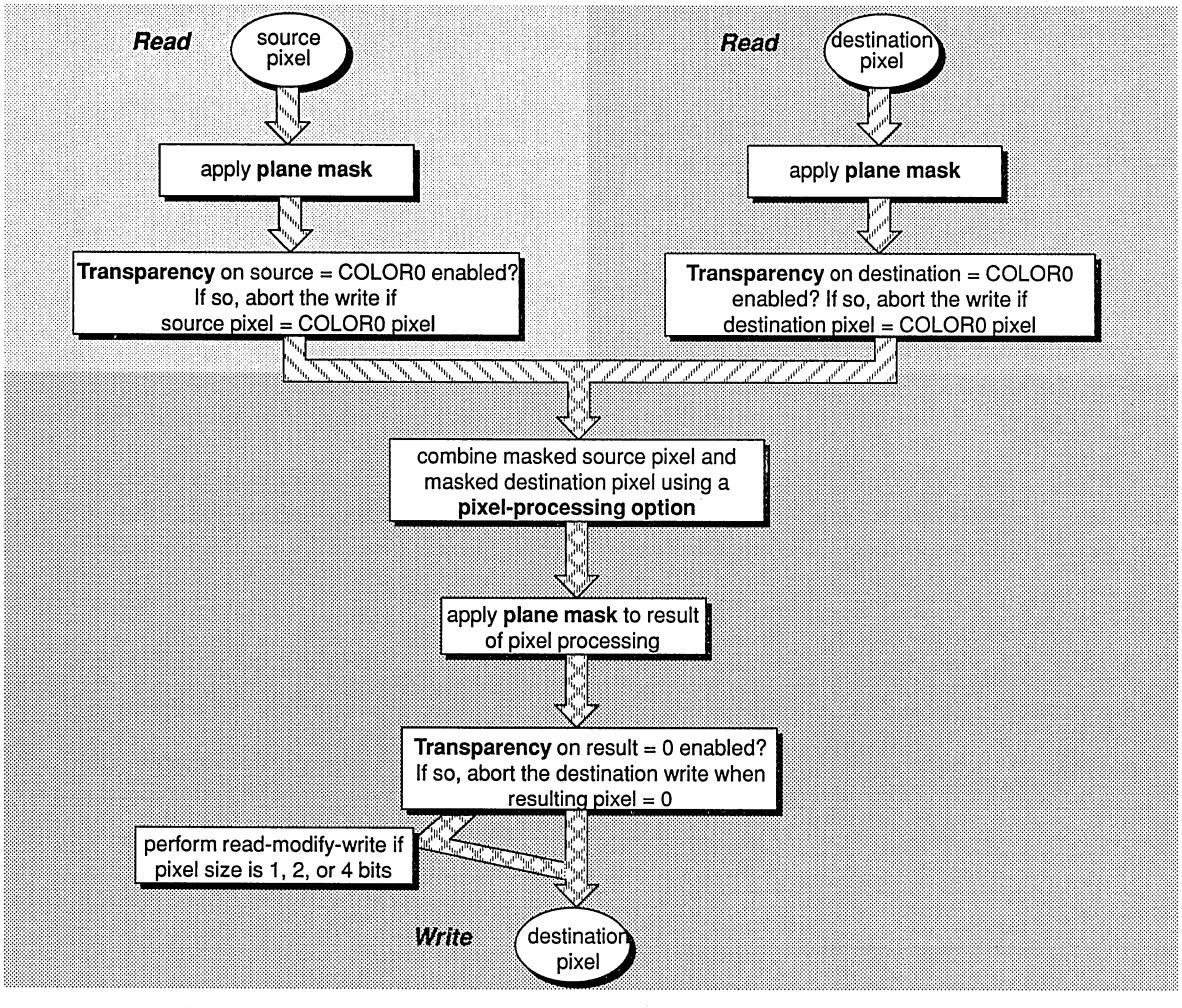
Table 12–2. Summary of Graphics Operations (Continued)

	Description	Refer to...
XY Addressing	XY addressing is not an operation in the sense of transparency or pixel processing; however, using XY addresses instead of linear addresses can simplify program design. Many of the graphics instructions allow you to choose whether you use XY or linear addresses. Because XY addressing allows you to specify an address in terms of Cartesian coordinates, XY addressing is especially useful for manipulating 2-dimensional pixel arrays and for writing pixel data to a screen. For certain instructions, the TMS34020 automatically converts XY addresses to linear addresses for internal use.	page 12-47

Figure 12–1 shows how these operations interact when the TMS34020 transfers a source pixel to a destination location, combining the source pixel data with the destination pixel data. This is a general illustration; certain operations are not performed if they are not enabled or selected.

As Figure 12–1 shows, source and destination pixels are first read from memory and modified by the plane mask. Pixel processing is then performed on the modified pixel values. The plane mask is applied to the result. Bits that are 1s in the PMASK are read as 0s from the source array (however, these same bits are protected during a pixel write), Transparency is applied, depending on the mode chosen.

Figure 12–1. Graphics Operations Interaction



Notes: 1) You can enable only 1 of the 3 transparency modes at any time.

2) If you're using the pixel-processing replace option, the TMS34020 does not read the destination pixel *unless* you've enabled transparency-on-destination or plane masking.

12.3 Single-Pixel Instructions

Definition

A single pixel is defined by

- its **address**, which can be a linear address (this is an absolute memory address) or an XY address. Linear addresses must be aligned on pixel boundaries.
- its **size** (in bits), which is defined in the PSIZE register.

Instructions

Two types of instructions draw single pixels.

- The **PIXT instructions** transfer individual pixels from a source location to a destination location. The PIXTs support three combinations of source-to-destination transfers.

source	destination
A- or B-file register	memory
memory	A- or B-file register
memory	memory

- The **DRAV instruction** draws a pixel that is identified by an XY address. After drawing the pixel, DRAV advances the address by adding an XY increment to the original XY address. The value written to the destination pixel is the value in the COLOR1 register.

  DRAV is especially useful for drawing circles, ellipses, arcs, and other curves. Selecting an appropriate XY increment allows an incremental algorithm to plot each pixel on a curve and then determine where the next pixel will be drawn. The next pixel is usually one of the 8 pixels immediately adjacent to the pixel just drawn.

  You can also use DRAV for drawing straight lines; however, you can draw lines more efficiently with LINE or FLINE.

Graphics operations

You can use the following graphics operations with single-pixel instructions.

window checking	You can use this when the destination pixel has an XY address.
transparency	Use any of the transparency modes.
plane masking	You can set the plane mask.
pixel processing	Use any pixel-processing option, but remember that the arithmetic options are valid only when the pixel size is at least 2 bits.

12.4 Line Instructions

Definition

A line is defined by these implied operands:

- address of one of the line's endpoints
- magnitude of the major and minor axes of the line
- signs of the X and Y increments
- initial value of a decision variable
- initial value of a count variable that monitors drawing progress
- pixel values (in the COLOR0 and COLOR1 registers)
- pixel size (in the PSIZE register)
- pattern (in the PATTERN register)

Instructions

- The **LINE instruction** implements the inner loop of Bresenham's line-drawing algorithm. The endpoint that you provide for LINE must be an XY address. When drawing a line, you might typically

  Step 1: Determine the XY coordinates of the line's endpoints.

  Step 2: Calculate the implied operands listed above. For the most efficient operation, clip the line to the dimensions of the display window before you calculate the implied operands.

  Step 3: Draw the line with the LINE instruction.

- The **FLINE instruction** is a faster implementation of Bresenham's line-drawing routine. FLINE operates similarly to LINE, but the endpoint that you provide for FLINE must be a linear address. Because you cannot use XY addressing with FLINE, FLINE does not support window checking.

- The **LINIT instruction** initializes the implied operands that the LINE and FLINE instructions use. LINIT provides you with a quick, simple method for performing the initialization; LINIT uses a line's XY endpoints to set up the necessary B-file registers. LINIT also sets the status bits so that you can easily determine a line's location in relation to the current window, and you can determine whether a line can be trivially rejected.

Graphics operations

You can use the following graphics operations with the LINE and FLINE instructions.

window checking	You can use window checking with LINE; you can't use window checking with FLINE—FLINE doesn't support XY addressing.
transparency	Use any of the transparency modes.
plane masking	You can set the plane mask.
pixel processing	Use any pixel-processing option, but remember that the arithmetic options are valid only when the pixel size is at least 2 bits.

12.5 Pixel-Array Instructions

Definition

A 2-dimensional pixel array is defined by these implied operands:

- **Starting pixel address.** This is the address of the pixel with the lowest address in the array (in a display, this is usually the pixel at the top left of the array). Two registers supply starting array addresses; these addresses can be linear addresses or XY addresses.

  SADDR (B0) identifies the starting address of a *source array.*

  DADDR (B2) identifies the starting address of a *destination array.*

- **Array width** (DX) and **array height** (DY) in pixels (defined in the DYDX register). The 16 LSBs of DYDX identify the array width; the 16 MSBs identify the height. Source and destination arrays use the same DYDX value.

- **Array pitch** (the difference between the linear addresses of any two vertically adjacent pixels in an array). Two registers supply array pitch values.

  SPTCH (B1) identifies the pitch of a *source array.*

  DPTCH (B3) identifies the pitch of a *destination array.*

- The **pixel size** is defined in the PSIZE register. Only one pixel size is valid at a time, defining the size of all pixels in the array.

Instructions

The TMS34020 supports two types of array instructions.

- The **PIXBLT instructions** are a powerful set of raster operations that combine the pixels in a source array with the pixels in a destination array. The TMS34020 copies each pixel in the source array to the corresponding location in the destination array. The source pixel can simply replace the destination pixel, or the two can be combined using any pixel-processing option.The source and destination arrays must have the same width and height, but they can have different pitches.

  The TMS34020 supports several PIXBLT instructions,with variations in the types of addresses that define the source and destination arrays.

source array	destination array	instructions	
binary	linear	PIXBLT B, L	VBLT
binary	XY	PIXBLT B, XY	
linear	linear	PIXBLT L,L	PIXBLT L,M,L
linear	XY	PIXBLT L,XY	
XY	linear	PIXBLT XY,L	
XY	XY	PIXBLT XY,XY	

- The **FILL instructions** fill a destination pixel array with the pixel value in the COLOR1 register. You can think of a FILL operation as a special type of PIXBLT that does not use a source array.

  The TMS34020 supports two FILL instructions and two special-purpose FILLs, PFILL and VFILL. PFILL is a pattern-fill instruction, and VFILL takes advantage of VRAM block-mode accesses.

destination array	instructions	
linear	FILL L	VFILL
XY	FILL XY	PFILL XY

Graphics operations

You can use the following graphics operations with pixel-array instructions.

window checking

You can use window checking whenever the destination array is identified by an XY address.

transparency

- You can use transparency with any array instruction except VBLT or VFILL..
- The binary PIXBLTs do not apply transparency to unexpanded source data.

plane masking

- You can use the plane mask with any array instruction. Note that for VBLT and VFILL, the masking comes from the VRAM write mask and not from the PMASK register.
- The binary PIXBLTs don't apply the plane mask to the unexpanded source data.

pixel processing

- Use any of the pixel-processing options with the basic PIXBLTs.
- You cannot use pixel processing with VBLT or VFILL.

12.5.1 PIXBLTs with XY and Linear Addressing

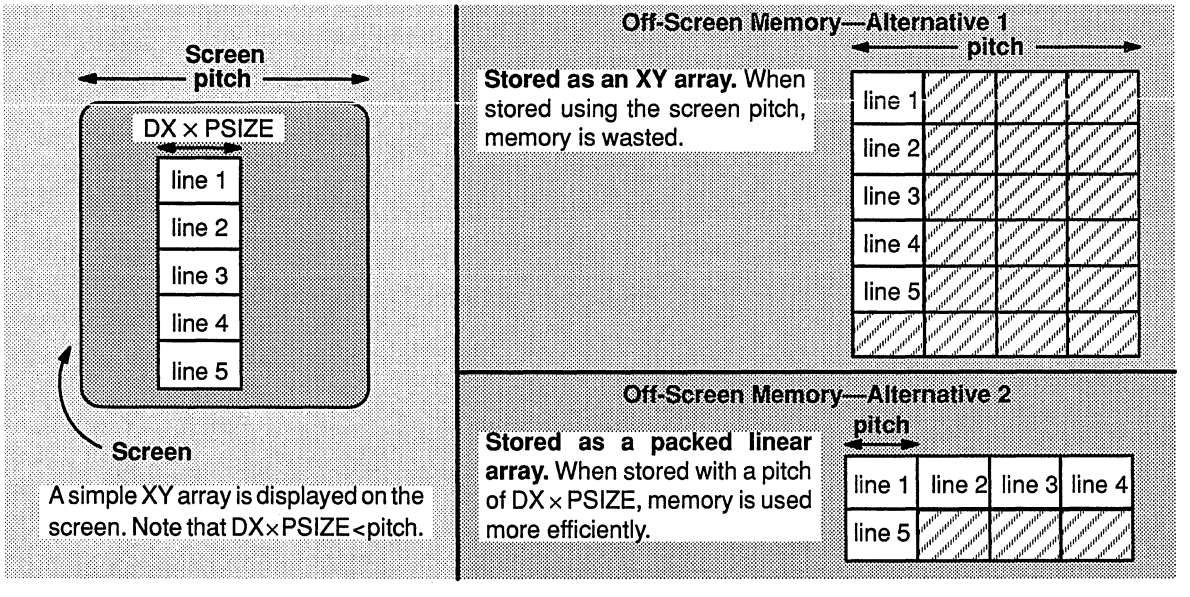
Some PIXBLTs allow you to specify the source and destination arrays' start addresses in XY or linear format. You can use the following combinations of XY and linear arrays.

Source Array		Destination Array	Instruction
Linear	→	Linear	PIXBLT L, L
Linear	→	XY	PIXBLT L, XY
XY	→	Linear	PIXBLT XY, L
XY	→	XY	PIXBLT XY, XY

12.5.1.1 Packed Linear Arrays

XY addressing provides a convenient method for manipulating on-screen objects. XY addressing supports both automatic window clipping and conversion of Cartesian coordinates to memory addresses. When you store pixel arrays off screen, however, you may use memory more efficiently if you use linear addressing. For example, you can pack successive rows of a linear pixel array into memory without gaps between the end of one row and the beginning of the next. The pitch of a linear array is set to the product of the array width (in pixels) and the pixel size (in bits). Figure 12–2 shows how storing an array in packed linear format can save memory space.

Figure 12–2. How XY and Linear Arrays are Stored in Off-Screen Memory



To store an XY array in packed linear form, use the PIXBLT XY, L instruction; to move a packed linear array onto a screen, use PIXBLT L,XY.

12.5.1.2 Selecting the Starting Corner for a PIXBLT

A PIXBLT's default starting address is the lowest pixel address in the source array. As Figure 12–3 shows, however, some PIXBLTs can start at any of the array's four corners. If the source and destination arrays overlap, it may be necessary to change a PIXBLT's starting corner.

Table 12–3 lists the PIXBLTs that can use this corner-adjust feature; note that for some PIXBLTs, the TMS34020 automatically adjusts the corner—for others, you must manually adjust the corner.

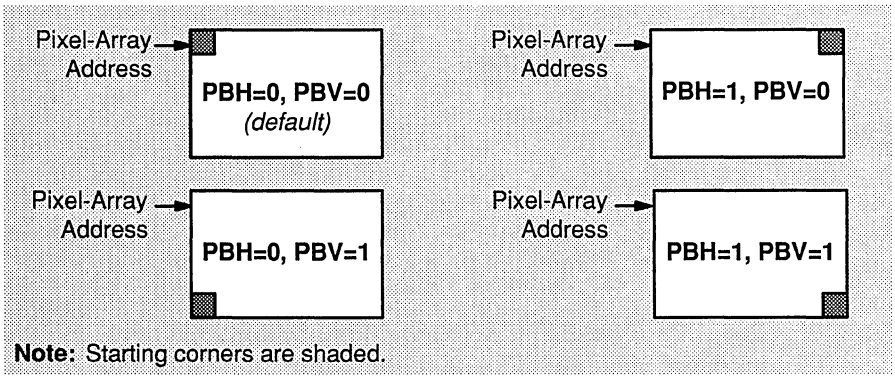
Table 12–3. PIXBLTs That Can Start from Any Corner

Automatic adjust	Manual adjust
PIXBLT L, XY	PIXBLT L, L
PIXBLT XY, L	PIXBLT L, M, L
PIXBLT XY, XY	

By default, a PIXBLT first processes the pixel with the lowest address. The PIXBLT then processes the remaining pixels from left to right within each row, beginning at the top row and moving toward the bottom row. The pixel at the lower right corner of the array is processed last.

When you manually adjust the PIXBLT's starting corner, you must control the sequence in which pixels are moved when the arrays overlap, so that destination pixels do not write over source pixels that haven't been transferred yet.

Figure 12–3. Possible Starting Corners



As Figure 12–3 shows, the PBV[[CONTROL]] and PBH[[CONTROL]] bits determine the starting corner for the PIXBLT. PBH controls horizontal movement; PBV controls vertical movement.

PBH=0 Pixels are processed from **left to right** (increasing X direction).

PBH=1 Pixels are processed from **right to left** (decreasing X direction).

PBV=0 Rows are processed from **top to bottom** (increasing Y direction).

PBV=1 Rows are processed from **bottom to top** (decreasing Y direction).

A PIXBLT processes all the pixels in one row before moving to the next row.

You supply the arrays' default starting addresses in the SADDR and DADDR registers. The starting corner adjustment is automatic or manual, depending on the types of addresses that define the arrays.

- When one or both arrays are XY arrays, the TMS34020 automatically calculates the actual starting address identified by PBH and PBV from the default starting address and the array size. As the PIXBLT executes, it automatically adjusts SADDR and DADDR to the address of the corner selected by PBH and PBV.

- When you use an alternate starting corner with the PIXBLT L,L or PIXBLT L,M,L instructions, *you* must adjust the addresses in SADDR and DADDR to the corner selected by PBH and PBV.

12.5.2 Binary (Color Expanding) PIXBLTs

A **binary array** is a 2-dimensional array of 1-bit pixels. You can use a binary array for storing an object's shape information separately from attributes such as color and intensity. The shape is stored in compressed form as a bitmap of 1s and 0s. The color information is added as a PIXBLT draws the shape on the screen: the 1s in the bitmap are expanded to the COLOR1 value, and the 0s are expanded to the COLOR0 value.

The TMS34020 supports two instructions that allow you to expand a binary array to a linear or an XY array. These PIXBLTs are called **binary PIXBLTs** or **color-expanding PIXBLTs**.

Source Array		Destination Array	Instruction
Binary	→	Linear	PIXBLT B, L
Binary	→	XY	PIXBLT B, XY

Binary PIXBLTs provide several benefits.

- You can easily change foreground and background colors by changing the values in the COLOR0 and COLOR1 registers; this doesn't change the source array.
- When the destination pixel size is greater than 1 bit, some applications (such as storing fonts off-screen in a binary format) require less memory than you would need in a nonbinary format.
- Font loading also takes less time—binary PIXBLTs execute quickly because there is less source data to read.

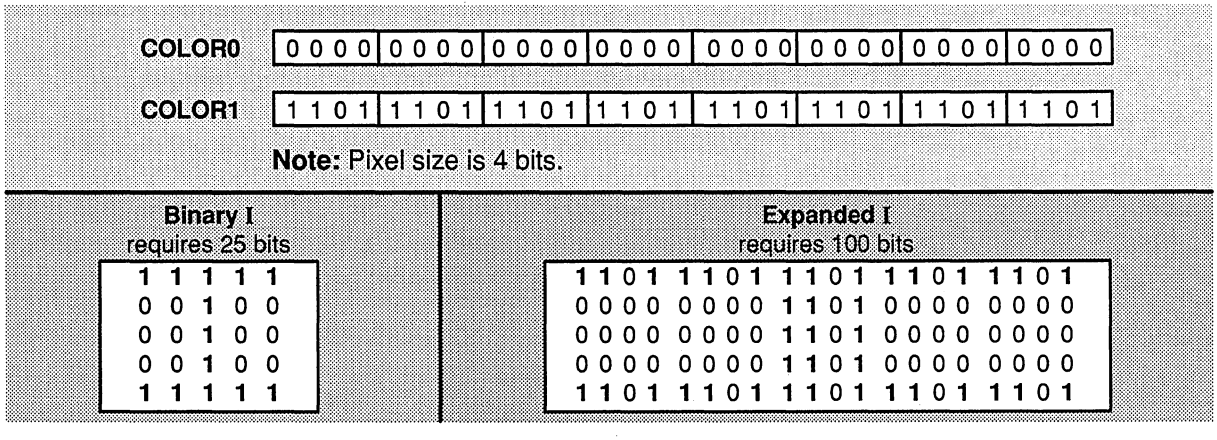
Note:

Binary PIXBLTs expand the source data before performing any pixel processing, transparency, or plane masking. They do not use the plane mask when reading the source data.

Binary PIXBLTs are especially useful in applications that use bitmapped text. Using the XY and linear PIXBLTs to draw and store bitmapped text is cumbersome, requiring you to store a complete copy of each font in off-screen memory. This could consume large amounts of memory. If, for example, you needed each font in several colors, you would need multiple copies of each font (one copy for each color).

Binary PIXBLTs provide a better solution for this application.Typically, text characters appear as a foreground color drawn on top of a background color. Each text character can be stored in a binary bitmap where 1s represent a foreground color (COLOR1) and 0s represent the background color (COLOR0). This way, you don't need a full bitmap representation of each character (where each foreground and background pixel would require several bits of storage). Figure 12–4 shows a sample character that is stored in a binary array and then expanded using COLOR0 and COLOR1.

Figure 12–4. An Example of the Color-Expand Operation



Using binary PIXBLTs, you might follow these steps to write text to the display:

Step 1: Load the text font into memory as one table of binary bitmaps in packed, linear form. (Array pitch = DX × pixel size; for binary arrays, the pixel size = 1, so the pitch for a binary array = DX.)

Step 2: Let COLOR0 contain the background color and COLOR1 contain the foreground color.

Step 3: Use a binary PIXBLT, which will copy the character from memory and expand it onto the display bitmap.

As each character is drawn to the screen, the source array is set to point to the appropriate letter in the binary table. The destination array is defined as the next free space on the screen.

Note:

Set the PSIZE register to the size of the *expanded* destination pixels. Binary PIXBLTs always treat source pixels as 1-bit pixels.

The expand function is also useful in applications that generate shapes or patterns dynamically. During the first stage of this process, a compressed image is constructed in an off-screen buffer area at 1 bit per pixel. The image is built

up of geometric objects such as rectangles, circles, or polygons. Patterns can also be added. When complete, the compressed image is color-expanded onto the screen. This method defers the application of color until the final stage.

12.5.3 Masked PIXBLT

The masked PIXBLT (PIXBLT L,M,L) is an extension of the PIXBLT L,L instruction. The masked PIXBLT uses a **mask array**, a 1-bit-per-pixel array that determines on a pixel-by-pixel basis whether the particular destination pixel is overwritten. In effect, the mask array controls which pixels are forced to be treated as transparent, but this occurs independently of transparency.

The TMS34020 expands the bits in the mask array, then combines them with the plane mask and transparency detection. This combination provides bit-by-bit control of which destination bits are protected from overwriting.

The masked PIXBLT uses two B-file registers:

- MADDR (B10) identifies the mask array's starting address.
- MPTCH (B11) identifies the mask array's pitch.

The masked PIXBLT is useful for manipulating nonrectangular display windows; you could use the mask array to define the shape of the window. You can also use the masked PIXBLT for manipulating patterned objects; load the pattern into the source array and load the object's shape into the mask.

12.5.4 VRAM Block-Mode PIXBLT (VBLT)

The TMS34020 supports a PIXBLT that takes advantage of VRAM block writes. The VBLT instruction is similar to the PIXBLT B,L instruction because it expands a binary source array to a linear destination array; however, there are several important differences.

- VBLT works only when the **entire** destination array lies in VRAM that supports block writes.
- As VBLT reads the source array, its actions depend on the value of the current source pixel.

  pixel value = 0 VBLT does nothing; it does not alter the corresponding destination pixel. VBLT can check an entire line in a source array; if all the values in a line are 0, VBLT doesn't write the line.

  pixel value = 1 VBLT tells the appropriate VRAM to copy the contents of the VRAM color register into the destination.

- VBLT uses the contents of the VRAM color register to represent 1s in the source array. Before executing VBLT, use the VLCOL instruction to copy the contents of the COLOR1 register into the VRAM color register.

- The VRAM color registers are a minimum of 4 bits wide; thus, VBLT supports pixel sizes of 4,8,16, and 32 bits. (VBLT doesn't support 1- and 2-bit pixels.)
- The source and destination array addresses in SADDR and DADDR must contain linear addresses.
- The destination array pitch must be an integer multiple of 128 (80_{16}) bits.
- VBLT executes approximately twice as quickly as PIXBLT B,L. The TMS34020 normally writes 32 bits of data during each memory write cycle. VRAM block mode enables the TMS34020 to write 128 bits during each memory write cycle.
- VBLT **does not** support pixel processing, transparency, XY addressing, or window checking.
- VBLT supports plane masking if VEN[[CONFIG]] is set to 1. VBLT uses the VRAM write mask to perform plane masking.

The sequence of events required to use a VBLT might be

Step 1: Load the required COLOR1 value in B9.

Step 2: Execute VLCOL, copying COLOR1 into the VRAM color registers.

Step 3: Set up the remaining implied operands (SADDR, DADDR, etc.).

Step 4: Execute VBLT.

Note:

For more information about the VRAM block-write, color—mask, and write-mask features, refer to Sections 8.11 (VRAM Write-Mask Local-Memory Cycles, page 8-34) and 8.12 (VRAM Block-Write Local-Memory Cycles, page 8-37).

12.5.5 FILLs

The FILL instruction fills a destination pixel array with the value in the COLOR1 register. A FILL is similar to a PIXBLT that uses no source array. The source pixel value used in pixel processing is the value in the COLOR1 register. The TMS34020 supports two basic FILL instructions; the destination array can be specified in either XY or linear format.

12.5.6 Horizontal Pattern Fill (PFILL)

The PFILL instruction is an extension of the basic FILL instruction. Instead of filling with solid COLOR1, PFILL expands the contents of the PATTERN register (B13). PFILL replaces each 0 in the pattern with a COLOR0 pixel; PFILL replaces each 1 with a COLOR1 pixel. PFILL uses this pattern of pixels to fill each horizontal line of the destination array. The pattern repeats at least once every 32 pixels (it may repeat more frequently, if you like).

For PFILL, the destination array address in DADDR should contain an XY address. PFILL can use window checking.

12.5.7 VRAM Block Mode-Fill (VFILL)

The VFILL instruction is an extension of the basic FILL instruction. VFILL takes advantage of VRAM block-mode memory accesses. VFILL's relationship to the FILL instruction is analogous to VBLT's relationship to the PIXBLT B,L instruction. Note that DADDR must contain a linear address; VFILL cannot use XY addressing or window checking.

Unlike VBLT, VFILL does not need to read data from memory. Thus, VFILL can use back-to-back block-mode writes. VFILL executes approximately four times faster than the corresponding FILL.

One useful application of VFILL is to use it in combination with VBLT to write text to a screen:

Step 1: Load the background color into the COLOR1 register.

Step 2: Execute VLCOL to copy the COLOR1 value into the VRAM color registers.

Step 3: Set up VFILL's implied operands and execute VFILL to write the background color to the screen.

Step 4: Load the foreground color value into the COLOR1 register.

Step 5: Execute VLCOL to copy the COLOR1 value into the VRAM color registers.

Step 6: Set up VBLT's implied operands and execute VBLT to write the text character to the screen.

Note:

For more information about the VRAM block-write, color—mask, and write-mask features, refer to Sections 8.11 (VRAM Write-Mask Local-Memory Cycles, page 8-34) and 8.12 (VRAM Block-Write Local-Memory Cycles, page 8-37).

12.6 Auxiliary Graphics Instructions

In addition to the single-pixel, line, and pixel-array instructions, the TMS34020 supports several auxiliary graphics instructions. These instructions do not draw pixels; they help you to obtain information or set up implied operands for the graphics drawing instructions. (An exception to this is TFILL.)

- **FPIXEQ** and **FPIXNE** (find pixel value)

  These instructions search through memory, comparing pixel values to the pixel value in the COLOR0 register.

  - FPIXEQ stops searching when it finds a pixel that equals the COLOR0 value.
  - FPIXNE stops searching when it finds a pixel that doesn't equal the COLOR0 value.

  Usually, you will want to search through a pixel array in memory. You must supply the starting address for the block of memory that will be searched. The instructions can search forward or backward from the initial location.

  These instructions are especially useful for seed fills and data compression.

- **GETPS** and **EXGPS** (get and exchange pixel size)

  The PSIZE register identifies the current pixel size. The GETPS and EXGPS instructions provide you with easy methods for getting and changing the PSIZE value.

  - GETPS copies the contents of PSIZE into a general-purpose register.
  - EXGPS exchanges the contents of the PSIZE register with the contents of a general-purpose register.

- **RPIX** (replicate pixel)

  The RPIX instruction replicates a pixel value within a general-purpose register. RPIX uses the current pixel size to replicate the pixel value the correct number of times to produce a 32-bit value within the register. For example, if the current pixel size is 4, RPIX will replicate the pixel value 8 times to produce a 32-bit value; if the current pixel size is 16, RPIX will replicate the pixel value twice; etc.

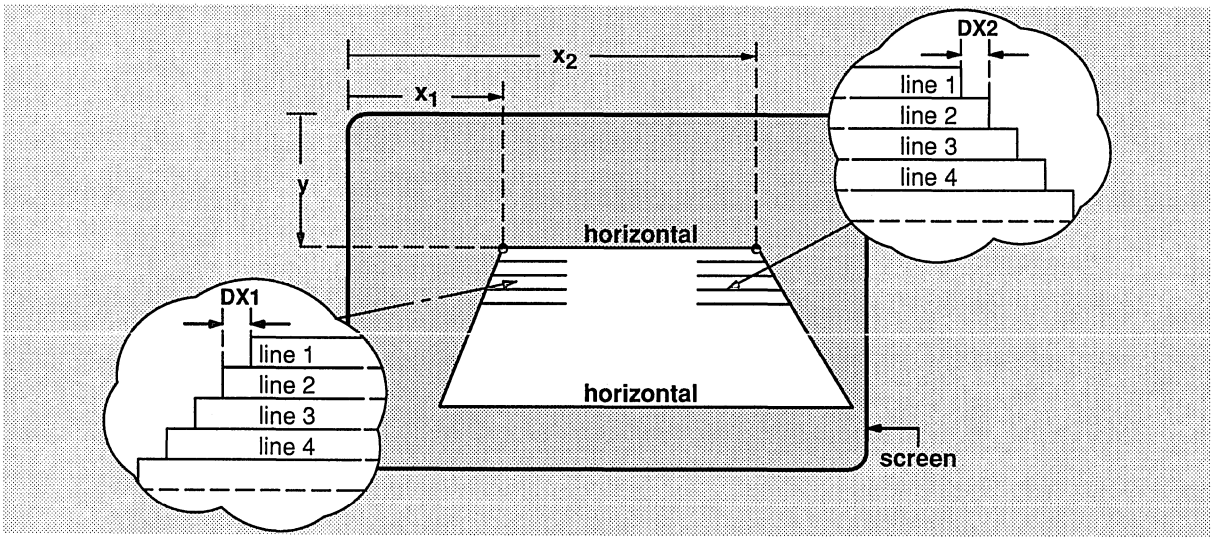
- **CLIP** (array clip)

  The TMS34020 supports a CLIP instruction that adjusts an array specified by the DADDR register and the DYDX register to fit within a window. The adjusted values replace the original values.

❑ TFILL (trapezoidal fill)

The TFILL instruction fills a trapezoidal area by drawing a series of horizontal lines. Figure 12–5 shows a trapezoid that is drawn to the screen.

Figure 12–5. A Trapezoidal Fill



x_1 and x_2 define the length and position of a horizontal line. TFILL draws a line, adjusts x_1 and x_2 by a specified amount, and increments y to point to the next position where a line could be drawn. Executing TFILL once draws only one line; to draw a trapezoid, you must call TFILL an appropriate number of times.

12.7 Window Checking

The TMS34020 allows you to define a rectangular window and to determine if a pixel lies within the window. It does this by comparing the pixel's XY address to the window's starting and ending points.

Window checking allows you to select an area of the screen that can or cannot be affected by pixel writes. Window checking does not affect data writes by non-graphics instructions; it affects only pixel writes by the following instructions:

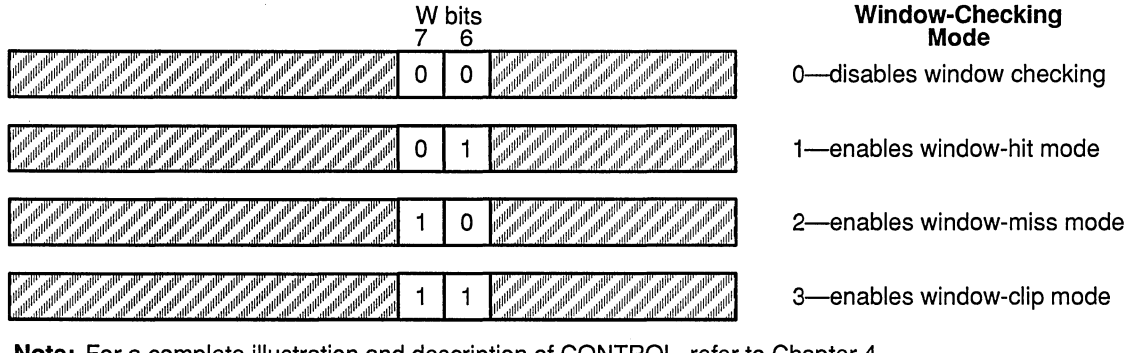
DRAV	PIXTs	LINE
FILL XY	PFILL XY	TFILL
PIXBLT L, XY	PIXBLT XY, XY	PIXBLT B, XY

Note:

Window checking works with XY addresses only—you cannot define a window with linear addresses, nor can you compare a linear address to a window.

The TMS34020 supports four window-checking modes, selected by the value in W[[CONTROL]]. Note that the four modes affect single-pixel, line, and pixel array instructions in different ways.

Figure 12–6. Setting the W[[CONTROL]] Bits to Select a Window-Checking Mode



Note: For a complete illustration and description of CONTROL, refer to Chapter 4.

12.7.1 Defining a Window

To define the window, you must

- Load the window's starting corner (minimum XY address) into WSTART.
- Load the window's ending corner (maximum XY address) into WEND.

Window start and end coordinates are *signed* 16-bit values.

Figure 12–7 illustrates a window in relation to display memory (a screen) and identifies the window's starting and ending points.

Figure 12–7. Specifying Window Limits

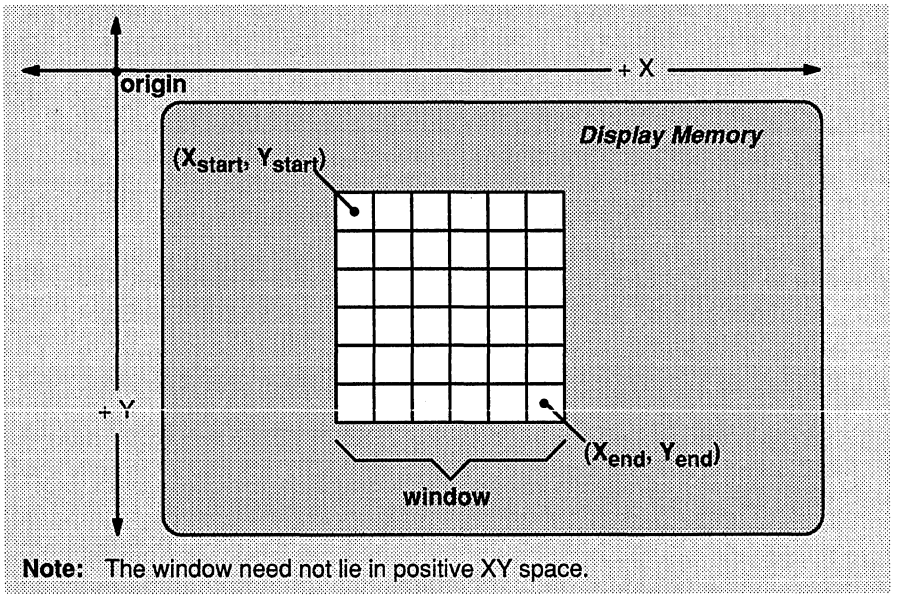


Figure 12–7 shows that a pixel with coordinates (X,Y) lies within the window if $X_{start} \leq X \leq X_{end}$ and $Y_{start} \leq Y \leq Y_{end}$. If a pixel does not meet these conditions, it lies outside the window.

The window is empty when $X_{start} > X_{end}$ or $Y_{start} > Y_{end}$ (that is, the window contains no pixels). In this case, window checking detects all destination pixel addresses as lying outside the window.

Note that the condition $X_{start} = X_{end}$ and $Y_{start} = Y_{end}$ identifies a window that contains a single pixel.

12.7.2 Window-Violation Interrupt

A window-violation interrupt is requested by setting WVP[[INTPEND]] to 1; this happens when either of these situations occurs:

- W=1 and a pixel lies inside the window.
- W=2 and an attempt is made to write to a pixel outside the window.

The interrupt occurs if WVE[[INTENB]] = 1 and IE[[ST]] = 1. Even if the window-violation interrupt is disabled (IE=0 or WVE=0), you can detect a window violation by testing the value of WVP[[INTPEND]].

When a window-violation interrupt occurs, the current graphics instruction aborts. The registers that change during these instructions contain the intermediate values that existed at the time the violation was detected.

12.7.3 Window Checking for Single-Pixel Instructions

Table 12–4 describes the window-checking modes for single-pixel instructions.

Table 12–4. Window-Checking Modes for Single-Pixel Instructions

Mode	Effect
0	Ignore window-checking information
1	*Window hit.* The instruction draws no pixel, but it requests an interrupt if the pixel is **inside** the window. (This is a useful pick mode for mice.)
2	*Window miss.* The instruction draws a pixel only if the pixel lies **inside** the window; the instruction requests an interrupt if the pixel lies **outside** the window.
3	*Window clip.* The instruction draws a pixel only if the pixel lies **inside** the window; however, the instruction requests no interrupts.

The V[[ST]] (overflow) bit identifies the pixel's relationship to the window. When window checking is turned off (window mode 0), the V bit is unaffected. When window checking is enabled (mode 1, 2, or 3), here's how V is affected:

- If the pixel is **outside** the window, V is set to 1.
- If the pixel is **inside** the window, V is cleared to 0.

12.7.4 Window Checking for Pixel-Array Instructions

Table 12–5 describes the window-checking modes for pixel-array instructions.

Table 12–5. Window-Checking Modes for Pixel Array Instructions

Mode	Effect
0	Ignore window-checking information
1	*Window hit.* No pixels are drawn, but the specified destination array is clipped to lie within the window. If any pixel lies within the window, a window-violation interrupt is generated; the DADDR and DYDX registers are adjusted to be the starting address, width, and height that define the intersection of the destination array with the window. This window-checking function is useful for determining the intersection of any two rectangles on the screen.
2	*Window miss.* DADDR and DYDX are compared to the window dimensions. If any pixel in the array lies outside the window, the instruction aborts, and a window-violation interrupt is requested. Unlike window mode 1, however, DADDR and DYDX are not altered as a result of the window comparison (DADDR will change as pixels are drawn). If the entire array lies within the window, then the instruction proceeds.
3	*Window clip.* The instruction draws only those pixels that lie **inside** the window and requests no interrupts. No time is wasted attempting to draw pixels outside the window. DYDX is not changed.

The V[[ST]] (overflow) bit identifies the array's relationship to the window. When window checking is turned off (mode 0), V is unaffected.

12.7.4.1 Detecting a Window Hit (Window-Checking Mode 1)

You can use window-checking mode 1 to pick an object on the screen by moving the cursor to the object's position and selecting it. The actions taken depend on the array's location in relation to the window.

- If the array lies completely **outside,**
  - V is set to 1 and
  - no interrupt is requested.
- If *any part* of the array lies **inside,**
  - V is cleared to 0 and
  - an interrupt is requested.

To determine which object the cursor is pointing to, a program first sets the window to a small region surrounding the position of the cursor. The program then steps again through the same display list used to draw the current screen until one of the objects causes a window interrupt. The object causing the interrupt should be the object that the cursor is pointing to. If no object causes an interrupt, the pick window can be enlarged and the process repeated until the object is found. If two objects cause interrupts, the size of the pick window can be reduced until only one object causes an interrupt.

12.7.4.2 Detecting a Window Miss (Window-Checking Mode 2)

Window-checking mode 2 permits a PIXBLT or FILL instruction to be aborted if any pixel in the destination array lies outside the window. The actions taken depend on the array's location in relation to the window.

- If *any part* of the array lies **outside**,
  - V is set to 1,
  - the instruction aborts, and
  - an interrupt request is generated.
- If the array lies completely **inside**,
  - V is cleared to 0,
  - no interrupt request is generated, and
  - the entire destination array is written.

12.7.4.3 Window Clipping (Window-Checking Mode 3)

In this mode, the TMS34020 draws pixels that lie inside the window. The actions taken depend on the array's location in relation to the window.

- If the array lies completely **outside**,
  - V is set to 1 and
  - no interrupt is requested.
- If the array lies **partly inside** and **partly outside**,
  - V is set to 1,
  - no interrupt is requested, and
  - the portion of the destination array lying within the window is written.

- If the array lies completely **inside**,
  - V is cleared to 0,
  - no interrupt is requested, and
  - the entire destination array is written.

When the instruction begins executing, the destination array is automatically preclipped to lie within the window before the first pixel is transferred. No execution time is lost attempting to write destination pixels that lie outside the window. In the case of a PIXBLT, the source array is also preclipped to fit the adjusted dimensions of the destination array before the transfer begins.

12.7.4.4 Clip Instruction for Preclipping a Pixel Array

The TMS34020 supports a CLIP instruction that adjusts the array specified by the DADDR register and the DYDX register to fit within a window. The adjusted values replace the original values. The status bits are set as follows:

Z set to 1 if the array lies entirely outside the window, cleared to 0 if any part of the array lies within the window.

V set to 1 if any part of the array lies outside the window, cleared to 0 if the array lies entirely within the window

The CLIP instruction is especially useful when there is a possibility that the array could overflow the maximum X and/or Y dimensions of the screen; such an array is called an **overflowing array**.

CLIP is the only instruction that can handle window overflows. If you think that executing a particular instruction could cause a window overflow, do not use a PIXBLT with an XY destination address. Instead, use CLIP, CVXYL, then a PIXBLT with a linear destination address.

12.7.5 Window Checking for the LINE Instruction

Table 12–6 describes window-checking modes for the LINE instructions.

Table 12–6. Window-Checking Modes for the LINE Instruction

Mode	Effect
0	Ignore window-checking information
1	*Window hit.* Points on the line are calculated, but no pixels are drawn. As soon as the line moves inside the window, V is cleared, the line-draw aborts, and the instruction requests an interrupt. (V is cleared to 0 if part of the line lies inside the window; otherwise, V is set to 1.)
2	*Window miss.* Points on the line are calculated and drawn. As soon as the line moves outside the window, V is set to 1, the line aborts, and the instruction requests an interrupt. (V is set to 1 if part of the line lies outside the window; otherwise, V is cleared.)
3	*Window clip.* All points on the line are calculated; points that lie inside the window are drawn. V is cleared to 0 if the last pixel on the line lies inside the window; otherwise, V is set to 1.

12.7.5.1 Line Clipping

The TMS34020 supports two methods for clipping a straight line to the boundaries of a rectangular window.

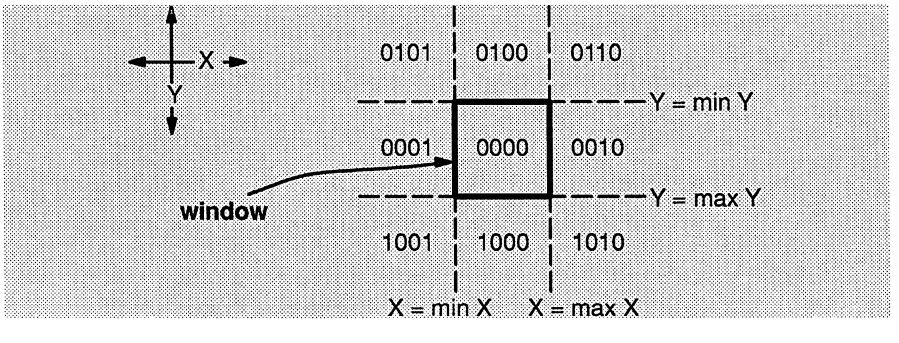
- In **postclipping**, each pixel on the line is compared to the window limits just before being drawn. If the pixel lies outside the window, the write is inhibited. W=3 window-checking mode is selected; window checking is automatically performed in parallel with execution of the LINE instruction, so no overhead is added to the time to draw a pixel. However, unless this form of clipping is used carefully, another type of overhead may become significant. For example, in a CAD environment, if only a small portion of a system diagram is displayed at once, a great deal of time could be spent performing calculations for points (or entire lines) that lie off screen.
- **Preclipping** determines, before drawing, which pixels in a line lie within the window. The algorithm draws only these pixels and makes no attempt to write pixels outside the window. A preclipped line may take less time to draw since no drawing calculations are performed for pixels lying outside the window. Preclipping is usually faster than postclipping, depending on the likelihood of a line lying outside the window.

The first step in preclipping a series of lines is to identify any lines that lie either entirely inside or outside the window. This is accomplished by using an outcode technique similar to that of the Cohen-Sutherland algorithm.

- Lines lying entirely outside the window are trivially rejected and consume no more processing time.
- Lines lying entirely inside the window are drawn.
- Any remaining lines cross one or more window boundaries and require intersection calculations to identify portions lying within a window.

The Cohen-Sutherland method for determining where a line lies in relation to a window uses **outcodes** that identify the location of the line's endpoints in relation to the window. Figure 12–8 illustrates the outcodes assigned to an endpoint that falls within certain window regions.

Figure 12–8. Outcodes for Line Endpoints



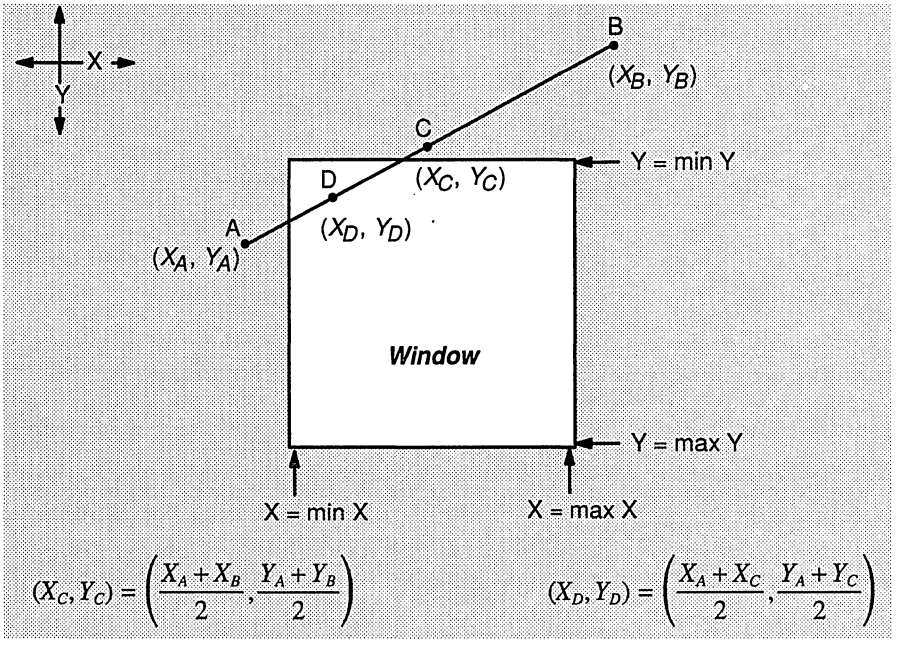
In this method, each region is assigned a 4-bit outcode. When an endpoint of a line falls within a particular region, it is assigned the outcode for that region. The outcodes determine the treatment of the lines.

- The outcode within the window is 0000_2. If the two endpoints of a line both have outcodes 0000_2, the line lies entirely within the window and is drawn.
- If the bitwise-AND of the outcodes of the two endpoints $\neq 0000_2$, the line lies entirely outside the window and is rejected.
- Lines that fall into neither of these categories may be partially visible within the window. Visible portions must be identified and drawn.

The TMS34020's CPW (compare point to window) instruction is a single-cycle instruction that compares an endpoint to all sides of the window.

Midpoint subdivision is an efficient method for finding the in-window portion of a line that crosses a window boundary. This method ensures that drawing calculations are performed only for pixels lying within the window. Figure 12–9 illustrates the midpoint-subdivision technique.

Figure 12–9. Using Midpoint Subdivision to Determine Which Portion of a Line Lies Within a Window



In Figure 12–9, line *AB* lies partially within the window. Here's how midpoint subdivision finds the portion that lies within the window.

Step 1: Determine the coordinates of the line's midpoint at *C*.

Step 2: Comparing the outcodes of *B* and *C* shows that segment *BC* lies entirely outside the window and can be trivially rejected.

Step 3: Subdivide the part of segment *AC* that lies within the window.

Step 4: Determine the coordinates of point *D*, which is the midpoint of *AC*. Point *D* lies within the window.

Step 5: Now invoke the LINE instruction twice, once for segment *DC* and again for segment *DA*. Point *D* is the starting point for both cases. Window-checking mode 2 is used while drawing both segments, but the window-violation interrupt is disabled. When each line crosses the window boundary, the TMS34020 detects this and aborts the LINE instruction. Thus, the LINE instruction performs drawing calculations for only those portions of *DA* and *DC* that lie within the window.

12.7.5.2 Using LINIT and FLINE for Preclipped Line Drawing

The TMS34020's LINIT instruction uses the XY coordinates of a line's endpoints (X_0, Y_0 and X_1, Y_1) to determine the operands required for the LINE and FLINE instructions. LINIT also sets up the status bits as follows.

Status bit set to...	When...	Notes
N = 1	$X_0 = X_1$	The line is vertical
C = 1	CPW (X_0,Y_0) AND CPW (X_1,Y_1) $\neq 0$	The line lies entirely outside the window
Z = 1	$Y_0 = Y_1$	The line is horizontal
V = 1	(X_0,Y_0) or (X_1,Y_1) lies outside the window	

After you execute the LINIT instruction, you can examine the status bits to identify the following types of lines.

- a line that lines **entirely outside a window** (C=1)
- a **horizontal** line (Z = 1)
- a **vertical** line (N = 1)
- a **point** (Z=1 and N=1)
- a line that lies **partially within a window** (you can then use the midpoint subdivision and the LINE instruction)
- a line that lies **entirely inside a window** (use FLINE to draw the line)

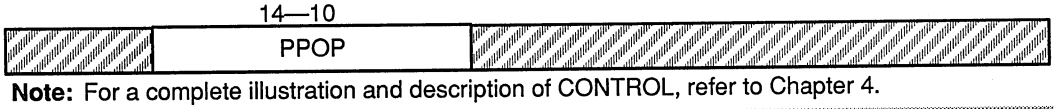
12.8 Pixel Processing

Pixel processing (sometimes known as raster processing) controls how source pixels are combined with destination pixels. When the TMS34020 reads a pixel from its source location, the processor combines the pixel with the corresponding destination pixel using the selected pixel-processing option. The TMS34020 writes the result to the destination pixel location. The TMS34020 supports 16 Boolean and and 6 arithmetic pixel-processing options.

- Boolean operations are performed in bitwise fashion on pixels of 1, 2, 4, 8, 16, or 32 bits.
- Arithmetic operations treat operand pixels of 2, 4, 8, 16, or 32 bits as unsigned binary numbers. The arithmetic operations are especially useful for encoding color or intensity information when a display uses multiple bits per pixel. For example, the MAX and MIN operations allow two objects with antialiased edges to be smoothly merged into a single image.

The PPOP[[CONTROL]] selects the pixel-processing option. Figure 12–10 shows the PPOP bits in the CONTROL register.

Figure 12–10. The PPOP [[CONTROL]] Bits



Note: For a complete illustration and description of CONTROL, refer to Chapter 4.

Table 12–7 lists all of the TMS34020's pixel-processing options.

Table 12–7. Pixel-Processing Options

	Code	Operation	Code	Operation
Boolean Operations	00000	Source → Destination	01000	Source OR Destination → Destination
	00001	Source AND Destination → Destination	01001	Destination → Destination
	00010	Source AND ~Destination → Destination	01010	Source XOR Destination → Destination
	00011	0s → Destination	01011	~Source AND Destination → Destination
	00100	Source OR ~Destination → Destination	01100	1s → Destination
	00101	Source XNOR Destination → Destination	01101	~Source OR Destination → Destination
	00110	~Destination → Destination	01110	Source NAND Destination → Destination
	00111	Source NOR Destination → Destination	01111	~Source → Destination

Table 12–7. Pixel-Processing Options (Continued)

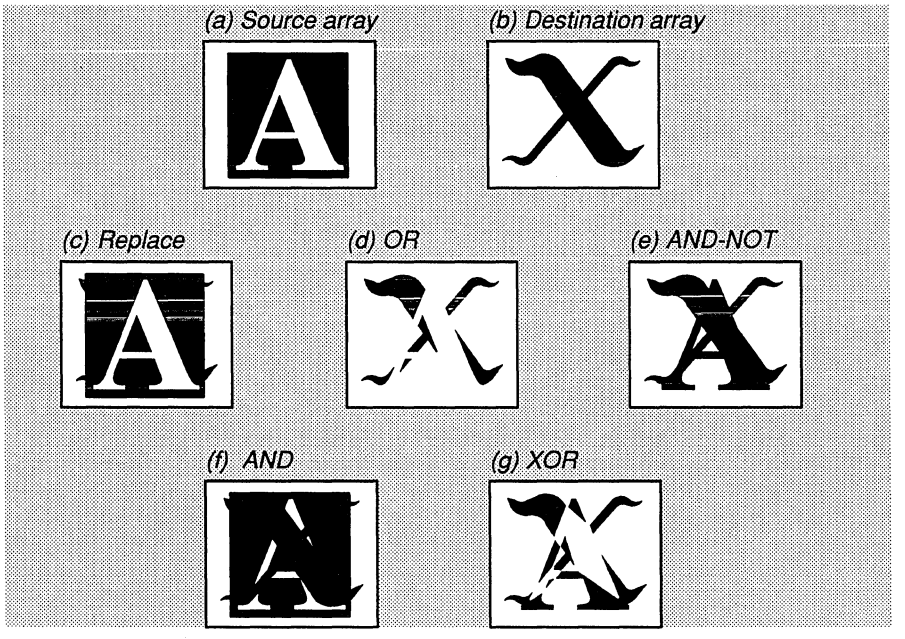
	Code	Operation	Code	Operation
Arithmetic Operations	10000	Source + Destination → Destination	10010	MAX(Source, Destination) → Destination
	10001	Source ADDS Destination → Destination	10101	MIN(Source, Destination) → Destination
	10010	Destination – Source → Destination	10110—11111	Reserved
	10011	Source SUBS Destination → Destination		

Pixel-processing options 10000_2 and 10010_2 correspond to standard 2s complement addition and subtraction. A result that overflows the specified pixel size causes the pixel value to wrap around within its 2-, 4-, 8-, 16-, or 32-bit range. Carry bits, however, are prevented from propagating to adjacent pixels.

12.8.1 Boolean Processing Examples

Figure 12–11 illustrates the effects of five commonly used Boolean operations when applied to 1-bit pixels. Black regions contain 0s, and white regions contain 1s. Figure 12–11 *(a)* and *(b)* show the original source and destination arrays. The source operand in *(a)* is the letter ***A***. The destination in *(b)* is a calligraphic-style ***X***.

Figure 12–11. Examples of Operations on Single-Bit Pixels



- **Replace destination with source.** A simple replacement operation overwrites the pixels of the destination array with those of the source. Figure 12–11 *(c)* shows the letter A written over the center portion of a larger X using the replace operation. The rectangular region around the letter A obscures a portion of the X lying outside the A pattern. Other operations allow only those pixels corresponding to the A pattern within the rectangle to be replaced, permitting the background pattern to show through. These are the logical-OR and logical-AND-NOT (NOT source AND destination) operations. The replace-with-transparency operation performs similarly in color systems.

- **Logical-OR of source with destination.** Figure 12–11 *(d)* illustrates the use of the logical-OR operation during a PIXBLT. For a 1-bit-per-pixel display, the OR function leaves the destination pixels unaltered in locations corresponding to 0s in the source pixel array. Destination pixels in positions corresponding to 1s in the source are forced to 1s.

- **Logical-AND of NOT-source with destination.** Logically ANDing the negated source with the destination is complementary to the logical-OR operation. Destination pixels corresponding to 0s in the source array remain unaltered, but those corresponding to 1s in the source are forced to 0s. Figure 12–11 *(e)* shows an example of the AND-NOT PIXBLT operation (notice the negative image of the letter A). For comparison, Figure 12–11 *(f)* shows the result of simply ANDing the source and destination.

- **Exclusive-OR of source with destination.** The XOR operation is useful for making patterns stand out on a screen in instances where it is not known in advance whether the background will be 1s or 0s. At every point where the source array contains a pixel value of 1, the corresponding pixel of the destination array is flipped; that is, a 1 is converted to a 0, and vice versa. XOR is a reversible operation; by XORing the same source to the same destination twice, the original destination is restored. These properties make the XOR operation useful for placing and removing temporary objects such as cursors, and in rubberbanding lines. As Figure 12–11 *(g)* shows, however, the object may be difficult to see if both the source and destination arrays contain intricate shapes.

12.8.2 Multiple-Bit Pixel Operations

The Boolean operations described in Section 12.8.1 are sufficient for single-bit pixel operations. However, they may be inappropriate for multiple-bit pixel operations, especially when you are using color. For example, the result of a bitwise-OR operation on a black-and-white (1-bit-per-pixel) display is easily predicted: ORing black and white yields white. However, the result of this operation is less intuitive when applied to multiple-bit pixels. For example, in a population-density map, colors may be used to represent numeric values. If one color, such as red, represents one level of population density, and blue represents another, what happens when the two colors are bitwise-ORed? When pixels represent numeric values, numerical operations such as addition and subtraction yield more useful results.

Boolean operations are usually inadequate for merging antialiased objects into a single bitmapped image. Older graphics systems that are limited to Boolean operations on pixels are incapable of supporting many practical applications on multiple-bit-per-pixel images. For instance, where two antialiased lines cross, AND and OR operations yield chaotic pixel intensities that defeat the purpose of the antialiasing. However, merging the two lines by means of the TMS34020's MAX operation (for white on black) or MIN operation (for black on white) yields a smooth and aesthetically pleasing image.

12.8.2.1 Examples of Boolean and Arithmetic Operations

Figure 12–12 illustrates Boolean and arithmetic operations on multiple-bit pixels.

- Figure 12–12 *(a)* illustrates the source array, which contains a red letter ***A***. The red pixels have the value 8 (1000_2) and the black background pixels have the value 0 (0000_2).

- Figure 12–12 *(b)* shows the destination array, which contains a yellow ***X***. The yellow pixels have the value 12 (1100_2) and the pixels in the blue background pixels have the value 2 (0010_2).

Boolean operations can be applied to multiple-bit pixels by combining the corresponding bits of each pair of source and destination pixels on a bit-by-bit basis according to the specified Boolean operation. Figure 12–12 *(c)* through *(g)* shows the effects of combining the source and destination arrays using the replace, OR, AND-NOT, AND, and XOR PIXBLT operations. Compare these to Figure 12–11 (page 12-28).

Arithmetic operations treat 2-, 4-, 8-, 16-, and 32-bit pixels as unsigned binary numbers. An n-bit pixel represents a positive integer in the range 0 to $2n$–1 (all 1s). Examples of arithmetic operations on source and destination pixels are shown in Figure 12–12 *(i)* through *(n)*.

Figure 12–12. Examples of Boolean and Arithmetic Operations

(a)Source

(b)Destination

(c)Source replaces destination

(d)Source OR destination

(e)Source AND destination

(f)Source AND destination

(g)Source XOR destination

(h)Replace with transparency

(i)Add

(j)Subtract

(k)Add with saturation

(l)Subtract with saturation

(m)MAX

(n)MIN

- **Figure 12–12 *(i)* and *(j)*: Simple addition and subtraction.** (i) shows the result of adding the source and destination arrays, using simple binary 2s complement addition. When the sum of the two pixels exceeds the maximum pixel value, the result overflows. Figure 12–12 *(j)* shows the result of subtracting the source array from the destination array. Underflow occurs for those pixels whose calculated difference is negative.

  Simple addition and subtraction are complementary operations. They are reversible operations in the same sense as the XOR operation: by adding a source pixel to a destination pixel, and then subtracting the same source pixel, the original destination pixel is recovered.

- **Figure 12–12 *(k)* and *(l)*: Add and subtract with saturate.** The arithmetic add and subtract operations are binary 2s complement operations that allow overflow and underflow. An add-with-saturate operation stops the result at the maximum unsigned value without allowing the result to overflow. For example, with 4 bits per pixel, adding 0010_2 to 1110_2 produces 1111_2. Similarly, a subtract-with-saturate operation stops the result at 0 without allowing it to underflow.

  Figure 12–12 *(k)* and *(l)* illustrates addition with saturation and subtraction with saturation. In these examples, the pixel size is 4 bits. By dedicating a different color to each value, the effects of each PIXBLT operation become more visible.

  An alternate method of encoding 4-bit pixels uses the 16 values 0 to 15 to represent increasing intensities of a single color component: red, green, or blue. The addition and subtraction operations now have obvious meaning: increasing or decreasing the intensity by specified amounts. At 12 bits per pixel, 4 bits of intensity can be dedicated to each of the three color components. Arithmetic operations are then performed on the corresponding components of each pair of source and destination pixels.

  Figure 12–13 (page 12-33) presents examples in which the pixel values represent intensities of a gray from black to white.

- **Figure 12–12 *(m)*: Maximum.** MAX compares two pixel values and replaces the destination pixel with the larger value. In some respects, MAX is the arithmetic equivalent of the Boolean OR function (compare Figure 12–12 *(m)* with Figure 12–11 *(b)*). The use of MAX in gray-scale and color displays is similar to that of OR in simple black and white.

  If the MSBs in each pixel are assigned to represent object priority (whether an object appears in front of or behind another object), the MAX operation can be used to replace only those pixels of the destination array whose priorities are lower than those of the corresponding pixels in the source array. This allows an object to be drawn to the screen so that it appears either in front of or behind other objects previously drawn. In Figure 12–12 *(m)*, the red A has a numerical value that is greater than that of the blue background, but less than that of the X.

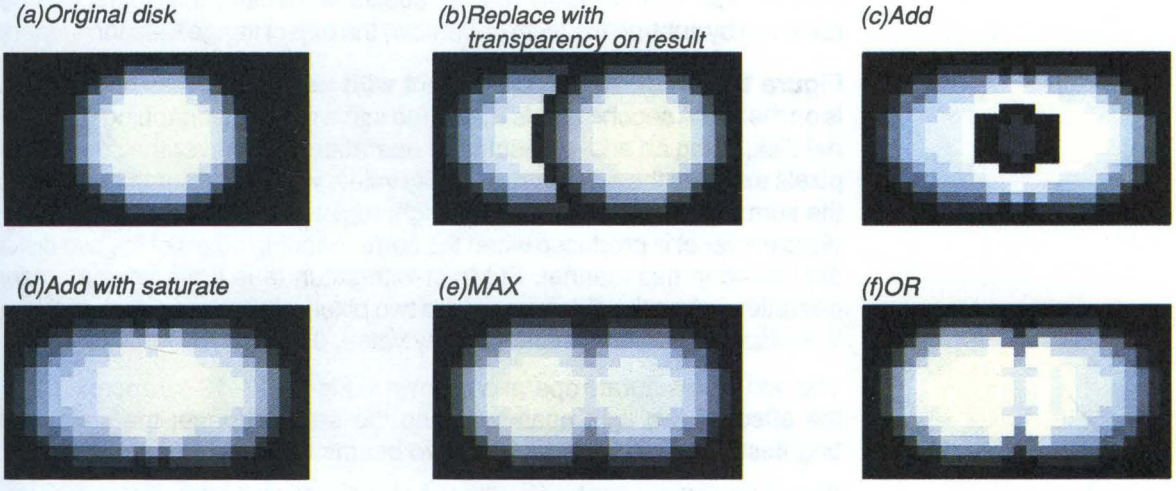
The MAX function is also useful for smoothly combining two antialiased objects that overlap.

- **Figure 12–12 *(n)*: Minimum.** Figure 12–12 *(n)* illustrates the results of the MIN operation on the source and destination arrays. MIN compares two pixel values and replaces the destination pixel with the smaller value. MIN is similar to the Boolean AND function. MIN can be used with priority-encoded pixel values, similar to MAX, but the effect is reversed. In Figure 12–12 *(n)*, the priorities of the two objects are reversed from that of the MAX example shown in Figure 12–12 *(m)*. The MIN operation also has uses similar to those of MAX in smoothly combining antialiased objects that overlap.

12.8.2.2 Operations on Pixel Intensity

Figure 12–13 illustrates the visual effects of various PIXBLT operations on two intersecting disks. In these examples, each pixel is a 4-bit value representing an intensity from 0 (black) to 15 (white). Before the PIXBLT operation, only a single disk resides on the screen, as Figure 12–13 *(a)* shows. The intensity of the disk is greatest at the center (where the value is 12), and gradually falls off as the distance from the center increases. Figure 12–13 *(b)* through *(f)* shows the effects of combining a second, identical disk with the first. Figure 12–13 *(b)* through *(e)* is produced using arithmetic operations; *(f)* is the result of a logical-OR of the source and destination.

Figure 12–13. Examples of Operations on Pixel Intensity



The gradual change in intensity at the edge of the disk in Figure 12–13 *(a)* is similar to the result produced by antialiasing techniques that reduce jagged-edge effects. A text font might be stored in antialiased form, for example, to give the text a smoother appearance. When two characters from the font table are

PIXBLTed to adjacent positions on the screen, they may overlap slightly. The particular arithmetic or Boolean operation selected for the PIXBLT determines the way in which the antialiased edges of the characters are combined within regions of overlap.

- **Figure 12–13 *(b)*: Replace with transparency on result.** A second disk is PIXBLTed into a position near the first disk, using replace-with-transparency on result=0. Those pixels of the first disk that lie within the rectangular region containing the second disk, but are not part of the second disk, remain intact. Visually, the second disk (at the right) appears to lie in front of the original disk (at the left). However, assuming that the gradual change in intensity at the perimeter of the disks is done for the purpose of antialiasing, the sharp edge that results where the second disk covers the first defeats this purpose. In other applications, this sharp edge may be desirable; for example, it might be used to make a text character or a cursor stand out from the background. The replace-with-transparency-on-result operation also supports object priority by writing objects to the screen in ascending order of priority.

- **Figure 12–13 *(c)*: Add with overflow and subtract with underflow.** A second disk is PIXBLTed into an area overlapping the first disk, using an add-with-overflow operation. In this example, when 1 is added to an intensity of 15, the sum is truncated to 4 bits to produce the result 0. The effect of arithmetic overflow is visible at the intersection of the two disks as discontinuities in intensity.

  This effect is useful for making objects stand out against a cluttered background. Add with overflow has an additional benefit: the object can be removed by subtracting (with underflow) the object image from the screen.

- **Figure 12–13 *(d)*: Add and subtract with saturation.** The original disk is on the left. A second disk is PIXBLTed into a region overlapping the original disk, using an add-with-saturate operation. Whenever the sum of two pixels exceeds the maximum intensity value, which is 15 for this example, the sum is replaced with 15. The bright region that occurs where the two disks intersect is produced when the corresponding pixels of the two disks are added in this manner. Subtract-with-saturate is the complementary operation; when the difference of the two pixel values is negative, the sum is replaced by the minimum intensity value, 0.

  The add-with-saturate operation shown in Figure 12–13 *(d)* approximates the effect of two light beams striking the same surface; the surface is brightest in the area in which the two beams overlap.

  These operations can be used to achieve an effect similar to that of an airbrush in painting. Consider a display system that represents each pixel as 12 bits and dedicates 4 bits each to represent the intensities of the three color components, red, green, and blue. This method permits the intensity of each component to be directly manipulated. With each pass of the

simulated airbrush over the same area of the screen, the color changes gradually toward the color of the paint in the airbrush. For example, assume that the paint is yellow (a mixture of red and green). Each time a pixel is touched by the airbrush, the intensity of the red and green components is increased by 1, and the intensity of the blue component is decreased by 1. With each sweep of the airbrush, the affected area of the screen turns more yellow until the red and green components reach the maximum intensity value (and are not allowed to overflow), and the blue component reaches 0 (and is not allowed to underflow).

- **Figure 12–13 *(e)*: MAX and MIN operations.** The original disk is on the left. A second disk is PIXBLTed into the rectangular region to its right using the MAX operation. In the region in which the disks overlap, each pair of corresponding pixels from the two disks is compared and the greater value is selected. This produces a relatively smooth blending of the two disks. Unlike add with saturate, the MAX function does not generate a hot spot where two objects intersect.

  The visual effect achieved using the MAX operation is desirable in an application, for instance, in which white antialiased lines are constructed on top of each other over a black background. MAX also smooths out places in which the lines are overlapped by antialiased text. MAX is successful in maintaining two visually distinct antialiased objects, while the add-with-saturate tends to run them together.

  MIN, which is complementary to MAX, can be used similarly to smooth the appearance of intersecting black antialiased lines and text on a white background.

  The MAX and MIN operations are particularly useful in color applications in which the number of bits per color gun is small (8 bits or less). Other operators could also be used to smooth the transition between the two overlapping antialiased objects in Figure 12–13 *(e)*, but any additional accuracy attained by using a more complex smoothing function would probably be lost in truncating the result to the resolution of the integer used to represent the intensity at each point.

12.9 Transparency

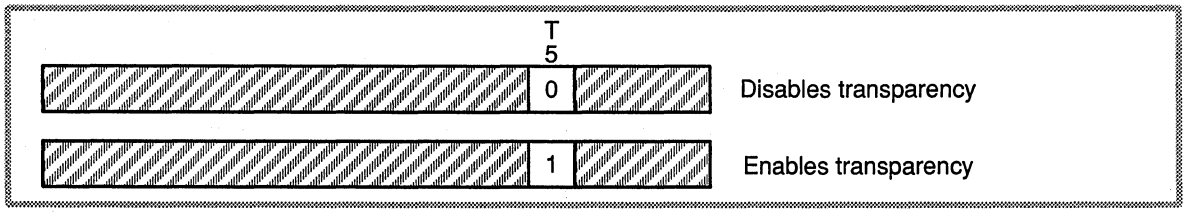
Transparency is a graphics operation that allows you to decide which pixels are visible. If you want to draw an object on a background or on top of another object, you can make pixels in the current object transparent so that the background or object beneath shows through. In some cases, several pixels in a rectangular pixel array that contains an object may not be part of the object itself; you can use transparency to hide the pixels that aren't part of the object.This is useful for ensuring that only the object, and not the rectangle surrounding it, is drawn on the screen.

The CONTROL register controls transparency:

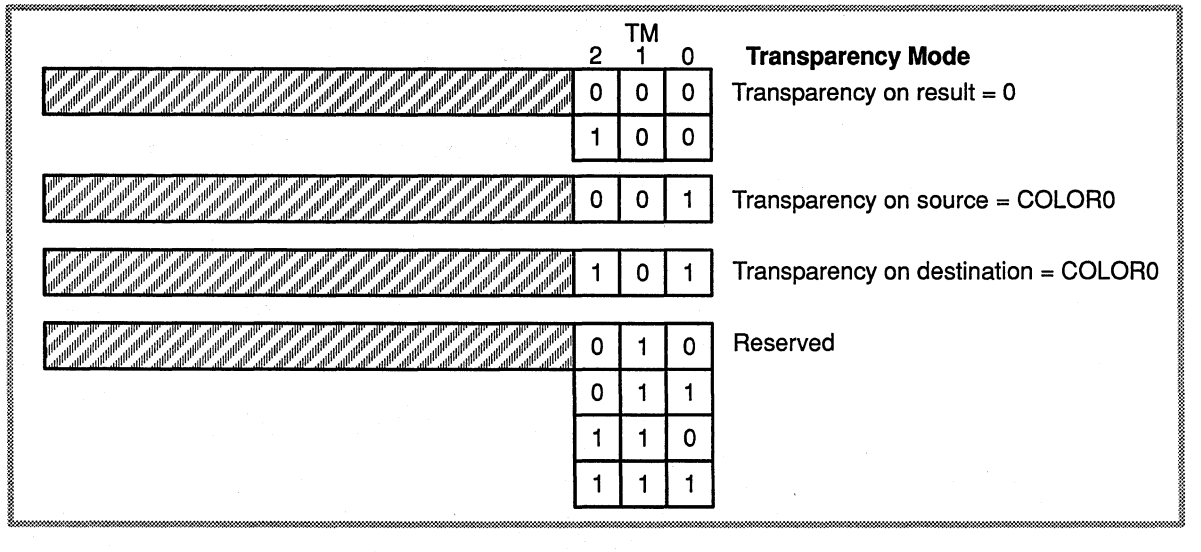
- T[[CONTROL]] (bit 5) enables transparency.
- TM[[CONTROL]] (bits 0—2) determines which transparency mode is selected when transparency is enabled.

Figure 12–14. Enabling Transparency and Selecting a Transparency Mode

(a) Setting the T bit to enable or disable transparency



(b) Setting the TM bits to select the transparency mode

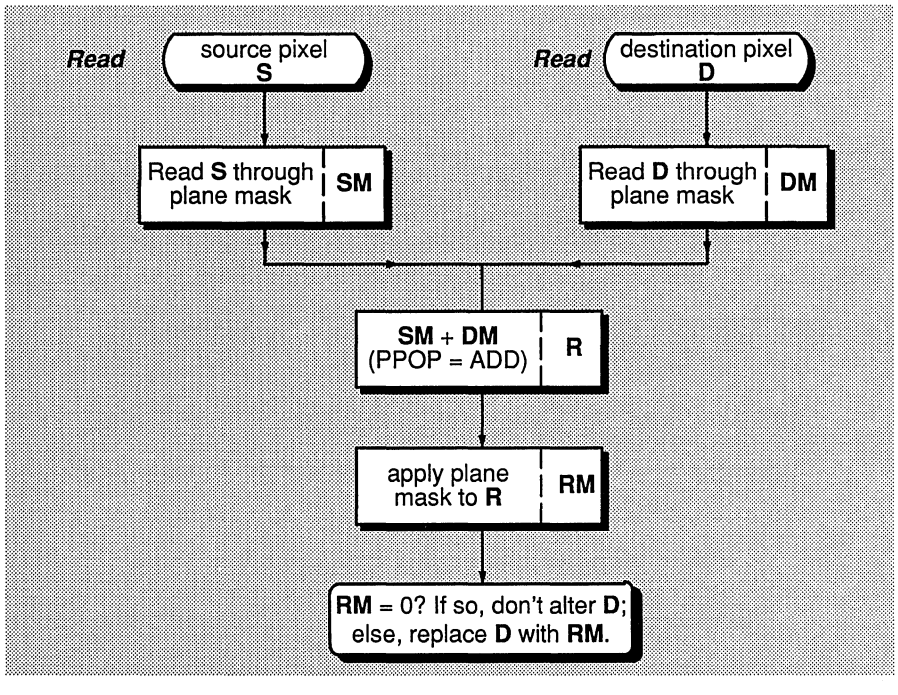


Note: For a complete illustration and description of CONTROL, refer to Chapter 4.

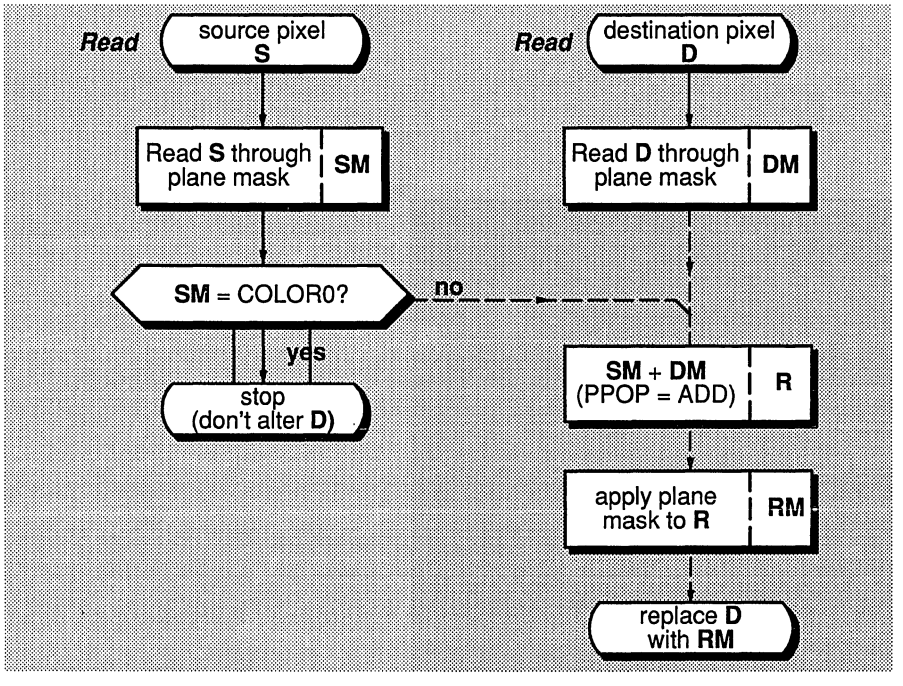
Transparency works for any graphics instruction that writes pixels; transparency does not affect nonpixel data writes.

Example 12–1 through Example 12–3 illustrate PIXT *Rs, *Rd instructions that use the three transparency modes. In these PIXT examples, both the source and destination are read, regardless of whether or not the source is transparent. Note that PIXBLTs operate similarly, but they read in a long word full of pixels and then test for transparency; if transparency-on-source=COLOR0 is enabled and all the pixels read are transparent, then no destination read (or write) occurs.

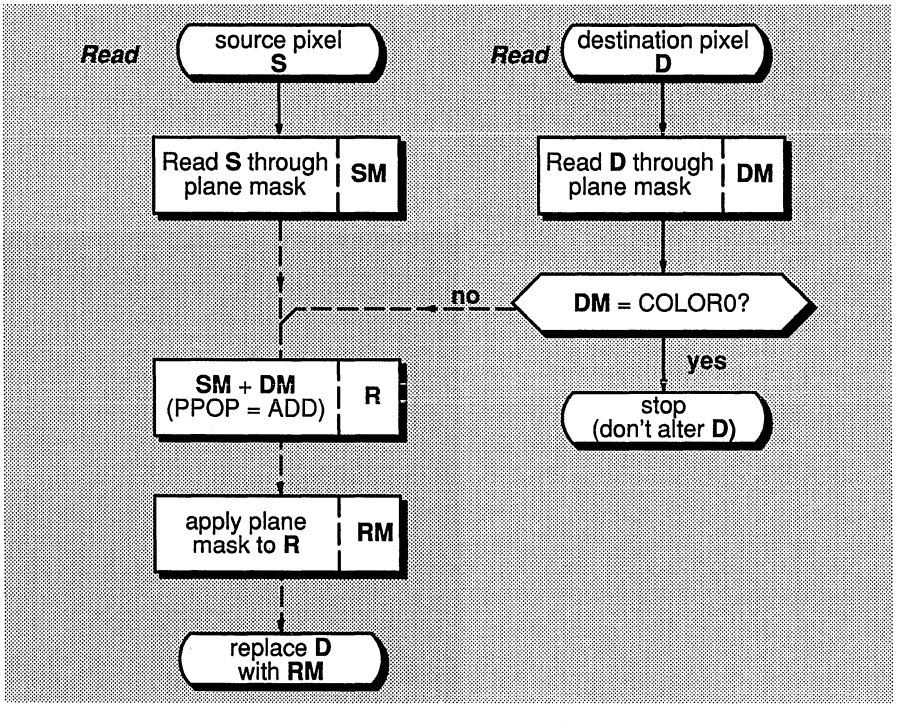
*Example 12–1. Transparency on Result = 0 for PIXT *Rs, *Rd*



*Example 12–2. Transparency on Source = COLOR0 for PIXT *Rs, *Rd*



*Example 12–3. Transparency on Destination = COLOR0 for PIXT *Rs, *Rd*



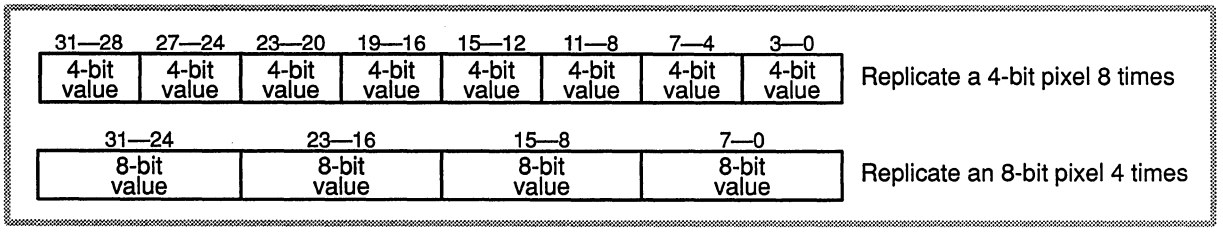
12.10 Plane Masking

The plane mask protects certain bits within pixels; graphics instructions do not modify mask-protected pixels.

The PMASK register contains the plane-mask value. Each bit in the plane mask corresponds to a bit position in a pixel. The 1s in the mask identify protected bits; the 0s identify modifiable bits. Pixel bits that are protected by the plane mask are always read as 0s during read cycles and are protected from alteration during write cycles. No status or control bit enables or disables plane masking, but you can effectively disable it by loading PMASK with all 0s (this is the default following reset).

The width of a quantity in the plane mask is the same as the pixel size. To maintain a consistent effect on all the pixels within a destination region, regardless of their position within the destination words, you should replicate the mask for a single pixel to fill the entire 32-bit PMASK register. For example, if the pixel size is 4 bits, replicate the 4-bit mask 8 times. These 8 copies of the mask are applied to the 8 pixels in a long word written to or read from memory. Similarly, replicate the value 16 times for a 2-bit pixel, twice for a 16-bit pixel, etc.

Figure 12–15. Replicating the Plane-Mask Value Through PMASK



You can use the RPIX instruction to replicate the plane-mask value in a general-purpose register and then move the value into PMASK.

The plane mask allows you to manipulate the bits within pixels as though the display memory were organized into bit planes (or color planes) that can be selectively protected from modification. The number of planes equals the number of bits per pixel. Consider an example in which the pixel size is 4 bits. The bits within each pixel are numbered 0—3 and belong to planes 0—3, respectively. All the bits numbered 0 in all the pixels form plane 0, all the bits numbered 1 in all the pixels form plane 1, and so on. The plane mask allows you to manipulate one or more planes independently of the others. If a display memory contains four planes, for example, you can dedicate three of the planes to 8-color graphics and use the fourth plane to overlay text in a single color. You can set the plane mask so that the text plane can be modified without affecting the graphics planes, and vice versa.

The plane mask affects only pixel accesses performed during execution of the PIXBLT, FILL, PIXT, DRAV, and LINE instructions. Data accesses by non-graphics instructions are not affected.

Instructions use the plane mask differently during reads and writes:

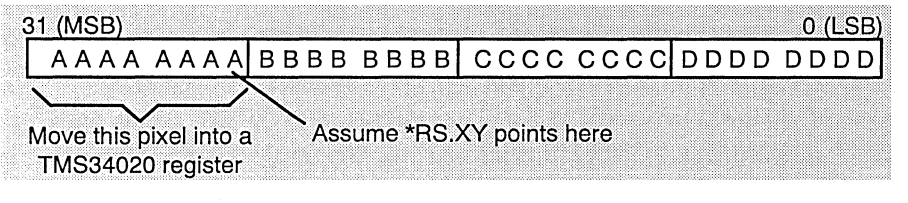
- **Pixel read.** The 0s in PMASK correspond to unprotected bits in the source pixel that are seen by the TMS34020 to contain the actual values read from memory. The 1s in PMASK correspond to protected bits in the source pixel that are seen as 0s by the TMS34020, regardless of the values read from memory.
- **Pixel write.** The 0s in PMASK specify those bits in the destination pixel in memory that may be altered. The 1s in PMASK specify protected bits in the destination pixel that cannot be altered.

When a pixel is transferred from a source to a destination location, plane masking is applied to the values read from the source and destination locations (before pixel processing). As the operands are read from memory, the bits protected by the plane mask are replaced with 0s before any pixel-processing operation is performed. Transparency detection can be applied to the masked data if the appropriate transparency mode is enabled. When the source and destination pixels have been combined to form the result, the plane mask is applied once more. If transparency-on-result=0 is enabled, it is applied to this result.

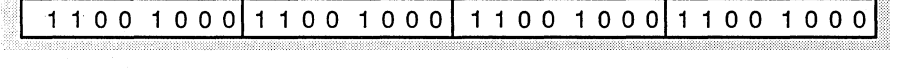
Source pixels that originate from registers are not affected by the plane mask and undergo pixel processing in unmodified form. Instructions that obtain their source pixels from registers include PIXT *Rs,*Rd* and PIXT *Rs,*Rd*.XY. Figure 12–16 shows how the TMS34020 applies the plane mask to pixel data during the read cycle of a PIXT **Rs*.XY, *Rd* instruction.

Figure 12–16. Read Cycle with Plane Masking, Transparency on Result = 0

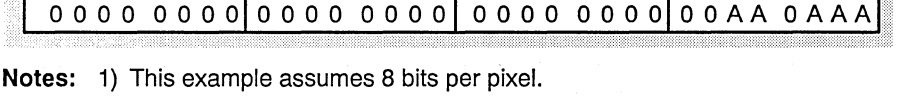
(a) Original data in memory (4 pixels)



(b) Plane mask (PMASK)



(c) Data read into TMS34020 register



Notes: 1) This example assumes 8 bits per pixel.

2) The pixel moved into the TMS34020 register is right-justified. All register bits to the left of the pixel are zero-filled.

(a) shows a 32-bit word that contains four 8-bit pixels.

(b) shows the addressed pixel ANDed with the inverse of the plane mask.

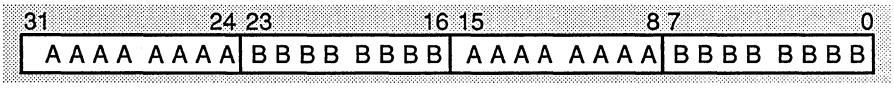
(c) contains the result, and shows that the bits within the data word that correspond to 1s in the mask are cleared to 0s.

After plane masking, the designated pixel is loaded into the 8 LSBs of the 32-bit destination register, and the 24 MSBs of the register are filled with 0s.

Figure 12–17 shows the transfer of 4 pixels during the course of a PIXBLT operation with transparency-on-result=0, pixel size = 8 bits, and the pixel-processing replace option. The inverse of PMASK is ANDed with the source data. Because the replace option is used, transparency is applied to the entire resulting pixel. In other words, the result controls the write in the manner described in the previous discussion of transparency. Since the 3 LSBs of the source pixel in bits 8—15 are 0s, and the rest of the pixel is masked off, the entire source pixel is interpreted as transparent. The memory interface logic generates an internal mask to govern which bits are modified during a write cycle. This mask contains 0s in the bits corresponding to the transparent pixel.

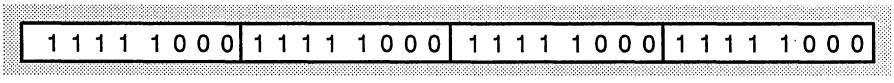
Figure 12–17. Write Cycle with Transparency on Result=0 and Plane Masking

(a) Original destination data in memory (4 pixels)

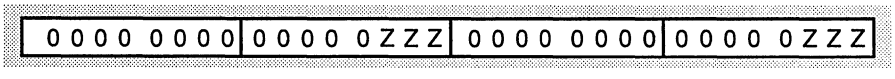


(b) Source data in memory (to be moved)

(c) Plane mask (PMASK register)



*(d) Mask source data (*SRC • $\overline{\text{PMASK}}$*)*



(e) Transparency mask based on the source data (see note 2)

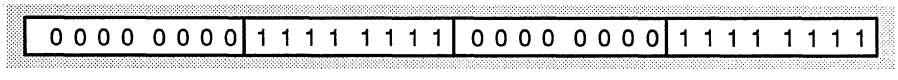
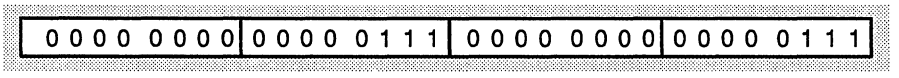
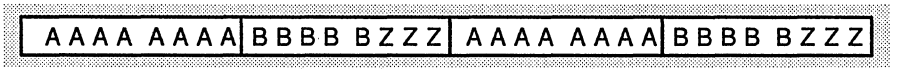


Figure 12–17. Write Cycle with Transparency on Result=0 and Plane Masking (Continued)

(f) Combined mask ($\overline{\text{PMASK}}$ • transparency mask*)*



(g) Resulting memory data after write cycle
*((*combined mask • source data) + ($\overline{\text{combined mask}}$ • destination data*))*



Notes: 1) This example assumes 8 bits per pixel.
2) Because this example uses the pixel-processing replace option, the source data is effectively the destination data.

(a) shows the original data at the destination location.

(b) shows the source data.

(c) shows the source data is ANDed with the inverse of the plane mask.

(d) shows the intermediate result produced by *(c)*. This result is used to generate the transparency mask in *(e)*, which is ANDed with the inverse of the plane mask in *(c)* to produce the composite mask shown in *(f)*.

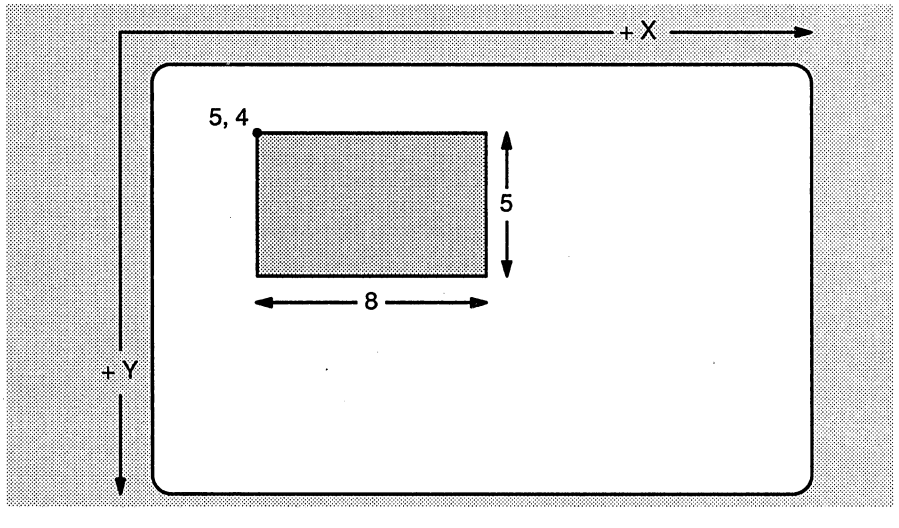
(g) shows the result, produced by replacing with the source only those bits of the destination corresponding to 1s in the composite mask in *(f)*.

12.11 Setting Up the Implied Operands for Graphics Instructions

The TMS34020 graphics instructions use the B-file registers and several of the I/O registers as **implied operands**. Just as the TMS34020 obtains information from regular instruction operands, the TMS34020 obtains information from these registers that it needs in order to properly execute the instruction. For example, when you use the ADD instruction, the TMS34020 expects you to supply some information: the values you would like to add. For the ADD instruction, you supply this information as operands to ADD in the same source statement. Some graphics instructions, such as the PIXT instructions, use operands in this same manner. However, they also use implied operands. Some of the information that the TMS34020 needs is not supplied in the source statement with the instruction; the TMS34020 expects to find this information in the appropriate implied-operand registers. You must set up these implied operands *before* you can use a graphics instruction.

The code segment in Example 12–4 shows how you might set up the implied operands for a FILL XY instruction. Figure 12–18 illustrates the area and dimensions of the fill.

Figure 12–18. Filled Area for Example 12–4



Note:

Chapter 4 contains complete descriptions of all the B-file and I/O registers.

Example 12–4. Setting Up Implied Operands for a FILL Instruction

```
DADDR    .set  B2        ; Equate the registers names
DPTCH    .set  B3        ; with their locations in the
OFFSET   .set  B4        ; register file or with their
WSTART   .set  B5        ; addresses in the I/O memory
WEND     .set  B6        ; map. This allows you to call
DYDX     .set  B7        ; them by name later.
COLOR1   .set  B9
CONTROL  .set  0C0000190h
CONVDP   .set  0C0000140h
PMASK    .set  0C0000160h
PSIZE    .set  0C0000150h
         .
         .
         .
*  Set up the CONTROL register to choose the pixel *
*  processing replace option, enable transparency, *
*  select transparency-on-destination=COLOR0, and  *
*  select window-checking mode 2. Don't modify     *
*  any other bits                                  *
         MOVE  @CONTROL, A1     ; assume FS0 = 32 bits
         ORI   000083E5h, A1
         MOVE  A1, @CONTROL, 1  ; assume FS1 = 16 bits
         MOVI  099999999h, COLOR1
         MOVI  4*1024, DPTCH
         SETCDP                 ; initialize CONVDP
         CLR   OFFSET
         MOVI  [0,0], WSTART
         MOVI  [479,639], WEND
         MOVI  [5,8], DYDX
         CLR   A1             ; disable PMASK
         MOVE  A1, @PMASK
         MOVK  4, A1            ; 4-bit pixels
         MOVE  A1, @PSIZE
         MOVI  [4,5], DADDR
         FILL  XY               ; execute the instruction
```

Note: The purpose of this illustration is clarity, not efficiency.

Some of these registers may contain values that you'll use for more than one graphics instruction; if this is the case, it isn't necessary to explicitly load the register each time you use a graphics instruction. For example, if you always use the same pixel size, you need to load the PSIZE register only once. Some instructions, however, use the contents of certain registers as scratchpads and alter the register contents. In this case, you cannot assume that the register will contain the value that you originally supplied.

No single graphics instruction uses *all* of the implied operands. Table 12–8 identifies which implied operands each graphics instruction uses. Table 12–8 *(a)* lists the implied operands that are B-file registers; Table 12–8 *(b)* lists the implied operands that are I/O registers. A √ symbol indicates that an instruction uses the register as an implied operand.

Table 12–8. Summary of Implied Operands Used by the Graphics Instructions

(a) B-file registers

	SADDR (B0)	SPTCH (B1)	DADDR (B2)	DPTCH (B3)	OFFSET (B4)	WSTART (B5)	WEND (B6)	DYDX (B7)	COLOR0 (B8)	COLOR1 (B9)	MADDR (B10)	COUNT (B10)	MPTCH (B11)	INC1 (B11)	INC2 (B12)	PATTERN (B13)	TEMP (B14)
BLMOVE	√		√					√									
CLIP			√			√	√	√									
CVDXYL				√													
CVMXYL													√				
CVSXYL		√															
CVXYL				√	√												
DRAV				√	√	√	√			√							
FILL L			√	√				√									
FILL XY			√	√	√	√	√	√									
FLINE	√		√	√				√	√	√		√		√	√	√	
FPIXEQ									√		√		√				
FPIXNE									√		√		√				
LINE	√		√	√	√	√	√	√	√			√		√	√	√	
PFILL			√	√	√			√	√	√						√	
PIXBLT B, L	√	√	√	√	√			√	√	√							
PIXBLT B, XY	√	√	√	√	√	√	√	√	√	√							
PIXBLT L, L	√	√	√	√	√			√									
PIXBLT L, XY	√	√	√	√	√	√	√	√									
PIXBLT XY, L	√	√	√	√	√			√									
PIXBLT XY, XY	√	√	√	√	√	√	√	√									
PIXBLT L, M, L	√	√	√	√	√			√			√		√				
PIXT *Rs*, **Rd*.**XY**				√	√	√	√										
PIXT **Rs*.**XY**, *Rd*		√			√												
PIXT **Rs*.**XY**, **Rd*.**XY**		√		√	√												
SETCDP				√													
SETCMP													√				
SETCSP		√															
TFILL	√	√	√	√	√	√	√	√		√	√		√				
VBLT	√	√	√	√				√									
VFILL		√	√	√				√									
VLCOL										√							

Table 12–8. Summary of Implied Operands Used by the Graphics Instructions (continued)

(b) I/O registers

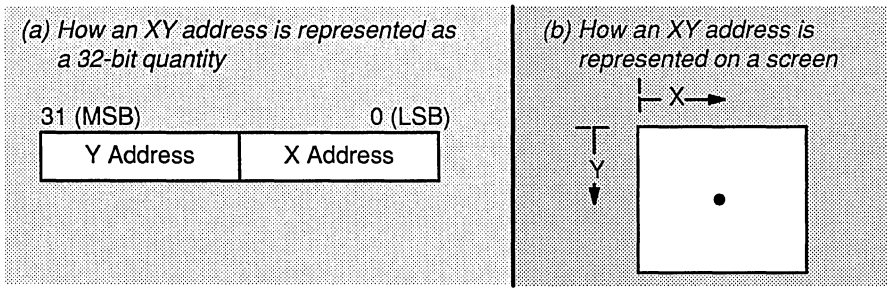
	CONTROL											
	PPOP field	T bit	TM field	W field	PBH bit	PBV bit	CONFIG (VEN bit)	CONVDP	CONVMP	CONVSP	PMASK	PSIZE
CVDXYL								√				√
CVMXYL									√			√
CVSXYL										√		√
CVXYL								√				√
DRAV	√	√	√	√				√			√	√
FILL L	√	√	√								√	√
FILL XY	√	√	√	√				√			√	√
FLINE	√	√	√					√			√	√
FPIXEQ											√	√
FPIXNE											√	√
LINE								√			√	√
PIXBLT B, L	√	√	√								√	√
PIXBLT B, XY	√	√	√	√				√			√	√
PIXBLT L, L	√	√	√		√	√					√	√
PIXBLT L, XY	√	√	√	√	√	√		√		√	√	√
PIXBLT XY, L	√	√	√		√	√		√		√	√	√
PIXBLT XY, XY	√	√	√	√	√	√		√		√	√	√
PIXBLT L, M, L	√	√	√		√	√					√	√
PIXT *Rs*, **Rd*	√	√	√								√	√
PIXT **Rs*, **Rd*	√	√	√									√
PIXT *Rs*, **Rd*.**XY**	√	√	√	√				√		√	√	√
PIXT **Rs*.**XY**, *Rd*												√
PIXT **Rs*.**XY**, **Rd*.**XY**								√		√		√
SETCDP								√				
SETCMP									√			
SETCSP										√		
TFILL	√	√	√	√				√		√	√	√
VBLT							√				√	√
VFILL							√			√	√	√

12.12 Converting an XY Address to a Linear Address

The TMS34020 allows you to access a pixel by using its linear address or by using an XY address. It's usually easier and more natural to use XY addressing because screens are typically configured on an XY grid.

Figure 12–19 *(a)* shows how an XY address is represented as a 32-bit quantity. The 16 MSBs form the Y coordinate and the16 LSBs form the X coordinate. Both coordinates are signed. Figure 12–19 *(b)* illustrates the relationship between an XY address and the position of a pixel in an array.

Figure 12–19. How an XY Address Is Represented



12.12.1 Manual XY-to-Linear Conversion

Although the TMS34020 allows you to specify XY addresses, it usually converts them to linear addresses before it uses them. When it does this, it uses the following equation.

$$\textit{linear bit address} = (Y \times \textit{array pitch}) + (X \times \textit{pixel size}) + \textit{offset}$$

Note:

All legal pixel sizes (1, 2, 4, 8, 16, and 32) are powers of 2. Thus, the $X \times$ *pixel size* portion of the XY-to-linear conversion equation can be reduced to a shift.

In some cases, you may find it necessary to manually convert an XY address to a linear address. You can convert an XY address to a linear address by using one of these instructions.

- CVSXYL converts the address using the conversion value in CONVSP.
- CVDXYL converts the address using the conversion value in CONVDP.
- CVMXYL converts the address using the conversion value in CONVMP.

In order to use any of these instructions, you must

- Load the XY address into a general-purpose register, and
- Load a conversion value into the appropriate CONVxP register; this conversion value is based on an array pitch. There are three CONVxP (conversion) registers; each is associated with a pitch register. Table 12–9 lists the conversion registers and associated pitch registers.

Table 12–9. TMS34020 Conversion (CONVxP) Registers

Array	Conversion Register		Pitch Register	
source	CONVSP	(C000 0130h)	SPTCH	(B1)
destination	CONVDP	(C000 0140h)	DPTCH	(B3)
mask	CONVMP	(C000 0180h)	MPTCH	(B11)

The TMS34020 supports three instructions for loading the correct values into the CONVxP registers.

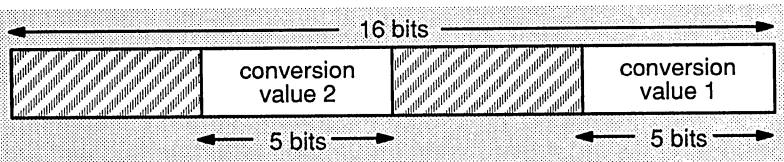
- SETCSP loads CONVSP according to the value in SPTCH.
- SETCDP loads CONVDP according to the value in DPTCH.
- SETCMP loads CONVMP according to the value in MPTCH.

The TMS34020 supports three categories of array pitch, allowing you to choose between conversion speed and flexibility in defining pitch values. These categories include

- **Power of 2.** When the array pitch is a power of 2, then the *Y× array size* operation can be performed as a shift. Reducing both multiplications to shifts decreases the linear-address conversion referred to by the CVxXYL instruction to 3 cycles. **This is the most efficient calculation.**
- **Two powers of 2.** If the pitch can be expressed as the sum of two powers of 2 (for example, 1280 = 1024 + 256), then the *Y× array pitch* operation can be calculated by summing two shifts of the Y value. The CVxXYL conversion time for this method is 4 cycles.
- **Arbitrary pitch.** A pitch can be any value. If the pitch is not a power of 2 or cannot be reduced to a sum of powers of 2, then the *Y× array pitch* operation must be calculated with a full 16-bit-by-32-bit multiplication. The CVxXYL conversion time for an arbitrary pitch consumes approximately 15 cycles.

Figure 12–20 shows how the TMS34020 determines the contents of a CONVxP register from the appropriate pitch register.

Figure 12–20. How Values Are Contained in a CONVxP Register



- When conversion value 1 ≠ 0 and conversion value 2 = 0, the pitch is a power of 2 (1 shift)
- When conversion value 1 ≠ 0 and conversion value 2 ≠ 0, the pitch is a sum of 2 powers of 2 (2 shifts)
- When conversion value 1 = 0, the pitch is an arbitrary value (multiplication)

Note:

Be careful to set up the CONVxP registers correctly; *conversion value 1* = 0 **always** causes the TMS34020 to perform a multiplication. This can be very time consuming.

12.12.2 The CONVxP Registers, Corner Adjusting, and Preclipping

If either a source or destination array is specified in XY format, then the contents of the CONVSP and CONVDP registers are used in instances in which the Y component of the starting address must be adjusted prior to the start of the PIXBLT. The Y component may require adjustment, either to preclip the array or to select a starting pixel in one of the lower two corners of the array. For this reason, the CONVSP and CONVDP registers are sometimes listed as implied operands for a PIXBLT instruction.

Chapter 13

TMS34020 Assembly Language Instruction Set

This section describes the TMS34020 assembly language instruction set (in alphabetical order). Related subjects, such as addressing modes, are presented first.

You'll find these topics on the following pages:

13.1 Addressing Modes and Operand Formats

The TMS34020 instruction set supports eight addressing modes. Most instructions have register-direct operands or a combination of register-direct and immediate operands; however, the move and graphics instructions use more complex combinations of operands. This section discusses the TMS34020 addressing modes and defines the symbols used in instruction syntax to indicate an addressing mode.

13.1.1 Immediate Values and Constants

An instruction syntax may use one of these symbols to indicate an immediate *source* operand:

IW is a 16-bit (short) signed immediate value.
IL is a 32-bit (long) signed immediate value.
K is a 5-bit constant.

Instructions that have immediate source operands have register-direct destination operands. Many instructions that have an immediate value can use either a short or a long value.

Figure 13–1 illustrates a MOVI (move immediate) instruction whose first operand is a 32-bit immediate value. This is the syntax for this MOVI:

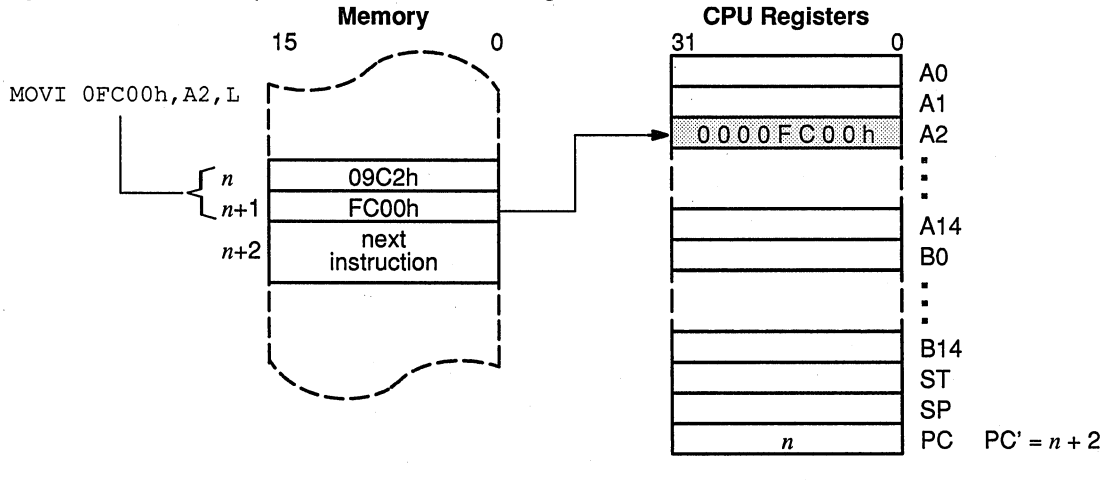
MOVI *IL, Rd [,* **L***]*

The instruction in Figure 13–1 is as follows:

```
MOVI  0FC0h, A2, L
```

Figure 13–1 shows the object code in memory (at word *n*) and the effect of the instruction on the CPU registers. The value 0FC0h is copied into register A2 as a zero-extended 32-bit value. (Note that this is a 2-word instruction; the next instruction to be executed is at words *n* + 2.)

Figure 13–1. An Example of Immediate Addressing



13.1.2 Absolute Addresses

An instruction syntax may use one of these symbols to indicate an absolute operand:

@SAddress	is a **source** address that contains the source data.
@DAddress	is a **destination** address.

Note that the @ character is entered as part of the operand (this distinguishes it from an immediate operand).

Figure 13–2 illustrates a MOVB (move byte) instruction that has an absolute operand (the first parameter is a 32-bit source address). This is the syntax for this MOVB:

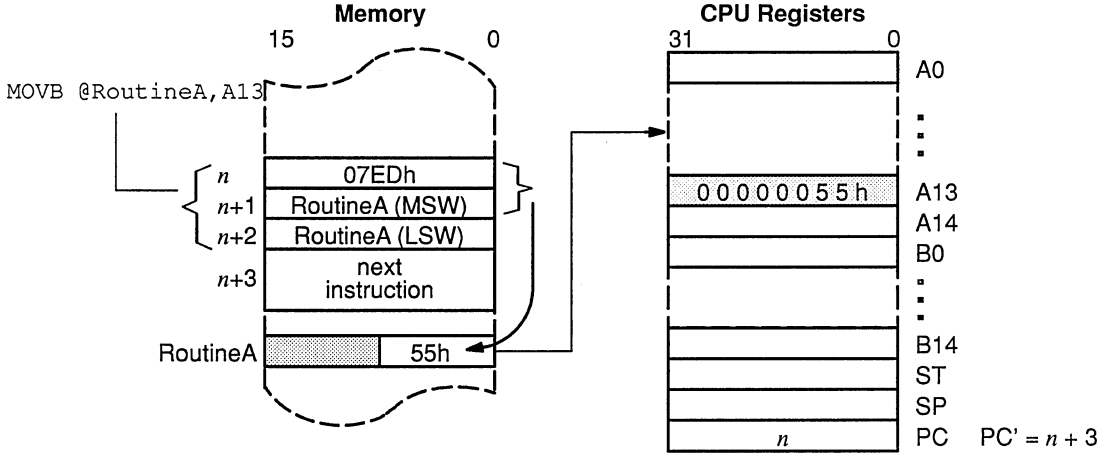
MOVB *@SAddress, Rd*

The instruction in Figure 13–2 is as follows:

```
MOVB  @RoutineA, A13
```

Figure 13–2 shows the object code in memory (at word *n*) and the effect of the instruction on the CPU registers. `@RoutineA` is the address of a byte; this MOVB instruction copies the byte at address `RoutineA` into register A13. (Note that this is a 3-word instruction; the next instruction to be executed is at word *n* + 3.)

Figure 13–2. An Example of Absolute Addressing



13.1.3 Register-Direct Operands

An instruction syntax may use one of these symbols to indicate a register-direct operand:

Rs is a **source** register that contains the source data.
Rd is a **destination** register that will contain the result.

When both operands of an instruction are register-direct operands, the registers *must be in the same file.* (The MOVE *Rs,Rd* instruction is an exception to this rule.)

Figure 13–3 illustrates a MOVE instruction that has two register-direct operands. This is the syntax for this MOVE:

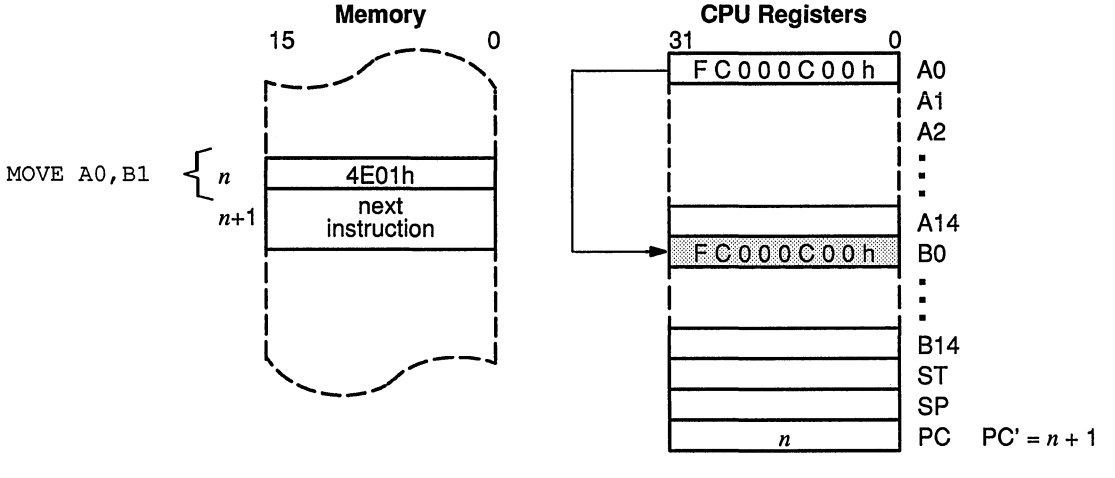
MOVE *Rs, Rd*

The example shows this instruction:

```
MOVE  A0, B1
```

Figure 13–3 shows the object code in memory (at word n) and the effect of the instruction on the CPU registers. The entire contents of register A0 are copied into register B1. (Note that this is a 1-word instruction; the next instruction to be executed is at word $n + 1$.)

Figure 13–3. An Example of Register-Direct Addressing



13.1.4 Register-Indirect Operands

An instruction syntax may use one of these symbols to indicate a register-indirect operand:

Rs* is a register that contains the address of the **source data.
Rd* is a register that contains the **destination address.

Note that the * character is entered as part of the operand (this distinguishes it from a register-direct operand).

Figure 13–4 illustrates a MOVE (move field) instruction that has two register-indirect operands. This is the syntax for this MOVE:

MOVE **Rs, *Rd [, F]*

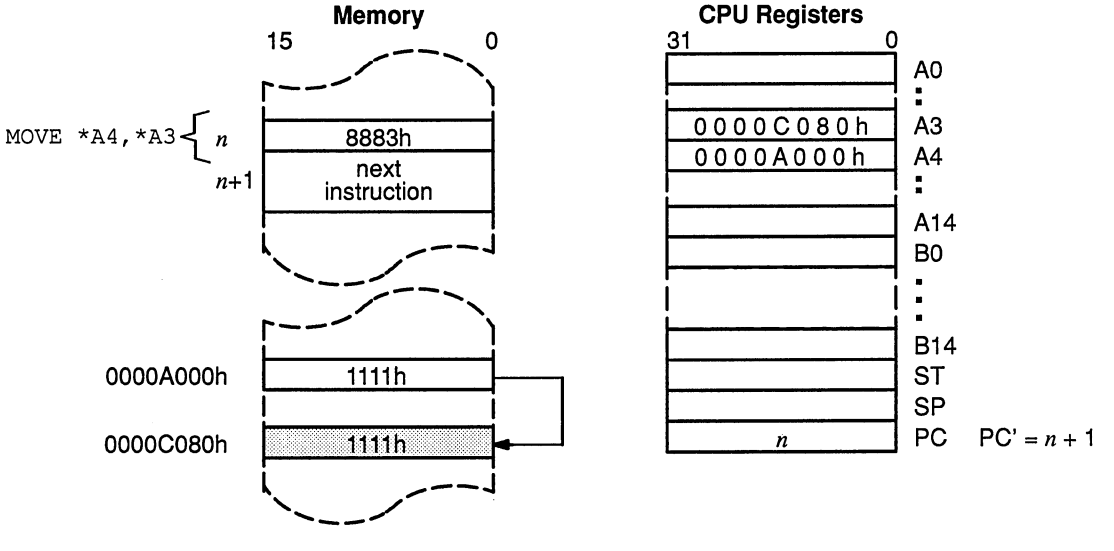
For more information about field moves, refer to Section 13.3.6 on page 13.3.6.

The example shows this instruction:

```
MOVE  *A4, *A3
```

Figure 13–4 shows the object code in memory (at word *n*) and the effect of the instruction on the destination address. The contents of register A4 specify the address of data to be moved; the contents of register A3 specify the destination address. Assume that the field size for the move is 16 bits; the 16 bits of data at *A4 are moved to the location at *A3. (Note that this is a 1-word instruction: the next instruction to be executed is at word *n* + 1.)

Figure 13–4. An Example of Register-Indirect Addressing



13.1.5 Register-Indirect with Offset

An instruction syntax may use one of these symbols to indicate a register-indirect operand that uses a signed offset:

Rs(Offset)* is a **source address formed by adding an offset to the contents of the source register.

Rd(Offset)* is a **destination address formed by adding an offset to the contents of the destination register.

The offset is used only to form an address — the contents of the register are not affected. Note that the * character is entered as part of the operand.

Figure 13–5 illustrates a MOVE (move field) instruction; the first operand of this instruction is a register-direct operand; the second operand is a register-indirect operand with an offset. This is the syntax for this MOVE:

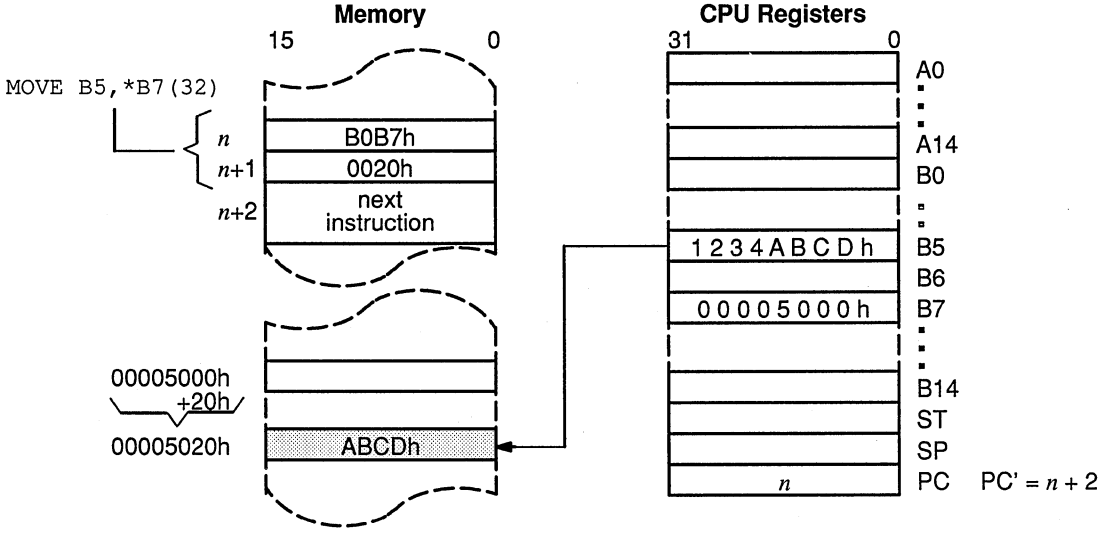
MOVE *Rs, *Rd(offset) [, F]*

The example shows this instruction:

```
MOVE   B5, *B7(32)
```

Figure 13–5 shows the object code in memory (at word *n*) and the effect of the instruction on the destination location. The destination address is specified by adding the offset (32 bits) to the contents of register B7; this yields a destination location of 05020h. Assume that the field size for the move is 16 bits; the 16 LSBs in register B5 are copied into the destination location. (Note that this is a 2-word instruction; the next instruction is at word *n* + 2.)

Figure 13–5. An Example of Register-Indirect with Offset Addressing



13.1.6 Register-Indirect with Postincrement

An instruction syntax may use one of these symbols to indicate a register-indirect operand that is postincremented:

Rs+* is a register that contains the address of the **source data.
Rd+* is a register that contains the **destination address.

Note that the * and + characters are entered as part of the operand. After the operation is performed, the contents of the specified source or destination register are incremented by the field size used for the operation.

Figure 13–6 illustrates a MOVE (move field) instruction; both the source and the destination operands are postincremented register-indirect operands. This is the syntax for this MOVE:

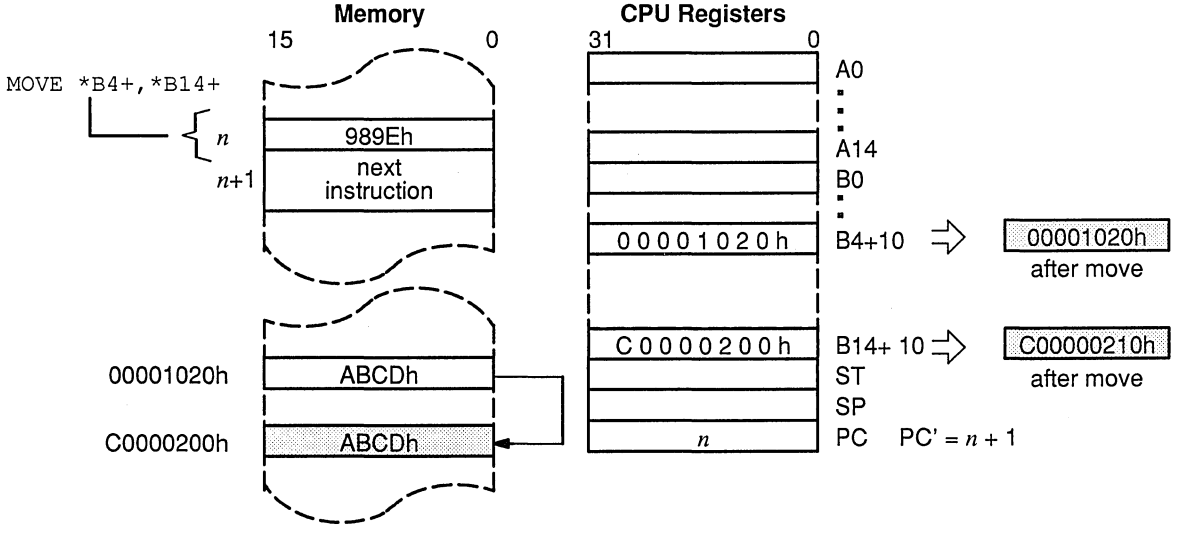
MOVE **Rs+, *Rd+ [, F]*

The example shows this instruction:

```
MOVE  *B4+, *B14+
```

Figure 13–6 shows the object code in memory (at word n) and the effect of the instruction on the destination location and the CPU registers. The contents of register B4 are the address of the source data; the contents of register B14 specify the destination address. Assume that the field size for the move is 16 bits; the 16 bits of data at the source address are copied into the destination location. After the move, both registers are incremented by 16 bits (1 word). (Note that this is a 1-word instruction; the next instruction to be executed is at word n + 1.)

Figure 13–6. An Example of Register-Indirect with Postincrement Addressing



13.1.7 Register-Indirect with Predecrement

An instruction syntax may use one of these symbols to indicate a register-indirect operand that is predecremented.

Before the operation is performed, the contents of the specified source or destination register are decremented by the field size used for the operation.

–Rs the decremented register contents are the address of the **source** data.

–Rd the decremented register contents specify the **destination** address.

Note that the * and – characters are entered as part of the operand.

Figure 13–7 illustrates a MOVE (move field) instruction; the source operand is a register-direct operand, and the destination operand is a predecremented register-indirect operand. This the syntax for this MOVE:

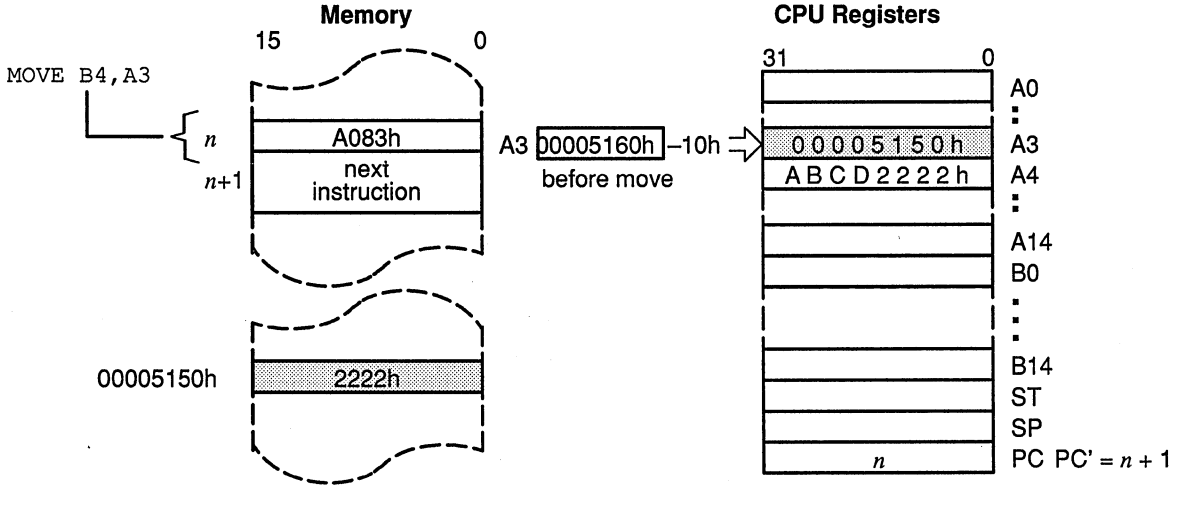
MOVE *Rs, *–Rd [, F]*

The example shows this instruction:

```
MOVE  A4, *-A3
```

Figure 13–7 shows the object code in memory (at word *n*) and the effect of the instruction on the destination location and the CPU registers. Assume that the field size for the move is 16 bits. Register A4 contains the source data. The contents of register A3, *minus the field size* (16 bits, or 1 word) form the destination address 5150h. The 16 LSBs in A4 are copied to address 5150h. (Note that this is a 1-word instruction; the next instruction to be executed is at word *n* +1.)

Figure 13–7. An Example of Register-Indirect with Predecrement Addressing



13.1.8 Register-Indirect in XY Mode

An instruction syntax may use one of these symbols to indicate that a register operand contains an XY address.

*Rs.**XY** is a register that contains the XY address of the **source** data.
*Rd.**XY** is a register that contains the XY **destination** address.

Note that the * and **.XY** characters are entered as part of the operand. Here's an example that uses an indirect-XY destination operand:

```
PIXT  A0, *A6.XY
```

This instruction moves the pixel data at the least significant end of register A0 into the XY address specified by the contents of register A6.

13.2 Summary Table

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
ABS *Rd* Store absolute value	1	1	0000 0011 100R DDDD
ADD *Rs, Rd* Add registers	1	1	0100 000S SSSR DDDD
ADDC *Rs, Rd* Add registers with carry	1	1	0100 001S SSSR DDDD
ADDI *IW, Rd* Add immediate (16 bits)	2	2	0000 1011 000R DDDD
ADDI *IL, Rd* Add immediate (32 bits)	3	2,3†	0000 1011 001R DDDD
ADDK *K, Rd* Add constant (5 bits)	1	1	0001 00KK KKKR DDDD
ADDXY *Rs, Rd* Add registers in XY mode	1	1	1110 000S SSSR DDDD
ADDXYI *IL, Rd* Add immediate in XY mode	3	2,3†	0000 1100 000R DDDD
AND *Rs, Rd* AND registers	1	1	0101 000S SSSR DDDD
ANDI *IL, Rd* AND immediate (32 bits)	3	2,3†	0000 1011 100R DDDD

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
ANDN *Rs, Rd* AND register with complement (32 bits)	1	1	0101 001S SSSR DDDD
ANDNI *IL, Rd* AND not immediate (32 bits)	3	2, 3 †	0000 1011 100R DDDD
BLMOVE *S, D* Block move	1	complex instruction	0000 0000 1111 00SD
BTST *K, Rd* Test register bit, constant	1	1	0001 11KK KKKR DDDD
BTST *Rs, Rd* Test register bit, register	1	2	0100 101S SSSR DDDD
CALL *Rs* Call subroutine indirect	1	3+(1) 3+(4) ‡	0000 1001 001R DDDD
CALLA *Address* Call subroutine address	3	see instruction	0000 1101 0101 1111
CALLR *Address* Call subroutine relative	2	3+(1) 3+(4) ‡	0000 1101 0011 1111
CEXEC *size, instruction [, ID]* Coprocessor internal operation execution, long	3	2 (1) 3 (1) †	0000 0110 0000 0000
CEXEC *size, instruction [, ID]* Coprocessor internal operation execution, short	2	2(1)	1101 1000 0CCC CCCS
CLIP Clip an array to fit within a window	1	complex instruction	0000 1000 1111 0010
CLR *Rd* Clear register	1	1	0101 011R DDDR DDDD
CLRC Clear carry	1	1	0000 0011 0010 0000
CMOVCG Rd_1 *[,* Rd_2 *[size]], command [, ID]* Move coprocessor to register	3	¶	0000 0110 011R DDDD
CMOVCM **Rd+, transfers, size, command [, ID]* Coprocessor to memory indirect (postincrement)	3	5+[cnt–1] 6+[cnt+1]†	0000 0110 101R DDDD
CMOVCM *–*Rd, transfers, size, command [, ID]* Coprocessor to memory indirect (predecrement)	3	5+[cnt–1] 6+[cnt+1]†	0000 0110 110R DDDD
CMOVCS *command [, ID]* Move from coprocessor to status register	3	4, 5 †	0000 0110 0110 0000
CMOVGC *Rs, command [, ID]* One register to coprocessor	3	2(1) 3(1) †	0000 0110 001R SSSS
CMOVGC Rs_1, Rs_2, *size, command [, ID]* Two registers to coprocessor	3	3(1) 4(1) †	0000 0110 010R SSSS

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
CMOVMC **Rs+, transfers, size, command [, ID]* Memory indirect (postincrement) to coprocessor, constant count	3	5+[cnt–1] 6+[cnt+1]†	0 0 0 0 0 1 1 0 1 0 0 T T T T
CMOVMC *–*Rs, transfers, size, command [, ID]* Memory indirect (predec) to coprocessor, constant count	3	5+[cnt–1] 6+[cnt+1]†	0 0 0 0 1 0 0 0 0 0 1 T T T T
CMOVMC **Rs+, Rd, size, command [, ID]* Memory indirect (postinc) to coprocessor, register count	3	¶	0 0 0 0 0 1 1 0 1 1 1 R S S S S
CMP *Rs, Rd* Compare registers	1	1	0 1 0 0 1 0 0 S S S R D D D D
CMPI *IW, Rd* Compare immediate (16 bits)	2	2	0 0 0 0 1 0 1 1 0 1 0 R D D D D
CMPI *IL, Rd* Compare immediate (32 bits)	3	2, 3 †	0 0 0 0 1 0 1 1 0 1 1 R D D D D
CMPK Compare to set status bits	1	1	0 0 1 1 0 1 K K K K K R D D D D
CMPXY *Rs, Rd* Compare X and Y half of registers	1	1	1 1 1 0 0 1 0 S S S S R D D D D
CPW *Rs, Rd* Compare point to window	1	1	1 1 1 0 0 1 1 S S S S R D D D D
CVDXYL *Rd* Convert destination XY address to linear	1	¶	0 0 0 0 1 0 1 0 1 0 0 R D D D D
CVMXYL *Rd* Convert mask address to linear	1	¶	0 0 0 0 1 0 1 0 0 1 1 R D D D D
CVSXYL *Rs, Rd* Convert source XY address to linear	1	¶	1 1 1 0 1 0 1 S S S S R D D D D
CVXYL *Rs, Rd* Convert XY address to linear	1	¶	1 1 1 0 1 0 0 S S S S R D D D D
DEC *Rd* Decrement register	1	1	0 0 0 1 0 1 0 0 0 0 1 R D D D D
DINT Disable interrupts	1	3	0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0
DIVS *Rs, Rd* Divide registers signed	1	¶	0 1 0 1 1 0 0 S S S S R D D D D
DIVU *Rs, Rd* Divide registers unsigned	1	¶	0 1 0 1 1 0 1 S S S S R D D D D

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
DRAV *Rs, Rd* Draw and advance	1	¶	1111 011S SSSR DDDD
DSJ *Rd, Address* Decrement register and skip jump	2	2 no jump 3 if jump	0000 1101 100R DDDD
DSJEQ *Rd, Address* Conditionally decrement register and skip jump	2	2 no jump 3 if jump	0000 1101 101R DDDD
DSJNE *Rd, Address* Conditianally decrement register and skip jump	2	2 no jump 3 if jump	0000 1101 110R DDDD
DSJS *Rd, Address* Decrement register and skip jump, short	1	2 no jump 3 if jump	0011 1 D offset R DDDD
EINT Enable interrupts	1	3	0000 1101 0110 0000
EMU Initiate emulation	1	¶	0000 0001 000 count
EXGF *Rd, F* Exchange field size	1	F0 1 F1 2	1101 01F1 000R DDDD
EXGPC *Rd* Exchange program counter with register	2	2	0000 0001 001R DDDD
EXGPS *Rd* Exchange pixel size	1	2(1)	0000 0010 101R DDDD
FILL L Fill array with processed pixels, linear	1	complex instruction	0000 1111 1100 0000
FILL XY Fill array with processed pixels, XY	1	complex instruction	0000 1111 1110 0000
FLINE {0 | 1} Fast line draw with linear addressing	1	¶	1101 1110 Z001 1010
FPIXEQ Find pixel equal to COLOR0	1	complex instruction	0000 1010 1011 1011
FPIXNE Find pixel not equal to COLOR0	1	complex instruction	0000 1010 1101 1011
GETPC *Rd* Get program counter into register	1	1	0000 0001 010R DDDD
GETPS *Rd* Get pixel size into register	1	2	0000 0010 110R DDDD
GETST *Rd* Get status register into register	1	1	0000 0001 100R DDDD

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
IDLE Wait for interrupt	1	¶	0000 0000 0100 0000
INC Increment register	1	1	0001 0000 001R DDDD
JA*cc Address* Jump absolute conditional	3	3 no jump 4 if jump	1100 code 1000 0000
JR*cc Address* Jump relative conditional, long	2	1 no jump 2 if jump	1100 code 0000 0000
JR*cc Address* Jump relative conditional, short	1	2 no jump 3 if jump	1100 code xxxx xxxx
JUMP *Rs* Jump indirect	1	2	0000 0001 011R DDDD
LINE {0 | 1} Line draw	1	¶	1101 1111 Z001 1010
LINIT Line intialization	1	9	0000 1100 0101 0111
LMO *Rs, Rd* Leftmost one	1	1	0110 101S SSSR DDDD
MMFM *Rs, [, List]* Move multiple registers from memory	2	¶	0000 1001 101R DDDD
MMTM *Rs, [, List]* Move multiple registers to memory	2	¶	0000 1001 100R DDDD
MODS *Rs, Rd* Modulus signed	1	¶	0110 110S SSSR DDDD
MODU *Rs, Rd* Modulus unsigned	1	35 3 if Rs = 0	0110 111S SSSR DDDD
MOVB *Rs, *Rd* Move byte, register to indirect	1	§	1000 110S SSSR DDDD
MOVB **Rs, Rd* Move byte, indirect to register	1	§	1000 111S SSSR DDDD
MOVB **Rs(offset), Rd* Move byte, indirect with offset to register	2	§	1010 111S SSSR DDDD
MOVB **Rs(SOffset), *Rd(DOffset)* Move byte, indirect with offset to indirect with offset	3	§	1011 110S SSSR DDDD
MOVB *Rs, @DAddress* Move byte, register to absolute	3	§	0000 0101 111R SSSS

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
MOVB *@SAddress, Rd* Move byte, absolute to register	3	§	0000 0111 111R DDDD
MOVB *@SAddress, @DAddress* Move byte, absolute to absolute	5	§	0000 0011 0100 0000
MOVE *Rs, Rd* Move register to register	1	1	0100 11MS SSSR DDDD
MOVE *Rs, *Rd [, F]* Move field, register to indirect	1	§	1000 00FS SSSR DDDD
MOVE *Rs, –*Rd [, F]* Move field, register to indirect (predecrement)	1	§	1010 00FS SSSR DDDD
MOVE *Rs, *Rd+ [, F]* Move field, register to indirect (postincrement)	1	§	1001 00FS SSSR DDDD
MOVE **Rs, Rd [, F]* Move field, indirect to register	1	§	1000 01FS SSSR DDDD
MOVE *–*Rs, Rd [, F]* Move field, indirect (predecrement) to register	1	§	1010 01FS SSSR DDDD
MOVE **Rs+, Rd [, F]* Move field, indirect (postincrement) to register	1	§	1001 01FS SSSR DDDD
MOVE **Rs, *Rd [, F]* Move field, indirect to indirect	1	§	1000 10FS SSSR DDDD
MOVE *–*Rs,–*Rd [, F]* Move field, indirect (predecrement)to indirect (predecrement)	1	§	1010 10FS SSSR DDDD
MOVE **Rs+, *Rd+* Move field, indirect (postincrement) to indirect (postincrement)	1	§	1001 10FS SSSR DDDD
MOVE *Rs, *Rd(offset) [, F]* Move field, register to indirect with offset	2	§	1011 00FS SSSR DDDD
MOVE **Rs(offset), Rd [, F]* Move field, indirect with offset to register	2	§	1011 01FS SSSR DDDD
MOVE **Rs(offset), *Rd+ [, F]* Move field, indirect with offset to indirect (postincrement)	2	§	1101 00FS SSSR DDDD
MOVE **Rs(SOffset), *Rd(DOffset) [, F]* Move field, indirect with offset to indirect with offset	3	§	1011 10FS SSSR DDDD
MOVE *Rs, @DAddress [, F]* Move field, register to absolute	3	§	0000 01F1 100R SSSS

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
MOVE *@SAddress, Rd [, F]* Move field, absolute to register	3	§	0000 01F1 101R DDDD
MOVE *@SAddress, *Rd+ [, F]* Move field, absolute to indirect (postinc)	3	§	1101 01F0 000R DDDD
MOVE *@SAddress,@DAddress [, F]* Move field, absolute to absolute	5	§	0000 01F1 1100 0000
MOVI *IW, Rd* Move immediate (16 bits)	2	2	0000 1001 110R DDDD
MOVI *IL, Rd* Move immediate (32 bits)	3	2, 3 †	0000 1001 111R DDDD
MOVK *K, Rd* Move constant (5 bits)	1	1	0001 10KK KKKR DDDD
MOVX *Rs, Rd* Move X half of register	1	1	1110 110S SSSR DDDD
MOVY *Rs, Rd* Move Y half of register	1	1	1110 111S SSSR DDDD
MPYS *Rs, Rd* Multiply registers (signed)	1	5+FS1/2	0101 110S SSSR DDDD
MPYU *Rs, Rd* Multiply registers (unsigned)	1	¶	0101 111S SSSR DDDD
MWAIT Memory wait	1	minimun of 2	0000 0000 1000 0000
NEG *Rd* Negate register	1	1	0000 0011 101R DDDD
NEGB *Rd* Negate register with borrow	1	1	0000 0011 110R DDDD
NOP No operation	1	1	0000 0011 0000 0000
NOT *Rd* Complement register	1	1	0000 0011 111R DDDD
OR *Rs, Rd* OR registers	1	1	0101 010S SSSR DDDD
ORI *IL, Rd* OR immediate (32 bits)	3	2, 3 †	0000 1011 101R DDDD
PFILL XY Pattern fill	1	complex instruction	0000 1010 0011 0111
PIXBLT B, L Pixel block transfer, binary to linear	1	complex instruction	0000 1111 1000 0000

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
PIXBLT B, XY Pixel block transfer and expand, binary to XY	1	complex instruction	0000 1111 1010 0000
PIXBLT L, L Pixel block transfer, linear to linear	1	complex instruction	0000 1111 0000 0000
PIXBLT L, M, L Pixel block transfer, linear to linear with mask	1	complex instruction	0000 1110 0001 0111
PIXBLT L, XY Pixel block transfer, linear to XY	1	complex instruction	0000 1111 0010 0000
PIXBLT XY, L Pixel block transfer, XY to linear	1	complex instruction	0000 1111 0100 0000
PIXBLT XY, XY Pixel block transfer, XY to XY	1	complex instruction	0000 1111 0110 0000
PIXT *Rs,*Rd* Pixel transfer, register to indirect		2 + *P*	1111 100S SSSR DDDD
PIXT *Rs, *Rd* **.XY** Pixel transfer, register to indirect XY		¶	1111 000S SSSR DDDD
PIXT **Rs, Rd* Pixel transfer, indirect to register	1	3	1111 101S SSSR DDDD
PIXT **Rs, *Rd* Pixel transfer, indirect to indirect	1	4 + *P*	1111 110S SSSR DDDD
PIXT **Rs*.**XY**, *Rd* Pixel transfer, indirect XY to register	1	6 + *CS*	1111 001S SSSR DDDD
PIXT **Rs*.**XY**, **Rd*.**XY** Pixel transfer, indirect XY to indirect XY	1	¶	1111 010S SSSR DDDD
POPST Pop status register from stack	1	6, 7‡	0000 0001 1100 0000
PUSHST Push status register onto stack	1	2(1) 2(2)‡	0000 0001 1110 0000
PUTST *Rs* Copy register into status	1	3	0000 0001 101R DDDD
RETI Return from interrupt	1	¶	0000 1001 0100 0000
RETM Return from monitor	1	¶	0000 1000 0110 0000
RETS *[N]* Return from subroutine	1	¶	0000 1001 011N NNNN
REV *Rd* Return revision level	1	1	0000 0000 001R DDDD

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
RL *K, Rd* Rotate left, constant	1	1	0011 00KK KKKR DDDD
RL *Rs, Rd* Rotate left, register	1	1	0110 100S SSSR DDDD
RMO Rightmost one	1	1	0111 101S SSSR DDDD
RPIX *Rd* Replicate pixel	1	¶	0000 0010 100R DDDD
SETC Set carry	1	1	0000 1101 1110 DDDD
SETCDP Set CONVDP	1	¶	0000 0010 0111 0011
SETCMP Set CONVMP	1	¶	0000 0010 1111 1011
SETCSP Set CONVSP	1	¶	0000 0010 0101 0001
SETF *FS, FE, F* Set field parameters	1	1	0000 01F1 01FS SSSS
SEXT *Rd, F* Sign extend to long	1	2	0000 01F1 000R DDDD
SLA *K, Rd* Shift left arithmetic, constant	1	3	0010 00KK KKKR DDDD
SLA *Rs, Rd* Shift left arithmetic, register	1	3	0110 000S SSSR DDDD
SLL *K, Rd* Shift left logical, constant	1	1	0010 01KK KKKR DDDD
SLL *Rs, Rd* Shift left logical, register	1	1	0110 001S SSSR DDDD
SRA *K, Rd* Shift right arithmetic, constant	1	1	0010 10KK KKKR DDDD
SRA *Rs, Rd* Shift right arithmetic, register	1	1	0110 010S SSSR DDDD
SRL *K, Rd* Shift right logical, constant	1	1	0010 11KK KKKR DDDD
SRL *Rs, Rd* Shift right logical, register	1	1	0110 011S SSSR DDDD
SUB *Rs, Rd* Subtract registers	1	1	0100 010S SSSR DDDD

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

Syntax and Description	Words	Machine States	16-Bit Instruction Word MSB LSB
SUBB *Rs, Rd* Subtract registers with borrow	1	1	0100 011S SSSR DDDD
SUBI *IW, Rd* Subtract immediate (16 bits)	2	2	0000 1011 111R DDDD
SUBI *IL, Rd* Subtract immediate (32 bits)	3	2, 3 ‡	0000 1101 000R DDDD
SUBK *K, Rd* Subtract constant (5 bits)	1	1	0001 01KK KKKR DDDD
SUBXY *Rs, Rd* Subtract registers in XY mode	1	1	1110 001S SSSR DDDD
SWAPF *Rs, Rd,* 0 Swap field	1	¶	0111 111S SSSR DDDD
TFILL XY Trapezoid fill	1	complex instruction	0000 1110 1111 1010
TRAP *N* Software interrupt	1	¶	0000 1001 000N NNNN
TRAPL Software interrupt, signed	2	¶	0000 1000 0000 1111
VBLT Linear VRAM pixel block transfer	1	complex instruction	0000 1000 0101 0111
VFILL Linear VRAM fast fill	1	complex instruction	0000 1010 0101 0111
VLCOL Latch COLOR1 into the VRAM color registers	1	2(1)	0000 1010 0000 0000
XOR *Rs, Rd* Exclusive OR registers	1	1	0101 011S SSSR DDDD
XORI *IL, Rd* Exclusive OR immediate value (32 bits)	3	2, 3 †	0000 1011 110R DDDD
ZEXT *Rd, F* Zero extend to long	1	1	0000 01F1 001R DDDD

† First value if immediate data is long-word aligned, second value if immediate data is not long-word aligned
‡ First value if the SP is long-word aligned, second value if the SP is not long-word aligned
¶ See Section 15.1 page 15-2.
§ See Section 15.2 page 15-10.

13.3 Move Instructions Summary

The TMS34020 supports a variety of move instructions, allowing you to move immediate values into registers, move data between registers, and move data between registers and memory. Table 13–1 summarizes the various types of move instructions.

Table 13–1. Summary of MOVE Instructions

Move Type	Mnemonic	Description
Register	MOVE	Move register to register
Constant	MOVK	Move constant (5 bits)
	MOVI	Move immediate (16 bits)
	MOVI	Move immediate (32 bits)
XY	MOVX	Move 16 LSBs of register (X half)
	MOVY	Move 16 MSBs of register (Y half)
Multiple register	MMFM	Move multiple registers from memory
	MMTM	Move multiple registers to memory
Byte	MOVB	Move byte (8 bits, 9 addressing modes)
Field	MOVE	Move field to/from memory/register (18 addressing modes)

13.3.1 Register-to-Register Moves

The **MOVE** *Rs,Rd* instruction is a register-to-register move; it moves a full 32 bits of data between any two general-purpose registers. *This is the only MOVE instruction that allows you to move data between register files A and B.*

13.3.2 Value-to-Register Moves

The **MOVI** and **MOVK** instructions move immediate values into registers. MOVK moves a zero-extended value into a register; the value must be in the range of 1 to 32. The MOVI instruction has two forms; it can move a 16-bit or a 32-bit immediate value.

13.3.3 XY Moves

The **MOVX** and **MOVY** instructions move values into the 16 LSBs or 16 MSBs, respectively, of a register.

13.3.4 Multiple-Register Moves

The **MMTM** and **MMFM** instructions use register-direct operands. MMTM allows you to save several register values in memory; MMFM allows you to retrieve register values from memory. Both instructions have two types of operands:

- The *Rp* operand is a *register pointer.* For the MMTM instruction, Rp contains the memory address where MMTM stores the register values. For the MMFM instruction, Rp contains the memory address from which MMFM loads the stored register values.
- The *register list* operand is an optional list of registers. It specifies which registers are stored or retrieved and also indicates the storing or retrieval order.

Note that Rp and all the registers in the list *must be in the same register file.* If you are saving or restoring registers that are from both files, two MMTMs are required.

13.3.5 Byte Moves

The **MOVB** instruction is a special form of the MOVE instruction; when you use MOVB, the field size is restricted to 8 bits. MOVB supports nine combinations of operand formats. There are three basic combinations:

- Register to memory (requires a field insertion),
- Memory to register (requires a field extraction), and
- Memory to memory (requires both field insertion and extraction).

Note that the MOVB instruction does not move data between registers.

The MOVB instruction allows a byte to begin on any bit boundary in memory. The byte's memory address points to the LSB of the byte. When a byte is moved into a register, the byte's LSB coincides with the register's LSB; the byte is sign-extended into the 24 MSBs of the register.

See Table 13–4, page 13-154, for a summary of the valid combinations of operand formats for the MOVB instruction.

Sequences of byte moves are more efficient if the byte addresses are aligned on even 8-bit boundaries.

13.3.6 Field Moves

The **MOVE** instruction supports eighteen combinations of operand formats. There are four basic combinations:

- Register to register
- Register to memory
- Memory to register
- Memory to memory

The MOVE instruction moves a *field.* A field is a configurable data structure that is identified by its starting address and its length. Field lengths can range from 1 to 32 bits. A field's memory address points to the LSB of the field; the field occupies contiguous bits. A field in a register is right-justified within the register; the field's LSB coincides with the register's LSB.

Note that all forms of the MOVE instruction have an optional **F** parameter. (MOVE Rs,Rd is an exception to this; it doesn't have an F parameter because it always moves 32 bits.) **F** selects the field size and field extension for the MOVE:

- If **F=0,** FS0 and FE0 determine the field size and extension.
- If **F=1,** FS1 and FE1 determine the field size and extension.

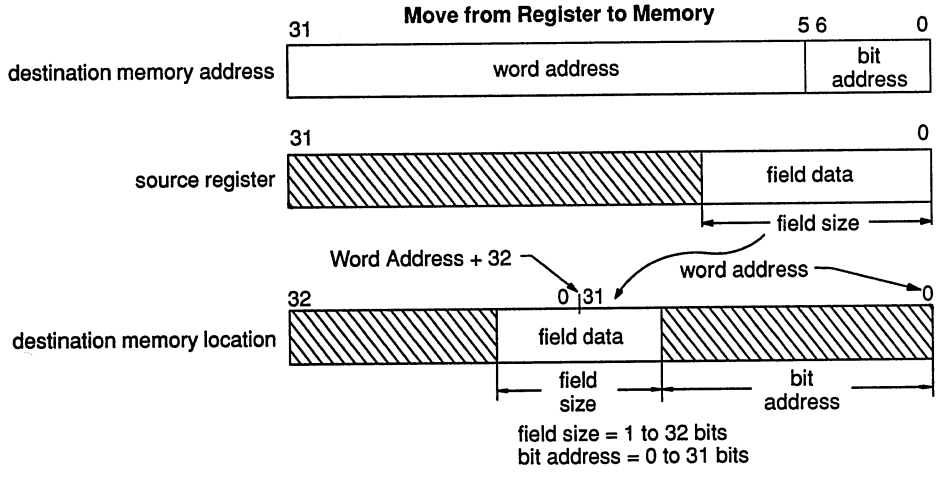
If you don't specify 0 or 1, 0 is used as the default. The selected field size determines the size of the field that is moved. A moved field is either sign-extended or zero-extended, depending on the value of the appropriate field extension bit. You can use the SETF instruction to set the field size and extension.

See Table 13–5, page 13-159, for a summary of the valid combinations of operand formats for the MOVE instruction.

13.3.6.1 Register-to-Memory Moves

Figure 13–8 illustrates the register-to-memory move operation. In this type of move, the source register contains the right-justified field data (width is specified by the field size). The destination location is the bit position pointed to by the destination memory address. The address consists of two portions: one defines the starting word in which the field is to be written and the other defines an offset into that word, the bit address. Depending on the bit address within this word and the field size, the destination location may extend into one or two long words.

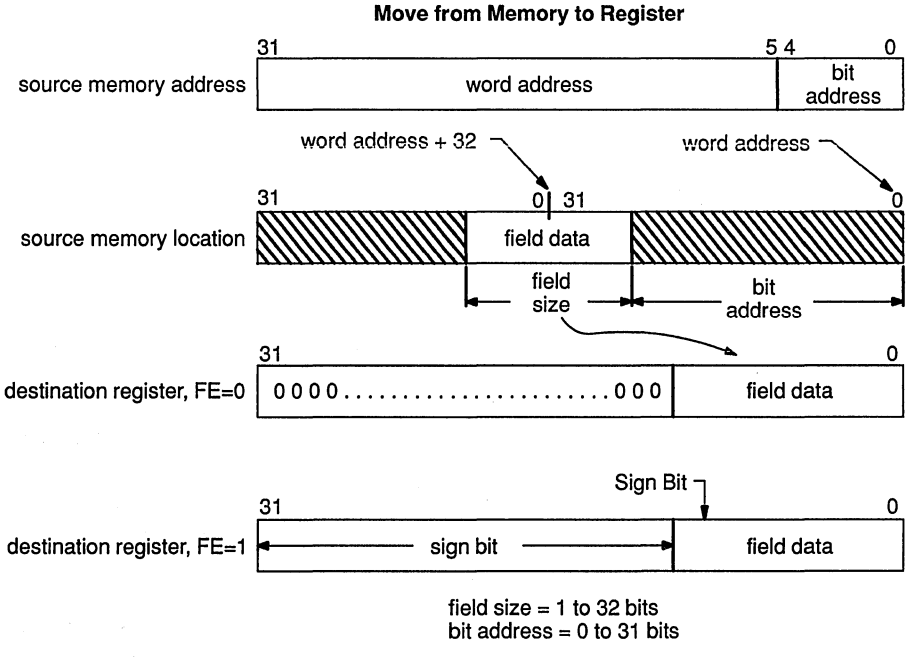
Figure 13–8. Register-to-Memory Moves



13.3.6.2 Memory-to-Register Moves

Figure 13–9 shows the memory-to-register move operation. The source memory location is the bit position pointed to by the source address. The address consists of two portions: one defines the starting word from which the field is to be read and the other defines an offset into that word, the bit address. Depending on the bit address within this word and the field size, the source location may extend into two or more long words. After the move, the destination register LSBs contain the right-justified field data (width is specified by the field size). The registers's MSBs contain either all 1s or all 0s.

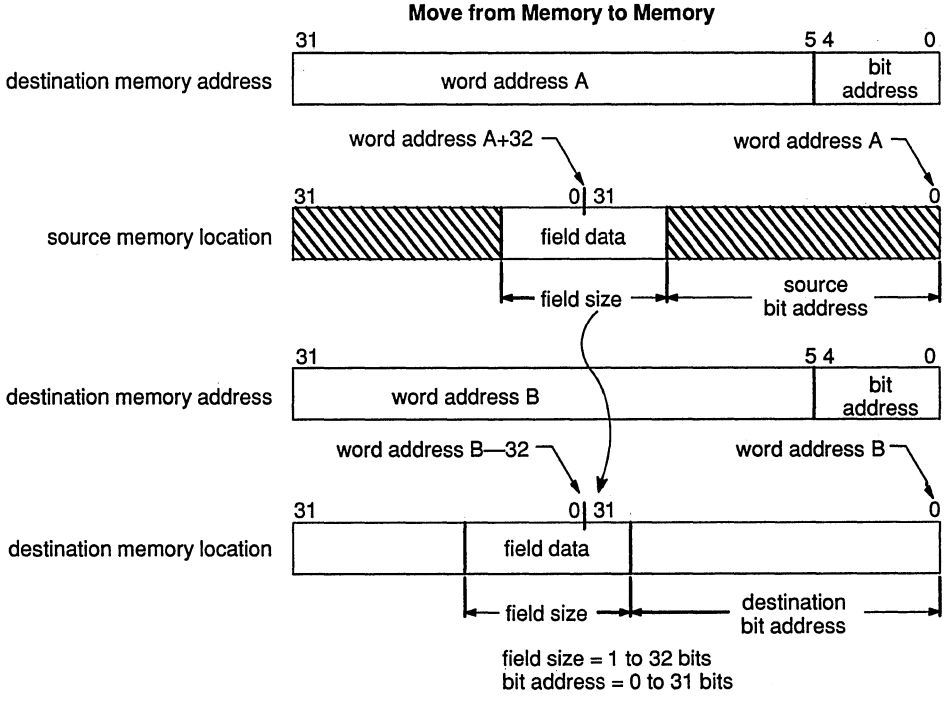
Figure 13–9. Memory-to-Register Moves



13.3.6.3 Memory-to-Memory Moves

Figure 13–10 shows a memory-to-memory field move operation. The source memory location is the bit position pointed to by the source address. The destination location is the bit position pointed to by the destination memory address. Depending on the bit addresses within the respective words and the field size, either the source location or destination locations may extend into two or more long words. After the move, the destination location contains the field data from the source memory location.

Figure 13–10. Memory-to-Memory Moves



13.4 Arithmetic, Logical, and Compare Instructions

The TMS34020 supports a full range of arithmetic, logical, and compare instructions. Most of these instructions use register-direct operands; some use a combination of immediate and register-direct operands. Some instructions have several versions; each uses a different operand format. For example, the *ADD* instruction has several versions:

- The **ADD** instruction uses register-direct operands for both the source and destination operands.
- The **ADDI** instruction uses an immediate source with a destination register.
- The **ADDK** instruction uses a 5-bit constant as the source operand with a destination register.
- The **ADDXY** instruction is similar to the ADD instruction because both operands are register-direct operands; however, the registers contain XY values.

Some instructions that have immediate values as source operands (such as the ADDI instruction) have two forms: a *short form* and a *long form.* In the short form, the source operand is a *16-bit* immediate value, and the instruction occupies *two words.* In the long form, the source operand is a *32-bit* immediate value, and the instruction occupies *three words.* Each form of the instruction has an optional third operand: **W** for short and **L** for long. If you don't use the **W** or **L** operand, the assembler chooses the short or the long form, depending on the size of the source operand. Using **W** or **L** *forces* the assembler to use the short or long form, respectively. If you use **W** and the source value is greater than 16 bits, the assembler discards all but the 16 LSBs and issues a warning message. If you use **L** and the source value is less than 32 bits, the assembler sign-extends the value to 32 bits.

Some instructions that use immediate operands have only one version. In this case, the operand is long (32 bits).

Note:

When an instruction's source and destination operands are both register-direct operands, the registers *must be in the same file.* (The MOVE *Rs, Rd* instruction is an exception to this rule.)

13.5 Program-Control and Context-Switching Instructions

The TMS34020 supports a variety of instructions that allow you to control program flow and to save and restore information by letting you do the following:

- call and return from subroutines
- enable or disable interrupts
- set software interrupts
- set, save, or restore status information
- use jump instructions to redirect program flow

Most of these instructions use register-direct or absolute operands; however, several of them have no operands.

13.5.1 Subroutine Calls and Returns

The TMS34020 allows you to call a subroutine in three ways:

- indirectly, by loading an address into a register
- directly, by using an absolute address
- relatively, by specifying an address that is an offset

The CALL instructions automatically save status information on the stack. The RETS instruction pops status information off of the stack and returns control to the program or routine that called the subroutine.

13.5.2 Interrupt Handling

The TMS34020's EINT and DINT instructions allow you to enable or disable hardware interrupts by providing control of the IE status bit. The TMS34020 also supports a TRAP instruction that provides you with control over 32 software interrupts. TRAPL allows 64K (–32,768 to +32,767) interrupts.

13.5.3 Setting, Saving, and Restoring Status Information

Although some instructions automatically save or restore status information, you will often want explicit control over these functions. The TMS34020 supports several instructions that allow you to save and restore PC and ST information. The TMS34020 also supports a SETF instruction that allows you to set field-0/field-1 information in the status register.

13.5.4 Jump Instructions

The TMS34020 supports both conditional and unconditional jumps. The conditional jumps use absolute operands or a combination of register-direct and absolute operands.

- There are four DSJ instructions.
  - **DSJ** and **DSJS** decrement the contents of a register and jump to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction.

    DSJ provides a jump range of –32,768 to +32,767 words; DSJS provides a jump range of ±32 words (excluding 0).

- The operation of **DSJEQ** and **DSJNE** depends on the value of the Z (zero) status bit.

  DSJEQ decrements the contents of Rd when **Z=1** and jumps to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction. If **Z=0,** DSJEQ skips the jump, and execution continues with the next instruction.

  DSJNE decrements the contents of Rd when **Z=0** and jumps to the specified address if the new contents of Rd do not equal 0. If Rd is decremented to 0, then execution continues with the next instruction. If **Z=1,** DSJNE skips the jump, and execution continues with the next instruction.

  The address specified for the DSJ instructions is relative; the assembler uses this address automatically to calculate a displacement, and then it inserts the displacement into the instruction.

  DSJEQ and DSJNE provide a jump range of 32K words (excluding 0).

- The **JUMP** instruction is unconditional. The source register contains the address for the jump.

- The conditional jump instructions, **JA***cc* and **JR***cc*, use the condition codes listed in Table 13–2.

  The JRcc instruction has a long and a short form. The short form supports a jump range of ±127 words (excluding 0). The long form supports a jump range of ±32K words (excluding 0).

  The 32-bit address specified for the JA*cc* instruction is absolute; the assembler inserts this address into words 2 and 3 of the instruction. The address specified for the JRcc instructions is relative; the assembler uses this address automatically to calculate a displacement, and then it inserts the displacement into the instruction. The short form has an 8-bit displacement that is inserted into bits 0—7 of the opcode; the opcode is 1 word long. The long form has 16-bit displacement; the opcode is 2 words long, and the displacement occupies the entire 16 bits of the second word.

Table 13–2 lists the condition codes used with the JR*cc* and JA*cc* instructions. (To use the codes, replace the *cc* with the appropriate mnemonic code; for example, JRUC, JALS, JRYGT, etc.) Before using one of these jump instructions, use the CMP, CMPI, or CMPXY instruction; the compare instructions set the condition codes for the jump by subtracting a source value from a destination value. The first mnemonics code column in Table 13–2 lists the codes that can be used for a jump following a CMP or CMPI. The second mnemonics code column list codes that can be used for a jump following a CMPXY (codes that are preceded with an *X* can be used with the result of the X comparison and codes that are preceded with a *Y* can be used with the result of the Y comparison).

Table 13–2. Condition Codes for JRcc and JAcc Instructions

	Mnemonic				
	Non XY	**XY**	**Result of Compare**	**Status Bits**	**Code**
Unconditional Compares	UC	—	Unconditional	Don't care	0000
Unsigned Compares	LO (C) (B)	— YN	Dst lower than Src	C	1000
	LS	YLE	Dst lower or same as Src	C + Z	0010
	HI	YGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	HS (NC) (NB)	YNN	Dst higher or same as Src	$\overline{C}$	1001
	EQ (Z)	— YZ	Dst = Src	Z	1010
	NE (NZ)	— YNZ	Dst ≠ Src	$\overline{Z}$	1011
Signed Compares	LT	XLE	Dst < Src	$(N \cdot V) + (\overline{N}\ \overline{V})$	0100
	LE	—	Dst ≤ Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V) + Z$	0110
	GT	—	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	GE	XGT	Dst ≥ Src	$(N \cdot V) + (\overline{N}\ \overline{V})$	0101
	EQ (Z)	— YZ	Dst = Src	Z	1010
	NE (NZ)	— YNZ	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	Z (EQ)	YZ	Result = zero	Z	1010
	NZ (NE)	YNZ	Result ≠ zero	$\overline{Z}$	1011
	P	—	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	N	XZ	Result is negative	N	1110
	NN	XNZ	Result is nonnegative	$\overline{N}$	1111
General Arithmetic	Z (EQ)	YZ	Result is 0	Z	1010
	NZ (NE)	YNZ	Result ≠ 0	$\overline{Z}$	1011
	C (LO) (B)	YN	Carry set on result	C	1000
	NC (HS) (NB)	YNN	No carry on result	C	1001
	B (JALO) (C)	YN	Borrow set on result	C	1000
	NB (HS) (NC)	YNN	No borrow on result	$\overline{C}$	1001
	V †	XN	Overflow on result	V	1100

Note: A mnemonic code in parentheses is an alternate code for the preceding code.

Key: † Also used for window clipping
+ Logical OR
· Logical AND
— Logical NOT

13.6 Shift Instructions

The TMS34020 supports several instructions that left-rotate, left-shift, or right-shift the contents of the destination register. These instructions use register-direct operands or a combination of register-direct and immediate operands; the shift amount is specified by the value of a 5-bit constant **or** by the value specified in the 5 LSBs of a source register. (Note that the **SRA** *Rs, Rd* and **SRL** *Rs, Rd* use the 2s complement of Rd's 5 LSBs.)

- The **RL** instruction left-rotates the contents of the destination register. (This rotation is a barrel shift.) The bits shifted out of the MSB are shifted into the LSB. The C (carry) bit is set to the final value shifted out of the MSB.
- The **SLA** instruction left-shifts the contents of the destination register. 0s are shifted into the LSBs. The MSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the MSB. If either the N (sign) bit or any of the bits shifted out differ from the original sign bit, the V (overflow) bit is set.
- The **SLL** instruction left-shifts the contents of the destination register. 0s are shifted into the LSBs. The MSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the MSB. The main difference between SLL and SLA is that SLL does not check to see if the sign bit changes.
- The **SRA** instruction right-shifts the contents of the destination register. The value of the sign bit is shifted into the MSBs; this sign-extends the value and preserves the original value of the sign bit. The LSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the LSB.
- The **SRL** instruction right-shifts the contents of the destination register. 0s are shifted into the MSBs, beginning with bit 31. The LSBs are shifted out through the C (carry) bit so that the C bit is set to the final value shifted out of the LSB. The main difference between SRL and SRA is that SRL does not preserve the original value of the sign bit.

13.7 XY Instructions

Table 13–3 summarizes the instructions that use XY addresses. This is useful for specifying pixel addresses on the screen. Many of the graphics instructions use XY addressing; the TMS34020 instruction set also supports several other instructions that allow you to manipulate XY addresses.

An XY address is a 32-bit address that is divided into two parts. Within a register, the 16 LSBs contain the X half of the address; the 16 MSBs contain the Y half. The two parts are treated as completely separate values; for example, when you use ADDXY, the X half does not propagate into the Y half.

Table 13–3. Summary for XY Instructions

Instruction	Description	Instruction	Description
ADDXYI *IL, Rd*	Add *IL* to register in XY	**MOVX** *Rs, Rd*	Move X half of Rs to X half of Rd
ADDXY *Rs, Rd*	Add registers in XY	**MOVY** *Rs, Rd*	Move Y half of Rs to Y half of Rd
CPW *Rs, Rd*	Compare point to window	**PFILL**	Pattern fill
CMPXY *Rs, Rd*	Compare registers in XY mode	**PIXBLT B, XY**	Pixel block transfer (binary to XY)
CVDXYL *Rd*	Convert destination XY address to linear	**PIXBLT L, XY**	Pixel block transfer (linear to XY)
CVMXYL *Rd*	Convert mask to linear	**PIXBLT XY, L**	Pixel block transfer (XY to linear)
CVSXYL *Rs, Rd*	Convert source XY address to linear	**PIXBLT XY, XY**	Pixel block transfer (XY to XY)
CVXYL *Rs, Rd*	Convert XY address to linear address	**PIXT** *Rs, *Rd*.**XY**	Pixel transfer (register to indirect XY)
DRAV *Rs, Rd*	Draw and advance	**PIXT** **Rs*.**XY**, *Rd*	Pixel transfer (indirect XY to register)
FILL XY	Fill array with processed pixels	**PIXT** **Rs*.**XY**, **Rd*.**XY**	Pixel transfer (indirect XY to indirect XY)
LINE {0 | 1}	Line draw	**SUBXY** *Rs, Rd*	Subtract registers in XY mode
LINIT	Line initialization	**TFILL**	Trapezoid fill

- The **PIXBLT** and **FILL** instructions use XY source and/or destination addresses.
- The **PIXT** instructions use the contents of registers as XY addresses.
- The **LINE** instruction draws a line along points that are calculated as XY addresses.
- The **MOVX/MOVY** instructions move the X or Y half of a source register into the X or Y half of a destination register.
- The arithmetic and logical instructions add, subtract, or compare the X and Y halves of the registers separately.

13.8 Instructions New to the TMS34020

The following is a list of new instructions for the TMS34020; these instructions were not included with the TMS34010.

Instruction Name	Instruction Name
ADDXYI	IDLE
BLMOVE	LINIT
CEXEC, 2 versions	MWAIT
CLIP	PFILL XY
CMOVCG	PIXBLT L,M,L
COMVCM, 2 versions	RETM
CMOVCS	RMO
CMOVGC, 2 versions	RPIX
CMOVMC, 3 versions	SETCDP
CMPK	SETCMP
CVDXYL	SETCSP
CVMXYL	SWAPF
CVSXYL	TFILL
EXGPS	TRAPL
FPIXEQ	VBLT
FPIXNE	VFILL
FLINE	VLCOL
GETPS	

13.9 Alphabetical Instruction Reference

The remainder of this section is an alphabetical reference of the TMS34020 assembly language instructions. Most instructions begin on a new page, and contains the following information:

- **Syntax:** Shows you how to enter an instruction. (The Preface describes the symbols used in instruction syntaxes.)
- **Execution:** Illustrates the effects of instruction execution on CPU registers and memory.
- **Instruction Words:** Shows the object code generated by an instruction.
- **Description:** Discusses the purpose of the instruction and any other general information related to the instruction.
- **Machine States:** Lists the instruction cache-enabled cycle timing.
- **Status Bits:** Lists the effects of instruction execution on the status bits (N, C, Z, and V).
- **Examples:** Show the effects of the instruction on memory and registers using various sets of data and initial conditions.

Several instructions discuss additional topics; for example, the conditional jump instructions list the conditions codes and mnemonics for various jumps, and the graphics instructions list the implied operands that they use.

Syntax

ABS *Rd*

Execution

|Rd| → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	0	0	R	Rd			

Description

ABS stores the absolute value of the contents of the destination register back into the destination register. This is accomplished by:

- Subtracting the contents of the destination register data from 0 **and**
- Storing the result back into Rd if status bit N indicates that the result is positive.

If the result of the subtraction is negative, then the original contents of the destination register are retained.

Machine States

1

Status Bits

N Set to the sign of the result of 0 – Rd; typically, N=*0* if the original contents of Rd are negative (unless Rd = 80000000h), 1 otherwise
C Unaffected
Z *1* if the original data is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise; an overflow occurs if Rd contains 80000000h (80000000h is returned)

Examples

Code	Before	After	
	A1	N C Z V	A1
ABS A1	7FFFFFFFh	1 x 0 0	7FFFFFFFh
ABS A1	FFFFFFFFh	0 x 0 0	00000001h
ABS A1	80000000h	1 x 0 1	80000000h
ABS A1	80000001h	0 x 0 0	7FFFFFFFh
ABS A1	00000001h	1 x 0 0	00000001h
ABS A1	00000000h	0 x 1 0	00000000h
ABS A1	FFFA0011h	0 x 0 0	0005FFEFh

Syntax

ADD *Rs, Rd*

Execution

Rs + Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	Rs				R	Rd			

Description

ADD adds the contents of the source register to the contents of the destination register and stores the result in the destination register.

You can use the ADD instruction with the ADDC instruction to perform multiple-precision arithmetic.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before		After	
	A1	**A0**	**N C Z V**	**A0**
`ADD A1,A0`	FFFFFFFFh	FFFFFFFFh	1 1 0 0	FFFFFFFEh
`ADD A1,A0`	FFFFFFFFh	00000001h	0 1 1 0	00000000h
`ADD A1,A0`	FFFFFFFFh	00000002h	0 1 0 0	00000001h
`ADD A1,A0`	FFFFFFFFh	80000000h	0 1 0 1	7FFFFFFFh
`ADD A1,A0`	FFFFFFFFh	80000001h	1 1 0 0	80000000h
`ADD A1,A0`	7FFFFFFFh	80000001h	0 1 1 0	00000000h
`ADD A1,A0`	7FFFFFFFh	80000000h	1 0 0 0	FFFFFFFFh
`ADD A1,A0`	7FFFFFFFh	00000001h	1 0 0 1	80000000h
`ADD A1,A0`	00000002h	00000002h	0 0 0 0	00000004h

Syntax

ADDC *Rs, Rd*

Execution

Rs + Rd + C → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	Rs				R	Rd			

Description

ADDC adds the contents of the source register, the carry bit, and the contents of the destination register and then stores the result in the destination register. Note that the status bits are set on the final result.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code		Before			After	
		C	A1	A0	NCZV	A0
ADDC	A1,A0	1	FFFFFFFFh	FFFFFFFFh	1 1 0 0	FFFFFFFFh
ADDC	A1,A0	1	FFFFFFFFh	00000001h	0 1 0 0	00000001h
ADDC	A1,A0	1	FFFFFFFFh	00000002h	0 1 0 0	00000002h
ADDC	A1,A0	1	FFFFFFFFh	80000000h	1 1 0 0	80000000h
ADDC	A1,A0	1	FFFFFFFFh	80000001h	1 1 0 0	80000001h
ADDC	A1,A0	1	FFFFFFFFh	80000001h	0 1 0 0	80000001h
ADDC	A1,A0	1	FFFFFFFFh	80000000h	0 1 1 0	00000000h
ADDC	A1,A0	1	7FFFFFFFh	00000001h	1 0 0 1	80000001h
ADDC	A1,A0	1	00000002h	00000002h	0 0 0 0	00000005h
ADDC	A1,A0	0	FFFFFFFFh	FFFFFFFFh	1 1 0 0	FFFFFFFEh
ADDC	A1,A0	0	FFFFFFFFh	00000001h	0 1 1 0	00000000h
ADDC	A1,A0	0	FFFFFFFFh	00000002h	0 1 0 0	00000001h
ADDC	A1,A0	0	FFFFFFFFh	80000000h	0 1 0 1	7FFFFFFFh
ADDC	A1,A0	0	FFFFFFFFh	80000001h	1 1 0 0	80000000h
ADDC	A1,A0	0	7FFFFFFFh	80000001h	0 1 1 0	00000000h
ADDC	A1,A0	0	7FFFFFFFh	80000000h	1 0 0 0	FFFFFFFFh
ADDC	A1,A0	0	7FFFFFFFh	00000001h	1 0 0 1	80000000h
ADDC	A1,A0	0	00000002h	00000002h	0 0 0 0	00000004h

Syntax

ADDI *IW, Rd* [, **W**]

Execution

Rd + 16-bit immediate value → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	R	Rd			
16-bit value															

Description

This ADDI instruction adds a sign-extended, 16-bit immediate value to the contents of the destination register and stores the result in the destination register. (The symbol *IW* in the syntax above represents a 16-bit, sign-extended immediate value.)

The assembler uses the short (16 bit) add if the immediate value is previously defined and is in the range –32,768 to 32,767. You can force the assembler to use the short form by following the register operand with a **W:**

```
ADDI   IW,Rd,W
```

If you use the **W** parameter and the value is outside the legal range, the assembler discards all but the 16 LSBs and issues an appropriate warning message.

You can use the ADDI instruction with the ADDC instruction to perform multiple-precision arithmetic.

Machine States

2

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	**N C Z V**	**A0**
`ADDI 1,A0`	FFFFFFFFh	0 1 1 0	00000000h
`ADDI 2,A0`	FFFFFFFFh	0 1 0 0	00000001h
`ADDI 1,A0`	7FFFFFFFh	1 0 0 1	80000000h
`ADDI 2,A0`	00000002h	0 0 0 0	00000004h
`ADDI 32767,A0`	00000002h	0 0 0 0	00008001h
`ADDI 0FFFF0010h,A0,W`	FFFFFFF0h	0 1 1 0	00000000h

Syntax

ADDI *IL, Rd* [, **L**]

Execution

Rd + 32-bit immediate value → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	1	R	Rd			
16 LSBs of 32-bit value															
16 MSBs of 32-bit value															

Description

This ADDI instruction adds a 32-bit, signed immediate value to the contents of the destination register and stores the result in the destination register. (The symbol IL in the syntax above represents a 32-bit, signed immediate value.)

The assembler uses the long (32 bit) ADDI if it cannot use the short form. You can force the assembler to use the long form by following the register operand with an **L:**

```
ADDI IL,Rd,L
```

Machine States

2 if the immediate data is long-word aligned
3 if the immediate data is not long-word aligned

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code		Before	After	
		A0	N C Z V	A0
ADDI	0FFFFFFFFh,A0	FFFFFFFFh	1 1 0 0	FFFFFFFEh
ADDI	80000000h,A0	FFFFFFFFh	0 1 0 1	7FFFFFFFh
ADDI	80000000h,A0	7FFFFFFFh	1 0 0 0	FFFFFFFFh
ADDI	32768,A0	7FFFFFFFh	1 0 0 1	80007FFFh
ADDI	2,A0,L	FFFFFFFFh	0 1 0 0	00000001h

Syntax

ADDK *K, Rd*

Execution

Rd + 5-bit constant → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	K					R	Rd			

Description

ADDK adds a 5-bit constant to the contents of the destination register and stores the result in the destination register. (The symbol K in the syntax above represents a 5-bit constant.)

The constant is treated as an unsigned number in the range 1—32; if the original value of K=32, then K is converted to 0 in the opcode. The assembler issues an error if you try to add 0 to a register with this instruction.

You can use the ADDK instruction with the ADDC instruction to perform multiple-precision arithmetic.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A0	**N C Z V**	**A0**
`ADDK 1,A0`	FFFFFFFFh	0 1 1 0	00000000h
`ADDK 2,A0`	FFFFFFFFh	0 1 0 0	00000001h
`ADDK 1,A0`	7FFFFFFFh	1 0 0 1	80000000h
`ADDK 1,A0`	80000000h	1 0 0 0	80000001h
`ADDK 32,A0`	80000000h	1 0 0 0	80000020h
`ADDK 32,A0`	00000002h	0 0 0 0	00000022h

Syntax

ADDXY *Rs, Rd*

Execution

X half of Rs + X half of Rd → X half of Rd
Y half of Rs + Y half of Rd → Y half of Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	Rs				R	Rd			

Description

ADDXY adds the signed source X value to the signed destination X value, adds the signed source Y value to the signed destination Y value, and stores the result in the destination register. The source and destination registers are treated as if they contained separate X and Y values. Any carry out from the lower (X) half of the register does not propagate into the upper (Y) half.

If you only want to add the X halves together, then one of the Y values must be 0 (the method for adding the Y halves is similar).

You can use this instruction to manipulate XY addresses in the register file; ADDXY is also useful for incremental figure drawing.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if resulting X field is all 0s, *0* otherwise
C The sign bit of the Y half of the result
Z *1* if Y field is all 0s, *0* otherwise
V The sign bit of the X half of the result

Examples

Code	Before		After	
	A1	A0	A0	N C Z V
ADDXY A1,A0	00000000h	00000000h	00000000h	1 0 1 0
ADDXY A1,A0	00000000h	00000001h	00000001h	0 0 1 0
ADDXY A1,A0	00000000h	00010000h	00010000h	1 0 0 0
ADDXY A1,A0	00000000h	00010001h	00010001h	0 0 0 0
ADDXY A1,A0	0000FFFFh	00000001h	00000000h	1 0 1 0
ADDXY A1,A0	0000FFFFh	00010001h	00010000h	1 0 0 0
ADDXY A1,A0	0000FFFFh	00000002h	00000001h	0 0 1 0
ADDXY A1,A0	0000FFFFh	00010002h	00010001h	0 0 0 0
ADDXY A1,A0	FFFF0000h	00010000h	00000000h	1 0 1 0
ADDXY A1,A0	FFFF0000h	00010001h	00000001h	0 0 1 0
ADDXY A1,A0	FFFF0000h	00020000h	00010000h	1 0 0 0
ADDXY A1,A0	FFFF0000h	00020001h	00010001h	0 0 0 0
ADDXY A1,A0	FFFFFFFFh	00010001h	00000000h	1 0 1 0
ADDXY A1,A0	FFFFFFFFh	00010002h	00000001h	0 0 1 0
ADDXY A1,A0	FFFFFFFFh	00020001h	00010000h	1 0 0 0
ADDXY A1,A0	FFFFFFFFh	00020002h	00010001h	0 0 0 0

Syntax

ADDXYI *IL, Rd*

Execution

LSW (X half) of 32-bit immediate value + X half of Rd → X half of Rd
MSW (Y half) of 32-bit immediate value + Y half of Rd → Y half of Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	0	0	0	R	Rd			
16 LSBs of 32-bit immediate value (X half)															
16 MSBs of 32-bit immediate value (Y half)															

Description

ADDXYI adds the 32-bit immediate value to the destination register in XY mode and stores the result back into the destination register.

Machine States

2 if the immediate data is long-word aligned
3 if the immediate data is not long-word aligned

Status Bits

N *1* if resulting X field is all 0s, *0* otherwise
C The sign bit of the Y half of the result
Z *1* if Y field is all 0s, *0* otherwise
V The sign bit of the X half of the result

Examples

Code	Before A0	After A0	N C Z V
`ADDXYI 000000000h,A0`	00000000h	00000000h	1 0 1 0
`ADDXYI 000000000h,A0`	00000001h	00000001h	0 0 1 0
`ADDXYI 000000000h,A0`	00010000h	00010000h	1 0 0 0
`ADDXYI 000000000h,A0`	00010001h	00010001h	0 0 0 0
`ADDXYI 00000FFFFh,A0`	00000001h	00000000h	1 0 1 0
`ADDXYI 00000FFFFh,A0`	00010001h	00010000h	1 0 0 0
`ADDXYI 00000FFFFh,A0`	00000002h	00000001h	0 0 1 0
`ADDXYI 00000FFFFh,A0`	00010002h	00010001h	0 0 0 0
`ADDXYI 0FFFF0000h,A0`	00010000h	00000000h	1 0 1 0
`ADDXYI 0FFFF0000h,A0`	00010001h	00000001h	0 0 1 0
`ADDXYI 0FFFF0000h,A0`	00020000h	00010000h	1 0 0 0
`ADDXYI 0FFFF0000h,A0`	00020001h	00010001h	0 0 0 0
`ADDXYI 0FFFFFFFFh,A0`	00010001h	00000000h	1 0 1 0
`ADDXYI 0FFFFFFFFh,A0`	00010002h	00000001h	0 0 1 0
`ADDXYI 0FFFFFFFFh,A0`	00020001h	00010000h	1 0 0 0
`ADDXYI 0FFFFFFFFh,A0`	00020002h	00010001h	0 0 0 0

Syntax

AND *Rs, Rd*

Execution

Rs AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rs				R	Rd			

Description

AND bitwise-ANDs the contents of the source register with the contents of the destination register and then stores the result in the destination register. Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	A1	A0	N C Z V	A0
`AND A1,A0`	FFFFFFFFh	FFFFFFFFh	x x 0 x	FFFFFFFFh
`AND A1,A0`	FFFFFFFFh	00000000h	x x 1 x	00000000h
`AND A1,A0`	00000000h	00000000h	x x 1 x	00000000h
`AND A1,A0`	AAAAAAAAh	55555555h	x x 1 x	00000000h
`AND A1,A0`	AAAAAAAAh	AAAAAAAAh	x x 0 x	AAAAAAAAh
`AND A1,A0`	55555555h	55555555h	x x 0 x	55555555h
`AND A1,A0`	55555555h	AAAAAAAAh	x x 1 x	00000000h

Syntax

ANDI *IL, Rd*

Execution

32-bit immediate value AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	0	0	R	Rd			
1s complement of 16 LSBs of *IL*															
1s complement of 16 MSBs of *IL*															

Description

ANDI bitwise-ANDs the value of a 32-bit immediate value with the contents of the destination register and then stores the result in the destination register. (The symbol IL in the syntax above represents a 32-bit immediate value.)

This is an alternate mnemonic for `ANDNI IL, Rd`. Note that the assembler stores the 1s complement of IL in the 2 extension words.

Machine States

2 if the immediate data is long-word aligned
3 if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code		Before	After				
		A0	**N**	**C**	**Z**	**V**	**A0**
`ANDI`	`0FFFFFFFFh,A0`	FFFFFFFFh	x	x	0	x	FFFFFFFFh
`ANDI`	`0FFFFFFFFh,A0`	00000000h	x	x	1	x	00000000h
`ANDI`	`00000000h,A0`	00000000h	x	x	1	x	00000000h
`ANDI`	`0AAAAAAAAh,A0`	55555555h	x	x	1	x	00000000h
`ANDI`	`0AAAAAAAAh,A0`	AAAAAAAAh	x	x	0	x	AAAAAAAAh
`ANDI`	`55555555h,A0`	55555555h	x	x	0	x	55555555h
`ANDI`	`55555555h,A0`	AAAAAAAAh	x	x	1	x	00000000h

Syntax

ANDN *Rs, Rd*

Execution

(NOT Rs) AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	Rs				R	Rd			

Description

ANDN bitwise-ANDs the 1s complement of the contents of Rs with the contents of Rd and then stores the result in the destination register.

Rs and Rd must be in the same register file. Note that `ANDN Rn, Rn` has the same effect as `CLR Rn`.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	A1	A0	N C Z V	A0
`ANDN A1,A0`	FFFFFFFFh	FFFFFFFFh	x x 1 x	00000000h
`ANDN A1,A0`	FFFFFFFFh	00000000h	x x 1 x	00000000h
`ANDN A1,A0`	00000000h	00000000h	x x 1 x	00000000h
`ANDN A1,A0`	AAAAAAAAh	55555555h	x x 0 x	55555555h
`ANDN A1,A0`	AAAAAAAAh	AAAAAAAAh	x x 1 x	00000000h
`ANDN A1,A0`	55555555h	55555555h	x x 1 x	00000000h
`ANDN A1,A0`	55555555h	AAAAAAAAh	x x 0 x	AAAAAAAAh

Syntax

ANDNI *IL, Rd*

Execution

(NOT 32-bit immediate value) AND Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	0	0	R	Rd			
16 LSBs of *IL*															
16 MSBs of *IL*															

Description

ANDNI bitwise-ANDs the 1s complement of a 32-bit immediate value with the contents of the destination register and then stores the result in the destination register. (The symbol IL in the syntax above represents a 32-bit immediate value.) ANDI also uses this opcode.

Machine States

2 if the immediate data is long-word aligned
3 if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code		Before	After	
		A0	N C Z V	A0
ANDNI	0FFFFFFFFh,A0	FFFFFFFFh	x x 1 x	00000000h
ANDNI	0FFFFFFFFh,A0	00000000h	x x 1 x	00000000h
ANDNI	00000000h,A0	00000000h	x x 1 x	00000000h
ANDNI	0AAAAAAAAh,A0	55555555h	x x 0 x	55555555h
ANDNI	0AAAAAAAAh,A0	AAAAAAAAh	x x 1 x	00000000h
ANDNI	55555555h,A0	55555555h	x x 1 x	00000000h
ANDNI	55555555h,A0	AAAAAAAAh	x x 0 x	AAAAAAAAh

Syntax

BLMOVE *S, D*

Execution

n bits from address in SADDR → *n* bits at address in DADDR

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	1	1	0	0	S	D

Description

BLMOVE transfers a specified number of bits of data starting from a specified source address to a specified destination address. BLMOVE is unusual in 2 respects:

- BLMOVE deals with a continuous block of memory (no pitch concept as in PIXBLT).
- BLMOVE moves a specified number of bits of data (not pixels).

The values of *S* and *D* determine how a block is moved:

- If *S*=1 and *D*=1, then

  B0, the source address, may be aligned on any bit boundary.

  B2, the destination address, may be aligned on any bit boundary.

  B7 should contain the number of bits to be moved.

  As BLMOVE executes, B2 is incremented to reflect the current state of the move. At the end of the move, B0 is incremented by the initial value stored in B7 so that B0 points to the bit that would have been moved next, had the move continued. At the end of the move, B2 points to the bit after the last bit moved.

- If *S*=0 and *D*=1, then

  B0, the source address, must be 32-bit long-word aligned.

  B2, the destination address, may be aligned on any bit boundary.

  B7 should contain the number of bits to be moved.

  As BLMOVE executes, B2 is incremented to reflect the current state of the move; B0 remains fixed. At the end of the move, B0 is incremented by the initial value stored in B7, and B2 points to the bit after the last bit moved.

- If *S*=1 and *D*=0, then

  B0, the source address, may be aligned on any bit boundary.

  B2, the destination address, must be 32-bit long-word aligned.

  B7 should contain the number of bits to be moved.

  As BLMOVE executes, B2 and B0 remain fixed. At the end of the move, B2 and B0 are incremented by the initial value stored in B7.

- If *S*=0 and *D*=0, then

  B0, the source address, must be 32-bit long-word aligned.

  B2, the destination address, must be 32-bit long-word aligned.

  B7 should contain the number of bits to be moved.

  As BLMOVE executes, B2 and B0 remain fixed. At the end of the move, B2 and B0 are incremented by the initial value stored in B7.

In all cases, the DYDX register initially contains a count of the number of bits to be moved. This count is decremented as BLMOVE executes, so DYDX reflects the number of bits remaining to be moved. At the end of the move, DYDX will contain 0.

Interrupts

If BLMOVE is interrupted, the PC is decremented to point back to the BLMOVE instruction, the PC and ST are pushed onto the stack, and the program control branches to the appropriate interrupt trap routine. At the end of the trap routine, BLMOVE restarts, so the trap routine must restore the B-file registers used by BLMOVE.

Implied Operands

Register	Name	Format	Description
B0	SADDR	Linear	Source block address
B2	DADDR	Linear	Destination block address
B7	DYDX	Integer	Number of bits to move

Machine States

complex instruction

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

This example shows how the memcpy C runtime-support function could be implemented using this instruction. The memcpy function is invoked with 3 arguments on the C parameter stack (pointed to by A14):

s1 destination address
s2 source address
n number of BYTES to move

Note that this function does not check for overlapping memory areas

```
        .globl  _memcpy    ; provide reference for external calls
STK     .set    A14        ; C–parameter stack pointer
SADDR   .set    B0         ; Source address register
DADDR   .set    B2         ; Destination address register
DYDX    .set    B7         ; Delta X/delta Y register
TEMP    .set    B14        ; Temporary register
_memcpy:
        mmtm    SP,SADDR,DADDR,DYDX,TEMP ;save the required registers
        move    STK,TEMP                 ; copy C–stack to B–file register
        move    *–TEMP,DADDR,1           ; pop s1 (FS 1 assumed to be 32)
        move    *–TEMP,SADDR,1           ; pop s2
        move    *–TEMP,DYDX,1            ; pop n (byte count)
        move    TEMP,STK                 ; Update C stack
        sll     3,DYDX                   ; convert to a bit count
        blmove    1,1                    ; perform the block move
        mmfm    SP,SADDR,DADDR,DYDX,TEMP ; restore the required registers
        rets    2              ; return to caller (with C–calling convention)
```

Syntax

BTST *constant, Rd*

Execution

Set status on value of constant in Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1s complement of constant					R	Rd			

Description

BTST tests a bit in the destination register and sets status bit Z accordingly. This form of the BTST instruction uses a 5-bit constant to specify the bit in Rd that is tested. The constant value must be an absolute expression that evaluates to a number in the range 0 to 31; if the constant value is greater than 31, the assembler issues a warning and truncates the value of the constant to its 5 LSBs.

Note that the assembler 1s-complements the constant value before inserting it into the opcode.

Machine States

1

Status Bits

N̄ Unaffected
C Unaffected
Z *1* if the bit tested is 0, *0* if the bit tested is 1
V Unaffected

Examples

Code	Before A0	After N C Z V
`BTST 0,A0`	55555555h	x x 0 x
`BTST 15,A0`	55555555h	x x 1 x
`BTST 31,A0`	55555555h	x x 1 x
`BTST 0,A0`	AAAAAAAAh	x x 1 x
`BTST 15,A0`	AAAAAAAAh	x x 0 x
`BTST 31,A0`	AAAAAAAAh	x x 0 x
`BTST 0,A0`	FFFFFFFFh	x x 0 x
`BTST 15,A0`	FFFFFFFFh	x x 0 x
`BTST 31,A0`	FFFFFFFFh	x x 0 x
`BTST 0,A0`	00000000h	x x 1 x
`BTST 15,A0`	00000000h	x x 1 x
`BTST 31,A0`	00000000h	x x 1 x

Syntax

BTST *Rs, Rd*

Execution

Set status on value of specified bit in Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	Rs				R	Rd			

Description

BTST tests a bit in the destination register and sets status bit Z accordingly. This form of the BTST instruction uses the 5 LSBs of the source register to specify the bit in Rd that is tested (the symbol Rs in the syntax above represents the source register). Note that the 27 MSBs of Rs are ignored.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the bit tested is 0, *0* if the bit tested is 1
V Unaffected

Examples

Code	Before		After
	A1	**A0**	**N C Z V**
`BTST A1,A0`	00000000h	55555555h	x x 0 x
`BTST A1,A0`	0000000Fh	55555555h	x x 1 x
`BTST A1,A0`	0000001Fh	55555555h	x x 1 x
`BTST A1,A0`	00000000h	AAAAAAAAh	x x 1 x
`BTST A1,A0`	0000000Fh	AAAAAAAAh	x x 0 x
`BTST A1,A0`	0000001Fh	AAAAAAAAh	x x 0 x
`BTST A1,A0`	FFFFFF8Fh	FFFF7FFFh	x x 0 x
`BTST A1,A0`	00000000h	FFFFFFFFh	x x 0 x
`BTST A1,A0`	0000000Fh	FFFFFFFFh	x x 0 x
`BTST A1,A0`	0000001Fh	FFFFFFFFh	x x 0 x
`BTST A1,A0`	00000000h	00000000h	x x 1 x
`BTST A1,A0`	0000000Fh	00000000h	x x 1 x
`BTST A1,A0`	0000001Fh	00000000h	x x 1 x

Syntax

CALL *Rs*

Execution

PC' → TOS
Rs → PC
SP – 32 → SP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	1	R	Rs			

Description

CALL pushes the address of the next instruction (PC') onto the stack, then jumps to a subroutine whose address is contained in the source register. You can use this instruction for indexed subroutine calls. Note that when Rs is the SP, SP is decremented **after** being written to the PC (the PC contains the original value of SP).

The TMS34020 always sets the 4 LSBs of the program counter to 0, so instructions are always word aligned.

The stack pointer (SP) points to the top of the stack; the stack is located in external memory. The stack grows in the direction of decreasing linear addresses. PC' is pushed onto the stack, and the SP is predecremented by 32 before the return address is loaded onto the stack. Stack pointer alignment affects timing as indicated in **Machine States**, below.

Use the RETS instruction (page 13-220) to return from a subroutine.

Machine States

3 + (1) if the SP is aligned
3 + (4) if the SP is not aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

```
CALL A0
```

Before			**After**	
A0	**PC**	**SP**	**PC**	**SP**
01234560h	04442210h	F000020h	01234560h	F000000h

Memory contains the following values after instruction execution:

Address	**Data**
F000010h	2220h
F000020h	0444h

Syntax

CALLA *Address*

Execution

PC' → TOS
Address → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	1	0	1	1	1	1	1
16 LSBs of Address															
16 MSBs of Address															

Description

CALLA pushes the address of the next instruction (PC') onto the stack, then jumps to the address contained in the 2 extension words. The *Address* operand is a 32-bit absolute address. This instruction is used for long or externally referenced calls (greater than ±32K words).

The 4 LSBs of the program counter are always set to 0, so instructions are always word aligned.

The stack pointer (SP) points to the top of the stack; the stack is located in external memory. The stack grows in the direction of decreasing linear address. PC' is pushed onto the stack, and the SP is predecremented by 32 before the return address is loaded onto the stack. Stack pointer alignment affects timing as indicated in **Machine States**, below.

Use the RETS instruction (page 13-220) to return from a subroutine.

Machine States

3 if immediate data is long-word aligned
4 if SP is also long-word aligned
3 + (3) if immediate data is not long-word aligned
4 + (3) if SP is also not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

```
CALLA 01234560h
```

Before		**After**	
PC	**SP**	**PC**	**SP**
04442210h	0F000020h	01234560h	0F000000h

Memory contains the following values after instruction execution:

Address	**Data**
F000010h	2240h
F000020h	0444h

Syntax

CALLR *Address*

Execution

PC' → TOS
PC' + (offset × 16) → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	1	1	1	1	1	1
offset															

Description

CALLR pushes the address of the next instruction (PC') onto the stack, then jumps to the subroutine at the address specified by the sum of the next instruction address and the signed word offset. This instruction is used for calls within a specified module or section.

The *Address* operand is a 32-bit address within ±32K words (–32,768 to 32,767) **of the PC.** The address must be defined within the current section; the assembler does not accept an address value that is externally defined or defined within a different section than PC'. The assembler calculates the offset value for the opcode as (Address – PC')/16, where PC' is the address of the instruction word immediately following the second word of the CALLR instruction.

The 4 LSBs of the program counter are always set to 0, so instructions are always word aligned.

The stack pointer (SP) points to the top of the stack; the stack is located in external memory. The stack grows in the direction of decreasing linear address. The PC is pushed on to the stack, and the SP is predecremented by 32 before the return address is loaded onto the stack. Stack pointer alignment affects timing as indicated in **Machine States,** below.

Use the RETS instruction (page 13-217) to return from a subroutine.

Machine States

3 + (1) if the SP is aligned
3 + (4) if the SP is not aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	**Before**		**After**	
	PC	**SP**	**PC**	**SP**
`CALLR 0447FFF0h`	04400000h	0F000020h	0447FFF0h	0F000000h
`CALLR 04480000h`	04400000h	0F000020h	04480000h	0F000000h

Memory contains the following values after instruction execution:

Address	**Data**
F000010h	0020h
F000020h	0440h

Syntax

CEXEC *size, command* [*, ID*] [,L]

Execution

ID, command → Coprocessor command

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
8 LSBs of coprocessor command								size	0	0	0	0	0	0	0
coprocessor ID					13 MSBs of coprocessor command										

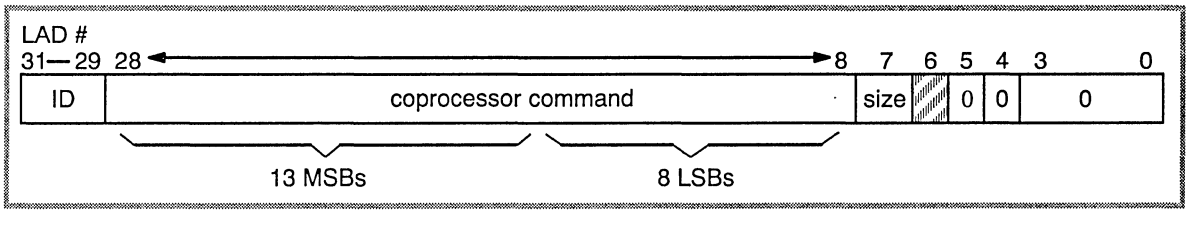
Description

CEXEC sends a 21-bit *command* to a coprocessor. The coprocessor may operate on the *command* without any transfer of data. The *size* operand is a value of 0 or 1 (0 is the default); the coprocessor interprets the size bit to determine the size of the values to be operated upon:

- If *size* = 0, then the coprocessor uses **32-bit values**.
- If *size* = 1, then the coprocessor uses **64-bit values**.

The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

The TMS34020 assembler checks the coprocessor command mode bits to determine which form of CEXEC is used. If both mode bits (6 and 7 of the coprocessor command) are 0, a short CEXEC is generated, otherwise a long CEXEC is generated. You can force a long CEXEC by placing an **,L** at the end of the CEXEC operand.

Machine States

2 (1) if the immediate data is long-word aligned
3 (1) if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This example sends coprocessor 2 the instruction *wxyz* and tells the coprocessor to use 32-bit values.

```
CEXEC 0,wxyz,2
```

Example 2

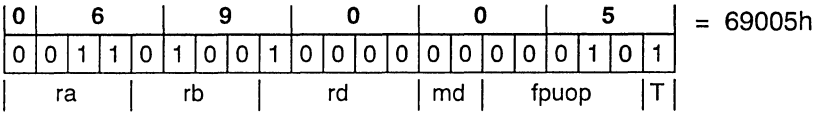
This example sends the default coprocessor the instruction *qrst* and tells the coprocessor to use 64-bit values.

```
CEXEC 1,qrst
```

Example 3

This example compares the single-precision (Size = 0, T = 1) contents (fpuop = 00010) of TMS34082 coprocessor 3 (ID = 3) registers RA3 (ra = 0011) and RB4 (rb = 0100). The result is stored in the coprocessor's RB0 register (rd = 10000).

```
CMPF_RA3_RB4  .set   069005h
              CEXEC  0,CMPF_RA3_RB4,3 ; size = 0 ID = 3
```



= 69005h

Syntax

CEXEC *size*, *command* [, *ID*]

Execution

ID, *command* → Coprocessor command

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	6 LSBs of coprocessor command						size
coprocessor ID			13 MSBs of coprocessor command												

Description

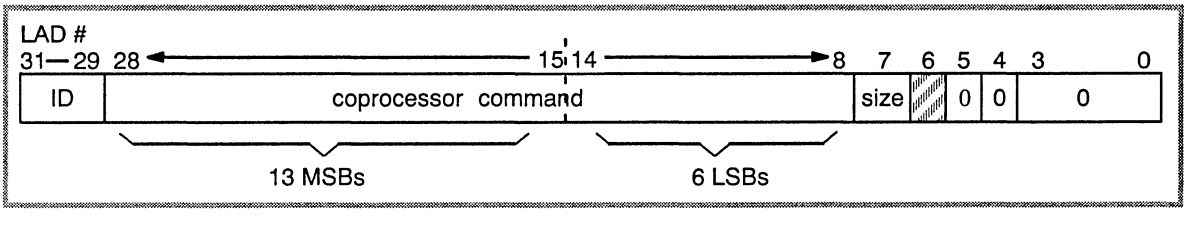
The CEXEC instruction sends a 21-bit *command* to a coprocessor. This version of CEXEC outputs bits 14 and 15 as 0 when sending the instruction, so that 19 bits of the 21-bit *command* may be set to any combination of 1s and 0s necessary to represent a particular coprocessor command (all 21 bits are valid—but only 19 may be specified). The coprocessor operates on the *instruction* without any transfer of data. The *size* operand is a value of 0 or 1; it determines the size of the values that the coprocessor uses:

- If *size* = 0, then the coprocessor uses **32-bit values.**
- If *size* = 1, then the coprocessor uses **64-bit values**.

The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The assembler uses this version of CEXEC when bits 6 and 7 of the coprocessor instruction (corresponding to bits 14 and 15 on the LAD bus) are 0. You can force a long CEXEC by placing an **,L** at the end of the CEXEC operand.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

2 (1)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1 This example sends coprocessor 2 the instruction *wxyz* and tells the coprocessor to use 32-bit values.

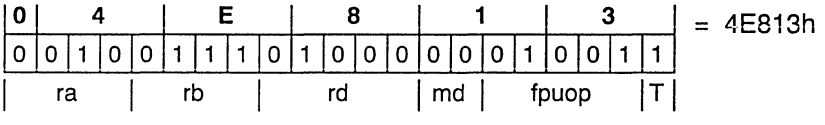
```
CEXEC 0,wxyz,2
```

Example 2 This example sends coprocessor 0 the instruction *qrst* and tells the coprocessor to use 64-bit values.

```
CEXEC 1,qrst
```

Example 3 This example divides the double-precision (Size = 1, T = 1) contents (fpuop = 01001) of the default TMS34082 coprocessor registers RA2 (ra = 0010) by RB7 (rb = 0111). The result is stored in the coprocessor's RA8 register (rd = 01000).

```
DIVD_RA2_RB7 .set   04E813h
             CEXEC  1,DIVD_RA2_RB7   ; size = 1 ID = default
```



= 4E813h

Syntax

CLIP

Execution

CLIP destination array
adjusted DYDX → DYDX
adjusted DADDR → DADDR

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	1	1	0	0	1	0

Description

CLIP adjusts the pixel array, specified by the XY address contained in DADDR and the XY dimensions contained in DYDX, to fit within the rectangular window specified by WSTART and WEND. The adjusted pointer and dimensions replace the original values in DADDR and DYDX. If the array currently fits in the window and no adjustment is necessary, the V bit will be set to 0. If the array lies entirely outside the window, then the Z bit is set to a 1, and DADDR and DYDX are left unchanged.

CLIP is the only instruction that can deal with overflowing arrays; an overflowing array is different from an array that simply strays outside a window. For more information, refer to subsection 12.7.4.4, Clip Instruction for Preclipping a Pixel Array, on page 12-23.

Implied Operands

Register	Name	Format	Description
B2	DADDR	XY	Destination array address
B5	WSTART	XY	Window start corner
B6	WEND	XY	Window end corner
B7	DYDX	XY	Array dimensions

Machine States

Complex Instruction

Status Bits

N Unaffected
C Unaffected
Z Z = 1 if the array lies entirely outside window, *0* if the array lies all or partially inside the window
V *1* if any portion of the array lies outside the window, *0* otherwise

Example

This is an example of a C-compatible assembly routine which fills a rectangle on the screen. The routine takes these 4 arguments: width, height, xleft, and ytop. Note that the CLIP instruction is used to clip the rectangle to the screen.

This routine makes the following assumptions:

- These B registers and I/O registers have been set up by the calling program:

  B-file registers DPTCH, OFFSET, WSTART, WEND and COLOR1

  I/O registers CONTROL, CONVDP, PSIZE, PMASK and CONFIG

- The system contains a global flag _vfill_ok which is cleared if the VFILL is not possible. Reasons for this may be:
  - DPTCH is not an integral multiple of 80h
  - PSIZE is 1 or 2
  - Pixel processing is not set to replace
  - Transparency is not set
  - The system does not contain VRAMs which support this feature

```
DADDR   .set    B2                      ;Destination address register
DYDX    .set    B7                      ;Delta X/delta Y register
CONTROL .set    0C00000B0h              ;Control register

        .globl  _fill_rect      ; provide reference for external calls
        .ref    _vfill_ok               ; flag to enable VFILLs

_fill_rect:
        mmtm    SP,B2,B7,B10,B11,B12    ;save required registers
        move    A14,B10         ;move c-stack pointer into B-file
        move    *-B10,DYDX,1            ;get width
        move    *-B10,B12,1             ;get height
        sll     16,B12
        movy    B12,DYDX                ;concatenate width & height
        move    *-B10,DADDR,1           ;get xleft
        move    *-B10,B12,1             ;get ytop
        move    B10,A14                 ;restore c-stack pointer
        sll     16,B12
        movy    B12,DADDR               ;concatenate xleft & ytop
        move    @_vfill_ok,A8,1         ;get state of vfill flag
        jrz     no_vfill
        clip                            ;clip to the window
        jrz     exit                    ;if outside the window, exit
        cvdxyl  DADDR                   ;convert to linear dest address
        vlcol                           ;load VRAM color latches
        vfill   L                       ;perform linear fill
        jruc    exit
no_vfill:
        fill    XY              ;fill the rectangle using standard fill
exit:
        mmfm    SP,B2,B7,B10,B11,B12    ;restore required registers
        rets    2
```

Syntax

CLR *Rd*

Execution

Rd XOR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rd				R	Rd			

Description

CLR clears the destination register by XORing the contents of the register with itself. This is an alternate mnemonic for XOR Rd,Rd (page 13-266).

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1*
V Unaffected

Examples

Code	Before A0	After A0	NCZV
`CLR A0`	FFFFFFFFh	00000000h	x x 1 x
`CLR A0`	00000001h	00000000h	x x 1 x
`CLR A0`	80000000h	00000000h	x x 1 x
`CLR A0`	AAAAAAAAh	00000000h	x x 1 x

Syntax

CLRC

Execution

0 → C

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0

Description

CLRC sets the C (carry) bit in the status register to 0; the rest of the status register is unaffected. Use SETC instruction (page 13-226) to set the C bit.

This instruction is useful for returning a true/false value (in the carry bit) from a subroutine without using a general-purpose register.

Machine States

1

Status Bits

B Unaffected
C *0*
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	ST	N C Z V	ST	N C Z V
CLRC	F0000000h	1 1 1 1	B0000000h	1 0 1 1
CLRC	40000010h	0 1 0 0	00000010h	0 0 0 0
CLRC	B000001Fh	1 0 1 1	B000001Fh	1 0 1 1

Syntax

CMOVCG *Rd*$_1$ [, *Rd*$_2$ [, *size*]], *command* [, *ID*]

Execution

ID, *command* → Coprocessor command
Coprocessor → *Rd*$_1$, *Rd*$_2$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rd$_1$			
8 LSBs of coprocessor command								size	0	0	R	Rd$_2$			
coprocessor ID			13 MSBs of coprocessor command												

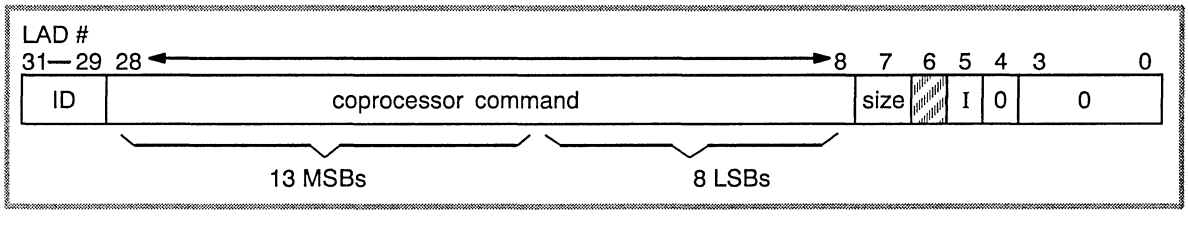
Description

CMOVCG moves one or two 32-bit values from a coprocessor to the specified TMS34020 destination register or registers:

- If *size* = 0, then the coprocessor moves one 32-bit value into *Rd*$_1$. *Rd*$_2$ is ignored; bits 0 to 4 of the second instruction word are set to 0.
- If *size* = 1, then the coprocessor moves a 64-bit value in to *Rd*$_1$ and *Rd*$_2$. The order in which the MSBs and LSBs are transferred depends on the coprocessor used.

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to define the source data for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



I is the coprocessor parameter index bit. For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

If size = 0
- 4 if the immediate data is long-word aligned
- 5 if the immediate data is not long-word aligned

If size = 1
- 5 if the immediate data is long-word aligned
- 6 if the immediate data is not long-word aligned

Status Bits

N *1* if the last 32-bit value read is negative, *0* otherwise
C Unaffected
Z *1* if the last 32-bit value read is 0, *0* otherwise
V 0

Example 1 This example moves 32 bits, specified by coprocessor instruction *wxyz,* from coprocessor 2 to register A0.

```
CMOVCG A0,wxyz,2
```

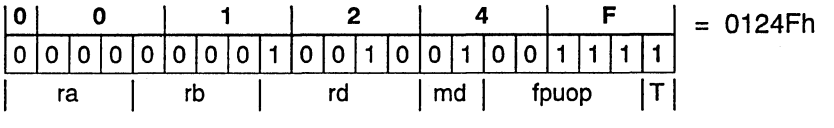
Example 2 This example moves one 64-bit value, specified by coprocessor instruction *jklm*, from coprocessor 5 to registers A3 and B7.

```
CMOVCG A3,B7,1,jklm,5
```

Example 3 This example moves a double-precision floating-point value from TMS34082 coprocessor number 2. The 32 MSBs are stored in A0 and the 32 LSBs are stored in A1. This example assumes that the LOAD bit of the TMS34082 configuration register is set for MSBs transferred before LSBs.

The coprocessor command moves a double-precision (T = 1, size = 1) value from the coprocessor register (fpuop = 00111) RB2 (rd = 10010) to a TMS34020 register (md = 01).

```
MOV_RB2_20      .set    00124FH
                CMOVCG  A0,A1,1,MOV_RB2_20,2; size = 1 ID = 2
```



Syntax

CMOVCM **Rd+, count, size, command* [*, ID*]

Execution

ID, command → Coprocessor command

If size = 0, Repeat *count* times (1 ≤ *count* ≤ 32):
Coprocessor → **Rd*
Rd + 32 → *Rd*

If size = 1, Repeat *count* times (1 ≤ *count* ≤ 16):
Coprocessor → **Rd*
Rd + 32 → *Rd*
Coprocessor → **Rd*
Rd + 32 → *Rd*

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	1	R	Rd			
8 LSBs of coprocessor command								size	0	0	transfers				
coprocessor ID			13 MSBs of coprocessor command												

Description

This version of the CMOVCM instruction moves one or more values (depending on the value of *transfers*) from a coprocessor to memory. Before instruction execution, *Rd* contains the 32-bit address of the first location in memory; after each transfer, the value in *Rd* is incremented by 32 to point to the next address.The *size* operand is a value of 0 or 1; it determines the size of the values that are transferred:

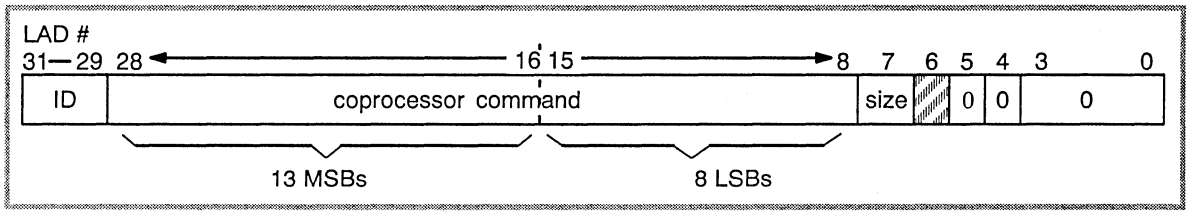
- If *size* = 0, then *count* specifies the number of **32-bit transfers** to make. In this case, *count* must be a value of 1 → 32.
- If *size* = 1, then *count* specifies the number of **64-bit transfers** to make. In this case, *count* must be a value of 1 → 16.

The value of *transfers* is set by the assembler, according to the values of *size* and *count*:

- If *size* = 0 and *count* = 1 → 31, then *transfers* = *count*
  If *size* = 0 and *count* = 32, then *transfers* = 0
- If *size* = 1 and *count* = 1 → 15, then *transfers* = 2x *count*
  If *size* = 1 and *count* = 16, then *transfers* = 0

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the source data for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

5 + [transfers –1] if the immediate data is long-word aligned
6 + [transfers –1] if the immediate data is not long-word aligned
(assumes that there are no 32-bit transfers)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This example moves thirty-two 32-bit values, specified by the coprocessor instruction *wxyz*, to memory from coprocessor 2 (register A0 contains the address of the first memory location).

```
CMOVCM *A0+,32,0,wxyz,2
```

Example 2

This example moves eight 64-bit values, specified by the coprocessor instruction *qrst*, to memory from the default coprocessor (register B7 contains the address of the first memory location).

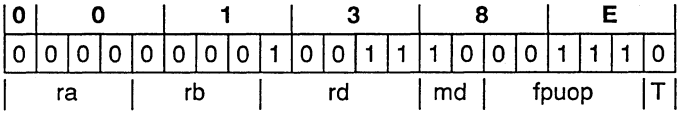
```
CMOVCM *B7+,8,1,qrst
```

Example 3

This example moves 5 (count = 5) 32-bit integers from coprocessor 1 (ID = 1) to the postincremented memory block pointed to by A3 . The coprocessor command specifies a TMS34082 Move To Host Memory (fpuop = 00111, md = 10) of integer (size = 0, T = 0) quantities starting in coprocessor register RB3 (rd = 10011).

The memory location pointed to originally by A3 will receive the first 32-bit integer transferred (from coprocessor register RB3). Memory location A3+10h will receive the second integer (from RB4). The third through fifth integers will be placed into A3+20h, A3+30h, and A3+40h (from RB5, RB6, and RB7 respectively).

```
MOVE5_RB3   .set     00138Eh
            CMOVCM   *A3+,5,0,MOVE5_RB3,1; count=5 size=0 ID=1
```



Syntax

CMOVCM *–*Rd, count, size, command* [*, ID*]

Execution

ID, command → Coprocessor command

If size= 0, Repeat *count* times (1 ≤ *count* ≤ 32):
Rd – 32 → *Rd*
Coprocessor → **Rd*

If size = 1, Repeat *count* times (1 ≤ *count* ≤ 16):
Rd – 32 → *Rd*
Coprocessor → **Rd*
Rd – 32 → *Rd*
Coprocessor → **Rd*

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	0	R	Rd			
8 LSBs of coprocessor instruction								size	0	0	transfers				
coprocessor ID			13 MSBs of coprocessor instruction												

Description

This version of the CMOVCM instruction moves one or more values (depending on the value of *transfers*) from a coprocessor to memory. Before instruction execution, *Rd* contains the 32-bit address of the first location in memory; before each transfer, the value in *Rd* is decremented by 32 to point to the next address.The *size* operand is a value of 0 or 1; it determines the size of the values that are transferred:

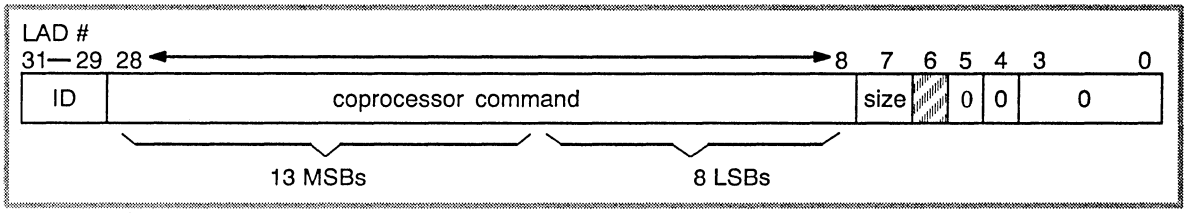
- If *size* = 0, then *count* specifies the number of **32-bit transfers** to make. In this case, *count* must be a value of 1 → 32.
- If *size* = 1, then *count* specifies the number of **64-bit transfers** to make. In this case, *count* must be a value of 1 → 16.

The value of *transfers* is set by the assembler according to the values of *size* and *count*:

- If *size* = 0 and *count* = 1 → 31, then *transfers* = *count*
  If *size* = 0 and *count* = 32, then *transfers* = 0
- If *size* = 1 and *count* = 1 → 15, then *transfers* = 2x *count*
  If *size* = 1 and *count* = 16, then *transfers* = 0

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the source data for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

5 + [transfers –1] if the immediate data is long-word aligned
6 + [transfers –1] if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This example moves nineteen 32-bit values, specified by coprocessor instruction *wxyz*, from coprocessor 3 to memory (register A3–32 specifies the first memory address).

```
CMOVCM –*A3,19,0,wxyz,3
```

Example 2

This example moves two 64-bit values, specified by coprocessor instruction *qrst*, from the default coprocessor to memory (register A3–32 specifies the first memory address).

```
CMOVCM –*A0,2,1,qrst
```

Example 3

This example moves eight 64-bit values, specified by coprocessor instruction *wxyz*, from coprocessor 3 to memory (register B7–32 specifies the first memory address).

```
CMOVCM –*B7,8,1,wxyz,3
```

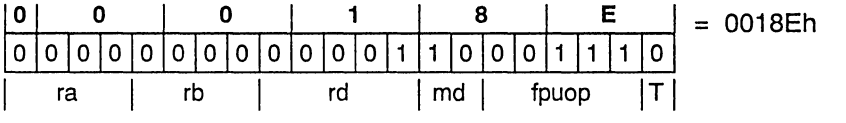
Example 4

This example moves 5 (count = 5) 32-bit quantities (size = 0) from the default TMS34082 coprocessor to the predecremented memory block pointed to by A1. The coprocessor command specifies a Move to Host Memory (fpuop = 00111, md = 10) of integer quantities (size = 0, T = 0), with the starting source coprocessor register RA1 (rd = 00001).

The first integer is transferred from the default coprocessor's RA1 register, the second integer number from RA2, the third from RA3, the fourth from RA4, and the last from RA5.

The memory location pointed to by A1–10h is the destination for the 32 bits of the first integer to be transferred. Memory location A1–20h is the destination for the second integer, with the remaining destination addresses as A1–30h, A1–40h, and A1–50h. After the transfer, A1 will point to the last integer transferred.

```
MOVE_RA1_20 .set    00018Eh
            CMOVCM  *–A1,5,0,MOVE_RA1_20;count = 5 size = 0
                                         ;ID = default
```



= 0018Eh

Syntax

CMOVCS *command* [, *ID*]

Execution

ID, *command* → Coprocessor command
Coprocessor → ST

Instruction Words

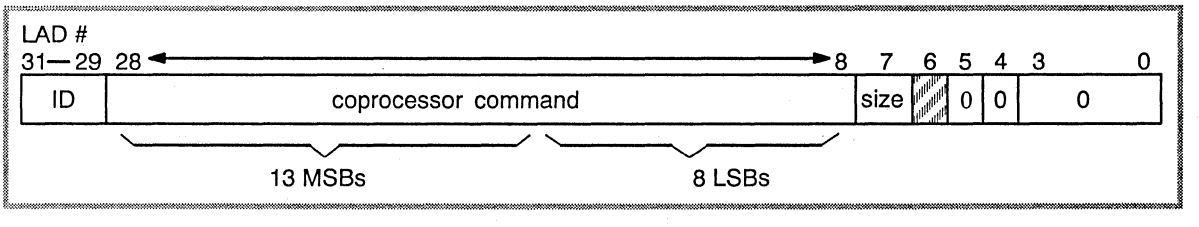
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0
8 LSBs of coprocessor command								0	0	0	0	0	0	0	1
coprocessor ID				13 MSBs of coprocessor command											

Description

CMOVCS moves one value from a coprocessor to the TMS34020 status register. (Note that this is a special coding of the CMOVCG instruction.) The value is masked by the TMS34020 so that its 4 MSBs are placed in the status register N, C, Z, and V bits. The remaining bits from the coprocessor are ignored by the TMS34020.

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the source data for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

4 if the immediate data is long-word aligned
5 if the immediate data is not long-word aligned

Status Bits

N Set to bit 31 from the coprocessor data.
C Set to bit 30 from the coprocessor data.
Z Set to bit 29 from the coprocessor data.
V Set to bit 28 from the coprocessor data.

Example 1

This example moves data from coprocessor 2 to the status register after issuing the coprocessor instruction *wxyz*.

```
CMOVCS wxyz,2
```

Example 2

This example moves data from the default coprocessor to the status register after issuing the coprocessor instruction *qrst*.

```
CMOVCS qrst
```

Syntax

CMOVGC *Rs*, *command* [, *ID*]

Execution

ID, *command* → Coprocessor command
Rs → Coprocessor

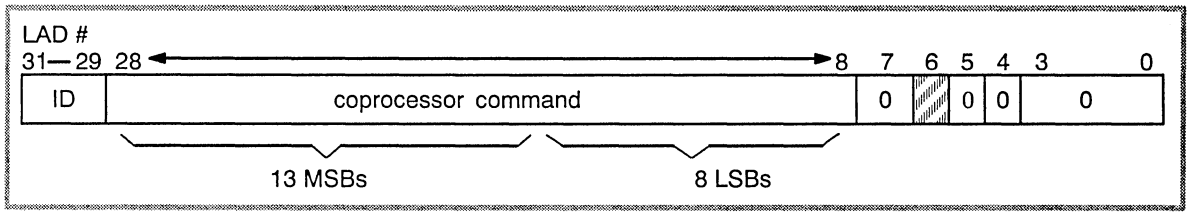
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
8 LSBs of coprocessor command								0	0	0	0	0	0	0	0
coprocessor ID					13 MSBs of coprocessor command										

Description

This version of the CMOVGC instruction moves the contents of a TMS34020 register to a coprocessor. The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the destination for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

2 (1) if the immediate data is long-word aligned
3 (1) if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This example moves 32 bits to coprocessor 2 (the destination is specified by the coprocessor instruction *wxyz*) from register A0.

```
CMOVGC A0,wxyz,2
```

Example 2

This example moves 32 bits to the default coprocessor (the destination is specified by the coprocessor instruction *qrst*) from register B7.

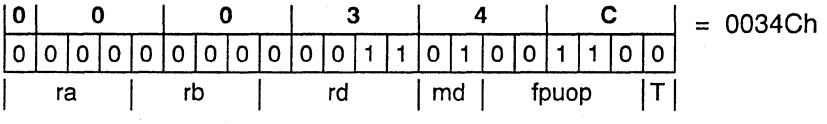
```
CMOVGC B7,qrst
```

Example 3

This example loads a TMS34082 register with a 32-bit integer from a TMS34020 register. The source register is B5 and the destination register is in the default coprocessor.

The coprocessor command loads (fpuop = 00110) the 32-bit integer (T = 0, size = 0), sent from the TMS34020 register (md = 01), into register RA3 (rd = 00011).

```
MOVE_R3   .set    00034CH
          CMOVGC B5,MOVE_RA3   ; ID = default
```



= 0034Ch

Syntax

CMOVGC *Rs_1*, *Rs_2*, *size*, *command* [, *ID*]

Execution

ID, *command* → Coprocessor command
Rs_1, *Rs_2* → Coprocessor

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
8 LSBs of coprocessor command								size	0	0	R	Rs_2			
coprocessor ID			13 MSBs of coprocessor command												

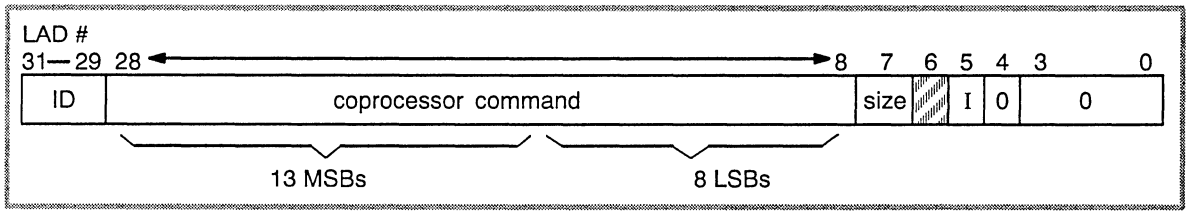
Description

This version of the CMOVGC instruction moves the contents of two TMS34020 registers to a coprocessor.The *size* operand is a value of 0 or 1; the coprocessor interprets the *size* bit to determine how to move the data:

- If *size* = 0, then the values that are moved are two separate **32-bit values**.
- If *size* = 1, then the values that are moved are two halves of a single **64-bit value**.

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the destination for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



I is the coprocessor parameter index bit. For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

3 (1) if the immediate data is long-word aligned
4 (1) if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This instruction moves two 32-bit values to coprocessor 2 (the destination is specified by the coprocessor instruction *wxyz*)from registers A0 and B3.

```
CMOVGC A0,B3,0,wxyz,2
```

Example 2

This instruction moves one 64-bit value to the default coprocessor (the destination is specified by the coprocessor instruction *qrst*)from registers B7 and B8.

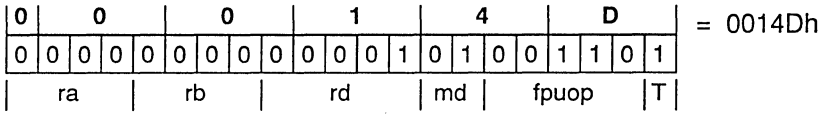
```
CMOVGC B7,B8,1,qrst
```

Example 3

This example moves two 32-bit floating-point values (in registers B3 and B4) into a TMS34082 coprocessor number 0 register.

The coprocessor command loads (fpuop = 00110) the 32-bit floating point (T = 1, size = 0) data from the TMS34020 registers (md=01) into RA1 (rd = 00001) and RA2.

```
MOVEF_RA1     .set   00014DH
              CMOVGC B3,B4,0,MOVEF_RA1,0 ; size = 0 ID = 0
```



= 0014Dh

Syntax

CMOVMC **Rs+, count, size, command [, ID]*

Execution

ID, command → Coprocessor command

If size= 0, Repeat *count* times (1 ≤ *count* ≤ 32):
**Rs* → Coprocessor
Rs + 32 → *Rd*

If size = 1, Repeat *count* times (1 ≤ *count* ≤ 16):
**Rs* → Coprocessor
Rs + 32 → *Rd*
**Rs* → Coprocessor
Rs + 32 → *Rd*

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	transfers				
8 LSBs of coprocessor command								size	0	0	R	Rs			
coprocessor ID			13 MSBs of coprocessor command												

Description

This version of the CMOVMC instruction moves one or more values (depending on the value of *transfers*) from memory pointed to by *Rs* to the coprocessor designated by the coprocessor instruction. Before instruction execution, *Rs* contains the 32-bit memory address of a 32-bit value; after each transfer, the value in *Rs* is incremented by 32 bits to point to the next address. The *size* operand is a value of 0 or 1; it determines the size of the values that are transferred:

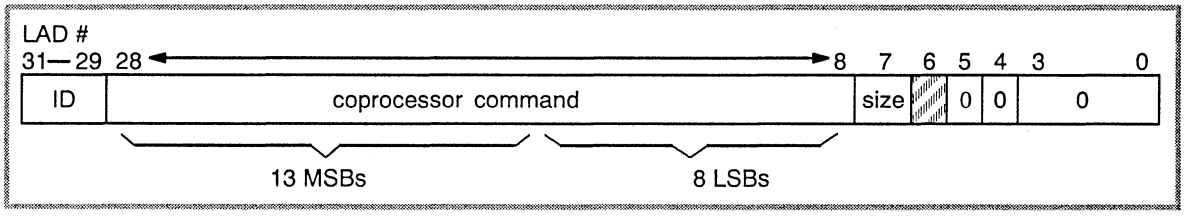
- If *size* = 0, then *count* specifies the number of **32-bit transfers** to make. In this case, *count* must be a value of 1 → 32.
- If *size* = 1, then *count* specifies the number of **64-bit transfers** to make. In this case, *count* must be a value of 1 → 16.

The value of *transfers* is set by the assembler according to the values of *size* and *count*:

- If *size* = 0 and *count* = 1 → 31, then *transfers* = *count*
  If *size* = 0 and *count* = 32, then *transfers* = 0
- If *size* = 1 and *count* = 1 → 15, then *transfers* = 2x *count*
  If *size* = 1 and *count* = 16, then *transfers* = 0

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the destination for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

5 + [transfers –1] if the immediate data is long-word aligned
6 + [transfers –1] if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This instruction moves thirty-two 32-bit values to coprocessor 2 (the destination is specified by the coprocessor instruction *wxyz*)from memory (register A0 specifies the first memory address).

```
CMOVMC *A0+,32,0,wxyz,2
```

Example 2

This instruction moves eight 64-bit values to the default coprocessor (the destination is specified by the coprocessor instruction *wxyz*) from memory (register B7 specifies the first memory address).

```
CMOVMC *B7+,8,1,wxyz
```

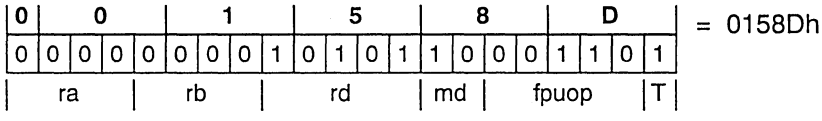
Example 3

This example moves 3 (count = 3) 64-bit floating-point quantities (size = 1, T = 1) from the memory block pointed to by A3 to coprocessor 1 (ID = 1). The coprocessor command specifies a TMS34082 *move to coprocessor registers* (fpuop = 00110) from TMS34020 memory (md = 10) with the starting destination of RB5 (rd = 10101). This example assumes that the LOAD bit of the TMS34082 configuration register is set to transfer the MSBs of the double values before the LSBs.

The memory location pointed to by A3 should contain the 32 MSBs of the first double number to be transferred. Memory location A3+10h should contain the 32 LSBs of the first double number. The MSBs of the second number are found at A3+20h, followed by its LSBs at A3+30h, and then the MSBs and LSBs of the third double number.

The first 64-bit single-precision number is transferred to coprocessor one register RB5, the second double number to RB6 and the third to RB7.

```
MOVD3_RB5 equ     00158Dh
          CMOVMC *A3+,3,1,MOVD3_RB5,1; count = 3 size = 1
                                     ; ID = 1
```



= 0158Dh

Syntax

CMOVMC *–*Rs count, size, command [, ID]*

Execution

ID, command → Coprocessor command

If size = 0, Repeat *count* times (1 ≤ *count* ≤ 32):
Rs – 32 → *Rs*
**Rs* → Coprocessor

If size = 1, Repeat *count* times (1 ≤ *count* ≤ 16):
Rs – 32 → *Rs*
**Rs* → Coprocessor
Rs – 32 → *Rs*
**Rs* → Coprocessor

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	transfers				
8 LSBs of coprocessor command								size	0	0	R	Rs			
coprocessor ID			13 MSBs of coprocessor command												

Description

This version of the CMOVMC instruction moves one or more values (depending on the value of *transfers*) from memory pointed to by *Rs* to the coprocessor designated by the coprocessor instruction. Before the first transfer, *Rs* is decremented to contain the 32-bit memory address of the first 32-bit value to transfer. The *size* operand is a value of 0 or 1; it determines the size of the values that are transferred:

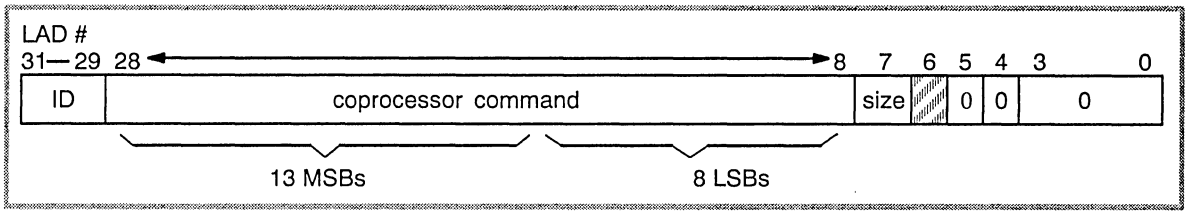
- If *size* = 0, then *count* specifies the number of **32-bit transfers** to make. In this case, *count* must be a value of 1 → 32.
- If *size* = 1, then *count* specifies the number of **64-bit transfers** to make. In this case, *count* must be a value of 1 → 16.

The value of *transfers* is set by the assembler according to the values of *size* and *count*:

- If *size* = 0 and *count* = 1 → 31, then *transfers* = *count*
  If *size* = 0 and *count* = 32, then *transfers* = 0
- If *size* = 1 and *count* = 1 → 15, then *transfers* = 2x *count*
  If *size* = 1 and *count* = 16, then *transfers* = 0

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the destination for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified in the coprocessor directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

5 + [transfers –1] if the immediate data is long-word aligned
6 + [transfers –1] if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This instruction moves thirty-two 32-bit values to coprocessor 2 (the destination is specified by the coprocessor instruction *wxyz*) from memory (register A0 specifies the first memory address).

```
CMOVMC –*A0,32,0,wxyz,2
```

Example 2

This instruction moves eight 64-bit values to the default coprocessor (the destination is specified by the coprocessor instruction *wxyz*) from memory (register B7 specifies the first memory address).

```
CMOVMC –*B7+,8,1,wxyz
```

Example 3

This example demonstrates the predecrement transfer of data to the TMS34082 using the typical C subroutine calling convention. Assume that the routine _COPROC_SUB requires two 32-bit integer quantities and 3 double quantities, all passed on the program stack.The result is a double value, returned on the program stack. Assume that field size 1 is 32 bits (FS1 = 32), and that the LOAD bit of the configuration register is set to 0 so the MSBs are transferred before LSBs.

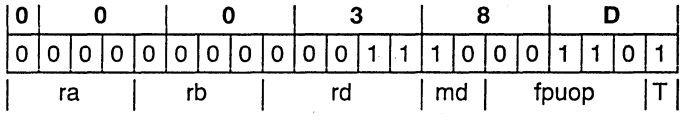
_COPROC_SUB moves the values from the stack to the coprocessor using the CMOVMC opcode in predecrement mode. It first moves the 3 double values to the coprocessor. The command MC_3_DOUB performs a move to coprocessor registers (fpuop = 00110) of 3 (count = 3) double precision values (T = 1, size = 1) from the memory pointed to (md = 10) by the predecremented value of STK, to coprocessor registers starting at RA3 (rd = 00011).

Next, the two 32-bit integer values are moved to the coprocessor using the MC_2_INT command. It performs a move to coprocessor registers (fpuop = 00110) of 2 (count = 2) integer values (T = 0, size = 0) from the memory pointed to (md = 10) by the predecremented value of STK to coprocessor registers starting at RB1 (rd = 10001).

After manipulating the data in the coprocessor, the result is loaded from the coprocessor and placed onto the stack. This is performed by the CM_1_DOUB command of the CMOVCM opcode in predecrement mode. Predecrement mode is used to preserve the proper ordering of MSBs and LSBs on the stack. The stack pointer is kept pointing in the proper place by the 2 stack pointer adds. Interrupts are disabled when the stack pointer is not properly positioned. You may prefer to use other methods of ensuring program stack position integrity if interrupts are used for timing-critical functions.

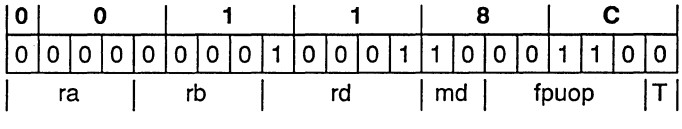
The CM_1_DOUB command performs a move of coprocessor registers (fpuop = 00111) to TMS34020 memory (md = 10) of one (count = 1) double-precision value (T = 1, size = 1) from coprocessor register RB5 (rd = 10101) to the predecremented memory location pointed to by STK.

```
                .GLOBAL _COPROC_SUB
MC_3_DOUB:      .set    00038Dh
```



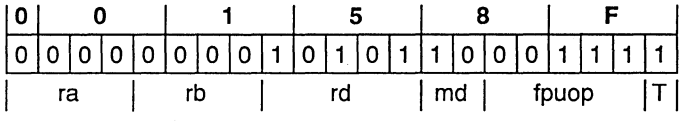
= 0038Dh

```
MC_2_INT: .set  00118Ch
```



= 0118Ch

```
CM_1_DOUB:      .set 00158Fh
```



= 0158Fh

```
MOVE    INT_2,*STK+,1
MOVE    INT_1,*STK+,1
MOVE    DOUB_3_LO,*STK+,1
MOVE    DOUB_3_HI,*STK+,1
MOVE    DOUB_2_LO,*STK+,1
MOVE    DOUB_2_HI,*STK+,1
MOVE    DOUB_1_LO,*STK+,1
MOVE    DOUB_1_HI,*STK+,1
CALLA   _COPROC_SUB
MOVE    –*STK,A0,1        ;GET DOUBLE RESULT FROM STACK, HI
MOVE    –*STK,A1,1        ;LO
.
.
.
.
_COPROC_SUB:
MMTM    SP,A0,A1
CMOVCM –*STK,3,1,MC_3_DOUB
CMOVCM –*STK,2,0,MC_2_INT

.               ;OTHER COPROCESSOR COMMANDS
.
.
.
DINT
ADDI    40h,STK                 ;MAKE ROOM FOR DOUBLE VALUE
CMOVMC –*STK,1,1,CM_1_DOUB
ADDI    40h,STK                 ;POINT STACK TO CORRECT TOP
EINT
MMFM    SP,A0,A1
RETS
```

Syntax

CMOVMC **Rs+, Rd, size, command [, ID]*

Execution

ID, command → Coprocessor command

Repeat number of times specified by *Rd*
**Rs* → Coprocessor
Rs + 32 → *Rs*

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
8 LSBs of coprocessor command								size	0	0	R	Rs			
coprocessor ID				13 MSBs of coprocessor command											

Description

This version of the CMOVMC instruction moves one or more 32-bit values (depending on the value in *Rd*) from memory to the specified coprocessor register. Before instruction execution, *Rs* contains the 32-bit memory address of a 32-bit value; after each transfer, the value in *Rs* is incremented by 32 bits to point to the next address. The *size* operand is a value of 0 or 1; it determines the size of the values that are transferred:

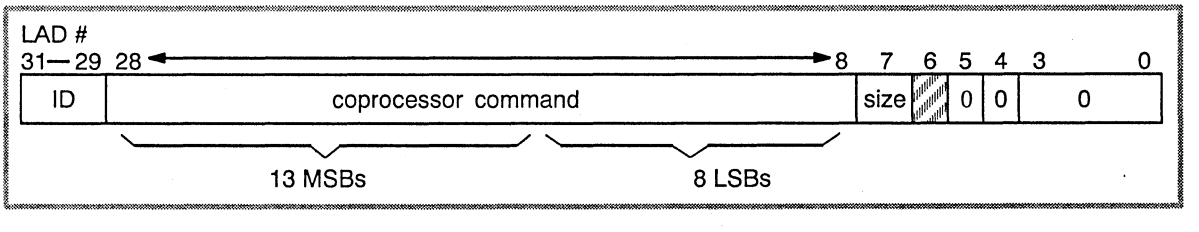
- If *size* = 0, then **32-bit** values are moved. In this case, *Rd* should specify the number of 32-bit values to move.
- If *size* = 1, then **64-bit values** are moved. In this case, *Rd* should specify **twice** the number of 64-bit values to move.

The value of *Rd (number of transfers)* is interpreted by the TMS34020 as follows:

- If *Rd* = 1 → 31, then the number of 32-bit transfers = *Rd*
- If *Rd* = *0* then the number of 32-bit transfers = 32

The *command* operand specifies an instruction (21 bits of information define the instruction) that the coprocessor should execute to specify the destination for the move. The *ID* operand is an optional 3-bit coprocessor identification code; if you don't supply this operand, it defaults to the value specified with the .coproc directive.

The output of this instruction on the LAD bus at $\overline{\text{ALTCH}}$ low during the command cycle (when SF is high) is as follows:



For more information, refer to Section 10.3, Formats of Commands Passed to a Coprocessor, on page 10-5.

Machine States

5 + [register value –1] if the immediate data is long-word aligned
6 + [register value –1] if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example 1

This instruction moves the number of 32-bit values specified by A2 to coprocessor 2 (the destination is specified by the coprocessor instruction *wxyz*) from memory (register A0 contains the first memory address).

```
CMOVMC *A0+,A2,0,wxyz,2
```

Example 2

This instruction moves the number of 64-bit values specified by B7÷2 to the default coprocessor (the destination is specified by the coprocessor instruction *wxyz*) from memory (register A3 contains the first memory address).

```
CMOVMC *A3+,B7,1,qrst
```

Example 3

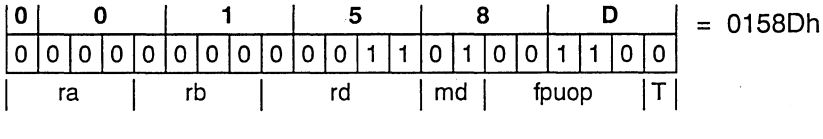
This example moves 64-bit quantities (size = 1, T = 1) from the postincremented memory block pointed to by A3 to TMS34082 coprocessor number 1 (ID = 1). Assume that the LOAD bit in the TMS34082 configuration register is set for MSBs transfer before LSBs transfer. Because register B4 specifies the number of 32-bit transfers to be performed, it should contain twice the number of 64-bit quantities to be transferred.

The coprocessor command specifies a TMS34082 *move to coprocessor registers* (fpuop = 00110) from TMS34020 memory (md = 10) of double quantities (T = 1, size = 1), with the starting destination of RB5 (rd = 10101).

The memory location pointed to originally by A3 should contain the 32 MSBs of the first double number to be transferred. Memory location A3+10h should contain the 32 LSBs of the first double number. The remaining data should be in the MSBs before LSBs form.

The two 32-bit halves of the first double number will be transferred to coprocessor one register RB5, the second double number to RB6, etc.

```
MOVD_RB5  .set    00158Dh
          CMOVMC  *A3+,B4,1,MOVD_RB5,1   ; count = 3 size = 1
                                         : ID = 1
```



= 0158Dh

Syntax

CMP *Rs, Rd*

Execution

Set status bits on the result of Rd – Rs

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	Rs				R	Rd			

Description

CMP sets the status bits on the result of subtracting the contents of Rs from the contents of Rd. This is a nondestructive compare; the contents of the registers are not affected. This instruction is often used in conjunction with the JA*cc* or JR*cc* conditional jump instructions.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if a there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before A1	Before A0	After N C Z V	Jumps Taken
`CMP A1,A0`	00000001h	00000001h	0 0 1 0	UC,NN,NC,Z,NV,LS,GE,LE,HS
`CMP A1,A0`	00000001h	00000002h	0 0 0 0	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
`CMP A1,A0`	00000001h	FFFFFFFFh	1 0 0 0	UC,N,NC,NZ,NV,P,HI,LT,LE,HS
`CMP A1,A0`	00000001h	80000000h	0 0 0 1	UC,NN,NC,NZ,V,HI,LT,LE,HS
`CMP A1,A0`	FFFFFFFFh	7FFFFFFFh	1 1 0 1	UC,N,C,NZ,V,LS,GE,GT,LO
`CMP A1,A0`	FFFFFFFFh	80000000h	1 1 0 0	UC,N,C,NZ,NV,LS,LT,LE,LO
`CMP A1,A0`	80000000h	7FFFFFFFh	1 1 0 1	UC,N,C,NZ,V,LS,GE,GT,LO

Syntax

CMPI *IW*, *Rd* [, **W**]

Execution

Set status bits on the result of Rd – 16-bit immediate value

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	1	0	R	Rd			
1s complement of *IW*															

Description

CMPI sets the status bits on the result of subtracting a 16-bit, sign-extended immediate value from the contents of the destination register. (The symbol *IW* in the syntax above represents a 16-bit, signed immediate value.) This is a nondestructive compare; the contents of the destination register are not affected. This instruction is often used in conjunction with the JA*cc* or JR*cc* conditional jump instructions.

Note that the assembler inserts the 1s complement of the 16-bit value into the second instruction word.

The assembler uses the short form of the CMPI instruction if the immediate value is previously defined and is in the range –32,768 to 32,767. You can force the assembler to use the short form by following the register operand with **W:**

```
CMPI  IW,Rd,W
```

The assembler truncates the upper bits and issues an appropriate warning message if the value is greater than 16 bits.

Machine States

2

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before A0	After N C Z V	Jumps Taken
`CMPI 1,A0`	00000002h	0 0 0 0	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
`CMPI 1,A0`	00000001h	0 0 1 0	UC,NN,NC,Z,NV,LS,GE,LE,HS
`CMPI 1,A0`	00000000h	1 1 0 0	UC,N,C,NZ,NV,LS,LT,LE,LO
`CMPI 1,A0`	FFFFFFFFh	1 0 0 0	UC,N,NC,NZ,NV,P,HI,LT,LE,HS
`CMPI 1,A0`	80000000h	0 0 0 1	UC,NN,NC,NZ,V,HI,LT,LE,HS
`CMPI –2,A0`	00000000h	0 1 0 0	UC,NN,C,NZ,NV,P,LS,GE,GT,LO
`CMPI –2,A0`	FFFFFFFFh	0 0 0 0	UC,NN,NC,NZ,NV,P,LI,GE,GT,HS
`CMPI –2,A0`	FFFFFFFEh	0 0 1 0	UC,NN,NC,Z,NV,LS,GE,LE,HS
`CMPI –2,A0`	FFFFFFFDh	1 1 0 0	UC,N,C,NZ,NV,LS,LT,LE,LO
`CMPI –1,A0`	7FFFFFFFh	1 1 0 1	UC,N,C,NZ,V,LS,GE,GT,LO

Syntax

CMPI *IL*, *Rd* [, **L**]

Execution

Set status bits on the result of Rd – 32-bit immediate value

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	1	1	R	Rd			
1s complement of 16 LSBs of *IL*															
1s complement of 16 MSBs of *IL*															

Description

CMPI sets the status bits on the result of subtracting a 32-bit, signed immediate value from the contents of the destination register. (The *IL* symbol in the syntax above represents a 32-bit, signed immediate value.) This is a nondestructive compare; the contents of the destination register are not affected.

Note that the assembler inserts the 1s complement of the 16 LSBs of the value into the second instruction word and inserts the 1s complement of the 16 MSBs of the value into the third instruction word.

The assembler uses this form of the CMPI instruction if it cannot use the short form. You can force the assembler to use the long form by following the register operand with an **L:**

```
CMPI  IL, Rd, L
```

This instruction is often used in conjunction with the JA*cc* or JR*cc* conditional jump instructions.

Machine States

2 if the immediate data is long-word aligned
3 if the immediate data is not long-word aligned

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code		Before	After	Jumps Taken
		A0	N C Z V	
CMPI	8000h,A0	00008001h	0 0 0 0	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
CMPI	8000h,A0	00008000h	0 0 1 0	UC,NN,NC,Z,NV,LS,GE,LE,HS
CMPI	8000h,A0	00007FFFh	1 1 0 0	UC,N,C,NZ,NV,LS,LT,LE,LO
CMPI	8000h,A0	FFFFFFFFh	1 0 0 0	UC,N,NC,NZ,NV,P,HI,LT,LE,HS
CMPI	8000h,A0	80007FFFh	0 0 0 1	UC,NN,NC,NZ,V,HI,LT,LE,HS
CMPI	0FFFF7FFFh,A0	00000000h	0 1 0 0	UC,NN,C,NZ,NV,P,LS,GE,GT,LO
CMPI	0FFFF7FFEh,A0	FFFF7FFFh	0 0 0 0	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
CMPI	0FFFF7FFEh,A0	FFFF7FFEh	0 0 1 0	UC,NN,NC,Z,NV,LS,GE,LE,HS
CMPI	0FFFF7FFEh,A0	FFFF7FFDh	1 1 0 0	UC,N,C,NZ,NV,LS,LT,LE,LO
CMPI	0FFFF7FFFh,A0	7FFF7FFFh	1 1 0 1	UC,N,C,NZ,V,LS,GE,GT,LO

Syntax

CMPK *constant, Rd*

Execution

Rd – constant → Status

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	constant					R	Rd			

Description

CMPK subtracts the 5-bit constant from the contents of the destination register (Rd) and sets status on the result of the subtraction. The constant is an unsigned number in the range 1—32.

The assembler automatically builds the 5 LSBs into the opcode. Note that constant=0 in the opcode corresponds to the value 32; the assembler converts the value 32 to 0. Using this instruction, the assembler issues an error if you try to compare 0 with a register.

This instruction does not alter the contents of Rd.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before A0	After N C Z V	Jumps Taken
`cmpk 1,A0`	`00000002h`	0 0 0 0	UC,NN,NC,NZ,NV,P,HI,GE,GT,HS
`cmpk 2,A0`	`00000002h`	0 0 1 0	UC,NN,NC,Z,NV,LS,GE,LE,HS
`cmpk 32,A0`	`00000000h`	1 1 0 0	UC,N,C,NZ,NV,LS,LT,LE,LO
`cmpk 16,A0`	`FFFFFFFFh`	1 0 0 0	UC,N,NC,NZ,NV,P,HI,LT,LE,HS
`cmpk 1,A0`	`80000000h`	0 0 0 1	UC,NN,NC,NZ,V,HI,LT,LE,HS

Syntax

CMPXY *Rs, Rd*

Execution

Set status bits on the results of
X half of Rd – X half of Rs
Y half of Rd – Y half of Rs

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	Rs				R	Rd			

Description

CMPXY compares the source register to the destination register in XY mode and sets the status bits as if a subtraction had been performed. This is a nondestructive compare; the contents of the register are not affected. The source and destination registers are treated as signed XY registers. Note that no overflow detection is provided.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if source X field = destination X field, *0* otherwise
C Sign bit of Y half of the result
Z *1* if source Y field = destination Y field, *0* otherwise
V Sign bit of X half of the result

Examples

Code	Before A1	Before A0	After N C Z V	Jumps Taken
`CMPXY A1,A0`	00090009h	00010001h	0 1 0 1	NN,C,NZ,V,LS,LT
`CMPXY A1,A0`	00090009h	00090001h	0 0 1 1	NN,NC,Z,V,LS,LT
`CMPXY A1,A0`	00090009h	00010009h	1 1 0 0	N,C,NZ,NV,LS,LT
`CMPXY A1,A0`	00090009h	00090009h	1 0 1 0	N,NC,Z,NV,LS,LT
`CMPXY A1,A0`	00090009h	00000010h	0 1 0 0	NN,C,NZ,NV,LS,GE
`CMPXY A1,A0`	00090009h	00090010h	0 0 1 0	NN,NC,Z,NV,LS,GE
`CMPXY A1,A0`	00090009h	00100000h	0 0 0 1	NN,NC,NZ,V,HI,LT
`CMPXY A1,A0`	00090009h	00100009h	1 0 0 0	N,NC,NZ,NV,HI,LT
`CMPXY A1,A0`	00090009h	00100010h	0 0 0 0	NN,NC,NZ,NV,HI,GE

Syntax

CPW *Rs, Rd*

Execution

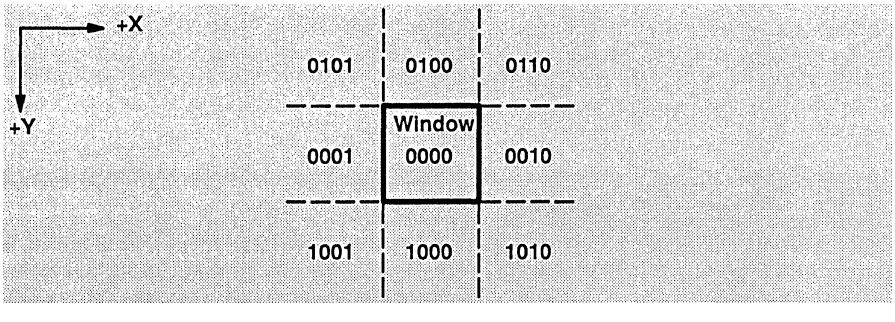
Point code → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	Rs				R	Rd			

Description

CPW compares a point represented by an XY value in the source register to the window limits in the WSTART and WEND registers and places a 4-bit *outcode* in bits 5 through 8 of the destination register. The remaining 28 bits of the destination register are set to *0*. The contents of the source register are treated as an XY address that consists of 16-bit signed X and Y values. WSTART and WEND are also treated as signed XY-format registers. WSTART and WEND can contain signed values. The location of the point with respect to the window is encoded as shown below; the outcode is loaded into the destination register.



The following list describes the contents of the destination register after CPW execution:

Bit Position: Contents:

0—4 0s
5 1 if X half of WSTART > X half of Rs, 0 otherwise
6 1 if X half of Rs > X half of WEND, 0 otherwise
7 1 if Y half of WSTART > Y half of Rs, 0 otherwise
8 1 if Y half of Rs > Y half of WEND, 0 otherwise
9—31 0s

This instruction can also be used to trivially reject lines that do not intersect with a window. If the CPW codes for the 2 points defining the line are ANDed together and the result is nonzero, then the line must lie completely outside the window (and does not intersect it). A *0* result indicates that the line *may* intersect the window, and a more rigorous test must be applied. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Rs and Rd must be in the same register file.

Implied Operands

Register	Name	Format	Description
B5	WSTART	XY	Window start. Defines inclusive starting corner of window (lesser value corner).
B6	WEND	XY	Window end. Defines inclusive ending corner of window (greater value corner).

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if point lies outside window, *0* otherwise

Examples

You must select appropriate implied operand values before executing the CPW instruction. In this example, the implied operands are set up as follows, specifying a block of 36 pixels.

WSTART = 5,5
WEND = A,A

```
CPW A1,A0
```

Before		After	
A1	**N C Z V**	**A0**	**N C Z V**
00040004h	x x x 0	000000A0h	x x x 1
00040005h	x x x 0	00000080h	x x x 1
0004000Ah	x x x 0	00000080h	x x x 1
0004000Bh	x x x 1	000000C0h	x x x 1
00050004h	x x x 1	00000020h	x x x 1
00050005h	x x x 0	00000000h	x x x 0
0005000Ah	x x x 0	00000000h	x x x 0
0005000Bh	x x x 0	00000040h	x x x 1
000A0004h	x x x 0	00000020h	x x x 1
000A0005h	x x x 1	00000000h	x x x 0
000A000Ah	x x x 1	00000000h	x x x 0
000A000Bh	x x x 0	00000040h	x x x 1
000B0004h	x x x 0	00000120h	x x x 1
000B0005h	x x x 0	00000100h	x x x 1
000B000Ah	x x x 0	00000100h	x x x 1
000B000Bh	x x x 0	00000140h	x x x 1

Syntax

CVDXYL *Rd*

Execution

(Y half of Rd × DPTCH) + (X half of Rd × PSIZE) + (A4 or B4) → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	1	0	0	R	Rd			

Description

CVDXYL converts the XY value contained in the destination register to a linear value, by using the destination pitch, and writes the result back to the destination register.

If Rd is an A file register, then A4 is used as the offset; if Rd is a B file register, then B4 is used as the offset. CONVDP (based on DPTCH) is used to effect the multiply of the Y half by the pitch. PSIZE provides the pixel size used to multiply by the X half (done by shift).

Use the SETCDP instruction (page 13-227) to set up CONVDP. For more information, refer to Section 12.12, Converting an XY Address to a Linear Address, on page 12-47.

Implied Operands

Address	Name	Description and Elements (Bits)
A4 or B4	OFFSET	linear screen origin (0,0)
C0000140	CONVDP	XY-to-linear conversion (destination)
C0000150	PSIZE	Pixel size (1,2,4,8,16,32)

Machine States

pitch is	a power of 2:	2
	2 powers of 2:	3
	arbitrary	14

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

The examples assume the following definitions:

```
DADDR    .set  B2       ; Destination address register
DPTCH    .set  B3       ; Destination pitch register
OFFSET   .set  B4       ; XY offset register
TEMP     .set  B14      ; Temporary register
```

Example 1

Screen dimensions of 640 by 480 by 4 bits-per-pixel; screen pitch of 4096 (the smallest power of 2 greater than 640 × 4).

```
; the following code will typically be done once
: at initialization time
        setf   16,0,0           ; set field size 0 to 16
        movk   4,TEMP
        exgps  TEMP             ; set the pixel size to 4
        movi   01000h,DPTCH     ; set up display pitch
        movi   0100h,OFFSET     ; set up the screen offset
        setcdp          ; set CONVDP to 0013h (LMO of DPTCH)
        mwait
; set up XY address in DADDR and convert to linear
        movi   00010001h,DADDR
        cvdxyl DADDR            ; This sets DADDR to 1104h
```

Example 2

Screen dimensions of 640 by 480 by 8 bits-per-pixel; screen pitch of 5120 (640 × 8) which is a sum of 2 powers of 2.

```
; the following code will typically be done once
: at initialization time
        setf   16,0,0           ; set field size 0 to 16
        movk   8,TEMP
        exgps                   ; pixel size of 8
        movi   01400h,DPTCH     ; set up display pitch
        clr    OFFSET           ; set up the screen offset
        setcdp                  ; set CONVDP to 1513h
                                ; (2 LMOs of DPTCH)
        mwait
; set up XY address in DADDR and convert to linear
        movi   00010001h,DADDR
        cvdxyl DADDR            ; This sets DADDR to 1408h
```

Example 3

Perform an XY to linear conversion of a nonscreen bitmap with an arbitrary pitch of 224.

```
; the following code will typically be done once at
; initialization time
        setf   16,0,0           ; set field size 0 to 16
        movk   16,TEMP          ;
        exgps                   ; pixel size of 16
; set up the value of DPTCH, CONVDP, have user variables
; for offset and address
        movi   0E0h,DPTCH
        setcdp                  ; set CONVDP to 0000h (which
                                ; indicates arbitrary pitch)
        mwait
        movi   0FF300000h,A4    ; offset points to an area
                                ; of DRAM
        movi   00010001h,A7
        cvdxyl A7               ; this sets A7 to FF3000F0h
```

Syntax

CVMXYL *Rd*

Execution

(Y half of Rd × MPTCH) + (X half of Rd) → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	R	Rd			

Description

CVMXYL converts the XY value contained in the destination register to a linear value, by using the mask pitch, and writes the result back to the destination register.

Note that *no* offset is added when you use CVMXYL. CONVMP (based on MPTCH) is used to effect the multiply of the Y half by the pitch.

Use the SETCMP instruction (page 13-228) to set up CONVMP. For more information, refer to Section 12.12, Converting an XY Address to a Linear Address, on page 12-47.

Implied Operands

Address	Name	Description and Elements (Bits)
C0000150	PSIZE	Pixel size (1,2,4,8,16,32)
C0000180	CONVMP	XY-to-linear conversion (destination)

Machine States

pitch is	a power of 2:	2
	2 powers of 2:	3
	arbitrary	14

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

The examples assume the following definitions:

```
MADDR     .set  B10              ; Mask address register
MPTCH     .set  B11              ; Mask pitch register
TEMP      .set  B14              ; Temporary register
```

Example 1

Screen dimensions of 640 by 480 by 4 bits-per-pixel; screen pitch of 4096 (the smallest power of 2 greater than 640 × 4)

```
; the following code will typically be done once
; at initialization time
       setf   16,0,0          ; set field size 0 to 16
       movi   01000h,MPTCH    ; set up mask pitch
       setcmp                 ; set CONVMP to 0013h (LMO
                              ; of MPTCH)
       mwait
; set up XY address in MADDR and convert to linear
       movi   00010001h,MADDR
       cvmxyl MADDR           ; This sets MADDR to 1001h
```

Example 2

Screen dimensions of 640 by 480 by 8 bits-per-pixel; screen pitch of 5120 (640 × 8) which is a sum of 2 powers of 2.

```
;  the following code will typically be done once
;  at initialization time
        setf    16,0,0          ; set field size 0 to 16
        movi    01400h,MPTCH    ; set up mask pitch
        setcmp                  ; set CONVMP to 1513h
                                ; (2 LMOs of MPTCH)
        mwait
;  set up XY address in MADDR and convert to linear
        movi    00010001h,MADDR
        cvmxyl MADDR            ; This sets MADDR to 1401h
```

Example 3

Perform an XY to linear conversion of a nonscreen bitmap with an arbitrary pitch of 224.

```
;  the following code will typically be done once
;  at initialization time
        setf    16,0,0          ; set field size 0 to 16
        movi    0E0h,MPTCH      ; set up mask pitch
        setcmp                  ; set CONVMP to 0000h (which
                                ; indicates arbitrary pitch)
        mwait
; set up XY address in MADDR and convert to linear
        movi    00010001h,MADDR
        cvmxyl MADDR            ; This sets MADDR to E1h
```

Syntax

CVSXYL *Rs, Rd*

Execution

(Y half of Rd × SPTCH) + (X half of Rd × PSIZE) + Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	Rs				R	Rd			

Description

CVSXYL converts the XY value contained in the destination register to a linear value, by using the source pitch, and writes the result back to the destination register. The source register contains the offset used in the conversion process. CONVSP (based on SPTCH) is used to effect the multiply of the Y half by the pitch. PSIZE provides the pixel size used to multiply by the X half (done by shift).

Use the SETCSP instruction (page 13-229) to set up CONVSP. For more information, refer to Section 12.12, Converting an XY Address to a Linear Address, on page 12-47.

This instruction allows the programmer to have independent source and destination offsets. The source offset can be any register, and the destination offset is B4. As an example, assume B14 is the source offset. To synthesize PIXBLT XY,XY with independent offsets use CVSXYL B14,B0 to convert the source pointer to linear and follow with a PIXBLT L,XY.

Implied Operands

Address	Name	Description and Elements (Bits)
C0000130	CONVSP	XY-to-linear conversion (source)
C0000150	PSIZE	Pixel size (1,2,4,8,16,32)

Machine States

pitch is	a power of 2:	2
	2 powers of 2:	3
	arbitrary	14

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

```
CVSXYL A0, A1
```

Before	A0	= 00000100
	A1	= 00010001
	CONVSP	= 0013
	PSIZE	= 0008
After	A1	= 00001108

Syntax

CVXYL *Rs, Rd*

Execution

(Y half of Rs × DPTCH) + (X half of Rs × PSIZE) + offset → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	Rs				R	Rd			

Description

CVXYL converts an XY address to a linear address:

- The source register contains an XY address. The signed X value occupies the 16 LSBs of the register, and the signed Y value occupies the 16 MSBs. The X value must be positive.
- The XY address is converted into a 32-bit linear address, which is stored in the destination register. For more information, refer to Section 12.12, Converting an XY Address to a Linear Address, on page 12-47.

The offset value (see execution) is in the OFFSET register. The CONVDP value is used to determine the shift amount for the Y value, while the PSIZE register determines the X shift amount.

Use the SETCDP instruction (page 13-227) to set up CONVDP.

Rs and Rd must be in the same register file.

Implied Operands

Address	Name	Description and Elements (Bits)
B3	DPTCH (linear)	Destination pitch
B4	OFFSET (linear)	Screen origin (location 0,0)
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)

Machine States

pitch is	a power of 2:	3
	2 powers of 2:	4
	arbitrary	14

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before A0	After OFFSET	PSIZE	CONVDP	A1
CVXYL A0,A1	00400030h	00000000h	0010h	0014h	00020300h
CVXYL A0,A1	00400030h	00000000h	0008h	0014h	00020180h
CVXYL A0,A1	00400030h	00000000h	0004h	0014h	00020000h
CVXYL A0,A1	00400030h	00008000h	0004h	0014h	00028000h
CVXYL A0,A1	00400030h	0F000000h	0004h	0014h	0F020000h
CVXYL A0,A1	00400030h	00000000h	0002h	0014h	00020060h
CVXYL A0,A1	00400030h	00000000h	0001h	0014h	00020030h
CVXYL A0,A1	00400030h	00000000h	0001h	0013h	00040030h
CVXYL A0,A1	00400030h	00000000h	0001h	0015h	00010000h

CONVDP = 0013h corresponds to DPTCH = 00001000h
CONVDP = 0014h corresponds to DPTCH = 00000800h
CONVDP = 0015h corresponds to DPTCH = 00000400h

Syntax

DEC *Rd*

Execution

Rd – 1 → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	0	1	R	Rd			

Description

DEC subtracts 1 from the contents of the destination register and stores the result in the destination register. This instruction is an alternate mnemonic for `SUBK 1,Rd`.

You can use the DEC instruction with the SUBB instruction to perform multiple-precision arithmetic.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A1	A1	N C Z V
`DEC A1`	00000010h	0000000Fh	0 0 0 0
`DEC A1`	00000001h	00000000h	0 0 1 0
`DEC A1`	00000000h	FFFFFFFFh	1 1 0 0
`DEC A1`	FFFFFFFFh	FFFFFFFEh	1 0 0 0
`DEC A1`	80000000h	7FFFFFFFh	0 0 0 1

Syntax

DINT

Execution

0 → IE

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0

Description

DINT disables interrupts by clearing the global interrupt enable bit (IE[ST]) to 0. All interrupts except reset, NMI, bus fault, and ILLOP are disabled; the interrupt enable mask in the INTENB register is ignored. The remainder of the status register is unaffected.

The EINT instruction enables interrupts.

Machine States

3

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected
IE *0*

Examples

Code	Before ST	After ST
DINT	00000010h	00000010h
DINT	00200010h	00000010h

Syntax

DIVS *Rs, Rd*

Execution

If Rd is an even-numbered register

$$\frac{Rd : Rd + 1}{Rs} \rightarrow Rd, \quad \text{remainder} \rightarrow Rd + 1$$

If Rd is an odd-numbered register

$$\frac{Rd}{Rs} \rightarrow Rd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rs				R	Rd			

Description

DIVS performs a signed 32-bit or 64-bit divide. The source register contains the 32-bit signed divisor. The destination register contains a 32-bit signed dividend or the most significant half of a 64-bit signed dividend, depending on whether Rd is an odd register (for example, A1 or B3) or an even register (for example, A8 or B2):

Rd Even DIVS performs a signed divide of the 64-bit operand contained in the 2 consecutive registers, starting at the specified destination register, by the 32-bit contents of the source register. The specified even-numbered destination register, Rd, contains the 32 MSBs of the dividend. The next consecutive register (which is odd-numbered) contains the 32 LSBs of the dividend. The quotient is stored in the destination register, and the remainder is stored in the following register (Rd+1). The remainder is always the same sign as the dividend (in Rd:Rd+1). Avoid using A14 or B14 as the destination register, since this overwrites the SP; the assembler issues a warning in this case.

Rd Odd DIVS performs a signed divide of the 32-bit operand contained in the destination register by the 32-bit value in the source register. The quotient is stored in the destination register; the remainder is not returned.

Rs and Rd must be in the same register file.

Machine States

Rd Odd: 39 (normal case)
41 (if result = 80000000h)
7 (if Rs = 0)

Rd Even: 40 (normal case)
41 (if result = 80000000h)
7 (if Rs = 0)

Status Bits

N **0 if**
Rs = 0, **or**
Rd is even and Rd ≥ Rs, **or**
Quotient is nonnegative.
1 if
Result = 80000000h **or**
Quotient is negative.

C Unaffected

Z **0 if**
Rs = 0, **or**
Rd is even and Rd ≥ Rs, **or**
Result = 80000000h, **or**
Quotient ≠ 0.
1 if
Quotient = 0.

V **1** if quotient overflows (cannot be represented by 32 bits), **0** otherwise. The following conditions cause an overflow and set the overflow flag:
Divisor (Rs) is 0.
Quotient cannot be contained within 32 bits.

Example 1

This example divides the contents of registers A0 and A1 by the contents of register A2, stores the result in register A0, and stores the remainder in A1. Note that the contents of register A2 are not affected by instruction execution.

```
DIVS  A2,A0
```

Before			After			
A0	**A1**	**A2**	**A0**	**A1**	**A2**	**NCZV**
12345678h	87654321h	87654321h	D95BC60Ah	5CA1DD7h	87654321h	1 x 0 0
EDCBA987h	789ABCDFh	87654321h	26A439F6h	EA35E229h	87654321h	0 x 0 0
EDCBA987h	789ABCDFh	789ABCDFh	D95BC60Ah	EA35E229h	789ABCDFh	1 x 0 0
12345678h	87654321h	789ABCDFh	26A439F6h	15CA1DD7h	789ABCDFh	0 x 0 0
12345678h	87654321h	00000000h	12345678h	87654321h	00000000h	0 x 0 1
00000000h	00000000h	00000000h	00000000h	00000000h	00000000h	0 x 0 1
00000000h	00000000h	87654321h	00000000h	00000000h	87654321h	0 x 1 0
87654321h	00000000h	87654321h	87654321h	00000000h	87654321h	0 x 0 1

Example 2

This example divides the contents of register A1 by the contents of of register A2 and stores the result in register A1. Note that the contents of register A2 are not affected by instruction execution.

```
DIVS A2,A1
```

Before			After			
A0	**A1**	**A2**	**A0**	**A1**	**A2**	**NCZV**
00000000h	87654321h	12345678h	00000000h	FFFFFFFAh	12345678h	1 x 0 0
00000000h	87654321h	0EDCBA988h	00000000h	00000006h	EDCBA988h	0 x 0 0
00000000h	789ABCDFh	0EDCBA988h	00000000h	FFFFFFFAh	EDCBA988h	1 x 0 0
00000000h	789ABCDFh	12345678h	00000000h	00000006h	12345678h	0 x 0 0
00000000h	87654321h	00000000h	00000000h	87654321h	00000000h	0 x 0 1
00000000h	00000000h	00000000h	00000000h	00000000h	00000000h	0 x 0 1

Syntax

DIVU *Rs, Rd*

Execution

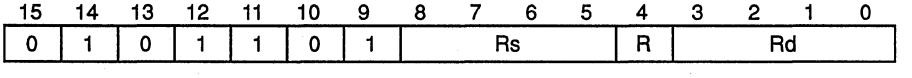
If Rd is an even-numbered register

$$\frac{Rd : Rd + 1}{Rs} \rightarrow Rd, \quad \text{remainder} \rightarrow Rd + 1$$

If Rd is an odd-numbered register

$$\frac{Rd}{Rs} \rightarrow Rd$$

Instruction Words



Description

DIVU performs an unsigned 32-bit or 64-bit divide. The source register contains the 32-bit divisor. The destination register contains a 32-bit dividend or the most significant half of a 64-bit dividend, depending on whether Rd is an odd register (for example, A1 or B3) or an even register (for example, A8 or B2):

Rd Even DIVU performs an unsigned divide of the 64-bit operand contained in the 2 consecutive registers, starting at the destination register, by the 32-bit contents of the source register. The specified even-numbered destination register, Rd, contains the 32 MSBs of the dividend. The next consecutive register (which is odd-numbered) contains the 32 LSBs of the dividend. The quotient is stored in the destination register, and the remainder is stored in the following register (Rd+1). Avoid using A14 or B14 as the destination register, since this overwrites the SP; the assembler issues a warning in this case.

Rd Odd DIVU performs an unsigned divide of the 32-bit operand contained in the destination register by the 32-bit value in the source register. The quotient is stored in the destination register; the remainder is not returned.

Rs and Rd must be in the same register file.

Machine States

Rd Odd: 37 (normal case)
7 (if Rs = 0)

Rd Even: 37 (normal case)
5 (if Rs = 0 or Rs ≤ Rd)

Status Bits

N Unaffected

C Unaffected

Z **0 if**
Rs = 0, **or**
Rd is even and Rd ≥ Rs, **or**
Quotient ≠ 0.
1 if
Quotient = 0.

V **1** if quotient overflows (cannot be represented by 32 bits), **0** otherwise. The following conditions cause an overflow and set the overflow flag:
Divisor (Rs) is 0.
Quotient cannot be contained within 32 bits.

Example 1

This instruction divides the contents of registers A0 and A1 by the contents of register A2, stores the unsigned result in register A0, and stores the remainder in A1. Note that the contents of register A2 are not affected by instruction execution.

```
DIVU  A2,A0
```

Before			After			
A0	**A1**	**A2**	**A0**	**A1**	**A2**	**NCZ V**
12345678h	87654321h	789ABCDFh	26A439F6h	15CA1DD7h	789ABCDFh	x x 0 0
12345678h	87654321h	00000000h	12345678h	87654321h	00000000h	x x 0 1
00000000h	00000000h	00000000h	00000000h	00000000h	00000000h	x x 0 1
00000000h	00000000h	87654321h	00000000h	00000000h	87654321h	x x 1 0
87654321h	00000000h	87654321h	87654321h	00000000h	87654321h	x x 0 1

Example 2

This instruction divides the contents of register A1 by the contents of register A2 and stores the unsigned result in register A1. Note that the contents of register A2 are not affected by instruction execution.

```
DIVU  A2,A1
```

Before			After			
A0	**A1**	**A2**	**A0**	**A1**	**A2**	**NCZ V**
00000000h	789ABCDFh	12345678h	00000000h	00000006h	12345678h	x x 0 0
00000000h	12345678h	00000000h	00000000h	12345678h	00000000h	x x 0 1
00000000h	00000000h	00000000h	00000000h	00000000h	00000000h	x x 0 1
00000000h	00000000h	87654321h	00000000h	00000000h	87654321h	x x 1 0
00000000h	87654321h	87654321h	00000000h	00000001h	87654321h	x x 0 0

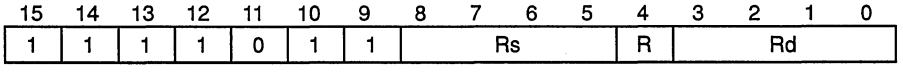
Syntax

DRAV *Rs, Rd*

Execution

COLOR1 pixels → *Rd
X half of Rs + X of half Rd → X half of Rd
Y half of Rs + Y of half Rd → Y half of Rd

Instruction Words



Description

DRAV writes the pixel value in the COLOR1 register to the location pointed to by the XY address in the destination register. Following the write, the XY address in the destination register is incremented by the value in the source register: the X half of Rs is added to the X half of Rd, and the Y half of Rs is added to the Y half of Rd. Any carry out from the lower (X) half of the register does not propagate into the upper (Y) half.

Use the SETCDP instruction (page 13-228) to set up CONVDP.

COLOR1 bits 0—31 are output on data bus lines 0—31, respectively. The pixel data used from COLOR1 is that which aligns to the destination location, so 32-bit patterns can be implemented. Rs and Rd must be in the same register file.

Implied Operands

Register	Name	Format	Description
B3	DPTCH	Linear	Destination pitch
B4	OFFSET	Linear	Screen origin (location 0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B9	COLOR1	Pixel	Pixel color

Address	Name	Description and Elements (Bits)
C00000B0h	CONTROL	PPOP Pixel-processing operations (22 options) W Window-checking operation T Transparency operation TM Selects 1 of 3 transparency options
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value by the time the DRAV instruction begins executing. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT (page 13-178) instruction. Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Pixel Processing

Set PPOP[[CONTROL]] to select a pixel-processing operation. This operation is applied to the pixel as it is moved to the destination location. At reset, the default pixel-processing operation is *replace* (S → D). For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window Checking

Select a window-checking mode by setting W[[CONTROL]]. If you select an active window-checking mode (W = 1, 2, or 3), the WSTART and WEND registers define the XY starting and ending corners of a rectangular window. The X and Y values in both WSTART and WEND can signed. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Transparency

You can enable transparency for this instruction by setting T[[CONTROL]] to 1. Select 1 of 3 transparency options by setting TM[[CONTROL]]. For more information, refer to Section 12.9, Transparency, on page 12-36.

Plane Masking

The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Shift Register Transfers

When this instruction is executed and CST bit is set, the normal memory read and write operations become SRT reads and writes.

Machine States

Refer to Section 15.1 on page 15-2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if a window violation occurs, *0* otherwise; unaffected if window clipping is not used.

Examples

These DRAV examples use the following implied operand setup.

Register File B		**I/O Registers**
DPTCH)	= 200h	CONVDP = 0016h
OFFSET	= 00010000h	
WSTART	= 00100000h	
WEND	= 003C0040h	
COLOR1	= FFFFFFFFh	

Assume that memory contains the following values before instruction execution:

Address	**Data**
00018040h	8888h

Note that the initial XY address in A0 is equivalent to linear address 18040h.

Code	Before					After			
		A0	A1	PSIZE	PPOP	W	PMASK	A0	@18040h
DRAV	A1,A0	00400040h	00100010h	0001h	00000	00	0000h	00500050h	8889h
DRAV	A1,A0	00400020h	00100010h	0002h	00000	00	0000h	00500030h	888Bh
DRAV	A1,A0	00400010h	00100010h	0004h	00000	00	0000h	00500020h	888Fh
DRAV	A1,A0	00400008h	00100010h	0008h	00000	00	0000h	00500018h	88FFh
DRAV	A1,A0	00400004h	00100010h	0010h	00000	00	0000h	00500014h	FFFFh
DRAV	A1,A0	00400004h	0000FFFFh	0010h	01010	00	0000h	00400003h	0000h
DRAV	A1,A0	00400004h	FFFF0000h	0010h	10011	00	0000h	003F0004h	0000h
DRAV	A1,A0	00400004h	00010001h	0010h	00000	11	0000h	00410005h	0000h
DRAV	A1,A0	00400004h	00400004h	0010h	00000	00	00FFh	00800008h	FF00h

Syntax

DSJ *Rd, Address*

Execution

Rd – 1 → Rd
If Rd ≠ 0, then (offset × 16) + PC' → PC
If Rd = 0, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	0	R	Rd			
offset															

Description

DSJ decrements the contents of the destination register by 1. Depending on the decremented value of Rd, the TMS34020 either jumps or skips the jump:

- **Rd –1 ≠ 0.** The updated PC (used in the jump address calculation) points to the instruction word that immediately follows the second word of the DSJ instruction. The signed word offset is converted to a bit offset by multiplying by 16. The new PC address is then obtained by adding the resulting signed offset (offset × 16) to the current PC.
- **Rd –1 = 0.** The TMS34020 skips the jump and continues program execution with the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address – PC')/16, where PC' is the address of the instruction word immediately following the second word of the DSJEQ instruction; this results in a jump range of –32,768 to +32,767 words. (The offset is the second instruction word of the opcode.)

The DSJ instruction is useful for large loops involving a counter. For shorter loops, the assembler automatically translates the DSJ mnemonic into a DSJS instruction.

Machine States

2 if no jump
3 if jump

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before A5	After A5	Jump taken?
`DSJ A5,LOOP`	00000009h	00000008h	Yes
`DSJ A5,LOOP`	00000001h	00000000h	No
`DSJ A5,LOOP`	00000000h	FFFFFFFFh	Yes

Syntax

DSJEQ *Rd, Address*

Execution

If Z = 1, then Rd – 1 → Rd
If Rd ≠ 0, then PC' + (offset × 16) → PC
If Rd = 0, then go to next instruction

If Z = 0, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	0	1	R	Rd			
offset															

Description

The DSJEQ instruction evaluates the status Z bit. Depending on the value of that bit, the TMS34020 either skips the jump, or decrements Rd and then makes a decision to jump or skip the jump:

- **Z = 1.** The TMS34020 decrements the contents of the destination register by 1.
  - **Rd – 1 ≠ 0.** The TMS34020 jumps relative to the current PC. The current PC points to the instruction word that immediately follows the second word of the DSJEQ instruction. The signed word offset is converted to a bit offset by multiplying by 16. The new PC address is then obtained by adding the resulting signed offset (offset × 16) to the address of the next instruction.
  - **Rd – 1 = 0.** The TMS34020 skips the jump and continues program execution at the next sequential instruction.
- **Z = 0.** The TMS34020 skips the jump and continues program execution at the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address – PC')/16, where PC' is the address of the instruction word immediately following the second word of the DSJ instruction; this results in a jump range of –32,768 to +32,767 words. (The offset is the second instruction word of the opcode.)

You can use this instruction after an explicit or implicit compare to 0. Additional information on these types of compares can be obtained in the CMP, CMPI, and MOVE-to-register instructions.

Machine States

2 if no jump
3 if jump

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	A5	N C Z V	A5	Jump taken?
`DSJEQ A5,LOOP`	00000009h	x x 1 x	00000008h	Yes
`DSJEQ A5,LOOP`	00000001h	x x 1 x	00000000h	No
`DSJEQ A5,LOOP`	00000000h	x x 1 x	FFFFFFFFh	Yes
`DSJEQ A5,LOOP`	00000009h	x x 0 x	00000009h	No
`DSJEQ A5,LOOP`	00000001h	x x 0 x	00000001h	No
`DSJEQ A5,LOOP`	00000000h	x x 0 x	00000000h	No

Syntax

DSJNE *Rd, Address*

Execution

If Z = 0, then Rd – 1 → Rd
  If Rd ≠ 0, then PC' + (offset × 16) → PC
  If Rd = 0, then go to next instruction
If Z = 1, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	1	0	R	Rd			
offset															

Description

The DSJNE instruction evaluates the status Z bit. Depending on the value of that bit, the TMS34020 either skips the jump or decrements Rd and then makes a decision to jump or skip the jump:

- **Z = 0.** The TMS34020 decrements the contents of Rd by 1.
  - **Rd – 1 ≠ 0.** The TMS34020 jumps relative to the current PC. The current PC points to the instruction word that immediately follows the second word of the DSJNE instruction. The signed word offset is converted to a bit offset by multiplying by 16. The new PC address is then obtained by adding the resulting signed offset (offset × 16) to the address of the next instruction.
  - **Rd – 1 = 0.** The TMS34020 skips the jump and continues program execution at the next sequential instruction.
- **Z = 1.** The TMS34020 skips the jump and continues program execution at the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address – PC')/16, where PC' is the address of the instruction word immediately following the second word of the DSJNE instruction; this results in a jump range of –32,768 to +32,767 words. (The offset is the second instruction word of the opcode.)

You can use this instruction after an explicit or implicit compare to 0. Additional information on these types of compares can be obtained in the CMP, CMPI, and MOVE-to-register instructions.

Machine States

2 if no jump
3 if jump

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before A5	N C Z V	After A5	Jumps taken?
`DSJNE A5,LOOP`	00000009h	x x 1 x	00000009h	No
`DSJNE A5,LOOP`	00000001h	x x 1 x	00000001h	No
`DSJNE A5,LOOP`	00000000h	x x 1 x	00000000h	No
`DSJNE A5,LOOP`	00000009h	x x 0 x	00000008h	Yes
`DSJNE A5,LOOP`	00000001h	x x 0 x	00000000h	No
`DSJNE A5,LOOP`	00000000h	x x 0 x	FFFFFFFFh	Yes

Syntax

DSJS *Rd, Address*

Execution

Rd – 1 → Rd
If Rd ≠ 0, then PC' + (offset × 16) → PC
If Rd = 0, then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	D	Offset					R	Rd			

Fields

D is a 1-bit direction bit (from PC' to Address):
D=0 forward jump
D=1 backward jump

Description

DSJS decrements the contents of the destination register by 1. Depending on the result, the TMS34020 either jumps or skips the jump:

- **Rd – 1 ≠ 0.** The TMS34020 jumps relative to PC'. PC' points to the instruction word that immediately follows the DSJS instruction. Internally, the 5-bit word offset is multiplied by 16 to convert it to a bit offset. This allows a jump range of –30 to +32 words from PC'.
  - **If direction bit D = 0.** The new PC address is obtained by adding the resulting offset to PC'.
  - **If direction bit D = 1.** The new PC address is obtained by subtracting the resulting offset from PC'.
- **Rd – 1 = 0.** The TMS34020 skips the jump and continues program execution at the next sequential instruction.

The *Address* operand is a 32-bit address. The assembler calculates the offset as (Address – PC')/16; this results in a jump range of –30 to +32 words from PC'. (The offset is encoded as part of the instruction word.)

Note that the assembler also calculates the value of *D* and generates a warning is the jump is not in the range.

This instruction is useful for coding tight loops for cache-resident routines.

Machine States

2 if no jump
3 if jump

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before A5	After A5	Jump taken?
`DSJS A5,LOOP`	00000009h	00000008h	Yes
`DSJS A5,LOOP`	00000001h	00000000h	No
`DSJS A5,LOOP`	00000000h	FFFFFFFFh	Yes

Syntax

EINT

Execution

1 → IE

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	1	1	0	0	0	0	0

Description

EINT sets the global interrupt enable bit (IE) to 1, allowing interrupts to be enabled. When IE=1, individual interrupts are enabled by setting the appropriate bits in the INTENB interrupt mask register. The rest of the status register is unaffected.

The DINT instruction disables interrupts.

Machine States

3

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected
IE *1*

Examples

Code	Before ST	After ST
`EINT`	00000010h	00200010h
`EINT`	00200010h	00200010h

Syntax

EMU *count*

Execution

Conditionally enter emulator mode

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	count				

Description

This instruction is not for general use. It is supplied to proved support for emulation of the TMS34020.

Machine States

8 (or more if EMU mode is entered)

Status Bits

N Indeterminate
C Indeterminate
Z Indeterminate
V Indeterminate

Syntax

EXGF *Rd*[, *F*]

Execution

Rd → FS0, FE0 **or** Rd → FS1, FE1
FS0, FE0 → Rd **or** FS1, FE1 → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	F	1	0	0	0	R	Rd			

Description

EXGF exchanges the 6 LSBs of the destination register with the selected 6 bits of field information (field size and field extension). Bit 5 of the 6-bit quantity in Rd is exchanged with the field-extension value. The upper 26 bits of Rd are cleared.

31	30	29	28	26	25	22	21	11	10—6	5	4—0
N	C	Z	V	BF	IX	SS	IE	FE1	FS1	FE0	FS0

Note: Shaded portions are reserved.

EXGF's *F* parameter is optional:

F=0 selects FS0, FE0 to be exchanged
F=1 selects FS1, FE1 to be exchanged

If you do not specify an F parameter, the default is 0.

For more information, refer to Section 4.1, The Status Register, on page 4-2.

Machine States

F0 1
F1 2

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	A5	ST	A5	ST
`EXGF A5,0`	FFFFFFC0h	F0000FFFh	0000003Fh	F0000FC0h
`EXGF A5,1`	FFFFFFC0h	F0000FFFh	0000003Fh	F000003Fh

Syntax

EXGPC *Rd*

Execution

Rd → PC, PC' → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	1	R	Rd			

Description

EXGPC exchanges the next program counter value with the destination register contents. After this instruction has been executed, the destination register contains the address of the instruction immediately following the EXGPC instruction.

Note that the TMS34020 sets the 4 LSBs of the program counter to 0 (word aligned).

This instruction provides a *quick call* capability by saving the return address in a register (rather than on the stack). The return from the call is accomplished by repeating the instruction at the end of the subroutine. Note that the subroutine address must be reloaded following each call-return operation.

Machine States

2

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After	
	A1	**PC**	**A1**	**PC**
`EXGPC A1`	00001C10h	00002080h	00002090h	00001C10h
`EXGPC A1`	00001C50h	00002080h	00002090h	00001C50h

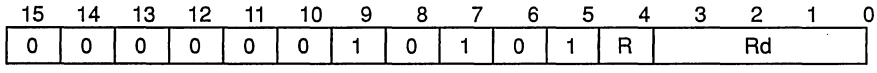
Syntax

EXGPS *Rd*

Execution

Rd → PC, PC' → Rd

Instruction Words



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	1	R	Rd			

Description

EXGPS exchanges the contents of the destination register with the contents of PSIZE register. The pixel size is assumed to be a legal value (1, 2, 4, 8, 16, or 32). The destination register is loaded with the pixel size, and its previous contents are written with a 16-bit write to the PSIZE register. For more information, refer to Section 12.6, Auxiliary Graphics Instructions, on page 12-17.

Implied Operands

Address	Name	Description and Operations
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)

Machine States

2 (1)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

```
EXGPS A0
```

Before	A0 =	00000001	PSIZE =	0008
After	A0 =	00000008	PSIZE =	0001

Syntax

FILL L

Execution

COLOR1 pixels → pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0

Description

FILL processes a set of source pixel values (specified by the COLOR1 register) with a destination pixel array.

This instruction operates on a 2-dimensional array of pixels using pixels defined in the COLOR1 register. As the FILL proceeds, the source pixels are combined with destination pixels according to the selected graphics operations.

Note that the **L** parameter in the instruction syntax does not represent a value or a register; the **L** is entered as part of the instruction and identifies the starting address of the pixel array as a linear address.

The following set of implied operands govern the operation of the instruction and define both the source pixels and the destination array.

Implied Operands

Register	Name	Format	Description
B2 †	DADDR	Linear	Pixel array starting address
B3	DPTCH	Linear	Pixel array pitch
B7	DYDX	XY	Pixel array dimensions (rows:columns)
B9	COLOR1	Pixel	Fill color or 32-bit pattern

† Changed by FILL during execution.

Address	Name	Description and Operations
C00000B0h	CONTROL	PPOP Pixel-processing operations (22 options) T Enables transparency operation TM Selects 1 of 3 transparency options
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write (when setting up the I/O registers) may not, in some cases, contain the desired value when you execute the FILL instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT (page 13-178) instruction. Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Destination Address

The contents of the DADDR, DPTCH, and DYDX registers define the location of the destination pixel array. For a detailed description, refer to Section 12.5, Pixel Array Instructions, on page 12-8.

Pixel Processing

Set PPOP[[CONTROL]] to select a pixel-processing operation. This operation is applied to the pixel as it is moved to the destination location. There are 16 Boolean and 6 arithmetic operations; the default operation at reset is *replace* (S→D). Note that the destination data is read through the plane mask and then processed. The 6 arithmetic operations do not operate with a pixel size of 1 bit per pixel. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window Checking

Window checking **cannot** be used with this instruction. The contents of the WSTART and WEND registers are ignored.

Corner Adjust

There is no corner adjust for this instruction. The direction of the FILL is fixed as increasing linear addresses.

Transparency

You can enable transparency for this instruction by setting T[[CONTROL]] to 1. You can also select 1 of 3 transparency options by setting TM[[CONTROL]]. For more information, refer to Section 12.9, Transparency, on page 12-36.

Interrupts

This instruction can be interrupted at a word or row boundary of the destination array. For more information, refer to Section 6.6, Interrupting Graphics Operations, on page 6-13.

Plane Masking

The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Shift Register Transfers

If CST[[DPYTCTL]] is set, each memory read or write initiated by the FILL generates a shift-register-transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.) For more information, refer to subsection 9.13.4, VRAM Bulk Initialization, on page 9-47.

Machine States

complex instruction

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

These FILL examples use the following implied operand setup.

Register File B:		**I/O Registers:**
DADDR	= 00002010h	PSIZE = 0008h
DPTCH	= 00000080h	
DYDX	= 0002000Dh	
COLOR1	= 30303030h	

Assume that memory contains the following values before instruction execution.

Linear Address	**Data**
00002000h	1100h, 3322h, 5544h, 7766h, 9988h, BBAAh, DDCCh, FFEEh
00002080h	1100h, 3322h, 5544h, 7766h, 9988h, BBAAh, DDCCh, FFEEh

Example 1

This example uses the pixel-processing *replace (S → D)* operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, PPOP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data
00002000h	1100h, 3030h, 3030h, 3030h, 3030h, 3030h, 3030h, FF30h
00002080h	1100h, 3030h, 3030h, 3030h, 3030h, 3030h, 3030h, FF30h

Example 2

This example uses the *($\overline{S}$ and D) → D* pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 2C00h (T=0, PPOP=01010).

After instruction execution, memory contains the following values:

Linear Address	Data
00002000h	1100h, 0302h, 4544h, 4746h, 8988h, 8B8Ah, CDCCh, FFCEh
00002080h	1100h, 0302h, 4544h, 4746h, 8988h, 8B8Ah, CDCCh, FFCEh

Example 3

This example uses plane masking — the 4 MSBs are masked. Before instruction execution, PMASK = 0F0F0h and CONTROL = 0000h (T=0, PPOP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data
00002000h	1100h, 3020h, 5040h, 7060h, 9080h, B0A0h, D0C0h, FFE0h
00002080h	1100h, 3020h, 5040h, 7060h, 9080h, B0A0h, D0C0h, FFE0h

Syntax

FILL XY

Execution

COLOR1 pixels → pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0

Description

FILL processes a set of source pixel values (specified by the COLOR1 register) with a destination pixel array.

This instruction operates on a 2-dimensional array of pixels using pixels defined in the COLOR1 register. As the FILL proceeds, the source pixels are combined with destination pixels according to the selected graphics operations.

Note that the **XY** parameter in the instruction syntax does not represent a value or a register; it is entered as part of the instruction and identifies the starting address of the pixel array as an XY address.

The following set of implied operands govern the operation of the instruction and define both the source pixels and the destination array.

Implied Operands

Register	Name	Format	Description
B2 †‡	DADDR	XY	Pixel array starting address
B3	DPTCH	Linear	Pixel array pitch
B4	OFFSET	Linear	Screen origin (address of 0,0)
B5	WSTART	XY	Window starting corner
B6	WEND	XY	Window ending corner
B7 †‡	DYDX	XY	Pixel array dimensions (rows:columns)
B9	COLOR1	Pixel	Fill color or 32-bit pattern

† Changed by FILLXY during execution.

‡ Used for common rectangle function with window hit operation (W=1).

Address	Name	Description and Elements (Bits)
C00000B0h	CONTROL	PPOP Pixel-processing operations (22 options) W Window-checking operation T Enables transparency TM Selects 1 of 3 transparency options
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the FILL

XY instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT instruction (page 13-178). Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Destination Array

The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers. For a detailed description, refer to Section 12.5, Pixel Array Instructions, on page 12-8.

Pixel Processing

Pixel processing can be used with this instruction. PPOP[[CONTROL]] specifies the pixel-processing operation that is applied to pixels as they are processed with the destination array. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace (S → D)* operation. Note that the destination data is read through the plane mask and then processed. The 6 arithmetic operations do not operate with a pixel size of 1 bit per pixel. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window Checking

The window operations can be used with this instruction. You can select window pick, violation detect, or preclipping by setting W[[CONTROL]] to 1, 2, or 3, respectively. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Corner Adjust

There is no corner adjust for this instruction. The direction of the FILL is fixed as increasing linear addresses.

Transparency

You can enable transparency for this instruction by setting T[[CONTROL]] to 1. Select 1 of 3 transparency modes by setting TM[[CONTROL]]. For more information, refer to Section 12.9, Transparency, on page 12-36.

Interrupts

This instruction can be interrupted at a word or row boundary of the destination array. For more information, refer to Section 6.6, Interrupting Graphics Operations, on page 6-13.

Plane Masking

The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Shift Register Transfers

If CST[[DPYCTL]] is set, each memory read or write initiated by the FILL generates a shift-register-transfer read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.) For more information, refer to subsection 9.13.4, VRAM Bulk Initialization, on page 9-47.

Machine States

complex instruction

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

These FILL examples use the following implied operand setup.

Register File B:		**I/O Registers:**	
DADDR	= 00520007h	CONVDP	= 0017h
DPTCH	= 00000100h	PSIZE	= 0004h
OFFSET	= 00010000h	PMASK	= 0000h
WSTART	= 0030000Ch	CONTROL	= 0000h
WEND	= 00530014h		(W=00, T=0, PPOP=00000)
DYDX	= 00030012h		
COLOR1	=		FFFFFFFFh

Assume that memory contains the following values before instruction execution.

Linear Address	**Data**
00015200h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
00015300h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
00015400h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh

Example 1

This example uses the *replace* (S → D) pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PPOP=00000).

After instruction execution, memory contains the following values:

Linear Address	**Data**
00015200h	3210h, F654h, FFFFh, FFFFh, FFFFh, FFFFh, BA9Fh, FEDCh
00015300h	3210h, F654h, FFFFh, FFFFh, FFFFh, FFFFh, BA9Fh, FEDCh
00015400h	3210h, F654h, FFFFh, FFFFh, FFFFh, FFFFh, BA9Fh, FEDCh

XY Addressing

	X Address
Y	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 5 6 7 8 9 A B C D E F
Address 52	0 1 2 3 4 5 6 F F F F F F F F F F F F F F F F 9 A B C D E F
Address 53	0 1 2 3 4 5 6 F F F F F F F F F F F F F F F F 9 A B C D E F
Address 54	0 1 2 3 4 5 6 F F F F F F F F F F F F F F F F 9 A B C D E F

Example 2

This example uses the *(D XOR S)* → *D* pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 2800h (T=0, W=00, PPOP=01010).

After instruction execution, memory contains the following values:

	X Address 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
Y Address 52	0 1 2 3 4 5 6 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 9 A B C D E F
53	0 1 2 3 4 5 6 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 9 A B C D E F
54	0 1 2 3 4 5 6 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 9 A B C D E F

Example 3

This example uses window operation 3: the destination is clipped. Before instruction execution, PMASK = 0000h and CONTROL = 00C0h (T=0, W=11, PPOP=00000).

After instruction execution, memory contains the following values:

	X Address 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
Y Address 52	0 1 2 3 4 5 6 7 8 9 A B F F F F F F F F 5 6 7 8 9 A B C D E F
53	0 1 2 3 4 5 6 7 8 9 A B F F F F F F F F 5 6 7 8 9 A B C D E F
54	0 1 2 3 4 5 6 7 8 9 A B F F F F F F F F 5 6 7 8 9 A B C D E F

Example 4

This example uses plane masking: the MSB is masked. Before instruction execution, PMASK = 8888h and CONTROL = 0000h (T=0, W=00, PPOP=00000).

After instruction execution, memory contains the following values:

	X Address 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
Y Address 52	0 1 2 3 4 5 6 7 F F F F F F F F 7 7 7 7 7 7 7 7 F 9 A B C D E F
53	0 1 2 3 4 5 6 7 F F F F F F F F 7 7 7 7 7 7 7 7 F 9 A B C D E F
54	0 1 2 3 4 5 6 7 F F F F F F F F 7 7 7 7 7 7 7 7 F 9 A B C D E F

Syntax

FLINE {**0** | **1**}

Execution

The two execution algorithms for FLINE are explained below. These algorithms are similar, varying only in their treatment of the case when d=0.

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	0	Z	0	0	1	1	0	1	0

Fields

The assembler sets bit 7 in the instruction word (the Z bit) to 0 or 1, depending on which FLINE algorithm you select:
Z=0 selects algorithm 0
Z=1 selects algorithm 1

Description

FLINE is a version of LINE. The major differences are:

- FLINE is faster than LINE.
- FLINE does not support windowing.
- FLINE requires B2 to be linear.

FLINE performs the inner loop of Bresenham's line-drawing algorithm. This type of line draw plots a series of points (x_i,y_i) either diagonally or laterally with respect to the previous point. Movement from pixel to pixel always proceeds in a dominant lateral direction. The algorithm may or may not also increment in the direction with the smaller dimension (this produces a diagonal movement). Two XY-format registers supply the XY increment values for the two possible movements. The FLINE instruction maintains a decision variable, *d*, that acts as an error term, controlling movement in either the dominant or diagonal direction. The algorithm operates in one of two modes, depending on how the condition *d*=0 is treated.

During FLINE execution, some portion of a line from (x_0,y_0) to (x_1,y_1) is drawn. The line is drawn so that the axis with the largest extent has dimension *a*, and the axis with the least extent has dimension *b*. Thus, *a* is the larger (in absolute terms) of $y_1 - y_0$, or $x_1 - x_0$ and *b* is the smaller of the two. This means that $a \geq b \geq 0$.

The LINIT instruction (page 13-146) is designed to do most of the setup for FLINE with the exception of the XY to linear conversion of DADDR.

The following values must be supplied to draw a line from (x_0,y_0) to (x_1,y_1):

- Set the value in DADDR to be the **linear** address of the first pixel in the line at (x_0,y_0).
- Use the line endpoints to determine the major and minor dimensions (*a* and *b*, respectively) for the line draw; then set the DYDX register to this value (*b*:*a*).
- Place the signed XY increment for a movement in the diagonal (or minor) direction ($d \geq 0$ for Z=0, $d > 0$ for Z=1) in the INC1 register.

- Place the signed XY increment for a movement in the dominant (or major) direction ($d < 0$ for Z=0, $d \leq 0$ for Z=1) in the INC2 register.
- Set the initial value of the decision variable in register B0 to $2b - a$.
- Set the initial count value in the COUNT register to $a + 1$.
- Set the COLOR1 and COLOR0 registers.
- Set the PATTERN register to the required pattern.

  FLINE handles the contents of PATTERN in the same way as LINE (unlike PFILL XY). With FLINE, the first pixel drawn is controlled by bit 0 of the PATTERN register.

  The PATTERN register contains a 32-bit repeating line-style pattern. If bit 0 of PATTERN is *0*, then the first pixel drawn by LINE is a COLOR0 pixel. If bit 0 of PATTERN is *1*, then the first pixel drawn by LINE is a COLOR1 pixel. The second pixel drawn by LINE is controlled by bit 1 of B13, and so on. If the line is longer than 32 pixels, the PATTERN is reused cyclically; therefore, the 33rd pixel on the line is once again controlled by bit 0 of PATTERN. As each pixel is drawn, the contents of PATTERN are rotated right (circular shifted) by 1 bit. The LSB of the rotated pattern controls the next pixel the instruction puts out.

  If PATTERN contains all 1s, the line is drawn in a solid color using the replicated pixel value contained in COLOR1; if PATTERN contains all 0s, the line is drawn in a solid color using COLOR0.

The FLINE instruction may use one of two algorithms, depending on the value of Z.

Algorithm 0 (Z=0):

```
While COUNT > 0
        COUNT = COUNT – 1
        Draw the next pixel
        If d ≥ 0
            d = d + 2b – 2a
            POINTER = POINTER + INC1
        Else d = d + 2b;
            POINTER = POINTER + INC2
```

Algorithm 1 (Z=1):

```
While COUNT > 0
        COUNT = COUNT – 1
        Draw the next pixel
        If d > 0
            d = d + 2b – 2a
            POINTER = POINTER + INC1
        Else d = d + 2b;
            POINTER = POINTER + INC2
```

For more information about FLINE, refer to Section 12.4, Line Instructions, on page 12-7.

Implied Operands

Register		Name	Format	Description
B0	†	SADDR	Integer	Decision variable (d)
B2	†	DADDR	Linear	Starting point
B3		DPTCH	Linear	Destination pitch (only used if CONVDP specifies arbitrary pitch)
B7		DYDX	XY	(b: a) = Minor: major dimension
B8	‡	COLOR0	Pixel	COLOR0
B9	¶	COLOR1	Pixel	COLOR1
B10	†	COUNT	Integer	Loop count
B11		INC1	XY	Minor axis (diagonal) increment
B12		INC2	XY	Major axis (dominant) increment
B13		PATTERN	Pattern	Pattern register

† These registers are changed by instruction execution.

‡ This register is only required if there are 0's in the pattern.

§ This register is only required if there are 1's in the pattern.

Address	Name	Description and Elements (Bits)
C00000B0h	CONTROL	PPOP Pixel-processing operations (22 options) T Transparency operation TM Sets transparency mode
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the FILL XY instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT instruction (page 13-178). Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Pixel Processing

PPOP[[CONTROL]] specifies the operation to be applied to the pixel as it is written. There are 22 operations; the default case at reset is the pixel-processing *replace* (S → D) operation. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window Checking

Window checking **cannot** be used with this instruction. The line must be preclipped; for more information, refer to Section 12.7, Window Checking, on page 12-19.

Transparency

You can enable transparency for this instruction by setting T[[CONTROL]] to 1. Select 1 of 3 transparency options by setting TM[[CONTROL]]. For more information, refer to Section 12.9, Transparency, on page 12-36.

Plane Masking

The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Interrupts

This instruction can be interrupted at a pixel boundary. For more information, refer to Section 6.6, Interrupting Graphics Operations, on page 6-13.

Machine States

Refer to Section 15.1 on page 15-2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

This is an example of a C-compatible assembly routine which draws a solid line on the screen. It takes 4 parameters from the C parameter stack: (xstart, ystart) and (xend, yend), and uses LINIT to initialize the B-file implied operands for LINE or FLINE. It also special-cases horizontal and vertical lines and uses a FILL for these.

This routine makes the assumes that the following B registers and I/O registers have been initialized by the caller:

B-file registers DPTCH, OFFSET, WSTART, WEND and COLOR1
I/O registers CONTROL, CONVDP, PSIZE, and PMASK

```
STK             .set    A14             ; C-parameter stack pointer
DADDR           .set    B2              ; Destination address register
DYDX            .set    B7              ; Delta X/delta Y register
INC1            .set    B11             ; Minor axis (diagonal) increment
PATTERN         .set    B13             ; Pattern register

                .globl  _draw_line      ; Reference for external calls
_draw_line:     mmtm    SP,A0,A1
                mmtm    SP,B2,B7,B10,B11,B12,B13,B14
                move    *-A14,A0,1      ; get xs
                move    *-A14,A1,1      ; get ys
                sll     16,A1
                movy    A1,A0           ; concatenate ys::xs
                move    *-A14,A1,1      ; get xe
                move    *-A14,A8,1      ; get ye
                sll     16,A8
                movy    A8,A1           ; concatenate ye::xe
                move    A0,DADDR
                move    A1,DYDX
                movi    -1,PATTERN
                linit
                jrc     exit            ; line is entirely outside window
                jrp     cliptst         ; line neither vertical or horizontal
```

```
; Confirmed that line is vertical or horizontal, so FILL can be used
; instead of LINE. Now determine whether line points in the +x or
; +y direction, or in the -x or -y direction.
vorh:       cmpxy  A0,A1          ; a = xe - xs, b = ye - ys
            jrv    negdir         ; jump if -x direction
            jrc    negdir         ; jump if -y direction
; Horizontal or vertical line points in +x or +y direction.
            move   A0,DADDR       ; DADDR = ys::xs
            subxy  A0,A1          ; a = xe - xs, b = ye - ys
            move   A1,DYDX        ; DYDX = length of line
            jruc   vorhline       ;
; Horizontal or vertical line points in -x or -y direction.
negdir:     move   A1,DADDR       ; DADDR = ye::xe
            subxy  A1,A0          ; a = xs - xe, b = ys - ye
            move   A0,DYDX        ; DYDX = length of line
; Draw the vertical or horizontal line.
vorhline:   addi   010001h,DYDX   ; ++DX, ++DY
            fill   XY             ; draw line
            jruc   exit
cliptst:    jrv    draw           ; clipping required use LINE
            cvdxyl DADDR          ; convert to linear address
            move   INC1,INC1      ; does line point up or down?
            jrlt   fdown
            fline  0              ; draw Bresenham line
            jruc   exit
fdown:      fline  1              ; draw Bresenham line
            jruc   exit
draw:       move   INC1,INC1      ; does line point up or down?
            jrlt   down
            line   0              ; draw Bresenham line
            jruc   exit
down:       line   1              ; draw Bresenham line
exit:       mmfm   SP,B2,B7,B10,B11,B12,B13,B14
            mmfm   SP,A0,A1
            rets   2              ;return to caller
```

Syntax

FPIXEQ

Execution

Scan from pixel pointed to by MADDR to find the first pixel equal to COLOR0, up to the number of pixels specified in MPTCH.

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	1	0	1	1	1	0	1	1

Description

FPIXEQ compares pixels in memory to the value in the COLOR0 register. The search ends when the current pixel is equal to the COLOR0 pixel of the number of pixels specified in MPTCH has been searched. The search takes place in either an postincrementing or a predecrementing fashion. MPTCH specifies the number of pixels to search. The count is positive in the postincrementing case (*search right*) and negative in the predecrementing case (*search left*).

The *magnitude* of the MPTCH counter decreases as the search continues.

If the instruction finds a pixel, Z is set to 1, the instruction aborts. MADDR is left pointing to the next pixel to check (postincrementing case) or the last pixel checked (predecrementing case).

If the instruction is interrupted, the pointers are set so the FPIXEQ resumes automatically. That is, the PC is decremented to point back to the FPIXEQ, the ST and PC are stacked, MADDR is left pointing to the next pixel to check (postincrementing case) or the last pixel checked (predecrementing case, and before the TRAP routine is started.The RETI at the end of the TRAP resumes the FPIXEQ instruction from the next pixel to be checked.For more information, refer to Section 12.6, Auxiliary Graphics Instructions, on page 12-17.

This instruction is useful for seedfills, data compression, and edge-flag fills.

Implied Operands

Register	Name	Format	Description
B8	COLOR0	Pixel	COLOR0
B10	MADDR	Linear	Search start pointer
B11	MPTCH	Integer	Number of pixels to search

Address	Name	Description and Elements (Bits)
C0000150h	PSIZE	Pixel size
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the FPIXEQ instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT (page 13-178) instruction. Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Pixel Processing Pixel processing **cannot** be used with this instruction.

Window Checking Window checking **cannot** be used with this instruction.

Transparency Transparency **cannot** be used with this instruction.

Plane Masking The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Machine States complex instruction

Status Bits

N Unaffected
C Unaffected
Z *1* if pixel found, *0* otherwise
V Unaffected

Syntax

FPIXNE

Execution

Scan from pixel pointed to by MADDR to find first pixel that is **not** equal to COLOR0, up to the number of pixels specified in MPTCH.

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	1	1	0	1	1	0	1	1

Description

FPIXNE compares pixels in memory to the value in the COLOR0 register. The search ends when the current pixel is not equal to the COLOR0 pixel. The search takes place in either an postincrementing or a predecrementing fashion. MPTCH specifies the maximum number of pixels to search. Note that this is **not** an XY value.The count is positive in the postincrementing case (*search right*) and negative in the predecrementing case (*search left*).

If the instruction finds a pixel, Z is set to 1, the instruction aborts. MADDR is left pointing to the next pixel to check (postincrementing case) or the last pixel checked (predecrementing case).

The *magnitude* of the MPTCH counter decreases as the search continues. If the instruction is interrupted, the pointers are set so the FPIXNE resumes automatically. That is, the PC is decremented to point back to the FPIXNE, the ST and PC are stacked, MADDR is left pointing to the next pixel to check (postincrementing case) or the last pixel checked (predecrementing case, and the TRAP routine is started.The RETI at the end of the TRAP resumes the FPIXNE instruction from the next pixel to be checked. For more information, refer to Section 12.6, Auxiliary Graphics Instructions, on page 12-17.

Implied Operands

Register	Name	Format	Description
B10	MADDR	Linear	Search start pointer
B11	MPTCH	Integer	Number of pixels to search
B8	COLOR0	Pixel	COLOR0

Address	Name	Description and Elements (Bits)
C0000150h	PSIZE	Pixel size
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the FPIXNE instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT (page 13-178) instruction. Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Pixel Processing

Pixel processing **cannot** be used with this instruction.

Window Checking

Window checking **cannot** be used with this instruction.

Transparency Transparency **cannot** be used with this instruction.

Plane Masking The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Machine States complex instruction

Status Bits

N Unaffected
C Unaffected
Z *1* if pixel found, *0* otherwise
V Unaffected

Syntax

GETPC *Rd*

Execution

PC' → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	0	R	Rd			

Description

GETPC copies the PC into the destination register (Rd). When the instruction completes, Rd contains the address of the instruction word immediately following the GETPC instruction. Execution continues with the next instruction. You can use GETPC with the EXGPC and JUMP instructions for quick call on jump operations. You can also use GETPC to access relocatable data areas whose position relative to the code area is known at assembly time.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After
	PC	A1
`GETPC A1`	00001BD0h	00001BE0h
`GETPC A1`	00001C10h	00001C20h

Syntax

GETPS *Rd*

Execution

PSIZE → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	1	0	R	Rd			

Description

GETPS fetches the pixel size stored in the PSIZE register and loads it into the destination register. The pixel size is assumed to be a legal value (1, 2, 4, 8, 16, or 32). For more information, refer to Section 12.6, Auxiliary Graphics Instructions, on page 12-17.

Implied Operands

Address	Name	Description and Elements (Bits)
C0000150h	PSIZE	Pixel Size (1,2,4,8,16,32)

Machine States

2

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before	After
	A0	**A0**
`GETPS A0`	00000035	00000008
(PSIZE = 8)		

Syntax **GETST** *Rd*

Execution ST → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	0	R	Rd			

Description GETST copies the contents of the status register into the destination register.

31	30	29	28	26	25	22	21	11	10—6	5	4—0
N	C	Z	V	BF	IX	SS	IE	FE1	FS1	FE0	FS0

Note: Shaded portions are reserved.

For more information, refer to Section 4.1, The Status Register, on page 4-2.

Machine States 1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before ST	After A1
`GETST A1`	20200010h	20200010h
`GETST A1`	00000010h	00000010h

Syntax

IDLE

Execution

Halt execution until interrupted

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Description

IDLE waits for an interrupt to occur. While IDLE waits for an interrupt, an internal microcode loop executes, and the PC continually points to the IDLE instruction. When the TMS34020 takes an interrupt, the PC value that is pushed onto the system stack points to the instruction that immediately follows the IDLE instruction. Upon return from the interrupt, the PC is restored and execution begins at the instruction immediately following the IDLE instruction.

An interrupt request that is not enabled is ignored by the IDLE instruction.

Machine States

Refer to Section 15.1 on page 15-2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Syntax

INC *Rd*

Execution

Rd + 1 → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	1	R	Rd			

Description

INC adds 1 to the contents of the destination register and stores the result in the destination register. This instruction is an alternate mnemonic for `ADDK 1,Rd`.

You can accomplish multiple-precision arithmetic by using INC in conjunction with the ADDC instruction.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a carry, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before	After	
	A1	A1	N C Z V
`INC A1`	00000000h	00000001h	0 0 0 0
`INC A1`	0000000Fh	00000010h	0 0 0 0
`INC A1`	FFFFFFFFh	00000000h	0 1 1 0
`INC A1`	FFFFFFFEh	FFFFFFFFh	1 0 0 0
`INC A1`	7FFFFFFFh	80000000h	1 0 0 1

Syntax

JA*condition Address*

Execution

If condition *true,* then Address → PC
If condition *false,* then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	code				1	0	0	0	0	0	0	0
16 LSBs of *Address*															
16 MSBs of *Address*															

Fields

code is a 4-bit digit that identifies the condition for the jump within the opcode. (See the condition codes table below.)

Description

The JA*condition* instruction conditionally jumps to an absolute address. The *condition* is part of a mnemonic that represents the condition for the jump; for example, if *condition* is UC, then the instruction is JAUC. (See the condition mnemonics and codes listed below.) If the specified condition is **true,** the TMS34020 jumps to the address and continues execution from that point. If the specified condition is **false,** the TMS34020 skips the jump and continues execution at the next sequential instruction. Note that the 4 LSBs of the program counter are hardwired to 0.

The *Address* operand in the syntax represents the 32-bit absolute address. Note that the second and third instruction words contain the address for the jump.

The JA*condition* instructions are usually used in conjunction with the CMP and CMPI instructions. The JAV and JANV instructions can also be used to detect window violations or CPW status.

Machine States

3 if no jump, else 4

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Flags for Branch			Code	Flags for Branch		
	NCZV	NCZV	NCZV		NCZV	NCZV	NCZV
JAUC HERE	x x x x			JAV HERE	x x x 1		
JAP HERE	0 x 0 x			JANZ HERE	x x 0 x		
JALS HERE	x x 1 x	x 1 x x		JANN HERE	0 x x x		
JAHI HERE	x 0 0 x			JANV HERE	x x x 0		
JALT HERE	0 x x 1	1 x x 0		JAN HERE	1 x x x		
JAGE HERE	0 x x 0	1 x x 1		JAB HERE	x 1 x x		
JALE HERE	0 x x 1	1 x x 0	x x 1 x	JANB HERE	x 0 x x		
JAGT HERE	0 x 0 0	1 x 0 1		JALO HERE	x 1 x x		
JAc HERE	x 1 x x			JAHS HERE	x 0 0 x	x x 1 x	
JANC HERE	x 0 x x			JANE HERE	x x 0 x		
JAZ HERE	x x 1 x			JAEQ HERE	x x 1 x		

Note that the TMS34020 jumps when any one or more of the *Flags for Branch* listed above are set as indicated.

Condition Codes

	Mnemonic				
	Non XY	**XY**	**Result of Compare**	**Status Bits**	**Code**
Unconditional Compares	JAUC	—	Unconditional	Don't care	0000
Unsigned Compares	JALO (JAC) (JAB)	— JAYN	Dst lower than Src	C	1000
	JALS	JAYLE	Dst lower or same as Src	C + Z	0010
	JAHI	JAYGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	JAHS (JANC) (JANB)	JAYNN	Dst higher or same as Src	$\overline{C}$	1001
	JAEQ (JAZ)	— JAYZ	Dst = Src	Z	1010
	JANE (JANZ)	— JAYNZ	Dst ≠ Src	$\overline{Z}$	1011
Signed Compares	JALT	JAXLE	Dst < Src	$(N \cdot V) + (\overline{N}\ \overline{V})$	0100
	JALE	—	Dst ≤ Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V) + Z$	0110
	JAGT	—	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	JAGE	JAXGT	Dst ≥ Src	$(N \cdot V) + (\overline{N}\ \overline{V})$	0101
	JAEQ (JAZ)	— JAYZ	Dst = Src	Z	1010
	JANE (JANZ)	— JAYNZ	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	JAZ (JAEQ)	JAYZ	Result = 0	Z	1010
	JANZ (JANE)	JAYNZ	Result ≠ 0	$\overline{Z}$	1011
	JAP	—	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	JAN	JAXZ	Result is negative	N	1110
	JANN	JAXNZ	Result is nonnegative	$\overline{N}$	1111
General Arithmetic	JAZ (JAEQ)	JAYZ	Result is 0	Z	**1010**
	JANZ (JANE)	JAYNZ	Result ≠ 0	$\overline{Z}$	**1011**
	JAC (JALO) (JAB)	JAYN	Carry set on result	C	**1000**
	JANC (JAHS) (JANB)	JAYNN	No carry on result	C	**1001**
	JAB (JALO) (JAC)	JAYN	Borrow set on result	C	**1000**
	JANB (JAHS) (JANC)	JAYNN	No borrow on result	$\overline{C}$	**1001**
	JAV †	JAXN	Overflow on result	V	**1100**

Note: A mnemonic code in parentheses is an alternate code for the preceding code.

Key: † Also used for window clipping
+ Logical OR
· Logical AND
— Logical NOT

Syntax

JR*condition Address*

Execution

If condition *true,* then offset + PC' → PC
If condition *false,* then go to next instruction

Instruction Words

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
1	1	0	0	code	offset

Fields

code is a 4-bit digit that identifies the condition for the jump within the opcode. (See the condition codes table below.)

Description

JR*condition* conditionally jumps to an address that is relative to the current PC. *Condition* is part of a mnemonic; it represents the condition for the jump. For example, if *condition* is UC, the instruction is JRUC. (See the conditions and codes listed below.) If the condition is **true,** the TMS34020 jumps to a new location. The assembler calculates the new address by adding the address of the next instruction (PC') to the signed word offset. The TMS34020 then continues execution from this point. If the condition is **false,** the TMS34020 skips the jump and continues execution at the next sequential instruction.

The *Address* operand is a 32-bit relative address. The assembler calculates the offset as (Address – PC')/16 (where PC' is the address of the instruction word immediately following the jump instruction) and inserts the resulting 8-bit offset into the opcode. The range for this form of the JR*condition* instruction is ±128 words (excluding 0).

If the offset is outside the range of ±128 words, the assembler automatically substitutes the longer form of the JR*condition* instruction. If the offset is 0, the assembler substitutes a NOP. The assembler does not accept an address that is externally defined or an address that is relative to a different section than the PC. Note that the 4 LSBs of the PC are always 0 (word aligned).

The JR*condition* instructions are often used with the CMP and CMPI instructions. The JRV and JRNV instructions can also be used to detect window violations or CPW status.

Machine States

1 if no jump, else 2

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Flags for Branch			Code	Flags for Branch		
	NCZV	N C Z V	N C Z V		NCZV	NCZV	NCZV
JRUC HERE	x x x x			JRC HERE	x 1 x x		
JRP HERE	0 x 0 x			JRNC HERE	x 0 x x		
JRLS HERE	x x 1 x	x 1 x x		JRZ HERE	x x 1 x		
JRHI HERE	x 0 0 x			JRNZ HERE	x x 0 x		
JRLT HERE	0 x x 1	1 x x 0		JRV HERE	x x x 1		
JRGE HERE	0 x x 0	1 x x 1		JRNV HERE	x x x 0		
JRLE HERE	0 x x 1	1 x x 0	x x 1 x	JRN HERE	1 x x x		
JRGT HERE	0 x 0 0	1 x 0 1		JRNN HERE	0 x x x		

Note that the TMS34020 jumps when any one or more of the *Flags for Branch* listed above are set as indicated.

Condition Codes

	Mnemonic				
	Non XY	**XY**	**Result of Compare**	**Status Bits**	**Code**
Unconditional Compares	JRUC	—	Unconditional	Don't care	0000
Unsigned Compares	JRLO (JRC) (JRB)	— JRYN	Dst lower than Src	C	1000
	JRLS	JRYLE	Dst lower or same as Src	C + Z	0010
	JRHI	JRYGT	Dst higher than Src	$\overline{C} \cdot \overline{Z}$	0011
	JRHS (JRNC) (JRNB)	JRYNN	Dst higher or same as Src	$\overline{C}$	1001
	JREQ (JRZ)	— JRYZ	Dst = Src	Z	1010
	JRNE (JRNZ)	— JRYNZ	Dst ≠ Src	$\overline{Z}$	1011
Signed Compares	JRLT	JRXLE	Dst < Src	$(N \cdot V) + (\overline{N}\ \overline{V})$	0100
	JRLE	—	Dst ≤ Src	$(N \cdot \overline{V}) + (\overline{N} \cdot V) + Z$	0110
	JRGT	—	Dst > Src	$(N \cdot V \cdot \overline{Z}) + (\overline{N} \cdot \overline{V} \cdot \overline{Z})$	0111
	JRGE	JRXGT	Dst ≥ Src	$(N \cdot V) + (\overline{N}\ \overline{V})$	0101
	JREQ (JRZ)	— JRYZ	Dst = Src	Z	1010
	JRNE (JRNZ)	— JRYNZ	Dst ≠ Src	$\overline{Z}$	1011
Compare to Zero	JRZ (JREQ)	JRYZ	Result = 0	Z	1010
	JRNZ (JRNE)	JRYNZ	Result ≠ 0	$\overline{Z}$	1011
	JRP	—	Result is positive	$\overline{N} \cdot \overline{Z}$	0001
	JRN	JRXZ	Result is negative	N	1110
	JRNN	JRXNZ	Result is nonnegative	$\overline{N}$	1111
General Arithmetic	JRZ (JREQ)	JRYZ	Result is 0	Z	**1010**
	JRNZ (JRNE)	JRYNZ	Result ≠ 0	$\overline{Z}$	**1011**
	JRC (JRLO) (JRB)	JRYN	Carry set on result	C	**1000**
	JRNC (JRHS) (JRNB)	JRYNN	No carry on result	C	**1001**
	JRB (JRLO) (JRC)	JRYN	Borrow set on result	C	**1000**
	JRNB (JRHS) (JRNC)	JRYNN	No borrow on result	$\overline{C}$	**1001**
	JRV †	JRXN	Overflow on result	V	**1100**

Note: A mnemonic code in parentheses is an alternate code for the preceding code.

Key: † Also used for window clipping
+ Logical OR
· Logical AND
— Logical NOT

Syntax

JRcondition *Address*

Execution

If condition *true,* then offset + PC' → PC
If condition *false,* then go to next instruction

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	code				0	0	0	0	0	0	0	0
offset															

Fields

code is a 4-bit digit that identifies the condition for the jump within the opcode. (See the condition codes on page 13-138.)

Description

The JR*condition* instruction conditionally jumps to an address that is relative to the current PC. The *condition* is part of a mnemonic that represents the condition for the jump; for example, if *condition* is UC, then the instruction is JRUC. (See the condition mnemonics and codes listed in on page 13-138.) If the specified condition is **true,** the TMS34020 jumps to a new location. The assembler calculates the address of this location by adding the address of the next instruction (PC') to the signed word offset. The TMS34020 then continues execution from this point. If the specified condition is **false,** the TMS34020 skips the jump and continues execution at the next sequential instruction.

The *Address* operand in the syntax represents the 32-bit relative address. The assembler calculates the offset as (Address – PC')/16 (where PC' is the address of the instruction word immediately following the jump instruction) and inserts the resulting offset into the second instruction word of the opcode. The range for this form of the JR*condition* instruction is –32,768 to +32,767 words (excluding 0).

If the offset is 0, the assembler substitutes a NOP instruction. If the address is out of range, the assembler uses the JA*condition* instruction instead. The assembler does not accept an address that is externally defined or an address that is relative to a different section than the PC. Note that the 4 LSBs of the program counter are always 0 (word aligned).

The JR*condition* instructions are commonly used in conjunction with the CMP and CMPI instructions. The JRV and JRNV instructions can also be used to detect window violations or CPW status.

Machine States

2 if no jump, else 3

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Flags for Branch			Code	Flags for Branch		
	NCZV	N C Z V	NCZV		NCZV	NCZV	NCZV
JRUC HERE	x x x x			JRV HERE	x x x 1		
JRP HERE	0 x 0 x			JRNZ HERE	x x 0 x		
JRLS HERE	x x 1 x	x 1 x x		JRNN HERE	0 x x x		
JRHI HERE	x 0 0 x			JRNV HERE	x x x 0		
JRLT HERE	0 x x 1	1 x x 0		JRN HERE	1 x x x		
JRGE HERE	0 x x 0	1 x x 1		JRB HERE	x 1 x x		
JRLE HERE	0 x x 1	1 x x 0	x x 1 x	JRNB HERE	x 0 x x		
JRGT HERE	0 x 0 0	1 x 0 1		JRLO HERE	x 1 x x		
JRC HERE	x 1 x x			JRHS HERE	x 0 0 x	x x 1 x	
JRNC HERE	x 0 x x			JRNE HERE	x x 0 x		
JRZ HERE	x x 1 x			JREQ HERE	x x 1 x		

Note that the TMS34020 jumps when any one or more of the *Flags for Branch* listed above are set as indicated.

Syntax

JUMP *Rs*

Execution

Rs → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	1	1	R	Rd			

Description

JUMP jumps to the address contained in the source register. The TMS34020 sets the 4 LSBs of the program counter to 0 (word aligned). This instruction can be used in conjunction with the GETPC and/or EXGPC instructions.

Machine States

2

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After
	A1	**PC**	**PC**
`JUMP A1`	00001EE0h	00555550h	00001EE0h
`JUMP A1`	00001EE5h	00555550h	00001EE0h
`JUMP A1`	FFFFFFFFh	00555550h	FFFFFFF0h

Syntax

LINE {**0** | **1**}

Execution

The two execution algorithms for the LINE instruction are explained below. These algorithms are similar, varying only in their treatment of *d*=0.

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	Z	0	0	1	1	0	1	0

Fields

The assembler sets bit 7 in the instruction word (the Z bit) to 0 or 1, depending on which LINE algorithm you select:

Z=0 selects algorithm 0
Z=1 selects algorithm 1

Description

LINE is an alternative implementation of the FLINE instruction (page 13-121). The major differences between LINE and FLINE are the following:

- LINE is slower than FLINE.
- LINE supports windowing (FLINE does not).
- LINE requires B2 to be in XY format.

LINE performs the inner loop of Bresenham's line-drawing algorithm. This type of line draw plots a series of points (x_i,y_i) either diagonally or laterally with respect to the previous point. Movement from pixel to pixel always proceeds in a dominant lateral direction. The algorithm may or may not also increment in the direction with the smaller dimension (this produces a diagonal movement). Two XY-format registers supply the XY increment values for the two possible movements. The LINE instruction maintains a decision variable, *d*, that acts as an error term, controlling movement in either the dominant or diagonal direction. The algorithm operates in one of two modes, depending on how the condition *d*=0 is treated.

During LINE execution, some portion of a line $[(x_0,y_0)(x_1,y_1)]$ is drawn. The line is drawn so that the axis with the largest extent has dimension *a*, and the axis with the least extent has dimension *b*. Thus, *a* is the larger (in absolute terms) of $y_1 - y_0$ or $x_1 - x_0$, and *b* is the smaller of the two. This means that $a \geq b \geq 0$.

The LINIT instruction provides a simple method for setting up the LINE instruction's implied operands. For more information, refer to the LINIT instruction (page 13-146).

The following values must be supplied to draw a line from (x_0,y_0) to (x_1,y_1):

- Set the value in DADDR to be the **linear** address of the first pixel in the line at (x_0,y_0).
- Use the line endpoints to determine the major and minor dimensions (*a* and *b*, respectively) for the line draw; then set the DYDX register to this value (*b*:*a*).

- Place the signed XY increment for a movement in the diagonal (or minor) direction ($d \geq 0$ for Z=0, $d > 0$ for Z=1) in the INC1 register.
- Place the signed XY increment for a movement in the dominant (or major) direction ($d < 0$ for Z=0, $d \leq 0$ for Z=1) in the INC2 register.
- Set the initial value of the decision variable in register B0 to $2b - a$.
- Set the initial count value in the COUNT register to $a + 1$.
- Set the COLOR1 and COLOR0 registers.
- Set the PATTERN register to the required pattern.

  LINE handles the contents of PATTERN in the same way as FLINE (unlike PFILL XY). With LINE, the first pixel drawn is controlled by bit 0 of the PATTERN register.

  The PATTERN register contains a 32-bit repeating line-style pattern. If bit 0 of PATTERN is *0*, then the first pixel drawn by LINE is a COLOR0 pixel. If bit 0 of PATTERN is *1*, then the first pixel drawn by LINE is a COLOR1 pixel. The second pixel drawn by LINE is controlled by bit 1 of B13, and so on. If the line is longer than 32 pixels, the PATTERN is reused cyclically; therefore, the 33rd pixel on the line is once again controlled by bit 0 of PATTERN. As each pixel is drawn, the contents of PATTERN are rotated right (circular shifted) by 1 bit. The LSB of the rotated pattern controls the next pixel the instruction puts out.

  If PATTERN contains all 1s, the line is drawn in a solid color using the replicated pixel value contained in COLOR1; if PATTERN contains all 0s, the line is drawn in a solid color using COLOR0.

The LINE instruction may use one of two algorithms, depending on the value of Z.

Algorithm 0 (Z=0):

```
While COUNT > 0
        COUNT = COUNT – 1
        Draw the next pixel
        If d ≥ 0
            d = d + 2b – 2a
            POINTER = POINTER + INC1
        Else d = d + 2b;
            POINTER = POINTER + INC2
```

Algorithm 1 (Z=1):

```
While COUNT > 0
        COUNT = COUNT – 1
        Draw the next pixel
        If d > 0
            d = d + 2b – 2a
            POINTER = POINTER + INC1
        Else d = d + 2b;
            POINTER = POINTER + INC2
```

Implied Operands

Register	Name	Format	Description
B0 †	SADDR	Integer	Decision variable (d)
B2 †	DADDR	XY	Starting point (y_i, x_i), usually (y_0, x_0)
B3 ‡	DPTCH	Linear	Destination pitch
B4	OFFSET	Linear	Screen origin (0,0)
B5	WSTART	XY	Window starting corner
B	WEND	XY	Window ending corner
B7	DYDX	XY	(b: a) = Minor: major dimension
B8	COLOR0	Pixel	COLOR0
B9	COLOR1	Pixel	COLOR1
B10 †	COUNT	Integer	Loop count
B11	INC1	XY	Minor axis (diagonal) increment
B12	INC2	XY	Major axis (dominant) increment
B13	PATTERN	Pattern	Pattern register

† These registers are changed by instruction execution.

‡ Required only when pitch is an arbitrary, nonpower of 2.

Address	Name	Description and Elements (Bits)
C00000B0h	CONTROL	PPOP Pixel-processing operations (22 options) W Window-clipping operation T Transparency operation TM Sets transparency mode
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the LINE instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT (13-177). Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Pixel Processing

PPOP[[CONTROL]] specifies the operation to be applied to the pixel as it is written. There are 22 operations; the default case at reset is the pixel-processing *replace* (S → D) operation. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window Checking

Window clipping or picking is selected by setting W[[CONTROL]] to the appropriate value. The WSTART and WEND registers define the window in XY-coordinate space. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Transparency

You can enable transparency for this instruction by setting T[[CONTROL]] to 1. Select 1 of 3 transparency options by setting TM[[CONTROL]]. For more information, refer to Section 12.9, Transparency, on page 12-36.

Plane Masking

The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Interrupts

LINE may be interrupted after every pixel in the line draw except for the last pixel. Note that a LINE instruction that is aborted because of window checking options 1 or 2 does not decrement the PC before pushing it on the stack. In this case, the LINE is not resumed after returning from the interrupt service routine. For more information, refer to Section 6.6, Interrupting Graphics Instructions, on page 6-13.

Machine States

Refer to Section 15.1 on page 15-2.

Status Bits

N Undefined
C Undefined
Z Undefined
V Set depending upon window operation

Example

Refer to example for FLINE on page 13-124.

Syntax

LINIT

Execution

2b – a → B0
(b:a) → B7
a + 1 → B10
minor axis (diagonal) XY increment → B11
major axis (dominant) XY increment → B12

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	0	1	0	1	0	1	1	1

Description

This line initialization instruction uses the start and end points for the line to set up the implied B-file registers as required by the LINE and FLINE instructions. The startpoint is assumed to be in B2 and the endpoint in B7. Note that FLINE expects a linear DADDR, so when LINIT is used in conjunction with FLINE, CVXYL should be executed on DADDR before executing FLINE.

The V bit in status is set to indicate if both start and end points lie within the window. The N and Z bits are set on the X and Y zero detects on the difference between the two points. This allows for detection of the special cases of horizontal and vertical lines as well as single pixel lines. The C bit is set to indicate that the line may be trivially rejected.

For additional information, refer to Section 12.4, Line Instructions, on page 12-7; FLINE on page 13-121, LINE on page 13-142, subsection 12.7.5, Window Checking for Line Instructions, on page 12-23, and subsection 12.7.5.2, Using LINIT and FLINE for Preclipping Line Drawing, on page 12-26.

Implied Operands

Register	Name	Format	Description	
B0	SADDR	Linear	Decision variable	(output)
B2	DADDR	XY	Starting point (y_0, x_0)	(input)
B7	DYDX	XY	Ending point (y_1, x_1)	(input)
B7	DYDX	XY	b:a *minor:major line dimensions*	(output)
B10	COUNT	Integer	Count	(output)
B11		XY	Minor axis (diagonal increment)	(output)
B12		XY	Major axis (dominant increment)	(output)

Machine States

9

Status Bits

N = 1 if x0 = x1 (vertical line)
C = 1 if (CPW(y_0, x_0) & CPW(y_1,x_1)) is nonzero (line lies entirely outside window)
Z = 1 if $y_0 = y_1$ (horizontal line)
V = 1 if (y_0,x_0) or (y_1,x_1) lie outside the window (line lies partially outside window)

Examples

Refer to Section 15.1 on page 15-2.

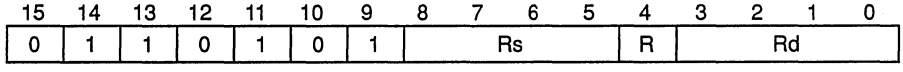
Syntax

LMO *Rs, Rd*

Execution

31 – (bit number of leftmost 1 in Rs) → Rd

Instruction Words



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	1	Rs				R	Rd			

Description

LMO locates the leftmost (most significant) 1 in the source register. It then loads the 1s complement of the **bit number** of the leftmost-1 bit into the 5 LSBs of the destination register. The 27 MSBs of the destination register are loaded with 0s. Bit 31 of Rs is the MSB (leftmost) and bit 0 is the LSB. If the source register contains all 0s, then the destination register is loaded with all 0s and status bit Z is set.

You can normalize the contents of the source register by following the LMO instruction with an RL Rs,Rd instruction, where Rs is the destination register of the LMO instruction and Rd is the source register.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the source register contents are 0, *0* otherwise
V Unaffected

Examples

Code	Before A0	After N C Z V	A1
`LMO A0,A1`	00000000h	x x 1 x	00000000h
`LMO A0,A1`	00000001h	x x 0 x	0000001Fh
`LMO A0,A1`	00000010h	x x 0 x	0000001Bh
`LMO A0,A1`	08000000h	x x 0 x	00000004h
`LMO A0,A1`	80000000h	x x 0 x	00000000h

Syntax

MMFM *Rp, register list*

Execution

For each register R*n* in the register list,
32 bits of data at the address specified in Rp → R*n*
Rp + 32 → Rp

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	0	1	R	Rp			
binary representation of the register list															

Description

MMFM loads the contents of a specified list of *either* A- or B-file registers (not both) from a block of memory.

- *Rp* is a register that points to the first location in the block of memory.
- The *register list* is a list of registers separated by commas (such as A0, A1, A9). These are the registers that MMFM loads new values into.

MMFM and MMTM are complementary instructions. MMFM reads a list of A- or B-file registers *from* memory, and MMTM writes a list of A- or B-file registers *to* memory. These instructions can be used to save and restore the contents of registers during, for example, subroutine calls and interrupts. All 32 bits of each register in the list are saved and then restored.

MMFM and MMTM use *Rp* as a pointer register. Rp acts as a stack pointer; MMTM pushes a list of registers onto a stack, and MMFM pops a list of registers from a stack. The stack grows toward lower addresses, similar to the way the SP register points to the system stack. The Rp can be any register that is not included in the register list and that is in the same file as the registers in the list. (SP can be treated as belonging to either register file.)

MMFM and MMTM always leave the Rp register adjusted to point to the new top of the stack following a push or pop operation. MMTM predecrements Rp by 32 prior to pushing each register in the list onto the stack. The last register pushed on the stack by MMTM is the highest numbered register in the list. MMFM postincrements Rp by 32 after popping each register in the list from the stack. The first register popped off the stack by MMFM is the highest numbered register in the list.

If SP is used as the Rp register, MMTM and MMFM push and pop register values to and from the system stack and leave SP correctly adjusted to point to the new top of the system stack.

Rp and the registers in the list must all be in the same register file. The assembler allows the registers in the list to be specified in any order; the highest numbered register is always restored first (that is, the value at the top of the stack—the lowest address in the stack—is loaded into the highest numbered register). Don't include Rp as one of the registers in the register list, because this produces unpredictable results. For the best performance, the original

contents of Rp should be aligned on a long-word boundary; the alignment of Rp affects the instruction timing as indicated in **Machine States,** below.

The second word of the MMFM instruction is a binary-mask representation of the registers in the list. The R bit (bit 4) in the first instruction word indicates which register file is affected; the bits that are set to 1 in the mask indicate which registers are restored. The bit assignments in the mask are

	SP	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
or	SP	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
(MSB)	15															0 (LSB)

Machine States

Refer to Section 15.1 on page 15-2.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

This example restores several B-file registers:

```
MMFM  B0,B1,B2,B3,B7,B12,B13,B14,SP
```

This instruction uses register B0 as the stack pointer. Assume that B0 = 00010000h; this is the address of the top of the stack. MMFM moves the data at this location into the LSW of the SP (which is the highest order register listed in this example). Assume that memory contains the following values before instruction execution:

Address	Data	Address	Data
000100F0h	1111h	00010070h	CCCCh
000100E0h	B1B1h	00010060h	BCBCh
000100D0h	2222h	00010050h	DDDDh
000100C0h	0B2B2h	00010040h	BDBDh
000100B0h	3333h	00010030h	EEEEh
000100A0h	B3B3h	00010020h	BEBEh
00010090h	7777h	00010010h	FFFFh
00010080h	B7B7h	00010000h	BFBFh

After the MMFM instruction is executed, the registers in the list have the following values:

B0 = 00010100h
B1 = 1111B1B1h
B2 = 2222B2B2h
B4 = 3333B3B3h
B8 = 7777B7B7h
B12 = CCCCBCBCh
B13 = DDDDBDBDh
B14 = EEEEBEBEh
SP = FFFFBFBFh

The other B-file registers (which weren't specified in the register list) are not affected by this instruction. Note that B0 now contains the value 10100h; the last part of the data that was restored was for B1, and B0 points to the word past that data.

Syntax

MMTM *Rp, register list*

Execution

For each register R*n* in the register list,

Rp – 32 → Rp

32 bits of data at the address specified in R*n* → Rp

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	0	0	R	Rp			
binary representation of the register list															

Description

MMTM stores the contents of a specified list of *either* A- or B-file registers (not both) in memory.

- *Rp* is a register that points to the first location in a block of memory.
- The *register list* is a list of registers that are separated by commas (such as A0, A1, A9). These are the registers that MMTM stores in memory.

MMTM and MMFM are complementary instructions. MMFM reads a list of A- or B-file registers *from* memory, and MMTM writes a list of A- or B-file registers *to* memory. These instructions can be used to save and restore the contents of registers during, for example, subroutine calls and interrupts. All 32 bits of each register in the list are saved and then restored.

MMTM and MMFM use *Rp* as a pointer register. Rp acts as a stack pointer; MMTM pushes a list of registers onto a stack, and MMFM pops a list of registers from a stack. The stack grows toward lower addresses, similar to the way the SP register points to the system stack. The Rp can be any register that is not included in the register list and that is in the same file as the registers in the list. (SP can be treated as belonging to either register file.)

MMTM and MMFM always leave the Rp register adjusted to point to the new top of the stack following a push or pop operation. MMTM predecrements Rp by 32 prior to pushing each register in the list onto the stack. The last register pushed on the stack by MMTM is the highest numbered register in the list. MMFM postincrements Rp by 32 after popping each register in the list from the stack. The first register popped off the stack by MMFM is the highest numbered register in the list.

If SP is used as the Rp register, MMTM and MMFM push and pop register values to and from the system stack and leave SP correctly adjusted to point to the new top of the system stack.

When MMTM execution is complete, the contents of the lowest order register in the list reside at the highest address in the memory stack, and Rp points to the address of the highest order register in the list.

Rp and the registers in the list must all be in the same register file. The assembler allows the registers in the list to be specified in any order; the lowest order register is always saved first. Don't include Rp as one of the registers in the register list, because this produces unpredictable results. For the best performance, the original contents of Rp should be aligned on a long-word

boundary; the alignment of Rp affects the instruction timing as shown in **Machine States,** below.

The second word of the MMTM instruction is a binary-mask representation of the registers in the list. The R bit (bit 4) in the first instruction word indicates which register file is affected; the bits that are set to 1 in the mask indicate which registers are restored. The bit assignments in the mask are

	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	SP
or	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	SP

(MSB) 15 **0 (LSE**

Machine States

Refer to Section 15.1 on page 15-2.

Status Bits

N Set to the sign of the result of 0 – Rp. (This value is typically 1 if the original contents of Rp are positive; otherwise, it is 0. The only exceptions to this are when Rp=80000000h and N is set to 0, and when Rp=0 and N is set to 1.)

C Unaffected

Z Unaffected

V Unaffected

Examples

This example saves the values of several A-file registers in memory:

```
MMTM   A1,A0,A2,A4,A8,A12,A13,A14,SP
```

This instruction uses register A1 as the stack pointer. Assume that A1 = 00100000h before instruction execution; this value is decremented by 32 to point to the address where the contents of A0 (the lowest order register in the list) are stored. Assume that the registers in the list contain the following values before instruction execution:

A0 = 0000A0A0h	A12 = CCCCACACh
A2 = 2222A2A2h	A13 = DDDDADADh
A4 = 4444A4A4h	A14 = EEEEAEAEh
A8 = 8888A8A8h	SP = FFFFAFAFh

MMTM saves these register values in memory as shown below:

Address	Data	Address	Data
000FFF00h	AFAFh	000FFF80h	A8A8h
000FFF10h	FFFFh	000FFF90h	8888h
000FFF20h	AEAEh	000FFFA0h	A4A4h
000FFF30h	EEEEh	000FFFB0h	4444h
000FFF40h	ADADh	000FFFC0h	A2A2h
000FFF50h	DDDDh	000FFFD0h	2222h
000FFF60h	ACACh	000FFFE0h	A0A0h
000FFF70h	CCCCh	000FFFF0h	0000h

After instruction execution, register A1 = 000FFF00h; this is the address of the last portion of register data that is saved.

Syntax

MODS *Rs, Rd*

Execution

Rd mod Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	0	Rs				R	Rd			

Description

MODS performs a 32-bit signed divide of the 32-bit dividend in the destination register by the 32-bit divisor in the source register, and returns a 32-bit remainder in the destination register. Regardless of whether the result is positive or negative, the magnitude of the remainder is always the same as it would be for a positive dividend and divisor. The remainder is the same sign as the dividend. The original contents of the destination register are always overwritten.

Rs and Rd must be in the same register file.

Machine States

40
41 if result = 80000000
3 if Rs = 0

Status Bits

N *0* if Rs is *0*
1 if Rs is not *0* and the result in Rd is –ve
0 if Rs is not *0* and the result in Rd is + ve

C Unaffected

Z *0* if Rs is *0*
1 if Rs is not *0* and the result in Rd is *0*
0 if Rs is not *0* and the result in Rd is not *0*

V If Rs is *0*, then V = 1, otherwise V = 0

Examples

Code	Before		After	
	A0	A1	N C Z V	A1
MODS A0,A1	00000000h	00000000h	0 x 0 1	00000000h
MODS A0,A1	00000000h	00000007h	0 x 0 1	00000007h
MODS A0,A1	00000000h	FFFFFFF9h	0 x 0 1	FFFFFFF9h
MODS A0,A1	00000004h	00000008h	0 x 1 0	00000000h
MODS A0,A1	00000004h	00000007h	0 x 0 0	00000003h
MODS A0,A1	00000004h	00000000h	0 x 1 0	00000000h
MODS A0,A1	00000004h	FFFFFFF9h	1 x 0 0	FFFFFFFDh
MODS A0,A1	00000004h	FFFFFFF8h	0 x 1 0	00000000h
MODS A0,A1	FFFFFFFCh	00000008h	0 x 1 0	00000000h
MODS A0,A1	FFFFFFFCh	00000007h	0 x 0 0	00000003h
MODS A0,A1	FFFFFFFCh	00000000h	0 x 1 0	00000000h
MODS A0,A1	FFFFFFFCh	FFFFFFF9h	1 x 0 0	FFFFFFFDh
MODS A0,A1	FFFFFFFCh	FFFFFFF8h	0 x 1 0	00000000h

Syntax

MODU *Rs, Rd*

Execution

Rd mod Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	1	1	Rs				R	Rd			

Description

MODU performs a 32-bit unsigned divide of the 32-bit dividend in the destination register by the 32-bit divisor in the source register, and returns a 32-bit remainder in the destination register. The original contents of the destination register are always overwritten.

Rs and Rd must be in the same register file.

Machine States

35
3 if Rs = 0

Status Bits

N Unaffected
C Unaffected
Z *0* if Rs=*0*, *1* if quotient is 0, *0* otherwise
V *1* if divisor Rs equals *0*, *0* otherwise

Examples

Code	Before		After	
	A0	A1	N C Z V	A1
`MODU A0,A1`	00000000h	00000000h	x x 0 1	00000000h
`MODU A0,A1`	00000000h	00000007h	x x 0 1	00000007h
`MODU A0,A1`	00000000h	FFFFFFF9h	x x 0 1	FFFFFFF9h
`MODU A0,A1`	00000004h	00000008h	x x 1 0	00000000h
`MODU A0,A1`	00000004h	00000007h	x x 0 0	00000003h
`MODU A0,A1`	00000004h	00000000h	x x 1 0	00000000h
`MODU A0,A1`	00000004h	FFFFFFF9h	x x 0 0	00000001h

MOVB Instructions

The **MOVB** instruction is a special form of the MOVE instruction that restricts the field size of the move to 8 bits. MOVB moves a single byte from its source to a specified destination.The following list describes characteristics common to all **MOVB** instructions.

- MOVB instructions move data from a register to memory, from memory to a register, and between memory locations, but they do not move data between registers.
- A byte can begin on any bit boundary in memory, although sequential byte moves are more efficient if the byte addresses are aligned on even 8-bit boundaries.
- All addresses are bit addresses.
- When a byte is moved into a register, the byte's LSB coincides with the register's LSB; the byte is sign-extended into the 24 MSBs of the register.
- If the source data is in a register, only the LSbyte is used.
- Rs and Rd must be in the same register file.
- The **status bits** are unaffected unless otherwise noted in the individual descriptions.
- For **machine states** information, refer to Section 15.2 on page 15-10.

Table 13–4. Summary of Operand Formats for the MOVB Instruction

		Destination			
		Rd	**Rd*	**Rd(DOffset)*	@DAddress
Source	*Rs*		√	√	√
	**Rs*	√	√		
	**Rs(SOffset)*	√		√	
	@SAddress	√			√

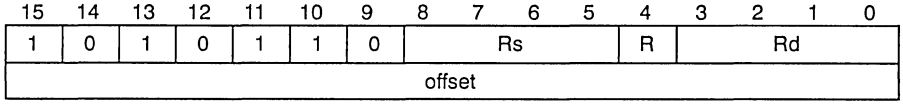
The MOVB instruction has nine operand combinations, which are listed below with their corresponding instruction words and descriptions.

MOVB *Rs, *Rd*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	1	0	Rs				R	Rd			

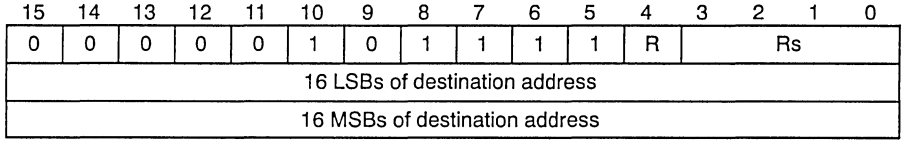
Moves the LSbyte of Rs to the memory address contained in the Rd.

MOVB *Rs, *Rd(offset)*



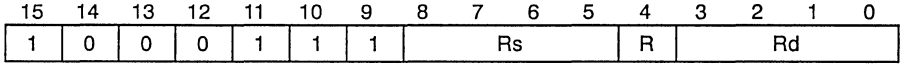
Moves the LSbyte of Rs to the destination memory address. The destination address is formed by adding the signed 16-bit offset to the contents of Rd.

MOVB *Rs, @DAddress*



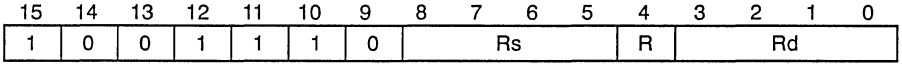
Moves the LSbyte of Rs to the destination address.

MOVB *Rs, Rd



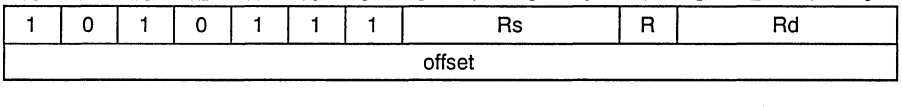
Moves a byte from the source address contained in Rs into Rd. This instruction also compares the source data to 0.† See **Status Bits** for more information.

MOVB *Rs, *Rd



Moves a byte from the source address contained in Rs to the destination address contained in Rd.

MOVB *Rs(offset), Rd



Moves a byte from the source address to the destination register. The source data's memory address is a bit address and is formed by adding the signed 16-bit offset to the contents of Rs. This instruction also compares the source data to 0. †See **Status Bits** for more information.

MOVB *Rs(SOffset), *Rd(DOffset)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	Rs				R	Rd			
source offset															
destination offset															

Moves a byte from the source address to the destination address. Both addresses are formed by adding the source or destination signed 16-bit offset to the contents of Rs or Rd, respectively.

MOVB @SAddress, *Rd

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	1	1	1	R	Rd			
16 LSBs of source address															
16 MSBs of source address															

Moves a byte from the source address to Rd. This instruction also compares the source data to 0. †See **Status Bits** for more information.

MOVB @SAddress, @DAddress

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0
16 LSBs of source address															
16 MSBs of source address															
16 LSBs of destination address															
16 MSBs of destination address															

Moves a byte from the source address to the destination address.

†The following status bits information applies only to MOVB instructions with these addressing modes:

MOVB *Rs, Rd
MOVB *Rs(offset), Rd
MOVB @SAddress, Rd

Status Bits

N *1* if the sign-extended data moved is negative, *0* otherwise
C Unaffected
Z *1* if the sign-extended data moved is 0, *0* otherwise
V *0*

MOVB Examples

Example 1

Assume that memory contains the following values:

Address	Data
1000h	0000h
1010h	0000h

Code		Before	After		
		A0	A1	@1000h	@1010h
MOVB	A0,*A1	89ABCDEFh	00001000h	00EFh	0000h
MOVB	A0,*A1	89ABCDEFh	00001009h	DE00h	0001h
MOVB	A0,*A1(1)	89ABCDEFh	00001000h	01DEh	0000h
MOVB	A0,*A1(−1)	89ABCDEFh	00001001h	00EFh	0000h
MOVB	A0,@1000h	89ABCDEFh	xxxxxxxxx	00EFh	0000h
MOVB	A0,@100Ch	89ABCDEFh	xxxxxxxxx	F000h	000Eh

Example 2

Assume that memory contains the following values:

Address	Data
1000h	00EFh
1010h	89ABh

Code		Before	After				
		A0	A1	N	C	Z	V
MOVB	*A0,A1	00001000h	FFFFFFEFh	1	x	0	0
MOVB	*A0,A1	00001001h	00000077h	0	x	0	0
MOVB	*A0,A1	00001008h	00000000h	0	x	1	0
MOVB	*A0,A1	0000100Ch	FFFFFFB0h	1	x	0	0
MOVB	*A0(0),A1	00001000h	FFFFFFEFh	1	x	0	0
MOVB	*A0(8),A1	00001000h	00000000h	0	x	1	0
MOVB	*A0(−1),A1	0000100Dh	FFFFFFB0h	1	x	0	0
MOVB	@1000h,A1	xxxxxxxxx	FFFFFFEFh	1	x	0	0
MOVB	@100Ch,A1	xxxxxxxxx	FFFFFFB0h	1	x	0	0

Example 3

Assume that memory contains the following values:

Address	Data
1000h	CDEFh
1010h	89ABh
2000h	0000h
2010h	0000h

Code		Before		After	
		A0	A1	@2000h	@2010h
MOVB	*A0,*A1	00001000h	00002000h	00EFh	0000h
MOVB	*A0,*A1	00001000h	00002001h	01DEh	0000h
MOVB	*A0,*A1	00001000h	00002009h	DE00h	0001h
MOVB	*A0,*A1	00001001h	00002000h	00F7h	0000h
MOVB	*A0,*A1	00001001h	00002001h	01EEh	0000h
MOVB	*A0,*A1	0000100Ch	00002009h	7800h	0001h
MOVB	*A0(0),*A1(0)	00001000h	00002000h	00EFh	0000h
MOVB	*A0(12),*A1(9)	00001000h	00002000h	7800h	0001h
MOVB	@1000h,@2000h	xxxxxxxxx	xxxxxxxxx	00EFh	0000h
MOVB	@100Ch,@2009h	xxxxxxxxx	xxxxxxxxx	7800h	0001h

Syntax

MOVE *Rs, Rd*

Execution

Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	M	Rs				R	Rd			

Description

MOVE moves the 32 bits of data from the source register to the destination register. Note that this is not a field move; therefore, the field size has no effect. The source and destination registers can be any of the 31 locations in the on-chip register file. Note that this is the only MOVE instruction that allows the source and destination registers to be in different files. This instruction also performs an implicit compare to 0 of the register data.

Fields

The assembler sets bit 9 (the M bit) in the instruction word to specify whether the move is within a register file or whether it crosses the register files. The assembler sets M to 0 if the source and destination registers are in the same file; it sets M to 1 if the registers are in different files.

The assembler sets bit 4 (the R bit) in the instruction word to specify the file the registers are in. The assembler sets R to 0 if the registers are in file A; it sets R to 1 if the registers are in file B.

Note that when M=0, R specifies the register file for *both* registers; if M=1, R specifies the register file for the *source register.*

Machine States

1

Status Bits

N *1* if the 32-bit data moved is negative, *0* otherwise
C Unaffected
Z *1* if the 32-bit data moved is 0, *0* otherwise
V *0*

Examples

Code	Before A0	After A1	B1	NCZV
`MOVE A0, A1`	0000FFFFh	0000FFFFh	xxxxxxxxh	0x00
`MOVE A0, A1`	00000000h	00000000h	xxxxxxxxh	0x10
`MOVE A0, A1`	FFFFFFFFh	FFFFFFFFh	xxxxxxxxh	1x00
`MOVE A0, B1`	0000FFFFh	xxxxxxxxh	0000FFFFh	0x00
`MOVE A0, B1`	00000000h	xxxxxxxxh	00000000h	0x10
`MOVE A0, B1`	FFFFFFFFh	xxxxxxxxh	FFFFFFFFh	1x00

MOVE Instructions

The following list describes characteristics common to all **MOVE** instructions (***except* MOVE** *Rs, Rd*). For information on MOVE *Rs, Rd*, refer to page 13-158.

- The **MOVE** instruction moves a field of 1—32 bits, depending on the selected field size.The optional F parameter determines the field size and extension for the move.
  - **F=0** selects the field size of 0 (FS0).
  - **F=1** selects the field size of 1 (FS1).
  - The SETF instruction sets the field size and extension.
  - If you do not supply a value for F, MOVE uses the value of field 0.
- The field is right-justified within the source register.
- Rs and Rd must in the same register file.
- The **status bits** are unaffected unless otherwise noted in the individual descriptions.
- For **machine states** information, refer to Section 15.2 on page 15-10.
- The destination address is a bit address.

Table 13–5. Summary of Operand Formats for the MOVE Instruction

		Destination					
		Rd	**Rd*	**Rd+*	*–*Rd*	**Rd(DOffset)*	@DAddress
Source	*Rs*	√	√	√	√	√	√
	**Rs*	√	√				
	**Rs+*	√		√			
	*–*Rs*	√			√		
	**Rs*(SOffset)	√		√		√	
	@SAddress	√		√			√

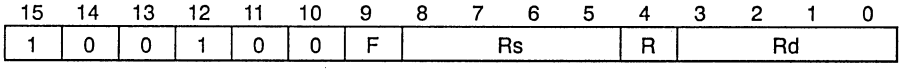
The **MOVE** instruction has 18 operand combinations, which are listed below with their corresponding instruction words and descriptions.

MOVE *Rs, *Rd [,F]*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	F	Rs				R	Rd			

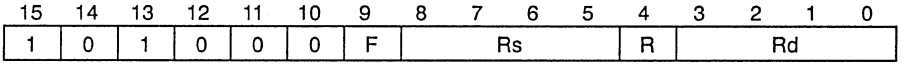
Moves a field from Rs to the address specified in Rd.

MOVE *Rs, *Rd+ [,F]



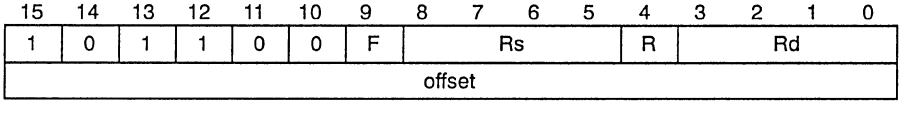
Moves a field from Rs to the address contained in the destination register. After the move, the contents of Rd are postincremented by the selected field size.

MOVE *Rs, –*Rd [,F]



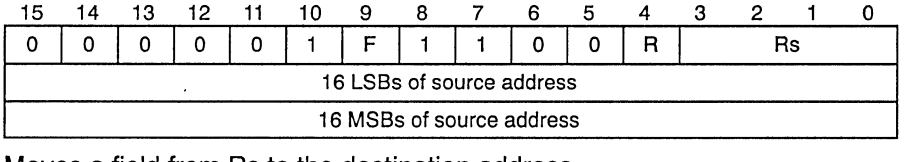
Moves a field from the Rs to the address contained in Rd. Before the move, the destination address is determined by subtracting the field size from the contents of Rd. (This value is also the final value for the register.)

MOVE *Rs, *Rd(offset) [,F]



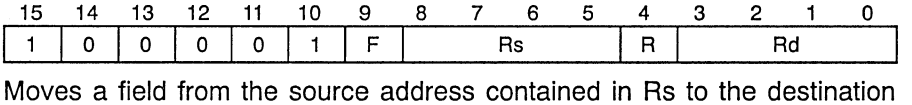
Moves a field from the Rs to the destination address. The destination address is formed by adding the signed 16-bit offset to the contents of Rd.

MOVE *Rs, @DAddress [,F]



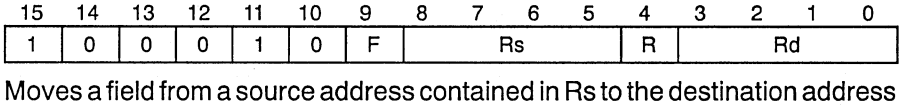
Moves a field from Rs to the destination address.

MOVE *Rs, Rd [,F]



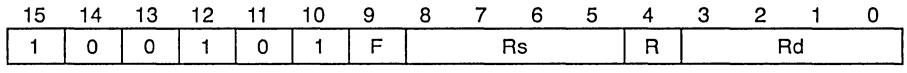
Moves a field from the source address contained in Rs to the destination address contained in Rd. When the field is moved into the destination register, it is right-justified and sign-extended or zero-extended to 32 bits (depending on the value of FE). This instruction also compares the source data to 0. † See **Status Bits** for more information.

MOVE *Rs, *Rd [,F]



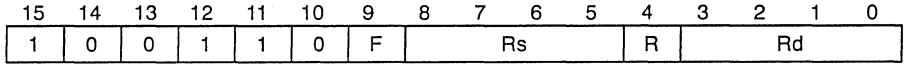
Moves a field from a source address contained in Rs to the destination address contained in Rd.

MOVE *Rs+, Rd [,F]



Moves a field from a source address into Rd. Rs contains the address of the field; after the move, the contents of the source register are incremented by the field size. When the field is moved into Rd, it is right-justified and sign- or zero-extended to 32 bits (depending on the value of the current FE bit). This instruction also performs an implicit compare to 0 of the field data. † See **Status Bits** for more information.

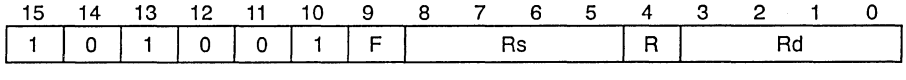
MOVE *Rs+, *Rd+ [,F]



Moves a field from one address to another. Rs contains the bit address of the field; Rd contains the bit address of the field's destination. After the move, the contents of both registers are incremented by the field size.

If Rs and Rd specify the same register, the data read from the location pointed to by the original contents of Rs is written to the location pointed to by the incremented value of Rs(Rd).

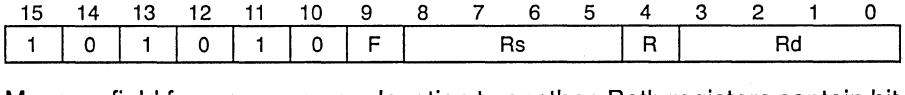
MOVE –*Rs, Rd [,F]



Moves a field from a source address into Rd. Rs contains a bit address; before the move, the contents of Rs are decremented by the field size to form the address of the field. (This value is also the final value for the register.) When the field is moved into Rd, it is right-justified and sign- or zero-extended to 32 bits (depending on the value of the current FE bit). This instruction also performs an implicit compare to 0 of the field data.

If Rs and Rd are the same register, the pointer information is overwritten by the data fetched. † See **Status Bits** for more information.

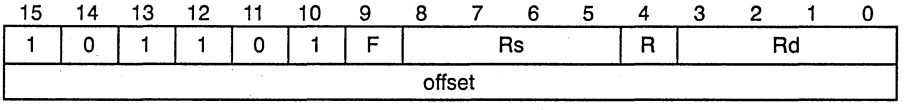
MOVE –*Rs, –*Rd [,F]



Moves a field from one memory location to another. Both registers contain bit addresses; before the move, the contents of both registers are decremented by the field size.

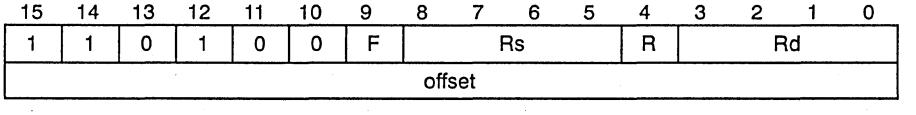
If Rs and Rd are the same register, then the final contents of the register are its original contents decremented by twice the field size.

MOVE *Rs(offset), Rd [,F]



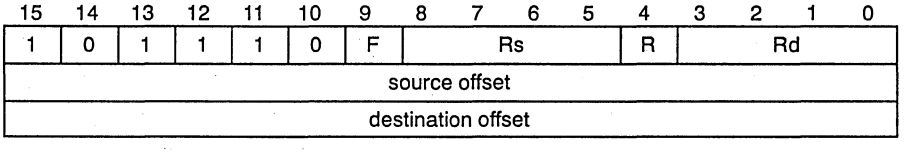
Moves a field from the source address into Rd. The source address is formed by adding a signed, 16-bit offset to the contents of Rs. When the field is moved into Rd, it is right-justified and sign- or zero-extended to 32 bits (depending on the value of the current FE bit). This instruction also performs an implicit compare to 0 of the field data. † See **Status Bits** for more information.

MOVE *Rs(offset), *Rd+ [,F]



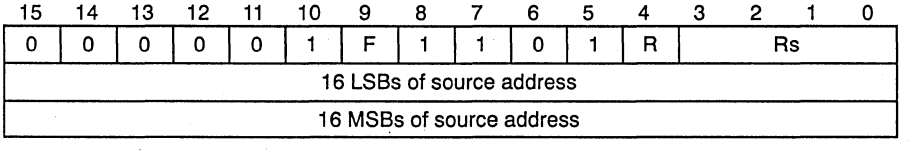
Moves a field from one memory location to another. The source address is formed by adding the contents of Rs to the signed 16-bit offset. Rd contains the address of the field's destination; after the move, the contents of Rd are incremented by the selected field size.

MOVE *Rs(SOffset), *Rd(DOffset) [,F]



Moves a field from one memory location to another. The source address is formed by adding a signed 16-bit offset to the contents of Rs. The destination address is formed by adding a signed 16-bit offset to the contents of Rd.

MOVE @SAddress, Rd [,F]



Moves a field from memory to the destination register. SAddress is a 32-bit address. When the field is moved into the destination register, it is right-justified

and sign- or zero-extended to 32 bits (depending on the selected value of FE). This instruction also compares the source data to 0. † See **Status Bits** for more information.

MOVE @*SAddress*, **Rd+ [,F]*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	F	0	0	0	0	R	Rs			
16 LSBs of source address															
16 MSBs of source address															

Moves a field from one location in memory to another. The source address is a 32-bit address; the destination address is specified by the contents of Rd. After the move, the contents of the destination register are incremented by the field size.

MOVE @*SAddress*, @*DAddress [,F]*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	1	1	0	0	0	0	0	0
16 LSBs of source address															
16 MSBs of source address															
16 LSBs of destination address															
16 MSBs of destination address															

Moves a field from one location in memory to another. Both addresses are 32-bit addresses.

† The following status bits information applies only to these MOVEs:
MOVE **Rs, Rd [,F]*
MOVE **Rs+, Rd [,F]*
MOVE *–*Rs, Rd [,F]*
MOVE **Rs(offset), Rd [,F]*
MOVE *@SAddress, Rd [,F]*

Status Bits

N *1* if the field-extended data moved to register is negative, *0* otherwise
C Unaffected
Z *1* if the field-extended data moved to register is 0, *0* otherwise
V *0*

MOVE Examples

Example 1

This is an example of the following MOVE instructions:

MOVE *Rs, *Rd*
MOVE *Rs, *Rd+*
MOVE *Rs, –*Rd*
MOVE *Rs, *Rd(offset)*
MOVE *Rs, @DAddress*

Assume that memory contains the following value before instruction execution:

Address	Data
15500h	0000h
15510h	0000h
15520h	0000h

Register A0 = FFFFFFFFh

Code	Before A1	FS0/1	After A1	@15500h	@15510h	@15520h
MOVE A0,*A1,0	00015500h	5/x	00015500h	001Fh	0000h	0000h
MOVE A0,*A1,1	00015503h	x/8	00015503h	078Fh	0000h	0000h
MOVE A0,*A1,0	00015508h	13/x	00015508h	FF00h	001Fh	0000h
MOVE A0,*A1,1	0001550Ch	x/24	0001550Ch	F000h	FFFFh	000Fh
MOVE A0,*A1+,1	0001551Dh	x/16	0001552Dh	0000h	E000h	1FFFh
MOVE A0,*A1+,0	00015516h	19/x	00015529h	0000h	FFC0h	01FFh
MOVE A0,*A1+,1	00015500h	x/32	00015520h	FFFFh	FFFFh	0000h
MOVE A0,–*A1,0	0001530h	5/x	000152Bh	0000h	0000h	F800h
MOVE A0,–*A1,1	000152Dh	x/8	0001525h	0000h	0000h	1FE0h
MOVE A0,–*A1,0	0001528h	13/x	000151Bh	0000h	F800h	00FFh
MOVE A0,*A1(0000h),1	00015500h	x/1	00015500h	0001h	0000h	0000h
MOVE A0,*A1(0FFFh),0	00014501h	19/x	00014501h	FFFFh	0007h	0000h
MOVE A0,*A1(7FFFh),1	0000D501h	x/22	0000D501h	FFFFh	003Fh	0000h
MOVE A0,*A1(8000h),0	0001D500h	27/x	0001D500h	FFFFh	07FFh	0000h
MOVE A0,@1550Bh,1	xxxxxxxx	x/16	xxxxxxxx	F800h	07FFh	0000h
MOVE A0,@15512h,0	xxxxxxxx	27/x	xxxxxxxx	0000h	FFFCh	1FFFh
MOVE A0,@1550Ch,1	xxxxxxxx	x/32	xxxxxxxx	F000h	FFFFh	0FFFh

Example 2

This is an example of the following MOVE instructions:

MOVE **Rs, Rd*
MOVE **Rs+, Rd*
MOVE *–*Rs, Rd*
MOVE **Rs(offset), Rd*
MOVE *@SAddress, Rd*

Assume that memory contains the following value before instruction execution:

Address	Data	Address	Data
15500h	7770h	15530h	3333h
15510h	7777h	15540h	4444h
15520h	0000h	15550h	5555h

Code	Before A0	FS0/1	FE0/1	After A0	A1	NCZV
MOVE *A0,A1,1	00015500h	x/1	x/1	00015500h	00000000h	0x10
MOVE *A0,A1,0	00015500h	5/x	0/x	00015500h	00000010h	0x00
MOVE *A0,A1,1	00015500h	x/5	x/1	FFFFFFF0h	00000000h	1x00
MOVE *A0,A1,0	00015500h	5/x	0/x	00015500h	00000010h	0x00
MOVE *A0,A1,0	00015500h	18/x	0/x	00037770h	00000010h	0x00
MOVE *A0+,A1,0	00015500h	12/x	0/x	0001550Ch	00000770h	0x00
MOVE *A0+,A1,1	00015500h	x/12	x/1	0001550Ch	00000770h	0x00
MOVE *A0+,A1,0	00015500h	27/x	0/x	0001551Bh	07777770h	0x00
MOVE *A0+,A1,1	00015500h	x/27	x/1	0001551Bh	FF777770h	1x00
MOVE –*A0,A1,0	00015520h	31/x	1/x	00015501h	3BBBBBB8h	0x00
MOVE –*A0,A1,0	00015520h	x/31	x/0	00015501h	3BBBBBB8h	0x00
MOVE –*A0,A1,0	00015520h	32/x	X/x	00015500h	77777770h	0x00
MOVE *A0(0020h),A1,1	0001551Ch	x/13	x/0	0001551Ch	00000443h	0x00
MOVE *A0(00FFh),A1,0	00015435h	16/x	1/x	00015435h	00004333h	0x00
MOVE *A0(7FFFh),A1,1	0000D531h	x/22	x/1	0000D531h	00000443h	0x00
MOVE *A0(8000h),A1,0	0001D530h	27/x	1/x	0001D530h	FC443333h	1x00
MOVE *A0(0FFECh),A1,0	0001554Dh	32/x	0/x	0001554Dh	AAA22219h	1x00
MOVE @15504h,A1,0	xxxxxxxx	1/x	18/x	xxxxxxxx	FFFF7777h	1x00
MOVE @15500h,A1,1	xxxxxxxx	x/0	x/18	xxxxxxxx	00037770h	0x00
MOVE @15501h,A1,0	xxxxxxxx	0/x	30/x	xxxxxxxx	3BBBBBB8h	0x00
MOVE @15501h,A1,1	xxxxxxxx	x/1	x/30	xxxxxxxx	FBBBBBB8h	1x00

Example 3 This is an example of the following MOVE instructions:

MOVE **Rs, *Rd*
MOVE **Rs+, *Rd+*
MOVE *–*Rs, –*Rd*
MOVE *@SAddress, @DAddress*
MOVE *@SAddress, *Rd+*

Assume that memory contains the following value before instruction execution:

Address	Data	Address	Data
15500h	FFFFh	15530h	0000h
15510h	FFFFh	15540h	0000h
15520h	FFFFh	15550h	0000h

Code	Before A0	A1	FS0/1	After A0	A1	@15530h	@15540h	@1550h
MOVE *A0,*A1,1	00015500h	00015530h	x/1	00015500h	00015530h	0001h	0000h	0000h
MOVE *A0,*A1,0	00015500h	00015534h	5/x	00015500h	00015534h	01F0h	0000h	0000h
MOVE *A0,*A1,1	00015500h	0001553Ah	x/10	00015500h	0001553Ah	FC00h	000Fh	0000h
MOVE *A0,*A1,0	00015500h	0001553Fh	19/x	00015500h	0001553Fh	8000h	FFFFh	0003h
MOVE *A0+,*AH,1	00015510h	00015532h	x/7	00015517h	00015539h	01FCh	0000h	0000h
MOVE *A0+,*AH,0	00015511h	0001553Ah	13/x	0001551Fh	00015547h	FC00h	007Fh	0000h
MOVE *A0+,*AH,1	00015513h	0001553Fh	x/8	0001551Bh	00015547h	8000h	007Fh	0000h
MOVE *A0+,*AH,0	00015510h	0001553Ah	28/x	0001552Ch	00015556h	FC00h	FFFFh	003Fh
MOVE *–A0,*–A1,0	00015527h	00015555h	31/x	00015508h	00015536h	FFC0h	FFFFh	001Fh
MOVE *–A0,*–A1,1	00015527h	00015550h	x/31	00015508h	00015531h	FFFEh	FFFFh	0000h
MOVE *–A0,*–A1,0	0001552Ah	00015550h	32/x	0001550Ah	00015530h	FFFFh	FFFFh	0000h
MOVE *–A0,*–A1,1	00015520h	0001555Ah	x/32	00015500h	0001553Ah	FC00h	FFFFh	03FFh
MOVE @15500h,*A1+,1	0001553Ah	xxxxxxxx	x/10	00015544h	xxxxxxxx	FC00h	000Fh	0000h
MOVE @15500h,*A1+,0	0001553Ah	xxxxxxxx	19/x	00015552h	xxxxxxxx	8000h	FFFFh	0003h
MOVE @1550Dh,*A1+,1	0001553Ah	xxxxxxxx	28/x	0001554Ch	xxxxxxxx	FFFFh	0FFFh	0000h
MOVE @15505h,*A1+,0	0001553Ah	xxxxxxxx	x/32	0001554Dh	xxxxxxxx	FFE0h	0FFFh	0000h
MOVE @15500h,@15530h,1	xxxxxxxx	xxxxxxxx	x/1	xxxxxxxx	xxxxxxxx	0001h	0000h	0000h
MOVE @15500h,@15534h,0	xxxxxxxx	xxxxxxxx	5/x	xxxxxxxx	xxxxxxxx	01F0h	0000h	0000h
MOVE @15500h,@15530h,1	xxxxxxxx	xxxxxxxx	x/7	xxxxxxxx	xxxxxxxx	007Fh	0000h	0000h
MOVE @15500h,@15530h,0	xxxxxxxx	xxxxxxxx	13/x	xxxxxxxx	xxxxxxxx	1FFFh	0000h	0000h

Example 4 This is an example of the following MOVE instructions:

MOVE **Rs(offset), *Rd+*
MOVE **Rs(offset), *Rd(offset)*

Assume that memory contains the following value before instruction execution:

Address	Data	Address	Data
15500h	0000h	15530h	3333h
15510h	0000h	15540h	4444h
15520h	0000h	15550h	5555h

Code	Before A0	A1	FS0/1	After A1	@15530h	@15540h	@1550h
MOVE *A0(0000h),*A1+,1	00015530h	0015500h	x/1	00015501h	0001h	0000h	0000h
MOVE *A0(00FFh),*A1+,1	00015535h	001550Ch	16/x	0001551Ch	3000h	0433h	0000h
MOVE *A0(0FFFh),*A1+,1	00015531h	00015510h	19/x	00015523h	0000h	3333h	0004h
MOVE *A0(0FFE0h),*A1+,1	00015558h	00015508h	x/31	00015527h	3300h	4444h	0055h
MOVE *A0(0001h),*A1(0000h),0	0001552Fh	00015504h	5/x	00015504h	0130h	0000h	0000h
MOVE *A0(000Fh),*A1(000Fh),0	0001552Dh	000154FDh	8/x	000154FDh	3000h	0004h	0000h
MOVE *A0(7FFFh),*A1(8000h),1	0000D531h	0001D508h	x/22	0001D508h	3300h	0433h	0000h
MOVE *A0(0FFF2h),*A1(7FFFh),0	00015540h	0000D501h	x/25	0000D501h	0CCCh	0111h	0000h

Syntax

MOVI *IW, Rd* [**, W**]

Execution

16-bit immediate value → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	1	0	R	Rd			
16-bit value															

Description

MOVI stores a 16-bit, sign-extended immediate value in the destination register. (*IW* in the instruction syntax represents the 16-bit value.)

The assembler uses the short form if the immediate value has been previously defined and is in the range –32,768 through 32,767. You can force the assembler to use the short form by following the register operand with **,W**:

```
MOVI IW,Rd,W
```

The assembler truncates the upper bits and issues an appropriate warning message.

Machine States

2

Status Bits

N *1* if the data being moved is negative, *0* otherwise
C Unaffected
Z Unaffected
V *1* if the data being moved is 0, *0* otherwise

Examples

Code		After	
		A0	**N C Z V**
MOVI	32767,A0	00007FFFh	0 x 0 0
MOVI	1,A0	00000001h	0 x 0 0
MOVI	0,A0	00000000h	0 x 1 0
MOVI	–1,A0	FFFFFFFFh	1 x 0 0
MOVI	–32768,A0	FFFF8000h	1 x 0 0
MOVI	0000h,A0	00000000h	0 x 1 0
MOVI	7FFFh,A0	00007FFFh	0 x 0 0

Syntax

MOVI *IL*, *Rd* [, **L**]

Execution

32-bit immediate value → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	1	1	1	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description

MOVI stores a 32-bit immediate value in the destination register. (IL in the instruction syntax represents the 32-bit value.)

The assembler uses this opcode if it cannot use the `MOVI IW,Rd` opcode or if the long opcode is forced by following the register operand with, **L:**

```
MOVI IL,Rd,L
```

Machine States

2 if immediate data is long-word aligned
3 if immediate data is not long-word aligned

Status Bits

N *1* if the data being moved is negative, *0* otherwise
C Unaffected
Z *1* if the data being moved is 0, *0* otherwise
V *0*

Examples

Code		After A0	N C Z V
MOVI	2147483647,A0	7FFFFFFFh	0 x 0 0
MOVI	32768,A0	00008000h	0 x 0 0
MOVI	–32769,A0	FFFF7FFFh	1 x 0 0
MOVI	–2147483648,A0	80000000h	1 x 0 0
MOVI	8000h,A0	00008000h	0 x 0 0
MOVI	0FFFFFFFFh,A0	FFFFFFFFh	1 x 0 0
MOVI	0FFFFh,A0,L	FFFFFFFFh	1 x 0 0

Syntax

MOVK *constant, Rd*

Execution

5-bit constant → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	constant					R	Rd			

Description

MOVK stores a 5-bit constant in the destination register. The constant is treated as an unsigned number in the range 1—32, where constant = 0 in the opcode corresponds to a value of 32. The resulting constant value is zero-extended to 32 bits.

Note that you cannot set a register to 0 with this instruction. You can clear a register by XORing the register with itself; use `CLR Rd` (an alternate mnemonic for XOR *Rs, Rd*) to accomplish this. Both these methods alter the Z bit (set it to 1).

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	After A0
`MOVK 1,A0`	00000001h
`MOVK 8,A0`	00000008h
`MOVK 16,A0`	00000010h
`MOVK 32,A0`	00000020h

Syntax

MOVX *Rs, Rd*

Execution

X half of Rs → X half of Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	Rs				R	Rd			

Description

MOVX moves the X half of the source register (16 LSBs) to the X half of the destination register. The Y halves of both registers are unaffected.

You can also use the MOVX and MOVY instructions for handling packed 16-bit quantities and XY addresses. You can use the RL instruction to swap the contents of X and Y.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before A0	A1	After A1
`MOVX A0,A1`	00000000h	FFFFFFFFh	FFFF0000h
`MOVX A0,A1`	12345678h	00000000h	00005678h
`MOVX A0,A1`	FFFFFFFFh	00000000h	0000FFFFh

Syntax

MOVY *Rs, Rd*

Execution

Y half of Rs → Y half of Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	Rs				R	Rd			

Description

MOVY moves the Y half of the source register (16 MSBs) to the Y half of the destination register. The X halves of both registers are unaffected.

You can also use the MOVX and MOVY instructions for handling packed 16-bit quantities and XY addresses. You can use the RL instruction to swap the contents of X and Y.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before		After
	A0	**A1**	**A1**
`MOVY A0,A1`	00000000h	FFFFFFFFh	0000FFFFh
`MOVY A0,A1`	12345678h	00000000h	12340000h
`MOVY A0,A1`	FFFFFFFFh	00000000h	FFFF0000h

Syntax

MPYS *Rs, Rd*

Execution

If Rd is an even-numbered register, Rs × Rd → Rd:Rd+1
If Rd is an odd-numbered register, Rs × Rd → Rd

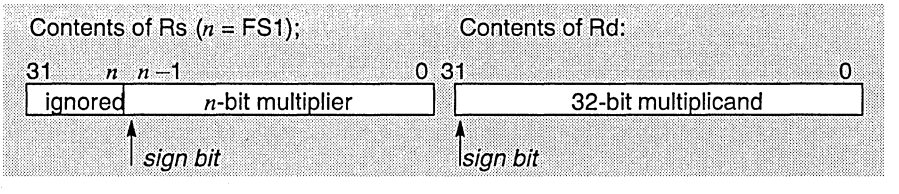
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	Rs				R	Rd			

Description

MPYS performs a signed multiply of a variably sized field in the source register by the 32 bits in the destination register. This produces a 32-bit to a 64-bit result, depending on the register and field definitions. Note that Rs and Rd must be in the same register file.

The value of field size 1 (FS1) defines the size of the multiplier in Rs. FS1 may have any **even** value n from 2 to 32 (that is, $n = 2, 4, 6 \ldots 30, 32$). The instruction executes a 32-bit-by-n-bit multiply — multiplying the 32 bits in Rd by the n bits in Rs. All values are signed. The MSB of the source field (bit $n-1$ in Rs) defines the sign of the field; the bits to the left of bit n are ignored. The MSB of Rd defines the sign of the multiplicand.

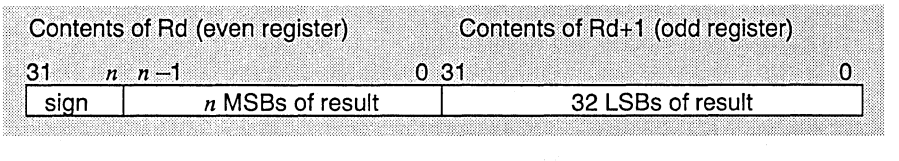


MPYS has two modes, depending on whether Rd is even or odd:

- **Rd Even:**

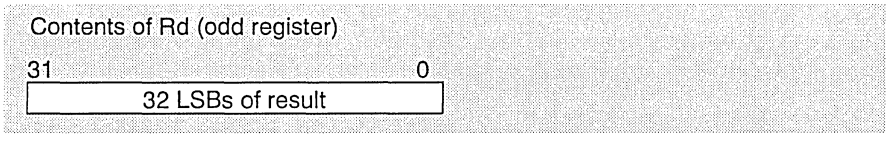
  MPYS multiplies the contents of Rd by the n-bit field in Rs, and stores the result in 2 consecutive registers, Rd and Rd+1. (For example, if Rd=B4, the result is stored in registers B4 and B5.) The result is sign-extended and right-justified; the 32 MSBs are stored in Rd, and the 32 LSBs are stored in Rd+1. Note that all 32 bits of both registers are used, regardless of the field size of the multiply.

  Do not use A14 or B14 as the destination register, because Rd+1 (A15 or B15) is the stack pointer register (SP). It is not desirable to write over the contents of the SP.



Rd Odd:

MPYS multiplies the contents of Rd by the n-bit field in Rs, and stores the 32 LSBs of the result in Rd; neither Rs nor Rd+1 are changed. If the result is greater than 32 bits, the extra MSBs are discarded, regardless of the field size. The N and Z status bits, however, are set according to the full result, including the MSBs that are discarded.



Machine States

5 + (field size)/2

Status Bits

N *1* if the result is negative, *0* otherwise
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Example 1

```
MPYS A1, A0
```

Before			After		
A0	**A1**	**FS1**	**A0**	**A1**	**NCZV**
00000000h	00000000h	32	00000000h	00000000h	0 x 1 x
7FFFFFFFh	7FFFFFFFh	32	3FFFFFFFh	00000001h	0 x 0 x
7FFFFFFFh	FFFFFFFFh	32	FFFFFFFFh	80000001h	1 x 0 x
FFFFFFFFh	7FFFFFFFh	32	FFFFFFFFh	80000001h	1 x 0 x
FFFFFFFFh	FFFFFFFFh	32	00000000h	00000001h	0 x 0 x
80000000h	7FFFFFFFh	32	C0000000h	80000000h	1 x 0 x
80000000h	80000000h	32	40000000h	00000000h	0 x 0 x
80000001h	80000000h	32	3FFFFFFFh	80000000h	0 x 0 x
8040156Fh	7FF3B074h	32	C0262CDCh	53E486F8h	1 x 0 x
8040156Fh	7FF3B074h	24	000624B1h	53E486F8h	0 x 0 x
8040156Fh	7FF3B074h	20	FFFE28B2h	594486F8h	1 x 0 x
8040156Fh	7FF3B074h	16	000027B2h	17EC86F8h	0 x 0 x
8040156Fh	7FF3B074h	14	000007C2h	1C0206F8h	0 x 0 x
8040156Fh	7FF3B074h	8	FFFFFFC6h	1D0766F8h	1 x 0 x
8040156Fh	7FF3B074h	6	00000005h	FCFF3BF8h	0 x 0 x
8040156Fh	7FF3B074h	4	FFFFFFFEh	01004158h	1 x 0 x
8040156Fh	7FF3B074h	2	00000000h	00000000h	0 x 1 x

Example 2

```
MPYS A0,A1
```

Before			After		
A0	**A1**	**FS1**	**A0**	**A1**	**NCZV**
00000000h	00000000h	32	unchanged	00000000h	0 x 1 x
7FFFFFFFh	7FFFFFFFh	32	unchanged	00000001h	0 x 0 x
7FFFFFFFh	7FFFFFFFh	32	unchanged	80000001h	1 x 0 x
FFFFFFFFh	7FFFFFFFh	32	unchanged	80000001h	1 x 0 x
FFFFFFFFh	FFFFFFFFh	32	unchanged	00000001h	0 x 0 x
80000000h	7FFFFFFFh	32	unchanged	80000000h	1 x 0 x
80000000h	80000000h	32	unchanged	00000000h	0 x 0 x
80000001h	80000000h	32	unchanged	80000000h	0 x 0 x
7FF3B074h	80401056h	32	unchanged	53E486F8h	1 x 0 x
7FF3B074h	80401056h	24	unchanged	53E486F8h	0 x 0 x
7FF3B074h	80401056h	20	unchanged	594486F8h	1 x 0 x
7FF3B074h	80401056h	16	unchanged	17EC86F8h	0 x 0 x
7FF3B074h	80401056h	14	unchanged	1C0206F8h	0 x 0 x
7FF3B074h	80401056h	8	unchanged	1D0766F8h	1 x 0 x
7FF3B074h	80401056h	6	unchanged	FCFF3BF8h	0 x 0 x
7FF3B074h	80401056h	4	unchanged	01004158h	1 x 0 x
7FF3B074h	80401056h	2	unchanged	00000000h	0 x 1 x

Syntax

MPYU *Rs, Rd*

Execution

If Rd is an even-numbered register: Rs × Rd → Rd:Rd+1
If Rd is an odd-numbered register: Rs × Rd → Rd

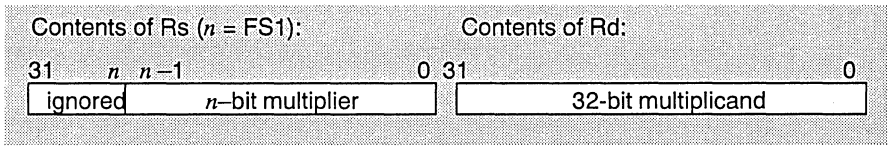
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	Rs				R	Rd			

Description

MPYU performs an unsigned multiply of a variably-sized field in the source register by the 32 bits in the destination register. This produces a 32-bit to a 64-bit result, depending on the register and field definitions. Note that Rs and Rd must be in the same register file.

The value of field size 1 (FS1) defines the size of the multiplier in Rs. FS1 may have any **even** value *n* from 2 to 32 (that is, *n* = 2, 4, 6 ... 30, 32). The instruction executes a 32-bit-by-*n*-bit multiply — multiplying the 32 bits in Rd by the *n* bits in Rs. All values are unsigned.

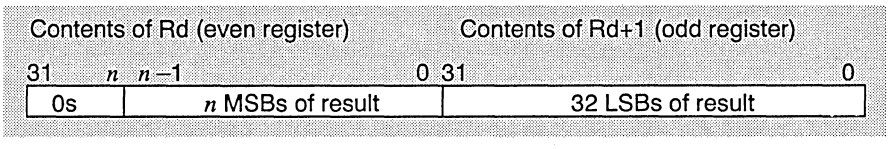


MPYS has two modes, depending on whether Rd is even or odd:

❑ **Rd Even:**

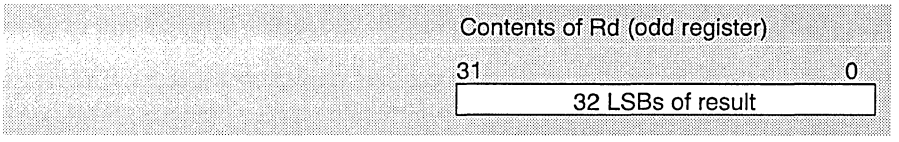
MPYU multiplies the contents of Rd by the *n*-bit field in Rs and stores the result in 2 consecutive registers, Rd and Rd+1. (For example, if Rd=B4, the result is stored in registers B4 and B5.) The result is zero-extended and right-justified; the 32 MSBs are stored in Rd, and the 32 LSBs are stored in Rd+1. Note that all 32 bits of both registers are used, regardless of the field size of the multiply.

Do not use A14 or B14 as the destination register, because Rd+1 (A15 or B15) is the stack pointer register (SP). It is not desirable to write over the contents of the SP.



❑ **Rd Odd:**

MPYU multiplies the contents of Rd by the *n*-bit field in Rs and stores the 32 LSBs of the result in Rd; Rs is not changed. If the result is greater than 32 bits, the extra MSBs are discarded, regardless of the field size. The Z status bit, however, is set according to the full result, including the MSBs that are discarded.



Machine States

Rs nonnegative: 5 + (field size)/2
Rs negative: 6 + (field size)/2

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Example 1

```
MPYU A1,A0
```

Before			After		
A0	**A1**	**FS1**	**A0**	**A1**	**NCZV**
FFFF0000h	10000000h	32	0FFFF000h	00000000h	x x 0 x
FFFF0000h	10001010h	32	1000000Fh	EFF00000h	x x 0 x
FFFF0000h	10001010h	16	0000100Fh	EFF00000h	x x 0 x
FFFF0000h	10001010h	8	0000000Fh	FFF00000h	x x 0 x
FFFF0000h	10001010h	4	00000000h	00000000h	x x 1 x
08001056h	0003B074h	32	00001D83h	DC4486F8h	x x 0 x
08001056h	0003B074h	16	00000583h	AB4286F8h	x x 0 x
08001056h	0003B074h	14	00000183h	A31786F8h	x x 0 x
08001056h	0003B074h	8	00000003h	A00766F8h	x x 0 x
08001056h	0003B074h	6	00000001h	A0035178h	x x 0 x
08001056h	0003B074h	4	00000000h	20004158h	x x 0 x
08001056h	0003B074h	2	00000000h	00000000h	x x 1 x

Example 2

```
MPYU A0,A1
```

Before			After		
A0	**A1**	**FS1**	**A0**	**A1**	**NCZV**
10000000h	FFFF0000h	32	unchanged	00000000h	x x 0 x
10001010h	FFFF0000h	32	unchanged	EFF00000h	x x 0 x
10001010h	FFFF0000h	16	unchanged	EFF00000h	x x 0 x
10001010h	FFFF0000h	8	unchanged	FFF00000h	x x 0 x
10001010h	FFFF0000h	4	unchanged	00000000h	x x 1 x
0003B074h	08001056h	32	unchanged	DC4486F8h	x x 0 x
0003B074h	08001056h	16	unchanged	AB4286F8h	x x 0 x
0003B074h	08001056h	14	unchanged	A31786F8h	x x 0 x
0003B074h	08001056h	8	unchanged	A00766F8h	x x 0 x
0003B074h	08001056h	6	unchanged	A0035178h	x x 0 x
0003B074h	08001056h	4	unchanged	20004158h	x x 0 x
0003B074h	08001056h	2	unchanged	00000000h	x x 1 x

Syntax

MWAIT

Execution

Wait for current memory cycle to complete

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Description

MWAIT delays further instruction execution to allow any pending write cycle to complete. If no write cycle is currently pending, the next instruction begins execution immediately. If a write cycle is pending, execution of the next instruction is delayed until the write cycle completes.

MWAIT is typically used to ensure that all pending I/O register updates have been completed prior to beginning a graphics instruction execution that depends on the values in the I/O registers. It may also be used to ensure that a pending write to a register in a memory-mapped peripheral external to the TMS34020 has completed prior to executing an instruction whose operation depends on the value in the register. Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Machine States

minimum of 2

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

```
        MOVK        4, B10
SETF    16,0,0
MOVE    B10, @C0000150h        ; load PSIZE
MWAIT                          ; wait for write to complete
DRAV    A0,A2
           ; In this case the 16 bit MOVE to PSIZE results
           ; in 1 hidden state at the time MWAIT is
           ; entered. MWAIT will take 2 cycles to execute.

MOVK    4, B10
SETF    6,0,0
MOVE    B10, @C0000150h        ; load PSIZE
MWAIT                          ; wait for write to complete
DRAV    A0,A2
           ; In this case the 6 bit MOVE to PSIZE results
           ; in 2 hidden states at the time MWAIT is
           ; entered. MWAIT will take 3 cycles to execute.
```

Syntax

NEG *Rd*

Execution

2s complement of Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	0	1	R	Rd			

Description

NEG stores the 2s complement of the contents of the destination register back into the destination register.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow (Rd ≠ 0), *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow (Rd = 80000000h), *0* otherwise

Examples

Code	Before	After				
	A0	N	C	Z	V	A0
NEG A0	00000000h	0	0	1	0	00000000h
NEG A0	55555555h	1	1	0	0	AAAAAAABh
NEG A0	7FFFFFFFh	1	1	0	0	80000001h
NEG A0	80000000h	1	1	0	1	80000000h
NEG A0	80000001h	0	1	0	0	7FFFFFFFh
NEG A0	FFFFFFFFh	0	1	0	0	00000001h

Syntax

NEGB *Rd*

Execution

(2s complement of Rd) – C → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	1	0	R	Rd			

Description

NEGB takes the 2s complement of the destination register's contents and decrements by 1 if the borrow bit (C) is set; the result is stored in the destination register. This instruction can be used in sequence with itself and with the NEG instruction for negating multiple-register quantities.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before		After	
	A0	C	N C Z V	A0
NEGB A0	00000000h	0	0 0 1 0	00000000h
NEGB A0	00000000h	1	1 1 0 0	FFFFFFFFh
NEGB A0	55555555h	0	1 1 0 0	AAAAAAABh
NEGB A0	55555555h	1	1 1 0 0	AAAAAAAAh
NEGB A0	7FFFFFFFh	0	1 1 0 0	80000001h
NEGB A0	7FFFFFFFh	1	1 1 0 0	80000000h
NEGB A0	80000000h	0	1 1 0 1	80000000h
NEGB A0	80000000h	1	0 1 0 0	7FFFFFFFh
NEGB A0	80000001h	0	0 1 0 0	7FFFFFFFh
NEGB A0	80000001h	1	0 1 0 0	7FFFFFFEh
NEGB A0	FFFFFFFFh	0	0 1 0 0	00000001h
NEGB A0	FFFFFFFFh	1	0 1 1 0	00000000h

Syntax **NOP**

Execution No operation

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0

Description The program counter is incremented to point to the next instruction. The processor status is otherwise unaffected.

You can use the NOP instruction to pad loops and perform other timing functions.

Machine States 1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

Code	Before	After
	PC	PC
NOP	00020000h	00020010h

Syntax

NOT *Rd*

Execution

NOT Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	1	1	1	R	Rd			

Description

NOT stores the 1s complement of the destination register's contents back into the destination register.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**N C Z V**	**A0**
`NOT A0`	00000000h	x x 0 x	FFFFFFFFh
`NOT A0`	55555555h	x x 0 x	AAAAAAAAh
`NOT A0`	FFFFFFFFh	x x 1 x	00000000h
`NOT A0`	80000000h	x x 0 x	7FFFFFFFh

Syntax

OR *Rs, Rd*

Execution

Rs OR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	Rs				R	Rd			

Description

This instruction bitwise-ORs the contents of the source register with the contents of the destination register; the result is stored in the destination register.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	A0	**A1**	**A1**	**N C Z V**
`OR A0,A1`	FFFFFFFFh	00000000h	FFFFFFFFh	x x 0 x
`OR A0,A1`	00000000h	FFFFFFFFh	FFFFFFFFh	x x 0 x
`OR A0,A1`	55555555h	AAAAAAAAh	FFFFFFFFh	x x 0 x
`OR A0,A1`	00000000h	00000000h	00000000h	x x 1 x

Syntax

ORI *IL*, *Rd*

Execution

32-bit immediate value OR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	0	1	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description

This instruction bitwise-ORs a 32-bit immediate value with the contents of the destination register and stores the result in the destination register. (*IL* in the syntax represents the 32-bit value.)

Machine States

2 if immediate data is long-word aligned
3 if immediate data is long-word aligned

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before A0	After A0	N C Z V
`ORI 0FFFFFFFFh,A0`	00000000h	FFFFFFFFh	x x 0 x
`ORI 00000000h,A0`	FFFFFFFFh	FFFFFFFFh	x x 0 x
`ORI 0AAAAAAAAh,A0`	55555555h	FFFFFFFFh	x x 0 x
`ORI 00000000h,A0`	00000000h	00000000h	x x 1 x

Syntax **PFILL XY**

Execution COLOR0 and COLOR1 pixels → pixel array (with processing)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1

Description

PFILL XY fills a pixel array, one row at a time, with a 2-color pattern. The pattern is defined by the PATTERN register. The 2 colors are defined by the COLOR0 and COLOR1 registers. PFILL replaces the 1s in pattern with the pixel value in COLOR1; it replaces the 0s in the pattern with the pixel value in COLOR0.

To fill an array with a 2-dimensional pattern, execute PFILL once for each row of the array. If the width of the specified fill region (defined by *DX* in DYDX) is more than 32 pixels, PFILL replicates the same 32-bit pattern as many times as necessary to fill the row. After each line is drawn, you will typically update the contents of the PATTERN register to define the next row of the pattern.

If you do not update the PATTERN register between rows, or if the number of rows in the fill region is ≠1, then the same 1-dimensional pattern is repeated for each row of the destination array. If the destination array pitch is a power of 2, and a pattern is drawn to the screen in this manner, then the filled area appears to contain stripes. If the destination array pitch is not a power of 2, then the pattern is defined for only the first line of the array.

Aligning a pattern

The contents of the PATTERN register control the pattern. As an example, consider the pixel addressed by the XY address in DADDR at the start of the PFILL instruction. Let that pixel be the nth pixel from the least significant end of a long-word boundary, where n is in the range:

$$0 \le n \le \left(\frac{32}{\text{PSIZE}} - 1\right)$$

Bit n in the PATTERN register determines if the first pixel drawn is a COLOR0 or COLOR1 pixel. Bit $n+1$ determines if the second pixel drawn is a COLOR0 or COLOR1 pixel, and so on. The PATTERN register works cyclically to draw a line. If the *DX* value in DYDX is large, then eventually bit 31 of the PATTERN register will be used to control an output pixel. If a further pixel is drawn, then it will be controlled by bit 0 of the pattern and so on.

Prealigning a pattern

The last example demonstrated that PFILL does not perform any internal alignment of the PATTERN register. This cuts the overhead time required to start executing and enables you to perform a pattern prealignment to suit your needs.

Consider this case where no prealignment is performed:

PSIZE		= 04h
DADDR	(B2)	= 00000 0000h
DPTCH	(B3)	= 00000 0000h
OFFSET	(B4)	= 00000 0000h
DYDX	(B7)	= 00020 0060h
COLOR0	(B8)	= 00000 0000h
COLOR1	(B9)	= 0FFFFFFFFh
PATTERN	(B13)	= 0FFF0 00FFh

For this example, PFILL draws a rectangle 96 pixels wide and 32 pixels high. The rectangle contains vertical stripes, alternating between COLOR1 and COLOR1. The first pixel drawn, at bit address 00000000h, is controlled by bit 0 of the PATTERN register.

If the screen is clear and the X part of DADDR is incremented by 1 to 00000001h, PFILL will redraw the rectangle. The first pixel drawn by PFILL, now at bit address 00000004h, will be controlled by bit 1 of the pattern register. The drawn pattern now appears as if it were fixed relative to the screen (not the rectangle edge).

This continues as the X component of DADDR is incremented until DADDR = 000000008h; at this point the first pixel drawn by PFILL, now at bit address 000000020h, will no longer be in the first 32-bit long-word of the screen. In this case, by the argument used above, the first drawn pixel is controlled, once again, by bit 0 of the PATTERN register and thus changing from DADDR = 000000007h to DADDR = 0000000008h the pattern will appear to *jump* within the rectangle. This may not always be a desirable way to manage the pattern. You may wish to do one of the following:

- Create a pattern that appears to be fixed relative to the screen background.
- Create a pattern that appears fixed relative to the edge of the rectangle.

Placing the pattern relative to the screen background

There are three ways to fixing or placing a pattern relative to the screen background.

1) Let the number of pixels in a long-word be p, where

$$p = \frac{32}{\text{PSIZE}}.$$

If the pattern in the PATTERN register repeats every p pixels, then it will appear fixed with respect to the screen background.

If you set B13 = FF00FF00FFh in the last example, the pattern will not jump.

2) Use a pixel size of 1 bit.

3) Manually rotate the contents of B13 before executing PFILL. The rotation amount depends on the following two things:
   - Pixel size
   - X component of DADDR

   Let the total number of bits controlled by the entire pattern (that is 32 × PSIZE bits), be known as a *super-word* (range 32 to 1024 bits in size).

   Let the long-word containing the pixel addressed by DADDR at the start of the PFILL XY be the n^{th} long-word in a super-word (range 0 to 32).

   Let the number of pixels in a 32-bit long-word be p, where

$$p = \frac{32}{\text{PSIZE}} \quad \text{(range 0 to 32).}$$

   Before starting PFILL XY, the pattern should be rotated right by m × p bits, before placing it in PATTERN.

   This may appear complex, but because pixel size is usually fixed, the prealignment operation can be reduced to a simple sequence of instructions. For example, at 4 bits per pixel, ANDing the XY address in DADDR (available before execution of PFILL) with 018h yields the value $m \times p$ which can be used to rotate the pattern before placing it into the PATTERN register. At other pixel sizes the following will yield $m \times p$:

PSIZE	$m \times p$	
1	000h AND DADDR	always 0 (no rotation required)
2	010h AND DADDR	
4	018h AND DADDR	
8	01Ch AND DADDR	
16	01Eh AND DADDR	
32	01Fh AND DADDR	

Placing the pattern relative to the rectangle

Placing or fixing the pattern relative to the rectangle means that the first pixel drawn by PFILL is always controlled by bit 0 of the PATTERN register. (This is how the LINE and FLINE instructions use PATTERN register.)

You can achieve a similar effect for PFILL by rotating the pattern left by a certain amount before placing it into the PATTERN register. The rotation amount depends on the following two things:

- Pixel size
- X component of DADDR

The rotation amount is derived by ANDing DADDR with a constant as follows:

PSIZE	Rotate Amount	
1	01Fh AND DADDR	
2	00Fh AND DADDR	
4	007h AND DADDR	
8	003h AND DADDR	
16	001h AND DADDR	
32	000h AND DADDR	always 0 (no rotation required)

Note:

This description describes a striped rectangle, but in practice PFILL is used to pattern-fill a single line followed by a change of pattern before pattern filling a second line, and so on. The reference to a rectangle is made for the purpose of illustration only.

Implied Operands

Register	Name	Format	Description
B2	DADDR	XY	Destination pixel block address
B3 †	DPTCH	Linear	Destination pixel block pitch
B7	DYDX	XY	Dimensions of drawn rectangle
B13	PATTERN	Binary	Pattern register
B14	POFFSET	Integer	Offset into the pattern

† If DY > 1, then DPTCH must be a power of 2, or the pattern will not be well defined.

Address	Name	Description and Elements (Bits)	
C00000B0h	CONTROL	PPOP	Pixel-processing operations (22 options)
		T	Transparency operation
		TM	Sets transparency mode
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)	
C0000160h	PMASK (32 bits)	Plane mask — pixel format	

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PFILL instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT (page 13-177). Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Pixel Processing

Pixel processing can be used with this instruction. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window Checking

Window checking **can** be used with this instruction.

Transparency You can enable transparency for this instruction by setting T[[CONTROL]] to 1. Select 1 of 3 transparency modes by setting TM[[CONTROL]]. For more information, refer to Section 12.9, Transparency, on page 12-36.

Plane Masking The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Corner Adjust Corner adjust **cannot** be used with this instruction.

Machine States Complex Instruction

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples This is an example of a C-compatible assembly routine which draws a rectangle on the screen; the screen is filled with a 16 × 16 binary pattern. This routine expects 5 arguments on the C parameter stack: width, height, xleft, ytop, and a pointer to the pattern.

This routine assumes the following registers were previously initialized by the caller:

B-file registers	DPTCH, OFFSET, WSTART, WEND, COLOR1, COLOR0
I/O registers	CONTROL, CONVDP, PSIZE and PMASK

```
STK       .set    A14              ;C-parameter stack pointer
DADDR     .set    B2               ;Destination address register
DYDX      .set    B7               ;Delta X/delta Y register
PATTERN   .set    B13              ;Pattern register
          .globl  _fill_patnrect   ;provide reference for external calls
_fill_patnrect:
          mmtm     SP,A0,A1,A2,A3  ;save required registers
          mmtm    SP,B0,B1,B2,B7,B10,B11,B13,B14
          move    STK,B14
          move    *-B14,DYDX,1     ;pop w
          move    *-B14,B10,1      ;pop h
          move    *-B14,DADDR,1    ;pop xleft
          move    *-B14,B11,1      ;pop ytop
          move    B14,STK
          move    *-STK,A3,1       ;pop pointer to pattern
          sll     16,B10
          movy    B10,DYDX         ;concatenate w, h
          sll     16,B11
          movy    B11,DADDR        ;concatenate xleft, ytop
          clip                     ;clip the rectangle to the window
          jrz     exit             ;jump if rectangle outside window
          move    DYDX,A1          ;set up y count
          move    A1,A2
          srl     16,A1
          movi    00010000H,A0
```

```
        movy    A0,A2
        move    A2,DYDX
        move    DADDR,A2
        getps   B0              ;calculate pattern adjustment
        rmo     B0,B0
        neg     B0
        movk    32,B1
        srl     B0,B1           ;number pixels per 32 bit word
        subk    1,B1            ;so complement will count pix's wrd
        move    DADDR,B0
        andn    B1,B0           ;address rounded to pix's/word bndry
        neg     B0              ;shift count = -(LSBs of x)
loop:
        move    A3,B10          ;pattern start address
        movk    15,B11          ;load 4-bit mask
        sll     16,B11          ;align mask with 4 LSBs of y
        and     DADDR,B11       ;isolate 4 LSBs of y
        srl     12,B11          ;convert y to index value
        add     B11,B10         ;index into pattern
        move    *B10,B10,0      ;get 16-bit row of pattern
        move    B10,B11
        sll     16,B11
        movy    B11,B10         ;replicate row to 32 bits
        rl      B0,B10          ;align pattern for x address
        move    B10,PATTERN     ;load aligned pattern
        pfill   XY
        addxy   A0,A2
        move    A2,DADDR
        dsj     A1,loop
exit:
        mmfm    SP,B0,B1,B2,B7,B10,B11,B13,B14
        mmfm    SP,A0,A1,A2,A3      ;restore required registers
        rets    2                   ;return
```

PIXBLT Instructions

The **PIXBLT** instruction moves a 2-dimensional array of pixels from one memory location to another. Section 12.5, Pixel-Array Instructions, on page 12-8 provides additional information about the PIXBLT instructions. The following list describes characteristics common to all **PIXBLT** instructions. Note that **PIXBLT L,M,L** is discussed independently on page 13-204.

- The source and destination addresses of the arrays are designated by the SADDR and DADDR registers, respectively.
- **B**, **L**, and **XY** *are not actually operands.* Instead, they identify the source or destination array starting addresses as binary, linear, or XY addresses. B, L, and XY are referred to as qualifiers.
- Qualifiers are entered exactly as shown in the syntax; for example, `PIXBLT B,L`. The first qualifier indicates the format of the starting address of the source array; the second qualifier indicates the format of the starting address of the destination array.
- You can select a **pixel-processing** option by setting PPOP[[CONTROL]]. When the PIXBLT has binary source data, the pixel-processing operation is applied to *expanded pixels* as they are processed with the destination array; that is, the data is *first expanded* and *then processed.* There are 16 Boolean and 6 arithmetic operations; the default case at reset is the S → D operation. Not that the 6 arithmetic operations do not operate with pixel sizes of 1 bit per pixel. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.
- You can enable **transparency** by setting T[[CONTROL]] to 1. The TMS34020 supports 3 transparency modes; TM[[CONTROL]] selects 1 of 3 transparency options. For more information, refer to Section 12.9, Transparency, on page 12-36.
- The **plane mask** is enabled. For more information, refer to Section 12.10, Plane Masking, on page 12-39.
- This instruction can be **interrupted** at a word or row boundary of the destination array. For more information, refer to Section6.6, Interrupting Graphics Instructions, on page 6-13.
- If CST[[DPYCTL]] is set, each memory read or write initiated by the PIXBLT generates a **shift register transfer** read or write cycle at the selected address. This operation can be used for bulk memory clears or transfers. (Not all VRAMs support this capability.) For more information, refer to subsection 9.13.4, VRAM Bulk Initialization, on page 9-47.
- The **status bits** are undefined unless otherwise noted in the individual descriptions.
- The machine states are not presented because the PIXBLT instructions are complex instructions.

Table 13–6. Summary of Array Types for the PIXBLT Instruction

		Destination Array	
		Linear	XY
Source Array	Binary	√	√
	Linear	√	√
	XY	√	√

Table 13–7. Summary of B-File Registers for PIXBLT Instructions

		Format						
Reg.	**Name**	**B, L**	**B, XY**	**L, L**	**L, XY**	**XY, L**	**XY, XY**	**Description**
B0	SADDR	Linear	Linear	Linear	Linear	XY	XY	Source pixel array starting address
B1	SPTCH	Linear	Linear	Linear	Linear	Linear	Linear	Source pixel array pitch
B2	DADDR	Linear	XY	Linear	XY	Linear	XY	Destination pixel array starting address
B3	DPTCH	Linear	Linear	Linear	Linear	Linear	Linear	Destination pixel array pitch
B4	OFFSET		Linear		Linear	Linear	Linear	Screen origin (0,0)
B5	WSTART		XY		XY		XY	Window starting corner
B6	WEND		XY		XY		XY	Window ending corner
B7	DYDX	XY	XY	XY	XY	XY	XY	Pixel array dimensions (rows:columns)
B8	COLOR0	Pixel	Pixel					Background expansion color
B9	COLOR1	Pixel	Pixel					Foreground expansion color
B14		res	res	res	res	res	res	Reserved register

Note: PIXBLT L,M,L is discussed independently on page 13-204.

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT. Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Table 13–8. Summary of I/O Registers for the PIXBLT Instructions

Address	Name	Format						Description and Elements
		B, L	B, XY	L, L	L, XY	XY, L	XY, XY	
C00000B0h	CONTROL	√	√	√	√	√	√	PPOP–Pixel-processing operations (22 options)
			√		√		√	W – Window clipping or pick operation
		√	√	√	√	√	√	T – Enables transparency
		√	√	√	√	√	√	TM – selects 1 of 3 transparency options
				√	√	√	√	PBH – PIXBLT horizontal direction
				√	√	√	√	PBV – PIXBLT vertical direction
C0000130h	CONVSP		√		√	√	√	XY-to-linear conversion (source pitch) Used for source preclipping.
C0000140h	CONVDP		√		√	√	√	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	√	√	√	√	√	√	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	/	√	√	√	√	√	Plane mask — pixel format

Note: PIXBLT L,M,L is discussed independently on page 13-204.

The **PIXBLT** instruction has 6 combinations, which are listed below with their corresponding instruction words and descriptions. Note that **PIXBLT L,M,L** is discussed independently on page 13-204.

PIXBLT B, L
binary to linear

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0

Description

This instruction expands, transfers, and processes a binary source pixel array; it operates on 2-dimensional arrays of pixels using linear starting addresses for both the source and the destination. The source pixel array is treated as a 1-bit-per-pixel array. As the PIXBLT proceeds, the source pixels are expanded and then combined with the corresponding destination pixels based on the selected graphics operations.

Source Array

The source pixel array for the expand operation is defined by the contents of the SADDR, SPTCH, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Source Expansion The actual values of the source pixels are determined by the interaction of the source array with contents of the COLOR1 and COLOR0 registers. In the expansion operation, a **1** bit in the source array selects a pixel from the COLOR1 register for operation on the destination array. A **0** bit in the source array selects a COLOR0 pixel for this purpose. The pixels selected from the COLOR1 and COLOR0 registers are those that align directly with their intended position in the destination array word.

Destination Array The location of the destination pixel block is defined by the contents of the DADDR, DPTCH, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Corner Adjust No corner adjust is performed for this instruction.The pixel transfer simply proceeds in the order of increasing linear addresses.

Window Checking Window operations are not enabled for this instruction. The contents of the WSTART and WEND registers are ignored.

PIXBLT B, XY
binary to XY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	0	1	0	0	0	0	0

Description This PIXBLT instruction expands, transfers, and processes a binary source pixel array with a destination pixel array; it operates on 2-dimensional arrays of pixels using a linear starting address for the source and an XY address for the destination. The source pixel array is treated as a 1-bit-per-pixel array. As the PIXBLT proceeds, the source pixels are expanded and then combined with the corresponding destination pixels based on the selected graphics operations.

Source Array The source pixel array for the expand operation is defined by the contents of the SADDR, SPTCH, DYDX, and (possibly) CONVSP registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Source Expansion The actual values of the source pixels are determined by the interaction of the source array with contents of the COLOR1 and COLOR0 registers. In the expansion operation, a **1** bit in the source array selects a pixel from the COLOR1 register for operation on the destination array. A **0** bit in the source array selects a COLOR0 pixel for this purpose. The pixels selected from the COLOR1 and COLOR0 registers are those that align directly with their intended position in the destination array word.

Destination Array The location of the destination pixel block is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Corner Adjust No corner adjust is performed for this instruction. The transfer executes in the order of increasing linear addresses.

Window Checking You can use window checking with this instruction by setting the W bits in the CONTROL register to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Status Bits

N Undefined
C Undefined
Z Undefined
V 1 if a window violation occurs, *0* otherwise; undefined if window checking is not enabled (W=00)

PIXBLT L, L
linear to linear

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0

Description The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array; it operates on 2-dimensional arrays of pixels using linear starting addresses for both the source and the destination. As the PIXBLT proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Source Array The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Corner Adjust PBH[[CONTROL]] and PBV[[CONTROL]] govern the direction of the PIXBLT. To set up the corner adjust, refer to subsection 12.5.1.2, Selecting the Starting Corner for a PIXBLT, on page 12-10.

Window Checking Window operations are not enabled for this instruction. The contents of the WSTART and WEND registers are ignored.

PIXBLT L, XY
linear to XY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	0	1	0	0	0	0	0

Description The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array; it operates on 2-dimensional arrays of pixels using a linear starting address for the source array and an XY address for the destination array. As the PIXBLT proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Source Array The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, DYDX, and (possibly) CONVSP registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Corner Adjust PBH[[CONTROL]] and PBV[[CONTROL]] govern the direction of the PIXBLT. To set up the corner adjust, refer to subsection 12.5.1.2, Selecting the Starting Corner for a PIXBLT, on page 12-10.

Window Checking You can use window checking with this instruction by setting W[[CONTROL]] to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Status Bits

N Undefined
C Undefined
Z Undefined
V 1 if window violation occurs, 0 otherwise; undefined if window checking is not enabled ($W=00_2$)

PIXBLT XY, L
XY to linear

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	0	0	0	0	0

Description The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array; it operates on 2-dimensional arrays of pixels using an XY starting address for the source pixel array and a linear address for the destination array. As the PIXBLT proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Source Array The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, CONVSP, OFFSET, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Destination Array The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, DYDX, and (potentially) CONVDP registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Corner Adjust PBH[[CONTROL]] and PBV[[CONTROL]] govern the direction of the PIXBLT. To set up the corner adjust, refer to subsection 12.5.1.2, Selecting the Starting Corner for a PIXBLT, on page 12-10.

Window Checking Window operations are not enabled for this instruction. The contents of the WSTART and WEND registers are ignored.

PIXBLT XY, XY

XY to XY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	1	0	0	0	0	0

Description

The PIXBLT instruction transfers and processes a source pixel array with a destination pixel array; it operates on 2-dimensional arrays of pixels using XY starting addresses for both the source and destination pixel arrays. As the PIXBLT proceeds, the source pixels are combined with the corresponding destination pixels based on the selected graphics operations.

Source Array

The source pixel array for the processing operation is defined by the contents of the SADDR, SPTCH, CONVSP, OFFSET, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Destination Array

The location of the destination pixel array is defined by the contents of the DADDR, DPTCH, CONVDP, OFFSET, and DYDX registers. For more details, refer to Section 12.5, Pixel-Array Instructions, on page 12-8.

Corner Adjust

PBH[CONTROL] and PBV[CONTROL] govern the direction of the PIXBLT. To set up the corner adjust, refer to subsection 12.5.1.2, Selecting the Starting Corner for a PIXBLT, on page 12-10.

Window Checking

You can use window checking with this instruction by setting W[CONTROL] to the desired value. If you select window checking mode 1, 2, or 3, the WSTART and WEND registers define the XY starting and ending corners of a rectangular window. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V 1 if a window violation occurs, 0 otherwise; unaffected if window clipping not enabled

Transparency example for PIXBLT B, L

Before executing the PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup:

Register File B:

SADDR = 00002030h
SPTCH = 00000100h
DADDR = 00033000h
DPTCH = 00001000h
DYDX = 00020010h
COLOR0 = FEDCFEDCh
COLOR1 = BA98BA98h

I/O Registers:

PSIZE = 0010h

Additional implied operand values are listed with each example. For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data							
02000h	xxxxh,	xxxxh,	xxxxh,	1234h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	5678h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
33000h	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh
33080h	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh
34000h	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh
34080h	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh

Example 1

This example uses the *replace* (S → D) pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
33000h	FEDCh,	FEDCh,	BA98h,	FEDCh,	BA98h,	BA98h,	FEDCh,	FEDCh
33080h	FEDCh,	BA98h,	FEDCh,	FEDCh,	BA98h,	FEDCh,	FEDCh,	FEDCh
34000h	FEDCh,	FEDCh,	FEDCh,	BA98h,	BA98h,	BA98h,	BA98h,	FEDCh
34080h	FEDCh,	BA98h,	BA98h,	FEDCh,	BA98h,	FEDCh,	BA98h,	FEDCh

Example 2

This example uses transparency with COLOR0 = 00000000h. Before instruction execution, PMASK = 0000h and CONTROL = 0020h (T=1, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
33000h	FFFFh,	FFFFh,	BA98h,	FFFFh,	BA98h,	BA98h,	FFFFh,	FFFFh
33080h	FFFFh,	BA98h,	FFFFh,	FFFFh,	BA98h,	FFFFh,	FFFFh,	FFFFh
34000h	FFFFh,	FFFFh,	FFFFh,	BA98h,	BA98h,	BA98h,	BA98h,	FFFFh
34080h	FFFFh,	BA98h,	BA98h,	FFFFh,	BA98h,	FFFFh,	BA98h,	FFFFh

Window-clipping example for PIXBLT B, XY

Before executing the PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup:

Register File B:

SADDR	= 00002010h
SPTCH	= 00000010h
DADDR	= 00300022h
DPTCH	= 00001000h
OFFSET	= 00010000h
WSTART	= 00000026h
WEND	= 00400050h
DYDX	= 00040010h
COLOR0	= 00000000h
COLOR1	= 7C7C7C7Ch

I/O Registers:

PSIZE	= 0008h
CONVSP	= 001Bh
CONVDP	= 0013h

Additional implied operand values are listed with each example. For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data						
2000h	xxxxh,	0123h	4567h,	89ABh, CDEFh,	xxxxh,	xxxxh,	xxxh
40000h to 43200h	FFFFh						

Example 1

This example uses the *replace* (S → D) pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data						
41100h	FFFFh,	7C7Ch,	007Ch,	7C00h,	007Ch,	007Ch,	007Ch, 0000h
41180h	007Ch,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh, FFFFh
42100h	FFFFh,	7C7Ch,	7C00h,	7C00h,	7C00h,	007Ch,	7C00h, 0000h
42180h	7C00h,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh, FFFFh
43100h	FFFFh,	7C7Ch,	7C7Ch,	7C00h,	7C7Ch,	007Ch,	7C7Ch, 0000h
43180h	7C7Ch,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh,	FFFFh, FFFFh

XY Addressing

		X Address																		
Y		**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**2A**	**2B**	**2C**	**2D**	**2E**	**2F**	**30**	**31**	**32**	**33**	**34**
Address	**30**	FF	FF	7C	7C	00	00	00	7C	00	00	7C	00	00	00	00	00	00	00	FF	FF	FF
	31	FF	FF	7C	7C	7C	00	00	7C	7C	00	7C	00	7C	00	00	00	7C	00	FF	FF	FF
	32	FF	FF	7C	7C	00	7C	00	7C	00	7C	7C	00	00	7C	00	00	00	7C	FF	FF	FF
	33	FF	FF	7C	7C	7C	7C	00	7C	7C	7C	7C	00	7C	7C	00	00	7C	7C	FF	FF	FF

Example 2

This example uses window operation 3 (clipped destination). Before instruction execution, PMASK = 0000h and CONTROL = 00C0h (T=0, W=11, PP=00000).

After instruction execution, memory contains the following values:

XY Addressing

Y Address	X Address: 20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F	30	31	32	33	34
30	FF	FF	FF	FF	FF	FF	00	7C	00	00	7C	00	00	00	00	00	00	00	FF	FF	FF
31	FF	FF	FF	FF	FF	FF	00	7C	7C	00	7C	00	7C	00	00	00	7C	00	FF	FF	FF
32	FF	FF	FF	FF	FF	FF	00	7C	00	7C	7C	00	00	7C	00	00	00	7C	FF	FF	FF
33	FF	FF	FF	FF	FF	FF	00	7C	7C	7C	7C	00	7C	7C	00	00	7C	7C	FF	FF	FF

Pixel-processing example for PIXBLT L, L

Before executing the PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup:

Register File B:

SADDR	= 00002004h
SPTCH	= 00000080h
DADDR	= 00002228h
DPTCH	= 00000080h
OFFSET	= 00000000h
DYDX	= 0000200Dh

I/O Registers:

PSIZE = 0004h

Additional implied operand values are listed with each example. For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data							
02000h	000xh,	1111h	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh	xxxxh
02280h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example 1

This example uses the replace (S → D) pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	FFxxh,	EEEFh,	DDDEh,	xCCDh,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	00xxh,	1110h,	2221h,	x332h,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example 2

This example uses the (D – S) → D pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4800h T=0, W=00, PP=10010).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	OFFxxh,	111Fh,	2221h,	x332h,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	OFFxxh,	111Fh,	2221h,	x332h,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Plane mask example for L, XY

Before executing the PIXBLT instruction, load the implied operand registers with appropriate values. This PIXBLT examples uses the following implied operand setup:

Register File B:

SADDR	=	00002004h
SPTCH	=	00000080h
DADDR	=	00520007h
DPTCH	=	00000100h
OFFSET	=	00001000h
WSTART	=	0030000Ch
WEND	=	00530014h
DYDX	=	00030016h

I/O Registers:

PSIZE	=	0004h
PMASK	=	0000h
CONVDP	=	0017h
CONTROL	=	0000h

(W=00, T=0, PP=00000)

For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data							
02000h	3210h,	7654h,	BA98h,	FEDCh,	3210h,	7654h,	BA98h,	FEDCh
02080h	3210h,	7654h,	BA98h,	FEDCh,	3210h,	7654h,	BA98h,	FEDCh
02100h	3210h,	7654h,	BA98h,	FEDCh,	3210h,	7654h,	BA98h,	FEDCh
15200h to 15480h	8888h							

Example

This example uses the replace (S → D) pixel-processing operation. Before instruction execution, PMASK = 7777H and CONTROL = 0000h (T=0, W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
15200h,	8888h,	1888h,	5432h,	9876h,	DCBAh,	10FEh,	5432h,	8886h
15300h	8888h,	1888h,	5432h,	9876h,	DCBAh,	10FEh,	5432h,	8886h
15400h	8888h,	1888h,	5432h,	9876h,	DCBAh,	10FEh,	5432h,	8886h

XY Addressing

		X Address																													
Y		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	B	C	D	E	F
Address	52	8	8	8	8	8	8	8	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	8	8	8	8
	53	8	8	8	8	8	8	8	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	8	8	8	8
	54	8	8	8	8	8	8	8	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	8	8	8	8

Example for PIXBLT XY, L

Before executing the PIXBLT instruction, load the implied operand registers with appropriate values. This PIXBLT examples use the following implied operand setup:

Register File B:

SADDR	=	00400001h
SPTCH	=	00000080h
DADDR	=	00002228h
DPTCH	=	00000080h
OFFSET	=	00010000h
WSTART	=	00300009h
WEND	=	00420012h
DYDX	=	00030016h

I/O Registers:

PSIZE	=	0004h
CONVSP	=	0016h
CONVDP	=	0016h
PMASK	=	0000h
CONTROL	=	0000h (W=00, T=0, PP=00000)

For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data							
02000h	000xh,	1111h	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02080h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh	xxxxh
02280h	xxxxh,	xxxxh,	FFxxh,	FFFFh,	FFFFh,	xFFFh,	xxxxh	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example This example uses transparency. Before instruction execution, PMASK = 0000h and CONTROL = 0200h (T=1,W=00, PP=00000).

After instruction execution, memory contains the following values:

Linear Address	Data							
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02000h	000xh,	1111h,	2222h,	xx33h,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02100h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02180h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh
02200h	xxxxh	xxxxh,	FFxxh,	111Fh	2221h,	x332h,	xxxxh,	xxxxh
02280h	xxxxh,	xxxxh,	FFxxh,	111Fh,	2221h,	x332h,	xxxxh,	xxxxh
02300h	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh,	xxxxh

Example for PIXBLT XY, XY

Before executing the PIXBLT instruction, load the implied operand registers with appropriate values. These PIXBLT examples use the following implied operand setup:

Register File B:

SADDR	=	00200004h
SPTCH	=	00000200h
DADDR	=	00410004h
DPTCH	=	00000200h
OFFSET	=	00010000h
WSTART	=	00300009h
WEND	=	00420012h
DYDX	=	00030016h

I/O Registers:

PSIZE	=	0004h
CONVSP	=	0016h
CONVDP	=	0016h
PMASK	=	0000h
CONTROL	=	0000h

(W=00, T=00, PP=00000)

For this example, assume that memory contains the following data before instruction execution.

Linear Address	Data
04000h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
04200h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
04400h	3210h, 7654h, BA98h, FEDCh, 3210h, 7654h, BA98h, FEDCh
18200h to 18680h	3333h

Example

This example uses the (D ADDS S) → D pixel-processing operation. Before instruction execution, PMASK = 0000h and CONTROL = 4400h (T=0, W=00, PP=10001).

After instruction execution, memory contains the following values:

XY Addressing

		X Address																													
Y		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**B**	**C**	**C**	**D**	**E**	**F**
A																																	
d	**41**	3	3	3	3	7	8	9	A	B	C	D	E	F	F	F	F	3	4	5	6	7	8	9	A	B	C	3	3	3	3	3	3
d																																	
r	**41**	3	3	3	3	7	8	9	A	B	C	D	E	F	F	F	F	3	4	5	6	7	8	9	A	B	C	3	3	3	3	3	3
e																																	
s	**41**	3	3	3	3	7	8	9	A	B	C	D	E	F	F	F	F	3	4	5	6	7	8	9	A	B	C	3	3	3	3	3	3
s																																	

Syntax

PIXBLT L, M, L

Execution

Linear pixel array to linear pixel array using a binary mask array

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	0	0	1	0	1	1	1

Description

This instruction transfers a pixel array from the source location specified by a linear address in SADDR to the destination location specified by a linear address in DADDR, which is under the control of the binary mask pixel array specified by a linear address in MADDR. The array dimensions are in DYDX.

Each source pixel is combined with the destination pixel according to the selected pixel-processing option. The resulting pixel can then be written to the destination pixel only if the corresponding bit in the mask array is a *1*.

Implied Operands

Register		Name	Format	Description
B0	†	SADDR	Linear	Source pixel array address
B1		SPTCH	Linear	Source pixel array pitch
B2	†	DADDR	Linear	Destination pixel array address
B3		DPTCH	Linear	Destination pixel array pitch
B7		DYDX	b:a	Dimensions of drawn rectangle
B10	†	MADDR	Linear	Mask pixel array address
B11		MPTCH	Linear	Mask array pitch
B12 & B14	†	Reserved Temporary Registers		

† These registers are changed by instruction execution

Address	Name	Description and Elements (Bits)
C00000B0h	CONTROL	PPOP Pixel-processing operations (22 options) T Transparency operation TM Sets transparency mode PBH PIXBLT horizontal direction PBV PIXBLT vertical direction
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format

Corner Adjust

PBH[CONTROL] and PBV[CONTROL] govern the direction of the PIXBLT. To set up the corner adjust, refer to subsection 12.5.1.2, Selecting the Starting Corner for a PIXBLT, on page 12-10.

Window Checking

Window operations are not enabled for this instruction. The contents of the WSTART and WEND registers are ignored.

Pixel Processing

Select a pixel processing option for this instruction by setting PPOP[[CONTROL]]. The pixel processing option is applied to pixels as they are processed with the destination array. Note that the data is read through the plane mask and then processed. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace* (S → D) operation. The 6 arithmetic operations do not operate with pixel sizes of 1 or 2 bits per pixel. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Transparency

You can enable **transparency** by setting T[[CONTROL]] to 1. The TMS34020 supports 3 transparency modes; TM[[CONTROL]] selects 1 of 3 transparency options. For more information, refer to Section 12.9, Transparency, on page 12-36.

Plane Masking

The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Machine States

complex instruction

Status Bits

N Undefined
C Undefined
Z Undefined
V Undefined

PIXT Instructions

The **PIXT** instruction transfers a pixel from one location to another. The following list describes characteristics common to all **PIXT** instructions.

- Rs and Rd must be in the same register file.
- The **plane mask** is enabled for all PIXT instructions. For more information, refer to Section 12.10, Plane Masking, on page 12-39.
- The **status bits** are undefined unless otherwise noted in the individual descriptions.
- For **machine states** information, refer to Section 15.1 on page 15-2.

Section 12.3, Single-Pixel Instructions, on page 12-6 provides additional information about the PIXBLT instructions.

Table 13–9. Summary of Operand Formats for the PIXT Instructions

		Destination Pixel		
		Rd	**Rd*	**Rd*.XY
Source Pixel	*Rs*		√	√
	**Rs*	√	√	
	**Rs*.XY	√		√

Table 13–10.Summary of B-File Registers for PIXT Instructions

		Format			
Reg.	**Name**	*Rs, *Rd*.**XY**	**Rs*.**XY**, *Rd*	**Rs*.**XY**, **Rd*.**XY**	**Description**
B1	SPTCH		Linear	Linear	Source pixel array pitch
B3	DPTCH	Linear	Linear	Linear	Destination pixel array pitch
B4	OFFSET	Linear	Linear	Linear	Screen origin (0,0)
B5	WSTART	XY		XY	Window starting corner
B6	WEND	XY		XY	Window ending corner

Due to the pipelining of memory writes, the *last* I/O register that you write to may not, in some cases, contain the desired value when you execute the PIXBLT instruction. To ensure that this register contains the correct value for execution, you may want to follow the write to that location with an MWAIT (page 13-177). Refer to Section 4.5.6 on page 4-13 for a description of the potential latency of writes to I/O registers.

Table 13–11. Summary of I/O Registers for the PIXT Instructions

Address	Name	*Rs, *Rd	*Rs, *Rd.XY	*Rs, Rd	*Rs, *Rd	*Rs.XY, Rd	*Rs.XY, *Rd.XY	Description and Elements
C00000B0h	CONTROL	√	√		√		√	PPOP — Pixel processing operations (22 options)
			√				√	W — Window clipping or pick operation
		√	√		√		√	T — Enables transparency
		√	√		√		√	TM — Selects transparency options
C0000130h	CONVSP					√	√	XY-to-linear conversion (source pitch) Used for source preclipping.
C0000140h	CONVDP		√				√	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	√	√	√	√	√	√	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	/	√	√	√	√	√	Plane mask — pixel format

The **PIXT** instruction has 6 addressing modes, which are listed below with their corresponding instruction words and descriptions.

PIXT *Rs, *Rd*
register to memory

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	Rs				R	Rd			

The source pixel is the 1, 2, 4, 8, 16, or 32 LSBs of the source register, depending on the pixel size specified in the PSIZE register. The destination register contains a linear address; the source pixel is transferred to this memory location.

You can select a **pixel processing** option to use with this instruction. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window checking cannot be used with this instruction.

You can enable **transparency** by setting T[CONTROL] to 1. The TMS34020 supports 3 transparency modes; TM[CONTROL] selects 1 of 3 transparency options. At reset, the default case for transparency is *off*. For more information, refer to Section 12.9, Transparency, on page 12-36.

PIXT *Rs, *Rd.XY
register to memory

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	Rs				R	Rd			

The source pixel is the 1, 2, 4, 8, 16, or 32 LSBs of the source register, depending on the pixel size specified in the PSIZE register. The destination register contains an XY address; the X value occupies the 16 LSBs of the register, and the Y value occupies the 16 MSBs. The source pixel is moved to the XY address specified in Rd.

You can use **window checking** with this instruction by setting W[[CONTROL]] to the desired value. For more information, refer to Section 12.7, Window Checking, on page 12-19.

You can select a **pixel processing** option to use with this instruction. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

You can enable **transparency** by setting T[[CONTROL]] to 1. The TMS34020 supports 3 transparency modes; TM[[CONTROL]] selects 1 of 3 transparency options. For more information, refer to Section 12.9, Transparency, on page 12-36.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if pixel is outside the window and W = 1, 2, 3; *0* otherwise. Unaffected if W = 0.

PIXT *Rs, Rd
memory to register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	Rs				R	Rd			

The source register contains a linear address; the pixel at this address is transferred into the destination register. When the pixel is moved into Rd, it is right-justified and zero-extended to 32 bits, according to the pixel size specified in the PSIZE register.

Window checking cannot be used with this instruction. The W bits are ignored.

Pixel processing cannot be used with this instruction.

Transparency cannot be used with this instruction.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

PIXT *Rs, *Rd
memory to memory

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	Rs				R	Rd			

The source and destination registers both contain linear addresses. The address in Rs is the address of the source pixel; the pixel is moved into the address in Rd.

You can select a **pixel processing** option to use with this instruction. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window checking cannot be used with this instruction.

You can enable **transparency** by setting T[CONTROL] to 1. The TMS34020 supports 3 transparency modes; TM[CONTROL] selects 1 of 3 transparency options. For more information, refer to Section 12.9, Transparency, on page 12-36.

PIXT *Rs.XY, Rd
memory to register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	Rs				R	Rd			

The source register contains an XY address; the X value occupies the 16 LSBs of the register, and the Y value occupies the 16 MSBs. The address in Rs is the address of the source pixel; this pixel is moved into the destination register. When the pixel is moved into Rd, it is right-justified and zero-extended to 32 bits according to the pixel size specified in the PSIZE register.

Pixel processing cannot be used with this instruction.

Transparency cannot be used with this instruction.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

PIXT *Rs.XY, *Rd.XY
memory to memory

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	Rs				R	Rd			

The source and destination registers both contain XY addresses; the X value occupies the 16 LSBs of the register, and the Y value occupies the 16 MSBs. Rs contains the address of the source pixel; Rd contains the address where the pixel is moved.

You can use **window checking** with this instruction by setting W[CONTROL] to the desired value. For more information, refer to Section 12.7, Window Checking, on page 12-19.

You can select a **pixel processing** option to use with this instruction. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

You can enable **transparency** by setting T[CONTROL] to 1. The TMS34020 supports 3 transparency modes; TM[CONTROL] selects 1 of 3 transparency options. At reset, the default case for transparency is *off.* For more information, refer to Section 12.9, Transparency, on page 12-36.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V *1* if the pixel lies outside the window and W=1, W=2, or W=3; *0* otherwise. Unaffected if W=0.

Section 12.3, Single-Pixel Instructions, on page 12-6 provides additional information about the PIXT instructions.

PIXT examples

Example 1

```
PIXT A0, *A1
```

	Before					After		
	A0	**A1**	**@20500H**	**PSIZE**	**PP**	**T**	**PMASK**	**@20500h**
1)	0000FFFFh	00020500h	0000h	0001h	00000	0	0000h	0001h
1)	0000FFFFh	00020500h	0000h	0002h	00000	0	0000h	0003h
1)	0000FFFFh	00020500h	0000h	0004h	00000	0	0000h	000Fh
1)	0000FFFFh	00020500h	0000h	0008h	00000	0	0000h	00FFh
1)	0000FFFFh	00020500h	0000h	0010h	00000	0	0000h	FFFFh
1)	00000006h	00020508h	0000h	0004h	00000	0	0000h	0600h
2)	00000006h	00020508h	0300h	0004h	01010	0	0000h	0500h
3)	00000006h	00020508h	0100h	0004h	00001	0	0000h	0000h
4)	00000006h	00020508h	0100h	0004h	00001	0	0000h	0100h
5)	00000006h	00020508h	0000h	0004h	00000	0	AAAAh	0400h

Notes:
1) S replaces D
2) (S XOR D) = 0, replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency is on, D is not replaced
5) S replaces unmasked bit of D

Example 2

Before executing a PIXT instruction, load the implied operand registers with appropriate values. These PIXT examples use thefollowing implied operand setup:

Register File B:
DPTCH = 00000800h
OFFSET = 00000000h
WSTART = 00300020h
WEND = 00500142h

I/O Registers:
CONVDP = 0014h

```
PIXT A0, *A1.XY
```

Before	After							
A0	A1	@20500H	PSIZE	PP	W	T	PMASK	@20500h
1) 0000FFFFh	00400500h	0000h	0001h	00000	00	0	0000h	0001h
1) 0000FFFFh	00400280h	0000h	0002h	00000	00	0	0000h	0003h
1) 0000FFFFh	00400140h	0000h	0004h	00000	00	0	0000h	000Fh
1) 0000FFFFh	004000A0h	0000h	0008h	00000	00	0	0000h	00FFh
1) 0000FFFFh	00400050h	0000h	0010h	00000	00	0	0000h	FFFFh
1) 00000006h	00400142h	0000h	0004h	00000	00	0	0000h	0600h
2) 00000006h	00400142h	0300h	0004h	01010	00	0	0000h	0500h
3) 00000006h	00400142h	0100h	0004h	00001	00	0	0000h	0000h
4) 00000006h	00400142h	0100h	0004h	00001	00	0	0000h	0100h
5) 00000006h	00400142h	0000h	0004h	00000	00	0	AAAAh	0400h
6) 00000006h	00400142h	0000h	0004h	00000	00	0	0000h	0600h
7) 00000006h	00400143h	0000h	0004h	00000	00	0	0000h	0000h
8) 00000006h	00400143h	0000h	0004h	00000	00	0	0000h	0000h

XY Address in A1 = Linear Address 20500h

Notes:

1) S replaces D
2) (S XOR D) = 0, replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency is on, D is not replaced
5) S replaces unmasked bit of D
6) Window Option = 3, D inside window, S replaces D
7) Window Option = 3, D outside window, D not replaced, V bit set in status register
8) Window Option = 2, D outside window, D not replaced, WV interrupt

Example 3 Assume that memory contains the following values:

Address	Data
@20500h	0FFFFh
@20510h	3333h

```
PIXT *A0,A1
```

Before			After
A0	PSIZE	PMASK	A1
00020500h	0001h	0000h	00000001h
00020500h	0001h	FFFFh	00000000h
00020500h	0002h	0000h	00000003h
00020500h	0002h	5555h	00000002h
00020500h	0004h	0000h	0000000Fh
00020510h	0004h	9999h	00000002h
00020500h	0008h	0000h	000000FFh
00020510h	0008h	5454h	00000023h
00020500h	0010h	0000h	0000FFFFh
00020500h	0010h	BA98h	00004567h
00020510h	0010h	BA98h	00000123h

```
PIXT *A0,*A1
```

	Before						After		
	AO	**A1**	**@20500H**	**PSIZE**	**PP**	**T**	**PMASK**	**@20500h**	**20510h**
1)	00020500h	00020508h	000Fh	0001h	00000	0	0000h	010Fh	xxxx
1)	00020500h	00020508h	000Fh	002h	00000	0	0000h	030Fh	xxxx
1)	00020500h	00020508h	000Fh	0004h	00000	0	0000h	0F0Fh	xxxx
1)	00020500h	00020508h	00EFh	0008h	00000	0	0000h	EFEFh	xxxx
1)	00020500h	00020508h	1234h	0010h	00000	0	0000h	3434h	xx12h
2)	00020500h	00020508h	030Fh	0004h	01010	0	0000h	0C0Fh	xxxx
3)	00020500h	00020508h	010Eh	0004h	00001	0	0000h	000Eh	xxxx
4)	00020500h	00020508h	020Eh	0004h	00001	0	0000h	020Eh	xxxx
5)	000020500h	00020508h	000Fh	0004h	00000	0	AAAAh	050Fh	xxxx

Notes:

1) S replaces D
2) (S XOR D) replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency in on, D not replaced
5) S replaces unmasked bits of D

Example 4

These PIXT examples use the following implied operand setup.

Register File B:
DPTCH =800h
OFFSET =00000000h

I/O Registers:
CONVSP = 0014h

Assume that memory address @20500h contains CF3Fh before instruction execution.

```
PIXT *A0.XY,A1
```

Before			After
A0	**PSIZE**	**PMASK**	**A1**
00400500h	0001h	0000h	00000001h
00400500h	0001h	FFFFh	00000000h
00400280h	0002h	0000h	00000003h
0400280h	0002h	AAAAh	00000001h
00400140h	0004h	0000h	0000000Fh
00400140h	0004h	9999h	00000006h
004000A0h	0008h	0000h	0000003Fh
004000A0h	0008h	8989h	00000036h
00400050h	0010h	0000h	0000CFCFh
00400050h	0010h	7310h	00008C2F

Note:

The XY addresses stored in register A1 in these examples translate to the linear memory address 20500h. The pitch of the line source was not changed for any of these examples

Example 5 These PIXT examples use the following implied operand setup.

Register File B:

SPTCH	=	800h
DPTCH	=	800h
OFFSET	=	00000000h
WSTART	=	00300020h
WEND	=	00500142h

I/O Registers:

CONVSP	=	0014h
CONVDP	=	0014h

```
PIXT *A0.XY,*A1.XY
```

	Before A0	After A1	@20500H	PSIZE	PP	W	T	PMASK	@20500h	@20510h
1)	00400500h	00400508h	000Fh	0001h	00000	00	0	0000h	010Fh	xxxx
1)	00400280h	00400284h	000Fh	0002h	00000	00	0	0000h	030Fh	xxxx
1)	00400140h	0400142h	000Fh	0004h	00000	00	0	0000h	0F0Fh	xxxx
1)	004000A0h	004000A1h	000Fh	0008h	00000	00	0	0000h	EFEFh	xxxx
1)	0040005Fh	00400051h	00EFh	0010h	00000	00	0	0000h	CDEFh	CDEFh
2)	00400050h	00400142h	0306h	0004h	01010	00	0	0000h	0506h	xxxx
3)	00400140h	00400142h	0106h	0004h	00001	00	0	0000h	0006h	xxxx
4)	00400140h	00400142h	0106h	0004h	10001	00	0	0000h	0106h	xxxx
5)	00400140h	00400142h	0006h	0004h	00001	00	0	0000h	0406h	xxxx
6)	00400140h	00400142h	0006h	0004h	00000	11	0	AAAAh	0606h	xxxx
7)	00400140h	00400142h	0006h	0004h	00000	11	0	0000h	0006h	xxxx
8)	00400140h	00400143h	0006h	0004h	00000	10	0	0000h	0006h	xxxxy

XY Address in A1 = Linear Address 20500h

Notes:

1) S replaces D
2) (S XOR D) replaces D
3) (S AND D) = 0, transparency is off, D is replaced
4) (S + D) = 0, transparency in on, D not replaced
5) S replaces unmasked bits of D
6) Window Option = 3, D inside window, S replaces D
7) Window Option = 3, D outside window, D not replaced, V bit set in status register
8) Window Option = 2, D outside window, D not replaced, WV interrupt generated, V bit set in status register

Syntax **POPST**

Execution *SP+ → ST

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

Description POPST pops the status register from the stack and increments the SP by 32 after the status register is removed from the stack.

31	30	29	28	26	25	22	21	11	10—6	5	4—0
N	C	Z	V	BF	IX	SS	IE	FE1	FS1	FE0	FS0

Note: Shaded portions are reserved.

For more information, refer to Section 4.1, The Status Register, on page 4-2.

Machine States 6 if the SP is aligned
7 if the SP is not aligned

Status Bits All bits are restored.

Examples Assume that memory contains the following values before instruction execution:

Address	**Data**
0FF00000h	0010h
0FF00010h	C000h

Examples

Code	**Before**	**After**	
	SP	**ST**	**SP**
`POPST`	0FF00000h	C0000010h	0FF00020h

Syntax **PUSHST**

Execution ST → –*SP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0

Description PUSHST writes the status register contents to the address contained in the SP–32.

31	30	29	28	26	25	22	21	11	10—6	5	4—0
N	C	Z	V	BF	IX	SS	IE	FE1	FS1	FE0	FS0

Note: Shaded portions are reserved.

For more information, refer to Section 4.1, The Status Register, on page 4-2.

Machine States
2 (1) if the SP is aligned
2 (2) if the SP is not aligned

Status Bits
N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

Code	Before		After
	SP	ST	SP
PUSHST	0FF00020h	C0000010h	0FF00000h

Memory contains the following values after instruction execution:

Address	Data
0FF00010h	0010h
0FF00020h	C000h

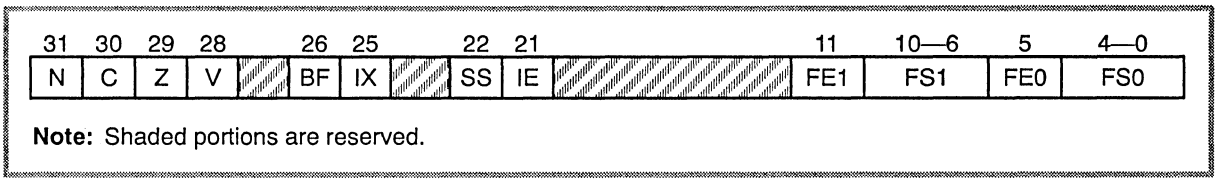
Syntax **PUTST** *Rs*

Execution Rs → ST

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	1	0	1	R	Rd			

Description PUTST copies the contents of the specified register into the status register.



Note: Shaded portions are reserved.

For more information, refer to Section 4.1, The Status Register, on page 4-2.

Machine States 3

Status Bits

N Set from bit 31 of Rs
C Set from bit 30 of Rs
Z Set from bit 29 of Rs
V Set from bit 28 of Rs

Example

Code	Before	After
	A0	**ST**	**ST**
`PUTST A0`	C0000010h	xxxxxxxxh	C0000010h

Syntax

RETI

Execution

*SP+ → ST
*SP+ → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0

Description

RETI returns to an interrupted routine from an interrupt service routine. The instruction restores the ST and PC to their original values that were stored on the system stack.

The stack is located in external memory and the top is indicated by the stack pointer (SP). The stack grows in the direction of decreasing linear address. The ST and PC are popped from the stack and the SP is incremented by 32 after each register is removed from the stack.

Note:

RETI checks the IX (instruction execution) and BF (bus fault) bits in the restored ST register. If IX or BF is set, the RETI expects to find the internal register values that define the state of the TMS34020 on the stack along with the ST and PC.

If this is the case, the RETI restores the additional register values that were pushed on the stack and clears the IX and BF bits in the restored ST value.

The CONTROL register and any B-file registers modified by an interrupt routine should be restored before RETI is executed. Otherwise, interrupted instructions may not resume execution correctly.

Machine States

52 if BF status bit = 1
38 if IX status bit = 1
else 7

Status Bits

N Copy of corresponding bit in stack location
C Copy of corresponding bit in stack location
Z Copy of corresponding bit in stack location
V Copy of corresponding bit in stack location
IE Copy of corresponding bit in stack location

Interrupts

If the IE bit in the restored ST is a 1, interrupts are enabled by the time the RETI instruction finishes executing. If an interrupt request is active during the last state of the RETI instruction, and the interrupt is enabled in the INTENB register, the interrupt will be taken immediately following the RETI. If the source of the interrupt is not cleared automatically, the interrupt service routine should take steps to clear the source of the interrupt. If this is not done, the interrupt will be serviced repeatedly. Sections 6.7, External Interrupts, on page 6-15,

6.8, Internal Interrupts, on page 6-16, and 6.9, The Bus Fault Interrupt, on page 6-19 discuss each interrupt and the details for clearing the source of the interrupt.

Examples

Assume that memory contains the following values before instruction execution:

Address	Data
CCC0000h	0010h
CCC0010h	C000h
CCC0020h	FFF0h
CCC0030h	0044h

Code	Before	After		
	SP	ST	PC	SP
`RETI`	CCC0000h	C0000010h	0044FFF0h	CCC0040h

Syntax

RETM

Execution

*SP+ → ST
*SP+ → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0

Description

RETM is used at the end of a single step trap routine. RETM acts similar to RETI, but RETM forces the next instruction from the interrupted program to be read directly from memory, that is, it is not read from the cache. The fetched instruction is executed and the single step trap is then taken again; this sequence repeats.

Note:

RETM uses the cache read mechanism to access the next instruction in the interrupted code. When the single-step bit (bit 22 in ST) is set the cache fills are blocked, so if the next instruction in the interrupted code is not already in cache when RETM is executed, then the single step trap will be taken repeatedly without executing any of the main program opcodes. This makes RETM unsuitable for terminating single-step traps.

Machine States

52 if BF status bit = 1
38 if IX status bit = 1
else 10

Status Bits

N Copy of corresponding bit in stack location
C Copy of corresponding bit in stack location
Z Copy of corresponding bit in stack location
V Copy of corresponding bit in stack location
IE Copy of corresponding bit in stack location

Examples

Assume that memory contains the following values before instruction execution:

Address	**Data**
CCC0000h	0010h
CCC0010h	C000h
CCC0020h	FFF0h
CCC0030h	0044h

Code	**Before**	**After**		
	SP	**ST**	**PC**	**SP**
`RETM`	CCC0000h	C0000010h	0044FFF0h	CCC0040h

Syntax

RETS [*N*]

Execution

*SP → PC (*N* defaults to 0)
SP + 32 + 16*N* → SP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	1	1	N				

Description

RETS returns from a subroutine by popping the program counter from the stack and incrementing the stack pointer.

The parameter *N* is a value in the range of 0 to 31; it specifies the number of words by which the stack pointer SP is incremented after the return address is popped from the system stack. N is optional; if the value of N is not specified explicitly, the assembler sets it to the default value of *0*.

Following completion of the RETS instruction, execution continues at the address pointed to by the PC popped from the stack.

Machine States

5
6 if the stack isn't aligned

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Assume that memory contains the following values before instruction execution:

Address	**Data**
0FF00000h	0FFF0h
0FF00010h	0001h

Code	**Before**	**After**	
	SP	**PC**	**SP**
`RETS`	0FF00000h	0001FFF0h	0FF00020h
`RETS 1`	0FF00000h	0001FFF0h	0FF00030h
`RETS 2`	0FF00000h	0001FFF0h	0FF00040h
`RETS 16`	0FF00000h	0001FFF0h	0FF00120h
`RETS 31`	0FF00000h	0001FFF0h	0FF00210h

Syntax

REV *Rd*

Execution

revision number → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	R	Rd			

Description

REV stores the number which uniquely identifies the revision of silicon in the destination register. The format of the REV number is:

bits 0—2 silicon revision number

bit 3 = 1 if TMS34010 (if bit 3 = 0, then TMS34020; bits 3 and 4 cannot both be 1)

bit 4 = 1 if TMS34020 (if bit 4 = 0, then TMS34010; bits 3 and 4 cannot both be 1))

bits 5—15 reserved for future generation parts

bits 16—23 spin-offs

bits 24—31 reserved

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

```
REV A0
```

Before A0 = FFFFFFFF

After A0 = 00000010 (TMS34020 revision 1.0)

After A0 = 00000011 (TMS34020 revision 2.0)

Syntax

RL *constant, Rd*

Execution

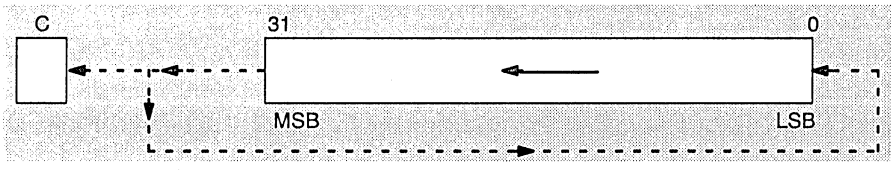
left-rotate Rd by constant → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	0	constant					R	Rd			

Description

RL rotates the contents of the destination register left by the specifed number of bits. RL performs a circular left shift that moves each bit shifted out the MSB of the register into the register's LSB. The rotate count is specified as a value in the range 0 to 31 and is stored in the 5-bit constant field of the RL instruction word.



The assembler only accepts absolute expressions for the rotate count. If the specified rotation value is greater than 31, the assembler issues a warning and sets the constant to its 5 LSBs.

The carry bit is set to the value of the last bit that is shifted out of the MSB (this value is the same as the final value of the LSB). You can use a rotate count of 0 to clear the carry and test a register for 0 simultaneously.

Machine States

1

Status Bits

N Unaffected
C Set to value of bit [32 – constant], *0* for rotate count of constant = 0
Z *1* if result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A1	N C Z V	A1
`RL 0,A1`	0000000Fh	x 0 0 x	0000000Fh
`RL 1,A1`	F0000000h	x 1 0 x	E0000001h
`RL 4,A1`	F0000000h	x 1 0 x	0000000Fh
`RL 5,A1`	F0000000h	x 0 0 x	0000001Eh
`RL 30,A1`	F0000000h	x 1 0 x	3C000000h
`RL 5,A1`	00000000h	x 0 1 x	00000000h

Syntax

RL *Rs, Rd*

Execution

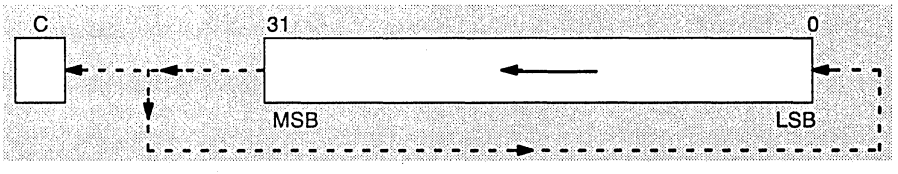
left-rotate Rd by 5 LSBs of Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	0	0	Rs				R	Rd			

Description

RL rotates the contents of the destination register left by the number of bits specified in the source register. RL performs a circular left shift that moves each bit shifted out of the MSB of the register into the register's LSB. The rotate count is specified as a value in the range 0 to 31 and is taken from the 5 LSBs of the source register; the 27 MSBs of the source register are ignored.



The carry bit is set to the value of the last bit that is shifted out of the MSB (this value is the same as the final value of the LSB). You can use a rotate count to 0 to clear the carry and text Rd for 0 simultaneously.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Set to value of bit [32 – Rs], *0* for rotate count of 0
Z *1* if result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After		
	5 LSBs A0	A1	N C Z V	A1
RL A0,A1	0 0 0 0 0	0000000Fh	x 0 0 x	0000000Fh
RL A0,A1	0 0 1 0 0	F0000000h	x 1 0 x	0000000Fh
RL A0,A1	0 0 1 0 1	F0000000h	x 0 0 x	0000001Eh
RL A0,A1	1 1 1 1 1	F0000000h	x 0 0 x	78000000h
RL A0,A1	x x x x x	00000000h	x 0 1 x	00000000h

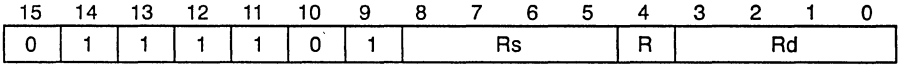
Syntax

RMO *Rs, Rd*

Execution

bit number of rightmost 1 in Rs → Rd

Instruction Words



Description

The RMO instruction locates the rightmost (least significant) *1* in the source register. It then loads the bit number of the rightmost *1* bit into the destination register. Bit *31* of Rs is the MSB (leftmost) and bit *0* is the LSB (rightmost). If there are no *1* bits in the source register, then the destination result is 0 and status bit Z is set .

The rightmost *1* in the source register can be right-justified by following the RMO instruction with RL *Rs*, *Rd* instruction, where Rs is the destination register of the RMO instruction and Rd is the source register.

The source and destination registers must be in the same register.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the source register contents are 0, *0* otherwise.
V Unaffected

Examples

Code	Before A0	After NCZV	A1
`RMO A0,A1`	00000000h	xx1x	00000000h
`RMO A0,A1`	00000001h	xx0x	00000000h
`RMO A0,A1`	00000010h	xx0x	00000004h
`RMO A0,A1`	08000000h	xx0x	0000001Bh
`RMO A0,A1`	80000000h	xx0x	0000001Fh

Syntax

RPIX *Rd*

Execution

$Rd_{new} = Rd_{old}$

LS pixel replicated $\left(\frac{32}{PSIZE}\right)$ 5 times

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	R	Rd			

Description

RPIX replicates the pixel value in the specified destination register. Prior to executing the instruction, you should right-justifiy the value in Rd. The pixel size is specified by PSIZE and must be 1, 2, 4, 8, 16, or 32 bits. Immediately following completion of the instruction, the pixel value will have been replicated throughout the 32 bits of the register.

Given a pixel size of *n* bits, the replication operation replaces the original pixel value with 32/*n* copies of the pixel. The replication process overwrites the 32–*n* bits to the left of the original pixel. For more information, refer to Section 12.6, Auxiliary Graphics Instructions, on page 12-17.

Implied Operands

Address	Name	Description and Elements (Bits)
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)

Machine States

2 if PSIZE = 32
4 if if PSIZE = 16
5 if PSIZE = 8
6 if PSIZE = 4
7 if PSIZE = 2
8 if PSIZE = 1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

```
RPXL A0
        PSIZE = 8    Before   A0 = XXXXXX34
                     After    A0 = 3434343434      Cycles = 5

RPXL B8
        PSIZE = 4    Before   B8 = XXXXXXXA
                     After    B8 = AAAAAAAA        Cycles = 6
```

Syntax **SETC**

Execution 1 → C

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	1	1	1	0	0	0	0	0

Description SETC sets the carry bit (C) in the status register to 1. The rest of the status register is unaffected.

31	30	29	28	26	25	22	21	11	10—6	5	4—0
N	C	Z	V	BF	IX	SS	IE	FE1	FS1	FE0	FS0

Note: Shaded portions are reserved.

This instruction is useful for returning a true/false value (in the carry bit) from a subroutine without using a general-purpose register.

Machine States 1

Status Bits

N Unaffected
C *1*
Z Unaffected
V Unaffected

Examples

Code	Before ST	N C Z V	After ST	N C Z V
SETC	00000000h	0 0 0 0	40000000h	0 1 0 0
SETC	B0000010h	1 0 1 1	F0000010h	1 1 1 1
SETC	4000001Fh	0 1 0 0	4000001Fh	0 1 0 0

Syntax

SETCDP

Execution

Destination pitch conversion factor → CONVDP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	0	0	1	1

Description

SETCDP loads the CONVDP register with the appropriate value used in XY to linear conversion based on the DPTCH register.

Remember to execute MWAIT after SETCDP to ensure that the CONVDP register has been set before using its value in a CVXYL or similar instruction. For more information, refer to Section 12.11, Setting up the Implied Operands for Graphics Instructions, on page 12-43.

Implied Operands

Address	Name	Description and Elements (Bits)
B3	DPTCH (linear)	Destination array pitch
C0000140h	CONVDP	Destination pitch conversion register

Machine States

pitch is	a power of 2:	4(1)
	2 powers of 2:	6(1)
	arbitrary	3(1)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Before: B3 = 00001000h (512 × 8)
After: C0000140 = 0013h

Before: B3 = 00000400h (128 × 8)
After: C0000140 = 0015h

Before: B3 = 00001400h (640 × 8)
After: C0000140 = 1513h

Before: B3 = 00000019h (25 × 1)
After: C0000140 = 0000h

Syntax

SETCMP

Execution

Mask pitch conversion factor → CONVMP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	1	1	1	1	0	1	1

Description

SETCMP loads the CONVMP register with the appropriate value used in XY to linear conversion based on the MPTCH register.

Remember to execute MWAIT after SETCMP to ensure that the CONVMP register has been set before using its value in a CVMXYL or similar instruction. For more information, refer to Section 12.11, Setting up the Implied Operands for Graphics Instructions, on page 12-43.

Implied Operands

Address	Name	Description and Elements (Bits)
B11	MPTCH (linear)	Mask array pitch
C0000180h	CONVMP	Mask pitch conversion register

Machine States

pitch is a power of 2: 4(1)
2 powers of 2: 6(1)
arbitrary 3(1)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Before: B3 = 00001000 (512 × 8)
After: C0000180 = 0013

Before: B3 = 00000400 (128 × 8)
After: C0000180 = 0015

Before: B3 = 00001400 (640 × 8)
After: C0000180 = 1513

Before: B3 = 00000019 (25 × 1)
After: C0000180 = 0000

Syntax

SETCSP

Execution

Source pitch conversion factor → CONVSP

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	1

Description

SETCSP loads the CONVSP register with the appropriate value used in XY to linear conversion based on the SPTCH register.

Remember to execute MWAIT after SETCSP to ensure that the CONVSP register has been set before using its value in a CVSXYL or similar instruction. For more information, refer to Section 12.11, Setting up the Implied Operands for Graphics Instructions, on page 12-43.

Implied Operands

Address	Name	Description and Elements (Bits)
B1	SPTCH (linear)	Source array pitch
C0000130h	CONVSP	Source pitch conversion register

Machine States

pitch is a power of 2: 4(1)
2 powers of 2: 6(1)
arbitrary 3(1)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Before: B3 = 00001000 (512 × 8)
After: C0000130 = 0013

Before: B3 = 00000400 (128 × 8)
After: C0000130 = 0015

Before: B3 = 00001400 (640 × 8)
After: C0000130 = 1513

Before: B3 = 00000019 (25 × 1)
After: C0000130 = 0000

Syntax

SETF *FS, FE* [, *F*]

Execution

FS, FE → ST

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	0	1	FE	FS				

Description

SETF loads specified field size (FS) and field extension (FE) values into the status register; depending on the value of the *F* parameter, this information sets the field size and extension for either field 0 or field 1. (The remainder of the status register is not affected.)

31	30	29	28	26	25	22	21	11	10—6	5	4—0
N	C	Z	V	BF	IX	SS	IE	FE1	FS1	FE0	FS0

Note: Shaded portions are reserved.

For more information, refer to Section 4.1, The Status Register, on page 4-2.

- The *FS* parameter is a value between 1 and 32; it selects the field size. (Note that an FS value of 0 in the opcode corresponds to an actual selected field size of 32.)
- The *FE* parameter is a value of 0 or 1:

  FE=0 selects zero-extension for a field.
  FE=1 selects sign-extension for a field.
- The *F* parameter is optional; the default value for *F* is 0. The *F* value determines whether the SETF instruction sets the field size and extension for field 0 or for field 1.

  F=0 selects FS0, FE0 to be altered.
  F=1 selects FS1, FE1 to be altered.

Each MOVE instruction also has an *F* parameter that selects the field size and extension of either field 0 or field 1 for the individual move. You can use the SETF instruction to prepare for MOVE instructions.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Examples

Code	Before ST	After ST
`SETF 32,0,0`	xxxxx000h	xxxxx000h
`SETF 32,1,0`	xxxxx000h	xxxxx020h
`SETF 31,1,0`	xxxxx000h	xxxxx03Fh
`SETF 16,0,0`	xxxxx000h	xxxxx010h
`SETF 32,0,1`	xxxxx000h	xxxxx000h
`SETF 32,1,1`	xxxxx000h	xxxxx800h
`SETF 31,1,1`	xxxxx000h	xxxxxFC0h
`SETF 16,0,1`	xxxxx000h	xxxxx400h

Syntax

SEXT *Rd* [, *F*]

Execution

field in Rd → sign-extended field Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	0	0	0	R	Rd			

Description

SEXT sign-extends the right-justified field contained in the destination register by copying the MSB of the field data into all the nonfield bits of the destination register. The size of the field is determined by the current field size. The optional *F* parameter, which must be specified as a 0 or a 1, selects the field size:

F=0 selects FS0 for the field size.
F=1 selects FS1 for the field size.

The default value for *F* is 0.

Machine States

2

Status Bits

N *1* if the result is negative, *0* otherwise
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	FS0/1	A0	N C Z V	A0
`SEXT A0,0`	17/x	00008000h	0 x 0 x	00008000h
`SEXT A0,0`	16/x	00008000h	1 x 0 x	FFFF8000h
`SEXT A0,0`	15/x	00008000h	0 x 1 x	00000000h
`SEXT A0,1`	x/17	00008000h	0 x 0 x	00008000h
`SEXT A0,1`	x/16	00008000h	1 x 0 x	FFFF8000h
`SEXT A0,1`	x/15	00008000h	0 x 1 x	00000000h

Syntax

SLA *constant, Rd*

Execution

left-shift Rd by constant → Rd

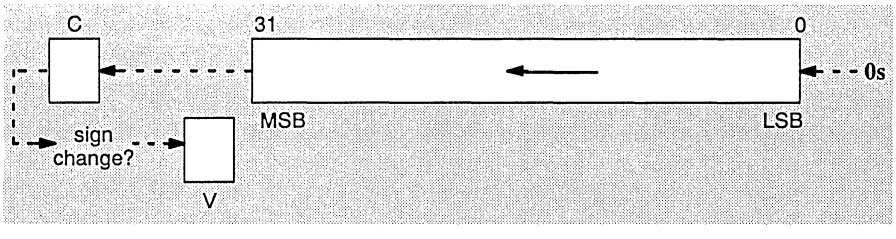
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	constant					R	Rd			

Description

SLA left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by a 5-bit constant; this is a value between 0 and 31.

As shown in the diagram, 0s are shifted into the LSBs. The last bit shifted out of the destination register (the original value of bit [32 – constant]) is shifted into the carry bit. If either the new sign bit (N) or any of the bits shifted out of the register differ from the original sign bit, the overflow bit (V) is set.



The assembler accepts only absolute expressions for the shift count. If the shift count is greater than 31, the assembler issues a warning and sets the constant to its 5 LSBs.

Note that SLA executes slower than SLL because it provides overflow detection.

Machine States

3

Status Bits

N *1* if the result is negative, *0* otherwise
C Set to the value of bit [32 – constant], *0* for shift count of 0
Z *1* if a 0 result generated, *0* otherwise
V *1* if the MSB changes during shift operation, *0* otherwise

Examples

Code	Before A1	After A1	N C Z V
SLA 0,A1	33333333h	33333333h	0 0 0 0
SLA 0,A1	CCCCCCCCh	CCCCCCCh	1 0 0 0
SLA 1,A1	CCCCCCCCh	99999998h	1 1 0 0
SLA 2,A1	33333333h	CCCCCCCCh	1 0 0 1
SLA 2,A1	CCCCCCCCh	33333330h	0 1 0 1
SLA 3,A1	CCCCCCCCh	66666660h	0 0 0 1
SLA 5,A1	CCCCCCCCh	99999980h	1 1 0 1
SLA 30,A1	CCCCCCCCh	00000000h	0 1 1 1
SLA 31,A1	CCCCCCCCh	00000000h	0 0 1 1
SLA 31,A1	00000000h	00000000h	0 0 1 0

Syntax

SLA Rs, Rd

Execution

left-shift Rd by 5 LSBs of Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	Rs				R	Rd			

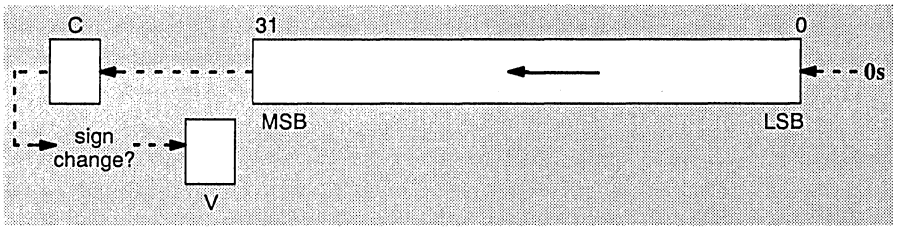
Description

SLA left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by the 5 LSBs of Rs (the 27 MSBs are ignored); this produces a shift count from 0 to 31.

The last bit shifted out of the destination register (the original value of bit [32–Rs]) is shifted into the carry bit. If either the new sign bit (N) or any of the bits shifted out of the register differ from the original sign bit, the overflow bit (V) is set.

31	30	29	28	26	25	22	21	11	10—6	5	4—0
N	C	Z	V	BF	IX	SS	IE	FE1	FS1	FE0	FS0

Note: Shaded portions are reserved.



Note that SLA executes slower than SLL because it provides overflow detection.

Machine States

3

Status Bits

N *1* if the result is negative, *0* otherwise
C Set to the value of [32 – Rs], *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V *1* if the MSB changes during shift operation, *0* otherwise

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	N C Z V
SLA A0,A1	00000	33333333h	33333333h	0 0 0 0
SLA A0,A1	00000	CCCCCCCCh	CCCCCCC Ch	1 0 0 0
SLA A0,A1	00001	CCCCCCCCh	99999998h	1 1 0 0
SLA A0,A1	00010	33333333h	CCCCCCCCh	1 0 0 1
SLA A0,A1	00010	CCCCCCCCh	33333330h	0 1 0 1
SLA A0,A1	00011	CCCCCCCCh	66666660h	0 0 0 1
SLA A0,A1	00101	CCCCCCCCh	99999980h	1 1 0 1
SLA A0,A1	11110	CCCCCCCCh	00000000h	0 1 1 1
SLA A0,A1	11111	CCCCCCCCh	00000000h	0 0 1 1
SLA A0,A1	11111	00000000h	00000000h	0 0 1 0

Syntax

SLL *constant, Rd*

Execution

left-shift Rd by constant → Rd

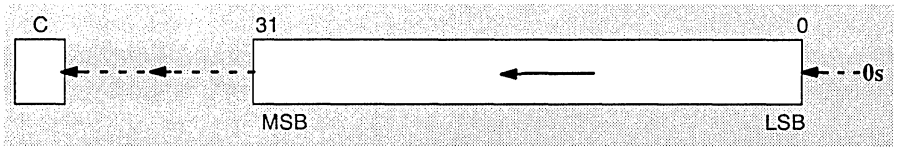
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	constant					R	Rd			

Description

SLL left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by a 5-bit constant, which is a value between 0 and 31.

The last bit shifted out of the destination register (the original value of bit [32 – constant]) is shifted into the carry bit. 0s are shifted into the LSBs. This instruction differs from the SLA instruction only in its effect on the overflow (V) bit.



The assembler only accepts absolute expressions for the shift count. If the specified shift count is greater than 31, the assembler issues a warning and sets the constants to its 5 LSBs.

Machine States

1

Status Bits

N Unaffected
C set to the value of bit [32 – constant], *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A1	A1	N C Z V
`SLL 0,A1`	00000000h	00000000h	x 0 1 x
`SLL 0,A1`	88888888h	88888888h	x 0 0 x
`SLL 1,A1`	88888888h	11111110h	x 1 0 x
`SLL 4,A1`	88888888h	88888880h	x 0 0 x
`SLL 30,A1`	FFFFFFFCh	00000000h	x 1 1 x
`SLL 31,A1`	FFFFFFFCh	00000000h	x 0 1 x

Syntax

SLL *Rs, Rd*

Execution

left-shift Rd by 5 LSBs of Rs → Rd

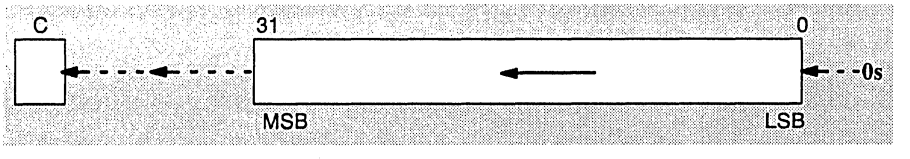
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	Rs				R	Rd			

Description

SLL left-shifts the contents of the destination register by a specified number of bits. The shift count is specified by the 5 LSBs of Rs (the 27 MSBs are ignored); this produces a shift count between 0 and 31.

The last bit shifted out of the destination register (the original value of bit [32 – Rs]) is shifted into the carry bit. 0s are shifted into the LSBs. This instruction differs from the SLA instruction only in its effect on the overflow (V) bit.



Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C set to the value of bit [32 – Rs], *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	N C Z V
`SLL A0,A1`	00000	00000000h	00000000h	x 0 1 x
`SLL A0,A1`	00000	88888888h	88888888h	x 0 0 x
`SLL A0,A1`	00001	88888888h	11111110h	x 1 0 x
`SLL A0,A1`	00100	88888888h	88888880h	x 0 0 x
`SLL A0,A1`	11110	FFFFFFFCh	00000000h	x 1 1 x
`SLL A0,A1`	11111	FFFFFFFCh	00000000h	x 0 1 x

Syntax

SRA *constant, Rd*

Execution

right-shift Rd by constant → Rd

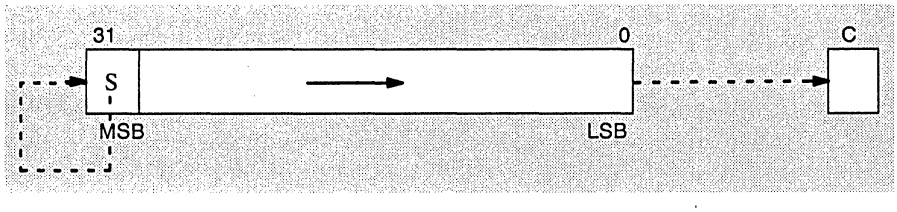
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	2s complement of constant					R	Rd			

Description

SRA right-shifts the contents of the destination register by a specified number of bits. The shift count is specified by constant which is a 5-bit immediate value; this produces a shift count of 0 to 31.

The last bit shifted out of the destination register (the original value of [constant–1]) is shifted into the carry bit. The sign bit (MSB) is extended into the MSBs.



The assembler accepts only absolute expressions for the shift count. If the specified shift amount is greater than 31, the assembler issues a warning, takes the 2s complement of the constant and places it in the opcode.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C Set to the value of [constant – 1], *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before A1	After A1	N C Z V
`SRA 0,A1`	00000000h	00000000h	0 0 1 x
`SRA 0,A1`	FFFF0000h	FFFF0000h	1 0 0 x
`SRA 8,A1`	7FFF0000h	007FFF00h	0 0 0 x
`SRA 8,A1`	FFFF0000h	FFFFFF00h	1 0 0 x
`SRA 30,A1`	7FFF0000h	00000001h	0 1 0 x
`SRA 31,A1`	7FFF0000h	00000000h	0 1 1 x
`SRA 31,A1`	FFFF0000h	FFFFFFFFh	1 1 0 x

Syntax

SRA *Rs, Rd*

Execution

right-shift Rd by 2s complement of 5 LSBs in Rs → Rd

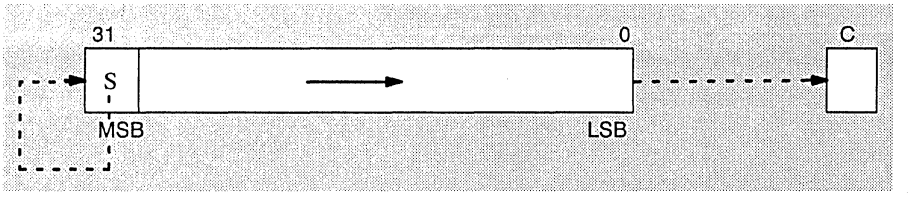
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	Rs				R	Rd			

Description

SRA right-shifts the contents of the destination register by a specified number of bits. The shift amount is specified by the 2s complement of the 5 LSBs of Rs (the 27 MSBs of Rs are ignored); this produces a shift count between 0 and 31.

The last bit shifted out of the destination register (the original value of bit [shift amount – 1]) is shifted into the carry bit. The sign bit (MSB) is extended into the MSBs.



Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C Set to the value of bit [shift amount – 1], *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	N C Z V
SRA A0,A1	00000	00000000h	00000000h	0 0 1 x
SRA A0,A1	00000	FFFF0000h	FFFF0000h	1 0 0 x
SRA A0,A1	11111	7FFF0000h	3FFF8000h	0 0 0 x
SRA A0,A1	11111	FFFF0000h	FFFF8000h	1 0 0 x
SRA A0,A1	11000	7FFF0000h	007FFF00h	0 0 0 x
SRA A0,A1	11000	FFFF0000h	FFFFFF00h	1 0 0 x
SRA A0,A1	00010	7FFF0000h	00000001h	0 1 0 x
SRA A0,A1	00001	7FFF0000h	00000000h	0 1 1 x
SRA A0,A1	00001	FFFF0000h	FFFFFFFFh	1 1 0 x

Syntax

SRL *constant, Rd*

Execution

right-shift Rd by constant → Rd

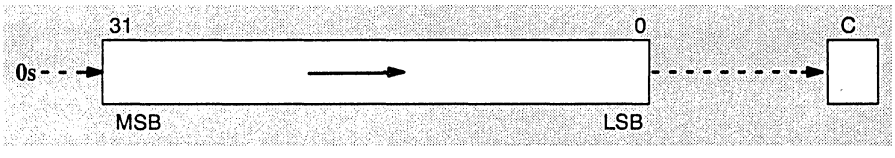
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	2s complement of constant					R	Rd			

Description

SRL right-shifts the contents of the destination register by a specified number of bits. The shift amount is specified by the constant which is a 5-bit immediate value; this produces a shift count between 0 and 31.

The last bit shifted out of the destination register (the original value of [constant–1]) is shifted into the carry bit. 0s are shifted into the MSBs.



The assembler accepts only absolute expressions for the shift count. If the specified shift amount is greater than 31, the assembler issues a warning, takes the 2s complement of the constant and places it in the opcode.

Machine States

1

Status Bits

N Unaffected
C Set to the value of [constant – 1], *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A1	A1	N C Z V
`SRL 0,A1`	00000000h	00000000h	x 0 1 x
`SRL 0,A1`	7FFFFFFFh	7FFFFFFFh	x 0 0 x
`SRL 1,A1`	7FFFFFFFh	3FFFFFFFh	x 1 0 x
`SRL 8,A1`	7FFF0000h	007FFF00h	x 0 0 x
`SRL 30,A1`	7FFF0000h	00000001h	x 1 0 x
`SRL 31,A1`	7FFF0000h	00000000h	x 1 1 x
`SRL 31,A1`	3FFF0000h	00000000h	x 0 1 x

Syntax

SRL *Rs, Rd*

Execution

right-shift Rd by 2s complement of 5 LSBs in Rs → Rd

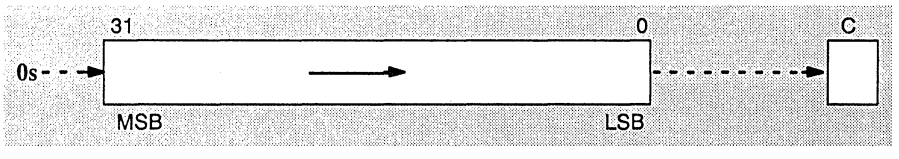
Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	1	Rs				R	Rd			

Description

SRL right-shifts the contents of the destination register by a specified number of bits. The shift amount is specified by the 2s complement of the 5 LSBs of Rs (the 27 MSBs of Rs are ignored); this produces a shift value of 0 to 31.

The last bit shifted out of the destination register (the original value of bit [shift amount – 1]) is shifted into the carry bit. 0s are shifted into the MSBs.



Machine States

1

Status Bits

N Unaffected
C Set to the value of bit [shift amount – 1], *0* for shift count of 0
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	5 LSBs A0	A1	A1	N C Z V
SRL A0,A1	00000	00000000h	00000000h	x 0 1 x
SRL A0,A1	00000	7FFFFFFFh	7FFFFFFFh	x 0 0 x
SRL A0,A1	11111	7FFFFFFFh	3FFFFFFFh	x 1 0 x
SRL A0,A1	11000	7FFF0000h	007FFF00h	x 0 0 x
SRL A0,A1	00010	7FFF0000h	00000001h	x 1 0 x
SRL A0,A1	00001	7FFF0000h	00000000h	x 1 1 x
SRL A0,A1	00001	3FFF0000h	00000000h	x 0 1 x

Syntax

SUB *Rs, Rd*

Execution

Rd – Rs → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	Rs				R	Rd			

Description

SUB subtracts the contents of the source register from the contents of the destination register and stores the result in the destination register.

You can accomplish multiple-precision arithmetic by using SUB in conjunction with the SUBB instruction.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before		After	
	A0	A1	N C Z V	A0
SUB A1,A0	7FFFFFF2h	7FFFFFF1h	0 0 0 0	00000001h
SUB A1,A0	7FFFFFF2h	7FFFFFF2h	0 0 1 0	00000000h
SUB A1,A0	7FFFFFF1h	7FFFFFF2h	1 1 0 0	FFFFFFFFh
SUB A1,A0	7FFFFFF1h	FFFFFFFFh	0 1 0 0	7FFFFFF2h
SUB A1,A0	7FFFFFFFh	FFFFFFFFh	1 1 0 1	80000000h
SUB A1,A0	FFFFFFFDh	FFFFFFFFh	1 1 0 0	FFFFFFFEh
SUB A1,A0	FFFFFFFDh	FFFFFFFDh	0 0 1 0	00000000h
SUB A1,A0	FFFFFFFEh	FFFFFFFDh	0 0 0 0	00000001h
SUB A1,A0	FFFFFFFFh	00000001h	1 0 0 0	FFFFFFFEh
SUB A1,A0	80000000h	00000001h	0 0 0 1	7FFFFFFFh

Syntax

SUBB *Rs, Rd*

Execution

Rd – Rs – C → Rd (the carry bit acts as a borrow)

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	Rs				R	Rd			

Description

SUBB subtracts both the contents of the source register and the carry bit from the contents of the destination register, and stores the result in the destination register.

You can use this instruction with the SUB, SUBK, and SUBI instructions for extended-precision arithmetic.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before			After	
	C	A0	A1	N C Z V	A0
SUBB A1,A0	0	00000002h	00000001h	0 0 0 0	00000001h
SUBB A1,A0	1	00000002h	00000001h	0 0 1 0	00000000h
SUBB A1,A0	0	00000002h	00000002h	0 0 1 0	00000000h
SUBB A1,A0	1	00000002h	00000002h	1 1 0 0	FFFFFFFFh
SUBB A1,A0	0	00000002h	00000003h	1 1 0 0	FFFFFFFFh
SUBB A1,A0	0	7FFFFFFEh	FFFFFFFFh	0 1 0 0	7FFFFFFFh
SUBB A1,A0	0	7FFFFFFEh	FFFFFFFEh	1 1 0 1	80000000h
SUBB A1,A0	1	7FFFFFFEh	FFFFFFFEh	0 1 0 0	7FFFFFFFh
SUBB A1,A0	0	FFFFFFFEh	FFFFFFFFh	1 1 0 0	FFFFFFFFh
SUBB A1,A0	0	FFFFFFFEh	FFFFFFFEh	0 0 1 0	00000000h
SUBB A1,A0	1	FFFFFFFEh	FFFFFFFEh	1 1 0 0	FFFFFFFFh
SUBB A1,A0	0	FFFFFFFEh	FFFFFFFDh	0 0 0 0	00000001h
SUBB A1,A0	1	FFFFFFFEh	FFFFFFFDh	0 0 1 0	00000000h
SUBB A1,A0	0	80000001h	00000001h	1 0 0 0	80000000h
SUBB A1,A0	1	80000001h	00000001h	0 0 0 1	7FFFFFFFh
SUBB A1,A0	0	80000001h	00000002h	0 0 0 1	7FFFFFFFh

Syntax

SUBI *IW*, *Rd* [, **W**]

Execution

Rd – IW → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	1	1	R	Rd			
1s complement of IW															

Description

SUBI subtracts a sign-extended, 16-bit immediate value from the contents of the destination register, and stores the result in the destination register. (The *IL* in the syntax represents a sign-extended, 16-bit immediate value.)

The assembler uses this form of the SUBI instruction if the immediate value was previously defined and is in the range –32,768 to 32,767. You can force the assembler to use the short form by following the register operand with **,W**:

```
SUBI  IW, Rd, W
```

The assembler truncates any upper bits and issues an appropriate warning message. You can accomplish multiple-precision arithmetic by using SUBI in conjunction with the SUBB instruction.

Machine States

2

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before A0	After A0	N C Z V
SUBI 32765,A0	00007FFEh	00000001h	0 0 0 0
SUBI 32766,A0	00007FFEh	00000000h	0 0 1 0
SUBI 32767,A0	00007FFEh	FFFFFFFFh	1 1 0 0
SUBI 32766,A0	80007FFEh	80000000h	1 0 0 0
SUBI 32767,A0	80007FFEh	7FFFFFFFh	0 0 0 1
SUBI –32766,A0	FFFF8001h	FFFFFFFFh	1 1 0 0
SUBI –32767,A0	FFFF8001h	00000000h	0 0 1 0
SUBI –32768,A0	FFFF8001h	00000001h	0 0 0 0
SUBI –32767,A0	FFFF8000h	7FFFFFFFh	0 1 0 0
SUBI –32768,A0	7FFF8000h	80000000h	1 1 0 1

Syntax

SUBI *IL, Rd* [**, L**]

Execution

Rd – IL → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	R	Rd			
1s complement of 16 LSBs of IL															
1s complement of 16 MSBs of IL															

Description

SUBI subtracts a signed 32-bit immediate value from the contents of the destination register, and stores the result in the destination register. (The *IL* in the syntax represents a signed 32-bit immediate value.)

The assembler uses this version of the SUBI instruction if it cannot use the SUBI IW,Rd opcode, or if you request the long opcode by following the register operand with **,L**:

```
SUBI  IL, Rd, L
```

You can accomplish multiple-precision arithmetic by using SUBI in conjunction with the SUBB instruction.

Machine States

2 if immediate data is long-word aligned
3 if immediate data is not long-word aligned

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before A0	After A0	N C Z V
SUBI 2147483647,A0	7FFFFFFFh	00000000h	0 0 0 1
SUBI 32768,A0	00008001h	00000001h	0 0 0 0
SUBI 32769,A0	00008001h	00000000h	0 0 1 0
SUBI 32770,A0	00008001h	FFFFFFFFh	1 1 0 0
SUBI 32768,A0	80008000h	80000000h	1 0 0 0
SUBI 32769,A0	80008000h	7FFFFFFFh	0 0 0 1
SUBI –2147483648,A0	80000000h	00000000h	0 0 1 0
SUBI –32769,A0	FFFF7FFEh	FFFFFFFFh	1 1 0 0
SUBI –32770,A0	FFFF7FFEh	00000000h	0 0 1 0
SUBI –32771,A0	FFFF7FFEh	00000001h	0 0 0 0
SUBI –32770,A0	7FFF7FFDh	7FFFFFFFh	0 1 0 0
SUBI –32771,A0	7FFF7FFDh	80000000h	1 1 0 1

Syntax

SUBK *constant, Rd*

Execution

Rd – constant → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	constant					R	Rd			

Description

SUBK subtracts the 5-bit constant from the contents of the destination register; the result is stored in the destination register. The constant is an unsigned number in the range 1—32. Note that constant=0 in the opcode corresponds to the value 32; the assembler converts the value 32 to 0. Using this instruction, the assembler issues an error if you try to subtract 0 from a register.

You can accomplish multiple-precision arithmetic by using SUBK in conjunction with the SUBB instruction.

Machine States

1

Status Bits

N *1* if the result is negative, *0* otherwise
C *1* if there is a borrow, *0* otherwise
Z *1* if the result is 0, *0* otherwise
V *1* if there is an overflow, *0* otherwise

Examples

Code	Before A0	After A0	N C Z V
`SUBK 5,A0`	00000009h	00000004h	0 0 0 0
`SUBK 9,A0`	00000009h	00000000h	0 0 1 0
`SUBK 32,A0`	00000009h	FFFFFFE9h	1 1 0 0
`SUBK 1,A0`	80000000h	7FFFFFFFh	0 0 0 1

Syntax

SUBXY *Rs, Rd*

Execution

Rd.**X** – Rs.**X** → Rd.**X**
Rd.**Y** – Rs.**Y** → Rd.**Y**

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	Rs				R	Rd			

Description

SUBXY subtracts the source X and Y values individually from the destination X and Y values; the result is stored in the destination register.

You can use this instruction for manipulating XY addresses; it is particularly useful for incremental figure drawing. These addresses are stored as XY pairs in the register file.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N *1* if source X field = destination X field, *0* otherwise
C *1* if source Y field > destination Y field, *0* otherwise
Z *1* if source Y field = destination Y field, *0* otherwise
V *1* if source X field > destination X field, *0* otherwise

Examples

Code	Before		After	
	A0	**A1**	**A0**	**N C Z V**
`SUBXY A1,A0`	00090009h	00010001h	00080008h	0 0 0 0
`SUBXY A1,A0`	00090009h	00090001h	00000008h	0 0 1 0
`SUBXY A1,A0`	00090009h	00010009h	00080000h	1 0 0 0
`SUBXY A1,A0`	00090009h	00090009h	00000000h	1 0 1 0
`SUBXY A1,A0`	00090009h	00000010h	0009FFF9h	0 0 0 1
`SUBXY A1,A0`	00090009h	00090010h	0000FFF9h	0 0 1 1
`SUBXY A1,A0`	00090009h	00100000h	FFF90009h	0 1 0 0
`SUBXY A1,A0`	00090009h	00100009h	FFF90000h	1 1 0 0
`SUBXY A1,A0`	00090009h	00100010h	FFF9FFF9h	0 1 0 1

Syntax

SWAPF **Rs, Rd,0*

Execution

Field specified by *Rs and FS0 → Rd
Rd → field specified by *Rs and FS0

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	Rs				R	Rd			

Description

This instruction performs a read (modify) write operation on a field in the memory space. It exchanges the field specified by the contents of Rs and FS0 with Rd. The new contents of Rd are right-justified and either sign- or zero-extended, depending on the value of FE0.

The main reason for the inclusion of this instruction is to allow the implementation of the *test and set* and *test and clear* operations needed for the lowest level of interprocess and interprocessor synchronization.

Note that this instruction does *not* complete until the write is complete (implicit MWAIT). This makes the instruction useful in some I/O register operations. Once the instruction starts, host access requests will not be granted until all the memory SWAPF accesses required are complete. If the read (modify) write is interrupted after the read by a screen refresh or a loss of of bus grant ($\overline{GI}$ high), or if a retry or bus fault occurs at any time during the cycle, the operation is restarted from the beginning of the read. This makes the operation indivisible. The bus lock status code is output during all SWAPF cycles.

Note:

The following restrictions apply to SWAPF:

1) The field must **not** span a 32-bit word boundary. The field is ignored if any part of it is not contained in the same 32-bit word specified by the bit address contained in Rs.

2) If SWAPF is used to access 16-bit memories, any part of the field not contained in the first 16-bit word is ignored.

Machine States

Refer to Section 15.1 on page 15-2.

Status Bits

N *1* if the field-extended data moved to register is negative, *0* otherwise.
C Unaffected
Z *1* if the field-extended data moved to register 0, *0* otherwise.
V *0*

Examples

```
                                    ;Test and Set—wait for resource
            SETF   1,0,0            ;Single bit
            MOVI   SEMA,A1          ;Bit to test and set
WAS_SET:    MOVK   1,A0             ;Set is not already set
            SWAPF  *A1,A0           ;Test and set
            JRNE   WAS_SET          ;Already set—did not get resource
                                    ;Test and Clear—wait for resource
            SETF   1,0,0            ;Single bit
            MOVI   SEMA,A1          ;Bit to test and clear
WAS_CLR:    CLR    A0               ;Clear if not already clear
            SWAPF  *A1,A0           ;Test and clear
            JREQ   WAS_CLR          ;Already clear—did not get resource
                                    ;Graphics mode save
            MOVI   CONTROL+5,A1     ;Point at CONTROL register
            SETF   10,0,0           ;Ten bits
            MOVI   NEWMODE,A0       ;New value
            SWAPF  *A1,A0           ;Read oldmode, set new mode
            CALL   GRAPHOP          ;Perform some operation
            MOVE   A0,*A1,0         ;Restore old mode
```

Syntax

TFILL XY

Execution

COLOR1 pixels fill the horizontal line from (X_1, Y) to (X_2, Y) then

$X_1 := X_1 + DX_1$
$X_2 := X_2 + DX_2$
$Y := Y + 1$

Instruction Words

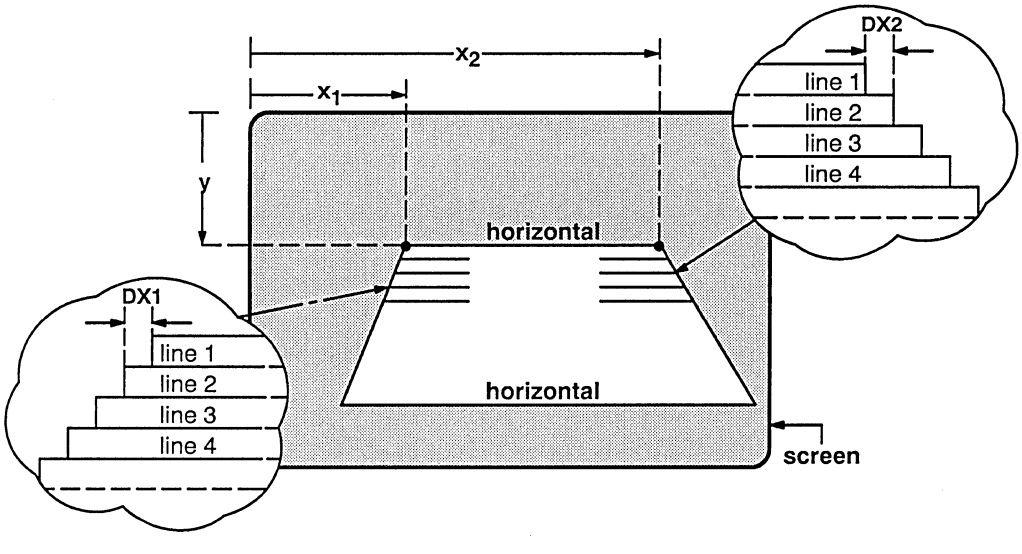
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	1	1	1	1	1	0	1	0

Description

TFILL draws a horizontal line and then adjusts its implied operands. The operands are set so that each subsequent call to TFILL will draw one more horizontal line, creating lines that build up to form a trapezoid.

The trapezoid is defined as shown in the diagram:

Figure 13–11. A Trapezoidal Fill



Note that the coordinate parameters for this instruction are specified in the fixed-point format; that is, the 16 MSBs define the signed part of the coordinate and the 16 LSBs define the fractional part of the coordinate. This is the case for both X and Y coordinates, although the Y coordinate will never have a fractional part.

The DX_1 and DX_2 values can have fractional components. This allows for nonintegral slopes at the trapezoid sides. The fractional components are used to determine the new endpoints for the next line. However, only the 16 MSBs are used to determine the XY address of the endpoints.

Note that if $X_2 \leq X_1$ no pixels are drawn, but the contents of X_1 and X_2 are still incremented by DX_1 and DX_2 respectively.

Implied Operands

Register	Name	Format	Description
B0	SADDR	Fixed	X coordinate of X1
B1	SPTCH	Fixed	DX1 (adjustment for X1)
B2	DADDR	XY	Used as temp (not user determined)
B3	DPTCH	Linear	Destination pixel array pitch (usually screen pitch)
B4	OFFSET	Linear	Screen offset
B5	WSTART	XY	Window start
B6	WEND	XY	Window end
B7	DYDX	Fixed	X coordinate of of X2
B9	COLOR1	Pixel	Foreground color
B10	MADDR	Fixed	DX2 (adjustment for X2)
B11	MPTCH	Fixed	Y coordinate of X1 and X2

Address	Name	Description and Elements (Bits)
C00000B0h	CONTROL	PP — Pixel-processing operations (22 options) W — Window checking operation T — Transparency operation TMODE — Selects 1 of 3 transparency options
C0000140h	CONVDP	XY-to-linear conversion (destination pitch)
C0000150h	PSIZE	Pixel size (1,2,4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask – pixel format

To set up the initial values for X_1, X_2, and Y from 2 starting addresses (X_1, Y) and (X_2, Y), complete the following steps:

1) Use MOVY to copy the Y address into MPTCH.
2) Use SLL to shift the 2 XY addresses left by 16 bits. This results in 2 fixed-point X coordinates.
3) Use MOVY to copy the 2 X addresses into SADDR and DYDX, respectively.

Pixel Processing

Pixel processing can be used with this instruction. PPOP[[CONTROL]] specifies the pixel-processing operation that is applied to pixels as they are processed with the destination array. There are 16 Boolean and 6 arithmetic operations; the default case at reset is the *replace (S → D)* operation. Note that the destination data is read through the plane mask and then processed. The 6 arithmetic operations do not operate with a pixel size of 1 bit per pixel. For more information, refer to Section 12.8, Pixel Processing, on page 12-27.

Window Checking

The window operations can be used with this instruction. For more information, refer to Section 12.7, Window Checking, on page 12-19.

Transparency

You can enable transparency for this instruction by setting T[CONTROL] to 1. Select 1 of 3 transparency modes by setting TM[CONTROL]. For more information, refer to Section 12.9, Transparency, on page 12-36.

Interrupts

This instruction can be interrupted at a word or row boundary of the destination array. For more information, refer to Section 6.6, Interrupting Graphics Operations, on page 6-13.

Plane Masking

The plane mask is enabled for this instruction. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

This is an example of a C-compatible assembly routine which draws trapeziums on the screen, using the TFILL instruction.This function has 6 arguments:

(x1a, x2a, ya) – coordinates of top of trapezoid
(x1b, x2b, yb) – coordinates of bottom of trapezoid

This routine assumes the following registers have been initialized by the caller:

B-file registers	DPTCH, OFFSET, WSTART, WEND, and COLOR1
I/O registers	CONTROL, CONVDP, PSIZE and PMASK

```
STK     .set    A14                 ; C-parameter stack pointer
SADDR   .set    B0                  ; Source address register
SPTCH   .set    B1                  ; Source pitch register
DYDX    .set    B7                  ; Delta X/delta Y register
MADDR   .set    B10                 ; Mask address register
MPTCH   .set    B11                 ; Mask pitch register          .globl  _tfill
_tfill:
        mmtm    SP,B0,B1,B2,B7,B10,B11,B12,B13,B14
        move    STK,B14             ;get C-parameter stack into B-file
        move    *-B14,SADDR,1       ;pop x1a
        sll     16,SADDR            ;convert to fixed point
        move    *-B14,DYDX,1        ;pop x2a
        sll     16,DYDX             ;convert to fixed point
        move    *-B14,MPTCH,1       ;pop ya
        move    *-B14,SPTCH,1       ;pop x1b
        sll     16,SPTCH            ;convert to fixed point
        move    *-B14,B13,1         ;pop x2b
        sll     16,B13              ;convert to fixed point
        move    *-B14,B12,1         ;pop yb
        move    B14,STK             ;update C-parameter stack
        sub     SADDR,SPTCH         ;delta x1
        sub     DYDX,B13            ;delta x2
        sub     MPTCH,B12           ;delta y
        divs    B12,SPTCH           ;dx1
```

```
        divs    B12,B13             ;dx1 (cant use MADDR since divs
                                    ;requires odd numbered register)
        move    B13,MADDR           ;copy into MADDR
        sll     16,MPTCH            ;convert y to fixed point
loop:
        tfill
        dsjs    B12,loop
        mfm     SP,B0,B1,B2,B7,B10,B11,B12,B13,B14
        rets    2
```

Syntax

TRAP N

Execution

PC → –*SP
ST → –*SP
trap vector *N* → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	*N*				

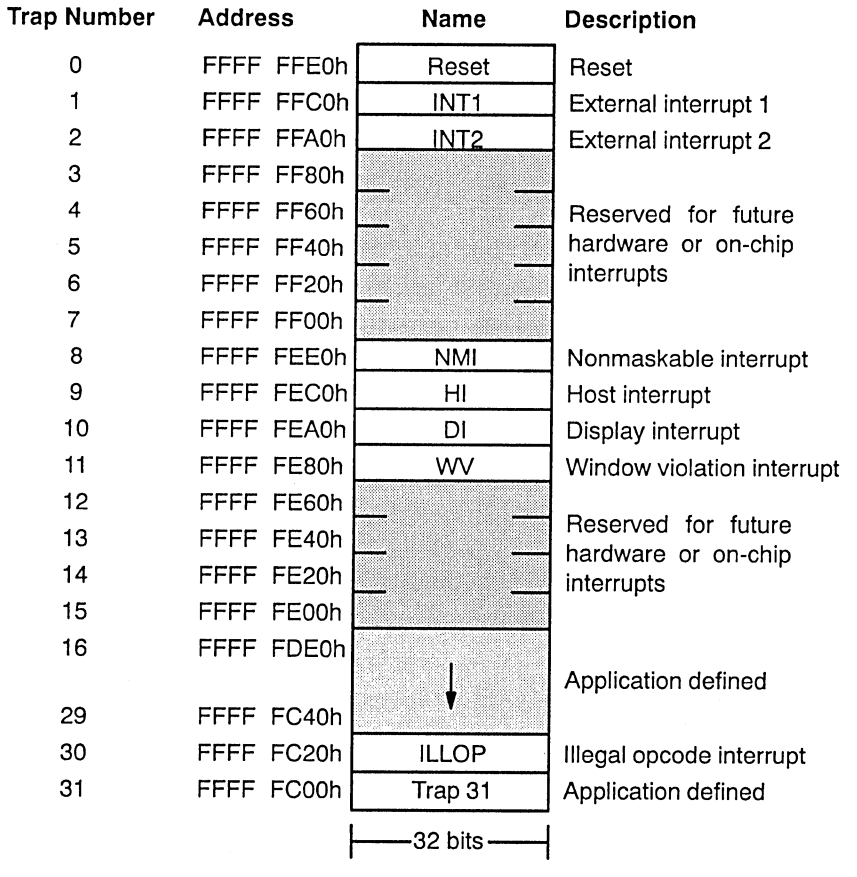
Description

TRAP executes a software interrupt. The *N* parameter is a trap number from 0 to 31 that selects the trap to be executed. During a software interrupt:

- The 32-bit return address, PC (the address of the next instruction), is pushed on the stack.
- The 32-bit status register, ST, is pushed on the stack.
- The stack pointer, SP, is decremented by 64.
- The IE (interrupt enable) bit in ST is set to 0, disabling maskable interrupts, and ST is set to 00000010h.
- Finally, the trap vector is loaded into the PC.

The TMS34020 generates the trap vector addresses as shown on the following page:

Figure 13–12. Vector Address Map



Note: Traps 0—31 may be accessed by either TRAP or TRAPL instructions.

The stack, which is located in external memory, grows toward lower addresses. Unlike an interrupt, a software trap cannot be disabled.

Note:

1) TRAP 0 differs from all other traps; it does not save the old status register or program counter. This may be useful in cases where the stack pointer is corrupted or uninitialized; such a situation could cause an erroneous write.

2) NMIM[HSTCTLH] does not affect the operation of TRAP 8: the PC and ST are always pushed onto the stack.

Machine States

7 if TRAP 0,
else 10 if ST aligned,
else 12

Status Bits

N *0*
C *0*
Z *0*
V *0*

Examples

Assume that memory contains the following values before instruction execution:

Address	Data
FFFFFC00	0000
FFFFFC10	FFE0
FFFFFC20	0000
FFFFFC30	FFD0
FFFFFFC0	0000
FFFFFFD0	FFB0
FFFFFFE0	0000
FFFFFFF0	FFA0

Code	Before		After		
	PC	SP	PC	SP	ST
TRAP 0	xxxxxxxxh	80000000	FFA00000	80000000h	00000010h
TRAP 1	xxxxxxxxh	80000000	FFB00000	7FFFFFC0h	00000010h
TRAP 30	xxxxxxxxh	80000000	FFD00000	7FFFFFC0h	00000010h
TRAP 31	xxxxxxxxh	80000000	FFE00000	7FFFFFC0h	00000010h

Syntax

TRAPL

Execution

PC → –*SP
ST → –*SP
trap vector *N* → PC

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	1
16-bit trap number *N*															

Description

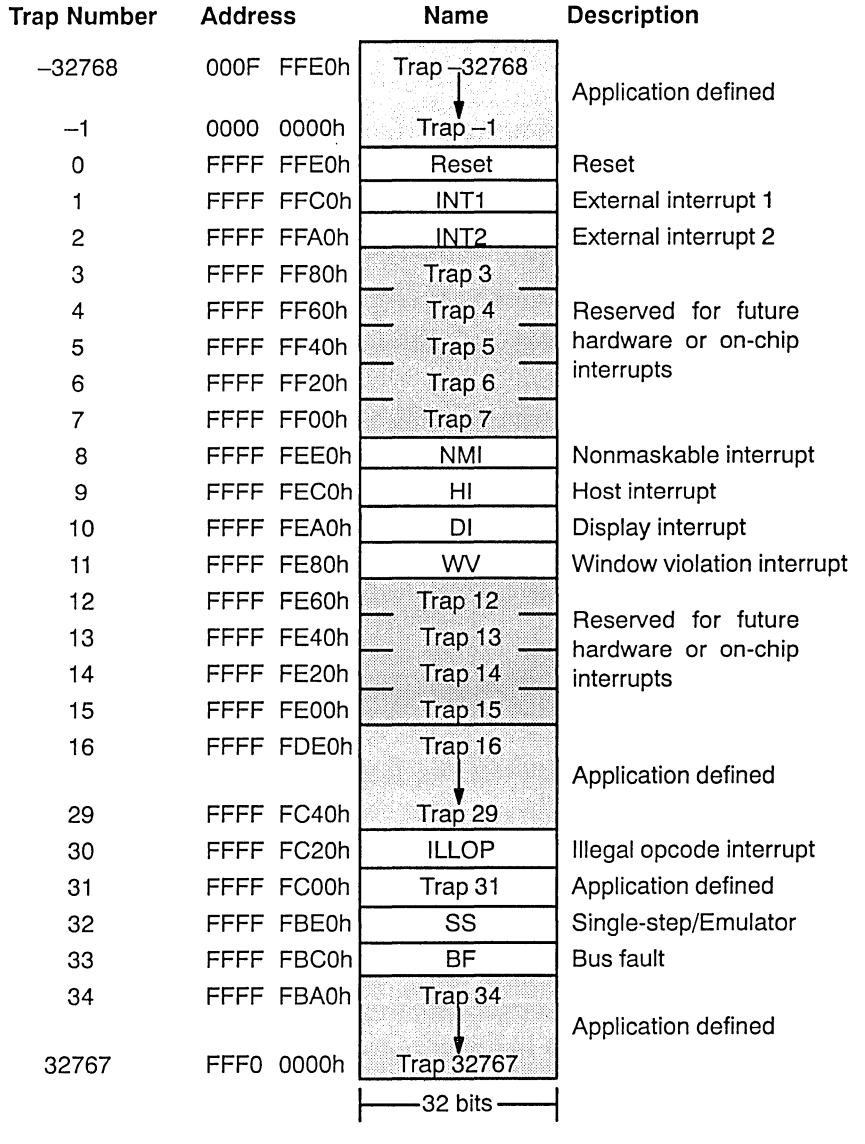
TRAPL executes a software interrupt. The *N* parameter is a signed number from –32,768 to 32,767. The trap address is formed by taking the 16-bit signed immediate operand *N*, shifting it left by 5 bits and then sign-extending it. TRAPL can cover a significantly larger address range than the TRAP instruction. During a software interrupt:

- The 32-bit return address, PC (the address of the next instruction), is pushed on the stack.
- The 32-bit status register, ST, is pushed on the stack.
- The stack pointer, SP, is decremented by 64.
- The IE (interrupt enable) bit in ST is set to 0, disabling maskable interrupts, and ST is set to 00000010h.
- Finally, the trap vector is loaded into the PC.

Note that unlike TRAP 0, the TRAPL 0 is not treated as an exception. That is, TRAPL 0 saves the PC and ST on the stack, whereas TRAP 0 does not.

The TMS34020 generates the trap vector addresses as shown on the following page:

Figure 13–13. Vector Address Map



Notes: 1) Traps (–1)— (–32,768) use the memory at the bottom of the address space for vector addresses. Traps 0—32,767 use the memory at the top of the address space.

2) Traps 0—31 may be accessed by either TRAP or TRAPL instructions.

3) Traps (–1)— (–32,768) and 32—32,767 are only accessed by TRAPL.

4) Traps 3—7 and 12—15 are reserved for future interrupts.

Machine States

10 if ST is aligned
else 12

Status Bits

N *0*
C *0*
Z *0*
V *0*

Examples

Assume that memory contains the following values before instruction execution:

Address	Data
FFFFFBC0	0000
FFFFFBD0	FF0E
FFFFFC00	0000
FFFFFC10	FFE0
FFFFFFE0	0000
FFFFFFF0	FFA0
00000000	0000
00000010	FF0F

Code	Before		After		
	PC	SP	PC	SP	ST
TRAPL –1	xxxxxxxxh	80000000	FF0F0000	7FFFFFC0h	00000010h
TRAPL 0	xxxxxxxxh	80000000	FFA00000	7FFFFFC0h	00000010h
TRAPL 31	xxxxxxxxh	80000000	FFE00000	7FFFFFC0h	00000010h
TRAPL 33	xxxxxxxxh	80000000	FF0E0000	7FFFFFC0h	00000010h

Syntax

VBLT B, L

Execution

Binary pixel array → linear pixel array using VRAM block write

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	0	1	0	1	1	1

Description

VBLT moves a binary array of pixels defined by SADDR and DYDX to a corresponding block defined by DADDR and DYDX using VRAM block mode expansion. Both SADDR and DADDR contain linear starting addresses.There is an expansion implicit in the transfer, such that a bit value of *1* in the source data is written to the destination array as a COLOR1 pixel (from the VRAM color register). Source bits of the value *0* leave the corresponding destination pixel unchanged. Note this instruction assumes that the VRAM color register has been loaded by the VLCOL instruction. For more information, refer to subsection 12.5.4, VRAM Block Mode, page 12-14.

Note:

1) DPTCH must be an integral multiple of 80h, **and**

2) this instruction works only if the PSIZE is 4,8,16, and 32.

Implied Operands

Register	Name	Format	Description
B0	SADDR	Linear	Source pixel array address
B1	SPTCH	Linear	Source pixel array pitch
B2	DADDR	Linear	Destination pixel array address
B3	DPTCH	Linear	Destination pixel array pitch
B7	DYDX	XY	Pixel array dimensions
B14	TEMP	Temp	Intermediate value

Address	Name	Description and Elements (Bits)
C0000150h	PSIZE	Pixel size (4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format
C00001A0h	CONFIG	Bit 8 (VEN) enables VRAM write mask

Address	Name	Description
VRAM Color Register	Pixel	Must have COLOR1 pixels loaded using VLCOL
VRAM Write Mask	Pixel	Loaded automatically when PMASK is written and VEN = 1

Pixel Processing

Pixel processing is not possible with this instruction, because the pixel data is written from the VRAM color register into the VRAM memory array.

Window Checking Window checking **cannot** be used with this instruction.

Transparency Transparency bits are ignored. This instruction has a form of implicit transparency in that source pixels which are 0 correspond to destination pixels which are not changed.

Plane Masking The plane mask is implemented in the VRAM using the write mask function, enabled by VEN[[CONFIG]]. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Interrupts This instruction can be interrupted at a word or row boundary of the destination array. For more information, refer to Section 6.6, Interrupting Graphics Instructions, on page 6-13.

Corner Adjust Corner adjust **cannot** be used with this instruction.

Machine States complex instruction

Status Bits

N Undefined
C Undefined
Z Undefined
V Undefined

Example This is an example of a C-compatible assembly routine which draws a character on the screen using the VBLT instruction. It expects the following arguments on the C parameter stack: width, height, xleft, ytop, and a pointer to the start of the character data. The character data should be a binary representation of the character.

This routine makes the following assumptions:

- These B registers and I/O registers have been initialized by the calling program:

  B-file registers DPTCH, OFFSET, WSTART, WEND, COLOR0, COLOR1

  I/O registers CONTROL, CONVDP, PSIZE, PMASK and CONFIG

- The system contains a global flag _vblt_ok which is cleared if the VBLT is not possible. Reasons for this may be:
  - DPTCH is not an integral multiple of 80 hex
  - PSIZE is 1 or 2
  - Pixel processing is not set to replace
  - Transparency is not set to source equals 0
  - The system does not contain VRAMs that support this feature

```
STK     .set    A14                 ; C-parameter stack pointer
SADDR   .set    B0                  ; Source address register
SPTCH   .set    B1                  ; Source pitch register
DADDR   .set    B2                  ; Destination address register
DPTCH   .set    B3                  ; Dest. pitch register
DYDX    .set    B7                  ; Delta X/delta Y register
        .globl  _vblt     ; provide reference for external calls

        .ref    _vblt_ok            ; flag to enable VBLTs
_vblt:
        mmtm    SP,B0,B1,B2,B7,B10,B11,B12 ;save required registers
        move    STK,B10             ;move c-stack pointer into B-file
        move    *-B10,DYDX,1        ;get width
        move    DYDX,SPTCH          ;save the width as source pitch
        move    *-B10,B12,1         ;get height
        sll     16,B12
        movy    B12,DYDX            ;concatenate width & height
        move    *-B10,DADDR,1       ;get xleft
        move    *-B10,B12,1         ;get ytop
        move    *-B10,SADDR,1       ;get source address
        move    B10,STK             ;restore c-stack pointer
        sll     16,B12
        movy    B12,DADDR           ;concatenate xleft & ytop
        move    @_vblt_ok,A8,1      ;get state of vblt flag
        jrz     no_vblt
vblt:
        clip                        ;clip to the window
        jrz     exit                ;if outside the window, exit
        cvdxyl  DADDR               ;convert to linear dest address
        vlcol                       ;load VRAM color latches
        vblt    B,L                 ;perform linear vblt
        jruc    exit
no_vblt:
        pixblt  B,XY
exit:
        mmfm    SP,B0,B1,B2,B7,B10,B11,B12 ;restore required registers
rets    2
```

Syntax

VFILL L

Execution

Contents of VRAM color latch → array of pixels

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	0	1	0	1	1	1

Description

VFILL fills an array of pixels defined by DADDR and DYDX using the VRAM block mode writes. Note this instruction assumes that the VRAM color register has been loaded by the VLCOL instruction. For more information, refer to subsection 12.5.7, VRAM Block Mode Fill, on page 12-16.

Note:

1) DPTCH must be an integral multiple of 80h, **and**
2) this instruction works only for PSIZE's 4,8,16, and 32.

Implied Operands

Register	Name	Format	Description
B2	DADDR	Linear	Destination pixel block address
B3	DPTCH	Linear	Destination pixel block pitch
B7	DYDX	XY	Pixel block dimensions

Address	Name	Description and Elements (Bits)
C0000150h	PSIZE	Pixel size (4,8,16,32)
C0000160h	PMASK (32 bits)	Plane mask — pixel format
C00001A0h	CONFIG	Bit 8 (VEN) enables VRAM write mask

Address	Name	Description
VRAM Color Register	Pixel	Must have COLOR1 pixels loaded using VLCOL
VRAM Write Mask	Pixel	Loaded automatically when PMASK is written and VEN = 1

Pixel Processing

Pixel processing is not possible with this instruction, because the pixel data is written from the VRAM color register into the VRAM memory array.

Window Checking

Window checking **cannot** be used with this instruction.

Transparency

Transparency **cannot** be used with this instruction.

Plane Masking

The plane mask is implemented in the VRAM using the write mask function, enabled by VEN[[CONFIG]]. For more information, refer to Section 12.10, Plane Masking, on page 12-39.

Interrupts

This instruction can be interrupted at a word or row boundary of the destination array. For more information, refer to Section 6.6, Interrupting Graphics Instructions, on page 6-13.

Corner Adjust

There is no corner adjust for this instruction.

Machine States

complex instruction

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

This is an example of a C-compatible assembly routine which fills a rectangle on the screen. The routine takes these 4 arguments: width, height, xleft, and ytop. Note that the CLIP instruction is used to clip the rectangle to the screen.

This routine makes the following assumptions:

- The calling program sets up these registers:
  B-file registers DPTCH, OFFSET, WSTART, WEND and COLOR1
  I/O registers CONTROL, CONVDP, PSIZE, PMASK and CONFIG
- The system contains a global flag _vfill_ok which is cleared if the VFILL is not possible. Reasons for this may be:
  - DPTCH is not an integral multiple of 80 hex
  - PSIZE is 1 or 2
  - Pixel processing is not set to replace
  - Transparency is not set
  - The system does not contain VRAMs that support this feature

```
DADDR       .set   B2                    ;Destination address register
DYDX        .set   B7                    ;Delta X/delta Y register
CONTROL     .set   0C00000B0h            ;Control IO register
            .globl _fill_rect            ;provide reference for external calls
            .ref   _vfill_ok             ; flag to enable VFILLs
_fill_rect: mmtm   SP,B2,B7,B10,B11,B12 ;save required registers
            move   A14,B10               ;move c-stack pointer into B-file
            move   *-B10,DYDX,1          ;get width
            move   *-B10,B12,1           ;get height
            sll    16,B12
            movy   B12,DYDX              ;concatenate width & height
            move   *-B10,DADDR,1         ;get xleft
            move   *-B10,B12,1           ;get ytop
            move   B10,A14               ;restore c-stack pointer
            sll    16,B12
            movy   B12,DADDR             ;concatenate xleft & ytop
            move   @_vfill_ok,A8,1       ;get state of vfill flag
            jrz    no_vfill
            clip                         ;clip to the window
            jrz    exit                  ;if outside the window, exit
            cvdxyl DADDR                 ;convert to linear dest address
            vlcol                        ;load VRAM color latches
            vfill  L                     ;perform linear fill
            jruc   exit
no_vfill:   fill   XY                    ;fill the rectangle using standard fill
exit:       mmfm   SP,B2,B7,B10,B11,B12 ;restore required registers
            rets   2
```

Syntax

VLCOL

Execution

COLOR1 → Color Registers in all VRAMS

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0

Description

VLCOL writes the value in the COLOR1 register to the color registers in all external VRAMS. The field size is ignored and the flood write outputs to nominal address 0. This instruction should be executed before attempting to use VFILL or VBLT. This instruction performs color expansion in the VRAM as pixels are written. The VRAM color registers are used for this purpose.

Implied Operands

Register	Name	Format	Description
B9	COLOR1	Pixel	COLOR1

Machine States

2 (1)

Status Bits

N Unaffected
C Unaffected
Z Unaffected
V Unaffected

Example

This is an example of a C-compatible assembly routine which fills a rectangle on the screen. The routine takes these 4 arguments: width, height, xleft, and ytop. Note that the CLIP instruction is used to clip the rectangle to the screen.

This routine makes the following assumptions:

- These B-registers and I/O registers have been set up by the calling program:

  B-file registers DPTCH, OFFSET, WSTART, WEND and COLOR1

  I/O registers CONTROL, CONVDP, PSIZE, PMASK and CONFIG

- The system contains a global flag _vfill_ok which is cleared if the VFILL is not possible. Reasons for this may be:
  - DPTCH is not an integral multiple of 80 hex
  - PSIZE is 1 or 2
  - Pixel processing is not set to replace
  - Transparency is not set
  - The system does not contain VRAMs that support this feature

```
DADDR   .set    B2                      ;Destination address register
DYDX    .set    B7                      ;Delta X/delta Y register
CONTROL .set    0C00000B0h              ;Control IO register

        .globl  _fill_rect      ; provide reference for external calls
        .ref    _vfill_ok               ; flag to enable VFILLs

_fill_rect:
        mmtm    SP,B2,B7,B10,B11,B12    ;save required registers
        move    A14,B10         ;move c-stack pointer into B-file
        move    *-B10,DYDX,1            ;get width
        move    *-B10,B12,1             ;get height
        sll     16,B12
        movy    B12,DYDX                ;concatenate width & height
        move    *-B10,DADDR,1           ;get xleft
        move    *-B10,B12,1             ;get ytop
        move    B10,A14                 ;restore c-stack pointer
        sll     16,B12
        movy    B12,DADDR               ;concatenate xleft & ytop
        move    @_vfill_ok,A8,1         ;get state of vfill flag
        jrz     no_vfill
        clip                            ;clip to the window
        jrz     exit                    ;if outside the window, exit
        cvdxyl  DADDR                   ;convert to linear dest address
        vlcol                           ;load VRAM color latches
        vfill   L                       ;perform linear fill
        jruc    exit
no_vfill:
        fill    XY              ;fill the rectangle using standard fill
exit:
        mmfm    SP,B2,B7,B10,B11,B12    ;restore required registers
        rets    2
```

Syntax

XOR *Rs, Rd*

Execution

Rs XOR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rs				R	Rd			

Description

XOR bitwise-exclusive-ORs the contents of the source register with the contents of the destination register, and stores the result in the destination register.

You can use this instruction to clear registers (for example, `XOR B0,B0`); the CLR instruction also supports this function.

Rs and Rd must be in the same register file.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After	
	A0	A1	N C Z V	A1
`XOR A0,A1`	FFFFFFFFh	00000000h	x x 0 x	FFFFFFFFh
`XOR A0,A1`	FFFFFFFFh	AAAAAAAAh	x x 0 x	55555555h
`XOR A0,A1`	FFFFFFFFh	FFFFFFFFh	x x 1 x	00000000h

Syntax

XORI *IL, Rd*

Execution

IL XOR Rd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	1	1	0	R	Rd			
16 LSBs of IL															
16 MSBs of IL															

Description

XORI bit-wise exclusive-ORs a 32-bit immediate data with the contents of the destination register and stores the result in the destination register. (The *IL* parameter in the syntax above represents a 32-bit immediate value.)

Machine States

2 if the immediate data is long-word aligned
3 if the immediate data is not long-word aligned

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before	After	
	A0	**N C Z V**	**A0**
`XORI 0FFFFFFFFh,A0`	00000000h	x x 0 x	FFFFFFFFh
`XORI 0FFFFFFFFh,A0`	AAAAAAAAh	x x 0 x	55555555h
`XORI 0FFFFFFFFh,A0`	FFFFFFFFh	x x 1 x	00000000h
`XORI 00000000h,A0`	00000000h	x x 1 x	00000000h
`XORI 00000000h,A0`	FFFFFFFFh	x x 0 x	FFFFFFFFh

Syntax

ZEXT *Rd* [, *F*]

Execution

field in Rd → zero-extended field Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	F	1	0	0	1	R	Rd			

Description

ZEXT zero-extends a right-justified field in the destination register by zeroing all the nonfield bits in Rd. The size of the field is determined by the current field size. The optional *F* parameter, which must be specified as a 0 or a 1, selects the field size:

F=0 selects FS0 for the field size.
F=1 selects FS1 for the field size.

The default value for *F* is 0.

Machine States

1

Status Bits

N Unaffected
C Unaffected
Z *1* if the result is 0, *0* otherwise
V Unaffected

Examples

Code	Before		After		
	FS0	FS1	A0	N C Z V	A0
`ZEXT A0,0`	32	x	FFFFFFFFh	x x 0 x	FFFFFFFFh
`ZEXT A0,0`	31	x	FFFFFFFFh	x x 0 x	7FFFFFFFh
`ZEXT A0,0`	1	x	FFFFFFFFh	x x 0 x	00000001h
`ZEXT A0,0`	16	x	FFFF0000h	x x 1 x	00000000h
`ZEXT A0,1`	x	16	FFFF0000h	x x 1 x	00000000h

Chapter 14

TMS34082 Pseudo-ops

Many TMS34020 applications require floating-point operations. The TMS34082 Floating-Point Processor is designed specifically to serve as a coprocessor in a TMS34020 system. To extend the TMS34020's direct interface to the TMS34082, the TMS34020 supports a subset of the TMS34082 assembly-language instruction set by supplying a group of **TMS34082 pseudo-ops**. These pseudo-ops are special versions of the TMS34020's general-purpose coprocessor instructions. Instead of designing a protocol for sending instructions and data back and forth between the TMS34020 and TMS34082, you can use these pseudo-ops, which are hard-coded versions of instructions such as the CMOVCG instruction.

This chapter provides a general description of the pseudo-op protocol and provides an alphabetical reference to the TMS34082 pseudo-ops.

14.1 Overview and Key Features of the TMS34082

The TMS34082 is a high-speed floating-point processor, implemented in TI's advanced 1-micron CMOS technology. On a single chip, the TMS34082 combines a 16-bit sequencer, a 3-operand FPU, and 22 64-bit data registers. An instruction register controls FPU execution, and a status register retains the most recent FPU status outputs. The TMS34082 also contains 8 control registers and a 2-deep stack.

The TMS34082 is fully compatible with IEEE Standard 754–1985 for binary floating-point arithmetic. Floating-point operands can be in either single- or double-precision IEEE format.

Key features and benefits include

- Closely coupled with the TMS34020
  - Direct TMS34020 instruction extension
  - Multiple-TMS34082 capability
- Internal programs for vector, matrix, and graphics operation
- Fast multiply/accumulate cycle time
  - 40 MHz (TMS34082-40) . . . 50 ns
  - 32 MHz (TMS34082-32) . . . 60 ns
- External memory addressing capability
  - External program storage (up to 64K words)
  - External data storage (up to 64K words)
- Full IEEE standard 754–1985 compatibility
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Square root
  - Comparison
- Selectable data formats
  - 32-bit integer
  - 32-bit, single-precision floating-point value
  - 64-bit, double-precision floating-point value
- Supported by TMS34020 code-generation tools
- More than 30 complex instructions targeted at graphics math
- Use as a floating-point coprocessor eliminates the need for external logic interface
- Standardized approach to floating-point for full system compatibility
- Eliminates multiple-cycle software implementation
- Superior performance for 2-D and 3-D graphics applications

14.2 Pseudo-op Format

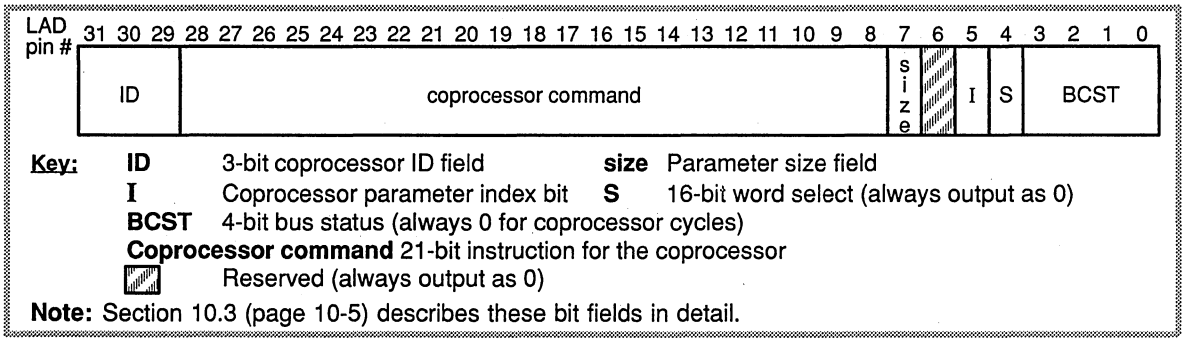
Section 10.2, Overview of the Coprocessor Interface (page 10-3), lists the TMS34020's general-purpose coprocessor instructions. Section 10.3, Format of Commands Passed to a Coprocessor (page 10-5), describes the general format of these instructions; specific implementations of this format depend entirely on the coprocessor that you choose to include in your system.

The TMS34020 provides a set of TMS34082 pseudo-ops. These pseudo-ops extend the general-purpose format by hard-coding TMS34082 opcodes into certain fields of the general-purpose coprocessor instructions.

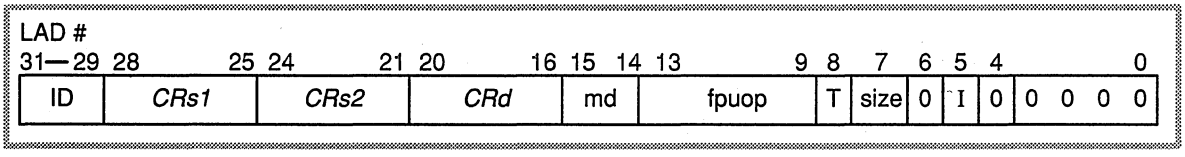
Figure 10–1 (which appears in Section10.3, page 10-5) is repeated below as Figure 14–1 *(a)*. This figure shows the general format of coprocessor information that is placed on the LAD bus. Figure 10–1 *(b)* shows the TMS34082 implementation of this format.

Figure 14–1. Coprocessor Instruction Information on the LAD bus

(a) General Format



(b) TMS34082-Specific Format



- As Figure 10–1 *(b)* shows, the **bus cycle status code** portion is 0000_2. This indicates that the local-memory cycle generated by this type of instruction is a coprocessor cycle.
- Bit 4, the **S** (16-bit word select) bit, is also 0; this indicates that only 32-bit accesses will occur.
- The **I** bit serves the same purpose for the TMS34082 as it does for other processors (refer to subsection 10.3.4, Coprocessor Parameter Index, on page 10-7).

- Bit 7 still serves as the **size** (parameter size) bit. The TMS34082 uses the LSB of the coprocessor command as $\overline{\mathbf{T}}$ (bit 8) that works with the size bit to identify the type and size of the parameter(s) that are passed to the TMS34082:

T	size	Operand Type
0	0	32-bit integer
0	1	reserved
1	0	single-precision (32-bit) floating-point number
1	1	double-precision (64-bit) floating-point number

- The **coprocessor command** is divided into 5 fields:
  - The 5-bit **fpuop** field contains the opcode of a TMS34082 assembly-language instruction.
  - The 2-bit **md** field conveys the coprocessor command's addressing mode:

Mode	Operation
00_2	FPU internal operation, no jumps or external moves
01_2	Transfer to/from TMS34020 registers
10_2	Transfer to/from TMS34020 local memory
11_2	External microcode

  - **CRd** is the TMS34082 destination register.
  - **CRs_2** is the second TMS34082 source register for instructions that use two source operands. CRs_2 also serves as the count operand for instructions that use a count operand. CRs_2 must be a member of the TMS34082 B register file.
  - **CRs_1** is the TMS34082 source register for instructions that have one source operand; it is the first source register for instructions that use two source operands. CRs_1 must be a member of the TMS34082 A register file.
- The **ID** field serves the same purpose in the TMS34082 protocol as it serves in the general-purpose protocol. For more details, refer to subsection 10.3.1, Coprocessor ID, on page 10-5. The pseudo-ops default to an ID of 000_2; to define another ID as the current ID, use the .coproc assembler directive.

For the TMS34082, these bits are hard-coded into special versions of the TMS34020's general-purpose coprocessor instructions. As an example, Figure 14–2 compares the general syntax of the CMOVGC instruction (a general-purpose coprocessor instruction) to the LOAD-and-ADD (ADD) pseudo-op.

Figure 14–2. How General Coprocessor Instruction Syntax Corresponds to TMS34082 Pseudo-ops

(a) General syntax for CMOVGC instruction

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	*Rs1*			
8 LSBs of coprocessor command								size	0	0	R	*Rs2*			
coprocessor ID			13 MSBs of coprocessor command												

(b) General syntax for TMS34082 ADD pseudo-op

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	*Rs1*			
0	1	0	0	0	0	0	0	0	0	0	R	*Rs2*			
ID			*CRs1*				*CRs2*					*CRd*			

Note: This portion of the figure shows the ADD pseudo-op. ADD is a special form of the TMS34020 CMOVGC instruction. The coprocessor command and size portions of the CMOVGC instruction are hardcoded to specifically match the needs of the TMS34082's ADD instruction.

(c) What is output on the LAD bus for the ADD pseudo-op

LAD # 31—29	28 25	24 21	20 16	15 14	13 9	8	7	6	5	4	3 2 1 0
ID	*CRs1*	*CRs2*	*CRd*	01	00000	0	0	0	0	0	0 0 0 0

Note:

To identify the 3-bit ID, use the .coproc assembler directive (refer to the *TMS340 Family Code Generation Tools User's Guide* for information about the .coproc directive).

14.3 Register Operands

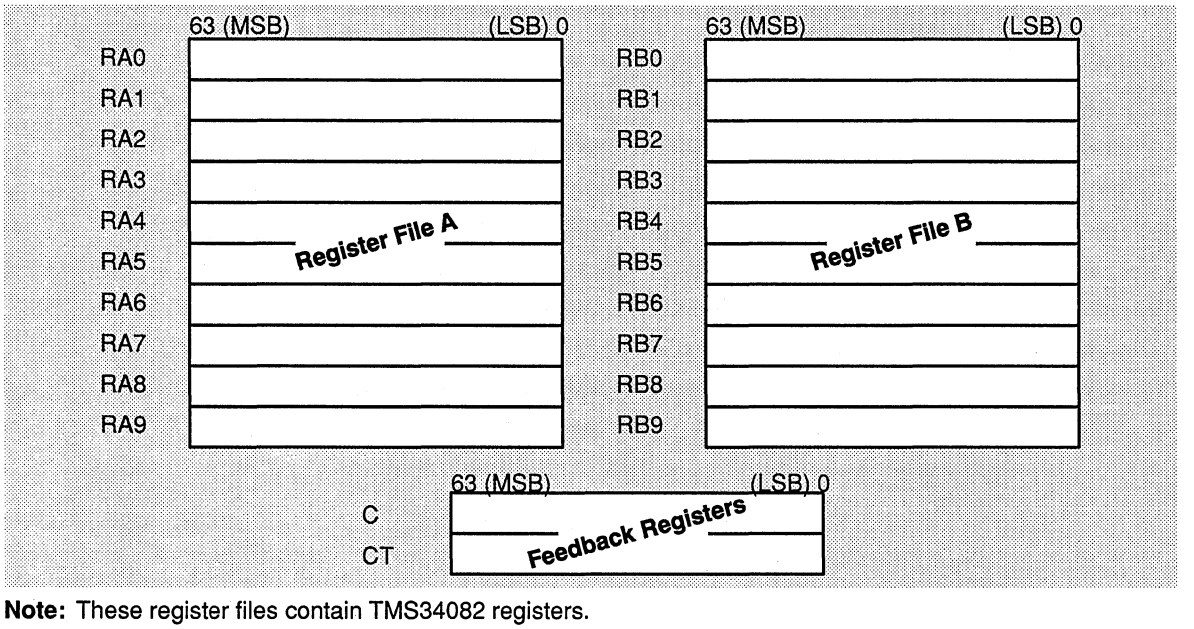
The TMS34082 pseudo-ops use register operands only. Table 14–1 lists the register-operand symbols used in the psuedo-op syntaxes in this chapter.

Table 14–1. Symbols Used in Pseudo-op Syntax Listings

Symbol	Description	Symbol	Description
CRs	TMS34082 source register	*CRd*	TMS34082 destination register
CRs_1	For pseudo-ops that use 2 TMS34082 source registers, this register supplies the first operand This operand must be a TMS34082 A-file register	CRs_2	For pseudo-ops that use 2 TMS34082 source registers, this register supplies the second operand This operand must be a TMS34082 B-file register
Rs	TMS34020 source register	*Rd*	TMS34020 destination register

Note that some pseudo-ops use information from a TMS34020 register or place information into a TMS34020 register. In this case, *Rs* or *Rd* should be a TMS34020 general-purpose register (A0—A14 or B0—B14), just as it would be for a TMS34020 instruction. Most of the pseudo-ops, however, use TMS34082 registers as operands. As Figure 14–3 shows, the TMS34082 contains 2 register banks of 10 64-bit registers, plus 2 feedback registers.

Figure 14–3. TMS34082 Registers That Can Be Used as Pseudo-op Operands



Note: These register files contain TMS34082 registers.

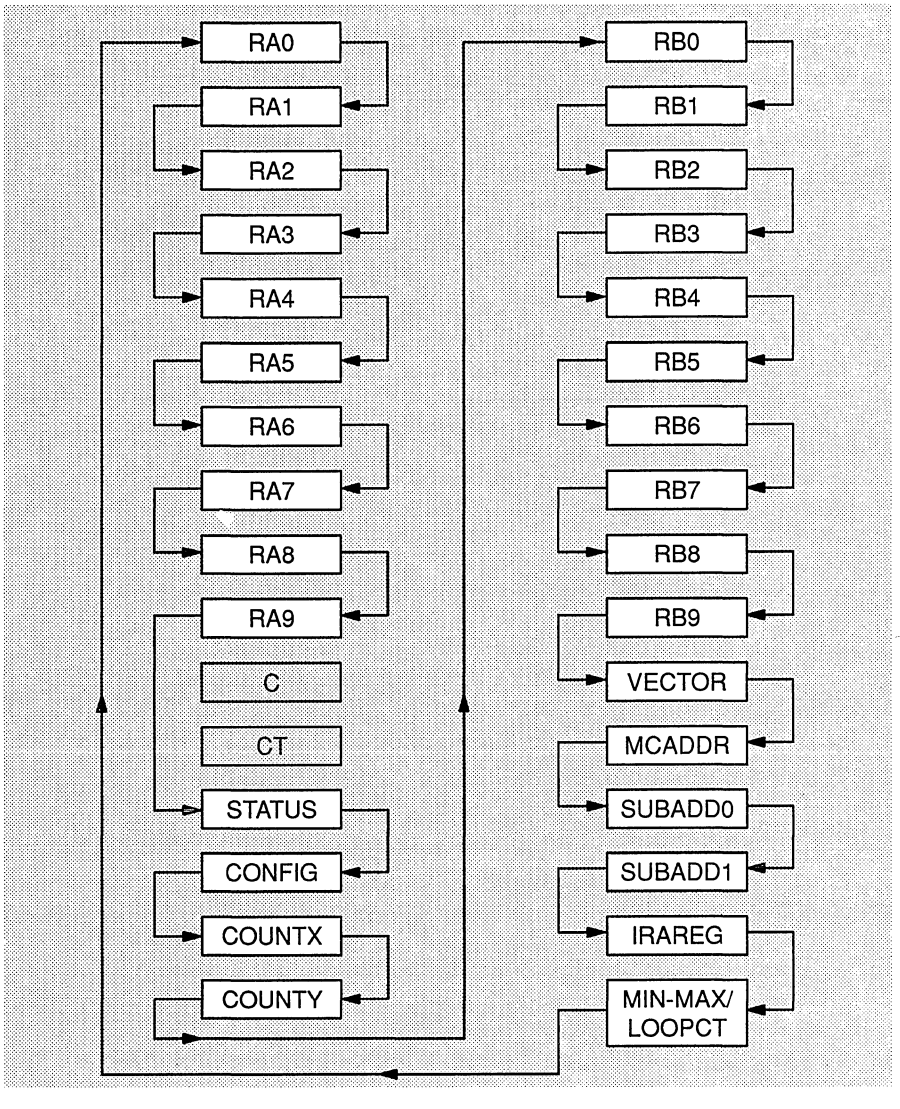
Most pseudo-ops operate on one value from TMS34082 register file A or B, and return the result to file A, file B, or one of the feedback registers. Valid operand/register use includes:

CRs or CRs_1: RA0—RA9 CRs_2: RB0—RB9

CRd: RA0—RA9, RB0—RB9, C, and CT

When more than one value is requested from/sent to the TMS34082, the registers are read from/written to in the sequence shown in Figure 14–4. Note that the control, status, and stack registers are in the middle of the list. The sequence bypasses C and CT because they can't be accessed externally.

Figure 14–4. TMS34082 Register Sequence List



Syntax **ABORT**

Execution Halts coprocessor

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
Default ID			0	0	0	0	1	0	1	1	0	0	0	0	0

Description ABORT halts the operation of the TMS34082 coprocessor and places the coprocessor in a *wait-for-next-instruction* state. Register values are indeterminate.

Machine States 2

Instruction Type CEXEC, short

Example

```
ABORT
```

This example halts the TMS34082 coprocessor.

Syntax

ABS *CRs, CRd*

Execution

|CRs| → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
Default ID			CRs				0	0	1	0	CRd				

Operands

CRs Coprocessor source register containing the 32-bit integer operand

CRd Coprocessor destination register

Description

ABS takes the absolute value of the contents (integer) of CRs and stores the result in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
ABS RA6, RB7
```

This example takes the absolute value of the contents of RA6 and stores the result in RB7.

Syntax

ABS *Rs, CRs, CRd*

Execution

Rs → CRs
|CRs| → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Default ID			CRs				0	0	1	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit integer value to coprocessor

CRs Coprocessor register to contain the 32-bit integer operand

CRd Coprocessor destination register

Description

ABS loads the contents (integer) of Rs into CRs, takes the absolute value of the contents of CRs, and stores the result in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

3 if the first instruction word is long-word aligned
2 if the first instruction word is not long-word aligned

Instruction Type

CMOVGC, one register

Example

```
ABS A5, RA6, RB7
```

This example loads the contents of TMS34020 register A5 into coprocessor register RA6, takes the absolute value of the contents of RA6, and stores the result in RB7.

Syntax **ABSD** *CRs, CRd*

Execution |CRs| → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
Default ID			CRs				0	0	1	0	CRd				

Operands

CRs Coprocessor source register containing a 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description ABSD takes the absolute value of the contents of CRs and stores the result in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States 2

Instruction Type CEXEC, short

Example

```
ABSD RA6,  RB7
```

This example takes the absolute value of the contents of RA6 and stores the result in RB7.

Syntax

ABSF *CRs, CRd*

Execution

|CRs| → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
Default ID			CRs				0	0	1	0	CRd				

Operands

CRs Coprocessor source register containing a 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

ABSF takes the absolute value of the contents (single-precision value) of CRs and stores the result in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
ABSF RA6, RB7
```

This example takes the absolute value of the contents of RA6 and stores the result in RB7.

Syntax

ABSF *Rs, CRs, CRd*

Execution

Rs → CRs
|CRs| → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			CRs				0	0	1	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to coprocessor

CRs Coprocessor register containing a 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

ABSF loads the contents (single-precision value) of Rs into CRs, takes the absolute value of the contents of CRs, and stores the result in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
ABSF A5, RA6, RB7
```

This example loads TMS34020 register A5 into coprocessor register RA6, takes the absolute value of the contents of RA6, and stores the result in RB7.

Syntax **ADD** *CRs_1, CRs_2, CRd*

Execution $CRs_1 + CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 32-bit integer operand

CRs_2 Coprocessor register containing the second 32-bit integer operand

CRd Coprocessor destination register

Description ADD adds the contents (integer) of CRs_1 and CRs_2 and stores the result in CRd.

Machine States 2

Instruction Type CEXEC, short

Example

```
ADD RA5, RB6, RB7
```

This example adds the contents of RA5 and RB6 and stores the result in RB7.

Syntax

ADD *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 + CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	0	0	0	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2					CRd			

Operands

Rs_1 TMS34020 source register for the first 32-bit integer value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit integer value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit integer operand

CRs_2 Coprocessor register to contain the second 32-bit integer operand

CRd Coprocessor destination register

Description

ADD loads the contents (integer) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, adds the contents of CRs_1 and CRs_2, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
ADD A5,  A6, RA5, RB6, RB7
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6 respectively, adds the contents of RA5 and RB6, and stores the result in RB7.

Syntax

ADDD *CRs_1, CRs_2, CRd*

Execution

$CRs_1 + CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	1	1
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 64-bit double-precision floating-point operand

CRs_2 Coprocessor register containing the second 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

ADDD adds the contents (double-precision value) of CRs_1 and CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
ADDD RA5, RB6, RA7
```

This example adds the contents of RA5 and RB6 and stores the result in RA7.

Syntax

ADDF *CRs_1, CRs_2, CRd*

Execution

$CRs_1 + CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register containing the second 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

ADDF adds the contents (single-precision value) of CRs_1 and CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
ADDF RA5, RB6, RB7
```

This example adds the contents of RA5 and RB6 and stores the result in RB7.

Syntax

ADDF *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 + CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	0	0	1	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first 32-bit single-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit single-precision floating-point value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register to contain the second 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

ADDF loads the contents (single-precision value) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, adds CRs_1 and CRs_2, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
ADDF A5,  A6, RA5, RB6, RA7
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6 respectively, adds the contents of RA5 and RB6, and stores the result in RA7.

Syntax

CHECK *Rd*

Execution

If coprocessor is busy
    FFFF FFFFh → Rd
If coprocessor is idle
    0000 0000h → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rs_1			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Default ID			0	0	0	0	1	1	0	1	0	0	0	0	0

Operands

Rd Destination register for status information

Description

CHECK checks the status of the coprocessor. If the TMS34082 coprocessor is busy, CHECK sets all the bits in Rd to 1. If the TMS34082 coprocessor is idle, CHECK sets all the bits in Rd to 0.

Machine States

5 if the first instruction word is long word-aligned
4 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
CHECK A4
```

If the TMS34082 coprocessor is busy, this example sets all the bits in register A4 to 1. If the TMS34082 coprocessor is idle, this example resets all the bits in register A4 to 0.

Syntax

CMP *CRs_1, CRs_2*

Execution

Flags (CRs_1 – CRs_2) → Coprocessor Status Registers

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0
Default ID			CRs_1				CRs_2				0	0	0	0	0

Operands

CRs_1 Coprocessor register containing the first 32-bit integer operand

CRs_2 Coprocessor register containing the second 32-bit integer operand

Description

CMP subtracts the contents (integer) of CRs_2 from CRs_1 and sets the appropriate status bits in the coprocessor status register.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CMP RA5, RB6
```

This example subtracts the contents of RA5 from RB6 and sets the status bits in the coprocessor status register.

Syntax

CMP *Rs_1, Rs_2, CRs_1, CRs_2*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
Flags ($CRs_1 - CRs_2$) → Coprocessor Status Registers

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	1	0	0	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				0	0	0	0	0

Operands

Rs_1 TMS34020 source register for the first 32-bit integer value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit integer value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit integer operand

CRs_2 Coprocessor register to contain the second 32-bit integer operand

Description

CMP loads the contents (integer) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts CRs_2 from CRs_1, and sets the appropriate status bits in the coprocessor status register.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
CMP A5, A6, RA5, RB6
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6, subtracts the contents of RB6 from RA5, and sets the status bits in the coprocessor status register

Syntax

CMPD *CRs_1, CRs_2*

Execution

Flags (CRs_1 – CRs_2) → Coprocessor Status Registers

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	0	1	1
Default ID			CRs_1				CRs_2				0	0	0	0	0

Operands

CRs_1 Coprocessor register containing the first 64-bit double-precision floating-point operand

CRs_2 Coprocessor register containing the second 64-bit double-precision floating-point operand

Description

CMPD subtracts the contents (double-precision value) of CRs_2 from CRs_1 and sets the appropriate status bits in the coprocessor status register.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CMPD RA5, RB6
```

This example subtracts the contents of RB6 from RA5 and sets the status bits in the coprocessor status register.

Syntax

CMPF *CRs_1, CRs_2*

Execution

Flags (CRs_1 – CRs_2) → Coprocessor Status Register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0
Default ID			CRs_1				CRs_2				0	0	0	0	0

Operands

CRs_1 Coprocessor register containing the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register containing the second 32-bit single-precision floating-point operand

Description

CMPF subtracts the contents (single-precision value) of CRs_2 from CRs_1 and sets the appropriate status bits in the coprocessor status register.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CMPF RA5, RB6
```

This example subtracts the contents of RB6 from RA5 and sets the status bits in the coprocessor status register.

Syntax

CMPF *Rs_1, Rs_2, CRs_1, CRs_2*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
Flags ($CRs_1 - CRs_2$) → Coprocessor Status Register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	1	0	1	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				0	0	0	0	0

Operands

Rs_1 TMS34020 source register for first the 32-bit single-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit single-precision floating-point value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register to contain the second 32-bit single-precision floating-point operand

Description

CMPF loads the contents (single-precision value) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts CRs_2 from CRs_1, and sets the appropriate status bits in the coprocessor status register.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
CMPF A5, A6, RA5, RB6
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6 respectively, subtracts the contents of RB6 from the contents of RA5, and sets the status bits in the coprocessor status register.

Syntax

CVDF *CRs, CRd*

Execution

(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
Default ID			CRs				0	1	0	0	CRd				

Operands

CRs Coprocessor source register containing a 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

CVDF converts a 64-bit IEEE double-precision floating-point number to a 32-bit IEEE single-precision floating-point number. The double-precision number resides in CRs, and the converted single-precision number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CVDF RA5, RA7
```

This example converts the contents of RA5 to a single-precision floating-point number and stores the result in RA7.

Syntax

CVDI *CRs, CRd*

Execution

(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
Default ID			CRs				0	1	0	1	CRd				

Operands

CRs Coprocessor source register containing a 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

CVDI converts a 64-bit IEEE double-precision floating-point number to a 32-bit integer number. The double-precision number resides in CRs, and the converted integer number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CVDI RA5, RB7
```

This example converts the contents of RA5 to an integer and stores the result in RB7.

Syntax

CVFD *CRs, CRd*

Execution

(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
Default ID			CRs				0	1	0	0	CRd				

Operands

CRs Coprocessor source register containing a 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

CVFD converts a 32-bit IEEE single-precision floating-point value to a 64-bit IEEE double-precision floating-point value. The single-precision number resides in CRs, and the converted double-precision number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CVFD RA5, RB7
```

This example converts the contents of RA5 to a double-precision number and stores the result in RB7.

Syntax

CVFD *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			CRs				0	1	0	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to coprocessor

CRs Coprocessor register to contain the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

CVFD loads the contents (single-precision) of Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 64-bit IEEE double-precision floating-point value. The single-precision number resides in CRs, and the converted double-precision number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
CVFD B5, RA5, RA7
```

This example loads TMS34020 register B5 into coprocessor register RA5, converts the contents of RA5 to a double-precision number, and stores the result in RA7.

Syntax

CVFI *CRs, CRd*

Execution

(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
Default ID			CRs				0	1	0	1	CRd				

Operands

CRs Coprocessor source register containing a 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

CVFI converts a 32-bit IEEE single-precision floating-point value to a 32-bit integer value. The single-precision number resides in CRs, and the converted integer number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CVFI RA5, RA7
```

This example converts the contents of RA5 to an integer and stores the result in RA7.

Syntax

CVFI *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			CRs				0	1	0	1	CRd				

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to coprocessor

CRs Coprocessor register to contain the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

CVFI loads the contents (single-precision) of Rs into CRs and converts the 32-bit IEEE single-precision floating-point value to a 32-bit integer value. The single-precision number resides in CRs, and the converted integer number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
CVFI B5, RA5, RB7
```

This example loads TMS34020 register B5 into coprocessor register RA5, converts the contents of RA5 to an integer, and stores the result in RB7.

Syntax

CVID *CRs, CRd*

Execution

(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
Default ID			CRs				0	1	1	0	CRd				

Operands

CRs Coprocessor source register containing the 32-bit integer operand

CRd Coprocessor destination register

Description

CVID converts a 32-bit integer value to a 64-bit IEEE double-precision floating-point value. The integer resides in CRs, and the converted double-precision number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CVID RA5, RB7
```

This example converts the contents of RA5 to a double-precision number and stores the result in RB7.

Syntax

CVID *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs			
0	1	0	1	1	1	1	1	1	0	0	R	Rs			
Default ID			CRs				0	1	1	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit integer values to coprocessor

CRs Coprocessor source register to contain the 32-bit integer operand

CRd Coprocessor destination register

Description

CVID loads the contents (integer) of Rs into CRs and converts a 32-bit integer value to a 64-bit IEEE double-precision floating-point value. The integer resides in CRs, and the converted double-precision number resides in CRd. (Constraints of the TMS34082 require that the integer in Rs be sent as both words of the 64-bit transfer.)

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

4 if first instruction word is long word-aligned
3 if first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
CVID B5, RA5, RA7
```

This example loads TMS3420 register B5 into coprocessor register RA5, converts the contents of RA5 to a double-precision number, and stores the result in RA7.

Syntax

CVIF *CRs, CRd*

Execution

(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
Default ID			CRs				0	1	1	0	CRd				

Operands

CRs Coprocessor source register containing the 32-bit integer operand

CRd Coprocessor destination register

Description

CVIF converts a 32-bit integer value to a 32-bit IEEE single-precision floating-point value. The integer resides in CRs, and the converted single-precision number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
CVIF RA5, RA7
```

This example converts the contents of RA5 to a single-precision number and stores the result in RA7.

Syntax

CVIF *Rs, CRs, CRd*

Execution

Rs → CRs
(CRs) → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			CRs				0	1	1	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit integer value to coprocessor

CRs Coprocessor source register to contain the 32-bit integer operand

CRd Coprocessor destination register

Description

CVIF loads the contents (integer) of Rs into CRs and converts a 32-bit integer value to a 32-bit IEEE single-precision floating-point value. The integer resides in CRs, and the converted single-precision number resides in CRd.

The coprocessor source register, CRs, must be in the A coprocessor file.

Machine States

3 if first instruction word is long word-aligned
2 if first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
CVIF RA5, RA7
```

This example converts the contents of RA5 to a single-precision number and stores the result in RA7.

Syntax

DIVD *CRs_1, CRs_2, CRd*

Execution

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	0	1	1	1
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 64-bit double-precision floating-point operand

CRs_2 Coprocessor register containing the second 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

DIVD divides the contents (double-precision value) of CRs_1 by CRs_2 and stores the result CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
DIVD RA5, RB6, RA7
```

This example divides the contents of RA5 by RB6 and stores the result in RA7.

Syntax

DIVF *CRs_1, CRs_2, CRd*

Execution

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	0	1	1	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register containing the second 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

DIVF divides the contents (single-precision value) of CRs_1 by CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
DIVF RA5, RB6, RA7
```

This example divides the contents of RA5 by RB6 and stores the result in RA7.

Syntax

DIVF *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$

$Rs_2 \rightarrow CRs_2$

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	0	1	1	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first 32-bit floating-point single-precision value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit floating-point single-precision value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register to contain the second 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

DIVF loads the contents (single precision, floating point) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, divides the contents of CRs_1 by CRs_2, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
DIVF A5, A6, RA5, RB6, RA7
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6 respectively, divides the contents of RA5 by RB6, and stores the result in RA7.

Syntax

DIVS *CRs_1, CRs_2, CRd*

Execution

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	0	1	0	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 32-bit integer operand

CRs_2 Coprocessor register containing the second 32-bit integer operand

CRd Coprocessor destination register

Description

DIVS divides the contents (integer) of CRs_1 by CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
DIVS RA5, RB6, RB7
```

This example divides the contents of RA5 by RB6 and stores the result in RB7.

Syntax

DIVS *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$

$Rs_2 \rightarrow CRs_2$

$$\left(\frac{CRs_1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	0	1	0	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first 32-bit integer value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit integer value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit integer operand

CRs_2 Coprocessor register to contain the second 32-bit integer operand

CRd Coprocessor destination register

Description

DIVS loads the contents (integer) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, divides the contents of CRs_1 by CRs_2, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
DIVS A5, A6, RA5, RB6, RB7
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6 respectively, divides the contents of RA5 by RB6, and stores the result in RB7.

Syntax

GETCST

Execution

Coprocessor Status Register → ST

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0
0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1
Default ID			0	0	0	0	0	0	0	0	0	1	1	0	0

Description

GETCST loads 4 MSBs of the coprocessor status register (STATUS) into the TMS34020 status register (ST).

Machine States

5 if the first instruction word is long word-aligned
4 if the first instruction word is not long word-aligned

Instruction Type

CMOVCS

Example

```
GETCST
```

This example sends the coprocessor status register to the TMS34020. The TMS34020 takes the value and masks off the 4 MSBs; it then stuffs the values in the TMS34020 status register corresponding to the N, C, Z, V bits.

Syntax

INVD *CRs_2, CRd*

Execution

$$\left(\frac{1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	1	0	1	1
Default ID			0	0	0	0	CRs_2				CRd				

Operands

CRs_2 Coprocessor register-B file containing the 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

INVD divides 1.0 by the contents (double precision) of CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
INVD RB3, RA1
```

This example divides 1.0 by RB3 and stores the result in RA1.

Syntax

INVF *CRs_2, CRd*

Execution

$$\left(\frac{1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	1	0	1	0
Default ID			0	0	0	0	CRs_2				CRd				

Operands

CRs_2 Coprocessor register-B file containing the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

INVF divides 1.0 by the contents (single precision, floating point) of CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
INVF RB3, RA1
```

This example divides 1.0 by RB3 and stores the result in RA1.

Syntax

INVF *Rs, CRs_2, CRd*

Execution

$Rs \rightarrow CRs_2$

$$\left(\frac{1}{CRs_2}\right) \rightarrow CRd$$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0
Default ID			0	0	0	0	CRs_2				CRd				

Operands

Rs TMS34020 source register for the 32-bit floating-point single-precision value to coprocessor

CRs_2 Coprocessor register-B file to contain the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

INVF loads the contents (single precision, floating point) of Rs into CRs_2, divides 1.0 by CRs_2, and stores the result in CRd.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
INVF A7, RB3, RA1
```

This example loads TMS34020 register A7 into coprocessor register RB3, divides 1.0 by RB3, and stores the result in RA1.

Syntax

JUMPC *n*

Execution

Execute external coprocessor instructions found at address 2 × *n*

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
1	1	*n*					0	0	0	0	0	0	0	0	0
Default ID			0	0	0	0	0	0	0	0	0	0	0	0	0

Operands

n Specifies the address to which the TMS34082 instruction execution is sent

Description

JUMPC begins execution of TMS34082 external instructions stored in TMS34082 local memory. The starting address is specified as TMS34082 local memory address 2 × *n*. Usually, a jump table is stored in these locations to permit complex operations.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CEXEC, long

Example

```
JUMPC 4
```

This example executes TMS34082 instructions stored in the default TMS34082's local memory. The executed instructions are stored beginning in address 8.

Syntax

MOVD *Rs_1, Rs_2, CRd*

Execution

Rs_1, Rs_2 → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	1	1	0	1	1	0	0	R	Rs_2			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs_1 TMS34020 source register for the 32 MSBs (sign, exponent, and 20 MSBs of mantissa) of the 64-bit double-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for the 32 LSBs of the 64-bit double-precision floating-point value to coprocessor

CRd Coprocessor destination register that holds the 64-bit double-precision floating-point value

Description

MOVD moves the double-precision value in Rs_1 and Rs_2 into CRd. Rs_1 holds the 32 MSBs, and Rs_2 holds the 32 LSBs of the double. You must set the LOAD bit of the TMS34082 configuration register to 0 to indicate that the MSBs are transferred before the LSBs.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

This example uses the MOVD instruction to load a 64-bit double-precision value into register RB5. Note that the 32 MSBs of the value are loaded into a1 and then the 32 LSBs are loaded into A0. Assume that the LOAD bit of the configuration register is set to 0, indicating transfers of MSBs before LSBs.

```
00000000                        .ieeefl
00000000      0540              setf   32,0,0
00000010      05a0              move   @dval,a0,0
00000020  00000000"
00000040      05a1              move   @dval+32,a1,0
00000050  00000020"
00000070      0641              movd   a1,a0,rb5
00000080      5f80
00000090      1f95
00000000                        .data
00000000  8a6a51ad  dval:       .double 347.6942238
00000020  4075bb1b
```

Syntax

MOVD **Rs+, CRd, Rd*

Execution

If TMS34082 **LOAD bit = 0** **and Rd = 0** Repeat 16 times *Rs → CRd (32 MSBs) Rs + 32 → Rs *Rs → CRd (32 LSBs) Rs + 32 → Rs advance to next coprocessor register	If TMS34082 **LOAD bit = 1** **and Rd = 1 → 31** Repeat Rd/2 times *Rs → CRd (32 MSBs) Rs + 32 → Rs *Rs → CRd (32 LSBs) Rs + 32 → Rs advance to next coprocessor register
If TMS34082 **LOAD bit = 1** **and Rd = 0** Repeat 16 times *Rs → CRd (32 LSBs) Rs + 32 → Rs *Rs → CRd (32 MSBs) Rs + 32 → Rs advance to next coprocessor register	If TMS34082 **LOAD bit = 1** **and Rd = 1 → 31** Repeat Rd/2 times *Rs → CRd (32 LSBs) Rs + 32 → Rs *Rs → CRd (32 MSBs) Rs + 32 → Rs advance to next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	0	1	1	0	1	1	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register (indirect postincrement) containing the address of the first 32-bits of the first double-precision value to move to the coprocessor

CRd Coprocessor destination register to hold the first 64-bit double-precision floating-point value

Rd TMS34020 register containing the number of 32-bit transfers to make. This value must in the range 0 to 31.

- If *Rd* = 0, then *32* 32-bit transfers are made
- If *Rd* = 1 → 31 then *Rd* 32-bit transfers are made

Note that because 64-bit doubles require two 32-bit moves, an odd number in Rd will give unpredictable results.

Description

MOVD moves 64-bit double-precision values from memory beginning at the address in Rs into coprocessor registers beginning at CRd. After each transfer, the contents of Rs are incremented; after every two 32-bit transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. The number of 32-bit transfers made is determined by the contents of Rd. The results will be unpredictable if Rd is an odd number.

The TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

- If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)
- If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Machine States

If Rd = 0 and	Rs is aligned	36
If Rd = 0 and	Rs is nonaligned	37
If Rd = 1 → 31 and	Rs is aligned	5 + (Rd – 1)
If Rd = 1 → 31 and	Rs is nonaligned	6 + (Rd – 1)

Instruction Type

CMOVMC, postincrement, register count

Example

```
MOVD *A5+, RB7, B7
```

This example moves 64-bit double-precision values from the TMS34020 memory location pointed to by A5 to coprocessor registers beginning with RB7. After each 32-bit transfer, register A5 is incremented; after every two 32-bit transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. B7 holds the number of 32-bit transfers to be made.

Syntax

MOVD **Rs+, CRd, [, count]*

Execution

If TMS34082 **LOAD bit = 0**
Repeat *count* times
*Rs → CRd (32 MSBs)
Rs + 32 → Rs
*Rs → CRd (32 LSBs)
Rs + 32 → Rs
advance to next coprocessor register

If TMS34082 **LOAD bit = 1**
Repeat *count* times
Repeat *count* times
*Rs → CRd (32 LSBs)
Rs + 32 → Rs
*Rs → CRd (32 MSBs)
Rs + 32 → Rs
advance to next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	transfers				
1	0	0	0	1	1	0	1	1	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs — TMS34020 source register (indirect postincrement) containing the address of the first 32-bits of the first double-precision value to move to the coprocessor

CRd — Coprocessor destination register that holds the first 64-bit double-precision floating-point value

count — Contains the number of 64-bit transfers to make. This value must in the range 1 to 16; the default value is 1. *Count* determines the value of transfers:

- If *count* = 16, then transfers = 0
- If *count* = 1 → 15, then transfers = 2 × *count*

Description

MOVD moves 64-bit double-precision values from memory beginning at the address in Rs into coprocessor registers beginning at CRd. After each transfer, the contents of Rs are incremented; after every two 32-bit transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. The number of 64-bit transfers made is determined by the contents of *count*.

The TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

- If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)
- If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Machine States

Rs Aligned	5 + ((count × 2) − 1)
Rs Nonaligned	6 + ((count × 2) − 1)

Instruction Type CMOVMC, postincrement, constant count

Example

```
MOVD *A5+, RB7, 4
```

This example moves four 64-bit double-precision values from the TMS34020 memory location pointed to by A5 to coprocessor registers beginning with RB7. After each 32-bit transfer, register A5 is incremented; after every two 32-bit transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 64-bit transfers are made (eight 32-bit transfers).

Syntax

MOVD *–*Rs, CRd* [, *count*]

Execution

If TMS34082 **LOAD bit = 0**
Repeat *count* times
Rs – 32 → Rs
*Rs → CRd (32 MSBs)
Rs – 32 → Rs
*Rs → CRd (32 LSBs)
advance to next coprocessor register

If TMS34082 **LOAD bit = 1**
Repeat *count* times
Rs – 32 → Rs
*Rs → CRd (32 LSBs)
Rs – 32 → Rs
*Rs → CRd (32 MSBs)
advance to next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	transfers				
1	0	0	0	1	1	0	1	1	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register (indirect predecrement) containing the address of the bit immediately following the 64-bits used to store the first 64-bit double-precision floating-point value that is transferred

CRd Coprocessor destination register that holds the first 64-bit double-precision floating-point value

count Contains the number of 64-bit transfers to make. This value must in the range 1 to 16; the default value is 1. *Count* determines the value of transfers:

- If *count* = 16, then transfers = 0
- If *count* = 1 → 15, then transfers = 2 × *count*

Description

MOVD moves 64-bit double-precision values from memory beginning at the address (Rs –32) into coprocessor registers beginning at CRd. Before each 32-bit transfer, the contents of Rs are decremented; after every two 32-bit transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. The number of 64-bit transfers made is determined by the contents of *count*.

The TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

- If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)
- If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Machine States

Rs Aligned	5 + ((count × 2) – 1)
Rs Nonaligned	6 + ((count × 2) – 1)

Instruction Type CMOVMC, predecrement, constant count

Example

```
MOVD —*A5, RB7, 4
```

This example moves four 64-bit double-precision values from the TMS34020 memory location pointed to by (A5 – 32) to coprocessor registers beginning with RB7. Before each 32-bit transfer, register A5 is decremented; after every two 32-bit transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 64-bit transfers are made (eight 32-bit transfers).

Syntax

MOVD *CRd, *Rd+* [, *count*]

Execution

If TMS34082 **LOAD bit = 0**
Repeat *count* times
CRd (32 MSBs) →*Rd
Rd + 32 → Rd
CRd (32 LSBs) → *Rd
Rd + 32 → Rd
advance to next coprocessor register

If TMS34082 **LOAD bit = 1**
Repeat *count* times
CRd (32 LSBs) → *Rd
Rd + 32 → Rd
CRd (32 MSBs) → *Rd
Rd + 32 → Rd
advance to next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	1	R	Rd			
1	0	0	0	1	1	1	1	1	0	0	transfers				
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register for the first 64-bit double-precision floating-point value to the TMS34020 memory

Rd TMS34020 register (indirect postincrement) containing the address of the first double-precision value transferred

count Contains the number of 64-bit transfers to make. This value must in the range 1 to 16; the default value is 1. *Count* determines the value of transfers:

- If *count* = 16, then transfers = 0
- If *count* = 1 → 15, then transfers = 2 × *count*

Description

MOVD moves the 64-bit double-precision values from coprocessor registers beginning at CRd to memory beginning at the address in Rd. After each 32-bit transfer, Rd is incremented, and after every two transfers, the coprocessor register is advanced to the next register in the coprocessor register sequence. The number of 64-bit transfers made is determined by the contents of *count*.

The TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

- If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)
- If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Machine States

Rd aligned	5 + (count*2 – 1)
Rd nonaligned	6 + (count*2 – 1)

Instruction Type

CMOVCM, postincrement, constant count

Example

```
MOVD RB7, *A5+, 2
```

This example moves four 64-bit double-precision values from coprocessor registers beginning at RB7 to TMS34020 memory pointed to by A5. After each 32-bit transfer, register A5 is incremented, and after every two transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that two 64-bit transfers are made (four 32-bit transfers).

Syntax

MOVD *CRd, –*Rd* [, *count*]

Execution

If TMS34082 **LOAD bit = 0**
Repeat *count* times
Rd – 32 → Rd
CRd (32 MSBs) →*Rd
Rd – 32 → Rd
CRd (32 LSBs) → *Rd
advance to next coprocessor register

If TMS34082 **LOAD bit = 1**
Repeat *count* times
Rd – 32 → Rd
CRd (32 LSBs) → *Rd
Rd – 32 → Rd
CRd (32 MSBs) → *Rd
advance to next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	0	R	Rd			
1	0	0	0	1	1	1	1	1	0	0	transfers				
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register for the first double-precision value to TMS34020 memory

Rd TMS34020 register (indirect predecrement) containing the address of the bit immediately following the 64-bits used to store the first 64-bit double-precision floating-point value that is transferred

count Contains the number of 64-bit transfers to make. This value must in the range 1 to 16; the default value is 1. *Count* determines the value of transfers:

- If *count* = 16, then transfers = 0
- If *count* = 1 → 15, then transfers = 2 × *count*

Description

MOVD moves the 64-bit double-precision values from coprocessor registers beginning at CRd to memory beginning at the address (Rd – 32). Before each 32-bit transfer, Rd is decremented; after every two 32-bit transfers, the coprocessor register is advanced to the next register in the coprocessor register sequence. The number of 64-bit transfers made is determined by the contents of *count*.

The TMS34082 configuration register LOAD bit determines whether the LSBs or the MSBs will be moved first:

- If the LOAD bit = 1, then the LSBs are moved first (32 LSBs of the fraction)
- If the LOAD bit = 0, then the MSBs are moved first (sign, exponent, and 20 MSBs of the fraction)

The LOAD bit default is 0.

Machine States

Rd aligned	5 + (count*2 – 1)
Rd nonaligned	6 + (count*2 – 1)

Instruction Type

CMOVCM, predecrement, constant count

Example

```
MOVD RB7, –*A5, 2
```

This example moves two 64-bit double-precision values from coprocessor registers beginning at RB7 to TMS34020 memory pointed to by (A5 – 32). Before each 32-bit transfer, register A5 is decremented; after every two 32-bit transfers, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that two 64-bit transfers are made (four 32-bit transfers).

Syntax

MOVD *CRs_1, CRd*

Execution

$CRs_1 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	0	1	1	1
Default ID			CRs_1				0	0	0	1	CRd				

Operands

CRs_1 Coprocessor source register A that holds the 64-bit double-precision floating-point value

CRd Coprocessor destination register

Description

MOVD moves a 64-bit double-precision value from CRs_1 (register A) to CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MOVD RA7, RB4
```

This example moves the 64-bit double-precision value from coprocessor register RA7 to coprocessor register RB4.

Syntax

MOVD *CRs_2, CRd*

Execution

$CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	0	1	1
Default ID			CRs_2				0	0	0	1	CRd				

Operands

CRs_2 Coprocessor source register B that holds the 64-bit double-precision floating-point value

CRd Coprocessor destination register

Description

MOVD moves a 64-bit double-precision floating-point value from CRs_2 (register B) to CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MOVD RB3, RB4
```

This example moves the 64-bit double-precision value from coprocessor register RB3 to coprocessor register RB4.

Syntax

MOVE *Rs, CRd*

Execution

Rs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register for 32-bit integer value to coprocessor

CRd Coprocessor destination register to hold the 32-bit integer

Description

MOVE moves the contents (integer) of Rs into CRd.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
MOVE A5, RA7
```

This example moves the contents of TMS34020 register A5 into coprocessor register RA7.

Syntax

MOVE *Rs_1, Rs_2, CRd*

Execution

$Rs_1 \rightarrow$ CRd
$Rs_2 \rightarrow$ CRd+1

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	1	1	0	0	0	0	0	R	Rs_2			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs_1 TMS34020 source register for the first 32-bit integer value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit integer value to coprocessor

CRd Coprocessor destination register that holds the first 32-bit integer value. The second integer will be placed in the next register in the coprocessor register sequence list.

Description

MOVE moves the contents (integer) of Rs_1 and Rs_2 into CRd and CRd +1.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
MOVE A5, A6, RA7
```

This instruction moves the contents of TMS34020 registers A5 and A6 into coprocessor register RA7 and RA8, respectively.

Syntax

MOVE **Rs +, CRd, Rd*

Execution

If **RD = 0**
Repeat 32 times
*Rs → CRd
Rs + 32 → Rs
advance to next coprocessor register

If **Rd = 1 → 31**
Repeat *Rd* times
*Rs → CRd
Rs + 32 → Rs
advance to next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	0	1	1	0	0	0	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register (indirect postincrement) containing the address of the first 32-bit integer to move to the coprocessor

CRd Coprocessor destination register to hold the first 32-bit integer operand

Rd TMS34020 register containing the number of 32-bit transfers to make. This value must in the range 0 to 31

- If *Rd* = 0, then *32* 32-bit transfers are made
- If *Rd* = 1 → 31, then *Rd* 32-bit transfers are made

Description

MOVE moves integer values from memory beginning at the address in Rs into coprocessor registers beginning at CRd. After each transfer, Rs is incremented, and CRd is advanced to the next register in the coprocessor register sequence list. The number of 32-bit transfers made is determined by the contents of *Rd*.

Machine States

If Rd = 0 and	Rs is aligned	36
If Rd = 0 and	Rs is nonaligned	37
If Rd = 1 → 31 and	Rs is aligned	5 + (Rd – 1)
If Rd = 1 → 31 and	Rs is nonaligned	6 + (Rd – 1)

Instruction Type

CMOVMC, postincrement, register count

Example

```
MOVE *A5+, RA7, B7
```

This instruction moves integer values from TMS34020 memory location pointed to by A5 to coprocessor registers beginning at RA7. After each 32-bit transfer, register A5 is incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. B7 holds the number of 32-bit transfers to be made.

Syntax

MOVE **Rs+, CRd, [, count]*

Execution

Repeat *count* times
*Rs → CRd
Rs + 32 → Rs
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	transfers				
1	0	0	0	1	1	0	0	0	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs — TMS34020 source register (indirect postincrement) containing the address of the first 32-bit integer to move to the coprocessor

CRd — Coprocessor destination register to hold the first 32-bit integer operand

count — Contains the number of 32-bit transfers to make. This value must be in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVE moves 32-bit integer values from memory beginning at the address in Rs into coprocessor registers beginning at CRd. After each transfer, Rs is incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVMC, postincrement, constant count

Example

```
MOVE *A5+, RB7, 4
```

This example moves four 32-bit integer values from TMS34020 memory location pointed to by A5 to coprocessor registers beginning at RB7. After each 32-bit transfer, register A5 is incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVE *–*Rs, CRd* [*, count*]

Execution

Repeat *count* times
Rs – 32 → Rs
*Rs → CRd
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	transfers				
1	0	0	0	1	1	0	0	0	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register (indirect postincrement) containing the address of the bit immediately after first 32-bit integer to move to the coprocessor

CRd Coprocessor destination register to hold the first 32-bit integer operand

count Contains the number of 32-bit transfers to make. This value must in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVE moves 32-bit integer values from memory beginning at the address in (Rs – 32) into coprocessor registers beginning at CRd. Before each transfer, the contents of Rs are decremented; after each transfer, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVMC, predecrement, constant count

Example

```
MOVE –*A5, RB7, 4
```

This example moves four 32-bit integer values from TMS34020 memory location pointed to by (A5 – 32) to coprocessor registers beginning at RB7. Before each 32-bit transfer, register A5 is decremented; after each transfer, coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVE *CRd, *Rd+* [, *count*]

Execution

Repeat *count* times
CRs → *Rd
Rd + 32 → Rd
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	1	R	Rd			
1	0	0	0	1	1	1	0	0	0	0	transfers				
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register for the first 32-bit integer value to TMS34020 memory

Rd TMS34020 register (indirect postincrement) containing the address for the first integer transferred

count Contains the number of 32-bit transfers to make. This value must in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVE moves the 32-bit integer values from coprocessor registers beginning at CRd to memory beginning at the address in Rd. After each 32-bit transfer, Rd is incremented, and the coprocessor register is advanced to the next register in the coprocessor register sequence. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVCM, postincrement, constant count

Example

```
MOVE RB7, *A5+, 4
```

This example moves four 32-bit integer values from coprocessor registers beginning at RB7 to TMS34020 memory pointed to by A5. After each 32-bit transfer, register A5 is incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVE *CRd, –*Rd* [, *count*]

Execution

Repeat *count* times
Rd – 32 → Rd
CRd → *Rd
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	0	1	1	1	0	0	0	0	transfers				
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register for the first 32-bit integer value to TMS34020 memory

Rd TMS34020 register (indirect predecrement) containing the address of the bit immediately following the 32-bits used to store the first 32-bit integer value transferred

count Contains the number of 32-bit transfers to make. This value must in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVE moves the 32-bit integer values from coprocessor registers beginning at CRd to memory beginning at the address (Rd–32). Before each 32-bit transfer, Rd is decremented; after each 32-bit transfer, the coprocessor register is advanced to the next register in the coprocessor register sequence. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVCM, predecrement, constant count

Example

```
MOVE RB7, –*A5, 4
```

This example moves four 32-bit integer values from coprocessor registers beginning at RB7 to TMS34020 memory pointed to by (A5 – 32). Before each 32-bit transfer, register A5 is decremented; after each 32-bit transfer, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVE *CRd, Rd*

Execution

CRd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rd			
0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register holding the 32-bit integer value

Rd TMS34020 destination register

Description

MOVE moves 32-bit integer from coprocessor register CRd to TMS34020 register Rd.

Machine States

5 if the first instruction word is long word-aligned
4 if the first instruction word is not long word-aligned

Instruction Type

CMOVCG, one register

Example

```
MOVE RA7, A5
```

This example moves the contents of coprocessor register RA7 to TMS34020 register A5.

Syntax

MOVE *CRs_1, CRd*

Execution

$CRs_1 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	0	1	0	0
Default ID			CRs_1				0	0	0	1	CRd				

Operands

CRs_1 Coprocessor source register A that holds the 32-bit integer value

CRd Coprocessor destination register

Description

MOVE moves 32-bit integer value from CRs_1 (register A) to CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MOVE RA7, RB4
```

This example moves the 32-bit integer value from coprocessor register RA7 to coprocessor register RB4.

Syntax

MOVE *CRs_2, CRd*

Execution

$CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	0	0	0
Default ID			CRs_2				0	0	0	1	CRd				

Operands

CRs_2 Coprocessor source register B that holds the 32-bit integer value

CRd Coprocessor destination register

Description

MOVE moves a 32-bit integer value from CRs_2 (register B) to CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MOVE RB3, RB4
```

This example moves the 32-bit integer value from coprocessor register RB3 to coprocessor register RB4.

Syntax

MOVF *Rs, CRd*

Execution

Rs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to coprocessor

CRd Coprocessor destination register to hold the 32-bit single-precision floating-point value

Description

MOVF moves the contents (single-precision value) of Rs into CRd.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
MOVF A5, RA7
```

This example moves the contents of TMS34020 register A5 into coprocessor register RA7.

Syntax

MOVF *Rs_1, Rs_2, CRd*

Execution

$Rs_1 \rightarrow$ CRd
$Rs_2 \rightarrow$ CRd+1

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	1	1	0	1	0	0	0	R	Rs_2			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs_1 TMS34020 source register for the first 32-bit single-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit single-precision floating-point value to coprocessor

CRd Coprocessor destination register to hold the first single-precision value. The second single-precision value will be placed in the next register in the coprocessor register sequence list.

Description

MOVF moves the contents (single-precision value) of Rs_1 and Rs_2 into CRd and CRd+1.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
MOVD A5, A6, RB7
```

This example moves the contents of TMS34020 registers A5 and A6 into coprocessor registers RB7 and RB8.

Syntax

MOVF **Rs+, CRd, Rd*

Execution

If **Rd = 0**
Repeat 32 times
*Rs → CRd
Rs + 32 → Rs
advance to next coprocessor register

If **Rd = 1 → 31**
Repeat *Rd* times
*Rs → CRd
Rs + 32 → Rs
advance to next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	1	R	Rd			
1	0	0	0	1	1	0	1	0	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register (indirect postincrement) containing the address of the first 32-bit single-precision floating-point value to move to the coprocessor

CRd Coprocessor destination register to hold the first 32-bit single-precision floating-point value

Rd TMS34020 register containing the number of 32-bit transfers to make. This value must in the range 0 to 31

- If *Rd* = 0, then *32* 32-bit transfers are made
- If *Rd* = 1 → 31, then *Rd* 32-bit transfers are made

Description

MOVF moves 32-bit single-precision values from memory beginning at the address in Rs into coprocessor registers beginning at CRd. After each transfer, Rs is incremented, and CRd is advanced to the next register in the coprocessor register sequence list. The number of 32-bit transfers made is determined by the contents of *Rd*.

Machine States

If Rd = 0 and	Rs is aligned	36
If Rd = 0 and	Rs is nonaligned	37
If Rd = 1 → 31 and	Rs is aligned	5 + (*Rd* – 1)
If Rd = 1 → 31 and	Rs is nonaligned	6 + (*Rd* – 1)

Instruction Type

CMOVMC, postincrement, register count

Example

```
MOVF *A5+, RB7, B7
```

This instruction moves 32-bit single-precision values from TMS34020 memory location pointed to by A5 to coprocessor registers beginning at RA7. After each 32-bit transfer, register A5 is incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. B7 holds the number of 32-bit transfers to be made.

Syntax

MOVF **Rs+, CRd* [, *count*]

Execution

Repeat *count* times
*Rs → CRd
Rs + 32 → Rs
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	0	transfers				
1	0	0	0	1	1	0	1	0	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register (indirect postincrement) containing the address of the first 32-bit single-precision floating-point value to move to the coprocessor

CRd Coprocessor destination register to hold the first 32-bit single-precision floating-point value

count Contains the number of 32-bit transfers to make. This value must in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVF moves 32-bit single-precision values from memory beginning at the address in Rs into coprocessor registers beginning at CRd. After each transfer, the contents of Rs are incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVMC, postincrement, constant count

Example

```
MOVF *A5+, RB7, 4
```

This example moves four 32-bit single-precision values from TMS34020 memory location pointed to by A5 to coprocessor registers beginning at RB7. After each 32-bit transfer, register A5 is incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVF *–*Rs, CRd* [, *count*]

Execution

Repeat *count* times
Rs – 32 → Rs
*Rs → CRd
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	1	transfers				
1	0	0	0	1	1	0	1	0	0	0	R	Rs			
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

Rs TMS34020 source register (indirect postincrement) containing the address of the bit immediately after first 32-bit single-precision floating-point value to move to the coprocessor

CRd Coprocessor destination register to hold the first 32-bit single-precision floating-point value

count Contains the number of 32-bit transfers to make. This value must be in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVF moves 32-bit single-precision values from memory beginning at the address (Rs – 32) into coprocessor registers beginning at CRd. Before each transfer, the contents of Rs are decremented; after each transfer, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVMC, predecrement, constant count

Example

```
MOVF –*A5, RB7, 4
```

This example moves four 32-bit single-precision values from TMS34020 memory location pointed to by (A5 – 32) to coprocessor registers beginning at RB7. Before each 32-bit transfer, register A5 is decremented; after each transfer, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVF *CRd, *Rd+* [, *count*]

Execution

Repeat *count* times
CRd → *Rd
Rd + 32 → Rd
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	0	1	R	Rd			
1	0	0	0	1	1	1	1	0	0	0	transfers				
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register for the first 32-bit single-precision floating-point value to TMS34020 memory

Rd TMS34020 register (indirect postincrement) containing the address for the first 32-bit single-precision floating-point value transferred

count Contains the number of 32-bit transfers to make. This value must be in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVF moves the 32-bit single-precision values from coprocessor registers beginning at CRd to memory beginning at the address in Rd. After each 32-bit transfers, Rd is incremented, and the coprocessor register is advanced to the next register in the coprocessor register sequence. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVCM, postincrement, constant count

Example

```
MOVF RB7, *A5+, 4
```

This example moves four 32-bit single-precision values from coprocessor registers beginning at RB7 to TMS34020 memory pointed to by A5. After each 32-bit transfer, register A5 is incremented, and the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVF *CRd, –*Rd* [, *count*]

Execution

Repeat *count* times
Rd – 32 → Rd
CRd → *Rd
advance to the next coprocessor register

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	1	1	0	R	Rd			
1	0	0	0	1	1	1	1	0	0	0	transfers				
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register for the first 32-bit single-precision floating-point value to TMS34020 memory

Rd TMS34020 register (indirect predecrement) containing the address of the bit immediately following the 32-bits used to store the first 32-bit single-precision floating-point value transferred

count Contains the number of 32-bit transfers to make. This value must in the range 1 to 32; the default value is 1. *Count* determines the value of transfers:

- If *count* = 32, then transfers = 0
- If *count* = 1 → 31, then transfers = *count*

Description

MOVF moves the 32-bit single-precision values from coprocessor registers beginning at CRd to memory beginning at the address (Rd – 32). Before each 32-bit transfer, Rd is decremented; after each transfer, the coprocessor register is advanced to the next register in the coprocessor register sequence. The number of 32-bit transfers made is determined by the contents of *count*.

Machine States

Rs Aligned	5 + (*count* – 1)
Rs Nonaligned	6 + (*count* – 1)

Instruction Type

CMOVCM, predecrement, constant count

Example

```
MOVF RB7, –*A5, 4
```

This example moves four 32-bit single-precision values from coprocessor registers beginning at RB7 to TMS34020 memory pointed to by (A5 – 32). Before each 32-bit transfer, register A5 is decremented; after each 32-bit transfer, the coprocessor destination is advanced to the next register in the coprocessor register sequence list. *Count* specifies that four 32-bit transfers are made.

Syntax

MOVF *CRd, Rd*

Execution

CRd → Rd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	1	R	Rd			
0	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			0	0	0	0	0	0	0	0	CRd				

Operands

CRd Coprocessor source register for the 32-bit single-precision floating-point value

Rd TMS34020 destination register

Description

MOVF moves the contents (single-precision value) of CRd to Rd.

Machine States

5 if the first instruction word is long word-aligned
4 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
MOVF RA7, A5
```

This example moves the 32-bit single-precision value from coprocessor register RA7 to TMS34020 register A5.

Syntax

MOVF *CRs$_1$, CRd*

Execution

CRs$_1$ → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	0	1	1	0
Default ID			CRs$_1$				0	0	0	1	CRd				

Operands

CRs$_1$ Coprocessor source register A that holds the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

MOVF moves the contents (single-precision value) of CRs$_1$ (register A) to CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MOVF RA7, RB4
```

This example moves the 32-bit single-precision value from coprocessor register RA7 to coprocessor register RB4.

Syntax

MOVF *CRs_2, CRd*

Execution

$CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	0	1	0
Default ID			CRs_2				0	0	0	1	CRd				

Operands

CRs_2 Coprocessor source register B that holds the 32-bit single-precision floating-point value

CRd Coprocessor destination register

Description

MOVF moves 32-bit single-precision value from CRs_2 (register B) to CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MOVF RB3, RB4
```

This example moves the 32-bit single-precision value from coprocessor register RB3 to coprocessor register RB4.

Syntax

MPYD *CRs_1, CRs_2, CRd*

Execution

$CRs_1 \times CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	0	0	1	1
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 64-bit double-precision floating-point operand

CRs_2 Coprocessor register containing the second 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

MPYS multiplies the contents (double-precision value) of CRs_1 by the contents of CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MPYD RA5, RB6, RA7
```

This example multiplies the contents of RA5 by RB6 and stores the result in RA7.

Syntax

MPYF *CRs_1, CRs_2, CRd*

Execution

$CRs_1 \times CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	0	0	1	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register containing the second 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

MPYF multiplies the contents (single-precision value) of CRs_1 by the contents of CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MPYF RA5, RB6, RA7
```

This example multiplies the contents of RA5 by RB6 and stores the result in RA7.

Syntax

MPYF *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 \times CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	0	0	1	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first 32-bit single-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit single-precision floating-point value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit single-precision floating-point operand

CRs_2 Coprocessor register to contain the second 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

MPYS loads the contents (single-precision value) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, multiplies $CRs_1 \times CRs_2$, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
MPYF A5, A6, RA5, RB6, RA7
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6 respectively, multiplies the contents of RA5 by RB6, and stores the result in RA7.

Syntax

MPYS *CRs_1, CRs_2, CRd*

Execution

$CRs_1 \times CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor register containing the first 32-bit integer operand

CRs_2 Coprocessor register containing the second 32-bit integer operand

CRd Coprocessor destination register

Description

MPYS multiplies the contents (integer) of CRs_1 by the contents of CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
MPYS RA5, RB6, RB7
```

This example multiplies the contents of RA5 by RB6 and stores the result in RB7.

Syntax

MPYS *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 \times CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	1	0	0	0	0	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first 32-bit integer value to coprocessor

Rs_2 TMS34020 source register for the second 32-bit integer value to coprocessor

CRs_1 Coprocessor register to contain the first 32-bit integer operand

CRs_2 Coprocessor register to contain the second 32-bit integer operand

CRd Coprocessor destination register

Description

MPYS loads the contents (integer) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, multiplies $CRs_1 \times CRs_2$, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
MPYS A5, A6, RA5, RB6, RB7
```

This example loads TMS34020 registers A5 and A6 into coprocessor registers RA5 and RB6, multiplies the contents of RA5 by RB6, and stores the result in RB7.

Syntax

NEG *CRs, CRd*

Execution

–CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
Default ID			CRs				0	0	1	1	CRd				

Operands

CRs Coprocessor source register containing the 32-bit integer operand

CRd Coprocessor destination register

Description

NEG takes the 2s complement of the contents (integer) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
NEG RA5, RB7
```

This example takes the 2s complement of the contents of RA5 and stores the result in RB7.

Syntax

NEG *Rs, CRs, CRd*

Execution

Rs → CRs
–CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Default ID			CRs				0	0	1	1	CRd				

Operands

Rs TMS34020 source register for the 32-bit integer value to coprocessor

CRs Coprocessor register to contain the 32-bit integer operand

CRd Coprocessor destination register

Description

NEG loads the contents (integer) of Rs into CRs, takes the 2s complement of the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
NEG A5, RA5, RB7
```

This example loads TMS34020 register A5 into coprocessor register RA5, takes the 2s complement of the contents of RA5, and stores the result in RB7.

Syntax

NEGD *CRs, CRd*

Execution

–CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
Default ID			CRs				0	0	1	1	CRd				

Operands

CRs Coprocessor register containing the 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

NEGD negates the contents (double-precision value) of register CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
NEGD RA5, RB7
```

This example negates the contents of RA5 and stores the result in RB7.

Syntax

NEGF *CRs, CRd*

Execution

–CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
Default ID			CRs				0	0	1	1	CRd				

Operands

CRs Coprocessor register containing the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

NEGF negates the contents (single-precision value) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
NEGF RA5, RA7
```

This example negates the contents of RA5 and stores the result in RA7.

Syntax

NEGF *Rs, CRs, CRd*

Execution

Rs → CRs
–CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			CRs				0	0	1	1	CRd				

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to coprocessor

CRs Coprocessor register to contain the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

NEGF loads the contents (single-precision value) of Rs into CRs, negates the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
NEGF A5, RA5, RB7
```

This example loads TMS34020 register A5 into coprocessor register RA5, negates the contents of RA5, and stores the result in RB7.

Syntax

NOT *CRs, CRd*

Execution

NOT CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
Default ID			CRs				0	0	0	1	CRd				

Operands

CRs Coprocessor source register containing the 32-bit integer operand

CRd Coprocessor destination register

Description

NOT takes the 1s complement of the contents (integer) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
NOT RA5, RA7
```

This example takes the 1s complement of the contents of RA5 and stores the result in RA7.

Syntax

NOT *Rs, CRs, CRd*

Execution

Rs → CRs
NOT CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Default ID			CRs				0	0	0	1	CRd				

Operands

Rs TMS34020 source register for the 32-bit integer value to coprocessor

CRs Coprocessor register to contain the 32-bit integer operand

CRd Coprocessor destination register

Description

NOT loads the contents (integer) of Rs into the CRs, takes the 1s complement of the contents of register CRs, and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
NOT A5, RA5, RA7
```

This example loads TMS34020 register A5 into coprocessor register RA5, takes the 1s complement of the contents of RA5, and stores the result in RA7.

Syntax **SQR** *CRs, CRd*

Execution CRs × CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
Default ID			CRs				1	0	0	0	CRd				

Operands

CRs Coprocessor source register containing the 32-bit integer operand

CRd Coprocessor destination register

Description

SQR squares the contents (integer) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States 2

Instruction Type CEXEC, short

Example

```
SQR RA5, RA7
```

This example squares the contents of RA5 and stores the result in register RA7.

Syntax

SQR *Rs, CRs, CRd*

Execution

Rs → CRs
CRs × CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Default ID			CRs				1	0	0	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit integer value to coprocessor

CRs Coprocessor register to contain the 32-bit integer operand

CRd Coprocessor destination register

Description

SQR loads the contents (integer) of Rs into CRs, squares the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
SQR A5, RA5, RB7
```

This example loads TMS34020 register A5 into coprocessor register RA5, squares the contents of RA5, and stores the result in RB7.

Syntax

SQRD *CRs, CRd*

Execution

CRs × CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
Default ID			CRs				1	0	0	0	CRd				

Operands

CRs Coprocessor register containing the 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

SQRD squares the contents (double-precision value) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SQRD RA5, RA7
```

This example squares the contents of RA5 and stores the result in RA7.

Syntax

SQRF *CRs, CRd*

Execution

CRs × CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
Default ID			CRs				1	0	0	0	CRd				

Operands

CRs Coprocessor source register containing the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SQRF squares the contents (single-precision value) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SQRF RA5, RB7
```

This example squares the contents of RA5 and stores the result in RB7.

Syntax

SQRF *Rs, CRs, CRd*

Execution

Rs → CRs
CRs × CRs → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			CRs				1	0	0	0	CRd				

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to coprocessor

CRs Coprocessor register to contain the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SQRF loads the contents of Rs into CRs, squares the contents (single-precision value) of CRs, and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
SQRF A5, RA5, RB7
```

This example loads TMS34020 register A5 into coprocessor register RA5, squares the contents of RA5, and stores the result in RB7.

Syntax

SQRT *CRs, CRd*

Execution

$\sqrt{CRs} \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	0
Default ID			CRs				1	0	0	1	CRd				

Operands

CRs Coprocessor register containing the 32-bit integer operand

CRd Coprocessor destination register

Description

SQRT takes the square root of the contents (integer) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SQRT RA5, RB7
```

This example takes the square root of the contents of RA5 and stores the result in RB7.

Syntax

SQRT *Rs, CRs, CRd*

Execution

Rs → CRs

$\sqrt{CRs}$ → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Default ID			CRs				1	0	0	1	CRd				

Operands

Rs TMS34020 source register for the 32-bit integer value to coprocessor

CRs Coprocessor register to contain the 32-bit integer operand

CRd Coprocessor destination register

Description

SQRT loads the contents (integer) of Rs into CRs, takes the square root of the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
SQRT A5, RA5, RA7
```

This example loads TMS34020 register A5 into coprocessor register RA5, takes the square root of the contents of RA5, and stores the result in RA7.

Syntax

SQRTD *CRs, CRd*

Execution

$\sqrt{CRs} \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1
Default ID			CRs				1	0	0	1	CRd				

Operands

CRs Coprocessor register containing the 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

SQRTD takes the square root of the contents (double-precision value) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SQRTD RA5, RA7
```

This example takes the square root of the contents of RA5 and stores the result in RA7.

Syntax

SQRTF *CRs, CRd*

Execution

$\sqrt{CRs} \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	1	1	1	1	1	0
Default ID			CRs				1	0	0	1	CRd				

Operands

CRs Coprocessor register containing the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SQRTF takes the square root of the contents (single-precision value) of CRs and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SQRTF RA5, RA7
```

This example takes the square root of the contents of RA5 and stores the result in RA7.

Syntax

SQRTF *Rs, CRs, CRd*

Execution

Rs → CRs

$\sqrt{CRs}$ → CRd

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	1	R	Rs			
0	1	0	1	1	1	1	1	0	0	0	0	0	0	0	0
Default ID			CRs				1	0	0	1	CRd				

Operands

Rs TMS34020 source register for the 32-bit single-precision floating-point value to coprocessor

CRs Coprocessor register to contain the 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SQRTF loads the contents (single-precision value) of Rs into CRs, takes the square root of the contents of CRs, and stores the result in CRd.

The source register, CRs, must be in the A coprocessor register file.

Machine States

3 if the first instruction word is long word-aligned
2 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, one register

Example

```
SQRTF A5, RA5, RA7
```

This example loads TMS34020 register A5 into coprocessor register RA5, takes the square root of the contents of RA5, and stores the result in RA7.

Syntax

SUB *CRs_1, CRs_2, CRd*

Execution

$CRs_1 - CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	1	0	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor A register containing the 32-bit minuend integer operand

CRs_2 Coprocessor B register containing the 32-bit subtrahend integer operand

CRd Coprocessor destination register

Description

SUB subtracts the contents (integer) of CRs_2 from CRs_1 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SUB RA5, RB3, RA7
```

This example subtracts the contents of RB3 from RA5 and stores the result in RA7.

Syntax

SUB *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 - CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	0	1	0	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first (minuend) 32-bit integer value to coprocessor

Rs_2 TMS34020 source register for the second (subtrahend) 32-bit integer value to coprocessor

CRs_1 Coprocessor A register to contain the 32-bit minuend integer operand

CRs_2 Coprocessor B register to contain the 32-bit subtrahend integer operand

CRd Coprocessor destination register

Description

SUB loads the contents (integer) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts the contents of CRs_2 from CRs_1, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
SUB A0, B6, RA5, RB3, RA7
```

This example loads TMS34020 registers A0 and B6 into coprocessor registers RA5 and RB3, subtracts the contents of RB3 from RA5, and stores the result in RA7.

Syntax

SUB *CRs_2, CRs_1, CRd*

Execution

$CRs_2 - CRs_1 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	1	0	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor A register containing the 32-bit subtrahend integer operand

CRs_2 Coprocessor B register containing the 32-bit minuend integer operand

CRd Coprocessor destination register

Description

SUB subtracts the contents (integer) of CRs_1 from CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SUB RB5, RA3, RA7
```

This example subtracts the contents of RA3 from RB5 and stores the result in RA7.

Syntax

SUB *Rs_2, Rs_1, CRs_2, CRs_1, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_2 - CRs_1 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	1	1	0	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first (subtrahend) 32-bit integer value to coprocessor

Rs_2 TMS34020 source register for the second (minuend) 32-bit integer value to coprocessor

CRs_1 Coprocessor A register to contain the 32-bit subtrahend integer operand

CRs_2 Coprocessor B register to contain the 32-bit minuend integer operand

CRd Coprocessor destination register

Description

SUB loads the contents (integer) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts the contents of CRs_1 from CRs_2, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
SUB B6, A0, RB5, RA3, RA7
```

This example loads TMS34020 registers B6 and A0 into coprocessor registers RB5 and RA3, subtracts the contents of RA3 from RB5, and stores the result in RA7.

Syntax

SUBD *CRs_1, CRs_2, CRd*

Execution

$CRs_1 - CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	1	1	1
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor A register containing the minuend 64-bit double-precision floating-point operand

CRs_2 Coprocessor B register containing the subtrahend 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

SUBD subtracts the contents (double-precision value) of CRs_2 from CRs_1 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SUBD RA5, RB3, RA7
```

This example subtracts the contents of RB3 from RA5 and stores the result in RA7.

Syntax

SUBD *CRs_2, CRs_1, CRd*

Execution

$CRs_2 - CRs_1 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	1	1	1
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor A register containing the subtrahend 64-bit double-precision floating-point operand

CRs_2 Coprocessor B register containing the minuend 64-bit double-precision floating-point operand

CRd Coprocessor destination register

Description

SUBD subtracts the contents (double-precision value) of CRs_1 from CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SUBD RB5, RA3, RA7
```

This example subtracts the contents of RA3 from RB5 and stores the result in RA7.

Syntax

SUBF *CRs_1, CRs_2, CRd*

Execution

$CRs_1 - CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	0	1	1	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor A register containing the minuend 32-bit single-precision floating-point operand

CRs_2 Coprocessor B register containing the subtrahend 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SUBF subtracts the contents (single-precision value) of CRs_2 from CRs_1 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SUBF RA5, RB3, RA7
```

This example subtracts the contents of RB3 from RA5 and stores the result in RA7.

Syntax

SUBF *Rs_1, Rs_2, CRs_1, CRs_2, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_1 - CRs_2 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	0	1	1	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2				CRd				

Operands

Rs_1 TMS34020 source register for the first (minuend) 32-bit single-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for the second (subtrahend) 32-bit single-precision floating-point value to coprocessor

CRs_1 Coprocessor A register to contain the minuend 32-bit single-precision floating-point operand

CRs_2 Coprocessor B register to contain the subtrahend 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SUBF loads the contents (single-precision value) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts the contents of CRs_2 from CRs_1, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
SUBF A0, B6, RA5, RB3, RA7
```

This example loads TMS34020 registers A0 and B6 into coprocessor registers RA5 and RB3, subtracts the contents of RB3 from RA5, and stores the result in RA7.

Syntax

SUBF *CRs_2, CRs_1, CRd*

Execution

$CRs_2 - CRs_1 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	0	0	0	0	0	1	1	1	0
Default ID			CRs_1				CRs_2				CRd				

Operands

CRs_1 Coprocessor A register containing the subtrahend 32-bit single-precision floating-point operand

CRs_2 Coprocessor B register containing the minuend 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SUBF subtracts the contents (single-precision value) of CRs_1 from CRs_2 and stores the result in CRd.

Machine States

2

Instruction Type

CEXEC, short

Example

```
SUBF RB5, RA3, RA7
```

This example subtracts the contents of RA3 from RB5 and stores the result in RA7.

Syntax

SUBF *Rs_2, Rs_1, CRs_2, CRs_1, CRd*

Execution

$Rs_1 \rightarrow CRs_1$
$Rs_2 \rightarrow CRs_2$
$CRs_2 - CRs_1 \rightarrow CRd$

Instruction Words

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	R	Rs_1			
0	1	0	0	0	1	1	1	0	0	0	R	Rs_2			
Default ID			CRs_1				CRs_2					CRd			

Operands

Rs_1 TMS34020 source register for the first (subtrahend) 32-bit single-precision floating-point value to coprocessor

Rs_2 TMS34020 source register for the second (minuend) 32-bit single-precision floating-point value to coprocessor

CRs_1 Coprocessor A register to contain the subtrahend 32-bit single-precision floating-point operand

CRs_2 Coprocessor B register to contain the minuend 32-bit single-precision floating-point operand

CRd Coprocessor destination register

Description

SUBF loads the contents (single-precision value) of Rs_1 and Rs_2 into CRs_1 and CRs_2 respectively, subtracts the contents of CRs_1 from CRs_2, and stores the result in CRd.

Machine States

4 if the first instruction word is long word-aligned
3 if the first instruction word is not long word-aligned

Instruction Type

CMOVGC, two registers

Example

```
SUBF B6, A0, RB5, RA3, RA7
```

This example loads TMS34020 registers B6 and A0 into coprocessor registers RB5 and RA3, subtracts the contents of RA3 from RB5, and stores the result in RA7.

Chapter 15

Instruction Timing

This chapter summarizes the timings of the TMS34020 assembly-language instruction set. It contains two sections:

Please note these characteristics about the timings listed in this book:

- Numbers identify TMS34020 machine states.
- All timings assume that the cache is enabled and that the instruction is in the cache.
- Numbers in parentheses identify hidden cycles.

  The TMS34020 may execute some instructions in parallel, "hiding" some instruction states. Hidden cycles are memory-write cycles that occur at the end of an instruction. The machine states consumed by the instruction that the CPU is executing hide the machine states consumed by the write cycles. These hidden cycles are not counted against he instruction that incurs them, but are counted against subsequent instructions. If an instruction uses the local bus before all of the hidden cycles have been overlapped by subsequent instructions, that instruction must wait for the hidden cycles to complete.
- These timings assume that
  - All memory requests are granted when requested; no higher priority memory requests are pending.
  - When the CPU requests page-mode access, the memory grants it.
  - No wait states occur.
  - No retries occur.

15.1 Timing for All Instructions Except MOVEs and MOVBs

This section lists the instructions for all instructions except the MOVE and MOVB instructions. Please note that

- If the timing for an instruction states that this is a ***complex instruction***, than no simple formula is available for providing the timing for the instruction. The number of machine states consumed by this instruction's execution will vary depending on the circumstances of its execution.
- Instruction timing for graphics instructions varies, depending on the pixel-processing option you've selected. The timing formulas for graphics instructions (such as DRAV and LINE) ask you to add the values shown in Table 15–1 into your timing calculations.

Table 15–1. Effects of Pixel-Processing Options on Graphics Instructions

Pixel-Processing Option	Number of Cycles Required for the Following Pixel Sizes		
	1	2 or 4	8, 16, or 32
Replace	0 (2)	0 (2)	0 (1)
ADD	—	2 (2)	2 (1)
ADDS	—	3 (2)	3 (1)
SUB	—	2 (2)	2 (1)
SUBS	—	3 (2)	3 (1)
MAX	—	3 (2)	3 (1)
MIN	—	3 (2)	3 (1)
PPCODE	2 (2)	2 (2)	2 (1)

Instruction	Number of machine cycles consumed by instruction execution
ABS	1
ADD	1
ADDC	1
ADDI (short)	2
ADDI (long)	2 if the immediate data is long-word aligned 3 if the immediate data is not long-word aligned
ADDK	1
ADDXY	1
ADDXYI	2 if the immediate data is long-word aligned 3 if the immediate data is not long-word aligned
AND	1

Instruction	Number of machine cycles consumed by instruction execution
ANDI	2 if the immediate data is long-word aligned 3 if the immediate data is not long-word aligned
ANDN	1
ANDNI	2 if the immediate data is long-word aligned 3 if the immediate data is not long-word aligned
BLMOVE	***complex instruction***
BTST (constant)	1
BTST (register)	1
CALL	3 + (1) if the SP is aligned 3 + (4) if the SP is not aligned
CALLA	3 if immediate data is long-word aligned, 4 if SP is also long-word aligned 3+(3) if immediate data is not long-word aligned, 4+(3) if SP is also not long-word aligned
CALLR	3 + (1) if the SP is long-word aligned 3 + (4) if the SP is long-word not aligned
CEXEC (long)	2 (1) if the immediate data is long-word aligned 3 (1) if the immediate data is not long-word aligned
CEXEC (short)	2 (1)
CLIP	***complex instruction***
CLR	1
CLRC	1
CMOVCG	Single: 4 if the immediate data is long-word aligned 5 if the immediate data is not long-word aligned Double: 5 if the immediate data is long-word aligned 6 if the immediate data is not long-word aligned
CMOVCM (count*+)	5 + [count–1] if the immediate data is long-word aligned 6 + [count–1] if it is not (count is the number of 32-bit transfers)
CMOVCM (count–*)	5 + [count–1] if the immediate data is long-word aligned 6 + [count–1] if it is not (count is the number of 32-bit transfers)
CMOVCS	4 if the immediate data is long-word aligned 5 if it is not
CMOVGC (one register)	2 (1) if the immediate data is long-word aligned 3 (1) if it is not
CMOVGC (two registers)	3 (1) if the immediate data is long-word aligned 4 (1) if it is not

Instruction	Number of machine cycles consumed by instruction execution
CMOVMC (constant*+)	5 + [constant–1] if the immediate data is long-word aligned 6 + [constant–1] if it is not (constant is the number of 32-bit transfers)
CMOVMC (constant*–)	5 + [constant–1] if the immediate data is long-word aligned 6 + [constant–1] if it is not (constant is the number of 32-bit transfers)
CMOVMC (register*+)	5 + [register value–1] if the immediate data is long-word aligned 6 + [register value–1] if it is not (the register value is the number of 32-bit transfers)
CMP	1
CMPI (long)	2 if the immediate data is long-word aligned 3 if it is not
CMPI (short)	2
CMPK	1
CMPXY	1
CPW	1
CVDXYL	pitch is a power of 2: 2 2 powers of 2: 3 arbitrary: 14
CVMXYL	pitch is a power of 2: 2 2 powers of 2: 3 arbitrary: 14
CVSXYL	pitch is a power of 2: 2 2 powers of 2: 3 arbitrary: 14
CVXYL	pitch is a power of 2: 3 2 powers of 2: 4 arbitrary: 15
DEC	1
DINT	3
DIVS	Rd Odd: 39 (normal case) 41 (if result = 80000000h) 7 (if Rs = 0) Rd Even: 40 (normal case) 41 (if result = 80000000h) 7 (if Rs = 0 or Rs ≤ Rd)
DIVU	Rd Odd: 37 (normal case) 7 (if Rs = 0) Rd Even: 37 (normal case) 5 (if Rs = 0 or Rs ≤ Rd)

Instruction	Number of machine cycles consumed by instruction execution
DRAV	*Window option* — *0*: *inside* 4+*P*+*CD*, *outside* 4+*P*+*CD*; *1*: *inside* 5, *outside* 3; *2*: *inside* 4+*P*+*CD*, *outside* 5; *3*: *inside* 4+*P*+*CD*, *outside* 3
	Key: *P* Selected pixel-processing option; see Table 15–1. *CD* Complexity of destination pitch. *CD* = 0 if CONVDP contains a power of 2; *CD* = 1 if CONVDP contains a sum of powers of 2; *CD* = 12 if CONVDP contains an arbitrary pitch.
DSJ	2 if no jump 3 if jump
DSJEQ	2 if no jump 3 if jump
DSJNE	2 if no jump 3 if jump
DSJS	2 if no jump 3 if jump
EINT	3
EMU	8 (more if the TMS34020 enters emulation mode)
EXGF	1 if F0 2 if F1
EXGPC	2
EXGPS	2 (1)
FILL L	***complex instruction***
FILL XY	***complex instruction***
FLINE	$12 + 3CD + [2 + P]E + 3$ **Key:** *P* Selected pixel-processing option; see Table 15–1. If the number of hidden cycles is greater than 1, then $P = P +$ (hidden cycles – 1). *E* Total number of pixels drawn. *CD* Complexity of destination pitch. *CD* = 0 if CONVDP contains a power of 2; *CD* = 1 if CONVDP contains a sum of powers of 2; *CD* = 12 if CONVDP contains an arbitrary pitch.
FPIXEQ	***complex instruction***
FPIXNE	***complex instruction***
GETPC	1
GETPS	2
GETST	1
IDLE	minimum execution time of 1 cycle before taking interrupt EMU: 5 cycles min before responds to halt NMI mode1: 8 NMI mode0, HINT, DPYINT, WINT, INT1, or INT2: 11 if SP aligned, else 13
INC	1

Instruction	Number of machine cycles consumed by instruction execution
JA*cc*	3 if no jump, else 4
JR*cc* (short)	1 if no jump, else 2
JR*cc* (long)	2 if no jump, else 3
JUMP	2
LINE	Window option 0: 13 + 3*CD* + [3 +*P*]*E* + 2 Window option 1: 13 + 3*CD* + [3 +*P*]*Q* + 2 Window option 2: 13 + 3*CD* + [3 +*P*]*E* + *WV* + 2 Window option 3: 13 + 3*CD* + [3 +*P*]*E* + 3*Q* + 2 **Key:** *P* Selected pixel-processing option; see Table 15–1, but ignore the hidden cycles. *WV* =3 if there is a window violation, = 0 otherwise. *Q* Total number of pixels calculated but not drawn. *E* Total number of pixels drawn. *CD* Complexity of destination pitch. *CD* = 0 if CONVDP contains a power of 2; *CD* = 1 if CONVDP contains a sum of powers of 2; *CD* = 12 if CONVDP contains an arbitrary pitch.
LINIT	9
LMO	1
MMFM	*# registers moved* 1 2 3 4 *n* *# cycles* 6 7 8 9 *n* + 5
MMTM	*# registers moved* 1 2 3 4 *n* *long-word aligned* 4(1) 6(1) 7(1) 8(1) [4 + *n*](1) *byte aligned* 4(1) 8(1) 9(1) 10(1) [6 + *n*](1) *bit aligned* 4(2) 9(2) 10(2) 11(2) [7 + *n*](1) **Note:** Add 1 to all timings if the MMTM instruction is not long-word aligned.
MODS	40 41 if result = 8000 0000h 3 if Rs = 0
MODU	35 3 if Rs = 0
MOVE *Rs*, *Rd*	1
MOVI (long)	2 if immediate data is long-word aligned 3 if it isn't
MOVI (short)	2
MOVK	1
MOVX	1
MOVY	1
MPYS	Rs negative: 5 + (field size 1)/2 Rs positive: 6 + (field size 1)/2
MPYU	5 + (field size 1)/2

Instruction	Number of machine cycles consumed by instruction execution
MWAIT	minimum of 2
NEG	1
NEGB	1
NOP	1
NOT	1
OR	1
ORI	2 if immediate data is long-word aligned 3 if it isn't
PFILL	***complex instruction***
PIXBLT B, L	***complex instruction***
PIXBLT B, XY	***complex instruction***
PIXBLT L, L	***complex instruction***
PIXBLT L, M, L	***complex instruction***
PIXBLT L, XY	***complex instruction***
PIXBLT XY, L	***complex instruction***
PIXBLT XY, XY	***complex instruction***
PIXT *Rs*, **Rd*	2 + *P*
PIXT *Rs*, **Rd*.XY	*Window option* — *0*: *inside* 4+*CD*+*P*, *outside* 4+*CD*+*P*; *1*: *inside* 5, *outside* 3; *2*: *inside* 4+*CD*+*P*, *outside* 5+*CD*; *3*: *inside* 3+*CD*+*P*, *outside* 3+*CD*
PIXT **Rs*, *Rd*	3
PIXT **Rs*, **Rd*	4 + *P*
PIXT **Rs*.XY, *Rd*	6 + *CS*
PIXT **Rs*.XY, **Rd*.XY	*Window option* — *0*: *inside* 7+*CS*+*CD*+*P*, *outside* 7+*CS*+*CD*+*P*; *1*: *inside* 5, *outside* 3; *2*: *inside* 7+*CS*+*CD*+*P*, *outside* 5+*CD*; *3*: *inside* 7+*CS*+*CD*+*P*, *outside* 3+*CD*

Key: *P* Selected pixel-processing option; see Table 15–1 (page 15-2).
CD Complexity of destination pitch. *CD* = 0 if CONVDP contains a power of 2; *CD* = 1 if CONVDP contains a sum of powers of 2; *CD* = 12 if CONVDP contains an arbitrary pitch.
CS Complexity of source pitch. *CS* = 0 if CONVSP contains a power of 2; *CS* = 1 if CONVSP contains a sum of powers of 2; *CS* = 12 if CONVSP contains an arbitrary pitch.

POPST	6 if the SP is aligned 7 if it isn't
PUSHST	2 (1) if the SP is aligned 2 (2) if it isn't
PUTST	3

Instruction	Number of machine cycles consumed by instruction execution
RETI	52 if BF status bit = 1 38 if IX status bit = 1 else 7
RETM	52 if BF status bit = 1 38 if IX status bit = 1 else 10
RETS	5 6 if the stack isn't aligned
REV	1
RL (constant)	1
RL (register)	1
RMO	1
RPIX	2 if PSIZE = 32 4 if PSIZE = 16 5 if PSIZE = 8 6 if PSIZE = 4 7 if PSIZE = 2 8 if PSIZE = 1
SETC	1
SETCDP	pitch is a power of 2: 4(1) 2 powers of 2: 6(1) arbitrary: 3(1)
SETCMP	pitch is a power of 2: 4(1) 2 powers of 2: 6(1) arbitrary: 3(1)
SETCSP	pitch is a power of 2: 4(1) 2 powers of 2: 6(1) arbitrary: 3(1)
SETF	1
SEXT	2
SLA (constant)	3
SLA (register)	3
SLL (constant)	1
SLL (register)	1
SRA (constant)	1
SRA (register)	1
SRL (constant)	1
SRL (register)	1

Instruction	Number of machine cycles consumed by instruction execution
SUB	1
SUBB	1
SUBI (long)	2 if the immediate data is long-word aligned 3 if it isn't
SUBI (short)	2
SUBK	1
SUBXY	1
SWAPF	5
TFILL	***complex instruction***
TRAP	7 if TRAP 0, else 10 if ST aligned else 12
TRAP L	10 if ST aligned else 12
VBLT	***complex instruction***
VFILL	***complex instruction***
VLCOL	2 (1)
XOR	1
XORI	2 if the immediate data is long-word aligned 3 if it isn't
ZEXT	1

15.2 Timing for MOVE and MOVB Instructions

This section contains the timing for MOVE and MOVB instructions. These timings are divided into three categories:

- Timings for memory-to-register moves (reads)
- Timings for register-memory moves (writes)
- Timings for memory-to-memory moves

General assumptions

The timing of the move instructions depends on how the accessed field is aligned in memory. The following cases of field alignment characterize the move instruction timing.

1) The field is aligned on the boundaries of a long word or on any byte boundaries.

2) At least one end of the field is not aligned to a byte boundary.

3) The field crosses a long-word boundary, but both ends are aligned on byte boundaries.

4) The field crosses a long-word boundary, and only one end is aligned on a byte boundary.

5) The field crosses a long-word boundary, and neither end is aligned on a byte boundary.

Table 15–2. Cases Table for MOVE and MOVB Timings

Case Number	Number of Read Cycles Required	Number of Write Cycles Required
1	2	2
2	2	3
3	3	3
4	3	4
5	3	5

The timing tables refer to these cases.

Memory-to-register moves

	Case				
Instruction	**1**	**2**	**3**	**4**	**5**
MOVB **Rs, Rd*	4	4	—	—	5
MOVB **Rs(SOffset), Rd*	6	6	—	—	7
MOVB *@SAddress, Rd*	5/6	5/6	—	—	6/7
MOVE **Rs, Rd*	3	3	4	4	4
sign extended:	4	4	5	5	5
MOVE **Rs+, Rd*	3	3	4	4	4
sign extended:	4	4	5	5	5
MOVE *–*Rs, Rd*	4	4	5	5	5
sign extended:	5	5	6	6	6
MOVE **Rs(SOffset), Rd*	4	4	5	5	5
sign extended:	6	6	7	7	7
MOVE *@Rs, Rd*	4/5	4/5	5/6	5/6	5/6
sign extended:	5/6	5/6	6/7	6/7	6/7

Register-to-memory moves

	Case				
Instruction	**1**	**2**	**3**	**4**	**5**
MOVB *Rs, *Rd*	1(1)	1(2)	—	—	1(4)
big endian	2	2(1)	—	—	2(3)
MOVB *Rs, *Rd*	3(1)	3(2)	—	—	3(4)
MOVB *Rs, @Rd*	2(1)/3(1)	2(2)/3(2)	—	—	2(4)/3(4)
big endian	3(1)/3(1)	3(2)/3(2)	—	—	3(4)/3(4)
MOVE *Rs, *Rd*	1(1)	1(2)	1(2)	1(3)	1(4)
big endian	2(1)	2(2)	2(2)	2(3)	2(4)
MOVE *Rs, *Rd+*	1(1)	1(2)	1(2)	1(3)	1(4)
	2(1)	2(2)	2(2)	2(3)	2(4)
MOVE *Rs, –*Rd*	2(1)	2(2)	2(2)	2(3)	2(4)
MOVE *Rs, –*Rd*	3(1)	3(2)	3(2)	3(3)	3(4)
MOVE *Rs, @Rd*	2(1)/3(1)	2(2)/3(2)	2(2)/3(2)	2(3)/3(3)	2(4)/3(4)
big endian	3(1)/3(1)	3(2)/3(2)	3(2)/3(2)	3(3)/3(3)	3(4)/3(4)

Memory-to-memory moves

First, look in Table 15–2 (page 15-10) to find the source alignment (case 1–5) and the destination alignment (case1–5). Then, useTable 15–3 to find which column to use in the timing table below.

Table 15–3. Source/Destination Alignment for MOVE and MOVB Timings

	Destination				
Source	**1**	**2**	**3**	**4**	**5**
1	A	C	C	H	E
2	A	C	C	H	E
3	B	D	D	G	F
4	B	D	D	G	F
5	B	D	D	G	F

R/W Cycles	2/2	3/2	2/3	3/3	2/5	3/5	3/4	2/4
	A	B	C	D	E	F	G	H
MOVB **Rs*, **Rd*	3(1)	4(1)	3(2)	4(2)	3(4)	4(4)		
MOVB **Rs(SOffset)*, **Rd(DOffset)*	5(1)	6(1)	5(2)	6(2)	5(2)	6(4)		
MOVB *@SAddress,@DAddress*								
even	5(1)	6(1)	5(2)	6(2)	5(4)	6(4)		
odd	7(1)	8(1)	7(2)	8(2)	7(4)	8(4)		
MOVE **Rs*, **Rd*	3(1)	4(1)	3(2)	4(2)	3(4)	4(4)	4(3)	3(3)
MOVE **Rs+*, **Rd+*	3(1)	4(1)	3(2)	4(2)	3(4)	4(4)	4(3)	3(3)
MOVE *–*Rs*, *–*Rd*	4(1)	5(1)	4(2)	5(2)	4(4)	5(4)	5(3)	4(3)
MOVE **Rs(SOffset)*, **Rd+*	5(1)	6(1)	5(2)	6(2)	5(4)	6(4)	6(3)	5(3)
MOVE **Rs(SOffset)*, **Rd(DOffset)*	5(1)	6(1)	5(2)	6(2)	5(4)	6(4)	6(3)	5(3)
MOVE *@Rs*, **Rd+*								
even	4(1)	5(1)	4(2)	5(2)	4(4)	5(4)	5(3)	4(3)
odd	5(1)	6(1)	5(2)	6(2)	5(4)	6(4)	6(3)	5(3)
MOVE *@Rs*, *@Rd*								
even	5(1)	6(1)	5(2)	6(2)	5(4)	6(4)	6(3)	5(3)
odd	7(1)	8(1)	7(2)	8(2)	7(4)	8(4)	8(3)	7(3)

Appendix A

Test and Emulation Considerations

This appendix provides information that you'll need if you're building a TMS34020 target system and you plan to use the TMS34020 Emulator. The TMS34020 Emulator supports realtime in-circuit emulation; key features include

- **Serial scan-path technology.** The emulator uses TI's revolutionary serial scan-path technology, eliminating the need for the typical emulator target cable, which uses a full device pinout. Instead, the target system needs only a 12-pin header to connect between the TMS34020 and the TMS34020 emulator board through the emulation target cable.
- **PC-compatible emulator board.** The emulator board is a PC/XT-compatible emulator board. It provides a high-speed communication path between a PC and the TMS34020.
- **Symbolic debugger with windowed interface.** The emulator's symbolic debugger provides the following features through its windowed interface:
  - Ability to upload/download application code and emulation setup
  - Software breakpoints on selected instructions
  - Single-step execution
  - Access to registers and memory
  - TMS34020 patch assembler/disassembler
  - Benchmark timing

The remainder of this appendix contains information about setting up your target system.

A.1 Overview of an Emulation System

Figure A–1 shows a typical setup using the emulator, target cable, and your target system.

Figure A–1. Typical Setup Using the TMS34020 Emulator and Your Target System

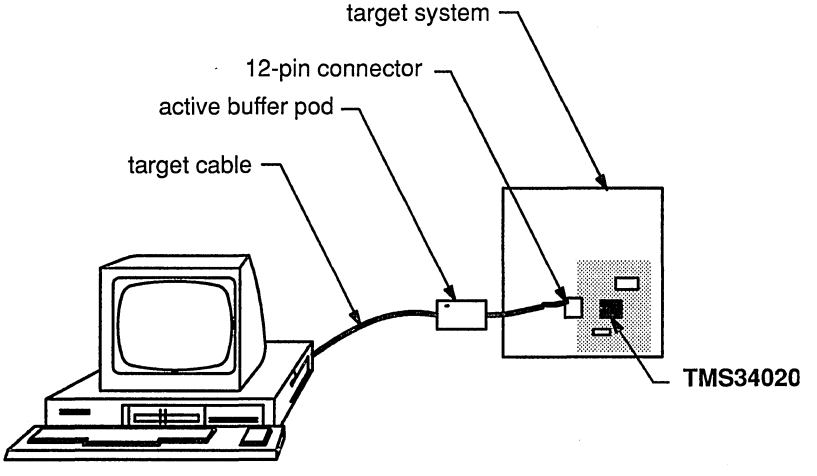
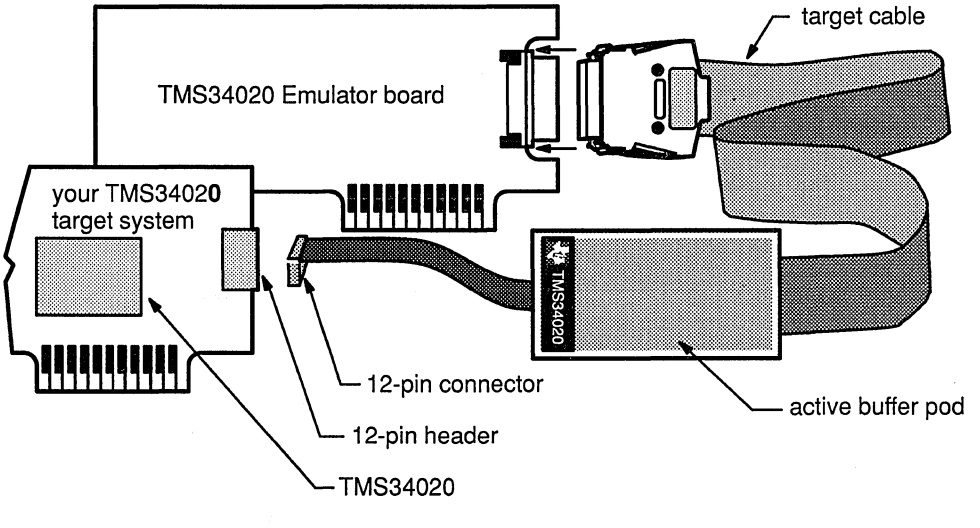


Figure A–2 shows how you connect the emulator and target cable to your target system.

Figure A–2. Connecting the TMS34020 Emulator to Your Target System

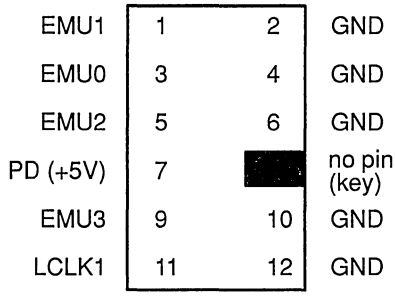


A.2 Emulation Connector (12-Pin Header)

To use the target cable, your target system must have a 12-pin header (2 rows of 6 pins) with the connections that are shown in Figure A–3. The header pins connect directly to the TMS34020 except when the header is farther than 2 inches from the TMS34020 (see Section A.3 on page A-4).

Figure A–3. 12-Pin Header Signals

Header Dimensions:
Pin-to-pin spacing: 0.100 inches (X, Y)
Pin width: 0.025 inches square post
Pin length: 0.235 inches nominal



Signal	Description	TMS34020 Pin Number
EMU0	Emulation pin 0	J1
EMU1	Emulation pin 1	J3
EMU2	Emulation pin 2	K1
EMU3	Emulation pin 3	H2
LCLK1	TMS34020 local clock 1	H1
PD	Presence detect. Indicates that the cable is connected and target system is powered up. Tie PD to +5 volts in the target system.	

Although you can use other headers, recommended parts include

straight header, unshrouded — DuPont Connector Systems part number 67996–112

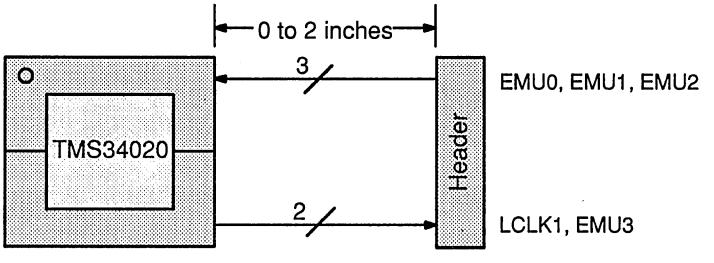
right-angle header, unshrouded — DuPont Connector Systems part number 68405–112

right-angle header, 4-wall shrouded — AMP, Incorporated part number 103167–3

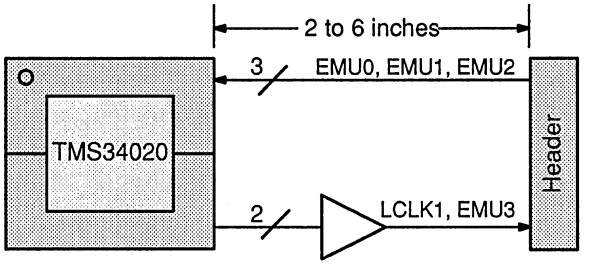
A.3 Signal Buffering

It is extremely important to provide high-quality signals between the emulator and the TMS34020 on the target system. In many cases, the signal must be buffered to produce a high-quality signal. The need for signal buffering and placement of the emulation header can be divided into 3 categories:

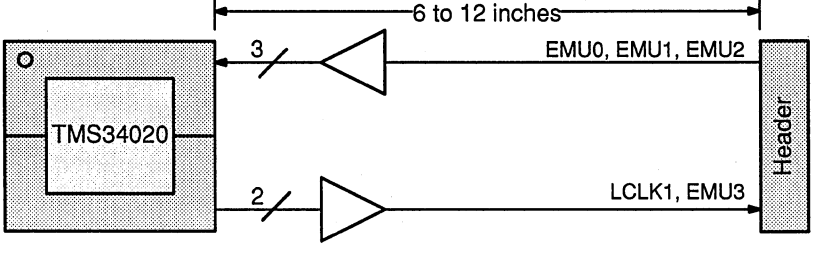
- **No signal buffering.** In this situation, the distance between the header and the TMS34020 should be no more than 2 inches.



- **Buffered transmission signals.** In this situation, the distance between the emulation header and the TMS34020 is greater than 2 inches but less than 6 inches. The transmission signals—LCLK1 and EMU3—are buffered through the same package.



- **All signals buffered.** The distance between the emulation header and the TMS34020 is greater than 6 inches but less than 12 inches. All TMS34020 emulation signals—EMU0, EMU1, EMU2, and EMU3—are buffered through the same package.



A.4 Buffer Delays

The absolute maximum propagation delay for both –32 and –40 TMS34020 devices is 10 ns. The buffer is noninverting, and all emulation signals that are buffered should be buffered through the same package.

The distance between the TMS34020 and the buffers depends on the PWB layout and loading on LCLK1. However, Texas Instruments suggests that the distance be as short as possible and less than 4 inches.

When you buffer LCLK1, don't place another device between the buffer output and header. Connecting another device to this signal could cause false triggering of the device due to cable reflections (see Figure A–4).

Figure A–4. LCLK1 Buffer Restrictions

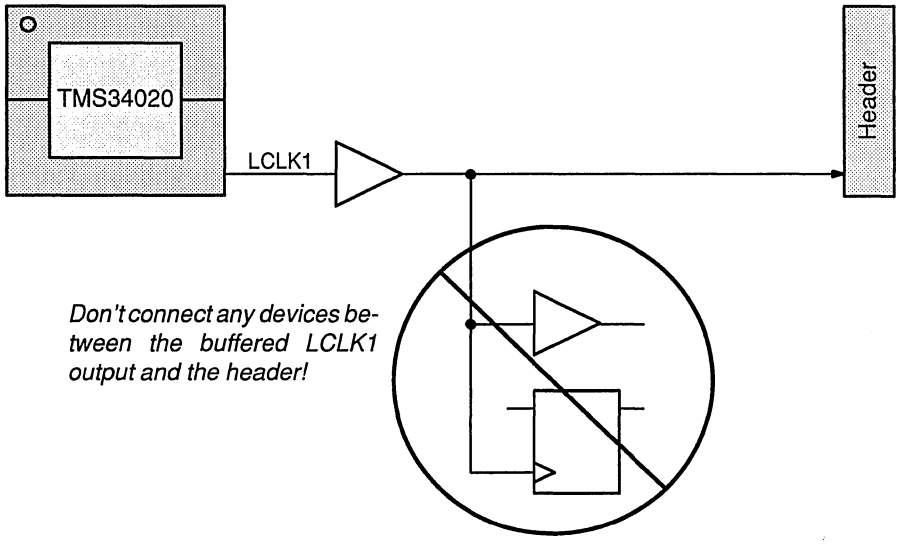
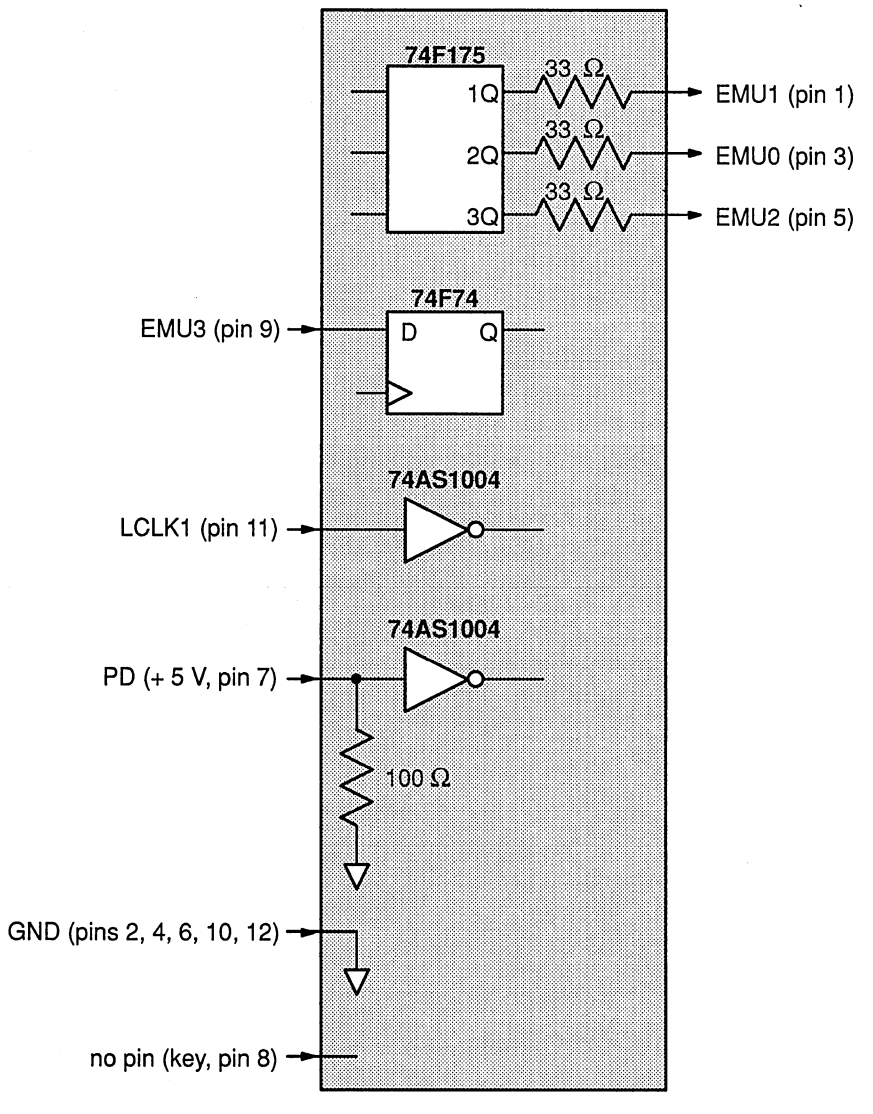


Figure A–5 shows a portion of logic in the emulator pod. Note that 33-Ω resistors are added to EMU0, EMU1, and EMU2; this minimizes cable reflections.

Figure A–5. Emulator Pod Interface



A.5 Design Considerations

When designing a TMS34020 target system, please observe these hardware and software emulation constraints. Portions of these design considerations are advanced information and may not apply to all Texas Instruments emulators.

- **Reset and interrupts.** When an emulator is active, the TMS34020 will service reset and interrupts only if the emulator is in an execution mode. The target system must provide a reset to the TMS34020 before the emulator is activated.

- **Host/emulation coordination.** If the emulator has stopped execution of the TMS34020 (program execution is halted), the TMS34020 will continue to respond to host port accesses. If TMS34020 program execution is required to provide a response to a host access, the host could hang or timeout. Also, functions such as reset, interrupts, NMI, and HLT will not take effect until the emulator is placed back in an execution mode; this could also hang the host application if a response is required. Emulators and host applications typically use timeouts to keep from hanging if a TMS34020 function is not performed properly. If both the emulator and host are accessing the TMS34020 memory space at the same time, false timeouts could occur in both the emulator and the host.

Note:

Both the host and emulator can access the same memory space at effectively the same time. Thus, the emulator's memory display could be inaccurate if the host is modifying a memory location within the display range.

To minimize these conflicts, the host can use 3 bits within HSTCTLL to grant access of the TMS34020 to the emulator. These bits are:

- EMR (emulator request),
- EMG (emulator grant), and
- EMIEN (emulator interrupt enable).

The emulator sets EMR when the emulator requires access to the device. If EMIEN is set, a host interrupt is generated via the $\overline{\text{HINT}}$ pin. When the host sets EMG, the interrupt is cleared and the emulator performs its pending function.

TMS34020 execution will be stopped immediately if an emulation halt condition (such as a breakpoint) is encountered, although emulation access of the TMS34020 will not start until EMG is set. The host processor can use either the host interrupt or the EMR bit to indicate that an emulator halted the TMS34020.

When the emulator no longer requires access to the device, the emulator clears EMR. Once again, this causes a host interrupt if EMIEN is set. The host interrupt is deactivated when the host clears EMG.

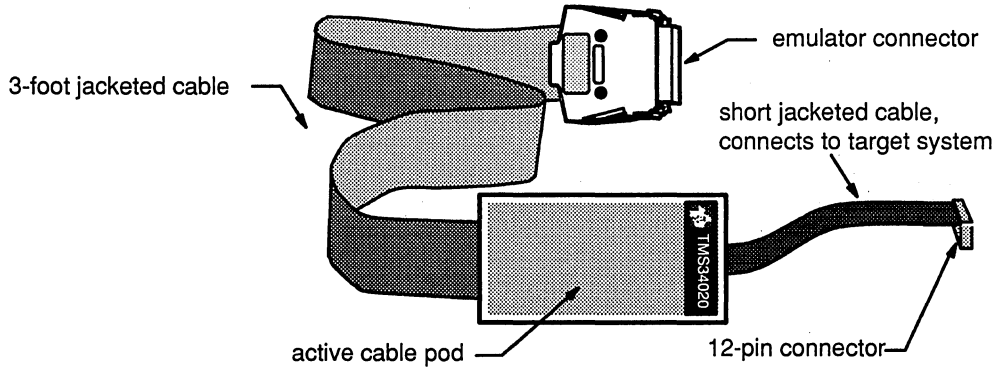
Using this handshake protocol is optional and should be used in applications that are sensitive to emulation access of the TMS34020. Before attempting to integrate this protocol into your system, consult the *TMS34020 XDS Emulator User's Guide* for additional information.

A.6 Mechanical Dimensions

Figure A–6 shows the TMS34020 emulator target cable, which consists of

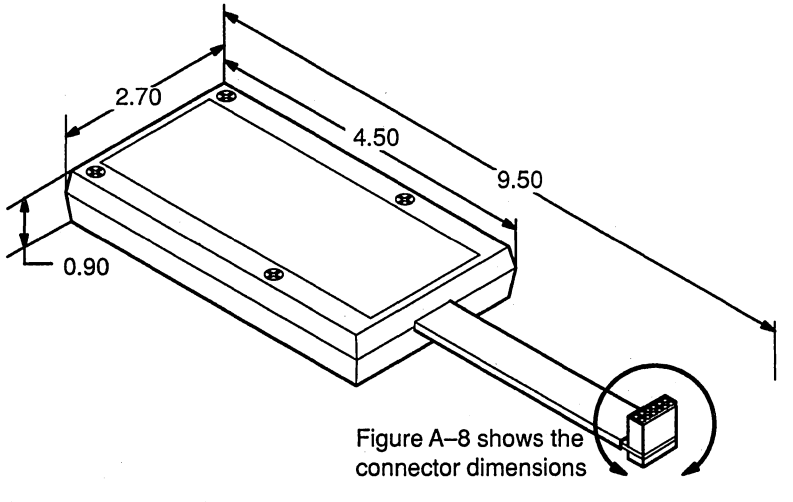
- an emulator connector,
- a 3-foot section of jacketed cable,
- an active cable pod,
- a short section of jacketed cable that connects to the target system, and
- a 12-pin connector that connects to the target system's 12-pin header.

Figure A–6. Target Cable



The overall cable length is approximately 3'10". Figure A–7 shows the mechanical dimensions for the target cable pod. The cable pod box is nonconductive plastic with 4 recessed metal screws.

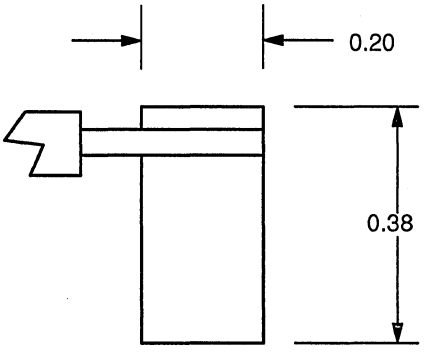
Figure A–7. Pod Dimensions



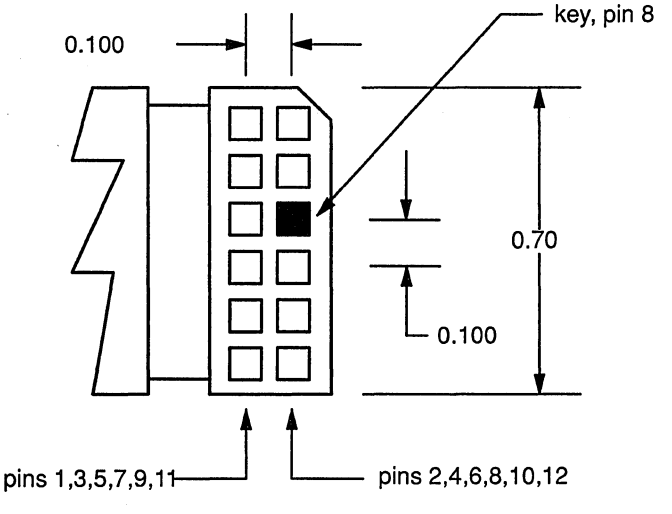
Note: All dimensions are in inches and are nominal dimensions unless otherwise specified.

Figure A–8. 12-Pin Connector Dimensions

(a) Side view



(b) Top view



Notes: 1) All dimensions are in inches and are nominal dimensions unless otherwise specified.

2) Pin-to-pin spacing on the connector is 0.100 inches in both the X and Y planes.

Appendix B

Glossary

A

address/status subcycle: First part of a local-memory cycle, sometimes referred to as *row-address time.*

aliasing: Stairstep effect on a raster display of a line or arc segment.

$\overline{\text{ALTCH}}$: Address latch signal. You can use the high-to-low transition of $\overline{\text{ALTCH}}$ to capture the address and status present on the LAD bus.

antialiasing: Method for reducing the severity of aliasing effects by adjusting the intensity of a pixel according the pixel's proximity to the line or edge of an object.

B

back porch: Portion of horizontal or vertical blanking that follows the trailing edge of the horizontal- or vertical-sync pulse.

bandwidth: Number of bits per second that can be transferred by a device.

BEN: Big-endian enable (bit 0 of CONFIG register). BEN=0 (default) selects little-endian addressing mode; BEN=1 selects big-endian addressing mode.

big-endian: Addressing mode in which the "big" or most significant end of an address (bit 31) points to the least significant end (bit 0) of a word of data.

binary array: 2-dimensional bitmap in which each pixel is represented as a single bit (a 0 or a 1).

bitblt: Bit-aligned block transfer. Transfer of a rectangular array of pixel information from one location in a bitmap to another.

bitmap: 1. Digital representation of an image in which bits are mapped to pixels. 2. Block of memory used to hold raster images in a device-specific format.

bit plane: Hardware used as a storage medium for a bitmap.

black level: Amplitude of the composite signal at which the beam of the picture tube is extinguished (becomes black) to blank retrace of the beam. This level is established at 75% of the signal amplitude.

blanking signals: Pulses that extinguish the scanning beam during horizontal or vertical retrace periods.

breakpoint: Point within a routine at which the routine may be interrupted by external intervention.

BSFLTD: Bus-fault data registers (32-bit I/O register, address C000 0320h). The TMS34020's memory controller saves the LAD data into BSFLTD when a bus fault occurs on a CPU-initiated memory access.

BSFLTDL: 16 LSBs of BSFLTD, accessed at address C000 320h.

BSFLTDH: 16 MSBs of BSFLTD, accessed at address C000 330h.

BSFLTST: Bus-fault status register (16-bit I/O register, address C000 02D0h). The TMS34020's memory controller saves its state in BSFLTST before it signals that a bus fault occurred.

BUSFLT: Bus fault signal. External logic asserts BUSFLT to indicate that a fault occurred on the current bus cycle.

C

cache memory: A fast, on-chip memory.

cache hit: The cache contains the requested instruction word.

cache miss: The cache does not contain the requested instruction word.

CAD: Computer-aided design.

CAMD: Column-address mode. Shifts the column address on the RCA bus to allow mixing of DRAM and VRAM address matrices.

$\overline{\text{CAS}}$: Column-address strobes ($\overline{\text{CAS}}$0—$\overline{\text{CAS}}$3). Drive the $\overline{\text{CAS}}$ inputs of DRAMs and VRAMs.

CBP: Configuration byte protect (bit 4 of CONFIG register). CBP=0 is the default; CBP=1 write-protects the LSbyte of CONFIG until a reset occurs.

CD: Cache disable (bit 15 of CONTROL register). CD=0 (default) enables cache operation; CD=1 forces the TMS34020 to ignore the contents of the cache and to fetch instructions from memory.

CF: Cache flush (bit 14 of HSTCTLH register). Setting CF to 1 flushes and disables the cache. Normal cache operation resumes when CF is cleared to 0.

clipping: Removing parts of display elements that lie outside a defined boundary (the boundary is usually a window or a viewport).

COLOR0: Background color register (B8). Identifies the replacement color for 0-value pixels in a source array.

COLOR1: Foreground color register (B9). Identifies the replacement color for pixels that will be altered in the destination array.

column-address time: See *data subcycle*.

composite video: Color-picture signal plus all blanking and sync signals. The signals include luminance and chrominance signals, vertical- and horizontal-sync pulses, vertical- and horizontal-blanking pulses, and the color-burst signal.

CONFIG: Configuration register (16-bit I/O register, address C000 01A0h). Contains fields that selectively enable/disable various aspects of system configuration.

CONTROL: Memory control register (16-bit I/O register, addresses C000 00B0h and C000 0190h). Controls various aspects of CPU activity.

CONVDP: Destination pitch conversion factor register (16-bit I/O register, address C000 0140h). Contains a control parameter used for converting an XY destination address to a linear address.

CONVMP: Mask pitch conversion factor register (16-bit I/O register, address C000 0180h). Contains a control parameter used for converting an XY mask address to a linear address.

CONVSP: Source pitch conversion factor register (16-bit I/O register, address C000 0130h). Contains a control parameter used for converting an XY source address to a linear address.

coprocessor: An additional processor in a system; extends the functionality of the main processor. For example, the TMS34082 is a coprocessor for the TMS34020; in a TMS34020 system, the TMS34082 adds floating-point capabilities to the TMS34020's functions.

CSD: Composite-sync direction (bit 2 of DPYCTL register). When the $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ pin is configured as CSYNC (CVD=0), CSD determines if $\overline{\text{CSYNC}}$ is configured as in input (CSD=0) or an output (CSD=1).

CST: CPU shift-register transfer enable (bit 11 of DPYCTL register). When CST=1, the TMS34020 converts pixel accesses into VRAM shift-register transfer cycles.

CVD: Composite video disable (bit 3 of DPYCTL register). Controls the functions of the $\overline{\text{CSYNC}}$/$\overline{\text{HBLNK}}$ and $\overline{\text{CBLNK}}$/$\overline{\text{VBLNK}}$ pins. CVD=0 selects $\overline{\text{CSYNC}}$ and $\overline{\text{CBLNK}}$; CVD=1 selects $\overline{\text{HBLNK}}$ and $\overline{\text{VBLNK}}$.

D

DAC: Digital-to-analog converter.

DADDR: Destination address register (B2). Contains the destination array address for graphics instructions.

data subcycle: Second part of a local-memory cycle, sometimes referred to as *column-address time*.

DDIN: Data bus direction input-enable signal. Drives the active-high input enables on bidirectional transceivers.

$\overline{\text{DDOUT}}$: Data bus direction output-enable signal. Drives the active-low output enables on bidirectional transceivers.

DGIS: Direct graphics interface standard.

DIE: Display interrupt enable (bit 10 of INTENB register). Setting DIE to 1 enables the display interrupt.

DIP: Display interrupt pending (bit 10 of INTPEND register). DIP is set to 1 when a display interrupt is requested.

DINC: Display increment registers (32-bit I/O register, address C000 0240h). Contains the increment value for the DPYNX register.

DINCL: 16 LSBs of DINC, accessed at address C000 0240h.

DINCH: 16 MSBs of DINC, accessed at address C000 0250h.

display area: Rectangular portion of the physical display screen in which information is visibly displayed; does not include the border area.

display element: Basic graphic element that can be used to construct a display image.

display memory: Area of memory used to hold the graphics image output to the video monitor.

display pitch: Difference in memory addresses between two vertically adjacent positions on the screen.

dotclock: Clock that cycles the rate at which video data is output to a CRT.

DPTCH: Destination pitch register (B3). Defines the linear difference between starting addresses of adjacent rows in a destination array.

DPYADR: Display address register. Provides compatibility with the TMS34010.

DPYCTL: Display control register (16-bit I/O register, address C000 0080h). Controls video timing and VRAM serial-register transfers.

DPYINT: Display interrupt register (16-bit I/O register, address C000 00A0h). Identifies the next scan line (in some circumstances, the next *half* scan line) at which a display interrupt can be requested.

DPYNX: Display next address registers (32-bit I/O register, address C000 0220h). Contains a 32-bit address that is output during a screen-refresh cycle.

DPYNXL: 16 LSBs of DPYNX, accessed at address C000 0220h.

DPYNXH: 16 MSBs of DPYNX, accessed at address C000 0230h.

DPYMSK: Display mask register (16-bit I/O register, address C000 02E0h). When midline reload screen refreshes are enabled, DPYMSK determines which bits of DPYNX & DPYST correspond to the tap-point portion of the address output during screen-refresh cycles.

DPYST: Display start address registers (32-bit I/O register, address C000 0200h). Contains a 32-bit address that points to the pixel at the left of the 1st line displayed on the screen.

DPYSTL: 16 LSBs of DPYST, accessed at address C000 0200h.

DPYSTH: 16 MSBs of DPYST, accessed at address C000 0210h.

DPYSTRT: Display start address register. Provides compatibility with the TMS34010.

DPYTAP: Display tap-point address register. Provides compatibility with the TMS34010.

DQ: Data in/data out pin for a VRAM.

DRAM: Dynamic RAM.

DRAM refresh: Maintenance of data stored in dynamic RAMs. Data are stored in DRAMs as electrical charges across a grid of capacitive cells. The charge stored in a cell will leak off over time unless the data is refreshed.

DYDX: Delta Y/delta X register (B7). Defines the X and Y dimensions of a rectangular destination array.

E

EMIEN: Emulator host-interrupt enable (bit 12 of HSTCTLL register). The value of EMIEN determines if EMG XOR EMR asserts $\overline{\text{HINT}}$ active low (EMIEN=1) or not (EMIEN=0).

EMG: Emulator handshake (bit 11 of HSTCTLL register). In an emulation system, the host sets EMG to 1 to gran the emulator access to TMS34020 memory.

EMR: Emulator handshake (bit 10 of HSTCTLL register). In an emulation system, the emulator sets EMR to 1 to request access to TMS34020 memory.

ENV: Enable video (bit 15 of DPYCTL register). ENV enables (ENV=1) or disables (ENV=0) the video screen.

F

field: 1. Group of contiguous bits in a register or memory location, dedicated to a particular function or representing a single entity. 2. Software-configurable data type supported by the TMS34010 and TMS34020; the field length can be programmed to be any value in the range of 1 to 32 bits.

fill: Solid coloring or shading of a display surface, often achieved as a pattern of horizontal segments.

frame: 1. Time required to refresh an entire screen. 2. Screen image output during a single vertical sweep.

frame buffer: Portion of memory used to buffer raster data to be output to a CRT. Frame buffer contents are often referred to as the bitmap of the display and contain the logical pixels corresponding to the points on the monitor screen.

front porch: Portion of a vertical- or horizontal-blanking pulse that precedes the leading edge of the vertical- or horizontal-sync pulse.

G

GI: Bus grant input. External bus arbitration logic pulls $\overline{GI}$ low to enable the TMS34020 to gain access to the local-memory bus.

GKS: Graphics kernel system. Application programmer's standard interface to a graphics display.

gray scale: Scale of light intensities from black to white.

GSP: Graphics system processor. A single-chip device embodying all the processing power and control capabilities necessary to manage a high-performance bitmapped graphics system. The TMS34010 and TMS34020 are GSPs.

H

HA: Host address input bus (HA5—HA31). A host processor requests an address over these lines.

HACK: Halt acknowledge (bit 4 of HSTCTLH register). Setting the HLT bit halts TMS34020 execution at the next interruptible instruction boundary; the TMS34020 sets HACK when the halt actually takes place.

HBFI: Host-bus-fault interrupt (bit 14 of HSTCTLL register). The TMS34020 sets HBFI to 1 if a bus fault occurs on a host access.

HBREN: Host-bus-fault/retry-interrupt (bit 15 of HSTCTLL register). If HBREN=1, the TMS34020 interrupts the host when a retry or bus fault occurs.

HBS: Host byte select-bus (HBS0—HBS3). Identify the bytes to be selected within a specific word.

HCOUNT: Horizontal count register (16-bit I/O register, address C000 01D0h). HCOUNT counts the number of VCLK periods per horizontal scan line.

$\overline{\text{HCS}}$: Host chip-select signal. A host drives $\overline{\text{HCS}}$ low to latch the current address and byte-select requests.

HDST: Host data strobe signal.

HEBLNK: Horizontal end blank register (16-bit I/O register, address C000 0030h). HEBLNK identifies the endpoint for the horizontal blanking interval.

HESERR: Horizontal end serration register (16-bit I/O register, address C000 0270h). HESERR determines the endpoint for the composite-sync pulse during the serration region of vertical blanking.

HESYNC: Horizontal end sync register (16-bit I/O register, address C000 0010h). HESYNC identifies the endpoint for horizontal sync.

HIE: Host interrupt enable (bit 9 of INTENB register). Setting HIE to 1 enables the host interrupt.

high impedance: The third state of a three-state output driver, in which the output is driven neither high or low but behaves as an open connection.

HIP: Host interrupt pending (bit 9 of INTPEND register). HIP is set to 1 when a host interrupt is requested.

HINC: Host increment (bit 12 of HSTCTLH register). Setting HINC to 1 enables the TMS34020 to compare the fetched address to the address requested by a host processor, to increment the current address, and to prefetch the contents of the next address.

$\overline{\text{HINT}}$: Host interrupt signal.

HLB0, HLB1: Host last byte (bits 5&6 of HSTCTLH register). The HLB code tells the TMS34020 which byte of a 32-bit word that a host processor will access last. The TMS34020 uses this information to determine the correct time to prefetch the next word.

HLT: Halt TMS34020 program execution (bit 15 of HSTCTLH register). Setting HLT to 1 suspends TMS34020 instruction processing at the next instruction boundary.

$\overline{\text{HOE}}$: Host output-enable signal.

hold signal: Signal capable of controlling a processor bus; sent to a bus arbiter to request bus control. Typically, the arbiter grants the request by sending a hold-acknowledge signal to the requestor.

horizontal back porch: Portion of horizontal blanking that follows the trailing edge of the horizontal-sync pulse.

horizontal-blanking interval: Time during which the display is blanked to cover the horizontal retracing of the electron beam on a screen.

horizontal front porch: Portion of a horizontal-blanking pulse that precedes the leading edge of the horizontal-sync pulse.

horizontal sync: Synchronization signal that enables horizontal retrace of the electron beam on a screen.

host address bus: Lines used by a host processor to identify the address of a TMS34020 local-memory location.

host processor: Main processor in a system.

HPFW: Host prefetch-after-write enable (bit 10 of HSTCTLH register). When host prefetches are enabled (HINC=1), the value of HPFW determines if the TMS34020 performs prefetches after reads (HPFW=0) or after writes (HPFW=1).

HRDY: Host ready signal. Driven high when the TMS34020 is ready to complete a host-initiated access.

$\overline{\text{HREAD}}$: Host read strobe. Driven low during a host's read request.

HRYI: Host-retry interrupt (bit 13 of HSTCTLL register). The TMS34020 sets HRYI to 1 if it retries a host access.

HSBLNK: Horizontal start blank register (16-bit I/O register, address C000 0050h). HSBLNK identifies the startpoint for the horizontal blanking interval.

HSD: Horizontal-sync direction (bit 0 of DPYCTL register). Determines if $\overline{\text{HSYNC}}$ is configured as an input (HSD=0) or an output (HSD=1).

HSTADRL: Host address register. Provides compatibility with the TMS34010.

HSTADRH: Host address register. Provides compatibility with the TMS34010.

HSTCTLH: Host control I/O register, high word (16-bit I/O register, address C000 0100h). Controls aspects of host-interface communications.

HSTCTLL: Host control I/O register, low word (16-bit I/O register, address C000 00F0h). Controls aspects of host-interface communications.

HSTDATA: Host data I/O register. Provides compatibility with the TMS34010.

HTOTAL: Horizontal total register (16-bit I/O register, address C000 0070h). Number of VCLK periods per horizontal scan line; defines the startpoint for the horizontal sync pulse.

$\overline{\text{HWRITE}}$: Host write strobe. Driven low during a host's write request.

I

interlaced video: Video system in which odd-numbered scan lines (odd field) are interlaced with even-numbered scan lines (even field). The odd and even fields constitute one frame. In effect, the number of transmitted pictures is doubled; this reduces flicker.

IHOST: Internal host interface address registers (4 32-bit registers: IHOST1, address C000 0308h; IHOST2, address C000 03A0h; IHOST3, address C000 03C0h; IHOST4, address C000 03E0h). The TMS34020 uses these registers for storing information provided by the host.

implied operand: A register value that must be supplied for an instruction to execute properly. The B-file registers and several of the I/O registers serve as implied operands for the TMS34020's graphics instructions.

INTENB: Interrupt enable register (16-bit I/O register, address C000 0110h). Selective enables /disables external interrupts 1 and 2, the host interrupt, the display interrupt, and the window violation interrupt.

INTPEND: Interrupt pending register (16-bit I/O register, address C000 0120h). Identifies the pending/not pending status of external interrupts 1 and 2, the host interrupt, the display interrupt, and the window violation interrupt.

INTIN: Interrupt-in (bit 3 of HSTCTLL register).

INTOUT: Interrupt-out (bit 7 of HSTCTLL register).

K

K: 1) 1024. 2) Approximately 1000. 3) A 5-bit constant for a TMS34020 instruction.

Kbyte: Approximately 1000 bytes.

L

LAD bus: 32-bit local address/data multiplexed bus (LAD0—LAD31).

little-endian: An addressing mode in which the "little" or least significant end of an address (bit 0) points to the least significant end (bit 0) of a word of data.

long word: 32-bit word.

look-up table: Table used during scan conversion of a digital image that converts color-map addresses into the actual color values displayed.

LRDY: Local ready signal. External circuitry drives LRDY low to stop the TMS34020 from completing a local-memory cycle.

LRU: Least recently used (cache-replacement algorithm). When a cache miss occurs, this algorithm selects the cache segment that will be overwritten, based on the likelihood that the data in the discarded segment will not be needed again for some time. The LRU algorithm selects the segment that was used least recently.

LSB: Least significant bit.

LSbyte: Least significant byte.

LSW: Least significant word.

M

mask: Pattern used to control retention or elimination of portions of another pattern.

Mbyte: Megabyte.

memory map: Map of memory space, partitioned into functional blocks.

MPTCH: Mask pitch register (B11). Defines the linear difference between starting addresses of adjacent rows in a mask array.

MSB: Most significant bit.

MSbyte: Most significant byte.

MSGIN: Message-in (bits 0—2 of HSTCTLL register).

MSGOUT: Message-out (bits 4—6 of HSTCTLL register).

MSW: Most significant word.

N

NIL: Noninterlaced video enable (bit 14 of DPYCTL register). The value of NIL selects interlaced video timing (NIL=0) or noninterlaced video timing (NIL=1).

NMI: Nonmaskable interrupt (bit 8 of HSTCTLH register). A host processor sets NMI to send a nonmaskable interrupt to the TMS34020.

NMIM: Nonmaskable interrupt mode (bit 9 of HSTCTLH register). If NMIM=0, the TMS34020 saves the PC and ST contents on the stack before executing a nonmaskable interrupt routine. If NMIM=1, the TMS34020 discards the PC and ST contents before executing the NMI routine.

nonmaskable interrupt: Interrupt request that cannot be disabled.

NTSC: National television system committee. Group representing a wide range of interests in the television broadcasting and video industry; NTSC is instrumental in developing graphics and video standards.

O

OFFSET: XY-address offset register (B4). OFFSET contains the linear address of the 1st pixel in the XY-coordinate address space.

operand: Any one of the quantities entering into or arising out of an operation.

origin: Zero intersection of X and Y axes from which all points are calculated.

P

palette: Digital look-up table used in a graphics display for translating data from the bitmap into the pixel values to be shown on the screen.

pan: Apparent horizontal or vertical movement of a graphics screen or window over an image contained in a frame buffer that is too large to be completely displayed in a single static picture.

PATTERN: Fill-pattern register (B13).

PBH: PIXBLT horizontal direction (bit 8 of CONTROL register). PBH=0 (default) selects left-to-right pixel processing; PBH=1 selects right-to-left processing.

PBV: PIXBLT vertical direction (bit 9 of CONTROL register). PBV=0 (default) selects top-to-bottom pixel processing; PBV=1 selects bottom-to-top processing.

pending: Requested but not yet performed. For example, a pending interrupt is an interrupt that has been requested but has not yet been serviced.

PGA: Pin grid array (type of chip package).

$\overline{\text{PGMD}}$: Page-mode signal. Memory decode logic asserts $\overline{\text{PGMD}}$ low if the currently addressed memory supports page-mode accesses.

phase: The time interval for each clock period in a system is divided into phases; one phase corresponds to the time when the clock signal is high, the other to the time that the signal is low.

PHIGS: Programmer's hierarchical interactive graphics standard.

pipelining: Design technique for reducing the effective propagation delay per operation by partitioning the operation into a series of stages, each of which performs a portion of the operation. A series of data is typically clocked through the pipeline in sequential fashion, advancing one stage per clock period.

pitch: Difference in starting addresses of two adjacent rows of pixels in a 2-dimensional pixel array.

pixel: Picture element. 1. Smallest controllable point of light on a display screen. 2. In a bitmapped display, the logical data structure that contains the attributes to be shown at the corresponding physical pixel position on a display screen.

pixel-processing option: Boolean or arithmetic operation for combining two pixel values (source and destination); defined by PPOP⟦CONTROL⟧.

PIXBLT: Pixel-block transfer. Pixel-array operation in which each pixel is represented by one or more bits. PIXBLTs are a superset of bitblts and include commonly-used Boolean functions as well as integer arithmetic and multi-bit operations.

plane: (also bit plane or color plane) Bitmap layer in a multiple-bit-per-pixel display device. If the pixel size is n bits and the bits in each pixel are numbered 0 to n–1, plane 0 is made up of 0-numbered bits in all the pixels, and plane n–1 is made up of n–1-numbered bits in all the pixels. A layered graphics display allows planes or groups of planes to be manipulated independently of the other planes.

PMASK: Plane mask registers (32-bit I/O register, address C000 0160h). PMASK contains a mask of 0s and 1s; the 1s represent protected destination bits, and the 0s represent modifiable destination bits.

PMASKL: 16 LSBs of PMASK, accessed at address C000 0160h.

PMASKH: 16 MSBs of PMASK, accessed at address C000 0170h.

PPOP: Pixel-processing operation (bits 10—14 of CONTROL register). Selects a method for combining source and destination pixels. You can choose from 16 Boolean and 6 arithmetic operations; the default operation is S→D (source pixels replace destination pixels).

propagation delay: Time required for a change in logic level at an input to a circuit to be translated into a resulting change at an output.

protocol: Set of rules, formats, and procedures governing the exchange of information.

pseudo-op: (pseudo-operation) An operation which is not part of the computer's operation repertoire as realized by hardware; hence, an extension of the set of machine operations.

PSIZE: Pixel size register (16-bit I/O register, address C000 0150h). Defines the current pixel size as 1, 2, 4, 8, 16, or 32 bits.

pulse width: Time interval between specified reference points on the leading and trailing edges of a pulse waveform.

Q

QFP: Quad flat package (type of chip package).

quarter phase: One-fourth of a local-memory cycle.

R

$\overline{R0}$, $\overline{R1}$: Bus request and control signals. These signals identify the type of request for use of the bus in a multiprocessor system.

RAM: Random access memory. A memory from which all information can be obtained with approximately the same time delay by choosing an address randomly and without first searching through a vast amount of irrelevant data.

$\overline{RAS}$: Row-address strobe. Drives the $\overline{RAS}$ inputs of DRAMs and VRAMs.

raster: Rectangular grid of picture elements whose intensity levels are manipulated to represent images. In a bitmapped display, the bits within the frame buffer are mapped to the raster pattern of a display screen.

raster graphics: Computed graphics in which a display image is composed of a pixel array arranged in rows and columns.

raster-op: Arithmetic or logical combination that takes place during the transfer of a pixel array from one location to another.

raster scan: Grid pattern traced by the electron beam on a display screen.

RCA: Multiplexed row-/column-address bus (RCA0—RCA12). At the beginning of a memory-access cycle, identifies the row address for DRAMs; later in the cycle, the bus identifies the column address.

RCM0, RCM1: RCA0—RCA12 row address configuration (bits 1&2 of CONFIG register). Determines which bits of the logical address are output on RCA0—RCA12 at row-address time.

ready signal: Signal from a memory or memory-mapped peripheral that informs the processor when a memory cycle is about to complete. Slower memories and peripherals must extend the length of the memory cycle by negating the ready signal (in other words, by sending the processor a "not ready" signal) until the cycle can be completed.

REFADR: Refresh pseudo-address register (16-bit I/O register, address C000 01F0h). Contains the address output during DRAM-refresh cycles.

refresh: Method of restoring the charge capacitance to a memory device (such as a DRAM or VRAM) or of restoring memory contents.

request strobe: Any control signal that begins or ends a read request or a write request.

reset: Restore to normal action and initial conditions.

resolution: Number of visible, distinguishable units in the device coordinate space.

retrace: Line traced by the scanning beam(s) of a display screen as it travels from the end of one horizontal (or vertical) line or field to the beginning of the next horizontal (or vertical) line or field.

RGB monitor: Red-green-blue monitor. Type of monitor capable of displaying colors; has separate inputs for the three signals that drive the red, green, and blue guns of a display.

relative coordinates: Location of a point relative to the location of another point.

ROM: Read-only memory.

rotate: Transform an item or display by revolving it around an axis or center point.

row-address time: See *address/status subcycle.*

RR0—RR2: Refresh rate (bits 10—12 of CONFIG register). Determines the frequency of DRAM refreshes.

RST: Reset (bit 7 of HSTCTLH register). Setting this bit has the same effect as asserting $\overline{\text{RESET}}$ low; however, *only* the TMS34020 is reset (other devices in the system are not affected).

S

SADDR: Source address register (B0). Contains the source array address for graphics instructions.

SAM: Serial access memory or serial data register.

scale: Size change made by multiplying or dividing coordinate dimensions by a scale factor (a constant value).

scan line: Horizontal line traced across a display screen by the electron beam in a monitor or similar raster-scan device.

SCOUNT: Shift clock counter register (16-bit I/O register, address C000 02C0h). During horizontal blanking, SCOUNT is loaded with the right-justified tap-point value and is then incremented once on the rising edge of each SCLK pulse.

screen refresh: Operation of dumping the contents of the frame buffer to a CRT monitor in synchronization with the movement of the electron beam.

scrolling: Moving a display vertically or horizontally.

serial register transfer: Transfer between the RAM storage and internal serial register in a VRAM.

SETHCNT: Set horizontal count register (16-bit I/O register, address C000 0310h). During external horizontal or composite video, SETHCNT is loaded into HCOUNT when $\overline{\text{HSYNC}}$ or $\overline{\text{CSYNC}}$ is pulsed.

setup time: Minimum amount of time that valid data must be present at an input before the device is clocked; ensures proper data acceptance.

SETVCNT: Set vertical count register (16-bit I/O register, address C000 0300h). During external horizontal or composite video, SETVCNT is loaded into VCOUNT when $\overline{\text{VSYNC}}$ or $\overline{\text{CSYNC}}$ is pulsed.

SF: Special-function signal that drives a VRAM's DSF pin.

$\overline{\text{SIZE16}}$: Bus size signal. Memory decode logic may pull $\overline{\text{SIZE16}}$ low if the currently addressed memory or port supports only 16-bit transfers.

SPTCH: Source pitch register (B1). Defines the linear difference between starting addresses of adjacent rows in a source array.

SRAM: Static RAM.

SRE: Screen-refresh enable (bit 12 of DPYCTL register). Setting SRE to 1 when video is enabled (ENV) enables screen-refresh cycles.

SRINC: Screen-refresh address increment value (bits 5—31 of DINC registers). Defines the amount by which the address in SRNX is incremented after a screen-refresh cycle.

SRNX: Next screen-refresh address (bits 5—31 of DPYNX registers). Represents the long-word address that is output during a screen-refresh cycle.

SRST: Screen-refresh start address (bits 5—31 of DPYST registers). Contains the address of the pixel at the left of the 1st line displayed on the screen.

SSA: Cache segment start address register.

SSV: Split-shift-register midline-reload enable (bit 6 of DPYCTL register). Determines whether split-shift-register midline reload is disabled (SSV=0) or enabled (SSV=1 and SRE=1).

stairstepping: Visual effect in bitmapped display devices; produces images by brightening or dimming individual pixels in a pixel array. Also called aliasing.

strobe: Any control signal that begins or ends a memory access.

subsegment: Block of 4 long words in a cache segment. Each of the 4 cache segments contains 8 subsegments, for a total of 32 long words per segment.

T

T: Pixel transparency (bit 5 of CONTROL register). T=1 enables transparency; T=0 (default) disables transparency.

tap point: Column address provided to a VRAM during a memory-to-serial-register cycle. The column address specifies the point at which the shift register is to be tapped; in other words, which cell of the serial register is to be connected to the VRAM's serial output.

TM: Transparency mode (bits 0—2 of CONTROL register). Selects the transparency mode for pixel operations.

trace: Line of the graphics display.

transformation: Geometric alteration of a graphics display, such as scaling, translation, or rotation.

transparency: Pixel attribute that renders a source pixel invisible so that portions of the destination array show through portions of the source array.

$\overline{\text{TR}}/\overline{\text{QE}}$: Transfer/output enable signal. Drives the $\overline{\text{TR}}/\overline{\text{QE}}$ input of VRAMs.

V

VCE: Video capture enable (bit 7 of DPYCTL register). Selects memory-to-register screen-refresh cycles (VCE=0) or register-to memory screen-refresh cycles (VCE=1).

VCOUNT: Vertical count register (16-bit I/O register, address C000 01C0h). VCOUNT counts the horizontal scan lines in the video display.

VEBLNK: Vertical end blanking register (16-bit I/O register, address C000 0020h). VEBLNK defines the endpoint for the vertical blanking interval.

VEN: VRAM internal register load enable (bit 8 of CONFIG register). VEN=1 enables the TMS34020 to use VRAMs with internal write-mask and color registers; VEN=0 (default) prohibits this.

vertical back porch: Portion of vertical blanking that follows the trailing edge of the vertical-sync pulse.

vertical-blanking interval: Time during which the display is blanked to cover the vertical retracing of an electron beam.

vertical-blanking pulse: Positive or negative pulse developed during vertical retrace, appearing at the end of each field. Used to blank out scanning lines during the vertical-retrace interval.

vertical front porch: Portion of a vertical-blanking pulse that precedes the leading edge of the vertical-sync pulse.

vertical sync: Synchronization signal that enables vertical retrace of the electron beam of a display screen.

VESYNC: Vertical end sync register (16-bit I/O register, address C000 0000h). VESYNC defines the endpoint of the vertical-sync pulse; in interlaced video, it also defines the endpoint of the 2^{nd} equalization region.

VRAM: Video RAM. A dual-ported memory device for computer graphics applications, containing two interfaces: one that allows a processor to read/write data from an internal memory array, a second that provides a serial stream of screen-refresh data to a display screen.

VSBLNK: Vertical start blank register (16-bit I/O register, address C000 0040h). VSBLNK defines the startpoint for the vertical blanking interval.

VSD: Vertical sync direction (bit 1 of DPYCTL register). Determines if $\overline{\text{VSYNC}}$ is configured as in input (VSD=0) or an output (VSD=1).

VTOTAL: Vertical total register (16-bit I/O register, address C000 0060h). Number of horizontal scan lines in the display; defines the startpoint for the vertical-sync pulse.

W

W: Window checking (bits 6&7 of CONTROL register). Selects the action that the TMS34020 takes when a pixel operation would write a pixel inside or outside defined window limits.

wait state: Clock period inserted into a memory cycle in order to permit accesses of slower memories and slower memory-mapped devices.

$\overline{\text{WE}}$: Write enable signal. Drives the $\overline{\text{WE}}$ inputs of DRAMs and VRAMs.

WEND: Window ending address register (B6). WEND contains the XY address of the most significant pixel of the clipping window.

WSTART: Window starting address register (B5). WSTART contains the XY address of the least significant pixel of the clipping window.

window: Defined rectangular area of a virtual space on a display screen.

window checking: Checking a pixel's address to see if it lies inside or outside the boundaries of a defined window.

WVE: Window-violation interrupt enable (bit 11 of INTENB register). Setting WVE to 1 enables the window-violation interrupt.

WVP: Window-violation interrupt pending (bit 11 of INTPEND register). WVP is set to 1 when a window-violation interrupt is requested.

X

X1E: External interrupt 1 enable (bit 1 of INTENB register). Setting X1E to 1 enables external interrupt 1.

X2E: External interrupt 2 enable (bit 2 of INTENB register). Setting X2E to 1 enables external interrupt 2.

X1P: External interrupt 1 pending (bit 1 of INTPEND register). X1P is set to 1 when an external interrupt 1 is requested.

X2P: External interrupt 2 pending (bit 2 of INTPEND register). X2P is set to 1 when an external interrupt 2 is requested.

Y

YZCNT: Y-zoom count (bits 0—4 of DPYNX registers). Determines when the address in SRNX can be incremented.

YZINC: Y-zoom increment value (bits 0—4 of DINC registers). This value provides the increment value for the Y-zoom feature; valid values include 0, 2, 4, 8, 16, and 32.

Y-zoom: TMS34020 feature that aids in display magnification.

Z

zoom: Scaling a display (or display item) so it is magnified or reduced on the screen.

TEXAS INSTRUMENTS

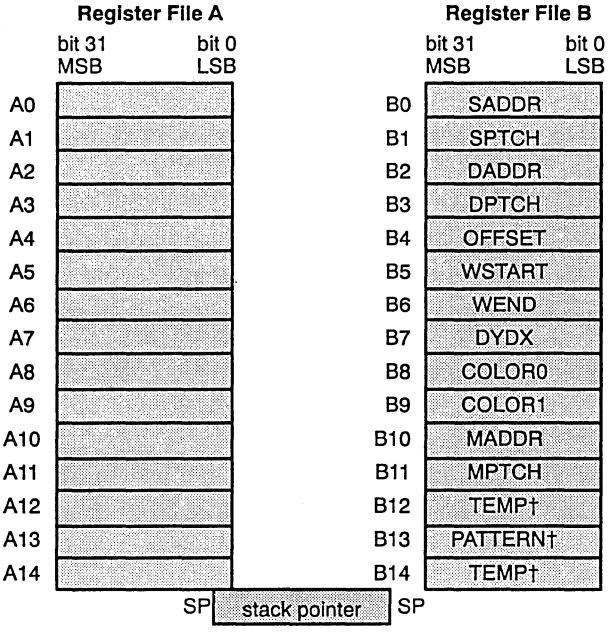
TMS34020 Reference Card

Phone Numbers

TI Customer Response Center (CRC) Hotline: (800) 232–3200

Graphics Hotline: (713) 274–2340

General-Purpose Register Files



† The line instructions use these registers for a different purpose. Some graphics instructions use these registers as temporary registers.

Initial State Following Reset

Immediately following reset,

- All I/O registers are cleared to 0000h. (Possible exceptions are HLT[[HSTCTLH]], REFADR, and SCOUNT).
- General-purpose register files A and B are uninitialized.
- The ST is set to 0000 0010h.
- The PC is uninitialized.
- The cache SSA registers are uninitialized.
- The cache LRU stack is set to the sequence 0, 1, 2, 3.
- All cache P flags are cleared.
- The DRAM refresh-pending counter is set to 9.

I/O Registers

Register	Offset
BSFLTDH	0330h
BSFLTDL	0320h
BSFLTST	02D0h
CONFIG	01A0h
CONTROL	00B0h or 0190h
CONVDP	0140h
CONVMP	0180h
CONVSP	0130h
DINCH	0250h
DINCL	0240h
DPYADR	01E0h
DPYCTL	0080h
DPYINT	00A0h
DPYNXH	0230h
DPYNXL	0220h
DPYMSK	02E0h
DPYSTH	0210h
DPYSTL	0200h
DPYSTRT	0090h
DPYTAP	01B0h
HCOUNT	01D0h
HEBLNK	0030h
HESERR	0270h
HESYNC	0010h
HSTADRH	00E0h
HSTADRL	00D0h
HSTCTLH	0100h
HSTCTLL	00F0h
HSTDATA	00C0h
HSBLNK	0050h
HTOTAL	0070h
IHOST	0380h to 03F0h
INTENB	0110h
INTPEND	0120h
PMASKH	0170h
PMASKL	0160h
PSIZE	0150h
REFADR	01F0h
SCOUNT	02C0h
SETHCNT	0310h
SETVCNT	0300h
VCOUNT	01C0h
VEBLNK	0020h
VESYNC	0000h
VSBLNK	0040h
VTOTAL	0060h

Note: Register address = C000 000h + offset.

CONTROL Register (C000 00B0h)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CD	PPOP					PBH	PBV	W		T			TM		

TM 000 transparency on result=0
001 transparency on source=COLOR0
100 transparency on result=0
101 transparency on dest.=COLOR0

T 0 disables trans.
1 enables trans.

W 00 no windowing
01 window hit
10 window miss
11 window clip

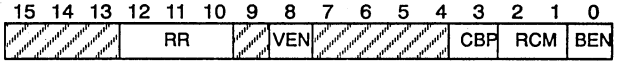
PBV 0 PIXBLT processes top to bottom
1 PIXBLT processes bottom to top

PBH 0 PIXBLT processes left to right
1 PIXBLT processes right to left

PPOP pixel-processing option

CD 0 enables cache
1 disables cache

CONFIG Register (C000 01A0h)



BEN 0 selects little-endian addressing *(default)*
1 selects big-endian addressing

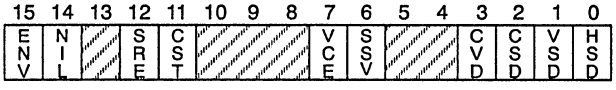
RCM determines which logical address bits are output at row-address time

CBP 1 write-protects CONFIG's LSbyte
0 no write protection

VEN 0 system has special-feature VRAMs
1 system has no special-feature VRAMs

RR DRAM refresh rate

DPYCTL Register (C000 0080h)



HSD 0 $\overline{\text{HSYNC}}$ is an input
1 $\overline{\text{HSYNC}}$ is an output

VSD 0 $\overline{\text{VSYNC}}$ is an input
1 $\overline{\text{VSYNC}}$ is an output

CSD When CVD=0,
0 $\overline{\text{CSYNC}}$ is an input
1 $\overline{\text{CSYNC}}$ is an output

CVD selects $\overline{\text{CYSNC}}$/$\overline{\text{HBLNK}}$
0 selects $\overline{\text{CSYNC}}$
1 selects $\overline{\text{HBLNK}}$

SSV 0 disables midline reload
1 enables midline reload when SRE=1

VCE screen-refresh mode
1 mem-to-reg cycles
0 reg-to-mem cycles

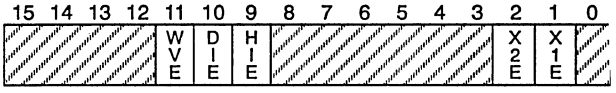
CST 0 normal pixel-access cycles
1 pixel-access cycles become serial-register-transfer cycles

SRE 0 disables automatic screen refresh
1 enables screen refresh when ENV=1

NIL 0 interlaced video
1 noninterlaced video

ENV 0 blanks screen
1 enables display

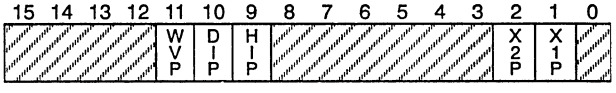
INTENB Register (C000 0110h)



IE status bit must be enabled before these interrupts are enabled

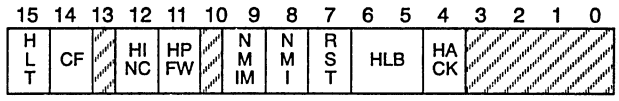
X1E 1 enables int. 1
X2E 1 enables int. 2
HIE 1 enables host int.
DIE 1 enables display int.
WVE 1 enables window-violation int.

INTPEND Register (C000 0120h)



X1P 1 int. 1 pending
X2P 1 int. 2 pending
HIP 1 host int. pending
DIP 1 display int. pending
WVP 1 window-violation int. pending

HSTSTLH Register (C000 0100h)



HACK 0 '34020 is running
1 '34020 is halted

HLB identifies last byte that host will access

RST 0 normal operation
1 reset '34020

NMI 0 no NMI request
1 host requests NMI

NMIM 0 save context when there's an NMI
1 discard context

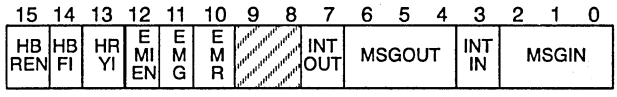
HPFW 0 prefetch after any access
1 prefetch after writes

HINC 0 disables prefetch & autoincrement
1 enables prefetch & autoincrement

CF 0 no effect
1 flush cache

HLT 0 allow '34020 to run
1 halt '34020 instruction execution

HSTSTLL Register (C000 00F0h)



MSGIN message from host to '34020

INTIN 0 no interrupt to '34020
1 host interrupt request to '34020

MSGOUT message from '34020 to host

INTOUT 0 no interrupt to host
1 '34020 interrupt request to host

EMG/EMR 00 no request, no interrupt
01 host request from EMU, interrupt (if enabled)
10 host released by EMU, interrupt (if enabled)
11 host grant to EMU, no interrupt

EMIEN 0 no interrupt to host
1 interrupt to host

HRYI 0 host access not retried
1 host access retried

HBFI 0 host access not faulted
1 host access faulted

HBREN If HRYI or HBFI is set,
0 no interrupt to host
1 interrupt to host

LAD Bus Status Codes

Code	Bus Status	Type
0000	Coprocessor cycle	misc.
0001	Emulator operation	(00xx)
0010	Host cycle	
0011	DRAM refresh	
0100	Video-generated VRAM serial-register trans.	VRAM
0101	CPU-generated VRAM serial-register trans.	(01xx)
0110	Write-mask load	
0111	Color-register load	
1000	Data access	CPU
1001	Cache fill	(1xxx)
1010	Instruction fetch	
1011	Interrupt-vector fetch	
1100	Bus-locked operation	
1101	Pixel operation	
1110	Block write	
1111	Reserved	

Memory Map

Address Range	Size	Use
FFFF FFE0h FFFF FBC0h	34 words	Interrupt & trap vectors
FFFF FBA0h FFFF E000h	222 words	Reserved for interrupt & extended trap vectors
FFFF DFE0h FFF0 0000h	32,512 words	General use & extended trap vectors
FFEF FFE0h C000 2000h	2^{25}–33,024 words (35,521,408 words)	General use
C000 1FE0h C000 0400h	224 words	Reserved for I/O registers
C000 03E0h C000 0000h	32 words	I/O registers
BFFF FFE0h 0010 0000h	3×2^{25}–32K words (100,630,528 words)	General use
000F FFE0h 0000 0000h	32,768 words	General use & extended trap vectors

Interrupt Priorities

Interrupt	Priority	Source	Description
$\overline{\text{RESET}}$	1	external/ internal	Device reset
BF	2	external	Bus fault interrupt
NMI	3	internal	Nonmaskable interrupt
HI	4	internal	Host interrupt.
DI	5	internal	Display interrupt
WV	6	internal	Window violation interrupt
INT1	7	external	External interrupt 1
INT2	8	external	External interrupt 2
SS	9	internal	Single-step interrupt
ILLOP	10	internal	Illegal-opcode interrupt

Vector Address Map

Trap#	Address	Desc.	Trap#	Address	Desc.
–32768 to –1	000F FFE0h to 0000 0000h	Application specific	12 to 15	FFFF FE60h to FFFF FDE0h	Reserved
0	FFFF FFE0h	RESET	16 to 29	FFFF FDE0h to FFFF FC40h	Application specific
1	FFFF FFC0h	INT1			
2	FFFF FFA0h	INT2	30	FFFF FC20h	ILLOP
3 to 7	FFFF FF80h to FFFF FE00h	Reserved	31	FFFF FC00h	Application specific
8	FFFF FEE0h	NMI	32	FFFF FBE0h	SS
9	FFFF FEC0h	HI	33	FFFF FBC0h	BF
10	FFFF FEA0h	DI	34 to 32767	FFFF FBA0h to FFF0 0000h	Application specific
11	FFFF FE80h	WV			

TMS34020 Assembly Language Instruction Set

ABS *Rd*	**CMPI** *IL,Rd*
ADD *Rs,Rd*	**CMPK**
ADDC *Rs,Rd*	**CMPXY** *Rs,Rd*
ADDI *IW,Rd*	**CPW** *Rs,Rd*
ADDI *IL,Rd*	**CVDXYL** *Rd*
ADDK *K,Rd*	**CVMXYL** *Rd*
ADDXY *Rs,Rd*	**CVSXYL** *Rs,Rd*
ADDXYI *IL,Rd*	**CVXYL** *Rs,Rd*
AND *Rs,Rd*	**DEC** *Rd*
ANDI *IL,Rd*	**DINT**
ANDN *Rs,Rd*	**DIVS** *Rs,Rd*
ANDNI *IL,Rd*	**DIVU** *Rs,Rd*
BLMOVE *S,D*	**DRAV** *Rs,Rd*
BTST *K,Rd*	**DSJ** *Rd,Address*
BTST *Rs,Rd*	**DSJEQ** *Rd,Address*
CALL *Rs*	**DSJNE** *Rd,Address*
CALLA *Addr*	**DSJS** *Rd,Address*
CALLR *Addr*	**EINT**
CEXEC *size,instruction[,ID]*	**EMU**
CEXEC *size,instruction[,ID]*	**EXGF** *Rd,F*
CLIP	**EXGPC** *Rd*
CLR *Rd*	**EXGPS** *Rd*
CLRC	**FILL L**
CMOVCG *Rd_1[,Rd_2[size]], command[,ID]*	**FILL XY**
CMOVCM **Rd+, transfers, size,command[, ID]*	**FLINE** {0 | 1}
CMOVCM *–*Rd,transfers,size, command[,ID]*	**FPIXEQ**
CMOVCS *command[,ID]*	**FPIXNE**
CMOVGC *Rs,command[,ID]*	**GETPC** *Rd*
CMOVGC *Rs_1,Rs_2,size,command [,ID]*	**GETPS** *Rd*
CMOVMC **Rs+,transfers, size,command[,ID]*	**GETST** *Rd*
CMOVMC *–*Rs,transfers,size, command[,ID]*	**IDLE**
CMOVMC **Rs+,Rd,size,command [,ID]*	**INC**
CMP *Rs,Rd*	**JA***cc Address*
CMPI *IW,Rd*	**JR***cc Address*
	JR*cc Address*
	JUMP *Rs*
	LINE {0 | 1}
	LINIT
	LMO *Rs,Rd*

TMS34020 Assembly Language Instruction Set (continued)

MMFM *Rs,[,List]*	**NEGB** *Rd*
MMTM *Rs,[,List]*	**NOP**
MODS *Rs,Rd*	**NOT** *Rd*
MODU *Rs,Rd*	**OR** *Rs,Rd*
MOVB *Rs,*Rd*	**ORI** *IL,Rd*
MOVB **Rs,Rd*	**PFILL XY**
MOVB **Rs(Offset),Rd*	**PIXBLT B,L**
MOVB **Rs(SOffset), *Rd(DOffset)*	**PIXBLT B,XY**
MOVB *Rs,@DAddress*	**PIXBLT L,L**
MOVB *@SAddress,Rd*	**PIXBLT L,M,L**
MOVB *@SAddress, @DAddress*	**PIXBLT L,XY**
MOVE *Rs,Rd*	**PIXBLT XY,L**
MOVE *Rs,*Rd[,F]*	**PIXBLT XY,XY**
MOVE *Rs,–*Rd[,F]*	**PIXT** *Rs,*Rd*
MOVE *Rs,*Rd+[,F]*	**PIXT** *Rs,*Rd* **.XY**
MOVE **Rs,Rd[,F]*	**PIXT** **Rs,Rd*
MOVE *–**Rs,Rd[,F]*	**PIXT** **Rs.XY,Rd*
MOVE **Rs+,Rd[,F]*	**PIXT** **Rs.XY,*Rd.XY*
MOVE **Rs,*Rd[,F]*	**POPST**
MOVE *–**Rs,–*Rd[,F]*	**PUSHST**
MOVE **Rs+,*Rd+*	**PUTST** *Rs*
MOVE *Rs,*Rd(Offset)[,F]*	**RETI**
MOVE **Rs(Offset),Rd[,F]*	**RETM**
MOVE **Rs(Offset),*Rd+[,F]*	**RETS** *[N]*
MOVE **Rs(SOffset), *Rd(DOffset)[,F]*	**REV** *Rd*
MOVE *Rs,@DAddress[,F]*	**RL** *K,Rd*
MOVE *@SAddress,Rd[,F]*	**RL** *Rs,Rd*
MOVE *@SAddress,*Rd+[,F]*	**RMO**
MOVE *@SAddress, @DAddress[,F]*	**RPIX** *Rd*
MOVI *IW,Rd*	**SETC**
MOVI *IL,Rd*	**SETCDP**
MOVK *K,Rd*	**SETCMP**
MOVX *Rs,Rd*	**SETCSP**
MOVY *Rs,Rd*	**SETF** *FS,FE,F*
MPYS *Rs,Rd*	**SEXT** *Rd,F*
MPYU *Rs,Rd*	**SLA** *K,Rd*
MWAIT	**SLA** *Rs,Rd*
NEG *Rd*	**SLL** *K,Rd*

TMS34020 Assembly Language Instruction Set (continued)

SRA *K,Rd*	**SWAPF** *Rs,Rd,0*
SRA *Rs,Rd*	**TFILL XY**
SRL *K,Rd*	**TRAP** *N*
SRL *Rs,Rd*	**TRAPL**
SUB *Rs,Rd*	**VBLT**
SUBB *Rs,Rd*	**VFILL**
SUBI *IW,Rd*	**VLCOL**
SUBI *IL,Rd*	**XOR** *Rs,Rd*
SUBK *K,Rd*	**XORI** *IL,Rd*
SUBXY *Rs,Rd*	**ZEXT** *Rd,F*

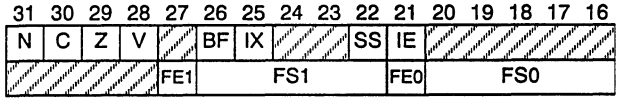
Boolean Pixel-Processing Options

00000	Source → Destination
00001	Source AND Destination → Destination
00010	Source AND ~Destination → Destination
00011	0s → Destination
00100	Source OR ~Destination → Destination
00101	Source XNOR Destination → Destination
00110	~Destination → Destination
00111	Source NOR Destination → Destination
01000	Source OR Destination → Destination
01001	Destination → Destination
01010	Source XOR Destination → Destination
01011	~Source AND Destination → Destination
01100	1s → Destination
01101	~Source OR Destination → Destination
01110	Source NAND Destination → Destination
01111	~Source → Destination

Arithmetic Pixel-Processing Options

10000	Source + Destination → Destination
10001	ADDS(Source, Destination) → Destination
10010	Destination – Source → Destination
10011	SUBS(Source, Destination) → Destination
10010	MAX(Source, Destination) → Destination
10101	MIN(Source, Destination) → Destination
10110—11111	Reserved

Status Register



Note: Shaded portions are reserved.

Index

B

C

D

E

F

G

H

M

S

T

V

W

X

Y

Z

TEXAS INSTRUMENTS

Printed in U.S.A., August 1990
2564006-9761 revision *

SPVU019