

# **TMS320C4x**

## *User's Guide*

# ***TMS320C4x*** ***User's Guide***

2564090-9721 revision A  
May 1991



## **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.

Copyright © 1991, Texas Instruments Incorporated

## Read This First

---

---

---

The purpose of this user's guide is to serve as a reference book for the TMS320C40 and TMS320C40-40 digital signal processors. Throughout the book, all references to the TMS320C40 apply to the TMS320C40-40 as well, unless an exception is noted. This document provides information to assist managers and hardware/software engineers in application development.

### *How to Use This Manual*

This document contains the following chapters:

- Chapter 1 Introduction**  
A general description of the TMS320C40, its key features, and typical applications.
- Chapter 2 Architectural Overview**  
Functional block diagrams. TMS320C40 design description, hardware components, and device operation. Instruction set summary.
- Chapter 3 CPU Registers, Memory, and Cache**  
Description of the registers in the CPU primary register file and expansion register file. Memory maps. Instruction cache architecture, algorithm, and control bits.
- Chapter 4 Data Formats and Floating-Point Operation**  
Description of signed and unsigned integer and floating-point formats. Discussion of floating-point multiplication, addition, subtraction, normalization, rounding, conversions, and reciprocals.
- Chapter 5 Addressing**  
Addressing types. Operation, encoding, and implementation of addressing modes. Format descriptions. Circular and bit-reversed addressing. System stack management.

**Chapter 6 Program Flow Control**

Software control of program flow using repeat modes, different types of branching, traps, interrupts, and interlocked operations. Reset operation, including resulting values in registers and on pins.

**Chapter 7 External Bus Operation**

Discussion of the two 80-pin local and global memory interfaces. Programmable wait-states. Memory access timing. Signal group control. Interlocked instructions. Interrupt acknowledge timing.

**Chapter 8 Communication Ports**

Description of the six, bidirectional, 160-megabit-per-second (at 40-ns cycle time) communication ports designed for sharing tasks between processors. Memory maps of the ports and their registers. Port operation and coordination of port activity with CPU and DMA coprocessors.

**Chapter 9 DMA Coprocessors and 'C40 Timers**

DMA coprocessor operation. Description of coprocessor registers (channel control, channel address, index, transfer count, and link pointer). Use in unified and split mode. Priority and CPU/DMA arbitration. Autoinitialization and interrupts. Operation of the 'C40 timers; their registers (global control, timer counter, and period).

**Chapter 10 Pipeline Operation**

Discussion of 'C40 pipeline operations. This includes pipeline conflicts and methods for resolving these. Clocking of memory accesses.

**Chapter 11 Assembly Language Instructions**

Functional listing of instructions. Condition code definitions (for conditional instructions such as branch conditional). Alphabetized individual instruction descriptions with examples.

**Chapter 12 Software Applications**

Software application examples for using various TMS320C40 instruction-set and programming features. Code listings enhance explanations.

**Chapter 13 Hardware Applications**

Hardware design techniques and application examples for interfacing to memories, peripherals, or other microcomputers/microprocessors. Code listings, schematics, and timing diagrams facilitate explanations.

**Chapter 14 TMS320C4x Signal Descriptions and Electrical Characteristics**

Pin locations and pin descriptions. 'C40 dimensions and package description. Electrical characteristics. Signal timing and characteristics.

## Appendix A TMS320C40 Sockets

Two sockets available for the TMS320C40.

## Appendix B XDS510 Design Considerations

Considerations for designing your TMS320C40 target system for use with the XDS510 emulator.

## References

The publications in the following reference list contain useful information regarding functions, operations, and applications of digital signal processing. These books also provide other references to many useful technical papers. The reference list is organized into categories of general DSP, speech, image processing, and digital control theory, and is alphabetized by author.

### □ General Digital Signal Processing:

Antoniou, Andreas, *Digital Filters: Analysis and Design*. New York, NY: McGraw-Hill Company, Inc., 1979.

Brigham, E. Oran, *The Fast Fourier Transform*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1974.

Burrus, C.S., and Parks, T.W., *DFT/FFT and Convolution Algorithms*. New York, NY: John Wiley and Sons, Inc., 1984.

*Digital Signal Processing Applications with the TMS320 Family*. Texas Instruments, 1986; Prentice-Hall, Inc., 1987.

Gold, Bernard, and Rader, C.M., *Digital Processing of Signals*. New York, NY: McGraw-Hill Company, Inc., 1969.

Hamming, R.W., *Digital Filters*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

IEEE ASSP DSP Committee (Editor), *Programs for Digital Signal Processing*. New York, NY: IEEE Press, 1979.

Jackson, Leland B., *Digital Filters and Signal Processing*. Hingham, MA: Kluwer Academic Publishers, 1986.

Jones, D.L., and Parks, T.W., *A Digital Signal Processing Laboratory Using the TMS32010*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Lim, Jae, and Oppenheim, Alan V. (Editors), *Advanced Topics in Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Morris, L. Robert, *Digital Signal Processing Software*. Ottawa, Canada: Carleton University, 1983.

Meyer, Riamund and Schwartz, Karl , *FFT Implementation on DSP Chips— Theory and Practice*, Proceedings of ICAASP 90, vol. 3, 1990

Oppenheim, Alan V. (Editor), *Applications of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Oppenheim, Alan V., and Schafer, R.W., *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

Oppenheim, Alan V., and Willsky, A.N., with Young, I.T., *Signals and Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983.

Parks, T.W., and Burrus, C.S., *Digital Filter Design*. New York, NY: John Wiley and Sons, Inc., 1987.

Rabiner, Lawrence R., and Gold, Bernard, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

Sorensen, Henrik V., Jones, Douglas, Heideman, M.T., and Burris, C.S., *Real-Valued Fast Fourier Transform Algorithms*, IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-35, no. 6, June 1987.

Treichler, J.R., Johnson, Jr., C.R., and Larimore, M.G., *Theory and Design of Adaptive Filters*. New York, NY: John Wiley and Sons, Inc., 1987.

□ **Speech:**

Gray, A.H., and Markel, J.D., *Linear Prediction of Speech*. New York, NY: Springer-Verlag, 1976.

Jayant, N.S., and Noll, Peter, *Digital Coding of Waveforms*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

Papamichalis, Panos, *Practical Approaches to Speech Coding*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Rabiner, Lawrence R., and Schafer, R.W., *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

□ **Image Processing:**

Andrews, H.C., and Hunt, B.R., *Digital Image Restoration*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

Gonzales, Rafael C., and Wintz, Paul, *Digital Image Processing*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1977.

Pratt, William K., *Digital Image Processing*. New York, NY: John Wiley and Sons, 1978.

## □ Digital Control Theory:

Jacquot, R., *Modern Digital Control Systems*. New York, NY: Marcel Dekker, Inc., 1981.

Katz, P., *Digital Control Using Microprocessors*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

Kuo, B.C., *Digital Control Systems*. New York, NY: Holt, Reinholt and Winston, Inc., 1980.

Moroney, P., *Issues in the Implementation of Digital Feedback Compensators*. Cambridge, MA: The MIT Press, 1983.

Phillips, C., and Nagle, H., *Digital Control System Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

## Style and Symbol Conventions

This document uses the following conventions:

- Program listings, program examples, interactive displays, file names, and symbol names are shown in a special font. Examples use a bold version of the special font for emphasis. Here is a sample program listing:

```
0011 0005 0001      .field    1, 2
0012 0005 0003      .field    3, 4
0013 0005 0006      .field    6, 3
0014 0006                .even
```

- In syntax descriptions, the instruction, command, or directive is in a **bold face font** and parameters are in *italics*. Portions of a syntax that are in **bold face** should be entered as shown; portions of a syntax that are in *italics* describe the *type* of information that should be entered. Here is an example of an instruction:

**CMPF3** *src2,src3*

Note: Although the instruction mnemonic (CMPF3 in this example) is in capital letters, the 'C40 assembler is **not case sensitive** — it can assemble mnemonics entered in either upper or lower case.

**CMPF3** is the instruction mnemonic. This instruction has two parameters, indicated by *src2* and *src3*.

- ❑ Square brackets ( [ and ] ) identify an optional parameter. If you use an optional parameter, you must specify the information within the brackets; however, you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

**LDP *src* [,*DP*]**

The **LDP** instruction is shown with two parameters; one is optional. The first parameter, *src*, is required. The second parameter, *DP*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

- ❑ Braces ( { and } ) indicate a list. The symbol | (read as *or*) separates items within the list. Here's an example of a list:

{ \* | \*+ | \*- }

This provides three choices: \*, \*+, or \*-.

Unless the list is enclosed in square brackets, you must choose one item from the list.

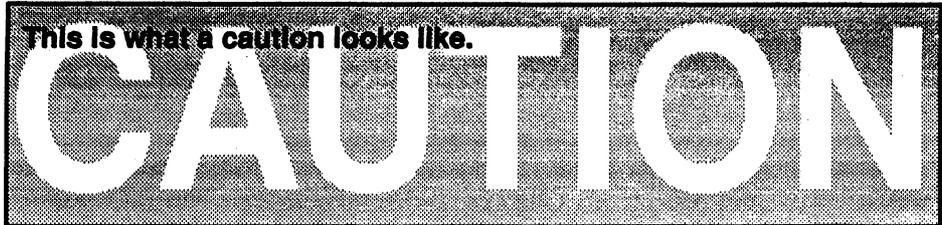
- ❑ The following is the format for a varying number of parameters. For example, the **.byte** directive can have up to 100 parameters. The syntax for this directive is

**.byte *value*<sub>1</sub> [, ... , *value*<sub>*n*</sub>]**

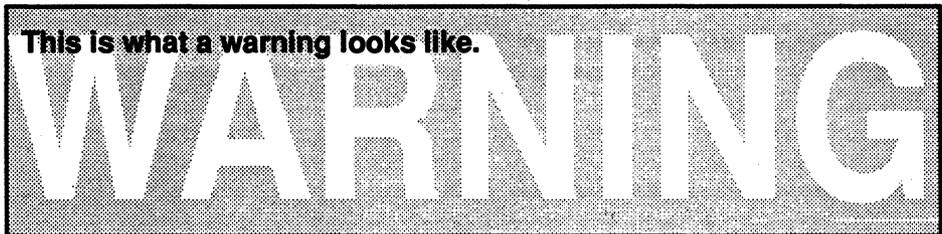
This syntax shows that **.byte** must have at least one value parameter, but you have the option of supplying additional value parameters separated by commas.

## **Information About Cautions and Warnings**

- ❑ A **caution** describes a situation that could potentially damage your software or equipment.



- ❑ A **warning** describes a situation that could potentially cause harm to you.



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

## **Trademarks**

ABEL is a trademark of the Data I/O Corporation.

SPOX is a trademark of Spectron Microsystems, Inc.



# Contents

---

---

---

<b>1</b>	<b>Introduction</b>	<b>1-1</b>
1.1	The TMS320 Family	1-2
1.2	Parallel Processing	1-3
1.3	TMS320C4x Features	1-4
1.3.1	TMS320C40 Device Key Features	1-4
1.3.2	Communication Port Benefits	1-7
1.3.3	DMA Coprocessor Benefits	1-8
1.3.4	TMS320C40 Parallel Processing Development Tools Key Features	1-9
1.4	Applications	1-11
<b>2</b>	<b>Architectural Overview</b>	<b>2-1</b>
2.1	Central Processing Unit (CPU)	2-4
2.1.1	Multiplier	2-4
2.1.2	Arithmetic Logic Unit (ALU)	2-4
2.1.3	Auxiliary Register Arithmetic Units (ARAUs)	2-6
2.1.4	CPU Primary Register File	2-6
2.1.5	CPU Expansion Register File	2-9
2.2	Memory Organization	2-10
2.2.1	RAM, ROM, and Cache	2-10
2.2.2	Memory Maps	2-12
2.2.3	Memory Addressing Modes	2-15
2.3	Instruction Set Summary	2-16
2.4	Internal Bus Operation	2-26
2.5	External Bus Operation	2-27
2.5.1	Interrupts	2-27
2.5.2	Interlocked Instructions	2-27
2.6	Peripherals	2-28
2.6.1	Communication Ports	2-29
2.6.2	Direct Memory Access (DMA)	2-29
2.6.3	Timers	2-29
<b>3</b>	<b>CPU Registers, Memory, and Cache</b>	<b>3-1</b>
3.1	CPU Primary Register File	3-3

3.1.1	Extended-Precision Registers (R0–R11)	3-4
3.1.2	Auxiliary Registers (AR0–AR7)	3-5
3.1.3	Data-Page Pointer (DP)	3-5
3.1.4	Index Registers (IR0, IR1)	3-5
3.1.5	Block-Size Register (BK)	3-5
3.1.6	System Stack Pointer (SP)	3-5
3.1.7	Status Register (ST)	3-5
3.1.8	DMA Coprocessor Interrupt Enable Register (DIE)	3-8
3.1.9	CPU Internal Interrupt Enable Register (IIE)	3-10
3.1.10	IIOF Flag Register (IIF) Controls External Pins IIOF(3 – 0), Timer/DMA Flags	3-12
3.1.11	Block-Repeat (RS, RE) and Repeat-Count (RC) Registers	3-14
3.1.12	Program Counter (PC)	3-14
3.1.13	Reserved Bits and Compatibility	3-14
3.2	CPU Expansion Register File	3-15
3.3	RESET Vector Mapping	3-17
3.4	Memory	3-18
3.4.1	Overall Memory Map	3-19
3.4.2	Peripheral Bus Memory Map	3-20
3.5	Instruction Cache Architecture	3-25
3.5.1	Cache Algorithm	3-27
3.5.2	Cache and System Memory	3-28
3.5.3	Cache Control Bits	3-29
<b>4</b>	<b>Data Formats and Floating-Point Operation</b>	<b>4-1</b>
4.1	Signed Integer Formats	4-3
4.1.1	Short Integer Format	4-3
4.1.2	Single-Precision Integer Format	4-3
4.2	Unsigned-Integer Formats	4-4
4.2.1	Short Unsigned-Integer Format	4-4
4.2.2	Single-Precision Unsigned-Integer Format	4-4
4.3	Floating-Point Formats	4-5
4.3.1	Short Floating-Point Format	4-6
4.3.2	Single-Precision Floating-Point Format	4-7
4.3.3	Extended-Precision Floating-Point Format	4-8
4.3.4	Conversion Between Floating-Point Formats	4-9
4.4	Floating-Point Conversions (IEEE Std. 754/C4x)	4-11
4.4.1	Converting IEEE Format to Twos-Complement Floating-Point Format	4-12
4.4.2	Converting Twos-Complement Floating-Point Format to IEEE Format	4-13

4.5	Floating-Point Multiplication .....	4-15
4.6	Floating-Point Addition and Subtraction .....	4-20
4.7	Normalization (NORM Instruction) .....	4-24
4.8	Rounding (RND Instruction) .....	4-26
4.9	Floating-Point-to-Integer Conversion (FIX Instruction) .....	4-28
4.10	Integer-to-Floating-Point Conversion (FLOAT Instruction) .....	4-30
4.11	Reciprocal (RCPF Instruction) .....	4-31
	4.11.1 Reciprocal Algorithm .....	4-32
4.12	Reciprocal Square Root (RSQRF Instruction) .....	4-33
	4.12.1 Reciprocal Square Root Algorithm .....	4-34
	4.12.2 Background on the Reciprocal Square Root .....	4-35
<b>5</b>	<b>Addressing .....</b>	<b>5-1</b>
5.1	Types of Addressing .....	5-2
	5.1.1 Register Addressing .....	5-3
	5.1.2 Direct Addressing .....	5-4
	5.1.3 Indirect Addressing .....	5-5
	5.1.4 Immediate Addressing .....	5-17
	5.1.5 PC-Relative Addressing .....	5-17
5.2	Groups of Addressing Modes .....	5-19
	5.2.1 General Addressing Modes .....	5-19
	5.2.2 Three-Operand Addressing Modes .....	5-20
	5.2.3 Parallel Addressing Modes .....	5-23
	5.2.4 Conditional-Branch Addressing Modes .....	5-24
5.3	Circular Addressing .....	5-25
5.4	Bit-Reversed Addressing .....	5-30
5.5	System and User Stack Management .....	5-31
	5.5.1 Stacks .....	5-32
	5.5.2 Queues and Dequeues .....	5-33
<b>6</b>	<b>Program Flow Control .....</b>	<b>6-1</b>
6.1	Repeat Modes .....	6-2
	6.1.1 Repeat-Mode Initialization .....	6-2
	6.1.2 RPTB and RPTBD Initialization .....	6-3
	6.1.3 RPTS Initialization .....	6-4
	6.1.4 Repeat-Mode Operation .....	6-4
6.2	Delayed Branches .....	6-7
6.3	Calls, Traps, Branches, Jumps, and Returns .....	6-9
6.4	Unifying Traps and Interrupts .....	6-11
	6.4.1 Initialization .....	6-11
	6.4.2 Operation .....	6-11

6.5	Interlocked Operations .....	6-13
6.6	Reset Operation .....	6-18
6.7	Interrupts .....	6-23
6.7.1	Interrupt Control Bits .....	6-24
6.7.2	Prioritization and Control .....	6-24
<b>7</b>	<b>External Bus Operation .....</b>	<b>7-1</b>
7.1	Global (and Local) Memory Interface Control Signals .....	7-3
7.2	Memory Interface Control Registers .....	7-6
7.3	Use of the Global Memory Interface Registers .....	7-12
7.3.1	Mapping Addresses to Strobes .....	7-12
7.3.2	Page Size Operation .....	7-13
7.4	Programmable Wait States .....	7-15
7.5	Timing .....	7-17
7.6	Using Enabled Signals to Control Signal Group .....	7-38
7.7	Interlocked-Instructions Definition and Bus Timing .....	7-39
7.8	IACK Timing .....	7-47
<b>8</b>	<b>Communication Ports .....</b>	<b>8-1</b>
8.1	Introduction .....	8-2
8.2	Communication Port Features .....	8-3
8.3	Operational Overview .....	8-5
8.4	Communication Port Memory Map and Registers .....	8-8
8.4.1	Communication Port Control Registers (CPCRs) .....	8-9
8.4.2	Input Port Register .....	8-9
8.4.3	Output Port Register .....	8-9
8.5	Communication Port Operation .....	8-12
8.5.1	Port Arbitration Units (PAUs) .....	8-12
8.5.2	Module Reset .....	8-14
8.5.3	Halting of Input and Output FIFOs .....	8-15
8.6	Coordinating Communication Port Activity With CPU and DMA Coprocessors .....	8-17
8.7	Communication Port Timing .....	8-18
8.7.1	Timing Table and Figures .....	8-18
8.7.2	Synchronizer Timing .....	8-31
<b>9</b>	<b>DMA Coprocessor and 'C40 Timers .....</b>	<b>9-1</b>
9.1	Introduction .....	9-2
9.2	DMA Coprocessor Functional Description .....	9-3
9.3	DMA Coprocessor Registers .....	9-7
9.3.1	DMA Channel Control Register .....	9-7
9.3.2	DMA Channel Address and Index Registers .....	9-16

9.3.3	DMA Channel Transfer-Counter and Auxiliary-Transfer-Count Registers .....	9-18
9.3.4	DMA-Channel Link-Pointer and Auxiliary-Link-Pointer Registers .....	9-19
9.4	DMA Channels in Unified and Split Modes .....	9-20
9.5	DMA Coprocessor Internal Priority Schemes .....	9-22
9.5.1	Fixed Priority Scheme .....	9-22
9.5.2	Rotating Priority Scheme .....	9-22
9.5.3	Split Mode and DMA Channel Arbitration .....	9-24
9.6	CPU and DMA Coprocessor Arbitration .....	9-27
9.7	Data Transfer Modes .....	9-28
9.7.1	Running Under TRANSFER MODE = 00 <sub>2</sub> .....	9-28
9.7.2	Running Under TRANSFER MODE = 01 <sub>2</sub> .....	9-28
9.7.3	Running Under TRANSFER MODE = 10 <sub>2</sub> .....	9-29
9.7.4	Running Under TRANSFER MODE = 11 <sub>2</sub> .....	9-30
9.8	Autoinitialization .....	9-31
9.8.1	Fun With Link Pointers .....	9-38
9.9	DMA Coprocessor and Interrupts .....	9-40
9.9.1	Interrupts and Synchronization of DMA Channels .....	9-41
9.10	TMS320C40 Timers .....	9-45
9.10.1	Timer Global-Control Register .....	9-47
9.10.2	Timer Period and Counter Registers .....	9-50
9.10.3	Timer Pulse Generation .....	9-50
9.10.4	Timer Operation Modes .....	9-52
<b>10</b>	<b>Pipeline Operation .....</b>	<b>10-1</b>
10.1	Pipeline Structure .....	10-2
10.2	Pipeline Conflicts .....	10-4
10.2.1	Branch Conflicts .....	10-4
10.2.2	Register Conflicts .....	10-8
10.2.3	Memory Conflicts .....	10-11
10.3	Resolving Memory Conflicts .....	10-18
10.4	Clocking of Memory Accesses .....	10-20
10.4.1	Program Fetches .....	10-20
10.4.2	Data Loads and Stores .....	10-21
<b>11</b>	<b>Assembly Language Instructions .....</b>	<b>11-1</b>
11.1	Assembly Language Instructions — Instruction Set .....	11-3
11.1.1	Load-and-Store Instructions .....	11-3
11.1.2	Two-Operand Instructions .....	11-4
11.1.3	Three-Operand Instructions .....	11-6

11.1.4	Program Control Instructions .....	11-6
11.1.5	Interlocked Operations Instructions .....	11-7
11.1.6	Parallel Operations Instructions .....	11-8
11.2	Condition Codes and Flags .....	11-10
11.3	Individual Instructions .....	11-13
11.3.1	Symbols and Abbreviations .....	11-13
11.3.2	Optional Assembler Syntaxes .....	11-15
11.3.3	Individual Instruction Descriptions .....	11-17
<b>12</b>	<b>Software Applications .....</b>	<b>12-1</b>
12.1	Processor Initialization .....	12-3
12.1.1	Reset Process .....	12-3
12.1.2	Initialization .....	12-3
12.2	Program Control .....	12-9
12.2.1	Subroutines .....	12-9
12.2.2	Software Stack .....	12-13
12.2.3	Interrupt Service Routines .....	12-14
12.2.4	Delayed Branches .....	12-22
12.2.5	Repeat Modes .....	12-23
12.2.6	Computed GOTOs to Select Subroutines at Runtime .....	12-27
12.3	Logical and Arithmetic Operations .....	12-28
12.3.1	Bit Manipulation .....	12-28
12.3.2	Block Moves .....	12-29
12.3.3	Byte and Half-Word Manipulation .....	12-30
12.3.4	Bit-Reversed Addressing .....	12-31
12.3.5	Integer and Floating-Point Division .....	12-33
12.3.6	Square Root .....	12-38
12.3.7	Extended-Precision Arithmetic .....	12-41
12.3.8	Floating-Point Format Conversion: IEEE to/from TMS320C40 .....	12-42
12.4	Application-Oriented Operations .....	12-46
12.4.1	Companding .....	12-46
12.4.2	FIR, IIR, and Adaptive Filters .....	12-51
12.4.3	Matrix-Vector Multiplication .....	12-61
12.4.4	Fast Fourier Transforms (FFT) .....	12-63
12.4.5	Lattice Filters .....	12-88
12.5	Programming Tips .....	12-94
12.5.1	C-Callable Routines .....	12-94
12.5.2	Hints for Optimizing Assembly Code .....	12-95
12.6	Peripherals .....	12-97
12.6.1	Timers .....	12-97

---

12.6.2	Communication Ports .....	12-98
12.6.3	Direct Memory Access .....	12-101
<b>13</b>	<b>Hardware Applications .....</b>	<b>13-1</b>
13.1	System Configuration Options Overview .....	13-3
13.1.1	Categories of Interfaces on the TMS320C40 .....	13-3
13.2	Boot Loader Description and External ROM Interfacing .....	13-5
13.2.1	TMS320C40 Boot Loader Description/Operation .....	13-5
13.2.2	Boot Load Sequence .....	13-6
13.2.3	Examples of External Memory Loads .....	13-8
13.2.4	Communication Port Loading .....	13-8
13.2.5	External ROM Interfacing to the TMS320C40 .....	13-9
13.2.6	IIOF(3–1) Pin Loading .....	13-14
13.2.7	TMS320C40 Boot Loader Source Program .....	13-14
13.3	Global and Local Bus Interface .....	13-20
13.3.1	Zero Wait-State Interface to RAMs .....	13-20
13.4	Wait States and Ready Generation .....	13-27
13.4.1	ORing of the Ready Signals (STRBx SWW = 10) .....	13-28
13.4.2	ANDing of the Ready Signals (STRBx SWW = 11) .....	13-28
13.4.3	External Ready Generation .....	13-29
13.4.4	Ready Control Logic .....	13-30
13.4.5	Example Circuit .....	13-31
13.4.6	Page Switching Techniques .....	13-32
13.5	Parallel Processing Interfaces .....	13-37
13.5.1	Message Broadcasting From One TMS320C40 to Many TMS320C40's .....	13-37
13.5.2	Shared Global Memory Interface With Fair Bus Arbitration ..	13-38
13.5.3	Shared Bus Interface Overview .....	13-43
13.6	Bus Arbitration .....	13-48
13.6.1	Arbitration Implementation .....	13-48
13.6.2	Arbitration Alternatives .....	13-70
13.6.3	Global Bus Arbitration and Transfer Timing .....	13-70
13.6.4	Arbitration Protocol Limitations .....	13-74
13.7	Reset Signal Generation Control Function .....	13-75
<b>14</b>	<b>TMS320C4x Signal Descriptions and Electrical Characteristics .....</b>	<b>14-1</b>
14.1	Pinout and Pin Assignments .....	14-2
14.2	Signal Descriptions .....	14-7
14.3	TMS320C4x Mechanical Data .....	14-11
14.4	Electrical Specifications .....	14-12
14.5	Signal Transition Levels .....	14-14

---

14.6	Timing .....	14-15
<b>A</b>	<b>TMS320C4x Sockets .....</b>	<b>A-1</b>
A.1	Tool-Activated ZIF PGA Socket (TAZ) .....	A-2
A.2	Handle-Actuated ZIF PGA Socket (HAZ) .....	A-3
<b>B</b>	<b>XDS510 Design Considerations .....</b>	<b>B-1</b>
B.1	Header and Header Signals .....	B-2
B.2	Bus Protocol .....	B-3
B.3	Cable Pod .....	B-4
B.4	Test Clock Generated in Target System .....	B-7
B.5	Multiprocessor Configuration .....	B-8
B.6	Emulation Timing Calculations .....	B-11
<b>Index</b>	<b>.....</b>	<b>Index-1</b>

# Figures

---

---

1-1	TMS320 Family of Devices	1-2
1-2	TMS320C40 Throughput Increases Use of Communication Ports	1-7
1-3	TMS320C40 Throughput Increases Use of DMA Coprocessor	1-8
1-4	Matrix of TMS320 DSP Applications	1-11
2-1	TMS320C40 Block Diagram	2-2
2-2	Central Processing Unit (CPU)	2-5
2-3	Memory Organization	2-11
2-4	Memory Maps	2-13
2-5	Peripheral Memory Map	2-14
2-6	Peripheral Modules	2-28
3-1	Extended-Precision Register Floating-Point Format	3-4
3-2	Extended-Precision Register Integer Format	3-4
3-3	Status Register	3-6
3-4	DMA Interrupt Enable Register Bit Functions	3-8
3-5	Internal Interrupt Enable Register (IIE)	3-11
3-6	Interrupt Flag Register (IIF)	3-12
3-7	Trap Vector Table (TVT)	3-15
3-8	Interrupt-Vector Table (IVT)	3-16
3-9	Memory Maps	3-19
3-10	Peripheral Memory Map	3-20
3-11	Memory Interface Control Registers	3-21
3-12	Analysis Module Registers	3-21
3-13	Timer Registers	3-22
3-14	Communication Port Memory Map	3-23
3-15	DMA Coprocessor Memory Map	3-24
3-16	Address Partitioning for Cache Control Algorithm	3-25
3-17	Instruction Cache Architecture	3-26
4-1	Short Integer Format and Sign Extension of Short Integer	4-3
4-2	Single-Precision Integer Format	4-3

4-3	Short Unsigned-Integer Format and Zero Fill .....	4-4
4-4	Single-Precision Unsigned-Integer Format .....	4-4
4-5	Generic Floating-Point Format .....	4-5
4-6	Short Floating-Point Format .....	4-6
4-7	Single-Precision Floating-Point Format .....	4-7
4-8	Extended-Precision Floating-Point Format .....	4-8
4-9	IEEE Single-Precision Std. 754 Floating-Point Format .....	4-11
4-10	TMS320C4x Single-Precision Twos-Complement Floating-Point Format .....	4-11
4-11	Flowchart for Floating-Point Multiplication .....	4-16
4-12	Flowchart for Floating-Point Addition .....	4-21
4-13	Flowchart for NORM Instruction Operation .....	4-24
4-14	Flowchart for Floating-Point Rounding by the RND Instruction .....	4-27
4-15	Flowchart for Floating-Point-to-Integer Conversion by FIX Instructions ...	4-29
4-16	Flowchart for Integer-to-Floating-Point Conversion by FLOAT Instructions .....	4-30
4-17	RCPF Instruction Algorithm .....	4-31
4-18	Newton-Raphson Algorithm for Computing the Reciprocal .....	4-32
4-19	RSQRF Instruction Algorithm .....	4-33
4-20	Newton-Raphson Algorithm for Computing the Reciprocal Square Root .....	4-34
5-1	Direct Addressing .....	5-4
5-2	Encoding for 24-Bit PC-Relative Addressing Mode .....	5-18
5-3	Encoding for General Addressing Modes .....	5-20
5-4	Encoding for Type 1 Three-Operand Addressing Modes (‘C30 and ‘C40) .....	5-22
5-5	Encoding for Type 2 Three-Operand Addressing Modes (‘C40 Only) ...	5-22
5-6	Encoding for Parallel Addressing Modes .....	5-23
5-7	Encoding for Conditional-Branch Addressing Modes .....	5-24
5-8	Flowchart for Circular Addressing .....	5-26
5-9	Circular Buffer Implementation .....	5-27
5-10	Circular Addressing Example .....	5-28
5-11	Data Structure for FIR Filters .....	5-29
5-12	FIR Filter Code Using Circular Addressing .....	5-29
5-13	Bit-Reversed Addressing Example .....	5-30
5-14	System Stack Configuration .....	5-31

---

5-15	Implementations of High-to-Low Memory Stacks .....	5-32
5-16	Implementations of Low-to-High Memory Stacks .....	5-33
6-1	Repeat-Mode Control Algorithm .....	6-5
6-2	CALL Response Timing .....	6-10
6-3	Unified Flow of Traps and Interrupts .....	6-11
6-4	Multiple TMS320C40s Sharing Global Memory .....	6-16
6-5	Interrupt-Vector Table (IVT) .....	6-26
6-6	Interrupt Processing .....	6-27
7-1	Global and Local Memory Interface Control Signals .....	7-3
7-2	Format for the Memory-Interface Control Registers .....	7-7
7-3	Effects of STRB ACTIVE on Global Memory Bus Memory Map .....	7-12
7-4	STRBx PAGESIZE Fields Example .....	7-13
7-5	$\overline{\text{STRB}}$ and $\overline{\text{RDY}}$ Timing .....	7-17
7-6	Read Same Page, Read Same Page, Write Same Page Sequence .....	7-18
7-7	Write Same Page, Write Same Page, Read Same Page Sequence .....	7-19
7-8	Read Same Page, Read Different Page, Read Same Page Sequence ...	7-20
7-9	Write Same Page, Write Different Page, Write Same Page Sequence ...	7-21
7-10	Write Same Page, Read Different Page, Write Different Page Sequence .....	7-22
7-11	Read Different Page, Read Different Page, Write Same Page Sequence .....	7-23
7-12	Write Different Page, Write Different Page, Read Same Page Sequence .....	7-24
7-13	Read Same Page, Write Different Page, Read Different Page Sequence .....	7-25
7-14	Read Same Page, Idle One Cycle, Read Same Page Sequence .....	7-26
7-15	Write Same Page, Idle One Cycle, Write Different Page Sequence .....	7-27
7-16	Idle, Read Different Page, Idle Sequence .....	7-28
7-17	Idle, Write Same Page, Idle Sequence .....	7-29
7-18	Write Different or Same Page, Idle, Idle Sequence .....	7-30
7-19	Read Same Page on STRB1, Read Same Page on STRB0, Read Same Page on STRB1 Sequence When STRB SWITCH = 0 .....	7-31
7-20	Read Same Page on STRB1, Read Same Page on STRB0, Read Same Page on STRB1 Sequence When STRB SWITCH = 1 .....	7-32
7-21	Read Same Page on STRB1, Read Same Page on STRB0, Read Different Page on STRB1 Sequence When STRB SWITCH = 0 .....	7-33

7-22	Read Same Page on STRB1, Read Same Page on STRB0, Read Different Page on STRB1 Sequence When STRB SWITCH = 1 ...	7-34
7-23	Write Same Page on STRB1, Write Same Page on STRB0, Read Same Page on STRB1 Sequence .....	7-35
7-24	Read With One Wait State .....	7-36
7-25	Write With One Wait State .....	7-37
7-26	Using Enabled Signals to Put Signal Groups in a High-Impedance State .....	7-38
7-27	LDII or LDFI External Access .....	7-40
7-28	STII or STFI External Access .....	7-42
7-29	SIGI External Access Timing .....	7-44
7-30	SIGI When LOCK Is Already Low .....	7-46
7-31	$\overline{\text{TACK}}$ Timing .....	7-48
8-1	Communication Port Block Diagram .....	8-4
8-2	TMS320C40 Communication-Port Interface-Connection Example .....	8-5
8-3	Communication Port Memory Map .....	8-8
8-4	Communication Port Control Register (CPCR) .....	8-10
8-5	Communication Port Arbitration Unit State Diagram .....	8-13
8-6	Signal-Naming Example .....	8-18
8-7	Token Transfer Sequence .....	8-23
8-8	End of Token Transfer Sequence Followed by a Word Transfer and the Beginning of a Second Word Transfer .....	8-24
8-9	End of a Word Transfer Followed by a Word Transfer .....	8-25
8-10	End of a Word Transfer Followed by an Idle State and Token Transfer ...	8-26
8-11	End of a Word Transfer Followed by an Overlapping Token Transfer ....	8-27
8-12	End of the Transfer of the Last Word in an Output FIFO Followed by an Idle Condition Until Another Word Is Available to Be Transferred .....	8-28
8-13	End of a Word Transfer Followed by a Not Ready Due to the Input FIFO Becoming Full, Continuing Once the Input FIFO Is No Longer Full .....	8-29
8-14	Post-Reset State for an Output Port .....	8-30
8-15	Post-Reset State for an Input Port .....	8-30
8-16	Type-One Synchronizer Minimum Delay .....	8-31
8-17	Type-One Synchronizer Maximum Delay .....	8-31
8-18	Type-Two Synchronizer Minimum Delay .....	8-32
8-19	Type-Two Synchronizer Maximum Delay .....	8-32
9-1	DMA Coprocessor Memory Map .....	9-4

---

9-2	Subsections Where DMA Channel Registers Are Described .....	9-5
9-3	DMA Channel Control Register .....	9-8
9-4	DMA-Coprocessor Address Generation .....	9-17
9-5	DMA Coprocessor Transfer-Count Registers .....	9-18
9-6	DMA Coprocessor Link Pointer Registers .....	9-19
9-7	Typical Unified Mode DMA Channel Configuration .....	9-21
9-8	Typical Split-Mode DMA Configuration .....	9-21
9-9	Rotating Priority Mode Example of the DMA Coprocessor .....	9-23
9-10	DMA Read and Write Sequence Example .....	9-23
9-11	Example of a Priority Wheel .....	9-24
9-12	Example of a Channel Priority Scheme in Split Mode .....	9-25
9-13	Service Sequence for Split Mode Priority Example .....	9-26
9-14	Running a DMA Channel Under Transfer Mode 10 <sub>2</sub> .....	9-29
9-15	Running a DMA Channel Under Transfer Mode 11 <sub>2</sub> .....	9-30
9-16	Store New Values of DMA Channel Registers in Memory (SPLIT MODE = 0) .....	9-32
9-17	Store New Values of DMA Channel Registers in Memory (SPLIT MODE = 1) .....	9-33
9-18	Store New Values of DMA Channel Registers in Memory (SPLIT MODE = 1 and Auxiliary Transfer Counter = 0) .....	9-33
9-19	DMA Channel Control Register Bits That Can Be Modified by Autoinitialization Under Unified Mode .....	9-35
9-20	DMA Channel Control Register Bits That Can Be Modified by Autoinitialization of the Primary Channel Under Split Mode .....	9-35
9-21	DMA Channel Control Register Bits That Can Be Modified by Autoinitialization of the Auxiliary Channel Under Split Mode .....	9-36
9-22	Self-Referential Link Pointer .....	9-38
9-23	Referring to a New Link Pointer .....	9-39
9-24	No DMA Synchronization .....	9-41
9-25	DMA Source Synchronization .....	9-42
9-26	DMA Destination Synchronization .....	9-43
9-27	DMA Source and Destination Synchronization .....	9-44
9-28	Timer Block Diagram .....	9-45
9-29	Memory-Mapped Timer Locations .....	9-46
9-30	Timer Global-Control Register .....	9-47
9-31	Timer Timing .....	9-51

9-32	Timer I/O Port Configurations .....	9-52
9-33	Timer Modes as Defined by CLKSRC and FUNC .....	9-53
10-1	TMS320C40 Pipeline Structure .....	10-3
10-2	Two-Operand Instruction Word .....	10-21
10-3	Three-Operand Instruction Word .....	10-21
10-4	Multiply or CPU Operation With a Parallel Store .....	10-22
10-5	Two Parallel Stores .....	10-23
10-6	Parallel Multiplies and Adds .....	10-24
11-1	Status Register .....	11-11
12-1	Data Memory Organization for an FIR Filter .....	12-51
12-2	Data Memory Organization for a Single Biquad .....	12-54
12-3	Data Memory Organization for N Biquads .....	12-56
12-4	Data Memory Organization for Matrix-Vector Multiplication .....	12-61
12-5	Structure of the Inverse Lattice Filter .....	12-89
12-6	Data Memory Organization for Inverse Lattice Filters .....	12-89
12-7	Structure of the (Forward) Lattice Filter .....	12-92
13-1	External Interfaces to the TMS320C40 .....	13-3
13-2	Possible System Configurations .....	13-4
13-3	Circuit for Generation of a Low IIOF Signal for Boot Loader Selection ..	13-14
13-4	Boot Loader Source Program .....	13-14
13-5	TMS320C40 Interface to Zero-Wait-State SRAM .....	13-22
13-6	Consecutive Reads Followed by a Write .....	13-24
13-7	Consecutive Writes Followed by a Read .....	13-24
13-8	TMS320C40 Interface to Zero-Wait-State SRAMs, Two Strokes .....	13-26
13-9	Logic for Generation of 0, 1, or 2 Wait States for Multiple Devices .....	13-31
13-10	State Machine and Equation for the 16R4 PLD .....	13-33
13-11	Page Switching for the Cypress Semiconductor CY7C185 .....	13-35
13-12	Timing for Read Operations Using Bank Switching .....	13-36
13-13	Message Broadcasting by One 'C40 to Many 'C40s .....	13-38
13-14	TMS320C40 Parallel DSP System Architectures .....	13-40
13-15	TMS320C40 Shared Memory Interface .....	13-44
13-16	TMS320C40 Shared Memory and Bus Controller Interface .....	13-45
13-17	Successful TMS320C40 Arbitration and Data Read From Shared Bus Memory Followed by an Unsuccessful Arbitration Contest .....	13-47
13-18	Shared Bus Interface PLD State Machine .....	13-49

13-19	PLD Equations for Programming the 16R4 PLD (First-Level Logic) . . . . .	13-51
13-20	PLD Equations for Programming the 16R4 PLD . . . . .	13-60
13-21	Global Bus Controllor PLD (Rotating Priority Mode Only) . . . . .	13-62
13-22	PLD Equations for Programming the 16R8 PLD . . . . .	13-63
13-23	PLD Equations for Programming the 16R6 PLD . . . . .	13-68
13-24	Successful TMS320C40 Arbitration; Data Read; Data Read . . . . .	13-72
13-25	Successful TMS320C40 Arbitration and Data Write From Shared Bus Memory Followed by an Unsuccessful Arbitration Contest . . . . .	13-73
13-26	Successful 'C40 Arbitration; Consecutive Data Writes; Arbitration Win Followed by Successive Writes and an Arbitration Loss . . . . .	13-74
13-27	Reset Circuit . . . . .	13-75
13-28	Voltage on the TMS320C40 $\overline{\text{RESET}}$ Pin . . . . .	13-76
14-1	TMS320C40 Pinout (Bottom View) . . . . .	14-2
14-2	TMS320C40 325-Pin PGA Dimensions . . . . .	14-11
14-3	Test Load Circuit . . . . .	14-13
14-4	TTL-Level Outputs . . . . .	14-14
14-5	TTL-Level Inputs . . . . .	14-14
14-6	X2/CLKIN Timing . . . . .	14-15
14-7	H1/H3 Timing . . . . .	14-15
14-8	Memory ((L) $\overline{\text{STRB}}$ = 0) Read . . . . .	14-17
14-9	Memory ((L) $\overline{\text{STRB}}$ = 0) Write . . . . .	14-18
14-10	$\overline{\text{DE}}$ , $\overline{\text{AE}}$ , and $\overline{\text{CE}}$ Enable Timing . . . . .	14-19
14-11	Timing for (L) $\overline{\text{LOCK}}$ When Executing LDFI or LDII . . . . .	14-20
14-12	Timing for (L) $\overline{\text{LOCK}}$ When Executing a STFI or STII . . . . .	14-21
14-13	Timing for (L) $\overline{\text{LOCK}}$ When Executing SIGI . . . . .	14-22
14-14	Timing Parameters for (L)PAGE(0,1) . . . . .	14-23
14-15	Timing for Loading IIF Register (IIOF Pins) When Configured as an Output Pin . . . . .	14-24
14-16	Change of IIOF From Output to Input Mode . . . . .	14-25
14-17	Change of IIOF From Input to Output Mode . . . . .	14-26
14-18	$\overline{\text{RESET}}$ Timing . . . . .	14-27
14-19	IOOF(3 — 0) Response Timing . . . . .	14-29
14-20	$\overline{\text{IACK}}$ Timing . . . . .	14-30
14-21	Communication-Port Word-Transfer Cycle Timing . . . . .	14-31
14-22	Communication Port Byte Timing (Write and Read) . . . . .	14-32
14-23	Communication Token Transfer Sequence From an Input to an Output Port . . . . .	14-33

---

14-24	Communication Token Transfer Sequence From an Output to an Input Port .....	14-35
14-25	Timer Pin Timings .....	14-37
14-26	JTAG Emulation Timings .....	14-38
A-1	Tool-Activated ZIF Socket .....	A-2
A-2	Handle-Activated ZIF Socket .....	A-3
B-1	14-pin Header Signals and Header Dimensions .....	B-2
B-2	Emulator Pod Interface .....	B-5
B-3	Emulator Pod Timings .....	B-6
B-4	Target-System Generated Test Clock .....	B-7
B-5	Multiprocessor Connections .....	B-8
B-6	Unbuffered Signals .....	B-9
B-7	Buffered Signals .....	B-9

# Tables

2-1	CPU Primary Registers .....	2-7
2-2	Instruction Set Summary .....	2-16
2-3	Parallel Instruction Set Summary .....	2-23
3-1	CPU Primary Register File .....	3-3
3-2	Status Register Bits Summary .....	3-6
3-3	DMA Channels 0 and 1 Synchronization Interrupts (DMA0 and DMA1) ...	3-9
3-4	DMA Channels 2 to 5 Synchronization Interrupts (DMA2 to DMA5) .....	3-9
3-5	Summary of Interrupt Enable Register Bits (IIE) .....	3-11
3-6	IIF Register Bits Summary .....	3-13
3-7	CPU Expansion Registers .....	3-15
3-8	Four RESET Vector Locations Chosen by Values on Pins RESETLOC(1,0) .....	3-17
3-9	Combined Effect of the CE and CF Bits .....	3-29
4-1	Rules for Converting IEEE Format to Twos-Complement Floating-Point Format .....	4-12
4-2	Rules for Converting Twos-Complement Floating-Point Format to IEEE Format .....	4-13
5-1	CPU Register/Assembler Syntax and Function .....	5-3
5-2	Indirect Addressing .....	5-6
5-3	Three-Operand Instruction Address Forms .....	5-20
5-4	Index Steps and Bit-Reversed Addressing .....	5-30
6-1	Repeat-Mode Registers .....	6-2
6-2	Interlocked Operations .....	6-13
6-3	Pin Operation at Reset .....	6-18
7-1	Global Memory Interface Control Signals .....	7-4
7-2	Global Memory Port Status for STRB0 and STRB1 Accesses .....	7-5
7-3	Bit Definitions for Both Memory Interface Control Registers .....	7-8
7-4	Page Size as Defined by STRB0/1 PAGESIZE Bits .....	7-9

---

7-5	Address Ranges Specified by STRB ACTIVE Bits .....	7-10
7-6	Address Ranges Specified by LSTRB ACTIVE Bits .....	7-11
7-7	Wait-State Generation for Each Value of SWW .....	7-16
8-1	CPCR Bit Functions .....	8-10
8-2	PAU State Definitions .....	8-12
8-3	Summary of Input and Output FIFO Halting .....	8-16
8-4	Handshaking Events in Communication Port Intercommunication .....	8-20
8-5	Communication Port Signals and Synchronizer Delays .....	8-32
9-1	DMA Channel Control Register Bit Definitions .....	9-8
9-2	DMA PRI Bits and CPU/DMA Arbitration Rules .....	9-14
9-3	TRANSFER MODE and AUX TRANSFER MODE Field Description .....	9-14
9-4	SYNCH MODE Field Description .....	9-15
9-5	START and AUX START Field Description .....	9-15
9-6	STATUS and AUX STATUS Field Description .....	9-16
9-7	DMA PRI Bits and CPU/DMA Arbitration Rules .....	9-27
9-8	TRANSFER MODE Field Description Summary .....	9-28
9-9	Effect of SYNC MODE and AUTOINIT MODE Bits in Autoinitialization ...	9-37
9-10	Timer Global-Control Register Bits Summary .....	9-48
9-11	Result of a Write of Specified Values of GO and $\overline{HLD}$ .....	9-49
10-1	One Program Fetch and One Data Access for Maximum Performance .	10-18
10-2	One Program Fetch and Two Data Accesses for Maximum Performance	10-19
11-1	Load-and-Store Instructions .....	11-4
11-2	Two-Operand Instructions .....	11-5
11-3	Three-Operand Instructions .....	11-6
11-4	Program Control Instructions .....	11-7
11-5	Interlocked Operations Instructions .....	11-7
11-6	Parallel Instructions .....	11-8
11-7	Output Value Formats .....	11-10
11-8	Condition Codes and Flags .....	11-12
11-9	Instruction Symbols .....	11-14
11-10	CPU Register Syntax .....	11-17
12-1	Relationship of RESETLOC(1,0) Pins to RESET Vector Location .....	12-3
12-2	TMS320C40 FFT Timing Benchmarks .....	12-88
13-1	Boot Loader Mode Selection Using Pins IIOF(3-1) .....	13-5

---

13-2	Structure of Source Program Data Stream .....	13-7
13-3	Page Switching Interface Timing .....	13-36
14-1	TMS320C40 Pin Assignments Sorted by Signal Name .....	14-3
14-2	TMS320C40 Pin Assignments Sorted by Pin Number .....	14-5
14-3	TMS320C40 Signal Descriptions .....	14-7
14-4	Absolute Maximum Ratings Over Specified Temperature Range .....	14-12
14-5	Recommended Operating Conditions .....	14-12
14-6	Electrical Characteristics Over Specified Free-Air Temperature Range ..	14-13
14-7	Timing Parameters for CLKIN, H1, H3 (Figure 14-6 and Figure 14-7) ..	14-16
14-8	Timing Parameters for a Memory ( $\overline{\text{L}}\text{STRB} = 0$ ) Read/Write .....	14-17
14-9	$\overline{\text{DE}}$ , $\overline{\text{AE}}$ , and $\overline{\text{CE}}$ Enable Timing .....	14-19
14-10	Timing Parameters for $\overline{\text{(L)}}\text{LOCK}$ When Executing LDFI or LDII .....	14-20
14-11	Timing Parameters for $\overline{\text{(L)}}\text{LOCK}$ When Executing STFI or STII .....	14-21
14-12	Timing Parameters for $\overline{\text{(L)}}\text{LOCK}$ When Executing SIGI .....	14-22
14-13	Timing Parameters for (L)PAGE(0,1) During Memory Accesses to a Different Page .....	14-23
14-14	Timing Parameters for Loading IIF Register When Configured as an Output Pin .....	14-24
14-15	Timing Parameters of IIOF Changing From Output to Input Mode .....	14-25
14-16	Timing Parameters of IIOF Changing From Input to Output Mode .....	14-26
14-17	Timing Parameters for $\overline{\text{RESET}}$ (Figure 14-18) .....	14-28
14-18	Timing Parameters for IOOF(3- 0) .....	14-29
14-19	Timing Parameters for $\overline{\text{ACK}}$ .....	14-30
14-20	Communication-Port Word-Transfer Cycle Timing .....	14-31
14-21	Communication-Port Byte Timing (Write and Read) .....	14-32
14-22	Communication Token Transfer Sequence From an Input to an Output Port (Figure 14-23) .....	14-34
14-23	Communication Token Transfer Sequence From an Output to an Input Port (Figure 14-24) .....	14-36
14-24	Timing Parameters for Timer Pin .....	14-37
14-25	Timing Parameters for JTAG Emulation .....	14-38
B-1	14-Pin Header Signal Description .....	B-2
B-2	Emulator Pod Timing Parameters .....	B-6

# Examples

---

---

4-1	Floating-Point Multiply (Both Mantissas = -2.0) .....	4-18
4-2	Floating-Point Multiply (Both Mantissas = 1.5) .....	4-18
4-3	Floating-Point Multiply (Both Mantissas = 1.0) .....	4-19
4-4	Floating-Point Multiply Between Positive and Negative Numbers .....	4-19
4-5	Floating-Point Multiply by Zero .....	4-19
4-6	Floating-Point Addition .....	4-22
4-7	Floating-Point Subtraction .....	4-22
4-8	Floating-Point Addition With a 32-Bit Shift .....	4-23
4-9	Floating-Point Addition/Subtraction and Zero .....	4-23
4-10	NORM Instruction .....	4-25
5-1	Direct Addressing .....	5-4
5-2	Auxiliary Register Indirect .....	5-8
5-3	Indirect With Predisplacement Add .....	5-8
5-4	Indirect With Predisplacement Subtract .....	5-9
5-5	Indirect With Predisplacement Add and Modify .....	5-9
5-6	Indirect With Predisplacement Subtract and Modify .....	5-10
5-7	Indirect With Postdisplacement Add and Modify .....	5-10
5-8	Indirect With Postdisplacement Subtract and Modify .....	5-11
5-9	Indirect With Postdisplacement Add and Circular Modify .....	5-11
5-10	Indirect With Postdisplacement Subtract and Circular Modify .....	5-12
5-11	Indirect With Preindex Add .....	5-12
5-12	Indirect With Preindex Subtract .....	5-13
5-13	Indirect With Preindex Add and Modify .....	5-13
5-14	Indirect With Preindex Subtract and Modify .....	5-14
5-15	Indirect With Postindex Add and Modify .....	5-14
5-16	Indirect With Postindex Subtract and Modify .....	5-15
5-17	Indirect With Postindex Add and Circular Modify .....	5-15
5-18	Indirect With Postindex Subtract and Circular Modify .....	5-16

---

5-19	Indirect With Postindex Add and Bit-Reversed Modify .....	5-16
5-20	Immediate Addressing .....	5-17
5-21	PC-Relative Addressing .....	5-17
6-1	RPTB Operation .....	6-5
6-2	Incorrectly Placed Standard Branch .....	6-6
6-3	Incorrectly Placed Delayed Branch .....	6-6
6-4	Incorrectly Placed Delayed Branches .....	6-8
6-5	Busy-Waiting Loop .....	6-15
6-6	Task Counter Manipulation .....	6-15
6-7	Implementation of V(S) .....	6-16
6-8	Implementation of P(S) .....	6-16
10-1	Standard Branch .....	10-5
10-2	Delayed Branch .....	10-6
10-3	Using BcondAF and BcondAT Instructions .....	10-7
10-4	Write to an AR Followed by an AR for Address Generation .....	10-9
10-5	A Read of ARs Followed by ARs for Address Generation .....	10-10
10-6	Program Wait Until CPU Data Access Completes .....	10-12
10-7	Program Wait Due to Multicycle Access .....	10-13
10-8	Multicycle Program Memory Fetches .....	10-13
10-9	Single Store Followed by Two Reads .....	10-14
10-10	Parallel Store Followed by Single Read .....	10-15
10-11	Busy External Port .....	10-16
10-12	Multicycle Data Reads .....	10-17
10-13	Conditional Calls and Traps .....	10-17
12-1	Processor Initialization Example .....	12-4
12-2	Regular Subroutine Call (Dot Product) .....	12-10
12-3	Zero-Overhead Subroutine Call (Dot Product) .....	12-12
12-4	Use of Interrupts for Software Polling .....	12-15
12-5	Context-Save for the TMS320C40 .....	12-17
12-6	Context-Restore for the TMS320C40 .....	12-18
12-7	Use of One Interrupt Signal for Two Different Services .....	12-20
12-8	Interrupt Service Routine .....	12-21
12-9	Delayed Branch Execution .....	12-23
12-10	Use of Block Repeat to Find a Maximum or a Minimum .....	12-23

---

12-11	Loop Using Delayed Block Repeat .....	12-25
12-12	Loop Using Single Repeat .....	12-26
12-13	Computed GOTO .....	12-27
12-14	Use of TSTB for Software-Controlled Interrupt .....	12-28
12-15	Copy a Bit From One Location to Another .....	12-29
12-16	Block Move Under Program Control .....	12-30
12-17	Use of Packing Data From Half-Word FIFO to 32-Bit Data Memory ....	12-30
12-18	Use of Unpacking 32-Bit Data Into Four-Byte-Wide Data Array .....	12-31
12-19	Bit-Reversed Addressing .....	12-32
12-20	Integer Division .....	12-35
12-21	Inverse of a Floating-Point Number With 32-Bit Mantissa Accuracy ....	12-37
12-22	Reciprocal of the Square Root of a Positive Floating-Point .....	12-39
12-23	64-Bit Addition .....	12-41
12-24	64-Bit Subtraction .....	12-42
12-25	32-Bit by 32-Bit Multiplication .....	12-42
12-26	IEEE to TMS320C40 Conversion Within Block Memory Transfer .....	12-45
12-27	TMS320C40 to IEEE Conversion Within Block Memory Transfer .....	12-45
12-28	$\mu$ -Law Compression .....	12-47
12-29	$\mu$ -Law Expansion .....	12-48
12-30	A-Law Compression .....	12-49
12-31	A-Law Expansion .....	12-50
12-32	FIR Filter .....	12-52
12-33	IIR Filter (One Biquad) .....	12-54
12-34	IIR Filters ( $N > 1$ Biquads) .....	12-56
12-35	Adaptive FIR Filter (LMS Algorithm) .....	12-59
12-36	Matrix Times a Vector Multiplication .....	12-62
12-37	Complex, Radix-2, DIF FFT .....	12-65
12-38	Table With Twiddle Factors for a 64-Point FFT .....	12-68
12-39	Complex, Radix-4, DIF FFT .....	12-70
12-40	Real, Radix-2 FFT .....	12-75
12-41	Faster Version Complex, Radix-2 DIT FFT .....	12-78
12-42	Inverse Lattice Filter .....	12-90
12-43	Lattice Filter .....	12-92
12-44	Maximum Frequency Timer Clock Setup .....	12-98

---

12-45	Read Data From Communication Port With CPU ICFULL Interrupt . . . .	12-100
12-46	Write Data to Communication Port With Polling Method . . . . .	12-101
12-47	Array Initialization With DMA . . . . .	12-102
12-48	DMA Transfer With Communication Port ICRDY Synchronization . . . . .	12-103
12-49	DMA Split-Mode Transfer With External Interrupt Synchronization . . . . .	12-104
12-50	DMA Autoinitialization With Communication Port ICRDY . . . . .	12-106
12-51	Single-Interrupt-Driven DMA Transfer . . . . .	12-107
13-1	Byte-Wide Configured Memory . . . . .	13-10
13-2	16-Bits-Wide Configured Memory . . . . .	13-12
13-3	32-Bits-Wide Configured Memory . . . . .	13-13



# Chapter 1

## Introduction

---

---

---

Texas Instruments' TMS320C4x generation floating-point processors are designed specifically to meet the needs of parallel processing and other real-time embedded applications. TMS320C4x products consist of both parallel processing devices and development tools. With world-class parallel-processing development tools, designers are able to fully utilize the immense performance of 275 MOPS (millions of operations per second) and 320 Mbytes per second throughput made available by the TMS320C4x generation.

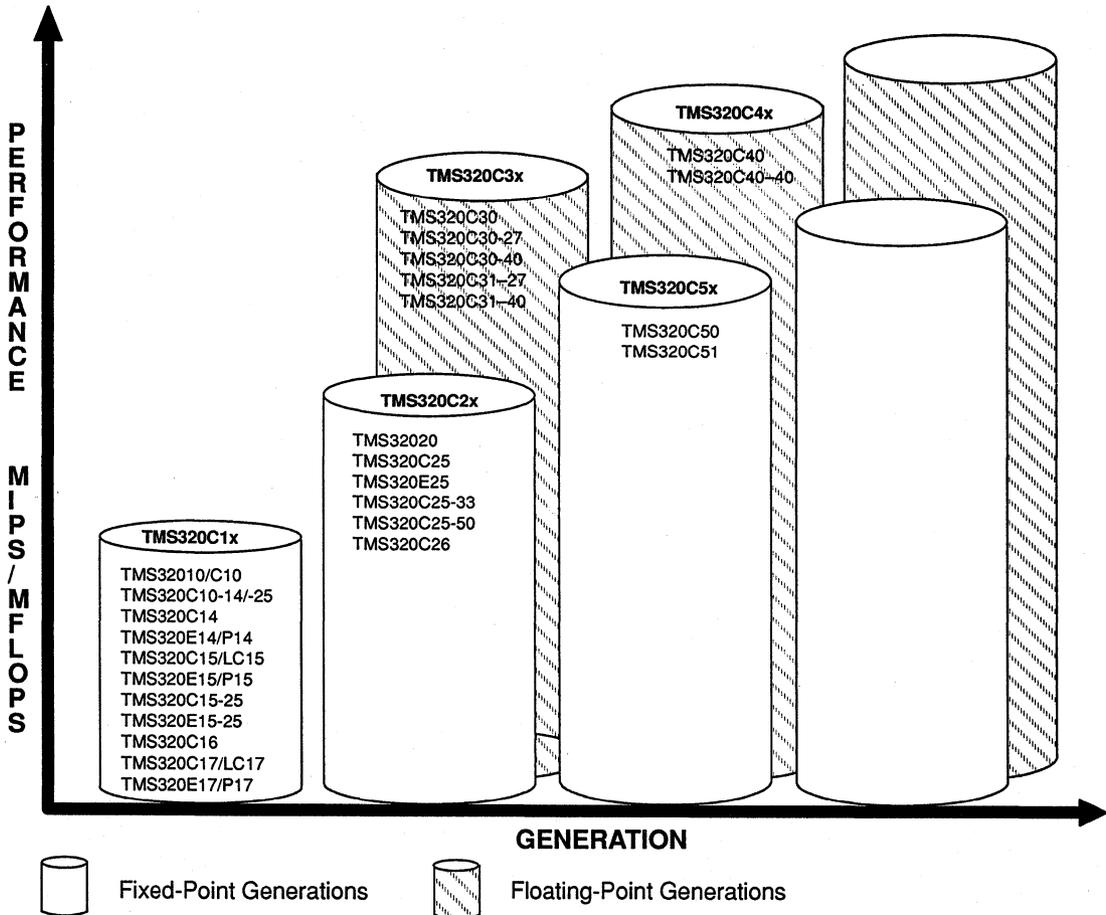
This chapter provides a brief overview of the TMS320C4x generation. Major topics covered are as follows:

<b>Section</b>	<b>Page</b>
1.1 The TMS320 Family .....	1-2
1.2 Parallel Processing .....	1-3
1.3 The TMS320C40x Generation .....	1-4
1.4 Applications .....	1-11

## 1.1 The TMS320 Family

The TMS320C4x is one of five generations in the TMS320 family of digital signal processors. The TMS320C1x, TMS320C2x, and TMS320C5x offer designers a complete line of general-purpose and application-specific fixed-point DSPs. The TMS320C3x and TMS320C4x generations round out the TMS320 family, providing an ensemble of floating-point DSPs. The TMS320 family has blossomed from a single device introduced in 1982, the TMS32010, to nearly thirty different products across five CPU architectures. On-chip hardware multipliers, register files, barrel shifters, ALUs, ROM, RAM, caches, and I/O peripherals along with massive internal busing (all within a product as programmable as a general-purpose microprocessor), make TI's TMS320 devices ideal for the gamut of computer-intensive applications.

Figure 1-1. TMS320 Family of Devices



## 1.2 Parallel Processing

The need for parallel processing is quickly growing. As floating-point performance requirements grow exponentially, semiconductor manufacturers can no longer meet the need with single processing elements. Processors not designed for parallel processing are inadequate for the task, as interprocessor communication quickly saturates device I/O and adversely affects computing efficiency. Products in the TMS320C3x generation made the first step in addressing the need for parallel processing by providing designers with two external interface ports, each with a comprehensive memory interface. This yields an immense amount of I/O bandwidth. Devices in the TMS320C4x generation go several steps further by incorporating on-chip hardware to facilitate high-speed interprocessor communication and concurrent I/O without degrading CPU performance. These features, coupled with a host of sophisticated parallel processing development tools, make the TMS320C4x generation of floating-point processors ideal for realtime embedded applications.

## 1.3 TMS320C4x Features

The TMS320C4x generation consists of two equally important aspects, parallel processing devices and parallel processing development tools.

### 1.3.1 TMS320C40 Device Key Features

The Primary features of the TMS320C4x devices are:

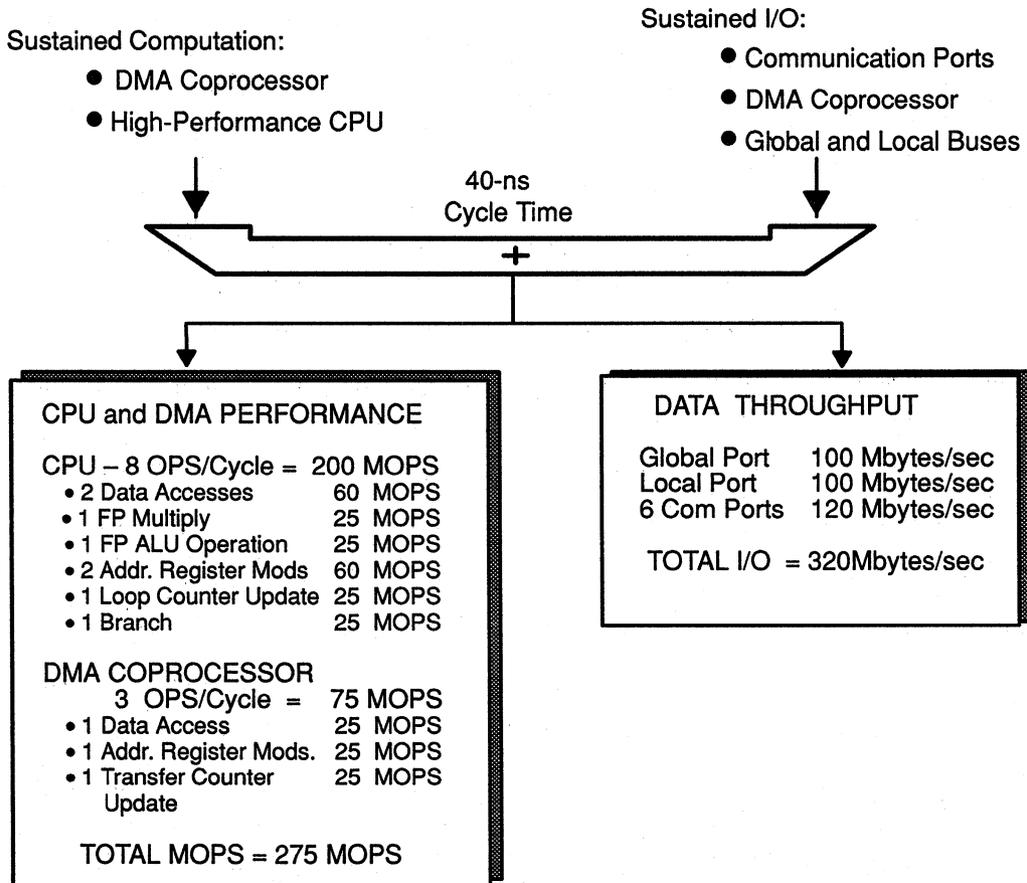
- ❑ Six communication ports for high speed interprocessor communication. Communication port key features include:
  - 20-Mbytes/sec asynchronous transfer rate at each port for maximum data throughput
  - Direct (glueless) processor-to-processor communication for ease of use
  - Bidirectional transfers for maximum communication flexibility
- ❑ Six-channel DMA coprocessor for concurrent I/O and CPU operation, thereby maximizing sustained CPU performance by alleviating the CPU of burdensome I/O. DMA coprocessor key features include:
  - Concurrent data transfers and CPU operation for sustained CPU performance
  - Self-programming (autoinitialize) capability for each channel, thereby not requiring the CPU for initialization, maximizing sustained CPU performance
  - Data transfers to and from anywhere in the processor's memory map for maximum flexibility
- ❑ High-performance DSP CPU capable of 275 MOPS and 320 Mbytes/sec. CPU key features include:
  - Eleven operations per cycle throughput, resulting in massive computing parallelism and sustained CPU performance
  - 40-ns and 50-ns instruction cycle times
  - 40/32-bit single-cycle floating-point/integer multiplier for high performance in computationally intensive algorithms
  - Single-cycle IEEE floating-point conversion for efficient interface to IEEE-compatible processors
  - Hardware divide and inverse square root support for high performance
  - Byte and half-word manipulation capabilities for fast data (un)packing

- Source code compatible with TMS320C3x generation for easy upward and downward mobility
- Support for linear, circular, and bit-reversed addressing for high performance
- Single-cycle branches, calls, and returns for fast program control
- Single-cycle barrel shifter for 0–31 single-cycle right or left shifts for fast bit manipulation
- Relocatable reset and interrupt vectors for easy integration into parallel processing systems
- Two identical external data and address buses supporting shared memory systems and high data rate, single-cycle transfers. Key features include:
  - High port data-transfer rate of 100 Mbytes/sec
  - 16-Gbyte continuous program/data/peripheral address space for maximum design flexibility
  - Status pins that signal type of memory access requested for fast, intelligent bus arbitration in shared memory systems
  - Separate address, data, and control-enable pins for high-speed bus arbitration
  - Four sets of memory-control signals support different speed memories in hardware, enabling efficient use of low- and high-speed memories
- On-chip analysis module supporting efficient, state of the art parallel processing debug. Key features include:
  - Separate breakpoint comparators for program, data, and DMA accesses, providing onchip hardware breakpoint capabilities for fast debug and development
  - Discontinuity stack for hardware trace, facilitating fast debug and development
  - Event counter for accurate benchmarking and profiling
  - JTAG interface for standard system connection

- ❑ On-chip program cache and dual-access/single-cycle RAM for increased memory access performance. On-chip memory key features include:
  - 512-byte instruction cache for increased system performance
  - 8K-bytes of single-cycle dual access program or data RAM for increased system performance and lower system cost
  - Bootloader (ROM based) supporting program bootup via 8-, 16- or 32-bit memories over any one of the communication ports
- ❑ Separate internal program, data, and DMA coprocessor buses for support of massive concurrent I/O of program and data throughput, thereby maximizing sustained CPU performance.

Summed up, the total device performance is 275 MOPS and 320 Mbytes/sec as noted below.

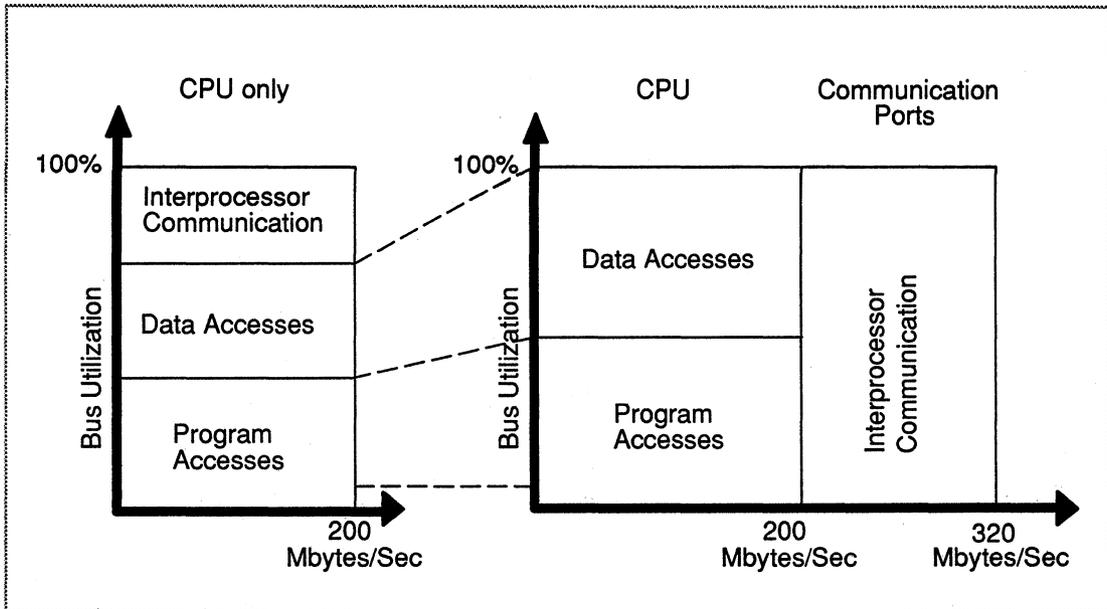
### TMS320C40 Performance



### 1.3.2 Communication Port Benefits

Without the six communication ports, 120 Mbytes/sec of processor throughput must be squeezed over one or both of the external memory interfaces, thereby saturating processor throughput, likewise turning the system into a complex shared memory architecture. With the communication ports, bandwidth is plentiful (illustrated in Figure 1–2).

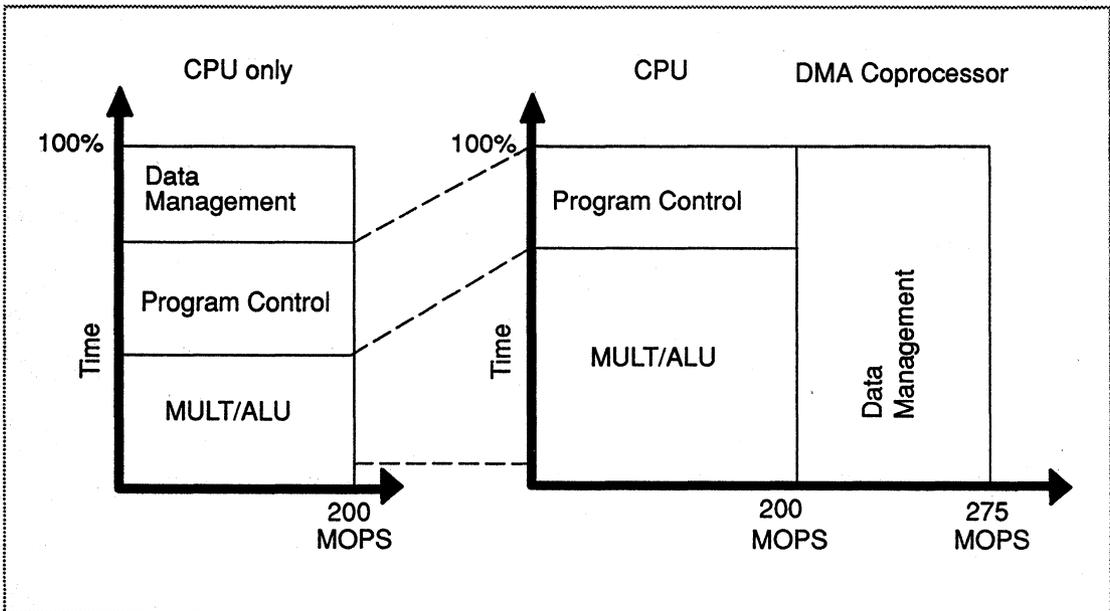
Figure 1–2. TMS320C40 Throughput Increases Use of Communication Ports



### 1.3.3 DMA Coprocessor Benefits

Without the DMA coprocessor, the CPU would have to use computational MOPS to transfer data within the processor's memory map. With the DMA coprocessor, the CPU can focus its entire 200 MOPS of performance on quality computational tasks while the DMA coprocessor takes care of the burdensome I/O. This is illustrated in Figure 1-3.

Figure 1-3. TMS320C40 Throughput Increases Use of DMA Coprocessor



### 1.3.4 TMS320C40 Parallel Processing Development Tools Key Features

The primary TMS320C4x development tools are as follows:

- ❑ Parallel processing in-circuit emulator (XDS510)
  - Able to debug both C and assembly code simultaneously using the graphical user-interface based source-level debugger
  - Can debug any number of TMS320C4x devices in a system with a single XDS510 controller card
  - Can globally stop, start and single step all or any combination of 'C40s in a system.
- ❑ Parallel processing development system
  - Host-independent evaluation board with four 'C40s
  - Each 'C40 connected to every other 'C40 via their communication ports, enabling designers to efficiently test different system topologies
  - Interfaces directly to XDS510 emulator, creating a complete parallel processing development environment.
- ❑ Parallel processing optimizing ANSI C compiler
  - Parallel runtime support library for easy implementation of data and message passing between tasks (or processors) in parallel processing systems
  - C—source and target-specific optimizations for dense, optimal code
  - Plum-Hall validated to ANSI standard for maximum code portability
- ❑ SPOX parallel processing DSP operating system
  - Parallel processing support for easy message passing within a multitasking environment
  - Communication port, DMA coprocessor, and memory interface drivers for fast development of C code without detailed knowledge of the hardware
  - Multitasking real-time kernel for fast implementation of multitasking system
  - DSP math library for fast development of DSP applications (using optimized assembly language routines)
- ❑ Parallel processing assembler/linker
  - Directives to map program and data code on specific processors for fast integration and debug of parallel processing code
  - Relocatable modules for maximum code flexibility

- ❑ Hardware verification and full functional models
  - Simulation of multiple 'C40's and associated logic for accurate development (via software simulation) of parallel processing systems
  - Accurate simulation of device bus cycles and functional execution for fast development of product hardware
  - Supports various workstation and PC environments
- ❑ State accurate simulator
  - Provides cycle-by-cycle simulation of all aspects of the TMS320C4x
  - Low-cost way to simulate key software kernels
  - Supported on a host of workstation and PC platforms

## 1.4 Applications

Below is a list of classical DSP applications along with a number of embedded real-time applications which need the computational performance offered by TMS320 devices. The real time performance, low device costs, and comprehensive development tools are the primary aspects that which make Texas Instruments TMS320 devices the preferred solution in the following applications:

Figure 1-4. Matrix of TMS320 DSP Applications

General-Purpose DSP	Graphics/Imaging	Instrumentation
Digital Filtering Convolution Correlation Hilbert Transforms Fast Fourier Transforms Adaptive Filtering Windowing Waveform Generation	3-D Transformations Rendering Robot Vision Image Transmission/Compression Pattern Recognition Image Enhancement Homomorphic Processing Workstations Animation/Digital Map	Spectrum Analysis Function Generation Pattern Matching Seismic Processing Transient Analysis Digital Filtering Phase-Locked Loops
Voice/Speech	Control	Military
Voice Mail Speech Vocoding Speech Recognition Speaker Verification Speech Enhancement Speech Synthesis Text-to-Speech Neural Networks	Disk Control Servo Control Robot Control Laser Printer Control Engine Control Motor Control Kalman Filtering	Secure Communications Radar Processing Sonar Processing Image Processing Navigation Missile Guidance Radio Frequency Modems Sensor Fusion
Telecommunications		Automotive
Echo Cancellation ADPCM Transcoders Digital PBXs Line Repeaters Channel Multiplexing 1200- to 19200-bps Modems Adaptive Equalizers DTMF Encoding/Decoding Data Encryption	FAX Cellular Telephones Speaker Phones Digital Speech Interpolation (DSI) X.25 Packet Switching Video Conferencing Spread Spectrum Communications	Engine Control Vibration Analysis Antiskid Brakes Adaptive Ride Control Global Positioning Navigation Voice Commands Digital Radio Cellular Telephones
Consumer	Industrial	Medical
Radar Detectors Power Tools Digital Audio/TV Music Synthesizer Toys and Games Solid-State Answering Machines	Robotics Numeric Control Security Access Power Line Monitors Visual Inspection Lathe Control CAM	Hearing Aids Patient Monitoring Ultra Sound Equipment Diagnostic Tools Prosthetics Fetal Monitors MR Imaging



# Architectural Overview

---



---



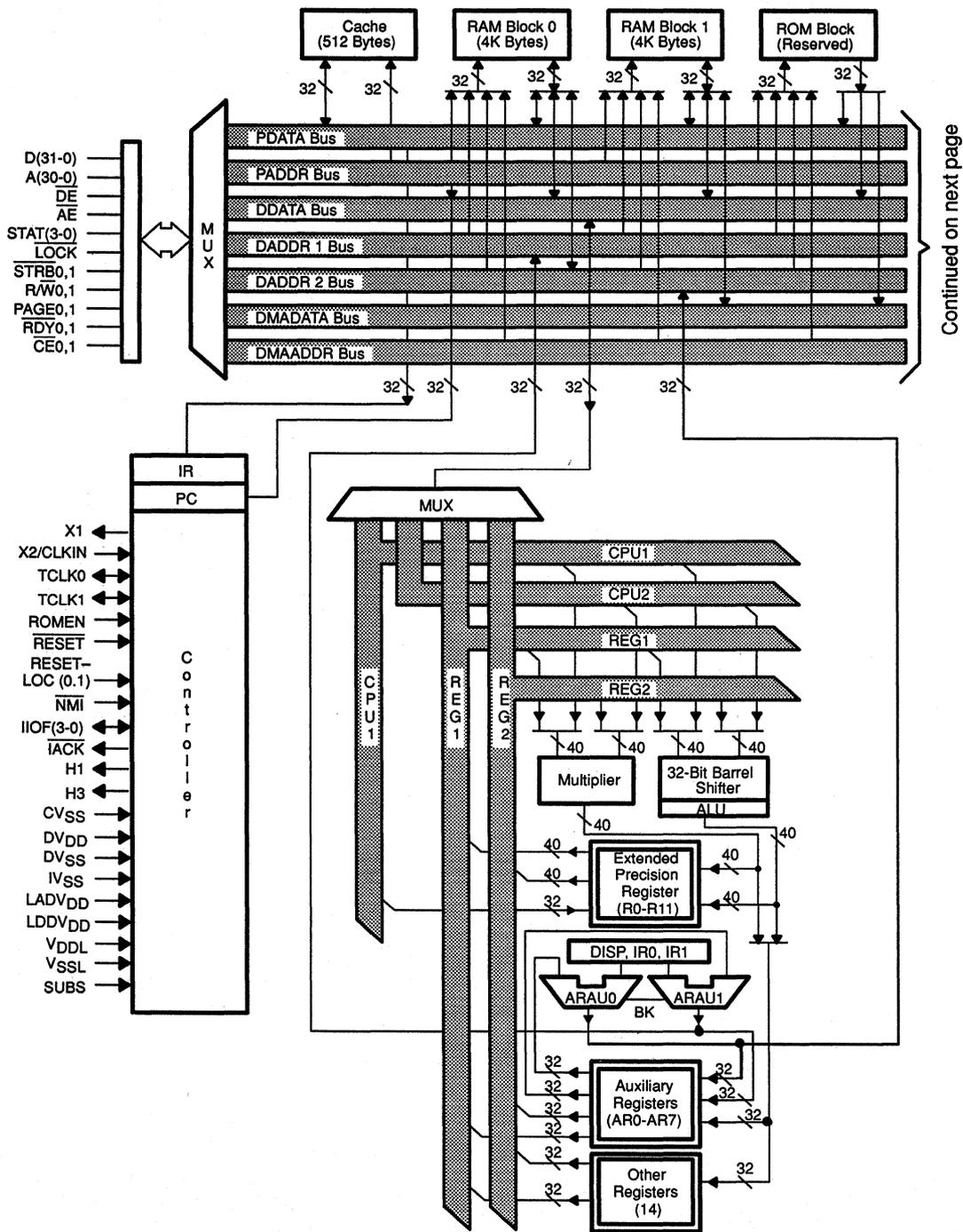
---

The TMS320C40's high performance is achieved through the precision and wide dynamic range of the floating-point units, large on-chip memory, a high degree of parallelism, and the six-channel DMA coprocessor. Figure 2–1, beginning on the next page, is a block diagram of the TMS320C40.

This chapter gives an architectural overview of the TMS320C40 processor. Major areas of discussion are listed below.

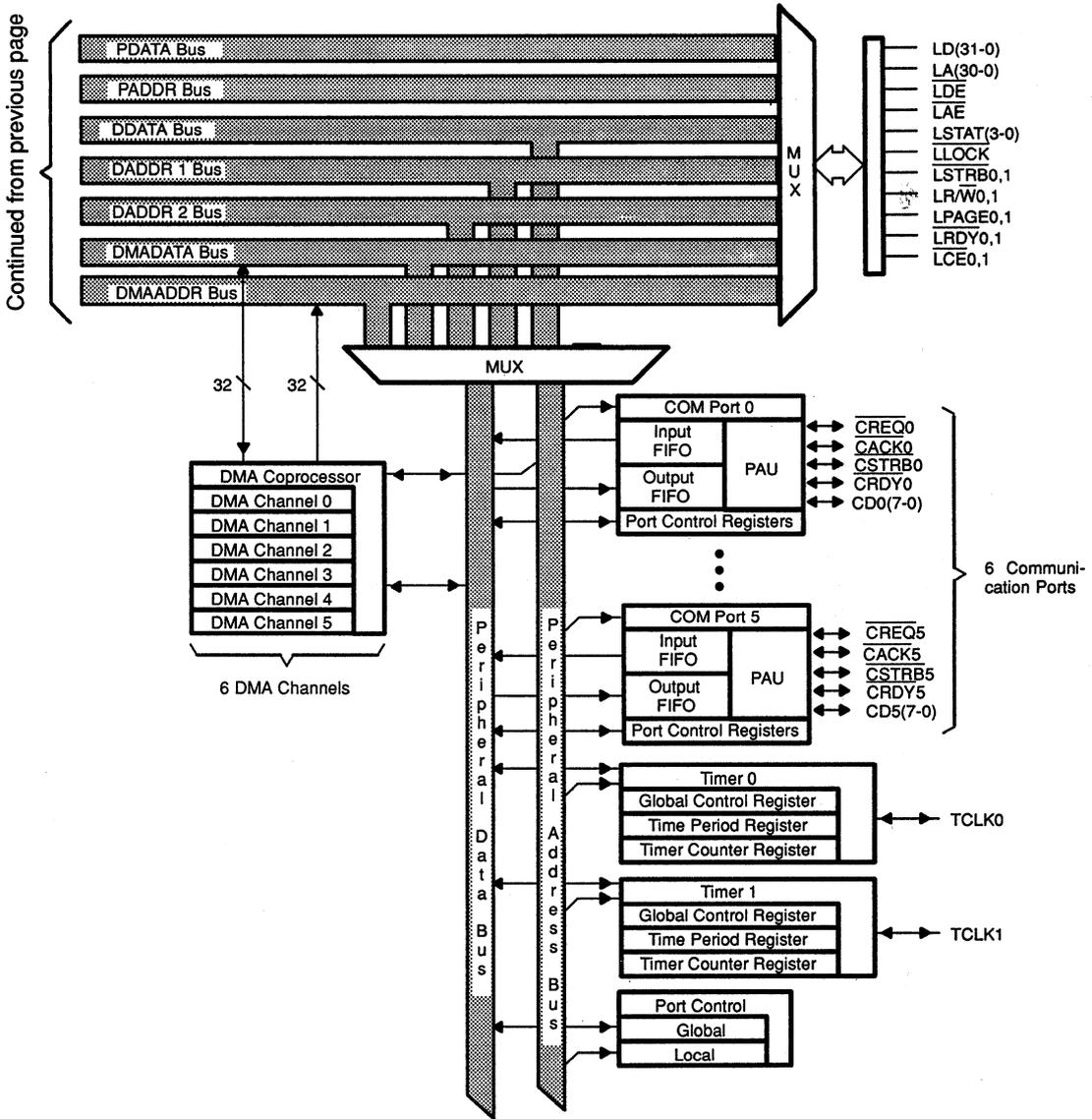
<b>Section</b>	<b>Page</b>
2.1 Central Processing Unit (CPU) .....	2-4
■ Floating-point/integer multiplier .....	2-4
■ ALU for floating-point, integer, and logical operations .	2-4
■ 32-bit barrel shifter .....	2-4
■ Internal buses (CPU1/CPU2 and REG1/REG2) .....	2-4
■ Auxiliary register arithmetic units (ARAUs) .....	2-6
■ Primary register file .....	2-6
■ CPU expansion register file .....	2-9
2.2 Memory Organization) .....	2-10
■ RAM, ROM, and cache .....	2-10
■ Memory maps .....	2-12
■ Memory addressing modes .....	2-15
2.3 Instruction Set Summary .....	2-16
2.4 Internal Bus Operation .....	2-26
2.5 External Bus Operation .....	2-27
2.6 Peripherals .....	2-28
■ Communication ports .....	2-29
■ Direct memory access (DMA) coprocessor .....	2-29
■ Timers .....	2-29

Figure 2-1. TMS320C40 Block Diagram



Continued on next page

Figure 2-1. TMS320C40 Block Diagram (Concluded)



## 2.1 Central Processing Unit (CPU)

The TMS320C40 has a register-based CPU architecture. The CPU comprises the following components:

- ❑ Floating-point/integer multiplier
- ❑ ALU for performing arithmetic: floating-point, integer, and logical operations
- ❑ 32-bit barrel shifter
- ❑ Internal buses (CPU1/CPU2 and REG1/REG2)
- ❑ Auxiliary register arithmetic units (ARAUs)
- ❑ CPU register file

Figure 2–2 shows the various CPU components that are discussed in the succeeding subsections.

### 2.1.1 Multiplier

The multiplier performs single-cycle multiplications on 32-bit integer and 40-bit floating-point values. The TMS320C40 implementation of floating-point arithmetic allows for floating-point operations at fixed-point speeds via a 40-ns instruction cycle and a high degree of parallelism. To gain even higher throughput, you can use parallel instructions to perform a multiply and ALU operation in a single cycle.

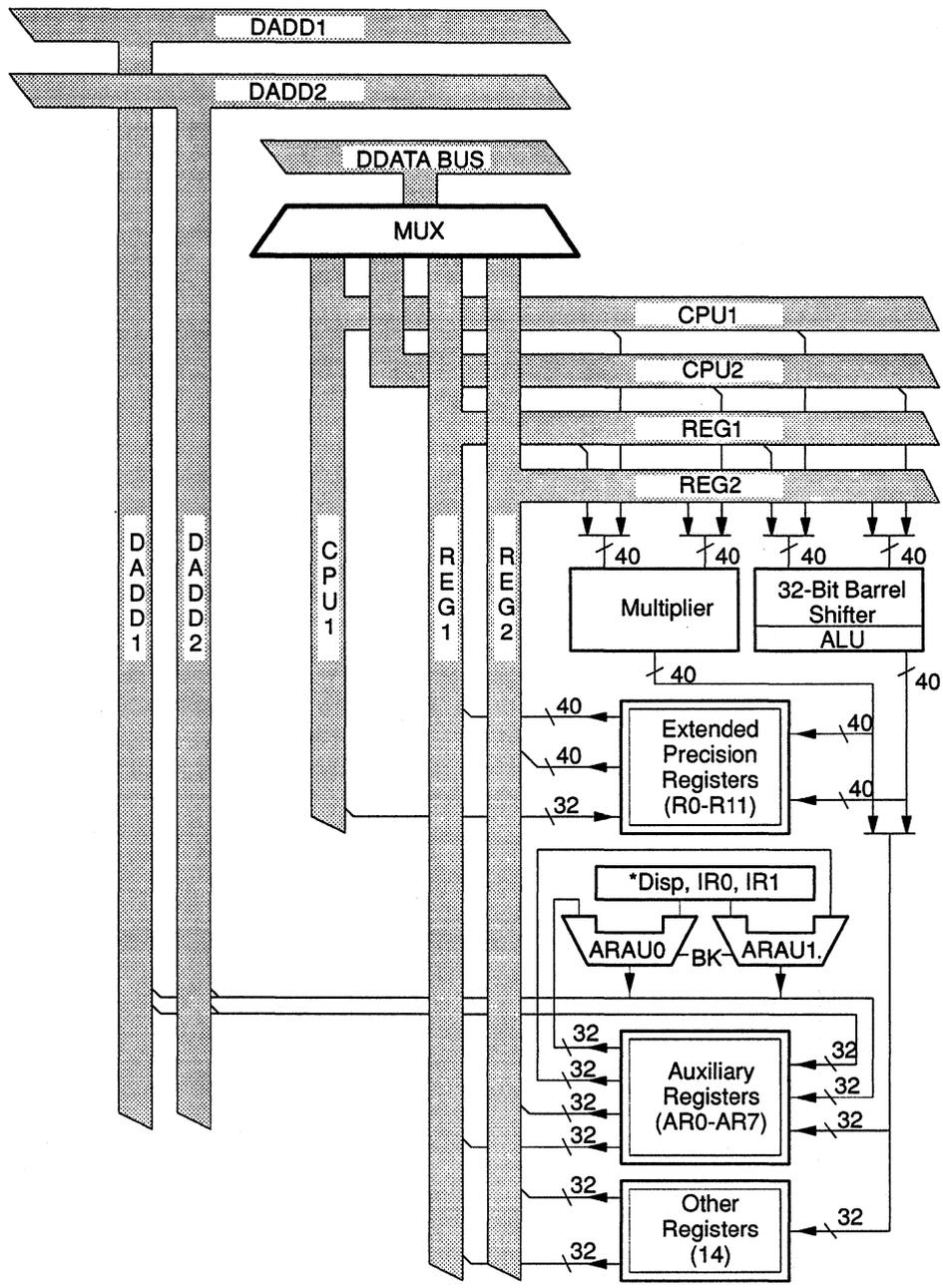
When the multiplier performs floating-point multiplication, the inputs are 40-bit floating-point numbers, and the result is a 40-bit floating-point number. When the multiplier performs integer multiplication, the input data is 32 bits and yields either the 32 most significant bits or 32 least significant bits of the resulting 64-bit product. Refer to Chapter 4 for detailed information on data formats and floating-point operation.

### 2.1.2 Arithmetic Logic Unit (ALU)

The ALU performs single-cycle operations on 32-bit integer, 32-bit logical, and 40-bit floating-point data, including single-cycle integer and floating-point conversions. Results of the ALU are always maintained in 32-bit integer or 40-bit floating-point formats. The barrel shifter is used to shift up to 32 bits left or right in a single cycle.

Internal buses, CPU1/CPU2 and REG1/REG2, carry two operands from memory and two operands from the register file, thus allowing parallel multiplies and adds/subtracts on four integer or floating-point operands in a single cycle.

Figure 2-2. Central Processing Unit (CPU)



\* Disp = an 8-bit integer displacement carried in a program control instruction

### 2.1.3 Auxiliary Register Arithmetic Units (ARAUs)

Two auxiliary register arithmetic units (ARAU0 and ARAU1) can generate two addresses in a single cycle. The ARAUs operate in parallel with the multiplier and ALU. They support addressing with displacements, index registers (IRO and IR1), and circular and bit-reversed addressing. Refer to Chapter 5 for a description of addressing modes.

### 2.1.4 CPU Primary Register File

The TMS320C40 primary register file provides 32 registers in a multiport register file that is tightly coupled to the CPU. Table 2–1 lists register names and functions, followed by the section number and page of each description. (The expansion register file is described in subsection 2.1.5 on page 2-9.)

All of the primary register file registers can be operated upon by the multiplier and ALU, and can be used as general-purpose registers. However, the registers also have some special functions. For example, the 12 extended-precision registers are especially suited for maintaining floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide system functions such as addressing, stack management, processor status, interrupts, and block repeat. Refer to Chapter 3 for detailed information on the CPU registers. Refer to Chapter 5 for register usage in addressing.

The **extended-precision registers (R0–R11)** are capable of storing and supporting operations on 32-bit integer and 40-bit floating-point numbers. Any instruction that assumes the operands are floating-point numbers uses bits 39–0. If the operands are either signed or unsigned integers, only bits 31–0 are used, and bits 39–32 remain unchanged. This is true for all shift operations. Refer to Chapter 4 for extended-precision register formats for floating-point and integer numbers.

The 32-bit **auxiliary registers (AR0–AR7)** can be accessed by the CPU and modified by the two auxiliary register arithmetic units (ARAUs). The primary function of the auxiliary registers is the generation of 32-bit addresses. They can also be used as loop counters or as 32-bit general-purpose registers that can be modified by the multiplier and ALU. Refer to Chapter 5 for detailed information and examples of the use of auxiliary registers in addressing.

Table 2-1. CPU Primary Registers

Assembler Syntax	Assigned Function Name	For Further Description, See:	
		Paragraph	Page
R0	Extended-precision register 0	3.1.1	3-4
R1	Extended-precision register 1	3.1.1	3-4
R2	Extended-precision register 2	3.1.1	3-4
R3	Extended-precision register 3	3.1.1	3-4
R4	Extended-precision register 4	3.1.1	3-4
R5	Extended-precision register 5	3.1.1	3-4
R6	Extended-precision register 6	3.1.1	3-4
R7	Extended-precision register 7	3.1.1	3-4
R8	Extended-precision register 8	3.1.1	3-4
R9	Extended-precision register 9	3.1.1	3-4
R10	Extended-precision register 10	3.1.1	3-4
R11	Extended-precision register 11	3.1.1	3-4
AR0	Auxiliary register 0	3.1.2	3-5
AR1	Auxiliary register 1	3.1.2	3-5
AR2	Auxiliary register 2	3.1.2	3-5
AR3	Auxiliary register 3	3.1.2	3-5
AR4	Auxiliary register 4	3.1.2	3-5
AR5	Auxiliary register 5	3.1.2	3-5
AR6	Auxiliary register 6	3.1.2	3-5
AR7	Auxiliary register 7	3.1.2	3-5
DP	Data-page pointer	3.1.3	3-5
IR0	Index register 0	3.1.4	3-5
IR1	Index register 1	3.1.4	3-5
BK	Block-size register	3.1.5	3-5
SP	System stack pointer	3.1.6	3-5
ST	Status register	3.1.7	3-5
DIE	DMA Coprocessor interrupt enable	3.1.8	3-8
IIE	Internal-interrupt enable register	3.1.9	3-10
IIF	IIOF flag register	3.1.10	3-12
RS	Repeat start address	3.1.11	3-14
RE	Repeat end address	3.1.11	3-14
RC	Repeat counter	3.1.11	3-14

The **data page pointer (DP)** is a 32-bit register. The 16 LSBs of the data page pointer are used by the direct addressing mode as a pointer to the page of data being addressed. The 'C40 can address up to 64K pages, each page containing 64K words. The data page pointer is illustrated in Figure 5-1 on page 5-4.

The 32-bit **index registers** contain the value used by the auxiliary register arithmetic unit (ARAU) to compute an indexed address. Refer to Chapter 5 for examples of the use of index registers in addressing (see subsection 5.1.3, page 5-5, and Section 5.4, page 5-30).

The ARAU uses the 32-bit **block size register (BK)** in circular addressing to specify the data block size. (Circular addressing is described in Section 5.3 on page 5-25.)

The **system stack pointer (SP)** is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. A push performs a preincrement, and a pop performs a postdecrement of the system stack pointer. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH and POP instructions. Refer to Section 5.5, page 5-31, for information about system stack management.

The **status register (ST)** contains global information relating to the state of the CPU. Typically, operations set the condition flags of the status register according to whether the result is zero, negative, etc. This includes register load and store operations as well as arithmetic and logical functions. When the status register is loaded, however, a bit-for-bit replacement is performed with the contents of the source operand, regardless of the state of any bits in the source operand. Therefore, following a load, the contents of the status register are identically equal to the contents of the source operand. This allows the status register to be easily saved and restored. See Table 3-2 on page 3-6 for definitions of the status register bits.

The **DMA coprocessor interrupt enable register (DIE)** is a 32-bit register containing 2- and 3-bit fields to designate the interrupt synchronization scheme for each of the six DMA channels. It allows each DMA channel to service a corresponding input communication port and output communication port. Also, each DMA channel can be synchronized with external interrupts or the on-chip timers. This register is described in subsection 3.1.8 on page 3-8.

The **CPU internal interrupt enable register (IIE)** is also a 32-bit register (described in subsection 3.1.9 on page 3-10). This register enables/disables interrupts for the six communication ports, both timers, and the six DMA coprocessor channels.

The **IIOF flag register (IIF)** controls the function (general-purpose I/O or interrupt) of the four external pins (IIOF0 to IIOF3). Interrupts can be level or edge triggered. Subsection 3.1.10 on page 3-12 provides further description.

The 32-bit **repeat counter (RC)** register specifies the number of times a block of code is to be repeated when performing a block repeat. When the processor is operating in the repeat mode, the 32-bit **repeat start address register (RS)** contains the starting address of the block of program memory to be repeated, and the 32-bit **repeat end address register (RE)** contains the ending address of the block to be repeated. Further information is in subsection 3.1.11 on page 3-14.

The **program counter (PC)** is a 32-bit register containing the address of the next instruction to be fetched. Although the PC is not part of the CPU register

---

file, it is a register that can be modified by instructions that modify the program flow.

### 2.1.5 CPU Expansion Register File

Besides the CPU primary register file (just covered in subsection 2.1.4, starting on page 2-6), the expansion register file contains two special registers that act as pointers:

- IVTP register (points to the interrupt-vector table, which is shown in Figure 3-8 on page 3-16),
- TVTP register (points to the trap vector table (TVT), which defines vectors for 512 interrupts. This is described in Figure 3-7 on page 3-15).

These two registers are fully described in Section 3.2 on page 3-15.

## 2.2 Memory Organization

The total memory reach of the TMS320C40 is 4G (giga or billion) 32-bit words (4 Gbytes). Program memory (on-chip RAM or ROM and external memory) as well as registers affecting timers, communication ports, and DMA channels are contained within this space. This allows tables, coefficients, program code, and data to be stored in either RAM or ROM. Thus, memory usage is maximized, and memory space allocated as desired.

By manipulating one external pin (ROMEN, pin AK4), the first one-mega-word area of memory (0000 0000h to 000F FFFFh) can be configured to be part of the local address bus *or* configured to address the on-chip ROM when using the boot loader (with remaining space reserved). (This is further discussed in Section 3.4 on page 3-18.)

### 2.2.1 RAM, ROM, and Cache

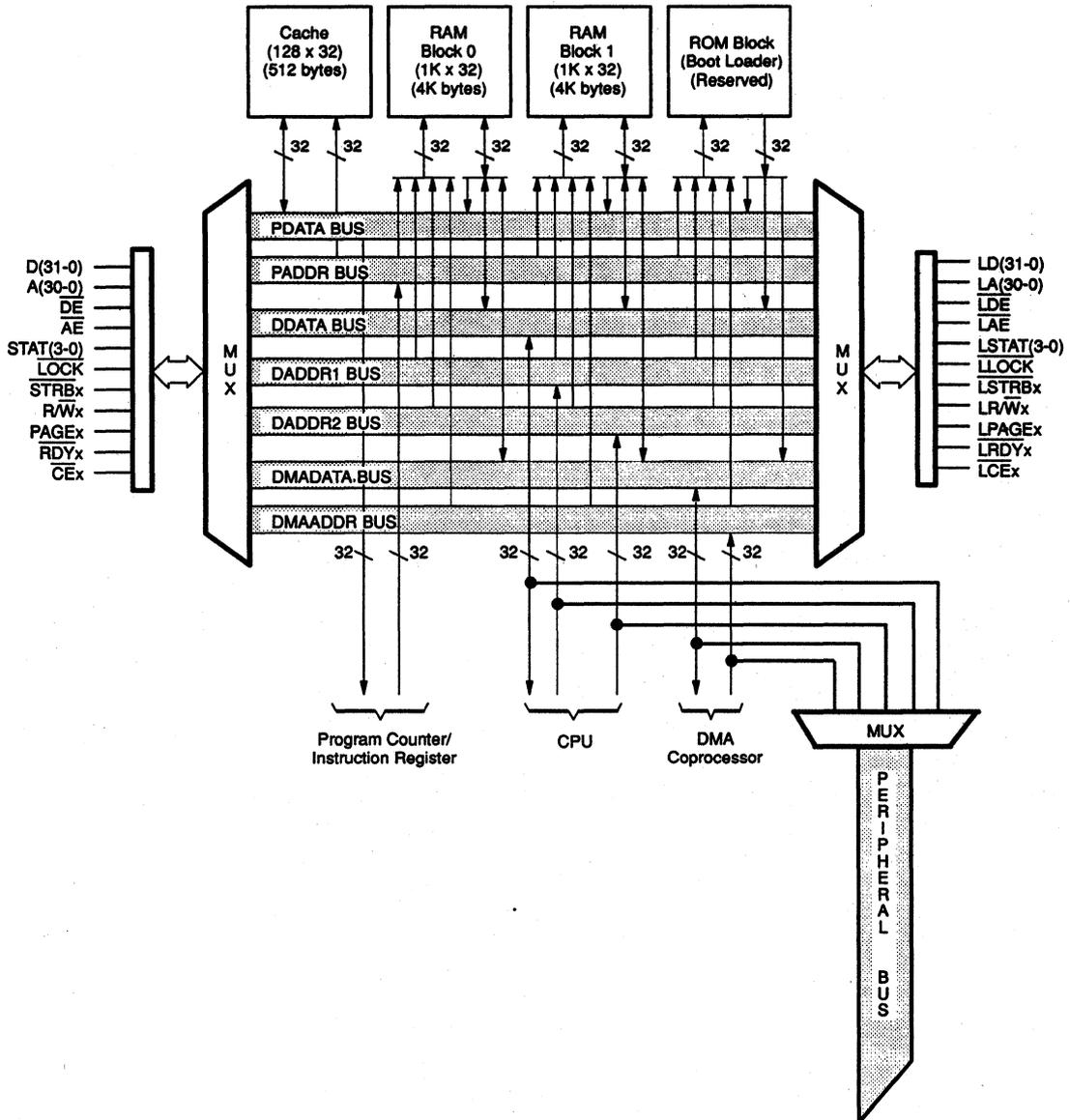
Figure 2–3 shows how the memory is organized on the TMS320C40. RAM blocks 0 and 1 are 4K bytes (1K x 32 bits) each. The ROM block is reserved and contains a boot loader. Each RAM and ROM block is capable of supporting two accesses in a single cycle. The separate program buses, data buses, and DMA buses allow for parallel program fetches, data reads and writes, and DMA operations. For example: the CPU can access two data values in one RAM block and perform an external program fetch in parallel with the DMA coprocessor loading another RAM block, all within a single cycle.

The reserved ROM block (upper right in Figure 2–3) contains a boot loader. This loader supports loading of program and data at reset time. Loading is from 8-, 16-, or 32-bit wide memories or any one of the six communication ports. Section 13.2 (page 13-5) explains the boot loader in detail.

A 128 x 32-bit instruction cache is provided to store often-repeated sections of code, thus greatly reducing the number of needed off-chip accesses. This allows for code to be stored off-chip in slower, lower-cost memories. The external buses are also freed for use by the DMA, external memory fetches, or other devices in the system.

For further information about the memory and instruction cache, refer to Section 3.4 (memory organization — page 3-18) and Section 3.5 (cache memory — page 3-25).

Figure 2-3. Memory Organization



## 2.2.2 Memory Maps

Two memory maps are available as shown in Figure 2–4; the one selected depends upon the level at external pin ROMEN. Both maps in the figure illustrate the 4-gigaword reach of the 'C40; however, they differ in the first 1 megaword of memory in which:

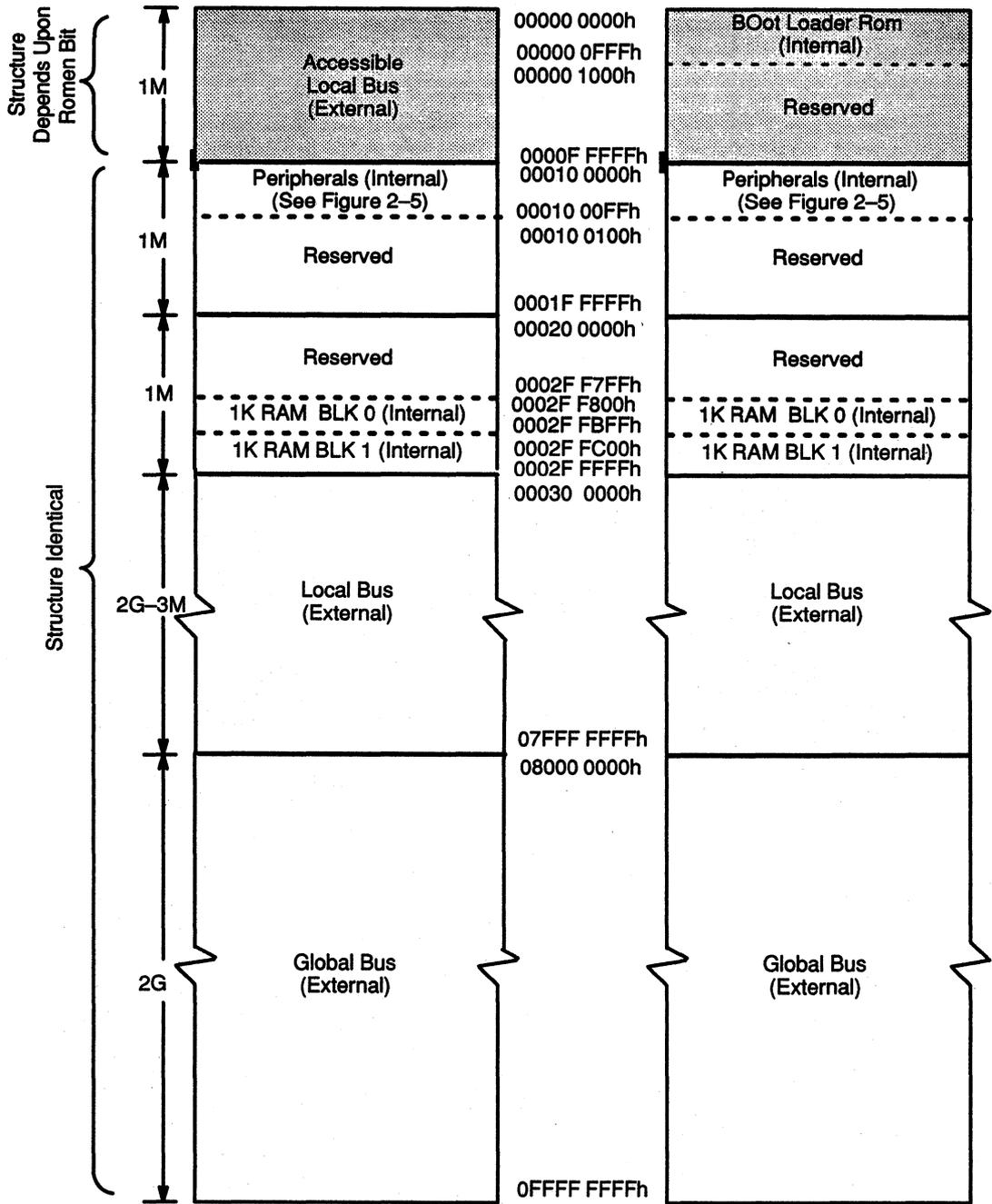
- ❑ A one at external pin ROMEN (pin AK4) causes internal ROM to be enabled at 0000h with the one-megaword space reserved (0000 0000h – 000F FFFFh). This is shown in the right side of the figure.
- ❑ A zero at ROMEN causes addresses 0000 0000h – 000F FFFFh to be accessible on the local bus. This is shown in the left side of the figure.

The rest of the memory map is the same for either level of ROMEN:

- ❑ The second megaword of memory is devoted to peripherals (as shown in Figure 2–5).
- ❑ The third megaword of memory contains the two 1K (4K-byte) blocks of RAM (BLK0 and BLK1 as shown at 002F F800h – 002F FFFFh).
- ❑ The rest of the first 2 gigawords (0030 0000h – 7FFF FFFFh) is on the local bus (external).
- ❑ The second 2 gigawords (8000 0000h – FFFF FFFFh) are on the global bus (external).

Section 3.4 (page 3-18) describes the memory maps in greater detail. Sections 7.1, 7.2, and 7.3, beginning on page 7-3, discuss the local and global interfaces to these memories. The peripheral bus map and the vector locations for reset, interrupts, and traps are also explained.

Figure 2-4. Memory Maps



(a) Internal ROM Disabled (ROMEN = 0)

(b) Internal ROM Enabled (ROMEN = 1)

Figure 2-5. Peripheral Memory Map

2

0010 0000h	Local and Global Port Control (16 words) (See paragraph 3.4.2.1, Figure 3-11)
0010 000Fh	
0010 0010h	Analysis Block Registers (16 words) (See paragraph 3.4.2.2, Figure 3-12)
0010 001Fh	
0010 0020h	Timer 0 Registers (16 words) (See paragraph 3.4.2.3, Figure 3-13, page 3-22)
0010 002Fh	
0010 0030h	Timer 1 Registers (16 words) (See paragraph 3.4.2.3, Figure 3-13, page 3-22)
0010 003Fh	
0010 0040h	Communication Port 0 (16 words) (See paragraph 3.4.2.4, Figure 3-14, page 3-23)
0010 004Fh	
0010 0050h	Communication Port 1 (16 words) (See paragraph 3.4.2.4, Figure 3-14, page 3-23)
0010 005Fh	
0010 0060h	Communication Port 2 (16 words) (See paragraph 3.4.2.4, Figure 3-14, page 3-23)
0010 006Fh	
0010 0070h	Communication Port 3 (16 words) (See paragraph 3.4.2.4, Figure 3-14, page 3-23)
0010 007Fh	
0010 0080h	Communication Port 4 (16 words) (See paragraph 3.4.2.4, Figure 3-14, page 3-23)
0010 008Fh	
0010 0090h	Communication Port 5 (16 words) (See paragraph 3.4.2.4, Figure 3-14, page 3-23)
0010 009Fh	
0010 00A0h	DMA Coprocessor Channel 0 (16 words) (See paragraph 3.4.2.5, Figure 3-15, page 3-24.)
0010 00AFh	
0010 00B0h	DMA Coprocessor Channel 1 (16 words) (See paragraph 3.4.2.5, Figure 3-15, page 3-24.)
0010 00BFh	
0010 00C0h	DMA Coprocessor Channel 2 (16 words) (See paragraph 3.4.2.5, Figure 3-15, page 3-24.)
0010 00CFh	
0010 00D0h	DMA Coprocessor Channel 3 (16 words) (See paragraph 3.4.2.5, Figure 3-15, page 3-24.)
0010 00DFh	
0010 00E0h	DMA Coprocessor Channel 4 (16 words) (See paragraph 3.4.2.5, Figure 3-15, page 3-24.)
0010 00EFh	
0010 00F0h	DMA Coprocessor Channel 5 (16 words) (See paragraph 3.4.2.5, Figure 3-15, page 3-24.)
0010 00FFh	

### 2.2.3 Memory Addressing Modes

The TMS320C40 supports a base set of general-purpose instructions as well as arithmetic-intensive instructions that are particularly suited for digital signal processing and other numeric-intensive applications. Refer to Chapter 5 for detailed information on addressing.

Four groups of addressing modes are provided on the TMS320C40 (major headings below). Each group uses two or more of several different addressing types, as shown for each group in the following list:

- 1) General addressing modes:
  - Register. The operand is a CPU register.
  - Immediate. The operand is a 16-bit immediate value.
  - Direct. The operand is the contents of a 32-bit address (concatenation of 16 bits of the data page pointer and a 16-bit operand).
  - Indirect. A 32-bit auxiliary register indicates the address of the operand.
- 2) Three-operand addressing modes:
  - Register (same as for general addressing mode).
  - Indirect (same as for general addressing mode).
  - Immediate (same as for general addressing mode).
- 3) Parallel addressing modes:
  - Register. The operand is an extended-precision register.
  - Indirect (same as for general addressing mode).
- 4) Branch addressing modes:
  - Register (same as for general addressing mode).
  - PC-relative. A signed 16-bit displacement *or* a 24-bit displacement is added to the PC.

## 2.3 Instruction Set Summary

Table 2–2 lists the TMS320C40 instruction set in alphabetical order. Each table entry shows the instruction mnemonic, description, and operation. Refer to Chapter 11 for a functional listing of the instructions and individual instruction descriptions.

Table 2–2. Instruction Set Summary

Mnemonic	Description	Operation
ABSF	Absolute value of a floating-point number	$ src  \rightarrow Rn$
ABSI	Absolute value of an integer	$ src  \rightarrow Dreg$
ADDC	Add integers with carry	$src + Dreg + C \rightarrow Dreg$
ADDC3	Add integers with carry (3-operand)	$src1 + src2 + C \rightarrow Dreg$
ADDF	Add floating-point values	$src + Rn \rightarrow Rn$
ADDF3	Add floating-point values (3-operand)	$src1 + src2 \rightarrow Rn$
ADDI	Add integers	$src + Dreg \rightarrow Dreg$
ADDI3	Add integers (3-operand)	$src1 + src2 + \rightarrow Dreg$
AND	Bitwise logical-AND	$Dreg \text{ AND } src \rightarrow Dreg$
AND3	Bitwise logical-AND (3-operand)	$src1 \text{ AND } src2 \rightarrow Dreg$
ANDN	Bitwise logical-AND with complement	$Dreg \text{ AND } \overline{src} \rightarrow Dreg$
ANDN3	Bitwise logical-ANDN (3-operand)	$src1 \text{ AND } \overline{src2} \rightarrow Dreg$
ASH	Arithmetic shift	If count $\geq 0$ : (Shifted Dreg left by count) $\rightarrow$ Dreg Else: (Shifted Dreg right by  count ) $\rightarrow$ Dreg
ASH3	Arithmetic shift (3-operand)	If count $\geq 0$ : (Shifted src left by count) $\rightarrow$ Dreg Else: (Shifted src right by  count ) $\rightarrow$ Dreg

### LEGEND:

**src** general addressing modes  
**src1** three-operand addressing modes  
**src2** three-operand addressing modes  
**Csrc** conditional-branch addressing modes  
**Sreg** register address (any register)  
**count** shift value (general addressing modes)  
**SP** stack pointer  
**GIE** global interrupt enable register  
**RM** repeat mode bit  
**TOS** top of stack

**Dreg** register address (any register)  
**Rn** register address (R0 — R11)  
**Daddr** destination memory address  
**ARn** auxiliary register n (AR0 — AR7)  
**cond** condition code (see Table 11–8)  
**ST** status register  
**RE** repeat interrupt register  
**RS** repeat start register  
**PC** program counter  
**C** carry bit

Table 2-2. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
<i>Bcond</i>	Branch conditionally (standard)	If <i>cond</i> = true: If <i>Csrc</i> is a register, <i>Csrc</i> → PC If <i>Csrc</i> is a value, <i>Csrc</i> + PC + 1 → PC Else: PC + 1 → PC
<i>BcondAF</i>	Branch conditionally delayed and annul if false	If <i>cond</i> is true: If <i>src</i> is a register: <i>src</i> → PC If <i>src</i> is a displacement: <i>src</i> + PC of branch + 3 → PC Else: If <i>cond</i> is false, annul execute phase results of next 3 instructions and continue
<i>BcondAT</i>	Branch conditionally delayed and annul if true	If <i>cond</i> is true: If <i>src</i> is a register: <i>src</i> → PC annul execute phase results of next 3 instructions If <i>src</i> is a displacement: <i>src</i> + PC of branch + 3 → PC annul execute phase results of next 3 instructions Else: continue
<i>BcondD</i>	Branch conditionally (delayed)	If <i>cond</i> = true: If <i>Csrc</i> is a register, <i>Csrc</i> → PC If <i>Csrc</i> is a value, <i>Csrc</i> + PC + 3 → PC Else: PC + 1 → PC
BR	Branch unconditionally (standard)	<i>Csrc</i> + PC + 1 → PC
BRD	Branch unconditionally (delayed)	<i>Csrc</i> + PC + 3 → PC
CALL	Call subroutine	PC + 1 → TOS <i>Csrc</i> + PC + 1 → PC
<i>CALLcond</i>	Call subroutine conditionally	If <i>cond</i> = true: PC + 1 → TOS If <i>Csrc</i> is a register, <i>Csrc</i> → PC If <i>Csrc</i> is a value, <i>Csrc</i> + PC → PC Else: PC + 1 → PC
CMPF	Compare floating-point values	Set flags on <i>Rn</i> – <i>src</i>
CMPF3	Compare floating-point values (3-operand)	Set flags on <i>src1</i> – <i>src2</i>
CMPI	Compare integers	Set flags on <i>Dreg</i> – <i>src</i>
CMPI3	Compare integers (3-operand)	Set flags on <i>src1</i> – <i>src2</i>
<i>DBcond</i>	Decrement and branch conditionally (standard)	<i>ARn</i> – 1 → <i>ARn</i> If <i>cond</i> = true and <i>ARn</i> ≥ 0: If <i>Csrc</i> is a register, <i>Csrc</i> → PC If <i>Csrc</i> is a value, <i>Csrc</i> + PC + 1 → PC Else: PC + 1 → PC

Table 2-2. Instruction Set Summary (Continued)

2

Mnemonic	Description	Operation
DBcondD	Decrement and branch conditionally (delayed)	ARn - 1 → ARn If <i>cond</i> = true and ARn ≥ 0: If <i>Csrc</i> is a register, <i>Csrc</i> → PC If <i>Csrc</i> is a value, <i>Csrc</i> + PC + 3 → PC Else: PC + 1 → PC
FIX	Convert floating-point value to integer	Fix ( <i>src</i> ) → Dreg
FLOAT	Convert integer to floating-point value	Float( <i>src</i> ) → Rn
FRIEEE	Convert from IEEE format	Convert <i>src</i> from IEEE format → Dreg
IACK	Interrupt acknowledge	Perform a dummy read with $\overline{IACK} = 0$ At end of dummy read, set IACK = 0
IDLE	Idle until interrupt	PC + 1 → PC, then Idle until next interrupt
LATcond	Link and trap conditionally	If <i>cond</i> is true: ST(GIE) → ST(PGIE) ST(CF) → ST(PCF) 0 → ST(GIE) 1 → ST(CF) PC of LAcond + 4 → R11 trap vector N → PC Else: continue
LAJ	Link and jump	PC + 4 → R11 PC of LAJ + 3 + <i>src</i> → PC
LAJcond	Link and jump conditional	If <i>cond</i> is true and <i>src</i> is a register: PC of LAJcond + 4 → R11 & <i>src</i> → PC If <i>cond</i> is true and <i>src</i> is a displacement: PC of LAJcond + 4 → R11, & <i>src</i> + PC of LAJcond + 3 + → PC Else, continue
LBb	Load byte	Sgn extended byte (byte 3,2,1,0) of <i>src</i> → Dreg
LBUb	Load byte unsigned	Unsigned byte (byte 3,2,1,0) of <i>src</i> → Dreg
LDA	Load address register	<i>src</i> → Dreg
LDE	Load floating-point exponent	<i>src</i> (exponent) → Rn(exponent)
LDEP	Load integer from expansion register file to primary register file	<i>src</i> → Dreg

**LEGEND:**

src general addressing modes  
**src1** three-operand addressing modes  
**src2** three-operand addressing modes  
**Csrc** conditional-branch addressing modes  
**Sreg** register address (any register)  
**count** shift value (general addressing modes)  
**SP** stack pointer  
**GIE** global interrupt enable register  
**RM** repeat mode bit  
**TOS** top of stack

**Dreg** register address (any register)  
**Rn** register address (R0 — R11)  
**Daddr** destination memory address  
**ARn** auxiliary register n (AR0 — AR7)  
**cond** condition code (see Table 11-8)  
**ST** status register  
**RE** repeat interrupt register  
**RS** repeat start register  
**PC** program counter

Table 2–2. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
LDF	Load floating-point value	$src \rightarrow Rn$
LDFcond	Load floating-point value conditionally	If $cond = true$ , $src \rightarrow Rn$ Else: $Rn$ is not changed
LDFI	Load floating-point value, interlocked	Signal interlocked operation $src \rightarrow Rn$
LDHI	Load 16 MSBs with 16-bit immediate	$src \rightarrow 16$ MSBs of $Dreg$
LDI	Load integer	$src \rightarrow Dreg$
LDIcond	Load integer conditionally	If $cond = true$ , $src \rightarrow Dreg$ Else: $Dreg$ is not changed
LDII	Load integer, interlocked	Signal interlocked operation $src \rightarrow Dreg$
LDM	Load floating-point mantissa	$src$ (mantissa) $\rightarrow Rn$ (mantissa)
LDP	Load data page pointer	$src \rightarrow$ data page pointer
LDPE	Load integer from primary register file to expansion register file	$src \rightarrow Dreg$
LDPK	Load data page pointer immediate	$src \rightarrow DP$
LHw	Load half word	Sign-extended half word of $src \rightarrow Dreg$
LHUw	Load half word unsigned	Unsigned half word of $src \rightarrow Dreg$
LSH	Logical shift	If $count \geq 0$ : ( $Dreg$ left-shifted by $count$ ) $\rightarrow Dreg$ Else: ( $Dreg$ right-shifted by $ count $ ) $\rightarrow Dreg$
LSH3	Logical shift (3-operand)	If $count \geq 0$ : ( $src$ left-shifted by $count$ ) $\rightarrow Dreg$ Else: ( $src$ right-shifted by $ count $ ) $\rightarrow Dreg$
LWLct	Load word, left shifted	$src \ll (0,1,2,3)$ bytes and merged with $Dreg \rightarrow Dreg$
LWRct	Load word, right shifted	$src \gg (0,1,2,3)$ bytes and merged with $Dreg \rightarrow Dreg$
MBct	Merge byte, left shifted	8 LSBs of $src \ll (0,1,2,3)$ bytes and merged with $Dreg \rightarrow Dreg$
MHct	Merge half word, left shifted	16 LSBs of $src \ll (0,1)$ half words and merged with $Dreg \rightarrow Dreg$
MPYF	Multiply floating-point values	$src \times Rn \rightarrow Rn$
MPYF3	Multiply floating-point value (3-operand)	$src1 \times src2 \rightarrow Rn$
MPYI	Multiply integers	$src \times Dreg \rightarrow Dreg$
MPYI3	Multiply integers (3-operand)	$src1 \times src2 \rightarrow Dreg$

Table 2-2. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
MPYSHI	Multiply signed integer and produce 32 MSBs	$dst \times src \rightarrow Dreg$
MPYSHI3	Multiply signed integer and produce 32 MSBs, 3 operand	$src1 \times src2 \rightarrow Dreg$
MPYUHI	Multiply unsigned integer and produce 32 MSBs	$Dreg \times src \rightarrow Dreg$
MPYUHI3	Multiply unsigned integer and produce 32 MSBs, 3 operand	$src1 \times src2 \rightarrow Dreg$
NEGB	Negate integer with borrow	$0 - src - C \rightarrow Dreg$
NEGF	Negate floating-point value	$0 - src \rightarrow Rn$
NEGI	Negate integer	$0 - src \rightarrow Dreg$
NOP	No operation	Modify ARn if specified
NORM	Normalize floating-point value	Normalize ( $src$ ) $\rightarrow Rn$
NOT	Bitwise logical-complement	$\overline{src} \rightarrow Dreg$
OR	Bitwise logical-OR	$Dreg \text{ OR } src \rightarrow Dreg$
OR3	Bitwise logical-OR (3-operand)	$src1 \text{ OR } src2 \rightarrow Dreg$
POP	Pop integer from stack	$*SP-- \rightarrow Dreg$
POPF	Pop floating-point value from stack	$*SP-- \rightarrow Rn$
PUSH	Push integer on stack	$Sreg \rightarrow *++ SP$
PUSHF	Push floating-point value on stack	$Rn \rightarrow *++ SP$
RCPF	Reciprocal floating point	16-bit reciprocal of $src \rightarrow dst$
RETS $cond$	Return from subroutine conditionally	If $cond = \text{true or missing}$ : $*SP-- \rightarrow PC$ Else: continue
RND	Round floating-point value	Round ( $src$ ) $\rightarrow Rn$

**LEGEND:**

**src** general addressing modes  
**src1** three-operand addressing modes  
**src2** three-operand addressing modes  
**Csrc** conditional-branch addressing modes  
**Sreg** register address (any register)  
**count** shift value (general addressing modes)  
**SP** stack pointer  
**GIE** global interrupt enable register  
**RM** repeat mode bit  
**TOS** top of stack

**Dreg** register address (any register)  
**Rn** register address (R0 — R11)  
**Daddr** destination memory address  
**ARn** auxiliary register n (AR0 — AR7)  
**cond** condition code (see Table 11-8)  
**ST** status register  
**RE** repeat interrupt register  
**RS** repeat start register  
**PC** program counter  
**C** carry bit

Table 2-2. Instruction Set Summary (Continued)

Mnemonic	Description	Operation
ROL	Rotate left	Dreg rotated left 1 bit → Dreg
ROLC	Rotate left through carry	Dreg rotated left 1 bit through carry → Dreg
ROR	Rotate right	Dreg rotated right 1 bit → Dreg
RORC	Rotate right through carry	Dreg rotated right 1 bit through carry → Dreg
RPTB	Repeat block of instructions	$src \rightarrow RE$ $1 \rightarrow ST (RM)$ Next PC → RS
RPTBD	Repeat block delayed	If $src$ is an immediate value (displacement) $src + PC + 3 \rightarrow RE$ Else: $src \rightarrow RE$ $1 \rightarrow ST (RM)$ PC of RPTBD + 4 → RS
RPTS	Repeat single instruction	$src \rightarrow RC$ $1 \rightarrow ST (RM)$ Next PC → RS Next PC → RE
RSQRF	Reciprocal of square root floating point	16-bit reciprocal of square root of $src \rightarrow Dreg$
SIGI	Signal, interlocked	Signal interlocked operation Wait for interlock acknowledge Clear interlock
STF	Store floating-point value	$Rn \rightarrow Daddr$
STFI	Store floating-point value, interlocked	$Rn \rightarrow Daddr$ Signal end of interlocked operation
STI	Store integer	$Sreg \rightarrow Daddr$
STII	Store integer, interlocked	$Sreg \rightarrow Daddr$ Signal end of interlocked operation
STIK	Store integer immediate value	$src \rightarrow Dreg$
SUBB	Subtract integers with borrow	$Dreg - src - C \rightarrow Dreg$
SUBB3	Subtract integers with borrow (3-operand)	$src1 - src2 - C \rightarrow Dreg$
SUBC	Subtract integers conditionally	If $Dreg - src \geq 0$ : [( $Dreg - src$ ) << 1] OR 1 → Dreg Else: $Dreg << 1 \rightarrow Dreg$

Table 2-2. Instruction Set Summary (Concluded)

Mnemonic	Description	Operation
SUBF	Subtract floating-point values	$Rn - src \rightarrow Rn$
SUBF3	Subtract floating-point values (3-operand)	$src1 - src2 \rightarrow Rn$
SUBI	Subtract integers	$Dreg - src \rightarrow Dreg$
SUBI3	Subtract integers (3-operand)	$src1 - src2 \rightarrow Dreg$
SUBRB	Subtract reverse integer with borrow	$src - Dreg - C \rightarrow Dreg$
SUBRF	Subtract reverse floating-point value	$src - Rn \rightarrow Rn$
SUBRI	Subtract reverse integer	$src - Dreg \rightarrow Dreg$
SWI	Software interrupt	Perform emulator interrupt sequence
TOIEEE	Convert to IEEE format	Convert $src$ to IEEE format $\rightarrow dst$
TRAP $cond$	Trap conditionally	If $cond = true$ or missing: Next PC $\rightarrow * ++ SP$ Trap vector N $\rightarrow PC$ 0 $\rightarrow ST$ (GIE) Else: continue
TSTB	Test bit fields	$Dreg AND src$
TSTB3	Test bit fields (3-operand)	$src1 AND src2$
XOR	Bitwise exclusive-OR	$Dreg XOR src \rightarrow Dreg$
XOR3	Bitwise exclusive-OR (3-operand)	$src1 XOR src2 \rightarrow Dreg$

**LEGEND:**

**src** general addressing modes  
**src1** three-operand addressing modes  
**src2** three-operand addressing modes  
**Csrc** conditional-branch addressing modes  
**Sreg** register address (any register)  
**count** shift value (general addressing modes)  
**SP** stack pointer  
**GIE** global interrupt enable register  
**RM** repeat mode bit  
**TOS** top of stack

**Dreg** register address (any register)  
**Rn** register address (R0 — R11)  
**Daddr** destination memory address  
**ARn** auxiliary register n (AR0 — AR7)  
**addr** 24-bit immediate address (label)  
**cond** condition code (see Table 11-8)  
**ST** status register  
**RE** repeat interrupt register  
**RS** repeat start register  
**PC** program counter  
**C** carry bit

Table 2-3. Parallel Instruction Set Summary

Mnemonic	Description	Operation
Parallel Arithmetic With Store Instructions		
ABSF    STF	Absolute value of a floating-point	src2  → dst1    src3 → dst2
ABSI    STI	Absolute value of an integer	src2  → dst1    src3 → dst2
ADDF3    STF	Add floating-point	src1 + src2 → dst1    src3 → dst2
ADDI3    STI	Add integer	src1 + src2 → dst1    src3 → dst2
AND3    STI	Bitwise logical-AND	src1 AND src2 → dst1    src3 → dst2
ASH3	Arithmetic shift	If count ≥ 0: src2 << count → dst1    src3 → dst2 Else: src2 >>  count  → dst1    src3 → dst2
FIX    STI	Convert floating-point to integer	Fix(src2) → dst1    src3 → dst2
FLOAT    STF	Convert integer to floating-point	Float(src2) → dst1    src3 → dst2
FRIIEEE    STF	Parallel FRIIEEE and STF	Convert src2 from IEEE format → dst1 in parallel with src3 → dst2
LDF    STF	Load floating-point	src2 → dst1    src3 → dst2
LDI    STI	Load integer	src2 → dst1    src3 → dst2
LSH3	Logical shift	If count ≥ 0: src2 << count → dst1    src3 → dst2 Else: src2 >>  count  → dst1    src3 → dst2

**LEGEND (for parallel instructions):**

src1 register addr (R0 — R11)  
 src3 register addr (R0 — R11)  
 dst1 register addr (R0 — R11)  
 op3 – register addr (R0 or R1)

src2 indirect addr (disp = 0, 1, IR0, IR1)  
 src4 indirect addr (disp = 0, 1, IR0, IR1)  
 dst2 indirect addr (disp = 0, 1, IR0, IR1)  
 op6 register addr (R2 or R3)

op1, op2, op4, op5 – Two of these operands must be specified using register addr, and two must be specified using indirect.

Table 2-3. Parallel Instruction Set Summary (Continued)

Mnemonic	Description	Operation
MPYF3    STF	Multiply floating-point and store	$src1 \times src2 \rightarrow dst1$    $src3 \rightarrow dst2$
MPYI3    STI	Multiply integer	$src1 \times src2 \rightarrow dst1$    $src3 \rightarrow dst2$
NEGF    STF	Negate floating-point	$0 - src2 \rightarrow dst1$    $src3 \rightarrow dst2$
TOIEE    STF	Convert to IEEE floating point format	convert $src2$ to IEEE format $\rightarrow dst1$    $src3 \rightarrow dst2$
Parallel Arithmetic With Store Instructions (Concluded)		
NEGI    STI	Negate integer	$0 - src2 \rightarrow dst1$    $src3 \rightarrow dst2$
NOT    STI	Complement	$\overline{src1} \rightarrow dst1$    $src3 \rightarrow dst2$
OR3    STI	Bitwise logical-OR	$src1 \text{ OR } src2 \rightarrow dst1$    $src3 \rightarrow dst2$
STF    STF	Store floating-point	$src1 \rightarrow dst1$    $src3 \rightarrow dst2$
STI    STI	Store integer	$src1 \rightarrow dst1$    $src3 \rightarrow dst2$
SUBF3    STF	Subtract floating-point	$src1 - src2 \rightarrow dst1$    $src3 \rightarrow dst2$
SUBI3    STI	Subtract integer	$src1 - src2 \rightarrow dst1$    $src3 \rightarrow dst2$
XOR3    STI	Bitwise exclusive-OR	$src1 \text{ XOR } src2 \rightarrow dst1$    $src3 \rightarrow dst2$

**LEGEND (for parallel instructions):**

**src1** register addr (R0 — R11)  
**src3** register addr (R0 — R11)  
**dst1** register addr (R0 — R11)  
**op3** register addr (R0 or R1)

**src2** indirectaddr (disp = 0, 1, IR0, IR1)  
**src4** indirect addr (disp = 0, 1, IR0, IR1)  
**dst2** indirect addr (disp = 0, 1, IR0, IR1)  
**op6** register addr (R2 or R3)

**op1,op2,op4,op5** – Two of these operands must be specified using register addr, and two must be specified using indirect.

Table 2-3. Parallel Instruction Set Summary (Concluded)

Mnemonic	Description	Operation
Parallel Load Instructions		
LDF    LDF	Load floating-point	$src2 \rightarrow dst1$    $src4 \rightarrow dst2$
LDF    STF	Load floating point and store floating point	$src2 \rightarrow dst1$    $src3 \rightarrow dst2$
LDI    LDI	Load integer	$src2 \rightarrow dst1$    $src4 \rightarrow dst2$
LSH3    STI	Logical shift, 3 operand, and store integer	If $count \geq 0$ : $src2 \ll count \rightarrow dst1$ Else: $src2 \gg  count  \rightarrow dst1$    $src3 \rightarrow dst2$
LSH3    STI	Logical shift 3 and store integer	$src2 \rightarrow dst1$    $src3 \rightarrow dst2$
Parallel Multiply And Add/Subtract Instructions		
MPYF3    ADDF3	Multiply and add floating-point	$op1 \times op2 \rightarrow op3$    $op4 + op5 \rightarrow op6$
MPYF3    SUBF3	Multiply and subtract floating-point	$op1 \times op2 \rightarrow op3$    $op4 - op5 \rightarrow op6$
MPYI3    ADDI3	Multiply and add integer	$op1 \times op2 \rightarrow op3$    $op4 + op5 \rightarrow op6$
MPYI3    SUBI3	Multiply and subtract integer	$op1 \times op2 \rightarrow op3$    $op4 - op5 \rightarrow op6$

**LEGEND (for parallel instructions):****src1** register addr (R0 — R11)**src3** register addr (R0 — R11)**dst1** register addr (R0 — R11)**op3** — register addr (R0 or R1)**src2** indirectaddr (disp = 0, 1, IR0, IR1)**src4** indirect addr (disp = 0, 1, IR0, IR1)**dst2** indirect addr (disp = 0, 1, IR0, IR1)**op6** register addr (R2 or R3)**op1, op2, op4, op5** — Two of these operands must be specified using register addr, and two must be specified using indirect.

## 2.4 Internal Bus Operation

2

A large portion of the TMS320C40's high performance is due to internal busing and parallelism. Separate buses allow for parallel program fetches, data accesses, and DMA accesses:

- ❑ **program buses** PADDR and PDATA
- ❑ **data buses** DADDR1, DADDR2, and DDATA
- ❑ **DMA buses** DMAADDR and DMADATA

These buses connect all of the physical spaces (on-chip memory, off-chip memory, and on-chip peripherals) supported by the TMS320C40. Figure 2–3 shows these internal buses and their connection to on-chip and off-chip memory blocks.

The program counter (PC) is connected to the 32-bit program address bus (PADDR). The instruction register (IR) is connected to the 32-bit program data bus (PDATA). These buses can fetch a single instruction word every machine cycle.

The 32-bit data address buses (DADDR1 and DADDR2) and the 32-bit data data bus (DDATA) support two data memory accesses every machine cycle. The DDATA bus carries data to the CPU over the CPU1 and CPU2 buses. The CPU1 and CPU2 buses can carry two data memory operands to the multiplier, ALU, and register file every machine cycle. Also internal to the CPU are register buses REG1 and REG2, which can carry two data values from the register file to the multiplier and ALU every machine cycle. Figure 2–2 shows the buses internal to the CPU section of the processor.

The DMA controller is supported with a 32-bit address bus (DMAADDR) and a 32-bit data bus (DMADATA). These buses allow the DMA to perform memory accesses in parallel with the memory accesses occurring from the data and program buses.

## 2.5 External Bus Operation

The TMS320C40 provides two identical external interfaces: the global memory interface and the local memory interface. Each consists of a 32-bit data bus, a 31-bit address bus, and two sets of control signals. Both buses can be used to address external program/data memory or I/O space. The buses also have external  $\overline{\text{RDY}}$  signals for wait-state generation with wait states inserted under software control. Chapter 7 covers external bus operation.

### 2.5.1 Interrupts

The TMS320C40 supports four external interrupts (IIOF3–0), a number of internal interrupts, a nonmaskable, external NMI interrupt, and a nonmaskable external RESET signal, which sets the processor to a known state. The DMA and communication ports have their own internal interrupts. When the CPU responds to the interrupt, the  $\overline{\text{ACK}}$  pin can be used to signal an external interrupt acknowledge. Section 6.7 (beginning on page 6-23) covers RESET and interrupt processing.

### 2.5.2 Interlocked Instructions

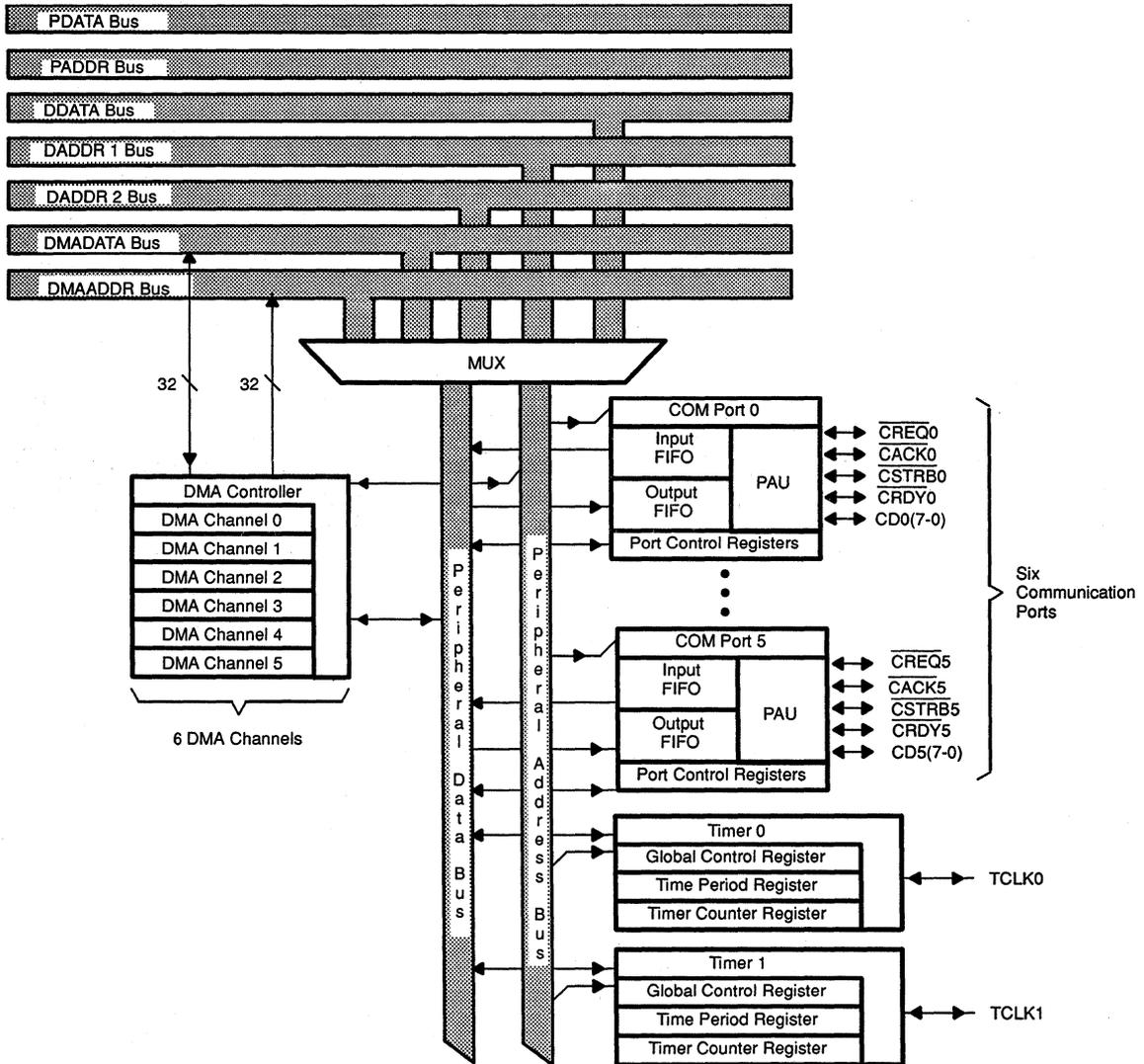
In order for multiple processors to access global memory and share data in a coherent manner, arbitration is necessary. This arbitration (handshaking) is the purpose of the TMS320C40's interlocked operations, handled through the Interlocked instructions (explained in Section 6.4 on page 6-11).

## 2.6 Peripherals

2

All TMS320C40 peripherals are controlled through memory-mapped registers on a dedicated peripheral bus. This peripheral bus is composed of a 32-bit data bus and a 32-bit address bus. This peripheral bus permits straightforward communication to the peripherals. The TMS320C40 peripherals include two timers and two serial ports. Figure 2-6 shows the peripherals with associated buses and signals.

Figure 2-6. Peripheral Modules



## 2.6.1 Communication Ports

Six high-speed communication ports provide rapid processor-to-processor communication through each port's dedicated communication interfaces. Coupled with the 'C40's two memory interfaces (global and local), this allows you to construct a parallel processor system that attains optimum system performance by the distributing of tasks among several processors. Each 'C40 can pass the results of its work to another, enabling each 'C40 to continue working. Chapter 8 explains communication port operation in detail.

Communication port features:

- ❑ 160-megabit per second (20-Mbytes or 5-Mwords per second) bidirectional data transfer operations (at 40-ns cycle time)
- ❑ direct (glueless) processor-to-processor communication via eight data lines and four control lines
- ❑ buffering of all data transfers, both input and output
- ❑ automatic arbitration provided to ensure communication synchronization
- ❑ synchronization between the CPU or direct-memory access (DMA) coprocessor and the six communication ports via internal interrupts and internal ready signals.

## 2.6.2 Direct Memory Access (DMA)

The six channels of the on-chip Direct Memory Access (DMA) coprocessor can read from or write to any location in the memory map without interfering with the operation of the CPU. This allows interfacing to slow external memories and peripherals without reducing throughput to the CPU. The DMA coprocessor contains its own address generators, source and destination registers, and transfer counter. Dedicated DMA address and data buses allow for minimization of conflicts between the CPU and the DMA coprocessor. A DMA operation consists of a block or single-word transfer to or from memory. A key feature of the DMA coprocessor is its ability to automatically reinitialize each channel following a data transfer. Refer to Chapter 9 for detailed information on the DMA coprocessor.

## 2.6.3 Timers

The two timer modules are general-purpose 32-bit timer/event counters with two signaling modes and internal or external clocking. They can signal internally to the 'C40 or externally to the outside world at specified intervals, or they can count external events. Each timer has an I/O pin that can be used as an input clock to the timer, as an output signal driven by the timer, or as a general-purpose I/O pin. Timers are described in detail in Section 9.10 on page 9-45.



## CPU Registers, Memory, and Cache

The CPU **primary register file** contains 32 registers that can be used as operands by the multiplier and ALU (arithmetic logic unit). The register file includes the auxiliary registers, extended-precision registers, and index registers. These registers support addressing, floating-point/integer operations, stack management, processor status, block repeats, branching, and interrupts.

The CPU **expansion register file** contains two registers — the interrupt vector table pointer (IVTP) and the trap vector table pointer (TVTP).

The TMS320C40 accesses a total memory space of 4G (giga = 1 billion) 32-bit words (16 gigabytes) of program, data, and I/O space. Two internal RAM blocks of 1K × 32 bits each (4K bytes) and an internal ROM block containing a boot loader permit two accesses per block in a single cycle.

A 128 × 32-bit instruction cache stores often-repeated sections of code. The cache greatly reduces the number of off-chip accesses, allowing code to be stored off-chip in slower, lower-cost memories without degrading performance. The cache also speeds data fetches to the same physical space as the program by not burdening the bus with program instruction fetches. Three bits in the CPU status register control the clear, enable, or freeze of the cache.

This chapter describes in detail each of the CPU registers, the memory maps, and the instruction cache. Major topics are as follows:

Section	Page
3.1 CPU Primary Register File .....	3-3
■ Extended-Precision Registers (R0–R11) .....	3-4
■ Auxiliary Registers (AR0–AR7) .....	3-5
■ Data-Page Pointer (DP) .....	3-5
■ Index Registers (IR0, IR1) .....	3-5
■ Block-Size Register (BK) .....	3-5
■ System Stack Pointer (SP) .....	3-5

<b>Section</b>	<b>Page</b>
■ Status Register (ST) .....	3-5
■ DMA Interrupt Enable Register (DIE) .....	3-8
■ Internal Interrupt Enable Register (IIE) .....	3-10
■ Interrupt Flag Register (IIF) Controls External Pins IIOF(3 – 0),Timer/DMA Flags .....	3-12
■ Block-Repeat (RS, RE) and Repeat-Count (RC) Registers .....	3-14
■ Program Counter (PC) .....	3-14
■ Reserved Bits and Compatibility .....	3-14
3.2 CPU Expansion Register File .....	3-15
■ CPU Expansion Registers .....	3-15
■ Trap Vector Table (TVT) .....	3-15
3.3 Reset Vector Mapping .....	3-17
3.4 Memory .....	3-18
■ Memory Maps .....	3-19
■ Peripheral Bus Memory Map .....	3-20
3.5 Instruction Cache Architecture .....	3-25
■ Cache Algorithm .....	3-27
■ Cache and System Memory .....	3-28
■ Cache Control Bits .....	3-29

### 3.1 CPU Primary Register File

The TMS320C40 provides 32 registers in a multiport register file that is tightly coupled to the CPU. **The PC (program counter) is not included in the 32 registers.** The registers' names and assigned function are listed in Table 3-1.

All of these registers can be used as operands by the multiplier and ALU, and can be used as general-purpose 32-bit registers. However, the registers also have some special functions for which they are particularly appropriate. For example, the 12 extended-precision registers are especially

3

Table 3-1. CPU Primary Register File

Assembler Syntax	Register Machine Value (hex)	Assigned Function Name	See Paragraph	On Page
R0	00	Extended-precision register 0	3.1.1	3-4
R1	01	Extended-precision register 1	3.1.1	3-4
R2	02	Extended-precision register 2	3.1.1	3-4
R3	03	Extended-precision register 3	3.1.1	3-4
R4	04	Extended-precision register 4	3.1.1	3-4
R5	05	Extended-precision register 5	3.1.1	3-4
R6	06	Extended-precision register 6	3.1.1	3-4
R7	07	Extended-precision register 7	3.1.1	3-4
R8	1C	Extended-precision register 8	3.1.1	3-4
R9	1D	Extended-precision register 9	3.1.1	3-4
R10	1E	Extended-precision register 10	3.1.1	3-4
R11	1F	Extended-precision register 11	3.1.1	3-4
AR0	08	Auxiliary register 0	3.1.2	3-5
AR1	09	Auxiliary register 1	3.1.2	3-5
AR2	0A	Auxiliary register 2	3.1.2	3-5
AR3	0B	Auxiliary register 3	3.1.2	3-5
AR4	0C	Auxiliary register 4	3.1.2	3-5
AR5	0D	Auxiliary register 5	3.1.2	3-5
AR6	0E	Auxiliary register 6	3.1.2	3-5
AR7	0F	Auxiliary register 7	3.1.2	3-5
DP	10	Data-page pointer	3.1.3	3-5
IR0	11	Index register 0	3.1.4	3-5
IR1	12	Index register 1	3.1.4	3-5
BK	13	Block-size register	3.1.5	3-5
SP	14	System stack pointer	3.1.6	3-5
ST	15	Status register	3.1.7	3-5
DIE	16	DMA coprocessor interrupt enable	3.1.8	3-8
IIE	17	Internal-interrupt enable register	3.1.9	3-10
IIF	18	IIOF flag register (IIOF3-0, timers, DMA)	3.1.10	3-12
RS	19	Repeat start address	3.1.11	3-14
RE	1A	Repeat end address	3.1.11	3-14
RC	1B	Repeat counter	3.1.11	3-14

well suited for maintaining extended-precision floating-point results. The eight auxiliary registers support a variety of indirect addressing modes and can be used as general-purpose 32-bit integer and logical registers. The remaining registers provide system functions such as addressing, stack management, processor status, interrupts, and block repeat. Refer to Chapter 5 for detailed information and examples of the use of CPU registers in addressing.

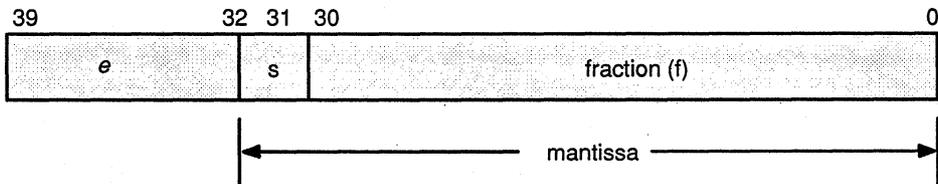
### 3.1.1 Extended-Precision Registers (R0–R11)

The 12 extended-precision registers (R0–R11) can store and support operations on 32-bit integer and 40-bit floating-point numbers. These registers consist of two separate and distinct regions:

- bits 39–32: dedicated to storage of the exponent (*e*) of the floating-point number.
- bits 31–0: store the mantissa of the floating-point number:
  - bit 31: sign bit (*s*),
  - bits 30–0: the fraction (*f*).

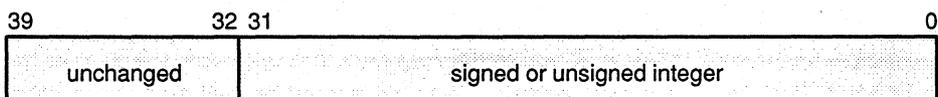
Any instruction that assumes the operands are floating-point numbers uses bits 39–0. Figure 3–1 illustrates the storage of 40-bit floating-point numbers in the extended-precision registers.

Figure 3–1. Extended-Precision Register Floating-Point Format



For integer operations, bits 31–0 of the extended-precision registers contain the integer (signed or unsigned). Any instruction that assumes the operands are either signed or unsigned integers uses only bits 31–0. Bits 39–32 remain unchanged. This is true for all shift operations. The storage of 32-bit integers in the extended-precision registers is shown in Figure 3–2.

Figure 3–2. Extended-Precision Register Integer Format



### 3.1.2 Auxiliary Registers (AR0–AR7)

The eight 32-bit auxiliary registers (AR0–AR7) can be accessed by the CPU and modified by the two auxiliary register arithmetic units (ARAUs). The primary function of the auxiliary registers is the generation of 32-bit addresses. However, they can also operate as loop counters in indirect addressing or as 32-bit general-purpose registers that can be modified by the multiplier and ALU. Refer to Chapter 5 for detailed information and examples of the use of auxiliary registers in addressing.

### 3.1.3 Data-Page Pointer (DP)

The data-page pointer (DP) is a 32-bit register whose 16 LSBs are used by the direct addressing mode as a pointer to the page of data being addressed. Data pages are 64K words long with a total of 64K (65,536) pages. Bits 31–16 are reserved; they are always read as zeroes and **should not be modified by writing to the register**. The DP can be loaded by using the LDP pseudo-instruction or the LDI instruction. Figure 5–1 on page 5-4 describes this register's function.

### 3.1.4 Index Registers (IR0, IR1)

The 32-bit index registers (IR0 and IR1) are used by the auxiliary register arithmetic unit (ARAU) for indexing the address. IR0 is also used for bit-reversed addressing. Refer to Chapter 5 for detailed information and examples of the use of index registers in addressing. (Subsection 5.1.3 on page 5-5 covers use of the IR in indirect addressing; see the examples starting on page 5-12. Section 5.4 on page 5-30 describes using it with bit-reversed addressing).

### 3.1.5 Block-Size Register (BK)

The 32-bit block-size register (BK) is used by the ARAU in circular addressing to specify the data block size (see Section 5.3 on page 5-25).

### 3.1.6 System Stack Pointer (SP)

The system stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The SP always points to the last element pushed onto the stack. The SP is manipulated by interrupts, traps, calls, returns, and the PUSH, PUSHF, POP, and POPF instructions. Pushes and pops of the stack perform preincrement and postdecrement, respectively, on all 32 bits of the SP. Refer to Section 5.5 on page 5-31 for information about system stack management.

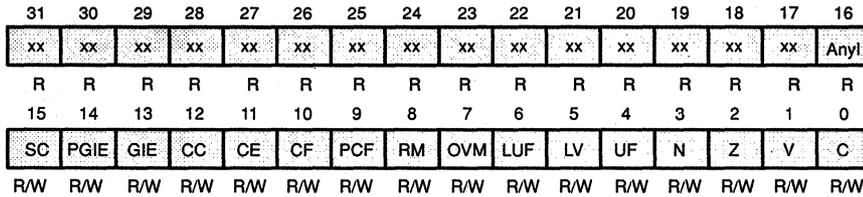
### 3.1.7 Status Register (ST)

The status register (ST) contains global information relating to the CPU state. Typically, operations set the condition flags of the status register ac-

ording to whether the result is zero, negative, etc. This includes register load and store operations as well as arithmetic and logical functions. However, when the ST is loaded, the contents of the load instruction’s source operand replace the ST current contents bit for bit, regardless of the state of any bit(s) in the source operand. Therefore, following an ST load, the contents of the ST are identical to the contents of the source operand. This allows the status register to be saved easily and restored. At system reset, 0 is written to this register.

The format of the status register is shown in Figure 3–3. Table 3–2 defines the status register bits, their names, and functions.

Figure 3–3. Status Register



NOTE: xx = reserved bit.  
R = read, W = write.

Table 3–2. Status Register Bits Summary

Bit	Bit Field Name	Function
0†	C	Carry condition flag
1†	V	Overflow condition flag
2†	Z	Zero condition flag
3†	N	Negative condition flag
4†	UF	Floating-point underflow condition flag
5†	LV	Latched overflow condition flag
6†	LUF	Latched floating-point underflow condition flag
7	OVM	Overflow mode flag. This flag affects only integer operations. If OVM = 0, the overflow mode is turned off; integer results that overflow are treated in no special way. If OVM = 1, a) integer results overflowing in the positive direction are set to the most positive 32-bit twos-complement number (7FFF FFFFh). b) integer results overflowing in the negative direction are set to the most negative 32-bit twos-complement number (8000 0000h). Note that the functions of bits V and LV are independent of the setting of OVM.
8	RM	Repeat mode flag. If RM = 1, the PC is being modified in either the repeat-block or repeat-single mode.

† The seven condition flags (ST bits 0 – 6) are defined in Section 11.2 on page 11-10.

Table 3–2. Status Register Bits Summary (Continued)

Bit	Bit Field Name	Function															
9	PCF	Previous state of bit CF. When a trap executes or an interrupt is taken, bit CF is set to 1. When this occurs, the PCF bit is set to the CF bit's value before the trap or interrupt. Note that the RETI and RETID instructions copy PCF to the CF bit.															
10	CF	Cache freeze. Set CF = 1 to freeze cache (cache is not updated) including LRU (least recently used) stack manipulation. If the cache is enabled (CE = 1), fetches from the cache are allowed, but modification of the cache contents is not allowed. Cache clearing (CC=1) is allowed. At reset, this bit is set to zero. When CF=0, cache clearing (CC=1) is allowed. CF is set to one when a trap or interrupt is taken. Also, the RETI and RETID instructions copy PCF to the CF bit.															
11	CE	Cache enable. Set CE = 1 to enable the cache, allowing the cache to be used according to the LRU (least recently used) cache algorithm. Set CE = 0 to disable the cache; preventing cache updates or modifications (thus, no cache fetches can be made). At reset, 0 is written to this bit. Cache clearing (CC = 1) is allowed when CE=0. The following describe the combination of the CE and CF bits: <table border="1" style="margin-left: 40px;"> <thead> <tr> <th>CE</th> <th>CF</th> <th>Effect</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>Cache not enabled</td> </tr> <tr> <td>0</td> <td>1</td> <td>Cache not enabled</td> </tr> <tr> <td>1</td> <td>0</td> <td>Cache enabled and not frozen</td> </tr> <tr> <td>1</td> <td>1</td> <td>Cache enabled but frozen (cache read only)</td> </tr> </tbody> </table>	CE	CF	Effect	0	0	Cache not enabled	0	1	Cache not enabled	1	0	Cache enabled and not frozen	1	1	Cache enabled but frozen (cache read only)
CE	CF	Effect															
0	0	Cache not enabled															
0	1	Cache not enabled															
1	0	Cache enabled and not frozen															
1	1	Cache enabled but frozen (cache read only)															
12	CC	Cache clear. CC = 1 invalidates all entries in the cache (contents not guaranteed, "garbage"). This bit is always cleared after it is written to and thus always read as 0. At reset, 0 is written to this bit. All cache P flags = 0 when cache is cleared.															
13	GIE	Global interrupt enable. If GIE = 1, the CPU responds to an enabled interrupt. If GIE = 0, the CPU does not respond to an enabled interrupt (when a trap executes or an interrupt is taken, bit GIE is set to 0). This bit does not affect interrupts on the NMI pin. The IDLE, LAT, RETI, RETID, and TRAP instructions affect this bit's value.															
14	PGIE	Previous state of bit GIE. When a trap executes or an interrupt is taken, bit GIE is set to 0. When this occurs, the PGIE bit is set to the GIE bit's value before the trap or interrupt. Note that the RETI <i>cond</i> and RETI <i>cond</i> D instructions copy PGIE to the GIE bit. At reset, this bit is set to 0.															
15	SET COND	This bit determines how condition flags (ST bits 0 – 6) are set: If SET COND = 0, condition-flags are set if the operation's target is any extended-precision register (R0 – R11) compatible with the TMS320C30. This bit is set to 0 at reset. If SET COND = 1, condition flags are set if the target of the operation is <b>any</b> register in the primary register files <i>except</i> the status register. Condition flags are always set when a CMPF, CMPL, CMPF3, CMPI3, TSTB, or TSTB3 instruction is executed.															
16	ANALYSIS	In analysis mode — state information for emulation. Read only.															
17 – 31	Reserved	Value undefined. Read only. Reserved for an identification value. This value is set by Texas Instruments (e.g., to identify device types and revisions).															

### 3.1.8 DMA Coprocessor Interrupt Enable Register (DIE)

The 32-bit DMA interrupt enable register (DIE), shown in Figure 3–4, is broken into six subfields that determine which interrupts can be used to control the synchronization for each of the six DMA coprocessor channels. At reset, all zeroes are written to the register.

3

Figure 3–4. DMA Interrupt Enable Register Bit Functions

31	29	28	26	25	23	22	20	
DMA5 WRITE			DMA5 READ			DMA4 WRITE		DMA4 READ
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
19	17	16	14	13	11	10	8	
DMA3 WRITE			DMA3 READ			DMA2 WRITE		DMA2 READ
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
7	6	5	4	3	2	1	0	
DMA1 WRITE		DMA1 READ		DMA0 WRITE		DMA0 READ		
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

R = Read W = Write

Table 3–3 summarizes the interrupt activity for each of the four possible combinations of two-bit values in DMA0 and DMA1 (bottom of Figure 3–4). Likewise, Table 3–4 (page 3-9) summarizes the interrupts enabled by three-bit values in DMA2 through DMA5.

**Note: DMA Coprocessor Uses Signals to Synchronize**

The interrupts in Table 3–3 and Table 3–4 (ICRDY<sub>x</sub>, OCRDY<sub>x</sub>, TIM0, etc.) are *not vectored*. The DMA uses these as signals to synchronize DMA coprocessor transfers. This is explained in Section 9.9 on page 9-40.

Table 3–3. DMA Channels 0 and 1 Synchronization Interrupts (DMA0 and DMA1)

Bit Value (in DMA0 or DMA1)	Interrupt Enabled at DMA0 or DMA1				Interrupt Source for DMA Synchronization
	DMA0 Read	DMA0 Write	DMA1 Read	DMA1 Write	
0 0	None	None	None	None	--
0 1	ICRDY0	OCRDY0	ICRDY1	OCRDY1	From communication port
1 0	$\overline{\text{IIOF0}}$	$\overline{\text{IIOF1}}$	$\overline{\text{IIOF2}}$	$\overline{\text{IIOF3}}$	From external pins $\overline{\text{IIOF0}}$ – $\overline{\text{IIOF3}}$
1 1	TIM0	TIM0	TIM0	TIM0	From timer TIM0

This interrupt synchronization scheme allows each DMA channel to service a corresponding input communication port and output communication port. Also, each DMA channel can be synchronized with external interrupts and the on-chip timers.

Table 3–4. DMA Channels 2 to 5 Synchronization Interrupts (DMA2 to DMA)5

Bit Value (in DMA2 to DMA5)	Interrupt Enabled at DMA2–DMA5†		Interrupt Source for DMA Synchronization
	†DMA <sub>x</sub> Read	†DMA <sub>x</sub> Write	
0 0 0	None	None	--
0 0 1	†ICRDY <sub>x</sub>	†OCRDY <sub>x</sub>	From communication port
0 1 0	$\overline{\text{IIOF0}}$	$\overline{\text{IIOF0}}$	} From external pins INT0 – INT3
0 1 1	$\overline{\text{IIOF1}}$	$\overline{\text{IIOF1}}$	
1 0 0	$\overline{\text{IIOF2}}$	$\overline{\text{IIOF2}}$	
1 0 1	$\overline{\text{IIOF3}}$	$\overline{\text{IIOF3}}$	
1 1 0	TIM0	TIM0	} From timers TIM0 and TIM1
1 1 1	TIM1	TIM1	

† The *x* in DMA<sub>x</sub> is the DMA channel number, which is also the number for the corresponding ICRDY<sub>x</sub> and OCRDY<sub>x</sub> interrupts. For example, an 001<sub>2</sub> in both DMA2 READ and DMA5 WRITE would enable interrupts ICRDY2 and OCRDY5, respectively. All other viable bit values (010<sub>2</sub> to 111<sub>2</sub>) are the same (as shown in the table) for DMA2 through DMA5.

Note that each DMA channel looks not only at the DMA synchronous interrupts selected but also at the synchronization mode that the channel is currently using (see Table 9–4 on page 9-15). The synchronization mode is specified by the DMA channel control registers located in the DMA coprocessor.

### **3.1.9 CPU Internal Interrupt Enable Register (IIE)**

The 32-bit internal interrupt enable register, shown in Figure 3–5, enables/disables the following interrupts for the CPU:

- Timers 0 and 1,
- For communication ports 0–5:
  - Input-buffer full,
  - Input-buffer ready,
  - Output-buffer ready,
  - Output-buffer empty .
- DMA coprocessor channels 0–5.

Figure 3–5 shows the IIE register bits, and Table 3–5 describes the interrupt enabled, depending on the bit value. A 1 read means the corresponding interrupt is enabled; a 0 indicates disabled. At reset, all zeroes are written to the register.

Figure 3-5. Internal Interrupt Enable Register (IIE)

31	30	29	28	27	26	25			
ETINT1	EDMA-INT5	EDMA-INT4	EDMA-INT3	EDMA-INT2	EDMA-INT1	EDMA-INT0			
R/W	R/W	R/W	R/W	R/W	R/W	R/W			
24	23	22	21	20	19	18	17		
EOC-EMPTY5	EOC-RDY5	EIC-RDY5	EIC-FULL5	EOC-EMPTY4	EOC-RDY4	EIC-RDY4	EIC-FULL4		
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W		
16	15	14	13	12	11	10	9		
EOC-EMPTY3	EOC-RDY3	EIC-RDY3	EIC-FULL3	EOC-EMPTY2	EOC-RDY2	EIC-RDY2	EIC-FULL2		
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W		
8	7	6	5	4	3	2	1	0	
EOC-EMPTY1	EOC-RDY1	EIC-RDY1	EIC-FULL1	EOC-EMPTY0	EOC-RDY0	EIC-RDY0	EIC-FULL0	ETINT0	
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	

R = Read, W = Write, R/W = Read/Write

Table 3-5. Summary of Interrupt Enable Register Bits (IIE)

IIE Bit Field Name	IIE Bit Numbers						Port No.	Enables/Disables (note 1)
	0	1	2	3	4	5		
EICFULL <sub>x</sub> (Note 2)	1	5	9	13	17	21		Comm. port x input-buffer full interrupt
EICRDY <sub>x</sub> (Note 2)	2	6	10	14	18	22		Comm. port x input-buffer ready interrupt
EOCRDY <sub>x</sub> (Note 2)	3	7	14	15	19	23		Comm. port x output-buffer ready interrupt
EOEMPTY <sub>x</sub> (Note 2)	4	8	12	16	20	24		Comm. port x output-buffer empty interrupt
EDMAINT <sub>x</sub> (Note 2)	25	26	27	28	29	30		DMA coprocessor channel x interrupt
ETINT0	0							Timer 0 interrupt
ETINT1	31							Timer 1 interrupt

- NOTES: 1 The x represents a corresponding communication port number (0 – 5) or DMA coprocessor channel number (0 – 5). For example, ones in bits 5 and 25 enable interrupts for (a) input-buffer full at communication port 1 and for (b) DMA coprocessor channel 0. (A 1 enables the interrupt; a 0 disables it.)
2. Communication port bits are shaded according to communication port number. For example, communication port 0's bit numbers are in the first group of vertical shading. Thus, communication port 0's bits are 1, 2, 3, 4; communication-port 1's bits are 5, 6, 7, 8; etc. The DMA coprocessor channel interrupts are shown the same way (e.g., EDMAINT0 at bit 25, EDMAINT1 at bit 26, etc.).

### 3.1.10 IIOF Flag Register (IIF) Controls External Pins IIOF(3 – 0), Timer/DMA Flags

The IIF register controls the external interrupt pins IIOF(3 – 0). Use it to specify:

- which IIOF pins are used for general-purpose I/O and which are used for interrupts,
- whether a general-purpose pin is input (read only) or output (read/write),
- whether an interrupt pin is for edge-triggered or level-triggered interrupts,
- if an interrupt is enabled or disabled.

Figure 3–6 depicts the IIF register bits. Table 3–6 (page 3-13) explains these bits in detail. Interrupt traps are shown in Figure 3–7 (page 3-15). Interrupts are further explained in Section 6.7 on page 6-23.

Figure 3–6. Interrupt Flag Register (IIF)

31	30	29	28	27	26	25	24
TINT1	DMAINT5	DMAINT4	DMAINT3	DMAINT2	DMAINT1	DMAINT0	TINT0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	NMI
R	R	R	R	R	R	R	R
15	14	13	12	11	10	9	8
EIIOF3	FLAG3	TYPE3	FUNC3	EIIOF2	FLAG2	TYPE2	FUNC2
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
7	6	5	4	3	2	1	0
EIIOF1	FLAG1	TYPE1	FUNC1	EIIOF0	FLAG0	TYPE0	FUNC0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

R = Read (only), R/W = Read/Write, xx = Reserved, read as 0

Table 3–6. IIF Register Bits Summary

Bit Field Name	IIF Bit Nos.				Port No.	Function (Note 1)	
	0	1	2	3			
FUNC <sub>x</sub> (note 2)	0	4	8	12		Mode of pin IIOF <sub>x</sub> : If FUNC <sub>x</sub> = 0, pin IIOF <sub>x</sub> is a <b>general-purpose I/O (R/W) pin</b> . If FUNC <sub>x</sub> = 1, pin IIOF <sub>x</sub> is an <b>interrupt (R) pin</b> .	
TYPE <sub>x</sub> (note 2)	1	5	9	13		Type of function for pin IIOF <sub>x</sub> : If pin IIOF <sub>x</sub> is a general-purpose I/O pin (FUNC <sub>x</sub> = 0): TYPE <sub>x</sub> = 0 makes IIOF <sub>x</sub> an <b>input pin</b> . TYPE <sub>x</sub> = 1 makes IIOF <sub>x</sub> an <b>output pin</b> . If pin IIOF <sub>x</sub> is an interrupt pin (FUNC <sub>x</sub> = 1): TYPE <sub>x</sub> = 0 makes IIOF <sub>x</sub> an <b>edge-triggered latched interrupt</b> , TYPE <sub>x</sub> = 1 makes IIOF <sub>x</sub> a <b>level-triggered unlatched interrupt</b> .	
FLAG <sub>x</sub> (note 2)	2	6	10	14		Flag for pin IIOF <sub>x</sub> : If pin IIOF <sub>x</sub> is a general-purpose <b>input pin</b> (FUNC <sub>x</sub> = 0, TYPE <sub>x</sub> = 0), FLAG <sub>x</sub> = the value of pin IIOF <sub>x</sub> and is <b>read only</b> . If pin IIOF <sub>x</sub> is a general-purpose <b>output pin</b> (FUNC <sub>x</sub> = 0, TYPE <sub>x</sub> = 1), FLAG <sub>x</sub> = the value on pin IIOF <sub>x</sub> and is <b>R/W</b> . If pin IIOF <sub>x</sub> is an interrupt pin (FUNC <sub>x</sub> = 1): FLAG <sub>x</sub> = 0 if interrupt is <b>not asserted</b> . FLAG <sub>x</sub> = 1 if interrupt is <b>asserted</b> . If 0 (zero) is written to FLAG <sub>x</sub> , the corresponding interrupt is cleared unless an interrupt is on the same pin; in that case, the interrupt will be set.	
EIOF <sub>x</sub> (note 2)	3	7	11	15		Disable/enable external interrupt: EIOF <sub>x</sub> = 0 disables external interrupts at pin IIOF <sub>x</sub> . EIOF <sub>x</sub> = 1 enables external interrupts at pin IIOF <sub>x</sub> .	
NMI	16					Nonmaskable Interrupt flag (NMI). The NMI interrupt (on the external $\overline{\text{NMI}}$ pin) behaves like other interrupts, except it cannot be masked (disabled) by the GIE bit (ST bit 13) or by writing to the NMI bit itself. It is temporarily masked during delayed branches and multicycle CPU operations. At reset, this bit is cleared. An asserted interrupt is cleared only by servicing the interrupt. NMI is a negative-going, edge-triggered, latched interrupt. It is read only. Reading NMI as 0 indicates the interrupt is not asserted. Reading NMI as 1 indicates the interrupt is asserted.	
Reserved	17 – 23					Reserved; read as zeroes.	
TINT0 TINT1	24 31						Timer interrupt flags 0 and 1: Reading TINT <sub>x</sub> as 0 indicates the timer interrupt is not asserted. Reading TINT <sub>x</sub> as 1 indicates the timer interrupt is asserted. A zero written to this bit clears the interrupt unless the interrupt is asserted at the same time; in that case, the interrupt will be shown as asserted.
DMAINT <sub>x</sub>	25 – 30					Interrupt flag for DMA coprocessor channels 0 to 5. Reading DMAINT <sub>x</sub> as 0 indicates the channel interrupt is not asserted. Reading DMAINT <sub>x</sub> as 1 indicates the channel interrupt is asserted. A zero written to this bit clears the interrupt unless the interrupt is asserted at the same time; in that case, the interrupt will be shown as asserted.	

NOTES: 1 The x represents the corresponding IIOF interrupt pin (IIOF3–IIOF0). R = Read, /W = Read/Write  
 2. Shading organizes each communication port's bits the same as shown for the IIE register in Table 3–5 (see note 2) on page 3-11. For example, bits 0, 1, 2, 3 apply to pin IIOF0; bits 4, 5, 6, 7 apply to IIOF1, etc.

3

### 3.1.11 Block-Repeat (RS, RE) and Repeat-Count (RC) Registers

The 32-bit repeat start address register (RS) contains the starting address of the block of program memory to be repeated when operating in the repeat mode.

The 32-bit repeat end address register (RE) contains the ending address of the block of program memory to be repeated when operating in the repeat mode.

The repeat-count register (RC) is a 32-bit register used to specify the number of times a block of code is to be repeated when performing a block repeat. If RC contains the number  $n$ , the loop will be executed  $n + 1$  times.

### 3.1.12 Program Counter (PC)

The program counter (PC) is a 32-bit register containing the address of the next instruction to be fetched. While the program counter is not part of the CPU register file, it is a register that can be modified by instructions that modify the program flow.

### 3.1.13 Reserved Bits and Compatibility

In order to retain compatibility with future members of the TMS320C4x family of microprocessors, reserved bits that are read as zero must be written as zero. Reserved bits that have an undefined value **must not** have their current value modified. In other cases, maintain the reserved bits as specified.

### 3.2 CPU Expansion Register File

This expansion register file contains two special control registers:

- ❑ Interrupt-vector table pointer register (IVTP),
- ❑ Trap-vector table pointer (TVTP).

Table 3–7. CPU Expansion Registers

Assembler Syntax	Function Name
IVTP	Interrupt-vector table pointer. Points to start of interrupt-vector table (shown in Figure 3–8).
TVTP	Trap-vector table pointer. Points to start of the 512-trap-vector table (shown at page bottom).

Use the **LDEP instruction** to load (copy) an expansion register to a primary register (e.g., to any of the auxiliary registers AR0 – AR7, see Table 3–1 on page 3-3). For example:

```
LDEP    IVTP, AR5           ; IVTP contents to AR5
```

Likewise, use the **LDPE instruction** to load (copy) a primary register to an expansion register. Neither of these instructions affects the status register condition flags.

```
LDPE    AR5, IVTP         ; AR5 contents to IVTP
```

Note that both the interrupt-vector table and the trap-vector table **are required to lie on a 512-word boundary**; thus, the **nine least-significant bits of these pointers are zeroes** (i.e.,  $10\ 0000\ 0000_2 = 512 = 200h$ ). Write only zeroes to these bits (though the register forces these to zeroes).

The 32-bit **IVTP register** points to (is essentially the base address for) the interrupt-vector table (IVT) in memory. The contents of this table are depicted in Figure 3–8 on page 3-16.

The 32-bit **TVTP register** is essentially the base address for the trap-vector table (TVT) in memory. This table, depicted below, contains the vectors for the TRAP instruction’s 512-trap addresses (TRAP0–TRAP511).

The interrupt (including RESET — see Section 3.3) and trap maps can be configured to overlap. At reset, IVTP and TVTP are set to all zeroes.

Figure 3–7. Trap Vector Table (TVT)

TVTP + 000h	TRAP0
TVTP + 001h	TRAP1
•	TO
•	TRAP509
TVTP + 1FEh	TRAP510
TVTP + 1FFh	TRAP511

Figure 3–8. Interrupt-Vector Table (IVT)

IVTP + 000h	Reserved	Note 1	IVTP + 01Dh	ICFULL4	Note 5
IVTP + 001h	NMI	Note 2	IVTP + 01Eh	ICRDY4	
IVTP + 002h	TINT0	Note 3	IVTP + 01Fh	OCRDY4	
IVTP + 003h	IIOF0	Note 4	IVTP + 020h	OEMPTY4	
IVTP + 004h	IIOF1		IVTP + 021h	ICFULL5	
IVTP + 005h	IIOF2		IVTP + 022h	ICRDY5	
IVTP + 006h	IIOF3		IVTP + 023h	OCRDY5	
IVTP + 007h	Unused	Note 5	IVTP + 024h	OEMPTY5	
IVTP + 008h			Note 6	IVTP + 025h	DMA INT0
IVTP + 009h				IVTP + 026h	DMA INT1
IVTP + 00Ah				IVTP + 027h	DMA INT2
IVTP + 00Bh	IVTP + 028h			DMA INT3	
IVTP + 00Ch	ICFULL0	Note 3	IVTP + 029h	DMA INT4	
IVTP + 00Dh	ICRDY0		IVTP + 02Ah	DMA INT5	
IVTP + 00Eh	OCRDY0		IVTP + 02Bh	TINT1	
IVTP + 00Fh	OEMPTY0		IVTP + 02Ch	Unused	
IVTP + 010h	ICFULL1		IVTP +		Note 5
IVTP + 011h	ICRDY1	IVTP + •			
IVTP + 012h	OCRDY1	IVTP + •			
IVTP + 013h	OEMPTY1	IVTP + •			
IVTP + 014h	ICFULL2	Note 3	IVTP + 03Eh	Reserved	
IVTP + 015h	ICRDY2		IVTP + 03Fh		
IVTP + 016h	OCRDY2				
IVTP + 017h	OEMPTY2				
IVTP + 018h	ICFULL3				
IVTP + 019h	ICRDY3				
IVTP + 01Ah	OCRDY3				
IVTP + 01Bh	OEMPTY3				
IVTP + 01Ch					

- Notes:**
- 1) Reserved for the **reset** vector when IVTP = 0000 0000h and RESETLOC(1,0) = 0 0<sub>2</sub> or when IVTP=08000 0000h and RESETLOC(1,0) = 1 0<sub>2</sub>. See Table 3–8.
  - 2) NMI (nonmaskable interrupt) is discussed in Section 9.9, page 9-40.
  - 3) **Timer** interrupts TINT0 and TINT1 are enabled and programmed by the IIE register (subsection 3.1.9, page 3-10) and monitored at the IIF register (subsection 3.1.10, page 3-12).
  - 4) External **pins** IIOF0–IIOF5 are programmed in the DIE register (subsection 3.1.8, page 3-8) and IIF register.
  - 5) The **communication port** I/O buffers full/ready interrupts are enabled by the DIE and IIE registers and also discussed in Table 8–1, page 8-10 (OUTPUT LEVEL & INPUT LEVEL bits).
  - 6) **DMA** interrupts are enabled at the IIE register and DMA channel control register (at bits TCC and AUX TCC explained in Table 9–1 on page 9-8).

### 3.3 RESET Vector Mapping

The 'C40s RESET vector can reside in any one of four memory locations. The value on two external pins (RESETLOC(1,0)) determines the RESET vector location as shown in the following table.

Table 3–8. Four RESET Vector Locations Chosen by Values on Pins RESETLOC(1,0)

Value at RESETLOCx Pin		Get RESET Vector From Memory Address	Comment
RESETLOC1	RESETLOC0		
0	0	00000 0000 <sub>16</sub>	Local Bus
0	1	07FFF FFFF <sub>16</sub>	Local Bus
1	0	08000 0000 <sub>16</sub>	Global Bus
1	1	0FFFF FFFF <sub>16</sub>	Global Bus

Note that if pin ROMEN = 1 and the vector at 0000 0000h is enabled (pins RESETLOC(1,0) = 00), then the vector is mapped to address 0 of internal ROM.

This mapping scheme of the RESET vector allows the TMS320C40 to be integrated easily into systems having other processors with fixed RESET vector locations. It also allows you to make the RESET vector either external or internal (on-chip ROM) to the processor.

### 3.4 Memory

The TMS320C40's memory space of 4 giga words (4 billion  $\times$  32 bits where  $1\text{ G} = 2^{30}$ ) is shown in the two memory maps in Figure 3–9. These maps differ only by the makeup of the lowest address space at 0000 0000h to 0000 0FFFh. This makeup is configured by the value at pin ROMEN (onchip — reserved — ROM enable, pin AK4):

- ❑ **ROMEN = 1.** Addresses 0000h – 0FFFh are an accessible onchip ROM block (reserved), and 0000 1000h – 000F FFFFh are reserved
- ❑ **ROMEN = 0.** The on-chip (reserved) ROM is disabled, and addresses 0000 0000h – 000F FFFFh are accessible over the local bus.

Memory in both maps starting at 10 0000h is not affected by ROMEN (as described for addresses 00000h – FFFFFh above). A general summary of address ranges:

- ❑ **0000 0000h – 000F FFFFh:** Can be local bus or on-chip (reserved) ROM, depending on the value of pin ROMEN.
- ❑ **0010 0000h – 0010 00FFh:** Internal peripherals (DMA coprocessor, communications ports, timers, etc.)
- ❑ **0010 0100h – 001F FFFFh:** Internal peripheral region.
- ❑ **0020 0000h – 002F F7FFh:** Reserved.
- ❑ **002F F800h – 002F FBFFh:** 1K RAM Block 0.
- ❑ **002F FC00h – 002F FFFFh:** 1K RAM Block 1.
- ❑ **0030 0000h – 07FFF FFFFh:** Local bus. If ROMEN = 1, another part of the local bus is at 00 0000h – 0F FFFFh. These addresses activate the local bus.
- ❑ **08000 0000h – 0FFFF FFFFh:** Global bus.

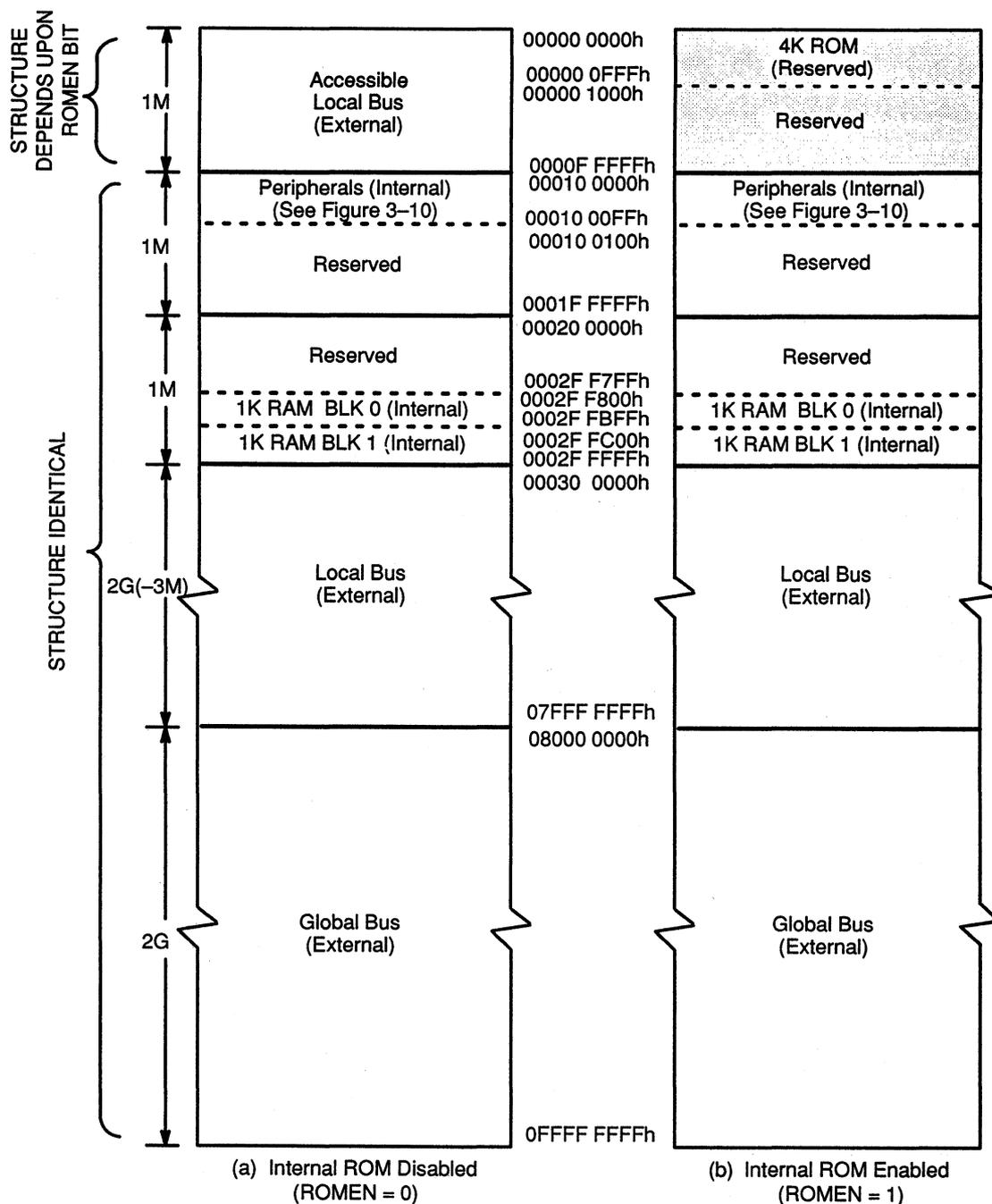
**Instructions cannot be accessed in these 3 areas.**

CPU data accesses and DMA accesses can be made from any unreserved part of the 'C40 memory map. Instruction fetches can take place at any unreserved area of the 'C40 memory map except at the peripheral space (addresses 0010 0000h – 0010 00FFh).

**The 'C40's internal ROM is currently reserved for TI internal use only.**

### 3.4.1 Overall Memory Map

Figure 3-9. Memory Maps



3

### 3.4.2 Peripheral Bus Memory Map

This map resides in addresses 0010 0000h – 0010 00FFh as shown in the memory map, Figure 3–9. Each peripheral requires a 16-word area.

Figure 3–10. Peripheral Memory Map

0010 0000h	Local and Global Port Control (16 words) (See subsection 3.4.2.1 and Figure 3–11)
0010 000Fh	
0010 0010h	Analysis Module Block Registers (16 words) (See subsection 3.4.2.2 and Figure 3–12)
0010 001Fh	
0010 0020h	Timer 0 Registers (16 words) (See subsection 3.4.2.3 and Figure 3–13)
0010 002Fh	
0010 0030h	Timer 1 Registers (16 words) (See subsection 3.4.2.3 and Figure 3–13)
0010 003Fh	
0010 0040h	Communication Port 0 (16 words) (See subsection 3.4.2.4 and Figure 3–14)
0010 004Fh	
0010 0050h	Communication Port 1 (16 words) (See subsection 3.4.2.4 and Figure 3–14)
0010 005Fh	
0010 0060h	Communication Port 2 (16 words) (See subsection 3.4.2.4 and Figure 3–14)
0010 006Fh	
0010 0070h	Communication Port 3 (16 words) (See subsection 3.4.2.4 and Figure 3–14)
0010 007Fh	
0010 0080h	Communication Port 4 (16 words) (See subsection 3.4.2.4 and Figure 3–14)
0010 008Fh	
0010 0090h	Communication Port 5 (16 words) (See subsection 3.4.2.4 and Figure 3–14)
0010 009Fh	
0010 00A0h	DMA Coprocessor Channel 0 (16 words) (See subsection 3.4.2.5 and Figure 3–15, page 3–24.)
0010 00AFh	
0010 00B0h	DMA Coprocessor Channel 1 (16 words) (See subsection 3.4.2.5 and Figure 3–15, page 3–24.)
0010 00BFh	
0010 00C0h	DMA Coprocessor Channel 2 (16 words) (See subsection 3.4.2.5 and Figure 3–15, page 3–24.)
0010 00CFh	
0010 00D0h	DMA Coprocessor Channel 3 (16 words) (See subsection 3.4.2.5 and Figure 3–15, page 3–24.)
0010 00DFh	
0010 00E0h	DMA Coprocessor Channel 4 (16 words) (See subsection 3.4.2.5 and Figure 3–15, page 3–24.)
0010 00EFh	
0010 00F0h	DMA Coprocessor Channel 5 (16 words) (See subsection 3.4.2.5 Figure 3–15, page 3–24.)
0010 00FFh	

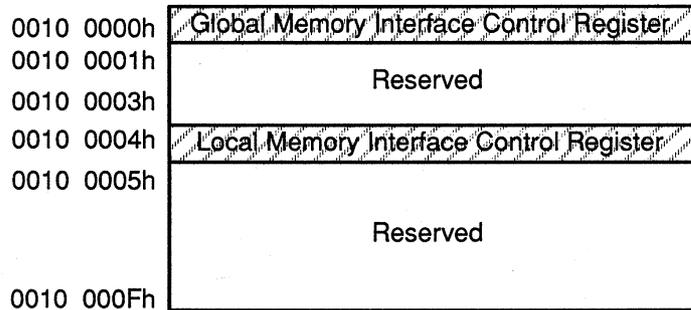
### 3.4.2.1 Local and Global Memory Interface Control Registers

These registers control the local and global memory interfaces. They occupy the first 16-word block of the peripheral bus memory map, shown in Figure 3–10. The registers themselves are shown in Figure 3–11. Chapter 7 covers the operation of these registers. A detailed description of these is shown in Figure 7–2 and Table 7–3 (pages 7-7 and 7-8).

These registers define:

- the page sizes used for the two strobes of each port,
- address ranges over which the strobes are active,
- wait states, and
- other similar operations that compose the memory interfaces.

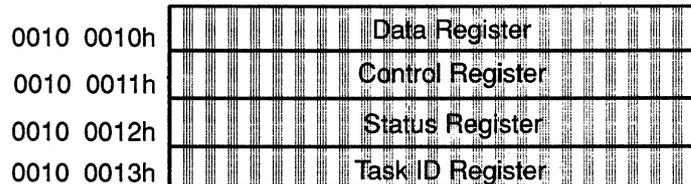
Figure 3–11. Memory Interface Control Registers



### 3.4.2.2 Analysis Module Registers

These registers, the second 16-word block in the peripheral bus memory Map (Figure 3–10), are shown below in Figure 3–12. These registers are reserved for emulation functions.

Figure 3–12. Analysis Module Registers

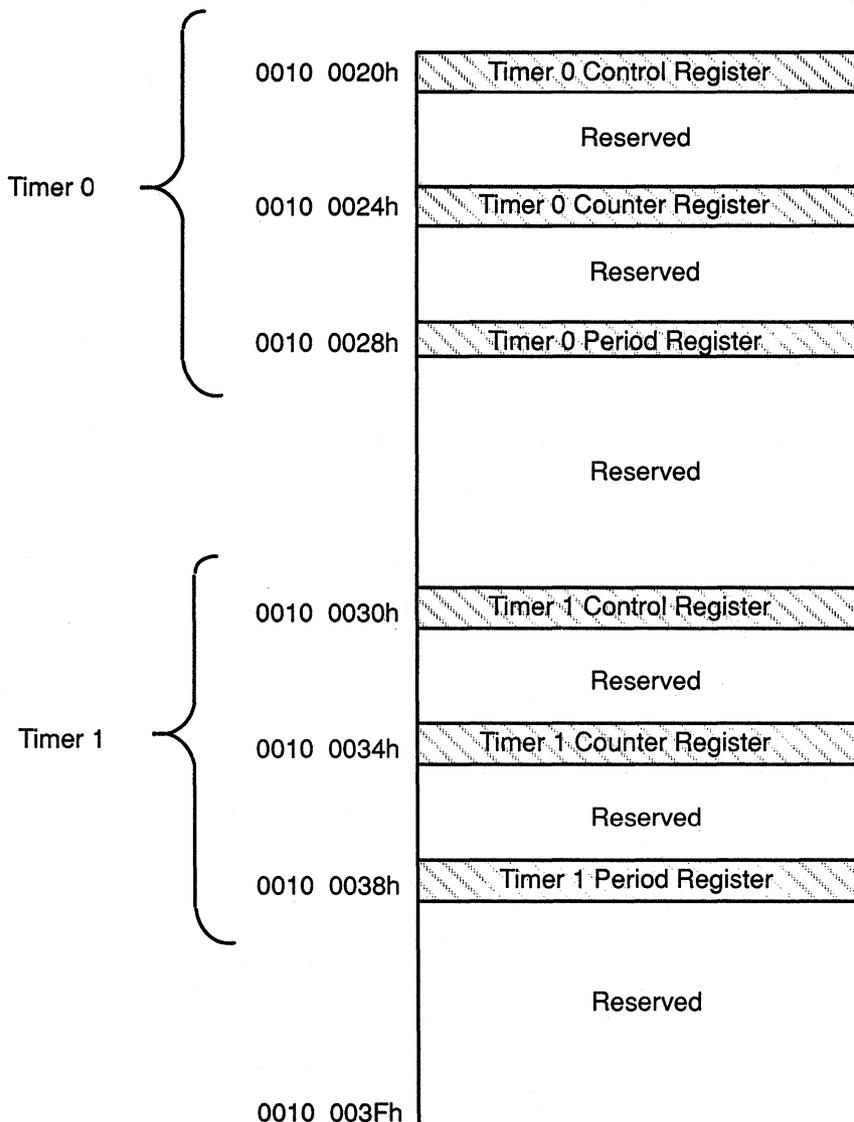


### 3.4.2.3 Timer Registers

This group of registers occupies the 0010 0020h – 0010 003Fh range in the peripheral bus memory map, Figure 3–10, on page 3-20. Timers and their registers are covered in detail in Section 9.10 on page 9-45.

Figure 3–13. Timer Registers

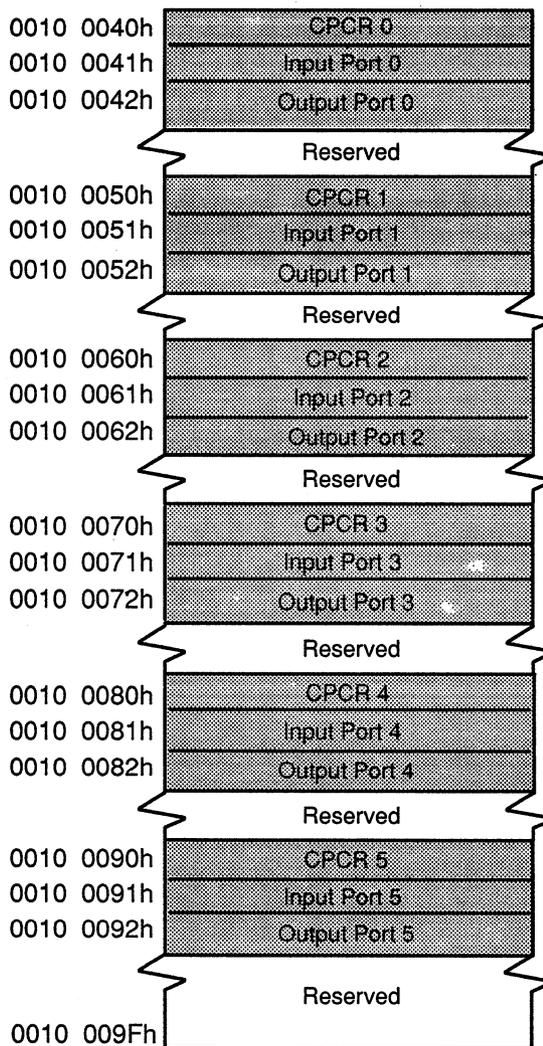
3



### 3.4.2.4 Communication Port Memory Map

The communication-port control registers (CPCR) and input and output FIFO buffers are illustrated below in Figure 3–14. This is the central group of registers in the peripheral bus memory map, Figure 3–10, on page 3-20. These are described in more detail in Chapter 8.

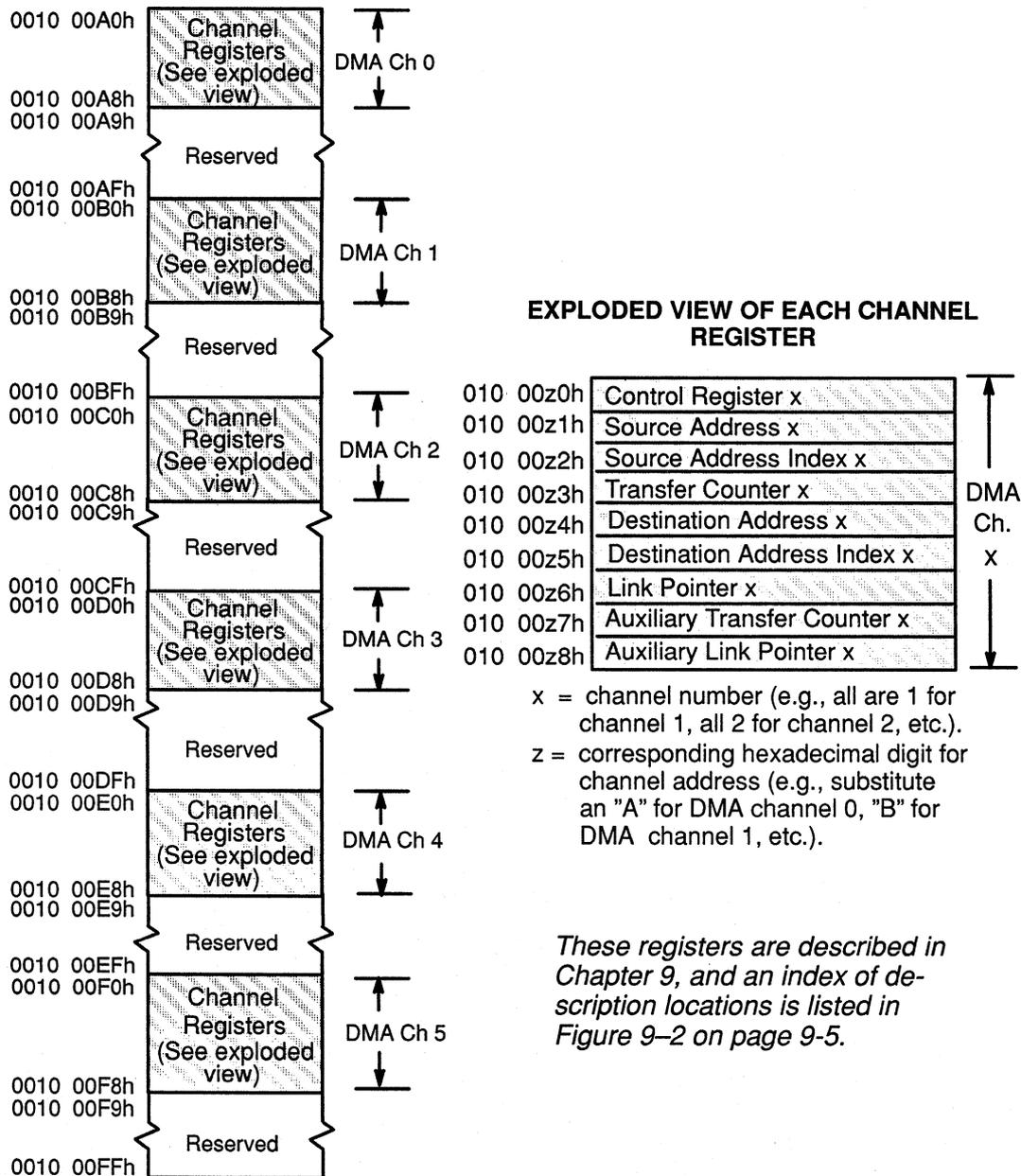
Figure 3–14. Communication Port Memory Map



### 3.4.2.5 DMA Coprocessor Registers

The DMA registers (shown below) are the bottom block of registers in the peripheral bus memory map (Figure 3–10 on page 3-20). These registers are described in Chapter 9. Figure 9–2, page 9-5, is an index to subjects.

Figure 3–15. DMA Coprocessor Memory Map



3

### 3.5 Instruction Cache Architecture

The  $128 \times 32$ -bit instruction cache speeds instruction fetches and lowers system cost. The instruction cache allows the use of slow external memories while still achieving single-cycle access performance. The cache also frees the external bus from program fetches, thus, allowing the use of these buses for DMA or other system needs. The cache can operate in a completely automatic fashion without the need for external intervention. It uses a form of the LRU (least recently used) cache update algorithm.

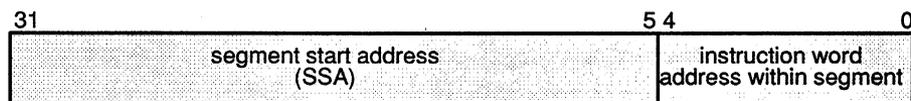
3

The instruction cache (see Figure 3–17 on page 3-26) contains 128 32-bit words of RAM, enough to hold 128 words of program memory. It is divided into four 32-word segments. Associated with each segment is a 27-bit segment start address (SSA) register. For each word in the cache, there is a corresponding single-bit present (P) flag.

When the CPU requests an instruction word, a check is made to determine whether the word is already in the instruction cache. The partitioning of an instruction address as used by the cache control algorithm is shown in Figure 3–16. The 27 most significant bits (MSBs) of the instruction address select the segment, and the 5 least significant bits define the address of the instruction word within the pertinent segment. The 27 MSBs of the instruction address are compared with the four SSA registers. If a match is found, the relevant P flag is checked. The **P flag** indicates whether or not the word within a particular segment is already present in cache memory:

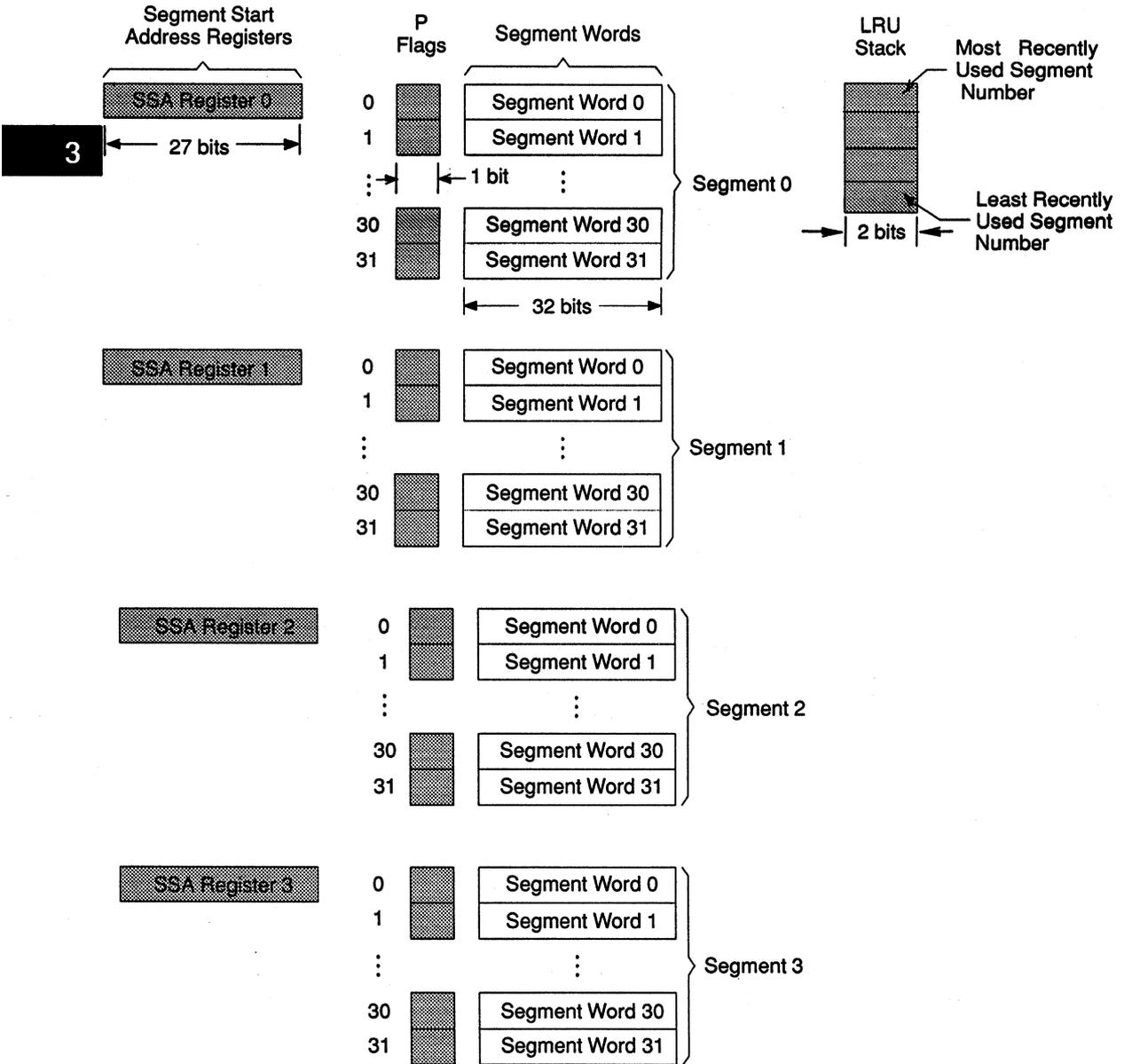
- ☐ P = 1: the word is already present in cache memory.
- ☐ P = 0: location in cache is invalid (e.g., contains garbage).

Figure 3–16. Address Partitioning for Cache Control Algorithm



If there is no match, one of the segments must be replaced by the new data. The segment replaced in this circumstance is determined by the LRU (least recently used) algorithm. The LRU stack (see upper right of Figure 3–17 on page 3-26) is maintained for this purpose.

Figure 3-17. Instruction Cache Architecture



The LRU stack keeps track of which segment (0 – 3) qualifies as the least recently used after each access to the cache. Each time a segment is accessed, its segment number is removed from the LRU stack and pushed onto the top of the LRU stack. Therefore, the number at the top of the stack is the most recently used segment number, and the number at the bottom of the stack is the least recently used segment number.

At RESET, the following occur in the instruction cache:

- ❑ all P flags are set to zero, and
- ❑ the LRU stack is initialized with segment no. 0 at the top followed by 1, 2, and 3 at the bottom. If any two SSA registers are equal (due to RESET conditions) and a cache hit occurs, the instruction word is fetched from the most recently used segment.

When a replacement is necessary, the least recently used segment is selected for replacement. Also, the 32 P flags for the segment to be replaced are set to 0, and the segment's SSA register is replaced with the 27 MSBs of the instruction address.

### 3.5.1 Cache Algorithm

When the TMS320C40 requests an instruction word from external memory, the two possible actions are a *cache hit* or a *cache miss*.

- ❑ **Cache Hit.** The cache contains the requested instruction, and the following actions occur:
  - The instruction word is read from the cache.
  - The number of the segment containing the word is removed from the LRU stack and pushed to the top of the LRU stack (if not already at the top), thus moving the other segment numbers toward the bottom of the stack.
- ❑ **Cache Miss.** The cache does not contain the instruction. Types of cache misses are
  - **Subsegment miss.** The segment address register matches the instruction address, but the relevant P flag is not set. The following actions occur:
    - The instruction word is read from memory and copied into the cache.
    - The number of the segment containing the word is removed from the LRU stack and pushed to the top of the LRU stack (if not already at the top), thus moving the other segment numbers toward the bottom of the stack.
    - The relevant P flag is set.

- **Segment miss.** None of the segment addresses matches the instruction address. The following actions occur:
  - The least recently used segment is selected for replacement. The P flags for all 32 words are cleared.
  - The SSA register for the selected segment is loaded with the 27 MSBs of the address of the requested instruction word.
  - The instruction word is fetched and copied into the cache. It goes into the appropriate word of the least recently used segment. The P flag for that word is set to 1.
  - The number of the segment containing the instruction word is removed from the LRU stack and pushed to the top of the LRU stack, thus moving the other segment numbers toward the bottom of the stack.

### 3.5.2 Cache and System Memory

Only instructions may be fetched from the program cache. All reads and writes of data in memory bypass the cache. Program fetches from internal memory do not modify the cache and do not generate cache hits or misses. The program cache is a single-access memory block. Dummy program fetches (i.e., following a branch) can generate cache misses and cache updates.

**Avoid using self-modifying code.** If an instruction resides in cache and the corresponding location in primary memory is modified, the copy of the instruction in cache is not modified.

**Cache can be used more efficiently** by aligning program code on 32-word address boundaries. Do this by using the ALIGN directive when coding assembly language.

### 3.5.3 Cache Control Bits

Four cache control bits are located in the CPU status register: the cache clear bit (**CC**), the cache enable bit (**CE**), the cache freeze bit (**CF**), and the previous cache freeze bit (**PCF**) as shown in Figure 3–3 on page 3-6. The definitions of these bits are repeated below from Table 3–2.

**Cache Clear Bit (CC).** Set  $CC = 1$  to invalidate all entries in the cache (contents not guaranteed, "garbage"). This bit is always cleared after it is written to; thus, it is always read as 0. At reset, 0 is written to this bit. The cache P flag = 0 when cache is cleared.

**Cache Enable Bit (CE).** Set  $CE = 1$  to enable the cache, allowing the cache to be used according to the LRU (least recently used) cache algorithm. Set  $CE = 0$  to disable the cache; this prevents cache updates or modifications (thus no cache fetches can be made). At reset, 0 is written to this bit. Cache clearing ( $CC = 1$ ) is allowed when  $CE=0$ .

**Cache Freeze Bit (CF).** Set  $CF = 1$  to freeze the cache (cannot be written to) including freezing of LRU (least recently used) stack manipulation. If the cache is enabled ( $CE = 1$ ), fetches from the cache are allowed, but modification of the cache contents is not allowed. Cache clearing ( $CC=1$ ) is allowed. At reset, this bit is set to zero. When  $CF=0$ , cache clearing ( $CC=1$ ) is allowed.  $CF$  is set to one when a trap or interrupt is taken. Also, the RETI and RETID instructions copy PCF to the CF bit.

Table 3–9 defines the effect of the CE and CF bits used in combination.

Table 3–9. Combined Effect of the CE and CF Bits

CE	CF	Effect
0	0	Cache not enabled
0	1	Cache not enabled
1	0	Cache enabled and not frozen
1	1	Cache enabled and frozen

**Previous Cache Freeze Bit (PCF).** When an interrupt or trap vector is taken, the CF value is copied to the PCF bit and the CF bit is set to 1. This protects the cache during interrupt processing and is particularly useful when code loops are interrupted. The interrupt service routine may optionally use the cache under software control. Interrupts may also be nested, providing that the status register is saved prior to enabling the interrupts. When the instructions RETI*cond* and RETI*condD* are executed to complete interrupt processing, the contents of the PCF bit are copied to the CF bit.



## Data Formats and Floating-Point Operation

---

---

---

In the TMS320C40 architecture, data is organized into three fundamental types: integer, unsigned-integer, and floating-point. Note that the terms, integer and signed-integer, are considered to be equivalent. The TMS320C40 supports short and single-precision formats for signed and unsigned integers. It also supports short, single-precision and extended-precision formats for floating-point data.

Floating-point operations make fast, trouble-free, accurate, and precise computations. Specifically, the TMS320C40 implementation of floating-point arithmetic facilitates floating-point operations at integer speeds while preventing problems with overflow, operand alignment, and other burdensome tasks common in integer operations.

This chapter discusses in detail the data formats and floating-point operations supported on the TMS320C40. Major topics in this section are as follows:

Section	Page
4.1 Signed Integer Formats .....	4-3
■ Short Integer Format .....	4-3
■ Single-Precision Integer Format .....	4-3
4.2 Unsigned-Integer Formats .....	4-4
■ Short Unsigned-Integer Format ) .....	4-4
■ Single-Precision Unsigned-Integer Format .....	4-4
4.3 Floating-Point Formats .....	4-5
■ Short Floating-Point Format .....	4-6
■ Single-Precision Floating-Point Format .....	4-7

<b>Section</b>	<b>Page</b>
■ Extended-Precision Floating-Point Format .....	4-8
■ Conversion Between Floating-Point Formats .....	4-9
4.4 Floating-Point Conversions, IEEE/C4x .....	4-11
■ Converting IEEE Format to Twos Complement Floating-Point Format .....	4-12
■ Converting Twos Complement Floating-Point Format to IEEE Format .....	4-13
4.5 Floating-Point Multiplication .....	4-15
4.6 Floating-Point Addition and Subtraction .....	4-20
4.7 Normalization, (NORM Instruction) .....	4-24
4.8 Rounding, (RND Instruction) .....	4-26
4.9 Floating-Point to Integer Conversions, FIX Instruction .....	4-28
4.10 Integer to Floating-Point Conversion, FLOAT Instruction .....	4-30
4.11 Reciprocal of Number, RCPF Instruction .....	4-31
4.12 Reciprocal of Square Root, RSQRF Instruction .....	4-33

## 4.1 Signed Integer Formats

The TMS320C40 supports two integer formats: a 16-bit short integer format and a 32-bit single-precision integer format. When extended-precision registers are used as integer operands, only bits 31–0 are used; bits 39–32 remain unchanged and unused.

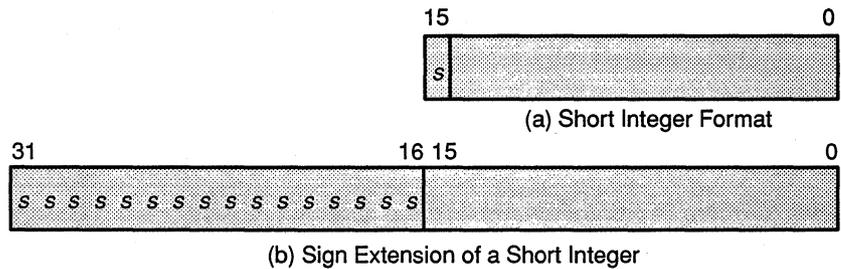
### 4.1.1 Short Integer Format

The short integer format is a 16-bit two's-complement integer format used for immediate integer operands. For those instructions that assume integer operands, this format is sign extended to 32 bits (see Figure 4–1). The range of an integer  $si$ , represented in the short integer format, is:

$$-2^{15} \leq si \leq 2^{15} - 1$$

In Figure 4–1 and other figures in this chapter, **s** = sign bit.

Figure 4–1. Short Integer Format and Sign Extension of Short Integer



### 4.1.2 Single-Precision Integer Format

In the single-precision integer format, the integer is represented in two's-complement notation. The range of an integer  $sp$ , represented in the single-precision integer format, is  $-2^{31} \leq sp \leq 2^{31} - 1$ . Figure 4–2 shows the single-precision integer format.

Figure 4–2. Single-Precision Integer Format



## 4.2 Unsigned-Integer Formats

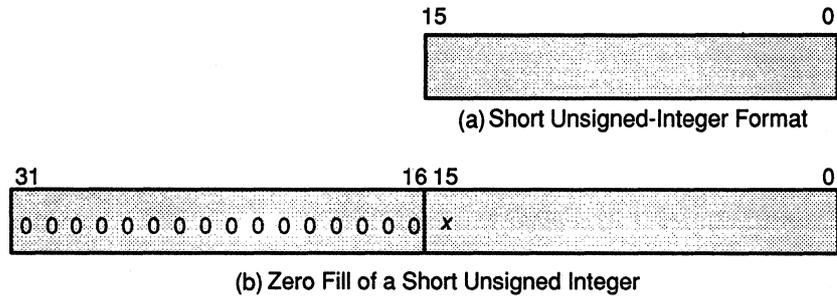
Two unsigned-integer formats are supported on the TMS320C40: a 16-bit short format and a 32-bit single-precision format. In extended-precision registers, the unsigned-integer operands use only bits 31–0; bits 39–32 remain unchanged.

### 4.2.1 Short Unsigned-Integer Format

Figure 4–3 shows the 16-bit, short, unsigned-integer format used for immediate unsigned-integer operands. For those instructions that assume unsigned-integer operands, this format is zero filled to 32 bits. In Figure 4–3 below,  $x$  = MSB (1 or 0).

4

Figure 4–3. Short Unsigned-Integer Format and Zero Fill



### 4.2.2 Single-Precision Unsigned-Integer Format

In the single-precision unsigned-integer format, the number is represented as a 32-bit value, as shown in Figure 4–4.

Figure 4–4. Single-Precision Unsigned-Integer Format



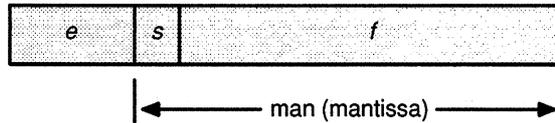
### 4.3 Floating-Point Formats

All TMS320C40 floating-point formats consist of three fields: an **exponent field (e)**, a **single-bit sign field (s)**, and a **fraction field (f)**. These are stored as shown in Figure 4–5. The exponent field is a twos-complement number. The sign field and fraction field may be considered as one unit and referred to as the **mantissa field (man)**. The mantissa is used to represent a normalized twos-complement number. In a normalized representation, a most significant nonsign bit is implied, thus providing an additional bit of precision. The value of a floating-point number  $x$  as a function of the fields  $e$ ,  $s$ , and  $f$  is given as

4

$$\begin{aligned}
 x &= 01.f \times 2^e && \text{if } s = 0 \\
 x &= 10.f \times 2^e && \text{if } s = 1 \\
 x &= 0 && \text{if } e = \text{most negative twos-complement} \\
 &&& \text{value or the specified exponent field width}
 \end{aligned}$$

Figure 4–5. Generic Floating-Point Format



Note:  $e$  = exponent field  
 $s$  = single-bit sign field  
 $f$  = fraction field

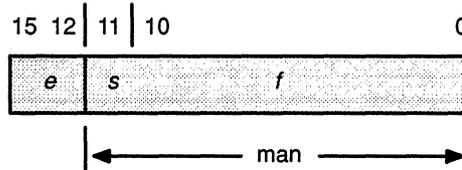
**Three floating-point formats are supported** on the TMS320C40:

- ❑ a short floating-point format (for immediate floating-point operands) consisting of a 4-bit exponent, 1 sign bit, and an 11-bit fraction,
- ❑ a single-precision format consisting of an 8-bit exponent, 1 sign bit, and a 23-bit fraction, and
- ❑ an extended-precision format consisting of an 8-bit exponent, 1 sign bit, and a 31-bit fraction.

### 4.3.1 Short Floating-Point Format

In the short floating-point format, floating-point numbers are represented by a two's-complement 4-bit exponent field (*e*) and a two's-complement 12-bit mantissa field (*man*) with an implied most significant nonsign bit.

Figure 4-6. Short Floating-Point Format



4

Operations are performed with an implied binary point between bits 11 and 10. When the implied most significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point two's-complement number *x* in the short floating-point format is given by

$$\begin{aligned}
 x &= 01.f \times 2^e && \text{if } s = 0 \\
 x &= 10.f \times 2^e && \text{if } s = 1 \\
 x &= 0 && \text{if } e = -8, s = 0, f = 0
 \end{aligned}$$

You must use the following reserved values to represent zero in the short floating-point format:

$$\begin{aligned}
 e &= -8 \\
 s &= 0 \\
 f &= 0
 \end{aligned}$$

The following examples illustrate the range and precision of the short floating-point format:

$$\begin{aligned}
 \text{Most Positive:} & \quad x = (2 - 2^{-11}) \times 2^7 = 2.5594 \times 10^2 \\
 \text{Least Positive:} & \quad x = 1 \times 2^{-7} = 7.8125 \times 10^{-3} \\
 \text{Least Negative:} & \quad x = (-1 - 2^{-11}) \times 2^{-7} = -7.8163 \times 10^{-3} \\
 \text{Most Negative:} & \quad x = -2 \times 2^7 = -2.5600 \times 10^2
 \end{aligned}$$

### 4.3.2 Single-Precision Floating-Point Format

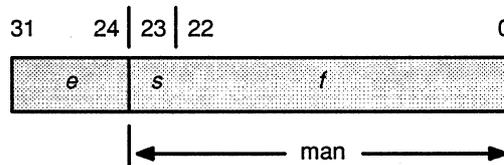
In the single-precision format, the floating-point number is represented by an 8-bit exponent field ( $e$ ) and a two's-complement 24-bit mantissa field ( $man$ ) with an implied most significant nonsign bit.

Operations are performed with an implied binary point between bits 23 and 22. When the implied most significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point number  $x$  is given by

$$\begin{aligned} x &= 01.f \times 2^e && \text{if } s = 0 \\ x &= 10.f \times 2^e && \text{if } s = 1 \\ x &= 0 && \text{if } e = -128, s = 0, f = 0 \end{aligned}$$

4

Figure 4-7. Single-Precision Floating-Point Format



You must use the following reserved values to represent zero in the single-precision floating-point format:

$$\begin{aligned} e &= -128 \\ s &= 0 \\ f &= 0 \end{aligned}$$

The following examples illustrate the range and precision of the single-precision floating-point format.

- Most Positive:  $x = (2 - 2^{-23}) \times 2^{127} = 3.4028234 \times 10^{38}$
- Least Positive:  $x = 1 \times 2^{-127} = 5.8774717 \times 10^{-39}$
- Least Negative:  $x = (-1 - 2^{-23}) \times 2^{-127} = -5.8774724 \times 10^{-39}$
- Most Negative:  $x = -2 \times 2^{127} = -3.4028236 \times 10^{38}$

### 4.3.3 Extended-Precision Floating-Point Format

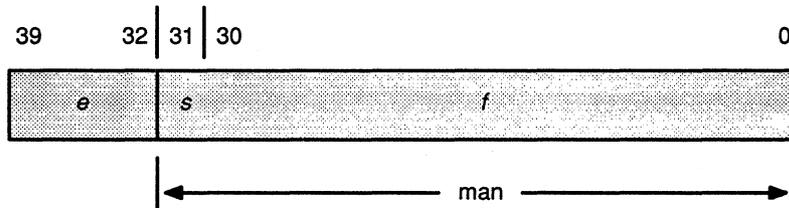
In the extended-precision format, the floating-point number is represented by an 8-bit exponent field ( $e$ ) and a 32-bit mantissa field ( $man$ ) with an implied most significant nonsign bit.

Operations are performed with an implied binary point between bits 31 and 30. When the implied most significant nonsign bit is made explicit, it is located to the immediate left of the binary point. The floating-point number  $x$  is given by:

4

$$\begin{aligned} x &= 01.f \times 2^e && \text{if } s = 0 \\ x &= 10.f \times 2^e && \text{if } s = 1 \\ x &= 0 && \text{if } e = -128, s = 0, f = 0 \end{aligned}$$

Figure 4-8. Extended-Precision Floating-Point Format



You must use the following reserved values to represent zero in the extended-precision floating-point format:

$$e = -128$$

$$s = 0$$

$$f = 0$$

The following examples illustrate the range and precision of the extended-precision floating-point format:

Most Positive:  $x = (2 - 2^{-31}) \times 2^{127} = 3.4028236683 \times 10^{38}$

Least Positive:  $x = 1 \times 2^{-127} = 5.8774717541 \times 10^{-39}$

Least Negative:  $x = (-1 - 2^{-31}) \times 2^{-127} = -5.8774717569 \times 10^{-39}$

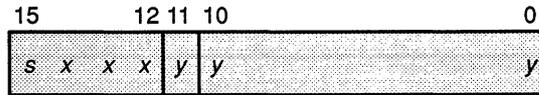
Most Negative:  $x = -2 \times 2^{127} = -3.4028236691 \times 10^{38}$

### 4.3.4 Conversion Between Floating-Point Formats

Floating-point operations assume several different formats for inputs and outputs. These formats often require conversion from one floating-point format to another (e.g., short floating-point format to extended-precision floating-point format). Format conversions occur automatically in hardware, with no overhead, as a part of the floating-point operations. Examples of the four conversions are shown below. When a floating-point format zero is converted to a greater precision format, it is always converted to a valid representation of zero in that format. In the figures below,  $s$  = sign bit of the exponent.

4

- ❑ Short floating-point format conversion to single-precision floating-point format.



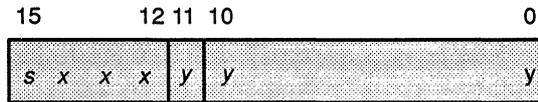
(a) Short Floating-Point Format



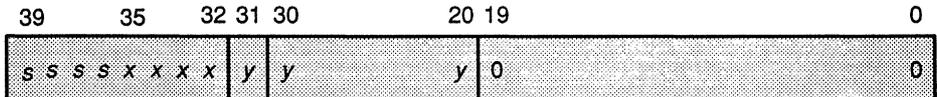
(b) Single-Precision Floating-Point Format

In this format, the exponent field is sign extended and the fraction field filled with zeros.

- ❑ Short floating-point format conversion to extended-precision floating-point format.



(a) Short Floating-Point Format



(b) Extended-Precision Floating-Point Format

The exponent field in this format is sign extended and the fraction field filled with zeros.

- Single-precision floating-point format conversion to extended precision floating-point format.



(a) Single-Precision Floating-Point Format



(b) Extended-Precision Floating-Point Format

The fraction field is filled with zeros.

- Extended-precision floating-point format conversion to single-precision floating-point format.



(a) Extended-Precision Floating-Point Format

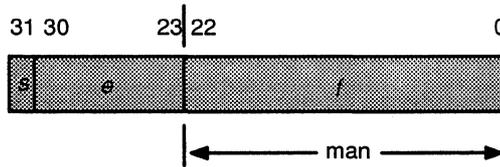


(b) Single-Precision Floating-Point Format

The fraction field is truncated.

### 4.4 Floating-Point Conversions (IEEE Std. 754/'C4x)

Figure 4–9. IEEE Single-Precision Std. 754 Floating-Point Format



This IEEE format is depicted in Figure 4–9 above. The following five cases define the value  $v$  of a number expressed in this format:

- 1) If  $e = 255$  and  $f \neq 0$ , then  $v = \text{NaN}^\dagger$
- 2) If  $e = 255$  and  $f = 0$ , then  $v = (-1)^s \text{infinite}$
- 3) If  $0 < e < 255$ , then  $v = (-1)^s \times 2^{e-127}(1.f)$
- 4) If  $e = 0$  and  $f \neq 0$ , then  $v = (-1)^s \times 2^{-126}(0.f)$
- 5) If  $e = 0$  and  $f = 0$ , then  $v = (-1)^s 0$  (zero).

where  $s$  = sign bit;  $e$  = the exponent field;  $f$  = the fraction field.

For the above five representations,  $e$  is treated as an unsigned integer. Case 1 generates NaN (not a number) and is primarily used for software signaling. Case 4 represents a denormalized number. Case 5 represents positive and negative zero.

4

Figure 4–10. TMS320C4x Single-Precision Twos-Complement Floating-Point Format ‡



In comparison, Figure 4–10 shows the the 'C40 twos-complement floating-point format. In this format, two cases can be used to define value  $v$  of a number:

- 1) If  $e = -128$  and  $f \neq 0$ , then  $v = 0$
- 2) If  $e \neq -128$  then  $v = 2^e(ss.f)$

where  $s$  = sign bit;  $e$  = the exponent field;  $f$  = the fraction field.

† NaN = not a number

‡ Same format as for the TMS320C3x

For this representation,  $e$  is treated as a twos-complement integer. The fraction and sign bit form a normalized twos-complement mantissa.

**Note: Symbols to Differentiate Between IEEE and C40 Formats**

In order to differentiate between the symbols used to define these two formats, all IEEE fields are subscripted with an IEEE (e.g.,  $e_{IEEE}$ ,  $s_{IEEE}$ , etc.). Similarly, all twos-complement fields are subscripted with a two (i.e.,  $e_{two}$ ,  $s_{two}$ ,  $f_{two}$ ).

**4 4.4.1 Converting IEEE Format to Twos-Complement Floating-Point Format**

The most common conversion is the IEEE to twos-complement format. This conversion is done according to rules in the following table:

Table 4-1. Rules for Converting IEEE Format to Twos-Complement Floating-Point Format

Case	If These Values Are Present			Then These Values Equal			
	$e_{IEEE}$	$s_{IEEE}$	$f_{IEEE}$	$e_{two}$	$s_{two}$	$f_{two}$	$s_{IEEE}$
1	255	1		7Fh	1	00 0000h	
2	255	0		7Fh	0	7F FFFFh	
3	$0 < e_{IEEE} < 255$	0		$e_{IEEE} - 7Fh$		$f_{IEEE}$	0
4	$0 < e_{IEEE} < 255$	1	$\neq 0$	$e_{IEEE} - 7Fh$		$\bar{f}_{IEEE} + 1^{\dagger}$	1
5	$0 < e_{IEEE} < 255$	1	0	$e_{IEEE} - 80h$		0	1
6	0			80h	0	00 0000h	

$\dagger \bar{f}_{IEEE}$  = ones complement of  $f_{IEEE}$ .

**Case 1** maps the IEEE positive NaNs and positive infinity to the single-precision twos-complement most positive number. Overflow is also signaled to allow you to check for these special cases.

**Case 2** maps the IEEE negative NaNs and negative infinity to the single-precision twos-complement most negative number. Overflow is also signaled to allow you to check for these special cases.

**Case 3** maps the IEEE positive normalized numbers to the identical value in the twos-complement positive number.

**Case 4** maps the IEEE negative normalized numbers with a nonzero fraction to the identical value in the twos-complement negative number.

**Case 5** maps the IEEE negative normalized numbers with a zero fraction to the identical value in the twos-complement negative number.

**Case 6** maps the IEEE positive and negative denormalized numbers and positive and negative zeroes to a twos-complement zero.

The TMS320C40 assumes that the IEEE numbers are stored as an integer in memory or in a register. When converted, they are always placed in an extended-precision register by using the exponent and fraction fields of these registers. Any arithmetic operations that are performed on the fraction field of the IEEE number should be performed **only** on the IEEE **fraction field**. The eight LSBs of the extended-precision register are set to zero.

#### 4.4.2 Converting Twos-Complement Floating-Point Format to IEEE Format

This conversion is done according to rules in the following table:

Table 4-2. Rules for Converting Twos-Complement Floating Point Format to IEEE Format

Case	If These Values Are Present			Then These Values Equal		
	$e_{two}$	$s_{two}$	$f_{two}$	$e_{IEEE}$	$s_{IEEE}$	$f_{IEEE}$
1	-128			00h	0	00 0000h
2	-127			00h	0	00 0000h
3	$-126 \leq e_{two} \leq 127$	0		$e_{two}+7Fh$	0	$f_{two}$
4	$-126 \leq e_{two} \leq 127$	1	$\neq 0$	$e_{two}+7Fh$	0	$\bar{f}_{two}+1^\dagger$
5	$-126 \leq e_{two} \leq 127$	1	0	$e_{two}+7Eh$	1	00 0000h
6	127	1	0	FFh	1	00 0000h

$^\dagger \bar{f}_{two}$  = ones complement of  $f_{two}$ .

**Case 1** maps a twos-complement zero to a positive IEEE zero.

**Case 2** maps the twos-complement numbers that are too small to be represented as normalized IEEE numbers to a positive IEEE zero.

**Case 3** maps the positive twos-complement numbers that are not covered by case 2 into the identically valued IEEE number.

**Case 4** maps the negative twos-complement numbers with a nonzero fraction that are not covered in case 2 into the identically valued IEEE number.

**Case 5** maps all the negative twos-complement numbers with a zero fraction, except for the most negative twos-complement number and those that are not covered in case 2, into the identically valued IEEE number.

**Case 6** maps the most negative twos-complement number to the IEEE negative infinity.

The TMS320C4x assumes that the twos-complement numbers are in memory or are in an extended-precision register using the exponent and fraction field of the register (shown in Figure 4–10 on page 4-11). If the value is in an extended-precision register, then only the 24 MSBs of the fraction field are manipulated as the fraction field and for detection of the special cases. The result of the conversion goes to a register as an integer.

## 4.5 Floating-Point Multiplication

A floating-point number  $\alpha$  can be written in floating-point format as in the following formula, where  $\alpha(\text{man})$  is the mantissa and  $\alpha(\text{exp})$  is the exponent.

$$\alpha = \alpha(\text{man}) \times 2^{\alpha(\text{exp})}$$

The product of  $\alpha$  and  $b$  is  $c$ , defined as

$$c = \alpha \times b = \alpha(\text{man}) \times b(\text{man}) \times 2^{(\alpha(\text{exp})+b(\text{exp}))}$$

$$c(\text{man}) = \alpha(\text{man}) \times b(\text{man})$$

$$c(\text{exp}) = \alpha(\text{exp}) + b(\text{exp})$$

4

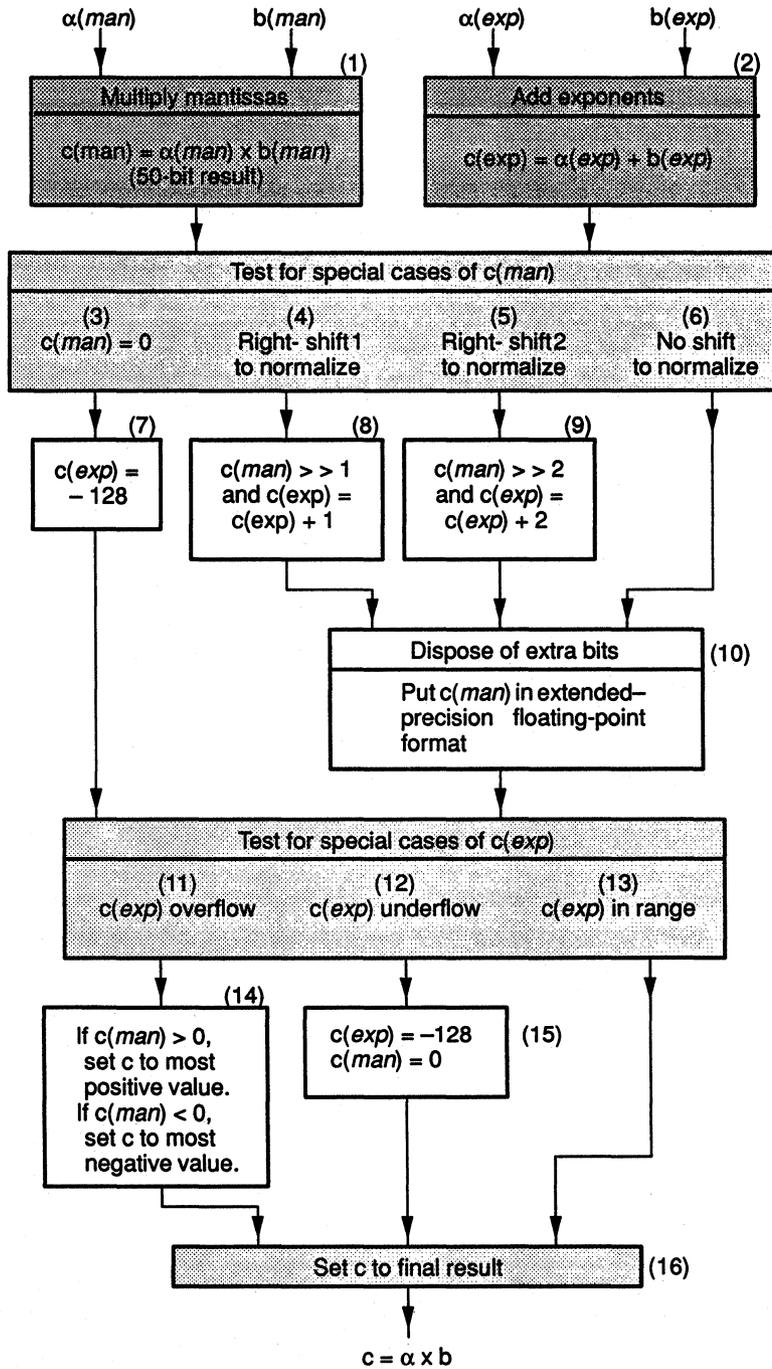
During floating-point multiplication, source operands are always assumed to be in the extended-precision floating-point format:

If the source of the operands is in **short floating-point format**, it is *extended* to the extended-precision floating-point format.

If the source of the operands is in **single-precision floating-point format**, it is *extended* to extended-precision format.

These conversions occur automatically in hardware with no overhead. All results of floating-point multiplications are in the extended-precision format. These multiplications occur in a single cycle.

Figure 4-11. Flowchart for Floating-Point Multiplication



4

Figure 4–11 is a flowchart showing floating-point multiplication:

- 1) In **step 1** (steps are shown as numbers in parentheses), the 32-bit source operand mantissas are multiplied, producing a 64-bit result  $c(man)$ . (Note that input and output data are always represented as normalized numbers.)
- 2) In **step 2**, the exponents are added, yielding  $c(exp)$ .
- 3) **Steps 3 through 6** check for special cases.
- 4) **Step 3** checks for whether  $c(man)$  in extended-precision format is equal to zero. If  $c(man)$  is zero, **step 7** sets  $c(exp)$  to  $-128$ , thus yielding the representation for zero.
- 5) **Steps 4 and 5** normalize the result.
- 6) If a right shift of one is necessary, then in **step 8**,  $c(man)$  is right-shifted one bit, and one is added to  $c(exp)$ .
- 7) If a right shift of two is necessary, then in **step 9**,  $c(man)$  is right-shifted two bits, and two is added to  $c(exp)$ . **Step 6** occurs when the result is normalized.
- 8) In **step 10**,  $c(man)$  is set in the extended-precision floating-point format.
- 9) **Steps 11 through 16** check for special cases of  $c(exp)$ .
- 10) In **step 14**, if  $c(exp)$  has overflowed (**step 11**) in the positive direction, then  $c(exp)$  is set to the most positive extended-precision format value. If  $c(exp)$  has overflowed in the negative direction, then  $c(exp)$  is set to the most negative extended-precision format value.
- 11) If  $c(exp)$  has underflowed (**step 12**), then  $c$  is set to zero (**step 15**); i.e.,  $c(man) = 0$  and  $c(exp) = -128$ .



To place this number in the proper normalized format, it is necessary to shift the mantissa one place to the right and add one to the exponent. This yields

$$\begin{array}{r}
 01.100000000000000000000000 \times 2^{\alpha(\text{exp})} \\
 \times 01.100000000000000000000000 \times 2^{\text{b}(\text{exp})} \\
 \hline
 01.001000 \times 2^{(\alpha(\text{exp})+\text{b}(\text{exp})+1)}
 \end{array}$$

**Example 4-3. Floating-Point Multiply (Both Mantissas = 1.0)**

Let

$$\begin{aligned}
 \alpha &= 1.0 \times 2^{\alpha(\text{exp})} = 01.000000000000000000000000 \times 2^{\alpha(\text{exp})} \\
 \text{b} &= 1.0 \times 2^{\text{b}(\text{exp})} = 01.000000000000000000000000 \times 2^{\text{b}(\text{exp})}
 \end{aligned}$$

4

where a and b are both represented in binary form according to the single-precision floating-point format. Then

$$\begin{array}{r}
 01.000000000000000000000000 \times 2^{\alpha(\text{exp})} \\
 \times 01.000000000000000000000000 \times 2^{\text{b}(\text{exp})} \\
 \hline
 0001.000 \times 2^{(\alpha(\text{exp})+\text{b}(\text{exp}))}
 \end{array}$$

This number is in the proper normalized format. Therefore, no shift of the mantissa or modification of the exponent is necessary.

These examples have shown cases where the product of two normalized numbers can be normalized with a shift of zero, one, or two. For all normalized inputs with the floating-point format used by the TMS320C40, a normalized result can be produced by a shift of zero, one, or two.

**Example 4-4. Floating-Point Multiply Between Positive and Negative Numbers**

Let

$$\begin{aligned}
 \alpha &= 1.0 \times 2^{\alpha(\text{exp})} = 01.000000000000000000000000 \times 2^{\alpha(\text{exp})} \\
 \text{b} &= -2.0 \times 2^{\text{b}(\text{exp})} = 10.000000000000000000000000 \times 2^{\text{b}(\text{exp})}
 \end{aligned}$$

Then

$$\begin{array}{r}
 01.000000000000000000000000 \times 2^{\alpha(\text{exp})} \\
 \times 10.000000000000000000000000 \times 2^{\text{b}(\text{exp})} \\
 \hline
 1110.000 \times 2^{(\alpha(\text{exp})+\text{b}(\text{exp}))}
 \end{array}$$

The result is  $c = -2.0 \times 2^{(\alpha(\text{exp}) + \text{b}(\text{exp}))}$

**Example 4-5. Floating-Point Multiply by Zero**

All multiplications by a floating-point zero yield a result of zero ( $f=0$ ,  $s=0$ , and  $\text{exp} = -128$ ).

## 4.6 Floating-Point Addition and Subtraction

In floating-point addition and subtraction, two floating-point numbers  $\alpha$  and  $b$  can be defined as

$$\alpha = \alpha(\text{man}) \times 2^{\alpha(\text{exp})}$$

$$b = b(\text{man}) \times 2^{b(\text{exp})}$$

The sum (or difference) of  $\alpha$  and  $b$  can be defined as

$$c = \alpha \pm b$$

$$= (\alpha(\text{man}) \pm (b(\text{man}) \times 2^{-(\alpha(\text{exp})-b(\text{exp}))})) \times 2^{\alpha(\text{exp})},$$

$$\text{if } \alpha(\text{exp}) \geq b(\text{exp})$$

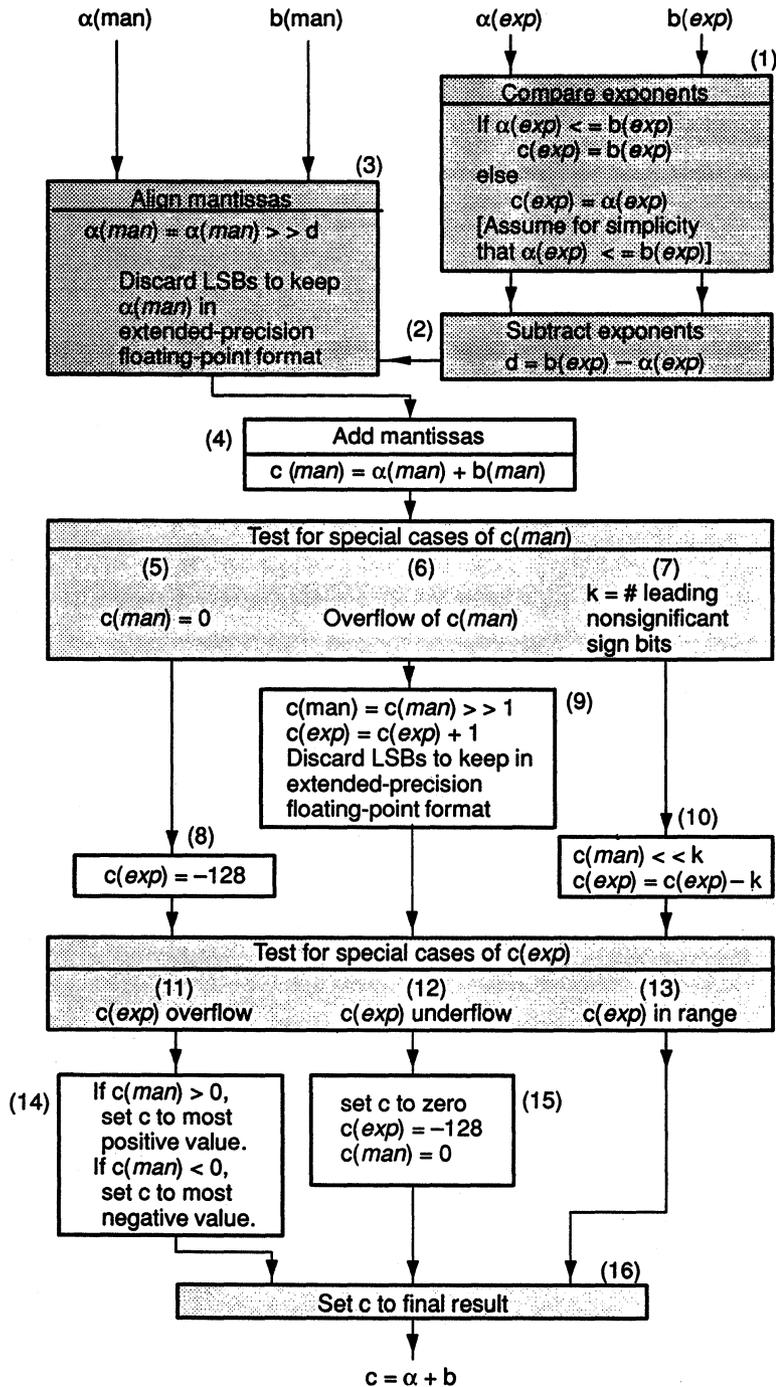
$$= ((\alpha(\text{man}) \times 2^{-(b(\text{exp})-\alpha(\text{exp}))}) \pm b(\text{man})) \times 2^{b(\text{exp})},$$

$$\text{if } \alpha(\text{exp}) < b(\text{exp})$$

Figure 4–12 is the flowchart for floating-point addition. Since this flowchart assumes signed data, it is also appropriate for floating-point subtraction. In this figure, it is assumed that  $\alpha(\text{exp}) \leq b(\text{exp})$ . In step 1 (steps are numbers in parentheses), the source exponents are compared, and  $c(\text{exp})$  is set equal to the largest of the two source exponents. In step 2,  $d$  is set to the difference of the two exponents. In step 3, the mantissa with the smallest exponent, in this case  $\alpha(\text{man})$ , is right shifted  $d$  bits in order to align the mantissas. After the mantissas have been aligned, they are added (step 4).

Steps 5 through 7 check for a special case of  $c(\text{man})$ . If  $c(\text{man})$  is zero (step 5), then  $c(\text{exp})$  is set to its most negative value (step 8) to yield the correct representation of zero. If  $c(\text{man})$  has overflowed  $c$  (step 6), then in step 9,  $c(\text{man})$  is right shifted one bit, and one is added to  $c(\text{exp})$ . In step 10, the result is normalized. In steps 11 and 12, special cases of  $c(\text{exp})$  are tested. If  $c(\text{exp})$  has overflowed, then  $c$  is set to the most positive extended-precision value if it is positive; otherwise, it is set to the most negative extended-precision value.

Figure 4-12. Flowchart for Floating-Point Addition







## 4.7 Normalization (NORM Instruction)

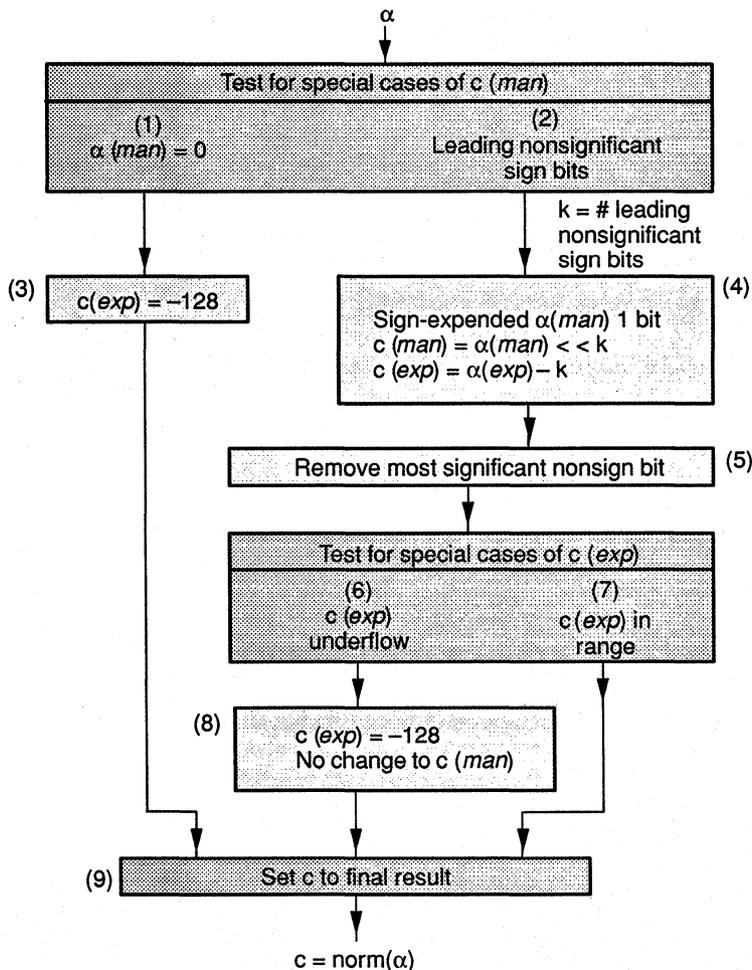
The NORM instruction normalizes an extended-precision floating-point number that is assumed to be unnormalized. Since the number is assumed to be unnormalized, no implied most significant nonsign bit is assumed. The NORM instruction executes the following three steps:

- 1) Locates the most significant nonsign bit of the floating-point number.
- 2) Left shifts to normalize the number.
- 3) Adjusts the exponent.

Given the extended-precision floating-point value  $\alpha$  to be normalized, the normalization,  $\text{norm}(\ )$ , is performed as shown in Figure 4-13.

4

Figure 4-13. Flowchart for NORM Instruction Operation





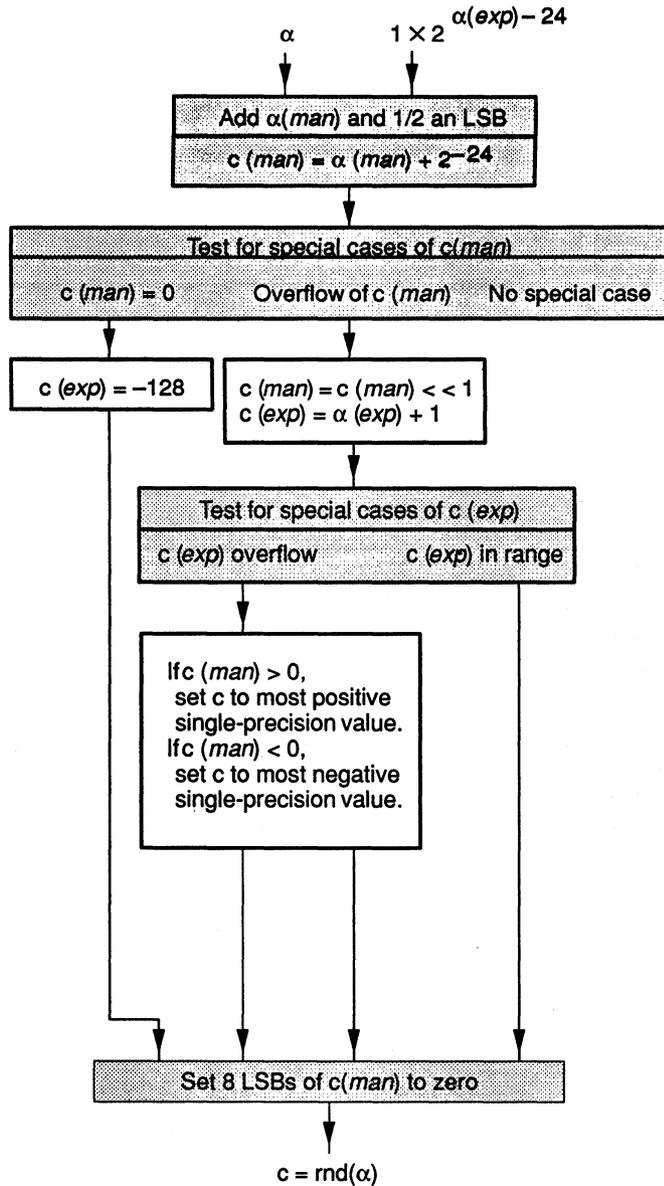
## 4.8 Rounding (RND Instruction)

The RND instruction rounds a number from the extended-precision floating-point format to the single-precision floating-point format. Rounding is similar to floating-point addition. Given the number  $\alpha$  to be rounded, the following operation is performed first.

$$c = \alpha(man) \times 2^{\alpha(exp)} + (1 \times 2^{\alpha(exp)-24})$$

Next, a conversion from extended-precision floating-point to single-precision floating-point format is performed. Given the extended-precision floating-point value, the rounding,  $\text{rnd}()$ , is performed as shown in Figure 4–14.

Figure 4-14. Flowchart for Floating-Point Rounding by the RND Instruction



4

## 4.9 Floating-Point-to-Integer Conversion (FIX Instruction)

Floating-point to integer conversion, using the FIX instructions, allows extended-precision floating-point numbers to be converted to single-precision integers in a single cycle. The floating-point to integer conversion of the value  $x$  is referred to here as  $\text{fix}(x)$ . The conversion does not overflow if  $\alpha$ , the number to be converted, is in the range

$$-2^{31} \leq \alpha \leq 2^{31} - 1$$

First, you must be certain that

$$\alpha(\text{exp}) \leq 30$$

If these bounds are not met, an overflow occurs. If an overflow occurs in the positive direction, the output is the most positive integer. If an overflow occurs in the negative direction, the output is the most negative integer. If  $\alpha(\text{exp})$  is within the valid range, then  $\alpha(\text{man})$ , with implied bit included, is sign-extended and right-shifted ( $\text{rs}$ ) by the amount

$$\text{rs} = 31 - \alpha(\text{exp})$$

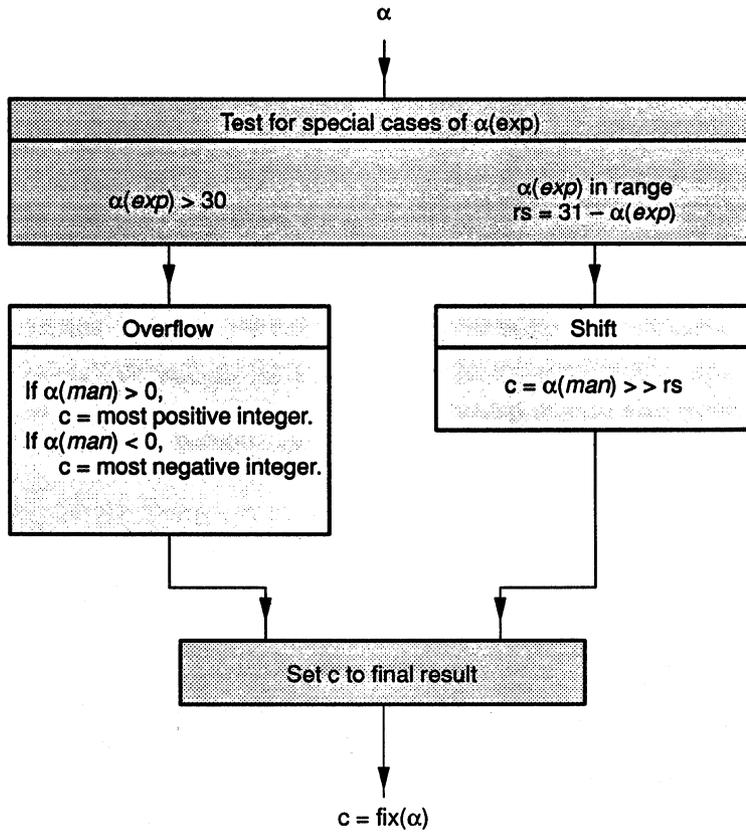
This right shift ( $\text{rs}$ ) shifts out those bits corresponding to the fractional part of the mantissa. For example:

If  $0 \leq x < 1$ , then  $\text{fix}(x) = 0$ .

If  $-1 \leq x < 0$ , then  $\text{fix}(x) = -1$ .

The flowchart for the floating-point to integer conversion is shown in Figure 4-15.

Figure 4-15. Flowchart for Floating-Point-to-Integer Conversion by FIX Instructions



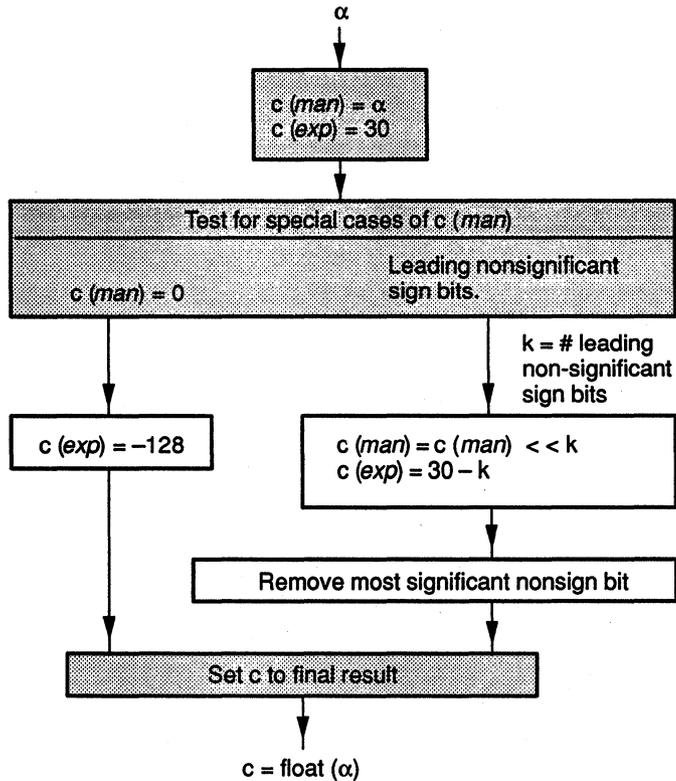
4

## 4.10 Integer-to-Floating-Point Conversion (FLOAT Instruction)

Integer to floating-point conversion, using the FLOAT instruction, allows single-precision integers to be converted to extended-precision floating-point numbers. The flowchart for this conversion is shown in Figure 4-16.

Figure 4-16. Flowchart for Integer-to-Floating-Point Conversion by FLOAT Instructions

4



## 4.11 Reciprocal (RCPF Instruction)

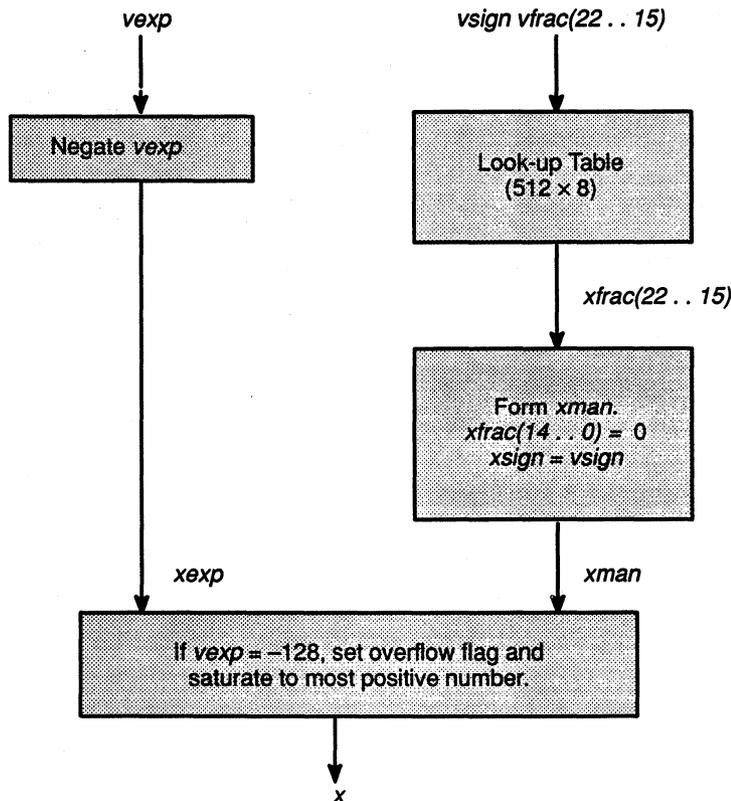
The RCPF instruction generates a satisfactory estimate of the reciprocal of a floating-point number. The estimate has the correct exponent, and the mantissa is often accurate to the eighth binary position (mantissa error is thus  $< 2^{-8}$ ). Also, this estimate may be used as a seed for an algorithm to compute the reciprocal to even greater accuracy. (The Newton–Raphson algorithm, described in this section, is one such case.)

Figure 4–17 below depicts the algorithm used by instruction RCPF.

- ❑ The input is assumed to be  $v = vman \times 2^{vexp}$ .
- ❑ The output is assumed to be  $x = xman \times 2^{xexp}$ .
- ❑  $vexp$  is negated.
- ❑ If  $vexp = -128$ , the result is saturated to the most positive number, and the overflow flag is set. The N condition flag is set to the same sign as  $vsign$ .

4

Figure 4–17. RCPF Instruction Algorithm





## 4.12 Reciprocal Square Root (RSQRF Instruction)

The RSQRF instruction generates an estimated reciprocal of the square root of a floating-point number. It parallels some of the operational characteristics of the RCPF instruction (Section 4.11) in that the RSQRF:

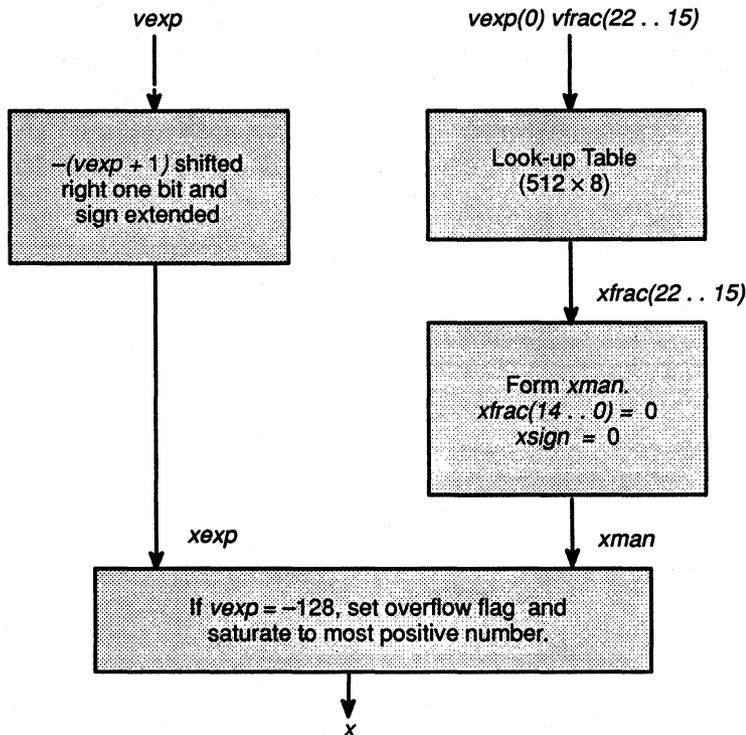
- ❑ it generates an estimate (in this case the reciprocal of the square root of a floating-point number),
- ❑ the mantissa is accurate to the eighth binary place (mantissa error is  $< 2^{-8}$ ), and
- ❑ often, this is a satisfactory estimate of the reciprocal of a number's square root; in other cases, it may be used as a seed for an algorithm that computes the reciprocal square root to an even greater accuracy.

4

Figure 4-19 depicts the RSQRF algorithm.

- ❑ The input is assumed to be  $v = vman \times 2^{vexp}$ .
- ❑ The output is assumed to be  $x = xman \times 2^{xexp}$ .
- ❑  $vexp + 1$  is negated and shifted right one bit with sign extension.
- ❑ If  $vexp = -128$ , the result is saturated to the most positive number, and the overflow flag is set.

Figure 4-19. RSQRF Instruction Algorithm





The TMS320C4x program to implement this algorithm is shown in Figure 4–20. Each step of the algorithm is labeled, and the corresponding accuracy achieved is noted at the end of the step. The algorithm takes only ten machine cycles (compared to 30 cycles on the 'C3x without a look-up table).

#### 4.12.2 Background on the Reciprocal Square Root

In many applications, normalization of data values is necessary. Often, the normalizing factor is the square root of another quantity. For example, when one vector is given, the unit vector in the same direction as the original vector can be found by normalizing the original vector by the length of the vector. This involves division by a square root. The 'C40 provides a simple way to directly determine this quantity, instead of going through a two-step approach of finding the square root and then finding the reciprocal of the square root.

4

Of course, given the result of this algorithm, the square root is found by a simple multiplication:

$$v = v x[n]$$

where  $x[n]$  is the estimate of  $1/\sqrt{v}$  as determined by the Newton–Raphson algorithm or some other algorithm.



## Addressing

---

---

---

---

The TMS320C40 supports five groups of powerful addressing modes. Six types of addressing may be used within the groups, which facilitates access of data from memory, registers, and the instruction word. This chapter details the operation, encoding, and implementation of the addressing modes. It also discusses the management of system stacks, queues, and deques in memory. The major topics in this chapter:

<b>Section</b>	<b>Page</b>
5.1 Types of Addressing .....	5-2
■ Register .....	5-3
■ Direct .....	5-4
■ Indirect .....	5-5
■ Immediate .....	5-17
■ PC-Relative .....	5-17
5.2 Groups of Addressing Modes .....	5-19
■ General Addressing Modes .....	5-19
■ Three-Operand Addressing Modes .....	5-20
■ Parallel Addressing Modes .....	5-23
■ Conditional-Branch Addressing Modes .....	5-24
5.3 Circular Addressing .....	5-25
5.4 Bit-Reversed Addressing .....	5-30
5.5 System Stack Management .....	5-31

## 5.1 Types of Addressing

Five **types of addressing** allow access of data from memory, registers, and the instruction word:

	<u>Sub-section</u>	<u>Page</u>
<input type="checkbox"/> Register addressing	5.1.1	5-3
<input type="checkbox"/> Direct addressing	5.1.2	5-4
<input type="checkbox"/> Indirect addressing	5.1.3	5-5
<input type="checkbox"/> Immediate addressing	5.1.4	5-17
<input type="checkbox"/> PC Relative addressing	5.1.5	5-17

Some types of addressing are appropriate for some instructions and not others. For this reason, the types of addressing are used in the four different **groups of addressing modes** as follows:

5

	<u>Sub-section</u>	<u>Page</u>
<input type="checkbox"/> General addressing modes (G):	5.2.1	5-19
<input type="checkbox"/> Register		
<input type="checkbox"/> Direct		
<input type="checkbox"/> Indirect		
<input type="checkbox"/> Immediate		
<input type="checkbox"/> Three-operand addressing modes (T):	5.2.2	5-20
<input type="checkbox"/> Register		
<input type="checkbox"/> Immediate		
<input type="checkbox"/> Indirect		
<input type="checkbox"/> Parallel addressing modes (P):	5.2.3	5-23
<input type="checkbox"/> Register		
<input type="checkbox"/> Indirect		
<input type="checkbox"/> Conditional-branch addressing modes (B):	5.2.4	5-24
<input type="checkbox"/> Register		
<input type="checkbox"/> PC-relative		

The six types of addressing are discussed first (subsections 5.1.1 through 5.1.5, beginning on the next page), followed by the five groups of addressing modes (section 5.2, beginning on page 5-19).

### 5.1.1 Register Addressing

In register addressing, a CPU register contains the operand, as shown in this example:

```
ABSF  R1      ; R1 = |R1|
```

The syntax for the CPU registers, the assembler syntax, and the assigned function for those registers are listed in Table 5–1.

Table 5–1. CPU Register/Assembler Syntax and Function

Register Machine Value	Assembler Syntax	Assigned Function
00h	R0	Extended-precision register
01h	R1	Extended-precision register
02h	R2	Extended-precision register
03h	R3	Extended-precision register
04h	R4	Extended-precision register
05h	R5	Extended-precision register
06h	R6	Extended-precision register
07h	R7	Extended-precision register
1Ch	R8	Extended-precision register
1Dh	R9	Extended-precision register
1Eh	R10	Extended-precision register
1Fh	R11	Extended-precision register
08h	AR0	Auxiliary register 0
09h	AR1	Auxiliary register 1
0Ah	AR2	Auxiliary register 2
0Bh	AR3	Auxiliary register 3
0Ch	AR4	Auxiliary register 4
0Dh	AR5	Auxiliary register 5
0Eh	AR6	Auxiliary register 6
0Fh	AR7	Auxiliary register 7
10h	DP	Data-page pointer
11h	IR0	Index register 0
12h	IR1	Index register 1
13h	BK	Block-size register
14h	SP	Active stack pointer
15h	ST	Status register
16h	DIE	DMA coprocessor interrupt enable
17h	IIE	Internal interrupt enable register
18h	IIF	IIOF pins and interrupt flag register
19h	RS	Repeat start address
1Ah	RE	Repeat end address
1Bh	RC	Repeat counter

## 5.1.2 Direct Addressing

In direct addressing, the data address is formed by the concatenation of the 16 least significant bits of the data page pointer (DP) with the 16 least significant bits of the instruction word (*expr*). This results in 65536 pages (64K words per page), giving you a large address space without requiring a change of the page pointer. The syntax and operation for direct addressing are listed below.

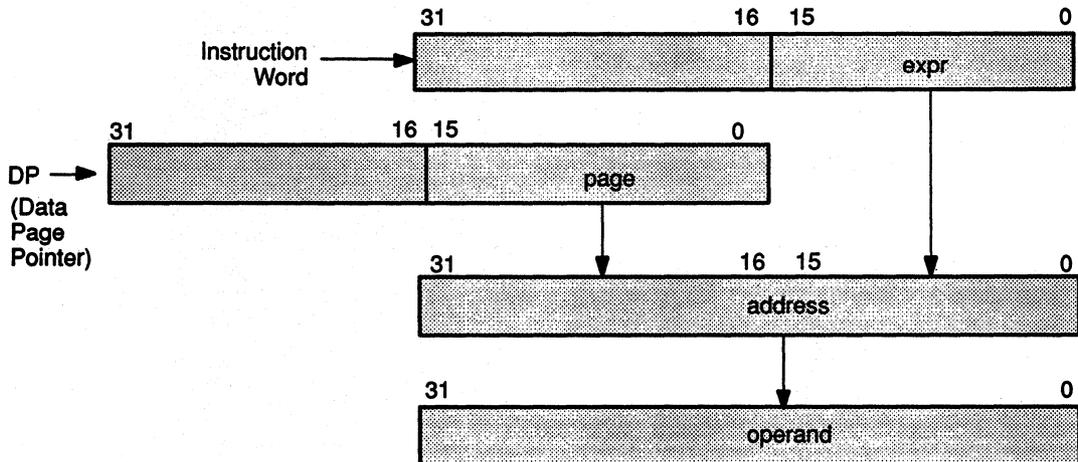
**Syntax:**        @*expr*

**Operation:**    address = DP concatenated with *expr*

Figure 5–1 shows the formation of the data address. Example 5–1 gives an instruction example with data before and after instruction execution.

5

Figure 5–1. Direct Addressing



Example 5–1. Direct Addressing

```
ADDI    @0BCDEh, R7
```

**Before Instruction:**

DP = 108Ah

R7 = 11h

Data at 108A BCDEh = 1234 5678h

**After Instruction:**

DP = 108Ah

R7 = 1234 5689h

Data at 108A BCDEh = 1234 5678h

### 5.1.3 Indirect Addressing

Indirect addressing is used to specify the address of an operand in memory through the contents of an auxiliary register, optional displacements, and index registers. This arithmetic is performed by the auxiliary register arithmetic units (ARAUs) and is unsigned. (All 32 bits of the auxiliary and index registers are used in indirect addressing.)

The flexibility of indirect addressing is possible because the ARAUs on the TMS320C40 are used to modify auxiliary registers in parallel with operations within the main CPU. Indirect addressing is specified by a five-bit field in the instruction word, referred to as the **mod field** (in the left side of Table 5–2 on page 5-6 as well as in the examples that follow). A **displacement** is either an explicit unsigned 5-bit or 8-bit integer contained in the instruction word or an implicit displacement of one. Two index registers, IR0 and IR1, can also be used in indirect addressing, enabling the use of 32-bit indirect displacements. In some cases, an addressing scheme using circular or bit-reversed addressing is optional. The mechanism for generating addresses in circular addressing is discussed in Section 5.3, bit-reversed in Section 5.4.

Table 5–2 lists the various kinds of indirect addressing, along with the value of the modification (mod) field, assembler syntax, operation, and function for each. The succeeding 18 examples show the operation for each kind of indirect addressing.

Table 5-2. Indirect Addressing

Mod Field	Syntax	Operation	Description
<b>Indirect Addressing with Displacement</b>			
00000	*+ARn(displacement)	addr = ARn + disp	With predisplacement add
00001	*- ARn(displacement)	addr = ARn - disp	With predisplacement subtract
00010	*++ARn(displacement)	addr = ARn + disp ARn = ARn + disp	With predisplacement add and modify
00011	*-- ARn(displacement)	addr = ARn - disp ARn = ARn - disp	With predisplacement subtract and modify
00100	*ARn++(displacement)	addr = ARn ARn = ARn + disp	With postdisplacement add and modify
00101	*ARn--(displacement)	addr = ARn ARn = ARn - disp	With postdisplacement subtract and modify
00110	*ARn++(displacement)%	addr = ARn ARn = circ(ARn + disp)	With postdisplacement add and circular modify
00111	*ARn--(displacement)%	addr = ARn ARn = circ(ARn - disp)	With postdisplacement subtract and circular modify
<b>Indirect Addressing with Index Register IRO</b>			
01000	*+ARn(IRO)	addr = ARn + IRO	With preindex (IRO) add
01001	*- ARn(IRO)	addr = ARn - IRO	With preindex (IRO) subtract
01010	*++ARn(IRO)	addr = ARn + IRO ARn = ARn + IRO	With preindex (IRO) add and modify
01011	*-- ARn(IRO)	addr = ARn - IRO ARn = ARn - IRO	With preindex (IRO) subtract and modify
01100	*ARn++(IRO)	addr = ARn ARn = ARn + IRO	With postindex (IRO) add and modify
01101	*ARn--(IRO)	addr = ARn ARn = ARn - IRO	With postindex (IRO) subtract and modify
01110	*ARn++(IRO)%	addr = ARn ARn = circ(ARn + IRO)	With postindex (IRO) add and circular modify
01111	*ARn--(IRO)%	addr = ARn ARn = circ(ARn) - IRO	With postindex (IRO) subtract and circular modify

**LEGEND:**

addr = memory address  
 ARn = auxiliary register AR0 - AR7  
 IRn = index register IRO or IR1  
 disp = displacement (5 bits or 8 bits on 'C40)  
 ++ = add and modify  
 -- = subtract and modify  
 circ() = address in circular addressing  
 % = where circular addressing is performed

Table 5-2. Indirect Addressing (Concluded)

Mod Field	Syntax	Operation	Description
<b>Indirect Addressing with Index Register IR1</b>			
10000	*+ ARn(IR1)	addr = ARn + IR1	With preindex (IR1) add
10001	*- ARn(IR1)	addr = ARn - IR1	With preindex (IR1) subtract
10010	*++ ARn(IR1)	addr = ARn + IR1 ARn = ARn + IR1	With preindex (IR1) add and modify
10011	*-- ARn(IR1)	addr = ARn - IR1 ARn = ARn - IR1	With preindex (IR1) subtract and modify
10100	* ARn ++ (IR1)	addr = ARn ARn = ARn + IR1	With postindex (IR1) add and modify
10101	*ARn -- (IR1)	addr = ARn ARn = ARn - IR1	With postindex (IR1) subtract and modify
10110	* ARn ++ (IR1)%	addr = ARn ARn = circ(ARn + IR1)	With postindex (IR1) add and circular modify
10111	* ARn -- (IR1)%	addr = ARn ARn = circ(ARn - IR1)	With postindex (IR1) subtract and circular modify
<b>Indirect Addressing (Special Cases)</b>			
11000	*ARn	addr = ARn	Indirect
11001	*ARn ++ (IR0)B	addr = ARn ARn = B(ARn + IR0)	With postindex (IR0) add and bit-reversed modify

5

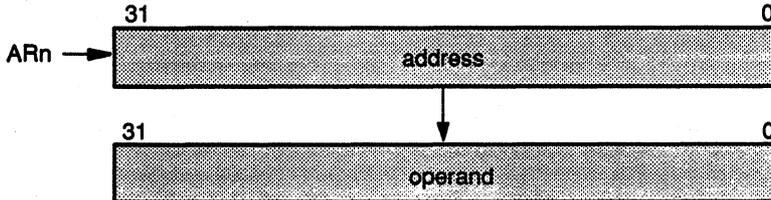
**LEGEND:**

- addr = memory address
- ARn = auxiliary register AR0 - AR7
- IRn = index register IR0 or IR1
- disp = displacement
- ++ = add and modify
- = subtract and modify
- circ( ) = address in circular addressing
- % = where circular addressing is performed
- B = where bit-reversed addressing is performed

**Example 5-2. Auxiliary Register Indirect**

An auxiliary register (AR $n$ ) contains the address of the operand to be fetched.

**Operation:** operand address = AR $n$   
**Assembler Syntax:** \*AR $n$   
**Modification Field:** 11000

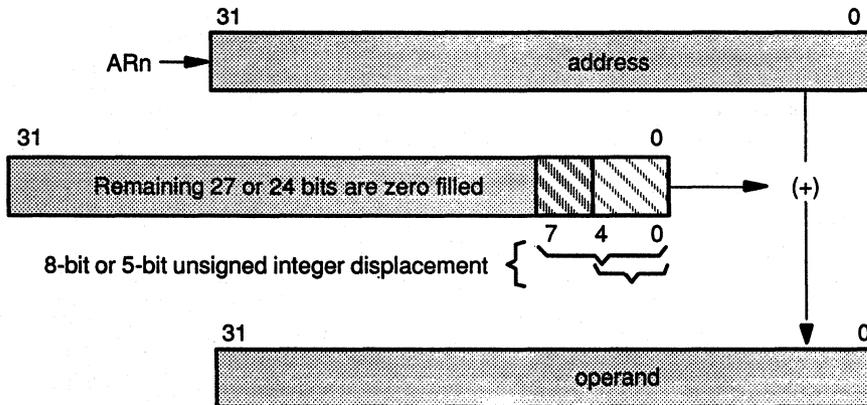


5

**Example 5-3. Indirect With Predisplacement Add**

The address of the operand to be fetched is the sum of an auxiliary register (AR $n$ ) and the displacement (disp). The displacement is either a 5-bit or 8-bit unsigned integer contained in the instruction word or an implied value of 1.

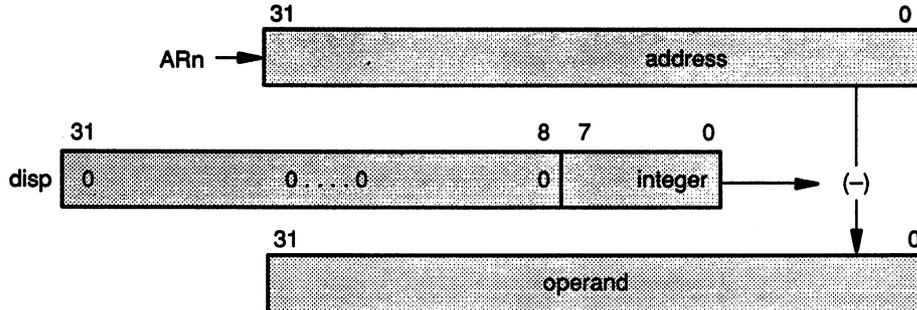
**Operation:** operand address = AR $n$  + disp  
**Assembler Syntax:** \*+AR $n$ (disp)  
**Modification Field:** 00000



**Example 5-4. Indirect With Predisplacement Subtract**

The address of the operand to be fetched is the contents of an auxiliary register (ARn) minus the displacement (disp). The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

**Operation:** operand address =  $ARn - disp$   
**Assembler Syntax:**  $*- ARn(displ)$   
**Modification Field:** 00001

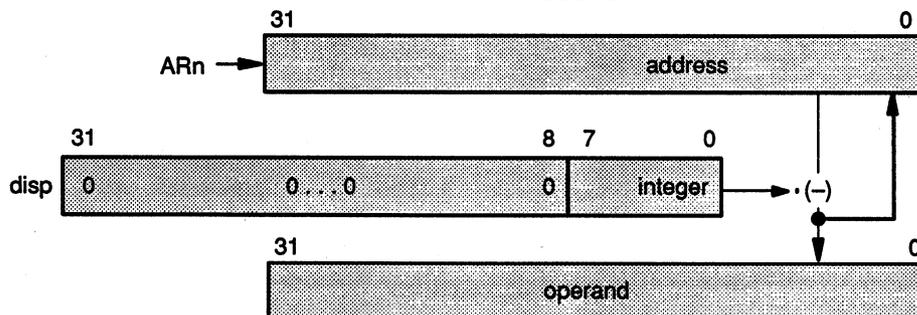


5

**Example 5-5. Indirect With Predisplacement Add and Modify**

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and the displacement (disp). The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1. After the data is fetched, the auxiliary register is updated with the address generated.

**Operation:** operand address =  $ARn + disp$   
 $ARn = ARn + disp$   
**Assembler Syntax:**  $*++ ARn(displ)$   
**Modification Field:** 00010



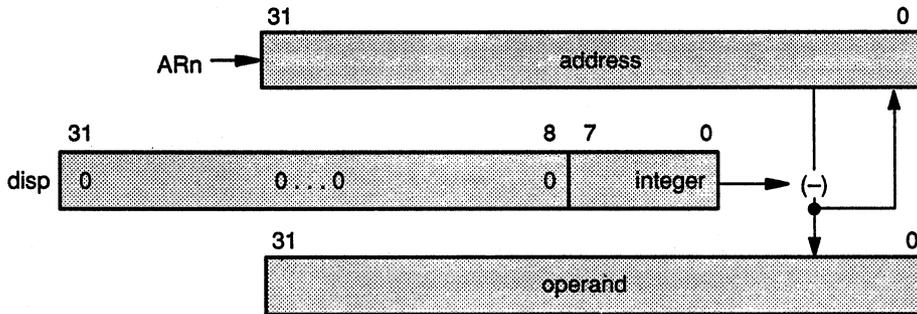
**Example 5-6. Indirect With Predisplacement Subtract and Modify**

The address of the operand to be fetched is the contents of an auxiliary register (ARn) minus the displacement (disp). The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1. After the data is fetched, the auxiliary register is updated with the address generated.

**Operation:** operand address =  $ARn - disp$   
 $ARn = ARn - disp$

**Assembler Syntax:** \*-- ARn(disp)

**Modification Field:** 00011



5

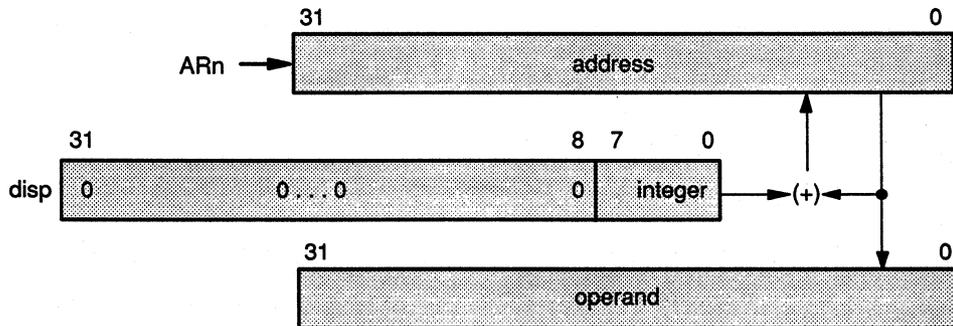
**Example 5-7. Indirect With Postdisplacement Add and Modify**

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the displacement (disp) is added to the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

**Operation:** operand address =  $ARn$   
 $ARn = ARn + disp$

**Assembler Syntax:** \*ARn++ (disp)

**Modification Field:** 00100



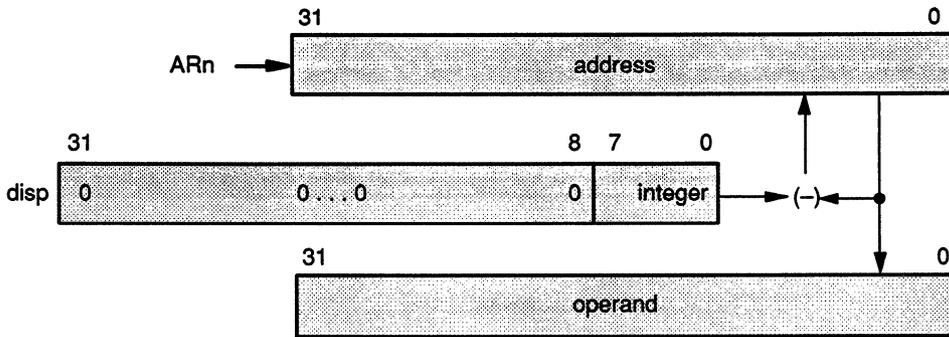
**Example 5-8. Indirect With Postdisplacement Subtract and Modify**

The address of the operand to be fetched is the contents of an auxiliary register (AR $n$ ). After the operand is fetched, the displacement (disp) is subtracted from the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

**Operation:** operand address = AR $n$   
 AR $n$  = AR $n$  - disp

**Assembler Syntax:** \*AR $n$  -- (disp)

**Modification Field:** 00101



5

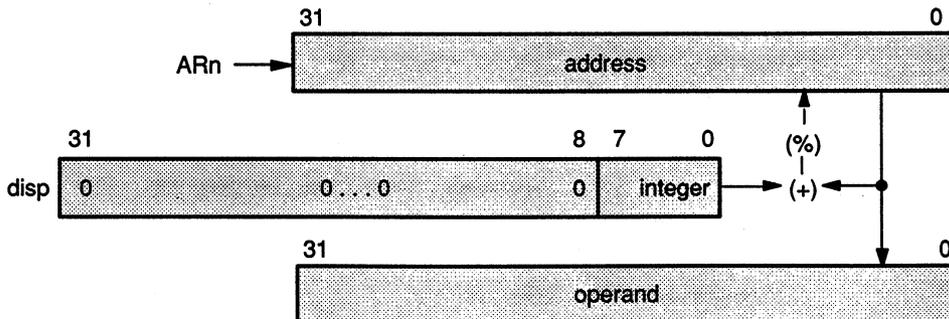
**Example 5-9. Indirect With Postdisplacement Add and Circular Modify**

The address of the operand to be fetched is the contents of an auxiliary register (AR $n$ ). After the operand is fetched, the displacement (disp) is added to the contents of the auxiliary register using circular addressing. This result is used to update the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

**Operation:** operand address = AR $n$   
 AR $n$  = circ(AR $n$  + disp)

**Assembler Syntax:** \*AR $n$  ++ (disp)%

**Modification Field:** 00110



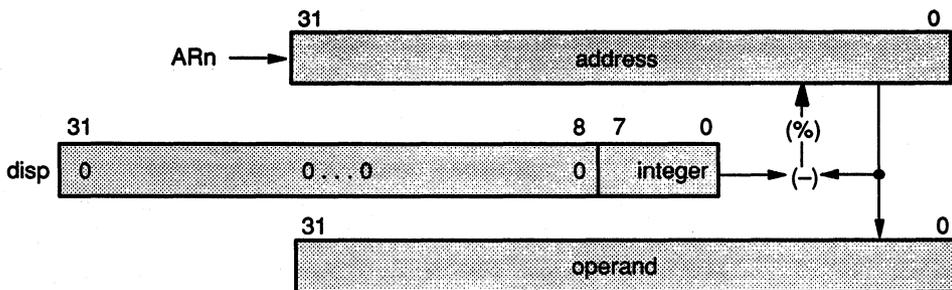
**Example 5-10. Indirect With Postdisplacement Subtract and Circular Modify**

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the displacement (disp) is subtracted from the contents of the auxiliary register through circular addressing. This result is used to update the auxiliary register. The displacement is either an 8-bit unsigned integer contained in the instruction word or an implied value of 1.

**Operation:** operand address = ARn  
 $ARn = \text{circ}(ARn - \text{disp})$

**Assembler Syntax:** \*ARn--(disp)%

**Modification Field:** 00111

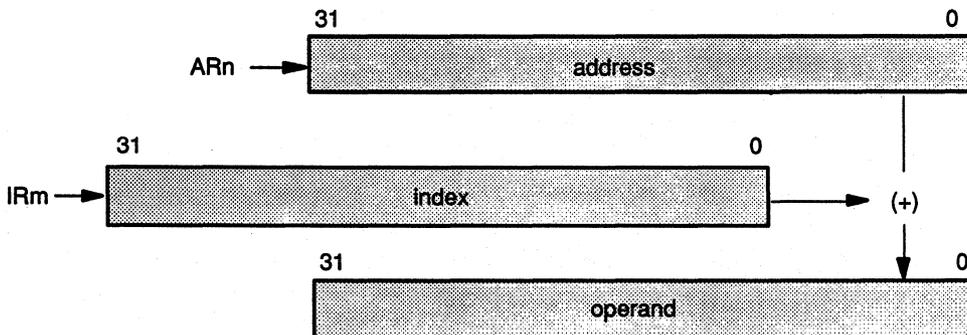
**Example 5-11. Indirect With Preindex Add**

The address of the operand to be fetched is the sum of an auxiliary register (ARn) and an index register (IR0 or IR1).

**Operation:** operand address =  $ARn + IRm$

**Assembler Syntax:** \*+ ARn(IRm)

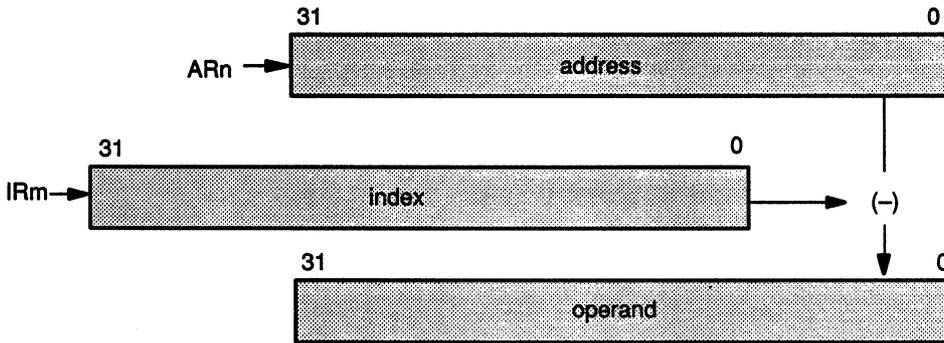
**Modification Field:** 01000 if  $m = 0$   
 10000 if  $m = 1$



**Example 5–12. Indirect With Preindex Subtract**

The address of the operand to be fetched is the difference between an auxiliary register ( $AR_n$ ) and an index register ( $IR_0$  or  $IR_1$ ).

<b>Operation:</b>	$\text{operand address} = AR_n - IR_m$
<b>Assembler Syntax:</b>	$*- AR_n(IR_m)$
<b>Modification Field:</b>	01001 if $m = 0$ 10001 if $m = 1$

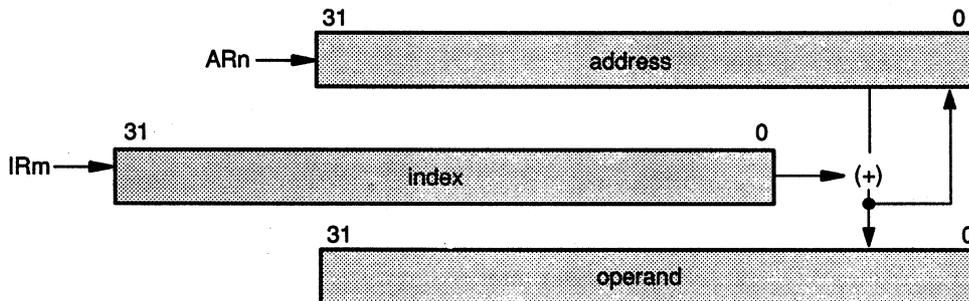


5

**Example 5–13. Indirect With Preindex Add and Modify**

The address of the operand to be fetched is the sum of an auxiliary register ( $AR_n$ ) and an index register ( $IR_0$  or  $IR_1$ ). After the data is fetched, the auxiliary register is updated with the address generated.

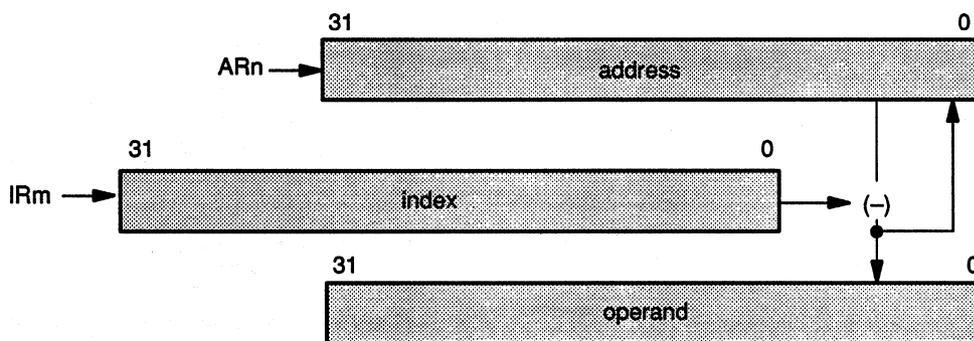
<b>Operation:</b>	$\text{operand address} = AR_n + IR_m$ $AR_n = AR_n + IR_m$
<b>Assembler syntax:</b>	$*++ AR_n(IR_m)$
<b>Modification Field:</b>	01010 if $m = 0$ 10010 if $m = 1$



**Example 5-14. Indirect With Preindex Subtract and Modify**

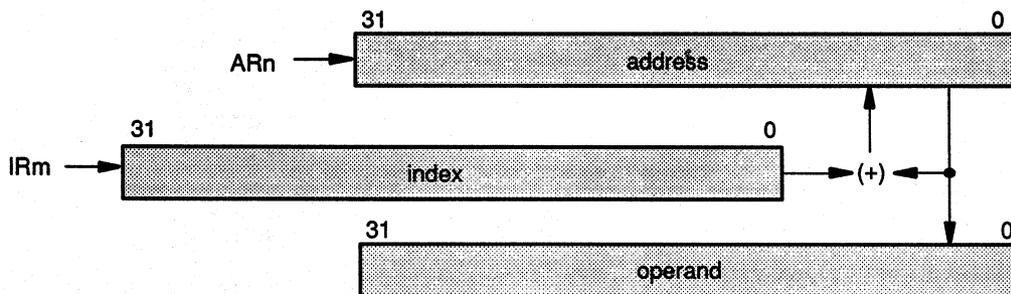
The address of the operand to be fetched is the difference between an auxiliary register (AR $n$ ) and an index register (IR0 or IR1). The resulting address becomes the new contents of the auxiliary register.

<b>Operation:</b>	operand address = AR $n$ - IR $m$ AR $n$ = AR $n$ - IR $m$
<b>Assembler Syntax:</b>	*-- AR $n$ (IR $m$ )
<b>Modification Field:</b>	01011 if $m = 0$ 10011 if $m = 1$

**Example 5-15. Indirect With Postindex Add and Modify**

The address of the operand to be fetched is the contents of an auxiliary register (AR $n$ ). After the operand is fetched, the index register (IR0 or IR1) is added to the auxiliary register.

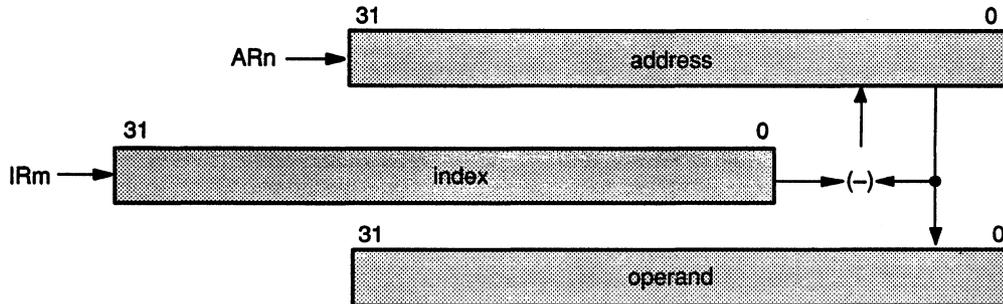
<b>Operation:</b>	operand address = AR $n$ AR $n$ = AR $n$ + IR $m$
<b>Assembler Syntax:</b>	*AR $n$ ++ (IR $m$ )
<b>Modification Field:</b>	01100 if $m = 0$ 10100 if $m = 1$



**Example 5–16. Indirect With PostIndex Subtract and Modify**

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is subtracted from the auxiliary register.

<b>Operation:</b>	operand address = ARn $ARn = ARn - IRm$
<b>Assembler Syntax:</b>	*ARn-- (IRm)
<b>Modification Field:</b>	01101 if m = 0 10101 if m = 1

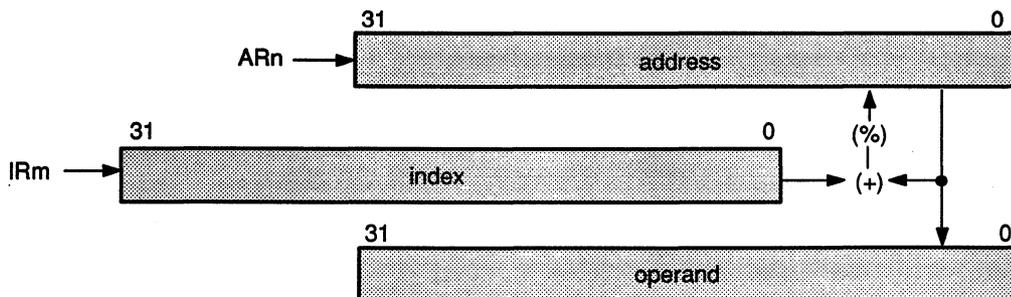


5

**Example 5–17. Indirect With Postindex Add and Circular Modify**

The address of the operand to be fetched is the contents of an auxiliary register (ARn). After the operand is fetched, the index register (IR0 or IR1) is added to the auxiliary register. This value is evaluated using circular addressing and replaces the contents of the auxiliary register.

<b>Operation:</b>	operand address = ARn $ARn = \text{circ}(ARn + IRm)$
<b>Assembler Syntax:</b>	*ARn++ (IRm)%
<b>Modification Field:</b>	01110 if m = 0 10110 if m = 1



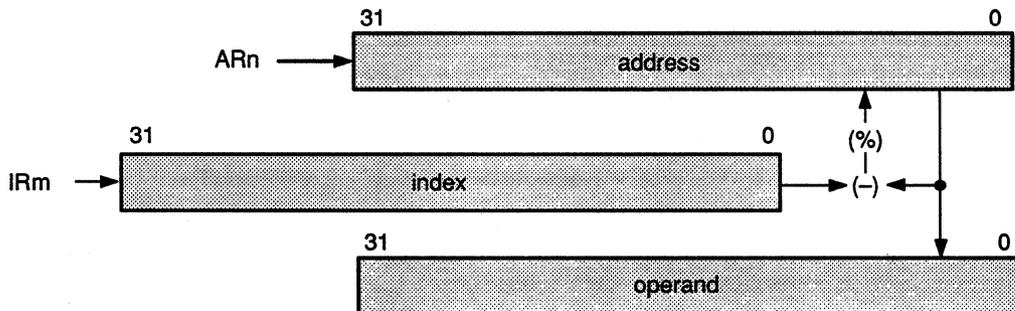
**Example 5-18. Indirect With Postindex Subtract and Circular Modify**

The address of the operand to be fetched is the contents of an auxiliary register ( $AR_n$ ). After the operand is fetched, the index register ( $IR_0$  or  $IR_1$ ) is subtracted from the auxiliary register. This result is evaluated through circular addressing and replaces the contents of the auxiliary register.

**Operation:** operand address =  $AR_n$   
 $AR_n = \text{circ}(AR_n - IR_m)$

**Assembler Syntax:**  $*AR_n--(IR_m)\%$

**Modification Field:** 01111 if  $m = 0$   
 10111 if  $m = 1$

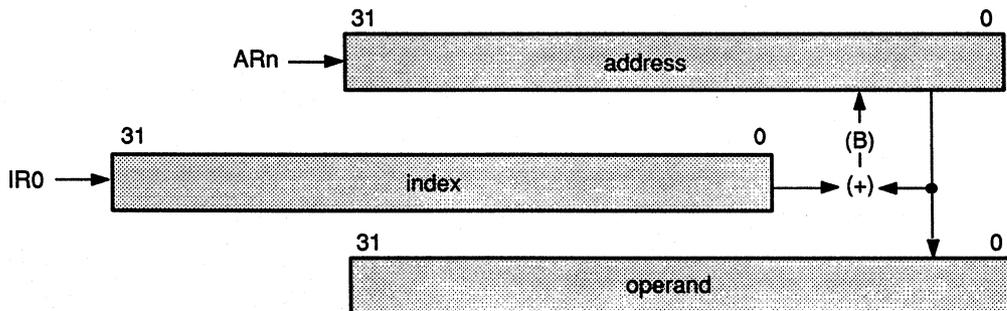
**Example 5-19. Indirect With Postindex Add and Bit-Reversed Modify**

The address of the operand to be fetched is the contents of an auxiliary register ( $AR_n$ ). After the operand is fetched, the index register ( $IR_0$ ) is added to the auxiliary register. This addition is performed with a reverse-carry propagation and can be used to yield a bit-reversed (B) address. This value replaces the contents of the auxiliary register.

**Operation:** operand address =  $AR_n$   
 $AR_n = B(AR_n + IR_0)$

**Assembler Syntax:**  $*AR_n++(IR_0)B$

**Modification Field:** 11001



## 5.1.4 Immediate Addressing

In immediate addressing, the operand is a 16-bit immediate value contained in the 16 least significant bits of the instruction word (*expr*). Depending upon the data types assumed for the instruction, the immediate operand may be a two's-complement integer, an unsigned integer, or a floating-point number. This is the syntax for this mode:

**Syntax:**                    *expr*

Example 5–20 gives an instruction example with before- and after-instruction data.

### Example 5–20. Immediate Addressing

Instruction	Before Instruction:	After Instruction:
SUBI 1, R0	R0 = 0h	R0 = 00 FFFF FFFFh
LDI 0FFFFh, R0	R0 = 0h	R0 = 00 FFFF FFFFh
LDF 5.0, R0	R0 = 0h	R0 = 02 2000 0000h
OR 0FFFFh, R0	R0 = 0h	R0 = 00 0000 FFFFh

5

## 5.1.5 PC-Relative Addressing

PC-relative addressing is used for branching. Instructions of this type include *Bcond*, *BcondD*, *BcondAF*, *BcondAT*, *DBcond* and *DBcondD* (repeat block), and *LAJ* (link and jump). It replaces the value of the PC with the contents of the 16 or 24 least significant bits of the instruction word. The assembler takes the *src* (a label or address) specified by the user and generates a displacement. If the branch is a standard branch, this displacement is equal to [*label* – (*PC* + 1)]. If the branch is a delayed branch, this displacement is equal to [*label* – (*PC* + 3)].

The displacement is stored as a 16-bit signed integer in the least significant bits of the instruction word.

**Syntax:**                    *expr*

Example 5–21 gives an instruction example with before- and after-instruction data.

### Example 5–21. PC-Relative Addressing

```
BU    NEWPC    ; pc=1,NEWPC= 5,displacement= 3
```

**Before Instruction:**

PC = 1h

**After Instruction:**

PC = 5h

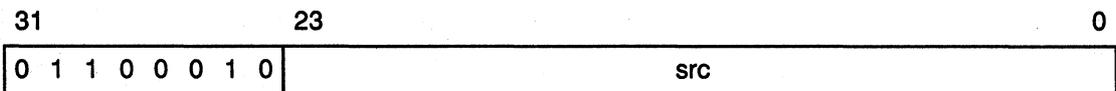
The 24-bit addressing mode is used to encode the program control instructions (e.g., BR, BRD, CALL, RPTB, RPTBD, LAJ). Depending on the instruction, the new PC value is derived by adding a 24-bit signed value in the instruction word with the present PC value. Bit 24 determines the type of branch (D=0 for a standard branch or D=1 for a delayed branch). Some of these instructions are encoded in Figure 5–2.

Figure 5–2. Encoding for 24-Bit PC-Relative Addressing Mode

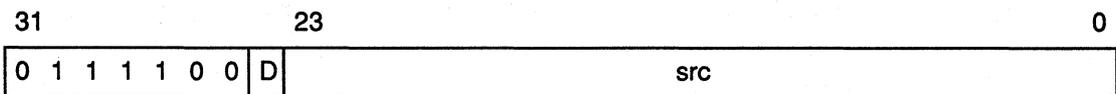
(a) BR, BRD: unconditional branches (delayed and not delayed)



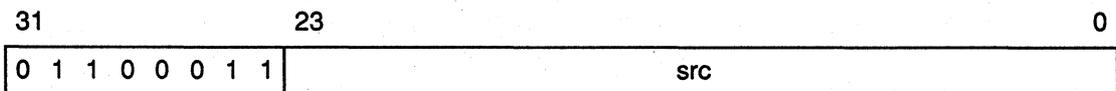
(b) CALL: unconditional subroutine call



(c) RPTB, RPRBD: repeat block (not delayed and delayed)



(d) LAJ: link and jump (return address in extended-precision register R11)



5

## 5.2 Groups of Addressing Modes

Six types of addressing (covered in Section 5.1, beginning on page 5-2) form these four groups of addressing modes:

	<b>Subsection</b>	<b>Page</b>
□ General addressing modes (G)	5.2.1	5-19
□ Three-operand addressing modes (T)	5.2.2	5-20
□ Parallel addressing modes (P)	5.2.3	5-23
□ Conditional-branch addressing modes (B)	5.2.4	5-24

### 5.2.1 General Addressing Modes

5

Instructions that use the general addressing modes are general-purpose instructions, such as ADDI, MPYF, and LSH. Such instructions usually have this form:

$$dst \text{ operation } src \rightarrow dst$$

where the destination operand is signified by *dst* and the source operand by *src*; operation defines an operation to be performed with the general addressing modes to specify certain operands. Bits 31 – 29 are zero, indicating general addressing mode instructions. Bits 22 and 21 specify the general addressing mode (G) field, which defines how bits 15 through 0 are to be interpreted for addressing the *src* operand.

Options for bits 22 and 21 (G field) are as follows:

0 0	register (all CPU registers unless specified otherwise)
0 1	direct
1 0	indirect
1 1	immediate

If the *src* and *dst* fields contain register specifications, the value in these fields contains the CPU register addresses as defined by Table 5–1. For the general addressing modes, the following values of AR<sub>n</sub> are valid for indirect addressing:

$$AR_n, 0 \leq n \leq 7$$

Figure 5–3 shows the encoding for the general addressing modes. The notation mod<sub>n</sub> indicates the modification field that goes with the AR<sub>n</sub> field. Refer to Table 5–2 for further information.

Figure 5-3. Encoding for General Addressing Modes

31	29 28	23 22	21 20	16 15	11 10	8 7	5 4	0
0 0 0	operation	0 0	dst	0 0 0 0 0 0 0 0 0 0 0	src			
0 0 0	operation	0 1	dst	direct				
0 0 0	operation		dst	modn	ARn	disp		
0 0 0	operation	0 1	dst	immediate				
		G	Destination	Source Operands				

### 5.2.2 Three-Operand Addressing Modes

The 19 three-operand instructions on the 'C40 use the eight address forms listed in Table 5-3:

5

Table 5-3. Three-Operand Instruction Address Forms

Type 1†

T	src1 addressing modes	src2 addressing modes	dst‡
00	register mode (any CPU register)	register mode (any CPU register)	Rx
01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)	Rx
10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)	Rx
11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)	Rx

Type 2†

T	src1 addressing modes	src2 addressing modes	dst‡
00	register mode (any CPU register)	8-bit signed immediate	Rx
01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)	Rx
10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate	Rx
11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)	Rx

† The 'C40 recognizes either type 1 or type 2 instructions; the 'C30 recognizes only type 1.

‡ Rx = any register in the CPU (primary) register file for the respective processor.

The object values differ for three-operand instructions, depending on the assembler used:

- ❑ the TMS320C3x assembler recognizes only type 1 formats and sets bits 31–28 to **0010<sub>2</sub>**.
- ❑ the TMS320C4x assembler recognizes both types and sets bits 31–28 to **0010<sub>2</sub>** for type 1 and to **0011<sub>2</sub>** for type 2.

The 'C4x processor executes both types (1 and 2). The 'C30 executes only the type 1 format. The three-operand instructions MPYSHI3 and MPYUHI3 are unique to the 'C40.

All instructions **except four** can use all four of the type 2 address forms shown in Table 5–3. These exceptions, which can use only address forms 2 and 4 in type 2, are the floating-point instructions ADDF3, CMPF3, MPYF3, and SUBF3.

The remaining 15 three-operand instructions are ADDC3, ADDI3, AND3, ANDN3, ASH3, CMPI3, LSH3, MPYI3, MPYSHI3, MPYUHI3, OR3, SUBB3, SUBI3, TSTB3, and XOR3.

Note that **the 3 can be omitted** from a three-operand instruction mnemonic.

Bits 22 and 21 specify the three-operand addressing mode (T) field, which defines how bits 15 – 0 are to be interpreted for addressing the *src* operands. Bits 15 – 8 define the *src1* address, and bits 7– 0 define the *src2* address.

5

Figure 5–4 and Figure 5–5 show the encoding for 'C4x three-operand addressing (the 'C30 recognizes only the format in Figure 5–4). The notation **modm** or **modn** indicate that the modification field goes with the ARm or ARn field, respectively. Refer to Table 5–2 (page 5-6) for further information.

The 8-bit signed immediate value supports left shifts, right shifts, and memory increment and decrement operations. The immediate value is not available for floating-point operations.

These instructions greatly help reduce code size, both assembled and compiled. They also give noticeable performance improvements in DSP and other computationally intensive applications and general-purpose code.

Figure 5-4. Encoding for Type 1 Three-Operand Addressing Modes ('C30 and 'C40)

31	28	27	23	22	21	20	16	15	13	12	11	10	8	7	5	4	3	2	0
0	0	1	0	operation	0	0	dst	0	0	0	src1			0	0	0	src2		
0	0	1	0	operation	0	1	dst	modn			ARn		0	0	0	src2			
0	0	1	0	operation	1	0	dst	0	0	0	src1			modn		ARn			
0	0	1	0	operation	1	1	dst	modn			ARn		modm		ARm				
				T					src1				src2						

Figure 5-5. Encoding for Type 2 Three-Operand Addressing Modes ('C40 Only)

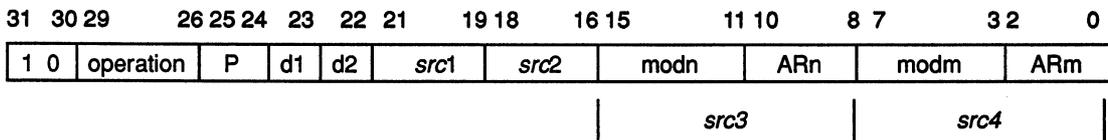
31	28	27	23	22	21	20	16	15	13	12	10	8	7	3			2	0
0	0	1	1	operation	0	0	dst	0	0	0	Rn		immediate					
0	0	1	1	operation	0	1	dst	0	0	0	Rn		disp		ARn			
0	0	1	1	operation	1	0	dst	disp			ARn		immediate					
0	0	1	1	operation	1	1	dst	disp			ARn		disp		ARm			
				T					src1				src2					

5

### 5.2.3 Parallel Addressing Modes

Instructions that use parallel addressing, indicated by || (two vertical bars), allow for the greatest amount of parallelism possible. The destination operands are indicated as *d1* and *d2*, signifying *dst1* and *dst2*, respectively (see Figure 6–4). The source operands, signified by *src1* and *src2*, use the extended-precision registers. The parallel operation to be performed is called operation.

Figure 5–6. Encoding for Parallel Addressing Modes



The parallel addressing mode (P) field specifies how the operands are to be used, i.e., whether they are source or destination. The specific relationship between the P field and the operands is detailed in the description of the individual parallel instructions (see Chapter 11). However, the operands are always encoded in the same way. Bits 31 and 30 are set to the value of 10, indicating parallel addressing mode instructions. Bits 25 and 24 specify the parallel addressing mode (P) field, which defines how bits 21 – 0 are to be interpreted for addressing the *src* operands. Bits 21 – 19 are used to define the *src1* address, bits 18 – 16 to define the *src2* address, bits 15 – 8 the *src3* address, and bits 7 – 0 the *src4* address. The notations *modn* and *modm* indicate which modification field goes with which ARn or ARm (auxiliary register) field, respectively. The parallel addressing operands are listed below.

- src1* = *Rn* (0 ≤ *n* ≤ 7 for extended-precision registers R0 – R7)
- src2* = *Rn* (0 ≤ *n* ≤ 7 for extended-precision registers R0 – R7)
- d1* If 0, *dst1* is R0. If 1, *dst1* is R1.
- d2* If 0, *dst2* is R2. If 1, *dst2* is R3.
- P 0 ≤ P ≤ 3
- src3* indirect (disp = 0, 1, IR0, IR1)
- src4* indirect (disp = 0, 1, IR0, IR1)

As in the three-operand addressing mode, indirect addressing in the parallel addressing mode allows for displacements of 0 or 1 and the use of the index registers (IR0 and IR1). The displacement of 1 is implied and is not explicitly coded in the instruction word.

In the encoding shown for this mode in Figure 5–6, if the *src3* and *src4* fields use the same auxiliary register, both addresses are correctly generated, but



### 5.3 Circular Addressing

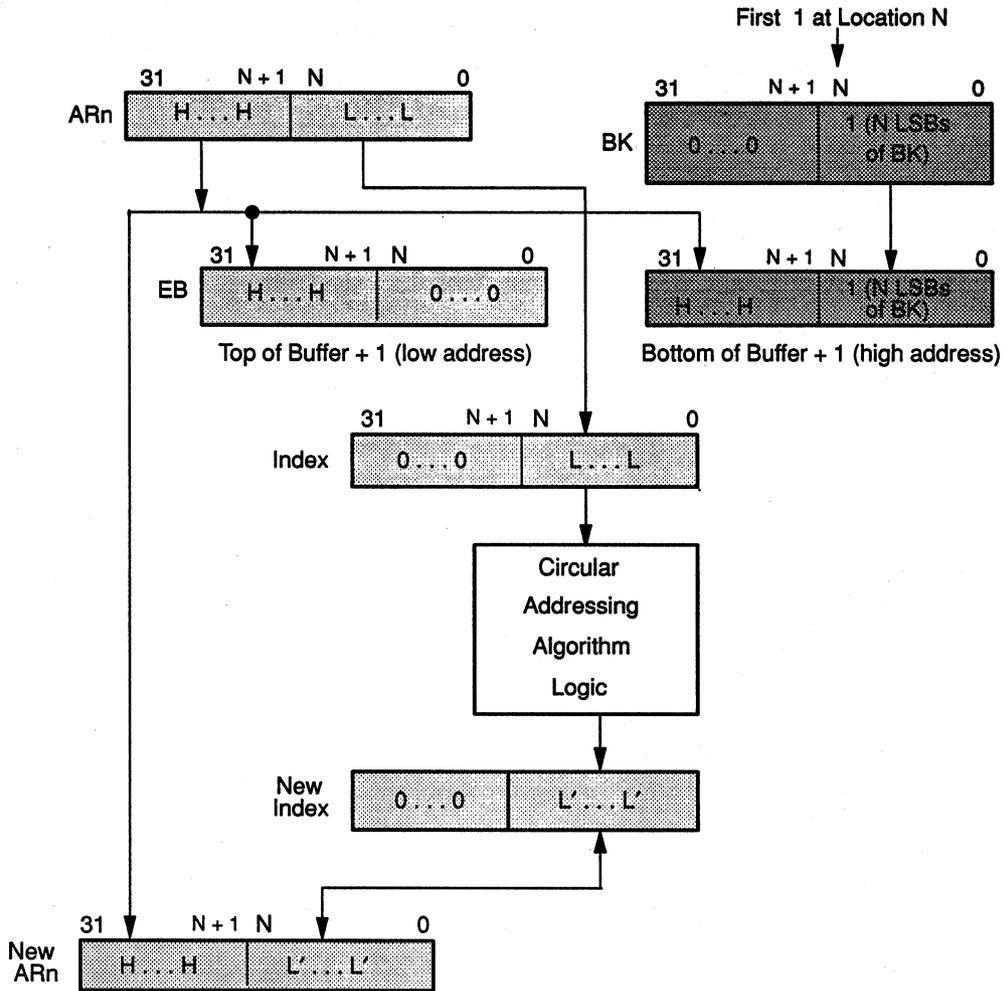
Many algorithms, such as convolution and correlation, require the implementation of a circular buffer in memory. In convolution and correlation, the circular buffer is used to implement a sliding window that contains the most recent data to be processed. As new data is brought in, the new data overwrites the oldest data. The key to the implementation of a circular buffer is the implementation of a circular addressing mode. This section describes the circular addressing mode of the TMS320C40.

The block-size register (BK) specifies the size of the circular buffer. The bottom of the circular buffer is specified by the first 1 (one) bit (counting from the most significant bit to the least significant bit) in the lower 16 bits of the BK register, plus a user-selected auxiliary register (ARn). With the location of the first 1 bit specified as bit N, the address at the top of the buffer is referred to as the effective base (EB) and is equal to bits 31 through (N+1) of ARn with bits N through 0 of EB being zero.

5

Figure 5–8 illustrates the relationships among the block-size register (BK), the auxiliary registers (ARn), the bottom of the circular buffer, the top of the circular buffer, and the index into the circular buffer.

Figure 5-8. Flowchart for Circular Addressing



LEGEND:

- |        |                        |      |                         |
|--------|------------------------|------|-------------------------|
| $AR_n$ | = auxiliary register n | $L$  | = low-order bits        |
| $BK$   | = block-size register  | $L'$ | = new low-order bits    |
| $EB$   | = effective base       | LSB  | = least significant bit |
| $H$    | = high-order bits      | $N$  | = bit value             |

5

In circular addressing, index refers to the N LSBs of the auxiliary register selected, and step is the quantity being added to or subtracted from the auxiliary register. Follow these two rules when you use circular addressing:

- ❑ The step used must be less than or equal to the block-size.
- ❑ The first time the circular queue is addressed, the auxiliary register must be pointing to an element in the circular queue.

The algorithm for circular addressing is as follows:

If  $0 \leq \text{index} + \text{step} < \text{BK}$ :

$\text{index} = \text{index} + \text{step}$ .

Else if  $\text{index} + \text{step} \geq \text{BK}$ :

$\text{index} = \text{index} + \text{step} - \text{BK}$ .

Else if  $\text{index} + \text{step} < 0$ :

$\text{index} = \text{index} + \text{step} + \text{BK}$ .

5

Figure 5–9 shows how the circular buffer is implemented. It illustrates the relationship of the quantities generated and the elements in the circular buffer.

Figure 5–9. Circular Buffer Implementation

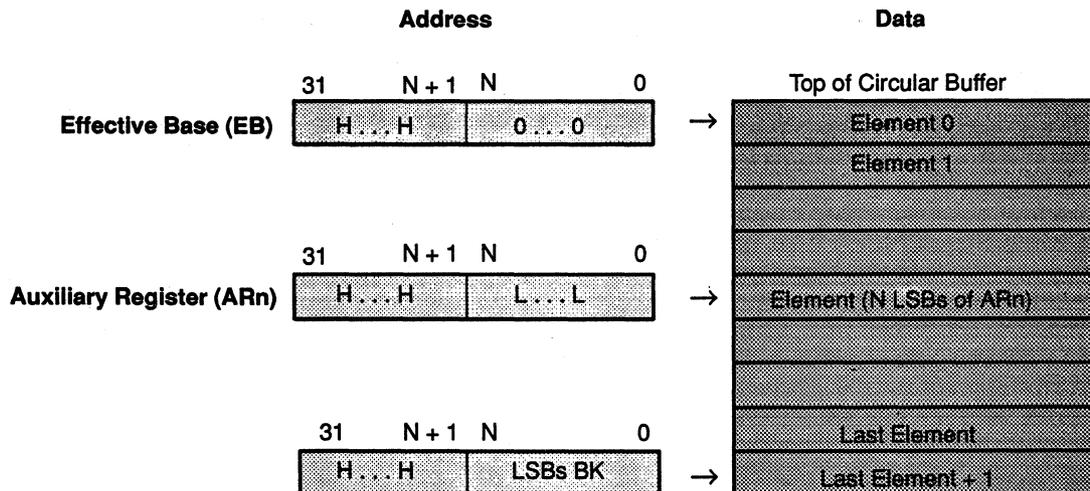


Figure 5–10 gives an example of the operation of circular addressing. Assuming that all ARs are four bits, let  $AR0 = 0000_2$ , and  $BK = 0110_2$  (block-size of 6). This example shows a sequence of modifications and the resulting value of AR0. It also shows how the pointer steps through the circular queue with a variety of step sizes (both incrementally and decrementally).

Figure 5–10. Circular Addressing Example

```

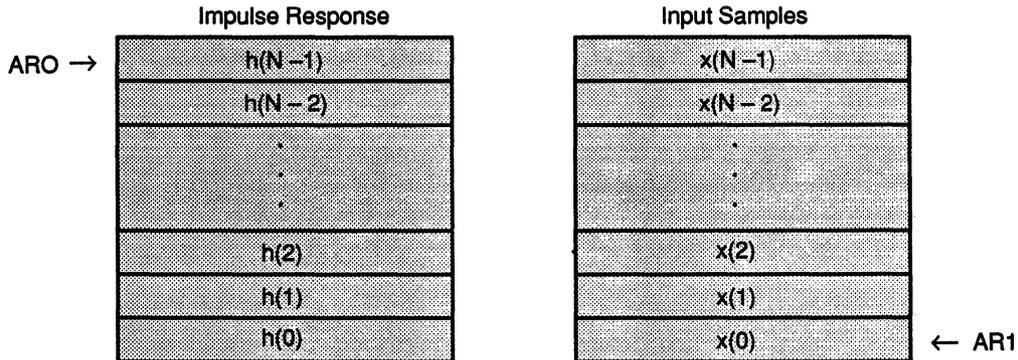
*AR0 ++ (5)%      ; AR0 = 0   (0th value)
*AR0 ++ (2)%      ; AR0 = 5   (1st value)
*AR0 -- (3)%      ; AR0 = 1   (2nd value)
*AR0++(6)%        ; AR0 = 4   (3rd value)
*AR0 -- --%       ; AR0 = 4   (4th value)
*AR0               ; AR0 = 3   (5th value)
    
```

5

Value	Data	Address
0th →	Element 0	0
2nd →	Element 1	1
	Element 2	2
5th →	Element 3	3
4th, 3rd →	Element 4	4
1st →	Element 5 (Last Element)	5
	Last Element + 1	6

Circular addressing is especially useful for the implementation of FIR filters. Figure 5–11 shows one possible data structure for FIR filters. Note that the initial value of AR0 points to  $h(N-1)$ , and the initial value of AR1 points to  $x(0)$ . Circular addressing is used in the TMS320C40 code for the FIR filter shown in Figure 5–12.

Figure 5–11. Data Structure for FIR Filters



5

Figure 5–12. FIR Filter Code Using Circular Addressing

```

* Initialization
*
    LDI    N,BK           ; Load block size.
    LDI    H,ARO         ; Load pointer to impulse response.
    LDI    X,AR1        ; Load pointer to bottom of input
                        ; sample buffer.
*
*
TOP  LDF    IN, R3       ; Read input sample.
     STF    R3,*AR1++%  ; Store with other samples.
                        ; and point to top of buffer.

     LDF    0,R0        ; Initialize R0.
     LDF    0,R2        ; Initialize R2.
*
*   Filter
*
     RPTS   N - 1      ; Repeat next instruction.
     MPYF3  *ARO++%,*AR1++%,R0
||   ADDF3  R0,R2,R2   ; Multiply and accumulate.
     ADDF   R0,R2      ; Last product accumulated.
*
     STF    R2,Y       ; Save result.
     B     TOP        ; Repeat.

```

## 5.4 Bit-Reversed Addressing

Bit-reversed addressing on the TMS320C40 enhances execution speed and program memory for FFT algorithms that use a variety of radices. One auxiliary register points to the physical location of a data value. IR0 specifies one-half the size of the FFT; e.g., the value contained in IR0 must be equal to  $2^{n-1}$  where  $n$  is an integer and the FFT size is  $2^n$ . When you add IR0 to the auxiliary register by using bit-reversed addressing, addresses are generated in a bit-reversed fashion. The largest index for bit reversed is 00FF FFFFh.

To illustrate this kind of addressing, assume 8-bit auxiliary registers. Let AR2 contain the value 0110 0000<sub>2</sub> (96<sub>10</sub>). This is the base address of the data in memory. Let IR0 contain the value 0000 1000<sub>2</sub> (8). Figure 5-13 shows a sequence of modifications of AR2 and the resulting values of AR2.

**5** Figure 5-13. Bit-Reversed Addressing Example

```

*AR2++(IR0)B      ; AR2 = 0110 0000 (0th value)
*AR2++(IR0)B      ; AR2 = 0110 1000 (1st value)
*AR2++(IR0)B      ; AR2 = 0110 0100 (2nd value)
*AR2++(IR0)B      ; AR2 = 0110 1100 (3rd value)
*AR2++(IR0)B      ; AR2 = 0110 0010 (4th value)
*AR2++(IR0)B      ; AR2 = 0110 1010 (5th value)
*AR2++(IR0)B      ; AR2 = 0110 0110 (6th value)
*AR2              ; AR2 = 0110 1110 (7th value)
    
```

Table 5-4 shows the relationship of the index steps and the four LSBs of AR2. As you can see, you can find the four LSBs by reversing the bit pattern of the steps.

Table 5-4. Index Steps and Bit-Reversed Addressing

Step	Bit Pattern	Bit-Reversed Pattern	Bit-Reversed Step
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

## 5.5 System and User Stack Management

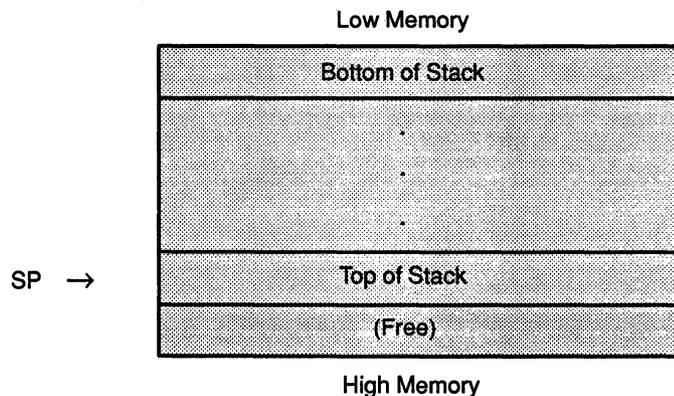
The TMS320C40 provides a dedicated system stack pointer (SP) for building stacks in memory. The auxiliary registers can also be used to build a variety of more general linear lists. This section discusses the implementation of the following types of linear lists:

- Stack** A linear list for which all insertions and deletions are made at one end of the list.
- Queue** A linear list for which all insertions are made at one end of the list, and all deletions are made at the other end.
- Deque** A double-ended queue linear list for which insertions and deletions are made at either end of the list.

The system stack pointer (SP) is a 32-bit register that contains the address of the top of the system stack. The system stack fills from low-memory address to high-memory address (see Figure 5–14). The SP always points to the last element pushed onto the stack. A push performs a preincrement, and a pop performs a postdecrement of the system stack pointer.

The program counter is pushed onto the system stack on subroutine calls, traps, and interrupts. It is popped from the system stack on returns. The system stack can be pushed and popped with the PUSH, POP, PUSHF, and POPF instructions.

Figure 5–14. System Stack Configuration



### 5.5.1 Stacks

Stacks can be built from low to high memory or high to low memory. Two cases for each type of stack are shown. You can build stacks by using the preincrement/decrement and postincrement/decrement modes of modifying the auxiliary registers (AR). You can implement stack growth from high to low memory in two ways:

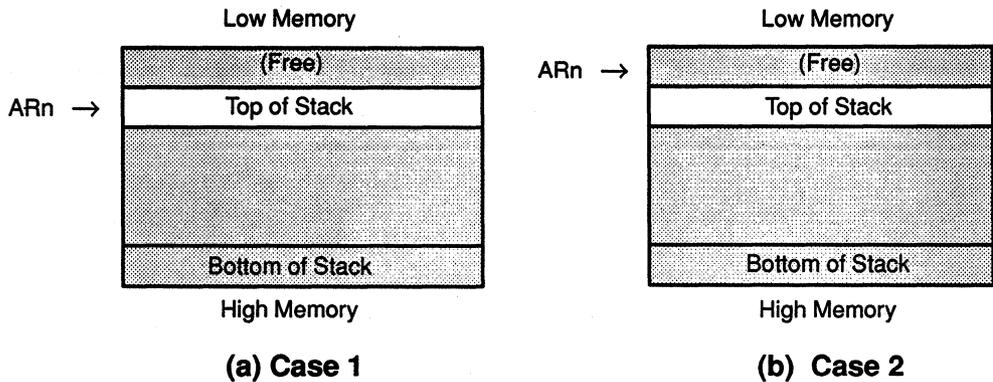
**Case 1:** Store to memory using  $*--ARn$  to push data onto the stack and reads from memory using  $*ARn++$  to pop data off the stack.

**Case 2:** Store to memory using  $*ARn--$  to push data onto the stack and read from memory using  $*++ARn$  to pop data off the stack.

Figure 5–15 illustrates these two cases. The only difference is that in case 1, the AR always points to the top of the stack, and in case 2, the AR always points to the next free location on the stack.

5

Figure 5–15. Implementations of High-to-Low Memory Stacks



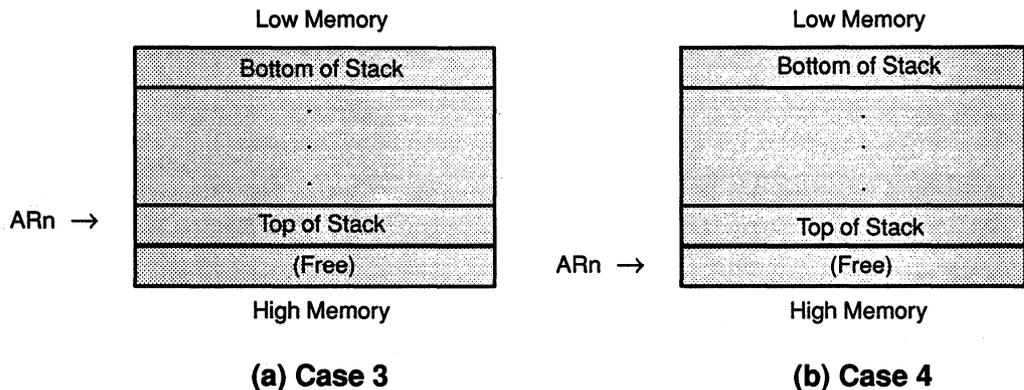
You can implement stack growth from low to high memory in two ways:

**Case 3:** Store to memory using  $*++ ARn$  to push data onto the stack and reads from memory using  $*ARn --$  to pop data off the stack.

**Case 4:** Stores to memory using  $*ARn ++$  to push data onto the stack and reads from memory using  $*-- ARn$  to pop data off the stack.

Figure 5–16 shows these two cases. In the case 3, the AR always points to the top of the stack. In case 4, the AR always points to the next free location on the stack.

Figure 5–16. Implementations of Low-to-High Memory Stacks



## 5.5.2 Queues and Dequeues

The implementations of queues and dequeues is based upon the manipulation of the auxiliary registers for user stacks. For queues, two auxiliary registers are used: one to mark the front of the queue from which data is popped and the other to mark the rear of the queue where data is pushed.

For dequeues, two auxiliary registers are also necessary. One is used to mark one end of the dequeue, and the other is used to mark the other end. Data can be popped or pushed from either end.



## Program Flow Control

---

---

---

The TMS320C40 provides a complete set of constructs that facilitate software and hardware control of the program flow. Software control includes repeats, branches, calls, traps, and returns. Hardware control includes reset and interrupts. Because programming includes a variety of constructs, you can select the one suited for your particular application.

Several interlocked operations instructions provide flexible multiprocessor support and, through the use of external signals, a powerful means of synchronization. They also guarantee the integrity of the communication and result in a high-speed operation.

6

The TMS320C40 supports a nonmaskable external reset signal and a number of internal and external interrupts. These functions can be programmed for a particular application.

This chapter discusses the following major topics:

<b>Section</b>	<b>Page</b>
6.1 Repeat Modes .....	6-2
■ Initialization .....	6-2/6-4
■ Operation .....	6-4
6.2 Delayed Branches .....	6-7
6.3 Calls, Traps, Branches, Jumps, and Returns .....	6-9
6.4 Unifying Traps and Interrupts .....	6-11
6.5 Interlocked Operations .....	6-13
6.6 Reset Operation .....	6-18
6.7 Interrupts .....	6-23
■ Interrupt Control Bits .....	6-24
■ Prioritization and Control .....	6-24

## 6.1 Repeat Modes

The repeat modes of the TMS320C40 can implement zero-overhead looping. For many algorithms, most execution time is spent in an inner kernel of code. Using the repeat modes allows these time-critical sections of code to be executed in the shortest possible time.

The TMS320C40 provides three instructions to support zero-overhead looping: RPTB, RPTBD (repeat a block of code/delayed) and RPTS (repeat a single instruction):

- ❑ RPTB and RPTBD causes a block of code to be repeated a specified number of times, and
- ❑ RPTS causes a single instruction to be repeated a number of times and reduces the bus traffic by fetching the instruction only once.

Three registers (RS, RE, and RC) are associated with the updating of the program counter when it is updated in a repeat mode, as described in Table 6–1 below.

**6** Table 6–1. Repeat-Mode Registers

Register	Function
RS	Repeat start address register. Holds the address of the first instruction of the block of code to be repeated.
RE	Repeat end address register. Holds the address of the last instruction of the block of code to be repeated.
RC	Repeat-count register. Contains one less than the number of times the block remains to be repeated.

### 6.1.1 Repeat-Mode Initialization

Two bits are important to the operation of RPTB, RPTBD and RPTS: the RM and S bits.

- ❑ The **RM** (repeat-mode flag) bit in the status register specifies whether the processor is running in the repeat mode.
  - If RM = 0, fetches are not made in repeat mode.
  - If RM = 1, fetches are made in repeat mode.
- ❑ The **S bit** is internal to the processor and cannot be programmed, but this bit is necessary to fully describe the operation of RPTB and RPTS.
  - If RM = 1 and S = 0, RPTB or RPTBD is executing. Program fetches are from memory.
  - If RM = 1 and S = 1, RPTS is executing. After the first fetch (from memory), program fetches are from the instruction register (IR).

The correct operation of the repeat modes requires that all of the above registers and status register fields be initialized correctly. The RPTB, RPTBD, and RPTS instructions perform this initialization in slightly different ways (see subsections 6.1.2 and 6.1.3).

### 6.1.2 RPTB and RPTBD Initialization

The execution sequence of RPTB *src* or RPTBD *src* is nearly the same:

- 1) Loads the start address of the block into RS (repeat start address register).
  - a) For **RPTB**, this is the next address following the instruction:  

$$\text{PC of RPTB} + 1 \rightarrow \text{RS}$$

**or**
  - b) For **RPTBD**, this is the fourth address following the instruction:  

$$\text{PC of RPTBD} + 4 \rightarrow \text{RS}$$
- 2) Loads the end address of the block into RE (repeat end address register).
  - a) In *PC-relative mode*, the 24-bit *src* operand plus RS is the end address:  
 For **RPTB**,  

$$\text{src} + \text{PC of RPTB} + 1 \rightarrow \text{RE}$$

**or**

 For **RPTBD**,  

$$\text{src} + \text{PC of RPTBD} + 3 \rightarrow \text{RE}$$
  - b) In *register mode*, the contents of the *src* register is the end address:  

$$\text{contents of } \textit{src} \text{ register} \rightarrow \text{RE}$$
- 3) Sets the status register to indicate the repeat mode of operation.  

$$1 \rightarrow \text{RM status register bit (repeat mode flag)}$$
- 4) Indicates that this is the repeat block mode of operation.  

$$0 \rightarrow \text{S bit (bit is internal to processor; not programmable)}$$

The last bit of information required is the number of times to repeat the block. The value is determined by properly initializing the RC (repeat count) register. Because the execution of RPTB and RPTBD does not load the RC, you must load this register yourself. A typical setup of the block repeat operation is shown below.

```
LDI    15, RC ; 15 → RC
RPTB   LOOP ; LOOP → RE, PC + 1 → RS, 1 → RM, 0 → S
```

The repeat modes repeat a block of code at least once in a typical operation. The repeat counter should be loaded with one less than the number of times to execute the block; i.e., an RC value of 0 executes the block of code one time, or an RC value of 4 would execute the block five times. All block repeats initiated by RPTB or RPTBD can be interrupted.

### 6.1.3 RPTS Initialization

When RPTS *src* is executed, the following sequence of operations occurs:

- 1) PC + 1 → RS
- 2) PC + 1 → RE
- 3) 1 → RM status register bit
- 4) 1 → S bit
- 5) *src* → RC (repeat count register)

The RPTS instruction loads all registers and mode bits necessary for the operation of the single instruction repeat mode. Step 1 loads the start address of the block into RS. Step 2 loads the end address into the RE (end address of the block). Since this is a repeat of a single instruction, the start address and the end address are the same. Step 3 sets the status register to indicate the repeat mode of operation. Step 4 indicates that this is the repeat single-instruction mode of operation. Step 5 loads *src* into RC.

Repeats of a single instruction initiated by RPTS are not interruptible, because the RPTS fetches the instruction word only once and then keeps it in the instruction register for reuse. An interrupt would cause the instruction word to be lost. The refetching of the instruction word from the instruction register reduces memory accesses and, in effect, acts as a one-word program cache. If it is necessary to have a single instruction that is repeatable and interruptible, you can use the RPTB instruction.

6

### 6.1.4 Repeat-Mode Operation

Information in the repeat-mode registers and associated control bits is used to control the modification of the PC when the fetches are being made in repeat mode. The repeat modes compare the contents of the RE register (repeat end address register) with the program counter (PC). If they match and the repeat counter is nonnegative, the repeat counter is decremented, the PC is loaded with the repeat start address, and the processing continues. The fetches and appropriate status bits are modified as necessary. Note that the repeat counter (RC) is never modified when the repeat-mode flag (RM) is 0. The maximum number of repeats occurs when RC = 0 8000 0001h. This will result in 0 8000 0001h repetitions. The detailed algorithm for the update of the PC is shown in Figure 6–1.

The RPTB and RPTS are four-cycle instructions. These four cycles of overhead are incurred only on the first pass through the loop. All subsequent passes through the loop are accomplished with zero cycles of overhead. In Example 6–1, the block of code from STLOOP to ENDL0P is repeated sixteen times.

## Example 6-1. RPTB Operation

```

                LD    15,RC    ; Load repeat counter with 15
                RPTB  ENDLOP   ; Execute the block of code
STLOOP         ; from STLOOP to ENDLOP 16 times
                .
                .
                .
                ENDLOP

```

Figure 6-1. Repeat-Mode Control Algorithm

```

if RM == 1      ; If in repeat mode (RPTB or RPTS)
if S == 1      ; If RPTS
  if first time through ; If this is the first fetch
    fetch instruction from memory ; Fetch instruction from memory
  else          ; If not the first fetch
    fetch instruction from IR      ; Fetch instruction from IR
RC - 1 → RC    ; Decrement RC
if RC < 0      ; If RC is negative
              ; Repeat single mode completed
  0 → ST(RM)   ; Turn off repeat mode bit
  0 → S        ; Clear S
  PC + 1 → PC  ; Increment PC
else if S == 0 ; If RPTB
  fetch instruction from memory ; Fetch instruction from memory
if PC == RE    ; If this is the end of the block
  RC - 1 → RC  ; Decrement RC
if RC ≥ 0     ; If RC is not negative
  RS → PC     ; Set PC to start of block
else if RC < 0 ; If RC is negative
  0 → ST(RM)   ; Turn off repeat mode bits
  0 → S        ; Clear S
  PC + 1 → PC  ; Increment PC

```

Using the repeat block mode of modifying the PC facilitates analysis of what would happen in the case of branches within the block. Assume that the next value of the PC will be either PC + 1 or the contents of the RS register. It is thus apparent that this method of block repeat allows branching within the repeated block. Execution can go anywhere within the user's code via interrupts, subroutine calls, etc. For proper modification of the loop counter, the last instruction of the loop must be fetched. **By writing a 0 into the repeat counter or writing 0 into the RM bit of the status register, you can stop the repeating of the loop prior to completion.**

Since the block repeat modes modify the program counter, other instructions cannot modify the program counter at the same time. **Two rules** apply:

**Rule 1:** The last instruction in the block (or the only instruction in a block of size one) cannot be a *Bcond*, *DBcond*, *CALL*, *CALLcond*, *TRAPcond*, *RETIcond*, *RETScond*, *IDLE*, *RPTB*, or *RPTS*. Example 6–2 shows an incorrectly placed standard branch.

**Rule 2:** None of the last four instructions from the bottom of the block (or the only instruction in a block of size one) can be a *Bcond D*, *BRD*, or *DBcondD*, *RPTBD*, *LAJ*, *LAJcond*, *LATcond*, *BcondAF*, *BcondAT*, or *RETIcond*. Example 6–3 shows an incorrectly placed delayed branch.

**If either of these rules is violated, the PC will be undefined.**

#### Example 6–2. Incorrectly Placed Standard Branch

```

LD      15,RC      ; Load repeat counter with 15
RPTB   ENDLOP     ; Execute block of code
STLOOP ; from STLOOP to ENDLOP 16 times
.
.
.
ENDLOP BR    OOPS ; This branch violates rule 1

```

#### Example 6–3. Incorrectly Placed Delayed Branch

```

LD      15,RC      ; Load repeat counter with 15
RPTB   ENDLOP     ; Execute block of code
STLOOP ; from STLOOP to ENDLOP 16 times
.
.
.
BRD    OOPS       ; This branch violates rule 2
ADDF
MPYF
ENDLOP SUBF

```

Block repeats (*RPTB* and *RPTBD*) are nestable. Since all of the control is defined by the *RS*, *RE*, *RC*, and *ST* registers, these registers must be saved and stored in order to nest block repeats. The status register *RM* bit can be used to determine whether the block repeat mode is active. For example, if you write an interrupt service routine that requires the use of *RPTB* or *RPTBD*, it is possible that the interrupt associated with the routine may occur during another block repeat. The interrupt service routine can check the *RM* bit. If this bit is set, the interrupt routine saves *RS*, *RE*, *RC*, and *ST*. The interrupt routine can then perform a block repeat. Before returning to the interrupted routine, the interrupt routine restores *RS*, *RE*, *RC*, and *ST*. If the *RM* bit is not set, you don't need to save and restore these registers.

## 6.2 Delayed Branches

The TMS320C40 offers two main types of branching: standard and delayed. **Standard branches** empty the pipeline before performing the branch; this guarantees correct management of the program counter and results in a TMS320C40 branch taking four cycles. Included in this class are repeats, calls, returns, and traps.

**Delayed branches** without annulling do not empty the pipeline but, rather, guarantee that the next three instructions will execute before the program counter is modified by the branch. Delayed branches with annulling may conditionally annul the next three instructions. The result is a branch that requires only a single cycle, thus making the speed of the delayed branch very close to the optimal block repeat modes of the TMS320C40. However, unlike block repeat modes, delayed branches may be used in situations other than looping. Every delayed branch has a standard branch counterpart that is used when a delayed branch cannot be used. The delayed branches without annulling are *BcondD*, *BRD*, and *DBcondD*. Those with annulling are *BcondAT* and *BcondAF*.

Conditional delayed branches use the conditions, reflected in the status register, that existed at the end of the instruction preceding the branch. They do not depend upon the instructions following the delayed branch. Delayed branches without annulling guarantee that the next three instructions will execute, regardless of other pipeline conflicts.

When a delayed branch is fetched, it remains pending until the three instructions that follow are executed. **None of the three instructions immediately after a delayed branch can be any of the following** (see **Example 6-4**):

<i>Bcond</i>	<i>BRD</i>	<i>IDLE</i>	<i>RETIcondD</i>
<i>BcondD</i>	<i>DBcond</i>	<i>LAJ</i>	<i>RETScond</i>
<i>BcondAF</i> †	<i>DBcondD</i>	<i>LAJcond</i>	<i>RPTB</i>
<i>BcondAT</i> †	<i>CALL</i>	<i>LATcond</i>	<i>RPTBD</i>
<i>BR</i>	<i>CALLcond</i>	<i>RETIcond</i>	<i>RPTS</i>
			<i>TRAPcond</i>

† *BcondAF* and *BcondAT* are described in Section 6.3 on page 6-9.

**Incorrectly used delayed branches can leave the PC undefined.**

Delayed branches disable interrupts until the three instructions following the delayed branch are completed. This is independent of whether or not the branch is taken.

*Example 6-4. Incorrectly Placed Delayed Branches*

```

B1:    BD    L1
        NOP
        NOP
B2:    B     L2          ; This branch is incorrectly placed
        NOP
        NOP
        NOP
        .
        .
        .
    
```

**6**

- The **BcondAT** and **BcondAF** instructs both branch if conditions are **true**, but
  - **BcondAT** executes but annuls (cancels effect of — except for time delay) the execute phase of the next three instructions following **BcondAT**. Then it takes the branch. If *cond* is **false**, execution continues immediately after the **BcondAT**.
  - **BcondAF** first executes the next three instructions following the **BcondAF**. Then it takes the branch. If *cond* is **false**, execution continues immediately after the **BcondAF** but the execution phase of the first three instructions are annulled.

### 6.3 Calls, Traps, Branches, Jumps and Returns

Calls and traps provide a means of executing a subroutine or function while providing a return to the calling routine.

The **CALL**, **CALLcond**, and **TRAPcond** instructions store the value of the PC on the stack before changing the PC's contents. The **RETScond** or **RETIcond** (standard or delayed) instructions return execution from traps and calls using the value on the stack.

- ❑ **CALL** places the next PC value on the stack and places the *src* (source) operand into the PC. The *src* is a 24-bit PC-relative or register value. Figure 6–2 shows CALL response timing.
- ❑ **CALLcond** is similar to the CALL instruction (above) except that (1) it executes only if a specific condition is true (the 20 conditions — including unconditional — are listed in Section 11.2 on page 11-10 ) and (2) the *src* is either a PC-relative displacement or in register addressing mode.
- ❑ **TRAPcond** also executes only if a specific condition is true (same conditions as for the **CALLcond** instruction). When it executes, (1) interrupts are disabled with 0 written to bit GIE of the ST, (2) the next PC value is stored on the stack, and (3) a vector is retrieved from one of the addresses from 20h to 3Fh and loaded into the PC. The particular address corresponds to a trap number in the instruction. Using **RETIcond** or **RETIcondD** to return re-enables interrupts if the status register's GIE bit was set previously.
- ❑ **RETScond** returns execution from any of the above three instructions by popping the top of the stack to the PC. For **RETScond** to execute, the specified condition must be true. Conditions are the same as for the **CALLcond** instruction.
- ❑ **RETIcond** returns from traps or calls in the same way as the **RETScond** (above) does with the addition that **RETIcond** also copies the PGIE and PCF bit values into the GIE and CF bits of the status register. Conditions are the same as for the **CALLcond** instruction.
- ❑ **RETIcondD** returns from traps or calls the same way as the **RETIcond** (above) does with the addition that **RETIcondD** also *first executes* the next three instructions immediately following the **RETIcondD**. Conditions are the same as for the **CALLcond** instruction.
- ❑ Link and jump (**LJ**), link and jump conditional (**LJcond**), and link and trap conditional (**LATcond**) each provide a return address in extended-precision register R11.
  - After it executes the three instructions that follow it, **LJ** jumps to an address derived by the concatenation of the most significant 8 bits of the PC and the 24-bit *src* address in the instruction.

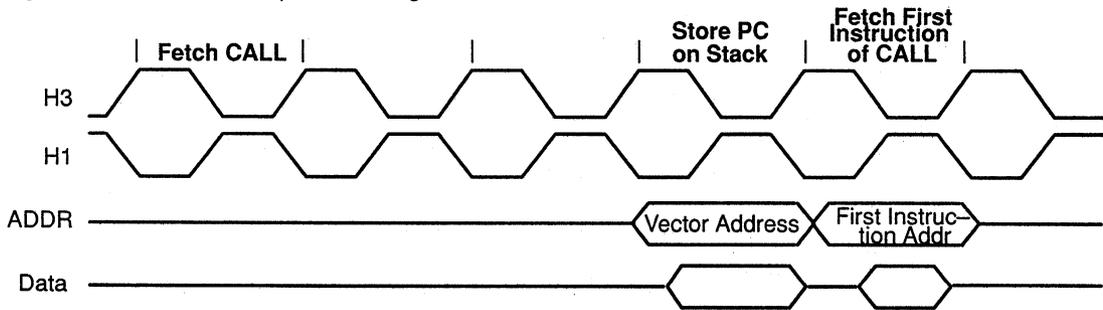
- **LAJcond** destination address is either PC-relative (a displacement) or the contents of a specified register. If the condition is true, LAJcond first executes the three instructions following the LAJcond before making the jump. If the condition is not true, execution continues immediately after the LAJcond instruction.
- After it executes the three instructions that follow it, **LATcond** calls one of the 512 available trap vectors pointed to by the trap vector table pointer (TVTP) in Section 3.2 on page 3-15. The vector value is loaded into the PC.

Functionally, calls and traps accomplish the same task: namely, a subfunction is called and executed, and control is then returned to the calling function. Traps offer several advantages:

- 1) Interrupts are automatically disabled when a trap is executed. This allows critical code to execute without risk of being interrupted. Thus, traps are usually terminated with a RETIcond or RETIcondD instruction to re-enable interrupts if the status register GIE bit was set previously.
- 2) You can use traps to indirectly call functions. This is particularly beneficial when a kernel of code contains the basic subfunctions to be used by applications. In this case, the functions in the kernel can be modified and relocated without recompiling each application.

6

Figure 6-2. CALL Response Timing



## 6.4 Unifying Traps and Interrupts

Traps and interrupts on the TMS320C4x are unified in all forms of operation *except initialization*.

### 6.4.1 Initialization

At *initialization*:

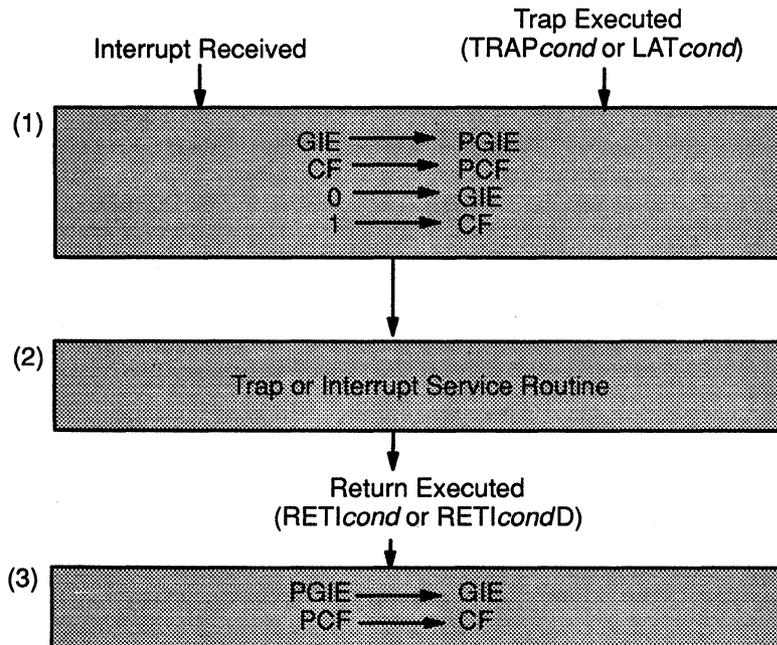
- ❑ **Traps** are always triggered by a software mechanism, either with *TRAPcond* (conditional trap) or *LATcond* (link and trap conditionally delayed).
- ❑ **Interrupts** are triggered by hardware events (i.e., external interrupts, DMA interrupts, or communication channel interrupts).

### 6.4.2 Operation

Figure 6–3 shows the unified flow of traps and interrupts.

For an **interrupt**, step (1) in the figure happens after completion of the last instruction that was fetched before completion of the interrupt flush. This guarantees later restoration of correct flag values.

Figure 6–3. Unified Flow of Traps and Interrupts



**LATcond** (link and trap conditionally) is a delayed instruction that provides a single-cycle trap that is very useful for error detection and correction. Since **LATcond** is a delayed instruction, the three instructions following **LATcond** should not modify the GIE or CF status register bits (this could result in storing incorrect values of these two bits).

The **RETIcond** and **RETIcondD** instructions manipulate the status flags as shown in step (3) in the figure. **RETIcondD** provides a delayed return from a trap or interrupt. Since traps and interrupts are unified, the **RETIcond** provides a return from either.

In general, you should not directly modify the PGIE or PCF status register bits except when putting the status register on a stack for recursive interrupts or traps.

## 6.5 Interlocked Operations

One of the most common parallel processing configurations is the sharing of global memory by multiple processors. In order for multiple processors to access this global memory and share data in a coherent manner, some sort of arbitration or handshake is necessary. The TMS320C40 interlocked operations meet this requirement for arbitration. More details are given in Section 7.7 on page 7-39. Examples in this section show you how interlocked operations can be used to implement:

- ❑ A busy-waiting loop used to synchronize processors at the software level (Example 6-5, page 6-15),
- ❑ A counter shared between cooperative processors defining the number of times a task should be done by the processors (Example 6-6 on page 6-15),
- ❑ Semaphores used to ease the programming of critical sections (Example 6-7 and Example 6-8 on page 6-16).

The TMS320C40 has five instructions referred to as interlocked operations. Through the use of external signals, these instructions provide powerful synchronization mechanisms. They also guarantee the integrity of the communication and result in a high-speed operation. The interlocked-operation instruction group is listed in Table 6-2.

Table 6-2. Interlocked Operations

Instruction	Description	Operation
LDFI	Load floating-point value from memory into a register, interlocked when <i>external</i> memory accessed	Signal interlocked src → dst
LDII	Load integer from memory into a register, interlocked when <i>external</i> memory accessed	Signal interlocked src → dst
SIGI	Load floating-point value from memory into a register, interlocked when <i>external</i> memory accessed	Signal interlocked Clear interlock
STFI	Store floating-point value from a register to memory, interlocked when <i>external</i> memory accessed	src → dst Clear interlock
STII	Store integer from a register to memory, interlocked when <i>external</i> memory accessed	src → dst Clear interlock

The interlocked operations use the global- and local-bus signals,  $\overline{LOCK}$  and  $\overline{LLOCK}$ , to reflect a currently executing interlocked operation. This signal is active (low) when any of the interlocked instructions in Table 6-2 are executing.

The external timing for the interlocked loads and stores is the same as for standard load and stores. You can extend the interlocked loads and stores like standard accesses by using the appropriate ready signal ( $\overline{\text{RDY}}_x$  or  $\overline{\text{LRDY}}_x$ ).

The **LDFI** and **LDII** instructions perform the following actions:

- 1) Pull  $\overline{\text{(L)LOCK}}$  low.
- 2) Execute an LDF or LDI instruction.
- 3) Extend the read cycle until the appropriate ready signal is received. Complete the instruction.
- 4) Leave  $\overline{\text{(L)LOCK}}$  active low until changed by an STF<sub>I</sub>, STII, or SIGI.

The read/write operation is identical to any other read/write cycle except for the special use of  $\overline{\text{(L)LOCK}}$ . The *src* operand for LDFI and LDII is always a direct or indirect memory address.  $\overline{\text{(L)LOCK}}$  is set to 0 only if the *src* is located off-chip (i.e.,  $\overline{\text{STRB}}$  or  $\overline{\text{LSTRB}}$  is active). If on-chip memory is accessed, then  $\overline{\text{(L)LOCK}}$  is not asserted, and the operation is as an LDF or LDI from internal memory.

6

The **STFI** and **STII** instructions perform the following operations:

- 1) Begin a write cycle. The state of  $\overline{\text{(L)LOCK}}$  does not change. If it is low, an interlocked operation occurs. If high, the operation is as if an STF or STI is performed (not interlocked).
- 2) Execute an STF or STI instruction and extend the write cycle until the appropriate ready is signaled.
- 3) After the write cycle, bring  $\overline{\text{(L)LOCK}}$  inactive (high).

As in the case for LDFI and LDII, the *dst* of STF<sub>I</sub> and STII affects  $\overline{\text{(L)LOCK}}$ . If *dst* is located off-chip ( $\overline{\text{STRB}}(0,1)$  or  $\overline{\text{LSTRB}}(0,1)$  is active),  $\overline{\text{(L)LOCK}}$  is set to a 1. If on-chip memory is accessed, then  $\overline{\text{(L)LOCK}}$  is not asserted, and the operations are as a STF or STI to internal memory.

The **SIGI** instruction functions as follows:

- 1) Pulls  $\overline{\text{(L)LOCK}}$  low.
- 2) Executes an LDI instruction.
- 3) Extends the read cycle until the appropriate ready signal is received. Completes the instruction.
- 4) Brings  $\overline{\text{(L)LOCK}}$  back inactive high.

Interlocked operations can be used to implement a busy-waiting loop, to manipulate a multiprocessor counter, to implement a simple semaphore mechanism, or to perform synchronization between two TMS320C40s. The following examples illustrate the usefulness of the interlocked operations instructions.

Example 6–5 shows the implementation of a busy-waiting loop. If location LOCK is the interlock for a critical section of code, and a nonzero means the lock is busy, the algorithm for a busy-waiting loop can be used as shown.

*Example 6–5. Busy-Waiting Loop*

```

        LDI    1,R0           ; Put 1 in R0
L1:    LDII   @LOCK,R1       ; Load lock value into R1
        STII   R0,@LOCK     ; Set lock value to 1
        BNZ   L1            ; If lock is not 0, read it again

```

Example 6–6 shows how a location COUNT may contain a count of the number of times a particular operation needs to be performed. This operation may be performed by any processor in the system. If the count is zero, the processor waits until it is nonzero before beginning processing. The example also shows the algorithm for modifying COUNT correctly.

*Example 6–6. Task Counter Manipulation*

```

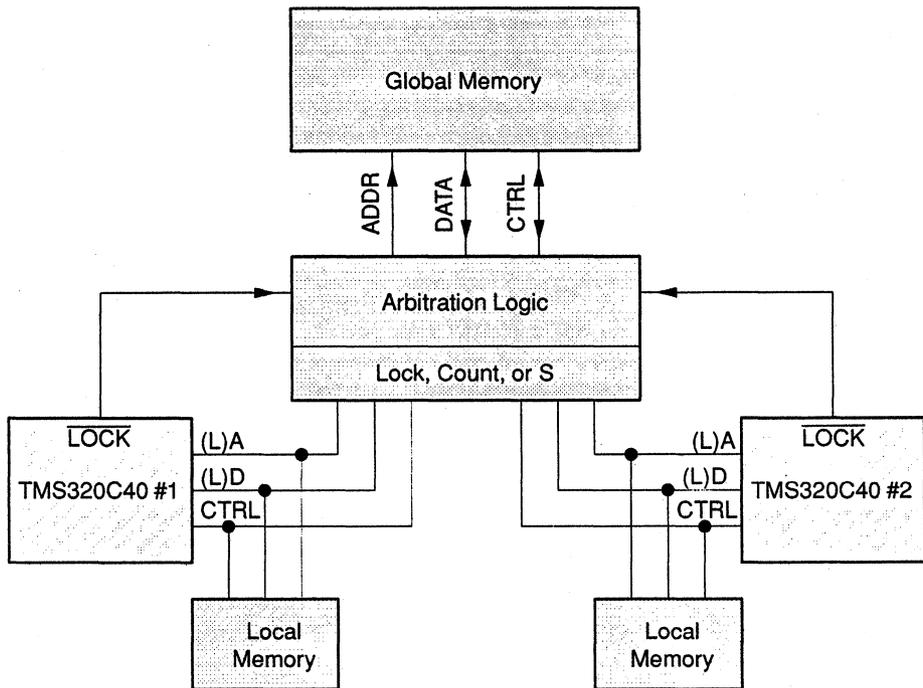
        LDI    0,R0
WAIT:  LDII   @COUNT,R1    ; Read current value of counter
        BZD   WAIT         ; If COUNT = 0, try again
        LDNZ  1,R0         ; If COUNT not zero, decrement it
        SUBI  R0,R1
        STII  R1,@COUNT   ; Update COUNT

```

6

Figure 6–4 illustrates multiple TMS320C40s sharing global memory and using the interlocked instructions as in Example 6–7 and Example 6–8.

Figure 6-4. Multiple TMS320C40s Sharing Global Memory



6

Example 6-7. Implementation of V(S)

```
V: LDII  @S,R1
    ADDI 1,R0
    STII  R0,@S ; S + 1 → S
```

Example 6-8. Implementation of P(S)

```
LDI 0,R0
P: LDII @S,R1 ; Read semaphore's current value
    BZD P ; If S = 0, go to P and try again
    LDNZ 1,R0 ; If S is not 0, decrement it
    SUBI R0,R1
    STII R1,@S ; Update S
```

Sometimes it may be necessary for several processors to access some shared data or other common resources. The portion of code that must access the shared data is called a *critical section*.

To ease the programming of critical sections, semaphores may be used. Semaphores are variables that can take only nonnegative integer values. Two primitive, indivisible operations are defined on semaphores (with S being a semaphore):

```
V(S):      S + 1 → S
P(S):      P: if (S == 0), go to P
            else S - 1 → S
```

Indivisibility of V(S) and P(S) means that when these processes access and modify the semaphore S, they are the only processes doing so.

To enter a critical section, a P operation is performed on a common semaphore, e.g., S (S is initialized to 1). The first processor performing P(S) will be able to enter its critical section. All other processors are blocked because S has become 0. After leaving its critical section, the processor performs a V(S), thus allowing another processor to execute P(S) successfully.

6

The TMS320C40 code for V(S) is shown in Example 6–7, and code for P(S) is shown in Example 6–8. Compare the code in Example 6–8 to the code in Example 6–6.

## 6.6 Reset Operation

The TMS320C40 supports a nonmaskable external reset signal ( $\overline{\text{RESET}}$ ), which is used to perform system reset. This section discusses the reset operation.

At powerup, the state of the TMS320C40 processor is undefined. You can use the  $\overline{\text{RESET}}$  signal to place the processor in a known state. This signal must be asserted low for 10 or more H1 clock cycles to guarantee a system reset. H1 is an output clock signal generated by the TMS320C40 (see Chapter 13 for more information).

Reset affects the other pins on the device in either a synchronous or asynchronous manner. The synchronous reset is gated by the TMS320C40s internal clocks. The asynchronous reset directly affects the pins, and it is faster than the synchronous reset. Table 6-3 shows the state of the TMS320C40s pins after  $\overline{\text{RESET}} = 0$ . Each pin is described according to whether the pin is reset synchronously or asynchronously.

Table 6-3. Pin Operation at Reset

Signal	Pins	Type	Description
<b>Global Bus External Interface (80 pins)</b>			
D(31-0)	32	I/O/T	Synchronous reset. Placed in high-impedance state.
$\overline{\text{DE}}$	1	I	Reset has no effect.
A(30-0)	31	O/T	Synchronous reset. Placed in high-impedance state.
$\overline{\text{AE}}$	1	I	Reset has no effect.
STAT(3-0)	4	O	Synchronous reset. Set to all ones.
$\overline{\text{LOCK}}$	1	O	Synchronous reset. Set to one.
$\overline{\text{STRB0}}$	1	O/T	Synchronous reset. Set to one.
$\overline{\text{R/W0}}$	1	O/T	Synchronous reset. Set to one.
PAGE0	1	O/T	Synchronous reset. Set to zero.
$\overline{\text{RDY0}}$	1	I	Reset has no effect.
$\overline{\text{CE0}}$	1	I	Reset has no effect.
$\overline{\text{STRB1}}$	1	O/T	Synchronous reset. Set to one.
$\overline{\text{R/W1}}$	1	O/T	Synchronous reset. Set to one.
PAGE1	1	O/T	Synchronous reset. Set to zero.
$\overline{\text{RDY1}}$	1	I	Reset has no effect.
$\overline{\text{CE1}}$	1	I	Reset has no effect.

Table Continued on Next Page

Table 6–3. Pin Operation at Reset (Continued)

Signal	Pins	Type	Description
<b>Local Bus External Interface (80 pins)</b>			
LD(31–0)	32	I/O/T	Synchronous reset. Placed in high-impedance state.
$\overline{\text{LDE}}$	1	I	Reset has no effect.
LA(30–0)	31	O/T	Synchronous reset. Placed in high-impedance state.
$\overline{\text{LAE}}$	1	I	Reset has no effect.
LSTAT(3–0)	4	O	Synchronous reset. Set to all ones.
$\overline{\text{LLOCK}}$	1	O	Synchronous reset. Set to one.
$\overline{\text{LSTRB0}}$	1	O/T	Synchronous reset. Set to one.
LR $\overline{\text{W0}}$	1	O/T	Synchronous reset. Set to one.
LPAGE0	1	O/T	Synchronous reset. Set to zero.
$\overline{\text{LRDY0}}$	1	I	Reset has no effect.
$\overline{\text{LCE0}}$	1	I	Reset has no effect.
$\overline{\text{LSTRB1}}$	1	O/T	Synchronous reset. Set to one.
LR $\overline{\text{W1}}$	1	O/T	Synchronous reset. Set to one.
LPAGE1	1	O/T	Synchronous reset. Set to zero.
$\overline{\text{LRDY1}}$	1	I	Reset has no effect.
$\overline{\text{LCE1}}$	1	I	Reset has no effect.
<b>Communication Port 0 Interface (12 pins)</b>			
C0D(7–0)	8	I/O	Synchronous reset. Placed in high-impedance state.
$\overline{\text{CREQ0}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CACK0}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CSTRB0}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CRDY0}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.

Table Continued on Next Page

Table 6-3. Pin Operation at Reset (Continued)

Signal	Pins	Type	Description
<b>Communication Port 1 Interface (12 pins)</b>			
C1D(7-0)	8	I/O	Synchronous reset. Placed in high-impedance state.
$\overline{\text{CREQ1}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CACK1}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CSTRB1}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CRDY1}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
<b>Communication Port 2 Interface (12 pins)</b>			
C2D(7-0)	8	I/O	Synchronous reset. Placed in high-impedance state.
$\overline{\text{CREQ2}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CACK2}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CSTRB2}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CRDY2}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
<b>Communication Port 3 Interface (12 pins)</b>			
C3D(7-0)	8	I/O	Synchronous reset. Placed in high-impedance state.
$\overline{\text{CREQ3}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CACK3}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CSTRB3}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CRDY3}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
<b>Communication Port 4 Interface (12 pins)</b>			
C4D(7-0)	8	I/O	Synchronous reset. Placed in high-impedance state.
$\overline{\text{CREQ4}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CACK4}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CSTRB4}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CRDY4}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
<b>Communication Port 5 Interface (12 pins)</b>			
C5D(7-0)	8	I/O	Synchronous reset. Placed in high-impedance state.
$\overline{\text{CREQ5}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CACK5}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CSTRB5}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.
$\overline{\text{CRDY5}}$	1	I/O	Asynchronous reset. Placed in high-impedance state.

Table Concluded on Next Page

Table 6-3. Pin Operation at Reset (Concluded)

Signal	Pins	Type	Description
<b>Interrupts, I/O Flags, Reset, Timer (12 pins)</b>			
IIOF(0 – 3)	4	I/O	Asynchronous reset. Placed in high-impedance state.
NMI		I	Reset has no effect.
$\overline{\text{TACK}}$	1	I	Synchronous reset.
RESET	1	I	RESET input pin
RESETLOC(1,0)	2	I	Reset has no effect.
ROMEN	1	I	Reset has no effect.
TCLK0	1	I/O	Asynchronous reset. Placed in high-impedance state.
TCLK1	1	I/O	Asynchronous reset. Placed in high-impedance state.
<b>Clock and Power (4 pins)</b>			
X1	1	O	Reset has no effect.
X2/CLKIN	1	I	Reset has no effect.
H1	1	I	Synchronous reset. Will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition.
H3	1	I	Synchronous reset. Will go to its initial state when $\overline{\text{RESET}}$ makes a 1 to 0 transition.
<b>Emulation (7 pins)</b>			
TCK	1	I	Reset has no effect.
TDO	1	O	Reset has no effect.
TDI	1	I	Reset has no effect.
TMS	1	I	Reset has no effect.
$\overline{\text{TRST}}$	1	I	Reset has no effect.
EMU0	1	I/O	Undefined.
EMU1	1	I/O	Undefined.

At system reset, the following additional operations are performed:

- ❑ Timer registers (Section 9.10 on page 9-45) are set.
  - Timer global control register set to zero except that bit DATIN is set to the value on pin TCLK.
  - Timer counter and timer period registers set to zeroes.
- ❑ Communications port control registers (subsection 8.4.1 on page 8-9) set to zeroes.
- ❑ External memory interface control registers (Section 7.2 on page 7-6) are set to 3E39 FFF0h.
- ❑ DMA channel control register, DMA transfer counter, and DMA auxiliary transfer counter (subsection 9.3.1 on page 9-7) are set to zeroes.
- ❑ The following CPU registers are loaded with zeroes (each described in Chapter 3):
  - ST (CPU status register)
  - IIE (CPU internal interrupt enable register)
  - IIF (interrupt flag register; controls pins IIOF(3–0))
  - DIE (DMA internal enable register)
  - IVTP (interrupt-vector table pointer)
  - TVTP (trap-vector table pointer)
- ❑ Then the reset vector is read from its location and loaded into the PC. This vector contains the start address of the system reset routine.
- ❑ Execution begins. Refer to Section 12.1 on page 12-3 for an example of a processor initialization routine.

Multiple TMS320C40s driven by the same system clock may be reset and synchronized. When the 1-to-0 transition of RESET occurs, the processor is placed on a well-defined internal phase, and all of the TMS320C40s will come up on the same internal phase.

## 6.7 Interrupts

The TMS320C40 supports multiple internal and external interrupts, which can be used for a variety of applications. This section discusses the operation of these interrupts. Additional information regarding internal interrupts can be found in Section 8.4 (page 8-8), Section 8.6 (page 8-17), Table 8-1 (communication ports on page 8-10), Section 9.9 (DMA on page 9-40), and Section 9.10 (timers on page 9-45).

The four external interrupts (IIOF0–IIOF3 as shown in Figure 6-6) are enabled at the **IIE register** (subsection 3.1.9, page 3-10). They are synchronized internally. They are sampled on the falling edge of H1 and passed through a series of H1/H3 delays internally. Once synchronized, the interrupt input will set the corresponding interrupt flag register (IIF) bit if the interrupt is active. These are the external interrupts and their corresponding interrupt vectors (the latter shown in Figure 6-6 on page 6-27):

IIOF Pin & Interrupt	Interrupt Vector Location
IIOF0	IVTP + 003h
IIOF1	IVTP + 004h
IIOF2	IVTP + 005h
IIOF3	IVTP + 006h

These interrupts are prioritized in that one is selected over the other if both come on the same clock cycle (IIOF0 the highest, IIOF1 next, etc.). When an interrupt is taken, the status register ST(GIE) bit is reset to 0, disabling any other incoming interrupt (except NMI — nonmaskable interrupt). This prevents any other interrupt (IIOF0–3) from assuming program control until the ST(GIE) bit is set back to 1. The NMI (an incoming low on pin AJ5, signal **NMI**) is not masked by the ST(GIE) bit. On a return from an interrupt routine, the RETI and RETI*cond* instructions place the value that is in the ST(PGIE) bit into the ST(GIE) bit, returning it to its value before the context switch.

Even though the **NMI** is nonmaskable, it is temporarily masked during delayed branches and multicycle CPU operations. NMI is a negative-going, edge-triggered, latched interrupt.

External interrupts can be effectively either edge- or level-triggered, depending on how the TYPE fields are set in the **IIF register** (see Table 3-6 on page 3-13). An external interrupt must be held low for at least one H1/H3 cycle to be recognized by the TMS320C40. For level-triggered interrupts, if the interrupt is held low for between one and two cycles, then only one interrupt is recognized. If the interrupt is held low two or more cycles, more than one interrupt may be recognized, depending on how rapidly interrupts are serviced.

### 6.7.1 Interrupt Control Bits

When a particular interrupt is processed by the CPU or DMA controller, the corresponding interrupt flag bit is cleared by the internal interrupt acknowledge signal. It should be noted, however, that for level-triggered interrupts, if  $\overline{\text{IIOFn}}$  is still low when the interrupt acknowledge signal occurs, the interrupt flag bit will be cleared for only one cycle and then set again because  $\overline{\text{IIOFn}}$  is still low. Accordingly, it is theoretically possible that, depending on when the **IIF register** (described in subsection 3.1.10 on page 3-12) is read, this bit may be zero even though  $\overline{\text{IIOFn}}$  is zero. When the TMS320C40 is reset, zero is written to the interrupt flag register, thereby clearing all pending interrupts.

The interrupt flag register bits may be read and written to under software control. If, at the IIF register,  $\text{FUNCx} = 0$  and  $\text{TYPEx} = 1$ , then external pin  $\overline{\text{IIOFx}}$  can be written to. Writing a 1 to the IIF register  $\text{FLAGx}$  bit has the same effect as an incoming interrupt received on the corresponding pin. In this way, all interrupts may be triggered and/or cleared through software. Since the interrupt bits also may be read ( $\text{TYPEx} = 0$ ), the interrupt pins may be polled in software when an interrupt-driven interface is not required.

Internal interrupts operate in a similar manner. In the IIF register, the bit corresponding to an internal interrupt (e.g.,  $\text{TINT0}$ ,  $\text{TINT1}$ ) may be read and written to through software. Writing a 1 sets the interrupt latch, and writing a 0 clears it. All internal interrupts are one H1/H3 cycle in length.

The CPU global interrupt enable bit (GIE), located in the CPU status register (ST), controls all CPU interrupts. All DMA interrupts are controlled by the DMA enable register bits and the SYNC bits of the DMA channel control registers (described in Figure 9-2 and Table 9-1 on page 9-8). The DMA interrupts are not dependent upon ST(GIE) and are local to the DMA.

To provide for maximum performance in servicing interrupts, the interrupt acknowledge (IACK) instruction is provided. IACK drives the  $\overline{\text{IACK}}$  pin and performs a dummy read. The read is performed from the address specified by the IACK instruction operand. When IACK is used, it typically is placed in the early portion of an interrupt service routine. For certain applications, it may be better suited at the end of the interrupt service routine or be totally unnecessary.

### 6.7.2 Prioritization and Control

The prioritization of interrupts is handled by the CPU according to the interrupt vector table shown in Figure 6-6. Prioritization is according to position in the table — those with displacements closest to the IVTP base address are higher in priority (i.e., NMI is higher than  $\text{TINT0}$ , which is higher than

IIOF0, etc.). Note that interrupt TINT0 is located at IVTP + 2 while the TINT1 vector is after the communication port and DMA coprocessor interrupts at IVTP + 2Bh.

*Prioritization* means an interrupt in a higher position in the interrupt vector table (Figure 6–6) will be accepted over one in a lower position *when both are received in the same clock cycle*. It *does not* mean, for example, that IIOF3 must wait until service routines for IIOF2, IIOF1, and IIOF0 are completed (when ST(GIE) = 1).

If the DMA coprocessor is not using interrupts for synchronization of transfers, it will not be affected by the processing of the CPU interrupts. If the CPU is involved in a pipeline conflict (branch, register, or memory), it will not respond to the interrupts until that conflict is resolved. It is therefore possible to interrupt the CPU and DMA coprocessor simultaneously with the same or different interrupts and, in effect, synchronize their activities. For example, it may be necessary to cause a high-priority DMA coprocessor transfer that avoids but conflicts with the CPU, i.e., makes the DMA coprocessor a higher priority than the CPU. This may be accomplished by using an interrupt that causes the CPU to trap to an interrupt routine that contains an IDLE instruction. Then, if the same interrupt is used to synchronize DMA coprocessor transfers, the DMA coprocessor transfer counter can be used to generate an interrupt and, thus, return control to the CPU following the DMA coprocessor transfer.

6

Since the DMA coprocessor and CPU share the same set of interrupt flags, the DMA coprocessor may clear an interrupt flag before the CPU can respond to it. For example, if the CPU interrupts are disabled, the DMA coprocessor can respond to interrupts and thus clear the associated interrupt flags.

Note the following situations:

- ❑ If there is a delayed branch in the pipeline, interrupts are held pending until after the branch.
- ❑ If the interrupt occurs in the first cycle of the fetch of an instruction, the fetched instruction is discarded (not executed), and the address of that instruction is pushed to the top of the system stack.
- ❑ If the interrupt occurs after first cycle of the fetch (in the case of a multi-cycle fetch due to wait states), that instruction is executed, and the address of the next instruction to be fetched is pushed to the top of the system stack.
- ❑ If no program fetch is occurring, then no new fetch is performed.

After the address of the appropriate instruction has been pushed, the interrupt vector is fetched and loaded into the PC, and executed continues.

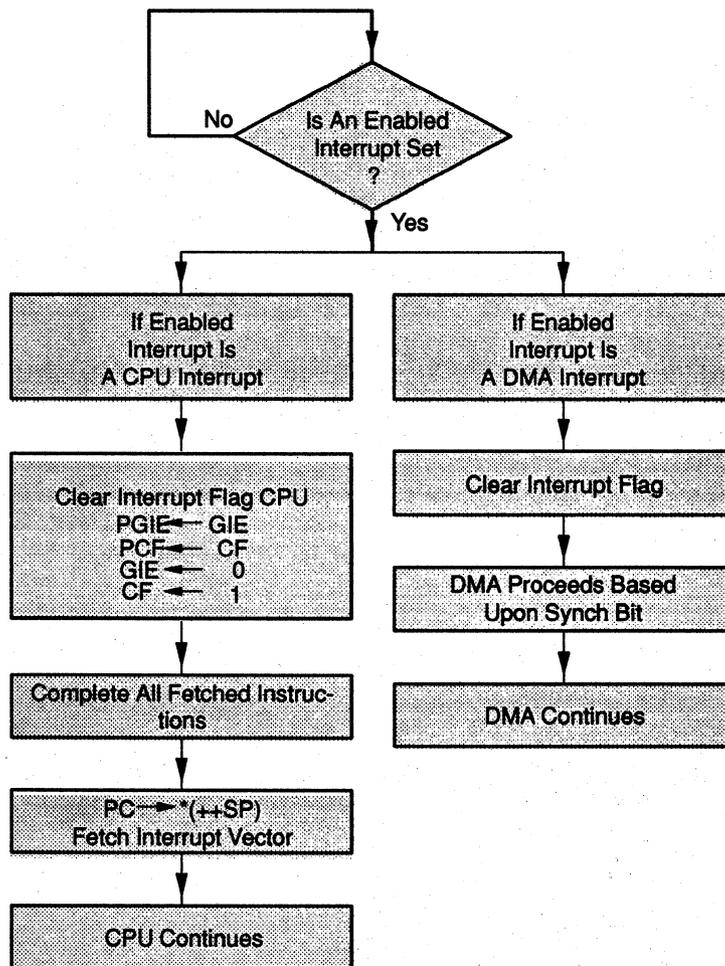
Figure 6-5. Interrupt-Vector Table (IVT)

IVTP + 000h	Reserved	Note 1	IVTP + 01Dh	ICFULL4	Note 5		
IVTP + 001h	NMI	Note 2	IVTP + 01Eh	ICRDY4			
IVTP + 002h	TINT0	Note 3	IVTP + 01Fh	OCRDY4			
IVTP + 003h	$\overline{\text{IIOF}}_0$	Note 4	IVTP + 020h	OCEMPTY4			
IVTP + 004h	$\overline{\text{IIOF}}_1$		IVTP + 021h	ICFULL5			
IVTP + 005h	$\overline{\text{IIOF}}_2$		IVTP + 022h	ICRDY5			
IVTP + 006h	$\overline{\text{IIOF}}_3$		IVTP + 023h	OCRDY5			
IVTP + 007h	Unused	Note 5	IVTP + 024h	OCEMPTY5	Note 6		
IVTP + 00Ch			Note 5	IVTP + 025h		DMA INT0	
IVTP + 00Dh				ICFULL0		IVTP + 026h	DMA INT1
IVTP + 00Eh	ICRDY0			IVTP + 027h		DMA INT2	
IVTP + 00Fh	OCRDY0			IVTP + 028h		DMA INT3	
IVTP + 010h	OCEMPTY0			IVTP + 029h	DMA INT4		
IVTP + 011h	ICFULL1			IVTP + 02Ah	DMA INT5		
IVTP + 012h	ICRDY1			IVTP + 02Bh	TINT1	Note 3	
IVTP + 013h	OCRDY1			IVTP + 02Ch	Unused	Note 5	
IVTP + 014h	OCEMPTY1			IVTP +			Unused
IVTP + 015h	ICFULL2	IVTP +		Unused			
IVTP + 016h	ICRDY2	IVTP +	Unused				
IVTP + 017h	OCRDY2	IVTP +					
IVTP + 018h	OCEMPTY2	IVTP +			Unused		
IVTP + 019h	ICFULL3	IVTP +				Unused	
IVTP + 01Ah	ICRDY3	IVTP +		Unused			
IVTP + 01Bh	OCRDY3	IVTP + 03Eh	Reserved				
IVTP + 01Ch	OCEMPTY3	IVTP + 03Fh					

- Notes:**
- 1) Reserved for the **reset** vector when IVTP = 0000 0000h and RESETLOC(1,0) = 0 0<sub>2</sub> or when IVTP=08000 0000h and RESETLOC(1,0) = 1 0<sub>2</sub>. See Table 3-8.
  - 2) NMI (non-maskable interrupt) is discussed in Section 9.9, page 9-40.
  - 3) **Timer** interrupts TINT0 and TINT1 are enabled and programmed by the IIE register (subsection 3.1.9, page 3-10) and monitored at the IIF register (subsection 3.1.10, page 3-12).
  - 4) External pins  $\overline{\text{IIOF}}_0$ — $\overline{\text{IIOF}}_5$  are programmed in the DIE register (subsection 3.1.8, page 3-8) and the IIF register (subsection 3.1.10, page 3-12).
  - 5) The **communication port** I/O buffers full/ready interrupts are enabled by the DIE and IIE registers and also discussed in Table 8-1, page 8-10 (OUTPUT LEVEL & INPUT LEVEL bits).
  - 6) **DMA** interrupts are enabled at the IIE register and DMA channel control register (at bits TCC and AUX TCC explained in Table 9-1 on page 9-8).

The TMS320C40 allows the CPU and DMA coprocessor to respond to and process interrupts in parallel. Figure 6–6 shows interrupt processing flow. The interrupts are polled, and the CPU and DMA coprocessor begin processing them. In the interrupt flow pertaining to the CPU (left side of figure), the interrupt flag corresponding to the highest priority enabled interrupt is cleared, and GIE is set to 0. The CPU completes all fetched instructions. The interrupt vector is fetched and loaded into the PC, and the CPU continues execution. The DMA coprocessor cycle (right side of figure) is similar to that for the CPU. After the pertinent interrupt flag is cleared, the DMA coprocessor proceeds according to the status of the SYNCH bits in the DMA coprocessor global control register.

Figure 6–6. Interrupt Processing





6

# External Bus Operation

---

---

---

The TMS320C40 has two identical 80-pin parallel external interfaces: the **global memory interface** and the **local memory interface**. Each interface has the following features:

- ❑ separate 80-pin configurations, each with its own 32-bit data bus and 31-bit address bus,
- ❑ single-cycle reads and pipelined writes,
- ❑ independent enable signals for data, address, and control lines,
- ❑ bus-request and bus-lock signaling for share memory parallel processing,
- ❑ user-controlled mapping of addresses to either of two sets of independent strobes for different speed memories,
- ❑ look-ahead bus status signals for defining current and requested bus operations for parallel processing arbitration,
- ❑ selectable wait states (both software- and hardware-controlled),
- ❑ signals that indicate when memory page boundaries are crossed. This supports
  - page-mode and static-column decode DRAMs,
  - high-speed SRAM banks, and
  - slower-speed memory banks and I/O devices.

---

**Note: Description Covers Both Interfaces in this Chapter**

This chapter covers both the global memory interface and the local memory interface. However, only the global memory interface is shown throughout this chapter because it is identical in every way to the local memory interface except that (1) they have different positions in the memory map, and (2) the control signals for the local memory interface have an additional “L” prefix (as described in Figure 7–1 on page 7-3).

---

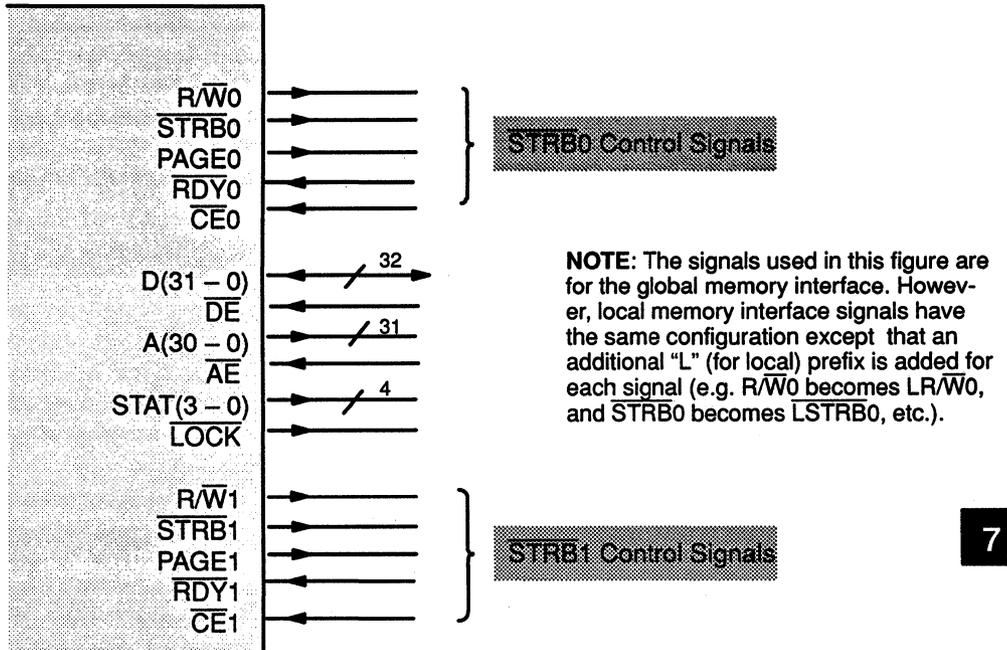
Principal headings within this chapter:

<b>Section</b>	<b>Page</b>
7.1 Global (and Local) Memory Interface Control Signals .....	7-3
7.2 Memory Interface Control Registers .....	7-6
7.3 Use of the Global Memory Interface Registers .....	7-12
7.4 Programmable Wait States .....	7-15
7.5 Timing .....	7-17
7.6 Using Enabled Signals to Control Signal Group .....	7-38
7.7 Interlocked-Instructions Definition and Timing .....	7-39
7.8 $\overline{\text{IACK}}$ Timing .....	7-47

## 7.1 Global (and Local) Memory Interface Control Signals

As explained in the Note on page 7-1, this text covers the global memory interface control signals; it also applies to the local memory interface control signals (with the exceptions stated in the note).

Figure 7-1. Global and Local Memory Interface Control Signals



As shown in Figure 7-1, the global memory interface has two sets of control signals,  $\overline{STRB0}$  and  $\overline{STRB1}$ . The global memory port control registers (Section 7.2 on page 7-6) define which set of registers is active.

Table 7-1. Global Memory Interface Control Signals

Signal†	Type§	Description
R/W(0,1)	O/Z	Specifies memory read (active high) or write (active low) mode.
STRB(0,1)	O/Z	Interface access strobe.
PAGE(0,1)	O/Z	Memory-page enable signal for STRB(0,1) accesses.
RDY(0,1)	I	Indicates external memory is ready to be accessed.
CE(0,1)	I	Control signal enable for R/Wx, STRBx, and PAGEx signals. When high (a one), it places the corresponding R/Wx, STRBx, and PAGEx signals in high-impedance state (x=0 for CE0 and x=1 for CE1).
DE	I	When high (a one), places data lines D31 – 0 in high-impedance state.
AE	I	When high (a one), places address lines A30 – 0 in high-impedance state.
STAT(3 – 0)‡	O	Four lines to define status or function of the memory port as shown in Table 7-2 (next page).
LOCK‡	O	Indicates if an interlocked access is underway (0 = access underway; 1 = access not underway). LOCK is changed <i>only</i> by the interlocked instructions.

† This table applies to both the global memory interface and local memory interface (local memory interface signals have an additional “L” prefix). The numbers in parentheses mean that either a 0 (zero) or a 1 can follow the prefix shown to the left of the parentheses. A zero indicates STRB0 control signals (shown in Figure 7-1), and a one indicates STRB1 control signals.

§ O = output; I = input; Z = high impedance (three-stated).

‡ STAT(3 – 0) and LOCK **cannot** be placed in the high-impedance state by an external control signal.

Table 7-2 on the next page shows how pins STAT3 to STAT0 define the current status of the global memory port. For many bus accesses, these signals provide information about the access that is about to begin. The code for a SIGI instruction read is useful for distinguishing between a SIGI read and a LDII or LDFI read.

The **bus idle** status code is 1111<sub>2</sub> (bottom of Table 7-2), which simplifies modular shared-bus multiprocessor interfaces, because pull-up resistors can be used to signal the idle condition when processor cards are not attached to the shared bus.

Table 7-2. Global Memory Port Status for  $\overline{STRB0}$  and  $\overline{STRB1}$  Accesses

Value at Pins †				Status
STAT3	STAT2	STAT1	STAT0	
0	0	0	0	$\overline{STRB0}$ access, program read
0	0	0	1	$\overline{STRB0}$ access, data read
0	0	1	0	$\overline{STRB0}$ access, DMA read
0	0	1	1	$\overline{STRB0}$ access, SIGI (instruction) read
0	1	0	0	Reserved
0	1	0	1	$\overline{STRB0}$ access, data write
0	1	1	0	$\overline{STRB0}$ access, DMA write
0	1	1	1	Reserved
1	0	0	0	$\overline{STRB1}$ access, program read
1	0	0	1	$\overline{STRB1}$ access, data read
1	0	1	0	$\overline{STRB1}$ access, DMA read
1	0	1	1	$\overline{STRB1}$ access, SIGI (instruction) read
1	1	0	0	Reserved
1	1	0	1	$\overline{STRB1}$ access, data write
1	1	1	0	$\overline{STRB1}$ access, DMA write
1	1	1	1	Idle

† This table applies to both the global memory interface and local memory interface (for local memory interface signals, add an additional "L" prefix such as LSTAT3, LSTAT2, etc.).

## 7.2 Memory Interface Control Registers

As explained in the Note on page 7–1, this text covers the global memory interface control signals; it also applies to the local memory interface control signals (with the exceptions stated in the note).

Figure 7–2 shows the memory map for both the global and local memory interface control registers. Each register can be programmed to control its respective memory interface by defining:

- ❑ page sizes for the two strobes,
- ❑ when strobes are active,
- ❑ wait states,
- ❑ other operations that control the memory interface.

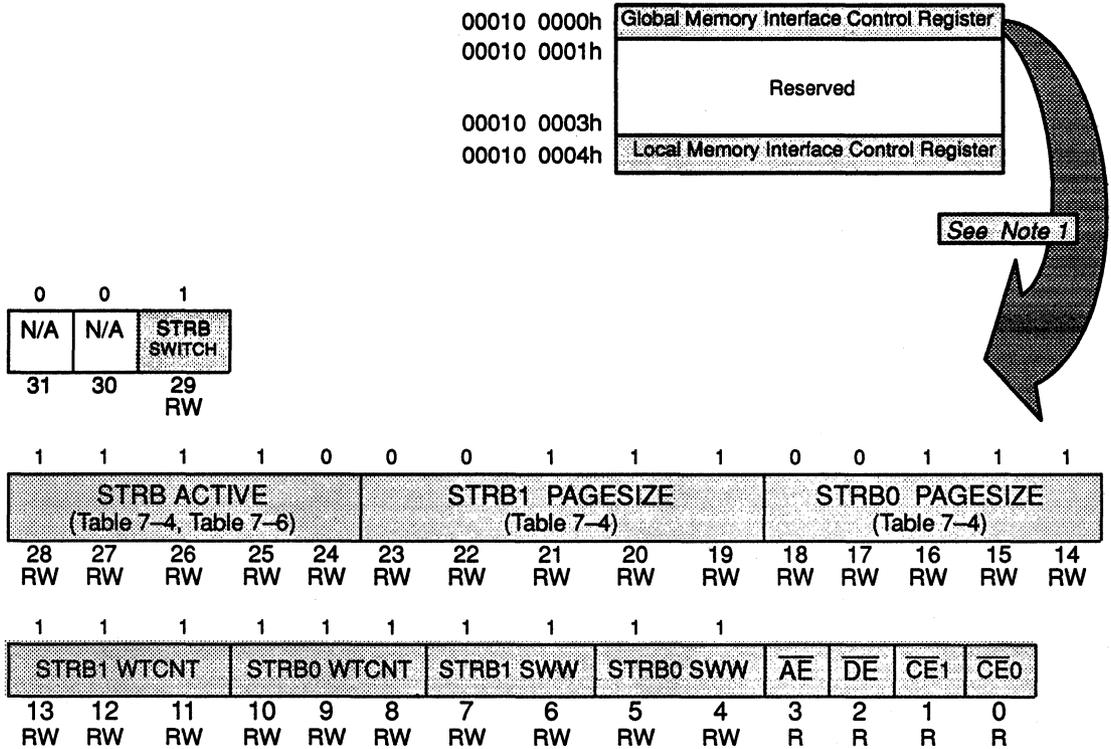
Table 7–3 (on page 7-8) describes the fields in these registers.

At **reset**, the binary values shown above each bit in Figure 7–1 are written to the global memory interface control register. Values in bits 3 – 0 are the values at these bits' respective pins ( $\overline{AE}$ ,  $\overline{DE}$ ,  $\overline{CE1}$ , and  $\overline{CE0}$ ). This reset condition has the following effects (for the local and global bus):

- ❑  $\overline{STRB0}$  and  $\overline{STRB1}$  ( $\overline{LSTRB0}$  and  $\overline{LSTRB1}$ ) page sizes are set to  $00111_2$  (256 words).
- ❑  $\overline{STRB0}$  and  $\overline{STRB1}$  ( $\overline{LSTRB0}$  and  $\overline{LSTRB1}$ ) wait states are set to 7 cycles.
- ❑  $\overline{STRB0}$  and  $\overline{STRB1}$  ( $\overline{LSTRB0}$  and  $\overline{LSTRB1}$ ) accesses require an external ready signal and an internal ready signal generated by the software wait-state generator.
- ❑  $\overline{STRB0}$  ( $\overline{LSTRB0}$ ) is active for all addresses over the global (local) memory interface.
- ❑ Back-to-back reads that switch from  $\overline{STRB0}$  to  $\overline{STRB1}$  (or  $\overline{STRB1}$  to  $\overline{STRB0}$ ) result in the insertion of a single cycle between these reads.

As shown in Figure 7–2, fields STRB1 SWW and STRB0 WW are both set to  $11_2$  to allow the internal ready signal to be generated by  $RDY_{wcnt}$  (on-chip wait-state counter) and external  $\overline{RDY}$ .

Figure 7-2. Format for the Memory-Interface Control Registers



- NOTES:**
1. The register cell figure (immediately above) contains global memory interface control register mnemonics. However, local memory interface control register mnemonics can be visualized by adding an "L" prefix to each mnemonic in the figure (e.g., LSTRB SWW, LCE0, etc.).
  2. The 1s and 0s above each bit are the binary values written to the register at reset. The values at bits 3 – 0 are defined by the values of their respective external pins (AE, DE, CE1, and CE0).
  3. These registers are shown in the overall memory map in Figure 3-9 and Figure 3-10 on pages 3-19 and 3-20, respectively.
  4. RW = read/write; R = read.

Table 7-3. Bit Definitions for Both Memory Interface Control Registers

Bit No.	Mnemonic†	Description†
0	$\overline{CE0}$	<b>Value of external pin <math>\overline{CE0}</math></b> (after it passes through an internal synchronizer). The value is not latched.
1	$\overline{CE1}$	<b>Value of external pin <math>\overline{CE1}</math></b> (after it passes through an internal synchronizer). The value is not latched.
2	$\overline{DE}$	<b>Value of external pin <math>\overline{DE}</math></b> (after it passes through an internal synchronizer). The value is not latched.
3	$\overline{AE}$	<b>Value of external pin <math>\overline{AE}</math></b> (after it passes through an internal synchronizer). The value is not latched.
4 – 5	STRB0 SWW	<b>Software wait states for STRB0 access.</b> In conjunction with STRB0 WTCNT, this field defines the mode of wait-state generation. Actual wait states are explained in Section 7.4 and in Table 7-7 on page 7-16.
6 – 7	STRB1 SWW	<b>Software wait states for STRB1 access.</b> In conjunction with STRB1 WTCNT, this field defines the mode of wait-state generation. Actual wait states are explained in Section 7.4 and in Table 7-7 on page 7-16.
8 – 10	STRB0 WTCNT	<b>Software wait-state count for STRB0 accesses.</b> Specifies the number of cycles to use when software wait states are active. Three-bit range is from $000_2$ (zero) to $111_2$ (seven).
11 – 13	STRB1 WTCNT	<b>Software wait-state count for STRB1 accesses.</b> Specifies the number of cycles to use when software wait states are active. Three-bit range is from $000_2$ (zero) to $111_2$ (seven).
14 – 18	STRB0 PAGESIZE	<b>Page size for STRB0 accesses.</b> Specifies number of MSBs of the address to use to define the bank size for STRB0 accesses. See range table in Table 7-4 on page 7-9.
19 – 23	STRB1 PAGESIZE	<b>Page size for STRB1 accesses.</b> Specifies number of MSBs of the address to use to define the bank size for STRB1 accesses. See range table in Table 7-4 on page 7-9.
24 – 28	STRB ACTIVE	Specifies <b>address ranges</b> over which $\overline{STRB0}^\dagger$ and $\overline{STRB1}^\dagger$ are active. See ranges in Table 7-5 on page 7-10 for <b>STRB ACTIVE</b> and Table 7-6 on page 7-11 for <b>LSTRB ACTIVE</b> .
29	STRB SWITCH	<b>Inserts a single cycle between back-to-back reads</b> that switch from $\overline{STRB0}$ to $\overline{STRB1}$ (or vice versa). When a <b>1</b> , insert cycle. When a <b>0</b> , don't insert cycle.
30 – 31	Reserved	Read as zeroes.

† Mnemonics used are for the global memory interface control register. For the local memory interface control register, add the prefix "L" to each mnemonic (e.g.,  $\overline{LCE0}$ ,  $\overline{LCE1}$ ,  $\overline{LSTRB1}$ , etc.). The description remains the same for the local memory interface control register.

Table 7-4. Page Size as Defined by STRB0/1 PAGESIZE Bits †

STRBx PAGESIZE Field‡	External Address Bus Bits Defining the Current Page	External Address Bus Bits Defining Address on a Page	Page Size (32-Bit Wds)
00000–00110	Reserved	Reserved	Reserved
00111	30 — 8	7 — 0	$2^8 = 256$
01000	30 — 9	8 — 0	$2^9 = 512$
01001	30 — 10	9 — 0	$2^{10} = 1K$
01010	30 — 11	10 — 0	$2^{11} = 2K$
01011	30 — 12	11 — 0	$2^{12} = 4K$
01100	30 — 13	12 — 0	$2^{13} = 8K$
01101	30 — 14	13 — 0	$2^{14} = 16K$
01110	30 — 15	14 — 0	$2^{15} = 32K$
01111	30 — 16	15 — 0	$2^{16} = 64K$
10000	30 — 17	16 — 0	$2^{17} = 128K$
10001	30 — 18	17 — 0	$2^{18} = 256K$
10010	30 — 19	18 — 0	$2^{19} = 512K$
10011	30 — 20	19 — 0	$2^{20} = 1M$
10100	30 — 21	20 — 0	$2^{21} = 2M$
10101	30 — 22	21 — 0	$2^{22} = 4M$
10110§	30 — 23	22 — 0	$2^{23} = 8M$
10111	30 — 24	23 — 0	$2^{24} = 16M$
11000	30 — 25	24 — 0	$2^{25} = 32M$
11001	30 — 26	25 — 0	$2^{26} = 64M$
11010	30 — 27	26 — 0	$2^{27} = 128M$
11011	30 — 28	27 — 0	$2^{28} = 256M$
11100	30 — 29	28 — 0	$2^{29} = 512M$
11101	30	29 — 0	$2^{30} = 1G$
11110	None	30 — 0	$2^{31} = 2G$
11111	Reserved	Reserved	Reserved

† Mnemonics used are for the global memory interface control register. For the local memory interface control register, add the prefix “L” to each mnemonic (e.g., LSTRB0 PAGESIZE, LSTRB1 PAGESIZE, etc.). The description remains the same for the local memory interface control register.

‡ The “x” in STRBx means that the data in the columns are for STRB0 or STRB1 as well as for LSTRB0 and LSTRB1, as explained in the note above.

§ An STRBx PAGESIZE field of 10110<sub>2</sub> is depicted in Figure 7-4 on page 7-13.

Table 7-5. Address Ranges Specified by STRB ACTIVE Bits †

STRB ACTIVE Field	STRB0 Active Address Range	STRB0 Active Address Range Size	STRB1 Active Address Range
00000–01110	Reserved	Reserved	Reserved
01111	8000 0000 — 8000 FFFF	$2^{16} = 64\text{K}$	8001 0000 — FFFF FFFF
10000	8000 0000 — 8001 FFFF	$2^{17} = 128\text{K}$	8002 0000 — FFFF FFFF
10001	8000 0000 — 8003 FFFF	$2^{18} = 256\text{K}$	8004 0000 — FFFF FFFF
10010	8000 0000 — 8007 FFFF	$2^{19} = 512\text{K}$	8008 0000 — FFFF FFFF
10011	8000 0000 — 800F FFFF	$2^{20} = 1\text{M}$	8010 0000 — FFFF FFFF
10100	8000 0000 — 801F FFFF	$2^{21} = 2\text{M}$	8020 0000 — FFFF FFFF
10101	8000 0000 — 803F FFFF	$2^{22} = 4\text{M}$	8040 0000 — FFFF FFFF
10110	8000 0000 — 807F FFFF	$2^{23} = 8\text{M}$	8080 0000 — FFFF FFFF
10111	8000 0000 — 80FF FFFF	$2^{24} = 16\text{M}$	8100 0000 — FFFF FFFF
11000	8000 0000 — 81FF FFFF	$2^{25} = 32\text{M}$	8200 0000 — FFFF FFFF
11001	8000 0000 — 83FF FFFF	$2^{26} = 64\text{M}$	8400 0000 — FFFF FFFF
11010	8000 0000 — 87FF FFFF	$2^{27} = 128\text{M}$	8800 0000 — FFFF FFFF
11011	8000 0000 — 8FFF FFFF	$2^{28} = 256\text{M}$	9000 0000 — FFFF FFFF
11100	8000 0000 — 9FFF FFFF	$2^{29} = 512\text{M}$	A000 0000 — FFFF FFFF
11101	8000 0000 — BFFF FFFF	$2^{30} = 1\text{G}$	C000 0000 — FFFF FFFF
11110	8000 0000 — FFFFFFFF	$2^{31} = 2\text{G}$	None
11111	Reserved	Reserved	Reserved

† Address ranges specified by the LSTRB ACTIVE bits are listed in Table 7-6.

Table 7-6. Address Ranges Specified by LSTRB ACTIVE Bits†

LSTRB ACTIVE Field	LSTRB0 Active Address Range	LSTRB0 Active Address Range Size	LSTRB1 Active Address Range
0000–01110	Reserved	Reserved	Reserved
01111	0000 0000 — 0000FFFF	$2^{16} = 64\text{K}$	0001 0000 — 7FFFFFFF
10000	0000 0000 — 0001FFFF	$2^{17} = 128\text{K}$	0002 0000 — 7FFFFFFF
10001	0000 0000 — 0003FFFF	$2^{18} = 256\text{K}$	0004 0000 — 7FFFFFFF
10010	0000 0000 — 0007FFFF	$2^{19} = 512\text{K}$	0008 0000 — 7FFFFFFF
10011	0000 0000 — 000FFFFFFF	$2^{20} = 1\text{M}$	0010 0000 — 7FFFFFFF
10100	0000 0000 — 001FFFFFFF	$2^{21} = 2\text{M}$	0020 0000 — 7FFFFFFF
10101	0000 0000 — 003FFFFFFF	$2^{22} = 4\text{M}$	0040 0000 — 7FFFFFFF
10110	0000 0000 — 007FFFFFFF	$2^{23} = 8\text{M}$	0080 0000 — 7FFFFFFF
10111	0000 0000 — 00FFFFFFF	$2^{24} = 16\text{M}$	0100 0000 — 7FFFFFFF
11000	0000 0000 — 01FFFFFFF	$2^{25} = 32\text{M}$	0200 0000 — 7FFFFFFF
11001	0000 0000 — 03FFFFFFF	$2^{26} = 64\text{M}$	0400 0000 — 7FFFFFFF
11010	0000 0000 — 07FFFFFFF	$2^{27} = 128\text{M}$	0800 0000 — 7FFFFFFF
11011	0000 0000 — 0FFFFFFF	$2^{28} = 256\text{M}$	1000 0000 — 7FFFFFFF
11100	0000 0000 — 1FFFFFFF	$2^{29} = 512\text{M}$	2000 0000 — 7FFFFFFF
11101	0000 0000 — 3FFFFFFF	$2^{30} = 1\text{G}$	4000 0000 — 7FFFFFFF
11110	0000 0000 — 7FFFFFFF	$2^{31} = 2\text{G}$	None
11111	Reserved	Reserved	Reserved

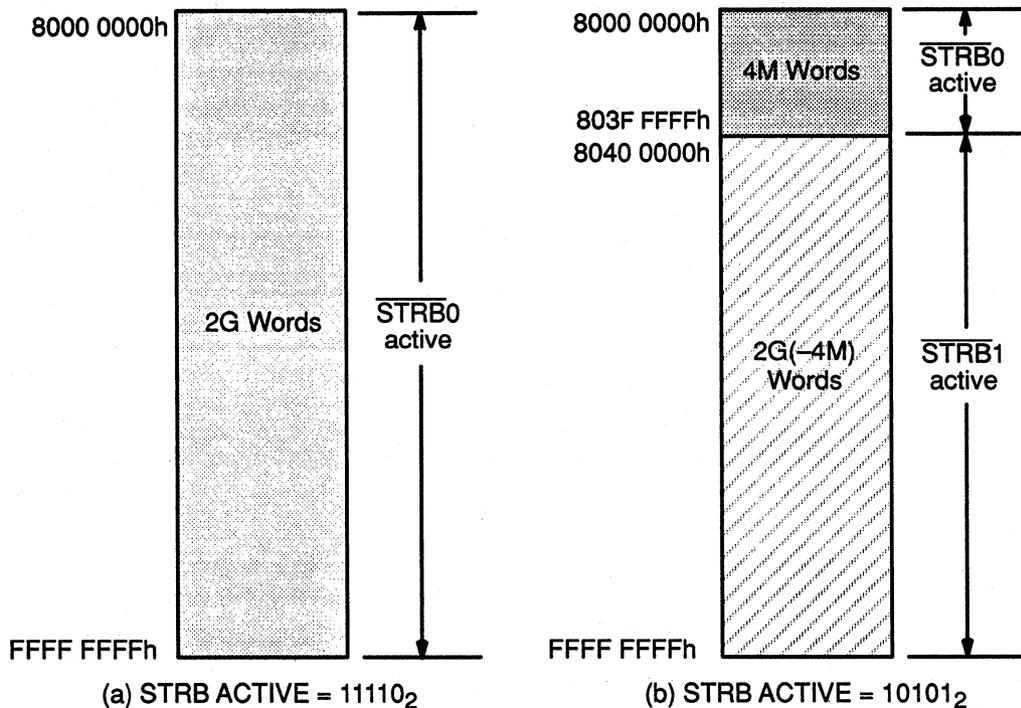
† Address ranges specified by the STRB ACTIVE bits are listed in Table 7-5.

## 7.3 Use of the Global Memory Interface Registers

### 7.3.1 Mapping Addresses to Strobes

Figure 7–3 demonstrates the relationship between the STRB ACTIVE bits (defined in Table 7–3, page 7-8) and the address ranges over which signals  $\overline{\text{STRB}}_0$  and  $\overline{\text{STRB}}_1$  are active. Note that the address ranges of  $\overline{\text{STRB}}_x$  and  $\overline{\text{LSTRB}}_x$  also govern the ranges of their associated signals  $\overline{\text{RDY}}_x$ ,  $\overline{\text{LRDY}}_x$ ,  $\text{R}/\overline{\text{W}}_x$ ,  $\text{LR}/\overline{\text{W}}_x$ ,  $\text{PAGE}_x$ ,  $\text{LPAGE}_x$ , etc. (where  $x = 1$  or  $0$ ).

Figure 7–3. Effects of STRB ACTIVE on Global Memory Bus Memory Map



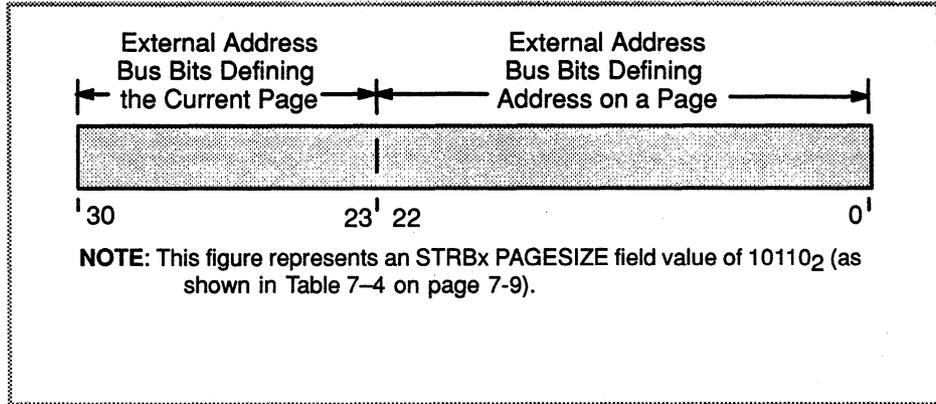
**NOTE:** Shown here are two examples for the global memory map. The entire 'C40 memory map (local and global) is shown in Figure 3–9 on page 3-19. Note that the highest address for  $\overline{\text{LSTRB}}_1$  (local bus) is 7FFF FFFFh.

Example (a) of Figure 7–3 shows the reset condition ( $\text{STRB ACTIVE} = 11110_2$ ). In this case, signal  $\overline{\text{STRB}}_0$  is active over the entire address range of the global memory bus (see Table 7–4 for lookup table of STRB ACTIVE).

Example (b) of Figure 7–3 shows the global memory bus memory map when  $\text{STRB ACTIVE} = 10101_2$ . In this case,  $\overline{\text{STRB0}}$  is active from addresses 8000 0000h – 803F FFFFh, and  $\overline{\text{STRB1}}$  is active from addresses 8040 0000h – FFFF FFFFh (as shown in Table 7–4 for an  $\text{STRB ACTIVE}$  of  $10101_2$ ).

### 7.3.2 Page Size Operation

Figure 7–4.  $\text{STRBx PAGESIZE}$  Fields Example



The TMS320C40 external interface allows you to specify (using a 31-bit address) independent page sizes for the different sets of external strobes. This capability, shown in the example in Figure 7–4, gives you a great deal of flexibility in the design of external high-speed, high-density memory systems and the use of slower external peripheral devices.

The  $\text{STRB0 PAGESIZE}$  and  $\text{STRB1 PAGESIZE}$  fields in the memory interface control register (shown in Figure 7–2 on page 7-7) work in the same manner to specify the page size for the corresponding strobe. Table 7–4 (page 7-9) illustrates the relationship between the  $\text{PAGESIZE}$  field and the bits of the address used to define the current page and the resulting page size. Page size begins at 256 words (with external address-bus bits 7 – 0 defining the address on a page, and ranges up to 2G words with external address bus bits 30 – 0 defining the location on a page. The example in Figure 7–4 shows how a pagesize field value of  $10110_2$  is translated into bits 30 – 23 defining the current page and bits 22 – 0 defining address on a page.

Changing from one page to another causes a cycle to be inserted in the external access sequence in order for external logic to reconfigure itself appropriately. The memory interface control logic keeps track of the address used for the last access for each  $\overline{\text{STRB}}$ . When an access begins, the PAGE signal corresponding to the active  $\overline{\text{STRB}}$  goes inactive (high) if the access is to a new page. The PAGE0 and PAGE1 signals are independent of one another, each having its own page-size logic.

At reset, the page-control logic is initialized so that the extra cycle is inserted for the first access to the two strobe interfaces.

The local memory interface has a similar set of control registers.

## 7.4 Programmable Wait States

Control wait-state generation by manipulating memory-mapped control registers associated with both the global and local interfaces. Use the STRBx WTCNT field to load an internal timer, and use the STRBx SWW field to select one of the following four modes of wait-state generation:

- External  $\overline{\text{RDY}}$
- WTCNT-generated  $\overline{\text{RDY}}_{\text{wtcnt}}$
- Logical-AND of  $\overline{\text{RDY}}$  and  $\overline{\text{RDY}}_{\text{wtcnt}}$
- Logical-OR of  $\overline{\text{RDY}}$  and  $\overline{\text{RDY}}_{\text{wtcnt}}$

Application of wait states and ready are covered in Section 13.4 on page 13-27.

The four modes are used to generate the internal ready signal,  $\overline{\text{RDY}}_{\text{int}}$ , that controls accesses. As long as  $\overline{\text{RDY}}_{\text{int}} = 1$ , the current external access is extended. When  $\overline{\text{RDY}}_{\text{int}} = 0$ , the current access completes. Since the use of programmable wait states for both external interfaces is identical, only the global bus interface is described in the following paragraphs.

$\overline{\text{RDY}}_{\text{wtcnt}}$  is an internally generated ready signal. When an external access is begun, the value in WTCNT is loaded into a counter. WTCNT may be any value from 0 through 7. The counter is decremented every H1/H3 clock cycle until it becomes 0. Once the counter is set to 0, it remains set to 0 until the next access. While the counter is nonzero,  $\overline{\text{RDY}}_{\text{wtcnt}} = 1$ . While the counter is 0,  $\overline{\text{RDY}}_{\text{wtcnt}} = 0$ .

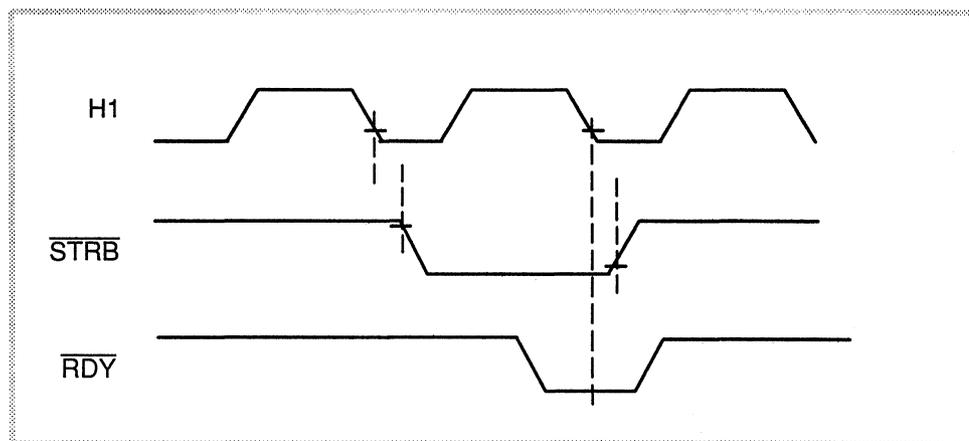
Table 7-7 is the truth table for each value of SWW, showing the different values at  $\overline{\text{RDY}}$ ,  $\overline{\text{RDY}}_{\text{wtcnt}}$ , and  $\overline{\text{RDY}}_{\text{int}}$ .

Table 7-7. Wait-State Generation for Each Value of SWW

SWW Value	$\overline{\text{RDY}}$	$\overline{\text{RDY}}_{\text{wcnt}}$	$\overline{\text{RDY}}_{\text{int}}$	$\overline{\text{RDY}}_{\text{int}}$
00	0	0	0	$\overline{\text{RDY}}_{\text{int}}$ is dependent only upon $\overline{\text{RDY}}$ . $\overline{\text{RDY}}_{\text{wcnt}}$ is ignored.
00	0	1	0	
00	1	0	1	
00	1	1	1	
01	0	0	0	$\overline{\text{RDY}}_{\text{int}}$ is dependent only upon $\overline{\text{RDY}}_{\text{wcnt}}$ . $\overline{\text{RDY}}$ is ignored.
01	0	1	1	
01	1	0	0	
01	1	1	1	
10	0	0	0	$\overline{\text{RDY}}_{\text{int}}$ is the logical-OR (electrical-AND, since these signals are low true) of $\overline{\text{RDY}}$ and $\overline{\text{RDY}}_{\text{wcnt}}$ .
10	0	1	0	
10	1	0	0	
10	1	1	1	
11	0	0	0	$\overline{\text{RDY}}_{\text{int}}$ is the logical-AND (electrical-OR, since these signals are low true) of $\overline{\text{RDY}}$ and $\overline{\text{RDY}}_{\text{wcnt}}$ .
11	0	1	1	
11	1	0	1	
11	1	1	1	

## 7.5 Timing

Figure 7-5.  $\overline{STRB}$  and  $\overline{RDY}$  Timing



**Note:** Dotted lines emphasize the relationships between signals that is further explained in the accompanying text below.

Throughout this chapter, no distinction is made between global and local interface signals and between  $\overline{STRB0}$  and  $\overline{STRB1}$ , except for clarity.

As shown in Figure 7-5,  $\overline{STRB}$  changes on the falling edge of H1, and  $\overline{RDY}$  is sampled on the falling edge of H1. Throughout the other timing diagrams in this section, **the following general rules** apply to the logical timing of the parallel external interfaces:

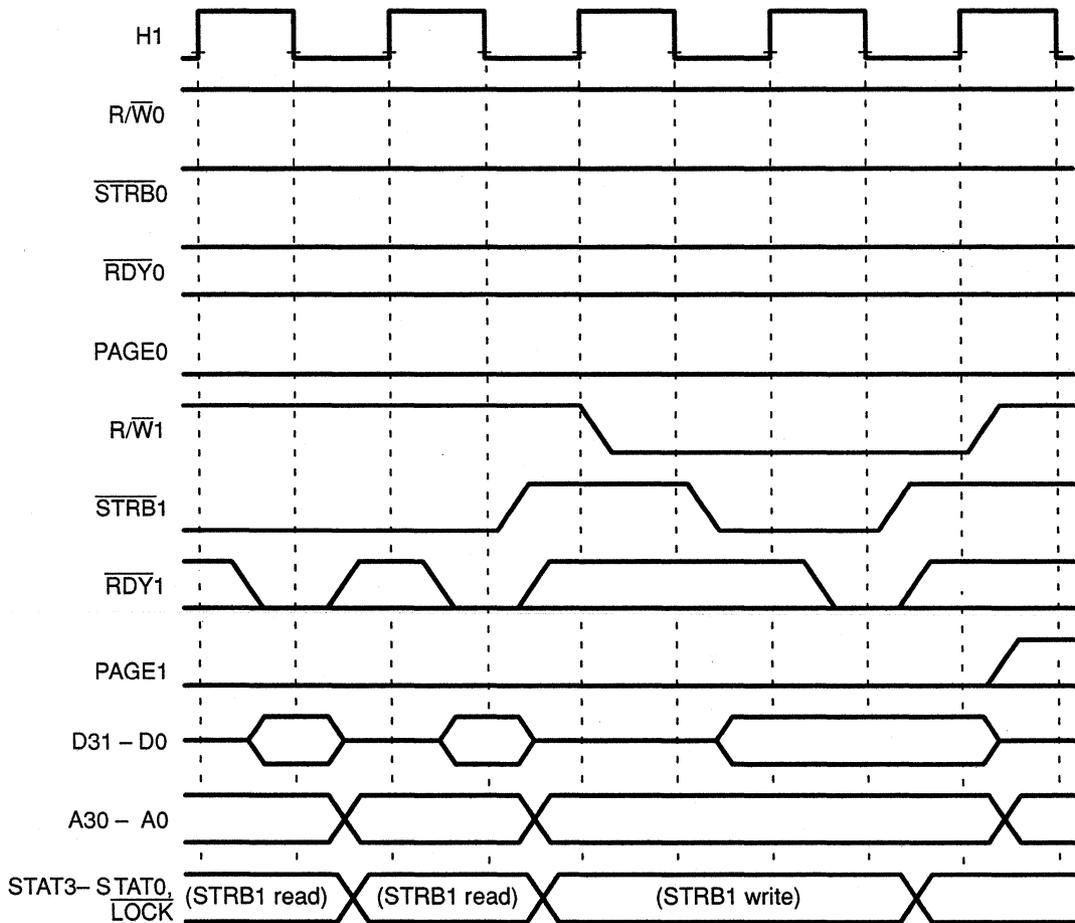
7

- 1) Changes of  $R/\overline{W}$  are always framed by  $\overline{STRB}$ .
- 2) A page boundary crossing for a particular  $\overline{STRB}$  results in the corresponding PAGE signal going high for one cycle.
- 3)  $R/\overline{W}$  transitions are always on an H1 rising.
- 4)  $\overline{STRB}$  transitions are always on an H1 falling.
- 5)  $\overline{RDY}$  is always sampled on an H1 falling.
- 6) On a read, data is always sampled on an H1 falling.
- 7) On a write, data is always driven out on H1 falling.
- 8) On a write, data is always stopped from being driven on H1 rising.
- 9) Following a read, the status, and PAGE signal change on H1 falling. The address changes on H1's falling edge.
- 10) Following a write, status and PAGE signals change on H1 falling; the address changes on H1 rising.

- 11) The fetch of an interrupt vector over an external interface is identified by the status signals for that interface (STAT or LSTAT) as a data read.
- 12) The interlocked operation status signals ( $\overline{\text{LOCK}}$  and  $\overline{\text{LLOCK}}$ ) have the same timing as the STAT and LSTAT status signals, respectively.
- 13) Any time PAGE goes high,  $\overline{\text{STRB}}$  goes high.

Figure 7-6 illustrates a read, read, write sequence. This figure assumes that all three accesses are to the same page and that they are  $\overline{\text{STRB1}}$  accesses. This timing diagram illustrates that back-to-back reads to the same page are single-cycle accesses. When the transition from a read to a write is done,  $\overline{\text{STRB}}$  goes high for one cycle in order to frame the R/W signal changing.

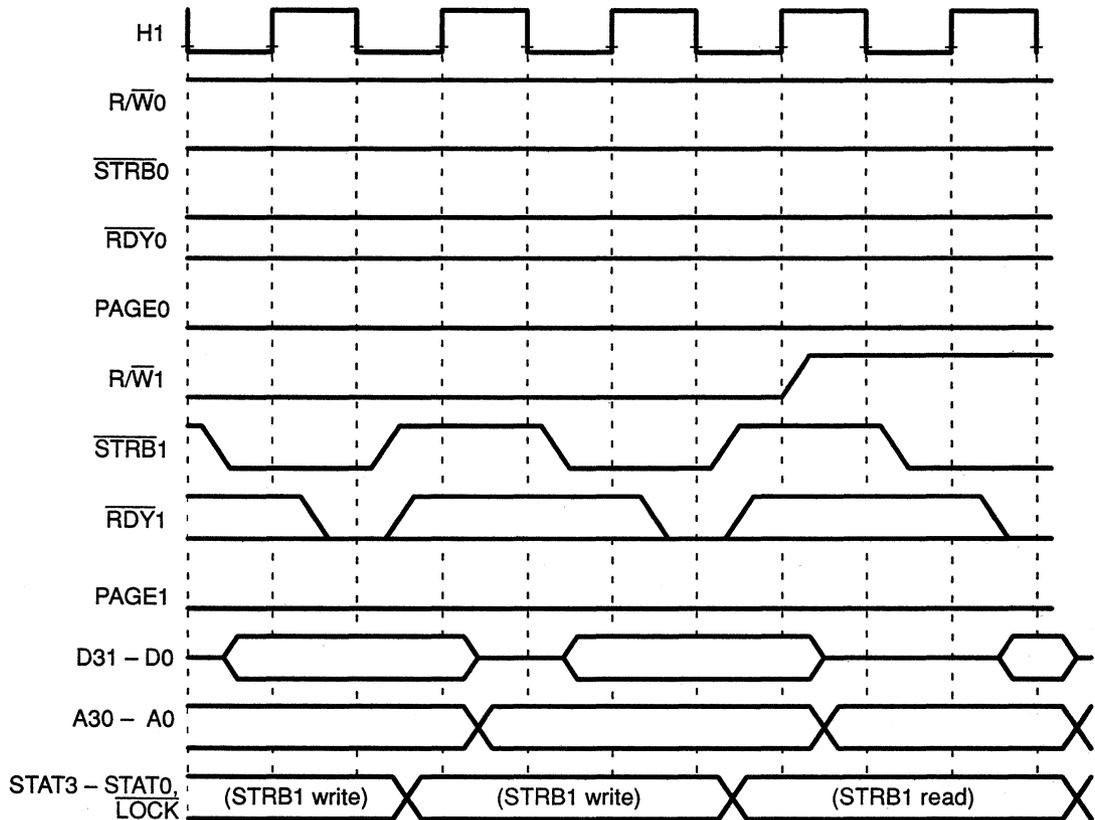
Figure 7-6. Read Same Page, Read Same Page, Write Same Page Sequence



7

Figure 7-7 shows that  $\overline{\text{STRB}}$  goes high between back-to-back writes. As in Figure 7-6,  $\overline{\text{STRB}}$  goes high between a write and a read, and it frames the  $\text{R}/\overline{\text{W}}$  transition.

Figure 7-7. Write Same Page, Write Same Page, Read Same Page Sequence



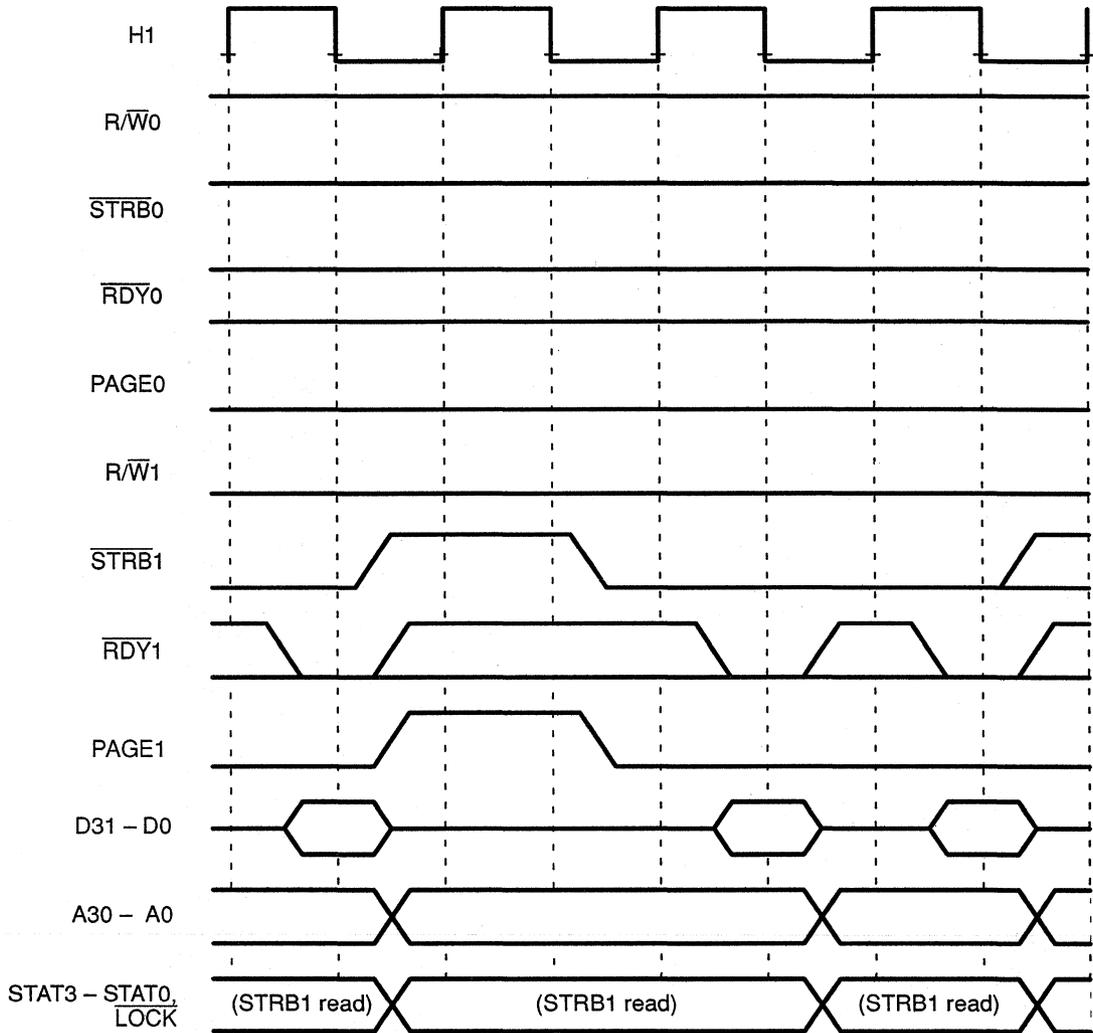
7

**Note: Strobe and Ready Further Defined**

Strobe and ready are discussed from the application viewpoint in Sections 13.3 (page 13-20) and 13.4 (page 13-27) respectively.

Figure 7-8 shows that going from one page to another on back-to-back reads causes an extra cycle to be inserted, and the transition is signaled by PAGE going high for one cycle. Also, STRB1 goes high for one cycle.

Figure 7-8. Read Same Page, Read Different Page, Read Same Page Sequence



7

Figure 7-9 shows that on back-to-back writes, when a page switch occurs, it is signaled with PAGE going high for one cycle.

Figure 7-9. Write Same Page, Write Different Page, Write Same Page Sequence

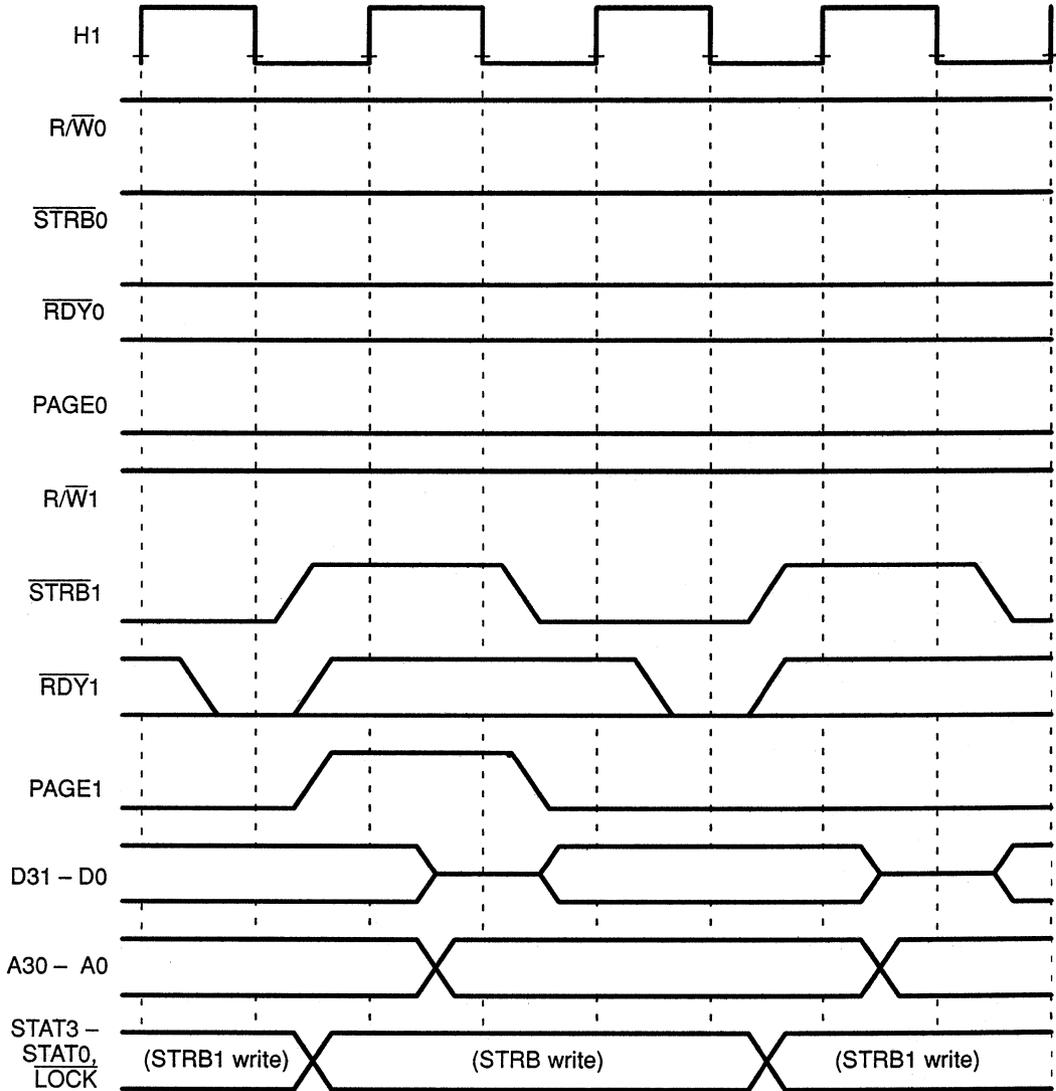
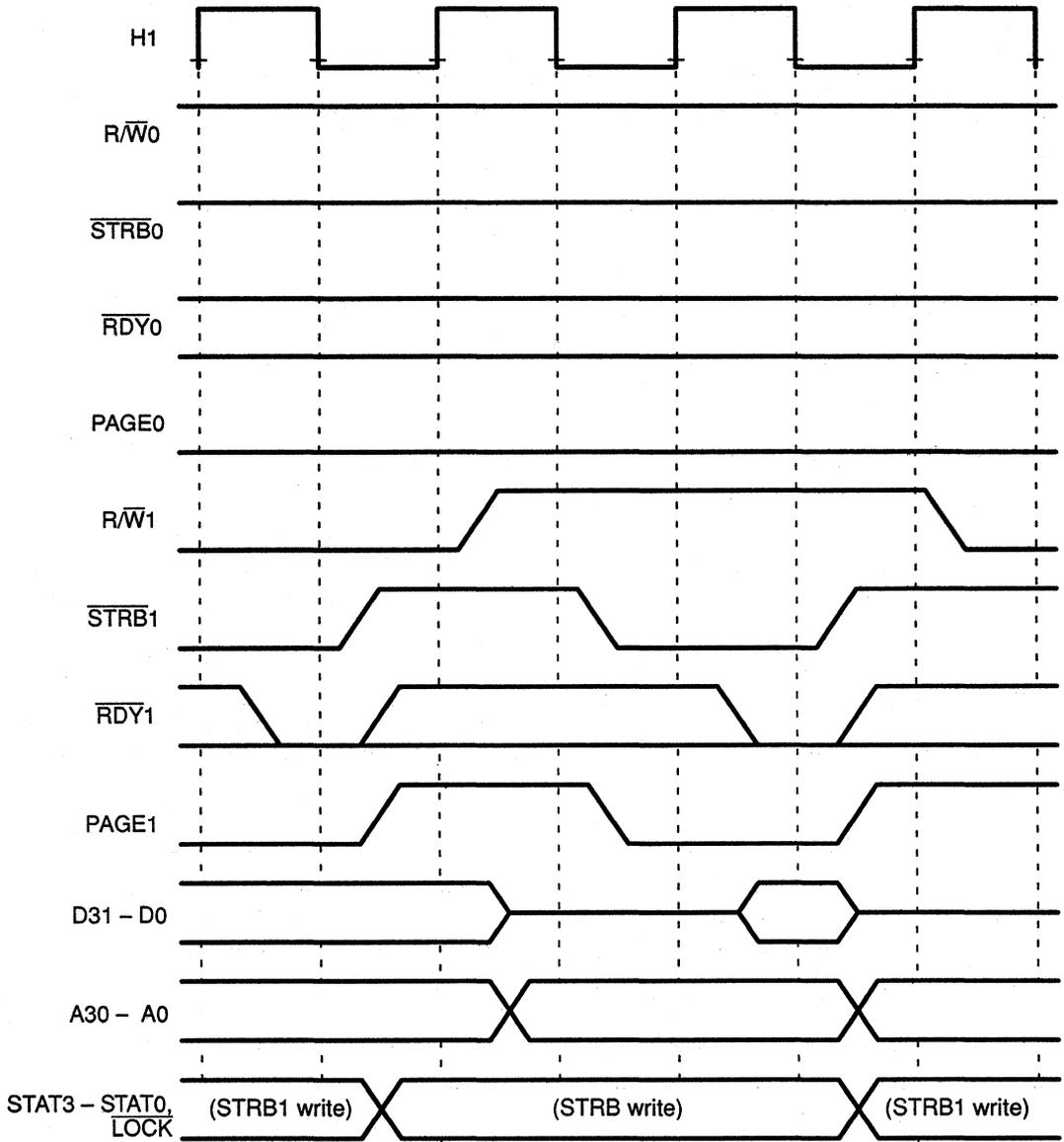


Figure 7-10. Write Same Page, Read Different Page, Write Different Page Sequence



7

Figure 7-11. Read Different Page, Read Different Page, Write Same Page Sequence

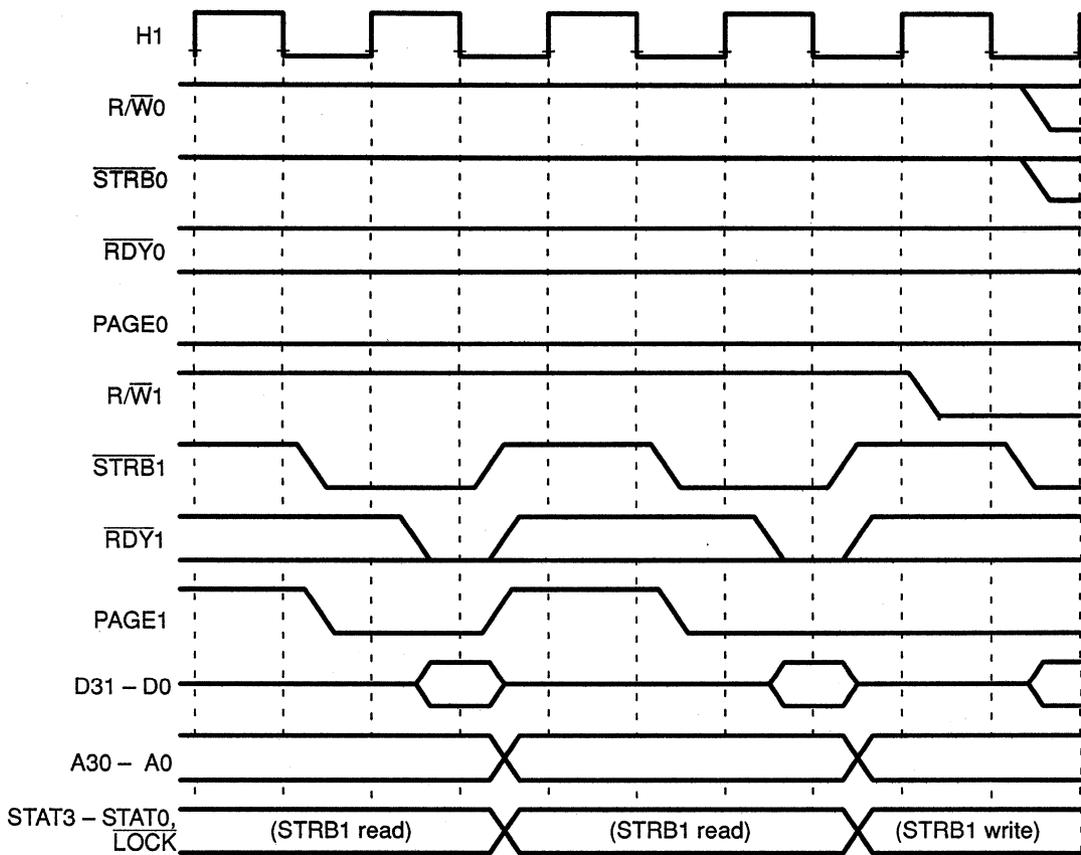
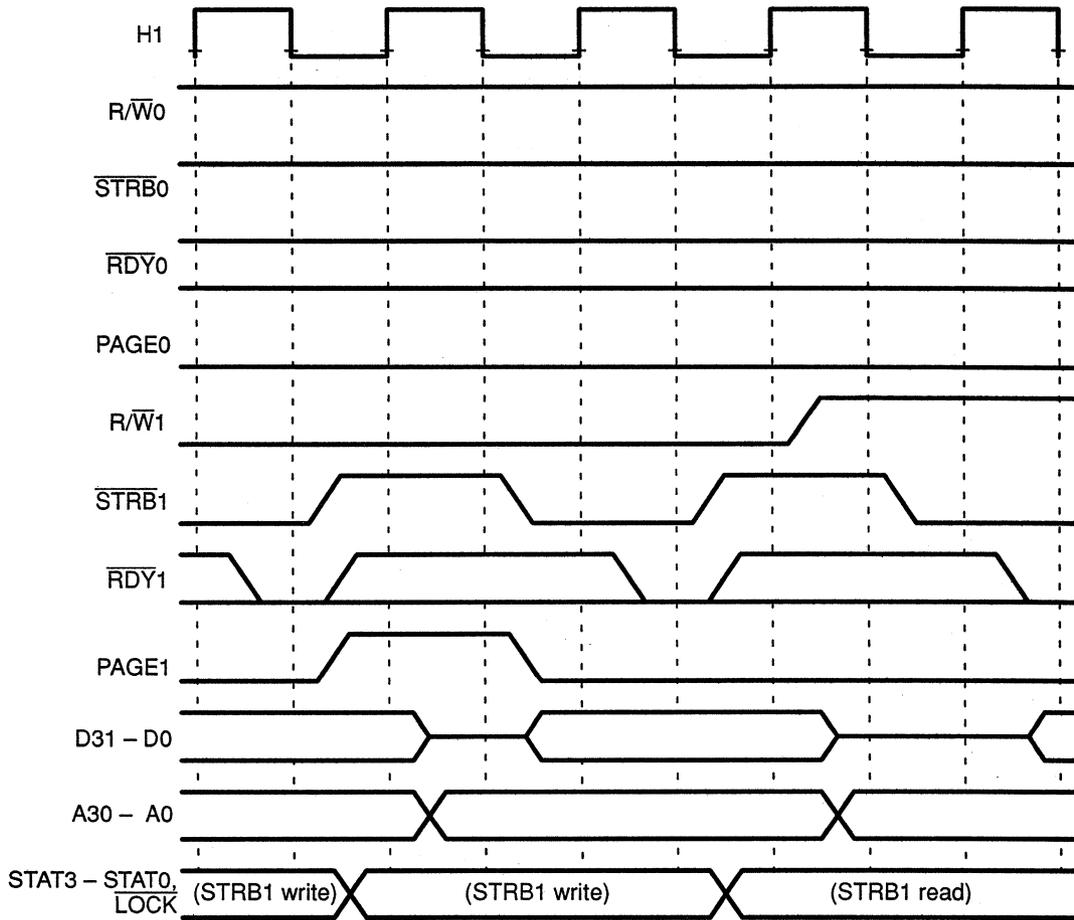


Figure 7-12. Write Different Page, Write Different Page, Read Same Page Sequence



7

Figure 7-13. Read Same Page, Write Different Page, Read Different Page Sequence

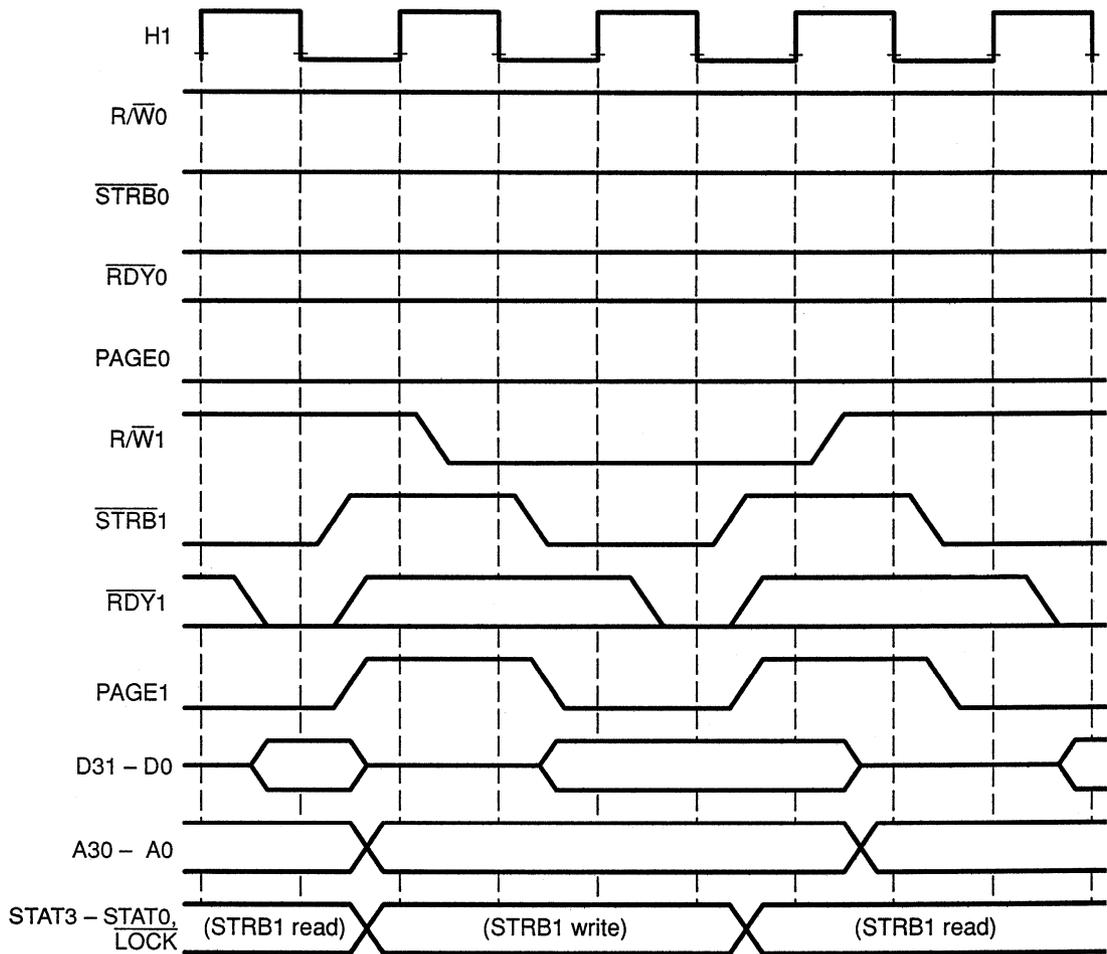
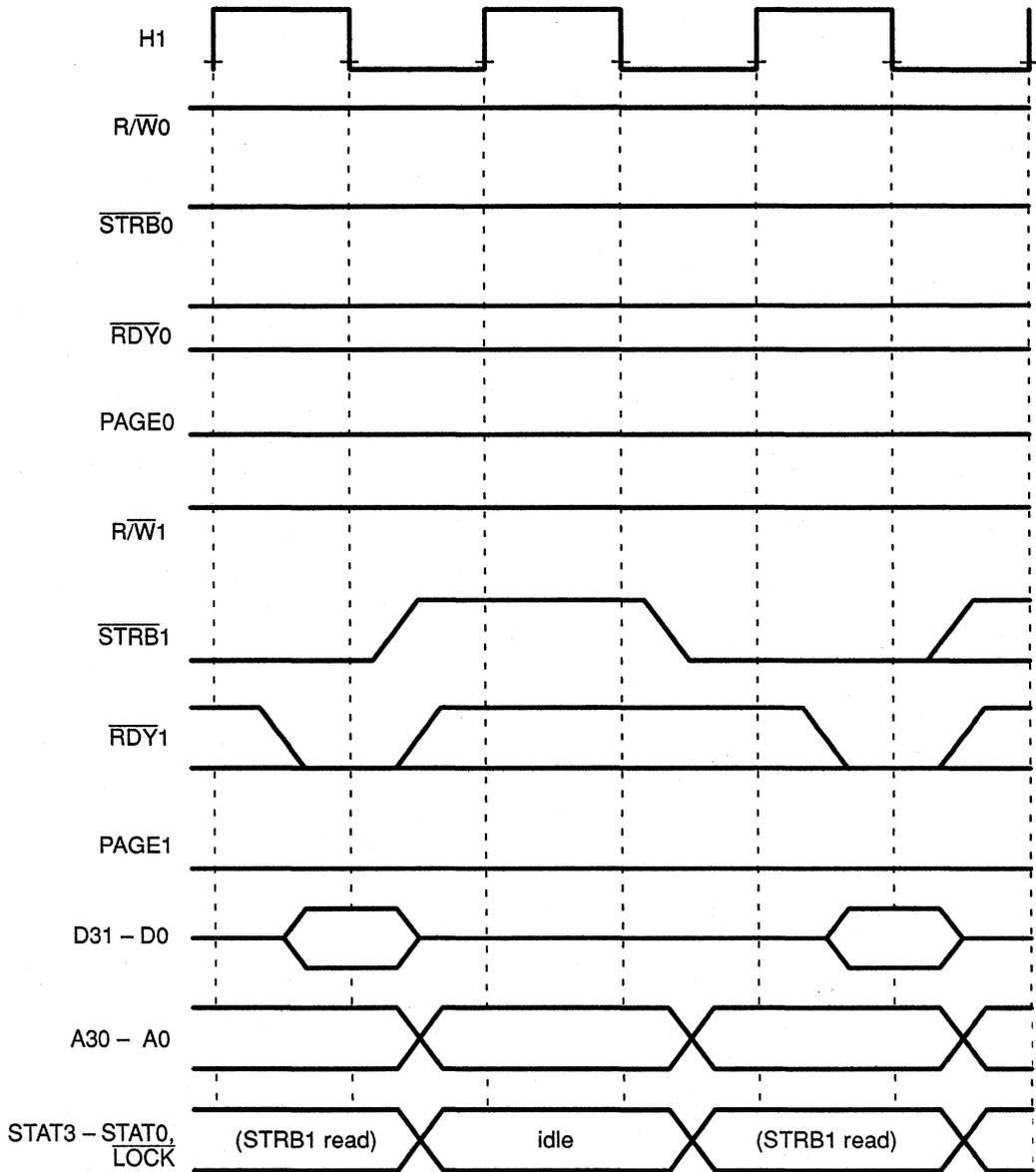


Figure 7-14 to Figure 7-18 illustrate the idle bus cycles. Idle bus cycle timing is similar to read cycle timing. The primary differences are that no data is read, STRB is held high, and RDY is ignored.

Figure 7-14. Read Same Page, Idle One Cycle, Read Same Page Sequence



7

Figure 7-15. Write Same Page, Idle One Cycle, Write Different Page Sequence

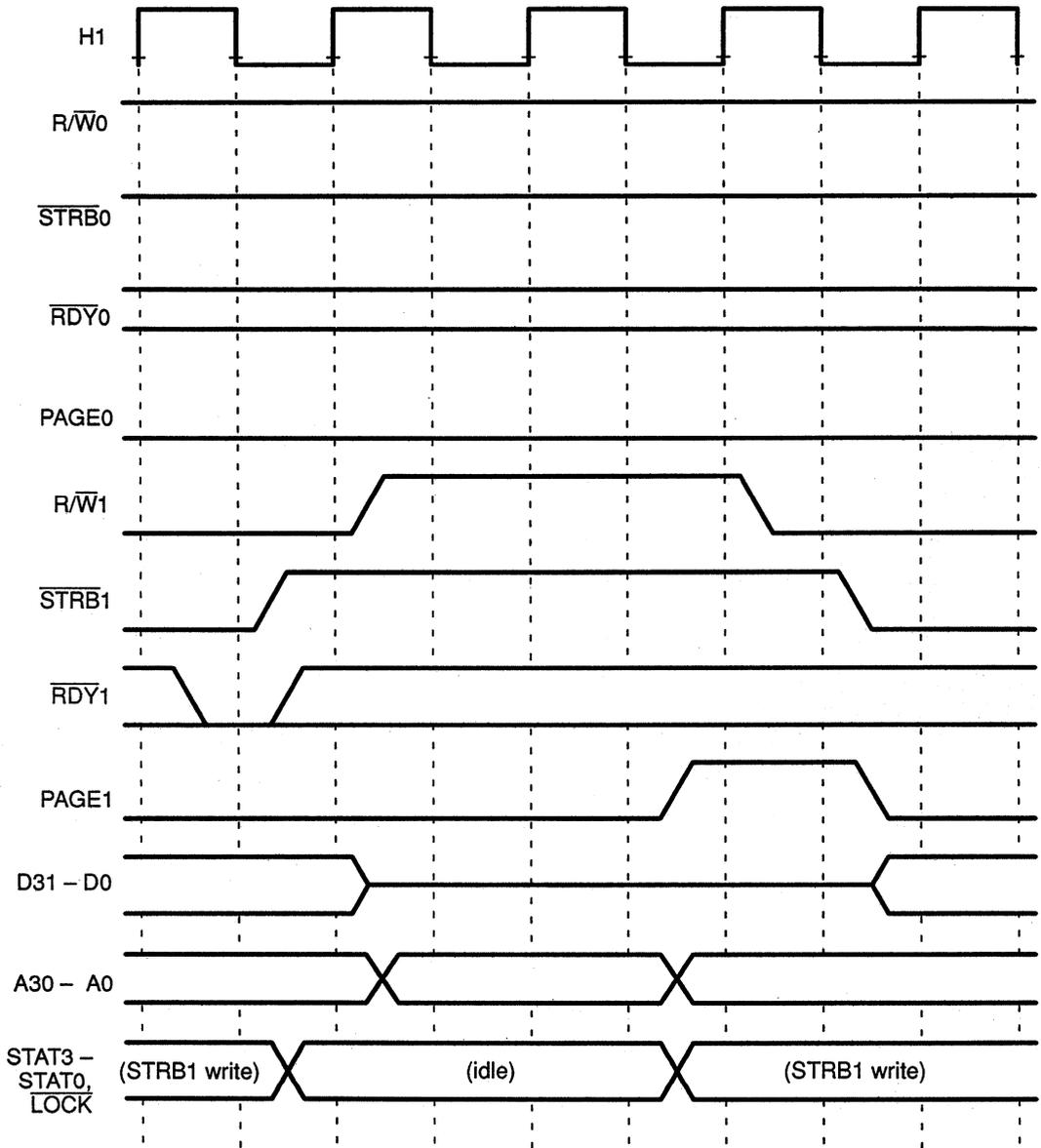
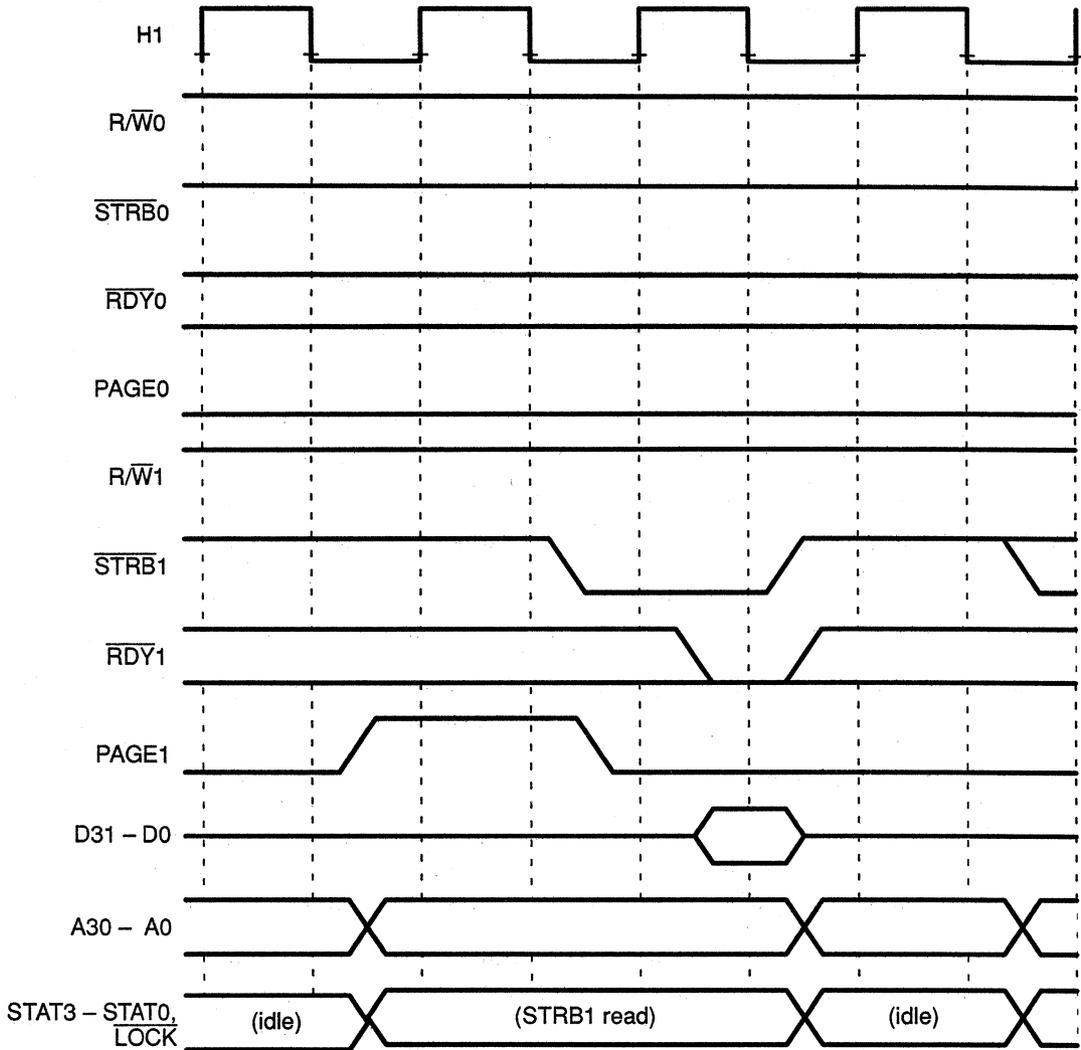


Figure 7-16. Idle, Read Different Page, Idle Sequence



7

Figure 7-17. Idle, Write Same Page, Idle Sequence

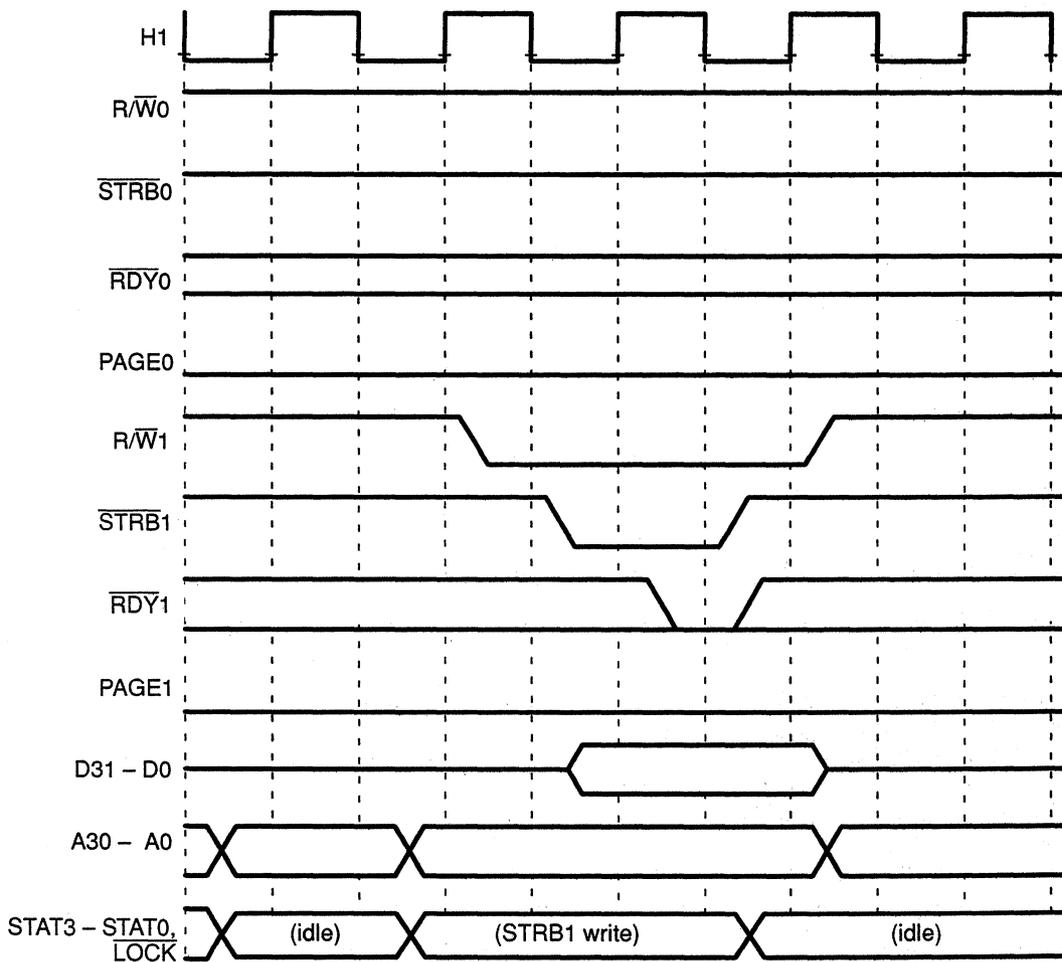
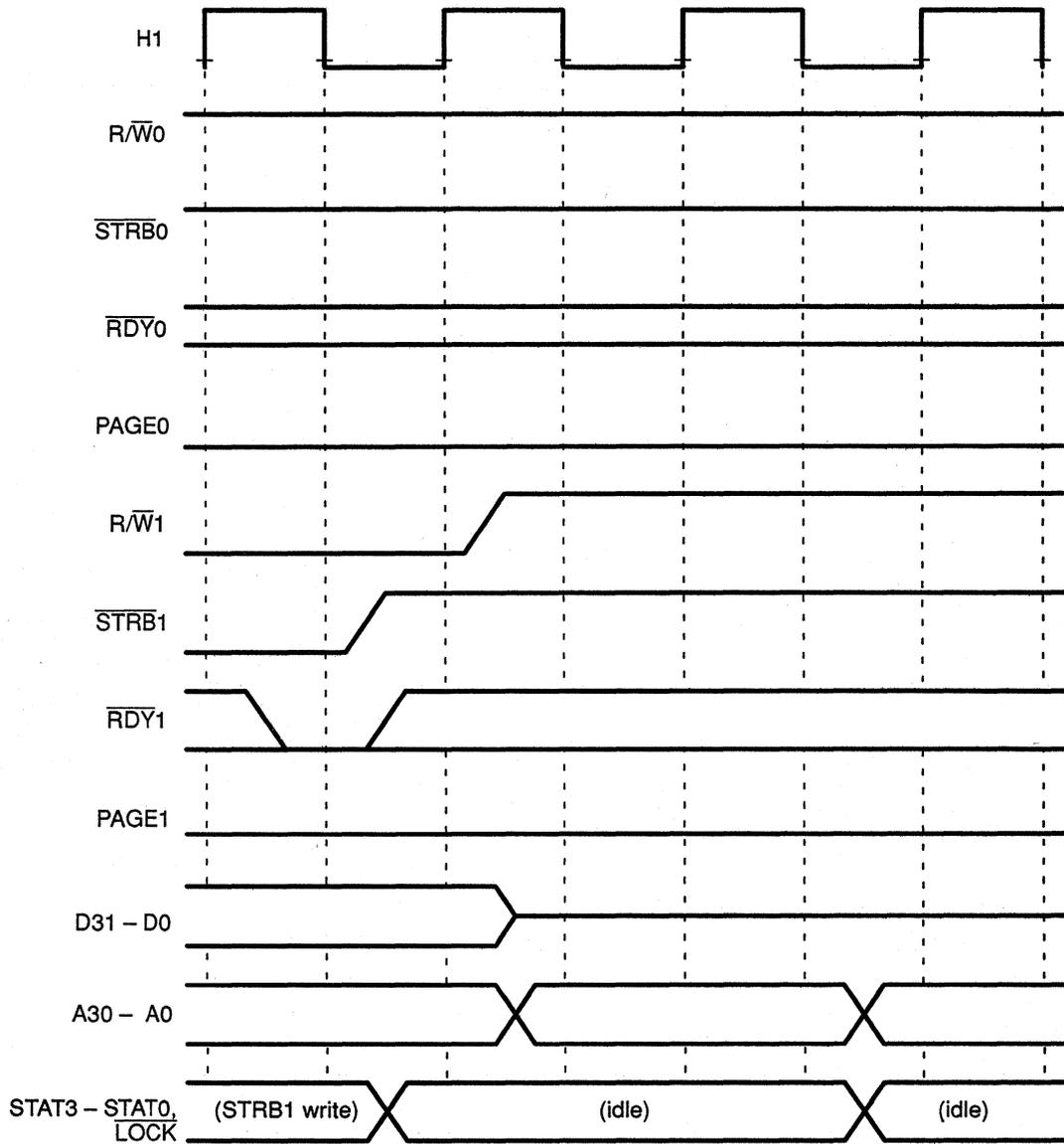


Figure 7-18. Write Different or Same Page, Idle, Idle Sequence



7

Figure 7–19 illustrates an  $\overline{\text{STRB1}}$  read followed by an  $\overline{\text{STRB0}}$  read when  $\text{STRB SWITCH} = 0$ . This mode allows the reads to be back to back, with no cycles inserted between the reads when the back-to-back reads are activating different strobes.

Figure 7–19. Read Same Page on  $\overline{\text{STRB1}}$ , Read Same Page on  $\overline{\text{STRB0}}$ , Read Same Page on  $\overline{\text{STRB1}}$  Sequence When  $\text{STRB SWITCH} = 0$

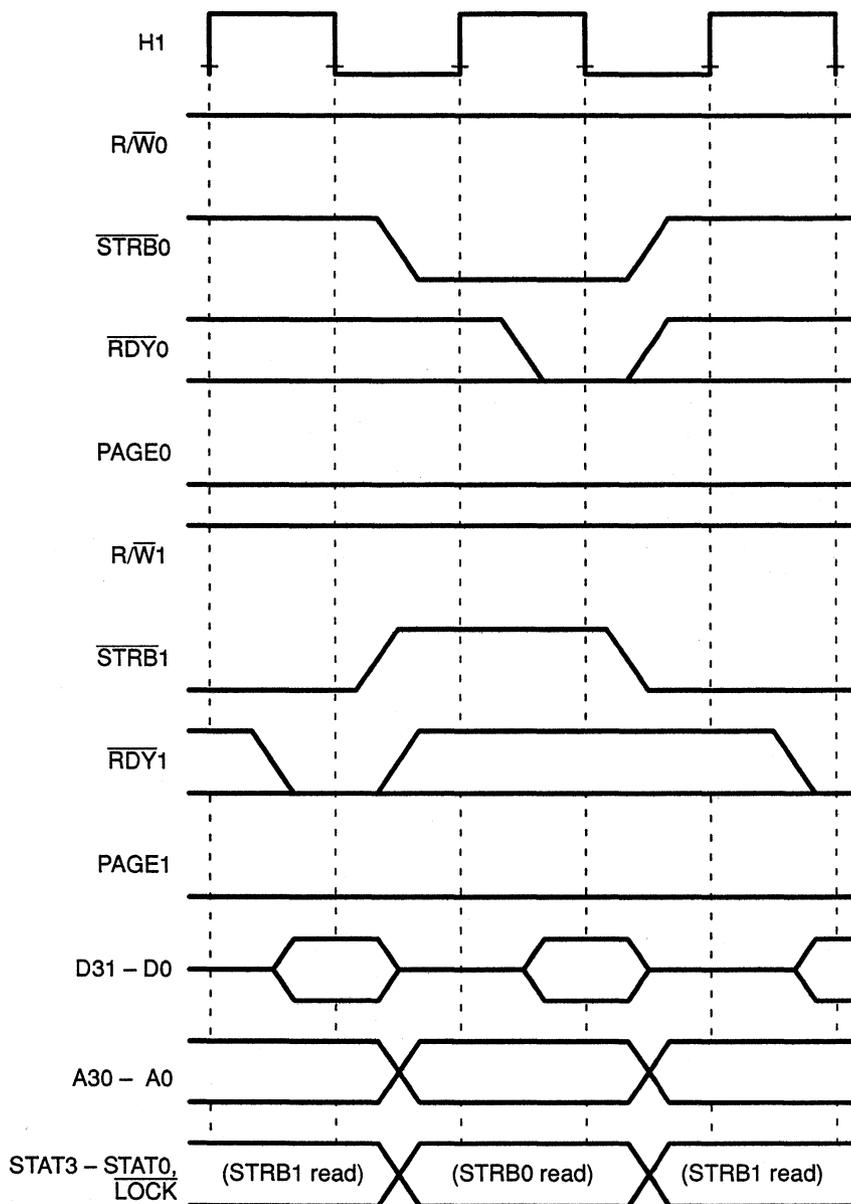
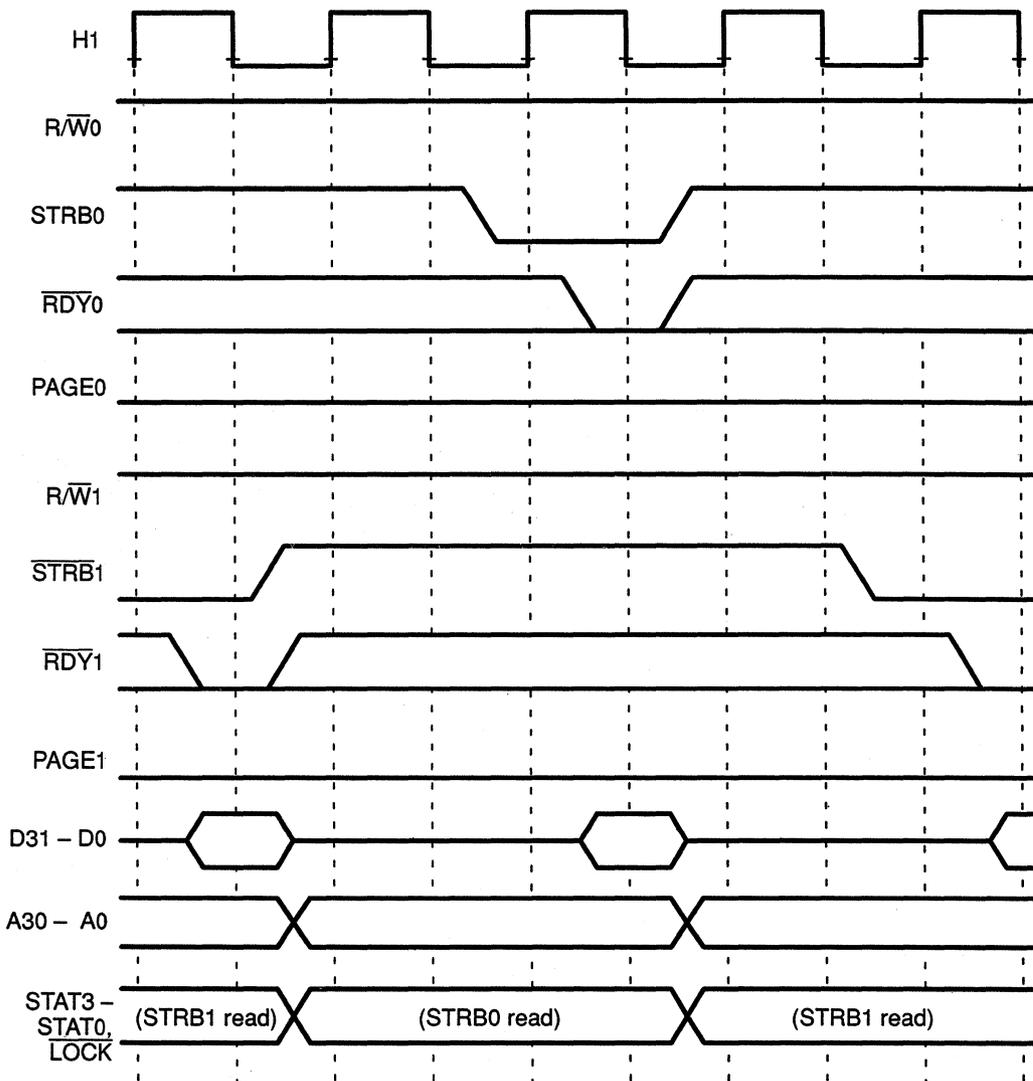


Figure 7–20 illustrates an  $\overline{\text{STRB1}}$  read followed by an  $\overline{\text{STRB0}}$  read when  $\text{STRB SWITCH} = 1$ . In this mode, a cycle is inserted between back-to-back reads that activate different strobes. If your system memory configuration is such that bus conflicts can occur during back-to-back reads on different strobes, this mode provides one cycle between these strobe transitions to avoid the bus conflicts.

Figure 7–20. Read Same Page on  $\overline{\text{STRB1}}$ , Read Same Page on  $\overline{\text{STRB0}}$ , Read Same Page on  $\overline{\text{STRB1}}$  Sequence When  $\text{STRB SWITCH} = 1$



7

Figure 7-21 is similar to Figure 7-19 except that the second read using  $\overline{\text{STRB1}}$  is to a different page than the first read (using  $\overline{\text{STRB1}}$ ).

Figure 7-21. Read Same Page on  $\overline{\text{STRB1}}$ , Read Same Page on  $\overline{\text{STRB0}}$ , Read Different Page on  $\overline{\text{STRB1}}$  Sequence When  $\text{STRB SWITCH} = 0$

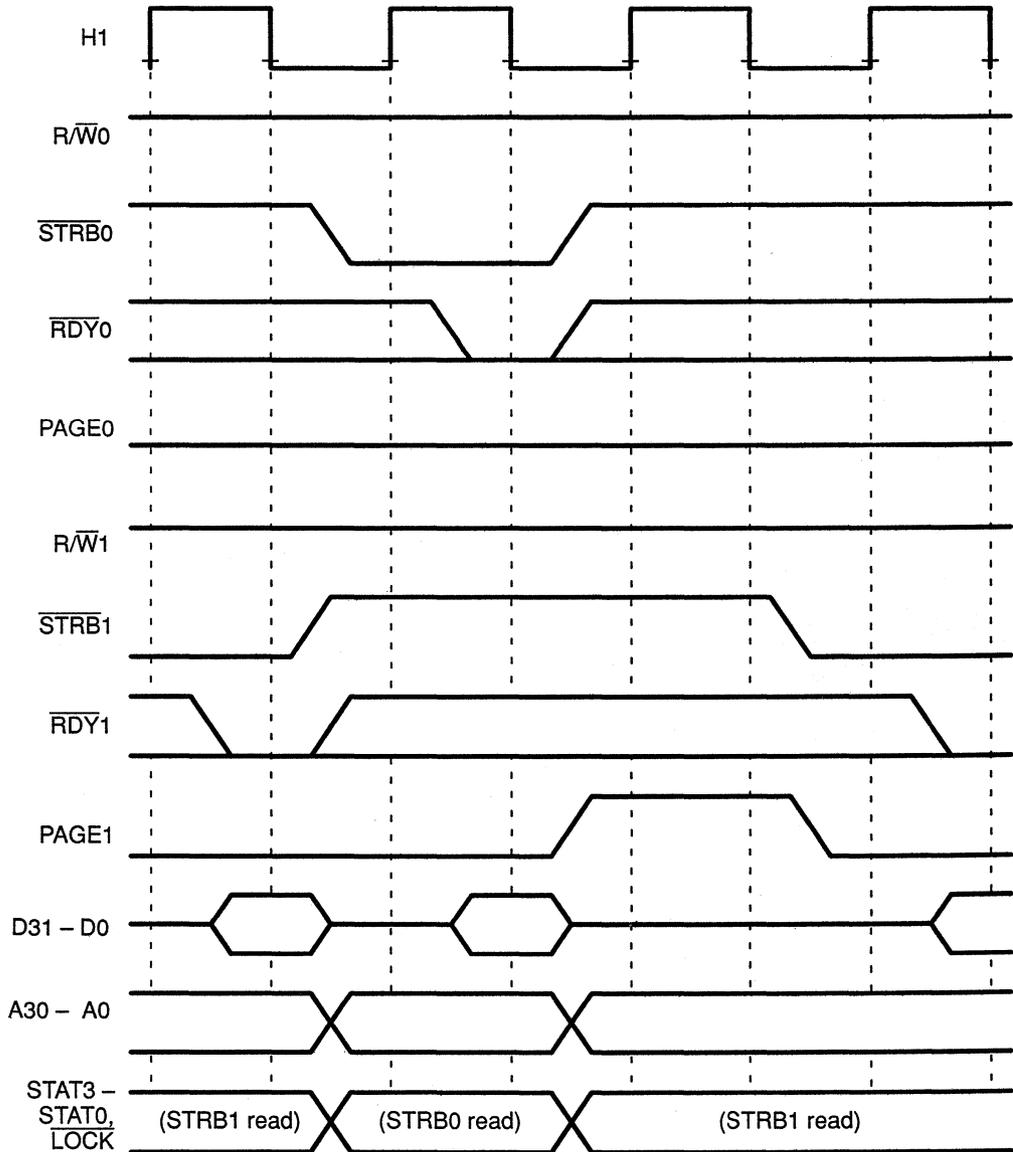
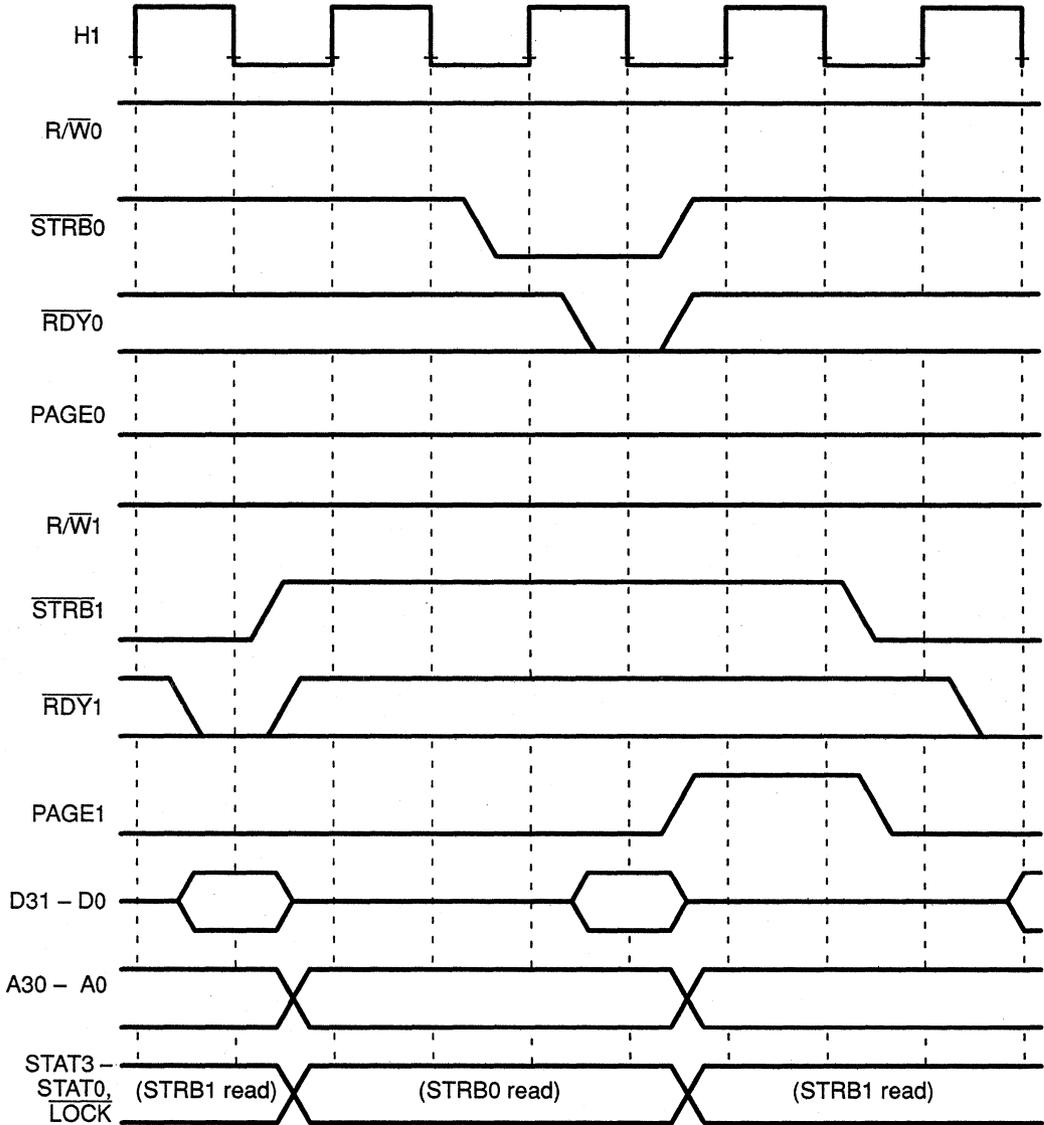


Figure 7-22 is similar to Figure 7-20 except that the second read using  $\overline{\text{STRB1}}$  is to a different page than the first read (using  $\overline{\text{STRB1}}$ ).

Figure 7-22. Read Same Page on  $\overline{\text{STRB1}}$ , Read Same Page on  $\overline{\text{STRB0}}$ , Read Different Page on  $\overline{\text{STRB1}}$  Sequence When STRB SWITCH = 1



7

Figure 7-23. Write Same Page on  $\overline{\text{STRB1}}$ , Write Same Page on  $\overline{\text{STRB0}}$ , Read Same Page on  $\overline{\text{STRB1}}$  Sequence

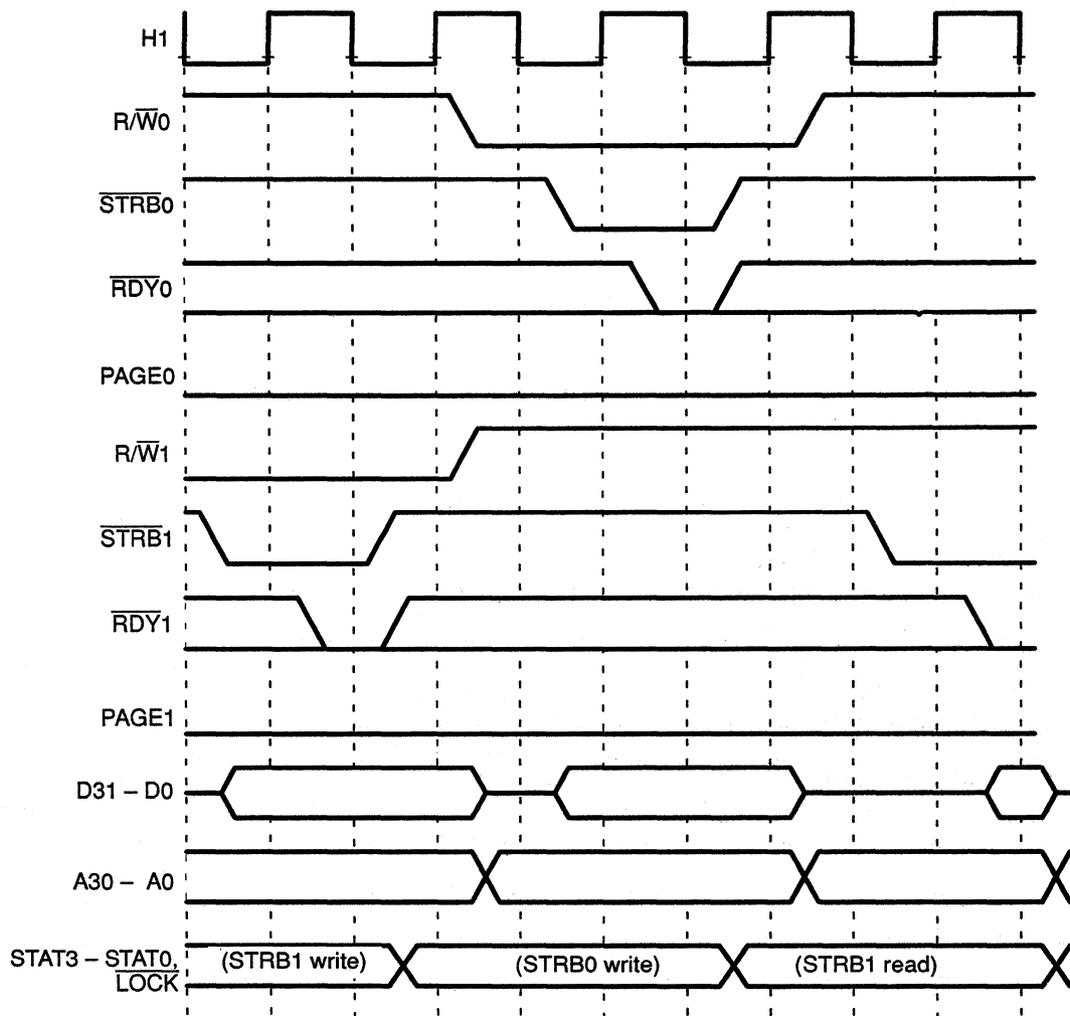
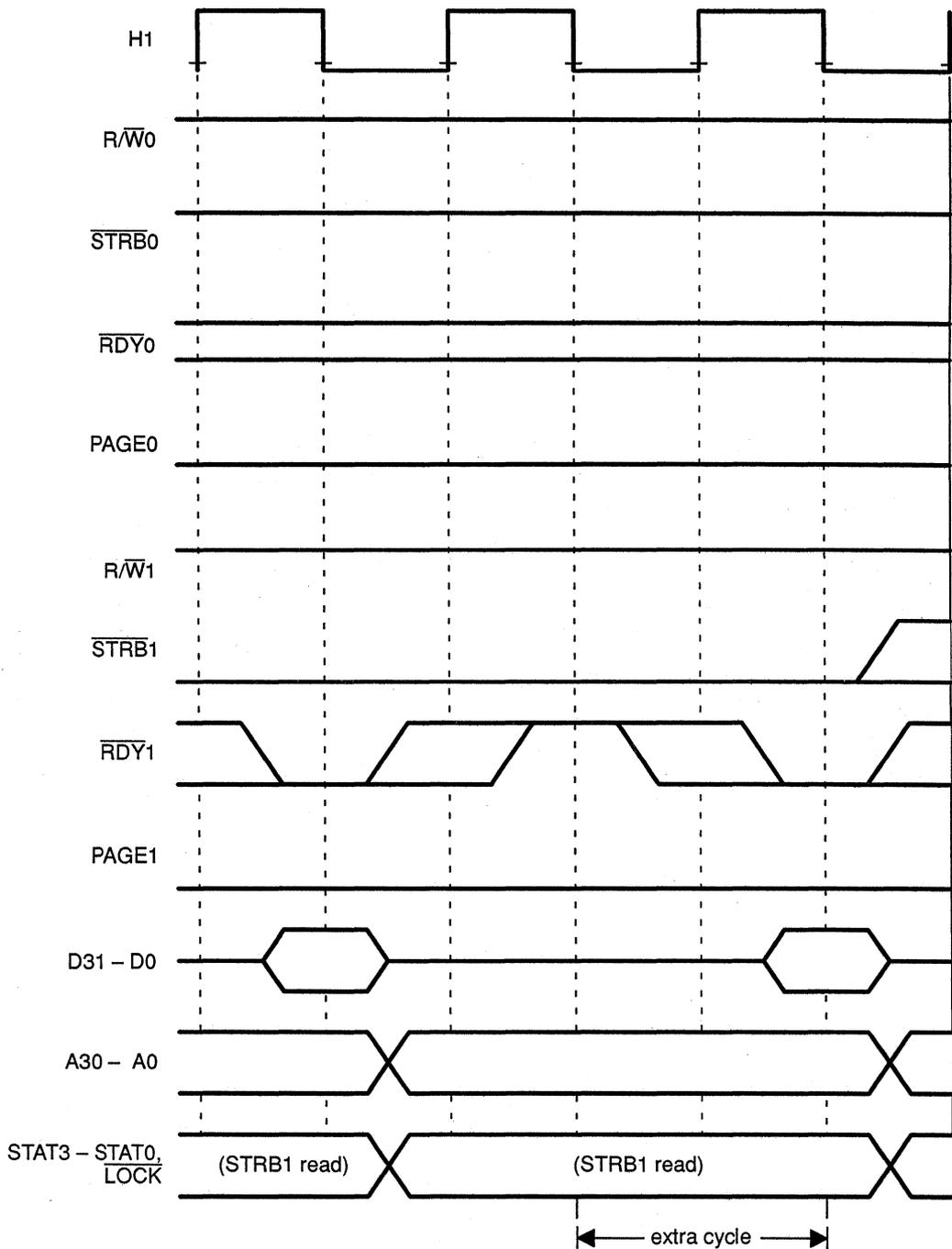
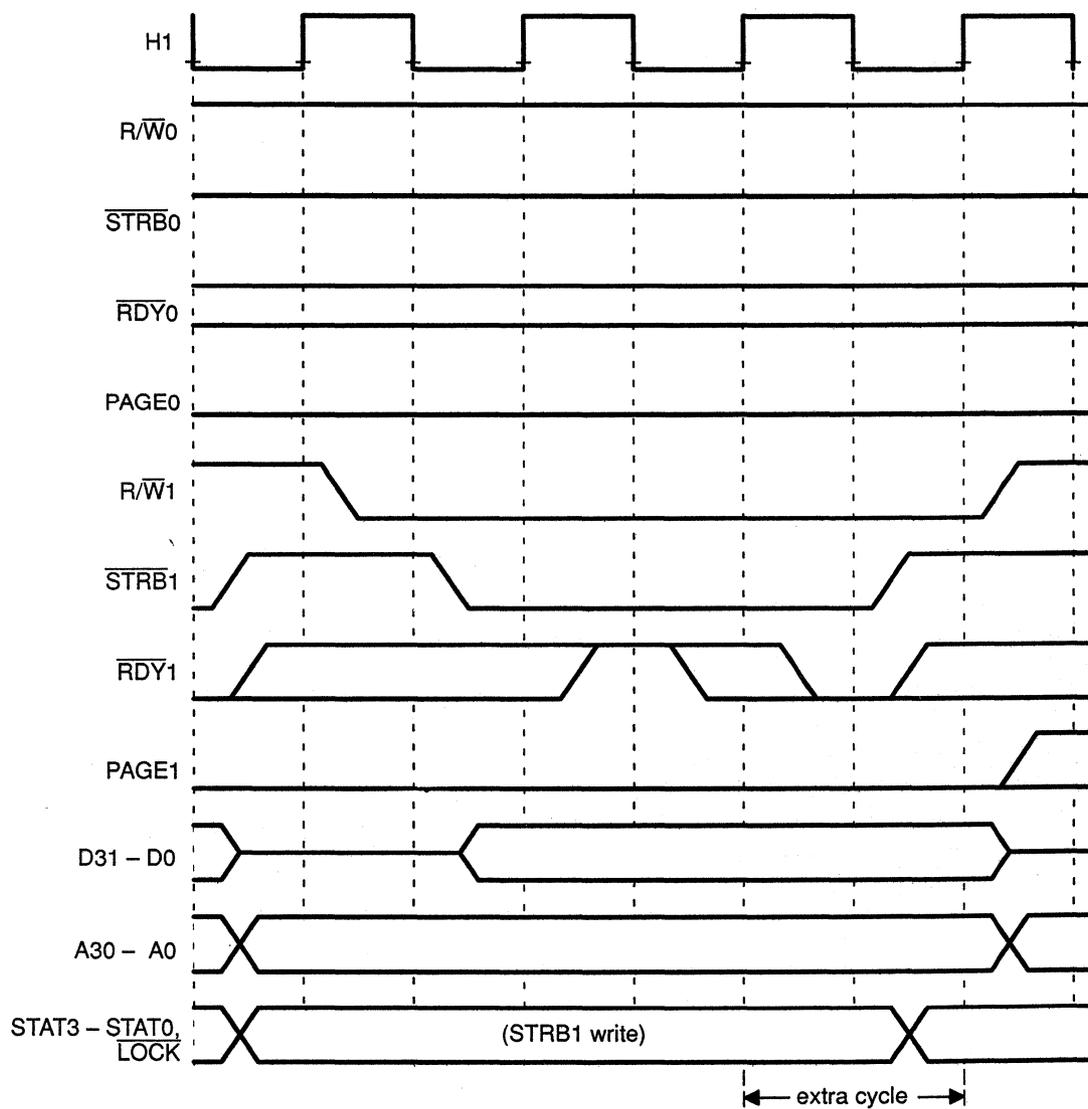


Figure 7-24. Read With One Wait State



7

Figure 7-25. Write With One Wait State



## 7.6 Using Enabled Signals to Control Signal Group

Figure 7–26. Using Enabled Signals to Put Signal Groups in a High-Impedance State

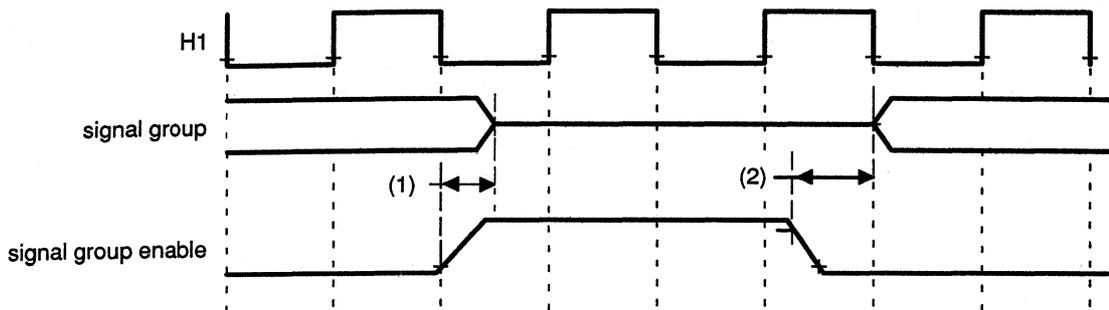


Figure 7–26 shows the use of an enable signal to control the corresponding signal group. For example, signal  $\overline{DE}$  controls the global external-interface signals D31–D0. The enable signals are unsynchronized inputs that turn off the corresponding output buffers. Some time period (shown by period (1) in Figure 7–26) after the enable signal goes high, the corresponding signal group goes into a high-impedance state. Then, some time period after the enable signal goes low (period (2) in Figure 7–26), the signal group comes out of a high-impedance state. Of course, if the signal group is already in a high-impedance state before the enable signal goes high, the group will come out of the high-impedance state (when the enable signal goes low) only if the signal group is in a state requiring it to do so. For example, a data bus that was not being driven will be driven after being enabled if an access is pending for the data bus.

7

**If you intend to use internally generated wait states, be certain that nothing inappropriate occurs when a bus is disabled.** This is because it is possible to have a bus in a high-impedance state and with internally generated wait states. In this case, data that is **written** will not be seen externally, and data that is **read** will be whatever value is sampled on the high-impedance bus.

## 7.7 Interlocked-Instructions Definition and Bus Timing

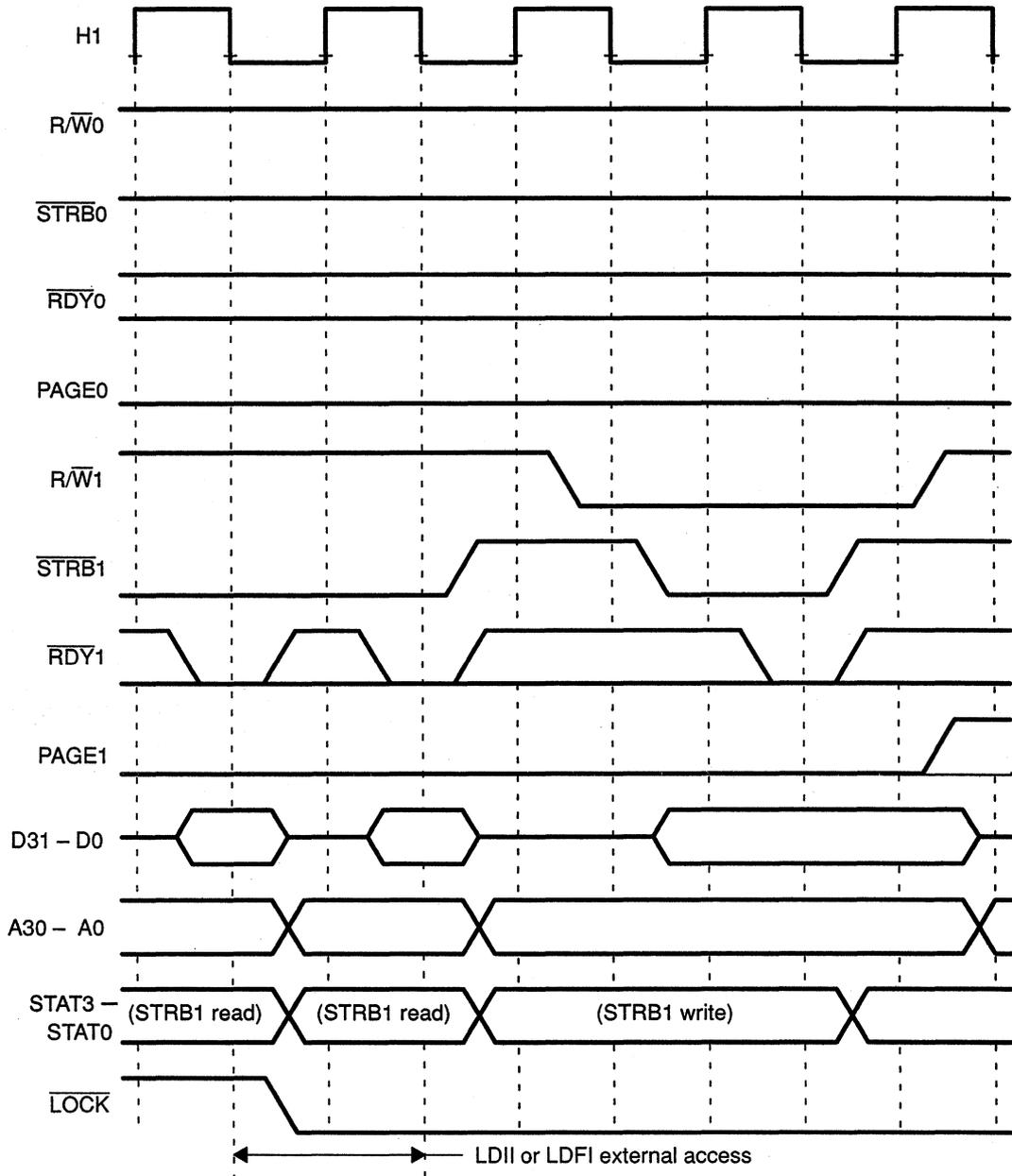
The  $\overline{\text{LOCK}}$  and  $\overline{\text{LLOCK}}$  bus-lock signals are manipulated by the interlocked instructions LDII, LDFI, STII, STFI, and SIGI. As noted, the timing of the  $\overline{\text{LOCK}}$  and  $\overline{\text{LLOCK}}$  pins is the same as pins STAT(3 — 0) and LSTAT(3 — 0). Instructions LDII, LDFI, STII, STFI, and SIGI manipulate the bus-lock signals *only* when an external memory access is made.

Except for the manipulation of the bus-locked signals, the LDII (Load Integer Interlocked) and LDFI (Load Floating Point Interlocked) instructions are like (in all ways) the comparable LDI (Load Integer) and LDF (Load Floating Point) in terms of the operation performed and the bus operation. LDII and LDFI perform as follows:

- 1) The read cycle is begun, and the appropriate bus-lock signal is placed in the active-low state.
- 2) The read cycle is extended until the appropriate ready signal is active.
- 3) Throughout the read cycle and to its conclusion, the bus-lock signal is kept in an active-low state until modified by a subsequent STII, STFI, or SIGI instruction.

Figure 7-27 is an example of an LDII or LDFI external access.

Figure 7-27. LDII or LDFI External Access



7

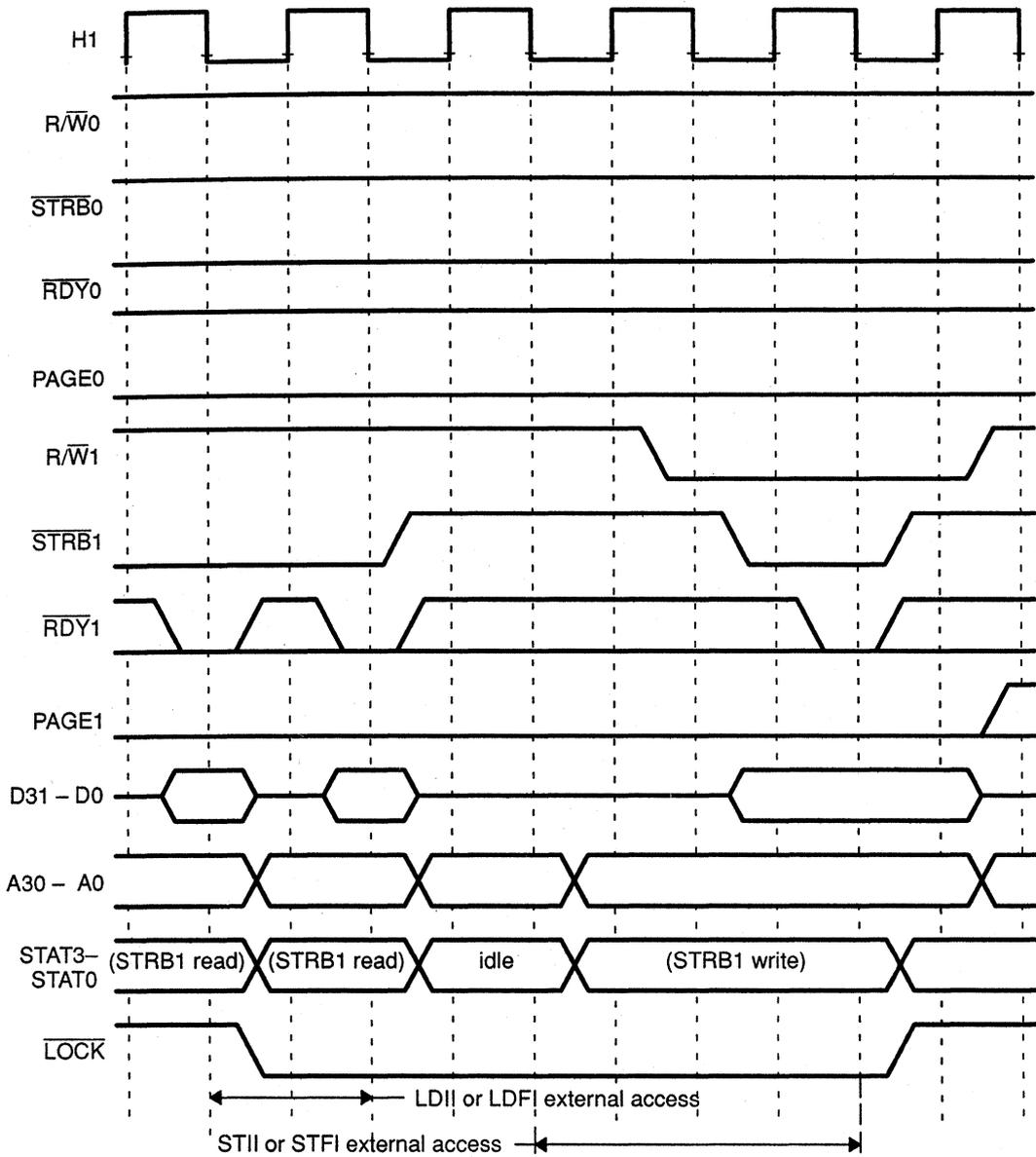
Except for manipulation of the bus-locked signals, the STII (Store Integer Interlocked) and STFI (Store Floating Point, Interlocked) instructions are the same as the comparable STI (Store Integer) and STF (Store Floating Point) in terms of execution and bus operation. STII and STFI operate as follows:

- 1) The store cycle is begun, and the appropriate bus-lock signal is kept in its current state. In most cases, the interlocked store is preceded by an interlocked load, and the bus-lock signal is kept low. Otherwise, the bus-lock signal is high, and the interlocked store looks like a not-interlocked store.
- 2) The store cycle is extended until the appropriate ready signal is active.
- 3) When the corresponding  $\overline{\text{STRB}}$  goes high at the end of the store cycle (the corresponding STAT(0–3) also changes at this time), the corresponding bus-lock signal also goes high.

An STII or STFI instruction to internal memory has no effect on the bus-lock signals.

Figure 7–28 is an example of an STII or STFI external access following the previous interlocked load (shown in Figure 7–27) and an idle cycle. This is the timing for an interlocked load/interlocked store sequence.

Figure 7-28. STII or STFI External Access



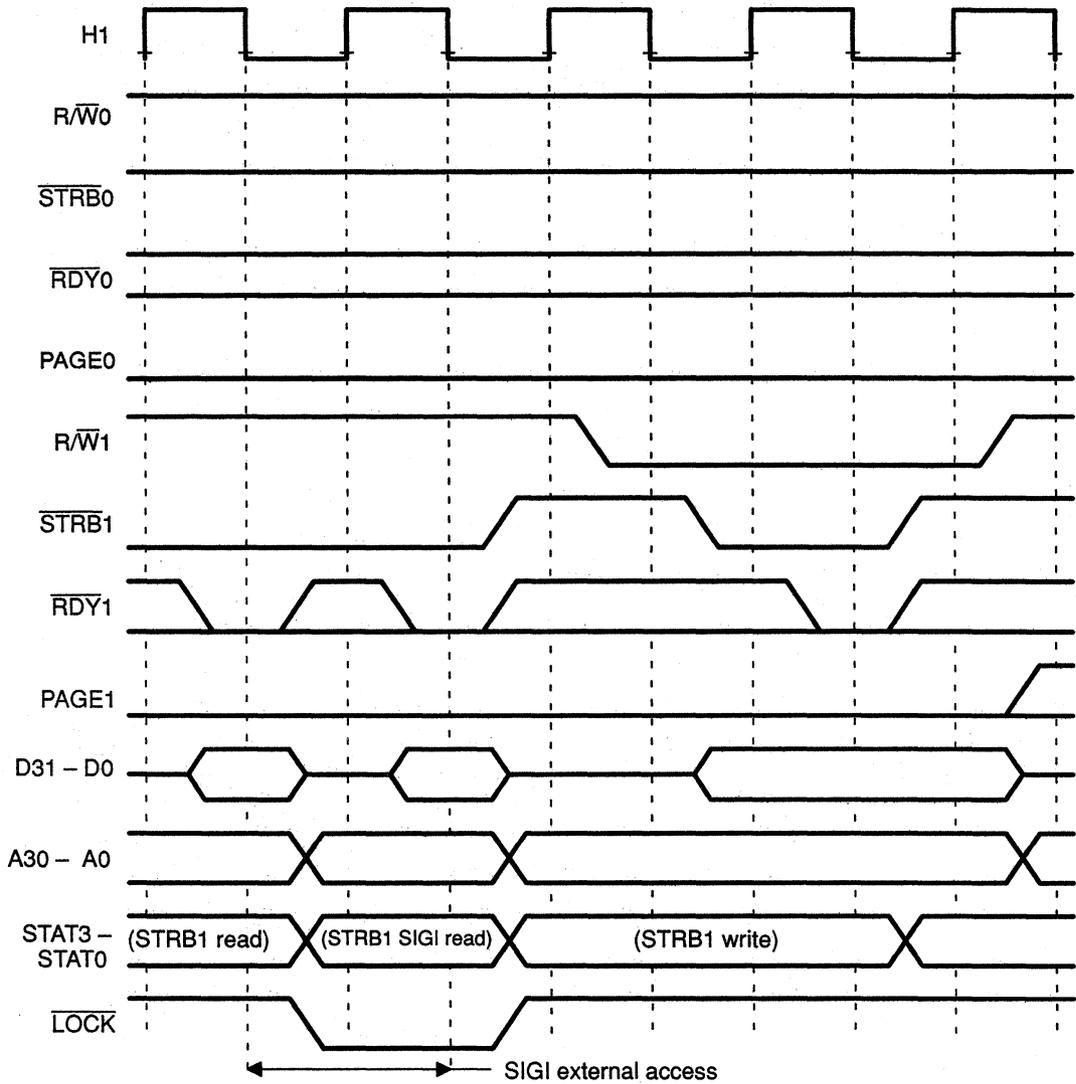
7

The SIGI instruction (signal interlocked) is similar to the LDII and LDIF instructions. The SIGI functions as follows:

- 1) The read cycle is begun, and the appropriate bus-lock signal is placed in the active-low state.
- 2) The read cycle is extended until the appropriate ready signal is active.
- 3) When the read operation is complete, the bus-lock signal is brought high with the same timing as the status signals changing.

Figure 7–29 is an example of a SIGI external access.

Figure 7-29. SIGI External Access Timing

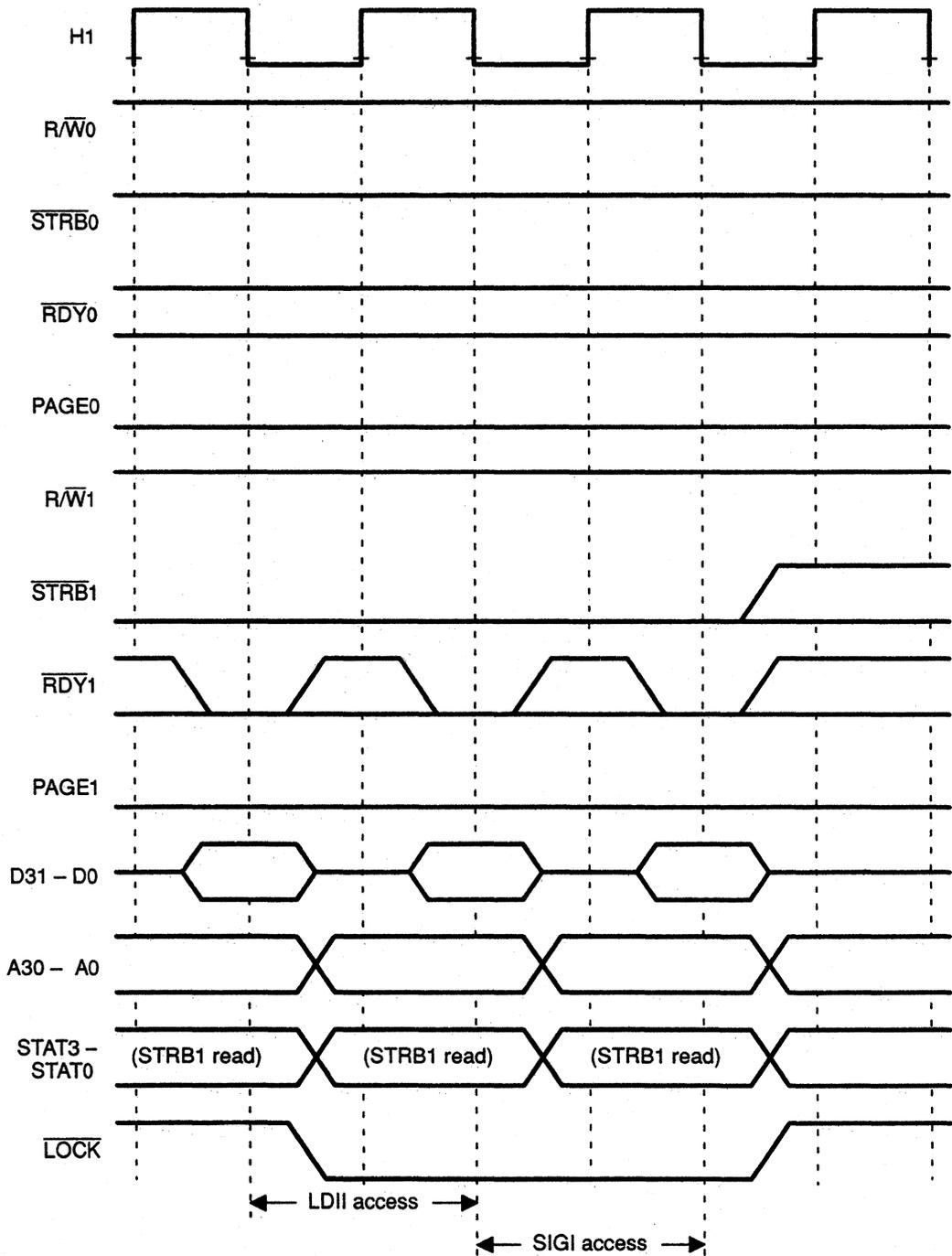


7

The SIGI instruction can be used in a variety of ways. In some applications, you may wish to externally modify semaphores, perhaps with special-purpose logic. If so, SIGI can be used to perform a single-cycle interlocked access of the semaphore. The SIGI instruction can also be used simply to perform an external read and to signal that a particular point in your code has been reached.

Figure 7–30 illustrates timing for SIGI if the  $\overline{\text{LOCK}}$  signal is already low. This could happen when a SIGI follows an LDII instruction. Since  $\overline{\text{LOCK}}$  is already low, the only effect SIGI has on  $\overline{\text{LOCK}}$  is to bring it high.

Figure 7-30. SIGI When  $\overline{\text{LOCK}}$  Is Already Low



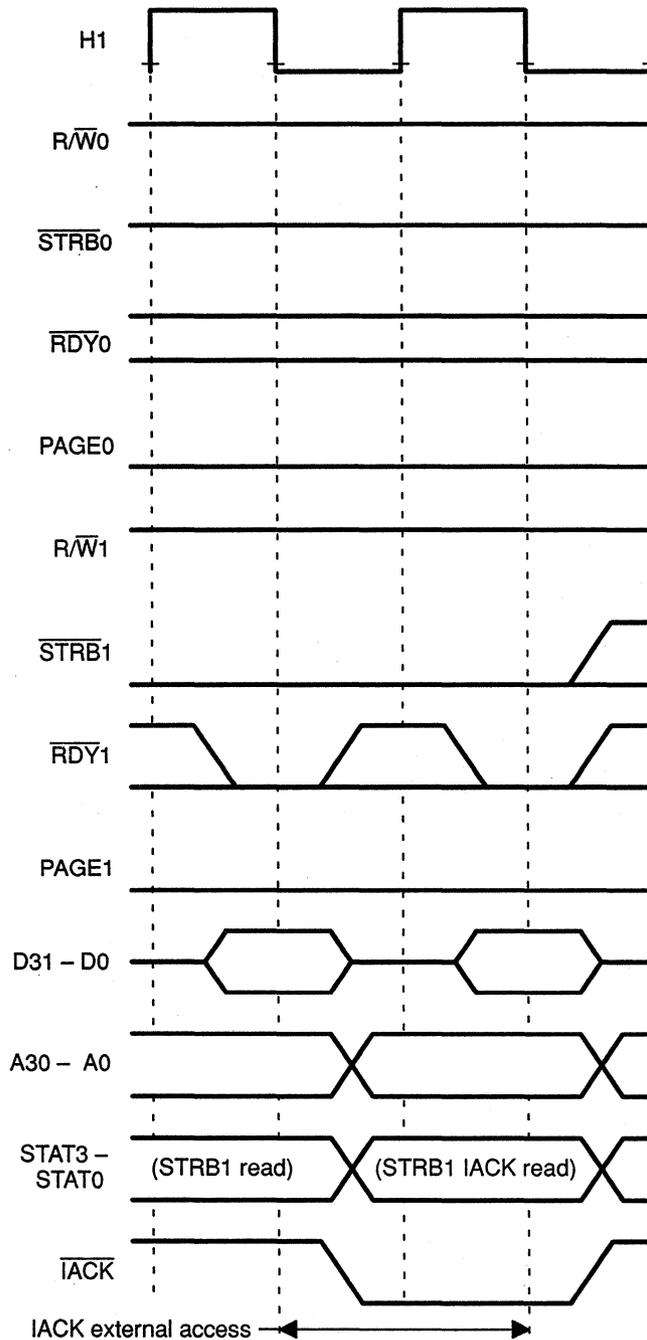
7

## 7.8 $\overline{IACK}$ Timing

The  $\overline{IACK}$  pin is affected by the IACK (interrupt acknowledge) instruction. The timing of the pin is similar to that of the  $\overline{LOCK}$  pin when used by the SIGI instruction. In all respects (timing, extension with wait states, etc.) the  $\overline{IACK}$  behaves like a  $\overline{LOCK}$  or STAT signal. The only difference is that there is only one  $\overline{IACK}$  pin.

The timing for the  $\overline{IACK}$  pin is shown in Figure 7-31. Like the interlocked instructions, the IACK instruction affects  $\overline{IACK}$  *only* for an external access.

Figure 7-31. IACK Timing



7

## Communication Ports

---

---

---

---

This chapter provides technical information for the communication ports of the TMS320C40 digital signal processor (DSP). This chapter is divided into the following major sections:

Section	Page
8.1 Introduction .....	8-2
8.2 Communication Port Features .....	8-3
8.3 Operational Overview .....	8-5
8.4 Communication Port Memory Map and Registers .....	8-8
■ Communication Port Control Register (CPCRs) .....	8-9
■ Input Port Register .....	8-9
■ Output Port Register .....	8-9
8.5 Communication Port Operation .....	8-12
■ Port Arbitration Units (PAUs) .....	8-12
■ Module Reset .....	8-14
■ Halting of Input and Output FIFOs .....	8-15
8.6 Coordinating Communication Port Activity with CPU and DMA Coprocessors .....	8-17
8.7 Communication Port Timing .....	8-18
■ Timing Table and Figures .....	8-18
■ Synchronizer Timing .....	8-31

## 8.1 Introduction

A parallel processor system supports optimum system performance by distributing tasks between two or more processors. This sharing of tasks between two or more TMS320C40 DSPs requires that each be able to pass the results of its work to another; passing of results enables both DSPs to continue working. Processor-to-processor communication is critical in multiprocessor-system design.

High-performance multiprocessing requires rapid transfer of data between processors. To ensure this rapid transfer of data, the TMS320C40 provides the following:

- ❑ **Shared memory** — The 'C40 global- and local-memory interfaces enable easy construction of efficient multiprocessor-based shared memory systems.
- ❑ **High-speed communication ports** — The 'C40's six high-speed bidirectional communication ports provide rapid processor-to-processor communication on six dedicated communication interfaces.

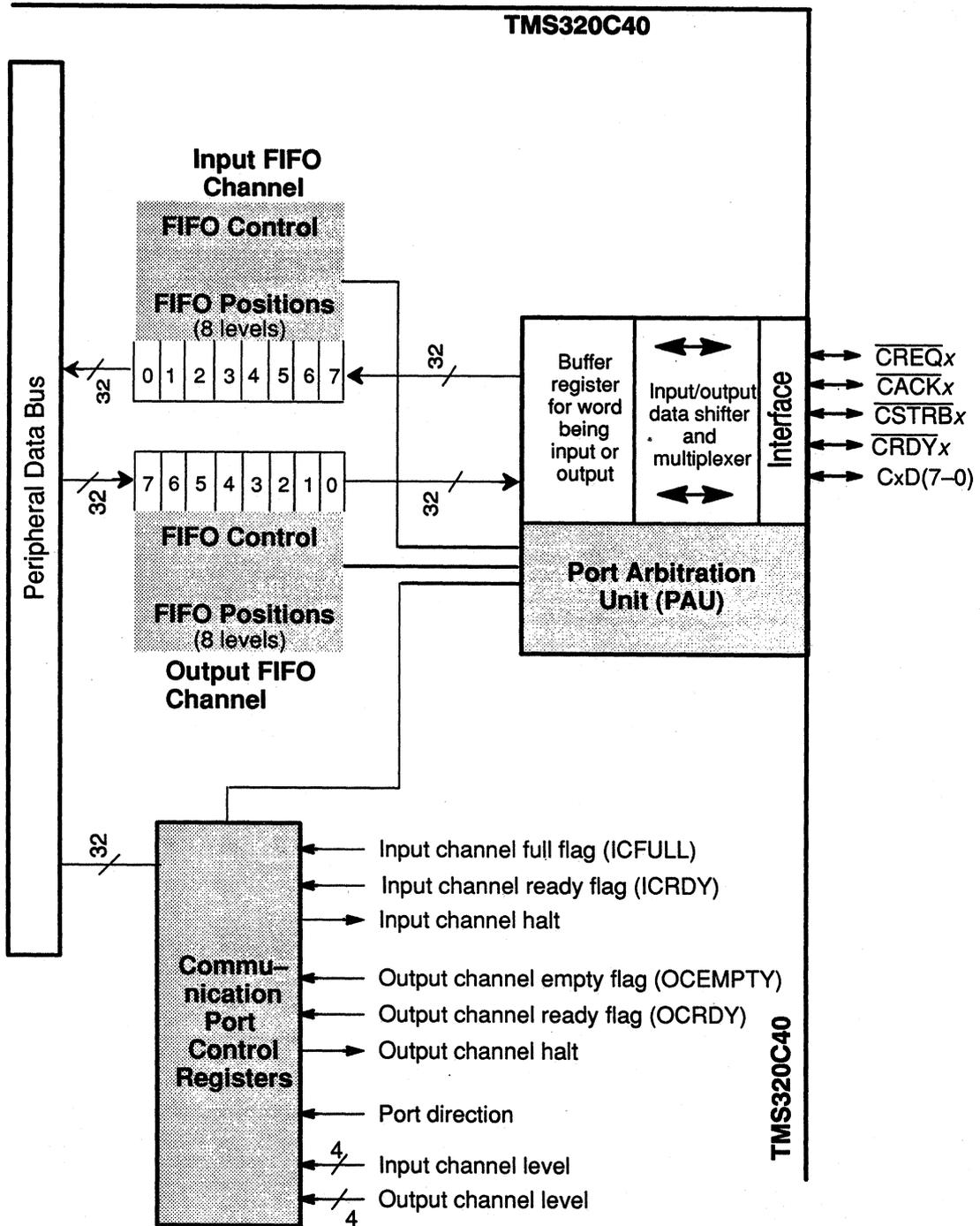
Although memory sharing has advantages in some applications, a shared bus seriously limits processor communication bandwidth for many applications. Using the high-speed communication ports eliminates this obstacle.

## 8.2 Communication Port Features

Key features of each TMS320C40 communication port:

- ❑ 160-megabit per second (5-megaword per second) bidirectional data transfer operations (at 40-ns cycle time)
- ❑ direct (glueless) processor-to-processor communication via eight data lines and four control lines
- ❑ buffering of all data transfers, both input and output
- ❑ automatic arbitration and handshaking to ensure communication synchronization
- ❑ synchronization between the CPU or direct-memory access (DMA) coprocessor and the six communication ports via internal interrupts and internal ready signals
- ❑ support of a wide variety of multiprocessor architectures, including rings, trees, hypercubes, bidirectional pipelines, two-dimensional Euclidean grids, hexagonal grids, and three-dimensional grids.

Figure 8-1. Communication Port Block Diagram



8

### 8.3 Operational Overview

The 'C40 contains six identical high-speed communication ports, each of which provides a bidirectional communication interface to an external device. Figure 8–1 shows the internal architecture of a single communication port. Each port contains the following components:

- ❑ **Input FIFO channel** — provides an 8-level, 32-bit wide first-in-first-out (FIFO) input buffer that isolates the 'C40 from the port communication data bus and buffers data received from an external device via the bus.
- ❑ **Output FIFO channel** — provides an 8-level, 32-bit wide FIFO output buffer that isolates the 'C40 from the port communication data bus and buffers data to be sent to an external device via the bus.
- ❑ **Port arbitration unit (PAU)** — handles the arbitration tasks associated with the movement of data between a 'C40 and an external device via the port communication data bus. Signals arbitrated and controlled by the PAU are shown in Figure 8–2. The PAU is described in detail in subsection 8.5.1 on page 8-12.
- ❑ **Communication port control register (CPCR)** — allows you to control the communication port functions and data transfer operations between a 'C40 and an external device via the communication port data bus.

Figure 8–2. TMS320C40 Communication-Port Interface-Connection Example

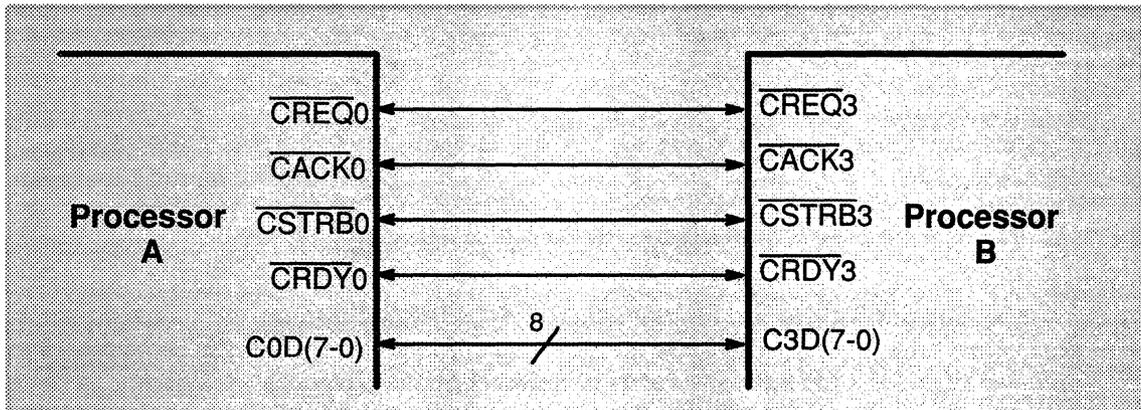


Figure 8–2 is an example of two 'C40 DSPs connected via their communication ports. This simple communication interface consists of the following bidirectional control and data lines:

- ❑  $\overline{\text{CREQ}}_x$  — communication port token request. A 'C40 activates this signal to request the use of the communication port data bus.

- ❑  $\overline{CACK}_x$  — communication port token acknowledge. A 'C40 activates this signal to relinquish ownership of the communication port data bus upon receiving a  $\overline{CREQ}_x$  from another 'C40.
- ❑  $\overline{CSTRB}_x$  — communication port strobe. A sending 'C40 activates this signal to indicate that it has placed valid data on the communication port data bus.
- ❑  $\overline{CRDY}_x$  — communication port ready. A receiving 'C40 activates this signal to indicate that it has received data via the communication port data bus.
- ❑  $CxD(7-0)$  — communication port data bus. This bus carries data bidirectionally between two 'C40s or between a 'C40 and some other device.

Figure 8–2 shows two 'C40s connected via their communication ports. The communication port data bus,  $CD(7-0)$ , and its associated control signals transfer data in either direction between 'C40s A and B. The PAUs in the two 'C40s cooperate to generate the signals and control sequences necessary to ensure orderly data transfers at the highest possible rate. To avoid conflicts on the bus, these PAUs arbitrate bus ownership, allowing only one DSP to transmit at any given time. Either of the PAUs can relinquish bus ownership when the other DSP has data to send.

Signals  $\overline{CREQ}_x$  and  $\overline{CACK}_x$  handle the handshaking arbitration between the two DSPs:

- 1) The PAU that does not own the data bus ( $CxD(7-0)$ ) activates  $\overline{CREQ}_x$  to request bus ownership.
- 2) The PAU owning the bus then activates  $\overline{CACK}_x$  to acknowledge the request and relinquish bus ownership to the requesting PAU.
- 3) In this manner, these signals transfer a token (or priority) from one PAU to another, and the PAU receiving the token gains ownership of the bus.

During a data transfer operation:

- 1) The CPU or DMA coprocessor of the sending DSP writes data to the output FIFO (of a communication port) via a memory-mapped address (listed in Figure 8–3).
- 2) The communication port then places the data on  $CxD(7-0)$  and activates  $\overline{CSTRB}_x$  to signal the receiving communication port that the bus contains valid data.
- 3) Upon receiving the data in its input FIFO, the receiving communication port activates  $\overline{CRDY}_x$  to indicate that it has received the data.
- 4) The CPU or DMA coprocessor of the receiving DSP may then read the data from the input FIFO via a memory-mapped address (listed in Figure 8–3).

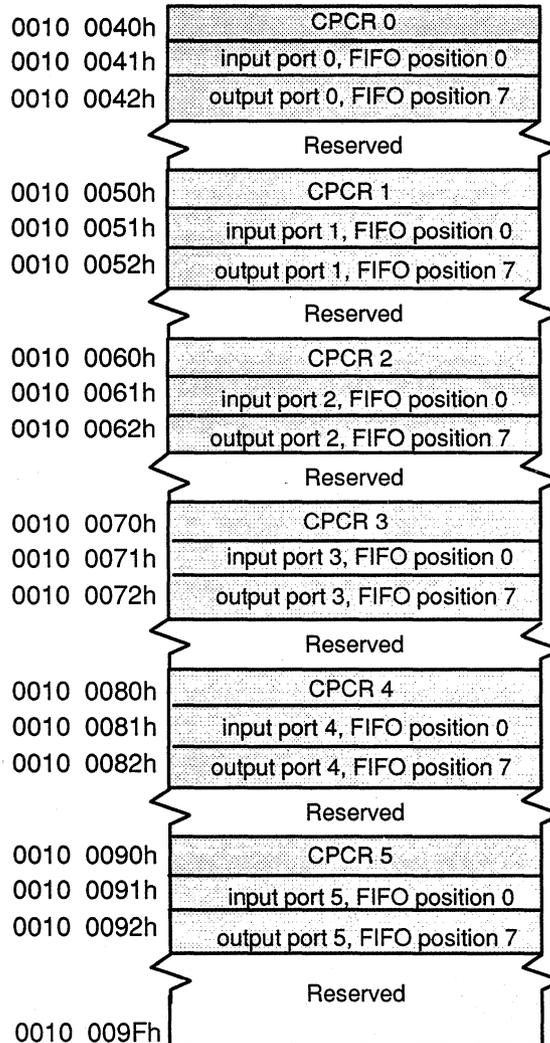
Each of the input and output FIFOs can buffer a maximum of eight 32-bit words.

Buffering provided by the input and output FIFOs is very important. This buffering allows for a high degree of decoupling of computation and communication overhead. When 'C40s A and B are connected via their communication ports, the effective length of the FIFOs becomes 16 levels. This is because the output path from A to B is the concatenation of the eight levels of the output FIFO of A with the eight levels of the input FIFO of B. This also applies for the output path from B to A.

## 8.4 Communication Port Memory Map and Registers

Figure 8–3 shows the memory map for the 'C40 communication port control registers (CPCRs) and their associated input FIFOs and output FIFOs. The lowest three addresses of each port's 16-address block are mapped to a CPCr and its associated input and output FIFOs. Fields (bits) within a CPCr are shown in Figure 8–4.

Figure 8–3. Communication Port Memory Map



For example, the addresses for **communication port 0** point to (see Figure 8–3):

- ❑ address 00010 0040h: CPCR 0
- ❑ address 00010 0041h: input port register 0, FIFO level 0
- ❑ address 00010 0042h : output port register 0, FIFO level 7
- ❑ address range 00010 0043h–00010 004Fh: reserved.

#### 8.4.1 Communication Port Control Registers (CPCRs)

Figure 8–4 shows the format of a TMS320C40 CPCR, which contains control and status bits for its associated communication port. Table 8–1 lists the CPCR bits and fields and describes their functions. Figure 8–3 lists the memory locations of the CPCRs.

If an output port that is full is written to, the peripheral bus interface latches the word written. On subsequent accesses to the peripheral bus, a *not ready* is given. This condition goes away when an empty position appears in the output FIFO. This results in the peripheral bus input latch being transferred to the output buffer at the communication port.

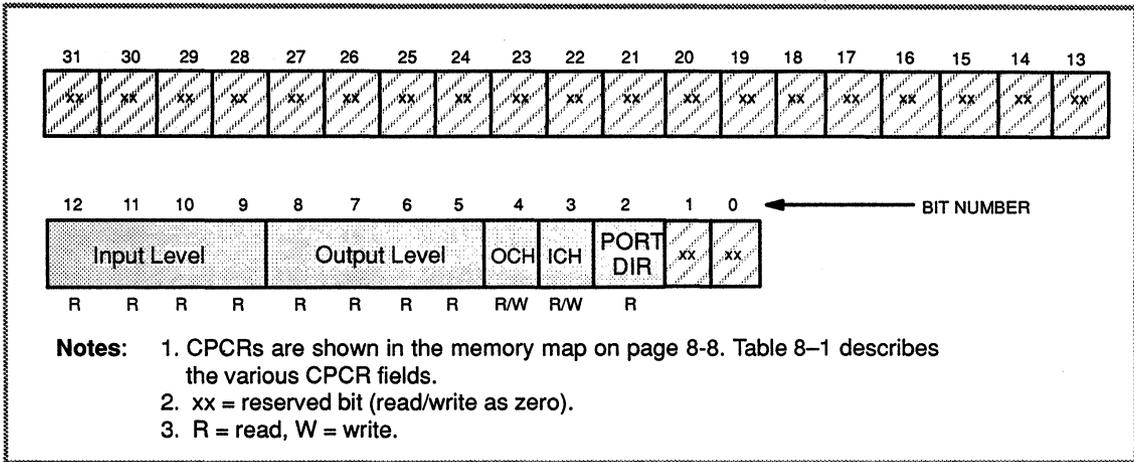
#### 8.4.2 Input Port Register

This read-only register contains the contents of position 0 of the input FIFO, the oldest value in the FIFO. If this register is written to, its contents remain unchanged.

#### 8.4.3 Output Port Register

This write-only register interfaces to position 7 of the output FIFO (level 7 — the newest value in the FIFO). If this register is read, its contents remain unchanged, and the value read is undefined (garbage).

Figure 8-4. Communication Port Control Register (CPCR)



- Notes:**
1. CPCR's are shown in the memory map on page 8-8. Table 8-1 describes the various CPCR fields.
  2. xx = reserved bit (read/write as zero).
  3. R = read, W = write.

Table 8-1. CPCR Bit Functions

Bit Nos.	Field Name	Function
0-1	Reserved	Undefined
2	PORT DIR	<b>Port Direction.</b> Bit determines the direction of data transfer operations for the communication port. <ul style="list-style-type: none"> <li>• PORT DIR = 0: port is in the output mode</li> <li>• PORT DIR = 1: port is in the input mode.</li> </ul>
3	ICH	<b>Input Channel Halt.</b> <ul style="list-style-type: none"> <li>• Write a 1 to ICH to halt the input channel. When the input channel is halted, PORT DIR is set to zero.</li> <li>• Set ICH to 0 when the input channel is to be unhalted; otherwise, the input channel cannot signal externally when it is ready to receive.</li> </ul>
4	OCH	<b>Output Channel Halt.</b> <ul style="list-style-type: none"> <li>• Write a 1 to this bit to immediately halt the output channel. However, the communication port is still able to accept a token request from the input channel.</li> <li>• Set this bit to 0 to allow the output channel to transfer data.</li> </ul>

(Table concluded on next page)

Table 8-1. *CPCR Bit Functions (Concluded)*

Bit Nos.	Field Name	Function
5-8	OUTPUT LEVEL	<p><b>Output FIFO Level.</b> Contents of this 4-bit field:</p> <ul style="list-style-type: none"> <li>• 0000<sub>2</sub> (0): indicates an empty output FIFO.</li> <li>• 0001<sub>2</sub> (1): through 0111<sub>2</sub> (7): indicates the number of full positions in the output FIFO.</li> <li>• 1111<sub>2</sub> (15): indicates a full output FIFO</li> </ul> <p>An empty output buffer (OUTPUT LEVEL = 0000<sub>2</sub>) causes an unlatched, positive level-triggered interrupt (OEMPTY = 1) to be sent to the CPU. When the CPU or DMA coprocessor writes to the empty output FIFO, OEMPTY is set to 0, and it remains in that state until the buffer is again empty. An output FIFO with one or more empty levels also causes an unlatched, positive level-triggered interrupt (OCRDY = 1) to be sent to the CPU and the DMA coprocessor. This condition causes a READY/NOT READY signal to be generated when the CPU or DMA coprocessor attempts to write to the output FIFO.</p>
9-12	INPUT LEVEL	<p><b>Input FIFO level.</b> Contents of this 4-bit field:</p> <ul style="list-style-type: none"> <li>• 0000<sub>2</sub> (0): indicates an empty input FIFO.</li> <li>• 0001<sub>2</sub> (1): through 0111<sub>2</sub> (7): indicates the number of full positions in the input FIFO.</li> <li>• 1111<sub>2</sub> (15): indicates a full input FIFO.</li> </ul> <p>A full input FIFO (INPUT LEVEL = 1111<sub>2</sub>) causes an unlatched, positive level-triggered interrupt (ICFULL = 1) to be sent to the CPU. When the CPU or DMA coprocessor reads from the full input FIFO, ICFULL is set to 0 and remains in that state until the FIFO is again full. An input FIFO with one or more full levels also causes an unlatched, positive level-triggered interrupt (ICRDY = 1) to be sent to the CPU and the DMA coprocessor. This condition causes a READY/NOT READY signal to be generated when the CPU or DMA coprocessor attempts to read from the output FIFO.</p>
13-31	Reserved	Undefined

## 8.5 Communication Port Operation

### 8.5.1 Port Arbitration Units (PAUs)

The PAU is responsible for arbitrating between two devices to determine which device has possession of the communication port data bus at any given time. This arbitration allows the bus ownership token to be passed back and forth between two devices connected via their communication ports. During this arbitration process, the PAU is in one of the four states listed in Table 8–2.

Table 8–2. PAU State Definitions

Summary	PAU State	PAU Status
1. PAU has token (PORT DIR = 0) 2. Channel not in use	0 0	The PAU currently has possession of the bus ownership token, and its associated communication channel is not in use. Under this condition, the PORT DIR bit of the associated CPCR is 0 (output).
1. PAU does not have token (PORT DIR = 1). 2. Token not requested by PAU (OUTPUT LEVEL = 0).	0 1	The PAU currently does not have possession of the bus ownership token and has not requested the token. Under this condition, the PORT DIR bit equals 1 (input), and the OUTPUT LEVEL field equals 0 (empty output FIFO).
1. PAU has token (PORT DIR = 0). 2. Channel is in use (OUTPUT LEVEL ≠ 0).	1 0	The PAU currently has possession of the bus ownership token, and its associated communication channel is in use. Under this condition, the PORT DIR bit equals 0 (output), and the OUTPUT LEVEL field <i>does not</i> equal 0.
1. PAU does not have token (PORT DIR = 1). 2. Token requested by PAU (OUTPUT LEVEL ≠ 0).	1 1	The PAU currently does not have possession of the bus ownership token but has requested the token. Under this condition, the PORT DIR bit equals 1 (input), and the OUTPUT LEVEL field <i>does not</i> equal 0.

8

Figure 8–5 shows the state diagram and controlling equations for the PAU state transitions. The figure also includes comments describing how the state transitions correspond to various system-level processes.

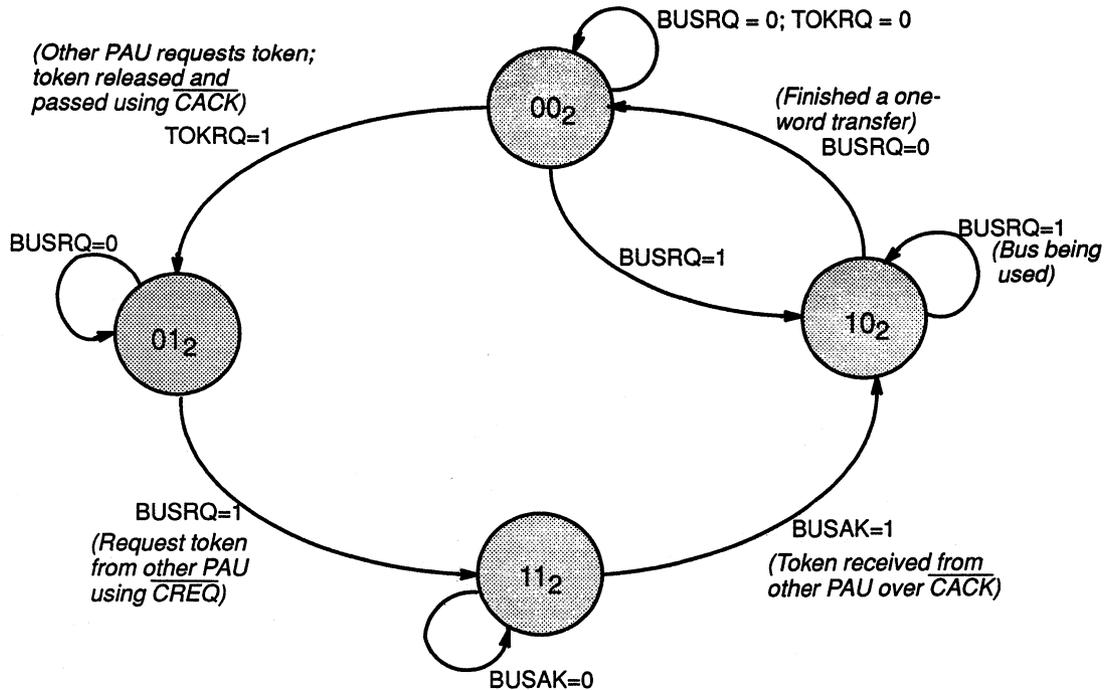
To place data on the communication port data bus, the PAU must arbitrate between:

- ❑ on-chip requests to output data on the communication channel data bus (CD(7– 0))
- ❑ external requests received via the  $\overline{\text{CREQ}}$  line

This arbitration is accomplished by passing the bus-ownership token between PAUs associated with different communication ports. The PAU containing the token has ownership of the communication port data bus. At system reset, half of the communication channels associated with a particular

'C40 have token ownership (communication ports 0, 1, 2), and the other half (communication ports 3, 4, 5) do not. This token passing is done via the  $\overline{CREQ}$  and  $\overline{CACK}$  lines.

Figure 8-5. Communication Port Arbitration Unit State Diagram



To help understand the port arbitration scheme represented in Figure 8-5, consider a data transfer operation from 'C40 A to 'C40 B. The transfer begins with PAU A in state 00<sub>2</sub> and PAU B in state 01<sub>2</sub>. If PAU A receives a request (BUSRQ = 1) from its output buffer to use the communication port data bus, it allows the output buffer to transmit one word immediately and enter state 10<sub>2</sub>. After the output buffer transmits one word, it removes the bus request (BUSRQ = 0), and PAU A returns to state 00<sub>2</sub>.

If PAU B receives a request from its output buffer to use the bus, it activates  $\overline{CREQ}$  to request the token from PAU A. PAU A detects this request via the state variable TOKRQ and then activates the  $\overline{CACK}$  line to transfer the bus ownership token to PAU B. PAU B then generates an internal bus acknowledge (BUSACK) to indicate that it has gained bus ownership. As a result of this token transfer operation, PAU A enters state 01<sub>2</sub>, and PAU B enters state 10<sub>2</sub>.

Because a PAU always returns to state 00<sub>2</sub> after transmitting a single word, token passing can be accomplished by 'C40s A and B alternately transmitting single words. This process provides a fair means of bus arbitration that prevents either of the output buffers (A's or B's) from being continually blocked.

If an input buffer becomes full, it will not activate  $\overline{\text{CRDY}}$  at the beginning of the transmission of the first byte that would overflow the buffer. This condition prevents data transfer operations in either direction until the situation is resolved. This can be done by reading data from the full input buffer.

### 8.5.2 Module Reset

At system reset, the input and output channels both assume an empty state, causing all values in the input and output buffers to be lost. The  $\overline{\text{CREQ}}$ ,  $\overline{\text{CACK}}$ ,  $\overline{\text{CSTRB}}$ , and  $\overline{\text{CRDY}}$  signals assume an inactive (high) state and CxD(7–0) enters its tristate mode (see Figure 8–14 and Figure 8–15 on page 8-30). These signals remain in these states as long as system reset is active and, following system reset, the value placed on CxD(7–0) by the communication port that is configured for output is undefined.

**At system reset, communication ports 0, 1, and 2 assume the following states:**

- PAU is reset to state 00<sub>2</sub>:** The PAU has possession of the bus ownership token, and the channel is not in use.
- ICRDY = 0:** The input channel is empty and is not ready to be read from.
- ICH = 0:** The input channel is not in its halted state.
- OCRDY = 1:** The output channel is not full and is ready to be written to.
- OCH = 0:** The output channel is not in its halted state.
- PORT DIR = 0:** The communication port is configured for output operation.
- INPUT LEVEL = 0:** The input channel is empty.
- OUTPUT LEVEL = 0:** The output channel is empty.

**At system reset, communication ports 3, 4, and 5 assume the following states:**

- PAU reset to state 01<sub>2</sub>:** The PAU does not have possession of the bus ownership token, and the token is not requested.
- ICRDY = 0:** The input channel is empty and is not ready to be read from.

- ICH = 0:** The input channel is not in its halted state.
- OCRDY = 1:** The output channel is not full and is ready to be written to.
- OCH = 0:** The output channel is not in its halted state.
- PORT DIR = 1:** The communication port is configured for input operation.
- INPUT LEVEL = 0:** The input channel is empty.
- OUTPUT LEVEL = 0:** The output channel is empty.

Based on these reset conditions, ports 0, 1, and 2 of one DSP should be connected to ports 3, 4, and 5 of the other.

#### Connect Ports to Opposite Input/Output Reset Mode

At reset, ports 0, 1, and 2 are configured as output ports (PORT DIR = 0) and ports 3, 4, and 5 are configured as input ports (PORT DIR = 1). When connecting ports, connect each to a port that would be in the *opposite* direction at reset (any of 0, 1, or 2 connected to any of 3, 4, or 5).

### 8.5.3 Halting of Input and Output FIFOs

The halting of the input and output FIFOs of a communication channel is controlled by the ICH and OCH bits (input-channel and output-channel halt bits) of the communication port control register (Figure 8-4 on page 8-10). The goal of input FIFO halting is to halt the input FIFO as soon as possible, but without the loss of data being input. A summary of the halt/unhalted conditions is provided in Table 8-3 on page 8-16.

When the input FIFO is halted, it will not signal a ready when the first incoming byte is received. At that point, the data transfer is frozen until the input FIFO is unhalted or a system reset occurs. If the input FIFO is unhalted later, the transfer will continue without any loss of data.

A communication port with an FIFO that is either halted or is full and inactive will not acknowledge a token request. This assures that the communication port's output channel remains open.

If a communication port's input FIFO is halted during a token request from the communication port to which it is connected, then the token request is acknowledged before halting.

Table 8-3. Summary of Input and Output FIFO Halting

Halted/Unhalted	If the Port Has Token	If the Port Does Not Have Token
Input halted Output unhalted	a. Won't release token b. Will transmit data	a. Won't signal ready when first byte is received (transfer frozen) b. If halted after first byte is received, it will receive rest of word (will signal ready and then halt the input)
Input unhalted Output halted	a. Won't transmit data b. If halted after first byte sent, will complete word transfer and then halt the output c. Will release token	a. Will receive data b. Will not request token
Input halted Output halted	a. Won't release token b. Won't transmit data c. If halted after first byte sent, will complete word transfer and then halt the output	a. Won't signal ready when first byte is received (transfer frozen) b. If halted after first byte received, it will receive rest of word and then halt the input c. Will not request token

Output FIFO halting is analogous to input FIFO halting. Assume that DSP A output FIFO has OCH = 1. Then the output FIFO will be halted, based upon its current state.

- ❑ If communication port A *does not have the token*, the output FIFO is halted, and no request is made for the token.
- ❑ If communication port A *has the token and is currently transmitting a word*, then after the word is transmitted, no new transfers will be begun.
- ❑ If communication port A has the token **and** the input FIFO is not halted **and** the output FIFO is halted, then *it will transfer the token* when requested by communication port B.
- ❑ If communication port A has the token **and** the input FIFO is halted **and** the output FIFO is halted, then *it will not transfer the token* when requested by communication port B.
- ❑ When coming out of the halted state, if the communication channel still has the token, it may transmit data if necessary. If it needs the token, it will arbitrate for the token as usual.

## 8.6 Coordinating Communication Port Activity With CPU and DMA Coprocessors

The communication ports support several principle modes of synchronization:

- ❑ a ready/not ready signal that can halt CPU and DMA accesses to a communication port
- ❑ interrupts that can be used to signal the CPU and DMA

The most basic synchronization mechanism is based on a ready/not-ready signal. If the DMA or CPU attempt to read an empty input FIFO or write to a full output FIFO, a not-ready signal is returned and the DMA or CPU continues to read or write until a ready signal is received. The ready signal for the output channel is OCRDY (output channel ready), which is also an interrupt signal. The ready signal for the input channel is ICRDY (input channel ready), which is also an interrupt signal.

Interrupts are often a useful form of synchronization. Each communication port generates four different interrupt signals, as listed below (interrupt traps for these are shown in Figure 3–8 on page 3-16):

- ❑ ICFULL (input channel full)
- ❑ ICRDY (input channel ready)
- ❑ OCRDY (output channel ready)
- ❑ OCEMPTY (output channel empty)

The CPU can respond to all four of these interrupt signals. The DMA coprocessor can respond to the ICRDY and OCRDY interrupt signals.

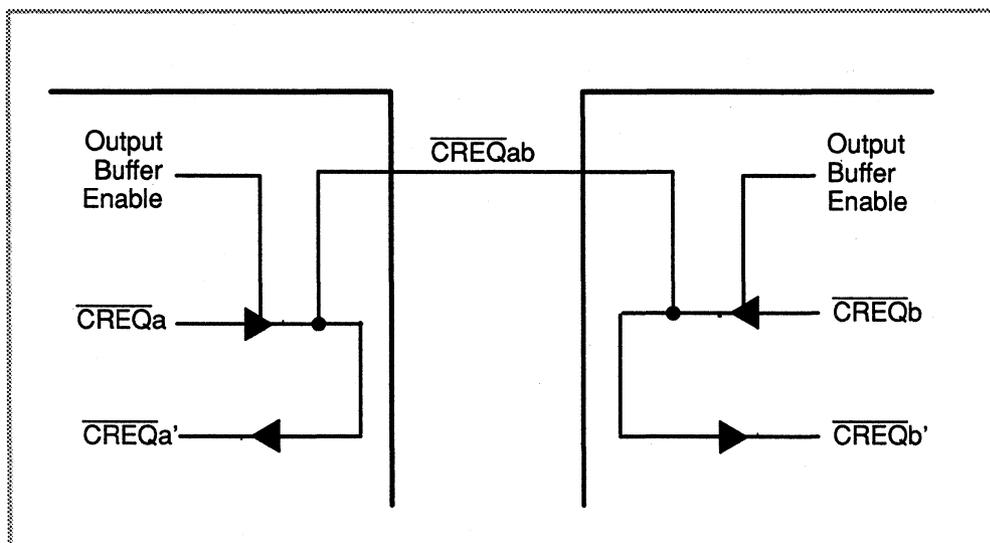
## 8.7 Communication Port Timing

In order to accurately describe the timing of the operation of the communication ports, it is important to differentiate between the internal signals applied to the pins and the external signal seen. All signals are buffered and can be placed in a high-impedance state. See Figure 8–6.

In this discussion, internal signals applied to a buffer are identified by suffixes:

- a suffix 'a' for processor A (for example,  $\overline{\text{CSTRBa}}$ )
- a suffix 'b' for processor B (for example,  $\overline{\text{CSTRBb}}$ )
- a suffix 'ab' for the external signal between the two connected communication ports (for example  $\overline{\text{CSTRBab}}$  and  $\overline{\text{CREQab}}$ )
- a suffix followed by a single quote for the value that the processor sees by sampling the input pad (for example  $\text{CPTRa}'$ )

Figure 8–6. Signal-Naming Example



### 8.7.1 Timing Table and Figures

Table 8–4 and the timing figures that follow depict timing sequences in communication between TMS320C40s using their communication ports. Table 8–4 lists handshaking and communication during this intercommunication. Steps in the table are shown by numbers in the figures. Events 1 through 36 in the table compose a token request and token transfer sequence.

**Figure 8–7** *Token Transfer Sequence* (page 8-23).

- 1) At the start, the communication port on processor A has the token and is idle.
- 2) The communication port on processor B requests the token and, after receiving the token, transfers a word, one byte at a time:
  - a) the first byte is bits 7–0
  - b) the second is bits 15–8
  - c) the third is bits 23–16
  - d) the fourth is 31–24
- 3) Once a token-requesting communication port receives the token request acknowledge, it will always transmit a word.

**Figure 8–8** *End of Token Transfer Sequence Followed by a Word Transfer and the Beginning of a Second Word Transfer* (page 8-24).**Figure 8–9** *End of a Word Transfer Followed by a Word Transfer* (page 8-25).**Figure 8–10** *End of a Word Transfer Followed by an Idle State and Token Transfer* (page 8-26).

- 1) The communication port data bus becomes idle because the output FIFO on processor B is empty.
- 2) The communication port on processor A requests the token, which is then transferred to it by the communication port on processor B.

**Figure 8–11** *End of a Word Transfer Followed by an Overlapping Token Transfer* (page 8-27).

- 1) As shown, the token request is received by the communication port on processor B.
- 2) The communication port on processor B sees the ready signal for the last byte of the word being transmitted.
- 3) Then the communication port releases the token.
- 4) However, the communication port will not release the token if the token request is received by the processor port B after the processor port B sees the ready signal for the last byte of the word being transmitted.
- 5) If the communication port on processor B does not have another word in the output FIFO to transmit, it will release the token.

**Figure 8–12** *End of the Transfer of the Last Word in an Output FIFO Followed by an Idle Condition Until Another Word Is Available to Be Transferred* (page 8-28). This begins with a word transfer followed by an idle state due to an empty output FIFO. Then a word is written to the output FIFO and transferred.**Figure 8–13** *End of a Word Transfer Followed by a Not Ready Due to the Input FIFO Becoming Full, Continuing Once the Input FIFO Is No Longer Full* (page 8-29). This shows the use of the ready line to generate wait states.

- 1) In this case, a word is transferred that fills the input FIFO of the communication port of processor A.
- 2) At the beginning of transmission of the next word, the communication port on processor A does not signal that it is ready until the input FIFO is no longer full.

Table 8-4. Handshaking Events in Communication Port Intercommunication

† Event No.	Description
1	B requests the token by bringing $\overline{\text{CREQb}}$ low.
2	A sees the token request when $\overline{\text{CREQa}}$ goes low.
3	After a type 1 delay from $\overline{\text{CREQa}}$ falling, A acknowledges the request by bringing $\overline{\text{CACKa}}$ low.
4	B sees the acknowledgement from A when $\overline{\text{CACKb}}$ goes low.
5	A switches $\overline{\text{CRDYa}}$ from tristate to high on the first H1 rising after $\overline{\text{CACKa}}$ falling.
6	A tristates $\text{CaD}(7-0)$ on the first H1 rising after $\overline{\text{CACKa}}$ falling.
7	B switches $\overline{\text{CSTRBb}}$ from tristate to high after $\overline{\text{CACKb}}$ falling.
8	B brings $\overline{\text{CREQb}}$ high after a type 1 delay from $\overline{\text{CACKb}}$ falling.
9	A sees $\overline{\text{CREQa}}$ go high.
10	A brings $\overline{\text{CACKa}}$ high after $\overline{\text{CREQa}}$ goes high.
11	A tristates $\overline{\text{CSTRBa}}$ after $\overline{\text{CACKa}}$ goes high.
12	A tristates $\overline{\text{CACKa}}$ after $\overline{\text{CREQa}}$ goes high and after $\overline{\text{CACKa}}$ goes high.
13	A switches $\overline{\text{CREQa}}$ from tristate to high after $\overline{\text{CREQa}}$ goes high.
14	B tristates $\overline{\text{CREQb}}$ after $\overline{\text{CREQb}}$ goes high.
15	B switches $\overline{\text{CACKb}}$ from tristate to high after $\overline{\text{CREQb}}$ goes high.
16	B tristates $\overline{\text{CRDYb}}$ on H1 rising after $\overline{\text{CREQb}}$ goes high.
17	B drives the first byte onto $\text{CbD}(7-0)$ on H1 rising after $\overline{\text{CREQb}}$ goes high.
18	A sees the first byte on $\text{Ca'D}(7-0)$ .
19	B brings $\overline{\text{CSTRBb}}$ low on the second H1 rising after $\overline{\text{CREQb}}$ rising.
20	A sees $\overline{\text{CSTRBa}}$ go low, signaling valid data.
21	A reads the data and brings $\overline{\text{CRDYa}}$ low
22	B sees $\overline{\text{CRDYb}}$ go low, signaling data has been read,
23	B drives the second byte on $\text{CbD}(7-0)$ after $\overline{\text{CRDYb}}$ goes low.
24	A sees the second byte on $\text{Ca'D}(7-0)$ .

† Event No. corresponds to numbers in the timing diagrams that follow.

Table Continued on Next Page

Table 8-4. Handshaking Events in Communication Port Intercommunication (Continued)

† Event No.	Description
25	B brings $\overline{\text{CSTRBb}}$ high after $\overline{\text{CRDYb}}$ ' goes low.
26	A sees $\overline{\text{CSTRBa}}$ ' go high.
27	A brings $\overline{\text{CRDYa}}$ high after $\overline{\text{CSTRBa}}$ ' goes high.
28	B sees $\overline{\text{CRDYb}}$ ' go high.
29	B brings $\overline{\text{CSTRBb}}$ low after $\overline{\text{CRDYb}}$ ' goes high.
30	A sees $\overline{\text{CSTRBa}}$ ' go low, signaling valid data.
31	A reads the data and brings $\overline{\text{CRDYa}}$ low.
32	B sees $\overline{\text{CRDYb}}$ ' go low, signaling data has been read.
33	B drives the third byte on $\text{CbD}(7-0)$ after $\overline{\text{CRDYb}}$ ' goes low.
34	A sees the third byte on $\text{CaD}(7-0)$ .
35	B brings $\overline{\text{CSTRBb}}$ high after $\overline{\text{CRDYb}}$ ' goes low.
36	A sees $\overline{\text{CSTRBa}}$ ' go high.
37	A brings $\overline{\text{CRDYa}}$ high after $\overline{\text{CSTRBa}}$ ' goes high.
38	B sees $\overline{\text{CRDYb}}$ go high.
39	B brings $\overline{\text{CSTRBb}}$ low after $\overline{\text{CRDYb}}$ ' goes high.
40	A sees $\overline{\text{CSTRBa}}$ ' go low, signaling valid data.
41	A reads the data and brings $\overline{\text{CRDYa}}$ low.
42	B sees $\overline{\text{CRDYb}}$ ' go low, signaling data has been read.
43	B drives the fourth byte on $\text{CbD}(7-0)$ after $\overline{\text{CRDYb}}$ ' goes low.
44	A sees the fourth byte on $\text{CaD}(7-0)$ .
45	B brings $\overline{\text{CSTRBb}}$ high after $\overline{\text{CRDYb}}$ ' goes low.
46	A sees $\overline{\text{CSTRBa}}$ ' go high.
47	A brings $\overline{\text{CRDYa}}$ high after $\overline{\text{CSTRBa}}$ ' goes high.
48	B sees $\overline{\text{CRDYb}}$ ' go high.
49	B brings $\overline{\text{CSTRBb}}$ low after $\overline{\text{CRDYb}}$ ' goes high.
50	A sees $\overline{\text{CSTRBa}}$ ' go low, signaling valid data.
51	A reads the data and brings $\overline{\text{CRDYa}}$ low.
52	B sees $\overline{\text{CRDYb}}$ ' go low, signaling data has been read.
53	B brings $\overline{\text{CSTRBb}}$ high after $\overline{\text{CRDYb}}$ ' goes low.

† Event No. corresponds to numbers in the timing diagrams that follow.

Table Concluded on Next Page

Table 8-4. Handshaking Events in Communication Port Intercommunication (Concluded)

† Event No.	Description
54	A sees $\overline{\text{CSTRBa}}$ ' go high.
55	A brings $\overline{\text{CRDYa}}$ high after $\overline{\text{CSTRBa}}$ ' goes high.
56	B sees $\overline{\text{CRDYb}}$ ' go high.
57	B drives the first byte of the next word onto $\text{CbD}(7-0)$ after a type 2 delay from $\overline{\text{CRDYb}}$ ' falling (52).
58	A sees the first byte of the next word on $\text{CaD}(7-0)$ .
59	B lowers $\overline{\text{CSTRBb}}$ after a type 2 delay from $\overline{\text{CRDYb}}$ ' falling.

† Event No. corresponds to numbers in the timing diagrams that follow.

These events are identified by event number in the following figures that describe the communication port timing.

Figure 8-7. Token Transfer Sequence

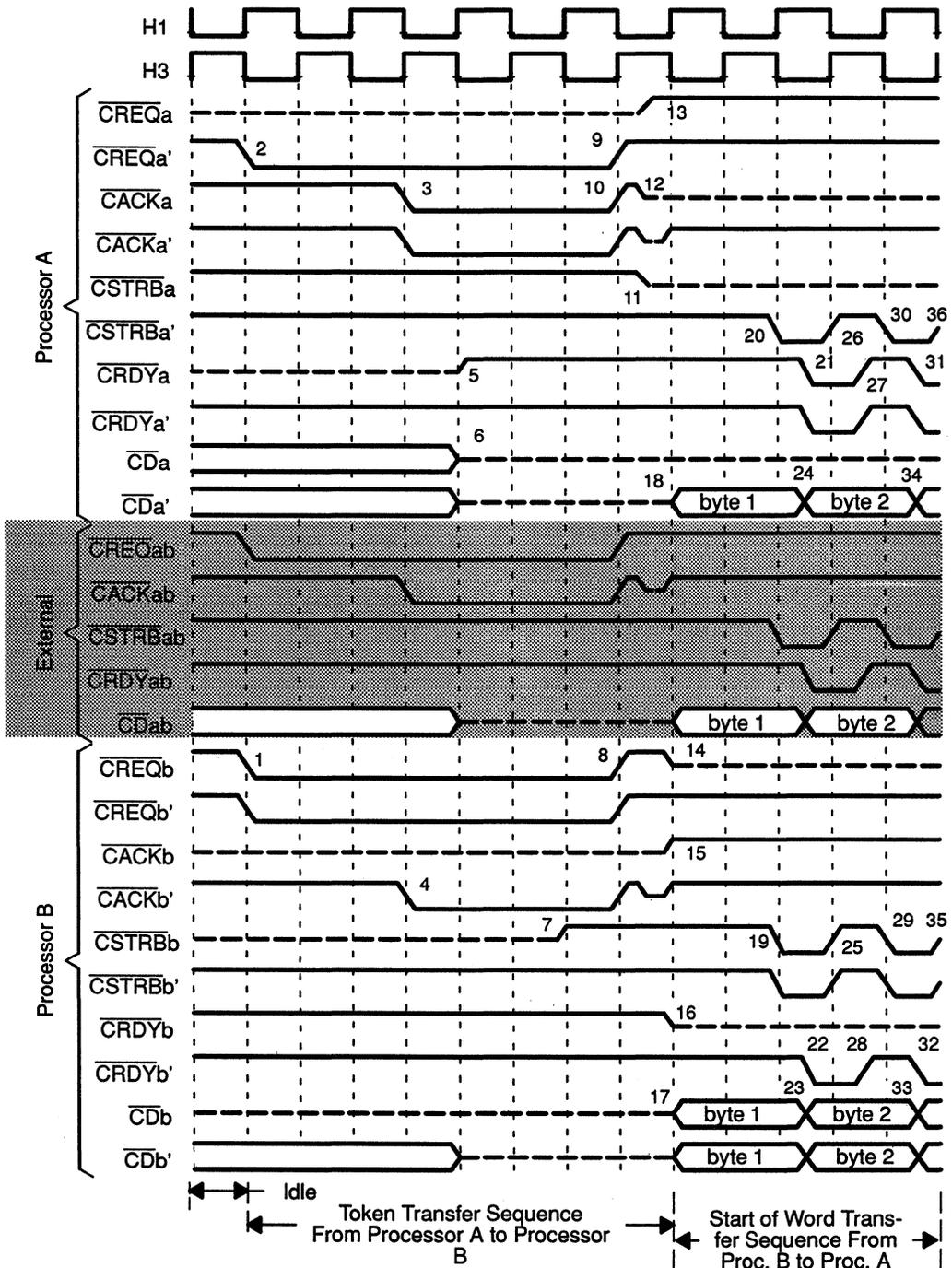
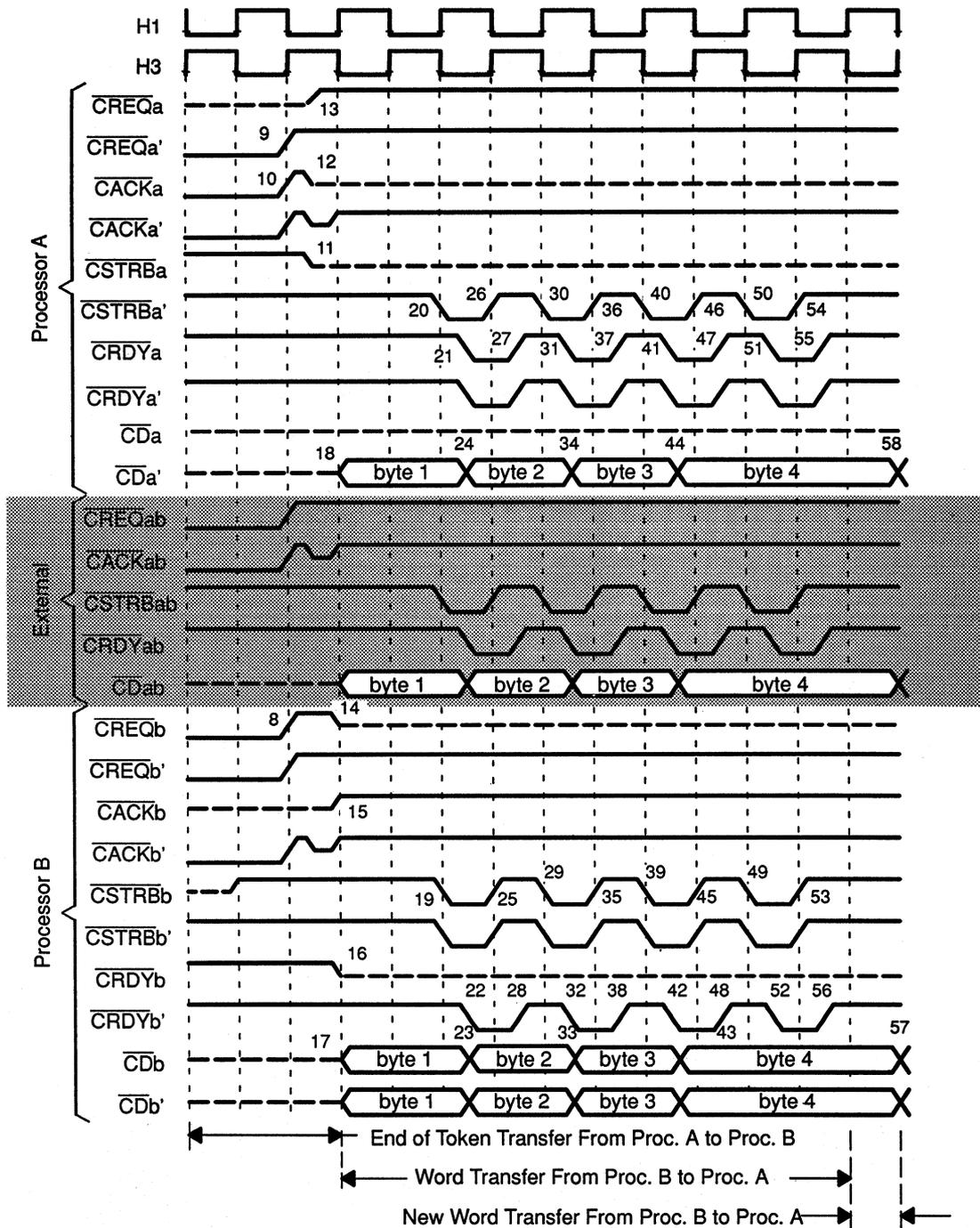


Figure 8-8. End of Token Transfer Sequence Followed by a Word Transfer and the Beginning of a Second Word Transfer



8

Figure 8-9. End of a Word Transfer Followed by a Word Transfer

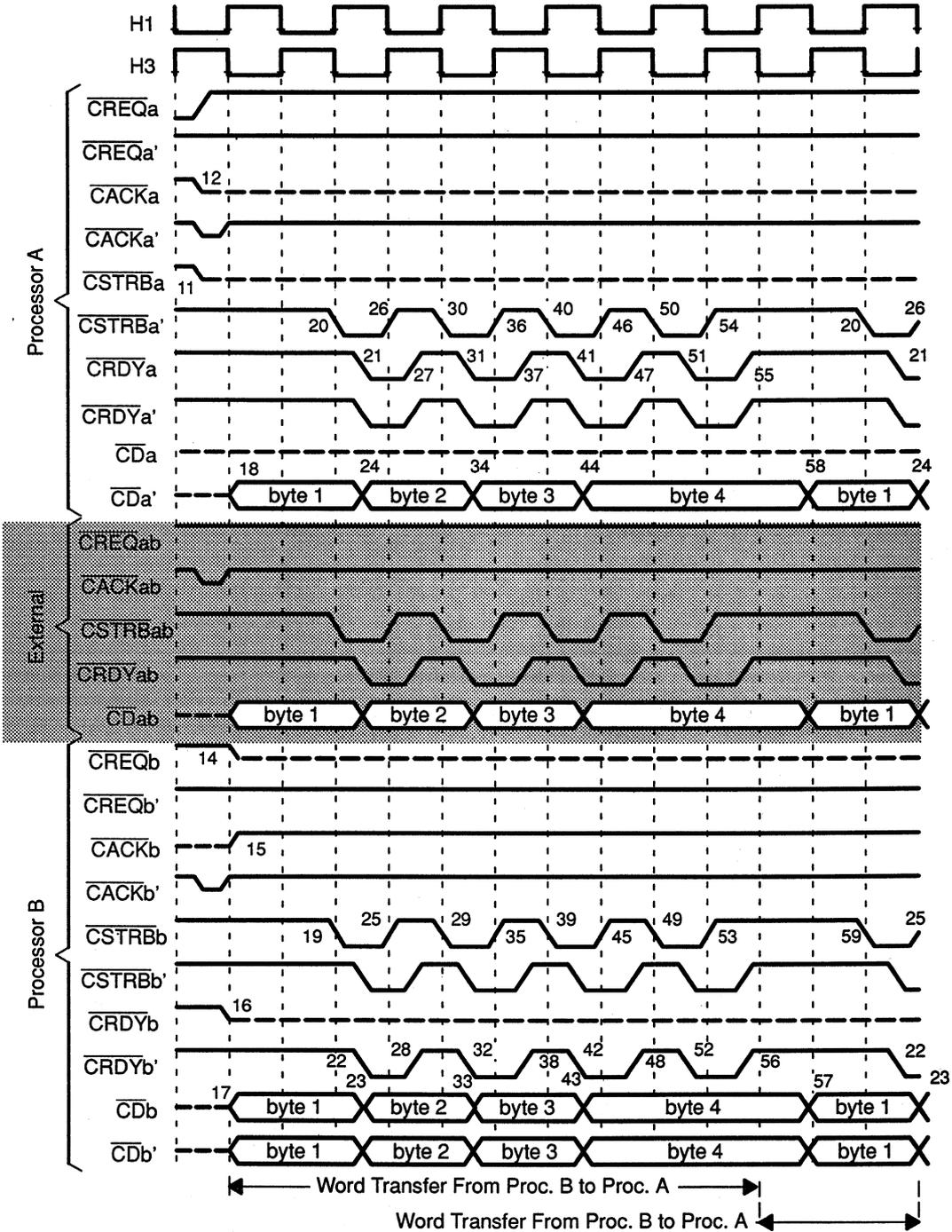
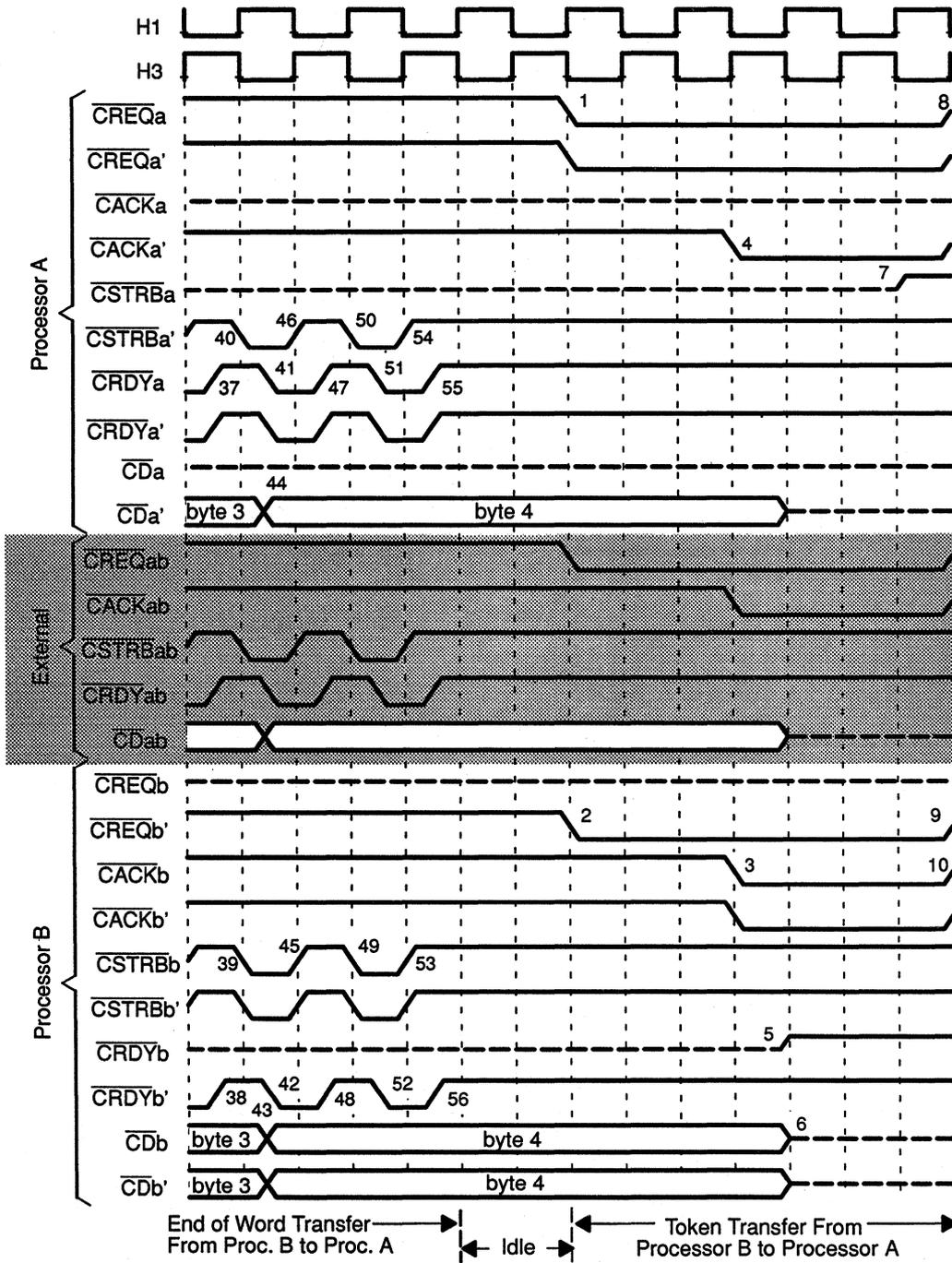
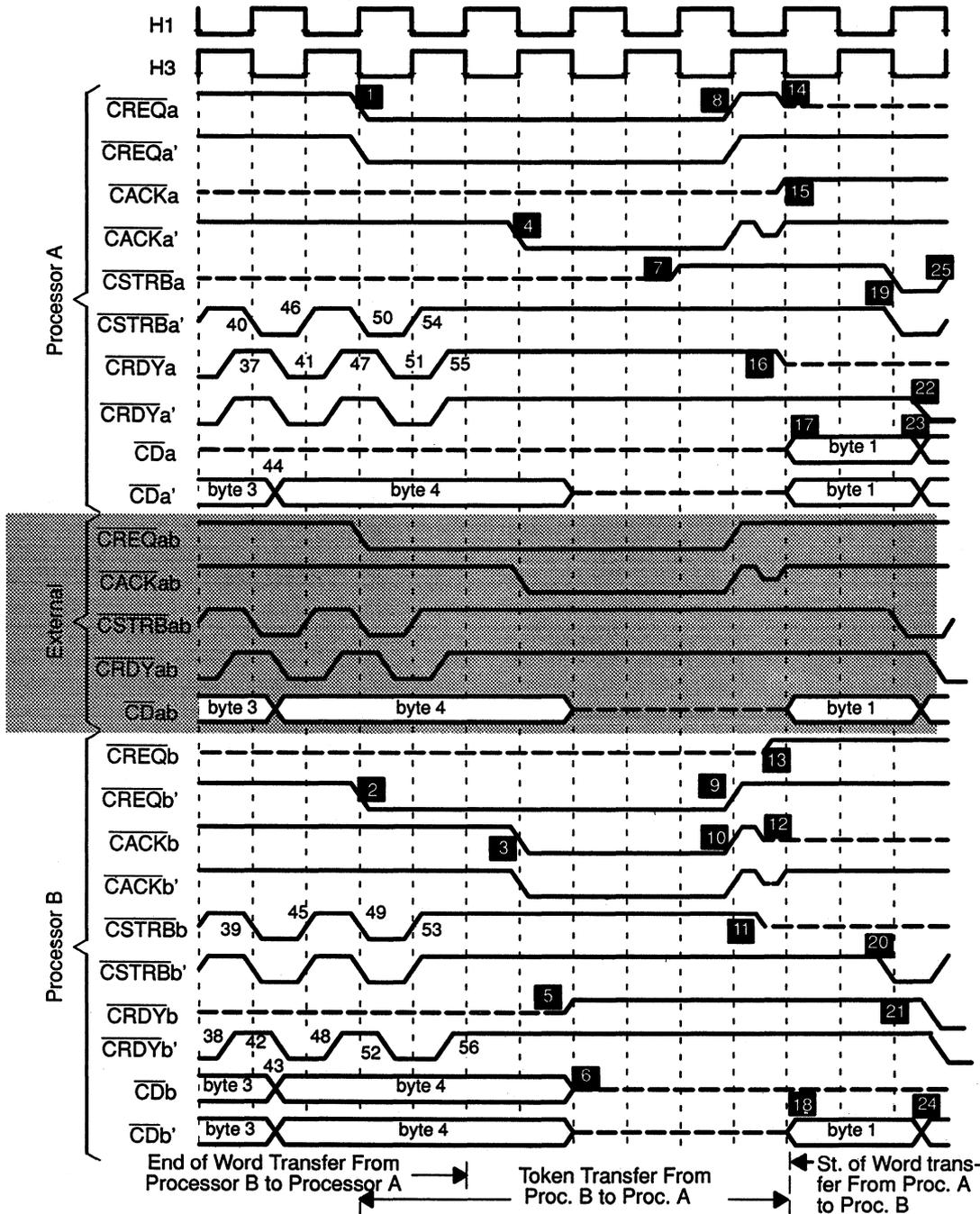


Figure 8-10. End of a Word Transfer Followed by an Idle State and Token Transfer



8

Figure 8-11. End of a Word Transfer Followed by an Overlapping Token Transfer



8

NOTE: Events 1 — 25 are complements to the description in Table 8-4 on page 8-20 (i.e., if "a" is in the description, substitute "b" and vice versa, — CDa' becomes CDb'; CDb' becomes Cba').

Figure 8-12. End of the Transfer of the Last Word in an Output FIFO Followed by an Idle Condition Until Another Word Is Available to Be Transferred

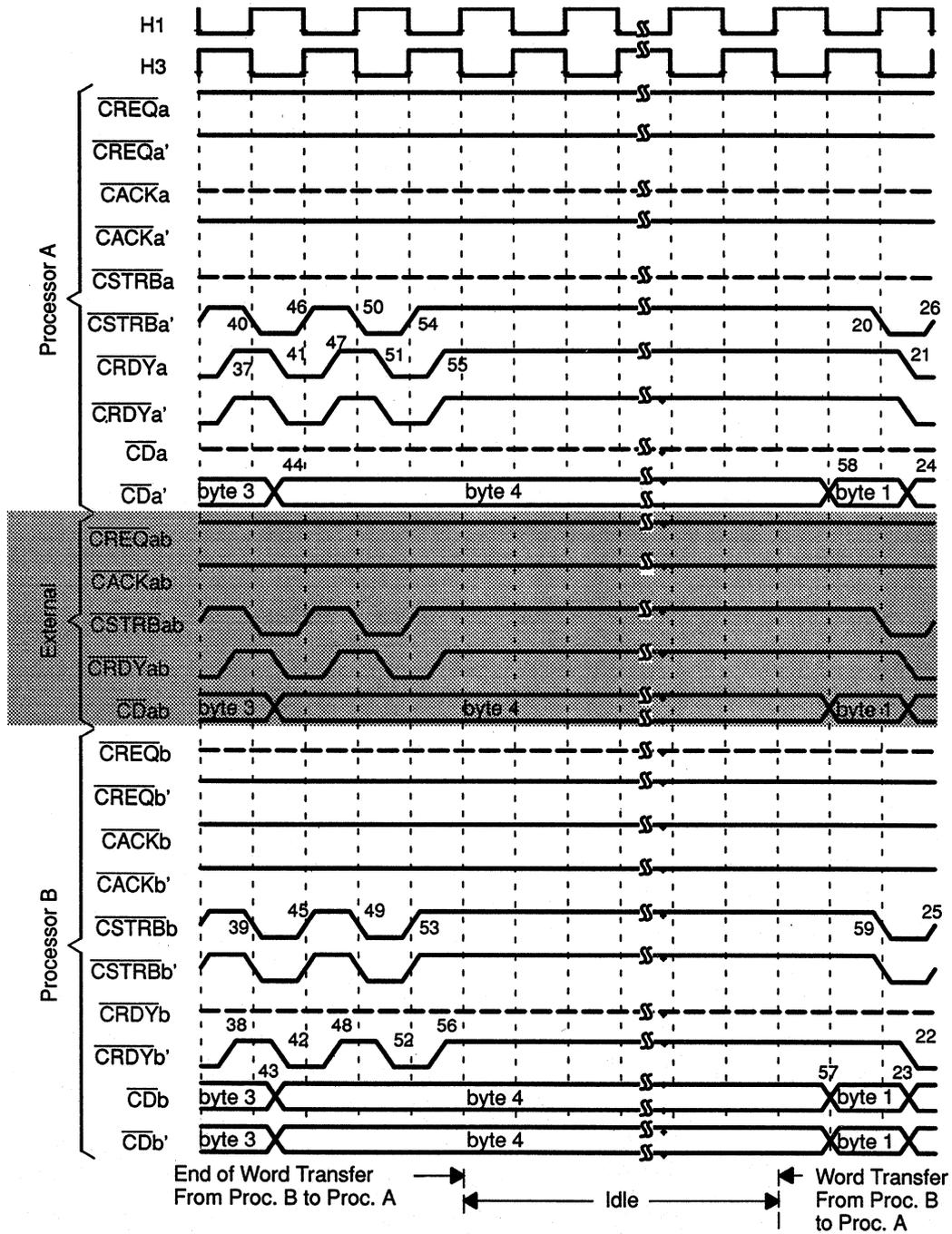


Figure 8-13. End of a Word Transfer Followed by a Not Ready Due to the Input FIFO Becoming Full, Continuing Once the Input FIFO Is No Longer Full

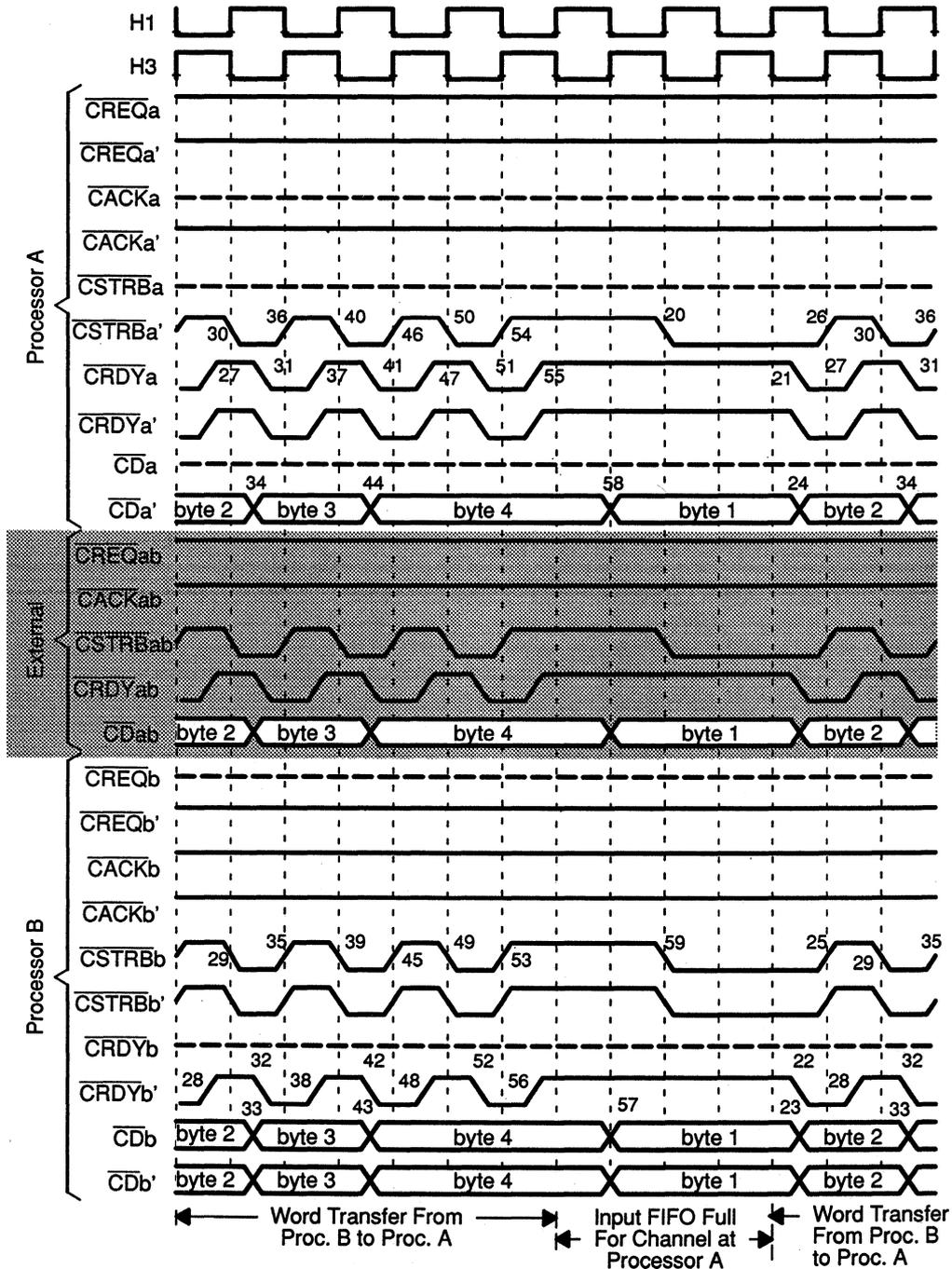


Figure 8–14 illustrates the state of the signals of a communication port initialized by a reset as an output port (ports 0, 1, and 2 are configured as output ports at reset). For this case,  $\overline{\text{CREQ}}$  and  $\overline{\text{CRDY}}$  are in a high-impedance state.  $\overline{\text{CACK}}$  and  $\overline{\text{CSTRB}}$  are high, and undefined values are on CD(7–0).

Figure 8–14. Post-Reset State for an Output Port

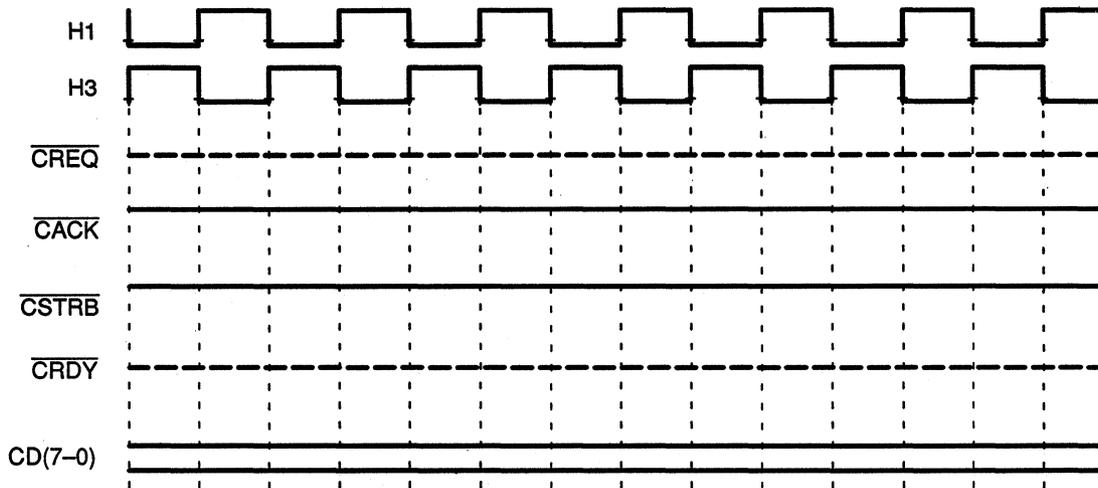
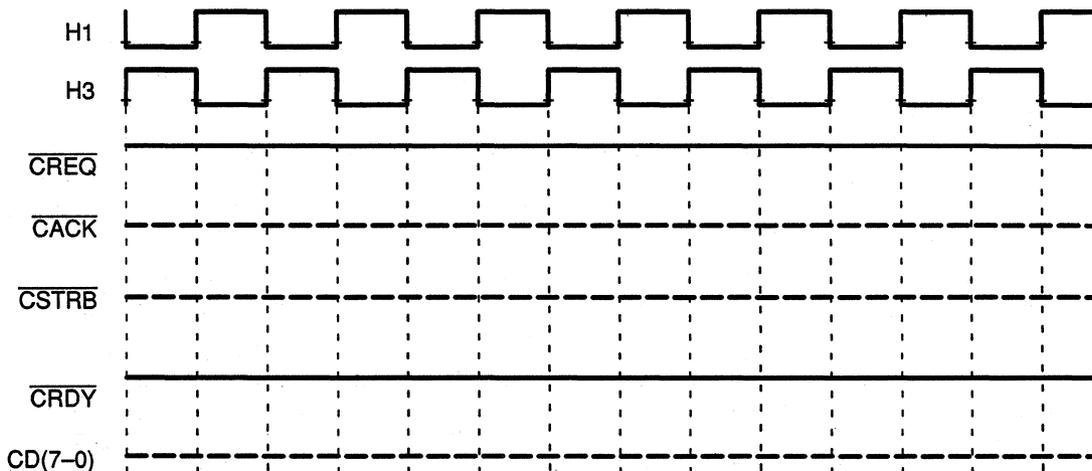


Figure 8–15 illustrates the state of the signals of a communication port initialized by a reset as an input port (ports 3, 4, and 5 are configured as input ports at reset). For this case,  $\overline{\text{CREQ}}$  and  $\overline{\text{CRDY}}$  are high.  $\overline{\text{CACK}}$ ,  $\overline{\text{CSTRB}}$ , and CD(7–0) are all in a high-impedance state.

8

Figure 8–15. Post-Reset State for an Input Port



### 8.7.2 Synchronizer Timing

The synchronizers used in the port arbitration unit are of two types. Type-one synchronizers cause delays that vary from 1 to 2 machine clocks from the receiving of an input on a pin until the response on output pin (ignoring analog delays). Type-two synchronizer delays range from 1.5 to 2.5 machine clocks delay.

**Type-one synchronizers** recognize an input when H1 is high, then pass it through an **H3-high/H1-high series of delays**. The response is at the start of the following H3 high.

The **minimum** type-one synchronizer **delay** of one machine clock will occur when the input changes just before H1 goes low. This delay is shown in Figure 8–16.

The **maximum** type-one synchronizer **delay** of two machine clocks will occur when the input changes just after H1 goes low. This delay is shown in Figure 8–17.

Figure 8–16. Type-One Synchronizer Minimum Delay

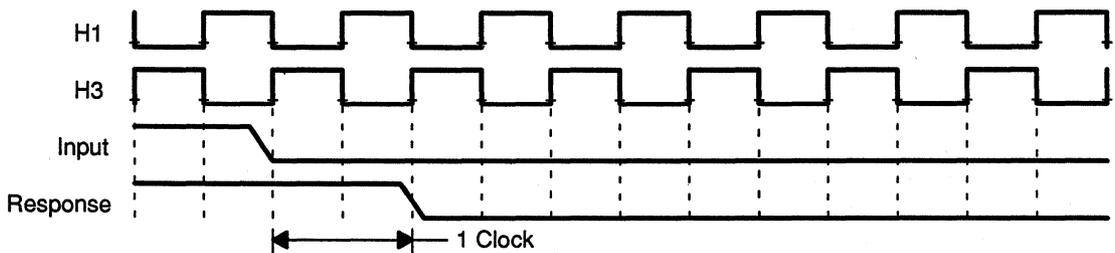
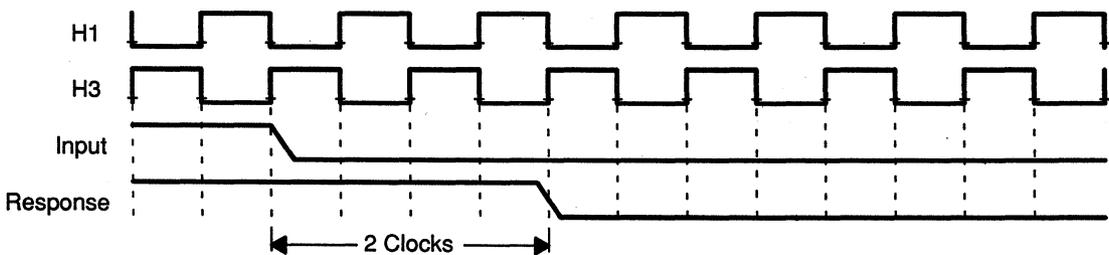


Figure 8–17. Type-One Synchronizer Maximum Delay



**Type-two synchronizers** first recognize an input when H1 is high, then pass it through an **H3-high/H1-high/H3-high series of delays**. The response is at the start of the following H1 high.

The **minimum** type-two synchronizer **delay** of 1.5 machine clocks occurs when the input changes just before H1 goes low. This delay is shown in Figure 8-18.

The **maximum** type-two synchronizer **delay** of 2.5 machine clocks occurs when the input changes just after H1 goes low. This delay is shown in Figure 8-19.

Using these two types of synchronizers, the synchronizer delays for the communication port signals are tabulated in Table 8-5.

Figure 8-18. Type-Two Synchronizer Minimum Delay

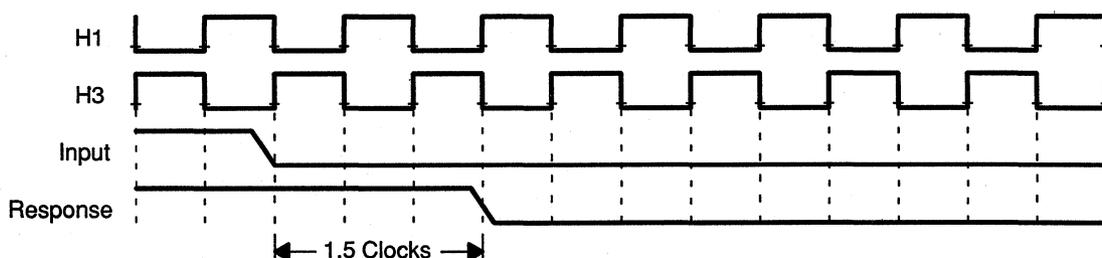


Figure 8-19. Type-Two Synchronizer Maximum Delay

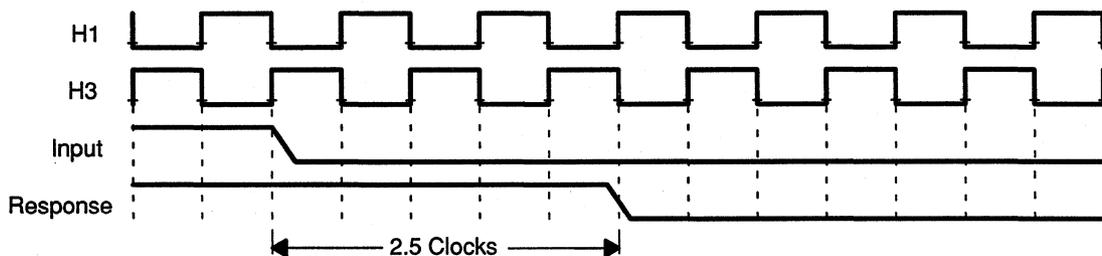


Table 8-5. Communication Port Signals and Synchronizer Delays

Input Signal to Output Signal	Delay Type	Min. Delay (clock cycles)	Max. Delay (clock cycles)
$\overline{\text{CREQ}}\downarrow$ to $\overline{\text{CACK}}\downarrow$	One	1	2
$\overline{\text{CACK}}\downarrow$ to $\overline{\text{CREQ}}\uparrow$	One	1	2
$\overline{\text{CRDY}}\downarrow$ to CD valid for a new word	Two	1.5	2.5
$\overline{\text{CACK}}\downarrow$ to $\overline{\text{CSTRB}}$ active		0.5	1.5
$\overline{\text{CRDY}}\downarrow$ to $\overline{\text{CSTRB}}\downarrow$ between back-to-back word transfers	Two	1.5	2.5

## DMA Coprocessor and 'C40 Timers

This chapter provides technical information for two important TMS320C40 ('C40) functions: the direct memory access (DMA) coprocessor and the timers. Both are on-chip parts of the 'C40 digital signal processor (DSP). The first nine major sections of this chapter cover the DMA coprocessor; the last section covers the timers.

<b>Section</b>	<b>Page</b>
9.1 Introduction .....	9-2
9.2 DMA Coprocessor Functional Description .....	9-3
9.3 DMA Coprocessor Registers .....	9-7
■ DMA Channel Control Register .....	9-7
■ DMA Channel Address and Index Registers .....	9-16
■ DMA Channel Transfer-Counter Register .....	9-18
■ DMA Channel Link-Pointer Register .....	9-19
9.4 DMA Coprocessor Channels in Unified and Split Mode .....	9-20
9.5 DMA Coprocessor Internal Priority Schemes .....	9-22
9.6 CPU and DMA Coprocessor Arbitration .....	9-27
9.7 Data Transfer Modes .....	9-28
9.8 Autoinitialization .....	9-31
9.9 DMA Coprocessor and Interrupts .....	9-40
9.10 TMS320C40 Timers .....	9-45

**Note: DMA Programming Examples in Chapter 12**

Besides the descriptions of DMA operation in this section, programming examples and explanations are provided in Chapter 12.

## 9.1 Introduction

The primary benefit of the DMA coprocessor is to maximize sustained CPU performance by completely alleviating the CPU of burdensome I/O duties.

The DMA coprocessor supports six DMA channels that perform transfers to and from anywhere in the processor's memory map. For example, transfers can be made to/from on-chip memory, off-chip memory, and any of the six on-chip communication ports. The DMA coprocessor can automatically reinitialize its registers via linked lists stored in memory, allowing the DMA to run continuously without any intervention by the central processor unit (CPU). The DMA coprocessor can build up circular buffers in memory and perform linear and bit-reversed addressing.

The DMA coprocessor provides you with an unprecedented level of performance and flexibility for a DSP on-chip DMA coprocessor. The key features of the 'C40 DMA coprocessor are:

- ❑ six DMA channels for memory-to-memory transfers under unified mode; a special split mode supporting 12 DMA channels for communication port to/from memory transfers
- ❑ autoinitialization of DMA channel control registers, via linked lists stored in memory, at the start of a block transfer
- ❑ concurrent CPU and DMA coprocessor operation with DMA transfers at the same rate as the CPU (supported by separate internal DMA address and data buses)
- ❑ source and destination address registers with variable indices allowing stepping through matrices by row or column
- ❑ bit-reversed addressing for FFTs
- ❑ synchronization of data transfers via external and internal interrupts

## 9.2 DMA Coprocessor Functional Description

The TMS320C40 DMA coprocessor improves data transfer rates in systems that must perform:

- memory to memory transfers
- data transfers from an I/O device to memory
- data transfers from memory to an I/O device
- transfers of data between the on-chip communication ports and memory.
- data transfer of a single value to a block of memory for memory fill and initialization.

The DMA coprocessor can transfer data in a linear fashion or in a bit-reversed fashion for FFT applications; it can transfer matrix data in a row or column fashion.

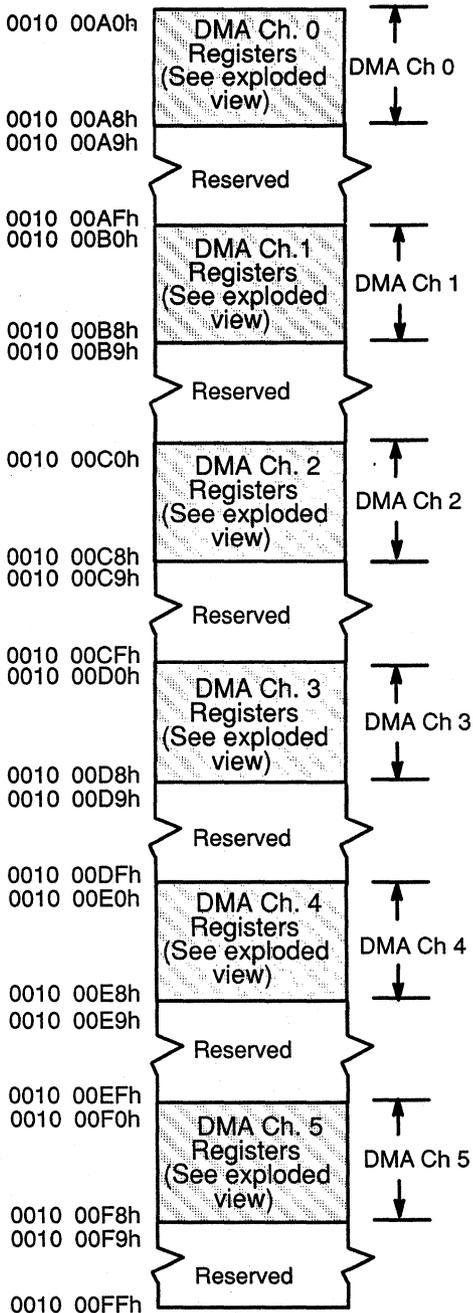
The DMA coprocessor is a self-programming device that allows data transfers to occur *without any intervention from the CPU*. This allows data to be moved onto and off of the 'C40 without any CPU distraction. The result is a processor which has a concurrent I/O rate that can keep up with the CPU's high computation rate. The address map of the DMA coprocessor registers is shown in Figure 9–1. The major registers of the DMA coprocessor are:

- control register
- source register
- source index register
- destination register
- destination index register
- transfer counter register
- link pointer register

Subsections that describe these are listed in Figure 9–2 and in Section 9.3.

The DMA coprocessor has dedicated on-chip DMA address and DMA data buses. All accesses made by the six DMA channels are arbitrated in the DMA coprocessor and take place over these dedicated buses. The DMA channels can run constantly or may be triggered by an external or internal interrupt, including an interrupt generated by the on-chip timers and communication ports.

Figure 9-1. DMA Coprocessor Memory Map



**EXPLODED VIEW OF EACH CHANNEL REGISTER**

010 00z0h	DMA Ch. Control Register x	DMA Ch. x
010 00z1h	Source Address x	
010 00z2h	Source Address Index x	
010 00z3h	Transfer Counter x	
010 00z4h	Destination Address x	
010 00z5h	Destination Address Index x	
010 00z6h	Link Pointer x	
010 00z7h	Auxiliary Transfer Counter x	
010 00z8h	Auxiliary Link Pointer x	

x = channel number (e.g., a 1 for all registers in channel 1, a 2 for all registers in channel 2, etc.).  
 z = corresponding hexadecimal digit for channel address (e.g., substitute an "A" for DMA channel 0; "B" for DMA channel 1, etc.).

**The subsections describing these registers are listed in Figure 9-2 and in Section 9.3 on page 9-7.**

Figure 9–2. Subsections Where DMA Channel Registers Are Described

Memory Address	DMA Register		Described in Section	On Page
010 00z0h	DMA Ch. Control Register x	DMA Ch. x	9.3.1	9-7
010 00z1h	Source Address x		9.3.2	9-16
010 00z2h	Source Address Index x		9.3.2	9-16
010 00z3h	Transfer Counter x		9.3.3	9-18
010 00z4h	Destination Address x		9.3.2	9-16
010 00z5h	Destination Address Index x		9.3.2	9-16
010 00z6h	Link Pointer x		9.3.4	9-19
010 00z7h	Auxiliary Transfer Counter x		9.3.3	9-18
010 00z8h	Auxiliary Link Pointer x	9.3.4	9-19	

x = channel number (e.g., all are 1 for channel 1, all 2 for channel 2, etc.).

z = corresponding hexadecimal digit for channel address (e.g., substitute "A" for DMA channel 0; "B" for DMA channel 1, etc. See Figure 9–1).

For example, if a block of data is to be transferred from one region in memory to another region in memory:

- 1) The **source address register** of a DMA channel is loaded with the address of the source memory location.
- 2) The **destination address register** of the same DMA channel is loaded with the address of the destination memory location.
- 3) The **transfer counter** is loaded with the number of words to be transferred.
- 4) If sequential memory accesses are required, the **source address index register** as well as the **destination address index register** would be set to 1.
- 5) The appropriate modes can be set up to synchronize the DMA coprocessor reads and writes to interrupts via the DMA channel control register.
- 6) Then, the DMA coprocessor can be started via the DMA START field in the DMA channel control register.

#### A DMA transfer consists of two steps:

- 1) The source data value is **read** by the DMA channel and stored in a temporary register.
- 2) The temporary register value is **written** to the destination address.

During every data write, the transfer counter is decremented. The block transfer can be terminated when the transfer counter goes to zero *and* the write of the last transfer is complete.

After a read by the DMA channel, the source-index register is added to the source-address register. After a write by the DMA channel, the destination-

index register is added to the destination-address register. (Both index registers contain signed values.) This allows for variable step sizes or continual reads and/or writes from/to memory. In the case of an index register equaling zero, the DMA coprocessor transfers data from/to a fixed location.

**At the completion of a block transfer**, the DMA coprocessor can be programmed to do several things:

- most importantly, autoinitialize itself at the start of the next block transfer. Each DMA channel can read new control register values from memory (as well as the other registers in Figure 9–2), load these values into its register block, and, according to the values loaded, begin another block transfer. *This autoinitialization is done without any intervention by the CPU.*
- generated an interrupt to signal that the block transfer is complete
- stop until reprogrammed

A special split-mode allows the DMA channels to have the source and destination paths split and bound to a communication port. In this mode, the **DMA-channel source path** (source-address register, source-index register, transfer-counter register, and link-pointer register) forms the **primary** split channel and is used to move data from a location in the processor's memory map to a communication port. The **DMA-channel destination path** (destination-address register, destination-index register, auxiliary transfer-counter register, and auxiliary link-pointer register) is the **auxiliary** split channel and is used to move data from the same communication port to a location in the processor's memory map.

**Note: DMA Coprocessor Programming Examples in Chapter 12**

Besides the descriptions of DMA coprocessor operation in this section, programming examples and explanations are provided in Chapter 12.

## 9.3 DMA Coprocessor Registers

The DMA coprocessor has nine registers designated as follows (for location, see Figure 9–2 on page 9-5):

- ❑ DMA channel control register (subsection 9.3.1)
- ❑ DMA-channel source-address register (subsection 9.3.2, page 9-16 )
- ❑ DMA-channel source-address-index register (subsection 9.3.2, page 9-16 )
- ❑ DMA-channel destination-address register (subsection 9.3.2, page 9-16 )
- ❑ DMA-channel destination-address-index register (subsection 9.3.2, page 9-16 )
- ❑ DMA-channel transfer-count register (subsection 9.3.3 on page 9-18)
- ❑ DMA-channel auxiliary-transfer-count register (subsection 9.3.3 on page 9-18)
- ❑ DMA-channel link-pointer register (subsection 9.3.4 on page 9-19)
- ❑ DMA-channel auxiliary-link-pointer register (subsection 9.3.4 on page 9-19)

Each DMA channel has one of each of these registers, discussed in the following paragraphs.

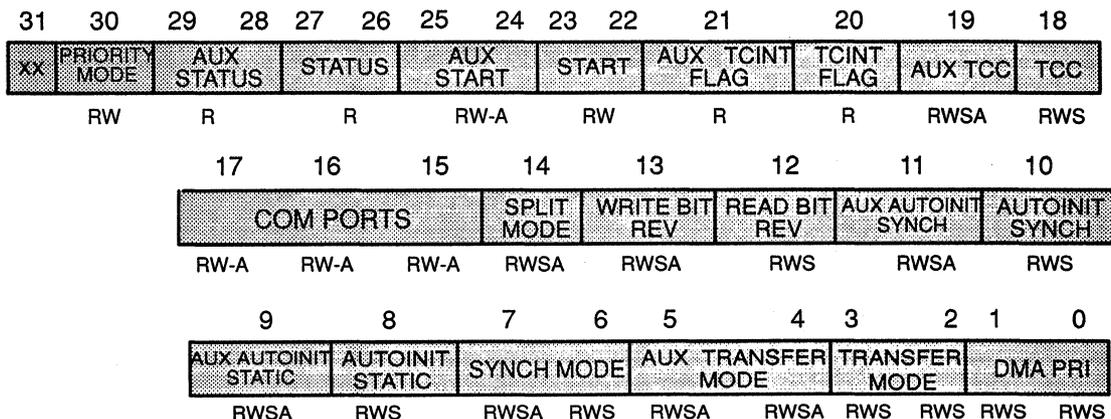
### 9.3.1 DMA Channel Control Register

The format of the DMA channel control register is shown in Figure 9–3. Table 9–1 defines the register bits, the bit names, and the bit functions.

At reset, the DMA channel control register is set to zero. This makes the DMA channel lower priority than the CPU, sets up the source address and destination address to be calculated via linear addressing, and configures the DMA channel in the unified mode.

When an external interrupt is used for DMA coprocessor transfer synchronization, the CPU is responsible for configuring external interrupts as edge- or level-triggered interrupts (as set in the applicable FUNCx and TYPEx bits of the interrupt flag register (subsection 3.1.10 on page 3-12)).

Figure 9-3. DMA Channel Control Register



- R – Bit may be read.
- W – Bit may be written.
- S – Bit is shadowed during autoinitialization (no changes take affect until autoinitialization is complete.)
- A – Bit is auxiliary for autoinitialization.
- xx – Reserved.

Table 9-1. DMA Channel Control Register Bit Definitions

Bit Nos.	Mnemonic	Read/Write	Description
0 – 1	DMA PRI	R/W	DMA coprocessor priority. Defines the arbitration rules to be used when a DMA channel and the CPU are requesting the same resource. Affects all DMA coprocessor modes. Rules listed in <b>Table 9-2, page 9-14.</b>
2 – 3	TRANSFER MODE	R/W	Defines the transfer mode used by the DMA channel. Affects unified mode and the primary channel in split mode. Bits defined in <b>Table 9-3 on page 9-14.</b>
4 – 5	AUX TRANSFER MODE	R/W	Defines the transfer mode used by the DMA channel. Affects the <i>auxiliary</i> channel in split mode only. Bits defined in <b>Table 9-3 on page 9-14.</b>

Table continued on next page

Table 9-1. DMA Channel Control Register Bit Definitions (Continued)

Bit Nos.	Mnemonic	Read/Write	Description
6–7	SYNC MODE	R/W	Determines the mode of synchronization to be used when performing data transfers, as shown in <b>Table 9-4 on page 9-15</b> . <b>Note:</b> If a DMA channel is interrupt driven for both reads and writes, <i>and</i> the interrupt for the write comes before the interrupt for the read, the interrupt for the write is latched by the DMA channel. After the read is complete, the write can be executed.
8	AUTOINIT STATIC	R/W	<b>If bit = 0</b> , the link pointer is incremented during autoinitialization. <b>If bit = 1</b> , the link pointer is <i>not</i> incremented (it is static) during autoinitialization. This affects unified mode and primary channel in split mode. It is useful to keep the auxiliary link pointer constant when autoinitializing from the on-chip communication ports or other stream-oriented devices (such as first-in first-out (FIFO) memory buffers).
9	AUX AUTOINIT STATIC	R/W	Acts the same as for the AUTOINIT STATIC mode above, except that this affects the auxiliary channel in split mode only.
10	AUTOINIT SYNC	R/W	Has an effect only in the DMA coprocessor sync mode (bits 6–7 above). Affects the interrupt that is enabled by the DMA interrupt enable register (see Figure 3-4, page 3-8) used for DMA reads: <b>If bit = 0</b> , the interrupt is ignored, and the autoinitialization reads are not synchronized with any interrupt signals. <b>If bit = 1</b> , then the interrupt is recognized and is also used to synchronize the autoinitialization reads. This affects the unified mode and the primary channel in split mode (see bit 14, SPLIT MODE). The effect of this bit and the SYNC MODE bit in autoinitialization is summarized in Table 9-9 on page 9-37.
11	AUX AUTOINIT SYNC	R/W	Acts the same as the AUTOINIT SYNC bit above except that it affects DMA-coprocessor write autoinitialization sync in unified mode and the auxiliary channel in split mode. The effect of this bit and the SYNC MODE bits in autoinitialization is summarized in Table 9-9 on page 9-37.

Table continued on next page

Table 9-1. DMA Channel Control Register Bit Definitions (Continued)

Bit Nos.	Mnemonic	Read/Write	Description
12	READ BIT REV	R/W	<p>If bit = 0, the source address is modified using 32-bit linear addressing.</p> <p>If bit = 1, the source address is modified using 24-bit bit-reversed addressing.</p> <p>Affects unified mode and primary channel in split mode.</p>
13	WRITE BIT REV	R/W	<p>If bit = 0, the destination address is modified using 32-bit linear addressing.</p> <p>If bit = 1, the destination address is modified using 24-bit bit-reversed addressing.</p> <p>Affects unified mode and auxiliary channel in split mode.</p>
14	SPLIT MODE	R/W	<p>This controls the DMA coprocessor mode of operation.</p> <p>If bit = 0, DMA transfers are memory to memory. This is referred to as <b>unified mode</b>.</p> <p>If bit = 1, <b>split mode</b> is entered with the DMA split into two channels, allowing a single DMA channel to perform memory-to-communication-port and communication-port-to-memory transfers.</p> <p>The split mode may be modified by autoinitialization in unified mode or by autoinitialization by the auxiliary channel in split mode. Split mode is further described in Section 9.4.</p>
15 – 17	COM PORT	R/W	<p>These bits define a communication port (000<sub>2</sub> to 101<sub>2</sub>) to be used for DMA transfers.</p> <p>If <b>SPLIT MODE = 0</b>, then COM PORT has no affect on the operation of the DMA channel.</p> <p>If <b>SPLIT MODE = 1</b>, then COM PORT defines which of the six communication ports to use with the DMA channel.</p> <p>The COM PORT may be modified by autoinitialization in unified mode or by autoinitialization by the auxiliary channel in split mode.</p>

Table continued on next page

Table 9-1. DMA Channel Control Register Bit Definitions (Continued)

Bit Nos.	Mnemonic	Read/Write.	Description
18	TCC	R/W	<p>Transfer counter interrupt control.</p> <p><b>If TCC = 1</b>, a DMA channel interrupt pulse is sent to the CPU after the transfer counter makes a transition to zero and the write of the last transfer is complete. If enabled, the corresponding DMA interrupt (DMA INT0—INT5) occurs as shown in Figure 3-8, p. 3-16.</p> <p><b>If TCC = 0</b>, a DMA channel interrupt pulse is not sent to the CPU when the transfer counter makes a transition to zero.</p> <p>Affects unified mode and the primary channel in split mode. DMA channel interrupts to the CPU are <i>edge triggered</i>.</p>
19	AUX TCC	R/W	<p>Auxiliary transfer counter interrupt control.</p> <p><b>If bit = 1</b>, a DMA channel interrupt pulse is sent to the CPU after the auxiliary transfer counter makes a transition to zero and the write of the last transfer is complete. If enabled, the corresponding DMA interrupt (DMA INT0—INT5) occurs as shown in Figure 3-8, p. 3-16.</p> <p><b>If bit = 0</b>, a DMA channel interrupt pulse is not sent to the CPU when the auxiliary transfer counter makes a transition to zero.</p> <p>Affects the auxiliary channel in split mode only.</p>
20	TCINT FLAG	R	<p>Transfer counter interrupt flag.</p> <p>This flag is set to 1 whenever the transfer counter makes a transition to zero and the write of the last transfer is completed. Whenever the DMA channel control register is read, this flag is cleared <i>unless</i> the flag is being set by the DMA in the same cycle as the read (in such case, TCINT is not cleared).</p> <p>The TCINT FLAG is affected by the unified mode and the primary channel in split mode.</p>

Table continued on next page

Table 9-1. DMA Channel Control Register Bit Definitions (Continued)

Bit Nos.	Mnemonic	Read/Write	Description
21	AUX TCINT FLAG	R	<p>Auxiliary transfer counter interrupt flag. This flag is set to 1 whenever the auxiliary transfer counter makes a transition to zero and the write of the last transfer is completed. Whenever the DMA control register is read, this flag is cleared <i>unless</i> the flag is being set by the DMA coprocessor in the same cycle as the read (in such case AUX TCINT is not cleared). The AUX TCINT FLAG is affected by the auxiliary channel in split mode.</p> <p><i>Since only one DMA-channel interrupt is available for a DMA channel, you can determine what event had set the interrupt by examining the TCINT FLAG and the AUX TCINT FLAG.</i></p>
22–23	START	R/W	<p>Starts and stops the DMA channel in several different ways (listed in <b>Table 9-5, page 9-15</b>). START affects the unified mode and the primary channel in split mode. If used to hold a channel in the middle of an autoinit sequence, the START and AUX START bits will hold the autoinit sequence.</p> <p>If the START or AUX START bits are being modified by the DMA channel (for example, to force a halt code of 10<sub>2</sub> on a transfer-counter terminated block transfer) and a write is being performed by an external source to the DMA channel control register, <b>internal modification of the START or AUX START bits by the DMA channel has priority</b>. See TRANSFER MODE bits value of 0 1<sub>2</sub>, (Table 9-3,).</p>
24–25	AUX START	R/W	<p>Starts and stops the DMA channel in several different ways (listed in <b>Table 9-5, page 9-15</b>) AUX START affects the auxiliary channel in split mode only.</p>
26–27	STATUS	R	<p>Indicates the status of the DMA channel as listed in <b>Table 9-6, page 9-16</b>. STATUS is updated in the unified mode and by the primary channel in the split mode. Updates are done every cycle.</p> <p>The STATUS and AUX STATUS bits (Table 9-6) are used to determine the current status of the DMA channels and to determine if the DMA channel has halted or has been reset after writing to the START or AUX START bits.</p>

Table concluded on next page

Table 9-1. DMA Channel Control Register Bit Definitions (Concluded)

Bit Nos.	Mnemonic	Read/ Write	Description
28–29	AUX STATUS	R	Indicates the status of the DMA channel as listed in <b>Table 9-6, page 9-16</b> . STATUS is updated by the auxiliary channel in split mode only. Updates are done every cycle.
30	PRIORITY MODE	R/W	Priority mode of DMA channel access: <b>0 = Rotating</b> priority as shown in <b>Section 9.5 (on page 9-22)</b> . <b>1 = Fixed</b> priority as shown in Section 9.5. This <b>bit is available only</b> at DMA channel 0 (zero).
31			Reserved.

Table 9-2. DMA PRI Bits and CPU/DMA Arbitration Rules

DMA PRI Bit Nos: 1 – 0	Effect
0 0	DMA coprocessor access is <i>lower</i> priority than the CPU access. If the DMA channel and the CPU are requesting the same resource, then the CPU will proceed. Bits are set this way at reset.
0 1	If the DMA channel and the CPU are requesting the same resource, then the <i>CPU will proceed</i> . Then, <b>after</b> the CPU access is <b>complete</b> , if the DMA coprocessor and CPU are <b>again</b> requesting the same resource, the <i>DMA coprocessor will proceed</i> . This priority rule provides a fair arbitration scheme by alternating CPU accesses with a DMA channel's access.
1 0	Reserved.
1 1	DMA coprocessor access is <i>higher</i> priority than the CPU access. If the DMA channel and the CPU are requesting the same resource, then the DMA will proceed.

Table 9-3. TRANSFER MODE and AUX TRANSFER MODE Field Description

TRANSFER MODE Bit Nos: 3 – 2 5 – 4	Effect
0 0	Transfers are <i>not</i> terminated by the transfer counter, and <i>no</i> autoinitialization is performed. TCINT (transfer counter interrupt) can still be used to cause an interrupt when the transfer counter makes a transition to zero. The DMA channel continues to run. Note that the address continues incrementing while the transfer count rolls over to its maximum value of 0FFFF FFFFh.
0 1	Transfers are terminated by the transfer counter. No autoinitialization is performed. A halt code of 10 <sub>2</sub> is placed in the START field when transfers are completed.
1 0	Autoinitialization is performed when the transfer counter goes to zero without waiting for CPU intervention.
1 1	The DMA channel is autoinitialized when the CPU restarts the DMA coprocessor by using the DMA register in the CPU. When the transfer counter goes to zero, operation is halted until the CPU starts the DMA coprocessor by using the START field in the DMA channel control register (bits 22–23 and 24–25, Table 9-5). A halt code of 10 <sub>2</sub> is placed in the START field by the DMA coprocessor.

Table 9-4. SYNCH MODE Field Description

SYNCH MODE Bit Nos: 7 – 6	Effect
0 0	No synchronization. Interrupts are ignored.
0 1	Source synchronization. A read will not be performed until an enabled interrupt occurs.
1 0	Destination synchronization. A write will not be performed until an enabled interrupt occurs.
1 1	Source and destination synchronization. A read is performed when an enabled interrupt occurs. Then, a write is performed when an enabled interrupt occurs. The interrupts used are specified by the DMA READ and DMA WRITE fields of the DMA interrupt enable (DIE) register (subsection 3.1.8 on page 3-8).

Table 9-5. START and AUX START Field Description

START Bit Nos: 23 – 22 25 – 24	Effect
0 0	DMA channel reset. DMA channel read or write cycles in progress are completed (not aborted); any data read is ignored. Any pending (not started) read or write is canceled. The DMA channel is reset so that when it starts, a new transaction begins; that is, a read is performed. In this start mode, stopping is immediate with no other registers loaded.
0 1	Halts the DMA channel on the first available read or write boundary. If the read or write has begun, the read or write is completed before stopping (i.e., in the middle or at the end of a DMA channel transfer). If a read or write has not begun, no read or write is started. In this start mode, stopping is immediate with no other registers loaded).
1 0	Halts the DMA channel on the first available transfer boundary. If a DMA transfer has begun, the entire transfer is completed, including both cycles (both read and write operations), before stopping. If a transfer has not begun, none is started. In this start mode, stopping is immediate with no other registers loaded.
1 1	DMA start. Writing $11_2$ to this field starts the DMA process using the values in the channel's DMA channel registers (Figure 9-1). If the DMA is in auto-initialization, all DMA registers are loaded before starting the operation. The DMA coprocessor starts from reset if previously reset (START bits = $00_2$ ) or restarts from the previous state if previously halted (START bits = $01_2$ or $10_2$ ).

Table 9–6. STATUS and AUX STATUS Field Description

STATUS Bit Nos: 27 – 26 29 – 28	Meaning
0 0	DMA process is in the middle of the DMA transfer (between the write and read operations). This is the value at RESET, after a halt on a transfer boundary, or after a block transfer.
0 1	DMA process is being held (for any reason) in the middle of a DMA transfer; that is, in the middle of the read/write operation.
1 0	Reserved.
1 1	DMA channel is not being held or reset.

### 9.3.2 DMA Channel Address and Index Registers

As shown in Figure 9–4, both the DMA coprocessor source-address and destination-address registers have an associated index register. After each DMA channel read (source address) or write (destination address), the corresponding (source or destination) address generator adds the index register to the address register *and places the result in the address register*. In this way, the address register acts as accumulator because it retains the sum of itself and its index register.

#### Address Register + Index Register → Address Register

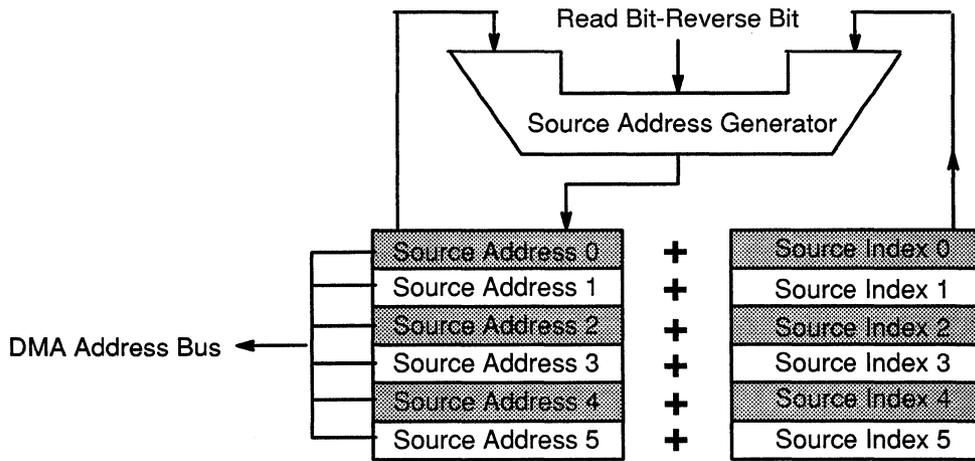
The values in these registers are undefined at reset.

Depending upon bits 12 and 13 (READ BIT REV and WRITE BIT REV) of the DMA channel control register, the addition may be either:

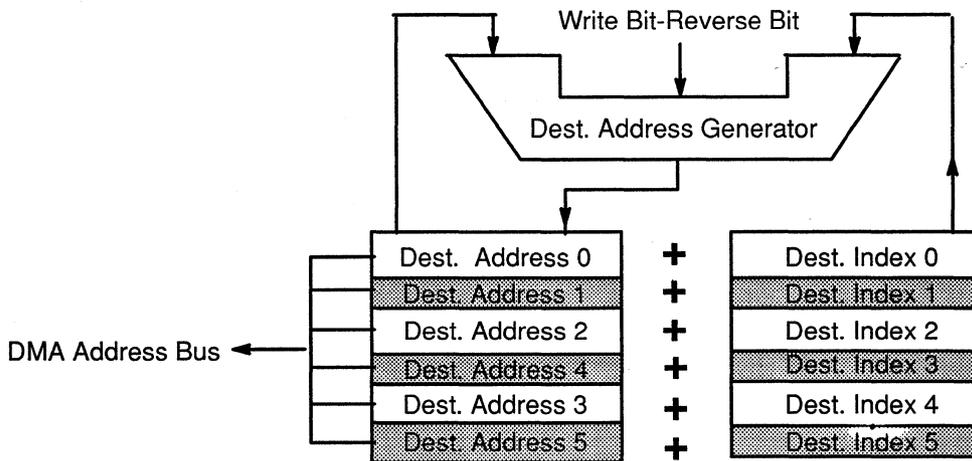
- linear** (normal addition): READ BIT REV = 0 or WRITE BIT REV = 0, or
- bit reversed** (reverse carry propagation): READ BIT REV = 1 or WRITE BIT REV = 1.

Both index values (source or destination) are *signed* values.

Figure 9-4. DMA-Coprocessor Address Generation



(a) Source Address Register Operation



(b) Destination Address Register Operation

### 9.3.3 DMA Channel Transfer-Counter and Auxiliary-Transfer-Count Registers

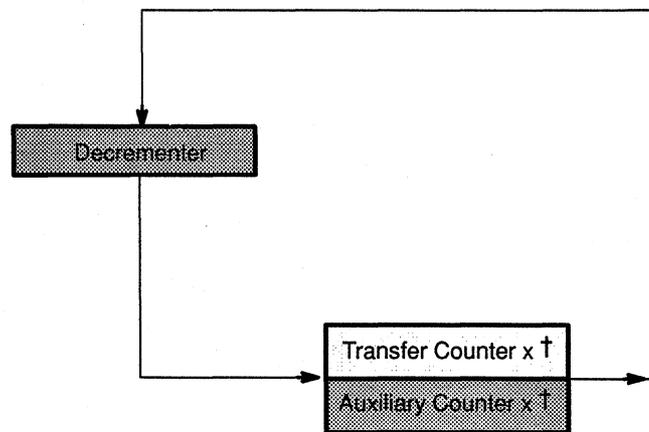
These registers contain the number of words to be transmitted.

Figure 9–5 shows the six transfer counters and the six auxiliary transfer counters. Each auxiliary transfer counter is used when the DMA channel is in split mode (described in Section 9.4 on page 9-20). The values in these registers are set to zero at reset.

The counters are decremented after completing the address fetch for the write portion of a transfer. The TCINT FLAG and AUX TCINT FLAG (bits 20 and 21 of the DMA channel control register, Figure 9–3 on page 9-8) are not set *until* the counter is decremented and the write of the last transfer is completed. Correspondingly, the interrupt will not be seen by the CPU interrupt controller until the transfer counter is decremented and the write of the last transfer is completed.

The decremter checks for equality with zero after the decrement is performed. As a result, if the count register has a value of 1, then the DMA channel can be halted after only one transfer is performed. The count is treated as an unsigned integer. *Transfers may be halted when a zero count is detected after a decrement. If the DMA coprocessor channel is not halted after the transfer reaches zero, the counter will continue decrementing below zero.*

Figure 9–5. DMA Coprocessor Transfer-Count Registers



† x = DMA channel number (0–5)

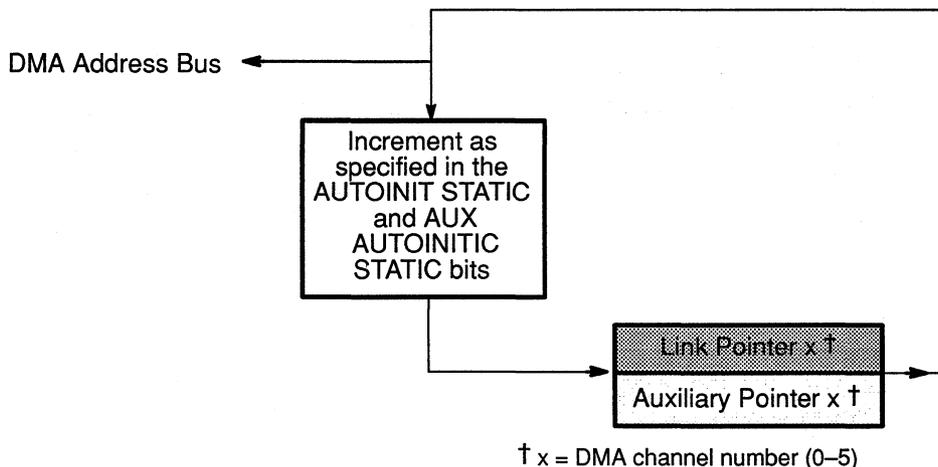
### 9.3.4 DMA-Channel Link-Pointer and Auxiliary-Link-Pointer Registers

The link pointers specify the address from which to load the new DMA channel register values when autoinitialization is performed. When a channel has exhausted its counter, it will (if appropriately configured) use the link pointer to reload itself. Figure 9–6 illustrates the DMA coprocessor link address registers. The values in these registers are undefined at reset.

For example, under autoinitialization, the steps to load the channel registers for DMA channel 0 (as shown in Figure 9–1 on page 9-4) would be:

- 1) Get link pointer for next DMA operation. Pointer is memory address containing contents of **first** DMA channel 0 register (channel control register as shown in Figure 9–1 on page 9-4).
- 2) Bring in contents pointed to by pointer and write to address 010 00A0h (first word of DMA channel 0 registers as shown in Figure 9–1).
- 3) Increment link pointer. (Skip this step if AUTOINIT STATIC bit = 1.)
- 4) Bring in next word and write to address 010 00A1h.
- 5) Repeat until entire block of registers is loaded for DMA channel 0.

Figure 9–6. DMA Coprocessor Link Pointer Registers



## 9.4 DMA Channels in Unified and Split Modes

Unified and split mode are depicted in separate diagrams (Figure 9–7 and Figure 9–8 on the next page). The split mode transforms one DMA channel into two DMA channels:

- ❑ **Primary Channel:** one dedicated to reading data from a location in the memory map (external/internal) and writing it to a communication port
- ❑ **Auxiliary Channel:** one dedicated to receiving data from a communication port and writing it to a location in the memory map

To accommodate the six communication ports, all six DMA channels can support this split mode (DMA channels 0–5).

The SPLIT MODE bit (bit 14 of the DMA channel control register, Figure 9–3) controls the DMA unified or split mode:

- ❑ For **unified mode** (Figure 9–7): Set SPLIT MODE bit to 0 (zero)
- ❑ For **split mode** (Figure 9–8): Set SPLIT MODE bit to 1

The COM PORT field of the DMA channel control register (bits 15–17 as shown in Figure 9–3) defines which communication port is used (port 0–5). Figure 9–8 shows typical operations using one communication port.

- ❑ The **transfer counter register** controls the primary channel transfers.
- ❑ The **auxiliary transfer counter register** controls the auxiliary channel transfers (both these registers shown in Figure 9–1, page 9-4).

DMA channel arbitration in split mode is described in subsection 9.5.3 on page 9-24.

Figure 9-7. Typical Unified Mode DMA Channel Configuration

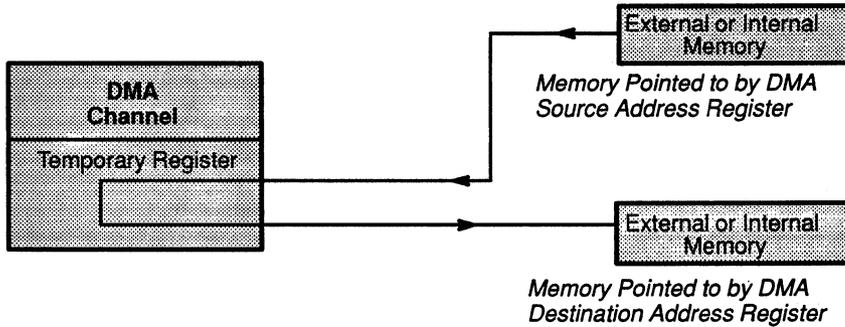
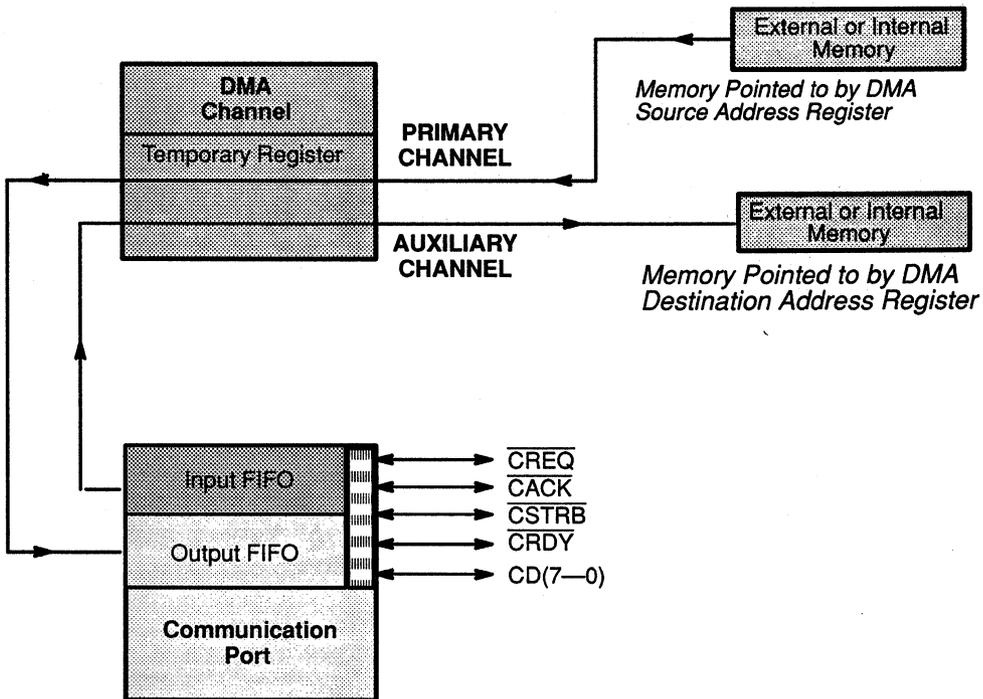


Figure 9-8. Typical Split-Mode DMA Configuration



## 9.5 DMA Coprocessor Internal Priority Schemes

Within the DMA coprocessor, two priority schemes are used to designate which channel is serviced next:

- a fixed priority scheme with channel 0 always having the highest priority and channel 5 the lowest,
- a rotating priority scheme which places the just-serviced channel at the bottom of the priority list.

Select the desired scheme by setting bit 30 (PRIORITY MODE) of DMA **channel 0's** DMA channel control register (Figure 9–3 and Table 9–1 on page 9-8):

- PRIORITY MODE = 0 = rotating priority
- PRIORITY MODE = 1 = fixed priority

### 9.5.1 Fixed Priority Scheme

This scheme provides a fixed priority (unchanging) for each channel as follows:

Highest priority	<b>0</b>
	<b>1</b>
	<b>2</b>
	<b>3</b>
	<b>4</b>
Lowest priority	<b>5</b>

To set up this scheme, set the PRIORITY MODE bit (bit 30) of **channel 0's** DMA channel control register to 1 (one).

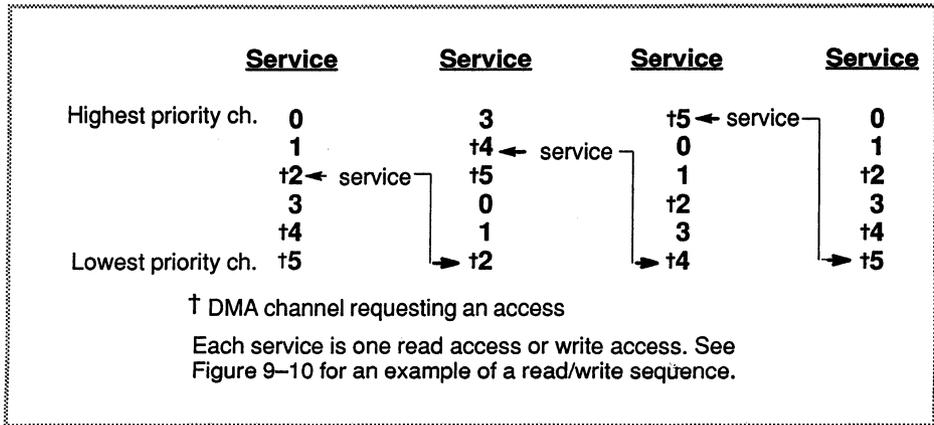
### 9.5.2 Rotating Priority Scheme

In a rotating priority scheme, the last channel serviced becomes the lowest priority channel. The other channels sequentially rotate through the priority list with the next lowest channel from the just-serviced channel becoming the highest priority on the following request. The priority rotates every time the most-recent priority-granted channel completes its access. Figure 9–9 and Figure 9–11 illustrate the rotation of priority across several DMA coprocessor accesses. At system reset, the channels are ordered from highest to lowest priority (0, 1, 2, 3, 4, 5).

To set up this scheme, set the PRIORITY MODE bit (bit 30) of **channel 0's** DMA control register to 0 (zero).

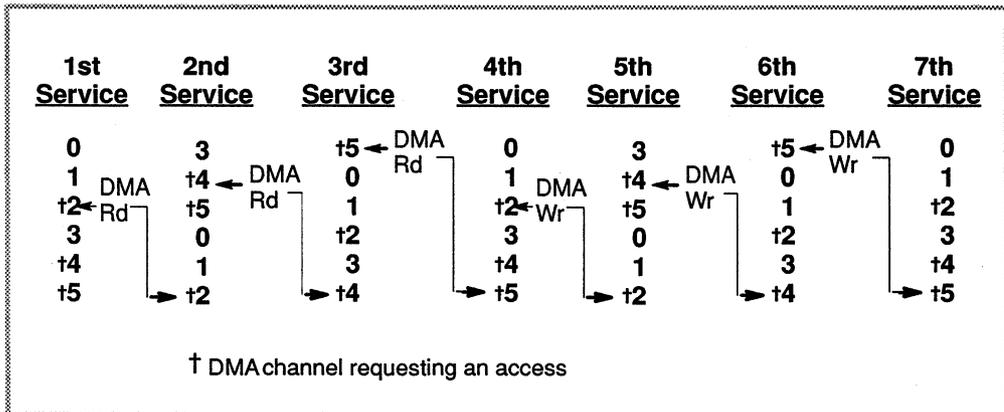
The DMA coprocessor handles channel arbitration on an access-by-access basis; that is, a DMA channel must contend for both the read and the write access in unified mode.

Figure 9-9. Rotating Priority Mode Example of the DMA Coprocessor



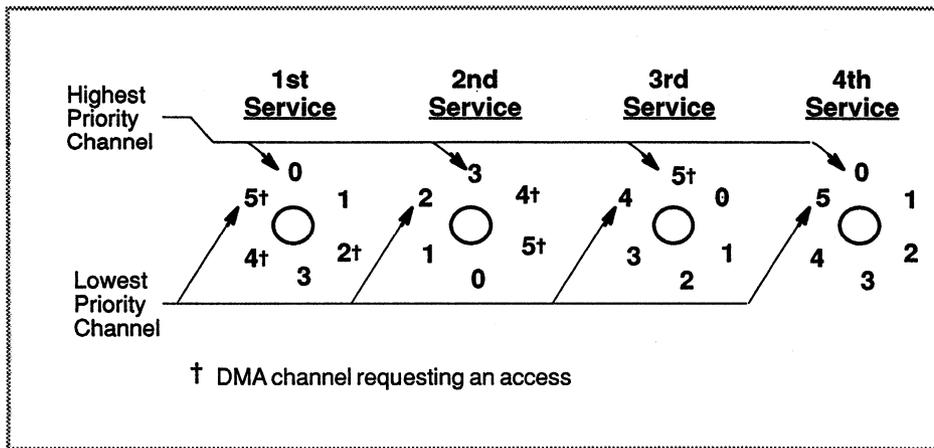
At the start of the example in Figure 9-9, channels 2, 4, and 5 are requesting service. Since channel 2 has the highest priority, it is serviced first. It then becomes the lowest priority channel. The highest priority channel then becomes channel three. On the following services, channels 4 and 5 are taken care of in a similar fashion. Each service means one read access or one write access. Figure 9-10 shows the entire read and write sequence.

Figure 9-10. DMA Read and Write Sequence Example



Another way to visualize the rotation of priorities is shown in Figure 9-11. This example shows the same results as in Figure 9-9. It helps to make clear the rotating nature of the priority scheme. Priority decreases from highest to lowest in a clockwise direction. The priority rotates in a counter clockwise direction with the most recently serviced channel ending up in the lowest priority position.

Figure 9–11. Example of a Priority Wheel



With the rotating priority scheme, any DMA channel requesting service is guaranteed to be recognized after a number of higher priority requests have been serviced. The maximum number of requests are:

- five in unified mode
- eleven in split mode

This provides a fair means of preventing a channel from monopolizing the system.

DMA channels that are running and are not synchronized via interrupts are always requesting service.

### 9.5.3 Split Mode and DMA Channel Arbitration

When a DMA channel is running in split mode, arbitration between channels is similar to that just discussed. The split-mode DMA channel has the same priority as the unified DMA channel. The only issue is how to arbitrate between the primary split channel and the auxiliary split channel. Both split channels alternate priorities via a rotating priority scheme between each other.

When a DMA channel is in split mode and both paths are simultaneously reset via the START and AUX START bits, the output (primary) channel has priority over the input (auxiliary) channel. Both the START and AUX START bits must be written at the same time in order to achieve this reset condition.

The priority scheme for channels is slightly different than the scheme for primary and auxiliary channels within a channel:

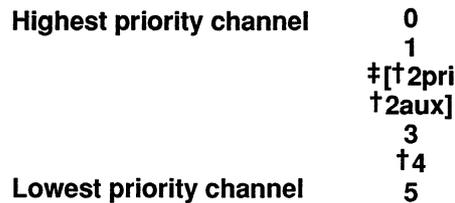
- for channels, priority changes after a read or a write
- for the primary and auxiliary channels within a channel, priority changes after a complete read and write.

Figure 9–12 is an example of two channels contending for the DMA bus: channel 2 (a split channel) and channel 4. In this case:

- only channel 2 (i.e., *not* channel 4) is being run in split mode
- its primary channel is identified as 2pri
- its auxiliary channel is identified as 2aux
- the paths requesting service are identified with a †

In the example described below, channel 4 will do one complete transfer (read and write) for each complete transfer of either channel 2pri or 2aux.

Figure 9–12. Example of a Channel Priority Scheme in Split Mode



† DMA channel requesting an access

‡ Split channels requesting access

2pri = the primary split channel of channel 2

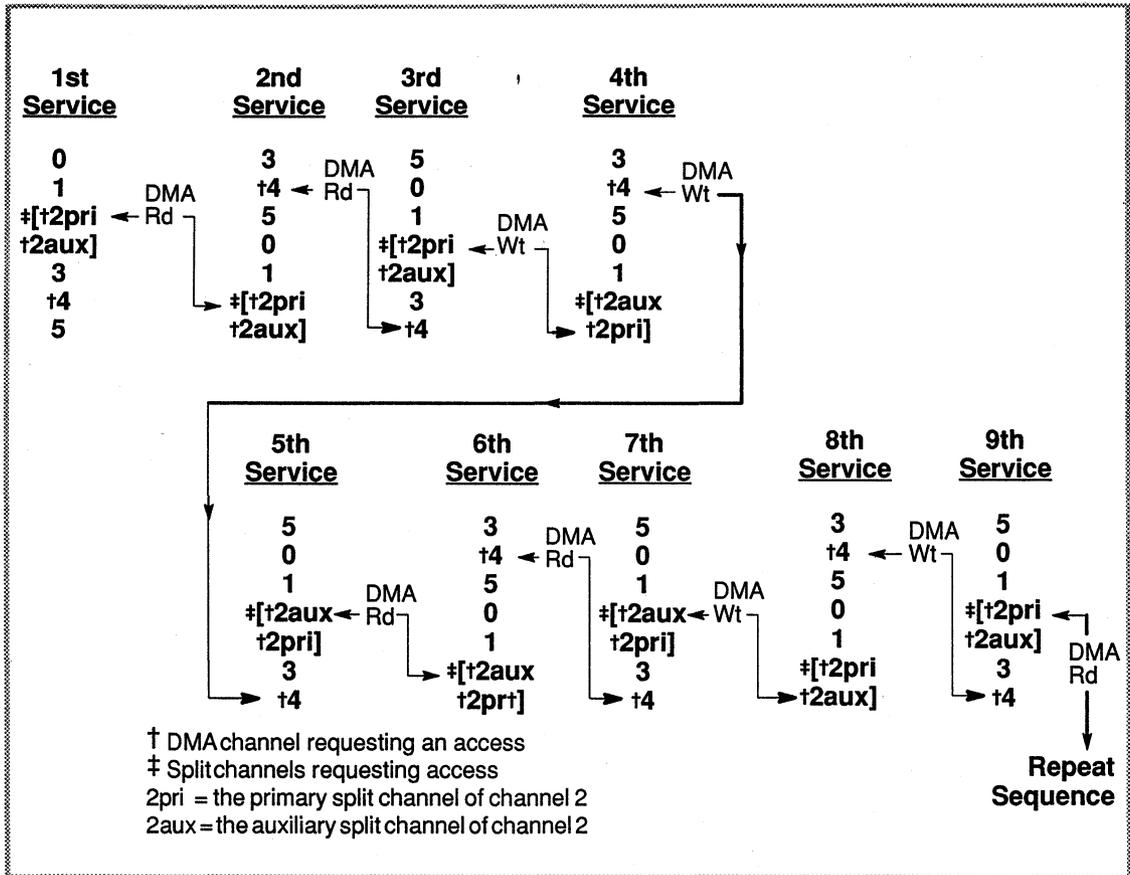
2aux = the auxiliary split channel of channel 2

The channel priority scheme in Figure 9–12 is further shown sequentially in Figure 9–13 (on the next page):

- 1) The **first** service is a request by the primary split channel of channel 2 (2pri). 2pri reads, and then channel 2 is moved to the lowest priority level, but 2pri remains the higher priority channel of channel 2.
- 2) On the **second** service, channel 4, now a higher priority than channel 2, reads its source address.
- 3) On the **third** service, the value read by 2pri is written to its destination address, and channel 2 is moved to the lowest priority level. Also, 2pri is moved to a lower priority than 2aux, channel 2's auxiliary channel. Note that the split channel that just completed a read retains a higher priority than the other split channel until the data is written to the destination address.

- 4) On the **fourth** service, the value read by channel 4 in service 2 is now written to its destination address and the channel becomes the lowest priority.

Figure 9-13. Service Sequence for Split Mode Priority Example



- 5) In the **fifth** service, 2aux is read and channel 2 becomes the lowest priority.
- 6) On the **sixth** service, channel 4 is read again, and it becomes the lowest priority.
- 7) On the **seventh** and **eighth** services, the 2aux and channel 4 values that were read in services 5 and 6 are now written to their destination addresses. After the channel is written, it assumes the lowest priority.
- 8) In the **ninth** service, 2pri is read again as in the **first** service, and the read/write cycle continues as begun in the **first** service.

## 9.6 CPU and DMA Coprocessor Arbitration

The DMA coprocessor has its own internal buses for transferring data. Only when a resource conflict exists between the DMA coprocessor and the CPU is it necessary for arbitration between these two.

When the CPU and DMA coprocessor arbitrate for memory access, the memory address along with the channel's DMA PRI bits (bits 0 and 1 of the channel control register) are used in this arbitration scheme. These bits are described in Table 9–7 below. Higher priority DMA channels will be serviced before lower priority DMA channels whose requested address does not conflict with a CPU access or who have higher priority than the CPU.

The DMA PRI bits of the channel control register (of the DMA channel arbitrating with the CPU) define the arbitration rules. These rules apply whenever the CPU and the highest priority requesting channel request the same resource. Otherwise, the CPU and DMA coprocessor access may proceed in parallel.

All arbitration between the CPU and the DMA coprocessor is on an access basis; that is, the DMA coprocessor must contend for the read and the write accesses of a DMA transfer in unified mode and split mode.

Table 9–7. DMA PRI Bits and CPU/DMA Arbitration Rules

DMA PRI (Bits 1–0)	Effect
0 0 <sub>2</sub>	DMA access is lower priority than the CPU access. If the DMA channel and the CPU are requesting the same resource, then the CPU will proceed. (DMA PRI bits are set to 00 <sub>2</sub> at reset.)
0 1 <sub>2</sub>	If the DMA channel and the CPU are requesting the same resource, then the <i>CPU will proceed</i> . Then, <b>after</b> the CPU access is <b>complete</b> , if the DMA coprocessor and CPU are <b>again</b> requesting the same resource, the <i>DMA coprocessor will proceed</i> . This priority rule provides a fair arbitration scheme by alternating CPU accesses with a DMA channel's access.
1 0 <sub>2</sub>	Reserved
1 1	DMA access is higher priority than the CPU access. If the DMA channel and the CPU are requesting the same resource, <i>the DMA will proceed</i> .

## 9.7 Data Transfer Modes

Each DMA channel can operate in four types of data transfer modes. These modes differ on:

- whether or not they use autoinitialization
- how they operate if autoinitialization is in effect or not.

Table 9–8 and the following paragraphs describe these data transfers.

Table 9–8. TRANSFER MODE Field Description Summary

TRANSFER MODE (Bits 3–2 and 5–4)	Transfer Mode Summary	Sub-section
0 0 <sub>2</sub>	Transfers are not terminated by the transfer counter. No autoinitialization is performed. The TCINT (transfer count interrupt) bits can still be used to cause an interrupt when the transfer counter makes a transition to zero. The DMA channel continues to run.	9.7.1
0 1 <sub>2</sub>	Transfers are terminated by the transfer counter. No autoinitialization is performed. A half code of 10 <sub>2</sub> is placed in the START field (bits 22–23 and bits 24–25 of the DMA channel control register when transfers are complete).	
1 0 <sub>2</sub>	Autoinitialization is performed when the transfer counter goes to zero without waiting for CPU intervention.	9.7.3
1 1 <sub>2</sub>	The DMA channel is autoinitialized when the CPU restarts the DMA coprocessor by using the DMA channel control register in the CPU. When the transfer counter goes to zero, operation is halted until the CPU starts the DMA coprocessor by using the START field in the DMA channel control register. A halt code of 10 <sub>2</sub> is placed in the START field by the DMA.	9.7.4

### 9.7.1 Running Under TRANSFER MODE = 00<sub>2</sub>

9

When TRANSFER MODE = 00<sub>2</sub>, transfers are not terminated when the transfer counter goes to zero, and no autoinitialization is performed. Even though the transfer counter does not halt transfers, an interrupt can be generated on the transfer counter transition to zero, causing TCINT FLAG bit = 1. If the DMA coprocessor channel is not halted after the transfer reaches zero, the counter will continue decrementing below zero.

### 9.7.2 Running Under TRANSFER MODE = 01<sub>2</sub>

When TRANSFER MODE = 01<sub>2</sub>, transfers are terminated when the transfer counter goes to zero, and no autoinitialization is performed. When the transfer counter goes to zero, the DMA channel is halted by forcing 10<sub>2</sub> into the START or AUX START field.

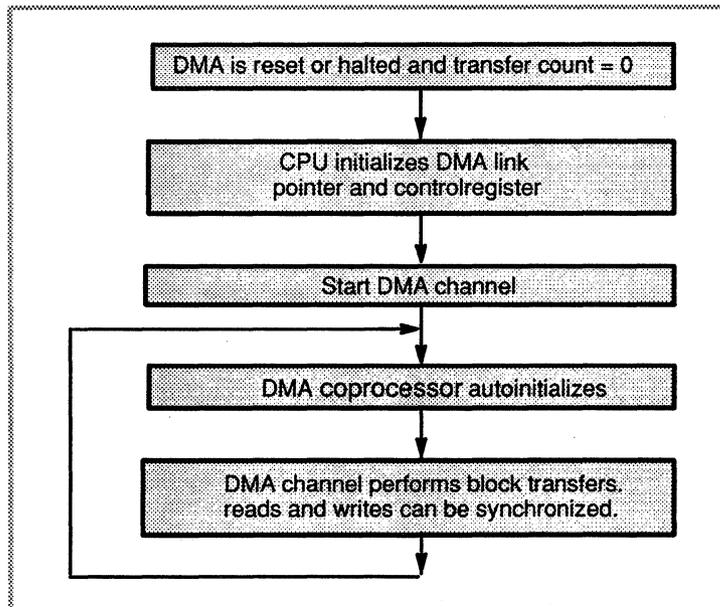
### 9.7.3 Running Under TRANSFER MODE = 10<sub>2</sub>

This transfer mode allows the DMA channel to take care of itself. It can run continuously, change pointers and synchronization by the autoinitialization procedure, and turn itself off.

This mode always starts with the DMA channel reset (START or AUX START a 00<sub>2</sub>) or halted (these bits 01<sub>2</sub> or 10<sub>2</sub>) and the transfer counter at 0. This occurs after a system reset, after the DMA channel is software reset (00<sub>2</sub> written to the START or AUX START bits), or after the channel is halted (01<sub>2</sub> or 10<sub>2</sub> written to these bits). The process for setting up and running a DMA channel under transfer mode 10<sub>2</sub> is summarized in Figure 9–14.

- 1) After placing the DMA channel in the reset or halted state and the transfer counter at 0, initialize the channel for the desired operation. In this case, set the transfer mode bits to 10<sub>2</sub>. Since the DMA channel autoinitializes itself when started under this mode, the CPU needs only to initialize the DMA channel control register and the DMA channel link pointer. The other DMA channel registers are automatically set up by the autoinitialization process. Synchronization of reads and writes is allowed.
- 2) After initializing the DMA channel, the channel can be started by writing 11<sub>2</sub> to the START or AUX START bits.
- 3) After this, the DMA channel will perform the sequence: *autoinitialize and do a block transfer*.

Figure 9–14. Running a DMA Channel Under Transfer Mode 10<sub>2</sub>



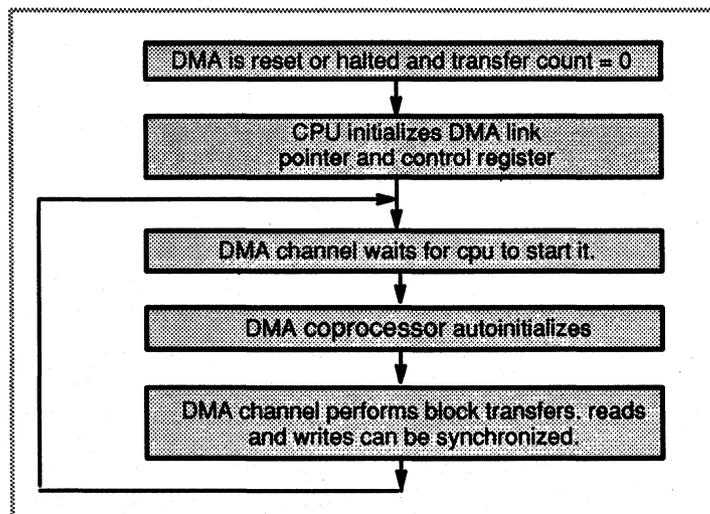
### 9.7.4 Running Under TRANSFER MODE = 11<sub>2</sub>

This transfer mode, besides having all of the advantages of autoinitialization, allows the CPU to very easily coordinate its operation with the operation of the DMA channels.

Like transfer mode (see subsection 9.7.3), this mode always starts out with the DMA channel reset or halted and the transfer count = 0. This occurs after a system reset, after the DMA channel is reset by writing 00<sub>2</sub> to the START or AUX START bits in the DMA channel control register, or after the channel is halted by writing 01<sub>2</sub> or 10<sub>2</sub> to these bits. The process for setting up and running a DMA channel under transfer mode 11<sub>2</sub> is summarized in Figure 9-15.

- 1) After placing the DMA channel in the halted or reset state and the transfer counter = 0, initialize the channel for the desired mode of operation. In this case, set the TRANSFER MODE bits to 11<sub>2</sub>. Since the DMA channel autoinitializes itself when started under this mode, the CPU needs to initialize only the DMA channel control register and the DMA channel link pointers. The other DMA channel registers are set up by the autoinitialization procedure.
- 2) After initializing the DMA channel, the channel can be started by writing 11<sub>2</sub> to the START bits.
- 3) Then, the DMA channel autoinitializes itself and does a block transfer.
- 4) When the transfer counter goes to zero, wait for the CPU to write a 11<sub>2</sub> to the START field of the DMA channel control register.
- 5) Then repeat the sequence *autoinitialize, transfer, and wait*.
- 6) When the transfer count goes to zero, the DMA channel can be halted by forcing 10<sub>2</sub> into the START or AUX START field.

Figure 9-15. Running a DMA Channel Under Transfer Mode 11<sub>2</sub>



## 9.8 Autoinitialization

When the DMA channel is operating in autoinitialization mode, the link pointer register and auxiliary link pointer register are used to initialize the registers that control the operation of the DMA channel. These pointers are memory-address locations for blocks of data that are to be loaded into the DMA register file, as shown in Figure 9–1 and Figure 9–2, beginning on page 9-4.

How this autoinitialization is done depends upon the current mode of operation of the DMA channel and the mode to which it is being autoinitialized. In all cases, either the link pointer or auxiliary link pointer (used in DMA split channel mode) contains the address of a block of memory that contains the new DMA channel register values (registers shown in Figure 9–1 on page 9-4).

During autoinitialization, the link pointer may be **incremented** (AUTO INIT STATIC = 0) or **held constant** (AUTO INIT STATIC = 1). (This is bit 8 or 9 of the channel control register, Figure 9–3 on page 9-8.)

- ❑ When the link pointer is **incremented**, the autoinitialization values are stored in sequential memory locations, and the link pointer or auxiliary link pointer is incremented in order to access each of these locations.
- ❑ Holding the linking pointer **constant** is very useful when autoinitializing the DMA channel from a stream-oriented device such as the on-chip communication ports or external FIFOs.

The SPLIT MODE bit (bit 14 in Figure 9–3 on page 9-8) defines the mode under which the DMA channel is currently running. When autoinitializing the DMA coprocessor, *do not change* the SPLIT MODE bit. This bit should be changed *only when the DMA coprocessor has been reset and halted* (see DMA START bit description, Table 9–5 on page 9-15).

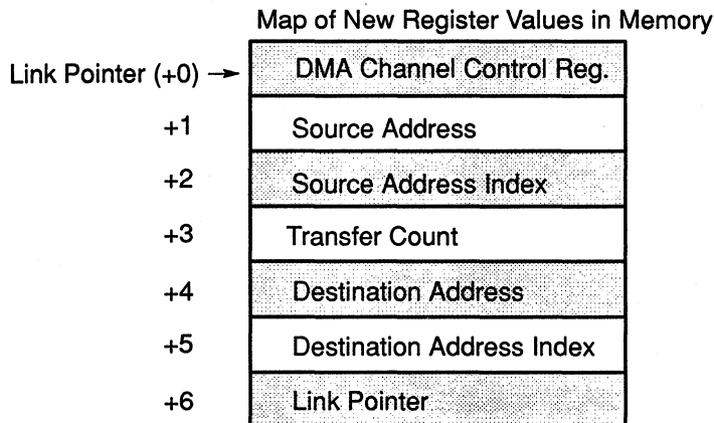
Autoinitialization is a DMA operation to the DMA coprocessor's registers; i.e., it reads the value pointed to by the link pointer and writes the value to the DMA register over the peripheral bus on the next available cycle.

If the DMA channel is performing memory-to-memory transfers (**SPLIT MODE = 0**), the link pointer is used. The DMA channel registers are loaded in the following order:

- 1) DMA channel control register
- 2) Source address register
- 3) Source address index register
- 4) Transfer count register
- 5) Destination address register
- 6) Destination address index register
- 7) Link pointer register

The storage of new values for these registers in memory is illustrated in Figure 9–16.

Figure 9–16. Store New Values of DMA Channel Registers in Memory (**SPLIT MODE = 0**)



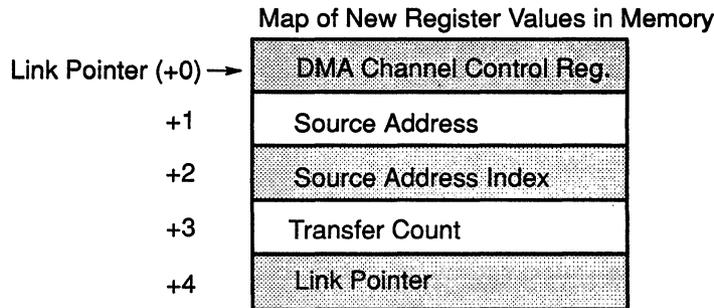
If the DMA channel is running in split mode (**SPLIT MODE = 1**), then the autoinitialize sequence depends upon which counter has terminated.

If the transfer-count register has gone to zero with **SPLIT MODE=1**, then the link-pointer register is used for autoinitialization. In this case, the DMA channel registers are loaded in the following order:

- 1) DMA channel control register
- 2) Source address register
- 3) Source address index register
- 4) Transfer count register
- 5) Link pointer register

The storage of the new values for these registers in memory is illustrated in Figure 9–17.

Figure 9–17. Store New Values of DMA Channel Registers in Memory (SPLIT MODE = 1)

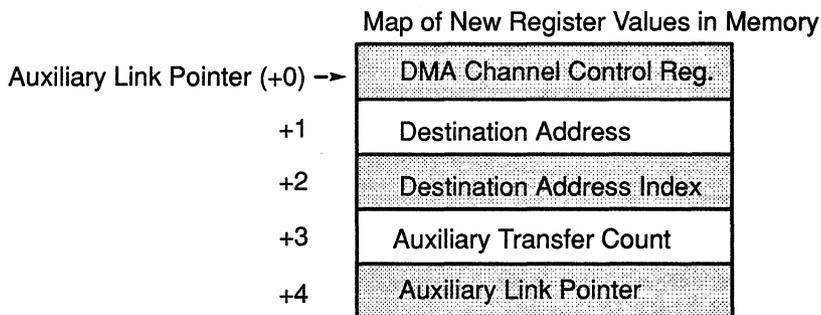


If the auxiliary counter has gone to zero with SPLIT MODE=1, then the auxiliary link pointer register is used for autoinitialization. In this case, the DMA channel registers are loaded in the following order:

- 1) DMA channel control register
- 2) Destination address register
- 3) Destination address index register
- 4) Auxiliary transfer count register
- 5) Auxiliary link pointer register

The storage of the new values of these registers in memory is illustrated in Figure 9–18.

Figure 9–18. Store New Values of DMA Channel Registers in Memory (SPLIT MODE = 1 and Auxiliary Transfer Counter = 0)



Usually, autoinitialization data will be stored in memory. In this case, synchronization for autoinitialization is not generally necessary. To disable the synchronization of data reads during autoinitialization, set AUTOINIT SYNCH (bits 10 and 11, DMA channel control register) to 0. In some cases, you may wish to transfer autoinitialization data in the same way as the synchronized data reads and writes. To synchronize autoinitialization based upon the interrupt identified with the READ SYNCH and WRITE SYNC fields (DIE register, page 3-8), set both the AUTOINIT SYNCH and AUX AUTOINIT SYNCH (bits 10–11 of DMA channel control register) to 1. In this way, autoinitialization will be synchronized only with the SYNCH MODE in effect.

The data reads for autoinitialization are arbitrated for by the DMA channels just like a typical DMA access. The only difference is that their synchronization is controlled by AUTOINIT SYNCH. A summary of the autoinitialization effect of the SYNC MODE and AUTOINIT SYNC bits is listed in Table 9–9 on page 9-37. *This table pertains to autoinitialization only.*

In **unified mode**, all of the writable control register bits are affected by autoinitialization. These bits are labeled in Figure 9–19.

In **split mode** during autoinitialization of the primary DMA channel, the writable, nonauxiliary bits may be modified, but auxiliary bits are protected (these bits are in Figure 9–20). In other words, only nonauxiliary bits are allowed to be modified by the CPU or DMA coprocessor. Also, if the auxiliary DMA channel is autoinitialized, the writable auxiliary bits may be modified, but nonauxiliary bits are protected. These bits are labeled in Figure 9–21.

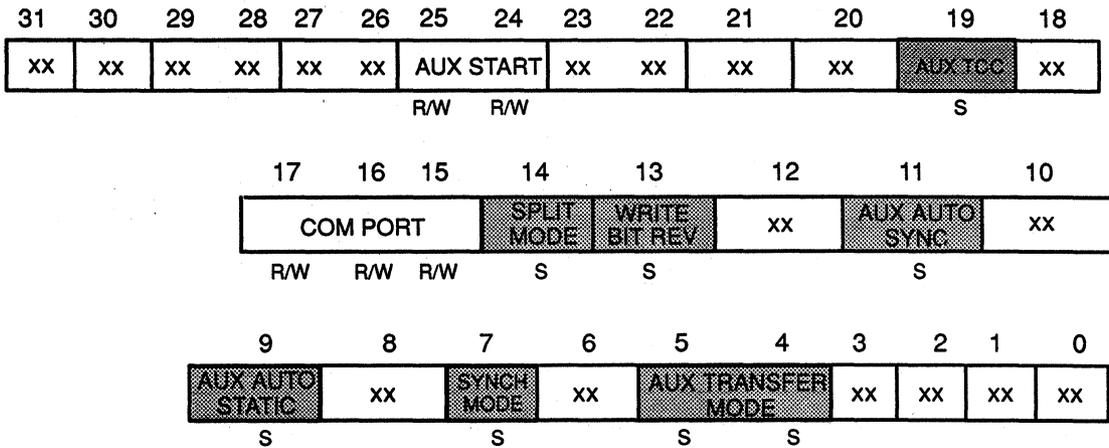
In all autoinitialization modes, the shadowed bits (Figure 9–19) that are writable (W-designated bits in Figure 9–3) do not have an affect until autoinitialization is complete. Unshadowed bits affect the autoinitialization sequence. In other words, at autoinitialization, shadowed bit values will be entered last after all registers are loaded (as specified by the link pointer).

Regardless of whether the DMA channel is running in unified mode or split mode, writes by the CPU or another external source to the DMA channel control register affect all writable bits.



## Autoinitialization

Figure 9–21. DMA Channel Control Register Bits That Can Be Modified by Autoinitialization of the Auxiliary Channel Under Split Mode



- s — These shadowed bits do not take affect until autoinitialization is complete.
- xx — Write protected during primary channel autoinitialization.

Autoinitialization synchronization is a function of

- ❑ the SYNC MODE bits (DMA channel control register bits 6 and 7) that control synchronization of data transfers, and
- ❑ the AUTOINIT SYNC bits (DMA channel control register bits 10 and 11) that affect only autoinitialization synchronization.

If the SYNC MODE bits are not set to synchronize data transfers (i.e., if the preceding data transfer is not synchronized on interrupts), then the DMA channel autoinitialization sequence will not be synchronized either. If the SYNC MODE bits are set to transfer data synchronously (i.e., if the preceding data transfer is synchronized), then the upcoming data channel autoinitialization sequence may be synchronized on either reads or writes or both (depending on whether the DMA coprocessor is in unified or split mode) as shown in Table 9–9. Note that for all combinations of the SYNC MODE and AUTOINIT SYNC bits not shaded in the table, the DMA channel autoinitialization sequence is not synchronized on interrupts.

Table 9–9. Effect of SYNC MODE and AUTOINIT MODE bits in Autoinitialization

These Bits of the DMA Channel Control Register		Cause Autoinitialization Synchronization To Occur On	
SYNC MODE	AUTOINIT SYNC		
Bit Nos: 7 — 6	Bit Nos: 11 — 10	Unified Mode	Split Mode
0 0	0 0	no sync	no sync
0 0	0 1	no sync	no sync
0 0	1 0	no sync	no sync
0 0	1 1	no sync	no sync
0 1	0 0	no sync	no sync
0 1	0 1	Read	Primary Ch
0 1	1 0	no sync	no sync
0 1	1 1	Read	Primary Ch
1 0	0 0	no sync	no sync
1 0	0 1	no sync	no sync
1 0	1 0	Write	Auxiliary Ch
1 0	1 1	Write	Auxiliary Ch
1 1	0 0	no sync	no sync
1 1	0 1	Read	Primary Ch
1 1	1 0	Write	Auxiliary Ch
1 1	1 1	Read Write	Primary Ch Auxiliary Ch

Autoinitialization Only

### 9.8.1 Fun With Link Pointers

For many applications, it is sufficient to autoinitialize the DMA channel with the same data each time. In this case, the new link-pointer value points to the start of the same block of data containing the new link pointer as illustrated in Figure 9–22. This particular example assumes a DMA channel that is not running in split mode.

If you want, you can get fancier. The new link pointer may point to a new set of register values as illustrated in Figure 9–23. This may be continued to any level you like. Have fun!

Figure 9–22. Self-Referential Link Pointer

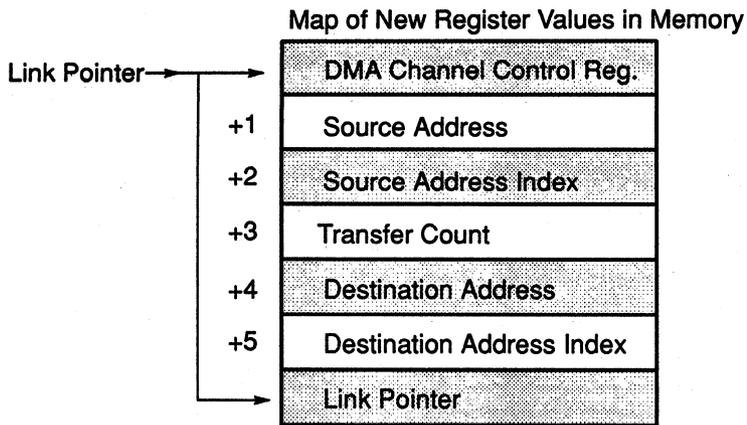
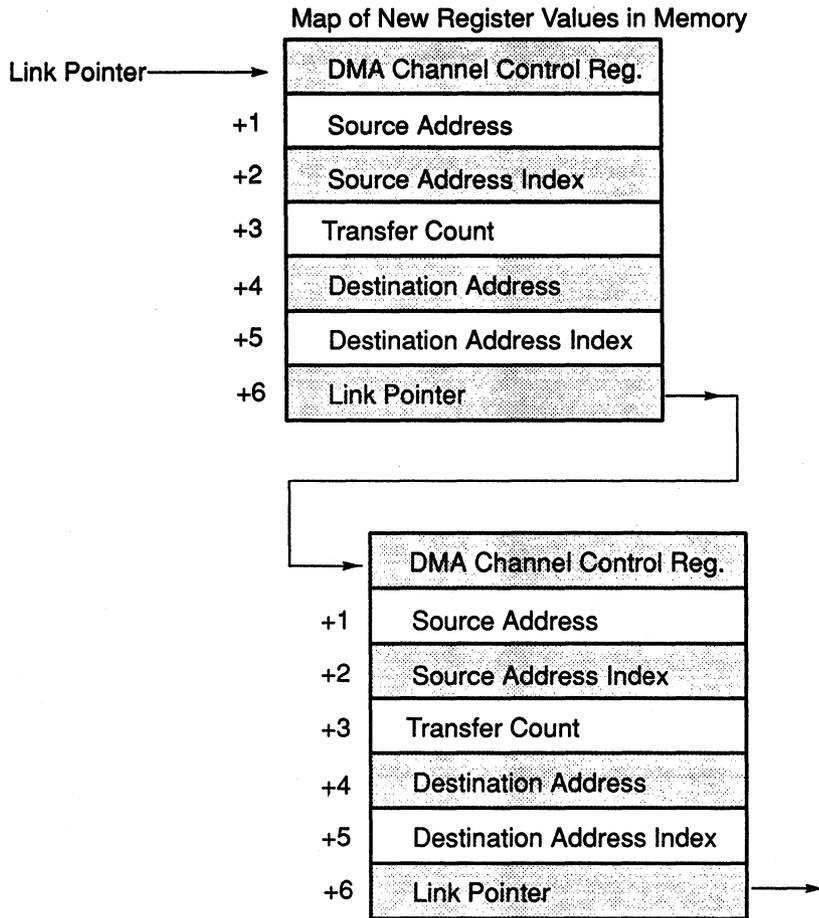


Figure 9-23. Referring to a New Link Pointer



## 9.9 DMA Coprocessor and Interrupts

All of the interrupts that the DMA coprocessor can see are first received by the CPU interrupt controller. If these interrupts are edge triggered, they are latched by the CPU in the appropriate interrupt-flag register. The **edge-triggered** interrupts are timer interrupts, DMA interrupts, and external interrupts that are configured as edge-triggered interrupts. Detailed information on interrupts is provided in Section 6.7 on page 6-23.

For **edge-triggered** interrupts, when the interrupt controller determines that the interrupt a DMA channel is waiting on has been latched into the interrupt flag, the CPU clears the interrupt flag and sends an interrupt pulse to the DMA channel. The DMA channel latches the interrupt locally until it can service the interrupt. At that time, the latched interrupt is cleared by the DMA coprocessor for two cycles.

**Level-triggered** interrupts that are generated by the communication ports and external interrupts that are configured as level-triggered interrupts are handled differently by the CPU interrupt controller. For level-triggered interrupts when the interrupt controller determines that the interrupt a DMA channel is waiting on has been received (recall that level-triggered interrupts are not latched by the CPU interrupt-controller), the CPU sends an interrupt pulse to the DMA channel. The DMA channel latches the interrupt locally until it can service the interrupt. At that time, the locally latched interrupt is cleared by the DMA coprocessor for two cycles.

The interrupt reset signal generated by the DMA coprocessor after a DMA interrupt is serviced has a higher priority over the interrupt set signal. Thus, the interrupt signal won't be continuously set even if the CPU is continuously sending the interrupt set signal. Hence, when the DMA-set priority scheme is used and a higher priority DMA channel is driven by continuous interrupt signals, the lower priority DMA channel can be serviced in between the higher priority DMA services.

The internal circuitry of the TMS320C40 guarantees proper operation between a communication port that generates level-triggered interrupts and the DMA channel that is synchronizing with those level-triggered interrupts. However, when you synchronize the DMA channels with external interrupts, it is better that these interrupt lines be configured as edge-triggered interrupts to ensure that only one interrupt is recognized.

Subsection 9.9.1 describes using interrupts to synchronize the DMA coprocessor. The interrupt mode for each channel is first selected in the DMA interrupt enable register, described with the CPU registers in subsection 3.1.8 on page 3-8.

### 9.9.1 Interrupts and Synchronization of DMA Channels

DMA channel transfers may be synchronized through the use of interrupts. The interrupt used is first selected by using the DMA interrupt enable register (subsection 3.1.8 on page 3-8).

Table 9-5 (page 9-15) describes the relationship between the SYNC MODE bits of the DMA channel control register and the four synchronization mechanisms performed:

- No synchronization (SYNC MODE =  $0\ 0_2$ )
- Source synchronization (SYNC MODE =  $0\ 1_2$ )
- Destination synchronization (SYNC MODE =  $1\ 0_2$ )
- Source and destination synchronization (SYNC MODE =  $1\ 1_2$ )

If the DMA split mode is selected, bits 6 and 7 of the DMA channel control register (page 9-15) are used to control channel synchronization:

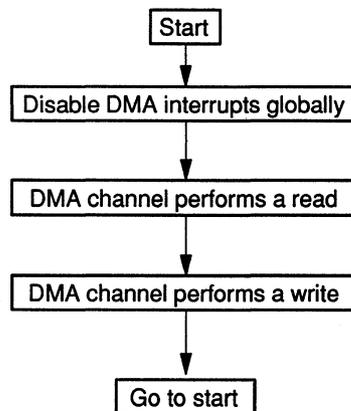
- bit 6** controls primary channel synchronization
- bit 7** controls auxiliary channel synchronization

#### No Synchronization (SYNC MODE = $0\ 0_2$ )

When SYNC MODE =  $0\ 0_2$ , no synchronization is performed. The DMA performs reads and writes whenever it has the priority to use the DMA bus. All interrupts are ignored and, therefore, are considered to be globally disabled. However, no bits in the DMA interrupt enable register are changed. Figure 9-24 shows the synchronization mechanism when SYNC MODE =  $0\ 0_2$ .

If an external interrupt is used for DMA interrupt synchronization, the external pin must be configured as a DMA interrupt pin (the DMA interrupt enable register is explained in subsection 3.1.8 on page 3-8).

Figure 9-24. No DMA Synchronization

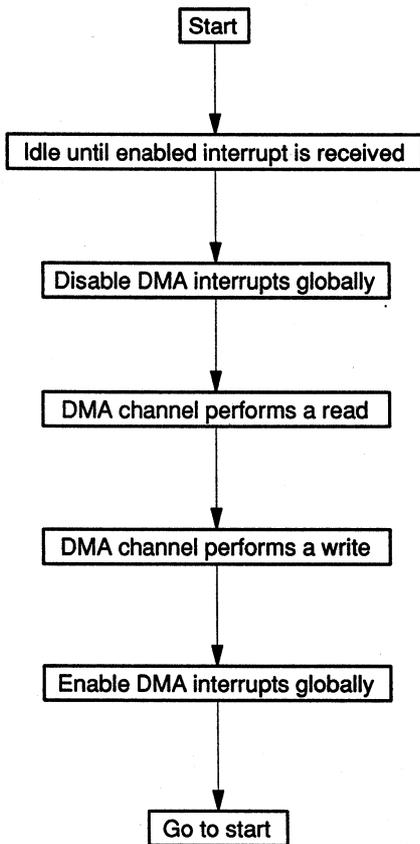


**Source Synchronization (SYNC MODE = 0 1<sub>2</sub>)**

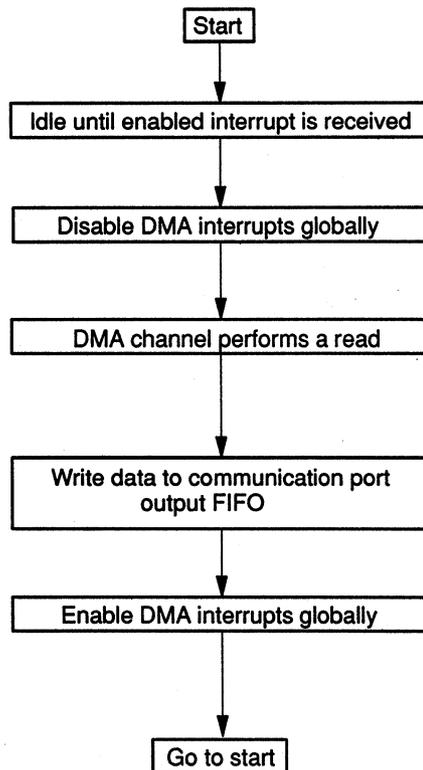
When SYNC MODE = 0 1<sub>2</sub>, the DMA coprocessor is synchronized to the source (see Figure 9–25). A read will not be performed until an interrupt is received by the DMA coprocessor (this also applies to the DMA primary channel in split mode as shown in Figure 9–25). Then, all DMA interrupts are disabled globally. However, no bits in the DMA interrupt enable register are changed.

Figure 9–25. DMA Source Synchronization

(a) DMA channel in unified mode



(b) Primary channel in split mode

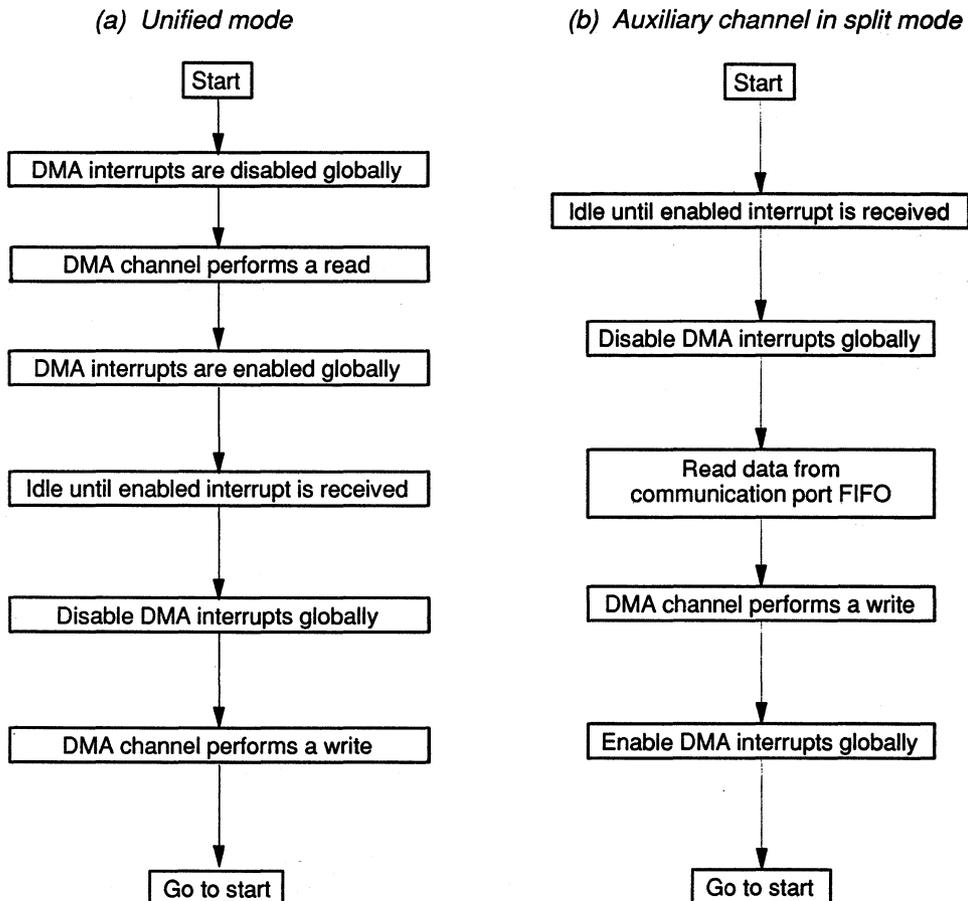


### Destination Synchronization (SYNC MODE = 1 0<sub>2</sub>)

When SYNC MODE = 1 0<sub>2</sub>, the DMA coprocessor is synchronized to the destination in unified mode. First, all interrupts are ignored until the read is complete. (Though the DMA interrupts are considered to be globally disabled, no bits in the DMA interrupt enable register are changed.) A write will not be performed until an interrupt is received by the DMA coprocessor. Figure 9–26 shows the synchronization mechanism when SYNC MODE = 1 0<sub>2</sub> in unified mode.

For the auxiliary channel in split mode, synchronization is similar to primary channel synchronization. The exception is that for the primary channel, the data is read from memory and written to a communication port output FIFO (shown on the right side of Figure 9–26). The auxiliary channel can read from a communication channel and write data to a memory address.

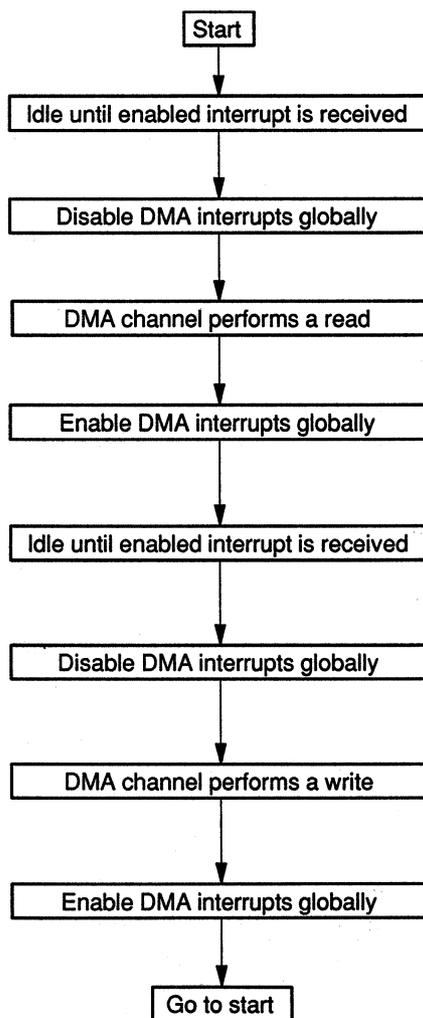
Figure 9–26. DMA Destination Synchronization



**Source and Destination Synchronization (SYNC MODE = 11<sub>2</sub>)**

When SYNC MODE = 11<sub>2</sub>, a read is performed when a read interrupt is received, and a write is performed on the write interrupt. If a write interrupt is received before a read interrupt, the write interrupt is latched and the DMA data write won't be executed until the read is completed. If DMA split mode is selected, it reacts as two independent synchronizations for the primary and auxiliary channels. Source and destination synchronization when SYNC MODE = 11<sub>2</sub> is shown in Figure 9-27.

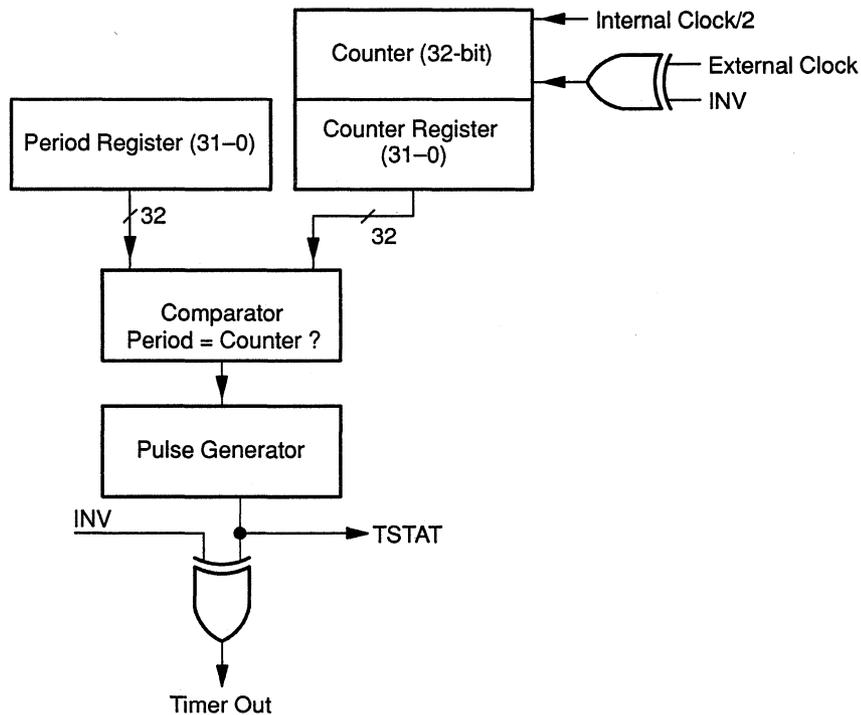
Figure 9-27. DMA Source and Destination Synchronization



## 9.10 TMS320C40 Timers

The TMS320C40 timer modules are general-purpose, 32-bit, timer/event counters, with two signaling modes and internal or external clocking (see Figure 9–28). The timer modules can be used to signal to the TMS320C40 or to the external world at specified intervals, or to count external events. **With an internal clock**, the timer can be used to signal an external A/D converter to start a conversion, or it can interrupt the TMS320C40 DMA controller to begin a data transfer. **With an external clock**, the timer can count external events and interrupt the CPU after a specified number of events. Available to each timer is an I/O pin that can be used as an input clock to the timer, an output clock signal, or a general-purpose I/O pin.

Figure 9–28. Timer Block Diagram



Three memory-mapped registers are used by each timer:

- Global-control register
- Period register
- Counter register

The **timer global-control register** determines the operating mode of the timer, monitors the timer status, and controls the function of the I/O pin of the timer. The **period register** specifies the timer's signaling frequency. The **timer counter register** contains the current value of the incrementing counter. The timer can be incremented on the rising edge or the falling edge of the input clock. The counter is zeroed whenever its value equals that in the period register. The pulse generator generates two types of external clock signals: pulse or clock. The memory map for the timer modules is shown in Figure 9–29.

Figure 9–29. Memory-Mapped Timer Locations

Register	Peripheral Address	
	Timer 0	Timer 1
Timer Global Control (See Table 9–10)	<b>808020h</b>	<b>808030h</b>
Reserved	808021h	808031h
Reserved	808022h	808032h
Reserved	808023h	808033h
Timer Counter (See subsection 9.10.2)	<b>808024h</b>	<b>808034h</b>
Reserved	808025h	808035h
Reserved	808026h	808036h
Reserved	808027h	808037h
Timer Period (See subsection 9.10.2)	<b>808028h</b>	<b>808038h</b>
Reserved	808029h	808039h
Reserved	80802Ah	80803Ah
Reserved	80802Bh	80803Bh
Reserved	80802Ch	80803Ch
Reserved	80802Dh	80803Dh
Reserved	80802Eh	80803Eh
Reserved	80802Fh	80803Fh

### 9.10.1 Timer Global-Control Register

The timer global control register is a 32-bit register that contains the global and port control bits for the timer module. Table 9–10 defines the register bits, names, and functions. Bits 3 – 0 are the port control bits; bits 11 – 6 are the timer global control bits. Figure 9–30 shows the 32-bit register. Note that at reset, all bits are set to 0 except for DATIN (set to the value read on TCLK).

Figure 9–30. Timer Global-Control Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx	xx	xx	xx	TSTAT	INV	CLKSRC	C/ $\bar{P}$	$\overline{\text{HLD}}$	GO	xx	xx	DATIN	DATOUT	$\bar{\text{i/O}}$	FUNC
				R/W	R/W	R/W	R/W	R/W	R/W			R	R/W	R/W	R/W

NOTE: xx = reserved bit, read as 0.  
R = read, W = write.

Table 9–10. Timer Global-Control Register Bits Summary

Bits	Name	Function
0	FUNC	FUNC controls the function of TCLK. If FUNC = 0, TCLK is configured as a general-purpose digital I/O port. If FUNC = 1, TCLK is configured as a timer pin (see Figure 9–33 for a description of the relationship between FUNC and CLKSRC).
1	I/O	If FUNC = 0 and CLKSRC = 0, TCLK is configured as a general-purpose I/O pin. In this case, if I/O = 0, TCLK is configured as a general-purpose input pin. If I/O = 1, TCLK is configured as a general-purpose output pin.
2	DATOUT	DATOUT drives TCLK when the TMS320C40 is in I/O port mode. DATOUT can also be used as an input to the timer.
3	DATIN	Data input on TCLK or DATOUT. A write has no effect.
4 – 5	Reserved	Read as 0.
6	GO	The GO bit resets and starts the timer counter. When GO = 1 and the timer is not held, the counter is zeroed and begins incrementing on the next rising edge of the timer input clock. The GO bit is cleared on the same rising edge. GO = 0 has no effect on the timer. Table 9–11 further defines these bits.
7	HLD	Counter hold signal. When this bit is zero, the counter is disabled and held in its current state. If the timer is driving TCLK, the state of TCLK is also held. The internal divide-by-two counter is also held so that the counter can continue where it left off when HLD is set to 1. The timer registers can be read and modified while the timer is being held. RESET has priority over HLD. Table 9–11 shows the effect of writing to GO and HLD.
8	C/P	Clock/pulse mode control. When C/P = 1, clock mode is chosen, and the signaling of the status flag and external output will have a 50 percent duty cycle. When C/P = 0, the status flag and external output will be active for one H1 cycle during each timer period (see Figure 9–31).
9	CLKSRC	Specifies the source of the timer clock. When CLKSRC = 1, an internal clock with frequency equal to one-half the H1 frequency is used to increment the counter. The INV bit has no effect on the internal clock source. When CLKSRC = 0, an external signal from the TCLK pin can be used to increment the counter. The external clock is synchronized internally, thus allowing external asynchronous clock sources that do not exceed the specified maximum allowable external clock frequency. This will be less than $f(H1)/2$ . (See Figure 9–33 for a description of the relationship between FUNC and CLKSRC).
10	INV	Inverter control bit. If an external clock source is used and INV = 1, the external clock is inverted as it goes into the counter. If the output of the pulse generator is routed to TCLK and INV = 1, the output is inverted before it goes to TCLK (see Figure 9–28.). If INV = 0, no inversion is performed on the input or output of the timer. The INV bit has no effect, regardless of its value, when TCLK is used in I/O port mode.
11	TSTAT	This bit indicates the status of the timer. It tracks the output of the uninverted TCLK pin. This flag sets a CPU interrupt on a transition from 0 to 1. A write has no effect.
12 – 31	Reserved	Read as 0.

Table 9–11 shows the result of a write using specified values of the GO and HLD bits in the timer global control register.

Table 9–11. *Result of a Write of Specified Values of GO and  $\overline{\text{HLD}}$*

GO (Bit 6)	$\overline{\text{HLD}}$ (Bit 7)	Result
0	0	All timer operations are held. No reset is performed.
0	1	Timer proceeds from state before write.
1	0	All timer operations are held, including zeroing of the counter. The GO bit is not cleared until the timer is taken out of hold.
1	1	Timer resets and starts.

### 9.10.2 Timer Period and Counter Registers

The 32-bit timer period register is used to specify the frequency of the timer signaling. The timer counter register is a 32-bit register that is reset to zero whenever it increments to the value of the period register. Both registers are set to 0 at reset. The locations of the registers are shown in Figure 9–29 on page 9-46.

Certain boundary conditions affect timer operation, such as a zero in the period register and an overflow of the counter. These conditions are listed as follows:

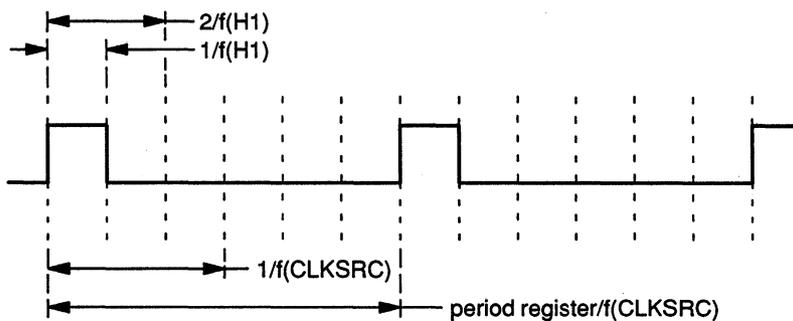
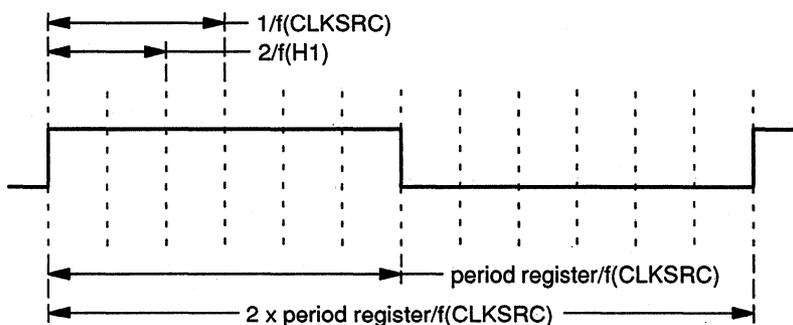
- When the period and counter registers are zero, the operation of the timer is dependent upon the  $C/\overline{P}$  mode selected. In pulse mode ( $C/\overline{P} = 0$ ), TSTAT is set and remains set. In clock mode ( $C/\overline{P} = 1$ ), the width of the cycle is  $2/f(H1)$ , and the external clocks are ignored.
- When the counter register is not 0 and the period register = 0, the counter will count, roll over to 0, and then behave as described immediately above (for both period and counter registers being zero).
- When the counter register is set to a value greater than the period register, the counter may overflow when being incremented. Once the counter reaches its maximum 32-bit value (0FFFF FFFFh), it simply clocks over to 0 and continues.

Writes from the peripheral bus override register updates from the counter and new status updates to the control register.

### 9.10.3 Timer Pulse Generation

The timer pulse generator (see Figure 9–28) can generate several different external signals. These signals may be inverted with the INV bit. The two basic modes are pulse mode and clock mode, as shown in Figure 9–31. In both modes, an internal clock source has a frequency of  $f(H1)/2$ , and an external clock source has a maximum frequency of less than  $f(H1)/2$ . Refer to timer timing in Chapter 14. In pulse mode ( $C/\overline{P} = 0$ ), the width of the pulse is  $1/f(H1)$ .

Figure 9-31. Timer Timing

(a) TSTAT and Timer Output ( $INV = 0$ ) When  $C/\bar{P} = 0$  (Pulse Mode)(b) TSTAT and Timer Output ( $INV = 0$ ) When  $C/\bar{P} = 1$  (Clock Mode)

The rate of timer signaling is determined by the frequency of the timer input clock and the period register. The following equations are valid with either an internal or an external timer clock:

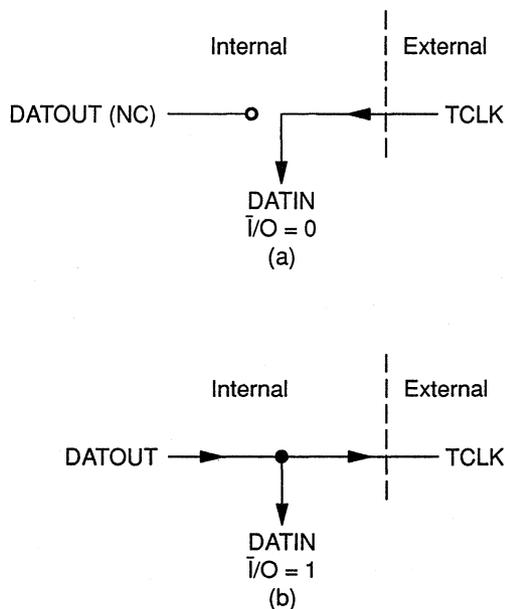
$$f(\text{pulse mode}) = f(\text{timer clock}) / \text{period register}$$

$$f(\text{clock mode}) = f(\text{timer clock}) / (2 \times \text{period register})$$

### 9.10.4 Timer Operation Modes

- ❑ The timer can receive its input and send its output in several different modes, depending upon the setting of CLKSRC, FUNC, and  $\bar{I}/O$ . The four timer modes of operation are defined as follows:
- ❑ If  $CLKSRC = 1$  and  $FUNC = 0$ , the timer input comes from the internal clock. The internal clock is not affected by the INV bit (bit 10 as shown in Figure 9–30 on page 9-47). In this mode, TCLK is connected to the I/O port control and can be used as a general-purpose I/O pin (see Figure 9–32). If  $\bar{I}/O = 0$ , TCLK is configured as a general-purpose input pin whose state can be read in DATIN. DATOUT has no effect on TCLK or DATIN. If  $\bar{I}/O = 1$ , TCLK is configured as a general-purpose output pin. DATOUT is placed on TCLK and can be read in DATIN.

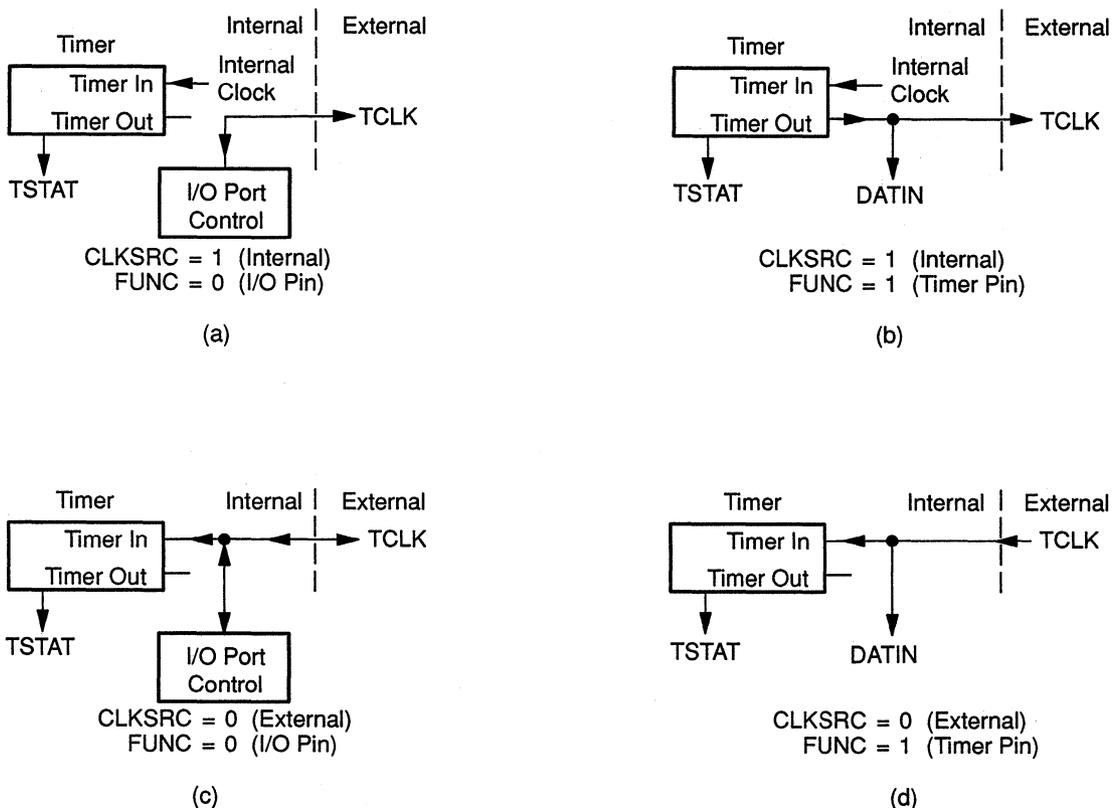
Figure 9–32. Timer I/O Port Configurations



- ❑ If  $CLKSRC = 1$  and  $FUNC = 1$ , the timer input comes from the internal clock, and the timer output goes to TCLK. This value may be inverted by using  $INV$ , and the value output on TCLK can be read in  $DATIN$ .
- ❑ If  $CLKSRC = 0$  and  $FUNC = 0$ , the timer is driven according to the status of the  $\bar{I}/O$  bit. If  $\bar{I}/O = 0$ , the timer input comes from TCLK. This value can be inverted by using  $INV$ , and the value of TCLK can be read in  $DATIN$ . If  $I/O = 1$ , TCLK is an output pin. Then, TCLK and the timer are both driven by  $DATOUT$ . All 0-to-1 transitions of  $DATOUT$  increment the counter.  $INV$  has no effect on  $DATOUT$ . The value of  $DATOUT$  can be read in  $DATIN$ .
- ❑ If  $CLKSRC = 0$  and  $FUNC = 1$ , TCLK drives the timer. If  $INV = 0$ , all 0-to-1 transitions of TCLK increment the counter. If  $INV = 1$ , all 1-to-0 transitions of TCLK increment the counter. The value of TCLK can be read in  $DATIN$ .

Figure 9–33 shows the four timer modes of operation.

Figure 9–33. Timer Modes as Defined by  $CLKSRC$  and  $FUNC$





## Pipeline Operation

---

---

---

Two characteristics of the TMS320C40 that contribute to its high performance are pipelining and concurrent I/O and CPU operation.

Five functional units control TMS320C40 operation: fetch, decode, read, execute, and DMA. Pipelining is the overlapping or parallel operations of the fetch, decode, read, and execute levels of a basic instruction.

By performing input/output operations, the DMA coprocessor reduces the need for the CPU to do so, thereby decreasing pipeline interference and enhancing the CPU's computational throughput.

Major topics discussed in this chapter are as follows:

<b>Section</b>	<b>Page</b>
10.1 Pipeline Structure .....	10-2
10.2 Pipeline Conflicts .....	10-4
■ Branch Conflicts .....	10-4
■ Register Conflicts .....	10-8
■ Memory Conflicts .....	10-11
10.3 Resolving Memory Conflicts .....	10-18
10.4 Clocking of Memory Accesses .....	10-20
■ Program Fetches .....	10-20
■ Data Loads and Stores .....	10-21

## 10.1 Pipeline Structure

The five major units of the TMS320C40 pipeline structure and their functions are as follows:

<b>Fetch Unit (F)</b>	Fetches the instruction words from memory and updates the program counter (PC).
<b>Decode Unit (D)</b>	Decodes the instruction word and performs address generation. Also controls any modification of the auxiliary registers and the stack pointer.
<b>Read Unit (R)</b>	If required, reads the operands from memory.
<b>Execute Unit (E)</b>	If required, reads the operands from the register file, performs the necessary operation, and writes results to the register file. If required, results of previous operations are written to memory.
<b>DMA Coprocessor (DMA)</b>	Reads and writes memory.

A basic instruction has four levels: fetch, decode, read, and execute. Figure 10–1 illustrates these four levels of the pipeline structure. The levels are indexed according to instruction and execution cycle. The perfect overlap in the pipeline, where all four units operate in parallel, occurs at cycle ( $m$ ). Those levels about to be executed are at  $m + 1$ , and those just executed are at  $m - 1$ . The TMS320C40 pipeline control allows a high-speed execution rate of one execution per cycle. It also manages pipeline conflicts so that they are transparent to the user. You do not need to take any special precautions to guarantee correct operation.

Figure 10-1. TMS320C40 Pipeline Structure

CYCLE	F	D	R	E
m-3	W	-	-	-
m-2	X	W	-	-
m-1	Y	X	W	-
m	Z	Y	X	W
m+1	-	Z	Y	X
m+2	-	-	Z	Y
m+3	-	-	-	Z

← Perfect overlap

**Notes:** 1) W, X, Y, and Z represent instructions.

2) F, D, R, E = fetch, decode, read, and execute, respectively.

Priorities from highest to lowest have been assigned to each of the functional units as follows:

- DMA (if configured as highest priority)
- Execute
- Read
- Decode
- Fetch
- DMA (if configured as lowest priority).

When the processing of an instruction is ready to pass to the next higher pipeline level, but that level is not ready to accept a new input, a pipeline conflict occurs. In this case, the lower priority unit waits until the higher priority unit completes its currently executing function.

## 10.2 Pipeline Conflicts

The pipeline conflicts of the TMS320C40 can be grouped into the following main categories:

- |                           |   |
|---------------------------|---|
| <b>Branch Conflicts</b>   | Involve most of those instructions or operations that read and/or modify the PC.                              |
| <b>Register Conflicts</b> | Involve delays that can occur when reading from or writing to registers that are used for address generation. |
| <b>Memory Conflicts</b>   | Occur when the internal units of the TMS320C40 compete for memory resources.                                  |

Each of these three types is discussed in the following sections. Examples are included. Note in these examples, when data is refetched or an operation is repeated, the symbol representing the stage of the pipeline is appended with a number. For example, if a fetch is performed again, the initial fetch is labeled F1 and the refetch is labeled F2. When an access is detained multiple cycles because of a not ready, the symbols RDY and RDY are used to indicate not ready and ready, respectively.

### 10.2.1 Branch Conflicts

#### 10.2.1.1 Standard Branches

The first class of pipeline conflicts occurs with standard (nondelayed) branches, i.e., BR, B*cond*, DB*cond*, CALL, IDLE, RPTB, RPTS, RETI*cond*, RETS*cond*, interrupts, and reset. Conflicts arise with these instructions and operations because during their execution, the pipeline is used only for the completion of the operation; other information fetched into the pipeline is discarded or refetched, or the pipeline is inactive. This is referred to as flushing the pipeline. Flushing the pipeline is necessary in these cases to guarantee that portions of succeeding instructions do not inadvertently get partially executed. TRAP*cond* and CALL*cond* are classified differently from the other types of branches and are considered later.

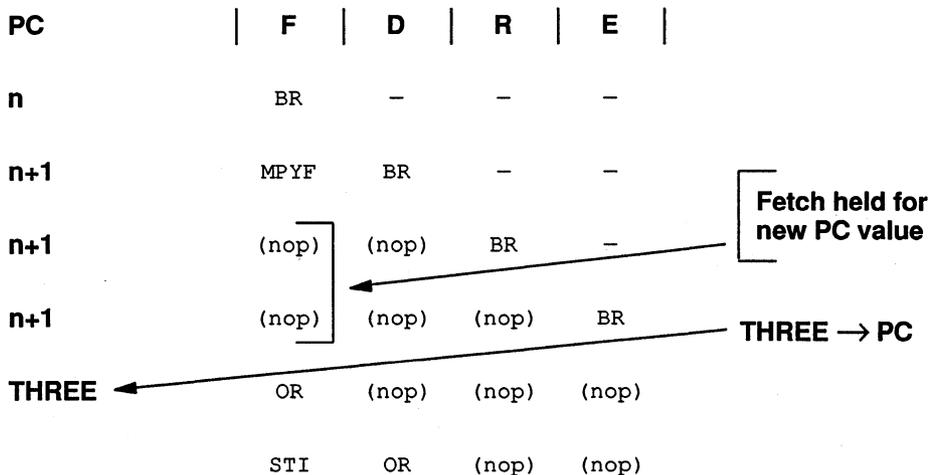
Example 10–1 shows the code and pipeline operation for a standard branch. Note that one dummy fetch is performed (F1), and then after the branch address is available, a new fetch (F2) is performed. This dummy fetch affects the cache.

Example 10-1. Standard Branch

```

BR      THREE      ; Unconditional branch
MPYF   ; Not executed
ADD    ; Not executed
SUBF   ; Not executed
AND    ; Not executed
.
.
.
THREE  OR          ; Fetched after BR is fetched
STI
.
.
    
```

PIPELINE OPERATION



RPTS and RPTB both flush the pipeline, allowing the RS, RE, and RC registers to be loaded at the proper time relative to the flow of the pipeline. If these registers are loaded without the use of RPTS or RPTB, no flushing of the pipeline occurs. If none of the repeat modes are being used, RS, RE, and RC may be used as general-purpose 32-bit registers without any pipeline conflicts occurring. In cases such as the nesting of RPTB due to nested interrupts, it may be necessary to load and store these registers directly while using the repeat modes. Since up to four instructions can be fetched before entering the repeat mode, loads should be followed by a branch to flush the pipeline. If the RC is changing when an instruction is loading it, the direct load takes priority over the modification made by the repeat mode logic.

### 10.2.1.2 Delayed Branches

Delayed branches are implemented to guarantee the fetching of the next three instructions. The delayed branches include BRD, BcondD, and DBcondD. Example 10–2 shows the code and pipeline operation for a delayed branch.

Example 10–2. Delayed Branch

```

BRD    THREE    ; Unconditional delayed branch
MPYF   ; Executed
ADD    ; Executed
SUBF   ; Executed
AND    ; Not executed
.
.
.
THREE  MPYF     ; Fetched after SUBF is fetched
.
.
.
    
```

#### PIPELINE OPERATION

PC	F	D	R	E	
n	BRD	-	-	-	
n+1	MPYF	BRD	-	-	<b>No execute delay</b>
n+2	ADDF	MPYF	BRD	-	
n+3	SUBF	ADDF	MPYF	BRD	<b>THREE → PC</b>
<b>THREE</b>	← MPYF	SUBF	ADDF	MPYF	

### 10.2.1.3 Delayed Branches With Annul Option

In addition to standard and delayed branches, the 'C40 supports delayed branches with an annulling option. These instructions include *BcondAT* (branch conditional, annul if true) and *BcondAF* (branch conditional, annul if false). The status of the condition (whether the *cond* specified is found true or false) controls whether or not a branch is performed (as in a delayed branch). The annulling operation cancels the effect of any operation performed in the execute phase of the three instructions following the *BcondAT* or *BcondAF*.

- ❑ If the condition is true, ***BcondAT*** annuls the effect of any operation performed in the execute phase of the three instructions that follow.
- ❑ If the condition is false, ***BcondAF*** annuls the effect of any operation performed in the execute phase of the three instructions that follow.

Example 10–3 uses both *BcondAT* and *BcondAF*.

#### Example 10–3. Using *BcondAF* and *BcondAT* Instructions

```

top:      LDI          *AR1, R0
          BNEGAT      bottom      ; If negative, branch and
          ADDI        *++AR2, R3   ; annul the execute phase
          MPYF        ; of ADDI, MPYF, and NOT.
          NOT         ; Otherwise, don't annul and
          SUBF        ; continue with SUBF.
          .
          .
          SUBI        1, R0
          BNNAF       top          ; If not negative, branch and
          ADDI        *++AR2, R3   ; do not annul the execute
          MPYF        ; phase of ADDI, MPYF, and
          NOT         ; NOT. Otherwise, annul ADDI,
bottom:   XOR         ; MPYF, and NOT, and continue
          ; with XOR.

```

At the start of Example 10–3, if the result of the load is negative (a *true* condition), the *BNEGAT* instruction causes a branch and also an annulment of the execute phase of the three instructions that follow it. As a result, the execute phase of the *ADDI* instruction does not occur, and register R3 is not updated by addition. However, the incrementing of AR2 and the reading of the data at the corresponding address do occur because these operators are in the decode and read phases of the pipeline, respectively, and thus are not annullable.

In short, operations that are annullable are:

- ❑ all writes to the register file that occur in the execute phase (ADDs, LDs, etc., but do not include LDA, LDPK, etc.).
- ❑ all stores to memory.

### 10.2.2 Register Conflicts

Register conflicts involve the reading or writing of registers used for addressing purposes. These registers are AR0–AR7, IR0, IR1, BK, DP, and SP. These conflicts occur when the pertinent register is not ready to be used. If an instruction writes to one of these registers, the decode unit cannot use that same register until the write is complete, i.e., until instruction execution is completed.

In Example 10–4, an auxiliary register is loaded, and the same auxiliary register is used on the next instruction. Since the decode stage needs the result of the write to the auxiliary register, the decode of this second instruction is delayed two cycles. Every time the decode is delayed, a refetch of the program word is performed; i.e., the first fetch of ADDF is at F1, followed by F2 and F3 (the final fetch). Since these are actual refetches, they can cause not only conflicts with the DMA controller but also cache hits and misses. (If a different AR register was used in the MPYF instruction (than was used in the LDI instruction), *no delay would occur.*)

Example 10–4. Write to an AR Followed by an AR for Address Generation

```

NEXT    LDI    7,AR2      ; 7 → AR2
        MPYF   *AR2,R0    ; Decode delayed 2 cycles
        ADDF
        FLOAT
    
```

PIPELINE OPERATION

PC	F	D	R	E	
n	LDI	—	—	—	Decode/address generation held for a new AR value
n+1	MPYF	LDI	—	—	
n+2	<b>ADDF</b>	MPYF	LDI	—	AR2 loaded
n+2	ADDF	MPYF	(nop)	LDI 7,AR2	
n+2	ADDF	MPYF	(nop)	(nop)	
n+3	FLOAT	ADDF	MPYF	(nop)	

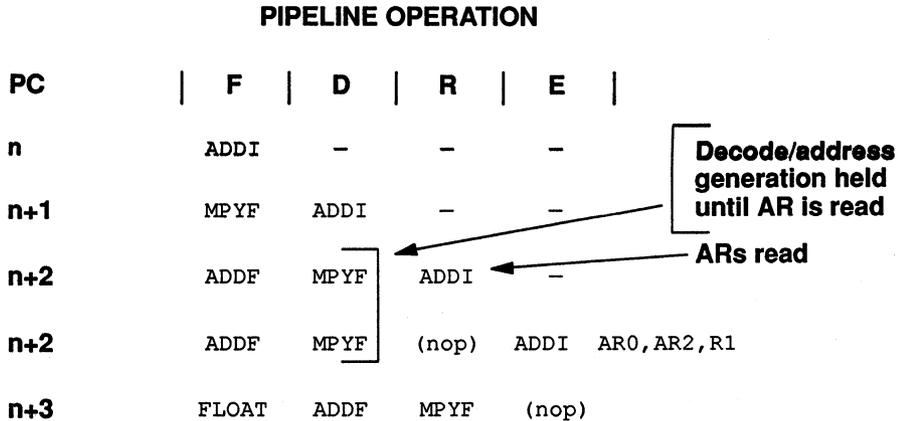
The case for reads of these registers is similar to the case for writes. If an instruction must read registers AR0–AR7 or SP, the use of those particular registers by the decode for the following instruction is delayed until the read is complete. The registers are read at the start of the execute cycle and therefore require only a one-cycle delay of the following decode. For four registers (IR0, IR1, BK, or DP), no delay is incurred upon a read.

In Example 10–5, two auxiliary registers are added together with the result going to an extended-precision register. The next instruction uses one of the same auxiliary registers as an address register. If the MPYF instruction used an AR register other than AR0 or AR2, *no delay would occur*.

Example 10–5. A Read of ARs Followed by ARs for Address Generation

```

NEXT   ADDI   AR0,AR2,R1      ; AR0 + AR2 → R1
        MPYF  *++AR2,R0      ; Decode delayed 1 cycle
        ADDF
        FLOAT
    
```



The DBR (decrement and branch) instruction’s use of auxiliary registers for loop counters is treated the same as if the use were for addressing. Therefore, the operation shown in the two previous examples can also occur for this instruction.

### 10.2.3 Memory Conflicts

Memory conflicts can occur when the memory bandwidth of a physical memory space is exceeded. For example, RAM blocks 0 and 1 and the ROM block can support only two accesses every cycle. The external interface can support only one access per cycle. Some conditions under which memory conflicts can be avoided are discussed in Section 10.3.

Memory pipeline conflicts consist of the following four types:

<b>Program Wait</b>	A program fetch is prevented from beginning.
<b>Program Fetch Incomplete</b>	A program fetch has begun but is not yet complete.
<b>Execute Only</b>	An instruction sequence requires three CPU data accesses in a single cycle.
<b>Hold Everything</b>	A primary or expansion bus operation must complete before another one can proceed.

These four types of memory conflicts are illustrated in examples and discussed in the paragraphs that follow.

#### **Program Wait**

Two conditions can prevent the program fetch from beginning:

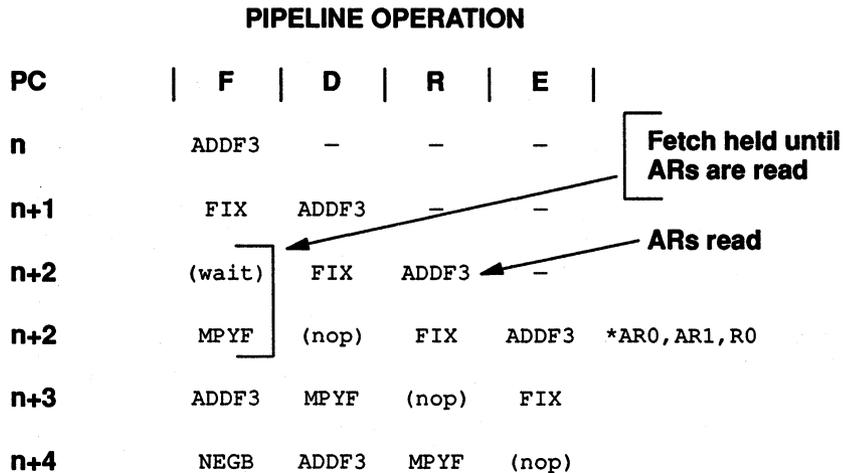
- ❑ The start of a CPU data access when
  - Two CPU data accesses are made to an internal RAM or ROM block, and a program fetch from the same block is necessary.
  - One of the external ports is starting a CPU data access, and a program fetch from the same port is necessary.
- ❑ A multicycle CPU data access or DMA data access over the external bus is needed.

Example 10–6 illustrates a program wait until a CPU data access completes. In this case, \*AR0 and \*AR1 are both pointing to data in RAM block 0, and the MPYF instruction will be fetched from RAM block 0. This results in the conflict shown. Since no more than two accesses can be made to RAM block 0 in a single cycle, the program fetch cannot begin and must wait until the CPU data accesses are complete.

Example 10–6. Program Wait Until CPU Data Access Completes

```

ADDF3 *AR0, *AR1, R0
FIX
MPYF
ADDF3
NEGB
    
```



Example 10–7 shows a program wait due to a multicycle data-data access or a multicycle DMA access. The ADDF, MPYF, and SUBF are fetched from some portion in memory other than the external port the DMA requires. The DMA begins a multicycle access. The program fetch corresponding to the CALL is made to the same external port the DMA is using.

Even if the DMA was configured as the lowest priority, a **multicycle access cannot be aborted**. The program fetch must therefore wait until the DMA access completes.

Example 10–7. Program Wait Due to Multicycle Access

PIPELINE OPERATION					
PC	F	D	R	E	
n	ADDF	-	-	-	
n+1	MPYF	ADDF	-	-	
n+2	SUBF	MPYF	ADDF	-	↑
n+3	(wait)	SUBF	MPYF	ADDF	2-cycle DMA access
n+3	CALL	(nop)	SUBF	MPYF	↓
n+4	-	CALL	(nop)	SUBF	

**Program Fetch Incomplete**

A program fetch incomplete occurs when a program fetch takes more than one cycle to complete due to wait states. In Example 10–8, the MPYF and ADDF are fetched from memory that supports single-cycle accesses. The SUBF is fetched from memory requiring one wait state. One example that demonstrates this conflict is a fetch across a bank boundary on the primary port.

Example 10–8. Multicycle Program Memory Fetches

PIPELINE OPERATION					
PC	F	D	R	E	
n	MPYF	-	-	-	
n+1	ADDF	MPYF	-	-	
n+2 $\overline{\text{RDY}}$	SUBF	ADDF	MPYF	-	↑
n+2 $\overline{\text{RDY}}$	SUBF	(nop)	ADDF	MPYF	1 wait state required
n+3	ADDI	SUBF	(nop)	ADDF	↓

### Execute Only

The Execute Only type of memory pipeline conflict occurs when a sequence of instructions requires three CPU data accesses in a single cycle. There are two cases in which this occurs:

- ❑ An instruction performs a store and is followed by an instruction that does two memory reads.
- ❑ An instruction performs two stores and is followed by an instruction that performs at least one memory read.

The first case is shown in Example 10–9. Since this sequence requires three data memory accesses and only two are available, only the execute phase of the pipeline is allowed to proceed. The dual reads required by the LDF || LDF is delayed one cycle. Note that a refetch of the next instruction can occur.

Example 10–9. Single Store Followed by Two Reads

```

        STF    R0, *AR1    ; R0 → *AR1
        LDF    *AR2, R1    ; *AR2 → R1 in parallel with
||      LDF    *AR3, R2    ; *AR3 → R2
    
```

#### PIPELINE OPERATION

PC	F	D	R	E	
n	STF	-	-	-	
n+1	LDF    LDF	STF	-	-	
n+2	W	LDF    LDF	STF	-	
n+3	X	W	LDF    LDF	STF R0, *AR1	<b>Write must complete before the 2 reads can complete.</b>
n+4	X	W	LDF    LDF	(nop)	
n+4	Y	X	W	LDF    LDF	*AR2, R1 and *AR3, R2

Example 10–10 shows a parallel store followed by a single load or read. Since the two parallel stores are required, the next CPU data memory read must wait a cycle before beginning. One program memory refetch may occur.

*Example 10–10. Parallel Store Followed by Single Read*

```

||      STF    R0,*AR0    ; R0 → *AR0 in parallel with
      STF    R2,*AR1    ; R2 → *AR1
      ADDF   @SUM,R1    ; R1 + @SUM → R1
      IACK
      ASH
  
```

**PIPELINE OPERATION**

PC	F	D	R	E
n	STF    STF	-	-	-
n+1	ADDF	STF    STF	-	-
n+2	IACK	ADDF	STF    STF	-
n+3	ASH	IACK	ADDF	STF    STF R0,*AR0 and R2,*AR1
n+4	ASH	IACK	ADDF	(nop)
n+4	-	ASH	IACK	ADDF

**Read must wait until the writes are complete**

### Hold Everything

There are three types of Hold Everything memory pipeline conflicts:

- ❑ A CPU data load or store cannot be performed because an external port is busy.
- ❑ An external load takes more than one cycle.
- ❑ Conditional calls and traps.

The first type of Hold Everything conflict occurs when one of the external ports is busy because of access that has started but is not complete. In Example 10–11, the first store is a two-cycle store. The CPU writes the data to an external port. The port control then takes two cycles to complete the data-data write. The LDF is a read over the same external port. Since the store is not complete, the CPU continues to attempt LDF until the port is available.

#### Example 10–11. Busy External Port

```
STF    R0, @DMA1
LDF    @DMA2, R0
```

#### PIPELINE OPERATION

PC	F	D	R	E	
n	STF	—	—	—	
n+1	LDF	STF	—	—	
n+2	W	LDF	STF	—	
n+2	W	LDF	(nop)	STF	↑
n+2	W	LDF	(nop)	(nop)	2-cycle external bus ↓ write access
n+3	X	W	LDF	(nop)	
n+4	Y	X	W	LDF	

The second type of Hold Everything conflict involves multicycle data reads. The read has begun and continues until completed. In Example 10–12, the LDF is performed from an external memory that requires several cycles to complete.

Example 10–12. Multicycle Data Reads

LDF @DMA, R0

PIPELINE OPERATION

PC	F	D	R	E	
n	LDF	-	-	-	
n+1	I	LDF	-	-	
n+2	J	I	LDF	-	↑
n+3	K (dummy)	I	LDF	-	↓
n+3	K <sub>2</sub>	J	I	LDF	

2-cycle external bus  
read access

The final type of Hold Everything conflict deals with conditional calls and traps, which are different from the other branch instructions. Whereas the other branch instructions are conditional loads, the conditional calls and traps are conditional stores, which take one more cycle than a conditional branch (see Example 10–13). The added cycle is used to push the return address after the call condition is evaluated.

Example 10–13. Conditional Calls and Traps

PIPELINE OPERATION

PC	F	D	R	E	
n	CALLcond	-	-	-	
n+1	I	CALLcond	-	-	
n+1	(nop)	(nop)	CALLcond	-	
n+1	(nop)	(nop)	(nop)	CALLcond	
n+1	(nop)	(nop)	(nop)	CALLcond	PC store cycle
n+2/CALLaddr	I	(nop)	(nop)	(nop)	

### 10.3 Resolving Memory Conflicts

If program fetches and data accesses are performed in such a manner that the resources being used cannot provide the necessary bandwidth, the program fetch is delayed until the data access is complete. Certain configurations of program fetch and data accesses yield conditions under which the TMS320C40 can achieve maximum throughput.

Table 10–1 shows how many accesses can be performed from the different memory spaces when it is necessary to do a program fetch and a single data access, and still achieve maximum performance (one cycle). Four cases achieve one-cycle maximization.

Table 10–1. One Program Fetch and One Data Access for Maximum Performance

Case No.	Global Bus Accesses	Accesses From Dual-Access Internal Memory	Local Bus Or Peripheral Accesses
1	1	1	–
2	1	–	1
3	–	2 from any combination of internal memory	–
4	–	1	1

Table 10–2 shows how many accesses can be performed from the different memory spaces when it is necessary to do a program fetch and two data accesses, still achieving maximum performance (one cycle). Six cases achieve this maximization.

*Table 10–2. One Program Fetch and Two Data Accesses for Maximum Performance*

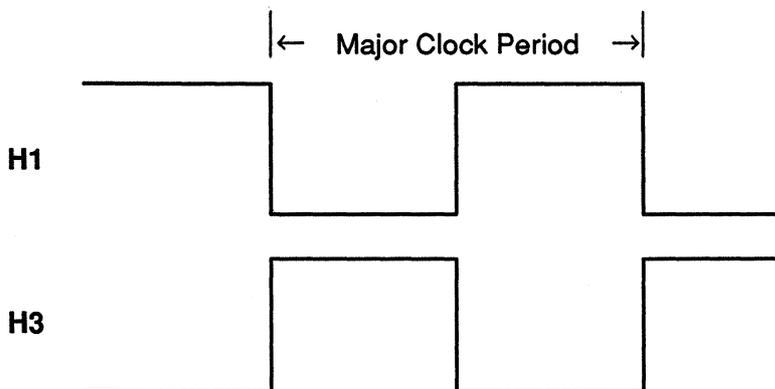
Case No.	Global Bus Accesses	Accesses From Dual-Access Internal Memory	Local Or Peripheral Bus Accesses
1	1	2 from any combination of internal memory	–
2†	1 program	1 Data	1 data
3†	1 data	1 Data	1 program
4	–	2 from same internal memory block and 1 from a different internal memory block	–
5	–	3 from different internal memory blocks	–
6	–	2 from any combination of internal memory	1
7	1 program	2 data	1 DMA
8	1 DMA	2 data	1 program

† For Cases 2 and 3, see Three-Operand Instruction Memory Reads on page 10-21.

## 10.4 Clocking of Memory Accesses

Internal clock phases (H1 and H3) and their relationship to memory accesses are discussed in this section to show how the TMS320C40 handles multiple memory accesses. Whereas the previous section discussed the interaction between sequences of instructions, this section discusses the flow of data on an individual instruction basis.

Each major clock period of 40 ns is composed of two minor clock periods of 20 ns, labeled H3 and H1 (these times assume a 50-MHz 'C40). The active clock period for H3 and H1 is the time when that signal is high.



The precise operation of memory reads and writes can be defined according to these minor clock periods. The types of memory operations that can occur are program fetches, data loads and stores, and DMA accesses.

### 10.4.1 Program Fetches

Internal program fetches are always performed during H3 unless a single data store must occur at the same time because of another instruction in the pipeline. In this case, the program fetch occurs during H1 and the data store during H3.

External program fetches always start at the beginning of H3 with the address being presented on the external bus. At the end of H1, they are completed with the latching of the instruction word.



If only one of the source operands is from memory (either *src1* or *src2*) and is located in internal memory, the data is read during H1. If the single memory source operand is in external memory, the read starts at the beginning of H3, with the address being presented on the external bus, and completes with the latching of the data word at the end of H1.

If both source operands are to be fetched from memory, then several cases occur. If both operands are located in internal memory, the *src1* read is performed during H3 and *src2* during H1, thus completing two memory reads in a single cycle.

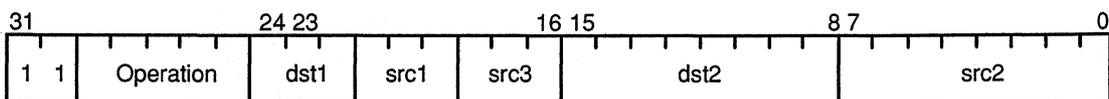
If *src1* is in internal memory and *src2* is in external memory, the *src2* access begins at the start of H3 and latches at the end of H1. At the same time, the *src1* access to internal memory is performed during H3. Again, two memory reads are completed in a single cycle.

If *src1* is in external memory and *src2* is in internal memory, two cycles are necessary to complete the two reads. In the first cycle, the internal *src2* access is performed. The *src1* is also performed, but not latched until the next H3.

If *src1* and *src2* are both from external memory, two cycles are required to complete the two reads. In the first cycle, the *src1* access is performed and loaded on the next H3; in the second cycle, the *src2* access is performed and loaded on that cycle's H1.

### Operations with Parallel Stores

Figure 10–4. Multiply or CPU Operation With a Parallel Store



The next class of instructions includes all instructions that have a store in parallel with another instruction. Bits 31 and 30 for these instructions are equal to 1 1<sub>2</sub>.

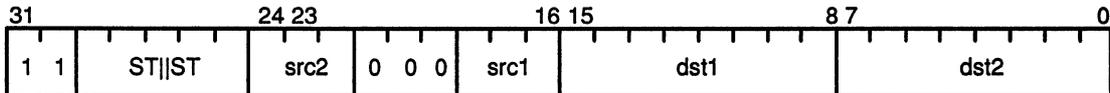
For operations that perform a multiply or ALU operation in parallel with a store, the instruction word format is shown in Figure 10–4. If the store operation to *dst2* is external or internal, it is performed during H3. Two bus cycles are required for external stores, but only one CPU cycle is necessary to complete the write.

If the memory read operation is external, it starts at the beginning of H3 and latches at the end of H1. If the memory read operation is internal, it is performed during H1. Note that memory reads are performed by the CPU

during the read (R) phase of the pipeline, and stores are performed during the execute (E) phase.

The instruction word format for instructions that have parallel stores to memory is shown in Figure 10–5. If both destination operands, *dst1* and *dst2*, are located in internal memory, *dst1* is stored during H3 and *dst2* during H1, thus completing two memory stores in a single cycle.

Figure 10–5. Two Parallel Stores



If *dst1* is in external memory and *dst2* is in internal memory, the *dst1* store begins at the start of H3. The *dst2* store to internal memory is performed during H1. Two bus cycles are required for the external store, but only one CPU cycle is necessary to complete the write. Again, two memory stores are completed in a single cycle.

If *dst1* is in internal memory and *dst2* is in external memory, an additional bus cycle is necessary to complete the *dst2* store. Only one CPU cycle is necessary to complete the write, but the port access requires three bus cycles. In the first cycle, the internal *dst1* store is performed during H3, and *dst2* is written to the port during H1. During the next cycle, the *dst2* store is performed on the external bus, beginning in H3, and executes as normal through the following cycle.

If *dst1* and *dst2* are both written to external memory, a single CPU cycle is still all that is necessary to complete the stores. In this case, four bus cycles are required.

- 1) In the first cycle, both *dst1* and *dst2* are written to the port, and the external bus access for *dst1* begins.
- 2) The store for *dst1* is completed on the second cycle, and the store for *dst2* begins on the third external bus cycle.
- 3) Finally, the store for *dst2* is completed on the fourth external bus cycle.



## Assembly Language Instructions

---

---

---

The TMS320C40 assembly language instruction set supports numeric-intensive, signal processing, and general-purpose applications. The instructions are organized into these major groups: load-and-store, two- or three-operand arithmetic/logical, parallel, program control, and interlocked operations instructions. The addressing modes used with the instructions are described in Chapter 5.

The TMS320C40 instruction set can also use one of 20 condition codes with any of the 10 conditional instructions, such as *LDFcond*. This chapter defines the condition codes and flags.

The assembler allows optional syntax forms to simplify the assembly language for special-case instructions. These optional forms are listed and explained.

Each of the individual instructions is described and listed in alphabetical order. An example instruction (on pages 11-15 through 11-17) demonstrates the special format used and explains its content.

This chapter discusses the following major topics:

<b>Section</b>	<b>Page</b>
11.1 Instruction Set .....	11-3
■ Load-and-Store Instructions .....	11-3
■ Two-Operand Arithmetic/Logical Instructions .....	11-4
■ Three-Operand Arithmetic/Logical Instructions .....	11-6
■ Program Control Instructions .....	11-6
■ Interlocked Operations Instructions .....	11-7
■ Parallel Operations Instructions .....	11-8

<b>Section</b>	<b>Page</b>
11.2 Condition Codes and Flags .....	11-10
11.3 Individual Instructions .....	11-13
■ Symbols and Abbreviations Used in Instructions .....	11-13
■ Optional Assembler Syntaxes .....	11-15
■ Individual instruction descriptions, alphabetized (includes syntax, operation, operands, encoding, description, cycles, status bits, mode bit, examples) .	11-17

## 11.1 Assembly Language Instructions — Instruction Set

The TMS320C40 instruction set is exceptionally well-suited to digital signal processing and other numeric-intensive applications. All instructions are a single machine word long, and most instructions take a single cycle to execute. In addition to multiply and accumulate instructions, the TMS320C40 possesses a full complement of general-purpose instructions.

The instruction set contains 135 instructions organized into the following functional groups:

- Load-and-store
- Two-operand arithmetic/logical
- Three-operand arithmetic/logical
- Program control
- Interlocked operations
- Parallel operations

Each of these groups is discussed in the succeeding subsections.

### 11.1.1 Load-and-Store Instructions

The TMS320C40 supports 23 load-and-store instructions (see Table 11–1). These instructions can

- Load a word from memory into a register,
- Store a word from a register into memory, or
- Manipulate data on the system stack.

Two of these instructions can load data conditionally. This is useful for locating the maximum or minimum value in a data set. See Section 12.2 for detailed information on condition codes.

Table 11–1. Load-and-Store Instructions

Instruction	Description	Instruction	Description
LBb	Load byte (signed)	LDPK	Load DP register immediate
LBUb	Load byte (unsigned)	LHW	Load half-word signed
LDA	Load address register	LHUw	Load half-word unsigned
LDE	Load floating-point exponent	LWLct	Load word left-shifted
LDEP	Load integer, expansion file register to primary register	LWRct	Load word right-shifted
LDF	Load floating-point value	POPF	Pop floating-point value from stack
LDFcond	Load floating-point value	PUSH	Push integer on stack
LDHI	Load 16-bit unsigned immediate into 16 MSBs	PUSHF	Push floating-point value on stack
LDI	Load integer	STF	Store floating-point value
LDIcond	Load integer conditionally	STI	Store integer
LDM	Load floating-point mantissa	STIK	Store integer immediate
LDPE	Load integer, primary register to expansion file register		

### 11.1.2 Two-Operand Instructions

The TMS320C40 supports a complete set of 43 two-operand arithmetic and logical instructions. The two operands are the source and destination. The source operand may be a memory word, a register, or a constant. The destination operand is always a register.

These instructions provide integer, floating-point, or logical operations, and multiprecision arithmetic. Table 11–2 lists these instructions.

Table 11-2. Two-Operand Instructions

Instruction	Description	Instruction	Description
ABSF	Absolute value of a floating-point number	NEGI	Negate integer
ABSI	Absolute value of an integer	NORM	Normalize floating-point value
ADDCT†	Add integers with carry	NOT	Bitwise logical-complement
ADDFT†	Add floating-point values	ORT†	Bitwise logical-OR
ADDIT	Add integers	RCPF	Reciprocal floating point
ANDT†	Bitwise logical-AND	RND	Round floating-point value
ANDNT†	Bitwise logical-AND with complement	ROL	Rotate left
ASHT†	Arithmetic shift	ROLC	Rotate left through carry
CMPFT†	Compare floating-point values	ROR	Rotate right
CMPIT†	Compare integers	RORC	Rotate right through carry
FIX	Convert floating-point value to integer	RSQRF	Reciprocal of square root, floating point
FLOAT	Convert integer to floating-point value	SUBBT†	Subtract integers with borrow
FRIEEE	Convert IEEE floating-point format to twos-complement floating-point for-	SUBC	Subtract integers conditionally
LSHT†	Logical shift	SUBFT†	Subtract floating-point values
MBct	Merge byte, left shifted	SUBIT†	Subtract integer
MHct	Merge half-word, left shifted	SUBRB	Subtract reverse integer with borrow
MPYFT†	Multiply floating-point values	SUBRF	Subtract reverse floating-point value
MPYIT†	Multiply integers	SUBRI	Subtract reverse integer
MPYSHIT†	Multiply signed integer, 32-MSB product	TOIEEE	Convert twos complement to IEEE format
MPYUHI†	Multiply unsigned integer, 32-MSB product	TSTBT†	Test bit fields
NEGB	Negate integer with borrow	XORT†	Bitwise exclusive-OR
NEGF	Negate floating-point value		

† Two- and three-operand versions

### 11.1.3 Three-Operand Instructions

Most instructions contain two or three operands. The 19 three-operand instructions allow the TMS320C40 to read two operands from memory or the CPU register file in a single cycle and store the results in a register. The following differentiates the two- and three-operand instructions:

- ❑ Two-operand instructions have a single-source operand (or shift count) and a destination operand.
- ❑ Three-operand instructions may have two source operands (or one source operand and a count operand) and a destination operand. A source operand may be a memory word, a register or a constant. The destination of a three-operand instruction is always a register.

Table 11–3 lists the instructions that have three-operand versions. Note that the 3 in the mnemonic can be omitted from three-operand instructions (see subsection 11.3.2).

Table 11–3. Three-Operand Instructions

Instruction	Description	Instruction	Description
ADDC3	Add with carry	MPYF3	Multiply floating-point values
ADDF3	Add floating-point values	MPYI3	Multiply integers
ADDI3	Add integers	MPYSHI3	Multiply signed integer, 32-MSB product
AND3	Bitwise logical-AND	MPYUHI3	Multiply unsigned integer, 32-MSB product
ANDN3	Bitwise logical-AND with complement	OR3	Bitwise logical-OR
ASH3	Arithmetic shift	SUBB3	Subtract integers with borrow
CMPP3	Compare floating-point values	SUBF3	Subtract floating-point values
CMPI3	Compare integers	SUBI3	Subtract integers
LSH3	Logical shift	TSTB3	Test bit fields
		XOR3	Bitwise exclusive-OR

### 11.1.4 Program Control Instructions

The program-control instruction group consists of all of those instructions (23) that affect program flow. The repeat mode allows repetition of a block of code (RPTB and RPTBD) or of a single line of code (RPTS). Both standard and delayed (single-cycle) branching are supported. Several of the program control instructions are capable of conditional operations (see Section 12.2 for detailed information on condition codes). Table 11–4 lists the program control instructions.

Table 11–4. Program Control Instructions

Instruction	Description	Instruction	Description
<i>Bcond</i>	Branch conditionally (standard)	<i>LAJcond</i>	Link and jump conditional
<i>BcondAF</i>	Branch conditionally delayed and annul if false	<i>LATcond</i>	Link and trap conditional
<i>BcondAT</i>	Branch conditionally delayed and annul if true	NOP	No operation
<i>BcondD</i>	Branch conditionally (delayed)	<i>RETicond</i>	Return from interrupt conditionally
BR	Branch unconditionally (standard)	<i>RETicondD</i>	Return from trap or interrupt, delayed
BRD	Branch unconditionally (delayed)	<i>RETScond</i>	Return from subroutine conditionally
CALL	Call subroutine	RPTB	Repeat block of instructions
<i>CALLcond</i>	Call subroutine conditionally	RPTBD	Repeat block, delayed
<i>DBcond</i>	Decrement and branch	RPTS	Repeat single instruction
<i>DBcondD</i>	Decrement and branch	SWI	Software interrupt
IDLE	Idle until interrupt	<i>TRAPcond</i>	Trap conditionally
LAJ	Link and jump		

### 11.1.5 Interlocked Operations Instructions

The interlocked operations instructions support multiprocessor communication and the use of external signals to allow for powerful synchronization mechanisms. They also guarantee the integrity of the communication and result in a high-speed operation. Refer to Chapter 7 for examples of the use of interlocked instructions.

Table 11–5. Interlocked Operations Instructions

Instruction	Description	Instruction	Description
LDFI	Load floating-point value, interlocked	STFI	Store floating-point value, interlocked
LDII	Load integer, interlocked	STII	Store integer, interlocked
SIGI	Signal, interlocked		

### 11.1.6 Parallel Operations Instructions

The parallel-operations instructions group makes a high degree of parallelism possible. Some of the TMS320C40 instructions can occur in pairs that will be executed in parallel. These instructions offer the following features:

- ❑ Parallel loading of registers,
- ❑ Parallel arithmetic operations, or
- ❑ Arithmetic/logical instructions used in parallel with a store instruction.

Each instruction in a pair is entered as a separate source statement. The second instruction in the pair must be preceded by two vertical bars (||). Table 11–6 lists the valid instruction pairs.

Table 11–6. Parallel Instructions

Mnemonic	Description
Parallel Arithmetic with Store Instructions	
ABSF    STF	Absolute value of a floating-point number and store floating-point value
ABSI    STI	Absolute value of an integer and store integer
ADDF3    STF	Add floating-point values and store floating-point value
ADDI3    STI	Add integers and store integer
AND3    STI	Bitwise logical-AND and store integer
ASH3    STI	Arithmetic shift and store integer
FIX    STI	Convert floating-point to integer and store integer
FLOAT    STF	Convert integer to floating-point value and store floating-point value
FRIEEE    STF	Convert IEEE floating-point format and store
LDF    STF	Load floating-point value and store floating-point value
LDI    STI	Load integer and store integer

Table concluded on next page.

Table 11-6. Parallel Instructions (Concluded)

Mnemonic	Description
<b>Parallel Arithmetic with Store Instructions</b>	
LSH3    STI	Logical shift and store integer
MPYF3    STF	Multiply floating-point values and store floating-point value
MPYI3    STI	Multiply integer and store integer
NEGF    STF	Negate floating-point value and store floating-point value
NEGI    STI	Negate integer and store integer
NOT3    STI	Complement value and store integer
OR3    STI	Bitwise logical-OR value and store integer
STF    STF	Store floating-point values
STI    STI	Store integers
SUBF3    STF	Subtract floating-point value and store floating-point value
TOIEEE    STF	Convert to IEEE format and store
SUBI3    STI	Subtract integer and store integer
XOR3    STI	Bitwise exclusive-OR values and store integer
<b>Parallel Load Instructions</b>	
LDF    LDF	Load floating-point
LDI    LDI	Load integer
<b>Parallel Multiply and Add/Subtract Instructions</b>	
MPYF3    ADDF3	Multiply and add floating-point
MPYF3    SUBF3	Multiply and subtract floating-point
MPYI3    ADDI3	Multiply and add integer
MPYI3    SUBI3	Multiply and subtract integer

## 11.2 Condition Codes and Flags

The TMS320C40 provides 20 condition codes (00000 – 10100, excluding 01011) that can be used with any of the conditional instructions, such as *RETScond* or *LDFcond*. The conditions include signed and unsigned comparisons, comparisons to zero, and comparisons based on the status of individual condition flags. Note that all conditional instructions can also accept the suffix U to indicate unconditional operation.

Seven condition flags provide information about properties of the result of arithmetic and logical instructions. The condition flags are stored in the status register (ST) and are affected by an instruction based upon the SET COND field (bit 15 of the status register).

- ❑ If SET COND = 0, the ST condition flags are set if the operation's target is any extended-precision register (R0–R11) .
- ❑ If SET COND = 1, the ST condition flags are **also** set if the operator's target is *any* register in the primary register file *except* the status register.
- ❑ The value of SET COND (0 or 1) does not affect the nature of the compare instructions (CMPF, CMPF3, CMPI, CMPI3, TSTB, or TSTB3).

The condition flags may be modified by most instructions when either of the preceding conditions is established and either of the following two cases occurs:

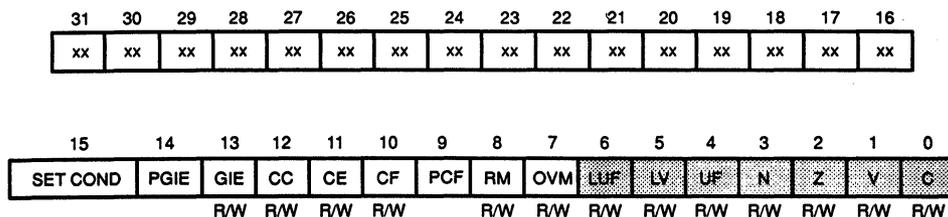
- ❑ A result is generated when the specified operation is performed to infinite precision. This is appropriate for compare-and-test instructions that do not store results in a register. It is also appropriate for arithmetic instructions that produce underflow or overflow.
- ❑ The output is written to the destination register as shown in Table 11–7. This is appropriate for other instructions that modify the condition flags.

Table 11–7. Output Value Formats

Type of Operation	Output Format
Floating-point	8-bit exponent, 1 sign bit, 31-bit fraction
Integer	32-bit integer
Logical	32-bit unsigned integer

Figure 11–1 shows the condition flags in the low-order bits of the status register. Following the figure is a list of status register condition flags and descriptions on how the flags are set by most instructions. For specific details of the effect of a particular instruction on the condition flags, see the description of that instruction in subsection 11.3.3.

Figure 11-1. Status Register



NOTE: xx = reserved bit.  
R = read, W = write.

- LUF Latched Underflow Condition Flag.** LUF is set whenever UF (floating-point underflow flag) is set. LUF may be cleared only by a processor reset or by modifying it in the status register (ST).
- LV Latched Overflow Condition Flag.** LV is set whenever V (overflow condition flag) is set. Otherwise, it is unchanged. LV may be cleared only by a processor reset or by modifying it in the status register (ST).
- UF Floating-Point Underflow Condition Flag.** A floating-point underflow occurs whenever the exponent of the result is less than or equal to  $-128$ . If a floating-point underflow occurs, UF is set, and the output value is set to 0. UF is cleared if a floating-point underflow does not occur.
- N Negative Condition Flag.** Logical operations assign N the state of the MSB of the output value. For integer and floating-point operations, N is set if the result is negative, and cleared otherwise. Zero is positive.
- Z Zero Condition Flag.** For logical, integer, and floating-point operations, Z is set if the output is 0, and cleared otherwise.
- V Overflow Condition Flag.** For integer operations, V is set if the result does not fit into the format specified for the destination (i.e.,  $-2^{32} \leq \text{result} \leq 2^{32} - 1$ ). Otherwise, V is cleared. For floating-point operations, V is set if the exponent of the result is greater than 127; otherwise, V is cleared. Logical operations always clear V.
- C Carry Flag.** When an integer addition is performed, C is set if a carry occurs out of the bit corresponding to the MSB of the output. When an integer subtraction is performed, C is set if a borrow occurs into the bit corresponding to the MSB of the output. Otherwise, for integer operations, C is cleared. The carry flag is unaffected by floating-point and logical operations. For shift instructions, this flag is set to the final value shifted out; for a zero shift count, this is set to zero.

Table 11–8 lists the condition mnemonic, code, description, and flag for each of the 19 condition codes.

Table 11–8. Condition Codes and Flags

Condition	Code	Description	Flag†
Unconditional Compares			
U	00000	Unconditional	Don't care
Unsigned Compares			
LO	00001	Lower than	C
LS	00010	Lower than or same as	C OR Z
HI	00011	Higher than	~C AND ~Z
HS	00100	Higher than or same as	~C
EQ	00101	Equal to	Z
NE	00110	Not Equal to	~Z
Signed Compares			
L	00111	Less than	N
LE	01000	Less than or equal to	N OR Z
GT	01001	Greater than	~N AND ~Z
GE	01010	Greater than or equal to	~N
EQ	00101	Equal to	Z
NE	00110	Not equal to	~Z
Compare to Zero			
Z	00101	Zero	Z
NZ	00110	Not zero	~Z
P	01001	Positive	~N AND ~Z
N	00111	Negative	N
NN	01010	Nonnegative	~N
Compare to Condition Flags			
NN	01010	Nonnegative	~N
N	00111	Negative	N
NZ	00110	Nonzero	~Z
Z	00101	Zero	Z
NV	01100	No overflow	~V
V	01101	Overflow	V
NUF	01110	No underflow	~UF
UF	01111	Underflow	UF
NC	00100	No carry	~C
C	00001	Carry	C
NLV	10000	No latched overflow	~LV
LV	10001	Latched overflow	LV
NLUF	10010	No latched floating-point underflow	~LUF
LUF	10011	Latched floating-point underflow	LUF
ZUF	10100	Zero or floating-point underflow	Z OR UF

† The ~ means logical complement (“not true” condition).

## 11.3 Individual Instructions

This section contains the individual assembly language instructions for the TMS320C40. The instructions are listed in alphabetical order. Information for each instruction includes assembler syntax, operation, operands, encoding, description, cycles, status bits, mode bit, and examples.

Definitions of the symbols and abbreviations, as well as optional syntax forms allowed by the assembler, precede the individual instruction description section. Also, an example instruction shows the special format used and explains its content.

A functional grouping of the instructions, as well as a complete instruction set summary, can be found in Section 11.1. Appendix B lists the opcodes for all the instructions. Refer to Chapter 6 for information on memory addressing. Code examples using many of the instructions are given in Chapter NO TAG, *Software Applications*.

### 11.3.1 Symbols and Abbreviations

Table 11–9 lists the symbols and abbreviations used in the individual instruction descriptions.

Table 11–9. Instruction Symbols

Symbol	Meaning
<i>src</i> <i>src1</i> <i>src2</i> <i>src3</i> <i>src4</i>	Source operand Source operand 1 Source operand 2 Source operand 3 Source operand 4
<i>dst</i> <i>dst1</i> <i>dst2</i> <i>disp</i> <i>cond</i> <i>count</i>	Destination operand Destination operand 1 Destination operand 2 Displacement Condition Shift count
G T P B	General addressing modes Three-operand addressing modes Parallel addressing modes Conditional-branch addressing modes
ARn IRn Rn RC RE RS ST	Auxiliary register n Index register n Register address n Repeat count register Repeat end address register Repeat start address register Status register
C GIE N PC RM SP	Carry bit Global interrupt enable bit Trap vector Program counter Repeat mode flag System stack pointer
x  $x \rightarrow y$ $x(\text{man})$ $x(\text{exp})$	Absolute value of x Assign the value of x to destination y Mantissa field (sign + fraction) of x Exponent field of x
op1    op2	Operation 1 performed in parallel with operation 2
x AND y x OR y x XOR y ~x	Bitwise logical-AND of x and y Bitwise logical-OR of x and y Bitwise logical-XOR of x and y Bitwise logical-complement of x
$x \ll y$ $x \gg y$ *++SP *SP--	Shift x to the left y bits Shift x to the right y bits Increment SP and use incremented SP as address Use SP as address and decrement SP

### 11.3.2 Optional Assembler Syntaxes

The assembler allows a relaxed syntax form for some instructions. These optional forms simplify the assembly language so that special-case syntax can be ignored. The following is a list of these optional syntax forms.

- ❑ The destination register can be omitted on unary arithmetic and logical operations when the same register is used as a source. For example,

ABSI R0,R0                    *can be written as*                    ABSI R0

Instructions affected: ABSI, ABSF, FIX, FLOAT, NEGB, NEGF, NEGI, NORM, NOT, RND.

- ❑ All 3-operand instructions can be written without the 3. For example,

ADDI3 R0,R1,R2            *can be written as*            ADDI R0,R1,R2

Instructions affected: ADDC3, ADDF3, ADDI3, AND3, ANDN3, ASH3, LSH3, MPYF3, MPYI3, OR3, SUBB3, SUBF3, SUBI3, XOR3, MPYSHI3, MPYUHI3.

This also applies to all the pertinent parallel instructions.

- ❑ All 3-operand comparison instructions can be written without the 3. For example,

CMPI3 R0,\*AR0            *can be written as*            CMPI R0,\*AR0

Instructions affected: CMPI3, CMPF3, TSTB3.

- ❑ Indirect operands with an explicit 0 displacement are allowed. In 3-operand or parallel instructions, operands with 0 displacement are automatically converted to no-displacement mode. For example:

LDI    \*+AR0(0),R1 *is legal*

Also

ADDI3 \*+AR0(0),R1,R2 *is equivalent to*    ADDI3 \*AR0,R1,R2

- ❑ Indirect operands can be written with no displacement; in which case, a displacement of one is assumed. For example,

LDI    \*AR0++(1),R0 *can be written*            LDI \*AR0++,R0

- ❑ All conditional instructions accept the suffix U to indicate unconditional operation. Also, the U can be omitted from unconditional short branch instructions. For example:

BU label                    *can be written*                    B label

- ❑ Labels can be written with or without a trailing colon. For example:

```
label10:    NOP
label11    NOP
label12:            (label assembles to next source line)
```

- ❑ Empty expressions are not allowed for the displacement in indirect mode:

LDI \*+AR0(),R0 *is not legal*

- ❑ Immediate-mode destination operands of BR and CALL can be written with an at sign (@):

BR label *can be written* BR @label

- ❑ The LDP pseudo-op can be used to load a register (DP by default) with the 16 MSBs of a relocatable address as follows:

LDP addr,REG *or* LDP @addr,REG

The at sign (@) is optional.

If the destination REG is the DP, LDP generates an LDPK instruction. Otherwise it generates an LDIU instruction. In both cases an immediate operand with a special relocation type is used.

- ❑ Parallel instructions can be written in either order. For example:

ADDI *can be written as* STI  
|| STI || ADDI

- ❑ The parallel bars indicating part 2 of a parallel instruction can be written anywhere on the line from column 0 to the mnemonic. For example:

ADDI *can be written as* ADDI  
|| STI || STI

- ❑ If the second operand of a parallel instruction is the same as the third (destination register) operand, the third operand can be omitted. This allows the writing of 3-operand parallel instructions that look like normal 2-operand instructions. For example,

ADDI \*AR0,R2,R2 *can be written as* ADDI \*AR0,R2  
|| MPYI \*AR1,R0,R0 || MPYI \*AR1,R0

Instructions affected (applies to all parallel instructions that have a register as the second operand): ADDI, ADDF, AND, MPYI, MPYF, OR, SUBI, SUBF, XOR.

- ❑ All commutative operations in parallel instructions can be written in either order. For example, the ADDI part of a parallel instruction can be written in either of two ways:

ADDI \*AR0,R1,R2 *or* ADDI R1,\*AR0,R2

The instructions affected are parallel instructions containing any of the following: ADDI, ADDF, MPYI, MPYF, AND, OR, XOR.

- ❑ Use the syntax in Table 11–10 to designate CPU registers in operands.

### 11.3.3 Individual Instruction Descriptions

Each assembly language instruction for the TMS320C40 is described in this section in alphabetical order. The description includes the assembler syntax, operation, operands, encoding, description, cycles, status bits, mode bit, and examples.

Table 11-10. CPU Register Syntax

Assembler Syntax	Register Machine Value (hex)	Assigned Function Name	Explained in Paragraph	On Page
R0	00	Extended-precision register 0	3.1.1	3-4
R1	01	Extended-precision register 1	3.1.1	3-4
R2	02	Extended-precision register 2	3.1.1	3-4
R3	03	Extended-precision register 3	3.1.1	3-4
R4	04	Extended-precision register 4	3.1.1	3-4
R5	05	Extended-precision register 5	3.1.1	3-4
R6	06	Extended-precision register 6	3.1.1	3-4
R7	07	Extended-precision register 7	3.1.1	3-4
R8	1C	Extended-precision register 8	3.1.1	3-4
R9	1D	Extended-precision register 9	3.1.1	3-4
R10	1E	Extended-precision register 10	3.1.1	3-4
R11	1F	Extended-precision register 11	3.1.1	3-4
AR0	08	Auxiliary register 0	3.1.2	3-5
AR1	09	Auxiliary register 1	3.1.2	3-5
AR2	0A	Auxiliary register 2	3.1.2	3-5
AR3	0B	Auxiliary register 3	3.1.2	3-5
AR4	0C	Auxiliary register 4	3.1.2	3-5
AR5	0D	Auxiliary register 5	3.1.2	3-5
AR6	0E	Auxiliary register 6	3.1.2	3-5
AR7	0F	Auxiliary register 7	3.1.2	3-5
DP	10	Data-page pointer	3.1.3	3-5
IR0	11	Index register 0	3.1.4	3-5
IR1	12	Index register 1	3.1.4	3-5
BK	13	Block-size register	3.1.5	3-5
SP	14	System stack pointer	3.1.6	3-5
ST	15	Status register	3.1.7	3-5
DIE	16	DMA Coprocessor interrupt enable	3.1.8	3-8
IIE	17	Internal-interrupt enable register	3.1.9	3-10
IIF	18	IIOF pins and interrupt flag register	3.1.10	3-12
RS	19	Repeat start address	3.1.11	3-14
RE	1A	Repeat end address	3.1.11	3-14
RC	1B	Repeat counter	3.1.11	3-14

## EXAMPLE *Example Instruction*

---

**Syntax**     **INST** *src, dst*

or

**INST1** *src2, dst1*  
|| **INST2** *src3, dst2*

Each instruction begins with an assembler syntax expression. Labels may be placed either before the command (instruction mnemonic) on the same line or on the preceding line in the first column. The optional comment field that concludes the syntax is not included in the syntax expression. Space(s) are required between each field (label, command, operand, and comment fields).

The syntax examples illustrate the common one-line syntax and the two-line syntax used in parallel addressing. Note that the two vertical bars || that indicate a parallel addressing pair can be placed anywhere before the mnemonic on the second line. The first instruction in the pair can have a label, but the second instruction cannot have a label.

**Operation**   |*src*| → *dst*

or

      |*src2*| → *dst1*  
|| *src3* → *dst2*

The instruction operation sequence describes the processing that takes place when the instruction is executed. For parallel instructions, the operation sequence is performed in parallel. Conditional effects of status register specified modes are listed for conditional instructions such as *Bcond*.

**Operands**   *src* general addressing modes (G):

0 0	register (any register in CPU primary register file)
0 1	direct
1 0	indirect
1 1	immediate

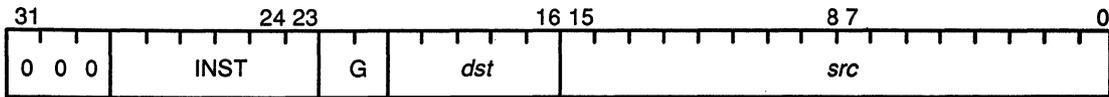
*dst* register (any register in CPU primary register file)

or

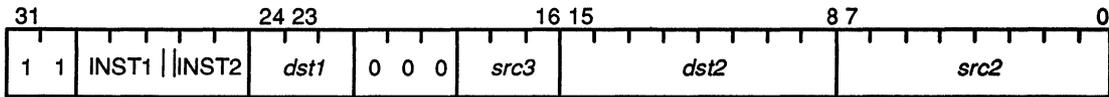
*src2* indirect (disp = 0, 1, IR0, IR1)  
*dst1* register (R0 — R7)  
*src3* register (R0 — R7)  
*dst2* indirect (disp = 0, 1, IR0, IR1)

Operands are defined according to the addressing mode and/or the type of addressing used. Note that indirect addressing uses displacements and the index registers. Refer to Chapter 5 for detailed information on addressing.

**Encoding**



or



Encoding examples are shown for general addressing and parallel addressing. The instruction pair for the parallel addressing example consists of INST1 and INST2.

**Description** Instruction execution and its effect on the rest of the processor or memory contents are described. Any constraints on the operands imposed by the processor or the assembler are discussed. The description parallels and supplements the information given by the operation block.

**Cycles** 1

The digit specifies the number of cycles required to execute the instruction.

- Status Bits**
- LUF** **Latched Floating-Point Underflow Condition Flag.** 1 if a floating-point underflow occurs, unchanged otherwise.
  - LV** **Latched Overflow Condition Flag.** 1 if an integer or floating-point overflow occurs, unchanged otherwise.
  - UF** **Floating-Point Underflow Condition Flag.** 1 if a floating-point underflow occurs, 0 otherwise.
  - N** **Negative Condition Flag.** 1 if a negative result is generated, 0 otherwise. In some instructions, this flag is the MSB of the output.
  - Z** **Zero Condition Flag.** 1 if a zero result is generated, 0 otherwise. For logical and shift instructions, 1 if a zero output is generated, 0 otherwise.
  - V** **Overflow Condition Flag.** 1 if an integer or floating-point overflow occurs, 0 otherwise.
  - C** **Carry Flag.** 1 if a carry or borrow occurs, 0 otherwise. For shift instructions, this flag is set to the value of the last bit shifted out; 0 for a shift count of 0.

The seven condition flags are stored in the status register (ST). They provide information about the properties of the result or output of arithmetic or logical operations.

## EXAMPLE *Example Instruction*

---

**Mode Bit** **OVM Overflow Mode Flag.** In general, integer operations are affected by the OVM bit value (described in Table 3–2 on page 3-6).

**Example** INST @98AEh, R5

### **Before Instruction:**

DP = 80h

R5 = 07 6690 0000h = 2.30562500e+02

Memory at 0080 98AEh = 5CDFh = 1.00001107e + 00

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### **After Instruction:**

DP = 80h

R5 = 00 6690 0000h = 1.80126953e + 00

Memory at 80 98AEh = 5CDFh = 1.00001107e + 00

LUF LV UF N Z V C = 0 0 0 0 0 0 0

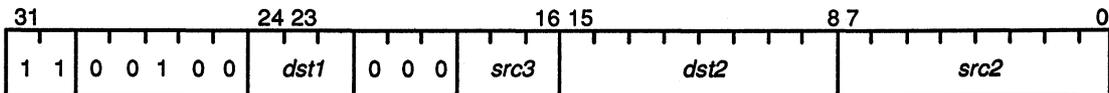
The sample code presented in the above format shows the effect of the code on system pointers (e.g., DP or SP), registers (e.g., R1 or R5), memory at specific locations, and the seven status bits. The values given for the registers include the leading zeros to show the exponent in floating-point operations. Decimal conversions are provided for all register and memory locations. The seven status bits are listed in the order in which they appear in the assembler and simulator (see Section 11.2 on page 11-10 and Table 11–8 on page 11-12 for further information on these seven status bits).



**Syntax**        **ABSF** *src2, dst1*  
                   || **STF** *src3, dst2*

**Operation**     |*src2* | → *dst1*  
                   || *src3* → *dst2*

**Operands**    *src2* indirect (disp = 0, 1, IR0, IR1)  
                   *dst1* register (R0 – R7)  
                   *src3* register (R0 – R7)  
                   *dst2* indirect (disp = 0, 1, IR0, IR1)

**Encoding**


**Description** A floating-point absolute value and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (ABSF) writes to the same register, then STF accepts as input the contents of the register before it is modified by the ABSF.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*. If *src3* and *dst1* point to the same register, *src3* is read before the write to *dst1*.

An overflow occurs if *src*(man) = 80000000h and *src*(exp) = 7Fh. The result is *dst*(man) = 7FFFFFFFh and *dst*(exp) = 7Fh.

**Cycles**        1

**Status Bits**   **LUF** Unaffected.  
                   **LV** 1 if a floating-point overflow occurs, unchanged otherwise.  
                   **UF** 0.  
                   **N** 0.  
                   **Z** 1 if a zero result is generated, 0 otherwise.  
                   **V** 1 if a floating-point overflow occurs, 0 otherwise.  
                   **C** Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

**Example**

```

    ABSF  *++AR3 (IR1) ,R4
||     STF  R4,*- AR7 (1)

```

**Before Instruction:**

AR3 = 80 9800h

IR1 = 0AFh

R4 = 733C0 0000h = 1.79750e + 02

AR7 = 80 98C5h

Data at 80 98AFh = 58B 4000h = -6.118750e + 01

Data at 80 98C4h = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR3 = 80 98AFh

IR1 = 0AFh

R4 = 574C0 0000h = 6.118750e + 01

AR7 = 80 98C5h

Data at 80 98AFh = 58B 4000h = -6.118750e + 01

Data at 80 98C4h = 733 C000h = 1.79750e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0

## ABS I Absolute Value of Integer

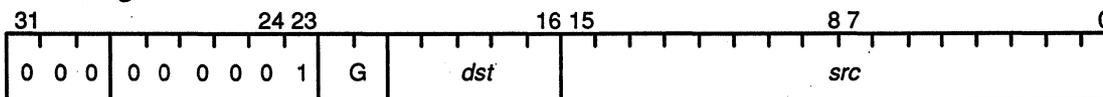
---

**Syntax**      **ABS I** *src*, *dst*

**Operation**     $|src| \rightarrow dst$

**Operands**    *src* general addressing modes (G):  
                  0 0    register (any register in CPU primary register file)  
                  0 1    direct  
                  1 0    indirect  
                  1 1    immediate  
  
                  *dst* register (any register in CPU primary register file)

### Encoding



**Description** The absolute value of the *src* operand is loaded into the *dst* register. The *src* and *dst* operands are assumed to be signed integers.

An overflow occurs if *src* = 8000 0000h. If ST(OVM) = 1, the result is *dst* = 7FFF FFFFh. If ST(OVM) = 0, the result is *dst* = 8000 0000h.

**Cycles**            1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 0.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit**      **OVM** Operation is affected by OVM bit value.

**Example 1** ABSI R0, R0  
or ABSI R0

**Before Instruction:**

R0 = 0FFFF FFCBh = - 53

**After Instruction:**

R0 = 035h = 53

**Example 2** ABSI \*AR1, R3

**Before Instruction:**

AR1 = 20h

R3 = 0h

Data at 20h = 0FFFF FFCBh = - 53

**After Instruction:**

AR1 = 20h

R3 = 35h = 53

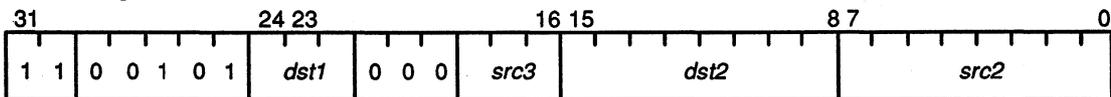
Data at 20h = 0FFFF FFCBh = - 53

**Syntax**        **ABSI** *src2, dst1*  
                   || **STI** *src3, dst2*

**Operation**     |*src2* | → *dst1*  
                   || *src3* → *dst2*

**Operands**    *src2* indirect (disp = 0, 1, IR0, IR1)  
                   *dst1* register (R0 – R7)  
                   *src3* register (R0 – R7)  
                   *dst2* indirect (disp = 0, 1, IR0, IR1)

**Encoding**



**Description** An integer absolute value and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ABSII) writes to the same register, then STI accepts as input the contents of the register before it is modified by the ABSII.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

An overflow occurs if *src* = 8000 0000h. If ST(OVM) = 1, the result is *dst* = 7FFF FFFFh. If ST(OVM) = 0, the result is *dst* = 8000 0000h.

**Cycles**        1

**Status Bits**   **LUF** Unaffected.  
                   **LV** 1 if an integer overflow occurs, unchanged otherwise.  
                   **UF** 0.  
                   **N** 0.  
                   **Z** 1 if a zero result is generated, 0 otherwise.  
                   **V** 1 if an integer overflow occurs, 0 otherwise.  
                   **C** Unaffected.

**Mode Bit**     **OVM** Operation is affected by OVM bit value.

**Example**      ABSI \*-AR5(1),R5  
|| STIR1,\*AR2--(IR1)

**Before Instruction:**

AR5 = 80 99E2h

R5 = 0h

R1 = 42h = 66

AR2 = 80 98FFh

IR1 = 0Fh

Data at 80 99E1h = 0FFFF FFCBh = - 53

Data at 80 98FFh = 2h = 2

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR5 = 80 99E2h

R5 = 35h = 53

R1 = 42h = 66

AR2 = 80 98F0h

IR1 = 0Fh

Data at 80 99E1h = 0FFFF FFCBh = - 53

Data at 80 98FFh = 42h = 66

LUF LV UF N Z V C = 0 0 0 0 0 0 0

## ADDC *Add Integer With Carry*

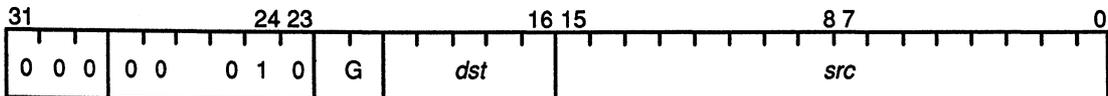
---

**Syntax**     **ADDC** *src, dst*

**Operation**    *dst + src + C* → *dst*

**Operands**    *src* general addressing modes (G):  
                  0 0    register (any register in CPU primary register file)  
                  0 1    direct  
                  1 0    indirect  
                  1 1    immediate  
  
                  *dst* register (any register in CPU primary register file)

### Encoding



**Description** The sum of the *dst* and *src* operands and the C (carry) flag is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

**Cycles**        1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a carry occurs, 0 otherwise.

**Mode Bit**     **OVM** Operation is affected by OVM bit value.

**Example**        ADDC R1, R5

### Before Instruction:

R1 = 00FFFF 5C25h = – 41,947

R5 = 00FFFF 019Eh = – 65,122

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

R1 = 00FFFF 5C25h = – 41,947

R5 = 00FFFE 5DC4h = – 107,068

LUF LV UF N Z V C = 0 0 0 0 0 0 0

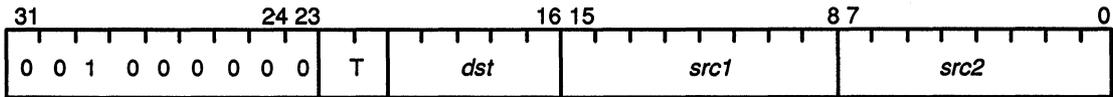
**Syntax**     **ADDC3** *src2, src1, dst*

**Operation**    *src1 + src2 + C → dst*

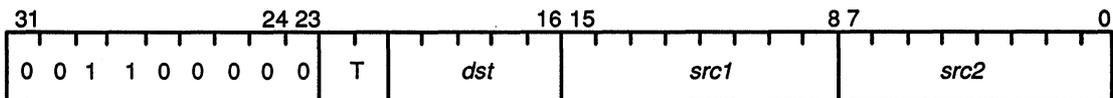
**Operands**    *src1, src2*    both type 1 or type 2 three-operand addressing modes  
                   *dst*             register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	<b>T</b>	<b>src1 addressing modes</b>	<b>src2 addressing modes</b>
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	<b>T</b>	<b>src1 addressing modes</b>	<b>src2 addressing modes</b>
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

## **ADDC3** *Add Integer With Carry, 3 Operands*

---

**Description** The sum of the *src1* and *src2* operands and value of the C (carry) flag is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0, the condition flags are modified if the destination register is R0–R11. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**U** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a carry occurs, 0 otherwise.

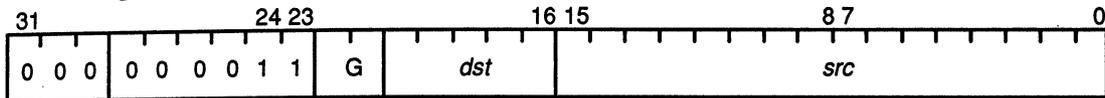
**Mode Bit** **OVM** Operation is affected by OVM bit value.

**Syntax**     **ADDF** *src, dst*

**Operation**   *dst + src → dst*

**Operands**   *src* general addressing modes (G):  
                   0 0   register (R0 – R11)  
                   0 1   direct  
                   1 0   indirect  
                   1 1   immediate  
  
                   *dst* register (R0 – R11)

**Encoding**



**Description**   The sum of the *dst* and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

**Cycles**        1

**Status Bits**   **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.  
                   **LV**  1 if a floating-point overflow occurs, unchanged otherwise.  
                   **UF**  1 if a floating-point underflow occurs, 0 otherwise.  
                   **N**   1 if a negative result is generated, 0 otherwise.  
                   **Z**   1 if a zero result is generated, 0 otherwise.  
                   **V**   1 if an floating-point overflow occurs, 0 otherwise.  
                   **C**   Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

**Example**       ADDF \*AR4++ (IR1) , R5

**Before Instruction:**

AR4 = 80 9800h  
 IR1 = 12Bh  
 R5 = 057980 0000h = 6.23750e+01  
 Data at 80 9800h = 86B 2800h = 4.7031250e + 02  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0 0

**After Instruction:**

AR4 = 80 992Bh  
 IR1 = 12Bh  
 R5 = 09052C 0000h = 5.3268750e+02  
 Data at 80 9800h = 86B 2800h = 4.7031250e + 02  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0 0

## ADDF3 Add Floating-Point Values, 3 Operands

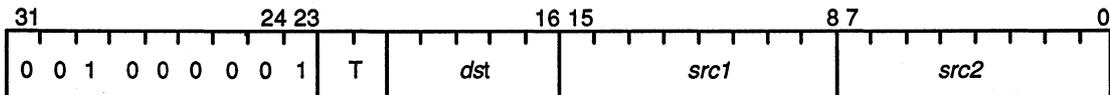
**Syntax**     **ADDF3**   *src2, src1, dst*

**Operation**   *src1 + src2 → dst*

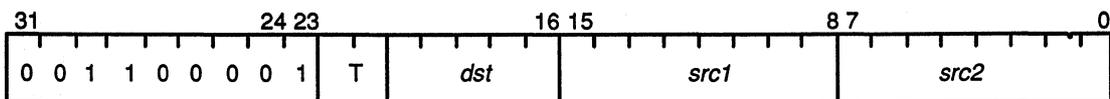
**Operands**   *src1, src2*   both type 1 or type 2 three-operand addressing modes  
                   *dst*            register mode (R0 – R11)

### Encoding

#### Type 1



#### Type 2



### Instruction Word Fields

	T	src1 addressing modes	src2 addressing modes
<b>Type 1</b>	00	register mode (R0 – R11)	register mode (R0 – R11)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (R0 – R11)
	10	register mode (R0 – R11)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	src1 addressing modes	src2 addressing modes
<b>Type 2</b>	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description** The sum of the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be floating-point numbers.

**Cycles** 1

**Status Bits**

- LUF** 1 if a floating-point underflow occurs, unchanged otherwise.
- LV** 1 if a floating-point overflow occurs, unchanged otherwise.
- UF** 1 if a floating-point underflow occurs, 0 otherwise.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if an floating-point overflow occurs, 0 otherwise.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** `ADDF3 *AR1 (2), *+AR1 (8), R4`

**Before Instruction:**

AR1 = 2FF820h

R4 = 0h

Data at 22F F822h = 700 F000h = 1.28940e + 02

Data at 22F F828h = 34C 2000h = 1.27590e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 0 0

**After Instruction:**

AR1 = 2F F820h

R4 = 070DB2 0000h = 1.41695313 e + 02

Data at 22F F828h = 34C 2000h = 1.27590e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 0 0

## ADDF3||STF *Parallel ADDF3 and STF*

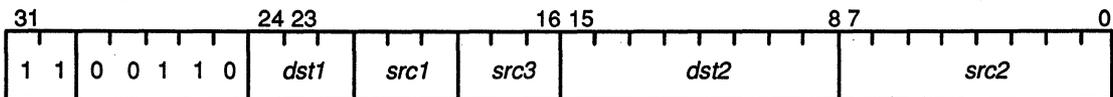
---

**Syntax**      **ADDF3** *src2, src1, dst1*  
              **|| STF** *src3, dst2*

**Operation**    *src1 + src2* → *dst1*  
              **||** *src3* → *dst2*

**Operands**    *src1* register (R0 – R7)  
                  *src2* indirect (disp = 0, 1, IR0, IR1)  
                  *dst1* register (R0 – R7)  
                  *src3* register (R0 – R7)  
                  *dst2* indirect (disp = 0, 1, IR0, IR1)

### Encoding



**Description** A floating-point addition and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (ADDF3) writes to the same register, then STF accepts as input the contents of the register before it is modified by the ADDF3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

**Cycles**      1

**Status Bits** **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.  
**LV** 1 if a floating-point overflow occurs, unchanged otherwise.  
**UF** 1 if a floating-point underflow occurs, 0 otherwise.  
**N** 1 if a negative result is generated, 0 otherwise.  
**Z** 1 if a zero result is generated, 0 otherwise.  
**V** 1 if an floating-point overflow occurs, 0 otherwise.  
**C** Unaffected.

**Mode Bit**    **OVM** Operation is not affected by OVM bit value.

**Example**      ADDF3    \*+AR3 (IR1) , R2, R5  
 ||    STF R4, \*AR2

**Before Instruction:**

AR3 = 809800h  
 IR1 = 0A5h  
 R2 = 070C80 0000h = 1.4050e + 02  
 R5 = 0h  
 R4 = 057B40 0000h = 6.281250e + 01  
 AR2 = 80 98F3h  
 Data at 80 98A5h = 733 C000h = 1.79750e + 02  
 Data at 80 98F3h = 0h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR3 = 80 9800h  
 IR1 = 0A5h  
 R2 = 070C80 0000h = 1.4050e+02  
 R5 = 082020 0000h = 3.20250e + 02  
 R4 = 057B40 0000h = 6.281250e + 01  
 AR2 = 80 98F3h  
 Data at 80 98A5h = 733 C000h = 1.79750e + 02  
 Data at 80 98F3h = 57B 4000h = 6.28125e + 01  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

## ADDI *Add Integer*

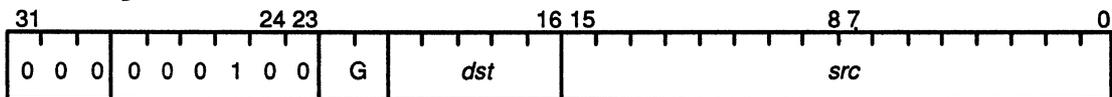
---

**Syntax**     **ADDI** *src, dst*

**Operation**   *dst + src* → *dst*

**Operands**   *src* general addressing modes (G):  
              0 0   register (any register in CPU primary register file)  
              0 1   direct  
              1 0   indirect  
              1 1   immediate  
  
              *dst* register (any register in CPU primary register file)

### Encoding



**Description** The sum of the *dst* and *src* operands is loaded into the the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a carry occurs, 0 otherwise.

**Mode Bit**   **OVM** Operation is affected by OVM bit value.

**Example**     ADDI   R3, R7

### Before Instruction:

R3 = 0FFFF FFCBh = – 53

R7 = 35h = 53

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

R3 = 0FFFF FFCBh = – 53

R7 = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

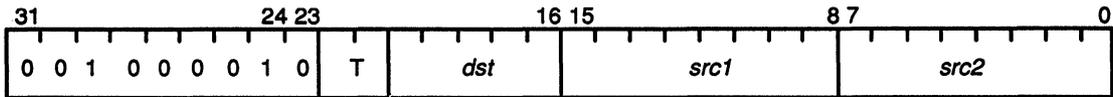
**Syntax**     **ADDI3** *src2, src1, dst*

**Operation**   *src1 + src2* → *dst*

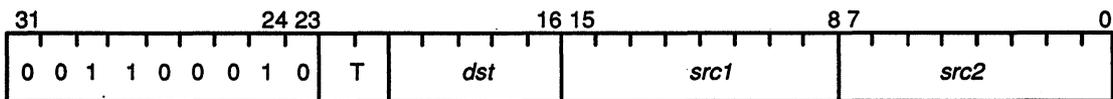
**Operands**   *src1, src2*   both type 1 or type 2 three-operand addressing modes  
                   *dst*             register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	<b>T</b>	<b>src1 addressing modes</b>	<b>src2 addressing modes</b>
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	<b>T</b>	<b>src1 addressing modes</b>	<b>src2 addressing modes</b>
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

## **ADDI3** *Add Integer, 3 Operands*

---

**Description** The sum of the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a carry occurs, 0 otherwise.

**Mode Bit** **OVM** Operation is affected by OVM bit value.



**Example**        ADDI3 \*AR0- -(IR0),R5,R0  
                  || STI R3,\*AR7

**Before Instruction:**

AR0 = 80 992Ch  
IR0 = 0Ch  
R5 = 0DCh = 220  
R0 = 0h  
R3 = 35h = 53  
AR7 = 80 983Bh  
Data at 80 992Ch = 12Ch = 300  
Data at 80 983Bh = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

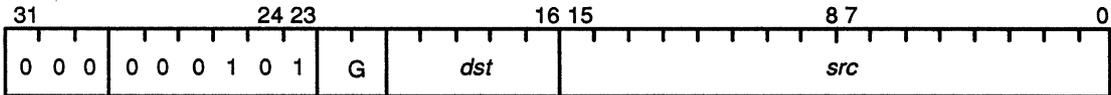
AR0 = 80 9920h  
IR0 = 0Ch  
R5 = 0DCh = 220  
R0 = 208h = 520  
R3 = 35h = 53  
AR7 = 80 983Bh  
Data at 80 992Ch = 12Ch = 300  
Data at 80 983Bh = 35h = 53  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**     **AND** *src*, *dst*

**Operands**   *dst* AND *src* → *dst*

**Operands**   *src* general addressing modes (G):  
                 0 0   register (any register in CPU primary register file)  
                 0 1   direct  
                 1 0   indirect  
                 1 1   immediate (not sign-extended)  
  
                 *dst* register (any register in CPU primary register file)

**Encoding**



**Description** The bitwise logical-AND between the *dst* and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

**Cycles**     1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

**Example**     AND    R1, R2

**Before Instruction:**

R1 = 80h

R2 = 0AFFh

LUF LV UF N Z V C = 0 0 0 0 0 0 1

**After Instruction:**

R1 = 80h

R2 = 80h

LUF LV UF N Z V C = 0 0 0 0 0 0 1

## AND3 Bitwise Logical-AND, 3 Operands

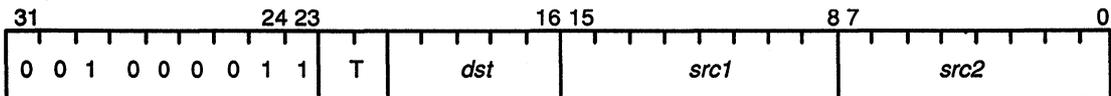
**Syntax**     **AND3** *src2, src1, dst*

**Operation**   *src1 & src2* → *dst*

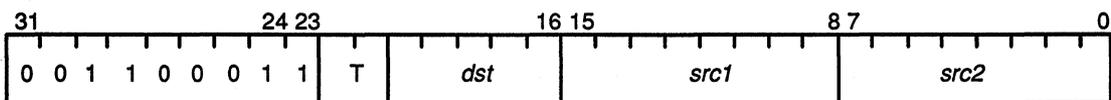
**Operands**   *src1, src2*   both type 1 or type 2 three-operand addressing modes  
                   *dst*            register mode (any register in CPU primary register file)

### Encoding

#### Type 1



#### Type 2



### Instruction Word Fields

	T	src1 addressing modes	src2 addressing modes
Type 1	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	src1 addressing modes	src2 addressing modes
Type 2	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description** The bitwise logical-AND between the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.



**Example**      AND3 \*+AR1 (IR0) , R4 , R7  
              || STI   R3 , \*AR2

**Before Instruction:**

AR1 = 8099F1h  
IR0 = 8h  
R4 = 0A323h  
R7 = 0h  
R3 = 35h = 53  
AR2 = 80 983Fh  
Data at 80 99F9h = 5C53h  
Data at 80 983Fh = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR1 = 80 99F1h  
R0 = 8h  
R4 = 0A323h  
R7 = 03h  
R3 = 35h = 53  
AR2 = 80 983Fh  
Data at 80 99F9h = 5C53h  
Data at 80 983Fh = 35h = 53  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

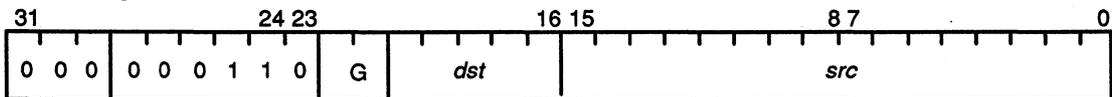
## ANDN Bitwise Logical-AND With Complement

**Syntax**      **ANDN** *src, dst*

**Operation**    *dst* AND  $\sim$ *src*  $\rightarrow$  *dst*

**Operands**    *src* general addressing modes (G):  
                  0 0    register (any register in CPU primary register file)  
                  0 1    direct  
                  1 0    indirect  
                  1 1    immediate (not sign-extended)  
  
                  *dst* register (any register in CPU primary register file)

### Encoding



**Description**    The bitwise logical-AND between the *dst* operand and the bitwise logical complement ( $\sim$ ) of the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

**Cycles**        1

**Status Bits**    If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF**    Unaffected.

**LV**     Unaffected.

**UF**     0.

**N**      MSB of the output.

**Z**      1 if a zero result is generated, 0 otherwise.

**V**      0.

**C**      Unaffected.

**Mode Bit**      **OVM** Operation is not affected by OVM bit value.

**Example**        ANDN @980Ch, R2

#### Before Instruction:

DP = 80h

R2 = 0C2Fh

Data at 80 980Ch = 0A02h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

#### After Instruction:

DP = 80h

R2 = 042Dh

Data at 80 980Ch = 0A02h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

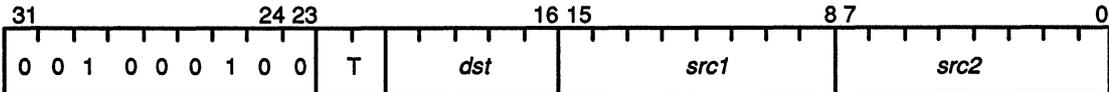
**Syntax**      **ANDN3** *src2, src1, dst*

**Operation**    *src1* AND  $\sim$  *src2*  $\rightarrow$  *dst*

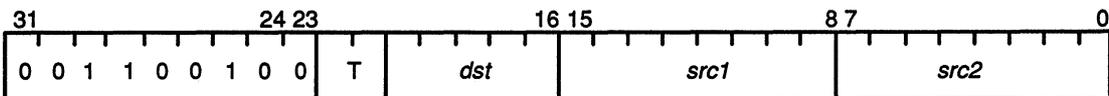
**Operands**    *src1, src2*    both type 1 or type 2 three-operand addressing modes  
                   *dst*                    register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

## **ANDN3** *Bitwise Logical-ANDN, 3 Operands*

---

**Description** The bitwise logical-AND between the *src1* operand and the bitwise logical complement (~) of the *src2* operand is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

**Cycles** 1

**Status Bits**

- LUF** Unaffected.
- LV** Unaffected.
- UF** 0.
- N** MSB of the output.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 0.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

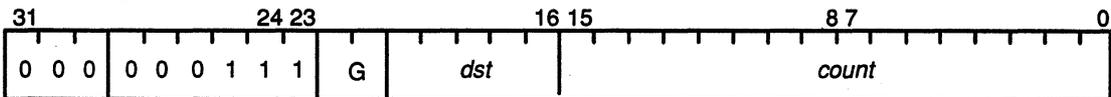
**Syntax**     **ASH** *count*, *dst*

**Operation**   If ( $\text{count} \geq 0$ ):  
                    $\text{dst} \ll \text{count} \rightarrow \text{dst}$   
 Else:  
                    $\text{dst} \gg |\text{count}| \rightarrow \text{dst}$

**Operands**    *count* general addressing modes (G):  
                   0 0 register (any register in CPU primary register file)  
                   0 1 direct  
                   1 0 indirect  
                   1 1 immediate

*dst* register (any register in CPU primary register file)

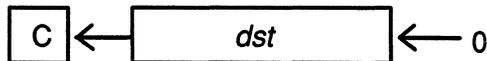
**Encoding**



**Description** The seven least-significant bits of the *count* operand are used to generate the twos-complement shift count of up to 32 bits.

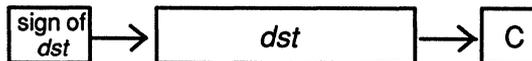
If the *count* operand is greater than zero, the *dst* operand is left-shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Arithmetic left-shift:



If the *count* operand is less than zero, the *dst* operand is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are sign-extended as it is right-shifted. Low-order bits are shifted out through the C (carry) bit.

Arithmetic right-shift:



If the *count* operand is zero, no shift is performed, and the C (carry) bit is set to 0. The *count* and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** Set to the value of the last bit shifted out. 0 for a shift *count* of 0.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example 1** ASH R1, R3

**Before Instruction:**

R1 = 10h = 16

R3 = 0A E000h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R1 = 10h

R3 = 0E0000 0000h

LUF LV UF N Z V C = 0 1 0 1 0 1 0

**Example 2** ASH @98C3h, R5

**Before Instruction:**

DP = 80h

R5 = 0AEC0 0001h

Data at 80 98C3h = 0FFE8 = - 24

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

DP = 80h

R5 = 0FFFF FFAEh

Data at 80 98C3h = 0FFE8 = - 24

LUF LV UF N Z V C = 0 0 0 1 0 0 1

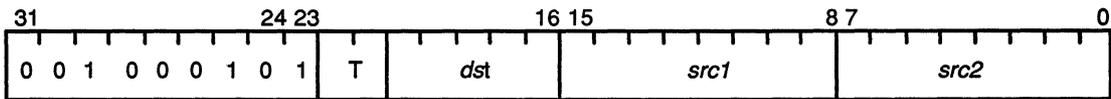
**Syntax**     **ASH3** *count, src, dst*

**Operation**   if (*count* ≥ 0)  
                   *src* << *count* → *dst*  
                   Else:  
                   *src* >> | *count* | → *dst*

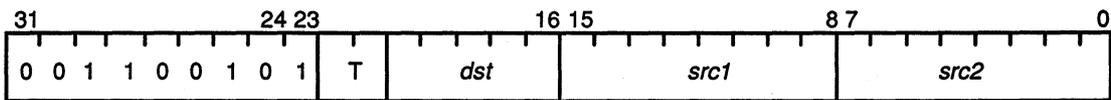
**Operands**   *src, count*   both type 1 or type 2 three-operand addressing modes  
                   *dst*            register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

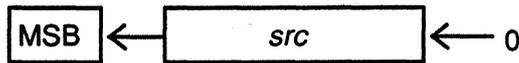
	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description** The seven least-significant bits of the *count* operand are used to generate the two's-complement shift count.

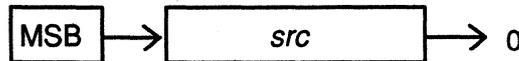
If the *count* operand is greater than zero, the *src* operand is left-shifted by the value of *count*. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the status register's C (carry) bit.

Arithmetic left-shift:



If the *count* operand is less than zero, the *src* operand is right-shifted by the absolute value of *count* (e.g.  $-4 =$  right-shift 4). The high-order bits of the *src* operand are sign-extended as they are right-shifted. Low-order bits are shifted out through the C (carry) bit.

Arithmetic right-shift:



If the *count* operand is zero, no shift is performed, and the C (carry) bit is set to 0. The *count*, *src*, and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** **LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** Set to the value of the last bit shifted out. 0 for a shift *count* of 0.

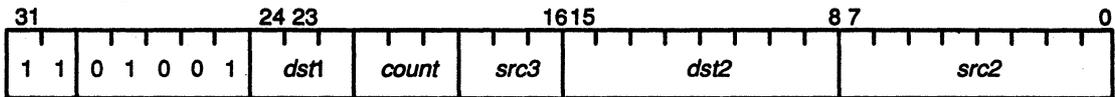
**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Syntax**        **ASH3** *count, src2, dst1*  
 || **STI** *src3, dst2*

**Operation**    If ( $count \geq 0$ ):  
                   *src2*  $\ll count \rightarrow dst1$   
 Else:  
                   *src2*  $\gg |count| \rightarrow dst1$   
 || *src3*  $\rightarrow dst2$

**Operands**     *count* register (R0 — R7)  
*src2* indirect (disp = 0, 1, IR0, IR1)  
*dst1* register (R0 — R7)  
*src3* register (R0 — R7)  
*dst2* indirect (disp = 0, 1, IR0, IR1)

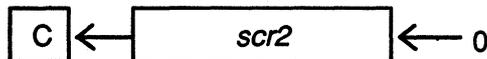
### Encoding



**Description** The seven least-significant bits of the *count* operand register are used to generate the two's-complement shift count of up to 32 bits.

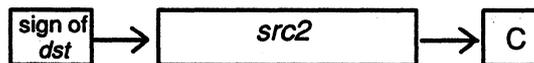
If the *count* operand is greater than zero, the *dst* operand is left-shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Arithmetic left-shift:



If the *count* operand is less than zero, the *dst* operand is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are sign-extended as it is right-shifted. Low-order bits are shifted out through the C (carry) bit.

Arithmetic right-shift:



If the *count* operand is zero, no shift is performed, and the C (carry) bit is set to 0. The *count* and *dst* operands are assumed to be signed integers.

All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (ASH3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the ASH3. If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

**Cycles** 1

**Status Bits**

- LUF** Unaffected.
- LV** 1 if an integer overflow occurs, unchanged otherwise.
- UF** 0.
- N** MSB of the output.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if an integer overflow occurs, 0 otherwise.
- C** Set to the value of the last bit shifted out. 0 for a shift count of 0.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example**

```
ASH3 R1, *AR6++(IR1), R0
|| STI R5, *AR2
```

**Before Instruction:**

```
AR6 = 80 9900h
IR1 = 8Ch
R1 = 0FFE8h = - 24
R0 = 0h
R5 = 35h = 53
AR2 = 80 98A2h
Data at 80 9900h = 0AE00 0000h
Data at 80 98A2h = 0h
LUF LV UF N Z V C = 0 0 0 0 0 0 0
```

**After Instruction:**

```
AR6 = 80 998Ch
IR1 = 8Ch
R1 = 0FFE8h = - 24
R0 = 0FFFF FFAEh
R5 = 35h = 53
AR2 = 80 98A2h
Data at 80 9900h = 0AE00 0000h
Data at 80 98A2h = 35h = 53
LUF LV UF N Z V C = 0 0 0 1 0 0 0
```



## **Bcond** *Branch Conditionally (Standard)*

---

**Example**    BZ R0

**Before Instruction:**

PC = 2B00h

R0 = 0003 FF00h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

PC = 3FF00h

R0 = 0003 FF00h

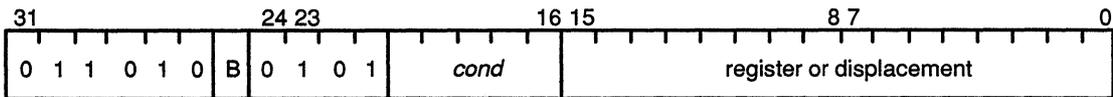
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**     **BcondAF** *src*

**Operation**    If (*cond* is true)  
                   If (*src* is a register)  
                   *src* → PC  
                   If (*src* is a displacement)  
                   *src* + PC of branch + 3 → PC  
                   Else:  
                   If (*cond* is false)  
                   annul execute phase results of next three instructions and continue

**Operands**    *src* conditional-branch addressing modes

**Encoding**



**Instruction Word Fields**

B	<i>src</i> addressing modes
0	register mode
1	PC-relative mode

**Description** If the condition is true, a branch and the three instructions following the branch instruction are executed. If the condition is false, it annuls the effect of the execute phase of the next three instructions and execution continues. If the *src* operand is in register mode, then the contents of the specified register are loaded into the PC. If the *src* operand is in PC-relative mode, then the sum of the PC of the branch instruction + 3 and the *src* is loaded into the PC. In PC-relative mode the *src* field is interpreted as a 16-bit signed integer.

None of the three instructions following the **BcondAF** may be an instruction that modifies the program flow. Interrupts are disabled for the duration of the **BcondAF** instruction. **BcondAF** is particular useful for controlling the exit at the bottom of a loop.

## **BcondAF** *Branch Conditionally Delayed and Annul If False*

---

**Cycles** 1

**Status Bits** **LUF** Unaffected.  
**LV** Unaffected.  
**UF** Unaffected.  
**N** Unaffected.  
**Z** Unaffected.  
**V** Unaffected.  
**C** Unaffected.

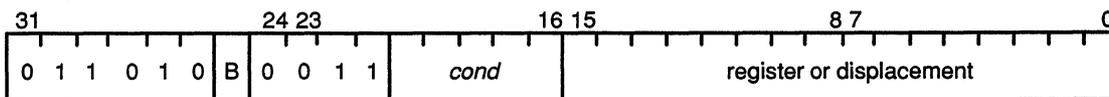
**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Syntax**     **BcondAT** *src*

**Operation**    If (*cond* is true)  
                   If (*src* is a register)  
                   *src* → PC  
                   annul execute phase results of next three instructions.  
                   If (*src* is a displacement)  
                   *src* + PC of branch +3 → PC  
                   annul execute phase results of next three instructions.  
                   Else, continue.

**Operands**    *src* conditional-branch addressing modes

**Encoding**



**Instruction Word Fields**

B	src addressing modes
0	register mode
1	PC-relative mode

**Description** If the condition is true, it performs a branch and annuls the effect of the execute phase of the next three instructions. If the *src* operand is expressed in register mode, then the contents of the specified register are loaded into the PC. If the *src* operand is in PC-relative mode, then the sum of the PC of the branch instruction + 3 and the *src* is loaded into the PC. In PC-relative mode, the *src* field is interpreted as a 16-bit signed integer.

None of the three instructions following the *BcondAT* may be an instruction that modifies the program flow. Interrupts are disabled for the duration of the *BcondAT* instruction.

*BcondAT* instruction will not annul the status signals at the external interfaces. The *BcondAT* is particular useful for controlling the entry at the top of the loop.

## **BcondAT** *Branch Conditionally Delayed and Annul If True*

---

**Cycles** 1

**Status Bits** **LUF** Unaffected.  
**LV** Unaffected.  
**UF** Unaffected.  
**N** Unaffected.  
**Z** Unaffected.  
**V** Unaffected.  
**C** Unaffected.

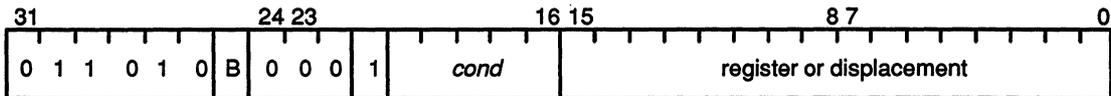
**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Syntax** **BcondD** *src*

**Operation** If *cond* is true:  
 If *src* is in register addressing mode (any register in CPU primary register file)  
     *src* → PC.  
 If *src* is in PC-relative mode (label or address),  
     displacement + PC + 3 → PC.  
 Else, continue.

**Operands** *src* conditional-branch addressing modes (B):  
     0     register  
     1     PC-relative

**Encoding**



**Description** **BcondD** signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch, and the three instructions following **BcondD** will not affect the *cond*. None of the three instructions following **BcondD** may be an instruction that modifies program flow.

A branch is performed if the condition is true. If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label – (PC of branch instruction + 3). This displacement is stored as a 16-bit signed integer in the 16 least significant bits of the branch instruction. This displacement is added to the PC of the branch instruction plus 3 to generate the new PC. The TMS320C40 provides 20 condition codes that can be used with this instruction (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags).

**Cycles** 1

**Status Bits** **LUF** Unaffected.  
**LV** Unaffected.  
**UF** Unaffected.  
**N** Unaffected.  
**Z** Unaffected.  
**V** Unaffected.  
**C** Unaffected.

**Mode Bit** **OV**M Operation is not affected by OVM bit value.

## **BcondD** *Branch Conditionally (Delayed)*

---

**Example** BNZD 36 (36 = 24h)

**Before Instruction:**

PC = 50h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

PC = 77h

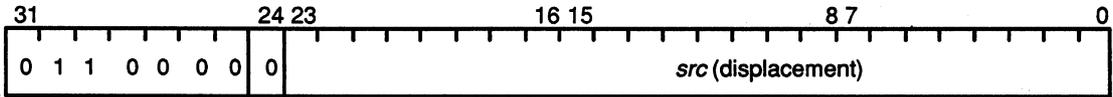
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**     **BR** *src*

**Operation**    $PC + 1 + src \rightarrow PC$

**Operands**   *src* 24-bit signed immediate displacement

**Encoding**



**Description** Performs an unconditional delayed branch. The *src* operand is assumed to be a 24-bit signed integer.

**Cycles**     4

**Status Bits**

- LUF** Unaffected.
- LV** Unaffected.
- UF** Unaffected.
- N** Unaffected.
- Z** Unaffected.
- V** Unaffected.
- C** Unaffected.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

## BRD *Branch Unconditionally (Delayed)*

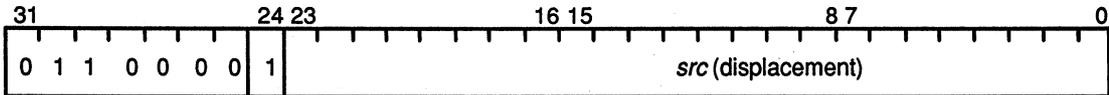
---

**Syntax** BRD *src*

**Operation** PC + 3 + *src* → PC

**Operands** *src* 24-bit signed immediate displacement

**Encoding**



**Description** Performs an unconditional delayed branch. The *src* operand is assumed to be a 24-bit signed integer. Interrupts are disabled during the BRD instruction.

The three instructions following the BRD instruction are fetched and executed. None of these three instructions may modify the program flow (e.g., affect the PC value).

**Cycles** 1

**Status Bits**

- LUF** Unaffected.
- LV** Unaffected.
- UF** Unaffected.
- N** Unaffected.
- Z** Unaffected.
- V** Unaffected.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.



## CALLcond *Call Subroutine Conditionally*

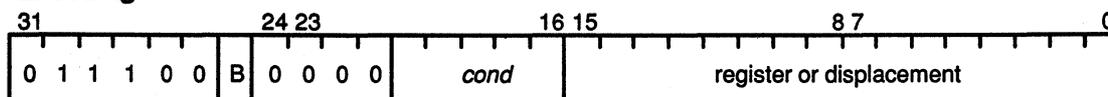
---

**Syntax**     **CALLcond src**

**Operation**   If *cond* is true:  
                Next PC  $\rightarrow$  \*++SP  
                If *src* is in register addressing mode (any register in CPU primary register file),  
                  *src*  $\rightarrow$  PC.  
                If *src* is in PC-relative mode (label or address),  
                  displacement + PC + 1  $\rightarrow$  PC.  
                Else, continue.

**Operands**   *src* conditional-branch addressing modes (B):  
                0     register  
                1     PC-relative

### Encoding



**Description** A call is performed if the condition is true. If the condition is true, the next PC value is pushed onto the system stack. If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative mode, the assembler generates a displacement: displacement = label – (PC of call instruction + 1). This displacement is stored as a 16-bit signed integer in the 16 least significant bits of the call instruction word. This displacement is added to the PC of the call instruction plus 1 to generate the new PC.

The TMS320C40 provides 20 condition codes that can be used with this instruction (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags).

**Cycles**     5

**Status Bits**   **LUF** Unaffected.  
                  **LV**  Unaffected.  
                  **UF**  Unaffected.  
                  **N**   Unaffected.  
                  **Z**   Unaffected.  
                  **V**   Unaffected.  
                  **C**   Unaffected.

**Mode Bit**    **OVM** Operation is not affected by OVM bit value.

**Example** CALLNZ R5

**Before Instruction:**

PC = 123h

SP = 80 9835h

R5 = 789h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

PC = 789h

SP = 80 9836h

R5 = 789h

Data at 80 9836h = 124h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

## CMPF Compare Floating-Point Values

---

**Syntax**      **CMPF** *src*, *dst*

**Operation**    *dst* – *src*

**Operands**    *src* general addressing modes (G):

    0 0    register (R0 – R11)

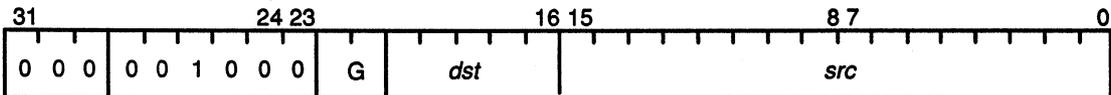
    0 1    direct

    1 0    indirect

    1 1    immediate

*dst* register (R0 – R11)

### Encoding



**Description**    The *src* operand is subtracted from the *dst* operand. The result is not loaded into any register, thus allowing for nondestructive compares. The *dst* and *src* operands are assumed to be floating-point numbers.

**Cycles**        1

**Status Bits**    **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.  
    **LV** 1 if a floating-point overflow occurs, unchanged otherwise.  
    **UF** 1 if a floating-point underflow occurs, 0 otherwise.  
    **N** 1 if a negative result is generated, 0 otherwise.  
    **Z** 1 if a zero result is generated, 0 otherwise.  
    **V** 1 if a floating-point overflow occurs, 0 otherwise.  
    **C** Unaffected.

**Mode Bit**      **OVM** Operation is not affected by OVM bit value.

**Example**        `CMPF *+AR4, R6`

### Before Instruction:

AR4 = 80 98F2h

R6 = 070C80 0000h = 1.4050e+02

Data at 80 98F3h = 070C 8000h = 1.4050e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

AR4 = 80 98F2h

R6 = 070C80 0000h = 1.4050e + 02

Data at 80 98F3h = 070C 8000h = 1.4050e + 02

LUF LV UF N Z V C = 0 0 0 0 1 0 0

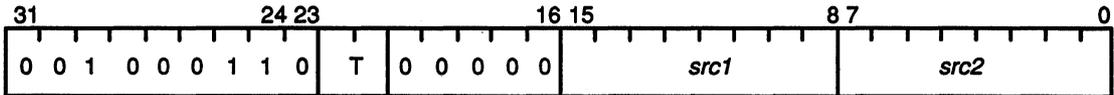
**Syntax** **CMPF3** *src2*, *src1*

**Operation** *src1* – *src2*

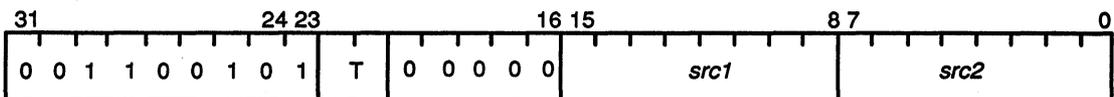
**Operands** *src1* – *src2* both type 1 or type 2 three-operand addressing modes

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 1</b>	00	register mode (R0 — R11)	register mode (R0 — R11)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (R0 — R11)
	10	register mode (R0 — R11)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 2</b>	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

### **CMPF3** *Compare Floating-Point Values, 3 Operands*

---

**Description** The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register. This allows for nondestructive compares. The *src1* and *src2* operands are assumed to be floating-point numbers.

**Cycles** 1

**Status Bits**

- LUF** 1 if a floating-point underflow occurs, unchanged otherwise.
- LV** 1 if a floating-point overflow occurs, unchanged otherwise.
- UF** 1 if a floating-point underflow occurs, 0 otherwise.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if a floating-point overflow occurs, 0 otherwise.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Syntax** **CMPI** *src, dst*

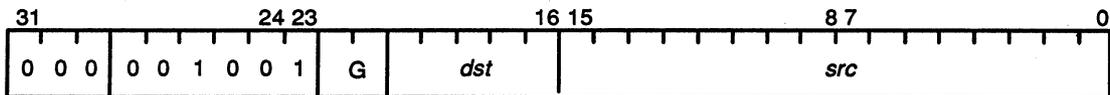
**Operation** *dst* – *src*

**Operands** *src* general addressing modes (G):

0 0	register (any register in CPU primary register file)
0 1	direct
1 0	indirect
1 1	immediate

*dst* register (any register in CPU primary register file)

**Encoding**



**Description** The *src* operand is subtracted from the *dst* operand. The result is not loaded into any register, thus allowing for nondestructive compares. The *dst* and *src* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits**

- LUF** Unaffected.
- LV** 1 if an integer overflow occurs, unchanged otherwise.
- UF** 0.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if an integer overflow occurs, 0 otherwise.
- C** 1 if a borrow occurs, 0 otherwise.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** `CMPI R3, R7`

**Before Instruction:**

R3 = 898h = 2200  
 R7 = 3E8h = 1000  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R3 = 898h = 2200  
 R7 = 3E8h = 1000  
 LUF LV UF N Z V C = 0 0 0 1 0 0 0

## CMPI3 *Compare Integer, 3 Operands*

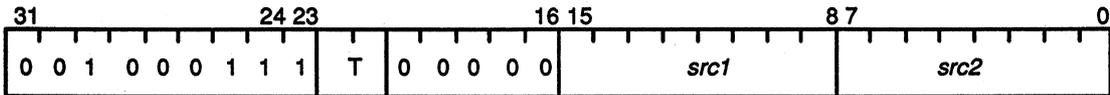
**Syntax**     **CMPI3** *src2, src1*

**Operation**   *src1 – src2*

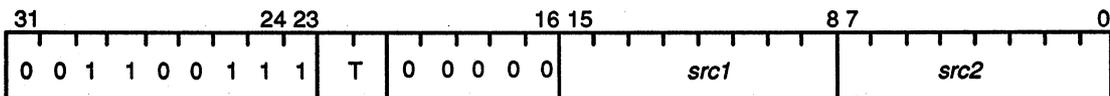
**Operands**   *src1 – src2* both type 1 or type 2 three-operand addressing modes

### Encoding

#### Type 1



#### Type 2



### Instruction Word Fields

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
Type 1	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
Type 2	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description** The *src2* operand is subtracted from the *src1* operand. The result is not loaded into any register. This allows for nondestructive compares. The *src1* and *src2* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** **LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a borrow occurs, 0 otherwise.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

## DBcond Decrement and Branch Conditionally (Standard)

---

**Syntax** DBcond ARn, src

**Operation** ARn - 1 → ARn

If *cond* is true and ARn ≥ 0 :

If *src* is in register addressing mode (any register in CPU primary register file),

*src* → PC.

If *src* is in PC-relative mode (label or address),

displacement + PC + 1 → PC.

Else, continue.

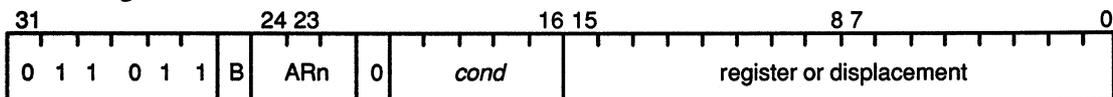
**Operands** *src* conditional-branch addressing modes (B):

0 register

1 PC-relative

ARn register (any register in CPU primary register file)

### Encoding



**Description** DBcond signifies a standard branch that executes in four cycles because the pipeline must be flushed if *cond* is true. The specified auxiliary register is decremented and a branch is performed if the condition is true and the specified auxiliary register is greater than or equal to zero.

The auxiliary register is treated as a 32-bit signed integer. The most significant eight bits are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 32 least significant bits of the auxiliary register. Note that the branch condition does not depend on the auxiliary register decrement.

If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* operand is expressed in PC-relative addressing mode, the assembler generates a displacement: displacement = label - (PC of branch instruction + 1). This integer is stored as a 16-bit signed integer in the 16 least significant bits of the branch instruction word. This displacement is added to the PC of the branch instruction plus 1 to generate the new PC.

The TMS320C40 provides 20 condition codes that can be used with this instruction (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags).

**Cycles** 4

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** DBLT AR3, R2

**Before Instruction:**

PC = 5Fh

AR3 = 12h

R2 = 9Fh

LUF LV UF N Z V C = 0 0 0 1 0 0 0

**After Instruction:**

PC = 9Fh

AR3 = 11h

R2 = 9Fh

LUF LV UF N Z V C = 0 0 0 1 0 0 0

## DBcondD *Decrement and Branch Conditionally (Delayed)*

---

**Syntax** DBcondD ARn, src

**Operation** ARn – 1 → ARn

If *cond* is true and ARn ≥ 0:

If *src* is in register addressing mode (any register in CPU primary register file),

src → PC

If *src* is in PC-relative mode (label or address)

displacement + PC + 3 → PC.

Else, continue.

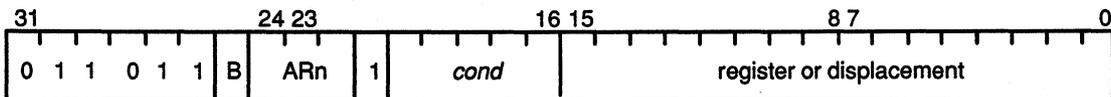
**Operands** src conditional-branch addressing modes (B):

0 register

1 PC-relative

ARn register (any register in CPU primary register file)

### Encoding



**Description** DBcondD signifies a delayed branch that allows the three instructions after the delayed branch to be fetched before the PC is modified. The effect is a single-cycle branch. The specified auxiliary register is decremented and a branch is performed if the condition is true and the specified auxiliary register is greater than or equal to zero. (The three instructions following the DBcondD must not affect the *cond*).

The auxiliary register is treated as a 32-bit signed integer. The most significant eight bits are unmodified by the decrement operation. The comparison of the auxiliary register uses only the 32 least significant bits of the auxiliary register. Note that the branch condition does not depend on the auxiliary register decrement.

If the *src* operand is expressed in register addressing mode, the contents of the specified register are loaded into the PC. If the *src* is expressed in PC-relative addressing, the assembler generates a displacement: displacement = label – (PC of branch instruction + 3). This displacement is added to the PC of the branch instruction plus 3 to generate the new PC. Note that bit 21 = 1 for a delayed branch.

The TMS320C40 provides 20 condition codes that can be used with this instruction (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags).

**Cycles**        1

**Status Bits**   **LUF** Unaffected.  
                   **LV**  Unaffected.  
                   **UF**  Unaffected.  
                   **N**  Unaffected.  
                   **Z**  Unaffected.  
                   **V**  Unaffected.  
                   **C**  Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

**Example**     DBZD     AR5, \$+110h

**Before Instruction:**

PC = 0h  
 AR5 = 67h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

PC = 110h  
 AR5 = 66h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

## FIX Floating-Point to Integer Conversion

---

**Syntax**     **FIX** *src, dst*

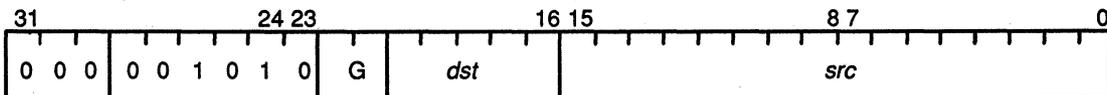
**Operation**   *fix(src) → dst*

**Operands**   *src* general addressing modes (G):

0 0	register (R0 — R11)
0 1	direct
1 0	indirect
1 1	immediate

*dst* register (any register in CPU primary register file)

### Encoding



**Description** The floating-point operand *src* is converted to the nearest integer less than or equal to it in value, and the result is loaded into the *dst* register. The *src* operand is assumed to be a floating-point number and the *dst* operand a signed integer.

The exponent field of the result register (if it has one) is not modified.

Integer overflow occurs when the floating-point number is too large to be represented as a 32-bit twos-complement integer. In the case of integer overflow, the result will be saturated in the direction of overflow.

**Cycles**     1

**Status Bits** If ST (SET COND) = 0 and, the condition flags are modified the destination register is R0 — R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

**Example**      `FIX R1,R2`

**Before Instruction:**

R1 = 0A2820 0000h = 1.3454e + 3

R2 = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R1 = 0A2820 0000h = 13454e + 3

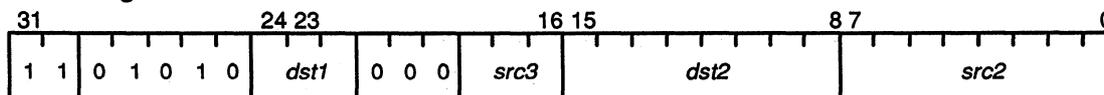
R2 = 541h = 1345

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**            **FIX** *src2, dst1*  
                       || **STI** *src3, dst2*

**Operation**        *fix(src2) → dst1*  
                       || *src3 → dst2*

**Operands**        *src2* indirect (disp = 0, 1, IR0, IR1)  
                       *dst1* register (R0 – R7)  
                       *src3* register (R0 – R7)  
                       *dst2* indirect (disp = 0, 1, IR0, IR1)

**Encoding**


**Description** A floating-point-to-integer conversion is performed. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register, and the operation being performed in parallel (FIX) writes to the same register, then STI accepts as input the contents of the register before it is modified by FIX.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

Integer overflow occurs when the floating-point number is too large to be represented as a 32-bit twos-complement integer. In the case of integer overflow, the result will be saturated in the direction of overflow.

**Cycles**            1

**Status Bits**    **LUF** Unaffected.  
                       **LV** 1 if an integer overflow occurs, unchanged otherwise.  
                       **UF** 0.  
                       **N** 1 if a negative result is generated, 0 otherwise.  
                       **Z** 1 if a zero result is generated, 0 otherwise.  
                       **V** 1 if an integer overflow occurs, 0 otherwise.  
                       **C** Unaffected.

**Mode Bit**        **OVM** Operation is not affected by OVM bit value.

**Example**      FIX      \*++AR4(1),R1  
                 || STI      R0,\*AR2

**Before Instruction:**

AR4 = 80 98A2h

R1 = 0h

R0 = 0DCh = 220

AR2 = 80 983Ch

Data at 80 98A3h = 733 C000h = 1.7950e + 02

Data at 80 983Ch = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR4 = 80 98A3h

R1 = 0B3h = 179

R0 = 0DCh = 220

AR2 = 80 983Ch

Data at 80 98A3h = 733 C000h = 1.79750e + 02

Data at 80 983Ch = 0DCh = 220

LUF LV UF N Z V C = 0 0 0 0 0 0 0

## FLOAT Integer to Floating-Point Conversion

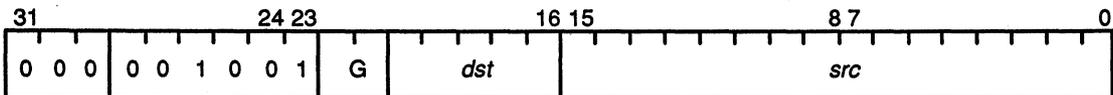
---

**Syntax**      **FLOAT** *src, dst*

**Operation**   float (*src*) → *dst*

**Operands**    *src* general addressing modes (G):  
                  0 0    register (any register in CPU primary register file)  
                  0 1    direct  
                  1 0    indirect  
                  1 1    immediate  
  
                  *dst* register (R0 – R11)

### Encoding



**Description**   The integer operand *src* is converted to the floating-point value equal to it, and the result loaded into the *dst* register. The *src* operand is assumed to be a signed integer, and the *dst* operand a floating-point number.

**Cycles**        1

**Status Bits**   **LUF** Unaffected.  
                  **LV**    Unaffected.  
                  **UF**    0.  
                  **N**    1 if a negative result is generated, 0 otherwise.  
                  **Z**    1 if a zero result is generated, 0 otherwise.  
                  **V**    0.  
                  **C**    Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

**Example**      **FLOAT** \*++AR2(2), R5

### Before Instruction:

AR2 = 80 9800h  
R5 = 034C 2000h = 1.27578125e + 01  
Data at 80 9802h = 0AEh = 174  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

AR2 = 80 9802h  
R5 = 072E0 0000h = 1.74e + 02  
Data at 80 9802h = 0AEh = 174  
LUF LV UF N Z V C = 0 0 0 0 0 0 0



**Example**

```
    FLOAT *+AR2(IR0),R6
||   STF R7,*AR1
```

**Before Instruction:**

AR2 = 80 98C5h

IR0 = 8h

R6 = 0h

R7 = 034C20 0000h = 1.27578125e + 01

AR1 = 80 9933h

Data at 80 98CDh = 0AEh = 174

Data at 80 9933h = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR2 = 80 98C5h

IR0 = 8h

R6 = 072E00 0000h = 1.740e + 02

R7 = 034C20 0000h = 1.27578125e + 01

AR1 = 80 9933h

Data at 80 98CDh = 0AEh = 174

Data at 80 9933h = 034C 2000h = 1.27578125e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 0

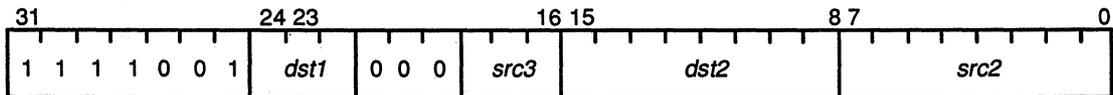


**Syntax**     **FRIEEE** *src2, dst1*  
               **|| STF** *src3, dst2*

**Operation**   convert *src2* from IEEE format → *dst1*  
                   in parallel with  
                   *src3* → *dst2*

**Operands**   *src2*   indirect mode (disp = 0, 1, IR0, IR1)  
                   *dst1*   register mode (R0 – R7)  
                   *src3*   register mode (R0 – R7)  
                   *dst2*   indirect mode (disp = 0, 1, IR0, IR1)

**Encoding**



**Description** The *src2* operand is converted from the IEEE floating-point format to the two's-complement format. The converted result goes into an extended-precision register *dst1* as a single-precision floating-point number.

A floating-point store is done in parallel.

If *src2* and *dst2* point to the same location, then *src2* is read before the write to *dst2*.

**Cycles**       1

**Status Bits**   **LUF** Unaffected.  
                   **LV**  Set if overflow, otherwise unchanged.  
                   **UF**  0.  
                   **N**  Sign of the result.  
                   **Z**  1 if result is 0, 0 otherwise.  
                   **V**  1 if overflow, 0 otherwise.  
                   **C**  Unaffected

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.



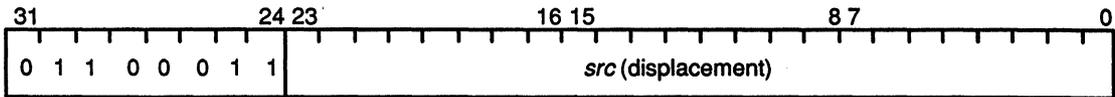


**Syntax** LAJ *src*

**Operation** PC of LAJ + 4 → extended-precision register R11  
*src* + 3 + PC of LAJ → PC

**Operands** *scr* 24-bit signed immediate displacement

**Encoding**



**Description** LAJ performs a single cycle subroutine call. The three instructions following the LAJ instruction are performed. The return address (address of the LAJ instruction + 4) is placed in extended-precision register R11. The address branched to is formed by adding the *src* operand to the PC of the LAJ instruction + 3.

None of the three instructions following the LAJ instruction should modify the program flow. Interrupts are disabled for the duration of the LAJ instruction.

**Cycles** 1

**Status Bits** LUF Unaffected.  
 LV Unaffected.  
 UF Unaffected.  
 N Unaffected.  
 Z Unaffected.  
 V Unaffected.  
 C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

## LAJcond *Link and Jump Conditionally*

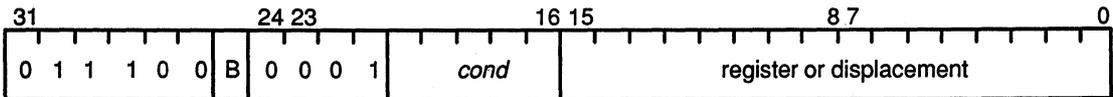
---

**Syntax** LAJcond *src*

**Operation** If (*cond* is true)  
If (*src* is a register)  
PC of LAJcond + 4 → extended-precision register R11  
*src* → PC  
If (*src* is a displacement)  
PC of LAJcond + 4 → extended-precision register R11  
*src* + PC of the LAJ + 3 → PC  
Else, continue.

**Operands** *src* conditional-branch addressing modes

### Encoding



### Instruction Word Fields

<b>B</b>	<b><i>src</i> addressing modes</b>
0	register mode
1	PC-relative mode

**Description** LAJcond performs a conditional single-cycle subroutine call. The three instructions following the LAJcond instruction are performed. The return address (address of the LAJ instruction + 4) is placed in extended-precision register R11. The address branched to is formed by either register mode or PC-relative mode.

None of the three instructions following the LAJcond instruction may modify the program flow. Interrupts are disabled for the duration of the LAJcond instruction.

**Cycles** 1

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

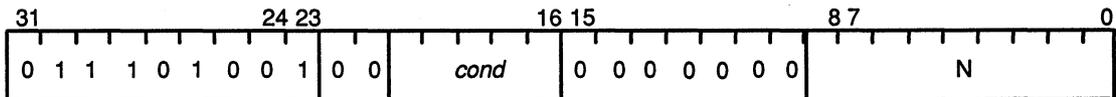
**Syntax**     **LATcond N**

**Operation**   If (*cond* is true)  
                   ST(GIE) → ST(PGIE)  
                   ST(CF) → ST(PCF)  
                   0 → ST(GIE)  
                   1 → ST(CF)  
                   PC of *LATcond* + 4 → extended-precision register R11  
                   trap vector N → PC

Else, continue.

**Operands**   N immediate mode – trap number ( $0 \leq N \leq 511$ )

**Encoding**



**Description** Performs a delayed conditional trap. If traps are to be nested, you may need to save the status register before executing *LATcond*. If the condition is true, ST bits GIE and CF are saved in PGIE and PCF in the status register. Then all interrupts are disabled (0 → GIE), and the cache is frozen (1 → CF). The contents of the PC of the *LATcond* + 4 are placed in R31, and the PC is loaded with the contents of the specified trap vector (N). If the condition is not true, then continue normal operation.

The three instructions following *LATcond* will be fetched and executed. They may not be instructions that modify the program flow or modify the status register.

**Cycles**     1

**Status Bits**   **LUF** Unaffected.  
                   **LV**   Unaffected.  
                   **UF**   Unaffected.  
                   **N**    Unaffected.  
                   **Z**    Unaffected.  
                   **V**    Unaffected.  
                   **C**    Unaffected.

**Mode Bit**    **OVM** Operation is not affected by OVM bit value.



**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LB2 R1, R2 ; sign extended byte 2 of R1 → R2

**Before Instruction:**

R1 = 00AB 0000h

R2 = 0000 0000h

**After Instruction:**

R1 = 00AB 0000h

R2 = FFFF FFABh

## **LBUb** Load Byte Unsigned

---

**Syntax**     **LBUb** *src*, *dst*

**Operation**   Byte (3, 2, 1, 0) of *src* → *dst*

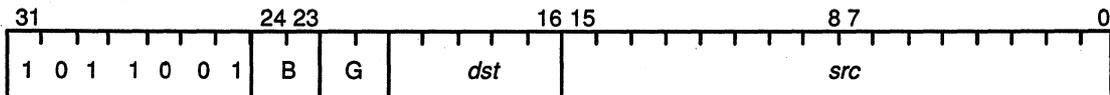
b = byte to load (3, 2, 1, 0)

3	2	1	0
---	---	---	---

 = b (byte designator 3 – 0)

**Operands**   *src* register, direct, or indirect addressing modes  
*dst* register mode (any register in CPU primary register file)

### **Encoding**



### **Instruction Word Fields**

<b>G</b>	<b>src addressing modes</b>
00	register mode (any CPU register)
01	direct mode
10	indirect mode

<b>B</b>	<b>src byte</b>
00	byte 0 LS byte
01	byte 1
10	byte 2
11	byte 3 MS byte

**Description**   The specified byte of the *src* operand is right-shifted without sign-extension, into the 8 LSBs of the *dst* register. The *src* byte is unsigned.

**Cycles**       1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** 0.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LB2 R1, R2

**Before Instruction:**

R1 = 00AB 0000h

R2 = 0000 0000h

**After Instruction:**

R1 = 00AB 0000h

R2 = 0000 00ABh

## LDA Load Address Register

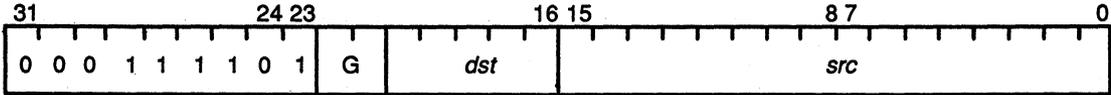
---

**Syntax** LDA

**Operation**  $src \rightarrow dst$

**Operands**  $src$  general addressing modes  
 $dst$  register mode (address registers only)

**Encoding**



**Instruction Word Fields**

G	src addressing modes
00	register mode (any CPU register)
01	direct mode
10	indirect mode
11	immediate mode

**Description** The  $src$  operand is loaded into the  $dst$  register. The  $dst$  register may be any of the address registers: AR0 – AR7, IR0, IR1, DP, BK or SP. The load is done by the end of the read phase of the pipeline. As a result, LDA is one cycle faster than LDI for loading these registers. (All operands are treated as signed integers.)

**Cycles** 1

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.



## **LDEP** Load Integer From Expansion Register File to Primary Register File

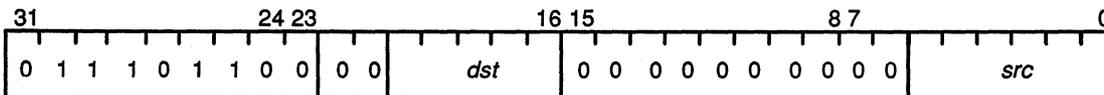
---

**Syntax**     **LDEP** *src*, *dst*

**Operation**   *src* → *dst*

**Operands**   *src* expansion register file register (IVTP or TVTP)  
              *dst* register mode (any register in CPU primary register file)

**Encoding**



**Description** This is a means to load a CPU register with the contents of the IVTP register (interrupt-trap table pointer) or the TVTP register. These registers are described in Section 3.2.

The *src* operand register from the expansion-register file is loaded into the *dst* register in the primary register file. The *dst* register content is assumed to be an integer.

**Cycles**     1

**Status Bits**   **LUF** Unaffected.  
                  **LV**  Unaffected.  
                  **UF**  Unaffected.  
                  **N**  Unaffected.  
                  **Z**  Unaffected.  
                  **V**  Unaffected.  
                  **C**  Unaffected.

**Mode Bit**    **OVM** Operation is not affected by OVM bit value.



## LDFcond *Load Floating-Point Value Conditionally*

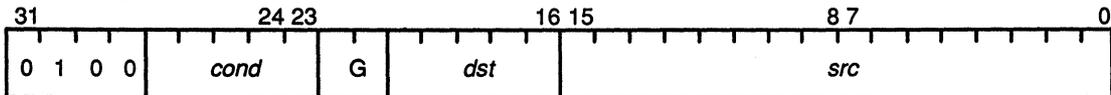
---

**Syntax**     **LDFcond** *src, dst*

**Operation**   If *cond* is true:  
              *src* → *dst*.  
              Else:  
              *dst* is unchanged.

**Operands**   *src* general addressing modes (G):  
              0 0   register (R0 – R11)  
              0 1   direct  
              1 0   indirect  
              1 1   immediate  
  
              *dst* register (R0 – R11)

### Encoding



**Description** If the condition is true, the *src* operand is loaded into the *dst* register. Otherwise, the *dst* register is unchanged. The *dst* and *src* operands are assumed to be floating-point numbers.

The TMS320C40 provides 20 condition codes that can be used with this instruction (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags). Note that an LDFU (load floating-point unconditionally) instruction is useful for loading R0 – R11 without affecting condition flags.

**Cycles**     1

**Status Bits**   **LUF** Unaffected.  
                  **LV**   Unaffected.  
                  **UF**   Unaffected.  
                  **N**    Unaffected.  
                  **Z**    Unaffected.  
                  **V**    Unaffected.  
                  **C**    Unaffected.

**Mode Bit**    **OVM** Operation is not affected by OVM bit value.

**Example** LDFZ R3,R5

**Before Instruction:**

R3 = 2CFF2C D500h = 1.77055560e +13

R5 = 5F0000 003Eh = 3.96140824e + 28

LUF LV UF N Z V C = 0 0 0 0 1 0 0

**After Instruction:**

R3 = 2CFF2C D500h = 1.77055560e +13

R5 = 2CFF2C D500h = 1.77055560e +13

LUF LV UF N Z V C = 0 0 0 0 1 0 0

## LDFI *Load Floating-Point Value, Interlocked*

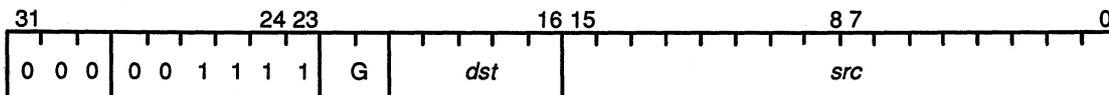
---

**Syntax** LDFI *src, dst*

**Operation** Signal interlocked operation.  
*src* → *dst*

**Operands** *src* general addressing modes (G):  
0 1 direct  
1 0 indirect  
*dst* register (R0 – R11)

### Encoding



**Description** The *src* operand is loaded into the *dst* register. An interlocked operation is signaled over LOCK or LLOCK. The *src* and *dst* operands are assumed to be floating-point numbers. Note that only direct and indirect modes are allowed. Refer to Section 6.5 (page 6-13) and Section 7.7 (page 7-39) for detailed descriptions.

**Cycles** 1

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF 0.  
N 1 if a negative result is generated, 0 otherwise.  
Z 1 if a zero result is generated, 0 otherwise.  
V 0.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** LDFI *\*+AR2, R7*

#### Before Instruction:

AR2 = 8098F1h

R7 = 0h

Data at 80 98F2h = 584 C000h = - 6.28125e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 0

#### After Instruction:

AR2 = 8098F1h

R7 = 0584C0 0000h = - 6.28125e + 01

Data at 80 98F2h = 584 C000h = - 6.28125e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 1



**Example**      LDF - AR1(IR0),R7  
                 || LDF \*AR7++(1),R3

**Before Instruction:**

AR1 = 80 985Fh

IR0 = 8h

R7 = 0h

AR7 = 80 988Ah

R3 = 0h

Data at 80 9857h = 70C 8000h = 1.4050e + 02

Data at 80 988Ah = 57B 4000h = 6.281250e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR1 = 80 9857h

R0 = 8h

R7 = 070C80 0000h = 1.4050e + 02

AR7 = 80 988Bh

R3 = 057B40 0000h = 6.281250e + 01

Data at 80 9857h = 70C 8000h = 1.4050e + 02

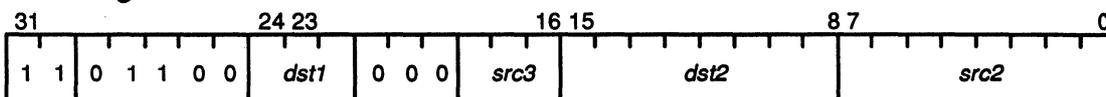
Data at 80 988Ah = 57B 4000h = 6.281250e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**        **LDF** *src2*, *dst1*  
 ||        **STF** *src3*, *dst2*

**Operation**     *src2* → *dst1*  
 ||        *src3* → *dst2*

**Operands**     *src2* indirect (disp = 0, 1, IR0, IR1)  
*dst1* register (R0 – R7)  
*src3* register (R0 – R7)  
*dst2* indirect (disp = 0, 1, IR0, IR1)

**Encoding**

**Description** A floating-point load and a floating-point store are performed in parallel.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

**Cycles**        1

**Status Bits**   **LUF** Unaffected.  
**LV**    Unaffected.  
**UF**    Unaffected.  
**N**     Unaffected.  
**Z**     Unaffected.  
**V**     Unaffected.  
**C**     Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

**Example**      LDF\*AR2- -(1),R1  
                 || STF R3,\*AR4++(IR1)

**Before Instruction:**

AR2 = 80 98E7h  
R1 = 0h  
R3 = 057B40 0000h = 6.28125e + 01  
AR4 = 80 9900h  
IR1 = 10h  
Data at 80 98E7h = 70C 8000h = 1.4050e + 02  
Data at 80 9900h = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR2 = 80 98E6h  
R1 = 070C80 0000h = 1.4050e + 02  
R3 = 057B40 0000h = 6.28125e + 01  
AR4 = 80 9910h  
IR1 = 10h  
Data at 80 98E7h = 70C 8000h = 1.4050e + 02  
Data at 80 9900h = 57B 4000h = 6.28125e + 01  
LUF LV UF N Z V C = 0 0 0 0 0 0 0



## LDI Load Integer

---

**Syntax** LDI *src*, *dst*

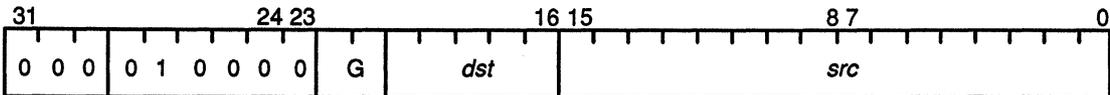
**Operation** *src* → *dst*

**Operands** *src* general addressing modes (G):

0 0	register (any register in CPU primary register file)
0 1	direct
1 0	indirect
1 1	immediate

*dst* register (any register in CPU primary register file)

### Encoding



**Description** The *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers. An alternate form of LDI, LDP, is used to load the data page pointer register (DP) or any other register with the eight MSBs of a relocatable address. See the LDP instruction in this chapter and subsection 11.3.2 (on page 11-15).

**Cycles** 1

**Status Bits**

- LUF** Unaffected.
- LV** Unaffected.
- UF** 0.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 0.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LDI \*--AR1 (IR0) , R5

**Before Instruction:**

AR1 = 2Ch

IR0 = 5h

R5 = 3C5h = 965

Data at 27h = 26h = 38

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR1 = 2Ch

IR0 = 5h

R5 = 26h = 38

Data at 27h = 26h = 38

LUF LV UF N Z V C = 0 0 0 0 0 0 0

## LDIcond Load Integer Conditionally

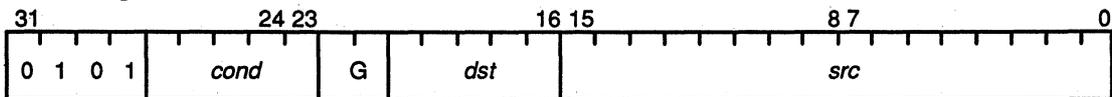
---

**Syntax**     **LDIcond** *src*, *dst*

**Operation**   If *cond* is true:  
                  *src* → *dst*,  
Else:  
                  *dst* is unchanged.

**Operands**   *src* general addressing modes (G):  
                  0 0 register (any register in CPU primary register file)  
                  0 1 direct  
                  1 0 indirect  
                  1 1 immediate  
  
                  *dst* register (any register in CPU primary register file)

### Encoding



**Description** If the condition is true, the *src* operand is loaded into the *dst* register. Otherwise, the *dst* register is unchanged. The *dst* and *src* operands are assumed to be signed integers.

The TMS320C40 provides 20 condition codes that can be used with this instruction (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags). Note that an LDIU (load integer unconditionally) instruction is useful for loading a selected CPU register without affecting the condition flags.

**Cycles**     1

**Status Bits** **LUF** Unaffected.  
**LV** Unaffected.  
**UF** Unaffected.  
**N** Unaffected.  
**Z** Unaffected.  
**V** Unaffected.  
**C** Unaffected.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

**Example** LDIZ R4, R6

**Before Instruction:**

R4 = 027Ch = 636

R6 = 0FE2h = 4,066

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R4 = 027Ch = 636

R6 = 0FE2h = 4,066

LUF LV UF N Z V C = 0 0 0 0 0 0 0

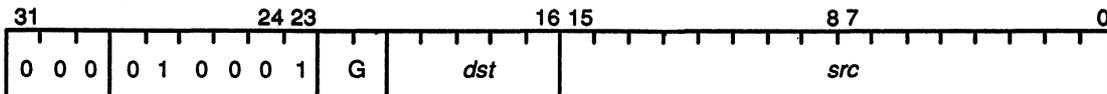
## LDII *Load Integer, Interlocked*

**Syntax** LDII *src, dst*

**Operation** Signal interlocked operation.  
*src* → *dst*

**Operands** *src* general addressing modes (G):  
0 1 direct  
1 0 indirect  
*dst* register (any register in CPU primary register file)

### Encoding



**Description** The *src* operand is loaded into the *dst* register. An interlocked operation is signaled over  $\overline{\text{LOCK}}$  or  $\overline{\text{LLOCK}}$ . The *src* and *dst* operands are assumed to be signed integers. Note that only the direct and indirect modes are allowed. Refer to Section 7.7 on page 7-39 for a detailed description.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LDII @985Fh, R3

### Before Instruction:

DP = 80

R3 = 0h

Data at 80 985Fh = 0DCh

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

DP = 80

R3 = 0DCh

Data at 80 985Fh = 0DCh

LUF LV UF N Z V C = 0 0 0 0 0 0 0



**Example**        LDI \*-AR1(1),R7  
                  || LDI \*AR7++(IR0),R1

**Before Instruction:**

AR1 = 80 9826h  
R7 = 0h  
AR7 = 80 98C8h  
IR0 = 10h  
R1 = 0h  
Data at 80 9825h = 0FAh = 250  
Data at 80 98C8h = 2EEh = 750  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR1 = 80 9826h  
R7 = 0FAh = 250  
AR7 = 80 98D8h  
IR0 = 10h  
R1 = 02EEh = 750  
Data at 80 9825h = 0FAh = 250  
Data at 80 98C8h = 2EEh = 750  
LUF LV UF N Z V C = 0 0 0 0 0 0 0



**Example**        LDI \*-AR1(1),R2  
                  || STI R7,\*AR5++(IR0)

**Before Instruction:**

AR1 = 80 98E7h  
R2 = 0h  
R7 = 35h = 53  
AR5 = 80 982Ch  
IR0 = 8h  
Data at 80 98E6h = 0DCh = 220  
Data at 80 982Ch = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR1 = 80 98E7h  
R2 = 0DCh = 220  
R7 = 35h = 53  
AR5 = 80 9834h  
IR0 = 8h  
Data at 80 98E6h = 0DCh = 220  
Data at 80 982Ch = 35h = 53  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax** LDM *src, dst*

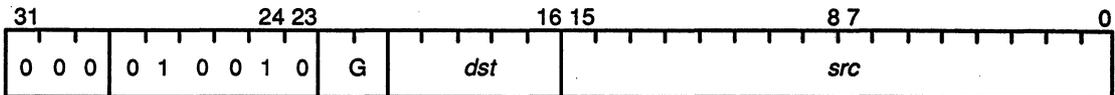
**Operation** *src* (man) → *dst* (man)

**Operands** *src* general addressing modes (G):

0 0 register (R0 – R11)  
 0 1 direct  
 1 0 indirect  
 1 1 immediate

*dst* register (R0 – R11)

**Encoding**



**Description** The mantissa field of the *src* operand is loaded into the mantissa field of the *dst* register. The *dst* exponent field is not modified. The *src* and *dst* operands are assumed to be floating-point numbers. If immediate addressing mode is used, bits 15 – 12 of the instruction word are forced to 0 by the assembler. If the source is in the memory, the 32-bit data are loaded into the mantisa field.

**Cycles** 1

**Status Bits** LUF Unaffected.  
 LV Unaffected.  
 UF Unaffected.  
 N Unaffected.  
 Z Unaffected.  
 V Unaffected.  
 C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** LDM 156.75, R2 (156.75 = 07 1CC0 0000h)

**Before Instruction:**

R2 = 0h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R2 = 00 1CC0 0000h = 1.22460938e + 00  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

## LDP Load Data Page Pointer

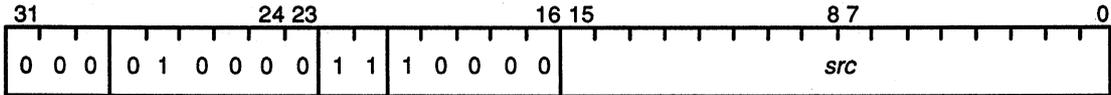
---

**Syntax** LDP *src*[,DP]

**Operation** *src* → Data page pointer

**Operands** *src* is the **16 MSBs** of the **absolute 32-bit** source address (*src*).  
*dst* is optional (data page pointer understood if “,DP” left out of operand)

### Encoding



**Description** This pseudo-op is an alternate form of the LDI instruction, except that LDP *is always* in the immediate addressing mode (bits 22 – 21 = 11<sub>2</sub>). The 16 MSBs of the *src* absolute 32-bit value (note that an *src* less than 32 bits will be zero filled to make the 32 bits) are loaded into the 16 LSBs of the data page pointer. (For example, an *src* of *any* 16-bit value will result in 16 zeroes placed in the DP (the 16 extended zeroes used to fill the *MSBs of the src* value)).

The 16 LSBs of the pointer are used in direct addressing as a pointer to the page of data being addressed. There is a total of 256 pages, each page 64K words long. Bits 31 – 16 of the pointer are reserved and should be kept to zero.

**Cycles** 1

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** LDP @809900h, DP  
or  
LDP @809900h

### Before Instruction:

DP = 6465h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

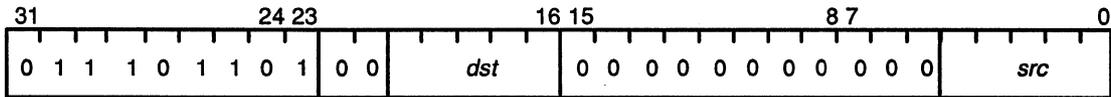
DP = 0080h (16 MSBs of 32-bit *src*, zeroes extended)  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax** LDPE src, dst

**Operation** src → dst

**Operands** src register mode (any register in CPU primary register file)  
dst expansion register file register (IVTP or TVTP)

**Encoding**



**Description** This is a means to load the IVTP register (interrupt-vector table pointer) or TVTP register (trap-vector table pointer). These registers are described in Section 3.2 on page 3-15.

The *src* operand register from the primary-register file is loaded into the *dst* register in the expansion register file. The *dst* operand is assumed to be an integer.

**Cycles** 1

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** LDPE AR, TVTP ; set trap-vector pointer



**Syntax** LHw *src*, *dst*

**Operation** Sign-extended half-word (0, 1) of *src* → *dst*

w = half-word to load (0, 1)

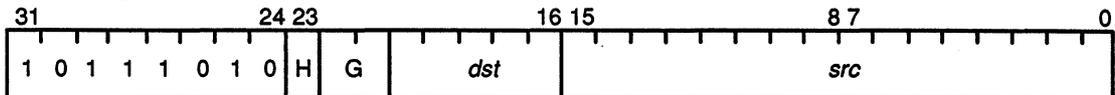
1	0
---	---

 = w designator

**Operands** *src* register, direct, or indirect addressing modes

*dst* register mode (any register in CPU primary register file)

**Encoding**



**Instruction Word Fields**

G	<b><i>src</i> addressing modes</b>
00	register mode ( $R_n$ , $0 \leq n \leq 31$ )
01	direct mode
10	indirect mode

H	<b><i>src</i> half-word</b>
0	half-word 0 (LS half-word)
1	half-word 1 (MS half-word)

**Description** The specified half-word of the *src* operand is sign-extended and right-shifted into the 16 LSBs of the *dst* register. The *src* half-word is signed.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LH0 R1, R2

**Before Instruction:**

R1 = ABCD EF12h

R2 = 1234 5678h

**After Instruction:**

R1 = ABCD EF12h

R2 = FFFF FF12h

**Syntax**     **LHUw** *src*. *dst*

**Operation**   Unsigned half-word (0, 1) of *src* → *dst*

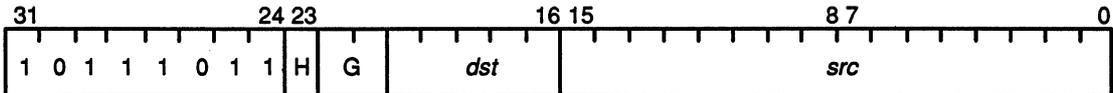
w = half-word to load (0, 1)

1	0
---	---

 = w designator

**Operands**   *src* register, direct, or indirect addressing modes  
*dst* register mode (any register in CPU primary register file)

**Encoding**



**Instruction Word Fields**

<b>G</b>	<b>src addressing modes</b>
00	register mode (any CPU register)
01	direct mode
10	indirect mode

<b>H</b>	<b>src half-word</b>
0	half-word 0 (LS half-word)
1	half-word 1 (MS half-word)

**Description**   The specified half-word of the *src* operand is unsigned and right-shifted into the 16 LSBs of the *dst* register. The *src* half-word is unsigned.

**Cycles**        1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0–R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** 0.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LHU0 R1, R2

**Before Instruction:**

R1 = ABCD EF12h

R2 = 1234 5678h

**After Instruction:**

R1 = ABCD EF12h

R2 = 0000 EF12h

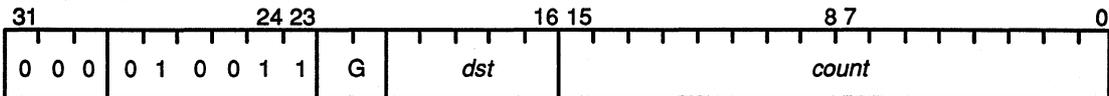
**Syntax**     **LSH** *count, dst*

**Operation**   If *count* ≥ 0:  
                   *dst* << *count* → *dst*  
                   Else:  
                   *dst* >> |*count*| → *dst*

**Operands**   *dst* general addressing modes (G):  
                   0 0   register (any CPU register)  
                   0 1   direct  
                   1 0   indirect  
                   1 1   immediate

*dst* register (any register in CPU primary register file)

**Encoding**



**Description**   The seven least significant bits of the *count* operand are used to generate the two's-complement shift count. If the *count* operand is greater than zero, the *dst* operand is left-shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Logical left-shift:

$$C \leftarrow dst \leftarrow 0$$

If the *count* operand is less than zero, the *dst* is right-shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are zero-filled as they are shifted to the right. Low-order bits are shifted out through the C (carry) bit.

Logical right-shift:

$$0 \rightarrow dst \rightarrow C$$

If the *count* operand is 0, no shift is performed, and the C (carry) bit is set to 0. The *count* operand is assumed to be a signed integer, and the *dst* operand is assumed to be an unsigned integer.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero output is generated, 0 otherwise.

**V** 0.

**C** Set to the value of the last bit shifted out. 0 for a shift *count* of 0.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LSH R4,R7

**Before Instruction:**

R4 = 018h = 24

R7 = 02ACh

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R4 = 018h = 24

R7 = 0AC00 0000h

LUF LV UF N Z V C = 0 0 0 1 0 1 0

**Example** LSH \*-AR5 (IR1), R5

**Before Instruction:**

AR5 = 80 9908h

IR0 = 4h

R5 = 00 12C0 0000h

Data at 80 9904h = 0FFF FFFF4h = -12

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR5 = 80 9908h

IR0 = 4h

R5 = 00 0001 2C00h

Data at 80 9904h = 0FFF FFFF4h = -12

LUF LV UF N Z V C = 0 0 0 0 0 0 0

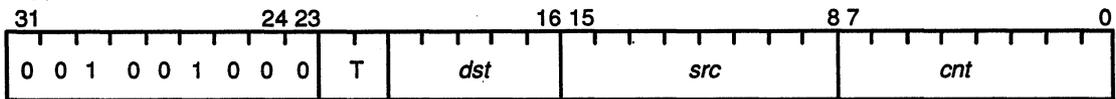
**Syntax**     **LSH3** *count, src, dst*

**Operation**   If *count* ≥ 0:  
                   *src* << *count* → *dst*  
 Else:  
                   *src* >> |*count*| → *dst*

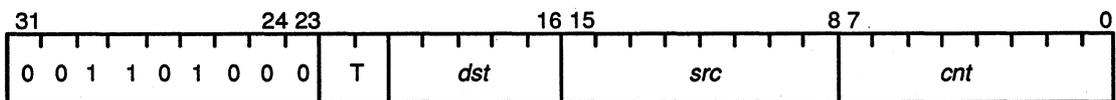
**Operands**   *src, count*   both type 1 or type 2 three-operand addressing modes  
                   *dst*            register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

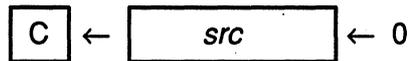
	T	src1 addressing modes	src2 addressing modes
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	src1 addressing modes	src2 addressing modes
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description** The seven least significant bits of the *count* operand are used to generate the twos-complement shift *count*.

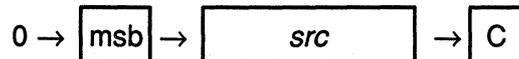
If the *count* operand is greater than zero, the *dst* operand is left shifted by the value of the *count* operand. Low-order bits shifted in are zero-filled, and high-order bits are shifted out through the C (carry) bit.

Logical left-shift:



If the *count* operand is less than zero, the *src* operand is right shifted by the absolute value of the *count* operand. The high-order bits of the *dst* operand are zero-filled as shifted to the right. Low-order bits are shifted out through the C (carry) bit.

Logical right-shift:



If the *count* operand is 0, no shift is performed and the C (carry) bit is set to 0. The *count* operand is assumed to be a signed integer. The *src* and *dst* operands are assumed to be unsigned integers.

If *count* is greater than 32, the LSB ends up in the carry (C) bit. If *count* is less than -32, 0 ends up in the carry bit. This also applies to LSH.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero output is generated, 0 otherwise.

**V** 0.

**C** Set to the value of the last bit shifted out. 0 for a shift *count* of 0. Unaffected if *dst* is not R0 – R7.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.



**Cycles** 1

**Status Bits** **LUF** Unaffected.  
**LV** Unaffected  
**UF** 0.  
**N** MSB of the output.  
**Z** 1 if a zero output is generated, 0 otherwise.  
**V** 0.  
**C** Set to the value of the last bit shifted out. 0 for a shift *count* of 0.

**Mode Bit** **OVM** Operation is affected by OVM bit value.

**Example**        LSH3 R2, \*++AR3(1), R0  
                  || STI R4, \*-AR5

**Before Instruction:**

R2 = 18h = 24  
AR3 = 8098C2h  
R0 = 0h  
R4 = 0DCh = 220  
AR5 = 80 98A3h  
Data at 80 98C3h = 0ACh  
Data at 80 98A2h = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R2 = 18h = 24  
AR3 = 8098C3h  
R0 = 0AC00 0000h  
R4 = 0DCh = 220  
AR5 = 80 98A3h  
Data at 80 98C3h = 0ACh  
Data at 80 98A2h = 0DCh = 220  
LUF LV UF N Z V C = 0 0 0 1 0 1 0

**Example**      LSH3R7, \*AR2- - (1), R2  
 || STI R0, \*+AR0 (1)

**Before Instruction:**

R7 = 0FFFFFFF4h = -12  
 AR2 = 80 9863h  
 R2 = 0h  
 R0 = 12Ch = 300  
 AR0 = 80 98B7h  
 Data at 80 9863h = 2C00 0000h  
 Data at 80 98B8h = 0h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R7 = 0FFFFFFF4h = -12  
 AR2 = 80 9862h  
 R2 = 2C000h  
 R0 = 12Ch = 300  
 AR0 = 80 98B7h  
 Data at 80 9863h = 2C00 0000h  
 Data at 80 98B8h = 12Ch = 300  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0



**Status Bits** If ST (SET COND) = 0 and the destination register is R0–R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** `LWL2 R1, R2`

**Before Instruction:**

R1 = ABCD EF12h

R2 = 1234 5678h

**After Instruction:**

R1 = ABCD EF12h (remains unchanged)  
EF12 0000h (left shifted interim value)

R2 = EF12 5678h (contents merged)

## LWRct *Load Word Right-Shifted*

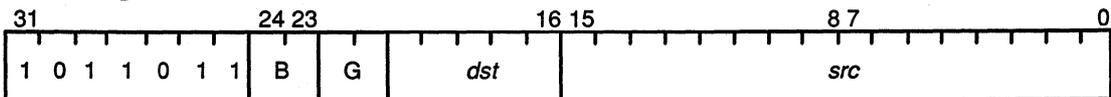
---

**Syntax** LWRct *src, dst*

**Operation** *src* >> {0, 1, 2, or 3} bytes and merged with *dst* → *dst*

**Operands** *ct* the count of bytes {0, 1, 2, or 3} to shift right ( $ct \times 8 = \text{shift in bits}$ )  
*src* register, direct, or indirect addressing modes  
*dst* register mode (any register in CPU primary register file)

### Encoding



### Instruction Word Fields

G	<i>src</i> addressing modes
00	register mode (any CPU register)
01	direct mode
10	indirect mode

B	<i>src</i> byte
00	no shift
01	shift right 1 byte space
10	shift right 2 byte spaces
11	shift right 3 byte spaces

**Description** The *src* operand is right shifted the specified number of bytes and merged with the bytes of the *dst* register that are above the right-shifted MSB of the *src* register. Sign is not extended.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** LWR1 AR1, R2

**Before Instruction:**

AR1 = ABCD EF12h

R2 = 1234 5678h

**After Instruction:**

AR1 = ABCD EF12h (remains unchanged)  
00AB CDEFh (right-shifted interim value)

R2 = 12AB CDEFh (contents merged)

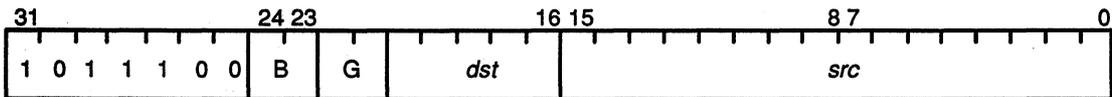
## MBct Merge Byte, Left-Shifted

**Syntax** MBct *src*, *dst*

**Operation** 8 LSBs of *src* << {0, 1, 2, or 3} bytes and merged with *dst* → *dst*

**Operands** *ct* the count of bytes {0, 1, 2, 3} to shift left ( $ct \times 8 =$  shift in bits)  
*src* register, direct, or indirect addressing modes  
*dst* register mode (any register in CPU primary register file)

### Encoding



### Instruction Word Fields

G	<i>src</i> addressing modes
00	register mode (any CPU register)
01	direct mode
10	indirect mode

B	<i>src</i> byte
00	no shift
01	shift left 1 byte space
10	shift left 2 byte spaces
11	shift left 3 byte spaces

**Description** The 8 LSBs of the *src* operand are left shifted (0, 1, 2, or 3) bytes and merged with the *dst* register.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** MB2 AR1, AR2

**Before Instruction:**

AR1 = ABCD EF12h

AR2 = 1234 5678h

**After Instruction:**

AR1 = ABCD EF12h (remains unchanged)  
00AB CDEFh (left-shifted interim value)

AR2 = 1212 5678h (contents merged)

## MHct Merge Half-Word, Left-Shifted

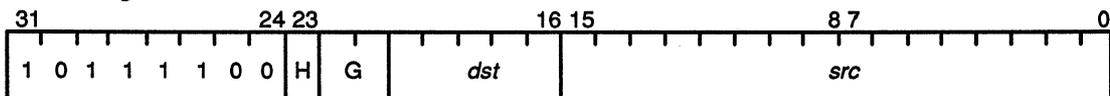
---

**Syntax** MHct *src*, *dst*

**Operation** 16 LSBs of *src* << {0, 1} half-words merged with *dst* → *dst*

**Operands** *ct* the count of half-word (16-bit) shifts  
*src* register, direct, or indirect addressing modes  
*dst* register mode (any register in CPU primary register file)

### Encoding



### Instruction Word Fields

G	<i>src</i> addressing modes
00	register mode (any CPU register)
01	direct mode
10	indirect mode

H	<i>src</i> byte
00	no shift
01	shift left 1 half-word (16 bits)

**Description** The 16 LSBs of the *src* operand are left shifted (0, 1) half-words and merged with the *dst* register.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** MH1 AR1, AR2

**Before Instruction:**

AR1 = ABCD EF12h

AR2 = 1234 5678h

**After Instruction:**

AR1 = ABCD EF12h (remains unchanged)  
EF12 0000h (left-shifted interim value)

AR2 = EF12 5678h (contents merged)

## MPYF Multiply Floating-Point Values

---

**Syntax** MPYF *src, dst*

**Operation**  $dst \times src \rightarrow dst$

**Operands** *src* general addressing modes (G):

0 0 register (R0 – R11)

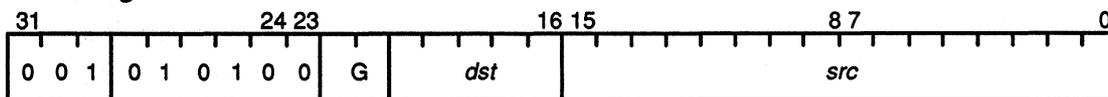
0 1 direct

1 0 indirect

1 1 immediate

*dst* register (R0 – R11)

### Encoding



**Description** The product of the *dst* and *src* operands is loaded into the *dst* register. The *src* operand is assumed to be a single-precision floating-point number, and the *dst* operand is an extended-precision floating-point number.

**Cycles** 1

**Status Bits** **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.

**LV** 1 if a floating-point overflow occurs, unchanged otherwise.

**UF** 1 if a floating-point underflow occurs, 0 otherwise.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if a floating-point overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** MPYF R0, R2

#### Before Instruction:

R0 = 07 0C80 0000h = 1.4050e + 02

R2 = 03 4C20 0000h = 1.27578125e + 01

LUF LV UF N Z V C = 0 0 0 0 0 0 0

#### After Instruction:

R0 = 07 0C80 0000h = 1.4050e + 02

R2 = 0A 600F 2000h = 1.79247266e + 03

LUF LV UF N Z V C = 0 0 0 0 0 0 0



### **MPYF3** *Multiply Floating Point Values, 3 Operands*

---

**Cycles** 1

**Status Bits**

- LUF** 1 if a floating-point underflow occurs, unchanged otherwise.
- LV** 1 if a floating-point overflow occurs, unchanged otherwise.
- UF** 1 if a floating-point underflow occurs, 0 otherwise.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if a floating-point is overflow occurs, 0 otherwise.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.



Any combination of addressing modes may be coded for the four possible source operands as long as two are coded as indirect and two are register. The assignment of the source operands *srcA* – *srcD* to the *src1* – *src4* fields varies, depending on the combination of addressing modes used; the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations in order to simplify processing.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

**Cycles** 1

**Status Bits** **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.  
**LV** 1 if a floating-point overflow occurs, unchanged otherwise.  
**UF** 1 if a floating-point underflow occurs, 0 otherwise.  
**N** 0.  
**Z** 0.  
**V** 1 if a floating-point overflow occurs, 0 otherwise.  
**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example**            MPYF3 R5++(1), \*- - AR1 (IR0), R0  
 ||    ADDF3 R5, R7, R3

**Before Instruction:**

AR5 = 80 98C5h

AR1 = 80 98A8h

IR0 = 4h

R0 = 0h

R5 = 07 33C0 0000h = 1.79750e + 02

R7 = 07 0C80 0000h = 1.4050e + 02

R3 = 0h

Data at 80 98C5h = 34C 0000h = 1.2750e + 01

Data at 80 98A4h = 111 0000h = 2.2500e + 00

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR5 = 80 98C6h

AR1 = 80 98A4h

IR0 = 4h

R0 = 04 6718 0000h = 2.88867188e + 01

R5 = 07 33C0 0000h = 1.79750e + 02

R7 = 07 0C80 0000h = 1.4050e + 02

R3 = 08 2020 0000h = 3.20250e + 02

Data at 80 98C5h = 34C 0000h = 1.2750e + 01

Data at 80 98A4h = 111 0000h = 2.2500e + 00

LUF LV UF N Z V C = 0 0 0 0 0 0 0

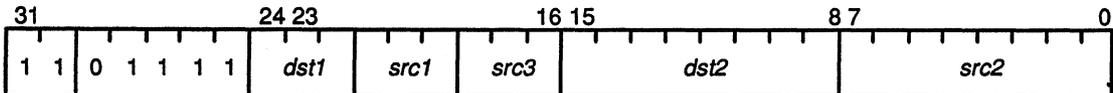
## MPYF3||STF *Parallel MPYF3 and STF*

**Syntax**            **MPYF3** *src2, src1, dst*  
                      || **STF** *src3, dst2*

**Operation**        *src1* × *src2* → *dst1*  
                      || *src3* → *dst2*

**Operands**        *src1* register (R0 – R7)  
                      *src2* indirect (disp = 0, 1, IR0, IR1)  
                      *dst1* register (R0 – R7)  
                      *src3* register (R0 – R7)  
                      *dst2* indirect (disp = 0, 1, IR0, IR1)

### Encoding



**Description** A floating-point multiplication and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYF3) writes to a register and the operation being performed in parallel (STF) reads from the same register, then the STF accepts as input the contents of the register before it is modified by the MPYF3.

If *src2* and *dst2* point to the same location, then *src2* is read before the write to *dst2*.

**Cycles**            1

**Status Bits**      **LUF** 1 if a floating-point underflow occurs, 0 unchanged otherwise.  
                      **LV** 1 if a floating-point overflow occurs, unchanged otherwise.  
                      **UF** 1 if a floating-point underflow occurs, 0 otherwise.  
                      **N** 1 if a negative result is generated, 0 otherwise.  
                      **Z** 1 if a zero result is generated, 0 otherwise.  
                      **V** 1 if a floating-point overflow occurs, 0 otherwise.  
                      **C** Unaffected.

**Mode Bit**        **OVM** Operation is not affected by OVM bit value.

**Example**           MPYF3 \*-AR2(1),R7,R0  
 || STFR3,\*AR0- -(IR0)

**Before Instruction:**

AR2 = 80 982Bh  
 R7 = 05 7B40 0000h = 6.281250e + 01  
 R0 = 0h  
 R3 = 08 6B28 0000h = 4.7031250e + 02  
 AR0 = 80 9860h  
 IR0 = 8h  
 Data at 80 982Ah = 70C8000h = 1.4050e + 02  
 Data at 80 9860h = 0h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR2 = 80 982Bh  
 R7 = 05 7B40 0000h = 6.281250e + 01  
 R0 = 0D 09E4 A000h = 8.82515625e + 03  
 R3 = 08 6B28 0000h = 4.7031250e + 02  
 AR0 = 80 9858h  
 IR0 = 8h  
 Data at 80 982Ah = 70C 8000h = 1.4050e + 02  
 Data at 80 9860h = 86B28 0000h = 4.7031250e + 02  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**            **MPYF3** *srcA, srcB, dst1*  
                       || **SUBF3** *srcC, srcD, dst2*

**Operands**

<table border="0"> <tr><td><i>srcA</i></td><td rowspan="4" style="font-size: 3em; vertical-align: middle;">}</td><td rowspan="4" style="padding-left: 10px;">Any two must be indirect (disp = 0, 1, IR0, IR1), and any two must be register (R0 – R7)</td></tr> <tr><td><i>srcB</i></td></tr> <tr><td><i>srcC</i></td></tr> <tr><td><i>srcD</i></td></tr> </table>	<i>srcA</i>	}	Any two must be indirect (disp = 0, 1, IR0, IR1), and any two must be register (R0 – R7)	<i>srcB</i>	<i>srcC</i>	<i>srcD</i>
<i>srcA</i>	}			Any two must be indirect (disp = 0, 1, IR0, IR1), and any two must be register (R0 – R7)		
<i>srcB</i>						
<i>srcC</i>						
<i>srcD</i>						

**Operation**        *srcA* × *srcB* → *dst1*  
                       || *srcD* – *srcC* → *dst2*

*dst1*            register (*d1*):  
                       0 = R0  
                       1 = R1

*dst2*            register (*d2*):  
                       0 = R2  
                       1 = R3

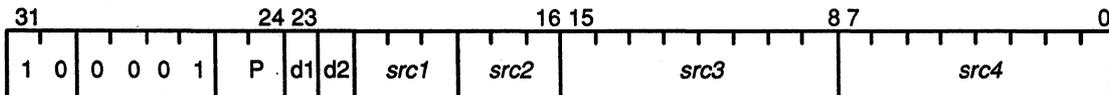
*src1*            register    (R0 – R7)  
*src2*            register    (R0 – R7)  
*src3*            indirect    (disp = 0, 1, IR0, IR1)  
*src4*            indirect    (disp = 0, 1, IR0, IR1)

P                parallel addressing modes (0 ≤ P ≤ 3)

Operation (P Field)

00	<i>src3</i> × <i>src4</i> , <i>src1</i> – <i>src2</i>
01	<i>src3</i> × <i>src1</i> , <i>src4</i> – <i>src2</i>
10	<i>src1</i> × <i>src2</i> , <i>src3</i> – <i>src4</i>
11	<i>src3</i> × <i>src1</i> , <i>src2</i> – <i>src4</i>

**Encoding**



**Description** A floating-point multiplication and a floating-point subtraction are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYF3) reads from a register, and the operation being performed in parallel (SUBF3) writes to the same register, then MPYF3 accepts as input the contents of the register before it is modified by the SUBF3.

Any combination of addressing modes may be coded for the four possible source operands as long as two are coded as indirect and two are register. The assignment of the source operands *srcA* – *srcD* to the *src1* – *src4* fields varies, depending on the combination of addressing modes used; the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations in order to simplify processing.

**Cycles** 1

**Status Bits** **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.

**LV** 1 if a floating-point overflow occurs, unchanged otherwise.

**UF** 1 if a floating-point underflow occurs, 0 otherwise.

**N** 0.

**Z** 0.

**V** 1 if a floating-point overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example**

```
MPYF3 R5, *++AR7(IR1), R0
|| SUBF3R7, *AR3- -(1), R2
or
MPYF3 *++AR7(IR1), R5, R0
|| SUBF3R7, *AR3- -(1), R2
```

**Before Instruction:**

```
R5 = 03 4C00 0000h = 1.2750e + 01
AR7 = 80 9904h
IR1 = 8h
R0 = 0h
R7 = 07 33C0 0000h = 1.79750e + 02
AR3 = 80 98B2h
R2 = 0h
Data at 80 990Ch = 111 0000h = 2.250e + 00
Data at 80 98B2h = 70C 8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0
```

**After Instruction:**

```
R5 = 03 4C00 0000h = 1.2750e + 01
AR7 = 80 990Ch
IR1 = 8h
R0 = 04 6718 0000h = 2.88867188e + 01
R7 = 07 33C0 0000h = 1.79750e + 02
AR3 = 80 98B1h
R2 = 05 E300 0000h = - 3.9250e + 01
Data at 80 990Ch = 111 0000h = 2.250e + 00
Data at 80 98B2h = 70C 8000h = 1.4050e + 02
LUF LV UF N Z V C = 0 0 0 0 0 0 0
```





**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unchanged.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit** **OVM** Operation is affected by OVM bit value.

**Note** Integer overflow occurs when any of the most significant 32 bits of the 64-bit result differs from the most significant bit of the 32-bit output value.

**Syntax**            **MPYI3** *srcA, srcB, dst1*  
                       || **ADDI3** *srcC, srcD, dst2*

**Operation**        *srcA* × *srcB* → *dst1*  
                       || *srcD* + *srcC* → *dst2*

**Operands**

$\left. \begin{array}{l} srcA \\ srcB \\ srcC \\ srcD \end{array} \right\}$  Any two must be indirect (disp = 0, 1, IR0, IR1), and  
 any two must be register (R0 – R7)

*dst1*            register (*d1*):  
                       0 = R0  
                       1 = R1

*dst2*            register (*d2*):  
                       0 = R2  
                       1 = R3

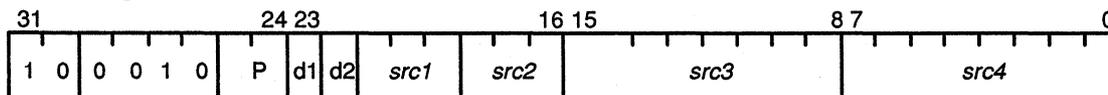
*src1*            register    (R0 – R7)  
*src2*            register    (R0 – R7)  
*src3*            indirect    (disp = 0, 1, IR0, IR1)  
*src4*            indirect    (disp = 0, 1, IR0, IR1)

P                parallel addressing modes (0 ≤ P ≤ 3)

Operation (P Field)

00            *src3* × *src4*, *src1* + *src2*  
 01            *src3* × *src1*, *src4* + *src2*  
 10            *src1* × *src2*, *src3* + *src4*  
 11            *src3* × *src1*, *src2* + *src4*

**Encoding**



**Description** An integer multiplication and an integer addition are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYI3) reads from a register and the operation being performed in parallel (ADDI3) writes to the same register, then MPYI3 accepts as input the contents of the register before it is modified by the ADDI3.

Any combination of addressing modes may be coded for the four possible source operands as long as two are coded as indirect and two are register. The assignment of the source operands *srcA* – *srcD* to the *src1* – *src4* fields varies, depending on the combination of addressing modes used; the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations in order to simplify processing.

**Cycles** 1

**Status Bits**

- LUF** Unchanged.
- LV** 1 if an integer overflow occurs, unchanged otherwise.
- UF** 0.
- N** 0.
- Z** 0.
- V** 1 if an integer overflow occurs, 0 otherwise.
- C** Unaffected.

**Mode Bit** **OV**M Operation is affected by OVM bit value.

**Example**

```

MPYI3 R7,R4,R0
|| ADDI3*-AR3,*AR5--(1),R3
    
```

**Before Instruction:**

```

R7 = 14h = 20
R4 = 64h = 100
R0 = 0h
AR3 = 80 981Fh
AR5 = 80 996Eh
R3 = 0h
Data at 80 981Eh = 0FFFF FFCBh = - 53
Data at 80 996Eh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0
    
```

**After Instruction:**

```

R7 = 14h = 20
R4 = 64h = 100
R0 = 07D0h = 2000
AR3 = 80 981Fh
AR5 = 80 996Dh
R3 = 0h
Data at 80 981Eh = 0FFFF FFCBh = - 53
Data at 80 996Eh = 35h = 53
LUF LV UF N Z V C = 0 0 0 0 0 0 0
    
```



**Example**       MPYI3 \*++AR0(1),R5,R7  
|| STI R2,\*-AR3(1)

**Before Instruction:**

AR0 = 80 995Ah

R5 = 32h = 50

R7 = 0h

R2 = 0DCh = 220

AR3 = 80 982Fh

Data at 80 995Bh = 0C8h = 200

Data at 80 982Eh = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR0 = 80 995Bh

R5 = 32h = 50

R7 = 2710h = 10000

R2 = 0DCh = 220

AR3 = 80 982Fh

Data at 80 995Bh = 0C8h = 200

Data at 80 982Eh = 0DCh = 220

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**            **MPYI3** *srcA, srcB, dst1*  
                       || **SUBI3** *srcC, srcD, dst2*

**Operation**        *srcA* × *srcB* → *dst1*  
                       || *srcD* − *srcC* → *dst2*

**Operands**

<table border="0"> <tr><td style="padding-right: 5px;">}</td><td style="padding-right: 5px;"><i>srcA</i></td></tr> <tr><td style="padding-right: 5px;">}</td><td style="padding-right: 5px;"><i>srcB</i></td></tr> <tr><td style="padding-right: 5px;">}</td><td style="padding-right: 5px;"><i>srcC</i></td></tr> <tr><td style="padding-right: 5px;">}</td><td style="padding-right: 5px;"><i>srcD</i></td></tr> </table>	}	<i>srcA</i>	}	<i>srcB</i>	}	<i>srcC</i>	}	<i>srcD</i>	<p>Any two must be indirect (disp = 0, 1, IR0, IR1), and any two must be register (R0 – R7)</p>
}	<i>srcA</i>								
}	<i>srcB</i>								
}	<i>srcC</i>								
}	<i>srcD</i>								

*dst1*            register (*d1*):  
                       0 = R0  
                       1 = R1

*dst2*            register (*d2*):  
                       0 = R2  
                       1 = R3

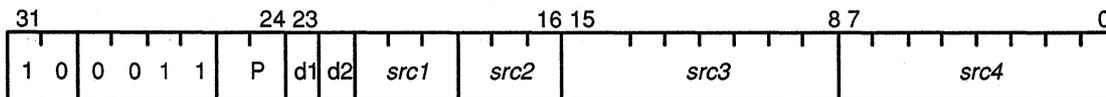
*src1*            register    (R0 – R7)  
*src2*            register    (R0 – R7)  
*src3*            indirect    (disp = 0, 1, IR0, IR1)  
*src4*            indirect    (disp = 0, 1, IR0, IR1)

P                parallel addressing modes (0 ≤ P ≤ 3)

Operation (P Field)

00	<i>src3</i> × <i>src4, src1</i> − <i>src2</i>
01	<i>src3</i> × <i>src1, src4</i> − <i>src2</i>
10	<i>src1</i> × <i>src2, src3</i> − <i>src4</i>
11	<i>src3</i> × <i>src1, src2</i> − <i>src4</i>

**Encoding**



**Description** An integer multiplication and an integer subtraction are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (MPYI3) reads from a register and the operation being performed in parallel (SUBI3) writes to the same register, then MPYI3 accepts as input the contents of the register before it is modified by the SUBI3.

Any combination of addressing modes may be coded for the four possible source operands as long as two are coded as indirect and two are register. The assignment of the source operands *srcA* – *srcD* to the *src1* – *src4* fields varies, depending on the combination of addressing modes used; the P field is encoded accordingly. The assembler may, when not significant, change the order of operands in commutative operations in order to simplify processing.

Integer overflow occurs when any of the most significant 16 bits of the 48-bit result differs from the most significant bit of the 32-bit output value.

**Cycles** 1

**Status Bits** **LUF** Unchanged.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 1 if an integer underflow occurs, 0 otherwise.

**N** 0.

**Z** 0.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit** **OVM** Operation is affected by OVM bit value.

**Example**        MPYI3 R2, \*++AR0(1), R0  
                  || SUBI3AR5- - (IR1), R4, R2  
                  or  
                  MPYI3 \*++AR0(1), R2, R0  
                  || SUBI3AR5- - (IR1), R4, R2

**Before Instruction:**

R2 = 32h = 50  
AR0 = 80 98E3h  
R0 = 0h  
AR5 = 80 99FCh  
IR1 = 0Ch  
R4 = 07D0h = 2000  
Data at 80 98E4h = 62h = 98  
Data at 80 99FCh = 4B0h = 1200  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

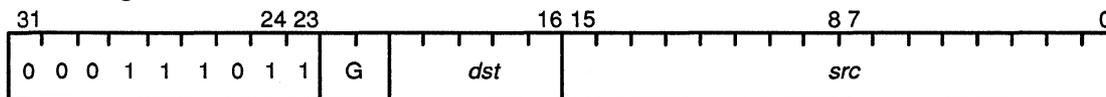
R2 = 320h = 800  
AR0 = 80 98E4h  
R0 = 01324h = 4900  
AR5 = 80 99F0h  
IR1 = 0Ch  
R4 = 07D0h = 2000  
Data at 80 98E4h = 62h = 98  
Data at 80 99FCh = 4B0h = 1200  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax**     **MPYSHI** *src, dst*

**Operation**    *dst* x *src* → *dst*

**Operands**    *src*    general addressing modes  
                   *dst*    register mode (any register in CPU primary register file)

**Encoding**



**Instruction Word Fields**

<b>G</b>	<b>src addressing modes</b>
00	register mode (any CPU register)
01	direct mode
10	indirect mode
11	immediate mode

**Description** The 32 MSBs of the product of the numbers at *dst* and *src* are loaded into the *dst* register. These numbers, when read, are assumed to be signed 32-bit integers. The result is assumed to be a signed 64-bit integer. The output to the *dst* register is the 32 most significant bits of the result.

**Cycles**        1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

- LUF** Unchanged.
- LV** Unchanged.
- UF** 0.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if all 64 bits of the product are 0, 0 otherwise.
- V** 0.
- C** Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

## MPYSHI3 *Multiply Signed Integer Producing 32 MSBs, 3 Operands*

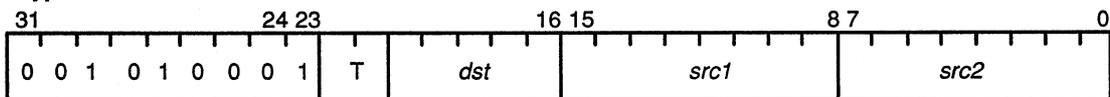
**Syntax**     **MPYSHI3** *src2, src1, dst*

**Operation**   *src1* × *src2* → *dst*

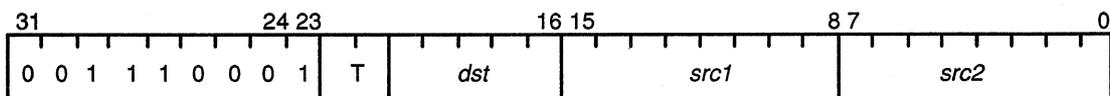
**Operands**   *src1*   type 1 or type 2 three-operand addressing modes  
               *src2*   type 1 or type 2 three-operand addressing modes  
               *dst*     register mode (any register in CPU primary register file)

### Encoding

#### Type 1



#### Type 2



### Instruction Word Fields

	T	src1 addressing modes	src2 addressing modes
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	src1 addressing modes	src2 addressing modes
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description**   The product of the numbers at the *src1* and *src2* operands is loaded into the *dst* register. The numbers at the *src1* and *src2* operands are assumed to be 32-bit signed integers. The result is assumed to be a signed 64-bit integer. The output to the *dst* register is the 32 most significant bits of the result.

**Cycles** 1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unchanged.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

## MPYUHI *Multiply Unsigned Integer and Produce 32 MSBs*

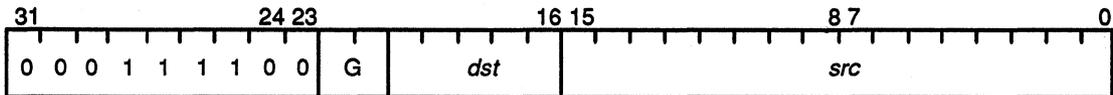
---

**Syntax**      **MPYUHI** *src, dst*

**Operation**    *dst* x *src* → *dst*

**Operands**    *src*    general addressing modes  
                   *dst*    register mode (any register in CPU primary register file)

**Encoding**



**Instruction Word Fields**

G	src addressing modes
00	register mode (any CPU register)
01	direct mode
10	indirect mode
11	immediate mode

**Description** The 32 MSBs of the product of the numbers at *dst* and *src* operands are loaded into the *dst* register. These numbers, when read, are assumed to be unsigned 32-bit integers. The result is assumed to be an unsigned 64-bit integer. The output to the *dst* register is the 32 most significant bits of the result.

**Cycles**      1

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unchanged.

**LV** Unchanged.

**UF** 0.

**N** 0.

**Z** 1 if all 64 bits of the product are 0, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit**    **OVM** Operation is not affected by OVM bit value.

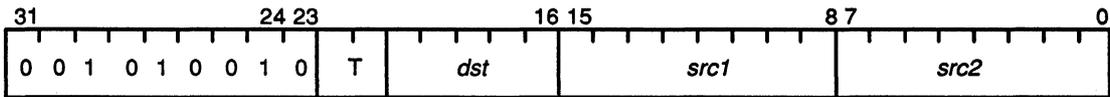
**Syntax** **MPYUHI3** *src2, src1, dst*

**Operation**  $src1 \times src2 \rightarrow dst$

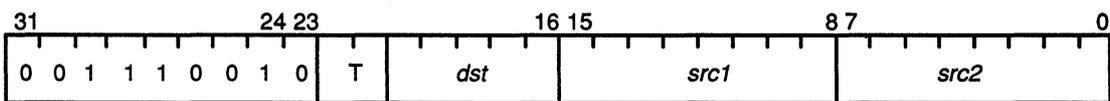
**Operands** *src1, src2* both type 1 or type 2 three-operand addressing modes  
*dst* register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	T	src1 addressing modes	src2 addressing modes
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	src1 addressing modes	src2 addressing modes
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description** The product of the numbers at the *src1* and *src2* operands is loaded into the *dst* register. The numbers at the *src1* and *src2* operands are assumed to be 32-bit signed integers. The result is assumed to be an unsigned 64-bit integer. The output to the *dst* register is the 32 most significant bits of the result.

**Cycles** 1

## **MPYUH13** *Multiply Unsigned Integer Producing 32 MSBs, 3 Operands*

---

**Status Bits** If ST (SET COND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SET COND) = 1, they are modified for all destination registers.

**LUF** Unchanged.

**LV** Unchanged.

**UF** 0.

**N** 0.

**Z** 1 if all 64 bits of the product are 0, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

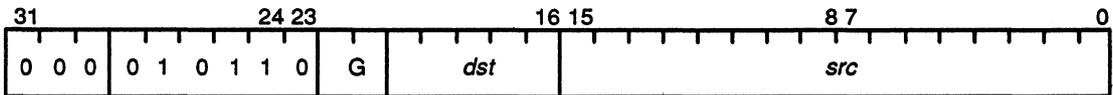
**Syntax**      **NEGB** *src*, *dst*

**Operation**     $0 - src - C \rightarrow dst$

**Operands**    *src* general addressing modes (G):  
                   0 0    register (any register in CPU primary register file)  
                   0 1    direct  
                   1 0    indirect  
                   1 1    immediate

*dst* register (any register in CPU primary register file)

**Encoding**



**Description**    The difference of the 0, *src*, and C operands, calculated as shown, is loaded into the *dst* register. The *dst* and *src* are assumed to be signed integers.

**Cycles**        1

**Status Bits**    If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

- LUF** Unaffected.
- LV** 1 if an integer overflow occurs, unchanged otherwise.
- UF** 0.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if an integer overflow occurs, 0 otherwise.
- C** 1 if a borrow occurs, 0 otherwise.

**Mode Bit**      **OVM** Operation is affected by OVM bit value.

**Example**        **NEGB** R5, R7

Before Instruction:

R5 = 0FFFF FFCBh = - 53  
 R7 = 0h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 1

After Instruction:

R5 = 0FFFF FFCBh = - 53  
 R7 = 34h = 52  
 LUF LV UF N Z V C = 0 0 0 0 0 0 1

## NEGF *Negate Floating-Point Value*

**Syntax**     **NEGF** *src, dst*

**Operation**   0 – *src* → *dst*

**Operands**   *src* general addressing modes (G):

    0 0   register (R0 – R11)

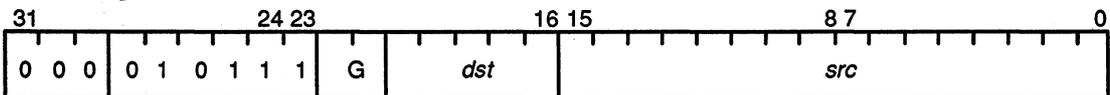
    0 1   direct

    1 0   indirect

    1 1   immediate

*dst* register (R0 – R11)

### Encoding



**Description** The difference of the 0 and *src* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

**Cycles**     1

**Status Bits** **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.

**LV** 1 if a floating-point overflow occurs, unchanged otherwise.

**UF** 1 if a floating-point underflow occurs, 0 otherwise.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if a floating-point overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

**Example**    NEGF \*++AR3(2), R1

### Before Instruction:

AR3 = 80 9800h

R1 = 05 7B40 0025h = 6.28125006e + 01

Data at 80 9802h = 70C 8000h = 1.4050e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

AR3 = 80 9802h

R1 = 07 F380 0000h = -1.4050e + 02

Data at 80 9802h = 70C 8000h = 1.4050e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0



**Example**        NEGF\*AR4- -(1),R7  
                  || STF R2,\*++AR5(1)

**Before Instruction:**

AR4 = 80 98E1h

R7 = 0h

R2 = 07 33C0 0000h = 1.79750e + 02

AR5 = 80 9803h

Data at 80 98E1h = 57 B40 0000h = 6.281250e + 01

Data at 80 9804h = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR4 = 80 98E0h

R7 = 05 84C0 0000h = - 6.281250e + 01

R2 = 07 33C0 0000h = 1.79750e + 02

AR5 = 80 9804h

Data at 80 98E1h = 57B 4000h = 6.281250e + 01

Data at 80 9804h = 733 C000h = 1.79750e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0

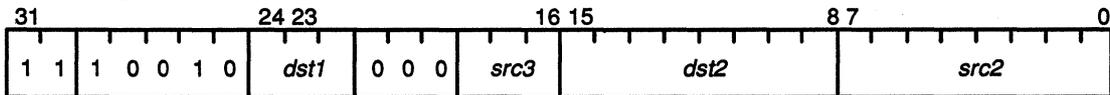


**Syntax**        **NEGI** *src2, dst1*  
                   || **STI** *src3, dst2*

**Operation**    0 – *src2* → *dst1*  
                   || *src3* → *dst2*

**Operands**    *src2* indirect (disp = 0, 1, IR0, IR1)  
                   *dst1* register (R0 – R7)  
                   *src3* register (R0 – R7)  
                   *dst2* indirect (disp = 0, 1, IR0, IR1)

**Encoding**



**Description** An integer negation and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (NEGI) writes to the same register, then STI accepts as input the contents of the register before it is modified by the NEGI.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

**Cycles**        1

**Status Bits** **LUF** Unaffected.  
**LV** 1 if an integer overflow occurs, unchanged otherwise.  
**UF** 0.  
**N** 1 if a negative result is generated, 0 otherwise.  
**Z** 1 if a zero result is generated, 0 otherwise.  
**V** 1 if an integer overflow occurs, 0 otherwise.  
**C** 1 if a borrow occurs, 0 otherwise.

**Mode Bit**    **OV**M Operation is affected by OVM bit value.

**Example**      NEGI \*-AR3, R2  
|| STI R2, \*AR1++

**Before Instruction:**

AR3 = 80 982Fh

R2 = 19h = 25

AR1 = 80 98A5h

Data at 80 982Eh = 0DCh = 220

Data at 80 98A5h = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR3 = 80 982Fh

R2 = 0FFFF FF24h = -220

AR1 = 80 98A6h

Data at 80 982Eh = 0DCh = 220

Data at 80 98A5h = 19h = 25

LUF LV UF N Z V C = 0 0 0 1 0 0 1

## NOP No Operation

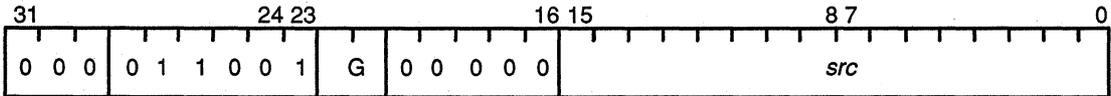
---

**Syntax**     **NOP** *src*

**Operation**   No ALU or multiplier operations.  
              ARn is modified if *src* is specified in indirect mode.

**Operands**   *src* general addressing modes (G):  
              0 0   register (no operation)  
              1 0   indirect (modify ARn,  $0 \leq n \leq 7$ )

### Encoding



**Description** If the *src* operand is specified in the indirect mode, the specified addressing operation is performed and a dummy memory read occurs. If the *src* operand is omitted, no operation is performed.

**Cycles**       1

**Status Bits** **LUF** Unaffected.  
**LV** Unaffected.  
**UF** Unaffected.  
**N** Unaffected.  
**Z** Unaffected.  
**V** Unaffected.  
**C** Unaffected.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

**Example**     NOP

**Before Instruction:**

PC = 3Ah

**After Instruction:**

PC = 3Bh

**Example**     NOP \*AR3- - (1)

**Before Instruction:**

PC = 5h

AR3 = 80 9900h

**After Instruction:**

PC = 6h

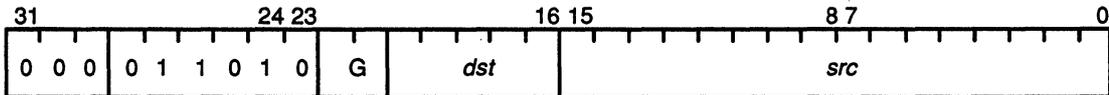
AR3 = 80 98FFh

**Syntax**      **NORM** *src, dst*

**Operation**    *norm (src) → dst*

**Operands**    *src* general addressing modes (G):  
                   0 0    register (R0 – R11)  
                   0 1    direct  
                   1 0    indirect  
                   1 1    immediate

**Encoding**



**Description** The *src* operand is assumed to be an unnormalized floating-point number; i.e., the implied bit is set equal to the sign bit. The *dst* is set equal to the normalized *src* operand with the implied bit removed. The *dst* operand exponent is set to the *src* operand exponent minus the size of the left shift necessary to normalize the *src*. The *dst* operand is assumed to be a normalized floating-point number.

For values of *src*:

- If *src* (exp) = -128 and *src* (man) = 0, then *dst* = 0, Z = 1, and UF = 0.
- If *src* (exp) = -128 and *src* (man) ≠ 0, then *dst* = 0, Z = 0, and UF = 1.
- For all other cases of the *src*, if a floating-point underflow occurs, then *dst* (man) is forced to 0 and *dst* (exp) = -128. If *src* (man) = 0, then *dst* (man) = 0 and *dst* (exp) = -128. Refer to Section 4.7 on page 4-24.

**Cycles**            1

**Status Bits**    **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.  
                   **LV** Unaffected.  
                   **UF** 1 if a floating-point underflow occurs, 0 otherwise.  
                   **N** 1 if a negative result is generated, 0 otherwise.  
                   **Z** 1 if a zero result is generated, 0 otherwise.  
                   **V** 0.  
                   **C** Unaffected.

**Mode Bit**        **OVM** Operation is not affected by OVM bit value.

## **NORM** *Normalize*

---

**Example**    `NORM R1, R2`

**Before Instruction:**

R1 = 04 0000 3AF5h

R2 = 07 0C80 0000h

LUF LV UF N Z V C = 0 0 0 0 0 0 0 0

**After Instruction:**

R1 = 04 0000 3AF5h

R2 = F2 6BD4 0000h = 1.12451613e - 04

LUF LV UF N Z V C = 0 0 0 0 0 0 0 0





**Example**

```

NOT    *+AR2, R3
||     STI    R7, *- -AR4 (IR1)

```

**Before Instruction:**

AR2 = 80 99CBh

R3 = 0h

R7 = 0DCh = 220

AR4 = 80 9850h

IR1 = 10h

Data at 80 99CCh = 0C2Fh

Data at 80 9840h = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR2 = 80 99CBh

R3 = 0FFFF F3D0h

R7 = 0DCh = 220

AR4 = 80 9840h

IR1 = 10h

Data at 80 99CCh = 0C2Fh

Data at 80 9840h = 0DCh = 220

LUF LV UF N Z V C = 0 0 0 1 0 0 0

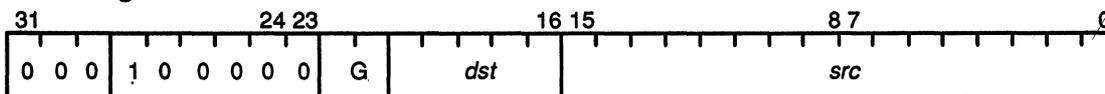
## OR Bitwise Logical OR

**Syntax** OR *src*, *dst*

**Operation** *dst* OR *src* → *dst*

**Operands** *src* general addressing modes (G):  
0 0 register (any register in CPU primary register file)  
0 1 direct  
1 0 indirect  
1 1 immediate (not sign-extended)  
*dst* register (any register in CPU primary register file)

### Encoding



**Description** The bitwise logical OR between the *src* and *dst* operands is loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

**Cycles** 1

**Status Bits** If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Example** OR \*++AR1 (IR1), R2

### Before Instruction:

AR1 = 80 9800h

IR1 = 4h

R2 = 01256 0000h

Data at 80 9804h = 2BCDh

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

AR1 = 80 9804h

IR1 = 4h

R2 = 01256 2BCDh

Data at 80 9804h = 2BCDh

LUF LV UF N Z V C = 0 0 0 0 0 0 0



## **OR3** *Bitwise Logical OR, 3 Operands*

---

**Description** The bitwise logical OR between the numbers at the *src1* and *src2* operands is loaded into the *dst* register. The numbers at the *src1*, *src2*, and *dst* operands are assumed to be unsigned integers.

**Cycles** 1

**Status Bits** If ST (SETCOND) = 0, the condition flags are modified if the destination register is R0—R11. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

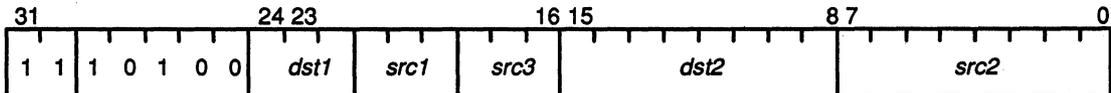
**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Syntax**        **OR3** *src2, src1, dst1*  
 || **STI** *src3, dst2*

**Operation**    *src1* OR *src2* → *dst1*  
 ||    *src3* → *dst2*

**Operands**    *src1* register (R0 – R7)  
                   *src2* indirect (disp = 0, 1, IR0, IR1)  
                   *dst1* register (R0 – R7)  
                   *src3* register (R0 – R7)  
                   *dst2* indirect (disp = 0, 1, IR0, IR1)

**Encoding**

A bitwise logical OR and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (OR3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the OR3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

**Cycles**        1

**Status Bits**    **LUF** Unaffected.  
                   **LV**    Unaffected.  
                   **UF**    0.  
                   **N**    MSB of the output.  
                   **Z**    1 if a zero result is generated, 0 otherwise.  
                   **V**    0.  
                   **C**    Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

**Example**       OR3 \*++AR2, R5, R2  
          || STIR6, \*AR1- -

**Before Instruction:**

AR2 = 80 9830h

R5 = 80 0000h

R2 = 0h

R6 = 0DCh = 220

AR1 = 80 9883h

Data at 80 9831h = 9800h

Data at 80 9883h = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR2 = 80 9831h

R5 = 80 0000h

R2 = 80 9800h

R6 = 0DCh = 220

AR1 = 80 9882h

Data at 80 9831h = 9800h

Data at 80 9883h = 0DCh = 220

LUF LV UF N Z V C = 0 0 0 0 0 0 0



## POPF POP Floating-Point Value

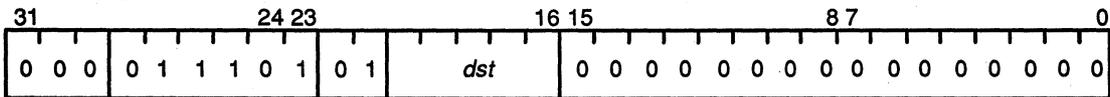
---

**Syntax** POPF *dst*

**Operation** \*SP-- → *dst1*

**Operands** *dst* register (R0 – R11)

**Encoding**



**Description** The top of the current system stack is popped and loaded into the *dst* register. The top of the stack is assumed to be a floating-point number. The POP is performed with a post decrement of the stack pointer.

**Cycles** 1

**Status Bits** LUF Unaffected.  
UF 0.  
LV Unaffected.  
N 1 if a negative result is generated, 0 otherwise.  
Z 1 if a zero result is generated, 0 otherwise.  
V 0.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** POPF R4

**Before Instruction:**

SP = 80 984Ah  
R4 = 02 5D2E 0123h = 6.91186578e + 00  
Data at 80 984Ah = 5F2C 1302h = 5.32544007e + 28  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

SP = 80 9849h  
R4 = 5F 2C13 0200h = 5.32544007e + 28  
Data at 80 984Ah = 5F2C 1302h = 5.32544007e + 28  
LUF LV UF N Z V C = 0 0 0 0 0 0 0



## **PUSHF** *PUSH Floating-Point Value*

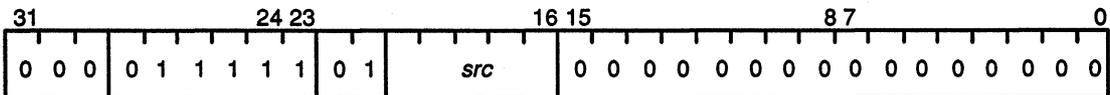
---

**Syntax**      **PUSHF** *src*

**Operation**    *src* → \*++SP

**Operands**    *src* register (R0 – R11)

**Encoding**



**Description** The contents of the *src* register are pushed onto the current system stack. The *src* is assumed to be a floating-point number. The PUSH is performed with a preincrement of the stack pointer.

**Cycles**      1

**Status Bits**    **LUF** Unaffected.  
                  **LV** Unaffected.  
                  **UF** Unaffected.  
                  **N** Unaffected.  
                  **Z** Unaffected.  
                  **V** Unaffected.  
                  **C** Unaffected.

**Mode Bit**      **OVM** Operation is not affected by OVM bit value.

**Example**      PUSHF R2

**Before Instruction:**

SP = 80 9801h  
R2 = 02 5C12 8081h = 6.87725854e + 00  
Data at 80 9802h = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

SP = 80 9802h  
R2 = 02 5C1 28081h = 6.87725854e + 00  
Data at 80 9802h = 025C 1280h = 6.87725830e + 00  
LUF LV UF N Z V C = 0 0 0 0 0 0 0



## RETIcond *Return From Interrupt or Trap Conditionally*

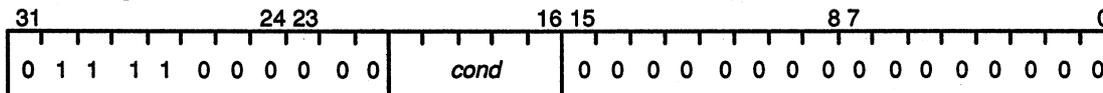
---

**Syntax**     **RETIcond**

**Operation**   If (*cond* is true)  
                  \*(SP) → PC  
                  ST(PGIE) → ST(GIE)  
                  ST(PCF) → ST(CF)  
                  Else, continue

**Operands**   None

**Encoding**



**Description** If the condition is true, then the top of the stack is popped to the PC, PGIE is copied to GIE, and PCF is copied to CF. If the condition is not true, then continue normal operation (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags).

**Cycles**       4

**Status Bits**   **LUF**   Unaffected.  
                  **LV**     Unaffected.  
                  **UF**     Unaffected.  
                  **N**      Unaffected.  
                  **Z**      Unaffected.  
                  **V**      Unaffected.  
                  **C**      Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.



## RETScond *Return from Subroutine Conditionally*

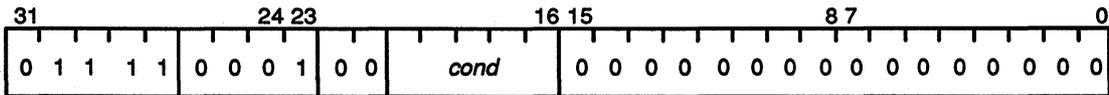
---

**Syntax**      **RETScond**

**Operation**    If *cond* is true:  
                  \*SP-- → PC.  
                  Else, continue.

**Operands**    None

**Encoding**



**Description**    A conditional return is performed. If the condition is true, the top of the stack is popped to the PC.

The TMS320C40 provides 20 condition codes that can be used with this instruction (see Section 11.2 on page 11-10 for a list of condition mnemonics, encoding, and flags).

**Cycles**        4

**Status Bits**    **LUF**    Unaffected.  
                  **LV**     Unaffected.  
                  **UF**     Unaffected.  
                  **N**      Unaffected.  
                  **Z**      Unaffected.  
                  **V**      Unaffected.  
                  **C**      Unaffected.

**Mode Bit**      **OVM** Operation is not affected by OVM bit value.

**Example**        RETSGE

**Before Instruction:**

PC = 123h  
SP = 80 983Ch  
Data at 80 983Ch = 456h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

PC = 456h  
SP = 80 983Bh  
Data at 80 983Ch = 456h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax** **RND** *src, dst*

**Operation**  $\text{rnd}(\text{src}) \rightarrow \text{dst}$

**Operands** *src* general addressing modes (G):

0 0 register (R0 – R11)

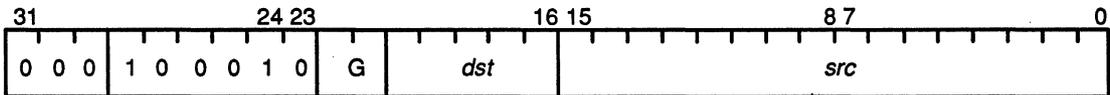
0 1 direct

1 0 indirect

1 1 immediate

*dst* register (R0 – R11)

**Encoding**



**Description** The result of rounding the *src* operand is loaded into the *dst* register. The *src* operand is rounded to the nearest single-precision floating-point value. If the *src* operand is exactly halfway between two single-precision values, it is rounded to the most positive of those values.

**Cycles** 1

**Status Bits**

- LUF** 1 if a floating-point underflow occurs, unchanged otherwise.
- LV** 1 if a floating-point overflow occurs, unchanged otherwise.
- UF** 1 if a floating-point underflow occurs, 0 otherwise.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if a floating-point overflow occurs, 0 otherwise.
- C** Unaffected.

**Mode Bit** **OVM** Operation is affected by OVM bit value.

**Example** RND R5, R2

**Before Instruction:**

R5 = 07 33C1 6EEFh = 1.79755599e + 02

R2 = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0 0

**After Instruction:**

R5 = 07 33C1 6EEFh = 1.79755599e + 02

R2 = 07 33C1 6F00h = 1.79755600e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0 0

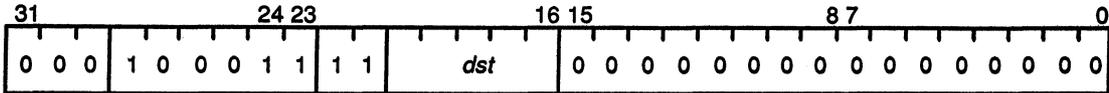
## ROL *Rotate Left*

**Syntax**     **ROL** *dst*

**Operation**   *dst* left-rotated 1 bit → *dst*

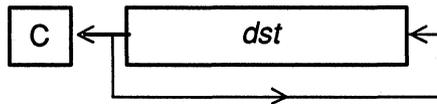
**Operands**   *dst* register (any register in CPU primary register file)

**Encoding**



**Description** The contents of the *dst* operand are left-rotated one bit and loaded into the *dst* register. This is a circular rotate with the MSB transferred into the LSB.

Rotate left:



**Cycles**     1

**Status Bits** If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero output is generated, 0 otherwise.

**V** 0.

**C** Set to the value of the bit rotated out of the high-order bit. Unaffected if *dst* is not R0 – R7.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

**Example**     ROL R3

**Before Instruction:**

R3 = 8002 5CD4h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R3 = 0004 B9A9h

LUF LV UF N Z V C = 0 0 0 0 0 0 1



## **ROLC** *Rotate Left Through Carry*

---

**Example** ROLC R3

**Before Instruction:**

R3 = 0000 0420h

LUF LV UF N Z V C = 0 0 0 0 0 0 1

**After Instruction:**

R3 = 00000 0841h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Example** ROLC R3

**Before Instruction:**

R3 = 8000 4281h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R3 = 0000 8502h

LUF LV UF N Z V C = 0 0 0 0 0 0 1





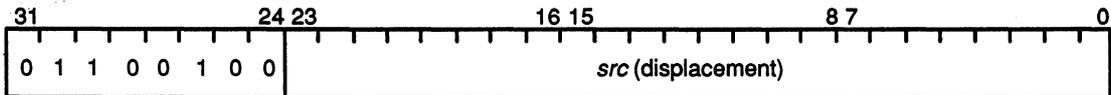
**Syntax** RPTB *src*

**Operation**  $src + PC + 1 \rightarrow RE$   
 $1 \rightarrow ST (RM)$   
 Next PC  $\rightarrow RS$

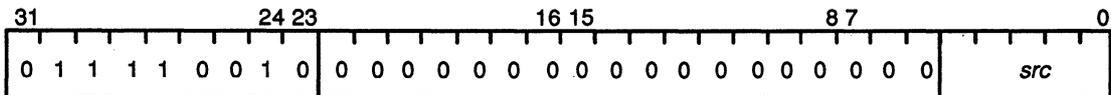
**Operands** *src* 24-bit signed immediate displacement or register mode

**Encoding**

For 24-bit signed immediate or register mode:



For register mode:



**Description** RPTB allows a block of instructions to be repeated a number of times without any penalty for looping.

It activates the block repeat mode of updating the PC. The *src* operand may be a 32-bit register value or a 24-bit signed immediate value (displacement). The resulting *src* address is the end address of the block to be repeated. This address is loaded into the repeat end address (RE) register. A 1 is written into the repeat mode bit of status register (ST(RM)) to indicate that the PC is to be updated in the repeat mode. The address of the next instruction is loaded into the repeat start address (RS) register.

**Cycles** 4

**Status Bits** LUF Unaffected.  
 LV Unaffected.  
 UF Unaffected.  
 N Unaffected.  
 Z Unaffected.  
 V Unaffected.  
 C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

## RPTBD Repeat Block Delayed

---

**Syntax** RPTBD *src*

**Operation** if *src* is an immediate value (displacement)

$src + PC + 3 \rightarrow RE$

Else:

$src \rightarrow RE$

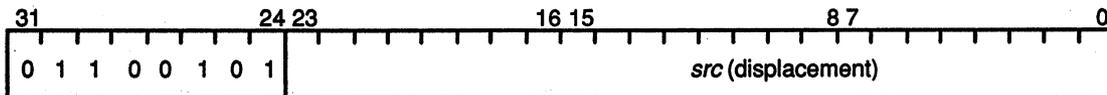
1  $\rightarrow$  ST(RM)

PC of RPTBD + 4  $\rightarrow$  RS

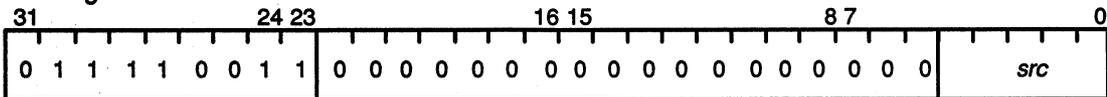
**Operands** *src* 24-bit signed immediate displacement or register mode

### Encoding

For 24-bit signed immediate or register mode:



For register mode:



**Description** RPTBD allows a block of instructions to be repeated a number of times without any penalty for looping and with single-cycle execution of the RPTBD instruction.

It activates the block repeat mode of updating the PC. The *src* operand may be a 32-bit register value or a 24-bit signed immediate value (displacement). The resulting *src* address is loaded into the repeat end address (RE) register (block end address). A 1 is written to the status-register repeat mode bit (ST(RM)), indicating the PC is to be updated in the repeat mode. The address of the next instruction +3 is loaded into the repeat start address (RS) register.

RPTBD does not flush the pipeline. The three instructions following RPTBD are executed and may not be an instruction that modifies the program flow. These three instructions are not part of the block that is repeated.

**Cycles** 1

**Status Bits**

LUF	Unaffected.
LV	Unaffected.
UF	Unaffected.
N	Unaffected.
Z	Unaffected.
V	Unaffected.
C	Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.



## RPTS *Repeat Single*

---

**Example** RPTS AR5

**Before Instruction:**

PC = 123h

ST = 0h

RS = 0h

RE = 0h

RC = 0h

AR5 = 0FFh

LUF LV UF N Z V C=0 0 0 0 0 0 0

**After Instruction:**

PC = 124h

ST = 100h

RS = 124h

RE = 124h

RC = 0FFh

AR5 = 0FFh

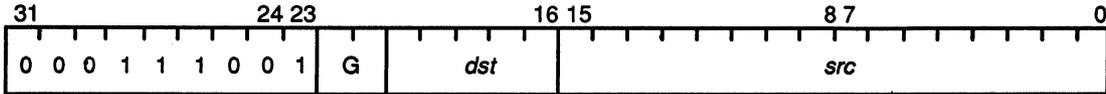
LUF LV UF N Z V C=0 0 0 0 0 0 0

**Syntax**      **RSQRF** *src, dst*

**Operation**    16-bit reciprocal of the square root of *src* → *dst*

**Operands**    *src*    extended-precision register, direct and indirect addressing modes  
                   *dst*    extended-precision register

**Encoding**



**Instruction Word Fields**

G	<i>src</i> addressing modes
00	extended-precision register (R0 – R11)
01	direct mode
10	indirect mode

**Description** The 16-bit approximation of the reciprocal of the square root of the number at the *src* operand is loaded into the *dst* register. The number at the *src* operand is assumed to be positive. The operation for negative inputs is undefined.

The value at the *dst* and *src* operands are assumed to be floating-point numbers.

**Cycles**            1

**Status Bits**    **LUF** Unchanged.  
                       **LV** 1 if input is zero unchanged otherwise.  
                       **UF** 0.  
                       **N** 0.  
                       **Z** 0  
                       **V** 1 if input is zero, 0 otherwise.  
                       **C** Unaffected.

**Mode Bit**        **OVM** Operation is not affected by OVM bit value.

## SIGI *Signal, Interlocked*

**Syntax**     **SIGI** *src, dst*

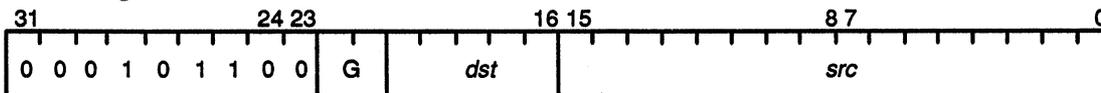
**Operation**    $\overline{\text{LOCK}}$  (or  $\overline{\text{LLOCK}}$ ) pin brought low.

*src* → *dst*

$\overline{\text{LOCK}}$  (or  $\overline{\text{LLOCK}}$ ) pin brought high.

**Operands**   *src*   direct and indirect addressing modes (assumed to be signed integer)  
              *dst*    register mode (assumed to be signed integer)

### Encoding



### Instruction Word Fields

G	<i>src</i> addressing modes
01	direct mode
10	indirect mode

**Description** An interlocking operation is signaled using the appropriate bus-lock signal ( $\overline{\text{LOCK}}$  or  $\overline{\text{LLOCK}}$ ) if and only if an external memory access is performed. The *src* and *dst* operands are assumed to be signed integers. After the read is performed, the bus-lock signal is deasserted. If an internal memory access is performed, SIGI will perform the read but will not assert a bus-lock signal.

The numbers at the *src* and *dst* operands are treated as signed integers.

**Cycles**     1

**Status Bits** If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**11 Mode Bit**   **OVM** Operation is not affected by OVM bit value.



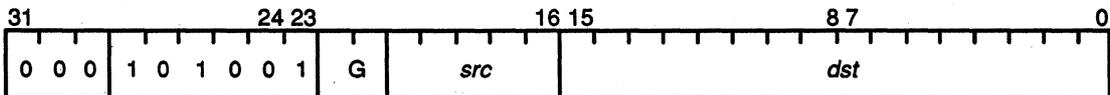
## STFI Store Floating-Point Value, Interlocked

**Syntax** STFI *src, dst*

**Operation** *src* → *dst*  
Signal end of interlocked operation.

**Operands** *src* register (R0 – R11)  
*dst* general addressing modes (G):  
0 1 direct  
1 0 indirect

### Encoding



**Description** The *src* register is loaded into the *dst* memory location. An interlocked operation is signaled over  $\overline{\text{LOCK}}$  or  $\overline{\text{LLOCK}}$ . The *src* and *dst* operands are assumed to be floating-point numbers. Refer to Section 7.7 on page 7-39 for detailed information.

**Cycles** 1

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** STFI R3, \*-AR4

### Before Instruction:

R3 = 07 33C0 0000h = 1.79750e + 02

AR4 = 80 993Ch

Data at 80 993Bh = 0h

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

R3 = 07 33C0 0000h = 1.79750e + 02

AR4 = 80 993Ch

Data at 80 993Bh = 733 C000h = 1.79750e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0



**Example**

```
STFR4, *AR3- -  
|| STF R3, *++AR5
```

**Before Instruction:**

```
R4 = 07 0C80 0000h = 1.4050e + 02  
AR3 = 80 9835h  
R3 = 07 33C0 0000h = 1.79750e + 02  
AR5 = 80 99D2h  
Data at 80 9835h = 0h  
Data at 80 99D3h = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0
```

**After Instruction:**

```
R4 = 07 0C80 0000h = 1.4050e + 02  
AR3 = 80 9834h  
R3 = 07 33C0 0000h = 1.79750e + 02  
AR5 = 80 99D3h  
Data at 80 9835h = 070C 8000h = 1.4050e + 02  
Data at 80 99D3h = 0733 C000h = 1.79750e + 02  
LUF LV UF N Z V C = 0 0 0 0 0 0 0
```

**Syntax** STI *src, dst*

**Operation** *src* → *dst*

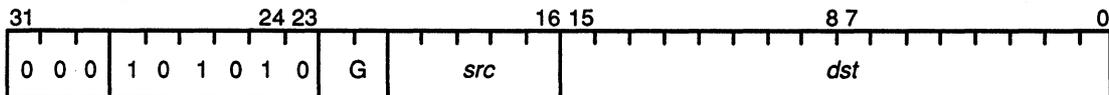
**Operands** *src* register (any register in CPU primary register file)

*dst* general addressing modes (G):

0 1 direct

1 0 indirect

**Encoding**



**Description** The *src* register is loaded into the *dst* memory location. The *src* and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits**

- LUF Unaffected.
- LV Unaffected.
- UF Unaffected.
- N Unaffected.
- Z Unaffected.
- V Unaffected.
- C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.

**Example** STI R4, @982Bh

**Before Instruction:**

DP = 80h

R4 = 42BD7h = 273,367

Data at 80 982Bh = 0E5FCh = 58,876

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

DP = 80h

R4 = 42 BD7h = 273,367

Data at 80 982Bh = 42BD7h = 273,367

LUF LV UF N Z V C = 0 0 0 0 0 0 0

## STII *Store Integer, Interlocked*

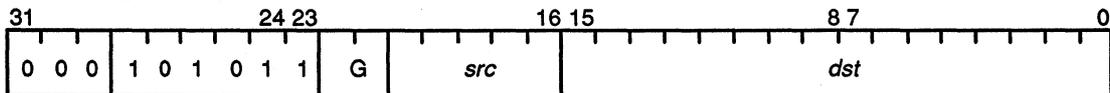
---

**Syntax**     **STII** *src, dst*

**Operation**   *src* → *dst*  
Signal end of interlocked operation.

**Operands**   *src* register (any register in CPU primary register file)  
*dst* general addressing modes (G):  
          0 1   direct  
          1 0   indirect

### Encoding



**Description** The *src* register is loaded into the *dst* memory location. An interlocked operation is signaled over  $\overline{\text{LOCK}}$  or  $\overline{\text{LLOCK}}$ . The *src* and *dst* operands are assumed to be signed integers. Refer to Section 7.7 on page 7-39 for detailed information.

**Cycles**     1

**Status Bits** **LUF** Unaffected.  
**LV** Unaffected.  
**UF** Unaffected.  
**N** Unaffected.  
**Z** Unaffected.  
**V** Unaffected.  
**C** Unaffected.

**Mode Bit**   **OVM** Operation is not affected by OVM bit value.

**Example**     STII R1, @98AEh

#### **Before Instruction:**

DP = 80h  
R1 = 78Dh  
Data at 80 98AEh = 25Ch

#### **After Instruction:**

DP = 80h  
R1 = 78Dh  
Data at 80 98AEh = 7BDh



**Example**            STI R0, \*++AR2 (IR0)  
                      || STI R5, \*AR0

**Before Instruction:**

R0 = 0DCh = 220  
AR2 = 80 9830h  
IR0 = 8h  
R5 = 35h = 53  
AR0 = 80 98D3h  
Data at 80 9838h = 0h  
Data at 80 98D3h = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

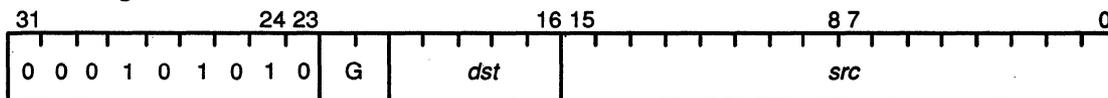
R0 = 0DCh = 220  
AR2 = 80 9838h  
IR0 = 8h  
R5 = 35h = 53  
AR0 = 80 98D3h  
Data at 80 9838h = 0DCh = 220  
Data at 80 98D3h = 35h = 53  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Syntax** STIK *src, dst*

**Operation** *src* → *dst*

**Operands** *src* 5-bit signed integer  
*dst* direct and indirect mode

**Encoding**



**Instruction Word Fields**

G	<i>dst</i> addressing modes
00	direct mode
11	indirect mode

**Description** The 5-bit signed integer *src* value is loaded into the *dst* memory location. The *src* and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.



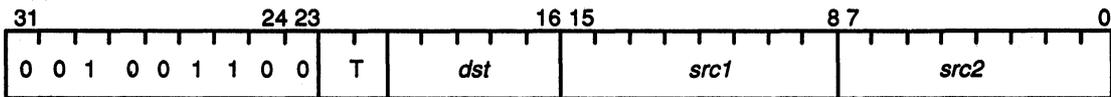
**Syntax**      **SUBB3** *src2, src1, dst*

**Operation**    *src1* – *src2* – C → *dst*

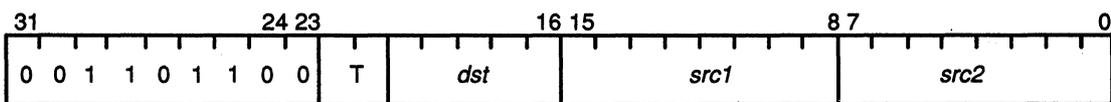
**Operands**    *src1, src2*    both type 1 or type 2 three-operand addressing modes  
                   *dst*                    register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode <sup>+</sup> +ARn(5-bit unsigned displacement)
	10	indirect mode <sup>+</sup> +ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode <sup>+</sup> +ARn1(5-bit unsigned displacement)	indirect mode <sup>+</sup> +ARn2(5-bit unsigned displacement)

## **SUBB3** *Subtract Integer With Borrow, 3 Operands*

---

**Description** The difference of the *src1* and *src2* operands and the C (carry) flag is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a borrow is generated, 0 otherwise.

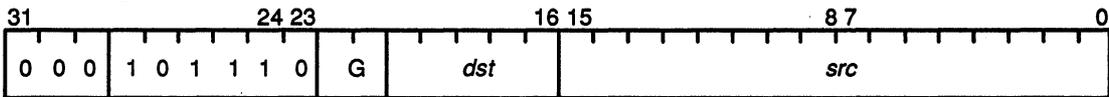
**Mode Bit** **OVM** Operation is affected by OVM bit value.

**Syntax**     **SUBC** *src*, *dst*

**Operation**   If ( $dst - src \geq 0$ ):  
                   ( $dst - src \ll 1$ ) OR 1  $\rightarrow dst$   
                   Else:  
                    $dst \ll 1 \rightarrow dst$

**Operands**    *src* general addressing modes (G):  
                   0 0    register (any register in CPU primary register file)  
                   0 1    direct  
                   1 0    indirect  
                   1 1    immediate  
  
                   *dst* register (any register in CPU primary register file)

**Encoding**



**Description**   The *src* operand is subtracted from the *dst* operand. The *dst* operand is loaded with a value that depends upon the result of the subtraction. If ( $dst - src$ ) is greater than or equal to zero, then ( $dst - src$ ) is left-shifted one bit, the least-significant bit is set to 1, and the result is loaded into the *dst* register. If ( $dst - src$ ) is less than zero, *dst* is left-shifted one bit and loaded into the *dst* register. The *dst* and *src* operands are assumed to be unsigned integers.

SUBC may be used to perform a single step of a multi-bit integer division. See subsection 12.3.4 for a detailed description.

**Cycles**        1

**Status Bits**   **LUF** Unaffected.  
                   **LV**   Unaffected.  
                   **UF**   Unaffected.  
                   **N**    Unaffected.  
                   **Z**    Unaffected.  
                   **V**    Unaffected.  
                   **C**    Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

## **SUBC** *Subtract Integer Conditionally*

---

**Example** SUBC @98C5h,R1

**Before Instruction:**

DP = 80h

R1 = 04F6h = 1270

Data at 80 98C5h = 492h = 1170

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

DP = 80h

R1 = 0C9h = 201

Data at 80 98C5h = 492h = 1170

LUF LV UF N Z V C = 0 0 0 0 0 0 0

**Example** SUBC 3000,R0 (3000 = 0BB8h)

**Before Instruction:**

R0 = 07D0h = 2000

LUF LV UF N Z V C = 0 0 0 0 0 0 0

After Instruction:

R0 = 0FA0h = 4000

LUF LV UF N Z V C = 0 0 0 0 0 0 0



## SUBF3 Subtract Floating-Point, 3 Operands

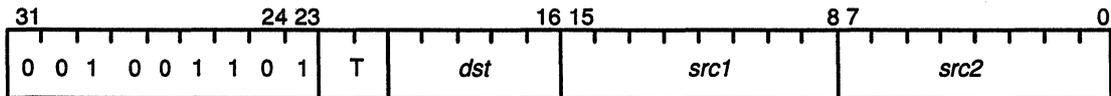
**Syntax**     **SUBF3** *src2, src1, dst*

**Operation**   *src1* – *src2* → *dst*

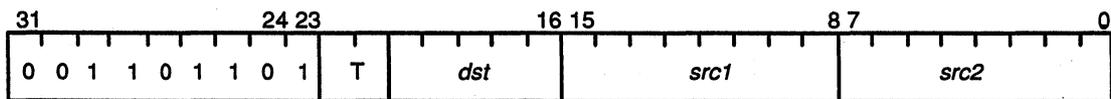
**Operands**   *src1, src2*   both type 1 or type 2 three-operand addressing modes  
                   *dst*            register mode (R0 – R11)

### Encoding

#### Type 1



#### Type 2



### Instruction Word Fields

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
Type 1	00	register mode (R0 — R11)	register mode (R0 — R11)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (R0 — R11)
	10	register mode (R0 — R11)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
Type 2	01	register mode (any CPU register)	indirect mode $*+ARn$ (5-bit unsigned displacement)
	11	indirect mode $*+ARn1$ (5-bit unsigned displacement)	indirect mode $*+ARn2$ (5-bit unsigned displacement)

**Description** The difference of the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be floating-point numbers.

**Cycles** 1

**Status Bits**

- LUF** 1 if a floating-point underflow occurs, unchanged otherwise.
- LV** 1 if an floating-point overflow occurs, unchanged otherwise.
- UF** 1 if a floating-point underflow occurs, 0 otherwise.
- N** 1 if a negative result is generated, 0 otherwise.
- Z** 1 if a zero result is generated, 0 otherwise.
- V** 1 if an floating-point overflow occurs, 0 otherwise.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

## SUBF3||STF *Parallel SUBF3 and STF*

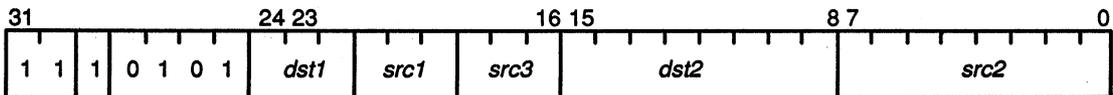
---

**Syntax**            **SUBF3** *src1, src2, dst1*  
                      || **STF** *src3, dst2*

**Operation**        *src2* – *src1* → *dst1*  
                      || *src3* → *dst2*

**Operands**        *src1* register (R0 – R7)  
                      *src2* indirect (disp = 0, 1, IR0, IR1)  
                      *dst1* register (R0 – R7)  
                      *src3* register (R0 – R7)  
                      *dst2* indirect (disp = 0, 1, IR0, IR1)

### Encoding



**Description** A floating-point subtraction and a floating-point store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STF) reads from a register and the operation being performed in parallel (SUBF3) writes to the same register, then STF accepts as input the contents of the register before it is modified by the SUBF3.

If *src3* and *dst1* point to the same location, *src3* is read before the write to *dst1*.

**Cycles**            1

**Status Bits**    **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.  
                      **LV** 1 if an floating-point overflow occurs, unchanged otherwise.  
                      **UF** 1 if a floating-point underflow occurs, 0 otherwise.  
                      **N** 1 if a negative result is generated, 0 otherwise.  
                      **Z** 1 if a zero result is generated, 0 otherwise.  
                      **V** 1 if an floating-point overflow occurs, 0 otherwise.  
                      **C** Unaffected.

**Mode Bit**        **OVM** Operation is not affected by OVM bit value.

**Example**       SUBF3 R1, \*-AR4 (IR1), R0  
 || STF R7, \*+AR5 (IR0)

**Before Instruction:**

R1 = 05 7B40 0000h = 6.28125e + 01  
 AR4 = 80 98B8h  
 IR1 = 8h  
 R0 = 0h  
 R7 = 07 33C0 0000h = 1.79750e + 02  
 AR5 = 80 9850h  
 IR0 = 10h  
 Data at 80 98B0h = 70C 8000h = 1.4050e + 02  
 Data at 80 9860h = 0h  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R1 = 05 7B40 0000h = 6.28125e + 01  
 AR4 = 8098B8h  
 IR1 = 8h  
 R0 = 06 1B60 0000h = 7.768750e + 01  
 R7 = 07 33C0 0000h = 1.79750e + 02  
 AR5 = 80 9850h  
 IR0 = 10h  
 Data at 80 98B0h = 70C 8000h = 1.4050e + 02  
 Data at 80 9860h = 733 C000h = 1.79750e + 02  
 LUF LV UF N Z V C = 0 0 0 0 0 0 0

## SUBI *Subtract Integer*

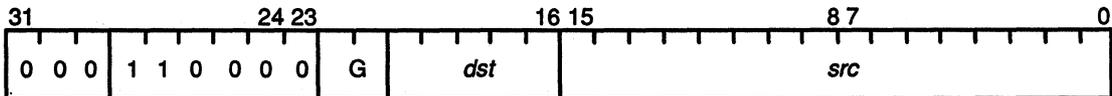
---

**Syntax**     **SUBI** *src, dst*

**Operation**   *dst* – *src* → *dst*

**Operands**   *src* general addressing modes (G):  
                0 0   register (any register in CPU primary register file)  
                0 1   direct  
                1 0   indirect  
                1 1   immediate  
  
                *dst* register (any register in CPU primary register file)

### Encoding



**Description** The difference of the *dst* operand minus the *src* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be signed integers.

**Cycles**       1

**Status Bits** If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a borrow occurs, 0 otherwise

**Mode Bit**   **OVM** Operation is affected by OVM bit value.

**Example**     SUBI 220, R7

### Before Instruction:

R7 = 226h = 550

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

R7 = 14Ah = 330

LUF LV UF N Z V C = 0 0 0 0 0 0 0

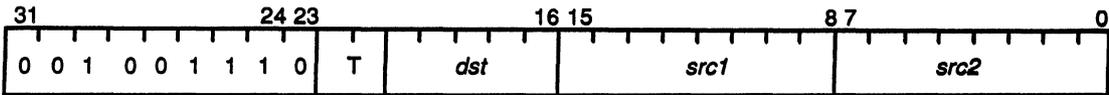
**Syntax**      **SUBI3** *src2, src1, dst*

**Operation**    *src1* – *src2* → *dst*

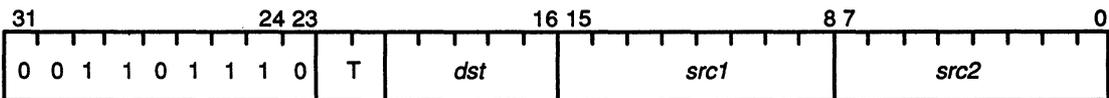
**Operands**    *src1, src2*    both type 1 or type 2 three-operand addressing modes  
                   *dst*                    register mode (any register in CPU primary register file)

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	<i>src1</i> addressing modes	<i>src2</i> addressing modes
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

## **SUBI3** *Subtract Integer, 3 Operands*

---

**Description** The result of the *src1* operand minus the *src2* operand is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** 1 if an integer overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an integer overflow occurs, 0 otherwise.

**C** 1 if a borrow is generated, 0 otherwise.

**Mode Bit** **OVM** Operation is affected by OVM bit value.



**Example**        SUBI3 R7, \*+AR2 (IR0), R1  
                 || STI R3, \*++AR7

**Before Instruction:**

R7 = 14h = 20  
AR2 = 80 982Fh  
IR0 = 10h  
R1 = 0h  
R3 = 35h = 53  
AR7 = 80 983Bh  
Data at 80 983Fh = 0DCh = 220  
Data at 80 983Ch = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

R7 = 14h = 20  
AR2 = 80 982Fh  
IR0 = 10h  
R1 = 0C8h = 200  
R3 = 35h = 53  
AR7 = 80 983Ch  
Data at 80 983Fh = 0DCh = 220  
Data at 80 983Ch = 35h = 53  
LUF LV UF N Z V C = 0 0 0 0 0 0 0



## SUBRF *Subtract Reverse Floating-Point Values*

---

**Syntax**     **SUBRF** *src, dst*

**Operation**   *src* – *dst* → *dst*

**Operands**   *src* general addressing modes (G):

    0 0   register (R0 – R11)

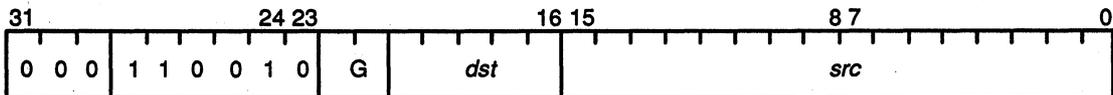
    0 1   direct

    1 0   indirect

    1 1   immediate

*dst* register (R0 – R11)

### Encoding



**Description** The result of the *src* operand minus the *dst* operand is loaded into the *dst* register. The *dst* and *src* operands are assumed to be floating-point numbers.

**Cycles**       1

**Status Bits** **LUF** 1 if a floating-point underflow occurs, unchanged otherwise.

**LV** 1 if a floating-point overflow occurs, unchanged otherwise.

**UF** 1 if a floating-point underflow occurs, 0 otherwise.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if a floating-point overflow occurs, 0 otherwise.

**C** Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

**Example**     SUBRF @9905h, R5

### Before Instruction:

DP = 80h

R5 = 05 7B40 0000h = 6.281250e + 01

Data at 80 9905h = 733 C000h = 1.79750e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0

### After Instruction:

DP = 80h

R5 = 06 69E0 0000h = 1.16937500e + 02

Data at 80 9905h = 733 C000h = 1.79750e + 02

LUF LV UF N Z V C = 0 0 0 0 0 0 0



## SWI *Software Interrupt*

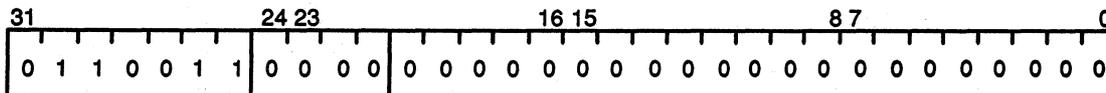
---

**Syntax** SWI

**Operation** Performs an emulation interrupt

**Operands** None

**Encoding**



**Description** The SWI instruction performs an emulator interrupt. This is a reserved instruction and should not be used in normal programming.

**Cycles** 4

**Status Bits** LUF Unaffected.  
LV Unaffected.  
UF Unaffected.  
N Unaffected.  
Z Unaffected.  
V Unaffected.  
C Unaffected.

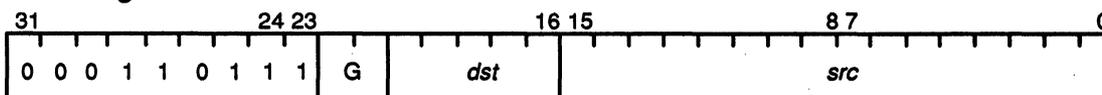
**Mode Bit** OVM Operation is not affected by OVM bit value.

**Syntax** **TOIEEE** *src, dst*

**Operation** convert *src* to IEEE format → *dst*

**Operands** *src* extended-precision register (R0 – R11),  
direct and indirect addressing modes  
*dst* extended-precision register

**Encoding**



**Instruction Word Fields**

G	src addressing modes
00	register mode (extended-precision register R0 – R11)
01	direct mode
10	indirect mode

**Description** The *src* operand is converted from a twos-complement floating-point format to the IEEE floating-point format.

The *src* operand is assumed to be a single-precision floating-point number. The converted result goes into the 32 MSBs of the *dst* register. STF can be used to store the result to memory.

**Cycles** 1

**Status Bits** **LUF** Unaffected.

**LV** 1 if an overflow occurs, unchanged otherwise.

**UF** 0.

**N** 1 if a negative result is generated, 0 otherwise.

**Z** 1 if a zero result is generated, 0 otherwise.

**V** 1 if an overflow occurs, 0 otherwise.

**C** Unaffected.

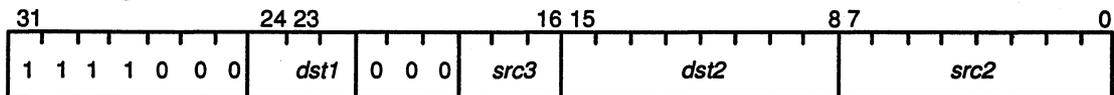
**Mode Bit** **OVM** Operation is not affected by OVM bit value.

**Syntax**            **TOIEEE** *src2, dst1*  
                       || **STF** *src3, dst2*

**Operation**        convert *src2* to IEEE format → *dst1*  
                       in parallel with  
                       *src3* → *dst2*

**Operands**        *src2*    indirect mode (disp = 0, 1, IRO, IR1)  
                       *dst1*    register mode (Rn1, 0 ≤ n1 ≤ 7)  
                       *src3*    register mode (Rn1, 0 ≤ n1 ≤ 7)  
                       *dst2*    indirect mode (disp = 0, 1, IRO, IR1)

**Encoding**



**Description**    The *src2* operand is converted from a twos-complement floating-point format to the IEEE floating-point format.

The *src2* operand is assumed to be a single-precision floating-point number. The converted result goes into the 32 MSBs of the *dst1* register. In parallel a floating-point store is done.

If *src2* and *dst2* point to the same location, then *src2* is read before the write to *dst2*.

**Cycles**            1

**Status Bits**    **LUF** Unaffected.  
                       **LV** 1 if an overflow occurs, unchanged otherwise.  
                       **UF** 0.  
                       **N** 1 if a negative result is generated, 0 otherwise.  
                       **Z** 1 if a zero result is generated, 0 otherwise.  
                       **V** 1 if an overflow occurs, 0 otherwise.  
                       **C** Unaffected.

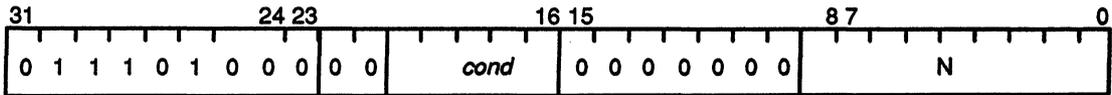
**Mode Bit**        **OVM** Operation is not affected by OVM bit value.

**Syntax** TRAPcond N

**Operation** If (*cond* is true)  
 ST(GIE) → ST(PGIE)  
 ST(CF) → ST(PCF)  
 0 → ST(GIE)  
 1 → ST(CF)  
 next PC → \*(++SP)  
 trap vector N → PC  
 Else, continue.

**Operands** N immediate mode ( $0 \leq N \leq 511$ )

**Encoding**



**Description** If traps are to be nested, you may need to save the status register before executing TRAPcond. If the condition is true, then GIE and CF are saved in PGIE and PCF in the status register, all interrupts are disabled (0 → GIE), and the cache is frozen (1 → CF). Then, the contents of the PC are pushed onto the system stack, and the PC is loaded with the contents of the specified trap vector (N). If the condition is not true, then continue normal operation.

**Cycles** 5

**Status Bits** GIE Set to 0 if TRAP executes.  
 LUF Unaffected.  
 LV Unaffected.  
 UF Unaffected.  
 N Unaffected.  
 Z Unaffected.  
 V Unaffected.  
 C Unaffected.

**Mode Bit** OVM Operation is not affected by OVM bit value.



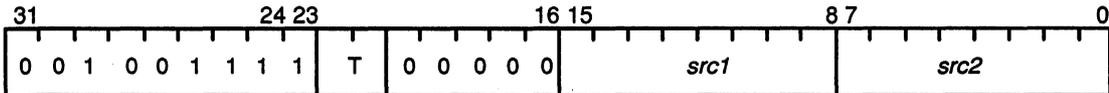
**Syntax**     **TSTB3** *src2*, *src1*

**Operation**   *src1* & *src2*

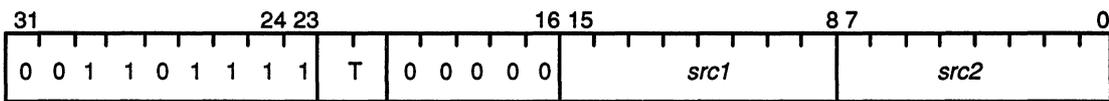
**Operands**   *src1*, *src2*   both type 1 or type 2 three-operand addressing modes

**Encoding**

**Type 1**



**Type 2**



**Instruction Word Fields**

	<b>T</b>	<b>src1 addressing modes</b>	<b>src2 addressing modes</b>
<b>Type 1</b>	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	<b>T</b>	<b>src1 addressing modes</b>	<b>src2 addressing modes</b>
<b>Type 2</b>	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

## TSTB3 *Test Bit Fields, 3 Operands*

---

**Description** The bitwise logical AND between the *src1* and *src2* operands is formed but is not loaded into any register. This allows for nondestructive compares. The *src1* and *src2* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits**

- LUF** Unaffected.
- LV** Unaffected.
- UF** 0.
- N** MSB of the output.
- Z** 1 if a zero output is generated, 0 otherwise.
- V** 0.
- C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.



## XOR3 *Bitwise Exclusive OR, 3 Operands*

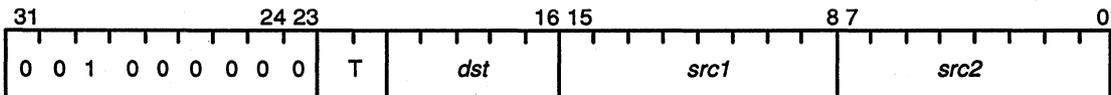
**Syntax**     **XOR3** *src2, src1, dst*

**Operation**   *src1 XOR src2 → dst*

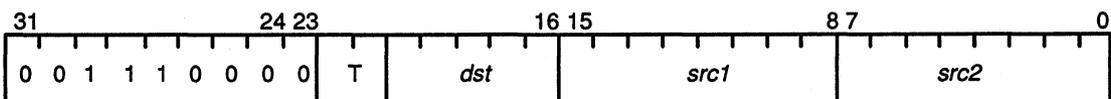
**Operands**   *src1, src2*   both type 1 or type 2 three-operand addressing modes  
                   *dst*            register mode (any register in CPU primary register file)

### Encoding

#### Type 1



#### Type 2



### Instruction Word Fields

	T	src1 addressing modes	src2 addressing modes
Type 1	00	register mode (any CPU register)	register mode (any CPU register)
	01	indirect mode (disp = 0, 1, IR0, IR1)	register mode (any CPU register)
	10	register mode (any CPU register)	indirect mode (disp = 0, 1, IR0, IR1)
	11	indirect mode (disp = 0, 1, IR0, IR1)	indirect mode (disp = 0, 1, IR0, IR1)

	T	src1 addressing modes	src2 addressing modes
Type 2	00	register mode (any CPU register)	8-bit signed immediate
	01	register mode (any CPU register)	indirect mode *+ARn(5-bit unsigned displacement)
	10	indirect mode *+ARn(5-bit unsigned displacement)	8-bit signed immediate
	11	indirect mode *+ARn1(5-bit unsigned displacement)	indirect mode *+ARn2(5-bit unsigned displacement)

**Description** The bitwise exclusive OR between the *src1* and *src2* operands is loaded into the *dst* register. The *src1*, *src2*, and *dst* operands are assumed to be signed integers.

**Cycles** 1

**Status Bits** If ST (SETCOND) = 0 and the destination register is R0 – R11, the condition flags are modified. If ST (SETCOND) = 1, they are modified for all destination registers.

**LUF** Unaffected.

**LV** Unaffected.

**UF** 0.

**N** MSB of the output.

**Z** 1 if a zero output is generated, 0 otherwise.

**V** 0.

**C** Unaffected.

**Mode Bit** **OVM** Operation is not affected by OVM bit value.

## XOR3||STI *Parallel XOR3 and STI*

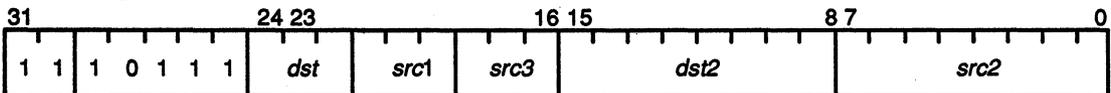
---

**Syntax**        **XOR3** *src2, src1, dst1*  
                  || **STI** *src3, dst2*

**Operation**     *src1 XOR src2* → *dst1*  
                  || *src3* → *dst2*

**Operands**    *src1* register (R0 – R7)  
                  *src2* indirect (disp = 0, 1, IR0, IR1)  
                  *dst1* register (R0 – R7)  
                  *src3* register (R0 – R7)  
                  *dst2* indirect (disp = 0, 1, IR0, IR1)

### Encoding



**Description** A bitwise exclusive-XOR and an integer store are performed in parallel. All registers are read at the beginning and loaded at the end of the execute cycle. This means that if one of the parallel operations (STI) reads from a register and the operation being performed in parallel (XOR3) writes to the same register, then STI accepts as input the contents of the register before it is modified by the XOR3.

If *src2* and *dst2* point to the same location, *src2* is read before the write to *dst2*.

**Cycles**        1

**Status Bits** **LUF** Unaffected.  
                  **LV** Unaffected.  
                  **UF** 0.  
                  **N** MSB of the output.  
                  **Z** 1 if a zero output is generated, 0 otherwise.  
                  **V** 0.  
                  **C** Unaffected.

**Mode Bit**     **OVM** Operation is not affected by OVM bit value.

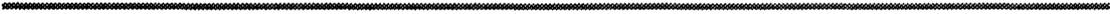
**Example** XOR3 \*AR1++, R3, R3  
||  
STI R6, \*-AR2 (IR0)

**Before Instruction:**

AR1 = 80 987Eh  
R3 = 85h  
R6 = 0DCh = 220  
AR2 = 80 98B4h  
IR0 = 8h  
Data at 80 987Eh = 85h  
Data at 80 98ACh = 0h  
LUF LV UF N Z V C = 0 0 0 0 0 0 0

**After Instruction:**

AR1 = 80 987Fh  
R3 = 0h  
R6 = 0DCh = 220  
AR2 = 80 98B4h  
IR0 = 8h  
Data at 80 987Eh = 85h  
Data at 80 98ACh = 0DCh = 220  
LUF LV UF N Z V C = 0 0 0 0 0 0 0



## Chapter 12

# Software Applications

---

---

---

This chapter explains how to use the instruction set, the architecture, and the interface of the 'C40. It presents coding examples for frequently used applications and discusses more involved examples and applications. It also defines the principles involved in the application and gives the corresponding assembly-language code for instructional purposes and for immediate use. Whenever the detailed explanation of the underlying theory is too extensive to be included in this manual, appropriate references are given for further information.

Major topics discussed in this chapter are listed below:

<b>Section</b>	<b>Page</b>
12.1 Processor Initialization .....	12-3
■ Reset Process .....	12-3
■ Initialization .....	12-3
12.2 Program Control .....	12-9
■ Regular and Zero-Overhead Subroutine Calls .....	12-9
■ Software Stack .....	12-13
■ Interrupt Service Routines .....	12-14
■ Delayed Branches .....	12-22
■ Repeat Modes .....	12-23
■ Computed GOTOs .....	12-27
12.3 Logical and Arithmetic Operations .....	12-28
■ Bit Manipulation .....	12-28
■ Block Moves .....	12-29
■ Byte and Half Word Manipulation .....	12-30
■ Bit Reversed Addressing .....	12-31

Section	Page
■ Division .....	12-33
■ Square Root .....	12-38
■ Extended-Precision Arithmetic .....	12-41
■ IEEE <==> C40 Floating-Point Conversions .....	12-42
12.4 Application-Oriented Operations .....	12-46
■ Companding (A-law/ $\mu$ -law) .....	12-46
■ FIR/IIR Filters (fixed and adaptive) .....	12-51
■ Matrix Math .....	12-61
■ FFT .....	12-63
■ Lattice Filters .....	12-88
12.5 Programming Tips .....	12-94
■ C-Callable Routines .....	12-94
■ Hints for Assembly Coding .....	12-95
12.6 Peripherals .....	12-97
■ Timer .....	12-97
■ Communication Port .....	12-98
■ Direct Memory Access .....	12-101

## 12.1 Processor Initialization

### 12.1.1 Reset Process

Before you execute a DSP algorithm, it is necessary to initialize the processor. Generally, initialization takes place any time the processor is reset.

When reset is activated by applying a low level to the RESET input for several cycles, the 'C40 terminates execution and puts the RESET vector in the program counter. The RESET vector of 'C40 may be mapped to one of four different locations, controlled by the value of the RESETLOC(1,0) pins at RESET (shown in Table 12-1). The RESET vector normally contains the address of the system initialization routine. The hardware reset also initializes various registers and status bits (reset conditions are further defined in Section 6.6 on page 6-18).

Table 12-1. Relationship of RESETLOC(1,0) Pins to RESET Vector Location

RESETLOC(1,0)	RESET Vector Address
0 0	0000 0000h
0 1	7FFF FFFFh
1 0	8000 0000h
1 1	FFFF FFFFh

After reset, initialize the processor further by executing instructions that set up operational modes, memory pointers, interrupts, and the remaining functions needed to meet system requirements.

### 12.1.2 Initialization

To configure the processor at reset, the following internal functions should be initialized:

- CPU expansion register file
- Memory-mapped registers
- Interrupt structure

Example 12–1 shows coding for initializing the 'C40 to the following machine state, in addition to the initialization performed during the hardware reset (for conditions after hardware reset, see Section 6.6 on page 6-18):

- All interrupts are enabled.
- The program cache is enabled.
- The overflow mode is disabled.
- The data memory page pointer is set to zero.
- The stack pointer is set to internal RAM address 002FFF00H
- The internal memory is filled with zeros.

Note that all constants larger than 16 bits should be placed in memory and accessed through direct or indirect addressing.

### Example 12–1. Processor Initialization Example

```
*
*      TITLE 'PROCESSOR INITIALIZATION EXAMPLE'
*
.global  RESET, INIT, BEGIN
.global  TIME0, TIME1, TINT0, TINT1
.global  NMI, INT0, INT1, INT2, INT3
.global  NON_MASK, ISR0, ISR1, ISR2, ISR3
.global  ICFULL0, ICRDY0, OCRDY0, OCEMPTY0
.global  ICFSR0, ICRSR0, OCRSR0, OCESR0
.global  ICFULL1, ICRDY1, OCRDY1, OCEMPTY1
.global  ICFSR1, ICRSR1, OCRSR1, OCESR1
.global  ICFULL2, ICRDY2, OCRDY2, OCEMPTY2
.global  ICFSR2, ICRSR2, OCRSR2, OCESR2
.global  ICFULL3, ICRDY3, OCRDY3, OCEMPTY3
.global  ICFSR3, ICRSR3, OCRSR3, OCESR3
.global  ICFULL4, ICRDY4, OCRDY4, OCEMPTY4
.global  ICFSR4, ICRSR4, OCRSR4, OCESR4
.global  ICFULL5, ICRDY5, OCRDY5, OCEMPTY5
.global  ICFSR5, ICRSR5, OCRSR5, OCESR5
.global  DINT0, DINT1, DINT2, DINT3, DINT4, DINT5
.global  DMA0, DMA1, DMA2, DMA3, DMA4, DMA5
.global  TRAP0, TRAP1, TRAP2, TRAP3, TRAP4, TRAP5
.global  TRP0, TRP1, TRP2, TRP3, TRP4, TRP5
```

## Example 12-1. Processor Initialization Example (Continued)

```

*
*   PROCESSOR INITIALIZATION FOR THE TMS320C40.
*
*   RESET AND INTERRUPT VECTOR SPECIFICATION. THIS ARRANGEMENT
*   ASSUMES THAT DURING LINKING, THE FOLLOWING TEXT SEGMENT
*   WILL BE PLACED TO START AT MEMORY LOCATION AS:
*
*   SEGMENT NAME | MEMORY LOCATION
*   -----
*   reset_adr    | SAME AS RESETLOC(1,0) SETUP
*   int_vect     | 00000200h
*   trap_vect    | 00000400h
*   .data        | 00000500h
*   .text        | 00000600h
*
*   NOTE THAT THE INTERRUPT AND TRAP VECTORS TABLE CAN BE
*   RELOCATED TO A 512-WORD BOUNDARY BY CHANGING THE VALUES
*   OF THE IVTP AND TVTP.
*
*   .sect "reset_adr" ; Named section for RESET vector
RESET .word INIT ; RS-load address INIT to PC
*
*   .sect "int_vect" ; Named section for interrupt
*   ; structures
*   .space 1 ; Reserved space
NMI .word NON_MASK ; Non Maskable Interrupt NMI-loads
*   ; address NMI to PC
TINT0 .word TIME0 ; Timer 0 interrupt processing
*
INT0 .word ISR0 ; INT0- loads address INT0 to PC
INT1 .word ISR1 ; INT1- loads address INT1 to PC
INT2 .word ISR2 ; INT2- loads address INT2 to PC
INT3 .word ISR3 ; INT3- loads address INT3 to PC
*   .space 6 ; Reserved space
ICFULL0 .word ICFSR0 ; Comm. port 0 input full processing
ICRDY0 .word ICRSR0 ; Comm. port 0 input ready processing
OCRDY0 .word OCRSR0 ; Comm. port 0 output ready processing
OCEMPTY0 .word OCESR0 ; Comm. port 0 output empty processing
*
ICFULL1 .word ICFSR1 ; Comm. port 1 input full processing
ICRDY1 .word ICRSR1 ; Comm. port 1 input ready processing
OCRDY1 .word OCRSR1 ; Comm. port 1 output ready processing
OCEMPTY1 .word OCESR1 ; Comm. port 1 output empty processing
*
ICFULL2 .word ICFSR2 ; Comm. port 2 input full processing
ICRDY2 .word ICRSR2 ; Comm. port 2 input ready processing
OCRDY2 .word OCRSR2 ; Comm. port 2 output ready processing
OCEMPTY2 .word OCESR2 ; Comm. port 2 output empty processing
*
ICFULL3 .word ICFSR3 ; Comm. port 3 input full processing
ICRDY3 .word ICRSR3 ; Comm. port 3 input ready processing
OCRDY3 .word OCRSR3 ; Comm. port 3 output ready processing
OCEMPTY3 .word OCESR3 ; Comm. port 3 output empty processing

```

Example 12-1. Processor Initialization Example (Continued)

```

ICFULL4 .word ICFSR4      ; Comm. port 4 input full processing
ICRDY4  .word ICRSR4      ; Comm. port 4 input ready processing
OCRDY4  .word OCRSR4      ; Comm. port 4 output ready processing
OCEMPTY4 .word OCESR4     ; Comm. port 4 output empty processing
*
ICFULL5 .word ICFSR5      ; Comm. port 5 input full processing
ICRDY5  .word ICRSR5      ; Comm. port 5 input ready processing
OCRDY5  .word OCRSR5      ; Comm. port 5 output ready processing
OCEMPTY5 .word OCESR5     ; Comm. port 5 output empty processing
*
DINT0   .word DMA0        ; DMA Channel 0 interrupt
DINT1   .word DMA1        ; DMA Channel 1 interrupt
DINT2   .word DMA2        ; DMA Channel 2 interrupt
DINT3   .word DMA3        ; DMA Channel 3 interrupt
DINT4   .word DMA4        ; DMA Channel 4 interrupt
DINT5   .word DMA5        ; DMA Channel 5 interrupt
TINT1   .word TIME1       ; Timer 1 interrupt processing
        .space 21         ; Reserved space
*
        .sect "trap_vect" ; Named section for trap structures
TRAP0   .word TRP0        ; Trap 0 vector processing begins
TRAP1   .word TRP1        ; Trap 1 vector processing begins
TRAP2   .word TRP2        ; Trap 2 vector processing begins
TRAP3   .word TRP3        ; Trap 3 vector processing begins
TRAP4   .word TRP4        ; Trap 4 vector processing begins
TRAP5   .word TRP5        ; Trap 5 vector processing begins
        .space 506       ; Leave space for the other 506 traps
*
*       IN THIS SECTION, CONSTANTS THAT CANNOT BE REPRESENTED
*       IN THE SHORT FORMAT ARE INITIALIZED.
        .data
MASK    .word 0FFFFFFFH
BLK0    .word 02FF800H    ; Beginning address of RAM block 0
BLK1    .word 02FFC00H    ; Beginning address of RAM block 1
CTRL    .word 0100000H    ; Pointer for peripheral-bus memory map
GLOINT  .word 0000000H    ; Init of global memory interface
        ; control (0)
LOCALINT .word 0000000H   ; Init of local memory interface
        ; control (4)
DMA0CTL .word 0000000H    ; Initialization for DMA 0 control (160)
DMA1CTL .word 0000000H    ; Initialization for DMA 1 control (176)
DMA2CTL .word 0000000H    ; Initialization for DMA 2 control (192)
DMA3CTL .word 0000000H    ; Initialization for DMA 3 control (208)
DMA4CTL .word 0000000H    ; Initialization for DMA 4 control (224)
DMA5CTL .word 0000000H    ; Initialization for DMA 5 control (240)
CP0CTL  .word 0000000H    ; Init of comm. port 0 control (64)
CP1CTL  .word 0000000H    ; Init of comm. port 1 control (80)
CP2CTL  .word 0000000H    ; Init of comm. port 2 control (96)
CP3CTL  .word 0000000H    ; Init of comm. port 3 control (112)
CP4CTL  .word 0000000H    ; Init of comm. port 4 control (128)
CP5CTL  .word 0000000H    ; Init of comm. port 5 control (144)
TIM0CTL .word 0000000H    ; Initialization of timer 0 control (32)
TIM1CTL .word 0000000H    ; Initialization of timer 1 control (48)

```

## Example 12-1. Processor Initialization Example (Continued)

```

ANACTL .word 0000000H ; Initialization of analysis module (16)
STCK .word 02FFF00H ; Beginning of stack
*
* .text
*
* THE ADDRESS AT RESET VECTOR DIRECTS EXECUTION TO BEGIN HERE
* FOR RESET PROCESSING THAT INITIALIZES THE PROCESSOR. WHEN
* RESET IS APPLIED, THE FOLLOWING REGISTERS ARE INITIALIZED
* TO ZERO:
*
* ST -- CPU STATUS REGISTER
* DIE -- DMA INTERRUPT ENABLE REGISTER
* IIE -- INTERNAL INTERRUPT ENABLE REGISTER
* IIF -- IIOF PINS AND INTERRUPT FLAG REGISTER
*
* IVTP -- INTERRUPT-VECTOR TABLE POINTER
* TVTP -- TRAP-VECTOR TABLE POINTER
*
* THE STATUS REGISTER HAS THE FOLLOWING ARRANGEMENT:
* BITS: 31-17 16 15 14 13 12 11 10
* FUNCTION: RESRV ANALYSIS SET PGIE GIE CC CE CF
* IDLE COND
* BITS: 9 8 7 6 5 4 3 2 1 0
* FUNCTION: PCF RM OVM LUF LV UF N Z V C
*
INIT LDPX MASK ; Point the DP register to page 0
LDI 1800H,ST ; Clear and enable cache, and
; disable OVM
*
* SET UP IVTP AND TVTP TO 200H AND 400H
*
LDI 0200H,AR0 ; Set Primary Register AR0 to 200H
LDPE AR0,IVTP ; Set Expansion Register IVTP to 200H
ADDI 0200H,AR0 ; Set Primary Register AR0 to 400H
LDPE AR0,TVTP ; Set Expansion Register TVTP to 400H
LDI @MASK,IE ; Enable all interrupts
*
* INTERNAL DATA MEMORY INITIALIZATION TO FLOATING POINT ZERO
*
LDI @BLK0,AR0 ; AR0 points to block 0
LDI @BLK1,AR1 ; AR1 points to block 1
LDF 0.0,R0 ; Zero register R0
RPTS 1023 ; Repeat 1024 times ...
STF R0,*AR0++(1) ; Zero out location in RAM block 0 and
|| STF R0,*AR1++(1) ; Zero out location in RAM block 1.
*
* THE PROCESSOR IS INITIALIZED. THE REMAINING APPLICATION-
* DEPENDENT PART OF THE SYSTEM (BOTH ON- AND OFF-CHIP SHOULD
* NOW BE INITIALIZED.
*
* FIRST, INITIALIZE THE CONTROL REGISTERS. IN THIS EXAMPLE,
* EVERYTHING IS INITIALIZED TO ZERO SINCE THE ACTUAL
* INITIALIZATION IS APPLICATION DEPENDENT.
*

```

Example 12-1. Processor Initialization Example (Concluded)

```

LDI    @CTRL,ARO           ; LOAD in ARO the pointer to control
                        ; registers

LDI    @GLOINT,R0
STI    R0,*ARO             ; Init global memory interface control
LDI    @LOCALINT,R0
STI    R0,*+ARO(4)        ; Init local memory interface control
LDI    @DMA0CTL,R0
STI    R0,*+ARO(160)      ; Init DMA 0 control
LDI    @DMA1CTL,R0
STI    R0,*+ARO(176)      ; Init DMA 1 control
LDI    @DMA2CTL,R0
STI    R0,*+ARO(192)      ; Init DMA 2 control
LDI    @DMA3CTL,R0
STI    R0,*+ARO(208)      ; Init DMA 3 control
LDI    @DMA4CTL,R0
STI    R0,*+ARO(224)      ; Init DMA 4 control
LDI    @DMA5CTL,R0
STI    R0,*+ARO(240)      ; Init DMA 5 control
LDI    @CP0CTL,R0
STI    R0,*+ARO(64)       ; Init communication port 0 control
LDI    @CP1CTL,R0
STI    R0,*+ARO(80)       ; Init communication port 0 control
LDI    @CP2CTL,R0
STI    R0,*+ARO(96)       ; Init communication port 0 control
LDI    @CP3CTL,R0
STI    R0,*+ARO(112)      ; Init communication port 0 control
LDI    @CP4CTL,R0
STI    R0,*+ARO(128)      ; Init communication port 0 control
LDI    @CP5CTL,R0
STI    R0,*+ARO(144)      ; Init communication port 0 control
LDI    @TIM0CTL,R0
STI    R0,*+ARO(32)       ; Init timer 0 control
LDI    @TIM1CTL,R0
STI    R0,*+ARO(48)       ; Init timer 1 control
LDI    @ANACTL,R0
STI    R0,*+ARO(16)       ; Init analysis module control
*
LDI    @STCK,SP           ; Initialize the stack pointer
OR     2000H,ST           ; Global interrupt enable
*
BR     BEGIN             ; Branch to the beginning of
                        ; application.

.end

```

## 12.2 Program Control

TMS320C40 instructions provide program control and facilitates high-speed processing. These instructions directly handle:

- ❑ Regular and zero-overhead subroutine calls
- ❑ Software stack
- ❑ Interrupts
- ❑ Delayed branches
- ❑ Single- and multiple-instruction loops without any overhead

### 12.2.1 Subroutines

The 'C40 provides two ways to invoke the subroutine calls: regular and zero-overhead. The **regular** and **zero-overhead** subroutine calls use software stack (SP) and extended-precision register R11 respectively to save the return address. The following subsections use example programs to explain how this works.

#### 12.2.1.1 Regular Subroutine Calls

The 'C40 has a 32-bit program counter (PC) and a practically unlimited software stack. The `CALL` and `CALLcond` subroutine calls cause the stack pointer to increment and store the contents of the next value of the PC counter on the stack. At the end of the subroutine, `RETScond` performs a conditional return.

Example 12–2 illustrates the use of a subroutine to determine the dot product between two vectors. Given two vectors of length  $N$ , represented by the arrays  $a[0], a[1], \dots, a[N-1]$  and  $b[0], b[1], \dots, b[N-1]$ , the dot product is computed from the expression

$$d = a[0] b[0] + a[1] b[1] + \dots + a[N-1] b[N-1]$$

Processing proceeds in the main routine to the point where the dot product is to be computed. It is assumed that the arguments of the subroutine have been appropriately initialized. At this point, a `CALL` is made to the subroutine, transferring control to that section of the program memory for execution, then returning to the calling routine via the `RETS` instruction when execution has completed. Note that for this particular example, it would suffice to save the register R2. However, a larger number of registers are saved for demonstration purposes. The saved registers are stored on the system stack, which should be large enough to accommodate the maximum anticipated storage requirements. Other methods of saving registers could be used equally well.

Example 12-2. Regular Subroutine Call (Dot Product)

```

*
*      TITLE REGULAR SUBROUTINE CALL (DOT PRODUCT)
*
*
*      MAIN ROUTINE THAT CALLS THE SUBROUTINE 'DOT' TO COMPUTE THE
*      DOT PRODUCT OF TWO VECTORS.
*
*      .
*      .
*      .
*      LDI      @blk0,AR0      ; AR0 points to vector a
*      LDI      @blk1,AR1      ; AR1 points to vector b
*      LDI      N,RC          ; RC contains the number of elements
CALL   DOT
*      .
*      .
*      .
*
*SUBROUTINE DOT
*
*
*EQUATION:  d = a(0) * b(0) + a(1) * b(1) + ... + a(N-1) * b(N-1)
*
*THE DOT PRODUCT OF a AND b IS PLACED IN REGISTER R0. N MUST
*BE GREATER THAN OR EQUAL TO 2.
*
*      ARGUMENT ASSIGNMENTS:
*      ARGUMENT | FUNCTION
*      -----+-----
*      AR0      | ADDRESS OF a(0)
*      AR1      | ADDRESS OF b(0)
*      RC       | LENGTH OF VECTORS (N)
*
*      REGISTERS USED AS INPUT: AR0, AR1, RC
*      REGISTER MODIFIED: R0
*      REGISTER CONTAINING RESULT: R0
*
*
*      .global  DOT
*
DOT    PUSH     ST           ; Save status register
      PUSH     R2          ; Use the stack to save R2's
      PUSHF    R2          ; bottom 32 and top 32 bits
      PUSH     AR0         ; Save AR0
      PUSH     AR1         ; Save AR1
      PUSH     RC          ; Save RC
      PUSH     RS
      PUSH     RE
*
*      ; Initialize R0:
      MPYF3    *AR0,*AR1,R0 ; a(0) * b(0) -> R0
||      SUBF   R2,R2,R2    ; Initialize R2.
      SUBI    2,RC        ; Set RC = N-2
*

```

```

*      DOT PRODUCT (1 <= i < N)

*
      RPTS      RC          ; Setup the repeat single.
      MPYF3    *++AR0(1), *++AR1(1), R0 ; a(i) * b(i) -> R0
      | |      ADDF3      R0, R2, R2    ; a(i-1)*b(i-1) + R2 -> R2
*
      ADDF3    R0, R2, R0          ; a(N-1)*b(N-1) + R2 -> R0
*
*      RETURN      SEQUENCE
*
      POP      RE
      POP      RS
      POP      RC          ; Restore RC
      POP      AR1        ; Restore AR1
      POP      AR0        ; Restore AR0
      POPF     R2          ; Restore top 32 bits of R2
      POP      R2          ; Restore bottom 32 bits of R2
      POP      ST          ; Restore ST
      RETS     ; Return

*
*      end
*
      .end

```

### 12.2.1.2 Zero-Overhead Subroutine Calls

Two 'C40 instructions, link and jump (LAJ) and link and jump conditional (LAJ*cond*), allow zero-overhead subroutine calls to be implemented on the 'C40. Unlike the CALL and CALL*cond* which put the value of PC+1 into the software stack, the LAJ and LAJ*cond* put the value of PC+4 into the extended-precision register R11. Three instructions following LAJ or LAJ*cond* will be executed before going to the subroutine. The restriction of these three instructions is the same as that of the three instructions following a delayed branch. At the end of the subroutine, a delayed branch conditional, B*cond*D, using the register addressing mode with R11 as source, can be used to perform a zero-overhead subroutine return.

For comparison, the same dot product example with zero-overhead subroutine call is given in the following example program.

Example 12-3. Zero-Overhead Subroutine Call (Dot Product)

```

*
*      TITLE ZERO-OVERHEAD SUBROUTINE CALL (DOT PRODUCT)
*
*
*      MAIN ROUTINE THAT CALLS THE SUBROUTINE 'DOT' TO COMPUTE THE
*      DOT PRODUCT OF TWO VECTORS.
*
*      .
*      .
*      .
LAJ      DOT
LDI      @blk0,AR0      ; AR0 points to vector a
LDI      @blk1,AR1      ; AR1 points to vector b
LDI      N,RC           ; RC contains the number of elements
*
*      .
*
*
*SUBROUTINE      DOT
*
*EQUATION:      d = a(0) * b(0) + a(1) * b(1) + ... + a(N-1) * b(N-1)
*
*      THE DOT PRODUCT OF a AND b IS PLACED IN REGISTER R0. N MUST
*      BE GREATER THAN OR EQUAL TO 2.
*
*      ARGUMENT ASSIGNMENTS:
*      ARGUMENT | FUNCTION
*      -----+-----
*      AR0      | ADDRESS OF a(0)
*      AR1      | ADDRESS OF b(0)
*      RC       | LENGTH OF VECTORS (N)
*
*
*      REGISTERS USED AS INPUT: AR0, AR1, RC
*      REGISTER MODIFIED: R0
*      REGISTER CONTAINING RESULT: R0
*
*
*      .global      DOT
*
DOT      PUSH      ST           ; Save status register
        PUSH      R2           ; Use the stack to save R2's
        PUSHF    R2           ; bottom 32 and top 32 bits
        PUSH     AR0          ; Save AR0
        PUSH     AR1          ; Save AR1
        PUSH     RC           ; Save RC
        PUSH     RS
        PUSH     RE
*
*      MPYF3      *AR0,*AR1,R0      ; Initialize R0:
||      SUBF      R2.R2,R2          ; a(0) * b(0) -> R0
        SUBI     2,RC              ; Initialize R2.
*      SUBI     2,RC              ; Set RC = N-2

```

```

*      DOT PRODUCT (1 <= i < N)
*
RPTS      RC                ; Setup the repeat single
MPYF3     *++AR0(1), *++AR1(1), R0 ; a(i) * b(i) -> R0
| | ADDF3     R0, R2, R2      ; a(i-1)*b(i-1) + R2 -> R2
*
*      ADDF3     R0, R2, R0      ; a(N-1)*b(N-1) + R2 -> R0
*
*      RETURN SEQUENCE
*
POP        RE
POP        RS
POP        RC                ; Restore RC
POP        AR1              ; Restore AR1
POP        AR0              ; Restore AR0
BUD        R11              ; Return
POPF       R2                ; Restore top 32 bits of R2
POP        R2                ; Restore bottom 32 bits of R2
POP        ST                ; Restore ST
*
*      end
*
*      .end

```

### 12.2.2 Software Stack

Location of the 'C40 software stack is determined by the contents of the stack pointer register (SP). The stack pointer increments from low to high values, and provisions should be made to accommodate the anticipated storage requirements. The stack can be used not only during the subroutine CALL and RETS, but also inside the subroutine as a place of temporary storage of the registers as shown in Example 12–2. SP always points to the **last value pushed onto the stack**.

The CALL and CALL*cond* instructions push the value of the program counter onto the stack, as do the interrupt routines. Then, RETS*cond* and RETI*cond* pop the stack and place the value in the program counter. The integer value of any register can be pushed onto and popped off the stack by using the PUSH and POP instructions.

Two additional instructions, PUSHF and POPF, are for floating-point numbers. These instructions can be used to pop and push floating-point numbers to registers R0 — R11. This feature is very useful for saving the extended precision registers (see Example 12–2 and Example 12–3). You can use PUSH and PUSHF on the same register to save the lower 32 and upper 32 bits. PUSH saves the lower 32 bits; PUSHF, the upper 32 bits. To recover this extended-precision number, execute a POPF followed by POP. It is important to do the integer and floating-point PUSH and POP in the above order. POPF forces the last eight bits of the extended-precision registers to zero.

The stack pointer (SP) can be read as well as written to. Multiple stacks for different program segments may be easily created. SP is not initialized by the hardware during reset; therefore, it is important to remember to initialize its value so that SP points to a predetermined memory location. This avoids the problem of SP attempting to write into ROM or write over other useful data.

### **12.2.3 Interrupt Service Routines**

There are two types of interrupts on the 'C40: maskable and nonmaskable. The maskable interrupts include internal and external interrupts. All the interrupts are vectored and prioritized. The vector table for the various interrupts is located in relation to the interrupt-vector table pointer (IVTP, shown in Section 3.2 on page 3-15). The nonmaskable interrupt (NMI) has the highest priority over other interrupts. Unlike other interrupts, the NMI cannot be masked by its own mask or by the GIE bit in the ST. It is temporarily masked during delayed branches and multicycle CPU operation.

When an interrupt occurs, the corresponding flag is set in the interrupt flag register (IIF — explained in subsection 3.1.10, page 3-12). For nonmaskable interrupt, if the corresponding NMI flag is set, NMI begins the interrupt processing, as long as the CPU is not executing delayed branches or multicycle operation. For maskable interrupts, in order to respond to the interrupt when the corresponding interrupt flag is set, the GIE bit in the ST must be set to enable maskable interrupts globally, and the corresponding bit in the interrupt enable register (IIE — described in subsection 3.1.9, page 3-10) or IIF register (for external interrupts) must be set also. Since pins IIOF(3 — 0) can be either general-purpose I/O or external interrupt pins, you must configure (using IIF register) those pins as interrupt pins to enable an external interrupt. Also, if the IIOF(3 — 0) pins are configured as interrupt pins, they can be configured (also at IIF register) as either edge-triggered or level-triggered interrupts. You can also write to the IIF register, making it possible to force an interrupt by software or to clear interrupts without processing them.

The interrupt flag register can be read, and action can be taken, depending on whether the interrupt has occurred. This is true even when the maskable interrupt is disabled. This can be useful when an interrupt-driven interface is not implemented. Example 12-4 shows the case in which a subroutine is called when external interrupt 1 has not occurred.

*Example 12–4. Use of Interrupts for Software Polling*

```
*      TITLE INTERRUPT POLLING
      .
      .
      TSTB 40H,IIF          ; Test if interrupt 1 has occurred
      CALLZ SUBROUTINE     ; If not, call subroutine
      .
      .
      .
```

When interrupt processing begins, the program counter is pushed on the stack, and the interrupt vector is loaded in the program counter. Interrupts are then disabled by setting GIE=0, and the program continues from the address loaded in the program counter. Since all maskable interrupts are disabled, interrupt processing may proceed without further interruption unless the interrupt service routine re-enables interrupts, or the NMI occurs.

Except for very simple interrupt service routines, it is important to assure that the processor context is saved during execution of this routine. The context must be saved before you execute the routine itself, and it must be restored after the routine is finished. The procedure is called context switching. Context switching is also useful for subroutine calls, especially when extensive use is made of the auxiliary and the extended-precision registers. Code examples of context switching and an interrupt service routine are provided in this section.

**12.2.3.1 Context Switching**

Context switching is commonly required when processing a subroutine call or interrupt. It may be quite extensive or simple, depending on system requirements. For the 'C40, the program counter is automatically pushed onto the stack. Important information in other 'C40 registers, such as the status, auxiliary, or extended-precision registers must be saved by special commands. The status register should be saved first and restored last in order to preserve the processor status without any further change caused by other context-switching instructions.

Example 12–4 and Example 12–5 show saving and restoring of the 'C40 state. In both examples, the stack is used for saving the registers, and it expands towards higher addresses. If you don't want to use the stack pointed at by the SP, you can create a separate stack by using an auxiliary register as the stack pointer. Registers saved in these examples:

- ❑ Status register (ST) — should be saved first and restored last
- ❑ Extended-precision registers R0 through R11
- ❑ Auxiliary registers AR0 through AR7
- ❑ Data-page pointer (DP)
- ❑ Index registers (IR0 and IR1)
- ❑ Block-size register (BK)
- ❑ Interrupt-related registers IIE, IIF, and DIE
- ❑ Repeat-related registers RS, RE, and RC

Example 12-5. Context-Save for the TMS320C40

```

*      TITLE CONTEXT-SAVE FOR THE TMS320C40
*
*      .global      SAVE
*
*      CONTEXT SAVE ON SUBROUTINE CALL OR INTERRUPT.
*
SAVE:   PUSH      ST                ; Save status register
*
*      SAVE THE EXTENDED PRECISION REGISTERS
*
      PUSH      R0                ; Save the lower 32 bits of R0
      PUSHF    R0                ; and the upper 32 bits
      PUSH      R1                ; Save the lower 32 bits of R1
      PUSHF    R1                ; and the upper 32 bits
      PUSH      R2                ; Save the lower 32 bits of R2
      PUSHF    R2                ; and the upper 32 bits
      PUSH      R3                ; Save the lower 32 bits of R3
      PUSHF    R3                ; and the upper 32 bits
      PUSH      R4                ; Save the lower 32 bits of R4
      PUSHF    R4                ; and the upper 32 bits
      PUSH      R5                ; Save the lower 32 bits of R5
      PUSHF    R5                ; and the upper 32 bits
      PUSH      R6                ; Save the lower 32 bits of R6
      PUSHF    R6                ; and the upper 32 bits
      PUSH      R7                ; Save the lower 32 bits of R7
      PUSHF    R7                ; and the upper 32 bits
      PUSH      R8                ; Save the lower 32 bits of R8
      PUSHF    R8                ; and the upper 32 bits
      PUSH      R9                ; Save the lower 32 bits of R9
      PUSHF    R9                ; and the upper 32 bits
      PUSH      R10               ; Save the lower 32 bits of R10
      PUSHF    R10               ; and the upper 32 bits
      PUSH      R11               ; Save the lower 32 bits of R11
      PUSHF    R11               ; and the upper 32 bits
*
*      SAVE THE AUXILIARY REGISTERS
*
      PUSH      AR0               ; Save AR0
      PUSH      AR1               ; Save AR1
      PUSH      AR2               ; Save AR2
      PUSH      AR3               ; Save AR3
      PUSH      AR4               ; Save AR4
      PUSH      AR5               ; Save AR5
      PUSH      AR6               ; Save AR6
      PUSH      AR7               ; Save AR7
*
*      SAVE THE REST REGISTERS FROM THE REGISTER FILE
*
      PUSH      DP                ; Save data page pointer
      PUSH      IR0               ; Save index register IR0
      PUSH      IR1               ; Save index register IR1
      PUSH      BK                ; Save block-size register
      PUSH      IIE               ; Save interrupt enable register
      PUSH      IIF               ; Save interrupt flag register

```

```

PUSH      DIE          ; Save DMA interrupt enable register
PUSH      RS           ; Save repeat start address
PUSH      RE           ; Save repeat end address
PUSH      RC           ; Save repeat counter

```

```

*
*   SAVE IS COMPLETE
*

```

**Example 12-6. Context-Restore for the TMS320C40**

```

*
*   TITLE CONTEXT-RESTORE FOR THE TMS320C40
*
*   .global   RESTR
*
*   CONTEXT RESTORE AT THE END OF A SUBROUTINE CALL OR INTERRUPT.
RESTR:
*
*   RESTORE THE REST REGISTERS FROM THE REGISTER FILE
*
POP RC          ; Restore repeat counter
POP RE         ; Restore repeat end address
POP RS         ; Restore repeat start address
POP DIE        ; Restore DMA interrupt enable register
POP IIF        ; Restore interrupt flag register
POP IIE        ; Restore interrupt enable register
POP BK         ; Restore block-size register
POP IR1        ; Restore index register IR1
POP IR0        ; Restore index register IR0
POP DP         ; Restore data page pointer
*
*   RESTORE THE AUXILIARY REGISTERS
*
POP AR7        ; Restore AR7
POP AR6        ; Restore AR6
POP AR5        ; Restore AR5
POP AR4        ; Restore AR4
POP AR3        ; Restore AR3
POP AR2        ; Restore AR2
POP AR1        ; Restore AR1
POP AR0        ; Restore AR0
*
*   RESTORE THE EXTENDED PRECISION REGISTERS
*
POPF          R11      ; Restore the upper 32 bits and
POP           R11      ; the lower 32 bits of R11
POPF          R10      ; Restore the upper 32 bits and
POP           R10      ; the lower 32 bits of R10
POPF          R9       ; Restore the upper 32 bits and
POP           R9       ; the lower 32 bits of R9
POPF          R8       ; Restore the upper 32 bits and
POP           R8       ; the lower 32 bits of R8
POPF          R7       ; Restore the upper 32 bits and
POP           R7       ; the lower 32 bits of R7
POPF          R6       ; Restore the upper 32 bits and
POP           R6       ; the lower 32 bits of R6

```

```

POPF      R5          ; Restore the upper 32 bits and
POP       R5          ; the lower 32 bits of R5
POPF      R4          ; Restore the upper 32 bits and
POP       R4          ; the lower 32 bits of R4
POPF      R3          ; Restore the upper 32 bits and
POP       R3          ; the lower 32 bits of R3
POPF      R2          ; Restore the upper 32 bits and
POP       R2          ; the lower 32 bits of R2
POPF      R1          ; Restore the upper 32 bits and
POP       R1          ; the lower 32 bits of R1
POPF      R0          ; Restore the upper 32 bits and
POP       R0          ; the lower 32 bits of R0
POP ST    ; Restore status register

```

```

*
*   RESTORE IS COMPLETE
*

```

### 12.2.3.2 Interrupt-Vector Table

The interrupt-vector table (IVT, shown in Figure 3–8 on page 3-16) of the 'C40 is relocatable. The location of the IVT is relative to the interrupt-vector table pointer (IVTP). The IVTP is a 32-bit expansion register that points to the base address of the IVT. Since the IVT is required to lie on a 512-word boundary, the 9 LSBs of the IVTP should always be zero. The two instructions, LDEP and LDPE, read from and write to the expansion registers, IVTP and trap-vector table pointer (TVTP). Example 12–6 shows how to change the value of the IVTP (it is similar to changing the value of the TVTP). With this relocatable feature, an interrupt signal can be used for different services. In Example 12–7, the IVTP is reset in the external INT0 interrupt service routines EINT0A and EINT0B. After the value of the IVTP is changed, CPU will go to a different interrupt service routine when the same interrupt signal occurs again.

*Example 12-7. Use of One Interrupt Signal for Two Different Services*

```
*      TITLE USE OF ONE INTERRUPT SIGNAL FOR TWO DIFFERENT SERVICES
*
*      IN THIS EXAMPLE, THE ADDRESS OF EINT0A AND EINT0B ARE IN
*      MEMORY LOCATION 03H AND 1003H RESPECTIVELY. ASSUMING THE IVTP
*      HAS NOT BEEN CHANGED AFTER DEVICE RESET AND THE EXTERNAL
*      INTERRUPT IIOF0 IS ENABLED. WHEN THE FIRST IIOF0 INTERRUPT
*      SIGNAL COMES IN, THE EINT0A ROUTINE WILL BE EXECUTED. AND THEN
*      IF THE NEXT IIOF0 INTERRUPT SIGNAL OCCURS, THE EINT0B ROUTINE
*      WILL BE EXECUTED, AND SO ON. THE EINT0A AND EINT0B ROUTINES
*      WILL TAKE TURN TO BE EXECUTED WHEN IIOF0 INTERRUPT SIGNAL
*      OCCURS.
*
*      External IIOF0 interrupt service routine A
*
*      .global    EINT0A
EINT0A:  .
        .
        .
        LDI      1000H,R0          ; Change IVTP to point to 1000H
        LDPE    R0,IVTP
        .
*
        RETI                    ; Return and enable interrupts
*
*      External IIOF0 interrupt service routine A
*
*      .global    EINT0B
EINT0B:  .
        .
        .
        LDI      0,R0             ; Change IVTP to point to 0
        LDPE    R0,IVTP
        .
*
        RETI                    ; Return and enable interrupts
```

### 12.2.3.3 Interrupt Priority

Interrupts on the 'C40 are automatically prioritized. This allows interrupts that occur simultaneously to be serviced in a predefined order. Infrequent, but lengthy, interrupt service routines may need to be interrupted by more frequently occurring interrupts. Since the GIE bit in ST is reset when the interrupt vector is taken, this nesting interrupt will occur only if it is the NMI interrupt or if the interrupt is re-enabled in the interrupt service routine.

In Example 12–8, the interrupt service routine for INT2 temporarily modifies the interrupt enable register (IIE) and interrupt flag register (IIF) to permit interrupt processing when an interrupt to INTO or NMI (but no other interrupt) occurs. When the routine has finished processing, the IIE register is restored to its original state. Notice that the RETI*cond* instruction not only pops the next program counter address from the stack, but also restores GIE and CF bits from the PGIE and PCF bits. This re-enables all interrupts that were enabled before the INT2 interrupt was serviced.

*Example 12–8. Interrupt Service Routine*

```

*      TITLE INTERRUPT SERVICE ROUTINE
*      .global  ISR2
*
ENABLE  .set      2000h
MASK    .set      9h
*
*      INTERRUPT PROCESSING FOR EXTERNAL INTERRUPT INT2-
*
ISR2:
PUSH    ST          ; Save status register
PUSH    DP          ; Save data page pointer
PUSH    IIE         ; Save interrupt enable register
PUSH    IIF
PUSH    R0          ; Save lower 32 bits and
PUSHF   R0          ; upper 32 bits of R0
PUSH    R1          ; Save lower 32 bits and
PUSHF   R1          ; upper 32 bits of R1
LDI     0,IIE       ; Unmask all internal interrupts
LDI     MASK, R0
MH0     R0, IIF     : Enable INT2
OR      ENABLE,ST   ; Enable all interrupts
*
*      MAIN PROCESSING SECTION FOR ISR2
*      .
*      .
*
XOR     ENABLE,ST   ; Disable all interrupts
POPF   R1          ; Restore upper 32 bits and
POP     R1          ; lower 32 bits of R1
POPF   R0          ; Restore upper 32 bits and
POP     R0          ; lower 32 bits of R0
POP     IIF
POP     IIE         ; Restore interrupt enable register
POP     DP          ; Restore data page register
POP     ST          ; Restore status register
*
RETI                                ; Return and enable interrupts

```

## 12.2.4 Delayed Branches

The 'C40 uses delayed branches to create single-cycle branching. The delayed branches operate like regular branches but do not flush the pipeline. Instead, the three instructions following a delayed branch are also executed. Similarly, besides delayed branches, 'C40 also uses link and jump (LAJ), link and trap (LAT), delayed repeat block (RPTBD), and delayed return from interrupt or trap conditionally (RETI $cond$ D) instructions to avoid the pipeline flush (as discussed in Section 6.3 on page 6-9) in the *Program Flow Control* chapter (Chapter 6), the only limitations are that **none** of the three instructions following a delayed branch can be a:

- Branch (standard or delayed)
- Branch and annul conditionally
- Call to a subroutine
- Link and jump instruction
- Link and trap instruction
- Return from a subroutine
- Return from an interrupt or trap (standard or delayed)
- Repeat instruction (standard or delayed)
- TRAP instruction
- IDLE instruction

Conditional delayed branches use the conditions that exist at the end of the instruction immediately preceding the delayed branch. Sometimes, a branch is necessary in the flow of a program, but fewer than three instructions can be placed after a delayed branch. For faster execution, it is still advantageous to use a delayed branch. This is shown in Example 12-9, with a NOP taking the place of the third unused instruction. The tradeoff is more instruction words for less execution time.

**Example 12-9. Delayed Branch Execution**

```

*          TITLE DELAYED BRANCH EXECUTION
.
.
.
LDF        *+AR1(5),R2 ; Load contents of memory to R2
BGED       SKIP        ; If loaded number >=0, branch (delayed)
LDFN       R2,R1       ; If loaded number <0, load it to R1
SUBF       3.0,R1      ; Subtract 3 from R1
NOP        ; Dummy operation to complete delayed
*          ; branch
MPYF       1.5,R1      ; Continue here if loaded number <0
.
.
.
SKIP       LDF         R1,R3      ; Continue here if loaded number >=0

```

**12.2.5 Repeat Modes**

The 'C40 supports looping without any overhead. For that purpose, there are three instructions: RPTB and RPTBD repeat a block of code, and RPTS repeats a single instruction. The three control registers

- ❑ RS (Repeat Start address),
- ❑ RE (Repeat End address), and
- ❑ RC (Repeat Counter)

contain the parameters that specify loop execution (refer to Section 6.1 on page 6-2 for a description of RPTB, RPTBD, and RPTS). Registers RS and RE are automatically set from the code, while RC must be set by the user, as shown in Example 12-10.

**Example 12-10. Use of Block Repeat to Find a Maximum or a Minimum**

```

*
*          TITLE USE OF BLOCK REPEAT TO FIND A MAXIMUM OR A MINIMUM
*
*          THIS ROUTINE FINDS THE MAXIMUM OR THE MINIMUM OF N=147 NUMBERS
.
.
.
LDI        146,RC      ; Initialize repeat counter to 147-1
LDI        @ADDR,AR0   ; AR0 points to the beginning
              ; of the array
LDF        *AR0++(1),R0 ; Initialize MAX or MIN to the
              ; first value
BLT        LOOP2       ; If it is a negative array, find the
              ; minimum

```

```
*
LOOP1  RPTB      MAX
        CMPF     *ARO,R0      ; Compare number to the maximum
MAX     LDFLT    *ARO,R0      ; If greater, this is a new maximum
        B        NEXT
LOOP2  RPTB      MIN
        CMPF     *ARO++(1),RO ; Compare number to the minimum
MIN     LDFLT    *-ARO(1),RO  ; If smaller, this is a new minimum
NEXT   .
        .
        .
```

### **12.2.5.1 Block Repeat**

The 'C40 supports both standard and delayed repeat block instructions (RPTB and RPTBD). RPTB and RPTBD are the same except that the three instructions following RPTBD *are not* included in the loop (but *are* included in the RPTB loop). For RPTBD, the loop starts at the fourth instruction following RPTBD. The restriction of these three following instructions is the same as that of the three instructions following a delayed branch. Since RPTBD is a single-cycle instruction, it is very useful in making the nesting loop program more efficient. Example 12-10 shows the use of the block repeat to find the maximum or the minimum value of 147 numbers. The elements of the array are either all positive or all negative numbers. Since the loop cannot be predetermined, the RPTBD instruction is not suitable here.

### **12.2.5.2 Specifies Restrictions in the Block-Repeat Construct**

Because the program counter is modified at the end of the loop according to the contents of registers RS, RE, and RC, no operation should attempt to modify the repeat counter or the program counter at the end of the loop to a different value.

In principle, it is possible to nest repeat blocks. However, there is only one set of control registers: RS, RE, and RC. It is, therefore, necessary to save these registers before entering an inside loop and to restore these registers after completing the inside loop. It takes four cycles overhead to save and restore these registers. Hence, sometimes, it may be more economical to implement a nested loop by the more traditional method of using a register as a counter, and then using a delayed branch rather than by using the nested repeat block approach.



### 12.2.5.3 Single-Instruction Repeat

The single-instruction repeat uses control registers RS, RE, and RC in the same way as does the block repeat. The advantage over the block repeat is that the instruction is fetched only once, and then the buses are available for moving operands. One difference to note is that the single-instruction repeat construct is not interruptible, while block repeat is interruptible.

Example 12–12 shows an application of the repeat-single construct. In this example, the sum of the products of two arrays is computed. The arrays are not necessarily different. If the arrays are  $a(i)$  and  $b(i)$ , and if each is of length  $N=512$ , register R0 will contain, after computation, this quantity:

$$a(1) b(1) + a(2) b(2) + \dots + a(N) b(N).$$

The value of the repeat counter (RC) is specified to be 511 in the instruction.

#### Example 12–12. Loop Using Single Repeat

```
*          TITLE LOOP USING SINGLE REPEAT
*
*          .
*          .
*          LDI      @ADDR1,AR0          ; AR0 points to array a(i)
*          LDI      @ADDR2,AR1          ; AR1 points to array b(i)
*
*          LDF      0.0,R0              ; Initialize R0
*
*          MPYF3    *AR0++(1),*AR1++(1),R1 ; Compute first product
*
*          RPTS     511                  ; Repeat 512 times
*
*          MPYF3    *AR0++(1),*AR1++(1),R1 ; Compute next product and
*          ADDF3    R1,R0,R0             ; accumulate the
*                                     ; previous one
*
*          ADDF     R1,R0                ; One final addition
*          .
*          .
*          .
```

## 12.2.6 Computed GOTOs to Select Subroutines at Runtime

Occasionally, it is convenient to select during runtime, and not during assembly, the subroutine to be executed. The 'C40's computed GOTO supports this selection. The computed GOTO is implemented by using the *CALLcond* instruction in the register addressing mode. This instruction uses the contents of the register as the address of the call. Example 12-13 shows the case of a task controller.

### Example 12-13. Computed GOTO

```

*      TITLE   COMPUTED GOTO
*
*      TASK CONTROLLER
*
*      THIS MAIN ROUTINE CONTROLS THE ORDER OF TASK EXECUTION (6 TASKS
*      IN THE PRESENT EXAMPLE). TASK0 THROUGH TASK5 ARE THE NAMES OF
*      SUBROUTINES TO BE CALLED. THEY ARE EXECUTED IN ORDER, TASK0,
*      TASK1, . . . TASK5. WHEN AN INTERRUPT OCCURS, THE INTERRUPT
*      SERVICE ROUTINE IS EXECUTED, AND THE PROCESSOR CONTINUES
*      WITH THE INSTRUCTION FOLLOWING THE IDLE INSTRUCTION. THIS
*      ROUTINE SELECTS THE TASK APPROPRIATE FOR THE CURRENT CYCLE,
*      CALLS THE TASK AS A SUBROUTINE, AND BRANCHES BACK TO THE IDLE
*      TO WAIT FOR THE NEXT SAMPLE INTERRUPT WHEN THE SCHEDULED TASK
*      HAS COMPLETED EXECUTION. R0 HOLDS THE OFFSET FROM THE BASE
*      ADDRESS OF THE TASK TO BE EXECUTED. BIT 15 (SET COND BIT) OF
*      STATUS REGISTER (ST) SHOULD BE SET TO 1.
*
*
*      LDI      5,IR0           ; Initialize IR0
*      LDI      @ADDR,AR1      ; AR1 holds the base address
*                               ; of the table
*      WAIT    IDLE           ; Wait for the next interrupt
*      ADDI     *+AR1(IR0),R1  ; Add the base address to the table
*                               ; entry number
*      SUBI     1,IR0          ; Decrement IR0
*      LDILT    5,IR0          ; If IR0<0, reinitialize it to 5
*      CALLU    R1             ; Execute appropriate task
*      BR WAIT
*
*      TSKSEQ  .word    TASK5   ; Address of TASK5
*              .word    TASK4   ; Address of TASK4
*              .word    TASK3   ; Address of TASK3
*              .word    TASK2   ; Address of TASK2
*              .word    TASK1   ; Address of TASK1
*              .word    TASK0   ; Address of TASK0
*
*      ADDR   .word    TSKSEQ

```

## 12.3 Logical and Arithmetic Operations

The 'C40 instruction set supports both integer and floating-point arithmetic and logical operations. The basic functions of such instructions can be combined to form more complex operations. This section examines examples of these operations:

- Bit manipulation
- Block moves
- Byte and half-word manipulation
- Bit-reversed addressing
- Integer and floating-point division
- Square root
- Extended-precision arithmetic
- Floating-point format conversion between IEEE and 'C40 formats

### 12.3.1 Bit Manipulation

Instructions for logical operations, such as AND, OR, NOT, ANDN, and XOR, can be used together with the shift instructions for bit manipulation. A special instruction, TSTB, tests bits. TSTB does the same operation as AND, but the result of the TSTB is used only to set the condition flags and is not written anywhere. Example 12–14 and Example 12–15 demonstrate the use of the several instructions for bit manipulation and testing.

#### Example 12–14. Use of TSTB for Software-Controlled Interrupt

```
*      TITLE  USE OF TSTB FOR SOFTWARE-CONTROLLED INTERRUPT
*
*      IN THIS EXAMPLE, ALL INTERRUPTS HAVE BEEN DISABLED BY
*      RESETTING THE GIE BIT OF THE STATUS REGISTER. WHEN AN
*      INTERRUPT ARRIVES, IT IS STORED IN THE IF REGISTER. THE
*      PRESENT EXAMPLE ACTIVATES THE INTERRUPT SERVICE ROUTINE INTR
*      WHEN IT DETECTS THAT INT2- HAS OCCURRED.
*
*
*
TSTB      4,IIF      ; Check if bit 2 of IF is set,
CALLNZ    INTR      ; and, if so, call subroutine INTR
*
*
*

```

*Example 12–15. Copy a Bit from One Location to Another*

```

*          TITLE  COPY A BIT FROM ONE LOCATION TO ANOTHER
*
*          BIT I OF R1 NEEDS TO BE COPIED TO BIT J OF R2.
*          ARO POINTS TO A LOCATION HOLDING I, AND IT IS ASSUMED THAT THE
*          NEXT MEMORY LOCATION HOLDS THE VALUE J.
*
*
*          .
*          .
*          .
*          LDI      1,R0
*          LSH      *AR0,R0          ; Shift 1 to align it with bit I
*          TSTB     R1,R0            ; Test the I-th bit of R1
*          BZD      CONT            ; If bit = 0, branch delayed
*          LDI      1,R0
*          LSH      *+AR0(1),R0      ; Align 1 with J-th location
*          ANDN     R0,R2            ; If bit = 0, reset J-th bit of R2
*          OR       R0,R2            ; If bit = 1, set J-th bit of R2
*          CONT
*
*          .
*          .
*          .

```

### 12.3.2 Block Moves

Because the 'C40 directly addresses a large amount of memory, blocks of data or program code can be stored off-chip in slow memories and then loaded on-chip for faster execution. Data can also be moved from on-chip to off-chip for storage or for multiprocessor data transfers.

Such data transfers can be accomplished efficiently in parallel with CPU operations using the DMA. The DMA operation is explained in detail in Chapter 9. An alternative to DMA is to perform data transfers under program control by using load and store instructions in a repeat mode. Example 12–16 shows the transfer of a block of 512 floating-point numbers from external memory to block 1 of the on-chip RAM.

**Example 12-16. Block Move Under Program Control**

```
*          TITLE  BLOCK MOVE UNDER PROGRAM CONTROL
*
extern    .word 01000H
block1    .word 02FFC00H
          .
          .
          LDI    @extern,AR0      ; Source address
          LDI    @block1,AR1     ; Destination address

          LDF    *AR0++,R0       ; Load the first number

          RPTS   510              ; Repeat following instruction
                                ; 511 times
          LDF    *AR0++,R0       ; Load the next number, and...
||         STF    R0,*AR1++      ; store the previous one

          STF    R0,*AR1         ; Store the last number
          .
          .
```

### 12.3.3 Byte and Half-Word Manipulation

A new set of instructions for byte and half-word accessibility, such as LB(3,2,1,0), LBU(3,2,1,0), LH(1,0), LHU(1,0), LWL(0,1,2,3), LWR(0,1,2,3), MB(3,2,1,0), and MH(1,0), are available on the 'C40. In application such as image processing, it is often important to be able to manipulate packed data. For example, the pixels in color images are often represented by four 8-bit unsigned quantities — red, green, blue and alpha — which are packed into a single 32-bit word. The byte and half-word instruction will make it very easy to manipulate this packed data.

Example 12-17 shows the case of packing data from a half-word FIFO to 32-bit data memory, and Example 12-18 shows the case of unpacking a 32-bit data array into a four-byte-wide data array (assuming the 32-bit data array contains four 8-bit unsigned numbers).

**Example 12-17. Use of Packing Data From Half-Word FIFO to 32-Bit Data Memory**

```
*          TITLE  USE OF PACKING DATA FROM HALF-WORD FIFO
*          TO 32-BIT DATA MEMORY
*
*          IN THIS EXAMPLE, EVERY TWO INPUT 16 BITS DATA
*          HAS BEEN PACKED INTO ONE 32-BIT DATA MEMORY. THE LOOP SIZE
*          USED HERE IS ARRAY SIZE, NOT THE INPUT DATA LENGTH.
          .
          .
          .
```



accessing, the auxiliary register is indexed by IR0, but with reverse carry propagation. Example 12–19 illustrates a 512-point complex FFT being moved from the place of computation (pointed at by AR0) to a location pointed at by AR1. In this example, real and imaginary parts XR(i) and XI(i) of the data are not stored in separate arrays, but they are interleaved with XR(0), XI(0), XR(1), XI(1), ..., XR(N1), XI(N1). Because of this arrangement, the length of the array is 2N instead of N, and IR0 is set to 512 instead of 256.

*Example 12–19. Bit-Reversed Addressing*

```

*
*      TITLE BIT-REVERSED ADDRESSING
*
*      THIS EXAMPLE MOVES THE RESULT OF THE 512-POINT FFT
*      COMPUTATION, POINTED AT BY AR0, TO A LOCATION POINTED AT
*      BY AR1. REAL AND IMAGINARY POINTS ARE ALTERNATING.
*
*
*
*      LDI      512, IR0
*      RPTBD   LOOP
*      LDI      2, IR1
*      LDI      511, RC      ; Repeat 511+1 times
*      LDF      **AR0(1), R1 ; Load first imaginary point
*
*      LDF      *AR0++(IR0)B, R0 ; Load real value (and point
||      STF      R1, **AR1(1) ; to next location) and store
*      ; the imaginary value
*
LOOP    LDF      **AR0(1), R1 ; Load next imaginary point
        ; and store
||      STF      R0, *AR1++(IR1) ; previous real value
*
*
*

```

In DMA bit-reversed addressing, there are two bits in the DMA control register to enable bit-reversed addressing on DMA reads and DMA write. The source address index register and destination address index register are used to define the size of the bit-reversed addressing. Their function is similar to the CPU index register IR0. For more detail information about DMA operation, refer to Chapter 9.

## 12.3.5 Integer and Floating-Point Division

'C40 has a single-cycle instruction, RCPF, to generate an estimate of the reciprocal of a floating-point number. This estimate has the correct exponent, and the mantissa is accurate to the eighth binary place (the error of the mantissa is  $< 2^{-8}$ ). Often, this is a satisfactory estimate of the reciprocal of a floating-point number. In other cases, this estimate may be used as a seed for an algorithm that computes the reciprocal to even greater accuracy. The Newton-Raphson algorithm described later is one such case.

For integer division, although the special instruction is not provided, the instruction set has the capacity to perform an efficient division routine. Besides, the rough estimate can be achieved through FLOAT, RCPF, and FIX instructions.

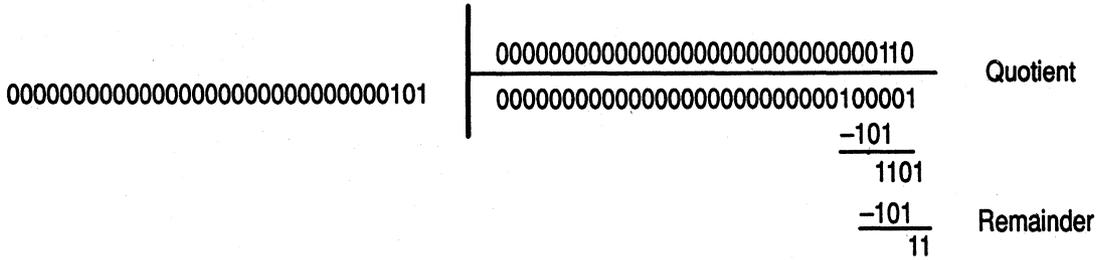
### 12.3.5.1 Integer Division

Division is implemented on the 'C40 by repeated subtractions using SUBC, a special conditional subtract instruction. Consider the case of a 32-bit positive dividend with  $i$  significant bits (and  $32-i$  sign bits), and a 32-bit positive divisor with  $j$  significant bits (and  $32-j$  sign bits). The repetition of the SUBC command  $i-j+1$  times produces a 32-bit result where the lower  $i-j+1$  bits are the quotient, and the upper  $31-i+j$  bits are the remainder of the division.

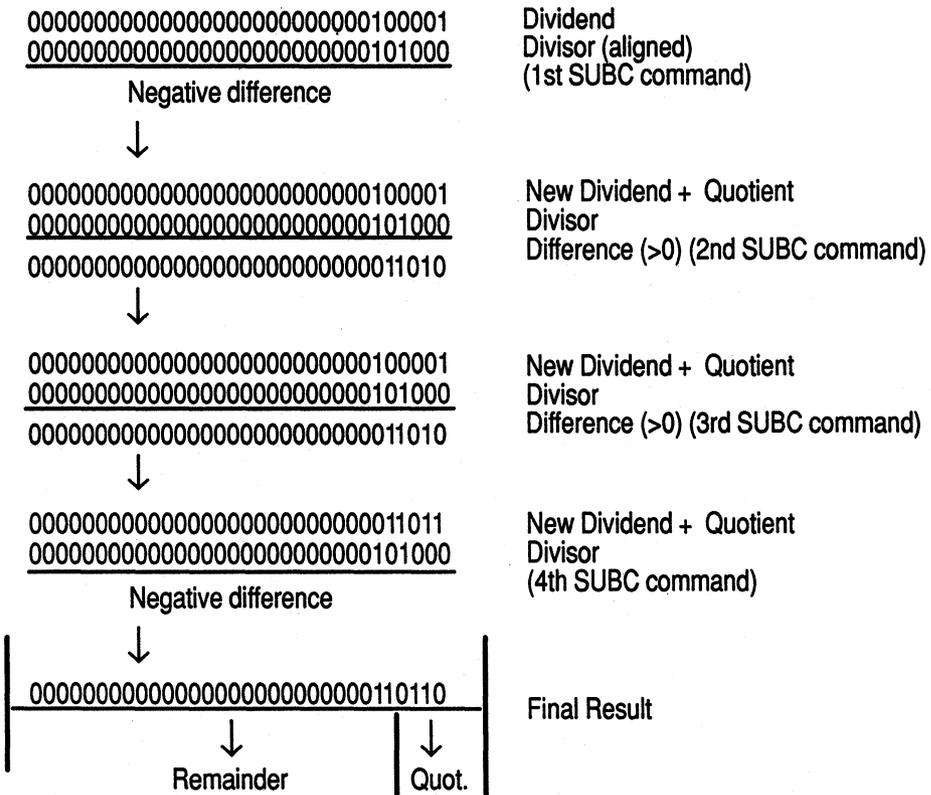
SUBC implements binary division in the same manner as long division. The divisor (assumed to be smaller than the dividend) is shifted left  $i-j$  times to be aligned with the dividend. Then, using SUBC, the shifted divisor is subtracted from the dividend. For each subtract that does not produce a negative answer, the dividend is replaced by the difference. It is then shifted to the left, and a one is put in the LSB. If the difference is negative, the dividend is simply shifted left by one. This operation is repeated  $i-j+1$  times.

As an example, consider the division of 33 by 5 using both long division and the SUBC method. In this case,  $i=6$ ,  $j=3$ , and the SUBC operation is repeated  $6-3+1=4$  times.

LONG DIVISION:



SUBC METHOD:



When the SUBC command is used, both the dividend and the divisor must be positive. Example 12-20 shows a realization of the integer division in which the sign of the quotient is properly handled. The last instruction before returning modifies the condition flag in case subsequent operations depend on the sign of the result.

Example 12-20. Integer Division

```

*
*   TITLE   INTEGER DIVISION
*
*   SUBROUTINE DIVI
*
*   INPUTS:  SIGNED INTEGER DIVIDEND IN R0,
*            SIGNED INTEGER DIVISOR IN R1.
*
*   OUTPUT:  R0/R1 into R0.
*
*   REGISTERS USED: R0-R3, IR0, IR1
*
*   OPERATION: 1. NORMALIZE DIVISOR WITH DIVIDEND
*              2. REPEAT SUBC
*              3. QUOTIENT IS IN LSBs OF RESULT
*
*   CYCLES: 31-62 (DEPENDS ON AMOUNT OF NORMALIZATION)
*
*   .globl  DIVI
*   SIGN    .set R2
*   TEMPF   .set R3
*   TEMP    .set IR0
*   COUNT   .set IR1
*
*   DIVI - SIGNED DIVISION
DIVI:
*
*   DETERMINE SIGN OF RESULT. GET ABSOLUTE VALUE OF OPERANDS.
*
*   XOR      R0,R1,SIGN      ; Get the sign
*   ABSI     R0
*   ABSI     R1
*   CMPI     R0,R1           ; Divisor > dividend ?
*   BGTD     ZERO           ; If so, return 0
*
*   NORMALIZE OPERANDS. USE DIFFERENCE IN EXPONENTS AS SHIFT COUNT
*   FOR DIVISOR, AND AS REPEAT COUNT FOR 'SUBC'.
*
*   FLOAT    R0,TEMPF       ; Normalize dividend
*   PUSHF    TEMPF          ; USH as float
*   POP      COUNT         ; POP as int
*   LSH      -24,COUNT      ; Get dividend exponent
*   FLOAT    R1,TEMPF       ; Normalize divisor
*   PUSHF    TEMPF          ; PUSH as float
*   POP      TEMP          ; POP as int
*   LSH      -24,TEMP       ; Get divisor exponent
*   SUBI     TEMP,COUNT     ; Get difference in exponents
*   LSH      COUNT,R1      ; Align divisor with dividend
*

```

```

*      DO COUNT+1 SUBTRACT & SHIFTS.
*
RPTS      COUNT
SUBC      R1,R0
*
*      MASK OFF THE LOWER COUNT+1 BITS OF R0
*
SUBRI     31,COUNT ; Shift count is (32 - (COUNT+1))
LSH      COUNT,R0 ; Shift left
NEGI     COUNT
LSH      COUNT,R0 ; Shift right to get result
*
*      CHECK SIGN AND NEGATE RESULT IF NECESSARY.
*
NEGI     R0,R1 ; Negate result
ASH      -31,SIGN ; Check sign
LDINZ   R1,R0 ; If set, use negative result
CMPI    0,R0 ; Set status from result RETS
*
*      RETURN ZERO.
*
ZERO:
LDI      0,R0
RETS
.end

```

If the dividend is less than the divisor and you want fractional division, you can perform a division after you determine the desired accuracy of the quotient in bits. If the desired accuracy is  $k$  bits, start by shifting the dividend left by  $k$  positions. Then apply the algorithm described above, where  $i$  should now be replaced by  $i + k$ . It is assumed that  $i + k$  is less than 32.

### **12.3.5.2 Computation of Floating-Point Inverse and Division**

This section presents a method of implementing a single-cycle RCPF instruction (reciprocal of a floating-point number) with an algorithm to extend the precision of the mantissa of the reciprocal of a floating-point number generated by RCPF instruction. The floating-point division can be obtained by multiplying the dividend and the reciprocal of the divisor.

The input to RCPF is assumed to be  $v - v(\text{man}) \times 2^{v(\text{exp})}$ . The output is  $x = x(\text{man}) \times 2^{x(\text{exp})}$ . The value  $v(\text{man})$  (or  $x(\text{man})$ ) is composed of three fields: the sign bit  $v(\text{sign})$ , an implied nonsign bit, and the fraction field  $v(\text{frac})$ .

The algorithm for RCPF uses these four rules:

- 1) If  $v > 0$ , then  $x(\text{exp}) = -v(\text{exp}) - 1$  and  $x(\text{man}) = 2/v(\text{man})$ .  
 For the special case where the ten MSBs of  $v(\text{man}) = 01.00000000b$ , then  $x(\text{man}) = 2 - 2^{-8} = 01.11111111b$ . In both cases, the 23 LSBs of  $x(\text{frac}) = 0$ .

- 2) If  $v < 0$ , then  $x(\text{exp}) = -v(\text{exp}) - 1$  and  $x(\text{man}) = 2/v(\text{man})$ .  
For the special case of the ten MSBs of  $v(\text{man}) = 10.0000000b$ , then  $x(\text{man}) = -1 - 2^{-8} = 10.1111111b$ . In both cases, the 23 LSBs of  $x(\text{frac}) = 0$ .
- 3) If  $v = 0$  ( $v(\text{exp}) = -128$ ), then  $x(\text{exp}) = 127$  and  $x(\text{man}) = 01.11111111111111111111111111111111b$ .  
In other words, if  $v = 0$ , then  $x$  becomes the largest positive number representable in the extended-precision floating-point format. The overflow flag (V) is set to 1.
- 4) If  $v(\text{exp}) = 127$ , then  $x(\text{exp}) = -128$  and  $x(\text{man}) = 0$ .  
The zero flag (Z) is set to 1.

The RCPF instruction gives an estimate of the reciprocal of a number. The Newton-Raphson algorithm may be used to further extend the precision of the mantissa. The algorithm is

$$x[n+1] = x[n](2.0 - vx[n])$$

$v$  is the number for which the reciprocal is desired.  $x[0]$  is the seed for the algorithm and is given by RCPF. At every iteration of the algorithm, the number of bits of accuracy in the mantissa doubles. Using RCPF, accuracy starts at eight bits. With one iteration, accuracy increases to 16 bits, and with the second iteration, accuracy increases to 32 bits in the mantissa. Example 12-21 shows the program to implement this algorithm on the 'C40.

*Example 12-21. Inverse of a Floating-Point Number With 32-Bit Mantissa Accuracy*

```

*
*   TITLE   INVERSE OF A FLOATING-POINT NUMBER
*           WITH 32-BIT MANTISSA ACCURACY
*
*   SUBROUTINE INVF
*
*   THE FLOATING-POINT NUMBER v IS STORED IN R0. AFTER THE
*   COMPUTATION IS COMPLETED, 1/v IS STORED IN R1.
*
*   TYPICAL CALLING SEQUENCE:
*   LAJU      INVF
*   LDF       v, R0
*   NOP      <---- can be other non-pipeline-break
*   NOP      <---- instructions
*
*   ARGUMENT ASSIGNMENTS:
*
*   ARGUMENT | FUNCTION
*   -----|-----
*   R0      | v = NUMBER TO FIND THE RECIPROCAL OF
*           | (UPON THE CALL)
*   R1      | 1/v (UPON THE RETURN)

```

```

*
* REGISTER USED AS INPUT: R0
* REGISTERS MODIFIED: R1, R2
* REGISTER CONTAINING RESULT: R1
* REGISTER USED FOR SUBROUTINE CALL: R11
*
* CYCLES: 8 WORDS: 8
*
* .global INVF
*
INVF: RCPF R0,R1 ; Get x[0] = the estimate of 1/v, R0 = v
*
MPYF3 R1,R0,R2
SUBRF 2.0,R2
MPYF R2,R1 ; End of first iteration
; (16 bits accuracy)
*
BUD R11 ; Delayed return to caller
*
MPYF3 R1,R0,R2
SUBRF 2.0,R2
MPYF R2,R1 ; End of second iteration
; (32 bits accuracy)
*
* R1 = 1/v, Return to caller
*
.end

```

### 12.3.6 Square Root

In many applications, normalization of data values is necessary. Often, the normalizing factor is the square root of another quantity. For example, given a vector, the unit vector in the same direction as the original vector can be found by normalizing the original vector by the length of the vector. This involves a division by a square root. The 'C40 provides a single-cycle instruction, RSQRF, to generate an estimate of the reciprocal of the square root of a positive floating-point number. This estimate has the correct exponent, and the mantissa is accurate to the eighth binary place (the error of the mantissa is  $< 2^{-8}$ ). Like the algorithm for RCPF, the algorithm for RSQRF uses these three rules:

- 1) If  $v(\text{exp})$  is even, then  $x(\text{exp}) = -(v(\text{exp})/2) - 1$  and  $x(\text{man}) = 2/\sqrt{v(\text{man})}$ . For the special case where the ten MSBs of  $y(\text{man}) = 01.00000000b$ , then  $x(\text{man}) = 2 - 2^{-8} = 01.11111111b$ . In both cases, the 23 LSBs of  $x(\text{frac}) = 0$ .





### 12.3.7 Extended-Precision Arithmetic

The TMS320C40 offers 32 bits of precision for integer arithmetic, and 24 bits of precision in the mantissa for floating-point arithmetic. For higher precision in floating-point operations, the twelve extended-precision registers R0 to R11 contain eight more bits of accuracy. Since no comparable extension is available for fixed-point arithmetic, this section discusses how fixed-point double precision can be achieved by using the capabilities of the processor. The technique consists of performing the arithmetic by parts and is similar to the way in which longhand arithmetic is done.

In the instruction set, operations ADDC (add with carry) and SUBB (subtract with borrow) use the status carry bit for extended-precision arithmetic. The carry bit is affected by the arithmetic operations of the ALU and by the rotate and shift instructions. It can also be manipulated directly by setting the status register to certain values. For proper operation, the overflow mode bit should be reset (OVM = 0) so that the accumulator results will not be loaded with the saturation values. Example 12–23 and Example 12–24 show 64-bit addition and 64-bit subtraction. The first operand is stored in the registers R0 (low word) and R1 (high word). The second operand is stored in R2 and R3, respectively. The result is stored in R0 and R1.

*Example 12–23. 64-Bit Addition*

```
*
*      TITLE 64-BIT ADDITION
*
*      TWO 64-BIT NUMBERS ARE ADDED TO EACH OTHER PRODUCING
*
*      A 64-BIT RESULT. THE NUMBERS X (R1,R0) AND Y (R3,R2)
*
*      ADDED, RESULTING IN W (R1,R0) .
*
*      R1   R0
*      + R3  R2
*      -----
*      R1   R0
*
*      ADDI   R2,R0
*      ADDC   R3,R1
```

**Example 12–24. 64-Bit Subtraction**

```

*
*      TITLE 64-BIT SUBTRACTION
*
*      TWO 64-BIT NUMBERS ARE SUBTRACTED FROM EACH OTHER
*      PRODUCING A 64-BIT RESULT. THE NUMBERS X (R1,R0) AND
*      Y (R3,R2) ARE SUBTRACTED, RESULTING IN W (R1,R0).
*
*      R1   R0
*      - R3  R2
*      -----
*      R1   R0
*
*      SUBI      R2,R0
*      SUBB     R3,R1
    
```

When two 32-bit numbers are multiplied, a 64-bit product results. To do this, 'C40 provides a 32 x 32-bit multiplier and two special instructions, MPYSHI (multiply signed integer and produce 32 MSBs) and MPYUHI (multiply unsigned integer and produce 32 MSBs). Example 12–25 shows the implementation of a 32-bit by 32-bit multiplication.

**Example 12–25. 32-Bit by 32-Bit Multiplication**

```

*
*      TITLE 32 x 32-BIT MULTIPLICATION
*
*      TWO 32-BIT NUMBERS ARE MULTIPLIED, PRODUCING A 64-BIT RESULT.
*      THE TWO NUMBERS X (R0) AND Y (R1) ARE MULTIPLIED, RESULTING
*      IN W (R3,R2).
*
*      R0
*      x R1
*      ----
*      R3  R2
*
*      MPYI3      R0,R1,R2
*      MPYSHI3   R0,R1,R3
    
```

**12.3.8 Floating-Point Format Conversion: IEEE to/from TMS320C40**

In fixed-point arithmetic, the binary point that separates the integer from the fractional part of the number is fixed at a certain location. For example, if a 32-bit number has the binary point after the most significant bit (which is also the sign bit), only fractional numbers (numbers with absolute values less than 1), can be represented. In other words, there is a number with 31 fractional bits called a Q31. All operations assume that the binary point is fixed at this location. The fixed-point system, although simple to implement in

hardware, imposes limitations in the dynamic range of the represented number. This causes scaling problems in many applications. You can avoid this difficulty by using floating-point numbers.

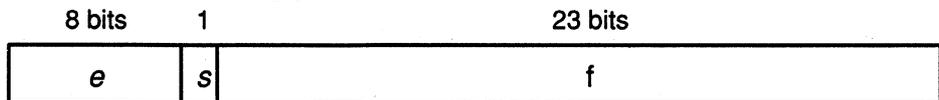
A floating-point number consists of a mantissa  $m$  multiplied by base  $b$  raised to an exponent  $e$ :

$$m * b^e$$

In current hardware implementations, the mantissa is typically a normalized number with an absolute value between 1 and 2, and the base is  $b = 2$ . Although the mantissa is represented as a fixed-point number, the actual value of the overall number floats the binary point because of the multiplication by  $b^e$ . The exponent  $e$  is an integer whose value determines the position of the binary point in the number. IEEE has established a standard format for the representation of floating-point numbers.

To achieve higher efficiency in the hardware implementation, the 'C40 uses a floating-point format that differs from the IEEE standard. However, 'C40 has two single-cycle instructions, TOIEEE and FRIEEE, for the format conversion. These two instructions can also be used with the STF instruction, which allows the data format to be converted within memory to memory transfer. This subsection describes briefly the two formats and presents an example program to convert between them.

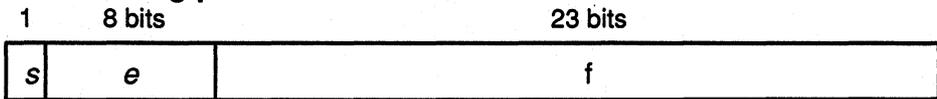
**TMS320C40 floating-point format:**



In a 32-bit word representing a floating-point number, the first 8 bits correspond to the exponent expressed in twos-complement format. One bit is for sign, and 23 bits are for the mantissa. The mantissa is expressed in twos-complement form with the binary point after the most significant nonsign bit. Since this bit is the complement of the sign bit  $s$ , it is suppressed. In other words, the mantissa actually has 24 bits. One special case occurs when  $e = -128$ . In this case, the number is interpreted as zero, independently of the values of  $s$  and  $f$  (which are by default set to zero). To summarize, the values of the represented numbers in the 'C40 floating-point format are as follows:

$$\begin{aligned}
 &2^e * (01.f) \text{ if } s = 0 \\
 &2^e * (10.f) \text{ if } s = 1 \\
 &0 \quad \quad \quad \text{if } e = -128
 \end{aligned}$$

**IEEE floating-point format:**



The IEEE floating-point format uses sign-magnitude notation for the mantissa, and offset by 127 for the exponent. In a 32-bit word representing a floating-point number, the first bit is the sign bit. The next 8 bits correspond to the exponent, expressed in an offset-by-127 format (the actual exponent is  $e-127$ ). The following 23 bits represent the absolute value of the mantissa with the most significant 1 implied. The binary point is after this most significant 1. In other words, the mantissa actually has 24 bits. There are several special cases, summarized below.

These are values of the represented numbers in the IEEE floating-point format:

$$(-1)^s * 2^{e-127} * (01.f) \quad \text{if } 0 < e < 255$$

Special cases:

$(-1)^s * 0.0$	if $e = 0$ and $f = 0$ (zero)
$(-1)^s * 2^{-126} * (0.f)$	if $e = 0$ and $f \neq 0$ (denormalized)
$(-1)^s * \text{infinity}$	if $e = 255$ and $f = 0$ (infinity)
NaN (not a number)	if $e = 255$ and $f \neq 0$

Based on these definitions of the formats, 'C40 has developed the hardware to do the conversion. It assumes that the source data for the IEEE format is in memory only and that for the 'C40 floating-point format, the source data is in either memory or an extended-precision register. The destination for both conversions must be in an extended-precision register. In the case of block memory transfer, the no penalty data format conversion can be achieved by parallel instruction with STF. Example 12-26 and Example 12-27 show the data format conversion within the data transformation between communication port and internal RAM.

**Example 12-26. IEEE to TMS320C40 Conversion Within Block Memory Transfer**

```

*      TITLE IEEE TO TMS320C40 CONVERSION WITHIN BLOCK MEMORY
*      TRANSFER
*
*      PROGRAM ASSUMES THAT THE INPUT FIFO OF COMMUNICATION PORT 0
*      ARE FULL OF IEEE FORMAT DATA. EIGHT DATA ARE TRANSFERRED FROM
*      COMMUNICATION PORT 0 TO INTERNAL RAM BLOCK 0 AND THE DATA
*      FORMAT ARE CONVERTED FROM IEEE FORMAT TO TMS320C40 FLOATING-
*      POINT FORMAT.
*
*
*      .
*      .
*      .
LDI      @CP0_IN,AR0    ; Load comm. port 0 input Fifo
                        ; address
LDI      @RAM0,AR1     ; Load internal RAM block 0 address
FRIEEE   *AR0,R0       ; Convert first data

RPTS     6
FRIEEE   *AR0,R0       ; Convert next data
|| STF   R0,*AR1++(1)  ; Store previous data
STF      R0,*AR1++(1) ; Store last data
*
*      .
*      .
*      .

```

**Example 12-27. TMS320C40 to IEEE Conversion Within Block Memory Transfer**

```

*      TITLE TMS320C40 TO IEEE CONVERSION WITHIN BLOCK MEMORY
*      TRANSFER
*
*      PROGRAM ASSUMES THAT THE OUTPUT FIFO OF COMMUNICATION PORT 0
*      IS EMPTY. EIGHT DATA ARE TRANSFERRED FROM INTERNAL RAM BLOCK 0
*      TO COMMUNICATION PORT 0 AND THE DATA FORMAT ARE CONVERTED FROM
*      TMS320C40 FLOATING-POINT FORMAT TO IEEE FORMAT.
*
*
*      .
*      .
*      .
LDI      @CP0_OUT,AR0  ; Load comm. port 0 output Fifo address
LDI      @RAM0,AR1    ; Load internal RAM block 0 address
TOIEEE   *AR1++(1),R0 ; Convert first data

RPTS     6
TDIEEE   *AR1++(1),R0 ; Convert next data
|| STF   R0,*AR0      ; Store previous data
STFR0,   *AR0         ; Store last data
*
*      .
*      .
*      .

```

## 12.4 Application-Oriented Operations

Certain features of the 'C40 architecture and instruction set facilitate the solution of numerically intensive problems. This section presents examples of applications that use these features, such as companding, filtering, matrix arithmetic, and fast Fourier transforms (FFT).

### 12.4.1 Companding

In the area of telecommunications, one of the primary concerns is to conserve the channel bandwidth and, at the same time, to preserve high speech quality. This is achieved by quantizing the speech samples logarithmically. It has been demonstrated that an 8-bit logarithmic quantizer produces speech quality equivalent to a 13-bit uniform quantizer. The logarithmic quantization is achieved by companding (COMpress/exPANDING). Two international standards have been established for companding: the  $\mu$ -law (used in the United States and Japan), and the A-law (used in Europe). Detailed descriptions of  $\mu$ -law and A-law companding are presented in an application report on companding routines included in the book *Digital Signal Processing Applications with the TMS320 Family* (literature number SPRA012A).

During transmission, logarithmically compressed data in sign-magnitude form are transmitted along the communications channel. If any processing is necessary, these data should be expanded to a 14-bit (for  $\mu$ -law) or 13-bit (for A-law) linear format. This operation occurs when data is received at the digital signal processor. After processing, and in order to continue transmission, the result is compressed back to 8-bit format and transmitted through the channel.

Example 12-28 and Example 12-29 show  $\mu$ -law compression and expansion (i.e., linear to  $\mu$ -law and  $\mu$ -law to linear conversion), while Example 12-30 and Example 12-31 show A-law compression and expansion. For expansion, using a look-up table is an alternative approach. It trades memory space for speed of execution. Since the compressed data is 8 bits long, a table with 256 entries can be constructed to contain the expanded data. If the compressed data is stored in the register AR0, the following two instructions will put the expanded data in register R0:

```
ADDI    @TABL,AR0    ; @TABL = BASE ADDRESS OF TABLE
LDI     *AR0,R0      ; PUT EXPANDED NUMBER IN R0
```

The same look-up table approach could be used for compression, but the required table length would then be 16,384 words for  $\mu$ -law or 8,192 words for A-law. If this memory size is not acceptable, the subroutines presented in Example 12-28 or Example 12-30 should be used.

Example 12-28.  $\mu$ -Law Compression

```

*
*      TITLE  $\mu$ -LAW COMPRESSION
*
*      SUBROUTINE MUCMPR
*
*      TYPICAL CALLING SEQUENCE:
*      LAJU      MUCMPR
*      LDI       v, R0
*      NOP       <----      can be other non-pipeline-break
*      NOP       <----      instructions
*
*      ARGUMENT ASSIGNMENTS:
*
*      ARGUMENT | FUNCTION
*      -----+-----
*      R0       | v = NUMBER TO BE CONVERTED
*
*      REGISTERS USED AS INPUT: R0
*      REGISTERS MODIFIED: R0, R1
*      REGISTER CONTAINING RESULT: R0
*
*      CYCLES: 15          WORDS: 15
*
*      .global MUCMPR
*
MUCMPR  LSH3      -6,R0,R1      ; Save sign of number
        ABSI      R0,R0
        CMPI     1FDEH,R0      ; If R0>0x1FDE,
        LDIGT    1FDEH,R0      ; saturate the result
        ADDI     33,R0         ; Add bias
        FLOAT    R0           ; Normalize: (seg+5)0WXYZx...x
        MPYF     0.03125,R0    ; Adjust segment number by 2**(-5)
        LSH      1,R0         ; (seg)WXYZx...x
        PUSHF    R0
        POP      R0           ; Treat number as integer
        LSH      -20,R0       ; Right-justify
        BUD      R11          ; Delayed return
        AND      080H,R1      ; Set sign bit
        ADDI     R1,R0         ; R0 = compressed number
        NOT      R0           ; Reverse all bits for transmission

```

Example 12-29.  $\mu$ -Law Expansion

```

*
*TITLE 'μ-LAW EXPANSION'
*
*      SUBROUTINE MUXPND
*
*      TYPICAL CALLING SEQUENCE:
*      LAJU      MUXPND
*      LDI       v, R0
*      NOP      <---- can be other non-pipeline-break
*      NOP      <---- instructions
*
*      ARGUMENT ASSIGNMENTS:
*
*          ARGUMENT|  FUNCTION
*          -----+-----
*          R0      |  v = NUMBER TO BE CONVERTED
*
*      REGISTERS USED AS INPUT: R0
*      REGISTERS MODIFIED: R0, R1, R2
*      REGISTER  CONTAINING RESULT: R0
*
*      CYCLES: 14 (WORST CASE)   WORDS: 14
*
*      .global  MUXPND
*
MUXPND  NOT      R0,R0      ; Complement bits
        AND3    0FH,R0,R1  ; Isolate quantization bin
        LSH     1,R1
        ADDI    33,R1      ; Add bias to introduce 1xxxx1
        LSH3    -4,R0      ; Isolate segment code
        TSTB    08H,R0    ; Test sign
        BZD     R11        ; if positive, delayed return
        AND     7,R0
        LSH3    R0,R1,R0   ; Shift and put result in R0
        SUBI    33,R0      ; Subtract bias
        BUD     R11        ; Delayed return
        NEGI    R0        ; Negate if a negative number
        NOP
        NOP
    
```

## Example 12-30. A-Law Compression

```

*
*      TITLE A-LAW COMPRESSION
*
*      SUBROUTINE ACMPR
*
*      TYPICAL CALLING SEQUENCE:
*      LAJACMPR
*      LDIV, R0
*      NOP<---- can be other non-pipeline-break
*      NOP<---- instructions
*
*      ARGUMENT ASSIGNMENTS:
*      ARGUMENT | FUNCTION
*      -----+-----
*      R0       | v = NUMBER TO BE CONVERTED
*
*      REGISTERS USED AS INPUT: R0
*      REGISTERS MODIFIED: R0, R1
*      REGISTER CONTAINING RESULT: R0
*
*      CYCLES:17      WORDS: 17
*
*      .global  ACMPR
*
ACMPR   LSH3      -5,R0,R1      ; Save sign of number
        ABSI      R0,R0
        CMPI     1FH,R0       ; If R0<0x20,
        BLED     END          ; Do linear coding
        CMPI     0FFFH,R0     ; If R0>0xFFF,
        LDIGT    0FFFH,R0     ; saturate the result
        LSH      -1,R0        ; Eliminate rightmost bit

        FLOAT    R0          ; Normalize: (seg+3)0WXYZx...x
        MPYF     0.125,R0     ; Adjust segment number by 2**(-3)
        LSH      1,R0         ; (seg)WXYZx...x
        PUSH     FR0
        POP      R0           ; Treat number as integer
        LSH      -20,R0       ; Right-justify

END     BUD      R11          ; Delayed return
        AND      080H,R1     ; Set sign bit
        ADDI     R1,R0        ; R0 = compressed number
        XOR      0D5H,R0     ; Invert even bits for
                               ; transmission
*

```

Example 12-31. A-Law Expansion

```

*
*      TITLE A-LAW EXPANSION
*
*      SUBROUTINE AXPND
*
*      TYPICAL CALLING SEQUENCE:
*      LAJU      AXPND
*      LDI       v, R0
*      NOP      <----      can be other non-pipeline-break
*      NOP      <----      instructions
*
*      ARGUMENT ASSIGNMENTS:
*
*      ARGUMENT | FUNCTION
*      -----+-----
*      R0       | v = NUMBER TO BE CONVERTED
*
*      REGISTERS USED AS INPUT: R0
*      REGISTERS MODIFIED: R0, R1, R2
*      REGISTER CONTAINING RESULT: R0
*
*      CYCLES: 19 (WORST CASE)   WORDS: 16
*
*      .global  AXPND
*
AXPND  XOR      0D5H,R0,R2      ; Invert even bits
      ASH3     -4,R2,R0        ; Store for bit sign
      AND      7,R0            ; Isolate segment code
      BZD      SKIP1
      AND3     0FH,R2,R1      ; Isolate quantization bin
      LSH     1,R1
      ADDI     1,R1            ; Create 0xxxx1
      ADDI     32,R1           ; Or 1xxxx1
      SUBI     1,R0
SKIP1  LSH3     R0,R1,R0        ; Shift and put result in R0
      TSTB     80H,R2          ; Test sign bit
      BZAT     R11             ; If positive, delayed return and
      ; annul next three instructions
      NEGI     R0              ; Negate if a negative number
      NOP
      NOP
      BU      R11              ; Return

```

## 12.4.2 FIR, IIR, and Adaptive Filters

Digital filters are a common requirement for digital signal processing systems. There are two types of digital filters: finite impulse response (FIR) and infinite impulse response (IIR). Each of these types can have either fixed or adaptable coefficients. In this section, the fixed-coefficient filters are presented first, and then the adaptive filters are discussed.

### 12.4.2.1 FIR Filters

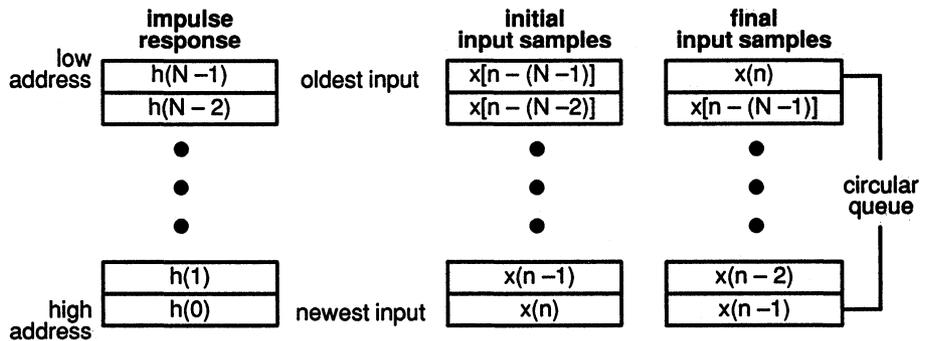
If the FIR filter has an impulse response  $h[0], h[1], \dots, h[N-1]$ , and  $x[n]$  represents the input of the filter at time  $n$ , the output  $y[n]$  at time  $n$  is given by this equation:

$$y[n] = h[0] x[n] + h[1] x[n-1] + \dots + h[N-1] x[n-(N-1)]$$

Two features of the 'C40 that facilitate the implementation of the FIR filters are parallel multiply/add operations and circular addressing. The first one permits the performance of a multiplication and an addition in a single machine cycle, while the second one makes a finite buffer of length  $N$  sufficient for the data  $x$ .

Figure 12-1 shows the arrangement of the memory locations in order to implement circular addressing, while Example 12-32 presents the 'C40 assembly code for an FIR filter.

Figure 12-1. Data Memory Organization for an FIR Filter



In order to set up circular addressing, initialize the block-size register BK to block length N. Also, the locations for signal x should start from a memory location whose address is a multiple of the smallest power of 2 that is greater than N. For instance, if N = 24, the first address for x should be a multiple of 32 (the lower 5 bits of the beginning address should be zero). To understand this requirement, look at Section 5.3 on page 5-25, *Circular Addressing*.

In Example 12-32, the pointer to the input sequence x is incremented and assumed to be moving from an older input to a newer input. At the end of the subroutine, AR1 will point to the position for the next input sample.

**Example 12-32. FIR Filter**

```

*
*      TITLE FIR FILTER
*
*
*      SUBROUTINE FIR
*
*      EQUATION:  $y(n) = h(0) * x(n) + h(1) * x(n-1) +$ 
*                 $\dots + h(N-1) * x(n-(N-1))$ 
*
*      TYPICAL CALLING SEQUENCE:
*
*      LOAD      AR0
*      LAJU     FIR
*      LOAD     AR1
*      LOAD     RC
*      LOAD     BK
*
*      ARGUMENT ASSIGNMENTS:
*
*      ARGUMENT | FUNCTION
*      -----+-----
*      AR0      | ADDRESS OF h(N-1)
*      AR1      | ADDRESS OF x(N-1)
*      RC       | LENGTH OF FILTER - 2 (N-2)
*      BK       | LENGTH OF FILTER (N)
*
*      REGISTERS USED AS INPUT: AR0, AR1, RC, BK
*      REGISTERS MODIFIED: R0, R2, AR0, AR1, RC
*      REGISTER CONTAINING RESULT: R0
*

```

```

*
*      CYCLES: 7 + N      WORDS: 9
*
*
FIR      .global  FIR
*
      RPTBD      CONV      ; Setup the repeat cycle.
*                      ; Initialize R0:
      MPYF3      *AR0++(1), *AR1++(1)%, R0 ; h(N-1) *x(n-(N-1)) ->R0
      LDF        0.0, R2      ; Initialize R2.
      NOP
*
*      FILTER      (1 <= i < N)
*
CONV     MPYF3      *AR0++(1), *AR1++(1)%, R0 ; h(N-1-i)*x(n-(N-1-i))->R0
||      ADDF3      R0, R2, R2      ; Multiply and add operation
*
      BUD        R11      ; Delayed return
      ADDF       R0, R2, R0      ; Add last product
      NOP
      NOP
*
*      end
*
      .end

```

### 12.4.2.2 IIR Filters

The transfer function of the IIR filters has both poles and zeros. Its output depends on both the input and the past output. As a rule, the filters need less computation than an FIR with similar frequency response, but the filters have the drawback of being sensitive to coefficient quantization. Most often, the IIR filters are implemented as a cascade of second-order sections, called biquads. Example 12-33 and Example 12-34 show the implementation for one biquad and for any number of biquads, respectively.

$$y[n] = a_1 y[n-1] + a_2 y[n-2] + b_0 x[n] + b_1 x[n-1] + b_2 x[n-2]$$

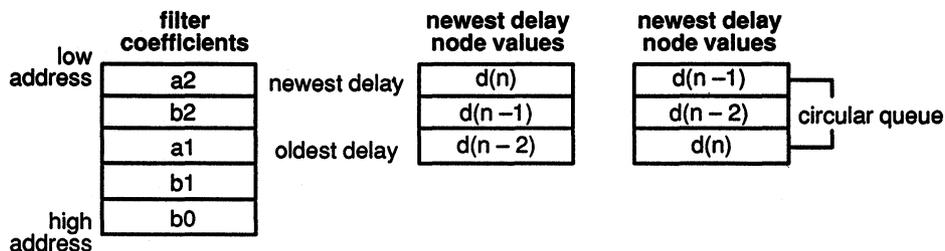
However, the following two equations are more convenient and have smaller storage requirements:

$$d[n] = a2 d[n-2] + a1 d[n-1] + x[n]$$

$$y[n] = b2 d[n-2] + b1 d[n-1] + b0 d[n]$$

Figure 12-2 shows the memory organization for this two-equation approach, an implementation of a single biquad on the 'C40.

Figure 12-2. Data Memory Organization for a Single Biquad



As in the case of FIR filters, the address for the start of the values  $d$  must be a multiple of 4; i.e., the last two bits of the beginning address must be zero. The block-size register BK must be initialized to 3.

Example 12-33. IIR Filter (One Biquad)

```
*      TITLE IIR filter
*
*      SUBROUTINE IIR1
*
*      IIR1 == IIR FILTER (ONE BIQUAD)
*
*      EQUATIONS: d(n) = a2 * d(n-2) + a1 * d(n-1) + x(n)
*                  y(n) = b2 * d(n-2) + b1 * d(n-1) + b0 * d(n)
*
*      OR          y(n) = a1*y(n-1) + a2*y(n-2) + b0*x(n) + b1*x(n-1)
*                  + b2*x(n-2)
*
*
*      TYPICAL CALLING SEQUENCE:
*
*      load      R2
*      LAJU     IIR1
*      load     AR0
*      load     AR1
*      load     BK
```

```

*
* ARGUMENT ASSIGNMENTS:
* ARGUMENT | FUNCTION
* -----+-----
* R2      | INPUT SAMPLE X(N)
* AR0     | ADDRESS OF FILTER COEFFICIENTS (A2)
* AR1     | ADDRESS OF DELAY MODE VALUES (D(N-2))
* BK      | BK = 3
*
* REGISTERS USED AS INPUT:  R2, AR0, AR1, BK
* REGISTERS MODIFIED:      R0, R1, R2, AR0, AR1
* REGISTER CONTAINING RESULT:  R0
*
* CYCLES: 8 WORDS: 8
*
* .global IIR1
IIR1
MPYF3    *AR0,*AR1,R0      ; a2 * d(n-2) -> R0
MPYF3    *++AR0(1),*AR1--(1)%,R1 ; b2 * d(n-2) -> R1
*
MPYF3    *++AR0(1),*AR1,R0    ; a1 * d(n-1) -> R0
|| ADDF3  R0,R2,R2            ; a2*d(n-2)+x(n) -> R2
*
MPYF3    *++AR0(1),*AR1--(1)%,R0 ; b1 * d(n-1) -> R0
|| ADDF3  R0,R2,R2            ; a1*d(n-1)+a2*d(n-2)
*                               ; +x(n) -> R2
*
BUD      R11                ; Delayed return
*
MPYF3    *++AR0(1),R2,R2      ; b0 * d(n) -> R2
|| STF    R2,*AR1++(1)%      ; Store d(n) and point to
*                               ; d(n-1)
*
ADDF     R0,R2                ; b1*d(n-1)+b0*d(n) -> R2
ADDF     R1,R2,R0             ; b2*d(n-2)+b1*d(n-1)
*                               ; +b0*d(n) -> R0
*
end
*
* .end

```

In the more general case, the IIR filter contains  $N > 1$  biquads. The equations for its implementation are given by the following pseudo-C language code:

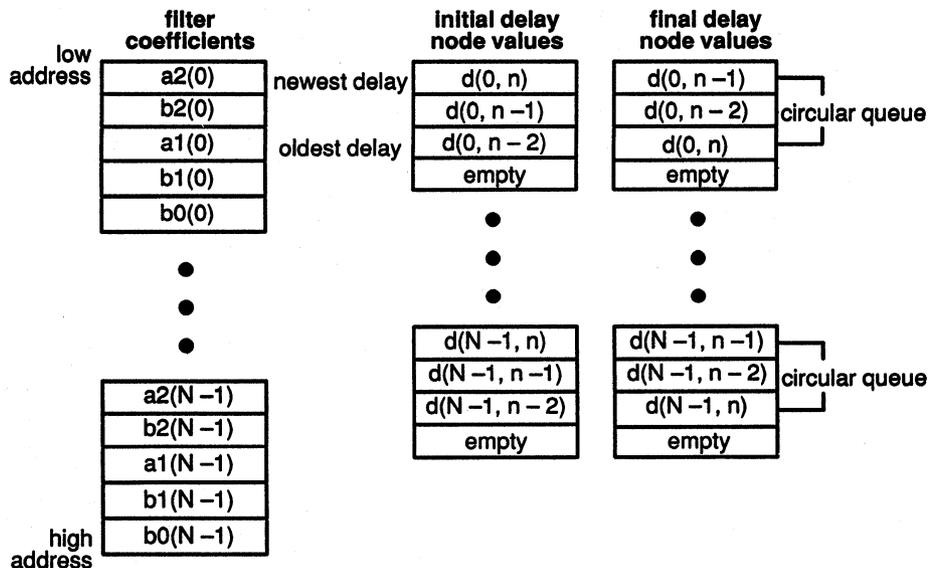
```

y[0,n] = x[n]
for (i=0; i<N; i++){
    d[i,n] = a2[i] d[i,n-2] + a1[i] d[i,n-1] + y[i-1,n]
    y[i,n] = b2[i] d[i-2] + b1[i] d[i,n-1] + b0[i] d[i,n]
}
y[n] = y[N-1,n]

```

Figure 12-3 shows the corresponding memory organization, while Example 12-34 shows the 'C40 assembly-language code.

Figure 12-3. Data Memory Organization for N Biquads



The block size register BK should be initialized to 3, and the beginning of each set of d values (i.e.,  $d[i, n]$ ,  $i = 0 \dots N-1$ ) should be at an address that is a multiple of 4 (the last two bits zero), as stated in the case of a single biquad.

Example 12-34. IIR Filters ( $N > 1$  Biquads)

```
*
* TITLE IIR FILTERS (N > BIQUADS)
*
* SUBROUTINE IIR2
*
* EQUATIONS: y(0, n) = x(n)
*
* FOR (i = 0; i < N; i++)
* {
* d(i, n) = a2(i) * d(i, n-2) + a1(i) * d(i, n-1) * y(i-1, n)
* y(i, n) = b2(i) * d(i, n-2) + b1(i) * d(i, n-1) * b0(i) * d(i, n)
* }
* y(n) = y(N-1, n)
*
* TYPICAL CALLING SEQUENCE:
*
* load R2
* load AR0
* load AR1
* load IRO
```



```

||      ADDF3      R0,R2,R2                ; First sum term
                                           ; of d(i,n).
*
MPYF3   *++AR0(1),*AR1-(1)%,R0          ; b1(i) * d(i,n-1) -> R0
||      ADDF3      R0,R2,R2                ; Secondsumterm
                                           ; of d(i,n).
*
LOOP    MPYF3      *++AR0(1),R2,R2        ; b0(i) * d(i,n) -> R2
||      STF        R2,*AR1-(1)%           ; Store d(i,n)
                                           ; point to d(i,n-2)
*
*      FINAL SUMMATION
*
BRD     R11                ; Delayed return
*
ADDF    R0,R2              ; First sum term
                                           ; of y(n-1,n)
ADDF3   R1,R2,R0          ; Second sum term of
                                           ; y(n-1,n)
LDI     *AR1--(IR1),R1     ; Return to first biquad
||      LDI        *AR1--(1)%,R2         ; Point to d(0,n-1)
*
*      end
*
      .end

```

### 12.4.2.3 Adaptive Filters (LMS Algorithm)

In some applications in digital signal processing, a filter must be adapted over time to keep track of changing conditions. The book *Theory and Design of Adaptive Filters* by Treichler, Johnson, and Larimore (Wiley-Interscience, 1987) presents the theory of adaptive filters. Although in theory, both FIR and IIR structures can be used as adaptive filters, the stability problems and the local optimum points that the IIR filters exhibit make them less attractive for such an application. Hence, until further research makes IIR filters a better choice, only the FIR filters are used in adaptive algorithms of practical applications.

In an adaptive FIR filter, the filtering equation takes this form:

$$y[n] = h[n,0]x[n] + h[n,1]x[n-1] + \dots + h[n,N-1]x[n-(N-1)]$$

The filter coefficients are time-dependent. In a least-mean-squares (LMS) algorithm, the coefficients are updated by an equation in this form:

$$h[n+1,i] = h[n,i] + b x[n-i], i = 0, 1, \dots, N-1$$

$b$  is a constant for the computation. The updating of the filter coefficients can be interleaved with the computation of the filter output so that it takes 3 cycles per filter tap to do both. The updated coefficients are written over the old filter coefficients. Example 12–35 shows the implementation of an adaptive FIR filter on the 'C40. The memory organization and the positioning of the data in memory should follow the same rules as the above FIR filter with fixed coefficients.

*Example 12–35. Adaptive FIR Filter (LMS Algorithm)*

```

*      TITLE ADAPTIVE FIR FILTER (LMS ALGORITHM)
*
*      SUBROUTINE LMS
*
*      LMS == LMS ADAPTIVE FILTER
*
*      EQUATIONS:   y(n) = h(n,0)*x(n) + h(n,1)*x(n-1) + ...
*                  + h(n,N-1)*x(n-(N-1))
*      FOR          (i = 0; i < N; i++) h(n+1,i) = h(n,i)
*                  + tmuerr * x(n-i)
*
*      TYPICAL CALLING SEQUENCE:
*
*      load      R4
*      load      ARO
*      LAJU      LMS
*      load      AR1
*      load      RC
*      load      BK
*
*
*      ARGUMENT ASSIGNMENTS:
*      ARGUMENT | FUNCTION
*      -----+-----
*      R4       | SCALE FACTOR (2 * mu * err)
*      ARO      | ADDRESS OF h(n,N-1)
*      AR1      | ADDRESS OF x(n-(N-1))
*      RC       | LENGTH OF FILTER - 2 (N-2)
*      BK       | LENGTH OF FILTER (N)
*
*
*      REGISTERS USED AS INPUT: R4, ARO, AR1, RC, BK
*      REGISTERS MODIFIED: R0, R1, R2, ARO, AR1, RC
*      REGISTER CONTAINING RESULT: R0
*
*      PROGRAM SIZE: 12 words
*
*      EXECUTION CYCLES: 6 + 3N
*

```

```

*      SETUP (i = 0)
*
      .global  LMS
LMS    RPTBD   LOOP           ; Setup the delayed repeat block.
*      ; Initialize R0:
      MPYF3   *AR0,*AR1,R0    ; h(n,N-1) * x(n-(N-1)) -> R0
||     SUBF3   R2, R2, R2     ; Initialize R2
*
*      ; Initialize R1:
      MPYF3   *AR1++(1),R4,R1 ; x(n-(N-1)) * tmuerr -> R1
      ADDF3   *AR0++(1),R1,R1 ; h(n,N-1) + x(n-(N-1)) *
*      ; tmuerr -> R1
*
*      FILTER AND UPDATE (1 <= I < N)
*      ; Filter:
      MPYF3   *AR0--(1),*AR1,R0 ; h(n,N-1-i) * x(n-(N-1-i)) -> R0
||     ADDF3   R0,R2,R2      ; Multiply and add operation.
*
*      ; UPDATE:
      MPYF3   *AR1++(1),R4,R1 ; x(n,N-(N-1-i)) * tmuerr -> R1
||     STF     R1,*AR0++(1)   ; R1 -> h(n+1,N-1-(i-1))
*
LOOP   ADDF3   *AR0++(1),R1,R1 ; h(n,N-1-i) + x(n-(N-1-i))
*      ; *tmuerr -> R1
*
*      BUD     R11           ; Delayed return
*
      ADDF3   R0,R2,R0       ; Add last product.
      STF     R1,*-AR0(1)    ; h(n,0) + x(n* tmuerr ->
*      ; h(n+1 ,0)
*
      NOP
*
*      end
*
      .end

```

### 12.4.3 Matrix-Vector Multiplication

In matrix-vector multiplication, a  $K \times N$  matrix of elements  $m(i,j)$ , having  $K$  rows and  $N$  columns, is multiplied by an  $N \times 1$  vector to produce a  $K \times 1$  result. The multiplier vector has elements  $v(j)$ , and the product vector has elements  $p(i)$ . Each one of the product-vector elements is computed by the following expression:

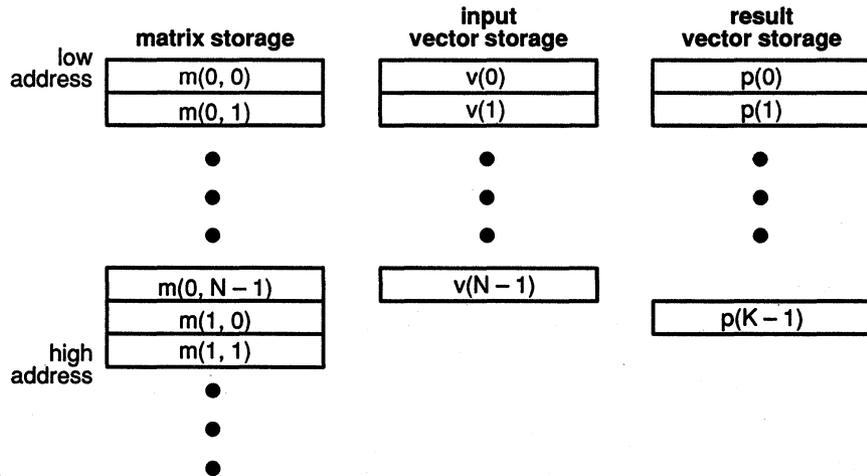
$$p(i) = m(i,0) v(0) + m(i,1) v(1) + \dots + m(i,N-1) v(N-1) \quad i = 0, 1, \dots, K-1$$

This is essentially a dot product, and the matrix-vector multiplication contains, as a special case, the dot product presented in Example 12-2 on page 12-10 and Example 12-3 on page 12-12. In pseudo-C format, the computation of the matrix multiplication is expressed by

```
for (i = 0; i < K; i++) {
    p(i) = 0
    for (j = 0; j < N; j++)
        p(i) = p(i) + m(i,j) * v(j)
}
```

Figure 12-4 shows the data memory organization for matrix-vector multiplication, and Example 12-36 shows the 'C40 assembly code to implement it. Note that in Example 12-36,  $K$  (number of rows) should be greater than 0, and  $N$  (number of columns) should be greater than 1.

Figure 12-4. Data Memory Organization for Matrix-Vector Multiplication



**Example 12-36. Matrix Times a Vector Multiplication**

```

*
*      TITLE MATRIX TIMES A VECTOR MULTIPLICATION
*
*      SUBROUTINE MAT
*
*      MAT == MATRIX TIMES A VECTOR OPERATION
*
*      TYPICAL CALLING SEQUENCE:
*
*      load      AR0
*      load      AR1
*      load      AR2
*      load      AR3
*      load      R1
*      CALL      MAT
*
*      ARGUMENT ASSIGNMENTS:
*
*      ARGUMENT | FUNCTION
*      -----|-----
*      AR0      | ADDRESS OF M(0,0)
*      AR1      | ADDRESS OF V(0)
*      AR2      | ADDRESS OF P(0)
*      AR3      | NUMBER OF ROWS - 1 (K-1)
*      R1       | NUMBER OF COLUMNS - 2 (N-2)
*
*      REGISTERS USED AS INPUT: AR0, AR1, AR2, AR3, R1
*      REGISTERS MODIFIED: R0, R2, AR0, AR1, AR2, AR3, IR0, RC
*
*      PROGRAM SIZE: 11
*
*      EXECUTION CYCLES: 5 + 7K + KN = 5 + ((N-1) + 8) * K
*
*      .global  MAT
*
*      SETUP
*
MAT      ADDI3    R1,2,IR0                ; IR0 = N
*
*      FOR (i = 0; i < K; i++) LOOP OVER THE ROWS.
*
ROWS     RPTBD   Dot                    ; Setup multply a row by a
*                                           ; column.
*      LDI      R1,RC                    ; Set loop counter
*      LDF      0.0,R2                   ; Initialize R2
*      MPYF3    *AR0++(1),*AR1++(1),R0   ; m(i,0) * v(0) -> R0
*

```

```

*      FOR (j = 1; j < N; j++) DO DOT PRODUCT OVER COLUMNS
*
DOT    MPYF3      *AR0++(1), *AR1++(1), R0 ; m(i, j) * v(j) -> R0
||    ADDF3      R0, R2, R2                ; m(i, j-1) *v(j-1) +
*                                           ; R2 -> R2.
*
*      DBD        AR3, ROWS                ; counts the number of rows
*                                           ; left.
*
*
*      ADDF      R0, R2                    ; last accumulate.
*      STF       R2, *AR2++(1)             ; result -> p(i)
*      NOP       *--AR1(IR0)              ; set AR1 to point to v(0).
*      !!! DELAYED BRANCH HAPPENS HERE !!!
*
*      RETURN SEQUENCE
*
*      RETS                                     ; return
*
*      end
*
*      .end

```

#### 12.4.4 Fast Fourier Transforms (FFT)

Fourier transforms are an important tool often used in digital signal processing systems. The purpose of the transform is to convert information from the time domain to the frequency domain. The inverse Fourier transform converts information back to the time domain from the frequency domain. Implementation of Fourier transforms that are computationally efficient are known as fast Fourier transforms (FFTs). The theory of FFTs can be found in books such as *DFT/FFT and Convolution Algorithms* by C.S. Burrus and T.W. Parks (John Wiley, 1985), and in the book *Digital Signal Processing Applications with the TMS320 Family*.

Certain 'C40 features that increase efficient implementation of numerically intensive algorithms are particularly well-suited for FFTs. The high speed of the device (40-ns cycle time) makes the implementation of realtime algorithms easier, while the floating-point capability eliminates the problems associated with dynamic range. The powerful indexing scheme in indirect addressing facilitates the access of FFT butterfly legs that have different spans. The repeat block implemented by the RPTB or RPTBD instruction reduces the looping overhead in algorithms heavily dependent on loops (such as the FFTs). This construct gives the efficiency of in-line coding but has the form of a loop. Since the output of the FFT is in scrambled (bit-reversed) order when the input is in regular order, it must be restored to the proper order. This rearrangement does not require extra cycles. The device has a special form of indirect addressing (bit-reversed addressing mode) that can be used when the FFT output is needed.

The 'C40 can implement this mode on either the CPU or DMA. This mode permits accessing the FFT output in the proper order. If the DMA transfer with bit-reversed addressing mode is used, there is no overhead for data input and output.

There are several types of FFTs:

- Radix-2 and radix-4 algorithms depending on the size of the FFT butterfly
- Decimation in time or frequency (DIT or DIF)
- Complex or real FFTs
- FFTs of different lengths, etc.

The examples in this section of FFT implementation are based on programs contained in the application report, *"An Implementation of FFT, DCT, and Other Transforms on the TMS320C30"*, by Panos Papamichalis in the *Digital Signal Processing Applications with the TMS320 Family*, volume III.

Example 12–37 and Example 12–38 show the implementation of a complex radix-2, DIF FFT on the 'C40. Example 12–37 contains the generic code of the FFT that can be used with any length number. However, for the complete implementation of an FFT, a table of twiddle factors (sines/cosines) is needed, and this table depends on the size of the transform. To retain the generic form of Example 12–37, the table with the twiddle factors (containing 1-1/4 complete cycles of a sine) is presented separately in Example 12–38 for the case of a 64-point FFT. A full cycle of a sine should have a number of points equal to the FFT size. In Example 12–38, the FFT length  $N$  and  $M$ , which is equal to the logarithm of  $N$  to base equal to the radix, are defined.  $M$  is the number of stages of the FFT. For a 64-point FFT,  $M = 6$  when using a radix-2 algorithm, or  $M = 3$  when using a radix-4 algorithm. If the table with the twiddle factors and the FFT code are kept in separate files, they should be connected at link time.

## Example 12-37. Complex, Radix-2, DIF FFT

```

*
*      TITLE COMPLEX, RADIX-2, DIF FFT
*
*      GENERIC PROGRAM FOR A LOOPED-CODE RADIX-2 FFT COMPUTATION
*      IN 320C40
*
*      THE PROGRAM IS DERIVED FROM THE BURRUS AND PARKS
*      BOOK, "DFT/FFT AND CONVOLUTION ALGORITHMS", PAGE 111.
*      THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY.
*      THE COMPUTATION IS DONE IN-PLACE, BUT THE
*      RESULT IS MOVED TO ANOTHER MEMORY SECTION TO
*      DEMONSTRATE THE BIT-REVERSED ADDRESSING.
*
*      THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A
*      .DATA SECTION. THIS DATA IS INCLUDED IN A SEPARATE
*      FILE TO PRESERVE THE GENERIC NATURE OF THE PROGRAM.
*      FOR THE SAME PURPOSE, THE SIZE OF THE FFT N AND
*      LOG2(N) ARE DEFINED IN A .GLOBL DIRECTIVE AND
*      SPECIFIED DURING LINKING.
*
*
*      .globl    FFT          ; Entry point for execution
*      .globl    N            ; FFT size
*      .globl    M            ; LOG2(N)
*      .globl    SINE        ; Address of sine table
*
INP    .usect    "IN",1024   ; Memory with input data
      .BSS     OUTP,1024   ; Memory with output data
*
      .text
*      INITIALIZE
FFTSIZ .word    N
LOGFFT .word    M
SINTAB .word    SINE
INPUT  .word    INP
OUTPUT .word    OUTP
*
FFT:   LDP      FFTSIZ      ; Command to load data page pointer
      LDI      @FFTSIZ,R7  ; R7=N2
      LSH3     -2,R7,IR1   ; IR1=N/4, pointer for SIN/COS table
      LDI      @LOGFFT,R9  ; R9 holds the remain stage number
      LSH3     1,R7,IR0    ; IR0=2*N1 (because of real/imag)
      LDI      1,R8        ; Initialize repeat counter of first
                        ; loop
      LDI      1,AR5       ; Initialize IE index (AR5=IE)
      LDI      @INPUT,R10  ; R10 points to X(I)

```

```

*      OUTER LOOP
LOOP:  RPTBD   BLK1           ; Setup for first loop
      LDI    R10,AR0        ; AR0 points to X(I)
      ADDI   R7,AR0,AR2     ; AR2 points to X(L)
      SUBI3  1,R8,RC        ; RC should be one less than
                               ; desired #

*      FIRST LOOP
      ADDF   *AR0,*AR2,R0   ; R0 = X(I) + X(L)
      SUBF   *AR2++,*AR0++,R1 ; R1 = X(I) - X(L)
      ADDF   *AR2,*AR0,R2   ; R2 = Y(I) + Y(L)
      SUBF   *AR2,*AR0,R3   ; R3 = Y(I) - Y(L)
      STF    R2,*AR0--      ; Y(I) = R2 and...
      STF    R3,*AR2--      ; Y(L) = R3
      BLK1  STF    R0,*AR0++(IR0) ; X(I) = R0 and...
      STF    R1,*AR2++(IR0) ; X(L) = R1 and AR0,2 = AR0,2 + 2*n
*      IF THIS IS THE LAST STAGE, YOU ARE DONE
      SUBI   1,R9
      BZD   END

*      MAIN INNER LOOP
      LDI    2,AR1          ; Init loop counter for inner loop
      LDI    @SINTAB,AR4    ; Initialize IA index (AR4=IA)
      ADDI   AR5,AR4        ; IA=IA+IE; AR4 points to cosine
      ADDI   R10,AR1,AR0    ; (X(I),Y(I)) pointer
      ADDI   2,AR1          ; Increase inner loop counter
INLOP: RPTBD   BLK2         ; Setup for second loop
      ADDI   R7,AR0,AR2     ; (X(L),Y(L)) pointer
      SUBI   1,R8,RC        ; RC should be one less than
                               ; desired #
      LDF    *AR4,R6        ; R6=SIN

*      SECOND LOOP
      SUBF   *AR2,*AR0,R2   ; R2=X(I)-X(L)
      SUBF   **AR2,**AR0,R1

*      MPYF   R2,R6,R0      ; R1 = Y(I) - Y(L)
                               ; R0 = R2*SIN and ...
      ADDF   **AR2,**AR0,R3 ; R3 = Y(I) + Y(L)
      MPYF   R1,**AR4(IR1),R3 ; R3 = R1 * COS and ...
      STF    R3,**AR0      ; Y(I) = Y(I) + Y(L)
      SUBF   R0,R3,R4      ; R4 = R1*COS - R2*SIN
      MPYF   R1,R6,R0      ; R0 = R1*SIN and...
      ADDF   *AR2,*AR0,R3   ; R3 = X(I) + X(L)
      MPYF   R2,**AR4(IR1),R3 ; R3 = R2 * COS and ...
      STF    R3,*AR0++(IR0)

*      ADDF   R0,R3,R5      ; X(I) = X(I) + X(L) and AR0=AR0 + 2*N1
                               ; R5 = R2*COS + R1*SIN
      BLK2  STF    R5,*AR2++(IR0) ; X(L) = R2*COS + R1*SIN, incr AR2
                               ; and...
      STF    R4,**AR2      ; Y(L) = R1*COS - R2*SIN

```

```

    CMPI      R7,AR1
    BNEAF    INLOP      ; Loop back to the inner loop
    ADDI     AR5,AR4    ; IA = IA + IE; AR4 points to cosine

    ADDI     R10,AR1,ARO ; (X(I),Y(I)) pointer
    ADDI     2,AR1     ; Increase inner loop counter

    LSH      1,R8      ; Increment loop counter for
                    ; next time
    BRD      LOOP      ; Next FFT stage (delayed)
    LSH      1,AR5     ; IE = 2*IE
    LDI      R7,IRO    ; N1 = N2
    LSH      -1,R7     ; N2 = N2/2

*      STORE RESULT OUT USING BIT-REVERSED ADDRESSING
END:   LDI      @FFTSIZ,IRO ; IRO = size of FFT = N
      SUBI3   2,IRO,RC    ; RC = N - 2
      LDI     2,IR1
      RPTBD   BITRV
      LDI     @INPUT,ARO
      LDI     @OUTPUT,AR1
      LDF     *+AR0(1),R0
*      BIT    REVERSE LOOP
      LDF     *AR0++(IRO)B,R1
||     STF     R0,*+AR1(1)
BITRV  LDF     *+AR0(1),R0
||     STF     R1,*AR1++(IR1)

      LDF     *AR0++(IRO)B,R1
||     STF     R0,*+AR1(1)
      STF     R1,*AR1++(IR1)
SELF   BR      SELF    ; Branch to itself at the end
      .end

```

*Example 12-38. Table with Twiddle Factors for a 64-Point FFT*

```
*
*      TITLE TABLE WITH TWIDDLE FACTORS FOR A 64-POINT FFT
*
*      FILE TO BE LINKED WITH THE SOURCE CODE FOR A 64-POINT,
*      RADIX-2 FFT.
*
      .globl      SINE
      .globl      N
      .globl      M

N      .set       64
M      .set       6

      .data

SINE   .float     0.000000
      .float     0.098017
      .float     0.195090
      .float     0.290285
      .float     0.382683
      .float     0.471397
      .float     0.555570
      .float     0.634393
      .float     0.707107
      .float     0.773010
      .float     0.831470
      .float     0.881921
      .float     0.923880
      .float     0.956940
      .float     0.980785
      .float     0.995185

COSINE .float     1.000000
      .float     0.995185
      .float     0.980785
      .float     0.956940
      .float     0.923880
      .float     0.881921
      .float     0.831470
      .float     0.773010
      .float     0.707107
      .float     0.634393
      .float     0.555570
      .float     0.471397
      .float     0.382683
      .float     0.290285
      .float     0.195090
      .float     0.098017
      .float     0.000000
      .float     -0.098017
      .float     -0.195090
      .float     -0.290285
      .float     -0.382683
      .float     -0.471397
```

```
.float    -0.555570
.float    -0.634393
.float    -0.707107
.float    -0.773010
.float    -0.831470
.float    -0.881921
.float    -0.923880
.float    -0.956940
.float    -0.980785
.float    -0.995185
.float    -1.000000
.float    -0.995185
.float    -0.980785
.float    -0.956940
.float    -0.923880
.float    -0.881921
.float    -0.831470
.float    -0.773010
.float    -0.707107
.float    -0.634393
.float    -0.555570
.float    -0.471397
.float    -0.382683
.float    -0.290285
.float    -0.195090
.float    -0.098017

.float    0.000000
.float    0.098017
.float    0.195090
.float    0.290285
.float    0.382683
.float    0.471397
.float    0.555570
.float    0.634393
.float    0.707107
.float    0.773010
.float    0.831470
.float    0.881921
.float    0.923880
.float    0.956940
.float    0.980785
.float    0.995185
```

The radix-2 algorithm has tutorial value because it is relatively easy to understand how the FFT algorithm functions. However, radix-4 implementations can increase the speed of the execution by reducing the overall arithmetic required. Example 12-39 shows the generic implementation of a complex, DIF FFT in radix-4. A companion table, like the one in Example 12-38, should have a value of  $M$  equal to the  $\log N$ , where the base of the logarithm is four.

Example 12-39. Complex, Radix-4, DIF FFT

```

*
*      TITLE COMPLEX, RADIX-4, DIF FFT
*
*      GENERIC PROGRAM TO DO A LOOPED-CODE RADIX-4 FFT COMPUTATION IN
*      THE TMS320C40.
*
*      THE PROGRAM IS DERIVED FROM THE BURRUS AND PARKS BOOK,
*      DFT/FFT AND CONVOLUTION ALGORITHMS, P. 117. THE
*      (COMPLEX) DATA RESIDE IN INTERNAL MEMORY, AND THE
*      COMPUTATION IS DONE IN-PLACE.
*
*      THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A
*      .DATA SECTION. THIS DATA IS INCLUDED IN A SEPARATE FILE TO
*      PRESERVE THE GENERIC NATURE OF THE PROGRAM. FOR THE SAME
*      PURPOSE, THE SIZE OF THE FFT N AND LOG4(N) ARE DEFINED IN A
*      .GLOBL DIRECTIVE AND SPECIFIED DURING LINKING.
*
*      IN ORDER TO HAVE THE FINAL RESULT IN BIT-REVERSED ORDER, THE
*      TWO MIDDLE BRANCHES OF THE RADIX-4 BUTTERFLY ARE INTERCHANGED
*      DURING STORAGE. NOTE THIS DIFFERENCE WHEN COMPARING WITH THE
*      PROGRAM IN P. 117 OF THE BURRUS AND PARKS BOOK.
*
*
*      .globl   FFT           ; Entry point for execution
*      .globl   N             ; FFT size
*      .globl   M             ; LOG4(N)
*      .globl   SINE          ; Address of sine table
INP  .usect   "IN",1024      ; Memory with input data
     .bss     OUTP,1024    ; Memory with output data
     .text
*
*      INITIALIZE
FFTSIZ .word   N           ; FFT size
LOGFFT .word   M           ; LOG4(FFTSIZ)
SINTAB .word   SINE        ; Sine/cosine table base
INPUT  .word   INP         ; Area with input data to process
OUTPUT .word   OUTP        ; Area with output data to process
*
FFT:   LDP     FFT         ; Command to load data page pointer
       LDI     @FFTSIZ,BK
LSH3   1,BK,IRO           ; IRO=2*N1 (because of real/imag)
       LSH3   -2,BK,IR1    ; IR1=N/4, pointer for SIN/COS table
       LDI     1,AR7       ; Initialize IE index
       LDI     1,R8        ; Initialize repeat counter of first
                           ; loop
       ADDI    2,IR1,R9    ; R9 = JT = R0/2 + 2
LSH    -1,BK             ; R7 = N2

```

```

*      OUTER LOOP
LOOP:  LDI      @INPUT,AR0      ; AR0 points to X(I)
      ADDI    BK,AR0,AR1      ; AR1 points to X(I1)
      ADDI    BK,AR1,AR2      ; AR2 points to X(I2)
      RPTBD   BLK1           ; Setup loop BLK1
      ADDI    BK,AR2,AR3      ; AR3 points to X(I3)
      SUBI3   1,R8,RC         ; RC should be one less
      ; than desired #
      LDF     **AR1,R0        ; R0 = Y(I1)

*      FIRST LOOP: BLK1
      ADDF    R0,**AR3,R3     ; R3 = Y(I1) + Y(I3)
      ADDF    **AR0,**AR2,R1  ; R1 = Y(I) + Y(I2)
      ADDF    R3,R1,R6        ; R6 = R1 + R3
      SUBF    **AR2,**AR0,R4  ; R4 = Y(I) - Y(I2)
      LDF     *AR2,R5         ; R5 = X(I2)
      ||     STF     R6,**AR0  ; Y(I) = R1 + R3
      SUBF    R3,R1           ; R1 = R1 - R3
      ADDF    *AR3,*AR1,R3    ; R3 = X(I1) + X(I3)
      ADDF    R5,*AR0,R1      ; R1 = X(I) + X(I2)
      ||     STF     R1,**AR1  ; Y(I1) = R1 - R3
      ADDF    R3,R1,R6        ; R6 = R1 + R3
      SUBF    R5,*AR0,R4      ; R4 = X(I) - X(I2)
      ||     STF     R6,*AR0++(IR0) ; X(I) = R1 + R3
      SUBF    R3,R1           ; R1 = R1 - R3
      SUBF    *AR3,*AR1,R6    ; R6 = X(I1) - X(I3)
      SUBF    R0,**AR3,R3     ; -R3 = Y(I1) - Y(I3)
      ||     STF     R1,*AR1++(IR0) ; X(I1) = R1-R3
      SUBF    R6,R4,R5        ; R5 = R4 - R6
      ADDF    R6,R4           ; R2 = R4 + R6
      STF     R5,**AR2        ; Y(I2) = R4 - R6
      ||     STF     R2,**AR3    ; Y(I3) = R4 + R6
      SUBF    R3,R2,R5        ; R5 = R2 - R3
      ADDF    R3,R2           ; R2 = R2 + R3
      STF     R2,*AR3++(IR0)  ; X(I3) = R2 + R3
BLK1  STF     R5,*AR2++(IR0)  ; X(I2) = R2 - R3
      ||     LDF     **AR1,R0    ; R0 = Y(I1)

*      IF THIS IS THE LAST STAGE, YOU ARE DONE
      CMPI    IR1,R8
BZD   END

*      MAIN INNER LOOP
      LDI     1,R10           ; Init IA1 index
      LDI     2,R11           ; Init loop counter for inner loop
      LDI     R11,AR0
      ADDI    @INPUT,AR0     ; (X(I),Y(I)) pointer
      ADDI    2,R11           ; Increment inner loop counter

```

```

INLOP:  ADDI      R10,AR7           ; IA1 = IA1 + IE
        ADDI      BK,AR0,AR1       ; (X(I1),Y(I1)) pointer
        CMPI      R9,R11          ; If LPCNT = JT, go to
        BZD       SPCL            ; special butterfly
        ADDI      BK,AR1,AR2       ; (X(I2),Y(I2)) pointer
        ADDI      BK,AR2,AR3       ; (X(I3),Y(I3)) pointer
        SUBI3     1,R8,RC          ; RC should be one less than
                                   ; desired #

        LDI       R10,AR4
        ADDI      @SINTAB,AR4      ; Create cosine index AR4
        ADDI      AR4,R10,AR5
        SUBI      1,AR5            ; IA2 = IA1 + IA1 - 1
        RPTBD     BLK2            ; Setup loop BLK2
        ADDI      10,AR5,AR6
        SUBI      1,AR6            ; IA3 = IA2 + IA1 - 1
        LDF       *+AR2,R7        ; R7 = Y(I2)

*      SECOND LOOP: BLK2
        ADDF      R7,*+AR0,R3      ; R3 = Y(I) + Y(I2)
        ADDF      *+AR3,*+AR1,R5   ; R5 = Y(I1) + Y(I3)
        ADDF      R5,R3,R6         ; R6 = R3 + R5
        SUBF      R7,*+AR0,R4      ; R4 = Y(I) - Y(I2)
        SUBF      R5,R3           ; R3 = R3 - R5
        ADDF      *AR2,*AR0,R1     ; R1 = X(I) + X(I2)
        ADDF      *AR3,*AR1,R5     ; R5 = X(I1) + X(I3)
        MPYF      R3,*+AR5(IR1),R6 ; R6 = R3*CO2
||      STF       R6,*+AR0        ; Y(I) = R3 + R5
        ADDF      R5,R1,R0         ; R0 = R1 + R5
        SUBF      *AR2,*AR0,R2     ; R2 = X(I) - X(I2)
        SUBF      R5,R1           ; R1 = R1 - R5
        MPYF      R1,*AR5,R0       ; R0 = R1*SI2
||      STF       R0,*AR0++(IR0)   ; X(I) = R1 + R5
        SUBF      R0,R6           ; R6 = R3*CO2 - R1*SI2
        SUBF      *+AR3,*+AR1,R5   ; R5 = Y(I1) - Y(I3)
        MPYF      R1,*+AR5(IR1),R0 ; R0 = R1*CO2
||      STF       R6,*+AR1        ; Y(I1) = R3*CO2 - R1*SI2
        MPYF      R3,*AR5,R6       ; R6 = R3*SI2
        ADDF      R0,R6           ; R6 = R1*CO2 + R3*SI2
        ADDF      R5,R2,R1         ; R1 = R2 + R5
        SUBF      R5,R2           ; R2 = R2 - R5
        SUBF      *AR3,*AR1,R5     ; R5 = X(I1) - X(I3)
        SUBF      R5,R4,R3        ; R3 = R4 - R5
        ADDF      R5,R4           ; R4 = R4 + R5
        MPYF      R3,*+AR4(IR1),R6 ; R6 = R3*CO1
||      STF       R6,*AR1++(IR0)   ; X(I1) = R1*CO2 + R3*SI2
        MPYF      R1,*AR4,R0       ; R0 = R1*SI1
        SUBF      R0,R6           ; R6 = R3*CO1 - R1*SI1
        MPYF      R1,*+AR4(IR1),R6 ; R6 = R1*CO1
||      STF       R6,*+AR2        ; Y(I2) = R3*CO1 - R1*SI1
        MPYF      R3,*AR4,R0       ; R0 = R3*SI1
        ADDF      R0,R6           ; R6 = R1*CO1 + R3*SI1
        MPYF      R4,*+AR6(IR1),R6 ; R6 = R4*CO3

```

```

||      STF      R6, *AR2++ (IR0)      ; X(I2) = R1*CO1 + R3*SI1
      MPYF      R2, *AR6, R0          ; R0 = R2*SI3
      SUBF      R0, R6                ; R6 = R4*CO3 - R2*SI3
      MPYF      R2, **AR6 (IR1), R6   ; R6 = R2*CO3
||      STF      R6, **AR3            ; Y(I3) = R4*CO3 - R2*SI3
      MPYF      R4, *AR6, R0          ; R0 = R4*SI3
      ADDF      R0, R6                ; R6 = R2*CO3 + R4*SI3
BLK2    STF      R6, *AR3++ (IR0)      ; x(i3) = R2*CO3 + R4*SI3
||      LDF      **AR2, R7            ; Load next Y(I2)
      CMPI      R11, BK
      BPD       INLOP                 ; LOOP BACK TO THE INNER LOOP
      LDI       R11, AR0
      ADDI      @INPUT, AR0           ; (X(I),Y(I)) pointer
      ADDI      2, R11                ; Increment inner loop counter
      BRD       CONT
      LSH       2, R8                 ; Increment repeat counter for
                                       ; next time
      LSH       2, AR7                ; IE = 4*IE
      LDI       BK, IR0               ; N1 = N2

*
SPECIAL BUTTERFLY FOR W=J
SPCL    RPTBD   BLK3                 ; Setup loop BLK3
      LSH      -1, IR1, AR4           ; Point to SIN(45)
      ADDI     @SINTAB, AR4           ; Create cosine index AR4 = CO21

      LDF      *AR2, R7               ; R7 = X(I2)
*SPCL LOOP: BLK3
      ADDF     R7, *AR0, R1            ; R1 = X(I) + X(I2)
      ADDF     **AR2, **AR0, R3        ; R3 = Y(I) + Y(I2)
      SUBF     **AR2, **AR0, R4        ; R4 = Y(I) - Y(I2)
      ADDE     *AR3, *AR1, R5         ; R5 = X(I1) + X(I3)
      SUBF     R1, R5, R6              ; R6 = R5 - R1
      ADDF     R5, R1                  ; R1 = R1 + R5
      ADDF     **AR3, **AR1, R5        ; R5 = Y(I1) + Y(I3)
      SUBF     R5, R3, R0              ; R0 = R3 - R5
      ADDF     R5, R3                  ; R3 = R3 + R5
      SUBF     R7, *AR0, R2            ; R2 = X(I) - X(I2)
||      STF     R3, **AR0              ; Y(I) = R3 + R5
      LDF     *AR3, R7                 ; R7 = X(I3)
||      STF     R1, *AR0++ (IR0)       ; X(I) = R1 + R5
      SUBF     R7, *AR1, R1            ; R1 = X(I1) - X(I3)
||      STF     R6, **AR1              ; Y(I1) = R5 - R1
      SUBF     **AR3, **AR1, R3        ; R3 = Y(I1) - Y(I3)
      ADDE     R3, R2, R3              ; R5 = R2 + R3
      SUBF     R2, R3, R2              ; R2 = -R2 + R3
      SUBF     R1, R4, R1              ; R3 = R4 - R1
      ADDF     R1, R4                  ; R4 = R4 + R1
      SUBF     R5, R3, R1              ; R1 = R3 - R5
      MPYF     R1, *AR4, R1            ; R1 = R1*CO21
||      STF     R0, *AR1++ (IR0)       ; X(I1) = R3 - R5
      ADD     R5, R3                   ; R3 = R3 + R5
      MPYF     R3, *AR4, R3            ; R3 = R3*CO21
||      STF     R1, **AR2              ; Y(I2) = (R3 - R5)*CO21
      SUBF     R4, R2, R1              ; R1 = R2 - R4

```

```

MPYF      R1,*AR4,R1      ; R1 = R1*CO21
||      STF      R3,*AR2++(IR0) ; X(I2) = (R3 + R5)*CO21
ADDF      R4,R2          ; R2 = R2 + R4
MPYF3     R2,*AR4,R2     ; R2 = R2*CO21
||      STF      R1,*+AR3      ; Y(I3) = -(R4 - R2)*CO21
BLK3     LDF      *AR2,R      ; Load next X(I2)
||      STF      R2,*AR3++(IR0) ; X(I3) = (R4 + R2)*CO21
CMPFI     R11,BK
BPD      INLOP          ; Loop back to the inner loop
LDI      R11,ARO
ADDI     @INPUT,ARO     ; (X(I),Y(I)) pointer
ADDI     2,R11          ; Increment inner loop counter
LSH      2,R8           ; Increment repeat counter for next
                        ; time
LSH      2,AR           ; IE = 4*IE
LDI      BK,IR0        ; N1 = N2
CONT     BRD      LOOP    ; Next FFT stage (delayed)
LSH      -2,BK         ; N2 = N2/4
LSH3     -1,BK,R9
ADDI     2,R9          ; JT = N2/2 + 2
*      STORE RESULT OUT USING BIT-REVERSED ADDRESSING
END:     LDI      @FFTSIZ,IR0 ; IR0 = size of FFT = N
SUBI3    2,IR0,RC      ; RC = N - 2
LDI      2,IR1
RPTBD    BITRV
LDI      @INPUT,ARO
LDI      @OUTPUT,AR1
LDF      *+AR0(1),R0
*      BIT REVERSE LOOP
LDF      *AR0++(IR0)B,R1
||      STF      R0,*+AR1(1)
BITRV    LDF      *+AR0(1),R0
||      STF      R1,*AR1++(IR1)
LDF      *AR0++(IR0)B,R1
||      STF      R0,*+AR1(1)
||      STF      R1,*AR1++(IR1)
SELF     BR      SELF   ; Branch to itself at the end.
.end

```

Most often, the data to be transformed is a sequence of real numbers. In this case, the FFT demonstrates certain symmetries that permit the reduction of the computational load even further. Example 12-40 shows the generic implementation of a real-valued, radix-2 FFT. For such an FFT, the total storage required for a length-N transform is only N locations; in a complex FFT, 2N are necessary. Recovery of the rest of the points is based on the symmetry conditions.

## Example 12-40. Real, Radix-2 FFT

```

*
*      TITLE REAL, RADIX-2 FFT
*
*      GENERIC PROGRAM TO DO A RADIX-2 REAL FFT COMPUTATION
*      IN 320C40.
*
*      THE PROGRAM IS DERIVED FROM THE PAPER BY SORENSEN ET AL.,
*      JUNE 1987 ISSUE OF THE TRANSACTIONS ON ASSP.
*
*      THE REAL DATA RESIDE IN INTERNAL MEMORY. THE COMPUTATION IS
*      DONE IN-PLACE. THE BIT-REVERSAL IS DONE AT THE BEGINNING OF
*      THE PROGRAM.
*
*      THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A .DATA
*      SECTION. THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE
*      THE GENERIC NATURE OF THE PROGRAM. FOR THE SAME PURPOSE, THE
*      SIZE OF THE FFT N AND LOG2(N) ARE DEFINED IN A .GLOBL
*      DIRECTIVE AND SPECIFIED DURING LINKING. THE LENGTH OF
*      THE TABLE IS  $N/4 + N/4 = N/2$ .
*
*
*      .globl      FFT                ; Entry point for execution
*      .globl      N                  ; FFT size
*      .globl      M                  ; LOG2(N)
*      .globl      SINE               ; Address of sine table
*      .bss        INP,1024          ; Memory with input data
*      .text
*
*      INITIALIZE
FFTSIZ      .word      N
LOGFFT      .word      M
SINTAB      .word      SINE
INPUT       .word      INP
FFT:        LDP         FFTSIZ        ; Command to load data page printer
*
*      DO THE BIT-REVERSING AT THE BEGINNING
LDI         @FFTSIZ,R8              ; R8 = N
SUBI        1,R8,RC                 ; RC should be one less
                                         ; than desired #
LDI         @SINTAB,R9
RPTBD      BITRV                    ; Setup for BITRV loop
LSH3       -1,R8,IR0                ; IR1 = half the size of FFT = N/2
LDI         @INPUT,AR0              ; AR0 points to X(I)
LDI         @INPUT,AR1              ; AR1 points to X(I)

```

```

*      DIGIT REVERSE COUNTER
      CMPI      AR1,AR0                ; Exchange locations only
      LDF      *AR0++(1),R1          ; if AR0<AR1
||
      LDF      *AR1++(IRO)B,R0
      LDFLT   *AR0,R0
      LDFLT   *AR1,R1
BITRV  STF      R0,*AR1
||
      STF      R1,*AR0

*      LENGTH-TWO BUTTERFLIES
      LDI      @INPUT,AR0            ; AR0 points to X(I)
      RPTBD   BLK1                   ; Setup for BLK1 loop
      SUBI3   1,IRO,RC               ; RC = (N/2) -1
      LDF     *+AR0(1),R2            ; R2 = X(I + 1)
      LDI     2,IRO                  ; IRO = 2 = N2

*      BLK1 LOOP
      ADDF    R2,*AR0,R0              ; R0 = X(I) + X(I + 1)
      SUBF    R2,*AR0,R1              ; R1 = X(I) - X(I + 1)
||
      STF     R0,*AR0++(1)            ; X(I) = X(I) + X(I + 1)
BLK1  LDF     *+AR0(IRO),R2          ; Load next X(I)
||
      STF     R1,*AR0++(1)            ; X(I + 1) = X(I) - X(I + 1)

*      FIRST PASS OF THE DO-20 LOOP (STAGE K = 2 IN DO-10 LOOP)
      LDI      @INPUT,AR0            ; AR0 points to X(I)
      RPTBD   BLK2                   ; Setup for BLK2 loop
      LSH3    -2,R8,RC               ; Repeat N/4 times
      SUBI    1,RC                   ; RC should be one less
                                           ; than desired #
      LDF     *+AR0(IRO),R2          ; R2 = X(I + 2)

*      BLK2 LOOP
      ADDF    R2,*AR0++(IRO),R0      ; R0 = X(I) + X(I + 2)
      SUBF    R2,*-AR0(IRO),R1      ; R1 = X(I) - X(I + 2)
||
      STF     R0,*-AR0(IRO)          ; X(I) = X(I) + X(I + 2)
      NEGF    *+AR0,R0               ; R0 = -X(I + 3)
||
      STF     R1,*AR0++(IRO)          ; X(I + 2) = X(I) - X(I + 2)
BLK2  LDF     *+AR0(IRO),R2          ; Load next X(I + 2)
||
      STF     R0,*-AR0               ; X(I + 3) = -X(I + 3)

*      MAIN LOOP (FFT STAGES)
      LSH3    -3,R8,IRO              ; IRO = E/2 index for E
      LDI     3,R11                  ; R11 holds the current
                                           ; stage number
      LDI     2,R4                   ; R4 = N4
      LDI     4,R3                   ; R3 = N2
LOOP  LDI     @INPUT,AR5              ; AR5 points to X(I)
      LSH3    2,R4,R10               ; Set loop counter
      ADDI3   IRO,R9,AR0             ; AR0 points to SIN/COS
                                           ; table

```

```

*      INNER LOOP (DO-20 LOOP IN THE PROGRAM)
INLOP  LDI      R4,IR1          ; IR1 = N4 or N2/2
      ADDI3   1,AR5,AR1       ; AR1 points to X(I1) = X(I + J)
      ADDI3   R3,AR1,AR3      ; AR3 points to
      ; X(I3) = X(I + J + N2)
      SUBI3   2,AR3,AR2       ; AR2 points to
      ; X(I2) = X(I - J + N2)
      ADDI    R3,AR2,AR4      ; AR4 points to
      ; X(I4) = X(I - J + N1)
      LDF     *AR5++(IR1),R0   ; R0 = X(I)
      ADDF    **AR5(IR1),R0,R1 ; R1 = X(I) + X(I + N2)
      SUBF    R0,**AR5(IR1),R0 ; R0 = -X(I) + X(I + N2)
||     STF    R1,*-AR5(IR1)    ; X(I) = X(I) + X(I + N2)
      NEGF    R0              ; R0 = X(I) - X(I + N2)
      NEGF    **AR5(IR1),R1   ; R1 = -X(I + N4 + N2)
||     STF    R0,*AR5         ; X(I + N2) = X(I) - X(I + N2)
      STF    R1,*AR5         ; X(I + N4 + N2) = -X(I + N4 + N2)
*      INNERMOST LOOP
      RPTBD   BLK3           ; Setup for BLK3 loop
      LSH3   -2,R8,         ; IR1=separation between
      ; SIN/COS tbls
      SUBI    2,R4,RC        ; Repeat N4 - 1 times
      LDF     *AR3,R5        ; R5 = X(I3)
*      BLK3 LOOP
      MPYF    R5,**AR0(IR1),R0 ; R0 = X(I3)*COS
      MPYF    *AR4,*AR0,R1    ; R1 = X(I4)*SIN
      MPYF    *AR4,**AR0(IR1),R1 ; R1 = X(I4)*COS
||     ADDF    R0,R1,R2      ; R2 = X(I3)*COS + X(I4)*SIN
      MPYF    R5,*AR0++(IR0),R0 ; R0 = X(I3)*SIN
      SUBF    R0,R1,R0      ; R0 = -X(I3)*SIN + X(I4)*COS
      SUBF    *AR2,R0,R1    ; R1 = -X(I2) + R0
      ADDF    *AR2,R0,R1    ; R1 = X(I2) + R0
||     STF    R1,*AR3++     ; X(I3) = -X(I2) + R0
      ADDF    *AR1,R2,R1    ; R1 = X(I1) + R2
||     STF    R1,*AR4--     ; X(I4) = X(I2) + R0
      SUBF    R2,*AR1,R1    ; R1 = X(I1) - R2
||     STF    R1,*AR1++     ; X(I1) = X(I1) + R2
BLK3   LDF     *AR3,R5      ; Load next X(I3)
||     STF    R1,*AR2--     ; X(I2) = X(I1) - R2
      CMPI   @FFTSIZ,R10
      BLTAF  INLOP         ; Loop back inner to theloop
      ADDI   R4,AR5        ; AR5 = I + N1
      ADDI   R10,R10
      ADDI3  IR0,R9,AR0    ; AR0 points to
      ; SIN/COS table
      ADDI   1,R11
      CMPI  @LOGFFT,R11
      BLEAF  LOOP
      LSH   -1,IR0        ; E = E/2
      LSH   1,R4         ; N4 = 2*N4
      LSH   1,R3         ; N2 = 2*N2
END    BR     END        ; Branch to itself at the end.
      .end

```

Example 12–37, Example 12–39, and Example 12–40 provide an easy understanding of the FFT algorithm functions. However, they are not optimized for fast speed execution of FFT. Example 12–41 shows a faster version of a radix-2 DIT FFT algorithm. This program uses a different twiddle factors table than the previous examples. The twiddle factors are stored in bit reversed order and with a table length of  $N/2$  ( $N = \text{FFT length}$ ). For instance, if the FFT length is 32, the twiddle factors table should be:

<u>Address</u>	<u>Coefficient</u>
0	$R\{WN(0)\} = \text{COS}(2*PI*0/32) = 1$
1	$-I\{WN(0)\} = \text{SIN}(2*PI*0/32) = 0$
2	$R\{WN(4)\} = \text{COS}(2*PI*4/32) = 0.707$
3	$-I\{WN(4)\} = \text{SIN}(2*PI*4/32) = 0.707$
.	.
.	.
.	.
12	$R\{WN(3)\} = \text{COS}(2*PI*3/32) = 0.831$
13	$-I\{WN(3)\} = \text{SIN}(2*PI*3/32) = 0.556$
14	$R\{WN(7)\} = \text{COS}(2*PI*7/32) = 0.195$
15	$-I\{WN(7)\} = \text{SIN}(2*PI*7/32) = 0.981$

*Example 12–41. Faster Version Complex, Radix-2 DIT FFT*

```

*      TITLE FASTER VERSION COMPLEX, RADIX-2 DIT FFT
*
*      GENERIC PROGRAM FOR A FAST LOOPED-CODE RADIX-2 DIT FFT
*      COMPUTATION IN TMS320C40
*
*      THE PROGRAM IS DERIVED FROM THE PAPER BY RAIMUND MEYER AND
*      AND KARL SCHWARZ, VOLUME 3, PROCEEDINGS OF ICASSP 90.
*      THE (COMPLEX) DATA RESIDE IN INTERNAL MEMORY. THE COMPUTATION
*      IS DONE IN-PLACE, BUT THE RESULT IS MOVED TO ANOTHER MEMORY
*      SECTION TO DEMONSTRATE THE BIT-REVERSED ADDRESSING.
*
*      FOR THIS PROGRAM THE MINIMUM FFT LENGTH IS 32 POINTS BECAUSE
*      OF THE SEPARATE STAGES. FIRST TWO PASSES ARE REALIZED AS A
*      FOUR BUTTERFLY LOOP SINCE THE MULTIPLIES ARE TRIVIAL. THE
*      MULTIPLIER IS ONLY USED FOR A LOAD IN PARALLEL WITH AN ADDF
*      OR SUBF.
*
*      THE TWIDDLE FACTORS ARE SUPPLIED IN A TABLE PUT IN A .DATA
*      SECTION. THIS DATA IS INCLUDED IN A SEPARATE FILE TO PRESERVE
*      THE GENERIC NATURE OF THE PROGRAM. FOR THE SAME PURPOSE, THE
*      SIZE OF THE FFT N AND LOG2(N) ARE DEFINED IN A .GLOBL
*      DIRECTIVE AND SPECIFIED DURING LINKING. THE LENGTH OF
*      THE TABLE IS N/2.

```

## Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

```

*
*
      .global    fft
      .global    n
      .global    nhalb
      .global    nviert
      .global    nachtel
      .global    m
      .global    sine
      .BSS       inp,2048          ; input vector length = 2n
                                   ; (depends of n)n)
      .BSS       outp,2048        ; output vector length = 2n
                                   ; (depends of n)n)

      .text
fftsize .word    n
fg4m2  .word    nviert-2
fg4m3  .word    nviert-3
fg8m2  .word    nachtel-2
fg2     .word    nhalb
fg2m3  .word    nhalb-3
logfft  .word    m
sintab  .word    sine
sintml  .word    sine-1
sintp2  .word    sine+2
input   .word    inp
inputp2 .word    inp+2
output  .word    outp

*
*      ar0 : AR + AI
*      ar1 : BR + BI
*      ar2 : CR + CI + CR' + CI'
*      ar3 : DR + DI
*      ar4 : AR' + AI'
*      ar5 : BR' + BI'
*      ar6 : DR' + DI'
*      ar7 : first twiddle factor = 1

fft:   ldp      fftsize          ; load page pointer
      ldi      @fg2,ir0         ; ir0 = n/2 = offset between inputs
      ldi      @sintab,ar7      ; ar7 points to twiddle factor 1
      ldi      @input,ar0       ; ar0 points to AR
      addi     ir0,ar0,ar1      ; ar1 points to BR
      addi     ir0,ar1,ar2      ; ar2 points to CR
      addi     ir0,ar2,ar3      ; ar3 points to DR
      ldi      ar0,ar4          ; ar4 points to AR'
      ldi      ar1,ar5          ; ar5 points to BR'
      ldi      ar3,ar6          ; ar6 points to DR'
      ldi      2,ir1            ; address offset
      lsh     -1,ir0            ; ir0 = n/4 = number of
                                   ; R4-butterflies
      subi     2,ir0,rc

```

Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

\*\*\*\*\*  
 \*-----FIRST 2 STAGES AS RADIX-4 BUTTERFLY-----  
 \*\*\*\*\*

\* fill pipeline

```

    addf    *ar2,*ar0,r4      ; r4 = AR + CR
    subf    *ar2,*ar0++,r5   ; r5 = AR - CR
    addf    *ar1,*ar3,r6     ; r6 = DR + BR
    subf    *ar1++,*ar3++,r7 ; r7 = DR - BR
    addf    r6,r4,r0         ; AR' = r0 = r4 + r6
    mpyf    *ar3++,*ar7,r1   ; r1 = DI , BR' = r3 = r4 - r6
||         r6,r4,r3
    subf    r1,*ar1,r0      ; r0 = BI + DI , AR' = r0
||         stf    r0,*ar4++
    subf    r1,*ar1++,r1    ; r1 = BI - DI , BR' = r3
||         stf    r3,*ar5++
    addf    r1,r5,r2        ; CR' = r2 = r5 + r1
    mpy     *ar2,*ar7,r1    ; r1 = CI , DR' = r3 = r5 - r1
||         subf   r1,r5,r3
    rptbd   blk1           ; Setup for radix-4 butterfly loop
    add     r1,*ar0,r2      ; r2 = AI + CI , CR' = r2
||         stf    r2,*ar2++(ir1)
    subf    r1,*ar0++,r6    ; r6 = AI - CI , DR' = r3
||         stf    r3,*ar6++
    addf    r0,r2,r4        ; AI' = r4 = r2 + r0

```

\* radix-4 butterfly loop

```

||         mpyf    *ar2--,*ar7,r0 ; r0 = CR , (BI' = r2 = r2 - r0)
    subf    r0,r2,r2
||         mpyf    *ar1++,*ar7,r1 ; r1 = BR , (CI' = r3 = r6 + r7)
    addf    r7,r6,r3
    addf    r0,*ar0,r4      ; r4 = AR + CR , (AI' = r4)
||         stf    r4,*ar4++
    subf    r0,*ar0++,r5    ; r5 = AR - CR , (BI' = r2)
||         stf    r2,*ar5++
    subf    r7,r6,r7        ; (DI' = r7 = r6 - r7)
    addf    r1,*ar3,r6      ; r6 = DR + BR , (DI' = r7)
||         stf    r7,*ar6++
    subf    r1,*ar3++,r7    ; r7 = DR - BR , (CI' = r3)
||         stf    r3,*ar2++
    addf    r6,r4,r0        ; AR' = r0 = r4 + r6
    mpyf    *ar3++,*ar7,r1   ; r1 = DI , BR' = r3 = r4 - r6
||         subf   r6,r4,r3
    addf    r1,*ar1,r0      ; r0 = BI + DI , AR' = r0
||         stf    r0,*ar4++
    subf    r1,*ar1++,r1    ; r1 = BI - DI , BR' = r3
||         stf    r3,*ar5++
    addf    r1,r5,r2        ; CR' = r2 = r5 + r1
    mpyf    *ar2,*ar7,r1    ; r1 = CI , DR' = r3 = r5 - r1
||         subf   r1,r5,r3
    addf    r1,*ar0,r2      ; r2 = AI + CI , CR' = r2
||         stf    r2,*ar2++(ir1)
    subf    r1,*ar0++,r6    ; r6 = AI - CI , DR' = r3
||         stf    r3,*ar6++
||         blk1   addf    r0,r2,r4 ; AI' = r4 = r2 + r0

```

## Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

```

* clear pipeline
    subf    r0,r2,r2      ; BI' = r2 = r2 - r0
    addf    r7,r6,r3      ; CI' = r3 = r6 + r7
    st      r4,*ar4      ; AI' = r4 , BI' = r2
||
    stf    r2,*ar5
    subf    r7,r6,r7      ; DI' = r7 = r6 - r7
    stf    r7,*ar6      ; DI' = r7 , CI' = r3
||
    stf    r3,*--ar2

*****
*----- THIRD TO LAST-2 STAGE -----*
*****

    ldi     @fg2,ir1
    subi   1,ir0,ar5
    ldi     1,ar6

    ldi     @sintab,ar7   ; pointer to twiddle factor
    ldi     0,ar4         ; group counter
    ldi     @input,ar0    ; upper real butterfly input
stufe
    ldi     ar0,ar2       ; upper real butterfly output
    addi    ir0,ar0,ar3   ; lower real butterfly output
    ldi     ar3,ar1       ; lower real butterfly input
    lsh     1,ar6         ; double group count
    lsh     -2,ar5        ; half butterfly count
    lsh     1,ar5         ; clear LSB
    lsh     -1,ir0        ; half step from upper to
                        ; lower real part

    lsh     -1,ir1
    addi    1,ir1         ; step from old imaginary to new
                        ; real value
    ldf     *ar1++,r6     ; dummy load, only for address
                        ; update

||
    ld ar7,r7           ; r7 = COS

gruppe

* fill pipeline
*
* ar0 = upper real butterfly input
* ar1 = lower real butterfly input
* ar2 = upper real butterfly output
* ar3 = lower real butterfly output
* the imaginary part has to follow

    ldf     *++ar7,r6     ; r6 = SIN
    mpyf    *ar1--,r6,r1  ; r1 = BI * SIN
||
    addf    *++ar4,r0,r3  ; dummy addf for counter update
    mpyf    *ar1,r7,r0    ; r0 = BR * COS
    mpyf    *ar1++,*ar7--,r0 ; r3 = TR = r0 + r1 , r0 = BR * SIN
||
    addf    r0,r1,r3
    rptbd   bfly1        ; Setup for loop bfly1
    spyf    *ar1++,r7,r1  ; r1 = BI * COS , r2 = AR - TR
||
    subf    r3,*ar0,r2
    add     *ar0++,r3,r5  ; r5 = AR + TR , BR' = r2
||
    stf    r2,*ar3++
    ldi     ar5,rc

```

Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

```

*   FIRST BUTTERFLY-TYPE:
*
*   TR = BR * COS + BI * SIN
*   TI = BR * SIN - BI * COS
*   AR' = AR + TR
*   AI' = AI - TI
*   BR' = AR - TR
*   BI' = AI + TI
*   loop bfly1
||   mpy   *+ar1,r6,r5           ; r5 = BI * SIN , (AR' = r5)
||   stf   r5,*ar2++
||   subf  r1,r0,r2             ; (r2 = TI = r0 - r1)
||   mpyf  *ar1,r7,r0          ; r0 = BR * COS ,
||                                   ; (r3 = AI + TI)
||   addf  r2,*ar0,r3
||   subf  r2,*ar0++,r4        ; (r4 = AI - TI , BI' = r3)
||   stf   r3,*ar3++
||   addf  r0,r5,r3            ; r3 = TR = r0 + r5
||   mpyf  *ar1++,r6,r0        ; r0 = BR * SIN ,
||                                   ; r2 = AR - TR
||   subf  r3,*ar0,r2
||   mpyf  *ar1++,r7,r1        ; r1 = BI * COS , (AI' = r4)
||   stf   r4,*ar2++
bfly1 ||   addf  *ar0++,r3,r5      ; r5 = AR + TR , BR' = r2
||   stf   r2,*ar3++
*   switch over to next group
||   subf  r1,r0,r2            ; r2 = TI = r0 - r1
||   addf  r2,*ar0,r3          ; r3 = AI + TI , AR' = r5
||   stf   r5,*ar2++
||   subf  r2,*ar0++(ir1),r4   ; r4 = AI - TI , BI' = r3
||   stf   r3,*ar3++(ir1)
||   nop   *ar1++(ir1)        ; address update
||   mpyf  *ar1--,r7,r1        ; r1 = BI * COS , AI' = r4
||   stf   r4,*ar2++(ir1)
||   mpyf  *ar1,r6,r0          ; r0 = BR * SIN
||   mpyf  *ar1++,*ar7++,r0    ; r3 = TR = r1 - r0 , r0 = BR
*   COS
||   subf  r0,r1,r3
||   rptbd bfly2              ; Setup for loop bfly2
||   mpyf  *ar1++,r6,r1        ; r1 = BI * SIN ,
||                                   ; r2 = AR - TR
||   subf  r3,*ar0,r2
||   addf  *ar0++,r3,r5        ; r5 = AR + TR , BR' = r2
||   stf   r2,*ar3++
||   ldi   ar5,rc

```

## Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

```

* SECOND BUTTERFLY-TYPE:
*
* TR = BI * COS - BR * SIN
* TI = BI * SIN + BR * COS
,* AR' = AR + TR
* AI' = AI - TI
* BR' = AR - TR
* BI' = AI + TI
*
loop bfly2
mpyf    *+ar1,r7,r5    ; r5 = BI * COS , (AR' = r5)
||      stf          r5,*ar2++
addf    r1,r0,r2      ; (r2 = TI = r0 + r1)
mpyf    *ar1,r6,r0    ; r0 = BR * SIN ,
                    (r3 = AI + TI)
||      addf        r2,*ar0,r3
sub     r2,*ar0++,r4  ; (r4 = AI - TI , BI' = r3)
||      stf          r3,*ar3++
subf    r0,r5,r3      ; TR = r3 = r5 - r0
mpyf    *ar1++,r7,r0  ; r0 = BR * COS , r2 = AR - TR
||      subf        r3,*ar0,r2
mpyf    *ar1++,r6,r1  ; r1 = BI * SIN , (AI' = r4)
||      stf          r4,*ar2++
bfly2   addf        *ar0++,r3,r5 ; r5 = AR + TR , BR' = r2
||      stf          r2,*ar3++
* clear pipeline
addf    r1,r0,r2      ; r2 = TI = r0 + r1
addf    r2,*ar0,r3    ; r3 = AI + TI
||      stf          r5,*ar2++ ; AR' = r5
||      cmpi        ar6,ar4
bnd     gruppe        ; do following 3 instructions
subf    r2,*ar0++(ir1),r4 ; r4 = AI - TI , BI' = r3
||      stf          r3,*ar3++(ir1)
||      ldf          *+ar7,r7 ; r7 = COS
||      stf          r4,*ar2++(ir1) ; AI' = r4
||      nop          *ar1++(ir1) ; branch here
* end of this butterfly group
cmpi    4,ir0        ; jump out after ld(n)-3 stage
bnzaf   stufe
ldi     @sintab,ar7  ; pointer to twiddle factor
ldi     0,ar4        ; group counter
ldi     @input,ar0   ; upper real butterfly input

*****
*-----SECOND LAST STAGE-----*
*****
ldi     @input,ar0    ; upper input
ldi     ar0,ar2      ; upper output
addi    ir0,ar0,ar1  ; lower input
ldi     ar1,ar3      ; lower output
ldi     @sintp2,ar7  ; pointer to twiddle factor
ldi     5,ir0        ; distance between two groups
ldi     @fg8m2,rc

* fill pipeline

```

## Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

```

* 5. to M. butterfly:
*   loop bf2end
||   ld      *ar7++,r7          ; r7 = COS , ((AI' = r4))
||   stf    r4,*ar2++
||   ldf    *ar7++,r6          ; r6 = SIN , (BR' = r2)
||   stf    r2,*ar3++
||   mpyf   *+ar1,r6,r5        ; r5 = BI * SIN , (AR' = r3)
||   stf    r3,*ar2++
||   addf   r1,r0,r2           ; (r2 = TI = r0 + r1)
||   mpyf   *ar1,r7,r0        ; r0 = BR * COS ,
||                                   ; (r3 = AI + TI)
||
||   addf   r2,*ar0,r3
||   sub    r2,*ar0++(ir0),r4  ; (r4 = AI - TI , BI' = r3)
||   stf    r3,*ar3++(ir0)
||   addf   r0,r5,r3           ; r3 = TR = r0 + r5
||   mpyf   *ar1++,r6,r0       ; r0 = BR * SIN , r2 = AR - TR
||   subf   r3,*ar0,r2
||   mpyf   *ar1++,r7,r1       ; r1 = BI * COS , (AI' = r4)
||   stf    r4,*ar2++(ir0)
||   addf   *ar0++,r3,r5       ; r5 = AR + TR , BR' = r2
||   stf    r2,*ar3++
||   mpyf   *+ar1,r6,r5        ; r5 = BI * SIN , (AR' = r5)
||   stf    r5,*ar2++
||   subf   r1,r0,r2           ; (r2 = TI = r0 - r1)
||   mpyf   *ar1,r7,r0        ; r0 = BR * COS ,
||                                   ; (r3 = AI + TI)
||
||   addf   r2,*ar0,r3
||   subf   r2,*ar0++,r4       ; (r4 = AI - TI , BI' = r3)
||   stf    r3,*ar3++
||   addf   r0,r5,r3           ; r3 = TR = r0 + r5
||   mpyf   *ar1++,r6,r0       ; r0 = BR * SIN , r2 = AR - TR
||   subf   r3,*ar0,r2
||   mpyf   *ar1++(ir0),r7,r1  ; r1 = BI * COS , (AI' = r4)
||   stf    r4,*ar2++
||   addf   *ar0++,r3,r3       ; r3 = AR + TR , BR' = r2
||   stf    r2,*ar3++
||   mpyf   *+ar1,r7,r5        ; r5 = BI * COS , (AR' = r3)
||   stf    r3,*ar2++
||   subf   r1,r0,r2           ; (r2 = TI = r0 - r1)
||   mpyf   *ar1,r6,r0        ; r0 = BR * SIN ,
||                                   ; (r3 = AI + TI)
||
||   addf   r2,*ar0,r3
||   sub    r2,*ar0++(ir0),r4  ; (r4 = AI - TI , BI' = r3)
||   stf    r3,*ar3++(ir0)
||   subf   r0,r5,r3           ; r3 = TR = r5 - r0
||   mpyf   *ar1++,r7,r0       ; r0 = BR * COS , r2 = AR - TR
||   subf   r3,*ar0,r2
||   mpyf   *ar1++,r6,r1       ; r1 = BI * SIN , (AI' = r4)
||   stf    r4,*ar2++(ir0)
||   addf   *ar0++,r3,r5       ; r5 = AR + TR , BR' = r2
||   stf    r2,*ar3++

```

## Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

```

||      mpyf      *+ar1,r7,r5          ; r5 = BI * COS , (AR' = r5)
||      stf       r5,*ar2++
||      addf      r1,r0,r2          ; z(r2 = TI = r0 + r1)
||      mpyf      *ar1,r6,r0          ; r0 = BR * SIN ,
||                                       ; r3 = AI + TI)
||      addf      r2,*ar0,r3
||      subf      r2,*ar0++,r4        ; (r4 = AI - TI ,
||                                       ; y(L) = BI' = r3)
||      stf       r3,*ar3++
||      subf      r0,r5,r3          ; r3 = TR = r5 - r0
||      mpyf      *ar1++,r7,r0        ; r0 = BR * COS ,
||                                       ; r2 = AR - TR
||      subf      r3,*ar0,r2
bf2end  mpyf      *ar1++(ir0),r6,r1    ; r1 = BI * SIN ,
||                                       ; r3 = AR + TR
||      addf      *ar0++,r3,r3

* clear pipeline
||      stf       r2,*ar3++          ; BR' = r2 , AI' = r4
||      stf       r4,*ar2++
||      add       r1,r0,r2          ; r2 = TI = r0 + r1
||      add       r2,*ar0,r3        ; r3 = AI + TI , AR' = r3
||      stf       r3,*ar2++
||      subf      r2,*ar0,r4        ; r4 = AI - TI , BI' = r3
||      stf       r3,*ar3
||      stf       r4,*ar2          ; AI' = r4

*****
*----- LAST STAGE -----*
*****
      ldi        @input,ar0          ; upper input
      ldi        ar0,ar2             ; upper output
      ldi        @inputp2,ar1        ; lower input
      ldi        ar1,ar3             ; lower output
      ldi        @sintp2,ar7         ; pointer to twiddle
      ; factors
      ldi        3,ir0
      ldi        @fg4m2,rc

* fill pipeline
* 1. butterfly: w^0
      addf      *ar0,*ar1,r6          ; AR' = r6 = AR + BR
      subf      *ar1++,*ar0++,r7      ; BR' = r7 = AR - BR
      addf      *ar0,*ar1,r4          ; AI' = r4 = AI + BI
      subf      *ar1++(ir0),*ar0++(ir0),r5 ; BI' = r5 = AI - BI

* 2. butterfly: w^M/4
      addf      *+ar1,*ar0,r3          ; AR' = r3 = AR + BI
      ldf       *-ar7,r1             ; r1 = 0 (for inner loop)
||      ldf       *ar1++,r0          ; r0 = BR (for inner loop)
||      rptbd     bflend             ; Setup for loop bflend
||      subf      *ar1++(ir0),*ar0++,r2 ; BR' = r2 = AR - BI
||      stf       r6,*ar2++          ; (AR' = r6)
||      stf       r7,*ar3++          ; (BR' = r7)
||      stf       r5,*ar3++(ir0)     ; (BI' = r5)

```

## Example 12-41. Faster Version Complex, Radix-2 DIT FFT (Continued)

```

* 3. to M. butterfly:
*   loop bflend
    ldf      *ar7++,r7           ; r7 = COS , ((AI' = r4))
||   stf      r4,*ar2++(ir0)
    ldf      *ar7++,r6           ; r6 = SIN , (BR' = r2)
||   stf      r2,*ar3++
    mpyf     *+ar1,r6,r5         ; r5 = BI * SIN ,
                                   ; (AR' = r3)
||   stf      r3,*ar2++
    addf     r1,r0,r2           ; (r2 = TI = r0 + r1)
    mpyf     *ar1,r7,r0         ; r0 = BR * COS ,
                                   ; (r3 = AI + TI)
||   addf     r2,*ar0,r3
    subf     r2,*ar0++(ir0),r4 ; (r4 = AI - TI ,
                                   ; BI' = r3)

||   stf      r3,*ar3++(ir0)
    addf     r0,r5,r3           ; r3 = TR = r0 + r5
    mpyf     *ar1++,r6,r0       ; r0 = BR * SIN ,
                                   ; r2 = AR - TR
||   subf     r3,*ar0,r2
    mpyf     *ar1++(ir0),r7,r1 ; r1 = BI *,COS
                                   ; (AI' = r4)
||   stf      r4,*ar2++(ir0)
    addf     *ar0++,r3,r3       ; r3 = AR + IR, BR' = r2
||   r2,*ar3++
    mpyf     *+ar1,r7,r5         ; r5 = BI * COS ,
                                   ; (AR' = r3)
||   stf      r3,*ar2++
    subf     r1,r0,r2           ; (r2 = TI = r0 - r1)
    mpyf     *ar1,r6,r0         ; r0 = BR * SIN ,
                                   ; (r3 = AI + TI)
||   addf     r2,*ar0,r3
    subf     r2,*ar0++(ir0),r4 ; (r4 = AI - TI ,
                                   ; BI' = r3)
||   stf      r3,*ar3++(ir0)
    subf     r0,r5,r3           ; r3 = TR = r0 - r5
    mpyf     *ar1++,r7,r0       ; r0 = BR * COS
                                   ; r2 = AR - IR
||   subf     r3,*ar0,r2
bflend mpyf     *ar1++(ir0),r6,r1 ; r1 = BI * SIN ,
                                   ; r3 = AR + TR
||   addf     *ar0++,r3,r3

* clear pipeline
    stf      r2,*ar3++           ; BR' = r2 , (AI' = r4)
||   stf      r4,*ar2++(ir0)
    addf     r1,r0,r2           ; r2 = TI = r0 + r1
    addf     r2,*ar0,r3         ; r3 = AI + TI , AR' = r3
||   stf      r3,*ar2++
    subf     r2,*ar0,r4         ; r4 = AI - TI , BI' = r3
||   stf      r3,*ar3
    stf      r4,*ar2           ; AI' = r4

```

**Example 12–41. Faster Version Complex, Radix-2 DIT FFT (Concluded)**

```

*****
*----- END OF FFT -----*
*****

*****
*----- BIT REVERSAL -----*
*****

        ldi        @fftsiz,ir0
        ldi        2,ir1
        ldi        @input,ar0
        ldi        @output,ar1
        ldi        @fftsiz,rc
        subi       2,rc

        ldf        ++ar0(1),r0
        rptb       bitrv
        ldf        *ar0++(ir0)b,r1
||      stf        r0,*ar1(1)
bitrv   ldf        ++ar0(1),r0
||      stf        r1,*ar1++(ir1)
        ldf        *ar0++(ir0)b,r1
||      stf        r0,*ar1(1)
        stf        r1,*ar1

end:     nop
        nop
        nop
        nop

self    br self
        .end

```

The 'C40 quickly executes FFT lengths up to 1024 points (complex) or 2048 (real), covering most applications, because it can do so almost entirely in on-chip memory. Table 12–2 summarizes the execution time required for FFT lengths between 64 and 1024 points for the four algorithms in Example 12–37, Example 12–39, Example 12–40, and Example 12–41.

Table 12-2. TMS320C40 FFT Timing Benchmarks

Number of Points	FFT Timing (In milliseconds)			
	Complex Radix-2 (Example 12-37)	Complex Radix-2 (Example 12-41)	Complex Radix-4 (Example 12-39)	Real Radix-2 (Example 12-40)
64	0.09112	0.0606	0.0694	0.04
128	0.2066	0.13316	—	0.09156
256	0.46288	0.3058	0.36756	0.20712
512	1.02636	0.69208	—	0.45988
1024	2.25544	1.54516	1.82924	1.01984

### 12.4.5 Lattice Filters

The lattice form is an alternative way of implementing digital filters; it has found applications in speech processing, spectral estimation, and other areas. In this discussion, the notation and terminology from speech processing applications are used.

If  $H(z)$  is the transfer function of a digital filter that has only poles,  $A(z) = 1/H(z)$  will be a filter having only zeros, and it will be called the inverse filter. The inverse lattice filter is shown in Figure 12-5. These equations describe the filter in mathematical terms:

$$\begin{aligned} f(i,n) &= f(i-1,n) + k(i) b(i-1,n-1) \\ b(i,n) &= b(i-1,n-1) + k(i) f(i-1,n) \end{aligned}$$

Initial conditions:

$$f(0,n) = b(0,n) = x(n)$$

Final conditions:

$$y(n) = f(p,n).$$

In the above equation,  $f(i,n)$  is the forward error,  $b(i,n)$  is the backward error,  $k(i)$  is the  $i$ -th reflection coefficient,  $x(n)$  is the input, and  $y(n)$  is the output signal. The order of the filter (i.e., the number of stages) is  $p$ . In the linear predictive coding (LPC) method of speech processing, the inverse lattice filter is used during analysis, and the (forward) lattice filter is used during speech synthesis.

Figure 12-5. Structure of the Inverse Lattice Filter

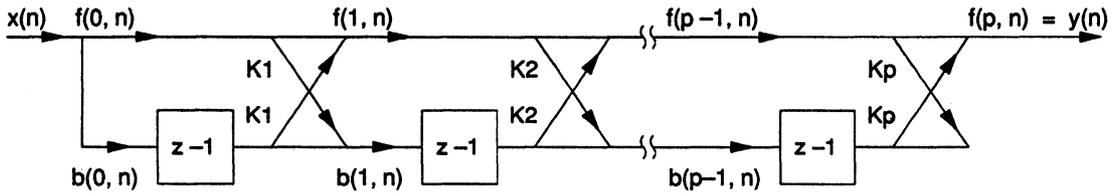
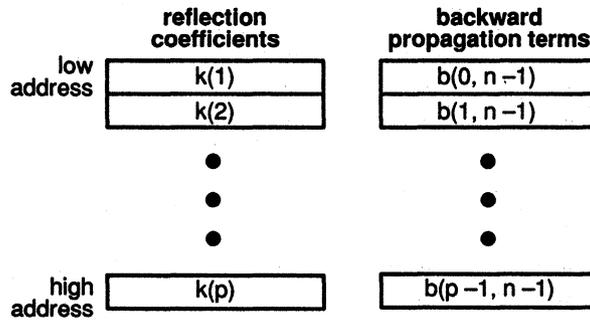


Figure 12-6 shows the data memory organization of the inverse lattice filter on the 'C40.

Figure 12-6. Data Memory Organization for Inverse Lattice Filters



Example 12-42. Inverse Lattice Filter

```

*      TITLE INVERSE LATTICE FILTER
*
*      SUBROUTINE LATINV
*
*      LATINV == LATTICE FILTER (LPC INVERSE FILTER - ANALYSIS)
*
*      TYPICAL CALLING SEQUENCE:
*
*      load      R2
*      LAJU      LATINV
*      load      AR0
*      load      AR1
*      load      RC
*
*      ARGUMENT ASSIGNMENTS:
*      ARGUMENT |  FUNCTION
*      -----+-----
*      R2      |  f(0,n) = x(n)
*      AR0     |  ADDRESS OF FILTER COEFFICIENTS (k(1))
*      AR1     |  ADDRESS OF BACKWARD PROPAGATION VALUES (b(0,n-1))
*      RC      |  RC = p - 2
*
*      REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*      REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*      REGISTER CONTAINING RESULT: R2 (f(p,n))
*
*      PROGRAM SIZE: 11 WORDS
*
*      EXECUTION CYCLES: 5 + 3p
*
*      .global LATINV
*
*      i = 1
*
LATINV  RPTBD      LOOP                ; Setup the delayed repeat
*                                     ; block loop
*      MPYF3      *AR0,*AR1,R0          ; k(1) * b(0,n-1) -> R0
*                                     ; Assume f(0,n) -> R2.
*      LDF        R2,R3                ; Put b(0,n) = f(0,n) -> R3.
*      MPYF3      *AR0++(1),R2,R1      ; k(1) * f(0,n) -> R1
*

```

```

*      2 <= i <= p (Repeat block loop start here)
*
MPYF3    *AR0, *++AR1(1), R0    ; k(i) * b(i-1, n-1) -> R0
|| ADDF3    R2, R0, R2          ; f(i-1-1, n) + k(i-1) * b(i-1-1, n-1)
*                                           ; = f(i-1, n) -> R2
*
*                                           ; b(i-1-1, n-1) + k(i-1) * f(i-1-1, n)
ADDF3    *-AR1(1), R1, R3      ; = b(i-1, n) -> R3
|| STF     R3, *-AR1(1)        ; b(i-1-1, n) -> b(i-1-1, n-1)
*
*                                           ; k(i) * f(i-1, n) -> R1
LOO MPYF3    *AR0++(1), R2, R1
*
*      I = P + 1 (CLEANUP)
*
BUD      R11                    ; Delayed return
ADDF3    R2, R0, R2             ; f(p-1, n) + k(p) * b(p-1, n-1)
*                                           ; = f(p, n) -> R2
*
ADDF3    *AR1, R1, R3           ; b(p-1, n-1) + k(p) * f(p-1, n)
*                                           ; = b(p, n) -> R3
|| STF    R3, *AR1              ; b(p-1, n) -> b(p-1, n-1)
*
*      end
*
*      .end

```

The structure of the forward lattice filter, shown in Figure 12-7, is similar to that of the inverse filter (also shown in the figure). These corresponding equations describe the lattice filter:

$$\begin{aligned}
 f(i-1, n) &= f(i, n) - k(i) b(i-1, n-1) \\
 b(i, n) &= b(i-1, n-1) + k(i) f(i-1, n)
 \end{aligned}$$

Initial conditions:

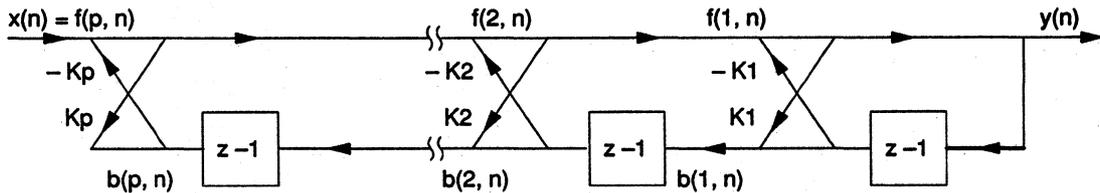
$$f(p, n) = x(n), b(i, n-1) = 0 \quad \text{for } i = 1, \dots, p$$

Final conditions:

$$y(n) = f(0, n).$$

The data memory organization is identical to that of the inverse filter, as shown in Figure 12-6. Example 12-43 shows the implementation of the lattice filter on the 'C40.

Figure 12-7. Structure of the (Forward) Lattice Filter



Example 12-43. Lattice Filter

```

*      TITLE LATTICE FILTER
*
*      SUBROUTINE LATTICE
*
*      LAJU      LATTICE
*      LOAD      AR0
*      LOAD      AR1
*      LOA       RC
*
*
*      ARGUMENT ASSIGNMENTS:
*      ARGUMENT | FUNCTION
*      -----+-----
*      R2       | F(P,N) = E(N) = EXCITATION
*      AR0      | ADDRESS OF FILTER COEFFICIENTS (K(P))
*      AR1      | ADDRESS OF BACKWARD PROPAGATION
*              | VALUES (B(P-1,N-1))
*      RC       | RC = P - 2
*
*      REGISTERS USED AS INPUT: R2, AR0, AR1, RC
*      REGISTERS MODIFIED: R0, R1, R2, R3, RS, RE, RC, AR0, AR1
*      REGISTER CONTAINING RESULT: R2 (f(0,n))
*
*      PROGRAM SIZE: 13 WORDS
    
```

Concluded on next page

## Example 12-43. Lattice Filter (Concluded)

```

*          EXECUTION CYCLES: 3 + 5P
*
*          .global  LATTICE
*
*
LATTICE  RPTBD      LOOP                ; Setup the delayed repeat
                                           ; block loop
MPYF3    *AR0,*AR1,R0                    ; K(P) * B(P-1,N-1) -> R0
SUBF3    R0,R2,R2                        ; Assume F(P,N) -> R2
NOP                                             ; F(P,N)-K(P)*B(P-1,N-1)
                                           ; = F(P-1,N) -> R2
*
*          2 <= I <= P (Repeat block loop start here)
*
MPYF3    *AR0,R2,R1                      ; K(I) * F(I-1,N) -> R1
MPYF3    * - -AR0(1),*-AR1(1),R0        ; K(I-1) *
                                           ; B(I-1-1,N-1) -> R0
ADDF3    *AR1 - -(1),R1,R3              ; B(I-1,N-1) + K(I)*F(I-1,N)
                                           ; = B(I,N) -> R3
*
LOOP     STF      R3,*+AR1(2)            ; B(I,N) -> B(I,N-1)
SUBF3    R0,R2,R2                        ; F(I-1,N)-K(I-1)
                                           ; *B(I-1-1,N-1)
                                           ; = F(I-1-1,N) -> R2
*
*
*          I = 1 (CLEANUP)
*
BUD      R11                              ; Delayed return
MPYF     *AR0,R2,R1                        ; K(1) * F(0,N) -> R1
ADDF3    *AR1,R1,R3                        ; B(0,N-1) + K(1)*F(0,N)
                                           ; = B(1,N) -> R3
*
STF      R3,*+AR1(1)                      ; B(1,N) -> B(1,N-1)
STF      R2,*AR1                          ; F(0,N) -> B(0,N-1)
*
*          end
*
*          end

```

## 12.5 Programming Tips

Programming style is highly personal and reflects each individual's preferences and experiences. The purpose of this section is not to impose any particular style. Instead, it emphasizes some of the features of the 'C40 that can help in producing faster and/or shorter programs. The tips cover both C compiler and assembly language programming.

### 12.5.1 C-Callable Routines

The 'C40 was designed with a large register file, software stack, and large memory space in order to implement a high-level language (HLL) compiler easily. The first such implementation supplied is a C compiler. Use of the C compiler increases the transportability of applications that have been tested on large, general-purpose computers and decreases their porting time.

To use the compiler efficiently, complete the following steps:

- 1) Write the application in the high-level language.
- 2) Debug the program.
- 3) Estimate if it runs in realtime.
- 4) If it doesn't, identify places where most of the execution time is spent.
- 5) Optimize these areas by writing assembly language routines that implement the functions.
- 6) Call the routines from the C program as C functions.

When writing a C program, you can increase the execution speed by maximizing the use of register variables. For more information, refer to the *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* (literature number SPRU034, due for release 3Q, 1991).

Certain conventions must be observed in writing a C-callable routine. These conventions are outlined in the *Runtime Environment* chapter of the *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide*. Certain registers are saved by the calling function, and others need to be saved by the called function. The C compiler manual helps achieve a clean interface. The end result is the readability and natural flow of a high-level language combined with the efficiency and special-feature use of assembly language.

## 12.5.2 Hints for Optimizing Assembly Code

Each program has particular requirements. Not all possible optimizations will make sense in every case. The suggestions presented in this section can be used as a checklist of available software tools.

- ❑ **Use delayed branches.** Delayed branches execute in a single cycle; regular branches execute in four. The three instructions that follow the delayed branch are executed whether the branch is taken or not. If fewer than three instructions are used, use the delayed branch and append NOPs. Machine cycles (time) are still being saved.
- ❑ **Use delayed subroutine call and return.** Regular subroutine CALL and RETS execute in four cycles. The delayed subroutine call can be achieved by using link and jump (LAJ) and delayed branches with R11 register mode (BUD R11) instructions. Both LAJ and BUD instructions execute in a single cycle. The rule for using LAJ instruction is the same as for delayed branches.
- ❑ **Apply the repeat single/block construct.** In this way, loops are achieved with no overhead. Nesting such constructs will not normally increase efficiency, so try to use the feature on the most often performed loop. The RPTBD is a single-cycle instruction, and the RPTS and RPTB are four-cycle instructions. The usage of RPTBD is similar to that of the delayed branches. Note that RPTS is not interruptible, and the executed instruction is not refetched for execution. This frees the buses for operands.
- ❑ **Use parallel instructions.** It is possible to have a multiply in parallel with an add (or subtract) and to have stores in parallel with any multiply or ALU operation. This increases the number of operations executed in a single cycle. For maximum efficiency, observe the addressing modes used in parallel instructions and arrange the data appropriately. You can have loads in parallel with any multiply or add (or subtract). Since the result of a multiply by one or an add of zero is the same as a load, parallel instructions with a data load can be implemented by substituting the load instruction with a multiply or an add instruction with one extra register containing a one or zero.

- ❑ **Maximize the use of registers.** The registers are an efficient way to access scratch-pad memory. Extensive use of the register file facilitates the use of parallel instructions and helps avoid pipeline conflicts when you use register addressing (register addressing is described in subsection 5.1.1 on page 5-3).
- ❑ **Use the cache.** Use cache especially in conjunction with slow external memory. The cache is transparent to the user, so make sure that it is enabled.
- ❑ **Use internal memory instead of external memory.** The internal memory (2K×32 bits RAM and 4K×32 bits ROM) is considerably faster to access. In a single cycle, two operands can be brought from internal memory. You can maximize performance if you use the DMA in parallel with the CPU to transfer data to internal memory before you operate on them.
- ❑ **Avoid pipeline conflicts.** If there is no problem with program speed, ignore this suggestion. For time-critical operations, make sure that cycles are not missed because of conflicts. To identify conflicts, run the trace function on the development tools (simulator, emulators) with the program tracing option enabled. The tracing immediately identifies the pipeline conflicts. Consult the appropriate section of this user's guide for an explanation of the reason for the conflict. You can then take steps to correct the problem.

The above checklist is not exhaustive, and it does not address some features outlined in more detail in the different sections of this manual. To learn how to exploit the full power of the 'C40, carefully study its architecture, hardware configuration, and instruction set described in this user's guide.

## 12.6 Peripherals

TMS320C40 peripheral modules include one analysis module, two timers, six direct memory access (DMA) controllers, and six high speed bi-directional communication ports. They are designed to improve system performance and decrease system cost without reducing the computational throughput of the CPU. These peripheral modules are controlled through memory-mapped registers located on the dedicated peripheral bus. The examples that show how to program the timer, communication port, and DMA operations are presented in the following subsections.

### 12.6.1 Timers

There are two general-purpose, 32-bit timers on the 'C40 device. Both timers are identical to and independent from each other (detailed information on the timers is in Section 9.10 on page 9-45). The timers are controlled by three registers: timer global control register, timer counter register, and timer period register. Pins TCLK0 and TCLK1 of 'C40 are dedicated for timers. These pins can be configured as either general-purpose data I/O or timer.

If bit 0 and bit 9 of the timer global control register are set to 0, the TCLKx pin is configured as a general-purpose data I/O pin. Timer counter and period registers have no effect on this configuration. Bit 1 of the timer global control register is used to configure TCLKx as an input or output pin. If TCLKx is configured as an output pin (bit 1 = 1), the data value in bit 2 of the timer global control register is shown on TCLKx. If TCLKx is configured as an input pin (bit 1 = 0), the signal on TCLKx is shown in bit 3 of the timer global control register.

If bit 0 of the timer global control register is set to 1, pin TCLKx is configured as a timer pin. The frequency of the timer signaling is specified by the timer period register. However, this assumes that the timer counter register equals 0 (writing 1 to bit 6 of the timer global control register will reset the counter register, too). If the timer counter register has a nonzero value in it, the first period will be different than the others. When the counter register is set to a value greater than the period register, the counter will count, roll over to 0, and continue counting to period register. Therefore, it is important to have correct values in the timer period and counter registers before starting the timer (writing a 1 to bit 7 of the timer global control register starts the timer).

The frequency of the timer signaling is determined by the frequency of the timer input clock and the period register. The following equations are valid with either an internal or an external timer clock:

$$f(\text{pulse mode}) = f(\text{timer clock}) / \text{period register}$$

$$f(\text{clock mode}) = f(\text{timer clock}) / (2 \times \text{period register})$$

When the period and counter register are zero, the operation of the timer is dependent upon the  $C/\bar{P}$  mode selected. In pulse mode ( $C/\bar{P} = 0$ ), TSTAT is set and remains set. In the other words, the frequency is equal to infinite. In clock mode ( $C/\bar{P} = 1$ ), the width of the cycle is  $2/f(H1)$ , and the external clock is ignored. Therefore, the maximum frequency of timer clock generated by internal clock is  $f(H1)/2$ . Example 12–44 shows how to set up the 'C40 timer to generate the maximum frequency clock through the TCLKx pin.

*Example 12–44. Maximum Frequency Timer Clock Setup*

```
*          TITLE MAXIMUM FREQUENCY TIMER CLOCK SETUP
*
*          THIS EXAMPLE SHOWS HOW TO SET UP TIMER TO GENERATE MAXIMUM
*          FREQUENCY TIMER CLOCK USING INTERNAL CLOCK. WHERE
*          "TIMER_REGISTER" SECTION IS LOCATED FROM 808020H.
*
TIMO_CTL_REG      .usect      "TIMER_REGISTER", 4
TIMO_CNT_REG      .usect      "TIMER_REGISTER", 4
TIMO_PRD_REG      .usect      "TIMER_REGISTER", 8
                .text
                .
                .
                LDI          0, R0
                STI          R0, @TIMO_PRD_REG
                LDI          3C1H, R0
                STI          R0, @TIMO_CTL_REG
                .
                .
                .end
```

### 12.6.2 Communication Ports

In order to provide direct processor-to-processor communication, 'C40 has six parallel bidirection communication ports (see Chapter 8). Since these ports have port arbitration units to handle the ownership of the communication port data bus between the processors, the programmer needs to concentrate only on the internal operation of the communication ports. For software, these communication ports can be treated as 32-bit on-chip data I/O FIFO buffers. Processor read/write data from/to communication is simple:

```
LDI @comm_port0_input,R0 ; Read data from comm. port 0  
or  
STI R0,@comm_port0_output ; Write data to comm. port 1
```

If the CPU or DMA reads from or writes to the communication port I/O FIFO and the I/O FIFO is either empty (on a read) or full (on a write), the read/write execution will be extended until the data is available in the input FIFO for a read, or the space is available in the output FIFO for a write. Sometimes, this can be used to synchronize the devices. However, this will slow down the processing speed and even hang up the processor. Avoid such situations.

Each 'C40 communication port provides four flags to indicate the status of the port:

- ICRDY** (input channel ready)
  - = 0, the input channel is empty and not ready to be read.
  - = 1, the input channel contains data and is ready to read.
- ICFULL** (input channel full)
  - = 0, the input channel is not full.
  - = 1, the input channel is full.
- OCRDY** (output channel ready)
  - = 0, the output channel is full and not ready to be written.
  - = 1, the output channel is not full and ready to be written.
- OCEMPTY** (output channel empty)
  - = 0, the output channel is not empty.
  - = 1, the output channel is empty.

These flags can be used to synchronize the CPU/DMA access to the communication port. Example 12–45 shows reading data from the communication port eight data at a time using the CPU ICFULL interrupt. Example 12–46 shows writing data to a communication port one datum at a time using the polling method. The example shows DMA reads/writes of data from/to the communication port (DMA is discussed in the next subsection, subsection 12.6.3).

**Example 12–45. Read Data from Communication Port With CPU ICFULL Interrupt**

```

*
* TITLE READ DATA FROM COMMUNICATION PORT WITH CPU
* ICFULL INTERRUPT
*
* THIS EXAMPLE ASSUMES THE ICFULL 0 INTERRUPT VECTOR IS SET IN THE
* CPU INTERRUPT VECTOR TABLE. THE EIGHT DATA ARE READ IN
* WHENEVER THE DATA IS FULL IN COMM PORT 0 INPUT FIFO.
*
.
.
LDA      @COMM_PORT0_CTL,AR2    ; Load comm port 0
                                ; control reg address
LDA      @COMM_PORT0_INPUT,AR0 ; Load comm port 0
                                ; input FIFO address
LDA      @INTERNAL_RAM,AR1     ; Load internal RAM address
AND3    0F7H,*AR2,R9          ; Unhalt comm port 0
                                ; input channel
STI      R9,*AR2
OR       04H,IIE              ; Enable ICRDY 0 interrupt
OR       02000H,ST            ; Enable CPU global interrupt
.
.
ICFULL0  PUSH      ST
        PUSH      RS
        PUSH      RE
        PUSH      RC
        RPTBD     READ          ; Setup for loop READ
        LDI       6,RC          ; Set repeat counter
        LDI       *AR0,R10     ; Read data from comm port 0
                                ; input
        NOP
READ     LDI       *AR0,R10     ; Read data from comm port 0
                                ; input
||      STI       R10,*AR1++(1) ; Store data into internal RAM
        STI       R10,*AR1++(1) ; Store data into internal RAM
        POP      RC
        POP      RE
        POP      RS
        POP      ST
        RETI

```

**Example 12–46. Write Data to Communication Port With Polling Method**

```

*
*      TITLE WRITE DATA TO COMMUNICATION PORT WITH POLLING METHOD
*
*      THE BIT 8 OF COMMUNICATION PORT 0 CONTROL REGISTER WILL BE
*      SET ONLY WHEN THE OUTPUT FIFO IS FULL. THIS EXAMPLE CHECKS
*      THIS BIT TO MAKE SURE THERE IS SPACE AVAILABLE IN
*      OUTPUT FIFO.
*
      .
      .
      .
      LDA    @COMM_PORT0_CTL,AR2    ; Load comm port 0 control reg
                                   ; address
      LDA    @COMM_PORT0_OUTPUT,AR0; Load comm port 0 output
                                   ; FIFO address
      LDA    @INTERNAL_RAM,AR1     ; Load internal RAM address
      AND3   0EFH,*AR2,R9         ; Unhalt comm port 0 output
                                   ; channel
      STI    R9,*AR2
      LDI    0100H,R9             ; Load mask for bit 8
WAIT:      TSTB   *AR2,R9         ; Check if output FIFO is full
      BZD    WAIT                ; If yes, check again
WRITE_COMM LDI    *AR1++(1), R10  ; Read data from internal RAM
      STI    R10,*ARO           ; Store data into comm port
                                   ; 0 output
      .
      .

```

**12.6.3 Direct Memory Access**

The 'C40 direct memory access (DMA) coprocessor supports six DMA channels (detailed information on DMA is in Chapter 9). These channels perform transfers to and from anywhere in the processor memory map. The DMA coprocessor is a self-programming device that allows data transfers to occur without any intervention from the CPU. It also provides a special split-mode to support 12 DMA channels for communication port memory transfer. This section contains examples of DMA programs from a very simple single-block memory-to-memory transfer to a sophisticated memory transfer with autoinitialization.

Example 12–47 shows one way for setting up DMA channel 2 to initialize an array to zero. This DMA transfer is set up to have higher priority over a CPU operation and to generate an interrupt flag, DMA INT2, after the transfer is completed. The DMA control register is set to 03040007H (refer to DMA control register bit functions in Table 9–1 on page 9-8 for further information on this setup).

*Example 12–47. Array initialization With DMA*

```

*
*           TITLE ARRAY INITIALIZATION WITH DMA
*
*           THIS EXAMPLE INITIALIZES A 128 ELEMENTS ARRAY TO ZERO. THE DMA
*           TRANSFER IS SET UP TO HAVE HIGHER PRIORITY OVER CPU OPERATION.
*           THE DMA INT2 INTERRUPT FLAG IS SET TO 1 AFTER THE TRANSFER IS
*           COMPLETED.
*
DMA2      .data
CONTROL  .word    001000C0H      ; DMA channel 2 map address
SOURCE   .word    00C40007H      ; DMA register initialization data
SRC_IDX  .word    ZERO
COUNT  .word    0
DESTIN   .word    ARRAY
DES_IDX  .word    1
ZERO     .word    0.0            ; Array initialization value 0.0
        .bss      ARRAY,128
        .text
START    LDP      @DMA2          ; Load data page pointer
        LDA      @DMA2,AR0      ; Point to DMA channel 2 registers
        LDI      @SOURCE,R0     ; Initialize DMA source register
        STI      R0,*+AR0(1)
        LDI      @SRC_IDX,R0    ; Initialize DMA source index
        ; register
        STI      R0,*+AR0(2)
        LDI      @COUNT,R0    ; Initialize DMA count register
        STI      R0,*+AR0(3)
        LDI      @DESTIN,R0    ; Initialize DMA destination
        ; register
        STI      R0,*+AR0(4)
        LDI      @DES_IDX,R0   ; Initialize DMA destination
        ; index register
        STI      R0,*+AR0(5)
        LDI      @CONTROL,R0   ; Start DMA channel 2 transfer
        STI      R0,*AR0
        .end

```

The DMA transfer can be synchronized with external interrupts, communication port ICRDY/OCRDY signals, and timer interrupts. In order to enable this feature, the SYNCH MODE field, bits 6–7, of the DMA control register must be configured to a proper value (Table 9–1 on page 9-8), and the corresponding bits of the DMA interrupt enable (DIE) register must be set. Example 12–48 sets up DMA channel 4 read synchronization with the communication port ICRDY signal. The DMA is set up to continuously transfer data from the communication port input register until the START field, bits 22–23 of the DMA control register, is changed by the CPU.

*Example 12–48. DMA Transfer With Communication Port ICRDY Synchronization*

```

*
*      TITLE      DMA TRANSFER WITH COMMUNICATION PORT ICRDY
*                  SYNCHRONIZATION
*
*      THIS EXAMPLE SETS UP DMA CHANNEL 4 TO TRANSFER DATA FROM
*      COMMUNICATION PORT INPUT REGISTER TO INTERNAL RAM WITH ICRDY
*      SIGNAL READ SYNCHRONIZATION. THE TRANSFER MODE OF THE DMA IS
*      SET TO 00. THEREFORE THE TRANSFER WON'T STOP UNTIL THE START
*      BITS OF THE DMA CONTROL REGISTER IS CHANGED.
*
*
DMA4      .data
CONTROL  .word    001000E0H      ; DMA channel 4 map address
SOURCE   .word    00C00040H      ; DMA register initialization data
SRC_IDX  .word    00100081H
COUNT  .word    0                ; Transfer counter is set to
*                                ; largest value
DESTIN   .word    002FF800H
DES_IDX  .word    1

START    .text
LDP      @DMA4                ; Load data page pointer
LDA      @DMA4,AR0            ; Point to DAM channel 4 registers
LDI      @SOURCE,R0           ; Initialize DMA source register
STI      R0,*,+AR0(1)
LDI      @SRC_IDX,R0          ; Initialize DMA source index register
STI      R0,*,+AR0(2)
LDI      @COUNT,R0          ; Initialize DMA count register
STI      R0,*,+AR0(3)
LDI      @DESTIN,R0           ; Initialize DMA destination register
STI      R0,*,+AR0(4)
LDI      @DES_IDX,R0          ; Initialize DMA destination index
*                                ; register
STI      R0,*,+AR0(5)
LDI      @CONTROL,R0          ; Start DMA channel 4 transfer
STI      R0,*,AR0

LDHI     010H,DIE             ; Enable ICRDY 4 read sync.
.end

```

If external interrupt signals are used for DMA transfer synchronization, then pins IIOF0-3 must be configured as interrupt pins also.

The 'C40 DMA split mode is another way besides memory map address to transfer data from/to the communication port. When the split-mode bit of the DMA control register is set, the DMA is separated into primary and auxiliary channels. The primary channel transfers data from memory to the communication port output register, and the auxiliary channel transfers data from the communication port to memory. The communication port number is selected in bits 15 – 17 of the DMA control register.

Example 12–49 shows how to set up DMA channel 1 into split mode. The DMA primary channel transfers data from internal RAM to communication port 3 using external interrupt INT2 synchronization and bit-reversed addressing. The DMA auxiliary channel transfers data from communication port 3 to internal RAM using external interrupt INT3 synchronization and linear addressing.

*Example 12–49. DMA Split-Mode Transfer With External Interrupt Synchronization*

```

*
*      TITLE   DMA SPLIT-MODE TRANSFER WITH EXTERNAL INTERRUPT
*              SYNCHRONIZATION
*
*      THIS EXAMPLE SETS UP DMA CHANNEL 1 TO SPLIT-MODE. THE PRIMARY
*      CHANNEL TRANSFERS DATA FROM INTERNAL RAM TO COMM PORT 3 OUTPUT
*      REGISTER WITH EXTERNAL INTERRUPT INT2 SYNCHRONIZATION AND BIT-
*      REVERSED ADDRESSING. THE AUXILIARY CHANNEL TRANSFERS DATA FROM
*      COMMUNICATION PORT 3 INPUT REGISTER TO INTERNAL RAM WITH
*      EXTERNAL INTERRUPT INT3 SYNCHRONIZATION AND LINEAR ADDRESSING.
*
*
*      .data
DMA1      .word    001000B0H      ; DMA channel 1 map address
CONTROL  .word    03CDD0D4H      ; DMA register initialization data
SOURCE   .word    002FFC00H
SRC_IDX  .word    08H           ; The same value as IR0 for bit-reversed
COUNT  .word    8
DESTIN   .word    002FF800H
DES_IDX  .word    1
AUX_CNT  .word    8
*
*      .text
STAR     LDP      @DMA1          ; Load data page pointer
          LDA      @DMA1,AR0      ; Point to DAM channel 1 registers
          LDI      @SOURCE,R0     ; Initialize DMA primary source register
          STI      R0,++AR0(1)
          LDI      @SRC_IDX,R0    ; nititalize DMA primary source index reg
          STI      R0,++AR0(2)
          LDI      @COUNT,R0    ; Initialize DMA primary count register
          STI      R0,++AR0(3)
          LDI      @DESTIN,R0    ; Initialize DMA aux destination
          ; register

```

```

STI      R0, *+AR0 (4)
LDI      @DES_IDX, R0      ; Initialize DMA aux destination
                                ; index register

STI      R0, *+AR0 (5)
LDI      @AUC_CNT, R0     ; Initialize DMA auxiliary count
                                ; register

STI      R0, *+AR0 (7)
LDI      @CONTROL, R0    ; Start DMA channel 1 transfer
STI      R0, *AR0

LDI      01100H, IIF     ; Configure INT2 and INT3 as
                                ; interrupt pins

LDI      0A0H, DIE      ; Enable INT2 read and INT3 write sync.
.end

```

An advantage of the 'C40 DMA is the autoinitialization feature. This allows you to set up the DMA transfer in advance and makes the DMA operation 100 percent independent from the CPU. When the DMA is operating in autoinitialization mode, the link pointer and auxiliary link pointer are used to initialize the registers that control the DMA operation. The link pointer may be incremented (AUTOINIT STATIC = 0 — shown in Table 9–1 on page 9-8) during autoinitialization or held constant (AUTOINIT STATIC = 1) during autoinitialization. This option allows autoinitialization values to be stored in sequential memory locations or in stream-oriented devices such as the on-chip communication ports or external FIFOs. When DMA SYNC MODE is enabled, The DMA autoinitialization operation can be configured to synchronize with the same signal too. Example 12–50 sets up DMA channel 0 to wait for the communication port to input the initialization value. After DMA autoinitialization is complete, the DMA channel starts transferring data from the communication port input register to internal RAM.

Example 12-50. DMA Autoinitialization With Communication Port ICRDY

```

*
*      TITLE DMA AUTOINITIALIZATION WITH COMMUNICATION PORT ICRDY
*
*      THIS EXAMPLE SETS UP DMA CHANNEL 0 TO WAIT FOR COMMUNICATION
*      PORT TO INPUT THE INITIALIZATION VALUE. THE DMA AUTOINITIAL-
*      IZATION AND TRANSFER ARE BOTH DRIVEN BY ICRDY 0 FLAG. AFTER
*      DMA AUTOINIT IS COMPLETED, THE DMA CHANNEL STARTS TRANSFERRING
*      DATA FROM COMM PORT INPUT REGISTER TO INTERNAL RAM WITH ICRDY
*      0 READ SYNCHRONIZATION. THE VALUES IN COMM PORT 0 INPUT FIFO
*      SHOULD BE:
*
*      SEQUENCE | VALUE
*      -----+-----
*      1        | 00C40047H (STOP AFTER TRANSFER COMPLETED)
*              | OR 00C4054BH (REPEAT AFTER TRANSFER COMPLETED)
*      2        | 00100041H
*      3        | 0H
*      4        | 20H
*      5        | 002FF800H
*      6        | 1H
*      7        | 00100041H
*
*      .data
DMA0      .word    001000A0H      ; DMA channel 0 map address
DMA_INIT  .word    0004054BH      ; DMA initialization control word
LINK      .word    00100041H      ; Comm port input register address
DMA_START .word    00C4054BH      ; DMA start control word
*      .text
START     LDP      @DMA0          ; Load data page pointer
          LDA      @DMA0,AR0      ; Point to DMA channel 0 registers
          LDI      @DMA_INIT,R0   ; Initialize DMA control register
          STI      R0,*AR0
          LDI      @LINK,R0       ; Initialize DMA link pointer
          STI      R0,*+AR0(6)
          LDI      @DMA_START,R0  ; Start DMA channel 0 transfer
          STI      R0,*AR0
          LDI      01H,DIE        ; Enable ICRDY 0 read sync.
          .end

```

The DMA autoinitialization and transfer will continue executing if the DMA autoinitialization is still enabled. Therefore, a DMA setup like the one in Example 12-50 can make it possible for the DMA operation to be controlled by an external device through the communication port.

With the autoinitialization feature, the 'C40 DMA can support a variety of DMA operations without slowing down CPU computation. A good example is a DMA transfer triggered by one interrupt signal. Usually, this is achieved by starting a DMA activity with a CPU interrupt service routine, but this utilizes CPU time. However, with the autoinitialization feature, 'C40 DMA can achieve this kind of setup without CPU interruption, as shown in Example 12–51. One method is to set up a single interrupt-driven dummy DMA transfer with autoinitialization. When the interrupt signal is set, the DMA will complete the dummy DMA transfer and start the autoinitialization for the desired DMA transfer.

*Example 12–51. Single-Interrupt-Driven DMA Transfer*

```

*
*      TITLE SINGLE INTERRUPT-DRIVEN DMA TRANSFER
*
*      THIS EXAMPLE SETS UP A DUMMY DMA TRANSFER FROM INTERNAL RAM
*      TO THE SAME MEMORY WITH EXTERNAL INT 0 SYNCHRONIZATION AND
*      AUTOINITIALIZATION FOR TRANSFERRING 64 DATA FROM LOCAL MEMORY
*      TO INTERNAL RAM. AFTER THE SECOND TRANSFER IS COMPLETED, THE
*      DMA IS RE-INITIALIZED TO FIRST DMA RANSFER SETUP.
*
      .data
DMA5      .word    001000F0H      ; DMA channel 5 map address
DMA_INIT .word    0000004BH      ; DMA initialization control word
LINK     .word    DMA1          ; 1st DMA link list address
DMA_START .word   00C0004BH     ; DMA start control word
DMA1     .word    00C0004BH     ; 1st dummy DMA transfer link list
          .word    002FF800H
          .word    00000000H
          .word    00000001H
          .word    002FF800H
          .word    00000000H
          .word    DMA2
DMA2     .word    00C4000BH     ; The desired DMA transfer link
          .word    00400000H     ; list
          .word    00000001H
          .word    00000040H
          .word    002FF800H
          .word    00000001H
          .word    DMA1
      .text
START    LDP      @DMA5          ; Load data page pointer
          LDA      @DMA5,AR0     ; Point to DMA channel 5 registers
          LDI      @DMA_INIT,R0  ; Initialize DMA control register
          STI      R0,*AR0
          LDI      @LINK,R0     ; Initialize DMA link pointer
          STI      R0,**AR0(6)
          LDI      @DMA_START,R0 ; Start DMA channel 5 transfer
          STI      R0,*AR0
          LDI      01H,IIF      ; Configure INT0 as interrupt pins
          LDHI     0800H,DIE     ; Enable INT 0 read sync. for
          ; DMA channel 5
      .end

```



# Hardware Applications

The TMS320C40's advanced interface design can be used to implement a wide variety of system configurations. Its two external buses and DMA capability provide a flexible parallel 32-bit interface to byte- or word-wide devices; the communication ports provide a glueless interface to other 'C40s; and the interrupt interface, communication ports, and general-purpose digital I/O provide communication with a multitude of peripherals.

This chapter describes how to use the 'C40's interfaces to connect to various external devices. Specific discussions include implementation of parallel interface to devices with and without wait states, parallel processing through the communication ports and port control logic, and system control function circuit design.

Major topics discussed in this chapter are as follows:

<b>Section</b>	<b>Page</b>
13.1 System Configuration Options Overview .....	13-3
■ Categories of Interfaces on the TMS320C40 .....	13-3
13.2 Boot Loader Description and External ROM Interfacing ...	13-5
■ TMS320C40 Boot Loader Description .....	13-5
■ Examples of External Memory Loads .....	13-8
■ Communication Port Loading .....	13-8
■ External ROM Interfacing to the TMS320C40 .....	13-9
■ External Memory Loading .....	13-14
■ TMS320C40 Boot Loader Source Program .....	13-14
13.3 Global and Local Bus Interface .....	13-20
■ Zero Wait-State Interface to RAMs .....	13-20
13.4 Ready Generation .....	13-27
■ ORing of the Ready Signals (SWW = 10) .....	13-28

■	ANDing of the Ready Signals (SWW = 11) .....	13-28
■	External Ready Generation .....	13-29
■	Ready Control Logic .....	13-30
■	Example Circuit .....	13-31
■	Page Switching Techniques .....	13-32
13.5	Parallel Processing Interfaces .....	13-37
■	Message Broadcasting From a TMS320C40 to Many TMS320C40's .....	13-37
■	Shared Global Memory Interface With Fair Arbitration .	13-38
■	Shared Bus Interface Overview .....	13-43
13.6	Bus Arbitration .....	13-48
■	Arbitration Implementation .....	13-48
■	Global Bus Arbitration and Transfer Timing .....	13-70
■	Arbitration Protocol Limitations .....	13-70
13.7	Reset Signal Generation Control Functions .....	13-75

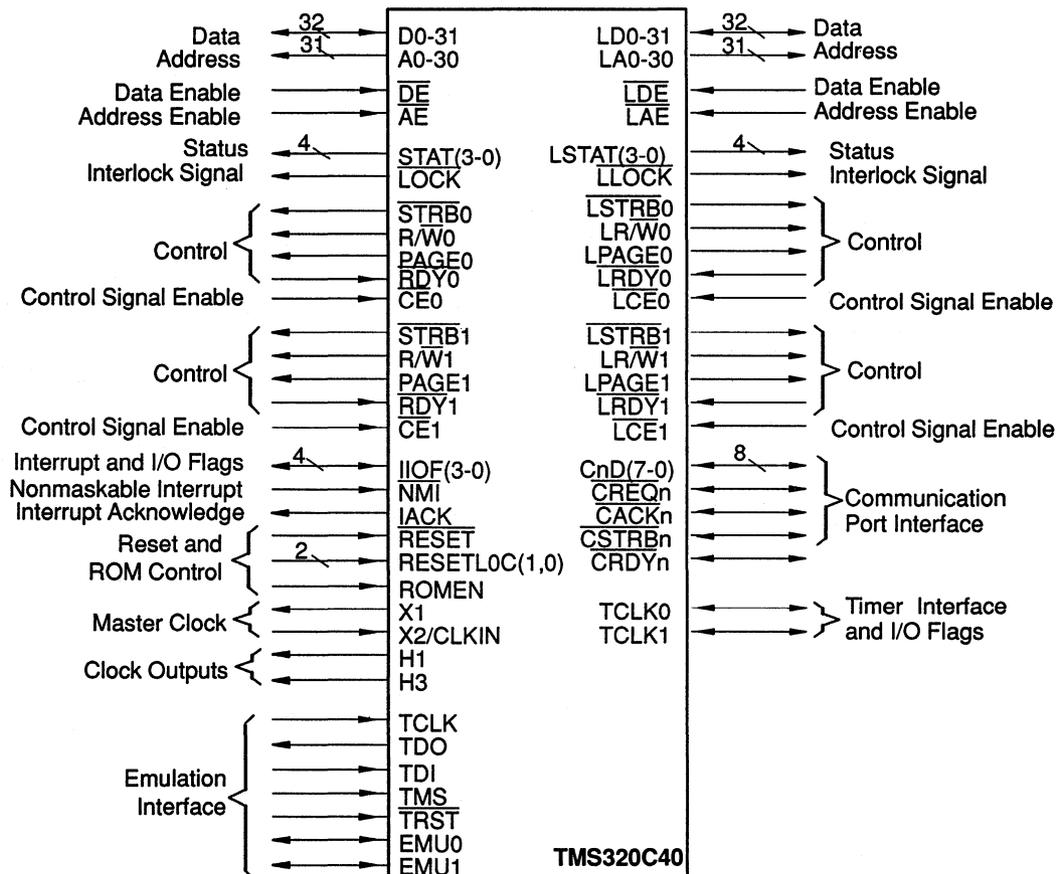
## 13.1 System Configuration Options Overview

The 'C40 interfaces connect to a wide variety of device types. Each of these interfaces is tailored to a particular family of devices.

### 13.1.1 Categories of Interfaces on the TMS320C40

The interface types on the 'C40 fall into several different categories, depending on the devices to which they are intended to be connected. Each interface comprises one or more signal lines, which transfer information and control its operation. Shown in Figure 13–1 are the signal line groupings for each of these interfaces.

Figure 13–1. External Interfaces to the TMS320C40



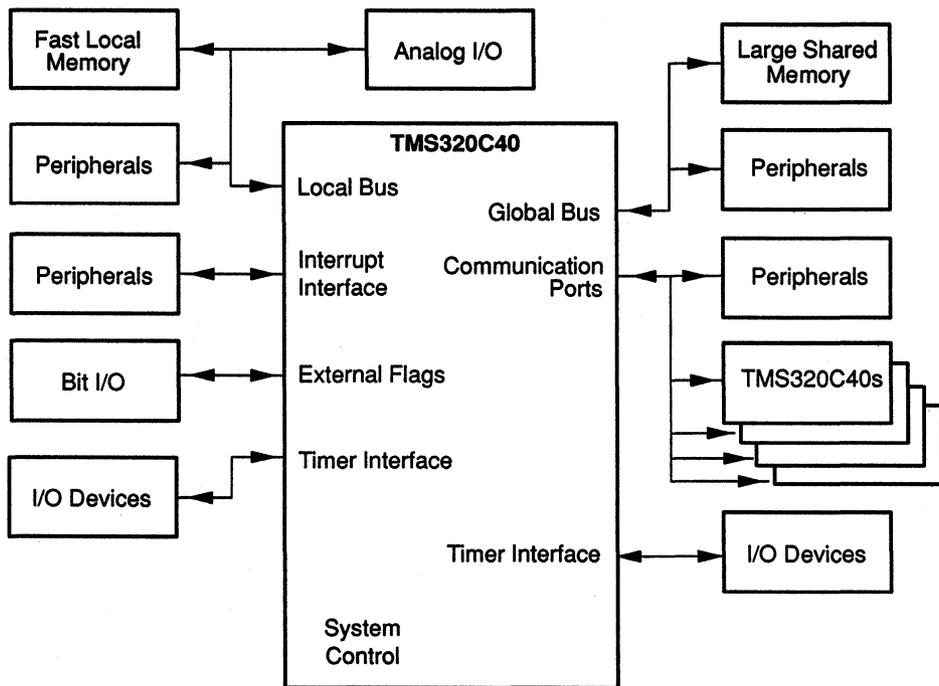
**Note:**  $n = 0$  for Communication Port 0,  $n = 1$  for Communication Port 1, etc.

Each interface is independent of the others, and different operations may be performed simultaneously on each interface. These pins are defined in more detail in Table 14–2 on page 14-5.

The global and local buses implement the primary memory-mapped interfaces to the device. These interfaces allow external devices such as DMA controllers and other microprocessors to share resources with one or more 'C40's through a common bus.

The devices that can be interfaced to the 'C40 include memory, DMA devices, and numerous parallel and serial peripherals and I/O devices. In addition, 'C40's can interface directly with each other, without external logic, through their communication ports or their external flag pins IIOF(0-3). Figure 13-2 illustrates a typical configuration of a 'C40 system with different types of external devices and the interfaces to which they are connected.

Figure 13-2. Possible System Configurations



The above block diagram in Figure 13-2 constitutes a more or less fully expanded system. In an actual design, any subset or superset of the illustrated configuration may be used.

## 13.2 Boot Loader Description and External ROM Interfacing

### 13.2.1 TMS320C40 Boot Loader Description/Operation

The boot loader provided in the on-chip ROM of the 'C40 can load and execute source programs that are received from a host processor, inexpensive ROM, or other standard memory devices. The 'C40 boot loader functions primarily as either a *memory boot loader* or a *communication port boot loader*.

- ❑ The **memory boot loader** supports user-definable byte, half-word, and full-word data formats, which allow the flexibility to load a source program from memories having widths of a byte, 16 bits, and 32 bits. The source programs to be loaded reside in one of six predefined memory locations: 0x0030 0000, 0x4000 0000, 0x6000 0000, 0x8000 0000, 0xA000 0000, and 0xC000 0000 as listed in Table 13–1.
- ❑ The **communication port boot loader** waits for the first data input from one of the six communication port channels and uses that channel to perform the boot load. Format of the incoming data stream is similar to that for a memory data stream except that the source memory width is excluded (format is described in Table 13–2, page 13-7).

Table 13–1 lists the pin values on IIOF(3–1), that select from which location the source program will be loaded.

Table 13–1. Boot Loader Mode Selection Using Pins IIOF(3–1)

External Pin			Source Program Location
IIOF3	IIOF2	IIOF1	
1	1	0	Load source program from address 0030 0000h
1	0	1	Load source program from address 4000 0000h
1	0	0	Load source program from address 6000 0000h
0	1	1	Load source program from address 8000 0000h
0	1	0	Load source program from address A000 0000h
0	0	1	Load source program from address C000 0000h
0	0	0	Reserved
1	1	1	Load source program from communication port

### 13.2.2 Boot Load Sequence

A general sequence of events in boot loading a source program is as follows:

- 1) Select the boot loader mode by resetting the processor while driving the on-chip ROM enable pin (ROMEN) high. The status of external pins IIOF(3–1) indicates where to find the source program to be loaded (memory or communication port). These options are listed in Table 13–1. (Pins IIOF(3–1) are read as the IIOF flags in the CPU IIF register (described in Table 3–6 on page 3-13).)
- 2) The boot loader takes the following steps to determine the source program's location:
  - a) If an IIF(3–1) value of  $110_2$  to  $001_2$  (6 to 1) is found, the source program is loaded from the corresponding memory address shown in the top six lines of Table 13–1.
  - b) If an IIF(3–1) value of  $000_2$  (0) is found, the boot program is exited.
  - c) If none of the combinations  $000_2$  –  $110_2$  are found, the boot loader program assumes loading will be via a communication port, and it starts checking communication port input channels (in the order port 0 through port 5). If no input is found from a communication port, the program returns to checking the status of the IIOF(3–1) pins again.
- 3) When the source program's data stream is found, the program is loaded at the address found in the fifth word of the data stream (format shown in Table 13–2) using the bus width specified in the first word (8, 16, or 32 bits wide). The first five words of the source program specify its loading and execution criteria. Remaining words are the source program(s) and vector table pointers as shown in Table 13–2:
- 4) An IACK instruction is executed. The IACK indicates the completion of the boot load sequence.
- 5) The source program is then executed (entry point is the first word of the *first* loaded program).

The data stream with its source program(s) should be in the format shown in Table 13–2. The contents of words 4 through  $n$  vary for the different source programs loaded throughout the entire data stream. The first three words and the last three words are nonvariables that affect each of the source-program blocks. The eight least significant bits of the first word specify

Table 13–2. Structure of Source Program Data Stream

Word	Contents
1	Memory width where source program resides (8, 16, or 32 bits wide)
2	Value to set in the global memory interface control register (shown in Figure 7–2, page 7-7, and Table 7–3).
3	Value to set in the local memory interface control register (shown in Figure 7–2, page 7-7, and Table 7–3).
4	Block size in words of the first program to be loaded (after this number of words are loaded, the next word should be all zeroes; if not, another block is assumed to follow).
5	Address to where the source program is loaded.
6	First word of source program.
$n$	Last word of source program (the program organized as words 4 through $n$ —these shaded words).
$n+1$	Word of all zeroes. (Note that if several source-program blocks were sent, word $n$ above would be the last word of the <i>last</i> source-program block. Each source-program block would have the format shown in words 4 through $n$ (shaded above). Then this word of all zeroes follows the <i>last</i> source program block).
$n+2$	IVTP value (interrupt vector table pointer, see Section 3.2 on page 3-15).
$n+3$	TVTP value (trap vector table pointer, see Section 3.2 on page 3-15).
$n+4$	Memory location for IACK instruction (see IACK instruction in Chapter 11).

the memory width. If byte or half-word wide is selected, the loading sequence is from LSBs to MSBs.

Each source program in a multiple block program transfer can be loaded to different specified destinations. Each program block specifies its own block size and destination address at the beginning of the block. End the entire block program loader function by appending an all-zero word (0x0000 0000h) to the *last* block (only).

The second and third last words of the source memory define the interrupt vector table pointer (IVTP) and the trap vector table pointer (TVTP). The last word of the source memory defines the memory location for the IACK instruction. Since the IACK instruction brings down the  $\overline{\text{IACK}}$  signal as data is read, the memory location specified in the IACK instruction has to be in external memory that is available in the system in order to bring the  $\overline{\text{IACK}}$  signal low. Then the processor begins execution of the first code block.

It is assumed that at least one block of source code will be loaded when the loader is invoked. Initial loader invocation with a block size of 0x00000000 produces unpredictable results.

**CAUTION**

### 13.2.3 Examples of External Memory Loads

Example 13–1, Example 13–2, and Example 13–3 respectively show memory images for memory configured as byte wide, 16-bit wide, and 32-bit wide. These examples assume that:

- ❑ The status of the IIOF(3–1) pins is  $110_2$  after reset is deasserted (memory load from 0x030 0000h — see Table 13–1 on page 13-5).
- ❑ The source program resides at memory location 0x030 0000h and defines the following:
  - Memory width for boot loader: 8, 16, or 32 bits
  - Global bus memory that requires one software wait state, external RDY (SWW = 11), page size = 64K for both  $\overline{\text{STRB0}}$  and  $\overline{\text{STRB1}}$ , and active address range = 1G for both  $\overline{\text{STRB0}}$  and  $\overline{\text{STRB1}}$ .
  - Local memory bus that requires two software wait states (SWW = 01), page size = 32K, and active address range = 1G for both  $\overline{\text{STRB0}}$  and  $\overline{\text{STRB1}}$ .
  - First block program with 294 words in length and destination address at 0x002F F840h.
  - Second block program with 64 words in length and destination address at 0x002F F800h.
  - IVTP and TVTP, which are overlapped and point to the beginning of the on-chip RAM.
  - Memory location of 0x30 0000h for IACK instruction.

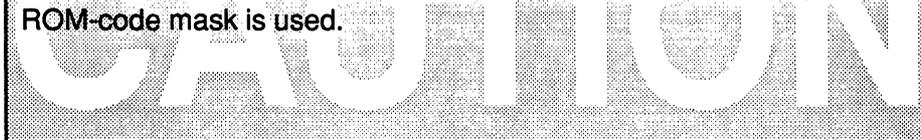
### 13.2.4 Communication Port Loading

A value of all ones on IIOF(3–1) signals that the source program is being transmitted via a communication port. Bringing all three of the IIOF(3–1) pins high also allows the pins to be used as interrupt lines without any external decode logic. With pins IIOF(3–1) all high at reset, the 'C40 determines which channel contains the program by polling the input level of each port. The input data sequence of the communication boot loader is the same as that of the memory boot loader except for the source memory width definition (because the memory width is fixed on the communication port boot loader).

### 13.2.5 External ROM Interfacing to the TMS320C40

When the 'C40's ROMEN input pin is high and RESETLOC(1,0)=00<sub>2</sub> during reset, the memory boot loader can load programs stored in off-chip ROM to any valid external or internal memory in the 'C40's memory map.

Because address zero (0) is reserved for the boot load, address zero should not be used for the reset vector when a user-defined, internal ROM-code mask is used.



Regardless of what width ROM is used (byte-wide, 16-, or 32-bit wide), the 8 LSBs of the first word read of the data stream specify the memory width. As shown in the three data stream examples starting with Example 13-1 on page 13-10, the first byte for each memory width is:

- ❑ 8-bit memories:           08h
- ❑ 16-bit memories:        0010h
- ❑ 32-bit memories:        00000020h

If 8- or 16-bit ROMs are used, the loading sequence is from LSBs to MSBs. The boot loader reads the contents of 16-bit wide memories (least significant half word first) and packs each pair of 16-bit half words to make a 32-bit word before loading each word to memory. Accordingly, the boot loader reads the contents of byte-wide memories (least significant byte first) and packs each group of four bytes into a 32-bit word before loading each word to memory. Since the boot loader does byte packing before loading, no external hardware is needed to pack the loaded bytes into a 32-bit word. For 32-bit wide ROMs, no byte packing is necessary, because the ROM data width matches that of the 'C40.

For 16-bit ROMs, the data read is expected to be in bit positions zero through fifteen. Thus, the half-word ROM's data lines should be interfaced to 'C40 data lines (L)D15-0. For byte-wide ROMs, the data read is expected to be in bit positions zero through seven. Hence, the byte-wide ROM's data lines should be interfaced to 'C40 data lines (L)D7-0. Even though the 'C40 does not require that unused data lines be pulled up to V<sub>CC</sub>, it is recommended that each unused data line be pulled up through separate 22-kilohm resistors to 5 volts for minimum power dissipation.

Example 13-1. Byte-Wide Configured Memory

Word	Address	Value	Comments
1	0300000h	08h	Memory width = 8 bits
	0300001h	00h	
	0300002h	00h	
	0300003h	00h	
2	0300004h	F0h	Global memory bus control word = 1D7BC9F0h (Described in Figure 7-2 on page 7-7.)
	0300005h	C9h	
	0300006h	7Bh	
	0300007h	1Dh	
3	0300008h	50h	Local memory bus control word = 1D739250h (Described in Figure 7-2 on page 7-7.)
	0300009h	92h	
	030000Ah	73h	
	030000Bh	1Dh	
4	030000Ch	26h	1st source program block size = 126h
	030000Dh	01h	
	030000Eh	00h	
	030000Fh	00h	
5	0300010h	40h	1st source program block starting addr = 2FF840h
	0300011h	F8h	
	0300012h	2Fh	
	0300013h	00h	
6 through 299	0300014h		1st source program block starts here (first word)
	•		•
	•		•
	•		•
	03004ABh		1st source program block ends here (last word)

Note: Shaded area identifies source program block.

Example concluded on next page

Example 13–1. Byte-Wide Configured Memory (Concluded)

Word	Address	Value	Comments
300	03004ACh	40h	2nd source program block size = 40
	03004ADh	00h	
	03004AEh	00h	
	03004AFh	00h	
301	03004B0h	00h	2nd source program block starting addr = 2FF800h
	03004B1h	F8h	
	03004B2h	2Fh	
	03004B3h	00h	
302 through 365	03004B4h		2nd source program block starts here (first word)
	•		•
	•		•
	•		•
	03005B3h		2nd source program block ends here (last word)
366	03005B4h	00h	Value 0 to terminate the program block load
	03005B5h	00h	
	03005B6h	00h	
	03005B7h	00h	
367	03005B8h	00h	IVTP = 002FF800h
	03005B9h	F8h	
	03005BAh	2Fh	
	03005BBh	00h	
368	03005BCh	00h	TVTP = 002FF800h
	03005BDh	F8h	
	03005BEh	2Fh	
	03005BFh	00h	
369	03005C0h	00h	Memory location for IACK instruction = 30 0000h  (This is the final word in the data stream.)
	03005C1h	00h	
	03005C2h	30h	
	03005C3h	00h	

Note: Shaded area identifies source program block.

Example 13-2. 16-Bits-Wide Configured Memory

Word	Address	Value	Comments
1	0300000h	0010h	Memory width = 16 bits
	0300001h	0000h	
2	0300002h	C9F0h	Global memory bus control word = 1D7BC9F0h
	0300003h	1D7Bh	
3	0300004h	9250h	Local memory bus control word = 1D739250h
	0300005h	1D73h	
4	0300006h	0126h	1st program block size = 126h
	0300007h	0000h	
5	0300008h	F840h	1st program block starting addr = 2FF840h
	0300009h	002Fh	
6 through 299	030000Ah		1st program block starts here (first word)
	•		•
	•		•
	•		•
	0300255h		1st program block ends here (last word)
300	0300256h	0040h	2nd program block size = 40h
	0300257h	0000h	
301	0300258h	F800h	2nd program block starting addr = 2FF800h
	0300259h	002Fh	
302 through 365	030025Ah		2nd program block starts here (first word)
	•		•
	•		•
	•		•
	03002D9h		2nd program block ends here (last word)
366	03002DAh	0000h	Value 0 to terminate the program block read
	03002DBh	0000h	
367	03002DCh	F800h	IVTP = 002FF800h
	03002DDh	002Fh	
368	03002DEh	F800h	TVTP = 002FF800h
	03002DFh	002Fh	
369	03002E0h	0000h	Memory location for IACK instruction = 30 0000h (This is the final word in the data stream.)
	03002E1h	0030h	

Note: Shaded areas identify source program blocks.

Example 13-3. 32-Bits-Wide Configured Memory

Word	Address	Value	Comments
1	0300000h	0000020h	Memory width = 32 bits
2	0300001h	1D7BC9F0h	Global memory bus control word = 01D7BC9F0h
3	0300002h	1D739250h	Local memory bus control word = 01D739250h
4	0300003h	00000126h	1st program block size = 126h
5	0300004h	002FF840h	1st program block starting addr = 2FF840h
6 through 299	0300005h		1st program block starts here (first word)
	•		•
	•		•
	•		•
	030012Ah		1st program block ends here (last word)
300	030012Bh	00000040h	2nd program block size = 40h
301	030012Ch	002FF800h	2nd program block starting addr = 2FF800h
302 through 365	030012Dh		2nd program block starts here (first word)
	•		•
	•		•
	•		•
	030016Ch		2nd program block ends here (last word)
366	030016Dh	00000000h	Value 0 to terminate the program block load
367	030016Eh	002FF800h	IVTP = 002FF800h
368	030016Fh	002FF800h	TVTP = 002FF800h
369	0300170h	00300000h	Address location for IACK instruction = 00300000h

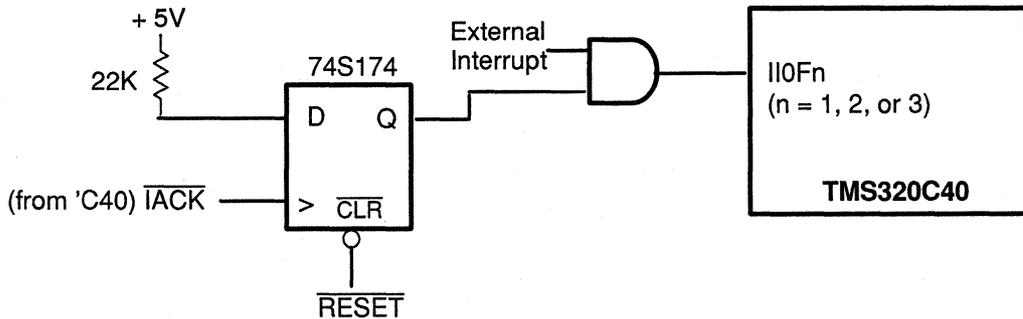
Note: Shaded areas identify source program blocks.

### 13.2.6 IIOF(3-1) Pin Loading

The load options are based upon the status of IIOF(3-1) as general-purpose input pins. Therefore, in order to select the correct boot loader mode, pins IIOF(3-1) must be kept at a constant valid status value for a certain time period (values listed in Table 13-1 on page 13-5). See the 'C40 boot loader program for detailed information — Figure 13-4 starting on page 13-14.

After the boot load is complete, the  $\overline{\text{IACK}}$  signal is brought down for one cycle. Figure 13-3 shows an example circuit that generates the IIOF(3-1) signals for boot load selection and also allows incoming external interrupts during normal mode of operation. In this example, after reset, the IIOF pins stay low until the  $\overline{\text{IACK}}$  signal is received.

Figure 13-3. Circuit for Generation of a Low IIOF signal for Boot Loader Selection



### 13.2.7 TMS320C40 Boot Loader Source Program

Figure 13-4. Boot Loader Source Program

```
*****
*
*   C40BOOT - TMS320C40 BOOT LOADER PROGRAM
*           (C) COPYRIGHT TEXAS INSTRUMENTS INC., 1990
*
*NOTE: 1.  AFTER DEVICE RESET, THE PROGRAM IS CHECKING
*           THE INPUT STATUS OF IIOF1-3 PINS AND COMMUNI-
*           CATION PORT INPUT FLAGS TO CONFIGURE ITSELF
*           WHEN ON CHIP ROM IS ENABLED (ROMEN=1). THE IIOF0
*           PIN IS ASSUMED TO BE PULLED HIGH.
*
*           2.  THE FUNCTION SELECTION OF IIOF1-3 IS LISTED AS:
*
*           IIOF3 IIOF2 IIOF1           FUNCTION
*           ---
*           1     1     0     Memory boot loader from 00300000H
*           1     0     1     Memory boot loader from 40000000H
*           1     0     0     Memory boot loader from 60000000H
```

Figure 13-4. Boot Loader Source Program (Continued)

*	0	1	1	Memory boot loader from 80000000H
*	0	1	0	Memory boot loader from A0000000H
*	0	0	1	Memory boot loader from C0000000H
*	0	0	0	Reserved
*	1	1	1	Communication Port boot loader

\* THE PROGRAM ASSUMES THE COMMUNICATION PORT BOOT  
 \* LOADER IS THE DEFAULT FUNCTION. IF NO OTHER  
 \* FUNCTION IS SELECTED, THE PROGRAM STARTS CHECKING  
 \* THE COMMUNICATION PORT INPUT CHANNELS. IF THERE IS  
 \* NO INPUT FROM A COMMUNICATION PORT, THE PROGRAM  
 \* RECHECKS THE IIOF(3-1) STATUS AGAIN.  
 \*

\* 3. MEMORY BOOT LOADER LOADS WORD, HALF-WORD, OR BYTE  
 \* WIDE PROGRAM TO DIFFERENT SPECIFIED LOCATIONS. THE 8  
 \* LSBs OF THE FIRST MEMORY SPECIFIES THE MEMORY WIDTH.  
 \* IF THE HALF-WORD OR BYTE WIDE PROGRAM IS SELECTED,  
 \* THE LSBs ARE LOADED FIRST AND THEN THE MSBs. THE NEXT  
 \* 2 WORDS CONTAIN THE CONTROL WORD FOR THE GLOBAL AND  
 \* LOCAL MEMORY INTERFACE CONTROL REGISTERS. NEXT COME  
 \* THE PROGRAM BLOCKS. THE FIRST TWO WORDS OF EACH  
 \* PROGRAM BLOCK CONTAIN THE BLOCK SIZE AND DESTINATION  
 \* ADDRESS WHERE THE PROGRAM IS TO BE LOADED. WHEN THE  
 \* ZERO BLOCK SIZE IS READ, THE PROGRAM BLOCK LOADING  
 \* IS TERMINATED. THE NEXT TWO WORDS ARE THE  
 \* INITIAL VALUES FOR THE IVTP AND TVTP REGISTERS.  
 \* AFTER THE BOOT LOADING IS COMPLETED, THE IACK SIG-  
 \* NAL WILL BE SENT OUT ACCORDING TO THE LAST WORD OF THE  
 \* SOURCE MEMORY, AND THE PROGRAM COUNTER WILL  
 \* BRANCH TO THE STARTING ADDRESS OF THE FIRST  
 \* PROGRAM BLOCK.  
 \*

\* 4. IF THE IIOF(3-1) ARE SETUP FOR COMMUNICATION PORT  
 \* BOOTLOADER, THE PROCESSOR WILL WAIT FOR THE FIRST  
 \* INPUT FROM AN INPUT COMMUNICATION CHANNEL AND USE  
 \* THAT CHANNEL TO PERFORM THE DOWNLOAD. THE BEGIN-  
 \* NING TWO WORDS SHOULD CONTAIN THE GLOBAL AND LOCAL  
 \* BUS CONTROL WORDS. SIMILAR TO THE MEMORY LOADER,  
 \* PROGRAM CAN BE LOADED INTO DIFFERENT MEMORY  
 \* BLOCKS. FIRST TWO WORD OF EACH PROGRAM BLOCK CON-  
 \* TAIN BLOCK SIZE AND MEMORY ADDRESS TO BE LOADED  
 \* INTO. WHEN THE ZERO BLOCK SIZE IS READ, THE PRO-  
 \* GRAM BLOCK LOADING IS TERMINATED. IN OTHER WORDS,  
 \* IN ORDER TO TERMINATE THE PROGRAM BLOCK LOADING, A  
 \* ZERO HAS TO BE ADDED AT THE END OF PROGRAM BLOCK.  
 \* THE FOLLOWING TWO WORDS ARE THE INITIAL VALUES FOR  
 \* THE IVTP AND TVTP REGISTERS. AFTER THE BOOT LOAD-  
 \* ING IS COMPLETED, THE IACK SIGNAL WILL BE SENT OUT  
 \* ACCORDING TO THE LAST WORD OF THE SOURCE MEMORY,  
 \* AND THE PROGRAM COUNTER WILL BRANCH TO THE START-  
 \* ING ADDRESS OF THE FIRST PROGRAM BLOCK.  
 \*

Figure 13-4. Boot Loader Source Program (Continued)

```

        .page
*****
*                               RESET VECTOR                               *
*****

        .sect "vectors"
RESET  .word  START           ; On hardware RESET go to START

*****
*                               TMS320C40 PROCESSOR BOOT LOADER           *
*****

        .text
START:  CMPI   04440H,IIF      ; Test IIOF0 pin condition
        BEQ   LIFETEST       ; If low, execute life test
        LDHI  0010H,ARO      ; Load peripheral mem. map start
                               addr 100000H
        LDHI  002FH,SP       ; Initialize stack pointer SP to
        OR   0FFF0H,SP       ; internal RAM address 2FFFF0H
        LDI   0,R0           ; Set start address flag off
        LDI   COM_LOAD,R10   ; Comm. port load subroutine
                               address -> R10

*
*   CHECK THE IIOF1-3 FOR THE BOOT LOADER
*
CHECK:  LDHI  0030H,AR1       ; Load memory address = 00300000H
        CMPI  04404H,IIF     ; Test function 110 condition
        BEQ   MEMORY        ; If true, execute memory boot
                               loader
*
        LDHI  04000H,AR1     ; Load memory address = 40000000H
        CMPI  04044H,IIF     ; Test function 101 condition
        BEQ   MEMORY        ; If true, execute memory boot
                               loader
*
        LDHI  06000H,AR1     ; Load memory address = 60000000H
        CMPI  04004H,IIF     ; Test function 100 condition
        BEQ   MEMORY        ; If true, execute memory boot
                               loader
*
        LDHI  08000H,AR1     ; Load memory address = 80000000H
        CMPI  00444H,IIF     ; Test function 011 condition
        BEQ   MEMORY        ; If true, execute memory boot
                               loader
*
        LDHI  0A000H,AR1     ; Load memory address = A0000000H
        CMPI  00404H,IIF     ; Test function 010 condition
        BEQ   MEMORY        ; If true, execute memory boot
                               loader
*
        LDHI  0C000H,AR1     ; Load memory address = C0000000H
        CMPI  00044H,IIF     ; Test function 001 condition
        BEQ   MEMORY        ; If true, execute memory boot
                               loader
*
        CMPI  00004H,IIF     ; Test function 000 condition
        BEQ   RESERVED      ; If true, branch to reserve

```

Figure 13-4. Boot Loader Source Program (Continued)

```

*-----*
*          COMMUNICATION PORT BOOT LOADER
*-----*
*
*   CHECK COMMUNICATION PORT INPUT CHANNEL
*
*       ADDI   040H,AR0,AR3 ; Point to comm. port 0
*                               control register addr
*       LDI    5,AR1        ; Set loop counter for
*                               CHECK_CH loop
CHECK_CH: LSH3   -9,*AR3,R1  ; Check comm port input
*       BNZ   LOAD1        ; If input exist, start comm
*                               port loader
*       ADDI   010H,AR3     ; Point to next comm. port
*                               channel addr
*       DBU   AR1,CHECK_CH ; Check next comm. port
*                               channel input
*       B     CHECK        ; Recheck the input flags
*-----*
*          MEMORY BOOT LOADER
*-----*
*
*   TEST MEMORY WORD WIDTH
*
MEMORY:  LDI   *AR1++(1),R1  ; Load the memory word width
*       LDI   W_WIDE,R10    ; Full-word size subroutine
*                               address -> R10
*       LSH   26,R1         ; Test bit5 of mem. width word
*       BN   LOAD0         ; If '1' start PGM loading
*                               (32 bits width)
*
*       NOP   *AR1++(1)    ; Jump last half word from
*                               mem. word
*       LDI   H_WIDE,R10   ; Half-word size subroutine
*                               address -> R10
*       LSH   1,R1         ; Test bit4 of mem. width word
*       BN   LOAD0         ; If '1' start PGM loading
*                               (16 bits width)
*
*       LDI   B_WIDE,R10   ; Byte size subroutine address
*                               -> R10
*       ADDI  2,AR1        ; Jump last 2 bytes from
*                               mem. word
*
*   START PROGRAM LOADING
*
LOAD0:  CALLU R10          ; Load new word according to
*                               mem. width
*       STI   AR2,*AR0     ; Set global bus control
*                               register
*       CALLU R10          ; Load new word according to
*                               mem. width
*       STI   AR2,*+AR0(4) ; Set local bus control
*                               register

```

Figure 13-4. Boot Loader Source Program (Continued)

```

LOAD2: CALLU R10          ; Load new word according to
*                               mem. width
        SUBI3 1,AR2,RC    ; Set block size for
*                               repeat loop
        CMPI  -1,RC       ; If 0 block size start PGM
        BEQ  IVTP_LOAD
*
        CALLU R10        ; Load new word according to
                               mem. width
        LDI  AR2,AR0      ; Set destination address
        LDI  R0,R0        ; Test start address loaded
*                               flag
        LDIZ AR2,R9       ; Load start address if flag
*                               off
        LDI  -1,R0        ; Set start & dest. address
*                               flag on
        SUBI 1,R10        ; Sub address with loop
        CALLU R10        ; Load block words according
*                               to mem. width
        LDI  1,R0         ; Set dest. address flag off
        ADDI 1,R10        ; Sub address without loop
        B    LOAD2        ; Jump to load a new block
*                               when loop completed
*
*   INITIALIZE IVTP AND TVTP REGISTERS
*
IVTP_LOAD:CALLU R10      ; Load new word according to
*                               mem. width
        LDPE AR2,IVTP     ; Load the IVTP pointer
TVTP_LOAD:CALLU R10      ; Load new word according to
*                               mem. width
        LDPE AR2,TVTP     ; Load the TVTP pointer
        CALLU R10        ; Load new word according to
*                               mem. width
        IACK *AR2        ; Send out IACK signal out
        BU    R9         ; Branch to start of program
*****
*   BYTE-WIDE MEMORY BOOT LOADER SUBROUTINE   *
*****
LOOP_B:  RPTB  LOAD_B     ; PGM load loop
B_WIDE:  LWLO  *AR1++(1),AR2 ; Load byte 0 (LSB)
        NOP
        LWL1  *AR1++(1),AR2 ; Join byte 1 with byte 0
        NOP
        LWL2  *AR1++(1),AR2 ; Join byte 2 with byte 0 & 1
        NOP
        LWL3  *AR1++(1),AR2 ; Join byte 3 with byte 0, 1,
*                               & 2
        LDI  R0,R0        ; Test load address flag
        BNN  B_END
LOAD_B:  STI  AR2,*AR0++(1) ; Store new word to dest.
*                               address
B_END:   RETSU          ; Return from subroutine

```

Figure 13-4. Boot Loader Source Program (Concluded)

```

*****
*   HALF-WORD WIDE MEMORY BOOT LOADER SUBROUTINE   *
*****
LOOP_H:   RPTB   LOAD_H       ; PGM load loop
H_WIDE:   LWLO   *AR1++(1),AR2; Load LSB half-word
          NOP
          LWL2   *AR1++(1),AR2; Join MSB half-word with
*                                     LSB half-word
          LDI    R0,R0         ; Test load address flag
          BNN    H_END

LOAD_H    STI    AR2,*AR0++(1); Store new word to dest.
*                                     address
H_END     RETSU                ; Return from subroutine

*****
*   FULL-WORD WIDE MEMORY BOOT LOADER SUBROUTINE   *
*****
LOOP_W    RPTB   LOAD_W       ; PGM load loop
W_WIDE    LDI    *AR1++(1),AR2; Read a new 32 bits word
          LDI    R0,R0         ; Test load address flag
          BNN    W_END

LOAD_W    STI    AR2,*AR0++(1); Store new word to dest.
*                                     address
W_END     RETSU                ; Return from subroutine

*****
*   COMMUNICATION PORT BOOT LOADER SUBROUTINE      *
*****
LOOP_C    RPTB   LOAD_C       ; PGM load loop
COM_LOAD  LSH3   -9,*AR3,R1    ; Check comm port input
          BZ     COM_LOAD      ; Wait for comm port input
          LDI    *+AR3(1),AR2  ; Read a new 32 bits word
          LDI    R0,R0         ; Test load address flag
          BNN    C_END

LOAD_C    STI    AR2,*AR0++(1); Store new word to dest.
*                                     address
C_END     RETSU                ; Return from subroutine

RESERVED:
          .end

```

### 13.3 Global and Local Bus Interface

The 'C40 uses the global and local buses to access the majority of its memory-mapped locations. Since these two memory interfaces are identical in every way, except for their positions in the memory map, each example in this memory interface section focuses on only one of the two interfaces. However, all of the examples are applicable to either the local or global bus. Additionally, each of the buses features two identical, mutually exclusive sets of control signals:

<u>Global Bus</u>	<u>Local Bus</u>
$\overline{\text{STRB0}}$	$\overline{\text{LSTRB0}}$
$\overline{\text{STRB1}}$	$\overline{\text{LSTRB1}}$
$\overline{\text{CE0}}$	$\overline{\text{LCE0}}$
$\overline{\text{CE1}}$	$\overline{\text{LCE1}}$
$\overline{\text{RDY0}}$	$\overline{\text{LRDY0}}$
$\overline{\text{RDY1}}$	$\overline{\text{LRDY1}}$

Also,  $\overline{\text{AE}}$  and  $\overline{\text{DE}}$  put the global bus in high impedance, and  $\overline{\text{LAE}}$  and  $\overline{\text{LDE}}$  put the local bus in high impedance.

Although both the global and the local buses can interface to a wide variety of devices, the devices most commonly interfaced are memories. Therefore, memory interface examples are used in this section.

#### 13.3.1 Zero Wait-State Interface to RAMs

For a full-speed, zero wait-state interface to any device, a 50-MHz 'C40 (40-ns instruction cycle time) requires a read access time of 21-ns from address stable to data valid. For most memories, the access time from chip enable is the same as access time from address; thus, it is possible to use 20-ns memories at full speed with a 50-MHz 'C40. However, to properly use 20-ns memories, there can be no long delays between the processor and the memories. Avoiding these delays is not always possible in practice, because of interconnection delays and the fact that gating is sometimes required for chip enable generation. In addition, if a memory device with an output enable is chosen, output enable must become active soon enough to ensure that the memory can meet the data valid timing requirements of the 'C40. For memories with 20-ns access times, the output enable active to data valid timing parameter is typically less than 10 ns.

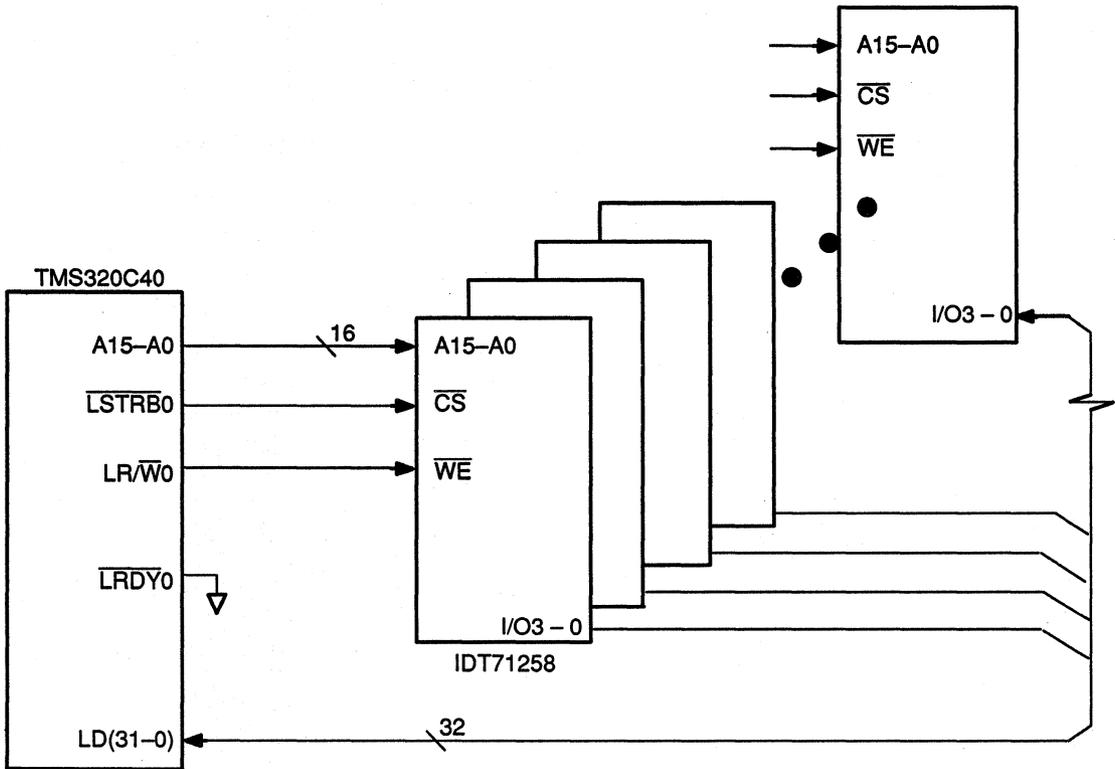
Currently available RAMs without output enable (OE) control lines include the 1-bit wide organized RAMs and most of the 4-bit wide RAMs. Those with OE controls include the byte-wide and a few of the 4-bit wide RAMs. Many of the fastest RAMs do not provide OE control; they use chip-enable (CE)

controlled write cycles to ensure that data outputs do not turn on for write operations. In CE-controlled write cycles, the write control line ( $\overline{WE}$ ) goes low before CE goes low, and internal logic holds the outputs disabled until the cycle is completed. Using CE-controlled write cycles is an efficient way to interface fast RAMs without OE controls to the 'C40 at full speed.

### 13.3.1.1 RAM Interface - Using One Local Strobe

Figure 13–5 shows the 'C40's local bus interfaced to the Integrated Device Technology™ IDT71258 20-ns 64K x 4-bit CMOS static RAMs with zero wait states using chip enable-controlled write cycles. These RAMs are arranged to implement 64K, 32-bit words located at addresses 00000h thru 0FFFFh (internal ROM is assumed to be disabled), which are the first 64K words in external memory. If these 64K words of SRAM are the only memory controlled by  $\overline{LSTRB0}$ , the LSTRB ACTIVE field of the local memory interface control register (LMICR) should be set to its minimum value 01111<sub>2</sub>, allowing  $\overline{LSTRB0}$  to be active only for the first 64K words of the 'C40's memory space. (The memory interface control register and its various fields are shown in Figure 7–2 on page 7-7). In addition, because this memory is the only memory interfaced to  $\overline{LSTRB0}$ ,  $\overline{LSTRB0}$  requires only one page. The PAGESIZE field of the LMICR should be set to 01111<sub>2</sub>. Also note that in Figure 13–5, the  $\overline{LRDY0}$  input is tied low, selecting zero wait states for all  $\overline{LSTRB0}$  accesses on the local bus. With all of the zero-wait-state memory controlled by  $\overline{LSTRB0}$ ,  $\overline{LSTRB1}$  can be used to control accesses to slower read-only memory devices or other types of memory.

Figure 13-5. TMS320C40 Interface to Zero-Wait-State SRAM



In this circuit implementation, no external logic is necessary to interface the 'C40 to the memory device. This glueless interface is possible because changes in  $\overline{LR/\overline{W}}$  are always framed by  $\overline{LSTRB}$ . For typical memory devices, it is necessary to hold the device inactive ( $\overline{CS}$  inactive) during changes in  $\overline{WE}$ ; this avoids undesired memory accesses while the address changes. The 'C40 ensures this by having  $\overline{LSTRB}$  always frame changes in  $\overline{LR/\overline{W}}$ . (See Section 7.5 on page 7-17 for more information.)

### 13.3.1.2 Consecutive Reads Followed by a Write Interface Timing

Figure 13–6 shows the timing of consecutive reads followed by a write. For consecutive reads,  $\overline{LSTRB0}$  stays active (low), and  $\overline{LR/\overline{W}}$  stays high as long as read cycles continue. The critical timing that must be met for back-to-back reads is the address-valid to data-valid time. The 'C40 requires zero-wait-state memories to have an address-valid to data-valid time of less than 21-ns. This can be explained in more detail as:

$$\text{one H1 cycle time} - [(\text{H1 low to address-valid time}) + (\text{data setup time before H1 low})]$$

For most memory devices, this time is the same as the memory access time, which is  $t_1 = 20$  ns. Thus, memories with access times of 25 ns or more cannot meet this timing.

Memory device timing is not as critical for zero-wait-state as for nonzero-wait-state write cycles, because of the two H1 cycle writes of the 'C40. The extra cycle gives  $\overline{LSTRB0}$  enough time to frame  $\overline{LR/\overline{W}}$ , preventing memories that go into high impedance slowly at the end of a read cycle from driving the bus during the subsequent write cycle. For the memory device used in this design (Figure 13–6), the data lines are guaranteed to be three stated ( $t_2 = 10$  ns) after  $\overline{CS}$  goes inactive, which gives more than 23 ns of margin before the 'C40 starts driving the bus with write data. Also, the extra cycle with  $\overline{LSTRB0}$  inactive prevents writes to random locations in memory while the address is changing between consecutive writes.

For the write cycles shown in Figure 13–6 and Figure 13–7, the RAM requires 15 ns of write data setup before  $\overline{CS}$  goes high, and this design provides at least 24 ns ( $t_3$ ). A data hold time of 0 ns ( $t_4$ ) is required by the RAM, and this design provides greater than 13 ns. Finally, the RAM's setup and hold times for address (with respect to  $\overline{CS}$  high) of 20 and 0 ns, respectively, are also met with a clear margin.

Figure 13-6. Consecutive Reads Followed by a Write

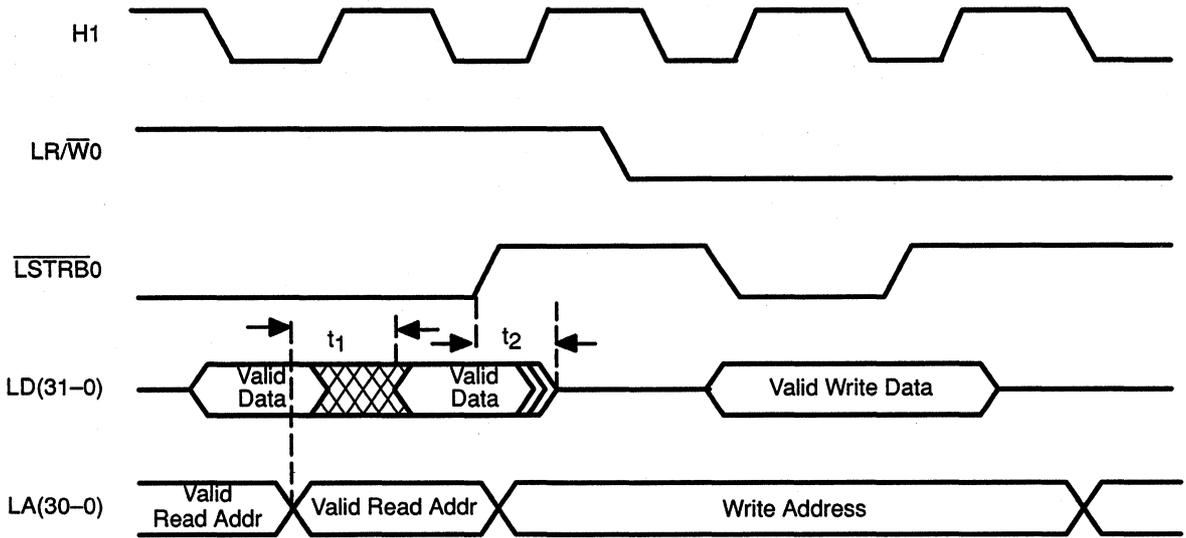
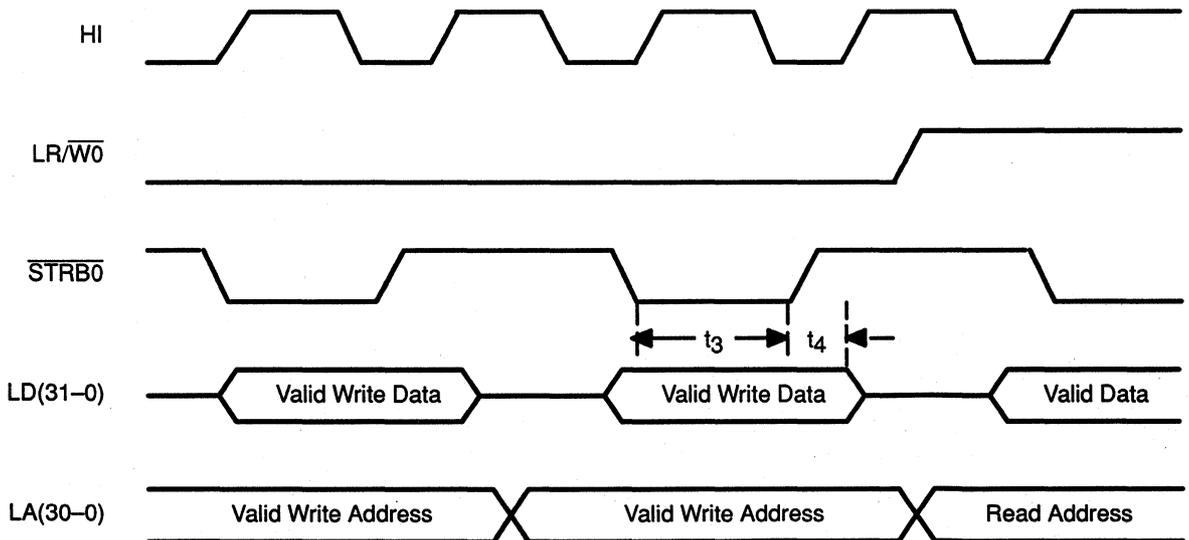


Figure 13-7. Consecutive Writes Followed by a Read



### 13.3.1.3 Consecutive Writes Followed by a Read Interface Timing

Figure 13–7 shows the timing of consecutive writes followed by a read. Notice that between consecutive writes,  $\overline{\text{LR}}/\overline{\text{W}}$  stays low, but  $\overline{\text{STRB0}}$  goes inactive to frame the write cycles. Although 'C40 zero-wait-state writes take two H1 cycles, internally (from the perspective of the CPU and DMA) writes appear to take one cycle if no accesses to that interface is already in progress.

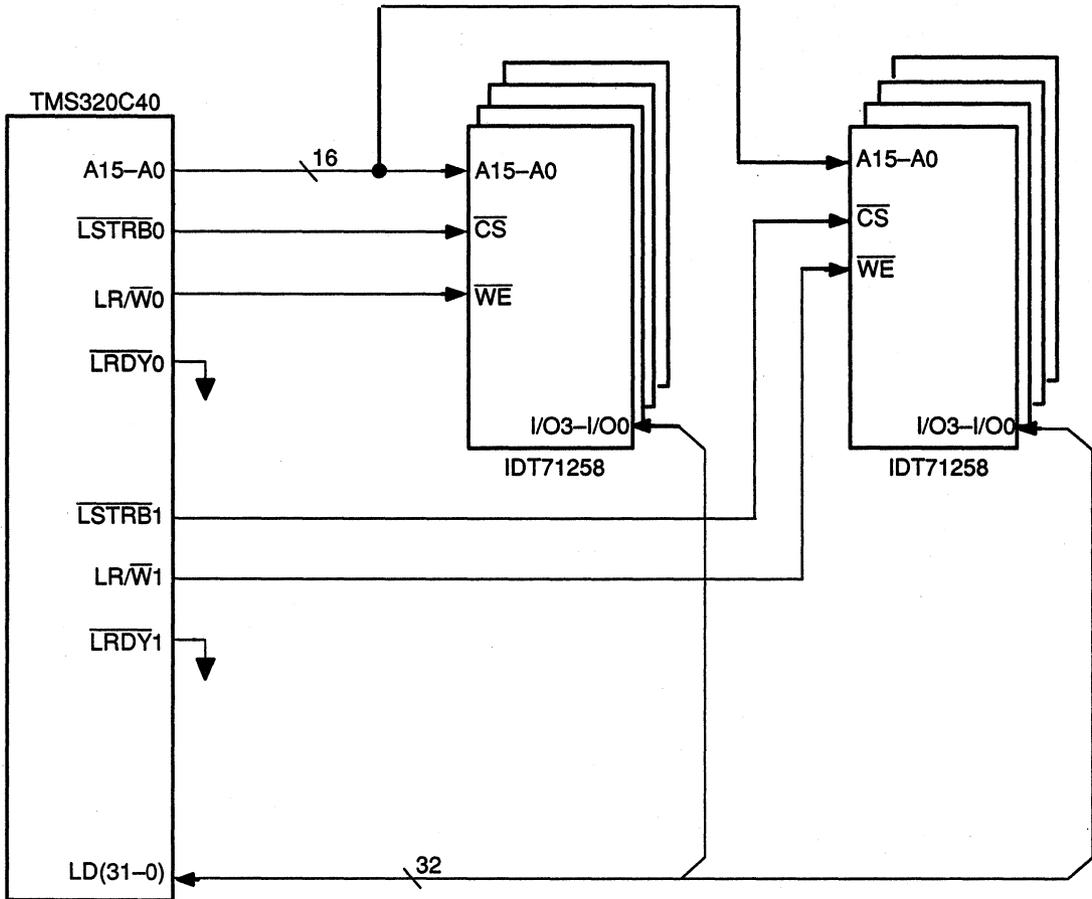
In the read cycle following the writes in Figure 13–7, the 'C40 requires zero-wait-state memories to have a  $\overline{\text{LSTRB}}$  active to data-valid time of less than 21 ns (one H1 cycle time minus (H1 low to  $\overline{\text{LSTRB}}$  active time plus data setup time before H1 low)). For most memory devices, this time is the same as the memory access time, which is  $t_1 = 20$  ns in this design. Thus, a margin of only 1 ns exists, leaving little time for  $\overline{\text{STRB}}$  gating if desired.

### 13.3.1.4 RAM Interface Using Both Local Strobes

Figure 13–8 shows the 'C40's local bus interfaced to IDT71258 RAMS — 20-ns 64K x 4-bit CMOS static RAMs with zero wait states using  $\overline{\text{CS}}$  controlled write cycles. These RAMs are arranged to allow 128K 32-bit words of local memory, which is implemented as two 64K x 32-bit banks. One bank is controlled by each of the two sets of control signals on the local bus. To map these memory devices properly in the 'C40's memory space, you must use the local memory interface control register (LMICR) to define which part of the local bus's memory space is mapped to each of the two strobes. In this implementation with internal ROM disabled,  $\overline{\text{LSTRB0}}$  is mapped to the first 64K words of the local space — addresses 0h through 0FFFFh, and  $\overline{\text{LSTRB1}}$  is mapped to the rest of the local space — addresses 10000h through 7FFF FFFFh. For this memory configuration, the LSTRB ACTIVE field of the local memory interface control register (LMICR) should be set to 01111<sub>2</sub>. Also, each  $\overline{\text{LSTRB}}$  requires only one page. The PAGESIZE field of the LMICR should be set to 01111<sub>2</sub>. Also, note that in Figure 13–8, the  $\overline{\text{LRDY}}$  inputs are tied low, selecting zero wait states for all accesses on the local bus.

Hence, through the use of the 'C40's four strobes (two each on the local and global buses), four different banks of memory can be decoded. In addition, the address decoding can be changed under program control by changing the LSTRB active field (bits 24–28) of the LMICR or the global memory interface control register (GMICR). If more than four banks of memory must be decoded or if the chosen memory device cannot meet the read cycle timing requirements for the 'C40 at zero wait states, page switching (discussed in subsection 13.4.6 on page 13-32) should be used to add an extra cycle to read accesses outside the current bank boundary.

Figure 13-8. TMS320C40 Interface to Zero-Wait-State SRAMs, Two Strobes



## 13.4 Wait States and Ready Generation

The use of wait states can greatly increase system flexibility and reduce hardware requirements over systems without wait-state capability. The 'C40 has the capability of generating wait states on either the global bus or the local bus, and both buses have independent sets of ready control logic. The buses' wait-state configuration is determined by the SWW and WTCNT fields of the local and global bus interface control registers (see Section 7.4, page 7-15, for a detailed description of the wait-state options).

This section discusses ready generation from the perspective of the *global bus* interface; however, wait-state operation on the *local bus* is the same as on the global bus, so this discussion pertains equally well to both (local and global). Also, the local and global buses each have two sets of control signals — R/W0, STRB0, RDY0, and R/W1, STRB1, RDY1 — with each set of control signals having its own ready signal, providing for more flexibility in support of external devices with different speeds. Since both strobes' ready signals share the same electrical characteristics, the following discussion focuses on one of the global bus's set of control signals.

Wait states are generated on the basis of:

- the internal wait-state generator,
- the external ready inputs ( $\overline{\text{RDY}}0$  or  $\overline{\text{RDY}}1$ ), or
- the logical AND or OR of the two (discussed in Section 7.4, page 7-15).

When enabled, internally generated wait states affect all external cycles, regardless of the address accessed. If different numbers of wait states are required for various external devices, the external  $\overline{\text{RDY}}$  input may be used to customize wait-state generation to specific system requirements.

If either the logical OR or electrical AND (since the signals are true low) of the external and wait-count ready signals is selected, the earlier of the two signals will generate a ready condition and allow the cycle to be completed. It is not required that both signals be present.

### Note: STRBx SWW Field Values

The STRBx SWW fields of the memory-interface control register are shown in Figure 7-2 (page 7-7) and explained in Table 7-7 (page 7-16).

### 13.4.1 ORing of the Ready Signals (STRBx SWW = 10)

The OR of the two ready signals can be used to implement wait states for devices that require a greater number of wait states than are implemented with internal logic (up to seven). This feature is useful, for example, if a system contains some fast and some slow devices. In this case:

- ❑ **Fast devices** can generate ready externally with a minimum of logic. When fast devices are accessed, the external hardware responds promptly with ready, which terminates the cycle.
- ❑ **Slow devices** can use the internal wait counter for larger numbers of wait states. When slow devices are accessed, the external hardware does not respond, and the cycle is appropriately terminated after the internal wait count.

The OR of the two ready signals may also be used if conditions occur that require termination of bus cycles before the number of wait states implemented with external logic. In this case, a shorter wait count is specified internally than the number of wait states implemented with the external ready logic, and the bus cycle is terminated after the wait count. This feature may also be used as a safeguard against inadvertent accesses to nonexistent memory that would never respond with ready and would therefore lock up the 'C40.

If the OR of the two ready signals is used, however, and the internal wait-state count is less than the number of wait states implemented externally, the external ready generation logic must have the ability to reset its sequencing to allow a new cycle to begin immediately following the end of the internal wait count. This requires that, under these conditions:

- ❑ consecutive cycles must be from independently decoded areas of memory (or from different pages in memory), and
- ❑ the external ready generation logic must be capable of restarting its sequence as soon as a new cycle begins.

Otherwise, the external ready generation logic may lose synchronization with bus cycles and therefore generate improperly timed wait states.

### 13.4.2 ANDing of the Ready Signals (STRBx SWW = 11)

If the logical AND (electrical OR) of the wait count and external ready signals is selected, the later of the two signals will control the internal ready signal, but both signals must occur. Accordingly, external ready control must be implemented for each wait-state device, and the wait count ready signal must be enabled.

This feature is useful if there are devices in a system that are equipped to provide a ready signal but cannot respond quickly enough to meet the

'C40's timing requirements. In particular, if these devices normally indicate a ready condition and, when accessed, respond with a wait until they become ready, the logical AND of the two ready signals can be used to save hardware in the system. In this case, the internal wait counter can provide wait states initially, and then the external ready can provide wait states after the external device has had time to send a not-ready indication. The internal wait counter then remains ready until the external device also becomes ready, which terminates the cycle.

Additionally, the AND of the two ready signals may be used for extending the number of wait states for devices that already have external ready logic implemented but require additional wait states under certain unique circumstances.

### 13.4.3 External Ready Generation

In the implementation of external ready generation hardware, the particular technique employed depends heavily on the specific characteristics of the system. The optimum approach to ready generation varies, depending on the relative number of wait-state and nonwait-state devices in the system and on the maximum number of wait states required for any one device. The approaches discussed here are intended to be general enough for most applications and are easily modifiable to comprehend many different system configurations.

In general, ready generation involves the following three functions:

- 1) Segmentation of the address space in some fashion to distinguish fast and slow devices.
- 2) Generation of properly timed ready indications.
- 3) Logical ORing of all the separate ready timing signals together to connect to the physical ready input.

Segmentation of the address space is required to obtain a unique indication of each particular area within the address space that requires wait states. This segmentation is commonly implemented in a system in the form of chip-select generation. Chip-select signals may be used to initiate wait states in many cases; however, occasionally, chip-select decoding considerations may provide signals that will not allow ready input timing requirements to be met. In this case, *coarse* address space segmentation may be made on the basis of a small number of address lines, where simpler gating allows signals to be generated more quickly. In either case, the signal indicating that a particular area of memory is being addressed is normally used to initiate the ready or wait-state signal.

Once the region of address space being accessed has been established, a timing circuit of some sort is normally used to provide a ready indication

to the processor at the appropriate point in the cycle to satisfy each device's unique requirements.

Finally, since indications of ready status from multiple devices are typically present, the signals are logically ORed by using a single gate to drive the  $\overline{\text{RDY}}$  input.

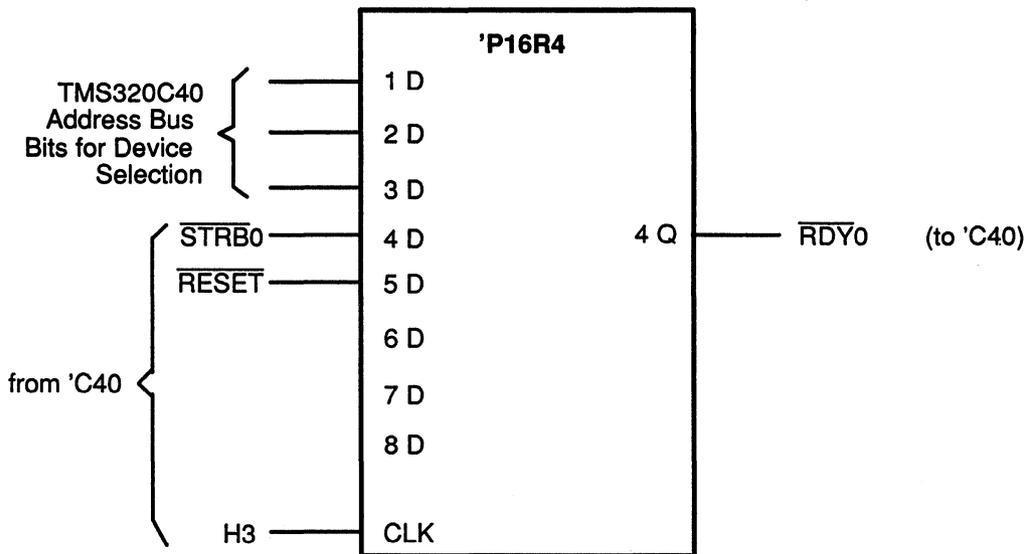
#### 13.4.4 Ready Control Logic

One of two basic approaches may be taken in the implementation of ready control logic, depending upon the state of the ready input between accesses. If  $\overline{\text{RDY}}$  is low between accesses, the processor is always ready unless a wait state is required; if  $\overline{\text{RDY}}$  is high between accesses, the processor will always enter a wait state unless a ready indication is generated.

If  $\overline{\text{RDY}}$  is low between accesses, control of devices that are zero-wait-state at full speed is straightforward; no action is necessary, because ready is always active unless otherwise required. Devices requiring wait states, however, must drive ready high fast enough to meet the input timing requirements. Then, after an appropriate delay, a ready indication must be generated. This can be quite difficult in many circumstances because wait-state devices are inherently slow and often require complex select decoding.

If  $\overline{\text{RDY}}$  is high between accesses, zero-wait-state devices, which tend to be inherently fast, can usually respond immediately with a ready indication. Wait-state devices may simply delay their select signals appropriately to generate a ready. Typically, this approach results in the most efficient implementation of ready control logic. Figure 13-9 shows a circuit of this type, which can be used to generate 0, 1, or 2 wait states for multiple devices in a system.

Figure 13–9. Logic for Generation of 0, 1, or 2 Wait States for Multiple Devices



### 13.4.5 Example Circuit

Figure 13–9 shows how a single, 7-ns 16R4 programmable logic device (PLD) can be used to generate 0, 1, and 2 wait states for multiple devices that are interfaced to a TMS320C40. In this example, distinct address bits are used to select the different wait-state devices. Here, each of the three address lines input to the 16R4 corresponds to a different speed device. For a single 16R4 implementation, up to ten different address bits can be used to select different speed devices.

The single output, 4Q, of the PLD is connected directly to the  $\overline{\text{RDY0}}$  input of the TMS320C40 to signal the completion of a bus access when external wait-state generation is desired (see Section 7.4 on page 7-15 for more information on TMS320C40 wait-state options). Since,  $\overline{\text{RDY0}}$  is sampled on the falling of H1, the H3 output clock is used as the PLD clock input.

Figure 13–10 shows the state machine and equation for programming the 16R4 PLD ready logic. The PLD language shown in this figure is ABEL. STRB0 is an input into the PLD that indicates that a valid TMS320C40 bus cycle is occurring. RESET can also be used to bring the state machine back to the idle state.

Notice that the  $\overline{\text{RDY0}}$  output of the PLD is not registered. An asynchronous  $\overline{\text{RDY0}}$  signal is necessary to generate a ready signal for zero-wait-state devices. When a zero-wait-state device is selected (ahi1 high in Figure 13–10 and STRB0 is low, the PLD asserts  $\overline{\text{RDY0}}$  low within 7 ns. Hence,  $\overline{\text{RDY0}}$

goes active fast enough to satisfy the 20-ns setup time of  $\overline{\text{RDY0}}$  low before H1 low.

For generation of  $\overline{\text{RDY0}}$  for one and two wait states, the device select address bits and  $\overline{\text{STRB0}}$  are delayed one and two cycles, respectively, by the PLD before a  $\overline{\text{RDY0}}$  is brought active low. The one H3-cycle delay required for one-wait-state device ready generation corresponds to state `wait_one` in Figure 13–10 and the two H3-cycle delay required for two-wait-state devices corresponds to state `wait_twoa` and `wait_twob`.

This 16R4 PLD-based design can be used to implement different numbers of wait states for multiple devices. More devices can be selected with TMS320C40 address lines, and a higher number of wait states can be produced with a PLD logic. Furthermore, this approach can be used in conjunction with the TMS320C40's internal wait-state generator.

### 13.4.6 Page Switching Techniques

The 'C40's programmable page switching feature can greatly ease system design when large amounts of memory or slow external peripheral devices are required. This feature can provide a time period for disabling all device selects that would not normally be present otherwise (refer to subsection 7.3.2 on page 7-13 for further information regarding page switching). During this interval, slow devices are allowed time to turn off before other devices have the opportunity to drive the data bus, thus avoiding bus contention.

When page switching is enabled, any time a portion of the high-order address lines changes, as defined by the contents of the  $\overline{\text{STRB0}}$  and  $\overline{\text{STRB1}}$  PAGESIZE fields (in the global and local memory interface control registers), the corresponding  $\overline{\text{STRB}}$  and PAGE go high for one full H1 cycle. Provided that  $\overline{\text{STRB}}$  is included in chip-select decodes, this causes all devices selected by that  $\overline{\text{STRB}}$  to be disabled during this period. The next page of devices is not enabled until  $\overline{\text{STRB}}$  and PAGE go low again.

If the high-order address lines remain constant during a read cycle, the memory access is the same as that of a memory access without page switching. In addition, page switching is not required during writes, because these cycles exhibit an inherent one-half H1 cycle setup of address information before  $\overline{\text{STRB}}$  goes low. Thus, when you use page switching for read/write devices, a minimum of half of one H1 cycle of address setup is provided for all accesses outside a page boundary. Therefore, large amounts of memory can be implemented without wait states or extra hardware required for isolation between pages. Also, note that access time for cycles during page switching is the same as that of cycles without page switching, and, accordingly, full-speed accesses may still be accomplished within each page.

The circuit shown in Figure 13–10 illustrates the use of page switching with the Cypress Semiconductor™ CY7B185 15-ns 8K x 8 BICMOS static RAM. This circuit implements 32K 32-bit words of memory with full-speed zero wait-state accesses within each page.

Figure 13–10. State Machine and Equation for the 16R4 PLD

```

0001 | module      ready_generation
0002 | title'      ready_generation logic for 0, 1 and 2
          wait state devices interfaced to
          TMS320C40'
0003 |
0004 |
0005 |         c40u5  device 'P16R4';
0006 |
0007 |         `inputs
0008 |         h3     Pin 1;
0009 |
0010 |
0011 |         `The following are TMS320C40 address bits used to
0012 |         `select the different speed devices. More can be used if
0013 |         `necessary. In this example, a zero wait state, a one wait
0014 |         `state, and a two wait state device are decoded with these
          `three address bits
0015 |
0016 |         ahi1    Pin 2; `when high selects zero wait state device
0017 |         ahi2    Pin 3; `when high selects one wait state device
0018 |         ahi3    Pin 4; `when high selects two wait state device
0019 |         strb0_  Pin 5; `indicates valid TMS320C40 bus cycle
0020 |         reset_  Pin 6; `reset signal from TMS320C40
0021 |
0022 |         `output
0023 |         rdy0_   Pin 12; `ready signal to TMS320C40
0024 |
0025 |         one_wait Pin 14; `internal flip-flop signal for 1 wait state
0026 |                 `device ready signal generation
0027 |         two_waita Pin 15; `internal flip-flop signal for first of the
two
0028 |         `wait states for 2 wait state devices
0029 |         two_waitb Pin 16; `internal flip-flop signal for second
0030 |                 `of the two wait states for 2 wait
0031 |                 state devices
0032 |
0033 |         `name substitutions for test vectors
0034 |         c,H,L,X = .C.,1,0,.X.;
0035 |
0036 |
0037 |         `state bits
0038 |         outstate = [one_wait, two_waita, two_waitb];
0039 |

```

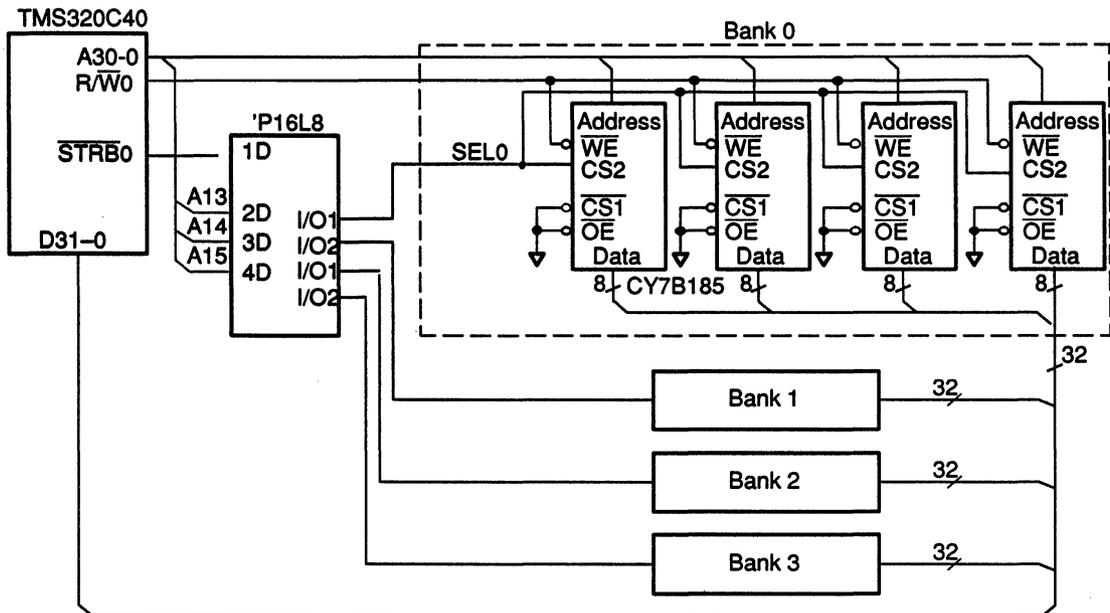
Figure 13-10. State Machine and Equation for the 16R4 PLD (Concluded)

```

0040 |         idle      = ^b111;
0041 |         wait_one  = ^b011;
0042 |         wait_twoa = ^b101;
0043 |         wait_twob = ^b110;
0044 |
0045 |
0046 |state_diagram  outstate
0047 |
0048 |state idle:
0049 |         if (reset_ & ahi2 & !strb0_) then wait_one
0050 |         else if (reset_ & ahi3 & !strb0_) then wait_twoa
0051 |         else      idle;
0052 |
0053 |
0054 |state wait_one:
0055 |         GOTO      idle;
0056 |
0057 |state wait_twoa:
0058 |         if (reset_) then wait_twob
0059 |         else      idle;
0060 |
0061 |state wait_twob:
0062 |         GOTO      idle;
0063 |
0064 |equations
0065 |         !rdy0__ = reset_ & ((ahi1 & !strb0_) # !one_wait #
0066 |         !two_waitb) ;
0067 |
0068 |@page
0069 |"Test 1st level global arbitration logic
0070 |test_vectors
0071 |[ [h3,ahi1,ahi2,ahi3,strb0_, reset_] -> [outstate, rdy0_]
0072 |[ [ c, X, X, X, X, L ] -> [idle, H ];
0073 |[ [ c, L, H, L, L, H ] -> [wait_one, L ];
0074 |[ [ c, X, X, X, X, L ] -> [idle, H ];
0075 |[ [ c, L, L, H, L, H ] -> [wait_twoa, H ];
0076 |[ [ c, X, X, X, X, L ] -> [idle, H ];
0077 |[ [ c, L, L, H, L, H ] -> [wait_twoa, H ];
0078 |[ [ c, X, X, X, X, L ] -> [idle, H ];
0079 |[ [ L, H, L, L, L, H ] -> [idle, L ];
0080 |[ [ c, H, L, L, L, H ] -> [idle, L ];
0081 |[ [ L, L, L, L, L, H ] -> [idle, H ];
0082 |[ [ c, L, H, L, L, H ] -> [wait_one, L ];
0083 |[ [ c, X, X, X, X, H ] -> [idle, H ];
0084 |[ [ c, L, L, H, L, H ] -> [wait_twoa, H ];
0085 |[ [ c, L, L, H, L, H ] -> [wait_twob, L ];
0086 |[ [ c, H, L, L, L, H ] -> [idle, L ];
0087 |[ [ c, X, X, X, H, H ] -> [idle, H ];
0088 |[ [ c, X, X, X, H, H ] -> [idle, H ];
0089 |end      ready_generation

```

Figure 13–11. Page Switching for the Cypress Semiconductor™ CY7C185



A 5-ns, '16L8 PLD decodes lines A15 – A13. These lines along with  $\overline{\text{STRB0}}$  select each of the four pages in this circuit. With the PAGESIZE field of  $\overline{\text{STRB0}}$  of the global memory interface control register set to 0Ch, the pages are selected on even 8K-word boundaries, starting at location zero in external memory space.

This circuit cannot be implemented without page switching, because data output's turn-on and turn-off delays cause bus conflicts, and full-speed accesses do not allow enough time for chip-select decoding for the four pages. Here, the propagation delay of the 16L8 is involved only during page switches, where there is sufficient time between cycles to allow new chip-selects to be decoded.

The timing of this circuit for read operations using page switching is shown in Figure 13–12. When a page switch occurs, the page address on address lines A30 – A13 is updated during the extra H1 cycle while  $\overline{\text{STRB0}}$  is high. Then, after chip-select decodes have stabilized and the previously selected page has disabled its outputs,  $\overline{\text{STRB}}$  goes low for the next read cycle. Further accesses occur at full speed with the normal bus timings, as long as another page switch is not necessary. Write cycles do not require page switching, because of the inherent address setup provided in their timings.

This timing is summarized in Table 13–3.

Figure 13-12. Timing for Read Operations Using Bank Switching

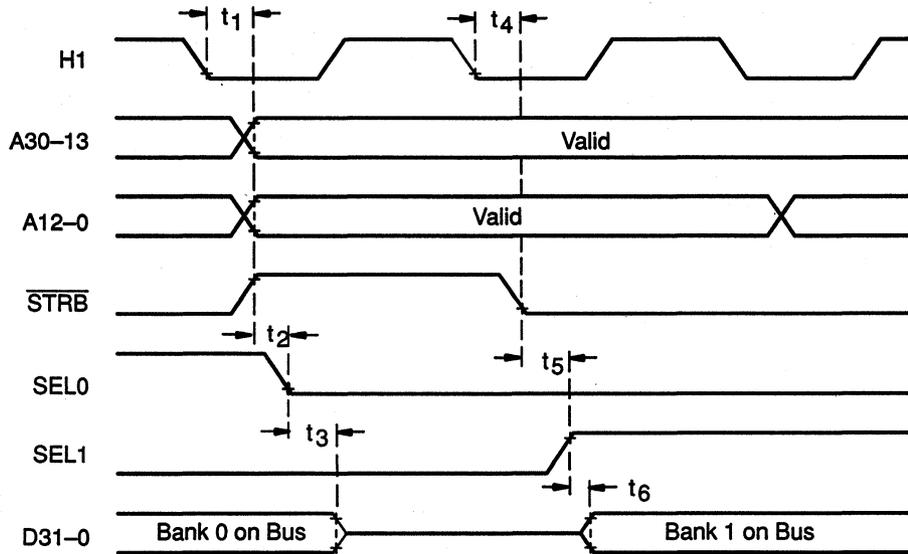


Table 13-3. Page Switching Interface Timing

Time Interval	Event	Time Period
$t_1$	H1 falling to address/STRB valid	7 ns
$t_2$	STRB to select delay	5 ns
$t_3$	Memory disable from select	8 ns
$t_4$	H1 falling to STRB	7 ns
$t_5$	STRB to select delay	5 ns
$t_6$	Memory output enable delay	3 ns

## 13.5 Parallel Processing Interfaces

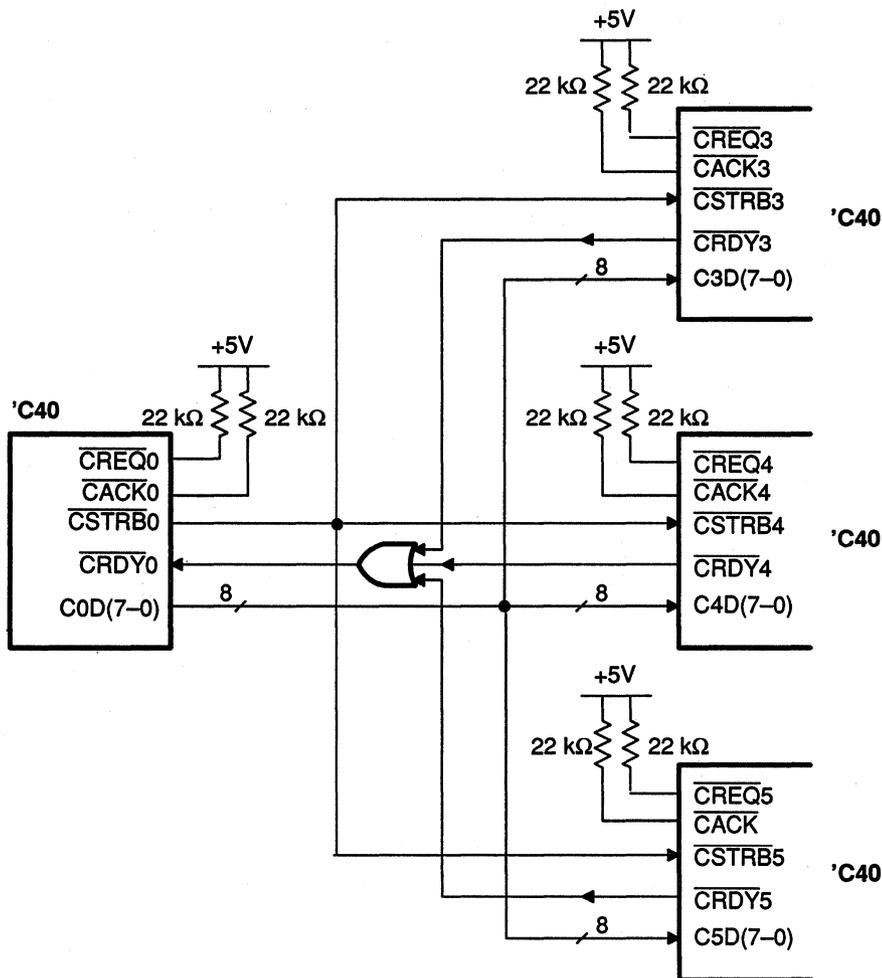
The 'C40 communication ports and support for shared memory are the keys to parallel processing design flexibility. Almost any number of processors can be linked together in a wide variety of configurations. In this section, Figure 13–14 (in three parts) illustrates 'C40 parallel processing configurations that are used to fulfill many signal processing system needs.

### 13.5.1 Message Broadcasting From One TMS320C40 to Many TMS320C40's

Message broadcasting from one 'C40 to many 'C40s requires a simple interface. The block diagram of one is shown in Figure 13–13. To simplify the interface, no token transferring is done. In this design, one 'C40 is the dedicated transmitter, and three 'C40s are dedicated receivers. No reset circuitry is needed because of the transmitter is communication port 0 and the receivers are communication ports 3, 4, and 5. At reset, 'C40 communication ports 0, 1, and 2 are output ports, and communication ports 3, 4, and 5, are input ports. Due to this fixed communications configuration, no token transfer is needed, allowing the  $\overline{CREQ}$  and  $\overline{CACK}$  pins of all processors to be individually pulled up to 5 volts through 22-k $\Omega$  resistors. Also, the  $\overline{STRB}$  pins of the communicating processors can be tied together along with the data lines CD7–0. However, if more than 5 receivers must be driven by a single transmitter at the 'C40s rated speed, the  $\overline{STRB}$  and CD7–0 lines need to be buffered. Since the 'C40 communication ports protocol is asynchronous, if the speed of broadcast is not critical, buffers are not needed as long as the number of receivers is less than 30. The  $\overline{CRDY}$  signal input by the transmitter communication port is generated by ORing the  $\overline{RDY}$  outputs of all of the receiver communication ports. The transmitter should not receive a  $\overline{RDY}$  signal until the receiver has received all data.

In addition, to ensure that the dedicated receiver 'C40s do not try to arbitrate for the communication port bus, you should halt the output ports of the receiver 'C40s by setting bit four of their communication port control registers to one.

Figure 13-13. Message Broadcasting by One 'C40 to Many 'C40s



### 13.5.2 Shared Global Memory Interface With Fair Bus Arbitration

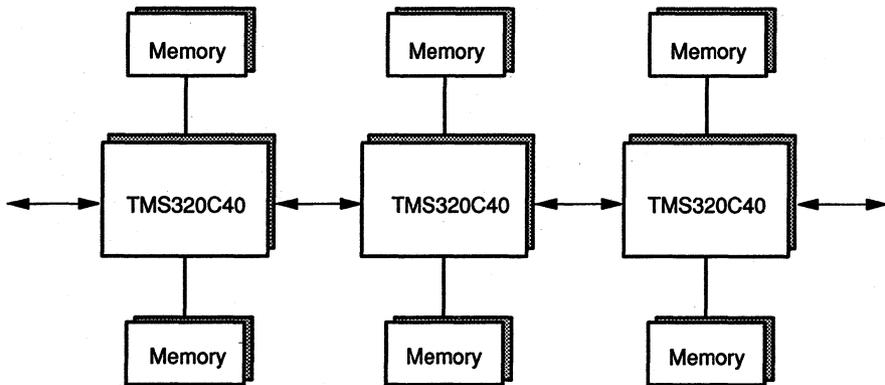
One of the most common multiprocessing system configurations is memory shared by each processor in a system. Shared memory is typically implemented by tying the processors' data and address lines together. However, the shared memory interface must guarantee that no more than one processor is driving the shared bus at any one time; it must also allow all processors sharing the bus to have a chance to access shared resources.

The 'C40 supports shared memory multiprocessing with its identical global and local port interfaces. Both interfaces have four status output signals, (L)STAT3-0, which identify what type of access is attempting to begin on

the bus. These signals identify whether the 'C40 port is idle, a DMA read is occurring, a  $\overline{\text{STRB1}}$  write is occurring, a  $\overline{\text{LOCKed}}$  access to memory is pending, etc. (as listed in Table 7-2, page 7-5). These signals can be interpreted to issue single access or locked access bus requests to a shared bus arbiter.

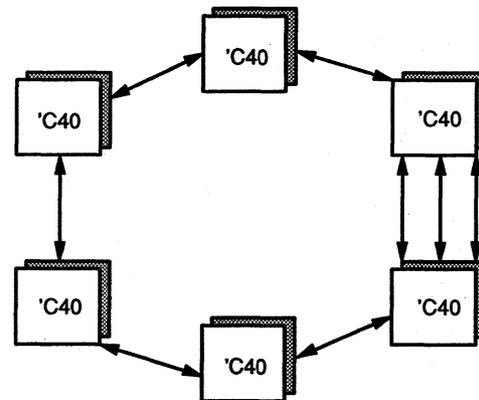
To support shared address control and data lines, the 'C40 provides the  $\overline{\text{(L)CE}}$ ,  $\overline{\text{(L)AE}}$ , and  $\overline{\text{(L)DE}}$  input signals. When disabled (made high), these signals three-state the control signals, address lines, and data lines, respectively, of the port. These bus enable lines are asynchronous inputs to the 'C40, which can quickly turn off bus drivers when another processor is accessing a shared resource. However, these signals asynchronously turn off the 'C40's local and global buses, without memory accesses being suspended. To ensure that data written is seen externally and data read is valid, the external  $\overline{\text{(L)RDY}}$  should be used for wait-state generation in shared memory designs. An  $\overline{\text{(L)RDY}}$  signal should not be sent to the 'C40 until the processor has regained access to the bus ( $\overline{\text{CE}}$ ,  $\overline{\text{AE}}$ ,  $\overline{\text{DE}}$  enabled) and has had enough time to complete its access. Hence with bus enable and status signals, the flexible bus interfaces of the 'C40 allow high-speed shared bus configurations to be implemented.

Figure 13-14. TMS320C40 Parallel DSP System Architectures



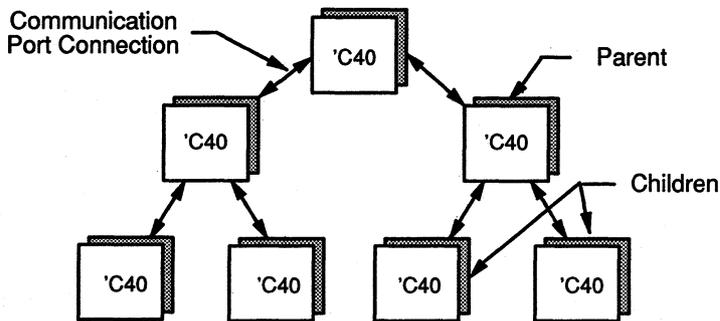
**PIPELINED LINEAR ARRAY**

For convolution and correlation and other pipelined operations in graphics and modem applications.



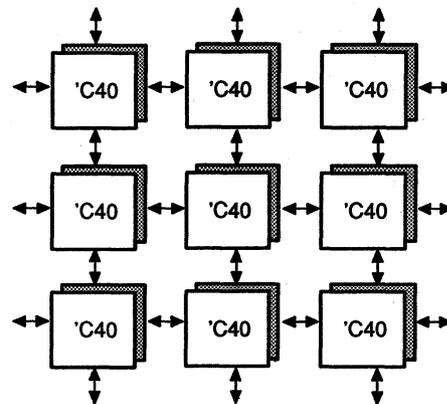
**BIDIRECTIONAL RING**

Clockwise and counterclockwise data flow. Group port for more I/O. Very effective for neural networks.



**TREE STRUCTURES**

Supports broadcasting and data searches for speech and image recognition applications.



**2D ARRAY**

Excellent for image processing.

Figure 13-14. TMS320C40 Parallel DSP System Architectures (Continued)

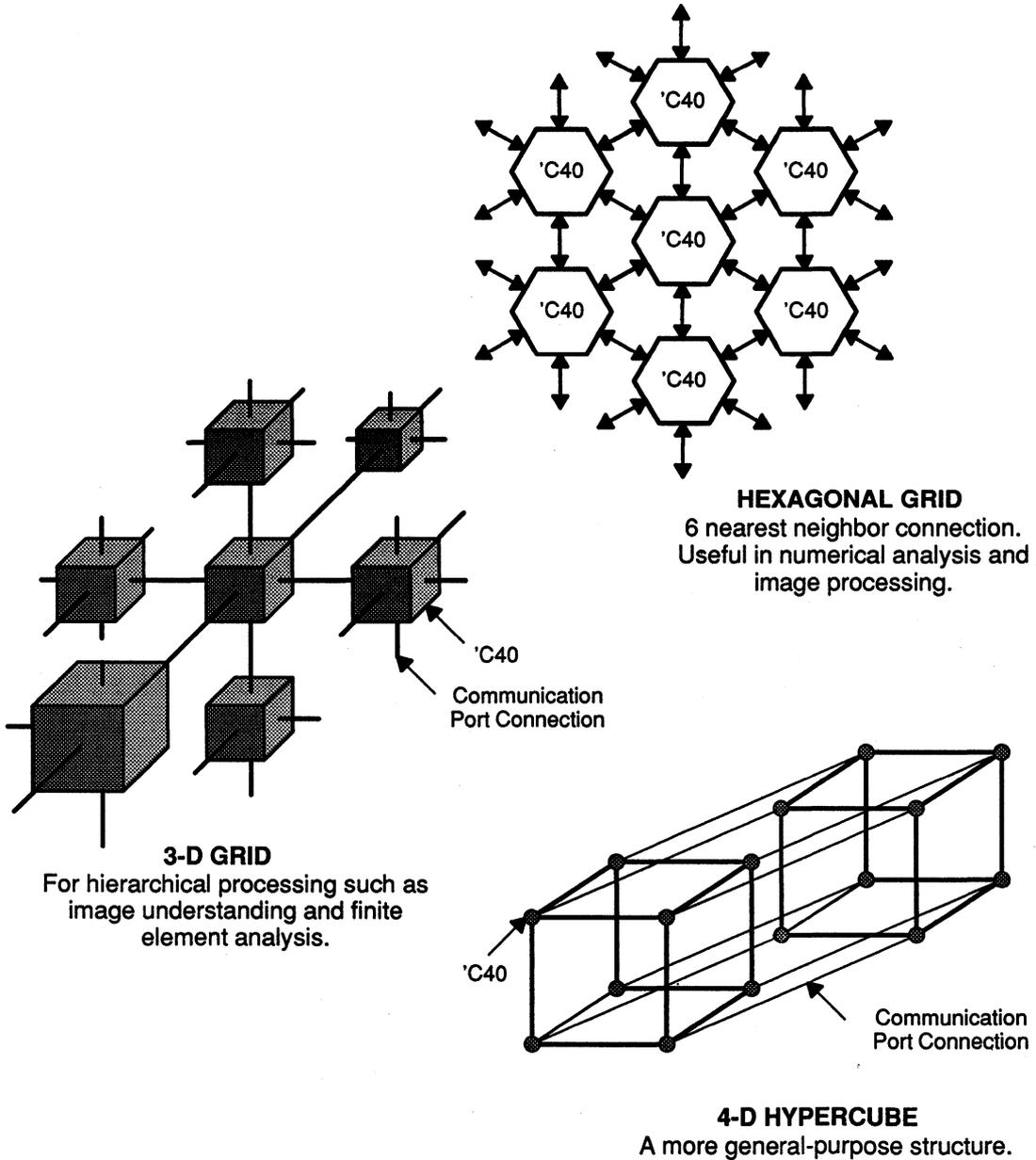
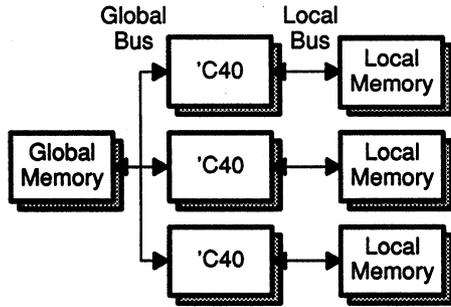
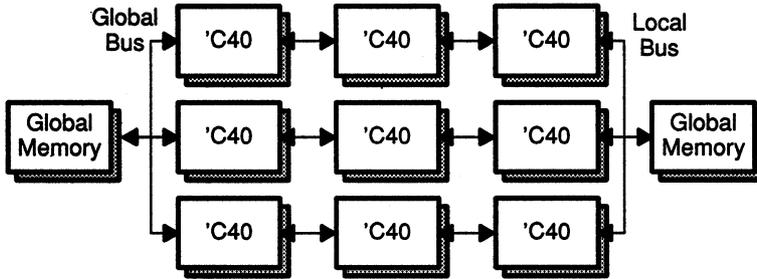


Figure concluded on next page

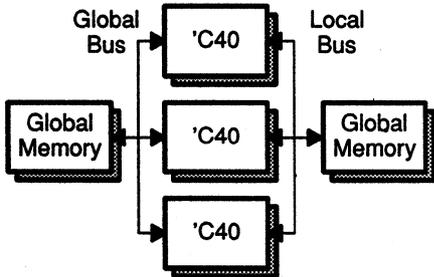
Figure 13-14. TMS320C40 Parallel DSP System Architectures (Concluded)



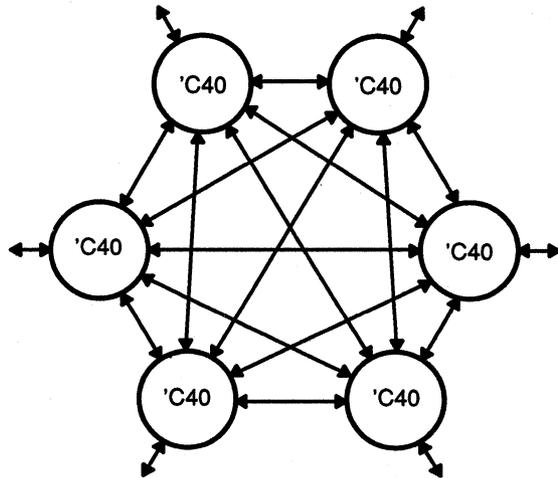
Memory interfaces support shared global memory and private local memory.



Architectures utilizing shared memory and communication ports are possible.



Memory interfaces also support shared global memory on the global bus and the local bus.



A truly limitless variety of configurations are possible.

In this section, a 'C40 shared memory example is shown. Four 'C40s share SRAM with their global buses tied together. A bus arbitrator implemented as a programmable logic device provides a fair scheme for processor access to the shared bus. The design shown here uses high speed parts but employs a fully asynchronous handshake protocol, which is still general, allowing varying speed 'C40s and processors other than 'C40s to be added to this bus configuration.

### 13.5.3 Shared Bus Interface Overview

Figure 13–15 and Figure 13–16 are *examples* of shared memory configurations. In these figures:

- Four 'C40s (each as shown in Figure 13–15) have their global buses tied together,
- Each shares 128K × 32 of one-wait-state SRAM,
- 64K of the memory is controlled by R/W0;  $\overline{\text{STRB0}}$  and the other 64K are controlled by R/W1 and  $\overline{\text{STRB1}}$ .

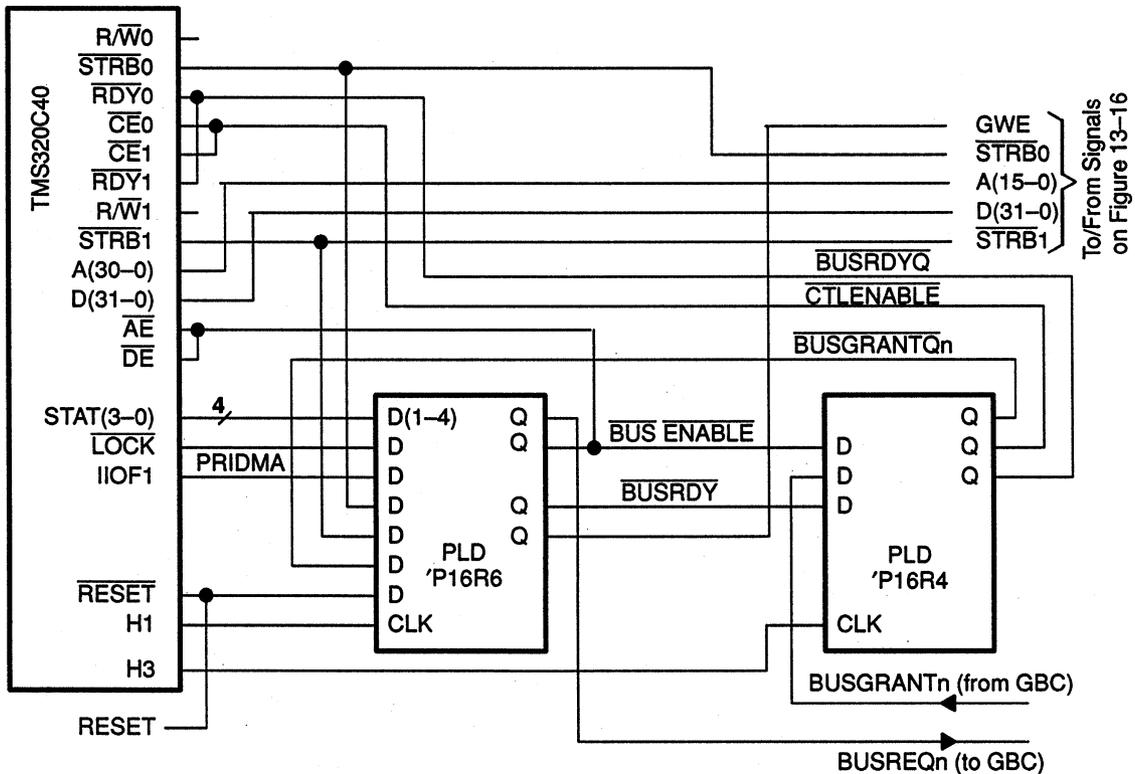
The memory devices are organized as 64K × 4, 35-ns SRAMs. Due to the 'C40's bus enable signals— $\overline{\text{AE}}$ ,  $\overline{\text{DE}}$  and  $\overline{\text{CE}}$ —all four 'C40s' data, address, and control lines can be tied together for a shared memory configuration. However, since 128K words of shared memory are being implemented on the global bus (shown in Figure 13–15 and Figure 13–16), the common address lines are buffered to provide adequate drive to the 16 required memory devices. Also, the memories' chip-enable lines are pulled up to 5 volts through 22-k $\Omega$  resistors to ensure that the memory devices are disabled when no 'C40 is accessing them.

The required shared global bus interface logic consists of two levels of bus arbitration logic implemented as programmable logic devices (PLD). Each of the 'C40s has an identical first level of logic that interfaces to the shared second level arbiter. The first level of logic for each of the four 'C40s consists of one 7-ns 16R6 PLD and one 7-ns 16R4 PLD (center of Figure 13–15). Each first level 16R6 PLD receives status and control signals from the corresponding 'C40, determines what kind of global bus transfer the associated 'C40 requires, and issues a global bus request signal to the global bus controller (GBC, bottom of Figure 13–16), which, with the bus-grant time-out counter, implements the second level of arbitration logic. The GBC is implemented with a 7-ns 16R8 PLD, and the timeout counter is implemented with a 7-ns 16R4 PLD. In addition to the two GBC PLDs, a 16L8 PLD is used to issue write enable signals to the shared memory.

Since typical high-speed PLDs do not have many registered I/O pins or multiple clock sources, each first-level 16R6 PLD uses a 16R4 PLD to synchronize some of the input and output signals, and the 16R8 GBC PLD uses external flip-flops to synchronize input signals.

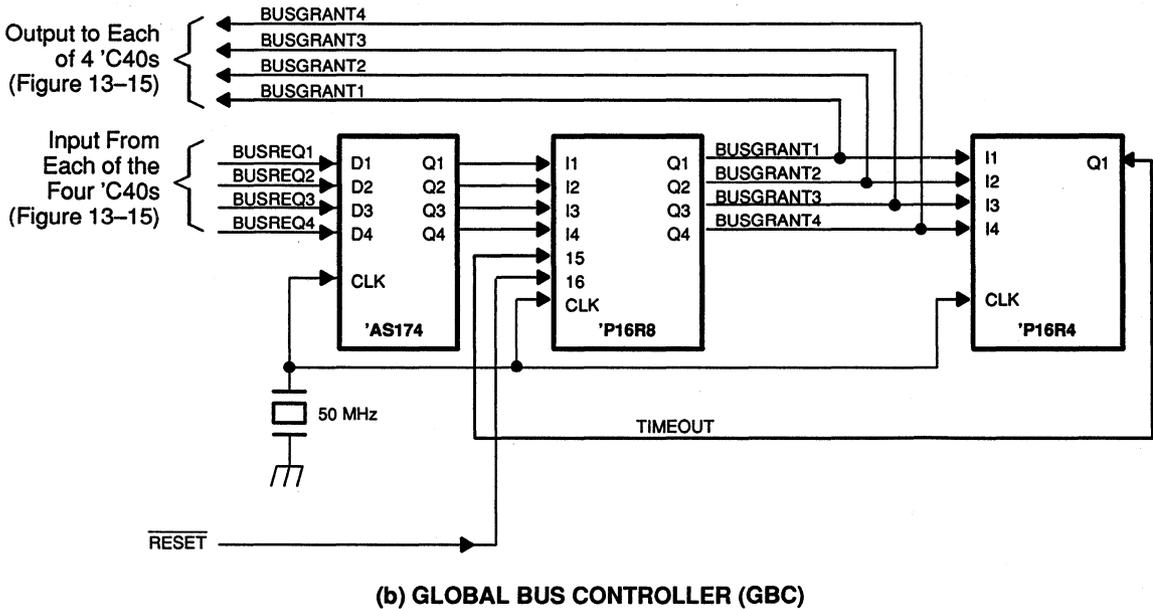
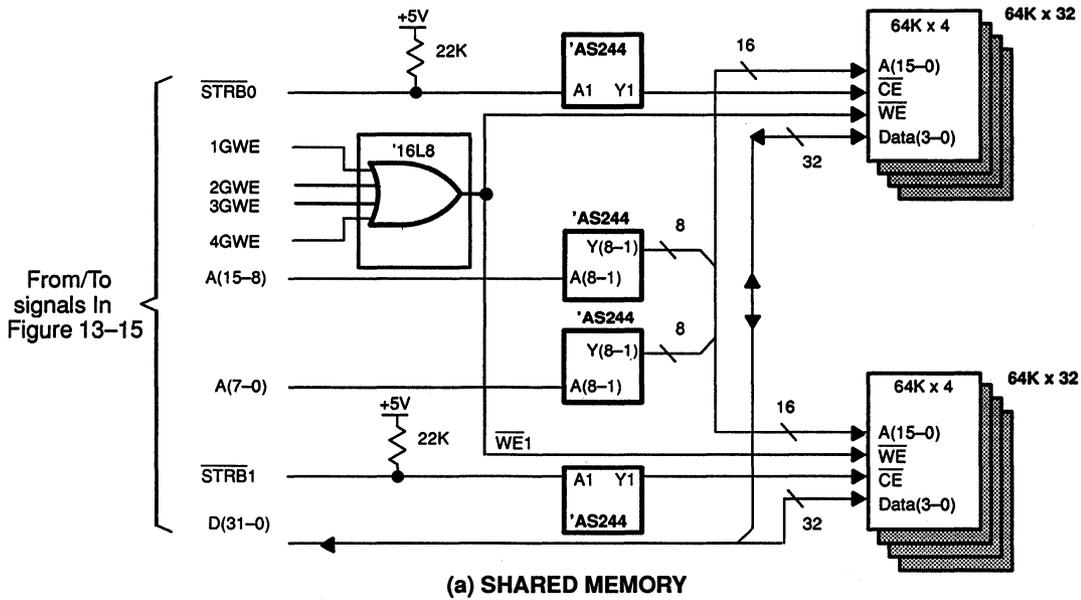
If a 'C40 requires uninterrupted, multicycle global bus transfers, the first-level PLD keeps its bus-request signal active until the uninterruptable cycles are complete. The bus controller performs arbitration between the 'C40s requesting the shared global bus. If a 'C40 is given access to the bus, the bus controller sends its first-level PLDs a bus grant signal. The first-level PLD then sends a bus enable signal to the 'C40, which brings its bus control, address, and data signals out of high impedance. The first-level PLD also sends a BUSRDYQ signal to the 'C40 to end each read or write cycle.

Figure 13-15. TMS320C40 Shared Memory Interface



- Notes:**
- 1) This figure represents one of four 'C40s and its interface (the 3 other 'C40s in the system have the same configuration).
  - 2) The shared memory (shared by the four 'C40s) and global bus controller are shown in Figure 13-16 on the next page.
  - 3) The fixed/rotating priority is a programmable option at the global bus controller (GBC).

Figure 13-16. TMS320C40 Shared Memory and Bus Controller Interface



For full-speed operation, the 'C40s run from separate, 50-MHz crystal-oscillator clock sources. For synchronization of shared bus control signals, the H3 output clock of each 'C40 serves as the 16R4 PLD synchronizer clock for first-level input and output signals. Also, the H1 output clock of each 'C40 serves as the state machine clock for each of the first-level, 16R6 PLDs. In addition, for high-speed bus controller synchronization, a 50-MHz crystal oscillator is used as the input clock for the 16R8 GBC PLD, the 16R4 timeout generator PLD, and the GBC input signal synchronizers. (Note: for fastest bus arbitration, the 'C40s sharing the bus can be synchronized by having common RESET and CLKIN inputs. If the 'C40s are synchronized in this way, the 50-MHz input to the second-level, global control PLDs can be the common CLKIN.) The AS174 D flip-flops are used as GBC input signal synchronizers.

Due to these arbitration synchronizer delays and the 35-ns SRAMs, access to the shared memory requires wait states. After an arbitration win, the first shared memory access requires three H1 cycles, and arbitration requires at least two H1 cycles from BUS REQUEST active to BUS ENABLE active. Figure 13-17 is a timing diagram of the arbitration contest. A bus master's first access after an arbitration win takes at least three H1 cycles; however, subsequent read or write accesses require only two H1 cycles. The three-cycles required for the first access provide enough time for the old bus master to stop driving the bus after an arbitration loss and enough time for new bus master control signals to go active and inactive to complete 35-ns memory accesses. Also, three-cycle memory accesses allow enough time for signal buffering (buffer delays are less than 15 ns with commercially available parts) between the processor bus and memory.

The subsection that follows covers the global bus configuration for use with this shared memory configuration.

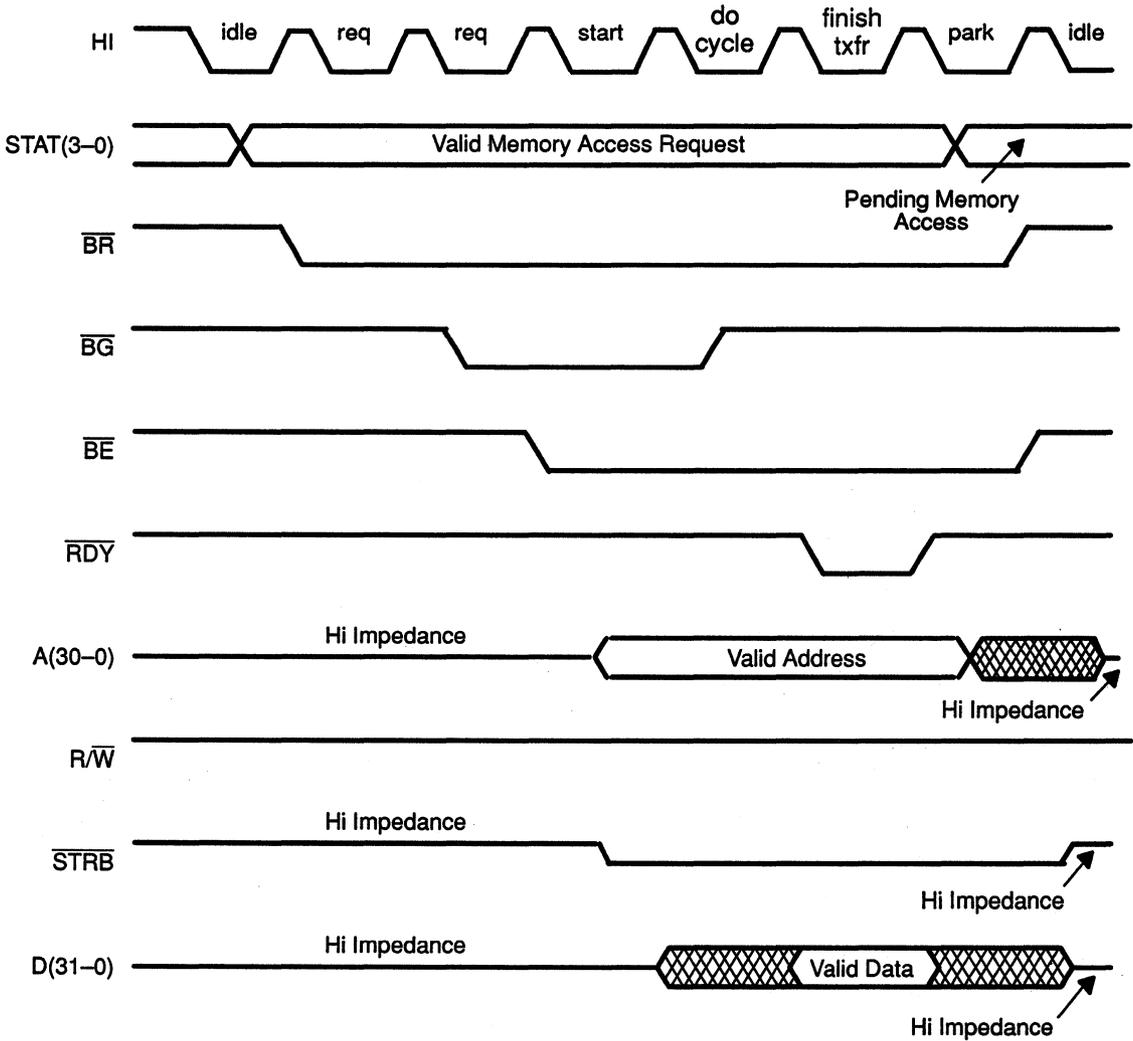
### 13.5.3.1 Global Memory Interface Control Register (GMICR) Configuration

For use in this shared memory configuration, the global bus should be configured as such at the GMICR:

```
SWW = 00 (RDYint = RDYext)
STRB ACTIVE = 011112
PAGESIZE = 011112
STRB SWITCH = 0
```

In addition, IIOF1 should be configured as a general-purpose output pin. IIOF1 high signals that a high-priority DMA request is active.

Figure 13-17. Successful TMS320C40 Arbitration and Data Read From Shared Bus Memory Followed by an Unsuccessful Arbitration Contest



## 13.6 Bus Arbitration

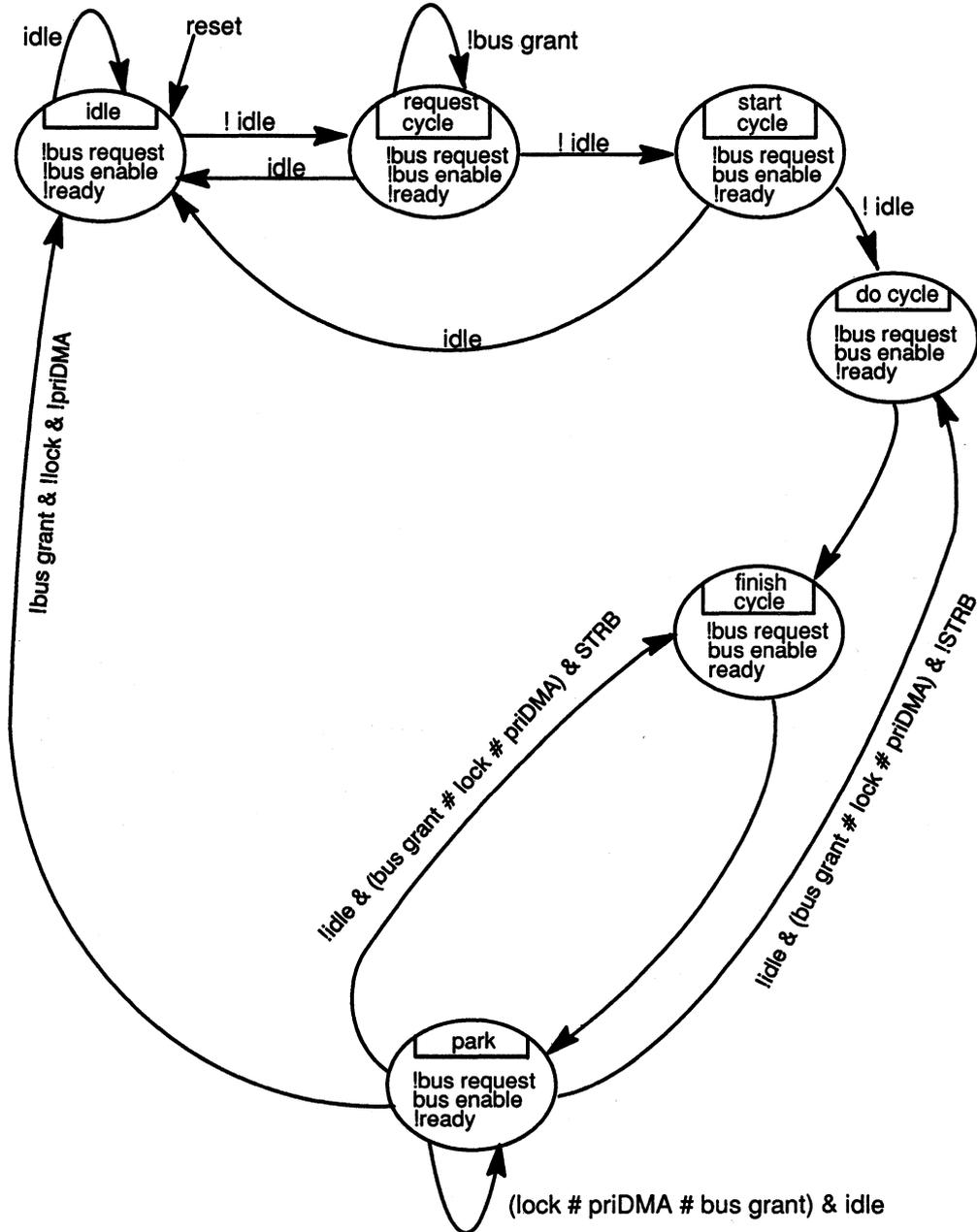
### 13.6.1 Arbitration Implementation

Arbitration on the bus is implemented with two levels of logic. The first level consists of four identically programmed 7-ns, 16R6 PLDs, and four identically programmed 16R4 PLDs, with one 16R4 and 16R6 associated with each 'C40. The signals needed for arbitration from the 'C40 are STAT(3-0),  $\overline{\text{STRB0}}$ ,  $\overline{\text{STRB1}}$ ,  $\overline{\text{LOCK}}$ , and IIOF1 (PAGE can be used in designs where page switching is necessary). IIOF1 should be configured as an output pin and should be used to indicate that a high-priority DMA transfer is active. Applications software should set IIOF1 low before priority DMA cycles are started. Figure 13-18 illustrates graphically the state machine for each of the first-level 16R6 PLDs. Each first-level PLD sends an active low  $\overline{\text{BUSREQ}}$  signal to the global bus controller, the second level of arbitration logic. The global bus controller sends a  $\overline{\text{BUSGRANT}}$  signal to the requesting 'C40's first level logic when it has been granted control of the global bus. If an interlocked or high-priority DMA bus request has been granted, the first-level logic will keep its  $\overline{\text{BUSREQ}}$  asserted low as long as interlocked or priority DMA cycles are required. The bus controller will see  $\overline{\text{BUSREQ}}$  remain active and will give the current 'C40 bus master access to the bus until the interlocked or priority DMA operations are complete.

After a high-priority DMA bus cycle is complete, the 'C40 applications software should clear (set IIOF1 to logic level 1). Accordingly, interlocked access to memory should always end in a SIGI, STII, or STFI operation to bring  $\overline{\text{LOCK}}$  inactive. If priority accesses are completed by making IIOF1 or  $\overline{\text{LOCK}}$  inactive, the first-level PLD will always have an opportunity to bring its  $\overline{\text{BUSREQ}}$  inactive, preventing shared bus deadlock.

When the 'C40 associated with a first-level PLD is not the global bus master (i.e., cannot access the global bus), the first-level PLD sends a logic level one  $\overline{\text{BUSREADY}}$  signal to that 'C40, extending any pending bus cycle until after the 'C40 becomes bus master and has completed an access. In addition, each first-level of logic sends both a  $\overline{\text{BUSENABLE}}$  and  $\overline{\text{CTLENABLE}}$  signal to the corresponding 'C40. The  $\overline{\text{BUSENABLE}}$  signal is connected to the  $\overline{\text{DE}}$  and  $\overline{\text{AE}}$  pins and  $\overline{\text{CTLENABLE}}$  is connected to the  $\overline{\text{CE}}$  pin of the corresponding 'C40. These two signals cause the following to be in high-impedance when another 'C40 in the system is accessing the shared bus: bus chip enable and the address and data lines.

Figure 13-18. Shared Bus Interface PLD State Machine



- Notes:**
- 1) In this state diagram, the output signals are all shown as active high for diagram clarity.
  - 2) A "!" in front of a signal indicates that it is not active (deasserted).
  - 3) & = logical AND of signals; # = logical OR of signals.

For proper system reset operation, a  $\overline{\text{RESET}}$  signal clears bus requests to the global bus controller and sends a logic level one  $\overline{\text{BUSRDY}}$  and  $\overline{\text{BUSENABLE}}$  signal to each 'C40 to extend upcoming bus cycles and three-state the bus until that 'C40 has been granted access to the bus.

Figure 13–19 shows equations for programming the 16R6 PLD used for the first-level logic. The PLD language shown in this figure is ABEL. ABEL's PLD language is used to describe the state machine illustrated in Figure 13–18.

---

**Note: Active-Low Indicators**

In listings (e.g., Figure 13–19), an underscore following a signal name (e.g., `busreq_`) indicates the signal is active low. (in regular text, such signals are overbarred (e.g.,  $\overline{\text{BUSREQ}}$ ).

---

The three PLD outputs — `busreq_`, `busenable_`, and `busrdy_` — are used for three of the output state bits. The `park_state` and `start_state` bits (used to indicate the park state and start state) are the fourth and fifth output state bits. Also included in the ABEL description are test vectors for the state machine.

The PLDs described in Figure 13–19 and Figure 13–20 work together to interface to the GBC. Figure 13–20 (page 13-60) shows equations for programming the 16R4 PLD used for synchronizing the first-level input and output signals.

Figure 13–19. PLD Equations for Programming the 16R4 PLD (First-Level Logic)

```

0001 |module      C40_local_glob_bus_interf
0002 |title'
0003 |DWG NAME    Local control (of shared bus arbitration
      |         (logic)
0004 |
0005 |DWG #
0006 |
0007 |COMPANY     TEXAS INSTRUMENTS INCROPORATED'
0008 |
0009 |c40u2       device          'P16R6';
0010 |
0011 |   "inputs for global interface logic
0012 |   h1                Pin 1;    "clock input
0013 |   priDMA_           Pin 2;    "flag req output used to signal
      |                   priority DMA
0014 |   stat3             Pin 3;    "stat3=0 STRB0 access, stat3=1
      |                   STRB1 access
0015 |   stat2             Pin 4;
0016 |   stat1             Pin 5;
0017 |   stat0             Pin 6;
0018 |   strb0_            Pin 7;
0019 |   strb1_            Pin 8;    "rdy signal from the external
      |                   expansion connector
0020 |   bg_               Pin 9;    "busgrant (from bus arbiter)
0021 |   lock_             Pin 12;
0022 |   reset_            Pin 19;
0023 |
0024 |   "outputs for global interface logic
0025 |   start_state       Pin 18;    "low if the output state is the
      |                   strt_cycle state
0026 |   busreq_           Pin 17;
0027 |   busenable_        Pin 16;
0028 |   busrdy_           Pin 15;
0029 |   park_state        Pin 14;    "low if the output state is the
      |                   park state
0030 |
0031 |   gwe_              Pin 13;    "write enable signal for shared
      |                   memory
0032 |
0033 |   "define machine state bits
0034 |   "[start,park,busreq_,busenable_,busrdy_];
0035 |
0036 |   idle              =        ^b111111;    "31
0037 |   req_cycle         =        ^b11011;    "27
0038 |   strt_cycl        =        ^b01001;    "09
0039 |   do_cycle          =        ^b11001;    "25

```

Figure 13-19. PLD Equations for Programming the 16R4 PLD (First-Level Logic) (Continued)

```

|040 |   fin_cycle = ^b11000;  "24
0041 |   park      = ^b10001;  "17
0042 |
0043 |   "convert to positive logic to make the test vectors easier to
      understand
0044 |
0045 |   lock      = !lock_;
0046 |   bg        = !bg_;
0047 |   priDMA    = !priDMA_;
0048 |   idle_stat = (stat2 & stat1 & stat0);  "the bus is idle
      when all are hi
0049 |
0050 |   "outstate
0051 |   ost       = [start_state,park_state,busreq_,
      busenable_,busrdy_];
0052 |
0053 |   c,H,L,X   = .C.,1,0,.X.;
0054 |@page
0055 |
0056 |state diagramost
0057 |
0058 |state idle:
0059 |   case (!reset_ # idle_stat)           :idle;
0060 |       ( reset_ & !idle_stat )         :req_cycle;
0061 |   endcase;
0062 |
0063 |
0064 |state req_cycle:
0065 |   case (!reset_ # idle_stat)           :idle;
0066 |       ( reset_ & bg_ & !idle_stat ) :req_cycle;
0067 |       ( reset_ & !bg_ & !idle_stat ):strt_cycl;
0068 |   endcase;
0069 |
0070 |
0071 |
0072 |state strt_cycl:
0073 |   case (!reset_)                       :idle;
0074 |       ( reset_)                        :do_cycle;
0075 |   endcase;
0076 |
0077 |
0078 |state do_cycle:
0079 |   case (!reset_)                       :idle;
0080 |       ( reset_)                        :fin_cycle;

```

Figure 13-19. PLD Equations for Programming the 16R4 PLD (First-Level Logic) (Continued)

```

0081 |         endcase;
0082 |
0083 |
0084 |
0085 | state fin_cycle:
0086 |     case
0087 |         (!reset_)      :idle;
0088 |         ( reset_)      :park;
0089 |     endcase;
0090 |
0091 |
0092 |
0093 | state park:
0094 |     case (!reset_ # (bg_ & lock_ & priDMA_)) :idle;
0095 |         ( reset_ & idle_stat & (!bg_ # !lock_ # !priDMA_))
0096 |         :park;
0097 |         ( reset_ & !idle_stat & ((!stat3 & strb0_) #
0098 |         (stat3 & strb1_))
0099 |         (!bg_ # !lock_ # !priDMA_)) :do_cycle;
0100 |         ( reset_ & ((!stat3 & !strb0_) # (stat3 & !strb1_))
0101 |         & (!bg_ # !lock_ # !priDMA_)) :fin_cycle;
0102 |
0103 |     endcase;
0104 |
0105 | equations
0106 |     !gwe_ := reset_ & stat2 & !idle_stat & ((!bus
0107 |     req_ & !bg_) # !busenable_);
0108 | @page
0109 | "Test 1st level global arbitration logic
0110 | test_vectors
0111 | ([h1,stat3,stat2,stat1,stat0,lock_,priDMA,strb0_,bg,strb1_,
0112 |     reset_]->[ost,gwe_])
0113 | [ c, X, H, H, H, X, X, X, X, H, L] -> [ idle,H];
0114 | [ c, X, H, L, H, H, L, X, L, H, H] -> [ req_cycle,H];
0115 | [ c, X, H, H, H, X, X, X, X, H, L] -> [ idle,H];
0116 | [ c, X, X, X, L, X, X, X, L, H, H] -> [ req_cycle,H];
0117 | [ c, X, H, H, X, L, X, X, X, H, H] -> [ strt_cycl,L];
0118 | [ c, X, H, H, H, X, X, X, X, H, L] -> [ idle,H];
0119 | "vector 7
0120 | [ c, X, X, X, L, X, X, X, L, H, H] -> [ req_cycle,H];

```

Figure 13-19. PLD Equations for Programming the 16R4 PLD (First-Level Logic) (Continued)

```

0121 |[ c, X, H, X, L, X, X, X, H, H, H] -> [strt_cycl,L];
0122 |[ c, X, H, X, L, X, X, X, H, H, H] -> [do_cycle,L];
0123 |[ c, X, H, H, H, X, X, X, X, H, L] -> [idle,H];
0124 |
0125 |[ c, X, H, L, H, H, L, X, L, H, H] -> [req_cycle,H];
0126 |[ c, X, H, X, L, X, X, X, H, H, H] -> [strt_cycl,L];
0127 |[ c, X, H, X, L, X, X, X, H, H, H] -> [do_cycle,L];
0128 |[ c, X, H, X, L, X, X, H, H, H, H] -> [fin_cycle,L];
0129 |[ c, X, H, H, H, X, X, X, X, H, L] -> [idle,H];
0130 |"vector 16
0131 |[ c, X, X, X, L, X, X, X, L, H, H] -> [req_cycle,H];
0132 |[ c, X, H, X, L, X, X, X, H, H, H] -> [strt_cycl,L];
0133 |[ c, X, H, X, L, X, X, X, H, H, H] -> [do_cycle,L];
0134 |[ c, X, H, X, L, X, X, H, H, H, H] -> [fin_cycle,L];
0135 |[ c, L, H, X, L, X, X, H, H, H, H] -> [park,L];
0136 |[ c, X, H, H, H, X, X, X, X, H, L] -> [idle,H];
0137 |
0138 |
0139 |
0140 |[ c, X, X, X, L, X, X, X, L, H, H] -> [req_cycle,H];
0141 |[ c, X, H, X, L, X, X, X, H, H, H] -> [strt_cycl,L];
0142 |[ c, X, H, X, L, X, X, X, H, H, H] -> [do_cycle,L];
0143 |[ c, X, H, X, L, X, X, H, H, H, H] -> [fin_cycle,L];
0144 |[ c, L, H, X, L, X, X, H, H, H, H] -> [park,L];
0145 |
0146 |"vector 09
0147 |[ c, X, H, X, X, H, L, X, L, H, H] -> [idle,L];
0148 |[ c, X, H, H, H, X, X, X, X, H, H] -> [idle,H];
0149 |[ c, X, L, H, L, H, L, X, L, H, H] -> [req_cycle,H];
0150 |[ c, , X, L, H, L, X, X, H, L, H] -> [req_cycle,H];
0151 |[ c, X, L, H, L, X, X, H, H, H, H] -> [strt_cycl,H];
0152 |[ c, X, L, H, L, X, X, H, L, H, H] -> [do_cycle,H];
0153 |[ c, X, L, H, L, H, L, H, L, H, H] -> [fin_cycle,H];
0154 |[ c, L, L, H, L, H, L, H, L, H, H] -> [park,H];
0155 |[ c, X, X, X, X, H, L, H, L, H, H] -> [idle,H];
0156 |
0157 |"vector 18
0158 |[ c, X, H, H, H, L, X, H, L, H, H] -> [idle,H];
0159 |[ c, H, L, X, H, L, X, H, L, H, H] -> [req_cycle];
0160 |[ c, H, L, X, H, L, X, H, L, H, H] -> [req_cycle,H];
0161 |[ c, H, L, X, H, X, X, H, H, H, H] -> [strt_cycl,H];
0162 |[ c, H, L, X, H, L, X, H, H, L, H] -> [do_cycle,H];
0163 |[ c, H, L, X, H, H, L, H, L, L, H] -> [fin_cycle,H];

```

Figure 13-19. PLD Equations for Programming the 16R4 PLD (First-Level Logic) (Continued)

```

0164 |[ c, H, L, X, H, H, L, H, X, L, H] -> [park,H];
0165 |[ L, H, L, X, H, H, L, H, X, H, H] -> [park,H];
0166 |[ c, H, L, X, H, H, H, H, L, L, H] -> [fin_cycle,H];
0167 |[ c, H, L, X, H, H, H, H, X, L, H] -> [park,H];
0168 |[ L, X, L, X, H, H, H, H, X, H, H] -> [park,H];
0169 |[ c, L, L, L, H, L, X, H, H, H, H] -> [do_cycle,H];
0170 |[ c, L, L, L, H, H, L, H, L, H, H] -> [fin_cycle,H];
0171 |[ c, L, L, L, H, H, L, H, X, L, H] -> [park,H];
0172 |[ c, H, H, H, H, H, H, H, X, H, H] -> [park,H];
0173 |[ c, H, H, H, H, H, H, H, X, H, H] -> [park,H];
0174 |[ c, X, X, X, X, H, L, H, L, H, H] -> [idle,H];
0175 |"vector 37
0176 |[ c, X, H, H, H, L, X, H, L, H, H] -> [idle,H];
0177 |[ c, X, L, L, L, L, X, H, L, H, H] -> [req_cycle,H];
0178 |[ c, X, L, L, L, L, X, H, L, H, H] -> [req_cycle,H];
0179 |[ c, X, L, L, L, X, X, H, H, H, H] -> [strt_cycl,H];
0180 |[ c, X, L, L, L, L, X, H, H, H, H] -> [do_cycle,H];
0181 |[ c, X, L, L, L, H, L, H, L, H, H] -> [fin_cycle,H];
0182 |[ c, H, L, L, L, H, L, H, X, H, H] -> [park,H];
0183 |[ L, H, L, L, L, H, L, H, X, H, H] -> [park,H];
0184 |[ c, H, L, L, L, L, H, H, L, L, H] -> [fin_cycle,H];
0185 |[ c, H, L, L, L, L, H, H, X, L, H] -> [park,H];
0186 |[ L, X, L, L, L, L, H, H, X, H, H] -> [park,H];
0187 |[ c, H, L, L, L, L, H, H, H, H, H] -> [do_cycle,H];
0188 |[ c, H, L, L, L, L, H, H, L, L, H] -> [fin_cycle,H];
0189 |[ c, H, L, L, L, H, L, H, X, L, H] -> [park,H];
0190 |[ L, H, L, L, L, H, L, H, X, H, H] -> [par,H];
0191 |[ c, H, H, H, H, L, H, H, X, H, H] -> [park,H];
0192 |[ c, H, H, H, H, L, H, H, X, H, H] -> [park,H];
0193 |[ c, L, L, X, X, H, L, L, H, H, H] -> [fin_cycle,H];
0194 |[ c, L, L, X, X, H, L, H, X, H, H] -> [park,H];
0195 |[ c, X, X, X, X, H, L, H, L, H, H] -> [idle,H];
0196 |"vector 62
0197 |[ c, X, H, H, H, L, X, H, L, H, H] -> [idle,H];
0198 |[ c, X, L, X, H, L, X, H, L, H, H] -> [req_cycle,H];
0199 |[ c, X, L, X, H, X, X, H, H, H, H] -> [strt_cycl,H];
0200 |[ c, X, L, X, H, L, X, H, H, L, H] -> [do_cycle,H];
0201 |[ c, H, L, X, H, H, L, H, L, L, H] -> [fin_cycle,H];
0202 |[ c, H, L, X, H, H, L, H, X, L, H] -> [park,H];
0203 |[ L, H, L, X, H, H, L, H, X, H, H] -> [park,H];
0204 |[ c, H, L, X, H, H, H, H, L, L, H] -> [fin_cycle,H];
0205 |[ c, H, L, X, H, H, H, H, X, L, H] -> [park,H];
0206 |[ L, H, L, X, H, H, H, H, X, H, H] -> [park,H];

```

Figure 13-19. PLD Equations for Programming the 16R4 PLD (First-Level Logic) (Concluded)

```

0207 |[ c, H, H, L, H, L, X, H, H, H, H] -> [do_cycle,L];
0208 |[ c, H, H, L, H, H, L, H, L, H, H] -> [fin_cycle,L];
0209 |[ c, H, H, L, H, H, L, H, X, H, H] -> [park,L];
0210 |[ c, X, L, L, L, L, H, H, H, H, H] -> [do_cycle,H];
0211 |[ c, H, L, L, L, L, H, H, L, L, H] -> [fin_cycle,H];
0212 |[ c, H, L, L, L, H, L, H, X, L, H] -> [park,H];
0213 |[ L, H, L, L, L, H, L, H, X, H, H] -> [park,H];
0214 |[ c, X, L, X, X, H, L, L, H, H, H] -> [fin_cycle,H];
0215 |[ c, L, L, X, X, H, L, H, X, H, H] -> [park,H];
0216 |[ c, H, H, H, H, X, X, H, H, H, H] -> [park,H];
0217 |[ c, H, H, H, H, H, H, H, H, H, H] -> [park,H];
0218 |[ c, X, X, X, X, H, L, H, L, H, H] -> [idle,H];
0219 |@page
0220 |[ c, X, H, X, L, X, X, X, L, H, H] -> [req_cycle,H];
0221 |[ c, X, H, X, L, X, X, X, H, H, H] -> [strt_cycl,L];
0222 |[ c, X, H, X, L, X, X, X, H, H, H] -> [do_cycle,L];
0223 |[ c, X, H, X, L, X, X, H, H, H, H] -> [fin_cycle,L];
0224 |[ c, L, H, X, L, X, X, H, H, H, H] -> [park,L];
0225 |[ c, X, H, X, X, H, L, X, L, H, H] -> [idle,L];
0226 |[ c, X, H, L, L, X, X, X, L, H, H] -> [req_cycle,H];
0227 |"([hl,stat3,stat2,stat1,stat0,lock_,priDMA, strb0_,bg, strb1,
|reset]->[outst,gwe_])
0228 |
0229 |
0230 |end    c40_local_glob_bus_interf
0231 |
0232 |
0233 |
0234 |
0235 |

```

The **six PLD states** are `idle`, `request_cycle`, `start_cycle`, `do_cycle`, `finish_cycle`, and `park`.

- 1) After reset, the first-level PLD's state machine starts in the `idle` state and transcends to the `request_cycle` state when a global bus transfer is required.
- 2) The transition to `request_cycle` occurs when any of the 'C40 status lines (STAT2–0) are low (when the status lines are all high, the bus is idle). In this state, the `BUSREQ` signal becomes active and is sent to the GBC PLD.
- 3) When the PLD receives a `BUSGRANT` signal, the state machine transitions to the `start_cycle` state. For the `start_cycle`, `do_cycle`, `finish_cycle` and `park` states, `BUSREQUEST` and `BUSENABLE` are active.
- 4) From the `start_cycle` state, the state machine transitions to the `do_cycle` state during the next H1 cycle.
- 5) From the `do_cycle` state, the state machine transitions to the `finish_cycle` state in the next H1 cycle. In this state, the `BUSRDY` signal is active. `BUSRDY` indicates to the 'C40 that the memory access has been completed and that another access can be started.
- 6) From the `finish_cycle` state, the state machine transitions to the `park` state during the next H1 cycle. `BUSRDY` goes inactive in anticipation of another bus cycle starting.

Bus parking is implemented for this bus arbitration protocol to allow the current bus master to retain control of the bus and continue making accesses to global memory as long as consecutive interlocked or priority DMA cycles are required or if no other processor is requesting use of the bus. Bus parking reduces memory access latency when only one 'C40 desires access to the global bus during any duration.

Notice that when the state machine leaves the `park` state, allowing the current bus master to perform another shared memory access, the state machine can transition to either the `finish_cycle` or `do_cycle` states, depending on the level of `STRB0` or `STRB1`. The `STRB` signal remaining low between accesses indicates back-to-back read cycles, which require only two H1 cycles to complete for 35-ns memories. Hence, the state machine transitions from the `park` state directly to `finish_cycle`. If the `STRB` signal goes high one H1 cycle and then back low between accesses, the state machine transitions from the `park` state to `do_cycle`, allowing the one cycle for the `STRB` high and two for the subsequent access.

The global bus controller (second-level logic) is implemented as a 16R8 PLD. This PLD takes as inputs the outputs of each of the four first-level PLDs. Hence, the GBC has four `BUSREQUEST` signals as inputs — one

from each of the four, first-level logic PLDs associated with each of the 'C40s. The GBC asserts four outputs, the  $\overline{\text{BUSGRANT}}$  signals associated with each 'C40's first-level arbitration logic.

Figure 13–21 illustrates graphically the state machine for the global bus controller. The GBC asserts low the  $\overline{\text{BUSGRANT}}$  signal associated with the 'C40 that wins an arbitration contest. This  $\overline{\text{BUSGRANT}}$  signal remains active until another processor desires access to the shared bus and a time-out signal has been received. The new contestant is not granted access to the shared bus until after the current bus master deasserts (brings high) its  $\overline{\text{BUSREQ}}$  signal, indicating it has finished its priority accesses or single nonpriority memory access. The system  $\overline{\text{RESET}}$  signal should also be an input to the GBC PLD. The system  $\overline{\text{RESET}}$  signal should clear (deassert high) all  $\overline{\text{BUSGRANT}}$  signals to any of the 'C40s and return the GBC state machine to an idle state.

The time-out signal is also a necessary input for the GBC because of the high speed of bus arbitration. Before taking a  $\overline{\text{BUSGRANT}}$  signal away, the GBC must guarantee that a bus arbitration winner has had a chance to see the  $\overline{\text{BUSGRANT}}$  and start using the bus. The timeout signal is generated by a counter implemented with a 16R4 PLD. The counter starts counting when a processor first receives a  $\overline{\text{BUSGRANT}}$  signal. It counts four cycles and then issues a time-out signal to the GBC indicating that the GBC can take away the current master's  $\overline{\text{BUSGRANT}}$  if necessary. Hence, the time-out counter provides at least four cycles for a 'C40's first level of logic to see a  $\overline{\text{BUSGRANT}}$  and start using the bus before the GBC can take the  $\overline{\text{BUSGRANT}}$  away. Figure 13–23 contains the ABEL PLD equations for the time-out counter.

The type of arbitration implemented in this GBC example is a **rotating** priority scheme. This rotating priority scheme provides fair arbitration among the four 'C40s sharing the global bus. In a rotating priority scheme, the last bus master becomes the lowest (last serviced) priority processor. The processors sequentially rotate throughout the priority list with the least recently serviced processor having the highest priority in subsequent arbitration contests. The priority rotates every time the bus request of the current bus master goes inactive and another processor desires access to shared memory. At system reset, the priorities are 1, 2, 3, or 4, with 1 being the highest or first serviced priority.

Figure 13–22 shows PLD equations for programming the 16R8 PLD used to implement the rotating priority global bus controller. ABEL's PLD language is used to describe the state machine shown in Figure 13–21.

**Note: Active-Low Indicators**

In listings (e.g., Figure 13–22), an underscore following a signal name (e.g., `busreq_`) indicates that the signal is active low. (In regular text, such signals are overbarred (e.g.,  $\overline{\text{BUSREQ}}$ ).

The PLD's four outputs are the four `busgrant_` lines, with each line giving a different 'C40 access to the shared bus. These four bits are also used as half of the output state bits. The other four state bits are used to indicate the ready state corresponding to each `busgrant` state. At reset, the GBC state machine goes to the `idle` state. All `busgrant_` signals are inactive. In the `idle` state, `br_` signals can be received from any of the four first-level PLDs. After arbitration, the state machine makes a transition to one of the `grx` states (where  $x = 1, 2, 3, \text{ or } 4$ ). The corresponding `busgrantx_` output signal goes active. The GBC stays in that state until a `busrequest_` and a time-out signal is received from another processors' first-level PLD. Once another `busrequest_` is received, the state machine transcends to the corresponding `bryx` state. In this state, the `busgrantx_` signal goes inactive. However, the GBC state machine stays in this state until the corresponding bus request (`brx_`) input goes inactive high, indicating that the current bus master has relinquished control of the shared bus. When `brx_` goes inactive, the state machine changes to the highest priority processor's `gry` state (where  $y = 1, 2, 3, \text{ or } 4$ ) that had its `br_` signal active.

Figure 13-20. PLD Equations for Programming the 16R4 PLD

```

0001 |module      c40_global_bus_interface
0002 |title'
0003 |
0004 |DWG
0005 |DWG #
0006 |
0007 |COMPANY      TEXAS INSTRUMENTS INCROPORATED'
0008 |
0009 |c40u1 device 'P16R4';
0010 |
0011 |    "inputs
0012 |        h3                Pin 1;
0013 |        bg_                Pin 7;
0014 |        busrdy_           Pin 8;    "busrdy from global interface PAL
0015 |        busenable_       Pin 9;    "busenable from global interface
                                PAL
0016 |
0017 |    "outputs
0018 |        ctrl_enable_     Pin 18;    "enable signal for control lines
0019 |        rdy_             Pin 17;    "rdy signal for shared SRAM
0020 |        sync_ae_         Pin 15;    "synchronized busenable signal
0021 |        bg_sync_         Pin 14;
0022 |
0023 |
0024 | "name substitutions
0025 |         CE_             = ctrl_enable_;
0026 |         bry_            = rdy_;
0027 |
0028 | "substitutions for test vectors
0029 |         c,H,L,X,       = .C.1,0,.X.;
0030 |
0031 | equations
0032 |         sync_ae_ :     = busenable_;
0033 |         !ctrl_enable_ = !sync_ae_ & !busenable_;
0034 |         rdy_ :        = busrdy_;
0035 |         bg_sync_ :    = bg_;
0036 |
0037 |@page
0038 |
0039 |
0040 |
0041 |
0042 | "Test 1st level global arbitration logic
0043 | test_vectors

```

Figure 13–20. PLD Equations for Programming the 16R4 PLD (Concluded)

```

0044 | ([h3,busenable_,busrdy_,bg_]->[CE_,bry_,bg_sync_])
0045 | [ c, H, H, H] -> [ H, H, H];
0046 | [ c, L, H, H] -> [ L, H, H];
0047 | [ c, L, H, L] -> [ L, H, L];
0048 | [ L, L, L, H] -> [ L, H, L];
0049 | [ c, L, L, H] -> [ L, L, H];
0050 | [ c, H, H, H] -> [ H, H, H];
0051 | [ c, H, H, H] -> [ H, H, H];
0052 | [ c, H, H, H] -> [ H, H, H];
0053 | [ c, H, H, H] -> [ H, H, H];
0054 | [ L, L, H, L] -> [ H, H, H];
0055 | [ c, L, H, H] -> [ L, H, H];
0056 | [ c, L, H, H] -> [ L, H, H];
0057 | [ c, L, L, H] -> [ L, L, H];
0058 | [ c, L, H, L] -> [ L, H, L];
0059 | [ c, L, L, L] -> [ L, L, L];
0060 | [ L, H, H, H] -> [ H, L, L];
0061 | [ c, H, H, H] -> [ H, H, H];
0062 | [ c, H, H, H] -> [ H, H, H];
0063 | [ c, H, H, L] -> [ H, H, L];
0064 |@page
0065 | [ c, H, H, L] -> [ H, H, L];
0066 | [ c, H, H, L] -> [ H, H, L];
0067 | [ c, H, H, L] -> [ H, H, L];
0068 | [ c, H, H, L] -> [ H, H, L];
0069 | [ c, L, H, H] -> [ L, H, H];
0070 | [ c, L, H, H] -> [ L, H, H];
0071 | [ L, L, L, H] -> [ L, H, H];
0072 | [ c, L, L, H] -> [ L, L, H];
0073 | [ c, H, H, H] -> [ H, H, H];
0074 | [ c, H, H, H] -> [ H, H, H];
0075 | [ c, H, H, H] -> [ H, H, H];
0076 | [ c, H, H, H] -> [ H, H, H];
0077 | [ L, L, H, H] -> [ H, H, H];
0078 | [ c, L, H, H] -> [ L, H, H];
0079 | [ c, L, H, H] -> [ L, H, H];
0080 | [ c, L, L, H] -> [ L, L, H];
0081 | [ c, L, H, H] -> [ L, H, H];
0082 | [ c, L, L, H] -> [ L, L, H];
0083 |
0084 |end      c40_global_bus_interface

```

Figure 13–21. Global Bus Controllor PLD (Rotating Priority Mode Only)

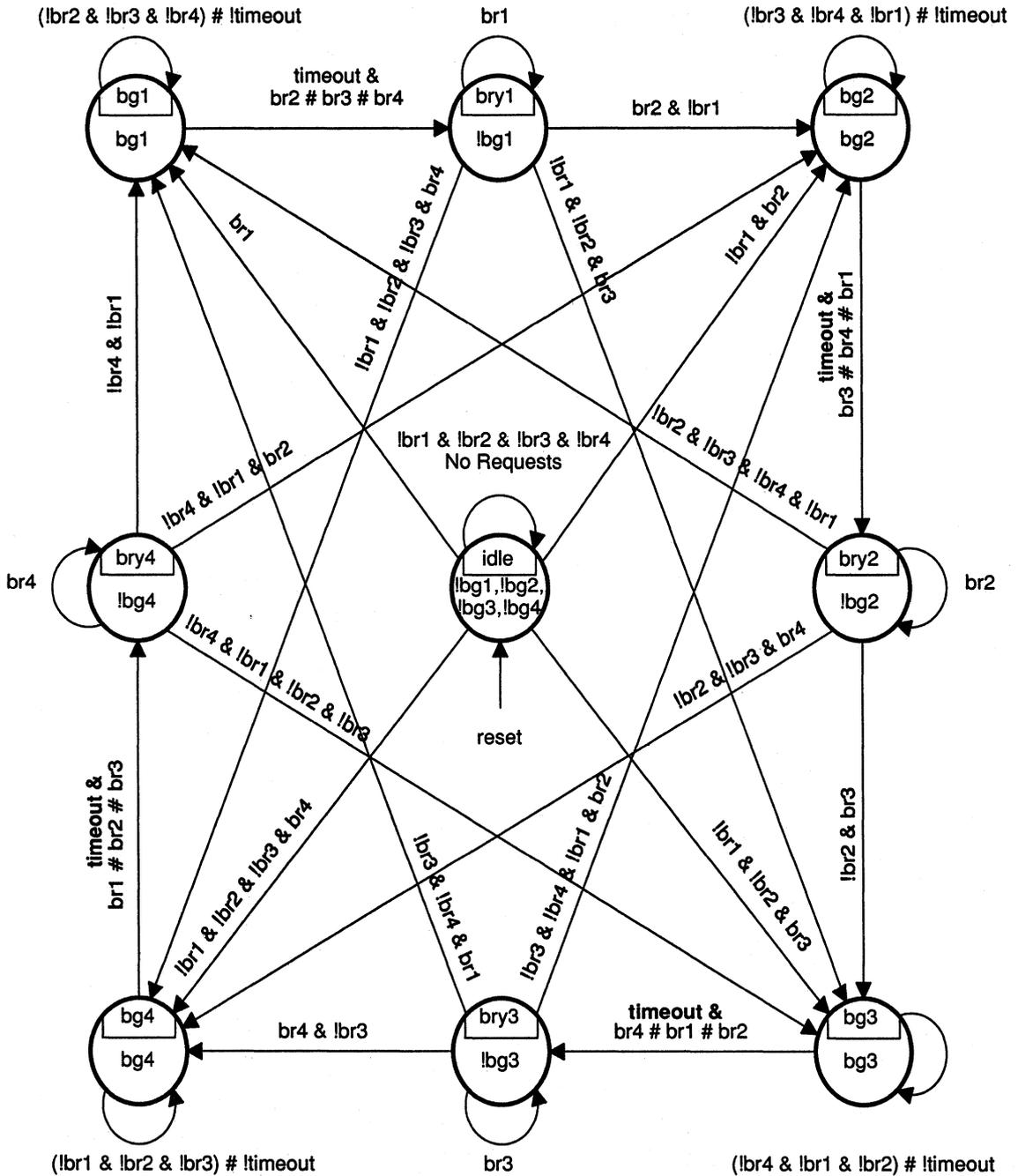


Figure 13–22. PLD Equations for Programming the 16R8 PLD

```

0001 |module global_bus_cntrl
0002 |title'
0003 |
0004 |DWG NAME Shared bus interface
0005 |DWG #
0006 |
0007 |COMPANY TEXAS INSTRUMENTS INCORPORATED'
0008 |
0009 |     xub5     device   'P16R8';
0010 |
0011 |     h50      Pin 1;   "50 MHz clock
0012 |     br1_     Pin 2;   "bus request 1
0013 |     br2_     Pin 3;   "bus request 2
0014 |     br3_     Pin 4;   "bus request 3
0015 |     br4_     Pin 5;   "bus request 4
0016 |     reset_   Pin 6;   "reset
0017 |     fix_rot_ Pin 7;   "fix/rot_ not used here but can be added
0018 |     oe       Pin 11;
0019 |     timeout_ Pin 8;
0020 |     vss      Pin 10;
0021 |
0022 |     bg1_     Pin 19;  "grant 1
0023 |     bg2_     Pin 18;  "grant 2
0024 |     bg3_     Pin 17;  "grant 3
0025 |     bg4_     Pin 16;  "grant 4
0026 |     s3       Pin 15;  "state 3
0027 |     s2       Pin 14;  "state 2
0028 |     s1       Pin 13;  "state 1
0029 |     s0       Pin 12;  "state 0
0030 |     vcc      Pin 20;
0031 |
0032 |     c,H,L,X  = .C.,1,0,.X.;
0033 |
0034 |     "define state machine bits
0035 |     bus_state = [s3,s2,s1,s0,bg4_,bg3_,bg2_,bg1_]
0036 |
0037 |     "states
0038 |     bry1     = ^b01111111; "ready 1
0039 |     bry2     = ^b10111111; "ready 2
0040 |     bry3     = ^b11011111; "ready 3
0041 |     bry4     = ^b11101111; "ready 4
0042 |     idle     = ^b11111111; "idle state
0043 |
0044 |     gr1      = ^b11111110; "grant 1
0045 |     gr2      = ^b11111101; "grant 2
0046 |     gr3      = ^b11111011; "grant 3
0047 |     gr4      = ^b11110111; "grant 4
0048 |
0049 |     "convert inputs to positive logic
0050 |     br1      = !br1_;
0051 |     br2      = !br2_;
0052 |     br3      = !br3_;
0053 |     br4      = !br4_;
0054 |     reset    = !reset_;
0055 |@page

```

Figure 13-22. PLD Equations for Programming the 16R8 PLD (Continued)

```

0056 |
0057 | state_diagrambus_state
0058 |     state     idle:
0059 |         if ( reset ) then idle
0060 |         else if ( !reset & !br1 & !br2 & br3 & br4)then gr4
0061 |         else if ( !reset & !br1 & !br2 & br3 ) then gr3
0062 |         else if ( !reset & !br1 & br2 ) then gr2
0063 |         else if ( !reset & br1 ) then gr1
0064 |         else idle;
0065 |
0066 |     state     bry4:
0067 |         if ( reset ) then idle
0068 |         else if ( !reset & br4 ) then bry4
0069 |         else if ( !reset & !br1 & !br2 & !br3 & !br4)then idle
0070 |         else if ( !reset & !br1 & !br2 & br3 & !br4) then gr3
0071 |         else if ( !reset & !br1 & br2 & !br4) then gr2
0072 |         else if ( !reset & br1 & !br4) then gr1;
0073 |
0074 |     state     bry3:
0075 |         if ( reset ) then idle
0076 |         else if ( !reset & br3 ) then bry3
0077 |         else if ( !reset & !br4 & !br1 & br2 & !br3) then idle
0078 |         else if ( !reset & !br4 & !br1 & br2 & !br3) then gr2
0079 |         else if ( !reset & !br4 & br1 & !br3) then gr1
0080 |         else if ( !reset & br4 & !br3) then gr4;
0081 |
0082 |     state     bry2:
0083 |         if ( reset ) then idle
0084 |         else if ( !reset & br2 ) then bry2
0085 |         else if ( !reset & !br3 & !br4 & !br1 & !br2) then idle
0086 |         else if ( !reset & !br3 & !br4 & br1 & !br2) then gr1
0087 |         else if ( !reset & !br3 & br4 & !br2) then gr4
0088 |         else if ( !reset & br3 & !br2) then gr3;
0089 |
0090 |     state     bry1:
0091 |         if ( reset ) then idle
0092 |         if ( !reset & br1 ) then bry1
0093 |         else if ( !reset & !br2 & !br3 & !br4 & !br1) then idle
0094 |         else if ( !reset & !br2 & !br3 & br4 & !br1) then gr4
0095 |         else if ( !reset & !br2 & br3 & !br1) then gr3
0096 |         else if ( !reset & br2 & !br1) then gr2;
0097 |
0098 |     state     gr4:
0099 |         if ( !reset & (timeout # !br1 & !br2 & !br3)) then gr4
0100 |         else if ( reset ) then idle
0101 |
0102 |
0103 |     state     gr3:
0104 |         if ( !reset & (timeout # !br4 & !br1 & !br2)) then gr3
0105 |         else if ( reset ) then idle
0106 |
0107 |
0108 |     state     gr2:

```

Figure 13-22. PLD Equations for Programming the 16R8 PLD (Continued)

```

0109 |           if (!reset & (timeout # !br3 & !br4 & !br1 )) then gr2
0110 |           else if ( reset ) then idle
0111 |
0112 |
0113 |           state      gr1:
0114 |           if ( !reset & (timeout # !br2 & !br3 & !br4 ) then gr1
0115 |           else if ( reset ) then idle
0116 |
0117 |@page
0115 |test_vectors
0116 |
0117 |"rotating priority vectors
0118 |([h50,br1,br2,br3,br4,timeout_,reset_] -> [bus_state])
0119 |"check for go to IDLE
0120 |[ c, X, X, X, X, X, L ] -> [idle];
0121 |[ c, L, L, L, H, X, H ] -> [gr4];
0122 |[ c, X, X, X, X, X, L ] -> [idle];
0123 |[ c, L, L, H, X, X, H ] -> [gr3];
0124 |[ c, X, X, X, X, X, L ] -> [idle];
0125 |[ c, L, H, X, X, X, H ] -> [gr2];
0126 |[ c, X, X, X, X, X, L ] -> [idle];
0127 |[ c, H, X, X, X, X, H ] -> [gr1];
0128 |[ c, X, X, X, X, X, L ] -> [idle];
0129 |[ c, L, L, L, H, X, H ] -> [gr4];
0130 |[ c, H, L, L, H, L, H ] -> [bry4];
0131 |[ c, X, X, X, X, X, L ] -> [idle];
0132 |[ c, L, L, H, X, X, H ] -> [gr3];
0133 |[ c, L, H, H, L, L, H ] -> [bry3];
0134 |[ c, X, X, X, X, X, L ] -> [idle];
0135 |[ c, L, H, X, X, X, H ] -> [gr2];
0136 |[ c, L, H, H, H, L, H ] -> [bry2];
0137 |[ c, X, X, X, X, X, L ] -> [idle];
0138 |[ c, H, X, X, X, X, H ] -> [gr1];
0139 |[ c, H, H, H, H, L, H ] -> [bry1];
0140 |[ c, X, X, X, X, X, L ] -> [idle];
0141 |
0142 |
0143 |[ c, X, X, X, L, X, H ] -> [idle];
0144 |[ c, L, L, H, X, X, H ] -> [gr3];
0145 |[ c, X, X, L, X, X, H ] -> [idle];
0146 |[ c, L, H, X, X, X, H ] -> [gr2];
0147 |[ c, X, L, X, X, X, H ] -> [idle];
0148 |[ c, H, X, X, X, X, H ] -> [gr1];
0149 |[ c, L, X, X, X, X, H ] -> [idle];
0150 |[ c, L, L, L, H, X, H ] -> [gr4];
0151 |[ c, H, L, L, H, L, H ] -> [bry4];

```

Figure 13-22. PLD Equations for Programming the 16R8 PLD (Continued)

```

0152 |[ c, L, L, L, L, X, H ] -> [idle];
0153 |[ c, L, L, H, X, X, H ] -> [gr3];
0154 |[ c, L, H, H, L, L, H ] -> [bry3];
0155 |[ c, L, L, L, L, X, H ] -> [idle];
0156 |[ c, L, H, X, X, X, H ] -> [gr2];
0157 |[ c, L, H, H, H, L, H ] -> [bry2];
0158 |[ c, L, L, L, L, X, H ] -> [idle];
0159 |[ c, H, X, X, X, X, H ] -> [gr1];
0160 |[ c, H, H, H, H, L, H ] -> [bry1];
0161 |[c, L, L, L, L, X, H, H ] -> [idle];
0162 |"vector 7
0163 |[ c, H, X, X, X, X, H ] -> [gr1];
0164 |[ c, H, X, X, X, H, H ] -> [gr1];
0165 |[ c, H, L, L, L, L, H ] -> [gr1];
0166 |[ c, H, X, X, H, L, H ] -> [bry1];
0167 |[ c, H, X, X, X, X, H ] -> [bry1];
0168 |[ c, H, X, X, X, X, H ] -> [bry1];
0169 |@page
0170 |"vector 15
0171 |[ c, L, L, L, H, X, H ] -> [gr4];
0172 |[ c, L, L, L, H, X, H ] -> [gr4];
0173 |[ c, L, L, L, H, X, H ] -> [gr4];
0174 |[ c, X, X, X, H, H, H ] -> [gr4];
0175 |[ c, X, X, H, X, L, H ] -> [bry4];
0176 |[ c, X, X, X, H, X, H ] -> [bry4];
0177 |[ c, X, X, X, H, X, H ] -> [bry4];
0178 |"vector 21
0179 |[ c, L, L, H, L, X, H ] -> [gr3];
0180 |[ c, L, L, H, L, X, H ] -> [gr3];
0181 |[ c, L, L, H, L, H, H ] -> [gr3];
0182 |[ c, X, X, H, X, H, H ] -> [gr3];
0183 |[ c, X, H, H, X, L, H ] -> [bry3];
0184 |[ c, X, X, H, X, X, H ] -> [bry3];
0185 |[ c, X, X, H, X, X, H ] -> [bry3];
0186 |"vector 27
0187 |[ c, L, H, L, L, X, H ] -> [gr2];
0188 |[ c, L, H, L, L, X, W ] -> [gr2];
0189 |[ c, X, H, X, X, H, H ] -> [gr2];
0190 |[ c, L, H, L, L, L, H ] -> [gr2];
0191 |[ c, H, H, X, X, L, H ] -> [bry2];
0192 |[ c, X, H, X, X, X, H ] -> [bry2];
0193 |[ c, X, H, X, X, X, H ] -> [bry2];
0194 |"vector 33
0195 |[ c, H, L, L, L, X, H ] -> [gr1];

```

Figure 13-22. PLD Equations for Programming the 16R8 PLD (Concluded)

```
0196 |[ c, H, X, H, X, L, H ] -> [bry1];
0197 |[ c, L, L, H, X, X, H ] -> [gr3];
0198 |[ c, H, X, H, X, L, H ] -> [bry3];
0199 |[ c, H, X, L, L, X, H ] -> [gr1];
0200 |[ c, H, H, X, X, L, H ] -> [bry1];
0201 |[ c, L, H, X, X, X, H ] -> [gr2];
0202 |[ c, X, H, X, H, L, H ] -> [bry2];
0203 |[ c, X, L, L, H, X, H ] -> [gr4];
0204 |[ c, X, H, X, H, L, H ] -> [bry4];
0205 |[ c, L, H, X, L, X, H ] -> [gr2];
0206 |[ c, X, H, H, X, L, H ] -> [bry2];
0207 |"vector 45
0208 |[ c, X, L, H, X, X, H ] -> [gr3];
0209 |[ c, X, X, H, H, L, H ] -> [bry3];
0210 |[ c, X, X, L, H, X, H ] -> [gr4];
0211 |[ c, H, X, X, H, L, H ] -> [bry4];
0212 |[ c, H, X, X, L, X, H ] -> [gr1];
0213 |[ c, H, H, X, X, L, H ] -> [bry1];
0214 |end    global_bus_cntrl
```

Figure 13-23. PLD Equations for Programming the 16R6 PLD

```

0001 |module      c40_global_timeout
0002 |title'
0003 |DWG NAME    global arbitration
0004 |DWG #
0005 |COMPANY     TEXAS INSTRUMENTS INCROPORATED
0006 |
0007 |DATE
0008 |
0009 |    c40u4      device                'P16R6';
0010 |
0011 |    "inputs
0012 |    h50        Pin 1;
0013 |    bg1_       Pin 2;
0014 |    bg2_       Pin 3;
0015 |    bg3_       Pin 4;
0016 |    bg4_       Pin 5;
0017 |    timeout_   Pin 13;    "output
0018 |    s1         Pin 16;
0019 |    s0         Pin 15;
0020 |
0021 |
0022 | "name substitution to increase readability
0023 |    bus_active = (!bg1_ # !bg2_ # !bg3_ # !bg4_);
0024 |
0025 | "define machine state bits
0026 | "[timeout_,s1,s0];
0027 |
0028 |    "states
0029 |
0030 |    idle       = ^b111;
0031 |    count1     = ^b110;
0032 |    count2     = ^b101;
0033 |    count3     = ^b100;
0034 |    time       = ^b011;
0035 |
0036 |    outstate   = [timeout_,s1,s0];
0037 |
0038 |    c,H,L,X    = .C.,1,0,.X.;
0039 |
0040 |
0041 |state_diagram outstate
0042 |
0043 |state idle:
0044 |if (!bus active) then idle

```

Figure 13-23. PLD Equations for Programming the 16R6 PLD (Continued)

```

0045 |     else count1;
0046 |
0047 |state count1:
0048 |     if (!bus_active) then idle
0049 |     else count2;
0050 |
0051 |state count2:
0052 |     if (!bus_active) then idle
0053 |     else count3;
0054 |
0055 |state count3:
0056 |     if (!bus_active) then idle
0057 |     else time;
0058 |
0059 |state time:  GOTO idle;
0060 |
0061 |@page
0062 |"Test counter
0063 |test_vectors
0064 |[ ([h50, bg1_,  bg2_,  bg3_,  bg4_] -> [outstate])
0065 |[  c,  H,      H,      H,      H ] -> [ idle];
0066 |[  c,  L,      H,      H,      H ] -> [count1];
0067 |[  c,  H,      H,      H,      H ] -> [ idle];
0068 |[  c,  L,      H,      H,      H ] -> [count1];
0069 |[  c,  X,      X,      X,      X ] -> [count2];
0070 |[  c,  H,      H,      H,      H ] -> [ idle];
0071 |[  c,  L,      H,      H,      H ] -> [count1];
0072 |[  c,  X,      X,      X,      X ] -> [count2];
0073 |[  c,  X,      X,      X,      X ] -> [count3];
0074 |[  c,  H,      H,      H,      H ] -> [ idle];
0075 |[  c,  L,      H,      H,      H ] -> [count1];
0076 |[  c,  X,      X,      X,      X ] -> [count2];
0077 |[  c,  X,      X,      X,      X ] -> [count3];
0078 |
0079 |[  c,  X,      X,      X,      X ] -> [ idle];
0080 |[  c,  H,      L,      H,      H ] -> [count1];
0081 |[  c,  X,      X,      X,      X ] -> [count2];
0082 |[  c,  X,      X,      X,      X ] -> [count3];
0083 |[  c,  X,      X,      X,      X ] -> [time];
0084 |[  c,  X,      X,      X,      X ] -> [ idle];
0085 |[  c,  H,      H,      L,      H ] -> [count1];
0086 |[  c,  X,      X,      X,      X ] -> [count2];
0087 |[  c,  X,      X,      X,      X ] -> [count3];
0088 |[  c,  X,      X,      X,      X ] -> [time];

```

Figure 13–23. PLD Equations for Programming the 16R6 PLD (Concluded)

```

0089 |[ c, X, X, X, X ] -> [ idle];
0090 |[ c, H, H, H, L ] -> [count1];
0091 |[ c, X, X, X, X ] -> [count2];
0092 |[ c, X, X, X, X ] -> [count3];
0093 |[ c, X, X, X, X ] -> [time];
0094 |[ c, X, X, X, X ] -> [ idle];
0095 |
0096 |
0097 |
0098 |
0099 |
0100 |end    c40_global_timeout
0101 |
0102 |
0103 |
0104 |
0105 |

```

### 13.6.2 Arbitration Alternatives

If more arbitration flexibility is desired, a fixed priority mode can be implemented in the global bus controller PLD. A fixed scheme can be used in conjunction with this rotating priority mode if a fixed/rotating input is added to the GBC PLD to allow either of the two arbitration methods. One of the spare IIOF pins can be configured as a general-purpose output pin to act as the arbitration mode control pin. For example, if FIX/ROT (IIOF2) = 0, the four 'C40s have rotating priorities; if FIX/ROT = 1, the four processors have fixed priorities. To reduce state machine complexity, the rotating priorities can be preset at system reset to the same values as in the fixed arbitration mode, with the processors having priorities of 1, 2, 3, or 4, with 1 being the highest (first serviced) priority.

### 13.6.3 Global Bus Arbitration and Transfer Timing

To illustrate the timing involved with global bus arbitration and data transfers, Figure 13–17 (page 13-47), Figure 13–24 (page 13-72), Figure 13–25 and Figure 13–26, show shared bus timings using the rotating priority arbitration configuration.

These figures represent a 'C40 requesting a shared bus access when it is not currently the bus master. Clock H1 is the output clock of the the 'C40 requesting access to the bus. Both clocks H1 and H3 have a rate of 25 MHz; however, the global bus controller (GBC) input clock is asynchronous with respect to H1 and H3 and has a rate of 50 MHz.

Due to the arbitration logic synchronizer delays and the 35-ns SRAMs, access to the shared memory requires wait states. A new bus master's first memory access after an arbitration win takes at least five H1 cycles (the five cycles include the time period from status lines active to the end of the read or write cycle), but subsequent reads or writes take only two H1 cycles. Two-cycle memory accesses allow enough time for control signals to go active and inactive to complete read or write cycles for 35-ns memories. They also allow processors to stop driving the bus before another processor starts driving the bus after a bus arbitration contest. Also, the two-cycle memory accesses allow enough time for signal buffering between the processor bus and memory (buffer delays are less than 15 ns with commercially available parts).

In Figure 13–17 (page 13-47), a 'C40 wins an arbitration contest immediately and does one read cycle. However, it loses arbitration for the next transfer on the shared bus (`busgrant_` goes inactive high) and the first-level PLD brings its `busrequest_` signal inactive high to signal the GBC that it has given up the bus. The first-level PLD at the same time sends bus disable signals ( $\overline{\text{BUSENABLE}}$  and  $\overline{\text{CTLENABLE}}$  high) to the  $\overline{\text{AE}}$ ,  $\overline{\text{DE}}$ , and  $\overline{\text{CE}}$  pins of the 'C40 to three-state the bus. The first-level PLD three-states the bus immediately because the GBC will give another processor access to the shared bus as soon as it sees this `BUSREQUEST` and a time-out go inactive.

Figure 13–24 shows a successful arbitration contest followed by successive reads. The 'C40 is allowed to do successive reads on the shared bus because no other processor desires access (`busgrant` stays active).

Figure 13–25 illustrates an arbitration win followed by a single write. Figure 13–26 shows an arbitration win followed by successive writes and an arbitration loss. The second write is allowed to occur because the `busgrant` going inactive is missed by the first-level PLDs, which synchronizes on H1 rising. The first-level PLD transcends to the `do_cycle` state because  $\overline{\text{STRB}}$  is high and the PLD has not seen the `busgrant` go inactive from the synchronizer output. Even though the first-level PLD sees that the `busgrant_` is taken away during the next H1/H3 cycle, it does not take away its `busrequest_` until the end of the second write cycle. Then, the `busrequest_` is made inactive, and the bus is disabled.

Figure 13-24. Successful TMS320C40 Arbitration; Data Read; Data Read

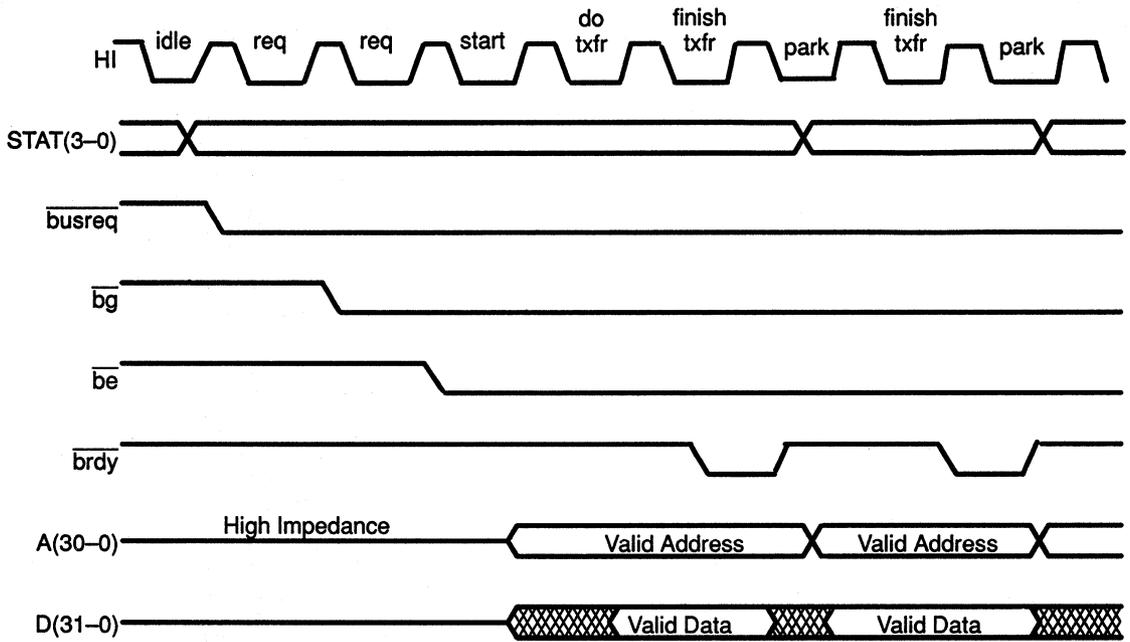


Figure 13-25. Successful TMS320C40 Arbitration and Data Write From Shared Bus Memory Followed by an Unsuccessful Arbitration Contest

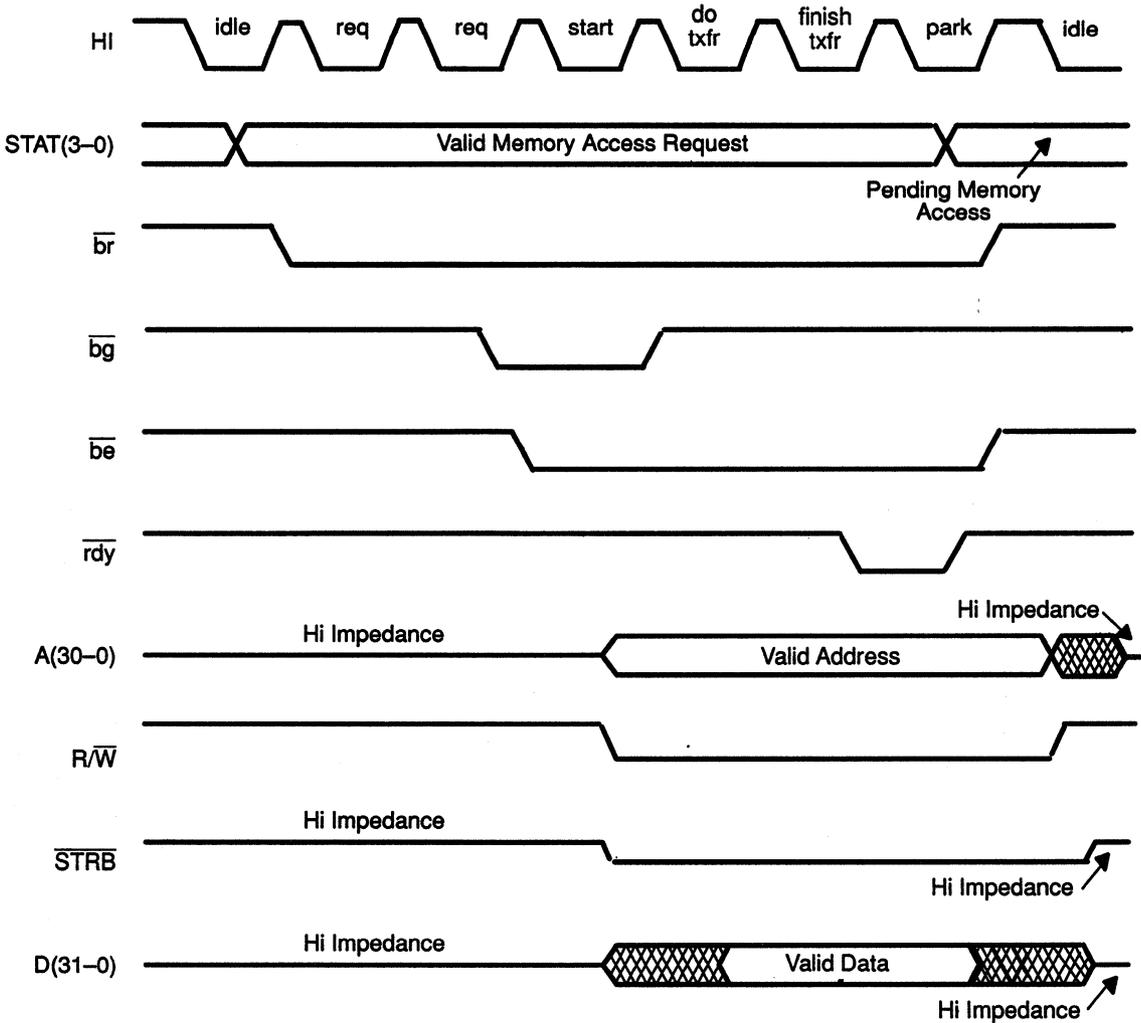
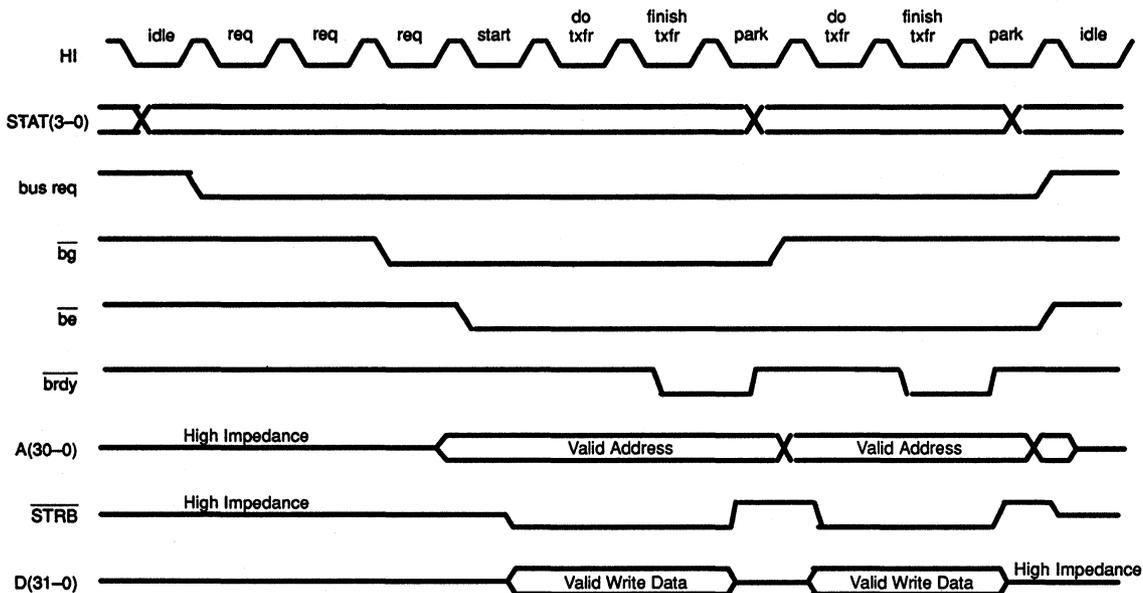


Figure 13–26. Successful 'C40 Arbitration; Consecutive Data Writes; Arbitration Win Followed by Successive Writes and an Arbitration Loss



### 13.6.4 Arbitration Protocol Limitations

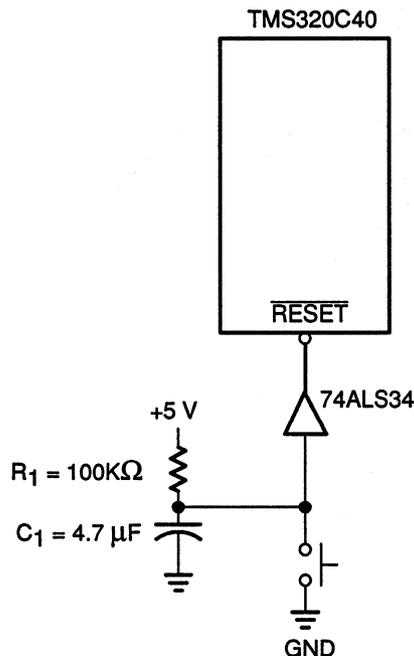
This shared bus arbitration protocol uses handshaking between the GBC and the processors sharing the global bus to ensure that only one processor is driving the bus at any given time. Nonetheless, the global bus controller should not allow another processor to become bus master until the previous master is guaranteed to release the bus completely. Since 'C40s have a bus disable ( $\overline{AE}$ ,  $\overline{DE}$ , or  $\overline{CE}$ ) time of less than 15 ns, bus turnoff time is not critical unless the GBC input clock frequency is greater than 50 MHz. However, if processors with slower turnoff times are used in a shared bus configuration with this protocol, the GBC input clock period cannot be less than the bus disable time of the slowest processor in the system. If the GBC input clock period is less than a processor's disable time, the GBC could give a new master ownership of the bus before the previous master is off the bus.

## 13.7 Reset Signal Generation Control Function

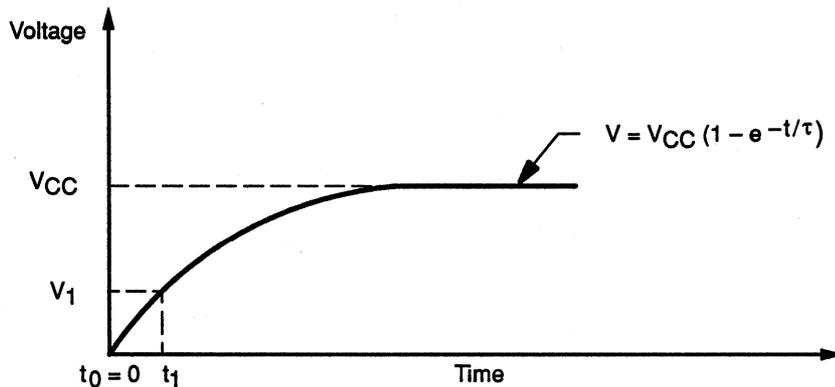
Several aspects of 'C40 system hardware design are critical to overall system operation. One such function is reset signal generation.

The reset input controls initialization of internal 'C40 logic and also causes execution of the system initialization software. For proper system initialization, the reset signal must be applied for at least ten H1 cycles, i.e., 400 ns for a 'C40 operating at 50.00 MHz. Upon powerup, however, it can take 20 ms or more before the system oscillator reaches a stable operating state. Therefore, the powerup reset circuit should generate a low pulse on the reset line for 100 to 200 ms. Once a proper reset pulse has been applied, the processor fetches the reset vector from location zero, which contains the address of the system initialization routine. Figure 13–27 shows a circuit that will generate an appropriate powerup or push button reset signal.

Figure 13–27. Reset Circuit



The voltage on the reset pin ( $\overline{\text{RESET}}$ ) is controlled by the  $R_1C_1$  network. After a reset, this voltage rises exponentially according to the time constant  $R_1C_1$ , as shown in Figure 13–28. In Figure 13–27, the 74ALS34 is used to provide a clean  $\overline{\text{RESET}}$  signal to the 'C40.

Figure 13–28. Voltage on the TMS320C40  $\overline{\text{RESET}}$  Pin

The duration of the low pulse on the  $\overline{\text{RESET}}$  pin is approximately  $t_1$ , which is the time it takes for the capacitor  $C_1$  to be charged to 1.5 V. This is approximately the voltage at which the reset input switches from a logic 0 to a logic 1. The capacitor voltage is expressed as

$$V = V_{CC} \left[ 1 - e^{-t/\tau} \right] \quad (5)$$

where  $\tau = R_1 C_1$  is the reset circuit time constant. Solving (5) for  $t$  results in

$$t = -R_1 C_1 \ln \left[ 1 - \frac{V}{V_{CC}} \right] \quad (6)$$

Setting the following:

$$R_1 = 100 \text{ k}\Omega$$

$$C_1 = 4.7 \text{ }\mu\text{F}$$

$$V_{CC} = 5 \text{ V}$$

$$V = V_1 = 1.5 \text{ V}$$

results in  $t = 167 \text{ ms}$ . Therefore, the reset circuit of Figure 13–27 provides a low pulse long enough to ensure the stabilization of the system oscillator upon powerup.

Note that if synchronization of multiple 'C40s is required, all processors should be provided with the same input clock and the same reset signal. After powerup, when the clock has stabilized, all processors may then be synchronized by generating a falling edge on the common reset signal. Since it is the falling edge of  $\overline{\text{RESET}}$  that establishes synchronization,  $\overline{\text{RESET}}$  must be high for at least ten H1 cycles initially. Following the falling edge,  $\overline{\text{RESET}}$  should remain low for at least ten H1 cycles and then be driven high. This sequencing of  $\overline{\text{RESET}}$  may be accomplished by using additional circuitry based on either RC time delays or counters.



# TMS320C4x Signal Descriptions and Electrical Characteristics

---



---



---

The sections in this chapter cover the following characteristics of the TMS320C4x:

Section	Page
14.1 Pinout and Pin Assignments .....	14-2
14.2 Signal Descriptions .....	14-7
14.3 TMS320C4x Mechanical Data .....	14-11
14.4 Electrical Specifications .....	14-12
14.5 Signal Transition Levels .....	14-14
14.6 Timing .....	14-15

---

**Note: Advance Information**

Unless otherwise noted, this chapter contains advance information on new products in the sampling or preproduction phases of development. Characteristic data and other specifications are subject to change without notice.

---

### 14.1 Pinout and Pin Assignments

The TMS320C40 (TMS320C4x generation) digital signal processor is available in a 325-pin grid array (PGA) package. The pinout of this package is shown in Figure 14–1. Pin assignments are listed in the following tables:

- ❑ Table 14–1: Pins sorted by signal name (alphanumeric listing)
- ❑ Table 14–2: Pins sorted by pin number (location on Figure 14–1)
- ❑ Table 14–3: Pins sorted by function, describing each (page 14-7)

Figure 14–1. TMS320C40 Pinout (Bottom View)

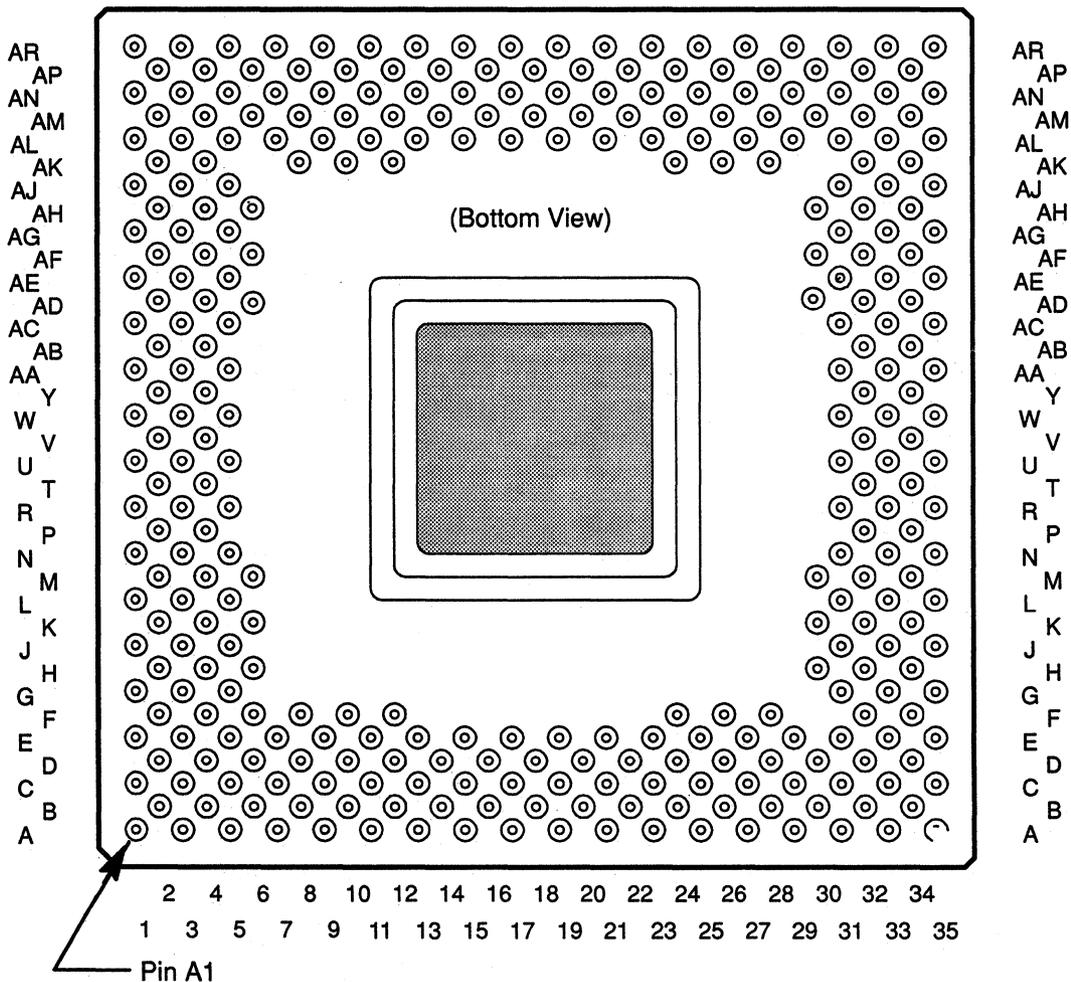


Table 14–1. TMS320C40 Pin Assignments Sorted by Signal Name

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
A0	D32	C0D6	AN7	C5D4	AM30	CVSS	E35	D31	F32
A1	B32	C0D7	AK8	C5D5	AP32	CVSS	AR25	$\overline{DE}$	AA31
A2	D30	C1D0	AL7	C5D6	AM32	CVSS	AE1	DVDD	AR11
A3	C29	C1D1	AP8	C5D7	AL31	CVSS	AR13	DVDD	AR29
A4	B30	C1D2	AM8	$\overline{CACK0}$	AN11	CVSS	A19	DVDD	A13
A5	F28	C1D3	AK12	$\overline{CACK1}$	AN13	CVSS	R35	DVDD	A7
A6	F24	C1D4	AK10	$\overline{CACK2}$	AM14	CVSS	AL1	DVDD	A17
A7	E29	C1D5	AN9	$\overline{CACK3}$	AM16	D0	U33	DVDD	L35
A8	C27	C1D6	AL9	$\overline{CACK4}$	AK32	D1	V32	DVDD	AR23
A9	D28	C1D7	AP10	$\overline{CACK5}$	AJ31	D2	T34	DVDD	A29
A10	B28	C2D0	AM18	$\overline{CE0}$	AA33	D3	U31	DVDD	L1
A11	F26	C2D1	AN19	$\overline{CE1}$	V34	D4	R33	DVDD	AC1
A12	C25	C2D2	AL19	$\overline{CRDY0}$	AP12	D5	P34	DVDD	AR17
A13	E27	C2D3	AP20	$\overline{CRDY1}$	AP14	D6	T32	DVDD	A23
A14	B26	C2D4	AM20	$\overline{CRDY2}$	AL15	D7	N33	DVDD	AJ1
A15	D26	C2D5	AN21	$\overline{CRDY3}$	AL17	D8	R31	DVSS	AJ35
A16	C23	C2D6	AL21	$\overline{CRDY4}$	AH30	D9	M34	DVSS	A21
A17	B24	C2D7	AP22	$\overline{CRDY5}$	AH32	D10	P32	DVSS	A25
A18	E25	C3D0	AM22	$\overline{CREQ0}$	AM10	D11	L33	DVSS	G35
A19	C21	C3D1	AN23	$\overline{CREQ1}$	AM12	D12	N31	DVSS	A11
A20	D24	C3D2	AL23	$\overline{CREQ2}$	AN15	D13	K34	DVSS	AG1
A21	B22	C3D3	AP24	$\overline{CREQ3}$	AN17	D14	M32	DVSS	AM2
A22	E23	C3D4	AM24	$\overline{CREQ4}$	AN33	D15	J33	DVSS	R1
A23	C19	C3D5	AN25	$\overline{CREQ5}$	AL33	D16	L31	DVSS	AR21
A24	D22	C3D6	AL25	$\overline{CSTRB0}$	AL11	D17	M30	DVSS	AR15
A25	B20	C3D7	AP26	$\overline{CSTRB1}$	AL13	D18	K32	DVSS	A15
A26	E21	C4D0	AN27	$\overline{CSTRB2}$	AP16	D19	H34	DVSS	AR27
A27	B18	C4D1	AM26	$\overline{CSTRB3}$	AP18	D20	J31	DVSS	G1
A28	C17	C4D2	AK24	$\overline{CSTRB4}$	AM34	D21	G33	DVSS	N35
A29	D20	C4D3	AL27	$\overline{CSTRB5}$	AK34	D22	K30	DVSS	AR9
A30	B16	C4D4	AP28	CVSS	AR19	D23	F34	EMU0	AA35
$\overline{AE}$	AG31	C4D5	AK26	CVSS	AR7	D24	H32	EMU1	AD34
C0D0	AP4	C4D6	AN29	CVSS	N1	D25	E33	GADVDD	B2
C0D1	AL5	C4D7	AM28	CVSS	AL35	D26	D34	GADVDD	AR1
C0D2	AN5	C5D0	AL29	CVSS	A27	D27	G31	GADVDD	U35
C0D3	AM4	C5D1	AP30	CVSS	A9	D28	C33	GDDVDD	V2
C0D4	AP6	C5D2	AK28	CVSS	E1	D29	H30	GDDVDD	A35
C0D5	AM6	C5D3	AN31	CVSS	J35	D30	E31	GDDVDD	A1

Table 14-1. TMS320C40 Pin Assignments Sorted by Signal Name (Concluded)

Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin
H1	AC3	LA25	R5	LD26	B4	STAT0	AD32
H3	AC5	LA26	T2	LD27	F8	STAT1	AE33
$\overline{\text{IACK}}$	W3	LA27	U3	LD28	D6	STAT2	AF34
IIOF0	AN3	LA28	T4	LD29	C3	STAT3	AE31
IIOF1	AL3	LA29	V4	LD30	E5	$\overline{\text{STRB0}}$	AD30
IIOF2	AH6	LA30	U5	LD31	F6	$\overline{\text{STRB1}}$	AC33
IIOF3	AK2	LADVDD	B34	LDDVDD	AR35	SUBS	C31
IVSS	AR5	LADVDD	AB2	LDDVDD	AP2	TCK	Y34
IVSS	AR31	LADVDD	AP34	LDDVDD	U1	TCLK0	AE3
IVSS	AG35	$\overline{\text{LAE}}$	AB4	$\overline{\text{LDE}}$	AD4	TCLK1	AD2
IVSS	A31	$\overline{\text{LCE0}}$	AG5	$\overline{\text{LLOCK}}$	AA5	TDO	AB34
IVSS	J1	$\overline{\text{LCE1}}$	AF2	$\overline{\text{LOCK}}$	W33	TDI	AC35
IVSS	A5	LD0	E19	LPAGE0	AH2	TMS	W35
LA0	D2	LD1	C15	LPAGE1	AG3	$\overline{\text{TRST}}$	AE35
LA1	D4	LD2	D18	$\overline{\text{LRDY0}}$	AF6	VDDL	AN1
LA2	E3	LD3	B14	$\overline{\text{LRDY1}}$	AE5	VDDL	AN35
LA3	F4	LD4	E17	$\overline{\text{LRW0}}$	AH4	VDDL	C35
LA4	H6	LD5	D16	$\overline{\text{LRW1}}$	AF4	VDDL	C1
LA5	F2	LD6	C13	LSTAT0	AA3	VSSL	A3
LA6	G5	LD7	E15	LSTAT1	Y4	VSSL	AR3
LA7	G3	LD8	B12	LSTAT2	Y2	VSSL	AR33
LA8	H4	LD9	D14	LSTAT3	W5	VSSL	A33
LA9	H2	LD10	C11	$\overline{\text{LSTRB0}}$	AJ3	X1	W1
LA10	K6	LD11	E13	$\overline{\text{LSTRB1}}$	AD6	X2/CLKIN	AA1
LA11	M6	LD12	B10	$\overline{\text{NMI}}$	AJ5		
LA12	J5	LD13	D12	PAGE0	AG33		
LA13	J3	LD14	C9	PAGE1	AB32		
LA14	K4	LD15	E11	$\overline{\text{RDY0}}$	Y32		
LA15	K2	LD16	F12	$\overline{\text{RDY1}}$	W31		
LA16	L3	LD17	D10	RESETLOC0	AF30		
LA17	L5	LD18	BB	RESETLOC1	AH34		
LA18	M2	LD19	E9	$\overline{\text{RESET}}$	AJ33		
LA19	M4	LD20	C7	ROMEN	AK4		
LA20	N3	LD21	F10	$\overline{\text{R/W0}}$	AF32		
LA21	N5	LD22	B6	$\overline{\text{R/W1}}$	AC31		
LA22	P2	LD23	D8				
LA23	P4	LD24	C5				
LA24	R3	LD25	E7				

Table 14-2. TMS320C40 Pin Assignments Sorted by Pin Number

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
A0	GDDV <sub>DD</sub>	AD30	$\overline{\text{STRB0}}$	AK24	C4D2	AM30	C5D4
A3	V <sub>SSL</sub>	AD32	STAT0	AK26	C4D5	AM32	$\overline{\text{C5D6}}$
A5	IV <sub>SS</sub>	AD34	EMU1	AK28	C5D2	AM34	$\overline{\text{CSTRB4}}$
A7	DV <sub>DD</sub>	AE1	CV <sub>SS</sub>	AK32	$\overline{\text{CACK4}}$	AN1	V <sub>DDL</sub>
A9	CV <sub>SS</sub>	AE3	TCLK0	AK34	$\overline{\text{CSTRB5}}$	AN3	IIOF0
A11	DV <sub>SS</sub>	AE5	$\overline{\text{LRDY1}}$	AL1	CV <sub>SS</sub>	AN5	C0D2
A13	DV <sub>DD</sub>	AE31	STAT3	AL3	IIOF1	AN7	C0D6
A15	DV <sub>SS</sub>	AE33	STAT1	AL5	C0D1	AN9	C1D5
A17	DV <sub>DD</sub>	AE35	$\overline{\text{TRST}}$	AL7	C1D0	AN11	$\overline{\text{CACK0}}$
A19	CV <sub>SS</sub>	AF2	$\overline{\text{LCE1}}$	AL9	C1D6	AN13	$\overline{\text{CACK1}}$
A21	DV <sub>SS</sub>	AF4	LR/ $\overline{\text{W1}}$	AL11	$\overline{\text{CSTRB0}}$	AN15	$\overline{\text{CREQ2}}$
A23	DV <sub>DD</sub>	AF6	$\overline{\text{LRDY0}}$	AL13	$\overline{\text{CSTRB1}}$	AN17	$\overline{\text{CREQ3}}$
A25	DV <sub>SS</sub>	AF30	RESETLOC0	AL15	$\overline{\text{CRDY2}}$	AN19	C2D1
A27	CV <sub>SS</sub>	AF32	R/ $\overline{\text{W0}}$	AL17	$\overline{\text{CRDY3}}$	AN21	C2D5
A29	DV <sub>DD</sub>	AF34	STAT2	AL19	C2D2	AN23	C3D1
A31	IV <sub>SS</sub>	AG1	DV <sub>SS</sub>	AL21	C2D6	AN25	C3D5
A33	V <sub>SSL</sub>	AG3	LPAGE1	AL23	C3D2	AN27	C4D0
A35	GDDV <sub>DD</sub>	AG5	$\overline{\text{LCE0}}$	AL25	C3D6	AN29	C4D6
AA1	X2/CLKIN	AG31	$\overline{\text{AE}}$	AL27	C4D3	AN31	C5D3
AA3	LSTAT0	AG33	PAGE0	AL29	C5D0	AN33	$\overline{\text{CREQ4}}$
AA5	$\overline{\text{LLOCK}}$	AG35	IV <sub>SS</sub>	AL31	C5D7	AN35	V <sub>DDL</sub>
AA31	$\overline{\text{DE}}$	AH2	LPAGE0	AL33	$\overline{\text{CREQ5}}$	AP2	LDDV <sub>DD</sub>
AA33	$\overline{\text{CEO}}$	AH4	LR/ $\overline{\text{W0}}$	AL35	CV <sub>SS</sub>	AP4	C0D0
AA35	EMU0	AH6	IIOF2	AM2	DV <sub>SS</sub>	AP6	C0D4
AB2	LADV <sub>DD</sub>	AH30	$\overline{\text{CRDY4}}$	AM4	C0D3	AP8	C1D1
AB4	$\overline{\text{LAE}}$	AH32	$\overline{\text{CRDY5}}$	AM6	C0D5	AP10	C1D7
AB32	PAGE1	AH34	RESETLOC1	AM8	C1D2	AP12	$\overline{\text{CRDY0}}$
AB34	TDO	AJ1	DV <sub>DD</sub>	AM10	$\overline{\text{CREQ0}}$	AP14	$\overline{\text{CRDY1}}$
AC1	DV <sub>DD</sub>	AJ3	$\overline{\text{LSTRB0}}$	AM12	$\overline{\text{CREQ1}}$	AP16	$\overline{\text{CSTRB2}}$
AC3	H1	AJ5	$\overline{\text{NMI}}$	AM14	$\overline{\text{CACK2}}$	AP18	$\overline{\text{CSTRB3}}$
AC5	H3	AJ31	$\overline{\text{CACK5}}$	AM16	$\overline{\text{CACK3}}$	AP20	C2D3
AC31	R/ $\overline{\text{W1}}$	AJ33	$\overline{\text{RESET}}$	AM18	C2D0	AP22	C2D7
AC33	$\overline{\text{STRB1}}$	AJ35	DV <sub>SS</sub>	AM20	C2D4	AP24	C3D3
AC35	TDI	AK2	IIOF3	AM22	C3D0	AP26	C3D7
AD2	TCLK1	AK4	ROMEN	AM24	C3D4	AP28	C4D4
AD4	$\overline{\text{LDE}}$	AK8	C0D7	AM26	C4D1	AP30	C5D1
AD6	$\overline{\text{LSTRB1}}$	AK10	C1D4	AM28	C4D7	AP32	C5D5
		AK12	C1D3			AP34	LADV <sub>DD</sub>

Table 14-2. TMS320C40 Pin Assignments Sorted by Pin Number (Concluded)

14

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
AR1	GADV <sub>DD</sub>	C1	VDDL	E1	CVSS	H2	LA9	P2	LA22
AR3	V <sub>SSL</sub>	C3	LD29	E3	LA2	H4	LA8	P4	LA23
AR5	IV <sub>SS</sub>	C5	LD24	E5	LD30	H6	LA4	P32	D10
AR7	CV <sub>SS</sub>	C7	LD20	E7	LD25	H30	D29	P34	D5
AR9	DV <sub>SS</sub>	C9	LD14	E9	LD19	H32	D24	R1	DV <sub>SS</sub>
AR11	DV <sub>DD</sub>	C11	LD10	E11	LD15	H34	D19	R3	LA24
AR13	CV <sub>SS</sub>	C13	LD6	E13	LD11	J1	IV <sub>SS</sub>	R5	LA25
AR15	DV <sub>SS</sub>	C15	LD1	E15	LD7	J3	LA13	R31	D8
AR17	DV <sub>DD</sub>	C17	A28	E17	LD4	J5	LA12	R33	D4
AR19	CV <sub>SS</sub>	C19	A23	E19	LD0	J31	D20	R35	CV <sub>SS</sub>
AR21	DV <sub>SS</sub>	C21	A19	E21	A26	J33	D15	T2	LA26
AR23	DV <sub>DD</sub>	C23	A16	E23	A22	J35	CV <sub>SS</sub>	T4	LA28
AR25	CV <sub>SS</sub>	C25	A12	E25	A18	K2	LA15	T32	D6
AR27	DV <sub>SS</sub>	C27	A8	E27	A13	K4	LA14	T34	D2
AR29	DV <sub>DD</sub>	C29	A3	E29	A7	K6	LA10	U1	LDDV <sub>DD</sub>
AR31	IV <sub>SS</sub>	C31	SUBS	E31	D30	K30	D22	U3	LA27
AR33	V <sub>SSL</sub>	C33	D28	E33	D25	K32	D18	U5	LA30
AR35	LDDV <sub>DD</sub>	C35	V <sub>DDL</sub>	E35	CV <sub>SS</sub>	K34	D13	U31	D3
B2	GADV <sub>DD</sub>	D2	LA0	F2	LA5	L1	DV <sub>DD</sub>	U33	D0
B4	LD26	D4	LA1	F4	LA3	L3	LA16	U35	GADV <sub>DD</sub>
B6	LD22	D6	LD28	F6	LD31	L5	LA17	V2	GDDV <sub>DD</sub>
B8	LD18	D8	LD23	F8	LD27	L31	D16	V4	LA29
B10	LD12	D10	LD17	F10	LD21	L33	D11	V32	D1
B12	LD8	D12	LD13	F12	LD16	L35	DV <sub>DD</sub>	V34	$\overline{CE}1$
B14	LD3	D14	LD9	F24	A6	M2	LA18	W1	X1
B16	A30	D16	LD5	F26	A11	M4	LA19	W3	$\overline{IACK}$
B18	A27	D18	LD2	F28	A5	M6	LA11	W5	LSTAT3
B20	A25	D20	A29	F32	D31	M30	D17	W31	$\overline{RDY}1$
B22	A21	D22	A24	F34	D23	M32	D14	W33	$\overline{LOCK}$
B24	A17	D24	A20	G1	DV <sub>SS</sub>	M34	D9	W35	TMS
B26	A14	D26	A15	G3	LA7	N1	CV <sub>SS</sub>	Y2	LSTAT2
B28	A10	D28	A9	G5	LA6	N3	LA20	Y4	LSTAT1
B30	A4	D30	A2	G31	D27	N5	LA21	Y32	$\overline{RDY}0$
B32	A1	D32	A0	G33	D21	N31	D12	Y34	TCK
B34	LADV <sub>DD</sub>	D34	D26	G35	DV <sub>SS</sub>	N33	D7		
						N35	DV <sub>SS</sub>		

## 14.2 Signal Descriptions

This section gives signal descriptions for the TMS320C40 device. Table 14–3 lists each signal, the number of pins, function, and operating mode(s), i.e., input, output, or high-impedance state as indicated by I, O, or Z. All pins labeled NC are not to be connected by the user. A line over a signal name (e.g.,  $\overline{\text{RESET}}$ ) indicates that the signal is active low (true at a logic 0 level). The signals are grouped according to function.

Table 14–3. TMS320C40 Signal Descriptions

Signal	Pins	Type†	Description
<b>Global Bus External Interface (80 pins)</b>			
D(31–0)	32	I/O/Z	32-bit data port of the global external interface
$\overline{\text{DE}}$	1	I	Data bus enable signal for the global external interface
A(30–0)	31	O/Z	31-bit address port of the global external interface
$\overline{\text{AE}}$	1	I	Address bus enable signal for the global bus interface
STAT(3–0)	4	O	Status signals for the global bus interface
$\overline{\text{LOCK}}$	1	O	Lock signal for the global bus interface
$\overline{\text{STRB0}}$ †	1	O/Z	Access strobe 0 for the global bus interface
R/W0	1	O/Z	Read/write signal for $\overline{\text{STRB0}}$ accesses
PAGE0	1	O/Z	Page signal for $\overline{\text{STRB0}}$ accesses
$\overline{\text{RDY0}}$	1	I	Ready signal for $\overline{\text{STRB0}}$ accesses
$\overline{\text{CE0}}$	1	I	Control enable for the $\overline{\text{STRB0}}$ , PAGE0, and R/W0 signals
$\overline{\text{STRB1}}$ †	1	O/Z	Access strobe 1 for the global bus interface
R/W1	1	O/Z	Read/write signal for $\overline{\text{STRB1}}$ accesses
PAGE1	1	O/Z	Page signal for $\overline{\text{STRB1}}$ accesses
$\overline{\text{RDY1}}$	1	I	Ready signal for $\overline{\text{STRB1}}$ accesses
$\overline{\text{CE1}}$	1	I	Control enable for the $\overline{\text{STRB1}}$ , PAGE1, and R/W1 signals

†  $\overline{\text{STRB0}}$  and  $\overline{\text{STRB1}}$  and associated signals (R/W1, R/W0, PAGE0, PAGE1, etc.) are effective over the address ranges defined by the STRB ACTIVE bits, as listed in Table 7–3 on page 7-8.

‡ I = input, O = output, Z = high impedance.

Table 14-3. TMS320C40 Signal Descriptions (Continued)

Signal	Pins	Type <sup>‡</sup>	Description
<b>Local Bus External Interface (80 pins)</b>			
LD(31-0)	32	I/O/Z	32-bit data port of the local external interface
$\overline{\text{LDE}}$	1	I	Data bus enable signal for the local external interface
LA(30-0)	31	O/Z	31-bit address port of the local external interface
$\overline{\text{LAE}}$	1	I	Address bus enable signal for the local bus interface
LSTAT(3-0)	4	O	Status signals for the local bus interface
$\overline{\text{LLOCK}}$	1	O	Lock signal for the local bus interface
$\overline{\text{LSTRB0}}$ †	1	O/Z	Access strobe 0 for the local bus interface
LR $\overline{\text{W0}}$	1	O/Z	Read/write signal for $\overline{\text{LSTRB0}}$ accesses
LPAGE0	1	O/Z	Page signal for $\overline{\text{LSTRB0}}$ accesses
$\overline{\text{LRDY0}}$	1	I	Ready signal for $\overline{\text{LSTRB0}}$ accesses
$\overline{\text{LCE0}}$	1	I	Control enable for the $\overline{\text{LSTRB0}}$ , LPAGE0, and LR $\overline{\text{W0}}$ signals
$\overline{\text{LSTRB1}}$ †	1	O/Z	Access strobe 1 for the local bus interface
LR $\overline{\text{W1}}$	1	O/Z	Read/write signal for $\overline{\text{LSTRB1}}$ accesses
LPAGE1	1	O/Z	Page signal for $\overline{\text{LSTRB1}}$ accesses
$\overline{\text{LRDY1}}$	1	I	Ready signal for $\overline{\text{LSTRB1}}$ accesses
$\overline{\text{LCE1}}$	1	I	Control enable for the $\overline{\text{LSTRB1}}$ , LPAGE1, and LR $\overline{\text{W1}}$ signals
<b>Communication Port 0 Interface (12 pins)</b>			
C0D(7-0)	8	I/O	Communication port 0 data bus
$\overline{\text{CREQ0}}$	1	I/O	Communication port 0 token request signal
$\overline{\text{CACK0}}$	1	I/O	Communication port 0 token request acknowledge signal
$\overline{\text{CSTRB0}}$	1	I/O	Communication port 0 data strobe signal
$\overline{\text{CRDY0}}$	1	I/O	Communication port 0 data ready signal

†  $\overline{\text{LSTRB0}}$  and  $\overline{\text{LSTRB1}}$  and associated signals (LR $\overline{\text{W1}}$ , LR $\overline{\text{W0}}$ , LPAGE0, LPAGE1, etc.) are effective over the address ranges defined by the STRB ACTIVE bits, as listed in Table 7-3 on page 7-8.

‡ I = input, O = output, Z = three-stated (high impedance).

Table 14-3. TMS320C40 Signal Descriptions (Continued)

Signal	Pins	Type <sup>‡</sup>	Description
<b>Communication Port 1 Interface (12 pins)</b>			
C1D(7 – 0)	8	I/O	Communication port 1 data bus
$\overline{\text{CREQ1}}$	1	I/O	Communication port 1 token request signal
$\overline{\text{CACK1}}$	1	I/O	Communication port 1 token request acknowledge signal
$\overline{\text{CSTRB1}}$	1	I/O	Communication port 1 data strobe signal
$\overline{\text{CRDY1}}$	1	I/O	Communication port 1 data ready signal
<b>Communication Port 2 Interface (12 pins)</b>			
C2D(7 – 0)	8	I/O	Communication port 2 data bus
$\overline{\text{CREQ2}}$	1	I/O	Communication port 2 token request signal
$\overline{\text{CACK2}}$	1	I/O	Communication port 2 token request acknowledge signal
$\overline{\text{CSTRB2}}$	1	I/O	Communication port 2 data strobe signal
$\overline{\text{CRDY2}}$	1	I/O	Communication port 2 data ready signal
<b>Communication Port 3 Interface (12 pins)</b>			
C3D(7 – 0)	8	I/O	Communication port 3 data bus
$\overline{\text{CREQ3}}$	1	I/O	Communication port 3 token request signal
$\overline{\text{CACK3}}$	1	I/O	Communication port 3 token request acknowledge signal
$\overline{\text{CSTRB3}}$	1	I/O	Communication port 3 data strobe signal
$\overline{\text{CRDY3}}$	1	I/O	Communication port 3 data ready signal
<b>Communication Port 4 Interface (12 pins)</b>			
C4D(7 – 0)	8	I/O	Communication port 4 data bus
$\overline{\text{CREQ4}}$	1	I/O	Communication port 4 token request signal
$\overline{\text{CACK4}}$	1	I/O	Communication port 4 token request acknowledge signal
$\overline{\text{CSTRB4}}$	1	I/O	Communication port 4 data strobe signal
$\overline{\text{CRDY4}}$	1	I/O	Communication port 4 data ready signal
<b>Communication Port 5 Interface (12 pins)</b>			
C5D(7 – 0)	8	I/O	Communication port 5 data bus
$\overline{\text{CREQ5}}$	1	I/O	Communication port 5 token request signal
$\overline{\text{CACK5}}$	1	I/O	Communication port 5 token request acknowledge signal
$\overline{\text{CSTRB5}}$	1	I/O	Communication port 5 data strobe signal
$\overline{\text{CRDY5}}$	1	I/O	Communication port 5 data ready signal

<sup>‡</sup> I = input, O = output, Z = three-stated (high impedance).

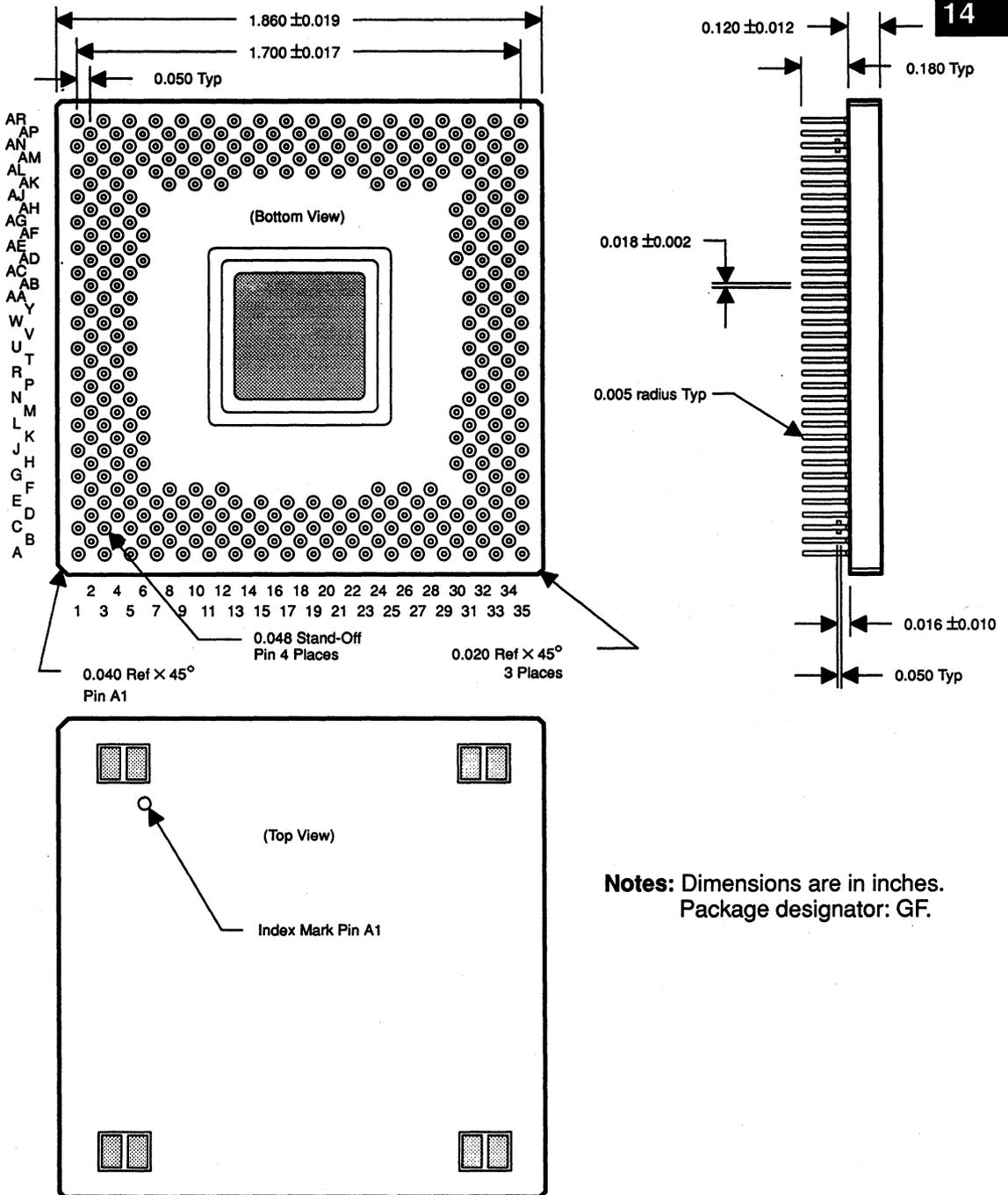
Table 14-3. TMS320C40 Signal Descriptions (Continued)

Signal	Pins	Type‡	Description
<b>Interrupts, I/O Flags, Reset, Timer (12 pins)</b>			
IIOF(3 – 0)	4	I/O	Interrupt and I/O flags
NMI	1	I	Nonmaskable interrupt. It is sensitive to a low-going edge.
$\overline{\text{TACK}}$	1	O	Interrupt acknowledge
$\overline{\text{RESET}}$	1	I	Reset signal
RESETLOC(1,0)	2	I	Reset-vector location pins
ROMEN	1	I	On-chip ROM enable (0 = disable, 1 = enable)
TCLK0	1	I/O	Timer 0 pin
TCLK1	1	I/O	Timer 1 pin
<b>Clock and Power (4 pins)</b>			
X1	1	O	Crystal pin
X2/CLKIN	1	I	Crystal/oscillator pin
H1	1	O	H1 clock
H3	1	O	H3 clock
<b>Emulation (7 pins)</b>			
TCK	1	I	JTAG test port clock
TDO	1	O/T	JTAG test port data out
TDI	1	I	JTAG test port data in
TMS	1	I	JTAG test port mode select
$\overline{\text{TRST}}$	1	I	JTAG test port reset
EMU0	1	I/O	Emulation pin 0
EMU1	1	I/O	Emulation pin 1

‡ I = input, O = output, Z = three-stated (high impedance).

### 14.3 TMS320C4x Mechanical Data

Figure 14-2. TMS320C40 325-Pin PGA Dimensions



## 14.4 Electrical Specifications

14

Table 14–4. Absolute Maximum Ratings Over Specified Temperature Range

Condition/Characteristic	Range
Supply voltage range, $V_{DD}$	– 0.3 V to 7 V
Input voltage range	– 0.3 V to 7 V
Output voltage range	– 0.3 V to 7 V
Operating case temperature range	0 °C to 85 °C
Storage temperature range	– 55 °C to 150 °C

**Notes:** 1) Stresses beyond those listed under “Absolute Maximum Ratings” may cause permanent damage to the device. This is a stress rating only; functional operation of the device at these or any other conditions beyond those indicated in the “Recommended Operating Conditions” table of this specification is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

2) All voltage values are with respect to  $V_{SS}$ .

Table 14–5. Recommended Operating Conditions

Parameter	Min	Nom	Max	Unit
$V_{DD}$ Supply voltages ( $DDV_{DD}$ , etc.)	4.75	5	5.25	V
$V_{SS}$ Supply voltages ( $CV_{SS}$ , etc.)		0		V
$V_{IH}$ High-level input voltage	2		$V_{DD} + 0.3$	V
$V_{IL}$ Low-level input voltage	–0.3		0.8	V
$I_{OH}$ High-level output current			–300	$\mu$ A
$I_{OL}$ Low-level output current			2	mA
T Operating free-air temperature	0		85	°C
$V_{TH}$ CLKIN high-level input voltage for CLKIN	2.6		$V_{DD} + 0.3$	V

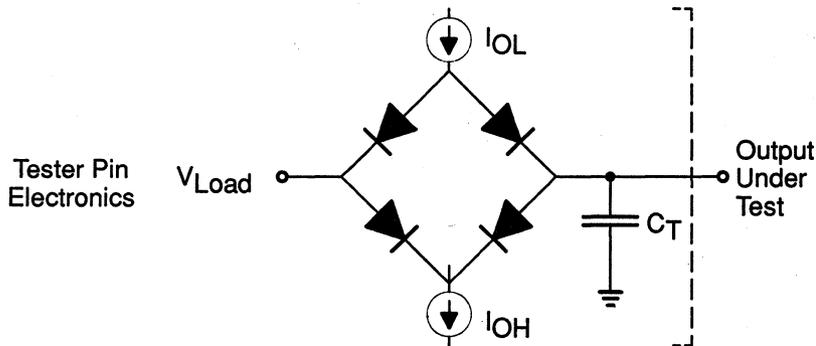
**Note:** Note 1 for Table 14–4 also applies to this table. All inputs and output voltages are TTL compatible.

Table 14–6. Electrical Characteristics Over Specified Free-Air Temperature Range

Electrical Characteristic	Min	Nom(Note 1)	Max	Unit
$V_{OH}$ High-level output voltage ( $V_{DD} = \text{Min}$ , $I_{OH} = \text{Max}$ )	2.4	3		V
$V_{OL}$ Low-level output voltage ( $V_{DD} = \text{Min}$ , $I_{OL} = \text{Max}$ )		0.3	0.6	V
$I_Z$ Three-state current ( $V_{DD} = \text{Max}$ )	-20		20	$\mu\text{A}$
$I_I$ Input current ( $V_I = V_{SS}$ to $V_{DD}$ )	-10		10	$\mu\text{A}$
$I_{IP}$ Input current (Inputs with internal pull-ups) (See Note 4)	-400		20	$\mu\text{A}$
$I_{CC}$ Supply current ( $T_A = 25^\circ\text{C}$ , $V_{DD} = \text{Max}$ , $f_x = \text{Max}$ )		350	850	mA
$C_I$ Input capacitance			15	pF
$C_O$ Output capacitance			15	pF

- Notes:** 1) All nominal values are at  $V_{DD} = 5\text{ V}$ ,  $T_A = 25^\circ\text{C}$ .  
 2)  $f_x$  is the input clock frequency. The maximum value is 50 MHz.  
 3) All input and output voltage levels are TTL compatible.  
 4) Pins with internal pull-up devices: TDI, TCK.  
 5) Pin with internal pull-down device: TRST.

Figure 14–3. Test Load Circuit



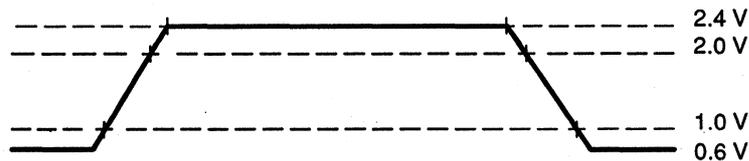
Where:  $I_{OL} = 2.0\text{ mA}$  (all outputs)  
 $I_{OH} = 300\ \mu\text{A}$  (all outputs)  
 $V_{Load} = 2.15\text{ V}$   
 $C_T = 80\text{ pF}$  typical load circuit capacitance.

## 14.5 Signal Transition Levels

TTL-level outputs are driven to a minimum logic-high level of 2.4 volts and to a maximum logic-low level of 0.6 volt. Output transition times are specified as follows.

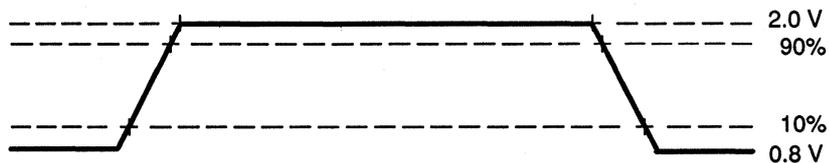
For a high-to-low transition on a TTL-compatible output signal, the level at which the output is said to be no longer high is 2.0 volts, and the level at which the output is said to be low is 1.0 volt. For a low-to-high transition, the level at which the output is said to be no longer low is 1.0 volt, and the level at which the output is said to be high is 2.0 volts.

Figure 14-4. TTL-Level Outputs



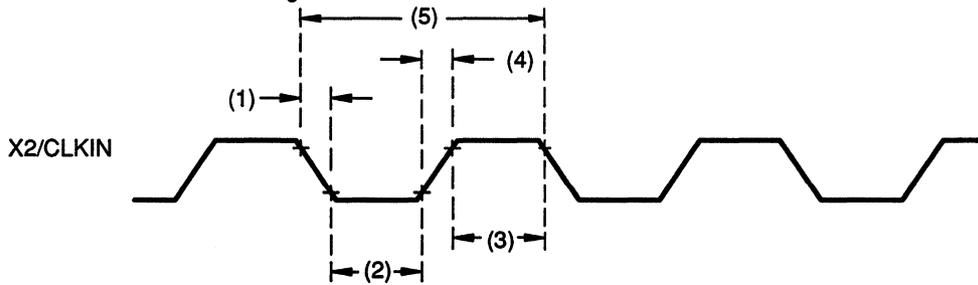
Transition times for TTL-compatible inputs are specified as follows. For a high-to-low transition on an input signal, the level at which the input is said to be no longer high is 2.0 volts, and the level at which the input is said to be low is 0.8 volt. For a low-to-high transition on an input signal, the level at which the input is said to be no longer low is 0.8 volt, and the level at which the input is said to be high is 2.0 volts.

Figure 14-5. TTL-Level Inputs



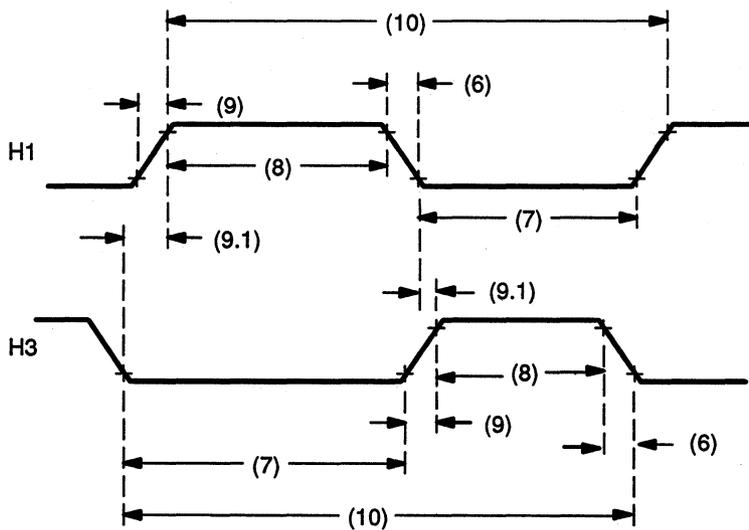
## 14.6 Timing

Figure 14–6. X2/CLKIN Timing



— Timing parameter table on next page —

Figure 14–7. H1/H3 Timing

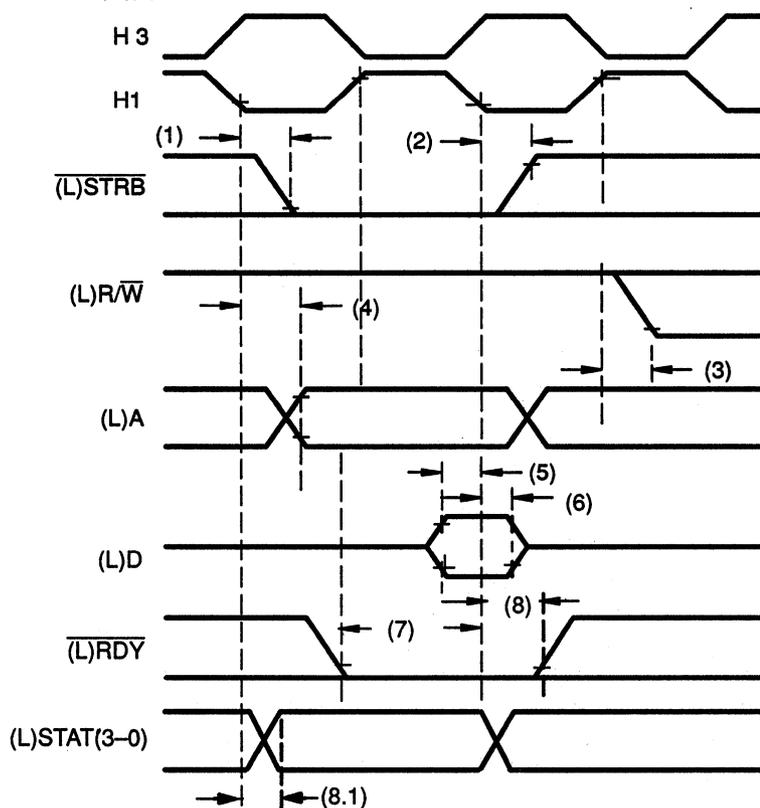


— Timing parameter table on next page —

Table 14–7. Timing Parameters for CLKIN, H1, H3 (Figure 14–6 and Figure 14–7)

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_f(\text{Cl})$	CLKIN fall time		4		4	ns
(2)	$t_w(\text{CLL})$	CLKIN low pulse duration $t_c(\text{Cl}) = \text{min}$	7		8		ns
(3)	$t_w(\text{CIH})$	CLKIN high pulse duration $t_c(\text{Cl}) = \text{min}$	7		8		ns
(4)	$t_r(\text{Cl})$	CLKIN rise time		4		4	ns
(5)	$t_c(\text{Cl})$	CLKIN cycle time	20		25		ns
(6)	$t_f(\text{H})$	H1/H3 fall time		3		3	ns
(7)	$t_w(\text{HL})$	H1/H3 low pulse duration	$P-6\ddagger$		$P-6\ddagger$		ns
(8)	$t_w(\text{HH})$	H1/H3 high pulse duration	$P-7\ddagger$		$P-7\ddagger$		ns
(9)	$t_r(\text{H})$	H1/H3 rise time		4		4	ns
(9.1)	$t_d(\text{HL-HH})$	Delay from H1(H3) low to H3(H1) high	0	5	0	5	ns
(10)	$t_c(\text{H})$	H1/H3 cycle time	40	485	50	500	ns

†  $P = t_c(\text{Cl})$  as shown in Figure 14–6.

Figure 14–8. Memory ( $\overline{(L)STRB} = 0$ ) ReadTable 14–8. Timing Parameters for a Memory ( $\overline{(L)STRB} = 0$ ) Read/Write

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{d(H1L-(L)SL)}$	H1 low to $\overline{(L)STRB}$ low		7		7	ns
(2)	$t_{d(H1L-(L)SH)}$	H1 low to $\overline{(L)STRB}$ high		7		7	ns
(3)	$t_{d(H1H-RWL)}$	H1 high to $(L)R/\overline{W}$ low		7		7	ns
(4)	$t_{d(H1L-A)}$	H1 low to $(L)A$ valid		7		11	ns
(5)	$t_{su(D)R}$	$(L)D$ valid before H1 low (read)	12			13	ns
(6)	$t_{h((L)D)R}$	$(L)D$ hold time after H1 low (read)	0		0		ns
(7)	$t_{su(L)RDY}$	$\overline{(L)RDY}$ valid before H1 low	20		20		ns
(8)	$t_{h((L)RDY)}$	$\overline{(L)RDY}$ hold time after H1 low	0		0		ns
(8.1)	$t_{d(H1L-S)}$	H1 low to $(L)STAT(3-0)$ valid		7		11	ns

**Note:** For consecutive reads,  $(L)R/\overline{W}$  stays high and  $\overline{(L)STRB}$  stays low.

— Table continued on next page —

Figure 14-9. Memory ((L)STRB = 0) Write

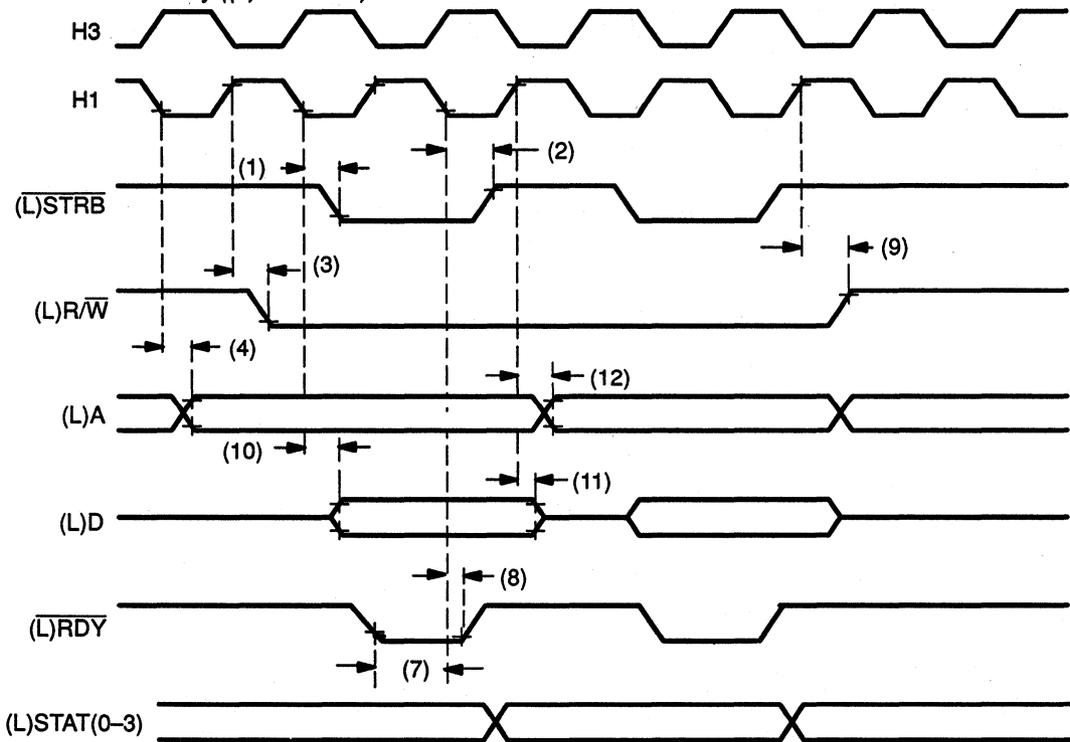
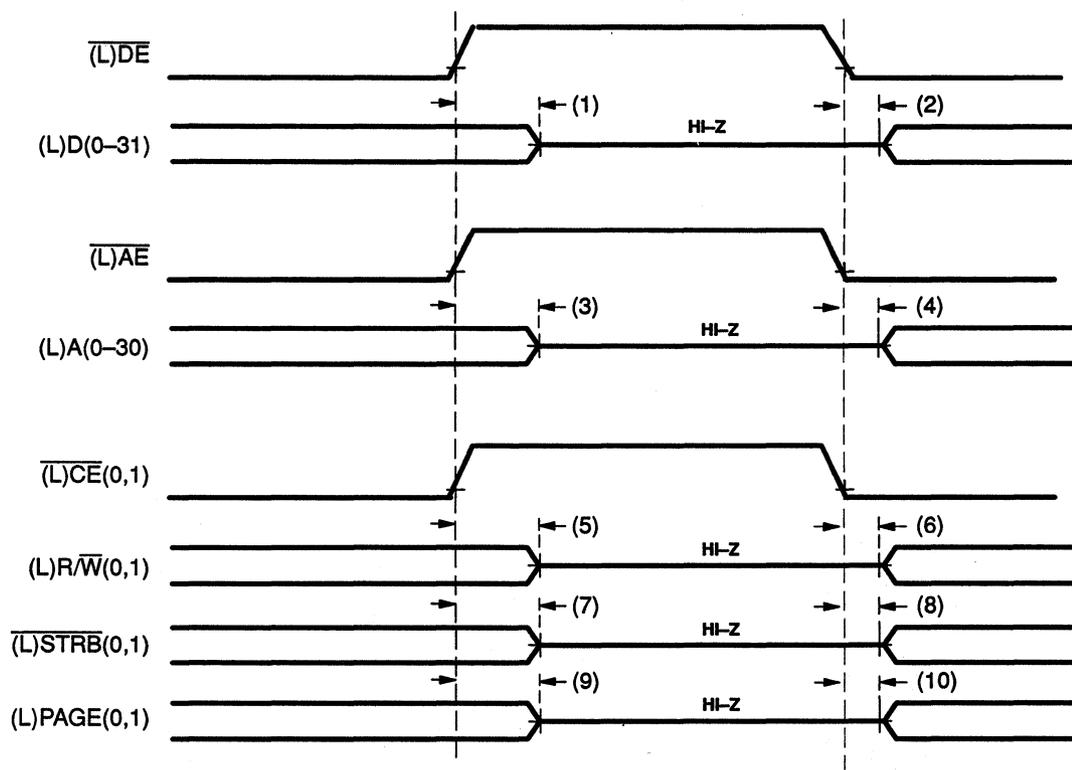


Table 14-8. Timing Parameters for a Memory ((L)STRB = 0) Read/Write (Concluded)

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(9)	$t_{d(H1H-(L)RWH)}$	H1 high to (L)R/W high (write)		7		7	ns
(10)	$t_{v((L)D)W}$	(L)D valid after H1 low (write)		16		16	ns
(11)	$t_{h((L)D)W}$	(L)D hold time after H1 high (write)	0		0		ns
(12)	$t_{d(H1H-A)}$	H1 high to A valid on back-to-back write cycles (write)		13		15	ns

**Note:** The delay for (L)RDY to become active after the address is valid should be a maximum of 13 ns for the 'C40 and 19 ns for the 'C40-40.

Figure 14-10.  $\overline{DE}$ ,  $\overline{AE}$ , and  $\overline{CE}$  Enable TimingTable 14-9.  $\overline{DE}$ ,  $\overline{AE}$ , and  $\overline{CE}$  Enable Timing

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_d(\text{DEH-DZ})$	Time $\overline{(L)DE}$ high to (L)D(0-31) HI-Z	0	15	0	15	ns
(2)	$t_d(\text{DEL-DV})$	Time $\overline{(L)DE}$ low to (L)D(0-31) valid	0	15	0	15	ns
(3)	$t_d(\text{AEH-AZ})$	Time $\overline{(L)AE}$ high to (L)A(0-31) HI-Z	0	15	0	15	ns
(4)	$t_d(\text{AEL-AV})$	Time $\overline{(L)AE}$ low to (L)A(0-31) valid	0	15	0	15	ns
(5)	$t_d(\text{CEH-RWZ})$	Time $\overline{(L)CE}$ high to (L)R/ $\overline{W}$ (0,1) HI-Z	0	15	0	15	ns
(6)	$t_d(\text{CEL-RWV})$	Time $\overline{(L)CE}$ low to (L)R/ $\overline{W}$ (0,1) valid	0	15	0	15	ns
(7)	$t_d(\text{CEH-STRBZ})$	Time $\overline{(L)CE}$ high to (L)STRB(0,1) in high impedance state	0	15	0	15	ns
(8)	$t_d(\text{CEL-STRBV})$	Time $\overline{(L)CE}$ low to (L)STRB(0,1) valid	0	15	0	15	ns
(9)	$t_d(\text{CEH-PAGEZ})$	Time $\overline{(L)CE}$ high to (L)PAGE(0,1) in high impedance state	0	15	0	15	ns
(10)	$t_d(\text{CEL-PAGEV})$	Time $\overline{(L)CE}$ low to (L)PAGE(0,1) valid	0	15	0	15	ns

Figure 14–11. Timing for  $\overline{(L)LOCK}$  When Executing LDFI or LDII

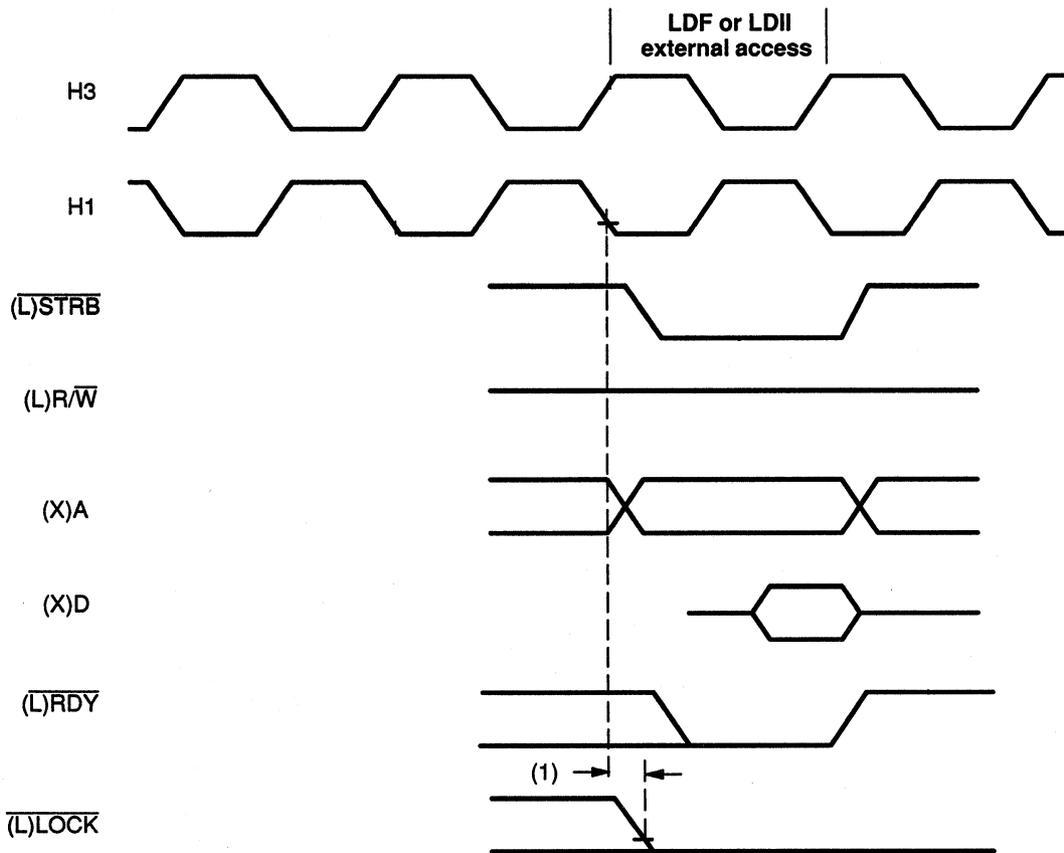
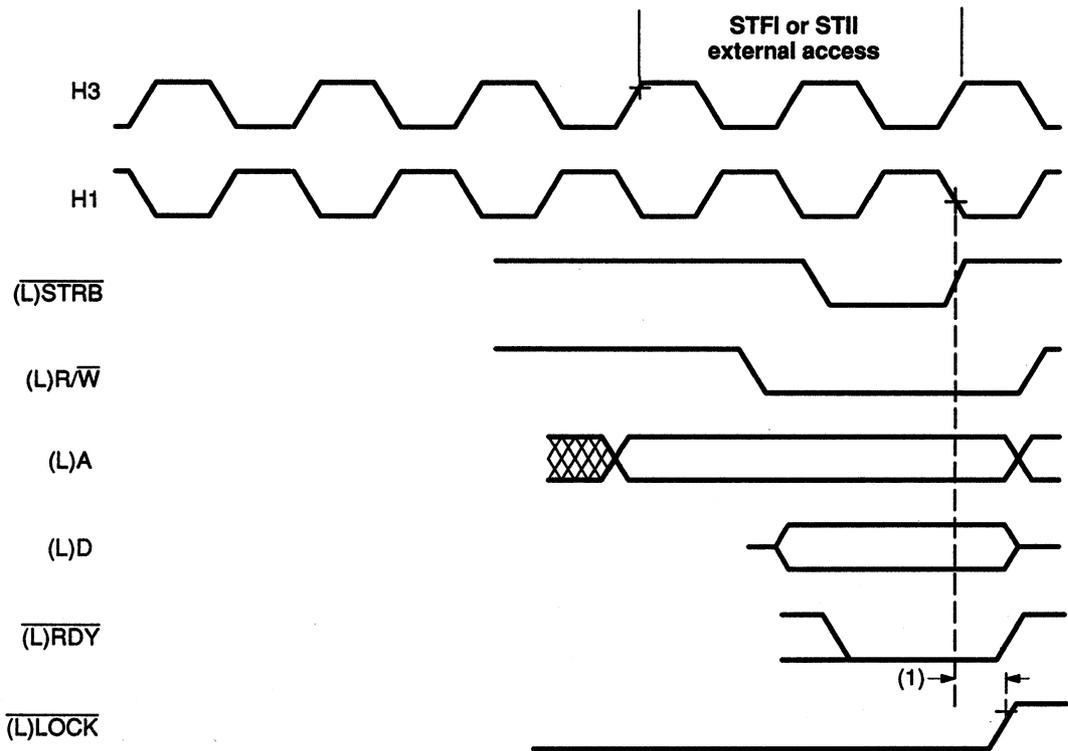


Table 14–10. Timing Parameters for  $\overline{(L)LOCK}$  When Executing LDFI or LDII

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_d(H1L-LOCKL)$	H1 low to $\overline{(L)LOCK}$ low		7		11	ns

Figure 14–12. Timing for  $\overline{(L)}\text{LOCK}$  When Executing a STFI or STIITable 14–11. Timing Parameters for  $\overline{(L)}\text{LOCK}$  When Executing STFI or STII

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{d(H1L-LOCKH)}$	H1 low to $\overline{(L)}\text{LOCK}$ high		7		11	ns

Figure 14–13. Timing for  $\overline{(L)}\text{LOCK}$  When Executing SIGI

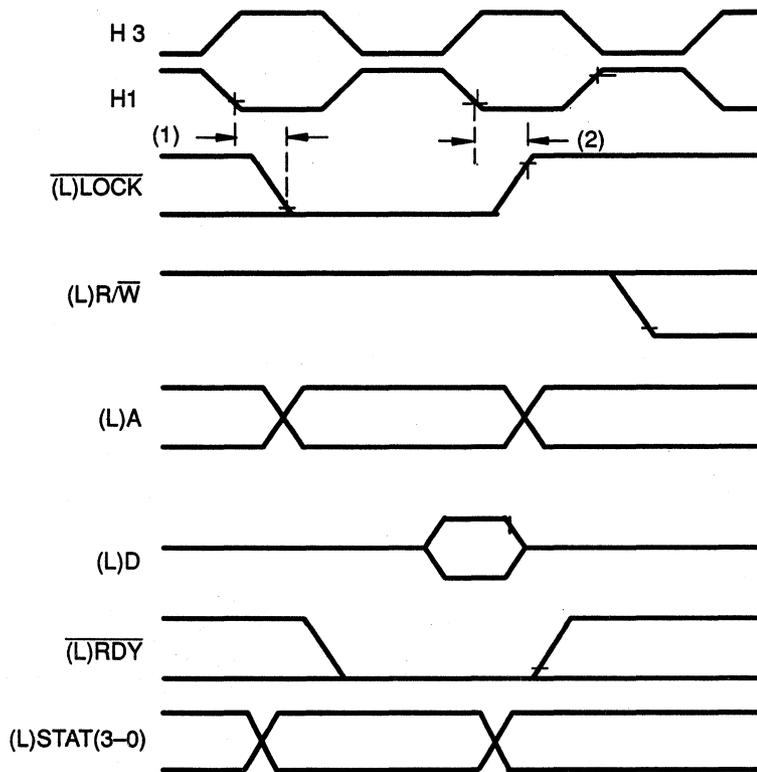


Table 14–12. Timing Parameters for  $\overline{(L)}\text{LOCK}$  When Executing SIGI

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_d(\text{H1L}-\text{LOCKL})$	H1 low to $\overline{(L)}\text{LOCK}$ high		7		11	ns
(2)	$t_d(\text{H1L}-\text{LOCKH})$	H1 low to $\overline{(L)}\text{LOCK}$ high		7		11	

Figure 14–14. Timing Parameters for (L)PAGE(0,1)

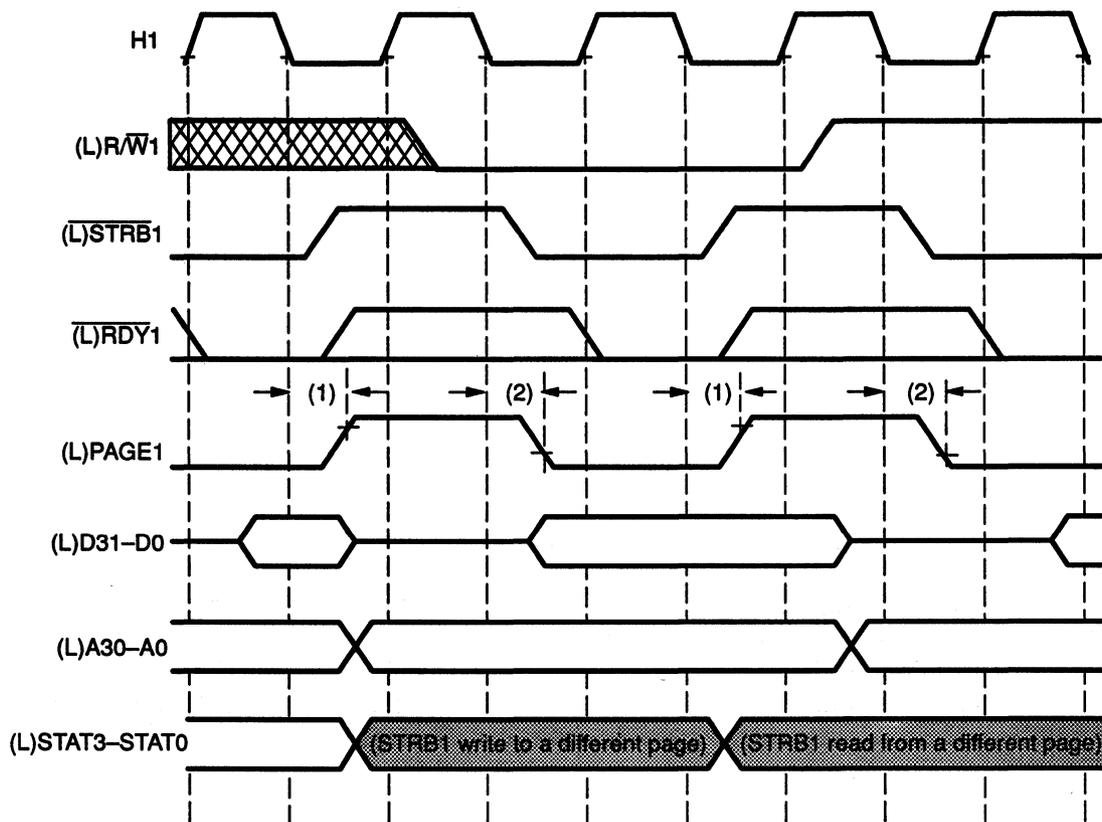


Table 14–13. Timing Parameters for (L)PAGE(0,1) During Memory Accesses to a Different Page

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_d(H1L-PH)$	H1 low to PAGE high for access to different page	0	7		11	ns
(2)	$t_d(H1L-PL)$	H1 low to PAGE low for access to different page	0	7		11	ns

Figure 14–15. Timing for Loading IIF Register (IIOF Pins) When Configured as an Output Pin

14

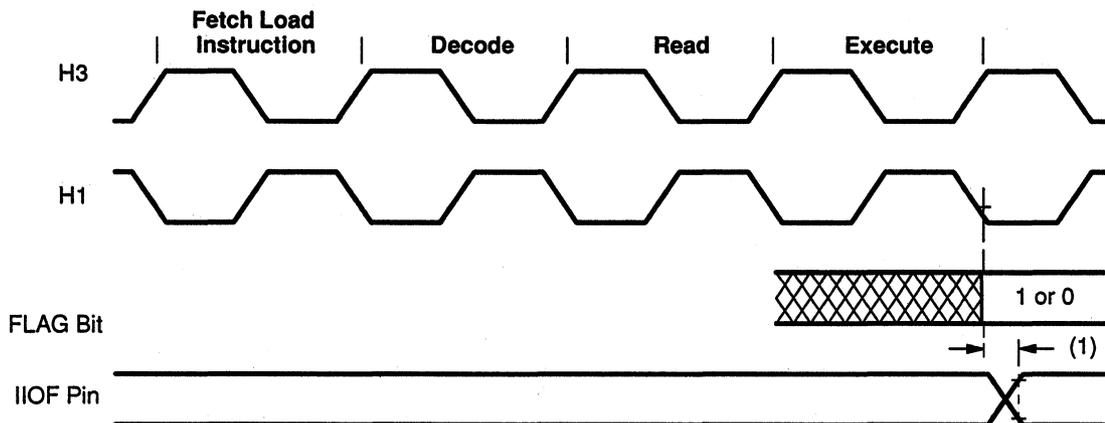


Table 14–14. Timing Parameters for Loading IIF Register When Configured as an Output Pin

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{v(H1L-IF)}$	H1 low to IIOF valid		11		12	ns

Figure 14–16. Change of IIOF From Output to Input Mode

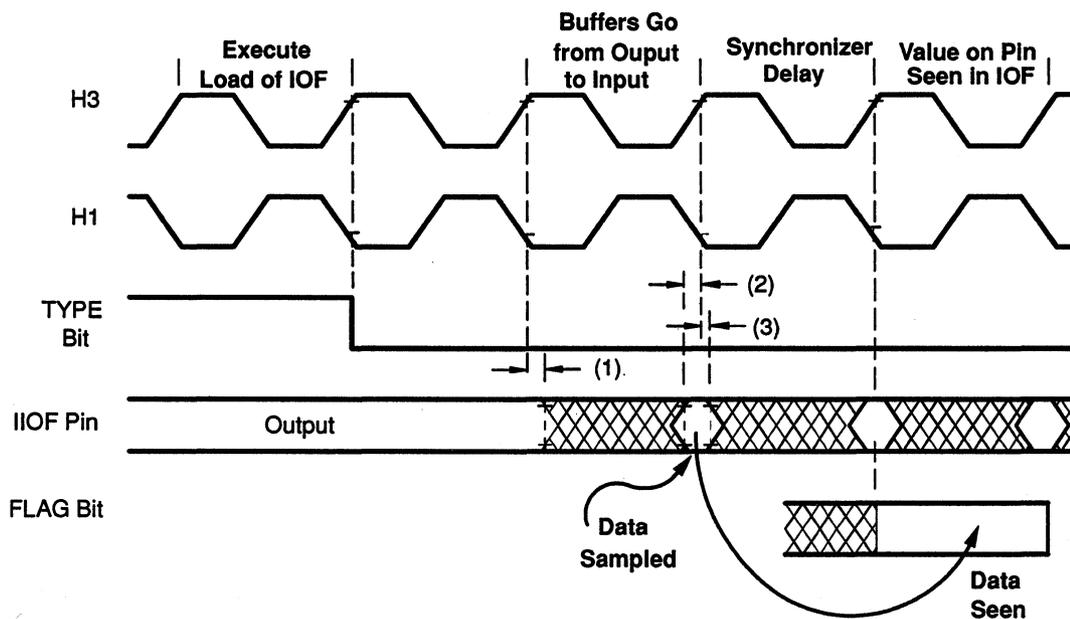


Table 14–15. Timing Parameters of IIOF Changing From Output to Input Mode

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{h(H1L-IF01)}$	IIOF hold after H1 low		11		12	ns
(2)	$t_{su(IF)}$	IIOF setup before H1 low	8		8		ns
(3)	$t_{h(IF)}$	IIOF hold after H1 low	0		0		ns

Figure 14–17. Change of IIOF From Input to Output Mode

14

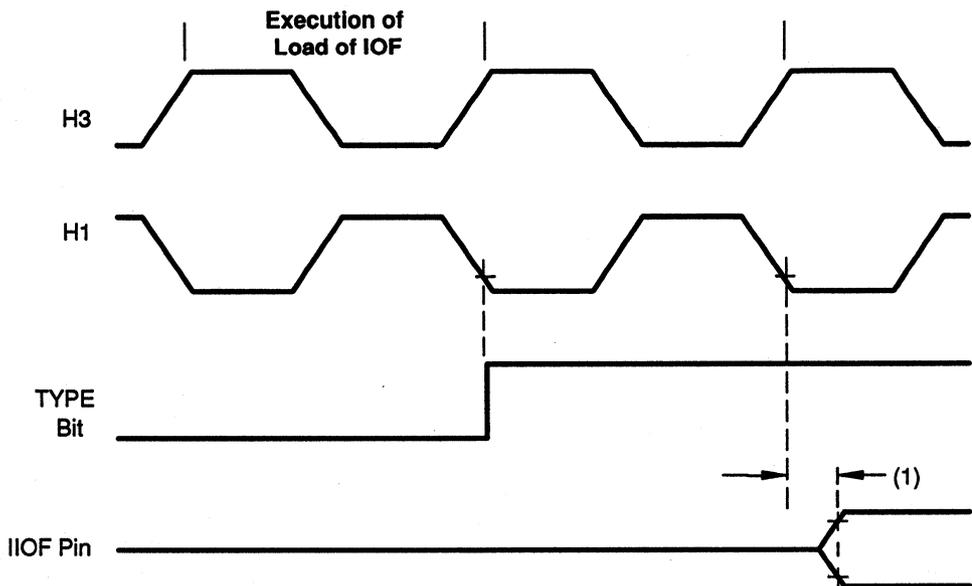
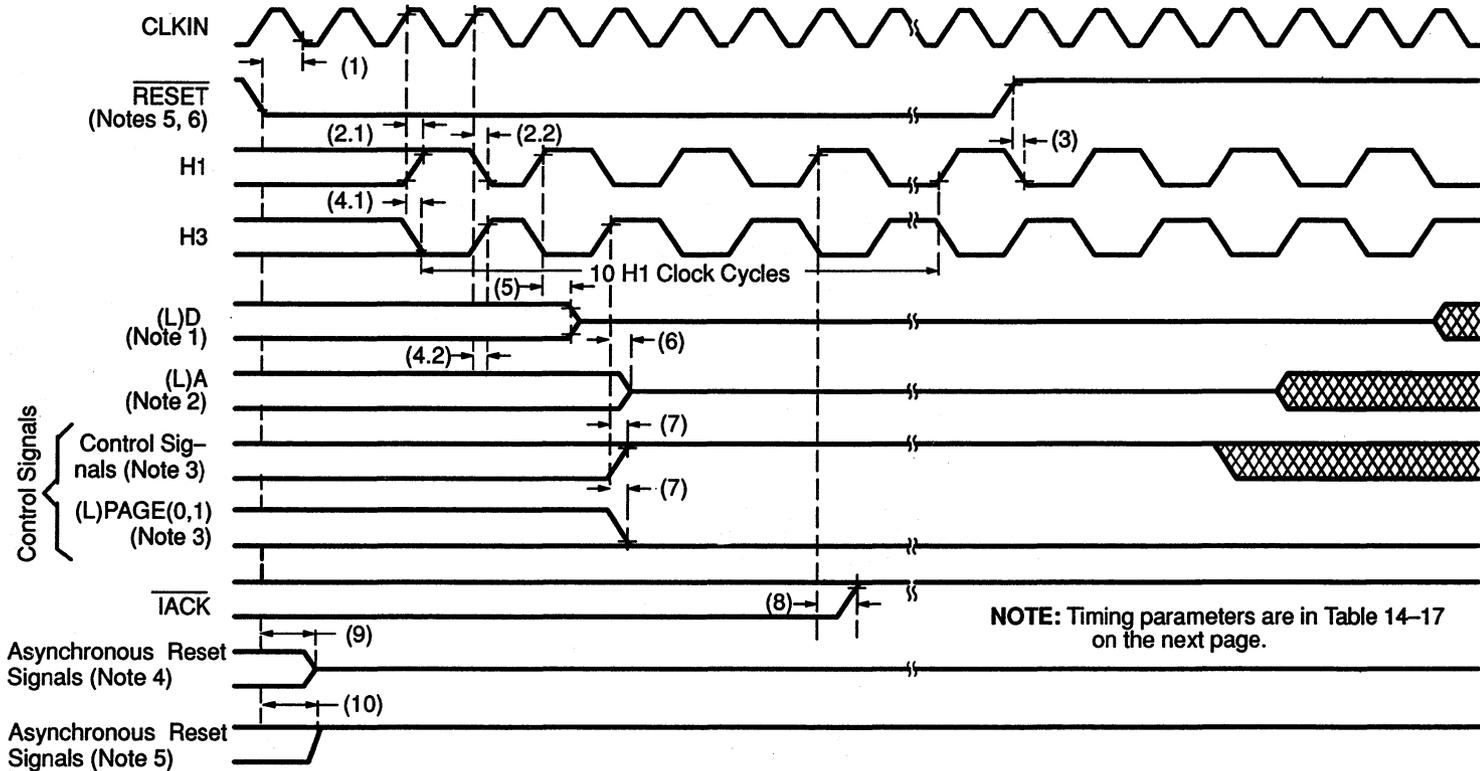


Table 14–16. Timing Parameters of IIOF Changing from Input to Output Mode

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_d(H1L-XFIO)$	H1 low to IIOF switching from input to output		16		16	ns

Figure 14-18.  $\overline{\text{RESET}}$  Timing

**Notes:** 1) (L)D includes D(31 - 0), LD(31 - 0), and CxD(7 - 0).

2) L(A) includes A(30 - 0).

3) Control signals  $\overline{\text{LSTRB0}}$ ,  $\overline{\text{LSTRB1}}$ ,  $\overline{\text{STRB0}}$ ,  $\overline{\text{STRB1}}$ , (L)STAT(3 - 0),  $\overline{\text{(L)LOCK}}$ , (L) $\overline{\text{R}\overline{\text{W}}0}$ , and (L) $\overline{\text{R}\overline{\text{W}}1}$  go high while (L)PAGE0, and (L)PAGE1 go low.

4) Asynchronously reset signals that go into high impedance after  $\overline{\text{RESET}}$  goes low include TCLK0, TCLK1, IIOF(3 - 0), and the communication port control signals  $\overline{\text{CREQx}}$ ,  $\overline{\text{CACKy}}$ ,  $\overline{\text{CSTRBy}}$ , and  $\overline{\text{CRDYx}}$  (where x = 0, 1, or 2, and y = 3, 4, or 5). (At reset, ports 0, 1, and 2 become outputs, and ports 3, 4, and 5 become inputs.)

5) Asynchronously reset signals that go to a high logic level after  $\overline{\text{RESET}}$  goes low include  $\overline{\text{CREQy}}$ ,  $\overline{\text{CACKx}}$ ,  $\overline{\text{CSTRBx}}$ , and  $\overline{\text{CRDYy}}$  (where x = 0, 1, or 2, and y = 3, 4, or 5).

6)  $\overline{\text{RESET}}$  is an asynchronous input and can be asserted at any point during a clock cycle. If the specified timings are met, the exact sequence shown will occur; otherwise, an additional delay of one clock cycle may occur.

Table 14–17. Timing Parameters for  $\overline{\text{RESET}}$  (Figure 14–18)

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{\text{su}}(\text{RESET})$	Setup for $\overline{\text{RESET}}$ before CLKIN low	8	$P^\dagger$	8	$P^\dagger$	ns
(2.1)	$t_{\text{d}}(\text{CLKINH-H1H})$	CLKIN high to H1 high	5	12	5	13	ns
(2.2)	$t_{\text{d}}(\text{CLKINH-H1L})$	CLKIN high to H1 low	5	12	5	13	ns
(3)	$t_{\text{su}}(\text{RESETH-H1L})$	Setup for $\overline{\text{RESET}}$ high before H1low and after 10 H1 clock cycles	8		8		ns
(4.1)	$t_{\text{d}}(\text{CLKINH-H3L})$	CLKIN high to H3 low	5	12	5	13	ns
(4.2)	$t_{\text{d}}(\text{CLKINH-H3H})$	CLKIN high to H3 high	5	12	5	13	ns
(5)	$t_{\text{dis}}(\text{H1H-XD})$	H1 high to (L)D high-impedance		15		15	ns
(6)	$t_{\text{dis}}(\text{H3H-XA})$	H3 high to (L)A high-impedance		15		15	ns
(7)	$t_{\text{d}}(\text{H3H-CONTROLH})$	H3 high to control signals high (low for (L)Page)		7		7	ns
(8)	$t_{\text{d}}(\text{H1H-IACKH})$	H1 high to $\overline{\text{IACK}}$ high		7		7	ns
(9)	$t_{\text{dis}}(\text{RESETL-ASYNCH})$	$\overline{\text{RESET}}$ low to asynchronously reset signals high-impedance		15		15	ns
(10)	$t_{\text{d}}(\text{RESETL-COMMH})$	$\overline{\text{RESET}}$ low to asynchronously reset signals high		10		10	ns

$^\dagger P = t_{\text{c}}(\text{cl})$ , the CLKIN period as shown in Figure 14–6.

Figure 14–19. IOOF(3–0) Interrupt Response Timing

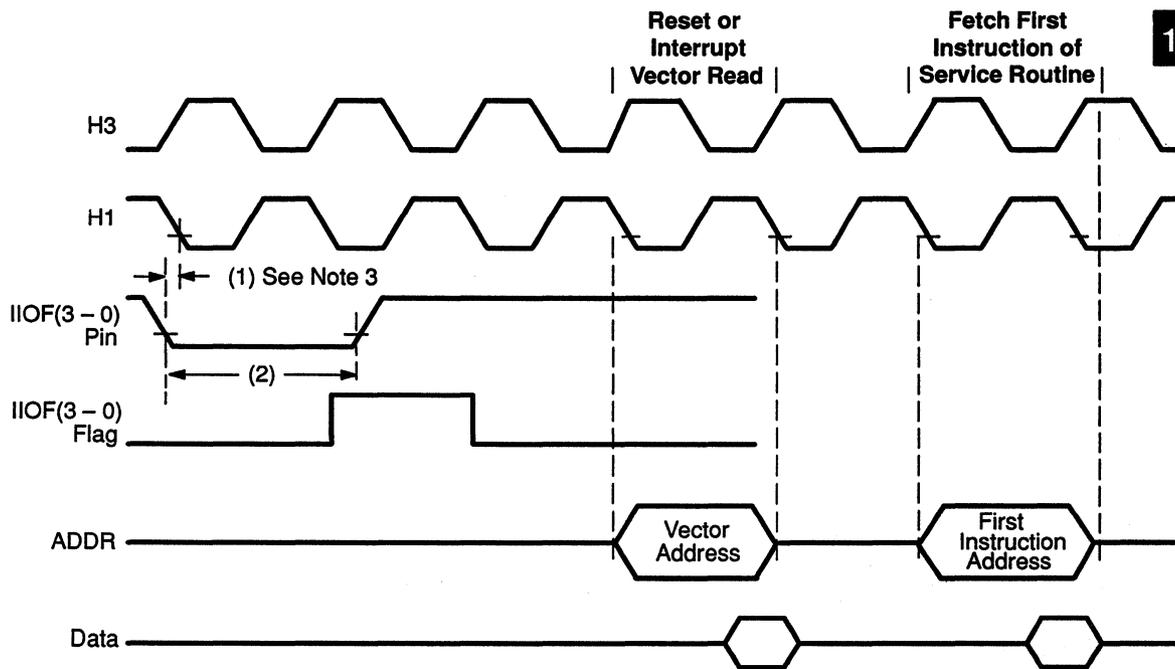


Table 14–18. Timing Parameters for IOOF(3–0)

No.	Name	Description	TMS320C40			TMS320C40-40			Unit
			Min	Typ	Max	Min	Typ	Max	
(1)	$t_{su}(IOOF)$	IOOF(3–0) setup before H1 low	11			12			ns
(2)	$t_w(IOOF)$ (See Note 1)	Interrupt pulse width to guarantee one interrupt seen	P	1.5P	<2P	P	1.5P	<2P	ns

**Notes:** 1) Interrupt pulse width must be at least 1 P wide (P = one H1 period) to guarantee it will be seen. It must be less than 2 P wide to guarantee it will be responded to only once. Recommended pulse width is 1.5 P.

2) IOOF is an asynchronous input and can be asserted at any point during a clock cycle. If the specified timings are met, the exact sequence shown will occur; otherwise, an additional delay of one clock cycle may occur.

3) The 'C40 can accept an interrupt from the same source every two H1 clock cycles.

4) For edge-triggered interrupts, only timing number (1) applies.

Figure 14–20.  $\overline{\text{IACK}}$  Timing

14

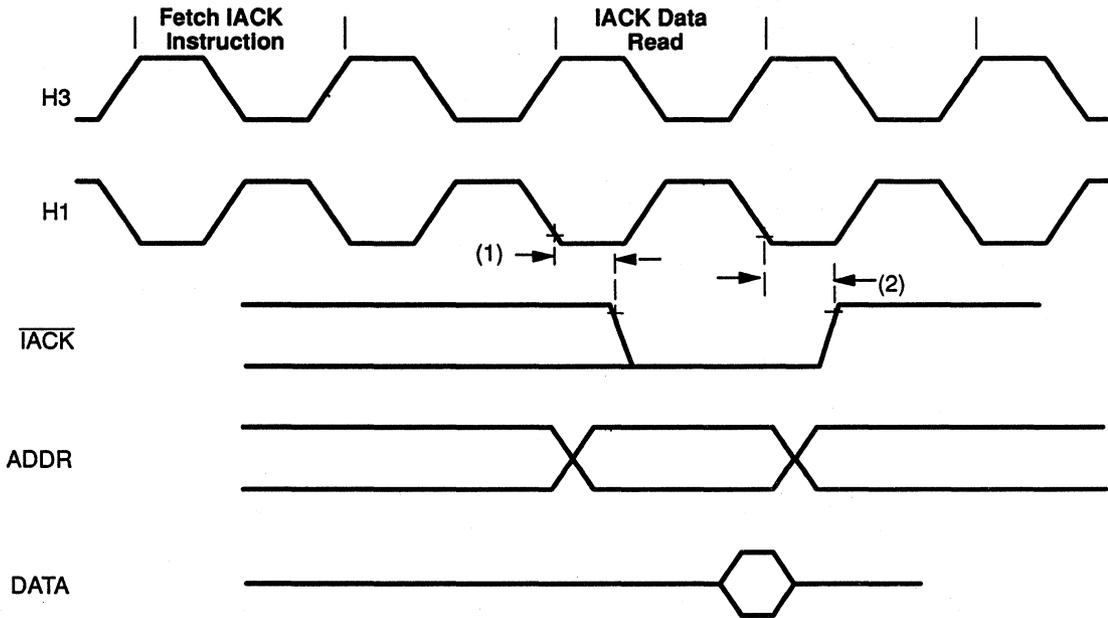
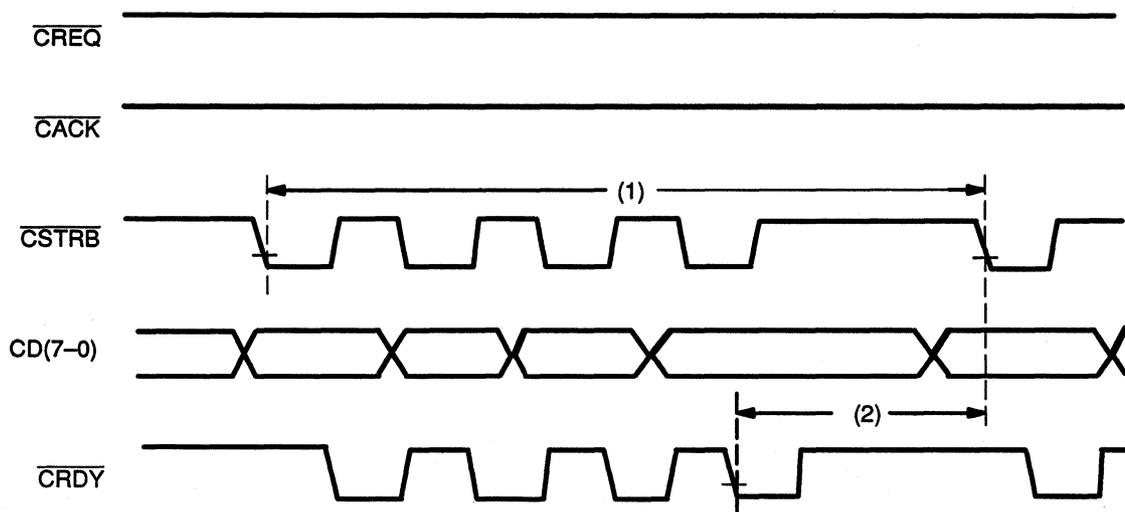


Table 14–19. Timing Parameters for  $\overline{\text{IACK}}$

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{d(H1H-IACKL)}$	H1 high to $\overline{\text{IACK}}$ low		7		7	ns
(2)	$t_{d(H1H-IACKH)}$	H1 high to $\overline{\text{IACK}}$ high during first cycle of IACK instruction data read		7		7	ns

**Note:** The  $\overline{\text{IACK}}$  output is active for the entire duration of the bus cycle and is therefore extended if the bus cycle utilizes wait states.

Figure 14–21. Communication-Port Word-Transfer Cycle Timing



**Note:** For correct operation during token exchange, the two communicating 'C40s must have CLKIN frequencies within a factor of 2 of each other (in other words, at most, one of the 'C40s can be twice as fast as the other).

Table 14–20. Communication-Port Word-Transfer Cycle Timing

No.	Name	Description	TMS320C40‡		TMS320C40-40‡		Unit
			Min†	Max†	Min†	Max†	
(1)	tWORD	Word transfer period (4 bytes = 1 word)	1.5P + 46	2.5P + 202	1.5P + 46	2.5P + 202	ns
(2)	t <sub>d(RL-SL)W</sub>	CRDY low to CSTRB low between back-to-back write cycles	1.5P + 7	2.5P + 28	1.5P + 7	2.5P + 28	ns

† P is the duration of the H1 clock period with a minimum value of 40 ns ( $P \geq 40$  ns).

‡ For these timing values, it is assumed that the 'C40 receiving data is ready to receive data.

Figure 14–22. Communication Port Byte Timing (Write and Read)

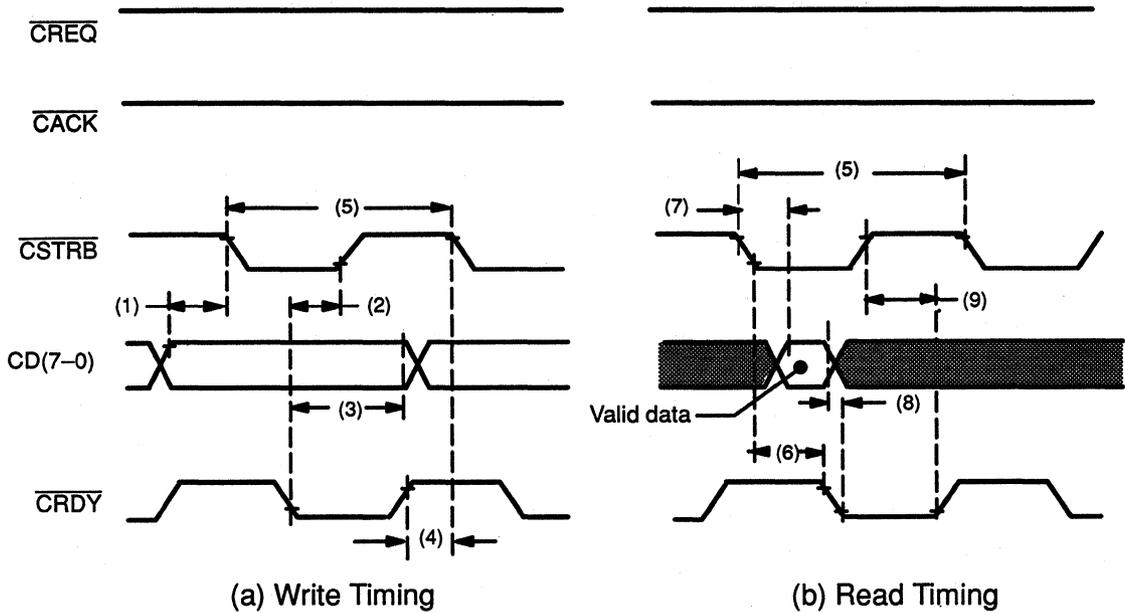
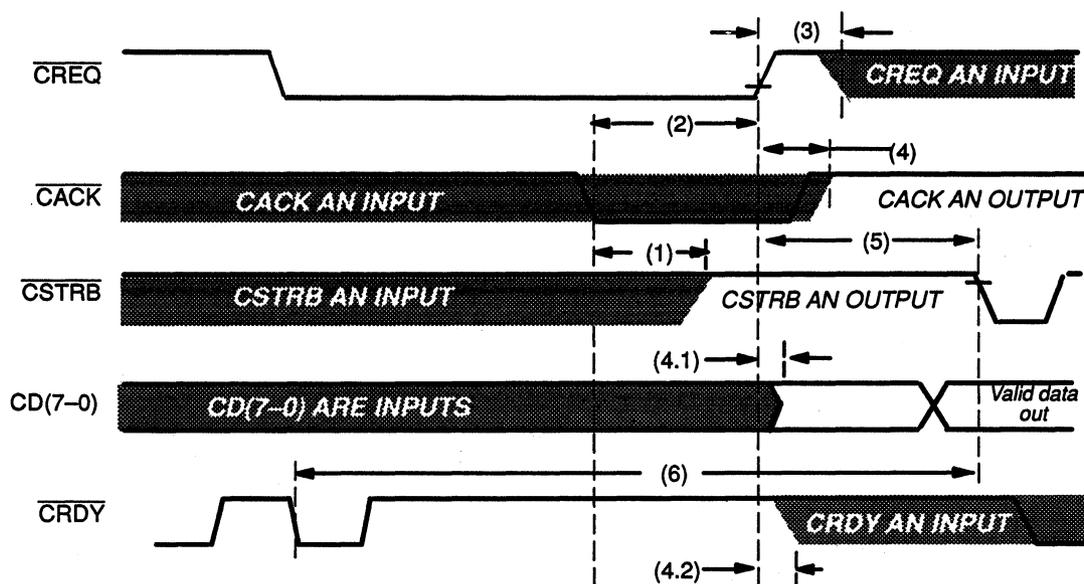


Table 14–21. Communication Port Byte Timing (Write and Read)

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{su}(CD)W$	Data valid before $\overline{CSTRB}$ (write)	2		2		ns
(2)	$t_{d}(RL-SH)W$	$\overline{CRDY}$ low to $\overline{CSTRB}$ high (write)	3	15	3	15	ns
(3)	$t_{h}(CD)W$	CD hold after $\overline{CRDY}$ low (write)	2		2		ns
(4)	$t_{d}(RH-SL)W$	$\overline{CRDY}$ high to $\overline{CSTRB}$ low for subsequent bytes (write)	3	15	3	15	ns
(5)	$t_{BYTE}$	Byte period	12	54	12	54	ns
(6)	$t_{d}(SL-RL)R$	$\overline{CSTRB}$ low to $\overline{CRDY}$ low (read)	3	12	3	12	ns
(7)	$t_{su}(CD)R$	CD valid after $\overline{CSTRB}$ (read)	0		0		ns
(8)	$t_{h}(CD)R$	CD held valid after $\overline{CRDY}$ low (read)	0		0		ns
(9)	$t_{d}(SH-RH)R$	$\overline{CSTRB}$ high to $\overline{CRDY}$ high (read)	3	12	3	12	ns

Figure 14-23. Communication Token Transfer Sequence From an Input to an Output Port



■ = When signal is an input (clear = when signal is an output).

**Note:** Before the token exchange,  $\overline{\text{CREQ}}$  and  $\overline{\text{CRDY}}$  are output signals asserted by the 'C40 that is receiving data.  $\overline{\text{CACK}}$ ,  $\overline{\text{CSTRB}}$ , and  $\text{CD}(7-0)$  are input signals asserted by the device sending data to the 'C40; these are asynchronous with respect to the H1 clock of the receiving 'C40. After token exchange,  $\overline{\text{CACK}}$ ,  $\overline{\text{CSTRB}}$ , and  $\text{CD}(7-0)$  become output signals, and  $\overline{\text{CREQ}}$  and  $\overline{\text{CRDY}}$  become inputs.

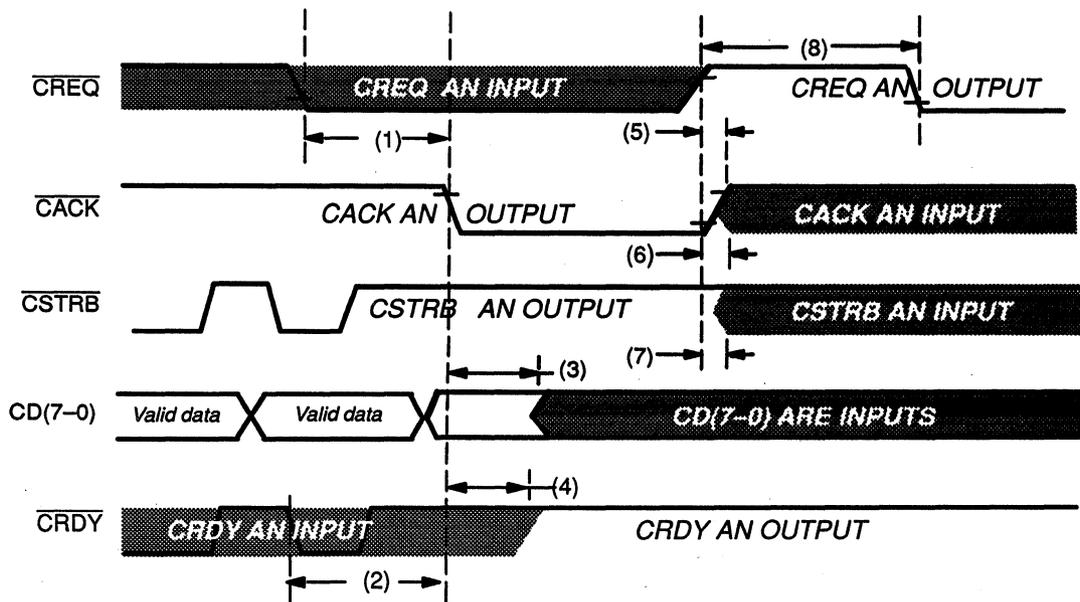
— Timing parameter table on next page —

Table 14–22. Communication Token Transfer Sequence From an Input to an Output Port  
(Figure 14–23)

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min†	Max†	Min†	Max†	
(1)†	$t_d(\text{AL-SO})\text{T}$	$\overline{\text{CACK}}$ low to $\overline{\text{CSTRB}}$ change from input to a high-level output	$0.5\text{P} + 6$	$1.5\text{P} + 22$	$0.5\text{P} + 6$	$1.5\text{P} + 22$	ns
(2)†	$t_d(\text{AL-RQH})\text{T}$	$\overline{\text{CACK}}$ low to start of $\overline{\text{CREQ}}$ going high for token request acknowledge	$\text{P} + 5$	$2\text{P} + 20$	$\text{P} + 5$	$2\text{P} + 20$	ns
(3)	$t_d(\text{RQH-RQI})\text{T}$	Start of $\overline{\text{CREQ}}$ going high to $\overline{\text{CREQ}}$ change from output to an input	$0.5\text{P} - 5$	$0.5\text{P} + 13$	$0.5\text{P} - 5$	$0.5\text{P} + 13$	ns
(4)	$t_d(\text{RQH-AO})\text{T}$	Start of $\overline{\text{CREQ}}$ going high to $\overline{\text{CACK}}$ change from an input to an output level high	$0.5\text{P} - 5$	$0.5\text{P} + 13$	$0.5\text{P} - 5$	$0.5\text{P} + 13$	ns
(4.1)	$t_d(\text{RQH-DO})\text{T}$	Start of $\overline{\text{CREQ}}$ going high to $\text{CD}(7-0)$ change from inputs driven to outputs driven	$0.5\text{P} - 5$	$0.5\text{P} + 13$	$0.5\text{P} - 5$	$0.5\text{P} + 13$	ns
(4.2)	$t_d(\text{RQH-RI})\text{T}$	Start of $\overline{\text{CREQ}}$ going high to $\text{CRDY}$ change from an output to an input	$0.5\text{P} - 5$	$0.5\text{P} + 13$	$0.5\text{P} - 5$	$0.5\text{P} + 13$	ns
(5)	$t_d(\text{RQH-SL})\text{T}$	Start of $\overline{\text{CREQ}}$ going high to $\overline{\text{CSTRB}}$ low for start of word transfer out	$1.5\text{P} - 8$	$1.5\text{P} + 9$	$0.5\text{P} - 8$	$1.5\text{P} + 9$	ns
(6)	$t_d(\text{RL-SL})\text{T}$	$\overline{\text{CRDY}}$ low at end of word input to $\overline{\text{CSTRB}}$ low for word output	$3.5\text{P} + 12$	$5.5\text{P} + 48$	$3.5\text{P} + 12$	$5.5\text{P} + 48$	ns

† These timing parameters result from synchronizer delays and are referenced from the falling edge of H1. The inputs (that cause the output-signal pins to change values) are sampled on H1 falling. The minimum delay occurs when the input condition occurs just before H1 falling, and the maximum delay occurs when the input condition occurs just after H1 falling.

Figure 14-24. Communication Token Transfer Sequence From an Output to an Input Port



14

 = When signal is an input (clear = when signal is an output).

**Note:** Before the token exchange,  $\overline{CACK}$ ,  $\overline{CSTRB}$ , and  $CD(7-0)$  are asserted by the 'C40 sending data.  $\overline{CREQ}$  and  $\overline{CRDY}$  are input signals asserted by the 'C40 receiving data and are asynchronous with respect to the H1 clock of the sending 'C40. After token exchange,  $\overline{CREQ}$  and  $\overline{CRDY}$  become outputs, and  $\overline{CSTRB}$ ,  $\overline{CACK}$ , and  $CD(7-0)$  become inputs.

— Timing parameter table on next page —

Table 14–23. Communication Token Transfer Sequence From an Output to an Input Port  
(Figure 14–24)

14

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)†	$t_{d(RQL-AL)T}$	$\overline{CREQ}$ low to start of $\overline{CACK}$ going low for token request acknowledge	P + 5	2P + 22	P + 5	2P + 22	ns
(2)†	$t_{d(RL-AL)T}$	$\overline{CRDY}$ low at end of word transfer out to start of $\overline{CACK}$ going low	P + 6	2P + 27	P + 6	2P + 27	ns
(3)	$t_{d(AL-CD)I}$	Start of $\overline{CACK}$ going low to CD(7–0) change from outputs to inputs	0.5P – 8	0.5P + 8	0.5P – 8	0.5P + 8	ns
(4)	$t_{d(AL-RO)T}$	Start of $\overline{CACK}$ going low to $\overline{CRDY}$ change from an input to output, high level	0.5P – 8	0.5P – 8	0.5P – 8	0.5P – 8	ns
(5)†	$t_{d(RQH-AQ)T}$	$\overline{CREQ}$ high to $\overline{CREQ}$ change from an input to output, high level	4	22	4	22	ns
(6)†	$t_{d(RQH-AI)T}$	$\overline{CREQ}$ high to $\overline{CACK}$ change from output to an input	4	22	4	22	ns
(7)†	$t_{d(RQH-SI)T}$	$\overline{CREQ}$ high to $\overline{CSTRB}$ change from output to an input	4	22	4	22	ns
(8)†	$t_{d(RQH-RQL)T}$	$\overline{CREQ}$ high to $\overline{CREQ}$ low for the next token request	P – 4	2P + 8	P – 4	2P + 8	ns

† These timing parameters result from synchronizer delays and are referenced from the falling edge of H1. The inputs (that cause the output-signal pins to change values) are sampled on H1 falling. The minimum delay occurs when the input condition occurs just before H1 falling, and the maximum delay occurs when the input condition occurs just after H1 falling.

Figure 14–25. Timer Pin Timings

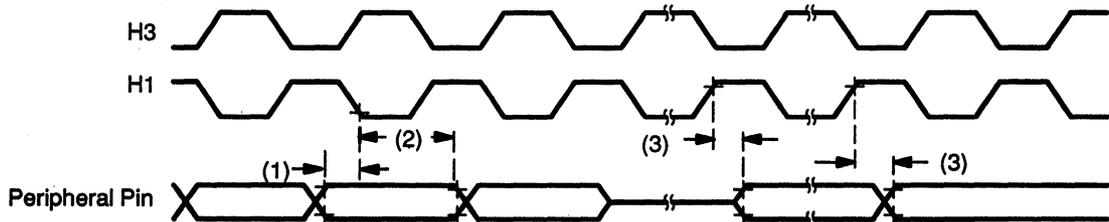


Table 14–24. Timing Parameters for Timer Pin

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{su}(TCLKH1L)$	TCLK setup before H1 low	9		10		ns
(2)	$t_h(TCLKH1L)$	TCLK hold after H1 low	0		0		ns
(3)	$t_d(TCLKH1H)$	TCLK valid after H1 high		7		7	ns

**Note:** Period and polarity of valid logic level are specified by contents of internal control registers.

Figure 14–26. JTAG Emulation Timings

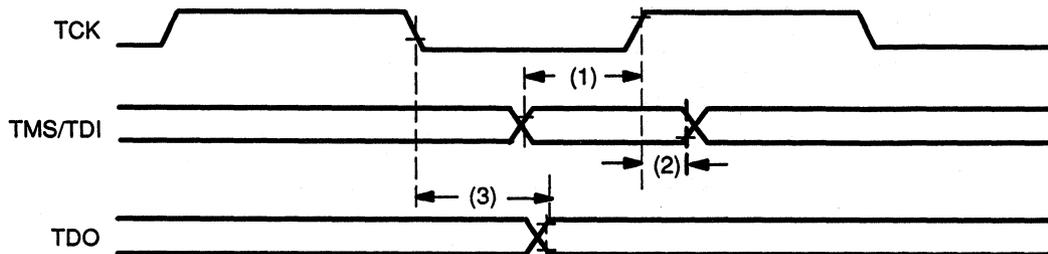


Table 14–25. Timing Parameters for JTAG Emulation

No.	Name	Description	TMS320C40		TMS320C40-40		Unit
			Min	Max	Min	Max	
(1)	$t_{su}(TMS-TCKH)$	TMS/TDI setup to TCK high	10		10		ns
(2)	$t_h(TMS/TDI)$	TMS/TDI hold from TCK high	5		5		ns
(3)	$t_d(TCKL-TDOV)$	TCK low to TDO valid	0	15	0	15	ns

## Appendix A

# TMS320C4x Sockets<sub>A</sub>

---

---

---

---

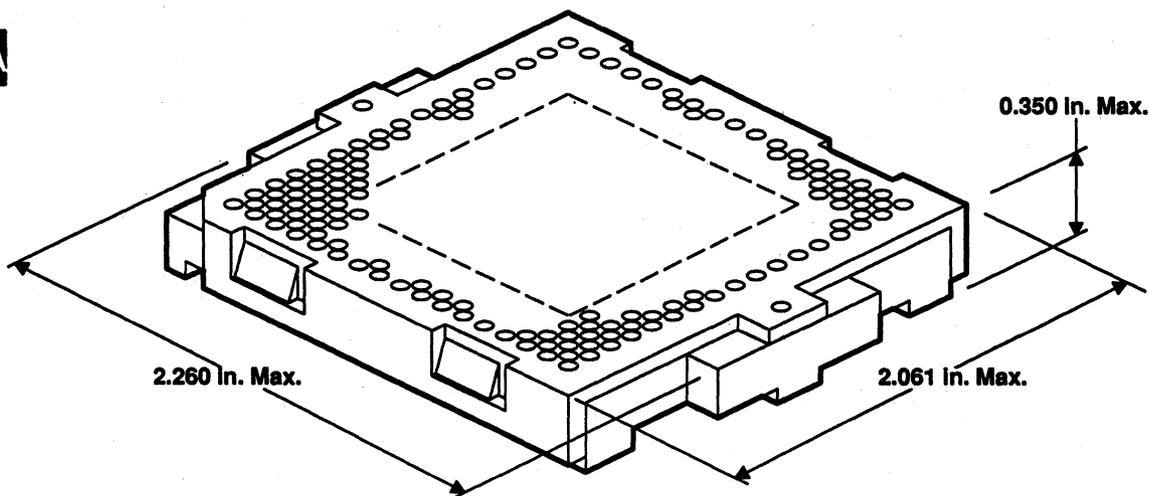
This appendix describes sockets available to accept the TMS320C4x pin grid array (PGA). Both sockets covered in this appendix feature zero insertion force (ZIF):

- a tool-activated ZIF socket (TAZ)
- a handle-activated ZIF socket (HAZ).

The sockets described herein are manufactured by AMP Incorporated®.

## A.1 Tool-Activated ZIF PGA Socket (TAZ)

Figure A-1. Tool-Activated ZIF Socket



This socket requires AMP™ actuator tool: 354234-1

### Description:

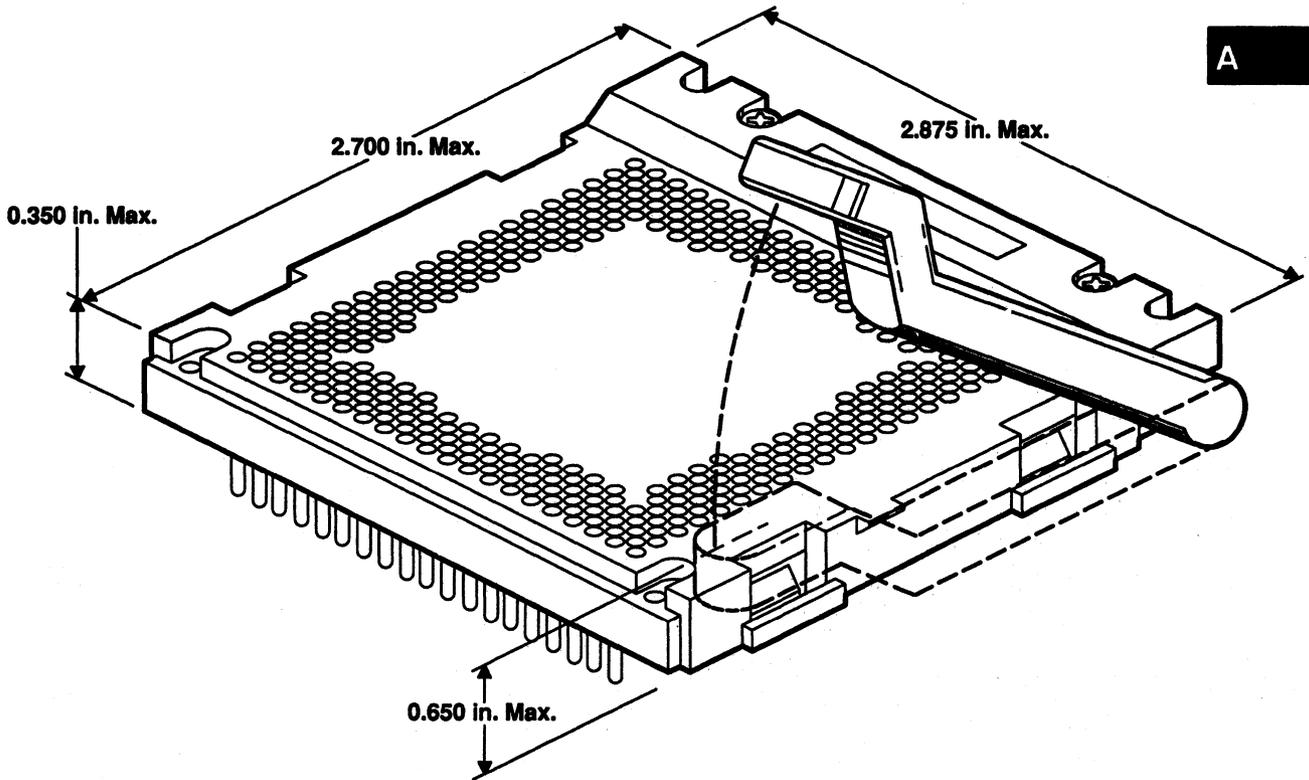
- AMP part number: 382533-9
- pin positions: 325
- soldertail length: 0.170 in. for PC boards 0.125 in. thick (other tail lengths available)

### Features:

- slightly larger than PGA device
- easy package loading because of large funnel entry
- zero insertion force
- contact wiping action during insertion ensures clean contact points
- spring-loaded cover ensures proper loading
- can be used with robotic insertion and removal
- its horizontal socket forces (vs. vertical) prevent damage to device

## A.2 Handle-Activated ZIF PGA Socket (HAZ)

Figure A-2. Handle-Activated ZIF Socket



### Description:

- ❑ AMP part number: 382320-9
- ❑ pin positions: 325
- ❑ solder tail length: 0.170 in. for pc boards 0.125 in. thick (other tail lengths available)
- ❑ Dimensions:
  - Height: 0.350 inch maximum to device plane and 0.650 inch to top of handle in closed position
  - Width: 2.700 by 2.875 inches maximum

**Features:**

- can be used for test and burn-in
- spring contacts are normally closed
- easy package loading because of large funnel entry
- zero insertion force
- contact wiping action during socket closing ensures clean contact points
- operating temperature is 160° C (burn-in capability)

**A**

## XDS510 Design Considerations

The information in this document is assist you in meeting the design requirements of the XDS510 emulator. This information supports XDS510 Cable no. 2563988-001, rev B.

The TMS320C4x family supports emulation through a dedicated emulation port. The emulation port is a superset of the IEEE 1149.1 (JTAG) standard and can be accessed by the XDS510 emulator. For details on the JTAG protocol, refer to the IEEE 1149.1 specification.

This appendix contains the following sections:

<b>Section</b>	<b>Page</b>
B.1 Header Signals .....	B-2
B.2 Bus Protocol .....	B-3
B.3 Cable Pod .....	B-4
B.4 Test Clock Generated in Test System .....	B-7
B.5 Processor Configuration .....	B-8
B.6 Emulation Timing Calculations .....	B-11

## B.1 Header and Header Signals

To perform emulation with the XDS510, your target system must have a 14-pin header (two 7-pin rows) with connections as shown in Figure B-1. Table B-1 describes the emulation signals.

Although you can use other headers, recommended parts include:

- Straight header, unshrouded      DuPont Electronics™ part number 67996-114
- Right-angle header, unshrouded      DuPont Electronics™ part number 68405-114

Figure B-1. 14-pin Header Signals and Header Dimensions

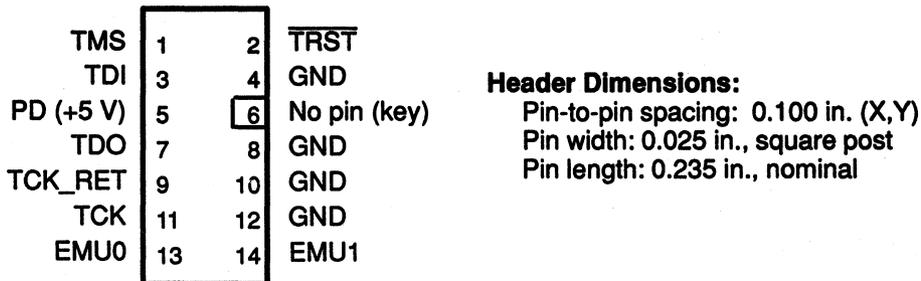


Table B-1. 14-Pin Header Signal Description

XDS510 Signal	†XDS510 State	†Target State	Description
TMS	O	I	JTAG test mode select
TDI	O	I	JTAG test data input
TDO	I	O	JTAG test data output†
TCK	O	I	JTAG test clock. TCK is a 10-MHz clock source from the emulation cable pod. This signal can be used to drive the system test clock.
TRST	O	I	JTAG test reset
EMU0	I	I/O	Emulation pin 0
EMU1	I	I/O	Emulation pin 1
PD	I	O	Presence detect. Indicates that the emulation cable is connected and that the target is powered up. PD should be tied to +5 volts in the target system.
TCK_RET	I	O	JTAG test clock return. Test clock input to the XDS510 emulator. May be a buffered or unbuffered version of TCK.

† I = input; O = output

## B.2 Bus Protocol

The IEEE 1149.1 specification covers the requirements for JTAG bus slave devices (such as the TMS320C4x family) and provides certain rules. Those rules are summarized as follows:

- ❑ The TMS/TDI inputs are sampled on the rising edge of the TCK signal of the device.
- ❑ The TDO output is clocked from the falling edge of the TCK signal of the device.

**B**

When JTAG devices are daisy-chained together, the TDO of one device has approximately a half TCK cycle set up to the next device's TDI signal. This type of timing scheme minimizes race conditions that would occur if both TDO and TDI were timed from the same TCK edge. The penalty for this timing scheme is a reduced TCK frequency.

The IEEE 1149.1 specification does not provide rules for JTAG bus master (XDS510) devices. Instead, it states that it expects a bus master to provide bus slave compatible timings. The XDS510 provides timings that meet the bus slave rules and also provides an optional timing mode that allows you to run the emulation at a much higher frequency for improved performance by avoiding the timing penalty described herein.

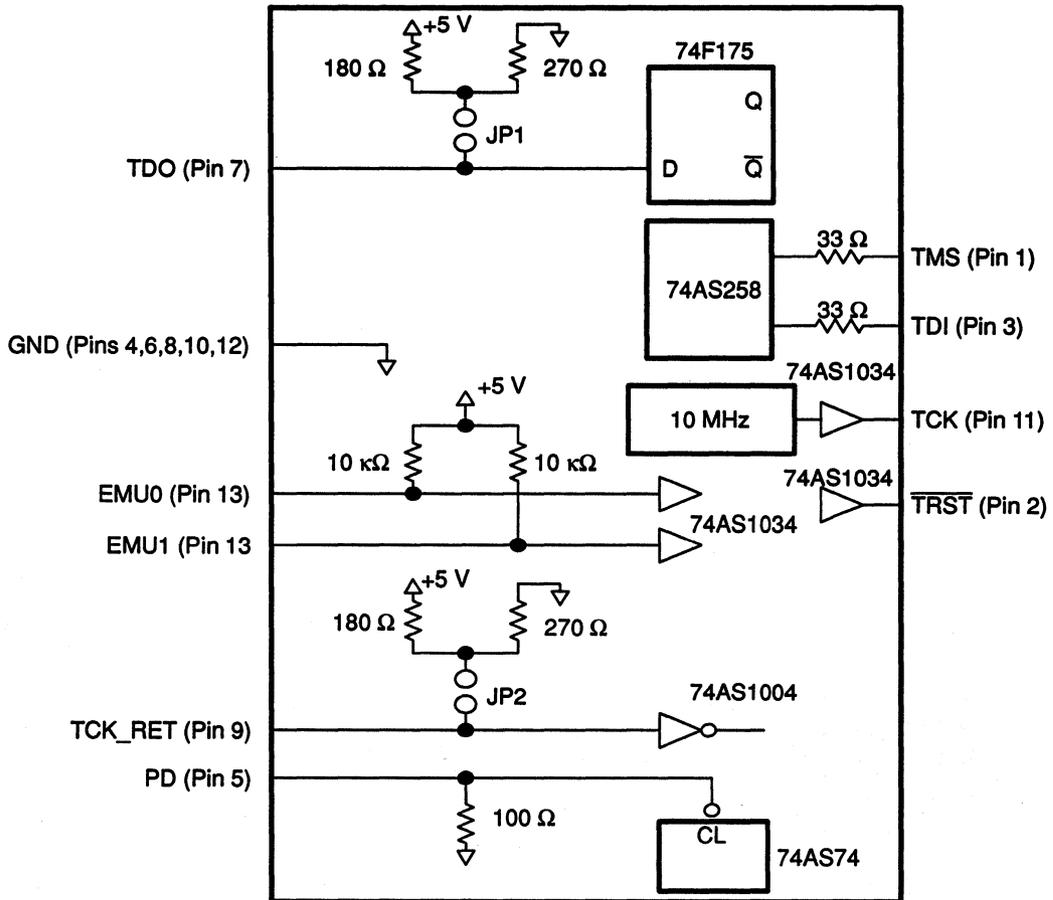
### B.3 Cable Pod

Figure B–2 shows a portion of the XDS510 emulator cable pod. These are the functional features of the emulator pod:

- ❑ Signals TDO and TCK\_RET can be parallel-terminated inside the pod if required by the application. The default is that these signals are not terminated.
- ❑ Signal TCK is driven with a 74AS1034 device. Because of the high current drive (48 mA  $I_{OL}/I_{OH}$ ), this signal can be parallel terminated. If TCK is tied to TCK\_RET, then you can use the parallel terminator in the pod.
- ❑ Signals TMS and TDI can be generated from the falling edge of TCK\_RET, according to the IEEE 1149.1 bus slave device timing rules. They can also be driven from the rising edge of TCK\_RET, which allows a higher TCK\_RET frequency. The default is to match the IEEE 1149.1 slave device timing rules. This is an emulator software option that can be selected when the emulator is invoked. In general, single-processor applications can benefit from the higher clock frequency. However, in multiprocessing applications, you may wish to use the IEEE 1149.1 bus slave timing mode to minimize emulation system timing constraints.
- ❑ Signals TMS and TDI are series terminated to reduce signal reflections.
- ❑ A 10-MHz test clock source is provided. You may also provide your own test clock for greater flexibility.

B

Figure B-2. Emulator Pod Interface



B

Figure B-3 and Table B-2 show the signal timings for the XDS510. Timing parameters are calculated from standard data sheet parts used in the cable pod. These timings are for reference only. Texas Instruments does not test or guarantee these timings.

The emulator pod uses TCK\_RET as its clock source for internal synchronization. TCK is provided as an optional target system test clock source.

Figure B-3. Emulator Pod Timings

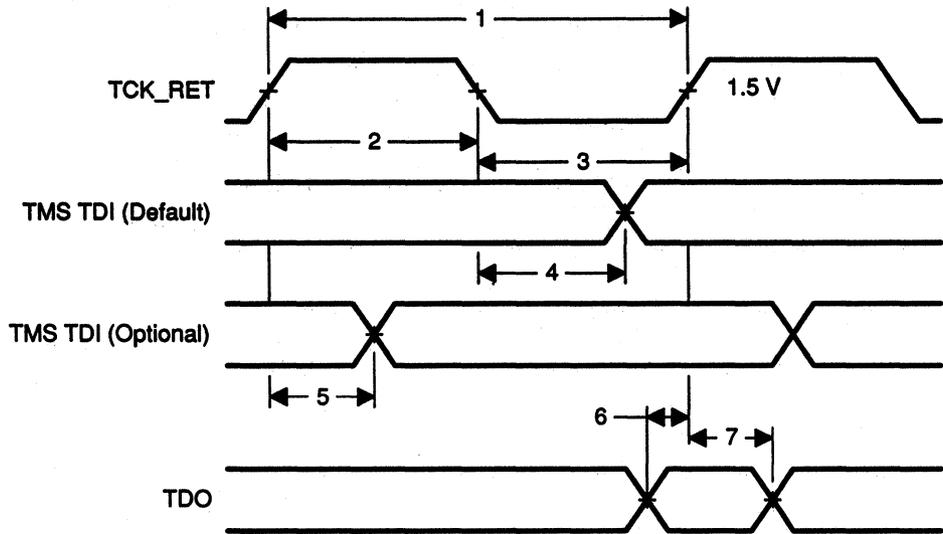


Table B-2. Emulator Pod Timing Parameters

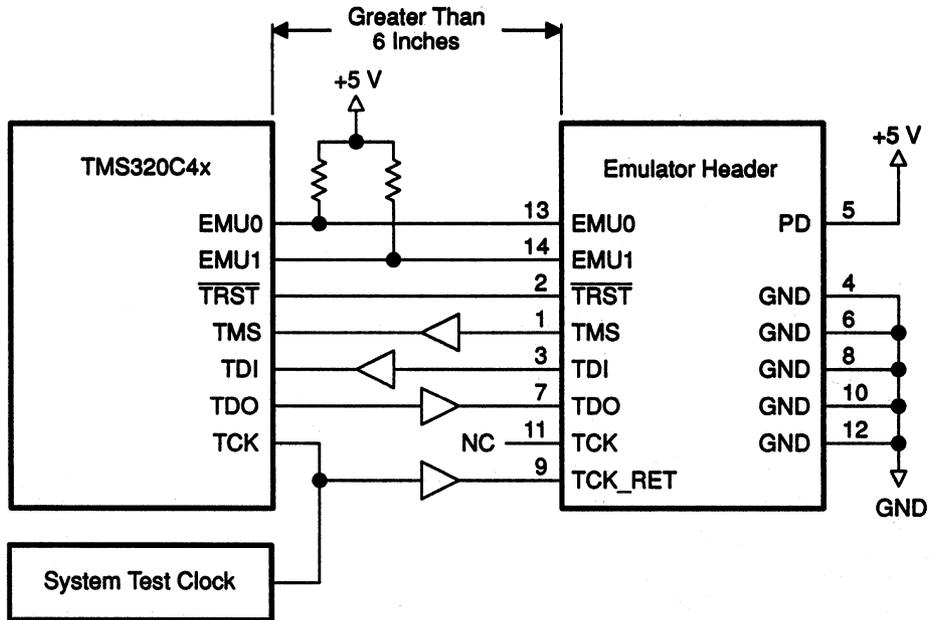
No.	Reference	Description	Min	Max	Unit
1	$t_{TCKmin}$ $t_{TCKmax}$	TCK_RET period	35	200	ns
2	$t_{TCKhighmin}$	TCK_RET high pulse duration	15		ns
3	$t_{TCKlowmin}$	TCK_RET low pulse duration	15		ns
4	$t_{d(XTMXmin)}$ $t_{d(XTMXmax)}$	TMS/TDI valid from TCK_RET low (default timing)	6	20	ns
5	$t_{d(XTMSmin)}$ $t_{d(XTMSmax)}$	TMS/TDI valid from TCK_RET high (optional timing)	7	24	ns
6	$t_{su(XTDOmin)}$	TDO setup time to TCK_RET high	3		ns
7	$t_{hd(XTDOmin)}$	TDO hold time from TCK_RET high	12		ns

It is extremely important to provide high-quality signals between the emulator and the target processor. If the distance between the emulation header and the processor is greater than 6 inches, the emulation signals should be buffered. Sections B.4 and B.5 illustrate typical connections between the target processor and the emulation header.

## B.4 Test Clock Generated in Target System

Figure 4 shows an application with the system test clock generated in the target system. In this application the TCK signal is left unconnected.

Figure B-4. Target-System Generated Test Clock



B

There are two benefits to having the target system generate the test clock:

- 1) You can set the test clock frequency to match your system requirements. The emulator provides only a single 10-MHz test clock.
- 2) You may have other devices in your system that require a test clock when the emulator is not connected.

## B.5 Multiprocessor Configuration

Figure B-5. Multiprocessor Connections

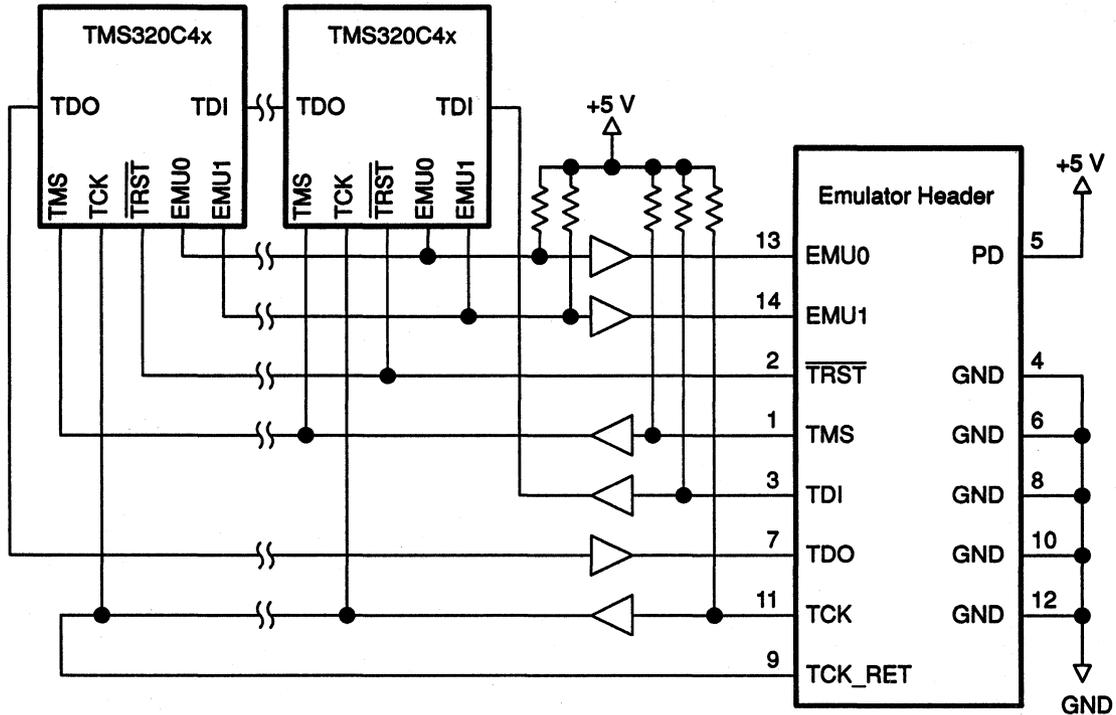
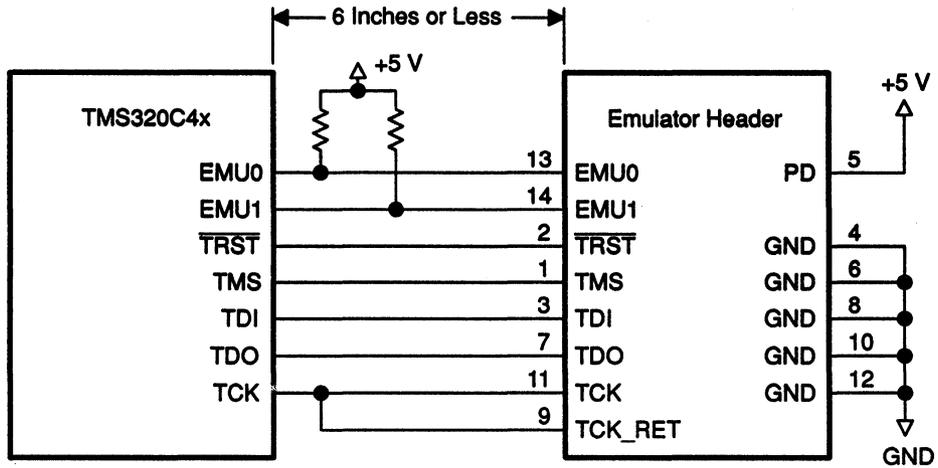


Figure B-5 shows a typical multiprocessor configuration. This is a daisy-chained configuration (TDO-TDI daisy-chained) that meets the minimum requirements of the IEEE 1149.1 specification. The emulation signals in this example are buffered to isolate the processors from the emulator and provide an adequate signal drive for the target system. One of the benefits of a JTAG test interface is that you can generally slow down the test clock to eliminate timing problems. Several key points to multiprocessor support are as follows:

- ❑ The processor TMS, TDI, TDO, and TCK should be buffered through the same physical package to better control timing skew.
- ❑ The input buffers for TMS, TDI, and TCK should have pullups to 5 volts. This will hold these signals at a known value when the emulator is not connected. A pullup of 4.7 k $\Omega$  or greater is suggested.
- ❑ Buffering EMU0 and EMU1 is optional but highly recommended to provide isolation. These are not critical signals and do not need to be buffered through the same physical package as TMS, TCK, TDI, and TDO. Unbuffered and buffered signals are shown in Figure B-6 and Figure B-7.

**No signal buffering.** In this situation, the distance between the header and the processor should be no more than 6 inches.

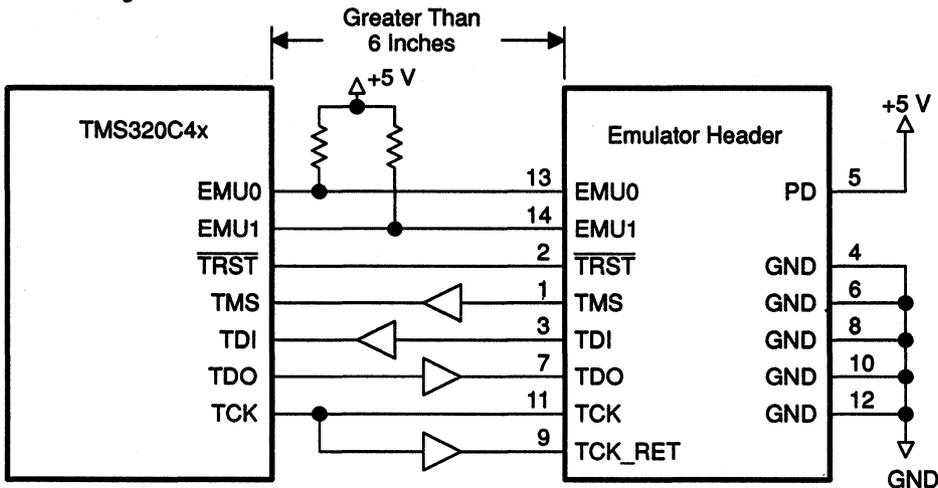
Figure B-6. Unbuffered Signals



**B**

**Emulation signals buffered.** The distance between the emulation header and the processor is greater than 6 inches. The emulation signals — TMS, TDI, TDO, and TCK\_RET — are buffered through the same package.

Figure B-7. Buffered Signals



- ❑ The EMU0 and EMU1 signals must have pullups to 5 volts. The pullup resistor value should be chosen to provide a signal rise-time less than 10  $\mu$ s. A 4.7 k $\Omega$  resistor is suggested for most applications. EMU0 – 1

are I/O pins on the 'C4x; however, they are only inputs to the XDS510. In general, these pins are used in multiprocessor systems to provide global run/stop operations.

- It is extremely important to provide high quality signals, especially on the processor TCK and the emulator TCK\_RET signal. In some cases, this may require you to provide special PWB trace routing and use termination resistors to match the trace impedance. The emulator pod does provide optional internal parallel terminators on the TCK\_RET, and TDO. TMS and TDI provide fixed series termination.

**B**

## B.6 Emulation Timing Calculations

Following are a few examples on how to calculate the emulation timings in your system. For actual target timing parameters, see the appropriate device data sheets.

<b>Assumptions:</b>	$t_{su}(TTMS)$	Target TMS/TDI setup to TCK high	10 ns
	$t_h(TTMS)$	Target TMS/TDI hold from TCK high	5 ns
	$t_d(TTDO)$	Target TDO delay from TCK low	15 ns
	$t_d(bufmax)$	Target buffer delay maximum	10 ns
	$t_d(bufmin)$	Target buffer delay minimum	1 ns
	$t(bufskew)$	Target buffer skew between two devices in the same package:	
		$[t_d(bufmax) - t_d(bufmin)] \times 0.15$	1.35 ns
	$t_{tckfactor}$	Assume a 40/60 duty cycle clock	0.4

Given in Table B-2 (page B-6):

$t_d(XTMSmax)$	XDS510 TMS/TDI delay from TCK_RET low, maximum	20 ns
$t_d(XTMX)$	min XDS510 TMS/TDI delay from TCK_RET low, minimum	6 ns
$t_d(XTMSmax)$	XDS510 TMS/TDI delay from TCK_RET high, max	24 ns
$t_d(XTMXmin)$	XDS510 TMS/TDI delay from TCK_RET high, minimum	7 ns
$t_{su}(XTDOmin)$	TDO setup time to XDS510 TCK_RET high	3 ns

There are two key timing paths to consider in the emulation design: the TCK\_RET/TMS/TDI ( $t_{prdtck\_TMS}$ ) path, and the TCK\_RET/TDO ( $t_{prdtck\_TDO}$ ) path.

In each case, the worst case path delay is calculated to determine the maximum system test clock frequency.

**Case 1:** Single processor, direct connection, TMS/TDI timed from TCK\_RET low (default timing).

$$\begin{aligned} t_{\text{prdtck\_TMS}} &= [t_d(\text{XTMSmax}) + t_{\text{su}}(\text{TTMS})] / t_{\text{tckfactor}} \\ &= (20 \text{ ns} + 10 \text{ ns}) / 0.4 \\ &= 75 \text{ ns} \quad (13.3 \text{ MHz}) \end{aligned}$$

$$\begin{aligned} t_{\text{prdtck\_TDO}} &= [t_d(\text{TTDO}) + t_{\text{su}}(\text{XTDOmin})] / t_{\text{tckfactor}} \\ &= (15 \text{ ns} + 3 \text{ ns}) / 0.4 \\ &= 45 \text{ ns} \quad (22.2 \text{ MHz}) \end{aligned}$$

In this case, the TCK/TMS path is the limiting factor.

**Case 2:** Single processor, direct connection, TMS/TDI timed from TCK\_RET high (optional timing).

$$\begin{aligned} t_{\text{prdtck\_TMS}} &= t_d(\text{XTMSmax}) + t_{\text{su}}(\text{TTMS}) \\ &= (24 \text{ ns} + 10 \text{ ns}) \\ &= 34 \text{ ns} \quad (29.4 \text{ MHz}) \end{aligned}$$

$$\begin{aligned} t_{\text{prdtck\_TDO}} &= [t_d(\text{TTDO}) + t_{\text{su}}(\text{XTDOmin})] / t_{\text{tckfactor}} \\ &= (15 \text{ ns} + 3 \text{ ns}) / 0.4 \\ &= 45 \text{ ns} \quad (22.2 \text{ MHz}) \end{aligned}$$

In this case, the TCK/TDO path is the limiting factor. One other thing to consider in this case is the TMS/TDI hold time. The minimum hold time for the XDS510 cable pod is 7 ns, which meets the 5-ns hold time of the target device.

**Case 3:** Single/multiple processor, TMS/TDI buffered input; TCK\_RET/TDO buffered output, TMS/TDI timed from TCK\_RET high (optional timing).

$$\begin{aligned} t_{\text{prdtck\_TMS}} &= t_d(\text{XTMSmax}) + t_{\text{su}}(\text{TTMS}) + 2 t_d(\text{bufmax}) \\ &= 24 \text{ ns} + 10 \text{ ns} + 2 (10) \\ &= 54 \text{ ns} \quad (18.5 \text{ MHz}) \end{aligned}$$

$$\begin{aligned} t_{\text{prdtck\_TDO}} &= \frac{t_d(\text{TTDO}) + t_{\text{su}}(\text{XTDOmin}) + t_{\text{bufskew}}}{t_{\text{tckfactor}}} \\ &= (15 \text{ ns} + 3 \text{ ns} + 1.35 \text{ ns}) / 0.4 \\ &= 58.4 \text{ ns} \quad (20.7 \text{ MHz}) \end{aligned}$$

In this case, the TCK/TMS path is the limiting factor. The hold time on TMS/TDI is also reduced by the buffer skew (1.35 ns) but still meets the minimum device hold time.

**Case 4:** Single/multiprocessor, TMS/TDI/TCK buffered input; TDO buffered output, TMS/TDI timed from TCK\_RET low (default timing).

$$\begin{aligned} t_{\text{prdtck\_TMS}} &= \frac{t_d(\text{XTMSmax}) + t_{\text{su}}(\text{TTMS}) + t_{\text{bufskew}}}{t_{\text{tckfactor}}} \\ &= (24 \text{ ns} + 10 \text{ ns} + 1.35 \text{ ns}) / 0.4 \\ &= 88.4 \text{ ns (11.3 MHz)} \end{aligned}$$

$$\begin{aligned} t_{\text{prdtck\_TDO}} &= \frac{t_d(\text{TTDO}) + t_{\text{su}}(\text{XTDOmin}) + t_d(\text{bufmax})}{t_{\text{tckfactor}}} \\ &= (15 \text{ ns} + 3 \text{ ns} + 10 \text{ ns}) / 0.4 \\ &= 70 \text{ ns (14.3 MHz)} \end{aligned}$$

In this case, the TCK/TMS path is the limiting factor.

In a multiprocessor application, it is necessary to ensure that the EUM0–1 lines can go from a logic low level to a logic high level in less than 10  $\mu\text{s}$ . This can be calculated as follows (remember that  $t = 5 \text{ RC}$ ):

$$\begin{aligned} t_{\text{rise}} &= 5(R_{\text{pullup}} \times N_{\text{devices}} \times C_{\text{load\_per\_device}}) \\ &= 5(4.7 \text{ k}\Omega \times 16 \times 15 \text{ pF}) \\ &= 5.64 \mu\text{s} \end{aligned}$$

B



**B**

# Index

Note: Primary sources are in **boldface**.

## A

- A-law compression, expansion, 12-46
- adaptive filters, 12-58
- ADDC instruction, 12-41
- addition, floating point, 4-20
- address buses
  - address reach (space), 1-5
  - external, 2-27
  - general, 1-5
- address range
  - LSTRB0,1-field specified, 7-11
  - STRB0,1-field specified, 7-10
- addressing modes
  - conditional branch, 2-15, 5-24
  - general, 2-15, 5-19
  - parallel, 2-15, 5-23
  - three operand, 2-15, 5-20
- addressing types, 5-2
  - direct addressing, 5-4
  - immediate, 5-17
  - indirect addressing, 5-5—5-16
  - PC relative, 5-17
  - register, 5-3
- ALU, 2-4
- analysis module
  - general, 1-5
  - registers, 3-21
- ANSI C compiler, 1-9
- applications
  - hardware, 13-1
  - list, 1-11

- software, 12-1
- ARAU (auxiliary register arithmetic unit), 2-6
- arithmetic logic unit (ALU), 2-4
- arithmetic operations, 12-28
- assembly language instructions, 11-1
  - categories, 11-3—11-9
    - interlocked operation*, 11-7
    - load and store*, 11-3
    - parallel operation*, 11-8
    - program control*, 11-6
    - three-operand*, 11-6
    - two-operand*, 11-4
  - condition codes, flags, 11-10
  - example instruction, 11-18
  - register syntax, 11-17
  - summary, 2-16—2-25, 11-3—11-9
  - symbols used to define, 11-14—11-17
  - syntax options, 11-15—11-17
- auxiliary register arithmetic units (ARAUs), 2-6
- auxiliary registers (AR0–7), 2-6, 3-5

## B

- Bcond instruction, 12-11
- BcondAF, BcondAT instructions, 6-8
- benchmarks, FFT timing, 12-88
- biquads, 12-53
- bit manipulation, 12-28
- bit-reversed addressing, 5-30, 12-31
  - modify example, 5-16
- block diagrams
  - communication port control register, 8-10
  - communication ports, 8-4, 8-5

block diagrams (*Continued*)

- CPU, 2-5
  - memory organization, 2-11
  - peripheral modules, 2-28
  - timers, 9-45
  - TMS320C40, 2-2
- block moves, 12-29
- block repeat, 6-2
- example, 12-24
  - registers (RS, RE), **3-14**, 12-26
- block repeat registers (RS, RE), **3-14**, 12-26
- block size (BK) register, **3-5**, 12-52
- boot loader, 13-5
- communication port, 13-8
  - external memory, 13-8
  - source program, 13-14
- branches, 6-7, 6-9, 12-22
- delayed, 6-7
- BRD instruction, 12-95
- bus operation
- arbitration, 13-48, 13-70
  - external, 2-27
  - internal, 2-26
- busy-waiting example, 6-15
- byte manipulation, 12-30

**C**

- cache, optimization of code, 12-96
- cache memory, 2-10
- algorithm, 3-27
  - architecture, 2-10, 3-25
  - control bits, 3-29
  - general, 1-6
  - hit, 3-27
  - instruction cache, 3-25
  - miss, 3-27
  - optimization of code, 12-96
  - size, 1-6
- CALL instruction, 6-9, 12-13
- CALLcond instruction, 6-9, 12-27
- calls, 6-9
- example code, 12-9
  - zero overhead, 12-11
- central processing unit, 2-4
- channel control register. *See* DMA channel control register
- circular addressing, 5-25
- circular modify example, 5-12
- communication port arbitration unit (PAU), 8-5
- communication port control register, 8-5, **8-10**
- field descriptions, 8-10
  - memory map, 3-23
- communication ports
- applications, 12-98
  - architecture, 2-29
  - benefits, 1-7
  - block diagram, 8-4, 8-5
  - control register. *See* communication port control register
  - features, 8-3
  - general, 1-4
  - memory map, 3-23, 8-8
  - port arbitration unit (PAU), 8-12
  - synchronizer timing, 8-32
  - throughput, 1-4, 1-7
  - timing, **8-18**, 8-32, 14-31
- companding, 12-46
- compiler, 1-9
- computed GOTOs, 12-27
- condition codes, flags, 11-12
- conditional delayed branches, 6-7
- conditional-branch addressing modes, 2-15, 5-24
- context switching, 12-15
- conversion of format
- 2s complement floating-point to IEEE, 4-13
  - extended-prec floating-point to single-prec floating-point, 4-10
  - floating point to integer, 4-28
  - IEEE single prec. std. 754, 4-11
  - IEEE std. 754, 4-11
  - IEEE to 2s complement floating-point, 4-12
  - IEEE to/from 'C40, 12-42
  - integer to floating point, 4-30

conversion of format (*Continued*)  
 short floating point to extended-prec.  
   floating point, 4-9  
 short floating point to single-prec. floating  
   point, 4-9  
 single-prec. floating-point to extended-  
   prec floating-point, 4-10  
 single-prec. 2s compl. floating-point, 4-11  
 counter example, 6-15  
 counter register (timer), 9-50  
   *See also* timers  
 CPU  
   architecture, 2-4  
   buses, 2-26  
   general, 1-4  
   instruction cycle times, 1-4  
   primary register file, 3-3  
   throughput, 1-4  
 CPU internal interrupt enable register (IIE),  
   2-8, **3-10**  
 CPU primary register file, 3-3  
 CPU registers, 3-3  
   auxiliary (AR0–AR7), 2-6, **3-5**  
   block repeat (RS, RE), **3-14**, 12-26  
   block size (BK), 2-7, **3-5**  
   data page pointer (DP), 2-7, **3-5**, 5-4  
   DMA interrupt enable (DIE), 2-8, **3-8**  
     *bit descriptions*, 3-9  
   extended precision (R0–R11), 2-6, **3-4**  
   IIOF flag register (IIF), 2-8, **3-12**  
   index (IR1, IR0), 2-7, **3-5**  
   internal interrupt enable (IIE), 2-8, **3-10**  
     *bit descriptions*, 3-11  
   list of, 2-7, 3-3  
   primary register file, 3-3  
   program counter (PC), 2-9, 2-26, **3-14**  
   repeat count (RC), 2-8, **3-14**, 6-2, **12-26**  
   repeat end address (RE), 3-14, 6-2  
   repeat start address (RS), 3-14, 6-2  
   reserved bits, 3-14  
   stack pointer (SP), 2-8, **3-5**  
     *application*, 12-13

status register (ST), 2-8, **3-5**, 11-11  
   *bit descriptions*, 3-6

**D**  
 data buses  
   external, 2-27  
   general, 1-5  
   transfer rate, 1-5  
 data page pointer (DP), 2-7, **3-5**, 5-4  
 delayed branches, 6-7  
   example, 12-23  
   incorrectly placed, 6-6, 6-7  
   optimization use, 12-95  
 dequeues (stack), 5-33  
 development tools. *See* software develop-  
   ment tools  
 dimensions ('C40), 14-11  
 direct addressing, 5-4  
 direct memory access. *See* DMA coproces-  
   sors  
 disabled interrupts by branch, 6-8  
 displacements, 5-5—5-16  
 division  
   floating point, 12-33  
   integer, 12-33  
 DMA. *See* DMA coprocessors  
 DMA channel control register, 9-7  
   AUTOINIT STATIC bit, 12-105  
   PRI bits, 9-14  
   bit definitions, 9-8  
   field descriptions, 9-8  
   START bits, **12-103**  
   STATUS bits, 9-16  
   SYNC MODE bits, **12-103**  
   TRANSFER MODE field, 9-28  
 DMA coprocessors  
   architecture, 2-29  
   autoinitialization, 9-31, 12-105  
     *example*, 12-107  
   benefits, 1-8  
   buses, 2-26  
   channel address register, 9-16

DMA coprocessors (*Continued*)

- channel control register
    - AUTOINIT STATIC bit*, 12-105
    - PRI bits*, 9-14
    - START bits*, 9-15
    - STATUS bits*, 9-16
    - SYNC MODE bits*, 9-15
    - TRANSFER bits*, 9-14
  - channel register map, 3-24
  - channel synchronization, 9-41—9-46
  - features, 9-2, 9-3
  - functional description, 9-3
  - general, 1-4, 2-29
  - index register, 9-16
  - interrupts, 9-40, 12-102
    - example of use*, 12-107
  - link–pointer register, 9-19, 9-38
    - example*, 12-105
  - memory mapped registers, 9-4
  - operation examples, 12-101
  - priorities, 9-22
  - priority wheel, 9-24
  - registers, 9-5, 9-7
  - split mode example, 12-104
  - START bits, 12-103
  - SYNC MODE bits, 12-103
  - synchronization of channels, 9-41—9-46
  - throughput, 1-8
  - transfer count register, 9-18
  - transfer description, 9-5, 12-103
  - TRANSFER MODE field, 9-28
  - unified and split modes, 9-20
- DMA interrupt enable register (DIE), 2-8, 3-8
- double precision, fixed point, 12-41

**E**

- edge-triggered interrupts, 6-23
- electrical characteristics, 14-13
- electrical specifications, 14-12
- emulator (XDS510), 1-9
- event counters. *See* timers
- expansion register file, 2-9, 3-15
  - interrupt vector table (IVT), 3-16, 6-26
    - application*, 12-19
  - trap vector table (TVT), 3-15

- extended precision number, floating-point format, 4-8
- extended precision registers, 2-6, 3-4, 12-41
  - floating point format, 3-4
  - integer format, 3-4
  - saving (example), 12-13
- external buses (global, local), wait states, 7-15
- external interrupts, 2-27

**F**

- fast Fourier transforms, 12-31, 12-63
  - DIF (decimation in frequency), 12-64, 12-65, 12-70
  - DIT (decimation in time), 12-64, 12-78
  - timing benchmarks, 12-88
  - twiddle factors, 12-68
- features (of TMS320C40), 1-4
- FFT. *See* fast Fourier transforms
- filters
  - adaptive, 12-58
  - FIR, 12-51, 12-59
  - IIR, 12-53, 12-54
  - lattice, 12-88
- FIR filters, 12-51, 12-59
- FIX instruction, 4-28, 12-33
- FLOAT instruction, 4-30, 12-33
- floating point
  - addition, 4-20
  - conversion (to/from IEEE), 12-42
  - conversion to integer, 4-28
  - extended-precision format, 4-8
  - format conversion, 4-9
  - formats, 12-43
    - IEEE*, 12-44
  - multiplication, 4-15
  - normalization, 4-20, 4-24
  - pop and push, 12-13
  - reciprocal, 4-31
  - register format, 3-4
  - rounding value, 4-26
  - short format, 4-6
  - single-precision format, 4-7
  - subtraction, 4-20
  - underflow, 4-21

flush pipeline, 12-22

formats

conversion, floating-point, 4-9

See also *conversion of formats*

floating point, 12-43

signed integer, 4-3

unsigned integer, 4-4

FRIEEE instruction, 12-43

## G

general addressing modes, 2-15, 5-19

global control register (timer), 9-47

global memory, 6-13, 6-17

interface, 2-27, 13-20

global memory interface. See memory interface (local, global)

GOTOs, 12-27

## H

H1/H3 timing, 14-15

## I

IACK instruction, 7-47

$\overline{\text{IACK}}$  pin, 7-47

timing, 14-30

ICFULL flag

description, 12-99

enabling, 3-11

ICRDY flag

description, 12-99

enabling, 3-11

example of use, 12-106

interrupt use, 3-9

IEEE std. 754 (conversions), 4-11

IIOF flag register (IIF), 2-8, **3-12**, 12-103

IIOF pins

boot loader use, 13-6

loading, 13-14

timing, 14-24, 14-29

IIR filters, 12-54

immediate addressing, 5-17

index registers (IR0, IR1), 2-7, **3-5**, 9-16

indirect addressing, 5-5

initialization of processor, 12-3

example code, 12-4

instruction cache, 3-25

instruction register (IR), 2-26

instruction set summary, 2-16—2-25,

**11-3—11-9**

functional groups, 11-3

instructions, Chapter 11

integer formats

short integer, 4-3

signed, 4-3

single-precision integer, 4-3

unsigned, 4-4

interfaces, 13-3

external, 13-3

memory. See memory interfaces (local, global)

parallel processing, 13-37

shared bus, 13-43

interlocked instructions, 2-27, **6-13**, 7-39

interlocked operations, 6-13

internal bus, 2-26

internal interrupt enable register (IIE), 2-8,

**3-10**

interrupt service routine, 12-14, 12-21

interrupt vector table (IVT), 3-16

application, 12-19

boot loader use, 13-7

interrupts, 2-27, **6-23**

answering, 12-28

communication port, 12-100

context switching, 12-15

control bits, 6-24

DMA, 6-25, 9-40, **12-102**

*example*, 12-107

edge/level triggered, 6-23

example, 12-28

external, 2-27

initiation condition, 6-11

NMI, 6-23, 12-14

prioritizing, 6-24

processing, 6-27

service routines, 12-14, 12-21

trap comparison, 6-11

vectors, 3-20, 6-25, 6-26

inverse of floating point, 12-36  
IR filters, 12-51, **12-53**  
IVTP. *See* interrupt vector table (IVT)

## J

JTAG emulation timing, 14-38, B-3  
jumps, 6-9  
    zero-overhead use, 12-11

## L

LAJ instruction, 6-9, 12-11, 12-22, 12-95  
LAJcond instruction, 6-9  
LAT instruction, 12-22  
LATcond instruction, 6-9, 6-11  
lattice filter, 12-88  
LB, LBU instructions, 12-30  
LDFI instruction, 6-13, 7-40  
    timing, 14-20  
LDII instruction, 6-13, 7-40  
    timing, 14-20  
level-triggered interrupts, 6-23  
LH, LHU instructions, 12-30  
LMS algorithm, 12-58  
local memory interface, 2-27, 13-20  
    *See also* memory interface (local, global)  
LOCK signal, 7-39  
logical operations, 12-28  
loops, 12-23—12-26  
LWL, LWR instructions, 12-30

## M

MB, MH instructions, 12-30  
mechanical data, 14-11  
memory, 2-10  
    *See also* memory interface  
    accesses  
        *fetches, 10-20*  
        *loads, stores, 10-21*  
        *pipeline, 10-20*  
        *timing, 10-20*  
    cache, 2-10, 3-25

memory (*Continued*)  
    communication ports memory map, 8-8  
    general organization, 2-10  
    global, 6-13, 6-17  
    interfaces. *See* memory interface (local, global)  
    memory interface control registers, 3-21  
    memory maps, 2-12  
        *analysis module registers, 3-21*  
        *communication ports, 3-23*  
        *DMA, 9-4*  
        *DMA coprocessors, 3-24*  
        *memory interface control registers, 3-21*  
        *overall description, 2-13*  
        *peripherals, 2-14*  
        *timer registers, 3-22, 9-46*  
    organization, 2-10, 3-18  
    pipeline conflicts, 10-11, 10-18  
    RAM, 2-10  
        *zero wait states, 13-21*  
    ranges, 7-10  
    registers. *See* memory interface control registers  
    ROM, 2-10  
        *interface to 'C40, 13-9*  
    ROMEN pin effect, 3-18  
    sharing, 6-16  
    timing, 7-17, 10-20, 14-18  
memory interface (local, global), 13-20  
    bus arbitration, 13-48  
    control registers. *See* memory interface control registers  
    control signals, 7-3  
    features, 7-1  
    RAM (zero wait states), 13-21  
    ready generation, 7-15, 7-17, 13-27  
    shared bus, 13-43  
    shared with bus arbitration, 13-38  
    signals, 7-3  
    strokes  
        *single, 13-21*  
        *two banks, 13-25*  
    timing, 7-17  
    wait states, 7-15, **13-27**

memory interface control registers, 3-21, 7-6  
 address ranges, 7-10, 7-11  
 bit contents, 7-7  
 boot loader use, 13-7  
 example configuration, 13-46  
 LSTRB ACTIVE field, 13-21  
 page size, 7-9  
 PAGESIZE field, 13-21, 13-32  
 reset effect, 7-6  
 STRBx SWW field, 7-16  
 STRBx SWW fields, 13-28  
 timing, 7-17  
 wait states, 7-15

memory maps, 2-12—2-14, **3-18**  
 analysis module registers, 3-21  
 communication ports, 3-23, 8-8  
 DMA, 9-4  
 DMA coprocessors, 3-24  
 memory interface control registers, 3-21  
 overall description, 2-13  
 peripherals, 2-14  
 timer registers, 3-22, 9-46

MPYI3 instruction, 12-42  
 MPYSHI3 instruction, 12-42

multiple processors, 6-13

multiplication, matrix vector, 12-61

multiplication, floating point, 4-15

multiplier, 2-4

**N**

nested block repeats, 6-6

NMI, 6-23

NORM instruction, 4-24

normalization, 12-38  
 floating point value, 4-20, 4-24

**O**

OCEMPTY flag  
 description, 12-99  
 enabling, 3-11

OCRDY flag  
 description, 12-99  
 enabling, 3-11  
 interrupt use, 3-9

optimization (assembler code), 12-95

overflow, 4-21, 4-28

**P**

P flag (cache), 3-25

packing data example, 12-30

page  
 size, 7-9, 7-13  
 switching, 13-32  
 timing, 14-23

parallel addressing modes, 2-15, 5-23

parallel instruction set  
 optimization use, 12-95  
 summary, 2-23—2-25

parallel processing  
 'C40-to-'C40, 13-37  
 general, 1-3  
 shared bus, 13-43

PAU (port arbitration unit), 8-12  
*See also* port arbitration unit  
 operation, 8-12

performance, 1-6

period register (timer), 9-50  
*See also* timers

peripheral bus, 2-28  
 communication port, 2-29  
 general architecture, 2-28  
 map, 3-22

pin (TMS320C40)  
 descriptions, 14-7  
 names, 14-2

pin states at reset, 6-18

pinouts, 14-2

pipeline, 10-1  
 conflicts  
*avoiding*, 12-96  
*branching*, 10-4  
*memory*, 10-11  
*memory (resolving)*, 10-18  
*registers*, 10-8  
 flush, 12-22  
 memory accesses, 10-20  
 structure, 10-2

POPF instruction, 12-13

port arbitration unit, (PAU) 8-5, **8-12**  
     synchronizer timing, 8-32  
 postdisplacement examples, 5-11  
 postindex examples, 5-15  
 predisplacement examples, 5-9  
 preindex examples, 5-13  
 primary register file (CPU), 2-6, 3-3  
 priority (memory)  
     fixed, 13-70  
     rotating, 13-58  
 priority wheel (DMA), 9-24  
 products, 320 family, 1-2  
 program  
     buses, 2-26  
     control, 12-9  
     flow, 6-1  
 program counter (PC), 2-9, 2-26, **3-14**  
 programming methodology, tips, 12-94  
 PUSHF instruction, 12-13

## Q

queues (stack), 5-33

## R

RAM, 2-10  
     zero wait states, 13-21  
 RCPF instruction, 4-31, 12-33, **12-36**  
 ready  
     generation, 7-15, 13-27  
     timing, 7-17  
 reciprocal (RCPF inst.), 4-31  
 reciprocal square root (RSQRF inst.), 4-33  
 register buses, 2-26  
 registers, 2-7  
     auxiliary (AR0–AR7), 2-6, **3-5**  
     block repeat (RS, RE), **3-14, 12-26**  
     block size (BK), 2-7, **3-5**  
     counter (timer), 9-50  
     data page pointer (DP), 2-7, **3-5, 5-4**  
     DMA interrupt enable (DIE), 2-8, **3-8**  
         *bit descriptions, 3-9*  
     extended precision (R0–R11), 2-6, **3-4**  
         *saving (example), 12-13*  
     global control (timer), 9-47

registers (*Continued*)  
     IIOF flag register (IIF), 2-8, **3-12, 6-23, 6-24**  
     index (IR1, IR0), 3-5  
     internal interrupt enable (IIE), 2-8, **3-10, 6-23**  
         *bit descriptions, 3-11*  
     optimization use, 12-96  
     period (timer), 9-50  
     pipeline conflicts, 10-8  
     program counter (PC), 2-9, 2-26, **3-14**  
     repeat count (RC), 2-8, **3-14, 6-2, 12-26**  
     repeat end address (RE), 3-14, 6-2  
     repeat start address (RS), 3-14, 6-2  
     reserved bits, 3-14  
     saved in context switches, 12-16  
     stack pointer (SP), 2-8, **3-5**  
         *application, 12-13*  
     status register (ST), 2-8, **3-5, 11-11**  
         *bit descriptions, 3-6*  
     repeat count register (RC), 2-8, **3-14, 6-2, 12-26**  
     repeat end address register (RE), 3-14, 6-2  
     repeat mode, RPTS initialization, 6-4  
     repeat modes (block, single instruction), 6-2  
         initialization, 6-2  
         optimization use, 12-95  
     repeat start address register (RS), 3-14, 6-2  
 reset, 3-17, **6-18, 12-3**  
     communication ports, 8-14  
     memory interface control registers, 7-6  
     operations performed, 6-22  
     pin states, 6-18  
     signal generation, 13-75  
     timing, 14-28  
     vector mapping, 3-17, 12-3  
     vectors, 6-25  
 RESETLOCx pins, 3-17, 12-3, 13-9  
 RETIcond instruction, 6-9, 6-12  
 RETIcondD instruction, 6-9, 12-22  
 RETScond instruction, 6-9  
 return from subroutine, 6-9  
 RND instruction, 4-26  
 ROM, 2-10  
 rounding of floating point value, 4-26

RPTB and RPTBD instructions, 6-3, 12-23, 12-63  
 optimization use, 12-95  
 RPTS instruction, 6-4  
 example, 12-23, 12-30  
 optimization use, 12-95  
 RSQRF instruction, 4-33, **12-38**

## S

segment start address (SSA) register, 3-25  
 semaphores, 6-17  
 shared bus interface, 13-43  
 short floating-point format, 4-6  
 SIGI instruction, 6-13, 7-44  
 timing, 14-22  
 signal descriptions, 14-7—14-10  
 signal transition levels, 14-14  
 signal-group control, 7-38  
 simulator, 1-10  
 software control, 6-1  
 software development tools, 1-9  
 ANSI C compiler, 1-9  
 assembler/linker, 1-9  
 compiler, 1-9  
 general, 1-9  
 linker, 1-9  
 simulator (state-accurate), 1-10  
 SPOX operating system, 1-9  
 XDS510 emulator, 1-9, B-1  
 split mode (DMA), 9-20, 12-104  
 SPOX operating system, 1-9  
 square root, 12-38  
 stack, 5-31, 5-33  
 dequeues, 5-33  
 queues, 5-33  
 stack pointer (SP), 2-8, **3-5**  
 application, 12-13  
 state diagram, port arbitration unit, 8-13  
 status register (ST), 2-8, **3-5**, 11-11  
 bit descriptions, 3-6  
 STFI instruction, 6-13, 7-42  
 timing, 14-21

STII instruction, 6-13, 7-42  
 timing, 14-21  
 strobe settings, 7-8  
 strobes, 7-12  
 timing, 7-17  
 wait states, 13-21  
 SUBB instruction, 12-41  
 SUBC instruction, 12-33  
 subroutines, 12-11, 12-15  
 calls. *See* calls  
 subtraction, floating point, 4-20  
 system configurations, 13-4

## T

test load circuit, 14-13  
 three-operand addressing modes, 2-15, 5-20  
 throughput, 1-4, 1-6  
 communication port, 1-7  
 DMA, 1-8  
 timer global control register, 9-47  
 diagram, bit summary, 9-47  
 timer registers, 3-22  
 timers, 9-45—9-54  
 applications, 12-97  
 architecture, 2-29  
 counter register, 9-45, 9-50  
 global control register, 9-46, **9-47**  
 operation nodes, 9-51  
 period control registers, 9-50  
 period register, 9-45, 9-50  
 timing, 14-37  
 timing  
 bus control, 14-19  
 memory access, 7-17, 14-18  
 parameters, 14-15—14-26  
 STRB, RDY, 7-17  
 TLCK0,1 pins, 12-97  
 TMS320 family, products 1-2  
 TOIEEE instruction, 12-43  
 trap vector table (TVT), **3-15**  
 boot loader use, 13-7—13-8  
 TRAPcond instruction, 6-9  
 traps, 6-9, 6-11

TSTB instruction, 12-28  
TTL levels, 14-14  
TVTP. *See* trap vector table (TVT)  
twiddle factor, 12-68, 12-78

## U

$\mu$ -law  
    compression, expansion, 12-46  
    conversion, linear, 12-46  
underflow, 4-20  
unified mode (DMA), 9-20  
unpacking data example, 12-31

## V

vectors (reset, interrupts), 6-25

## W

wait states, 7-15, 7-36, 7-37, **13-20**, **13-27**  
    bus disabled, 7-38  
    consecutive reads, then write, 13-23  
    consecutive writes, then read, 13-25  
    multiple waits circuitry, 13-31  
    requirements, 13-20  
word manipulation, 12-30

## X

XDS510 emulator, 1-9, B-1  
XDS510 emulator design considerations, B-1

