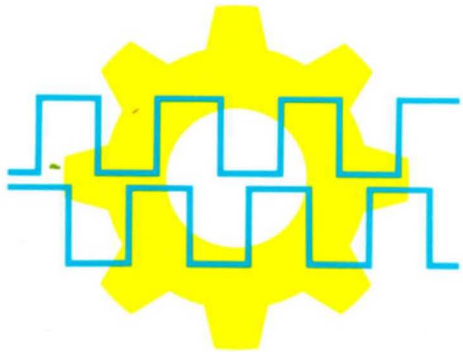
MC68EC030UM/AD



MC68EC030

32-BIT EMBEDDED CONTROLLER USER'S MANUAL

MC68EC030

32-BIT EMBEDDED CONTROLLER USER'S MANUAL

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Concluded)

LIST OF TABLES

LIST OF TABLES (Continued)

PREFACE

The *MC68EC030 User's Manual* describes the capabilities, operation, and programming of the MC68030 32-bit embedded controller. The manual consists of the following sections and appendix. For detailed information on the MC68EC030 instruction set, refer to M68000PM/AD, *M68000 Family Programmer's Reference Manual.*

Section 1. Introduction
Section 2. Data Organization and Addressing Capabilities
Section 3. Instruction Set Summary
Section 4. Processing States
Section 5. Signal Description
Section 6. On-Chip Cache Memories
Section 7. Bus Operation
Section 8. Exception Processing
Section 9. Access Control Unit
Section 10. Coprocessor Interface Description
Section 11. Instruction Execution Timing
Section 12. Applications Information
Section 13. Electrical Characteristics
Section 14. Ordering Information and Mechanical Data
Appendix A. MC68EC030 New Instructions
Index

NOTE

In this manual, assertion and negation are used to specify forcing a signal to a particular state. In particular, assertion and assert refer to a signal that is active or true; negation and negate indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

The audience of this manual includes systems designers, systems programmers, and applications programmers. Systems designers need some knowledge of all sections, with particular emphasis on Sections 1, 5, 6, 7, 13, and 14. Designers who implement a coprocessor for their system also need a thorough knowledge of Section 10. Systems programmers should become

familiar with Sections 1, 2, 3, 4, 6, 8, 9, and 11. Applications programmers can find most of the information they need in Sections 1, 2, 3, 4, 9, 11, 12, and Appendix A.

From a different viewpoint, the audience for this book consists of users of other M68000 Family members and those who are not familiar with the embedded controller. Users of the other family members can find references to similarities to and differences from the Motorola microprocessors throughout the manual. However, Section 1 specifically identifies the MC68EC030 within the rest of the family and contrast its differences.

SECTION 1
INTRODUCTION

The MC68EC030 is a second-generation, full 32-bit, enhanced embedded controller from Motorola. The MC68EC030 is a member of the M68000 Family of devices that combines a central processing unit (CPU) core, a data cache, an instruction cache, and an enhanced bus controller in a single VLSI device. The controller is designed to operate at clock speeds beyond 25 MHz. The MC68EC030 is implemented with 32-bit registers and data paths, 32-bit addresses, a rich instruction set, and versatile addressing modes.

The MC68EC030 is upward object-code compatible with all members of the M68000 Family and has the added features of an on-chip access control unit (ACU), a data cache, and an improved bus interface. It retains the flexible coprocessor interface pioneered in the MC68020 and provides full IEEE floating-point support through this interface with the MC68881 or MC68882 floating-point coprocessor. Also, the internal functional blocks of this embedded controller are designed to operate in parallel, allowing instruction execution to be overlapped. In addition to instruction execution, the internal caches, the on-chip ACU, and the external bus controller all operate in parallel.

The MC68EC030 fully supports the nonmultiplexed bus structure of the MC68020 and MC68030, with 32 bits of address and 32 bits of data. The MC68EC030 bus has an enhanced controller that supports both asynchronous and synchronous bus cycles and burst data transfers. It also supports the MC68020 and MC68030 dynamic bus sizing mechanism that automatically determines device port sizes on a cycle-by-cycle basis as the controller transfers operands to or from external devices.

A block diagram of the MC68EC030 is shown in Figure 1-1. The instructions and data required by the controller are supplied from the internal caches whenever possible. The bus controller manages the transfer of data between the CPU and memory or devices at the address.

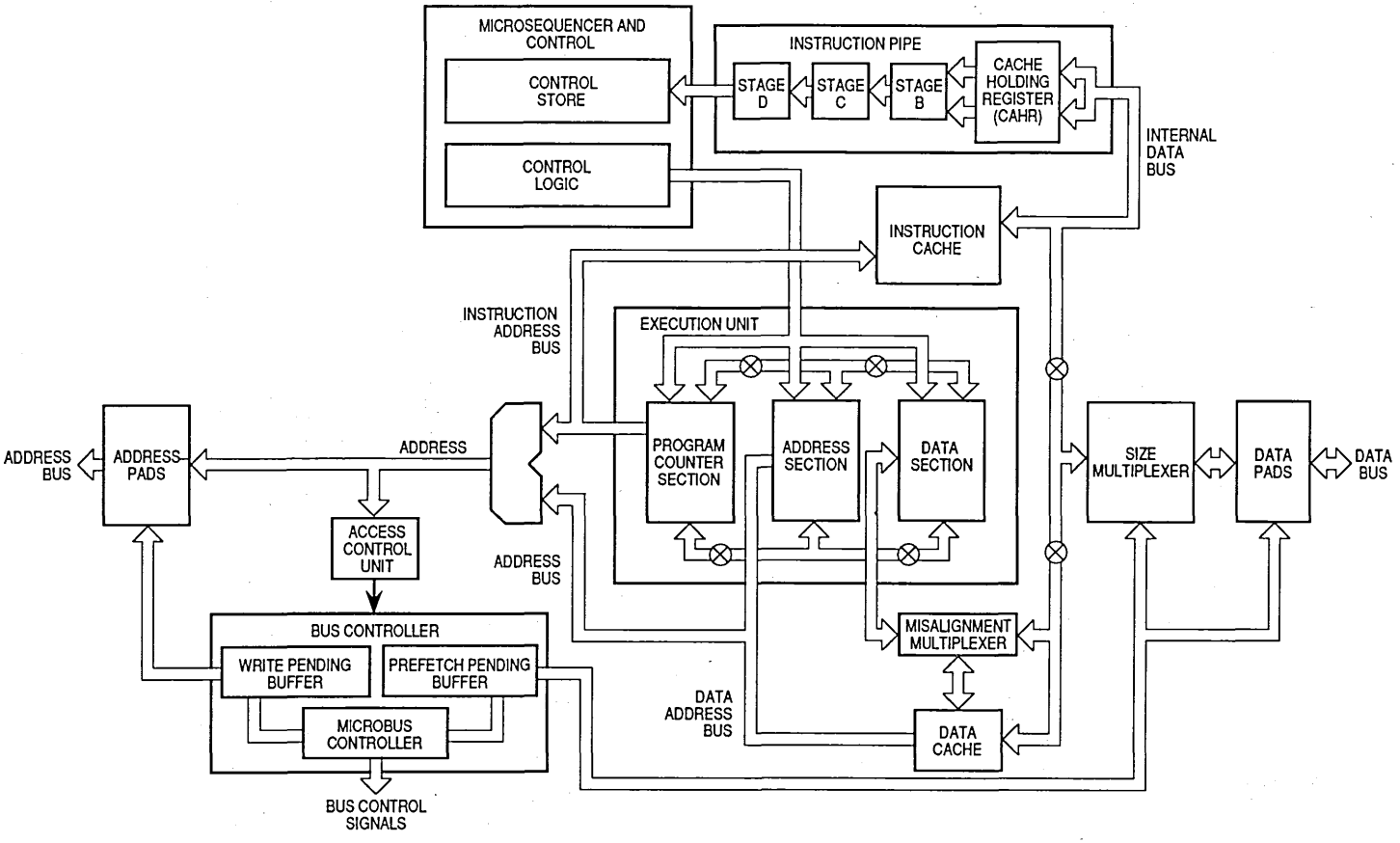


Figure 1-1. Block Diagram

1.1 FEATURES

The features of the MC68EC030 microprocessor are as follows:

- Object-Code Compatible with the M68000 Microprocessor Family
- Complete 32-Bit Nonmultiplexed Address and Data Buses
- 16 32-Bit General-Purpose Data and Address Registers
- Two 32-Bit Supervisor Stack Pointers and Seven Special-Purpose Control Registers
- 256-Byte Instruction Cache and 256-Byte Data Cache Can Be Accessed Simultaneously
- Two Access Control Registers Allow Cache Restrictions on Memory Accesses.
- Pipelined Architecture with Increased Parallelism Allows Accesses to Internal Caches To Occur in Parallel with Bus Transfers and Instruction Execution To Be Overlapped
- Enhanced Bus Controller Supports Asynchronous Bus Cycles (three clocks minimum), Synchronous Bus Cycles (two clocks minimum), and Burst Data Transfers (one clock minimum) all to the Address Space
- Dynamic Bus Sizing Supports 8-, 16-, 32-Bit Memories and Peripherals
- Support for Coprocessors with the M68000 Coprocessor Interface — e.g., Full IEEE Floating-Point Support Provided by the MC68881/MC68882 Floating-Point Coprocessors
- 4-Gbyte Addressing Range
- Implemented in Motorola's HCMOS Technology That Allows CMOS and HMOS (High-Density NMOS) Gates To Be Combined for Maximum Speed, Low Power, and Optimum Die Size

Both improved performance and increased functionality result from the on-chip implementation of the data and instruction caches. The enhanced bus controller and the internal parallelism also provide increased system performance. Finally, the improved bus interface, the reduction in physical size, and the lower power consumption combine to reduce system costs and satisfy cost/performance goals of the system designer.

1.2 MC68EC030 EXTENSIONS TO THE M68000 FAMILY

In addition to the on-chip instruction cache present in the MC68020, the MC68EC030 has an internal data cache. Data that is accessed during read cycles may be stored in the on-chip cache, where it is available for subsequent accesses. The data cache reduces the number of external bus cycles when the data operand required by an instruction is already in the data cache.

Performance is enhanced further because the on-chip caches can be internally accessed in a single clock cycle. In addition, the bus controller provides a two-clock-cycle synchronous mode and burst mode accesses that can transfer data in as little as one clock per long word.

The MC68EC030 enhanced embedded controller allows access control checking to operate in parallel with the CPU core, the internal caches, and the bus controller.

Additional signals support emulation and system analysis. External debug equipment can disable the on-chip caches to freeze the MC68EC030 internal state during breakpoint processing. In addition, the MC68EC030 indicates:

1. The start of a refill of the instruction pipe
2. Instruction boundaries
3. Pending trace or interrupt processing
4. Exception processing
5. Halt conditions

This status and control information allows external debugging equipment to trace the MC68EC030 activity and interact nonintrusively with the MC68EC030 to effectively reduce system debug effort.

1.3 PROGRAMMING MODEL

The programming model of the MC68EC030 consists of two groups of registers: the user model and the supervisor model. This corresponds to the user and supervisor privilege levels. User programs executing at the user privilege level use the registers of the user model. System software executing at the supervisor level uses the control registers of the supervisor level to perform supervisor functions.

Figure 1-2 shows the user programming model, consisting of 16 32-bit general-purpose registers and two control registers:

- General-Purpose 32-Bit Registers (D0–D7, A0–A7)
- 32-Bit Program Counter (PC)
- 8-Bit Condition Code Register (CCR)

The supervisor programming model consists of the registers available to the user plus 11 control registers:

- Two 32-Bit Supervisor Stack Pointers (ISP and MSP)
- 16-Bit Status Register (SR)
- 32-Bit Vector Base Register (VBR)
- Two 32-Bit Alternate Function Code Registers (SFC and DFC)
- 32-Bit Cache Control Register (CACR)
- 32-Bit Cache Address Register (CAAR)
- Two 32-Bit Access Control Registers (AC0 and AC1)
- 16-Bit ACU Status Register (ACUSR)

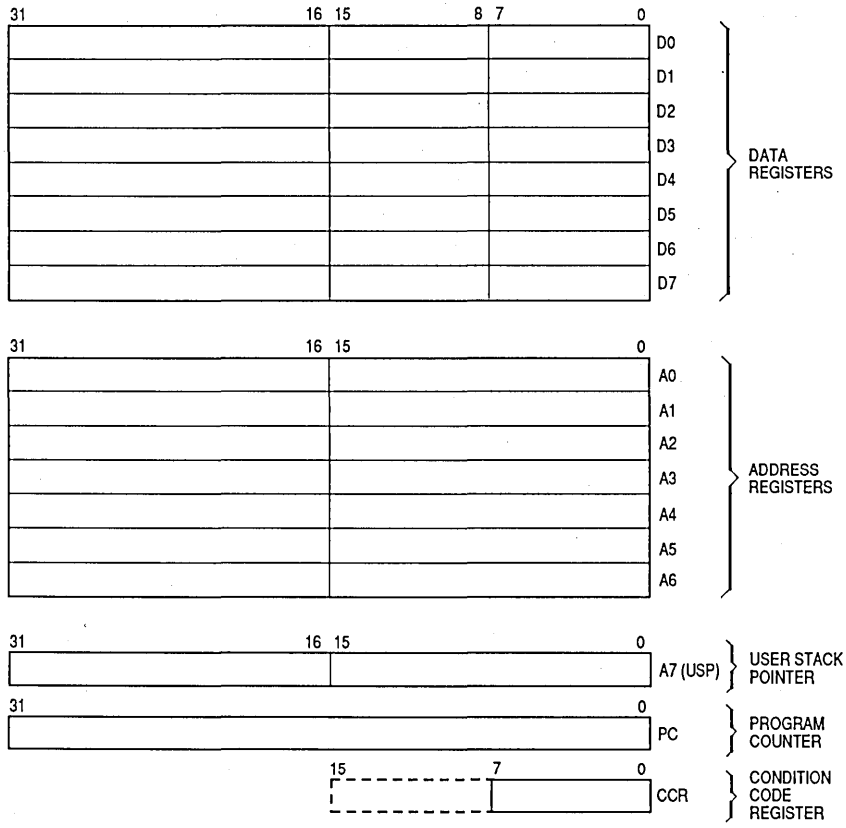


Figure 1-2. User Programming Model

The user programming model remains unchanged from previous M68000 Family microprocessors. The supervisor programming model supplements the user programming model and is used exclusively by the MC68EC030 system programmers who utilize the supervisor privilege level to implement sensitive operating system functions, I/O control, and access control subsystems. The supervisor programming model contains all the controls to access and enable the special features of the MC68EC030. This segregation was carefully planned so that all application software is written to run at the nonprivileged user level and migrates to the MC68EC030 from any M68000 platform without modification. Since system software is usually modified by system programmers when ported to a new design, the control features are properly placed in the supervisor programming model. For example, the access control feature of the MC68EC030 is new to the M68000 Family supervisor programming model, and the two access control registers are new

additions to the M68000 Family supervisor programming model for the MC68EC030. Only supervisor code uses this feature, and user application programs remain unaffected.

Registers D0–D7 are used as data registers for bit and bit field (1 to 32 bits), byte (8 bit), word (16 bit), long-word (32 bit), and quad-word (64 bit) operations. Registers A0–A6 and the user, interrupt, and master stack pointers are address registers that may be used as software stack pointers or base address registers. Register A7 (shown as A7′ and A7″ in Figure 1-3) is a register designation that applies to the user stack pointer in the user privilege level and to either the interrupt or master stack pointer in the supervisor privilege level. In the supervisor privilege level, the active stack pointer (interrupt or master) is called the supervisor stack pointer (SSP). In addition, the address registers may be used for word and long-word operations. All of the 16 general-purpose registers (D0–D7, A0–A7) may be used as index registers.

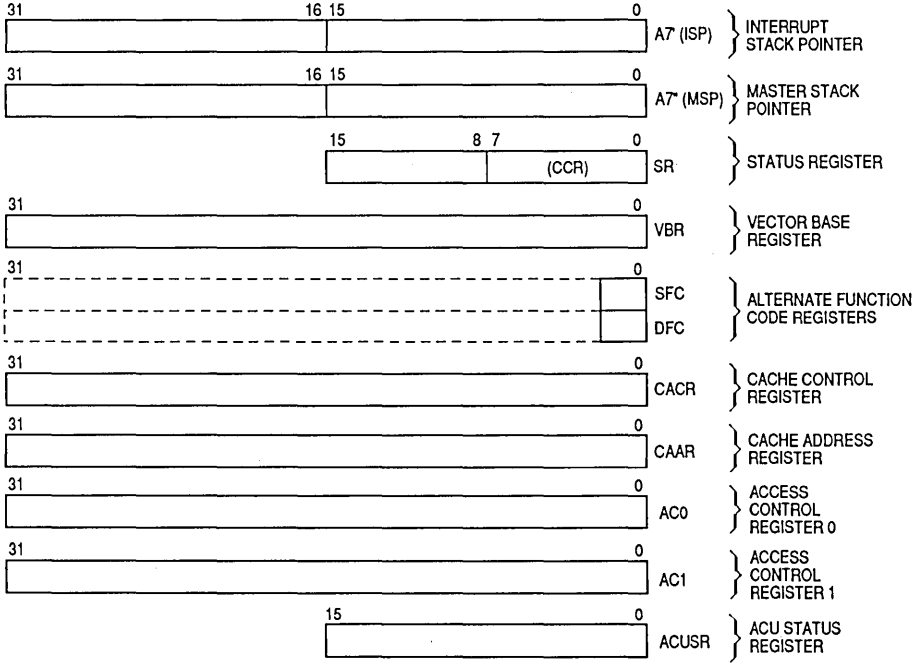


Figure 1-3. Supervisor Programming Model Supplement

The program counter (PC) contains the address of the next instruction to be executed by the MC68EC030. During instruction execution and exception processing, the controller automatically increments the contents of the PC or places a new value in the PC, as appropriate.

The status register, SR, (see Figure 1-4) stores the controller status. It contains the condition codes that reflect the results of a previous operation and can be used for conditional instruction execution in a program. The condition codes are extend (X), negative (N), zero (Z), overflow (V), and carry (C). The user byte containing the condition codes is the only portion of the status register information available in the user privilege level, and it is referenced as the CCR in user programs. In the supervisor privilege level, software can access the full status register, including the interrupt priority mask (three bits) as well as additional control bits. These bits indicate whether the controller is in:

1. One of two trace modes (T1, T0)
2. Supervisor or user privilege level (S)
3. Master or interrupt mode (M)

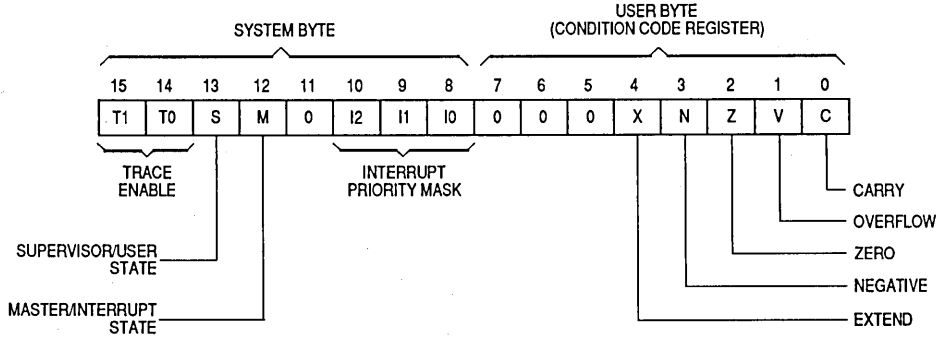


Figure 1-4. Status Register

The vector base register (VBR) contains the base address of the exception vector table in memory. The displacement of an exception vector is added to the value in this register to access the vector table.

Alternate function code registers, SFC and DFC, contain 3-bit function codes. Function codes can be considered extensions of the 32-bit linear address that optionally provide as many as eight 4-Gbyte address spaces. Function codes are automatically generated by the controller to select address spaces for data and program at the user and supervisor privilege levels and a CPU

address space for controller functions (e.g., coprocessor communications). Registers SFC and DFC are used by certain instructions to explicitly specify the function codes for operations.

The cache control register (CACR) controls the on-chip instruction and data caches of the MC68EC030. The cache address register (CAAR) stores an address for cache control functions.

The access control registers, AC0 and AC1, can each specify separate blocks of memory as accessible with restrictions. Function codes and the eight most significant bits of the address can be used to define the area of memory and type of access; either read, write, or both types of memory access can be controlled. The access control feature allows addresses that match the ACx registers to be cache inhibited, which is useful for marking I/O space as noncacheable.

The ACU status register (ACUSR) contains access control status information resulting from a test of the access control registers.

1.4 DATA TYPES AND ADDRESSING MODES

Seven basic data types are supported:

1. Bits
2. Bit Fields (Fields of consecutive bits, 1–32 bits long)
3. BCD Digits (Packed: 2 digits/byte; Unpacked: 1 digit/byte)
4. Byte Integers (8 bits)
5. Word Integers (16 bits)
6. Long-Word Integers (32 bits)
7. Quad-Word Integers (64 bits)

In addition, the instruction set supports operations on other data types such as memory addresses. The coprocessor mechanism allows direct support of floating-point operations with the MC68881 and MC68882 floating-point coprocessors as well as specialized user-defined data types and functions.

The 18 addressing modes, shown in Table 1-1, include nine basic types:

1. Register Direct
2. Register Indirect
3. Register Indirect with Index
4. Memory Indirect
5. Program Counter Indirect with Displacement
6. Program Counter Indirect with Index
7. Program Counter Memory Indirect
8. Absolute
9. Immediate

The register indirect addressing modes can also postincrement, predecrement, offset, and index addresses. The program counter relative mode also has index and offset capabilities. As in the MC68020 and MC68030, both modes are extended to provide indirect reference through memory. In addition to these addressing modes, many instructions implicitly specify the use of the condition code register, stack pointer, and/or program counter.

Table 1-1. Addressing Modes

Addressing Modes	Syntax
Register Direct	
Data Register Direct	Dn
Address Register Direct	An
Register Indirect	
Address Register Indirect	(An)
Address Register Indirect with Postincrement	(An)+
Address Register Indirect with Predecrement	−(An)
Address Register Indirect with Displacement	(d_{16},An)
Register Indirect with Index	
Address Register Indirect with Index (8-Bit Displacement)	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	(bd,An,Xn)
Memory Indirect	
Memory Indirect Postindexed	([bd,An],Xn,od)
Memory Indirect Preindexed	([bd,An,Xn],od)
Program Counter Indirect with Displacement	(d_{16},PC)
Program Counter Indirect with Index	
PC Indirect with Index (8-Bit Displacement)	(d_8,PC,Xn)
PC Indirect with Index (Base Displacement)	(bd,PC,Xn)
Program Counter Memory Indirect	
PC Memory Indirect Postindexed	([bd,PC],Xn,od)
PC Memory Indirect Preindexed	([bd,PC,Xn],od)
Absolute	
Absolute Short	(xxx).W
Absolute Long	(xxx).L
Immediate	#⟨data⟩

NOTES:

Dn = Data Register, D0–D7

An = Address Register, A0–A7

8, d_{16} = A twos-complement or sign-extended displacement; added as part of the effective address calculation; size is 8 (d_8) or 16 (d_{16}) bits; when omitted, assemblers use a value of zero.

Xn = Address or data register used as an index register; form is Xn.SIZE*SCALE, where SIZE is .W or .L (indicates index register size) and SCALE is 1, 2, 4, or 8 (index register is multiplied by SCALE); use of SIZE and/or SCALE is optional.

bd = A twos-complement base displacement; when present, size can be 16 or 32 bits.

od = Outer displacement, added as part of effective address calculation after any memory indirection; use is optional with a size of 16 or 32 bits.

PC = Program Counter

⟨data⟩ = Immediate value of 8, 16, or 32 bits

( ) = Effective Address

[ ] = Use as indirect access to long-word address.

1.5 INSTRUCTION SET OVERVIEW

The instructions in the MC68EC030 instruction set are listed in Table 1-2. The instruction set has been tailored to support structured high-level languages and sophisticated operating systems. Many instructions operate on bytes, words, or long words, and most instructions can use any of the 18 addressing modes.

1.6 THE ACCESS CONTROL UNIT

The ACU supports access controls by comparing the address, read/write, and function codes of the access to the address, read/write, and function codes of the ACx registers. If a match occurs and the ACx register is enabled, then the cacheability of the access is controlled by the cache inhibit bit of the ACx. If a match does not occur, the cacheability is determined by the cache inhibit in ($\overline{\text{CIIN}}$) signal. This allows parts of the memory map to be cache inhibited. The effect of the ACU can be duplicated in hardware by asserting $\overline{\text{CIIN}}$ on the appropriate addresses. The status of the ACx registers can be tested, and the results can be stored in the ACUSR.

Table 1-2. Instruction Set

Mnemonic	Description
ABCD	Add Decimal with Extend
ADD	Add
ADDA	Add Address
ADDI	Add Immediate
ADDQ	Add Quick
ADDX	Add with Extend
AND	Logical AND
ANDI	Logical AND Immediate
ASL, ASR	Arithmetic Shift Left and Right
Bcc	Branch Conditionally
BCHG	Test Bit and Change
BCLR	Test Bit and Clear
BFCHG	Test Bit Field and Change
BFCLR	Test Bit Field and Clear
BFEXTS	Signed Bit Field Extract
BFEXTU	Unsigned Bit Field Extract
BFFFO	Bit Field Find First One
BFINS	Bit Field Insert
BFSET	Test Bit Field and Set
BFTST	Test Bit Field
BKPT	Breakpoint
BRA	Branch
BSET	Test Bit and Set
BSR	Branch to Subroutine
BTST	Test Bit
CAS	Compare and Swap Operands
CAS2	Compare and Swap Dual Operands
CHK	Check Register Against Bound
CHK2	Check Register Against Upper and Lower Bounds
CLR	Clear
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CMPM	Compare Memory to Memory
CMP2	Compare Register Against Upper and Lower Bounds
DBcc	Test Condition, Decrement and Branch
DIVS, DIVSL	Signed Divide
DIVU, DIVUL	Unsigned Divide
EOR	Logical Exclusive OR
EORI	Logical Exclusive OR Immediate
EXG	Exchange Registers
EXT, EXTB	Sign Extend
ILLEGAL	Take Illegal Instruction Trap
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link and Allocate
LSL, LSR	Logical Shift Left and Right

Mnemonic	Description
MOVE	Move
MOVEA	Move Address
MOVE CCR	Move Condition Code Register
MOVE SR	Move Status Register
MOVE USP	Move User Stack Pointer
MOVEC	Move Control Register
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral
MOVEQ	Move Quick
MOVES	Move Alternate Address Space
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NEGX	Negate with Extend
NOP	No Operation
NOT	Logical Complement
OR	Logical Inclusive OR
ORI	Logical Inclusive OR Immediate
ORI CCR	Logical Inclusive OR Immediate to Condition Codes
ORI SR	Logical Inclusive OR Immediate to Status Register
PACK	Pack BCD
PEA	Push Effective Address
PMOVE	Move to/from AC Registers
PTESTR, PTESTW	Test ACU for an Address
RESET	Reset External Devices
ROL, ROR	Rotate Left and Right
ROXL, ROXR	Rotate with Extend Left and Right
RTD	Return and Deallocate
RTE	Return from Exception
RTR	Return and Restore Codes
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
Scc	Set Conditionally
STOP	Stop
SUB	Subtract
SUBA	Subtract Address
SUBI	Subtract Immediate
SUBQ	Subtract Quick
SUBX	Subtract with Extend
SWAP	Swap Register Words
TAS	Test Operand and Set
TRAP	Trap
TRAPcc	Trap Conditionally
TRAPV	Trap on Overflow
TST	Test Operand
UNLK	Unlink
UNPK	Unpack BCD

Mnemonic	Description
cpBcc	Branch Conditionally
cpDBcc	Test Coprocessor Condition, Decrement and Branch
cpGEN	Coprocessor General Instruction

Mnemonic	Description
cpRESTORE	Restore Internal State of Coprocessor
cpSAVE	Save Internal State of Coprocessor
cpScc	Set Conditionally
cpTRAPcc	Trap Conditionally

1.7 PIPELINED ARCHITECTURE

The MC68EC030 uses a three-stage pipelined internal architecture to provide for optimum instruction throughput. The pipeline allows as many as three words of a single instruction or three consecutive instructions to be decoded concurrently.

1.8 THE CACHE MEMORIES

Due to locality of reference, instructions and data that are used in a program have a high probability of being reused within a short time. Additionally, instructions and data operands that reside in proximity to the instructions and data currently in use also have a high probability of being utilized within a short period. To exploit these locality characteristics, the MC68EC030 contains two on-chip caches, a data cache, and an instruction cache.

Each of the caches stores 256 bytes of information, organized as 16 entries, each containing a block of four long words (16 bytes). The controller fills the cache entries either one long word at a time or, during burst mode accesses, four long words consecutively. The burst mode of operation not only fills the cache efficiently but also captures adjacent instruction or data items that are likely to be required in the near future due to locality characteristics of the executing task.

The caches improve the overall performance of the system by reducing the number of bus cycles required by the controller to fetch information from memory and by increasing the bus bandwidth available for other bus masters in the system. Addition of the data cache in the MC68EC030 extends the benefits of cache techniques to all memory accesses. During a write cycle, the data cache circuitry writes data to a cached data item as well as to the item in memory, maintaining consistency between data in the cache and that in memory. However, writing data that is not in the cache may or may not cause the data item to be stored in the cache, depending on the write allocation policy selected in the cache control register (CACR).

SECTION 2
DATA ORGANIZATION AND ADDRESSING CAPABILITIES

Most external references to memory by an embedded controller are either program references or data references; they either access instruction words or operands (data items) for an instruction. Program references refer to the program space, the section of memory that contains the program instructions and any immediate data operands residing in the instruction stream. M68000PM/AD, *M68000 Programmer's Reference Manual*, and **APPENDIX A MC68EC030 NEW INSTRUCTIONS** have descriptions of the instructions in the program space. Data references refer to the data space, the section of memory that contains the program data. Data items in the instruction stream can be accessed with the program counter relative addressing modes, and these accesses are classified as program references. A third type of external reference, classified as a CPU space reference, is used for coprocessor communications, interrupt acknowledge cycles, and breakpoint acknowledge cycles is classified as a CPU space reference. The MC68EC030 automatically sets the function codes to access the program space, the data space, or the CPU space for special functions as required. The access control unit (ACU) can use function codes to specify cacheability of separate program (read only) and data (read-write) memory areas.

This section describes the data organization and addressing capabilities of the MC68EC030. It lists the types of operands used by instructions and describes the registers and their use as operands. Next, the section describes the organization of data in memory and the addressing modes available to access data in memory. Last, the section describes the system stack and user program stacks and queues.

2.1 INSTRUCTION OPERANDS

The MC68EC030 supports a general-purpose set of operands to serve the requirements of a large range of applications. Operands of MC68EC030 instructions may reside in registers, in memory, or within the instructions themselves. An instruction operand might also reside in a coprocessor. An operand may be a single bit, a bit field of from 1 to 32 bits in length, a byte (8 bits), a word (16 bits), a long word (32 bits), or a quad word (64 bits). The

operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Coprocessors are designed to support special computation models that require very specific but widely varying data operand types and sizes. Hence, coprocessor instructions can specify operands of any size.

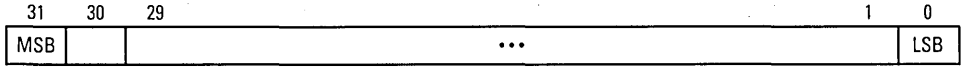
2.2 ORGANIZATION OF DATA IN REGISTERS

The eight data registers can store data operands of 1, 8, 16, 32, and 64 bits, addresses of 16 or 32 bits, or bit fields of 1 to 32 bits. The seven address registers and the three stack pointers are used for address operands of 16 or 32 bits. The control registers (SR, VBR, SFC, DFC, CACR, CAAR, AC0, AC1, and ACUSR) vary in size according to function. Coprocessors may define unique operand sizes and support them with on-chip registers accordingly.

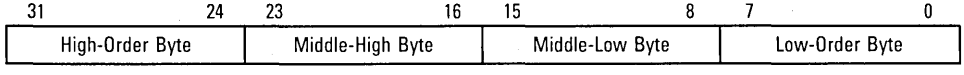
2.2.1 Data Registers

Each data register is 32 bits wide. Byte operands occupy the low-order 8 bits, word operands the low-order 16 bits, and long-word operands the entire 32 bits. When a data register is used as either a source or destination operand, only the appropriate low-order byte or word (in byte or word operations, respectively) is used or changed; the remaining high-order portion is neither used nor changed. The least significant bit of a long-word integer is addressed as bit zero, and the most significant bit is addressed as bit 31. For bit fields, the most significant bit is addressed as bit zero, and the least significant bit is addressed as the width of the field minus one. If the width of the field plus the offset is greater than 32, the bit field wraps around within the register. The following illustration shows the organization of various types of data in the data registers.

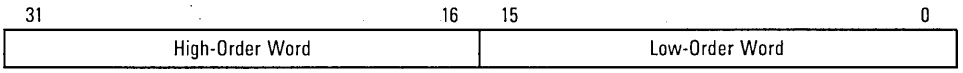
Bit (0≤Modulo (Offset)<31, Offset of 0 = MSB)



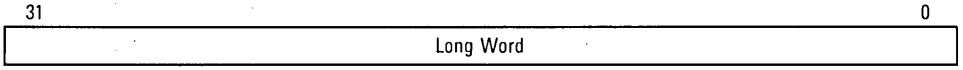
Byte



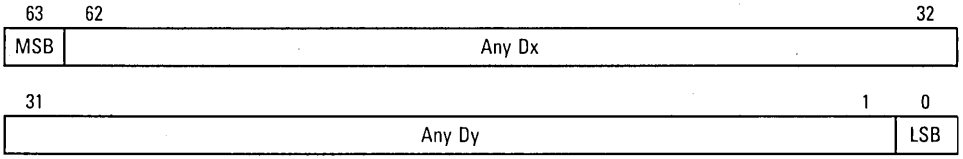
16-Bit Word



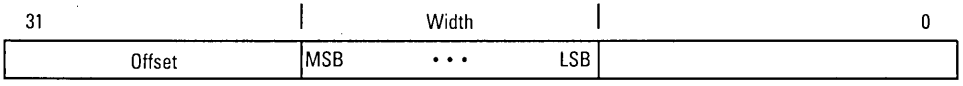
Long Word



Quad Word

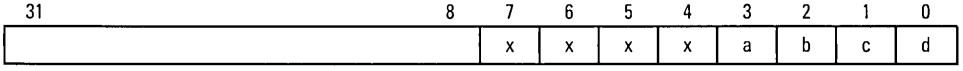


Bit Field (0≤Offset<32, 0<Width≤32)

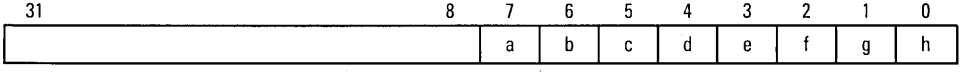


Note: If width + offset<32, bit field wraps around within the register.

Unpacked BCD (a = MSB)



Packed BCD (a = MSB First Digit, e = MSB Second Digit)



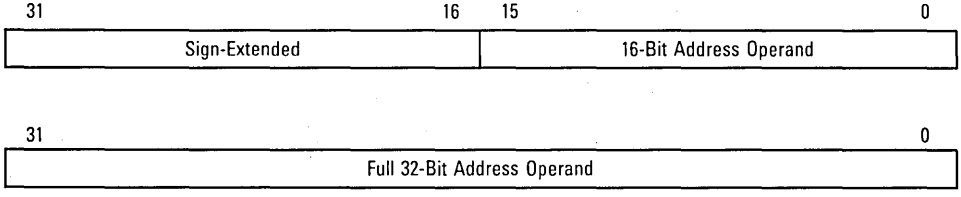
Data Organization in Data Registers

Quad-word data consists of two long words: for example, the product of 32-bit multiply or the quotient of 32-bit divide operations (signed and unsigned). Quad words may be organized in any two data registers without restrictions on order or pairing. There are no explicit instructions for the management of this data type, although the MOVEM instruction can be used to move a quad word into or out of the registers.

Binary-coded decimal (BCD) data represents decimal numbers in binary form. Although many BCD codes have been devised, the BCD instructions of the M68000 Family support formats in which the four least significant bits consist of a binary number having the numeric value of the corresponding decimal number. Two BCD formats are used. In the unpacked BCD format, a byte contains one digit; the four least significant bits contain the binary value and the four most significant bits are undefined. Each byte of the packed BCD format contains two digits; the least significant four bits contain the least significant digit.

2.2.2 Address Registers

Each address register and stack pointer is 32 bits wide and holds a 32-bit address. Address registers cannot be used for byte-sized operands. Therefore, when an address register is used as a source operand, either the low-order word or the entire long-word operand is used, depending upon the operation size. When an address register is used as the destination operand, the entire register is affected, regardless of the operation size. If the source operand is a word size, it is first sign-extended to 32 bits and then used in the operation to an address register destination. Address registers are used primarily for addresses and to support address computation. The instruction set includes instructions that add to, subtract from, compare, and move the contents of address registers. The following example shows the organization of addresses in address registers.



Address Organization in Address Registers

2.2.3 Control Registers

The control registers described in this section contain control information for supervisor functions and vary in size. With the exception of the user portion of the status register (CCR), they are accessed only by instructions at the supervisor privilege level.

The status register (SR), shown in Figure 1-4, is 16 bits wide. Only 12 bits of the status register are defined; all undefined values are reserved by Motorola for future definition. The undefined bits are read as zeros and should be written as zeros for future compatibility. The lower byte of the status register is the CCR. Operations to the CCR can be performed at the supervisor or user privilege level. All operations to the status register and CCR are word-sized operations, but for all CCR operations, the upper byte is read as all zeros and is ignored when written, regardless of privilege level.

The supervisor programming model (see Figure 1-3) shows the control registers. The cache control register (CACR) provides control and status information for the on-chip instruction and data caches. The cache address register (CAAR) contains the address for cache control functions. The vector base register (VBR) provides the base address of the exception vector table. All operations involving the CACR, CAAR, and VBR are long-word operations, whether these registers are used as the source or the destination operand.

The alternate function code registers (SFC and DFC) are 32-bit registers with only bits 2–0 implemented that contain the address space values (FC0–FC2) for the read or write operands of MOVES and PTEST instructions. The MOVEC instruction is used to transfer values to and from the alternate function code registers. These are long-word transfers; the upper 29 bits are read as zeros and are ignored when written.

The remaining control registers in the supervisor programming model are used by the acces control unit (ACU). The access control registers (AC0 and AC1) contain 32 bits each; they identify memory area cacheability. Data transfers to and from these registers are long-word transfers. The ACU status register (ACUSR) stores the status of the ACU after execution of a PTEST instruction. It is a 16-bit register, and transfers to and from the ACUSR are word transfers. Refer to **SECTION 9 ACCESS CONTROL UNIT** for more detail.

2.3 ORGANIZATION OF DATA IN MEMORY

Memory is organized on a byte-addressable basis where lower addresses correspond to higher order bytes. The address, N, of a long-word data item corresponds to the address of the most significant byte of the highest order word. The lower order word is located at address N + 2, leaving the least significant byte at address N + 3 (refer to Figure 2-1). Notice that the MC68EC030 does not require data to be aligned on word boundaries (refer to Figure 2-2), but the most efficient data transfers occur when data is aligned on the same byte boundary as its operand size. However, instruction words must be aligned on word boundaries.

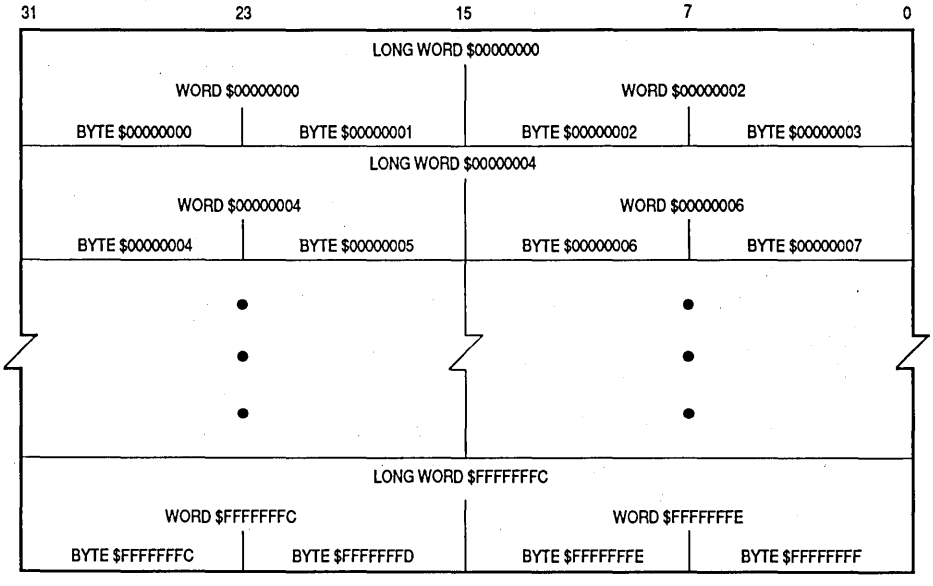
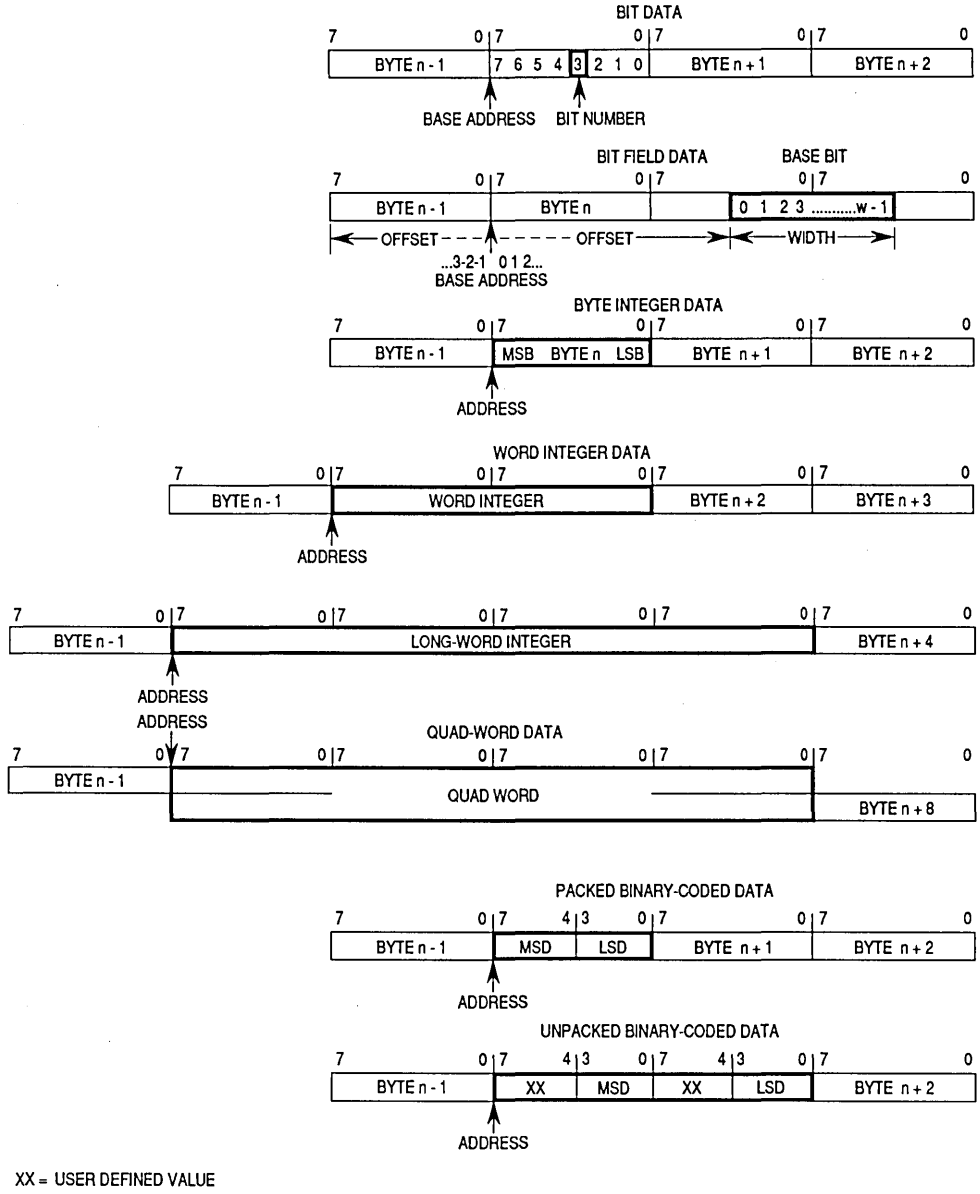


Figure 2-1. Memory Operand Address



XX = USER DEFINED VALUE

Figure 2-2. Memory Data Organization

The data types supported in memory by the MC68EC030 are bit and bit field data; integer data of 8, 16, or 32 bits; 32-bit addresses; and BCD data (packed and unpacked). These data types are organized in memory as shown in Figure 2-2. Note that all of these data types can be accessed at any byte address.

Coprocessors can implement any data types and lengths up to 255 bytes. For example, the MC68881/MC68882 floating-point coprocessors support memory accesses for quad-word-sized items (double-precision floating-point values).

A bit operand is specified by a base address that selects one byte in memory (the base byte) and a bit number that selects the one bit in this byte. The most significant bit of the byte is bit 7.

A bit field operand is specified by:

1. A base address that selects one byte in memory,
2. A bit field offset that indicates the leftmost (base) bit of the bit field in relation to the most significant bit of the base byte, and
3. A bit field width that determines how many bits to the right of the base bit are in the bit field.

The most significant bit of the base byte is bit field offset 0, the least significant bit of the base byte is bit field offset 7, and the least significant bit of the previous byte in memory is bit offset -1. Bit field offsets may have values in the range of -2^{31} to $2^{31}-1$, and bit field widths may range between 1 and 32 bits.

2.4 ADDRESSING MODES

The addressing mode of an instruction can specify the value of an operand (with an immediate operand), a register that contains the operand (with the register direct addressing mode), or how the effective address of an operand in memory is derived. An assembler syntax has been defined for each addressing mode.

Figure 2-3 shows the general format of the single effective address instruction operation word. The effective address field specifies the addressing mode for an operand that can use one of the numerous defined modes. The ⟨ea⟩ designation is composed of two 3-bit fields: the mode field and the register field. The value in the mode field selects one or a set of addressing modes. The register field specifies a register for the mode or a submode for modes that do not use registers.

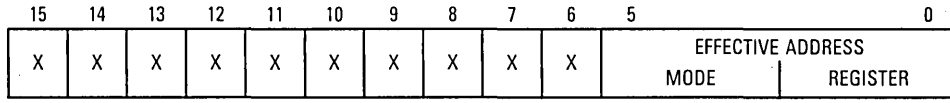


Figure 2-3. Single Effective Address Instruction Operation Word

Many instructions imply the addressing mode for one of the operands. The formats of these instructions include appropriate fields for operands that use only one addressing mode.

The effective address field may require additional information to fully specify the operand address. This additional information, called the effective address extension, is contained in an additional word or words and is considered part of the instruction. Refer to **2.5 EFFECTIVE ADDRESS ENCODING SUMMARY** for a description of the extension word formats.

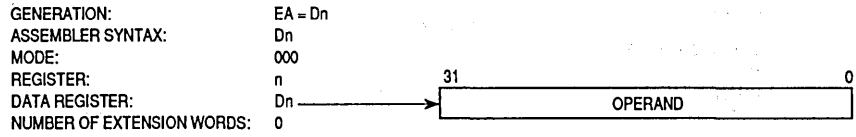
The notational conventions used in the addressing mode descriptions in this section are:

EA—Effective address
An—Address register n
Example: A3 is address register 3
Dn—Data register n
Example: D5 is data register 5
Xn.SIZE*SCALE—Denotes index register n (data or address), the index size (W for word, L for long word), and a scale factor (1, 2, 4, or 8, for no, word, long-word or quad-word scaling, respectively).
PC—The program counter
d_n—Displacement value, n bits wide
bd—Base displacement
od—Outer displacement
L—Long-word size
W—Word size
( )—Identify an indirect address in a register
[ ]—Identify an indirect address in memory

When the addressing mode uses a register, the register field of the operation word specifies the register to be used. Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used.

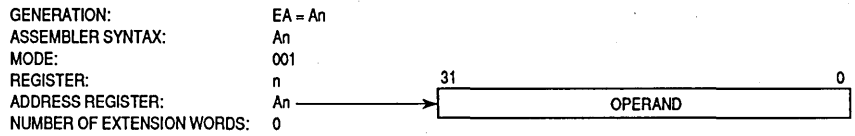
2.4.1 Data Register Direct Mode

In the data register direct mode, the operand is in the data register specified by the effective address register field.



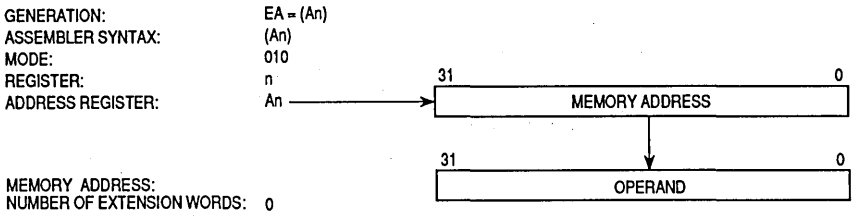
2.4.2 Address Register Direct Mode

In the address register direct mode, the operand is in the address register specified by the effective address register field.



2.4.3 Address Register Indirect Mode

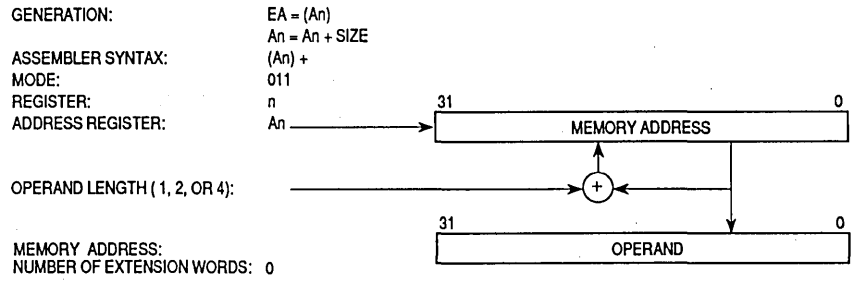
In the address register indirect mode, the operand is in memory, and the address of the operand is in the address register specified by the register field.



2.4.4 Address Register Indirect with Postincrement Mode

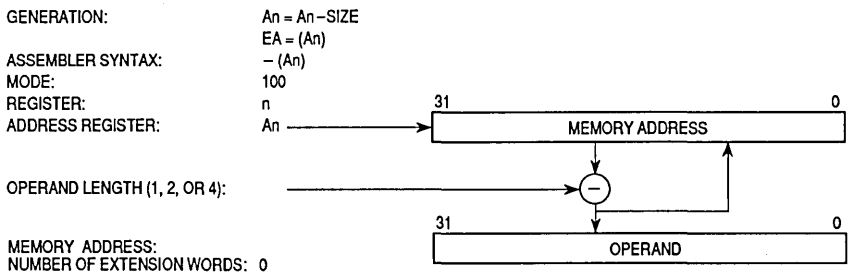
In the address register indirect with postincrement mode, the operand is in memory, and the address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or four, depending on the size of the operand: byte, word, or long word. Coprocessors may support incrementing for any size of operand up

to 255 bytes. If the address register is the stack pointer and the operand size is byte, the address is incremented by two rather than one to keep the stack pointer aligned to a word boundary.



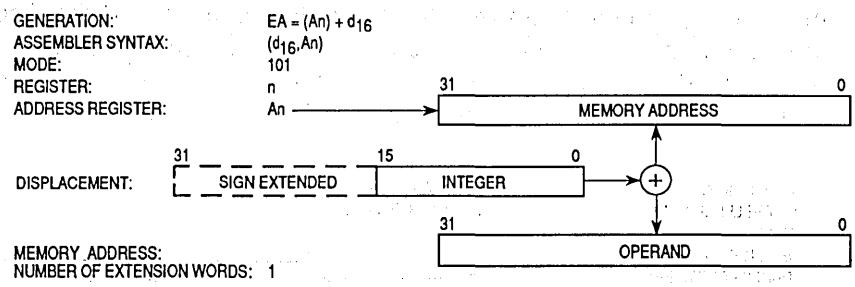
2.4.5 Address Register Indirect with Predecrement Mode

In the address register indirect with predecrement mode, the operand is in memory, and the address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four, depending on the operand size: byte, word, or long word. Coprocessors may support decrementing for any operand size up to 255 bytes. If the address register is the stack pointer and the operand size is byte, the address is decremented by two rather than one to keep the stack pointer aligned to a word boundary.



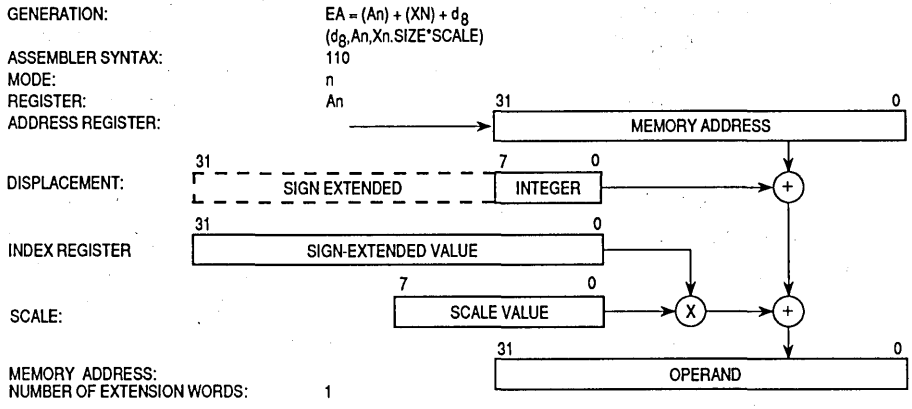
2.4.6 Address Register Indirect with Displacement Mode

In the address register indirect with displacement mode, the operand is in memory. The address of the operand is the sum of the address in the address register plus the sign-extended 16-bit displacement integer in the extension word. Displacements are always sign-extended to 32 bits prior to being used in effective address calculations.



2.4.7 Address Register Indirect with Index (8-Bit Displacement) Mode

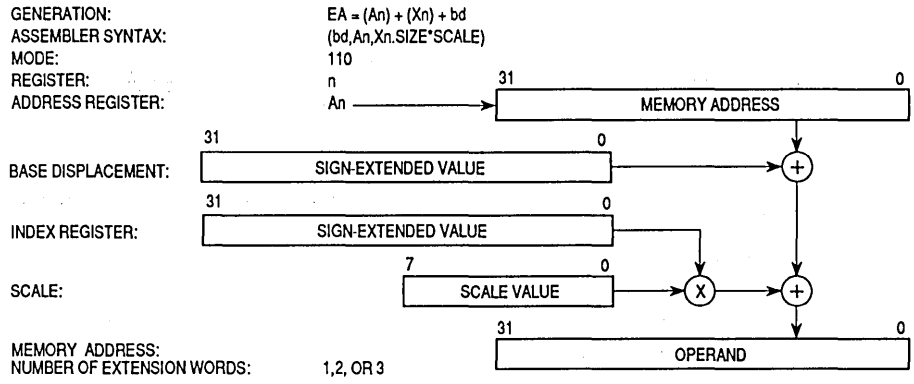
This addressing mode requires one extension word that contains the index register indicator and an 8-bit displacement. The index register indicator includes size and scale information. In this mode, the operand is in memory. The address of the operand is the sum of the contents of the address register, the sign-extended displacement value in the low-order eight bits of the extension word, and the sign-extended contents of the index register (possibly scaled). The user must specify the displacement, the address register, and the index register in this mode.



2.4.8 Address Register Indirect with Index (Base Displacement) Mode

This addressing mode requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The index register indicator includes size and scaling information. The operand is in memory. The address of the operand is the sum of the contents of the address register, the scaled contents of the sign-extended index register, and the base displacement.

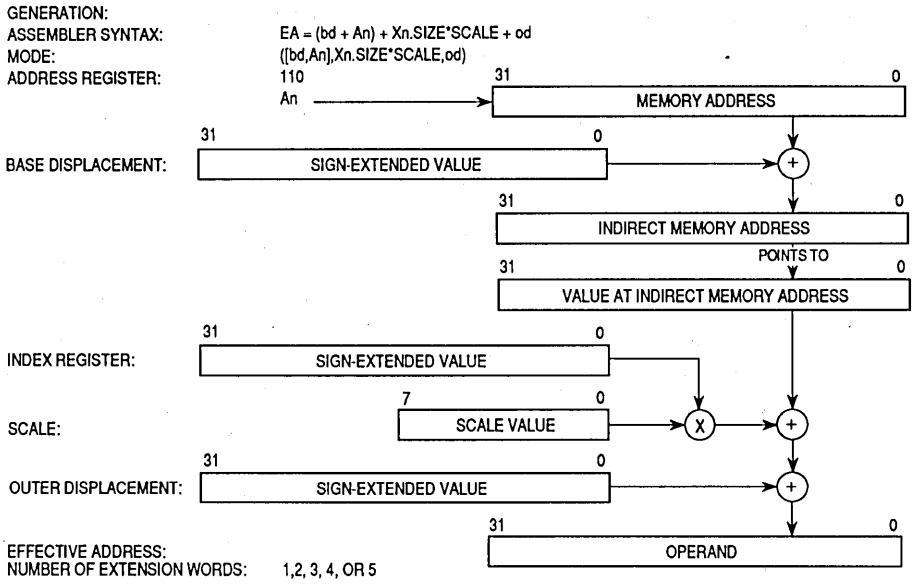
In this mode, the address register, the index register, and the displacement are all optional. If none is specified, the effective address is zero. This mode provides a data register indirect address when no address register is specified and the index register is a data register (Dn).



2.4.9 Memory Indirect Postindexed Mode

In this mode, the operand and its address are in memory. The controller calculates an intermediate indirect memory address using the base register (An) and base displacement (bd). The controller accesses a long word at this address and adds the index operand (Xn.SIZE*SCALE) and the outer displacement to yield the effective address. Both displacements and the index register contents are sign-extended to 32 bits.

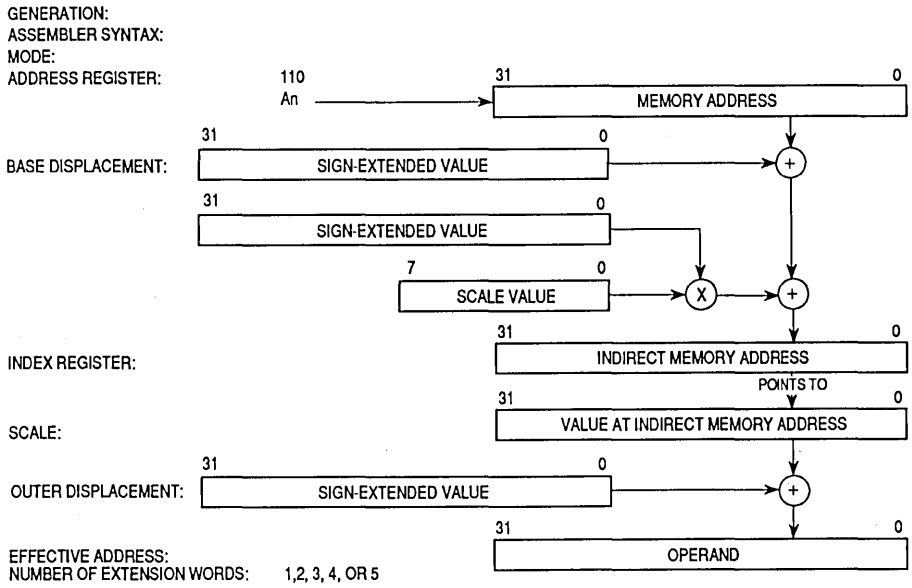
In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



2.4.10 Memory Indirect Preindexed Mode

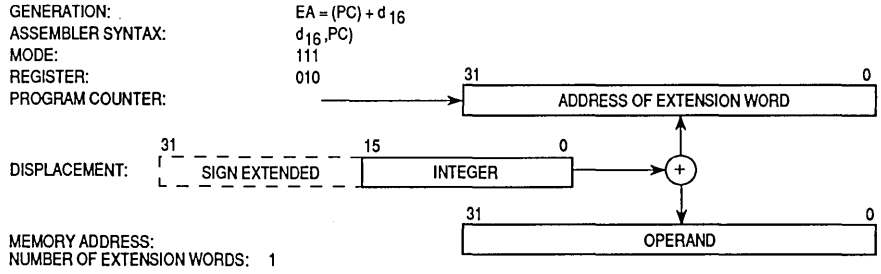
In this mode, the operand and its address are in memory. The controller calculates an intermediate indirect memory address using the base register (An), a base displacement (bd), and the index operand (Xn.SIZE * SCALE). The controller accesses a long word at this address and adds the outer displacement to yield the effective address. Both displacements and the index register contents are sign-extended to 32 bits.

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



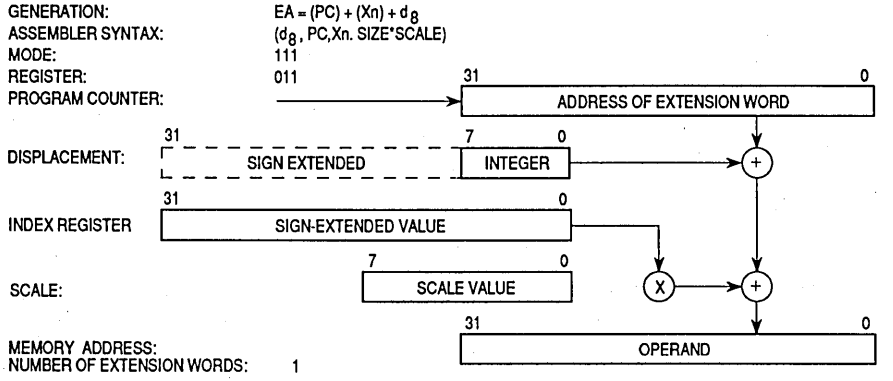
2.4.11 Program Counter Indirect with Displacement Mode

In this mode, the operand is in memory. The address of the operand is the sum of the address in the PC and the sign-extended 16-bit displacement integer in the extension word. The value in the PC is the address of the extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).



2.4.12 Program Counter Indirect with Index (8-Bit Displacement) Mode

This mode is similar to the address register indirect with index (8-bit displacement) mode described in **2.4.7 Address Register Indirect with Index (8-Bit Displacement) Mode**, but the PC is used as the base register. The operand is in memory. The address of the operand is the sum of the address in the PC, the sign-extended displacement integer in the lower eight bits of the extension word, and the sized, scaled, and sign-extended index operand. The value in the PC is the address of the extension word. This reference is a program space reference and is only allowed for reads. The user must include the displacement, the PC, and the index register when specifying this addressing mode.

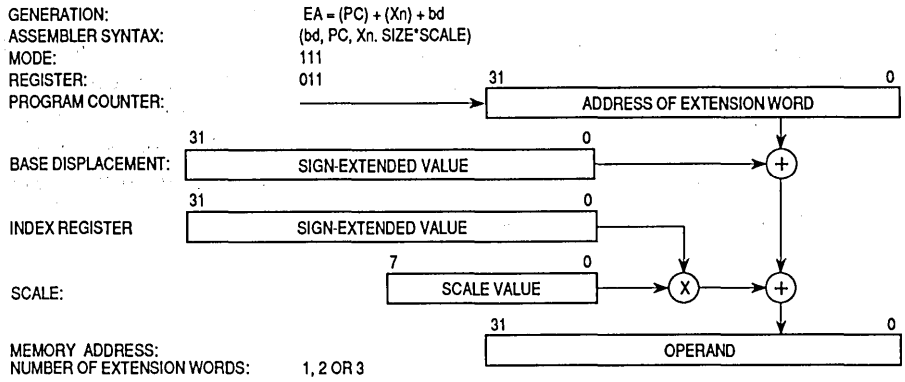


2.4.13 Program Counter Indirect with Index (Base Displacement) Mode

This mode is similar to the address register indirect with index (base displacement) mode described in **2.4.8 Address Register Indirect with Index (Base Displacement) Mode**, but the PC is used as the base register. It requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The operand is in memory. The address of the operand is the sum of the contents of the PC, the scaled contents of the sign-extended index register, and the base displacement. The value of the PC is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).

In this mode, the PC, the index register, and the displacement are all optional. However, the user must supply the assembler notation "ZPC" (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space without using the PC in calculating the effective

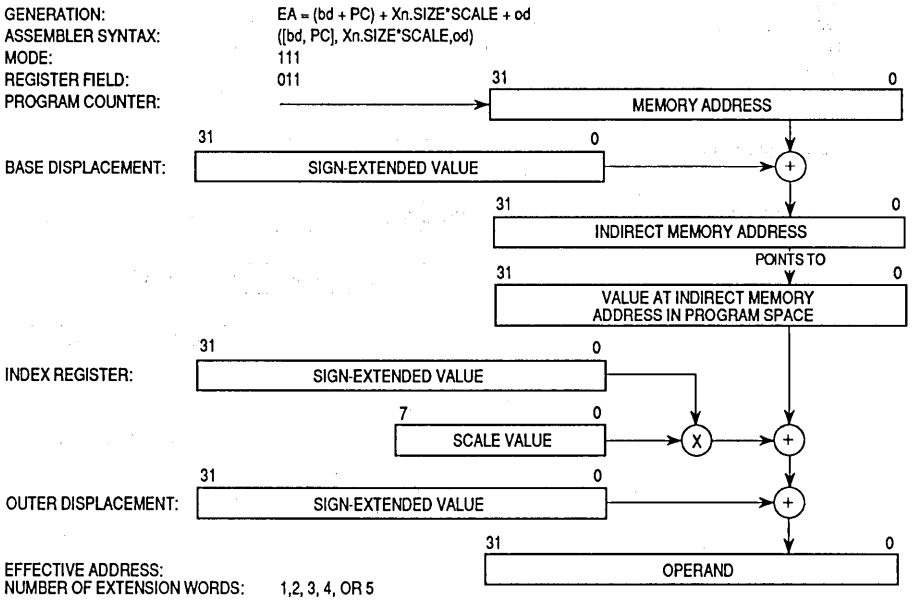
address. The user can access the program space with a data register indirect access by placing ZPC in the instruction and specifying a data register (Dn) as the index register.



2.4.14 Program Counter Memory Indirect Postindexed Mode

This mode is similar to the memory indirect postindexed mode described in **2.4.9 Memory Indirect Postindexed Mode**, but the PC is used as the base register. Both the operand and operand address are in memory. The controller calculates an intermediate indirect memory address by adding a base displacement (bd) to the PC contents. The controller accesses a long word at that address and adds the scaled contents of the index register and the optional outer displacement (od) to yield the effective address. The value of the PC used in the calculation is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).

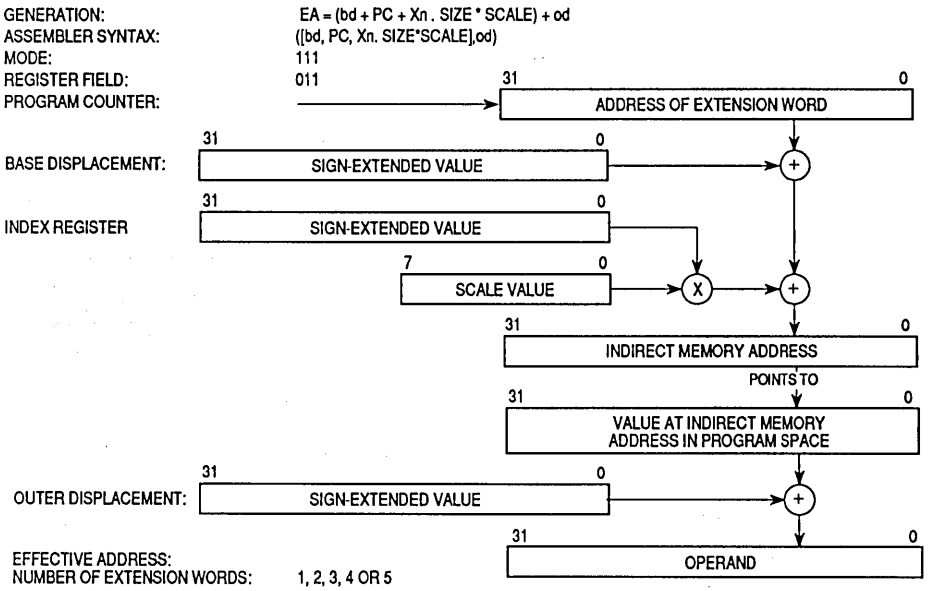
In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. However, the user must supply the assembler notation ZPC (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space without using the PC in calculating the effective address. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



2.4.15 Program Counter Memory Indirect Preindexed Mode

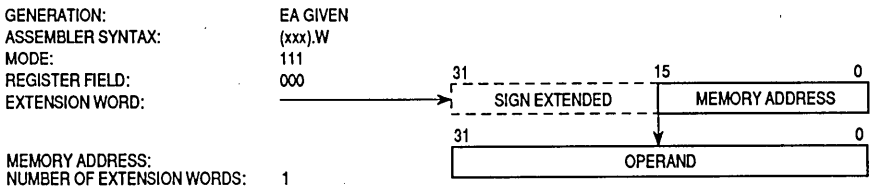
This mode is similar to the memory indirect preindexed mode described in **2.4.10 Memory Indirect Preindexed Mode**, but the PC is used as the base register. Both the operand and operand address are in memory. The controller calculates an intermediate indirect memory address by adding the PC contents, a base displacement (bd), and the scaled contents of an index register. The controller accesses a long word at that address and adds the optional outer displacement (od) to yield the effective address. The value of the PC is the address of the first extension word. The reference is a program space reference and is only allowed for reads (refer to **4.2 ADDRESS SPACE TYPES**).

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. However, the user must supply the assembler notation ZPC (zero value is taken for the PC) to indicate that the PC is not used. This allows the user to access the program space without using the PC in calculating the effective address. Both the base and outer displacements may be null, word, or long word. When a displacement is omitted or an element is suppressed, its value is taken as zero in the effective address calculation.



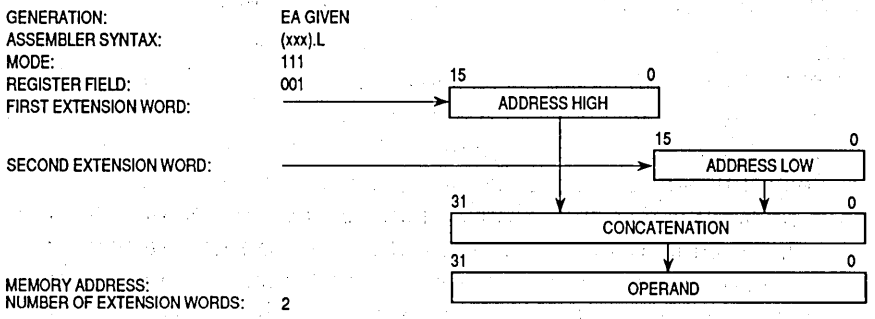
2.4.16 Absolute Short Addressing Mode

In this addressing mode, the operand is in memory, and the address of the operand is in the extension word. The 16-bit address is sign-extended to 32 bits before it is used.



2.4.17 Absolute Long Addressing Mode

In this mode, the operand is in memory, and the address of the operand occupies the two extension words following the instruction word in memory. The first extension word contains the high-order part of the address; the low-order part of the address is the second extension word.



2.4.18 Immediate Data

In this addressing mode, the operand is in one or two extension words:

Byte Operation
Operand is in the low-order byte of the extension word

Word Operation
Operand is in the extension word

Long-Word Operation
The high-order 16 bits of the operand are in the first extension word; the low-order 16 bits are in the second extension word.

Coprocessor instructions can support immediate data of any size. The instruction word is followed by as many extension words as are required.

GENERATION:	OPERAND GIVEN
ASSEMBLER SYNTAX:	#xxx
MODE FIELD:	111
REGISTER FIELD:	100
NUMBER OF EXTENSION WORDS:	1 or 2, EXCEPT FOR COPROCESSOR INSTRUCTIONS

2.5 EFFECTIVE ADDRESS ENCODING SUMMARY

Most of the addressing modes use one of the three formats shown in Figure 2-4. The single effective address instruction is in the format of the instruction word. The encoding of the mode field of this word selects the addressing mode. The register field contains the general register number or a value that selects the addressing mode when the mode field contains "111". Table 2-2 shows the encoding of these fields. Some indexed or indirect modes use the instruction word followed by the brief format extension word. Other indexed or indirect modes consist of the instruction word and the full format of extension words. The longest instruction for the MC68EC030 contains 10 extension words. It is a MOVE instruction with full format extension words for both the source and destination effective addresses and with 32-bit base displacements and 32-bit outer displacements for both addresses. However, coprocessor instructions can have any number of extension words. Refer to the coprocessor instruction formats in **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION**.

Single Effective Address Instruction Format

15	14	13	12	11	10	9	8	7	6	5	0
X	X	X	X	X	X	X	X	X	X	EFFECTIVE ADDRESS MODE	EFFECTIVE ADDRESS REGISTER

Brief Format Extension Word

15	14 12	11	10 9	8	7 0
D/A	REGISTER	W/L	SCALE	0	DISPLACEMENT

Full Format Extension Word(s)

15	14 12	11	10 9	8	7	6	5 4	3	2 0
D/A	REGISTER	W/L	SCALE	1	BS	IS	BD SIZE	0	I/IS
BASE DISPLACEMENT (0, 1, OR 2 WORDS)									
OUTER DISPLACEMENT (0, 1, OR 2 WORDS)									

Field	Definition
Instruction:	
Register	General Register Number
Extensions:	
Register	Index Register Number
D/A	Index Register Type 0 = Dn 1 = An
W/L	Word/Long-Word Index Size 0 = Sign-Extended Word 1 = Long Word
Scale	Scale Factor 00 = 1 01 = 2 10 = 4 11 = 8
BS	Base Register Suppress: 0 = Base Register Added 1 = Base Register Suppressed
IS	Index Suppress: 0 = Evaluate and Add Index Operand 1 = Suppress Index Operand
BD SIZE	Base Displacement Size: 00 = Reserved 01 = Null Displacement 10 = Word Displacement 11 = Long Displacement
I/IS	Index/Indirect Selection: Indirect and Indexing Operand Determined in Conjunction with Bit 6, Index Suppress

Figure 2-4. Effective Address Specification Formats

For effective addresses that use the full format, the index suppress (IS) bit and the index/indirect selection (I/IS) field determine the type of indexing and indirection. Table 2-1 lists the indexing and indirection operations corresponding to all combinations of IS and I/IS values.

Table 2-1. IS-I/IS Memory Indirection Encodings

IS	Index/Indirect	Operation
0	000	No Memory Indirection
0	001	Indirect Preindexed with Null Outer Displacement
0	010	Indirect Preindexed with Word Outer Displacement
0	011	Indirect Preindexed with Long Outer Displacement
0	100	Reserved
0	101	Indirect Postindexed with Null Outer Displacement
0	110	Indirect Postindexed with Word Outer Displacement
0	111	Indirect Postindexed with Long Outer Displacement
1	000	No Memory Indirection
1	001	Memory Indirect with Null Outer Displacement
1	010	Memory Indirect with Word Outer Displacement
1	011	Memory Indirect with Long Outer Displacement
1	100–111	Reserved

Effective address modes are grouped according to the use of the mode. They can be classified as follows:

Data — A data addressing effective address mode is one that refers to data operands.

Memory — A memory addressing effective address mode is one that refers to memory operands.

Alterable — An alterable addressing effective address mode is one that refers to alterable (writable) operands.

Control — A control addressing effective address mode is one that refers to memory operands without an associated size.

Table 2-2 shows the categories to which each of the effective addressing modes belong.

Table 2-2. Effective Addressing Mode Categories

Address Modes	Mode	Register	Data	Memory	Control	Alterable	Assembler Syntax
Data Register Direct	000	reg. no.	X	—	—	X	Dn
Address Register Direct	001	reg. no.	—	—	—	X	An
Address Register Indirect	010	reg. no.	X	X	X	X	(An)
Address Register Indirect with Postincrement	011	reg. no.	X	X	—	X	(An)+
Address Register Indirect with Predecrement	100	reg. no.	X	X	—	X	–(An)
Address Register Indirect with Displacement	101	reg. no.	X	X	X	X	(d_{16},An)
Address Register Indirect with Index (8-Bit Displacement)	110	reg. no.	X	X	X	X	(d_8,An,Xn)
Address Register Indirect with Index (Base Displacement)	110	reg. no.	X	X	X	X	(bd,An,Xn)
Memory Indirect Postindexed	110	reg. no.	X	X	X	X	([bd,An],Xn,od)
Memory Indirect Preindexed	110	reg. no.	X	X	X	X	([bd,An,Xn],od)
Absolute Short	111	000	X	X	X	X	(xxx).W
Absolute Long	111	001	X	X	X	X	(xxx).L
Program Counter Indirect with Displacement	111	010	X	X	X	—	(d_{16},PC)
Program Counter Indirect with Index (8-Bit) Displacement	111	011	X	X	X	—	(d_8,PC,Xn)
Program Counter Indirect with Index (Base Displacement)	111	011	X	X	X	—	(bd,PC,Xn)
PC Memory Indirect Postindexed	111	011	X	X	X	—	([bd,PC],Xn,od
PC Memory Indirect Preindexed	111	011	X	X	X	—	([bd,PC,Xn],od)
Immediate	111	100	X	X	—	—	#⟨data⟩

These categories are sometimes combined, forming new categories that are more restrictive. Two combined classifications are alterable memory or data alterable. The former refers to those addressing modes that are both alterable and memory addresses, and the latter refers to addressing modes that are both data and alterable.

2.6 PROGRAMMER'S VIEW OF ADDRESSING MODES

Extensions to the indexed addressing modes, indirection, and full 32-bit displacements provide additional programming capabilities for the MC68020, MC68030, and the MC68EC030. This section describes addressing techniques that exploit these capabilities and summarizes the addressing modes from a programming point of view.

Several of the addressing techniques described in this section use data registers and address registers interchangeably. While the MC68EC030 provides this capability, its performance has been optimized for addressing with address registers. The performance of a program that uses address registers in address calculations is superior to that of a program that similarly uses data registers. The specification of addresses with data registers should be used sparingly (if at all), particularly in programs that require maximum performance.

2.6.1 Addressing Capabilities

In the MC68020, MC68030, and the MC68EC030, setting the base register suppress (BS) bit in the full format extension word (see Figure 2-4) suppresses use of the base address register in calculating the effective address. This allows any index register to be used in place of the base register. Since any of the data registers can be index registers, this provides a data register indirect form (Dn). The mode could be called register indirect (Rn) since either a data register or an address register can be used. This addressing mode is an extension to the M68000 Family because the MC68020, MC68030, and MC68EC030 can use both the data registers and the address registers to address memory. The capability of specifying the size and scale of an index register (Xn.SIZE*SCALE) in these modes provides additional addressing flexibility. Using the SIZE parameter, either the entire contents of the index register can be used, or the least significant word can be sign-extended to provide a 32-bit index value (refer to Figure 2-5).

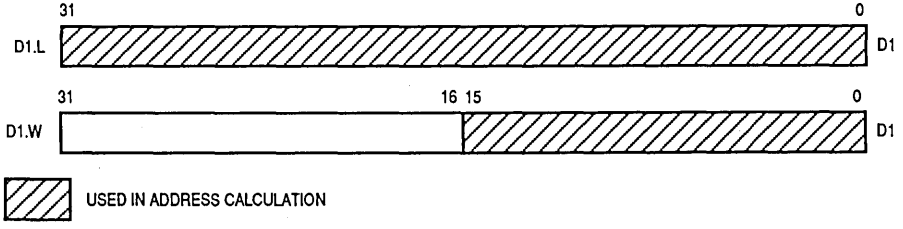


Figure 2-5. Using SIZE in the Index Selection

For the MC68020, MC68030, and the MC68EC030, the register indirect modes can be extended further. Since displacements can be 32 bits wide, they can represent absolute addresses or the results of expressions that contain absolute addresses. This allows the general register indirect form to be (bd,Rn) or (bd,An,Rn) when the base register is not suppressed. Thus, an absolute address can be directly indexed by one or two registers (refer to Figure 2-6).

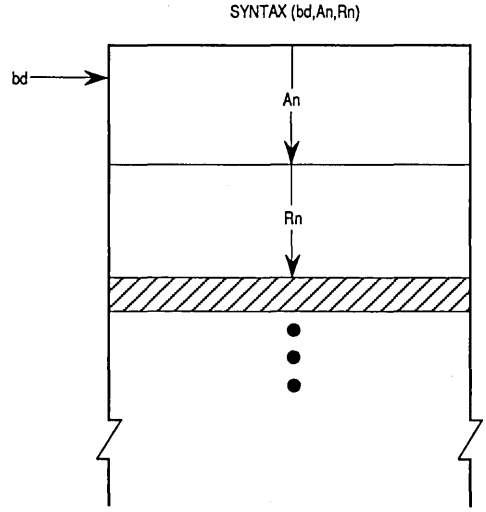


Figure 2-6. Using Absolute Address with Indexes

Scaling provides an optional shifting of the value in an index register to the left by zero, one, two, or three bits before using it in the effective address calculation (the actual value in the index register remains unchanged). This is equivalent to multiplying the register by one, two, four, or eight for direct subscripting into an array of elements of corresponding size using an arithmetic value residing in any of the 16 general registers. Scaling does not add to the effective address calculation time. However, when combined with the appropriate derived modes, it produces additional capabilities. Arrayed structures can be addressed absolutely and then subscripted, (bd,Rn*scale), for example. Optionally, an address register that contains a dynamic displacement can be included in the address calculation (bd,An,Rn*scale). Another variation that can be derived is (An,Rn*scale). In the first case, the array address is the sum of the contents of a register and a displacement, as shown in Figure 2-7. In the second example, An contains the address of an array, and Rn contains a subscript.

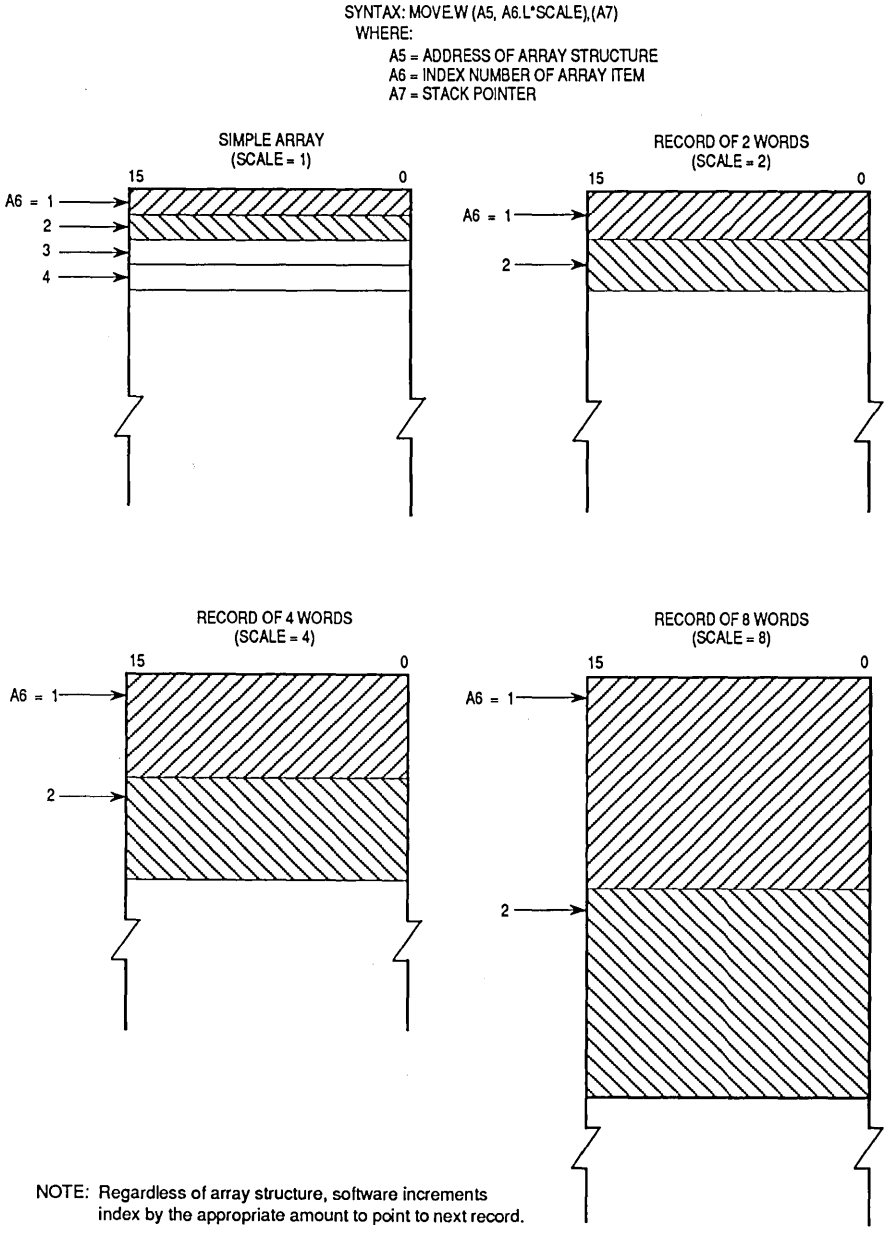


Figure 2-7. Addressing Array Items

The memory indirect addressing modes use a long-word pointer in memory to access an operand. Any of the modes previously described can be used

to address the memory pointer. Because the base and index registers can both be suppressed, the displacement acts as an absolute address, providing indirect absolute memory addressing (refer to Figure 2-8).

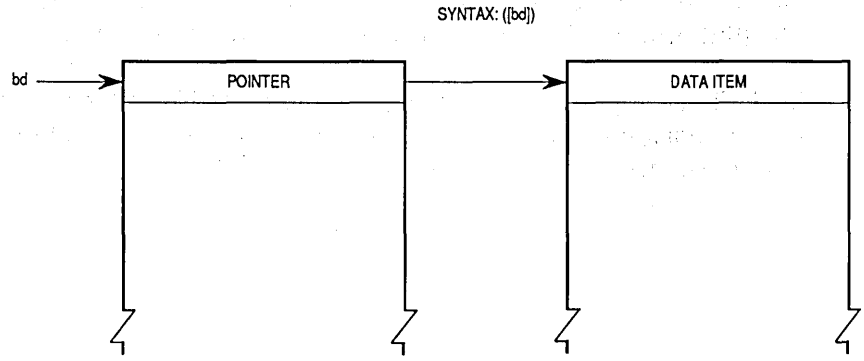


Figure 2-8. Using Indirect Absolute Memory Addressing

The outer displacement (od) available in the memory indirect modes is added to the pointer in memory. The syntax for these modes is ([bd,An],Xn,od) and ([bd,An,Xn],od). When the pointer is the address of a structure in memory and the outer displacement is the offset of an item in the structure, the memory indirect modes can access the item efficiently (refer to Figure 2-9).

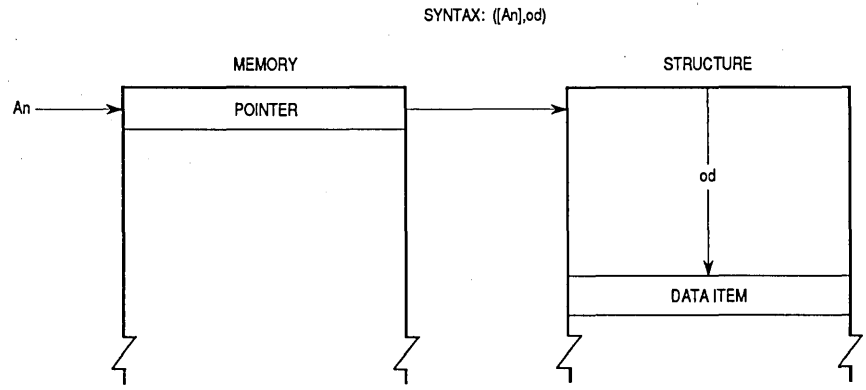


Figure 2-9. Accessing an Item in a Structure Using Pointer

Memory indirect addressing modes are used with a base displacement in five basic forms:

1. [bd,An] — Indirect, suppressed index register
2. ([bd,An,Xn]) — Preindexed indirect
3. ([bd,An],Xn) — Postindexed indirect
4. ([bd,An,Xn],od) — Preindexed indirect with outer displacement
5. ([bd,An],Xn,od) — Postindexed indirect with outer displacement

The indirect, suppressed index register mode (see Figure 2-10) uses the contents of register An as an index to the pointer located at the address specified by the displacement. The actual data item is at the address in the selected pointer.

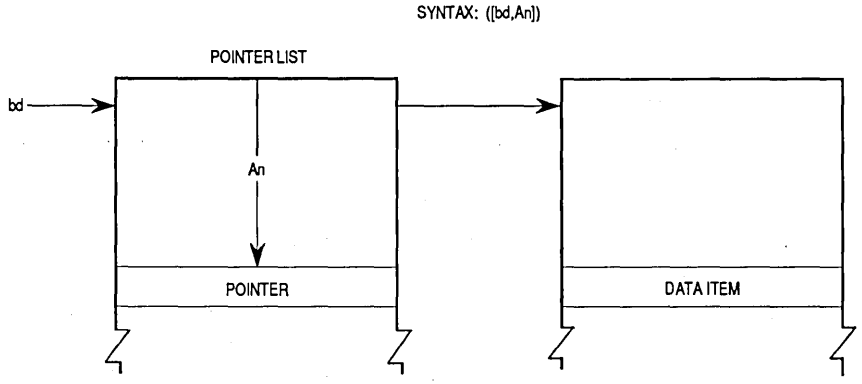


Figure 2-10. Indirect Addressing, Suppressed Index Register

The preindexed indirect mode (see Figure 2-11) uses the contents of An as an index to the pointer list structure at the displacement. Register Xn is the index to the pointer, which contains the address of the data item.

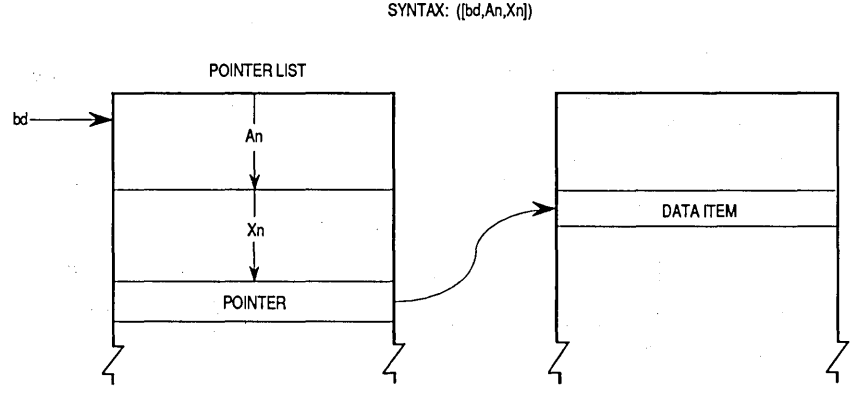


Figure 2-11. Preindexed Indirect Addressing

The postindexed indirect mode (see Figure 2-12) uses the contents of An as an index to the pointer list at the displacement. Register Xn is used as an index to the structure of data items located at the address specified by the pointer. Figure 2-13 shows the preindexed indirect addressing with outer displacement mode.

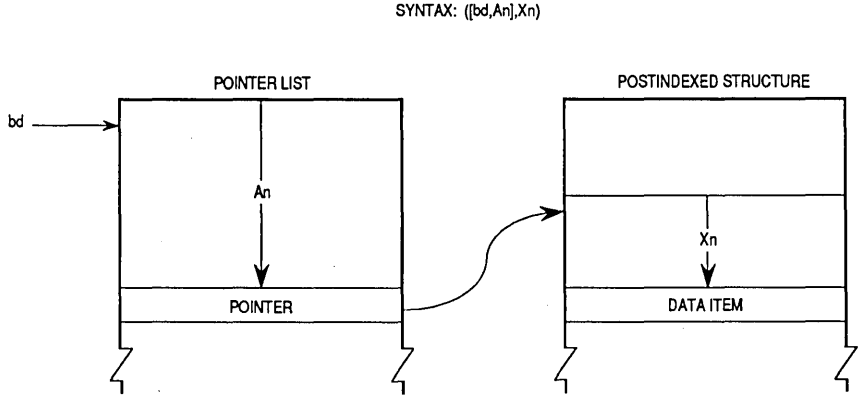


Figure 2-12. Postindexed Indirect Addressing

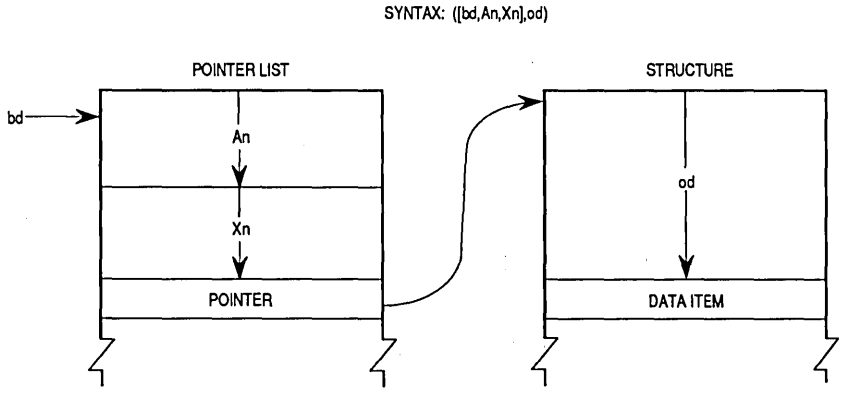


Figure 2-13. Preindexed Indirect Addressing with Outer Displacement

The postindexed indirect mode with outer displacement (see Figure 2-14) uses the contents of An as an index to the pointer list at the displacement. Register Xn is used as an index to the structure of data structures at the address in the pointer. The outer displacement (od) is the displacement of the data item within the selected data structure.

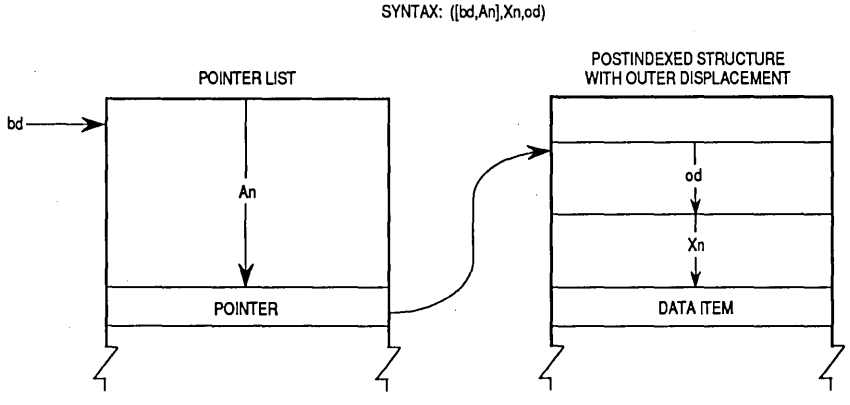


Figure 2-14. Postindexed Indirect Addressing with Outer Displacement

2.6.2 General Addressing Mode Summary

The addressing modes described in the previous section are derived from specific combinations of options in the indexing mode or a selection of two alternate addressing modes. For example, the addressing mode called register indirect (Rn) assembles as the address register indirect if the register is an address register. If Rn is a data register, the assembler uses the address register indirect with index mode using the data register as the indirect register and suppresses the address register by setting the base suppress bit in the effective address specification. Assigning an address register as Rn provides higher performance than using a data register as Rn. Another case is (bd,An), which selects an addressing mode depending on the size of the displacement. If the displacement is 16 bits or less, the address register indirect with displacement mode (d_{16},An) is used. When a 32-bit displacement is required, the address register indirect with index (bd,An,Xn) is used with the index register suppressed.

It is useful to examine the derived addressing modes available to a programmer (without regard to the MC68EC030 effective addressing mode actually encoded) because the programmer need not be concerned about these decisions. The assembler can choose the more efficient addressing mode to encode.

In the list of derived addressing modes that follows, common programming terms are used. The following definitions apply:

pointer — Long-word value in a register or in memory which represents an address.

base — A pointer combined with a displacement to represent an address.

index — A constant or variable value added into an effective address calculation. A constant index is a displacement. A variable index is always represented by a register containing the value.

disp — Displacement, a constant index.

subscript — The use of any of the data or address registers as a variable index subscript into arrays of items 1, 2, 4, or 8 bytes in size.

relative — An address calculated from the program counter contents. The address is position independent and is in program space. All other addresses but psaddr are in data space.

addr — An absolute address.

psaddr — An absolute address in program space. All other addresses but PC relative are in data space.

preindexed — All modes from absolute address through program counter relative.

postindexed — Any of the following modes:

addr	— Absolute address in data space
psaddr,ZPC	— Absolute address in program space
An	— Register pointer
disp,An	— Register pointer with constant displacement
addr,An	— Absolute address with single variable name
disp,PC	— Simple PC relative

The addressing modes defined in programming terms, which are derivations of the addressing modes provided by the MC68EC030 architecture, are as follows:

Immediate Data — #data:
The data is a constant located in the instruction stream.

Register Direct — Rn:
The contents of a register contain the operand.

Scanning Modes:
(An)+
Address register pointer automatically incremented after use.

−(An)
Address register pointer automatically decremented before use.

Absolute Address:

(addr)

Absolute address in data space.

(psaddr,ZPC)

Absolute address in program space. Symbol ZPC suppresses the PC, but retains PC relative mode to directly access the program space.

Register Pointer:

(Rn)

Register as a pointer.

(disp,Rn)

Register as a pointer with constant index (or base address).

Indexing:

(An,Rn)

Register pointer An with variable index Rn.

(disp,An,Rn)

Register pointer with constant and variable index (or a base address with a variable index).

(addr,Rn)

Absolute address with variable index.

(addr,An,Rn)

Absolute address with two variable indexes.

Subscripting:

(An,Rn*scale)

Address register pointer subscript.

(disp,An,Rn*scale)

Address register pointer subscript with constant displacement (or base address with subscript).

(addr, Rn*scale)

Absolute address with subscript.

(addr,An,Rn*scale)

Absolute address subscript with variable index.

Program Relative:

(disp,PC)
Simple PC relative.

(disp,PC,Rn)
PC relative with variable index.

(disp,PC,Rn*scale)
PC relative with subscript.

Memory Pointer:

([preindexed])
Memory pointer directly to data operand.

([preindexed],disp)
Memory pointer as base with displacement to data operand.

([postindexed],Rn)
Memory pointer with variable index.

([postindexed],disp,Rn)
Memory pointer with constant and variable index.

([postindexed],Rn*scale)
Memory pointer subscripted.

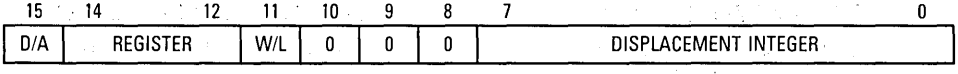
([postindexed], disp, Rn*scale)
Memory pointer subscripted with constant index.

2.7 M68000 FAMILY ADDRESSING COMPATIBILITY

Programs can be easily transported from one member of the M68000 Family to another in an upward compatible fashion. The user object code of each early member of the family is upward compatible with newer members and can be executed on the newer embedded controller without change. The address extension word(s) are encoded with the information that allows the MC68020/MC68030/MC68EC030 to distinguish the new address extensions to the basic M68000 Family architecture. The address extension words for the early MC68000/MC68008/MC68010 microprocessors and for the newer 32-bit MC68020/MC68030 microprocessors and MC68EC030 embedded controller are shown in Figure 2-15. Notice the encoding for SCALE used by the MC68EC030 is a compatible extension of the M68000 architecture. A value of zero for SCALE is the same encoding for both extension words; hence, software that uses this encoding is both upward and downward compatible across all processors in the product line. However, the other values of SCALE are not found in both extension formats; thus, while software can be easily migrated in an upward compatible direction, only nonscaled addressing is

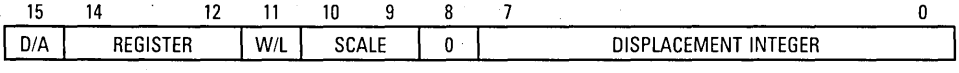
supported in a downward fashion. If the MC68000 were to execute an instruction that encoded a scaling factor, the scaling factor would be ignored and not access the desired memory address. The earlier microprocessors have no knowledge of the extension word formats implemented by newer processors; while they do detect illegal instructions, they do not decode invalid encodings of the extension words as exceptions.

MC68000/MC68008/MC68010 Address Extension Word



D/A: 0 = Data Register Select
1 = Address Register Select
W/L: 0 = Word-Sized Operation
1 = Long-Word-Sized Operation

MC68020/MC68030/MC68EC030 Address Extension Word



D/A: 0 = Data Register Select
1 = Address Register Select
W/L: 0 = Word-Sized Operation
1 = Long-Word-Sized Operation
SCALE: 00 = Scale Factor 1 (Compatible with MC68000)
01 = Scale Factor 2 (Extension to MC68000)
10 = Scale Factor 4 (Extension to MC68000)
11 = Scale Factor 8 (Extension to MC68000)

Figure 2-15. M68000 Family Address Extension Words

2.8 OTHER DATA STRUCTURES

Stacks and queues are widely used data structures. The MC68EC030 implements a system stack and also provides instructions that support the use of user stacks and queues.

2.8.1 System Stack

Address register seven (A7) is used as the system stack pointer (SP). Any of the three system stack registers is active at any one time. The M and S bits of the status register determine which stack pointer is used. When S=0 indicating user mode (user privilege level), the user stack pointer (USP) is

the active system stack pointer, and the master and interrupt stack pointers cannot be referenced. When S = 1 indicating supervisor mode (at supervisor privilege level) and M = 1, the master stack pointer (MSP) is the active system stack pointer. When S = 1 and M = 0, the interrupt stack pointer (ISP) is the active system stack pointer. This mode is the MC68EC030 default mode after reset and corresponds to the MC68000, MC68008, and MC68010 supervisor mode. The term supervisor stack pointer (SSP) refers to the master or interrupt stack pointers, depending on the state of the M bit. When M = 1, the term SSP (or A7) refers to the MSP address register. When M = 0, the term SSP (or A7) refers to the ISP address register. The active system stack pointer is implicitly referenced by all instructions that use the system stack. Each system stack fills from high to low memory.

A subroutine call saves the program counter on the active system stack, and the return restores it from the active system stack. During the processing of traps and interrupts, both the program counter and the status register are saved on the supervisor stack (either master or interrupt). Thus, the execution of supervisor code is independent of user code and the condition of the user stack; conversely, user programs use the user stack pointer independently of supervisor stack requirements.

To keep data on the system stack aligned for maximum efficiency, the active stack pointer is automatically decremented or incremented by two for all byte-sized operands moved to or from the stack. In long-word-organized memory, aligning the stack pointer on a long-word address significantly increases the efficiency of stacking exception frames, subroutine calls and returns, and other stacking operations.

2.8.2 User Program Stacks

The user can implement stacks with the address register indirect with postincrement and predecrement addressing modes. With address register An (n = 0–6), the user can implement a stack that is filled either from high to low memory or from low to high memory. Important considerations are as follows:

- Use the predecrement mode to decrement the register before its contents are used as the pointer to the stack.
- Use the postincrement mode to increment the register after its contents are used as the pointer to the stack.
- Maintain the stack pointer correctly when byte, word, and long-word items are mixed in these stacks.

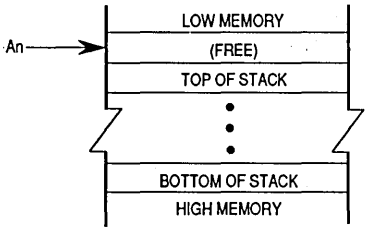
To implement stack growth from high to low memory, use:

–(An) to push data on the stack,

(An)+ to pull data from the stack.

2

For this type of stack, after either a push or a pull operation, register An points to the top item on the stack. This is illustrated as:

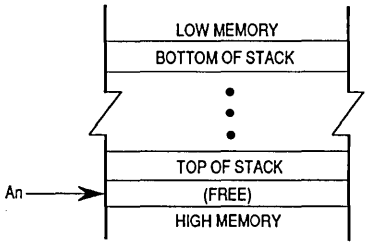


To implement stack growth from low to high memory, use:

(An)+ to push data on the stack,

–(An) to pull data from the stack.

In this case, after either a push or pull operation, register An points to the next available space on the stack. This is illustrated as:



2.8.3 Queues

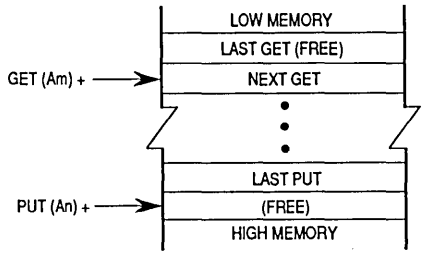
The user can implement queues with the address register indirect with post-increment or predecrement addressing modes. Using a pair of address registers (two of A0–A6), the user can implement a queue which is filled either from high to low memory or from low to high memory. Two registers are used because queues are pushed from one end and pulled from the other. One register, An, contains the "put" pointer; the other, Am, the "get" pointer.

To implement growth of the queue from low to high memory, use:

(An)+ to put data into the queue,

(Am)+ to get data from the queue.

After a "put" operation, the "put" address register points to the next available space in the queue, and the unchanged "get" address register points to the next item to be removed from the queue. After a "get" operation, the "get" address register points to the next item to be removed from the queue, and the unchanged "put" address register points to the next available space in the queue. This is illustrated as:



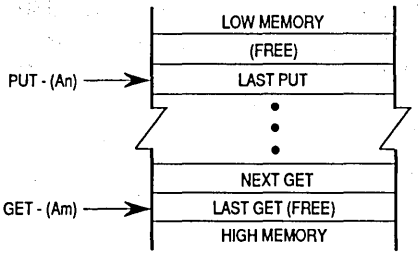
To implement the queue as a circular buffer, the relevant address register should be checked and adjusted, if necessary, before performing the "put" or "get" operation. The address register is adjusted by subtracting the buffer length (in bytes) from the register.

To implement growth of the queue from high to low memory, use:

–(An) to put data into the queue,

–(Am) to get data from the queue.

After a "put" operation, the "put" address register points to the last item placed in the queue, and the unchanged "get" address register points to the last item removed from the queue. After a "get" operation, the "get" address register points to the last item removed from the queue, and the unchanged "put" address register points to the last item placed in the queue. This is illustrated as:



To implement the queue as a circular buffer, the "get" or "put" operation should be performed first, and then the relevant address register should be checked and adjusted, if necessary. The address register is adjusted by adding the buffer length (in bytes) to the register contents.

SECTION 3
INSTRUCTION SET SUMMARY

This section briefly describes the MC68EC030 instruction set. Refer to the MC68000PM/AD, *MC68000 Programmer's Reference Manual*, for complete details on the MC68EC030 instruction set. **APPENDIX A MC68EC030 NEW INSTRUCTIONS** has details on PMOVE and PTEST instructions for the MC68EC030. The MC68EC030 executes all instructions the same as the MC68030, except the MC68EC030 does not execute all memory management unit instructions.

The following paragraphs include descriptions of the instruction format and the operands used by instructions, followed by a summary of the instruction set. The integer condition codes and floating-point details are discussed. Programming examples for selected instructions are also presented.

3.1 INSTRUCTION FORMAT

All MC68EC030 instructions consist of at least one word; some have as many as 11 words (see Figure 3-1). The first word of the instruction, called the operation word, specifies the length of the instruction and the operation to be performed. The remaining words, called extension words, further specify the instruction and operands. These words may be floating-point command words, conditional predicates, immediate operands, extensions to the effective address mode specified in the operation word, branch displacements, bit number or bit field specifications, special register specifications, trap operands, pack/unpack constants, or argument counts.

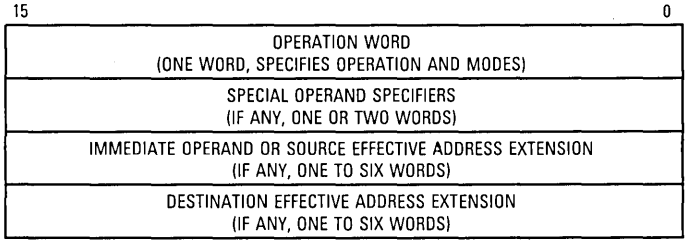


Figure 3-1. Instruction Word General Format

Besides the operation code, which specifies the function to be performed, an instruction defines the location of every operand for the function. Instructions specify an operand location in one of three ways:

1. Register Specification — A register field of the instruction contains the number of the register.
2. Effective Address — An effective address field of the instruction contains address mode information.
3. Implicit Reference — The definition of an instruction implies the use of specific registers.

The register field within an instruction specifies the register to be used. Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used. **SECTION 1 INTRODUCTION** contains register information.

Effective address information includes the registers, displacements, and absolute addresses for the effective address mode. **SECTION 2 DATA ORGANIZATION AND ADDRESSING CAPABILITIES** describes the effective address modes in detail.

Certain instructions operate on specific registers. These instructions imply the required registers.

3.2 INSTRUCTION SUMMARY

The instructions form a set of tools to perform the following operations:

Data Movement	Bit Field Manipulation
Integer Arithmetic	Binary-Coded Decimal Arithmetic
Logical	Program Control
Shift and Rotate	System Control
Bit Manipulation	Multiprocessor Communications

Each instruction type is described in detail in the following paragraphs.

The following notations are used in this section. In the operand syntax statements of the instruction definitions, the operand on the right is the destination operand.

An = any address register, A7–A0
Dn = any data register, D7–D0
Rn = any address or data register
CCR = condition code register (lower byte of status register)
cc = condition codes from CCR
SR = status register
SP = active stack pointer
USP = user stack pointer
ISP = supervisor/interrupt stack pointer
MSP = supervisor/master stack pointer
SSP = supervisor (master or interrupt) stack pointer
DFC = destination function code register
SFC = source function code register
Rc = control register (VBR, SFC, DFC, CACR)
MRc = ACR control register (AC0, AC1)
B, W, L = specifies a signed integer data type (twos complement) of byte, word, or long word
S = single-precision real data format (32 bits)
D = double-precision real data format (64 bits)
X = extended-precision real data format (96 bits, 16 bits unused)
P = packed BCD real data format (96 bits, 12 bytes)
FPm, FPn = any floating-point data register, FP7–FP0
PFcr = floating-point system control register (FPCR, FPSR, or FPIAR)
k = a twos-complement signed integer (−64 to +17) that specifies the format of a number to be stored in the packed BCD format
d = displacement; d_{16} is a 16-bit displacement
<ea> = effective address
list = list of registers, for example D3–D0
#<data> = immediate data; a literal integer
{offset:width} = bit field selection
label = assemble program label
[m] = bit m of an operand
[m:n] = bits m through n of operand
X = extend (X) bit in CCR
N = negative (N) bit in CCR
Z = zero (Z) bit in CCR

V = overflow (V) bit in CCR
C = carry (C) bit in CCR
+ = arithmetic addition or postincrement indicator
– = arithmetic subtraction or predecrement indicator
× = arithmetic multiplication
÷ = arithmetic division or conjunction symbol
~ = invert; operand is logically complemented
Λ = logical AND
V = logical OR
⊕ = logical exclusive OR
Dc = data register, D7–D0 used during compare
Du = data register, D7–D0 used during update
Dr, Dq = data registers, remainder or quotient of divide
Dh, Dl = data registers, high- or low-order 32 bits of product
MSW = most significant word
LSW = least significant word
MSB = most significant bit
FC = function code
{R/W} = read or write indicator
[An] = address extensions

3.2.1 Data Movement Instructions

The MOVE instructions with their associated addressing modes are the basic means of transferring and storing addresses and data. MOVE instructions transfer byte, word, and long-word operands from memory to memory, memory to register, register to memory, and register to register. Address movement instructions (MOVE or MOVEA) transfer word and long-word operands and ensure that only valid address manipulations are executed. In addition to the general MOVE instructions, there are several special data movement instructions: move multiple registers (MOVEM), move peripheral data (MOVEP), move quick (MOVEQ), exchange registers (EXG), load effective address (LEA), push effective address (PEA), link stack (LINK), and unlink stack (UNLK).

Table 3-1 is a summary of the integer and floating-point data movement operations.

Table 3-1. Data Movement Operations

Instruction	Operand Syntax	Operand Size	Operation
EXG	Rn, Rn	32	Rn ⇔ Rn
LEA	<ea>,An	32	<ea> ⇨ An
LINK	An,#<d>	16,32	Sp − 4 ⇨ SP; An ⇨ (SP); SP ⇨ An, SP + D ⇨ SP
MOVE MOVEA	<ea>,<ea> <ea>,An	8,16,32 16,32 ⇨ 32	source ⇨ destination
MOVEM	list,<ea> <ea>,list	16,32 16,32 ⇨ 32	listed registers ⇨ destination source ⇨ listed registers
MOVEP	Dn, (d_{16},An) (d_{16},An),Dn	16,32	Dn[31:24] ⇨ (An + d); Dn[23:16] ⇨ An + d + 2); Dn[15:8] ⇨ (An + d + 4); Dn[7:0] ⇨ (An + d + 6) (An + d) ⇨ Dn[31:24]; (An + d + 2) ⇨ Dn[23:16]; (An + d + 4) ⇨ Dn[15:8]; (An + d + 6) ⇨ Dn[7:0]
MOVEQ	#<data>,Dn	8 ⇨ 32	immediate data ⇨ destination
PEA	<ea>	32	SP − 4 ⇨ SP; <ea> ⇨ (SP)
UNLK	An	32	An ⇨ SP; (SP) ⇨ An; SP + 4 ⇨ SP

3.2.2 Integer Arithmetic Instructions

The integer arithmetic operations include the four basic operations of add (ADD), subtract (SUB), multiply (MUL), and divide (DIV) as well as arithmetic compare (CMP, CMPM, CMP2), clear (CLR), and negate (NEG). The instruction set includes ADD, CMP, and SUB instructions for both address and data operations with all operand sizes valid for data operations. Address operands consist of 16 or 32 bits. The clear and negate instructions apply to all sizes of data operands.

Signed and unsigned MUL and DIV instructions include:

- Word multiply to produce a long-word product
- Long-word multiply to produce a long-word or quad-word product
- Division of a long word divided by a word divisor (word quotient and word remainder)
- Division of a long word or quad word dividend by a long-word divisor (long-word quotient and long-word remainder)

A set of extended instructions provides multiprecision and mixed-size arithmetic. These instructions are add extended (ADDX), subtract extended (SUBX), sign extended (EXT), and negate binary with extend (NEGX). Refer to Table 3-2 for a summary of the integer arithmetic operations.

Table 3-2. Integer Arithmetic Operations

Instruction	Operand Syntax	Operand Size	Operation
ADD	Dn,⟨ea⟩	8, 16, 32	source + destination ⇨ destination
	⟨ea⟩,Dn	8, 16, 32	
ADDA	⟨ea⟩,An	16, 32	
ADDI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data + destination ⇨ destination
ADDQ	#⟨data⟩,⟨ea⟩	8, 16, 32	
ADDX	Dn,Dn	8, 16, 32	source + destination + X ⇨ destination
	−(An), −(An)	8, 16, 32	
CLR	⟨ea⟩	8, 16, 32	0 ⇨ destination
CMP	⟨ea⟩,Dn	8, 16, 32	destination − source
CMPA	⟨ea⟩,An	16, 32	
CMPI	#⟨data⟩,⟨ea⟩	8, 16, 32	destination − immediate data
CMPM	(An)+,(An)+	8, 16, 32	destination − source
CMP2	⟨ea⟩,Rn	8, 16, 32	lower bound ⟨ = Rn ⟨ = upper bound
DIVS/DIVU	⟨ea⟩,Dn	32/16 ⇨ 16:16	destination/source ⇨ destination (signed or unsigned)
	⟨ea⟩,Dr:Dq	64/32 ⇨ 32:32	
	⟨ea⟩,Dq	32/32 ⇨ 32	
DIVSL/DIVUL	⟨ea⟩,Dr:Dq	32/32 ⇨ 32:32	
EXT	Dn	8 ⇨ 16	sign extended destination ⇨ destination
	Dn	16 ⇨ 32	
EXTB	Dn	8 ⇨ 32	
MULS/MULU	⟨ea⟩,Dn	16 × 16 ⇨ 32	source × destination ⇨ destination (signed or unsigned)
	⟨ea⟩,Dl	32 × 32 ⇨ 32	
	⟨ea⟩,Dh:Dl	32 × 32 ⇨ 64	
NEG	⟨ea⟩	8, 16, 32	0 − destination ⇨ destination
NEGX	⟨ea⟩	8, 16, 32	0 − destination − X ⇨ destination
SUB	⟨ea⟩,Dn	8, 16, 32	destination = source ⇨ destination
	Dn,⟨ea⟩	8, 16, 32	
SUBA	⟨ea⟩,An	16, 32	
SUBI	#⟨data⟩,⟨ea⟩	8, 16, 32	destination − immediate data ⇨ destination
SUBQ	#⟨data⟩,⟨ea⟩	8, 16, 32	
SUBX	Dn,Dn	8, 16, 32	destination − source − X ⇨ destination
	−(An), −(An)	8, 16, 32	

3.2.3 Logical Instructions

The logical operation instructions (AND, OR, EOR, and NOT) perform logical operations with all sizes of integer data operands. A similar set of immediate instructions (ANDI, ORI, and EORI) provide these logical operations with all sizes of immediate data. The TST instruction compares the operand with zero arithmetically, placing the result in the condition code register. Table 3-3 summarizes the logical operations.

Table 3-3. Logical Operations

Instruction	Operand Syntax	Operand Size	Operation
AND	⟨ea⟩,Dn Dn,⟨ea⟩	8, 16, 32 8, 16, 32	source ∧ destination ⇨ destination
ANDI	#<data>,<ea>	8, 16,32	immediate data ∧ destination ⇨ destination
EOR	Dn,<data>,<ea>	8, 16, 32	source ⊕ destination ⇨ destination
EORI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data ⊕ destination ⇨ destination
NOT	⟨ea⟩	8, 16, 32	~ destination ⇨ destination
OR	⟨ea⟩,Dn Dn,⟨ea⟩	8, 16, 32 8, 16, 32	source V destination ⇨ destination
ORI	#⟨data⟩,⟨ea⟩	8, 16, 32	immediate data V destination ⇨ destination
TST	⟨ea⟩	8, 16, 32	source — 0 to set condition codes

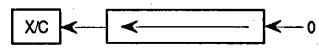
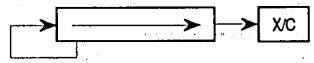
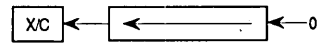
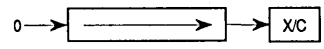
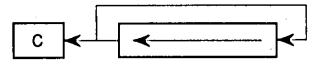
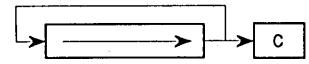
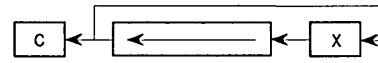
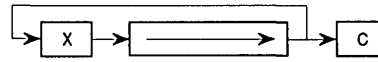
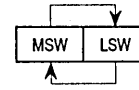
3.2.4 Shift and Rotate Instructions

The arithmetic shift instructions (ASR and ASL) and logical shift instructions (LSR and LSL) provide shift operations in both directions. The ROR, ROL, ROXR, and ROXL instructions perform rotate (circular shift) operations, with and without the extend bit. All shift and rotate operations can be performed on either registers or memory.

Register shift and rotate operations shift all operand sizes. The shift count may be specified in the instruction operation word (to shift from 1–8 places) or in a register (modulo 64 shift count).

Memory shift and rotate operations shift word-length operands one bit position only. The SWAP instruction exchanges the 16-bit halves of a register. Performance of shift/rotate instructions is enhanced so that use of the ROR and ROL instructions with a shift count of eight allows fast byte swapping. Table 3-4 is a summary of the shift and rotate operations.

Table 3-4. Shift and Rotate Operations

Instruction	Operand Syntax	Operand Size	Operation
ASL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ASR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
LSL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
LSR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROXL	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
ROXR	Dn,Dn #⟨data⟩,Dn ⟨ea⟩	8, 16, 32 8, 16, 32 16	
SWAP	Dn	32	

3.2.5 Bit Manipulation Instructions

Bit manipulation operations are accomplished using the following instructions: bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG). All bit manipulation operations can be performed on either registers or memory. The bit number is specified as immediate data or in a data register. Register operands are 32 bits long, and memory operands are 8 bits long. In Table 3-5, the summary of the bit manipulation operations, Z refers to bit 2, the zero bit of the status register.

Table 3-5. Bit Manipulation Operations

Instruction	Operand Syntax	Operand Size	Operation
BCHG	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) ➧ Z ➧ bit of destination
BCLR	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) ➧ Z; 0 ➧ bit of destination
BSET	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) ➧ Z; 1 ➧ bit of destination
BTST	Dn,⟨ea⟩ #⟨data⟩,⟨ea⟩	8, 32 8, 32	~ (⟨bit number⟩ of destination) ➧ Z

3.2.6 Bit Field Instructions

The MC68EC030 supports variable-length bit field operations on fields of up to 32 bits. The bit field insert (BFINS) instruction inserts a value into a bit field. Bit field extract unsigned (BFEXTU) and bit field extract signed (BFEXTS) extract a value from the field. Bit field find first one (BFFFO) finds the first bit that is set in a bit field. Also included are instructions that are analogous to the bit manipulation operations; bit field test (BFTST), bit field test and set (BFSET), bit field test and clear (BFCLR), and bit field test and change (BFCHG). Table 3-6 is a summary of the bit field operations.

Table 3-6. Bit Field Operations

Instruction	Operand Syntax	Operand Size	Operation
BFCHG	⟨ea⟩ {offset:width}	1–32	~ Field ➧ Field
BFCLR	⟨ea⟩ {offset:width}	1–32	0's ➧ Field
BFEXTS	⟨ea⟩ {offset:width},Dn	1–32	Field ➧ Dn; Sign Extended
BFEXTU	⟨ea⟩ {offset:width},Dn	1–32	Field ➧ Dn; Zero Extended
BFFFO	⟨ea⟩ {offset:width},Dn	1–32	Scan for first bit set in field; offset ➧ Dn
BFINS	Dn,⟨ea⟩ {offset:width}	1–32	Dn ➧ Field
BFSET	⟨ea⟩ {offset:width}	1–32	1's ➧ Field
BFTST	⟨ea⟩ {offset:width}	1–32	Field MSB ➧ N; ~ (OR of all bits in field) ➧ Z

NOTE: All bit field instructions set the N and Z bits as shown for BFTST before performing the specified operation.

3.2.7 Binary-Coded Decimal Instructions

Five instructions support operations on binary-coded decimal (BCD) numbers. The arithmetic operations on packed BCD numbers are add decimal with extend (ABCD), subtract decimal with extend (SBCD), and negate decimal with extend (NBCD). PACK and UNPACK instructions aid in the conversion of byte encoded numeric data, such as ASCII or EBCDIC strings, to BCD data and vice versa. Table 3-7 is a summary of the BCD operations.

Table 3-7. BCD Operations

Instruction	Operand Syntax	Operand Size	Operation
ABCD	Dn,Dn −(An),−(An)	8 8	$source_{10} + destination_{10} + X \blacklozenge destination$
NBCD	⟨ea⟩	8	$0 - destination_{10} - X \blacklozenge destination$
PACK	−(An),−(An) #⟨data⟩ Dn,Dn,#⟨data⟩	16 ◆ 8 16 ◆ 8	unpackaged source + immediate data ◆ packed destination
SBCD	Dn,Dn −(An),−(An)	8 8	$destination_{10} - source_{10} - X \blacklozenge destination$
UNPK	−(An),−(An) #⟨data⟩ Dn,Dn,#⟨data⟩	8 ◆ 16 8 ◆ 16	packed source ◆ unpacked source unpacked source + immediate data ◆ unpacked destination

3.2.8 Program Control Instructions

A set of subroutine call and return instructions and conditional and unconditional branch instructions perform program control operations. The no operation instruction (NOP) may be used to force synchronization of the internal pipelines. Table 3-8 summarizes these instructions.

Table 3-8. Program Control Operations

Instruction	Operand Syntax	Operand Size	Operation
Integer and Floating-Point Conditional			
Bcc	<label>	8,16,32	if condition true, then PC + d ⇨ PC
DBcc	Dn,<label>	16	if condition false, then Dn − 1 ⇨ Dn if Dn ≠ −1, then PC + d ⇨ PC
Scc	<ea>	8	if condition true, then 1's ⇨ destination; else 0's ⇨ destination
Unconditional			
BRA	<label>	8,16,32	PC + d ⇨ PC
BSR	<label>	8,16,32	SP − 4 ⇨ SP; PC ⇨ (SP); PC + d ⇨ PC
JMP	<ea>	none	destination ⇨ PC
JSR	<ea>	none	SP − 4 ⇨ SP; PC ⇨ (SP); destination ⇨ PC
NOP	none	none	PC + 2 ⇨ PC
Returns			
RTD	#<d>	16	(SP) ⇨ PC; SP + 4 + d ⇨ SP
RTR	none	none	(SP) ⇨ CCR; SP + 2 ⇨ SP; (SP) ⇨ PC; SP + 4 ⇨ SP
RTS	none	none	(SP) ⇨ PC; SP + 4 ⇨ SP

Letters cc in the integer instruction mnemonics Bcc, DBcc, and Scc specify testing one of the following conditions:

CC — Carry clear
LS — Lower or same
CS — Carry set
LT — Less than
EQ — Equal
MI — Minus
F — Never true*
NE — Not equal
GE — Greater or equal
PL — Plus
GT — Greater than
T — Always true*
HI — Higher
VC — Overflow clear
LE — Less or equal
VS — Overflow set

*Not applicable to the Bcc or cpBcc instructions.

3.2.9 System Control Instructions

Privileged instructions, trapping instructions, and instructions that use or modify the condition code register (CCR) provide system control operations. Table 3-9 summarizes these instructions. The TRAPcc instruction uses the same conditional tests as the corresponding program control instructions. All of these instructions cause the controller to flush the instruction pipe.

Table 3-9. System Control Operations

Instruction	Operand Syntax	Operand Size	Operation
Privileged			
ANDI	#<data>,SR	16	immediate data ∧ SR ⇨ SR
EORI	#<data>,SR	16	immediate data ⊕ SR ⇨ SR
MOVE	<ea>,SR SR,<ea>	16 16	source ⇨ SR SR ⇨ destination
MOVE	USP,An An,USP	32 32	USP ⇨ An An ⇨ USP
MOVEC	Rc,Rn Rn,Rc	32 32	Rc ⇨ Rn Rn ⇨ Rc
MOVES	Rn,<ea> <ea>,Rn	8,16,32	Rn ⇨ destination using DFC source using SFC ⇨ Rn
ORI	#<data>,SR	16	immediate data V SR ⇨ SR
RESET	none	none	assert $\overline{\text{RESET}}$ line
RTE	none	none	(SP) ⇨ SR; SP+2 ⇨ SP; (SP) ⇨ PC; SP+4 ⇨ SP; Restore stack according to format
STOP	#<data>	16	immediate data ⇨ SR; STOP
Trap Generating			
BKPT	#<data>	none	run breakpoint cycle, then trap as illegal instruction
CHK	<ea>,Dn	16,32	if Dn<0 or Dn>(ea), then CHK exception
CHK2	<ea>,Rn	8,16,32	if Rn<lower bound or Rn>upper bound, the CHK exception
ILLEGAL	none	none	SSP −2 ⇨ SSP; Vector Offset ⇨ (SSP); SSP −4 ⇨ SSP; PC ⇨ (SSP); SSP −2 ⇨ SSP; SR ⇨ (SSP); Illegal Instruction Vector Address ⇨ PC
TRAP	#<data>	none	SSP −2 ⇨ SSP; Format and Vector Offset ⇨ (SSP) SSP −4 ⇨ SSP; PC ⇨ (SSP); SSP −2 ⇨ SSP; SR ⇨ (SSP); Vector Address ⇨ PC
TRAPcc	none #<data>	none 16,32	if cc true, then TRAP exception
TRAPV	none	none	if V then take overflow TRAP exception
Condition Code Register			
ANDI	#<data>,CCR	8	immediate data ∧ CCR ⇨ CCR
EORI	#<data>,CCR	8	immediate data ⊕ CCR ⇨ CCR
MOVE	<ea>,CCR CCR,<ea>	16 16	source ⇨ CCR CCR ⇨ destination
ORI	#<data>,CCR	8	immediate data V CCR ⇨ CCR

3.2.10 Access Control Unit Instructions

PTEST performs a search of the access control registers, storing results in the ACU status register. PMOVE loads and stores ACU registers. Table 3-10 summarizes these instructions.

Table 3-10. ACU Instructions

Instruction	Operand Syntax	Operand Size	Operation
PTEST	⟨Function Code⟩, ⟨ea⟩,{R/W}	none	Information about ACR into ACU status register
PMOVE	Rn,⟨ea⟩ ⟨ea⟩,Rn	16,32 16,32	Register n ⧫ Destination Source ⧫ Register n
PFLUSH	(An)	none	No effect
PFLUSH.N	(An)	none	No effect
PTEST	(An)	none	Information about ACR into MMU status register

3.2.11 Multiprocessor Instructions

The TAS, CAS, and CAS2 instructions coordinate the operations of processors in multiprocessing systems. These instructions use read-modify-write bus cycles to ensure uninterrupted updating of memory. Coprocessor instructions control the coprocessor operations. Table 3-11 lists these instructions.

Table 3-11. Multiprocessor Operations (Read-Modify-Write)

Instruction	Operand Syntax	Operand Size	Operation
Read-Modify-Write			
CAS	Dc,Du,<ea>	8,16,32	destination — Dc ⧫ CC; if Z then Du ⧫ destination else destination ⧫ Dc
CAS2	Dc1:Dc2, Du1:Du2, (Rn):(Rn)	8,16,32	dual operand CAS
TAS	<ea>	8	destination — 0; set condition codes; 1 ⧫ destination [7]
Coprocessor			
cpBcc	⟨label⟩	16, 32	if cpcc true then pc + d ⧫ PC
cpDBcc	⟨label⟩,Dn	16	if cpcc false then Dn − 1 ⧫ Dn if Dn ≠ − 1, then PC + d ⧫ PC
cpGEN	User Defined	User Defined	operand ⧫ coprocessor
cp RESTORE	⟨ea⟩	none	restore coprocessor state from ⟨ea⟩
cpSAVE	⟨ea⟩	none	save coprocessor state at ⟨ea⟩
cpScc	⟨ea⟩	8	if cpcc true, then 1's ⧫ destination; else 0's ⧫ destination
cpTRAPcc	none #⟨data⟩	none 16, 32	if cpcc true then TRAPcc exception

3.3 INTEGER CONDITION CODES

The CCR portion of the SR contains five bits which indicate the results of many integer instructions. Program and system control instructions use certain combinations of these bits to control program and system flow.

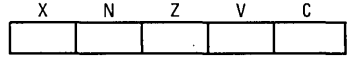
The first four bits represent a condition resulting from a controller operation. The X bit is an operand for multiprecision computations; when it is used, it is set to the value of the C bit. The carry bit and the multiprecision extend bit are separate in the M68000 Family to simplify programming techniques that use them (refer to Table 3-8 as an example).

The condition codes were developed to meet two criteria:

- Consistency — across instructions, uses, and instances
- Meaningful Results — no change unless it provides useful information

Consistency across instructions means that all instructions that are special cases of more general instructions affect the condition codes in the same way. Consistency across instances means that all instances of an instruction affect the condition codes in the same way. Consistency across uses means that conditional instructions test the condition codes similarly and provide the same results, regardless of whether the condition codes are set by a compare, test, or move instruction.

In the instruction set definitions, the CCR is shown as follows:



where:

X (extend)
Set to the value of the C bit for arithmetic operations. Otherwise not affected or set to a specified result.

N (negative)
Set if the most significant bit of the result is set. Cleared otherwise.

Z (zero)
Set if the result equals zero. Cleared otherwise.

V (overflow)
Set if arithmetic overflow occurs. This implies that the result cannot be represented in the operand size. Cleared otherwise.

C (carry)
Set if a carry out of the most significant bit of the operand occurs for an addition. Also set if a borrow occurs in a subtraction. Cleared otherwise.

3.3.1 Condition Code Computation

Most operations take a source operand and a destination operand, compute, and store the result in the destination location. Single-operand operations take a destination operand, compute, and store the result in the destination location. Table 3-12 lists each instruction and how it affects the condition code bits.

Table 3-12. Condition Code Computations

Operations	X	N	Z	V	C	Special Definition
ABCD	*	U	?	U	?	C = Decimal Carry $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
ADD, ADDI, ADDQ	*	*	*	?	?	$V = Sm \wedge Dm \wedge \overline{Rm} \vee \overline{Sm} \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge Dm \vee \overline{Rm} \wedge Dm \vee Sm \wedge \overline{Rm}$
ADDX	*	*	?	?	?	$V = Sm \wedge Dm \wedge \overline{Rm} \vee \overline{Sm} \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge Dm \vee \overline{Rm} \wedge Dm \vee Sm \wedge \overline{Rm}$ $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT, NOT, TAS, TST	—	*	*	0	0	
SUB, SUBI, SUBQ	*	*	*	?	?	$V = \overline{Sm} \wedge Dm \wedge \overline{Rm} \vee Sm \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge \overline{Dm} \vee Rm \wedge \overline{Dm} \vee Sm \wedge Rm$
SUBX	*	*	?	?	?	$V = \overline{Sm} \wedge Dm \wedge \overline{Rm} \vee Sm \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge \overline{Dm} \vee Rm \wedge \overline{Dm} \vee Sm \wedge Rm$ $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
CAS, CAS2, CMP, CMPI, CMPM	—	*	*	?	?	$V = \overline{Sm} \wedge Dm \wedge \overline{Rm} \vee Sm \wedge \overline{Dm} \wedge Rm$ $C = Sm \wedge \overline{Dm} \vee Rm \wedge \overline{Dm} \vee Sm \wedge Rm$
DIVS, DUVI	—	*	*	?	0	V = Division Overflow
MULS, MULU	—	*	*	?	0	V = Multiplication Overflow
SBCD, NBCD	*	U	?	U	?	C = Decimal Borrow $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{Ro}$
NEG	*	*	*	?	?	V = Dm ∧ Rm C = Dm ∨ Rm
NEGX	*	*	?	?	?	V = Dm ∧ Rm C = Dm ∨ Rm $Z = Z \wedge \overline{Rm} \wedge \ldots \wedge \overline{R0}$
BTST, BCHG, BSET, BCLR	—	—	?	—	—	$Z = \overline{Dn}$
BFTST, BFCHG, BFSET, BFCLR	—	?	?	0	0	$N = Dm$ $Z = \overline{Dm} \wedge \overline{DM-1} \wedge \ldots \wedge \overline{D0}$
BFEXTS, BFEXTU, BFFFO	—	?	?	0	0	$N = Sm$ $Z = \overline{Sm} \wedge \overline{Sm-1} \wedge \ldots \wedge \overline{S0}$
BFINS	—	?	?	0	0	$N = Dm$ $Z = \overline{Dm} \wedge \overline{DM-1} \wedge \ldots \wedge \overline{D0}$
ASL	*	*	*	?	?	$V = Dm \wedge (\overline{Dm-1} \vee \ldots \vee \overline{Dm-r}) \vee \overline{Dm} \wedge (\overline{DM-1} \vee \ldots + Dm-r)$ $C = \overline{Dm-r+1}$
ASL (R=0)	—	*	*	0	0	
LSL, ROXL	*	*	*	0	?	C = Dm−r+1
LSR (r=0)	—	*	*	0	0	

Table 3-12. Condition Code Computations (Continued)

Operations	X	N	Z	V	C	Special Definition
ROXL (r=0)	—	*	*	0	?	C=X
ROL	—	*	*	0	?	C=Dm−r+1
ROL (r=0)	—	*	*	0	0	
ASR, LSR, ROXR	*	*	*	0	?	C=Dr−1
ASR, LSR (r=0)	—	*	*	0	0	
ROXR (r=0)	—	*	*	0	?	C=X
ROR	—	*	*	0	?	C = Dr−1
ROR (r=0)	—	*	*	0	0	

— = Not Affected
U = Undefined, Result Meaningless
? = Other — See Special Definition
* = General Case
X = C
N = Rm
Z = $\overline{Rm} \wedge \ldots \wedge \overline{R0}$
Sm = Source Operand — Most Significant Bit
Dm = Destination Operand — Most Significant Bit
Rm = Result Operand — Most Significant Bit
R = Register Tested
n = Bit Number
r = Shift Count
LB = Lower Bound
UB = Upper Bound
$\wedge$ = Boolean AND
$\underline{V}$ = Boolean OR
$\overline{Rm}$ = NOT Rm

3.3.2 Conditional Tests

Table 3-13 lists the condition names, encodings, and tests for the conditional branch and set instructions. The test associated with each condition is a logical formula using the current states of the condition codes. If this formula evaluates to one, the condition is true. If the formula evaluates to zero, the condition is false. For example, the T condition is always true, and the EQ condition is true only if the Z bit condition code is currently true.

Table 3-13. Conditional Tests

Mnemonic	Condition	Encoding	Test
T*	True	0000	1
F*	False	0001	0
HI	High	0010	$\overline{C}\cdot\overline{Z}$
LS	Low or Same	0011	$C+Z$
CC(HS)	Carry Clear	0100	$\overline{C}$
CS(LO)	Carry Set	0101	C
NE	Not Equal	0110	$\overline{Z}$
EQ	Equal	0111	Z
VC	Overflow Clear	1000	$\overline{V}$
VS	Overflow Set	1001	V
PL	Plus	1010	$\overline{N}$
MI	Minus	1011	N
GE	Greater or Equal	1100	$N\cdot V+\overline{N}\cdot\overline{V}$
LT	Less Than	1101	$N\cdot\overline{V}+\overline{N}\cdot V$
GT	Greater Than	1110	$N\cdot V\cdot\overline{Z}+\overline{N}\cdot\overline{V}\cdot\overline{Z}$
LE	Less or Equal	1111	$Z+N\cdot\overline{V}+\overline{N}\cdot V$

• = Boolean AND
+ = Boolean OR
$\overline{N}$ = Boolean NOT N

*Not available for the Bcc instruction.

3.4 INSTRUCTION SET SUMMARY

Table 3-14 provides a alphabetized listing of the MC68EC030 instruction set listed by opcode, operation, and syntax.

Table 3-14 use notational conventions for the operands, the subfields and qualifiers, and the operations performed by the instructions. In the syntax descriptions, the left operand is the source operand, and the right operand is the destination operand. The following list contains the notations used in Table 3-14.

Notation for operands:

PC—Program counter
SR—Status register
V—Overflow condition code
Immediate Data—Immediate data from the instruction
Source—Source contents
Destination—Destination contents
Vector—Location of exception vector

+inf—Positive infinity
−inf—Negative infinity
<fmt>—Operand data format: byte (B), word (W), long (L), single (S), double (D), extended (X), or packed (P).
FPm—One of eight floating-point data registers (always specifies the source register)
FPn—One of eight floating-point data registers (always specifies the detination register)

3

Notation for subfields and qualifiers:

<bit> of <operand>—Selects a single bit of the operand
<ea>{offset:width}—Selects a bit field
(<operand>)—The contents of the referenced location
<operand>$_{10}$—The operand is binary coded decimal, operations are performed in decimal
(<address register>)—The register indirect operator
−(<address register>)—Indicates that the operand register points to the memory
(<address register>)+—Location of the instruction operand — the optional mode qualifiers are −, +, (d), and (d,ix)
#xxx or #<data>—Immediate data that follows the instruction word(s)

Notations for operations that have two operands, written <operand> <op> <operand>, where <op> is one of the following:

←—The source operand is moved to the destination operand
↔—The two operands are exchanged
+—The operands are added
−—The destination operand is subtracted from the source operand
×—The operands are multiplied
÷—The source operand is divided by the destination operand
<—Relational test, true if source operand is less than destination operand
>—Relational test, true if source operand is greater than destination operand
V—Logical OR
⊕—Logical exclusive OR
Λ—Logical AND

shifted by, rotated by—The source operand is shifted or rotated by the number of positions specified by the second operand

Notation for single-operand operations:

~<operand>—The operand is logically complemented

<operand>sign-extended—The operand is sign extended; all bits of the upper portion are made equal to the high-order bit of the lower portion

<operand>tested—The operand is compared to zero, and the condition codes are set appropriately

Notation for other operations:

TRAP—Equivalent to Format/Offset Word ➧ (SSP); SSP – 2 ➧ SSP; PC ➧ (SSP); SSP – 4 ➧ SSP; SR ➧ (SSP); SSP – 2 ➧ SSP; (vector) ➧ PC

STOP—Enter the stopped state, waiting for interrupts

If <condition> then <operations> else <operations>—The condition is tested. If true, the operations after "then" are performed. If the condition is false and the optional "else" clause is present, the operations after "else" are performed. If the condition is false and else is omitted, the instruction performs no operation. Refer to the Bcc instruction description as an example.

Table 3-14. Instruction Set Summary

Opcode	Operation	Syntax
ABCD	$Source_{10} + Destination_{10} + X$ ➧ Destination	ABCD Dy,Dx ABCD –(Ay),–(Ax)
ADD	Source + Destination ➧ Destination	ADD ⟨ea⟩,Dn ADD Dn,⟨ea⟩
ADDA	Source + Destination ➧ Destination	ADDA ⟨ea⟩,An
ADDI	Immediate Data + Destination ➧ Destination	ADDI #⟨data⟩,⟨ea⟩
ADDQ	Immediate Data + Destination ➧ Destination	ADDQ #⟨data⟩,⟨ea⟩
ADDX	Source + Destination + X ➧ Destination	ADDX Dy,Dx ADDX –(Ay),–(Ax)
AND	Source∧Destination ➧ Destination	AND ⟨ea⟩,Dn AND Dn,⟨ea⟩
ANDI	Immediate Data∧Destination ➧ Destination	ANDI #⟨data⟩,⟨ea⟩
ANDI to CCR	Source∧CCR ➧CCR	ANDI #⟨data⟩,CCR

Table 3-14. Instruction Set Summary (Continued)

Opcode	Operation	Syntax
ANDI to SR	If supervisor state the SourceΛSR ⇨ SR else TRAP	ANDI #⟨data⟩,SR
ASL,ASR	Destination Shifted by ⟨count⟩ ⇨ Destination	ASd Dx,Dy ASd #⟨data⟩,Dy ASd ⟨ea⟩
Bcc	If (condition true) then PC + d ⇨ PC	Bcc ⟨label⟩
BCHG	~(⟨number⟩ of Destination) ⇨ Z; ~(⟨number⟩ of Destination) ⇨ ⟨bit number⟩ of Destination	BCHG Dn,⟨ea⟩ BCHG #⟨data⟩,⟨ea⟩
BCLR	~(⟨bit number⟩ of Destination) ⇨ Z; 0 ⇨ ⟨bit number⟩ of Destination	BCLR Dn,⟨ea⟩ BCLR #⟨data⟩,⟨ea⟩
BFCHG	~(⟨bit field⟩ of Destination) ⇨ ⟨bit field⟩ of Destination	BFCHG ⟨ea⟩{offset:width}
BFCLR	0 ⇨ ⟨bit field⟩ of Destination	BFCLR ⟨ea⟩{offset:width}
BFEXTS	⟨bit field⟩ of Source ⇨ Dn	BFEXTS ⟨ea⟩{offset:width},Dn
BFEXTU	⟨bit offset⟩ of Source ⇨ Dn	BFEXTU ⟨ea⟩{offset:width},Dn
BFFFO	⟨bit offset⟩ of Source Bit Scan ⇨ Dn	BFFFO ⟨ea⟩{offset:width},Dn
BFINS	Dn ⇨ ⟨bit field⟩ of Destination	BFINS Dn,⟨ea⟩{offset:width}
BFSET	1s ⇨ ⟨bit field⟩ of Destination	BFSET ⟨ea⟩{offset:width}
BFTST	⟨bit field⟩ of Destination	BFTST ⟨ea⟩{offset:width}
BKPT	Run breakpoint acknowledge cycle; TRAP as illegal instruction	BKPT #⟨data⟩
BRA	PC + d ⇨ PC	BRA ⟨label⟩
BSET	~(⟨bit number⟩ of Destination) ⇨ Z; 1 ⇨ ⟨bit number⟩ of Destination	BSET Dn,⟨ea⟩ BSET #⟨data⟩,⟨ea⟩
BSR	SP − 4 ⇨ SP; PC ⇨ (SP); PC + d ⇨ PC	BSR ⟨label⟩
BTST	− (⟨bit number⟩ of Destination) ⇨ Z;	BTST Dn,⟨ea⟩ BTST #⟨data⟩,⟨ea⟩
CAS CAS2	CAS Destination — Compare Operand ⇨ cc; if Z, Update Operand ⇨ Destination else Destination ⇨ Compare Operand CAS2 Destination 1 — Compare 1 ⇨ cc; if Z, Destination 2 — Compare ⇨ cc; if Z, Update 1 ⇨ Destination 1; Update 2 ⇨ Destination 2 else Destination 1 ⇨ Compare 1; Destination 2 ⇨ Compare 2	CAS Dc,Du,⟨ea⟩ CAS2 Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)
CHK	If Dn < 0 or Dn > Source then TRAP	CHK ⟨ea⟩,Dn
CHK2	If Rn < lower bound or Rn > upper bound then TRAP	CHK2 ⟨ea⟩,Rn
CLR	0 ⇨ Destination	CLR ⟨ea⟩
CMP	Destination — Source ⇨ cc	CMP ⟨ea⟩,Dn
CMPA	Destination — Source	CMPA ⟨ea⟩,An
CMPI	Destination — Immediate Data	CMPI #⟨data⟩,⟨ea⟩
CMPM	Destination — Source ⇨ cc	CMPM (Ay)+,(Ax)+

Table 3-14. Instruction Set Summary (Continued)

Opcode	Operation	Syntax
CMP2	Compare Rn < lower-bound or Rn > upper-bound and Set Condition Codes	CMP2 ⟨ea⟩,Rn
cpBcc	If cpcc true then scanPC + d ⇨ PC	cpBcc ⟨label⟩
cpDBcc	If cpcc false then (Dn − 1 ⇨ Dn; If Dn ≠ − 1 then scanPC + d ⇨ PC)	cpDBcc Dn,⟨label⟩
cpGEN	Pass Command Word to Coprocessor	cpGEN ⟨parameters as defined by co-processor⟩
cpRESTORE	If supervisor state then Restore Internal State of Coprocessor else TRAP	cpRESTORE ⟨ea⟩
cpSAVE	If supervisor state then Save Internal State of Coprocessor else TRAP	cpSAVE ⟨ea⟩
cpScc	If cpcc true then 1s ⇨ Destination else 0s ⇨ Destination	cpScc ⟨ea⟩
cpTRAPcc	If cpcc true then TRAP	cpTRAPcc cpTRAPcc #⟨data⟩
DBcc	If condition false then (Dn − 1 ⇨ Dn; If Dn ≠ − 1 then PC + d ⇨ PC)	DBcc Dn,⟨label⟩
DIVS DIVSL	Destination/Source ⇨ Destination	DIVS.W ⟨ea⟩,Dn 32/16 ⇨ 16r:16q DIVS.L ⟨ea⟩,Dq 32/32 ⇨ 32q DIVS.L ⟨ea⟩,Dr:Dq 64/32 ⇨ 32r:32q DIVSL.L ⟨ea⟩,Dr:Dq 32/32 ⇨ 32r:32q
DIVU DIVUL	Destination/Source ⇨ Destination	DIVU.W ⟨ea⟩,Dn 32/16 ⇨ 16r:16q DIVU.L ⟨ea⟩,Dq 32/32 ⇨ 32q DIVU.L ⟨ea⟩,Dr:Dq 64/32 ⇨ 32r:32q DIVUL.L ⟨ea⟩,Dr:Dq 32/32 ⇨ 32r:32q
EOR	Source ⊕ Destination ⇨ Destination	EOR Dn,⟨ea⟩
EORI	Immediate Data ⊕ Destination ⇨ Destination	EORI #⟨data⟩,⟨ea⟩
EORI to CCR	Source ⊕ CCR ⇨ CCR	EORI #⟨data⟩,CCR
EORI to SR	If supervisor state the Source ⊕ SR ⇨ SR else TRAP	EORI #⟨data⟩,SR
EXG	Rx ⇔ Ry	EXG Dx,Dy EXG Ax,Ay EXG Dx,Ay EXG Ay,Dx
EXT EXTB	Destination Sign-Extended ⇨ Destination	EXT.W Dn extend byte to word EXT.L L Dn extend word to long word EXTB.L Dn extend byte to long word
ILLEGAL	SSP − 2 ⇨ SSP; Vector Offset ⇨ (SSP); SSP − 4 ⇨ SSP; PC ⇨ (SSP); SSp − 2 ⇨ SSP; SR ⇨ (SSP); Illegal Instruction Vector Address ⇨ PC	ILLEGAL
JMP	Destination Address ⇨ PC	JMP ⟨ea⟩

Table 3-14. Instruction Set Summary (Continued)

Opcode	Operation	Syntax
JSR	SP − 4 ⇨ SP; PC ⇨ (SP) Destination Address ⇨ PC	JSR ⟨ea⟩
LEA	⟨ea⟩ ⇨ An	LEA ⟨ea⟩,An
LINK	SP − 4 ⇨ SP; An ⇨ (SP) SP ⇨ An, SP + d ⇨ SP	LINK An,#⟨displacement⟩
LSL,LSR	Destination Shifted by ⟨count⟩ ⇨ Destination	LSd[5] Dx,Dy LSd[5] #⟨data⟩,Dy LSd[5] ⟨ea⟩
MOVE	Source ⇨ Destination	MOVE ⟨ea⟩,⟨ea⟩
MOVEA	Source ⇨ Destination	MOVEA ⟨ea⟩,An
MOVE from CCR	CCR ⇨ Destination	MOVE CCR,⟨ea⟩
MOVE to CCR	Source ⇨ CCR	MOVE ⟨ea⟩,CCR
MOVE from SR	If supervisor state then SR ⇨ Destination else TRAP	MOVE SR,⟨ea⟩
MOVE to SR	If supervisor state then Source ⇨ SR else TRAP	MOVE ⟨ea⟩,SR
MOVE USP	If supervisor state then USP ⇨ An or An ⇨ USP else TRAP	MOVE USP,An MOVE An,USP
MOVEC	If supervisor state then Rc ⇨ Rn or Rn ⇨ Rc else TRAP	MOVEC Rc,Rn MOVEC Rn,Rc
MOVEM	Registers ⇨ Destination Source ⇨ Registers	MOVEM register list,⟨ea⟩ MOVEM ⟨ea⟩,register list
MOVEP	Source ⇨ Destination	MOVEP Dx,(d,Ay) MOVEP (d,Ay),Dx
MOVEQ	Immediate Data ⇨ Destination	MOVEQ #⟨data⟩,Dn
MOVES	If supervisor state then Rn ⇨ Destination [DFC] or Source [SFC] ⇨ Rn else TRAP	MOVES Rn,⟨ea⟩ MOVES ⟨ea⟩,Rn
MULS	Source × Destination ⇨ Destination	MULS.W ⟨ea⟩,Dn 16 × 16 ⇨ 32 MULS.L ⟨ea⟩,Dl 32 × 32 ⇨ 32 MULS.L ⟨ea⟩,Dh:Dl 32 × 32 ⇨ 64
MULU	Source × Destination ⇨ Destination	MULU.W ⟨ea⟩,Dn 16 × 16 ⇨ 32 MULU.L ⟨ea⟩,Dl 32 × 32 ⇨ 32 MULU.L ⟨ea⟩,Dh:Dl 32 × 32 ⇨ 64
NBCD	0 − (Destination$_{10}$) − X ⇨ Destination	NBCD ⟨ea⟩
NEG	0 − (Destination) ⇨ Destination	NEG ⟨ea⟩
NEGX	0 − (Destination) − X ⇨ Destination	NEGX ⟨ea⟩
NOP	None	NOP
NOT	~Destination ⇨ Destination	NOT ⟨ea⟩

Table 3-14. Instruction Set Summary (Continued)

Opcode	Operation	Syntax
OR	Source V Destination ⇽ Destination	OR ⟨ea⟩,Dn OR Dn,⟨ea⟩
ORI	Immediate Data V Destination ⇽ Destination	ORI #⟨data⟩,⟨ea⟩
ORI to CCR	Source V CCR ⇽ CCR	ORI #⟨data⟩,CCR
ORI to SR	If supervisor state then Source V SR ⇽ SR else TRAP	ORI #⟨data⟩,SR
PACK	Source (Unpacked BCD) + adjustment ⇽ Destintion (Packed BCD)	PACK −(Ax),−(Ay),#⟨adjustment⟩ PACK Dx,Dy,#⟨adjustment⟩
PEA	Sp − 4 ⇽ SP; ⟨ea⟩ ⇽ (SP)	PEA ⟨ea⟩
PMOVE	If supervisor state then (Source) ⇽ MRn or MRn ⇽ (Destination) else TRAP	PMOVE MRn,⟨ea⟩ PMOVE ⟨ea⟩,MRn PMOVEFD ⟨ea⟩,MRn
PTEST	If supervisor state then Access control status ⇽ ACUSR; else TRAP	PTESTR (An) PTESTW (An)
RESET	If supervisor state then Assert $\overline{\text{RSTO}}$ Line else TRAP	RESET
ROL,ROR	Destination Rotated by ⟨count⟩ ⇽ Destination	ROd[5] Rx,Dy ROd[5] #⟨data⟩,Dy ROd[5] ⟨ea⟩
ROXL,ROXR	Destination Rotated with X by ⟨count⟩ ⇽ Destination	ROXd[5] Dx,Dy ROXd[5] #⟨data⟩,Dy ROXd[5] ⟨ea⟩
RTD	(SP) ⇽ PC; SP + 4 + d ⇽ SP	RTD #⟨displacement⟩
RTE	If supervisor state the (SP) ⇽ SR; SP + 2 ⇽ SP; (SP) ⇽ PC; SP + 4 ⇽ SP; restore state and deallocate stack according to (SP) else TRAP	RTE
RTR	(SP) ⇽ CCR; SP + 2 ⇽ SP; (SP) ⇽ PC; SP + 4 ⇽ SP	RTR
RTS	(SP) ⇽ PC; SP + 4 ⇽ SP	RTS
SBCD	$\text{Destination}_{10} - \text{Source}_{10} - X$ ⇽ Destination	SBCD Dx,Dy SBCD −(Ax),−(Ay)
Scc	If Condition True then 1s ⇽ Destination else 0s ⇽ Destination	Scc ⟨ea⟩
STOP	If supervisor state then Immediate Data ⇽ SR; STOP else TRAP	STOP #⟨data⟩
SUB	Destination − Source ⇽ Destination	SUB ⟨ea⟩,Dn SUB Dn,⟨ea⟩
SUBA	Destination − Source ⇽ Destination	SUBA ⟨ea⟩,An

Table 3-14. Instruction Set Summary (Concluded)

Opcode	Operation	Syntax
SUBI	Destination – Immediate Data ⇨ Destination	SUBI #⟨data⟩,⟨ea⟩
SUBQ	Destination – Immediate Data ⇨ Destination	SUBQ #⟨data⟩,⟨ea⟩
SUBX	Destination – Source – X ⇨ Destination	SUBX Dx,Dy SUBX –(Ax),–(Ay)
SWAP	Register [31:16] ⇔ Register [15:0]	SWAP Dn
TAS	Destination Tested ⇨ Condition Codes; 1 ⇨ bit 7 of Destination	TAS ⟨ea⟩
TRAP	SSP – 2 ⇨ SSP; Format/Offset ⇨ (SSP); SSP – 4 ⇨ SSP; PC ⇨ (SSP); SSP – 2 ⇨ SSP; SR ⇨ (SSP); Vector Address ⇨ PC	TRAP #⟨vector⟩
TRAPcc	If cc then TRAP	TRAPcc TRAPcc.W #⟨data⟩ TRAPcc.L #⟨data⟩
TRAPV	If V then TRAP	TRAPV
TST	Destination Tested ⇨ Condition Codes	TST ⟨ea⟩
UNLK	An ⇨ SP; (SP) ⇨ An; SP + 4 ⇨ SP	UNLK An
UNPK	Source (Packed BCD) + adjustment ⇨ Destination (Unpacked BCD)	UNPACK –(Ax),–(Ay),#⟨adjustment⟩ UNPACK Dx,Dy,#⟨adjustment⟩

NOTES:
1. Specifies either the instruction (IC), data (DC), or IC/DC caches.
2. Where r is rounding precision, S or D.
3. A list of any combination of the eight floating-point data registers, with individual register names separated by a slash (/); and/or contiguous blocks of registers specified by the first and last register names separated by a dash (–).
4. A list of any combination of the three floating-point system control registers (FPCR, FPSR, and FPIAR) with individual register names separated by a slash (/).
5. Where d is direction, L or R.

3.5 INSTRUCTION EXAMPLES

The following paragraphs provide examples of how to use selected instructions.

3.5.1 Using the CAS and CAS2 Instructions

The CAS instruction compares the value in a memory location with the value in a data register, and copies a second data register into the memory location if the compared values are equal. This provides a means of updating system counters, history information, and globally shared pointers. The instruction uses an indivisible read-modify-write cycle; after CAS reads the memory location, no other instruction can change that location before CAS has written the new value. This provides security in single-processor systems, in multitasking environments, and in multiprocessor environments. In a single-processor system, the operation is protected from instructions of an interrupt

routine. In a multitasking environment, no other task can interfere with writing the new value of a system variable. In a multiprocessor environment, the other processors must wait until the CAS instruction completes before accessing a global pointer.

The following code fragment shows a routine to maintain a count, in location SYS_CNTR, of the executions of an operation that may be performed by any process or processor in a system. The routine obtains the current value of the count in register D0 and stores the new count value in register D1. The CAS instruction copies the new count into SYS_CNTR if it is valid. However, if another user has incremented the counter between the time the count was stored and the read-modify-write cycle of the CAS instruction, the write portion of the cycle copies the new count in SYS_CNTR into D0, and the routine branches to repeat the test. The following code sequence guarantees that SYS_CNTR is correctly incremented.


```
            MOVE.W    SYS_CNTR,D0      get the old value of the counter
INC_LOOP    MOVE.W    D0,D1            make a copy of it
            ADDQ.W    #1,D1            and increment it
            CAS.W     D0,D1,SYS_CNTR   if counter value is still the same, update it
            BNE       INC_LOOP         if not, try again
```

The CAS and CAS2 instructions together allow safe operations in the manipulation of system linked lists. Controlling a single location, HEAD in the example, manages a last-in-first-out linked list (see Figure 3-2). If the list is empty, HEAD contains the NULL pointer (0); otherwise, HEAD contains the address of the element most recently added to the list. The code fragment shown in Figure 3-2 illustrates the code for inserting an element. The MOVE instructions load the address in location HEAD into D0 and into the NEXT pointer in the element being inserted, and the address of the new element into D1. The CAS instruction stores the address of the inserted element into location HEAD if the address in HEAD remains unaltered. If HEAD contains a new address, the instruction loads the new address into D0 and branches to the second MOVE instruction to try again.

```
SINSERT                                  ALLOCATE NEW ENTRY, ADDRESS IN A1
        MOVE.L    HEAD.D0                MOVE HEAD POINTER VALUE TO D0
SILOOP  MOVE.L    D0, (NEXT, A1)         ESTABLISH FORWARD LINK IN NEW ENTRY
        MOVE.L    A1, D1                 MOVE NEW ENTRY POINTER VALUE TO D1
        CAS.L     D0, D1, HEAD           IF WE STILL POINT TO TOP OF STACK, UPDATE THE HEAD POINTER
        BNE       SILOOP                 IF NOT, TRY AGAIN
```

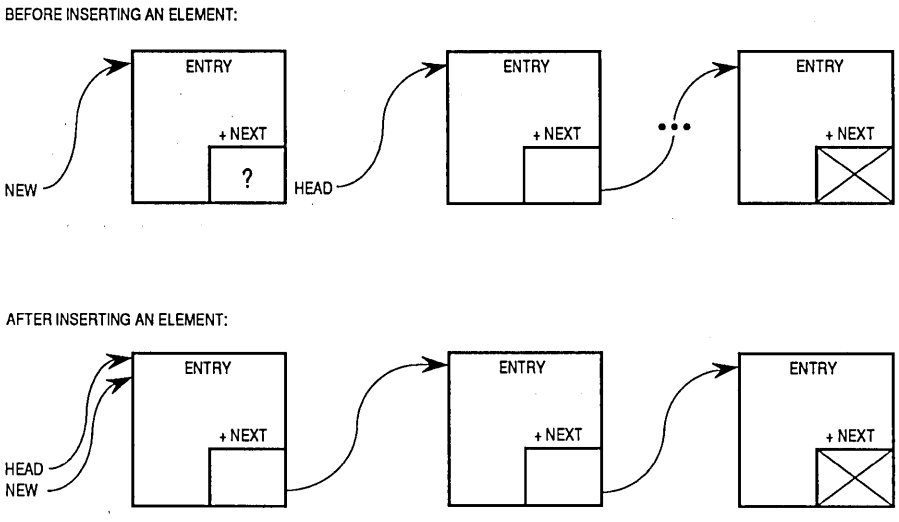


Figure 3-2. Linked List Insertion

The CAS2 instruction is similar to the CAS instruction except that it performs two comparisons and updates two variables when the results of the comparisons are equal. If the results of both comparisons are equal, CAS2 copies new values into the destination addresses. If the result of either comparison is not equal, the instruction copies the values in the destination addresses into the compare operands.

The next code (see Figure 3-3) fragment shows the use of a CAS2 instruction to delete an element from a linked list. The first LEA instruction loads the effective address of HEAD into A0. The MOVE instruction loads the address in pointer HEAD into D0. The TST instruction checks for an empty list, and the BEQ instruction branches to a routine at label SDEMPTY if the list is empty. Otherwise, a second LEA instruction loads the address of the NEXT pointer in the newest element on the list into A1, and the following MOVE instruction loads the pointer contents into D1. The CAS2 instruction compares the address of the newest structure to the value in HEAD and the address in D1 to the pointer in the address in A1. If no element has been inserted or deleted by another routine while this routine has been executing, the results of these comparisons are equal, and the CAS2 instruction stores the new

value into location HEAD. If an element has been inserted or deleted, the CAS2 instruction loads the new address in location HEAD into D0, and the BNE instruction branches to the TST instruction to try again.

```
SDELETE
            LEA     HEAD, A0        LOAD ADDRESS OF HEAD POINTER INTO A0
            MOVE.L  (A0), D0        MOVE VALUE OF HEAD POINTER INTO D0
SDLOOP      TST.L   D0              CHECK FOR NULL HEAD POINTER
            BEQ     SDEMPTY         IF EMPTY, NOTHING TO DELETE
            LEA     (NEXT, D0), A1  LOAD ADDRESS OF FORWARD LINK INTO A1
            MOVE.L  (A1), D1        PUT FORWARD LINK VALUE IN D1
            CAS2.L  D0:D1, D1:D1, (A0):(A1)
                                    IF STILL POINT TO ENTRY TO BE DELETED, THEN UPDATE HEAD AND
                                    FORWARD POINTERS
            BNE     SDLOOP          IF NOT, TRY AGAIN
SDEMPTY                             SUCCESSFUL DELETION, ADDRESS OF DELETED ENTRY IN D0 (MAY BE
                                    NULL)
```

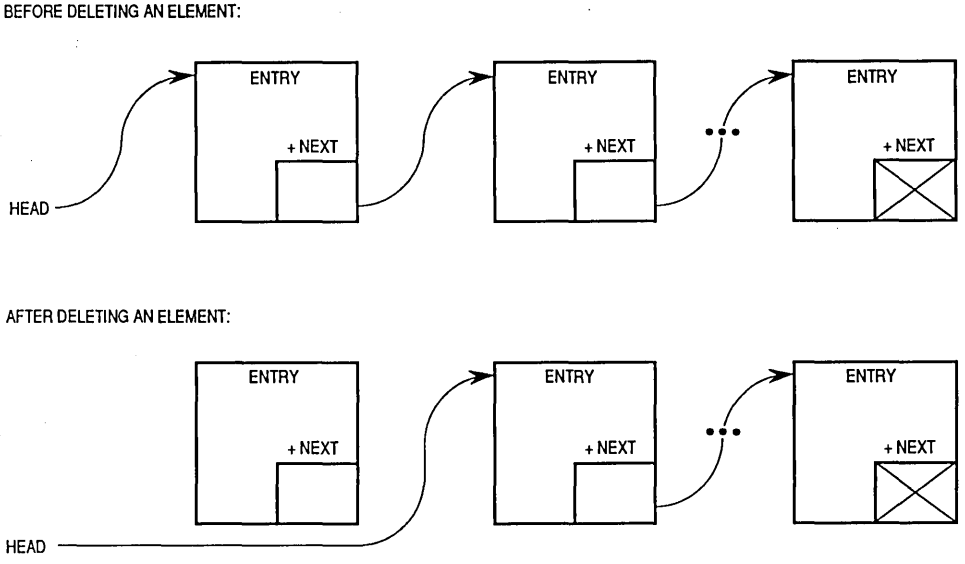


Figure 3-3. Linked List Deletion

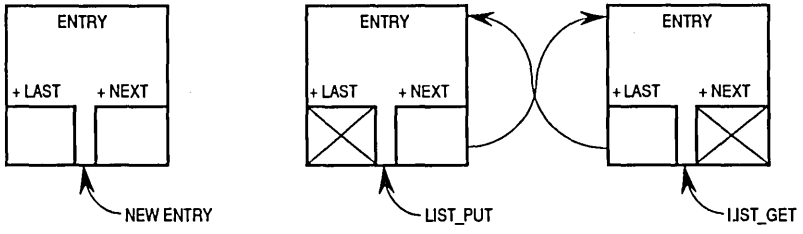
The CAS2 instruction can also be used to correctly maintain a first-in-first-out doubly linked list. A doubly linked list needs two controlled locations, LIST_PUT and LIST_GET, which contain pointers to the last element inserted in the list and the next to be removed, respectively. If the list is empty, both pointers are NULL (0).

The code fragment shown in Figure 3-4 illustrates the insertion of an element in a doubly linked list. The first two instructions load the effective addresses

of LIST_PUT and LIST_GET into registers A0 and A1, respectively. The next instruction moves the address of the new element into register D2. Another MOVE instruction moves the address in LIST_PUT into register D0. At label DILOOP, a TST instruction tests the value in D0, and the BEQ instruction branches to the MOVE instruction when D0 is equal to zero. Assuming the list is empty, this MOVE instruction is executed next; it moves the zero in D0 into the NEXT and LAST pointers of the new element. Then the CAS2 instruction moves the address of the new element into both LIST_PUT and LIST_GET, assuming that both of these pointers still contain zero. If not, the BNE instruction branches to the TST instruction at label DILOOP to try again. This time, the BEQ instruction does not branch, and the following MOVE instruction moves the address in D0 to the NEXT pointer of the new element. The CLR instruction clears register D1 to zero, and the MOVE instruction moves the zero into the LAST pointer of the new element. The LEA instruction loads the address of the LAST pointer of the most recently inserted element into register A1. Assuming the LIST_PUT pointer and the pointer in A1 have not been changed, the CAS2 instruction stores the address of the new element into these pointers.

```
DINSERT                                          (ALLOCATE NEW LIST ENTRY, LOAD ADDRESS INTO A2)
        LEA     LIST_PUT, A0        LOAD ADDRESS OF HEAD POINTER INTO A0
        LEA     LIST_GET, A1        LOAD ADDRESS OF TAIL POINTER INTO A1
        MOVE.L  A2, D2              LOAD NEW ENTRY POINTER INTO D2
        MOVE.L  (A0), D0            LOAD POINTER TO HEAD ENTRY INTO D0
        TST.L   D0                  IS HEAD POINTER NULL, (0 ENTRIES IN LIST)?
DILOOP  BEQ     DIEMPTY             IF SO, WE NEED ONLY TO ESTABLISH POINTERS
        MOVE.L  D0, (NEXT, A2)      PUT HEAD POINTER INTO FORWARD POINTER OF NEW ENTRY
        CLR.L   D1                  PUT NULL POINTER VALUE INTO D1
        MOVE.L  D1, (LAST, A2)      PUT NULL POINTER IN BACKWARD POINTER OF NEW ENTRY
        LEA     (LAST, D0), A1      LOAD BACKWARD POINTER OF OLD HEAD ENTRY INTO A1
        CAS2.L  D0:D1,D2:D2,(A0):(A1)  IF WE STILL POINT TO OLD HEAD ENTRY, UPDATE POINTERS
        BNE     DILOOP              IF NOT, TRY AGAIN
        BRA     DIDONE
DIEMPTY MOVE.L  D0, (NEXT, A2)      PUT NULL POINTER IN FORWARD POINTER OF NEW ENTRY
        MOVE.L  D0, (LAST, A2)      PUT NULL POINTER IN BACKWARD POINTER OF NEW ENTRY
        CAS2.L  D0:D0,D2:D2,(A0):(A1)  IF WE STILL HAVE NO ENTRIES, SET BOTH POINTERS TO THIS ENTRY
        BNE     DILOOP              IF NOT, TRY AGAIN
DIDONE                              SUCCESSFUL LIST ENTRY INSERTION
```

BEFORE INSERTING NEW ENTRY:



AFTER INSERTING NEW ENTRY:

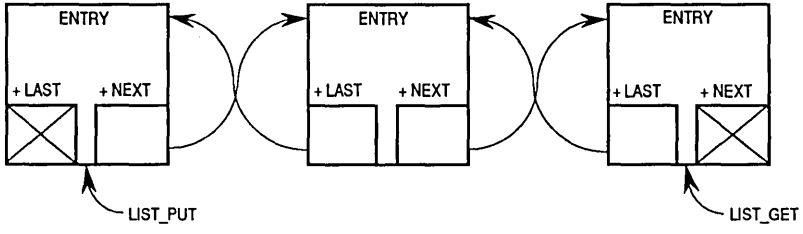


Figure 3-4. Doubly Linked List Insertion

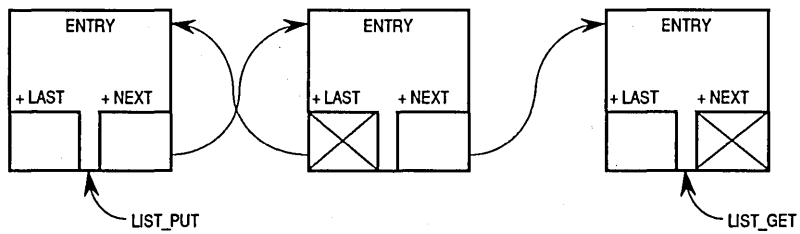
The code fragment to delete an element from a doubly linked list is similar (see Figure 3-5). The first two instructions load the effective addresses of pointers LIST_PUT and LIST_GET into registers A0 and A1, respectively. The MOVE instruction at label DDLOOP moves the LIST_GET pointer into register D1. The BEQ instruction that follows branches out of the routine when the pointer is zero. The MOVE instruction moves the LAST pointer of the element to be deleted into register D2. Assuming this is not the last element in the list, the Z condition code is not set, and the branch to label DDEMPTY does

not occur. The LEA instruction loads the address of the NEXT pointer of the element at the address in D2 into register A2. The next instruction, a CLR instruction, clears register D0 to zero. The CAS2 instruction compares the address in D1 to the LIST_GET pointer and to the address in register A2. If the pointers have not been updated, the CAS2 instruction loads the address in D2 into the LIST_GET pointer and zero into the address in register A2.

When the list contains only one element, the routine branches to the CAS2 instruction at label DDEMPTY after moving a zero pointer value into D2. This

```
DDELETE
                LEA     LIST_PUT, A0        GET ADDRESS OF HEAD POINTER IN A0
                LEA     LIST_GET, A1        GET ADDRESS OF TAIL POINTER IN A1
DDLOOP          MOVE.L  (A1),D1             MOVE TAIL POINTER INTO D1
                BEQ     DDDONE              IF NO LIST, QUIT
                MOVE.L  (LAST,D1),D2        PUT BACKWARD POINTER IN D2
                BEQ     DDEMPTY             IF ONLY ONE ELEMENT, UPDATE POINTERS
                LEA     (NEXT,D2),A2        PUT ADDRESS OF FORWARD POINTER IN A2
                CLR.L   D0                  PUT NULL POINTER VALUE IN D0
                CAS2.L  D1:D1,D2:D0,(A1):(A2)  IF BOTH POINTERS STILL POINT TO THIS ENTRY , UPDATE THEM
                BNE     DDLOOP              IF NOT, TRY AGAIN
                BRA     DDDONE
DDEMPTY         CAS2.L  D1:D1,D2:D2,(A1):(A0)  IF STILL FIRST ENTRY, SET HEAD AND TAIL POINTERS TO NULL
                BNE     DDLOOP              IF NOT, TRY AGAIN
DDDONE                                      SUCCESSFUL ENTRY DELETION, ADDRESS OF DELETED ENTRY IN D1
                                            (MAY BE NULL)
```

BEFORE DELETING ENTRY:



AFTER DELETING ENTRY:

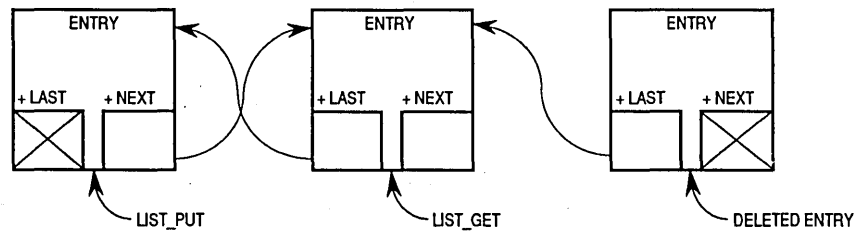


Figure 3-5. Doubly Linked List Deletion

instruction checks the addresses in LIST_PUT and LIST_GET to verify that no other routine has inserted another element or deleted the last element. Then the instruction moves zero into both pointers, and the list is empty.

3.5.2 Nested Subroutine Calls

The LINK instruction pushes an address onto the stack, saves the stack address at which the address is stored, and reserves an area of the stack. Using this instruction in a series of subroutine calls results in a linked list of stack frames.

The UNLK instruction removes a stack frame from the end of the list by loading an address into the stack pointer and pulling the value at that address from the stack. When the operand of the instruction is the address of the link address at the bottom of a stack frame, the effect is to remove the stack frame from the stack and from the linked list.

3.5.3 Bit Field Operations

One data type provided by the MC68EC030 is the bit field, consisting of as many as 32 consecutive bits. A bit field is defined by an offset from an effective address and a width value. The offset is a value in the range of -2^{31} through $2^{31}-1$ from the most significant bit (bit 7) at the effective address. The width is a positive number, 1–32. The most significant bit of a bit field is bit 0; the bits number in a direction opposite to the bits of an integer.

The instruction set includes eight instructions that have bit field operands. The insert bit field (BFINS) instruction inserts a bit field stored in a register into a bit field. The extract bit field signed (BFEXTS) instruction loads a bit field into the least significant bits of a register and extends the sign to the left, filling the register. The extract bit field unsigned (BFEXTU) also loads a bit field, but zero fills the unused portion of the destination register.

The set bit field (BFSET) instruction sets all the bits of a field to ones. The clear bit field (BFCLR) instruction clears a field. The change bit field (BFCHG) instruction complements all the bits in a bit field. These three instructions all test the previous value of the bit field, setting the condition codes accordingly. The test bit field (BFTST) instruction tests the value in the field, setting the condition codes appropriately without altering the bit field. The find first one in bit field (BFFFO) instruction scans a bit field from bit 0 to the right until it finds a bit set to one and loads the bit offset of the first set bit into the specified data register. If no bits in the field are set, the field offset and the field width is loaded into the register.

An important application of bit field instructions is the manipulation of the exponent field in a floating-point number. In the IEEE standard format, the most significant bit is the sign bit of the mantissa. The exponent value begins at the next most significant bit position; the exponent field does not begin on a byte boundary. The extract bit field (BFEXTU) instruction and the BFTST instruction are the most useful for this application, but other bit field instructions can also be used.

Programming of input and output operations to peripherals requires testing, setting, and inserting of bit fields in the control registers of the peripherals, which is another application for bit field instructions. However, control register locations are not memory locations; therefore, it is not always possible to insert or extract bit fields of a register without affecting other fields within the register.

Another widely used application for bit field instructions is bit-mapped graphics. Because byte boundaries are ignored in these areas of memory, the field definitions used with bit field instructions are very helpful.

3.5.4 Pipeline Synchronization with the NOP Instruction

Although the no operation (NOP) instruction performs no visible operation, it serves an important purpose. It forces synchronization of the integer unit pipeline by waiting for all pending bus cycles to complete. All previous integer instructions and floating-point external operand accesses complete execution before the NOP begins. The NOP instruction does not synchronize the FPU pipeline; floating-point instructions with floating-point register operand destinations can be executing when the NOP begins.

SECTION 4
PROCESSING STATES

This section describes the processing states of the MC68EC030. It describes the functions of the bits in the supervisor portion of the status register and the actions taken by the controller in response to exception conditions.

Unless the controller has halted, it is always in either the normal or the exception processing state. Whenever the controller is executing instructions or fetching instructions or operands, it is in the normal processing state. The controller is also in the normal processing state while it is storing instruction results or communicating with a coprocessor.

NOTE

Exception processing refers specifically to the transition from normal processing of a program to normal processing of system routines, interrupt routines, and other exception handlers. Exception processing includes all stacking operations, the fetch of the exception vector, and filling of the instruction pipe caused by an exception. It has completed when execution of the first instruction of the exception handler routine begins.

The controller enters the exception processing state when an interrupt is acknowledged, when an instruction is traced or results in a trap, or when some other exceptional condition arises. Execution of certain instructions or unusual conditions occurring during the execution of any instructions can cause exceptions. External conditions, such as interrupts, bus errors, and some coprocessor responses, also cause exceptions. Exception processing provides an efficient transfer of control to handlers and routines that process the exceptions.

A catastrophic system failure occurs whenever the controller receives a bus error or generates an address error while in the exception processing state. This type of failure halts the controller. For example, if during the exception processing of one bus error another bus error occurs, the MC68EC030 has not completed the transition to normal processing and has not completed saving the internal state of the machine, so the controller assumes that the system is not operational and halts. Only an external reset can restart a halted

controller. (When the controlelr executes a STOP instruction, it is in a special type of normal processing state, one without bus cycles. It is stopped, not halted.)

4.1 PRIVILEGE LEVELS

The controller operates at one of two levels of privilege: the user level or the supervisor level. The supervisor level has higher privileges than the user level. Not all controller or coprocessor instructions are permitted to execute in the lower privileged user level, but all are available at the supervisor level. This allows a separation of supervisor and user so the supervisor can protect system resources from uncontrolled access. The controller uses the privilege level indicated by the S bit in the status register to select either the user or supervisor privilege level and either the user stack pointer or a supervisor stack pointer for stack operations. The controller identifies a bus access (supervisor or user mode) via the function codes so that differentiation between supervisor and user can be maintained. The access control unit uses the indication of privilege level to control memory accesses to protect supervisor code, data, and resources from access by user programs.

In many systems, the majority of programs execute at the user level. User programs can access only their own code and data areas and can be restricted from accessing other information. The operating system typically executes at the supervisor privilege level. It has access to all resources, performs the overhead tasks for the user level programs, and coordinates their activities.

4.1.1 Supervisor Privilege Level

The supervisor level is the higher privilege level. The privilege level is determined by the S bit of the status register; if the S bit is set, the supervisor privilege level applies, and all instructions are executable. The bus cycles for instructions executed at the supervisor level are normally classified as supervisor references, and the values of the function codes on FC0–FC2 refer to supervisor address spaces.

In a multitasking operating system, it is more efficient to have a supervisor stack space associated with each user task and a separate stack space for interrupt associated tasks. The MC68EC030 provides two supervisor stacks, master and interrupt; the M bit of the status register selects which of the two is active. When the M bit is set to one, supervisor stack pointer references (either implicit or by specifying address register A7) access the master stack pointer (MSP). The operating system sets the MSP for each task to point to

The value of the M bit in the status register does not affect execution of privileged instructions; both master and interrupt modes are at the supervisor privilege level. Instructions that affect the M bit are MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, and RTE. Also, the controller automatically saves the M-bit value and clears it in the SR as part of the exception processing for interrupts.

All exception processing is performed at the supervisor privilege level. All bus cycles generated during exception processing are supervisor references, and all stack accesses use the active supervisor stack pointer.

4.1.2 User Privilege Level

The user level is the lower privilege level. The privilege level is determined by the S bit of the status register; if the S bit is clear, the controller executes instructions at the user privilege level.

Most instructions execute at either privilege level, but some instructions that have important system effects are privileged and can only be executed at the supervisor level. For instance, user programs are not allowed to execute the STOP instruction or the RESET instruction. To prevent a user program from entering the supervisor privilege level, except in a controlled manner, instructions that can alter the S bit in the status register are privileged. The TRAP #n instruction provides controlled access to operating system services for user programs.

The bus cycles for an instruction executed at the user privilege level are classified as user references, and the values of the function codes on FC0–FC2 specify user address spaces. When it is enabled, the access control unit of the controller uses the value of the function codes to distinguish between user and supervisor activity and to control cacheability to protected portions of the address space. While the controller is at the user level, references to the system stack pointer implicitly, or to address register seven (A7) explicitly, refer to the user stack pointer (USP).

4.1.3 Changing Privilege Level

To change from the user to the supervisor privilege level, one of the conditions that causes the controller to perform exception processing must occur. This causes a change from the user level to the supervisor level and can cause a change from the master mode to the interrupt mode. Exception processing saves the current values of the S and M bits of the status register

(along with the rest of the status register) on the active supervisor stack, and then sets the S bit, forcing the controller into the supervisor privilege level. When the exception being processed is an interrupt and the M bit is set, the M bit is cleared, putting the controller into the interrupt mode. Execution of instructions continues at the supervisor level to process the exception condition.

To return to the user privilege level, a system routine must execute one of the following instructions: MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, or RTE. The MOVE, ANDI, EORI, and ORI to SR and RTE instructions execute at the supervisor privilege level and can modify the S bit of the status register. After these instructions execute, the instruction pipeline is flushed and is refilled from the appropriate address space. This is indicated externally by the assertion of the $\overline{\text{REFILL}}$ signal.

The RTE instruction returns to the program that was executing when the exception occurred. It restores the exception stack frame saved on the supervisor stack. If the frame on top of the stack was generated by an interrupt, trap, or instruction exception, the RTE instruction restores the status register and program counter to the values saved on the supervisor stack. The controller then continues execution at the restored program counter address and at the privilege level determined by the S bit of the restored status register. If the frame on top of the stack was generated by a bus fault (bus error or address error exception), the RTE instruction restores the entire saved controller state from the stack.

4.2 ADDRESS SPACE TYPES

The controller specifies a target address space for every bus cycle with the function code signals according to the type of access required. In addition to distinguishing between supervisor/user and program/data, the controller can identify special controller cycles, such as the interrupt acknowledge cycle, and the access control unit can control access cacheability. Table 4-1 lists the types of accesses defined for the MC68EC030 and the corresponding values of function codes FC0–FC2.

Table 4-1. Address Space Encodings

FC2	FC1	FC0	Address Space
0	0	0	(Undefined, Reserved)*
0	0	1	User Data Space
0	1	0	User Program Space
0	1	1	(Undefined, Reserved)*
1	0	0	(Undefined, Reserved)*
1	0	1	Supervisor Data Space
1	1	0	Supervisor Program Space
1	1	1	CPU Space

*Address space 3 is reserved for user definition; 0 and 4 are reserved for future use by Motorola.

The memory locations of user program and data accesses are not predefined. Neither are the locations of supervisor data space. During reset, the first two long words beginning at memory location zero in the supervisor program space are used for controller initialization. No other memory locations are explicitly defined by the MC68EC030.

A function code of $7 ([FC2:FC0] = 111) selects the CPU address space. This is a special address space that does not contain instructions or operands but is reserved for special controller functions. The controller uses accesses in this space to communicate with external devices for special purposes. For example, all M68000 processors use the CPU space for interrupt acknowledge cycles. The MC68020, MC68030, and MC68EC030 also generate CPU space accesses for breakpoint acknowledge and coprocessor operations.

Supervisor programs can use the MOVES instruction to access all address spaces, including the user spaces and the CPU address space. Although the MOVES instruction can be used to generate CPU space cycles, this may interfere with proper system operation. Thus, the use of MOVES to access the CPU space should be done with caution.

4.3 EXCEPTION PROCESSING

An exception is defined as a special condition that pre-empts normal processing. Both internal and external conditions cause exceptions. External conditions that cause exceptions are interrupts from external devices, bus errors, coprocessor detected errors, and reset. Instructions, address errors, tracing, and breakpoints are internal conditions that cause exceptions. The TRAP, TRAPcc, TRAPV, cpTRAPcc, CHK, CHK2, RTE, and DIV instructions can

all generate exceptions as part of their normal execution. In addition, illegal instructions, privilege violations, and coprocessor protocol violations cause exceptions.

Exception processing, which is the transition from the normal processing of a program to the processing required for the exception condition, involves the exception vector table and an exception stack frame. The following paragraphs describe the vector table and a generalized exception stack frame. Exception processing is discussed in detail in **SECTION 8 EXCEPTION PROCESSING**. Coprocessor detected exceptions are discussed in detail in **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION**.

4.3.1 Exception Vectors

The vector base register (VBR) contains the base address of the 1024-byte exception vector table, which consists of 256 exception vectors. Exception vectors contain the memory addresses of routines that begin execution at the completion of exception processing. These routines perform a series of operations appropriate for the corresponding exceptions. Because the exception vectors contain memory addresses, each consists of one long word, except for the reset vector. The reset vector consists of two long words: the address used to initialize the interrupt stack pointer and the address used to initialize the program counter.

The address of an exception vector is derived from an 8-bit vector number and the VBR. The vector numbers for some exceptions are obtained from an external device; others are supplied automatically by the controller. The controller multiplies the vector number by four to calculate the vector offset, which it adds to the VBR. The sum is the memory address of the vector. All exception vectors are located in supervisor data space, except the reset vector, which is located in supervisor program space. Only the initial reset vector is fixed in the controller's memory map; once initialization is complete, there are no fixed assignments. Since the VBR provides the base address of the vector table, the vector table can be located anywhere in memory; it can even be dynamically relocated for each task that is executed by an operating system. Details of exception processing are provided in **SECTION 8 EXCEPTION PROCESSING**, and Table 8-1 lists the exception vector assignments.

4.3.2 Exception Stack Frame

Exception processing saves the most volatile portion of the current controller context on the top of the supervisor stack. This context is organized in a

format called the exception stack frame. This information always includes a copy of the status register, the program counter, the vector offset of the vector, and the frame format field. The frame format field identifies the type of stack frame. The RTE instruction uses the value in the format field to properly restore the information stored in the stack frame and to deallocate the stack space. The general form of the exception stack frame is illustrated in Figure 4-1. Refer to **SECTION 8 EXCEPTION PROCESSING** for a complete list of exception stack frames.

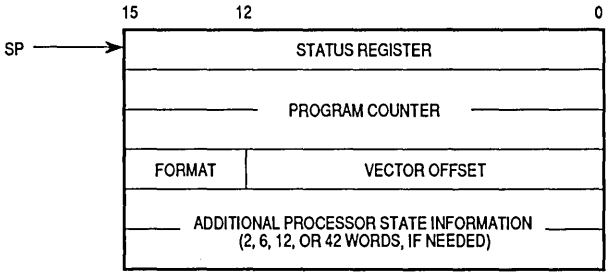


Figure 4-1. General Exception Stack Frame

SECTION 5
SIGNAL DESCRIPTION

This section contains brief descriptions of the input and output signals in their functional groups, as shown in Figure 5-1. Each signal is explained in a brief paragraph with reference to other sections that contain more detail about the signal and the related operations.

NOTE

In this section and in the remainder of the manual, **assertion** and **negation** are used to specify forcing a signal to a particular state. In particular, assertion and assert refer to a signal that is active or true; negation and negate indicate a signal that is inactive or false. These terms are used independently of the voltage level (high or low) that they represent.

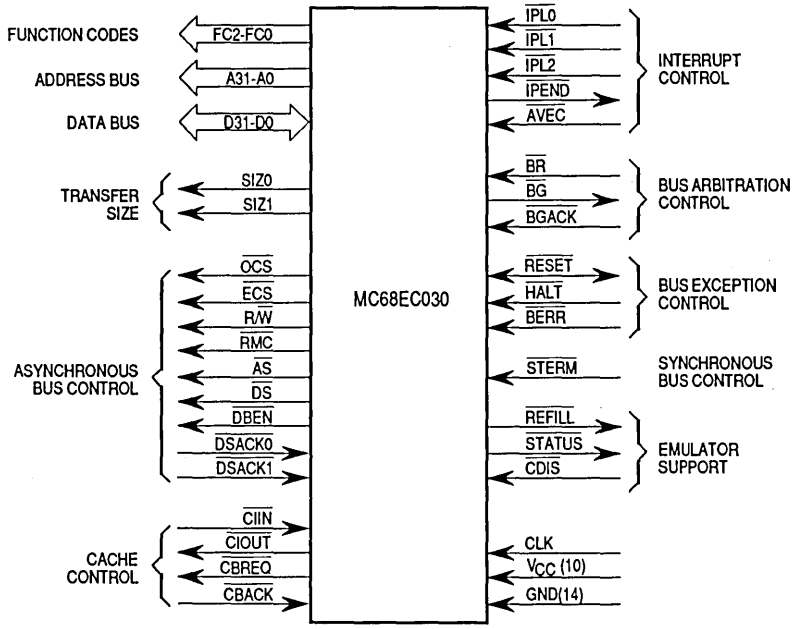


Figure 5-1. Functional Signal Groups

5.1 SIGNAL INDEX

The input and output signals for the MC68EC030 are listed in Table 5-1. Both the names and mnemonics are shown along with brief signal descriptions. For more detail on each signal, refer to the paragraph in this section named for the signal and the reference in that paragraph to a description of the related operations.

Guaranteed timing specifications for the signals listed in Table 5-1 can be found in M68EC030/D, *MC68EC030 Technical Summary*.

Table 5-1. Signal Index

Signal Name	Mnemonic	Function
Function Codes	FC0–FC2	3-bit function code used to identify the address space of each bus cycle.
Address Bus	A0–A31	32-bit address bus.
Data Bus	D0–D31	32-bit data bus used to transfer 8, 16, 24, or 32 bits of data per bus cycle.
Size	SIZ0/SIZ1	Indicates the number of bytes remaining to be transferred for this cycle. These signals, together with A0 and A1, define the active sections of the data bus.
Operand Cycle Start	$\overline{OCS}$	Identical operation to that of $\overline{ECS}$ except that $\overline{OCS}$ is asserted only during the first bus cycle of an operand transfer.
External Cycle Start	$\overline{ECS}$	Provides an indication that a bus cycle is beginning.
Read/Write	R/$\overline{W}$	Defines the bus transfer as a controller read or write.
Read-Modify-Write Cycle	$\overline{RMC}$	Provides an indicator that the current bus cycle is part of an indivisible read-modify-write operation.
Address Strobe	$\overline{AS}$	Indicates that a valid address is on the bus.
Data Strobe	$\overline{DS}$	Indicates that valid data is to be placed on the data bus by an external device or has been placed on the data bus by the MC68EC030.
Data Buffer Enable	$\overline{DBEN}$	Provides an enable signal for external data buffers.
Data Transfer and Size Acknowledge	$\overline{DSACK0}$/ $\overline{DSACK1}$	Bus response signals that indicate the requested data transfer operation is completed. In addition, these two lines indicate the size of the external bus port on a cycle-by-cycle basis and are used for asynchronous transfers.
Synchronous Termination	$\overline{STERM}$	Bus response signal that indicates a port size of 32 bits and that data may be latched on the next falling clock edge.
Cache Inhibit In	$\overline{CIIN}$	Prevents data from being loaded into the MC68EC030 instruction and data caches.
Cache Inhibit Out	$\overline{CIOUT}$	Reflects the CI bit in AT0 and AC1 in the ACU; indicates that external caches should ignore these accesses.

Table 5-1. Signal Index (Continued)

Signal Name	Mnemonic	Function
Cache Burst Request	$\overline{\text{CBREQ}}$	Indicates a burst request for the instruction or data cache.
Cache Burst Acknowledge	$\overline{\text{CBACK}}$	Indicates that the accessed device can operate in burst mode.
Interrupt Priority Level	$\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$	Provides an encoded interrupt level to the controller.
Interrupt Pending	$\overline{\text{IPEND}}$	Indicates that an interrupt is pending.
Autovector	$\overline{\text{AVEC}}$	Requests an autovector during an interrupt acknowledge cycle.
Bus Request	$\overline{\text{BR}}$	Indicates that an external device requires bus mastership.
Bus Grant	$\overline{\text{BG}}$	Indicates that an external device may assume bus mastership.
Bus Grant Acknowledge	$\overline{\text{BGACK}}$	Indicates that an external device has assumed bus mastership.
Reset	$\overline{\text{RESET}}$	System reset.
Halt	$\overline{\text{HALT}}$	Indicates that the controller should suspend bus activity.
Bus Error	$\overline{\text{BERR}}$	Indicates that an erroneous bus operation is being attempted.
Cache Disable	$\overline{\text{CDIS}}$	Dynamically disables the on-chip cache to assist emulator support.
Pipe Refill	$\overline{\text{REFILL}}$	Indicates when the MC68EC030 is beginning to fill pipeline.
Microsequencer Status	$\overline{\text{STATUS}}$	Indicates the state of the microsequencer.
Clock	CLK	Clock input to the controller.
Power Supply	V_{CC}	Power supply.
Ground	GND	Ground connection.
No Connect	NC	Do not connect.

5.2 FUNCTION CODE SIGNALS (FC0–FC2)

These three-state outputs identify the address space of the current bus cycle. Table 4-1 shows the relationship of the function code signals to the privilege levels and the address spaces. Refer to **4.2 ADDRESS SPACE TYPES** for more information.

5.3 ADDRESS BUS (A0–A31)

These three-state outputs provide the address for the current bus cycle, except in the CPU address space. Refer to **4.2 ADDRESS SPACE TYPES** for more information on the CPU address space. A31 is the most significant address signal. Refer to **7.1.2 Address Bus** for information on the address bus and its relationship to bus operation.

5.4 DATA BUS (D0–D31)

These three-state bidirectional signals provide the general-purpose data path between the MC68EC030 and all other devices. The data bus can transfer 8, 16, 24, or 32 bits of data per bus cycle. D31 is the most significant bit of the data bus. Refer to **7.1.4 Data Bus** for more information on the data bus and its relationship to bus operation.

5.5 TRANSFER SIZE SIGNALS (SIZ0, SIZ1)

These three-state outputs indicate the number of bytes remaining to be transferred for the current bus cycle. With A0, A1, $\overline{\text{DSACK0}}$, $\overline{\text{DSACK1}}$, and $\overline{\text{STERM}}$, SIZ0 and SIZ1 define the number of bits transferred on the data bus. Refer to **7.2.1 Dynamic Bus Sizing** for more information on the size signals and their use in dynamic bus sizing.

5.6 BUS CONTROL SIGNALS

The following signals control synchronous bus transfer operations for the MC68EC030.

5.6.1 Operand Cycle Start ($\overline{\text{OCS}}$)

This output signal indicates the beginning of the first external bus cycle for an instruction prefetch or a data operand transfer. $\overline{\text{OCS}}$ is not asserted for subsequent cycles that are performed due to dynamic bus sizing or operand misalignment. Refer to **7.1.1 Bus Control Signals** for information about the relationship of $\overline{\text{OCS}}$ to bus operation.

5.6.2 External Cycle Start ($\overline{ECS}$)

This output signal indicates the beginning of a bus cycle of any type. Refer to **7.1.1 Bus Control Signals** for information about the relationship of $\overline{ECS}$ to bus operation.

5.6.3 Read/Write (R/$\overline{W}$)

This three-state output signal defines the type of bus cycle. A high level indicates a read cycle; a low level indicates a write cycle. Refer to **7.1.1 Bus Control Signals** for information about the relationship of R/$\overline{W}$ to bus operation.

5.6.4 Read-Modify-Write Cycle ($\overline{RMC}$)

This three-state output signal identifies the current bus cycle as part of an indivisible read-modify-write operation; it remains asserted during all bus cycles of the read-modify-write operation. Refer to **7.1.1 Bus Control Signals** for information about the relationship of $\overline{RMC}$ to bus operation.

5.6.5 Address Strobe ($\overline{AS}$)

This three-state output indicates that a valid address is on the address bus. The function code, size, and read/write signals are also valid when $\overline{AS}$ is asserted. Refer to **7.1.3 Address Strobe** for information about the relationship of $\overline{AS}$ to bus operation.

5.6.6 Data Strobe ($\overline{DS}$)

During a read cycle, this three-state output indicates that an external device should place valid data on the data bus. During a write cycle, the data strobe indicates that the MC68EC030 has placed valid data on the bus. During two-clock synchronous write cycles, the MC68EC030 does not assert $\overline{DS}$. Refer to **7.1.5 Data Strobe** for more information about the relationship of $\overline{DS}$ to bus operation.

5.6.7 Data Buffer Enable ($\overline{DBEN}$)

This output is an enable signal for external data buffers. This signal may not be required in all systems. The timing of this signal may preclude its use in a system that supports two-clock synchronous bus cycles. Refer to **7.1.6 Data Buffer Enable** for more information about the relationship of $\overline{DBEN}$ to bus operation.

5.6.8 Data Transfer and Size Acknowledge ($\overline{DSACK0}$, $\overline{DSACK1}$)

These inputs indicate the completion of a requested data transfer operation. In addition, they indicate the size of the external bus port at the completion of each cycle. These signals apply only to asynchronous bus cycles. Refer to **7.1.7 Bus Cycle Termination Signals** for more information on these signals and their relationship to dynamic bus sizing.

5.6.9 Synchronous Termination ($\overline{STERM}$)

This input is a bus handshake signal indicating that the addressed port size is 32 bits and that data is to be latched on the next falling clock edge for a read cycle. This signal applies only to synchronous operation. Refer to **7.1.7 Bus Cycle Termination Signals** for more information about the relationship of $\overline{STERM}$ to bus operation.

5.7 CACHE CONTROL SIGNALS

The following signals relate to the on-chip caches.

5.7.1 Cache Inhibit Input ($\overline{CIIN}$)

This input signal prevents data from being loaded into the MC68EC030 instruction and data caches. It is a synchronous input signal and is interpreted on a bus-cycle-by-bus-cycle basis. $\overline{CIIN}$ is ignored during all write cycles. Refer to **6.1 ON-CHIP CACHE ORGANIZATION AND OPERATION** for information on the relationship of $\overline{CIIN}$ to the on-chip caches.

5.7.2 Cache Inhibit Output ($\overline{CIOUT}$)

This three-state output signal reflects the state of the CI bit in the access control unit registers (AC0 and AC1) for the referenced address, indicating that an external cache should ignore the bus transfer. When the referenced

address is within the specified area, the CI bit of the appropriate access control register controls the state of $\overline{\text{CIOUT}}$. Refer to **SECTION 9 ACCESS CONTROL UNIT** for more information about the access control unit. Also, refer to **SECTION 6 ON-CHIP CACHE MEMORIES** for the effect of $\overline{\text{CIOUT}}$ on the internal caches.

5.7.3 Cache Burst Request ($\overline{\text{CBREQ}}$)

This three-state output signal requests a burst mode operation to fill a line in the instruction or data cache. Refer to **6.1.3 Cache Filling** for filling information and **7.3.7 Burst Operation Cycles** for bus cycle information pertaining to burst mode operations.

5.7.4 Cache Burst Acknowledge ($\overline{\text{CBACK}}$)

This input signal indicates that the accessed device can operate in the burst mode and can supply at least one more long word for the instruction or data cache. Refer to **7.3.7 Burst Operation Cycles** for information about burst mode operation.

5.8 INTERRUPT CONTROL SIGNALS

The following signals are the interrupt control signals for the MC68EC030.

5.8.1 Interrupt Priority Level Signals ($\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$)

These input signals provide an indication of an interrupt condition and the encoding of the interrupt level from a peripheral or external prioritizing circuitry. $\overline{\text{IPL2}}$ is the most significant bit of the level number. For example, since the $\overline{\text{IPLn}}$ signals are active low, $\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$ equal to $5 corresponds to an interrupt request at interrupt level 2. Refer to **8.1.9 Interrupt Exceptions** for information on MC68EC030 interrupts.

5.8.2 Interrupt Pending ($\overline{\text{IPEND}}$)

This output signal indicates that an interrupt request has been recognized internally and exceeds the current interrupt priority mask in the status register (SR). This output is for use by external devices (coprocessors and other bus

masters, for example) to predict controller operation on the following instruction boundaries. Refer to **8.1.9 Interrupt Exceptions** for interrupt information. Also, refer to **7.4.1 Interrupt Acknowledge Bus Cycles** for bus information related to interrupts.

5.8.3 Autovector ($\overline{\text{AVEC}}$)

This input signal indicates that the MC68EC030 should generate an automatic vector during an interrupt acknowledge cycle. Refer to **7.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE** for more information about automatic vectors.

5

5.9 BUS ARBITRATION CONTROL SIGNALS

The following signals are the three bus arbitration control signals used to determine which device in a system is the bus master.

5.9.1 Bus Request ($\overline{\text{BR}}$)

This input signal indicates that an external device needs to become the bus master. This is typically a "wire-ORed" input (but does not need to be constructed from open-collector devices). Refer to **7.7 BUS ARBITRATION** for more information.

5.9.2 Bus Grant ($\overline{\text{BG}}$)

This output indicates that the MC68EC030 will release ownership of the bus master when the current controller bus cycle completes. Refer to **7.7.2 Bus Grant** for more information.

5.9.3 Bus Grant Acknowledge ($\overline{\text{BGACK}}$)

This input indicates that an external device has become the bus master. Refer to **7.7.3 Bus Grant Acknowledge** for more information.

5.10 BUS EXCEPTION CONTROL SIGNALS

The following signals are the bus exception control signals for the MC68EC030.

5.10.1 Reset ($\overline{\text{RESET}}$)

This bidirectional open-drain signal is used to initiate a system reset. An external reset signal resets the MC68EC030 as well as all external devices. A reset signal from the controller (asserted as part of the RESET instruction) resets external devices only; the internal state of the controller is not altered. Refer to **7.8 RESET OPERATION** for a description of reset bus operation and **8.1.1 Reset Exception** for information about the reset exception.

5.10.2 Halt ($\overline{\text{HALT}}$)

The halt signal indicates that the controller should suspend bus activity or, when used with $\overline{\text{BERR}}$, that the controller should retry the current cycle. Refer to **7.5 BUS EXCEPTION CONTROL CYCLES** for a description of the effects of $\overline{\text{HALT}}$ on bus operations.

5.10.3 Bus Error ($\overline{\text{BERR}}$)

The bus error signal indicates that an invalid bus operation is being attempted or, when used with $\overline{\text{HALT}}$, that the controller should retry the current cycle. Refer to **7.5 BUS EXCEPTION CONTROL CYCLES** for a description of the effects of $\overline{\text{BERR}}$ on bus operations.

5.11 EMULATOR SUPPORT SIGNALS

The following signals support emulation by providing a means for an emulator to disable the on-chip caches and access control unit and by supplying internal status information to an emulator. Refer to **SECTION 12 APPLICATIONS INFORMATION** for more detailed information on emulation support.

5.11.1 Cache Disable ($\overline{\text{CDIS}}$)

The cache disable signal dynamically disables the on-chip caches to assist emulator support. Refer to **6.1 ON-CHIP CACHE ORGANIZATION AND OPERATION** for information about the caches; refer to **SECTION 12 APPLICATIONS INFORMATION** for a description of the use of this signal by an emulator. $\overline{\text{CDIS}}$ does not flush the data and instruction caches; entries remain unaltered and become available again when $\overline{\text{CDIS}}$ is negated.

5.11.2 Pipeline Refill ($\overline{\text{REFILL}}$)

The pipeline refill signal indicates that the MC68EC030 is beginning to refill the internal instruction pipeline. Refer to **SECTION 12 APPLICATIONS INFORMATION** for a description of the use of this signal by an emulator.

5.11.3 Internal Microsequencer Status ($\overline{\text{STATUS}}$)

The microsequencer status signal indicates the state of the internal microsequencer. The varying number of clocks for which this signal is asserted indicates instruction boundaries, pending exceptions, and the halted condition. Refer to **SECTION 12 APPLICATIONS INFORMATION** for a description of the use of this signal by an emulator.

5.12 CLOCK (CLK)

The clock signal is the clock input to the MC68EC030. It is a TTL-compatible signal. Refer to **SECTION 12 APPLICATIONS INFORMATION** for suggestions on clock generation.

5.13 POWER SUPPLY CONNECTIONS

The MC68EC030 requires connection to a V_{CC} power supply, positive with respect to ground. The V_{CC} connections are grouped to supply adequate current for the various sections of the controller. The ground connections are similarly grouped. **SECTION 14 ORDERING INFORMATION AND MECHANICAL DATA** describes the groupings of V_{CC} and ground connections, and **SECTION 12 APPLICATIONS INFORMATION** describes a typical power supply interface.

5.14 NO CONNECTION

Do not connect to this pin.

5.15 SIGNAL SUMMARY

Table 5-2 provides a summary of the electrical characteristics of the signals discussed in this section.

Table 5-2. Signal Summary

Signal Function	Signal Name	Input/Output	Active State	Three-State
Function Codes	FC0–FC2	Output	High	Yes
Address Bus	A0–A31	Output	High	Yes
Data Bus	D0–D31	Input/Output	High	Yes
Transfer Size	SIZ0/SIZ1	Output	High	Yes
Operand Cycle Start	$\overline{\text{OCS}}$	Output	Low	No
External Cycle Start	$\overline{\text{ECS}}$	Output	Low	No
Read/Write	R/$\overline{\text{W}}$	Output	High/Low	Yes
Read-Modify-Write Cycle	$\overline{\text{RMC}}$	Output	Low	Yes
Address Strobe	$\overline{\text{AS}}$	Output	Low	Yes
Data Strobe	$\overline{\text{DS}}$	Output	Low	Yes
Data Buffer Enable	$\overline{\text{DBEN}}$	Output	Low	Yes
Data Transfer and Size Acknowledge	$\overline{\text{DSACK0}}$/ $\overline{\text{DSACK1}}$	Input	Low	—
Synchronous Termination	$\overline{\text{STERM}}$	Input	Low	—
Cache Inhibit In	$\overline{\text{CIIN}}$	Input	Low	—
Cache Inhibit Out	$\overline{\text{CIOUT}}$	Output	Low	Yes
Cache Burst Request	$\overline{\text{CBREQ}}$	Output	Low	Yes
Cache Burst Acknowledge	$\overline{\text{CBACK}}$	Input	Low	—
Interrupt Priority Level	$\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$	Input	Low	—
Interrupt Pending	$\overline{\text{IPEND}}$	Output	Low	No
Autovector	$\overline{\text{AVEC}}$	Input	Low	—
Bus Request	$\overline{\text{BR}}$	Input	Low	—
Bus Grant	$\overline{\text{BG}}$	Output	Low	No
Bus Grant Acknowledge	$\overline{\text{BGACK}}$	Input	Low	—
Reset	$\overline{\text{RESET}}$	Input/Output	Low	No
Halt	$\overline{\text{HALT}}$	Input	Low	—
Bus Error	$\overline{\text{BERR}}$	Input	Low	—
Cache Disable	$\overline{\text{CDIS}}$	Input	Low	—
Pipeline Refill	$\overline{\text{REFILL}}$	Output	Low	No
Microsequencer Status	$\overline{\text{STATUS}}$	Output	Low	No
Clock	CLK	Input	—	—
Power Supply	V_{CC}	Input	—	—
Ground	GND	Input	—	—
No Connect	NC	—	—	—

SECTION 6
ON-CHIP CACHE MEMORIES

The MC68EC030 embedded controller includes a 256-byte on-chip instruction cache and a 256-byte on-chip data cache that are accessed by addresses. These caches improve performance by reducing external bus activity and increasing instruction throughput.

Reduced external bus activity increases overall performance by increasing the availability of the bus for use by external devices (in systems with more than one bus master, such as a controller and a DMA device) without degrading the performance of the MC68EC030. An increase in instruction throughput results when instruction words and data required by a program are available in the on-chip caches and the time required to access them on the external bus is eliminated. Additionally, instruction throughput increases when instruction words and data can be accessed simultaneously.

As shown in Figure 6-1, the instruction cache and the data cache have separate on-chip address and data buses. The address buses are combined to provide the address to the access control unit (ACU). The MC68EC030 initiates an access to the appropriate cache for the requested instruction or data operand at the same time that it initiates an access for the cacheability of the address in the ACU. When a hit occurs in the instruction or data cache and the ACU does not invalidate the cacheability on a write, the information is transferred from the cache (on a read) or to the cache and the bus controller (on a write). When a hit does not occur, the address is used for an external bus cycle to obtain the instruction or operand. The ACU performs cacheability lookup in parallel with the cache lookup in case an external cycle is required, regardless of whether or not the required operand is located in one of the on-chip caches.

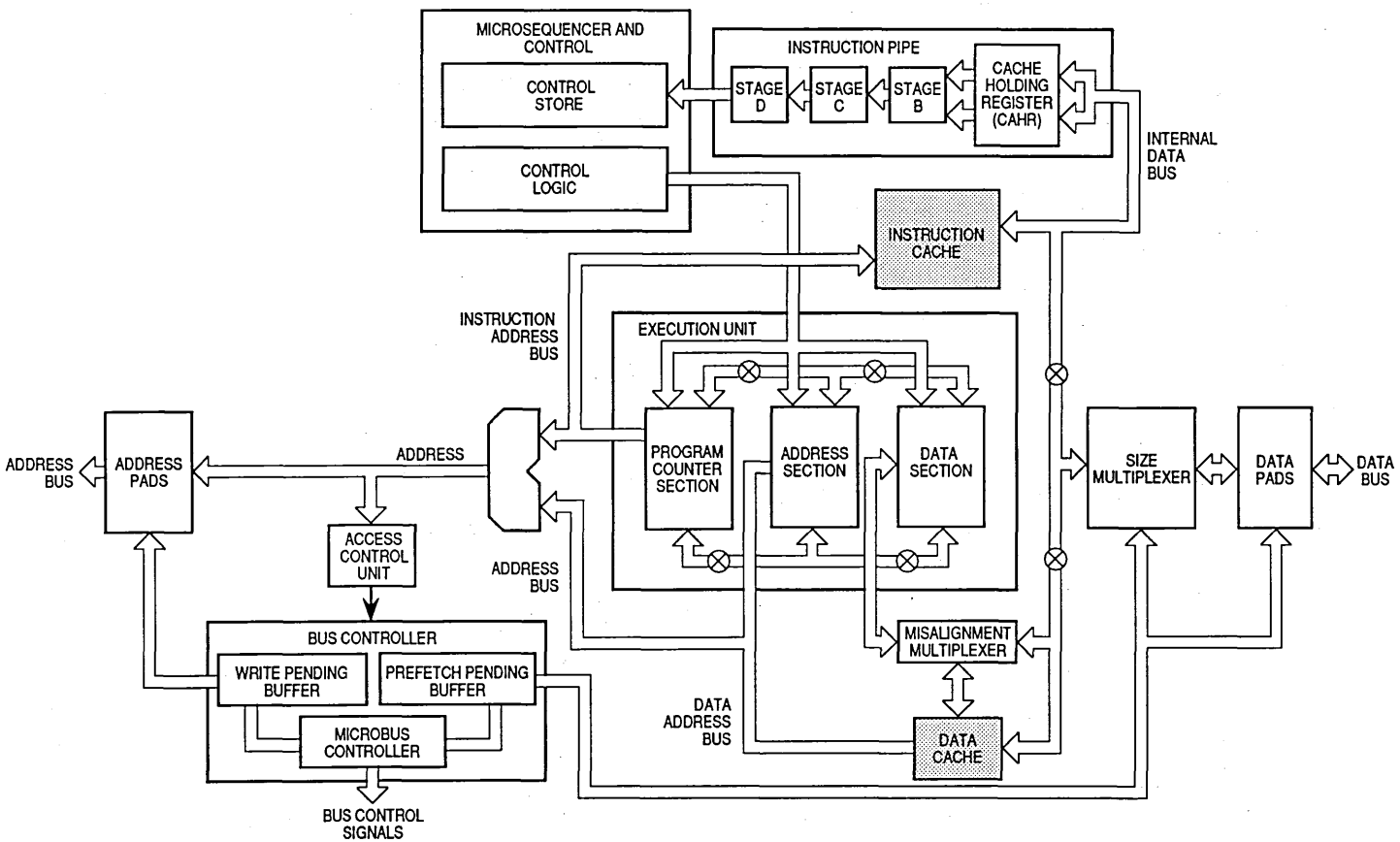


Figure 6-1. Internal Caches and the MC68EC030

6.1 ON-CHIP CACHE ORGANIZATION AND OPERATION

Both on-chip caches are 256-byte direct-mapped caches, each organized as 16 lines. Each line consists of four entries, and each entry contains four bytes. The tag field for each line contains a valid bit for each entry in the line; each entry is independently replaceable. When appropriate, the bus controller requests a burst mode operation to replace an entire cache line. The cache control register (CACR) is accessible by supervisor programs to control the operation of both caches.

System hardware can assert the cache disable ($\overline{\text{CDIS}}$) signal to disable both caches. The assertion of $\overline{\text{CDIS}}$ disables the caches, regardless of the state of the enable bits in CACR. $\overline{\text{CDIS}}$ is primarily intended for use by in-circuit emulators.

Another input signal, cache inhibit in ($\overline{\text{CIIN}}$), inhibits caching of data reads or instruction prefetches on a bus-cycle by bus-cycle basis. Examples of data that should not be cached are data for I/O devices and data from memory devices that cannot supply a full port width of data, regardless of the size of the required operand.

Subsequent paragraphs describe how $\overline{\text{CIIN}}$ is used during the filling of the caches.

An output signal, cache inhibit out ($\overline{\text{CIOUT}}$), reflects the state of the cache inhibit (CI) bit in the ACU access control register that corresponds to that address. When the appropriate CI bit is set for either a read or a write access, an external bus cycle is required. $\overline{\text{CIOUT}}$ is asserted and the instruction and data caches are ignored for the access. This signal can also be used by external hardware to inhibit caching in external caches.

Whenever a read access occurs and the required instruction word or data operand is resident in the appropriate on-chip cache (no external bus cycle is required), the ACU is completely ignored (see next two paragraphs). Therefore, the state of the corresponding CI bits in the ACU are also ignored. The ACU controls cacheability of all accesses that require external bus cycles; protections are checked, and the $\overline{\text{CIOUT}}$ signal is asserted appropriately.

An external access is defined as cacheable for either the instruction or data cache when all the following conditions apply:

- The cache is enabled with the appropriate bit in the CACR set.
- The $\overline{\text{CDIS}}$ signal is negated.
- The $\overline{\text{CIIN}}$ signal is negated for the access.
- The $\overline{\text{CIOUT}}$ signal is negated for the access.
- The ACU validates the access.

Because both the data and instruction caches are referenced by addresses, they should be flushed during a memory swap or when the ACU is first enabled. In addition, if an address is currently marked as cacheable and is later changed to the noncacheable (due to a context switch) *entries in the on-chip instruction or data cache corresponding to the old context must be first cleared (invalidated).* Otherwise, if on-chip cache entries are valid for addresses marked noncacheable, controller operation is unpredictable.

Data read and write accesses to the same address should also have consistent cacheability status to ensure that the data in the cache remains consistent with external memory. For example, if $\overline{\text{CIOUT}}$ is negated for read accesses within an address range and the ACU configuration is changed so that $\overline{\text{CIOUT}}$ is subsequently asserted for write accesses within the same range, those write accesses do not update data in the cache, and stale data may result. Similarly, when the ACU maps multiple function code addresses to the same address, all accesses to those addresses should have the same cacheability status.

6.1.1 Instruction Cache

The instruction cache is organized with a line size of four long words, as shown in Figure 6-2. Each of these long words is considered a separate cache entry as each has a separate valid bit. All four entries in a line have the same tag address. Burst filling all four long words can be advantageous when the time spent in filling the line is not long relative to the equivalent bus-cycle time for four nonburst long-word accesses, because of the probability that the contents of memory adjacent to or close to a referenced operand or instruction is also required by subsequent accesses. Dynamic RAMs supporting fast access modes (page, nibble, or static column) are easily employed to support the MC68EC030 burst mode.

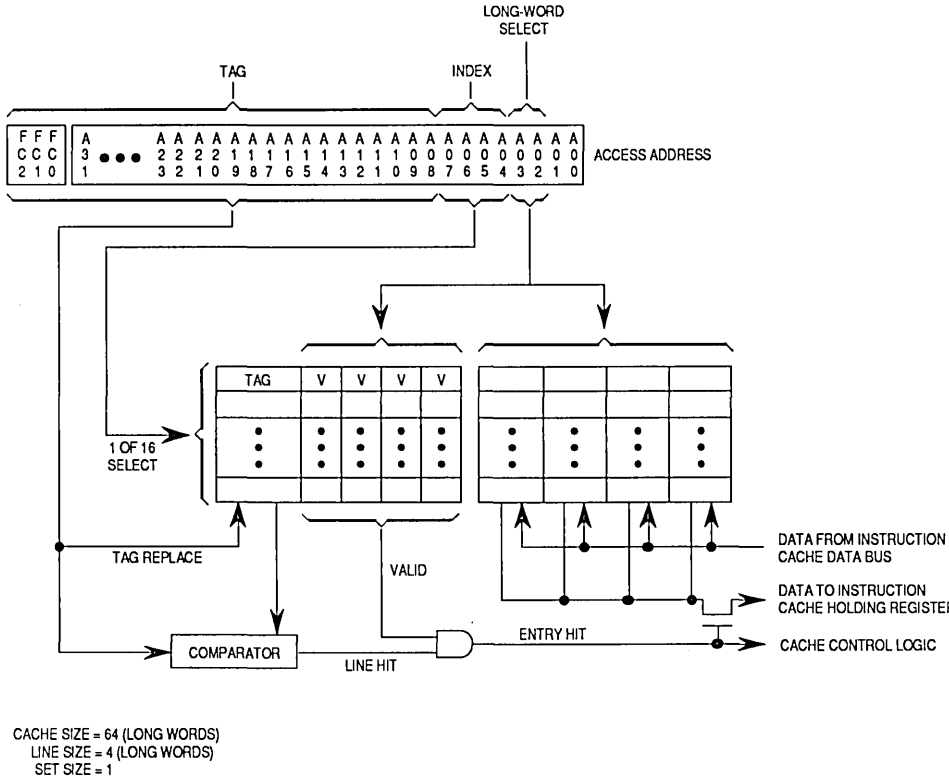


Figure 6-2. On-Chip Instruction Cache Organization

When enabled, the instruction cache is used to store instruction prefetches (instruction words and extension words) as they are requested by the CPU. Instruction prefetches are normally requested from sequential memory addresses except when a change of program flow occurs (e.g., a branch taken) or when an instruction is executed that can modify the status register, in which cases the instruction pipe is automatically flushed and refilled. The output signal $\overline{\text{REFILL}}$ indicates this condition. For more information on the operation of this signal, refer to **SECTION 12 APPLICATIONS INFORMATION.**

In the instruction cache, each of the 16 lines has a tag consisting of the 24 most significant address bits, the FC2 function code bit (used to distinguish between user and supervisor accesses), and the four valid bits (one corresponding to each long word). Refer to Figure 6-2 for the instruction cache organization. Address bits A7–A4 select one of 16 lines and its associated tag. The comparator compares the address and function code bits in the

selected tag with address bits A31–A8 and FC2 from the internal prefetch request to determine if the requested word is in the cache. A cache hit occurs when there is a tag match and the corresponding valid bit (selected by A3–A2) is set. On a cache hit, the word selected by address bit A1 is supplied to the instruction pipe.

When the address and function code bits do not match or the requested entry is not valid, a miss occurs. The bus controller initiates a long-word prefetch operation for the required instruction word and loads the cache entry, provided the entry is cacheable. A burst mode operation may be requested to fill an entire cache line. If the function code and address bits match and the corresponding long word is not valid (but one or more of the other three valid bits for that line are set) a single entry fill operation replaces the required long word only, using a normal prefetch bus cycle or cycles (no burst).

6.1.2 Data Cache

The data cache stores data references to any address space except CPU space (FC = $7), including those references made with PC relative addressing modes and accesses made with the MOVES instruction. Operation of the data cache is similar to that of the instruction cache, except for the address comparison and cache filling operations. The tag of each line in the data cache contains function code bits FC0, FC1, and FC2 in addition to address bits A31–A8. The cache control circuitry selects the tag using bits A7–A4 and compares it to the corresponding bits of the access address to determine if a tag match has occurred. Address bits A3–A2 select the valid bit for the appropriate long word in the cache to determine if an entry hit has occurred. Misaligned data transfers may span two data cache entries. In this case, the controller checks for a hit one entry at a time. Therefore, it is possible that a portion of the access results in a hit and a portion results in a miss. The hit and miss are treated independently. Figure 6-3 illustrates the organization of the data cache.

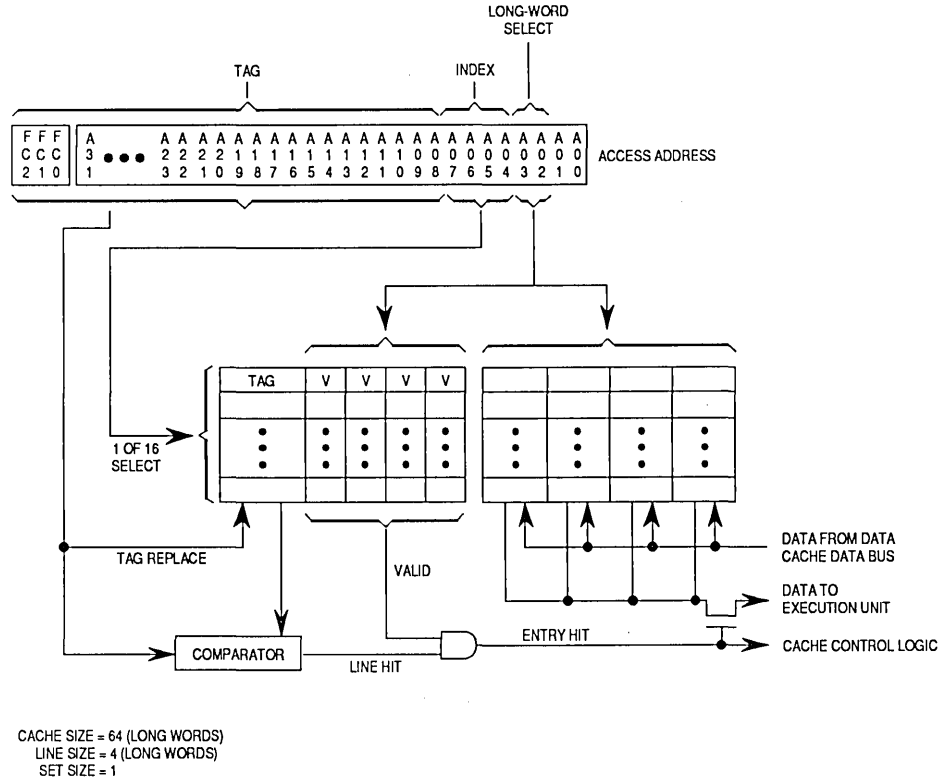


Figure 6-3. On-Chip Data Cache Organization

The operation of the data cache differs for read and write cycles. A data read cycle operates exactly like an instruction cache read cycle; when a miss occurs, an external cycle is initiated to obtain the operand from memory, and the data is loaded into the cache if the access is cacheable. In the case of a misaligned operand that spans two cache entries, two long words are required from memory. Burst mode operation may also be initiated to fill an entire line of the data cache. Read accesses from the CPU address space and address translation table search accesses are not stored in the data cache.

The data cache on the MC68EC030 is a writethrough cache. When a hit occurs on a write cycle, the data is written both to the cache and to external memory (provided the ACU validates the access), regardless of the operand size and even if the cache is frozen. If the ACU determines that the access is invalid, the write is aborted, the corresponding entry is invalidated, and a bus error exception is taken. Since the write to the cache completes before the write to external memory, the cache contains the new value even if the external

write terminates in a bus error. The value in the data cache might be used by another instruction before the external write cycle has completed, although this should not have any adverse consequences. Refer to **7.6 BUS SYNCHRONIZATION** for the details of bus synchronization.

6.1.2.1 WRITE ALLOCATION. The supervisor program can configure the data cache for either of two types of allocation for data cache entries that miss on write cycles. The state of the write allocation (WA) bit in the cache control register specifies either no write allocation or write allocation with partial validation of the data entries in the cache on writes.

When no write allocation is selected (WA = 0), write cycles that miss do not alter the data cache contents. In this mode, the controller does not replace entries in the cache during write operations. The cache is updated only during a write hit.

When write allocation is selected (WA = 1), the controller always updates the data cache on cacheable write cycles, but only validates an updated entry that hits or an entry that is updated with long-word data that is long-word aligned. When a tag miss occurs on a write of long-word data that is long-word aligned, the corresponding tag is replaced, and only the long word being written is marked as valid. The other three entries in the cache line are invalidated when a tag miss occurs on a misaligned long-word write or on a byte or word write, the data is not written in the cache, the tag is unaltered, and the valid bit(s) are cleared. Thus, an aligned long-word data write may replace a previously valid entry; whereas, a misaligned data write or a write of data that is not long word may invalidate a previously valid entry or entries.

Write allocation eliminates stale data that may reside in the cache by allowing the same location to be accessed by both supervisor and user mode cycles. Stale data conditions can arise when operating in the no-write-allocation mode and all the following conditions are satisfied:

- Multiple mapping (supervisor/user space aliasing) is allowed by the operating system.
- A read cycle loads a value for an aliased address into the data cache.
- A write cycle occurs, referencing the same aliased object as above but using a different privilege level, causing a cache miss and no update to the cache.
- The object is then read using the first alias, which provides stale data from the cache.

In this case, the data in the cache no longer matches that in memory and is stale. Since the write-allocation mode updates the cache during write cycles, the data in the cache remains consistent with memory. Note that when $\overline{\text{CIOUT}}$ is asserted, the data cache is completely ignored, even on write cycles operating in the write-allocation mode. Also note that since the $\overline{\text{CIIN}}$ signal is ignored on write cycles, cache entries may be created for noncacheable data (when $\overline{\text{CIIN}}$ is asserted on a write) when operating in the write-allocation mode. Figure 6-4 shows the manner in which each mode operates in two different situations.

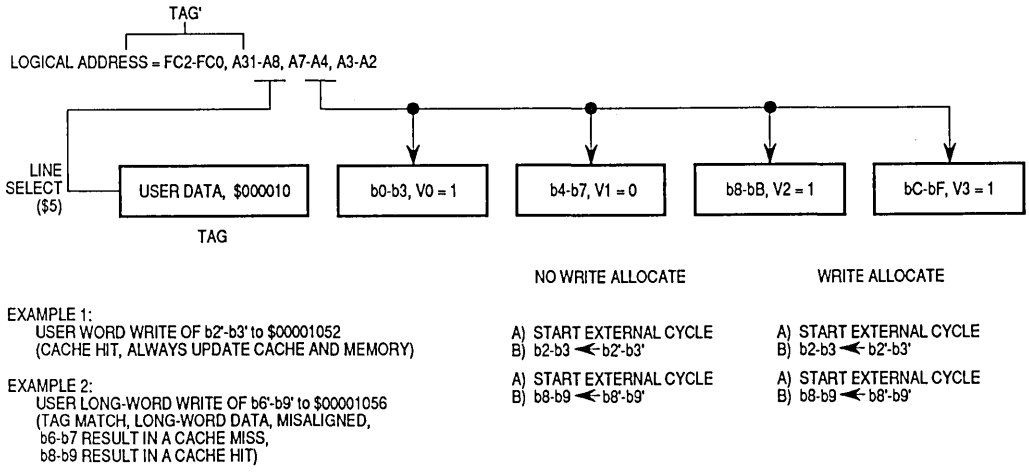


Figure 6-4. No-Write-Allocation and Write-Allocation Mode Examples

6.1.2.2 READ-MODIFY-WRITE ACCESSES. The read portion of a read-modify-write cycle is always forced to miss in the data cache. However, if the system allows internal caching of read-modify-write cycle operands ($\overline{\text{CIOUT}}$ and $\overline{\text{CIIN}}$ both negated), the controller either uses the data read from memory to update a matching entry in the data cache or creates a new entry with the read data in the case of no matching entry. The write portion of a read-modify-write operation also updates a matching entry in the data cache. In the case of a cache miss on the write, the allocation of a new cache entry for the data being written is controlled by the WA bit.

6.1.3 Cache Filling

The bus controller can load either cache in either of two ways:

- Single entry mode
- Burst fill mode

In the single entry mode, the bus controller loads a single long-word entry of a cache line. In the burst fill mode, an entire line (four long words) can be filled. Refer to **SECTION 7 BUS OPERATION** for detailed information about the bus cycles required for both modes.

6.1.3.1 SINGLE ENTRY MODE. When a cacheable access is initiated and a burst mode operation is not requested by the MC68EC030 or is not supported by external hardware, the bus controller transfers a single long word for the corresponding cache entry. An entire long word is required. If the port size of the responding device is smaller than 32 bits, the MC68EC030 executes all bus cycles necessary to fill the long word.

When a device cannot supply its entire port width of data, regardless of the size of the transfer, the responding device must consistently assert the cache inhibit input ($\overline{\text{CIIN}}$) signal. For example, a 32-bit port must always supply 32 bits, even for 8- and 16-bit transfers; a 16-bit port must supply 16 bits, even for 8-bit transfers. The MC68EC030 assumes that a 32-bit termination signal for the bus cycle indicates availability of 32 valid data bits, even if only 16 or 8 bits are requested. Similarly, the controller assumes that a 16-bit termination signal indicates that all 16 bits are valid. If the device cannot supply its full port width of data, it must assert $\overline{\text{CIIN}}$ for all bus cycles corresponding to a cache entry.

When a cacheable read cycle provides data with both $\overline{\text{CIIN}}$ and $\overline{\text{BERR}}$ negated, the MC68EC030 attempts to fill the cache entry. Figure 6-5 shows the organization of a line of data in the caches. The notation b0, b1, b2, and so forth identifies the bytes within the line. For each entry in the line, a valid bit in the associated tag corresponds to a long-word entry to be loaded. Since a single valid bit applies to an entire long word, a single entry mode operation must provide a full 32 bits of data. Ports less than 32 bits wide require several read cycles for each entry.

Figure 6-5 shows an example of a byte data operand read cycle starting at byte address $03 from an 8-bit port. Provided the data item is cacheable, this operation results in four bus cycles. The first cycle requested by the MC68EC030 reads a byte from address $03. The 8-bit $\overline{\text{DSACKx}}$ response

causes the MC68EC030 to fetch the remainder of the long word starting at address $00. The bytes are latched in the following order: b3, b0, b1, and b2. Note that during cache loading operations, devices must indicate the same port size consistently throughout all cycles for that long-word entry in the cache.

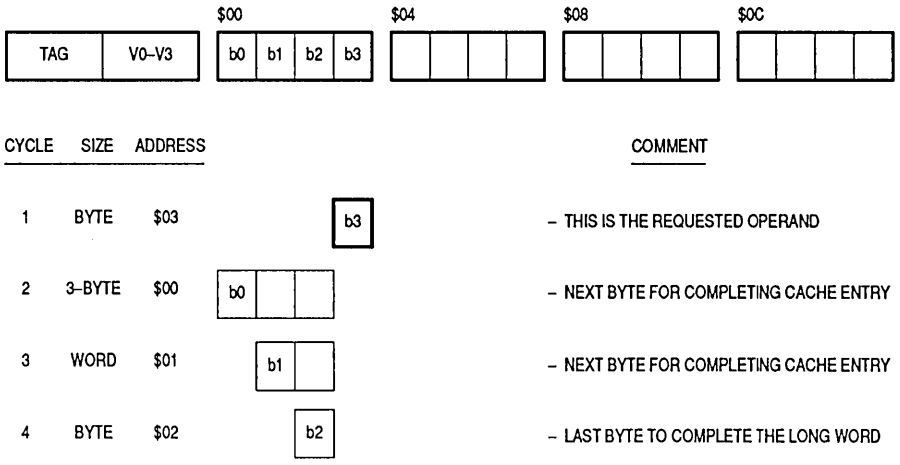


Figure 6-5. Single Entry Mode Operation — 8-Bit Port

Figure 6-6 shows the access of a byte data operand from a 16-bit port. This operation requires two read cycles. The first cycle requests the byte at address $03. If the device responds with a 16-bit $\overline{\text{DSACK}}$x encoding, the word at address $02 (including the requested byte) is accepted by the MC68EC030. The second cycle requests the word at address $00. Since the device again responds with a 16-bit $\overline{\text{DSACK}}$x encoding, the remaining two bytes of the long word are latched, and the cache entry is filled.

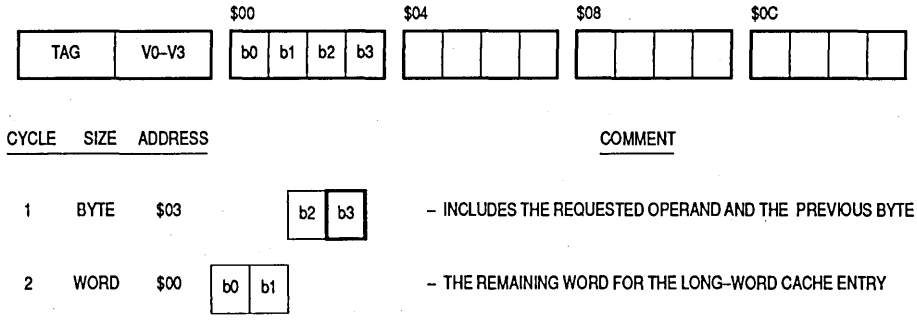


Figure 6-6. Single Entry Mode Operation — 16-Bit Port

With a 32-bit port, the same operation is shown in Figure 6-7. Only one read cycle is required. All four bytes (including the requested byte) are latched during the cycle.

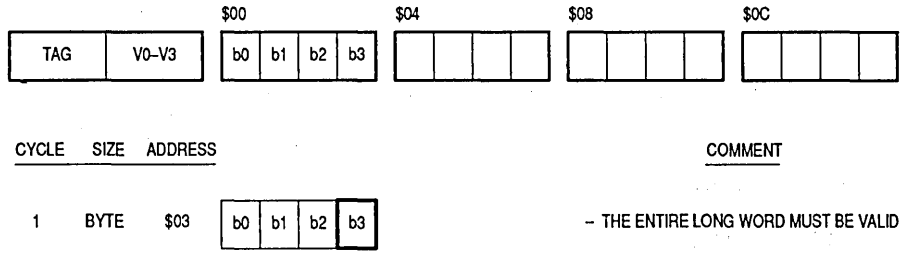


Figure 6-7. Single Entry Mode Operation — 32-Bit Port

If a requested access is misaligned and spans two cache entries, the bus controller attempts to fill both associated long-word cache entries. An example of this is an operand request for a long word on an odd-word boundary. The MC68EC030 first fetches the initial byte(s) of the operand (residing in the first long word) and then requests the remaining bytes to fill that cache entry (if the port size is less than 32 bits) before it requests the remainder of the operand and corresponding long word to fill the second cache entry. If the port size is 32 bits, the controller performs two accesses, one for each cache entry.

Figure 6-8 shows a misaligned access of a long word at address $06 from an 8-bit port requiring eight bus cycles to complete. Reading this long-word

operand requires eight read cycles, since accesses to all eight addresses return 8-bit port-size encodings. These cycles fetch the two cache entries that the requested long-word spans. The first cycle requests a long word at address $06 and accepts the first requested byte (b6). The subsequent transfers of the first long word are performed in the following order: b7, b4, b5. The remaining four read cycles transfer the four bytes of the second cache entry. The sequence of access for the entire operation is b6, b7, b4, b5, b8, b9, bA, and bB.

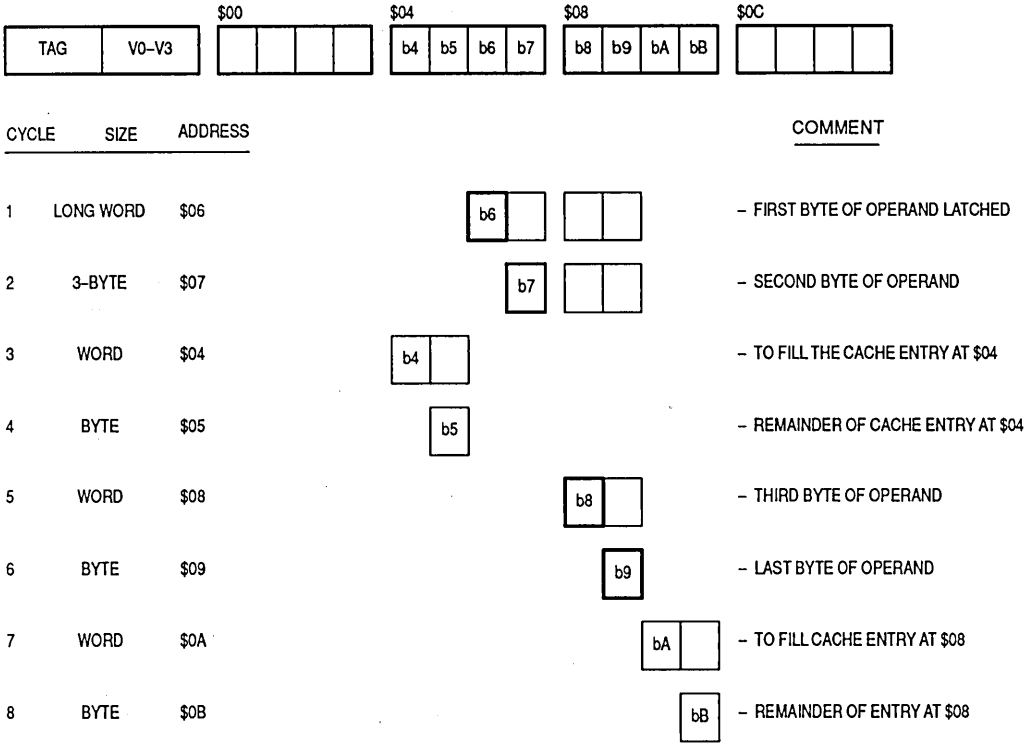


Figure 6-8. Single Entry Mode Operation — Misaligned Long Word and 8-Bit Port

The next example, shown in Figure 6-9, is a read of a misaligned long-word operand from devices that return 16-bit DSACKx encodings. The controller accepts the first portion of the operand, the word from address $06, and requests a word from address $04 to fill the cache entry. Next, the controller reads the word at address $08, the second portion of the operand, and stores it in the cache also. Finally, the controller accesses the word at $0A to fill the second long-word cache entry.

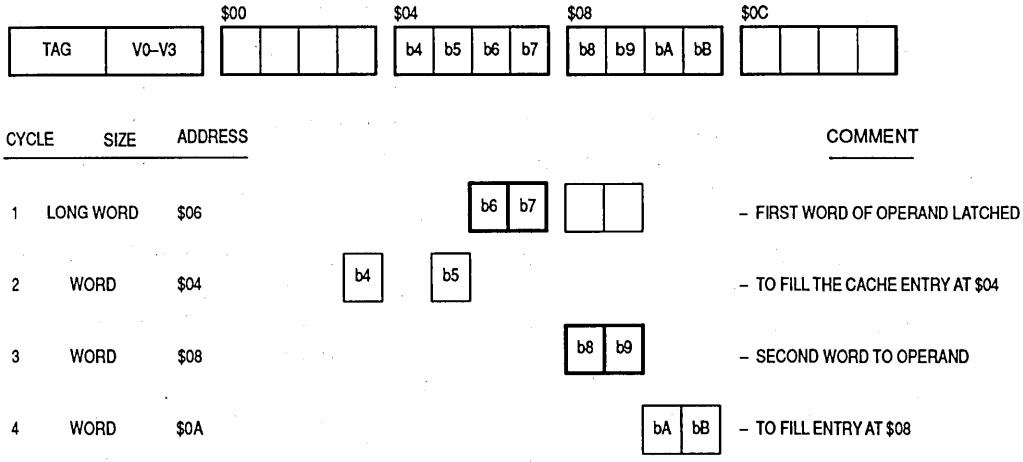


Figure 6-9. Single Entry Mode Operation — Misaligned Long Word and 16-Bit Port

Two read cycles are required for a misaligned long-word operand transfer from devices that return 32-bit DSACKx encodings. As shown in Figure 6-10, the first read cycle requests the long word at address $06 and latches the long word at address $04. The second read cycle requests and latches the long word corresponding to the second cache entry at address $08. Two read cycles are also required if STERM is used to indicate a 32-bit port instead of the 32-bit DSACKx encoding.

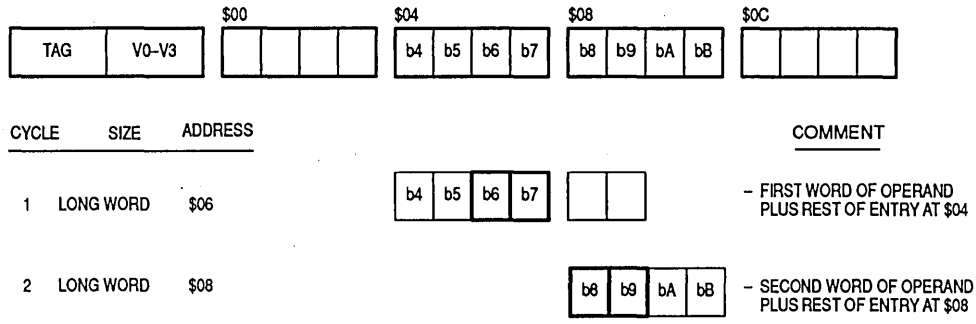


Figure 6-10. Single Entry Mode Operation — Misaligned Long Word and 32-Bit $\overline{\text{DSACK}}$x Port

If all bytes of a long word are cacheable, $\overline{\text{CIIN}}$ must be negated for all bus cycles required to fill the entry. If any byte is not cacheable, $\overline{\text{CIIN}}$ must be asserted for all corresponding bus cycles. The assertion of the $\overline{\text{CIIN}}$ signal prevents the caches from being updated during read cycles. Write cycles (including the write portion of a read-modify-write cycle) ignore the assertion of the $\overline{\text{CIIN}}$ signal and may cause the data cache to be altered, depending on the state of the cache (whether or not the write cycle hits), the state of the WA bit in the CACR, and the conditions indicated by the ACU.

The occurrence of a bus error while attempting to load a cache entry aborts the entry fill operation but does not necessarily cause a bus error exception. If the bus error occurs on a read cycle for a portion of the required operand (not the remaining bytes of the cache entry) to be loaded into the data cache, the controller immediately takes a bus error exception. If the read cycle in error is made only to fill the data cache (the data is not part of the target operand), no exception occurs, but the corresponding entry is marked invalid. For the instruction cache, the controller marks the entry as invalid, but only takes an exception if the execution unit attempts to use the instruction word(s).

6.1.3.2 BURST MODE FILLING. Burst mode filling is enabled by bits in the cache control register. The data burst enable bit must be set to enable burst filling of the data cache. Similarly, the instruction burst enable bit must be set to

enable burst filling of the instruction cache. When burst filling is enabled and the corresponding cache is enabled, the bus controller requests a burst mode fill operation in either of these cases:

- A read cycle for either the instruction or data cache misses due to the indexed tag not matching.
- A read cycle tag matches, but all long words in the line are invalid.

The bus controller requests a burst mode fill operation by asserting the cache burst request signal ($\overline{\text{CBREQ}}$). The responding device may sequentially supply one to four long words of cacheable data, or it may assert the cache inhibit input signal ($\overline{\text{CIIN}}$) when the data in a long word is not cacheable. If the responding device does not support the burst mode and it terminates cycles with $\overline{\text{STERM}}$, it should not acknowledge the request with the assertion of the cache burst acknowledge ($\overline{\text{CBACK}}$) signal. The MC68030 ignores the assertion of $\overline{\text{CBACK}}$ during cycles terminated with $\overline{\text{DSACK}}$x.

The cache burst request signal ($\overline{\text{CBREQ}}$) requests burst mode operation from the referenced external device. To operate in the burst mode, the device or external hardware must be able to increment the low-order address bits if required, and the current cycle must be a 32-bit synchronous transfer ($\overline{\text{STERM}}$ must be asserted) as described in **SECTION 7 BUS OPERATION**. The device must also assert $\overline{\text{CBACK}}$ (at the same time as $\overline{\text{STERM}}$) at the end of the cycle in which the MC68EC030 asserts $\overline{\text{CBREQ}}$. $\overline{\text{CBACK}}$ causes the controller to continue driving the address and bus control signals and to latch a new data value for the next cache entry at the completion of each subsequent cycle (as defined by $\overline{\text{STERM}}$), for a total of up to four cycles (until four long words have been read).

When a cache burst is initiated, the first cycle attempts to load the cache entry corresponding to the instruction word or data item explicitly requested by the execution unit. The subsequent cycles are for the subsequent entries in the cache line. In the case of a misaligned transfer when the operand spans two cache entries within a cache line, the first cycle corresponds to the cache entry containing the portion of the operand at the lower address.

Figure 6-11 illustrates the four cycles of a burst operation and shows that the second, third, and fourth cycles are run in burst mode. A distinction is made between the first cycle of a burst operation and the subsequent cycles because the first cycle is requested by the microsequencer and the burst fill cycles are requested by the bus controller. Therefore, when data from the first cycle is returned, it is immediately available for the execution unit (EU). However, data from the burst fill cycles is not available to the EU until the

burst operation is complete. Since the microsequencer makes two separate requests for misaligned data operands, only the first portion of the misaligned operand returned during a burst operation is available to the EU after the first cycle is complete. The microsequencer must wait for the burst operation to complete before requesting the second portion of the operand. Normally, the request for the second portion results in a data cache hit unless the second cycle of the burst operation terminates abnormally.

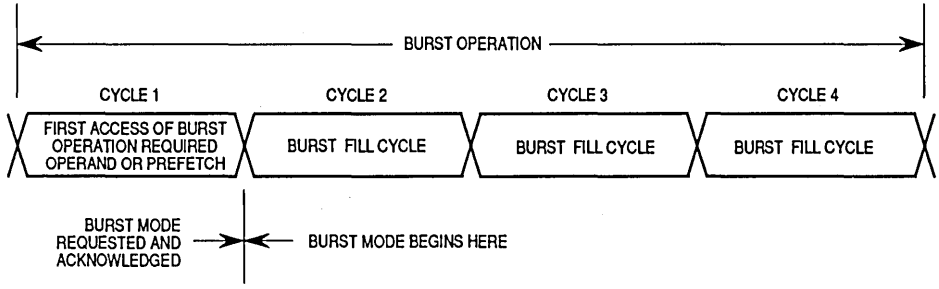


Figure 6-11. Burst Operation Cycles and Burst Mode

The bursting mechanism allows addresses to wrap around so that the entire four long words in the cache line can be filled in a single burst operation, regardless of the initial address and operand alignment. Depending on the structure of the external memory system, address bits A2 and A3 may have to be incremented externally to select the long words in the proper order for loading into the cache. The MC68EC030 holds the entire address bus constant for the duration of the burst cycle. Figure 6-12 shows an example of this address wraparound. The initial cycle is a long-word access from address $6. Because the responding device returns $\overline{\text{CBACK}}$ and $\overline{\text{STERM}}$ (signaling a 32-bit port), the entire long word at base address $04 is transferred. Since the initial address is $06 when $\overline{\text{CBREQ}}$ is asserted, the next entry to be burst filled into the cache should correspond to address $08, then $0C, and last, $00. This addressing is compatible with existing nibble-mode dynamic RAMs, and can be supported by page and static column modes with an external modulo 4 counter for A2 and A3.

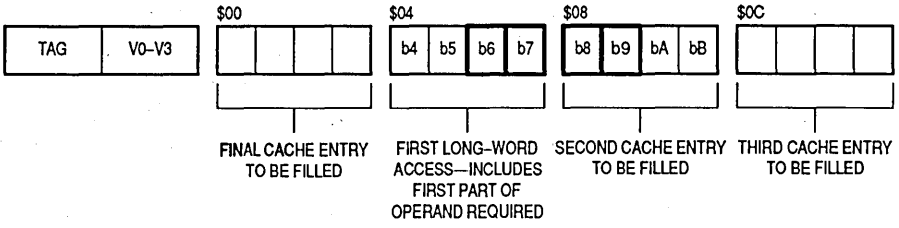


Figure 6-12. Burst Filling Wraparound Example

The MC68EC030 does not assert $\overline{\text{CBREQ}}$ during the first portion of a misaligned access if the remainder of the access does not correspond to the same cache line. Figure 6-13 shows an example in which the first portion of a misaligned access is at address $0F. With a 32-bit port, the first access corresponds to the cache entry at address $0C, which is filled using a single-entry load operation. The second access, at address $10 corresponding to the second cache line, requests a burst fill and the controller asserts $\overline{\text{CBREQ}}$. During this burst operation, long words $10, $14, $18, and $1C are all filled in that order.

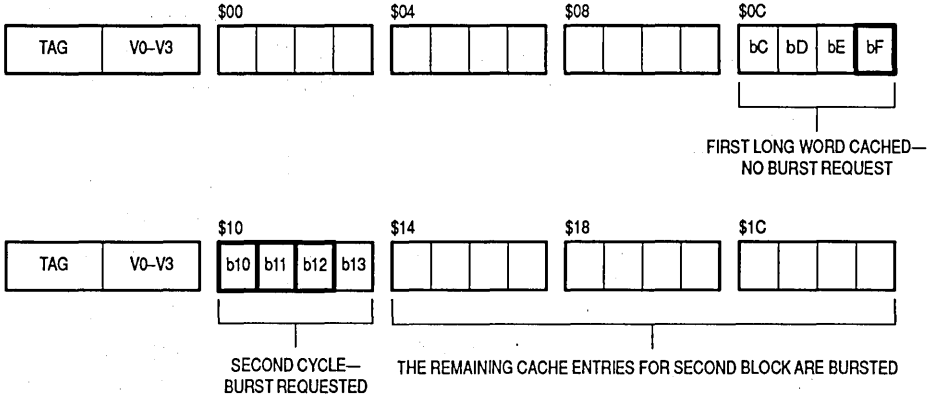


Figure 6-13. Deferred Burst Filling Example

The controller does not assert $\overline{\text{CBREQ}}$ if any of the following conditions exist:

- The appropriate cache is not enabled
- Burst filling for the cache is not enabled
- The cache freeze bit for the appropriate cache is set
- The current operation is the read portion of a read-modify-write operation
- The ACU has inhibited caching for the current address
- The cycle is for the first access of an operand that spans two cache lines (crosses a modulo 16 boundary)

Additionally, the assertion of $\overline{\text{CIIN}}$ and $\overline{\text{BERR}}$ and the premature negation of $\overline{\text{CBACK}}$ affect burst operation as described in the following paragraphs.

The assertion of $\overline{\text{CIIN}}$ during the first cycle of a burst operation causes the data to be latched by the controller, and if the requested operand is aligned (the entire operand is latched in the first cycle), the data is passed on to the instruction pipe or execution unit. However, the data is not loaded into its corresponding cache. In addition, the MC68EC030 negates $\overline{\text{CBREQ}}$, and the burst operation is aborted. If a portion of the requested operand remains to be read (due to misalignment), a second read cycle is initiated at the appropriate address with $\overline{\text{CBREQ}}$ negated.

The assertion of $\overline{\text{CIIN}}$ during the second, third, or fourth cycle of a burst operation prevents the data during that cycle from being loaded into the appropriate cache and causes $\overline{\text{CBREQ}}$ to negate, aborting the burst operation. However, if the data for the cycle contains part of the requested operand, the execution unit uses that data.

The premature negation of the $\overline{\text{CBACK}}$ signal during the burst operation causes the current cycle to complete normally, loading the data successfully transferred into the appropriate cache. However, the burst operation aborts and $\overline{\text{CBREQ}}$ negates.

A bus error occurring during a burst operation also causes the burst operation to abort. If the bus error occurs during the first cycle of a burst (i.e., before burst mode is entered), the data read from the bus is ignored, and the entire associated cache line is marked "invalid". If the access is a data cycle, exception processing proceeds immediately. If the cycle is for an instruction fetch, a bus error exception is made pending. This bus error is processed only if the execution unit attempts to use either instruction word. Refer to **11.2.2 Instruction Pipe** for more information about pipeline operation.

For either cache, when a bus error occurs after the burst mode has been entered (that is, on the second cycle or later), the cache entry corresponding to that cycle is marked invalid, but the controller does not take an exception (the microsequencer has not yet requested the data). In the case of an instruction cache burst, the data from the aborted cycle is completely ignored. Pending instruction prefetches are still pending and are subsequently run by the controller. If the second cycle is for a portion of a misaligned data operand fetch and a bus error occurs, the controller terminates the burst operation and negates $\overline{\text{CBREQ}}$. Once the burst terminates, the microsequencer requests a read cycle for the second portion. Since the burst terminated abnormally for the second cycle of the burst, the data cache results in a miss, and a second external cycle is required. If $\overline{\text{BERR}}$ is again asserted, the MC68EC030 then takes an exception.

On the initial access of a burst operation, a "retry" (indicated by the assertion of $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$) causes the controller to retry the bus cycle and assert $\overline{\text{CBREQ}}$ again. However, signaling a retry with simultaneous $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ during the second, third, or fourth cycle of a burst operation does not cause a retry operation, even if the requested operand is misaligned. Assertion of $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ during burst fill cycles of a burst operation causes independent bus error and halt operations. The controller remains halted until $\overline{\text{HALT}}$ is negated, and then handles the bus error as described in the previous paragraphs.

6.2 CACHE RESET

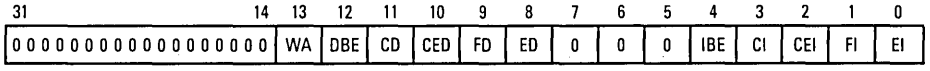
When a hardware reset of the controller occurs, all valid bits of both caches are cleared. The cache enable bits, burst enable bits, and the freeze bits in the cache control register (CACR) for both caches (refer to Figure 6-14) are also cleared, effectively disabling both caches. The WA bit in the CACR is also cleared.

6.3 CACHE CONTROL

Only the MC68EC030 cache control circuitry can directly access the cache arrays, but the supervisor program can set bits in the CACR to exercise control over cache operations. The supervisor also has access to the cache address register (CAAR), which contains the address for a cache entry to be cleared.

6.3.1 Cache Control Register

The CACR, shown in Figure 6-14, is a 32-bit register that can be written or read by the MOVEC instruction or indirectly modified by a reset. Five of the bits (4–0) control the instruction cache; six other bits (13–8) control the data cache. Each cache is controlled independently of the other, although a similar operation can be performed for both caches by a single MOVEC instruction. For example, loading a long word in which bits 3 and 11 are set into the CACR clears both caches. Bits 31–14 and 7–5 are reserved for Motorola definition. They are currently read as zeros and are ignored when written. For future compatibility, writes should not set these bits.



WA = Write Allocate
DBE = Data Burst Enable
CD = Clear Data Cache
CED = Clear Entry in Data Cache
FD = Freeze Data Cache
ED = Enable Data Cache
IBE = Instruction Burst Enable
CI = Clear Instruction Cache
CEI = Clear Entry in Instruction Cache
FI = Freeze Instruction Cache
EI = Enable Instruction Cache

Figure 6-14. Cache Control Register

6.3.1.1 WRITE ALLOCATE. Bit 13, the WA bit, is set to select the write-allocation mode (refer to **6.1.2.1 WRITE ALLOCATION**) for write cycles. Clearing this bit selects the no-write-allocation mode. A reset operation clears this bit. The supervisor should set this bit when it shares data with the user task. If the data cache is disabled or frozen, the WA bit is ignored.

6.3.1.2 DATA BURST ENABLE. Bit 12, the DBE bit, is set to enable burst filling of the data cache. Operating systems and other software set this bit when burst filling of the data cache is desired. A reset operation clears the DBE bit.

6.3.1.3 CLEAR DATA CACHE. Bit 11, the CD bit, is set to clear all entries in the data cache. Operating systems and other software set this bit to clear data from the cache prior to a context switch. The controller clears all valid bits in the data cache at the time a MOVEC instruction loads a one into the CD bit of the CACR. The CD bit is always read as a zero.

6.3.1.4 CLEAR ENTRY IN DATA CACHE. Bit 10, the CED bit, is set to clear an entry in the data cache. The index field of the CAAR (see Figure 6-15) corresponding to the index and long-word select portion of an address specifies the entry to be cleared. The controller clears only the specified long word by clearing the valid bit for the entry at the time a MOVEC instruction loads a one into the CED bit of the CACR, regardless of the states of the ED and FD bits. The CED bit is always read as a zero.

6.3.1.5 FREEZE DATA CACHE. Bit 9, the FD bit, is set to freeze the data cache. When the FD bit is set and a miss occurs during a read or write of the data cache, the indexed entry is not replaced. However, write cycles that hit in the data cache cause the entry to be updated even when the cache is frozen. When the FD bit is clear, a miss in the data cache during a read cycle causes the entry (or line) to be filled, and the filling of entries on writes that miss are then controlled by the WA bit. A reset operation clears the FD bit.

6.3.1.6 ENABLE DATA CACHE. Bit 8, the ED bit, is set to enable the data cache. When it is cleared, the data cache is disabled. A reset operation clears the ED bit. The supervisor normally enables the data cache, but it can clear ED for system debugging or emulation, as required. Disabling the data cache does not flush the entries. If it is enabled again, the previously valid entries remain valid and can be used.

6.3.1.7 INSTRUCTION BURST ENABLE. Bit 4, the IBE bit, is set to enable burst filling of the instruction cache. Operating systems and other software set this bit when burst filling of the instruction cache is desired. A reset operation clears the IBE bit.

6.3.1.8 CLEAR INSTRUCTION CACHE. Bit 3, the CI bit, is set to clear all entries in the instruction cache. Operating systems and other software set this bit to clear instructions from the cache prior to a context switch. The controller clears all valid bits in the instruction cache at the time a MOVEC instruction loads a one into the CI bit of the CACR. The CI bit is always read as a zero.

6.3.1.9 CLEAR ENTRY IN INSTRUCTION CACHE. Bit 2, the CEI bit, is set to clear an entry in the instruction cache. The index field of the CAAR (see Figure 6-15) corresponding to the index and long-word select portion of an address specifies the entry to be cleared. The controller clears only the specified long word by clearing the valid bit for the entry at the time a MOVEC instruction loads a one into the CEI bit of the CACR, regardless of the states of the EI and FI bits. The CEI bit is always read as a zero.

6.3.1.10 FREEZE INSTRUCTION CACHE. Bit 1, the FI bit, is set to freeze the instruction cache. When the FI bit is set and a miss occurs in the instruction cache, the entry (or line) is not replaced. When the FI bit is cleared to zero, a miss in the instruction cache causes the entry (or line) to be filled. A reset operation clears the FI bit.

6.3.1.11 ENABLE INSTRUCTION CACHE. Bit 0, the EI bit, is set to enable the instruction cache. When it is cleared, the instruction cache is disabled. A reset operation clears the EI bit. The supervisor normally enables the instruction cache, but it can clear EI for system debugging or emulation, as required. Disabling the instruction cache does not flush the entries. If it is enabled again, the previously valid entries remain valid and may be used.

6.3.2 Cache Address Register

The CAAR is a 32-bit register shown in Figure 6-15. The index field (bits 7–2) contains the address for the "clear cache entry" operations. The bits of this field correspond to bits 7–2 of addresses; they specify the index and a long word of a cache line. Although only the index field is used currently, all 32 bits of the register are implemented and are reserved for use by Motorola.

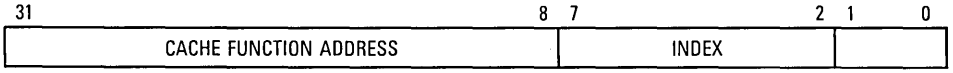


Figure 6-15. Cache Address Register

SECTION 7
BUS OPERATION

This section provides a functional description of the bus, the signals that control it, and the bus cycles provided for data transfer operations. It also describes the error and halt conditions, bus arbitration, and the reset operation. Operation of the bus is the same whether the controller or an external device is the bus master; the names and descriptions of bus cycles are from the point of view of the bus master. For exact timing specifications, refer to MC68EC030/D, *MC68EC030 Technical Summary*.

The MC68EC030 architecture supports byte, word, and long-word operands, allowing access to 8-, 16-, and 32-bit data ports through the use of asynchronous cycles controlled by the data transfer and size acknowledge inputs ($\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$).

Synchronous bus cycles controlled by the synchronous termination signal ($\overline{\text{STERM}}$) can only be used to transfer data to and from 32-bit ports.

The MC68EC030 allows byte, word, and long-word operands to be located in memory on any byte boundary. For a misaligned transfer, more than one bus cycle may be required to complete the transfer, regardless of port size. For a port less than 32 bits wide, multiple bus cycles may be required for an operand transfer due to either misalignment or a port width smaller than the operand size. Instruction words and their associated extension words must be aligned on word boundaries. The user should be aware that misalignment of word or long-word operands can cause the MC68EC030 to perform multiple bus cycles for the operand transfer; therefore, controller performance is optimized if word and long-word memory operands are aligned on word or long-word boundaries, respectively.

7.1 BUS TRANSFER SIGNALS

The bus transfers information between the MC68EC030 and an external memory, coprocessor, or peripheral device. External devices can accept or provide 8 bits, 16 bits, or 32 bits in parallel and must follow the handshake protocol described in this section. The maximum number of bits accepted or provided during a bus transfer is defined as the port width. The MC68EC030 contains an address bus that specifies the address for the transfer and a data bus that

transfers the data. Control signals indicate the beginning of the cycle, the address space and the size of the transfer, and the type of cycle. The selected device then controls the length of the cycle with the signal(s) used to terminate the cycle. Strobe signals, one for the address bus and another for the data bus, indicate the validity of the address and provide timing information for the data.

The bus can operate in an asynchronous mode identical to the MC68030 and MC68020 bus for any port width. The bus and control input signals used for asynchronous operation are internally synchronized to the MC68EC030 clock, introducing a delay. This delay is the time period required for the MC68EC030 to sample an asynchronous input signal, synchronize the input to the internal clocks of the controller, and determine whether it is high or low. Figure 7-1 shows the relationship between the clock signal and the associated internal signal of a typical asynchronous input.

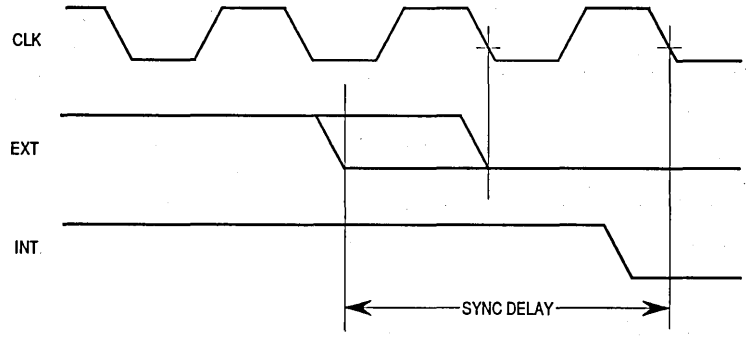


Figure 7-1. Relationship between External and Internal Signals

Furthermore, for all asynchronous inputs, the controller latches the level of the input during a sample window around the falling edge of the clock signal. This window is illustrated in Figure 7-2. To ensure that an input signal is recognized on a specific falling edge of the clock, that input must be stable during the sample window. If an input makes a transition during the window time period, the level recognized by the controller is not predictable; however, the controller always resolves the latched level to either a logic high or low before using it. In addition to meeting input setup and hold times for deterministic operation, all input signals must obey the protocols described in this section.

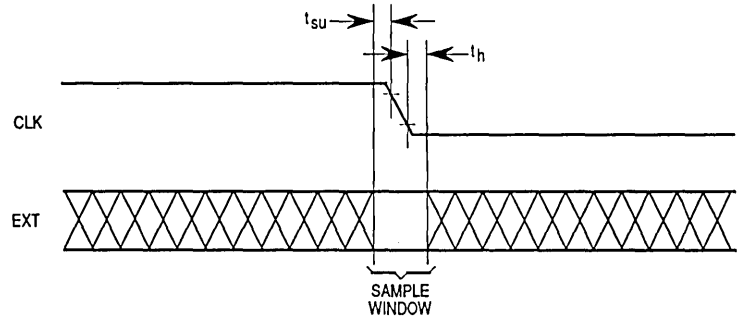


Figure 7-2. Asynchronous Input Sample Window

A device with a 32-bit port size can also provide a synchronous mode transfer. In synchronous operation, input signals are externally synchronized to the controller clock, and the synchronizing delay is not incurred.

Synchronous inputs ($\overline{\text{STERM}}$, $\overline{\text{CBACK}}$, and $\overline{\text{CIIN}}$) must remain stable during a sample window for all rising edges of the clock during a bus cycle (i.e., while address strobe ($\overline{\text{AS}}$) is asserted), regardless of when the signals are asserted or negated, to ensure proper operation. This sample window is defined by the synchronous input setup and hold times (see MC68EC030/D, *MC68EC030 Technical Summary*).

7.1.1 Bus Control Signals

The external cycle start ($\overline{\text{ECS}}$) signal is the earliest indication that the controller is initiating a bus cycle. The MC68EC030 initiates a bus cycle by driving the address, size, function code, read/write, and cache inhibit-out outputs and by asserting $\overline{\text{ECS}}$. However, if the controller finds the required program or data item in an on-chip cache, or if the ACU finds a fault with the access, the controller aborts the cycle before asserting $\overline{\text{AS}}$. $\overline{\text{ECS}}$ can be used to initiate various timing sequences that are eventually qualified with $\overline{\text{AS}}$. Qualification with $\overline{\text{AS}}$ may be required since, in the case of an internal cache hit, or an ACU fault, a bus cycle may be aborted after $\overline{\text{ECS}}$ has been asserted. The assertion of $\overline{\text{AS}}$ ensures that the cycle has not been aborted by these internal conditions.

During the first external bus cycle of an operand transfer, the operand cycle start ($\overline{\text{OCS}}$) signal is asserted with $\overline{\text{ECS}}$. When several bus cycles are required

to transfer the entire operand, $\overline{OCS}$ is asserted only at the beginning of the first external bus cycle. With respect to $\overline{OCS}$, an "operand" is any entity required by the execution unit, whether a program or data item.

The function code signals (FC0–FC2) are also driven at the beginning of a bus cycle. These three signals select one of eight address spaces (refer to Table 4-1) to which the address applies. Five address spaces are presently defined. Of the remaining three, one is reserved for user definition and two are reserved by Motorola for future use. The function code signals are valid while $\overline{AS}$ is asserted.

At the beginning of a bus cycle, the size signals (SIZ0 and SIZ1) are driven along with $\overline{ECS}$ and the FC0–FC2. SIZ0 and SIZ1 indicate the number of bytes remaining to be transferred during an operand cycle (consisting of one or more bus cycles) or during a cache fill operation from a device with a port size that is less than 32 bits. Table 7-2 shows the encoding of SIZ0 and SIZ1. These signals are valid while $\overline{AS}$ is asserted.

The read/write (R/$\overline{W}$) signal determines the direction of the transfer during a bus cycle. This signal changes state, when required, at the beginning of a bus cycle and is valid while $\overline{AS}$ is asserted. R/$\overline{W}$ only transitions when a write cycle is preceded by a read cycle or vice versa. The signal may remain low for two consecutive write cycles.

The read-modify-write cycle signal ($\overline{RMC}$) is asserted at the beginning of the first bus cycle of a read-modify-write operation and remains asserted until completion of the final bus cycle of the operation. The $\overline{RMC}$ signal is guaranteed to be negated before the end of state 0 for a bus cycle following a read-modify-write operation.

7.1.2 Address Bus

The address bus signals (A0–A31) define the address of the byte (or the most significant byte) to be transferred during a bus cycle. The controller places the address on the bus at the beginning of a bus cycle. The address is valid while $\overline{AS}$ is asserted.

7.1.3 Address Strobe

$\overline{AS}$ is a timing signal that indicates the validity of an address on the address bus and of many control signals. It is asserted one-half clock after the beginning of a bus cycle.

7.1.4 Data Bus

The data bus signals (D0–D31) comprise a bidirectional, nonmultiplexed parallel bus that contains the data being transferred to or from the controller. A read or write operation may transfer 8, 16, 24, or 32 bits of data (one, two, three, or four bytes) in one bus cycle. During a read cycle, the data is latched by the controller on the last falling edge of the clock for that bus cycle. For a write cycle, all 32 bits of the data bus are driven, regardless of the port width or operand size. The controller places the data on the data bus one-half clock cycle after $\overline{AS}$ is asserted in a write cycle.

7.1.5 Data Strobe

The data strobe ($\overline{DS}$) is a timing signal that applies to the data bus. For a read cycle, the controller asserts $\overline{DS}$ to signal the external device to place data on the bus. It is asserted at the same time as $\overline{AS}$ during a read cycle. For a write cycle, $\overline{DS}$ signals to the external device that the data to be written is valid on the bus. The controller asserts $\overline{DS}$ one full clock cycle after the assertion of $\overline{AS}$ during a write cycle. $\overline{DS}$ is most useful to asynchronous systems which can not use the synchronous set-up and hold times to clock edges.

7.1.6 Data Buffer Enable

The data buffer enable signal ($\overline{DBEN}$) can be used to enable external data buffers while data is present on the data bus. During a read operation, $\overline{DBEN}$ is asserted one clock cycle after the beginning of the bus cycle and is negated as $\overline{DS}$ is negated. In a write operation, $\overline{DBEN}$ is asserted at the time $\overline{AS}$ is asserted and is held active for the duration of the cycle. In a synchronous system supporting two-clock bus cycles, $\overline{DBEN}$ timing may prevent its use.

7.1.7 Bus Cycle Termination Signals

During asynchronous bus cycles, external devices assert the data transfer and size acknowledge signals ($\overline{DSACK0}$ and/or $\overline{DSACK1}$) as part of the bus protocol. During a read cycle, the assertion of $\overline{DSACKx}$ signals the controller to terminate the bus cycle and to latch the data. During a write cycle, the assertion of $\overline{DSACKx}$ indicates that the external device has successfully stored the data and that the cycle may terminate. These signals also indicate to the controller the size of the port for the bus cycle just completed, as shown in Table 7-1. Refer to **7.3.1 Asynchronous Read Cycle** for timing relationships of $\overline{DSACK0}$ and $\overline{DSACK1}$.

For synchronous bus cycles, external devices assert the synchronous termination signal (STERM) as part of the bus protocol. During a read cycle, the assertion of STERM causes the controller to latch the data. During a write cycle, it indicates that the external device has successfully stored the data. In either case, it terminates the cycle and indicates that the transfer was made to a 32-bit port. Refer to **7.3.2 Asynchronous Write Cycle** for timing relationships of STERM.

The bus error (BERR) signal is also a bus cycle termination indicator and can be used in the absence of DSACKx or STERM to indicate a bus error condition. It can also be asserted in conjunction with DSACKx or STERM to indicate a bus error condition, provided it meets the appropriate timing described in this section and in MC68EC030/D, *MC68EC030 Technical Summary*. Additionally, the BERR and HALT signals can be asserted together to indicate a retry termination. Again, the BERR and HALT signals can be asserted simultaneously in lieu of or in conjunction with the DSACKx or STERM signals.

Finally, the autovector (AVEC) signal can be used to terminate interrupt acknowledge cycles, indicating that the MC68EC030 should internally generate a vector number to locate an interrupt handler routine. AVEC is ignored during all other bus cycles.

7

7.2 DATA TRANSFER MECHANISM

The MC68EC030 architecture supports byte, word, and long-word operands allowing access to 8-, 16-, and 32-bit data ports through the use of asynchronous cycles controlled by DSACK0 and DSACK1. It also supports synchronous bus cycles to and from 32-bit ports, terminated by STERM. Byte, word, and long-word operands can be located on any byte boundary, but misaligned transfers may require additional bus cycles, regardless of port size.

When the controller requests a burst mode fill operation, it asserts the cache burst request (CBREQ) signal to attempt to fill four entries within a line in one of the on-chip caches. This mode is compatible with nibble, static column, or page mode dynamic RAMs. The burst fill operation uses synchronous bus cycles, each terminated by STERM, to fetch as many as four long words.

7.2.1 Dynamic Bus Sizing

The MC68EC030 dynamically interprets the port size of the addressed device during each bus cycle, allowing operand transfers to or from 8-, 16-, and 32-bit ports. During an asynchronous operand transfer cycle, the slave device signals its port size (byte, word, or long word) and indicates completion of the bus cycle to the controller through the use of the $\overline{\text{DSACKx}}$ inputs. Refer to Table 7-1 for $\overline{\text{DSACKx}}$ encodings and assertion results.

Table 7-1. $\overline{\text{DSACK}}$ Codes and Results

$\overline{\text{DSACK1}}$	$\overline{\text{DSACK0}}$	Result
H	H	Insert Wait States in Current Bus Cycle
H	L	Complete Cycle — Data Bus Port Size is 8 Bits
L	H	Complete Cycle — Data Bus Port Size is 16 Bits
L	L	Complete Cycle — Data Bus Port Size is 32 Bits

For example, if the controller is executing an instruction that reads a long-word operand from a long-word aligned address, it attempts to read 32 bits during the first bus cycle. (Refer to **7.2.2 Misaligned Operands** for the case of a word or byte address.) If the port responds that it is 32 bits wide, the MC68EC030 latches all 32 bits of data and continues with the next operation. If the port responds that it is 16 bits wide, the MC68EC030 latches the 16 bits of valid data and runs another bus cycle to obtain the other 16 bits. The operation for an 8-bit port is similar, but requires four read cycles. The addressed device uses the $\overline{\text{DSACKx}}$ signals to indicate the port width. For instance, a 32-bit device *always* returns $\overline{\text{DSACKx}}$ for a 32-bit port (regardless of whether the bus cycle is a byte, word, or long-word operation).

Dynamic bus sizing requires that the portion of the data bus used for a transfer to or from a particular port size be fixed. A 32-bit port must reside on data bus bits 0–31, a 16-bit port must reside on data bus bits 16–32, and an 8-bit port must reside on data bus bits 24–31. This requirement minimizes the number of bus cycles needed to transfer data to 8- and 16-bit ports and ensures that the MC68EC030 correctly transfers valid data. The MC68EC030 always attempts to transfer the maximum amount of data on all bus cycles; for a long-word operation, it always assumes that the port is 32 bit wide when beginning the bus cycle.

The bytes of operands are designated as shown in Figure 7-3. The most significant byte of a long-word operand is OP0, and OP3 is the least significant byte. The two bytes of a word-length operand are OP2 (most significant) and OP3. The single byte of a byte-length operand is OP3. These designations are used in the figures and descriptions that follow.

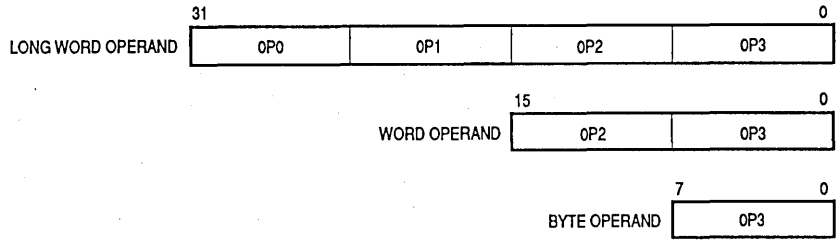


Figure 7-3. Internal Operand Representation

Figure 7-4 shows the required organization of data ports on the MC68EC030 bus for 8-, 16-, and 32-bit devices. The four bytes shown in Figure 7-4 are connected through the internal data bus and data multiplexer to the external data bus. This path is the means through which the MC68EC030 supports dynamic bus sizing and operand misalignment. Refer to **7.2.2 Misaligned Operands** for the definition of misaligned operand. The data multiplexer establishes the necessary connections for different combinations of address and data sizes.

The multiplexer takes the four bytes of the 32-bit bus and routes them to their required positions. For example, OP0 can be routed to D24–D31, as would be the normal case, or it can be routed to any other byte position to support a misaligned transfer. The same is true for any of the operand bytes. The positioning of bytes is determined by the size (SIZ0 and SIZ1) and address (A0 and A1) outputs.

The SIZ0 and SIZ1 outputs indicate the remaining number of bytes to be transferred during the current bus cycle, as shown in Table 7-2.

The number of bytes transferred during a write or noncacheable read bus cycle is equal to or less than the size indicated by the SIZ0 and SIZ1 outputs, depending on port width and operand alignment. For example, during the first bus cycle of a long-word transfer to a word port, the size outputs indicate that four bytes are to be transferred, although only two bytes are moved on that bus cycle. Cachable read cycles must always transfer the number of bytes indicated by the port size.

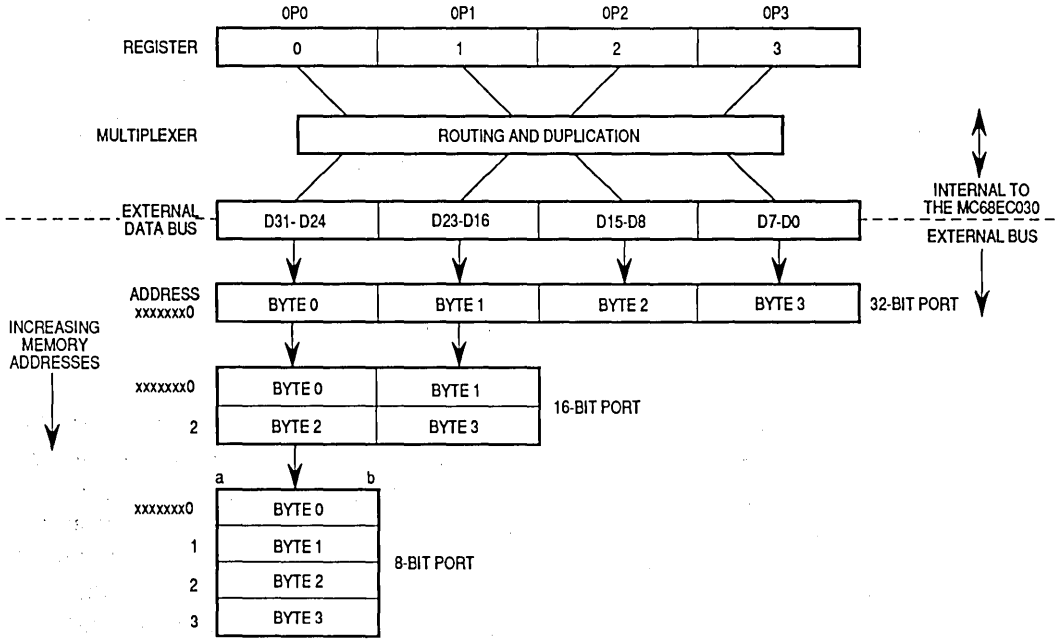


Figure 7-4. MC68EC030 Interface to Various Port Sizes

A0 and A1 also affect operation of the data multiplexer. During an operand transfer, A2–A31 indicate the long-word base address of that portion of the operand to be accessed; A0 and A1 indicate the byte offset from the base. Table 7-3 shows the encodings of A0 and A1 and the corresponding byte offsets from the long-word base.

Table 7-4 lists the bytes required on the data bus for read cycles that are cacheable. The entries shown as OPn are portions of the requested operand that are read or written during that bus cycle and are defined by SIZ0, SIZ1, A0, and A1 for the bus cycle. The PRn and the Nn bytes correspond to the previous and next bytes in memory, respectively, that must be valid on the data bus for the specified port size (long word or word) so that the internal caches operate correctly. (For cacheable accesses, the MC68EC030 assumes that all portions of the data bus for a given port size are valid.) This same table applies to noncacheable read cycles except that the bytes labeled PRn and Nn are not required and can be replaced by "don't cares".

Table 7-2. Size Signal Encoding

SIZ1	SIZ0	Size
0	1	Byte
1	0	Word
1	1	3 Bytes
0	0	Long Word

Table 7-3. Address Offset Encodings

A1	A0	Offset
0	0	+0 Bytes
0	1	+1 Byte
1	0	+2 Bytes
1	1	+3 Bytes

Table 7-4. Data Bus Requirements for Read Cycles

Transfer Size	Size		Address		Long-Word Port External Data Bytes Required				Word Port External Data Bytes Required		Byte Port External Data Bytes Required
	SIZ1	SIZ0	A1	A0	D31:D24	D23:D16	D15:D8	D7:D0	D31:D24	D23:D16	D31:D24
Byte	0	1	0	0	OP3	N	N1	N2	OP3	N	OP3
	0	1	0	1	PR	OP3	N	N1	PR	OP3	OP3
	0	1	1	0	PR1	PR	OP3	N	OP3	N	OP3
	0	1	1	1	PR2	PR1	PR	OP3	PR	OP3	OP3
Word	1	0	0	0	OP2	OP3	N	N1	OP2	OP3	OP2
	1	0	0	1	PR	OP2	OP3	N	PR	OP2	OP2
	1	0	1	0	PR1	PR	OP2	OP3	OP2	OP3	OP2
	1	0	1	1	PR2	PR1	PR	OP2	PR	OP2	OP2
3 Byte	1	1	0	0	OP1	OP2	OP3	N	OP1	OP2	OP1
	1	1	0	1	PR	OP1	OP2	OP3	PR	OP1	OP1
	1	1	1	0	PR1	PR	OP1	OP2	OP1	OP2	OP1
	1	1	1	1	PR2	PR1	PR	OP1	PR	OP1	OP1
Long Word	0	0	0	0	OP0	OP1	OP2	OP3	OP0	OP1	OP0
	0	0	0	1	PR	OP0	OP1	OP2	PR	OP0	OP0
	0	0	1	0	PR1	PR	OP0	OP1	OP0	OP1	OP0
	0	0	1	1	PR2	PR1	PR	OP0	PR	OP0	OP0

NOTE: The bytes labeled as Nn (Next n) and PRn (Previous n) are only required to be valid for cacheable read cycles. They can be interpreted as don't cares for noncacheable read cycles.

Table 7-5 lists the combinations of SIZ0, SIZ1, A0, and A1 and the corresponding pattern of the data transfer for write cycles from the internal multiplexer of the MC68EC030 to the external data bus.

Table 7-5. MC68EC030 Internal to External Data Bus Multiplexer — Write Cycles

Transfer Size	Size		Address		External Data Bus Connection			
	SIZ1	SIZ0	A1	A0	D31:D24	D23:D16	D15:D8	D7:D0
Byte	0	1	x	x	OP3	OP3	OP3	OP3
Word	1	0	x	0	OP2	OP3	OP2	OP3
	1	0	x	1	OP2	OP2	OP3	OP2
3 Byte	1	1	0	0	OP1	OP2	OP3	OP0*
	1	1	0	1	OP1	OP1	OP2	OP3
	1	1	1	0	OP1	OP2	OP1	OP2
	1	1	1	1	OP1	OP1	OP2*	OP1
Long Word	0	0	0	0	OP0	OP1	OP2	OP3
	0	0	0	1	OP0	OP0	OP1	OP2
	0	0	1	0	OP0	OP1	OP0	OP1
	0	0	1	1	OP0	OP0	OP1*	OP0

*Due to the current implementation, this byte is output but never used.
x = don't care
NOTE: The OP tables on the external data bus refer to a particular byte of the operand that is written on that section of the data bus.

Figure 7-5 shows the transfer of a long-word operand to a word port. In the first bus cycle, the MC68EC030 places the four operand bytes on the external bus. Since the address is long-word aligned in this example, the multiplexer follows the pattern in the entry of Table 7-5 corresponding to SIZ0_SIZ1_A0_A1 = 0000. The port latches the data on bits D16–D31 of the data bus, asserts $\overline{\text{DSACK1}}$ ($\overline{\text{DSACK0}}$ remains negated), and the controller terminates the bus cycle. It then starts a new bus cycle with SIZ0_SIZ1_A0_A1 = 1010 to transfer the remaining 16 bits. SIZ0 and SIZ1 indicate that a word remains to be transferred; A0 and A1 indicate that the word corresponds to an offset of two from the base address. The multiplexer follows the pattern corresponding to this configuration of the size and address signals and places the two least significant bytes of the long word on the word portion of the bus (D16–D31). The bus cycle transfers the remaining bytes to the word-size port. Figure 7-6 shows the timing of the bus transfer signals for this operation.

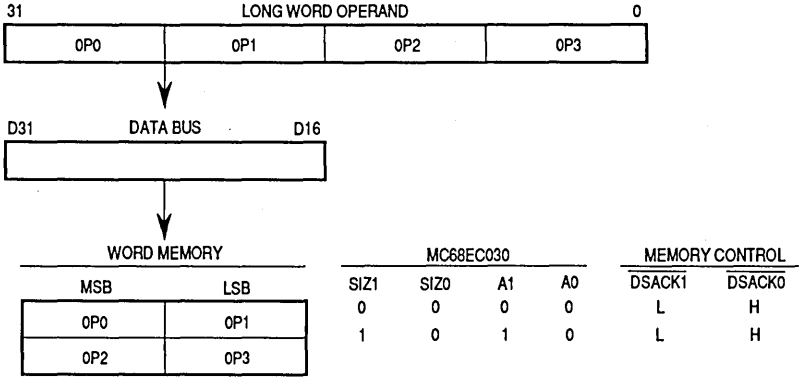


Figure 7-5. Example of Long-Word Transfer to Word Port

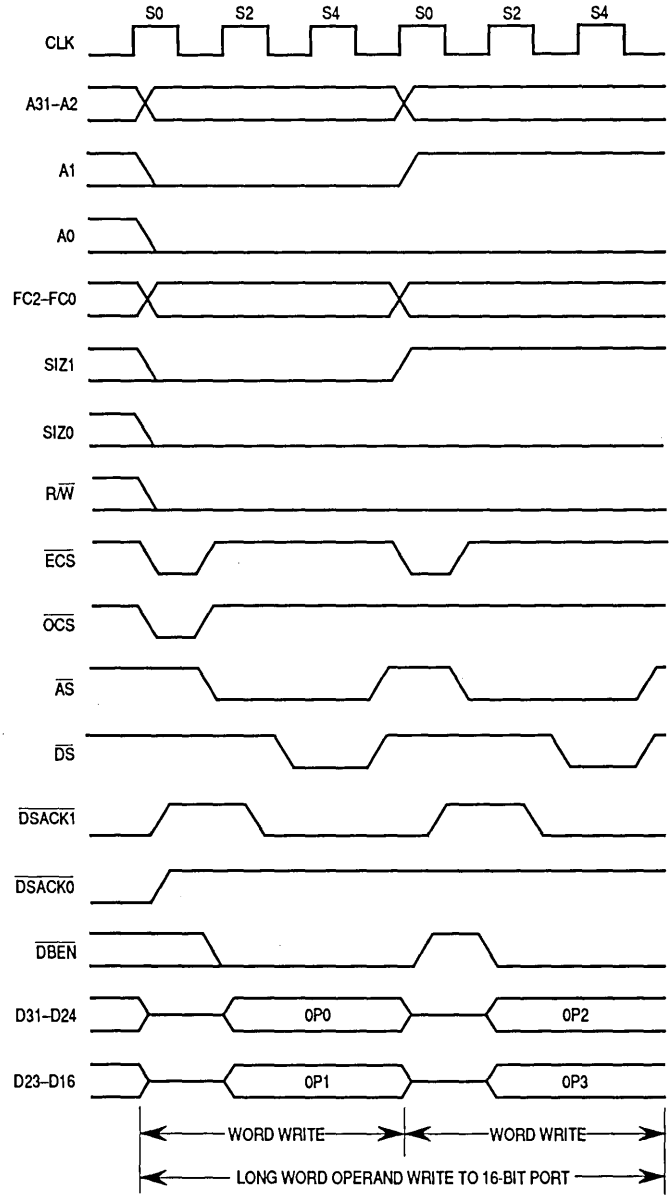


Figure 7-6. Long-Word Operand Write Timing (16-Bit Data Port)

Figure 7-7 shows a word transfer to an 8-bit bus port. Like the preceding example, this example requires two bus cycles. Each bus cycle transfers a single byte. The size signals for the first cycle specify two bytes; for the second cycle, one byte. Figure 7-8 shows the associated bus transfer signal timing.

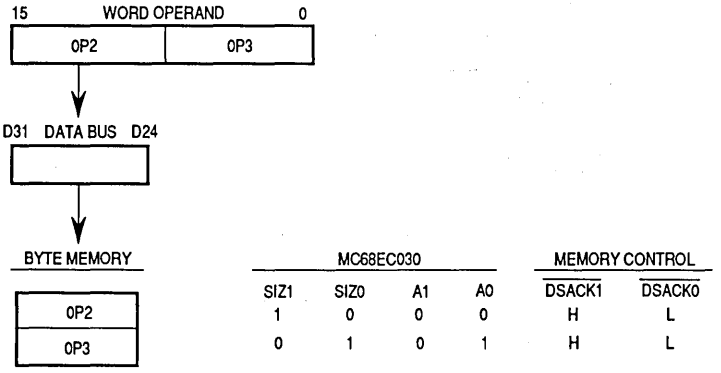


Figure 7-7. Example of Word Transfer to Byte Port

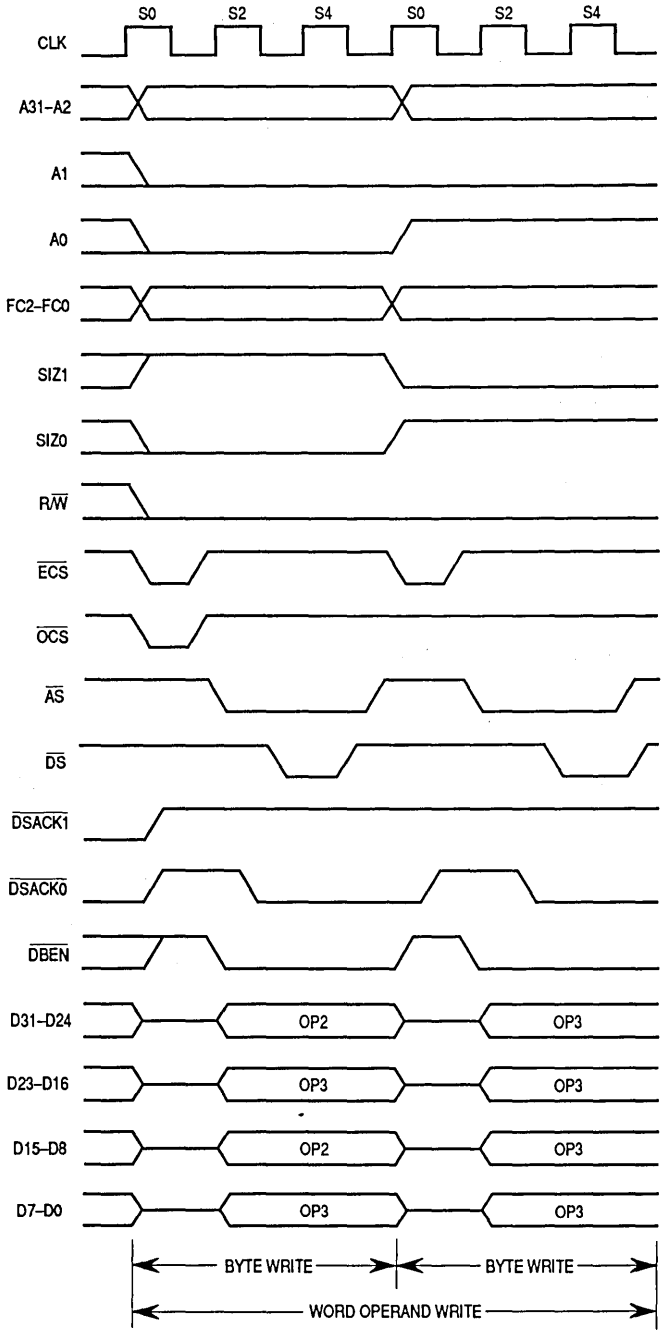


Figure 7-8. Word Operand Write Timing (8-Bit Data Port)

7.2.2 Misaligned Operands

Since operands may reside at any byte boundaries, they may be misaligned. A byte operand is properly aligned at any address; a word operand is misaligned at an odd address; a long word is misaligned at an address that is not evenly divisible by four. The MC68000, MC68008, and MC68010 implementations allow long-word transfers on odd-word boundaries but force exceptions if word or long-word operand transfers are attempted at odd-byte addresses. Although the MC68EC030 does not enforce any alignment restrictions for data operands (including PC relative data addresses), some performance degradation occurs when additional bus cycles are required for long-word or word operands that are misaligned. For maximum performance, data items should be aligned on their natural boundaries. All instruction words and extension words must reside on word boundaries. Attempting to prefetch an instruction word at an odd address causes an address error exception.

Figure 7-9 shows the transfer of a long-word operand to an odd address in word-organized memory, which requires three bus cycles. For the first cycle, the size signals specify a long-word transfer, and the address offset (A2:A0) is 001. Since the port width is 16 bits, only the first byte of the long word is transferred. The slave device latches the byte and acknowledges the data transfer, indicating that the port is 16 bits wide. When the controller starts the second cycle, the size signals specify that three bytes remain to be transferred with an address offset (A2:A0) of 010. The next two bytes are transferred during this cycle. The controller then initiates the third cycle, with the size signals indicating one byte remaining to be transferred. The address offset (A2:A0) is now 100; the port latches the final byte; and the operation is complete. Figure 7-10 shows the associated bus transfer signal timing.

Figure 7-11 shows the equivalent operation for a cacheable data read cycle.

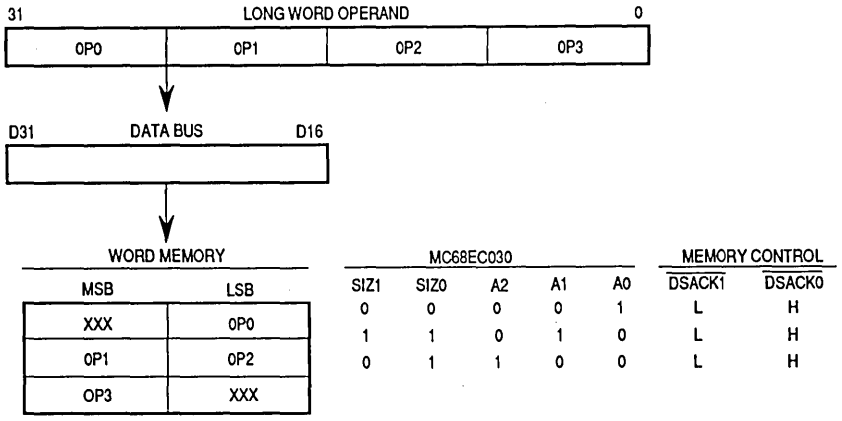


Figure 7-9. Misaligned Long-Word Transfer to Word Port Example

Figures 7-12 and 7-13 show a word transfer to an odd address in word-organized memory. This example is similar to the one shown in Figures 7-9 and 7-10 except that the operand is word sized and the transfer requires only two bus cycles.

Figure 7-14 shows the equivalent operation for a cacheable data read cycle.

Figures 7-15 and 7-16 show an example of a long-word transfer to an odd address in long-word-organized memory. In this example, a long-word access is attempted beginning at the least significant byte of a long-word-organized memory. Only one byte can be transferred in the first bus cycle. The second bus cycle then consists of a three-byte access to a long-word boundary. Since the memory is long-word organized, no further bus cycles are necessary.

Figure 7-17 shows the equivalent operation for a cacheable data read cycle.

7.2.3 Effects of Dynamic Bus Sizing and Operand Misalignment

The combination of operand size, operand alignment, and port size determines the number of bus cycles required to perform a particular memory access. Table 7-6 shows the number of bus cycles required for different operand sizes to different port sizes with all possible alignment conditions for write cycles and noncacheable read cycles.

Table 7-6. Memory Alignment and Port Size Influence on Write Bus Cycles

A1/A0	Number of Bus Cycles			
	00	01	10	11
Instruction*	1:2:4	N/A	N/A	N/A
Byte Operand	1:1:1	1:1:1	1:1:1	1:1:1
Word Operand	1:1:2	1:2:2	1:1:2	2:2:2
Long-Word Operand	1:2:4	2:3:4	2:2:4	2:3:4

Data Port Size — 32 Bits:16 Bits:8 Bits
*Instruction prefetches are always two words from a long-word boundary.

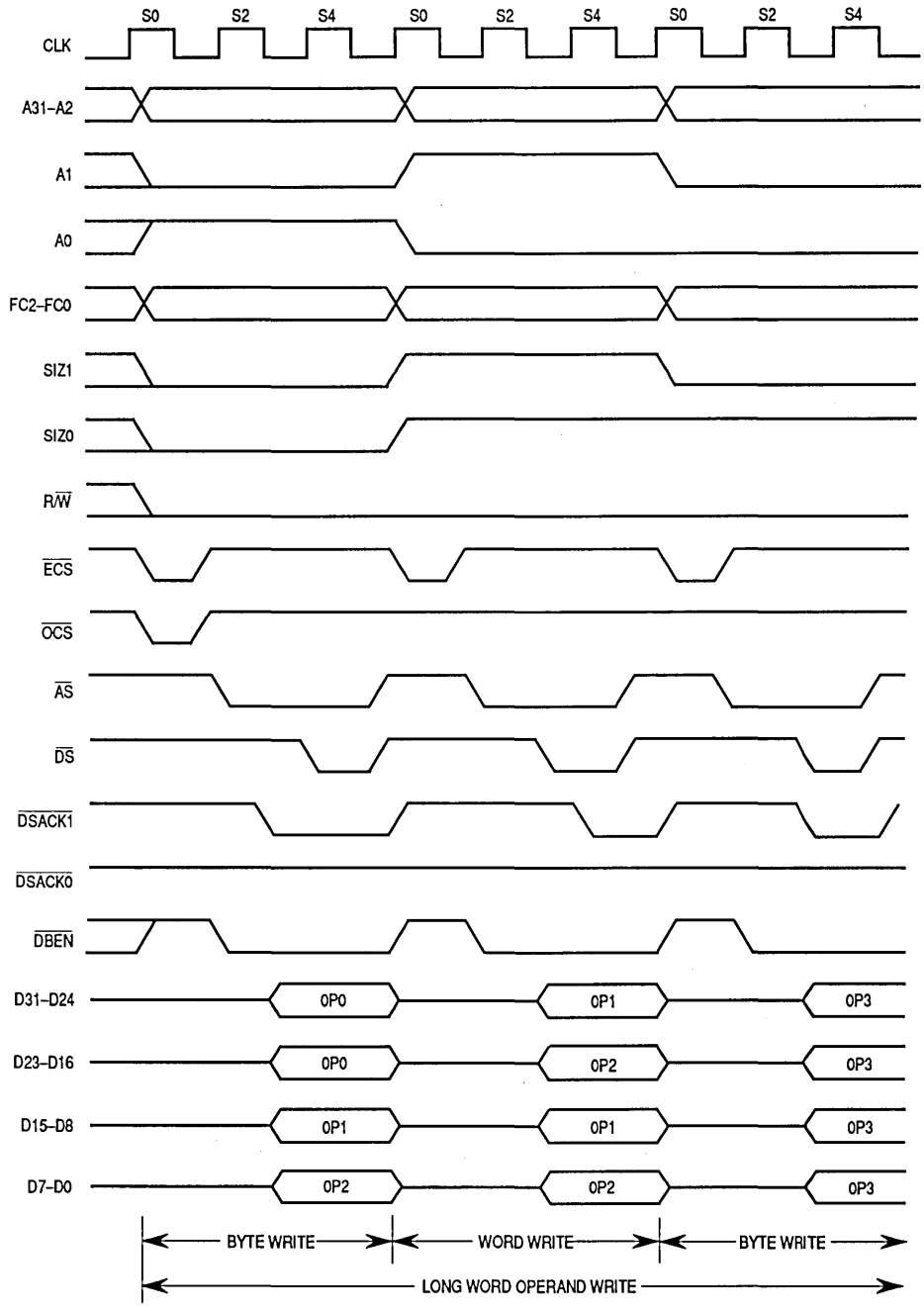
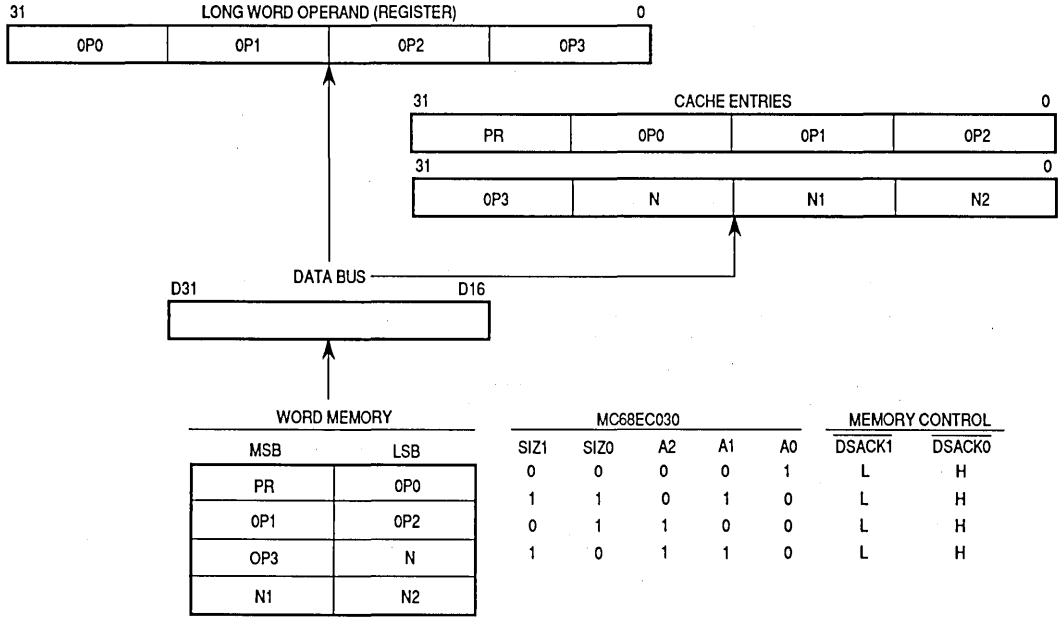


Figure 7-10. Misaligned Long-Word Transfer to Word Port



7

Figure 7-11. Misaligned Cacheable Long-Word Transfer from Word Port Example

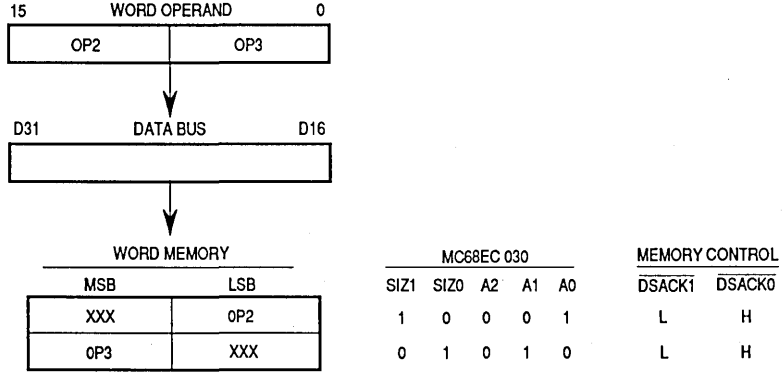


Figure 7-12. Misaligned Word Transfer to Word Port Example

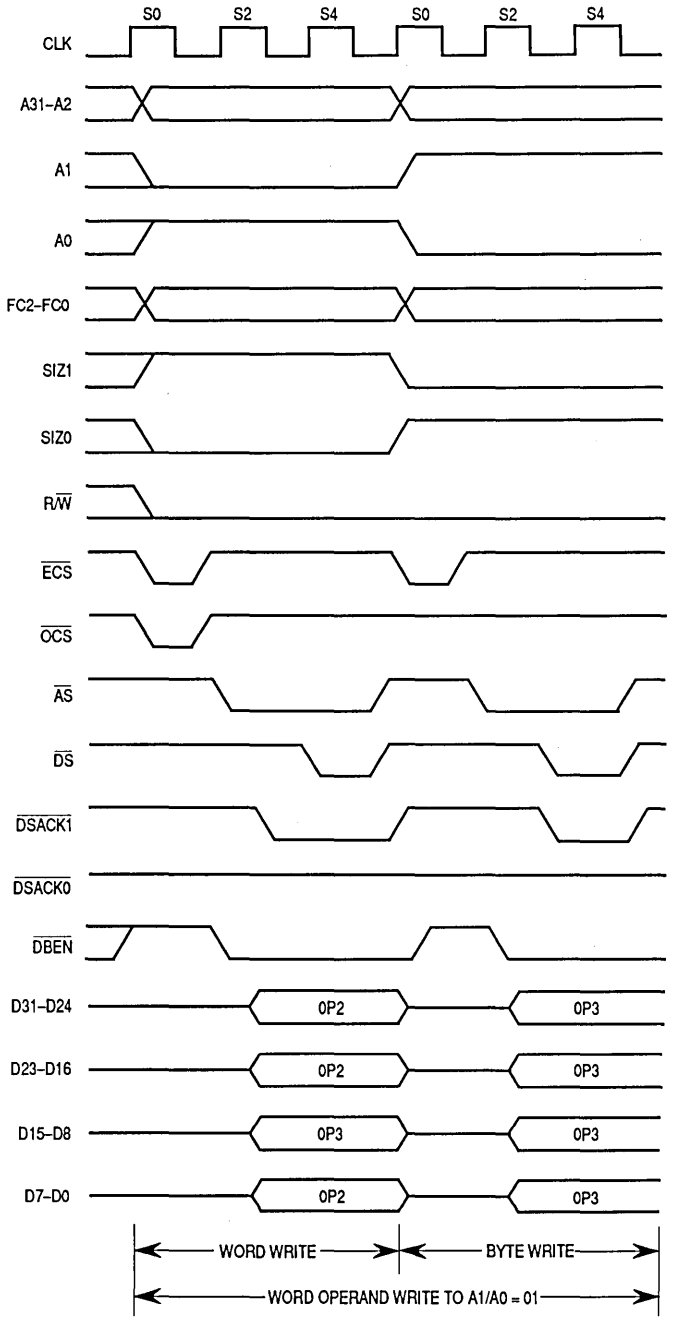


Figure 7-13. Misaligned Word Transfer to Word Port

This table shows that bus cycle throughput is significantly affected by port size and alignment. The MC68EC030 system designer and programmer should be aware of and account for these effects, particularly in time-critical applications.

Table 7-6 shows that the controller always prefetches instructions by reading a long word from a long-word address (A1:A0 = 00), regardless of port size or alignment. When the required instruction begins at an odd-word boundary, the controller attempts to fetch the entire 32 bits and loads both words into the instruction cache, if possible, although the second one is the required word. Even if the instruction access is not cached, the entire 32 bits are latched into an internal cache holding register from which the two instructions words can subsequently be referenced. Refer to **SECTION 11 INSTRUCTION EXECUTION TIMING** for a complete description of the cache holding register and pipeline operation.

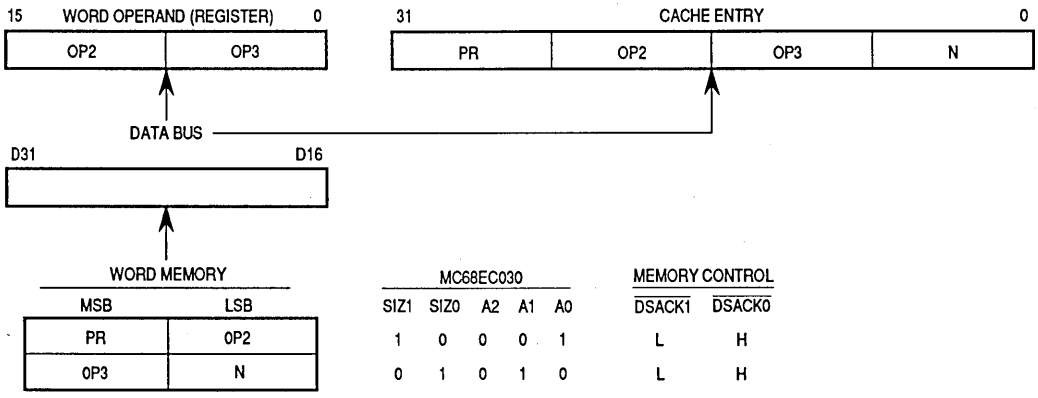


Figure 7-14. Example of Misaligned Cacheable Word Transfer from Word Bus

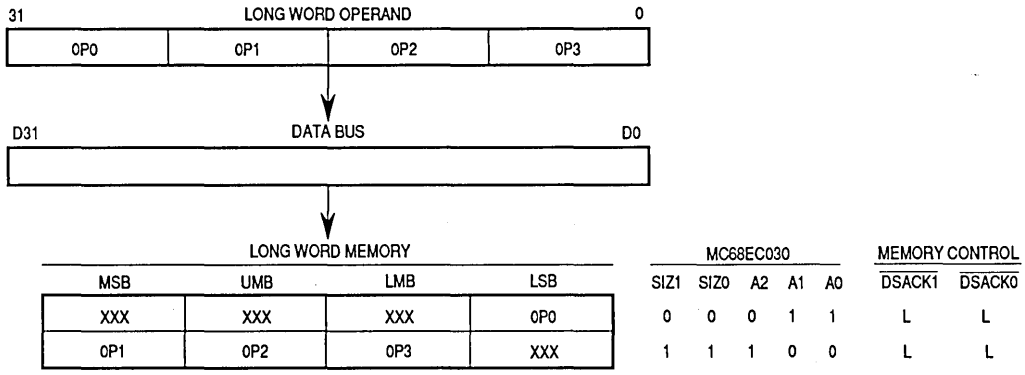


Figure 7-15. Misaligned Long-Word Transfer to Long-Word Port

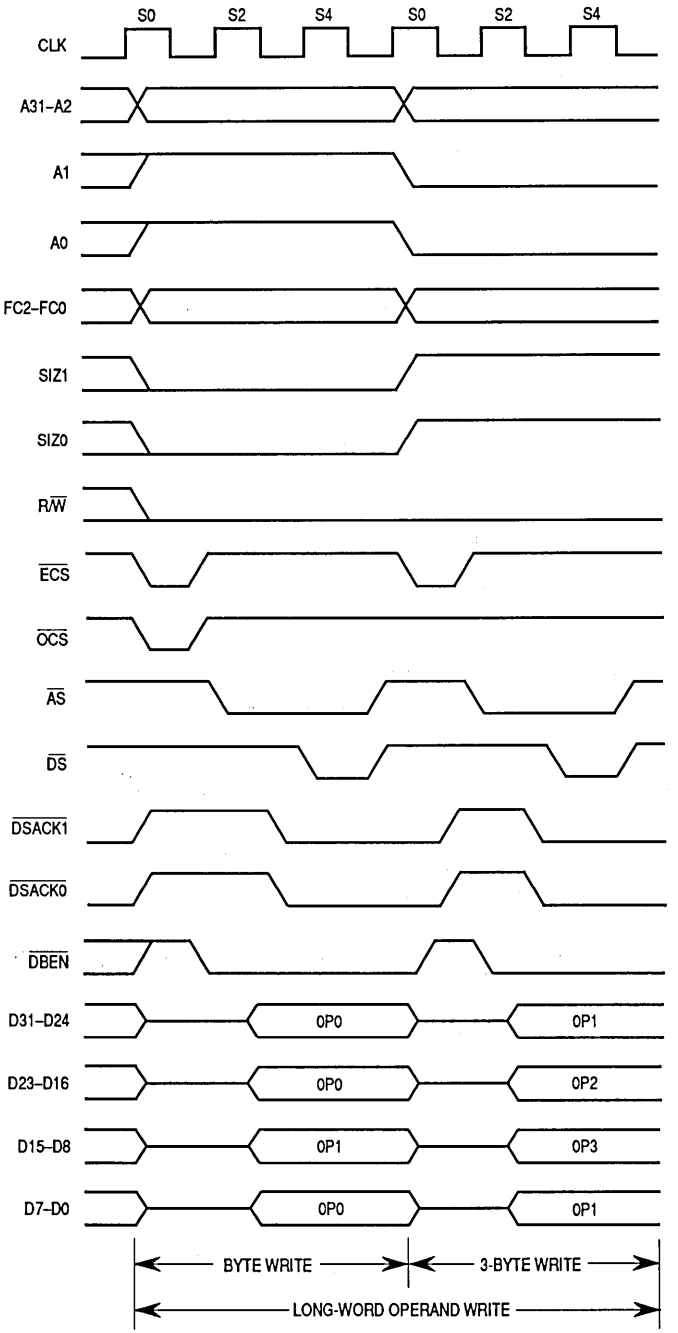


Figure 7-16. Misaligned Write Cycles to Long-Word Port

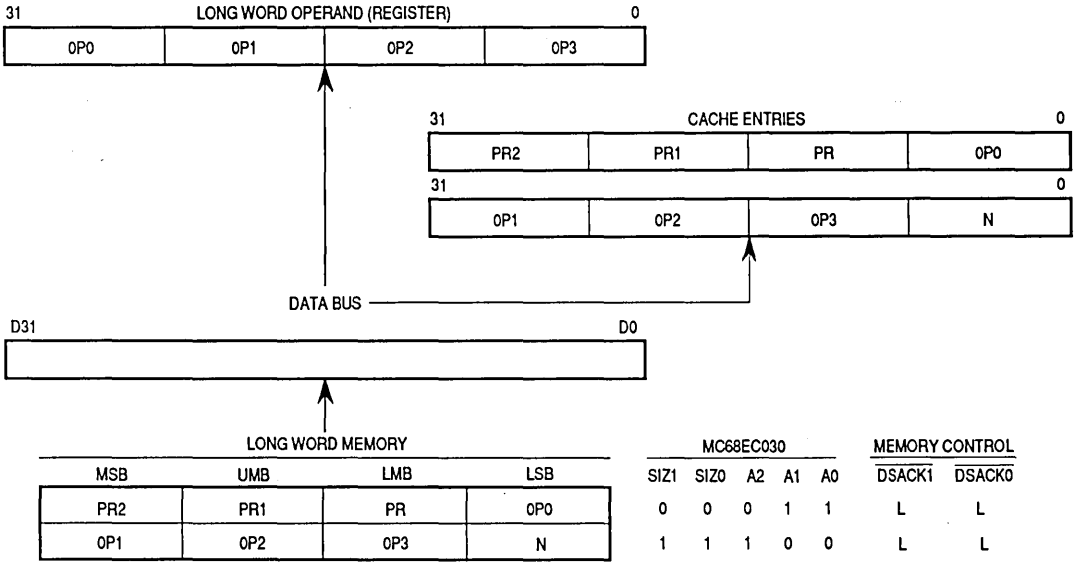


Figure 7-17. Misaligned Cacheable Long-Word Transfer from Long-Word Bus

7

7.2.4 Address, Size, and Data Bus Relationships

The data transfer examples show how the MC68EC030 drives data onto or receives data from the correct byte sections of the data bus. Table 7-7 shows the combinations of the size signals and address signals that are used to generate byte enable signals for each of the four sections of the data bus for noncacheable read cycles and all write cycles if the addressed device requires them. The port size also affects the generation of these enable signals as shown in the table. The four columns on the right correspond to the four byte enable signals. Letters B, W, and L refer to port sizes: B for 8-bit ports, W for 16-bit ports, and L for 32-bit ports. The letters B, W, and L imply that the byte enable signal should be true for that port size. A dash (—) implies that the byte enable signal does not apply.

Table 7-7. Data Bus Write Enable Signals for Byte, Word, and Long-Word Ports

Transfer Size	SIZ1	SIZ0	A1	A0	Data Bus Active Sections Byte (B) – Word (W) – Long-Word (L) Ports			
					D31:D24	D23:D16	D15:D8	D7:D0
Byte	0	1	0	0	B W L	—	—	—
	0	1	0	1	B	W L	—	—
	0	1	1	0	B W	—	L	—
	0	1	1	1	B	W	—	L
Word	1	0	0	0	B W L	W L	—	—
	1	0	0	1	B	W L	L	—
	1	0	1	0	B W	W	L	L
	1	0	1	1	B	W	—	L
3 Byte	1	1	0	0	B W L	W L	L	—
	1	1	0	1	B	W L	L	L
	1	1	1	0	B W	W	L	L
	1	1	1	1	B	W	—	L
Long Word	0	0	0	0	B W L	W L	L	L
	0	0	0	1	B	W L	L	L
	0	0	1	0	B W	W	L	L
	0	0	1	1	B	W	—	L

The MC68EC030 always drives all sections of the data bus because, at the start of a write cycle, the bus controller does not know the port size. The byte enable signals in the table apply only to read operations that are not to be internally cached and to write operations. For cacheable read cycles, during which the data is cached, the addressed port must drive all sections of the bus on which it resides.

The table shows that the MC68EC030 transfers the number of bytes specified by the size signals to or from the specified address unless the operand is misaligned or the number of bytes is greater than the port width. In these cases, the device transfers the greatest number of bytes possible for the port. For example, if the size is four bytes and the address offset (A1:A0) is 01, a 32-bit slave can only receive three bytes in the current bus cycle. A 16- or 8-bit slave can only receive one byte. The table defines the byte enables for all port sizes. Byte data strobes can be obtained by combining the enable signals with the data strobe signal. Devices residing on 8-bit ports can use the data strobe by itself since there is only one valid byte for every transfer. These enable or strobe signals select only the bytes required for write cycles or for noncacheable read cycles. The other bytes are not selected, which prevents incorrect accesses in sensitive areas such as I/O.

Figure 7-18 shows a logic diagram for one method for generating byte data enable signals for 16- and 32-bit ports from the size and address encodings and the read/write signal.

7.2.5 MC68EC030 versus MC68020 Dynamic Bus Sizing

The MC68EC030 supports the dynamic bus sizing mechanism of the MC68020 for asynchronous bus cycles (terminated with $\overline{\text{DSACKx}}$) with two restrictions. First, for a cacheable access within the boundaries of an aligned long word, the port size must be consistent throughout the transfer of each long word. For example, when a byte port resides at address $00, addresses $01, $02, and $03 must also correspond to byte ports. Second, the port must supply as much data as it signals as port size, regardless of the transfer size indicated with the size signals and the address offset indicated by A0 and A1 for cacheable accesses. Otherwise, dynamic bus sizing is identical in the MC68EC030 and MC68020. Dynamic bus sizing is identical in all respects in the MC68EC030 and MC68030.

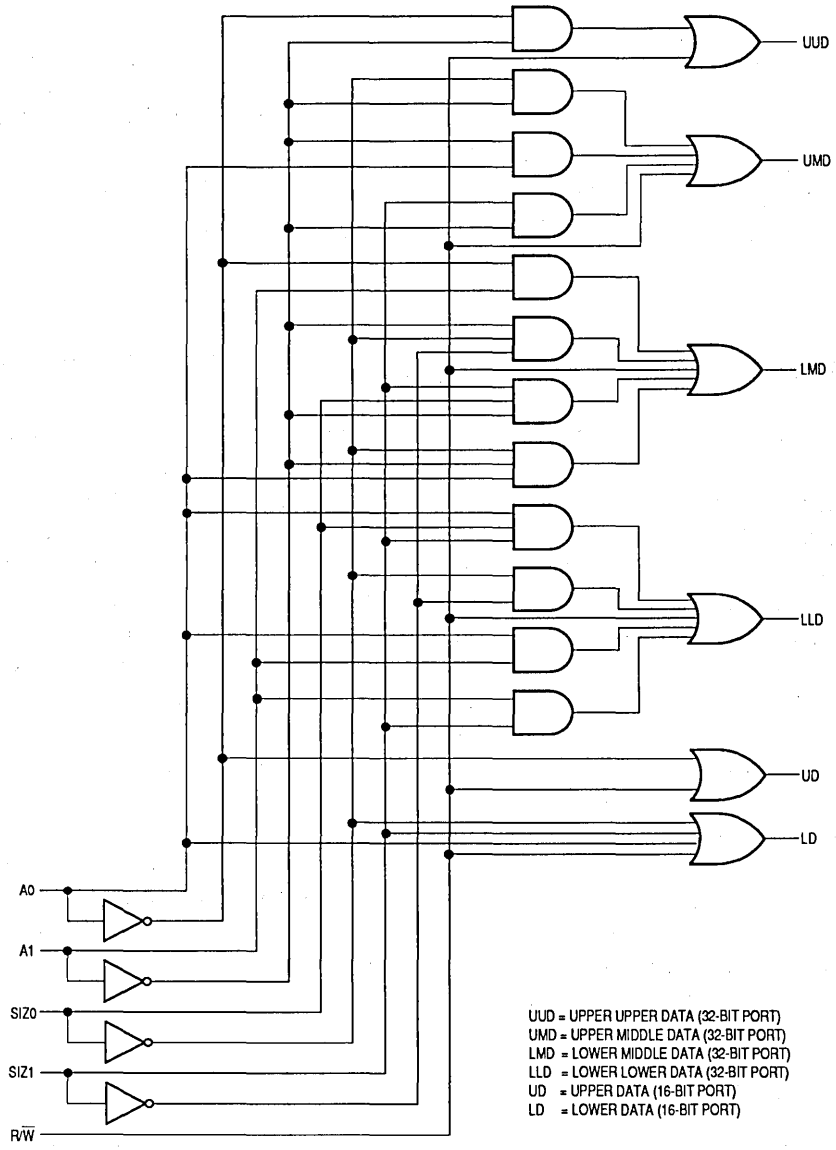
7.2.6 Cache Filling

The on-chip data and instruction caches, described in **SECTION 6 ON-CHIP CACHE MEMORIES**, are each organized as 16 lines of four long-word entries each. For each line, a tag contains the most significant bits of the address, FC2 (instruction cache) or FC0–FC2 (data cache), and a valid bit for each entry in the line. An entry fill operation loads an entire long word accessed from memory into a cache entry. This type of fill operation is performed when one entry of a line is not valid and an access is cacheable. A burst fill operation is requested when a tag miss occurs for the current cycle or when all four entires in the cache line are invalid (provided the cache is enabled and burst filling for the cache is enabled). The burst fill operation attempts to fill all four entries in the line. To support burst filling, the slave device must have a 32-bit port and must have a burst mode capability; that is, it must acknowledge a burst request with the cache burst acknowledge ($\overline{\text{CBACK}}$) signal. It must also terminate the burst accesses with $\overline{\text{STERM}}$ and place a long word on the data bus for each transfer. The device may continue to supply successive long words, asserting $\overline{\text{STERM}}$ with each one, until the cache line is full. For further information about filling the cache, both entry fills and burst mode fills, refer to **6.1.3 Cache Filling**, **7.3.4 Synchronous Read Cycle**, **7.3.5 Synchronous Write Cycle**, and **7.3.7 Burst Operation Cycles**, which discuss in detail the required bus cycles.

7.2.7 Cache Interactions

The organization and requirements of the on-chip instruction and data caches affect the interpretation of the $\overline{\text{DSACKx}}$ and $\overline{\text{STERM}}$ signals. Since the MC68EC030 attempts to load all data operands and instructions that are

cacheable into the on-chip caches, the bus may operate differently when caching is enabled. Specifically, on cacheable read cycles that terminate normally, the low-order address signals (A0 and A1) and the size signals do not apply.



NOTE: These select lines can be combined with the address decode circuitry, or all can be generated within the same programmed array logic unit.

Figure 7-18. Byte Data Select Generation for 16- and 32-Bit Ports

The slave device must supply as much aligned data on the data bus as its port size allows, regardless of the requested operand size. This means that an 8-bit port must supply a byte, a 16-bit port must supply a word, and a 32-bit port must supply an entire long word. This data is loaded into the cache. For a 32-bit port, the slave device ignores A0 and A1 and supplies the long word beginning at the long-word boundary on the data bus. For a 16-bit port, the device ignores A0 and supplies the entire word beginning at the lower word boundary on D16–D31 of the data bus. For a byte port, the device supplies the addressed byte on D24–D31.

If the addressed device cannot supply port-sized data or if the data should not be cached, the device must assert cache inhibit in ($\overline{\text{CIIN}}$) as it terminates the read cycle. If the bus cycle terminates abnormally, the MC68EC030 does not cache the data. For details of interactions of port sizes, misalignments, and cache filling, refer to **6.1.3 Cache Filling**.

The caches can also affect the assertion of $\overline{\text{AS}}$ and the operation of a read cycle. The search of the appropriate cache by the controller begins when the microsequencer requires an instruction or a data item. At this time, the bus controller may also initiate an external bus cycle in case the requested item is not resident in the instruction or data cache. If the bus is not occupied with another read or write cycle, the bus controller asserts the $\overline{\text{ECS}}$ signal (and the $\overline{\text{OCS}}$ signal, if appropriate). If an internal cache hit occurs, the external cycle aborts, and $\overline{\text{AS}}$ is not asserted. This makes it possible to have $\overline{\text{ECS}}$ asserted on multiple consecutive clock cycles. Notice that there is a minimum time specified from the negation of ECS to the next assertion of $\overline{\text{ECS}}$ (refer to MC68EC030/D, *MC68EC030 Technical Summary*.

Instruction prefetches can occur every other clock so that if, after an aborted cycle due to an instruction cache hit, the bus controller asserts $\overline{\text{ECS}}$ on the next clock, this second cycle is for a data fetch. However, data accesses that hit in the data cache can also cause the assertion of $\overline{\text{ECS}}$ and an aborted cycle. Therefore, since instruction and data accesses are mixed, it is possible to see multiple successive $\overline{\text{ECS}}$ assertions on the external bus if the controller is hitting in both caches and if the bus controller is free. Note that, if the bus controller is executing other cycles, these aborted cycles due to cache hits may not be seen externally. Also, $\overline{\text{OCS}}$ is asserted for the first *external* cycle of an operand transfer. Therefore, in the case of a misaligned data transfer where the first portion of the operand results in a cache hit (but the bus controller did not begin an external cycle and then abort it) and the second portion in a cache miss, $\overline{\text{OCS}}$ is asserted for the second portion of the operand.

7.2.8 Asynchronous Operation

The MC68EC030 bus may be used in an asynchonous manner. In that case, the external devices connected to the bus can operate at clock frequencies different from the clock for the MC68EC030. Asynchronous operation requires using only the handshake line ($\overline{AS}$, $\overline{DS}$, $\overline{DSACK1}$, $\overline{DSACK0}$, $\overline{BERR}$, and $\overline{HALT}$) to control data transfers. Using this method, $\overline{AS}$ signals the start of a bus cycle, and $\overline{DS}$ is used as a condition for valid data on a write cycle. Decoding the size outputs and lower address lines (A0 and A1) provides strobes that select the active portion of the data bus. The slave device (memory or peripheral) then responds by placing the requested data on the correct portion of the data bus for a read cycle or latching the data on a write cycle, and asserting the $\overline{DSACK1}$/$\overline{DSACK0}$ combination that corresponds to the port size to terminate the cycle. If no slave responds or the access is invalid, external control logic asserts the $\overline{BERR}$ or $\overline{BERR}$ and $\overline{HALT}$ signal(s) to abort or retry the bus cycle, respectively.

The $\overline{DSACKx}$ signals can be asserted before the data from a slave device is valid on a read cycle. The length of time that $\overline{DSACKx}$ may precede data is given by parameter #31, and it must be met in any asynchronous system to insure that valid data is latched into the controller. (Refer to MC68EC030/D, *MC68EC030 Technical Summary* for timing parameters.) Notice that no maximum time is specified from the assertion of $\overline{AS}$ to the assertion of $\overline{DSACKx}$. Although the controller can transfer data in a minimum of three clock cycles when the cycle is terminated with $\overline{DSACKx}$, the controller inserts wait cycles in clock period increments until $\overline{DSACKx}$ is recognized.

The $\overline{BERR}$ and/or $\overline{HALT}$ signals can be asserted after the $\overline{DSACKx}$ signal(s) is asserted. $\overline{BERR}$ and/or $\overline{HALT}$ must be asserted within the time given as parameter #48, after $\overline{DSACKx}$ is asserted in any asynchronous system. If this maximum delay time is violated, the controller may exhibit erratic behavior.

For asynchronous read cycles, the value of $\overline{CIIN}$ is internally latched on the rising edge of bus cycle state 4. Refer to **7.3.1 Asynchronous Read Cycle** for more details on the states for asynchonous read cycles.

During any bus cycle terminated by $\overline{DSACKx}$ or $\overline{BERR}$, the assertion of $\overline{CBACK}$ is completely ignored.

7.2.9 Synchronous Operation with DSACKx

Although cycles terminated with the DSACKx signals are classified as asynchronous and cycles terminated with STERM are classified as synchronous, cycles terminated with DSACKx can also operate synchronously in that signals are interpreted relative to clock edges.

The devices that use these cycles must synchronize the responses to the MC68EC030 clock to be synchronous. Since they terminate bus cycles with the DSACKx signals, the dynamic bus sizing capabilities of the MC68EC030 are available. In addition, the minimum cycle time for these cycles is also three clocks.

To support those systems that use the system clock to generate DSACKx and other asynchronous inputs, the asynchronous input setup time (parameter #47A) and the asynchronous input hold time (parameter #47B) are given. If the setup and hold times are met for the assertion or negation of a signal, such as DSACKx, the controller can be guaranteed to recognize that signal level on that specific falling edge of the system clock. If the assertion of DSACKx is recognized on a particular falling edge of the clock, valid data is latched into the controller (for a read cycle) on the next falling clock edge provided the data meets the data setup time (parameter #27). In this case, parameter #31 for asynchronous operation can be ignored. The timing parameters referred to are described in MC68EC030/D, *MC68EC030 Technical Summary*. If a system asserts DSACKx for the required window around the falling edge of S2 and obeys the proper bus protocol by maintaining DSACKx (and/or BERR/HALT) until and throughout the clock edge that negates AS (with the appropriate asynchronous input hold time specified by parameter #47B), no wait states are inserted. The bus cycle runs at its maximum speed (three clocks per cycle) for bus cycles terminated with DSACKx.

To assure proper operation in a synchronous system when BERR or BERR and HALT is asserted after DSACKx, BERR (and HALT) must meet the appropriate setup time (parameter #27A) prior to the falling clock edge one clock cycle after DSACKx is recognized. This setup time is critical, and the MC68EC030 may exhibit erratic behavior if it is violated.

When operating synchronously, the data-in setup and hold times for synchronous cycles may be used instead of the timing requirements for data relative to the DS signal.

The value of CIIN is latched on the rising edge of bus cycle state 4 for all cycles terminated with DSACKx.

7.2.10 Synchronous Operation with $\overline{\text{STERM}}$

The MC68EC030 supports synchronous bus cycles terminated with $\overline{\text{STERM}}$. These cycles, for 32-bit ports only, are similar to cycles terminated with $\overline{\text{DSACK}}$x. The main difference is that $\overline{\text{STERM}}$ can be asserted (and data can be transferred) earlier than for a cycle terminated with $\overline{\text{DSACK}}$x, causing the controller to perform a minimum access time transfer in two clock periods. However, wait cycles can be inserted by delaying the assertion of $\overline{\text{STERM}}$ appropriately.

Using $\overline{\text{STERM}}$ instead of $\overline{\text{DSACK}}$x in any bus cycle makes the cycle synchronous. Any bus cycle is synchronous if:

1. Neither $\overline{\text{DSACK}}$x nor $\overline{\text{AVEC}}$ is recognized during the cycle.
2. The port size is 32 bits.
3. Synchronous input setup and hold time requirements (specifications #60 and #61) for $\overline{\text{STERM}}$ are met.

7

Burst mode operation requires the use of $\overline{\text{STERM}}$ to terminate each of its cycles. The first cycle of any burst transfer must be a synchronous cycle as described in the preceding paragraph. The exact timing of this cycle is controlled by the assertion of $\overline{\text{STERM}}$, and wait cycles can be inserted as necessary. However, the minimum cycle time is two clocks. If a burst operation is initiated and allowed to terminate normally, the second, third, and fourth cycles latch data on successive falling edges of the clock at a minimum. Again, the exact timing for these subsequent cycles is controlled by the timing of $\overline{\text{STERM}}$ for each of these cycles, and wait cycles can be inserted as necessary.

Although the synchronous input signals ($\overline{\text{STERM}}$, $\overline{\text{CIIN}}$, and $\overline{\text{CBACK}}$) must be stable for the appropriate setup and hold times relative to every rising edge of the clock during which $\overline{\text{AS}}$ is asserted, the assertion or negation of $\overline{\text{CBACK}}$ and $\overline{\text{CIIN}}$ is internally latched on the rising edge of the clock for which $\overline{\text{STERM}}$ is asserted in a synchronous cycle.

The $\overline{\text{STERM}}$ signal can be generated from the address bus and function code value and does not need to be qualified with the $\overline{\text{AS}}$ signal. If $\overline{\text{STERM}}$ is asserted and no cycle is in progress (even if the cycle has begun, $\overline{\text{ECS}}$ is asserted and then the cycle is aborted), $\overline{\text{STERM}}$ is ignored by the MC68EC030.

Similarly, $\overline{\text{CBACK}}$ can be asserted independently of the assertion of $\overline{\text{CBREQ}}$. If a cache burst is not requested, the assertion of $\overline{\text{CBACK}}$ is ignored.

The assertion of $\overline{\text{CIIN}}$ is ignored when the appropriate cache is not enabled or when cache inhibit out ($\overline{\text{CIOUT}}$) is asserted. It is also ignored during write cycles or translation table searches.

NOTE

$\overline{\text{STERM}}$ and $\overline{\text{DSACKx}}$ should *never* be asserted during the same bus cycle.

7.3 DATA TRANSFER CYCLES

The transfer of data between the controller and other devices involves the following signals:

- Address Bus A0–A31
- Data Bus D0–D31
- Control Signals

The address and data buses are both parallel nonmultiplexed buses. The bus master moves data on the bus by issuing control signals, and the asynchronous/synchronous bus uses a handshake protocol to insure correct movement of the data. In all bus cycles, the bus master is responsible for de-skewing all signals it issues at both the start and the end of the cycle. In addition, the bus master is responsible for de-skewing the acknowledge and data signals from the slave devices. The following paragraphs define read, write, and read-modify-write cycle operations. An additional paragraph describes burst mode transfers.

Each of the bus cycles is defined as a succession of states. These states apply to the bus operation and are different from the controller states described in **SECTION 4 PROCESSING STATES**. The clock cycles used in the descriptions and timing diagrams of data transfer cycles are independent of the clock frequency. Bus operations are described in terms of external bus states.

7.3.1 Asynchronous Read Cycle

During a read cycle, the controller receives data from a memory, coprocessor, or peripheral device. If the instruction specifies a long-word operation, the MC68EC030 attempts to read four bytes at once. For a word operation, it attempts to read two bytes at once, and for a byte operation, one byte. For some operations, the controller requests a three-byte transfer. The controller properly positions each byte internally. The section of the data bus from

which each byte is read depends on the operand size, address signals (A0–A1), $\overline{CIIN}$ and $\overline{CIOUT}$, whether the internal caches are enabled, and the port size. Refer to **7.2.1 Dynamic Bus Sizing**, **7.2.2 Misaligned Operands**, and **7.2.6 Cache Filling** for more information on dynamic bus sizing, misaligned operands, and cache interactions.

Figure 7-19 is a flowchart of an asynchronous long-word read cycle. Figure 7-20 is a flowchart of a byte read cycle. The following figures show functional read cycle timing diagrams specified in terms of clock periods. Figure 7-21 corresponds to byte and word read cycles from a 32-bit port. Figure 7-22 corresponds to a long-word read cycle from an 8-bit port. Figure 7-23 also applies to a long-word read cycle, but from a 16-bit port.

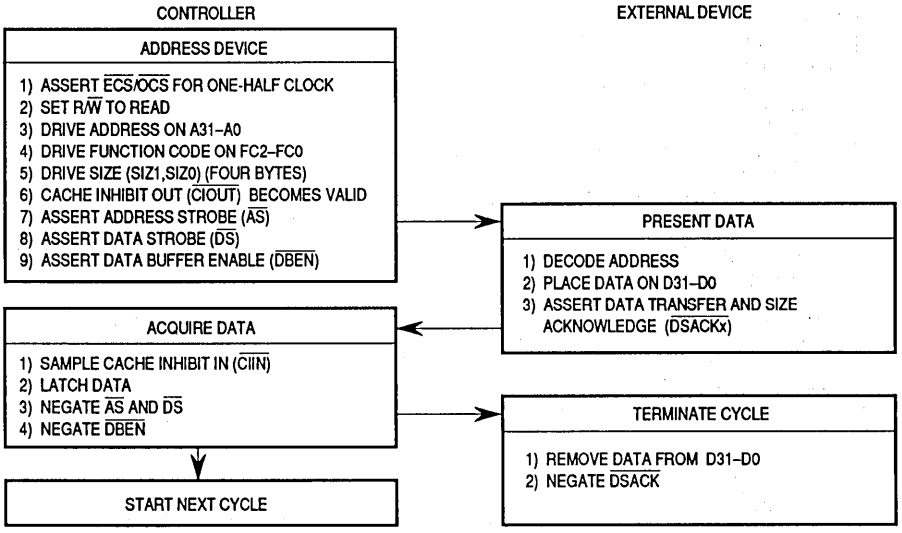


Figure 7-19. Asynchronous Long-Word Read Cycle Flowchart

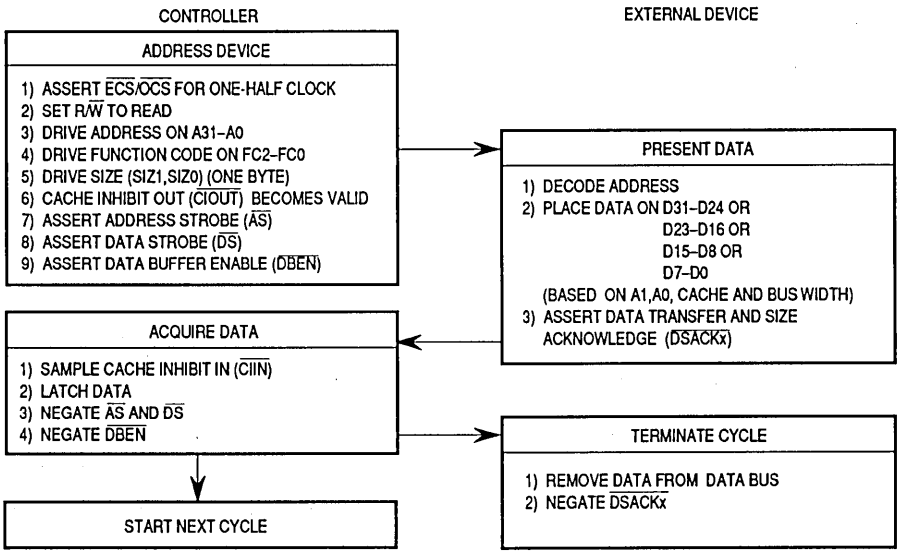


Figure 7-20. Asynchronous Byte Read Cycle Flowchart

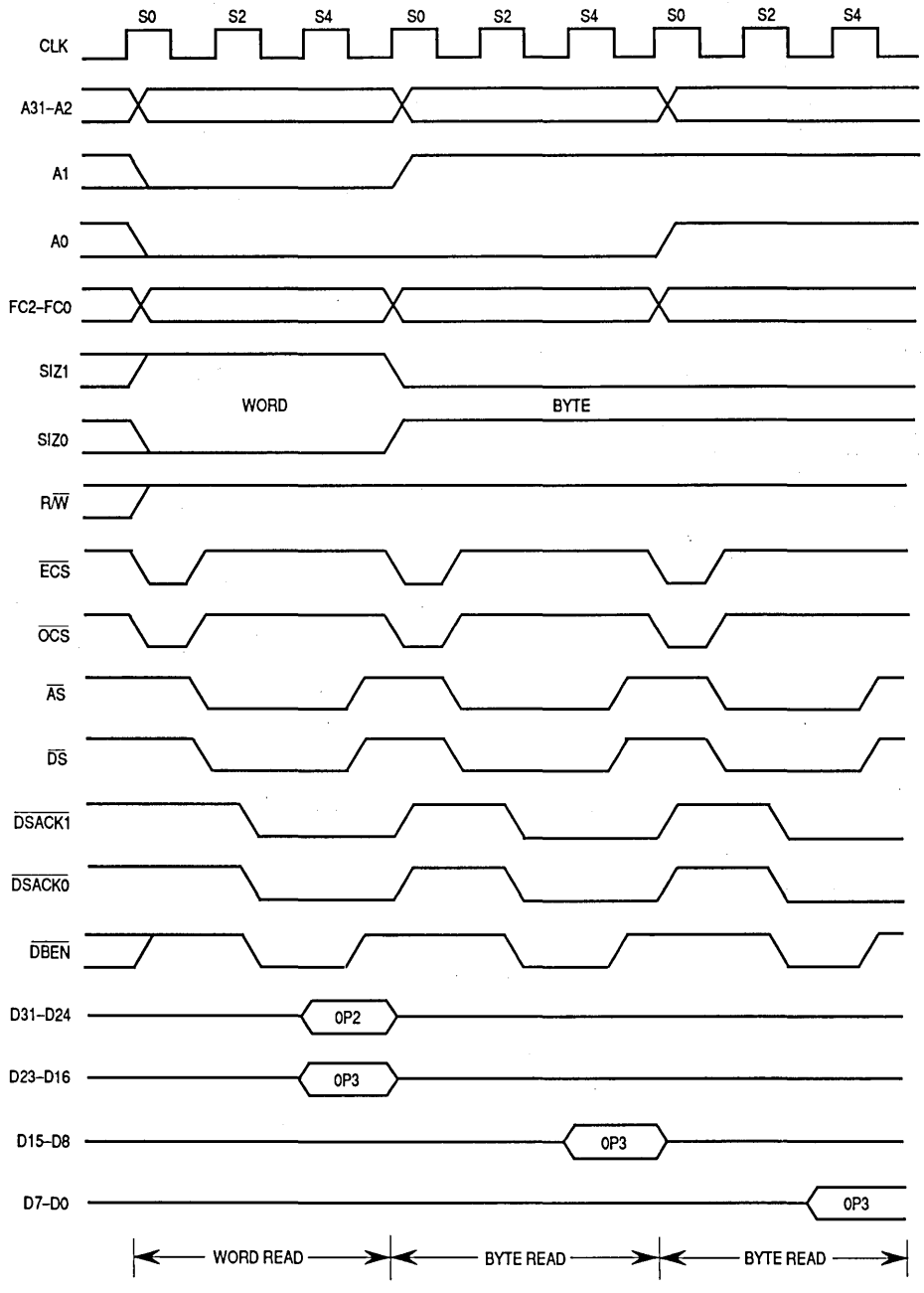


Figure 7-21. Asynchronous Byte and Word Read Cycles — 32-Bit Port

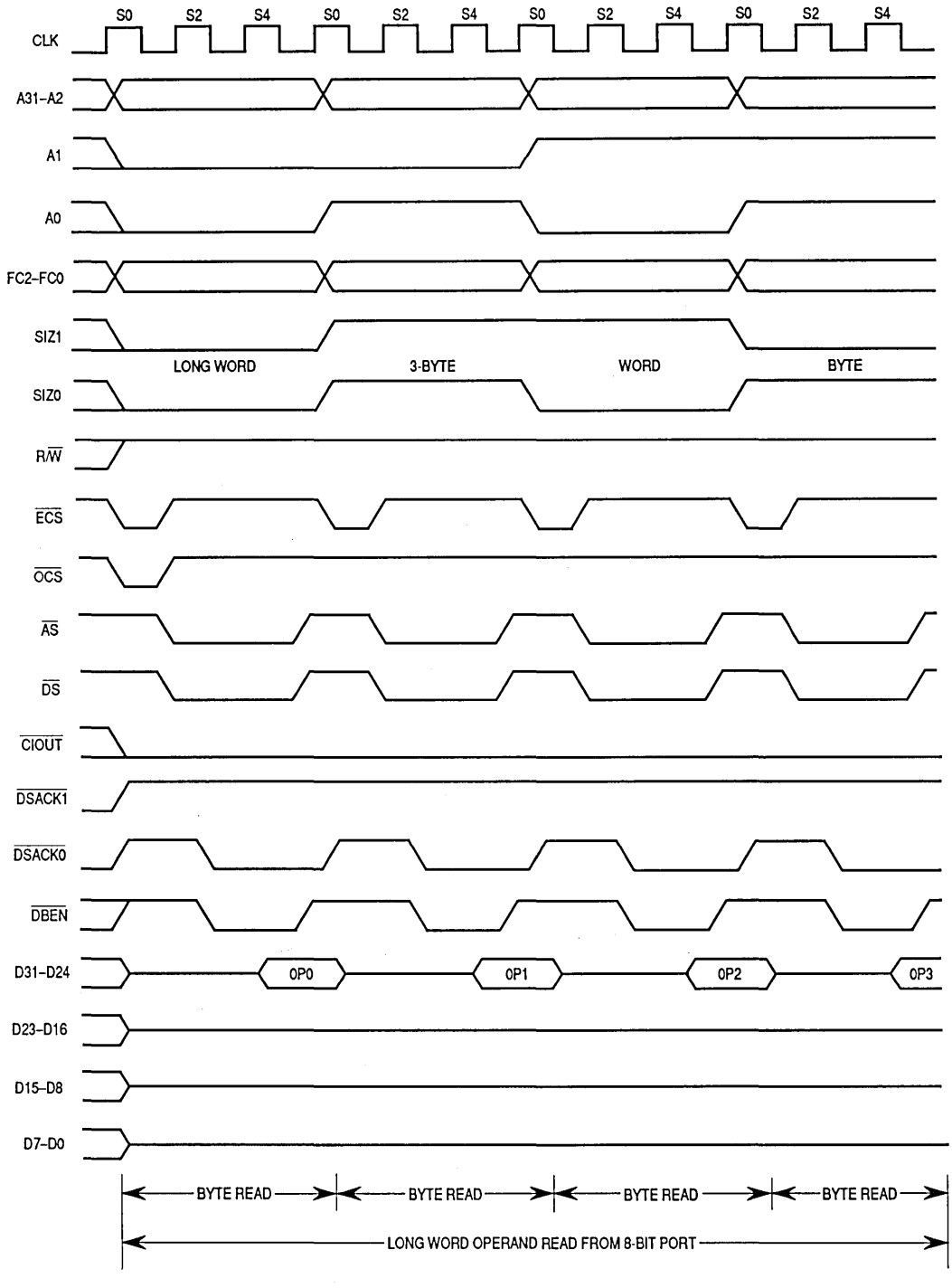


Figure 7-22. Long-Word Read — 8-Bit Port with $\overline{\text{CIOUT}}$ Asserted

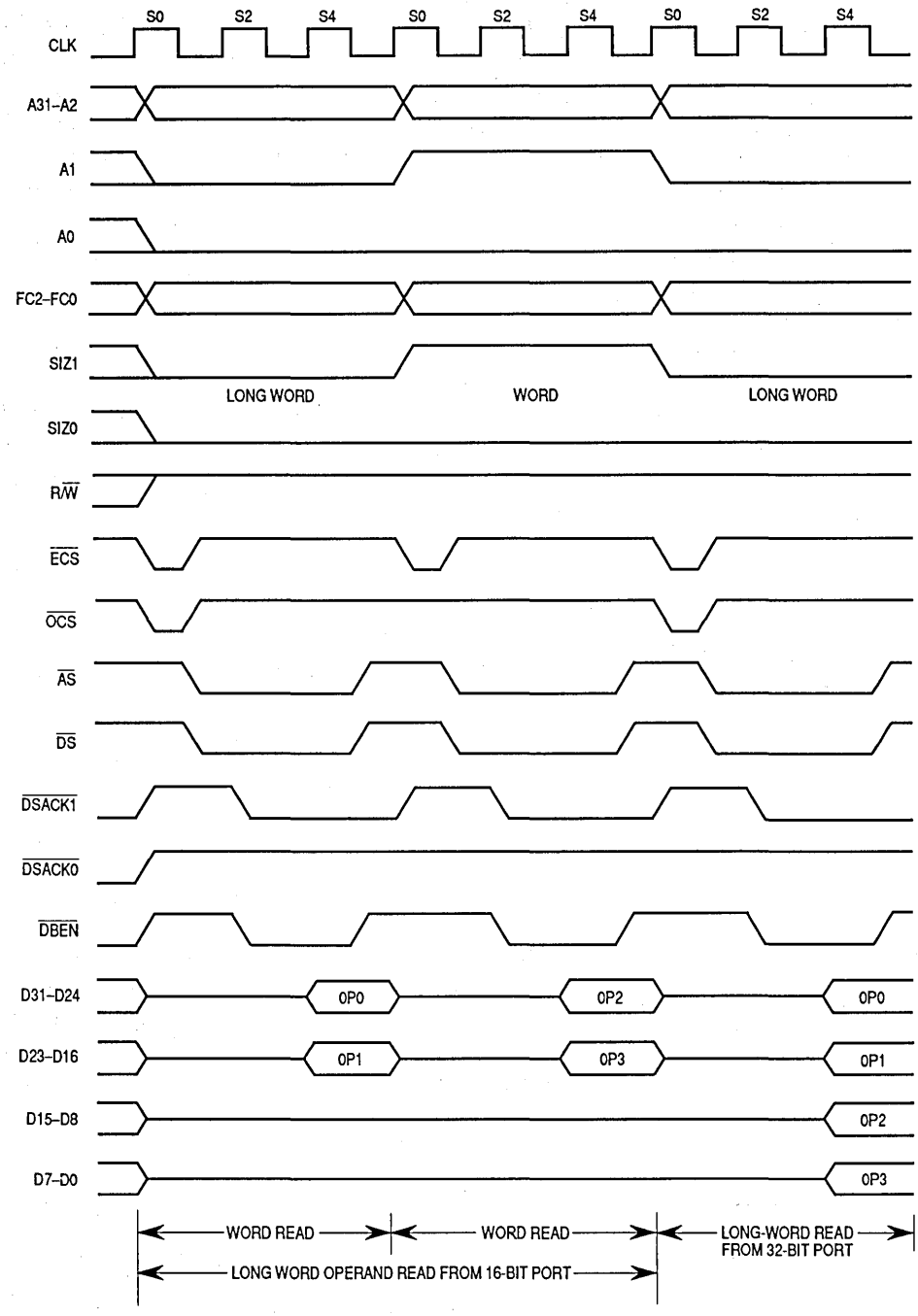


Figure 7-23. Long-Word Read — 16-Bit and 32-Bit Port

State 0

The read cycle starts in state 0 (S0). The controller drives $\overline{ECS}$ low, indicating the beginning of an external cycle. When the cycle is the first external cycle of a read operand operation, operand cycle start ($\overline{OCS}$) is driven low at the same time. During S0, the controller places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The controller drives R/$\overline{W}$ high for a read cycle and drives $\overline{DBEN}$ inactive to disable the data buffers. SIZ0–SIZ1 become valid, indicating the number of bytes requested to be transferred. $\overline{CIOUT}$ also becomes valid, indicating the state of the ACU CI bit in the access control register.

State 1

One-half clock later in state 1 (S1), the controller asserts $\overline{AS}$ indicating that the address on the address bus is valid. The controller also asserts $\overline{DS}$ also during S1. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

State 2

During state 2 (S2), the controller asserts $\overline{DBEN}$ to enable external data buffers. The selected device uses R/$\overline{W}$, SIZ0–SIZ1, A0–A1, $\overline{CIOUT}$, and $\overline{DS}$ to place its information on the data bus, and drives $\overline{CIIN}$ if appropriate. Any or all of the bytes (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. Concurrently, the selected device asserts $\overline{DSACK}$x.

State 3

As long as at least one of the $\overline{DSACK}$x signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), data is latched on the next falling edge of the clock, and the cycle terminates. If $\overline{DSACK}$x is not recognized by the start of state 3 (S3), the controller inserts wait states instead of proceeding to states 4 and 5. To ensure that wait states are inserted, both $\overline{DSACK0}$ and $\overline{DSACK1}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the controller continues to sample the $\overline{DSACK}$x signals on the falling edges of the clock until one is recognized.

State 4

The controller samples $\overline{CIIN}$ at the beginning of state 4 (S4). Since $\overline{CIIN}$ is defined as a synchronous input, whether asserted or negated, it must meet the appropriate synchronous input setup and hold times on every rising edge of the clock while $\overline{AS}$ is asserted. At the end of S4, the controller latches the incoming data.

State 5

The controller negates $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ during state 5 (S5). It holds the address valid during S5 to provide address hold time for memory systems. R/$\overline{W}$, SIZ0–SIZ1, and FC0–FC2 also remain valid throughout S5.

The external device keeps its data and $\overline{DSACKx}$ signals asserted until it detects the negation of $\overline{AS}$ or $\overline{DS}$ (whichever it detects first). The device must remove its data and negate $\overline{DSACKx}$ within approximately one clock period after sensing the negation of $\overline{AS}$ or $\overline{DS}$. $\overline{DSACKx}$ signals that remain asserted beyond this limit may be prematurely detected for the next bus cycle.

7.3.2 Asynchronous Write Cycle

During a write cycle, the controller transfers data to memory or a peripheral device.

7

Figure 7-24 is a flowchart of a write cycle operation for a long-word transfer. The following figures show the functional write cycle timing diagrams specified in terms of clock periods. Figure 7-25 shows two write cycles (between two read cycles with no idle time) for a 32-bit port. Figure 7-26 shows byte and word write cycles to a 32-bit port. Figure 7-27 shows a long-word write cycle to an 8-bit port. Figure 7-28 shows a long-word write cycle to a 16-bit port.

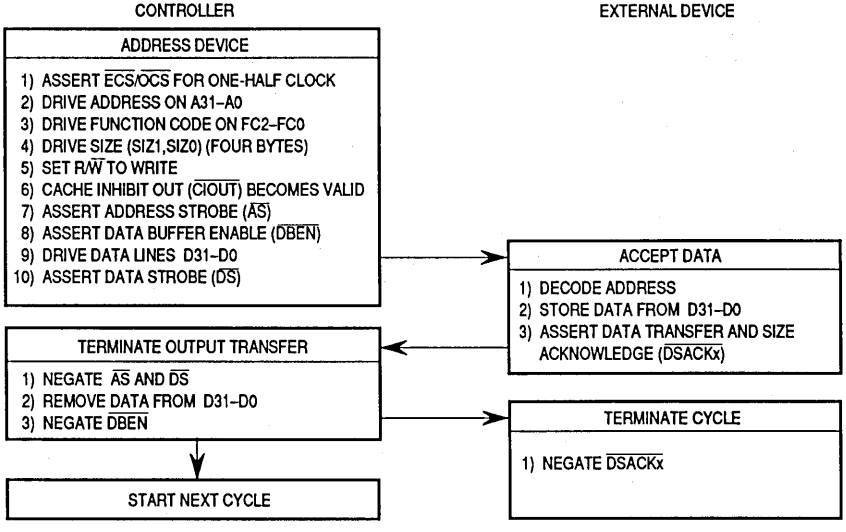


Figure 7-24. Asynchronous Write Cycle Flowchart

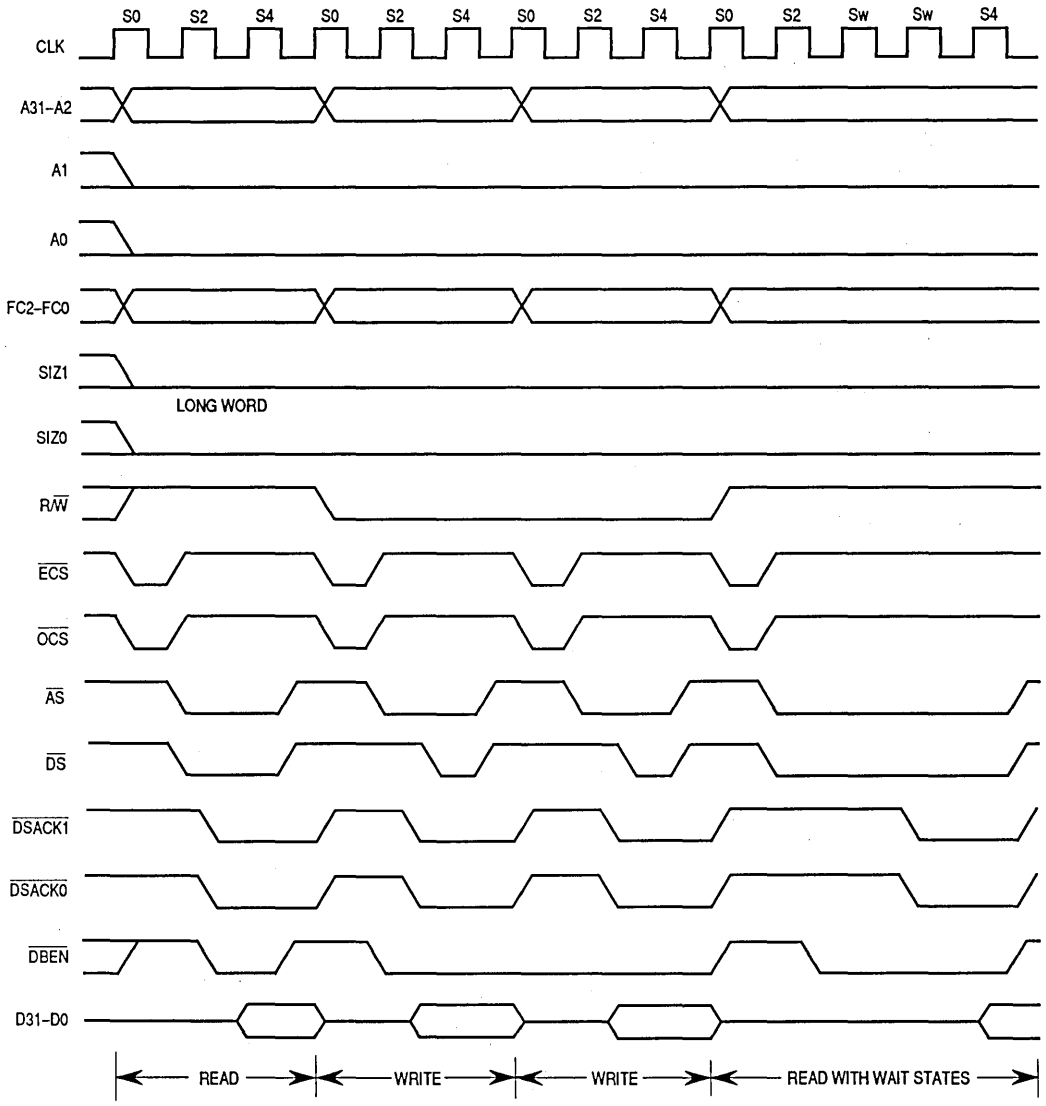


Figure 7-25. Asynchronous Read-Write-Read Cycles — 32-Bit Port

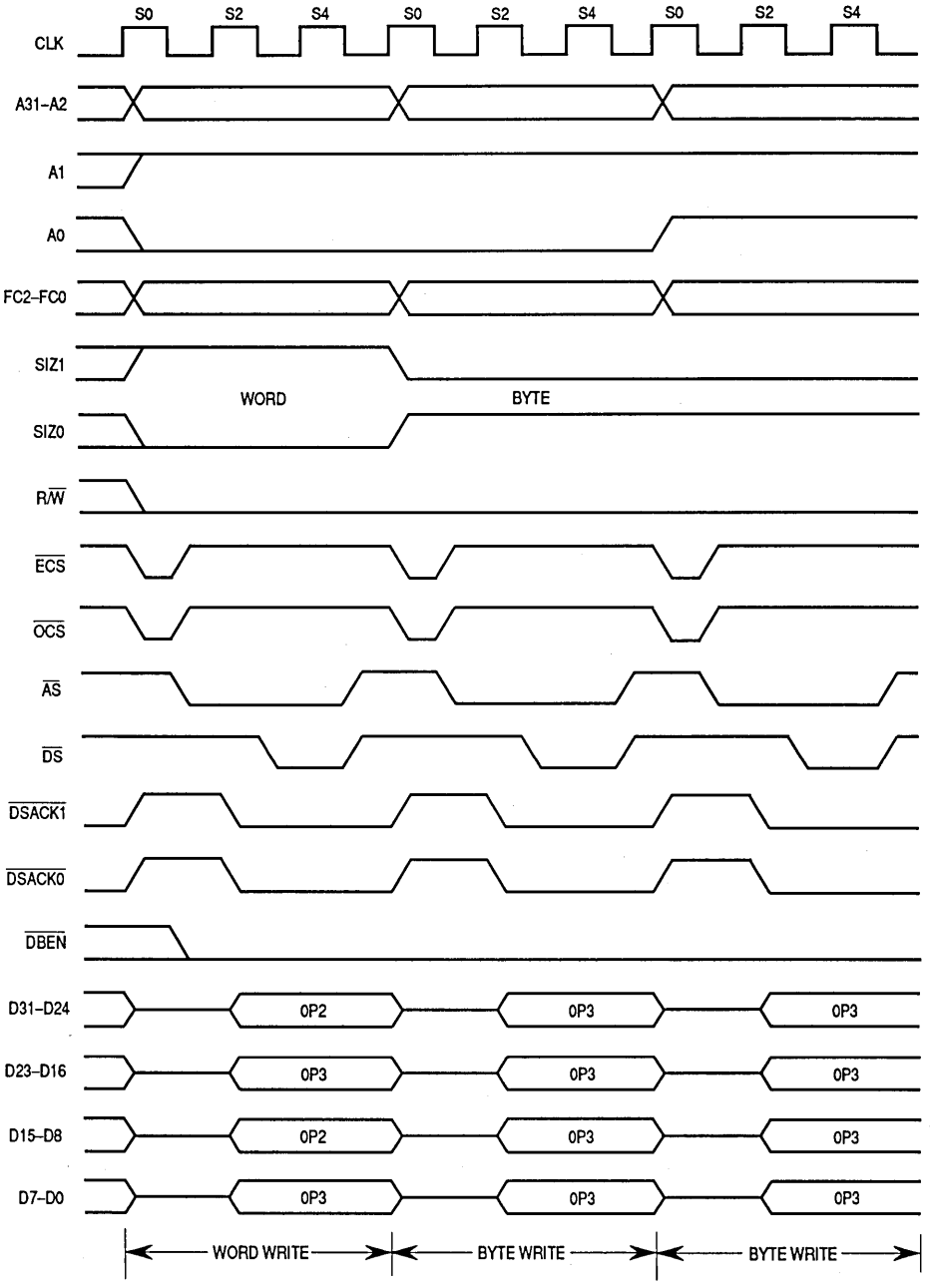


Figure 7-26. Asynchronous Byte and Word Write Cycles — 32-Bit Port

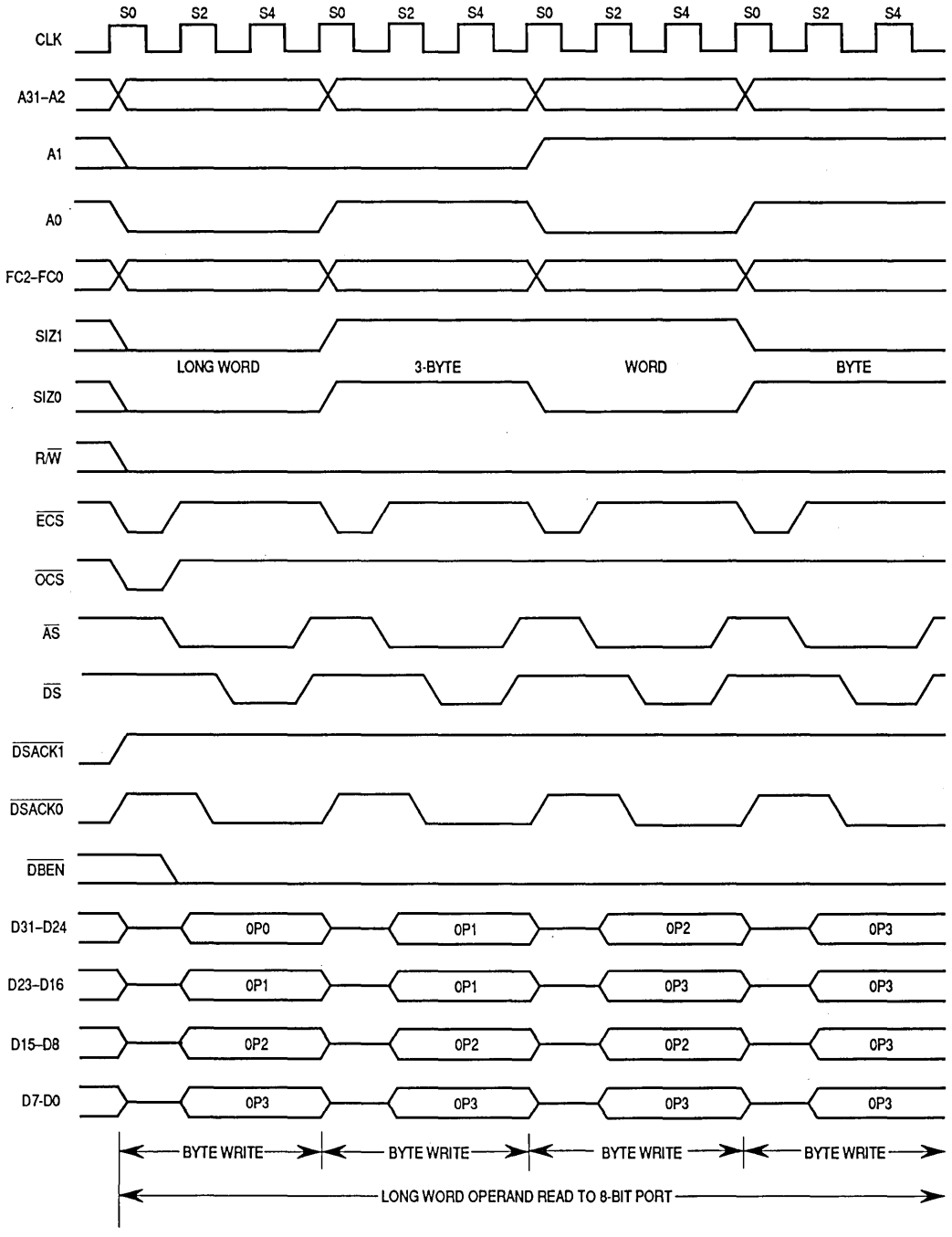


Figure 7-27. Long-Word Operand Write — 8-Bit Port

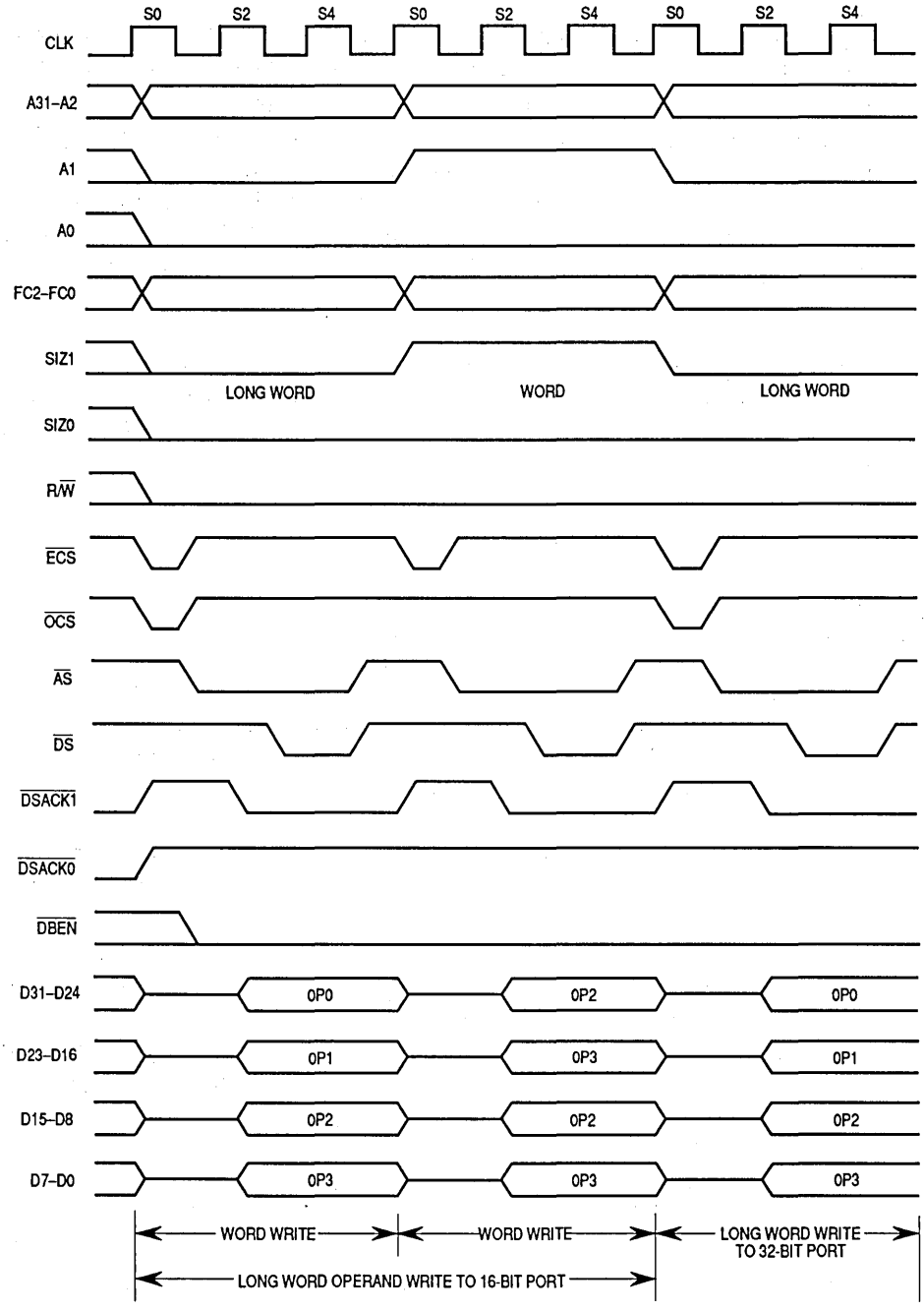


Figure 7-28. Long-Word Operand Write — 16-Bit Port

State 0

The write cycle starts in S0. The controller drives $\overline{ECS}$ low, indicating the beginning of an external cycle. When the cycle is the first external cycle of a write operation, $\overline{OCS}$ is driven low at the same time. During S0, the controller places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The controller drives R/$\overline{W}$ low for a write cycle. SIZ0–SIZ1 become valid, indicating the number of bytes to be transferred. $\overline{CIOUT}$ also becomes valid, indicating the state of the ACU CI bit in the access control register.

State 1

One-half clock later in S1, the controller asserts $\overline{AS}$, indicating that the address on the address bus is valid. The controller also asserts $\overline{DBEN}$ during S1, which can enable external data buffers. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

State 2

During S2, the controller places the data to be written onto the D0–D31, and samples $\overline{DSACKx}$ at the end of S2.

State 3

The controller asserts $\overline{DS}$ during S3, indicating that the data is stable on the data bus. As long as at least one of the $\overline{DSACKx}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), the cycle terminates one clock later. If $\overline{DSACKx}$ is not recognized by the start of S3, the controller inserts wait states instead of proceeding to S4 and S5. To ensure that wait states are inserted, both $\overline{DSACK0}$ and $\overline{DSACK1}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the controller continues to sample the $\overline{DSACKx}$ signals on the falling edges of the clock until one is recognized. The selected device uses R/$\overline{W}$, $\overline{DS}$, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate byte(s) of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7). SIZ0–SIZ1 and A0–A1 select the bytes of the data bus. If it has not already done so, the device asserts $\overline{DSACKx}$ to signal that it has successfully stored the data.

State 4

The controller issues no new control signals during S4.

State 5

The controller negates $\overline{AS}$ and $\overline{DS}$ during S5. It holds the address and data valid during S5 to provide address hold time for memory systems. R/$\overline{W}$, SIZ0–SIZ1, FC0–FC2, and $\overline{DBEN}$ also remain valid throughout S5.

The external device must keep $\overline{\text{DSACKx}}$ asserted until it detects the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$ (whichever it detects first). The device must negate $\overline{\text{DSACKx}}$ within approximately one clock period after sensing the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$. $\overline{\text{DSACKx}}$ signals that remain asserted beyond this limit may be prematurely detected for the next bus cycle.

7.3.3 Asynchronous Read-Modify-Write Cycle

The read-modify-write cycle performs a read, conditionally modifies the data in the arithmetic logic unit, and may write the data out to memory. In the MC68EC030 controller, this operation is indivisible, providing semaphore capabilities for multiprocessor systems. During the entire read-modify-write sequence, the MC68EC030 asserts the $\overline{\text{RMC}}$ signal to indicate that an indivisible operation is occurring. The MC68EC030 does not issue a bus grant ($\overline{\text{BG}}$) signal in response to a bus request ($\overline{\text{BR}}$) signal during this operation. The read portion of a read-modify-write operation is forced to miss in the data cache because the data in the cache would not be valid if another controller had altered the value being read. However, read-modify-write cycles may alter the contents of the data cache as described in **6.1.2. Data Cache.**

No burst filling of the data cache occurs during a read-modify-write operation.

The test and set (TAS) and compare and swap (CAS and CAS2) instructions are the only MC68EC030 instructions that utilize read-modify-write operations. Depending on the compare results of the CAS and CAS2 instructions, the write cycle(s) may not occur.

Figure 7-29 is a flowchart of the asynchronous read-modify-write cycle operation. Figure 7-30 is an example of a functional timing diagram of a TAS instruction specified in terms of clock periods.

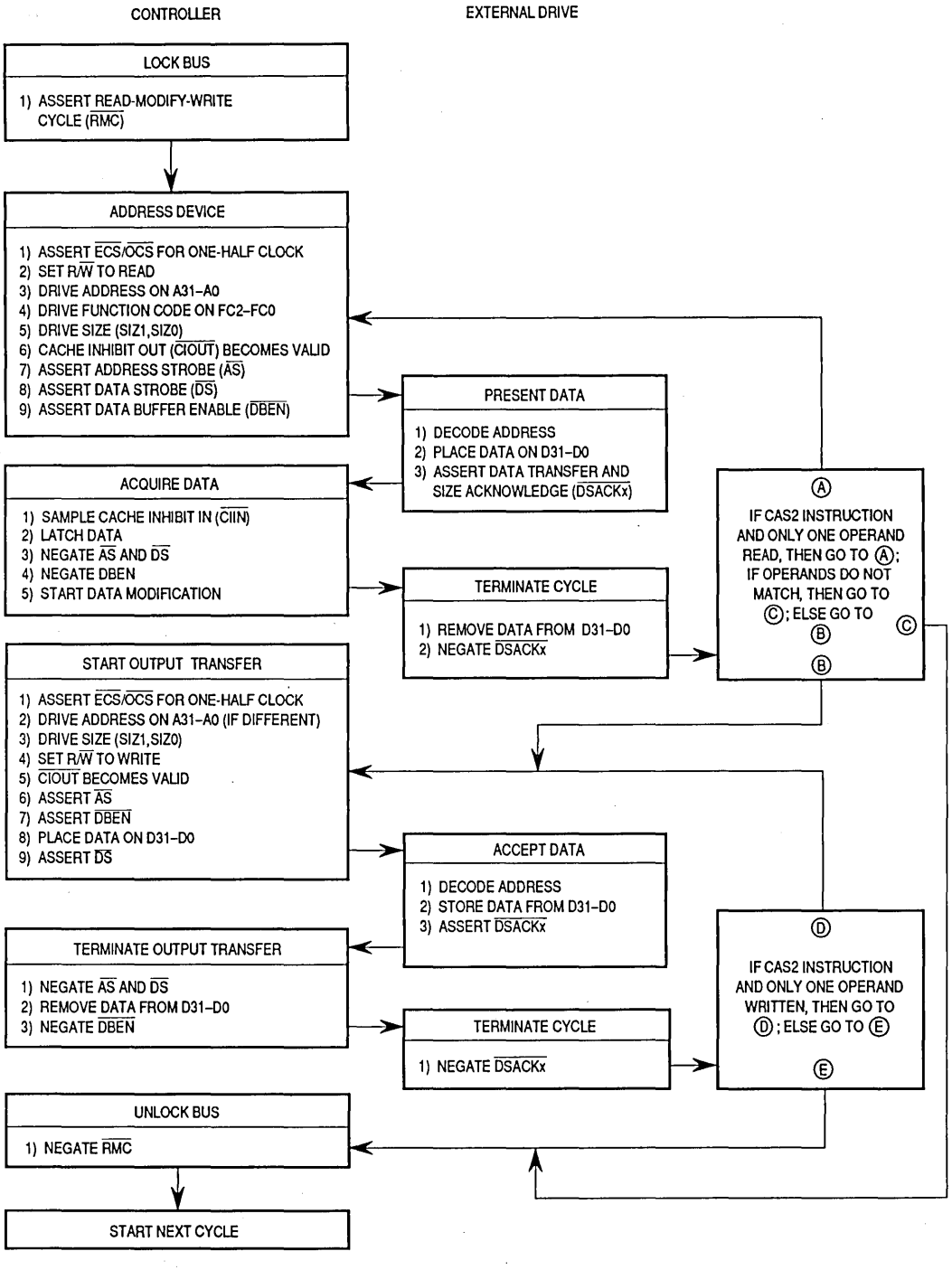


Figure 7-29. Asynchronous Read-Modify-Write Cycle Flowchart

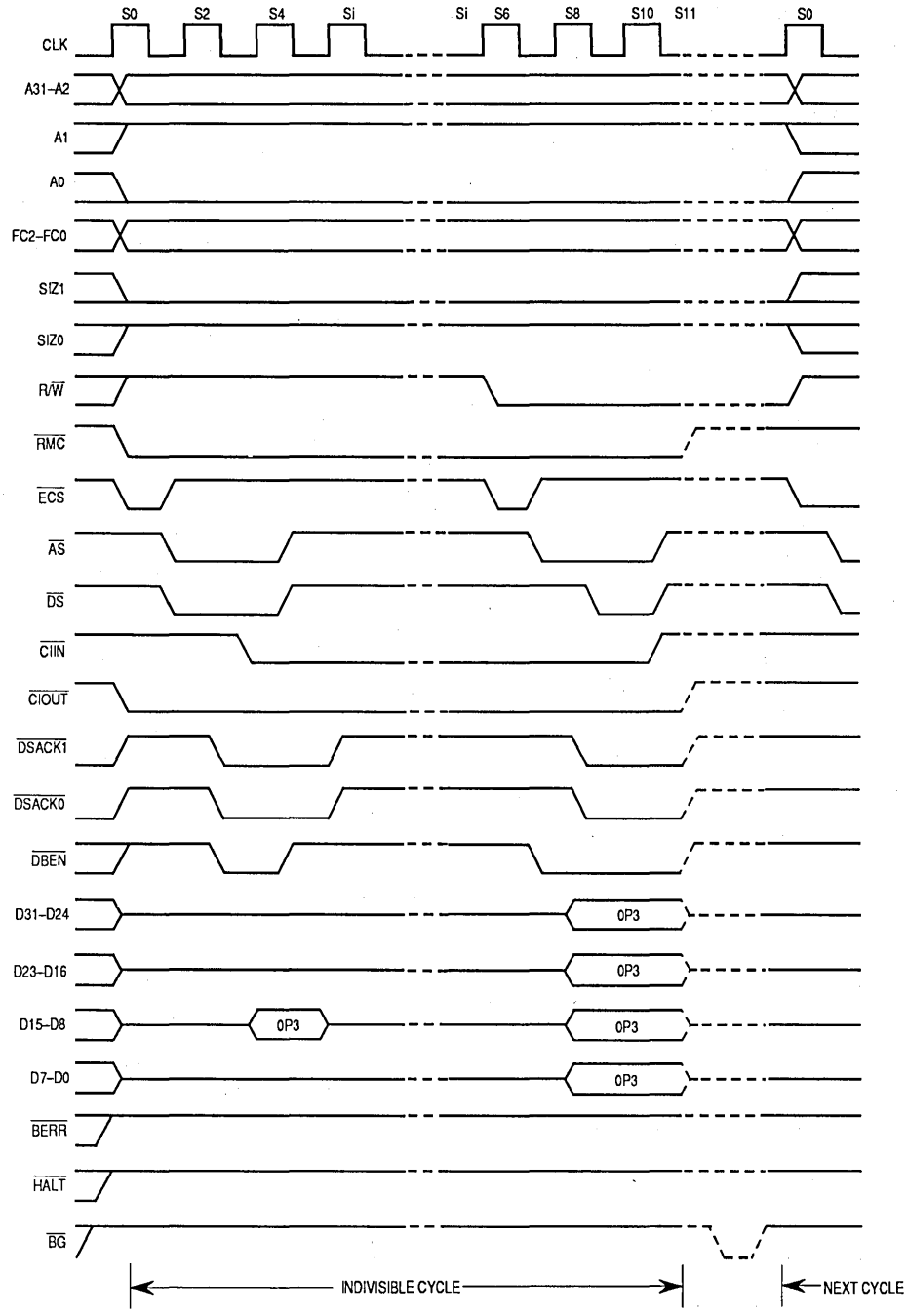


Figure 7-30. Asynchronous Byte Read-Modify-Write Cycle — 32-Bit Port (TAS Instruction with $\overline{\text{CIOUT}}$ or $\overline{\text{CIIN}}$ Asserted)

State 0

The controller asserts $\overline{ECS}$ and $\overline{OCS}$ in S0 to indicate the beginning of an external operand cycle. The controller also asserts $\overline{RMC}$ in S0 to identify a read-modify-write cycle. The controller places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the operation. SIZ0–SIZ1 become valid in S0 to indicate the operand size. The controller drives R/$\overline{W}$ high for the read cycle and sets $\overline{CIOUT}$ according to the value of the ACU CI bit in the access control register.

State 1

One-half clock later in S1, the controller asserts $\overline{AS}$, indicating that the address on the address bus is valid. The controller asserts $\overline{DS}$ during S1. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

State 2

During state 2 (S2), the controller drives $\overline{DBEN}$ active to enable external data buffers. The selected device uses R/$\overline{W}$, SIZ0–SIZ1, A0–A1, and $\overline{DS}$ to place information on the data bus. Any or all of the bytes (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. Concurrently, the selected device may assert the $\overline{DSACKx}$ signals.

State 3

As long as at least one of the $\overline{DSACKx}$ signals is recognized by the end of S2 (meeting the asynchronous input setup time requirement), data is latched on the next falling edge of the clock, and the cycle terminates. If $\overline{DSACKx}$ is not recognized by the start of S3, the controller inserts wait states instead of proceeding to S4 and S5. To ensure that wait states are inserted, both $\overline{DSACK0}$ and $\overline{DSACK1}$ must remain negated throughout the asynchronous input setup and hold times around the end of S2. If wait states are added, the controller continues to sample the $\overline{DSACKx}$ signals on the falling edges of the clock until one is recognized.

State 4

The controller samples the level of $\overline{CIIN}$ at the beginning of S4. At the end of S4, the controller latches the incoming data.

State 5

The controller negates $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ during S5. If more than one read cycle is required to read in the operand(s), S0–S5 are repeated for each read cycle. When finished reading, the controller holds the address, R/$\overline{W}$, and FC0–FC2 valid in preparation for the write portion of the cycle.

The external device keeps its data and $\overline{DSACKx}$ signals asserted until it detects the negation of $\overline{AS}$ or $\overline{DS}$ (whichever it detects first). The device

must remove the data and negate $\overline{\text{DSACKx}}$ within approximately one clock period after sensing the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$. $\overline{\text{DSACKx}}$ signals that remain asserted beyond this limit may be prematurely detected for the next portion of the operation.

Idle States

The controller does not assert any new control signals during the idle states, but it may internally begin the modify portion of the cycle at this time. S6–S11 are omitted if no write cycle is required. If a write cycle is required, the R/$\overline{\text{W}}$ signal remains in the read mode until S6 to prevent bus conflicts with the preceding read portion of the cycle; the data bus is not driven until S8.

State 6

The controller asserts $\overline{\text{ECS}}$ and $\overline{\text{OCS}}$ in S6 to indicate that another external cycle is beginning. The controller drives R/$\overline{\text{W}}$ low for a write cycle. $\overline{\text{CIOUT}}$ also becomes valid, indicating the state of the ACU CI bit in the access control register. Depending on the write operation to be performed, the address lines may change during S6.

State 7

In S7, the controller asserts $\overline{\text{AS}}$, indicating that the address on the address bus is valid. The controller also asserts $\overline{\text{DBEN}}$, which can be used to enable data buffers during S7. In addition, the $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) signal is negated during S7.

State 8

During S8, the controller places the data to be written onto D0–D31.

State 9

The controller asserts $\overline{\text{DS}}$ during S9 indicating that the data is stable on the data bus. As long as at least one of the $\overline{\text{DSACKx}}$ signals is recognized by the end of S8 (meeting the asynchronous input setup time requirement), the cycle terminates one clock later. If $\overline{\text{DSACKx}}$ is not recognized by the start of S9, the controller inserts wait states instead of proceeding to S10 and S11. To ensure that wait states are inserted, both $\overline{\text{DSACK0}}$ and $\overline{\text{DSACK1}}$ must remain negated throughout the asynchronous input setup and hold times around the end of S8. If wait states are added, the controller continues to sample $\overline{\text{DSACKx}}$ signals on the falling edges of the clock until one is recognized.

The selected device uses R/$\overline{\text{W}}$, $\overline{\text{DS}}$, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate section(s) of the data bus (D24–D31, D16–D23, D8–D15,

and D0–D7). SIZ0–SIZ1 and A0–A1 select the data bus sections. If it has not already done so, the device asserts $\overline{\text{DSACKx}}$ when it has successfully stored the data.

State 10

The controller issues no new control signals during S10.

State 11

The controller negates $\overline{\text{AS}}$ and $\overline{\text{DS}}$ during S11. It holds the address and data valid during S11 to provide address hold time for memory systems. R/$\overline{\text{W}}$ and FC0–FC2 also remain valid throughout S11.

If more than one write cycle is required, S6–S11 are repeated for each write cycle.

The external device keeps $\overline{\text{DSACKx}}$ asserted until it detects the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$ (whichever it detects first). The device must remove its data and negate $\overline{\text{DSACKx}}$ within approximately one clock period after sensing the negation of $\overline{\text{AS}}$ or $\overline{\text{DS}}$.

7.3.4 Synchronous Read Cycle

A synchronous read cycle is terminated differently from an asynchronous read cycle; otherwise, the cycles assert and respond to the same signals, in the same sequence. $\overline{\text{STERM}}$ rather than $\overline{\text{DSACKx}}$ is asserted by the addressed external device to terminate a synchronous read cycle. Since $\overline{\text{STERM}}$ must meet the synchronous setup and hold times with respect to all rising edges of the clock while $\overline{\text{AS}}$ is asserted, it does not need to be synchronized by the controller. Only devices with 32-bit ports may assert $\overline{\text{STERM}}$. $\overline{\text{STERM}}$ is also used with the $\overline{\text{CBREQ}}$ and $\overline{\text{CBACK}}$ signals during burst mode operation. It provides a two-clock (minimum) bus cycle for 32-bit ports and single-clock (minimum) burst accesses, although wait states can be inserted for these cycles as well. Therefore, a synchronous cycle terminated with $\overline{\text{STERM}}$ with one wait cycle is a three-clock bus cycle. However, note that $\overline{\text{STERM}}$ is asserted one-half clock later than $\overline{\text{DSACKx}}$ would be for a similar asynchronous cycle with zero wait cycles (also three clocks). Thus, if dynamic bus sizing is not needed, $\overline{\text{STERM}}$ can be used to provide more decision time in an external cache design than is available with $\overline{\text{DSACKx}}$ for three-clock accesses.

Figure 7-31 is a flowchart of a synchronous long-word read cycle. Byte and word operations are similar. Figure 7-32 is a functional timing diagram of a synchronous long-word read cycle.

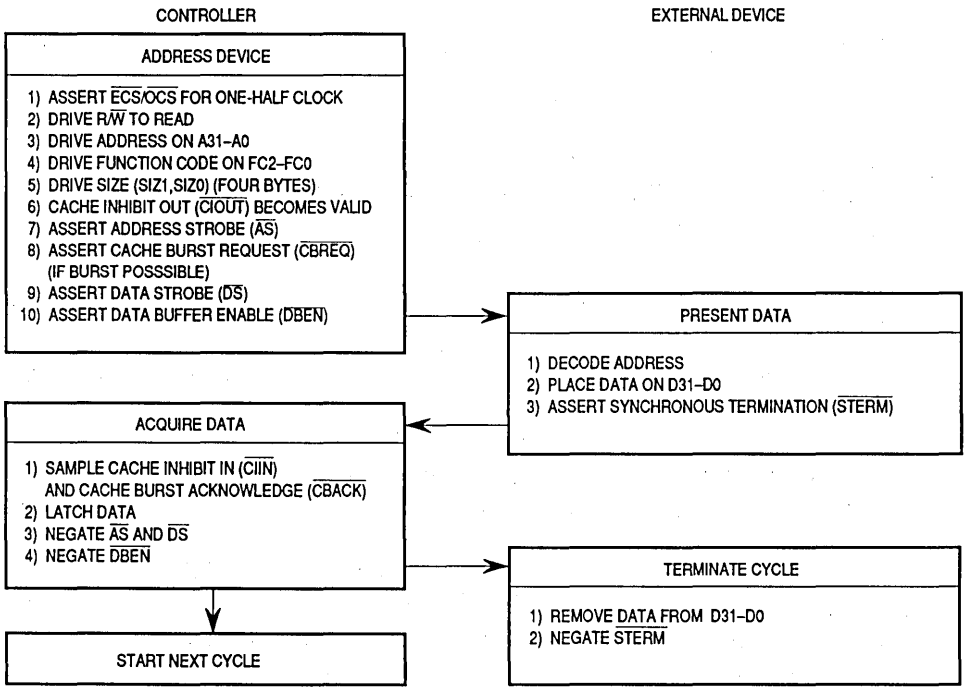


Figure 7-31. Synchronous Long-Word Read Cycle Flowchart — No Burst Allowed

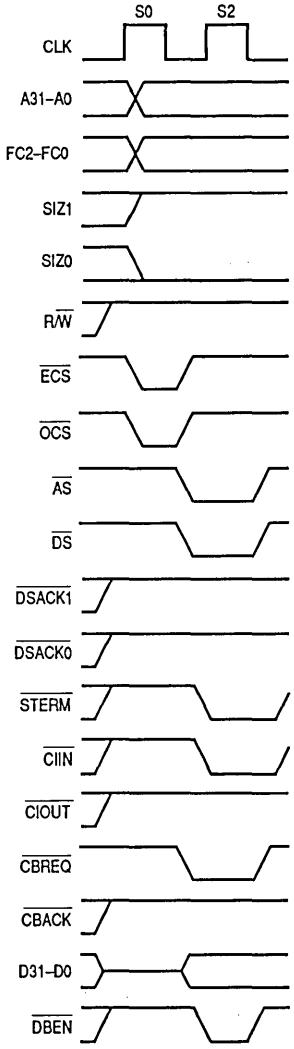


Figure 7-32. Synchronous Read with CIIN Asserted and CBACK Negated

State 0

The read cycle starts with S0. The controller drives $\overline{ECS}$ low, indicating the beginning of an external cycle. When the cycle is the first cycle of a read operand operation, $\overline{OCS}$ is driven low at the same time. During S0, the controller places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The controller drives R/$\overline{W}$ high for a read cycle and drives $\overline{DBEN}$ inactive to disable the data buffers. SIZ1–SIZ0 become valid, indicating the number of bytes to be transferred. $\overline{CIOUT}$ also becomes valid, indicating the state of the ACU CI bit in the access control register.

State 1

One-half clock later in S1, the controller asserts $\overline{AS}$, indicating that the address on the address bus is valid. The controller also asserts $\overline{DS}$ during S1. If the burst mode is enabled for the appropriate on-chip cache and all four long words of the cache entry are invalid, (i.e., four long words can be read in), $\overline{CBREQ}$ is asserted. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

7

State 2

The selected device uses R/$\overline{W}$, SIZ0–SIZ1, A0–A1, and $\overline{CIOUT}$ to place its information on the data bus. Any or all of the byte sections of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. During S2, the controller drives $\overline{DBEN}$ active to enable external data buffers. In systems that use two-clock synchronous bus cycles, the timing of $\overline{DBEN}$ may prevent its use. At the beginning of S2, the controller samples the level of $\overline{STERM}$. If $\overline{STERM}$ is recognized, the controller latches the incoming data at the end of S2. If the selected data is not to be cached for the current cycle or if the device cannot supply 32 bits, $\overline{CIIN}$ must be asserted at the same time as $\overline{STERM}$. In addition, the state of $\overline{CBACK}$ is latched when $\overline{STERM}$ is recognized.

Since $\overline{CIIN}$, $\overline{CBACK}$, and $\overline{STERM}$ are synchronous signals, they must meet the synchronous input setup and hold times for all rising edges of the clock while $\overline{AS}$ is asserted. If $\overline{STERM}$ is negated at the beginning of S2, wait states are inserted after S2, and $\overline{STERM}$ is sampled on every rising edge thereafter until it is recognized. Once $\overline{STERM}$ is recognized, data is latched on the next falling edge of the clock (corresponding to the beginning of S3).

State 3

The controller negates $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ during S3. It holds the address valid during S3 to simplify memory interfaces. R/$\overline{W}$, SIZ0–SIZ1, and FC0–FC2 also remain valid throughout S3.

The external device must keep its data asserted throughout the synchronous hold time for data from the beginning of S3. The device must remove its data within one clock after asserting $\overline{\text{STERM}}$ and negate $\overline{\text{STERM}}$ within two clocks after asserting $\overline{\text{STERM}}$; otherwise, the controller may inadvertently use $\overline{\text{STERM}}$ for the next bus cycle.

7.3.5 Synchronous Write Cycle

A synchronous write cycle is terminated differently from an asynchronous write cycle and the data strobe may not be useful. Otherwise, the cycles assert and respond to the same signal, in the same sequence. $\overline{\text{STERM}}$ is asserted by the external device to terminate a synchronous write cycle. The discussion of $\overline{\text{STERM}}$ in the preceding section applies to write cycles as well as to read cycles.

$\overline{\text{DS}}$ is not asserted for two-clock synchronous write cycles; therefore, the clock (CLK) may be used as the timing signal for latching the data. In addition, there is no time from the latest assertion of $\overline{\text{AS}}$ and the required assertion of $\overline{\text{STERM}}$ for any two-clock synchronous bus cycle. The system must qualify a memory write with the assertion of $\overline{\text{AS}}$ to ensure that the write is not aborted by internal conditions within the MC68EC030.

Figure 7-33 is a flowchart of a synchronous write cycle. Figure 7-34 is a functional timing diagram of this operation with wait states.

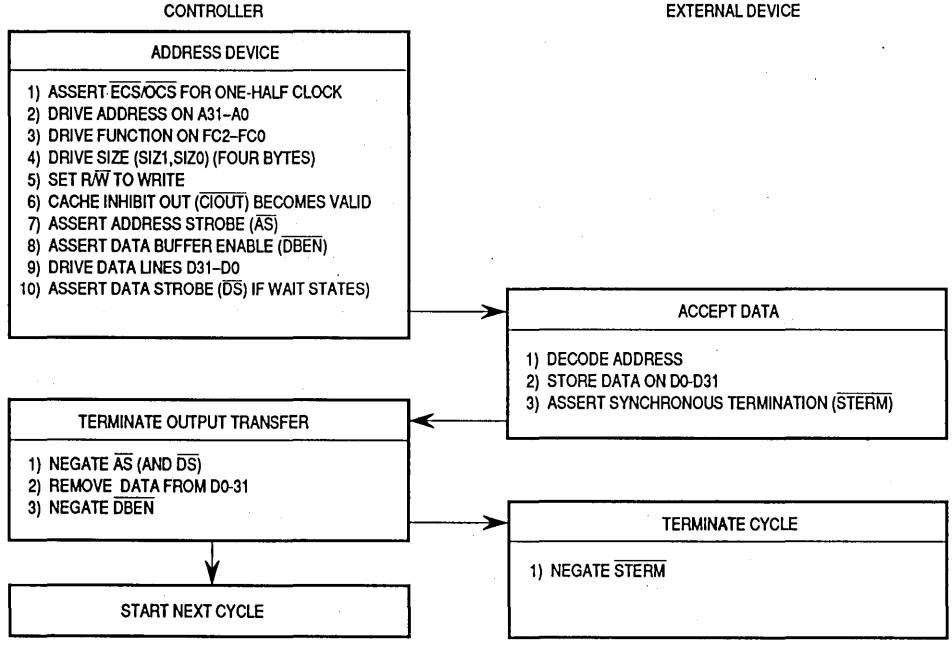


Figure 7-33. Synchronous Write Cycle Flowchart

State 0

The write cycle starts with S0. The controller drives $\overline{ECS}$ low, indicating the beginning of an external cycle. When the cycle is the first cycle of a write operation, $\overline{OCS}$ is driven low at the same time. During S0, the controller places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The controller drives R/$\overline{W}$ low for a write cycle. SIZ0–SIZ1 become valid, indicating the number of bytes to be transferred. $\overline{CIOUT}$ also becomes valid, indicating the state of the ACU CI bit in the access control register.

State 1

One-half clock later in S1, the controller asserts $\overline{AS}$, indicating that the address on the address bus is valid. The controller also asserts $\overline{DBEN}$ during S1, which may be used to enable the external data buffers. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

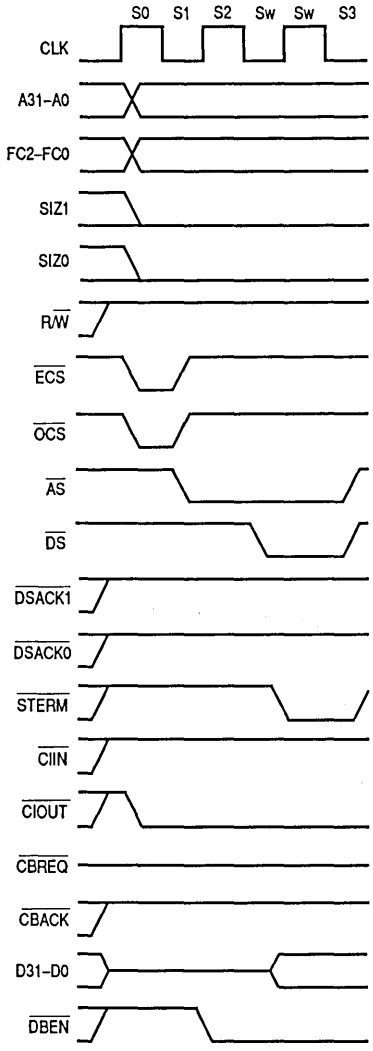


Figure 7-34. Synchronous Write Cycle with Wait States — CIOUT Asserted

State 2

During S2, the controller places the data to be written onto D0–D31. The selected device uses R/$\overline{W}$, CLK, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate section(s) of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7). SIZ0–SIZ1 and A0–A1 select the data bus sections. The device asserts $\overline{STERM}$ when it has successfully stored the data. If the device does not assert $\overline{STERM}$ by the rising edge of S2, the controller inserts wait states until it is recognized. The controller asserts $\overline{DS}$ at the end of S2 if wait states are inserted. For zero-wait-state synchronous write cycles, $\overline{DS}$ is not asserted.

State 3

The controller negates $\overline{AS}$ (and $\overline{DS}$, if necessary) during S3. It holds the address and data valid during S3 to simplify memory interfaces. R/$\overline{W}$, SIZ0–SIZ1, FC0–FC2, and $\overline{DBEN}$ also remain valid throughout S3.

The addressed device must negate $\overline{STERM}$ within two clock periods after asserting it, or the controller may use $\overline{STERM}$ for the next bus cycle.

7

7.3.6 Synchronous Read-Modify-Write Cycle

A synchronous read-modify-write operation differs from an asynchronous read-modify-write operation only in the terminating signal of the read and write cycles and in the use of CLK instead of $\overline{DS}$ latching data in the write cycle. Like the asynchronous operation, the synchronous read-modify-write operation is indivisible. Although the operation is synchronous, the burst mode is never used during read-modify-write cycles.

Figure 7-35 is a flowchart of the synchronous read-modify-write operation. Timing for the cycle is shown in Figure 7-36.

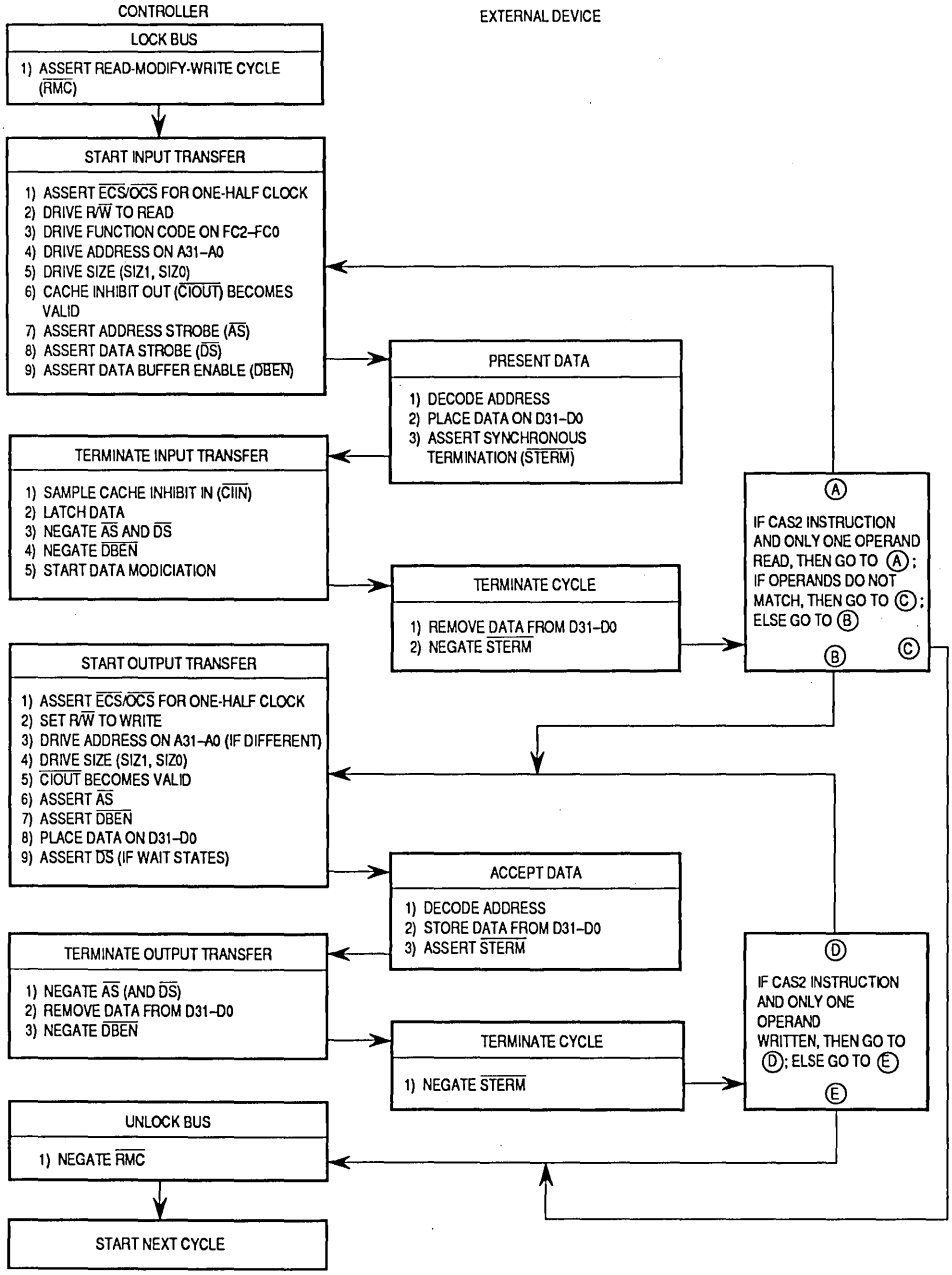


Figure 7-35. Synchronous Read-Modify-Write Cycle Flowchart

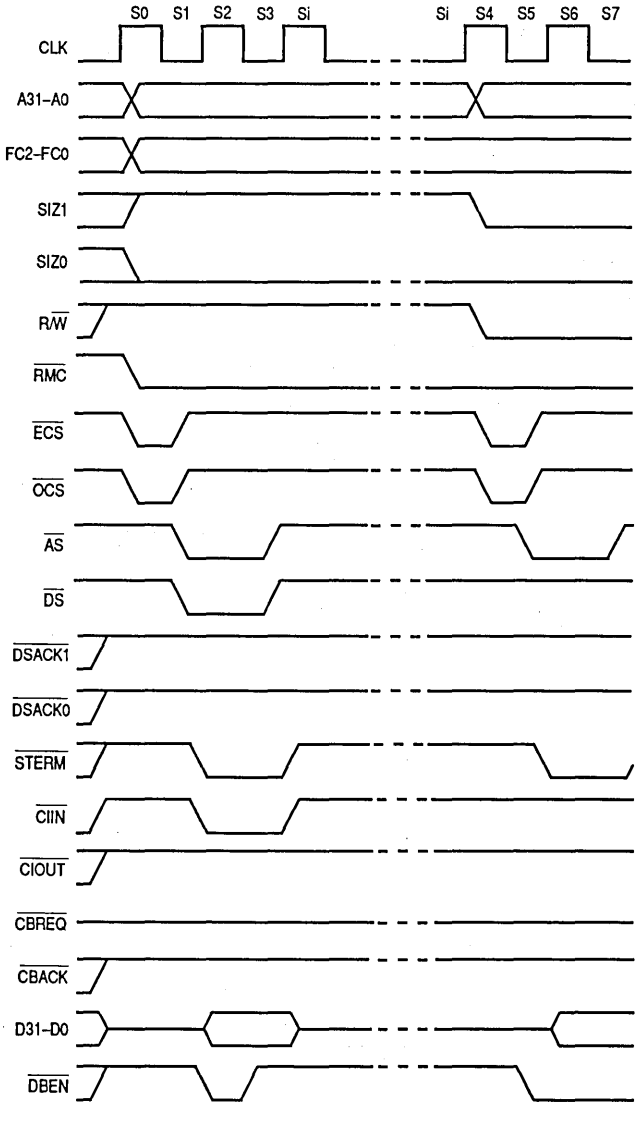


Figure 7-36. Synchronous Read-Modify-Write Cycle Timing — $\overline{\text{CIIN}}$ Asserted

State 0

The controller asserts $\overline{ECS}$ and $\overline{OCS}$ in S0 to indicate the beginning of an external operand cycle. The controller also asserts $\overline{RMC}$ in S0 to identify a read-modify-write cycle. The controller places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the operation. SIZ0–SIZ1 become valid in S0 to indicate the operand size. The controller drives R/$\overline{W}$ high for a read cycle and sets $\overline{CIOUT}$ to the value of the ACU CI bit in the access control register. The controller drives $\overline{DBEN}$ inactive to disable the data buffers.

State 1

One-half clock later in S1, the controller asserts $\overline{AS}$, indicating that the address on the address bus is valid. The controller also asserts $\overline{DS}$ during S1. In addition, the $\overline{ECS}$ (and $\overline{OCS}$, if asserted) signal is negated during S1.

State 2

The selected device uses R/$\overline{W}$, SIZ0–SIZ1, A0–A1, and $\overline{CIOUT}$ to place its information on the data bus. Any or all of the byte sections (D24–D31, D16–D23, D8–D15, and D0–D7) are selected by SIZ0–SIZ1 and A0–A1. During S2, the controller drives $\overline{DBEN}$ active to enable external data buffers. In systems that use two-clock synchronous bus cycles, the timing of $\overline{DBEN}$ may prevent its use. At the beginning of S2, the controller samples the level of $\overline{STERM}$. If $\overline{STERM}$ is recognized, the controller latches the incoming data. If the selected data is not to be cached for the current cycle or if the device cannot supply 32 bits, $\overline{CIIN}$ must be asserted at the same time as $\overline{STERM}$.

Since $\overline{CIIN}$ and $\overline{STERM}$ are synchronous signals, they must meet the synchronous input setup and hold times for all rising edges of the clock while $\overline{AS}$ is asserted. If $\overline{STERM}$ is negated at the beginning of S2, wait states are inserted after S2, and $\overline{STERM}$ is sampled on every rising edge thereafter until it is recognized. Once $\overline{STERM}$ is recognized, data is latched on the next falling edge of the clock (corresponding to the beginning of S3).

State 3

The controller negates $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ during S3. If more than one read cycle is required to read in the operand(s), S0–S3 are repeated accordingly. When finished with the read cycle, the controller holds the address, R/$\overline{W}$, and FC0–FC2 valid in preparation for the write portion of the cycle.

The external device must keep its data asserted throughout the synchronous hold time for data from the beginning of S3. The device must remove the data within one-clock cycle after asserting $\overline{STERM}$ to avoid bus con-

tention. It must also negate $\overline{\text{STERM}}$ within two clocks after asserting $\overline{\text{STERM}}$; otherwise, the controller may inadvertently use $\overline{\text{STERM}}$ for the next bus cycle.

Idle States

The controller does not assert any new control signals during the idle states, but it may begin the modify portion of the cycle at this time. The R/$\overline{\text{W}}$ signal remains in the read mode until S4 to prevent bus conflicts with the preceding read portion of the cycle; the data bus is not driven until S6.

State 4

The controller asserts $\overline{\text{ECS}}$ and $\overline{\text{OCS}}$ in S4 to indicate that an external cycle is beginning. The controller drives R/$\overline{\text{W}}$ low for a write cycle. $\overline{\text{CIOUT}}$ also becomes valid, indicating the state of the ACU CI bit in the access control register. Depending on the write operation to be performed, the address lines may change during S4.

State 5

In state 5 (S5), the controller asserts $\overline{\text{AS}}$ to indicate that the address on the address bus is valid. The controller also asserts $\overline{\text{DBEN}}$ during S5, which can be used to enable external data buffers.

State 6

During S6, the controller places the data to be written onto the D0–D31.

The selected device uses R/$\overline{\text{W}}$, CLK, SIZ0–SIZ1, and A0–A1 to latch data from the appropriate byte(s) of the data bus (D24–D31, D16–D23, D8–D15, and D0–D7). SIZ0–SIZ1 and A0–A1 select the data bus sections. The device asserts $\overline{\text{STERM}}$ when it has successfully stored the data. If the device does not assert $\overline{\text{STERM}}$ by the rising edge of S6, the controller inserts wait states until it is recognized. The controller asserts $\overline{\text{DS}}$ at the end of S6 if wait states are inserted. Note that for zero-wait-state synchronous write cycles, $\overline{\text{DS}}$ is not asserted.

State 7

The controller negates $\overline{\text{AS}}$ (and $\overline{\text{DS}}$, if necessary) during S7. It holds the address and data valid during S7 to simplify memory interfaces. R/$\overline{\text{W}}$ and FC0–FC2 also remain valid throughout S7.

If more than one write cycle is required, S8–S11 are repeated for each write cycle.

The external device must negate $\overline{\text{STERM}}$ within two clock periods after asserting it, or the controller may inadvertently use $\overline{\text{STERM}}$ for the next bus cycle.

7.3.7 Burst Operation Cycles

The MC68EC030 supports a burst mode for filling the on-chip instruction and data caches.

The MC68EC030 provides a set of handshake control signals for the burst mode. When a miss occurs in one of the caches, the MC68EC030 initiates a bus cycle to obtain the required data or instruction stream fetch. If the data or instruction can be cached, the MC68EC030 attempts to fill a cache entry. Depending on the alignment for a data access, the MC68EC030 may attempt to fill two cache entries. The controller may also assert $\overline{\text{CBREQ}}$ to request a burst fill operation. That is, the controller can fill additional entries in the line. The MC68EC030 allows a burst of as many as four long words.

The mechanism that asserts the $\overline{\text{CBREQ}}$ signal for burstable cache entries is enabled by the data burst enable (DBE) and instruction burst enable (IBE) bits of the cache control register (CACR) for the data and instruction caches, respectively. Either of the following conditions cause the MC68EC030 to initiate a cache burst request (and assert $\overline{\text{CBREQ}}$) for a cacheable read cycle:

- The address and function code signals of the current instruction or data fetch do not match the indexed tag field in the respective instruction or data cache.
- All four long words corresponding to the indexed tag in the appropriate cache are marked invalid.

However, the MC68EC030 does not assert $\overline{\text{CBREQ}}$ during the first portion of a misaligned access if the remainder of the access does not correspond to the same cache line. Refer to **6.1.3.1 SINGLE ENTRY MODE** for details.

If the appropriate cache is not enabled or if the cache freeze bit for the cache is set, the controller does not assert $\overline{\text{CBREQ}}$. $\overline{\text{CBREQ}}$ is not asserted during the read or write cycles of any read-modify-write operation.

The MC68EC030 allows burst filling only from 32-bit ports that terminate bus cycles with $\overline{\text{STERM}}$ and respond to $\overline{\text{CBREQ}}$ by asserting $\overline{\text{CBACK}}$. When the MC68EC030 recognizes $\overline{\text{STERM}}$ and $\overline{\text{CBACK}}$ and it has asserted $\overline{\text{CBREQ}}$, it maintains $\overline{\text{AS}}$, $\overline{\text{DS}}$, R/$\overline{\text{W}}$, A0–A31, FC0–FC2, SIZ0–SIZ1 in their current state throughout the burst operation. The controller continues to accept data on every clock during which $\overline{\text{STERM}}$ is asserted until the burst is complete or an abnormal termination occurs.

$\overline{\text{CBACK}}$ indicates that the addressed device can respond to a cache burst request by supplying one more long word of data in the burst mode. It can

be asserted independently of the CBREQ signal, and burst mode is only initiated if both of these signals are asserted for a synchronous cycle. If the MC68EC030 executes a full burst operation and fetches four long words, CBREQ is negated after STERM is asserted for the third cycle, indicating that the MC68EC030 only requests one more long word (the fourth cycle). CBACK can then be negated, and the MC68EC030 latches the data for the fourth cycle and completes the cache line fill.

The following conditions can abort a burst fill:

- CIIN asserted,
- BERR asserted, or
- CBACK negated prematurely.

The processing of a bus error during a burst fill operation is described in **7.5.1 Bus Errors.**

For the purposes of halting the controller or arbitrating the bus away from the controller with BR, a burst operation is a single cycle since AS remains asserted during the entire operation. If the HALT signal is asserted during a burst operation, the controller halts at the end of the operation. Refer to **7.5.3 Halt Operation** for more information about the halt operation. An alternate bus master requesting the bus with BR may become bus master at the end of the operation provided BR is asserted early enough to be internally synchronized before another controller cycle begins. Refer to **7.7 BUS ARBITRATION** for more information about bus arbitration.

The simultaneous assertion of BERR and HALT during a bus cycle normally indicates that the cycle should be retried. However, during the second, third, or fourth cycle of a burst operation, this signal combination indicates a bus error condition, which aborts the burst operation. In addition, the controller remains in the halted state until HALT is negated. For information about bus error processing, refer to **7.5.1. Bus Errors.**

Figure 7-37 is a flowchart of the burst operation. The following timing diagrams show various burst operations. Figure 7-38 shows burst operations for long-word requests with two wait states inserted in the first access and one wait cycle inserted in the subsequent accesses. Figure 7-39 shows a burst operation that fails to complete normally due to CBACK negating prematurely. Figure 7-40 shows a burst operation that is deferred because the entire operand does not correspond to the same cache line. Figure 7-41 shows a burst operation aborted by CIIN. Because CBACK corresponds to the *next* cycle, three long words are transferred even though CBACK is only asserted for two clock periods.

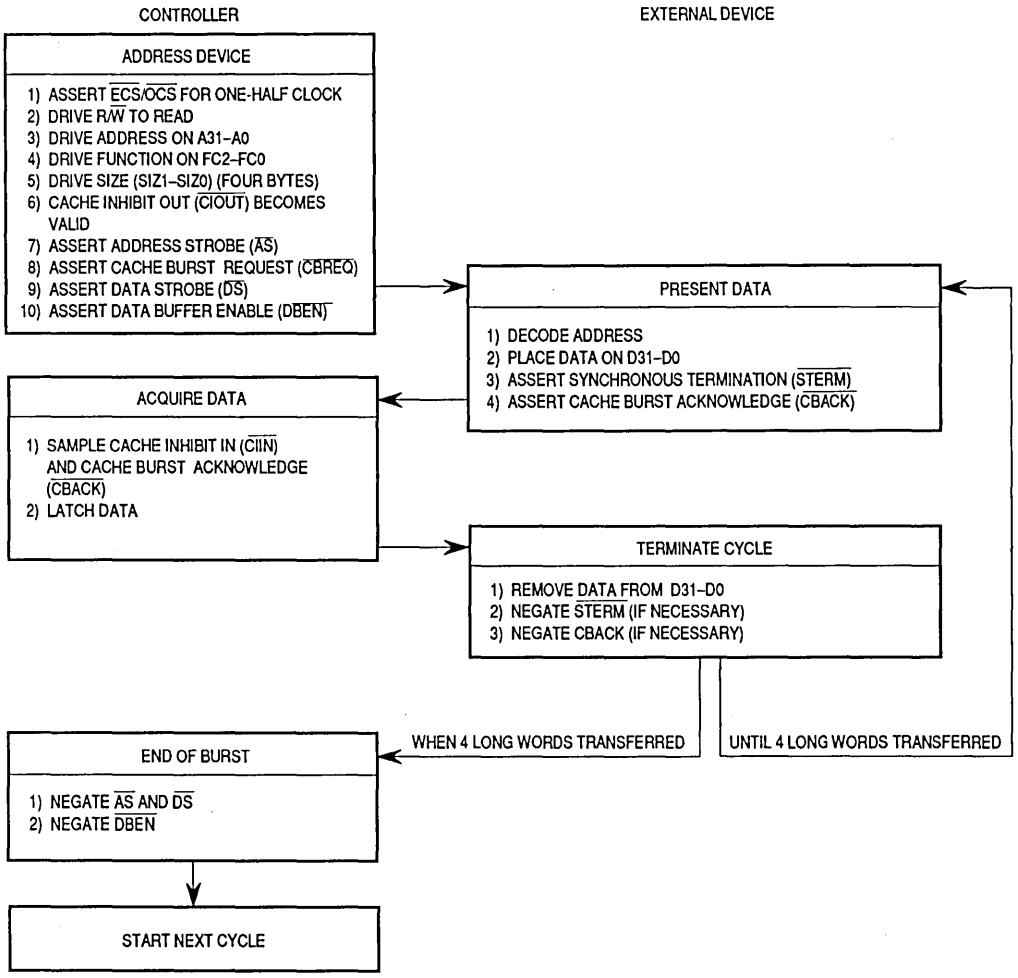


Figure 7-37. Burst Operation Flowchart — Four Long Words Transferred

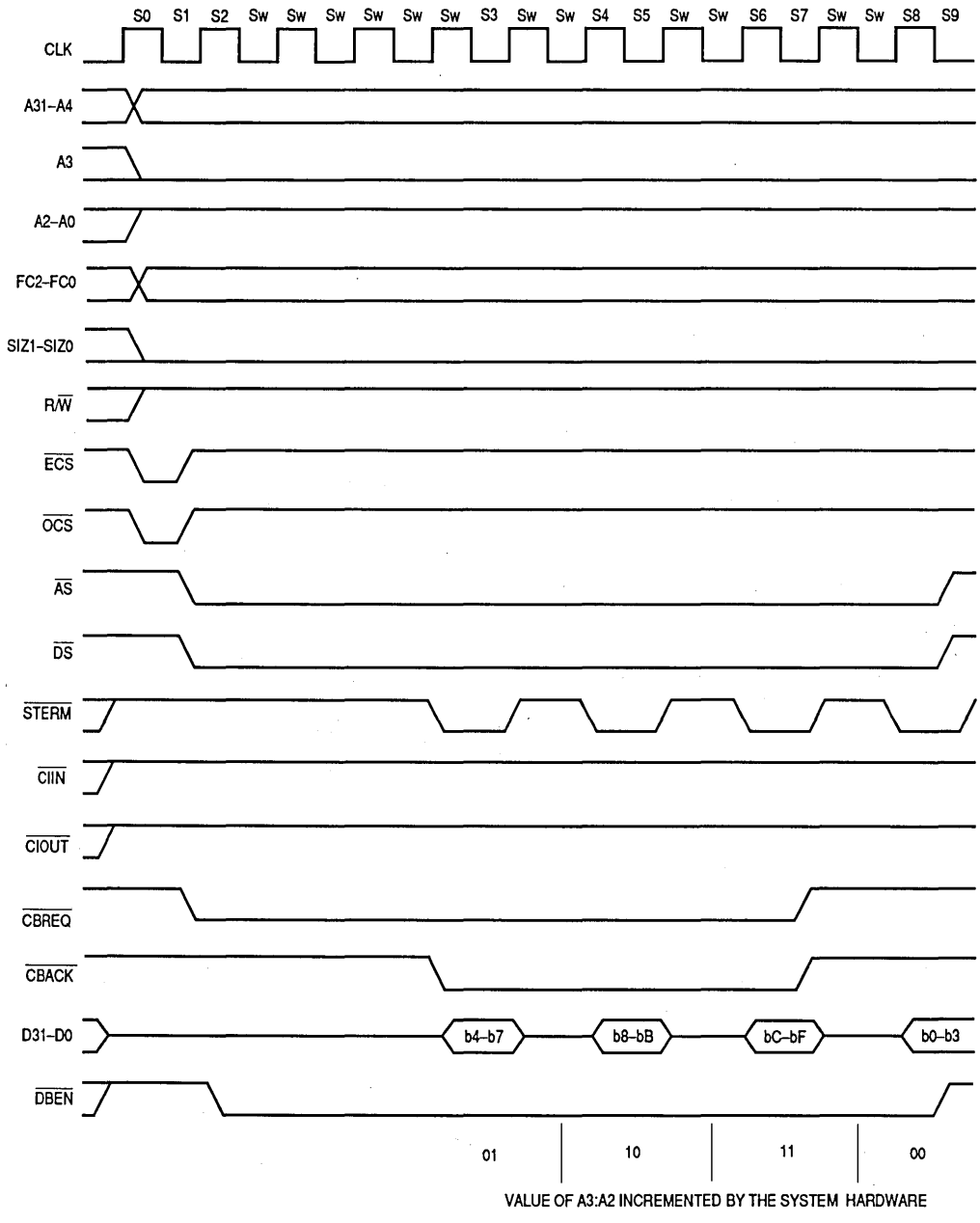
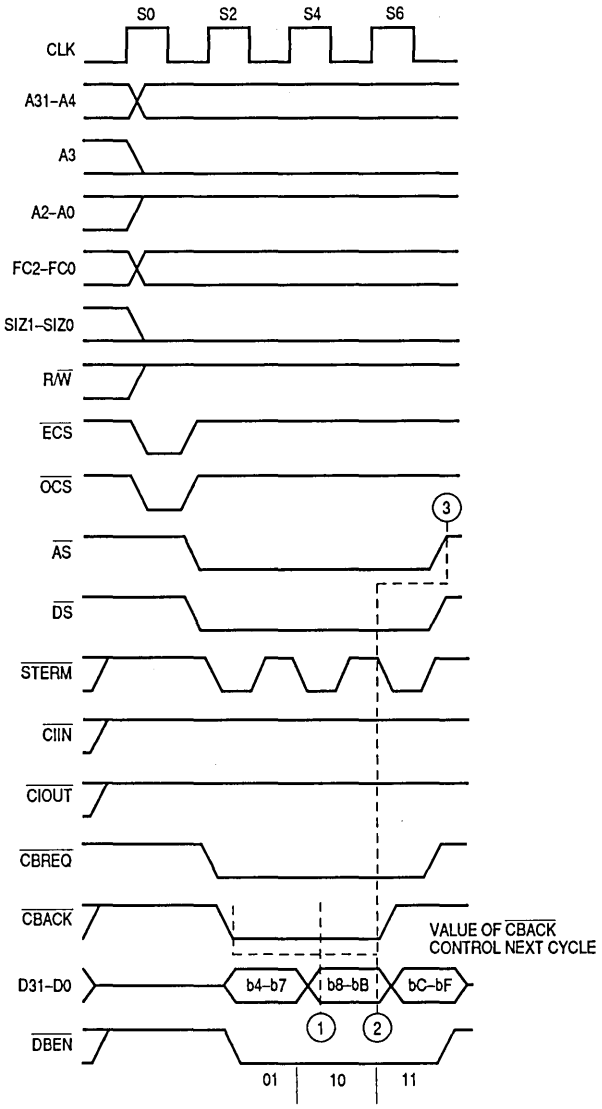


Figure 7-38. Long-Word Operand Request from $07 with Burst Request and Wait Cycle



NOTES:
1. Assertion of $\overline{\text{CBACK}}$ causes data to be placed on D31–D0.
2. Continued assertion of $\overline{\text{CBACK}}$ causes data to be placed on D31–D0.
3. Negation of $\overline{\text{CBACK}}$ causes $\overline{\text{AS}}$ to be negated.

Figure 7-39. Long-Word Operand Request from $07 with Burst Request — $\overline{\text{CBACK}}$ Negated Early

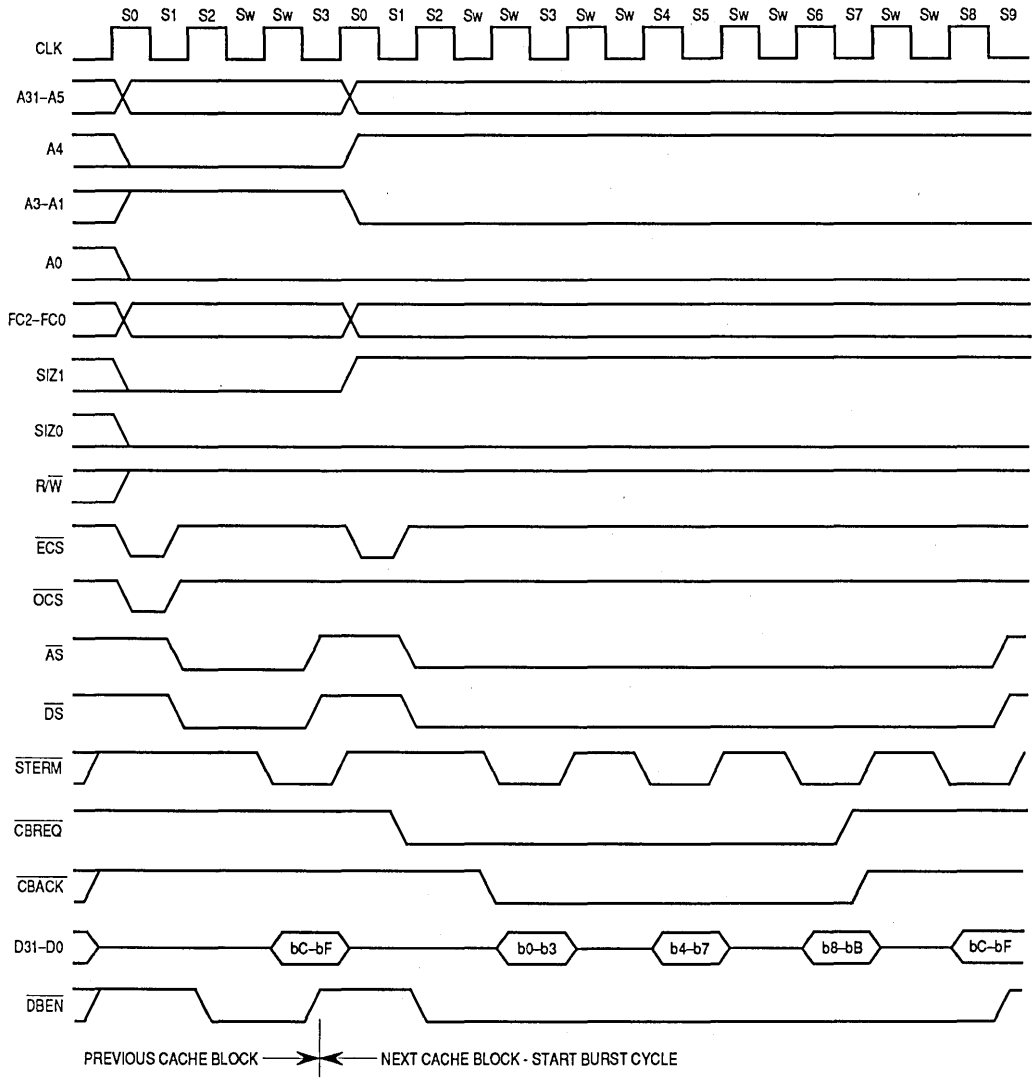


Figure 7-40. Long-Word Operand Request from $0E — Burst Fill Deferred

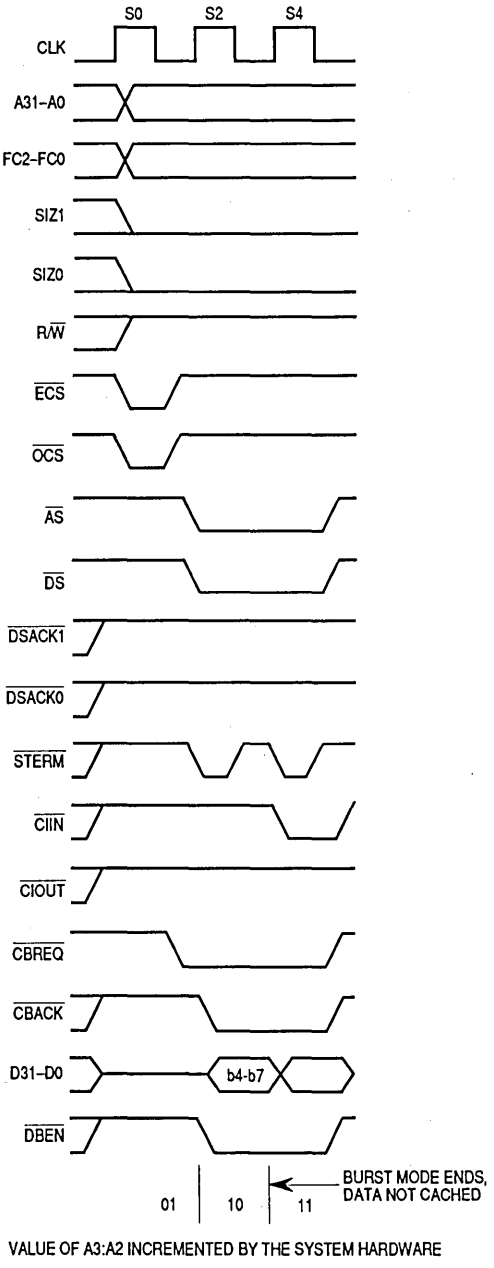


Figure 7-41. Long-Word Operand Request from $07 with Burst Request — $\overline{\text{CBACK}}$ and $\overline{\text{CIIN}}$ Asserted

The burst operation sequence begins with states S0–S3, which are very similar to those states for a synchronous read cycle except that $\overline{\text{CBREQ}}$ is asserted. S4–S9 perform the final three reads for a complete burst operation.

State 0

The burst operation starts with S0. The controller drives $\overline{\text{ECS}}$ low, indicating the beginning of an external cycle. When the cycle is the first cycle of a read operation, $\overline{\text{OCS}}$ is driven low at the same time. During S0, the controller places a valid address on A0–A31 and valid function codes on FC0–FC2. The function codes select the address space for the cycle. The controller drives R/$\overline{\text{W}}$ high, indicating a read cycle, and drives $\overline{\text{DBEN}}$ inactive to disable the data buffers. SIZ0–SIZ1 become valid, indicating the number of operand bytes to be transferred. $\overline{\text{CIOUT}}$ also becomes valid, indicating the state of the ACU CI bit in the access control register.

State 1

One-half clock later in S1, the controller asserts $\overline{\text{AS}}$ to indicate that the address on the address bus is valid. The controller also asserts $\overline{\text{DS}}$ during S1. $\overline{\text{CBREQ}}$ is also asserted, indicating that the MC68EC030 can perform a burst operation into one of its caches and can read in four long words. In addition, $\overline{\text{ECS}}$ (and $\overline{\text{OCS}}$, if asserted) is negated during S1.

State 2

The selected device uses R/$\overline{\text{W}}$, SIZ0–SIZ1, A0–A1, and $\overline{\text{CIOUT}}$ to place the data on the data bus. (The first cycle must supply the long word at the corresponding long-word boundary.) All of the byte sections (D24–D31, D16–D23, D8–D15, and D0–D7) of the data bus must be driven since the burst operation latches 32 bits on every cycle. During S2, the controller drives $\overline{\text{DBEN}}$ active to enable external data buffers. In systems that use two-clock synchronous bus cycles, the timing of $\overline{\text{DBEN}}$ may prevent its use. At the beginning of S2, the controller tests the level of $\overline{\text{STERM}}$. If $\overline{\text{STERM}}$ is recognized, the controller latches the incoming data at the end of S2. For the burst operation to proceed, $\overline{\text{CBACK}}$ must be asserted when $\overline{\text{STERM}}$ is recognized. If the data for the current cycle is not to be cached, $\overline{\text{CIIN}}$ must be asserted at the same time as $\overline{\text{STERM}}$. The assertion of $\overline{\text{CIIN}}$ also has the effect of aborting the burst operation.

Since $\overline{\text{CIIN}}$, $\overline{\text{CBACK}}$, and $\overline{\text{STERM}}$ are synchronous signals, they must meet the synchronous input setup and hold times for all rising edges of the clock while $\overline{\text{AS}}$ is asserted. If $\overline{\text{STERM}}$ is negated at the beginning of S2, wait states are inserted after S2, and $\overline{\text{STERM}}$ is sampled on every rising edge of the clock thereafter until it is recognized. Once $\overline{\text{STERM}}$ is recognized, data is latched on the next falling edge of the clock (corresponding to the beginning of S3).

State 3

The controller maintains $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ asserted during S3. It also holds the address valid during S3 for continuation of the burst. R/$\overline{W}$, SIZ0–SIZ1, and FC0–FC2 also remain valid throughout S3.

The external device must keep the data driven throughout the synchronous hold time for data from the beginning of S3. The device must negate $\overline{STERM}$ within one clock after asserting $\overline{STERM}$; otherwise, the controller may inadvertently use $\overline{STERM}$ prematurely for the next burst access. $\overline{STERM}$ need not be negated if subsequent accesses do not require wait cycles.

State 4

At the beginning of S4, the controller tests the level of $\overline{STERM}$. This state signifies the beginning of burst mode, and the remaining states correspond to burst fill cycles. If $\overline{STERM}$ is recognized, the controller latches the incoming data at the end of S4. This data corresponds to the second long word of the burst. If $\overline{STERM}$ is negated at the beginning of S4, wait states are inserted instead of S4 and S5, and $\overline{STERM}$ is sampled on every rising edge of the clock thereafter until it is recognized. As for synchronous cycles, the states of $\overline{CBACK}$ and $\overline{CIIN}$ are latched at the time $\overline{STERM}$ is recognized. The assertion of $\overline{CBACK}$ at this time indicates that the burst operation should continue, and the assertion of $\overline{CIIN}$ indicates that the data latched at the end of S4 should not be cached and that the burst should abort.

State 5

The controller maintains all the signals on the bus driven throughout S5 for continuation of the burst. The same hold times for $\overline{STERM}$ and data described for S3 apply here.

State 6

This state is identical to S4 except that once $\overline{STERM}$ is recognized, the third long word of data for the burst is latched at the end of S6.

State 7

During this state, the controller negates $\overline{CBREQ}$, and the memory device may negate $\overline{CBACK}$. Aside from this, all other bus signals driven by the controller remain driven. The same hold times for $\overline{STERM}$ and data described for S3 apply here.

State 8

This state is identical to S4 except that $\overline{CBREQ}$ is negated, indicating that the controller cannot continue to accept more data after this. The data latched at the end of S8 corresponds to the fourth long word of the burst.

State 9

The controller negates $\overline{AS}$, $\overline{DS}$, and $\overline{DBEN}$ during S9. It holds the address, R/$\overline{W}$, SIZ0–SIZ1, and FC0–FC2 valid throughout S9. The same hold times for data described for S3 apply here.

Note that the address bus of the MC68EC030 remains driven to a constant value for the duration of a burst transfer operation (including the first transfer before burst mode is entered). If an external memory system requires incrementing of the long-word base address to supply successive long words of information, this function must be performed by external hardware. Additionally, in the case of burst transfers that cross a 16-byte boundary (i.e., the first long word transferred is not located at A3/A2 = 00), the external hardware must correctly control the continuation or termination of the burst transfer as desired. The burst may be terminated by negating $\overline{CBACK}$ during the transfer of the most significant long word of the 16-byte image (A3/A2 = 11) or may be continued (with $\overline{CBACK}$ asserted) by providing the long word located at A3/A2 = 00 (i.e., the count sequence wraps back to zero and continues as necessary). The MC68EC030 caches assume the higher order address lines (A4–A31) remain unchanged as the long-word accesses wrap back around to A3/A2 = 00.

7.4 CPU SPACE CYCLES

FC0–FC2 select user and supervisor program and data areas as listed in Table 4-1. The area selected by FC0–FC2 = $7 is classified as the CPU space. The interrupt acknowledge, breakpoint acknowledge, and coprocessor communication cycles described in the following sections utilize CPU space.

The CPU space type is encoded on A16–A19 during a CPU space operation and indicates the function that the controller is performing. On the MC68EC030, three of the encodings are implemented as shown in Figure 7-42. All unused values are reserved by Motorola for future additional CPU space types.

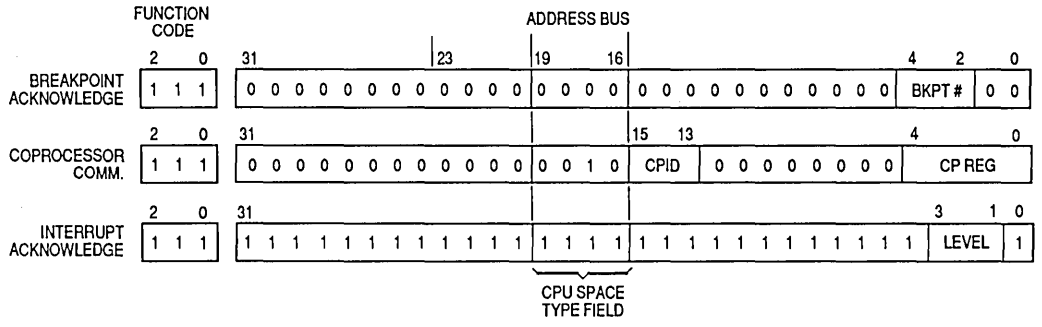


Figure 7-42. MC68EC030 CPU Space Address Encoding

7.4.1 Interrupt Acknowledge Bus Cycles

When a peripheral device signals the controller (with the $\overline{IPL0}$–$\overline{IPL2}$ signals) that the device requires service, and the internally synchronized value on these signals indicates a higher priority than the interrupt mask in the status register (or that a transition has occurred in the case of a level 7 interrupt), the controller makes the interrupt a pending interrupt. Refer to **8.1.9 Interrupt Exceptions** for details on the recognition of interrupts.

The MC68EC030 takes an interrupt exception for a pending interrupt within one instruction boundary (after processing any other pending exception with a higher priority). The following paragraphs describe the various kinds of interrupt acknowledge bus cycles that can be executed as part of interrupt exception processing.

7.4.1.1 INTERRUPT ACKNOWLEDGE CYCLE — TERMINATED NORMALLY. When the MC68EC030 processes an interrupt exception, it performs an interrupt acknowledge cycle to obtain the number of the vector that contains the starting location of the interrupt service routine.

Some interrupting devices have programmable vector registers that contain the interrupt vectors for the routines they use. The following paragraphs describe the interrupt acknowledge cycle for these devices. Other interrupting conditions or devices cannot supply a vector number and use the autovector cycle described in **7.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE.**

The interrupt acknowledge cycle is a read cycle. It differs from the asynchronous read cycle described in **7.3.1 Asynchronous Read Cycle** or the syn-

chronous read cycle described in **7.3.4 Synchronous Read Cycle** in that it accesses the CPU address space. Specifically, the differences are:

1. FC0–FC2 are set to seven (FC0/FC1/FC2 = 111) for CPU address space.
2. A1, A2, and A3 are set to the interrupt request level (the inverted values of $\overline{IPL0}$, $\overline{IPL1}$, and $\overline{IPL2}$, respectively).
3. The CPU space type field (A16–A19) is set to $F, the interrupt acknowledge code.
4. A20–A31, A4–A15, and A0 are set to one.

The responding device places the vector number on the data bus during the interrupt acknowledge cycle. Beyond this, the cycle is terminated normally with either $\overline{STERM}$ or $\overline{DSACKx}$. Figure 7-43 is the flowchart of the interrupt acknowledge cycle.

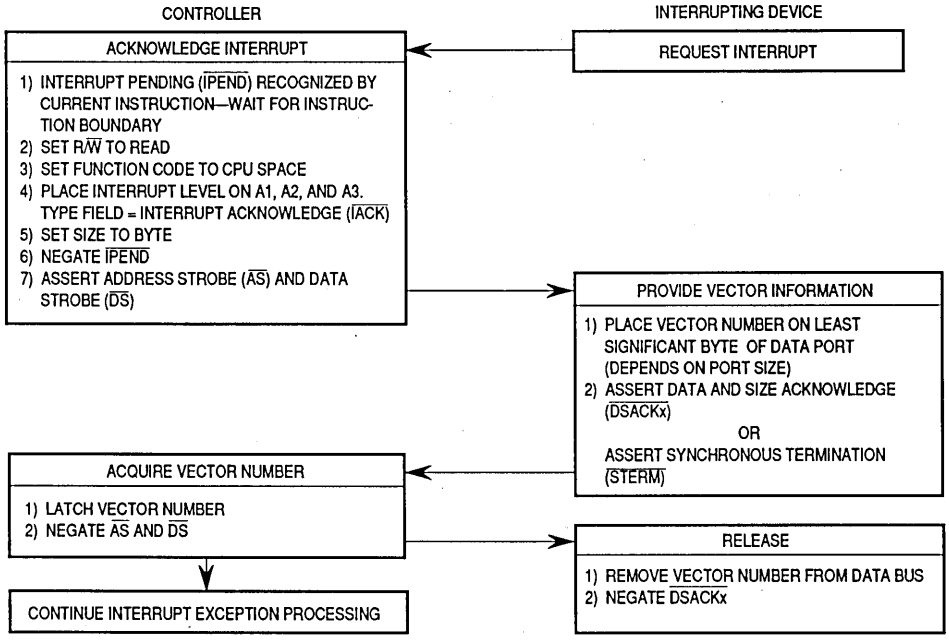


Figure 7-43. Interrupt Acknowledge Cycle Flowchart

Figure 7-44 shows the timing for an interrupt acknowledge cycle terminated with $\overline{DSACKx}$.

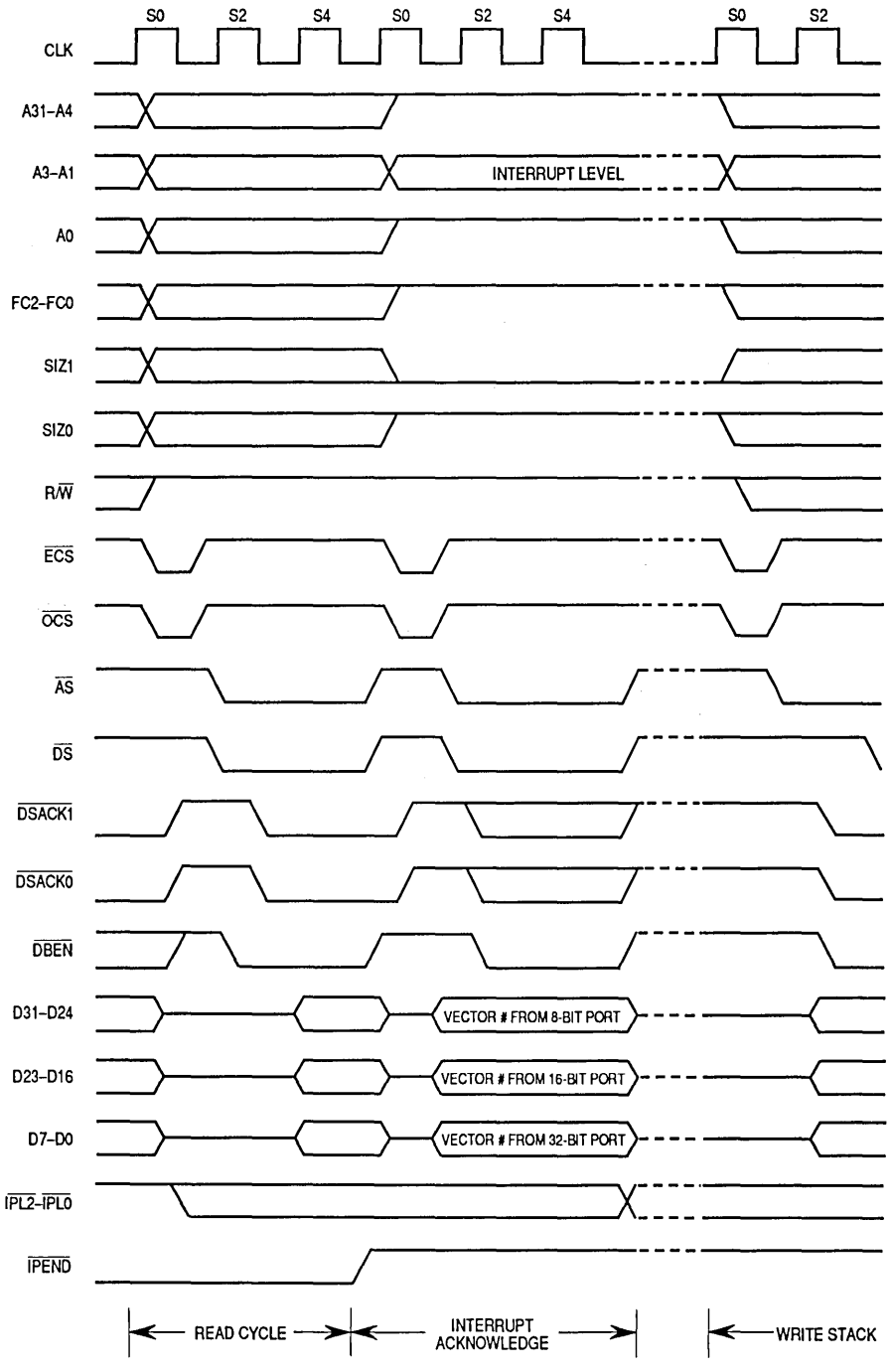


Figure 7-44. Interrupt Acknowledge Cycle Timing

7.4.1.2 AUTOVECTOR INTERRUPT ACKNOWLEDGE CYCLE. When the interrupting device cannot supply a vector number, it requests an automatically generated vector or autovector. Instead of placing a vector number on the data bus and asserting $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$, the device asserts the autovector signal ($\overline{\text{AVEC}}$) to terminate the cycle. Neither $\overline{\text{STERM}}$ nor $\overline{\text{DSACKx}}$ may be asserted during an interrupt acknowledge cycle terminated by $\overline{\text{AVEC}}$.

The vector number supplied in an autovector operation is derived from the interrupt level of the current interrupt. When $\overline{\text{AVEC}}$ is asserted instead of $\overline{\text{DSACK}}$ or $\overline{\text{STERM}}$ during an interrupt acknowledge cycle, the MC68EC030 ignores the state of the data bus and internally generates the vector number, the sum of the interrupt level plus 24 ($18). There are seven distinct autovectors that can be used, corresponding to the seven levels of interrupt available with signals $\overline{\text{IPL0}}$–$\overline{\text{IPL2}}$. Figure 7-45 shows the timing for an autovector operation.

7.4.1.3 SPURIOUS INTERRUPT CYCLE. When a device does not respond to an interrupt acknowledge cycle with $\overline{\text{AVEC}}$, $\overline{\text{STERM}}$, or $\overline{\text{DSACKx}}$, the external logic typically returns $\overline{\text{BERR}}$. The MC68EC030 automatically generates the spurious interrupt vector number, 24, instead of the interrupt vector number in this case. If $\overline{\text{HALT}}$ is also asserted, the controller retries the cycle.

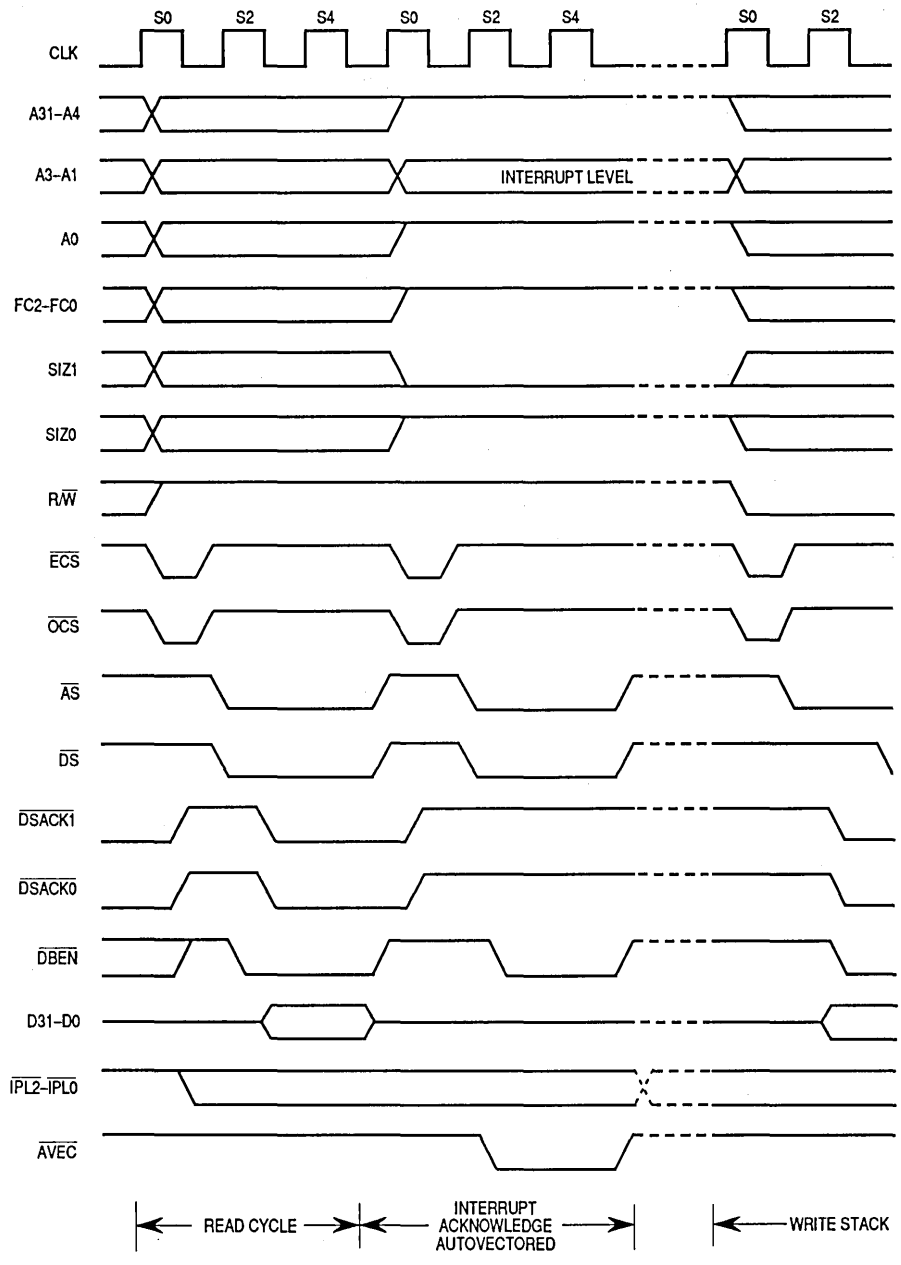


Figure 7-45. Autovector Operation Timing

7.4.2 Breakpoint Acknowledge Cycle

The breakpoint acknowledge cycle is generated by the execution of a breakpoint instruction (BKPT). The breakpoint acknowledge cycle allows the external hardware to provide an instruction word directly into the instruction pipeline as the program executes. This cycle accesses the CPU space with a type field of zero and provides the breakpoint number specified by the instruction on address lines A2–A4. If the external hardware terminates the cycle with $\overline{\text{DSACK}}$x or $\overline{\text{STERM}}$, the data on the bus (an instruction word) is inserted into the instruction pipe, replacing the breakpoint opcode, and is executed after the breakpoint acknowledge cycle completes. The breakpoint instruction requires a word to be transferred so that if the first bus cycle accesses an 8-bit port, a second cycle is required. If the external logic terminates the breakpoint acknowledge cycle with $\overline{\text{BERR}}$ (i.e., no instruction word available), the controller takes an illegal instruction exception. Figure 7-46 is a flowchart of the breakpoint acknowledge cycle. Figure 7-47 shows the timing for a breakpoint acknowledge cycle that returns an instruction word. Figure 7-48 shows the timing for a breakpoint acknowledge cycle that signals an exception.

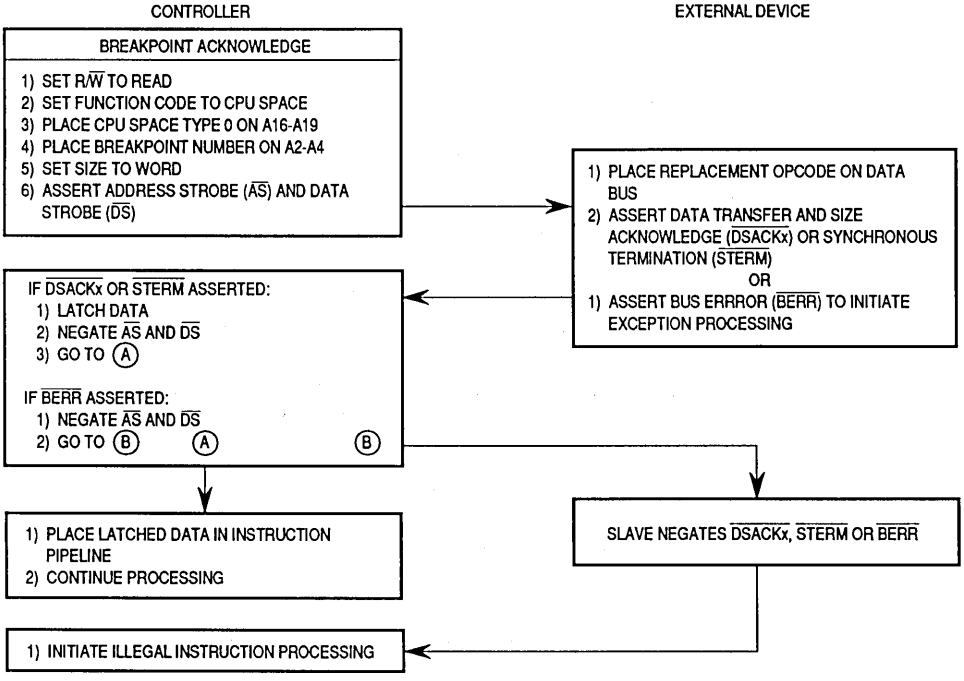


Figure 7-46. Breakpoint Operation Flow

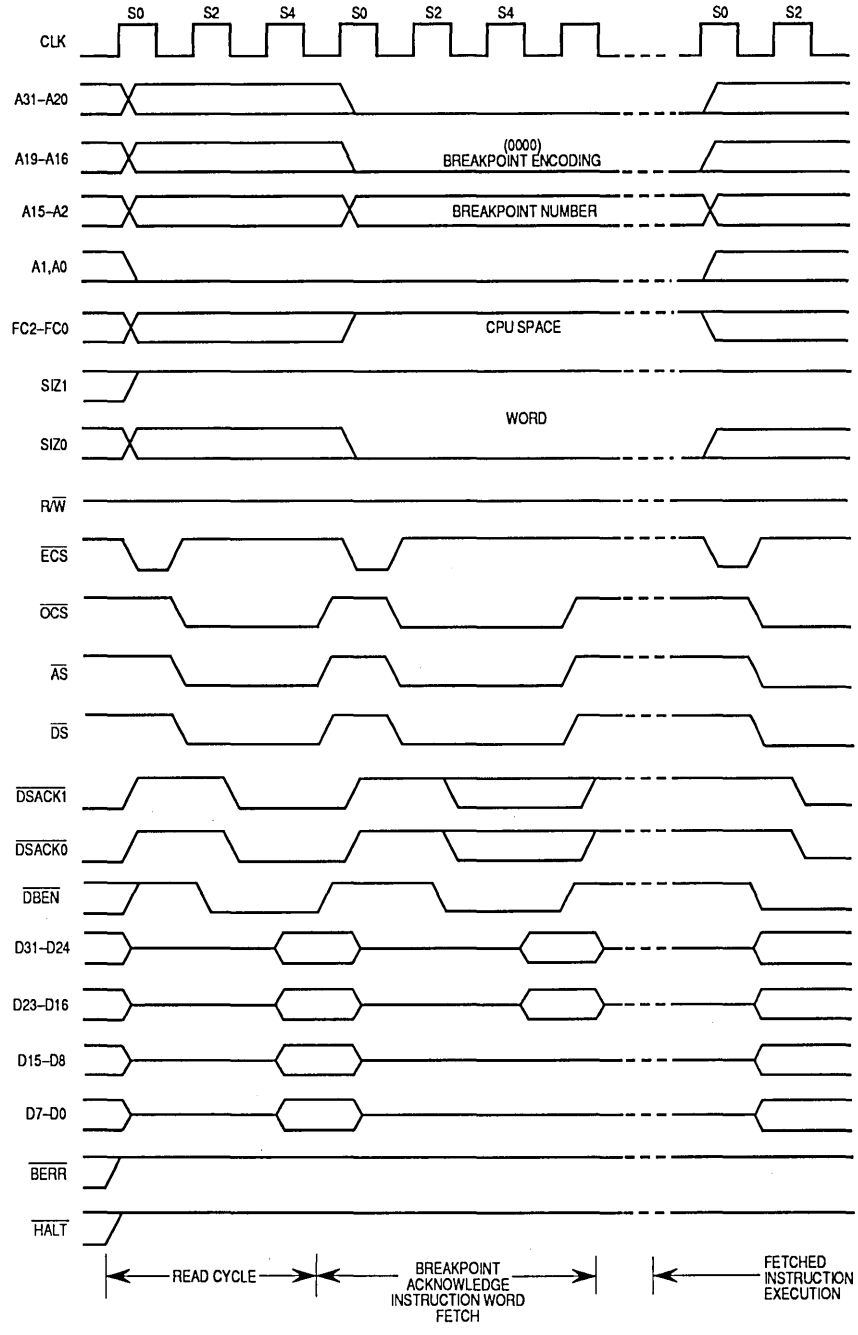


Figure 7-47. Breakpoint Acknowledge Cycle Timing

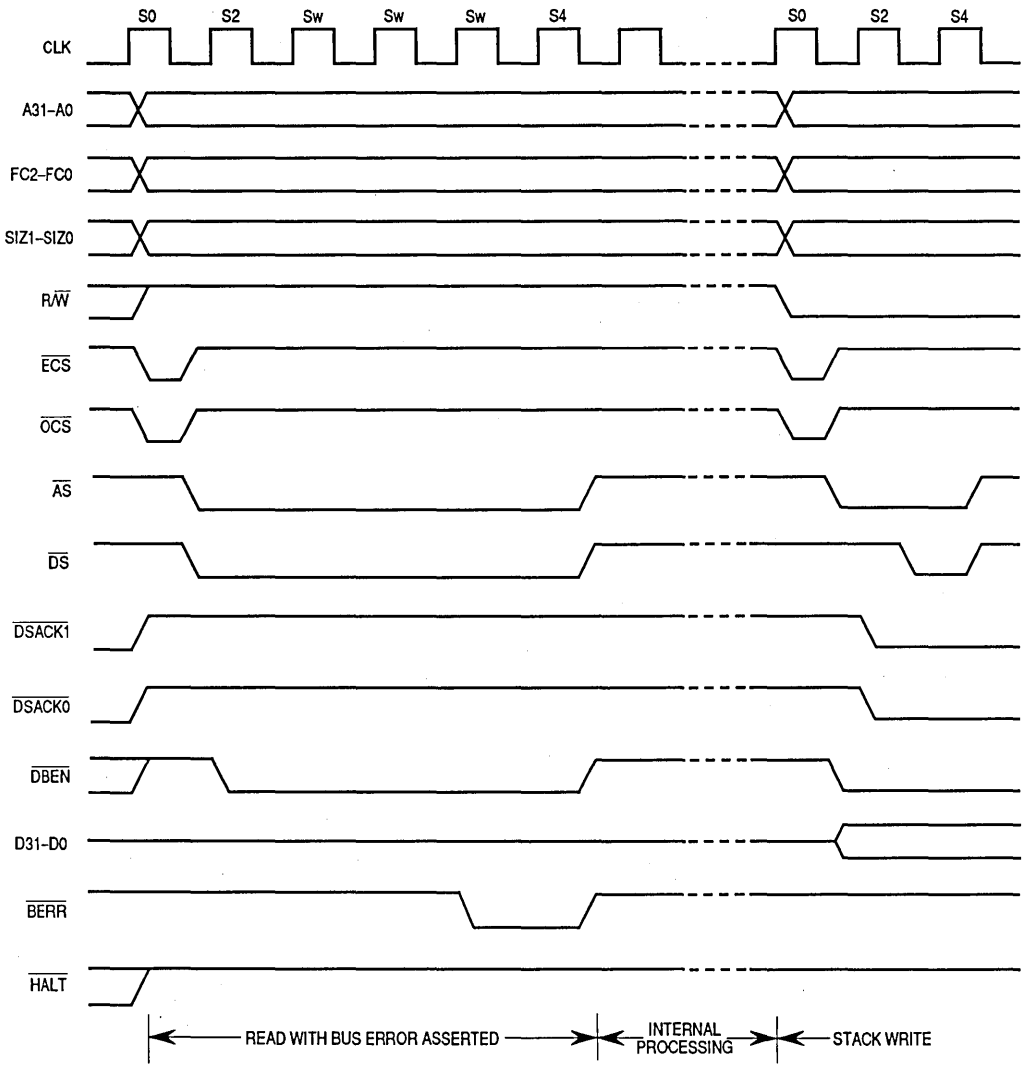


Figure 7-48. Breakpoint Acknowledge Cycle Timing (Exception Signaled)

7.4.3 Coprocessor Communication Cycles

The MC68EC030 coprocessor interface provides instruction-oriented communication between the controller and as many as seven coprocessors. The bus communication required to support coprocessor operations uses the MC68EC030 CPU space with a type field of $2.

Coprocessor accesses use the MC68EC030 bus protocol except that the address bus supplies access information rather than a 32-bit address. The CPU space type field (A16–A19) for a coprocessor operation is $2. A13–A15 contain the coprocessor identification number (CpID), and A0–A4 specify the coprocessor interface register to be accessed. Coprocessor accesses to a CpID of zero correspond to ACU instructions, some of which are not supported by the MC68EC030. These cycles can only be generated by the MOVES instruction. Refer to **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION** for further information.

7.5 BUS EXCEPTION CONTROL CYCLES

The MC68EC030 bus architecture requires assertion of either $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ from an external device to signal that a bus cycle is complete. $\overline{\text{DSACKx}}$, $\overline{\text{STERM}}$, or $\overline{\text{AVEC}}$ is not asserted if:

- The external device does not respond.
- No interrupt vector is provided.
- Various other application-dependent errors occur.

External circuitry can provide $\overline{\text{BERR}}$ when no device responds by asserting $\overline{\text{DSACKx}}$, $\overline{\text{STERM}}$, or $\overline{\text{AVEC}}$ within an appropriate period of time after the controller asserts $\overline{\text{AS}}$. This allows the cycle to terminate and the controller to enter exception processing for the error condition.

Another signal that is used for bus exception control is $\overline{\text{HALT}}$. This signal can be asserted by an external device for debugging purposes to cause single bus cycle operation or (in combination with $\overline{\text{BERR}}$) a retry of a bus cycle in error.

To properly control termination of a bus cycle for a retry or a bus error condition, $\overline{\text{DSACKx}}$, $\overline{\text{BERR}}$, and $\overline{\text{HALT}}$ can be asserted and negated with the rising edge of the MC68EC030 clock. This assures that when two signals are asserted simultaneously, the required setup time (#47A) and hold time (#47B) for both of them is met for the same falling edge of the controller clock.

(Refer to MC68EC030/D, *MC68EC030 Technical Summary* for timing requirements.) This or some equivalent precaution should be designed into the external circuitry that provides these signals.

The acceptable bus cycle terminations for asynchronous cycles are summarized in relation to $\overline{\text{DSACKx}}$ assertion as follows (case numbers refer to Table 7-8):

Normal Termination:
$\overline{\text{DSACKx}}$ is asserted; $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ remain negated (case 1).

Halt Termination:
$\overline{\text{HALT}}$ is asserted at same time or before $\overline{\text{DSACKx}}$, and $\overline{\text{BERR}}$ remains negated (case 2).

Bus Error Termination:
$\overline{\text{BERR}}$ is asserted in lieu of, at the same time, or before $\overline{\text{DSACKx}}$ (case 3) or after $\overline{\text{DSACKx}}$ (case 4), and $\overline{\text{HALT}}$ remains negated; $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{DSACKx}}$.

Retry Termination:
$\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ are asserted in lieu of, at the same time, or before $\overline{\text{DSACKx}}$ (case 5) or after $\overline{\text{DSACKx}}$ (case 6); $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{DSACKx}}$; $\overline{\text{HALT}}$ may be negated at the same time or after $\overline{\text{BERR}}$.

Table 7-8. DSACK, BERR, and HALT Assertion Results

Case No.	Control Signal	Asserted on Rising Edge of State		Result
		N	N+2	
1	DSACKx BERR HALT	A NA NA	S NA X	Normal cycle terminate and continue.
2	DSACKx BERR HALT	A NA A/S	S NA S	Normal cycle terminate and halt. Continue when HALT negated.
3	DSACKx BERR HALT	NA/A A NA	X S NA	Terminate and take bus error exception, possibly deferred.
4	DSACKx BERR HALT	A NA NA	X A NA	Terminate and take bus error exception, possibly deferred.
5	DSACKx BERR HALT	NA/A A A/S	X S S	Terminate and retry when HALT negated.
6	DSACKx BERR HALT	A NA NA	X A A	Terminate and retry when HALT negated.

LEGEND:
N — The number of current even bus state (e.g., S2, S4, etc.)
A — Signal is asserted in this bus state
NA — Signal is not asserted in this state
X — Don't care
S — Signal was asserted in previous state and remains asserted in this state

Table 7-8 shows various combinations of control signal sequences and the resulting bus cycle terminations. To ensure predictable operation, BERR and HALT should be negated according to the specifications in MC68EC030/D, *MC68EC030 Technical Summary*. DSACKx, BERR, and HALT may be negated after AS. If DSACKx or BERR remain asserted into S2 of the next bus cycle, that cycle may be terminated prematurely.

The termination signal for a synchronous cycle is STERM. An analogous set of bus cycle termination cases exists in relationship to STERM assertion. Note that STERM and DSACKx must never both be asserted in the same cycle. STERM has setup time (#60) and hold time (#61) requirements relative to each rising edge of the controller clock while AS is asserted. Bus error and retry terminations during burst cycles operate as described in **6.1.3.2 BURST MODE FILLING**, **7.5.1 Bus Error**, and **7.5.2 Retry Operation**.

For $\overline{\text{STERM}}$, the bus cycle terminations are summarized as follows (case numbers refer to Table 7-9):

Normal Termination:
$\overline{\text{STERM}}$ is asserted; $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ remain negated (case 1).

Halt Termination:
$\overline{\text{HALT}}$ is asserted before $\overline{\text{STERM}}$, and $\overline{\text{BERR}}$ remains negated (case 2).

Bus Error Termination:
$\overline{\text{BERR}}$ is asserted in lieu of, at the same time, or before $\overline{\text{STERM}}$ (case 3) or after $\overline{\text{STERM}}$ (case 4), and $\overline{\text{HALT}}$ remains negated; $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{STERM}}$.

Retry Termination:
$\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ are asserted in lieu of, at the same time, or before $\overline{\text{STERM}}$ (case 5) or after $\overline{\text{STERM}}$ (case 6); $\overline{\text{BERR}}$ is negated at the same time or after $\overline{\text{STERM}}$; $\overline{\text{HALT}}$ may be negated at the same time or after $\overline{\text{BERR}}$.

7

Table 7-9. $\overline{\text{STERM}}$, $\overline{\text{BERR}}$, and $\overline{\text{HALT}}$ Assertion Results

Case No.	Control Signal	Asserted on Rising Edge of State		Result
		N	N+2	
1	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	A NA NA	— — —	Normal cycle terminate and continue.
2	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	NA NA A/S	A NA S	Normal cycle terminate and halt. Continue when $\overline{\text{HALT}}$ negated.
3	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	NA A/S NA	A S NA	Terminate and take bus error exception, possibly deferred.
4	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	A A NA	— — —	Terminate and take bus error exception, possibly deferred.
5	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	NA A A/S	A S S	Terminate and retry when $\overline{\text{HALT}}$ negated.
6	$\overline{\text{STERM}}$ $\overline{\text{BERR}}$ $\overline{\text{HALT}}$	A A A	— — —	Terminate and retry when $\overline{\text{HALT}}$ negated.

LEGEND:
N — The number of current even bus state (e.g., S2, S4, etc.)
A — Signal is asserted in this bus state
NA — Signal is not asserted in this state
X — Don't care
S — Signal was asserted in previous state and remains asserted in this state
— — State N+2 not part of bus cycle

EXAMPLE A:

A system uses a watchdog timer to terminate accesses to an unpopulated address space. The timer asserts $\overline{\text{BERR}}$ after timeout (case 3).

EXAMPLE B:

A system uses error detection and correction on RAM contents. The designer may:

1. Delay $\overline{\text{DSACKx}}$ until data is verified; assert $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ simultaneously to indicate to the controller to automatically retry the error cycle (case 5) or, if data is valid, assert $\overline{\text{DSACKx}}$ (case 1).
2. Delay $\overline{\text{DSACKx}}$ until data is verified and assert $\overline{\text{BERR}}$ with or without $\overline{\text{DSACKx}}$ if data is in error (case 3). This initiates exception processing for software handling of the condition.
3. Return $\overline{\text{DSACKx}}$ prior to data verification. If data is invalid, $\overline{\text{BERR}}$ is asserted on the next clock cycle (case 4). This initiates exception processing for software handling of the condition.
4. Return $\overline{\text{DSACKx}}$ prior to data verification; if data is invalid, assert $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ on the next clock cycle (case 6). The memory controller can then correct the RAM prior to or during the automatic retry.

7.5.1 Bus Errors

The bus error signal can be used to abort the bus cycle and the instruction being executed. $\overline{\text{BERR}}$ takes precedence over $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ provided it meets the timing constraints described in MC68EC030/D, *MC68EC030 Technical Summary*. If $\overline{\text{BERR}}$ does not meet these constraints, it may cause unpredictable operation of the MC68EC030. If $\overline{\text{BERR}}$ remains asserted into the next bus cycle, it may cause incorrect operation of that cycle.

When the bus error signal is issued to terminate a bus cycle, the MC68EC030 may enter exception processing immediately following the bus cycle, or it may defer processing the exception. The instruction prefetch mechanism requests instruction words from the bus controller and the instruction cache before it is ready to execute them. If a bus error occurs on an instruction fetch, the controller does not take the exception until it attempts to use that instruction word. Should an intervening instruction cause a branch or should a task switch occur, the bus error exception does not occur.

The bus error signal is recognized during a bus cycle in any of the following cases:

- $\overline{\text{DSACKx}}$ (or $\overline{\text{STERM}}$) and $\overline{\text{HALT}}$ are negated and $\overline{\text{BERR}}$ is asserted.
- $\overline{\text{HALT}}$ and $\overline{\text{BERR}}$ are negated and $\overline{\text{DSACKx}}$ is asserted. $\overline{\text{BERR}}$ is then asserted within one clock cycle ($\overline{\text{HALT}}$ remains negated).
- $\overline{\text{BERR}}$ is asserted and recognized on the next falling clock edge following the rising clock edge on which $\overline{\text{STERM}}$ is asserted and recognized ($\overline{\text{HALT}}$ remains negated).

When the controller recognizes a bus error condition, it terminates the current bus cycle in the normal way. Figure 7-49 shows the timing of a bus error for the case in which neither $\overline{\text{DSACKx}}$ nor $\overline{\text{STERM}}$ is asserted. Figure 7-50 shows the timing for a bus error that is asserted after $\overline{\text{DSACKx}}$. Exceptions are taken in both cases. (Refer to **8.1.2 Bus Error Exception** for details of bus error exception processing.) When $\overline{\text{BERR}}$ is asserted during a read cycle that supplies data to either on-chip cache, the data in the cache is marked invalid. However, when a write cycle that writes data into the data cache results in an externally generated bus error, the data in the cache is not marked invalid.

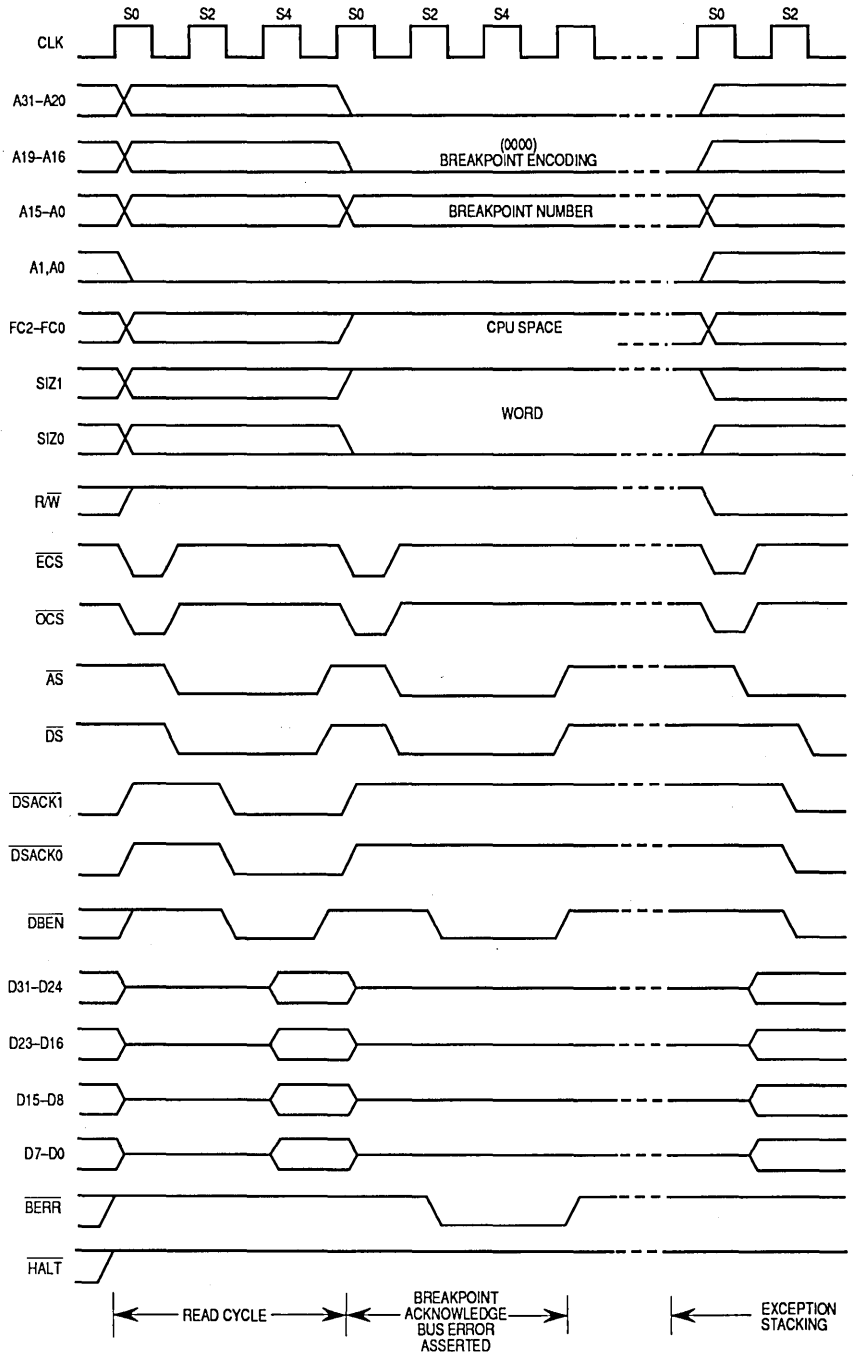


Figure 7-49. Bus Error without $\overline{\text{DSACKx}}$

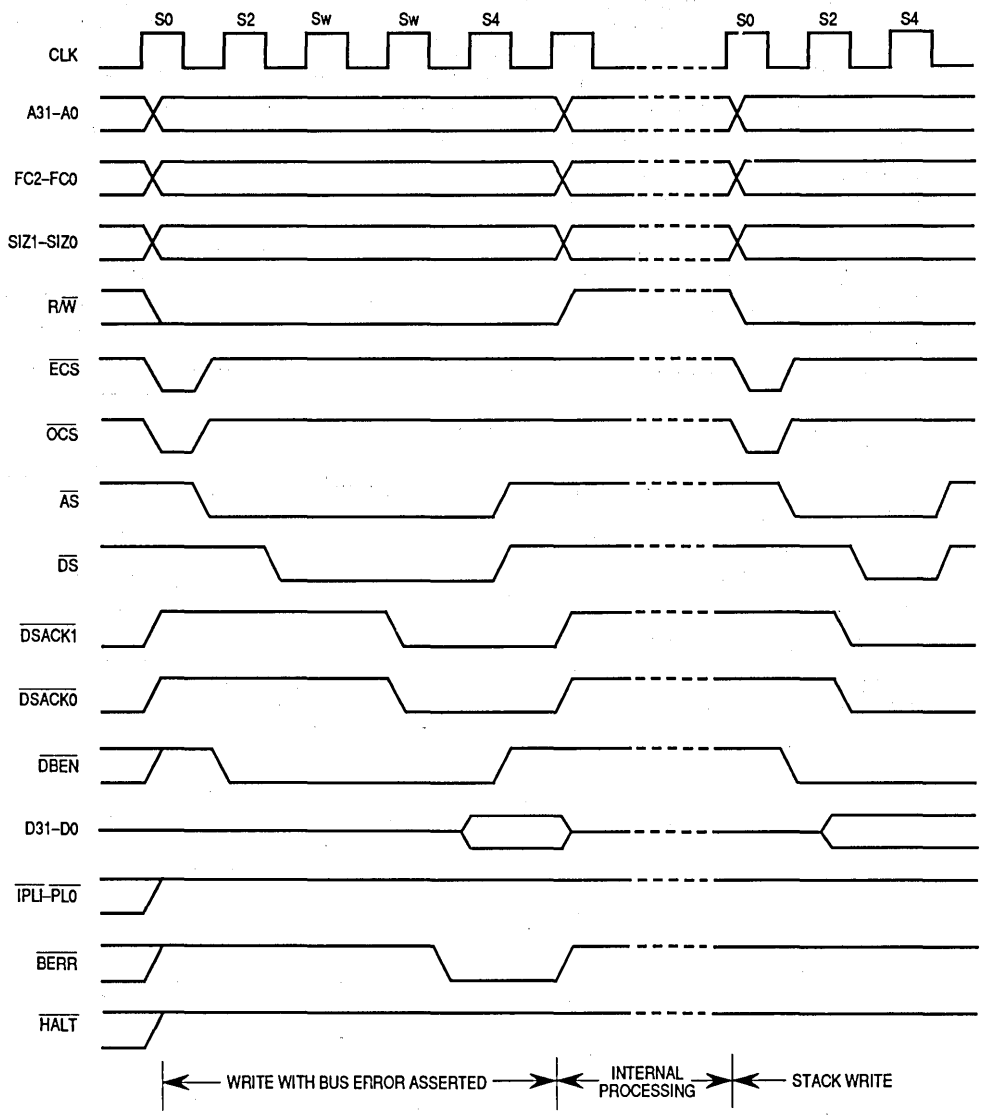


Figure 7-50. Late Bus Error with DSACKx

In the second case, where $\overline{\text{BERR}}$ is asserted after $\overline{\text{DSACKx}}$ is asserted, $\overline{\text{BERR}}$ must be asserted within specification #48 (refer to MC68EC030/D, *MC68EC030 Technical Summary*) for purely asynchronous operation, or it must be asserted and remain stable during the sample window, defined by specifications #27A and #47B, around the next falling edge of the clock after $\overline{\text{DSACKx}}$ is recognized. If $\overline{\text{BERR}}$ is not stable at this time, the controller may exhibit erratic behavior. $\overline{\text{BERR}}$ has priority over $\overline{\text{DSACKx}}$. In this case, data may be present on the bus, but may not be valid. This sequence may be used by systems that have memory error detection and correction logic and by external cache memories.

The assertion of $\overline{\text{BERR}}$ described in the third case (recognized after $\overline{\text{STERM}}$) has requirements similar to those described in the preceding paragraph. $\overline{\text{BERR}}$ must be stable throughout the sample window for the next falling edge of the clock, as defined by specifications #27A and #28A. Figure 7-51 shows the timing for this case.

A bus error occurring during a burst fill operation is a special case. If a bus error occurs during the first cycle of a burst, the data is ignored, the entire cache line is marked invalid, and the burst operation is aborted. If the cycle is for an instruction fetch, a bus error exception is made pending. This bus error is processed only if the execution unit attempts to use either of the two words latched during the bus cycle. If the cycle is for a data fetch, the bus error exception is taken immediately. Refer to **SECTION 11 INSTRUCTION EXECUTION TIMING** for more information about pipeline operation.

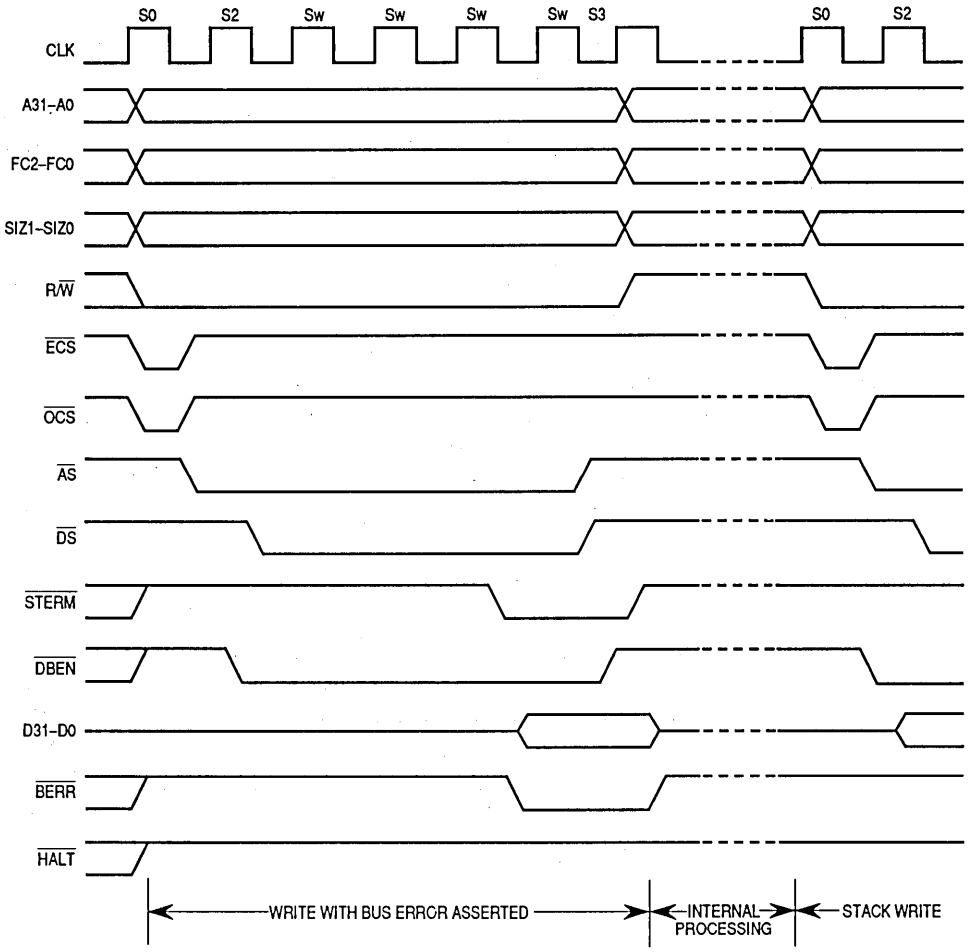


Figure 7-51. Late Bus Error with STERM — Exception Taken

When a bus error occurs after the burst mode has been entered (that is, on the second access or later), the controller terminates the burst operation, and the cache entry corresponding to that cycle is marked invalid, but the controller does not take an exception (see Figure 7-52). If the second cycle is for a portion of a misaligned operand fetch, the controller runs another read cycle for the second portion with CBREQ negated, as shown in Figure 7-53. If BERR is asserted again, the MC68EC030 then takes an exception. The MC68EC030 supports late bus errors during a burst fill operation; the timing is the same relative to STERM and the clock as for a late bus error in a normal synchronous cycle.

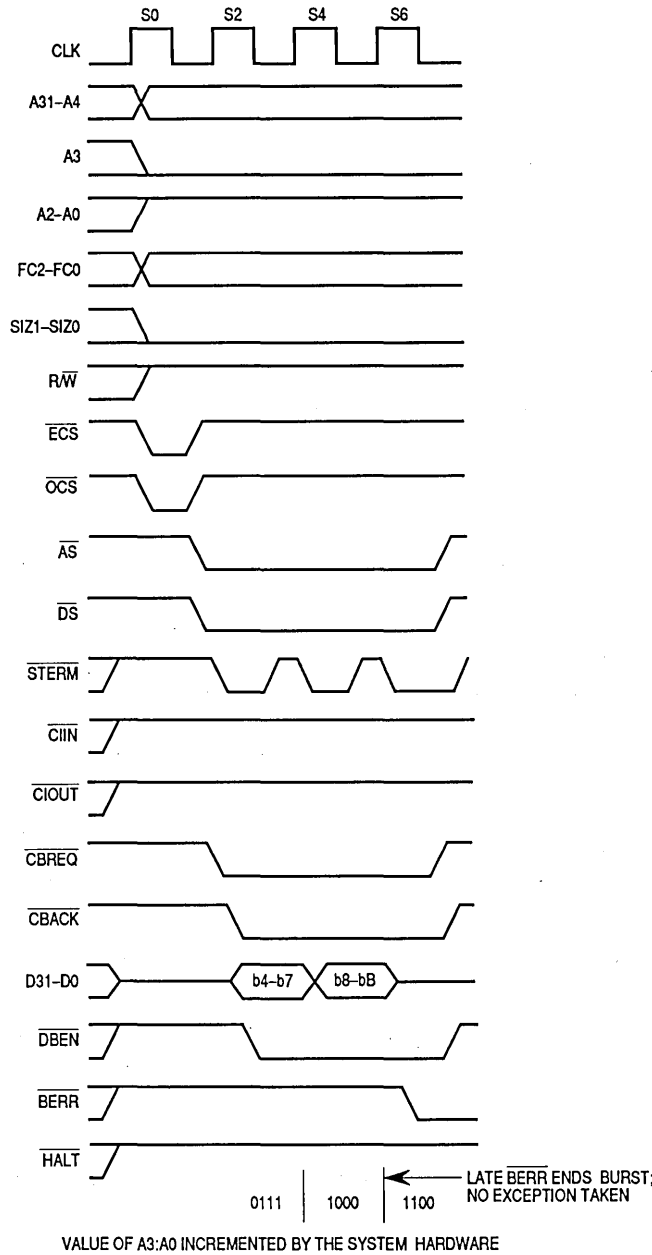


Figure 7-52. Long-Word Operand Request — Late $\overline{\text{BERR}}$ on Third Access

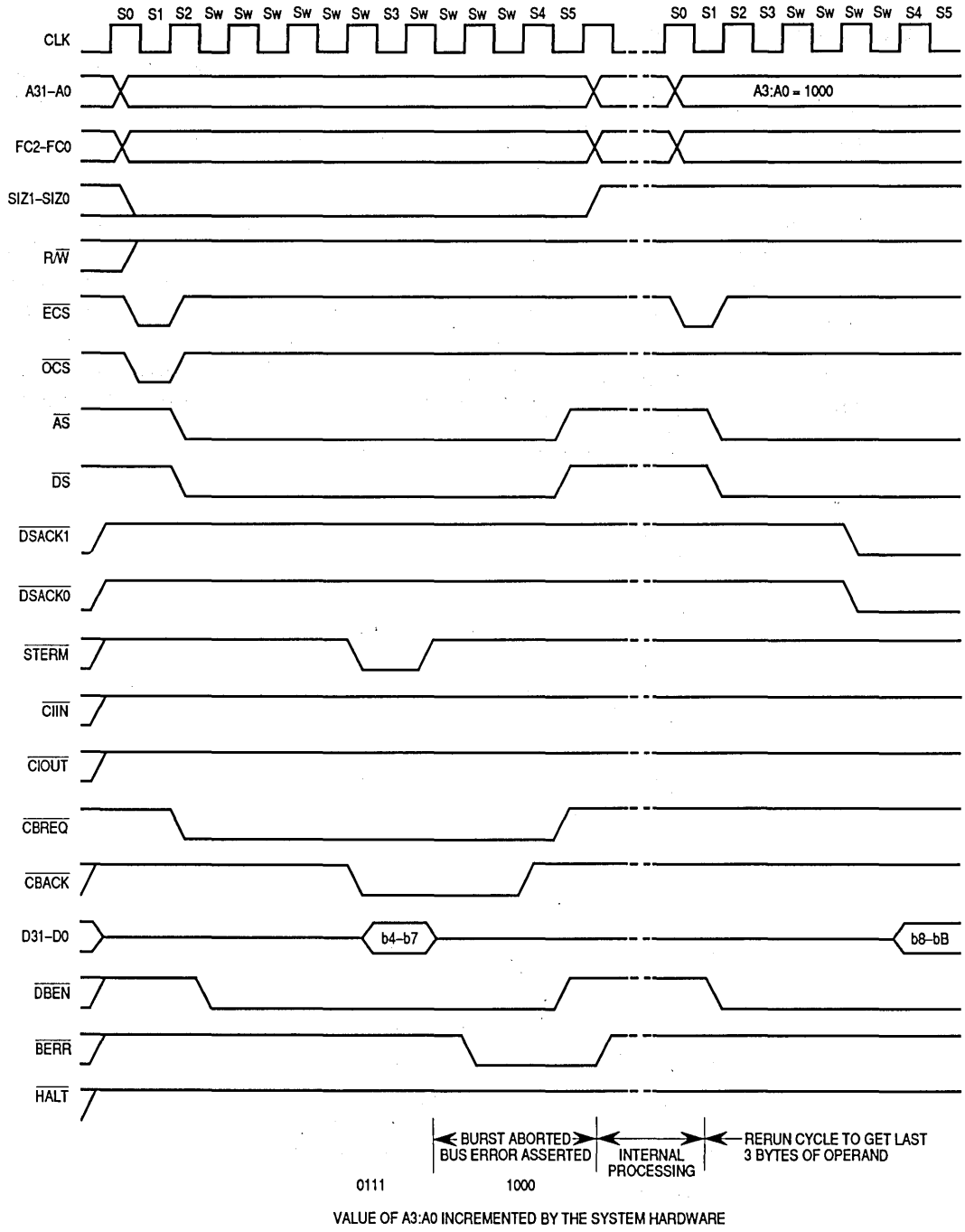


Figure 7-53. Long-Word Operand Request — $\overline{\text{BERR}}$ on Second Access

7.5.2 Retry Operation

When the $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ signals are both asserted by an external device during a bus cycle, the controller enters the retry sequence. A delayed retry, similar to the delayed bus error signal described previously, can also occur, both for synchronous and asynchronous cycles.

The controller terminates the bus cycle, places the control signals in their inactive state, and does not begin another bus cycle until the $\overline{\text{HALT}}$ signal is negated by external logic. After a synchronization delay, the controller retries the previous cycle using the same access information (address, function code, size, etc.) The $\overline{\text{BERR}}$ signal should be negated before S2 of the read cycle to ensure correct operation of the retried cycle. Figure 7-54 shows a retry operation of an asynchronous cycle, and Figure 7-55 shows a retry operation of a synchronous cycle.

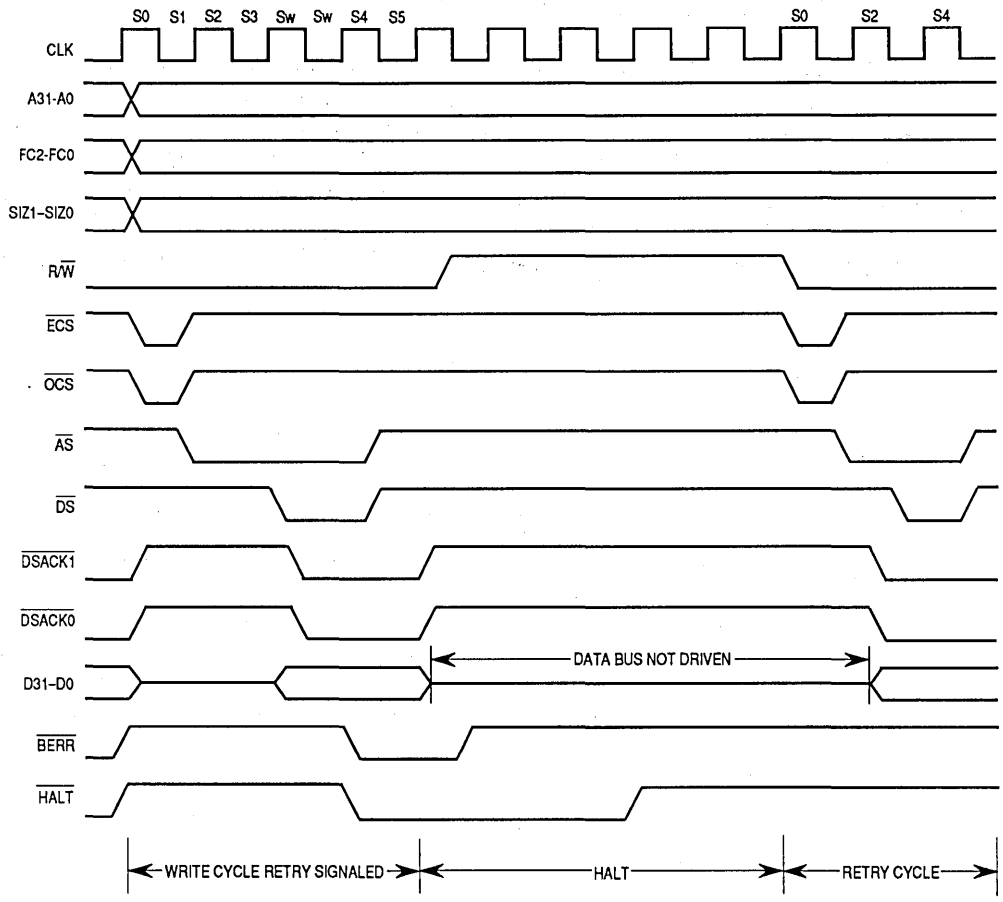


Figure 7-54. Asynchronous Late Retry

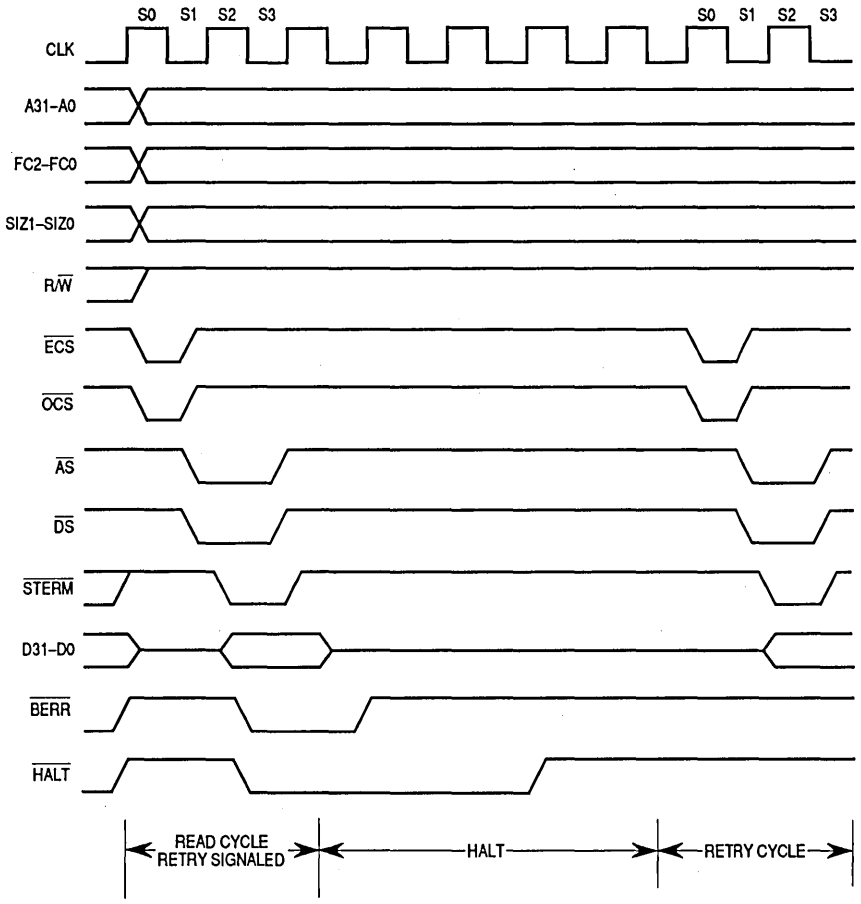


Figure 7-55. Synchronous Late Retry

The controller retries any read or write cycle of a read-modify-write operation separately; RMC remains asserted during the entire retry sequence.

On the initial access of a burst operation, a retry (indicated by the assertion of BERR and HALT) causes the controller to retry the bus cycle and assert CBREQ again. Figure 7-56 shows a late retry operation that causes an initial burst operation to be repeated. However, signaling a retry with simultaneous BERR and HALT during the second, third, or fourth cycle of a burst operation does not cause a retry operation, even if the requested operand is misaligned. Assertion of BERR and HALT during a subsequent cycle of a burst operation

causes independent $\overline{BERR}$ and $\overline{HALT}$ operations. The external bus activity remains halted until $\overline{HALT}$ is negated and the controller acts as previously described for the bus error during a burst operation.

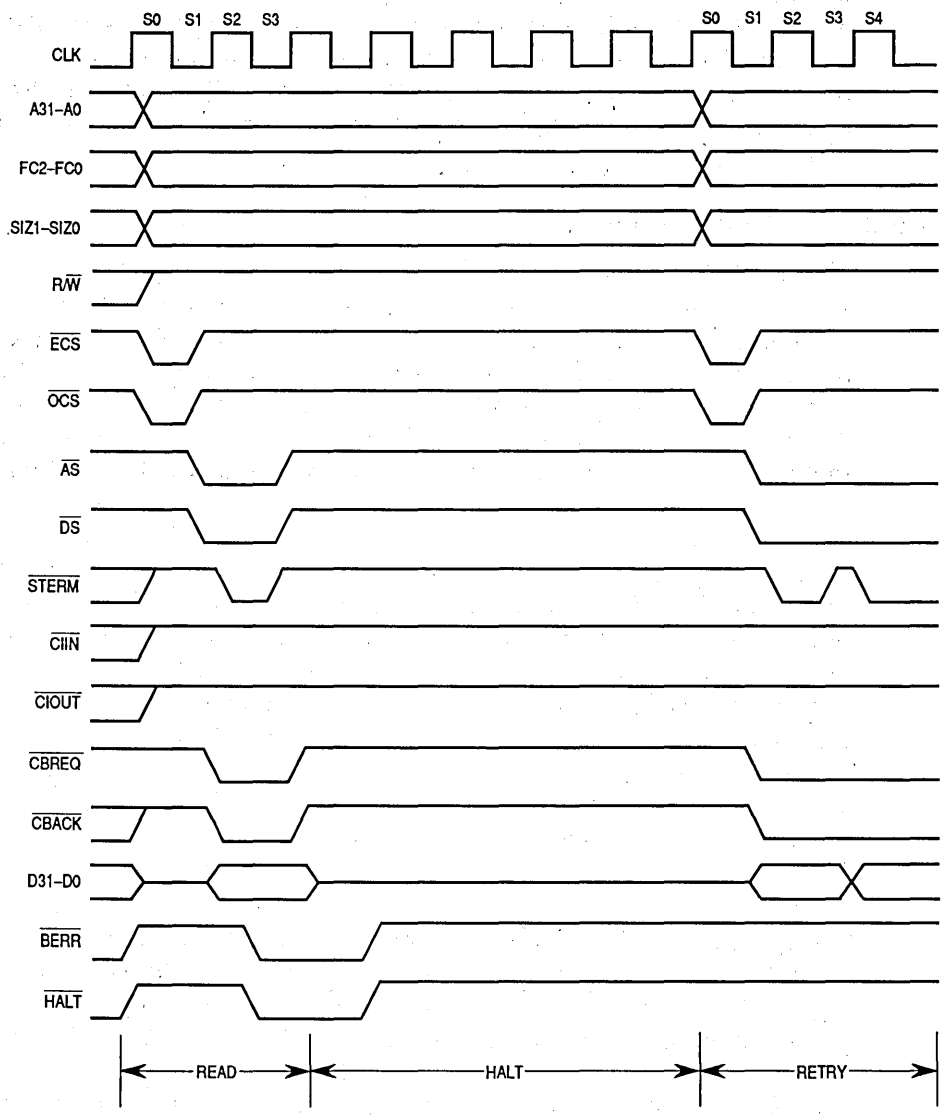


Figure 7-56. Late Retry Operation for a Burst

Asserting $\overline{BR}$ along with $\overline{BERR}$ and $\overline{HALT}$ provides a relinquish and retry operation. The MC68EC030 does not relinquish the bus during a read-modify-write operation, except during the first read cycle. Any device that requires the controller to give up the bus and retry a bus cycle during a read-modify-write cycle must either assert $\overline{BERR}$ and $\overline{BR}$ only ($\overline{HALT}$ must not be included) or use the single wire arbitration method discussed in **7.7.4 Bus Arbitration Control**. The bus error handler software should examine the read-modify-write bit in the special status word (refer to **8.2.1 Special Status Word**) and take the appropriate action to resolve this type of fault when it occurs.

7.5.3 Halt Operation

When $\overline{HALT}$ is asserted and $\overline{BERR}$ is not asserted, the MC68EC030 halts external bus activity at the next bus cycle boundary. $\overline{HALT}$ by itself does not terminate a bus cycle. Negating and reasserting $\overline{HALT}$ in accordance with the correct timing requirements provides a single-step (bus cycle to bus cycle) operation. The $\overline{HALT}$ signal affects external bus cycles only; thus, a program that resides in the instruction cache and performs no data writes (or reads that miss in the data cache) may continue executing, unaffected by the $\overline{HALT}$ signal.

The single-cycle mode allows the user to proceed through (and debug) external controller operations, one bus cycle at a time. Figure 7-57 shows the timing requirements for a single-cycle operation. Since the occurrence of a bus error while $\overline{HALT}$ is asserted causes a retry operation, the user must anticipate retry cycles while debugging in the single-cycle mode. The single-step operation and the software trace capability allow the system debugger to trace single bus cycles, single instructions, or changes in program flow. These controller capabilities, along with a software debugging package, give complete debugging flexibility.

When the controller completes a bus cycle with the $\overline{HALT}$ signal asserted, the data bus is placed in the high-impedance state, and bus control signals are driven inactive (not high-impedance state); the address, function code, size, and read/write signals remain in the same state. The halt operation has no effect on bus arbitration (refer to **7.7 BUS ARBITRATION**). When bus arbitration occurs while the MC68EC030 is halted, the address and control signals are also placed in the high-impedance state. Once bus mastership is returned to the MC68EC030, if $\overline{HALT}$ is still asserted, the address, function code, size, and read/write signals are again driven to their previous states. The controller does not service interrupt requests while it is halted, but it may assert the $\overline{IPEND}$ signal as appropriate.

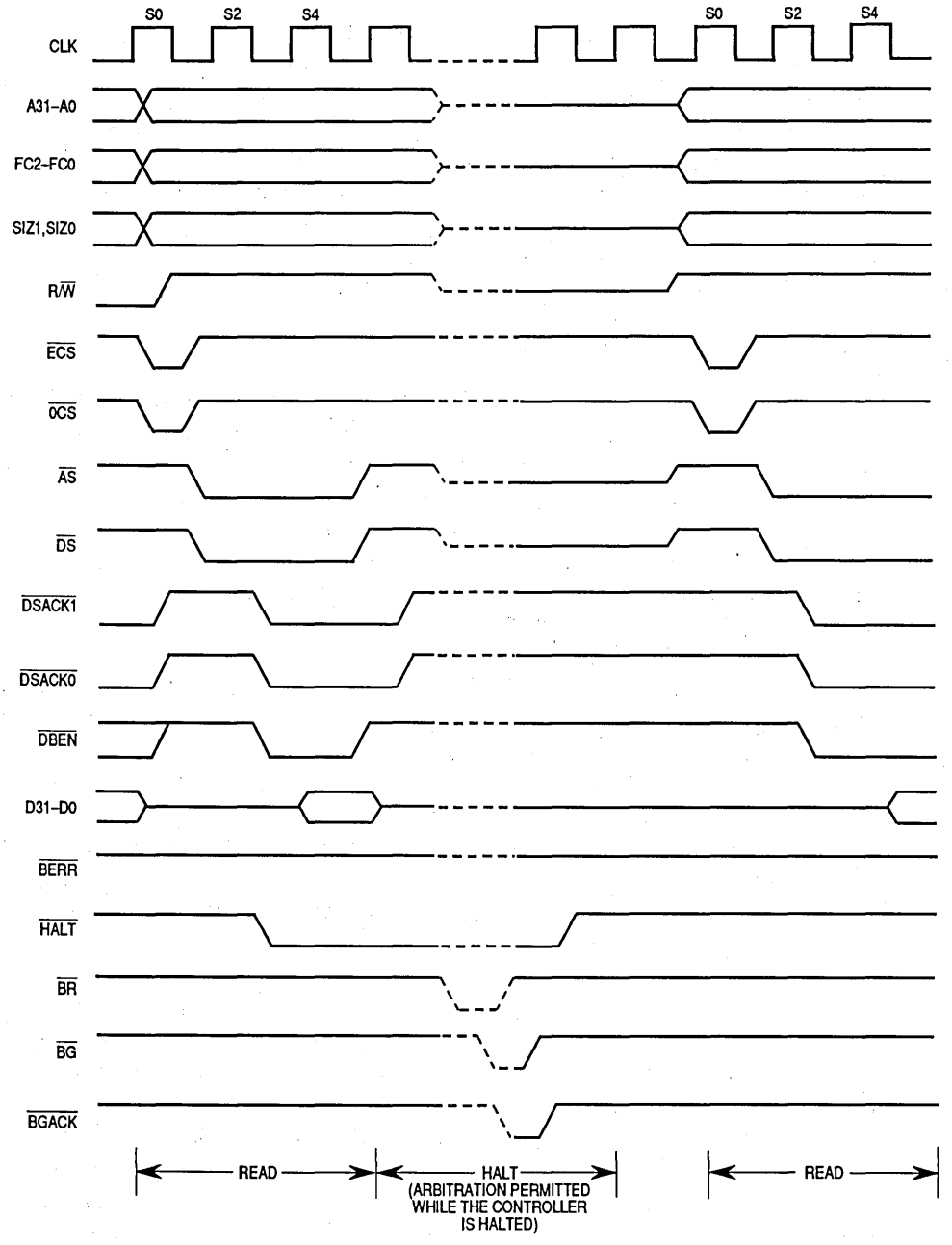


Figure 7-57. Halt Operation Timing

7.5.4 Double Bus Fault

When a bus error or an address error occurs during the exception processing sequence for a previous bus error, a previous address error, or a reset exception, the bus or address error causes a double bus fault. For example, the controller attempts to stack several words containing information about the state of the machine while processing a bus error exception. If a bus error exception occurs during the stacking operation, the second error is considered a double bus fault. Only an external reset operation can restart a halted controller. However, bus arbitration can still occur (refer to **7.7 BUS ARBITRATION**).

The MC68EC030 indicates that a double bus fault condition has occurred by continuously asserting the $\overline{\text{STATUS}}$ signal until the controller is reset. The controller asserts $\overline{\text{STATUS}}$ for one, two, or three clock periods to signal other microsequencer status indications. Refer to **SECTION 12 APPLICATIONS INFORMATION** for a description of the interpretation of the $\overline{\text{STATUS}}$ signal.

A second bus error or address error that occurs after exception processing has completed (during the execution of the exception handler routine or later) does not cause a double bus fault. A bus cycle that is retried does not constitute a bus error or contribute to a double bus fault. The controller continues to retry the same bus cycle as long as the external hardware requests it.

7.6 BUS SYNCHRONIZATION

The MC68EC030 overlaps instruction execution; that is, during bus activity for one instruction, instructions that do not use the external bus can be executed. Due to the independent operation of the on-chip caches relative to the operation of the bus controller, many subsequent instructions can be executed, resulting in seemingly nonsequential instruction execution. When this is not desired and the system depends on sequential execution following bus activity, the NOP instruction can be used. The NOP instruction forces instruction and bus synchronization in that it freezes instruction execution until all pending bus cycles have completed.

An example of the use of the NOP instruction for this purpose is the case of a write operation of control information to an external register, where the external hardware attempts to control program execution based on the data that is written with the conditional assertion of $\overline{\text{BERR}}$. If the data cache is enabled and the write cycle results in a hit in the data cache, the cache is updated. That data, in turn, may be used in a subsequent instruction before the external write cycle completes. Since the MC68EC030 cannot process the

bus error until the end of the bus cycle, the external hardware has not successfully interrupted program execution. To prevent a subsequent instruction from executing until the external cycle completes, a NOP instruction can be inserted after the instruction causing the write. In this case, bus error exception processing proceeds immediately after the write before subsequent instructions are executed. This is an irregular situation, and the use of the NOP instruction for this purpose is not required by most systems.

Note that even in a system with error detection/correction circuitry, the NOP is not required for this synchronization. Since the ACU always checks the validity of write cycles before they proceed to the data cache and are executed externally, the MC68EC030 is guaranteed to write correct data to the cache. Thus, there is no danger in subsequent instructions using erroneous data from the cache before an external bus error signals an error.

A bus synchronization example is given in Figure 7-58.

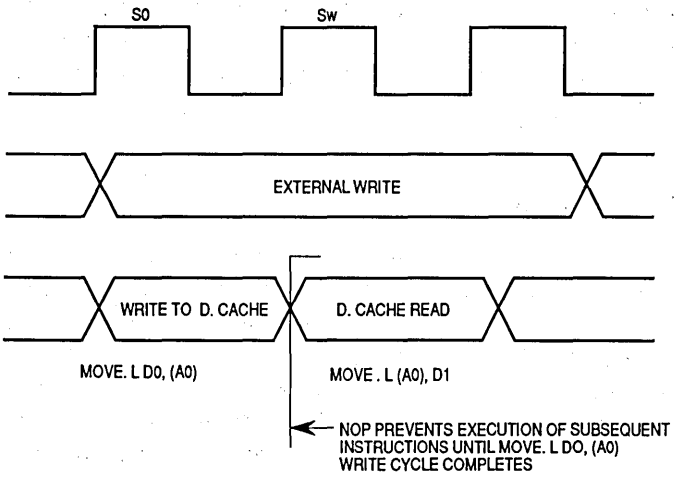


Figure 7-58. Bus Synchronization Example

7.7 BUS ARBITRATION

The bus design of the MC68EC030 provides for a single bus master at any one time: either the controller or an external device. One or more of the external devices on the bus can have the capability of becoming bus master. Bus arbitration is the protocol by which an external device becomes bus master; the bus controller in the MC68EC030 manages the bus arbitration signals so that the controller has the lowest priority. External devices that need to obtain the bus must assert the bus arbitration signals in the sequences described in the following paragraphs. Systems having several devices that can become bus master require external circuitry to assign priorities to the device so that, when two or more external devices attempt to become bus master at the same time, the one having the highest priority becomes bus master first. The sequence of the protocol is:

1. An external device asserts the bus request signal.
2. The controller asserts the bus grant signal to indicate that the bus will become available at the end of the current bus cycle.
3. The external device asserts the bus grant acknowledge signal to indicate that it has assumed bus mastership.

$\overline{BR}$ may be issued any time during a bus cycle or between cycles. $\overline{BG}$ is asserted in response to $\overline{BR}$; it is usually asserted as soon as $\overline{BR}$ has been synchronized and recognized, except when the MC68EC030 has made an internal decision to execute a bus cycle. Then, the assertion of $\overline{BG}$ is deferred until the bus cycle has begun. Additionally, $\overline{BG}$ is not asserted until the end of a read-modify-write operation (when $\overline{RMC}$ is negated) in response to a $\overline{BR}$ signal. When the requesting device receives $\overline{BG}$ and more than one external device can be bus master, the requesting device should begin whatever arbitration is required. The external device asserts $\overline{BGACK}$ when it assumes bus mastership and maintains $\overline{BGACK}$ during the entire bus cycle (or cycles) for which it is bus master. The following conditions must be met for an external device to assume mastership of the bus through the normal bus arbitration procedure:

- It must have received $\overline{BG}$ through the arbitration process.
- $\overline{AS}$ must be negated, indicating that no bus cycle is in progress, and the external device must ensure that all appropriate controller signals have been placed in the high-impedance state (by observing specification #7 in MC68EC030/D, *MC68EC030 Technical Summary*).
- The termination signal ($\overline{DSACKx}$ or $\overline{STERM}$) for the most recent cycle must have become inactive, indicating that external devices are off the bus (optional, refer to **7.7.3 Bus Grant Acknowledge**).
- $\overline{BGACK}$ must be inactive, indicating that no other bus master has claimed ownership of the bus.

Figure 7-59 is a flowchart showing the detail involved in bus arbitration for a single device. Figure 7-60 is a timing diagram for the same operation. This technique allows processing of bus requests during data transfer cycles.

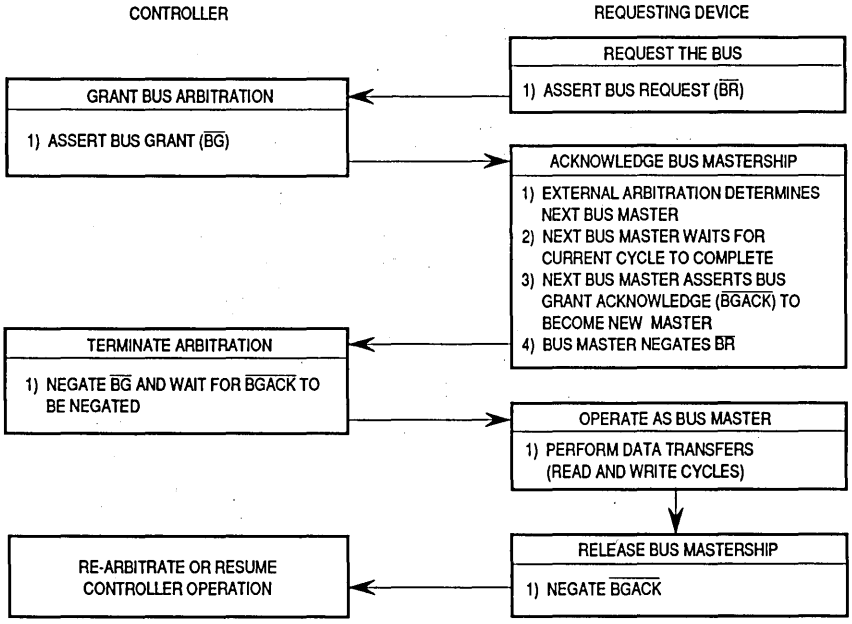


Figure 7-59. Bus Arbitration Flowchart for Single Request

The timing diagram shows that $\overline{BR}$ is negated at the time that $\overline{BGACK}$ is asserted. This type of operation applies to a system consisting of the controller and one device capable of bus mastership. In a system having a num-

ber of devices capable of bus mastership, the bus request line from each device can be wire-ORed to the controller. In such a system, more than one bus request can be asserted simultaneously.

The timing diagram in Figure 7-60 shows that $\overline{BG}$ is negated a few clock cycles after the transition of the $\overline{BGACK}$ signal. However, if bus requests are still pending after the negation of $\overline{BG}$, the controller asserts another $\overline{BG}$ within a few clock cycles after it was negated. This additional assertion of $\overline{BG}$ allows external arbitration circuitry to select the next bus master before the current bus master has finished with the bus. The following paragraphs provide additional information about the three steps in the arbitration process.

Bus arbitration requests are recognized during normal processing, $\overline{RESET}$ assertion, $\overline{HALT}$ assertion, and even when the controller has halted due to a double bus fault.

7.7.1 Bus Request

External devices capable of becoming bus masters request the bus by asserting $\overline{BR}$. This can be a wire-ORed signal (although it need not be constructed from open-collector devices) that indicates to the controller that some external device requires control of the bus. The controller is effectively at a lower bus priority level than the external device and relinquishes the bus after it has completed the current bus cycle (if one has started).

If no acknowledge is received while the $\overline{BR}$ is active, the controller remains bus master once $\overline{BR}$ is negated. This prevents unnecessary interference with ordinary processing if the arbitration circuitry inadvertently responds to noise or if an external device determines that it no longer requires use of the bus before it has been granted mastership.

7.7.2 Bus Grant

The controller asserts $\overline{BG}$ as soon as possible after receipt of $\overline{BR}$. This is immediately following internal synchronization except during a read-modify-write cycle or following an internal decision to execute a bus cycle. During a read-modify-write cycle, the controller does not assert $\overline{BG}$ until the entire operation has completed. $\overline{RMC}$ is asserted to indicate that the bus is locked. In the case of an internal decision to execute another bus cycle, $\overline{BG}$ is deferred until the bus cycle has begun.

$\overline{BG}$ may be routed through a daisy-chained network or through a specific priority-encoded network. The controller allows any type of external arbitration that follows the protocol.

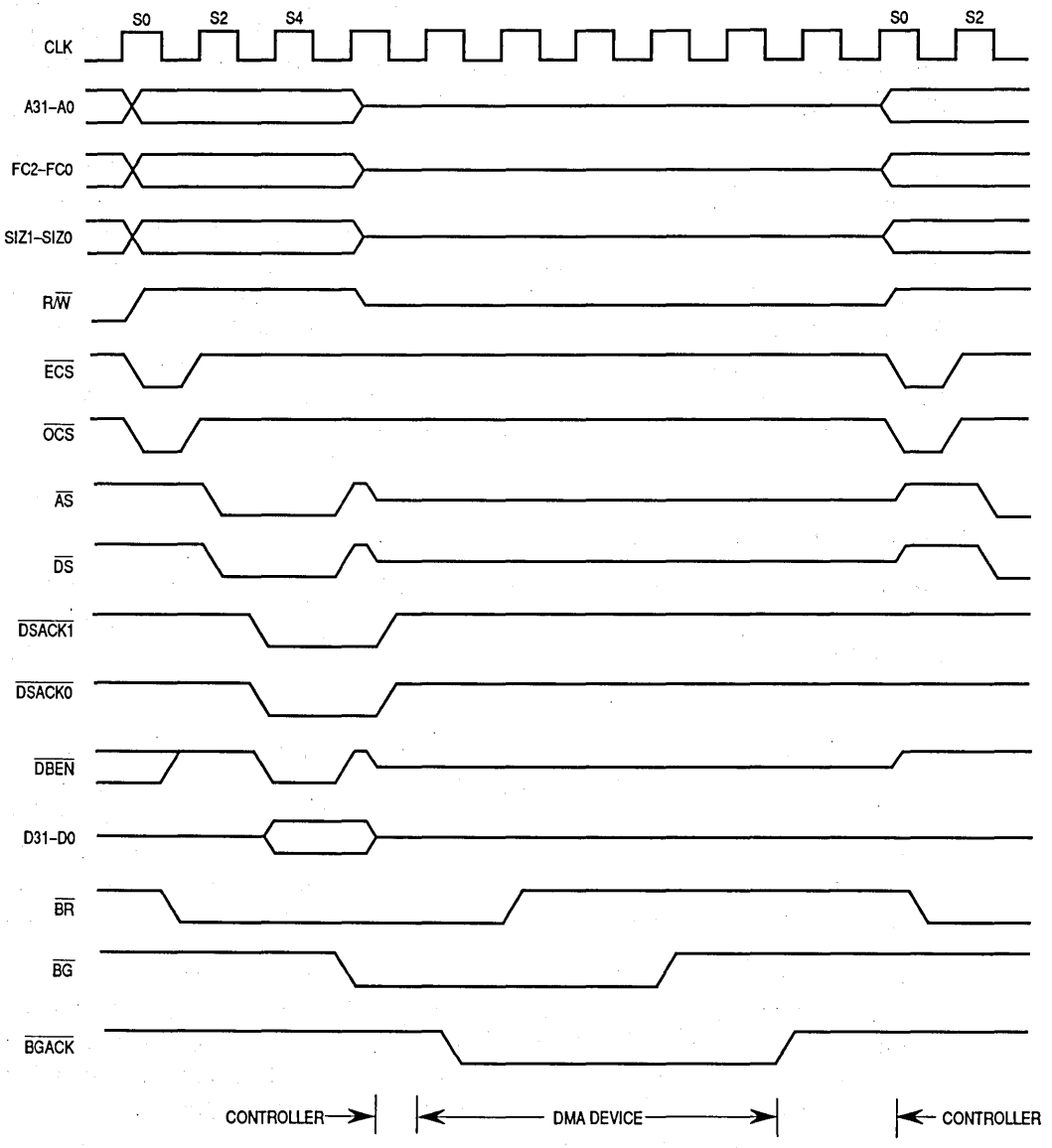


Figure 7-60. Bus Arbitration Operation Timing

7.7.3 Bus Grant Acknowledge

Upon receiving $\overline{BG}$, the requesting device waits until $\overline{AS}$, $\overline{DSACKx}$ (or synchronous termination, $\overline{STERM}$), and $\overline{BGACK}$ are negated before asserting its own $\overline{BGACK}$. The negation of the $\overline{AS}$ indicates that the previous master releases the bus after specification #7 (refer to MC68EC030/D, *MC68EC030 Technical Summary*). The negation of $\overline{DSACKx}$ or $\overline{STERM}$ indicates that the previous slave has completed its cycle with the previous master. Note that in some applications, $\overline{DSACKx}$ might not be used in this way.

General-purpose devices are then connected to be dependent only on $\overline{AS}$. When $\overline{BGACK}$ is asserted, the device is the bus master until it negates $\overline{BGACK}$. $\overline{BGACK}$ should not be negated until all bus cycles required by the alternate bus master are completed. Bus mastership terminates at the negation of $\overline{BGACK}$. The $\overline{BR}$ from the granted device should be negated after $\overline{BGACK}$ is asserted. If a $\overline{BR}$ is still pending after the assertion of $\overline{BGACK}$, another $\overline{BG}$ is asserted within a few clocks of the negation of $\overline{BG}$, as described in the **7.7.4 Bus Arbitration Control**. Note that the controller does not perform any external bus cycles before it reasserts $\overline{BG}$ in this case.

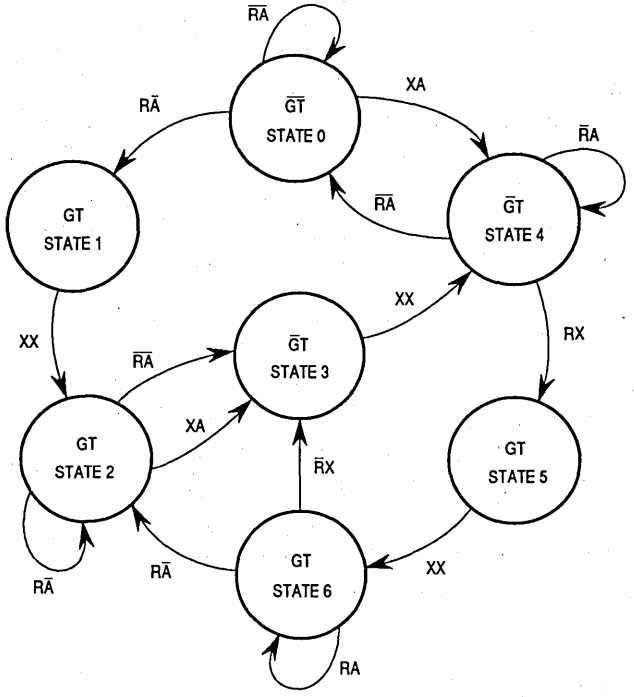
7.7.4 Bus Arbitration Control

The bus arbitration control unit in the MC68EC030 is implemented with a finite state machine. As discussed previously, all asynchronous inputs to the MC68EC030 are internally synchronized in a maximum of two cycles of the controller clock.

As shown in Figure 7-61, input signals labeled R and A are internally synchronized versions of the $\overline{BR}$ and $\overline{BGACK}$ signals, respectively. The $\overline{BG}$ output is labeled G, and the internal high-impedance control signal is labeled T. If T is true, the address, data, and control buses are placed in the high-impedance state after the next rising edge following the negation of $\overline{AS}$ and $\overline{RMC}$. All signals are shown in positive logic (active high), regardless of their true active voltage level.

State changes occur on the next rising edge of the clock after the internal signal is valid. The $\overline{BG}$ signal transitions on the falling edge of the clock after a state is reached during which G changes. The bus control signals (controlled by T) are driven by the controller, immediately following a state change, when bus mastership is returned to the MC68EC030.

State 0, at the top center of the diagram, in which G and T are both negated, is the state of the bus arbiter while the controller is bus master. Request R



R - BUS REQUEST
A - BUS GRANT ACKNOWLEDGE
G - BUS GRANT
T - THREE-STATE CONTROL TO BUS CONTROL LOGIC
X - DON'T CARE

NOTE: The $\overline{BG}$ output will not be asserted while $\overline{RMC}$ is asserted.

Figure 7-61. Bus Arbitration State Diagram

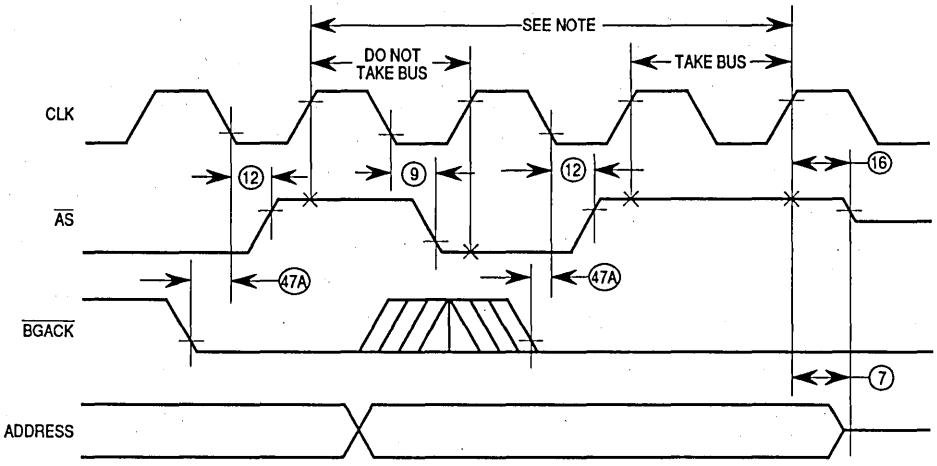
and acknowledge A keep the arbiter in state 0 as long as they are both negated. When a request R is received, both grant G and signal T are asserted (in state 1 at the top left). The next clock causes a change to state 2, at the lower left, in which G and T are held. The bus arbiter remains in that state until acknowledge A is asserted or request R is negated. Once either occurs, the arbiter changes to the center state, state 3, and negates grant G. The next clock takes the arbiter to state 4, at the upper right, in which grant G remains negated and signal T remains asserted. With acknowledge A asserted, the arbiter remains in state 4 until A is negated or request R is again asserted. When A is negated, the arbiter returns to the original state, state 0, and negates signal T. This sequence of states follows the normal sequence of signals for relinquishing the bus to an external bus master. Other states apply

to other possible sequences of combinations of R and A. As shown by the path from state 0 to state 4, $\overline{\text{BGACK}}$ alone can be used to place the controller's external bus buffers in the high-impedance state, providing single-wire arbitration capability.

The read-modify-write sequence is normally indivisible to support semaphore operations and multiprocessor synchronization. During this indivisible sequence, the MC68EC030 asserts the $\overline{\text{RMC}}$ signal and causes the bus arbitration state machine to ignore bus requests (assertions of $\overline{\text{BR}}$) that occur after the first read cycle of the read-modify-write sequence by not issuing bus grants (asserting $\overline{\text{BG}}$).

In some cases, however, it may be necessary to force the MC68EC030 to release the bus during an read-modify-write sequence. One way for an alternate bus master to force the MC68EC030 to release the bus applies only to the first read cycle of an read-modify-write sequence. The MC68EC030 allows normal bus arbitration during this read cycle; a normal relinquish and retry operation (asserting $\overline{\text{BERR}}$, $\overline{\text{HALT}}$, and $\overline{\text{BR}}$ at the same time) is used. Note that this method applies only to the first read cycle of the read-modify-write sequence, but this method preserves the integrity of the read-modify-write sequence without imposing any constraint on the alternate bus master.

A second method is single-wire arbitration, the timing of which is shown in Figure 7-62. An alternate master forces the MC68EC030 to release the bus by asserting $\overline{\text{BGACK}}$ and waits for $\overline{\text{AS}}$ to negate before taking the bus. It applies to all bus cycles of a read-modify-write sequence, but can cause system integrity problems if used improperly. The alternate bus master must guarantee the integrity of the read-modify-write sequence by not altering the contents of memory locations accessed by the read-modify-write sequence. Note that for the method to operate properly, $\overline{\text{AS}}$ must be observed to be negated (high) on two consecutive clock edges before the alternate bus master takes the bus. Waiting for this condition ensures that any current or pending bus activity has completed or has been pre-empted.



NOTE: The alternate bus master must sample $\overline{AS}$ high on two consecutive rising edges of the clock (after $\overline{BGACK}$ is recognized low) before taking the bus.

Figure 7-62. Single-Wire Bus Arbitration Timing Diagram

A timing diagram of the bus arbitration sequence during a controller bus cycle is shown in Figure 7-60. The bus arbitration sequence while the bus is inactive (i.e., executing internal operations such as a multiply instruction) is shown in Figure 7-63.

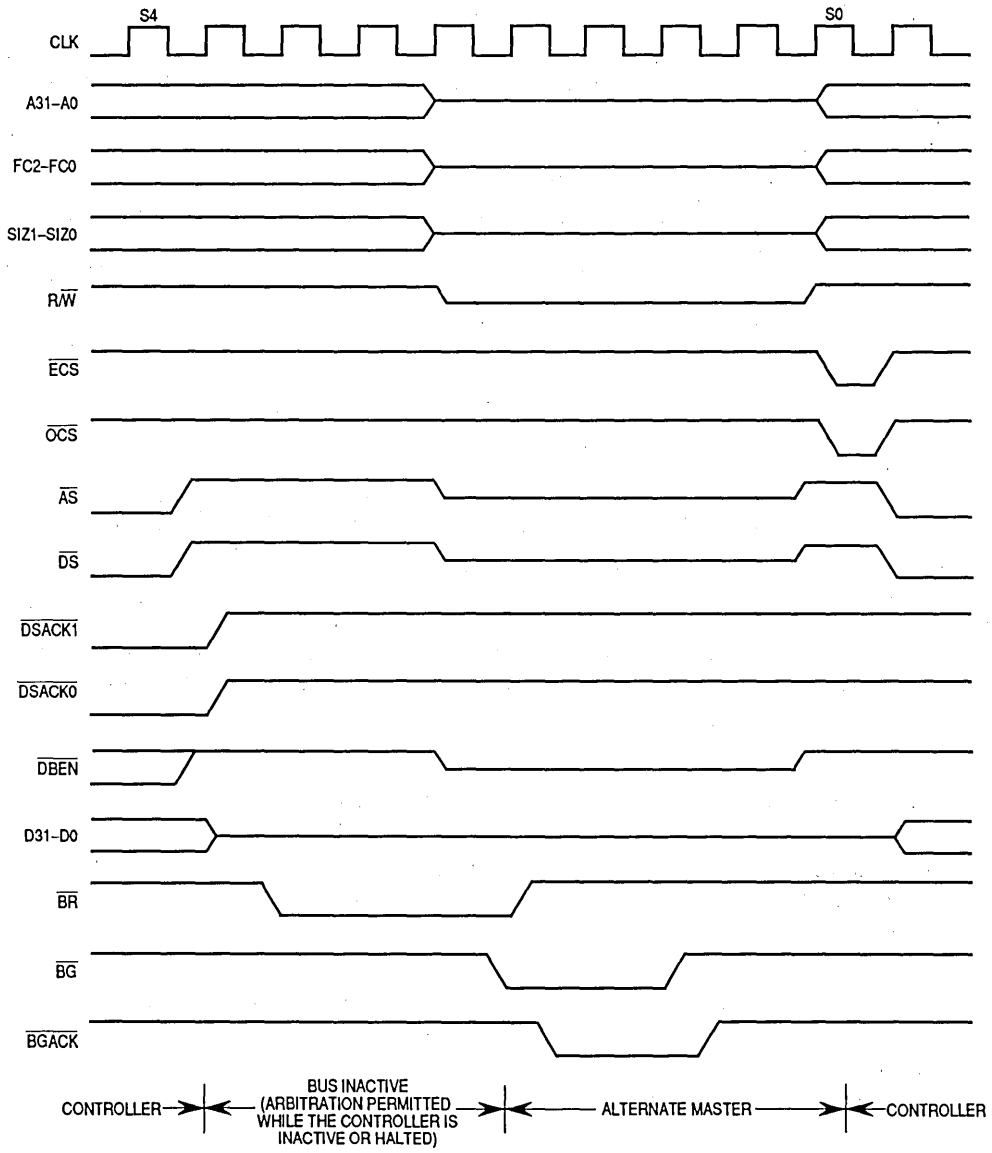


Figure 7-63. Bus Arbitration Operation (Bus Inactive)

7.8 RESET OPERATION

$\overline{\text{RESET}}$ is a bidirectional signal with which an external device resets the system or the controller resets external devices. When power is applied to the system, external circuitry should assert $\overline{\text{RESET}}$ for a minimum of 520 clocks after V_{CC} is within tolerance. Figure 7-64 is a timing diagram of the powerup reset operation, showing the relationships between $\overline{\text{RESET}}$, V_{CC}, and bus signals. The clock signal is required to be stable by the time V_{CC} reaches the minimum operating specification. During the reset period, the entire bus three-states (except for non-three-statable signals, which are driven to their inactive state). Once $\overline{\text{RESET}}$ negates, all control signals are driven to their inactive state, the data bus is in read mode, and the address bus is driven. After this, the first bus cycle for reset exception processing begins.

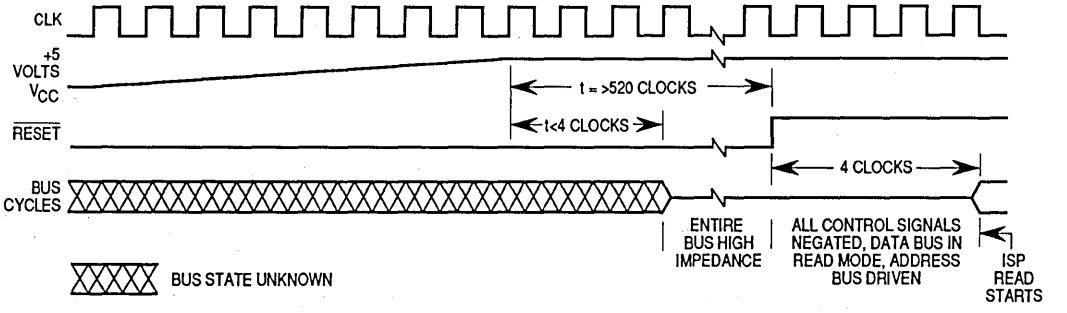


Figure 7-64. Initial Reset Operation Timing

The external $\overline{\text{RESET}}$ signal resets the controller and the entire system. Except for the initial reset, $\overline{\text{RESET}}$ should be asserted for at least 520 clock periods to ensure that the controller resets. Asserting $\overline{\text{RESET}}$ for 10 clock periods is sufficient for resetting the controller logic; the additional clock periods prevent a reset instruction from overlapping the external $\overline{\text{RESET}}$ signal.

Resetting the controller causes any bus cycle in progress to terminate as if $\overline{\text{DSACK}}$x, $\overline{\text{BERR}}$, or $\overline{\text{STERM}}$ had been asserted. In addition, the controller initializes registers appropriately for a reset exception. Exception processing for a reset operation is described in **8.1.1 Reset Exception**.

When a reset instruction is executed, the controller drives the $\overline{\text{RESET}}$ signal for 512 clock cycles. In this case, the controller resets the external devices of the system, and the internal registers of the controller are unaffected. The external devices connected to the $\overline{\text{RESET}}$ signal are reset at the completion of the reset instruction. An external $\overline{\text{RESET}}$ signal that is asserted to the

controller during execution of a reset instruction must extend beyond the reset period of the instruction by at least eight clock cycles to reset the controller. Figure 7-65 shows the timing information for the reset instruction.

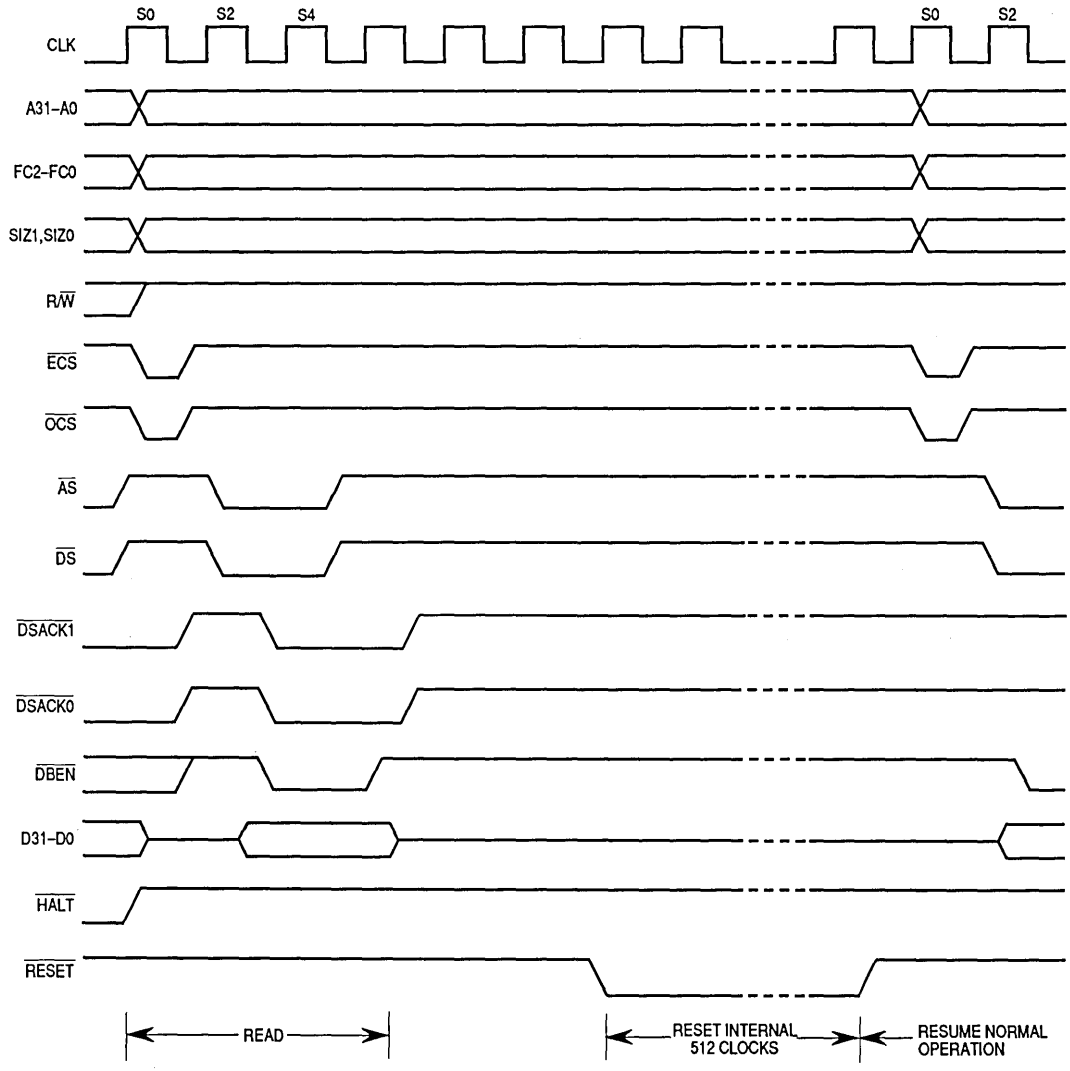


Figure 7-65. Processor-Generated Reset Operation

SECTION 8
EXCEPTION PROCESSING

Exception processing is defined as the activities performed by the controller in preparing to execute a handler routine for any condition that causes an exception. In particular, exception processing does not include execution of the handler routine itself. An introduction to exception processing, as one of the processing states of the MC68EC030 controller, was given in **SECTION 4 PROCESSING LEVELS**. This section describes exception processing in detail, describing the processing for each type of exception. It describes the return from an exception and bus fault recovery. For more detail on protocol violation and coprocessor-related exceptions, refer to **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION**. Also, for more detail on exceptions defined for floating-point coprocessors, refer to MC68881 UM/AD, *MC68881/MC68882 User's Manual*.

8.1 EXCEPTION PROCESSING SEQUENCE

Exception processing occurs in four functional steps. However, all individual bus cycles associated with exception processing (vector acquisition, stacking, etc.) are not guaranteed to occur in the order in which they are described in this section. Nonetheless, all addresses and offsets from the stack pointer are guaranteed to be as described.

The first step of exception processing involves the status register. The controller makes an internal copy of the status register. Then the controller sets the S bit, changing to the supervisor privilege level. Next, the controller inhibits tracing of the exception handler by clearing the T1 and T0 bits. For the reset and interrupt exceptions, the controller also updates the interrupt priority mask.

In the second step, the controller determines the vector number of the exception. For interrupts, the controller performs an interrupt acknowledge cycle (a read from the CPU address space type $F; see Figures 7-45 and 7-46) to obtain the vector number. For coprocessor-detected exceptions, the vector number is included in the coprocessor exception primitive response. (Refer to **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION** for a complete discussion of coprocessor exceptions.) For all other exceptions, internal

logic provides the vector number. This vector number is used in the last step to calculate the address of the exception vector. Throughout this section, vector numbers are given in decimal notation.

For all exceptions other than reset, the third step is to save the current controller context. The controller creates an exception stack frame on the active supervisor stack and fills it with context information appropriate for the type of exception. Other information may also be stacked, depending on which exception is being processed and the state of the controller prior to the exception. If the exception is an interrupt and the M bit of the status register is set, the controller clears the M bit in the status register and builds a second stack frame on the interrupt stack.

The last step initiates execution of the exception handler. The controller multiplies the vector number by four to determine the exception vector offset. It adds the offset to the value stored in the vector base register to obtain the memory address of the exception vector. Next, the controller loads the program counter (and the interrupt stack pointer (ISP) for the reset exception) from the exception vector table in memory. After prefetching the first three words to fill the instruction pipe, the controller resumes normal processing at the address in the program counter. Table 8-1 contains a description of all the exception vector offsets defined for the MC68EC030.

Table 8-1. Exception Vector Assignments

Vector Number(s)	Vector Offset		Assignment	$\overline{\text{STATUS}}$ Asserted
	Hex	Space		
0	000	SP	Reset Initial Interrupt Stack Pointer	—
1	004	SP	Reset Initial Program Counter	—
2	008	SD	Bus Error	Yes
3	00C	SD	Address Error	Yes
4	010	SD	Illegal Instruction	No
5	014	SD	Zero Divide	No
6	018	SD	CHK, CHK2 Instruction	No
7	01C	SD	cpTRAPcc, TRAPcc, TRAPV Instructions	No
8	020	SD	Privilege Violation	No
9	024	SD	Trace	Yes
10	028	SD	Line 1010 Emulator	No
11	02C	SD	Line 1111 Emulator	Yes
12	030	SD	(Unassigned, Reserved)	—
13	034	SD	Coprocessor Protocol Violation	No
14	038	SD	Format Error	No
15	03C	SD	Uninitialized Interrupt	Yes
16 Through 23	040 Through 05C	SD	Unassigned, Reserved	—

Table 8-1. Exception Vector Assignments (Continued)

Vector Number(s)	Vector Offset		Assignment	$\overline{\text{STATUS}}$ Asserted
	Hex	Space		
24	060	SD	Spurious Interrupt	Yes
25	064	SD	Level 1 Interrupt Autovector	Yes
26	068	SD	Level 2 Interrupt Autovector	Yes
27	06C	SD	Level 3 Interrupt Autovector	Yes
28	070	SD	Level 4 Interrupt Autovector	Yes
29	074	SD	Level 5 Interrupt Autovector	Yes
30	078	SD	Level 6 Interrupt Autovector	Yes
31	07C	SD	Level 7 Interrupt Autovector	Yes
32 Through 47	080 Through 0BC	SD SD	TRAP #0–15 Instruction Vectors	No
48	0C0	SD	FPCP Branch or Set on Unordered Condition	No
49	0C4	SD	FPCP Inexact Result	No
50	0C8	SD	FPCP Divide by Zero	No
51	0CC	SD	FPCP Underflow	No
52	0D0	SD	FPCP Operand Error	No
53	0D4	SD	FPCP Overflow	No
54	0D8	SD	FPCP Signaling NAN	No
55	0DC	SD	Unassigned, Reserved	No
56	0E0	SD	Defined for MC68030 not used by MC68EC030	No
57	0E4	SD	Defined for MC68851 not used by MC68EC030	No
58	0E8	SD	Defined for MC68851 not used by MC68EC030	No
59 Through 63	0EC Through 0FC	SD SD	Unassigned, Reserved	—
64 Through 255	100 Through 3FC	SD SD	User Defined Vectors (192)	Yes

SP = Supervisor Program Space
SD = Supervisor Data Space

As shown in Table 8-1, the first 64 vectors are defined by Motorola and 192 vectors are reserved for interrupt vectors defined by the user. However, external devices may use vectors reserved for internal purposes at the discretion of the system designer.

The MC68EC030 provides the $\overline{\text{STATUS}}$ signal to identify instruction boundaries and some exceptions. As shown in Table 8-2, $\overline{\text{STATUS}}$ indicates an instruction boundary and exceptions to be processed, depending on the state of the internal microsequencer. In addition, $\overline{\text{STATUS}}$ indicates when an ACU address translation cache miss has occurred and the controller is about to begin a table search access for the address that caused the miss. Instruction-related exceptions do not cause the assertion of $\overline{\text{STATUS}}$ as shown in Table

8-1. For $\overline{\text{STATUS}}$ signal timing information, refer to **SECTION 12 APPLICATIONS INFORMATION.**

Table 8-2. Microsequencer $\overline{\text{STATUS}}$ Indications

Asserted for	Indicates
1 Clock	Sequencer at instruction boundary will begin execution of next instruction.
2 Clocks	Sequencer at instruction boundary but will not begin the next instruction immediately due to: • pending trace exception OR • pending interrupt exception
3 Clocks	ACU address translation cache miss — controller to begin table serach OR Exception processing to begin for: • reset OR • bus error OR • address error OR • spurious interrupt OR • autovectored interrupt OR • F-line instruction (no coprocessor responded)
Continuously	Processor halted due to double bus fault.

8

8.1.1 Reset Exception

Assertion by external hardware of the $\overline{\text{RESET}}$ signal causes a reset exception. For details on the requirements for the assertion of $\overline{\text{RESET}}$, refer to **7.8 RESET OPERATION.**

The reset exception has the highest priority of any exception; it provides for system initialization and recovery from catastrophic failure. When reset is recognized, it aborts any processing in progress, and that processing cannot be recovered. Figure 8-1 is a flowchart of the reset exception, which performs the following operations:

1. Clears both trace bits in the status register to disable tracing.
2. Places the controller in the interrupt mode of the supervisor privilege level by setting the supervisor bit and clearing the master bit in the status register.
3. Sets the controller interrupt priority mask to the highest priority level (level 7).
4. Initializes the vector base register to zero ($00000000).
5. Clears the enable, freeze, and burst enable bits for both on-chip caches and the write-allocate bit for the data cache in the cache control register.

6. Invalidates all entries in the instruction and data caches.
7. Clears the enable bit in both access control registers of the ACU.
8. Generates a vector number to reference the reset exception vector (two long words) at offset zero in the supervisor program address space.
9. Loads the first long word of the reset exception vector into the interrupt stack pointer.
10. Loads the second long word of the reset exception vector into the program counter.

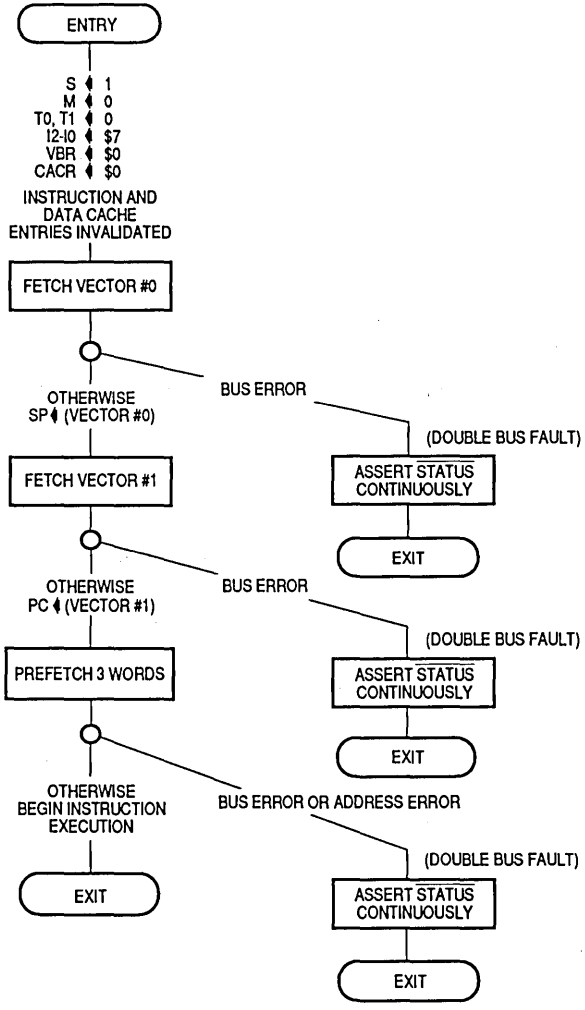


Figure 8-1. Reset Operation Flowchart

After the initial instruction prefetches, program execution begins at the address in the program counter. The reset exception does not save the value of either the program counter or the status register.

As described in **7.5.4 Double Bus Fault**, if bus error or address error occur during the exception processing sequence for a reset, a double bus fault occurs. The controller halts, and the $\overline{\text{STATUS}}$ signal is asserted continuously to indicate the halted condition.

Execution of the reset instruction does not cause a reset exception, nor does it affect any internal registers, but it does cause the MC68EC030 to assert the $\overline{\text{RESET}}$ signal, resetting all external devices.

8.1.2 Bus Error Exception

A bus error exception occurs when external logic aborts a bus cycle by asserting the $\overline{\text{BERR}}$ input signal. If the aborted bus cycle is a data access, the controller immediately begins exception processing. If the aborted bus cycle is an instruction prefetch, the controller may delay taking the exception until it attempts to use the prefetched information. The assertion of the $\overline{\text{BERR}}$ signal during the second, third, or fourth access of a burst operation does not cause a bus error exception, but the burst is aborted. Refer to **6.1.3.2 BURST MODE FILLING** and **7.5.1 Bus Errors** for details on the effects of bus errors during burst operation.

The controller begins exception processing for a bus error by making an internal copy of the current status register. The controller then enters the supervisor privilege level (by setting the S bit in the status register) and clears the trace bits. The controller generates exception vector number 2 for the bus error vector. It saves the vector offset, program counter, and the internal copy of the status register on the stack. The saved program counter value is the address of the instruction that was executing at the time the fault was detected. This is not necessarily the instruction that initiated the bus cycle, since the controller overlaps execution of instructions. The controller also saves the contents of some of its internal registers. The information saved on the stack is sufficient to identify the cause of the bus fault and recover from the error.

For efficiency, the MC68EC030 uses two different bus error stack frame formats. When the bus error exception is taken at an instruction boundary, less information is required to recover from the error, and the controller builds the short bus fault stack frame as shown in Table 8-7. When the exception is taken during the execution of an instruction, the controller must save its

entire state for recovery and uses the long bus fault stack frame shown in Table 8-7. The format code in the stack frame distinguishes the two stack frame formats. Stack frame formats are described in detail in **8.4 EXCEPTION STACK FRAME FORMATS.**

If a bus error occurs during the exception processing for a bus error, address error, or reset or while the controller is loading internal state information from the stack during the execution of an RTE instruction, a double bus fault occurs, and the controller enters the halted state as indicated by the continuous assertion of the $\overline{\text{STATUS}}$ signal. In this case, the controller does not attempt to alter the current state of memory. Only an external $\overline{\text{RESET}}$ can restart a controller halted by a double bus fault.

8.1.3 Address Error Exception

An address error exception occurs when the controller attempts to prefetch an instruction from an odd address. This exception is similar to a bus error exception, but is internally initiated. A bus cycle is not executed, and the controller begins exception processing immediately. After exception processing commences, the sequence is the same as that for bus error exceptions described in the preceding paragraphs, except that the vector number is 3 and the vector offset in the stack frame refers to the address error vector. Either a short or long bus fault stack frame may be generated. If an address error occurs during the exception processing for a bus error, address error, or reset, a double bus fault occurs.

8.1.4 Instruction Trap Exception

Certain instructions are used to explicitly cause trap exceptions. The TRAP #n instruction always forces an exception and is useful for implementing system calls in user programs. The TRAPcc, TRAPV, cpTRAPcc, CHK, and CHK2 instructions force exceptions if the user program detects an error, which may be an arithmetic overflow or a subscript value that is out of bounds.

The DIVS and DIVU instructions force exceptions if a division operation is attempted with a divisor of zero.

When a trap exception occurs, the controller copies the status register internally, enters the supervisor privilege level, and clears the trace bits. If tracing is enabled for the instruction that caused the trap, a trace exception is taken after the RTE instruction from the trap handler is executed, and the trace corresponds to the trap instruction; the trap handler routine is not

traced. The controller generates a vector number according to the instruction being executed; for the TRAP #n instruction, the vector number is 32 plus n. The stack frame saves the trap vector offset, the program counter, and the internal copy of the status register on the supervisor stack. The saved value of the program counter is the address of the instruction following the instruction that caused the trap. For all instruction traps other than TRAP #n, a pointer to the instruction that caused the trap is also saved. Instruction execution resumes at the address in the exception vector after the required instruction prefetches.

8.1.5 Illegal Instruction and Unimplemented Instruction Exceptions

An illegal instruction is an instruction that contains any bit pattern in its first word that does not correspond to the bit pattern of the first word of a valid MC68EC030 instruction or is a MOVEC instruction with an undefined register specification field in the first extension word. An illegal instruction exception corresponds to vector number 4 and occurs when the controller attempts to execute an illegal instruction.

An illegal instruction exception is also taken if a breakpoint acknowledge bus cycle (see **7.4.2 Breakpoint Acknowledge Cycle**) is terminated with the assertion of the bus error signal. This implies that the external circuitry did not supply an instruction word to replace the BKPT instruction word in the instruction pipe.

Instruction word patterns with bits [15:12] equal to $A are referred to as unimplemented instructions with A-line opcodes. When the controller attempts to execute an unimplemented instruction with an A-line opcode, an exception is generated with vector number 10, permitting efficient emulation of unimplemented instructions.

Instructions that have word patterns with bits [15:12] equal to $F, bits [11:9] equal to $0, and defined word patterns for subsequent words are legal ACU instructions. Execution of some of these instructions by the MC68EC030 can cause undefined results. They are not treated as unimplemented instructions. Refer to **SECTION 9 ACCESS CONTROL UNIT** for more details. Instructions that have bits [15:12] of the first words equal to $F, bits [11:9] equal to $0, and undefined patterns in subsequent words are treated as unimplemented instructions with F-line opcodes when execution is attempted in supervisor mode. When execution of the same instruction is attempted in user mode, a privilege violation exception is taken. The exception vector number for an unimplemented instruction with an F-line opcode is number 11.

The word patterns with bits [15:12] equal to $F and bits [11:9] not equal to zero are used for coprocessor instructions. When the controller identifies a coprocessor instruction, it runs a bus cycle referencing CPU space type $2 (refer to **4.2 ADDRESS SPACE TYPES**) and addressing one of seven coprocessors (1–7, according to bits [11:9]). If the addressed coprocessor is not included in the system and the cycle terminates with the assertion of the bus error signal, the instruction takes an unimplemented instruction (F-line opcode) exception. The system can emulate the functions of the coprocessor with an F-line exception handler. Refer to **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION** for more details.

Exception processing for illegal and unimplemented instructions is similar to that for instruction traps. When the controller has identified an illegal or unimplemented instruction, it initiates exception processing instead of attempting to execute the instruction. The controller copies the status register, enters the supervisor privilege level, and clears the trace bits, disabling further tracing. The controller generates the vector number, either 4, 10, or 11, according to the exception type. The illegal or unimplemented instruction vector offset, current program counter, and copy of the status register are saved on the supervisor stack, with the saved value of the program counter being the address of the illegal or unimplemented instruction. Instruction execution resumes at the address contained in the exception vector. It is the responsibility of the handling routine to adjust the stacked program counter if the instruction is emulated in software or is to be skipped on return from the handler.

8.1.6 Privilege Violation Exception

To provide system security, the following instructions are privileged:

ANDI TO SR
EOR to SR
cpRESTORE
cpSAVE
MOVE from SR
MOVE to SR
MOVE USP
MOVEC
MOVES
ORI to SR
PMOVE
PTEST
RESET
RTE
STOP

An attempt to execute one of the privileged instructions while at the user privilege level causes a privilege violation exception. Also, a privilege violation exception occurs if a coprocessor requests a privilege check and the controller is at the user level.

Exception processing for privilege violations is similar to that for illegal instructions. When the controller identifies a privilege violation, it begins exception processing before executing the instruction. The controller copies the status register, enters the supervisor privilege level, and clears the trace bits. The controller generates vector number 8, the privilege violation exception vector, and saves the privilege violation vector offset, the current program counter value, and the internal copy of the status register on the supervisor stack. The saved value of the program counter is the address of the first word of the instruction that caused the privilege violation. Instruction execution resumes after the required prefetches from the address in the privilege violation exception vector.

8.1.7 Trace Exception

To aid in program development, the M68000 processors include instruction-by-instruction tracing capability. The MC68EC030 can be programmed to trace all instructions or only instructions that change program flow. In the trace mode, an instruction generates a trace exception after it completes execution, allowing a debugger program to monitor execution of a program.

The T1 and T0 bits in the supervisor portion of the status register control tracing. The state of these bits when an instruction begins execution determines whether the instruction generates a trace exception after the instruction completes. Clearing both T bits disables tracing, and instruction execution proceeds normally. Clearing the T1 bit and setting the T0 bit causes an instruction that forces a change of flow to take a trace exception. Instructions that increment the program counter normally do not take the trace exception. Instructions that are traced in this mode include all branches, jumps, instruction traps, returns, and coprocessor instructions that modify the program counter flow. This mode also includes status register manipulations, because the controller must re-prefetch instruction words to fill the pipe again any time an instruction that can modify the status register is executed. The execution of the BKPT instruction causes a change of flow if the opcode replacing the BKPT is an instruction that causes a change of flow (i.e., a jump, branch, etc.). Setting the T1 bit and clearing the T0 bit causes the execution of all instructions to force trace exceptions. Table 8-3 shows the trace mode selected by each combination of T1 and T0.

Table 8-3. Tracing Control

T1	T0	Tracing Function
0	0	No Tracing
0	1	Trace on Change of Flow (BRA, JMP, etc.)
1	0	Trace on Instruction Execution (Any Instruction)
1	1	Undefined, Reserved

In general terms, a trace exception is an extension to the function of any traced instruction — that is, the execution of a traced instruction is not complete until the trace exception processing is completed. If an instruction does not complete due to a bus error or address error exception, trace exception processing is deferred until after the execution of the suspended instruction is resumed and the instruction execution completes normally. If an interrupt is pending at the completion of an instruction, the trace exception processing occurs before the interrupt exception processing starts. If an instruction forces an exception as part of its normal execution, the forced exception processing occurs before the trace exception is processed. See **8.1.12 Multiple Exceptions** for a more complete discussion of exception priorities.

When the controller is in the trace mode and attempts to execute an illegal or unimplemented instruction, that instruction does not cause a trace exception since it is not executed. This is of particular importance to an instruction emulation routine that performs the instruction function, adjusts the stacked program counter to skip the unimplemented instruction, and returns. Before returning, the trace bits of the status register on the stack should be checked. If tracing is enabled, the trace exception processing should also be emulated for the trace exception handler to account for the emulated instruction.

The exception processing for a trace starts at the end of normal processing for the traced instruction and before the start of the next instruction. The controller makes an internal copy of the status register and enters the supervisor privilege level. It also clears the T0 and T1 bits of the status register, disabling further tracing. The controller supplies vector number 9 for the trace exception and saves the trace exception vector offset, program counter value, and the copy of the status register on the supervisor stack. The saved value of the program counter is the address of the next instruction to be executed. Instruction execution resumes after the required prefetches from the address in the trace exception vector.

The STOP instruction does not perform its function when it is traced. A STOP instruction that begins execution with T1 = 1 and T0 = 0 forces a trace exception after it loads the status register. Upon return from the trace handler routine, execution continues with the instruction following the STOP, and the controller never enters the stopped condition.

8.1.8 Format Error Exception

Just as the controller checks that prefetched instructions are valid, the controller (with the aid of a coprocessor, if needed) also performs some checks of data values for control operations, including the coprocessor state frame format word for a cpRESTORE instruction and the stack frame format for an RTE instruction.

The RTE instruction checks the validity of the stack format code. For long bus cycle fault format frames, the RTE instruction also compares the internal version number of the controller to that contained in the frame at memory location SP + 54 (SP + $36). This check ensures that the controller can correctly interpret internal state information from the stack frame.

The cpRESTORE instruction passes the format word of the coprocessor state frame to the coprocessor for validation. If the coprocessor does not recognize the format value, it signals the MC68EC030 to take a format error exception. Refer to **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION** for details of coprocessor-related exceptions.

If any of the checks previously described determine that the format of the stacked data is improper, the instruction generates a format error exception. This exception saves a short format stack frame, generates exception vector number 14, and continues execution at the address in the format exception vector. The stacked program counter value is the address of the instruction that detected the format error.

8.1.9 Interrupt Exceptions

When a peripheral device requires the services of the MC68EC030 or is ready to send information that the controller requires, it may signal the controller to take an interrupt exception. The interrupt exception transfers control to a routine that responds appropriately.

The peripheral device uses the active-low interrupt priority level signals ($\overline{IPL0}$–$\overline{IPL2}$) to signal an interrupt condition to the controller and to specify

the priority of that condition. The three signals encode a value of zero through seven ($\overline{IPL0}$ is the least significant bit). High levels on all three signals correspond to no interrupt requested (level 0) and low levels on $\overline{IPL0}$–$\overline{IPL2}$ correspond to interrupt request level 7. Values 1–7 specify one of seven levels of prioritized interrupts; level seven has the highest priority. External circuitry can chain or otherwise merge signals from devices at each level, allowing an unlimited number of devices to interrupt the controller.

The $\overline{IPL0}$–$\overline{IPL2}$ interrupt signals must maintain the interrupt request level until the MC68EC030 acknowledges the interrupt to guarantee that the interrupt is recognized. The MC68EC030 continuously samples the $\overline{IPL0}$–$\overline{IPL2}$ signals on consecutive falling edges of the controller clock to synchronize and debounce these signals. An interrupt request that is the same for two consecutive falling clock edges is considered a valid input. Although the protocol requires that the request remain until the controller runs an interrupt acknowledge cycle for that interrupt value, an interrupt request that is held for as short a period as two clock cycles could be recognized.

The status register of the MC68EC030 contains an interrupt priority mask (I2, I1, I0, bits 10–8). The value in the interrupt mask is the highest priority level that the controller ignores. When an interrupt request has a priority higher than the value in the mask, the controller makes the request a pending interrupt. Figure 8-2 is a flowchart of the procedure for making an interrupt pending.

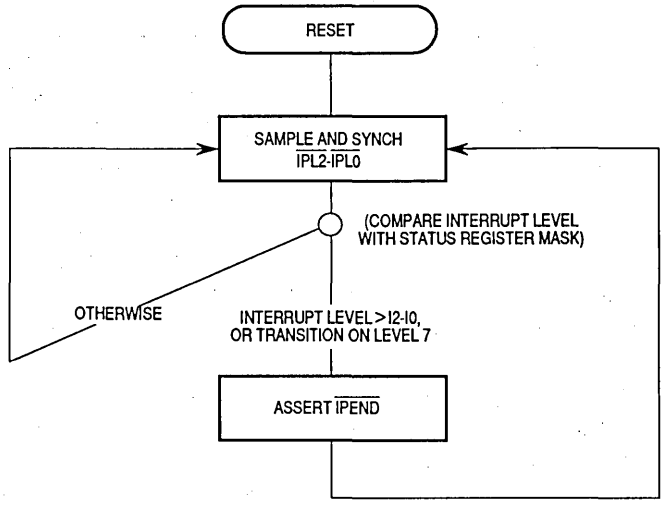


Figure 8-2. Interrupt Pending Procedure

When several devices are connected to the same interrupt level, each device should hold its interrupt priority level constant until its corresponding interrupt acknowledge cycle to ensure that all requests are processed.

Table 8-4 lists the interrupt levels, the states of $\overline{IPL2}$–$\overline{IPL0}$ that define each level, and the mask value that allows an interrupt at each level.

Table 8-4. Interrupt Levels and Mask Values

Requested Interrupt Level	Control Line Status			Interrupt Mask Level Required for Recognition
	$\overline{IP2}$	$\overline{IP1}$	$\overline{IP0}$	
0*	High	High	High	N/A*
1	High	High	Low	0
2	High	Low	High	0–1
3	High	Low	Low	0–2
4	Low	High	High	0–3
5	Low	High	Low	0–4
6	Low	Low	High	0–5
7	Low	Low	Low	0–7

*Indicates that no interrupt is requested.

Priority level 7, the nonmaskable interrupt (NMI), is a special case. Level 7 interrupts cannot be masked by the interrupt priority mask, and they are transition sensitive. The controller recognizes an interrupt request each time the external interrupt request level changes from some lower level to level 7, regardless of the value in the mask. Figure 8-3 shows two examples of interrupt recognitions, one for level 6 and one for level 7. When the MC68EC030 processes a level 6 interrupt, the status register mask is automatically updated with a value of 6 before entering the handler routine so that subsequent level 6 interrupts are masked. Provided no instruction that lowers the mask value is executed, the external request can be lowered to level 3 and then raised back to level 6 and a second level 6 interrupt is not processed. However, if the MC68EC030 is handling a level 7 interrupt (status register mask set to 7) and the external request is lowered to level 3 and then raised back to level 7, a second level 7 interrupt is processed. The second level 7 interrupt is processed because the level 7 interrupt is transition sensitive. A level 7 interrupt is also generated by a level comparison if the request level and mask level are at seven and the priority mask is then set to a lower level (with the MOVE to SR or RTE instruction, for example). As shown in Figure 8-3 for level 6 interrupt request level and mask level, this is the case for all interrupt levels.

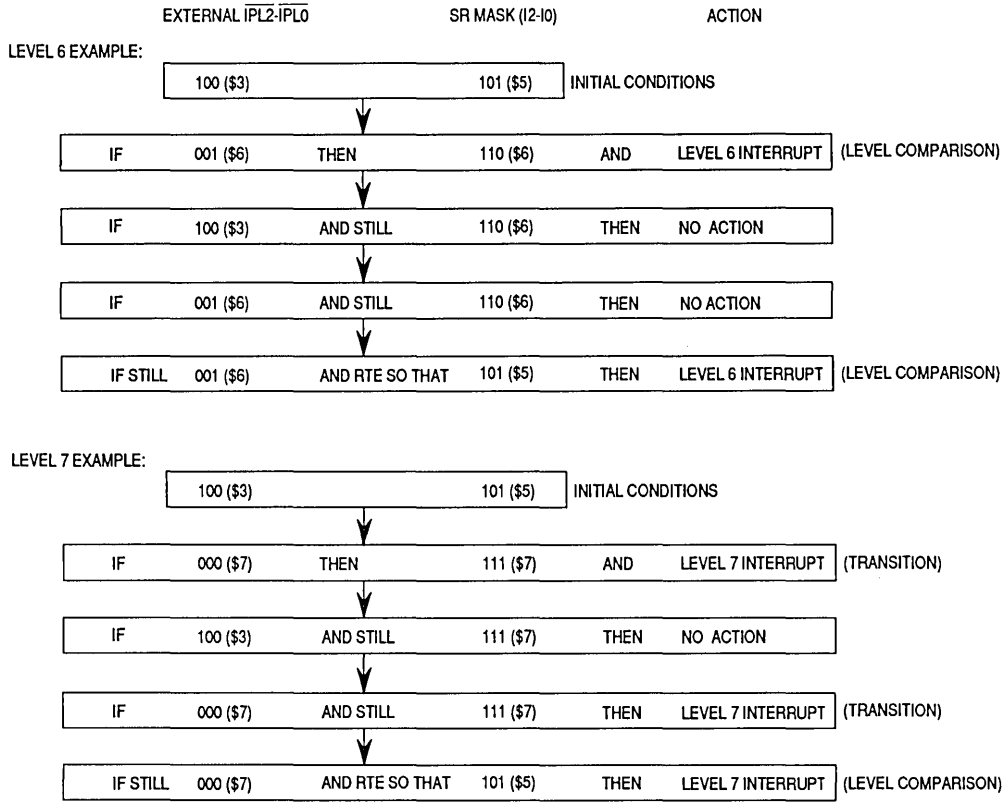


Figure 8-3. Interrupt Recognition Examples

Note that a mask value of 6 and a mask value of 7 both inhibit request levels 1–6 from being recognized. In addition, neither masks a transition to an interrupt request level of 7. The only difference between mask values of 6 and 7 occurs when the interrupt request level is 7 and the mask value is 7. If the mask value is lowered to 6, a second level 7 interrupt is recognized.

The MC68EC030 asserts the interrupt pending signal ($\overline{\text{IPEND}}$) when it makes an interrupt request pending. Figure 8-4 shows the assertion of $\overline{\text{IPEND}}$ relative to the assertion of an interrupt level on the $\overline{\text{IPLx}}$ lines. $\overline{\text{IPEND}}$ signals to external devices that an interrupt exception will be taken at an upcoming instruction boundary (following any higher priority exception).

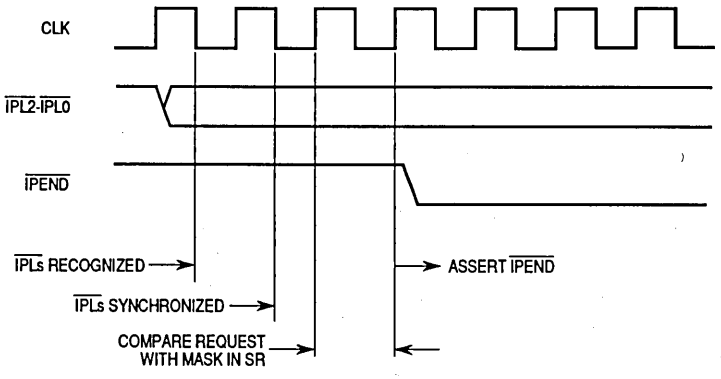


Figure 8-4. Assertion of IPEND

The state of the IPEND signal is internally checked by the controller once per instruction, independently of bus operation. In addition, it is checked during the second instruction prefetch associated with exception processing. Figure 8-5 is a flowchart of the interrupt recognition and associated exception processing sequence.

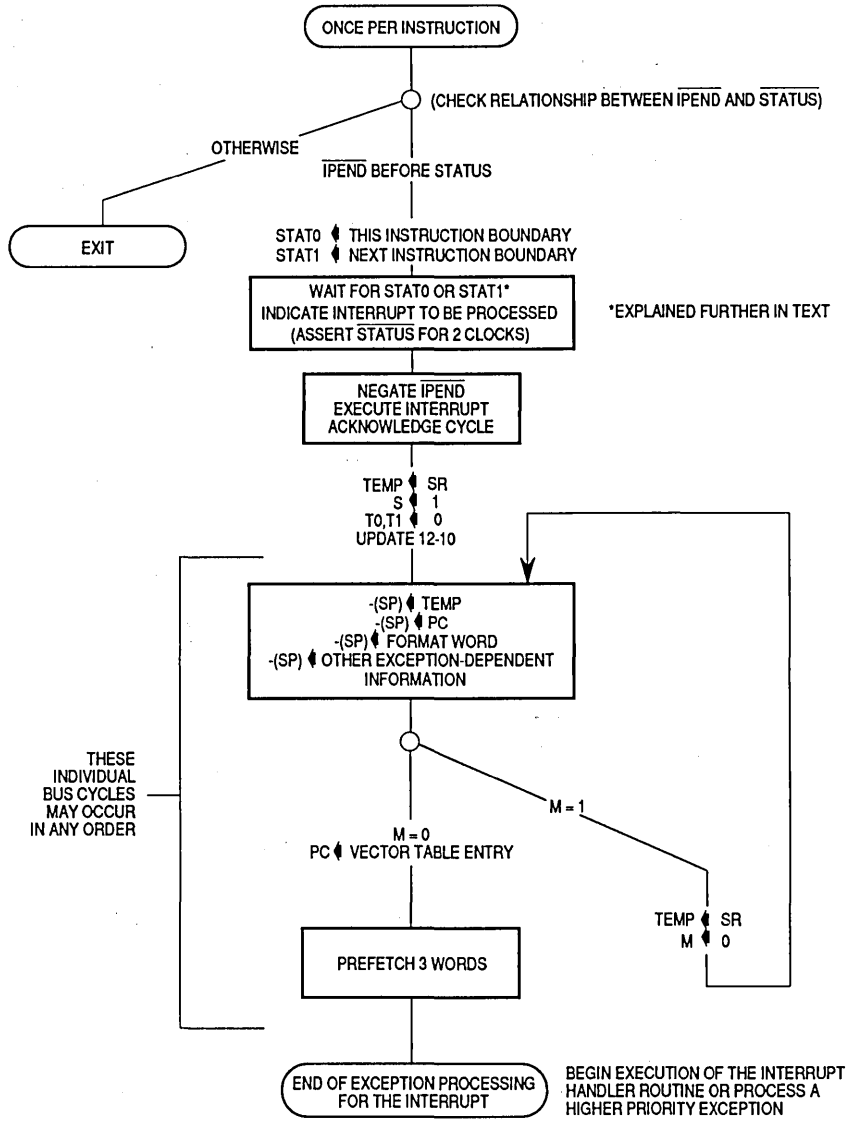
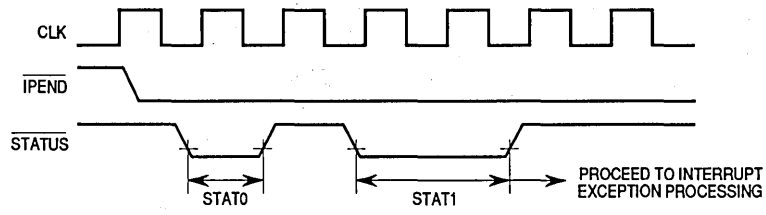


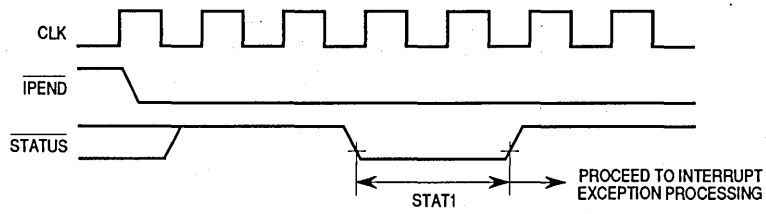
Figure 8-5. Interrupt Exception Processing Flowchart

To predict the instruction boundary during which a pending interrupt is processed, the timing relationship between the assertion of $\overline{\text{IPEND}}$ for that interrupt and the assertion of $\overline{\text{STATUS}}$ must be examined. Figure 8-6 shows two examples of interrupt recognition. The first assertion of $\overline{\text{STATUS}}$ after $\overline{\text{IPEND}}$ is denoted as STAT0. The next assertion of $\overline{\text{STATUS}}$ is denoted as

STAT1. If STAT0 begins on the falling edge of the clock immediately following the clock edge that caused $\overline{\text{IPEND}}$ to assert (as shown in example 1), STAT1 is at least two clocks long, and, when there are no other pending exceptions, the interrupt is acknowledged at the boundary defined by STAT1. If $\overline{\text{IPEND}}$ is asserted with more setup time to STAT0, the interrupt may be acknowledged at the boundary defined by STAT0 (as shown in example 2). In that case, STAT0 is asserted for two clocks, signaling this condition.



EXAMPLE 1: INTERRUPT EXCEPTION SIGNALED DURING STAT1



EXAMPLE 2: INTERRUPT EXCEPTION SIGNALED DURING STAT0

Figure 8-6. Examples of Interrupt Recognition and Instruction Boundaries

If no higher priority interrupt has been synchronized, the $\overline{\text{IPEND}}$ signal is negated during state 0 (S0) of an interrupt acknowledge cycle (refer to **7.4.1.1 INTERRUPT ACKNOWLEDGE CYCLE — TERMINATED NORMALLY**), and the $\overline{\text{IPLx}}$ signals for the interrupt being acknowledged can be negated at this time.

When processing an interrupt exception, the controller first makes an internal copy of the status register, sets the privilege level to supervisor, suppresses tracing, and sets the controller interrupt mask level to the level of the interrupt being serviced. The controller attempts to obtain a vector number from the interrupting device using an interrupt acknowledge bus cycle with the interrupt level number output on pins A1–A3 of the address bus. For a device that cannot supply an interrupt vector, the autovector signal ($\overline{\text{AVEC}}$) can be

asserted, and the MC68EC030 uses an internally generated autovector, which is one of vector numbers 25–31, that corresponds to the interrupt level number. If external logic indicates a bus error during the interrupt acknowledge cycle, the interrupt is considered spurious, and the controller generates the spurious interrupt vector number, 24. Refer to **7.4.1 Interrupt Acknowledge Bus Cycles** for complete interrupt bus cycle information.

Once the vector number is obtained, the controller saves the exception vector offset, program counter value, and the internal copy of the status register on the active supervisor stack. The saved value of the program counter is the address of the instruction that would have been executed had the interrupt not occurred. If the interrupt was acknowledged during the execution of a coprocessor instruction, further internal information is saved on the stack so that the MC68EC030 can continue executing the coprocessor instruction when the interrupt handler completes execution.

If the M bit of the status register is set, the controller clears the M bit and creates a throwaway exception stack frame on top of the interrupt stack as part of interrupt exception processing. This second frame contains the same program counter value and vector offset as the frame created on top of the master stack, but has a format number of 1 instead of 0 or 9. The copy of the status register saved on the throwaway frame is exactly the same as that placed on the master stack except that the S bit is set in the version placed on the interrupt stack. (It may or may not be set in the copy saved on the master stack.) The resulting status register (after exception processing) has the S bit set and the M bit cleared.

The controller loads the address in the exception vector into the program counter, and normal instruction execution resumes after the required prefetches for the interrupt handler routine.

Most M68000 Family peripherals use programmable interrupt vector numbers as part of the interrupt request/acknowledge mechanism of the system. If this vector number is not initialized after reset and the peripheral must acknowledge an interrupt request, the peripheral usually returns the vector number for the uninitialized interrupt vector, 15.

8.1.10 Breakpoint Instruction Exception

To use the MC68EC030 in a hardware emulator, it must provide a means of inserting breakpoints in the emulator code and of performing appropriate operations at each breakpoint. For the MC68000 and MC68008, this can be done by inserting an illegal instruction at the breakpoint and detecting the

illegal instruction exception from its vector location. However, since the vector base register on the MC68010, MC68020, MC68030, and MC68EC030 allows arbitrary relocation of exception vectors, the exception address cannot reliably identify a breakpoint. The MC68020, MC68030, and MC68EC030 provide a breakpoint capability with a set of breakpoint instructions, $4848–$484F, for eight unique breakpoints. The breakpoint facility also allows external hardware to monitor the execution of a program residing in the on-chip instruction cache without severe performance degradation.

When the MC68EC030 executes a breakpoint instruction, it performs a breakpoint acknowledge cycle (read cycle) from CPU space type $0 with address lines A2–A4 corresponding to the breakpoint number. Refer to Figure 7-44 for the CPU space type $0 addresses and to **7.4.2 Breakpoint Acknowledge Cycle** for a description of the breakpoint acknowledge cycle. The external hardware can return either $\overline{\text{BERR}}$, $\overline{\text{DSACK}}$x, or $\overline{\text{STERM}}$ with an instruction word on the data bus. If the bus cycle terminates with $\overline{\text{BERR}}$, the controller performs illegal instruction exception processing. If the bus cycle terminates with $\overline{\text{DSACK}}$x or $\overline{\text{STERM}}$, the controller uses the data returned to replace the breakpoint instruction in the internal instruction pipe and begins execution of that instruction. The remainder of the pipe remains unaltered. In addition, no stacking or vector fetching is involved with the execution of the instruction. Figure 8-7 is a flowchart of the breakpoint instruction execution.

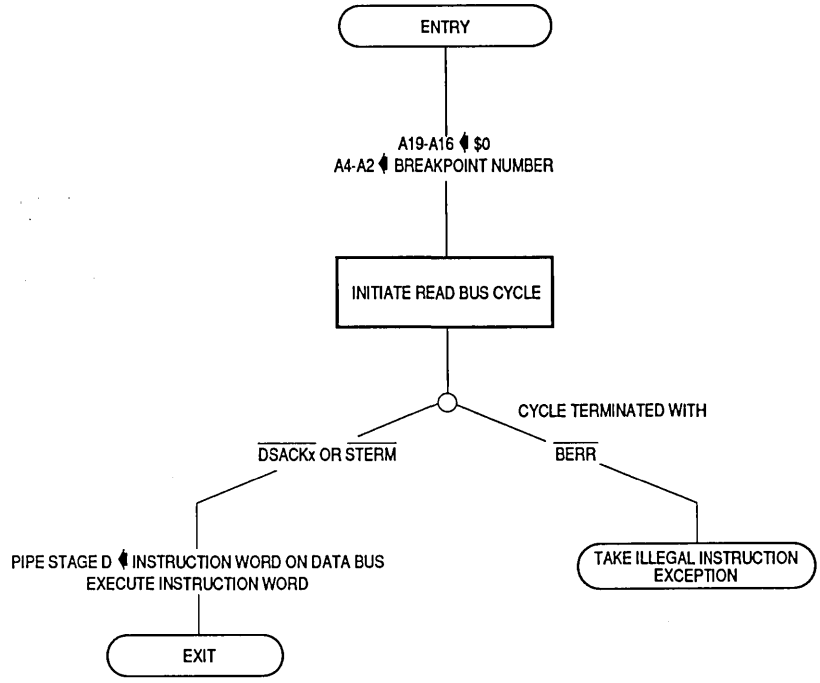


Figure 8-7. Breakpoint Instruction Flowchart

8

8.1.11 Multiple Exceptions

When several exceptions occur simultaneously, they are processed according to a fixed priority. Table 8-5 lists the exceptions, grouped by characteristics. Each group has a priority from 0–4. Priority 0 has the highest priority.

Table 8-5. Exception Priority Groups

Group/ Priority	Exception and Relative Priority	Characteristics
0	0.0 — Reset	Aborts all processing (instruction or exception) and does not save old context.
1	1.0 — Address Error 1.1 — Bus Error	Suspends processing (instruction or exception) and saves internal context.
2	2.0 — BKPT #n, CHK, CHK2, cp Mid-Instruction, cp Protocol Violation, cpTRAPcc, Divide by Zero, RTE, TRAP #n, TRAPV, ACU Configuration	Exception processing is part of instruction execution.
3	3.0 — Illegal Instruction, Line A, Unimplemented Line F, Privilege Violation, cp Pre-Instruction	Exception processing begins before instruction is executed.
4	4.0 — cp Post-Instruction 4.1 — Trace 4.2 — Interrupt	Exception processing begins when current instruction or previous exception processing is completed.

0.0 is the highest priority, 4.2 is the lowest.

As soon as the MC68EC030 has completed exception processing for a condition when another exception is pending, it begins exception processing for the pending exception instead of executing the exception handler for the original exception condition. Also, whenever a bus error or address error occurs, its exception processing takes precedence over lower priority exceptions and occurs immediately. For example, if a bus error occurs during the exception processing for a trace condition, the system processes the bus error and executes its handler before completing the trace exception processing. However, most exceptions cannot occur during exception processing, and very few combinations of the exceptions shown in Table 8-5 can be pending simultaneously.

The priority scheme is very important in determining the order in which exception handlers execute when several exceptions occur at the same time. As a general rule, the lower the priority of an exception, the sooner the handler routine for that exception executes. For example, if simultaneous trap, trace, and interrupt exceptions are pending, the exception processing for the trap occurs first, followed immediately by exception processing for the trace and then for the interrupt. When the controller resumes normal instruction execution, it is in the interrupt handler, which returns to the trace handler, which returns to the trap exception handler. This rule does not apply to the reset exception; its handler is executed first even though it has the highest priority because the reset operation clears all other exceptions.

8.1.12 Return from Exception

After the controller has completed exception processing for all pending exceptions, the controller resumes normal instruction execution at the address in the vector for the last exception processed. Once the exception handler has completed execution, the controller must return to the system context prior to the exception (if possible). The RTE instruction returns from the handler to the previous system context for any exception.

When the controller executes an RTE instruction, it examines the stack frame on top of the active supervisor stack to determine if it is a valid frame and what type of context restoration it requires. This section describes the processing for each of the stack frame types; refer to **8.3 COPROCESSOR CONSIDERATIONS** for a description of the stack frame type formats.

For a normal four-word frame, the controller updates the status register and program counter with the data read from the stack, increments the stack pointer by eight, and resumes normal instruction execution.

For the throwaway four-word stack, the controller reads the status register value from the frame, increments the active stack pointer by eight, updates the status register with the value read from the stack, and then begins RTE processing again, as shown in Figure 8-8. The controller reads a new format word from the stack frame on top of the active stack (which may or may not be the same stack used for the previous operation) and performs the proper operations corresponding to that format. In most cases, the throwaway frame is on the interrupt stack and when the status register value is read from the stack, the S and M bits are set. In that case, there is a normal four-word frame or a ten-word coprocessor mid-instruction frame on the master stack. However, the second frame may be any format (even another throwaway frame) and may reside on any of the three system stacks.

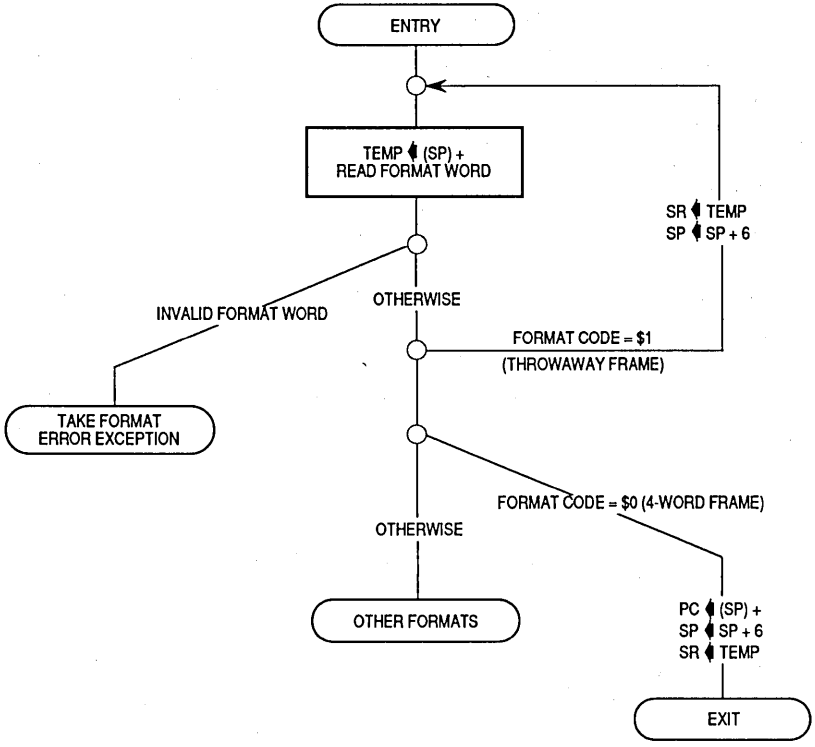


Figure 8-8. RTE Instruction for Throwaway Four-Word Frame

For the six-word stack frame, the controller restores the status register and program counter values from the stack, increments the active supervisor stack pointer by 12, and resumes normal instruction execution.

For the coprocessor mid-instruction stack frame, the controller reads the status register, program counter, instruction address, internal register values, and the evaluated effective address from the stack, restores these values to the corresponding internal registers, and increments the stack pointer by 20. The controller then reads from the response register of the coprocessor that initiated the exception to determine the next operation to be performed. Refer to **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION** for details of coprocessor-related exceptions.

For both the short and long bus fault stack frames, the controller first checks the format value on the stack for validity. In addition, for the long stack frame, the controller compares the version number in the stack with its own version

number. The version number is located in the most significant nibble (bits 15–12) of the word at location SP+$36 in the long stack frame. This validity check is required in a multiprocessor system to ensure that the data is properly interpreted by the RTE instruction. The RTE instruction also reads from both ends of the stack frame to make sure it is accessible. If the frame is invalid or inaccessible, the controller takes a format error or a bus error exception, respectively. Otherwise, the controller reads the entire frame into the proper internal registers, deallocates the stack, and resumes normal processing. Once the controller begins to load the frame to restore its internal state, the assertion of the $\overline{\text{BERR}}$ signal causes the controller to enter the halted state with the continuous assertion of the $\overline{\text{STATUS}}$ signal. Refer to **8.2 BUS FAULT RECOVERY** for a description of the processing that occurs after the frame is read into the internal registers.

If a format error or bus error exception occurs during the frame validation sequence of the RTE instruction, either due to any of the errors previously described or due to an illegal format code, the controller creates a normal four-word or a bus fault stack frame below the frame that it was attempting to use. In this way, the faulty stack frame remains intact. The exception handler can examine or repair the faulty frame. In a multiprocessor system, the faulty frame can be left to be used by another controller of a different type when appropriate.

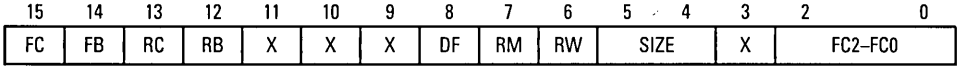
8.2 BUS FAULT RECOVERY

An address error exception or a bus error exception indicates a bus fault. The saving of the controller state for a bus error or address error is described in **8.1.2 Bus Error Exception**, and the restoring of the controller state by an RTE instruction is described in **8.1.13 Return from Exception**.

Controller accesses of either data items or the instruction stream can result in bus errors. When a bus error exception occurs while accessing a data item, the exception is taken immediately after the bus cycle terminates. A bus error occurring during an instruction stream access is not processed until the controller attempts to use the information (if ever) that the access should have provided. For instruction faults, when the short format frame applies, the address of the pipe stage B word is the value in the program counter plus four, and the address of the stage C word is the value in the program counter plus two. For the long format, the long word at SP+$24 contains the address of the stage B word; the address of the stage C word is the address of the stage B word minus two. Address error faults occur only for instruction stream accesses, and the exceptions are taken before the bus cycles are attempted.

8.2.1 Special Status Word (SSW)

The internal SSW (see Figure 8-9) is one of several registers saved as part of the bus fault exception stack frame. Both the short bus cycle fault format and the long bus cycle fault format include this word at offset $A. The bus cycle fault stack frame formats are described in detail at the end of this section.



FC —Fault on stage C of the instruction pipe
FB —Fault on stage B of the instruction pipe
RC —Rerun flag for stage C of the instruction pipe*
RB —Rerun flag for stage B of the instruction pipe*
DF —Fault/rerun flag for data cycle*
RM —Read-modify-write on data cycle
RW —Read/write for data cycle — 1 = read, 0 = write
SIZE —Size code for data cycle
FC2–FC0 —Address space for data cycle

*1 = Rerun Faulted bus Cycle, or run pending prefetch
0 = Do not rerun bus sycle
X = For internal use only

Figure 8-9. Special Status Word (SSW)

8

The SSW information indicates whether the fault was caused by an access to the instruction stream, data stream, or both. The high-order half of the SSW contains two status bits each for the B and C stages of the instruction pipe. The fault bits (FB and FC) indicate that the controller attempted to use a stage (B or C) and found it to be marked invalid due to a bus error on the prefetch for that stage. The fault bits can be used by a bus error handler to determine the cause(s) of a bus error exception. The rerun flag bits (RB and RC) are set to indicate that a fault occurred during a prefetch for the corresponding stage. A rerun bit is always set when the corresponding fault bit is set. The rerun bits indicate that the word in a stage of the instruction pipe is invalid, and the state of the bits can be used by a handler to repair the values in the pipe after an address error or a bus error, if necessary. If a rerun bit is set when the controller executes an RTE instruction, the controller may execute a bus cycle to prefetch the instruction word for the corresponding stage of the pipe (if it is required). If the rerun and fault bits are set for a stage of the pipe, the RTE instruction automatically reruns the prefetch cycle for that stage. The address space for the bus cycle is the program space for the privilege level indicated in the copy of the status register on the stack. If a rerun bit is cleared, the words on the stack for the corresponding stages of the pipe are accepted as valid; the controller assumes that there is no prefetch pending for the corresponding stage and that software has repaired or filled the image of the stage, if necessary.

If an address error exception occurs, the fault bits written to the stack frame are not set (they are only set due to a bus error, as previously described), and the rerun bits alone show the cause of the exception. Depending on the state of the pipeline, either RB and RC are both set, or RC alone is set. To correct the pipeline contents and continue execution of the suspended instruction, software must place the correct instruction stream data in the stage C and/or stage B images requested by the rerun bits and clear the rerun bits. The least significant half of the SSW applies to data cycles only. If the DF bit of the SSW is set, a data fault has occurred and caused the exception. If the DF bit is set when the controller reads the stack frame, it reruns the faulted data access; otherwise, it assumes that the data input buffer value on the stack is valid for a read or that the data has been correctly written to memory for a write (or that no data fault occurred). The RM bit of the SSW identifies a read-modify-write operation and the RW bit indicates whether the cycle was a read or write operation. The SIZE field indicates the size of the operand access, and the FC field specifies the address space for the data cycle. Data and instruction stream faults may be pending simultaneously; the fault handler should be able to recognize any combination of the FC, FB, RC, RB, and DF bits.

8.2.2 Using Software To Complete the Bus Cycles

One method of completing a faulted bus cycle is to use a software handler to emulate the cycle. This is the only method for correcting address errors. The handler should emulate the faulted bus cycle in a manner that is transparent to the instruction that caused the fault. For instruction stream faults, the handler may need to run bus cycles for both the B and C stages of the instruction pipe. The RB and RC bits identify the stages that may require a bus cycle; the FB and FC bits indicate that a stage was invalid when an attempt was made to use its contents. Those stages must be repaired. For each faulted stage, the software handler should copy the instruction word from the proper address space as indicated by the S bit of the copy of the status register saved on the stack to the image of the appropriate stage in the stack frame. In addition, the handler must clear the rerun bit associated with the stage that it has corrected. The handler should not change the fault bits FB and FC.

To repair data faults (indicated by DF = 1), the software should first examine the RM bit in the SSW to determine if the fault was generated during a read-modify-write operation. If RM = 0, the handler should then check the R/W bit of the SSW to determine if the fault was caused by a read or a write cycle. For data write faults, the handler must transfer the properly sized data from the data output buffer (DOB) on the stack frame to the location indicated by

the data fault address in the address space defined by the SSW. (Both the DOB and the data fault address are part of the stack frame at SP+$18 and SP+$10, respectively.) Data read faults only generate the long bus fault frame and the handler must transfer properly sized data from the location indicated by the fault address and address space to the image of the data input buffer (DIB) at location SP+$2C of the long format stack frame. Byte, word, and 3-byte operands are right-justified in the 4-byte data buffers. In addition, the software handler must clear the DF bit of the SSW to indicate that the faulted bus cycle has been corrected.

To emulate a read-modify-write cycle, the exception handler must first read the operation word at the program counter address (SP+2 of the stack frame). This word identifies the CAS, CAS2, or TAS instruction that caused the fault. Then the handler must emulate this entire instruction (which may consist of up to four long word transfers) and update the condition code portion of the status register appropriately, because the RTE instruction expects the entire operation to have been completed if the RM bit is set and the DF bit is cleared. This is true even if the fault occurred on the first read cycle.

To emulate the entire instruction, the handler must save the data and address registers for the instruction (with a MOVEM instruction, for example). Next, the handler reads and modifies (if necessary) the memory location. It clears the DF bit in the SSW of the stack frame and modifies the condition codes in the status register copy and the copies of any data or address registers required for the CAS and CAS2 instructions. Last, the handler restores the registers that it saved at the beginning of the emulation. Except for the data input buffer (DIB), the copy of the status register, and the SSW, the handler should not modify a bus fault stack frame. The only bits in the SSW that may be modified are DF, RB, and RC; all other bits, including those defined for internal use, must remain unchanged.

Address error faults must be repaired in software. Address error faults can be distinguished from bus error faults by the value in the vector offset field of the format word.

8.2.3 Completing the Bus Cycles with RTE

Another method of completing a faulted bus cycle is to allow the controller to rerun the bus cycles during execution of the RTE instruction that terminates the exception handler. This method cannot be used to recover from address errors. The RTE instruction is always executed. Unless the handler routine has corrected the error and cleared the fault (and cleared the rerun and DF

bits of the SSW), the RTE instruction can complete the bus cycle(s). If the DF bit is still set at the time of the RTE execution, the faulted data cycle is rerun by the RTE instruction. If the fault bit for a stage of the pipe is set and the corresponding rerun bit was not cleared by the software, the RTE reruns the associated instruction prefetch. The fault occurs again unless the cause of the fault, such as a non-resident page in a virtual memory system, has been corrected. If the rerun bit is set for a stage of the pipe and the fault bit is cleared, the associated prefetch cycle may or may not be run by the RTE instruction (depending on whether the stage is required).

If a fault occurs when the RTE instruction attempts to rerun the bus cycle(s), the controller creates a new stack frame on the supervisor stack after de-allocating the previous frame, and address error or bus error exception processing starts in the normal manner.

The read-modify-write operations of the MC68EC030 can also be completed by the RTE instruction that terminates the handler routine. The rerun operation, executed by the RTE instruction with the DF bit of the SSW set, reruns the entire instruction. If the cause of the error has been corrected, the handler does not need to emulate the instruction but can leave the DF bit set and execute the RTE instruction.

Systems programmers and designers should be aware that the ACU of the MC68EC030 treats any bus cycle with $\overline{RMC}$ asserted as a write operation for protection checking, regardless of the state of R/$\overline{W}$ signal. Otherwise, the potential for partially destroying system pointers with CAS and CAS2 instructions exists since one portion of the write operation could take place and the remainder be aborted by a bus error.

8.3 COPROCESSOR CONSIDERATIONS

Exception handler programmers should consider carefully whether to save and restore the context of a coprocessor at the beginning and end of handler routines for exceptions that can occur during the execution of a coprocessor instruction (i.e., bus errors, interrupts, and coprocessor-related exceptions). The nature of the coprocessor and the exception handler routine determines whether or not saving the state of one or more coprocessors with the cpSAVE and cpRESTORE instructions is required. If the coprocessor allows multiple coprocessor instructions to be executed concurrently, it may require its state to be saved and restored for all coprocessor-generated exceptions, regardless of whether or not the coprocessor is accessed during the handler routine.

The MC68882 floating-point coprocessor is an example of this type of coprocessor. On the other hand, the MC68881 floating-point coprocessor requires FSAVE and FRESTORE instructions within an exception handler routine only if the exception handler itself uses the coprocessor.

8.4 EXCEPTION STACK FRAME FORMATS

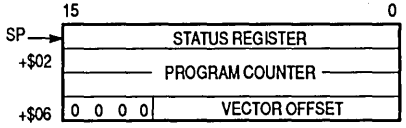
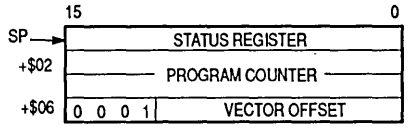
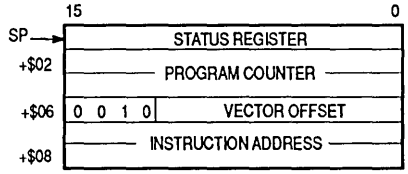
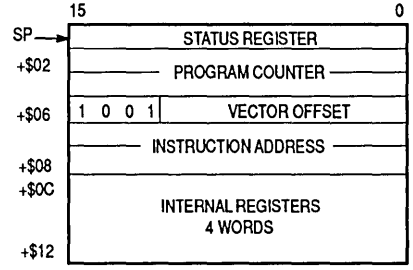
The MC68EC030 provides six different stack frames for exception processing. The set of frames includes the normal four- and six-word stack frames, the four-word throwaway stack frame, the coprocessor mid-instruction stack frame, and the short and long bus fault stack frames.

When the MC68EC030 writes or reads a stack frame, it uses long-word operand transfers wherever possible. Using a long-word-aligned stack pointer with memory that is on a 32-bit port greatly enhances exception processing performance. The controller does not necessarily read or write the stack frame data in sequential order.

The system software should not depend on a particular exception generating a particular stack frame. For compatibility with future devices, the software should be able to handle any type of stack frame for any type of exception.

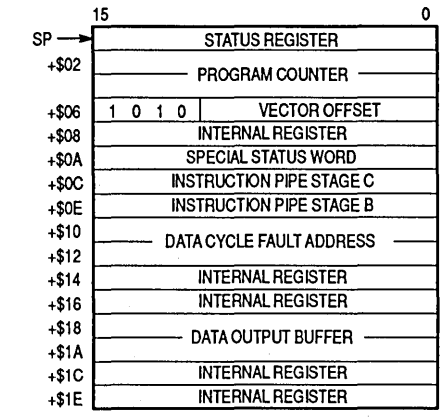
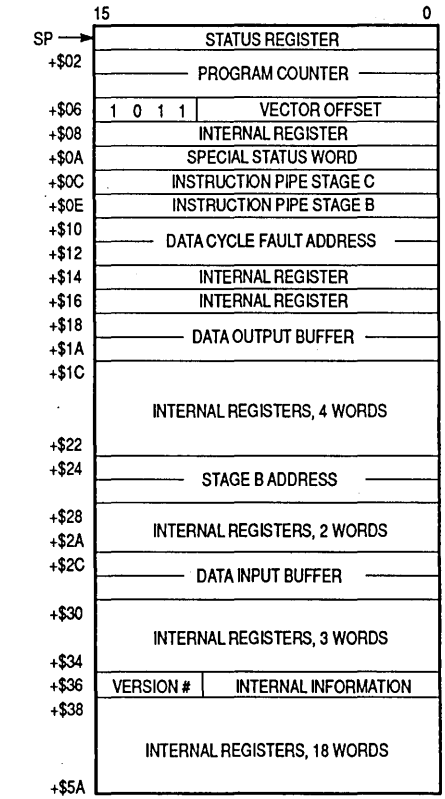
Table 8-6 summarizes the stack frames defined for the M68000 Family.

Table 8-6. Exception Stack Frames

Stack Frames	Exception Types (Stacked PC Points to)
 FOUR-WORD STACK FRAME — FORMAT $0	• Interrupt [Next instruction] • Format Error [RTE or cpRESTORE instruction] • TRAP #N [Next instruction] • Illegal Instruction [Illegal instruction] • A-Line Instruction [A-line instruction] • F-Line Instruction [F-line instruction] • Privilege Violation [First word of instruction causing Privilege Violation] • Coprocessor Pre-Instruction [Op-Word of instruction that returned the Take Pre-Instruction primitive]
 THROWAWAY FOUR-WORD STACK FRAME — FORMAT $1	• Created on Interrupt Stack during interrupt exception processing when transition from master state to interrupt state occurs [Next instruction — same as on master stack]
 SIX-WORD STACK FRAME — FORMAT $2	• CHK • CHK2 • cpTRAPcc • TRAPcc • TRAPV • Trace • Zero Divide • Coprocessor Post-Instruction [Next instruction for all these exceptions] INSTRUCTION ADDRESS is the address of the instruction that caused the exception
 COPROCESSOR AND INSTRUCTION STACK FRAME (10 WORDS) — FORMAT $9	• Coprocessor Mid-Instruction • Main-Detected Protocol Violation • Interrupt Detected during Coprocessor Instruction (supported with 'null come again with interrupts allowed' primitive) [Next word to be fetched from instruction stream for all these exceptions] INSTRUCTION ADDRESS is the address of the instruction that caused the exception

8

Table 8-6. Exception Stack Frames (Continued)

Stack Frames	Exception Types (Stacked PC Points to)
 SHORT BUS CYCLE FAULT STACK FRAME (16 WORDS) — FORMAT $A	• Address Error or Bus Error — Execution Unit at Instruction Boundary [Next instruction]
 LONG BUS CYCLE FAULT STACK FRAME (46 WORDS) — FORMAT $B	• Address Error or Bus Error — Instruction Execution in Progress [Address of instruction in execution when fault occurred — may not be the instruction that generated the faulted bus cycle]

SECTION 9
ACCESS CONTROL UNIT

The MC68EC030 includes an access control unit (ACU) that supports cacheability distinctions for address blocks.

The ACU completely overlaps address control checking time with other processing activity. ACU accesses operate in parallel with the on-chip instruction and data caches.

Figure 9-1 is a block diagram of the MC68EC030 showing the relationship of the ACU to the execution unit and the bus controller. For either an instruction or operand access, the MC68EC030 simultaneously searches for an address match in the instruction cache, the data cache, and the ACU.

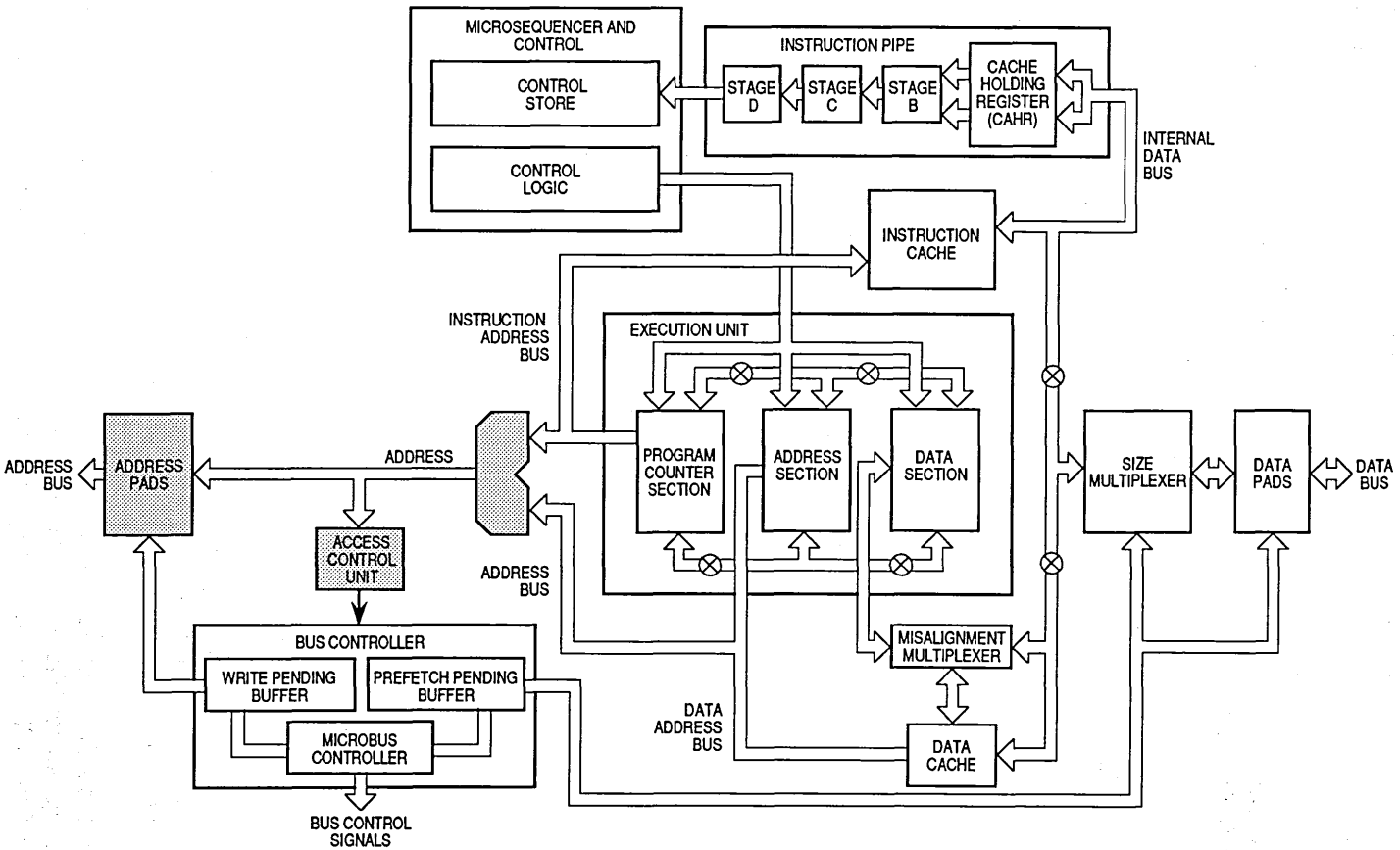


Figure 9-1. ACU Block Diagram

The programming model of the ACU (see Figure 9-2) consists of two access control registers and an ACU status register. These registers can only be accessed by supervisor programs. Each access control register can define a block of addresses that are marked as cacheable or noncacheable. The ACU status register contains status information from a test performed as a part of a PTEST instruction.

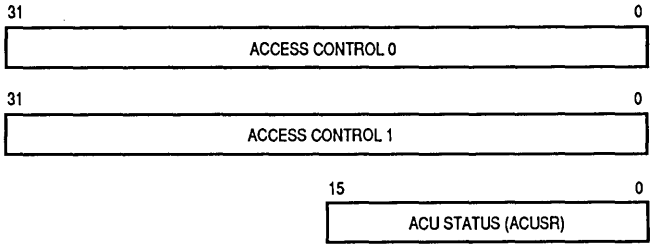


Figure 9-2. ACU Programming Model

9.1 EFFECT OF $\overline{\text{RESET}}$ ON ACU

When the MC68EC030 is reset by the assertion of the RESET signal, the E bits of the access control registers are cleared, disabling address access control.

9

9.2 ACCESS CONTROL

Two independent access control registers (AC0 and AC1) in the ACU optionally define two blocks of the address space that control cacheability for access to those address spaces. The ACU does not explicitly check write protection for the addresses in these blocks, but a block can be specified as cache inhibited for read cycles only. The blocks of addresses defined by the ACx registers include at least 16 Mbytes of address space; the two blocks can overlap or can be separate.

The following description of the address comparison assumes that both AC0 and AC1 are enabled; however, each ACx register can be independently disabled. A disabled ACx register is completely ignored.

When the ACU receives an address to be checked, the function code and the eight high-order bits of the address are compared to the block of addresses defined by AC0 and AC1. The address space block for each ACx register is

defined by the base function code, the function code mask, the base address, and the address mask. When a bit in a mask field is set, the corresponding bit of the base function code or base address is ignored in the function code and address comparison. Setting successively higher order bits in the address mask increases the size of the access-controlled block.

The address for the current bus cycle and an ACx register address match when the function code bits and address bits (not including masked bits) are equal. Each ACx register can specify either read accesses or write accesses as controlled. The internal R/$\overline{W}$ signal must match the R/W bit in the ACx register for the match to occur. The selection of the type of access (read or write) can also be masked. The read/write mask bit (RWM) must be set for access control of addresses used by instructions that execute read-modify-write operations. Otherwise, neither the read nor write portions of read-modify-write operations are matched with the ACx registers, regardless of the function code and address bits for the individual cycles within a read-modify-write operation.

By appropriately configuring an ACx register, flexible address mapping can be specified. For example, to control access to user program space with an ACx register, the RWM bit is set to 1, the FC BASE is set to $2, and the FC MASK is set to $0. To control access to supervisor data read accesses of addresses $00000000–$0FFFFFFF, the BASE ADDRESS field is set to $0X, the ADDRESS MASK is set to $0F, the R/W bit is set to 1, the RWM bit is set to 0, the FC BASE is set to $5, and the FC MASK field is set to $0. Since only read cycles are specified by the ACx register for this example, write accesses for this address range are cacheable unless cache inhibit in ($\overline{CIIN}$) or cache inhibit out ($\overline{CIOUT}$) is asserted.

Each ACx register can specify that the contents of addresses in its block should not be stored in either an internal or external cache. $\overline{CIOUT}$ is asserted and $\overline{CBREQ}$ is negated when an address matches the address specified by an ACx register and the cache inhibit (CI) bit in that ACx register is set. If the address matches in both registers, the CI bits are ORed together to generate the $\overline{CIOUT}$ signal. $\overline{CIOUT}$ is used by the on-chip instruction and data caches to inhibit caching of data associated with this address. The signal is available to external caches for the same purpose.

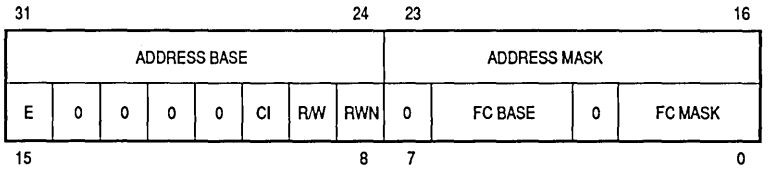
9.3 REGISTERS

The registers of the ACU described in the following paragraphs are part of the supervisor programming model for the MC68EC030. The PMOVE instruction is used to load or read the ACU registers (ACUSR, AC0, and AC1).

The three registers that control and provide status information for memory access control in the MC68EC030 are the two ACx registers and the ACU status register (ACUSR). These registers can be accessed only by programs that execute at the supervisor level.

9.3.1 Access Control Registers

The ACx registers are 32-bit registers that define blocks of address space controlled for cacheability status. The minimum size block that can be defined by either ACx register is 16 Mbytes of address space. The two ACx registers can specify blocks that overlap. An ACx register is shown in Figure 9-3.



ADDRESS BASE - VALUE OF A31-A24 THAT DEFINES BLOCK
ADDRESS MASK - BITS A31-A24 TO BE IGNORED

E - ENABLE
CI - CACHE INHIBIT
R/W - READ/WRITE
RWM - READ WRITE MASK
FC BASE - FUNCTION CODE VALUE FOR BLOCK
FC MASK - FUNCTION CODE BITS TO BE IGNORED

Figure 9-3. Access Control Register Format

The fields of an ACx register are as follows:

Enable (E)
This bit enables access control of the block defined by this register:
0 = Access control disabled
1 = Access control enabled
A reset operation clears this bit.

Cache Inhibit (CI)

This bit inhibits caching for the matching block:

0 = Caching allowed

1 = Caching inhibited

When this bit is set, the contents of a matching address are not stored in the internal instruction or data cache. Additionally, $\overline{\text{CIOUT}}$ is asserted and $\overline{\text{CBREQ}}$ is negated when this bit is set and a matching address is accessed, signaling external caches to inhibit caching for those accesses.

Read/Write (R/W)

This bit defines the type of access that is access controlled (for a matching address):

0 = Write accesses controlled

1 = Read accesses controlled

Read/Write Mask (RWM)

This bit masks the R/W field:

0 = R/W field used

1 = R/W field ignored

When RWM is set to one, both read and write accesses of a matching address are access controlled. For access control of read-modify-write cycles with matching addresses, RWM must be set to one. If the RWM bit equals zero, neither the read nor the write of any read-modify-write cycle is access controlled.

Function Code Base (FC BASE)

This 3-bit field defines the base function code for accesses to be access controlled with this register. Addresses with function codes that match the FC BASE field (and are otherwise eligible) are access controlled.

Function Code Mask (FC MASK)

This 3-bit field contains a mask for the FC BASE field. Setting a bit in this field causes the corresponding bit of the FC BASE field to be ignored.

Address Base (ADDRESS BASE)

This 8-bit field is compared with address bits A31–A24. Addresses that match in this comparison (and are otherwise eligible) are access controlled.

Address Mask (ADDRESS MASK)

This 8-bit field contains a mask for the ADDRESS BASE field. Setting a bit in this field causes the corresponding bit of the ADDRESS BASE field to be ignored. Blocks of memory larger than 16 Mbytes can be access controlled by setting some of the address mask bits to ones. The low-order bits of this field are normally set to define contiguous blocks larger than 16 Mbytes, although this is not required.

9.3.2 ACU Status Register

The ACUSR (see Figure 9-4) is a 16-bit register that contains the status information returned by execution of the PTEST instruction. The PTEST instruction searches the ACx registers to determine ACx match of a specified address. The AC bit in the ACUSR is set if a match occurred in either (or both) of the ACx registers.

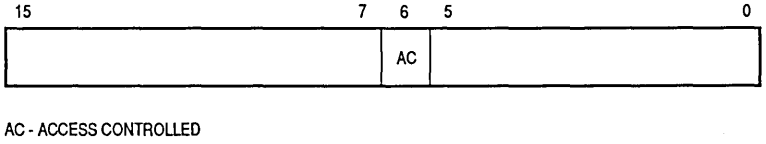


Figure 9-4. ACU Status Register (ACUSR) Format

9.4 ACU INSTRUCTIONS

The MC68EC030 instruction set includes two privileged instructions that perform ACU operations. A brief description of each instruction follows.

The PMOVE instruction transfers data between a CPU register or memory location and either of the two ACx registers. The operating system uses the PMOVE instruction to control and monitor ACU operation by manipulating and reading these registers.

The PTEST instruction searches the ACx registers for a specified function code and address, and sets the AC bit in the ACUSR to indicate a match occurred during the search. Note that the PTESTR and PTESTW instructions have different results when either ACx register matches the address and the R/W bit of that register is not masked.

The ACU instructions use the same opcodes and coprocessor identification (CpID) as the corresponding instructions of the MC68851, MC68030, and MC68040. F-line instructions with CpID = 0 (including MC68851 instructions), which the MC68EC030 does not support, automatically cause F-line unimplemented instruction exceptions when execution is attempted in the superviser mode. PFLUSH, PMOVEFD, and PLOAD with CpID = 0 may or may not cause F-line unimplemented instruction exceptions in the supervisor mode but can produce undefined results and should not be used. If execution of a unimplemented F-line instruction with CpID = 0 is attempted in the user mode, the MC68EC030 takes a privilege violation exception. F-line instructions with a CpID other than zero are executed as coprocessor instructions by the MC68EC030.

The MC68030 assembler can be used for the MC68EC030 as long as the following procedures are observed:

1. To access AC0 in the MC68EC030, use TT0 in the MC68030 assembler.
2. To access AC1 in the MC68EC030, use TT1 in the MC68030 assembler.
3. To access ACUSR in the MC68EC030, use MMUSR in the MC68030 assembler.
4. PTEST uses the assembler syntax of PTESTR <function code>,<ea>, #0 and PTESTW <function code>,<ea>, #0.
5. All other accesses to MMU functions are invalid, can produce undefined results, and should not be used.

9

Details of the two new instructions for the MC68EC030 are listed in **APPENDIX A MC68EC030 NEW INSTRUCTIONS**.

SECTION 10
COPROCESSOR INTERFACE DESCRIPTION

The M68000 Family of general-purpose microprocessors provides a level of performance that satisfies a wide range of computer applications. Special-purpose hardware, however, can often provide a higher level of performance for a specific application. The coprocessor concept allows the capabilities and performance of a general-purpose controller to be enhanced for a particular application without encumbering the main controller architecture. A coprocessor can efficiently meet specific capability requirements that must typically be implemented in software by a general-purpose controller. With a general-purpose main controller and the appropriate coprocessor(s), the processing capabilities of an embedded control system can be tailored to a specific application.

The MC68EC030 supports the M68000 coprocessor interface described in this section. The section is intended for designers who are implementing coprocessors to interface with the MC68EC030.

The designer of a system that uses one or more Motorola coprocessors (the MC68881 or MC68882 floating-point coprocessor, for example) does not require a detailed knowledge of the M68000 coprocessor interface. Motorola coprocessors conform to the interface described in this section. Typically, they implement a subset of the interface, and that subset is described in the coprocessor user's manual. These coprocessors execute Motorola defined instructions that are described in the user's manual for each coprocessor.

10.1 INTRODUCTION

The distinction between standard peripheral hardware and a M68000 coprocessor is important from a perspective of the programming model. The programming model of the main controller consists of the instruction set, register set, and memory map available to the programmer. An M68000 coprocessor is a device or set of devices that communicates with the main controller through the protocol defined as the M68000 coprocessor interface. The programming model for a coprocessor is different than that for a peripheral device. A coprocessor adds additional instructions and generally additional registers and data types to the programming model that are not directly

supported by the main controller architecture. The additional instructions are dedicated coprocessor instructions that utilize the coprocessor capabilities. The necessary interactions between the main controller and the coprocessor that provide a given service are transparent to the programmer. That is, the programmer does not need to know the specific communication protocol between the main controller and the coprocessor because this protocol is implemented in hardware. Thus, the coprocessor can provide capabilities to the user without appearing separate from the main controller.

In contrast, standard peripheral hardware is generally accessed through interface registers mapped into the memory space of the main controller. To use the services provided by the peripheral, the programmer accesses the peripheral registers with standard controller instructions. While a peripheral could conceivably provide capabilities equivalent to a coprocessor for many applications, the programmer must implement the communication protocol between the main controller and the peripheral necessary to use the peripheral hardware.

The communication protocol defined for the M68000 coprocessor interface is described in **10.2 COPROCESSOR INSTRUCTION TYPES**. The algorithms that implement the M68000 coprocessor interface are provided in the microcode of the MC68EC030 and are completely transparent to the MC68EC030 programmer's model. For example, floating-point operations are not implemented in the MC68EC030 hardware. In a system utilizing both the MC68EC030 and the MC68881 or MC68882 floating-point coprocessor, a programmer can use any of the instructions defined for the coprocessor without knowing that the actual computation is performed by the MC68881 or MC68882 hardware.

10.1.1 Interface Features

The M68000 coprocessor interface design incorporates a number of flexible capabilities. The physical coprocessor interface uses the main controller external bus, which simplifies the interface since no special-purpose signals are involved. With the MC68EC030, a coprocessor can use either the asynchronous or synchronous bus transfer protocol. Since standard bus cycles transfer information between the main controller and the coprocessor, the coprocessor can be implemented in whatever technology is available to the coprocessor designer. A coprocessor can be implemented as a VLSI device, as a separate system board, or even as a separate computer system.

Since the main controller and a M68000 coprocessor can communicate using the asynchronous bus, they can operate at different clock frequencies. The MC68882 can not have a clock frequency difference of more than one fre-

quency step lower than the MC68EC030 (e.g., a 40-MHz MC68EC030 and a 33-MHz MC68882 is valid, but a 25-MHz or slower MC68882 is not valid). The MC68881 does not have this restriction. The system designer can choose the speeds of a main controller and coprocessor that provide the optimum performance for a given system. If the coprocessor uses the synchronous bus interface all coprocessor signals and data must be synchronized with the main controller clock. Both the MC68881 and MC68882 floating-point coprocessors use the asynchronous bus handshake protocol.

The M68000 coprocessor interface also facilitates the design of coprocessors. The coprocessor designer must only conform to the coprocessor interface and does not need an extensive knowledge of the architecture of the main controller. Also, the main controller can operate with a coprocessor without having explicit provisions made in the main controller for the capabilities of that coprocessor. This provides a great deal of freedom in the implementation of a given coprocessor.

10.1.2 Concurrent Operation Support

The programmer's model for the M68000 Family of microprocessors is based on sequential, nonconcurrent instruction execution. This implies that the instructions in a given sequence must appear to be executed in the order in which they occur. To maintain a uniform programmer's model, any coprocessor extensions should also maintain the model of sequential, nonconcurrent instruction execution at the user level. Consequently, the programmer can assume that the images of registers and memory affected by a given instruction have been updated when the next instruction in the sequence accessing these registers or memory locations is executed.

The M68000 coprocessor interface provides full support of all operations necessary for nonconcurrent operation of the main controller and its associated coprocessors. Although the M68000 coprocessor interface allows concurrency in coprocessor execution, the coprocessor designer is responsible for implementing this concurrency while maintaining a programming model based on sequential nonconcurrent instruction execution.

For example, if the coprocessor determines that instruction "B" does not use or alter resources to be altered or used by instruction "A", instruction "B" can be executed concurrently (if the execution hardware is also available). Thus, the required instruction interdependencies and sequences of the program are always respected. The MC68882 coprocessor offers concurrent instruction execution while the MC68881 coprocessor does not. However, the MC68EC030 can execute instructions concurrently with coprocessor instruction execution in the MC68881.

10.1.3 Coprocessor Instruction Format

The instruction set for a given coprocessor is defined by the design of that coprocessor. When a coprocessor instruction is encountered in the main controller instruction stream, the MC68EC030 hardware initiates communication with the coprocessor and coordinates any interaction necessary to execute the instruction with the coprocessor. A programmer needs to know only the instruction set and register set defined by the coprocessor in order to use the functions provided by the coprocessor hardware.

The instruction set of an M68000 coprocessor uses a subset of the F-line operation words in the M68000 instruction set. The operation word is the first word of any M68000 Family instruction. The F-line operation word contains ones in bits 15–12 ([15:12] = 1111; refer to Figure 10-1); the remaining bits are coprocessor and instruction dependent. The F-line operation word may be followed by as many extension words as are required to provide additional information necessary for the execution of the coprocessor instruction.

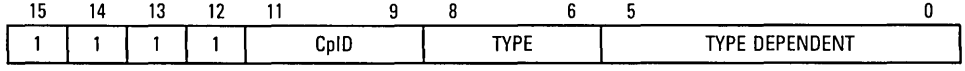


Figure 10-1. F-Line Coprocessor Instruction Operation Word

As shown in Figure 10-1, bits 9–11 of the F-line operation word encode the coprocessor identification code (CpID). The MC68EC030 uses the coprocessor identification field to indicate the coprocessor to which the instruction applies. F-line operation words, in which the CpID is zero, are not coprocessor instructions for the MC68EC030. If the CpID (bits 9–11) and the type field (bits 6–8) contain zeros, the instruction accesses the on-chip access control unit of the MC68EC030. If the instruction is not implemented by the ACU, an unimplemented instruction exception may or may not be taken. See **APPENDIX A MC68EC030 NEW INSTRUCTIONS** for ACU instructions. Instructions with a CpID of zero and a nonzero type field are unimplemented instructions that cause the MC68EC030 to begin exception processing. The MC68EC030 never generates coprocessor interface bus cycles with the CpID equal to zero (except via the MOVES instruction).

CpID codes of 001–101 are reserved for current and future Motorola coprocessors and CpID codes of 110–111 are reserved for user-defined coprocessors. The Motorola CpID code that is currently defined is 001 for the MC68881 or MC68882 floating-point coprocessor. By default, Motorola assemblers will use CpID code 001 when generating the instruction operation codes for the MC68881 or MC68882 coprocessor instructions.

The encoding of bits 0–8 of the coprocessor instruction operation word is dependent on the particular instruction being implemented (see **10.2 COPROCESSOR INSTRUCTION TYPES**).

10.1.4 Coprocessor System Interface

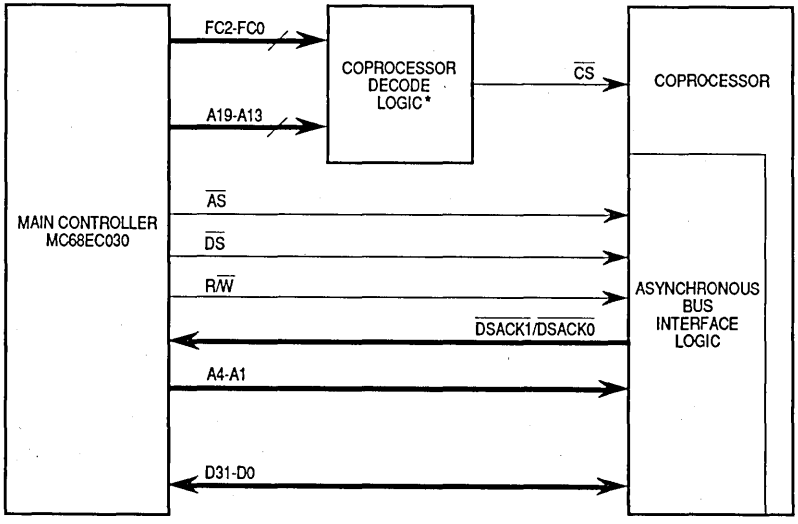
The communication protocol between the main controller and coprocessor necessary to execute a coprocessor instruction uses a group of interface registers, called coprocessor interface registers, resident within the coprocessor. By accessing one of these interface registers, the MC68EC030 hardware initiates coprocessor instructions. The coprocessor uses a set of response primitive codes and format codes defined for the M68000 coprocessor interface to communicate status and service requests to the main controller through these registers. The coprocessor interface registers (CIRs) are also used to pass operands between the main controller and the coprocessor. The CIR set, response primitives, and format codes are discussed in **10.3 COPROCESSOR INTERFACE REGISTER SET** and **10.4 COPROCESSOR RESPONSE PRIMITIVES**.

10.1.4.1 COPROCESSOR CLASSIFICATION. M68000 coprocessors can be classified into two categories depending on their bus interface capabilities. The first category, non-DMA coprocessors, consists of coprocessors that always operate as bus slaves. The second category, DMA coprocessors, consists of coprocessors that operate as bus slaves while communicating with the main controller across the coprocessor interface, but also have the ability to operate as bus masters, directly controlling the system bus.

If the operation of a coprocessor does not require a large portion of the available bus bandwidth or has special requirements not directly satisfied by the main controller, that coprocessor can be efficiently implemented as a non-DMA coprocessor. Since non-DMA coprocessors always operate as bus slaves, all external bus-related functions that the coprocessor requires are performed by the main controller. The main controller transfers operands from the coprocessor by reading the operand from the appropriate CIR and then writing the operand to a specified effective address with the appropriate address space specified on the function code lines. Likewise, the main controller transfers operands to the coprocessor by reading the operand from a specified effective address (and address space) and then writing that operand to the appropriate CIR using the coprocessor interface. The bus interface circuitry of a coprocessor operating as a bus slave is not as complex as that of a device operating as a bus master.

To improve the efficiency of operand transfers between memory and the coprocessor, a coprocessor that requires a relatively high amount of bus bandwidth or has special bus requirements can be implemented as a DMA coprocessor. DMA coprocessors can operate as bus masters. The coprocessor provides all control, address, and data signals necessary to request and obtain the bus and then performs DMA transfers using the bus. DMA coprocessors, however, must still act as bus slaves when they require information or services of the main controller using the M68000 coprocessor interface protocol.

10.1.4.2 CONTROLLER-COPROCESSOR INTERFACE. Figure 10-2 is a block diagram of the signals involved in an asynchronous non-DMA M68000 coprocessor interface. The synchronous interface is similar. Since the CpID on signals A13–A15 of the address bus is used with other address signals to select the coprocessor, the system designer can use several coprocessors of the same type and assign a unique CpID to each one.



FC2-FC0 = 111 ▸ CPU SPACE CYCLE
A19-A16 = 0010 ▸ COPROCESSOR ACCESS IN CPU SPACE
A15-A13 = xxx ▸ COPROCESSOR IDENTIFICATION
A4-A1 = rrrr ▸ COPROCESSOR INTERFACE REGISTER SELECTOR

*Chip select logic may be integrated into the coprocessor.
Address lines not specified above are "0" during coprocessor access.

Figure 10-2. Asynchronous Non-DMA M68000 Coprocessor Interface Signal Usage

The MC68EC030 accesses the registers in the CIR set using standard asynchronous or synchronous bus cycles. Thus, the bus interface implemented by a coprocessor for its interface register set must satisfy the MC68EC030 address, data, and control signal timing. The MC68EC030 never requests a burst operation during a coprocessor (CPU space) bus cycle, nor does it internally cache data read or written during coprocessor (CPU space) bus cycles. The MC68EC030 bus operation is described in detail in **SECTION 7 BUS OPERATION.**

During coprocessor instruction execution, the MC68EC030 executes CPU space bus cycles to access the CIR set. The MC68EC030 drives the three function code outputs high (FC2:FC0 = 111) identifying a CPU space bus cycle. The CIR set is mapped into CPU space in the same manner that a peripheral interface register set is generally mapped into data space. The information encoded on the function code lines and address bus of the MC68EC030 during a coprocessor access is used to generate the chip select signal for the coprocessor being accessed. Other address lines select a register within the interface set. The information encoded on the function code and address lines of the MC68EC030 during a coprocessor access is illustrated in Figure 10-3.

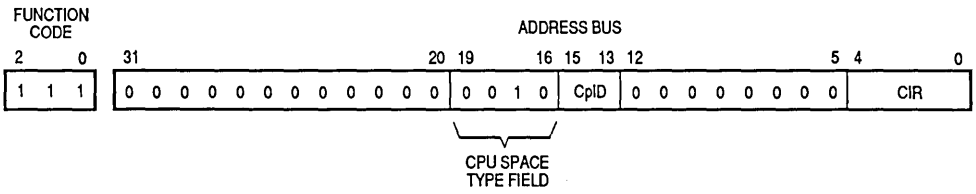


Figure 10-3. MC68EC030 CPU Space Address Encodings

Address signals A16–A19 specify the CPU space cycle type for a CPU space bus cycle. The types of CPU space cycles currently defined for the MC68EC030 are interrupt acknowledge, breakpoint acknowledge, and coprocessor access cycles. CPU space type $2 (A19:A16 = 0010) specifies a coprocessor access cycle.

Signals A13–A15 of the MC68EC030 address bus specify the coprocessor identification code CpID for the coprocessor being accessed. This code is transferred from bits 9–11 of the coprocessor instruction operation word (refer to Figure 10-1) to the address bus during each coprocessor access.

Thus, decoding the MC68EC030 function code signals and bits A13–A19 of the address bus provides a unique chip select signal for a given coprocessor. The function code signals and A16–A19 indicate a coprocessor access; A13–A15 indicate which of the possible seven coprocessors (001–111) is being accessed. Bits A20–A31 and A5–A12 of the MC68EC030 address bus are always zero during a coprocessor access.

10.1.4.3 COPROCESSOR INTERFACE REGISTER SELECTION. Figure 10-4 shows that the value on the MC68EC030 address bus during a coprocessor access addresses a unique region of the main controller's CPU address space. Signals A0–A4 of the MC68EC030 address bus select the CIR being accessed. The register map for the M68000 coprocessor interface is shown in Figure 10-5. The individual registers are described in detail in **10.3 COPROCESSOR INTERFACE REGISTER SET.**

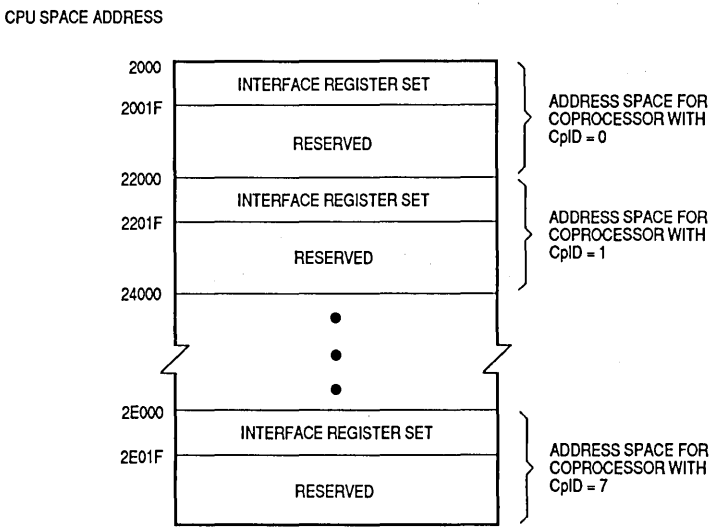


Figure 10-4. Coprocessor Address Map in MC68EC030 CPU Space

	31 16	15 0
00	RESPONSE*	CONTROL*
04	SAVE*	RESTORE*
08	OPERATION WORD	COMMAND*
0C	(RESERVED)	CONDITION*
10	OPERAND*	
14	REGISTER SELECT	(RESERVED)
18	INSTRUCTION ADDRESS	
1C	OPERAND ADDRESS	

Figure 10-5. Coprocessor Interface Register Set Map

10.2 COPROCESSOR INSTRUCTION TYPES

The M68000 coprocessor interface supports four categories of coprocessor instructions: general, conditional, context save, and context restore. The category name indicates the type of operations provided by the coprocessor instructions in the category. The instruction category also determines the CIR accessed by the MC68EC030 to initiate instruction and communication protocols between the main controller and the coprocessor necessary for instruction execution.

During the execution of instructions in the general or conditional categories, the coprocessor uses the set of coprocessor response primitive codes defined for the MC68000 coprocessor interface to request services from and indicate status to the main controller. During the execution of the instructions in the context save and context restore categories, the coprocessor uses the set of coprocessor format codes defined for the M68000 coprocessor interface to indicate its status to the main controller.

10.2.1 Coprocessor General Instructions

The general coprocessor instruction category contains data processing instructions and other general-purpose instructions for a given coprocessor.

10.2.1.1 FORMAT. Figure 10-6 shows the format of a general type instruction.

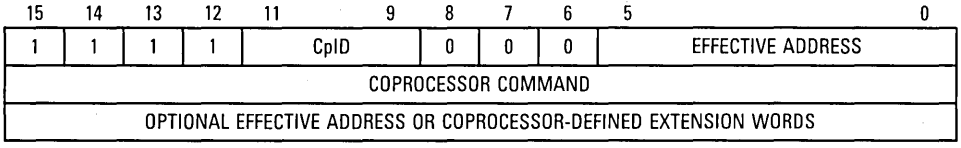


Figure 10-6. Coprocessor General Instruction Format (cpGEN)

The mnemonic cpGEN is a generic mnemonic used in this discussion for all general instructions. The mnemonic of a specific general instruction usually suggests the type of operation it performs and the coprocessor to which it applies. The actual mnemonic and syntax used to represent a coprocessor instruction is determined by the syntax of the assembler or compiler that generates the object code.

A coprocessor general type instruction consists of at least two words. The first word of the instruction is an F-line operation code (bits [15:12] = 1111). The CpID field of the F-line operation code is used during the coprocessor access to indicate which of the coprocessors in the system executes the instruction. During accesses to the coprocessor interface registers (refer to **10.1.4.2 CONTROLLER-COPROCESSOR INTERFACE**), the controller places the CpID on address lines A13–A15.

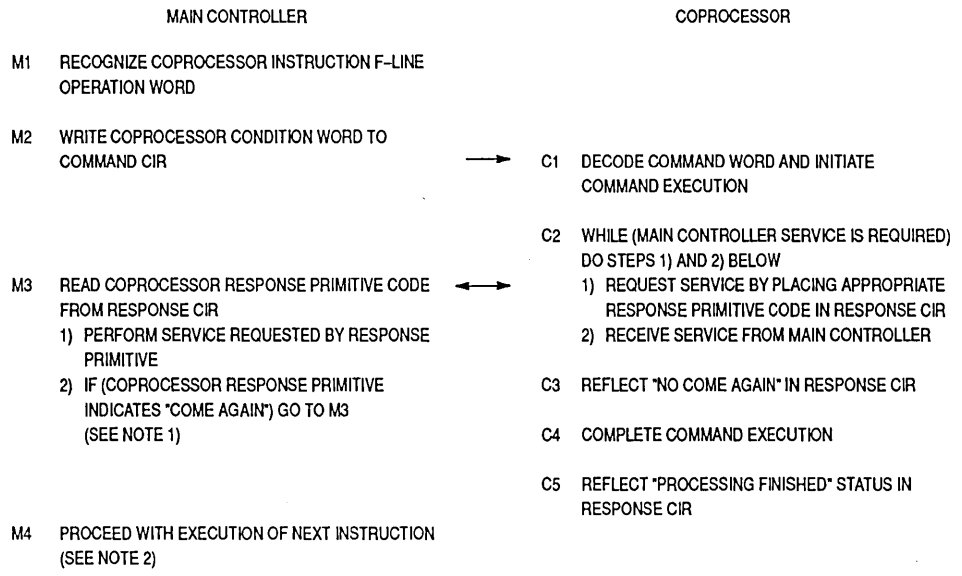
Bits [8:6] = 000 indicate that the instruction is in the general instruction category. Bits 0–5 of the F-line operation code sometimes encodes a standard M68000 effective address specifier (refer to **2.5 EFFECTIVE ADDRESS ENCODING SUMMARY**). During the execution of a cpGEN instruction, the coprocessor can use a coprocessor response primitive to request that the MC68EC030 perform an effective address calculation necessary for that instruction. Using the effective address specifier field of the F-line operation code, the controller then determines the effective addressing mode. If a coprocessor never requests effective address calculation, bits 0–5 can have any value (don't cares).

The second word of the general-type instruction is the coprocessor command word. The main controller writes this command word to the command CIR to initiate execution of the instruction by the coprocessor.

An instruction in the coprocessor general instruction category optionally includes a number of extension words following the coprocessor command word. These words can provide additional information required for the co-

processor instruction. For example, if the coprocessor requests that the MC68EC030 calculate an effective address during coprocessor instruction execution, information required for the calculation must be included in the instruction format as effective address extension words.

10.2.1.2 PROTOCOL. The execution of a cpGEN instruction follows the protocol shown in Figure 10-7. The main controller initiates communication with the coprocessor by writing the instruction command word to the command CIR. The coprocessor decodes the command word to begin processing the cpGEN instruction. Coprocessor design determines the interpretation of the coprocessor command word; the MC68EC030 does not attempt to decode it.



NOTES:
1. "Come Again" indicates that further service of the main controller is being requested by the coprocessor.
2. The next instruction should be the operation word pointed to by the ScanPC at this point. The operation of the MC68EC030 ScanPC is discussed in 10.4.1 ScanPC.

Figure 10-7. Coprocessor Interface Protocol for General Category Instructions

While the coprocessor is executing an instruction, it requests any required services from and communicates status to the main controller by placing coprocessor response primitive codes in the response CIR. After writing to

the command CIR, the main controller reads the response CIR and responds appropriately. When the coprocessor has completed the execution of an instruction or no longer needs the services of the main controller to execute the instruction, it provides a response to release the controller. The main controller can then execute the next instruction in the instruction stream. However, if a trace exception is pending, the MC68EC030 does not terminate communication with the coprocessor until the coprocessor indicates that it has completed all processing associated with the cpGEN instruction (refer to **10.5.2.5 TRACE EXCEPTIONS**).

The coprocessor interface protocol shown in Figure 10-7 allows the coprocessor to define the operation of each general category instruction. That is, the main controller initiates the instruction execution by writing the instruction command word to the command CIR and by reading the response CIR to determine its next action. The execution of the coprocessor instruction is then defined by the internal operation of the coprocessor and by its use of response primitives to request services from the main controller. This instruction protocol allows a wide range of operations to be implemented in the general instruction category.

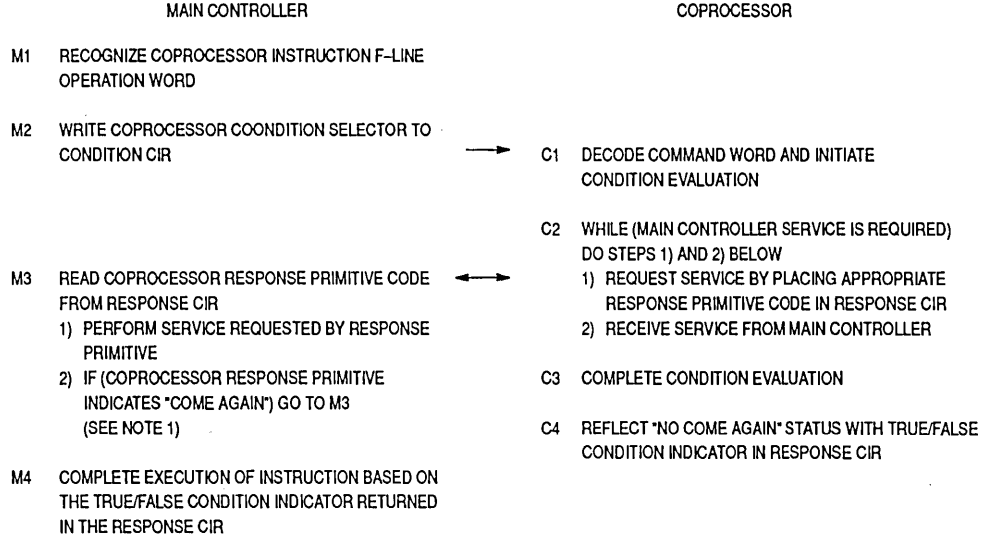
10.2.2 Coprocessor Conditional Instructions

The conditional instruction category provides program control based on the operations of the coprocessor. The coprocessor evaluates a condition and returns a true/false indicator to the main controller. The main controller completes the execution of the instruction based on this true/false condition indicator.

The implementation of instructions in the conditional category promotes efficient use of both the main controller's and the coprocessor's hardware. The condition specified for the instruction is related to the coprocessor operation and is, therefore, evaluated by the coprocessor. The instruction completion following the condition evaluation is, however, directly related to the operation of the main controller. The main controller performs the change of flow, the setting of a byte, or the TRAP operation, since its architecture explicitly implements these operations for its instruction set.

Figure 10-8 shows the protocol for a conditional category coprocessor instruction. The main controller initiates execution of an instruction in this category by writing a condition selector to the condition CIR. The coprocessor decodes the condition selector to determine the condition to evaluate. The coprocessor can use response primitives to request that the main controller provide services required for the condition evaluation. After evaluating the

condition, the coprocessor returns a true/false indicator to the main controller by placing a null primitive (refer to **10.4.4 Null Primitive**) in the response CIR. The main controller completes the coprocessor instruction execution when it receives the condition indicator from the coprocessor.



NOTE: 1. All coprocessor response primitives, except the Null primitive, that allow the "Come Again" primitive attribute must indicate "Come Again" when used during the execution of a conditional category instruction. If a "Come Again" attribute is not indicated in one of these primitives, the main controller will initiate protocol violation exception processing (see 10.6.2.1 PROTOCOL VIOLATIONS).

Figure 10-8. Coprocessor Interface Protocol for Conditional Category Instructions

10.2.2.1 BRANCH ON COPROCESSOR CONDITION INSTRUCTION. The conditional instruction category includes two formats of the M68000 Family branch instruction. These instructions branch on conditions related to the coprocessor operation. They execute similarly to the conditional branch instructions provided in the M68000 Family instruction set.

10.2.2.1.1 Format. Figure 10-9 shows the format of the branch on coprocessor condition instruction that provides a word-length displacement. Figure 10-10 shows the format of the instruction that includes a long-word displacement.

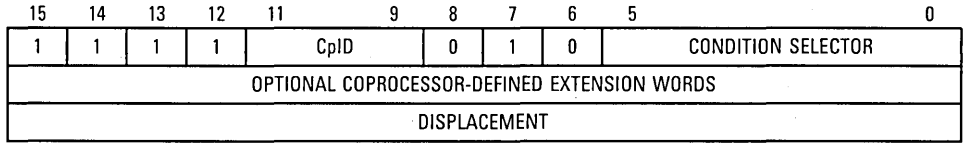


Figure 10-9. Branch on Coprocessor Condition Instruction (cpBcc.W)

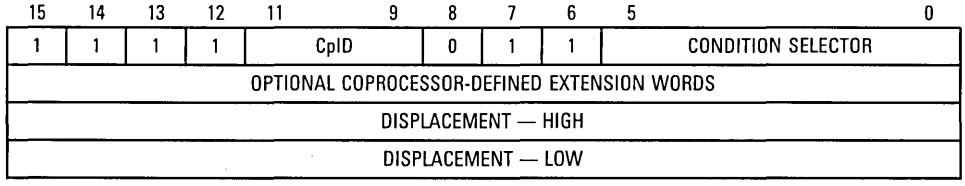


Figure 10-10. Branch On Coprocessor Condition Instruction (cpBcc.L)

The first word of the branch on coprocessor condition instruction is the F-line operation word. Bits [15:12] = 1111 and bits [11:9] contain the identification code of the coprocessor that is to evaluate the condition. The value in bits [8:6] identifies either the word or the long-word displacement format of the branch instruction, which is specified by the cpBcc.W or cpBcc.L mnemonic, respectively.

Bits [0–5] of the F-line operation word contain the coprocessor condition selector field. The MC68EC030 writes the entire operation word to the condition CIR to initiate execution of the branch instruction by the coprocessor. The coprocessor uses bits [0–5] to determine which condition to evaluate.

If the coprocessor requires additional information to evaluate the condition, the branch instruction format can include this information in extension words. Following the F-line operation word, the number of extension words is determined by the coprocessor design. The final word(s) of the cpBcc instruction format contains the displacement used by the main controller to calculate the destination address when the branch is taken.

10.2.2.1.2 Protocol. Figure 10-8 shows the protocol for the cpBcc.L and cpBcc.W instructions. The main controller initiates the instruction by writing the F-line operation word to the condition CIR to transfer the condition selector to the coprocessor. The main controller then reads the response CIR to determine its next action. The coprocessor can return a response primitive to

request services necessary to evaluate the condition. If the coprocessor returns the false condition indicator, the main controller executes the next instruction in the instruction stream. If the coprocessor returns the true condition indicator, the controller adds the displacement to the MC68EC030 scanPC (refer to **10.4.1 ScanPC**) to determine the address of the next instruction for the main controller to execute. The scanPC must be pointing to the location of the first word of the displacement in the instruction stream when the address is calculated. The displacement is a twos-complement integer that can be either a 16-bit word or a 32-bit long word. The controller sign-extends the 16-bit displacement to a long-word value for the destination address calculation.

10.2.2.2 SET ON COPROCESSOR CONDITION INSTRUCTION. The set on coprocessor condition instructions set or reset a flag (a data alterable byte) according to a condition evaluated by the coprocessor. The operation of this instruction is similar to the operation of the Scc instruction in the M68000 Family instruction set. Although the Scc instruction and the cpScc instruction do not explicitly cause a change of program flow, they are often used to set flags that control program flow.

10.2.2.2.1 Format. Figure 10-11 shows the format of the set on coprocessor condition instruction, denoted by the cpScc mnemonic.

15	14	13	12	11 9	8	7	6	5 0
1	1	1	1	CpID	0	0	1	EFFECTIVE ADDRESS
								CONDITION SELECTOR
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS								
OPTIONAL EFFECTIVE ADDRESS EXTENSION WORDS (0–5 WORDS)								

Figure 10-11. Set On Coprocessor Condition (cpScc)

The first word of the cpScc instruction is the F-line operation word. This word contains the CpID field in bits [9–11] and 001 in bits [8:6] to identify the cpScc instruction. The lower six bits of the F-line operation word are used to encode an M68000 Family effective addressing mode (refer to **2.5 EFFECTIVE ADDRESS ENCODING SUMMARY**).

The second word of the cpScc instruction format contains the coprocessor condition selector in bits [0–5]. Bits [6–15] of this word are reserved by Motorola and should be zero to ensure compatibility with future M68000 products. This word is written to the condition CIR to initiate the cpScc instruction.

If the coprocessor requires additional information to evaluate the condition, the instruction can include extension words to provide this information. The number of these extension words, which follow the word containing the coprocessor condition selector field, is determined by the coprocessor design.

The final portion of the cpScc instruction format contains zero to five effective address extension words. These words contain any additional information required to calculate the effective address specified by bits [0–5] of the F-line operation word.

10.2.2.2.2 Protocol. Figure 10-8 shows the protocol for the cpScc instruction. The MC68EC030 transfers the condition selector to the coprocessor by writing the word following the F-line operation word to the condition CIR. The main controller then reads the response CIR to determine its next action. The coprocessor can return a response primitive to request services necessary to evaluate the condition. The operation of the cpScc instruction depends on the condition evaluation indicator returned to the main controller by the coprocessor. When the coprocessor returns the false condition indicator, the main controller evaluates the effective address specified by bits [0–5] of the F-line operation word and sets the byte at that effective address to FALSE (all bits cleared). When the coprocessor returns the true condition indicator, the main controller sets the byte at the effective address to TRUE (all bits set to one).

10.2.2.3 TEST COPROCESSOR CONDITION, DECREMENT AND BRANCH INSTRUCTION. The operation of the test coprocessor condition, decrement and branch instruction is similar to that of the DBcc instruction provided in the M68000 Family instruction set. This operation uses a coprocessor evaluated condition and a loop counter in the main controller. It is useful for implementing DO-UNTIL constructs used in many high-level languages.

10.2.2.3.1 Format. Figure 10-12 shows the format of the test coprocessor condition, decrement and branch instruction, denoted by the cpDBcc mnemonic.

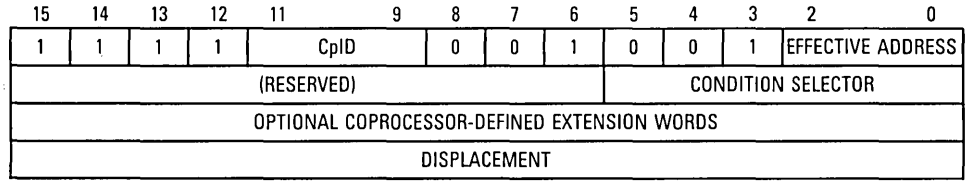


Figure 10-12. Test Coprocessor Condition, Decrement and Branch Instruction Format (cpDBcc)

The first word of the cpDBcc instruction is the F-line operation word. This word contains the CpID field in bits [9–11] and 001001 in bits [8:3] to identify the cpDBcc instruction. Bits [0:2] of this operation word specify the main controller data register used as the loop counter during the execution of the instruction.

The second word of the cpDBcc instruction format contains the coprocessor condition selector in bits [0–5] and should contain zeros in bits [6–15] to maintain compatibility with future M68000 products. This word is written to the condition CIR to initiate execution of the cpDBcc instruction by the coprocessor.

If the coprocessor requires additional information to evaluate the condition, the cpDBcc instruction can include this information in extension words. These extension words follow the word containing the coprocessor condition selector field in the cpDBcc instruction format.

The last word of the instruction contains the displacement for the cpDBcc instruction. This displacement is a twos-complement 16-bit value that is sign-extended to long-word size when it is used in a destination address calculation.

10.2.2.3.2 Protocol. Figure 10-8 shows the protocol for the cpDBcc instructions. The MC68EC030 transfers the condition selector to the coprocessor by writing the word following the operation word to the condition CIR. The main controller then reads the response CIR to determine its next action. The coprocessor can use a response primitive to request any services necessary to evaluate the condition. If the coprocessor returns the true condition indicator, the main controller executes the next instruction in the instruction stream. If the coprocessor returns the false condition indicator, the main controller decrements the low-order word of the register specified by bits [0–2] of the F-line operation word. If this register contains minus one (−1) after being

decremented, the main controller executes the next instruction in the instruction stream. If the register does not contain minus one (−1) after being decremented, the main controller branches to the destination address to continue instruction execution.

The MC68EC030 adds the displacement to the scanPC (refer to **10.4.1 ScanPC**) to determine the address of the next instruction. The scanPC must point to the 16-bit displacement in the instruction stream when the destination address is calculated.

10.2.2.4 TRAP ON COPROCESSOR CONDITION. The trap on coprocessor condition instruction allows the programmer to initiate exception processing based on conditions related to the coprocessor operation.

10.2.2.4.1 Format. Figure 10-13 shows the format of the trap on coprocessor condition instruction, denoted by the cpTRAPcc mnemonic.

15	14	13	12	11 9	8	7	6	5	4	3	2 0
1	1	1	1	CpID	0	0	1	1	1	1	OPMODE
(RESERVED)								CONDITION SELECTOR			
OPTIONAL COPROCESSOR-DEFINED EXTENSION WORDS											
OPTIONAL WORD											
OR LONG-WORD OPERAND											

Figure 10-13. Trap On Coprocessor Condition (cpTRAPcc)

The F-line operation word contains the CpID field in bits [9–11] and 001111 in bits [8:3] to identify the cpTRAPcc instruction. Bits [0–2] of the cpTRAPcc F-line operation word specify the number of optional operand words in the instruction format. The instruction format can include zero, one, or two operand words.

The second word of the cpTRAPcc instruction format contains the coprocessor condition selector in bits [0–5] and should contain zeros in bits [6–15] to maintain compatibility with future M68000 products. This word is written to the condition CIR of the coprocessor to initiate execution of the cpTRAPcc instruction by the coprocessor.

If the coprocessor requires additional information to evaluate a condition, the instruction can include this information in extension words. These ex-

tension words follow the word containing the coprocessor condition selector field in the cpTRAPcc instruction format.

The operand words of the cpTRAPcc F-line operation word follow the coprocessor-defined extension words. These operand words are not explicitly used by the MC68EC030, but can be used to contain information referenced by the cpTRAPcc exception handling routines. The valid encodings for bits [0–2] of the F-line operation word and the corresponding numbers of operand words are listed in Table 10-1. Other encodings of these bits are invalid for the cpTRAPcc instruction.

Table 10-1. cpTRAPcc Opmode Encodings

Opmode	Optional Words in Instruction Format
010	One
011	Two
100	Zero

10.2.2.4.2 Protocol. Figure 10-8 shows the protocol for the cpTRAPcc instructions. The MC68EC030 transfers the condition selector to the coprocessor by writing the word following the operation word to the condition CIR. The main controller then reads the response CIR to determine its next action. The coprocessor can, using a response primitive, request any services necessary to evaluate the condition. If the coprocessor returns the true condition indicator, the main controller initiates exception processing for the cpTRAPcc exception (refer to **10.5.2.4 cpTRAPcc INSTRUCTION TRAPS**). If the coprocessor returns the false condition indicator, the main controller executes the next instruction in the instruction stream.

10.2.3 Coprocessor Save and Restore Instructions

The coprocessor context save and context restore instruction categories in the M68000 coprocessor interface support multitasking programming environments. In a multitasking environment, the context of a coprocessor may need to be changed asynchronously with respect to the operation of that coprocessor. That is, the coprocessor may be interrupted at any point in the execution of an instruction in the general or conditional category to begin context change operations.

In contrast to the general and conditional instruction categories, the context save and context restore instruction categories do not use the coprocessor response primitives. A set of format codes defined by the M68000 coprocessor interface communicates status information to the main controller during the execution of these instructions. These coprocessor format codes are discussed in detail in **10.2.3.2 COPROCESSOR FORMAT WORDS.**

10.2.3.1 COPROCESSOR INTERNAL STATE FRAMES. The context save (cpSAVE) and context restore (cpRESTORE) instructions transfer an internal coprocessor state frame between memory and a coprocessor. This internal coprocessor state frame represents the state of coprocessor operations. Using the cpSAVE and cpRESTORE instructions, it is possible to interrupt coprocessor operation, save the context associated with the current operation, and initiate coprocessor operations with a new context.

A cpSAVE instruction stores a coprocessor's internal state frame as a sequence of long-word entries in memory. Figure 10-14 shows the format of a coprocessor state frame. During execution of the cpSAVE instruction, the MC68EC030 calculates the state frame effective address from information in the operation word of the instruction and stores a format word at this effective address. The controller writes the long words that form the coprocessor state frame to descending memory addresses, beginning with the address specified by the sum of the effective address and the format word-length field multiplied by four. During execution of the cpRESTORE instruction, the MC68EC030 reads the format word and long words in the state frame from ascending addresses, beginning with the effective address specified in the instruction operation word.

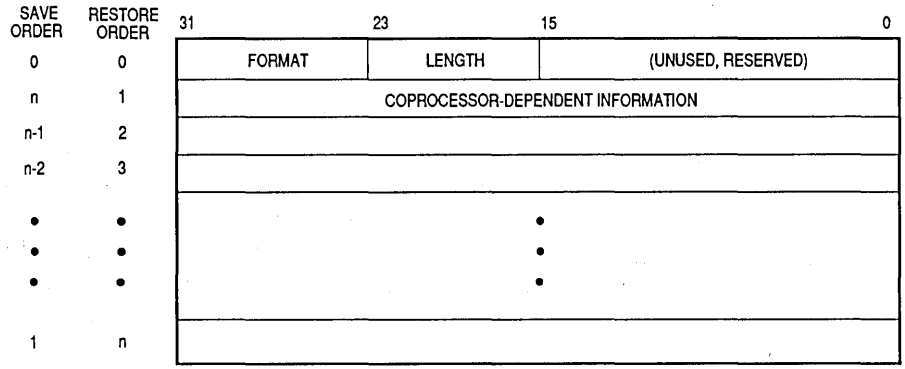


Figure 10-14. Coprocessor State Frame Format in Memory

The controller stores the coprocessor format word at the lowest address of the state frame in memory, and this word is the first word transferred for both the cpSAVE and the cpRESTORE instructions. The word following the format word does not contain information relevant to the coprocessor state frame, but serves to keep the information in the state frame a multiple of four bytes in size. The number of entries following the format word (at higher addresses) is determined.

The information in a coprocessor state frame describes a context of operation for that coprocessor. This description of a coprocessor context includes the program invisible state information and, optionally, the program visible state information. The program invisible state information consists of any internal registers or status information that cannot be accessed by the program but is necessary for the coprocessor to continue its operation at the point of suspension. Program visible state information includes the contents of all registers that appear in the coprocessor programming model and that can be directly accessed using the coprocessor instruction set. The information saved by the cpSAVE instruction must include the program invisible state information. If cpGEN instructions are provided to save the program visible state of the coprocessor, the cpSAVE and cpRESTORE instructions should only transfer the program invisible state information to minimize interrupt latency during a save or restore operation.

10.2.3.2 COPROCESSOR FORMAT WORDS. The coprocessor communicates status information to the main controller during the execution of cpSAVE and cpRESTORE instructions using coprocessor format words. The format words defined for the M68000 coprocessor interface are listed in Table 10-2.

Table 10-2. Coprocessor Format Word Encodings

Format Code	Length	Meaning
00	xx	Empty/Reset
01	xx	Not Ready, Come Again
02	xx	Invalid Format
03–0F	xx	Undefined, Reserved
10–FF	Length	Valid Format, Coprocessor Defined

The upper byte of the coprocessor format word contains the code used to communicate coprocessor status information to the main controller. The MC68EC030 recognizes four types of format words: empty/reset, not ready, invalid format, and valid format. The MC68EC030 interprets the reserved

format codes ($03–$0F) as invalid format words. The lower byte of the coprocessor format word specifies the size in bytes (which must be a multiple of four) of the coprocessor state frame. This value is only relevant when the code byte contains the valid format code (refer to **10.2.3.2.4 Valid Format Word**).

10.2.3.2.1 Empty/Reset Format Word. The coprocessor returns the empty/reset format code during a cpSAVE instruction to indicate that the coprocessor contains no user-specific information. That is, no coprocessor instructions have been executed since either a previous cpRESTORE of an empty/reset format code or the previous hardware reset. If the main controller reads the empty/reset format word from the save CIR during the initiation of a cpSAVE instruction, it stores the format word at the effective address specified in the cpSAVE instruction and executes the next instruction.

When the main controller reads the empty/reset format word from memory during the execution of the cpRESTORE instruction, it writes the format word to the restore CIR. The main controller then reads the restore CIR and, if the coprocessor returns the empty/reset format word, executes the next instruction. The main controller can initialize the coprocessor by writing the empty/reset format code to the restore CIR. When the coprocessor receives the empty/reset format code, it terminates any current operations and waits for the main controller to initiate the next coprocessor instruction. In particular, after the cpRESTORE of the empty/reset format word, the execution of a cpSAVE should cause the empty/reset format word to be returned when a cpSAVE instruction is executed before any other coprocessor instructions. Thus, an empty/reset state frame consists only of the format word and the following reserved word in memory (refer to Figure 10-14).

10.2.3.2.2 Not Ready Format Word. When the main controller initiates a cpSAVE instruction by reading the save CIR the coprocessor can delay the save operation by returning a not ready format word. The main controller then services any pending interrupts and reads the save CIR again. The not ready format word delays the save operation until the coprocessor is ready to save its internal state. The cpSAVE instruction can suspend execution of a general or conditional coprocessor instruction; the coprocessor can resume execution of the suspended instruction when the appropriate state is restored with a cpRESTORE. If no further main controller services are required to complete coprocessor instruction execution, it may be more efficient to complete the instruction and thus reduce the size of the saved state. The coprocessor designer should consider the efficiency of completing the instruction or of

suspending and later resuming the instruction when the main controller executes a cpSAVE instruction.

When the main controller initiates a cpRESTORE instruction by writing a format word to the restore CIR, the coprocessor should usually terminate any current operations and restore the state frame supplied by the main controller. Thus, the not ready format word should usually not be returned by the coprocessor during the execution of a cpRESTORE instruction. If the coprocessor must delay the cpRESTORE operation for any reason, it can return the not ready format word when the main controller reads the restore CIR. If the main controller reads the not ready format word from the restore CIR during the cpRESTORE instruction, it reads the restore CIR again without servicing any pending interrupts.

10.2.3.2.3 Invalid Format Word. When the format word placed in the restore CIR to initiate a cpRESTORE instruction does not describe a valid coprocessor state frame, the coprocessor returns the invalid format word in the restore CIR. When the main controller reads this format word during the cpRESTORE instruction, it writes the abort mask to the control CIR and initiates format error exception processing. The two least significant bits of the abort mask are 01; the fourteen most significant bits are undefined.

A coprocessor should usually not place an invalid format word in the save CIR when the main controller initiates a cpSAVE instruction. A coprocessor, however, may not be able to support the initiation of a cpSAVE instruction while it is executing a previously initiated cpSAVE or cpRESTORE instruction. In this situation, the coprocessor can return the invalid format word when the main controller reads the save CIR to initiate the cpSAVE instruction while either another cpSAVE or cpRESTORE instruction is executing. If the main controller reads an invalid format word from the save CIR, it writes the abort mask to the control CIR and initiates format error exception processing (refer to **10.5.1.5 FORMAT ERRORS**).

10.2.3.2.4 Valid Format Word. When the main controller reads a valid format word from the save CIR during the cpSAVE instruction, it uses the length field to determine the size of the coprocessor state frame to save. The length field in the lower eight bits of a format word is relevant only in a valid format word. During the cpRESTORE instruction, the main controller uses the length field in the format word read from the effective address in the instruction to determine the size of the coprocessor state frame to restore.

The length field of a valid format word, representing the size of the coprocessor state frame, must contain a multiple of four. If the main controller detects a value that is not a multiple of four in a length field during the execution of a cpSAVE or cpRESTORE instruction, the main controller writes the abort mask (refer to **10.2.3.2.3 Invalid Format Word**) to the control CIR and initiates format error exception processing.

10.2.3.3 COPROCESSOR CONTEXT SAVE INSTRUCTION. The M68000 coprocessor context save instruction category consists of one instruction. The coprocessor context save instruction, denoted by the cpSAVE mnemonic, saves the context of a coprocessor dynamically without relation to the execution of coprocessor instructions in the general or conditional instruction categories. During the execution of a cpSAVE instruction, the coprocessor communicates status information to the main controller by using the coprocessor format codes.

10.2.3.3.1 Format. Figure 10-15 shows the format of the cpSAVE instruction. The first word of the instruction is the F-line operation word, which contains the coprocessor identification code in bits [9–11] and an M68000 effective address code in bits [0–5]. The effective address encoded in the cpSAVE instruction is the address at which the state frame associated with the current context of the coprocessor is saved in memory.

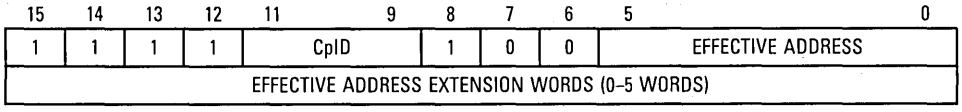


Figure 10-15. Coprocessor Context Save Instruction Format (cpSAVE)

The control alterable and predecrement addressing modes are valid for the cpSAVE instruction. Other addressing modes cause the MC68EC030 to initiate F-line emulator exception processing as described in **10.5.2.2 F-LINE EMULATOR EXCEPTIONS.**

The instruction can include as many as five effective address extension words following the cpSAVE instruction operation word. These words contain any additional information required to calculate the effective address specified by bits [0–5] of the operation word.

10.2.3.3.2 Protocol. Figure 10-16 shows the protocol for the coprocessor context save instruction. The main controller initiates execution of the cpSAVE instruction by reading the save CIR. Thus, the cpSAVE instruction is the only coprocessor instruction that begins by reading from a CIR. (All other coprocessor instructions write to a CIR to initiate execution of the instruction by the coprocessor.) The coprocessor communicates status information associated with the context save operation to the main controller by placing coprocessor format codes in the save CIR.

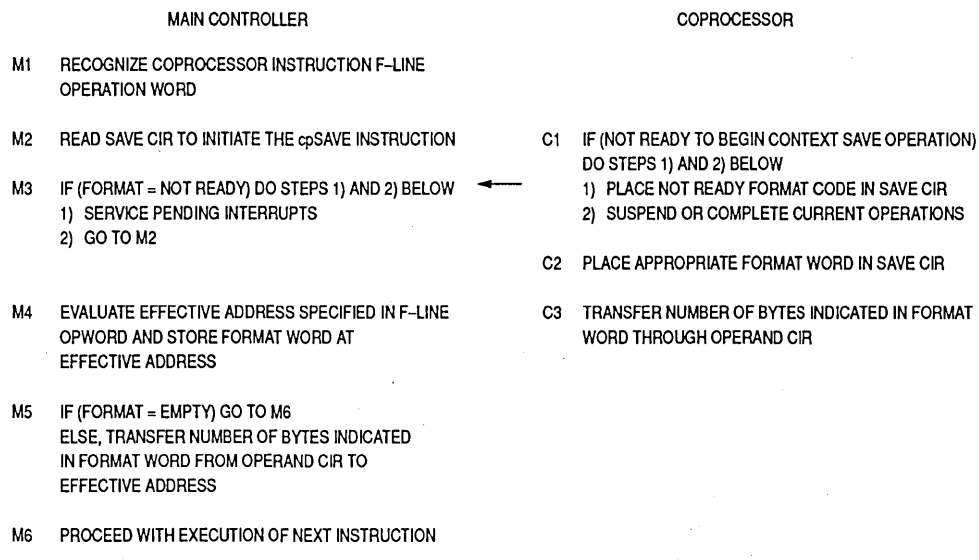


Figure 10-16. Coprocessor Context Save Instruction Protocol

If the coprocessor is not ready to suspend its current operation when the main controller reads the save CIR, it returns a "not ready" format code. The main controller services any pending interrupts and then reads the save CIR again. After placing the not ready format code in the save CIR, the coprocessor should either suspend or complete the instruction it is currently executing.

Once the coprocessor has suspended or completed the instruction it is executing, it places a format code representing the internal coprocessor state in the save CIR. When the main controller reads the save CIR, it transfers the format word to the effective address specified in the cpSAVE instruction. The lower byte of the coprocessor format word specifies the number of bytes of state information, not including the format word and associated null word, to be transferred from the coprocessor to the effective address specified. If

the state information is not a multiple of four bytes in size, the MC68EC030 initiates format error exception processing (refer to **10.5.1.5 FORMAT ERRORS**). The coprocessor and main controller coordinate the transfer of the internal state of the coprocessor using the operand CIR. The MC68EC030 completes the coprocessor context save by repeatedly reading the operand CIR and writing the information obtained into memory until all the bytes specified in the coprocessor format word have been transferred. Following a cpSAVE instruction, the coprocessor should be in an idle state — that is, not executing any coprocessor instructions.

The cpSAVE instruction is a privileged instruction. When the main controller identifies a cpSAVE instruction, it checks the supervisor bit in the status register to determine whether it is operating at the supervisor privilege level. If the MC68EC030 attempts to execute a cpSAVE instruction while at the user privilege level (status register bit [13] = 0), it initiates privilege violation exception processing without accessing any of the coprocessor interface registers (refer to **10.5.2.3 PRIVILEGE VIOLATIONS**).

The MC68EC030 initiates format error exception processing if it reads an invalid format word (or a valid format word whose length field is not a multiple of four bytes) from the save CIR during the execution of a cpSAVE instruction (refer to **10.2.3.2.3 Invalid Format Word**). The MC68EC030 writes an abort mask (refer to **10.2.3.2.3 Invalid Format Word**) to the control CIR to abort the coprocessor instruction prior to beginning exception processing. Figure 10-16 does not include this case since a coprocessor usually returns either a not ready or a valid format code in the context of the cpSAVE instruction. The coprocessor can return the invalid format word, however, if a cpSAVE is initiated while the coprocessor is executing a cpSAVE or cpRESTORE instruction and the coprocessor is unable to support the suspension of these two instructions.

10.2.3.4 COPROCESSOR CONTEXT RESTORE INSTRUCTION. The M68000 coprocessor context restore instruction category includes one instruction. The coprocessor context restore instruction, denoted by the cpRESTORE mnemonic, forces a coprocessor to terminate any current operations and to restore a former state. During the execution of a cpRESTORE instruction, the coprocessor can communicate status information to the main controller by placing format codes in the restore CIR.

10.2.3.4.1 Format. Figure 10-17 shows the format of the cpRESTORE instruction.

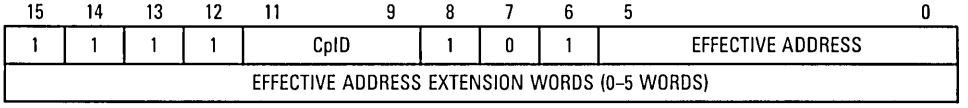


Figure 10-17. Coprocessor Context Restore Instruction Format (cpRESTORE)

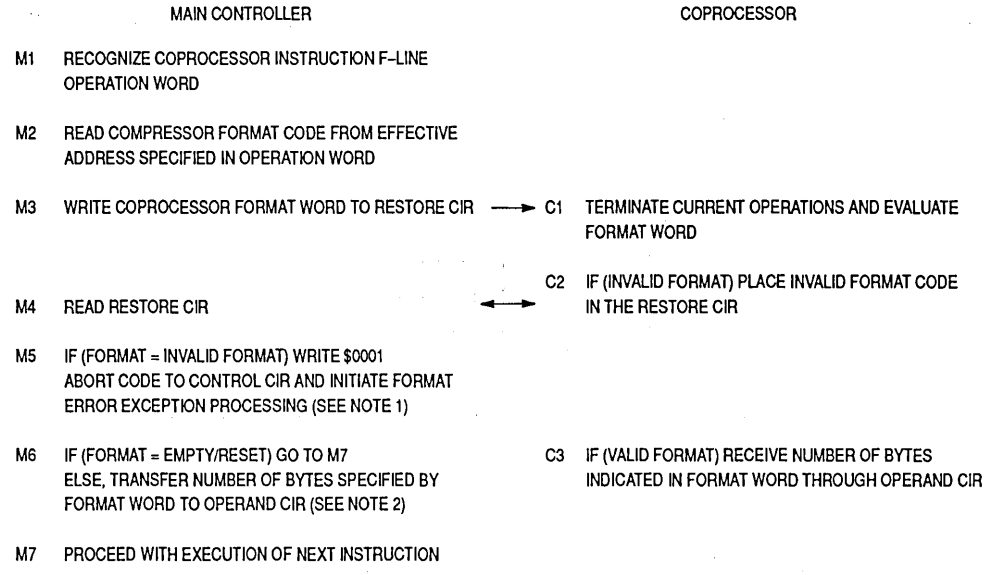
The first word of the instruction is the F-line operation word, which contains the coprocessor identification code in bits [9–11] and an M68000 effective addressing code in bits [0–5]. The effective address encoded in the cpRESTORE instruction is the starting address in memory where the coprocessor context is stored. The effective address is that of the coprocessor format word that applies to the context to be restored to the coprocessor.

The instruction can include as many as five effective address extension words following the first word in the cpRESTORE instruction format. These words contain any additional information required to calculate the effective address specified by bits [0–5] of the operation word.

All memory addressing modes except the predecrement addressing mode are valid. Invalid effective address encodings cause the MC68EC030 to initiate F-line emulator exception processing (refer to **10.5.2.2 F-LINE EMULATOR EXCEPTIONS**).

10.2.3.4.2 Protocol. Figure 10-18 shows the protocol for the coprocessor context restore instruction. When the main controller executes a cpRESTORE instruction, it first reads the coprocessor format word from the effective address in the instruction. This format word contains a format code and a length field. During cpRESTORE operation, the main controller retains a copy of the length field to determine the number of bytes to be transferred to the coprocessor during the cpRESTORE operation and writes the format word to the restore CIR to initiate the coprocessor context restore.

When the coprocessor receives the format word in the restore CIR, it must terminate any current operations and evaluate the format word. If the format word represents a valid coprocessor context as determined by the coprocessor design, the coprocessor returns the format word to the main controller through the restore CIR and prepares to receive the number of bytes specified in the format word through its operand CIR.



NOTES: 1. See 10.6.1.5 FORMAT ERROR.
2. The MC68EC030 uses the length field in the format word read during M2 to determine the number of bytes to read from memory and write to the operand CIR.

Figure 10-18. Coprocessor Context Restore Instruction Protocol

After writing the format word to the restore CIR the main controller continues the cpRESTORE dialog by reading that same register. If the coprocessor returns a valid format word, the main controller transfers the number of bytes specified by the format word at the effective address to the operand CIR.

If the format word written to the restore CIR does not represent a valid coprocessor state frame, the coprocessor places an invalid format word in the restore CIR and terminates any current operations. The main controller receives the invalid format code, writes an abort mask (refer to **10.2.3.2.3 Invalid Format Word**) to the control CIR, and initiates format error exception processing (refer to **10.5.1.5 FORMAT ERRORS**).

The cpRESTORE instruction is a privileged instruction. When the main controller accesses a cpRESTORE instruction, it checks the supervisor bit in the status register. If the MC68EC030 attempts to execute a cpRESTORE instruction while at the user privilege level (status register bit [13]=0), it initiates privilege violation exception processing without accessing any of the coprocessor interface registers (refer to **10.5.2.3 PRIVILEGE VIOLATIONS**).

10.3 COPROCESSOR INTERFACE REGISTER SET

The instructions of the M68000 coprocessor interface use registers of the CIR set to communicate with the coprocessor. These CIRs are not directly related to the coprocessor's programming model.

Figure 10-4 is a memory map of the CIR set. The registers denoted by asterisks (*) must be included in a coprocessor interface that implements coprocessor instructions in all four categories. The complete register model must be implemented if the system uses all of the coprocessor response primitives defined for the M68000 coprocessor interface.

The following paragraphs contain detailed descriptions of the registers.

10.3.1 Response CIR

The coprocessor uses the 16-bit response CIR to communicate all service requests (coprocessor response primitives) to the main controller. The main controller reads the response CIR to receive the coprocessor response primitives during the execution of instructions in the general and conditional instruction categories. The offset from the base address of the CIR set for the response CIR is $00. Refer to **10.4 COPROCESSOR RESPONSE PRIMITIVES.**

10.3.2 Control CIR

The main controller writes to the 2-bit control CIR to acknowledge coprocessor-requested exception processing or to abort the execution of a coprocessor instruction. The offset from the base address of the CIR set for the control CIR is $02. The control CIR occupies the two least significant bits of the word at that offset. The 14 most significant bits of the word are undefined. Figure 10-19 shows the format of this register.

Figure 10-19. Control CIR Format

When the MC68EC030 receives one of the three take exception coprocessor response primitives, it acknowledges the primitive by writing the exception acknowledge mask (102) to the control CIR, which sets the exception acknowledge (XA) bit. The MC68EC030 writes the abort mask (012), which sets

the abort (AB) bit, to the control CIR to abort any coprocessor instruction in progress. (The most significant 14 bits of both masks are undefined.) The MC68EC030 aborts a coprocessor instruction when it detects one of the following exception conditions:

- An F-line emulator exception condition after reading a response primitive
- A privilege violation exception as it performs a supervisor check in response to a supervisor check primitive
- A format error exception when it receives an invalid format word or a valid format word that contains an invalid length

10.3.3 Save CIR

The coprocessor uses the 16-bit save CIR to communicate status and state frame format information to the main controller while executing a cpSAVE instruction. The main controller reads the save CIR to initiate execution of the cpSAVE instruction by the coprocessor. The offset from the base address of the CIR set for the save CIR is $04. Refer to **10.2.3.2 COPROCESSOR FORMAT WORDS.**

10.3.4 Restore CIR

The main controller initiates the cpRESTORE instruction by writing a coprocessor format word to the 16-bit restore register. During the execution of the cpRESTORE instruction, the coprocessor communicates status and state frame format information to the main controller through the restore CIR. The offset from the base address of the CIR set for the restore CIR is $06. Refer to **10.2.3.2 COPROCESSOR FORMAT WORDS.**

10.3.5 Operation Word CIR

The main controller writes the F-line operation word of the instruction in progress to the 16-bit operation word CIR in response to a transfer operation word coprocessor response primitive (refer to **10.4.6 Transfer Operation Word Primitive**). The offset from the base address of the CIR set for the operation word CIR is $08.

10.3.6 Command CIR

The main controller initiates a general category instruction by writing the instruction command word, which follows the instruction F-line operation

word in the instruction stream, to the 16-bit command CIR. The offset from the base address of the CIR set for the command CIR is $0A.

10.3.7 Condition CIR

The main controller initiates a conditional category instruction by writing the condition selector to the 16-bit condition CIR. The offset from the base address of the CIR set for the condition CIR is $0E. Figure 10-20 shows the format of the condition CIR.

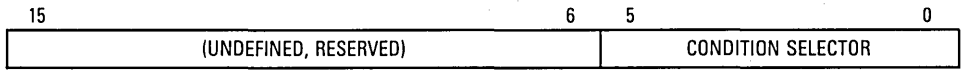


Figure 10-20. Condition CIR Format

10.3.8 Operand CIR

When the coprocessor requests the transfer of an operand, the main controller performs the transfer by reading from or writing to the 32-bit operand CIR. The offset from the base address of the CIR set for the operand CIR is $10.

The MC68EC030 aligns all operands transferred to and from the operand CIR to the most significant byte of this CIR. The controller performs a sequence of long-word transfers to read or write any operand larger than four bytes. If the operand size is not a multiple of four bytes, the portion remaining after the initial long-word transfers is aligned to the most significant byte of the operand CIR. Figure 10-21 shows the operand alignment used by the MC68EC030 when accessing the operand CIR.

10

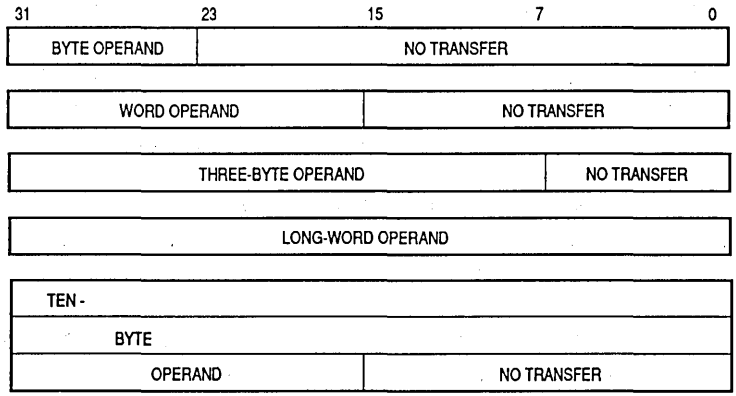


Figure 10-21. Operand Alignment for Operand CIR Accesses

10.3.9 Register Select CIR

When the coprocessor requests the transfer of one or more main controller registers or a group of coprocessor registers, the main controller reads the 16-bit register select CIR to identify the number or type of registers to be transferred. The offset from the base address of the CIR set for the register select CIR is $14. The format of this register depends on the primitive that is currently using it. Refer to **10.4 COPROCESSOR RESPONSE PRIMITIVES**.

10.3.10 Instruction Address CIR

When the coprocessor requests the address of the instruction it is currently executing, the main controller transfers this address to the 32-bit instruction address CIR. Any transfer of the scanPC is also performed through the instruction address CIR (refer to **10.4.17 Transfer Status Register and ScanPC Primitive**). The offset from the base address of the CIR set for the instruction address CIR is $18.

10.3.11 Operand Address CIR

When a coprocessor requests an operand address transfer between the main controller and the coprocessor, the address is transferred through the 32-bit operand address CIR. The offset from the base address of the CIR set for the operand address CIR is $1C.

10.4 COPROCESSOR RESPONSE PRIMITIVES

The response primitives are primitive instructions that the coprocessor issues to the main controller during the execution of a coprocessor instruction. The coprocessor uses response primitives to communicate status information and service requests to the main controller. In response to an instruction command word written to the command CIR or a condition selector in the condition CIR, the coprocessor returns a response primitive in the response CIR. Within the general and conditional instruction categories, individual instructions are distinguished by the operation of the coprocessor hardware and also by services specified by coprocessor response primitives provided by the main controller.

Subsequent paragraphs, beginning with **10.4.2 Coprocessor Response Primitive General Format**, consist of detailed descriptions of the M68000 coprocessor response primitives supported by the MC68EC030. Any response

primitive that the MC68EC030 does not recognize causes it to initiate protocol violation exception processing (refer to **10.5.2.1 PROTOCOL VIOLATIONS**). This processing of undefined primitives supports emulation of extensions to the M68000 coprocessor response primitive set by the protocol violation exception handler. Exception processing related to the coprocessor interface is discussed in **10.5 EXCEPTIONS**.

10.4.1 ScanPC

Several of the response primitives involve the scanPC, and many of them require the main controller to use it while performing services requested. These paragraphs describe the scanPC and tell how it operates.

During the execution of a coprocessor instruction, the program counter in the MC68EC030 contains the address of the F-line operation word of that instruction. A second register, called the scanPC, sequentially addresses the remaining words of the instruction.

If the main controller requires extension words to calculate an effective address or destination address of a branch operation, it uses the scanPC to address these extension words in the instruction stream. Also, if a coprocessor requests the transfer of extension words, the scanPC addresses the extension words during the transfer. As the controller references each word, it increments the scanPC to point to the next word in the instruction stream. When an instruction is completed, the controller transfers the value in the scanPC to the program counter to address the operation word of the next instruction.

The value in the scanPC when the main controller reads the first response primitive after beginning to execute an instruction depends on the instruction being executed. For a cpGEN instruction, the scanPC points to the word following the coprocessor command word. For the cpBcc instructions, the scanPC points to the word following the instruction F-line operation word. For the cpScc, cpTRAPcc, and cpDBcc instructions, the scanPC points to the word following the coprocessor condition specifier word.

If a coprocessor implementation uses optional instruction extension words with a general or conditional instruction, the coprocessor must use these words consistently so that the scanPC is updated accordingly during the instruction execution. Specifically, during the execution of general category instructions, when the coprocessor terminates the instruction protocol, the MC68EC030 assumes that the scanPC is pointing to the operation word of

the next instruction to be executed. During the execution of conditional category instructions, when the coprocessor terminates the instruction protocol, the MC68EC030 assumes that the scanPC is pointing to the word following the last of any coprocessor-defined extension words in the instruction format.

10.4.2 Coprocessor Response Primitive General Format

The M68000 coprocessor response primitives are encoded in a 16-bit word that is transferred to the main controller through the response CIR. Figure 10-22 shows the format of the coprocessor response primitives.

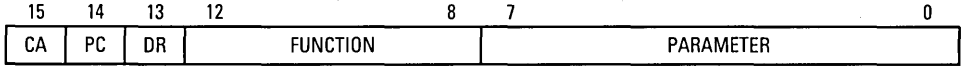


Figure 10-22. Coprocessor Response Primitive Format

The encoding of bits [0–12] of a coprocessor response primitive depends on the individual primitive. Bits [13–15], however, specify optional additional operations that apply to most of the primitives defined for the M68000 coprocessor interface.

Bit [15], the CA bit, specifies the "come again" operation of the main controller. When the main controller reads a response primitive from the response CIR with the come again bit set to one, it performs the service indicated by the primitive and then reads the response CIR again. Using the CA bit, a coprocessor can transfer several response primitives to the main controller during the execution of a single coprocessor instruction.

Bit [4], the PC bit, specifies the pass program counter operation. When the main controller reads a primitive with the PC bit set from the response CIR, the main controller immediately passes the current value in its program counter to the instruction address CIR as the first operation in servicing the primitive request. The value in the program counter is the address of the F-line operation word of the coprocessor instruction currently executing. The PC bit is implemented in all of the coprocessor response primitives currently defined for the M68000 coprocessor interface.

When an undefined primitive or a primitive that requests an illegal operation is passed to the main controller, the main controller initiates exception processing for either an F-line emulator or a protocol violation exception (refer to **10.5.2 Main-Controller-Detected Exceptions**). If the PC bit is set in one of

these response primitives, however, the main controller passes the program counter to the instruction address CIR before it initiates exception processing.

When the main controller initiates a cpGEN instruction that can be executed concurrently with main controller instructions, the PC bit is usually set in the first primitive returned by the coprocessor. Since the main controller proceeds with instruction stream execution once the coprocessor releases it, the coprocessor must record the instruction address to support any possible exception processing related to the instruction. Exception processing related to concurrent coprocessor instruction execution is discussed in **10.5.1 Coprocesssor-Detected Exceptions.**

Bit [13], the DR bit, is the direction bit. It applies to operand transfers between the main controller and the coprocessor. If DR = 0, the direction of transfer is from the main controller to the coprocessor (main controller write). If DR = 1, the direction of transfer is from the coprocessor to the main controller (main controller read). If the operation indicated by a given response primitive does not involve an explicit operand transfer, the value of this bit depends on the particular primitive encoding.

10.4.3 Busy Primitive

The busy response primitive causes the main controller to reinitiate a coprocessor instruction. This primitive applies to instructions in the general and conditional categories. Figure 10-23 shows the format of the busy primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	1	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 10-23. Busy Primitive Format

This primitive uses the PC bit as previously described.

Coprocessors that can operate concurrently with the main controller but cannot buffer write operations to their command or condition CIR use the busy primitive. A coprocessor may execute a cpGEN instruction concurrently with an instruction in the main controller. If the main controller attempts to initiate an instruction in the general or conditional instruction category while the coprocessor is concurrently executing a cpGEN instruction, the coprocessor can place the busy primitive in the response CIR. When the main controller reads this primitive, it services pending interrupts (using a pre-

instruction exception stack frame, refer to Figure 10-41). The controller then restarts the general or conditional coprocessor instruction that it had attempted to initiate earlier.

The busy primitive should only be used in response to a write to the command or condition CIR. It should be the first primitive returned after the main controller attempts to initiate a general or conditional category instruction. In particular, the busy primitive should not be issued after program-visible resources have been altered by the instruction. (Program-visible resources include coprocessor and main controller program-visible registers and operands in memory, but not the scanPC.) The restart of an instruction after it has altered program-visible resources causes those resources to have inconsistent values when the controller reinitiates the instruction.

The MC68EC030 responds to the busy primitive differently in a special case that can occur during a breakpoint operation (refer to **8.1.12 Multiple Exceptions**). This special case occurs when a breakpoint acknowledge cycle initiates a coprocessor F-line instruction, the coprocessor returns the busy primitive in response to the instruction initiation, and an interrupt is pending. When these three conditions are met, the controller re-executes the breakpoint acknowledge cycle after the interrupt exception processing has been completed. A design that uses a breakpoint to monitor the number of passes through a loop by incrementing or decrementing a counter may not work correctly under these conditions. This special case may cause several breakpoint acknowledge cycles to be executed during a single pass through a loop.

10

10.4.4 Null Primitive

The null coprocessor response primitive communicates coprocessor status information to the main controller. This primitive applies to instructions in the general and conditional categories. Figure 10-24 shows the format of the null primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	0	IA	0	0	0	0	0	0	PF	TF

Figure 10-24. Null Primitive Format

This primitive uses the CA and PC bits as previously described.

Bit [8], the IA bit, specifies the interrupts allowed optional operation. This bit determines whether the MC68EC030 services pending interrupts prior to re-reading the response CIR after receiving a null primitive. Interrupts are allowed when the IA bit is set.

Bit [1], the PF bit, shows the "processing finished" status of the coprocessor. That is, PF=1 indicates that the coprocessor has completed all processing associated with an instruction.

Bit [0], the TF bit, indicates the true/false condition during the execution of a conditional category instruction. TF=1 is the true condition specifier, and TF=0 is the false condition specifier. The TF bit is only relevant for null primitives with CA=0 that are used by the coprocessor during the execution of a conditional instruction.

The MC68EC030 processes a null primitive with CA=1 in the same manner whether executing a general or conditional category coprocessor instruction. If the coprocessor sets CA and IA to one in the null primitive, the main controller services pending interrupts (using a mid-instruction stack frame, refer to Figure 10-43) and reads the response CIR again. If the coprocessor sets CA to one and IA to zero in the null primitive, the main controller reads the response CIR again without servicing any pending interrupts.

A null, CA=0 primitive provides a condition evaluation indicator to the main controller during the execution of a conditional instruction and ends the dialogue between the main controller and coprocessor for that instruction. The main controller completes the execution of a conditional category coprocessor instruction when it receives the primitive. The PF bit is not relevant during conditional instruction execution since the primitive itself implies completion of processing.

Usually, when the main controller reads any primitive that does not have CA=1 while executing a general category instruction, it terminates the dialogue between the main controller and coprocessor. If a trace exception is pending, however, the main controller does not terminate the instruction dialogue until it reads a null, CA=0, PF=1 primitive from the response CIR (refer to **10.5.2.5 TRACE EXCEPTIONS**). Thus, the main controller continues to read the response CIR until it receives a null, CA=0, PF=1 primitive, and then performs trace exception processing. When IA=1, the main controller services pending interrupts before reading the response CIR again.

A coprocessor can be designed to execute a cpGEN instruction concurrently with the execution of main controller instructions and, also, buffer one write

operation to either its command or condition CIR. This type of coprocessor issues a null primitive with CA = 1 when it is concurrently executing a cpGEN instruction, and the main controller initiates another general or conditional coprocessor instruction. This primitive indicates that the coprocessor is busy and the main controller should read the response CIR again without reinitiating the instruction. The IA bit of this null primitive usually should be set to minimize interrupt latency while the main controller is waiting for the coprocessor to complete the general category instruction.

Table 10-3 summarizes the encodings of the null primitive.

Table 10-3. Null Coprocessor Response Primitive Encodings

CA	PC	IA	PF	TF	General Instructions	Conditional Instructions
x	1	x	x	x	Pass Program Counter to Instruction Address CIR, Clear PC Bit, and Proceed with Operation Specified by CA, IA, PF, and TF Bits	Same as General Category
1	0	0	x	x	Reread Response CIR, Do Not Service Pending Interrupts	Same as General Category
1	0	1	x	x	Service Pending Interrupts and Reread the Response CIR	Same as General Category
0	0	0	0	c	If (Trace Pending) Reread Response CIR; Else, Execute Next Instruction	Main Controller Completes Instruction Execution Based on TF = c
0	0	1	0	c	If (Trace Pending) Service Pending Interrupts and Reread Response CIR; Else, Execute Next Instruction	Main Controller Completes Instruction Execution Based on TF = c
0	0	x	1	c	Coprocessor Instruction Completed; Service Pending Exceptions or Execute Next Instruction	Main Controller Completes Instruction Execution Based on TF = c.

x = Don't Care
c = 1 or 0 Depending on Coprocessor Condition Evaluation

10.4.5 Supervisor Check Primitive

The supervisor check primitive verifies that the main controller is operating in the supervisor state while executing a coprocessor instruction. This primitive applies to instructions in the general and conditional coprocessor instruction categories. Figure 10-25 shows the format of the supervisor check primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Figure 10-25. Supervisor Check Primitive Format

This primitive uses the PC bit as previously described. Bit [15] is shown as one, but during execution of a general category instruction, this primitive performs the same operations regardless of the value of bit [15]. If this primitive is issued with bit [15] = 0 during a conditional category instruction, however, the main controller initiates protocol violation exception processing.

When the main controller reads the supervisor check primitive from the response CIR, it checks the value of the S bit in the status register. If S = 0 (main controller operating at user privilege level), the main controller aborts the coprocessor instruction by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR. The main controller then initiates privilege violation exception processing (refer to **10.5.2.3 PRIVILEGE VIOLATIONS**). If the main controller is at the supervisor privilege level when it receives this primitive, it reads the response CIR again.

The supervisor check primitive allows privileged instructions to be defined in the coprocessor general and conditional instruction categories. This primitive should be the first one issued by the coprocessor during the dialog for an instruction that is implemented as privileged.

10.4.6 Transfer Operation Word Primitive

The transfer operation word primitive requests a copy of the coprocessor instruction operation word for the coprocessor. This primitive applies to general and conditional category instructions. Figure 10-26 shows the format of the transfer operation word primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	0	1	1	1	0	0	0	0	0	0	0	0

Figure 10-26. Transfer Operation Word Primitive Format

This primitive uses the CA and PC bits as previously described. If this primitive is issued with CA = 0 during a conditional category instruction, the main controller initiates protocol violation exception processing.

When the main controller reads this primitive from the response CIR, it transfers the F-line operation word of the currently executing coprocessor instruction to the operation word CIR. The value of the scanPC is not affected by this primitive.

10.4.7 Transfer from Instruction Stream Primitive

The transfer from instruction stream primitive initiates transfers of operands from the instruction stream to the coprocessor. This primitive applies to general and conditional category instructions. Figure 10-27 shows the format of the transfer from instruction stream primitive.

15	14	13	12	11	10	9	8	7 … 0
CA	PC	0	0	1	1	1	1	LENGTH

Figure 10-27. Transfer from Instruction Stream Primitive Format

This primitive uses the CA and PC bits as previously described. If this primitive is issued with CA=0 during a conditional category instruction, the main controller initiates protocol violation exception processing.

Bits [0–7] of the primitive format specify the length, in bytes, of the operand to be transferred from the instruction stream to the coprocessor. The length must be an even number of bytes. If an odd length is specified, the main controller initiates protocol violation exception processing (refer to **10.5.2.1 PROTOCOL VIOLATIONS**).

This primitive transfers coprocessor-defined extension words to the coprocessor. When the main controller reads this primitive from the response CIR, it copies the number of bytes indicated by the length field from the instruction stream to the operand CIR. The first word or long word transferred is at the location pointed to by the scanPC when the primitive is read by the main controller, and the scanPC is incremented after each word or long word is transferred. When execution of the primitive has completed, the scanPC has been incremented by the total number of bytes transferred and points to the word following the last word transferred. The main controller transfers the operands from the instruction stream using a sequence of long-word writes to the operand CIR. If the length field is not an even multiple of four bytes, the last two bytes from the instruction stream are transferred using a word write to the operand CIR.

10.4.8 Evaluate and Transfer Effective Address Primitive

The evaluate and transfer effective address primitive evaluates the effective address specified in the coprocessor instruction operation word and transfers the result to the coprocessor. This primitive applies to general category instructions. If this primitive is issued by the coprocessor during the execution of a conditional category instruction, the main controller initiates protocol violation exception processing. Figure 10-28 shows the format of the evaluate and transfer effective address primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	1	0	0	0	0	0	0	0	0	0

Figure 10-28. Evaluate and Transfer Effective Address Primitive Format

This primitive uses the CA and PC bits as previously described.

When the main controller reads this primitive while executing a general category instruction, it evaluates the effective address specified in the instruction. At this point, the scanPC contains the address of the first of any required effective address extension words. The main controller increments the scanPC by two after it references each of these extension words. After the effective address is calculated, the resulting 32-bit value is written to the operand address CIR.

The MC68EC030 only calculates effective addresses for control alterable addressing modes in response to this primitive. If the addressing mode in the operation word is not a control alterable mode, the main controller aborts the instruction by writing a $0001 to the control CIR and initiates F-line emulation exception processing (refer to **10.5.2.2 F-LINE EMULATOR EXCEPTIONS**).

10.4.9 Evaluate Effective Address and Transfer Data Primitive

The evaluate effective address and transfer data primitive transfers an operand between the coprocessor and the effective address specified in the coprocessor instruction operation word. This primitive applies to general category instructions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main controller initiates protocol violation exception processing. Figure 10-29 shows the format of the evaluate effective address and transfer data primitive.

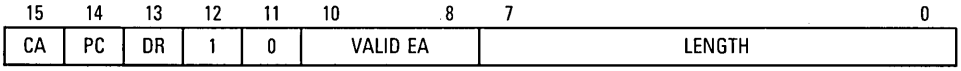


Figure 10-29. Evaluate Effective Address and Transfer Data Primitive Format

This primitive uses the CA, PC, and DR bits as previously described.

The valid effective address field (bits [8–10]) of the primitive format specifies the valid effective address categories for this primitive. If the effective address specified in the instruction operation word is not a member of the class specified by bits [8–10], the main controller aborts the coprocessor instruction by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR and by initiating F-line emulation exception processing. Table 10-4 lists the valid effective address field encodings.

Table 10-4. Valid Effective Address Codes

Field	Category
000	Control Alterable
001	Data Alterable
010	Memory Alterable
011	Alterable
100	Control
101	Data
110	Memory
111	Any Effective Address (No Restriction)

Even when the valid effective address fields specified in the primitive and in the instruction operation word match, the MC68EC030 initiates protocol violation exception processing if the primitive requests a write to a nonalterable effective address.

The length in bytes of the operand to be transferred is specified by bits [0–7] of the primitive format. Several restrictions apply to operand lengths for certain effective addressing modes. If the effective address is a main controller register (register direct mode), only operand lengths of one, two, or four bytes are valid; all other lengths (zero, for example) cause the main controller to initiate protocol violation exception processing. Operand lengths of 0–255 bytes are valid for the memory addressing modes.

The length of 0–255 bytes does not apply to an immediate operand. The length of an immediate operand must be one byte or an even number of bytes (less than 256), and the direction of transfer must be to the coprocessor; otherwise, the main controller initiates protocol violation exception processing.

When the main controller receives this primitive during the execution of a general category instruction, it verifies that the effective address encoded in the instruction operation word is in the category specified by the primitive. If so, the controller calculates the effective address using the appropriate effective address extension words at the current scanPC address and increments the scanPC by two for each word referenced. The main controller then transfers the number of bytes specified in the primitive between the operand CIR and the effective address using long-word transfers whenever possible. Refer to **10.3.8 Operand CIR** for information concerning operand alignment for transfers involving the operand CIR.

The DR bit specifies the direction of the operand transfer. DR = 0 requests a transfer from the effective address to the operand CIR, and DR = 1 specifies a transfer from the operand CIR to the effective address.

If the effective addressing mode specifies the predecrement mode, the address register used is decremented by the size of the operand before the transfer. The bytes within the operand are then transferred to or from ascending addresses beginning with the location specified by the decremented address register. In this mode, if A7 is used as the address register and the operand length is one byte, A7 is decremented by two to maintain a word-aligned stack.

For the postincrement effective addressing mode, the address register used is incremented by the size of the operand after the transfer. The bytes within the operand are transferred to or from ascending addresses beginning with the location specified by the address register. In this mode, if A7 is used as the address register and the operand length is one byte, A7 is incremented by two after the transfer to maintain a word aligned stack. Transferring odd length operands longer than one byte using the −(A7) or (A7)+ addressing modes can result in a stack pointer that is not word aligned.

The controller repeats the effective address calculation each time this primitive is issued during the execution of a given instruction. The calculation uses the current contents of any required address and data registers. The instruction must include a set of effective address extension words for each repetition of a calculation that requires them. The controller locates these

words at the current scanPC location and increments the scanPC by two for each word referenced in the instruction stream.

The MC68EC030 sign-extends a byte or word-sized operand to a long-word value when it is transferred to an address register (A0–A7) using this primitive with the register direct effective addressing mode. A byte or word-sized operand transferred to a data register (D0–D7) only overwrites the lower byte or word of the data register.

10.4.10 Write to Previously Evaluated Effective Address Primitive

The write to previously evaluated effective address primitive transfers an operand from the coprocessor to a previously evaluated effective address. This primitive applies to general category instructions. If the coprocessor uses this primitive during the execution of a conditional category instruction, the main controller initiates protocol violation exception processing. Figure 10-30 shows the format of the write to previously evaluated effective address primitive.

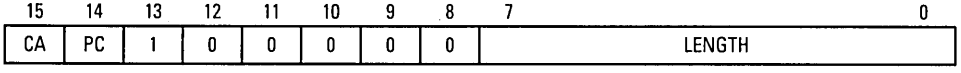


Figure 10-30. Write to Previously Evaluated Effective Address Primitive Format

This primitive uses the CA and PC bits as previously described.

Bits [0–7] of the primitive format specify the length of the operand in bytes. The MC68EC030 transfers operands between zero and 255 bytes in length.

When the main controller receives this primitive during the execution of a general category instruction, it transfers an operand from the operand CIR to an effective address specified by a temporary register within the MC68EC030. When a previous primitive for the current instruction has evaluated the effective address, this temporary register contains the evaluated effective address. Primitives that store an evaluated effective address in a temporary register of the main controller are the evaluate and transfer effective address, evaluate effective address and transfer data, and transfer multiple coprocessor registers primitive. If this primitive is used during an instruction in which the effective address specified in the instruction operation word has not been calculated, the effective address used for the write is undefined. Also, if the previously evaluated effective address was register direct, the address written to in response to this primitive is undefined.

The function code value during the write operation indicates either supervisor or user data space, depending on the value of the S bit in the MC68EC030 status register when the controller reads this primitive. While a coprocessor should request writes to only alterable effective addressing modes, the MC68EC030 does not check the type of effective address used with this primitive. For example, if the previously evaluated effective address was program counter relative and the MC68EC030 is at the user privilege level (S = 0 in status register), the MC68EC030 writes to user data space at the previously calculated program relative address (the 32-bit value in the temporary internal register of the controller).

Operands longer than four bytes are transferred in increments of four bytes (operand parts) when possible. The main controller reads a long-word operand part from the operand CIR and transfers this part to the current effective address. The transfers continue in this manner using ascending memory locations until all of the long-word operand parts are transferred, and any remaining operand part is then transferred using a one-, two-, or three-byte transfer as required. The operand parts are stored in memory using ascending addresses beginning with the address in the MC68EC030 temporary register.

The execution of this primitive does not modify any of the registers in the MC68EC030 programmer's model, even if the previously evaluated effective address mode is the predecrement or postincrement mode. If the previously evaluated effective addressing mode used any of the MC68EC030 internal address or data registers, the effective address value used is the final value from the preceding primitive. That is, this primitive uses the value from an evaluate and transfer effective address, evaluate effective address and transfer data, or transfer multiple coprocessor registers primitive without modification.

The take address and transfer data primitive described in the next section does not replace the effective address value that has been calculated by the MC68EC030. The address that the main controller obtains in response to the take address and transfer data primitive is not available to the write to previously evaluated effective address primitive.

A coprocessor can issue an evaluate effective address and transfer data primitive followed by this primitive to perform a read-modify-write operation that is not indivisible. The bus cycles for this operation are normal bus cycles that can be interrupted, and the bus can be arbitrated between the cycles.

10.4.11 Take Address and Transfer Data Primitive

The take address and transfer data primitive transfers an operand between the coprocessor and an address supplied by the coprocessor. This primitive applies to general and conditional category instructions. Figure 10-31 shows the format of the take address and transfer data primitive.

15	14	13	12	11	10	9	8	7	0
CA	PC	DR	0	0	1	0	1	LENGTH	

Figure 10-31. Take Address and Transfer Data Primitive Format

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA=0 during a conditional category instruction, the main controller initiates protocol violation exception processing.

Bits [0–7] of the primitive format specify the operand length, which can be from 0–255 bytes.

The main controller reads a 32-bit address from the operand address CIR. Using a series of long-word transfers, the controller transfers the operand between this address and the operand CIR. The DR bit determines the direction of the transfer. The controller reads or writes the operand parts to ascending addresses, starting at the address from the operand address CIR. If the operand length is not a multiple of four bytes, the final operand part is transferred using a one-, two-, or three-byte transfer as required.

The function code used with the address read from the operand address CIR indicates either supervisor or user data space according to the value of the S bit in the MC68EC030 status register.

10.4.12 Transfer to/from Top of Stack Primitive

The transfer to/from top of stack primitive transfers an operand between the coprocessor and the top of the currently active main controller stack (refer to **2.8.1 System Stack**). This primitive applies to general and conditional category instructions. Figure 10-32 shows the format of the transfer to/from top of stack primitive.

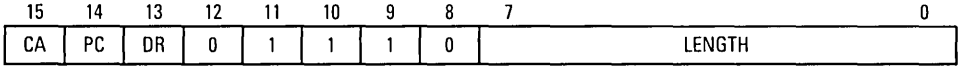


Figure 10-32. Transfer to/from Top of Stack Primitive Format

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA=0 during a conditional category instruction, the main controller initiates protocol violation exception processing.

Bits [0–7] of the primitive format specify the length in bytes of the operand to be transferred. The operand may be one, two, or four bytes in length; other length values cause the main controller to initiate protocol violation exception processing.

If DR=0, the main controller transfers the operand from the currently active system stack to the operand CIR. The implied effective address mode used for the transfer is the (A7)+ addressing mode. A one-byte operand causes the stack pointer to be incremented by two after the transfer to maintain word alignment of the stack.

If DR=1, the main controller transfers the operand from the operand CIR to the currently active stack. The implied effective address mode used for the transfer is the −(A7) addressing mode. A one-byte operand causes the stack pointer to be decremented by two before the transfer to maintain word alignment of the stack.

10.4.13 Transfer Single Main Controller Register Primitive

The transfer single main controller register primitive transfers an operand between one of the main controller's data or address registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 10-33 shows the format of the transfer single main controller register primitive.

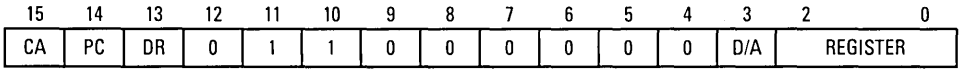


Figure 10-33. Transfer Single Main Controller Register Primitive Format

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA=0 during a conditional category instruction, the main controller initiates protocol violation exception processing.

Bit [3], the D/A bit, specifies whether the primitive transfers an address or data register. D/A=0 indicates a data register, and D/A=1 indicates an address register. Bits [2–0] contain the register number.

If DR=0, the main controller writes the long-word operand in the specified register to the operand CIR. If DR=1, the main controller reads a long-word operand from the operand CIR and transfers it to the specified data or address register.

10.4.14 Transfer Main Controller Control Register Primitive

The transfer main controller control register primitive transfers a long-word operand between one of its control registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 10-34 shows the format of the transfer main controller control register primitive. This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA=0 during a conditional category instruction, the main controller initiates protocol violation exception processing.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	0	1	0	0	0	0	0	0	0	0

Figure 10-34. Transfer Main Controller Control Register Primitive Format

When the main controller receives this primitive, it reads a control register select code from the register select CIR. This code determines which main controller control register is transferred. Table 10-5 lists the valid control register select codes. If the control register select code is not valid, the MC68EC030 initiates protocol violation exception processing (refer to **10.5.2.1 PROTOCOL VIOLATIONS**).

Table 10-5. Main Controller Control Register Selector Codes

Hex	Control Register
x000	Source Function Code (SFC) Register
x001	Destination Function Code (DFC) Register
x002	Cache Control Register (CACR)
x800	User Stack Pointer (USP)
x801	Vector Base Register (VBR)
x802	Cache Address Register (CAAR)
x803	Master Stack Pointer (MSP)
x804	Interrupt Stack Pointer (ISP)
All other codes cause a protocol violation exception	

After reading a valid code from the register select CIR, if DR = 0, the main controller writes the long-word operand from the specified control register to the operand CIR. If DR = 1, the main controller reads a long-word operand from the operand CIR and places it in the specified control register.

10.4.15 Transfer Multiple Main Controller Registers Primitive

The transfer multiple main controller registers primitive transfers long-word operands between one or more of its data or address registers and the coprocessor. This primitive applies to general and conditional category instructions. Figure 10-35 shows the format of the transfer multiple main controller registers primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	1	1	0	0	0	0	0	0	0	0	0

Figure 10-35. Transfer Multiple Main Controller Registers Primitive Format

This primitive uses the CA, PC, and DR bits as previously described. If the coprocessor issues this primitive with CA = 0 during a conditional category instruction, the main controller initiates protocol violation exception processing.

When the main controller receives this primitive, it reads a 16-bit register select mask from the register select CIR. The format of the register select mask is shown in Figure 10-36. A register is transferred if the bit correspond-

ing to the register in the register select mask is set to one. The selected registers are transferred in the order D0–D7 and then A0–A7.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A7	A6	A5	A4	A3	A2	A1	A0	D7	D6	D5	D4	D3	D2	D1	D0

Figure 10-36. Register Select Mask Format

If DR = 0, the main controller writes the contents of each register indicated in the register select mask to the operand CIR using a sequence of long-word transfers. If DR = 1, the main controller reads a long-word operand from the operand CIR into each register indicated in the register select mask. The registers are transferred in the same order, regardless of the direction of transfer indicated by the DR bit.

10.4.16 Transfer Multiple Coprocessor Registers Primitive

The transfer multiple coprocessor registers primitive transfers from 0–16 operands between the effective address specified in the coprocessor instruction and the coprocessor. This primitive applies to general category instructions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main controller initiates protocol violation exception processing. Figure 10-37 shows the format of the transfer multiple coprocessor registers primitive.

15	14	13	12	11	10	9	8	7	0
CA	PC	DR	0	0	0	0	1	LENGTH	

Figure 10-37. Transfer Multiple Coprocessor Registers Primitive Format

This primitive uses the CA, PC, and DR bits as previously described.

Bits [7–0] of the primitive format indicate the length in bytes of each operand transferred. The operand length must be an even number of bytes; odd length operands cause the MC68EC030 to initiate protocol violation exception processing (refer to **10.5.2.1 PROTOCOL VIOLATIONS**).

When the main controller reads this primitive, it calculates the effective address specified in the coprocessor instruction. The scanPC should be pointing to the first of any necessary effective address extension words when this primitive is read from the response CIR; the scanPC is incremented by two

for each extension word referenced during the effective address calculation. For transfers from the effective address to the coprocessor (DR = 0), the control addressing modes and the postincrement addressing mode are valid. For transfers from the coprocessor to the effective address (DR = 1), the control alterable and predecrement addressing modes are valid. Invalid addressing modes cause the MC68EC030 to abort the instruction by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR and to initiate F-line emulator exception processing (refer to **10.5.2.2 F-LINE EMULATOR EXCEPTIONS**).

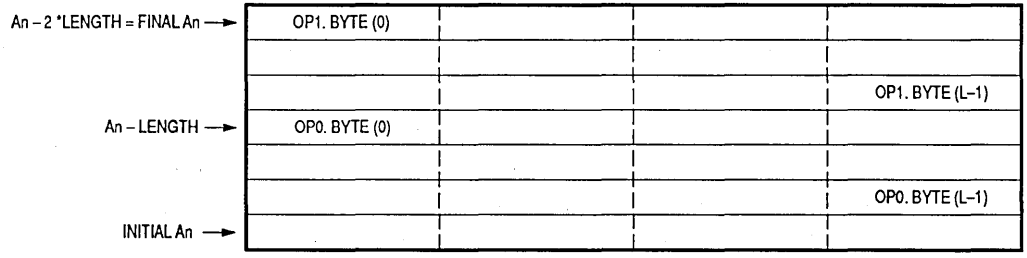
After performing the effective address calculation, the MC68EC030 reads a 16-bit register select mask from the register select CIR. The coprocessor uses the register select mask to specify the number of operands to transfer; the MC68EC030 counts the number of ones in the register select mask to determine the number of operands. The order of the ones in the register select mask is not relevant to the operation of the main controller. As many as 16 operands can be transferred by the main controller in response to this primitive. The total number of bytes transferred is the product of the number of operands transferred and the length of each operand specified in bits [0–7] of the primitive format.

If DR = 1, the main controller reads the number of operands specified in the register select mask from the operand CIR and writes these operands to the effective address specified in the instruction using long-word transfers whenever possible. If DR = 0, the main controller reads the number of operands specified in the register select mask from the effective address and writes them to the operand CIR.

For the control addressing modes, the operands are transferred to or from memory using ascending addresses. For the postincrement addressing mode, the operands are read from memory with ascending addresses also, and the address register used is incremented by the size of an operand after each operand is transferred. The address register used with the (An) + addressing mode is incremented by the total number of bytes transferred during the primitive execution.

For the predecrement addressing mode, the operands are written to memory with descending addresses, but the bytes within each operand are written to memory with ascending addresses. As an example, Figure 10-38 shows the format in long-word-oriented memory for two 12-byte operands transferred from the coprocessor to the effective address using the −(An) addressing mode. The controller decrements the address register by the size of an operand before the operand is transferred. It writes the bytes of the

operand to ascending memory addresses. When the transfer is complete, the address register has been decremented by the total number of bytes transferred. The MC68EC030 transfers the data using long-word transfers whenever possible.



NOTE: OP0. Byte (0) is the first byte written to memory.
OP0. Byte (L–1) is the last byte of the first operand written to memory.
OP1. Byte (0) is the first byte of the second operand written to memory.
OP1. Byte (L–1) is the last byte written to memory.

Figure 10-38. Operand Format in Memory for Transfer to –(An)

10.4.17 Transfer Status Register and ScanPC Primitive

Both the transfer status register and the scanPC primitive transfers values between the coprocessor and the main controller status register. On an optional basis, the scanPC also makes transfers. This primitive applies to general category instructions. If the coprocessor issues this primitive during the execution of a conditional category instruction, the main controller initiates protocol violation exception processing. Figure 10-39 shows the format of the transfer status register and scanPC primitive.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	0	1	SP	0	0	0	0	0	0	0	0

Figure 10-39. Transfer Status Register and ScanPC Primitive Format

This primitive uses the CA, PC, and DR bits as previously described.

Bit [8], the SP bit, selects the scanPC option. If SP = 1, the primitive transfers both the scanPC and status register. If SP = 0, only the status register is transferred.

If SP = 0 and DR = 0, the main controller writes the 16-bit status register value to the operand CIR. If SP = 0 and DR = 1, the main controller reads a 16-bit value from the operand CIR into the main controller status register.

If SP = 1 and DR = 0, the main controller writes the long-word value in the scanPC to the instruction address CIR and then writes the status register value to the operand CIR. If SP = 1 and DR = 1, the main controller reads a 16-bit value from the operand CIR into the status register and then reads a long-word value from the instruction address CIR into the scanPC.

With this primitive, a general category instruction can change the main controller program flow by placing a new value in the status register, in the scanPC, or new values in both the status register and the scanPC. By accessing the status register, the coprocessor can determine and manipulate the main controller condition codes, supervisor status, trace modes, selection of the active stack, and interrupt mask level.

The MC68EC030 discards any instruction words that have been prefetched beyond the current scanPC location when this primitive is issued with DR = 1 (transfer to main controller). The MC68EC030 then refills the instruction pipe from the scanPC address in the address space indicated by the status register S bit.

If the MC68EC030 is operating in the trace on change of flow mode (T1:T0 in the status register contains 01) when the coprocessor instruction begins to execute and if this primitive is issued with DR = 1 (from coprocessor to main controller), the MC68EC030 prepares to take a trace exception. The trace exception occurs when the coprocessor signals that it has completed all processing associated with the instruction. Changes in the trace modes due to the transfer of the status register to main controller take effect on execution of the next instruction.

10.4.18 Take Pre-Instruction Exception Primitive

The take pre-instruction exception primitive initiates exception processing using a coprocessor-supplied exception vector number and the pre-instruction exception stack frame format. This primitive applies to general and conditional category instructions. Figure 10-40 shows the format of the take pre-instruction exception primitive.

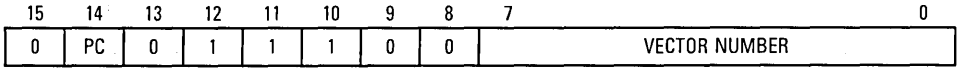


Figure 10-40. Take Pre-Instruction Exception Primitive Format

The primitive uses the PC bit as previously described. Bits [0–7] contain the exception vector number used by the main controller to initiate exception processing.

When the main controller receives this primitive, it acknowledges the coprocessor exception request by writing an exception acknowledge mask (refer to **10.3.2 Control CIR**) to the control CIR. The MC68EC030 then proceeds with exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**. The vector number for the exception is taken from bits [0–7] of the primitive, and the MC68EC030 uses the four-word stack frame format shown in Figure 10-41.

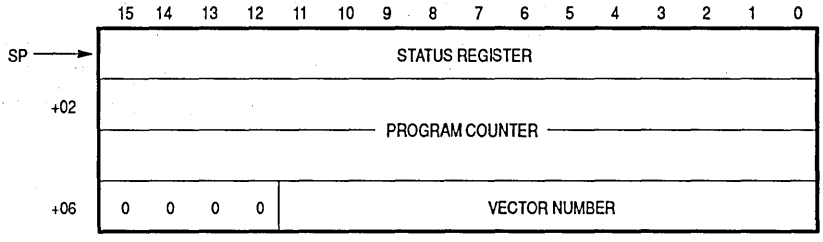


Figure 10-41. MC68EC030 Pre-Instruction Stack Frame

The value of the program counter saved in this stack frame is the F-line operation word address of the coprocessor instruction during which the primitive was received. Thus, if the exception handler routine does not modify the stack frame, an RTE instruction causes the MC68EC030 to return and reinitiate execution of the coprocessor instruction.

The take pre-instruction exception primitive can be used when the coprocessor does not recognize a value written to either its command CIR or condition CIR to initiate a coprocessor instruction. This primitive can also be used if an exception occurs in the coprocessor instruction before any program-visible resources are modified by the instruction operation. This primitive should not be used during a coprocessor instruction if program-visible resources have been modified by that instruction. Otherwise, since the MC68EC030 reinitiates the instruction when it returns from exception processing, the restarted instruction receives the previously modified resources in an inconsistent state.

One of the most important uses of the take pre-instruction exception primitive is to signal an exception condition in a cpGEN instruction that was executing concurrently with the main controller's instruction execution. If the coprocessor no longer requires the services of the main controller to complete a cpGEN instruction and the concurrent instruction completion is transparent to the programmer's model, the coprocessor can release the main controller by issuing a primitive with CA = 0. The main controller usually executes the next instruction in the instruction stream, and the coprocessor completes its operations concurrently with the main controller operation. If an exception occurs while the coprocessor is executing an instruction concurrently, the exception is not processed until the main controller attempts to initiate the next general or conditional instruction. After the main controller writes to the command or condition CIR to initiate a general or conditional instruction, it then reads the response CIR. At this time, the coprocessor can return the take pre-instruction exception primitive. This protocol allows the main controller to proceed with exception processing related to the previous concurrently executing coprocessor instruction and then return and reinitiate the coprocessor instruction during which the exception was signaled. The coprocessor should record the addresses of all general category instructions that can be executed concurrently with the main controller and that support exception recovery. Since the exception is not reported until the next coprocessor instruction is initiated, the controller usually requires the instruction address to determine which instruction the coprocessor was executing when the exception occurred. A coprocessor can record the instruction address by setting PC = 1 in one of the primitives it uses before releasing the main controller.

10.4.19 Take Mid-Instruction Exception Primitive

The take mid-instruction exception primitive initiates exception processing using a coprocessor-supplied exception vector number and the mid-instruction exception stack frame format. This primitive applies to general and conditional category instructions. Figure 10-42 shows the format of the take mid-instruction exception primitive.

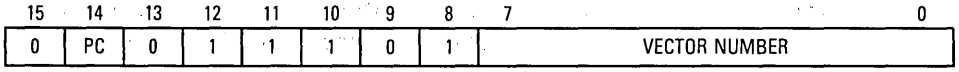


Figure 10-42. Take Mid-Instruction Exception Primitive Format

This primitive uses the PC bit as previously described. Bits [7–0] contain the exception vector number used by the main controller to initiate exception processing.

When the main controller receives this primitive, it acknowledges the coprocessor exception request by writing an exception acknowledge mask (refer to **10.3.2 Control CIR**) to the control CIR. The MC68EC030 then performs exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**. The vector number for the exception is taken from bits [0–7] of the primitive and the MC68EC030 uses the 10-word stack frame format shown in Figure 10-43.

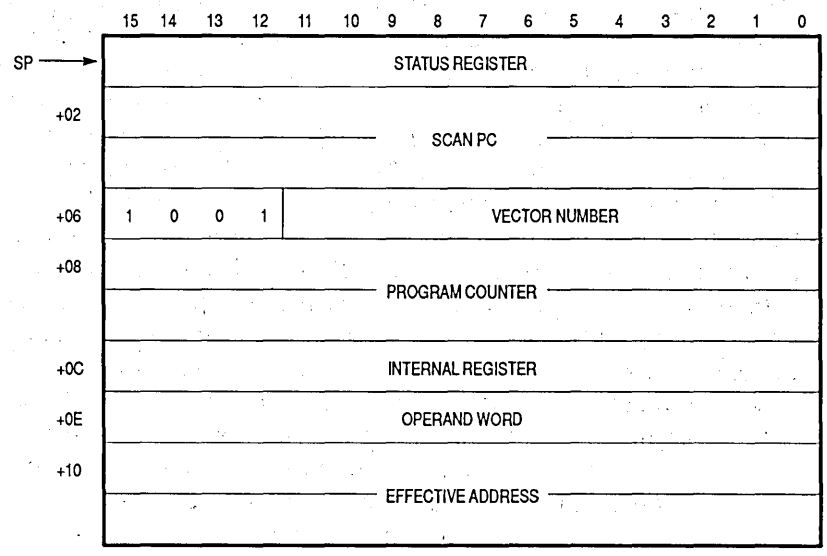


Figure 10-43. MC68EC030 Mid-Instruction Stack Frame

The program counter value saved in this stack frame is the operation word address of the coprocessor instruction during which the primitive is received. The scanPC field contains the value of the MC68EC030 scanPC when the primitive is received. If the current instruction does not evaluate an effective address prior to the exception request primitive, the value of the effective address field in the stack frame is undefined.

The coprocessor uses this primitive to request exception processing for an exception during the instruction dialog with the main controller. If the exception handler does not modify the stack frame, the MC68EC030 returns from the exception handler and reads the response CIR. Thus, the main controller attempts to continue executing the suspended instruction by reading the response CIR and processing the primitive it receives.

10.4.20 Take Post-Instruction Exception Primitive

The take post-instruction exception primitive initiates exception processing using a coprocessor-supplied exception vector number and the post-instruction exception stack frame format. This primitive applies to general and conditional category instructions. Figure 10-44 shows the format of the take post-instruction exception primitive.

15	14	13	12	11	10	9	8	7	0
0	PC	0	1	1	1	1	0	VECTOR NUMBER	

Figure 10-44. Take Post-Instruction Exception Primitive Format

This primitive uses the PC bit as previously described. Bits [0–7] contain the exception vector number used by the main controller to initiate exception processing.

When the main controller receives this primitive, it acknowledges the coprocessor exception request by writing an exception acknowledge mask (refer to **10.3.2 Control CIR**) to the control CIR. The MC68EC030 then performs exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE**. The vector number for the exception is taken from bits [0-7] of the primitive, and the MC68EC030 uses the six-word stack frame format shown in Figure 10-45.

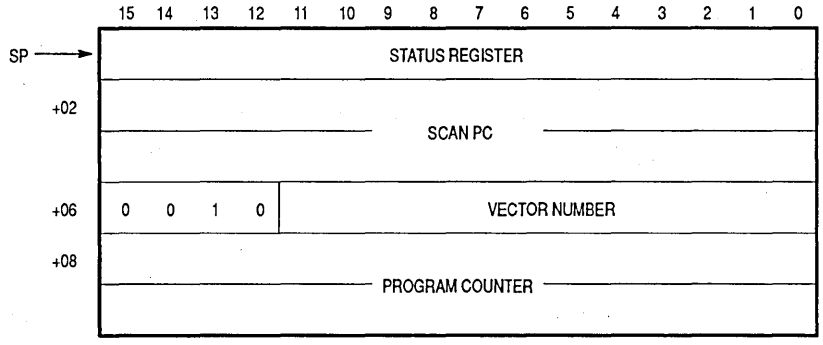


Figure 10-45. MC68EC030 Post-Instruction Stack Frame

The value in the main controller scanPC at the time this primitive is received is saved in the scanPC field of the post-instruction exception stack frame. The value of the program counter saved is the F-line operation word address of the coprocessor instruction during which the primitive is received.

When the MC68EC030 receives the take post-instruction exception primitive, it assumes that the coprocessor either completed or aborted the instruction with an exception. If the exception handler does not modify the stack frame, the MC68EC030 returns from the exception handler to begin execution at the location specified by the scanPC field of the stack frame. This location should be the address of the next instruction to be executed.

The coprocessor uses this primitive to request exception processing when it completes or aborts an instruction while the main controller is awaiting a normal response. For a general category instruction, the response is a release; for a conditional category instruction, it is an evaluated true/false condition indicator. Thus, the operation of the MC68EC030 in response to this primitive is compatible with standard M68000 Family instruction related exception processing (for example, the divide-by-zero exception).

10.5 EXCEPTIONS

Various exception conditions related to the execution of coprocessor instructions may occur. Whether an exception is detected by the main controller or by the coprocessor, the main controller coordinates and performs exception processing. Servicing these coprocessor-related exceptions is an extension of the protocol used to service standard M68000 Family exceptions. That is, when either the main controller detects an exception or is signaled by the

coprocessor that an exception condition has occurred, the main controller proceeds with exception processing as described in **8.1 EXCEPTION PROCESSING SEQUENCE.**

10.5.1 Coprocessor-Detected Exceptions

Exceptions that the coprocessor detects, also those that the main controller detects, are usually classified as coprocessor-detected exceptions. These exceptions can occur during M68000 coprocessor interface operations, internal operations, or other system-related operations of the coprocessor.

Most coprocessor-detected exceptions are signaled to the main controller through the use of one of the three take exception primitives defined for the M68000 coprocessor interface. The main controller responds to these primitives as previously described. However, not all coprocessor-detected exceptions are signaled by response primitives. Coprocessor-detected format errors during the cpSAVE or cpRESTORE instruction are signaled to the main controller using the invalid format word described in **10.2.3.4.3 Invalid Format Words.**

10.5.1.1 COPROCESSOR-DETECTED PROTOCOL VIOLATIONS. Protocol violation exceptions are communication failures between the main controller and coprocessor across the M68000 coprocessor interface. Coprocessor-detected protocol violations occur when the main controller accesses entries in the coprocessor interface register set in an unexpected sequence. The sequence of operations that the main controller performs for a given coprocessor instruction or coprocessor response primitive has been described previously in this section.

A coprocessor can detect protocol violations in various ways. According to the M68000 coprocessor interface protocol, the main controller always accesses the operation word, operand, register select, instruction address, or operand address CIRs synchronously with respect to the operation of the coprocessor. That is, the main controller accesses these five registers in a certain sequence, and the coprocessor expects them to be accessed in that sequence. As a minimum, all M68000 coprocessors should detect a protocol violation if the main controller accesses any of these five registers when the coprocessor is expecting an access to either the command or condition CIR. Likewise, if the coprocessor is expecting an access to the command or condition CIR and the main controller accesses one of these five registers, the coprocessor should detect and signal a protocol violation.

According to the M68000 coprocessor interface protocol, the main controller can perform a read of either the save or response CIRs or a write of either the restore or control CIRs asynchronously with respect to the operation of the coprocessor. That is, an access to one of these registers without the coprocessor explicitly expecting that access at that point can be a valid access. Although the coprocessor can anticipate certain accesses to the restore, response, and control coprocessor interface registers, these registers can be accessed at other times also.

The coprocessor cannot signal a protocol violation to the main controller during the execution of cpSAVE or cpRESTORE instructions. If a coprocessor detects a protocol violation during the cpSAVE or cpRESTORE instruction, it should signal the exception to the main controller when the next coprocessor instruction is initiated.

The main philosophy of the coprocessor-detected protocol violation is that the coprocessor should always acknowledge an access to one of its interface registers. If the coprocessor determines that the access is not valid, it should assert $\overline{\text{DSACKx}}$, to the main controller and signal a protocol violation when the main controller next reads the response CIR. If the coprocessor fails to assert $\overline{\text{DSACKx}}$, the main controller waits for the assertion of that signal (or some other bus termination signal) indefinitely. The protocol previously described ensures that the coprocessor cannot halt the main controller.

The coprocessor can signal a protocol violation to the main controller with the take mid-instruction exception primitive. To maintain consistency, the vector number should be 13, as it is for a protocol violation detected by the main controller. When the main controller reads this primitive, it proceeds as described in **10.4.19 Take Mid-Instruction Exception Primitive**. If the exception handler does not modify the stack frame, the MC68EC030 returns from the exception handler and reads the response CIR.

10.5.1.2 COPROCESSOR-DETECTED ILLEGAL COMMAND OR CONDITION WORDS. Illegal coprocessor command or condition words are values written to the command CIR or condition CIR that the coprocessor does not recognize. If a value written to either of these registers is not valid, the coprocessor should return the take pre-instruction exception primitive in the response CIR. When it receives this primitive, the main controller takes a pre-instruction exception as described in **10.4.18 Take Pre-Instruction Exception Primitive**. If the exception handler does not modify the main controller stack frame, an RTE instruction causes the MC68EC030 to reinitiate the instruction that took the exception. The coprocessor designer should ensure that the

state of the coprocessor is not irrecoverably altered by an illegal command or condition exception if the system supports emulation of the unrecognized command or condition word.

All Motorola M68000 coprocessors signal illegal command and condition words by returning the take pre-instruction exception primitive with the F-line emulator exception vector number 11.

10.5.1.3 COPROCESSOR DATA-PROCESSING EXCEPTIONS. Exceptions related to the internal operation of a coprocessor are classified as data-processing-related exceptions. These exceptions are analogous to the divide-by-zero exception defined by M68000 microprocessors and should be signaled to the main controller using one of the three take exception primitives containing an appropriate exception vector number. Which of these three primitives is used to signal the exception is usually determined by the point in the instruction operation where the main controller should continue the program flow after exception processing. Refer to **10.4.18 Take Pre-Instruction Exception Primitives, 10.4.19 Take Mid-Instruction Exception Primitive,** and **10.4.20 Take Post-Instruction Exception Primitive.**

10.5.1.4 COPROCESSOR SYSTEM-RELATED EXCEPTIONS. System-related exceptions detected by a DMA coprocessor include those associated with bus activity and any other exceptions (interrupts, for example) occurring external to the coprocessor. The actions taken by the coprocessor and the main controller depend on the type of exception that occurs.

When an address or bus error is detected by a DMA coprocessor, the coprocessor should store any information necessary for the main controller exception handling routines in system-accessible registers. The coprocessor should place one of the three take exception primitives encoded with an appropriate exception vector number in the response CIR. Which of the three primitives is used depends upon the point in the coprocessor instruction at which the exception was detected and the point in the instruction execution at which the main controller should continue after exception processing.

10.5.1.5 FORMAT ERRORS. Format errors are the only coprocessor-detected exceptions that are not signaled to the main controller with a response primitive. When the main controller writes a format word to the restore CIR during the execution of a cpRESTORE instruction, the coprocessor decodes this word

to determine if it is valid (refer to **10.2.3.3 COPROCESSOR CONTEXT SAVE INSTRUCTION**). If the format word is not valid, the coprocessor places the invalid format code in the restore CIR. When the main controller reads the invalid format code, it aborts the coprocessor instruction by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR. The main controller then performs exception processing using a four-word pre-instruction stack frame and the format error exception vector number 14. Thus, if the exception handler does not modify the stack frame, the MC68EC030 restarts the cpRESTORE instruction when the RTE instruction in the handler is executed. If the coprocessor returns the invalid format code when the main controller reads the save CIR to initiate a cpSAVE instruction, the main controller performs format error exception processing as outlined for the cpRESTORE instruction.

10.5.2 Main-Controller-Detected Exceptions

A number of exceptions related to coprocessor instruction execution are detected by the main controller instead of the coprocessor (they are still serviced by the main controller). These exceptions can be related to the execution of coprocessor response primitives, communication across the M68000 coprocessor interface, or the completion of conditional coprocessor instructions by the main controller.

10.5.2.1 PROTOCOL VIOLATIONS. The main controller detects a protocol violation when it reads a primitive from the response CIR that is not a valid primitive. The protocol violations that can occur in response to the primitives defined for the M68000 coprocessor interface are summarized in Table 10-6.

Table 10-6. Exceptions Related to Primitive Processing

Primitive	Protocol	F-Line	Other
Busy			
NULL			
Supervisory Check* Other: Privilege Violation if "S" Bit=0			X
Transfer Operation Word*			
Transfer from Instruction Stream* Protocol: If Length Field is Odd (Zero Length Legal)	X		

Table 10-6. Exceptions Related to Primitive Processing (Continued)

Evaluate and Transfer Effective Address Protocol: If Used with Conditional Instruction F-Line: If EA in Op-Word is NOT Control Alterable	X	X	
Evaluate Effective Address and Transfer Data Protocol: 1. If Used with Conditional Instructions 2. Length is Not 1, 2, or 4 and EA = Register Direct 3. If EA = Immediate and Length Odd and Greater Than 1 4. Attempt to Write to Nonalterable Address Even if Address Declared Legal in Primitive F-Line: Valid EA Field Does Not Match EA in Op-Word	X	X	
Write to Previously Evaluated Effective Address Protocol: If Used with Conditional Instruction	X		
Busy			
Take Address and Transfer Data*			
Transfer To/From Top of Stack* Protocol: Length Field Other Than 1, 2, or 4	X		
Transfer To/From Main Controller Register*			
Transfer To/From Main Controller Control Register Protocol: Invalid Control Register Select Code	X		
Transfer Multiple Main Controller Registers*			
Transfer Multiple Coprocessor Registers Protocol: 1. If Used with Conditional Instructions 2. Odd Length Value F-Line: 1. EA Not Control Alterable or (An)+ for CP to Memory Transfer 2. EA Not Control Alterable or −(An) for Memory to CP Transfer	X X		
Transfer Status and/or ScanPC Protocol: If Used with Conditional Instruction Other: 1. Trace — Trace Made Pending if MC68EC030 in "Trace on Change of Flow" Mode and DR = 1 2. Address Error — If Odd value Written to ScanPC	X		X
Take Pre-Instruction, Mid-Instruction, or Post-Instruction Exception Exception Depends on Vector Supplies in Primitive	X	X	X

*Use of this primitive with CA = 0 will cause protocol violation on conditional instructions.

Abbreviations:
EA = Effective Address
CP = Coprocessor

When the MC68EC030 detects a protocol violation, it does not automatically notify the coprocessor of the resulting exception by writing to the control CIR. The exception handling routine may, however, use the MOVES instruction to read the response CIR and thus determine the primitive that caused the MC68EC030 to initiate protocol violation exception processing. The main

controller initiates exception processing using the mid-instruction stack frame (refer to Figure 10-43) and the coprocessor protocol violation exception vector number 13. If the exception handler does not modify the stack frame, the main controller reads the response CIR again following the execution of an RTE instruction to return from the exception handler. This protocol allows extensions to the M68000 coprocessor interface to be emulated in software by a main controller that does not provide hardware support for these extensions. Thus, the protocol violation is transparent to the coprocessor if the primitive execution can be emulated in software by the main controller.

10.5.2.2 F-LINE EMULATOR EXCEPTIONS. The F-line emulator exceptions detected by the MC68EC030 are either explicitly or implicitly related to the encodings of F-line operation words in the instruction stream. If the main controller determines that an F-line operation word is not valid, it initiates F-line emulator exception processing. Any F-line operation word with bits [8:6] = 110 or 111 causes the MC68EC030 to initiate exception processing without initiating any communication with the coprocessor for that instruction. Also, an operation word with bits [8:6] = 000–101 that does not map to one of the valid coprocessor instructions in the instruction set causes the MC68EC030 to initiate F-line emulator exception processing, except those F-line instructions implemented on the MC68030 but not on the MC68EC030. See **APPENDIX A MC68EC030 NEW INSTRUCTIONS**. If the F-line emulator exception is either of these two situations, the main controller does not write to the control CIR prior to initiating exception processing.

F-line exceptions can also occur if the operations requested by a coprocessor response primitive are not compatible with the effective address type in bits [0–5] of the coprocessor instruction operation word. The F-line emulator exceptions that can result from the use of the M68000 coprocessor response primitives are summarized in Table 10-6. If the exception is caused by receiving an invalid primitive, the main controller aborts the coprocessor instruction in progress by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR prior to F-line emulator exception processing.

Another type of F-line emulator exception occurs when a bus error occurs during the coprocessor interface register access that initiates a coprocessor instruction. The main controller assumes that the coprocessor is not present and takes the exception.

When the main controller initiates F-line emulator exception processing, it uses the four-word pre-instruction exception stack frame (refer to Figure 10-41) and the F-line emulator exception vector number 11. Thus, if the exception handler does not modify the stack frame, the main controller attempts to restart the instruction that caused the exception after it executes an RTE instruction to return from the exception handler.

If the cause of the F-line exception can be emulated in software, the handler stores the results of the emulation in the appropriate registers of the programmer's model and in the status register field of the saved stack frame. The exception handler adjusts the program counter field of the saved stack frame to point to the next instruction operation word and executes the RTE instruction. The MC68EC030 then executes the instruction following the instruction that was emulated.

The exception handler should also check the copy of the status register on the stack to determine whether tracing is on. If tracing is on, the trace exception processing should also be emulated. Refer to **8.1.7 Trace Exception** for additional information.

10.5.2.3 PRIVILEGE VIOLATIONS. Privilege violations can result from the cpSAVE and cpRESTORE instructions and, also, from the supervisor check coprocessor response primitive. The main controller initiates privilege violation exception processing if it attempts to execute either the cpSAVE or cpRESTORE instruction when it is in the user state (S = 0 in status register). The main controller initiates this exception processing prior to any communication with the coprocessor associated with the cpSAVE or cpRESTORE instructions.

If the main controller is executing a coprocessor instruction in the user state when it reads the supervisor check primitive, it aborts the coprocessor instruction in progress by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR. The main controller then performs privilege violation exception processing.

If a privilege violation occurs, the main controller initiates exception processing using the four-word pre-instruction stack frame (refer to Figure 10-41) and the privilege violation exception vector number 8. Thus, if the exception handler does not modify the stack frame, the main controller attempts to restart the instruction during which the exception occurred after it executes an RTE to return from the handler.

10.5.2.4 cpTRAPcc INSTRUCTION TRAPS. If, during the execution of a cpTRAPcc instruction, the coprocessor returns the TRUE condition indicator to the main controller with a null primitive, the main controller initiates trap exception processing. The main controller uses the six-word post-instruction exception stack frame (refer to Figure 10-45) and the trap exception vector number 7. The scanPC field of this stack frame contains the address of the instruction following the cpTRAPcc instruction. The processing associated with the cpTRAPcc instruction can then proceed, and the exception handler can locate any immediate operand words encoded in the cpTRAPcc instruction using the information contained in the six-word stack frame. If the exception handler does not modify the stack frame, the main controller executes the instruction following the cpTRAPcc instruction after it executes an RTE instruction to exit from the handler.

10.5.2.5 TRACE EXCEPTIONS. The MC68EC030 supports two modes of instruction tracing, discussed in **8.1.7 Trace Exception**. In the trace on instruction execution mode, the MC68EC030 takes a trace exception after completing each instruction. In the trace on change of flow mode, the MC68EC030 takes a trace exception after each instruction that alters the status register or places an address other than the address of the next instruction in program counter.

The protocol used to execute coprocessor cpSAVE, cpRESTORE, or conditional category instructions does not change when a trace exception is pending in the main controller. The main controller performs a pending trace on instruction execution exception after completing the execution of that instruction. If the main controller is in the trace on change of flow mode and an instruction places an address other than that of the next instruction in the program counter, the controller takes a trace exception after it executes the instruction.

If a trace exception is not pending during a general category instruction, the main controller terminates communication with the coprocessor after reading any primitive with CA=0. Thus, the coprocessor can complete a cpGEN instruction concurrently with the execution of instructions by the main controller. When a trace exception is pending, however, the main controller must ensure that all processing associated with a cpGEN instruction has been completed before it takes the trace exception. In this case, the main controller continues to read the response CIR and to service the primitives until it receives either a null, CA=0, PF=1 primitive, or until exception processing caused by a take post-instruction exception primitive has completed. The coprocessor should return the null, CA=0 primitive with PF=0, while it is completing the execution of the cpGEN instruction. The main controller may

service pending interrupts between reads of the response CIR if IA=1 in these primitives (refer to Table 10-3). This protocol ensures that a trace exception is not taken until all processing associated with a cpGEN instruction has completed.

If T1:T0=01 in the MC68EC030 status register (trace on change of flow) when a general category instruction is initiated, a trace exception is taken for the instruction only when the coprocessor issues a transfer status register and scanPC primitive with DR=1 during the execution of that instruction. In this case, it is possible that the coprocessor is still executing the cpGEN instruction concurrently when the main controller begins execution of the trace exception handler. A cpSAVE instruction executed during the trace on change of flow exception handler could thus suspend the execution of a concurrently operating cpGEN instruction.

10.5.2.6 INTERRUPTS. Interrupt processing, discussed in **8.1.9 Interrupt Exceptions**, can occur at any instruction boundary. Interrupts are also serviced during the execution of a general or conditional category instruction under either of two conditions. If the main controller reads a null primitive with CA=1 and IA=1, it services any pending interrupts prior to reading the response CIR. Similarly, if a trace exception is pending during cpGEN instruction execution and the main controller reads a null primitive with CA=0, IA=1, and PF=0 (refer to **10.5.2.5 TRACE EXCEPTIONS**), the main controller services pending interrupts prior to reading the response CIR again.

The MC68EC030 uses the ten-word mid-instruction stack frame when it services interrupts during the execution of a general or conditional category coprocessor instruction. Since it uses this stack frame, the main controller can perform all necessary processing and then return to read the response CIR. Thus, it can continue the coprocessor instruction during which the interrupt exception was taken.

The MC68EC030 also services interrupts if it reads the not ready format word from the save CIR during a cpSAVE instruction. The MC68EC030 uses the normal four word pre-instruction stack frame when it services interrupts after reading the not ready format word. Thus, the controller can service any pending interrupts and execute an RTE to return and re-initiate the cpSAVE instruction by reading the save CIR.

10.5.2.7 FORMAT ERRORS. The MC68EC030 can detect a format error while executing a cpSAVE or cpRESTORE instruction if the length field of a valid format word is not a multiple of four bytes in length. If the MC68EC030 reads a format word with an invalid length field from the save CIR during the cpSAVE instruction, it aborts the coprocessor instruction by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR and initiates format error exception processing. If the MC68EC030 reads a format word with an invalid length field from the effective address specified in the cpRESTORE instruction, the MC68EC030 writes that format word to the restore CIR and then reads the coprocessor response from the restore CIR. The MC68EC030 then aborts the cpRESTORE instruction by writing an abort mask (refer to **10.3.2 Control CIR**) to the control CIR and initiates format error exception processing.

The MC68EC030 uses the four-word pre-instruction stack frame and the format error vector number 14 when it initiates format error exception processing. Thus, if the exception handler does not modify the stack frame, the main controller attempts to restart the instruction during which the exception occurred after it executes an RTE to return from the handler.

10.5.2.8 ADDRESS AND BUS ERRORS. Coprocessor-instruction-related bus faults can occur during main controller bus cycles to CPU space to communicate with a coprocessor or during memory cycles run as part of the coprocessor instruction execution. If a bus error occurs during the coprocessor interface register access that is used to initiate a coprocessor instruction, the main controller assumes that the coprocessor is not present and takes an F-line emulator exception as described in **10.5.2.2 F-LINE EMULATOR EXCEPTIONS.** That is, the controller takes an F-line emulator exception when a bus error occurs during the initial access to a CIR by a coprocessor instruction. If a bus error occurs on any other coprocessor access or on a memory access made during the execution of a coprocessor instruction, the main controller performs bus error exception processing as described in **8.1.2 Bus Error Exceptions**. After the exception handler has corrected the cause of the bus error, the main controller can return to the point in the coprocessor instruction at which the fault occurred.

An address error occurs if the MC68EC030 attempts to prefetch an instruction from an odd address. This can occur if the calculated destination address of a cpBcc or cpDBcc instruction is odd or if an odd value is transferred to the scanPC with the transfer status register and the scanPC response primitive. If an address error occurs, the MC68EC030 performs exception processing for a bus fault as described in **8.1.3 Address Error Exception**.

10.5.3 Coprocessor Reset

Either an external reset signal or a RESET instruction can reset the external devices of a system. The system designer can design a coprocessor to be reset and initialized by both reset types or by external reset signals only. To be consistent with the MC68EC030 design, the coprocessor should be affected by external reset signals only and not by RESET instructions, because the coprocessor is an extension to the main controller programming model and to the internal state of the MC68EC030.

10.6 COPROCESSOR SUMMARY

Coprocessor instruction formats are presented for reference. Refer to the M68000PM/AD, *M68000 Programmer's Reference Manual*, for detailed information on coprocessor instructions.

The M68000 coprocessor response primitive formats are shown in this section. Any response primitive with bits [13:8] = $00 or $3F causes a protocol violation exception. Response primitives with bits [13:8] = $0B, $18–$1B, $1F, $28–$2B, and $38–3B currently cause protocol violation exceptions; they are undefined and reserved for future use by Motorola.

BUSY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	1	0	0	1	0	0	0	0	0	0	0	0	0	0

TRANSFER MULTIPLE COPROCESSOR REGISTERS

15	14	13	12	11	10	9	8	7	0
CA	PC	DR	0	0	0	0	1	LENGTH	

TRANSFER STATUS REGISTER AND SCANPC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	0	1	SP	0	0	0	0	0	0	0	0

SUPERVISOR CHECK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	PC	0	0	0	1	0	0	0	0	0	0	0	0	0	0

TAKE ADDRESS AND TRANSFER DATA

15	14	13	12	11	10	9	8	7	0
CA	PC	DR	0	0	1	0	1	LENGTH	

TRANSFER MULTIPLE MAIN CONTROLLER REGISTERS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	0	1	1	0	0	0	0	0	0	0	0	0

TRANSFER OPERATION WORD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	0	1	1	1	0	0	0	0	0	0	0	0

NULL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	0	IA	0	0	0	0	0	0	PF	TF

EVALUATE AND TRANSFER EFFECTIVE ADDRESS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	1	0	0	0	0	0	0	0	0	0

TRANSFER SINGLE MAIN CONTROLLER REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
CA	PC	DR	0	1	1	0	0	0	0	0	0	D/A	REGISTER	

TRANSFER MAIN CONTROLLER CONTROL REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	0	1	1	0	1	0	0	0	0	0	0	0	0

TRANSFER TO/FROM TOP OF STACK

15	14	13	12	11	10	9	8	7	0
CA	PC	DR	0	1	1	1	0	LENGTH	

TRANSFER FROM INSTRUCTION STREAM

15	14	13	12	11	10	9	8	7	0
CA	PC	0	0	1	1	1	1	LENGTH	

EVALUATE EFFECTIVE ADDRESS AND TRANSFER DATA

15	14	13	12	11	10	9	8	7	0
CA	PC	DR	1	0	VALID EA			LENGTH	

TAKE PRE-INSTRUCTION EXCEPTION

15	14	13	12	11	10	9	8	7	0
0	PC	0	1	1	1	0	0	VECTOR NUMBER	

TAKE MID-INSTRUCTION EXCEPTION

15	14	13	12	11	10	9	8	7	0
0	PC	0	1	1	1	0	1	VECTOR NUMBER	

TAKE POST-INSTRUCTION EXCEPTION

15	14	13	12	11	10	9	8	7	0
0	PC	0	1	1	1	1	0	VECTOR NUMBER	

WRITE TO PREVIOUSLY EVALUATED EFFECTIVE ADDRESS

15	14	13	12	11	10	9	8	7	0
CA	PC	1	0	0	0	0	0	LENGTH	

SECTION 11
INSTRUCTION EXECUTION TIMING

This section describes the instruction execution and operations (table searches, etc.) of the MC68EC030 in terms of external clock cycles. It provides accurate execution and operation timing guidelines but not exact timings for every possible circumstance. This approach is used since exact execution time for an instruction or operation is highly dependent on memory speeds and other variables. The timing numbers presented in this section allow the assembly language programmer or compiler writer to predict actual cache-case and average no-cache-case timings needed to evaluate the performance of the MC68EC030. Additionally, the timings for exception processing, context switching, and interrupt processing are included so that designers of multi-tasking or real-time systems can predict task switch overhead, maximum interrupt latency, and similar timing parameters.

In this section, instruction and operation times are shown in clock cycles to eliminate clock frequency dependencies.

11.1 PERFORMANCE TRADEOFFS

The MC68EC030 maximizes average performance at the expense of worst case performance. The time spent executing one instruction can vary from zero to over 100 clocks. Factors affecting the execution time are the preceding and following instructions, the instruction stream alignment, residency of operands and instruction words in the caches, and operand alignment.

To increase the average performance of the MC68EC030, certain tradeoffs were made to increase best case performance and to decrease the occurrence of worst case behavior. For example, burst filling increases performance by prefetching data for later accesses, but it commits the external bus controller and a cache for a longer period.

The MC68EC030 can overlap data writes with instruction cache reads, data cache reads, and/or microsequencer execution. Instruction cache reads can be overlapped with data cache fills and/or microsequencer activity. Similarly,

data cache reads can be overlapped with instruction cache fills and/or microsequencer activity. The execution of an instruction that only accesses on-chip registers can be overlapped entirely with a concurrent data write generated by a previous instruction, if prefetches generated by that instruction are resident in the instruction cache.

11.2 RESOURCE SCHEDULING

Some of the variability in instruction execution timings results from the overlap of resource utilization. The controller can be viewed as consisting of eight independently scheduled resources. Since very little of the scheduling is directly related to instruction boundaries, it is impossible to make accurate estimates of the time required to execute a particular instruction without knowing the complete context within which the instruction is executing. The position of these resources within the MC68EC030 is shown in Figure 11-1.

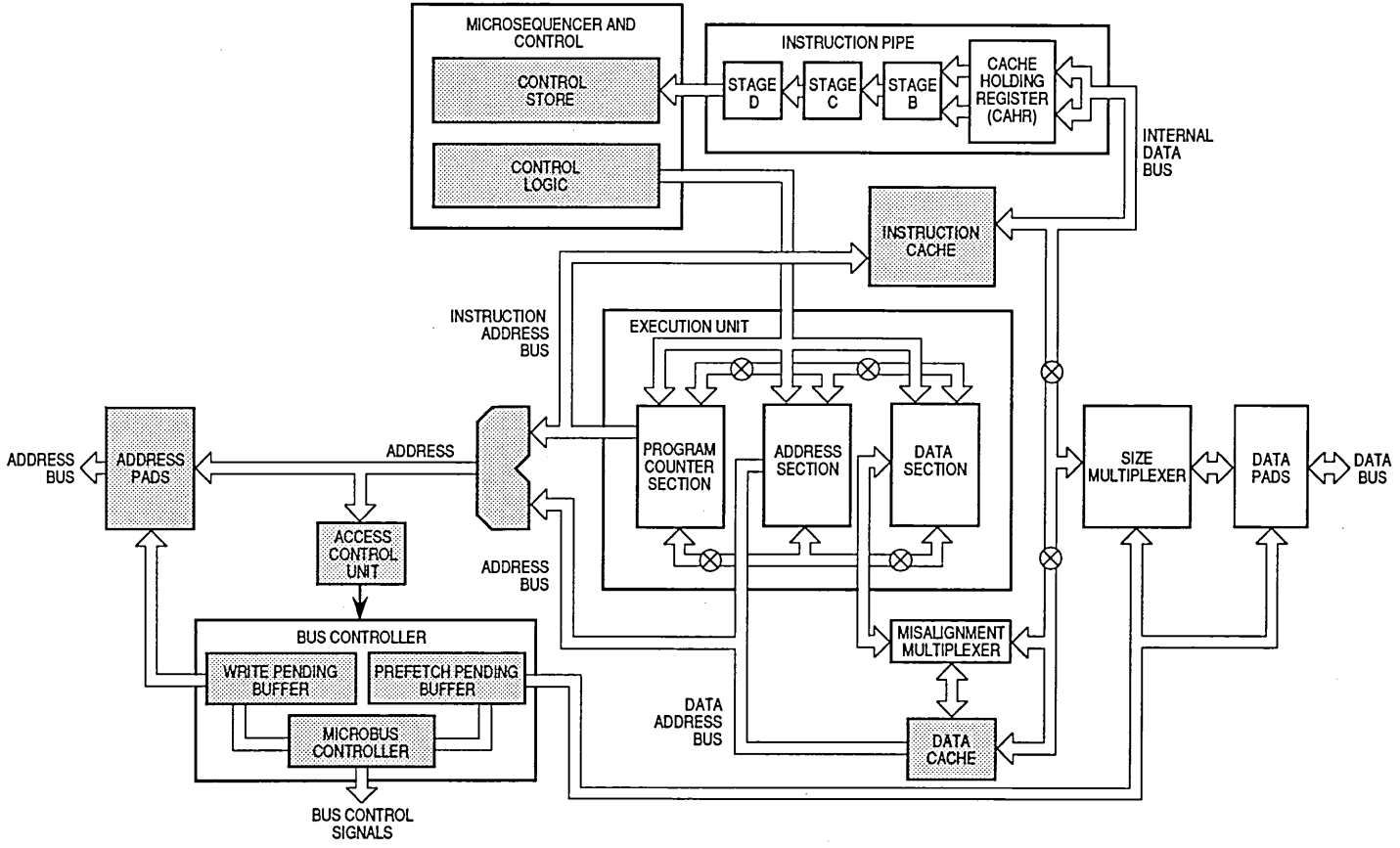


Figure 11-1. Block Diagram — Eight Independent Resources

11.2.1 Microsequencer

The microsequencer is either executing microinstructions or awaiting completion of accesses that are necessary to continue executing microcode. The bus controller is responsible for all bus activity. The microsequencer controls the bus controller, instruction execution, and internal controller operations such as calculation of effective addresses and setting of condition codes. The microsequencer initiates instruction word prefetches and controls the validation of instruction words in the instruction pipe.

11.2.2 Instruction Pipe

The MC68EC030 contains a three-word instruction pipe where instruction opcodes are decoded. As shown in Figure 11-1, instruction words (instruction operation words and all extension words) enter the pipe at stage B and proceed to stages C and D. An instruction word is completely decoded when it reaches stage D of the pipe. Each of the pipe stages has a status bit that reflects whether the word in the stage was loaded with data from a bus cycle that was terminated abnormally. Stages of the pipe are only filled in response to specific prefetch requests issued by the microsequencer.

Words are loaded into the instruction pipe from the cache holding register. While the individual stages of the pipe are only 16 bits wide, the cache holding register is 32 bits wide and contains the entire long word. This long word is obtained from the instruction cache or the external bus in response to a prefetch request from the microsequencer. When the microsequencer requests an even-word (long-word aligned) prefetch, the entire long word is accessed from the instruction cache or the external bus and loaded into the cache holding register, and the high-order word is also loaded into stage B of the pipe. The instruction word for the next sequential prefetch can then be accessed directly from the cache holding register, and no external bus cycle or instruction cache access is required. The cache holding register provides instruction words to the pipe, regardless of whether the instruction cache is enabled or disabled.

Prefetch requests are simultaneously submitted to the cache holding register, the instruction cache, and the bus controller. Thus, even if the instruction cache is disabled, an instruction prefetch may hit in the cache holding register and cause an external bus cycle to be aborted.

11.2.3 Instruction Cache

The instruction cache services the instruction prefetch portion of the microsequencer. The prefetch of an instruction that hits in the on-chip instruction cache causes no delay in instruction execution since no external bus activity is required for the prefetch. The instruction cache also interacts with the external bus during instruction cache fills following instruction cache misses.

11.2.4 Data Cache

The data cache services data reads and is updated on data writes. Data operands required by the execution unit that are accessed from the data cache cause no delay in instruction execution due to external bus activity for the data fetch. The data cache also interacts with the external bus during data cache fills following data cache misses.

11.2.5 Bus Controller Resources

Prefetches that miss in the instruction cache cause an external memory cycle to be performed. Similarly, when data reads miss in the on-chip data cache, an external memory cycle is required. The time required for either of these bus cycles may be overlapped with other internal activity.

The bus controller and microsequencer can operate on an instruction concurrently. The bus controller can perform a read or write while the microsequencer controls an effective address calculation or sets the condition codes. The microsequencer may also request a bus cycle that the bus controller cannot perform immediately. In this case, the bus cycle is queued and the bus controller runs the cycle when the current cycle is complete.

The bus controller consists of the microbus controller, the instruction fetch pending buffer, and the write pending buffer. These three resources carry out all writes and reads that miss in the on-chip caches.

11.2.5.1 INSTRUCTION FETCH PENDING BUFFER. The instruction prefetch mechanism includes a single long-word instruction fetch pending buffer. Interlocks are provided to prevent this buffer from being overwritten by an instruction prefetch request before a previously requested prefetch is completed.

11.2.5.2 WRITE PENDING BUFFER. The MC68EC030 incorporates a single write pending buffer, allowing the microsequencer to continue execution after the request for a write cycle proceeds to the bus controller. Interlocks prevent the microsequencer from overwriting this buffer.

11.2.5.3 MICROBUS CONTROLLER. The microbus controller performs the bus cycles issued to the bus controller by the rest of the controller. It implements any dynamic bus sizing required and also controls burst operations.

When prefetching instructions from external memory, the microbus controller utilizes long-word read cycles. The main controller reads two words, which may load two instructions at once or two words of a multi-word instruction into the cache holding register (and the instruction cache if it is enabled and not frozen). A special case occurs when prefetch that corresponds to an instruction word at an odd-word boundary is not found in the cache holding register (e.g., due to a branch to an odd-word location) with an instruction cache miss. From a 32-bit memory, the MC68EC030 reads both the even and odd words associated with the long-word base address in one bus cycle. From an 8- or 16-bit memory, the controller reads the even word before the odd word. Both the even and odd word are loaded into the cache holding register (and the instruction cache if it is enabled and not frozen).

11.3 INSTRUCTION EXECUTION TIMING CALCULATIONS

The instruction-cache-case timing, overlap, average no-cache-case timing, and actual instruction-cache-case execution time calculations are discussed in the following paragraphs.

11.3.1 Instruction-Cache Case

The instruction-cache-case (CC) time for an instruction is the total number of clock periods required to execute the instruction, provided all the corresponding instruction prefetches are resident in the on-chip instruction cache. All bus cycles are assumed to take two clock periods. The instruction-cache-case time does not assume any overlap with other instructions nor does it take into account hits in the on-chip data cache. The overall instruction-cache-case time for some instructions is divided into the instruction-cache-case time for the required effective address calculation (CCea) and the instruction-cache-case time for the remainder of the operation (CCop). The instruction-cache-case times for all instructions and addressing modes are listed in the tables of **11.6 INSTRUCTION TIMING TABLES**.

11.3.2 Overlap and Best Case

Overlap is the time, measured in clock periods, that an instruction executes concurrently with the previous instruction. In Figure 11-2, a portion of instructions A and B execute simultaneously. The overlap time decreases the overall execution time for the two instructions. Similarly, an overlap period between instructions B and C reduces the overall execution time of these two instructions.

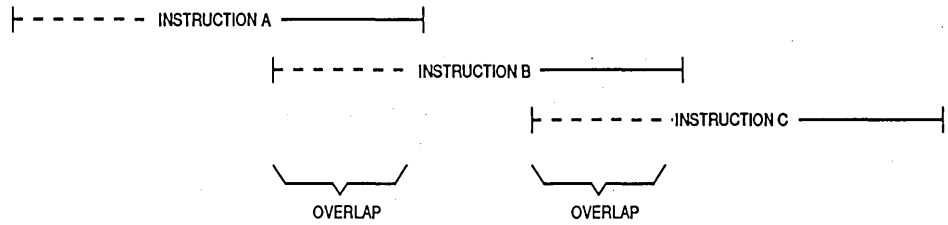


Figure 11-2. Simultaneous Instruction Execution

Each instruction contributes to the total overlap time. As shown in Figure 11-2, a portion of time at the beginning of the execution of instruction B can overlap the end of the execution time of instruction A. This time period is called the head of instruction B. The portion of time at the end of instruction A that can overlap the beginning of instruction B is called the tail of instruction A. The total overlap time between instructions A and B consists of the lesser of the tail of instruction A or the head of instruction B. Refer to the instruction timing tables in **11.6 INSTRUCTION TIMING TABLES** for head and tail times.

Figure 11-3 shows the relationship of the factors that comprise the instruction-cache-case time for either an effective address calculation (CCea) or for an operation (CCop). In Figure 11-12, the best case execution time for instruction B occurs when the instruction-cache-case times for instruction B and instruction A overlap so that the head of instruction B is completely overlapped with the tail of instruction A.

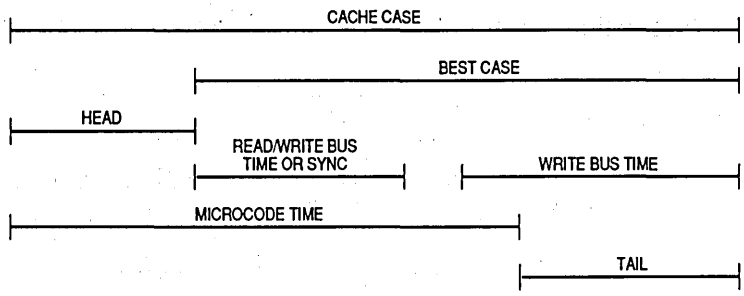


Figure 11-3. Derivation of Instruction Overlap Time

The nature of the instruction overlap and the fact that the heads of some instructions equal the total instruction-cache-case time for those instructions makes a zero net execution time possible. The execution time of an instruction is completely absorbed by overlap with the previous instruction.

11.3.3 Average No-Cache Case

The average no-cache-case (NCC) time for an instruction takes into account the time required for the microcode to execute plus the time required for all external bus activity. This time is calculated assuming both caches miss and the associated instruction prefetches require one external bus cycle per two instruction prefetches. Refer to **11.2.2 Instruction Pipe**. The average no-cache-case time also assumes no overlap. *All bus cycles are assumed to take two clock periods.* Average no-cache-case times for instructions and effective address calculations are listed in **11.6 INSTRUCTION TIMING TABLES**. *Because the no-cache-case times assume no overlap, the head and tail values listed in these tables do not apply to the no-cache-case values.*

Since the actual no-cache-case time depends on the alignment of prefetches associated with an instruction, both alignment cases were considered, and the value shown in the table is the average of the odd-word-aligned case and the even-word-aligned case (rounded up to an integral number of clocks). Similarly, the number of prefetch bus cycles is the average of these two cases rounded up to an integral number of bus cycles.

The effect of instruction alignment on timing is illustrated by the following example. The assumptions referred to in **11.6 INSTRUCTION TIMING TABLES** apply. Both the data cache and instruction cache miss on all accesses.

		Instruction
1.	MOVE.L	(d_{16},An,Dn),Dn
2.	CMPI.W	#⟨data⟩.W,(d_{16},An)

The instruction stream is positioned with even alignment in 32-bit memory as:

Address			
	n	MOVE	EA Ext
	n+4	d_{16}	CMPI
	n+8	#⟨data.W⟩	d_{16}
	n+12	. . .	. . .

Figure 11-4 shows controller activity for even alignment of the given instruction stream.

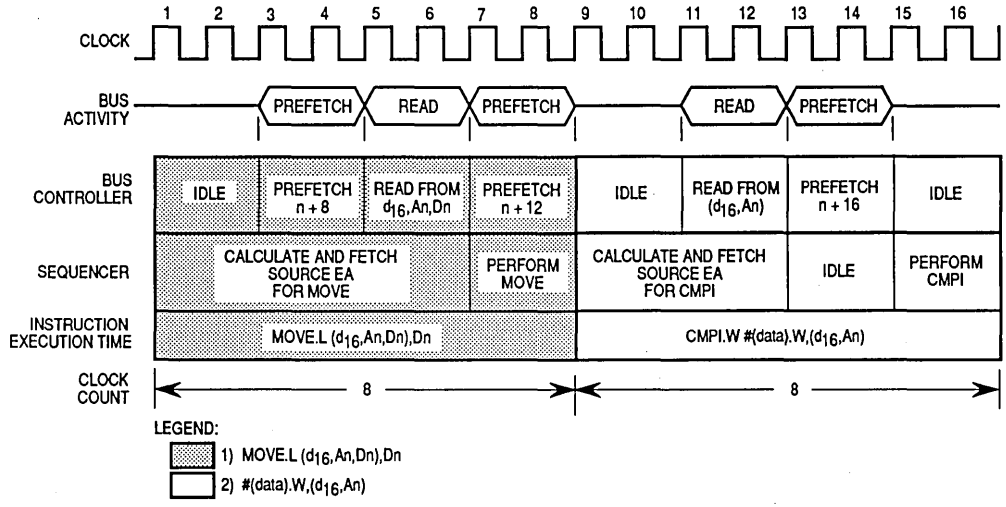


Figure 11-4. Controller Activity — Even Alignment

11

The instruction stream is positioned in 32-bit memory as:

Address			
	n	. . .	MOVE
	n+4	EA Ext	d_{16}
	n+8	CMPI	#⟨data.W⟩
	n+12	d_{16}	. . .

Figure 11-5 shows controller activity for odd alignment.

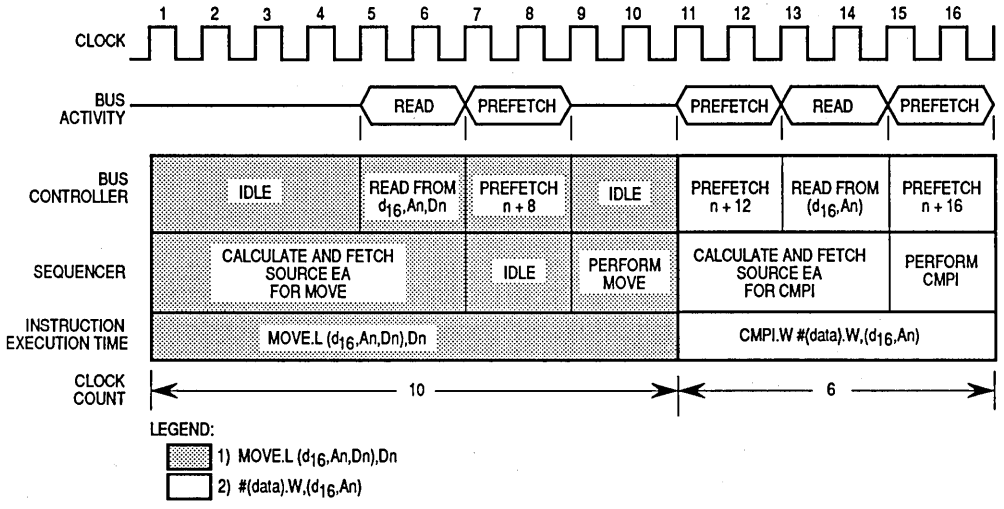


Figure 11-5. Processor Activity — Odd Alignment

Comparing the two alignments, the execution time of the MOVE instruction is 8 clocks for even alignment and 10 clocks for odd alignment, an average of 9 clocks. Referring to the table in **11.6.6 MOVE Instruction** and the table in **11.6.1 Fetch Effective Address (fea)**, the average no-cache-case time is 2+7=9 clocks. A similar calculation can be made of the CMPI instruction, which has an average no-cache-case time of seven clocks.

The average no-cache-case timing rather than the maximum no-cache-case timing gives a closer approximation of the actual timing of an instruction stream in many cases. The total execution time of the two instructions in the previous example is 16 clocks for both even and odd alignment. Adding the average no-cache-case timing of the given instructions also gives 16 clocks (9+7=16 clocks). It should be noted again that the no-cache-case time assumes no overlap. Therefore, the actual execution time of an instruction stream may be less than that given by adding the no-cache-case times. To factor in the effect of wait states for the no-cache case, refer to **11.5 EFFECT OF WAIT STATES**.

11.3.4 Actual Instruction-Cache-Case Execution Time Calculations

The overall execution time for an instruction may depend on the overlap with the previous and following instructions. Therefore, to calculate instruction execution time estimations, the entire code sequence to be evaluated must be analyzed as a whole. To derive the actual instruction-cache-case

execution times for an instruction sequence (under the assumptions listed in **11.6 INSTRUCTION TIMING TABLES**), the instruction-cache-case times listed in the tables must be used, and the proper overlap must be subtracted for the entire sequence. The formula for this calculation is as follows:

$$CC_1 + [CC_2 - \min(H_2,T_1)] + [(CC_3 - \min(H_3,T_2)] + \ldots \quad (11\text{-}1)$$

where:

CC_n is the instruction-cache-case time for an instruction,
T_n is the tail time for an instruction,
H_n is the head time for an instruction, and
min(a,b) is the minimum of parameters a and b.

The instruction-cache-case time for most instructions is composed of the instruction-cache-case time for the effective address calculation (CCea) overlapped with the instruction-cache-case time for the operation (CCop). The more specific formula is as follows:

$$CCea_1 + [CCop_1 - \min(Hop_1,Tea_1)] + [CCea_2 - \min(Hea_2,Top_1)] + [CCop_2 - \min(Hop_2,Tea_2)] + [CCea_3 - \min(Hea_3,Top_2)] + \ldots \quad (11\text{-}2)$$

where:

$CCea_n$ is the effective address time for the instruction-cache case,
$CCop_n$ is the instruction-cache-case time for the operation portion of an instruction,
Tea_n is the tail time for the effective address of an instruction,
Hop_n is the head time for the operation portion of an instruction,
Top_n is the tail time for the operation portion of an instruction,
Hea_n is the head time for the effective address of an instruction, and
min(a,b) is the minimum of parameters a and b.

The instructions that require the instruction-cache case, head, and tail of an effective address (CCea, Hea, and Tea) to be overlapped with CCop, Hop, and Top are footnoted in **11.6 INSTRUCTION TIMING TABLES**.

The actual instruction-cache-case execution time for a stream of instructions can be computed using Equation (11-1) or the general Equation (11-2). Equation (11-1) is used unless the instruction-cache case, head, and tail of an effective address are required.

An example using a series of instructions that require Equation (11-1) to calculate the instruction-cache-case execution time follows. The assumptions referred to in **11.6 INSTRUCTION TIMING TABLES** apply.

		Instruction
1.	ADD.L	A1,D1
2.	SUBA.L	D1,A2

Referring to the timing table in **11.6.8 Arithmetic/Logical Instructions**, the head, tail, and instruction-cache-case (CC) times for ADD.L A1,D1 and SUBA.L D1,A2 are found. There is no footnote directing the user to add an effective address time for either instruction. Since both of the instructions use register operands only, there is no need to add effective address calculation times. Therefore, the general Equation (11-1) can be used for both.

		Head	Tail	CC
1.	ADD.L A1,D1	2	0	2
2.	SUBA.L D1,A2	4	0	4

NOTE

The underlined numbers show the typical pattern for the comparison of head and tail in the following equation.

The following computations use Equation (11-1):

$$\begin{aligned}\text{Execution Time} &= CC_1 + [CC_2 - \min(H_2, T_1)] \\ &= 2 + [4 - \min(4,0)] \\ &= 2 + [4 - 0] \\ &= 6 \text{ clocks}\end{aligned}$$

11

Instructions that require the addition of an effective address calculation time from an appropriate table use the general Equation (11-2) to calculate the actual CC time. The CCea, Hea, and Tea values must be extracted from the appropriate effective address table (either fetch effective address, fetch immediate effective address, calculate effective address, calculate immediate effective address, or jump effective address) as indicated and included in Equation (11-2). All of the following instructions (except the last) require general Equation (11-2). The last instruction uses Equation (11-1).

		Instruction
1.	ADD.L	−(A1),D1
2.	AND.L	D1,([A2])
3.	MOVE.L	(A6),(8,A1)
4.	TAS	(A3)+
5.	NEG	D3

Using the appropriate operation and effective address tables from **11.6 INSTRUCTION TIMING TABLES**:

		Head	Tail	CC
1.	ADD.L –(A1),D1			
	Fetch Effective Address (fea) –(An)	2	2	4
	ADD EA,Dn	0	0	2
2.	AND.L D1,([A2])			
	fea ([B])	4	0	10
	AND Dn,EA	0	1	3
3.	MOVE.L (A6),(8,A1)			
	fea (An)	1	1	3
	MOVE Source,(d_{16},An)	2	0	4
4.	TAS (A3)+			
	Calculate Effective Address (cea) (An)+	0	0	2
	TAS Mem	3	0	12
5.	NEG D3	2	0	2

The following calculations use Equations (11-1) and (11-2):

$$\begin{aligned}\text{Execution Time} &= CCea_1 + [CCop_1 - \min(Hop_1,Tea_1)] + [CCea_2 - \min(Hea_2,Top_1)] + \\ &\quad [CCop_2 - \min(Hop_2,Tea_2)] + [CCea_3 - \min(Hea_3,Top_2)] + \\ &\quad [CCop_3 - \min(Hop_3,Tea_3)] + [CCea_4 - \min(Hop_4,Top_3)] + \\ &\quad [CCop_4 - \min(Hop_4,Top_3)] + [CCop_5 - \min(Hop_5,Top_4)] \\ &= 4 + [2 - \min(0,2)] + [10 - \min(4,0)] + [3 - \min(0,0)] + [3 - \min(1,1)] + \\ &\quad [4 - \min(2,1)] + [2 - \min(0,0)] + [12 - \min(3,0)] + [2 - \min(2,0)] \\ &= 4 + 2 + 10 + 3 + 2 + 3 + 2 + 12 + 2 \\ &= 40 \text{ clock periods}\end{aligned}$$

Notice that the last instruction did not require the general Equation (11-2) since there were no effective address (ea) additions. Therefore, Equation (11-1) is used:

$$CCop_5 - \min(Hop_5,Top_4)$$

When using the fetch immediate effective address (fiea) or the calculate immediate effective address (ciea) tables, the size of the data is significant in the timing calculations. For each effective address, a line is listed for word data, #<data>.W, and for long data, #<data>.L.

The total head of some effective address types extends through the effective address calculation and includes the head of the operation. These effective address calculations are marked in the head column as follows:

X + op head

where:

X is the head of the effective address alone.

An example using the fiea table and the X + op head notation is:

Instruction

1. EORI.W #$400,–(A1)
2. ADDI.L #$6000FF,D1

	Head	Tail	CC
1. EORI.W #$400,–(A1)			
fiea #<data>.W,–(An)	2	2	4
EORI #<data>,Mem	0	1	3
2. ADDI.L #$6000FF,D1			
fiea #<data>.L,D1	4 + op head	0	4
	6	0	4
ADDI #<data>,Dn	2(op head)	0	2

The following calculations use the general Equation (11-2):

$$\begin{aligned}\text{Execution Time} &= CCea_1 + [CCop_1 - \min(Hop_1, Tea_1)] + [CCea_2 - \min(Hea_2, Top_1)] + \\ &\quad [CCop_2 - \min(Hop_2, Tea_2)] \\ &= 4 + [3 - \min(0,2)] + [4 - \min(6,1)] + [2 - \min(2,0)] \\ &= 4 + 3 + 3 + 2 \\ &= 12 \text{ clock periods}\end{aligned}$$

Note that for the head of fiea #<data>.L,D1, 4 + op head, the resulting head of 6 is larger than the instruction-cache-case time of the fetch. A negative number for the execution time of that portion could result (e.g., $4 - \min(6,6) = -2$). This result would produce the correct execution time since the fetch was completely overlapped and the operation was partially overlapped by the same tail. No changes in the calculation for the operation execution time are required.

Many two-word instructions (e.g., MULU.L, DIV.L, BFSET, etc.) include the fetch immediate effective address (fiea) time or the calculate immediate effective address (ciea) time in the execution time calculation. The timing for immediate data of word length (#<data>.W) is used for these calculations. If the instruction has a source and a destination, the source EA is used for the table lookup. If the instruction is single operand, the effective address of that operand is used.

The following example includes multi-word instructions that refer to the fetch immediate effective address and calculate immediate effective address tables in **11.6 INSTRUCTION TIMING TABLES.**

	Instruction	
1.	MULU.L	(D7),D1:D2
2.	BFCLR	$6000{0:8}
3.	DIVS.L	#$10000,D3:D4

	Head	**Tail**	**CC**
1. MULU.L (D7),D1:D2			
fiea #<data>.W,Dn	2+op head	0	2
	4	0	2
MUL.L EA, Dn	2(op head)	0	44
2. BFCLR $6000{0:8}			
fiea #<data>.W,$XXX.W	4	2	6
BFCLR Mem(<5 bytes)	6	0	14
3. DIVS.L #$10000,D3:D4			
fiea #<data>.W,#<data>.L	6+op head	0	6
	6	0	6
DIVS.L EA,Dn	0(op head)	0	90

Use the general Equation (11-2) to compute:

$$\text{Execution Time} = CCea_1 + [CCop_1 - \min(Hop_1,Tea_1)] + [CCea_2 - \min(Hea_2,Top_1)] + [CCop_2 - \min(Hop_2,Tea_2)] + [CCea_3 - \min(Hea_3,Top_2)] + [CCop_3 - \min(Hop_3,Tea_3)]$$

$$= 2 + [44 - \min(2,0)] + [6 - \min(4,0)] + [14 - \min(6,2)] + [6 - \min(6,0)] + [90 - \min(0,0)]$$

$$= 2 + 44 + 6 + 12 + 6 + 90$$

$$= 160 \text{ clock periods}$$

NOTE

This CC time is a maximum since the times given for the MULU.L and DIVS.L are maximums.

11.4 EFFECT OF DATA CACHE

When the data accesses required by an instruction are in the data cache, reading these operands requires no bus cycles, and the execution time for the instruction may be minimized. Write accesses, however, always require bus cycles because the data cache is a write-through cache.

The effect of the data cache on operand read accesses can be factored into the actual instruction execution time as follows.

When a data cache hit occurs for the data fetch corresponding to either the fetch effective address table or the fetch immediate effective address table in **11.6 INSTRUCTION TIMING TABLES**, the following rules apply:

1a. if $Tail_t = 0$: No change in timing.

1b. if $Tail_t = 1$: $Tail = Tail_t - 1$
$CC = CC_t - 1$

1c. if $Tail_t > 1$: $Tail = Tail_t - (Tail_t - 1) = 1$
$CC = CC_t - (Tail_t - 1)$

where:
$Tail_t$ and CC_t are the values listed in the tables.

2. If the EA mode is memory indirect (two data reads), the tail and CC time are calculated as for one data read.

NOTE

Data cache hits cannot easily be accounted for in instruction and operation timings that include an operand fetch in the CCop (e.g., BFFFO and CHK2). The effect of a data cache hit on such CCop's has been ignored for computational purposes.

RMC cycles (e.g., TAS and CAS) are forced to miss on data cache reads. Therefore, a data cache hit has no effect on these instructions.

The following example assumes data cache hits. The lines that are corrected for data cache hits are printed in **boldface** type. These lines are used to calculate the instruction-cache-case execution time. References are to the preceding rules.

	Instruction	
1.	ADD.L	−(A1),D1
2.	AND.L	D1,([A2])
3.	MOVE.L	(A6),(8,A1)
4.	TAS	(A3)

		Head	Tail	CC
1.	ADD.L –(A1),D1			
	Fetch Effective Address			
	fea –(An)	2	2–1	4–1(1/0/0)
	***1c**	**2**	**1**	**3(1/0/0)**
	***ADD EA,Dn**	**0**	**0**	**2(0/0/1)**
2.	AND.L D1,([A2])			
	***1a & 2 fea ([B])**	**4**	**0**	**10(2/0/0)**
	***AND Dn,EA**	**0**	**1**	**3(0/0/1)**
3.	MOVE.L (A6),(8,A1)			
	fea (An)	1	1–1	3–1(1/0/0)
	***1b**	**1**	**0**	**2(1/0/0)**
	***MOVE Source, (d_{16},An)**	**2**	**0**	**4(0/0/1)**
4.	TAS (A3)+			
	***Cea (An)+**	**0**	**0**	**2(0/0/0)**
	***TAS Mem**	**0**	**0**	**12(1/0/1)**

*Corrected for data cache hits.

NOTE

It is helpful to include the number of operand reads and writes along with the number of instruction accesses in the CC column for computing the effect of data cache hits on execution time.

The following computations use the general Equation (11-2):

$$\begin{aligned}\text{Execution Time} &= CCea_1 + [CCop_1 - \min(Hop_1,Tea_1)] + [CCea_2 - \min(Hea_2,Top_1)] + \\ &\quad [CCop_2 - \min(Hop_2,Tea_2)] + [CCea_3 - \min(Hea_3,Top_2)] + \\ &\quad [CCop_3 - \min(Hop_3,Tea_3)] + [CCea_4 - \min(Hea_4,Top_3)] + \\ &\quad [CCop_4 - \min(Hop_4,Tea_4)] \\ &= 3 + [2 - \min(0,1)] + [10 - \min(4,0)] + [3 - \min(0,0)] + [2 - \min(1,1)] + \\ &\quad [4 - \min(2,0)] + [2 - \min(0,0)] + [12 - \min(0,0)] \\ &= 3 + 2 + 10 + 3 + 1 + 4 + 2 + 12 \\ &= 37 \text{ clock periods}\end{aligned}$$

11.5 EFFECT OF WAIT STATES

The constraints of a system design may require the insertion of wait states in memory cycles. When the bus or the memory device requires many wait states, instruction execution time is increased. However, one or two wait states may have little effect on instruction timing. Often the only effect of one or more wait states is to reduce bus idle time.

The effect of wait states on data accesses may be accounted for in the instruction-cache-case timings.

To add the effect of wait states on data accesses:

1a. For nonmemory indirect effective address timings that include an operand read, add the number of wait states (in clocks) to the tail and instruction-cache-case (CC) times. The head is not affected.

1b. For memory indirect effective address timings that use the calculate <ea> tables and have only one data read (for the address fetch), add the number of wait states to the CC time only. The head and tail are not affected.

1c. For memory indirect effective address timings (fetch <ea>) that have two data reads (for the address fetch), add the number of wait states for two reads to the CC time. Add the number of wait states for one data read to the tail. The head is not affected.

2a. For operation timings that include a data read (e.g., BFFF0 and TAS), add the number of wait states to the CC time only. Neither the head nor the tail are affected.

NOTE

The CC timing and tail of the MOVEM instruction are special cases for both data reads and writes. Equations for both the CC timing and the tail as a function of wait states are footnoted in the table in **11.6.7 Special-Purpose MOVE Instruction**.

2b. If the operation has more than one data read, add the total amount of wait states for all reads to the CC time. Neither the head nor the tail are affected. Refer to preceding note.

3a. For operation timings that include a data write, the number of wait states is added to the tail and the CC time. The head is not affected. Refer to preceding note.

3b. If there is more than one write in the operation, the tail is only increased by the wait states for one write. The CC timing is increased by the total amount of wait states for all writes. Refer to preceding note.

The following example calculates the instruction-cache-case execution time for the specified instruction stream with two wait states (four-clock reads and writes). The lines that are corrected for wait states are printed in boldface type and are used to calculate the instruction execution time. References are to the preceding rules.

		Instruction
1.	MOVE.L	($800,A2,D3),(A5,D2)
2.	ADD.L	D1,([$30,A4])
3.	BFCLR	($20,A5){1:5} — (<5 bytes)
4.	BFTST	($10,A3,D3){31:31} — (5 bytes)
5.	MOVEM	([A1,D1]),A1-A4 — 4 registers

Wait States = 2

		Head	**Tail**	**CC**
1.	MOVE.L ($800,A2,D3),(A5,D2)			
	fea (d_{16},An,Xn)	4	0+2	6+2(1/0/0)
	***1a**	**4**	**2**	**8(1/0/0)**
	MOVE Source,(B)	4	0+2	8+2(0/0/1)
	***3a**	**4**	**2**	**10(0/0/1)**
2.	ADD.L D1,([$30,A4])			
	fea ([d_{16},B])	4	0+2	12+4(2/0/0)
	***1c**	**4**	**2**	**16(2/0/0)**
	ADD Dn,EA	0	1+2	3+2(0/0/1)
	***3a**	**0**	**3**	**5(0/0/1)**
3.	BFCLR ($20,A5){1:5}			
	***ciea #<data>.W,(d_{16},An)**	**10**	**0**	**4(0/0/0)**
	Single EA Format			
	BFCLR Mem (< 5 bytes)	6	0+2	14+4(1/0/1)
	***2a & 3a**	**6**	**2**	**18(1/0/1)**

4.	BFTST ($10,A3,D3){31:31}			
	***ciea (d_{16},An,Xn)**	**14**	**0**	**8(0/0/0)**
	BFTST Mem (5 bytes)	6	0	14+4(2/0/0)
	***2b**	**6**	**0**	**18(2/0/0)**
5.	MOVEM ([A1,D1]),A1-A4			
	ciea ([B])	6	0	12+2(1/0/0)
	***1b**	**6**	0	**14(1/0/0)**
	MOVEM EA,RL	2	0	24+0(4/0/0)
	***2a & 2b**	**2**	**0**	**24(4/0/0)**

*Corrected for wait states.

NOTE

It is helpful to include the number of operand read and writes along with the number of instruction accesses in the CC column for computing the effect of wait states on execution time.

Using the general Equation (11-2), calculate as follows:

$$\begin{aligned}\text{Execution Time} &= CCea_1 + [CCop_1 - \min(Hop_1,Tea_1)] + [CCea_2 - \min(Hea_2,Top_1)] + \\ &\quad [CCop_2 - \min(Hop_2,Tea_2)] + [CCea_3 - \min(Hea_3,Top_2)] + \\ &\quad [CCop_3 - \min(Hop_3,Tea_3)] + [CCea_4 - \min(Hea_4,Top_3)] + \\ &\quad [CCop_4 - \min(Hop_4,Tea_4)] + [CCea_5 - \min(Hea_5,Top_4)] + \\ &\quad [CCop_5 - \min(Hop_5,Tea_5)] \\ &= 8 + [10 - \min(4,2)] + [16 - \min(4,2)] + \\ &\quad [5 - \min(0,2)] + [4 - \min(10,3)] + [18 - \min(6,0)] + [8 - \min(14,2)] + \\ &\quad [18 - \min(6,0)] + [14 - \min(6,0)] + \\ &\quad [24 - \min(2,0)] \\ &= 8 + 8 + 14 + 5 + 1 + 18 + 6 + 18 + 14 + 24 \\ &= 116 \text{ clock periods}\end{aligned}$$

The next example is the data cache hit example from **11.4 EFFECT OF DATA CACHE** with two wait states per cycle (four-clock read/write). Hits in the data cache and instruction cache are assumed. Three lines are shown for each timing. The first is the timing from the appropriate table. The second is the timing adjusted for a data cache hit. The third *adds wait states only to write operations*, since the read operations hit in the cache and cause no delay. The third line for each timing is used to calculate the instruction cache execution time; it is shown in boldface type.

	Instruction	
1.	ADD.L	−(A1),D1
2.	AND.L	D1,([A2])
3.	MOVE.L	(A6),(8,A1)
4.	TAS	(A3)+

		Head	Tail	CC
1.	ADD.L −(A1),D1			
	fea −(An)	2	2	4(1/0/0)
	*	2	1	3(1/0/0)
	**	**2**	**1**	**3(1/0/0)**
	ADD.L EA,Dn	0	0	2(0/1/0)
	*	0	0	2(0/1/0)
	**	**0**	**0**	**2(0/1/0)**
2.	AND.L D1,([A1])			
	fea ([B])	4	0	10(1/0/0)
	*	4	0	10(1/0/0)
	***	**4**	**0**	**12(1/0/0)**
	AND Dn,EA	0	1	3(0/0/1)
	*	0	1	3(0/0/1)
	**	**0**	**3**	**5(0/0/1)**
3.	MOVE.L (A6),(8,A1)			
	fea (An)	1	1	3(1/0/0)
	*	1	0	2(1/0/0)
	**	**1**	**0**	**2(1/0/0)**
	MOVE Source,(d_{16},An)	2	0	4(0/0/1)
	*	2	0	4(0/0/1)
	**	**2**	**2**	**6(0/0/1)**
4.	TAS (A3)+			
	Cea (An)	0	0	2(0/0/0)
	*	0	0	2(0/0/0)
	**	**0**	**0**	**2(0/0/0)**
	TAS Mem	3	0	12(1/0/1)
	*	3	0	12(1/0/1)
	**	**3**	**0**	**14(1/0/1)**

NOTES:
*Corrected for data cache hits.
**Corrected for wait states also (only on data writes).
***No data cache hit assumed for address fetch.

Using the general Equation (11-2), calculate as follows:

$$\begin{aligned}\text{Execution Time} &= CCea_1 + [CCop_1 - \min(Hea_1, Top_1)] + [CCea_2 - \min(Hea_2, Top_1)] + \\ &\quad [CCop_2 - \min(Hop_2, Tea_2)] + [CCea_3 - \min(Hea_3, Top_2)] + \\ &\quad [CCop_3 - \min(Hop_3, Tea_3)] + [CCea_4 - \min(Hea_4, Top_3)] + \\ &\quad [CCop_4 - \min(Hop_4, Tea_4)] \\ &= 3 + [2 - \min(0,1)]m + [12 - \min(4,0)] + \\ &\quad [5 - \min(0,0)] + [2 - \min(1,3)] + \\ &\quad [6 - \min(2,0)] + [2 - \min(0,2)] + \\ &\quad [14 - \min(3,0)] \\ &= 3 + 2 + 12 + 5 + 1 + 6 + 2 + 14 \\ &= 45 \text{ clock periods}\end{aligned}$$

A similar analysis can be constructed for the average no-cache case. Since the average no-cache-case time assumes two clock periods per bus cycle (i.e., no wait states), the timing given in the tables does not apply directly to systems with wait states. To approximate the average no-cache-case time for an instruction or effective address with W wait states, use the following formula:

$$NCC = NCC_t + (\#\text{ of data reads and writes}) \cdot W + (\text{max. }\#\text{ of instruction accesses}) \cdot W$$

where:

NCC_t is the no-cache-case timing value from the appropriate table.

The number of data reads, data writes, and maximum instruction accesses are found in the appropriate table.

The average no-cache-case timing obtained from this formula is equal to or greater than the actual no-cache-case timing since the number of instruction accesses used is a maximum (the values in the tables are always rounded up) and no overlap is assumed.

11.6 INSTRUCTION TIMING TABLES

All the following assumptions apply to the times shown in the tables in this section:

- All memory accesses occur with two-clock bus cycles and no wait states.
- All operands in memory, including the system stack, are long-word aligned.
- A 32-bit bus is used for communications between the MC68EC030 and system memory.
- The data cache is not enabled.
- No exceptions occur (except as specified).

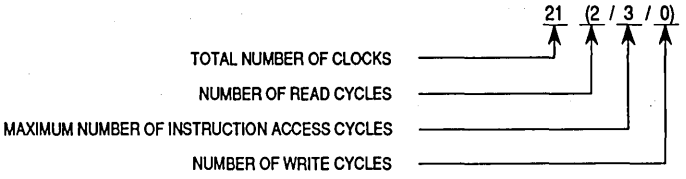
Four values are listed for each instruction and effective address:

1. Head,
2. Tail,
3. Instruction-cache case (CC) when the instruction is in the cache but has no overlap, and
4. Average no-cache case (NCC) when the instruction is not in the cache or the cache is disabled and there is no instruction overlap.

The only instances for which the size of the operand has any effect are the instructions with immediate operands and the ADDA and SUBA instructions. Unless specified otherwise, immediate byte and word operands have identical execution times.

The instruction-cache-case and average no-cache-case columns of the instruction timing tables contain four sets of numbers, three of which are enclosed in parentheses. The outer number is the total number of clocks for the given cache case and instruction. The first number inside the parentheses is the number of operand read cycles performed by the instruction. The second value inside the parentheses is the maximum number of instruction bus cycles performed by the instruction, including all prefetches to keep the instruction pipe filled. Because the second value is the average of the odd-word-aligned case and the even-word-aligned case (rounded up to an integral number of bus cycles), it is always greater than or equal to the actual number

of bus cycles (one bus cycle per two instruction prefetches). The third value within the parentheses is the number of write cycles performed by the instruction. One example from the instruction timing table is as follows:



The total numbers of bus-activity clocks and internal clocks (not overlapped by bus activity) of the instruction in this example are derived as follows:

(2 Reads • 2 Clocks/Read) + (3 Instruction Accesses • 2 Clocks/Access) +
(0 Writes • 2 Clocks/Write) = 10 Clocks of Bus Activity
21 Total Clocks − 10 Bus Activity Clocks = 11 Internal Clocks

The example used here is taken from a no-cache-case 'fetch effective address' time. The addressing mode is ([d_{32},B],I,d_{32}). The same addressing mode under the instruction-cache-case execution time entry is 18(2/0/0). For the instruction-cache-case execution time, no instruction accesses are required because the cache is enabled and the sequencer does not have to access external memory for the instruction words.

The first five timing tables deal exclusively with fetching and calculating effective addresses and immediate operands. The remaining tables are instruction and operation timings. Some instructions use addressing modes that are not included in the corresponding instruction timings. These cases refer to footnotes that indicate the additional table needed for the timing calculation. All read and write accesses are assumed to take two clock periods.

11.6.1 Fetch Effective Address (fea)

The fetch effective address table indicates the number of clock periods needed for the controller to calculate and fetch the specified effective address. The effective addresses are divided by their formats (refer to **2.5 Effective Address Encoding Summary**). For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% Dn	—	—	**0**(0/0/0)	**0**(0/0/0)
% An	—	—	**0**(0/0/0)	**0**(0/0/0)
(An)	1	1	**3**(1/0/0)	**3**(1/0/0)
(An)+	0	1	**3**(1/0/0)	**3**(1/0/0)
−(An)	2	2	**4**(1/0/0)	**4**(1/0/0)
(d_{16},An) or (d_{16},PC)	2	2	**4**(1/0/0)	**4**(1/1/0)
(xxx).W	2	2	**4**(1/0/0)	**4**(1/1/0)
(xxx).L	1	0	**4**(1/0/0)	**5**(1/1/0)
#⟨data⟩.B	2	0	**2**(0/0/0)	**2**(0/1/0)
#⟨data⟩.W	2	0	**2**(0/0/0)	**2**(0/1/0)
#⟨data⟩.L	4	0	**4**(0/0/0)	**4**(0/1/0)
BRIEF FORMAT EXTENSION WORD				
(d_8,An,Xn) or (d_8,PC,Xn)	4	2	**6**(1/0/0)	**6**(1/1/0)
FULL FORMAT EXTENSION WORD(S)				
(d_{16},An) or (d_{16},PC)	2	0	**6**(1/0/0)	**7**(1/1/0)
(d_{16},An,Xn) or (d_{16},PC,Xn)	4	0	**6**(1/0/0)	**7**(1/1/0)
([d_{16},An]) or ([d_{16},PC])	2	0	**10**(2/0/0)	**10**(2/1/0)
([d_{16},An],Xn) or ([d_{16},PC],Xn)	2	0	**10**(2/0/0)	**10**(2/1/0)
([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	2	0	**12**(2/0/0)	**13**(2/2/0)
([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	2	0	**12**(2/0/0)	**13**(2/2/0)
([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	2	0	**12**(2/0/0)	**14**(2/2/0)
([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	2	0	**12**(2/0/0)	**14**(2/2/0)
(B)	4	0	**6**(1/0/0)	**7**(1/1/0)
(d_{16},B)	4	0	**8**(1/0/0)	**10**(1/1/0)
(d_{32},B)	4	0	**12**(1/0/0)	**13**(1/2/0)
([B])	4	0	**10**(2/0/0)	**10**(2/1/0)
([B],I)	4	0	**10**(2/0/0)	**10**(2/1/0)
([B],d_{16})	4	0	**12**(2/0/0)	**13**(2/1/0)

11.6.1 Fetch Effective Address (fea) (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXTENSION WORD(S) (CONTINUED)				
([B],I,d_{16})	4	0	**12**(2/0/0)	**13**(2/1/0)
([B],d_{32})	4	0	**12**(2/0/0)	**14**(2/2/0)
([B],I,d_{32})	4	0	**12**(2/0/0)	**14**(2/2/0)
([d_{16},B])	4	0	**12**(2/0/0)	**13**(2/1/0)
([d_{16},B],I)	4	0	**12**(2/0/0)	**13**(2/1/0)
([d_{16},B],d_{16})	4	0	**14**(2/0/0)	**16**(2/2/0)
([d_{16},B],I,d_{16})	4	0	**14**(2/0/0)	**16**(2/2/0)
([d_{16},B],d_{32})	4	0	**14**(2/0/0)	**17**(2/2/0)
([d_{16},B],I,d_{32})	4	0	**14**(2/0/0)	**17**(2/2/0)
([d_{32},B])	4	0	**16**(2/0/0)	**17**(2/2/0)
([d_{32},B],I)	4	0	**16**(2/0/0)	**17**(2/2/0)
([d_{32},B],d_{16})	4	0	**18**(2/0/0)	**20**(2/2/0)
([d_{32},B],I,d_{16})	4	0	**18**(2/0/0)	**20**(2/2/0)
([d_{32},B],d_{32})	4	0	**18**(2/0/0)	**21**(2/3/0)
([d_{32},B],I,d_{32})	4	0	**18**(2/0/0)	**21**(2/3/0)

B= Base Address; 0, An, PC, Xn, An+Xn, PC+Xn. Form does not affect timing.
I= Index; 0, Xn
%= No clock cycles incurred by effective address fetch.

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn do not affect timing.

11.6.2 Fetch Immediate Effective Address (fiea)

The fetch immediate effective address table indicates the number of clock periods needed for the controller to fetch the immediate source operand and to calculate and fetch the specified destination operand. In the case of two-word instructions, this table indicates the number of clock periods needed for the controller to fetch the second word of the instruction and to calculate and fetch the specified source operand or single operand. The effective addresses are divided by their formats (refer to **2.5 Effective Address Encoding Summary**). For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

11.6.2 Fetch Immediate Effective Address (fiea) (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% #⟨data⟩.W,Dn	2+op head	0	**2**(0/0/0)	**2**(0/1/0)
% #⟨data⟩.L,Dn	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
#⟨data⟩.W,(An)	1	1	**3**(1/0/0)	**4**(1/1/0)
#⟨data⟩.L,(An)	1	0	**4**(1/0/0)	**5**(1/1/0)
#⟨data⟩.W,(An)+	2	1	**5**(1/0/0)	**5**(1/1/0)
#⟨data⟩.L,(An)+	4	1	**7**(1/0/0)	**7**(1/1/0)
#⟨data⟩.W,−(An)	2	2	**4**(1/0/0)	**4**(1/1/0)
#⟨data⟩.L,−(An)	2	0	**4**(1/0/0)	**5**(1/1/0)
#⟨data⟩.W,(d_{16},An)	2	0	**4**(1/0/0)	**5**(1/1/0)
#⟨data⟩.L,(d_{16},An)	4	0	**6**(1/0/0)	**8**(1/2/0)
#⟨data⟩.W,$XXX.W	4	2	**6**(1/0/0)	**6**(1/1/0)
#⟨data⟩.L,$XXX.W	6	2	**8**(1/0/0)	**8**(1/2/0)
#⟨data⟩.W,$XXX.L	3	0	**6**(1/0/0)	**7**(1/2/0)
#⟨data⟩.L,$XXX.L	5	0	**8**(1/0/0)	**9**(1/2/0)
#⟨data⟩.W,#⟨data⟩.L	6+op head	0	**6**(0/0/0)	**6**(0/2/0)
BRIEF FORMAT EXTENSION WORD				
#⟨data⟩.W,(d_8,An,Xn) or (d_8,PC,Xn)	6	2	**8**(1/0/0)	**8**(1/2/0)
#⟨data⟩.L,(d_8,An,Xn) or (d_8,PC,Xn)	8	2	**10**(1/0/0)	**10**(1/2/0)
FULL FORMAT EXTENSION WORD(S)				
#⟨data⟩.W,(d_{16},An) or (d_{16},PC)	4	0	**8**(1/0/0)	**9**(1/2/0)
#⟨data⟩.L,(d_{16},An) or (d_{16},PC)	6	0	**10**(1/0/0)	**11**(1/2/0)
#⟨data⟩.W,(d_{16},An,Xn) or (d_{16},PC,Xn)	6	0	**8**(1/0/0)	**9**(1/2/0)
#⟨data⟩.L,(d_{16},An,Xn) or (d_{16},PC,Xn)	8	0	**10**(1/0/0)	**11**(1/2/0)
#⟨data⟩.W,([d_{16},An]) or ([d_{16},PC])	4	0	**12**(2/0/0)	**12**(2/2/0)
#⟨data⟩.L,([d_{16},An]) or ([d_{16},PC])	6	0	**14**(2/0/0)	**14**(2/2/0)
#⟨data⟩.W,([d_{16},An],Xn) or ([d_{16},PC],Xn)	4	0	**12**(2/0/0)	**12**(2/2/0)
#⟨data⟩.L,([d_{16},An],Xn) or ([d_{16},PC],Xn)	6	0	**14**(2/0/0)	**14**(2/2/0)
#⟨data⟩.W,([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	4	0	**14**(2/0/0)	**15**(2/2/0)
#⟨data⟩.L,([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	6	0	**16**(2/0/0)	**17**(2/3/0)
#⟨data⟩.W,([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	4	0	**14**(2/0/0)	**15**(2/2/0)
#⟨data⟩.L,([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	6	0	**16**(2/0/0)	**17**(2/3/0)
#⟨data⟩.W,([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	4	0	**14**(2/0/0)	**16**(2/3/0)
#⟨data⟩.L,([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	6	0	**16**(2/0/0)	**18**(2/3/0)
#⟨data⟩.W,([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	4	0	**14**(2/0/0)	**16**(2/3/0)

11.6.2 Fetch Immediate Effective Address (fiea) (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXTENSION WORD(S) (CONTINUED)				
#⟨data⟩.L,([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	6	0	**16**(2/0/0)	**18**(2/3/0)
#⟨data⟩.W,(B)	6	0	**8**(1/0/0)	**9**(1/1/0)
#⟨data⟩.L,(B)	8	0	**10**(1/0/0)	**11**(1/2/0)
#⟨data⟩.W,(d_{16},B)	6	0	**10**(1/0/0)	**12**(1/2/0)
#⟨data⟩.L,(d_{16},B)	8	0	**12**(1/0/0)	**14**(1/2/0)
#⟨data⟩.W,(d_{32},B)	10	0	**14**(1/0/0)	**16**(1/2/0)
#⟨data⟩.L,(d_{32},B)	12	0	**16**(1/0/0)	**18**(1/3/0)
#⟨data⟩.W,([B])	6	0	**12**(2/0/0)	**12**(2/1/0)
#⟨data⟩.L,([B])	8	0	**14**(2/0/0)	**14**(2/2/0)
#⟨data⟩.W,([B],I)	6	0	**12**(2/0/0)	**12**(2/1/0)
#⟨data⟩.L,([B],I)	8	0	**14**(2/0/0)	**14**(2/2/0)
#⟨data⟩.W,([B],d_{16})	6	0	**14**(2/0/0)	**15**(2/2/0)
#⟨data⟩.L,([B],d_{16})	8	0	**16**(2/0/0)	**17**(2/2/0)
#⟨data⟩.W,([B],I,d_{16})	6	0	**14**(2/0/0)	**15**(2/2/0)
#⟨data⟩.L,([B],I,d_{16})	8	0	**16**(2/0/0)	**17**(2/2/0)
#⟨data⟩.W,([B],d_{32})	6	0	**14**(2/0/0)	**16**(2/2/0)
#⟨data⟩.L,([B],d_{32})	8	0	**16**(2/0/0)	**18**(2/3/0)
#⟨data⟩.W,([B],I,d_{32})	6	0	**14**(2/0/0)	**16**(2/2/0)
#⟨data⟩.L,([B],I,d_{32})	8	0	**16**(2/0/0)	**18**(2/3/0)
#⟨data⟩.W,([d_{16},B])	6	0	**14**(2/0/0)	**15**(2/2/0)
#⟨data⟩.L,([d_{16},B])	8	0	**16**(2/0/0)	**17**(2/2/0)
#⟨data⟩.W,([d_{16},B],I)	6	0	**14**(2/0/0)	**15**(2/2/0)
#⟨data⟩.L,([d_{16},B],I)	8	0	**16**(2/0/0)	**17**(2/2/0)
#⟨data⟩.W,([d_{16},B],d_{16})	6	0	**16**(2/0/0)	**18**(2/2/0)
#⟨data⟩.L,([d_{16},B],d_{16})	8	0	**18**(2/0/0)	**20**(2/3/0)
#⟨data⟩.W,([d_{16},B],I,d_{16})	6	0	**16**(2/0/0)	**18**(2/2/0)
#⟨data⟩.L,([d_{16},B],I,d_{16})	8	0	**18**(2/0/0)	**20**(2/3/0)
#⟨data⟩.W,([d_{16},B],d_{32})	6	0	**16**(2/0/0)	**19**(2/3/0)
#⟨data⟩.L,([d_{16},B],d_{32})	8	0	**18**(2/0/0)	**21**(2/3/0)
#⟨data⟩.W,([d_{16},B],I,d_{32})	6	0	**16**(2/0/0)	**19**(2/3/0)
#⟨data⟩.L,([d_{16},B],I,d_{32}) #⟨data⟩.W,([d_{32},B])	8 6	0 0	**18**(2/0/0) **18**(2/0/0)	**21**(2/3/0) **19**(2/2/0)
#⟨data⟩.L,([d_{32},B])	8	0	**20**(2/0/0)	**21**(2/3/0)
#⟨data⟩.W,([d_{32},B],I)	6	0	**18**(2/0/0)	**19**(2/2/0)
#⟨data⟩.L,([d_{32},B],I)	8	0	**20**(2/0/0)	**21**(2/3/0)
#⟨data⟩.W,([d_{32},B],d_{16})	6	0	**20**(2/0/0)	**22**(2/3/0)
#⟨data⟩.L,([d_{32},B],d_{16})	8	0	**22**(2/0/0)	**24**(2/3/0)
#⟨data⟩.W,([d_{32},B],I,d_{16})	6	0	**20**(2/0/0)	**22**(2/3/0)

11.6.2 Fetch Immediate Effective Address (fiea) (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXTENSION WORD(S) (CONTINUED)				
#(data).L,([d_{32},B],I,d_{16})	8	0	**22**(2/0/0)	**24**(2/3/0)
#(data).W,([d_{32},B],d_{32})	6	0	**20**(2/0/0)	**23**(2/3/0)
#(data).L,([d_{32},B],d_{32})	8	0	**22**(2/0/0)	**25**(2/4/0)
#(data).W,([d_{32},B],I,d_{32})	6	0	**20**(2/0/0)	**23**(2/3/0)
#(data).L,([d_{32},B],I,d_{32})	8	0	**22**(2/0/0)	**25**(2/4/0)

B = Base Address: 0, An, PC, Xn, An + Xn, PC + Xn. Form does not affect timing.
I = Index: 0, Xn
% = Total head for fetch immediate effective address timing includes the head time for the operation.

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn do not affect timing.

11.6.3 Calculate Effective Address (cea)

The calculate effective address table indicates the number of clock periods needed for the controller to calculate the specified effective address. Fetch time is only included for the first level of indirection on memory indirect addressing modes. The effective addresses are divided by their formats (refer to **2.5 Effective Address Encoding Summary**). For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% Dn	—	—	**0**(0/0/0)	**0**(0/0/0)
% An	—	—	**0**(0/0/0)	**0**(0/0/0)
(An)	2 + op head	0	**2**(0/0/0)	**2**(0/0/0)
(An) +	0	0	**2**(0/0/0)	**2**(0/0/0)
− (An)	2 + op head	0	**2**(0/0/0)	**2**(0/0/0)
(d_{16},An) or (d_{16},PC)	2 + op head	0	**2**(0/0/0)	**2**(0/1/0)
(xxx).W	2 + op head	0	**2**(0/0/0)	**2**(0/1/0)
(xxx).L	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)
BRIEF FORMAT EXTENSION WORD				
(d_8,An,Xn) or (d_8,PC,Xn)	4 + op head	0	**4**(0/0/0)	**4**(0/1/0)

11.6.3 Calculate Effective Address (cea) (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXENSION WORD(S)				
(d_{16},An) or (d_{16},PC)	2	0	**6**(0/0/0)	**6**(0/1/0)
(d_{16},An,Xn) or (d_{16},PC,Xn)	6+op head	0	**6**(0/0/0)	**6**(0/1/0)
([d_{16},An]) or ([d_{16},PC])	2	0	**10**(1/0/0)	**10**(1/1/0)
([d_{16},An],Xn) or ([d_{16},PC],Xn)	2	0	**10**(1/0/0)	**10**(1/1/0)
([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	2	0	**12**(1/0/0)	**13**(1/2/0)
([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	2	0	**12**(1/0/0)	**13**(1/2/0)
([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	2	0	**12**(1/0/0)	**13**(1/2/0)
([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	2	0	**12**(1/0/0)	**13**(1/2/0)
(B)	6+op head	0	**6**(0/0/0)	**6**(0/1/0)
(d_{16},B)	4	0	**8**(0/0/0)	**9**(0/1/0)
(d_{32},B)	4	0	**12**(0/0/0)	**12**(0/2/0)
([B])	4	0	**10**(1/0/0)	**10**(1/1/0)
([B],I)	4	0	**10**(1/0/0)	**10**(1/1/0)
([B],d_{16})	4	0	**12**(1/0/0)	**13**(1/1/0)
([B],I,d_{16})	4	0	**12**(1/0/0)	**13**(1/1/0)
([B],d_{32})	4	0	**12**(1/0/0)	**13**(1/2/0)
([B],I,d_{32})	4	0	**12**(2/0/0)	**13**(1/2/0)
([d_{16},B])	4	0	**12**(1/0/0)	**13**(1/1/0)
([d_{16},B],I)	4	0	**12**(1/0/0)	**13**(1/1/0)
([d_{16},B],d_{16})	4	0	**14**(1/0/0)	**16**(1/2/0)
([d_{16},B],I,d_{16})	4	0	**14**(1/0/0)	**16**(1/2/0)
([d_{16},B],d_{32})	4	0	**14**(1/0/0)	**16**(1/2/0)
([d_{16},B],I,d_{32})	4	0	**14**(1/0/0)	**16**(1/2/0)
([d_{32},B])	4	0	**16**(1/0/0)	**17**(1/2/0)
([d_{32},B],I)	4	0	**16**(1/0/0)	**17**(1/2/0)
([d_{32},B],d_{16})	4	0	**18**(1/0/0)	**20**(1/2/0)
([d_{32},B],I,d_{16})	4	0	**18**(1/0/0)	**20**(1/2/0)
([d_{32},B],d_{32})	4	0	**18**(1/0/0)	**20**(1/3/0)
([d_{32},B],I,d_{32})	4	0	**18**(1/0/0)	**20**(1/3/0)

B= Base address; 0, An, PC, Xn, An+Xn, PC+Xn. Form does not affect timing.
I= Index; 0, Xn
%= No clock cycles incurred by effective address calculation.

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn do not affect timing.

11.6.4 Calculate Immediate Effective Address (ciea)

The calculate immediate effective address table indicates the number of clock periods needed for the controller to fetch the immediate source operand and calculate the specified destination effective address. In the case of two-word instructions, this table indicates the number of clock periods needed for the controller to fetch the second word of the instruction and calculate the specified source operand or single operand. Fetch time is only included for the first level of indirection on memory indirect addressing modes. The effective addresses are divided by their formats (refer to **2.5 Effective Address Encoding Summary**). For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% #(data).W,Dn	2+op head	0	**2**(0/0/0)	**2**(0/1/0)
% #(data).L,Dn	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
% #(data).W,(An)	2+op head	0	**2**(0/0/0)	**2**(0/1/0)
% #(data).L,(An)	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
#(data).W,(An)+	2	0	**4**(0/0/0)	**4**(0/1/0)
#(data).L,(An)+	4	0	**6**(0/0/0)	**6**(0/1/0)
% #(data).W,−(An)	2+op head	0	**2**(0/0/0)	**2**(0/1/0)
% #(data).L,−(An)	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
% #(data).W,(d_{16},An)	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
% #(data).L,(d_{16},An)	6+op head	0	**6**(0/0/0)	**7**(0/2/0)
% #(data).W,$XXX.W	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
% #(data).L,$XXX.W	6+op head	0	**6**(0/0/0)	**6**(0/2/0)
% #(data).W,$XXX.L	6+op head	0	**6**(0/0/0)	**6**(0/2/0)
% #(data).L,$XXX.L	8+op head	0	**8**(0/0/0)	**8**(0/2/0)
BRIEF FORMAT EXTENSION WORD				
% #(data).W,(d_8,An,Xn) or (d_8,PC,Xn)	6+op head	0	**6**(0/0/0)	**6**(0/2/0)
% #(data).L,(d_8,An,Xn) or (d_8,PC,Xn)	8+op head	0	**8**(0/0/0)	**8**(0/2/0)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXTENSION WORD(S)				
#(data).W,(d_{16},An) or (d_{16},PC)	4	0	**8**(0/0/0)	**8**(0/2/0)
#(data).L,(d_{16},An) or (d_{16},PC)	6	0	**10**(0/0/0)	**10**(0/2/0)
% #(data).W,(d_{16},An,Xn) or (d_{16},PC,Xn)	8+op head	0	**8**(0/0/0)	**8**(0/2/0)
% #(data).L,(d_{16},An,Xn) or (d_{16},PC,Xn)	10+op head	0	**10**(0/0/0)	**10**(0/2/0)
#(data).W,([d_{16},An]) or ([d_{16},PC])	4	0	**12**(1/0/0)	**12**(1/2/0)
#(data).L,([d_{16},An]) or ([d_{16},PC])	6	0	**14**(1/0/0)	**14**(1/1/0)
#(data).W,([d_{16},An],Xn) or ([d_{16},PC],Xn)	4	0	**12**(1/0/0)	**12**(1/2/0)
#(data).L,([d_{16},An],Xn) or ([d_{16},PC],Xn)	6	0	**14**(1/0/0)	**14**(1/1/0)
#(data).W,([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	4	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	6	0	**16**(1/0/0)	**17**(1/3/0)
#(data).W,([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	4	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	6	0	**16**(1/0/0)	**17**(1/3/0)
#(data).W,([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	4	0	**14**(1/0/0)	**16**(1/3/0)
#(data).L,([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	6	0	**16**(1/0/0)	**17**(1/3/0)
#(data).W,([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	4	0	**14**(1/0/0)	**15**(1/3/0)
#(data).L,([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	6	0	**16**(1/0/0)	**17**(1/3/0)
% #(data).W,(B)	8+op head	0	**8**(0/0/0)	**8**(0/1/0)
% #(data).L,(B)	10+op head	0	**10**(0/0/0)	**10**(0/2/0)
#(data).W,(d_{16},B)	6	0	**10**(0/0/0)	**11**(0/2/0)
#(data).L,(d_{16},B)	8	0	**12**(0/0/0)	**13**(0/2/0)
#(data).W,(d_{32},B)	6	0	**14**(0/0/0)	**15**(0/2/0)
#(data).L,(d_{32},B)	8	0	**16**(0/0/0)	**17**(0/3/0)
#(data).W,([B])	6	0	**12**(1/0/0)	**12**(1/1/0)
#(data).L,([B])	8	0	**14**(1/0/0)	**14**(1/2/0)
#(data).W,([B],I)	6	0	**12**(1/0/0)	**12**(1/1/0)
#(data).L,([B],I)	8	0	**14**(1/0/0)	**14**(1/2/0)
#(data).W,([B],d_{16})	6	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([B],d_{16})	8	0	**16**(1/0/0)	**17**(1/2/0)
#(data).W,([B],I,d_{16})	6	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([B],I,d_{16})	8	0	**16**(2/0/0)	**17**(1/2/0)
#(data).W,([B],d_{32})	6	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([B],d_{32})	8	0	**16**(1/0/0)	**17**(1/3/0)
#(data).W,([B],I,d_{32})	6	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([B],I,d_{32})	8	0	**16**(1/0/0)	**17**(1/3/0)
#(data).W,([d_{16},B])	6	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([d_{16},B])	8	0	**16**(1/0/0)	**17**(1/2/0)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXTENSION WORD(S) (CONTINUED)				
#(data).W,([d_{16},B],I)	6	0	**14**(1/0/0)	**15**(1/2/0)
#(data).L,([d_{16},B],I)	8	0	**16**(1/0/0)	**17**(1/2/0)
#(data).W,([d_{16},B],d_{16})	6	0	**16**(1/0/0)	**18**(1/2/0)
#(data).L,([d_{16},B],d_{16})	8	0	**18**(1/0/0)	**20**(1/3/0)
#(data).W,([d_{16},B],I,d_{16})	6	0	**16**(1/0/0)	**18**(1/2/0)
#(data).L,([d_{16},B],I,d_{16})	8	0	**18**(1/0/0)	**20**(1/3/0)
#(data).W,([d_{16},B],d_{32})	6	0	**16**(1/0/0)	**18**(1/3/0)
#(data).L,([d_{16},B],d_{32})	8	0	**18**(1/0/0)	**20**(1/3/0)
#(data).W,([d_{16},B],I,d_{32})	6	0	**16**(1/0/0)	**18**(1/3/0)
#(data).L,([d_{16},B],I,d_{32})	8	0	**18**(1/0/0)	**20**(1/3/0)
#(data).W,([d_{32},B])	6	0	**18**(1/0/0)	**19**(1/2/0)
#(data).L,([d_{32},B])	8	0	**20**(1/0/0)	**21**(1/3/0)
#(data).W,([d_{32},B],I)	6	0	**18**(1/0/0)	**19**(1/2/0)
#(data).L,([d_{32},B],I)	8	0	**20**(1/0/0)	**21**(1/3/0)
#(data).W,([d_{32},B],d_{16})	6	0	**20**(1/0/0)	**22**(1/3/0)
#(data).L,([d_{32},B],d_{16})	8	0	**22**(1/0/0)	**24**(1/3/0)
#(data).W,([d_{32},B],I,d_{16})	6	0	**20**(1/0/0)	**22**(1/3/0)
#(data).L,([d_{32},B],I,d_{16})	8	0	**22**(1/0/0)	**24**(1/3/0)
#(data).W,([d_{32},B],d_{32})	6	0	**20**(1/0/0)	**22**(1/3/0)
#(data).L,([d_{32},B],d_{32})	8	0	**22**(1/0/0)	**24**(1/4/0)
#(data).W,([d_{32},B],I,d_{32})	6	0	**20**(1/0/0)	**22**(1/3/0)
#(data).L,([d_{32},B],I,d_{32})	8	0	**22**(1/0/0)	**24**(1/4/0)

B= Base address; 0, An, PC, Xn, An+Xn, PC+Xn. Form does not affect timing.
I= Index; 0, Xn
%= Total head for address timing includes the head time for the operation.

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn do not affect timing.

11.6.5 Jump Effective Address

The jump effective address table indicates the number of clock periods needed for the controller to calculate the specified effective address for the JMP or JSR instructions. Fetch time is only included for the first level of indirection on memory indirect addressing modes. The effective addresses are divided by their formats (refer to **2.5 Effective Address Encoding Summary**). For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
% (An)	2+op head	0	**2**(0/0/0)	**2**(0/0/0)
% (d_{16},An)	4+op head	0	**4**(0/0/0)	**4**(0/0/0)
% (xxx).W	2+op head	0	**2**(0/0/0)	**2**(0/0/0)
% (xxx).L	2+op head	0	**2**(0/0/0)	**2**(0/0/0)
BRIEF FORMAT EXTENSION WORD				
% (d_8,An,Xn) or (d_8,PC,Xn)	6+op head	0	**6**(0/0/0)	**6**(0/0/0)
FULL FORMAT EXTENSION WORD(S)				
(d_{16},An) or (d_{16},PC)	2	0	**6**(0/0/0)	**6**(0/0/0)
% (d_{16},An,Xn) or (d_{16},PC,Xn)	6+op head	0	**6**(0/0/0)	**6**(0/0/0)
([d_{16},An]) or ([d_{16},PC])	2	0	**10**(1/0/0)	**10**(1/1/0)
([d_{16},An],Xn) or ([d_{16},PC],Xn)	2	0	**10**(1/0/0)	**10**(1/1/0)
([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	2	0	**12**(1/0/0)	**12**(1/1/0)
([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	2	0	**12**(1/0/0)	**12**(1/1/0)
([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	2	0	**12**(1/0/0)	**12**(1/1/0)
([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	2	0	**12**(1/0/0)	**12**(1/1/0)
% (B)	6+op head	0	**6**(0/0/0)	**6**(0/0/0)
(d_{16},B)	4	0	**8**(0/0/0)	**9**(0/1/0)
(d_{32},B)	4	0	**12**(0/0/0)	**13**(0/1/0)
([B])	4	0	**10**(1/0/0)	**10**(1/1/0)
([B],I)	4	0	**10**(1/0/0)	**10**(1/1/0)
([B],d_{16})	4	0	**12**(1/0/0)	**12**(1/1/0)
([B],I,d_{16})	4	0	**12**(1/0/0)	**12**(1/1/0)
([B],d_{32})	4	0	**12**(1/0/0)	**12**(1/1/0)
([B],d_{32})	4	0	**12**(1/0/0)	**12**(1/1/0)
([B],I,d_{32})	4	0	**12**(1/0/0)	**12**(1/1/0)
([d_{16},B])	4	0	**12**(1/0/0)	**13**(1/1/0)
([d_{16},B],I)	4	0	**12**(1/0/0)	**13**(1/1/0)
([d_{16},B],d_{16})	4	0	**14**(1/0/0)	**15**(1/1/0)

11.6.5 Jump Effective Address (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXTENSION WORD(S) (CONTINUED)				
([d_{16},B],I,d_{16})	4	0	**14**(1/0/0)	**15**(1/1/0)
([d_{16},B],d_{32})	4	0	**14**(1/0/0)	**15**(1/1/0)
([d_{16},B],I,d_{32})	4	0	**14**(1/0/0)	**15**(1/1/0)
([d_{32},B])	4	0	**16**(1/0/0)	**17**(1/2/0)
([d_{32},B],I)	4	0	**16**(1/0/0)	**17**(1/2/0)
([d_{32},B],d_{16})	4	0	**18**(1/0/0)	**19**(1/2/0)
([d_{32},B],I,d_{16})	4	0	**18**(1/0/0)	**19**(1/2/0)
([d_{32},B],d_{32})	4	0	**18**(1/0/0)	**19**(1/2/0)
([d_{32},B],I,d_{32})	4	0	**18**(1/0/0)	**19**(1/2/0)

B= Base address; 0, An, PC, Xn, An+Xn, PC+Xn. Form does not affect timing.
I= Index; 0, Xn
%= Total head for effective address timing includes the head time for the operation.

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn do not affect timing.

11.6.6 MOVE Instruction

The MOVE instruction timing table indicates the number of clock periods needed for the controller to calculate the destination effective address and perform the MOVE or MOVEA instruction, including the first level of indirection on memory indirect addressing modes. The fetch effective address table is needed on most MOVE operations (source, destination dependent). The destination effective addresses are divided by their formats (refer to **2.5 Effective Address Encoding Summary**). For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

MOVE Source,Destination	Head	Tail	I-Cache Case	No-Cache Case
SINGLE EFFECTIVE ADDRESS INSTRUCTION FORMAT				
MOVE Rn, Dn	2	0	**2**(0/0/0)	**2**(0/1/0)
MOVE Rn, An	2	0	**2**(0/0/0)	**2**(0/1/0)
* MOVE EA,An	0	0	**2**(0/0/0)	**2**(0/1/0)
* MOVE EA,Dn	0	0	**2**(0/0/0)	**2**(0/1/0)
MOVE Rn,(An)	0	1	**3**(0/0/1)	**4**(0/1/1)
* MOVE SOURCE, (An)	2	0	**4**(0/0/1)	**5**(0/1/1)
MOVE Rn,(An)+	0	1	**3**(0/0/1)	**4**(0/1/1)
* MOVE SOURCE, (An)+	2	0	**4**(0/0/1)	**5**(0/1/1)
MOVE Rn,−(An)	0	2	**4**(0/0/1)	**4**(0/1/1)
* MOVE SOURCE, −(An)	2	0	**4**(0/0/1)	**5**(0/1/1)
* MOVE EA, (d_{16},An)	2	0	**4**(0/0/1)	**5**(0/1/1)
* MOVE EA,XXX.W	2	0	**4**(0/0/1)	**5**(0/1/1)
* MOVE EA,XXX.L	0	0	**6**(0/0/1)	**7**(0/2/1)
BRIEF FORMAT EXTENSION WORD				
* MOVE EA, (d_8,An,Xn)	4	0	**6**(0/0/1)	**7**(0/1/1)
FULL FORMAT EXTENSION WORD(S)				
* MOVE EA, (d_{16},An) or (d_{16},PC)	2	0	**8**(0/0/1)	**9**(0/2/1)
* MOVE EA, (d_{16},An,Xn) or (d_{16},PC,Xn)	2	0	**8**(0/0/1)	**9**(0/2/1)
* MOVE EA, ([d_{16},An],Xn) or ([d_{16},PC],Xn)	2	0	**10**(1/0/1)	**11**(1/2/1)
* MOVE EA,([d_{16},An],d_{16}) or ([d_{16},PC],d_{16})	2	0	**12**(1/0/1)	**14**(1/2/1)
* MOVE EA,([d_{16},An],Xn,d_{16}) or ([d_{16},PC],Xn,d_{16})	2	0	**12**(1/0/1)	**14**(1/2/1)
* MOVE EA,([d_{16},An],d_{32}) or ([d_{16},PC],d_{32})	2	0	**14**(1/0/1)	**16**(1/3/1)
* MOVE EA,([d_{16},An],Xn,d_{32}) or ([d_{16},PC],Xn,d_{32})	2	0	**14**(1/0/1)	**16**(1/3/1)
* MOVE EA,(B)	4	0	**8**(0/0/1)	**9**(0/1/1)
* MOVE EA,(d_{16},B)	4	0	**10**(0/0/1)	**12**(0/2/1)
* MOVE EA,(d_{32},B)	4	0	**14**(0/0/1)	**16**(0/2/1)
* MOVE EA,([B])	4	0	**10**(1/0/1)	**11**(1/1/1)
* MOVE EA,([B],I)	4	0	**10**(1/0/1)	**11**(1/1/1)
* MOVE EA,([B],d_{16})	4	0	**12**(1/0/1)	**14**(1/2/1)
* MOVE EA,([B],I,d_{16})	4	0	**12**(1/0/1)	**14**(1/2/1)
* MOVE EA,([B],d_{32})	4	0	**14**(1/0/1)	**16**(1/2/1)
* MOVE EA,([B],I,d_{32})	4	0	**14**(1/0/1)	**16**(1/2/1)

MOVE Source,Destination	Head	Tail	I-Cache Case	No-Cache Case
FULL FORMAT EXTENSION WORD(S) (CONTINUED)				
* MOVE EA,([d_{16},B])	4	0	**12**(1/0/1)	**14**(1/2/1)
* MOVE EA,([d_{16},B],I)	4	0	**12**(1/0/1)	**14**(1/2/1)
* MOVE EA,([d_{16},B],d_{16})	4	0	**14**(1/0/1)	**17**(1/2/1)
* MOVE EA,([d_{16},B],I,d_{16})	4	0	**14**(1/0/1)	**17**(1/2/1)
* MOVE EA,([d_{16},B],d_{32})	4	0	**16**(1/0/1)	**19**(1/3/1)
* MOVE EA,([d_{16},B],I,d_{32})	4	0	**16**(1/0/1)	**19**(1/3/1)
* MOVE EA,([d_{32},B])	4	0	**16**(1/0/1)	**18**(1/2/1)
* MOVE EA,([d_{32},B],I)	4	0	**16**(1/0/1)	**18**(1/2/1)
* MOVE EA,([d_{32},B],d_{16})	4	0	**18**(1/0/1)	**21**(1/3/1)
* MOVE EA,([d_{32},B],I,d_{16})	4	0	**18**(1/0/1)	**21**(1/3/1)
* MOVE EA,([d_{32},B],d_{32})	4	0	**20**(1/0/1)	**23**(1/3/1)
* MOVE EA,([d_{32},B],I,d_{32})	4	0	**20**(1/0/1)	**23**(1/3/1)

* Add Fetch Effective Address Time
Rn Is a Data or Address Register
SOURCE Is Memory or Immediate Data Address Mode
EA Is any Effective Address

11.6.7 Special-Purpose MOVE Instruction

The special-purpose MOVE timing table indicates the number of clock periods needed for the controller to fetch, calculate, and perform the special-purpose MOVE operation on the control registers or specified effective address. Footnotes indicate when to account for the appropriate effective address times. The total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

	Instruction		Head	Tail	I-Cache Case	No-Cache Case
	EXG	Ry,Rx	4	0	**4**(0/0/0)	**4**(0/1/0)
	MOVEC	Cr,Rn	6	0	**6**(0/0/0)	**6**(0/1/0)
	MOVEC	Rn,Cr – A	6	0	**6**(0/0/0)	**6**(0/1/0)
	MOVEC	Rn,Cr – B	4	0	**12**(0/0/0)	**12**(0/1/0)
	MOVE	CCR,Dn	2	0	**4**(0/0/0)	**4**(0/1/0)
*	MOVE	CCR,Mem	2	0	**4**(0/0/1)	**5**(0/1/1)
	MOVE	Dn,CCR	4	0	**4**(0/0/0)	**4**(0/1/0)
*	MOVE	EA,CCR	0	0	**4**(0/0/0)	**4**(0/1/0)
	MOVE	SR,Dn	2	0	**4**(0/0/0)	**4**(0/1/0)
*	MOVE	SR,Mem	2	0	**4**(0/0/1)	**5**(0/1/1)
#	MOVE	EA,SR	0	0	**8**(0/0/0)	**10**(0/2/0)
% +	MOVEM	EA,RL	2	0	**8+4n**(n/0/0)	**8+4n**(n/1/0)
% +	MOVEM	RL,EA	2	0	**4+2n**(0/0/n)	**4+2n**(0/1/n)
	MOVEP.W	Dn,(d_{16},An)	4	0	**10**(0/0/2)	**10**(0/1/2)
	MOVEP.W	(d_{16},An),Dn	2	0	**10**(2/0/0)	**10**(2/1/0)
	MOVEP.L	Dn,(d_{16},An)	4	0	**14**(0/0/4)	**14**(0/1/4)
	MOVEP.L	(d_{16},An),Dn	2	0	**14**(4/0/0)	**14**(4/1/0)
%	MOVES	EA,Rn	3	0	**7**(1/0/0)	**7**(1/1/0)
%	MOVES	Rn,EA	2	1	**5**(0/0/1)	**6**(0/1/1)
	MOVE	USP,An	4	0	**4**(0/0/0)	**4**(0/1/0)
	MOVE	An,USP	4	0	**4**(0/0/0)	**4**(0/1/0)
	SWAP	Dn	4	0	**4**(0/0/0)	**4**(0/1/0)

CR – A Control Registers USP, VBR, CAAR, MSP, and ISP
CR – B Control Registers SFC, DFC, and CACR
n Number of Register to Transfer (n>0)
RL Register List
* Add Calculate Effective Address Time
# Add Fetch Effective Address Time
% Add Calculate Immediate Address Time

\+ MOVEM EA,RL — For n Registers (n > 0) and w Wait States
I-Cache Case Timing = w ≤ 2: (8+4n)
w > 2: (8+4n)+(w – 2)n
Tail = 0 for all Wait States
MOVEM RL,EA — For n Registers (n > 0) and w Wait States
I-Cache Case Timing = w ≤ 2: (4+2n)+(n – 1)w
w > 2: (4+2n)+(n – 1)w+(w – 2)
Tail = w ≤ 2: (n – 1)w
w > 2: (n)w+(n)(w – 2)

11.6.8 Arithmetical/Logical Instructions

The arithmetical/logical operation timing table indicates the number of clock periods needed for the controller to perform the specified arithmetical/logical instruction using the specified addressing mode. Footnotes indicate when to account for the appropriate fetch effective address or fetch immediate effective address times. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

	Instruction		Head	Tail	I-Cache Case	No-Cache Case
	ADD	Rn,Dn	2	0	2(0/0/0)	2(0/1/0)
	ADDA.W	Rn,An	4	0	4(0/0/0)	4(0/1/0)
	ADDA.L	Rn,An	2	0	2(0/0/0)	2(0/1/0)
*	ADD	EA,Dn	0	0	2(0/0/0)	2(0/1/0)
*	ADD.W	EA,An	0	0	4(0/0/0)	4(0/1/0)
*	ADDA.L	EA,An	0	0	2(0/0/0)	2(0/1/0)
*	ADD	Dn,EA	0	1	3(0/0/1)	4(0/1/1)
	AND	Dn,Dn	2	0	2(0/0/0)	2(0/1/0)
*	AND	EA,Dn	0	0	2(0/0/0)	2(0/1/0)
*	AND	Dn,EA	0	1	3(0/0/1)	4(0/1/1)
	EOR	Dn,Dn	2	0	2(0/0/0)	2(0/1/0)
*	EOR	Dn,EA	0	1	3(0/0/1)	4(0/1/1)
	OR	Dn,Dn	2	0	2(0/0/0)	2(0/1/0)
*	OR	EA,Dn	0	0	2(0/0/0)	2(0/1/0)
*	OR	Dn,EA	0	1	3(0/0/1)	4(0/1/1)
	SUB	Rn,Dn	2	0	2(0/0/0)	2(0/1/0)
*	SUB	EA,Dn	0	0	2(0/0/0)	2(0/1/0)
*	SUB	Dn,EA	0	1	3(0/0/1)	4(0/1/1)
	SUBA.W	Rn,An	4	0	4(0/0/0)	4(0/1/0)
	SUBA.L	Rn,An	2	0	2(0/0/0)	2(0/1/0)
*	SUBA.W	EA,An	0	0	4(0/0/0)	4(0/1/0)
*	SUBA.L	EA,An	0	0	2(0/0/0)	2(0/1/0)
	CMP	Rn,Dn	2	0	2(0/0/0)	2(0/1/0)
*	CMP	EA,Dn	0	0	2(0/0/0)	2(0/1/0)
	CMPA	Rn,An	4	0	4(0/0/0)	4(0/1/0)
*	CMPA	EA,An	0	0	4(0/0/0)	4(0/1/0)
** +	CMP2	EA,Rn	2	0	20(1/0/0)	20(1/1/0)
* +	MULS.W	EA,Dn	2	0	28(0/0/0)	28(0/1/0)
** +	MULS.L	EA,Dn	2	0	44(0/0/0)	44(0/1/0)
* +	MULU.W	EA,Dn	2	0	28(0/0/0)	28(0/1/0)

11.6.8 Arithmetical/Logical Instructions (Continued)

Instruction			Head	Tail	I-Cache Case	No-Cache Case
** +	MULU.L	EA,Dn	2	0	**44**(0/0/0)	**44**(0/1/0)
+	DIVS.W	Dn,Dn	2	0	**56**(0/0/0)	**56**(0/1/0)
* +	DIVS.W	EA,Dn	0	0	**56**(0/0/0)	**56**(0/1/0)
** +	DIVS.L	Dn,Dn	6	0	**90**(0/0/0)	**90**(0/1/0)
** +	DIVS.L	EA,Dn	0	0	**90**(0/0/0)	**90**(0/1/0)
+	DIVU.W	Dn,Dn	2	0	**44**(0/0/0)	**44**(0/1/0)
* +	DIVU.W	EA,Dn	0	0	**44**(0/0/0)	**44**(0/1/0)
** +	DIVU.L	Dn,Dn	6	0	**78**(0/0/0)	**78**(0/1/0)
** +	DIVU.L	EA,Dn	0	0	**78**(0/0/0)	**78**(0/1/0)

*Add Fetch Effective Address Time
**Add Fetch Immediate Effective Address Time
+Indicates Maximum Time (Acutal time is data dependent)

11.6.9 Immediate Arithmetical/Logical Instructions

The immediate arithmetical/logical operation timing table indicates the number of clock periods needed for the controller to fetch the source immediate data value and to perform the specified arithmetic/logical operation using the specified destination addressing mode. Footnotes indicate when to account for the appropriate fetch effective or fetch immediate effective address times. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

	Instruction		Head	Tail	I-Cache Case	No-Cache Case
	MOVEQ	#⟨data⟩,Dn	2	0	2(0/0/0)	2(0/1/0)
	ADDQ	#⟨data⟩,Rn	2	0	2(0/0/0)	2(0/1/0)
*	ADDQ	#⟨data⟩,Mem	0	1	3(0/0/1)	4(0/1/1)
	SUBQ	#⟨data⟩,Rn	2	0	2(0/0/0)	2(0/1/0)
*	SUBQ	#⟨data⟩,Mem	0	1	3(0/0/1)	4(0/1/1)
**	ADDI	#⟨data⟩,Dn	2	0	2(0/0/0)	2(0/1/0)
**	ADDI	#⟨data⟩,Mem	0	1	3(0/0/1)	4(0/1/1)
**	ANDI	#⟨data⟩,Dn	2	0	2(0/0/0)	2(0/1/0)
**	ANDI	#⟨data⟩,Mem	0	1	3(0/0/1)	4(0/1/1)
**	EORI	#⟨data⟩,Dn	2	0	2(0/0/0)	2(0/1/0)
**	EORI	#⟨data⟩,Mem	0	1	3(0/0/1)	4(0/1/1)
**	ORI	#⟨data⟩,Dn	2	0	2(0/0/0)	2(0/1/0)
**	ORI	#⟨data⟩,Mem	0	1	3(0/0/1)	4(0/1/1)
**	SUBI	#⟨data⟩,Dn	2	0	2(0/0/0)	2(0/1/0)
**	SUBI	#⟨data⟩,Mem	0	1	3(0/0/1)	4(0/1/1)
**	CMPI	#⟨data⟩,Dn	2	0	2(0/0/0)	2(0/1/0)
**	CMPI	#⟨data⟩,Mem	0	0	2(0/0/0)	2(0/1/0)

*Add Fetch Effective Address Time
**Add Fetch Immediate Effective Address Time

11.6.10 Binary-Coded Decimal and Extended Instructions

The binary-coded decimal and extended instruction table indicates the number of clock periods needed for the controller to perform the specified operation using the given addressing modes. No additional tables are needed to calculate total effective execution time for these instructions. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	I-Cache Case	No-Cache Case
ABCD	Dn,Dn	0	0	**4**(0/0/0)	**4**(0/1/0)
ABCD	−(An),−(An)	2	1	**13**(2/0/1)	**14**(2/1/1)
SBCD	Dn,Dn	0	0	**4**(0/0/0)	**4**(0/1/0)
SBCD	−(An),−(An)	2	1	**13**(2/0/1)	**14**(2/1/1)
ADDX	Dn,Dn	2	0	**2**(0/0/0)	**2**(0/1/0)
ADDX	−(An),−(An)	2	1	**9**(2/0/1)	**10**(2/1/1)
SUBX	Dn,Dn	2	0	**2**(0/0/0)	**2**(0/1/0)
SUBX	−(An),−(An)	2	1	**9**(2/0/1)	**10**(2/1/1)
CMPM	(An)+,(An)+	0	0	**8**(2/0/0)	**8**(2/1/0)
PACK	Dn,Dn,#⟨data⟩	6	0	**6**(0/0/0)	**6**(0/1/0)
PACK	−(An),−(An),#⟨data⟩	2	1	**11**(1/0/1)	**11**(1/1/1)
UNPK	Dn,Dn,#⟨data⟩	8	0	**8**(0/0/0)	**8**(0/1/0)
UNPK	−(An),−(An),#⟨data⟩	2	1	**11**(1/0/1)	**11**(1/1/1)

11.6.11 Single Operand Instructions

The single operand instruction table indicates the number of clock periods needed for the controller to perform the specified operation on the given addressing mode. Footnotes indicate when it is necessary to account for the appropriate effective address time. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

	Instruction		Head	Tail	I-Cache Case	No-Cache Case
	CLR	Dn	2	0	2(0/0/0)	2(0/1/0)
**	CLR	Mem	0	1	3(0/0/1)	4(0/1/1)
	NEG	Dn	2	0	2(0/0/0)	2(0/1/0)
*	NEG	Mem	0	1	3(0/0/1)	4(0/1/1)
	NEGX	Dn	2	0	2(0/0/0)	2(0/1/0)
*	NEGX	Mem	0	1	3(0/0/1)	4(0/1/1)
	NOT	Dn	2	0	2(0/0/0)	2(0/1/0)
*	NOT	Mem	0	1	3(0/0/1)	4(0/1/1)
	EXT	Dn	4	0	4(0/0/0)	4(0/1/0)
	NBCD	Dn	0	0	6(0/0/0)	6(0/1/0)
	NBCD	Mem	0	1	5(0/0/1)	6(0/1/1)
	Scc	Dn	4	0	4(0/0/0)	4(0/1/0)
**	Scc	Mem	0	1	5(0/0/1)	5(0/1/1)
	TAS	Dn	4	0	4(0/0/0)	4(0/1/0)
**	TAS	Mem	3	0	12(1/0/1)	12(1/1/1)
	TST	Dn	0	0	2(0/0/0)	2(0/1/0)
*	TST	Mem	0	0	2(0/0/0)	2(0/1/0)

*Add Fetch Effective Address Time
**Add Calculate Effective Address Time

11.6.12 Shift/Rotate Instructions

The shift/rotate instruction table indicates the number of clock periods needed for the controller to perform the specified operation on the given addressing mode. Footnotes indicate when it is necessary to account for the appropriate effective address time. The number of bits shifted does not affect the execution time, unless noted. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

	Instruction		Head	Tail	I-Cache Case	No-Cache Case
	LSd	#⟨data⟩,Dy	4	0	**4**(0/0/0)	**4**(0/1/0)
%	LSd	Dx,Dy	6	0	**6**(0/0/0)	**6**(0/1/0)
+	LSd	Dx,Dy	8	0	**8**(0/0/0)	**8**(0/1/0)
*	LSd	Mem by 1	0	0	**4**(0/0/1)	**4**(0/1/1)
	ASL	#⟨data⟩,Dy	2	0	**6**(0/0/0)	**6**(0/1/0)
	ASL	Dx,Dy	4	0	**8**(0/0/0)	**8**(0/1/0)
*	ASL	Mem by 1	0	0	**6**(0/0/1)	**6**(0/1/1)
	ASR	#⟨data⟩,Dy	4	0	**4**(0/0/0)	**4**(0/1/0)
%	ASR	Dx,Dy	6	0	**6**(0/0/0)	**6**(0/1/0)
+	ASR	Dx,Dy	10	0	**10**(0/0/0)	**10**(0/1/0)
*	ASR	Mem by 1	0	0	**4**(0/0/1)	**4**(0/1/1)
	ROd	#⟨data⟩,Dy	4	0	**6**(0/0/0)	**6**(0/1/0)
	ROd	Dx,Dy	6	0	**8**(0/0/0)	**8**(0/1/0)
*	ROd	Mem by 1	0	0	**6**(0/0/1)	**6**(0/1/1)
	ROXd	Dn	10	0	**12**(0/0/0)	**12**(0/1/0)
*	ROXd	Mem by 1	0	0	**4**(0/0/0)	**4**(0/1/0)

d Direction of shift/rotate: L or R
* Add Fetch Effective Address Time
% Indicates shift count is less than or equal to the size of data
\+ Indicates shift count is greater than size of data

11.6.13 Bit Manipulation Instructions

The bit manipulation instruction table indicates the number of clock periods needed for the controller to perform the specified bit operation on the given addressing mode. Footnotes indicate when it is necessary to account for the appropriate effective address time. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

	Instruction		Head	Tail	I-Cache Case	No-Cache Case
	BTST	#⟨data⟩,Dn	4	0	4(0/0/0)	4(0/1/0)
	BTST	Dn,Dn	4	0	4(0/0/0)	4(0/1/0)
#	BTST	#⟨data⟩,Mem	0	0	4(0/0/0)	4(0/1/0)
*	BTST	Dn,Mem	0	0	4(0/0/0)	4(0/1/0)
	BCHG	#⟨data⟩,Dn	6	0	6(0/0/0)	6(0/1/0)
	BCHG	Dn,Dn	6	0	6(0/0/0)	6(0/1/0)
#	BCHG	#⟨data⟩,Mem	0	0	6(0/0/1)	6(0/1/1)
*	BCHG	Dn,Mem	0	0	6(0/0/1)	6(0/1/1)
	BCLR	#⟨data⟩,Dn	6	0	6(0/0/0)	6(0/1/0)
	BCLR	Dn,Dn	6	0	6(0/0/0)	6(0/1/0)
#	BCLR	#⟨data⟩,Mem	0	0	6(0/0/1)	6(0/1/1)
*	BCLR	Dn,Mem	0	0	6(0/0/1)	6(0/1/1)
	BSET	#⟨data⟩,Dn	6	0	6(0/0/0)	6(0/1/0)
	BSET	Dn,Dn	6	0	6(0/0/0)	6(0/1/0)
#	BSET	#⟨data⟩,Mem	0	0	6(0/0/1)	6(0/1/1)
*	BSET	Dn,Mem	0	0	6(0/0/1)	6(0/1/1)

*Add Fetch Effective Address Time
#Add Fetch Immediate Effective Address Time

11.6.14 Bit Field Manipulation Instructions

The bit field manipulation instruction table indicates the number of clock periods needed for the controller to perform the specified bit field operation using the given addressing mode. Footnotes indicate when it is necessary to account for the appropriate effective address time. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

	Instruction		Head	Tail	I-Cache Case	No-Cache Case
	BFTST	Dn	8	0	**8**(0/0/0)	**8**(0/1/0)
*	BFTST	Mem (<5 Bytes)	6	0	**10**(1/0/0)	**10**(1/1/0)
*	BFTST	Mem (5 Bytes)	6	0	**14**(2/0/0)	**14**(2/1/0)
	BFCHG	Dn	14	0	**14**(0/0/0)	**14**(0/1/0)
*	BFCHG	Mem (<5 Bytes)	6	0	**14**(1/0/1)	**14**(1/1/1)
*	BFCHG	Mem (5 Bytes)	6	0	**22**(2/0/2)	**22**(2/1/2)
	BFCLR	Dn	14	0	**14**(0/0/0)	**14**(0/1/0)
*	BFCLR	Mem (<5 Bytes)	6	0	**14**(1/0/1)	**14**(1/1/1)
*	BFCLR	Mem (5 Bytes)	6	0	**22**(2/0/2)	**22**(2/1/2)
	BFSET	Dn	14	0	**14**(0/0/0)	**14**(0/1/0)
*	BFSET	Mem (<5 Bytes)	6	0	**14**(1/0/1)	**14**(1/1/1)
*	BFSET	Mem (5 Bytes)	6	0	**22**(2/0/2)	**22**(2/1/2)
	BFEXTS	Dn	10	0	**10**(0/0/0)	**10**(0/1/0)
*	BFEXTS	Mem (<5 Bytes)	6	0	**12**(1/0/0)	**12**(1/1/0)
*	BFEXTS	Mem (5 Bytes)	6	0	**18**(2/0/0)	**18**(2/1/0)
	BFEXTU	Dn	10	0	**10**(0/0/0)	**10**(0/1/0)
*	BFEXTU	Mem (<5 Bytes)	6	0	**12**(1/0/0)	**12**(1/1/0)
*	BFEXTU	Mem (5 Bytes)	6	0	**18**(2/0/0)	**18**(2/1/0)
	BFINS	Dn	12	0	**12**(0/0/0)	**12**(0/1/0)
*	BFINS	Mem (<5 Bytes)	6	0	**12**(1/0/1)	**12**(1/1/1)
*	BFINS	Mem (5 Bytes)	6	0	**18**(2/0/2)	**18**(2/1/2)
	BFFFO	Dn	20	0	**20**(0/0/0)	**20**(0/1/0)
*	BFFFO	Mem (<5 Bytes)	6	0	**22**(1/0/0)	**22**(1/1/0)
*	BFFFO	Mem (5 Bytes)	6	0	**28**(2/0/0)	**28**(2/1/0)

*Add Calculate Immediate Effective Address Time

NOTE: A bit field of 32 bits may span 5 bytes that require two operand cycles to access or may span 4 bytes that require only one operand cycle to access.

11.6.15 Conditional Branch Instructions

The conditional branch instruction table indicates the number of clock periods needed for the controller to perform the specified branch on the given branch size, with complete execution times given. No additional tables are needed to calculate total effective execution time for these instructions. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parenthees. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	I-Cache Case	No-Cache Case
Bcc	(Taken)	6	0	**6**(0/0/0)	**8**(0/2/0)
Bcc.B	(Not Taken)	4	0	**4**(0/0/0)	**4**(0/1/0)
Bcc.W	(Not Taken)	6	0	**6**(0/0/0)	**6**(0/1/0)
Bcc.L	(Not Taken)	6	0	**6**(0/0/0)	**8**(0/2/0)
DBcc	(cc = False, Count Not Expired)	6	0	**6**(0/0/0)	**8**(0/2/0)
DBcc	(cc = False, Count Expired)	10	0	**10**(0/0/0)	**13**(0/3/0)
DBcc	(cc = True)	6	0	**6**(0/0/0)	**8**(0/1/0)

11.6.16 Control Instructions

The control instruction table indicates the number of clock periods needed for the controller to perform the specified operation. Footnotes indicate when it is necessary to account for the appropriate effective address time. For instruction-cache case and for no-cache case, the total number of clock cyclces is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Instruction			Head	Tail	I-Cache Case	No-Cache Case
ANDI to SR			4	0	**12**(0/0/0)	**14**(0/2/0)
EORI to SR			4	0	**12**(0/0/0)	**14**(0/2/0)
ORI to SR			4	0	**12**(0/0/0)	**14**(0/2/0)
ANDI to CCR			4	0	**12**(0/0/0)	**14**(0/2/0)
EORI to CCR			4	0	**12**(0/0/0)	**14**(0/2/0)
ORI to CCR			4	0	**12**(0/0/0)	**14**(0/2/0)
BSR			2	0	**6**(0/0/1)	**9**(0/2/1)
## CAS		(Successful Compare)	1	0	**13**(1/0/1)	**13**(1/1/1)
## CAS		(Unsuccessful Compare)	1	0	**11**(1/0/0)	**11**(1/1/0)
+ CAS2		(Successful Compare)	2	0	**24**(2/0/2)	**26**(2/2/2)
+ CAS2		(Unsuccessful Compare)	2	0	**24**(2/0/0)	**24**(2/2/0)
CHK	Dn,Dn	(No Exception)	8	0	**8**(0/0/0)	**8**(0/1/0)
+ CHK	Dn,Dn	(Exception Taken)	4	0	**28**(1/0/4)	**30**(1/3/4)
* CHK	EA,Dn	(No Exception)	0	0	**8**(0/0/0)	**8**(0/1/0)
* + CHK	EA,Dn	(Exception Taken)	0	0	**28**(1/0/4)	**30**(1/3/4)
# + CHK2	Mem,Rn	(No Exception)	2	0	**18**(1/0/0)	**18**(1/1/0)
# + CHK2	Mem,Rn	(Exception Taken)	2	0	**40**(2/0/4)	**42**(2/3/4)
% JMP			4	0	**4**(0/0/0)	**6**(0/2/0)
% JSR			0	0	**4**(0/0/1)	**7**(0/2/1)
** LEA			2	0	**2**(0/0/0)	**2**(0/1/0)
LINK.W			0	0	**4**(0/0/1)	**5**(0/1/1)
LINK.L			2	0	**6**(0/0/1)	**7**(0/2/1)
NOP			0	0	**2**(0/0/0)	**2**(0/1/0)
** PEA			0	2	**4**(0/0/1)	**4**(0/1/1)
RTD			2	0	**10**(1/0/0)	**12**(1/2/0)
RTR			1	0	**12**(2/0/0)	**14**(2/2/0)
RTS			1	0	**9**(1/0/0)	**11**(1/2/0)
UNLK			0	0	**5**(1/0/0)	**5**(1/1/0)

\+ Indicates Maximum Time
* Add Fetch Effective Address Time
** Add Calculate Effective Address Time
\# Add Fetch Immediate Address Time
Add Calculate Immediate Address Time
% Add Jump Effective Address Time

11.6.17 Exception-Related Instructions and Operations

The exception-related instruction and operation table indicates the number of clock periods needed for the controller to perform the specified exception-related action. No additional tables are needed to calculate total effective execution time for these operations. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data assumes two-clock reads and writes.

Instruction/Operation	Head	Tail	I-Cache Case	No-Cache Case
BKPT	1	0	**9**(1/0/0)	**9**(1/0/0)
Interrupt (I-Stack)	0	0	**23**(2/0/4)	**24**(2/2/4)
Interrupt (M-Stack)	0	0	**33**(2/0/8)	**34**(2/2/8)
RESET Instruction	0	0	**518**(0/0/0)	**518**(0/1/0)
STOP	0	0	**8**(0/0/0)	**8**(0/2/0)
TRACE	0	0	**22**(1/0/5)	**24**(1/2/5)
TRAP #n	0	0	**18**(1/0/4)	**20**(1/2/4)
Illegal Instruction	0	0	**18**(1/0/4)	**20**(1/2/4)
A-Line Trap	0	0	**18**(1/0/4)	**20**(1/2/4)
F-Line Trap	0	0	**18**(1/0/4)	**20**(1/2/4)
Privilege Violation	0	0	**18**(1/0/4)	**20**(1/2/4)
TRAPcc (Trap)	2	0	**22**(1/0/5)	**24**(1/2/5)
TRAPcc (No Trap)	4	0	**4**(0/0/0)	**4**(0/1/0)
TRAPcc.W (Trap)	5	0	**24**(1/0/5)	**26**(1/3/5)
TRAPcc.W (No Trap)	6	0	**6**(0/0/0)	**6**(0/1/0)
TRAPcc.L (Trap)	6	0	**26**(1/0/5)	**28**(1/3/5)
TRAPcc.L (No Trap)	8	0	**8**(0/0/0)	**8**(0/2/0)
TRAPV (Trap)	2	0	**22**(1/0/5)	**24**(1/2/5)
TRAPV (No Trap)	4	0	**4**(0/0/0)	**4**(0/1/0)

11.6.18 Save and Restore Operations

The save and restore operation table indicates the number of clock periods needed for the controller to perform the specified state save or to return from exception, with complete execution times and stack length given. No additional tables are needed to calculate total effective execution time for these operations. For instruction-cache case and for no-cache case, the total number of clock cycles is outside the parentheses. The number of read, prefetch, and write cycles is given inside the parentheses as (r/p/w). The read, prefetch, and write cycles are included in the total clock cycle number.

All timing data ssumes two-clock reads and writes.

Operation	Head	Tail	I-Cache Case	No-Cache Case
Bus Cycle Fault (Short)	0	0	**36**(1/0/10)	**38**(1/2/10)
Bus Cycle Fault (Long)	0	0	**62**(1/0/24)	**64**(1/2/24)
RTE (Normal Four Word)	1	0	**18**(4/0/0)	**20**(4/2/0)
RTE (Six Word)	1	0	**18**(4/0/0)	**20**(4/2/0)
RTE (Throwaway)	1	0	**12**(4/0/0)	**12**(4/0/0)
RTE (Coprocessor)	1	0	**26**(7/0/0)	**26**(7/2/0)
RTE (Short Fault)	1	0	**36**(10/0/0)	**26**(10/2/0)
RTE (Long Fault)	1	0	**76**(25/0/0)	**76**(25/2/0)

11.6.19 ACU Effective Address Calculation

The calculate effective address table for ACU instructions lists the number of clock periods needed for the controller to calculate various effective addresses. Fetch time is only included for the first level of indirection on memory indirect addressing modes. The total number of clock cycles is outside the parentheses. This total includes the number of read, prefetch, and write cycles, which are shown inside the parentheses as (r/pr/w).

11.6.19 ACU Effective Address Calculation (Continued)

Address Mode	Head	Tail	I-Cache Case	No-Cache Case
(An)	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
(d_{16},An)	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
(xxx).W	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
(xxx).L	6+op head	0	**6**(0/0/0)	**6**(0/2/0)
(d_8,An,Xn)	4+op head	0	**4**(0/0/0)	**4**(0/1/0)
FULL FORMAT EXTENSION WORD(S)				
(d_{16},An)	4	0	**8**(0/0/0)	**8**(0/2/0)
(d_{16},An,Xn)	4	0	**8**(0/0/0)	**8**(0/2/0)
([d_{16},An])	4	0	**12**(1/0/0)	**12**(1/2/0)
([d_{16},An],Xn)	4	0	**12**(1/0/0)	**12**(1/2/0)
([d_{16},An],d_{16})	2	0	**12**(1/0/0)	**12**(1/2/0)
([d_{16},An],Xn,d_{16})	4	0	**12**(1/0/0)	**12**(1/2/0)
([d_{16},An],d_{32})	4	0	**14**(1/0/0)	**14**(1/3/0)
([d_{16},An],Xn,d_{32})	4	0	**14**(1/0/0)	**14**(1/3/0)
(B)	8+op head	0	**8**(0/0/0)	**8**(0/1/0)
(d_{16},B)	6	0	**10**(0/0/0)	**10**(0/2/0)
(d_{32},B)	6	0	**16**(0/0/0)	**16**(0/2/0)
([B])	6	0	**12**(1/0/0)	**12**(1/1/0)
([B],I)	6	0	**12**(1/0/0)	**12**(1/1/0)
([B],d_{16})	6	0	**12**(1/0/0)	**12**(1/2/0)
([B],I,d_{16})	6	0	**12**(1/0/0)	**12**(1/2/0)
([B],d_{32})	6	0	**14**(1/0/0)	**14**(1/2/0)
([B],I,d_{32})	6	0	**14**(1/0/0)	**14**(1/2/0)
([d_{16},B])	6	0	**14**(1/0/0)	**14**(1/2/0)
([d_{16},B],I)	6	0	**14**(1/0/0)	**14**(1/2/0)
([d_{16},B],d_{16})	6	0	**14**(1/0/0)	**14**(1/2/0)
([d_{16},B],I,d_{16})	6	0	**14**(1/0/0)	**14**(1/2/0)
([d_{16},B],d_{32})	6	0	**16**(1/0/0)	**16**(1/3/0)
([d_{16},B],I,d_{32})	6	0	**16**(1/0/0)	**16**(1/3/0)
([d_{32},B])	6	0	**20**(1/0/0)	**20**(1/2/0)
([d_{32},B],I)	6	0	**20**(1/0/0)	**20**(1/2/0)
([d_{32},B],d_{16})	6	0	**20**(1/0/0)	**20**(1/3/0)
([d_{32},B],I,d_{16})	6	0	**20**(1/0/0)	**20**(1/3/0)
([d_{32},B],d_{32})	6	0	**22**(1/0/0)	**22**(1/3/0)
([d_{32},B],I,d_{32})	6	0	**22**(1/0/0)	**22**(1/3/0)

B= Base address; O, An, Xn, An+Xn. Form does not affect timing.
I= Index; O, Xn
*No separation on effective address and operation in timing. Head and tail are the operation's.

NOTE: Xn cannot be in B and I at the same time. Scaling and size of Xn do not affect timing.

11.6.20 ACU Instruction Timing

The ACU instruction timing table lists the numbers of clock periods needed for the ACU to perform the ACU instructions. The total number of clock cycles is outside the parentheses. It includes the numbers of read, prefetch, and write cycles, which are shown inside the parentheses as (r/pr/w).

Instruction	Head	Tail	I-Cache Case	No-Cache Case
PMOVE (from AC0, AC1)*	0	0	**8**(0/0/1)	**8**(0/1/1)
PMOVE (to AC0, AC1)*	0	0	**12**(1/0/0)	**14**(1/2/0)
PMOVE (from ACUSR)*	2	0	**4**(0/0/1)	**5**(0/1/1)
PMOVE (to ACUSR)*	0	0	**6**(1/0/0)	**6**(1/1/0)
PTEST[R:W] ⟨fc⟩,⟨ea⟩,#0*	0	0	**22**(0/0/0)	**22**(0/1/0)

*Add the appropriate effective address calculation time.

11.7 INTERRUPT LATENCY

In real-time systems, the response time required for a controller to service an interrupt is a very important factor pertaining to overall system performance. Processors in the M68000 Family support asynchronous assertion of interrupts and begin processing them on subsequent instruction boundaries. The average interrupt latency is quite short, but the maximum latency is often critical because real-time interrupts cannot require servicing in less than the maximum interrupt latency. The maximum interrupt latency for the MC68EC030 alone is approximately 200 clock cycles (for the MOVEM.L ([d_{32},An],Xn,d_{32}), D0–D7/A0–A7 instruction where the last data fetch is aborted with a bus error). This maximum interrupt latency can be greatly reduced by careful selection of instructions and addressing modes.

11

11.8 BUS ARBITRATION LATENCY

The MC68EC003 does not relinquish the external bus while it is performing a read-modify-write operation. A bus arbitration delay occurs when a coprocessor or other device delays or fails to assert $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ signals to terminate a bus cycle. The maximum delay in this case is undefined; it depends on the length of the delay in asserting the signals.

SECTION 12
APPLICATIONS INFORMATION

This section provides guidelines for using the MC68EC030. The following paragraph discusses the requirements for adapting the MC68EC030 to MC68030 designs. Next, adapting MC68EC030 to MC68020 designs is presented. Then, this section describes the use of the MC68881 and MC68882 coprocessors with the MC68EC030. The byte select logic is described next, followed by memory interface information. The use of the $\overline{\text{STATUS}}$ and $\overline{\text{REFILL}}$ signals, and power and ground considerations complete the section.

12.1 ADAPTING THE MC68EC030 TO MC68030 DESIGNS

The difference between the MC68EC030 and MC68030 is the streamlining of the MC68EC030 for embedded control application. Embedded control systems that need virtual memory capabilities should use an MC68030 or MC68040 instead. An MC68EC030 is pin compatible with an existing MC68030 system. The $\overline{\text{MMUDIS}}$ pin of the MC68030 is a no connect on the MC68EC030. The user code is identical for the MC68030 and MC68EC030. The supervisor code is identical except portions that deal with the MMU. PFLUSH and PLOAD MMU instructions must be removed from existing supervisor code.

12.2 ADAPTING THE MC68EC030 TO MC68020 DESIGNS

One way to utilize the MC68EC030 is in a system designed for the MC68020. The asynchronous buses of the MC68020 and MC68EC030 are compatible. This section describes configuring an adapter for the MC68EC030 to allow insertion into an existing MC68020-based system. Software and architectural differences between the two controllers are also discussed. The need for an adapter is absolute because the MC68020 and MC68EC030 are NOT pin compatible. Use of the adapter board provides the immediate capability for evaluating the programmer's model and instruction set of the MC68EC030 and for developing software to utilize the additional enhanced features of the MC68EC030. This adapter board also provides a relatively simple method for increasing the performance of an existing MC68020 system by insertion of a more advanced 32-bit controller with an on-chip data cache. Since the adapter board does not support the synchronous bus interface of the

MC68EC030, performance measurements for the MC68EC030 used in this manner are misleading when compared to a system designed specifically for the MC68EC030.

The adapter board plugs into the CPU socket of an MC68020 target system, drawing power, ground, and clock signals through the socket and running bus cycles in a fashion compatible with the MC68EC030. The only support hardware necessary is a single 1K-ohm pullup resistor and two capacitors for decoupling power and ground on the adapter board.

12.2.1 Signal Routing

Figure 12-1 shows the complete schematic for routing the signals of the MC68EC030 to the MC68020 header. All signals common to both controllers are directly routed to the corresponding signal of the other controller. The signals on the MC68EC030 that do not have a compatible signal on the MC68020 are either pulled up or left unconnected:

Pulled Up	**No Connect**
$\overline{\text{STERM}}$	$\overline{\text{STATUS}}$
$\overline{\text{CBACK}}$	$\overline{\text{REFILL}}$
$\overline{\text{CIIN}}$	$\overline{\text{CBREQ}}$
	$\overline{\text{CIOUT}}$

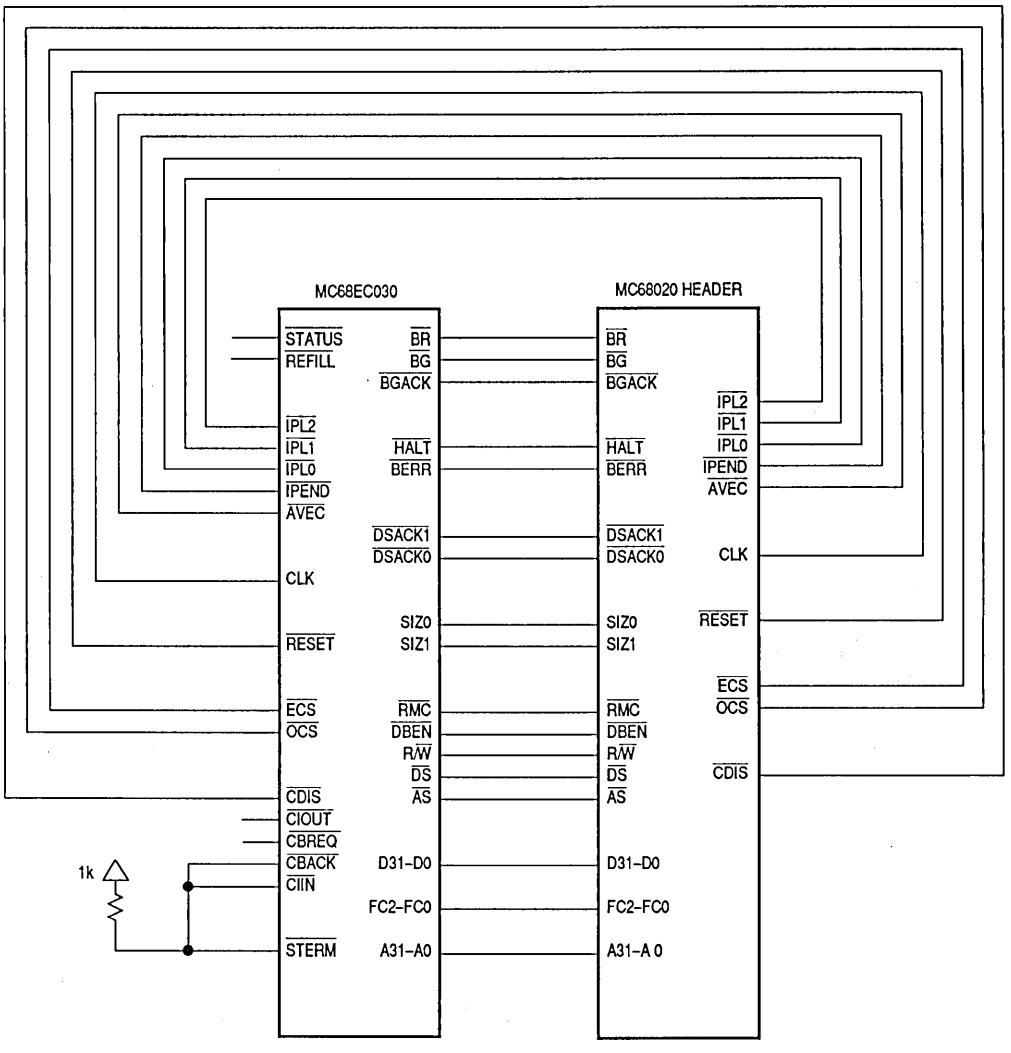


Figure 12-1. Signal Routing for Adapting the MC68EC030 to MC68020 Designs

12

12.2.2 Hardware Differences

Before enabling the on-chip caches of the MC68EC030, an important system feature must be checked. Because of the MC68EC030 cache organization and implementation, cacheable read bus cycles are expected to transfer the entire port width of data (as indicated by the $\overline{\text{DSACK}}$x encoding), regardless of how many bytes are actually requested by the SIZx pins. The MC68020 does not have this requirement, and system memory banks or peripherals may or may

not supply the amount of data required by the MC68EC030. If the target system does not supply the full port width with valid data for any cacheable instruction or data access, the user should either designate that area of memory as noncacheable (with the access control unit (ACU)) or not enable the corresponding on-chip cache(s). In some systems, modifying the target system hardware may also be an option; frequently, the byte select logic is generated by a single programmable array logic (PAL) device, which might easily be replaced or reprogrammed to select all bytes during read cycles from multibyte ports.

The $\overline{\text{HALT}}$ input-only signal of the MC68EC030 is slightly different than the bidirectional $\overline{\text{HALT}}$ signal of the MC68020. However, this difference should not cause any problems beyond eliminating an indication to the external system (e.g., lighting an LED) that the controller has halted due to a double bus fault.

When used in a system originally designed for both an MC68020 and an MC68851, the MC68851 may be left in the system or removed (and replaced with a jumpered header). However, if left in the system, the MC68851 is not accessible to the programmer with the M68000 coprocessor interface. All MMU instructions access the MC68EC030's on-chip ACU. This is true even if the MMU instruction is not supported by the ACU. The benefit in removing the MC68851 is that the minimum asynchronous bus cycle time is reduced from four clock cycles to three. An existing MC68020 system using memory management should upgrade to an MC68030 or MC68040 rather than to an MC68EC030.

If the MC68851 is removed and replaced with a jumpered header, the following MC68851 signals may need special system-specific consideration: $\overline{\text{CLI}}$, $\overline{\text{RMC}}$, $\overline{\text{LBRO}}$, $\overline{\text{LBG}}$, $\overline{\text{LBGACK}}$, and $\overline{\text{LBGI}}$. During translation table searches, the MC68851 asserts the $\overline{\text{CLI}}$ signal but not $\overline{\text{RMC}}$. In simple MC68020/MC68851 systems without logical bus arbitration or logical caches, the MC68851 jumper can have the following signals connected together:

$\overline{\text{LAS}}$ ⇔ $\overline{\text{PAS}}$
$\overline{\text{LBRO}}$ ⇔ $\overline{\text{PBR}}$
$\overline{\text{LBGI}}$ ⇔ $\overline{\text{PBG}}$
$\overline{\text{LBGACK}}$ ⇔ $\overline{\text{PBGACK}}$
LA(8-31) ⇔ PA(8-31)
$\overline{\text{CLI}}$ ⇔ no connect or $\overline{\text{LAS}}$

$\overline{\text{CLI}}$ has two connection options because some systems may use $\overline{\text{CLI}}$ to qualify the occurrence of CPU space cycles since the MC68851 $\overline{\text{PAS}}$ does not assert.

12.2.3 Software Differences

The instruction cache control bits in the cache control register (CACR) of the MC68EC030 are in the identical bit positions as the corresponding bits in the MC68020 CACR. However, the MC68EC030 has additional control bits for burst enable and data cache control. Because this adapter board does not support synchronous bus cycles (and thus burst mode), enabling burst mode through the CACR does not affect system operation in any way. Refer to **SECTION 6 ON-CHIP CACHE MEMORIES** for more information on the bit positions and functions of the CACR bits.

When used in a system originally designed for an MC68020, the programmer must be aware that the MC68EC030 does not support the CALLM and RTM instructions of the MC68020. If code is executed on the MC68EC030 using either the CALLM or RTM instructions, an unimplemented instruction exception is taken. If no ACU software development capability is desired and the cache behavior described under hardware differences is understood, the user may ignore the MC68EC030 ACU.

When the adapter is used in a system originally designed for the MC68020/MC68851 pair, the software differences described below also apply. The MC68EC030 ACU offers a subset of the MC68851 MMU features. The features not supported by the MC68EC030 ACU are as follows:

- Logical address to physical address translation
- On-chip breakpoint registers
- Task aliasing
- Instructions: PBcc, PDBcc, PRESTORE, PSAVE, PScc, PTRAPcc, PVALID, PFLUSH, PMOVE

Only control-alterable addressing modes are allowed for ACU instructions on the MC68EC030.

The MC68EC030 ACU also implements PLOAD and PMOVE differently from the MC68851 MMU.

A feature new to the MC68EC030 ACU (not on the MC68851) is the access control of two address blocks with the access control registers (see **SECTION 9 ACCESS CONTROL UNIT**).

12.3 FLOATING-POINT UNITS

Floating-point support for the MC68EC030 is provided by the MC68881 floating-point coprocessor and the MC68882 enhanced floating-point coprocessor. Both devices offer a full implementation of the IEEE Standard for Binary Floating-Point Arithmetic (754). The MC68882 is a pin and software-compatible upgrade of the MC68881, with an optimized interface to the main controller that provides over 1.5 times the performance of the MC68881 at the same clock frequency.

Both coprocessors provide a logical extension to the integer data processing capabilities of the main controller. They contain a very high-performance floating-point arithmetic unit and a set of floating-point data registers that are utilized in a manner that is analagous to the use of the integer data registers of the controller. The MC68881/MC68882 instruction set is a natural extension of all earlier members of the M68000 Family and supports all addressing modes and data types of the host MC68EC030. The programmer perceives the MC68EC030/coprocessor execution model as if both devices are implemented on one chip. In addition to supporting the full IEEE standard, the MC68881 and MC68882 provide a full set of trigonometric and transcendental functions, on-chip constants, and a full 80-bit extended-precision-real data format.

The interface of the MC68EC030 to the MC68881 or the MC68882 is easily tailored to system cost/performance needs. The MC68EC030 and the MC68881/MC68882 communicate via standard asynchronous M68000 bus cycles. All data transfers are performed by the main controller at the request of the MC68881/MC68882; thus, bus errors, address errors, and bus arbitration function as if the MC68881/MC68882 instructions are executed by the main controller. Up to seven floating-point coprocessors can simultaneously reside in an MC68EC030 system. The MC68881 and the controller may operate at different clock speeds. The MC68882 can only operate at one clock frequency step lower than the MC68EC030. That is, a 40-MHz MC68EC030 and a 33-MHz MC68882 is valid; a 25-MHz or lower MC68882 is invalid.

Figure 12-2 illustrates the coprocessor interface connection of an MC68881/MC68882 to an MC68EC030 (uses entire 32-bit data bus). The MC68881/MC68882 is configured to operate with a 32-bit data bus when both the A0 and SIZx pins are connected to V_{CC}. Refer to MC68881UM/AD, *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, for configuring the MC68881/MC68882 for smaller data bus widths. Note that the MC68EC030 $\overline{\text{CIIN}}$ signal is not used for the coprocessor interface because the MC68EC030 does not cache data obtained during CPU space accesses.

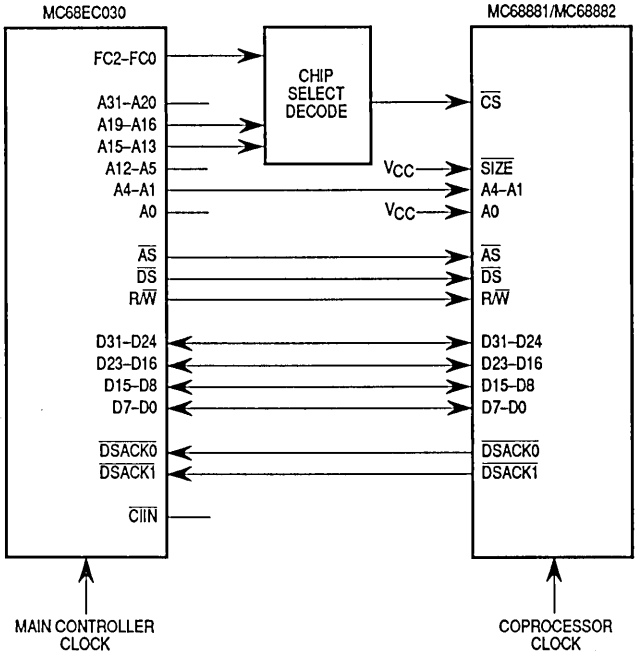


Figure 12-2. 32-Bit Data Bus Coprocessor Connection

The $\overline{CS}$ decode circuitry is asynchronous logic that detects when a particular floating-point coprocessor is addressed. The MC68EC030 signals used by the logic include the function code signals (FC0–FC2), and the address lines (A13–A19). Refer to **SECTION 10 COPROCESSOR INTERFACE DESCRIPTION** for more information about the encoding of these signals. All or a subset of these lines may be decoded, depending on the number of coprocessors in the system.

The major concern of a system designer is to design a $\overline{CS}$ interface that meets the AC electrical specifications for both the MC68EC030 (ECU) and the MC68881/MC68882 (FPCP) without adding unnecessary wait states to FPCP accesses. The following maximum specifications (relative to CLK low) meet these objectives:

t_{CLK} low to AS low(ECU Spec 1ECU Spec 47AFPCP Spec 19) (1)

t_{CLK} low to CS low(ECU Spec 1ECU Spec 47AFPCP Spec 19) (2)

Even though requirement (1) is not met under worst case conditions, if the ECU $\overline{AS}$ is loaded within specifications and the $\overline{AS}$ input to the FPCP is

unbuffered, the requirement is met under typical conditions. Designing the $\overline{CS}$ generation circuit to meet requirement (2) provides the highest probability that accesses to the FPCP occur without unnecessary wait states. A PAL 16L8 (see Figure 12-3) with a maximum propagation delay of 10 ns, programmed according to the equations in Figure 12-4, can be used to generate $\overline{CS}$. For a 25-MHz system, t_{CLK} low to $\overline{CS}$ low is less than or equal to 10 ns when this design is used. Should worst case conditions cause t_{CLK} low to $\overline{AS}$ low to exceed requirement (1), one wait state is inserted in the access to the FPCP; no other adverse effect occurs. Figure 12-5 shows the bus cycle timing for this interface. Refer to MC68881UM/AD, *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, for FPCP specifications.

The circuit that generates $\overline{CS}$ must meet another requirement. When a non-floating-point access immediately follows a floating-point access, $\overline{CS}$ (for the floating-point access) must be negated before $\overline{AS}$ and $\overline{DS}$ (for the subsequent access) are asserted. The PAL circuit previously described also meets this requirement.

For example, if a system has only one coprocessor, the full decoding of the ten signals (FC0–FC2 and A13–A19) provided by the PAL equations in Figure 12-4 is not absolutely necessary. It may be sufficient to use only FC0–FC1 and A16–A17. FC0–FC1 indicate when a bus cycle is operating in either CPU space ($7) or user-defined space ($3), and A16–A17 encode CPU space type as coprocessor space ($2). A13–A15 can be ignored in this case because they encode the coprocessor identification code (CpID) used to differentiate between multiple coprocessors in a system. Motorola assemblers always default to a CpID of $1 for floating-point instructions; this can be controlled with assembler directives if a different CpID is desired or if multiple coprocessors exist in the system.

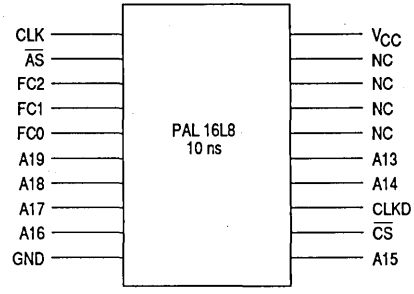


Figure 12-3. Chip Select Generation PAL

```
PAL16I8
FPCP CS GENERATION CIRCUITRY FOR 25 MHz OPERATION
MOTOROLA INC., AUSTIN, TEXAS
CLK    AS     FC2    FC1    FC0    A19    A18    A17    A16    GND
A15    /CS    /CLKD  A14    A13    NC     NC     NC     NC     VCC

CS  = FC2     * FC1    * FC0              ;cpu space = $7
    * /A19    * /A18   * A17    * /A16    ;coprocessor access = $2
    * /A15    * /A14   * A13              ;coprocessor id = $1
    * /CLK                               ;qualified by MPU clock low

    + FC2     * FC1    * FC0              ;cpu space = $7
    * /A19    * /A18   * A17    * /A16    ;coprocessor access = $2
    * /A15    * /A14   * A13              ;coprocessor id = $1
    * /AS                                ;qualified by address strobe low

    + FC2     * FC1    * FC0
    * /A19    * /A18   * A17    * /A16    ;coprocessor access = $2
    * /A15    * /A14   * A13              ;coprocessor id = $1
    * /CLKD                              ;qualified by CLKD (delayed CLK)

CLKD = CLK
```

Description: There are three terms to the CS generation. The first term denotes the earliest time CS can be asserted. The second term is used to assert CS until the end of the FPCP access. The third term is to ensure that no race condition occurs in case of a late AS.

Figure 12-4. PAL Equations

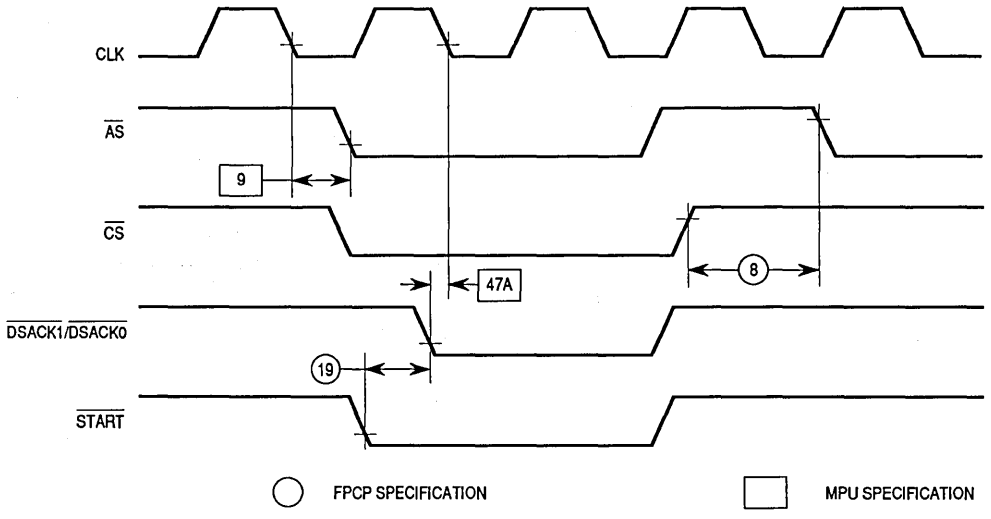


Figure 12-5. Bus Cycle Timing Diagram

12.4 BYTE SELECT LOGIC FOR THE MC68EC030

The architecture of the MC68EC030 allows it to support byte, word, and long-word operand transfers to any 8-, 16-, or 32-bit data port, regardless of alignment. This feature allows the programmer to write code that is not bus-width specific. When accessed, the peripheral or memory subsystem reports its actual port size to the controller, and the MC68EC030 then dynamically sizes the data transfer accordingly, using multiple bus cycles when necessary. The following paragraphs describe the generation of byte select control signals that enable the dynamic bus sizing mechanism, the transfer of differently sized operands, and the transfer of misaligned operands to operate correctly.

The following signals control the MC68EC030 operand transfer mechanism:

- A1, A0 = Address lines. The most significant byte of the operand to be transferred is addressed directly.
- SIZ1, SIZ0 = Transfer size. Output of the MC68EC030. These indicate the number of bytes of an operand remaining to be transferred during a given bus cycle.
- R/W = Read/Write. Output of the MC68EC030. For byte select generation in MC68EC030 systems, R/$\overline{W}$ must be included in the logic if the data from the device is cacheable.
- DSACK1, DSACK0 = Data transfer and size acknowledge. Driven by an asynchronous port to indicate the actual bus width of the port.
- STERM = Synchronous termination. Driven by a 32-bit synchronous port only.

The MC68EC030 assumes that 16-bit ports are situated on data lines D16–D31 and that 8-bit ports are situated on data lines D24–D31. This ensures that the following logic works correctly with the MC68EC030 on-chip internal-to-external data bus multiplexer. Refer to **SECTION 7 BUS OPERATION** for more details on the dynamic bus sizing mechanism.

The need for byte select signals is best illustrated by an example. Consider a long-word write cycle to an odd address in word-organized memory. The transfer requires three bus cycles to complete. The first bus cycle transfers the most significant byte of the long word on D16–D23. The second bus cycle transfers a word on D16–D31, and the last bus cycle transfers the least significant byte of the original long word on D24–D31. To prevent overwriting those bytes that are not used in these transfers, a unique byte data strobe must be generated for each byte when using devices with 16- and 32-bit port widths.

For noncacheable read cycles and all write cycles, the required active bytes of the data bus for any given bus transfer are a function of the size (SIZ0/SIZ1) and lower address (A0/A1) outputs (see Table 12-1). Individual strobes or select signals can be generated by decoding these four signals for every bus cycle. Devices residing on 8-bit ports can utilize $\overline{DS}$ alone since there is only one valid byte for any transfer.

Table 12-1. Data Bus Activity for Byte, Word, and Long-Word Ports

Transfer Size	SIZ1	SIZ0	A1	A0	Data Bus Active Sections Byte (B) – Word (W) – Long-Word (L) Ports			
					D31–D24	D23–D16	D15–D8	D7–D0
Byte	0	1	0	0	B W L	—	—	—
	0	1	0	1	B	W L	—	—
	0	1	1	0	B W	—	L	—
	0	1	1	1	B	W	—	L
Word	1	0	0	0	B W L	W L	—	—
	1	0	0	1	B	W L	L	—
	1	0	1	0	B W	W	L	L
	1	0	1	1	B	W	—	L
Three Byte	1	1	0	0	B W L	W L	L	—
	1	1	0	1	B	W L	L	L
	1	1	1	0	B W	W	L	L
	1	1	1	1	B	W	—	L
Long Word	0	0	0	0	B W L	W L	L	L
	0	0	0	1	B	W L	L	L
	0	0	1	0	B W	W	L	L
	0	0	1	1	B	W	—	L

During cacheable read cycles, the addressed device must provide valid data over its full bus width (as indicated by $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$). While instructions are always prefetched as long-word-aligned accesses, data fetches can occur with any alignment and size. Because the MC68EC030 assumes that the entire data bus port size contains valid data, cacheable data read bus cycles must provide as much data as signaled by the port size during a bus cycle. To satisfy this requirement, the R/$\overline{\text{W}}$ signal must be included in the byte select logic for the MC68EC030.

Figure 12-6 shows a block diagram of an MC68EC030 system with two memory banks. The PAL provides memory-mapped byte select signals for an asynchronous 32-bit port and unmapped byte select signals for other memory banks or ports. Figure 12-7 provides sample equations for the PAL.

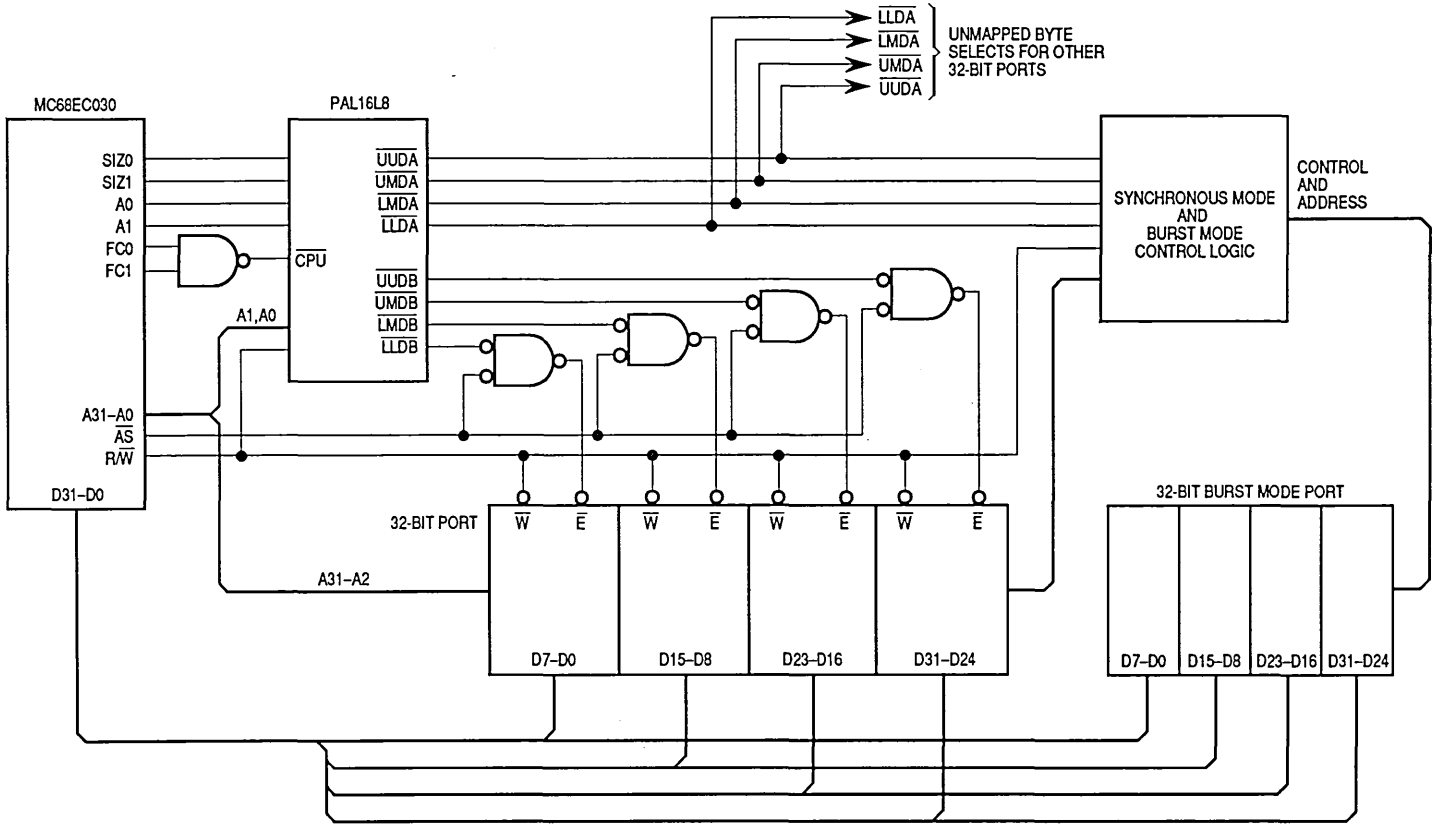


Figure 12-6. Example MC68EC030 Byte Select PAL System Configuration

```
PAL16L8
U1
MC68EC030 BYTE DATA SELECT GENERATION FOR 32-BIT PORTS, MAPPED AND UNMAPPED.
MOTOROLA INC., AUSTIN, TEXAS
A0      A1      SIZ0    SIZ1    RW      A18     A19     A20     A21     GND
/CPU    /UUDA   /UMDA   /LMDA   /LLDA   /UUDA   /UMDB   /LMDB   /LLDB   VCC

UUDA = RW                           ;enable upper byte on read of 32-bit port
     + /A0 * /A1                    ;directly addressed, any size
UMDA = RW                           ;enable upper middle byte on read of 32-bit port
     + A0 * /A1                     ;directly addressed, any size
     + /A1 * /SIZ0                  ;word aligned, size byte or three byte
     + /A1 * SIZ1                   ;word aligned, size is word or long word
LMDA = RW                           ;enable lower middle byte on read of 32-bit port
     + /A0 * A1                     ;directly addressed, any size
     + /A1 * /SIZ0 * /SIZ1          ;word aligned, size is long word
     + /A1 * SIZ0 * SIZ1            ;word aligned, size is three byte
     + /A1 * A0 * /SIZ0             ;word aligned, size is word or long word
LLDA = RW                           ;enable lower byte on read of 32-bit port
     + A0 * A1                      ;directly addressed, any size
     + A0 * SIZ0 * SIZ1             ;odd alignment, three byte size
     + /SIZ0 * /SIZ1                ;size is long word, any address
     + A1 * SIZ1                    ;word aligned, word or three byte size

UUDB = RW * /CPU * (addressb)               ;enable upper byte on read of 32-bit port
     + /A0 * /A1 * /CPU * (addressb)        ;directly addressed, any size
UMDB = RW * /CPU * (addressb)               ;enable upper middle byte on read of 32-bit port
     + A0 * /A1 * /CPU * (addressb)         ;directly addressed, any size
     + /A1 * /SIZ0 * /CPU * (addressb)      ;word aligned, size byte or three byte
     + /A1 * SIZ1 * /CPU * (addressb)       ;word aligned, size is word or long word
LMDB = RW * /CPU * (addressb)               ;enable lower middle byte on read of 32-bit port
     + /A0 * A1 * /CPU * (addressb)         ;directly addressed, any size
     + /A1 * /SIZ0 * /SIZ1 * /CPU * (addressb)  ;word aligned, size is long word
     + /A1 * SIZ0 * SIZ1 * /CPU * (addressb)    ;word aligned, size is three byte
     + /A1 * A0 * /SIZ0 * /CPU * (addressb)     ;word aligned, size is word or long word
LLDB = RW * /CPU * (addressb)               ;enable lower byte on read of 32-bit port
     + A0 * A1 * /CPU * (addressb)          ;directly addressed, any size
     + A0 * SIZ0 * SIZ1 * /CPU * (addressb) ;odd alignment, three byte size
     + /SIZ0 * /SIZ1 * /CPU * (addressb)    ;size is long word, any address
     + A1 * SIZ1 * /CPU * (addressb)        ;word aligned, word or three byte size
```

DESCRIPTION: Byte select signals for writing. On reads, all bytes selects are asserted if the respective memory block is addressed. The input signal /CPU prevents byte select assertion during CPU space cycles and is derived from NANDing FC0-FC1 or FC0-FC2. The label, (addressb), is a designer-selectable combination of address lines used to generate the proper address decode for the system's memory bank. With the address lines given here the decode block size is 256K bytes. A similar address might be included in the equations for UUDA, UMDA, etc. if the designer wishes them to be memory mapped also.

Figure 12-7. MC68EC030 Byte Select PAL Equations

The PAL equations and circuits presented here cannot be the optimal implementation for every system. Depending on the CPU clock frequency, memory access times, and system architecture, different circuits may be required.

12.5 CLOCK DRIVER

The MC68EC030 is designed to sustain high performance while using low-cost (DRAM) memory subsystems. Coupled with the MC88916 clock generation and distribution circuit, the MC68EC030 provides simple interface to lower speed memory subsystems. The MC88916 (see Figure 12-8) generates the clock signals required to efficiently control low-speed memory subsystems, simplifying system design requirements by providing clock generation and distribution.

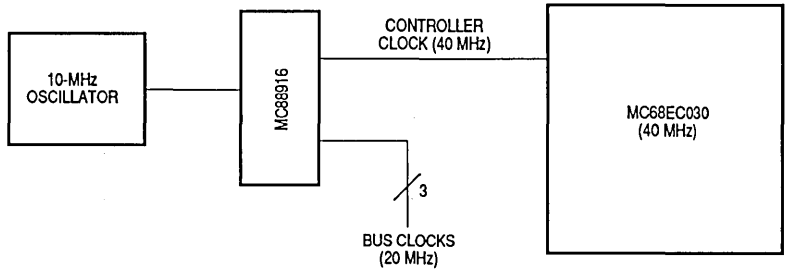


Figure 12-8. Low-Cost DRAM Clock Controller

The MC88916 phase-locked loop clock driver (see Figure 12-9) can also be used to provide clock inputs with greater resolution to a memory controller.

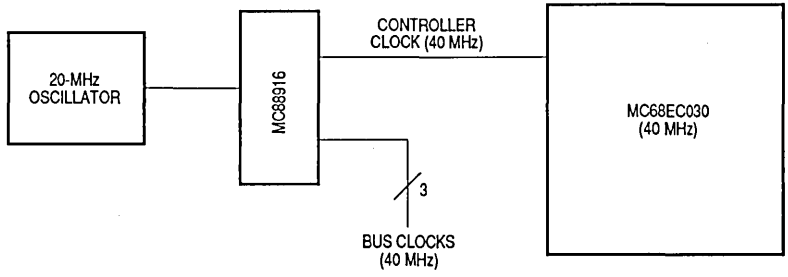


Figure 12-9. High-Resolution DRAM Clock Controller

12.6 MEMORY INTERFACE

The MC68EC030 is capable of running three types of external bus cycles as determined by the cycle termination and handshake signals (refer to **SECTION 7 BUS OPERATION**). The three types of bus cycles are as follows:

1. Asynchronous cycles, terminated by the $\overline{\text{DSACKx}}$ signals, have a minimum duration of three controller clock periods in which up to four bytes are transferred.
2. Synchronous cycles, terminated by the $\overline{\text{STERM}}$ signal, have a minimum duration of two controller clock periods in which up to four bytes are transferred.
3. Burst operation cycles, terminated by the $\overline{\text{STERM}}$ and $\overline{\text{CBACK}}$ signals, have a duration of as little as five controller clock periods in which up to four long words (16 bytes) are transferred.

During read operations, MC68EC030 controllers latch data on the last falling clock edge of the bus cycle, one-half clock before the bus cycle ends (burst mode is a special case). Latching data here, instead of the next rising clock edge, helps to avoid data bus contention with the next bus cycle and allows the MC68EC030 to receive the data into its execution unit sooner for a net performance increase.

Write operations also use this data bus timing to allow data hold times from the negating strobes and to avoid any bus contention with the following bus cycle. This usually allows the system to be designed with a minimum of bus buffers and latches.

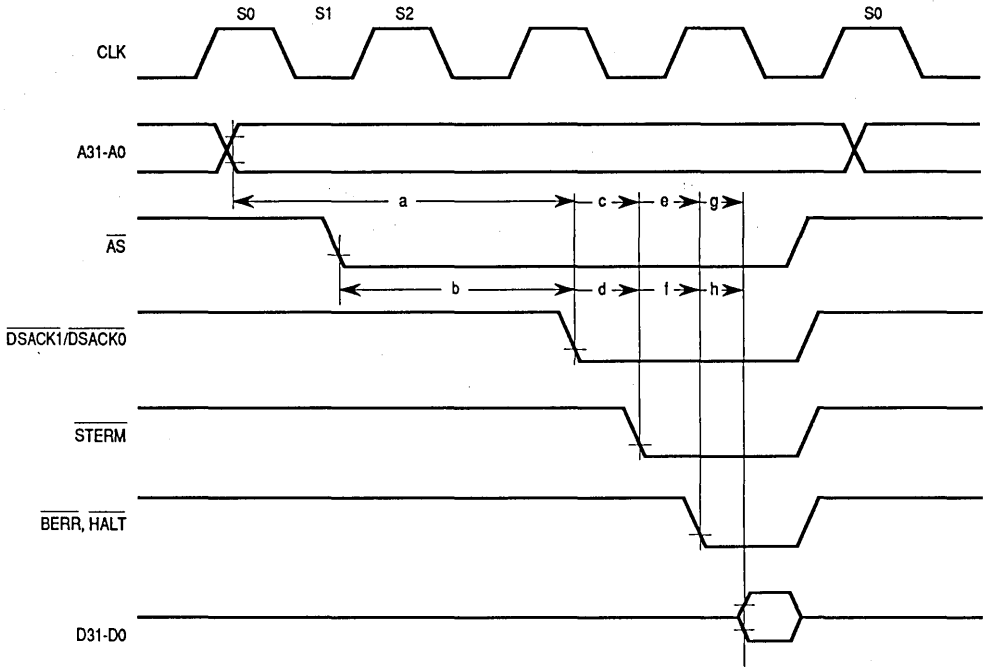
One of the benefits of the MC68EC030 on-chip caches is that the effect of external wait states on performance is lessened because the caches are always accessed in fewer than "no wait states" regardless of the external memory configuration.

12.6.1 Access Time Calculations

The timing paths that are critical in any memory interface are illustrated and defined in Figure 12-10. For burst transfers, the first long word transferred also uses these parameters, but the subsequent transfers are different and are discussed in **12.6.2 Burst Mode Cycles.**

The type of device that is interfaced to the MC68EC030 determines exactly which of the paths is most critical. The address-to-data paths are typically the critical paths for static devices since there is no penalty for initiating a

cycle to these devices and later validating that access with the appropriate bus control signal. Conversely, the address-strobe-to-data-valid path is often most critical for dynamic devices since the cycle must be validated before an access can be initiated. For devices that signal termination of a bus cycle before data is validated (e.g., error detection and correction hardware and some external caches) to improve performance, the critical path may be from the address or strobes to the assertion of $\overline{\text{BERR}}$ (or $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$). Finally, the address-valid-to-$\overline{\text{DSACKx}}$-or-$\overline{\text{STERM}}$-asserted path is most critical for very fast devices and external caches, since the time available between the address becoming valid and the $\overline{\text{DSACKx}}$ or $\overline{\text{STERM}}$ assertion to terminate the bus cycle is minimal. Table 12-2 provides the equations required to calculate the various memory access times assuming a 50-percent duty cycle clock.



NOTE: This diagram illustrates access time calculations only. DSACK1/DSACK0 and STERM should never be asserted together during the same bus cycle.

Parameter	Description	System	Equation
a	Address Valid to DSACKx Asserted	t_{AVDL}	12-1
b	Address Strobe Asserted to DSACKx Asserted	t_{SADL}	12-2
c	Address Valid to STERM Asserted	t_{AVSL}	12-3
d	Address Strobe Asserted to STERM Asserted	t_{SASL}	12-4
e	Address Valid to BERR/HALT Asserted	t_{AVBHL}	12-5
f	Address Strobe Asserted to BERR/HALT Asserted	t_{SABHL}	12-6
g	Address Valid to Data Valid	t_{AVDV}	12-7
h	Address Strobe Asserted to Data Valid	t_{SADV}	12-8

Figure 12-10. Access Time Computation Diagram

Table 12-2. Memory Access Time Equations at 40 MHz

	N=2	N=3	N=4	N=5	N=6
(12-1) $t_{AVDL} = (N-1)•t1 - t2 - t6 - t47A$	—	21.5 ns	46.5 ns	71.5 ns	96.5 ns
(12-2) $t_{SADL} = (N-2)•t1 - t9 - t47A$	—	13 ns	38 ns	63 ns	88 ns
(12-3) $t_{AVSL} = (N-1)•t1 - t6 - t60$	9 ns	34 ns	59 ns	84 ns	109 ns
(12-4) $t_{SASL} = (N-1)•t1 - t3 - t9 - t60$	0.5 ns	25 ns	50.5 ns	75.5 ns	100.5 ns
(12-5) $t_{AVBHL} = N•t1 - t2 - t6 - t27A$	20.5 ns	45.5 ns	70.5 ns	95.5 ns	120.5 ns
(12-6) $t_{SABHL} = (N-1)•t1 - t9 - t27A$	12 ns	37 ns	62 ns	87 ns	112 ns
(12-7) $t_{AVDV} = N•t1 - t2 - t6 - t27$	22.5 ns	47.5 ns	72.5 ns	97.5 ns	122.5 ns
(12-8) $t_{SADV} = (N-1)•t1 - t9 - t27$	14 ns	39 ns	64 ns	89 ns	114 ns

where:

tX = Refers to AC Electrical Specification #X
t1 = The Clock Period
t2 = The Clock Low Time
t3 = The Clock High Time
t6 = The Clock High to Address Valid Time
t9 = The Clock Low to AS Low Delay
t27 = The Data-In to Clock Low Setup Time
t27A = The BERR/HALT to Clock Low Setup Time
t47A = The Asynchronous Input Setup Time
t60 = The Synchronous Input to CLK High Setup Time
N = The Total Number of Clock Periods in the Bus Cycle (Nonburst)
(N≥2 for Synchronous Cycles; N≥3 for Asynchronous Cycles)

During asynchronous bus cycles, DSACK1 and DSACK0 are used to terminate the current bus cycle. In true asynchronous operations, such as accesses to peripherals operating at a different clock frequency, either or both signals may be asserted without regard to the clock, and then data must be valid a certain amount of time later as defined by specification 31. With a 25-MHz controller, this time is 28 ns after DSACKx asserts; with a 40-MHz controller, this time is 14 ns after DSACK asserts (both numbers vary with the actual clock frequency).

However, many local memory systems do not operate in a truly asynchronous manner because the memory control logic can either be related to the MC68EC030 clock or worst case propagation delays are known; thus, asynchronous setup times for the DSACKx signals can be guaranteed. The timing requirements for this pseudo-synchronous DSACKx generation is governed by the equation for tAVDL.

Synchronous cycles use the STERM signal to terminate the current bus cycle. In bus cycles of equal length, STERM has more relaxed timing requirements than DSACKx since an additional time is available when comparing t_{AVSL} (or t_{SASL}) to t_{AVDL} (or t_{SADL}). The only additional restriction is that STERM must meet the setup and hold times as defined by specifications 60 and 61,

respectively, for all rising edges of the clock during a bus cycle. The value for t_{SASL} when the total number of clock periods (N) equals two in Table 12-2 requires further explanation. Because the calculated value of this access time (see Equation 12-4 of Table 12-2) is zero under certain conditions, hardware cannot always qualify $\overline{STERM}$ with $\overline{AS}$ at all frequencies. However, such qualification is not a requirement for the MC68EC030. $\overline{STERM}$ can be generated by the assertion of $\overline{ECS}$, the falling edge of S0, or most simply by the output(s) of an address decode or comparator logic. Note that other devices in the system may require qualification of the access with $\overline{AS}$ since the MC68EC030 has the capability to initiate bus cycles and then abort them before the assertion of $\overline{AS}$.

Another way to optimize the CPU to memory access times in a system is to use a clock frequency less than the rated maximum of the specific MC68EC030 device. Table 12-3 provides calculated t_{AVDV} (see Equation 12-7 of Table 12-2) results for an MC68EC030RP25 and MC68EC030RP40 operating at various clock frequencies. If the system uses other clock frequencies, the above equations can be used to calculate the exact access times.

Table 12-3. Calculated tAVDV Values for Operation at Frequencies Less Than or Equal to the CPU Maximum Frequency Rating

Equation 12-7 t_{AVDV}		MC68EC030RP40		MC68EC030RP25		
Clocks Per Bus Cycle (N) and Type	Wait States	Clock at 33.3 MHz	Clock at 40 MHz	Clock at 16.67 MHz	Clock at 20 MHz	Clock at 25 MHz
2 Clock Synchronous	0	30	22.5	68	53	38
3 Clock Synchronous	1	60	47.5	128	103	78
3 Clock Asynchronous	0	60	47.5	128	103	78
4 Clock Synchronous	2	90	72.5	188	153	118
4 Clock Asynchronous	1	90	72.5	188	153	118
5 Clock Synchronous	3	120	97.5	248	203	158
5 Clock Asynchronous	2	120	97.5	248	203	158
6 Clock Synchronous	4	150	122.5	308	253	198
6 Clock Asynchronous	3	150	122.5	308	253	198

12.6.2 Burst Mode Cycles

The memory access times for burst mode bus cycles follow the above equations for the first access only. For the subsequent (second, third, and fourth) accesses, the memory access time calculations depend on the architecture of the burst mode memory system.

Architectural tradeoffs include the width of the burst memory and the type of memory used. If the memory is 128 bits wide, the subsequent operand accesses do not affect the critical timing paths. For example, if a 3-1-1-1 burst accesses 128-bit-wide memory, the first access is governed by the equations in Table 12-2 for N equal to three. The subsequent accesses also use these values as a base but have additional clock periods. The second access has one additional clock period, the third access has two additional clock periods, and the fourth has three additional clock periods. Thus, the access time for the first cycle determines the critical timing paths.

Memory that is 64 bits wide presents a compromise between the two configurations listed above.

12.7 DEBUGGING AIDS

The MC68EC030 supports the monitoring of internal microsequencer activity with the $\overline{\text{STATUS}}$ and $\overline{\text{REFILL}}$ signals. The use of these signals is described in the following paragraphs. A useful device to aid programming debugging is described in **12.8.2 Real-Time Instruction Trace.**

12.7.1 $\overline{\text{STATUS}}$ and $\overline{\text{REFILL}}$

The MC68EC030 provides the $\overline{\text{STATUS}}$ and $\overline{\text{REFILL}}$ signals to identify internal microsequencer activity associated with the processing of data in the pipeline. Since bus cycles are independently controlled and scheduled by the bus controller, information concerning the processing state of the microsequencer is not available by monitoring bus signals alone. The internal activity identified by the $\overline{\text{STATUS}}$ and $\overline{\text{REFILL}}$ signals include instruction boundaries, some exception conditions, whether the microsequencer has halted, and instruction pipeline refills. $\overline{\text{STATUS}}$ and $\overline{\text{REFILL}}$ track only internal microsequencer activity and are not directly related to bus activity.

As shown in Table 12-4, the number of consecutive clocks during which $\overline{\text{STATUS}}$ is asserted indicates an instruction boundary, an exception to be processed, or that the controller has halted. Note that the controller halted condition is an internal error state in which the microsequencer has shut

down due to a double bus fault and is not related to the external assertion of the $\overline{\text{HALT}}$ input signal. The $\overline{\text{HALT}}$ signal only affects bus operation, not the microsequencer.

Table 12-4. Microsequencer $\overline{\text{STATUS}}$ Indications

Asserted for	Indicates
1 Clock	Sequencer at instruction boundary will begin execution of next instruction
2 Clocks	Sequencer at instruction boundary but will not begin next instruction immediately due to: • pending trace exception OR • pending interrupt exception
3 Clocks	Exception processing to begin for: • reset OR • bus error OR • address error OR • spurious interrupt OR • autovectored interrupt OR • F-line instruction (no coprocessor responded)
Continuously	Processor halted due to double bus fault

The $\overline{\text{REFILL}}$ signal identifies when the microsequencer requests an instruction pipeline refill. Refill requests are a result of having to break sequential instruction execution to handle nonsequential events. Both exceptions and instructions can cause the assertion of $\overline{\text{REFILL}}$. Instructions that cause refills include branches, jumps, instruction traps, returns, coprocessor general instructions that modify the program counter flow, and status register manipulations. Logical and arithmetic operations affecting the condition codes of the status register do not result in a refill request. However, operations like the MOVE <ea>,SR instruction, which updates the status register, cause a refill request since this can change the program space as defined by the function codes. When the program space changes, the controller must fetch data from the new space to replace data already prefetched from the old program space. Similarly, operations that affect the address access control mechanism of the ACU cause a refill request. The test condition, decrement and branch (DBcc) instruction causes two refill requests when the condition being tested is false. To optimize branching performance, the DBcc instruction requests a refill before the condition is tested. If the condition is false, another refill is requested to continue with the next sequential instruction.

Figure 12-11 illustrates the relation between the CLK signal and normal instruction boundaries as identified by the $\overline{\text{STATUS}}$ signal. $\overline{\text{STATUS}}$ asserting for one clock cycle identifies normal instruction boundaries. Note that the

assertion of $\overline{\text{REFILL}}$ does not necessarily correspond to the assertion of $\overline{\text{STATUS}}$. Both $\overline{\text{STATUS}}$ and $\overline{\text{REFILL}}$ assert and negate from the falling edge of the CLK signal.

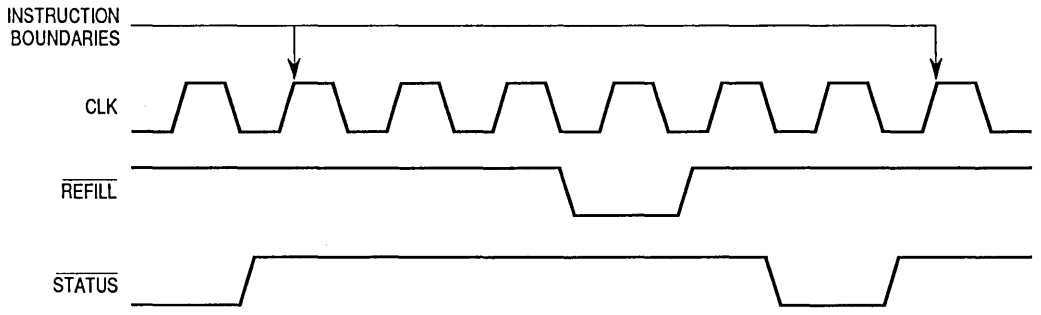


Figure 12-11. Normal Instruction Boundaries

Figure 12-12 shows a normal instruction boundary followed by a trace or interrupt exception boundary. $\overline{\text{STATUS}}$ asserting for two clock cycles identifies a trace or interrupt exception. Instruction boundary information is still present since both trace and interrupt exceptions are processed only at instruction boundaries. Before the exception handler instructions are prefetched, the $\overline{\text{REFILL}}$ signal asserts (not shown) to identify a change in program flow.

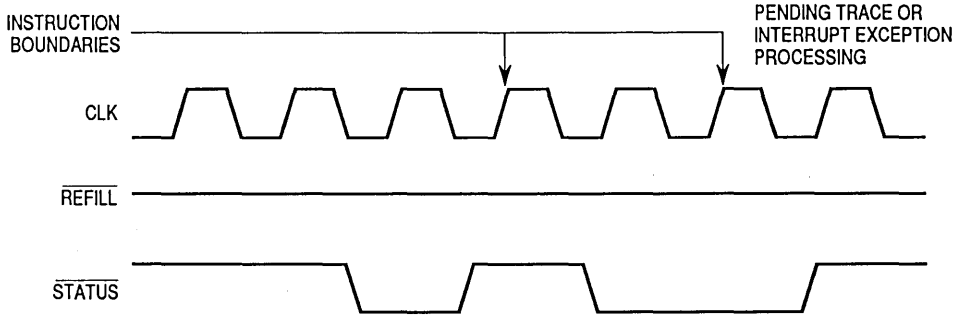


Figure 12-12. Trace or Interrupt Exception

Figure 12-13 illustrates the assertion of the $\overline{\text{STATUS}}$ signal for other exception conditions, which include reset, bus error, address error, spurious interrupt, autovectored interrupt, and F-line instruction when no coprocessor responds. Exception processing causes $\overline{\text{STATUS}}$ to assert for three clock cycles to indicate that normal instruction processing has stopped. Instruction boundaries cannot be determined in this case since these exceptions are processed immediately, not just at instruction boundaries.

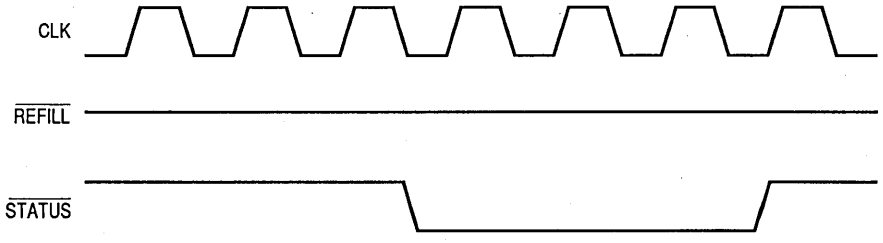


Figure 12-13. Other Exceptions

Figure 12-14 shows the assertion of $\overline{\text{STATUS}}$, indicating that the controller has halted due to a double bus fault. Once a bus error has occurred, any additional bus error exception occurring before the execution of the first instruction of the bus error handler routine constitutes a double bus fault. The controller also halts if it receives a bus error or address error during the vector table read operations or the prefetch for the first instruction after an external reset. $\overline{\text{STATUS}}$ remains asserted until the controller is reset.

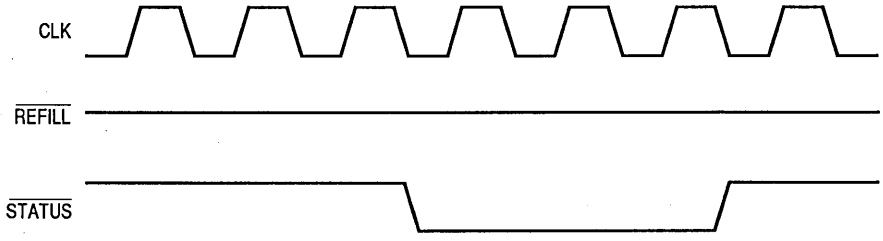


Figure 12-14. Controller Halted

12.7.2 Real-Time Instruction Trace

Microprocessor-based systems used for real-time applications typically lack development aids for program debug. The real-time environment does not allow program instruction execution to arbitrarily stop to handle debugging events. These systems include control applications where mechanical events cannot halt, such as robotics, automotive, and industrial control and emulator systems that may need to keep the target system executing in real time.

To solve the problems inherent with real-time systems, the MC68EC030 incorporates extra hardware-based features to enhance program debug. Real-time systems cannot take advantage of the trace exception mechanism built into all M68000 Family controllers since this takes processing time away from real-time events. Additional output pins have been incorporated into the MC68EC030 to gain real-time visibility into the controller. Tracing capability can be added by decoding MC68EC030 control signals to detect the cycles that are important for tracking. Post analysis of collected data allows for program debug.

Several problems exist with an external trace mechanism. These problems include determining which cycles are important for tracking program flow, detecting if instructions obtained in prefetch operations are discarded by the execution unit, and the inability of external trace circuitry to capture accesses to on-chip cache memories.

External trace hardware used for program debug must be synchronized to the MC68EC030 bus activity. Since all clock cycles are not traced in a program debug environment, the trace hardware requires a sampling signal. For external read and write operations, trace sampling occurs when the data bus contains valid data. Two modes of external bus operation are possible: the synchronous mode in which the system returns the $\overline{\text{STERM}}$ signal and the asynchronous mode in which the system responds with the $\overline{\text{DSACK1}}$ and/or the $\overline{\text{DSACK0}}$ signals. Both modes of bus operation need to generate a sampling signal when valid data is present on the bus. This allows for tracing data flow in and out of the controller, which is the basis for tracking program execution.

The pipelined architecture of the MC68EC030 prefetches instructions and operands to keep the three stages of the instruction pipe full. The pipeline allows concurrent operations to occur for up to three words of a single instruction or for up to three consecutive instructions. While sequential instruction execution is the norm, it is possible that prefetched data is not used by the execution unit due to a nonsequential event. The $\overline{\text{STATUS}}$ signal allows trace hardware to mark the progress of the execution unit as it processes

program memory operands and allows marking of some exceptions. Non-sequential events, where the entire pipeline needs to reload before continuing execution, are marked by the $\overline{\text{REFILL}}$ signal.

External hardware typically has no visibility into on-chip cache memory operations. However, the MC68EC030 provides a local address reference to increase visibility. Write operations are totally visible since the MC68EC030 implements a write-through policy allowing external hardware to capture data. For read operations from on-chip cache memories, the least significant byte of the address bus provides a local address reference.

The MC68EC030 begins an external cycle by driving the address bus and asserting the $\overline{\text{ECS}}$ signal. $\overline{\text{AS}}$ asserts later in the cycle to validate the address. If a hit occurs in the cache or the cache holding register, then the external cycle is aborted and $\overline{\text{AS}}$ is not asserted. In addition, the low-order address bits (A0–A7) are not involved in the address access control process performed by the on-chip ACU, creating a local address reference that can be used by trace functions. All read cycles from the onchip cache memories cannot be captured externally since the cache access does not depend on the availability of the external bus.

Figure 12-15 shows a trace interface circuit that can be used with a logic analyzer for program debug. The nine input signals ($\overline{\text{DSACK1}}$, $\overline{\text{DSACK0}}$, CLK, $\overline{\text{AS}}$, $\overline{\text{RESET}}$, $\overline{\text{STATUS}}$, $\overline{\text{REFILL}}$, $\overline{\text{STERM}}$, and $\overline{\text{ECS}}$) are connected to the MC68EC030 controller in the system under development. Six output signals are generated to aid in capturing and analyzing data. In addition to connecting the logic analyzer to the address bus, the data bus, and the bus control signals, the trace interface signals (SAMPLE, PHALT, FILL, EP, IE, and $\overline{\text{ECSC}}$) should also be connected. The external clock probe of the logic analyzer connects to the system CLK signal for synchronization. Setting up the logic analyzer for data capture requires that samples be taken on the falling edge of the CLK signal when the SAMPLE signal is high. Table 12-5 lists the parts required to implement this circuit.

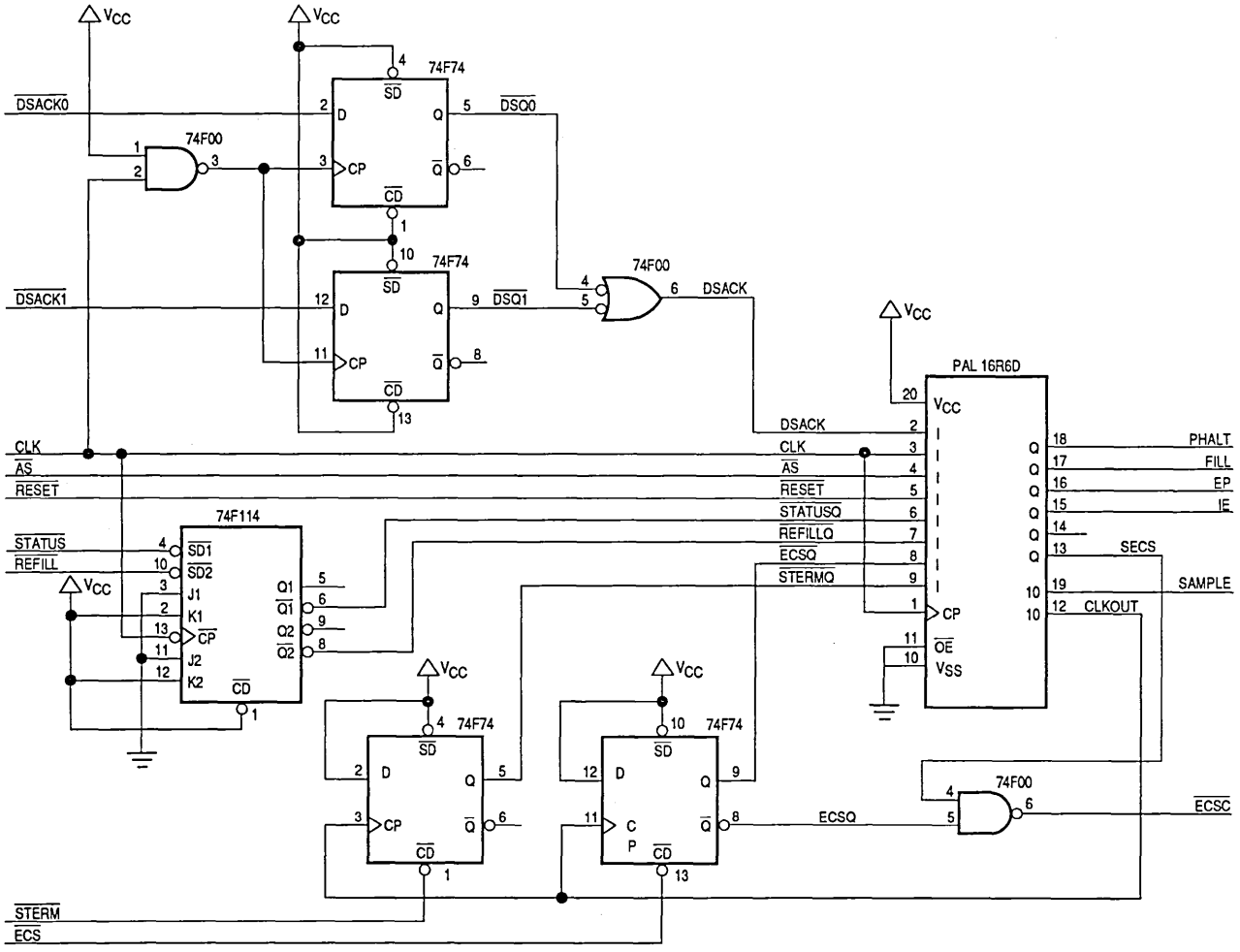


Figure 12-15. Trace Interface Circuit

Table 12-5. Parts List for Trace Interface Circuit

Quantity	Part	Part Description
1	74F00	Quad 2 Input NAND Gate
1	74F114	Dual JK Negative Edge-Triggered Flip-Flop
2	74F74	Dual D-Type Positive Edge-Triggered Flip-Flop
1	PAL16R6D	Programmable Logic Array, Ultra High Speed

SAMPLE is an active-high signal which qualifies the next falling edge of the CLK signal as the sampling point. Five types of conditions cause SAMPLE to assert:

1. An external bus cycle
2. An internal cache hit, including a hit in the cache holding register
3. An instruction boundary
4. Exception processing as marked by the EP signal discussed below
5. The controller halting

The remaining five output signals are used to qualify the information collected.

The PHALT signal indicates that the MC68EC030 has received a double bus fault and needs a reset operation to continue processing. PHALT asserts after the assertion of $\overline{\text{STATUS}}$ for greater than three clock cycles and generates a SAMPLE signal.

The FILL signal indicates a break in sequential instruction execution. FILL is a latched version of the $\overline{\text{REFILL}}$ signal and remains asserted until a sample is collected as indicated by the assertion of SAMPLE. The assertion of FILL does not generate a SAMPLE signal.

The EP signal indicates that the MC68EC030 is beginning exception processing for a reset, bus error, address error, spurious interrupt, autovectored interrupt, F-line instruction exception, trace exception, or interrupt exception. The EP signal asserts after $\overline{\text{STATUS}}$ negates from a two- or three-clock cycle assertion. The assertion of EP does generate a SAMPLE signal.

The IE signal indicates the execution unit has just finished processing an instruction. The IE signal asserts after $\overline{\text{STATUS}}$ negates from a one-clock cycle assertion. The assertion of IE also generates a SAMPLE signal.

The external cycle start condition ($\overline{ECSC}$) signal is used in conjunction with the $\overline{AS}$ signal to determine if the address bus and data bus are valid in the current trace sample. Table 12-6 lists the possible combinations of $\overline{AS}$ and $\overline{ECSC}$ and shows what parts of the traced address and data bus are valid. The assertion of $\overline{ECSC}$ does not generate a SAMPLE signal.

Table 12-6. $\overline{AS}$ and $\overline{ECSC}$ Encoding

$\overline{AS}$	$\overline{ECSC}$	Indicates
0	0	Both Address and Data Bus Are Valid
0	1	Both Address and Data Bus Are Valid
1	0	Address Bits (A0–A7) are Valid Address Bits (A8–A31) Are Invalid Data Bus Is Invalid
1	1	Both Address and Data Bus Are Invalid

Figure 12-16 shows the pin definitions for the PAL16R6 package used in the trace circuit. These definitions are used by the PAL equations listed in Figure 12-17.

12.8 POWER AND GROUND CONSIDERATIONS

The MC68EC030 is fabricated in Motorola's advanced HCMOS process, contains approximately 275,000 total transistor sites, and is capable of operating at clock frequencies of up to 40 MHz. While the use of CMOS for a device containing such a large number of transistors allows significantly reduced power consumption compared to an equivalent NMOS circuit, the high clock speed makes the characteristics of power supplied to the device very important. The power supply must be able to furnish large amounts of instantaneous current when the MC68EC030 performs certain operations, and it must remain within the rated specification at all times. To meet these requirements, more detailed attention must be given to the power supply connection to the MC68EC030 than is required for NMOS devices operating at slower clock rates.

For a solid power supply connection, 10 V_{CC} pins and 14 GND pins are provided. This allows two V_{CC} and four GND pins to supply power for the address bus and two V_{CC} and four GND pins to supply the data bus; the remaining V_{CC} and GND pins are used by the internal logic and clock generation circuitry. Table 12-7 lists the V_{CC} and GND pin assignments.

```
/*******************************************************/
/*  This device generates a sampling signal for tracing processor activity on  *
/*  an instruction level basis for the MC68EC030. In the pin definitions and   *
/*  equations listed below the following symbols are used:                     *
/*                              Symbol   Definition                            *
/*                                !      Logical NOT                          *
/*                                #      Logical OR                           *
/*                                &      Logical AND                          *
/*  In addition, the '.d' extension on signal names refers to the 'D' input of *
/*  the internal PAL flip flop.                                                *
/*******************************************************/
/*  Allowable Target Device Types : PAL16R6D High Speed PAL                    *
/*******************************************************/
/**     Inputs     **/
PIN 1          =  clk          ;    /* same as pin 3 CLK          */
PIN 2          =  DSACK        ;    /* Data Strobe Acknowledge    */
PIN 3          =  CLK          ;    /* MPU Clock Signal           */
PIN 4          =  !AS          ;    /* Address Strobe             */
PIN 5          =  !RESET       ;    /* System Reset Signal        */
PIN 6          =  !STATUSQ     ;    /* Latched STATUS Signal      */
PIN 7          =  !REFILLQ     ;    /* Latched REFILL Signal      */
PIN 8          =  !ECSQ        ;    /* Latched ECS Signal         */
PIN 9          =  !STERMQ      ;    /* Latched STERM Signal       */

/**     Outputs     **/
PIN 19         =  SAMPLE       ;    /* Sample Signal              */
PIN 18         =  PHALT        ;    /* Processor Halted           */
PIN 17         =  FILL         ;    /* REFILL received            */
PIN 16         =  EP           ;    /* Exception Pending          */
PIN 15         =  IE           ;    /* Instruction Executed       */
PIN 14         =  sc           ;    /* status complete            */
PIN 13         =  secs         ;    /* sampled ECS signal         */
PIN 12         =  CLKOUT       ;    /* Delayed CLK Signal         */
```

Figure 12-16 PAL Pin Definition

```
/**  Intermediate Equations    **/                  /*   State  =  PHALT  SC  EP  IE  */
     S0  =  !PHALT  &  !SC  &  !EP  &  !IE;       /*     0    =     0     0   0   0  */
     S1  =  !PHALT  &  !SC  &  !EP  &   IE;       /*     1    =     0     0   0   1  */
     S2  =  !PHALT  &  !SC  &   EP  &   IE;       /*     2    =     0     0   1   1  */
     S3  =  !PHALT  &  !SC  &   EP  &  !IE;       /*     3    =     0     0   1   0  */
     S4  =   PHALT  &   SC  &   EP  &   IE;       /*     4    =     1     1   1   1  */
     S5  =  !PHALT  &   SC  &  !EP  &   IE;       /*     5    =     0     1   0   1  */
     S6  =  !PHALT  &   SC  &   EP  &   IE;       /*     6    =     0     1   1   1  */
     S7  =  !PHALT  &   SC  &   EP  &  !IE;       /*     7    =     0     1   1   0  */

/**     Logic Equations     **/
!SAMPLE      = !SC  &  !AS  &  !SECS  #
                !SC  &  !DSACK  &  !STERMQ  &  !SECS  #
                !SC  &   AS  &  !DSACK  &  !STERMQ  &  SECS;

!PHALT.d     = !STATUSQ  #  !EP  #  IE  #  RESET;

!SC.d        =  RESET  #
                S0  #
                S1  &   STATUSQ  #
                S2  &   STATUSQ  #
                S4  &  !STATUSQ  #
                SC  &  !PHALT;

!EP.d        =  RESET  #
                S0  #
                S1  &  !STATUSQ  #
                S4  &  !STATUSQ  #
                SC  &  !PHALT;

!IE.d        =  RESET  #
                S0  &  !STATUSQ  #
                S2  &   STATUSQ  #
                S3  &  !STATUSQ  #
                SC  &  !STATUSQ;

!SECS.d      = !ECSQ;

!CLKOUT      = !CLK;

!FILL.d      = !REFILLQ  &  SAMPLE  #
                !FILL  &  !REFILLQ  #
                 RESET;
```

Figure 12-17. Logic Equations

Table 12-7. V_{CC} and GND Pin Assignments

Pin Group	V_{CC}	GND
Address Bus	C6, D10	C5, C7, C9, E11
Data Bus	L6, K10	J11, L9, L7, L5
$\overline{ECS}$, SIZx, $\overline{DS}$, $\overline{AS}$, $\overline{DBEN}$, $\overline{CBREQ}$, R/$\overline{W}$	K4	J3
FC0–FC2, $\overline{RMC}$, $\overline{OCS}$, $\overline{CIOUT}$, $\overline{BG}$	D4	E3
Internal Logic, $\overline{RESET}$, $\overline{STATUS}$, $\overline{REFILL}$, Misc.	H3, F2, F11, H11	L8, G3, F3, G11

To reduce the amount of noise in the power supplied to the MC68EC030 and to provide for instantaneous current requirements, common capacitive decoupling techniques should be observed. While there is no recommended layout for this capacitive decoupling, it is essential that the inductance between these devices and the MC68EC030 be minimized to provide sufficiently fast response time to satisfy momentary current demands and to maintain a constant supply voltage. It is suggested that a combination of low, middle, and high-frequency, high-quality capacitors be placed as close to the MC68EC030 as possible (e.g., a set of 10 mF, 0.1 mF, and 330 pF capacitors in parallel provides filtering for most frequencies prevalent in a digital system). Similar decoupling techniques should also be observed for other VLSI devices in the system.

In addition to the capacitive decoupling of the power supply, care must be taken to ensure a low-impedance connection between all MC68EC030 V_{CC} and GND pins and the system power supply planes. Failure to provide connections of sufficient quality between the MC68EC030 power supply pins and the system supplies will result in increased assertion delays for external signals, decreased voltage noise margins, and possible errors in internal logic.

SECTION 13
ELECTRICAL CHARACTERISTICS

The following paragraphs provide information on the maximum rating and thermal characteristics for the MC68EC030. Detailed information on timing specifications for power considerations, DC electrical characteristics, and AC timing specifications can be found in the MC68EC030/D, *MC68EC030 Technical Summary*.

13.1 MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage*	V_{CC}	−0.3 to +7.0	V
Input Voltage	V_{in}	−0.5 to +7.0	V
Operating Temperature Range	T_A	0 to 70	°C
Storage Temperature Range	T_{stg}	−55 to 150	°C

*A continuous clock must be supplied to the MC68EC030 when it is powered up.

This device contains protective circuitry against damage due to high static voltages or electrical fields; however, it is advised that normal precautions be taken to avoid application of any voltages higher than maximum-rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level (e.g., either GND or V_{CC}).

13.2 THERMAL CHARACTERISTICS — PGA PACKAGE

Characteristic	Symbol	Value	Rating
Thermal Resistance — Plastic			°C/W
Junction to Ambient	θ_{JA}	TBD*	
Junction to Case	θ_{JC}	TBD*	

*To Be Determined

13

SECTION 14
ORDERING INFORMATION AND MECHANICAL DATA

This section contains the pin assignments and package dimensions of the MC68EC030. In addition, detailed information is provided to be used as a guide when ordering.

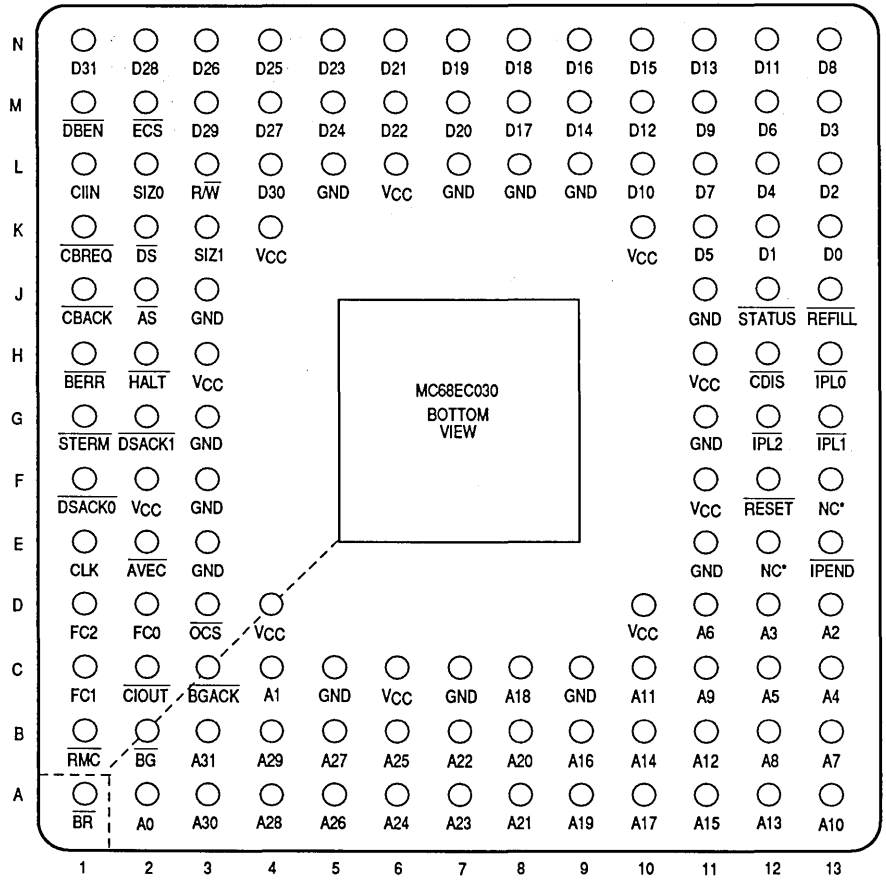
14.1 STANDARD MC68EC030 ORDERING INFORMATION

Package Type	Frequency (MHz)	Temperature	Order Number
Pin Grid Array	25.0	0°C to 70°C	MC68EC030RP25
RP Suffix	40.0	0°C to 70°C	MC68EC030RP40

14.2 PIN ASSIGNMENTS — PIN GRID ARRAY (RP SUFFIX)

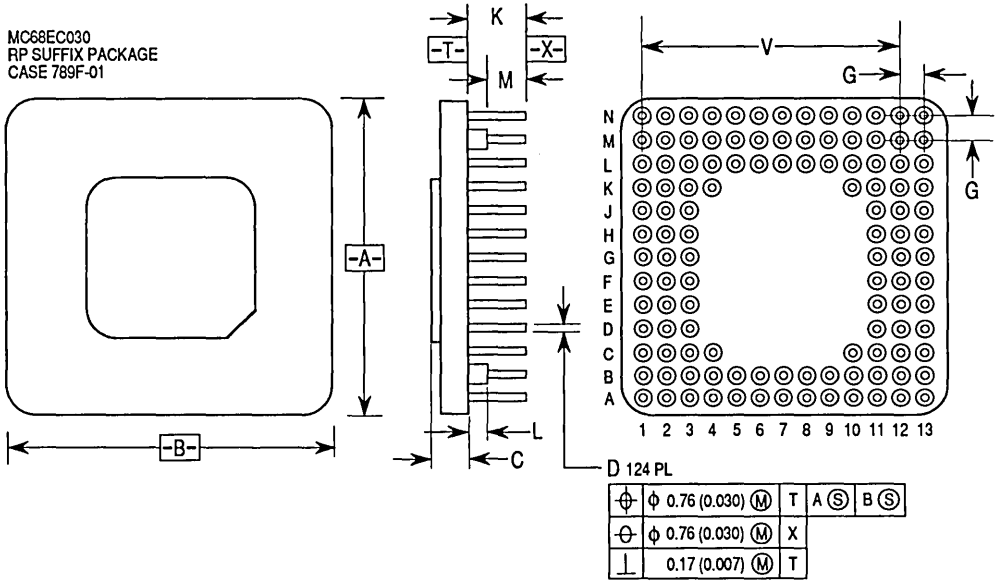
The V_{CC} and GND pins are separated into three groups to provide individual power supply connections for the address bus buffers, data bus buffers, and all other output buffers and internal logic.

Pin Group	V_{CC}	GND
Address Bus	C6, D10	C5, C7, C9, E11
Data Bus	L6, K10	J11, L9, L7, L5
$\overline{ECS}$, SIZx, $\overline{DS}$, $\overline{AS}$, $\overline{DBEN}$, $\overline{CBREQ}$, R/$\overline{W}$	K4	J3
FC0–FC2, $\overline{RMC}$, $\overline{OCS}$, $\overline{CIOUT}$, $\overline{BG}$	D4	E3
Internal Logic, $\overline{RESET}$, $\overline{STATUS}$, $\overline{REFILL}$, Misc.	H3, F2, F11, H11	L8, G3, F3, G11



14.3 PACKAGE DIMENSIONS

MC68EC030
RP SUFFIX PACKAGE
CASE 789F-01



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	34.04	35.05	1.340	1.380
B	34.04	35.05	1.340	1.380
C	2.92	3.18	0.115	0.135
D	0.44	0.55	0.017	0.022
G	2.54 BSC		0.100 BSC	
K	4.32	4.95	0.170	0.195
L	1.02	1.52	0.040	0.060
M	2.79	3.81	0.110	0.150
V	30.48 BSC		1.200 BSC	

NOTES:
1. DIMENSIONING AND TOLERANCING PER Y14.5M,1982.
2. CONTROLLING DIMENSION: INCH.
3. DIMENSION D INCLUDES LEAD FINISH.

PRELIMINARY

14

14

APPENDIX A
MC68EC030 NEW INSTRUCTIONS

This appendix gives details of the new instructions for the MC68EC030 embedded controller. Refer to M68000 PM/AD, *M68000 Programmer's Reference Manual*, for details of the other MC68EC030 instructions.

Operation: If supervisor state
then (Source) ➧ MRn or MRn ➧ (Destination)
else TRAP

Assembler Syntax: PMOVE MRn,⟨ea⟩
PMOVE ⟨ea⟩,MRn

Attributes: Size = (Word, Long, Quad)

Description: Moves the contents of the source effective address to the ACx register or moves the contents of the ACx register to the destination effective address.

The instruction is a long-word operation for the access control registers (AC0 and AC1). It is a word operation for the ACU status register (ACUSR).

Writing to the ACx registers enables or disables the access control register according to the E bit written. If the E bit is set to one, the access control register is enabled. If the E bit is zero, the register is disabled.

Condition Codes:
Not affected.

ACUSR:
Not affected (unless the ACUSR is specified as the destination operand).

Instruction Format (for ACUSR):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	1	1	0	0	0	R/W	0	0	0	0	0	0	0	0	0

Instruction Fields (for ACUSR):
Effective Address field — Specifies the memory location for the transfer.
R/W field — Specifies the direction of transfer:
0 — Memory to ACUSR
1 — ACUSR to memory

A

NOTE

The syntax of assemblers for the MC68851 use the symbol PSR for the ACUSR. The syntax of assemblers for the MC68030 uses the symbols TT0 and TT1 for AC0 and AC1.

Instruction Format (for ACx registers):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	P REG			R/W	0	0	0	0	0	0	0	0	0

Instruction Fields (for ACx registers):

Effective Address field — Specifies the memory location for the transfer.

P Reg field — Specifies the ACx register:

010 — Access control register 0.

011 — Access control register 1.

R/W field — Specifies the direction of transfer:

0 — Memory to ACUSR

1 — ACUSR to memory

A

Operation: If supervisor state
then address status ➧ ACUSR
else TRAP

Assembler Syntax: PTESTR ⟨function code⟩,⟨ea⟩
PTESTW ⟨function code⟩,⟨ea⟩

Attributes: Unsized

Description: This instruction searches the ACx registers for the address descriptor corresponding to the ⟨ea⟩ field and sets the bit of the ACUSR according to the status of the descriptor.

The ⟨function code⟩ operand is specified in one of the following ways:

1. Immediate — Three bits in the command word.
2. Data Register — The three least significant bits of the data register specified in the instruction.
3. Source Function Code Register.
4. Destination Function Code Register.

The effective address is the address to test.

Condition Codes:
Not affected.

ACUSR:

X	X	X	0	X	X	X	0	0	AC	0	0	0	X	X	X

X = May be 0 or 1

The AC bit is set if a match occurred in either (or both) of the ACx registers.

Instruction Field:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
1	0	0	0	0	0	R/W	0	REG					FC		

A

Instruction Fields:

Effective field — Specifies the address to be tested. Only control alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
–(An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#⟨data⟩	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC, Xn],od)	—	—
([bd,PC],Xn,od)	—	—

R/W field — Specifies simulating a read or write bus cycle:
0 — Write
1 — Read

Reg field — Specifies an address register for the instruction. When the A field contains 0, this field must contain 0.

FC field — Function code of address to be tested:
10XXX — Function code is specified as bits XXX.
01DDD — Function code is specified as bits 2:0 of data register DDD.
00000 — Function code is specified as SFC register.
00001 — Function code is specified as DFC register.

NOTE

The syntax for assemblers for the MC68030 is PTESTR ⟨function code⟩,⟨ea⟩,#0 and PTESTW ⟨function code⟩,⟨ea⟩,#0.

A

INDEX

— A —

— B —

— C —

— D —

— E —

— F —

— G —

— H —

— I —

— J —

— L —

— M —

I

I

— S —

—T—

— U —

— V —

— W —

A26578 PRINTED IN THE USA 11/90 GTE 13913 25,000 MPU YGABAA

Literature Distribution Centers:

USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.

EUROPE: Motorola Ltd.; European Literature Center; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.

JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141 Japan.

ASIA-PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Center, No. 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.