CPU32

REFERENCE MANUAL

PREFACE

This reference manual describes programming and operation of the CPU32 instruction processing module, found in the M68300 Family of embedded controllers. It is part of a multivolume set of manuals — each volume corresponds to a major module in the M68300 Family.

A user's manual for each device incorporating the CPU32 describes processor function and operation with reference to other modules within the device.

This manual consists of the following sections and appendix:

Section 1 Overview
Section 2 Architecture Summary
Section 3 Data Organization and Addressing Capabilities
Section 4 Instruction Set
Section 5 Processing States
Section 6 Exception Processing
Section 7 Development Support
Section 8 Instruction Execution Timing
Appendix A M68000 Family Summary
Index

NOTE

In this manual, the terms assertion and negation specifya particular logic state. **Assert** and **assertion** refer to an active or true signal. **Negate** and **negation** refer to an inactive or false signal. These terms are used independently of the voltage level that they represent.

This manual is written for systems designers, systems programmers, and applications programmers. Systems designers need general knowledge of the entire volume, with particular emphasis on Section 1, Section 7, and Appendix A — they will also need to be familiar with electrical specifications and mechanical data contained in the user's manual. Systems programmers should become familiar with Sections 1 through 6, Section 8, and Appendix A. Applications programmers can find most of the information they need in Sections 1 through 5, Section 8, and Appendix A.

This manual is also written for users of the M68000 Family that are not familiar with the CPU32. Although there are comparative references to other Motorola microprocessors throughout the manual, Section 1, Section 2, and Appendix A specifically identify the CPU32 within the M68000 Family, and discuss the differences betweeen it and related devices.

TABLE OF CONTENTS

Paragraph **Title** **Page**

SECTION 1 OVERVIEW

SECTION 2ARCHITECTURE SUMMARY

SECTION 3 DATA ORGANIZATION AND ADDRESSING CAPABILITIES

TABLE OF CONTENTS
(Continued)

Paragraph **Title** **Page**

SECTION 4 INSTRUCTION SET

TABLE OF CONTENTS
(Continued)

Paragraph **Title** **Page**

TABLE OF CONTENTS
(Continued)

Paragraph **Title** **Page**

SECTION 7 DEVELOPMENT SUPPORT

TABLE OF CONTENTS
(Continued)

Paragraph **Title** **Page**

SECTION 8 INSTRUCTION EXECUTION TIMING

TABLE OF CONTENTS
(Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS
(Continued)

LIST OF TABLES

LIST OF TABLES
(Continued)

SECTION 1 OVERVIEW

The CPU32, the first-generation instruction processing module of the M68300 Family, is based on the industry-standard MC68000 processor. It has many features of the MC68010 and MC68020, as well as unique features suited for high-performance controller applications. The CPU32 is source code and binary code compatible with the M68000 Family.

CPU32 power consumption during normal operation is low because it is a high-speed complementary metal-oxide semiconductor (HCMOS) device. Power consumption can be reduced to a minimum during periods of inactivity by executing the low-power stop (LPSTOP) instruction, which shuts down the CPU32 and other intermodule bus (IMB) submodules.

Ease of programming is an important consideration in using a microcontroller. The CPU32 instruction format reflects a predominately register-memory interaction philosophy. All data resources are available to all operations requiring those resources. There are eight multifunction data registers and seven general-purpose addressing registers. The data registers readily support 8-bit (byte), 16-bit (word), and 32-bit (long word) operand lengths for all operations. Address manipulation is supported by word and long-word operations. Although the program counter (PC) and stack pointers (SP) are special purpose registers, they are also available for most data addressing activities. Ease of program checking and diagnosis is enhanced by trace and trap capabilities at the instruction level.

As controller applications become more complex and control programs become larger, high-level language (HLL) will become the system designer's choice in programming languages. HLL aids rapid development of complex algorithms, with less error, and is readily portable. The CPU32 instruction set will efficiently support HLL.

1.1 Features

Features of the CPU32 are as follows:

- Fully Upward Object Code Compatible with M68000 Family
- Virtual Memory Implementation
- Loop Mode of Instruction Execution
- Fast Multiply, Divide, and Shift Instructions
- Fast Bus Interface with Dynamic Bus Port Sizing
- Improved Exception Handling for Controller Applications
- Enhanced Addressing Modes
  - — Scaled Index
  - — Address Register Indirect with Base Displacement and
  - — Expanded PC Relative Modes
  - — 32-Bit Branch Displacements
- Instruction Set Enhancements

— High-Precision Multiply and Divide
— Trap On Condition Codes
— Upper and Lower Bounds Checking
- Enhanced Breakpoint Instruction
- Trace on Change of Flow
- Table Lookup and Interpolate Instruction
- Low-Power Stop Instruction
- Hardware Breakpoint Signal, Background Mode
- 16.77-MHz Operating Frequency (–40 to 125°C)
- Fully Static Implementation

1.1.1 Virtual Memory

A system that supports virtual memory has a limited amount of high-speed physical memory that can be accessed directly by the processor and maintains an image of a much larger “virtual” memory on a secondary storage device. When the processor attempts to access a location in the virtual memory map that is not resident in physical memory, a page fault occurs. The access to that location is temporarily suspended while the necessary data is fetched from secondary storage and placed in physical memory. The suspended access is then restarted or continued. The CPU32 uses instruction restart, which requires that only a small portion of the internal machine state be saved. After correcting the fault, the machine state is restored, and the instruction is refetched and restarted. This process is completely transparent to the application program.

1.1.2 Loop Mode Instruction Execution

The CPU32 has several features that provide efficient execution of program loops. One of these features is the DBcc looping primitive. To increase the performance of the CPU32, a loop mode has been added to the processor. The loop mode is used by any single-word instruction that does not change the program flow. Loop mode is implemented in conjunction with the DBcc instruction. **Figure 1-1** shows the required form of an instruction loop for the processor to enter loop mode.

Loop mode is entered when DBcc is executed and loop displacement is –4. Once in loop mode, the processor performs only data cycles associated with the instruction and suppresses instruction fetches. Termination condition and count are checked after each execution of looped instruction data operations. The CPU automatically exits loop mode for interrupts or other exceptions.

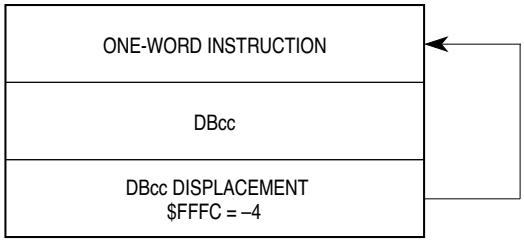


Figure 1-1 Loop Mode Instruction Sequence

1.1.3 Vector Base Register

The vector base register (VBR) contains the base address of the 1024-byte exception vector table. The table contains 256 exception vectors. Exception vectors are the memory addresses of routines that begin execution at the completion of exception processing. Each routine performs operations appropriate to the corresponding exception. Because exception vectors are memory addresses, each table entry is a single long word.

Each vector is assigned an 8-bit number. Vector numbers for some exceptions are obtained from an external device; others are supplied automatically by the processor. The processor multiplies the vector number by four to calculate vector offset, then adds the offset to the VBR base address. The sum is the memory address of the vector.

Because the VBR stores the vector table base address, the table can be located anywhere in memory. It can also be dynamically relocated for each task executed by an operating system. Details of exception processing are provided in **SECTION 6 EXCEPTION PROCESSING**.

1.1.4 Exception Handling

The processing of an exception occurs in four steps, with variations for different exception causes. During the first step, a temporary internal copy of the status register is made, and the status register is set for exception processing. During the second step, the exception vector is determined. During the third step, the current processor context is saved. During the fourth step, a new context is obtained, and the processor then proceeds with normal instruction execution.

Exception processing saves the most volatile portion of the current context by pushing it on the supervisor stack. This context is organized in a format called an exception stack frame. The stack frame always includes the status register and program counter at the time an exception occurs. To support generic handlers, the processor also places the vector offset in the exception stack frame and marks the frame with a format code. The return-from-exception (RTE) instruction uses the format code to determine what information is on the stack, so that context can be properly restored.

1.1.5 Enhanced Addressing Modes

Addressing in the CPU32 is register oriented. Most instructions allow the results of the specified operation to be placed either in a register or in memory. There is no need for extra instructions to store register contents in memory.

There are seven basic addressing modes:

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

The register indirect addressing modes include postincrement, predecrement, and offset capability. The PC relative mode also has index and offset capabilities. In addition to the addressing modes, many instructions implicitly specify the use of a status register, SP, and/or PC. Addressing is explained fully in **SECTION 3 DATA ORGANIZATION AND ADDRESSING CAPABILITIES**. A summary of M68000 Family addressing modes is found in **APPENDIX A M68000 FAMILY SUMMARY**.

1.1.6 Instruction Set

The instruction set of the CPU32 is very similar to that of the MC68020 (see Table 1-1). Two new instructions have been added to facilitate controller applications — low-power stop (LPSTOP) and table lookup and interpolate (TBL). The following M68020 instructions **are not implemented** on the CPU32:

BFxxx — Bit Field Instructions (BFCHG, BFCLR, BFEXTS, BFEXTU, BFFFO BFINS, BFSET, BFTST)
CALLM, RTM — Call Module, Return Module
CAS, CAS2 — Compare and Set (Read-Modify-Write Instructions)
cpxxx Coprocessor Instructions (cpBcc, cpDBcc, cpGEN, cp RESTORE, cpSAVE, cpScc, cpTRAPcc)
PACK, UNPK Pack, Unpack BCD Instructions

The CPU32 traps on unimplemented instructions and illegal effective addressing modes, allowing the user to emulate instructions or to define special-purpose functions. However, Motorola reserves the right to use all currently uniplemented instructions operation codes for future M68000 core enhancements.

See **SECTION 4 INSTRUCTION SET** for comprehensive information.

1.1.6.1 Table Lookup and Interpolation Instructions

To speed up real-time applications, a range of discrete data points is often precalculated from a continuous control function, then stored in memory. A full range of data can require an inordinate amount of memory. The table instructions make it possible

to store a sample of the full range and recover intermediate values quickly via linear interpolation. A round-to-nearest algorithm can be applied to the results.

Table 1-1 Instruction Set Summary

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
BGND	Background
BKPT	Breakpoint
BRA	Branch
BSET	Test Bit and Set
BSR	Branch to Subroutine
BTST	Test Bit
CHK, 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
LEA	Load Effective Address
LINK	Link and Allocate
LPSTOP	Low Power Stop
LSL, LSR	Logical Shift Left and Right
ILLEGAL	Take Illegal Instruction Trap
JMP	Jump
JSR	Jump to Subroutine

Mnemonic	Description
MOVE	Move
MOVE CCR	Move Condition Code Register
MOVE SR	Move Status Register
MOVE USP	Move User Stack Pointer
MOVEA	Move Address
MOVEC	Move Control Register
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral
MOVEQ	Move Quick
MOVES	Move Alternate Address Space
MULS, MULS.L	Signed Multiply
MULU, MULU.L	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NEGX	Negate with Extend
NOP	No Operation
OR	Logical Inclusive OR
ORI	Logical Inclusive OR Immediate
PEA	Push Effective 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
TBLS, TBLSN	Table Lookup and Interpolate (Signed)
TBLU, TBLUN	Table Lookup and Interpolate (Unsigned)
TAS	Test Operand and Set
TRAP	Trap
TRAPcc	Trap Conditionally
TRAPV	Trap on Overflow
TST	Test Operand
UNLK	Unlink

1.1.6.2 Low-Power Stop Instruction

The CPU32 is a fully static design. Power consumption can be reduced to a minimum during periods of inactivity by stopping the system clock. The CPU32 instruction set includes a low-power stop command (LPSTOP) that efficiently implements this capability. The processor will remain in stop mode until a user-specified interrupt, or reset, occurs.

1.1.7 Processing States

There are four processing states — normal, exception, background and halted.

Normal processing is associated with instruction execution. The bus is used to fetch instructions and operands, and to store results.

Exception processing is associated with interrupts, trap instructions, tracing, and other exception conditions.

Background processing allows interactive debugging of the system.

Halted processing is an indication of catastrophic hardware failure.

See **SECTION 5 PROCESSING STATES** for complete information.

1.1.8 Privilege States

The processor can operate at either of two privilege levels. Supervisor level is more privileged than user level — all instructions are available at supervisor level, but access is restricted at user level.

Effective use of privilege level can protect system resources from uncontrolled access. The state of the S bit in the status register determines access level and whether the stack pointer (USP) or the supervisor stack pointer (SSP) is used for stack operations.

See **SECTION 5 PROCESSING STATES** for a complete explanation of privilege levels.

1.2 Block Diagram

A block diagram of the CPU32 is shown in **Figure 1-2**. The functional elements operate concurrently. Essential synchronization of instruction execution and buss operation is maintained by the sequencer/control unit. The bus controller prefetches instructions and operands. A three-stage pipeline is used to hold and decode instructions prior to execution. The execution unit maintains the program counter under sequencer control. The bus control contains a write-pending buffer that allows the sequencer to continue execution of instructions after a request for a write cycle is queued. See **SECTION 8 INSTRUCTION EXECUTION TIMING** for a detailed explanation of instruction execution.

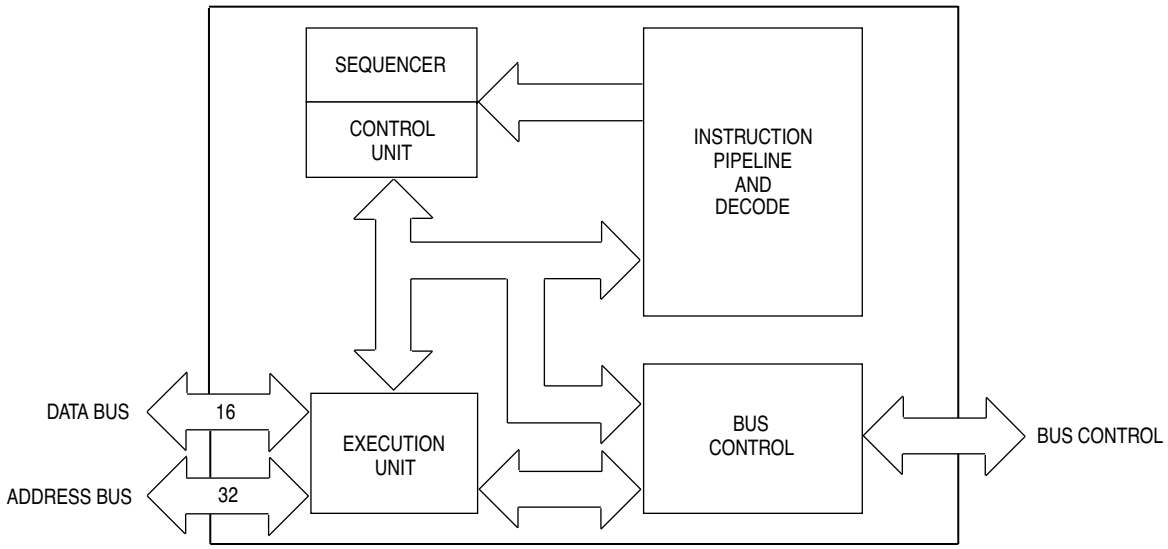


Figure 1-2 CPU32 Block Diagram

SECTION 2ARCHITECTURE SUMMARY

The CPU32 is upward source and object code compatible with the MC68000 and MC68010. It is downward source and object code compatible with the MC68020. Within the M68000 Family, architectural differences are limited to the supervisory operating state. User state programs can be executed unchanged on upward compatible devices.

The major CPU32 features are as follows:

- 32-Bit Internal Data Path and Arithmetic Hardware
- 32-Bit Address Bus Supported by 32-Bit Calculations
- Rich Instruction Set
- Eight 32-Bit General-Purpose Data Registers
- Seven 32-Bit General-Purpose Address Registers
- Separate User and Supervisor Stack Pointers
- Separate User and Supervisor State Address Spaces
- Separate Program and Data Address Spaces
- Many Data Types
- Flexible Addressing Modes
- Full Interrupt Processing
- Expansion Capability

2.1 Programming Model

The CPU32 programming model consists of two groups of registers that correspond to the user and supervisor privilege levels. User programs can only use the registers of the user model. The supervisor programming model, which supplements the user programming model, is used by CPU32 system programmers who wish to protect sensitive operating system functions. The supervisor model is identical to that of MC68010 and later processors.

The CPU32 has eight 32-bit data registers, seven 32-bit address registers, a 32-bit program counter, separate 32-bit supervisor and user stack pointers, a 16-bit status register, two alternate function code registers, and a 32-bit vector base register (see **Figure 2-1** and **Figure 2-2**).

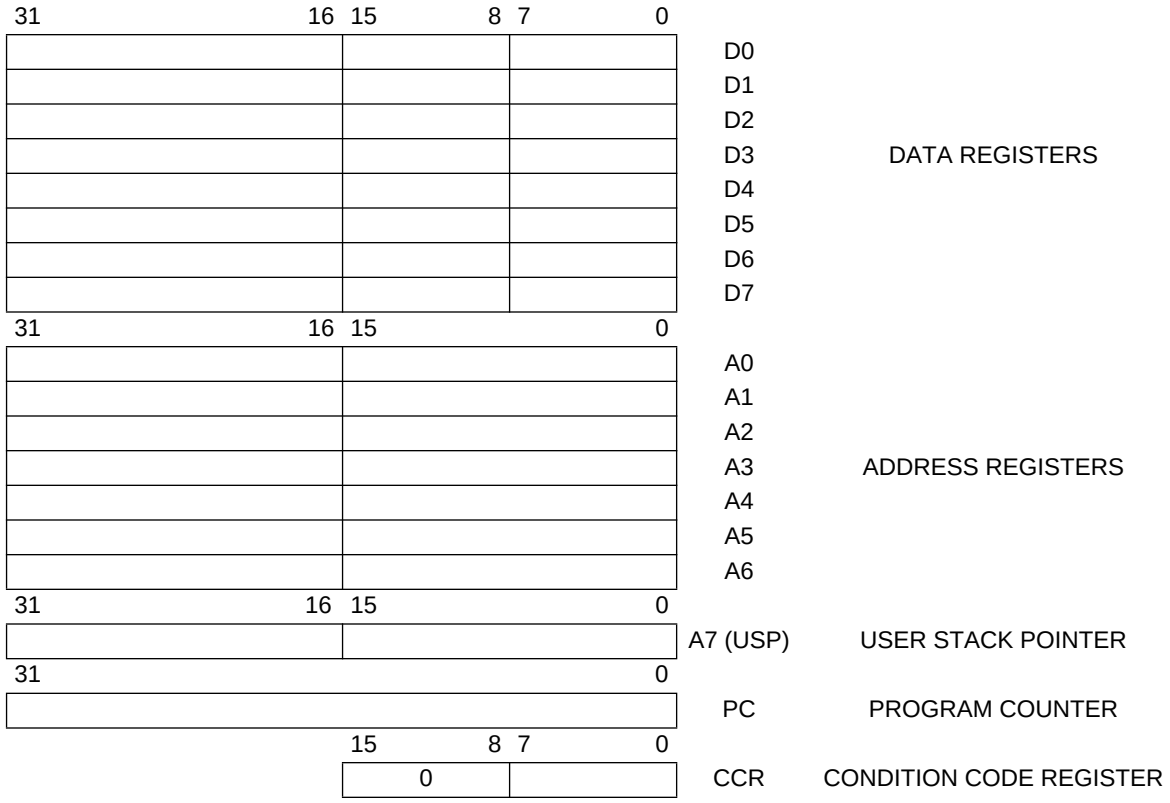


Figure 2-1 User Programming Model

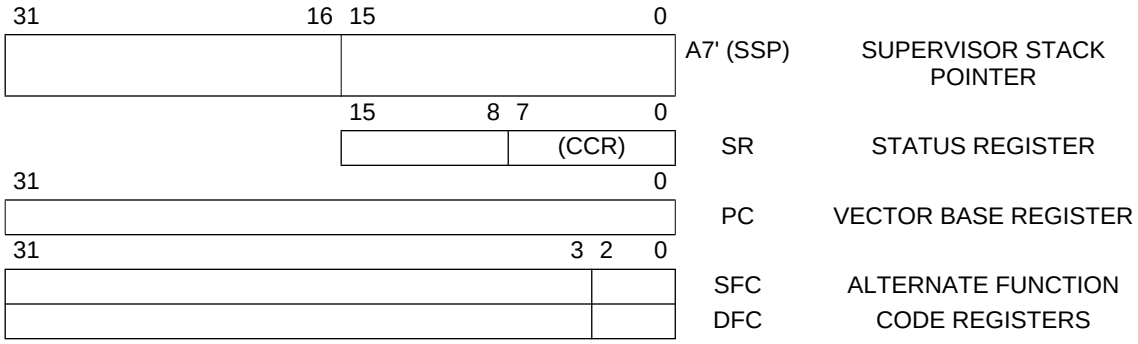


Figure 2-2 Supervisor Programming Model Supplement

2.2 Registers

Registers D7 to D0 are used as data registers for bit, byte (8-bit), word (16-bit), long-word (32-bit), and quad-word (64-bit) operations. Registers A6 to A0 and the user and supervisor 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 2-1**) is a register designation that applies to the user stack pointer in the user privilege level and to the supervisor stack pointer in the supervisor privilege level. In addition, address registers may be used for word and long-word operations. All of the 16 general-purpose registers (D7 to D0, A7 to A0) may be used as index registers.

The program counter (PC) contains the address of the next instruction to be executed by the CPU32. During instruction execution and exception processing, the processor automatically increments the contents of the PC or places a new value in the PC, as appropriate.

The status register (SR) (see **Figure 2-3**) contains condition codes, an interrupt priority mask (three bits), and three control bits. Condition codes reflect the results of a previous operation. The codes are contained in the low byte, or condition code register of the SR. The interrupt priority mask determines the level of priority an interrupt must have in order to be acknowledged. The control bits determine trace mode and privilege level. At user privilege level, only the condition code register is available. At supervisor privilege level, software can access the full status register.

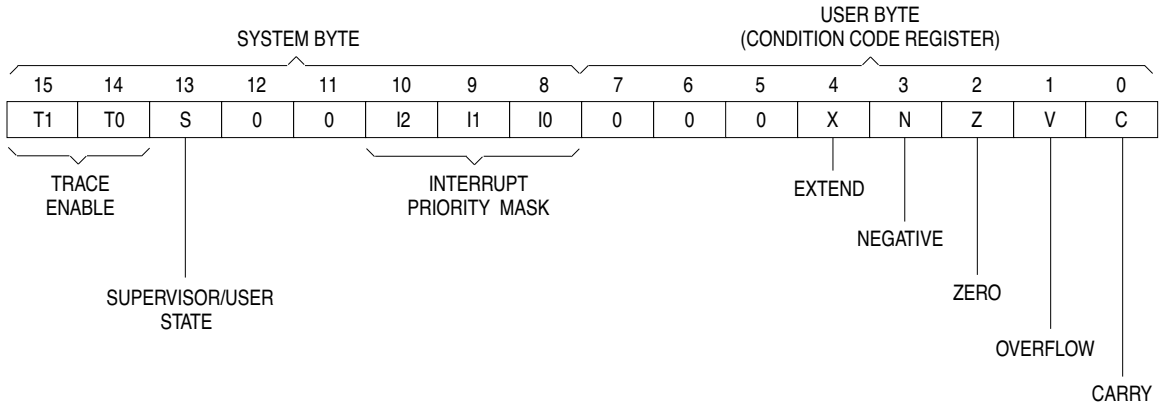


Figure 2-3 Status Register

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. The CPU32 generates a function code each time it accesses an address. Specific codes are assigned to each type of access. The codes can be used to select eight dedicated 4G-byte address spaces. The MOVE instructions can use registers SFC and DFC to specify the function code of a memory address.

2.3 Data Types

Six basic data types are supported:

1. Bits
2. Binary-Coded Decimal (BCD) Digits
3. Byte Integers (8 bits)
4. Word Integers (16 bits)
5. Long-Word Integers (32 bits)
6. Quad-Word Integers (64 bits)

2.3.1 Organization in Registers

The eight data registers can store data operands of 1, 8, 16, 32, and 64 bits and addresses of 16 or 32 bits. The seven address registers and the two stack pointers are used for address operands of 16 or 32 bits. The PC is 32 bits wide.

2.3.1.1 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 (LSB) of a long-word integer is addressed as bit zero, and the most significant bit (MSB) is addressed as bit 31. **Figure 2-4** shows the organization of various types of data in the data registers.

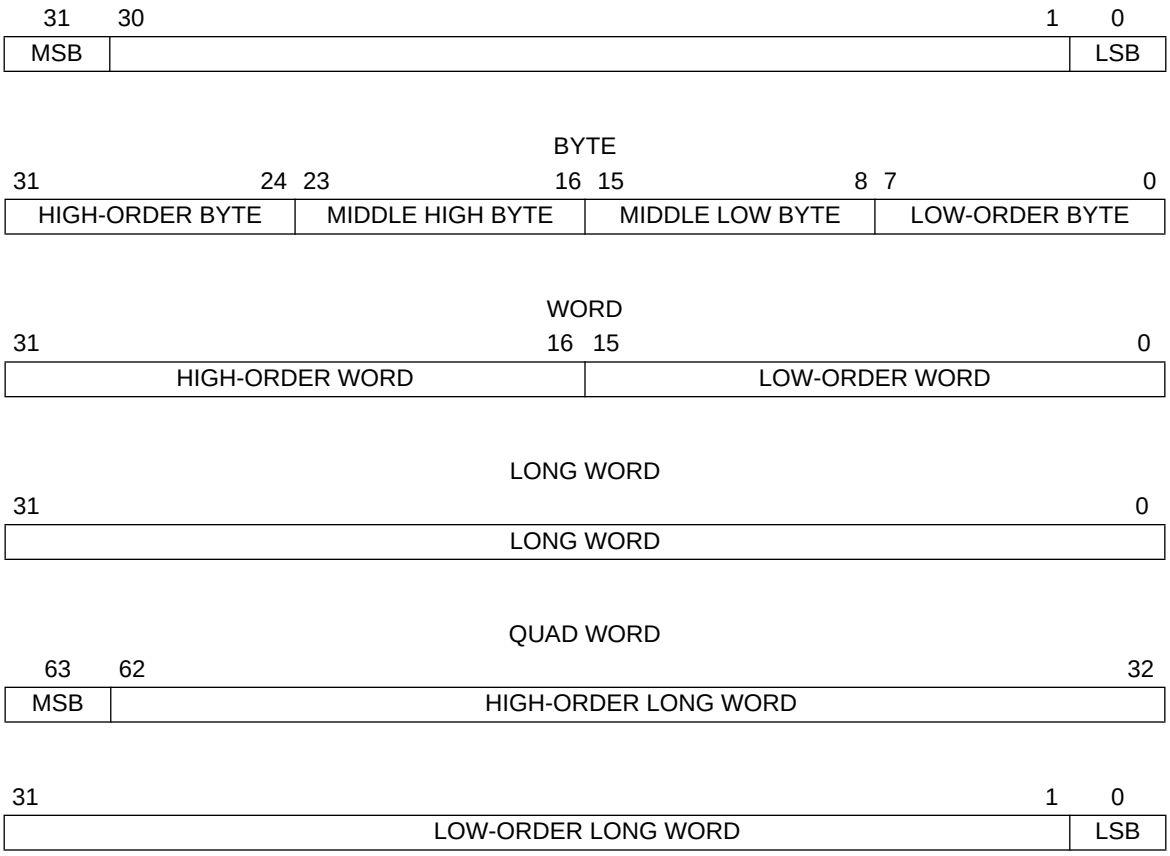


Figure 2-4 Data Organization in Data Registers

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; however, the MOVEM instruction can be used to move a quad word into or out of the registers.

BCD data represents decimal numbers in binary form. CPU32 BCD instructions use a format in which a byte contains two digits — the four LSB contain the low digit, and the four MSB contain the high digit. The ABCD, SBCD, and NBCD instructions operate on two BCD digits packed into a single byte.

2.3.1.2 Address Registers

Each address register and stack pointer holds a 32-bit address. Address registers cannot be used for byte-sized operands. 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 a destination operand, the entire register is affected, regardless of operation size. If the source operand is a word, it is first sign extended to 32 bits, and then used in the operation. Address registers can be used to support address computation. The instruction set includes instructions that add to, subtract from, compare, and move the contents of address registers. **Figure 2-5** shows the organization of addresses in address registers.

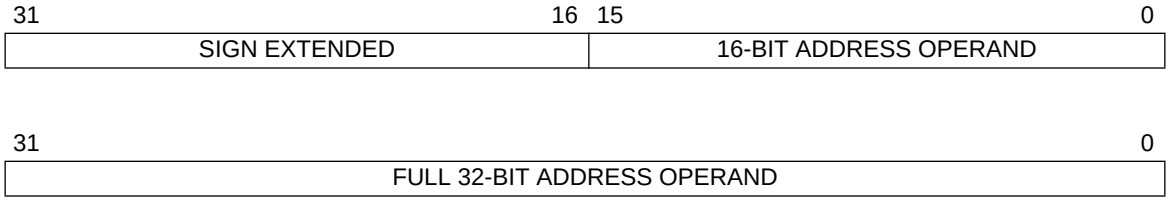


Figure 2-5 Address Organization in Address Registers

2.3.1.3 Control Registers

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

The SR shown in **Figure 2-3** is 16 bits wide. Only 11 bits of the SR are defined, and 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 SR is the CCR. Operations to the CCR can be performed at the supervisor or user privilege level. All operations to the SR and CCR are word-size operations. For all CCR operations, the upper byte is read as all zeros and is ignored when written, regardless of privilege level.

The alternate function code registers (SFC and DFC) are 32-bit registers with only bits [2:0] implemented. These bits contain address space values (FC2 to FC0) for the read or write operand of the MOVES instruction. 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.

2.3.2 Organization in Memory

Memory is organized on a byte-addressable basis. An address corresponds to a high-order byte. For example, the address (N) of a long-word data item is the address of the most significant byte of the high-order word. The address of the most significant byte of the low-order word is (N + 2), and the address of the least significant byte of the long word is (N + 3). The CPU32 requires data words and long words, as well as instruction words to be aligned on word boundaries. Data misalignment is not supported. **Figure 2-6** shows how operands and instructions are organized in memory. Note that (N + X) is below (N) — that is, address value increases as one moves down the page.

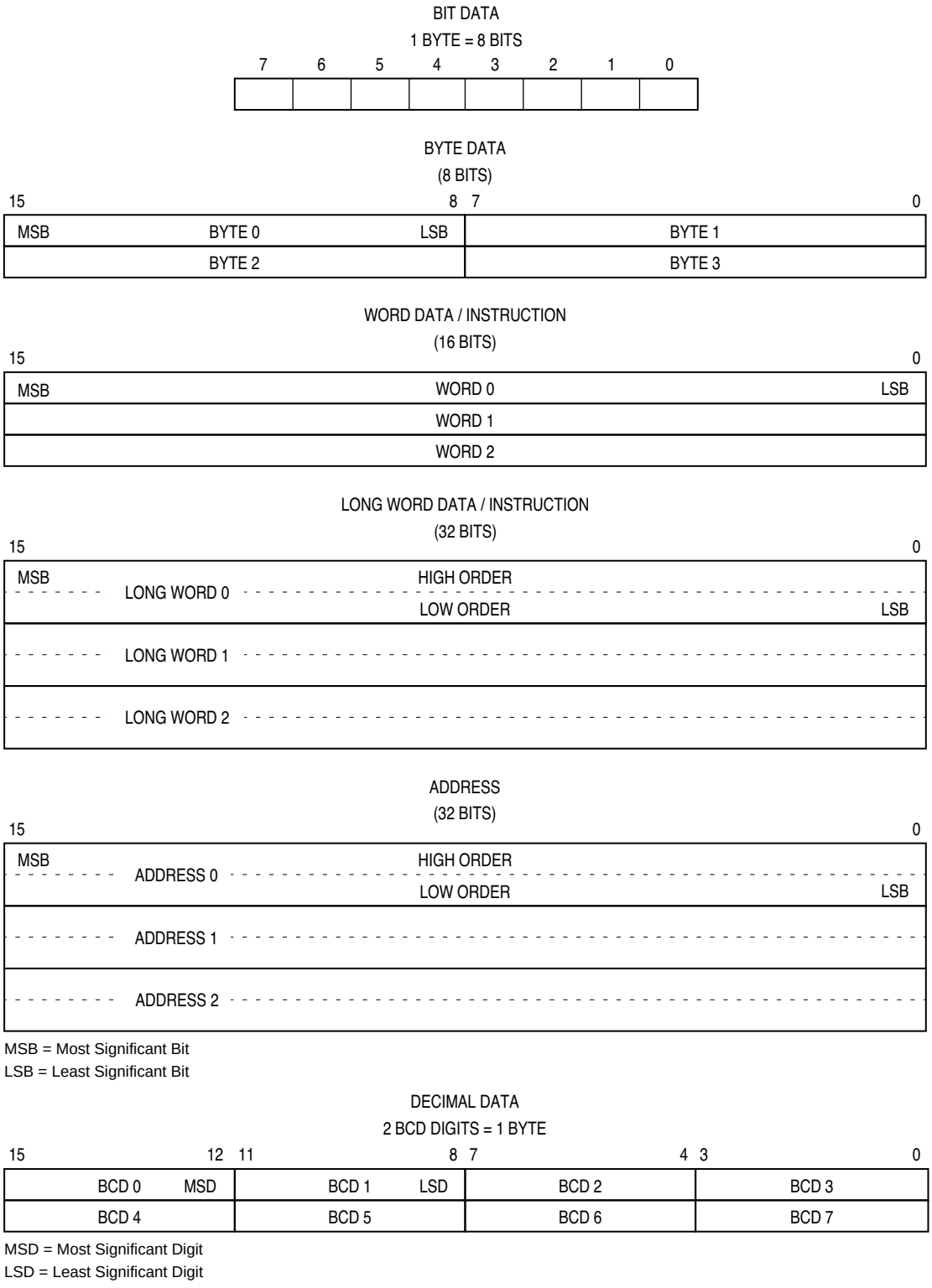


Figure 2-6 Memory Operand Addressing

SECTION 3 DATA ORGANIZATION AND ADDRESSING CAPABILITIES

The addressing mode of an instruction can specify the value of an operand (an immediate operand), a register that contains the operand (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 3-1 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 designation is composed of two 3-bit fields, the mode field and the register field. The value in the mode field selects a mode or a set of modes. The register field specifies a register for the mode or a submode for modes that do not use registers.

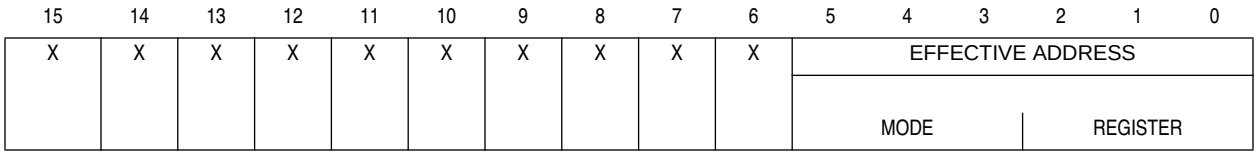


Figure 3-1 Single-Effective-Address Instruction Operation Word

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

Additional information may be needed to specify an operand address. This information is contained in an additional word or words called the effective address extension, and is considered part of an instruction. Address extension formats are discussed in **3.4.4 Effective Address Encoding Summary**.

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

3.1 Program and Data References

An M68000 Family processor makes two classes of memory references, each of which has a complete, separate logical address space.

References to opcodes and extension words are program space references.

Operand reads and writes are primarily data space references. Operand reads are from data space in all but two cases — immediate operands embedded in the instruction stream and operands addressed relative to the current program counter are program space references. All operand writes are to data space.

3.2 Notation Conventions

EA — Effective address
An — Address register n
    Example: A3 is address register 3
Dn — Data register n
    Example: D5 is data register 5
Rn — Any register, data or address
Xn.SIZE*SCALE —
    Index register n (data or address),
    Index size (W for word, L for long word),
    Scale factor (1, 2, 4, or 8 for byte, word, long-word or quad-word scaling)
PC — Program counter
SR — Status register
SP — Stack pointer
CCR — Condition code register
USP — User stack pointer
SSP — Supervisor stack pointer
dn — Displacement value, n bits wide
bd — Base displacement
L — Long-word size
W — Word size
B — Byte size
(An) — Identifies an indirect address in a register

3.3 Implicit Reference

Some instructions make implicit reference to the program counter, the system stack pointer, the user stack pointer, the supervisor stack pointer, or the status register. The following table shows the instructions and the registers involved:

Instruction	Implicit Registers
ANDI to CCR	SR
ANDI to SR	SR
BRA	PC
BSR	PC, SP
CHK (exception)	PC, SP
CHK2 (exception)	SSP, SR
DBcc	PC
DIVS (exception)	SSP, SR
DIVU (exception)	SSP, SR
EORI to CCR	SR
EORI to SR	SR
JMP	PC
JSR	PC, SP
LINK	SP
LPSTOP	SR
MOVE CCR	SR
MOVE SR	SR
MOVE USP	USP

Instruction	Implicit Registers
ORI to CCR	SR
ORI to SR	SR
PEA	SP
RTD	PC, SP
RTE	PS, SP, SR
RTR	PC, SP, SR
RTS	PC, SP
STOP	SR
TRAP (exception)	SSP, SR
TRAPV (exception)	SSP, SR
UNLK	SP

3.4 Effective Address

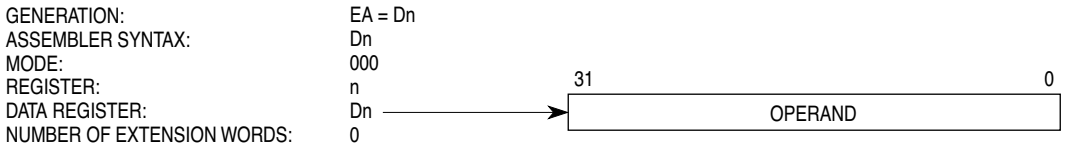
Most instructions specify the location of an operand by a field in the operation word called an effective address field or an effective address (⟨EA⟩). An EA is composed of two 3-bit subfields: mode specification field and register specification field. Each of the address modes is selected by a particular value in the mode specification subfield of the EA. The EA field may require further information to fully specify the operand. This information, called the EA extension, is in a following word or words and is considered part of the instruction (see **3.1 Program and Data References**).

3.4.1 Register Direct Mode

These EA modes specify that the operand is in one of the 16 multifunction registers.

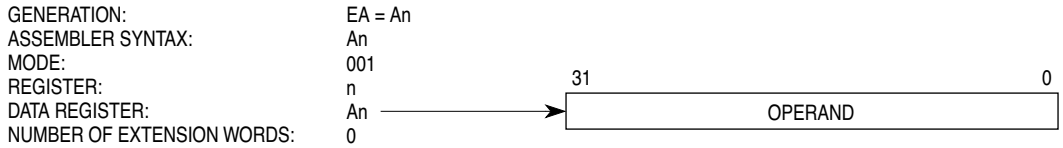
3.4.1.1 Data Register Direct

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



3.4.1.2 Address Register Direct

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

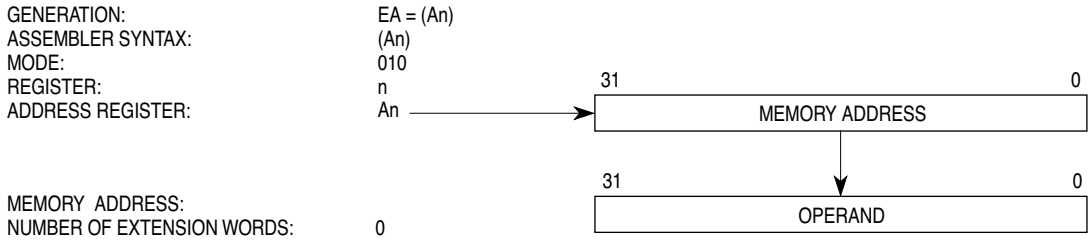


3.4.2 Memory Addressing Modes

These EA modes specify the address of the memory operand.

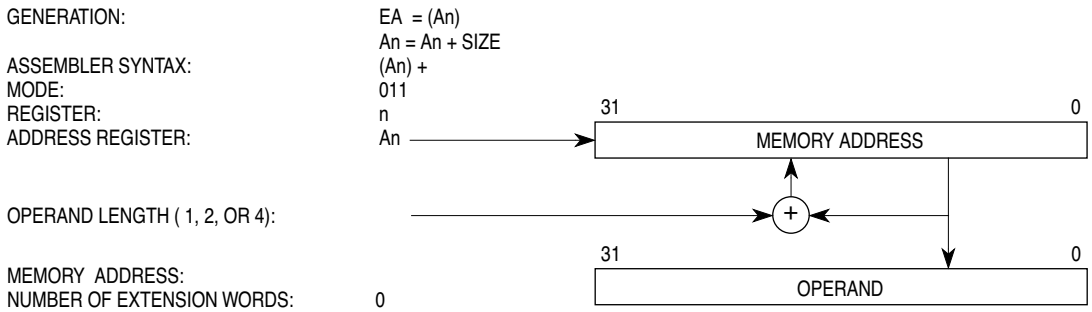
3.4.2.1 Address Register Indirect

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.



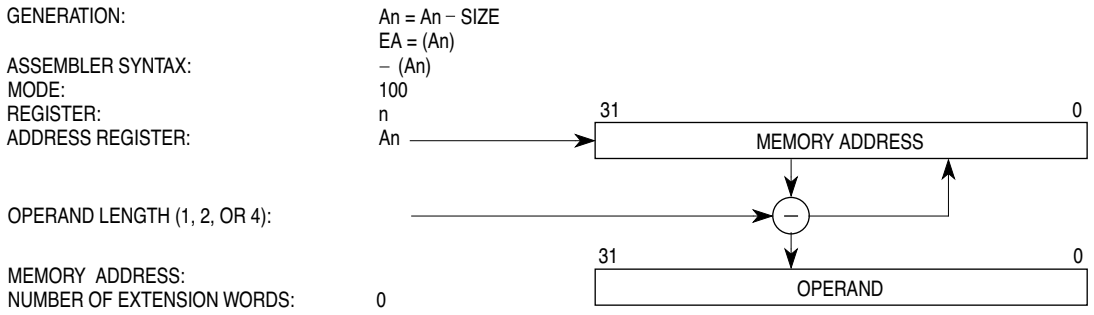
3.4.2.2 Address Register Indirect With Postincrement

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. 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.



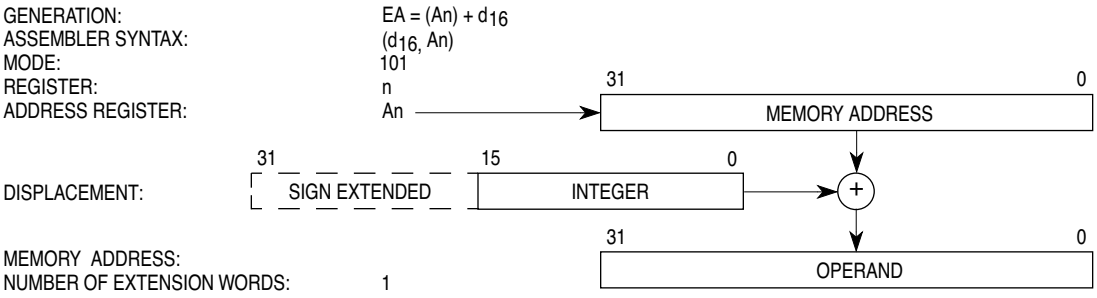
3.4.2.3 Address Register Indirect With Predecrement

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. 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.



3.4.2.4 Address Register Indirect With Displacement

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 before being used in EA calculations.

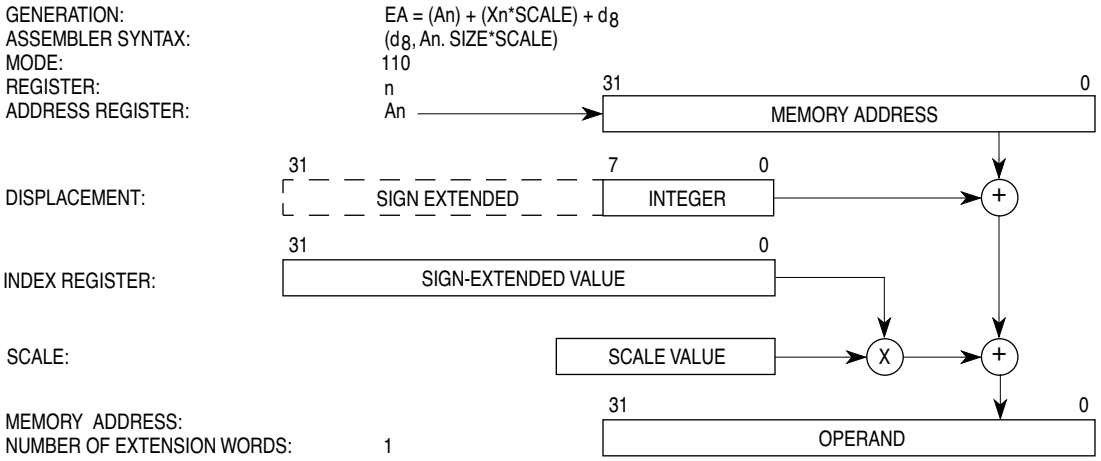


3.4.2.5 Address Register Indirect With Index (8-Bit Displacement)

This 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 displacement, address register, and index register.

This address mode can have either of two different formats of extension. The brief format (8-bit displacement) requires one word of extension and provides fast indexed addressing. The full format (16 and 32-bit displacement) provides optional displacement size. Both forms use an index operand.

For brief format addressing, the address of the operand is the sum of the address in the address register, the sign-extended displacement integer in the low-order eight bits of the extension word, and the index operand. The reference is classed as a data reference, except for the JMP and JSR instructions. The index operand is specified “Ri.sz*scl”.



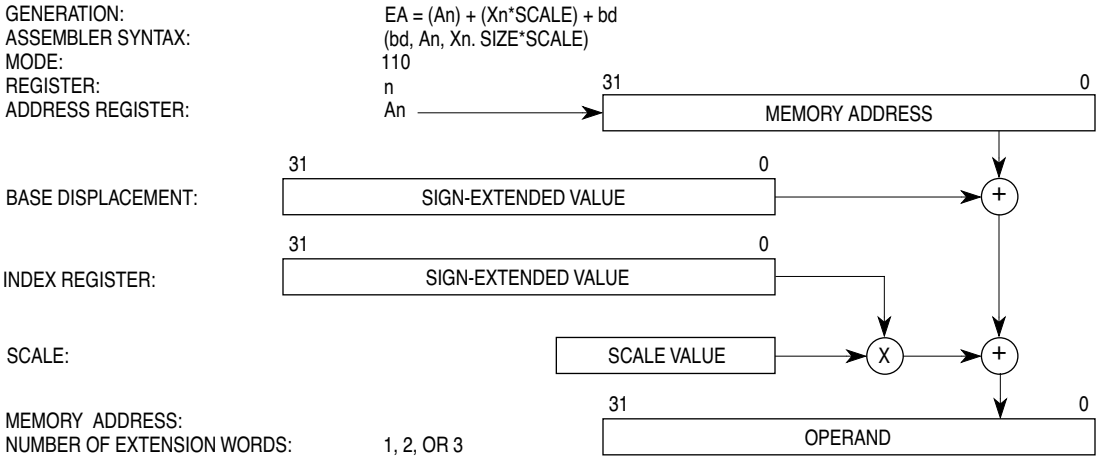
"Ri" specifies a general data or address register used as an index register. The index operand is derived from the index register. The index register is a data register if bit [15] = 0 in the first extension word and an address register if bit [15] = 1. The index register number is given by extension word bits [14:12].

Index size is referred to as "sz". It may be either "W" or "L". Index size is given by bit [11] of the extension word. If bit [11] = 0, the index value is the sign-extended low-order word integer of the index register (W). If bit [11] = 1, the index value is the long integer in the index register (L).

The term "scl" refers to index scale selection and may be 1, 2, 4, or 8. The index value is scaled according to bits [10:9]. Codes 00, 01, 10, or 11 select index scaling of 1, 2, 4, or 8, respectively.

3.4.2.6 Address Register Indirect With Index (Base Displacement)

The full format indexed 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 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 scaled contents of the sign-extended index register, and the base displacement.

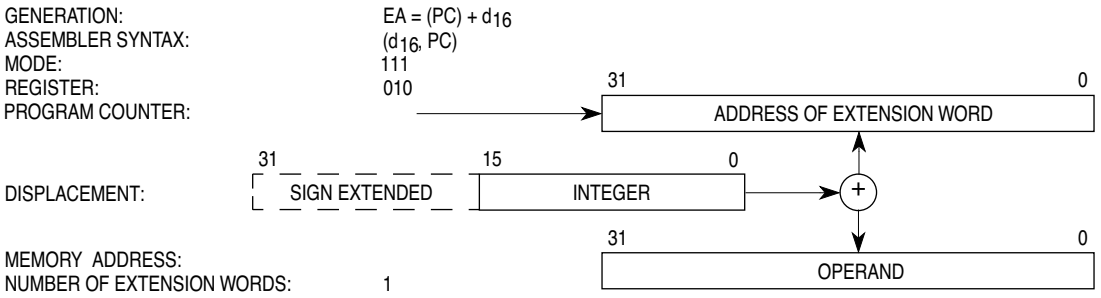


3.4.3 Special Addressing Modes

These special addressing modes do not use the register field to specify a register number but rather to specify a submode.

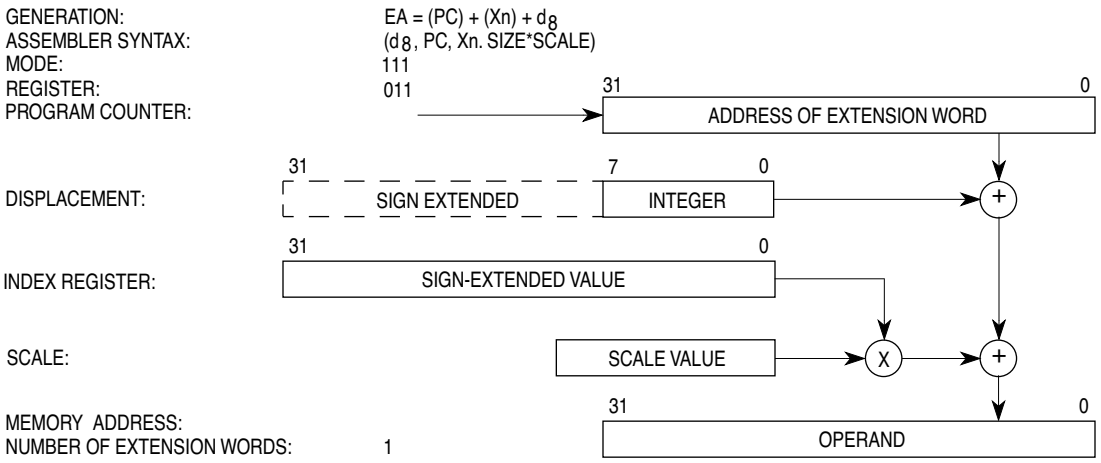
3.4.3.1 Program Counter Indirect With Displacement

In this mode, the operand is in memory. The address of the operand is the sum of the address in the program counter and the sign-extended 16-bit displacement integer in the extension word. The value in the program counter is the address of the extension word. The reference is a program space reference and is only allowed for read accesses.



3.4.3.2 Program Counter Indirect with Index (8-Bit Displacement)

This mode is similar to the address register indirect with index (8-bit displacement) mode described in **3.4.2.5 Address Register Indirect With Index (8-Bit Displacement)**, but the program counter is used as the base register.



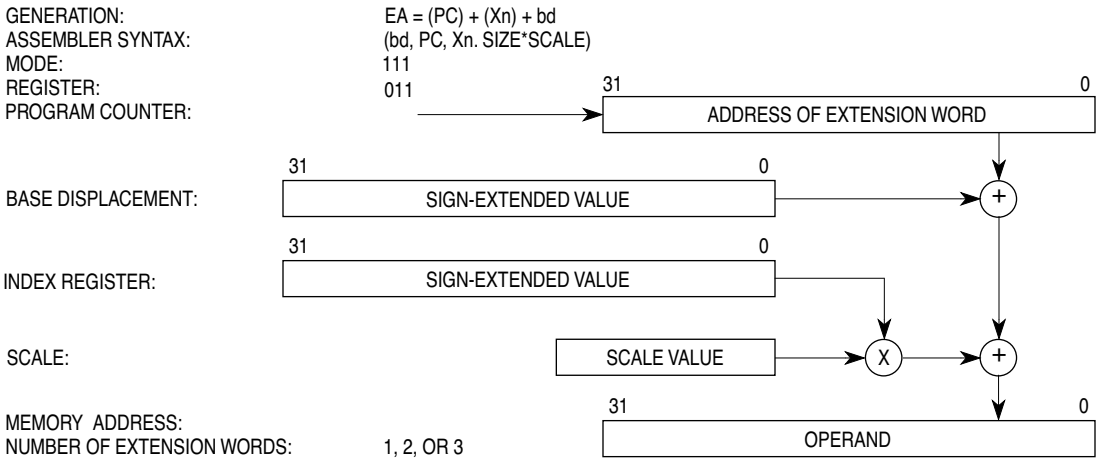
The operand is in memory. The address of the operand is the sum of the address in the program counter, the sign-extended displacement integer in the lower eight bits of the extension word, and the sized, scaled, and sign-extended index operand. The value in the program counter is the address of the extension word. This reference is a program space reference and is only allowed for reads. The user must include the displacement, the program counter, and the index register when specifying this addressing mode.

3.4.3.3 Program Counter Indirect with Index (Base Displacement)

This mode is similar to the address register indirect with index (base displacement) mode described in **3.4.2.6 Address Register Indirect With Index (Base Displacement)**, but the program counter is used as the base register. It requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement.

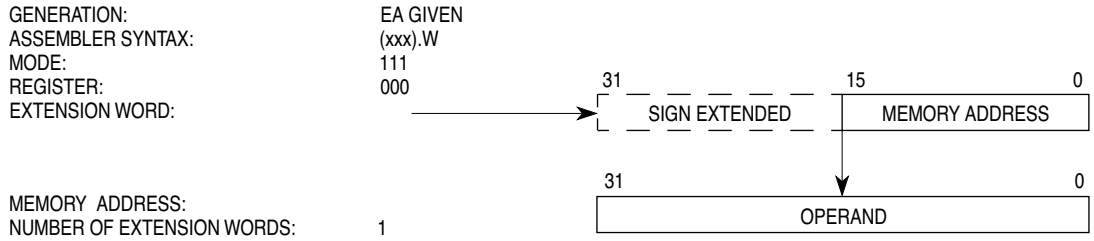
The operand is in memory. The address of the operand is the sum of the contents of the program counter, the scaled contents of the sign-extended index register, and the base displacement. The value of the program counter is the address of the first extension word. The reference is a program space reference and is only allowed for read accesses.

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



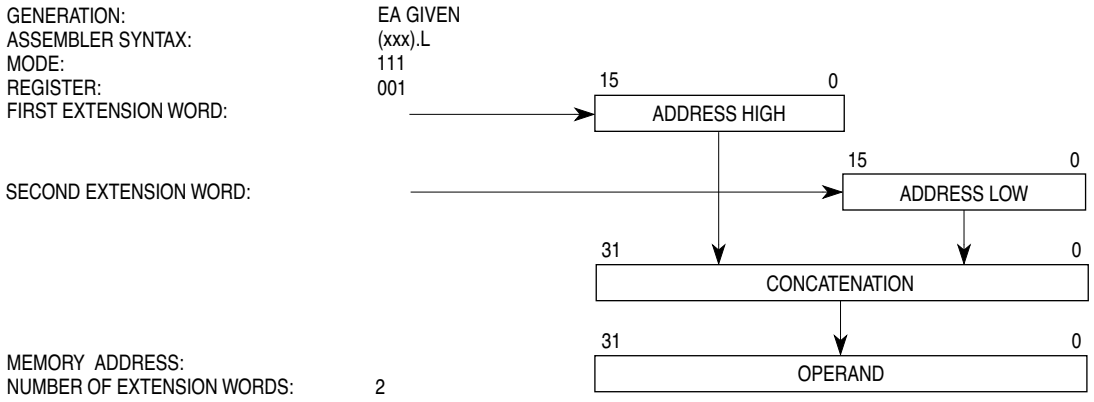
3.4.3.4 Absolute Short Address

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.



3.4.3.5 Absolute Long Address

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.



3.4.3.6 Immediate Data

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

Byte Operation

The operand is in the low-order byte of the extension word.

Word Operation

The 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.

GENERATION:	OPERAND GIVEN
ASSEMBLER SYNTAX:	#XXX
MODE:	111
REGISTER:	100
NUMBER OF EXTENSION WORDS:	1 OR 2

3.4.4 Effective Address Encoding Summary

Most addressing modes use one of the three formats shown in **Figure 3-2**. The single EA instruction is in the format of the instruction word. 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”.

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 CPU32 contains six extension words. It is a MOVE instruction with full format extension words for both source and destination EA and a 32-bit base displacement for both addresses.

SINGLE EA INSTRUCTION FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	EFFECTIVE ADDRESS					
										MODE		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)									

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

*Memory indirect addressing will cause illegal instruction trap; must be = 000 if IS = 1

Figure 3-2 Effective Address Specification Formats

EA modes can be classified as follows:

Data	A data addressing EA mode refers to data operands.
Memory	A memory addressing EA mode refers to memory operands.
Alterable	An alterable addressing EA mode refers to writable operands.
Control	A control addressing EA mode refers to unsized memory operands.

Categories are sometimes combined, forming new, more restrictive, categories. Two examples are alterable memory or alterable data. The former refers to addressing modes that are both alterable and memory addresses; the latter refers to addressing modes that are both alterable and data addresses. **Table 3-1** shows categories to which each of the EA modes belong.

3.5 Programming View of Addressing Modes

Extensions to indexed addressing modes, indirection, and full 32-bit displacements provide additional programming capabilities for the CPU32. The following paragraphs describe addressing techniques and summarize addressing modes from a programming point of view.

Table 3-1 Effective Addressing Mode Categories

Addressing Mode	Code	Register	Data	Memory	Control	Alterable	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)
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)							
Immediate	111	100	X	X	—	—	#(data)

3.5.1 Addressing Capabilities

In the CPU32, setting the base register suppress (BS) bit in the full format extension word (see **Figure 3-2**) suppresses use of the base address register in calculating the EA, allowing any index register to be used in place of the base register. Because any data register can be an index register, this provides a data register indirect form (Dn). This mode could also be called register indirect (Rn) because either a data register or an address register can be used to address memory — an extension of M68000 Family addressing capability.

The ability to specify the size and scale of an index register (Xn.SIZE ∗ SCALE) in these modes provides additional addressing flexibility. When 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 3-3**).

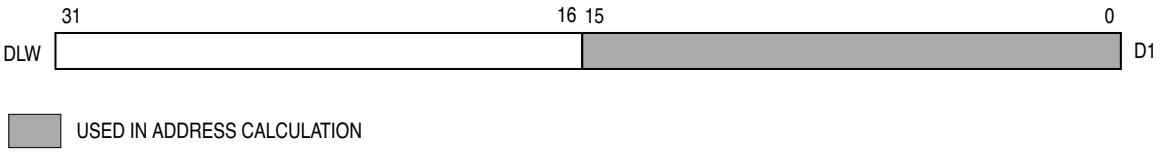


Figure 3-3 Using SIZE in the Index Selection

For the CPU32, the register indirect modes can be extended further. Because displacements can be 32 bits wide, they can represent absolute addresses or the results of expressions that contain absolute addresses. This scheme 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 3-4**).

Setting the index register suppress bit (IS) in the full format extension word suppresses the index operand. The indirect suppressed index register mode 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.

An optional scaling function supports direct array subscripting. An index register can be left shifted by zero, one, two, or three bits before use in an EA calculation, to scale for an array of elements of corresponding size. This is much more efficient than using an arithmetic value in one of the general-purpose registers to multiply the index register by one, two, four, or eight.

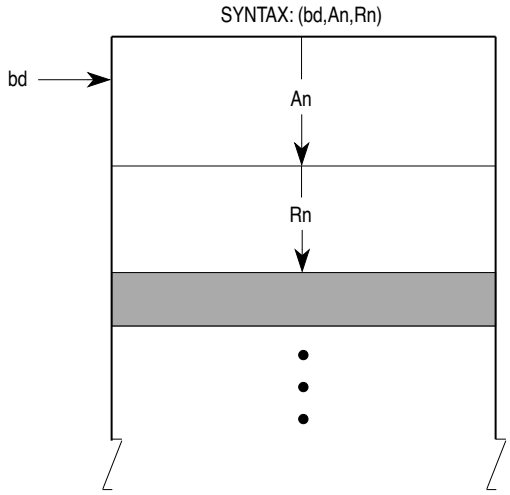


Figure 3-4 Using Absolute Address with Indexes

Scaling does not add to the EA calculation time. However, when combined with the appropriate derived modes, scaling produces additional capabilities. Arrayed structures can be addressed absolutely and then subscripted; for example, (bd, Rn ∗ SCALE). 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 (see **Figure 3-5**). In the second example, An contains the address of an array and Rn contains a subscript.

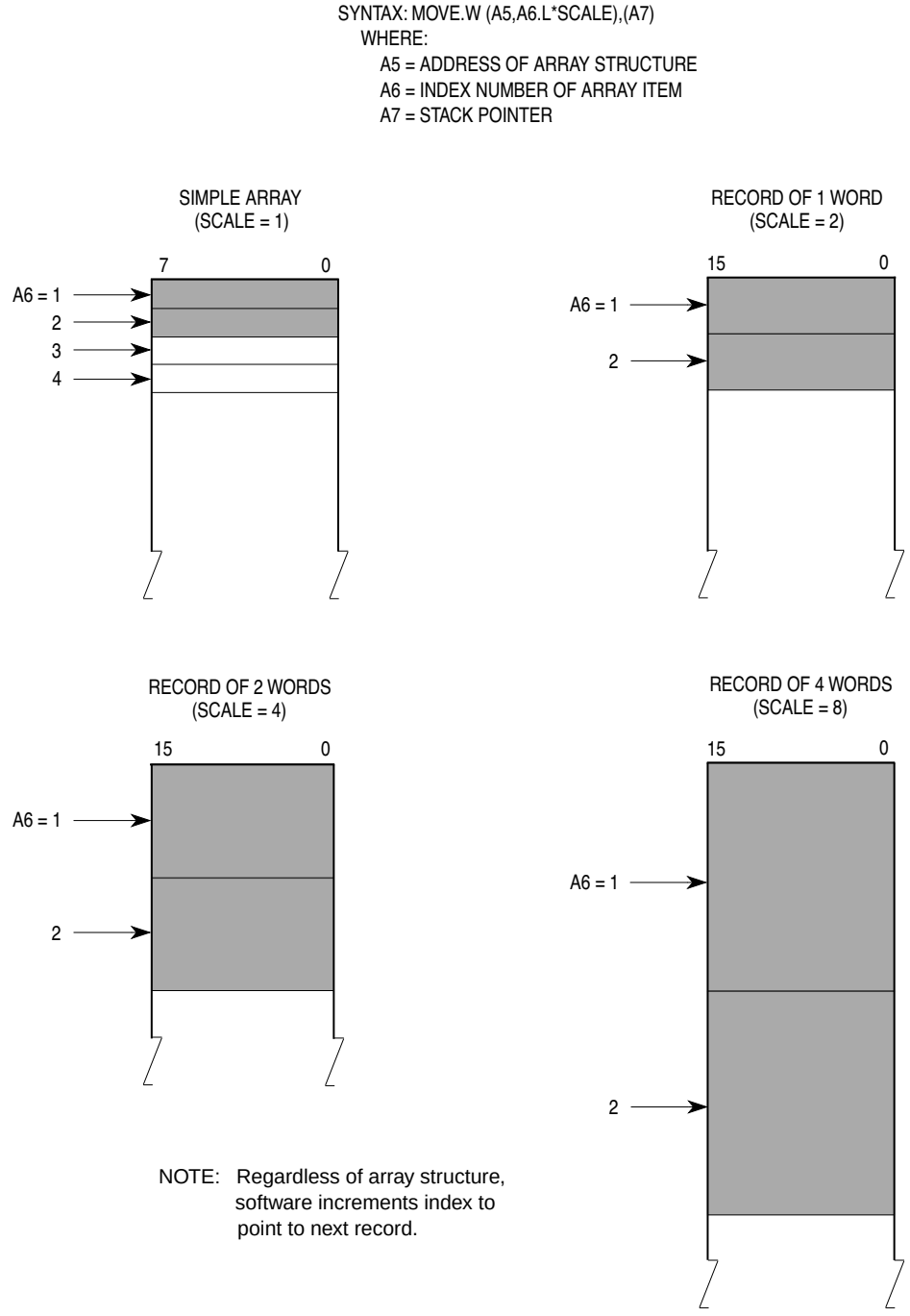


Figure 3-5 Addressing Array Items

3.5.2 General Addressing Mode Summary

The addressing modes described in the previous paragraphs 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 address register indirect if the register is an address register. If Rn is a data register, the assembler uses address register indirect with index mode, with a data register as the indirect register, and suppresses the address register by setting the base suppress bit in the EA 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 based 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 CPU32 EA mode actually encoded) because the programmer need not be concerned about these decisions. The assembler can choose the more efficient addressing mode to encode.

3.6 M68000 Family Addressing Capability

Programs can be easily transported from one member of the M68000 Family to another. The user object code of earlier members of the family is upwardly compatible with later members and can be executed without change. The address extension word(s) are encoded with information that allows the CPU32 to distinguish new additions to the basic M68000 Family architecture.

Earlier microprocessors have no knowledge of extension word formats implemented in later processors, and, while they do detect illegal instructions, they do not decode invalid encodings of the extension words as exceptions.

Address extension words for the early MC68000, MC68008, MC68010, and MC68020 microprocessors are shown in **Figure 3-6**.

MC6800/MC68008/MC68010 ADDRESS EXTENSION WORD

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

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

CPU32/MC68020 EXTENSION WORD

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

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 3-6 M68000 Family Address Extension Words

The encoding for SCALE used by the CPU32 and the MC68020 is a compatible extension of the M68000 architecture. A value of zero for SCALE is the same encoding for both extension words; thus, 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; therefore, 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 would not access the desired memory address.

3.7 Other Data Structures

In addition to supporting the array data structure with the index addressing mode, M68000 processors also support stack and queue data structures with the address register indirect postincrement and predecrement addressing modes. A stack is a last-in-first-out (LIFO) list; a queue is a first-in-first-out (FIFO) list. When data is added to a stack or queue, it is pushed onto the structure; when it is removed, it is “popped”, or pulled, from the structure. The system stack is used implicitly by many instructions; user stacks and queues may be created and maintained through use of addressing modes.

3.7.1 System Stack

Address register 7 (A7) is the system stack pointer (SP). The SP is either the supervisor stack pointer (SSP) or the user stack pointer (USP), depending on the state of the S bit in the status register. If the S bit indicates the supervisor state, the SSP is the SP, and the USP cannot be referenced as an address register. If the S bit indicates the user state, the USP is the active SP, and the SSP cannot be referenced. Each system

stack fills from high memory to low memory. The address mode –(SP) creates a new item on the active system stack, and the address mode (SP)+ deletes an item from the active system stack.

The program counter is saved on the active system stack on subroutine calls and is restored from the active system stack on returns. On the other hand, both the program counter and the status register are saved on the supervisor stack during the processing of traps and interrupts. Thus, the correct execution of the supervisor state code is not dependent on the behavior of user code, and user programs may use the USP arbitrarily.

To keep data on the system stack aligned properly, data entry on the stack is restricted so that data is always put in the stack on a word boundary. Thus, byte data is pushed on or pulled from the system stack in the high-order half of the word; the low-order half is unchanged.

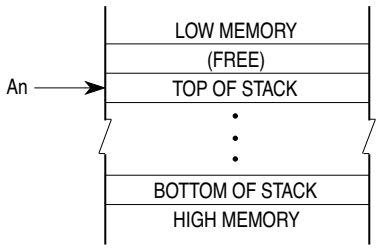
3.7.2 User Stacks

The user can implement stacks with the address register indirect with postincrement and predecrement addressing modes. With address register An (n = 0 to 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 SP correctly when byte, word, and long-word items are mixed in these stacks.

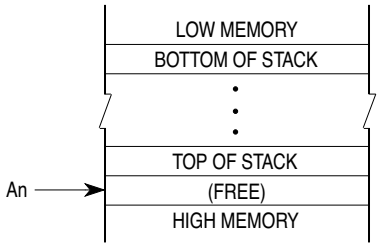
To implement stack growth from high to low memory, use –(An) to push data on the stack, (An)+ to pull data from the stack.

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



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 scheme is illustrated as follows:

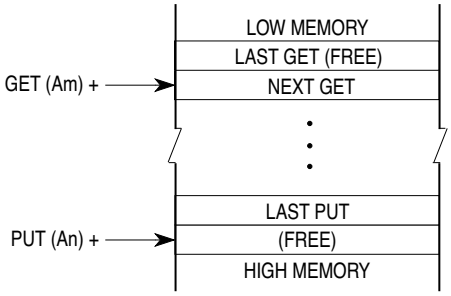


3.7.3 Queues

Queues can be implemented using the address register indirect with postincrement or predecrement addressing modes. Queues are pushed from one end and pulled from the other, and use two registers. A queue filled either from high to low memory or from low to high memory can be implemented with a pair (two of A0 to A6) of address registers. (An) is the “put” pointer and (Am) is 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.

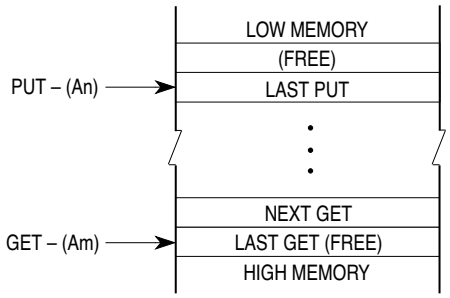
After a “put” operation, the “put” register points to the next available queue space, and the unchanged “get” register points to the next item to be removed from the queue. After a “get” operation, the “get” register points to the next item to be removed from the queue, and the unchanged “put” register points to the next available queue space, which is illustrated as follows:



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

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” 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” register points to the last item removed from the queue, and the unchanged “put” register points to the last item placed in the queue, which is illustrated as follows:



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 (if necessary) adjusted. The address register is adjusted by adding the buffer length (in bytes) to the register contents.

SECTION 4 INSTRUCTION SET

This section describes the set of instructions provided in the CPU32 and demonstrates their use. Descriptions of the instruction format and the operands used by instructions are included. After a summary of the instructions by category, a detailed description of each instruction is listed in alphabetical order. Complete programming information is provided, as well as a description of condition code computation and an instruction format summary.

The CPU32 instructions include machine functions for all the following operations:

- Data movement
- Arithmetic operations
- Logical operations
- Shifts and rotates
- Bit manipulation
- Conditionals and branches
- System control

The large instruction set encompasses a complete range of capabilities and, combined with the enhanced addressing modes, provides a flexible base for program development.

4.1 M68000 Family Compatibility

It is the philosophy of the M68000 Family that all user-mode programs can execute unchanged on a more advanced processor and that supervisor-mode programs and exception handlers should require only minimal alteration.

The CPU32 can be thought of as an intermediate member of the M68000 Family. Object code from an MC68000 or MC68010 may be executed on the CPU32, and many of the instruction and addressing mode extensions of the MC68020 are also supported.

4.1.1 New Instructions

Two instructions have been added to the M68000 instruction set for use in controller applications. These are the low-power stop (LPSTOP) and the table lookup and interpolation (TBL) commands.

4.1.1.1 Low-Power Stop (LPSTOP)

In applications where power consumption is a consideration, the CPU32 can force the device into a low-power standby mode when immediate processing is not required. The low-power mode is entered by executing the LPSTOP instruction. The processor remains in this mode until a user-specified or higher level interrupt, or a reset, occurs.

4.1.1.2 Table Lookup and Interpolation (TBL)

To maximize throughput for real-time applications, reference data is often precalculated and stored in memory for quick access. The storage of sufficient data points can require an inordinate amount of memory. The TBL instruction uses linear interpolation to recover intermediate values from a sample of data points, and thus conserves memory.

When the TBL instruction is executed, the CPU32 looks up two table entries bounding the desired result and performs a linear interpolation between them. Byte, word, and long-word operand sizes are supported. The result can be rounded according to a round-to-nearest algorithm, or returned unrounded along with the fractional portion of the calculated result (byte and word results only). This extra “precision” can be used to reduce cumulative error in complex calculations. See **4.6 Table Lookup and Interpolation Instructions** for examples.

4.1.2 Unimplemented Instructions

The ability to trap on unimplemented instructions allows user-supplied code to emulate unimplemented capabilities or to define special-purpose functions. However, Motorola reserves the right to use all currently unimplemented instruction operation codes for future M68000 enhancements. See **6.2.8 Illegal or Unimplemented Instructions** for more details.

4.2 Instruction Format

All instructions consist of at least one word. Some instructions can have as many as seven words, as shown in **Figure 4-1**. The first word of the instruction, called the operation word, specifies instruction length and the operation to be performed. The remaining words, called extension words, further specify the instruction and operands. These words may be immediate operands, extensions to the effective address mode specified in the operation word, branch displacements, bit number, special register specifications, trap operands, or argument counts.

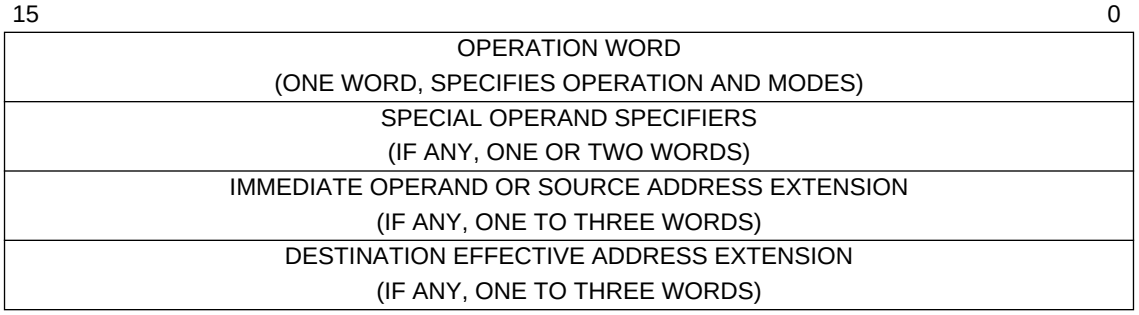


Figure 4-1 Instruction Word General Format

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:

- Register specification — A register field of the instruction contains the number of the register.
- Effective address — An effective address field of the instruction contains address mode information.
- 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 is an address or data register and how it is to be used. **SECTION 3 DATA ORGANIZATION AND ADDRESSING CAPABILITIES** contains detailed register information.

4.2.1 Notation

Except where noted, the following notation is used in this section:

Data	Immediate data from an instruction
Destination	Destination contents
Source	Source contents
Vector	Location of exception vector
An	Any address register (A7 to A0)
Ax, Ay	Address registers used in computation
Dn	Any data register (D7 to D0)
Rc	Control register (VBR, SFC, DFC)
Rn	Any address or data register
Dh, Dl	Data registers, high and low order 32 bits of product
Dr, Dq	Data registers, division remainder, division quotient
Dx, Dy	Data registers, used in computation
Dym, Dyn	Data registers, table interpolation values
Xn	Index register
[An]	Address extension
cc	Condition code
$d_{\#}$	Displacement Example: d_{16} is a 16-bit displacement
⟨ea⟩	Effective address
#⟨data⟩	Immediate data; a literal integer
label	Assembly program label
list	List of registers Example: D3–D0
[...]	Bits of an operand Examples: [7] is bit 7; [31:24] are bits 31 to 24
(...)	Contents of a referenced location Example: (Rn) refers to the contents of Rn

CCR	Condition code register (lower byte of status register) X — extend bit N — negative bit Z — zero bit V — overflow bit C — carry bit
PC	Program counter
SP	Active stack pointer
SR	Status register
SSP	Supervisor stack pointer
USP	User stack pointer
FC	Function code
DFC	Destination function code register
SFC	Source function code register
+	Addition or post increment
–	Subtraction or predecrement
/	Division or conjunction
*	Multiplication
=	Equal to
≠	Not equal to
>	Greater than
≥	Greater than or equal to
<	Less than
≤	Less than or equal to
•	Boolean AND
+	Boolean OR
⊕	Boolean XOR (exclusive OR)
$\overline{\text{not}}$	Boolean complement (operand is inverted)
BCD	Binary coded decimal, indicated by subscript Example: Source_{10} is a BCD source operand.
LSW	Least significant word
MSW	Most significant word
{R/W}	Read/write indicator

In description of an operation, a destination operand is placed to the right of source operands, and is indicated by an arrow (→).

4.3 Instruction Summary

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

Data movement	Bit manipulation
Integer arithmetic	Binary-coded decimal arithmetic
Logic	Program control
Shift and rotate	System control

The complete range of instruction capabilities combined with the addressing modes described previously provide flexibility for program development.

4.3.1 Condition Code Register

The condition code register portion of the status register contains five bits that indicate the result of a processor operation. **Table 4-1** lists the effect of each instruction on these bits. The carry bit and the multiprecision extend bit are separate in the M68000 Family to simplify programming techniques that use them. Refer to **Table 4-5** as an example.

Table 4-1 Condition Code Computations

Operations	X	N	Z	V	C	Special Definition
ABCD	*	U	?	U	?	C = Decimal Carry Z = Z • $\overline{Rm}$ •... • $\overline{R0}$
ADD, ADDI, ADDQ	*	*	*	?	?	V = Sm • Dm • $\overline{Rm}$ + Sm • Dm • Rm C = Sm • Dm; $\overline{Rm}$ • Dm + Sm • $\overline{Rm}$
ADDX	*	*	?	?	?	V = Sm • Dm • $\overline{Rm}$ + Sm • Dm • Rm C = Sm • Dm + $\overline{Rm}$ • Dm + Sm • $\overline{Rm}$ Z = Z • $\overline{Rm}$ •... • $\overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT, NOT, TAS, TST	—	*	*	0	0	
CHK	—	*	U	U	U	
CHK2, CMP2	—	U	?	U	?	Z = (R = LB) + (R = UB) C = (LB UB) • (IR < LB) + (R > UB) + (UB < LB) • (R > UB) • (R < LB)
SUB, SUBI, SUBQ	*	*	*	?	?	V = Sm • Dm • $\overline{Rm}$ + Sm • Dm • Rm C = Sm • Dm + Rm • Dm + Sm • Rm
SUBX	*	*	?	?	?	V = Sm • Dm • $\overline{Rm}$ + Sm • Dm • Rm C = Sm • Dm + Rm • Dm + Sm • Rm Z = Z • $\overline{Rm}$ •... • $\overline{R0}$
CMP, CMPI, CMPM	—	*	*	?	?	V = Sm • Dm • $\overline{Rm}$ + Sm • Dm • Rm C = Sm • Dm + Rm • Dm + Sm • Rm
DIVS, DIVU	—	*	*	?	0	V = Division Overflow
MULS, MULU	—	*	*	?	0	V = Multiplication Overflow
SBCD, NBCD	*	U	?	U	?	C = Decimal Borrow Z = Z • $\overline{Rm}$ •... • $\overline{R0}$
NEG	*	*	*	?	?	V = Dm • Rm C = Dm + Rm
NEGX	*	*	?	?	?	V = Dm • Rm C = Dm + Rm Z = Z • $\overline{Rm}$ •... • $\overline{R0}$

Table 4-1 Condition Code Computations (Continued)

Operations	X	N	Z	V	C	Special Definition
ASL	*	*	*	?	?	V = Dm • (Dm – 1 + ... + Dm – r) + Dm • (Dm-1 +...+ Dm – r) C = $\overline{Dm - r + 1}$
ASL (r = 0)		*	*	0	0	
LSL, ROXL	*	*	*	0	?	C = Dm – r + 1
LSR (r = 0)	—	*	*	0	0	
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	

Note: The following notation applies to this table only.

—	Not affected	Sm	Source operand MSB
U	Undefined	Dm	Destination operand MSB
?	See special definition	Rm	Result operand MSB
*	General case	R	Register tested
	X = C	r	Shift count
	N = Rm	LB	Lower bound
	Z = $\overline{Rm}$ •... • $\overline{R0}$	UB	Upper bound

4.3.2 Data Movement Instructions

The MOVE instruction is the basic means of transferring and storing address 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 4-2** is a summary of the data movement operations.

Table 4-2 Data Movement Operations

Instruction	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	⟨ea⟩, ⟨ea⟩	8, 16, 32	Source → Destination
MOVEA	⟨ea⟩, An	16, 32 → 32	Source → Destination

Table 4-2 Data Movement Operations

Instruction	Syntax	Operand Size	Operation
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

4.3.3 Integer Arithmetic Operations

The 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 dividend 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 extend (EXT), and negate binary with extend (NEGX). Refer to **Table 4-3** for a summary of the integer arithmetic operations.

Table 4-3 Integer Arithmetic Operations

Instruction	Syntax	Operand Size	Operation
ADD	Dn, ⟨ea⟩ ⟨ea⟩, Dn	8, 16, 32 8, 16, 32	Source + Destination → Destination
ADDA	⟨ea⟩, An	16, 32	Source + Destination → Destination
ADDI	#⟨data⟩, ⟨ea⟩	8, 16, 32	Immediate data + Destination → Destination
ADDQ	#⟨data⟩, ⟨ea⟩	8, 16, 32	Immediate data + Destination → Destination
ADDX	Dn, Dn – (An), – (An)	8, 16, 32 8, 16, 32	Source + Destination + X → Destination
CLR	⟨ea⟩	8, 16, 32	0 → Destination
CMP	⟨ea⟩, Dn	8, 16, 32	(Destination – Source), CCR shows results
CMPA	⟨ea⟩, An	16, 32	(Destination – Source), CCR shows results
CMPI	#⟨data⟩, ⟨ea⟩	8, 16, 32	(Destination – Data), CCR shows results
CMPM	(An) +, (An) +	8, 16, 32	(Destination – Source), CCR shows results

Table 4-3 Integer Arithmetic Operations

Instruction	Syntax	Operand Size	Operation
CMP2	⟨ea⟩, Rn	8, 16, 32	Lower bound Rn Upper bound, CCR shows result
DIVS/DIVU	⟨ea⟩, Dn	32/16 → 16 : 16	Destination / Source → Destination (signed or unsigned)
DIVSL/DIVUL	⟨ea⟩, Dr : Dq ⟨ea⟩, Dq ⟨ea⟩, Dr : Dq	64/32 → 32 : 32 32/32 → 32 32/32 → 32 : 32	Destination / Source → Destination (signed or unsigned)
EXT	Dn Dn	8 → 16 16 → 32	Sign extended Destination → Destination
EXTB	Dn	8 → 32	Sign extended Destination → Destination
MULS/MULU	⟨ea⟩, Dn ⟨ea⟩, Dl ⟨ea⟩, Dh : Dl	16 ∗ 16 → 32 32 ∗ 32 → 32 32 ∗ 32 → 64	Source ∗ Destination → Destination (signed or unsigned)
NEG	⟨ea⟩	8, 16, 32	0 – Destination → Destination
NEGX	⟨ea⟩	8, 16, 32	0 – Destination – X → Destination
SUB	⟨ea⟩, Dn Dn, ⟨ea⟩	8, 16, 32	Destination – Source → Destination
SUBA	⟨ea⟩, An	16, 32	Destination – Source → Destination
SUBI	#⟨data⟩, ⟨ea⟩	8, 16, 32	Destination – Data → Destination
SUBQ	#⟨data⟩, ⟨ea⟩	8, 16, 32	Destination – Data → Destination
SUBX	Dn, Dn – (An), – (An)	8, 16, 32 8, 16, 32	Destination – Source – X → Destination
TBLS/TBLU	⟨ea⟩, Dn Dym : Dyn, Dn	8, 16, 32	Dyn – Dym → Temp (Temp ∗ Dn [7 : 0]) → Temp (Dym ∗ 256) + Temp → Dn
TBLSN/TBLUN	⟨ea⟩, Dn Dym : Dyn, Dn	8, 16, 32	Dyn – Dym → Temp (Temp ∗ Dn [7 : 0]) / 256 → Temp Dym + Temp → Dn

4.3.4 Logic 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 arithmetically compares the operand with zero, placing the result in the condition code register. **Table 4-4** summarizes the logical operations.

Table 4-4 Logic Operations

Instruction	Syntax	Operand Size	Operation
AND	⟨ea⟩, Dn Dn, ⟨ea⟩	8, 16, 32 8, 16, 32	Source • Destination → Destination
ANDI	#⟨data⟩, ⟨ea⟩	8, 16, 32	Data • Destination → Destination
EOR	Dn, ⟨ea⟩	8, 16, 32	Source ⊕ Destination → Destination
EORI	#⟨data⟩, ⟨ea⟩	8, 16, 32	Data ⊕ Destination → Destination
NOT	⟨ea⟩	8, 16, 32	$\overline{\text{Destination}}$ → Destination
OR	⟨ea⟩, Dn Dn, ⟨ea⟩	8, 16, 32 8, 16, 32	Source + Destination → Destination
ORI	#⟨data⟩, ⟨ea⟩	8, 16, 32	Data + Destination → Destination
TST	⟨ea⟩	8, 16, 32	Source – 0, to set condition codes

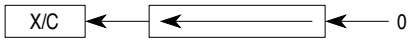
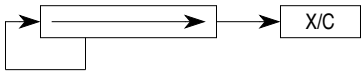
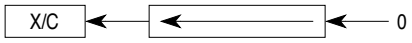
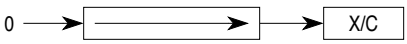
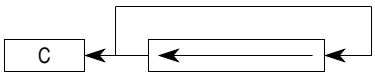
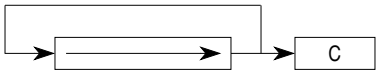
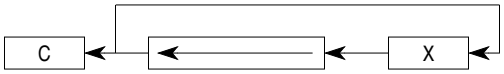
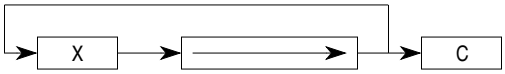
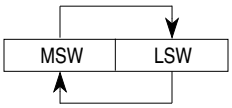
4.3.5 Shift and Rotate Instructions

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 to 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 4-5** is a summary of the shift and rotate operations.

Table 4-5 Shift and Rotate Operations

Instruction	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	16	

4.3.6 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 mem-

ory. 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. **Table 4-6** is a summary of bit manipulation instructions.

Table 4-6 Bit Manipulation Operations

Instruction	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

4.3.7 Binary-Coded Decimal (BCD) Instructions

Five instructions support operations on 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). **Table 4-7** is a summary of the BCD operations.

Table 4-7 Binary-Coded Decimal Operations

Instruction	Syntax	Operand Size	Operation
ABCD	Dn, Dn – (An), – (An)	8 8	$Source_{10}$ + $Destination_{10}$ + X → Destination
NBCD	⟨ea⟩	8 8	0 – $Destination_{10}$ – X → Destination
SBCD	Dn, Dn – (An), – (An)	8 8	$Destination_{10}$ – $Source_{10}$ – X → Destination

4.3.8 Program Control Instructions

A set of subroutine call and return instructions and conditional and unconditional branch instructions perform program control operations. **Table 4-8** summarizes these instructions.

Table 4-8 Program Control Operations

Instruction	Syntax	Operand Size	Operation
Conditional			
Bcc	⟨label⟩	8, 16, 32	If condition true, then PC + d → PC
DBcc	Dn, ⟨label⟩	16	If condition false, then Dn – 1 → PC; if Dn ≠ (– 1), then PC + d → PC
Scc	⟨ea⟩	8	If condition true, then destination bits are set to one; else, destination bits are cleared to zero
Unconditional			
BRA	⟨label⟩	8, 16, 32	PC + d → PC
BSR	⟨label⟩	8, 16, 32	SP – 4 → SP; PC → (SP); PC + d → PC

Table 4-8 Program Control Operations

Instruction	Syntax	Operand Size	Operation
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

To specify conditions for change in program control, condition codes must be substituted for the letters “cc” in conditional program control opcodes. Condition test mnemonics are given below. Refer to **4.3.10 Condition Tests** for detailed information on condition codes.

CC—Carry clear
CS—Carry set
EQ—Equal
F—False*
GE—Greater or equal
GT—Greater than
HI—High
LE—Less or equal
LS—Low or same
LT—Less than
MI—Minus
NE—Not equal
PL—Plus
T—True
VC—Overflow clear
VS—Overflow set

*Not applicable to the Bcc instruction

4.3.9 System Control Instructions

Privileged instructions, trapping instructions, and instructions that use or modify the condition code register provide system control operations. All of these instructions cause the processor to flush the instruction pipeline. **Table 4-9** summarizes the instructions. The preceding list of condition tests also applies to the TRAPcc instruction. Refer to **4.3.10 Condition Tests** for detailed information on condition codes.

Table 4-9 System Control Operations

Instruction	Syntax	Size	Operation
Privileged			
ANDI	#⟨data⟩, SR	16	Data • SR → SR
EORI	#⟨data⟩, SR	16	Data ⊕ SR → SR
MOVE	⟨ea⟩, SR SR, ⟨ea⟩	16 16	Source → SR SR → Destination
MOVEA	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	Data + SR → SR

Table 4-9 System Control Operations (Continued)

Instruction	Syntax	Size	Operation
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	Data → SR; STOP
LPSTOP	#⟨data⟩	none	Data → SR; interrupt mask → EBI; STOP
Trap Generating			
BKPT	#⟨data⟩	none	If breakpoint cycle acknowledged, then execute returned operation word, else trap as illegal instruction.
BGND	none	none	If background mode enabled, then enter background mode, else format/vector offset → – (SSP); PC → ⟩ (SSP); SR → ⟩ (SSP); (vector) → PC
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, then 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/vector offset → (SSP); SSP – 4 → SSP; PC → (SSP); SR → (SSP); vector address → PC
TRAPcc	none #⟨data⟩	none 16, 32	If cc true, then TRAP exception
TRAPV	none	none	If V set, then overflow TRAP exception
Condition Code Register			
ANDI	#⟨data⟩, CCR	8	Data • CCR → CCR
EORI	#⟨data⟩, CCR	8	Data ⊕ CCR → CCR
MOVE	⟨ea⟩, CCR CCR, ⟨ea⟩	16 16	Source → CCR CCR → Destination
ORI	#⟨data⟩, CCR	8	Data + CCR → CCR

4.3.10 Condition Tests

Conditional program control instructions and the TRAPcc instruction execute on the basis of condition tests. A condition test is the evaluation of a logical expression related to the state of the CCR bits. If the result is one, the condition is true. If the result is 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 true. **Table 4-10** lists each condition test.

Table 4-10 Condition Tests

Mnemonic	Condition	Encoding	Test
T	True	0000	1
F*	False	0001	0
HI	High	0010	$\overline{C} \bullet \overline{Z}$
LS	Low or Same	0011	$\overline{C} + \overline{Z}$
CC	Carry Clear	0100	$\overline{C}$
CS	Carry Set	0101	C

Table 4-10 Condition Tests (Continued)

Mnemonic	Condition	Encoding	Test
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	$N \bullet V + \overline{N} \bullet \overline{V}$
LT	Less Than	1101	$N \bullet \overline{V} + \overline{N} \bullet V$
GT	Greater Than	1110	$N \bullet V \bullet \overline{Z} + \overline{N} \bullet \overline{V} \bullet \overline{Z}$
LE	Less or Equal	1111	$Z; N \bullet \overline{V}; \overline{N} \bullet V$

* Not available for the Bcc instruction.

4.4 Instruction Details

The following paragraphs contain detailed information about each instruction in the CPU32 instruction set. The instruction descriptions are arranged alphabetically by instruction mnemonic. **Figure 4-2** shows the format of the instruction descriptions. **4.2.1 Notation** applies, with the following additions.

A. The attributes line specifies the size of the operands of an instruction. When an instruction can use operands of more than one size, a suffix is used with the mnemonic of the instruction:

.B	Byte
.W	Word
.L	Long word

B. In instruction set descriptions, changes in CCR bits are shown as follows:

*	Set according to result of operation
—	Not affected by operation
0	Cleared
1	Set
U	Undefined after operation

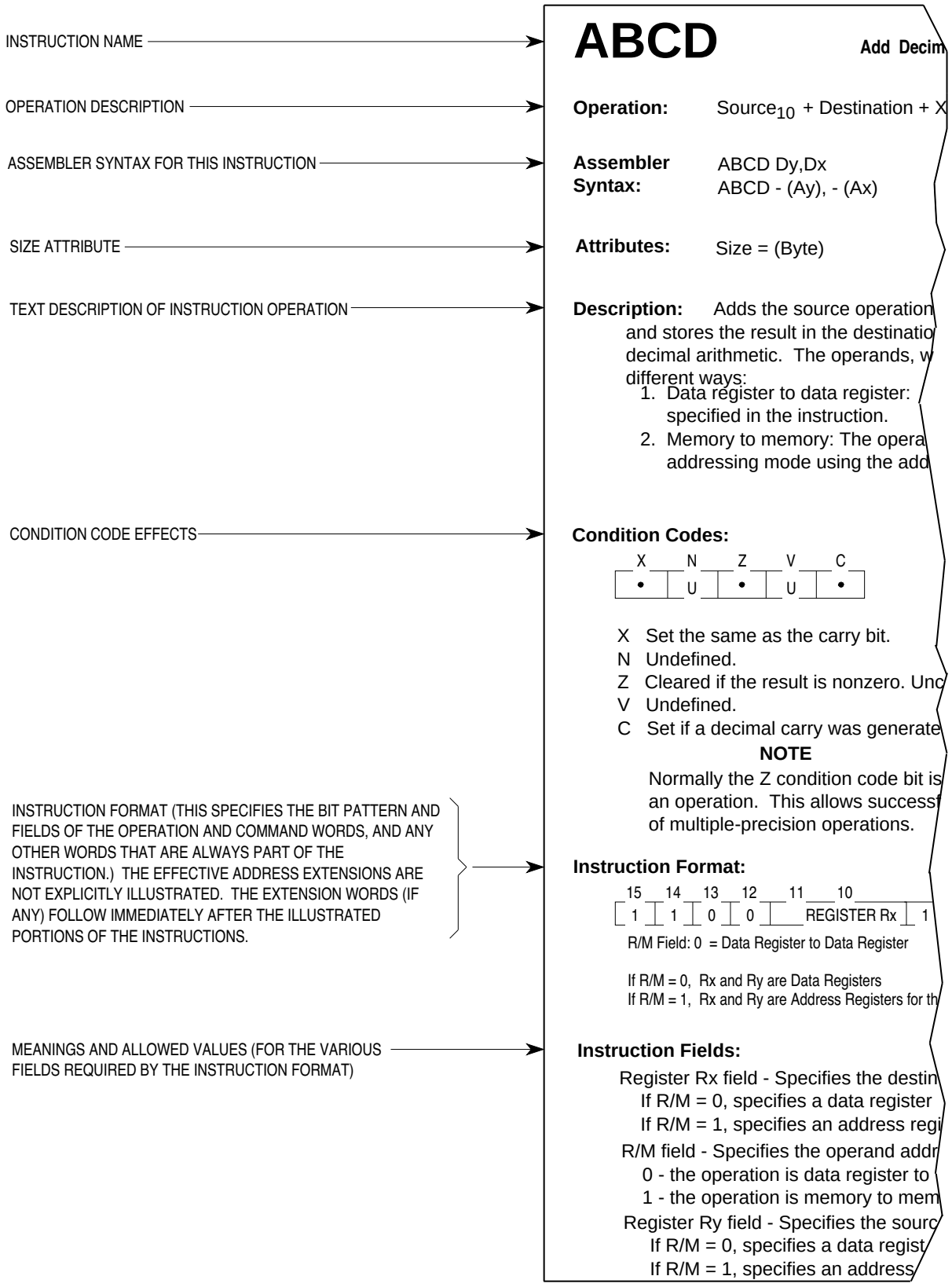


Figure 4-2 Instruction Description Format

ABCD — Add Decimal with Extend — ABCD

Operation: $Source_{10} + Destination_{10} + X \rightarrow Destination$

Assembler Syntax: ABCD Dy, Dx
ABCD – (Ay), – (Ax)

Attributes: Size = (Byte)

Description: Adds the source operand to the destination operand along with the extend bit, and stores the result in the destination location. The addition is performed using binary coded decimal arithmetic. The operands, which are packed BCD numbers, can be addressed in two different ways:

1. Data register to data register — Operands are contained in data registers specified by the instruction.
2. Memory to memory — Operands are addressed with the predecrement addressing mode using address registers specified by the instruction.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

X Set the same as the carry bit.
N Undefined.
Z Cleared if the result is nonzero. Unchanged otherwise.
V Undefined.
C Set if a decimal carry was generated. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	1	0	0	REGISTER Rx	1	0	0	0	0	R/M	REGISTER Ry

Instruction fields:

Register Rx field — Specifies the destination register:
If R/M = 0, specifies a data register
If R/M = 1, specifies an address register for predecrement addressing mode
R/M field — Specifies the operand addressing mode:
0 — the operation is data register to data register
1 — the operation is memory to memory
Register Ry field — Specifies the source register:
If R/M = 0, specifies a data register
If R/M = 1, specifies an address register for predecrement addressing mode

ADD — Add — ADD

Operation: Source + Destination → Destination

Assembler: ADD ⟨ ea⟩, Dn

Syntax: ADD Dn, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

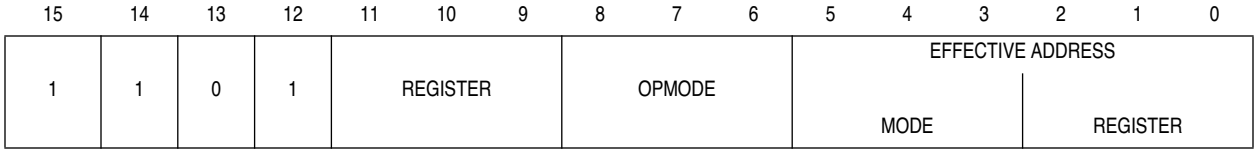
Description: Adds the source operand to the destination operand using binary addition, and stores the result in the destination location. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set the same as the carry bit.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow is generated. Cleared otherwise.
- C Set if a carry is generated. Cleared otherwise.

Instruction Format:



Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	⟨ea⟩ + ⟨Dn⟩ → ⟨Dn⟩
100	101	110	⟨Dn⟩ + ⟨ea⟩ → ⟨ea⟩

Effective Address Field — Determines addressing mode:
If the location specified is a source operand, all addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An*	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*Word and long word only

If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTES:
1. Dn mode is used when destination is a data register. Destination ⟨ea⟩ mode is invalid for a data register.
2. ADDA is used when the destination is an address register. ADDI and ADDQ are used when the source is immediate data. Most assemblers automatically make this distinction.

ADDA — Add Address — ADDA

Operation: Source + Destination → Destination

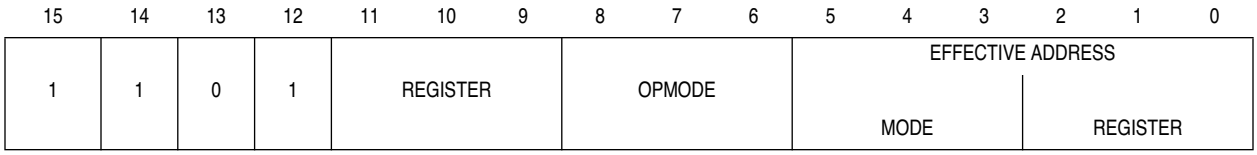
Assembler Syntax: ADDA ⟨ea⟩ An

Attributes: Size = (Word, Long)

Description: Adds the source operand to the destination address register and stores the result in the address register. The entire destination address register is used regardless of the operation size.

Condition Codes:
Not affected

Instruction Format:



Instruction Fields:

Register field — Specifies any of the eight address registers. This is always the destination.

Opmode field — Specifies the size of the operation:

011 — Word operation. The source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.

111 — Long operation.

Effective Address field — Specifies source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

ADDI — Add Immediate — ADDI

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ADDI #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

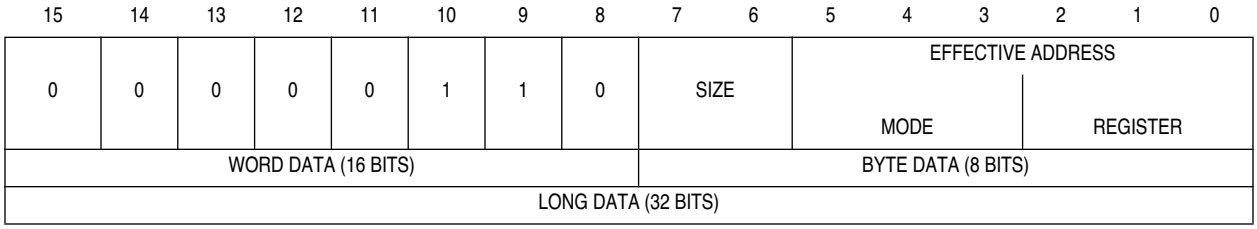
Description: Adds the immediate data to the destination operand, and stores the result in the destination location. The size of the immediate data must match the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set the same as the carry bit.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow is generated. Cleared otherwise.
- C Set if a carry is generated. Cleared otherwise.

Instruction Format:



Instruction Fields:

Size field — Specifies the size of the operation:

- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand.
Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
- (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

Immediate field — (Data immediately following the instruction):
If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

ADDQ — Add Quick — ADDQ

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ADDQ #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Adds an immediate value in the range (1–8) to the operand at the destination location. Word and long operations are allowed on the address registers. When adding to address registers, the condition codes are not altered, and the entire destination address register is used, regardless of the operation size.

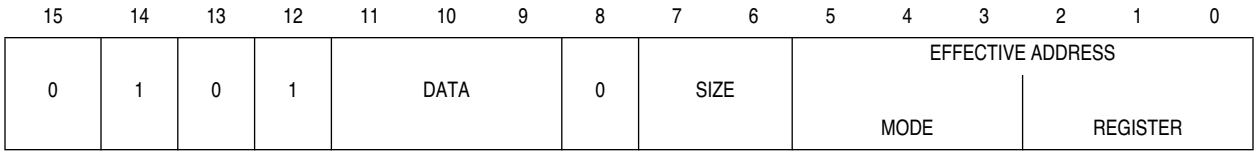
Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set the same as the carry bit.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow occurs. Cleared otherwise.
- C Set if a carry occurs. Cleared otherwise.

The condition codes are not affected when the destination is an address register.

Instruction Format:



ADDQ Add Quick ADDQ

Instruction Fields:

Data field — Three bits of immediate data, (9–11), with 0 representing a value of 8).

Size field — Specifies the size of the operation:

00 — Byte operation

01 — Word operation

10 — Long operation

Effective Address field — Specifies the destination location.

Only alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An*	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*Word and long only

ADDX — Add Extended — ADDX

Operation: Source + Destination + X → Destination

Assembler Syntax: ADDX Dy, Dx
ADDX – (Ay), – (Ax)

Attributes: Size = (Byte, Word, Long)

Description: Adds the source operand to the destination operand along with the extend bit and stores the result in the destination location. The operands can be addressed in two ways:

1. Data register to data register: Data registers specified by the instruction contain the operands.
2. Memory to memory: Address registers specified by the instruction address the operands using the predecrement addressing mode.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit.
N Set if the result is negative. Cleared otherwise.
Z Cleared if the result is nonzero. Unchanged otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a carry is generated. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Instruction Format:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	0	1	REGISTER Rx	1	SIZE	0	0	R/M	REGISTER Ry

ADDX Add Extended ADDX

Instruction Fields:

Register Rx field — Specifies the destination register:
- If R/M = 0, specifies a data register.
- If R/M = 1, specifies an address register for predecrement addressing mode.

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

R/M field — Specifies the operand address mode:
- 0 — The operation is data register to data register.
- 1 — The operation is memory to memory.

Register Ry field — Specifies the source register:
- If R/M = 0, specifies a data register.
- If R/M = 1, specifies an address register for predecrement addressing mode.

AND — Logical AND — AND

Operation: Source • Destination → Destination

Assembler Syntax: AND ⟨ea⟩,Dn
AND Dn, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

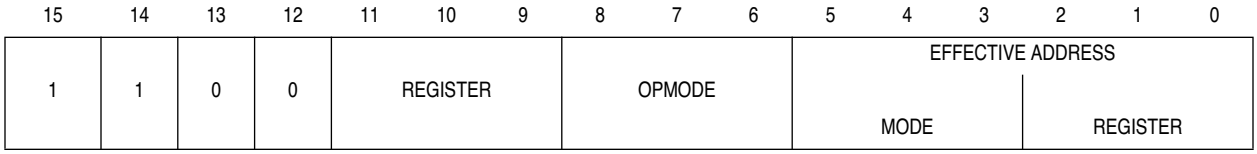
Description: Performs an AND operation of the source operand with the destination operand and stores the result in the destination location. The contents of an address register may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:



Instruction Fields:

Register field — Specifies any of the eight data registers.
Opmode field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) •(⟨Dn⟩) → Dn
100	101	110	(⟨Dn⟩) • (⟨ea⟩) → ea

Effective Address field — Determines addressing mode:

If the location specified is a source operand, only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTES:

1. The Dn mode is used when the destination is a data register; the destination ⟨ea⟩ mode is invalid for a data register.
2. Most assemblers use ANDI when the source is immediate data.

ANDI — AND Immediate — ANDI

Operation: Immediate Data • Destination → Destination

Assembler Syntax: ANDI #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an AND operation of the immediate data with the destination operand and stores the result in the destination location. The size of the immediate data must match the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand.
Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

Immediate field — (Data immediately following the instruction):
If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

ANDI to CCR

AND Immediate to Condition Code Register

ANDI to CCR

Operation: Source • CCR → CCR

Assembler Syntax: ANDI #⟨data⟩, CCR

Attributes: Size = (Byte)

Description: Performs an AND operation of the immediate operand with the condition codes and stores the result in the low-order byte of the status register.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Cleared if bit 4 of immediate operand is zero. Unchanged otherwise.
N Cleared if bit 3 of immediate operand is zero. Unchanged otherwise.
Z Cleared if bit 2 of immediate operand is zero. Unchanged otherwise.
V Cleared if bit 1 of immediate operand is zero. Unchanged otherwise.
C Cleared if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ANDI to SR

AND Immediate to the Status Register

(Privileged Instruction)

ANDI to SR

Operation: If supervisor state
then Source • SR →SR
else TRAP

Assembler Syntax: ANDI #⟨data⟩, SR

Attributes: Size = (Word)

Description: Performs an AND operation of the immediate operand with the contents of the status register and stores the result in the status register. All implemented bits of the status register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Cleared if bit 4 of immediate operand is zero. Unchanged otherwise.
N Cleared if bit 3 of immediate operand is zero. Unchanged otherwise.
Z Cleared if bit 2 of immediate operand is zero. Unchanged otherwise.
V Cleared if bit 1 of immediate operand is zero. Unchanged otherwise.
C Cleared if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
WORD DATA															

ASL, ASR — Arithmetic Shift — ASL, ASR

Operation: Destination Shifted by ⟨count⟩ → Destination

Assembler Syntax:
ASd Dx,Dy
ASd #⟨data⟩, Dy
ASd ⟨ea⟩
where d is direction, L or R

Attributes: Size = (Byte, Word, Long)

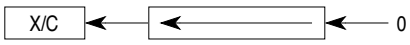
Description: Arithmetically shifts the bits of the operand in the direction (L or R) specified. The carry bit receives the last bit shifted out of the operand. The shift count for shifting a register may be specified in two ways:

1. Immediate — Shift count is specified by the instruction (shift range, 8–1).
2. Register — The shift count is the value in the data register specified by the instruction, modulo 64.

An operand in memory can be shifted one bit only, and the operand size is restricted to a word.

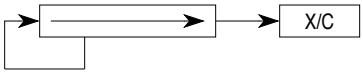
For ASL, the operand is shifted left; the number of positions shifted is the shift count. Bits shifted out of the high-order bit go to both the carry and the extend bits; zeros are shifted into the low-order bit. The overflow bit indicates if any sign changes occur during the shift.

ASL



For ASR, the operand is shifted right; the number of positions shifted is the shift count. Bits shifted out of the low-order bit go to both the carry and the extend bits; the sign-bit (MSB) is shifted into the high-order bit.

ASR



ASL, ASR — Arithmetic Shift — ASL, ASR

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set according to the last bit shifted out of the operand. Unaffected for a shift count of zero.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if the most significant bit is changed during the shift operation. Cleared otherwise.
- C Set according to the last bit shifted out of the operand. Cleared for a shift count of zero.

Instruction Format (Register Shifts):

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	COUNT/REGISTER	dr	SIZE	i/r	0	0	REGISTER

Instruction Fields (Register Shifts):

Count/Register field — Specifies shift count or register that contains shift count:

If i/r = 0, this field contains the shift count. The values one to seven represent counts of one to seven; value of zero represents a count of eight.

If i/r = 1, this field specifies the data register that contains the shift count (modulo 64).

dr field — Specifies the direction of the shift:

0 — Shift right

1 — Shift left

Size field — Specifies the size of the operation:

00 — Byte operation

01 — Word operation

10 — Long operation

i/r field:

If i/r = 0, specifies immediate shift count.

If i/r = 1, specifies register shift count.

Register field — Specifies a data register to be shifted.

ASL, ASR — Arithmetic Shift — ASL, ASR

Instruction Format (Memory Shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Shifts):

dr field — Specifies the direction of the shift:

0 — Shift right

1 — Shift left

Effective Address field — Specifies the operand to be shifted.

Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

Bcc — Branch Conditionally — Bcc

Operation: If (condition true) then PC+ d → PC

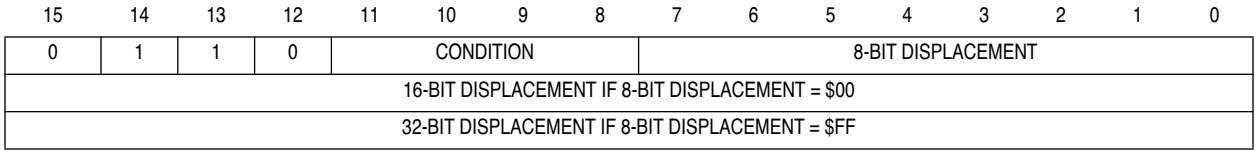
Assembler Syntax: Bcc ⟨label⟩Attributes:
Size = (Byte, Word, Long)

Description: If the specified condition is true, program execution continues at location (PC) + displacement. The PC contains the address of the instruction word of the Bcc instruction plus two. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC. If the 8-bit displacement field in the instruction word is zero, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones ($FF), the 32-bit displacement (long word immediately following the instruction) is used. Condition codes are specified as follows:

cc	Name	Code	Description	cc	Name	Code	Description
CC	Carry Clear	0100	$\overline{C}$	LS	Low or Same	0011	$\overline{C}$; $\overline{Z}$
CS	Carry Set	0101	C	LT	Less Than	1101	N • $\overline{V}$; $\overline{N}$ • V
EQ	Equal	0111	Z	MI	Minus	1011	N
GE	Greater or Equal	1100	N •V; $\overline{N}$ • $\overline{V}$	NE	Not Equal	0110	$\overline{Z}$
GT	Greater Than	1110	N • V • $\overline{Z}$; $\overline{N}$ • $\overline{V}$ • $\overline{Z}$	PL	Plus	1010	$\overline{N}$
HI	High	0010	$\overline{C}$ • $\overline{Z}$	VC	Overflow Clear	1000	$\overline{V}$
LE	Less or Equal	1111	Z; N • $\overline{V}$; $\overline{N}$ • V	VS	Overflow Set	1001	V

Condition Codes:
Not affected.

Instruction Format:



Instruction Fields:

Condition field — The binary code for one of the conditions listed in the table.

8-Bit Displacement field — Twos complement integer specifying the number of bytes between the branch instruction and the next instruction to be executed if the condition is met.

16-Bit Displacement field — Used for displacement when 8-bit displacement field contains$00.

32-Bit Displacement field — Used for displacement when 8-bit displacement field contains $FF.

NOTE

A branch to the instruction immediately following automatically uses 16-bit displacement because the 8-bit displacement field contains $00 (zero offset).

BCHG — Test a Bit and Change — BCHG

Operation: $\overline{(\langle \overline{\text{number}} \rangle \text{ of Destination})} \rightarrow Z$;
$\overline{(\langle \overline{\text{number}} \rangle \text{ of Destination})} \rightarrow \langle \text{bit number} \rangle$ of Destination

Assembler: BCHG Dn, ⟨ea⟩Syntax:
BCHG #⟨data⟩, ⟨ea⟩Attributes:
Size = (Byte, Long)

Description: Tests a specified bit in the destination operand, sets the Z condition code appropriately, then inverts the specified bit. When the destination is a data register, any of the 32 bits can be specified by the modulo 32 bit number. When the destination is a memory location, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation may be specified in either of two ways:

1. Immediate — The bit number is specified by a second instruction word
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

- X Not affected
- N Not affected
- Z Set if the bit tested is zero. Cleared otherwise
- V Not affected
- C Not affected

Instruction Format (Bit Number Static, specified as immediate data):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

BCHG — Test a Bit and Change — BCHG

Instruction Fields (Bit Number Static):

Bit Number field — Specifies the bit number.
Effective Address field — Specifies the destination location.
Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

*Long only; all others are byte only

Instruction Format (Bit Number Dynamic, specified in a register):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.
Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

*Long only; all others are byte only

BCLR — Test a Bit and Clear — BCLR

Operation: $\overline{(\langle \text{bit number} \rangle \text{ of Destination})} \rightarrow Z$;
$0 \rightarrow \langle \text{bit number} \rangle$ of Destination

Assembler Syntax: BCLR Dn, ⟨ea⟩
BCLR #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Long)

Description: Tests a specified bit in the destination operand, sets the Z condition code appropriately, then clears the bit. When a data register is the destination, any of the 32 bits can be specified by a modulo 32 bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. Immediate — The bit number is specified by a second instruction word.
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X Not affected
N Not affected
Z Set if the bit tested is zero. Cleared otherwise
V Not affected
C Not affected

Instruction Format (Bit Number Static, specified as immediate data):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

BCLR Test a Bit and Clear BCLR

Instruction Fields (Bit Number Static):

Bit Number field — Specifies the bit number.

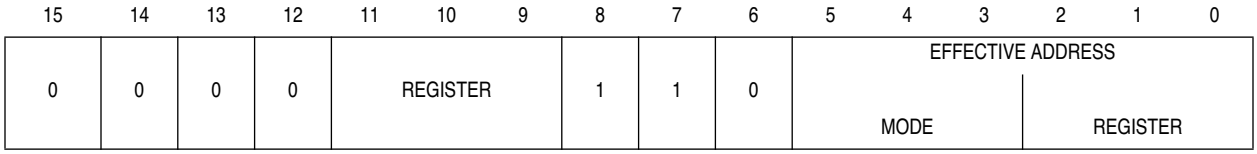
Effective Address field — Specifies the destination location.

Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

*Long only; all others are byte only

Instruction Format (Bit Number Dynamic, specified in a register):



Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

*Long only; all others are byte only

BGND — Enter Background Mode — BGND

Operation:
If (background mode enabled)
then enter Background Mode
else Format/Vector offset → – (SSP)
PC → – (SSP)
SR → – (SSP)
(Vector) → PC

Assembler Syntax: BGND

Attributes: Size = (Unsized)

Description: The processor suspends instruction execution and enters background mode (if enabled). The freeze output is asserted to acknowledge entrance into background mode. Upon exiting background mode, instruction execution continues with the instruction pointed to by the program counter.

If background mode is not enabled, the processor initiates illegal instruction exception processing. The vector number is generated to reference the illegal instruction exception vector. Background mode is covered in **SECTION 7 DEVELOPMENT SUPPORT**.

Condition Codes:

X	N	Z	V	C
—	—	—	—	—

X Not affected
N Not affected
Z Not affected
V Not affected
C Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	0	1	0

BKPT — Breakpoint — BKPT

Operation: Run breakpoint acknowledge cycle;
If acknowledged
then execute returned operation word
else TRAP as illegal instruction

Assembler Syntax: BKPT #⟨data⟩

Attributes: Unsized

Description: Executes a breakpoint acknowledge bus cycle. Bits [2:4] of the address bus are set to the value of the immediate data (0 to 7) and bits 0 and 1 of the address bus are set to 0.

The breakpoint acknowledge cycle accesses the CPU space, addressing type 0, and provides the breakpoint number specified by the instruction on address lines A4 to A2. If external hardware terminates the cycle with $\overline{\text{DSACKx}}$, the data on the bus (an instruction word) is inserted into the instruction pipe and is executed after the breakpoint instruction. The breakpoint instruction requires a word transfer — if the first bus cycle accesses an 8-bit port, a second cycle is required. If external logic terminates the breakpoint acknowledge cycle with BERR (i.e., no instruction word available) the processor takes an illegal instruction exception. Refer to **6.2.5 Software Breakpoints** for details of breakpoint operation.

This instruction supports breakpoints for debug monitors and real-time hardware emulators. The exact operation performed by the instruction is implementation-dependent. Typically, this instruction replaces an instruction in a program and the replaced instruction is returned by the breakpoint acknowledge cycle.

Condition Codes: Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	1	VECTOR		

Instruction Fields:

Vector field — Contains immediate data in the range (0–7). This is the breakpoint number.

BRA — Branch Always — BRA

Operation: PC + d → PC

Assembler Syntax: BRA ⟨label⟩

Attributes: Size = (Byte, Word, Long)

Description: Program execution continues at location (PC) + displacement. The PC contains the address of the instruction word of the BRA instruction plus two. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC. If the 8-bit displacement field in the instruction word is zero, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones ($FF), the 32-bit displacement (long word immediately following the instruction) is used.

Condition Codes: Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	8-BIT DISPLACEMENT							
16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00															
32-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $FF															

Instruction Fields:

8-Bit Displacement field — Twos complement integer specifying the number of bytes between the branch instruction and the next instruction to be executed.

16-Bit Displacement field — Used for a larger displacement when 8-bit displacement is $00.

32-Bit Displacement field — Used for a larger displacement when 8-bit displacement is $FF.

NOTE

A branch to the instruction immediately following automatically uses 16-bit displacement because the 8-bit displacement field contains $00 (zero offset).

BSET — Test a Bit and Set — BSET

Operation: $\overline{(\langle \text{bit number} \rangle \text{ of Destination})} \rightarrow Z$;
$1 \rightarrow \langle \text{bit number} \rangle$ of Destination

Assembler: BSET Dn, ⟨ea⟩Syntax:
BSET #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Long)

Description: Tests a bit in the destination operand, sets the Z condition code appropriately, then sets the specified bit in the destination operand. When a data register is the destination, any of the 32 bits can be specified by a modulo 32 bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in two ways:

1. Immediate — The bit number is specified by the second word of the instruction.
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X Not affected.
N Not affected
Z Set if the bit tested is zero. Cleared otherwise
V Not affected
C Not affected.

Instruction Format (Bit Number Static, specified as immediate data):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

BSET — Test a Bit and Set — BSET

Instruction Fields (Bit Number Static):

Bit Number field — Specifies the bit number.

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

*Long only; all others are byte only

Instruction Format (Bit Number Dynamic, specified in a register):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

*Long only; all others are byte only

BSR — Branch to Subroutine — BSR

Operation: SP – 4 → SP; PC → (SP); PC + d → PC

Assembler Syntax: BSR ⟨label⟩

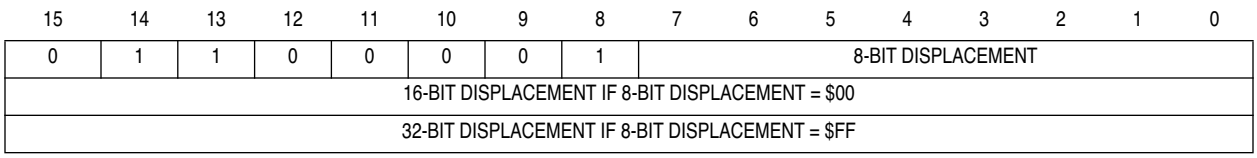
Attributes: Size = (Byte, Word, Long)

Description: Pushes the long word address of the instruction immediately following the BSR instruction onto the system stack. The PC contains the address of the instruction word plus two. Program execution then continues at location (PC) + displacement. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC. If the 8-bit displacement field in the instruction word is zero, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones ($FF), the 32-bit displacement (long word immediately following the instruction) is used.

Condition Codes:

Not affected.

Instruction Format:



Instruction Fields:

8-Bit Displacement field — Twos complement integer specifying the number of bytes between the branch instruction and the next instruction to be executed.

16-Bit Displacement field — Used for larger displacement when 8-bit displacement is $00.

32-Bit Displacement field — Used for larger displacement when 8-bit displacement is $FF.

NOTE

A branch to the instruction immediately following automatically uses 16-bit displacement because the 8-bit displacement field contains $00 (zero offset).

BTST — Test a Bit — BTST

Operation: – (⟨bit number⟩ of Destination) → Z

Assembler Syntax: BTST Dn, ⟨ea⟩
BTST #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Long)

Description: Tests a bit in the destination operand and sets the Z condition code appropriately. When a data register is the destination, any of the 32 bits can be specified by a modulo 32 bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit zero refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. Immediate — The bit number is specified by a second word of the instruction.
2. Register — The specified data register contains the bit number.

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X Not affected.
N Not affected.
Z Set if the bit tested is zero. Cleared otherwise.
V Not affected.
C Not affected.

Instruction Format (Bit Number Static, specified as immediate data):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

BTST — Test a Bit — BTST

Instruction Fields (Bit Number Static):

Bit Number field — Specifies the bit number.

Effective Address field — Specifies the destination location. Only data addressing modes areallowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*Long only; all others are byte only

Instruction Format (Bit Number Dynamic, specified in a register):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Bit Number Dynamic):

Register field — Specifies the data register that contains the bit number.

Effective Address field — Specifies the destination location. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*Long only; all others are byte only

CHK — Check Register Against Bounds — CHK

Operation: If Dn < 0 or Dn > Source then TRAP

Assembler Syntax: CHK ⟨ea⟩, Dn

Attributes: Size = (Word, Long)

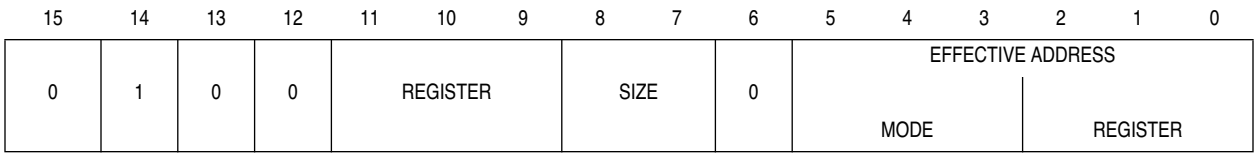
Description: Compares the value in the data register specified by the instruction to zero and to the upper bound (effective address operand). The upper bound is a twos complement integer. If the register value is less than zero or greater than the upper bound, a CHK instruction exception, vector number 6, occurs.

Condition Codes:

X	N	Z	V	C
—	*	U	U	U

X Not affected.
N Set if Dn < 0; cleared if Dn > effective address operand. Undefined otherwise.
Z Undefined.
V Undefined.
C Undefined.

Instruction Format:



Instruction Fields:

Register field — Specifies the data register that contains the value to be checked.
Size field — Specifies the size of the operation.
11 — Word operation.
10 — Long operation.
Effective Address field — Specifies the upper bound operand. Only data addressing modes areallowed as shown:

Effective Address field — Specifies the destination location. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register	Addressing Mode	Mode	Register
Dn*	000	Reg. number: Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	Reg. number: An	#⟨data⟩	—	—
(An) +	011	Reg. number: An			
– (An)	100	Reg. number: An			
(d_{16}, An)	101	Reg. number: An	(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An	(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An	(bd, PC, Xn)	111	011

*Long only; all others are byte only

CHK2 Check Register Against Bounds CHK2

Operation: If Rn < lower bound or Rn > upper bound then TRAP

Assembler Syntax: CHK2 ⟨ea⟩, Rn

Attributes: Size = (Byte, Word, Long)

Description: Compares the value in Rn to each bound. The effective address contains the bounds pair: the lower bound followed by the upper bound. For signed comparisons, the arithmetically smaller value should be used as the lower bound. For unsigned comparisons, the logically smaller value should be the lower bound.

The size of both data and the bounds can be specified as byte, word, or long. If Rn is a data register and the operation size is byte or word, only the appropriate low-order part of Rn is checked. If Rn is an address register and the operation size is byte or word, the bounds operands are sign-extended to 32 bits and the resultant operands are compared to the full 32 bits of An.

If the upper bound equals the lower bound, the valid range is a single value. If the register value is less than the lower bound or greater than the upper bound, a CHK instruction exception, vector number 6, occurs.

Condition Codes:

X	N	Z	V	C
—	U	*	U	*

X Not affected.
N Undefined.
Z Set if Rn is equal to either bound. Cleared otherwise.
V Undefined.
C Set if Rn is out of bounds. Cleared otherwise.

CHK2 — Check Register Against Bounds — CHK2

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	SIZE		0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
D/A	REGISTER			1	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the location of the bounds operands. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

D/A field — Specifies whether an address register or data register is to be checked.
- 0 — Data register.
- 1 — Address register.

Register field — Specifies the address or data register that contains the value to be checked.

CLR — Clear an Operand — CLR

Operation: 0 → Destination

Assembler Syntax: CLR ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Clears the destination operand to zero.

Condition Codes:

X	N	Z	V	C
—	0	1	0	0

X Not affected.
N Always cleared.
Z Always set.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7 6	5 4 3	2 1 0
0	1	0	0	0	0	1	0	SIZE	EFFECTIVE ADDRESS: MODE	EFFECTIVE ADDRESS: REGISTER

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. Number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

CMP — Compare — CMP

Operation: Destination – Source → cc

Assembler Syntax: CMP ⟨ea⟩, Dn

Attributes: Size = (Byte, Word, Long)

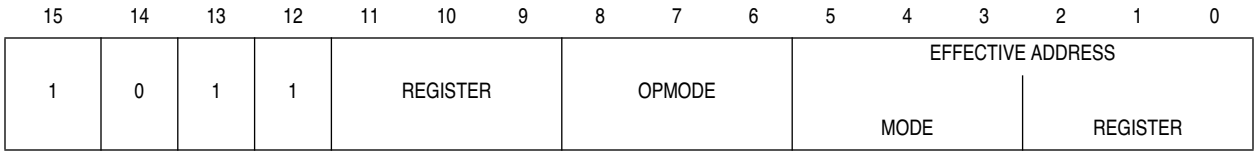
Description: Subtracts the source operand from the destination data register and sets condition codes according to the result. The data register is not changed.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow occurs. Cleared otherwise.
- C Set if a borrow occurs. Cleared otherwise.

Instruction Format:



Instruction Fields:

Register field — Specifies the destination data register.
Opmode field:

Byte	Word	Long	Operation
000	001	010	(⟨Dn⟩) – (⟨ea⟩)

Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An*	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*Word and long only

NOTE

CMPA is used when the destination is an address register. CMPI is used when the source is immediate data. CMPM is used for memory-to-memory compares. Most assemblers automatically make the distinction.

CMPA — Compare Address — CMPA

Operation: Destination – Source → cc

Assembler Syntax: CMPA ⟨ea⟩, An

Attributes: Size = (Word, Long)

Description: Subtracts the source operand from the destination address register and sets the condition codes according to the result. The address register is not changed. The size of the operation can be specified as word or long. Word length source operands are sign extended to 32-bits for comparison.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	1	1	REGISTER	OPMODE	EFFECTIVE ADDRESS: MODE	EFFECTIVE ADDRESS: REGISTER

Instruction Fields:

Register field — Specifies the destination address register.
Opmode field — Specifies the size of the operation:
011 — Word operation. The source operand is sign-extended to a long operand and the operation is performed on the address register using all 32 bits.
111 — Long operation.

Effective Address field — Specifies source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

CMPI — Compare Immediate — CMPI

Operation: Destination – Immediate Data → cc

Assembler Syntax: CMPI #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

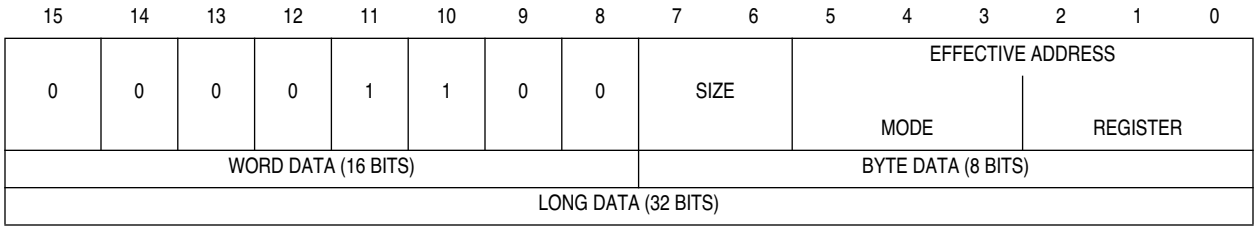
Description: Subtracts the immediate data from the destination operand and sets condition codes according to the result. The destination location is not changed. The size of the immediate data must match the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow occurs. Cleared otherwise.
C Set if a borrow occurs. Cleared otherwise.

Instruction Format:



Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data addressing modes, except immediate, are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Immediate field — (Data immediately following the instruction):

If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

CMPM Compare Memory CMPM

Operation: Destination – Source → cc

Assembler Syntax: CMPM (Ay)+, (Ax)+

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand from the destination operand and sets the condition codes according to the results. The destination location is not changed. The operands are always addressed with the postincrement addressing mode, using the address registers specified by the instruction.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if an overflow is generated. Cleared otherwise.
C Set if a borrow is generated. Cleared otherwise.

Instruction Format:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	1	1	REGISTER Ax	1	SIZE	0	0	1	REGISTER Ay

Instruction Fields:

Register Ax field — (always the destination). Specifies an address register in the postincrement addressing mode.
Size field — Specifies the size of the operation:
00 — Byte operation
01 — Word operation
10 — Long operation
Register Ay field — (always the source). Specifies an address register in the postincrement addressing mode.

CMP2 Compare Register Against Bounds CMP2

Operation: Compare Rn < lower-bound or
Rn > upper-bound
and Set Condition Codes

Assembler Syntax: CMP2 ⟨ea⟩, Rn

Attributes: Size = (Byte, Word, Long)

Description: Compares the value in Rn to each bound. The effective address contains the bounds pair: the lower bound followed by the upper bound. For signed comparisons, the arithmetically smaller value should be used as the lower bound. For unsigned comparisons, the logically smaller value should be the lower bound.

The size of the data and the bounds can be specified as byte, word, or long. If Rn is a data register and the operation size is byte or word, only the appropriate low-order part of Rn is checked. If Rn is an address register and the operation size is byte or word, the bounds operands are sign-extended to 32 bits and the resultant operands are compared to the full 32 bits of An.

If the upper bound equals the lower bound, the valid range is a single value.

NOTE

This instruction is identical to CHK2, except that it sets condition codes rather than taking an exception when the value in Rn is out of bounds.

Condition Codes:

X	N	Z	V	C
—	U	*	U	*

X Not affected.
N Undefined.
Z Set if Rn is equal to either bound. Cleared otherwise.
V Undefined.
C Set if Rn is out of bounds. Cleared otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	SIZE		0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
D/A	REGISTER			0	0	0	0	0	0	0	0	0	0	0	0

CMP2 Compare Register Against Bounds CMP2

Instruction Fields:

Size field — Specifies the size of the operation.
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the location of the bounds pair. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

D/A field — Specifies whether an address register or data register is compared.
- 0 — Data register.
- 1 — Address register.

Register field — Specifies the address or data register that contains the value to be checked.

DBcc Test Condition, Decrement, and Branch DBcc

Operation: If condition false then Dn – 1 → Dn;If Dn ≠ –1 then PC + d → PC

Assembler Syntax: DBcc Dn, ⟨label⟩

Attributes: Size = (Word)

Description: Controls a loop of instructions. The parameters are a condition code, a data register (counter), and a displacement value. The instruction first tests the condition (for termination); if it is true, no operation is performed. If the termination condition is not true, the low-order 16 bits of the counter data register are decremented by one. If the result is –1, execution continues with the next instruction. If the result is not equal to –1, execution continues at the location indicated by the current value of the PC, plus the sign-extended 16-bit displacement. The value in the PC is the address of the instruction word of the DBcc instruction plus two. The displacement is a twos complement integer that represents the relative distance in bytes from the current PC to the destination PC.

Condition code cc specifies one of the following conditions:

cc	Name	Code	Description	cc	Name	Code	Description
CC	Carry Clear	0100	$\overline{C}$	LS	Low or Same	0011	$\overline{C}$; $\overline{Z}$
CS	Carry Set	0101	C	LT	Less Than	1101	N • $\overline{V}$; $\overline{N}$ • V
EQ	Equal	0111	Z	MI	Minus	1011	N
F	Never equal	0001	0	NE	Not Equal	0110	$\overline{Z}$
GE	Greater or Equal	1100	N • V; $\overline{N}$ • $\overline{V}$	PL	Plus	1010	$\overline{N}$
GT	Greater Than	1110	N • V • $\overline{Z}$; $\overline{N}$ • $\overline{V}$ • $\overline{Z}$	T	Always true	0000	1
HI	High	0010	$\overline{C}$ • $\overline{Z}$	VC	Overflow Clear	1000	$\overline{V}$
LE	Less or Equal	1111	Z; N • $\overline{V}$; $\overline{N}$ • V	VS	Overflow Set	1001	V

Condition Codes:

Not affected.

DBcc Test Condition, Decrement, and Branch DBcc

Instruction Format:

15	14	13	12	11 10 9 8	7	6	5	4	3	2 1 0
0	1	0	1	CONDITION	1	1	0	0	1	REGISTER
DISPLACEMENT										

Instruction Fields:

Condition field — The binary code for one of the conditions listed in the table.
Register field — Specifies the data register used as the counter.
Displacement field — Specifies the number of bytes to branch.

NOTES:

1. Terminating condition is similar to UNTIL loop clauses of high-level languages. For example, DBMI can be stated decrement and branch until minus."
2. Most assemblers accept DBRA for DBF when a count terminates the loop (no condition is tested).
3. A program can enter a loop at the beginning, or by branching to the trailing DBcc instruction. Entering the loop at the beginning is useful for indexed addressing modes and dynamically specified bit operations. In this case, the control index count must be one less than the desired number of loop executions. However, when entering a loop by branching to the trailing DBcc instruction, the control count should equal the loop execution count so that the DBcc instruction will not branch and the main loop will not execute if a zero count occurs.

DIVS DIVSL — Signed Divide — DIVS DIVSL

Operation: Destination / Source → Destination

Assembler Syntax:
DIVS.W ⟨ea⟩, Dn32/16 → 16r:16q
DIVS.L ⟨ea⟩, Dq32/32 → 32q
DIVS.L ⟨ea⟩, Dr:Dq64/32 → 32r:32q
DIVSL.L ⟨ea⟩, Dr:Dq32/32 → 32r:32q

Attributes: Size = (Word, Long)

Description: Divides the signed destination operand by the signed source operand and stores the signed result in the destination. The instruction uses one of four forms.

The word form of the instruction divides a long word by a word. The result is a quotient in the lower word (least significant 16 bits) and a remainder in the upper word (most significant 16 bits) of the destination. The sign of the remainder is the same as the sign of the dividend.

The first long form divides a long word by a long word. The result is a long quotient; the remainder is discarded.

The second long form divides a quad word (in any two data registers) by a long word. The result is a long word quotient and a long word remainder.

The third long form divides a long word by a long word. The result is a long word quotient and a long word remainder.

Two special conditions may arise during the operation:

1. Division by zero causes a trap.
2. Overflow may be detected before instruction completion. If an overflow is detected, the overflow condition code is set and the operands are unaffected.

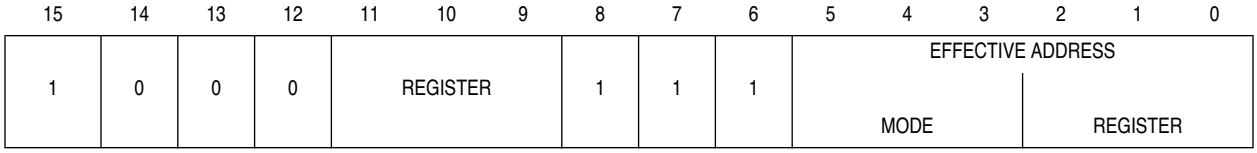
DIVS DIVSL — Signed Divide — DIVS DIVSL

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.
N Set if quotient is negative. Cleared otherwise. Undefined if overflow or divide by zero occurs.
Z Set if quotient is zero. Cleared otherwise. Undefined if overflow or divide by zero occurs.
V Set if division overflow occurs; undefined if divide by zero occurs. Cleared otherwise.
C Always cleared.

Instruction Format (word form):



Instruction Fields:

Register field — Specifies any of the eight data registers. This field always specifies the destination operand.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

NOTE

Overflow occurs if the quotient is larger than a 16-bit signed integer.

DIVS DIVSL — Signed Divide — DIVS DIVSL

Instruction Format (long form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dq			1	SIZE	0	0	0	0	0	0	0	REGISTER Dr		

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Register Dq field — Specifies a data register for the destination operand. The low-order 32 bits of the dividend come from this register, and the 32-bit quotient is loaded into this register.

Size field — Selects a 32 or 64 bit division operation.

0 — 32-bit dividend is in Register Dq.

1 — 64-bit dividend is in Dr:Dq.

Register Dr field — After the division, this register contains the 32-bit remainder. If Dr and Dq are the same register, only the quotient is returned. If Size is 1, the Dr field also specifies the data register that contains the high-order 32 bits of the dividend.

NOTE

Overflow occurs if the quotient is larger than a 32-bit signed integer.

DIVU DIVUL — Unsigned Divide

Operation: Destination/Source → Destination

Assembler Syntax:
DIVS.W ⟨ea⟩, Dn32/16 → 16r:16q
DIVS.L ⟨ea⟩, Dq32/32 → 32q
DIVS.L ⟨ea⟩, Dr:Dq64/32 → 32r:32q
DIVSL.L ⟨ea⟩, Dr:Dq32/32 →32r:32q

Attributes: Size = (Word, Long)

Description: Divides the unsigned destination operand by the unsigned source operand and stores the unsigned result in the destination. The instruction uses one of four forms.

The word form of the instruction divides a long word by a word. The result is a quotient in the lower word (least significant 16 bits) and a remainder in the upper word (most significant 16 bits) of the destination.

The first long form divides a long word by a long word. The result is a long quotient; the remainder is discarded.

The second long form divides a quad word (in any two data registers) by a long word. The result is a long word quotient and a long word remainder.

The third long form divides a long word by a long word. The result is a long word quotient and a long word remainder.

Two special conditions may arise during the operation:

1. Division by zero causes a trap.
2. Overflow may be detected before instruction completion. If an overflow is detected, the overflow condition code is set and the operands are unaffected.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.
N Set if quotient is negative. Cleared otherwise. Undefined if overflow or divide by zero occurs.
Z Set if quotient is zero. Cleared otherwise. Undefined if overflow or divide by zero occurs.
V Set if division overflow occurs; undefined if divide by zero occurs. Cleared otherwise.
C Always cleared.

DIVU DIVUL — Unsigned Divide — DIVU DIVUL

Instruction Format (word form):

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
1	0	0	0	REGISTER	0	1	1	EFFECTIVE ADDRESS: MODE	EFFECTIVE ADDRESS: REGISTER

Instruction Fields:

Register field — Specifies any of the eight data registers. This field always specifies the destination operand.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

NOTE

Overflow occurs if the quotient is larger than a 16-bit signed integer.

DIVU DIVUL — Unsigned Divide — DIVU DIVUL

Instruction Format (long form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dq			1	SIZE	0	0	0	0	0	0	0	REGISTER Dr		

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Register Dq field — Specifies a data register for the destination operand. The low-order 32 bits of the dividend come from this register, and the 32-bit quotient is loaded into this register.

Size field — Selects a 32 or 64 bit division operation.

0 — 32-bit dividend is in Register Dq.

1 — 64-bit dividend is in Dr:Dq.

Register Dr field — After the division, this register contains the 32-bit remainder. If Dr and Dq are the same register, only the quotient is returned. If Size is 1, this field also specifies the data register that contains the high-order 32 bits of the dividend.

NOTE

Overflow occurs if the quotient is larger than a 32-bit signed integer.

EOR — Exclusive OR — EOR

Operation: Source ⊕ Destination → Destination

Assembler Syntax: EOR Dn, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an exclusive OR operation on the destination operand using the source operand and stores the result in the destination location. The source operand must be a data register. The destination operand is specified in the effective address field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Instruction Format:

15	14	13	12	11 10 9	8 7 6	5 4 3	2 1 0
1	0	1	1	REGISTER	OPMODE	EFFECTIVE ADDRESS: MODE	EFFECTIVE ADDRESS: REGISTER

Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) ⊕ (⟨Dn⟩) → ⟨ea⟩

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTE

Memory to data register operations are not allowed. Most assemblers use EORI when the source is immediate data.

EORI — Exclusive OR Immediate — EORI

Operation: Immediate Data ⊕ Destination → Destination

Assembler Syntax: EORI #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an exclusive OR operation on the destination operand using the immediate data and the destination operand and stores the result in the destination location. The size of the immediate data must match the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Instruction Fields:

Size field — Specifies the size of the operation:

- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

Immediate field — (Data immediately following the instruction):

If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is next two immediate words.

EORI to CCR

Exclusive OR Immediate to Condition Code Register

EORI to CCR

Operation: Source ⊕ CCR → CCR

Assembler Syntax: EORI #⟨data⟩, CCR

Attributes: Size = (Byte)

Description: Performs an exclusive OR operation on the condition code register using the immediate operand, and stores the result in the condition code register (low-order byte of the status register). All implemented bits of the condition code register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Changed if bit 4 of immediate operand is one. Unchanged otherwise.
- N Changed if bit 3 of immediate operand is one. Unchanged otherwise.
- Z Changed if bit 2 of immediate operand is one. Unchanged otherwise.
- V Changed if bit 1 of immediate operand is one. Unchanged otherwise.
- C Changed if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

EORI to SR

Exclusive OR Immediate to Status Register (Privileged Instruction)

EORI to SR

Operation: If supervisor state
then Source ⊕ SR → SR
else TRAP

Assembler Syntax: EORI #⟨data⟩, SR

Attributes: Size = (Word)

Description: Performs an exclusive OR operation on the contents of the status register using the immediate operand, and stores the result in the status register. All implemented bits of the status register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Changed if bit 4 of immediate operand is one. Unchanged otherwise.
N Changed if bit 3 of immediate operand is one. Unchanged otherwise.
Z Changed if bit 2 of immediate operand is one. Unchanged otherwise.
V Changed if bit 1 of immediate operand is one. Unchanged otherwise.
C Changed if bit 0 of immediate operand is one. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

EXG — Exchange Registers — EXG

Operation: Rx ↔ Ry

Assembler: EXG Dx, Dy
Syntax: EXG Ax, Ay
EXG Dx, Ay
EXG Ay, Dx

Attributes: Size = (Long)

Description: Exchanges the contents of two 32-bit registers. The instruction performs three types of exchanges:

1. Exchange data registers.
2. Exchange address registers.
3. Exchange a data register and an address register.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11 10 9	8	7 6 5 4 3	2 1 0
1	1	0	0	REGISTER Rx	1	OPMODE	REGISTER Ry

Instruction Fields:

Register Rx field — Specifies either a data register or an address register depending on the mode. If the exchange is between data and address registers, this field always specifies the data register.

Opmode field — Specifies the type of exchange:
01000 — Data registers.
01001 — Address registers.
10001 — Data register and address register.

Register Ry field — Specifies either a data register or an address register depending on the mode. If the exchange is between data and address registers, this field always specifies the address register.

EXT EXTB — Sign Extend — EXT EXTB

Operation: Destination Sign-extended → Destination

Assembler Syntax:
EXT.W Dnextend byte to word
EXT.L Dnextend word to long word
EXTB.L Dnextend byte to long word

Attributes: Size = (Word, Long)

Description: Extends a byte in a data register to a word or a long word, or a word in a data register to a long word, by replicating the sign bit to the left. If the operation extends a byte to a word, bit [7] of the designated data register is copied to bits [15:8] of that data register. If the operation extends a word to a long word, bit [15] of the designated data register is copied to bits [31:16] of the data register. The EXTB form copies bit [7] of the designated register to bits [31:8] of the data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	OPMODE			0	0	0	REGISTER		

Instruction Fields:

Opmode field — Specifies the size of the sign-extension operation:
- 010 — Sign-extend low-order byte of data register to word.
- 011 — Sign-extend low-order word of data register to long.
- 111 — Sign-extend low-order byte of data register to long.

Register field — Specifies the data register is to be sign-extended.

ILLEGAL — Take Illegal Instruction Trap — ILLEGAL

Operation: SSP – 2 → SSP; Vector Offset → (SSP);
SSP – 4 → SSP; PC → (SSP);
SSP – 2 → SSP; SR → (SSP);
Illegal Instruction Vector Address → PC

Assembler Syntax: ILLEGAL

Attributes: Unsized

Description: Forces an illegal instruction exception, vector number 4. All other illegal instruction bit patterns are reserved for future extension of the instruction set and should not be used to force an exception.

Condition Codes:

Not affected

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0

JMP — Jump — JMP

Operation: Destination Address → PC

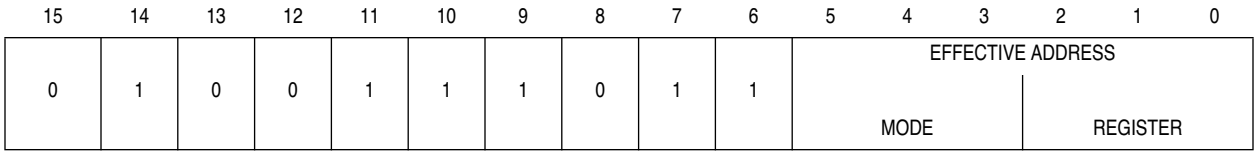
Assembler Syntax: JMP ⟨ea⟩

Attributes: Unsized

Description: Program execution continues at the effective address specified by the instruction. The addressing mode for the effective address must be a control addressing mode.

Condition Codes:
Not affected.

Instruction Format:



Instruction Fields:
Effective Address field — Specifies the address of the next instruction. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

JSR — Jump to Subroutine — JSR

Operation: SP – 4 → Sp; PC → (SP)
Destination Address → PC

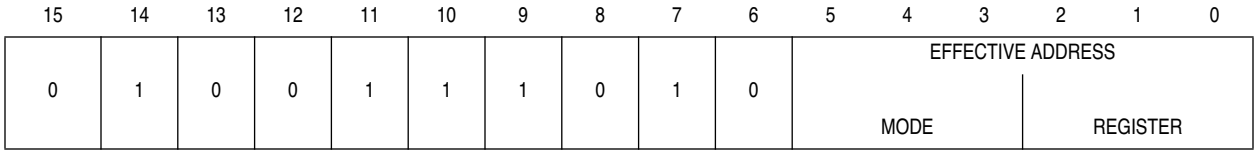
Assembler Syntax: JSR ⟨ea⟩

Attributes: Unsized

Description: Pushes the long word address of the instruction immediately following the JSR instruction onto the system stack. Program execution then continues at the address specified by the instruction.

Condition Codes:
Not affected.

Instruction Format:



Instruction Fields:
Effective Address field — Specifies the address of the next instruction. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

LEA — Load Effective Address — LEA

Operation: ⟨ea⟩ → An

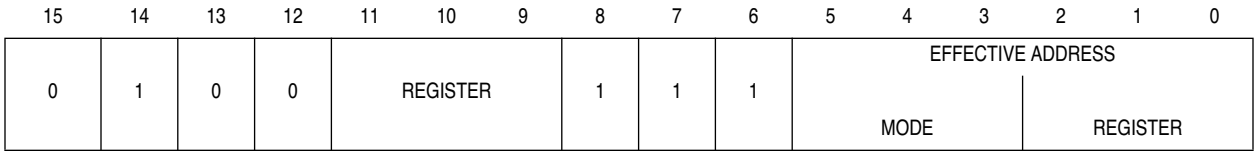
Assembler Syntax: LEA ⟨ea⟩, An

Attributes: Size = (Long)

Description: Loads the effective address into the specified address register. All 32 bits of the address register are affected by this instruction.

Condition Codes:
Not affected.

Instruction Format:



Instruction Fields:

Register field — Specifies the address register to be updated with the effective address.

Effective Address field — Specifies the address to be loaded into the address register. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

LINK — Link and Allocate — LINK

Operation: Sp – 4 → Sp; An → (SP);
SP → An; SP + d → SP

Assembler Syntax: LINK An, #⟨displacement⟩

Attributes: Size = (Word, Long)

Description: Pushes the contents of the specified address register onto the stack, then loads the updated stack pointer into the address register. Finally, adds the displacement value to the stack pointer. For word size operation, the displacement is the sign-extended word following the operation word. For long size operation, the displacement is the long word following the operation word. The address register occupies one long word on the stack. The user should specify a negative displacement to allocate stack area.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	REGISTER		
WORD DISPLACEMENT															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	0	1	REGISTER		
HIGH-ORDER DISPLACEMENT															
LOW-ORDER DISPLACEMENT															

Instruction Fields:
Register field — Specifies the address register for the link.
Displacement field — Specifies the twos complement integer to be added to the stack pointer.

NOTE

LINK and UNLK can be used to maintain a linked list of local data and parameter areas on the stack for nested subroutine calls.

LPSTOP Low Power Stop LPSTOP

Operation: If supervisor state
then Immediate Data → SR
Interrupt Mask → External Bus Interface (EBI)
STOP
else TRAP

Assembler Syntax: LPSTOP #⟨data⟩

Attributes: Size = (Word) Privileged

Description: The immediate operand is moved into the entire status register, the program counter is advanced to point to the next instruction, and the processor stops fetching and executing instructions. A CPU LPSTOP broadcast cycle is executed to CPU space $3 to copy the updated interrupt mask to the external bus interface (EBI). The internal clocks are stopped.

Execution of instructions resumes when a trace, interrupt, or reset exception occurs. A trace exception occurs if the trace state is on when the LPSTOP instruction is executed. If an interrupt request is asserted with a higher priority that the current priority level set by the new status register value, an interrupt exception occurs; otherwise the interrupt request is ignored. If the bit of the immediate data corresponding to the S bit is off, execution of the instruction causes a privilege violation. An external reset always initiates reset exception processing.

Condition Codes:

Set according to the immediate operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
IMMEDIATE DATA															

Instruction Fields:

Immediate field — Specifies the data to be loaded into the status register.

LSL, LSR — Logical Shift — LSL, LSR

Operation: Destination Shifted by ⟨count⟩ → Destination

Assembler Syntax:
LSd Dx, Dy
LSd #⟨data⟩, Dy
LSd ⟨ea⟩
where d is direction, L or R

Attributes: Size = (Byte, Word, Long)

Description: Shifts the bits of the operand in the direction specified (L or R). The carry bit receives the last bit shifted out of the operand.

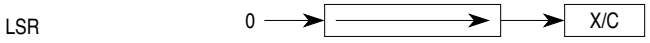
Shift count can be specified in one of two ways:

1. Immediate — The shift count (1–8) is specified by the instruction.
2. Register — The shift count is the value in the data register specified by the instruction, modulo 64.

The size of the operation for register destinations may be specified as byte, word, or long. The contents of memory, ⟨ea⟩, can be shifted one bit only, and the operand size is restricted to a word.

The LSL instruction shifts the operand to the left the number of positions specified as the shift count. Bits shifted out of the high-order bit go to both the carry and the extend bits; zeros are shifted into the low-order bits.

The LSR instruction shifts the operand to the right the number of positions specified as the shift count. Bits shifted out of the low-order bit go to both the carry and the extend bits; zeros are shifted into the high-order bits.



LSL, LSR — Logical Shift — LSL, LSR

Condition Codes:

X	N	Z	V	C
*	*	*	0	*

- X Set according to the last bit shifted out of the operand. Unaffected for a shift count of zero.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Set according to the last bit shifted out of the operand. Cleared for a shift count of zero.

Instruction Format (Register Shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/REGISTER			dr	SIZE		i/r	0	1	REGISTER		

Instruction Fields (Register Shifts):

Count/Register field — Specifies shift count or register that contains shift count:

If i/r = 0, this field contains the shift count. The values one to seven represent counts of one to seven; value of zero represents a count of eight.

If i/r = 1, this field specifies the data register that contains the shift count (modulo 64).

dr field — Specifies the direction of the shift:

0 — Shift right

1 — Shift left

Size field — Specifies the size of the operation:

00 — Byte operation

01 — Word operation

10 — Long operation

i/r field:

If i/r = 0, specifies immediate shift count.

If i/r = 1, specifies register shift count.

Register field — Specifies a data register to be shifted.

LSL, LSR — Logical Shift — LSL, LSR

Instruction Format (Memory Shifts):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Shifts):

dr field — Specifies the direction of the shift:

0 — Shift right

1 — Shift left

Effective Address field — Specifies the operand to be shifted.

Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

MOVE — Move Data from Source to Destination — MOVE

Operation: Source → Destination

Assembler Syntax: MOVE ⟨ea⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

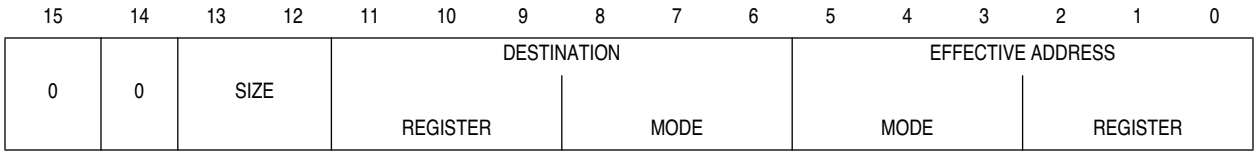
Description: Moves the data at the source to the destination location, and sets the condition codes according to the data.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Instruction Format:



Instruction Fields:

Size field — Specifies the size of the operand to be moved:

- 01 — Byte operation
- 11 — Word operation
- 10 — Long operation

Destination Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

Source Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An*	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*For byte size operation, address register direct is not allowed.

NOTES:

1. Most assemblers use MOVEA when the destination is an address register.
2. MOVEQ can be used to move an immediate 8-bit value to a data register.

MOVEA — Move Address — MOVEA

Operation: Source → Destination

Assembler Syntax: MOVEA ⟨ea⟩, An

Attributes: Size = (Word, Long)

Description: Moves the contents of the source to the destination address register. The size of the operation is specified as word or long. Word size source operands are sign-extended to 32-bit quantities.

Condition Codes:
Not affected.

Instruction Format:

15	14	13 12	11 10 9	8	7	6	5 4 3	2 1 0
0	0	SIZE	DST–REG	0	0	1	EFFECTIVE ADDRESS: MODE	EFFECTIVE ADDRESS: REGISTER

Instruction Fields:

Size field — Specifies the size of the operand to be moved:
- 11 — Word operation. The source operand is sign-extended to a long operand and all 32 bits are loaded into the address register.
- 10 — Long operation.

Destination Register (Dst-Reg) field — Specifies the destination address register.

Effective Address field — Specifies the location of the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

MOVE from CCR — Move from the Condition Code Register — MOVE from CCR

Operation: CCR → Destination

Assembler Syntax: MOVE CCR, ⟨ea⟩

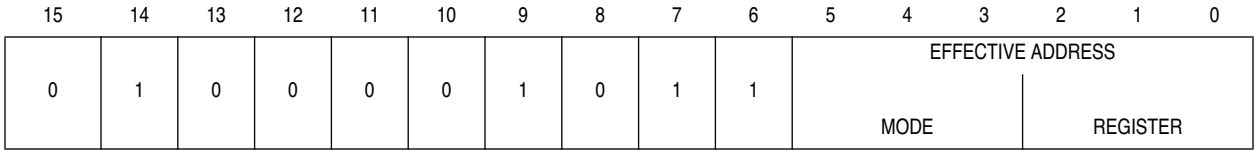
Attributes: Size = (Word)

Description: Moves the condition code bits (zero extended to word size) to the destination location. Unimplemented bits are read as zeros.

Condition Codes:

Not affected.

Instruction Format:



Instruction Fields:

Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTE

MOVE from CCR is a word operation. ANDI, ORI, and EORI to CCR are byte operations.

MOVE to CCR — Move to Condition Code Register — MOVE to CCR

Operation: Source → CCR

Assembler Syntax: MOVE ⟨ea⟩, CCR

Attributes: Size = (Word)

Description: Moves the low-order byte of the source operand to the condition code register. The upper byte of the source operand is ignored; the upper byte of the status register is not altered.

Condition Codes:

X	N	Z	V	C											
*	*	*	*	*											

X Set to the value of bit 4 of the source operand.
N Set to the value of bit 3 of the source operand.
Z Set to the value of bit 2 of the source operand.
V Set to the value of bit 1 of the source operand.
C Set to the value of bit 0 of the source operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

MOVE to CCR — Move to Condition Code Register — MOVE to CCR

Instruction Fields:

Effective Address field — Specifies the destination location. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

NOTE

MOVE to CCR is a word operation. ANDI, ORI, and EORI to CCR are byte operations.

MOVE from SR

Move from the Status Register (Privileged Instruction)

MOVE from SR

Operation: If supervisor state
then SR → Destination
else TRAP

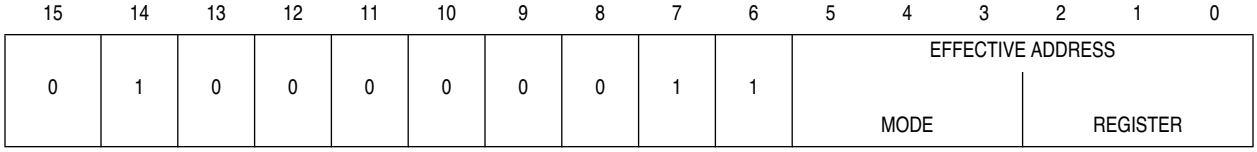
Assembler Syntax: MOVE SR, ⟨ea⟩

Attributes: Size = (Word)

Description: Moves the data in the status register to the destination location. The destination must be of word length. Unimplemented bits are read as zeros.

Condition Codes:
Not affected.

Instruction Format:



Instruction Fields:
Effective Address field — Specifies the destination location. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTE

Use the MOVE from CCR instruction to access only the condition codes.

MOVE to SR

Move to the Status Register

(Privileged Instruction)

MOVE to SR

Operation: If supervisor state
then Source →SR
else TRAP

Assembler Syntax: MOVE ⟨ea⟩, SR

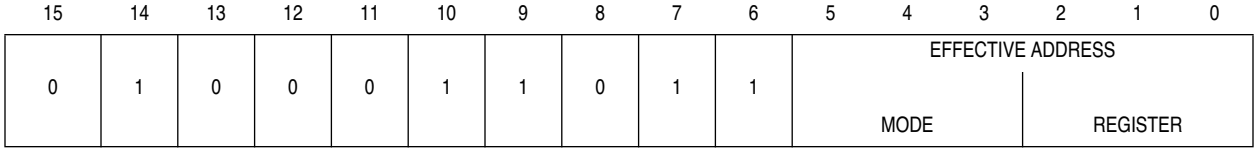
Attributes: Size = (Word)

Description: Moves the data in the source operand to the status register. The source operand is a word and all implemented bits of the status register are affected.

Condition Codes:

Set according to the source operand.

Instruction Format:



Instruction Fields:

Effective Address field — Specifies the destination location. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

MOVE USP — Move User Stack Pointer (Privileged Instruction) — MOVE USP

Operation: If supervisor state
then USP → An or An → USP
else TRAP

Assembler Syntax: MOVE USP, An
MOVE An, USP

Attributes: Size = (Long)

Description: Moves the contents of the user stack pointer to or from the specified address register.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	dr	REGISTER		

Instruction Fields:
dr field — Specifies the direction of transfer:
0 — Transfer the address register to the USP.
1 — Transfer the USP to the address register.
Register field — Specifies the address register for the operation.

MOVEC — Move Control Register — MOVEC

(Privileged Instruction)

Operation: If supervisor state
then Rc → Rn or Rn → Rc
else TRAP

Assembler Syntax: MOVEC Rc, Rn
MOVEC Rn, Rc

Attributes: Size = (Long)

Description: Moves the contents of the specified control register (Rc) to the specified general register (Rn), or copies the contents of the specified general register to the specified control register. MOVEC is always a 32-bit transfer even though the control register may be implemented with fewer bits. Unimplemented bits are read as zeros.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
A/D	REGISTER			CONTROL REGISTER											

Instruction Fields:
dr field — Specifies the direction of the transfer:
0 — Control register to general register.
1 — General register to control register.
A/D field — Specifies the type of general register:
0 — Data register.
1 — Address register.
Register field — Specifies the register number.
Control Register field — Specifies the control register.

Hex	Control Register
000	Source Function Code (SFC)
001	Destination Function Code (DFC)
800	User Stack Pointer (USP)
801	Vector Base Register (VBR)

Any other code causes an illegal instruction exception.

MOVEM Move Multiple Registers MOVEM

Operation: Registers → Destination
Source → Registers

Assembler Syntax: MOVEM register list, ⟨ea⟩
MOVEM ⟨ea⟩, register list

Attributes: Size = (Word, Long)

Description: Moves the contents of selected registers to or from consecutive memory locations starting at the location specified by the effective address. A register is selected if the bit in the mask field corresponding to that register is set. The instruction size determines whether 16 or 32 bits of each register are transferred. In the case of a word transfer to either address or data registers, each word is sign-extended to 32 bits, and the resulting long word is loaded into the associated register.

Selecting the addressing mode also selects the mode of operation of the MOVEM instruction, and only the control modes, the predecrement mode, and the postincrement mode are valid. If the effective address is specified by one of the control modes, the registers are transferred starting at the specified address, and the address is incremented by the operand length (2 or 4) following each transfer. The order of the registers is from data register 0 to data register 7, then from address register 0 to address register 7.

If the effective address is specified by the predecrement mode, only a register-to-memory operation is allowed. The registers are stored starting at the specified address minus the operand length (2 or 4), and the address is decremented by the operand length following each transfer. The order of storing is from address register 7 to address register 0, then from data register 7 to data register 0. When the instruction has completed, the decremented address register contains the address of the last operand stored. In the CPU 32, if the addressing register is also moved to memory, the value written is the decremented value.

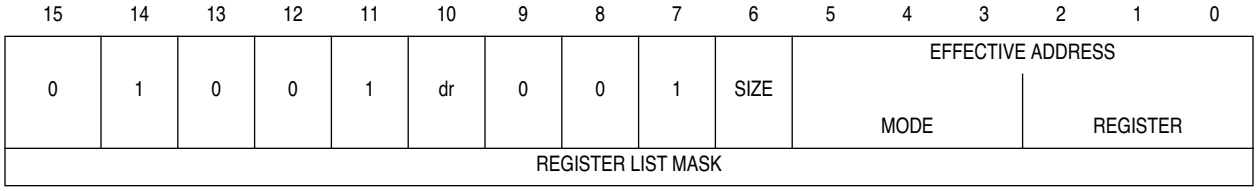
If the effective address is specified by the postincrement mode, only a memory-to-register operation is allowed. The registers are loaded starting at the specified address; the address is incremented by the operand length (2 or 4) following each transfer. The order of loading is the same as that of control mode addressing. When the instruction has completed, the incremented address register contains the address of the last operand loaded plus the operand length. In the CPU32, if the addressing register is also loaded from memory, the value loaded is the value fetched plus the operand length.

MOVEM — Move Multiple Registers — MOVEM

Condition Codes:

Not affected.

Instruction Format:



Instruction Field:

dr field — Specifies the direction of the transfer:

0 — Register to memory

1 — Memory to register

Size field — Specifies the size of the registers being transferred:

0 — Word transfer

1 — Long transfer

Effective Address field — Specifies the memory address for the operation. For register-to-memory transfers, only control alterable addressing modes, or the predecrement addressing mode are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

MOVEM Move Multiple Registers MOVEM

For memory-to-register transfers, only control addressing modes or the postincrement addressing mode are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Register List Mask field — Specifies the registers to be transferred. The low-order bit corresponds to the first register to be transferred; the high-order bit corresponds to the last register to be transferred. Thus, both for control modes and for the postincrement mode addresses, the mask correspondence is:

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

For predecrement mode addresses, the mask correspondence is reversed:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

NOTE

An extra read bus cycle occurs for memory operands. This accesses an operand at one address higher than the last register image required.

MOVEP — Move Peripheral Data — MOVEP

Operation: Source → Destination

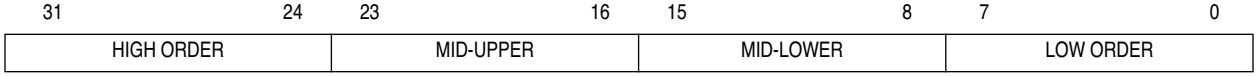
Assembler Syntax: MOVEP Dx, (d, Ay)
MOVEP (d, Ay), Dx

Attributes: Size = (Word, Long)

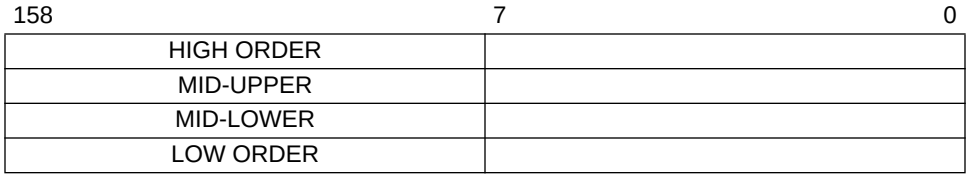
Description: Moves data between a data register and alternate bytes within the address space (typically assigned to a peripheral), starting at the location specified and incrementing by two. This instruction is designed for 8-bit peripherals on a 16-bit data bus. The high-order byte of the data register is transferred first and the low-order byte is transferred last. The memory address is specified by the address register indirect plus 16-bit displacement addressing mode. If the address is even, all the transfers are to or from the high-order half of the data bus; if the address is odd, all the transfers are to or from the low-order half of the data bus. The instruction also accesses alternate bytes on an 8- or 32-bit bus.

Example: Long transfer to/from an even address.

Byte Organization in Register

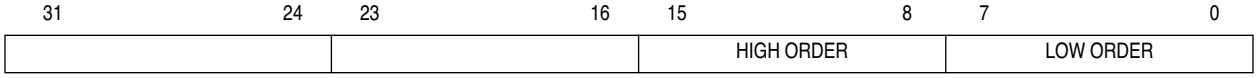


Byte Organization in Memory (Low Address at Top)

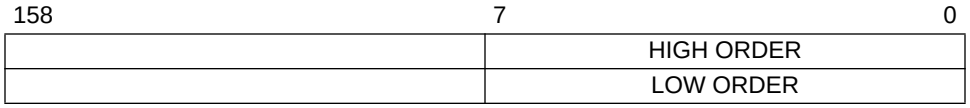


MOVEP — Move Peripheral Data — MOVEP

Example: Word transfer to/from an odd address
Byte Organization in Register



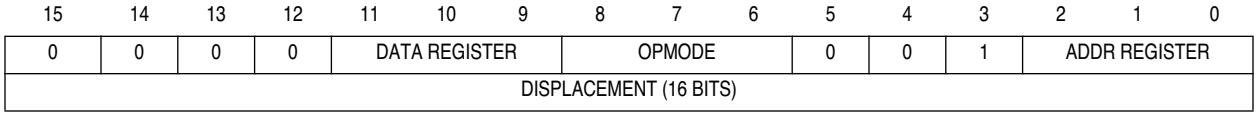
Byte Organization in Memory (Low Address at Top)



Condition Codes:

Not affected.

Instruction Format:



Instruction Fields:

Data Register field — Specifies the data register for the instruction.

Opmode field — Specifies the direction and size of the operation:

- 100 — Transfer word from memory to register.
- 101 — Transfer long from memory to register.
- 110 — Transfer word from register to memory.
- 111 — Transfer long from register to memory.

Address Register field — Specifies the address register which is used in the address register indirect plus displacement addressing mode.

Displacement field — Specifies the displacement used in the operand address.

MOVEQ — Move Quick — MOVEQ

Operation: Immediate Data → Destination

Assembler Syntax: MOVEQ #⟨data⟩, Dn

Attributes: Size = (Long)

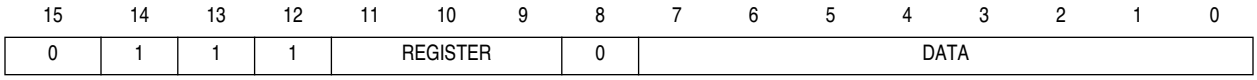
Description: Moves a byte of immediate data to a 32-bit data register. The data in an 8-bit field within the operation word is sign-extended to a long operand in the data register as it is transferred.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:



Instruction Fields:

Register field — Specifies the data register to be loaded.
Data field — Eight bits of data, which are sign-extended to a long operand.

MOVES Move Address Space (Privileged Instruction) MOVES

Operation: If supervisor state
then Rn → Destination [DFC] or Source [SFC] → Rn
else TRAP

Assembler: MOVES Rn, ⟨ea⟩Syntax:
MOVES ⟨ea⟩, Rn

Attributes: Size = (Byte, Word, Long)

Description: Moves the byte, word, or long operand from the specified general register to a location within the address space specified by the destination function code (DFC) register; or moves the byte, word, or long operand from a location within the address space specified by the source function code (SFC) register to the specified general register.

If the destination is a data register, the source operand replaces the corresponding low-order bits of the data register, depending on the size of the operation. If the destination is an address register, the source operand is sign-extended to 32 bits and then loaded into the address register.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
A/D	REGISTER			dr	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields:
Size field — Specifies the size of the operation:
00 — Byte operation
01 — Word operation
10 — Long operation

MOVES Move Address Space MOVES

(Privileged Instruction)

Effective Address field — Specifies the source or destination location within the alternate address space. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

A/D field — Specifies the type of general register:
0 — Data register
1 — Address register

Register field — Specifies the register number.

dr field — Specifies the direction of the transfer:
0 — From ⟨ea⟩ to general register
1 — From general register to ⟨ea⟩

NOTE

For either of the two following examples, which use the same address register as both source and destination, the value stored is undefined. The current implementations of the MC68010, CPU32, and MC68020 store the incremented or decremented value of An.

MOVES.x An, (An)+
MOVES.x An, –(An)

MULS — Signed Multiply — MULS

Operation: Source ∗ Destination → Destination

Assembler Syntax:
MULS.W ⟨ea⟩, Dn16x16 → 32
MULS.L ⟨ea⟩, Dl 32x32 → 32
MULS.L ⟨ea⟩, Dh:Dl32 x 32 → 64

Attributes: Size = (Word, Long)

Description:: Multiplies two signed operands yielding a signed result.

In the word form, the multiplier and multiplicand are both word operands, and the result is a long word operand. A register operand is the low-order word; the upper word of the register is ignored. All 32 bits of the product are saved in the destination data register.

In the long form, the multiplier and multiplicand are both long word operands, and the result is either a long word or a quad word. The long word result is the low-order 32 bits of the quad word result; the high-order 32 bits of the product are discarded.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.
N Set if the result is negative. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Set if overflow. Cleared otherwise.
C Always cleared.

NOTE

Overflow (V = 1) can occur only when multiplying 32-bit operands to yield a 32-bit result. Overflow occurs if the high-order 32 bits of the quad word product are not the sign extension of the low-order 32 bits.

MULS — Signed Multiply — MULS

Instruction Format (word form):

15	14	13	12	11 10 9	8	7	6	5 4 3	2 1 0
1	1	0	0	REGISTER	1	1	1	EFFECTIVE ADDRESS: MODE	EFFECTIVE ADDRESS: REGISTER

Instruction Fields:

Register field — Specifies a data register as the destination.

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

MULS — Signed Multiply — MULS

Instruction Format (long form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dl			1	SIZE	0	0	0	0	0	0	0	REGISTER Dh		

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Register Dl field — Specifies a data register for the destination operand. The 32-bit multiplicand comes from this register, and the low-order 32 bits of the product are loaded into this register.

Size field — Selects a 32- or 64-bit product.

0 — 32-bit product to be returned to register Dl.

1 — 64-bit product to be returned to Dh:Dl.

Register Dh field — If Size is 1, specifies the data register into which the high-order 32 bits of the product are loaded. If Dh = Dl and Size is 1, the results of the operation are undefined. This field is unused, otherwise.

MULU — Unsigned Multiply — MULU

Operation: Source ∗ Destination → Destination

Assembler Syntax:
MULU.W ⟨ea⟩, Dn16x16 → 32
MULU.L ⟨ea⟩, Dl32x32 → 32
MULU.L ⟨ea⟩, Dh:Dl32x32 →64

Attributes: Size = (Word, Long)

Description: Multiplies two unsigned operands yielding an unsigned result.

In the word form, the multiplier and multiplicand are both word operands, and the result is a long word operand. A register operand is the low-order word; the upper word of the register is ignored. All 32 bits of the product are saved in the destination data register.

In the long form, the multiplier and multiplicand are both long word operands, and the result is either a long word or a quad word. The long word result is the low-order 32 bits of the quad word result; the high-order 32 bits of the product are discarded.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

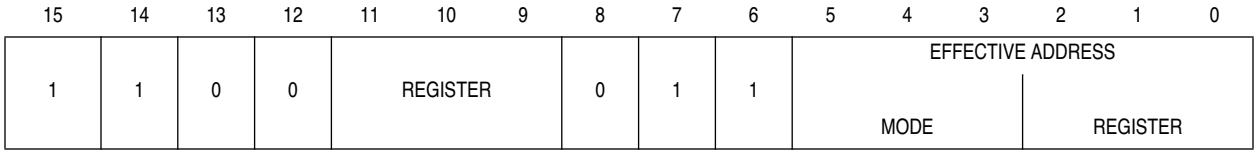
- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if overflow. Cleared otherwise.
- C Always cleared.

NOTE

Overflow (V=1) can occur only when multiplying 32-bit operands to yield a 32-bit result. Overflow occurs if any of the high-order 32 bits of the quad word product are not equal to zero.

MULU — Unsigned Multiply — MULU

Instruction Format (word form):



Instruction Fields:

Register field —Specifies a data register as the destination.

Effective Address field —Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

MULU — Unsigned Multiply — MULU

Instruction Format (long form):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dl			0	SIZE	0	0	0	0	0	0	0	REGISTER Dh		

Instruction Fields:

Effective Address field — Specifies the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Register Dl field — Specifies a data register for the destination operand. The 32-bit multiplicand comes from this register, and the low-order 32 bits of the product are loaded into this register.

Size field — Selects a 32- or 64-bit product.

0 — 32-bit product to be returned to Register Dl.

1 — 64-bit product to be returned to Dh:Dl.

Register Dh field — If Size is 1, specifies the data register into which the high-order 32 bits of the product are loaded. If Dh = Dl and Size is 1, the results of the operation are undefined.

NBCD — Negate Decimal with Extend — NBCD

Operation: $0 - (\text{Destination}_{10}) - X \rightarrow \text{Destination}$

Assembler Syntax: NBCD ⟨ea⟩

Attributes: Size = (Byte)

Description: Subtracts the destination operand and the extend bit from zero. The operation is performed using binary coded decimal arithmetic. The packed BCD result is saved in the destination location. This instruction produces the tens complement of the destination if the extend bit is zero, or the nines complement if the extend bit is one.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

- X Set the same as the carry bit.
- N Undefined.
- Z Cleared if the result is non-zero. Unchanged otherwise.
- V Undefined.
- C Set if a decimal borrow occurs. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of the operation. This allows successful tests for zero results upon completion of multiple precision operations.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

NBCD — Negate Decimal with Extend — NBCD

Instruction Fields:

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NEG — Negate — NEG

Operation: 0 – (Destination) → Destination

Assembler Syntax: NEG ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the destination operand from zero and stores the result in the destination location.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X — Set the same as the carry bit.
- N — Set if the result is negative. Cleared otherwise.
- Z — Set if the result is zero. Cleared otherwise.
- V — Set if an overflow occurs. Cleared otherwise.
- C — Cleared if the result is zero. Set otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operation.

- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

NEG — Negate — NEG

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NEGX — Negate with Extend — NEGX

Operation: 0 – (Destination) – X → Destination

Assembler Syntax: NEGX ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the destination operand and the extend bit from zero. Stores the result in the destination location.

Condition Codes:

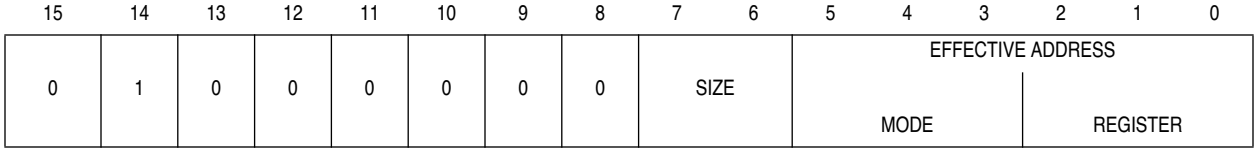
X	N	Z	V	C
*	*	*	*	*

- X — Set the same as the carry bit.
- N — Set if the result is negative. Cleared otherwise.
- Z — Set if the result is zero. Cleared otherwise.
- V — Set if an overflow occurs. Cleared otherwise.
- C — Cleared if the result is zero. Set otherwise.

NOTE

Normally, the Z condition bit is set via programming before the start of the operation. This allows successful tests for zero results upon completion of multiple precision operations.

Instruction Format:



Instruction Fields:

Size field — Specifies the size of the operation.

- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOP — No Operation — NOP

Operation: None

Assembler Syntax: NOP

Attributes: Unsized

Description: Performs no operation. The program counter is incremented, but processor state is otherwise unaffected. Execution continues with the instruction following the NOP instruction. The NOP instruction does not begin execution until all pending bus cycles are completed. This synchronizes the pipeline, and prevents instruction overlap.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

NOT — Logical Complement — NOT

Operation: $\overline{\text{Destination}} \rightarrow$ Destination

Assembler Syntax: NOT ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Calculates the ones complement of the destination operand and stores the result in the destination location.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

- X Not affected.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field — Specifies the size of the operation.

- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

OR — Inclusive Logical OR — OR

Operation: Source + Destination → Destination

Assembler Syntax: OR ⟨ea⟩, Dn
OR Dn, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an inclusive OR operation on the source operand and the destination operand and stores the result in the destination location. The contents of an address register may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER			OPMODE			EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) + (⟨Dn⟩) → Dn
100	101	110	(⟨Dn⟩) + (⟨ea⟩) → ea

Effective Address field — If the location specified is a source operand, only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTES:

1. If the destination is a data register, it must be specified using the destination Dn mode, not the destination ⟨ea⟩ mode.
2. Most assemblers use ORI when the source is immediate data.

ORI — Inclusive OR Immediate — ORI

Operation: Immediate Data; Destination → Destination

Assembler Syntax: ORI → #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

Description: Performs an inclusive OR operation on the immediate data and the destination operand and stores the result in the destination location. The size of the immediate data must match the operation size.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Always cleared

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Instruction Fields:

Size field — Specifies the size of the operation:

- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

Immediate field — (Data immediately following the instruction):

If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

ORI to CCR

Inclusive OR Immediate to Condition Code Register

ORI to CCR

Operation: Source; CCR → CCR

Assembler Syntax: ORI #⟨data⟩, CCR

Attributes: Size = (Byte)

Description: Performs an inclusive OR operation on the immediate operand and the condition codes and stores the result in the condition code register (low-order byte of the status register). All implemented bits of the condition code register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set if bit 4 of immediate operand is zero. Unchanged otherwise.
N Set if bit 3 of immediate operand is zero. Unchanged otherwise.
Z Set if bit 2 of immediate operand is zero. Unchanged otherwise.
V Set if bit 1 of immediate operand is zero. Unchanged otherwise.
C Set if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ORI to SR

Inclusive OR Immediate to Status Register (Privileged Instruction)

ORI to SR

Operation: If supervisor state
then Source; SR → SR
else TRAP

Assembler Syntax: ORI #⟨data⟩, SR

Attributes: Size = (Word)

Description: Performs an inclusive OR operation of the immediate operand and the contents of the status register and stores the result in the status register. All implemented bits of the status register are affected.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set if bit 4 of immediate operand is zero. Unchanged otherwise.
- N Set if bit 3 of immediate operand is zero. Unchanged otherwise.
- Z Set if bit 2 of immediate operand is zero. Unchanged otherwise.
- V Set if bit 1 of immediate operand is zero. Unchanged otherwise.
- C Set if bit 0 of immediate operand is zero. Unchanged otherwise.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
WORD DATA															

PEA — Push Effective Address — PEA

Operation: Sp – 4 → SP; ⟨ea⟩ → (SP)

Assembler Syntax: PEA ⟨ea⟩

Attributes: Size = (Long)

Description: Computes the effective address and pushes it onto the stack. The effective address must be a long word address.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Effective Address field — Specifies the address to be pushed onto the stack. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

RESET — Reset External Devices — RESET

(Privileged Instruction)

Operation: If supervisor state
then Assert RESET Line
else TRAP

Assembler Syntax: RESET

Attributes: Unsized

Description: Asserts the RESET signal for 512 clock periods, resetting all external devices. The processor state, other than the program counter, is unaffected and execution continues with the next instruction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

ROL, ROR Rotate (Without Extend) ROL, ROR

Operation: Destination Rotated by ⟨count⟩ → Destination

Assembler Syntax:
ROd Dx, Dy
ROd # ⟨data⟩, Dy
ROd ⟨ea⟩
where d is direction, L or R

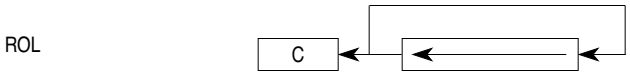
Attributes: Size = (Byte, Word, Long)

Description: Rotates the bits of the operand in the direction specified (L or R). The extend bit is not included in the rotation. For register rotation, the rotation count can be specified in either of two ways:

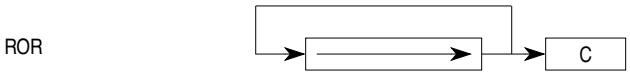
1. Immediate — The count (1-8) is specified by the instruction.
2. Register — The count is the value in the data register specified by the instruction, modulo 64.

The size of the operation for register destinations is specified as byte, word, or long. The contents of memory, ⟨ea⟩; can be rotated one bit only, and operand size is restricted to a word.

The ROL instruction rotates the bits of the operand to the left; the rotate count determines the number of bit positions rotated. Bits rotated out of the high-order bit go to the carry bit and also back into the low-order bit.



The ROR instruction rotates the bits of the operand to the right; the rotate count determines the number of bit positions rotated. Bits rotated out of the low-order bit go to the carry bit and also back into the high-order bit.



ROL, ROR — Rotate (Without Extend) — ROL, ROR

Condition Codes:

X	N	Z	V	C
—	*	*	0	*

X Not affected.
N Set if the most significant bit of the result is set. Cleared otherwise.
Z Set if the result is zero. Cleared otherwise.
V Always cleared.
C Set according to the last bit rotated out of the operand. Cleared when the rotate count is zero.

Instruction Format (Register Rotate):

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	1	1	0	COUNT/REGISTER	dr	SIZE	i/r	1	1	REGISTER

Instruction Fields (Register Rotate):

Count/Register field:
If i/r = 0, this field contains the rotate count. The values 1–7 represent counts of 1–7, and 0 specifies a count of 8.
If i/r = 1, this field specifies a data register that contains the rotate count (modulo 64).

dr field — Specifies the direction of the rotate:
0 — Rotate right
1 — Rotate left

Size field — Specifies the size of the operation:
00 — Byte operation
01 — Word operation
10 — Long operation

i/r field — Specifies the rotate count location:
If i/r = 0, immediate rotate count
If i/r = 1, register rotate count

Register field — Specifies a data register to be rotated

NOTE

Byte swapping in the low order word of a data register is best done with ROR/ROR, W #⟨8⟩, Dn. A special hardware assist has been provided to minimize operation execution.

ROL, ROR Rotate (Without Extend) ROL, ROR

Instruction Format (Memory Rotate):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Rotate):

dr field — Specifies the direction of the rotate:
- 0 — Rotate right
- 1 — Rotate left

Effective Address field — Specifies the operand to be rotated. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

ROXL, ROXR Rotate with Extend ROXL, ROXR

Operation: Destination Rotated with X by ⟨count⟩ → Destination

Assembler Syntax:
ROXd Dx, Dy
ROXd #⟨data⟩, Dy
ROXd ⟨ea⟩
where d is direction, L or R

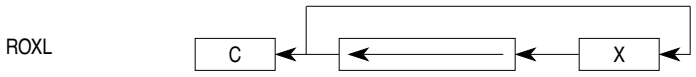
Attributes: Size = (Byte, Word, Long)

Description: Rotates the bits of the operand in the direction specified (L or R). The extend bit is included in the rotation. For register rotation, the rotation count can be specified in either of two ways:

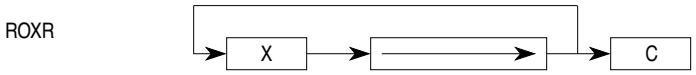
1. Immediate — The count (1–8) is specified by the instruction.
2. Register — The count is the value in the data register specified by the instruction, modulo 64.

The size of the operation for register destinations is specified as byte, word, or long. The contents of memory, ⟨ea⟩, can be rotated one bit only, and operand size is restricted to a word.

The ROXL instruction rotates the bits of the operand to the left; the rotate count determines the number of bit positions rotated. Bits rotated out of the high-order bit go to the carry bit and the extend bit; the previous value of the extend bit rotates into the low-order bit.



The ROXR instruction rotates the bits of the operand to the right; the rotate count determines the number of bit positions rotated. Bits rotated out of the low-order bit go to the carry bit and the extend bit; the previous value of the extend bit rotates into the high-order bit.



ROXL, ROXR Rotate with Extend ROXL, ROXR

Condition Codes:

X	N	Z	V	C
*	*	*	0	*

- X Set to the value of the last bit rotated out of the operand. Unaffected when count is zero.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Always cleared.
- C Set according to the last bit rotated out of the operand. Set to the value of the extend bit when count is zero.

Instruction Format (Register Rotate):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/REGISTER			dr	SIZE		i/r	1	0	REGISTER		

Instruction Fields (Register Rotate):

Count/Register field:
If i/r = 0, this field contains the rotate count. The values 1–7 represent counts of 1–7, and 0 specifies a count of 8.
If i/r = 1, this field specifies a data register that contains the rotate count (modulo 64).

dr field — Specifies the direction of the rotate:
0 — Rotate right
1 — Rotate left

Size field — Specifies the size of the operation:
00 — Byte operation
01 — Word operation
10 — Long operation

i/r field — Specifies the rotate count location:
If i/r = 0, immediate rotate count
If i/r = 1, register rotate count

Register field — Specifies a data register to be rotated

ROXL, ROXR Rotate with Extend ROXL, ROXR

Instruction Format (Memory Rotate):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields (Memory Rotate):

dr field — Specifies the direction of the rotate:
- 0 — Rotate right
- 1 — Rotate left

Effective Address field — Specifies the operand to be rotated. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

RTD — Return and Deallocate — RTD

Operation: (SP) → PC; SP + 4 + d → SP

Assembler Syntax: RTD #⟨displacement⟩

Attributes: Unsized

Description: Pulls the program counter value from the stack and adds the sign-extended 16-bit displacement value to the stack pointer. The previous program counter value is lost.

Condition Codes:

Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	0
DISPLACEMENT (16 BITS)															

Instruction Field:

Displacement field — Specifies the twos complement integer to be sign extended and added to the stack pointer.

RTE — Return from Exception — RTE

(Privileged Instruction)

Operation: If supervisor state
then (SP) → SR; SP + 2 → SP; (SP) → PC;
SP + 4 → SP;
restore state and de-allocate stack according to (SP)
else TRAP

Assembler Syntax: RTE

Attributes: Unsized

Description: Loads the processor state information stored in the exception stack frame located at the top of the stack into the processor. The instruction examines the stack format field in the format/offset word to determine how much information must be restored.

Condition Codes:

Set according to the condition code bits in the status register value restored from the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

Format/Offset word (in stack frame):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FORMAT				0	0	VECTOR OFFSET									

Format Field of Format/Offset Word:

Contains the format code, which implies the stack frame size (including the format/offset word).

0000 — Short Format, removes four words. Loads the status register and the program counter from the stack frame.

0001 — Throwaway Format, removes four words. Loads the status register from the stack frame and switches to the active system stack. Continues the instruction using the active system stack.

0010 — Instruction Error Format, removes six words. Loads the status register and the program counter from the stack frame and discards the other words.

1000 — MC68010 Long Format. The MC68020 takes a format error exception.

1001 — Coprocessor Mid-Instruction Format, removes 10 words. Resumes execution of coprocessor instruction.

1010 — MC68020 Short Format, removes 16 words and resumes instruction execution.

1011 — MC68020 Long Format, removes 46 words and resumes instruction execution.

Any other value in this field causes the processor to take a format error exception.

RTR — Return and Restore Condition Codes — RTR

Operation: (SP) → CCR; SP + 2 → SP;
(SP) → PC; SP + 4 → SP

Assembler Syntax: RTR

Attributes: Unsized

Description: Pulls the condition code and program counter values from the stack. The previous condition codes and program counter values are lost. The supervisor portion of the status register is unaffected.

Condition Codes:

Set to the condition codes from the stack.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

RTS — Return from Subroutine — RTS

Operation: (SP) → PC; SP + 4 → SP

Assembler Syntax: RTS

Attributes: Unsized

Description: Pulls the program counter value from the stack. The previous value is lost.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

SBCD — Subtract Decimal with Extend — SBCD

Operation: $Destination_{10} – Source_{10} – X \rightarrow Destination$

Assembler Syntax:
SBCD Dx, Dy
SBCD –(Ax), –(Ay)

Attributes: Size = (Byte)

Description: Subtracts the source operand and the extend bit from the destination operand and stores the result in the destination location. The subtraction is performed using binary coded decimal arithmetic; the operands are packed BCD numbers. The instruction has two modes:

1. Data register to data register: The data registers specified by the instruction contain the operands.
2. Memory to memory: The address registers specified by the instruction access the operands from memory using the predecrement addressing mode.

Condition Codes:

X	N	Z	V	C
*	U	*	U	*

- X Set the same as the carry bit.
- N Undefined.
- Z Cleared if the result is nonzero. Unchanged otherwise.
- V Undefined.
- C Set if a borrow (decimal) is generated. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	0	0	0	REGISTER Ry	1	0	0	0	0	R/M	REGISTER Rx

Instruction Fields:

Register Dy/Ay field — Specifies the destination register.
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

R/M field — Specifies the operand addressing mode:
0 — The operation is data register to data register.
1 — The operation is memory to memory.

Register Dx/Ax field — Specifies the source register:
If R/M = 0, specifies a data register.
If R/M = 1, specifies an address register for the predecrement addressing mode.

Scc — Set According to Condition Code — Scc

Operation: If Condition True
then set Destination
else clear Destination

Assembler Syntax: Scc ⟨ea⟩

Attributes: Size = (Byte)

Description: Tests the specified condition code. If the condition is true, sets all bits in the byte specified to 1 (TRUE). Otherwise, clears all bits to 0 (FALSE). Condition code cc specifies one of the following conditions:

cc	Name	Code	Description	cc	Name	Code	Description
CC	Carry Clear	0100	$\overline{C}$	LS	Low or Same	0011	$\overline{C}$; $\overline{Z}$
CS	Carry Set	0101	C	LT	Less Than	1101	N • $\overline{V}$; $\overline{N}$ • V
EQ	Equal	0111	Z	MI	Minus	1011	N
F	Never equal	0001	0	NE	Not Equal	0110	$\overline{Z}$
GE	Greater or Equal	1100	N • V; $\overline{N}$ • $\overline{V}$	PL	Plus	1010	$\overline{N}$
GT	Greater Than	1110	N • V • $\overline{Z}$; $\overline{N}$ • $\overline{V}$ • $\overline{Z}$	T	Always true	0000	1
HI	High	0010	$\overline{C}$ • $\overline{Z}$	VC	Overflow Clear	1000	$\overline{V}$
LE	Less or Equal	1111	Z; N • $\overline{V}$; $\overline{N}$ • V	VS	Overflow Set	1001	V

Condition Codes:
Not affected.

Instruction Format:

Scc — Set According to Condition Code — Scc

Instruction Fields:

Condition field — The binary code for one of the conditions listed in the table.
Effective Address field — Specifies the location in which the true/false byte is to be stored. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTE

A subsequent NEG.B instruction with the same effective address can be used to change the Scc result from TRUE or FALSE to the equivalent arithmetic value (TRUE = 1, FALSE = 0).

STOP — Load Status Register and Stop — STOP

(Privileged Instruction)

Operation: If supervisor state
then Immediate Data → SR; STOP
else TRAP

Assembler Syntax: STOP #⟨data⟩

Attributes: Unsized

Description: Moves the immediate operand into the status register (both user and supervisor portions), advances the program counter to point to the next instruction, and stops the fetching and executing of instructions. A trace, interrupt, or reset exception causes the processor to resume instruction execution. A trace exception occurs if instruction tracing is enabled (T0 = 1, T1=0) when the STOP instruction begins execution. If an interrupt request is asserted with a priority higher than the priority level set by the new status register value, an interrupt exception occurs; otherwise, the interrupt request is ignored. External reset always initiates reset exception processing.

Condition Codes:

Set according to the immediate operand.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
IMMEDIATE DATA															

Instruction Fields:

Immediate field — Specifies the data to be loaded into the status register.

SUB — Subtract — SUB

Operation: Destination – Source → Destination

Assembler Syntax: SUB ⟨ea⟩, Dn
SUB Dn, ⟨ea⟩

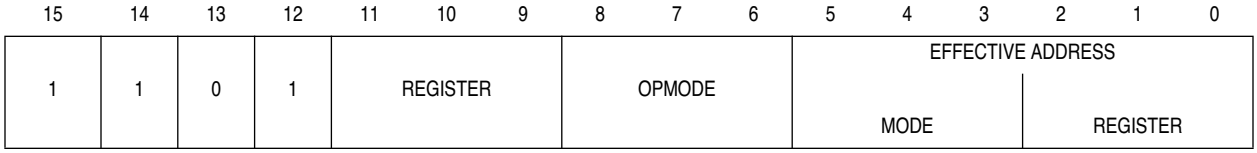
Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand from the destination operand and stores the result in the destination. The mode of the instruction indicates which operand is the source, which is the destination, and which is the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set to the value of the carry bit.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow is generated. Cleared otherwise.
- C Set if a borrow is generated. Cleared otherwise.



Instruction Fields:

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) – (⟨Dn⟩) → ⟨Dn⟩
100	101	110	(⟨Dn⟩) – (⟨ea⟩) → ⟨ea⟩

Effective Address field — Determines the addressing mode. If the location specified is a source operand, all addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An*	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*For byte size operation, address register direct is not allowed.

If the location specified is a destination operand, only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

NOTES:

1. If the destination is a data register, it must be specified as a destination Dn address, not as a destination ⟨ea⟩ address.
2. Most assemblers use SUBA when the destination is an address register, and SUBI or SUBQ when the source is immediate data.

SUBA — Subtract Address — SUBA

Operation: Destination – Source → Destination

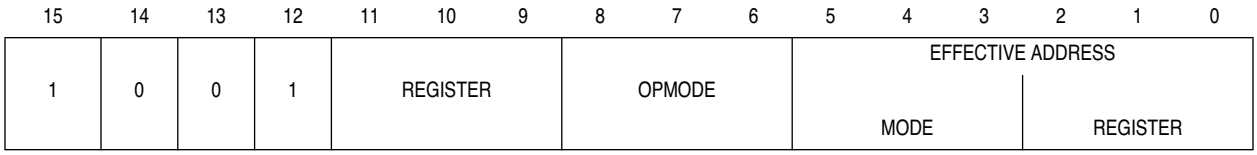
Assembler Syntax: SUBA ⟨ea⟩, An

Attributes: Size = (Word, Long)

Description: Subtracts the source operand from the destination address register and stores the result in the address register. Word size source operands are sign extended to 32-bit quantities prior to the subtraction.

Condition Codes:
Not affected.

Instruction Format:



Instruction Fields:

Register field — Specifies the destination, any of the eight address registers.

Opmode field — Specifies the size of the operation:

011 — Word operation. The source operand is sign extended to a long operand and theoperation is performed on the address register using all 32 bits.

111 — Long operation.

Effective Address field — Specifies the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

SUBI — Subtract Immediate — SUBI

Operation: Destination – Immediate Data → Destination

Assembler Syntax: SUBI #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

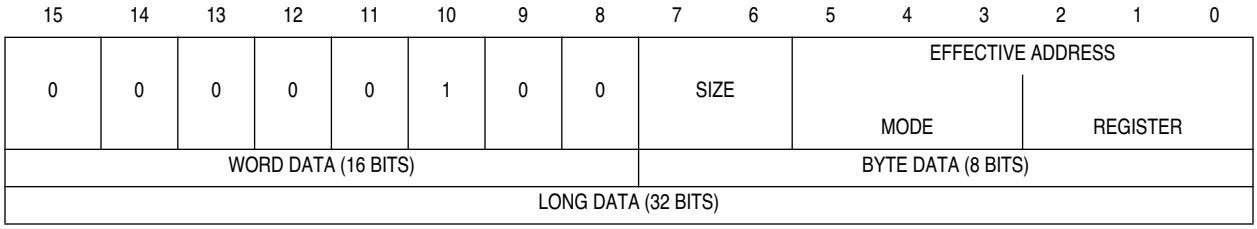
Description: Subtracts the immediate data from the destination operand and stores the result in the destination location. The size of the immediate data must match the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set to the value of the carry bit.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow occurs. Cleared otherwise.
- C Set if a borrow occurs. Cleared otherwise.

Instruction Format:



SUBI — Subtract Immediate — SUBI

Instruction Fields:

Size field — Specifies the size of the operation.

00 — Byte operation
01 — Word operation
10 — Long operation

Effective Address field — Specifies the destination operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

Immediate field — (Data immediately following the instruction)

If size = 00, the data is the low-order byte of the immediate word.
If size = 01, the data is the entire immediate word.
If size = 10, the data is the next two immediate words.

SUBQ — Subtract Quick — SUBQ

Operation: Destination – Immediate Data → Destination

Assembler Syntax: SUBQ #⟨data⟩, ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

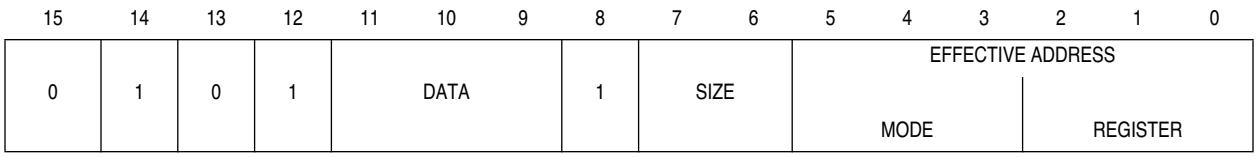
Description: Subtracts the immediate data (1–8) from the destination operand. Only word and long operations are allowed with address registers, and the condition codes are not affected. When subtracting from address registers, the entire destination address register is used, regardless of the operation size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set to the value of the carry bit.
- N Set if the result is negative. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if an overflow occurs. Cleared otherwise.
- C Set if a borrow occurs. Cleared otherwise.

Instruction Format:



SUBQ Subtract Quick SUBQ

Instruction Fields:

Data field — Three bits of immediate data; 1–7 represent immediate values of 1–7, and 0 represents 8.

Size field — Specifies the size of the operation:

00 — Byte operation
01 — Word operation
10 — Long operation

Effective Address field — Specifies the destination location. Only alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An*	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

*Word and long only

SUBX — Subtract with Extend — SUBX

Operation: Destination – Source – X → Destination

Assembler Syntax: SUBX Dx, Dy
SUBX –(Ax), –(Ay)

Attributes: Size = (Byte, Word, Long)

Description: Subtracts the source operand and the extend bit from the destination operand and stores the result in the destination location. The instruction has two modes:

1. Register to register: Data registers specified by the instruction contain the operands.
2. Memory to memory: Address registers specified by the instruction access operands from memory using predecrement addressing mode.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

- X Set to the value of the carry bit.
- N Set if the result is negative. Cleared otherwise.
- Z Cleared if the result is nonzero. Unchanged otherwise.
- V Set if an overflow occurs. Cleared otherwise.
- C Set if a carry occurs. Cleared otherwise.

NOTE

Normally the Z condition code bit is set via programming before the start of an operation. This allows successful tests for zero results upon completion of multiple-precision operations.

SUBX — Subtract with Extend — SUBX

Instruction Format:

15	14	13	12	11 10 9	8	7 6	5	4	3	2 1 0
1	0	0	1	REGISTER Rx	1	SIZE	0	0	R/M	REGISTER Ry

Instruction Fields:

Register Dy/Ay field — Specifies the destination register:

If R/M = 0, specifies a data register.

If R/M = 1, specifies an address register for the predecrement addressing mode.

Size field — Specifies the size of the operation:

00 — Byte operation

01 — Word operation

10 — Long operation

R/M field — Specifies the operand addressing mode:

0 — The operation is data register to data register.

1 — The operation is memory to memory.

Register Dx/Ax field — Specifies the source register:

If R/M = 0, specifies a data register.

If R/M = 1, specifies an address register for the predecrement addressing mode.

SWAP — Swap Register Halves — SWAP

Operation: Register [31:16] ↔ Register [15:0]

Assembler Syntax: SWAP Dn

Attributes: Size = (Word)

Description: Exchange the 16-bit words (halves) of a data register.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the 32-bit result is set. Cleared otherwise.
Z Set if the 32-bit result is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	REGISTER		

Instruction Fields:

Register field — Specifies the data register to swap.

TBLS TBLSN — Table Lookup and Interpolate (Signed) — TBLS TBLSN

Operation:

Rounded:
$ENTRY_{(n)} + \{(ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx\ [7:0]\} / 256 \rightarrow Dx$
Unrounded:
$ENTRY_{(n)} * 256 + \{(ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx\ [7:0]\} \rightarrow Dx$

Where $ENTRY_{(n)}$ and $ENTRY_{(n+1)}$ are either:
1. Consecutive entries in the table pointed to by the ⟨ea⟩ and indexed by Dx [15:8] * size or,
2. The registers Dym, Dyn, respectively

Assembler Syntax:

TBLS.⟨size⟩⟨ea⟩,Dx(Result rounded)
TBLSN.⟨size⟩⟨ea⟩,Dx(Result not rounded)
TBLS.⟨size⟩Dym:Dyn,Dx(Result rounded)
TBLSN.⟨size⟩Dym:Dyn, Dx(Result not rounded)

Attributes: Size = (Byte, Word, Long)

Description: The signed table lookup and interpolate instruction, TBLS, allows the efficient use of compressed linear data tables to model complex functions. The TBLS instruction has two modes of operation: table lookup and interpolate mode, and data register interpolate mode.

For table lookup and interpolate mode, data register Dx [15:0] contains the independent variable X. The effective address points to the start of a signed byte, word, or long-word table containing a linear representation of the dependent variable, Y, as a function of X. In general, the independent variable, located in the low-order word of Dx, consists of an 8-bit integer part and an 8-bit fractional part. An assumed radix point is located between bits 7 and 8. The integer part, Dx [15:8], is scaled by the operand size and is used as an offset into the table. The selected entry in the table is subtracted from the next consecutive entry. A fractional portion of this difference is taken by multiplying by the interpolation fraction, Dx [7:0]. The adjusted difference is then added to the selected table entry. The result is returned in the destination data register, Dx.

For register interpolate mode, the interpolation occurs using the Dym and Dyn registers in place of the two table entries. For this mode, only the fractional portion, Dx [7:0], is used in the interpolation, and the integer portion, Dx [15:8], is ignored. The register interpolation mode may be used with several table lookup and interpolations to model multidimensional functions.

TBLS TBLSN — Table Lookup and Interpolate (Signed) — TBLS TBLSN

Signed table entries range from -2^{n-1} to $2^{n-1} - 1$, where n is 8, 16, or 32 for byte, word, and long-word tables, respectively.

Rounding of the result is optionally selected via the 'R' instruction field. If R = 0 (TBLS), the fractional portion is rounded according to the round-to-nearest algorithm. The rounding procedure can be summarized by the following table.

Adjusted Difference Fraction	Rounding Adjustment
$n \le -\int$	–1
$-\int < n < \int$	+0
$n \ge \int$	+1

The adjusted difference is then added to the selected table entry. The rounded result is returned in the destination data register, Dx. Only the portion of the register corresponding to the selected size is affected.

	31 24	23 16	15 8	7 0
BYTE	UNAFFECTED	UNAFFECTED	UNAFFECTED	RESULT
WORD	UNAFFECTED	UNAFFECTED	RESULT	RESULT
LONG	RESULT	RESULT	RESULT	RESULT

If R =1 (TBLSN), the result is returned in register Dx without rounding. If the size is byte, the integer portion of the result is returned in Dx [15:8]. The integer portion of a word result is stored in Dx [23:8]. The least significant 24 bits of a long result are stored in Dx [31:8]. Byte and word results are sign extended to fill the entire 32-bit register.

	31 24	23 16	15 8	7 0
BYTE	SIGN EXTENDED	SIGN EXTENDED	RESULT	FRACTION
WORD	SIGN EXTENDED	RESULT	RESULT	FRACTION
LONG	RESULT	RESULT	RESULT	FRACTION

NOTE

A long-word result contains only the least significant 24 bits of integer precision.

TBLS TBLSN — Table Lookup and Interpolate (Signed) — TBLS TBLSN

For all sizes, the 8-bit fractional portion of the result is returned in the low byte of the data register, Dx [7:0]. User software can make use of the fractional data to reduce cumulative errors in lengthy calculations or implement rounding algorithms different from those provided by other forms of TBLS. The assumed radix point described previously places two restrictions on the programmer:

1. Tables are limited to 257 entries in length.
2. Interpolation resolution is limited to 1/256 the distance between consecutive table entries. The assumed radix point should not, however, be construed by the programmer as a requirement that the independent variable be calculated as a fractional number in the range $0 \le X \le 255$. On the contrary, X should be considered an integer in the range $0 \le X \le 65535$, realizing that the table is actually a compressed linear representation of a function in which only every 256th value is actually stored in memory.

See **4.6 Table Lookup and Interpolation Instructions** for more information on the TBLS/TBLSN instruction.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if the integer portion of an unrounded long result is not in the range, $-(2^{23}) \le \text{Result} \le (2^{23}) - 1$. Cleared otherwise.
- C Always cleared.

Instruction Format:

Table lookup and interpolate:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dx			1	R	0	1	SIZE		0	0	0	0	0	0

TBLS TBLSN — Table Lookup and Interpolate (Signed) — TBLS TBLSN

Data Register Interpolate:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	REGISTER Dym		
0	REGISTER Dx			1	R	0	0	SIZE		0	0	0	REGISTER Dyn		

Instruction Fields:

Effective address field (table lookup and interpolate mode only:

Specifies the source location. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Size field:

Specifies the size of operation.

00 — byte operation
01 — word operation
10 — long operation

Register field:

Specifies the destination data register, Dx. On entry, the register contains the interpolation fraction and entry number.

Dym, Dyn field:

If the effective address mode field is nonzero, this operand register is unused and should be zero. If the effective address mode field is zero, the surface interpolation variant of this instruction is implied, and Dyn specifies one of the two source operands.

Rounding mode field:

The ’R’ bit controls rounding of the final result. When R = 0, the result is rounded according to the round-to-nearest algorithm. When R = 1, the result is returned unrounded.

TBLU TBLUN — Table Lookup and Interpolate (Unsigned) — TBLU TBLUN

Operation:

Rounded:

$ENTRY_{(n)} + \{(ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx\ [7:0]\} / 256 \rightarrow Dx$

Unrounded:

$ENTRY_{(n)} * 256 + \{(ENTRY_{(n+1)} - ENTRY_{(n)}) * Dx\ [7:0]\} \rightarrow Dx$

Where $ENTRY_{(n)}$ and $ENTRY_{(n+1)}$ are either:
1. Consecutive entries in the table pointed to by the ⟨ea⟩ and indexed by Dx [15:8] * size or,
2. The registers Dym, Dyn respectively

Assembler Syntax:

TBLU.⟨size⟩⟨ea⟩,Dx(Result rounded)
TBLUN.⟨size⟩⟨ea⟩,Dx(Result not rounded)
TBLU.⟨size⟩Dym:Dyn,Dx(Result rounded)
TBLUN.⟨size⟩Dym:Dyn, Dx(Result not rounded)

Attributes: Size = (Byte, Word, Long)

Description: The signed table lookup and interpolate instruction, TBLU, allows the efficient use of compressed linear data tables to model complex functions. The TBLU instruction has two modes of operation: table lookup and interpolate mode, and data register interpolate mode.

For table lookup and interpolate mode, data register Dx [15:0] contains the independent variable X. The effective address points to the start of a signed byte, word, or long-word table containing a linear representation of the dependent variable, Y, as a function of X. In general, the independent variable, located in the low-order word of Dx, consists of an 8-bit integer part and an 8-bit fractional part. An assumed radix point is located between bits 7 and 8. The integer part, Dx [15:8], is scaled by the operand size and is used as an offset into the table. The selected entry in the table is subtracted from the next consecutive entry. A fractional portion of this difference is taken by multiplying by the interpolation fraction, Dx [7:0]. The adjusted difference is then added to the selected table entry. The result is returned in the destination data register, Dx.

For register interpolate mode, the interpolation occurs using the Dym and Dyn registers in place of the two table entries. For this mode, only the fractional portion, Dx [7:0], is used in the interpolation, and the integer portion, Dx [15:8], is ignored. The register interpolation mode may be used with several table lookup and interpolations to model multidimensional functions.

TBLU TBLUN — Table Lookup and Interpolate (Unsigned) — TBLU TBLUN

Unsigned table entries range from 0 to 2^{n-1} where n is 8, 16, or 32 for byte, word, and long-word tables, respectively. Unsigned and unrounded table results are zero extended.

Rounding of the result is optionally selected via the 'R' instruction field. If R = 0 (TBLU), the fractional portion is rounded according to the round-to-nearest algorithm. The rounding procedure can be summarized by the following table.

Adjusted Difference Fraction	Rounding Adjustment
n < ∫	+0
n ≥ ∫	+1

The adjusted difference is then added to the selected table entry. The rounded result is returned in the destination data register, Dx. Only the portion of the register corresponding to the selected size is affected.

	31 24	23 16	15 8	7 0
BYTE	UNAFFECTED	UNAFFECTED	UNAFFECTED	RESULT
WORD	UNAFFECTED	UNAFFECTED	RESULT	RESULT
LONG	RESULT	RESULT	RESULT	RESULT

If R = 1 (TABLUN), the result is returned in register Dx without rounding. If the size is byte, the integer portion of the result is returned in Dx (15:8). The integer portion of a word result is stored in Dx (23:8). The least significant 24 bits of a long result are stored in Dx (31:8). Byte and word results are zero extended to fill the entire 32-bit register.

	31 24	23 16	15 8	7 0
BYTE	ZERO EXTENDED	ZERO EXTENDED	RESULT	FRACTION
WORD	ZERO EXTENDED	RESULT	RESULT	FRACTION
LONG	RESULT	RESULT	RESULT	FRACTION

NOTE

A long-word result contains only the least significant 24 bits of integer precision.

TBLU TBLUN — Table Lookup and Interpolate (Unsigned) — TBLU TBLUN

For all sizes, the 8-bit fractional portion of the result is returned in the low byte of the data register, Dx (7:0). User software can make use of the fractional data to reduce cumulative errors in lengthy calculations or implement rounding algorithms different from those provided by other forms of TBLU. The assumed radix point described previously places two restrictions on the programmer:

1. Tables are limited to 257 entries in length.
2. Interpolation resolution is limited to 1/256 the distance between consecutive table entries.The assumed radix point should not, however, be construed by the programmer as a requirement that the independent variable be calculated as a fractional number in the range $0 \leq X \leq 255$. On the contrary, X should be considered to be an integer in the range $0 \leq X \leq 65535$, realizing that the table is actually a compressed linear representation of a function in which only every 256th value is actually stored in memory.

See **4.6 Table Lookup and Interpolation Instructions** for more information on the TBLU/TBLUN instruction.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- X Not affected.
- N Set if the most significant bit of the result is set. Cleared otherwise.
- Z Set if the result is zero. Cleared otherwise.
- V Set if the integer portion of an unrounded long result is not in the range,$0 \leq$ Result $\leq (2^{24}) - 1$. Cleared otherwise.
- C Always cleared.

Instruction Format:

Table Lookup and Interpolate:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dx			0	R	0	1	SIZE		0	0	0	0	0	0

TBLU TBLUN — Table Lookup and Interpolate (Unsigned) — TBLU TBLUN

Data Register Interpolate:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	REGISTER Dym		
0	REGISTER Dx			0	R	0	0	SIZE		0	0	0	REGISTER Dyn		

Instruction Fields:

Effective address field (table lookup and interpolate mode only):

Specifies the source location. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	—	—		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	—	—				
– (An)	—	—				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

Size field:

Specifies the size of operation.

00 — byte operation
01 — word operation
10 — long operation

Register field:

Specifies the destination data register, Dx. On entry, the register contains the interpolation fraction and entry number.

Dym, Dyn field:

If the effective address mode field is nonzero, this operand register is unused and should be zero. If the effective address mode field is zero, the surface interpolation variant of this instruction is implied, and Dyn specifies one of the two source operands.

Rounding mode field:

The ’R’ bit controls rounding of the final result. When R = 0, the result is rounded according to the round-to-nearest algorithm. When R = 1, the result is returned unrounded.

TAS — Test and Set an Operand — TAS

Operation: Destination Tested → Condition Codes; 1 → bit 7 of Destination

Assembler Syntax: TAS ⟨ea⟩

Attributes: Size = (Byte)

Description: Tests and sets the byte operand addressed by the effective address field. The instruction tests the current value of the operand and sets the N and Z condition bits appropriately. TAS also sets the high-order bit of the operand. The operation uses a read-modify-write memory cycle that completes the operation without interruption. This instruction supports use of a flag to coordinate several processors.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the most significant bit of the operand is currently set. Cleared otherwise.
Z Set if the operand was zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

TAS — Test and Set an Operand — TAS

Instruction Fields: Effective Address field — Specifies the location of the tested operand. Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An	—	—		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	—	—
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	—	—
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	—	—
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	—	—

TRAP — Trap — TRAP

Operation: SSP – 2 → SSP; Format/Offset → (SSP);
SSP – 4 → SSP; PC → (SSP); SSP – 2 → SSP;
SR → (SSP); Vector Address → PC

Assembler Syntax: TRAP #⟨vector⟩

Attributes: Unsized

Description: Causes a TRAP #⟨vector⟩ exception. A vector number is generated by adding the immediate vector operand to 32. The range of vector operand values is 0–5, thus there are 16 possible vector numbers.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	VECTOR			

Instruction Fields:
Vector field — Specifies the trap vector to be taken.

TRAPcc — Trap on Condition — TRAPcc

Operation: If cc then TRAP

Assembler Syntax: TRAPcc
TRAPcc.W #⟨data⟩TRAPcc.L #⟨data⟩

Attributes: Unsized or Size = (Word, Long)

Description: If the specified condition is true, causes a TRAPcc exception (vector number 7). The address of the next instruction word (current PC) is pushed onto the stack. If the condition is not true, the processor performs no operation and execution continues with the next instruction. The immediate data operand must be placed in the word(s) immediately following the operation word. It is available to the trap handler. Condition code cc specifies one of the following conditions.

cc	Name	Code	Description	cc	Name	Code	Description
CC	Carry Clear	0100	$\overline{C}$	LS	Low or Same	0011	$\overline{C}$; $\overline{Z}$
CS	Carry Set	0101	C	LT	Less Than	1101	N • $\overline{V}$; $\overline{N}$ • V
EQ	Equal	0111	Z	MI	Minus	1011	N
F	Never equal	0001	0	NE	Not Equal	0110	$\overline{Z}$
GE	Greater or Equal	1100	N • V; $\overline{N}$ • $\overline{V}$	PL	Plus	1010	$\overline{N}$
GT	Greater Than	1110	N • V • $\overline{Z}$; $\overline{N}$ • $\overline{V}$ • $\overline{Z}$	T	Always true	0000	1
HI	High	0010	$\overline{C}$ • $\overline{Z}$	VC	Overflow Clear	1000	$\overline{V}$
LE	Less or Equal	1111	Z; N • $\overline{V}$; $\overline{N}$ • V	VS	Overflow Set	1001	V

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	CONDITION				1	1	1	1	1	OPMODE		
OPTIONAL WORD															
OR LONG WORD															

Instruction Fields:
Condition field — The binary code for one of the conditions listed in the table.
Opmode field — Selects the instruction form.
010 — Instruction is followed by word-size operand.
011 — Instruction is followed by long-word-size operand.
100 — Instruction has no operand.

TRAPV — Trap on Overflow — TRAPV

Operation: If V then TRAP

Assembler Syntax: TRAPV

Attributes: Unsized

Description: If the CCR overflow bit is set, there is a TRAPV exception (vector number 7). If the bit is not set, the processor performs no operation and execution continues with the next instruction.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

TST — Test an Operand — TST

Operation: Destination Tested → Condition Codes

Assembler Syntax: TST ⟨ea⟩

Attributes: Size = (Byte, Word, Long)

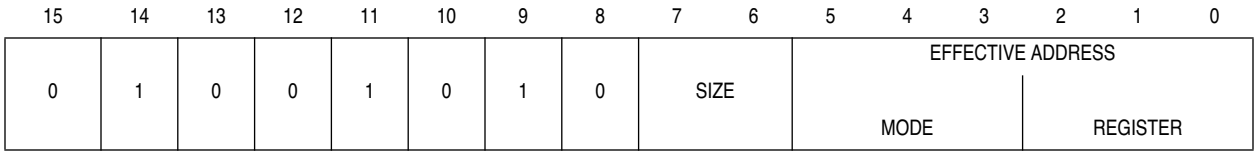
Description: Compares the operand with zero and sets condition codes according to the results of the test.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected.
N Set if the operand is negative. Cleared otherwise.
Z Set if the operand is zero. Cleared otherwise.
V Always cleared.
C Always cleared.

Instruction Format:



Instruction Fields:

Size field — Specifies the size of the operation:
- 00 — Byte operation
- 01 — Word operation
- 10 — Long operation

Effective Address field — Specifies the destination operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register		Addressing Mode	Mode	Register
Dn	000	Reg. number: Dn		(xxx).W	111	000
An*	001	Reg. number: An		(xxx).L	111	001
(An)	010	Reg. number: An		#⟨data⟩	111	100
(An) +	011	Reg. number: An				
– (An)	100	Reg. number: An				
(d_{16}, An)	101	Reg. number: An		(d_{16}, PC)	111	010
(d_8, An, Xn)	110	Reg. number: An		(d_8, PC, Xn)	111	011
(bd, An, Xn)	110	Reg. number: An		(bd, PC, Xn)	111	011

*Word or long word operation only

UNLK — Unlink — UNLK

Operation: An → SP; (SP) → An; SP + 4 → SP

Assembler Syntax: UNLK An

Attributes: Unsized

Description: Loads the stack pointer from the specified address register then loads the address register with a long word pulled from the top of the stack.

Condition Codes:
Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	REGISTER		

Instruction Fields:
Register field — Specifies the address register for the instruction.

4.5 Instruction Format Summary

A summary of the primary words in each instruction of the instruction set follows. The complete instruction definition consists of the primary words followed by the addressing mode operands such as immediate data fields, displacements, and index operands. The four most significant bits of the first (or only) primary word provide a means of categorizing the instructions. Table 4-11 is an operation code (opcode) map that lists an instruction category for each combination of these bits.

Table 4-11 Operation Code Map

Bits [15:12]	Operation
0000	Bit Manipulation/MOVEP/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/Scc/DBcc/TRAPcc
0110	Bcc/BSR/BRA
0111	MOVEQ
1000	OR/DIV/SBCD
1001	SUB/SUBX
1010	(Unassigned, Reserved)
1011	CMP/ EOR
1100	AND/MUL/ABCD/EXG
1101	ADD/ADDX
1110	Shift/Rotate/Bit Field
1111	Table Lookup and Interpolation

ORI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

ORI to CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ORI to SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
WORD DATA															

CMP2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	SIZE		0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
D/A	REGISTER			0	0	0	0	0	0	0	0	0	0	0	0

Size Field: 00 = Byte 01 = Word 10 = Long

CHK2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	SIZE		0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
D/A	REGISTER			1	0	0	0	0	0	0	0	0	0	0	0

Size Field: 00 = Byte 01 = Word 10 = Long

BTST (Dynamic)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

BCHG (Dynamic)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

BCLR (Dynamic)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	1	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

BSET (Dynamic)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	REGISTER			1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

MOVEP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	DATA REGISTER			OPMODE			0	0	1	ADDR REGISTER		
DISPLACEMENT (16 BITS)															

OPMODE FIELD: 100 = Transfer Word From Memory to Register
101 = Transfer Long From Memory to Register
110 = Transfer Word From Register to Memory
111 = Transfer Word From Register to Memory

ANDI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

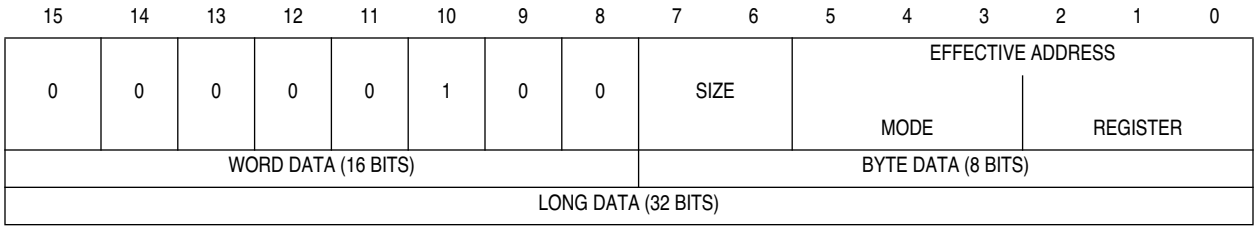
ANDI to CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

ANDI to SR

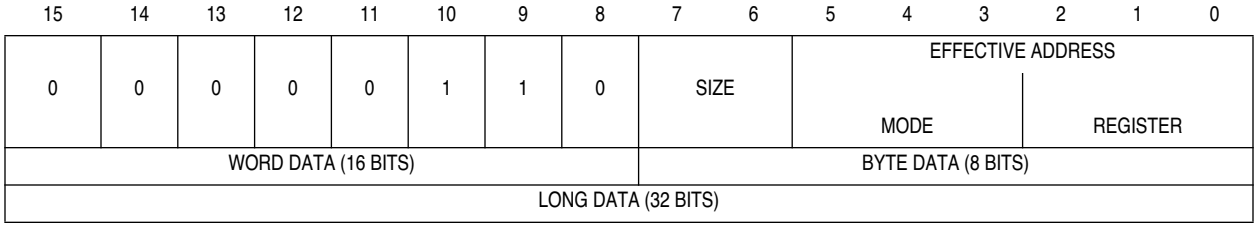
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0
WORD DATA															

SUBI



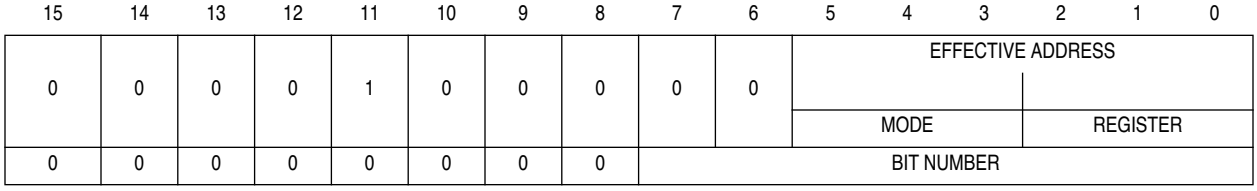
Size Field: 00 = Byte 01 = Word 10 = Long

ADDI



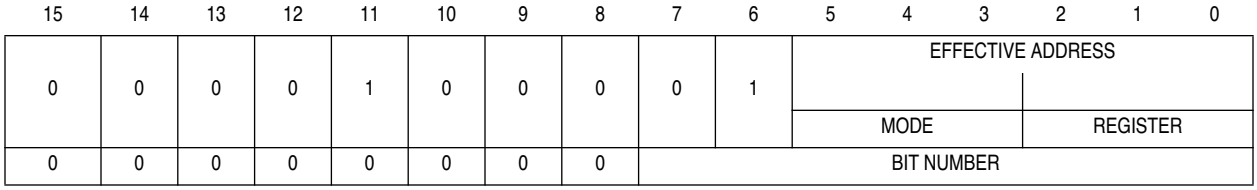
Size Field: 00 = Byte 01 = Word 10 = Long

BTST (Static)



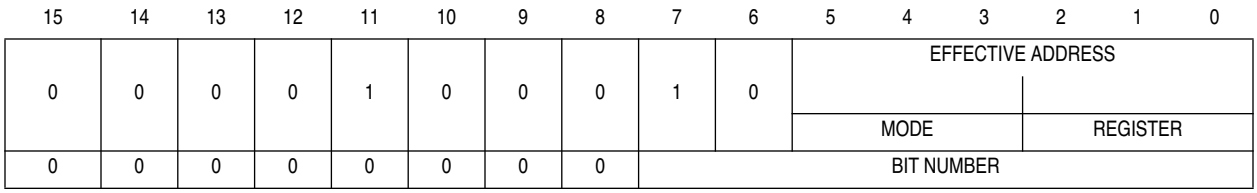
Bit Number Field: Modulo 32-bit selection

BCHG (Static)



Bit Number Field: Modulo 32-bit selection

BCLR (Static)



Bit Number Field: Modulo 32-bit selection

BSET (Static)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

Bit Number Field: Modulo 32-bit selection

EORI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

Size Field: 00 = Byte 01 = Word 10 = Long

EORI to CCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	BYTE DATA (8 BITS)							

EORI to SR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

CMPI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
WORD DATA (16 BITS)								BYTE DATA (8 BITS)							
LONG DATA (32 BITS)															

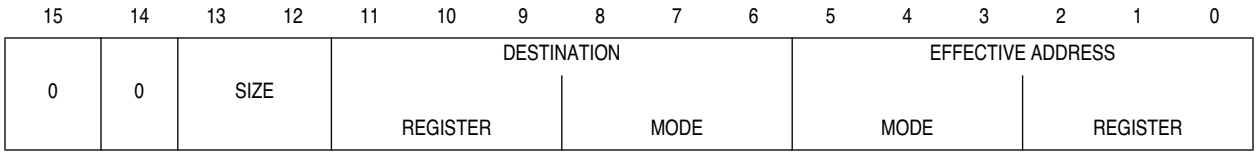
Size Field: 00 = Byte 01 = Word 10 = Long

MOVES

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
A/D	REGISTER			dr	0	0	0	0	0	0	0	0	0	0	0

dr Field: 0 = EA to Register 1 = Register to EA

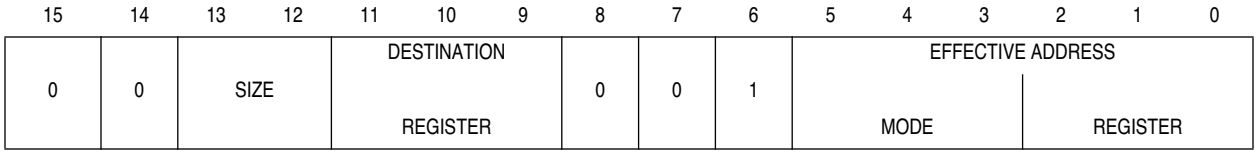
MOVE



Size Field: 00 = Byte 01 = Word 10 = Long

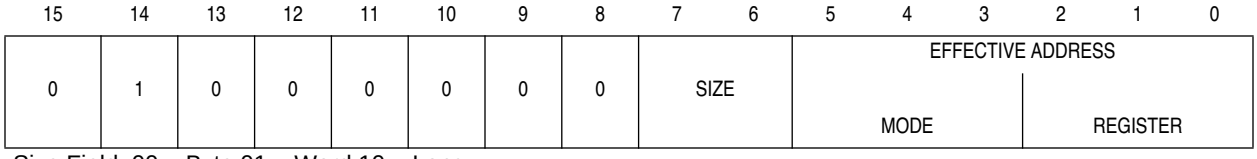
Note register and mode locations.

MOVEA



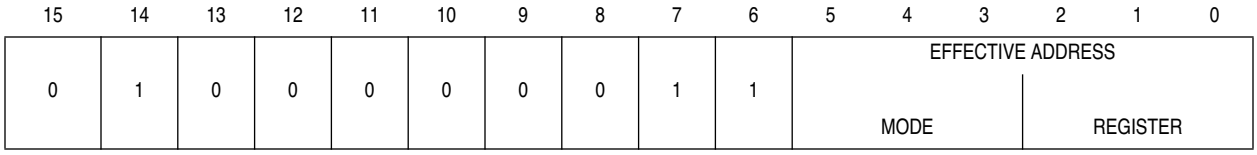
Size Field: 00 = Byte 01 = Word 10 = Long

NEGX

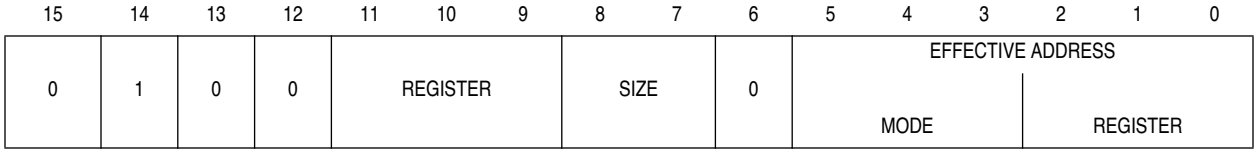


Size Field: 00 = Byte 01 = Word 10 = Long

MOVE from SR

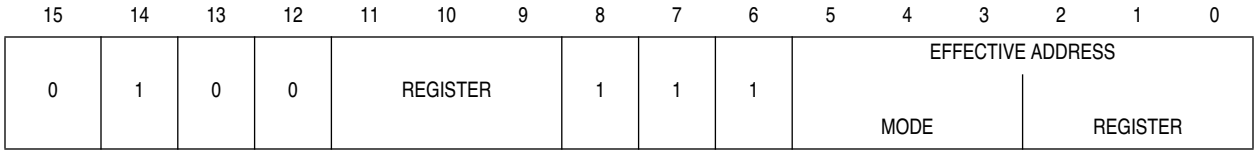


CHK

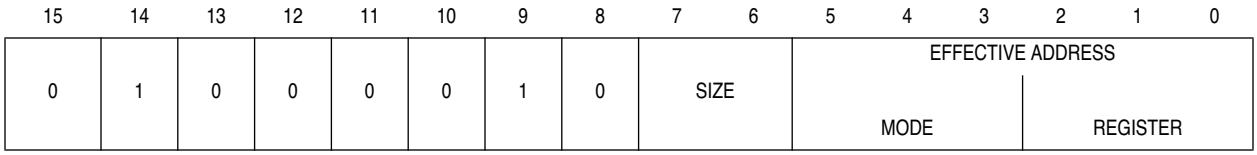


Size Field: 00 = Byte 01 = Word 10 = Long

LEA

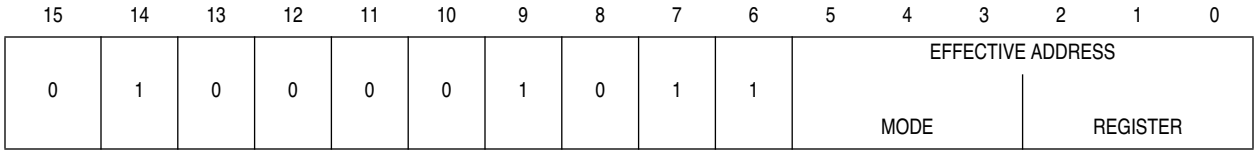


CLR

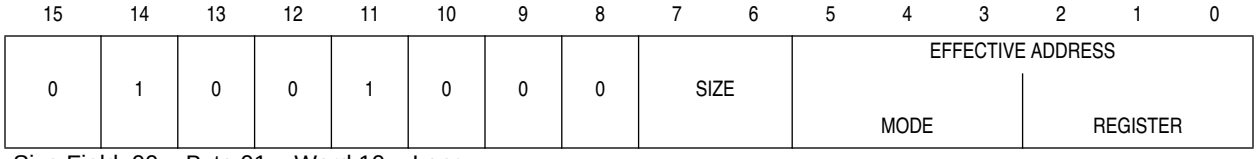


Size Field: 00 = Byte 01 = Word 10 = Long

MOVE from CCR

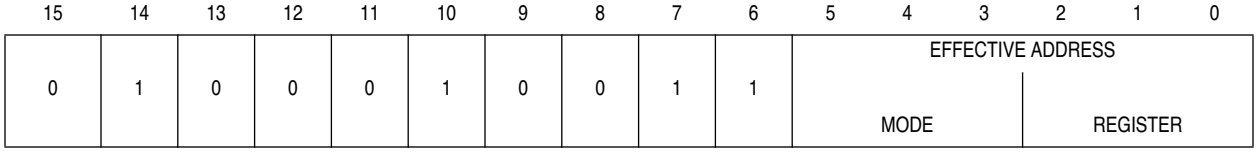


NEG

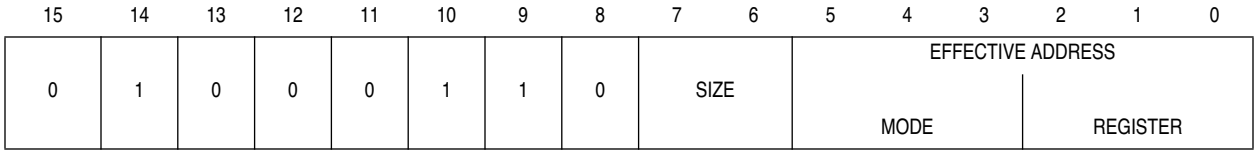


Size Field: 00 = Byte 01 = Word 10 = Long

MOVE to CCR

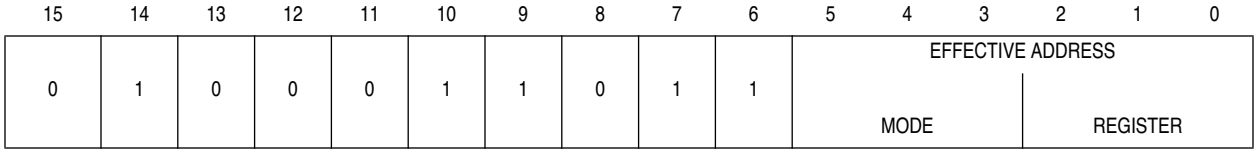


NOT

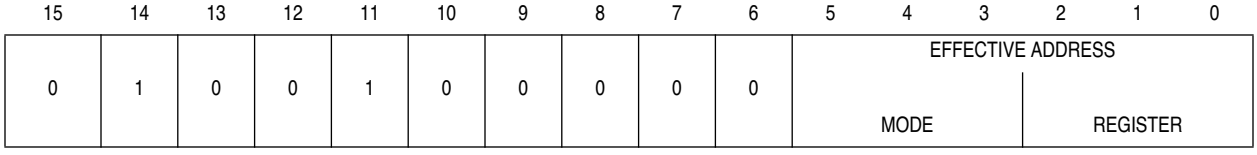


Size Field: 00 = Byte 01 = Word 10 = Long

MOVE to SR



NBCD



LINK Long

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	0	0	0	1	REGISTER		
HIGH-ORDER DISPLACEMENT															
LOW-ORDER DISPLACEMENT															

SWAP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	0	REGISTER		

BKPT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	0	0	1	VECTOR		

PEA

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

EXT, EXTB

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	0	OPMODE			0	0	0	REGISTER		

Opmode Field: 010 = Extend Word 011 = Extend Long 111 = Extend Byte Long

MOVEM

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	dr	0	0	1	SIZE	EFFECTIVE ADDRESS					
										MODE			REGISTER		
REGISTER LIST MASK															

Size Field: 00 = Byte 01 = Word 10 = Long

Register to EA Mask

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

EA to Register Mask

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D0	D1	D2	D3	D4	D5	D6	D7	A0	A1	A2	A3	A4	A5	A6	A7

TST

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Size Field: 00 = Byte 01 = Word 10 = Long

TAS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

BGND

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	0	1	0

ILLEGAL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	1	1	1	1	0	0

MULU (Long)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dl			0	SIZE	0	0	0	0	0	0	0	REGISTER Dh		

Size Field: 0 = Long Word Product 1 = Quad Word Product

MULS (Long)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dq			1	SIZE	0	0	0	0	0	0	0	REGISTER Dr		

Size Field: 0 = Long Word Product 1 = Quad Word Product

TRAP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	0	VECTOR			

LINK (Word)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	0	REGISTER		
WORD DISPLACEMENT															

UNLK

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	0	1	1	REGISTER		

MOVE USP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	0	DR	REGISTER		

DR Field: 0 = Move An to USP 1 = Move USP to An

RESET

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	0

NOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	0	1

STOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	0
IMMEDIATE DATA															

RTE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1

Format/Offset Word (in stack frame)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FORMAT				0	0	VECTOR OFFSET									

Format Field: Four bits imply frame size; only values 000–0010 and 1000–1011 are used.

RTD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	0
DISPLACEMENT (16 BITS)															

RTS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	0	1

TRAPV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	0

RTR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	0	1	1	1

MOVEC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
A/D	REGISTER			CONTROL REGISTER											

dr Field: 0 = Control Register to General Register 1 = General Register to Control Register

Control Register Field: $000 = SFC$801 = VBR
$001 = DFC$802 = CAAR
$002 = CACR$803 = MSP
$800 = USP$804 = ISP

JSR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

JMP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

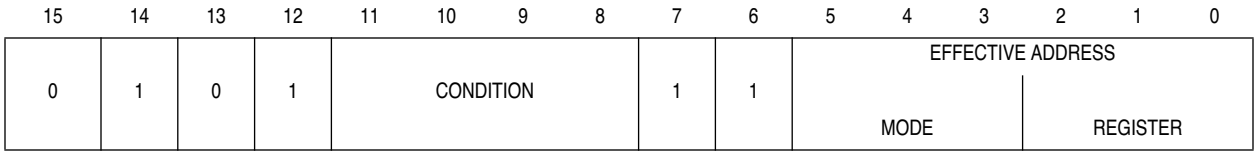
ADDQ

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	DATA			0	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

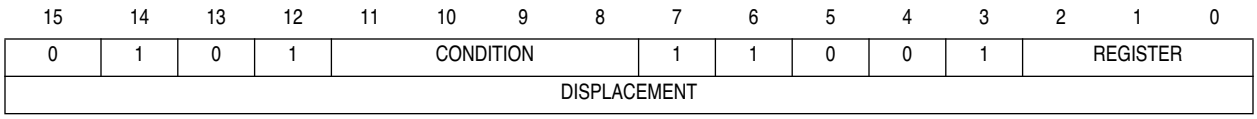
Data Field: Three bits of immediate data; 000–111 represent values of 1–7; 000 represents 8

Size Field: 00 = Byte 01 = Word 10 = Long

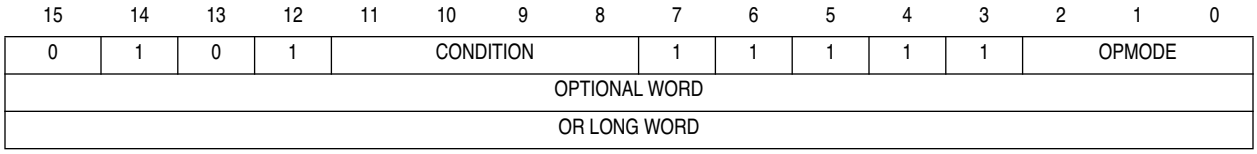
Scc



DBcc

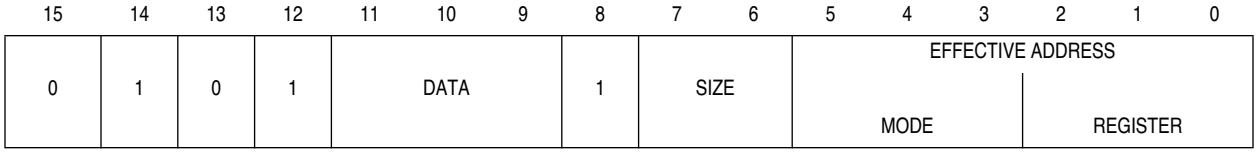


TRAPcc



Opmode Field: 010 = Word Operand 011 = Long Operand 100 = No Operand

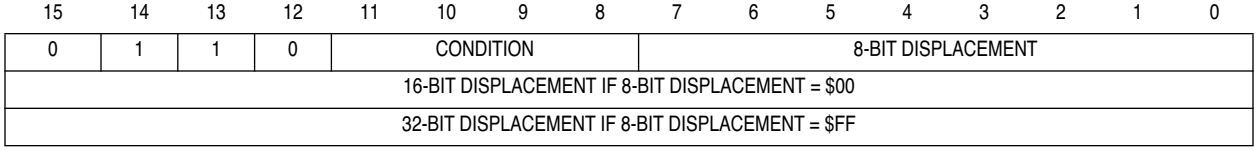
SUBQ



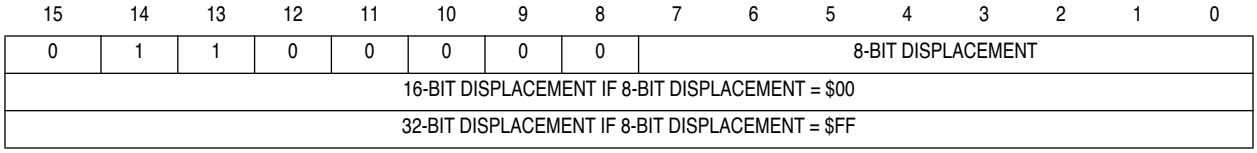
Data Field: Three bits of immediate data; 000–111 represent values of 1–7; 000 represents 8

Size Field: 00 = Byte 01 = Word 10 = Long

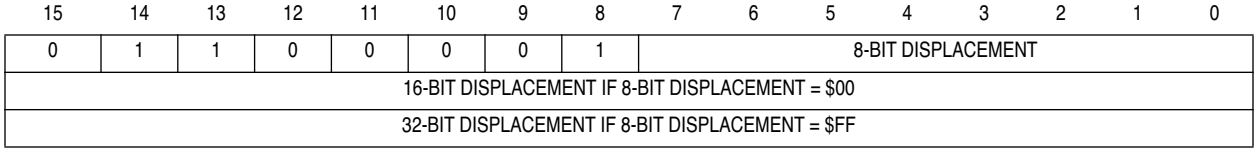
Bcc



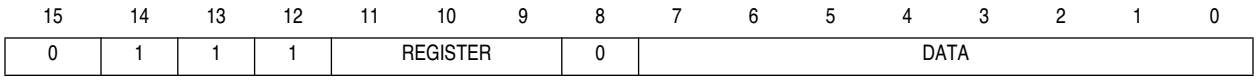
BRA



BSR

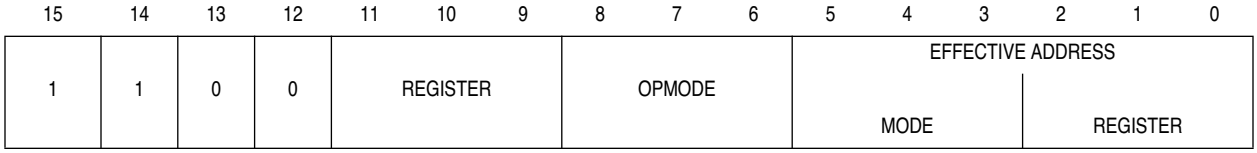


MOVEQ



Data Field: Data is sign extended to a long operand, and all 32 bits are transferred to the data register.

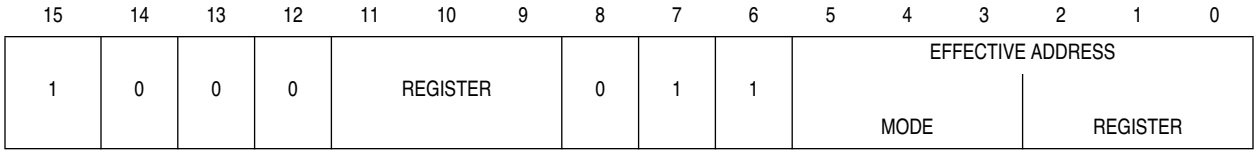
OR



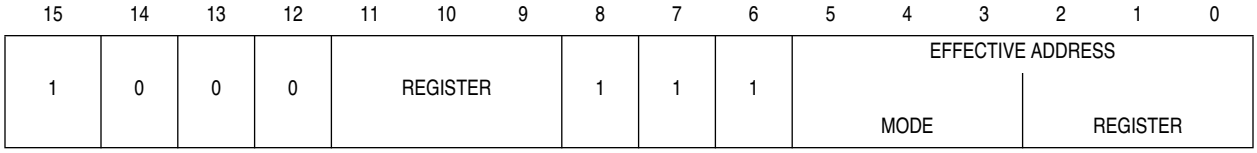
Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩); (⟨Dn⟩) → Dn
100	101	110	(⟨Dn⟩); (⟨ea⟩) → ea

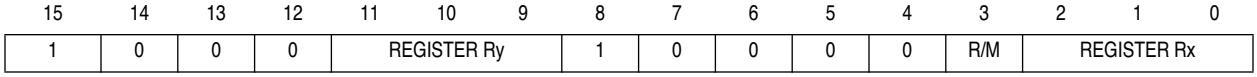
DIVU



DIVS



SBCD

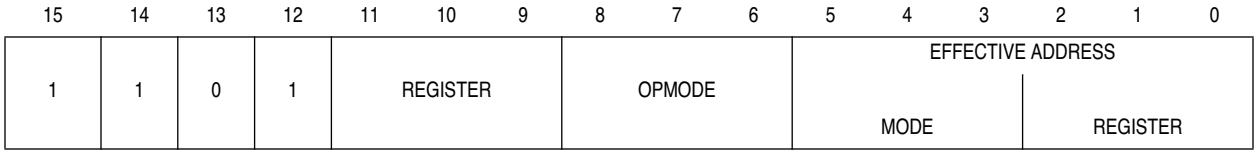


R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, both registers must be data registers

If R/M = 1, both registers must be address registers for Predecrement Addressing mode

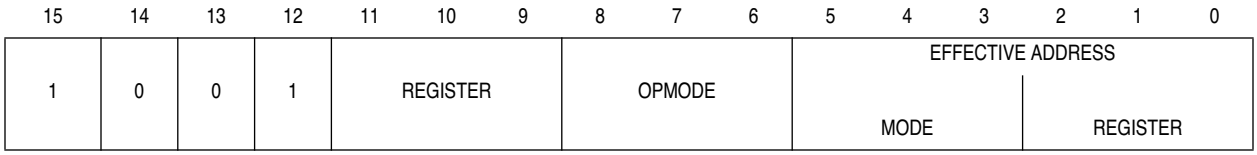
SUB



Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) – (⟨Dn⟩) → ⟨Dn⟩
100	101	110	(⟨Dn⟩) – (⟨ea⟩) → ⟨ea⟩

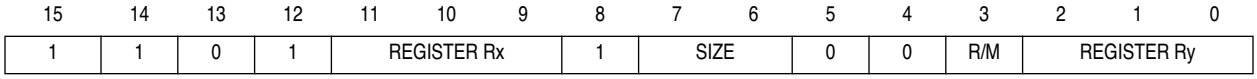
SUBA



Opmode Field:

Word	Long	Operation
011	111	(⟨An⟩) – (⟨ea⟩) → ⟨Dn⟩

SUBX



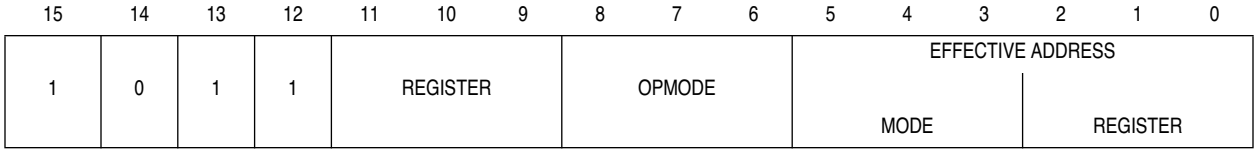
Size Field: 00 = Byte 01 = Word 10 = Long

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, both registers must be data registers

If R/M = 1, both registers must be address registers for Predecrement Addressing mode

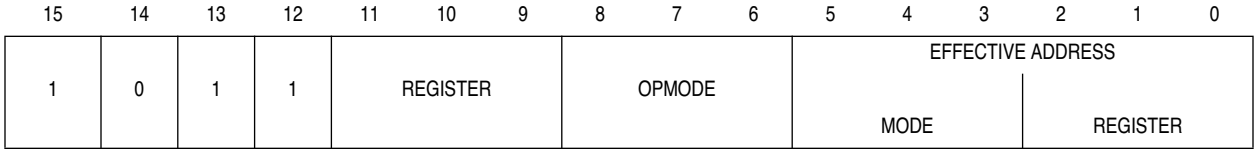
CMP



Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨Dn⟩) – (⟨ea⟩) → CCR

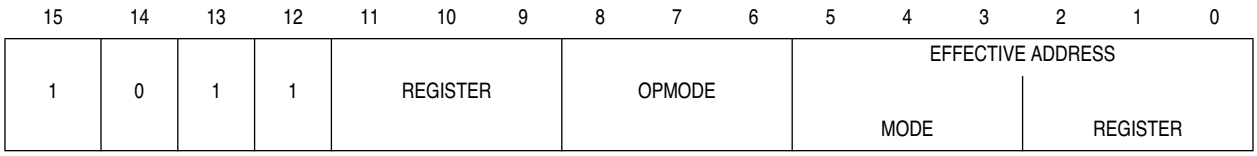
CMPA



Opmode Field:

Word	Long	Operation
011	111	(⟨An⟩) – (⟨ea⟩) → CCR

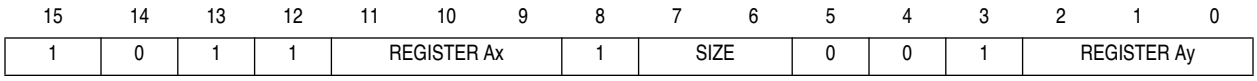
EOR



Opmode Field:

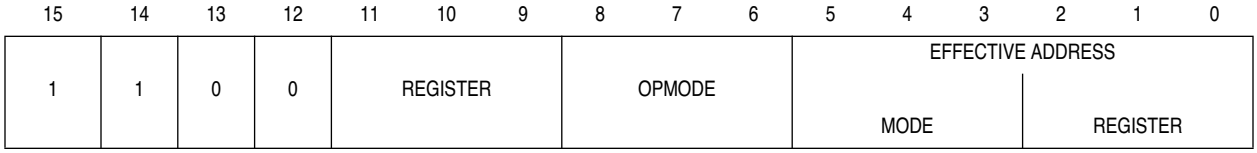
Byte	Word	Long	Operation
100	101	110	(⟨ea⟩) ⊕ (⟨Dn⟩) → ⟨ea⟩

CMPM



Size Field: 00 = Byte 01 = Word 10 = Long

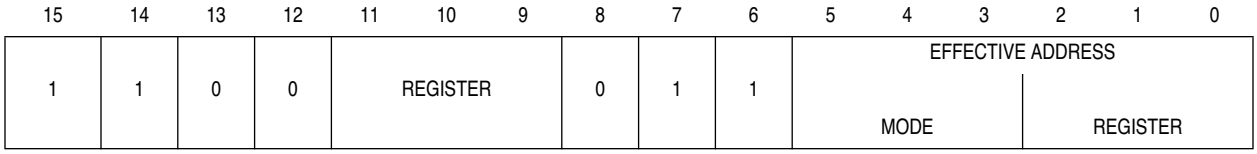
AND



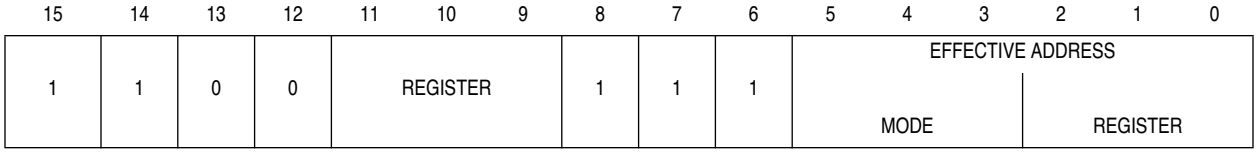
Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) • (⟨Dn⟩) → ⟨Dn⟩
100	101	110	(⟨Dn⟩) • (⟨ea⟩) → ⟨ea⟩

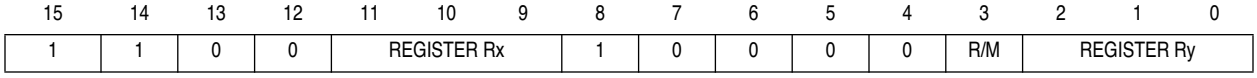
MULU (Word)



MULS (Word)



ABCD

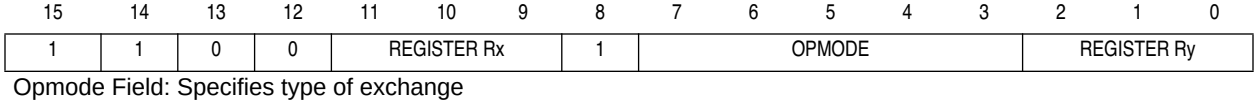


R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, both registers must be data registers

If R/M = 1, both registers must be address registers for Predecrement Addressing mode

EXG



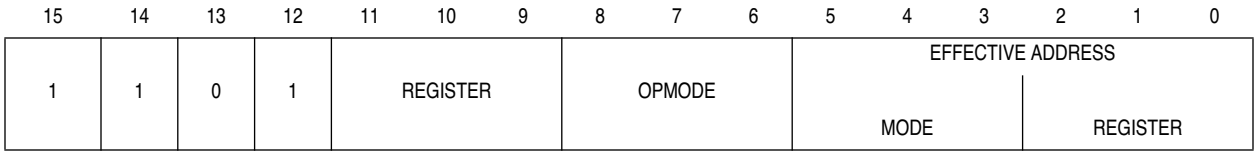
Opmode Field: Specifies type of exchange

01000 — Data Register Exchange

01001 — Address Register Exchange

10001 — Data Register / Address Register (Rx specifies data register, Ry specifies address register)

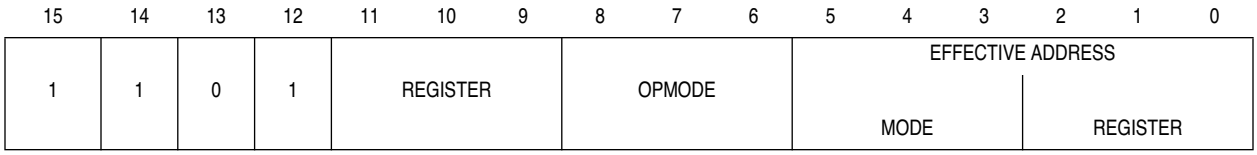
ADD



Opmode Field:

Byte	Word	Long	Operation
000	001	010	(⟨ea⟩) + (⟨Dn⟩) → ⟨Dn⟩
100	101	110	(⟨Dn⟩) + (⟨ea⟩) → ⟨ea⟩

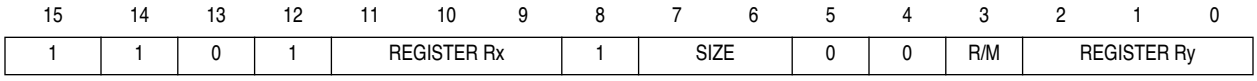
ADDA



Opmode Field:

Word	Long	Operation
011	111	(⟨ea⟩) + (⟨An⟩) → ⟨An⟩

ADDX



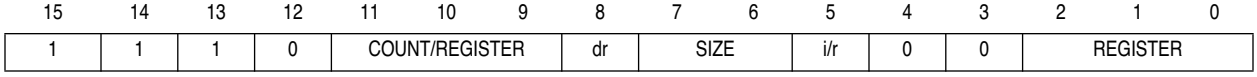
Size Field: 00 = Byte 01 = Word 10 = Long

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, both registers must be data registers

If R/M = 1, both registers must be address registers for Predecrement Addressing mode

ASL, ASR (Register)



Count/Register Field:

If I/R Field = 0, Specifies Shift Count

If I/R Field = 1, Specifies Data Register that contains Shift Count

dr Field: 0 = Right 1 = Left

Size Field: 00 = Byte 01 = Word 10 = Long

I/R Field: 0 = Immediate Shift Count 1 = Register Shift Count

LSL, LSR (Register)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/REGISTER			dr	SIZE		i/r	0	1	REGISTER		

Count/Register Field:

If I/R Field = 0, Specifies Shift Count

If I/R Field = 1, Specifies Data Register that contains Shift Count

dr Field: 0 = Right 1 = Left

Size Field: 00 = Byte 01 = Word 10 = Long

I/R Field: 0 = Immediate Shift Count 1 = Register Shift Count

ROXL, ROXR (Register)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/REGISTER			dr	SIZE		i/r	1	0	REGISTER		

Count/Register Field:

If I/R Field = 0, Specifies Shift Count

If I/R Field = 1, Specifies Data Register that contains Shift Count

dr Field: 0 = Right 1 = Left

Size Field: 00 = Byte 01 = Word 10 = Long

I/R Field: 0 = Immediate Shift Count 1 = Register Shift Count

ROL, ROR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	COUNT/REGISTER			dr	SIZE		i/r	1	1	REGISTER		

Count/Register Field:

If I/R Field = 0, Specifies Shift Count

If I/R Field = 1, Specifies Data Register that contains Shift Count

dr Field: 0 = Right 1 = Left

Size Field: 00 = Byte 01 = Word 10 = Long

I/R Field: 0 = Immediate Shift Count 1 = Register Shift Count

ASL, ASR (Memory)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	0	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

dr Field: 0 = Right 1 = Left

LSL, LSR (Memory)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	0	1	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

dr Field: 0 = Right 1 = Left

ROXL, ROXR (Memory)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

dr Field: 0 = Right 1 = Left

ROL, ROR (Memory)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	dr	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

dr Field: 0 = Right 1 = Left

LPSTOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
IMMEDIATE DATA															

TBLU, TBLUN (Data Register Interpolate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	REGISTER Dym		
	REGISTER Dx			0	R	0	0	SIZE		0	0	0	REGISTER Dyn		

R Field: 0 = Unrounded 1 = Rounded

TBLU, TBLUN (Lookup and Interpolate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dx			0	R	0	1	SIZE		0	0	0	0	0	0

R Field: 0 = Unrounded 1 = Rounded

TBLS, TBLSN (Data Register Interpolate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	0	0	REGISTER Dym		
	REGISTER Dx			1	R	0	0	SIZE		0	0	0	REGISTER Dyn		

R Field: 0 = Unrounded 1 = Rounded

TBLS, TBLSN (Lookup and Interpolate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	REGISTER Dx			1	R	0	1	SIZE		0	0	0	0	0	0
R Field: 0 = Unrounded 1 = Rounded															

4.6 Table Lookup and Interpolation Instructions

There are four table lookup and interpolate instructions. TBLS returns a signed, rounded byte, word, or long-word result. TBLSN returns a signed, unrounded byte, word, or long-word result. TBLU returns an unsigned, rounded byte, word, or long-word result. TBLUN returns an unsigned, unrounded byte, word, or long-word result. All four instructions support two types of interpolation data: an n-element table stored in memory, and a two-element range stored in a pair of data registers. The latter form provides a means of performing surface (3D) interpolation between two previously calculated linear interpolations.

The following examples show how to compress tables and use fewer interpolation levels between table entries. Example 1 (see **Figure 4-3**) demonstrates table lookup and interpolation for a 257-entry table, allowing up to 256 interpolation levels between entries. Example 2 (see **Figure 4-4**) reduces table length for the same data to four entries. Example 3 (see **Figure 4-5**) demonstrates use of an 8-bit independent variable with an instruction.

Two additional examples show how TBLSN can reduce cumulative error when multiple table lookup and interpolation operations are used in a calculation. Example 4 demonstrates addition of the results of three table interpolations. Example 5 illustrates use of TBLSN in surface interpolation.

4.6.1 Table Example 1: Standard Usage

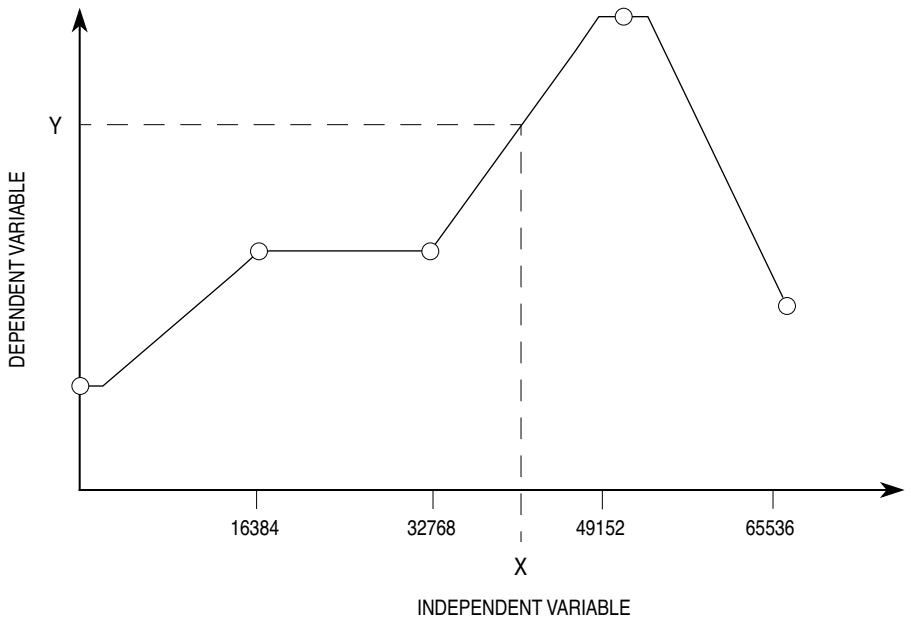


Figure 4-3 Table Example 1

The table consists of 257 word entries. As shown in **Figure 4-3**, the function is linear within the range $32768 \leq X \leq 49152$. Table entries within this range are as follows:

Entry Number	X Value	Y Value
128*	32768	1311
162	41472	1659
163	41728	1669
164	41984	1679
165	42240	1690
192*	49152	1966

*These values are the end points of the range.
All entries between these points fall on the line.

The table instruction is executed with the following bit pattern in Dx:

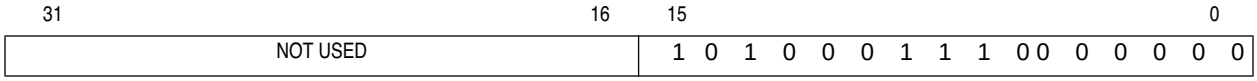


Table Entry Offset → Dx [8:15] = $A3 = 163

Interpolation Fraction → Dx [0:7] = $80 = 128

Using this information, the table instruction calculates dependent variable Y:

Y = 1669 + (128 (1679 – 1669)) / 256 = 1674

4.6.2 Table Example 2: Compressed Table

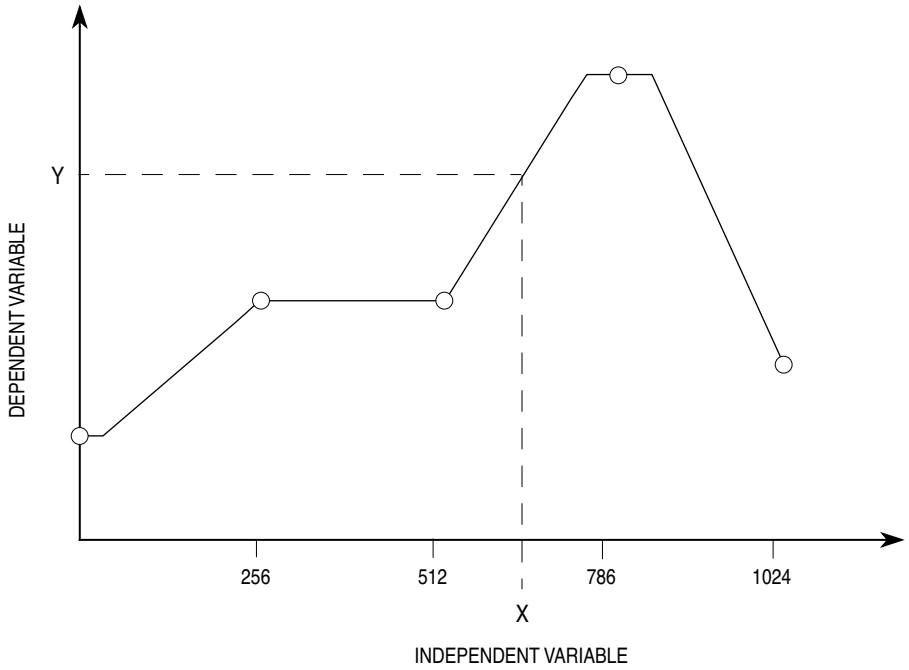


Figure 4-4 Table Example 2

In Example 2, the data from Example 1 has been compressed by limiting the maximum value of the independent variable. Instead of the range $0 \leq X = 65535$, X is limited to $0 \leq X \leq 1023$. The table has been compressed to only 5 entries, but up to 256 levels of interpolation are allowed between entries.

CAUTION

Extreme table compression with many levels of interpolation is possible only with highly linear functions.

The table entries within the range of interest are as follows:

Entry Number	X Value	Y Value
2	512	1311
3	786	1966

Since the table is reduced from 257 to 5 entries, independent variable X must be scaled appropriately. In this case the scaling factor is 64, and the scaling is done by a single instruction:

LSR.W #6,Dx

Thus, Dx now contains the following bit pattern:

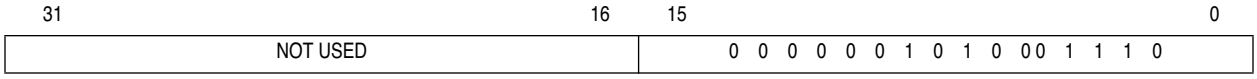


Table Entry Offset $\rightarrow$ Dx [8:15] = $02 = 2

Interpolation Fraction $\rightarrow$ Dx [0:7] = $8E = 142

Using this information, the table instruction calculates dependent variable Y: .

$$Y = 1331 + (142\ (1966 - 1311)) / 256 = 1674$$

The function chosen for Examples 1 and 2 is linear between data points. If another function had been used, interpolated values might not have been identical.

4.6.3 Table Example 3: 8-Bit Independent Variable

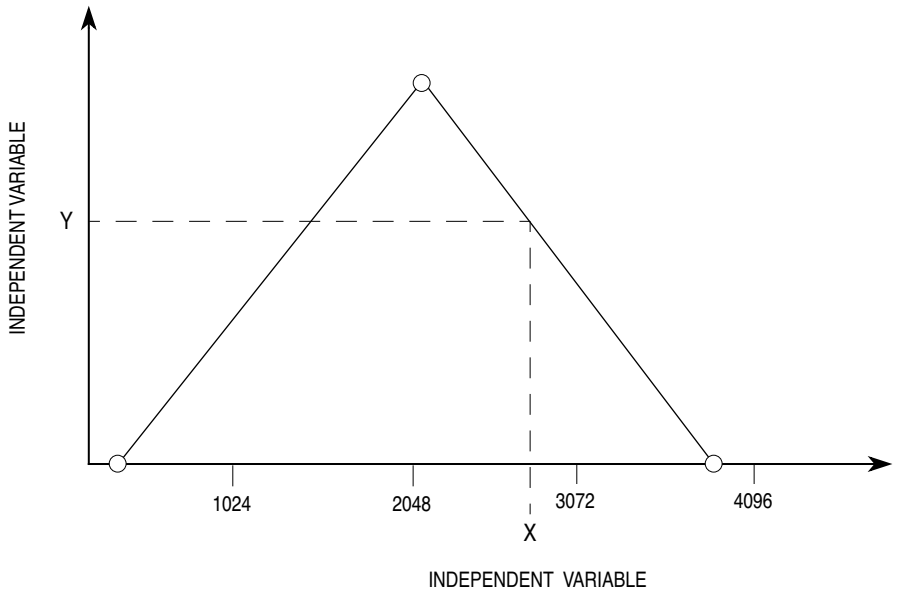


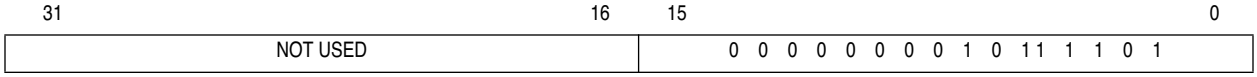
Figure 4-5 Table Example 3

This example shows how to use a table instruction within an interpolation subroutine. Independent variable X is calculated as an 8-bit value, allowing 16 levels of interpolation on a 17-entry table. X is passed to the subroutine, which returns an 8-bit result. The subroutine uses the following data, based on the function shown in **Figure 4-5**.

X (Subroutine)	X (Instruction)	Y
0	0	0
1	256	16
2	512	32
3	768	48
4	1024	64
5	1280	80
6	1536	96
7	1792	112
8	2048	128
9	2304	112
10	2560	96
11	2816	80
12	3072	64
13	3328	48
14	3584	32
15	3840	16
16	4096	0

The first column is the value passed to the subroutine, the second column is the value expected by the table instruction, and the third column is the result returned by the subroutine.

The following value has been calculated for independent variable X:



Since X is an 8-bit value, the upper four bits are used as a table offset and the lower four bits are used as an interpolation fraction. The following results are obtained from the subroutine:

Table Entry Offset → Dx [4:7] = $B = 11

Interpolation Fraction → Dx [0:3] = $D = 13

Thus, Y is calculated as follows:

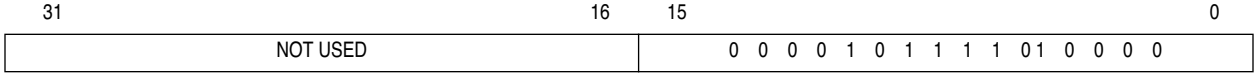
$$Y = 80 + (13\ (64 - 80)) / 16 = 67$$

If the 8-bit value for X were used directly by the table instruction, interpolation would be incorrectly performed between entries 0 and 1. Data must be shifted to the left four places before use:

LSL.W #4, Dx

The new range for X is $0 \leq X \leq 4096$; however, since a left shift fills the least significant digits of the word with zeroes, the interpolation fraction can only have one of 16 values.

After the shift operation, Dx contains the following value:



Execution of the table instruction using the new value in Dx yields:

Table Entry Offset → Dx [8:15] = $0B = 11

Interpolation Fraction → Dx [0:7] = $D0 = 208

Thus, Y is calculated as follows:

$$Y = 80 + (208\ (64 - 80)) / 256 = 67$$

4.6.4 Table Example 4: Maintaining Precision

In this example, three table lookup and interpolation (TLI) operations are performed and the results are summed. The calculation is done once with the result of each TLI rounded before addition and once with only the final result rounded. Assume that the result of the three interpolations are as follows (a "." indicates the binary radix point).

TLI # 1	0010 0000 . 0111 0000
TLI # 2	0011 1111 . 0111 0000
TLI # 3	0000 0001 . 0111 0000

First, the results of each TLI are rounded with the TBLS round-to-nearest-even algorithm. The following values would be returned by TBLS:

TLI # 1	0010 0000 .
TLI # 2	0011 1111 .
TLI # 3	0000 0001 .

Summing, the following result is obtained:

```
0010 0000 .
0011 1111 .
0000 0001 .
0110 0000 .
```

Now, using the same TLI results, the sum is first calculated and then rounded according to the same algorithm:

```
0010 0000 . 0111 0000
0011 1111 . 0111 0000
0000 0001 . 0111 0000
0110 0001 . 0101 0000
```

Rounding yields:

```
0110 0001 .
```

The second result is preferred. The following code sequence illustrates how addition of a series of table interpolations can be performed without loss of precision in the intermediate results:

```
L0:
        TBLSN.B     ⟨ea⟩, Dx
        TBLSN.B     ⟨ea⟩, Dx
        TBLSN.B     ⟨ea⟩, Dl
        ADD.L       Dx, Dm      Long addition avoids problems with carry
        ADD.L       Dm, Dl
        ASR.L#8,    Dl          Move radix point
        BCC.B       L1          Fraction MSB in carry
        ADDQ.B      #1, Dl
L1: . . .
```

4.6.5 Table Example 5: Surface Interpolations

The various forms of table can be used to perform surface (3D) TLIs. However, since the calculation must be split into a series of 2D TLIs, the possibility of losing precision in the intermediate results is possible. The following code sequence, incorporating both TBLS and TBLSN, eliminates this possibility.

```
L0:
        MOVE.W      Dx, Dl          Copy entry number and fraction number
        TBLSN.B     ⟨ea⟩, Dx
        TBLSN.B     ⟨ea⟩, Dl
        TBLS.W      Dx:Dl, Dm       Surface interpolation, with round
        ASR.L       #8, Dm          Read just the result
        BCC.B       L1              No round necessary
        ADDQ.B      #1, Dl          Half round up
L1: . . .
```

Before execution of this code sequence, Dx must contain fraction and entry numbers for the two TLI, and Dm must contain the fraction for surface interpolation. The ⟨ea⟩ fields in the TBLSN instructions point to consecutive columns in a 3D table. The TBLS size parameter must be word if the TBLSN size parameter is byte, and must be long word if TBLSN is word. Increased size is necessary because a larger number of significant digits is needed to accommodate the scaled fractional results of the 2D TLI.

4.7 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 for use. Using this instruction in a series of subroutine calls will generate 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 instruction operand is the address of the link address at the bottom of a stack frame, the effect is to remove the stack frame from both the stack and the linked list.

4.8 Pipeline Synchronization with the NOP Instruction

Although the no operation (NOP) instruction performs no visible operation, it does force synchronization of the instruction pipeline, since all previous instructions must complete execution before the NOP begins.

SECTION 5PROCESSING STATES

This section describes the processing states of the CPU32. It includes a functional description of the bits in the supervisor portion of the status register and an overview of actions taken by the processor in response to exception conditions.

5.1 State Transitions

The processor is in normal, background, or exception state unless halted.

When the processor fetches instructions and operands or executes instructions, it is in the normal processing state. The stopped condition, which the processor enters when a STOP or LPSTOP instruction is executed, is a variation of the normal state in which no further bus cycles are generated.

Background state is an alternate operational mode used for system debugging. Refer to **SECTION 7 DEVELOPMENT SUPPORT** for more information.

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 the stack operations, the exception vector fetch, and the filling of the instruction pipeline caused by an exception. Exception processing ends when execution of an exception handler routine begins. Refer to **SECTION 6 EXCEPTION PROCESSING** for comprehensive information.

A catastrophic system failure occurs if the processor detects a bus error or generates an address error while in the exception processing state. This type of failure halts the processor. For example, if a bus error occurs during exception processing caused by a bus error, the CPU32 assumes that the system is not operational and halts.

The halted condition should not be confused with the stopped condition. After the processor executes a STOP or LPSTOP instruction, execution of instructions can resume when a trace, interrupt, or reset exception occurs.

5.2 Privilege Levels

To protect system resources, the processor can operate with either of two levels of access — user or supervisor. Supervisor level is more privileged than user level. All instructions are available at the supervisor level, but execution of some instructions is not permitted at the user level. There are separate stack pointers for each level. The S bit in the status register indicates privilege level, and determines which stack pointer is used for stack operations. The processor identifies each bus access (supervisor or user mode) via function codes to enforce supervisor and user access levels.

In a typical system most programs execute at the user level. User programs can access only their own code and data areas, and are restricted from accessing other information. The operating system executes at the supervisor privilege level, has access

to all resources, performs the overhead tasks for the user level programs, and coordinates their activities.

5.2.1 Supervisor Privilege Level

If the S bit in the status register is set, supervisor privilege level applies, and all instructions are executable. The bus cycles generated for instructions executed in supervisor level are normally classified as supervisor references, and the values of the function codes on FC[2:0] refer to supervisor address spaces.

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

Instructions that have important system effects can only be executed at supervisor level. For instance, user programs are not permitted to execute STOP, LPSTOP, or RESET instructions. To prevent a user program from gaining privileged access, except in a controlled manner, instructions that can alter the S bit in the status register are privileged. The TRAP #n instruction provides controlled user access to operating system services.

5.2.2 User Privilege Level

If the S bit in the status register is cleared, the processor executes instructions at the user privilege level. 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 FC[2:0] specify user address spaces. While the processor is at the user level, implicit references to the system stack pointer and explicit references to address register seven (A7) refer to the user stack pointer (USP).

5.2.3 Changing Privilege Level

To change from user privilege level to supervisor privilege level, a condition that causes exception processing must occur. When exception processing begins, the current values in the status register, including the S bit, are saved on the supervisor stack, and then the S bit is set, enabling supervisory access. Execution continues at supervisor level until exception processing is complete.

To return to user access 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. These instructions execute only at supervisor privilege level, and can modify the S bit of the status register. After these instructions execute, the instruction pipeline is flushed, then refilled from the appropriate address space.

The RTE instruction causes a return to a program that was executing when an exception occurred. When RTE is executed, the exception stack frame saved on the supervisor stack can be restored in either of two ways.

If the frame was generated by an interrupt, breakpoint, trap, or instruction exception, the status register and program counter are restored to the values saved on the supervisor stack, and execution resumes at the restored program counter address, with access level determined by the S bit of the restored status register.

If the frame was generated by a bus error or an address error exception, the entire processor state is restored from the stack.

5.3 Types of Address Space

During each bus cycle, the processor generates function code signals that permit selection of eight distinct 4-Gigabyte address spaces. Not all devices that incorporate the CPU32 support a full complement of memory. (Refer to the appropriate user's manual for details.) Selection varies according to the access required. Automatic selection of supervisor and user space, and of program and data space, is provided. In addition, certain special processor cycles, such as the interrupt acknowledge cycle or the LPSTOP broadcast cycle are recognized, and appropriate codes are generated. **Table 5-1** shows function code values and the corresponding address space.

Table 5-1 Address Spaces

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.

Although an appropriate address space is selected, memory locations of user program and data, and of supervisor data, within that address space are not predefined. During reset, two long words beginning at memory location zero in the supervisor program space are used for processor initialization. No other memory locations are explicitly defined by the CPU32.

5.3.1 CPU Space Access

Function code $7 ([FC2:FC0] = 111) selects CPU address space. The processor communicates with external devices for special purposes by accessing this space. All M68000 processors use CPU space for interrupt acknowledge cycles. The CPU32 also uses CPU space for breakpoint acknowledge and the LPSTOP broadcast.

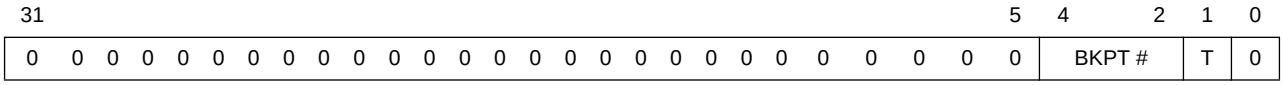
Supervisor programs can use the MOVES instruction to access all address spaces, including user spaces and CPU address space. Although the MOVES instruction can be used to generate CPU space cycles, doing so may interfere with proper system operation. Exercise caution when using MOVES to access CPU space.

Address bus encoding facilitates CPU space transactions. Bits A[19:16], the CPU space type field, show which transaction is being performed. Currently, only five of the 16 possible encodings are defined: 0000, 0001, 0010, 0011, and 1111. Of these, only 0000, 0011, and 1111 are supported by the CPU32.

Address bits A[31:20] are not present on all M68000 processors, and thus cannot be essential to CPU space transaction decoding. The function of other address bus bit fields depends on the transaction being performed. A description of each defined CPU space types follows.

5.3.1.1 Type 0000 — Breakpoint

This CPU space type is used for breakpoint acknowledge.



BKPT# field A[4:2] indicates the breakpoint number. Software breakpoints set this value to the number of the executing breakpoint instruction. Hardware breakpoints always set BKPT# to 7 (%111).

T bit A1 designates the type of breakpoint. T = 0 indicates a software breakpoint; T = 1 indicates a hardware breakpoint.

5.3.1.2 Type 0001 — MMU Access

This type of access is not supported by the CPU32 processor. This space is reserved for future use.

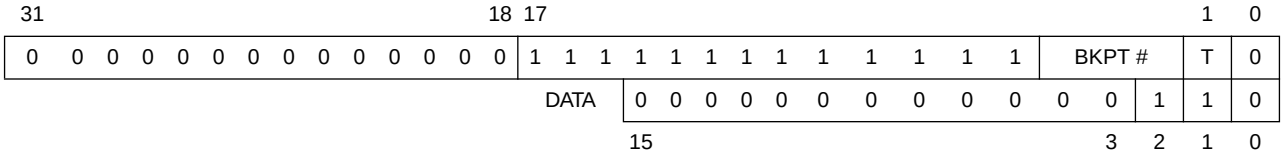
5.3.1.3 Type 0010 — Coprocessor Access

This type of access is not supported by the CPU32 processor. This space is reserved for future use.

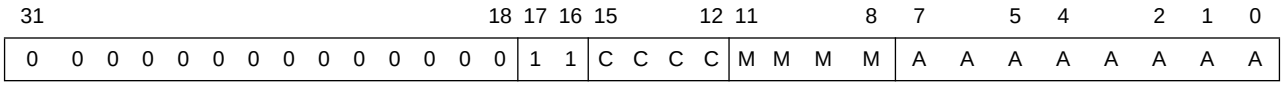
5.3.1.4 Type 0011 — Internal Register Access

Type 0011 space is used to access certain critical system configuration or control registers.

The CPU32 external bus interface interrupt mask register resides in CPU space. This register is written to when LPSTOP is executed, and masks off external interrupts while in stop mode. A[3:1] indicate the encoded interrupt mask level.



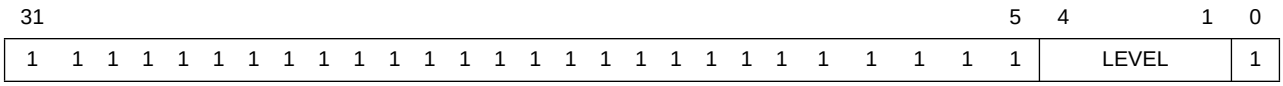
These control registers, reserved for future expansion, also reside in CPU space 3, and are only accessible through the MOVES command. The general format of this CPU space type is defined as follows:



A[15:12] are used as 1 of 16 external chip selects.
A[11: 8] are used as 1 of 16 internal module selects.
A[7:0] are used as 1 of 256 module register addresses.

5.3.1.5 Type 1111 — Interrupt Acknowledge

Interrupt acknowledge is a CPU space type used for interrupt acknowledge. A[4:1] indicate the encoded interrupt level being acknowledged.



SECTION 6 EXCEPTION PROCESSING

This section discusses system resources related to exception handling, exception processing sequence, and specific features of individual exception processing routines

6.1 Definition of Exception Processing

An exception is a special condition that preempts normal processing. Exception processing is the transition from normal mode program execution to execution of a routine that deals with an exception.

6.1.1 Exception Vectors

An exception vector is the address of a routine that handles an exception. The vector base register (VBR) contains the base address of a 1024-byte exception vector table, which consists of 256 exception vectors. Sixty-four vectors are defined by the processor, and 192 vectors are reserved for user definition as interrupt vectors. Except for the reset vector, each vector in the table is one long word in length. The reset vector is two long words in length. Refer to **Table 6-1** for information on vector assignment.

CAUTION

Because there is no protection on the 64 processor-defined vectors, external devices can access vectors reserved for internal purposes — this practice is strongly discouraged.

All exception vectors, except the reset vector, are located in supervisor data space. The reset vector is located in supervisor program space. Only the initial reset vector is fixed in the processor memory map. When initialization is complete, there are no fixed assignments. Since the VBR stores the vector table base address, the table can be located anywhere in memory. It can also be dynamically relocated for each task executed by an operating system.

Table 6-1 Exception Vector Assignments

Vector Number	Vector Offset			Assignment
	Dec	Hex	Space	
0	0	000	SP	Reset: Initial Stack Pointer
1	4	004	SP	Reset: Initial Program Counter
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Division
6	24	018	SD	CHK, CHK2 Instructions
7	28	01C	SD	TRAPcc, TRAPV Instructions
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12	48	030	SD	Hardware Breakpoint
13	52	034	SD	(Reserved, Coprocessor Protocol Violation)
14	56	038	SD	Format Error and Uninitialized Interrupt
15	60	03C	SD	Format Error and Uninitialized Interrupt
16–23	64 92	040 05C	SD	(Unassigned, Reserved)
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt Autovector
26	104	068	SD	Level 2 Interrupt Autovector
27	108	06C	SD	Level 3 Interrupt Autovector
28	112	070	SD	Level 4 Interrupt Autovector
29	116	074	SD	Level 5 Interrupt Autovector
30	120	078	SD	Level 6 Interrupt Autovector
31	124	07C	SD	Level 7 Interrupt Autovector
32–47	128 188	080 0BC	SD	Trap Instruction Vectors (0–15)
48–58	192 232	0C0 0E8	SD	(Reserved, Coprocessor)
59–63	236 252	0EC 0FC	SD	(Unassigned, Reserved)
64–255	256 1020	100 3FC	SD	User Defined Vectors (192)

Each vector is assigned an 8-bit number. Vector numbers for some exceptions are obtained from an external device; others are supplied by the processor. The processor multiplies the vector number by four to calculate vector offset, then adds the offset to the contents of the VBR. The sum is the memory address of the vector.

6.1.2 Types of Exceptions

An exception can be caused by internal or external events.

An internal exception can be generated by an instruction or by an error. The TRAP, TRAPcc, TRAPV, BKPT, CHK, CHK2, RTE, and DIV instructions can cause exceptions during normal execution. Illegal instructions, instruction fetches from odd addresses, word or long-word operand accesses from odd addresses, and privilege violations also cause internal exceptions.

Sources of external exception include interrupts, breakpoints, bus errors, and reset requests. Interrupts are peripheral device requests for processor action. Breakpoints are used to support development equipment. Bus error and reset are used for access control and processor restart.

6.1.3 Exception Processing Sequence

For all exceptions other than a reset exception, exception processing occurs in the following sequence. Refer to **6.2.1 Reset** for details of reset processing.

As exception processing begins, the processor makes an internal copy of the status register. After the copy is made, the processor state bits in the status register are changed — the S bit is set, establishing supervisor access level, and bits T1 and T0 are cleared, disabling tracing. For reset and interrupt exceptions, the interrupt priority mask is also updated.

Next, the exception number is obtained. For interrupts, the number is fetched ROM CPU space $F (the bus cycle is an interrupt acknowledge). For all other exceptions, internal logic provides a vector number.

Next, current processor status is saved. An exception stack frame is created and placed on the supervisor stack. All stack frames contain copies of the status register and the program counter for use by RTE. The type of exception and the context in which the exception occurs determine what other information is stored in the stack frame.

Finally, the processor prepares to resume normal execution of instructions. The exception vector offset is determined by multiplying the vector number by four, and the offset is added to the contents of the VBR to determine displacement into the exception vector table. The exception vector is loaded into the program counter. If no other exception is pending, the processor will resume normal execution at the new address in the PC.

6.1.4 Exception Stack Frame

During exception processing, the most volatile portion of the current context is saved on the top of the supervisor stack. This context is organized in a format called the exception stack frame.

The exception stack frame always includes the contents of status register and program counter at the time the exception occurred. To support generic handlers, the processor also places the vector offset in the exception stack frame and marks the frame with a format code. The format field allows an RTE instruction to identify stack information so that it can be properly restored.

The general form of the exception stack frame is illustrated in Figure 6-1. Although some formats are peculiar to a particular M68000 Family processor, format 0000 is always legal and always indicates that only the first four words of a frame are present. See **6.4 CPU32 Stack Frames** for a complete discussion of exception stack frames.

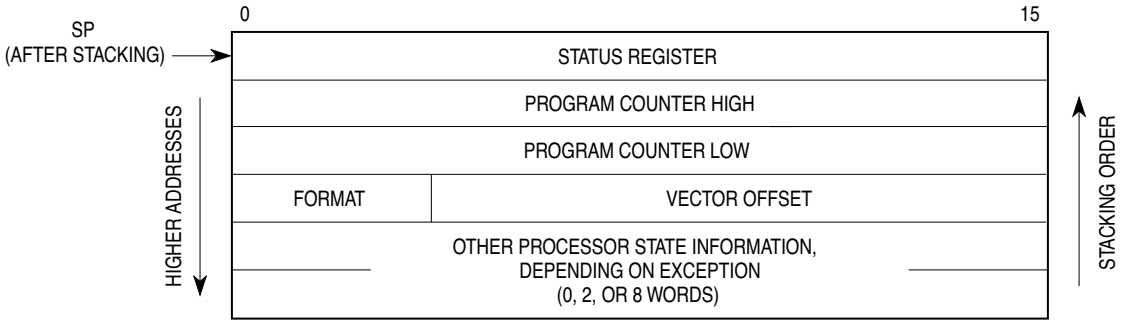


Figure 6-1 Exception Stack Frame

6.1.5 Multiple Exceptions

Each exception has been assigned a priority based on its relative importance to system operation. Priority assignments are shown in **Table 6-2**. Group 0 exceptions have the highest priorities. Group 4 exceptions have the lowest priorities. Exception processing for exceptions that occur simultaneously is done by priority, from highest to lowest.

Table 6-2 Exception Priority Groups

Group/ Priority	Exception and Relative Priority	Characteristics
0	Reset	Aborts all processing (instruction or exception); does not save old context
1.1 1.2	Address Error Bus Error	Suspends processing (instruction or exception); saves internal context
2	BKPT#n, CHK, CHK2, Division by Zero, RTE, TRAP#n, TRAPcc, TRAPV	Exception processing is a part of instruction execution
3	Illegal Instruction, Line A, Unimplemented Line F, Privilege Violation	Exception processing begins before instruction execution
4.1 4.2 4.3	Trace Hardware Breakpoint Interrupt	Exception processing begins when current instruction or previous exception processing is complete

It is important to be aware of the difference between exception processing mode and execution of an exception handler. Each exception has an assigned vector that points to an associated handler routine. Exception processing includes steps described in **6.1.3 Exception Processing Sequence**, but does not include execution of handler routines, which is done in normal mode.

When the CPU32 completes exception processing, it is ready to begin either exception processing for a pending exception, or execution of a handler routine. Priority assignment governs the order in which exception processing occurs, not the order in which exception handlers are executed.

As a general rule, when simultaneous exceptions occur, the handler routines for lower priority exceptions are executed before the handler routines for higher priority exceptions. For example, consider the arrival of an interrupt during execution of a TRAP instruction, while tracing is enabled. Trap exception processing (2) is done first, followed immediately by exception processing for the trace (4.1), and then by exception processing for the interrupt (4.3). Each exception places a new context on the stack. When the processor resumes normal instruction execution, it is vectored to the interrupt handler, which returns to the trace handler that returns to the trap handler.

There are special cases to which the general rule does not apply. The reset exception will always be the first exception handled, since reset clears all other exceptions. It is also possible for high priority exception processing to begin before low priority exception processing is complete. For example, if a bus error occurs during trace exception processing, the bus error will be processed and handled before trace exception processing is completed.

6.2 Processing of Specific Exceptions

The following paragraphs provide details concerning sources of specific exceptions, how each arises, and how each is processed.

6.2.1 Reset

Assertion of RESET by external hardware, or assertion of the internal RESET signal by an internal module, causes a reset exception. The reset exception has the highest priority of any exception. Reset is used for system initialization and for recovery from catastrophic failure. The reset exception aborts any processing in progress when it is recognized, and that processing cannot be recovered. Reset performs the following operations:

1. Clears T0 and T1 in the status register to disable tracing
2. Sets the S bit in the status register to establish supervisor privilege
3. Sets the interrupt priority mask to the highest priority level (%111)
4. Initializes the vector base register to zero ($00000000)
5. Generates a vector number to reference the reset exception vector
6. Loads the first long word of the vector into the interrupt stack pointer
7. Loads the second long word of the vector into the program counter
8. Fetches and initiates decode of the first instruction to be executed

Figure 6-2 is a flowchart of the reset exception.

After initial instruction prefetches, normal 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.

If a bus error or address error occurs during reset exception processing sequence, a double bus fault occurs. The processor halts, and the HALT signal is asserted to indicate the halted condition.

Execution of the RESET instruction does not cause a reset exception nor does it affect any internal CPU register, but it does cause the CPU32 to assert the RESET signal, resetting all internal and external peripherals.

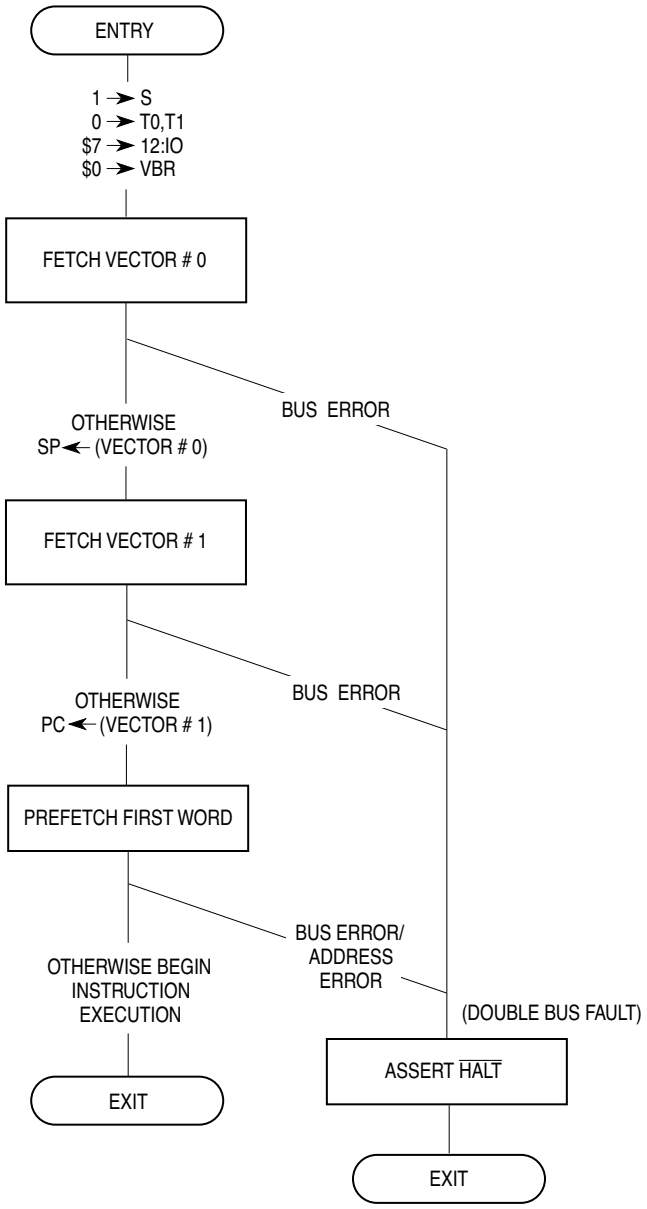


Figure 6-2 Reset Operation Flowchart

6.2.2 Bus Error

A bus error exception occurs when an assertion of the $\overline{\text{BERR}}$ signal is acknowledged. The $\overline{\text{BERR}}$ signal can be asserted by one of three sources:

1. External logic by assertion of the $\overline{\text{BERR}}$ input pin
2. Direct assertion of the internal $\overline{\text{BERR}}$ signal by an internal module
3. Direct assertion of the internal $\overline{\text{BERR}}$ signal by the on-chip hardware watchdog after detecting a no-response condition

Bus error exception processing begins when the processor attempts to use information from an aborted bus cycle.

When the aborted bus cycle is an instruction prefetch, the processor will not initiate exception processing unless the prefetched information is used. For example, if a branch instruction flushes an aborted prefetch, that word is not accessed, and no exception occurs.

When the aborted bus cycle is a data access, the processor initiates exception processing immediately, except in the case of released operand writes. Released write bus errors are delayed until the next instruction boundary or until another operand access is attempted.

Exception processing for bus error exceptions follows the regular sequence, but context preservation is more involved than for other exceptions because a bus exception can be initiated while an instruction is executing. Several bus error stack format organizations are utilized to provide additional information regarding the nature of the fault.

First, any register altered by a faulted-instruction effective address calculation is restored to its initial value. Then a special status word (SSW) is placed on the stack. The SSW contains specific Information about the aborted access — size, type of access (read or write), bus cycle type, and function code are saved. Finally, fault address, bus error exception vector number, program counter value, and a copy of the status register are saved.

If a bus error occurs during exception processing for a bus error, an address error, a reset, or while the processor is loading stack information during RTE execution, the processor halts. This simplifies isolation of catastrophic system failure by preventing processor interaction with stacks and memory. Only assertion of RESET can restart a halted processor.

6.2.3 Address Error

Address error exceptions occur when the processor attempts to access an instruction, word operand, or long-word operand at an odd address. The effect is much the same as an internally generated bus error. The exception processing sequence is the same as that for bus error, except that the vector number refers to the address error exception vector.

Address error exception processing begins when the processor attempts to use information from the aborted bus cycle.

If the aborted cycle is a data space access, exception processing begins when the processor attempts to use the data, except in the case of a released operand write. Released write exceptions are delayed until the next instruction boundary or attempted operand access.

An address exception on a branch to an odd address is delayed until the program counter is changed. No exception occurs if the branch is not taken. In this case, the fault address and return program counter value placed in the exception stack frame are the odd address, and the current instruction program counter points to the instruction that caused the exception.

If an address error occurs during exception processing for a bus error, another address error, or a reset, the processor halts.

6.2.4 Instruction Traps

Traps are exceptions caused by instructions. They arise from either processor recognition of abnormal conditions during instruction execution or from use of specific trapping instructions. Traps are generally used to handle abnormal conditions that arise in control routines.

The TRAP instruction, which always forces an exception, is useful for implementing system calls for user programs. The TRAPcc, TRAPV, CHK, and CHK2 instructions force exceptions when a program detects a run-time error. The DIVS and DIVU instructions force an exception if a division operation is attempted with a divisor of zero.

Exception processing for traps follows the regular sequence. If tracing is enabled when an instruction that causes a trap begins execution, a trace exception will be generated by the instruction, but the trap handler routine will not be traced (the trap exception will be processed first, then the trace exception).

The vector number for the TRAP instruction is internally generated — part of the number comes from the instruction itself. The trap vector number, program counter value, and a copy of the status register are saved on the supervisor stack. The saved program counter value is the address of the instruction that follows the instruction which generated the trap. For all instruction traps other than TRAP, a pointer to the instruction causing the trap is also saved in the fifth and sixth words of the exception stack frame.

6.2.5 Software Breakpoints

To support hardware emulation, the CPU32 must provide a means of inserting breakpoints into target code and of announcing when a breakpoint is reached.

The MC68000 and MC68008 can detect an illegal instruction inserted at a breakpoint when the processor fetches from the illegal instruction exception vector location. Since the VBR on the CPU32 allows relocation of exception vectors, the exception vector address is not a reliable indication of a breakpoint. CPU32 breakpoint support is provided by extending the function of a set of illegal instructions ($4848–$484F).

When a breakpoint instruction is executed, the CPU32 performs a read from CPU space $0, at a location corresponding to the breakpoint number (See **5.3 Types of Address Space**). If this bus cycle is terminated by $\overline{\text{BERR}}$, the processor performs illegal instruction exception processing. If the bus cycle is terminated by $\overline{\text{DSACK}}$, the processor uses the data returned to replace the breakpoint in the instruction pipeline and begins execution of that instruction.

6.2.6 Hardware Breakpoints

The CPU32 recognizes hardware breakpoint requests. Hardware breakpoint requests do not force immediate exception processing, but are left pending. An instruction

breakpoint is not made pending until the instruction corresponding to the request is executed.

A pending breakpoint can be acknowledged between instructions or at the end of exception processing. To acknowledge a breakpoint, the CPU performs a read from CPU space $0 at location $1E. See **5.3 Types of Address Space** for a detailed description of CPU space operations.

If the bus cycle terminates normally, instruction execution continues with the next instruction, as if no breakpoint request occurred. If the bus cycle is terminated by $\overline{\text{BERR}}$, the CPU begins exception processing. Data returned during this bus cycle is ignored.

Exception processing follows the regular sequence. Vector number 12 (offset $30) is internally generated. The program counter of the currently executing instruction, the program counter of the next instruction to execute, and a copy of the status register are saved on the supervisor stack.

6.2.7 Format Error

The processor checks certain data values for control operations. The validity of the stack format code and, in the case of a bus cycle fault format, the version number of the processor that generated the frame are checked during execution of the RTE instruction. This check ensures that the program does not make erroneous assumptions about information in the stack frame.

If the format of the control data is improper, the processor generates a format error exception. This exception saves a four-word format exception frame and then vectors through vector table entry number 14. The stacked program counter is the address of the RTE instruction that discovered the format error.

6.2.8 Illegal or Unimplemented Instructions

An instruction is illegal if it contains a word bit pattern that does not correspond to the bit pattern of the first word of a legal CPU32 instruction, if it is a MOVEC instruction that contains an undefined register specification field in the first extension word, or if it contains an indexed addressing mode extension word with bits [5:4] = 00 or bits [3:0] $\neq$ 0000.

If an illegal instruction is fetched during instruction execution, an illegal instruction exception occurs. This facility allows the operating system to detect program errors or to emulate instructions in software.

Word patterns with bits [15:12] = 1010 (referred to as A-line opcodes) are unimplemented instructions. A separate exception vector (vector 10, offset $28) is given to unimplemented instructions to permit efficient emulation.

Word patterns with bits [15:12] = 1111 (referred to as F-line opcodes) are used for M68000 Family instruction set extensions. They can generate an unimplemented instruction exception caused by the first extension word of the instruction or by the addressing mode extension word. A separate F-line emulation vector (vector 11, offset $2C) is used for the exception vector.

All unimplemented instructions are reserved for use by Motorola for enhancements and extensions to the basic M68000 architecture. Opcode pattern $4AFC is defined to be illegal on all M68000 Family members. Those customers requiring the use of an unimplemented opcode for synthesis of “custom instructions,” operating system calls, etc., should use this opcode.

Exception processing for illegal and unimplemented instructions is similar to that for traps. The instruction is fetched and decoding is attempted. When the processor determines that execution of an illegal instruction is being attempted, exception processing begins. No registers are altered.

Exception processing follows the regular sequence. The vector number is generated to refer to the illegal instruction vector or, in the case of an unimplemented instruction, to the corresponding emulation vector. The illegal instruction vector number, current program counter, and a 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.

6.2.9 Privilege Violations

To provide system security, certain instructions can be executed only at the supervisor access level An attempt to execute one of these instructions at the user level will cause an exception. The privileged exceptions are as follows:

- AND Immediate to SR
- EOR Immediate to SR
- LPSTOP
- MOVE from SR
- MOVE to SR
- MOVE USP
- MOVEC
- MOVES
- OR Immediate to SR
- RESET
- RTE
- STOP

Exception processing for privilege violations is nearly identical to that for illegal instructions. The instruction is fetched and decoded. If the processor determines that a privilege violation has occurred, exception processing begins before instruction execution.

Exception processing follows the regular sequence. The vector number (8) is generated to reference the privilege violation vector. Privilege violation vector offset, current program counter, and status register are saved on the supervisor stack. The saved program counter value is the address of the first word of the instruction causing the privilege violation.

6.2.10 Tracing

To aid in program development, M68000 processors include a facility to allow tracing of instruction execution. CPU32 tracing also has the ability to trap on changes in program flow. In trace mode, a trace exception is generated after each instruction executes, allowing a debugging program to monitor the execution of a program under test. The T1 and T0 bits in the supervisor portion of the status register are used to control tracing.

When T[1:0] = 00, tracing is disabled, and instruction execution proceeds normally (see **Table 6-3**).

Table 6-3 Tracing Control

T1	T0	Tracing Function
0	0	No tracing
0	1	Trace on change of flow
1	0	Trace on instruction execution
1	1	(Undefined; reserved)

When T[1:0] = 01 at the beginning of instruction execution, a trace exception will be generated if the program counter changes sequence during execution. All branches, jumps, subroutine calls, returns, and status register manipulations can be traced in this way. No exception occurs if a branch is not taken.

When T[1:0] = 10 at the beginning of instruction execution, a trace exception will be generated when execution is complete. If the instruction is not executed, either because an interrupt is taken or because the instruction is illegal, unimplemented, or privileged, an exception is not generated.

At the present time, T[1:0] = 11 is an undefined condition. It is reserved by Motorola for future use.

Exception processing for trace starts at the end of normal processing for the traced instruction and before the start of the next instruction. Exception processing follows the regular sequence (tracing is disabled so that the trace exception itself is not traced). A vector number is generated to reference the trace exception vector. The address of the instruction that caused the trace exception, the trace exception vector offset, the current program counter, and a copy of the status register are saved on the supervisor stack. The saved value of the program counter is the address of the next instruction to be executed.

A trace exception can be viewed as an extension to the function of any instruction. If a trace exception is generated by an instruction, the execution of that instruction is not complete until the trace exception processing associated with it is also complete:

If an instruction is aborted by a bus error or address error exception, trace exception processing is deferred until the suspended instruction is restarted and completed normally. An RTE from a bus error or address error will not be traced because of the possibility of continuing the instruction from the fault.

If an instruction is executed and an interrupt is pending on completion, the trace exception is processed before the interrupt exception.

If an instruction forces an exception, the forced exception is processed before the trace exception.

If an instruction is executed and a breakpoint is pending upon completion of the instruction, the trace exception is processed before the breakpoint.

If an attempt is made to execute an illegal, unimplemented, or privileged instruction while tracing is enabled, no trace exception will occur because the instruction is not executed. This is particularly important to an emulation routine that performs an instruction function, adjusts the stacked program counter to beyond the unimplemented instruction, and then returns. The status register on the stack must be checked to determine if tracing is on before the return is executed. If tracing is on, trace exception processing must be emulated so that the trace exception handler can account for the emulated instruction.

Tracing also affects normal operation of the STOP and LPSTOP instructions. If either begins execution with T1 set, a trace exception will be taken after the instruction loads the status register. Upon return from the trace handler routine, execution will continue with the instruction following STOP (LPSTOP), and the processor will not enter the stopped condition.

6.2.11 Interrupts

There are seven levels of interrupt priority and 192 assignable interrupt vectors within each exception vector table. Judicious use of multiple vector tables and hardware chaining will permit a virtually unlimited number of peripherals to interrupt the processor.

Interrupt recognition and subsequent processing are based on internal interrupt request signals ($\overline{IRQ7}$–$\overline{IRQ1}$) and the current priority set in status register priority mask I[2:0]. Interrupt request level zero ($\overline{IRQ7}$–$\overline{IRQ1}$ negated) indicates that no service is requested. When an interrupt of level one through six is requested via $\overline{IRQ6}$–$\overline{IRQ1}$, the processor compares the request level with the interrupt mask to determine whether the interrupt should be processed. Interrupt requests are inhibited for all priority levels less than or equal to the current priority. Level seven interrupts are nonmaskable.

$\overline{IRQ7}$–$\overline{IRQ1}$ are synchronized and debounced by input circuitry on consecutive rising edges of the processor clock. To be valid, an interrupt request must be held constant for at least two consecutive clock periods.

Interrupt requests do not force immediate exception processing, but are left pending. A pending interrupt is detected between instructions or at the end of exception processing — all interrupt requests must be held asserted until they are acknowledged by the CPU. If the priority of the interrupt is greater than the current priority level, exception processing begins.

Exception processing occurs as follows. First, the processor makes an internal copy of the status register. After the copy is made, the processor state bits in the status register are changed — the S bit is set, establishing supervisor access level, and bits T1 and T0 are cleared, disabling tracing. Then, priority level is set to the level of the interrupt and the processor fetches a vector number from the interrupting device (CPU space $F). The fetch bus cycle is classified as an interrupt acknowledge and the encoded level number of the interrupt is placed on the address bus.

If an interrupting device requests automatic vectoring, the processor generates a vector number (25 to 31) determined by the interrupt level number.

If the response to the interrupt acknowledge bus cycle is a bus error, the interrupt is taken to be spurious, and the spurious interrupt vector number (24) is generated.

The exception vector number, program counter, and status register are saved on the supervisor stack. The saved value of the program counter is the address of the instruction that would have executed if the interrupt had not occurred.

Priority level seven interrupt is a special case. Level seven interrupts are nonmaskable interrupts (NMI). Level seven requests are transition sensitive to eliminate redundant servicing and concomitant stack overflow. Transition sensitive means that the level seven input must change state before the CPU will detect an interrupt.

An NMI is generated each time the interrupt request level changes to level seven (regardless of priority mask value), and each time the priority mask changes from seven to a lower number while request level remains at seven.

Many M68000 peripherals provide for programmable interrupt vector numbers to be used in the system interrupt request/acknowledge mechanism. If the vector number is not initialized after reset and if the peripheral must acknowledge an interrupt request, the peripheral should return the uninitialized interrupt vector number (15).

See the system integration user's manual for detailed information on interrupt acknowledge cycles.

6.2.12 Return from Exception

When exception stacking operations for all pending exceptions are complete, the processor begins execution of the handler for the last exception processed. After the exception handler has executed, the processor must restore the system context in existence prior to the exception. The RTE instruction is designed to accomplish this task.

When RTE is executed, the processor examines the stack frame on top of the supervisor stack to determine if it is valid and determines what type of context restoration must be performed. See **6.4 CPU32 Stack Frames** for a description of stack frames.

For a normal four-word frame, the processor updates the status register and program counter with data pulled from the stack, increments the supervisor stack pointer by eight, and resumes normal instruction execution. For a six-word frame, the status reg-

ister and program counter are updated from the stack, the active supervisor stack pointer is incremented by 12, and normal instruction execution resumes.

For a bus fault frame, the format value on the stack is first checked for validity. In addition, the version number on the stack must match the version number of the processor that is attempting to read the stack frame. The version number is located in the most significant byte (bits [15:8]) of the internal register word at location SP + $14 in the stack frame. The validity check insures that stack frame data will be properly interpreted in multiprocessor systems.

If a frame is invalid, a format error exception is taken. If it is inaccessible, a bus error exception is taken. Otherwise, the processor reads the entire frame into the proper internal registers, de-allocates the stack (12 words), and resumes normal processing. Bus error frames for faults during exception processing require the RTE instruction to rewrite the faulted stack frame. If an error occurs during any of the bus cycles required by rewrite, the processor halts.

If a format error occurs during RTE execution, the processor creates a normal four-word fault stack frame below the frame that it was attempting to use. If a bus error occurs, a bus-error stack frame will be created. The faulty stack frame remains intact, so that it may be examined and repaired by an exception handler, or used by a different type of processor (e.g., an MC68010, MC68020, or a future M68000 processor) in a multiprocessor system.

6.3 Fault Recovery

There are four phases of recovery from a fault: recognizing the fault, saving the processor state, repairing the fault (if possible), and restoring the processor state. Saving and restoring the processor state are described in the following paragraphs.

The stack contents are identified by the special status word (SSW). In addition to identifying the fault type represented by the stack frame, the SSW contains the internal processor state corresponding to the fault.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TP	MV	0	TR	B1	B0	RR	RM	IN	RW	LG	SIZ		FUNC		

TP	BERR frame type
MV	MOVEM in progress
TR	Trace pending
B1	Breakpoint channel 1 pending
B0	Breakpoint channel 0 pending
RR	Rerun write cycle after RTE
RM	Faulted cycle was read-modify-write
IN	Instruction/other
RW	Read/write of faulted bus cycle
LG	Original operand size was long word
SIZ	Remaining size of faulted bus cycle
FUNC	Function code of faulted bus cycle

The TP field defines the class of the faulted bus operation. Two BERR exception frame types are defined. One is for faults on prefetch and operand accesses, and the other is for faults during exception frame stacking:

0 — Operand or prefetch bus fault
1 — Exception processing bus fault

MV is set when the operand transfer portion of the MOVEM instruction is in progress at the time of a bus fault. If a prefetch bus fault occurs while refetching the MOVEM opcode and extension word, both the MV and IN bits will be set.

0 — MOVEM was not in progress when fault occurred
1 — MOVEM in progress when fault occurred

TR indicates that a trace exception was pending when a bus error exception was processed. The instruction that generated the trace will not be restarted upon return from the exception handler. This includes MOVEM and released write bus errors indicated by the assertion of either MV or RR in the SSW.

0 — Trace not pending
1 — Trace pending

B1 indicates that a breakpoint exception was pending on channel 1 (external breakpoint source) when a bus error exception was processed. Pending breakpoint status is stacked, regardless of the type of bus error exception.

0 — Breakpoint not pending
1 — Breakpoint pending

B0 indicates that a breakpoint exception was pending on channel 0 (internal breakpoint source) when the bus error exception was processed. Pending breakpoint status is stacked, regardless of the type of bus error exception.

0 — Breakpoint not pending
1 — Breakpoint pending

RR will be set if the faulted bus cycle was a released write. If the write is completed (rerun) in the exception handler, the RR bit should be cleared before executing RTE. The bus cycle will be rerun if the RR bit is set upon return from the exception handler.

0 — Faulted cycle was read, RMW, or unreleased write
1 — Faulted cycle was a released write

Faulted RMW bus cycles set the RM bit. RM is ignored during unstacking.

0 — Faulted cycle was non-RMW cycle
1 — Faulted cycle was either the read or write of an RMW cycle

Instruction prefetch faults are distinguished from operand (both read and write) faults by the IN bit. If IN is cleared, the error was on an operand cycle; if IN is set, the error was on an instruction prefetch. IN is ignored during unstacking.

0 — Operand
1 — Prefetch

Read and write bus cycles are distinguished by the RW bit. Read bus cycles will set the bit, and write bus cycles will clear it. The bit is reloaded into the bus controller if the RR bit is set during unstacking.

0 — Faulted cycle was an operand write
1 — Faulted cycle was a prefetch or operand read

The LG bit indicates an original operand size of long word. LG is cleared if the original operand was a byte or word — SIZ will indicate original (and remaining) size. LG is set if the original was a long word — SIZ will indicate the remaining size at the time of fault.

LG is ignored during unstacking.

0 — Original operand size was byte or word

1 — Original operand size was long word

The SSW SIZ field shows operand size remaining when a fault was detected. This field does not indicate the initial size of the operand. It also does not necessarily indicate the proper status of a dynamically sized bus cycle. Dynamic sizing occurs on the external bus and is transparent to the CPU. Byte size is shown only when the original operand was a byte. The field is reloaded into the bus controller if the RR bit is set during unstacking. The SIZ field is encoded as follows:

00 — Long word

01 — Byte

10 — Word

11 — Unused, reserved

The function code for the faulted cycle is stacked in the FUNC field of the SSW, which is a copy of [FC2:FC0] for the faulted bus cycle. This field is reloaded into the bus controller if the RR bit is set during unstacking. All unused bits are stacked as zeros and are ignored during unstacking. Further discussion of the SSW is included in **6.3.1 Types of Faults**.

6.3.1 Types of Faults

An efficient implementation of instruction restart dictates that faults on some bus cycles be treated differently than faults on other bus cycles. The CPU32 defines four fault types: released write faults, faults during exception processing, faults during MOVEM operand transfer, and faults on any other bus cycle.

6.3.1.1 Type I: Released Write Faults

CPU32 instruction pipelining can cause a final instruction write to overlap the execution of a following instruction. A write that is overlapped is called a released write. Since the machine context for the instruction that queued the write is lost as soon as the following instruction starts, it is impossible to restart the faulted instruction.

Released write faults are taken at the next instruction boundary. The stacked program counter is that of the next unexecuted instruction. If a subsequent instruction attempts an operand access while a released write fault is pending, the instruction is aborted and the write fault is acknowledged. This action prevents stale data from being used by the instruction.

The SSW for a released write fault contains the following bit pattern:

15	14	13	12	11	10	9	8	7	6	5	4 3	2 0
0	0	0	TR	B1	B0	1	0	0	0	LG	SIZ	FUNC

TR, B1, and B0 are set if the corresponding exception is pending when the BERR exception is taken. Status regarding the faulted bus cycle is reflected in the SSW LG, SIZ, and FUNC fields.

The remainder of the stack contains the program counter of the next unexecuted instruction, the current status register, the address of the faulted memory location, and the contents of the data buffer which was to be written to memory. This data is written on the stack in the format depicted in **Figure 6-3**.

6.3.1.2 Type II: Prefetch, Operand, RMW, and MOVEP Faults

The majority of BERR exceptions are included in this category — all instruction prefetches, all operand reads, all RMW cycles, and all operand accesses resulting from execution of MOVEP (except the last write of a MOVEP Rn,⟨ea⟩ or the last write of MOVEM, which are type I faults). The TAS, MOVEP, and MOVEM instructions account for all operand writes not considered released.

All type II faults cause an immediate exception that aborts the current instruction Any registers that were altered as the result of an effective address calculation (i.e., postincrement or predecrement) are restored prior to processing the bus cycle fault.

The SSW for faults in this category contains the following bit pattern:

15	14	13	12	11	10	9	8	7	6	5	4 3	2 0
0	0	0	0	B1	B0	0	RM	IN	RW	LG	SIZ	FUNC

The trace pending bit is always cleared, since the instruction will be restarted upon return from the handler. Saving a pending exception on the stack would result in a trace exception being taken prior to restarting the instruction. If the exception handler does not alter the stacked SR trace bits, the trace is requeued when the instruction is started.

The breakpoint pending bits are stacked in the SSW, even though the instruction is restarted upon return from the handler. This avoids problems with bus state analyzer equipment that has been programmed to breakpoint only the first access to a specific location, or to count accesses to that location. If this response is not desired, the exception handler can clear the bits before return. The RM, IN, RW, LG, FUNC, and SIZ fields all reflect the type of bus cycle that caused the fault. If the bus cycle was an RMW, the RM bit will be set and the RW bit will show whether the fault was on a read or write.

6.3.1.3 Type III: Faults During MOVEM Operand Transfer

Bus faults that occur as a result of MOVEM operand transfer are classified as type III faults. MOVEM Instruction prefetch faults are type II faults.

Type III faults cause an immediate exception that aborts the current instruction. None of the registers altered during execution of the faulted instruction are restored prior to execution of the fault handler. This includes any register predecremented as a result of the effective address calculation or any register overwritten during instruction execution. Since postincremented registers are not updated until the end of an instruction, the register retains its preinstruction value unless overwritten by operand movement.

The SSW for faults in this category contains the following bit pattern:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
0	1	0	TR	B1	B0	RR	0	IN	RW	LG	SIZ		FUNC	

MV is set, indicating that MOVEM should be continued from the point where the fault occurred upon return from the exception handler. TR, B1, and B0 are set if a corresponding exception is pending when the BERR exception is taken. IN is set if a bus fault occurs while refetching an opcode or an extension word during instruction restart. RW, LG, SIZ, and FUNC all reflect the type of bus cycle that caused the fault. All write faults have the RR bit set, to indicate that the write should be rerun upon return from the exception handler.

The remainder of the stack frame contains sufficient information to continue MOVEM with operand transfer following a faulted transfer. The address of the next operand to be transferred, incremented or decremented by operand size, is stored in the faulted address location ($08). The stacked transfer counter is set to 16 minus the number of transfers attempted (including the faulted cycle). Refer to **Figure 6-3** for the stacking format.

6.3.1.4 Type IV: Faults During Exception Processing

The fourth type of fault occurs during exception processing. If this exception is a second address or bus error, the machine halts in the “double bus fault” condition. However, if the exception is one that causes a four- or six-word stack frame to be written, a bus cycle fault frame is written below the faulted exception stack frame.

The SSW for a fault within an exception contains the following bit pattern:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
1	0	0	TR	B1	B0	0	0	0	1	LG	SIZ		FUNC	
		15												0

TR, B1, and B0 are set if a corresponding exception is pending when the BERR exception is taken.

The contents of the faulted exception stack frame are included in the bus fault stack frame. The pre-exception status register and the format/vector word of the faulted frame are stacked. The type of exception can be determined from the format/vector word. If the faulted exception stack frame contains six words, the program counter of the instruction that caused the initial exception is also stacked. This data is placed on the stack in the format shown in **Figure 6-4**. The return address from the initial exception is stacked for RTE.

6.3.2 Correcting a Fault

Fault correction methods are discussed in the following paragraphs.

There are two ways to complete a faulted released write bus cycle. The first is to use a software handler. The second is to rerun the bus cycle via RTE.

Type II fault handlers must terminate with RTE, but specific requirements must also be met before an instruction is restarted.

There are three varieties of Type III operand fault recovery. The first is completion of an instruction in software. The second is conversion to Type II with restart via RTE. The third is continuation from the fault via RTE.

6.3.2.1 (Type I) Completing Released Writes via Software

To complete a bus cycle in software, a handler must first read the SSW function code field to determine the appropriate address space, then access the fault address pointer on the stack, and then transfer data from the stacked image of the output buffer to the fault address.

Because the CPU32 has a 16-bit internal data bus, long operands require two bus accesses. A fault during the second access of a long operand causes the LG bit in the SSW to be set. The SIZ field indicates remaining operand size. If operand coherency is important, the complete operand must be rewritten. After a long operand is rewritten, the RR bit must be cleared. Failure to clear the RR bit can cause RTE to rerun the bus cycle. Following rewrite, it is not necessary to adjust the program counter (or other stack contents) before executing RTE.

6.3.2.2 (Type I) Completing Released Writes via RTE

An exception handler can use the RTE instruction to complete a faulted bus cycle. When RTE executes, the fault address, data output buffer, program counter, and status register are restored from the stack. Any pending breakpoint or trace exceptions, as indicated by TR, B1, and B0 in the stacked SSW, are requeued during SSW restoration. The RR bit in the SSW is checked during the unstacking operation — if it is set, the RW, FUNC, and SIZ fields are restored and the released write cycle is rerun.

To maintain long-word operand coherence, stack contents must be adjusted prior to RTE execution. The fault address must be decremented by two if LG is set and SIZ indicates a remaining byte or word. SIZ must be set to long. All other fields should be left unchanged. The bus controller uses the modified fault address and SIZ field to rerun the complete released write cycle

Manipulating the stacked SSW can cause unpredictable results because RTE checks only the RR bit to determine if a bus cycle must be rerun. Inadvertent alteration of the control bits could cause the bus cycle to be a read instead of a write, or could cause access to a different address space than the original bus cycle. If the rerun bus cycle is a read, returned data will be ignored.

6.3.2.3 (Type II) Correcting Faults via RTE

Instructions aborted because of a type II fault are restarted upon return from the exception handler. A fault handler must establish safe restart conditions. If a fault is caused by a nonresident page in a demand-paged virtual memory configuration, the fault address must be read from the stack, and the appropriate page retrieved. An RTE instruction terminates the exception handler. After unstacking the machine state, the instruction is refetched and restarted.

6.3.2.4 (Type III) Correcting Faults via Software

Sufficient information is contained in the stack frame to complete MOVEM in software. After the cause of the fault is corrected, the faulted bus cycle must be rerun. Do the following to complete an instruction through software:

A. Setup for Rerun

Read the MOVEM opcode and extension from locations pointed to by stack frame PC and PC + 2. The effective address need not be recalculated, since the next operand address is saved in the stack frame. However, the opcode effective address field must be examined to determine how to update the address register and program counter when the instruction is complete.

Adjust the mask to account for operands already transferred. Subtract the stacked operand transfer count from 16 to obtain the number of operands transferred. Scan the mask using this count value. Each time a set bit is found, clear it and decrement the counter. When the count is zero, the mask is ready for use.

Adjust the operand address. If the predecrement addressing mode is in effect, subtract the operand size from the stacked value; otherwise, add the operand size to the stacked value.

B. Rerun Instruction

Scan the mask for set bits. Read/write the selected register from/to the operand address as each bit is found.

As each operand is transferred, clear the mask bit and increment (decrement) the operand address. When all bits in the mask are cleared, all operands have been transferred.

If the addressing mode is predecrement or postincrement, update the register to complete the execution of the instruction.

If the TR bit is set in the stacked SSW, create a six-word stack frame and execute the trace handler. If either B1 or B0 in the SSW is set, create another six word stack frame and execute the hardware breakpoint handler.

De-allocate the stack and return control to the faulted program.

6.3.2.5 (Type III) Correcting Faults By Conversion and Restart

In some situations it may be necessary to rerun all the operand transfers for a faulted instruction rather than continue from a faulted operand. Clearing the MV bit in the stacked SSW converts a type III fault into a type II fault. Consequently, MOVEM, like all other type II exceptions, will be restarted upon return from the exception handler. When a fault occurs after an operand has transferred, that transfer is not “undone”. However, these memory locations are accessed a second time when the instruction is restarted. If a register used in an effective address calculation is overwritten before a fault occurs, an incorrect effective address is calculated upon instruction restart.

6.3.2.6 (Type III) Correcting Faults via RTE

The preferred method of MOVEM bus fault recovery is to correct the cause of the fault and then execute an RTE instruction without altering the stack contents.

The RTE recognizes that MOVEM was in progress when a fault occurred, restores the appropriate machine state, refetches the instruction, repeats the faulted transfer, and continues the instruction.

MOVEM is the only instruction continued upon return from an exception handler. Although the instruction is refetched, the effective address is not recalculated, and the mask is rescanned the same number of times as before the fault — modifying the code prior to RTE can cause unexpected results.

6.3.2.7 (Type IV) Correcting Faults via Software

BERR exceptions can occur during exception processing while the processor is fetching an exception vector or while it is stacking. The same stack frame and SSW are used in both cases, but each has a distinct fault address. The stacked faulted exception format/vector word identifies the type of faulted exception and the contents of the remainder of the frame. A fault address corresponding to the vector specified in the stacked format/vector word indicates that the processor could not obtain the address of the exception handler.

A BERR exception handler should execute RTE after correcting a fault. RTE restores the internal machine state, fetches the address of the original exception handler, recreates the original exception stack frame, and resumes execution at the exception handler address.

If the fault is intractable, the exception handler should rewrite the faulted exception stack frame at SP + $14 + $06 and then jump directly to the original exception handler. The stack frame can be generated from the information in the BERR frame: the pre-exception status register (SP + $0C), the format/vector word (SP + $0E), and, if the frame being written is a six-word frame, the program counter of the instruction causing the exception (SP + $10). The return program counter value is available at SP + $02.

A stacked fault address equal to the current stack pointer may indicate that, although the first exception received a BERR while stacking, the BERR exception stacking was successfully completed. This is an extremely improbable occurrence, but the CPU32 supports recovery from it. Once the exception handler determines that the fault has been corrected, recovery can proceed as described previously. If the fault cannot be corrected, move the supervisor stack to another area of memory, copy all valid stack frames to the new stack, create a faulted exception frame on top of the stack, and resume execution at the exception handler address.

6.4 CPU32 Stack Frames

The CPU32 generates three different stack frames — the normal four- and six-word frames, and the twelve-word BERR stack frame.

6.4.1 Normal Four-Word Stack Frame

This stack frame is created by interrupt, format error, TRAP #n, illegal instruction, A-line and F-line emulator trap, and privilege violation exceptions. Depending on the exception type, the program counter value is either the address of the next instruction to be executed or the address of the instruction that caused the exception (see **Figure 6-3**).

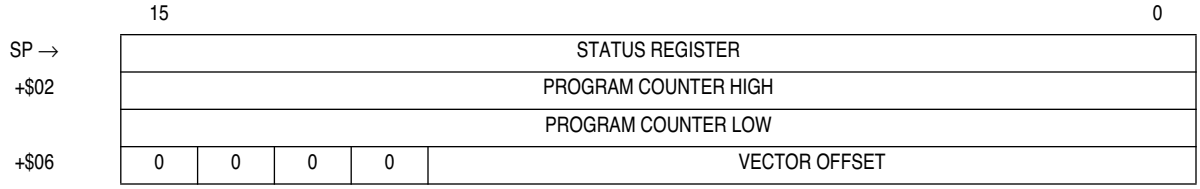


Figure 6-3 Format $0 — Four-Word Stack Frame

6.4.2 Normal Six-Word Stack Frame

This stack frame (see **Figure 6-4**) is created by instruction-related traps, which include CHK, CHK2, TRAPcc, TRAPV, and divide-by-zero, and by trace exceptions. The faulted instruction program counter value is the address of the instruction that caused the exception. The next program counter value (the address to which RTE returns) is the address of the next instruction to be executed.

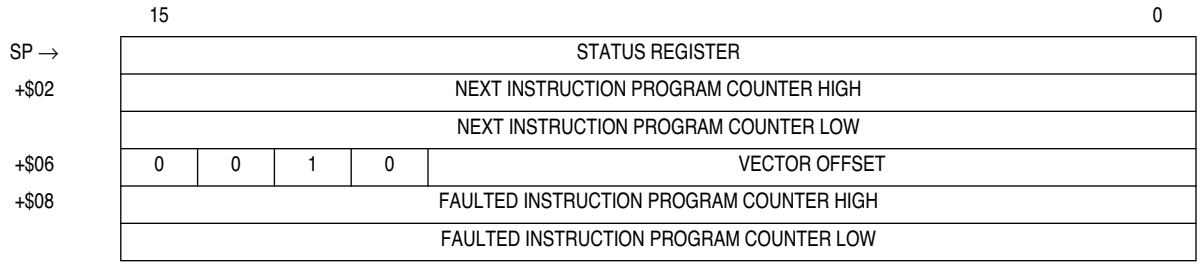


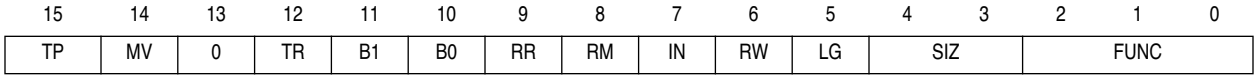
Figure 6-4 Format $2 — Six-Word Stack Frame

Hardware breakpoints also utilize this format. The faulted instruction program counter value is the address of the instruction executing when the breakpoint was sensed. Usually this is the address of the instruction that caused the breakpoint, but, because released writes can overlap following instructions, the faulted instruction program counter may point to an instruction following the instruction that caused the breakpoint. The address to which RTE returns is the address of the next instruction to be executed

6.4.3 BERR Stack Frame

This stack frame is created when a bus cycle fault is detected. The CPU32 BERR stack frame differs significantly from the equivalent stack frames of other M68000 Family members. The only internal machine state required in the CPU32 stack frame is the bus controller state at the time of the error, and a single register.

Bus operation in progress at the time of a fault is conveyed by the SSW.



The BERR stack frame is 12 words in length. There are three variations of the frame, each distinguished by different values in the SSW TP and MV fields.

An internal transfer count register appears at location SP + 14 in all bus error stack frames. The register contains an 8-bit microcode revision number, and, for type III faults, an 8-bit transfer count. Register format is shown in **Figure 6-5**.

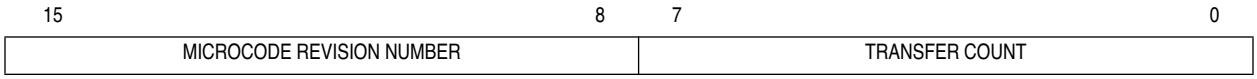


Figure 6-5 Internal Transfer Count Register

The microcode revision number is checked before a BERR stack frame is restored via RTE. In a multiprocessor system, this check insures that a processor using stacked information is at the same revision level as the processor that created it.

The transfer count is ignored unless the MV bit in the stacked SSW is set. If the MV bit is set, the least significant byte of the internal register is reloaded into the MOVEM transfer counter during RTE execution.

For faults occurring during normal instruction execution (both prefetches and non-MOVEM operand accesses) SSW [TP:MV] = 00. Stack frame format is shown in **Figure 6-6**.

Faults that occur during the operand portion of the MOVEM instruction.are identified by SSW [TP:MV] = 01. Stack frame format is shown in **Figure 6-7**.

When a bus error occurs during exception processing, SSW [TP:MV] = 10. The frame shown in **Figure 6-8** is written below the faulting frame. Stacking begins at the address pointed to by SP – 6 (SP value is the value before initial stacking on the faulted frame).

The frame can have either four or six words, depending on the type of error. Four word stack frames do not include the faulted instruction program counter (the internal transfer count register is located at SP + $10 and the SSW is located at SP + $12).

The fault address of a dynamically sized bus cycle is the address of the upper byte, regardless of the byte that caused the error.

	15					0
SP →	STATUS REGISTER					
+$02	RETURN PROGRAM COUNTER HIGH					
	RETURN PROGRAM COUNTER LOW					
+$06	1	1	0	0	VECTOR OFFSET	
+$08	FAULTED ADDRESS HIGH					
	FAULTED ADDRESS LOW					
+$0C	DBUF HIGH					
	DBUF LOW					
+$10	CURRENT INSTRUCTION PROGRAM COUNTER HIGH					
	CURRENT INSTRUCTION PROGRAM COUNTER LOW					
+$14	INTERNAL TRANSFER COUNT REGISTER					
+$16	0	0	SPECIAL STATUS WORD			

Figure 6-6 Format $C — BERR Stack for Prefetches and Operands

	15					0
SP →	STATUS REGISTER					
+$02	RETURN PROGRAM COUNTER HIGH					
	RETURN PROGRAM COUNTER LOW					
+$06	1	1	0	0	VECTOR OFFSET	
+$08	FAULTED ADDRESS HIGH					
	FAULTED ADDRESS LOW					
+$0C	DBUF HIGH					
	DBUF LOW					
+$10	CURRENT INSTRUCTION PROGRAM COUNTER HIGH					
	CURRENT INSTRUCTION PROGRAM COUNTER LOW					
+$14	INTERNAL TRANSFER COUNT REGISTER					
+$16	0	1	SPECIAL STATUS WORD			

Figure 6-7 Format $C — BERR Stack on MOVEM Operand

	15					0
SP →	STATUS REGISTER					
+$02	NEXT INSTRUCTION PROGRAM COUNTER HIGH					
	NEXT INSTRUCTION PROGRAM COUNTER LOW					
+$06	1	1	0	0	VECTOR OFFSET	
+$08	FAULTED ADDRESS HIGH					
	FAULTED ADDRESS LOW					
+$0C	PRE-EXCEPTION STATUS REGISTER					
	FAULTED EXCEPTION FORMAT/VECTOR WORD					
+$10	FAULTED INSTRUCTION PROGRAM COUNTER HIGH (SIX WORD FRAME ONLY)					
	FAULTED INSTRUCTION PROGRAM COUNTER LOW (SIX WORD FRAME ONLY)					
+$14	INTERNAL TRANSFER COUNT REGISTER					
+$16	0	1	SPECIAL STATUS WORD			

Figure 6-8 Format $C — Four- and Six-Word BERR Stack

SECTION 7 DEVELOPMENT SUPPORT

All M68000 Family members have the following special features that facilitate applications development:

Trace on Instruction Execution — All M68000 processors include an instruction-by-instruction tracing facility to aid in program development. The MC68020, MC68030, and CPU32 can also trace those instructions that change program flow. In trace mode, an exception is generated after each instruction is executed, allowing a debugger program to monitor execution of a program under test. See **6.2.10 Tracing** for more information.

Breakpoint Instruction — An emulator can insert software breakpoints into target code to indicate when a breakpoint occurs. On the MC68010, MC68020, MC68030, and CPU32, this function is provided via illegal instructions ($4848–$484F) that serve as breakpoint instructions. See **6.2.5 Software Breakpoints** for more information.

Unimplemented Instruction Emulation — When an attempt is made to execute an illegal instruction, an illegal instruction exception occurs. Unimplemented instructions (F-line, A-line) utilize separate exception vectors to permit efficient emulation of unimplemented instructions in software. See **6.2.8 Illegal or Unimplemented Instructions** for more information.

7.1 CPU32 Integrated Development Support

In addition to standard MC68000 family capabilities, the CPU32 has features to support advanced integrated system development. These features include background debug mode, deterministic opcode tracking, hardware breakpoints, and internal visibility in a single-chip environment.

7.1.1 Background Debug Mode (BDM) Overview

Microprocessor systems generally provide a debugger, implemented in software, for system analysis at the lowest level. The BDM on the CPU32 is unique because the debugger is implemented in CPU microcode.

BDM incorporates a full set of debug options — registers can be viewed and/or altered, memory can be read or written, and test features can be invoked.

A resident debugger simplifies implementation of an in-circuit emulator. In a common setup (see **Figure 7-1**), emulator hardware replaces the target system processor. A complex, expensive pod-and-cable interface provides a communication path between target system and emulator.

By contrast, an integrated debugger supports use of a bus state analyzer (BSA) for in-circuit emulation. The processor remains in the target system (see **Figure 7-2**) and the

interface is simplified. The BSA monitors target processor operation and the on-chip debugger controls the operating environment. Emulation is much “closer” to target hardware, and many interfacing problems (i.e., limitations on high-frequency operation, AC and DC parametric mismatches, and restrictions on cable length) are minimized.

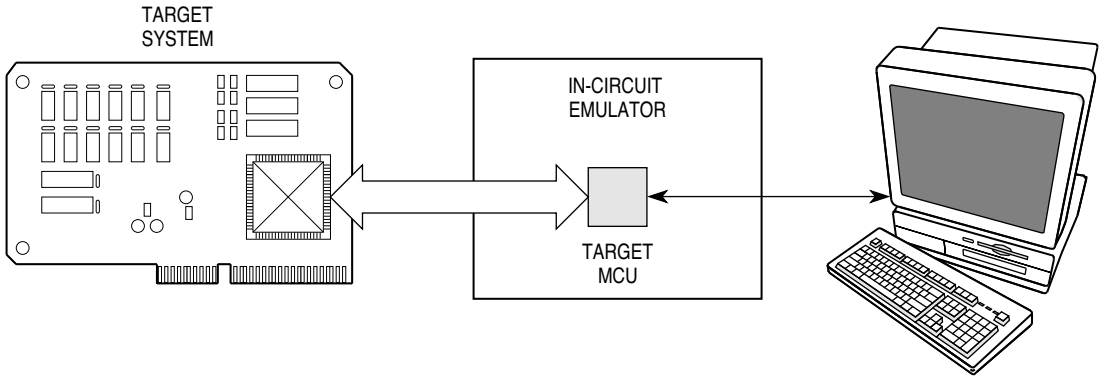


Figure 7-1 In-Circuit Emulator Configuration

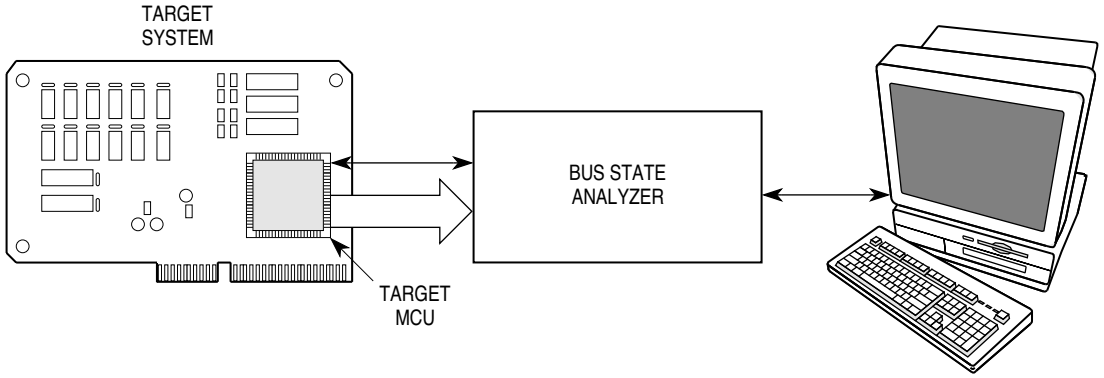


Figure 7-2 Bus State Analyzer Configuration

7.1.2 Deterministic Opcode Tracking Overview

CPU32 function code outputs are augmented by two supplementary signals that monitor the instruction pipeline. The instruction fetch (IFETCH) output identifies bus cycles in which data is loaded into the pipeline, and signals pipeline flushes. The instruction pipe (IPIPE) output indicates when each mid-instruction pipeline advance occurs and when instruction execution begins. These signals allow a BSA to synchronize with instruction stream activity. Refer to **7.3 Deterministic Opcode Tracking** for complete information.

7.1.3 On-Chip Hardware Breakpoint Overview

An external breakpoint input and an on-chip hardware breakpoint capability permit breakpoint trap on any memory access. Off-chip address comparators preclude breakpoints on internal accesses unless show cycles are enabled. Breakpoints on prefetched instructions, which are flushed from the pipeline before execution, are not acknowledged, but operand breakpoints are always acknowledged. Acknowledged breakpoints can initiate either exception processing or background debug mode (BDM). See **6.2.6 Hardware Breakpoints** for more information.

7.2 Background Debug Mode (BDM)

BDM is an alternate CPU32 operating mode. During BDM, normal instruction execution is suspended, and special microcode performs debugging functions under external control. **Figure 7-3** is a BDM block diagram.

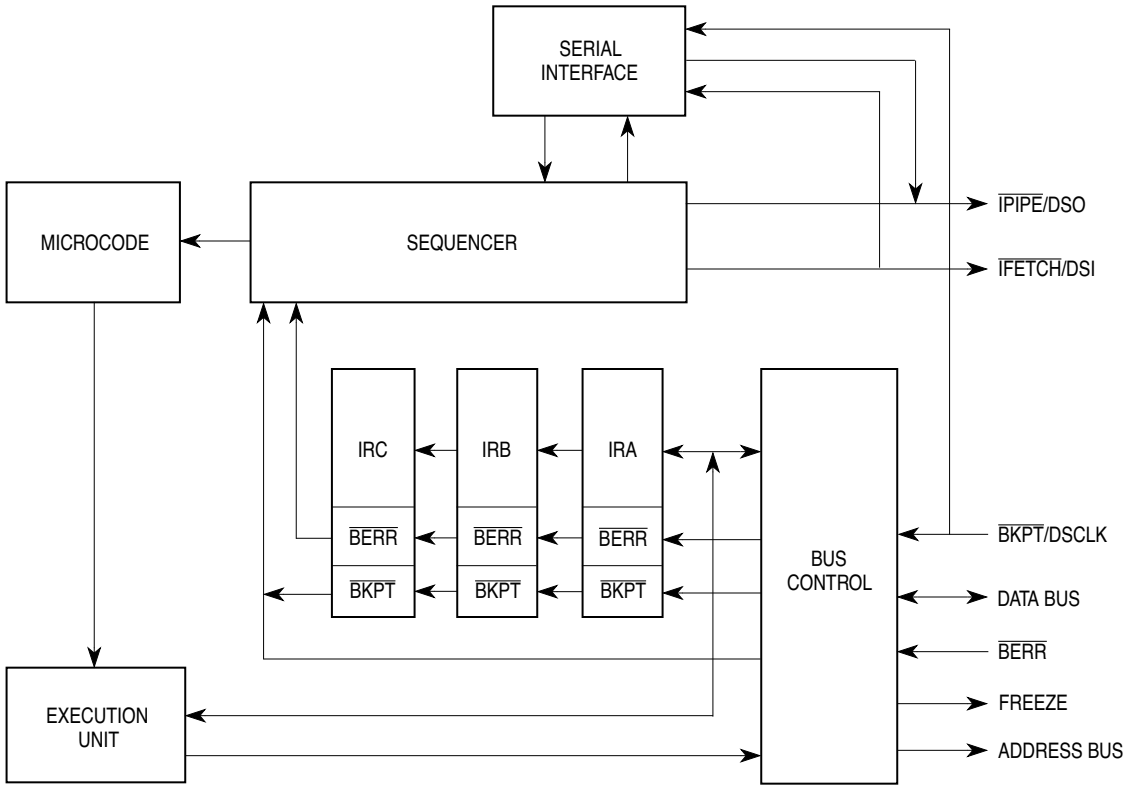


Figure 7-3 BDM Block Diagram

BDM can be initiated in several ways — by externally generated breakpoints, by internal peripheral breakpoints, by the background (BGND) instruction, or by catastrophic exception conditions. While in BDM, the CPU32 ceases to fetch instructions via the parallel bus and communicates with the development system via a dedicated, high-speed, SPI-type serial command interface.

7.2.1 Enabling BDM

Accidentally entering BDM in a non-development environment could lock up the CPU32 since the serial command interface would probably not be available. For this reason, BDM is enabled during reset via the breakpoint ($\overline{BKPT}$) signal.

BDM operation is enabled when $\overline{BKPT}$ is asserted (low), at the rising edge of $\overline{RESET}$. BDM remains enabled until the next system reset. A high $\overline{BKPT}$ signal on the trailing edge of RESET disables BDM. $\overline{BKPT}$ is relatched on each rising transition of $\overline{RESET}$. $\overline{BKPT}$ is synchronized internally, and must be held low for at least two clock cycles prior to negation of $\overline{RESET}$.

BDM enable logic must be designed with special care. If hold time on $\overline{BKPT}$ extends into the first bus cycle following reset, the bus cycle could inadvertently be tagged with a breakpoint. Refer to the system integration module user's manual for timing information.

7.2.2 BDM Sources

When BDM is enabled, any of several sources can cause the transition from normal mode to BDM. These sources include external breakpoint hardware, the BGND instruction, a double bus fault, and internal peripheral breakpoints. If BDM is not enabled when an exception condition occurs, the exception is processed normally. **Table 7-1** summarizes the processing of each source for both enabled and disabled cases. As shown in **Table 7-1**, the BKPT instruction never causes a transition into BDM.

Table 7-1 BDM Source Summary

Source	BDM Enabled	BDM Disabled
$\overline{BKPT}$	Background	Breakpoint Exception
Double Bus Fault	Background	Halted
BGND Instruction	Background	Illegal Instruction
BKPT Instruction	Opcode Substitution/ Illegal Instruction	Opcode Substitution/ Illegal Instruction

7.2.2.1 External $\overline{BKPT}$ Signal

Once enabled, BDM is initiated whenever assertion of $\overline{BKPT}$ is acknowledged. If BDM is disabled, a breakpoint exception (vector $0C) is acknowledged. The $\overline{BKPT}$ input has the same timing relationship to the data strobe trailing edge as does read cycle data. There is no breakpoint acknowledge bus cycle when BDM is entered.

7.2.2.2 BGND Instruction

An illegal instruction, $4AFA, is reserved for use by development tools. The CPU32 defines $4AFA (BGND) to be a BDM entry point when BDM is enabled. If BDM is disabled, an illegal instruction trap is acknowledged. Illegal instruction traps are discussed in **6.2.8 Illegal or Unimplemented Instructions**.

7.2.2.3 Double Bus Fault

The CPU32 normally treats a double bus fault, or two bus faults in succession, as a catastrophic system error, and halts. When this condition occurs during initial system debug (a fault in the reset logic), further debugging is impossible until the problem is corrected. In BDM, the fault can be temporarily bypassed, so that its origin can be isolated and eliminated.

7.2.2.4 Peripheral Breakpoints

CPU32 peripheral breakpoints are implemented in the same way as external breakpoints — peripherals request breakpoints by asserting the $\overline{\text{BKPT}}$ signal. Consult the appropriate peripheral user's manual for additional details on the generation of peripheral breakpoints.

7.2.3 Entering BDM

When the processor detects a breakpoint or a double bus fault, or decodes a BGND instruction, it suspends instruction execution and asserts the FREEZE output. This is the first indication that the processor has entered BDM. Once FREEZE has been asserted, the CPU enables the serial communication hardware and awaits a command.

The CPU writes a unique value indicating the source of BDM transition into temporary register A (ATEMP) as part of the process of entering BDM. A user can poll ATEMP and determine the source (see **Table 7-2**) by issuing a read system register command (RSREG). ATEMP is used in most debugger commands for temporary storage — it is imperative that the RSREG command be the first command issued after transition into BDM.

Table 7-2 Polling the BDM Entry Source

Source	ATEMP [31:16]	ATEMP [15:0]
Double Bus Fault	SSW*	$FFFF
BGND Instruction	$0000	$0001
Hardware Breakpoint	$0000	$0000

*Special status word (SSW) is described in detail in **6.3 Fault Recovery**.

A double bus fault during initial stack pointer/program counter (SP/PC) fetch sequence is distinguished by a value of $FFFFFFFF in the current instruction PC. At no other time will the processor write an odd value into this register.

7.2.4 Command Execution

Figure 7-4 summarizes BDM command execution. Commands consist of one 16-bit operation word and can include one or more 16-bit extension words. Each incoming word is read as it is assembled by the serial interface. The microcode routine corresponding to a command is executed as soon as the command is complete. Result operands are loaded into the output shift register to be shifted out as the next command is read. This process is repeated for each command until the CPU returns to normal operating mode.

7.2.5 Background Mode Registers

BDM processing uses three special purpose registers to keep track of program context during development. A description of each follows.

7.2.5.1 Fault Address Register (FAR)

The FAR contains the address of the faulting bus cycle immediately following a bus or address error. This address remains available until overwritten by a subsequent bus cycle. Following a double bus fault, the FAR contains the address of the last bus cycle. The address of the first fault (if there was one) is not visible to the user.

7.2.5.2 Return Program Counter (RPC)

The RPC points to the location where fetching will commence after transition from background mode to normal mode. This register should be accessed to change the flow of a program under development. Changing the RPC to an odd value will cause an address error when normal mode prefetching begins.

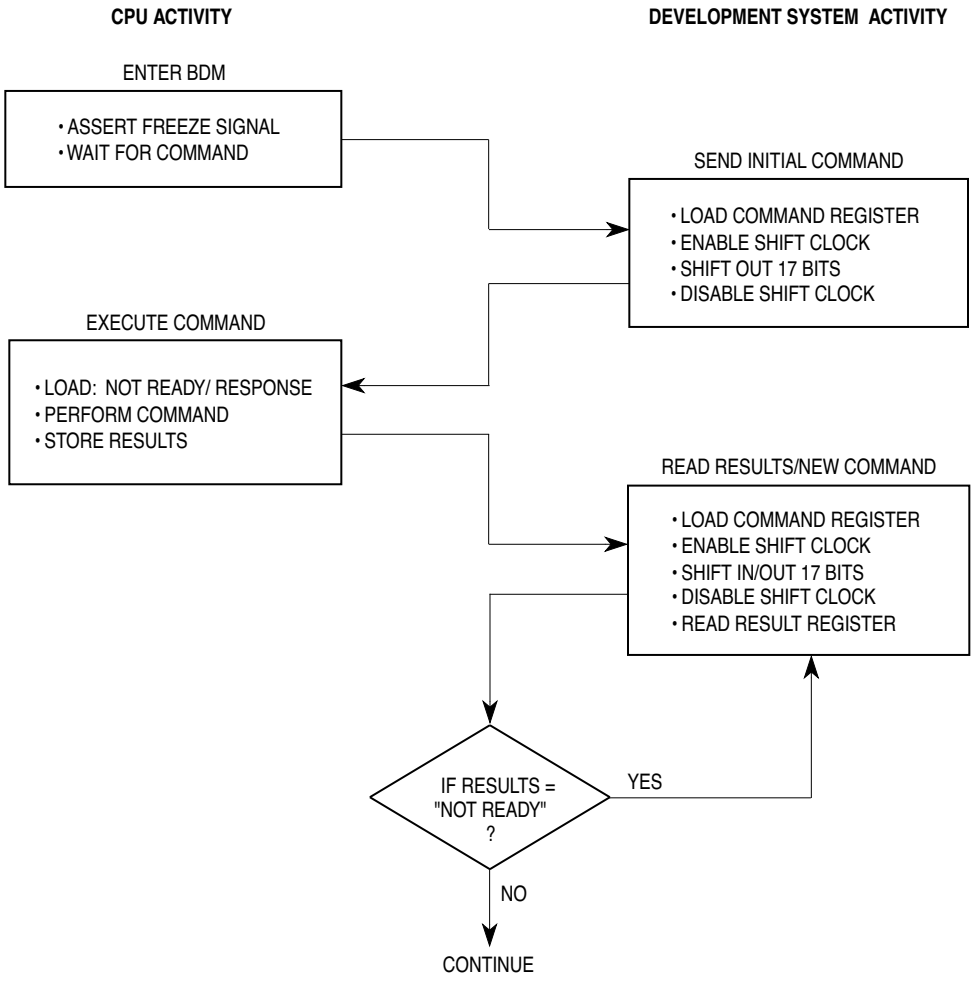


Figure 7-4 BDM Command Execution Flowchart

7.2.5.3 Current Instruction Program Counter (PCC)

The PCC holds a pointer to the first word of the last instruction executed prior to transition into background mode. Due to instruction pipelining, the instruction pointed to may not be the instruction which caused the transition. An example is a breakpoint on a released write. The bus cycle may overlap as many as two subsequent instructions before stalling the instruction sequencer. A breakpoint asserted during this cycle will not be acknowledged until the end of the instruction executing at completion of the bus cycle. PCC will contain $00000001 if BDM is entered via a double bus fault immediately out of reset.

7.2.6 Returning from BDM

BDM is terminated when a resume execution (GO) or call user code (CALL) command is received. Both GO and CALL flush the instruction pipeline and refetch instructions from the location pointed to by the RPC.

The return PC and the memory space referred to by the status register SUPV bit reflect any changes made during BDM. FREEZE is negated prior to initiating the first prefetch. Upon negation of FREEZE, the serial subsystem is disabled, and the signals revert to $\overline{\text{IPIPE}}$/$\overline{\text{IFETCH}}$ functionality.

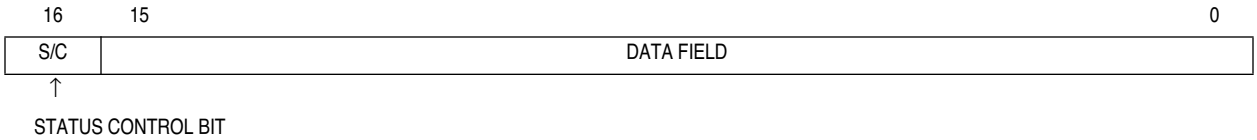
7.2.7 Serial Interface

Communication with the CPU32 during BDM occurs via a dedicated serial interface, which shares pins with other development features. The $\overline{\text{BKPT}}$ signal becomes the serial clock (DSCLK); serial input data (DSI) is received on $\overline{\text{IFETCH}}$, and serial output data (DSO) is transmitted on $\overline{\text{IPIPE}}$.

The serial interface uses a full-duplex synchronous protocol similar to the serial peripheral interface (SPI) protocol. The development system serves as the master of the serial link since it is responsible for the generation of DSCLK. If DSCLK is derived from the CPU32 system clock, development system serial logic is unhindered by the operating frequency of the target processor. Operable frequency range of the serial clock is from DC to one-half the processor system clock frequency.

The serial interface operates in full-duplex mode —data is transmitted and received simultaneously by both master and slave devices. In general, data transitions occur on the falling edge of DSCLK and are stable by the following rising edge of DSCLK. Data is transmitted MSB first, and is latched on the rising edge of DSCLK.

The serial data word is 17 bits wide — 16 data bits and a status/control bit.



Bit 16 indicates status of CPU-generated messages as shown in **Table 7-3**.

Table 7-3 CPU Generated Message Encoding

Bit 16	Data	Message Type
0	xxxx	Valid Data Transfer
0	FFFF	Command Complete; Status OK
1	0000	Not Ready with Response; Come Again
1	0001	BERR Terminated Bus Cycle; Data Invalid
1	FFFF	Illegal Command

Command and data transfers initiated by the development system should clear bit 16. The current implementation ignores this bit; however, Motorola reserves the right to use this bit for future enhancements.

7.2.7.1 CPU Serial Logic

CPU serial logic, shown in the left-hand portion of **Figure 7-5**, consists of transmit and receive shift registers and of control logic that includes synchronization, serial clock generation circuitry, and a received bit counter.

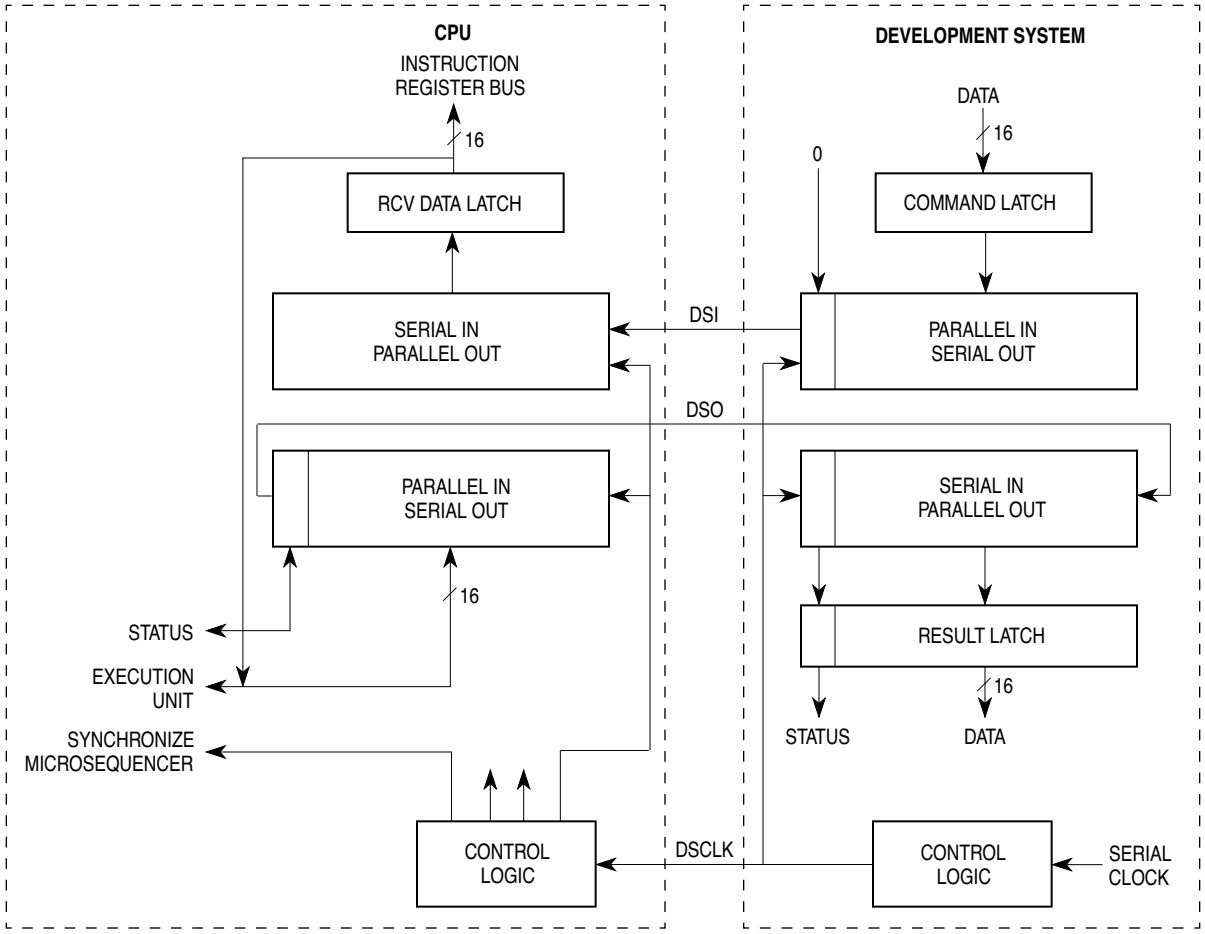


Figure 7-5 Debug Serial I/O Block Diagram

Both DSCLK and DSI are synchronized to on-chip clocks, thereby minimizing the chance of propagating metastable states into the serial state machine. Data is sampled during the high phase of CLKOUT. At the falling edge of CLKOUT, the sampled value is made available to internal logic. If there is no synchronization between CPU32 and development system hardware, the minimum hold time on DSI with respect to DSCLK is one full period of CLKOUT.

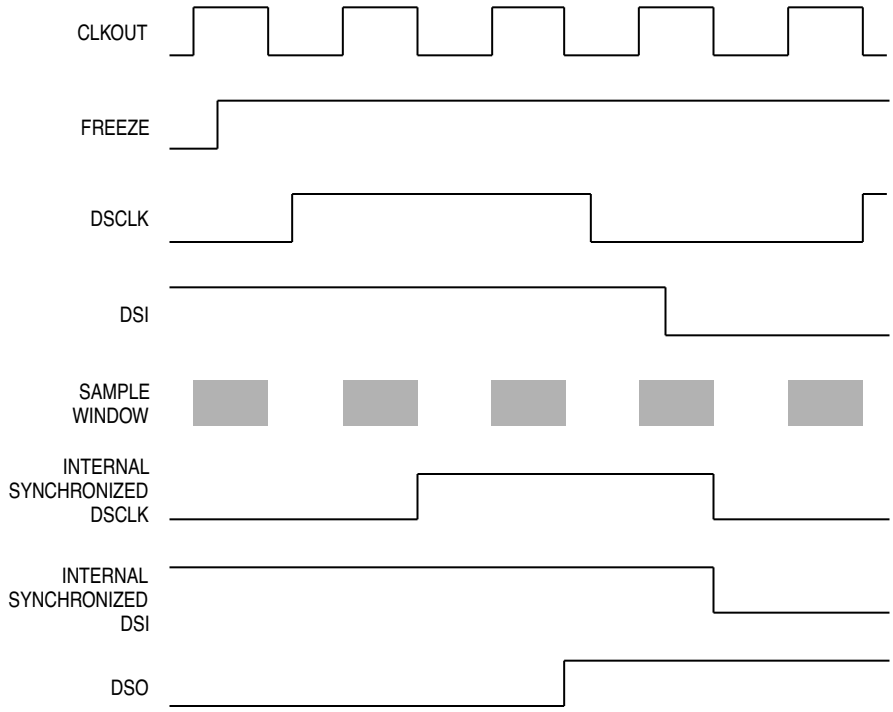


Figure 7-6 Serial Interface Timing Diagram

The serial state machine begins a sequence of events based on the rising edge of the synchronized DSCLK (see **Figure 7-6**). Synchronized serial data is transferred to the input shift register, and the received bit counter is decremented. One-half clock period later, the output shift register is updated, bringing the next output bit to the DSO signal. DSO changes relative to the rising edge of DSCLK and does not necessarily remain stable until the falling edge of DSCLK.

One clock period after the synchronized DSCLK has been seen internally, the updated counter value is checked. If the counter has reached zero, the receive data latch is updated from the input shift register. At this same time, the output shift register is reloaded with the “not ready/come again” response. Once the receive data latch has been loaded, the CPU is released to act on the new data. Response data overwrites the “not ready” response when the CPU has completed the current operation.

Data written into the output shift register appears immediately on the DSO signal. In general, this action changes the state of the signal from a high (“not ready” response status bit) to a low (valid data status bit) logic level. However, this level change only occurs if the command completes successfully. Error conditions overwrite the “not ready” response with the appropriate response that also has the status bit set.

A user can use the state change on DSO to signal hardware that the next serial transfer may begin. A time-out of sufficient length to trap error conditions that do not change the state of DSO should also be incorporated into the design. Hardware interlocks in the CPU prevent result data from corrupting serial transfers in progress.

7.2.7.2 Development System Serial Logic

The development system, as the master of the serial data link, must supply the serial clock. However, normal and BDM operations could interact if the clock generator is not properly designed.

Breakpoint requests are made by asserting $\overline{\text{BKPT}}$ to the low state in either of two ways. The primary method is to assert $\overline{\text{BKPT}}$ during a single bus cycle for which an exception is desired. Another method is to assert $\overline{\text{BKPT}}$, then continue to assert it until the CPU32 responds by asserting FREEZE. This method is useful for forcing a transition into BDM when the bus is not being monitored. Each of these methods requires a slightly different serial logic design to avoid spurious serial clocks.

Figure 7-7 represents the timing required for asserting $\overline{\text{BKPT}}$ during a single bus cycle.

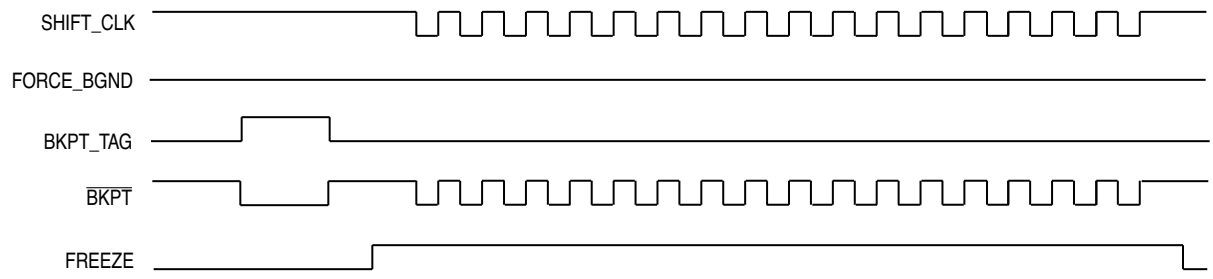


Figure 7-7 $\overline{\text{BKPT}}$ Timing for Single Bus Cycle

Figure 7-8 depicts the timing of the $\overline{\text{BKPT}}$/FREEZE method. In both cases, the serial clock is left high after the final shift of each transfer. This technique eliminates the possibility of accidentally tagging the prefetch initiated at the conclusion of a BDM session. As mentioned previously, all timing within the CPU is derived from the rising edge of the clock; the falling edge is effectively ignored.

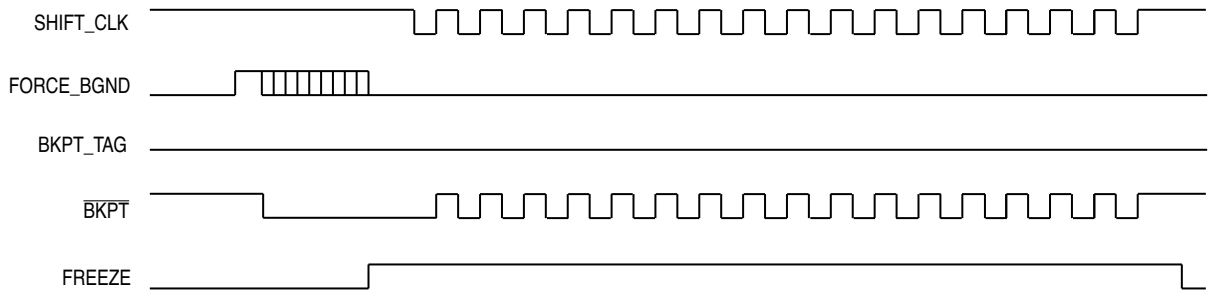


Figure 7-8 $\overline{\text{BKPT}}$ Timing for Forcing BDM

Figure 7-9 represents a sample circuit providing for both BKPT assertion methods. As the name implies, FORCE_BGND is used to force a transition into BDM by the assertion of $\overline{\text{BKPT}}$. FORCE_BGND can be a short pulse or can remain asserted until FREEZE is asserted. Once asserted, the set-reset latch holds $\overline{\text{BKPT}}$ low until the first SHIFT_CLK is applied.

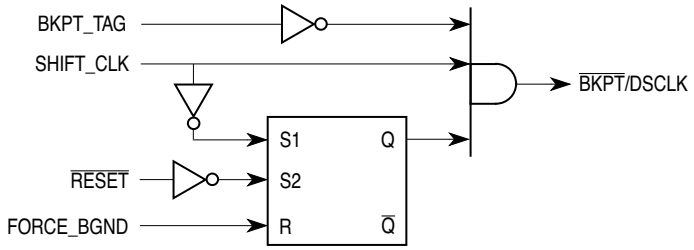


Figure 7-9 $\overline{\text{BKPT}}$/DSCLK Logic Diagram

BKPT_TAG should be timed to the bus cycles since it is not latched. If extended past the assertion of FREEZE, the negation of BKPT_TAG appears to the CPU32 as the first DSCLK.

DSCLK is the gated serial clock. Normally high, it pulses low for each bit to be transferred. At the end of the seventeenth clock period, it remains high until the start of the next transmission. Clock frequency is implementation dependent and may range from DC to the maximum specified frequency. Although performance considerations might dictate a hardware implementation, software solutions are not precluded, provided serial bus timing is maintained.

7.2.8 Command Set

Following is a description of the command set available in BDM.

7.2.8.1 Command Format

The following standard bit format is utilized by all BDM commands.

15 10	9	8	7 6	5	4	3	2 0
OPERATION	0	R/W	OP SIZE	0	0	A/D	REGISTER
EXTENSION WORD(S)							

Operation Field:

Commands are distinguished by the operation field. This 6-bit field provides for a maximum of 64 unique commands.

R/W Field:

Direction of operand transfer is specified by this field. When the bit is set, the transfer is from CPU to development system. When the bit is clear, data is written to the CPU or to memory from the development system.

Operand Size:

For sized operations, this field specifies the operand data size. All addresses are expressed as 32-bit absolute values. The size field is encoded as follows:

Encoding	Operand Size
00	Byte
01	Word
10	Long
11	Reserved

Address/Data (A/D) Field:

The A/D field is used by commands that operate on address and data registers. It determines whether the register field specifies a data or address register. One indicates an address register; zero, a data register. For other commands, this field may be interpreted differently.

Register Field:

In most commands, this field specifies the register number when operating on an address or data register.

Extension Words (as required):

At this time, no command requires an extension word to specify fully the operation to be performed, but some commands require extension words for addresses or immediate data. Addresses require two extension words because only absolute long addressing is permitted. Immediate data can be either one or two words in length — byte and word data each require a single extension word, long-word data requires two words. Both operands and addresses are transferred most significant word first.

7.2.8.2 Command Sequence Diagram

A command sequence diagram illustrates the serial bus traffic for each command. Each bubble in the diagram represents a single 17-bit transfer across the bus. The top half in each diagram corresponds to the data transmitted by the development system to the CPU; the bottom half corresponds to the data returned by the CPU in response to the development system commands. Command and result transactions are overlapped to minimize latency.

Figure 7-10 demonstrates the use of command sequence diagrams.

The cycle in which the command is issued contains the development system command mnemonic (in this example, read memory location). During the same cycle, the CPU responds with either the lowest order results of the previous command or with a command complete status (if no results were required).

During the second cycle, the development system supplies the high-order 16 bits of the memory address. The CPU returns a “not ready” response unless the received command was decoded as unimplemented, in which case the response data is the illegal command encoding. If an illegal command response occurs, the development system should retransmit the command.

NOTE

The “not ready” response can be ignored unless a memory bus cycle is in progress. Otherwise, the CPU can accept a new serial transfer with eight system clock periods.

In the third cycle, the development system supplies the low-order 16 bits of a memory address. The CPU always returns the “not ready” response in this cycle. At the completion of the third cycle, the CPU initiates a memory read operation. Any serial transfers that begin while the memory access is in progress return the “not ready” response.

Results are returned in the two serial transfer cycles following the completion of memory access. The data transmitted to the CPU during the final transfer is the opcode for the following command. Should a memory access generate either a bus or address error, an error status is returned in place of the result data.

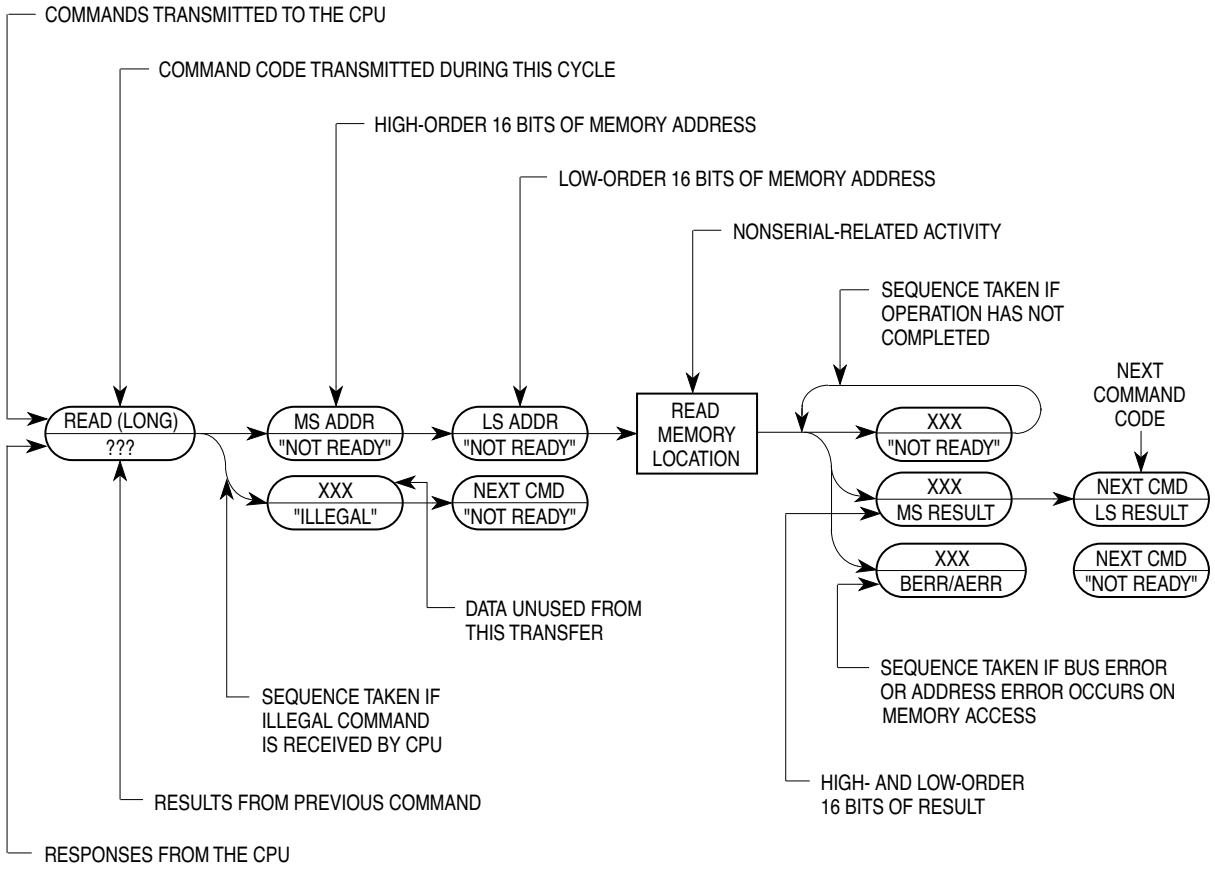


Figure 7-10 Command-Sequence-Diagram Example

7.2.8.3 Command Set Summary

The BDM command set is summarized in **Table 7-4**. Subsequent paragraphs contain detailed descriptions of each command.

Table 7-4 BDM Command Summary

Command	Mnemonic	Description
Read A/D Register	RAREG/RDREG	Read the selected address or data register and return the results via the serial interface.
Write A/D Register	WAREG/WDREG	The data operand is written to the specified address or data register.
Read System Register	RSREG	The specified system control register is read. All registers that can be read in supervisor mode can be read in BDM.
Write System Register	WSREG	The operand data is written into the specified system control register.
Read Memory Location	READ	Read the sized data at the memory location specified by the long-word address. The source function code register (SFC) determines the address space accessed.
Write Memory Location	WRITE	Write the operand data to the memory location specified by the long-word address. The destination function code register (DFC) register determines the address space accessed.
Dump Memory Block	DUMP	Used in conjunction with the READ command to dump large blocks of memory. An initial READ is executed to set up the starting address of the block and to retrieve the first result. Subsequent operands are retrieved with the DUMP command.
Fill Memory Block	FILL	Used in conjunction with the WRITE command to fill large blocks of memory. An initial WRITE is executed to set up the starting address of the block and to supply the first operand. Subsequent operands are written with the FILL command.
Resume Execution	GO	The pipeline is flushed and refilled before resuming instruction execution at the return PC.
Call User Code	CALL	Current PC is stacked at the location of the current SP. Instruction execution begins at user patch code.
Reset Peripherals	RST	Asserts RESET for 512 clock cycles. The CPU is not reset by this command. Synonymous with the CPU RESET instruction.
No Operation	NOP	NOP performs no operation and may be used as a null command.

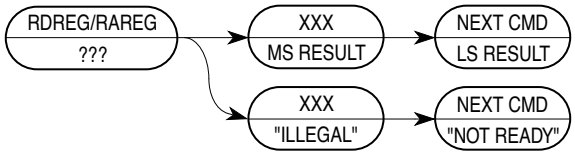
7.2.8.4 Read A/D Register (RAREG/RDREG)

Read the selected address or data register and return the results via the serial interface.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
0	0	1	0	0	0	0	1	1	0	0	0	A/D	REGISTER	

Command Sequence:



Operand Data:

None

Result Data:

The contents of the selected register are returned as a long-word value. The data is returned most significant word first.

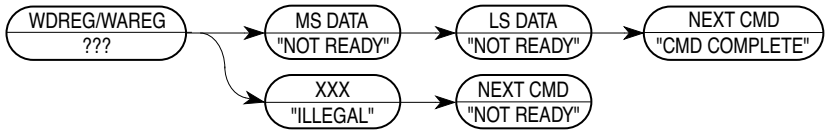
7.2.8.5 Write A/D Register (WAREG/WDREG)

The operand (long-word) data is written to the specified address or data register. All 32 bits of the register are altered by the write.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	0
0	0	1	0	0	0	0	0	1	0	0	0	A/D	REGISTER	

Command Sequence:



Operand Data:

Long-word data is written into the specified address or data register. The data is supplied most significant word first.

Result Data:

Command complete status ($0FFFF) is returned when register write is complete.

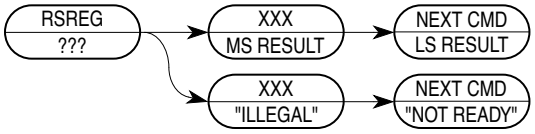
7.2.8.6 Read System Register (RSREG)

The specified system control register is read. All registers that can be read in supervisor mode can be read in BDM. Several internal temporary registers are also accessible.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	0
0	0	1	0	0	1	0	0	1	0	0	0	REGISTER	

Command Sequence:



Operand Data:

None

Result Data:

Always returns 32 bits of data, regardless of the size of the register being read. If the register is less than 32 bits, the result is returned zero extended.

Register Field:

The system control register is specified by the register field according to the following table:

System Register	Select Code
Return Program Counter (RPC)	0000
Current Instruction Program Counter (PCC)	0001
Status Register (SR)	1011
User Stack Pointer (USP)	1100
Supervisor Stack Pointer (SSP)	1101
Source Function Code Register (SFC)	1110
Destination Function Code Register (DFC)	1111
Temporary Register A (ATEMP)	1000
Fault Address Register (FAR)	1001
Vector Base Register (VBR)	1010

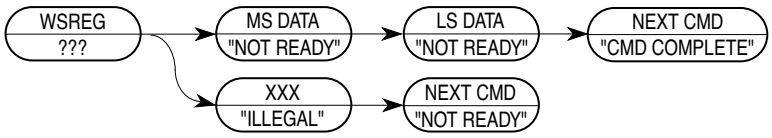
7.2.8.7 Write System Register (WSREG)

Operand data is written into the specified system control register. All registers that can be written in supervisor mode can be written in BDM. Several internal temporary registers are also accessible.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	0
0	0	1	0	0	1	0	0	1	0	0	0	REGISTER	

Command Sequence:



Operand Data:

The data to be written into the register is always supplied as a 32-bit long word. If the register is less than 32 bits, the least significant word is used.

Result Data:

“Command complete” status is returned when register write is complete.

Register Field:

The system control register is specified by the register field according to the following table. The FAR is a read-only register — any write to it is ignored.

System Register	Select Code
Return Program Counter (RPC)	0000
Current Instruction Program Counter (PCC)	0001
Status Register (SR)	1011
User Stack Pointer (USP)	1100
Supervisor Stack Pointer (SSP)	1101
Source Function Code Register (SFC)	1110
Destination Function Code Register (DFC)	1111
Temporary Register A (ATEMP)	1000
Fault Address Register (FAR)	1001
Vector Base Register (VBR)	1010

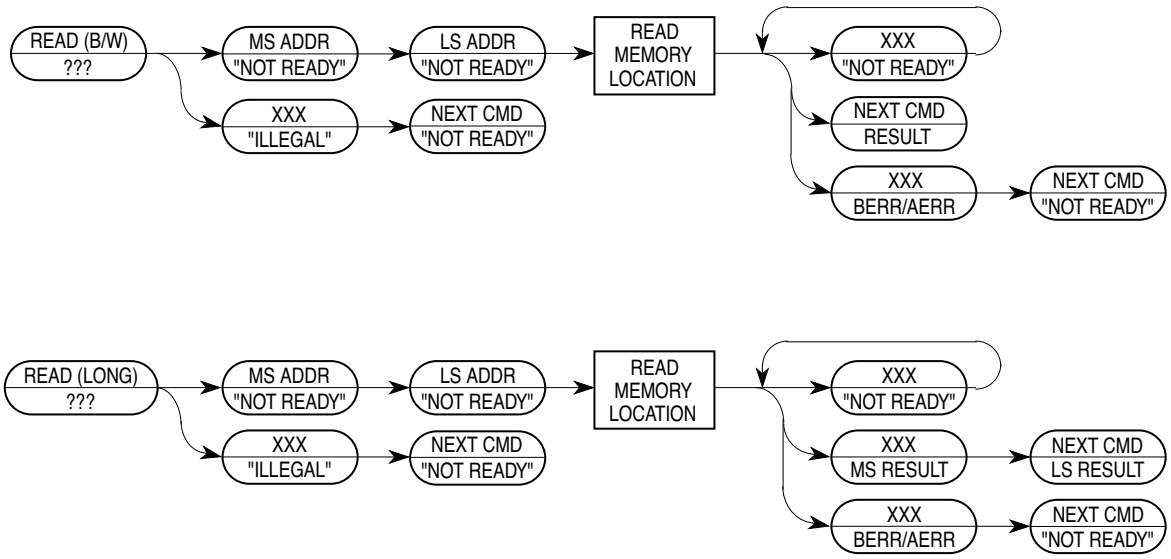
7.2.8.8 Read Memory Location (READ)

Read the sized data at the memory location specified by the long-word address. Only absolute addressing is supported. The SFC register determines the address space accessed. Valid data sizes include byte, word, or long word.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3			0
0	0	0	1	1	0	0	1	OP SIZE		0	0	0	0	0	0

Command Sequence:



Operand Data:

The single operand is the long-word address of the requested memory location.

Result Data:

The requested data is returned as either a word or long word. Byte data is returned in the least significant byte of a word result, with the upper byte cleared. Word results return 16 bits of significant data; long-word results return 32 bits.

A successful read operation returns data bit 16 cleared. If a bus or address error is encountered, the returned data is $10001.

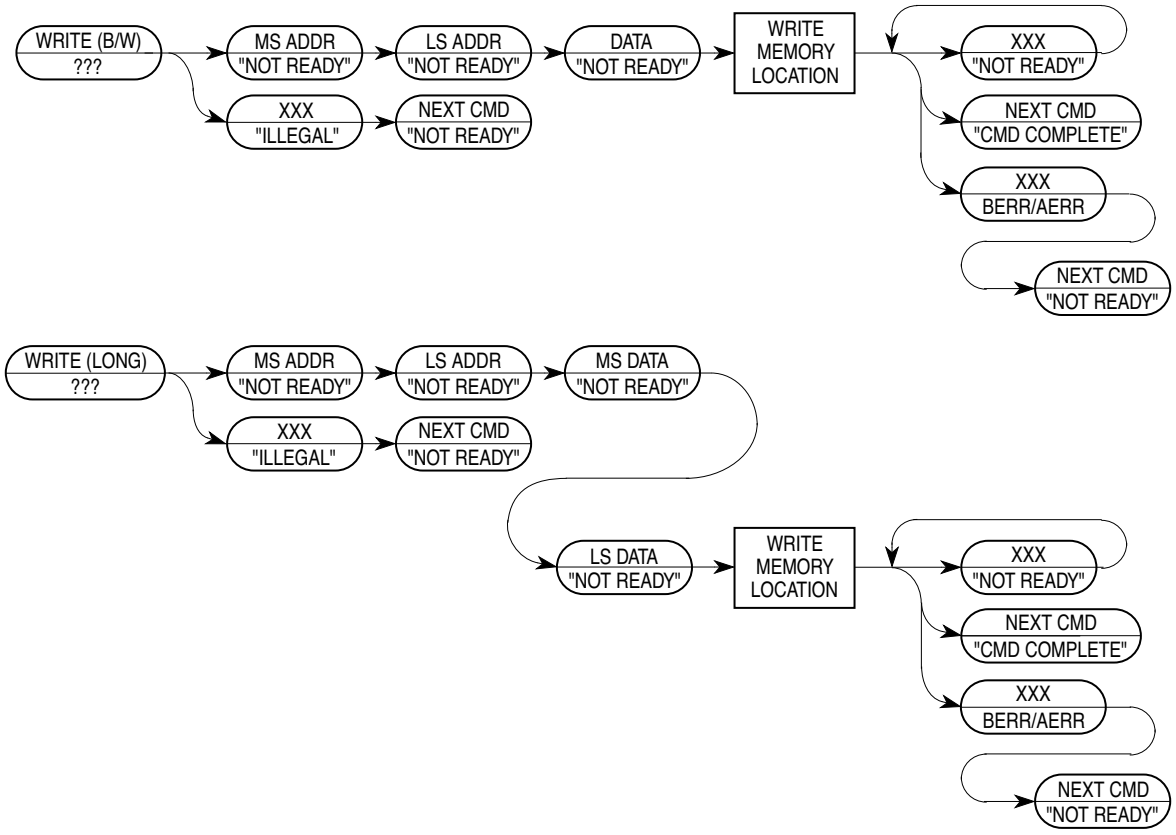
7.2.8.9 Write Memory Location (WRITE)

Write the operand data to the memory location specified by the long-word address. The destination function code (DFC) register determines the address space accessed. Only absolute addressing is supported. Valid data sizes include byte, word, and long word.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3			0
0	0	0	1	1	0	0	0	OP SIZE		0	0	0	0	0	0

Command Sequence:



Operand Data:

Two operands are required for this instruction. The first operand is a long-word absolute address that specifies a location to which the operand data is to be written. The second operand is the data. Byte data is transmitted as a 16-bit word, justified in the least significant byte. 16- and 32-bit operands are transmitted as 16 and 32 bits, respectively.

Result Data:

Successful write operations return a status of $0FFFF. Bus or address errors on the write cycle are indicated by the assertion of bit 16 in the status message and by a data pattern of $0001.

7.2.8.10 Dump Memory Block (DUMP)

DUMP is used in conjunction with the READ command to dump large blocks of memory. An initial READ is executed to set up the starting address of the block and to retrieve the first result. Subsequent operands are retrieved with the DUMP command. The initial address is incremented by the operand size (1, 2, or 4) and saved in a temporary register. Subsequent DUMP commands use this address, increment it by the current operand size, and store the updated address back in the temporary register.

NOTE

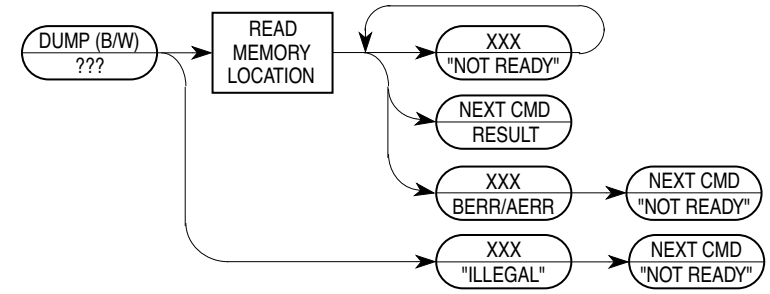
The DUMP command does not check for a valid address in the temporary register — DUMP is a valid command only when preceded by another DUMP or by a READ command. Otherwise, the results are undefined. The NOP command can be used for inter-command padding without corrupting the address pointer.

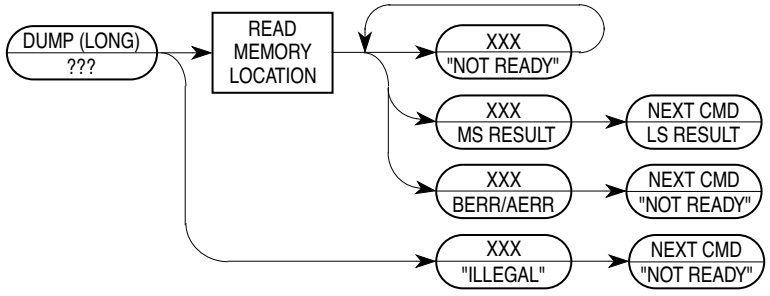
The size field is examined each time a DUMP command is given, allowing the operand size to be altered dynamically.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3			0
0	0	0	1	1	1	0	1	OP SIZE		0	0	0	0	0	0

Command Sequence:





Operand Data:

None

Result Data:

Requested data is returned as either a word or long word. Byte data is returned in the least significant byte of a word result. Word results return 16 bits of significant data; long-word results return 32 bits. Status of the read operation is returned as in the READ command: $0xxxx for success, $10001 for bus or address errors.

7.2.8.11 Fill Memory Block (FILL)

FILL is used in conjunction with the WRITE command to fill large blocks of memory. An initial WRITE is executed to set up the starting address of the block and to supply the first operand. Subsequent operands are written with the FILL command. The initial address is incremented by the operand size (1, 2, or 4) and is saved in a temporary register. Subsequent FILL commands use this address, increment it by the current operand size, and store the updated address back in the temporary register.

NOTE

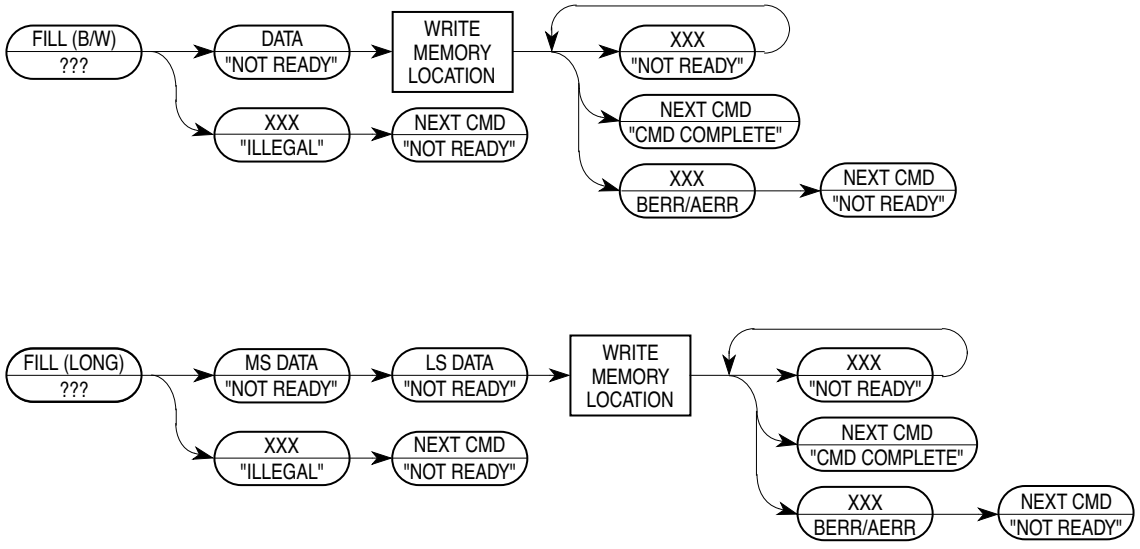
The FILL command does not check for a valid address in the temporary register — FILL is a valid command only when preceded by another FILL or by a WRITE command. Otherwise, the results are undefined. The NOP command can be used for inter-command padding without corrupting the address pointer.

The size field is examined each time a FILL command is given, allowing the operand size to be altered dynamically.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3			0
0	0	0	1	1	1	0	0	OP SIZE		0	0	0	0	0	0

Command Sequence:



Operand Data:

A single operand is data to be written to the memory location. Byte data is transmitted as a 16-bit word, justified in the least significant byte; 16- and 32-bit operands are transmitted as 16 and 32 bits, respectively.

Result Data:

Status is returned as in the WRITE command: $0FFFF for a successful operation and $10001 for a bus or address error during write.

7.2.8.12 Resume Execution (GO)

The pipeline is flushed and refilled before normal instruction execution is resumed. Prefetching begins at the return PC and current privilege level. If either the PC or SR is altered during BDM, the updated value of these registers is used when prefetching commences.

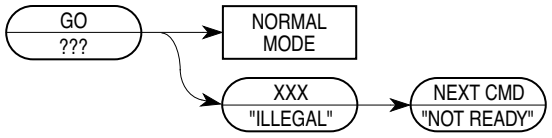
NOTE

The processor exits BDM when a bus error or address error occurs on the first instruction prefetch from the new PC — the error is trapped as a normal mode exception. The stacked value of the current PC may not be valid in this case, depending on the state of the machine prior to entering BDM. For address error, the PC does not reflect the true return PC. Instead, the stacked fault address is the (odd) return PC.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0

Command Sequence:



Operand Data:

None

Result Data:

None

7.2.8.13 Call User Code (CALL)

This instruction provides a convenient way to patch user code. The return PC is stacked at the location pointed to by the current SP. The stacked PC serves as a return address to be restored by the RTS command that terminates the patch routine. After stacking is complete, the 32-bit operand data is loaded into the PC. The pipeline is flushed and refilled from the location pointed to by the new PC. BDM is exited, and normal mode instruction execution begins.

NOTE

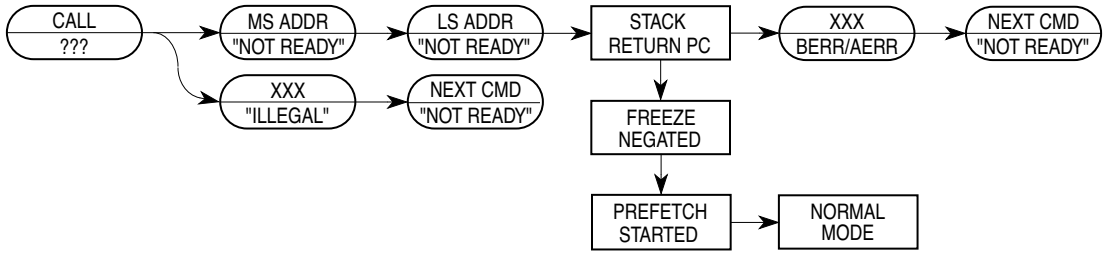
If a bus error or address error occurs during return address stacking, the CPU returns an error status via the serial interface and remains in BDM.

If a bus error or address error occurs on the first instruction prefetch from the new PC, the processor exits BDM and the error is trapped as a normal mode exception. The stacked value of the current PC may not be valid in this case, depending on the state of the machine prior to entering BDM. For address error, the PC does not reflect the true return PC. Instead, the stacked fault address is the (odd) return PC.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Command Sequence:



Operand Data:

The 32-bit operand data is the starting location of the patch routine, which is the initial PC upon exiting BDM.

Result Data:

None

As an example, consider the following code segment. It is supposed to output a character to an asynchronous communications interface adaptor — note that the routine fails to check the transmit data register empty (TDRE) flag.

```
CHKSTAT:    MOVE.B      ACIAS,D0          Move ACIA status to D0
            BEQ.B       CHKSTAT           Loop till condition true
            MOVE.B      DATA,ACIAD        Output data
            •
            •
            •
MISSING:    ANDI.B      #2,D0             Check for TDRE
            RTS                           Return to in-line code
```

BDM and the CALL command can be used to patch the code as follows:

1. Breakpoint user program at CHKSTAT
2. Enter BDM
3. Execute CALL command to MISSING
4. Exit BDM
5. Execute MISSING code
6. Return to user program.

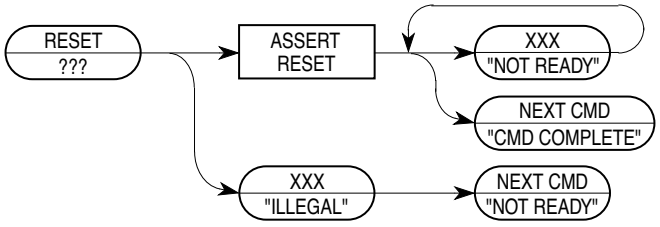
7.2.8.14 Reset Peripherals (RST)

RST asserts RESET for 512 clock cycles. The CPU is **not** reset by this command. This command is synonymous with the CPU RESET instruction.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0

Command Sequence:



Operand Data:

None

Result Data:

The “command complete” response ($0FFFF) is loaded into the serial shifter after negation of RESET.

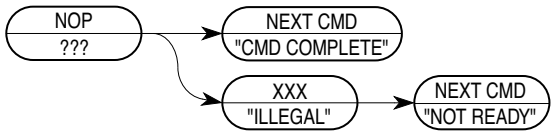
7.2.8.15 No Operation (NOP)

NOP performs no operation and may be used as a null command where required.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Command Sequence:



Operand Data:

None

Result Data:

The “command complete” response ($0FFFF) is returned during the next shift operation.

7.2.8.16 Future Commands

Unassigned command opcodes are reserved by Motorola for future expansion. All unused formats within any revision level will perform a NOP and return the ILLEGAL command response.

7.3 Deterministic Opcode Tracking

The CPU32 utilizes deterministic opcode tracking to trace program execution. Two signals, IPIPE and IFETCH, provide all the information required to analyze the operation of the instruction pipeline.

7.3.1 Instruction Fetch ($\overline{\text{IFETCH}}$)

$\overline{\text{IFETCH}}$ indicates which bus cycles are accessing data to fill the instruction pipeline. $\overline{\text{IFETCH}}$ is pulse-width modulated to multiplex two indications on a single pin. Asserted for a single clock cycle, $\overline{\text{IFETCH}}$ indicates that the data from the current bus cycle is to be routed to the instruction pipeline. $\overline{\text{IFETCH}}$ held low for two clock cycles indicates that the instruction pipeline has been flushed. The data from the bus cycle is used to begin filling the empty pipeline. Both user and supervisor mode fetches are signaled by $\overline{\text{IFETCH}}$.

Proper tracking of bus cycles via the $\overline{\text{IFETCH}}$ signal on a fast bus requires a simple state machine. On a two-clock bus, $\overline{\text{IFETCH}}$ may signal a pipeline flush with associated prefetch followed immediately by a second prefetch. That is, $\overline{\text{IFETCH}}$ remains asserted for three clocks, two clocks indicating the flush/fetch and a third clock signaling the second fetch. These two operations are easily discerned if the tracking logic samples $\overline{\text{IFETCH}}$ on the two rising edges of CLKOUT, which follow the address strobe (data strobe during show cycles) falling edge. Three-clock and slower bus cycles allow time for negation of the signal between consecutive indications and do not experience this operation.

7.3.2 Instruction Pipe ($\overline{\text{IPIPE}}$)

The internal instruction pipeline can be modeled as a three-stage FIFO (see **Figure 7-11**). Stage A is an input buffer — data can be used out of the stages B and C. $\overline{\text{IPIPE}}$ signals advances of instructions in the pipeline.

Instruction register A (IRA) holds incoming words as they are prefetched. No decoding takes place in the buffer. Instruction register B (IRB) provides initial decoding of the opcode and decoding of extension words —it is a source of immediate data. Instruction register C (IRC) supplies residual opcode decoding during instruction execution.

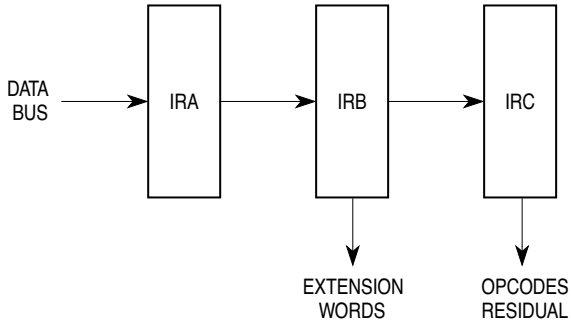


Figure 7-11 Functional Model of Instruction Pipeline

Assertion of $\overline{\text{IPIPE}}$ for a single clock cycle indicates the use of data from IRB. Regardless of the presence of valid data in IRA, the contents of IRB are invalidated when $\overline{\text{IPIPE}}$ is asserted. If IRA contains valid data, the data is copied into IRB (IRA → IRB), and the IRB stage is revalidated.

Assertion of $\overline{\text{IPIPE}}$ for two clock cycles indicates the start of a new instruction and subsequent replacement of data in IRC. This action causes a full advance of the pipeline (IRB → IRC and IRA → IRB). IRA is refilled during the next instruction fetch bus cycle.

Data loaded into IRA propagates automatically through subsequent empty pipeline stages. Signals that show the progress of instructions through IRB and IRC are necessary to accurately monitor pipeline operation. These signals are provided by IRA and IRB validity bits. When a pipeline advance occurs, the validity bit of the stage being loaded is set and the validity bit of the stage supplying the data is negated.

Because instruction execution is not timed to bus activity, $\overline{\text{IPIPE}}$ is synchronized with the system clock and not the bus. **Figure 7-12** illustrates the timing in relation to the system clock.

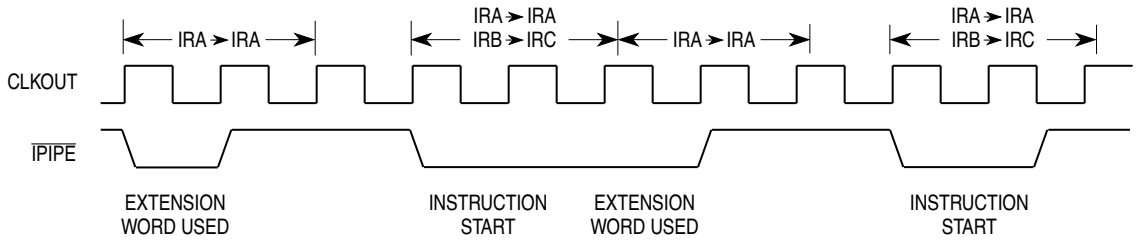


Figure 7-12 Instruction Pipeline Timing Diagram

$\overline{\text{IPIPE}}$ should be sampled on the falling edge of the clock.

The assertion of $\overline{\text{IPIPE}}$ for a single cycle after one or more cycles of negation indicates use of the data in IRB (advance of IRA into IRB). Assertion for two clock cycles indicates that a new instruction has started (both IRA → IRB and IRB → IRC transfers

have occurred). Loading IRC always indicates that an instruction is beginning execution — the opcode is loaded into IRC by the transfer.

In some cases, instructions using immediate addressing begin executing and initiate a second pipeline advance at the same time. $\overline{\text{IPIPE}}$ will not be negated between the two indications, which implies the need for a state machine to track the state of $\overline{\text{IPIPE}}$. The state machine can be resynchronized during periods of inactivity on the signal.

7.3.3 Opcode Tracking during Loop Mode

$\overline{\text{IPIPE}}$ and $\overline{\text{IFETCH}}$ continue to work normally during loop mode. $\overline{\text{IFETCH}}$ indicates all instruction fetches up through the point that data begins recirculating within the instruction pipeline. $\overline{\text{IPIPE}}$ continues to signal the start of instructions and the use of extension words even though data is being recirculated internally. $\overline{\text{IFETCH}}$ returns to normal operation with the first fetch after exiting loop mode.

SECTION 8 INSTRUCTION EXECUTION TIMING

This section describes the instruction execution timing of the CPU32. External clock cycles are used to provide accurate execution and operation timing guidelines, but not exact timing for every possible circumstance. This approach is used because exact execution time for an instruction or operation depends on concurrency of independently scheduled resources, on memory speeds, and on other variables.

An assembly language programmer or compiler writer can use the information in this section to predict the performance of the CPU32. Additionally, timing for exception processing is included so that designers of multitasking or real-time systems can predict task-switch overhead, maximum interrupt latency, and similar timing parameters. Instruction timing is given in clock cycles to eliminate clock frequency dependency.

8.1 Resource Scheduling

The CPU32 contains several independently scheduled resources. The organization of these resources within the CPU32 is shown in **Figure 8–1**. Some variation in instruction execution timing results from concurrent resource utilization. Because resource scheduling is not directly related to instruction boundaries, it is impossible to make an accurate prediction of the time required to complete an instruction without knowing the entire context within which the instruction is executing.

8.1.1 Microsequencer

The microsequencer either executes microinstructions or awaits completion of accesses necessary to continue microcode execution. The microsequencer supervises the bus controller, instruction execution, and internal processor operations such as calculation of effective address and setting of condition codes. It also initiates instruction word prefetches after a change of flow and controls validation of instruction words in the instruction pipeline.

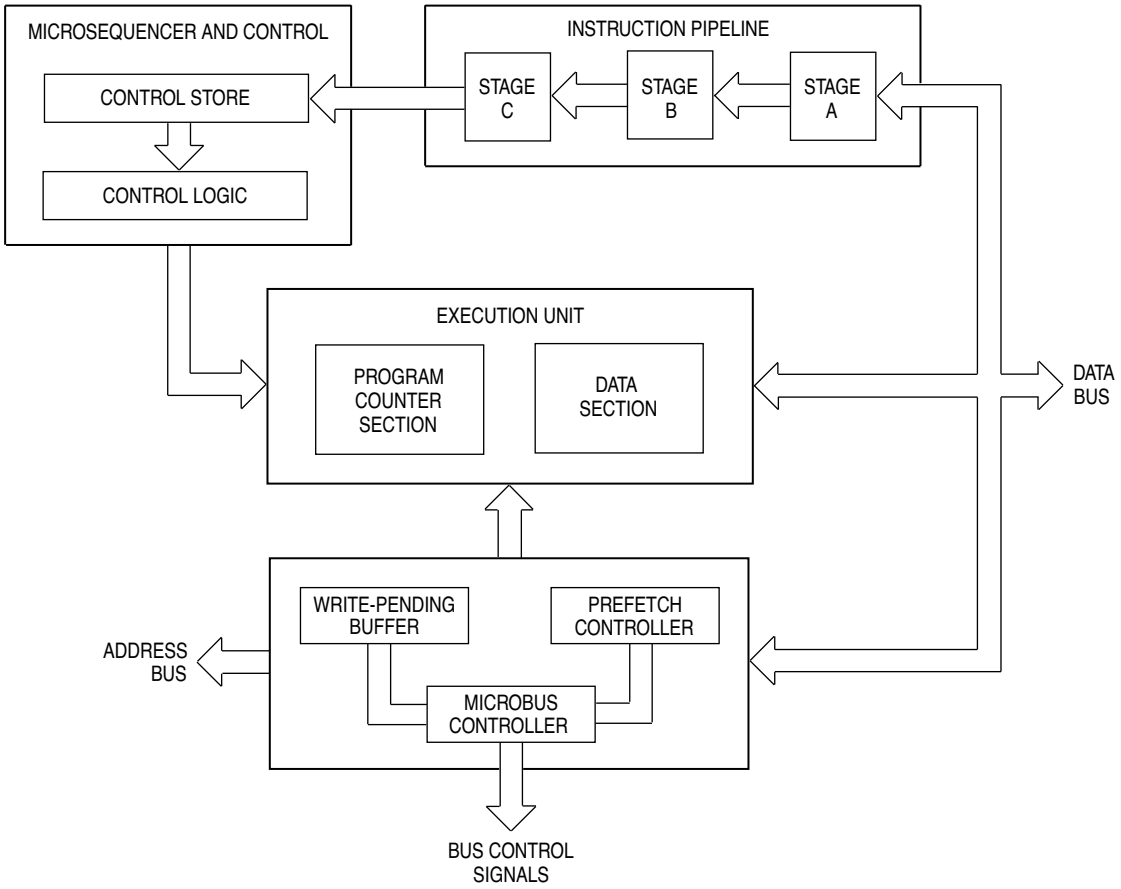


Figure 8–1 Block Diagram of Independent Resources

8.1.2 Instruction Pipeline

The CPU32 contains a two-word instruction pipeline where instruction opcodes are decoded. Each stage of the pipeline is initially filled under microsequencer control and subsequently refilled by the prefetch controller as it empties.

Stage A of the instruction pipeline is a buffer. Prefetches completed on the bus before stage B empties are temporarily stored in this buffer. Instruction words (instruction operation words and all extension words) are decoded at stage B. Residual decoding and execution take place in stage C.

Each pipeline stage has an associated status bit that shows whether the word in that stage was loaded with data from a bus cycle that terminated abnormally.

8.1.3 Bus Controller Resources

The bus controller consists of the instruction prefetch controller, the write-pending buffer, and the microbus controller. These three resources transact all reads, writes, and instruction prefetches required for instruction execution.

The bus controller and microsequencer operate concurrently. The bus controller can perform a read or write, or schedule a prefetch, while the microsequencer controls effective address calculation or sets condition codes.

The microsequencer can also request a bus cycle that the bus controller cannot perform immediately. When this happens, the bus cycle is queued, and the bus controller runs the cycle when the current cycle is complete.

8.1.3.1 Prefetch Controller

The instruction prefetch controller receives an initial request from the microsequencer to initiate prefetching at a given address. Subsequent prefetches are initiated by the prefetch controller whenever a pipeline stage is invalidated, either through instruction completion or through use of extension words. Prefetch occurs as soon as the bus is free of operand accesses previously requested by the microsequencer. Additional state information permits the controller to inhibit prefetch requests when a change in instruction flow (e.g. a jump or branch instruction) is anticipated.

In a typical program, 10 to 25 percent of the instructions causes a change of flow. Each time a change occurs, the instruction pipeline must be flushed and refilled from the new instruction stream. If instruction prefetches, rather than operand accesses, were given priority, many instruction words would be flushed unused, and necessary operand cycles would be delayed. To maximize available bus bandwidth, the CPU32 will schedule a prefetch only when the next instruction is not a change-of-flow instruction, and when there is room in the pipeline for the prefetch.

8.1.3.2 Write-Pending Buffer

The CPU32 incorporates a single-operand write-pending buffer. The buffer permits the microsequencer to continue execution after a request for a write cycle is queued in the bus controller. The time needed for a write at the end of an instruction can overlap the head cycle time for the following instruction, and thus reduce overall execution time. Interlocks prevent the microsequencer from overwriting the buffer.

8.1.3.3 Microbus Controller

The microbus controller performs bus cycles issued by the microsequencer. Operand accesses always have priority over instruction prefetches. Word and byte operands are accessed in a single CPU-initiated bus cycle, although the external bus interface may be required to initiate a second cycle when a word operand is sent to a byte-sized external port. Long operands are accessed in two bus cycles, most significant word first.

The instruction pipeline is capable of recognizing instructions that cause a change of flow. It informs the bus controller when a change of flow is imminent, and the bus controller refrains from starting prefetches that would be discarded due to the change of flow.

8.1.4 Instruction Execution Overlap

Overlap is the time, measured in clock cycles, that an instruction executes concurrently with the previous instruction. As shown in Figure 8-2, portions of instructions A and B execute simultaneously, so that total execution time is reduced. Because portions of instructions B and C also overlap, overall execution time for all three instructions is also reduced.

Each instruction contributes to the total overlap time. The portion of execution time at the end of instruction A that can overlap the beginning of instruction B is called the tail of instruction A. The portion of execution time at the beginning of instruction B that can overlap the end of instruction A is called the head of instruction B. The total overlap time between instructions A and B is the smaller tail of A and the head of B.

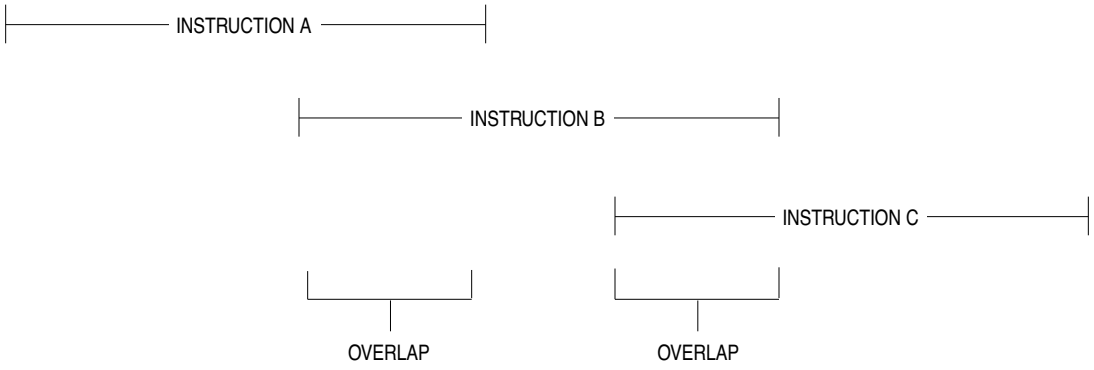


Figure 8-2 Simultaneous Instruction Execution

The execution time attributed to instructions A, B, and C after considering the overlap is illustrated in Figure 8-3. The overlap time is attributed to the execution time of the completing instruction. The following equation shows the method for calculating the overlap time:

$$\text{Overlap} = \min (\text{Tail}_N, \text{Head}_{N+1})$$

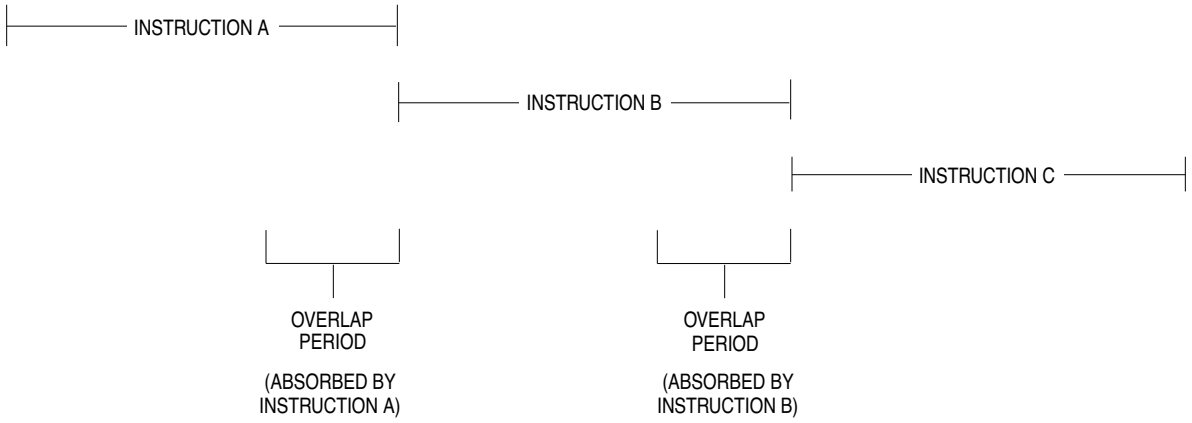


Figure 8–3 Attributed Instruction Times

8.1.5 Effects of Wait States

The CPU32 access time for on-chip memory and peripherals is two clocks. While two-clock external accesses are possible when the bus is operated in a synchronous mode, a typical external memory speed is three or more clocks.

All instruction times listed in this section are for word access only (unless an explicit exception is given), and are based on the assumption that both instruction fetches and operand cycles are to a two-clock memory. Any time a long access is made, time for the additional bus cycle(s) must be added to the overall execution time. Wait states due to slow external memory must be added to the access time for each bus cycle.

A typical application has a mixture of bus speeds —program execution from an off-chip ROM, accesses to on-chip peripherals, storage of variables in slow off-chip RAM, and accesses to external peripherals with speeds ranging from moderate to very slow. To arrive at an accurate instruction time calculation, each bus access must be individually considered. Many instructions have a head cycle count, which can overlap the cycles of an operand fetch to slower memory started by a previous instruction. In these cases, an increase in access time has no effect on the total execution time of the pair of instructions.

To trace instruction execution time by monitoring the external bus, note that the order of operand accesses for a particular instruction sequence is always the same — provided bus speed is unchanged, the interleaving of instruction prefetches with operands within each sequence is identical.

8.1.6 Instruction Execution Time Calculation

The overall execution time for an instruction depends on the amount of overlap with previous and following instructions. In order to calculate an instruction time estimate, the entire code sequence must be analyzed. To derive the actual instruction execution times for an instruction sequence, the instruction times listed in the tables must be adjusted to account for overlap.

The formula for this calculation is:

$$C_1 - \min(T_1, H_2) + C_2 - \min(T_2, H_3) + C_3 - \min(T_3, H_4) + ...$$

where:

C_N is the number of cycles listed for instruction N
H_N is the head time for instruction N
T_N is the tail time for instruction N
$\min(T_N, H_M)$ is the minimum of parameters T_N and H_M

The number of cycles for the instruction (C_N above), can include one or two effective address calculations in addition to the raw number in the cycles column. In these cases, calculate overall instruction time as if it were for multiple instructions, using the following equation:

$$\langle CEA \rangle - \min (T_{EA}, H_{OP}) + C_{OP}$$

where:

$\langle CEA \rangle$ is the instruction's effective address time
C_{OP} is the instruction's operation time
H_{OP} is the instruction operation's head time
T_{EA} is the effective address's tail time
min (T_N, H_M) is the minimum of parameters T_N and H_M

The overall head for the instruction is the head for the effective address, and the overall tail for the instruction is the tail for the operation. Therefore, the actual equation for execution time becomes:

$$C_{OP1} - \min (T_{OP1}, H_{EA2}) + \langle CEA \rangle_2 - \min (T_{EA2}, H_{OP2}) +$$

$$C_{OP2} - \min (T_{OP2}, H_{EA3}) + ...$$

Every instruction must prefetch to replace itself in the instruction pipe. Usually, these prefetches occur during or after an instruction. A prefetch is permitted to begin in the first clock of any indexed effective addressing mode operation.

Additionally, a prefetch for an instruction is permitted to begin two clocks before the end of an instruction, provided the bus is not being used. If the bus is being used, then the prefetch occurs at the next available time, when the bus would otherwise be idle.

8.1.7 Effects of Negative Tails

When the CPU32 changes instruction flow, the instruction decode pipeline must begin refilling before instruction execution can resume. Refilling forces a two-clock idle period at the end of the change of flow instruction. This idle period can be used to prefetch an additional word on the new instruction path.

Because of the stipulation that each instruction must prefetch to replace itself, the concept of negative tails has been introduced to account for these free clocks on the bus.

On a two-clock bus, it is not necessary to adjust instruction timing to account for the potential extra prefetch. The cycle times of the microsequencer and bus are matched and no additional benefit or penalty is obtained. In the instruction execution time equations, a zero should be used instead of a negative number.

Negative tails are used to adjust for slower fetches on slower buses. Normally, increasing the length of prefetch bus cycles directly affects the cycle count and tail values found in the tables.

In the following equations, negative tail values are used to negate the effects of a slower bus. The equations are generalized, however, so that they may be used on any speed bus with any tail value.

```
NEW_TAIL = OLD_TAIL + (NEW_CLOCK – 2)
IF ((NEW_CLOCK – 4) >0) THEN
    NEW_CYCLE = OLD_CYCLE + (NEW_CLOCK -2) + (NEW_CLOCK – 4)
ELSE
    NEW_CYCLE = OLD_CYCLE + (NEW _CLOCK – 2)
```

where:

NEW_TAIL/NEW_CYCLE is the adjusted tail/cycle at the slower speed
OLD_TAIL/OLD_CYCLE is the value listed in the instruction timing tables
NEW_CLOCK is the number of clocks per cycle at the slower speed

Note that many instructions listed as having negative tails are change of flow instructions, and that the bus speed used in the calculation is that of the new instruction stream.

8.2 Instruction Stream Timing Examples

The following programming examples provide a detailed examination of timing effects. In all examples, memory access is either from internal two-clock memory or from external synchronous memory, the bus is idle, and the instruction pipeline is full at start.

8.2.1 Timing Example 1: Execution Overlap

Figure 8-4 illustrates execution overlap caused by the bus controller's completion of bus cycles while the sequencer is calculating the next effective address. One clock is saved between instructions, as that is the minimum time of the individual head and tail numbers.

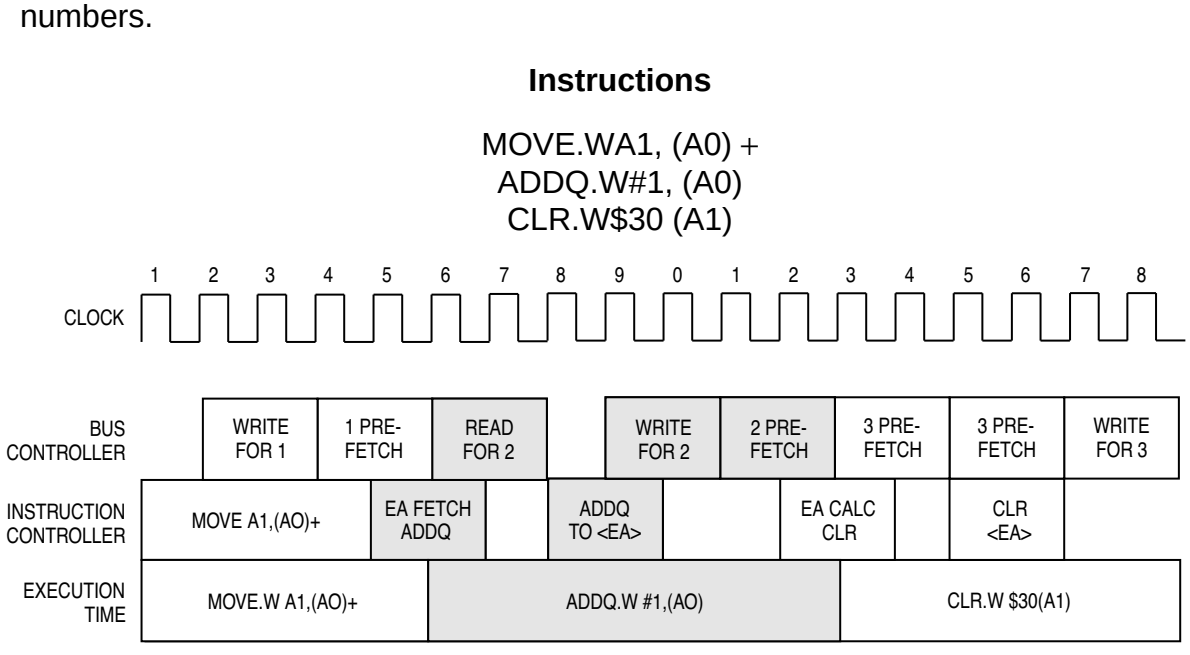


Figure 8-4 Example 1 — Instruction Stream

8.2.2 Timing Example 2: Branch Instructions

Example 2 shows what happens when a branch instruction is executed, in both the taken and not-taken cases. (Refer to Figures 8-5 and 8-6). The instruction stream is for a simple limit check with the variable already in a data register.

Instructions

MOVEQ#7, D1
CMP.LD1, D0
BLE.BNEXT
MOVE.LD1, (A0)

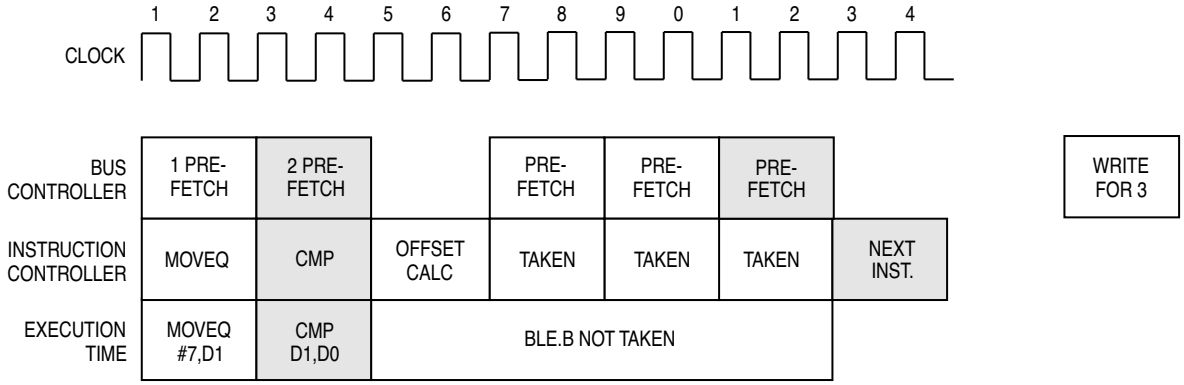


Figure 8-5 Example 2 — Branch Taken

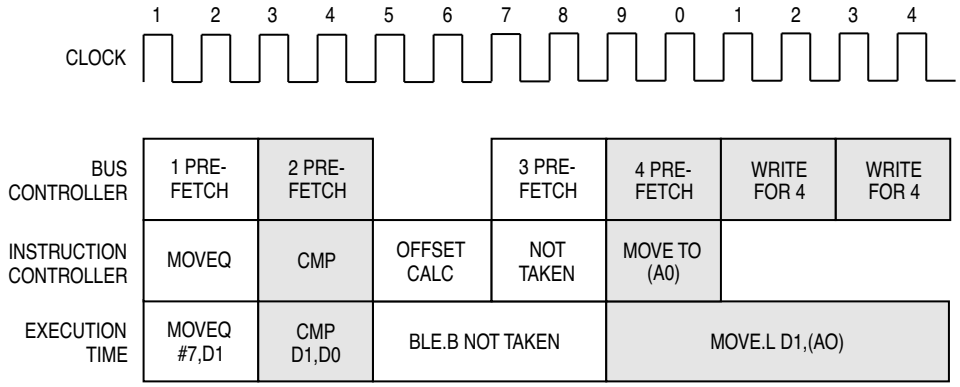


Figure 8-6 Example 2 — Branch Not Taken

8.2.3 Timing Example 3: Negative Tails

This example (Figure 8-7) shows how to use negative tail figures for branches and other change-of-flow instructions. In this example, bus speed is assumed to be four clocks per access. Instruction three is at the branch destination.

Instructions

MOVEQ#7, D1
BRA.WFARAWAY
MOVE.LD1, D0

Although the CPU32 has a two-word instruction pipeline, internal delay causes minimum branch instruction time to be three bus cycles. The negative tail is a reminder that an extra two clocks are available for prefetching a third word on a fast bus — on a slower bus, there is no extra time for the third word.

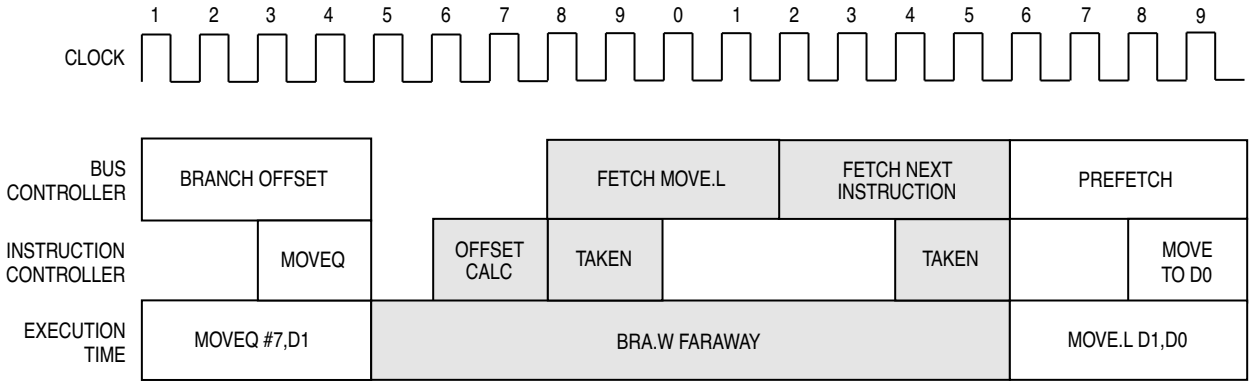


Figure 8-7 Example 3 — Branch Negative Tail

Example 3 illustrates three different aspects of instruction time calculation:

The branch instruction does not attempt to prefetch beyond the minimum number of words needed for itself.

The negative tail allows execution to begin sooner than would a three-word pipeline.

There is a one-clock delay due to late arrival of the displacement at the CPU.

Only changes of flow require negative tail calculation, but the concept can be generalized to any instruction — only two words are required to be in the pipeline, but up to three words may be present. When there is an opportunity for an extra prefetch, it is made. A prefetch to replace an instruction can begin ahead of the instruction, resulting in a faster processor.

8.3 Instruction Timing Tables

The following assumptions apply to the times shown in the tables in this section:

- A 16-bit data bus is used for all memory accesses.
- Memory access times are based on two clock bus cycles with no wait states.
- The instruction pipeline is full at the beginning of the instruction and is refilled by the end of the instruction.

Three values are listed for each instruction and addressing mode:

Head — The number of cycles available at the beginning of an instruction to complete a previous instruction write or to perform a prefetch.

Tail — The number of cycles an instruction uses to complete a write.

Cycles — Four numbers per entry, three contained in parentheses.

The outer number is the minimum number of cycles required for the instruction to complete.

Numbers within the parentheses represent the number of bus accesses performed by the instruction.

The first number is the number of operand read accesses performed by the instruction.

The second number is the number of instruction fetches performed by the instruction, including all prefetches that keep the instruction and the instruction pipeline filled.

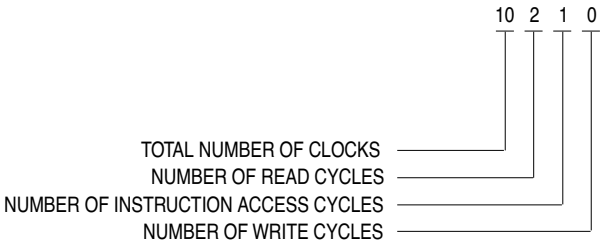
The third number is the number of write accesses performed by the instruction.

As an example, consider an ADD.L (12, A3, D7.W ∗ 4), D2 instruction.

Section **8.3.5 Arithmetic/Logic Instructions** shows that the instruction has a head = 0, a tail = 0, and cycles = 2 (0/1/0). However, in indexed, address register Indirect addressing mode, additional time is required to fetch the effective address.

Section **8.3.1 Fetch Effective Address** gives addressing mode data. For (d_8, An, Xn.Sz ∗ Scale), head = 4, tail = 2, cycles = 8 (2/1/0). Because this example is for a long access and the FEA table lists data for word accesses, add two clocks to the tail and to the number of cycles (“X” table notation), to obtain head = 4, tail = 4, cycles = 10 (2/1/0).

Assuming that no trailing write exists from the previous instruction, effective address calculation requires six clocks. Replacement fetch for the effective address occurs during these six clocks, leaving a head of four. If there is no time in the head to perform a prefetch, due to a previous trailing write, then additional time to do the prefetches must be allotted in the middle of the instruction or after the tail.



The total number of bus-activity clocks is:

(2 Reads × 2 Clocks/Read) + (1 Instruction Access × 2 Clocks/Access) +
(0 Writes × 2 Clocks/Write) = 6 Clocks of Bus Activity

The number of internal clocks (not overlapped by bus activity) is:

10 Clocks Total – 6 Clocks Bus Activity = 4 Internal Clocks

Memory read requires two bus cycles at two clocks each. This read time, implied in the tail figure for the effective address, cannot be overlapped with the instruction because the instruction has a head of zero.

An additional two clocks are required for the ADD instruction itself.

The total is 6 + 4 + 2 = 12 clocks. If bus cycles take more time (i.e., the memory is off-chip), add an appropriate number of clocks to each memory access.

The instruction sequence MOVE.L D0, (A0) followed by LSL.L #7, D2 provides an example of overlapped execution. The MOVE instruction has a head of zero and a tail of four, because it is a long write. The LSL instruction has a head of four. The trailing write from the MOVE overlaps the LSL head completely. Thus, the two-instruction sequence has a head of zero and a tail of zero, and a total execution of eight rather than 12 clocks.

General observations regarding calculation of execution time are as follows:

Any time the number of bus cycles is listed as “X”, substitute a value of one for byte and word cycles and a value of two for long cycles. For long bus cycles, usually add a value of two to the tail.

The time calculated for an instruction on a three-clock (or longer) bus is usually longer than the actual execution time. All times shown are for two-clock bus cycles.

If the previous instruction has a negative tail, then a prefetch for the current instruction can begin during the execution of that previous instruction.

Certain instructions requiring an immediate extension word (immediate word effective address, absolute word effective address, address register indirect with displacement effective address, conditional branches with word offsets, bit operations, LPSTOP, TBL, MOVEM, MOVEC, MOVES, MOVEP, MUL.L, DIV.L, CHK2, CMP2, and DBcc) are not permitted to begin until the extension word has been in the instruction pipeline for at least one cycle. This does not apply to long offsets or displacements.

8.3.1 Fetch Effective Address

The fetch effective address table indicates the number of clock periods needed for the processor to calculate and fetch the specified effective address. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction	Head	Tail	Cycles	Notes
Dn	–	–	0(0/0/0)	–
An	–	–	0(0/0/0)	–
(An)	1	1	3(X/0/0)	1
(An)+	1	1	3(X/0/0)	1
–(An)	2	2	4(X/0/0)	1
(d_{16},An) or (d_{16},PC)	1	3	5(X/1/0)	1,3
(xxx).W	1	3	5(X/1/0)	1
(xxx).L	1	5	7(X/2/0)	1
#⟨data⟩.B	1	1	3(0/1/0)	1
#⟨data⟩.W	1	1	3(0/1/0)	1
#⟨data⟩.L	1	3	5(0/2/0)	1
(d_8,An,Xn.Sz∗Sc) or (d_8,PC,Xn.Sz∗Sc)	4	2	8(X/1/0)	1,2,3,4
(0) (All Suppressed)	2	2	6(X/1/0)	1,4
(d_{16})	1	3	7(X/2/0)	1,4
(d_{32})	1	5	9(X/3/0)	1,4
(An)	1	1	5(X/1/0)	1,2,4
(Xm.Sz∗Sc)	4	2	8(X/1/0)	1,2,4
(An,Xm.Sz∗Sc)	4	2	8(X/1/0)	1,2,3,4
(d_{16},An) or (d_{16},PC)	1	3	7(X/2/0)	1,3,4
(d_{32},An) or (d_{32},PC)	1	5	9(X/3/0)	1,3,4
(d_{16},An,Xm) or (d_{16},PC,Xm)	2	2	8(X/2/0)	1,3,4
(d_{32},An,Xm) or (d_{32},PC,Xm)	1	3	9(X/3/0)	1,3,4
(d_{16},An,Xm.Sz∗Sc) or (d_{16},PC,Xm.Sz∗Sc)	2	2	8(X/2/0)	1,2,3,4
(d_{32},An,Xm.Sz∗Sc) or (d_{32},PC,Xm.Sz∗Sc)	1	3	9(X/3/0)	1,2,3,4
X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.				

NOTES:

1. The read of the effective address and replacement fetches overlap the head of the operation by the amount specified in the tail.
2. Size and scale of the index register do not affect execution time.
3. The program counter may be substituted for the base address register An.
4. When adjusting the prefetch time for slower buses, extra clocks may be subtracted from the head until the head reaches zero, at which time additional clocks must be added to both the tail and cycle counts.

8.3.2 Calculate Effective Address

The calculate effective address table indicates the number of clock periods needed for the processor to calculate a specified effective address. The timing is equivalent to fetch effective address except there is no read cycle. The tail and cycle time are reduced by the amount of time the read would occupy. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction	Head	Tail	Cycles	Notes
Dn	–	–	0(0/0/0)	–
An	–	–	0(0/0/0)	–
(An)	1	0	2(0/0/0)	–
(An)+	1	0	2(0/0/0)	–
–(An)	2	0	2(0/0/0)	–
(d_{16},An) or (d_{16},PC)	1	1	3(0/1/0)	1,3
(xxx).W	1	1	3(0/1/0)	1
(xxx).L	1	3	5(0/2/0)	1
(d_8,An,Xn.Sz∗Sc) or (d_8,PC,Xn.Sz∗Sc)	4	0	6(0/1/0)	2,3,4
(0) (All Suppressed)	2	0	4(0/1/0)	4
(d_{16})	1	1	5(0/2/0)	1,4
(d_{32})	1	3	7(0/3/0)	1,4
(An)	1	0	4(0/1/0)	4
(Xm.Sz∗Sc)	4	0	6(0/1/0)	2,4
(An,Xm.Sz∗Sc)	4	0	6(0/1/0)	2,4
(d_{16},An) or (d_{16},PC)	1	1	5(0/2/0)	1,3,4
(d_{32},An) or (d_{32},PC)	1	3	7(0/3/0)	1,3,4
(d_{16},An,Xm) or (d_{16},PC,Xm)	2	0	6(0/2/0)	3,4
(d_{32},An,Xm) or (d_{32},PC,Xm)	1	1	7(0/3/0)	1,3,4
(d_{16},An,Xm.Sz∗Sc) or (d_{16},PC,Xm.Sz∗Sc)	2	0	6(0/2/0)	2,3,4
(d_{32},An,Xm.Sz∗Sc) or (d_{32},PC,Xm.Sz∗Sc)	1	1	7(0/3/0)	1,2,3,4
X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.				

NOTES:

1. Replacement fetches overlap the head of the operation by the amount specified in the tail.
2. Size and scale of the index register do not affect execution time.
3. The program counter may be substituted for the base address register An.
4. When adjusting the prefetch time for slower buses, extra clocks may be subtracted from the head until the head reaches zero, at which time additional clocks must be added to both the tail and cycle counts.

8.3.3 MOVE Instruction

The MOVE instruction table indicates the number of clock periods needed for the processor to calculate the destination effective address and to perform a MOVE or MOVEA instruction. For entries with CEA or FEA, refer to the appropriate table to calculate that portion of the instruction time.

Destination effective addresses are divided by their formats (refer to **3.4.4 Effective Address Encoding Summary**). The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

When using this table, begin at the top and move downward. Use the first entry that matches both source and destination addressing modes.

Instruction	Head	Tail	Cycles
MOVE Rn, Rn	0	0	2(0/1/0)
MOVE ⟨FEA⟩, Rn	0	0	2(0/1/0)
MOVE Rn, (Am)	0	2	4(0/1/x)
MOVE Rn, (Am)+	1	1	5(0/1/x)
MOVE Rn, –(Am)	2	2	6(0/1/x)
MOVE Rn, ⟨CEA⟩	1	3	5(0/1/x)
MOVE ⟨FEA, (An)	2	2	6(0/1/x)
MOVE ⟨FEA⟩, (An)+	2	2	6(0/1/x)
MOVE ⟨FEA⟩, –(An)	2	2	6(0/1/x)
MOVE #, ⟨CEA⟩	2	2	6(0/1/x)*
MOVE ⟨CEA⟩, ⟨FEA⟩	2	2	6(0/1/x)
X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.			
*An # fetch effective address time must be added for this instruction: ⟨FEA⟩ +⟨CEA⟩ + ⟨OPER⟩			

NOTE: For instructions not explicitly listed, use the MOVE ⟨CEA⟩, ⟨FEA⟩ entry. The source effective address is calculated by the calculate effective address table, and the destination effective address is calculated by the fetch effective address table, even though the bus cycle is for the source effective address.

8.3.4 Special-Purpose MOVE Instruction

The special-purpose MOVE instruction table indicates the number of clock periods needed for the processor to fetch, calculate, and perform the special-purpose MOVE operation on control registers or a 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 numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
EXG	Rn, Rm	2	0	4(0/1/0)
MOVEC	Cr, Rn	10	0	14(0/2/0)
MOVEC	Rn, Cr	12	0	14-16(0/1/0)
MOVE	CCR, Dn	2	0	4(0/1/0)
MOVE	CCR, ⟨CEA⟩	0	2	4(0/1/1)
MOVE	Dn, CCR	2	0	4(0/1/0)
MOVE	⟨FEA⟩, CCR	0	0	4(0/1/0)
MOVE	SR, Dn	2	0	4(0/1/0)
MOVE	SR, ⟨CEA⟩	0	2	4(0/1/1)
MOVE	Dn, SR	4	–2	10(0/3/0)
MOVE	⟨FEA⟩, SR	0	–2	10(0/3/0)
MOVEM.W	⟨CEA⟩, RL	1	0	8 + n ∗ 4 (n + 1, 2, 0) [1]
MOVEM.W	RL, ⟨CEA⟩	1	0	8 + n ∗ 4 (0, 2, n) [1]
MOVEM.L	⟨CEA⟩, RL	1	0	12 + n ∗ 4(2n + 2, 2, 0)
MOVEM.L	RL, ⟨CEA⟩	1	2	10 + n • 4 (0, 2, 2n)
MOVEP.W	Dn, (d_{16}, An)	2	0	10(0/2/2)
MOVEP.W	(d_{16}, An), Dn	1	2	11(2/2/0)
MOVEP.L	Dn, (d_{16}, An)	2	0	14(0/2/4)
MOVEP.L	(d_{16}, An), Dn	1	2	19(4/2/0)
MOVES (Save)	⟨CEA⟩, Rn	1	1	3(0/1/0)
MOVES (Op)	⟨CEA⟩, Rn	7	1	11(X/1/0)
MOVES (Save)	Rn, ⟨CEA⟩	1	1	3(0/1/0)
MOVES (Op)	Rn, ⟨CEA⟩	9	2	12(0/1/X)
MOVE	USP, An	0	0	2(0/1/0)
MOVE	An, USP	0	0	2(0/1/0)
SWAP	Dn	4	0	6(0/1/0)

X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.

[1]Each bus cycle may take up to four clocks without increasing total execution time.

Cr = Control registers USP, VBR, SFC, and DFC

n = Number of registers to transfer

RL = Register List

< = Maximum time — certain data or mode combinations may execute faster.

NOTE: The MOVES instruction has an additional a save step which other instructions do not have. To calculate total the instruction time, calculate the Save, the effective address, and the Operation execution times, and combine in the order listed, using the equations given in **8.1.6 Instruction Execution Time Calculation**.

8.3.5 Arithmetic/Logic Instructions

The arithmetic/logic instruction table indicates the number of clock periods needed to perform the specified arithmetic/logical instruction using the specified addressing mode. Footnotes indicate when to account for the appropriate effective address times. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
ADD(A)	Rn, Rm	0	0	2(0/1/0)
ADD(A)	⟨FEA⟩, Rn	0	0	2(0/1/0)
ADD	Dn, ⟨FEA⟩	0	3	5(0/1/x)
AND	Dn, Dm	0	0	2(0/1/0)
AND	⟨FEA⟩, Dn	0	0	2(0/1/0)
AND	Dn, ⟨FEA⟩	0	3	5(0/1/x)
EOR	Dn, Dm	0	0	2(0/1/0)
EOR	Dn, ⟨FEA⟩	0	3	5(0/1/x)
OR	Dn, Dm	0	0	2(0/1/0)
OR	⟨FEA⟩, Dn	0	0	2(0/1/0)
OR	Dn, ⟨FEA⟩	0	3	5(0/1/x)
SUB(A)	Rn, Rm	0	0	2(0/1/0)
SUB(A)	⟨FEA⟩, Rn	0	0	2(0/1/0)
SUB	Dn, ⟨FEA⟩	0	3	5(0/1/x)
CMP(A)	Rn, Rm	0	0	2(0/1/0)
CMP(A)	⟨FEA⟩, Rn	0	0	2(0/1/0)
CMP2 (Save)*	⟨FEA⟩, Rn	1	1	3(0/1/0)
CMP2 (Op)	⟨FEA⟩, Rn	2	0	16 - 18(X/1/0)
MUL(S/U).W	⟨FEA⟩, Dn	0	0	26(0/1/0)
MUL(S/U).L (Save)*	⟨FEA⟩, Dn	1	1	3(0/1/0)
MUL(S/U).L (Op)	⟨FEA⟩, Dl	2	0	46 - 52(0/1/0)
MUL(S/U).L (Op)	⟨FEA⟩, Dn:Dl	2	0	46(0/1/0)
DIVU.W	⟨FEA⟩, Dn	0	0	32(0/1/0)
DIVS.W	⟨FEA⟩, Dn	0	0	42(0/1/0)
DIVU.L (Save)*	⟨FEA⟩, Dn	1	1	3(0/1/0)
DIVU.L (Op)	⟨FEA⟩, Dn	2	0	<46(0/1/0)
DIVS.L (Save)*	⟨FEA⟩, Dn	1	1	3(0/1/0)
DIVS.L (Op)	⟨FEA⟩, Dn	2	0	<62(0/1/0)
TBL(S/U)	Dn:Dm, Dp	26	0	28-30(0/2/0)
TBL(S/U) (Save)*	⟨CEA⟩, Dn	1	1	3(0/1/0)
TBL(S/U) (Op)	⟨CEA⟩, Dn	6	0	33-35(2X/1/0)
TBLSN	Dn:Dm, Dp	30	0	30-34(0/2/0)
TBLSN (Save)*	⟨CEA⟩, Dn	1	1	3(0/1/0)
TBLSN (Op)	⟨CEA⟩, Dn	6	0	35-39(2X/1/0)
TBLUN	Dn:Dm, Dp	30	0	34-40(0/2/0)
TBLUN (Save)*	⟨CEA⟩, Dn	1	1	3(0/1/0)
TBLUN (Op)	⟨CEA⟩, Dn	6	0	39-45(2X/1/0)
X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.				
< = Maximum time; certain data or mode combinations may execute faster.				
su = The execution time is identical for signed or unsigned operands.				

*These instructions have an additional save operation that other instructions do not have. To calculate total instruction time, calculate save, ⟨ea⟩, and operation execution times, then combine in the order shown, using equations in **8.1.6 Instruction Execution Time Calculation**. A save operation is not run for long word divide and multiply instructions when ⟨FEA⟩ = Dn,

8.3.6 Immediate Arithmetic/Logic Instructions

The immediate arithmetic/logic instruction table indicates the number of clock periods needed for the processor to fetch the source immediate data value and to perform the specified arithmetical/logical instruction using the specified addressing mode. Footnotes indicate when to account for the appropriate fetch effective or fetch immediate effective address times. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
MOVEQ	#, Dn	0	0	2(0/1/0)
ADDQ	#, Rn	0	0	2(0/1/0)
ADDQ	#, ⟨FEA⟩	0	3	5(0/1/x)
SUBQ	#, Rn	0	0	2(0/1/0)
SUBQ	#, ⟨FEA⟩	0	3	5(0/1/x)
ADDI	#, Rn	0	0	2(0/1/0)∗
ADDI	#, ⟨FEA⟩	0	3	5(0/1/x)∗
ANDI	#, Rn	0	0	2(0/1/0)∗
ANDI	#, ⟨FEA⟩	0	3	5(0/1/x)∗
EORI	#, Rn	0	0	2(0/1/0)∗
EORI	#, ⟨FEA⟩	0	3	5(0/1/x)∗
ORI	#, Rn	0	0	2(0/1/0)∗
ORI	#, ⟨FEA⟩	0	3	5(0/1/x)∗
SUBI	#, Rn	0	0	2(0/1/0)∗
SUBI	#, ⟨FEA⟩	0	3	5(0/1/x)∗
CMPI	#, Rn	0	0	2(0/1/0)∗
CMPI	#, ⟨FEA⟩	0	3	5(0/1/x)∗
X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.				
∗An # fetch effective address time must be added for this instruction: ⟨FEA⟩ +⟨FEA⟩ + ⟨OPER⟩.				

8.3.7 Binary-Coded Decimal and Extended Instructions

The binary-coded decimal and extended instruction table indicates the number of clock periods needed for the processor to perform the specified operation using the specified addressing mode. No additional tables are needed to calculate total effective execution time for these instructions. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
ABCD	Dn, Dm	2	0	4(0/1/0)
ABCD	–(An), –(Am)	2	2	12(2/1/1)
SBCD	Dn, Dm	2	0	4(0/1/0)
SBCD	–(An), –(Am)	2	2	12(2/1/1)
ADDX	Dn, Dm	0	0	2(0/1/0)
ADDX	–(An), –(Am)	2	2	10(2/1/1)
SUBX	Dn, Dm	0	0	2(0/1/0)
SUBX	–(An), –(Am)	2	2	10(2/1/1)
CMPM	(An)+, (Am)+	1	0	8(2/1/0)

8.3.8 Single Operand Instructions

The single operand instruction table indicates the number of clock periods needed for the processor to perform the specified operation using the specified addressing mode. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
CLR	Dn	0	0	2(0/1/0)
CLR	⟨CEA⟩	0	2	4(0/1/x)
NEG	Dn	0	0	2(0/1/0)
NEG	⟨FEA⟩	0	3	5(0/1/x)
NEGX	Dn	0	0	2(0/1/0)
NEGX	⟨FEA⟩	0	3	5(0/1/x)
NOT	Dn	0	0	2(0/1/0)
NOT	⟨FEA⟩	0	3	5(0/1/x)
EXT	Dn	0	0	2(0/1/0)
NBCD	Dn	2	0	4(0/1/0)
NBCD	⟨FEA⟩	0	2	6(0/1/1)
Scc	Dn	2	0	4(0/1/0)
Scc	⟨CEA⟩	2	2	6(0/1/1)
TAS	Dn	4	0	6(0/1/0)
TAS	⟨CEA⟩	1	0	10(0/1/1)
TST	⟨FEA⟩	0	0	2(0/1/0)
X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.				

8.3.9 Shift/Rotate Instructions

The shift/rotate instruction table indicates the number of clock periods needed for the processor to perform the specified operation on the given addressing mode. Footnotes indicate when to account for the appropriate effective address times. The number of bits shifted does not affect the execution time, unless noted. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles	Note
LSd	Dn, Dm	–2	0	(0/1/0)	1
LSd	#, Dm	4	0	6(0/1/0)	—
LSd	⟨FEA⟩	0	2	6(0/1/1)	—
ASd	Dn, Dm	–2	0	(0/1/0)	1
ASd	#, Dm	4	0	6(0/1/0)	—
ASd	⟨FEA⟩	0	2	6(0/1/1)	—
ROd	Dn, Dm	–2	0	(0/1/0)	1
ROd	#, Dm	4	0	6(0/1/0)	—
ROd	⟨FEA⟩	0	2	6(0/1/1)	—
ROXd	Dn, Dm	–2	0	(0/1/0)	2
ROXd	#, Dm	–2	0	(0/1/0)	3
ROXd	⟨FEA⟩	0	2	6(0/1/1)	—

NOTES:

1. Head and cycle times can be calculated as follows:
   Max (3 + (n/4) + mod(n,4) + mod (((n/4) + mod (n,4) + 1,2), 6)
   or derived from the following table.
2. Head and cycle times are calculated as follows: (count ≤ 63): max (3 + n+ mod (n + 1,2), 6).
3. Head and cycle times are calculated as follows: (count ≤ 8): max (2 + n+ mod (n,2), 6).

d = Direction (left or right)

Clocks	Shift Counts									
6	0	1	2	3	4	5	6	8	9	12
8	7	10	11	13	14	16	17	20		
10	15	18	19	21	22	24	25	28		
12	23	26	27	29	30	32	33	36		
14	31	34	35	37	38	40	41	44		
16	39	42	43	45	46	48	49	52		
18	47	50	51	53	54	56	57	60		
20	55	58	59	61	62					
22	63									

8.3.10 Bit Manipulation Instructions

The bit manipulation instruction table indicates the number of clock periods needed for the processor to perform the specified operation on the given addressing mode. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
BCHG	#, Dn	2	0	6(0/2/0)∗
BCHG	Dn, Dm	4	0	6(0/1/0)
BCHG	#, ⟨FEA⟩	1	2	8(0/2/1)∗
BCHG	Dn, ⟨FEA⟩	2	2	8(0/1/1)
BCLR	#, Dn	2	0	6(0/2/0)∗
BCLR	Dn, Dm	4	0	6(0/1/0)
BCLR	#, ⟨FEA⟩	1	2	8(0/2/1)∗
BCLR	Dn, ⟨FEA⟩	2	2	8(0/1/1)
BSET	#, Dn	2	0	6(0/2/0)∗
BSET	Dn, Dm	4	0	6(0/1/0)
BSET	#, ⟨FEA⟩	1	2	8(0/2/1)∗
BSET	Dn, ⟨FEA⟩	2	2	8(0/1/1)
BTST	#, Dn	2	0	4(0/2/0)∗
BTST	Dn, Dm	2	0	4(0/1/0)
BTST	#, ⟨FEA⟩	1	0	4(0/2/0)∗
BTST	Dn, ⟨FEA⟩	2	0	8(0/1/0)

∗An # fetch effective address time must be added for this instruction:
⟨FEA⟩ + ⟨FEA⟩ + ⟨OPER⟩

8.3.11 Conditional Branch Instructions

The conditional branch instruction table indicates the number of clock periods needed for the processor 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. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
Bcc	(taken)	2	–2	8(0/2/0)
Bcc.B	(not taken)	2	0	4(0/1/0)
Bcc.W	(not taken)	0	0	4(0/2/0)
Bcc.L	(not taken)	0	0	6(0/3/1)
DBcc	(T, not taken)	1	1	4(0/2/0)
DBcc	(F, –1, not taken)	2	0	6(0/2/0)
DBcc	(F, not –1, taken)	6	–2	10(0/2/0)
DBcc	(T, not taken)	4	0	6(0/1/0)∗
DBcc	(F, –1, not taken)	6	0	8(0/1/0)∗
DBcc	(F, not –1, taken)	6	0	10(0/0/0)∗

*In loop mode

8.3.12 Control Instructions

The control instruction table indicates the number of clock periods needed for the processor to perform the specified operation on the given addressing mode. Footnotes indicate when to account for the appropriate effective address times. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction		Head	Tail	Cycles
ANDI	#, SR	0	–2	12(0/2/0)
EORI	#, SR	0	–2	12(0/2/0)
ORI	#, SR	0	–2	12(0/2/0)
ANDI	#, CCR	2	0	6(0/2/0)
EORI	#, CCR	2	0	6(0/2/0)
ORI	#, CCR	2	0	6(0/2/0)
BSR.B		3	–2	13(0/2/2)
BSR.W		3	–2	13(0/2/2)
BSR.L		1	–2	13(0/2/2)
CHK	⟨FEA⟩, Dn (no ex)	2	0	8(0/1/0)
CHK	⟨FEA⟩, Dn (ex)	2	–2	42(2/2/6)
CHK2 (Save)	⟨FEA⟩, Dn (no ex)	1	1	3(0/1/0)
CHK2 (Op)	⟨FEA⟩, Dn (no ex)	2	0	18(X/0/0)
CHK2 (Save)	⟨FEA⟩, Dn (ex)	1	1	3(0/1/0)
CHK2 (Op)	⟨FEA⟩, Dn (ex)	2	–2	52(x + 2/1/6)
JMP	⟨CEA⟩	0	–2	6(0/2/0)
JSR	⟨CEA⟩	3	–2	13(0/2/2)
LEA	⟨CEA⟩, An	0	0	2(0/1/0)
LINK.W	An, #	2	0	10(0/2/2)
LINK.L	An, #	0	0	10(0/3/2)
NOP		0	0	2(0/1/0)
PEA	⟨CEA⟩	0	0	8(0/1/2)
RTD	#	1	–2	12(2/2/0)
RTR		1	–2	14(3/2/0)
RTS		1	–2	12(2/2/0)
UNLK	An	1	0	9(2/1/0)
X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles.				

NOTE: The CHK2 instruction involves a save step which other instructions do not have. To calculate total the instruction time, calculate the Save, the effective address, and the Operation execution times, and combine in the order listed, using the equations given in **8.1.6 Instruction Execution Time Calculation**.

8.3.13 Exception-Related Instructions and Operations

The exception-related instructions and operations table indicates the number of clock periods needed for the processor to perform the specified exception-related actions. No additional tables are needed to calculate total effective execution time for these instructions. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction	Head	Tail	Cycles
BKPT (Acknowledged)	0	0	14(1/0/0)
BKPT (Bus Error)	0	–2	35(3/2/4)
Breakpoint (Acknowledged)	0	0	10(1/0/0)
Breakpoint (Bus Error)	0	–2	42(3/2/6)
Interrupt	0	–2	30(3/2/4)*
RESET	0	0	518(0/1/0)
STOP	2	0	12(0/1/0)
LPSTOP	3	–2	25(0/3/1)
Divide-by-Zero	0	–2	36(2/2/6)
Trace	0	–2	36(2/2/6)
TRAP #	4	–2	29(2/2/4)
ILLEGAL	0	–2	25(2/2/4)
A-line	0	–2	25(2/2/4)
F-line (First word illegal)	0	–2	25(2/2/4)
F-line (Second word illegal) ea = Rn	1	–2	31(2/3/4)
F-line (Second word illegal) ea ≠ Rn (Save)	1	1	3(0/1/0)
F-line (Second word illegal) ea ≠ Rn (Op)	4	-2	29(2/2/4)
Privileged	0	–2	25(2/2/4)
TRAPcc (trap)	2	–2	38(2/2/6)
TRAPcc (no trap)	2	0	4(0/1/0)
TRAPcc.W (trap)	2	–2	38(2/2/6)
TRAPcc.W (no trap)	0	0	4(0/2/0)
TRAPcc.L (trap)	0	–2	38(2/2/6)
TRAPcc.L (no trap)	0	0	6(0/3/0)
TRAPV (trap)	2	–2	38(2/2/6)
TRAPV (no trap)	2	0	4(0/1/0)

*Minimum interrupt acknowledge cycle time is assumed to be three clocks.

NOTE: The F-line (Second word illegal) operation involves a save step which other operations do not have. To calculate, total the operation time, calculate the Save, then calculate effective address and the Operation execution times. Combine in the order listed, using the equations given in **8.1.6 Instruction Execution Time Calculation**.

8.3.14 Save and Restore Operations

The save and restore operations table indicates the number of clock periods needed for the processor to perform the specified state save or return from exception. Complete execution times and stack length are given. No additional tables are needed to calculate total effective execution time for these instructions. The total number of clock cycles is outside the parentheses. The numbers inside parentheses (r/p/w) are included in the total clock cycle number. All timing data assumes two-clock reads and writes.

Instruction	Head	Tail	Cycles
BERR on instruction	0	–2	<58(2/2/12)
BERR on exception	0	–2	48(2/2/12)
RTE (four-word frame)	1	–2	24(4/2/0)
RTE (six-word frame)	1	–2	26(4/2/0)
RTE (BERR on instruction)	1	–2	50(12/12/Y)
RTE (BERR on four-word frame)	1	–2	66(10/2/4)
RTE (BERR on six-word frame)	1	–2	70(12/2/6)
< = Maximum time is indicated — certain data or mode combinations execute faster.			
Y = If a bus error occurred during a write cycle, the cycle is rerun by the RTE.			

APPENDIX AM68000 FAMILY SUMMARY

Appendix A summarizes the characteristics of the microprocessors in the M68000 Family. The M68000 user's manual includes more detailed information about the MC68000 and MC68010 differences.

	MC68000	MC68010	CPU32	MC68020
Data Bus Size (Bits)	1 6	1 6	8, 16	8, 16, 32
Address Bus Size (Bits)	24	24	24	32
Instruction Cache (in Words)	—	3*	3*	128

*Three-word cache for the loop mode

Virtual Memory/Machine

MC68000	None
MC68010	Bus Error Detection, Instruction Continuation
CPU32	Bus Error Detection, Instruction Restart
MC68020	Bus Error Detection, Instruction Continuation

Coprocessor Interface

MC68000	Emulated in Software
MC68010	Emulated in Software
CPU32	Emulated in Software
MC68020	In Microcode

Word/Long-Word Data Alignment

MC68000	Word/Long-Word Data, Instructions, and Stack Must Be Word Aligned
MC68010	Word/Long-Word Data, Instructions, and Stack Must Be Word Aligned
CPU32	Word/Long-Word Data, Instructions, and Stack Must Be Word Aligned
MC68020	Only Instructions Must Be Word Aligned (Data Alignment Improves Performance)

Control Registers

MC68000	None
MC68010	SFC, DFC, VBR
CPU32	SFC, DFC, VBR
MC68020	SFC, DFC, VBR, CACR, CAAR

Stack Pointers

MC68000	USP, SSP
MC68010	USP, SSP
CPU32	USP, SSP
MC68020	USP, SSP (MSP, ISP)

Status Register Bits

MC68000	T, S, I0/I1/I2, X/N/Z/V/C
MC68010	T, S, I0/I1/I2, X/N/Z/V/C
CPU32	T1/T0, S, I0/I1/I2, X/N/Z/V/C
MC68020	T1/T0, S, M, I0/I1/I2, X/N/Z/V/C

Function Code/Address Space

MC68000	FC0 — FC2 = is Interrupt Acknowledge Only
MC68010	FC0 — FC2 = 7 is CPU Space
CPU32	FC0 — FC2 = 7 is CPU Space
MC68020	FC0 — FC2 = 7 is CPU Space

Indivisible Bus Cycles

MC68000	Use AS Signal
MC68010	Use AS Signal
CPU32	Use RMC Signal
MC68020	Use RMC Signal

Stack Frames

MC68000	Supports Original Set
MC68010	Supports Formats $0, $8
CPU32	Supports Formats $0, $2, $C
MC68020	Supports Formats $0, $1, $2, $9, $A, $B

Table A-1 M68000 instruction Set Extensions

Mnemonic	Description	CPU32	M68020
Bcc	Supports 32-Bit Displacement	◊	◊
BFxxxx	Bit Field Instructions (BFCHG, BFCLR, BFEXTS, BFEXTU, BFFO, BFINS, BFSET, BFTST)		◊
BGND	Background Operation	◊	
BKPT	New Instruction Function	◊	◊
BRA	Supports 32-Bit Displacement	◊	◊
BSR	Supports 32-Bit Displacement	◊	◊
CALLM	New Instruction		◊
CAS,CAS2	New Instruction		◊
CHK	Supports 32-Bit Operands	◊	◊
CHK2	New Instruction	◊	◊
CMP1	Supports Program Counter Relative Addressing	◊	◊
CMP2	New Instruction	◊	◊
cp	Coprocessor Instructions		◊
DIVS/DIVU	Supports 32-Bit and 64-Bit Operations	◊	◊
EXTB	Supports 8-Bit Extend to 32 Bits	◊	◊
LINK	Supports 32-Bit Displacement	◊	◊
LPSTOP	New Instruction	◊	
MOVEC	Supports New Control Registers	◊	◊
MULS/MULU	Supports 32-Bit Operands and 64-Bit Results	◊	◊
PACK	New Instruction		◊
RTM	New Instruction		◊
TBLSN,TBLUN TBLS,TBLU	New Instruction	◊	
TST	Supports Program Counter Relative, Immediate, and An Addressing	◊	◊
TRAPcc	New Instruction	◊	◊
UNPK	New Instruction		◊

Table A-2 M68000 Addressing Modes

Mode	Mnemonic	MC68010/ MC68000	CPU32	MC68020
Register Direct	Rn	◊	◊	◊
Address Register Indirect	(An)	◊	◊	◊
Address Register Indirect with Postincrement	(An)+	◊	◊	◊
Address Register Indirect with-(An) Predecrement	0 0	◊	◊	◊
Address Register Indirect with Displacement	(16, An)	◊	◊	◊
Address Register Indirect with Index (8-Bit Displacement)	(d8, An, Xn)	◊	◊	◊
Address Register Indirect with Index (Base Displacement)	(bd, An, Xn * SCALE)		◊	◊
Memory Indirect with Postincrement	([bd, An], Xn, Od)			◊
Memory Indirect with Predecrement	([bd, An, Xn], Od)			◊
Absolute Short	(xxx).W	◊	◊	◊
Absolute Long	(xxx).L	◊	◊	◊
Program Counter Indirect with Displacement	(d16, PC)	◊	◊	◊
Program Counter Indirect with Index (8-Bit Displacement)	(d8, PC, Xn)	◊	◊	◊
Program Counter Indirect with Index (Base Displacement)	(bd, PC, Xn * SCALE)		◊	◊
Immediate	#(data)	◊	◊	◊
Program Counter Memory Indirect with Postincrement	([bd, PC], Xn, od)			◊
Program Counter Memory Indirect with Predecrement	([bd, PC, Xn], od)			◊

INDEX

–U–

–V–

–W–