

CENTRAL  
PROCESSOR  
UNIT




REFERENCE  
MANUAL



**MOTOROLA**

# CPU32

## Reference Manual

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.



## PREFACE

This reference manual describes the capabilities, operation, and programming of the CPU32 instruction processing module found in the M68300 Family of embedded controllers. This manual is a part of a multivolume set of manuals — each volume corresponding to a major module in the M68300 Family. Each device in the M68300 Family also has a system integration user's manual that describes the function and operation of that particular device with references to the other volumes. The manual set for each device in the M68300 Family will require two to four volumes. This manual consists of the following sections and appendix:

- Section 1. Introduction
- Section 2. Architecture Summary
- Section 3. Addressing Modes
- 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

### NOTE

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

The audience of this manual includes systems designers, systems programmers, and applications programmers. Systems designers need some knowledge of all sections of this volume, with particular emphasis on Sections 1, 7, Appendix A, and the electrical specifications and mechanical data from the appropriate system integration user's manual. Systems programmers should become familiar with Sections 1, 2, 3, 4, 5, 6, 8, and Appendix A. Applications programmers can find most of the information they need in Sections 1, 2, 3, 4, 5, 8, and Appendix A.

From a different viewpoint, the audience for this book consists of users of the M68000 Family members and those not familiar with the CPU32. Users of the other M68000 Family members can find references to similarities to and differences from the other Motorola microprocessors throughout the manual. However, Sections 1, 2, and Appendix A specifically identify the CPU32 within the rest of the M68000 Family and contrast its differences.



# TABLE OF CONTENTS

Paragraph Number	Title	Page Number
<b>Section 1</b>		
<b>Overview</b>		
1.1	Features.....	1-2
1.1.1	Virtual Memory .....	1-2
1.1.2	Loop Mode Instruction Execution .....	1-3
1.1.3	Vector Base Register .....	1-4
1.1.4	Improved Exception Handling .....	1-4
1.1.5	Enhanced Addressing Modes.....	1-5
1.1.6	Instruction Set.....	1-5
1.1.6.1	Table Lookup and Interpolate Instructions .....	1-6
1.1.6.2	Low-Power STOP Instruction.....	1-6
1.1.7	Processing States .....	1-6
1.1.8	Privilege States.....	1-8
1.2	Block Diagram.....	1-8
<b>Section 2</b>		
<b>Architecture Summary</b>		
2.1	Programming Model .....	2-1
2.2	Registers.....	2-3
2.3	Data Types.....	2-4
2.3.1	Organization in Registers.....	2-4
2.3.1.1	Data Registers .....	2-5
2.3.1.2	Address Registers .....	2-6
2.3.1.3	Control Registers .....	2-6
2.3.2	Organization in Memory.....	2-7
<b>Section 3</b>		
<b>Data Organization and Addressing Capabilities</b>		
3.1	Program and Data References .....	3-2
3.2	Notation Conventions.....	3-2
3.3	Implicit Reference .....	3-3

## TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
3.4	Effective Address.....	3-4
3.4.1	Register Direct Mode.....	3-4
3.4.1.1	Data Register Direct .....	3-4
3.4.1.2	Address Register Direct .....	3-4
3.4.2	Memory Addressing Modes.....	3-5
3.4.2.1	Address Register Indirect.....	3-5
3.4.2.2	Address Register Indirect with Postincrement.....	3-5
3.4.2.3	Address Register Indirect with Predecrement.....	3-6
3.4.2.4	Address Register Indirect with Displacement .....	3-6
3.4.2.5	Address Register Indirect with Index (8-Bit Displacement) .....	3-6
3.4.2.6	Address Register Indirect with Index (Base Displacement) .....	3-8
3.4.3	Special Addressing Modes.....	3-8
3.4.3.1	Program Counter Indirect with Displacement.....	3-8
3.4.3.2	Program Counter Indirect with Index (8-Bit Displacement) .....	3-9
3.4.3.3	Program Counter Indirect with Index (Base Displacement) .....	3-9
3.4.3.4	Absolute Short Address.....	3-10
3.4.3.5	Absolute Long Address .....	3-10
3.4.3.6	Immediate Data .....	3-11
3.4.4	Effective Address Encoding Summary.....	3-11
3.5	Programming View of Addressing Modes .....	3-13
3.5.1	Addressing Capabilities .....	3-14
3.5.2	General Addressing Mode Summary .....	3-15
3.6	M68000 Family Addressing Capability.....	3-17
3.7	Other Data Structures.....	3-18
3.7.1	System Stack .....	3-18
3.7.2	User Stacks.....	3-19
3.7.3	Queues.....	3-20

### Section 4 Instruction Set

4.1	M68000 Family Compatibility .....	4-1
4.1.1	New Instructions.....	4-2
4.1.1.1	Low-Power STOP (LPSTOP) .....	4-2
4.1.1.2	Table Lookup and Interpolate (TBL) .....	4-2

## TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
4.1.2	Unimplemented Instructions .....	4-2
4.2	Instruction Format .....	4-3
4.3	Instruction Summary.....	4-4
4.3.1	Data Movement Instructions .....	4-5
4.3.2	Integer Arithmetic Operations .....	4-6
4.3.3	Logical Instructions.....	4-7
4.3.4	Shift and Rotate Instructions.....	4-8
4.3.5	Bit Manipulation Instructions .....	4-9
4.3.6	Binary-Coded Decimal (BCD) Instructions .....	4-9
4.3.7	Program Control Instructions .....	4-10
4.3.8	System Control Instructions.....	4-10
4.4	Instruction Details.....	4-12
4.4.1	Notation and Format.....	4-12
4.4.2	Condition Code Register.....	4-14
4.4.3	Condition Tests .....	4-17
4.4.4	Instruction Descriptions.....	4-18
4.5	Instruction Format Summary .....	4-177
4.6	Using the Table Instruction.....	4-192
4.6.1	Table Example 1: Standard Usage.....	4-193
4.6.2	Table Example 2: Compressed Table .....	4-194
4.6.3	Table Example 3: 8-Bit Independent Variable.....	4-196
4.6.4	Table Example 4: Maintaining Precision.....	4-198
4.6.5	Table Example 5: Surface Interpolations .....	4-200
4.7	Nested Subroutine Calls.....	4-201
4.8	Pipeline Synchronization with the NOP Instruction .....	4-201

### Section 5 Processing States

5.1	Privilege Levels .....	5-2
5.1.1	Supervisor Privilege Level .....	5-2
5.1.2	User Privilege Level .....	5-3
5.1.3	Changing Privilege Level .....	5-3
5.2	Address Space Types.....	5-4
5.2.1	Type 0000 — Breakpoint.....	5-5
5.2.2	Type 0001 — MMU Access .....	5-5
5.2.3	Type 0010 — Coprocessor Access .....	5-5



## TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
5.2.4	Type 0011 — Internal Register Access.....	5-6
5.2.5	Type 1111 — Interrupt Acknowledge.....	5-6
5.3	Exception Processing .....	5-6
5.3.1	Exception Vectors .....	5-7
5.3.2	Exception Stack Frame .....	5-7

### Section 6 Exception Processing

6.1	Exception Vectors.....	6-1
6.1.1	Types of Exceptions.....	6-1
6.1.2	Multiple Exceptions.....	6-3
6.1.3	Exception Stack Frame .....	6-4
6.1.4	Exception Processing Sequence .....	6-5
6.2	Processing of Specific Exceptions.....	6-5
6.2.1	Reset.....	6-6
6.2.2	Bus Error .....	6-6
6.2.3	Address Error.....	6-8
6.2.4	Instruction Traps.....	6-9
6.2.5	Software Breakpoints .....	6-10
6.2.6	Hardware Breakpoints.....	6-10
6.2.7	Format Error .....	6-11
6.2.8	Illegal or Unimplemented Instructions.....	6-11
6.2.9	Privilege Violations .....	6-12
6.2.10	Tracing .....	6-13
6.2.11	Interrupts.....	6-15
6.2.12	Return from Exception .....	6-16
6.3	Fault Recovery.....	6-18
6.3.1	Types of Faults.....	6-20
6.3.1.1	Type I: Released Write Faults.....	6-20
6.3.1.2	Type II: Prefetch, Operand, RMW, and MOVEP Faults.....	6-21
6.3.1.3	Type III: Faults During MOVEM Operand Transfers .....	6-22
6.3.1.4	Type IV: Faults During Exception Processing .....	6-23
6.3.2	Correcting the Fault .....	6-23
6.3.2.1	Completing Released Writes (Type I) via Software .....	6-23
6.3.2.2	Completing Released Writes (Type I) via RTE.....	6-24
6.3.2.3	Correcting Type II Faults via RTE.....	6-24

## TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
6.3.2.4	Correcting Type III Faults via Software.....	6-25
6.3.2.5	Correcting Type III Faults via RTE.....	6-26
6.3.2.6	Correcting Type IV Faults via Software.....	6-26
6.4	CPU32 Stack Frames.....	6-27
6.4.1	Normal Four-Word Stack Frame.....	6-27
6.4.2	Normal Six-Word Stack Frame.....	6-28
6.4.3	BERR Stack Frame.....	6-28

### Section 7

#### Development Support

7.1	CPU32 Integrated Development Support.....	7-1
7.1.1	Background Debug Mode (BDM) Overview.....	7-2
7.1.2	Deterministic Opcode Tracking Overview.....	7-3
7.1.3	On-Chip Hardware Breakpoint Overview.....	7-3
7.2	Background Debug Mode (BDM).....	7-3
7.2.1	Enabling BDM.....	7-3
7.2.2	BDM Sources.....	7-4
7.2.2.1	External BKPT Signal.....	7-5
7.2.2.2	BGND Instruction.....	7-5
7.2.2.3	Double Bus Faults.....	7-5
7.2.2.4	Peripheral Breakpoints.....	7-5
7.2.3	Entering BDM.....	7-6
7.2.4	Command Execution.....	7-6
7.2.5	Returning from BDM.....	7-7
7.2.6	Serial Interface.....	7-8
7.2.6.1	CPU Serial Logic.....	7-8
7.2.6.2	Development System Serial Logic.....	7-11
7.2.7	Command Set.....	7-13
7.2.7.1	Command Format.....	7-13
7.2.7.2	Command Sequence Diagrams.....	7-14
7.2.7.3	Command Set Summary.....	7-16
7.2.7.4	Read A/D Register (RAREG/RDREG).....	7-17
7.2.7.5	Write A/D Register (WAREG/WDREG).....	7-18
7.2.7.6	Read System Register (RSREG).....	7-19
7.2.7.7	Write System Register (WSREG).....	7-20
7.2.7.8	Read Memory Location (READ).....	7-21

## TABLE OF CONTENTS (Continued)

Paragraph Number	Title	Page Number
7.2.7.9	Write Memory Location (WRITE) .....	7-22
7.2.7.10	Dump Memory Block (DUMP) .....	7-24
7.2.7.11	Fill Memory Block (FILL) .....	7-26
7.2.7.12	Resume Execution (GO).....	7-28
7.2.7.13	Call User Code (CALL).....	7-29
7.2.7.14	Reset Peripherals (RST).....	7-31
7.2.7.15	No Operation (NOP).....	7-32
7.2.7.16	Future Commands .....	7-32
7.3	Deterministic Opcode Tracking.....	7-33
7.3.1	Instruction Fetch (IFETCH).....	7-33
7.3.2	Instruction Pipe (IPIPE) .....	7-33
7.3.3	Opcode Tracking during Loop Mode.....	7-35

### Section 8

#### Instruction Execution Timing

8.1	Resource Scheduling.....	8-1
8.1.1	Microsequencer.....	8-1
8.1.2	Instruction Pipeline .....	8-2
8.1.3	Bus Controller Resources.....	8-3
8.1.3.1	Prefetch Controller.....	8-3
8.1.3.2	Write-Pending Buffer.....	8-3
8.1.3.3	Microbus Controller .....	8-4
8.1.4	Instruction Execution Overlap .....	8-4
8.1.5	Effects of Wait States .....	8-5
8.2	Instruction Stream Timing Examples .....	8-6
8.2.1	Timing Example 1: Execution Overlap .....	8-6
8.2.2	Timing Example 2: Branch Instructions .....	8-7
8.2.3	Timing Example 3: Negative Tails .....	8-8
8.3	Instruction Timing Tables .....	8-9
8.3.1	Fetch Effective Address .....	8-12
8.3.2	Calculate Effective Address.....	8-13
8.3.3	MOVE Instruction.....	8-14
8.3.4	Special-Purpose MOVE Instruction.....	8-15
8.3.5	Arithmetic/Logical Instructions .....	8-16
8.3.6	Immediate Arithmetic/Logical Instructions.....	8-17
8.3.7	Binary-Coded Decimal and Extended Instructions.....	8-18

## TABLE OF CONTENTS (Concluded)

<b>Paragraph Number</b>	<b>Title</b>	<b>Page Number</b>
8.3.8	Single Operand Instructions.....	8-19
8.3.9	Shift/Rotate Instructions .....	8-20
8.3.10	Bit Manipulation Instructions .....	8-21
8.3.11	Conditional Branch Instructions.....	8-22
8.3.12	Control Instructions.....	8-23
8.3.13	Exception-Related Instructions and Operations .....	8-24
8.3.14	Save and Restore Operations.....	8-25

### **Appendix A M68000 Family Summary**

#### **Index**



## LIST OF ILLUSTRATIONS

Figure Number	Title	Page Number
1-1	Loop Mode Instruction Sequence .....	1-3
1-2	CPU32 Block Diagram .....	1-9
2-1	User Programming Model .....	2-2
2-2	Supervisor Programming Model Supplement .....	2-2
2-3	Status Register .....	2-4
2-4	Data Organization in Data Registers.....	2-5
2-5	Address Organization in Address Registers .....	2-6
2-6	Memory Operand Addressing .....	2-8
3-1	Single-Effective-Address Instruction Operation Word.....	3-1
3-2	Effective Address Specification Formats.....	3-12
3-3	Using SIZE in the Index Selection.....	3-14
3-4	Using Absolute Address with Indexes.....	3-15
3-5	Addressing Array Items.....	3-16
3-6	M68000 Family Address Extension Words .....	3-17
4-1	Instruction Word General Format .....	4-3
4-2	Instruction Description Format .....	4-19
4-3	Table Example 1.....	4-193
4-4	Table Example 2.....	4-194
4-5	Table Example 3.....	4-196
5-1	General Exception Stack Frame.....	5-8
6-1	Exception Stack Frame .....	6-4
6-2	Reset Operation Flowchart.....	6-7
6-3	Format \$0 — Four-Word Stack Frame .....	6-27
6-4	Format \$2 — Six-Word Stack Frame .....	6-28
6-5	Format \$C — BERR Stack for Prefetches and Operands .....	6-29
6-6	Internal Transfer Count Register.....	6-29
6-7	Format \$C — BERR Stack During Four- or Six-Word Stack .....	6-30
6-8	Format \$C — BERR Stack on MOVEM Operand.....	6-31

## LIST OF ILLUSTRATIONS (Continued)

Figure Number	Title	Page Number
7-1	Traditional In-Circuit Emulator Diagram .....	7-2
7-2	Bus State Analyzer Configuration .....	7-2
7-3	BDM Block Diagram .....	7-4
7-4	BDM Command Execution Flowchart .....	7-7
7-5	Debug Serial I/O Block Diagram .....	7-9
7-6	Serial Interface Timing Diagram .....	7-10
7-7	BKPT Timing for Single Bus Cycle .....	7-11
7-8	BKPT Timing for Forcing BDM .....	7-12
7-9	BKPT/DSCLK Logic Diagram .....	7-12
7-10	Command-Sequence-Diagram Example .....	7-15
7-11	Functional Model of Instruction Pipeline .....	7-34
7-12	Instruction Pipeline Timing Diagram .....	7-35
8-1	Block Diagram of Independent Resources .....	8-2
8-2	Simultaneous Instruction Execution .....	8-4
8-3	Attributed Instruction Times .....	8-5
8-4	Example 1 — Instruction Stream .....	8-6
8-5	Example 2 — Branch Taken .....	8-7
8-6	Example 2 — Branch Not Taken .....	8-7
8-7	Example 3 — Branch Negative Tail .....	8-8

# LIST OF TABLES

Table Number	Title	Page Number
1-1	Instruction Set Summary.....	1-7
3-1	Effective Addressing Mode Categories.....	3-13
4-1	Data Movement Operations.....	4-5
4-2	Integer Arithmetic Operations.....	4-6
4-3	Logical Operations.....	4-7
4-4	Shift and Rotate Operations.....	4-8
4-5	Bit Manipulation Operations.....	4-9
4-6	Binary-Coded Decimal Operations.....	4-9
4-7	Program Control Operations.....	4-10
4-8	System Control Operations.....	4-11
4-9	Condition Code Computations.....	4-14
4-10	Conditional Tests.....	4-17
4-11	Operation Code Map.....	4-177
5-1	Address Space Encodings.....	5-4
6-1	Exception Vector Assignments.....	6-2
6-2	Exception Groups.....	6-3
6-3	Tracing Control.....	6-13
7-1	BDM Source Summary.....	7-5
7-2	Polling the BDM Entry Source.....	7-6
7-3	CPU-Generated Message Encoding.....	7-9
7-4	BDM Command Summary.....	7-16





## SECTION 1 OVERVIEW

The CPU32, the instruction processing module of the M68300 Family, is based on the industry-standard MC68000 core processor with many features of the MC68010 and MC68020 as well as unique features suited for high-performance controller applications. The CPU32 is designed to provide a significant increase in performance over the MC68HC11 CPU to meet the demand for higher performance requirements for the 1990's while maintaining source code and binary code compatibility with the M68000 Family.

One major goal of the CPU32 is to increase system throughput. This increase could not be achieved by simply increasing the clock/bus frequency or adding a few new instructions to an existing 8-bit, MC6800-type CPU. A faster, more powerful CPU, capable of processing data sizes up to 32 bits, is included on the chip as a first step in realizing such a performance increase. As controller applications become more complex and control programs become larger, high-level languages (HLLs) will become the system designer's choice in programming languages. HLLs allow users to develop complex algorithms faster, with few errors, and provide easier portability. The CPU32 has an instruction set based on the M68000 Family, which can efficiently support HLLs.

Ease of programming is an important consideration in using a microcontroller. An instruction format implementing a register-memory interaction philosophy predominates the design, and all data resources are available to all operations requiring those resources. All eight multifunction data registers are available as data resources, and all seven general-purpose addressing registers are available for addressing data. Although the program counter (PC) and stack pointers (SP) are special-purpose registers, they are also available for most data addressing activities. The eight general-purpose 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. Ease of program checking and diagnosis is further enhanced by trace and trap capabilities at the instruction level.

## 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 Index
  - 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 at –40–125°C
- Fully Static Implementation

### 1.1.1 Virtual Memory

The full addressing range of the CPU32 is 16 Mbytes in each of eight address spaces. Even though most systems implement a smaller physical memory, the system can be made to appear to have a full 16 Mbytes of memory available to each user program by using virtual memory techniques.

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 instruction. 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.

The loop mode is entered when the DBcc instruction is executed and the loop displacement is  $-4$ . Once in loop mode, the processor performs only the data cycles associated with the instruction and suppresses all instruction fetches. The termination condition and count are checked after each execution of the data operations of the looped instruction. The CPU32 automatically exits the loop mode on interrupts or other exceptions.

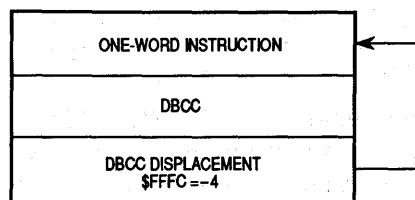
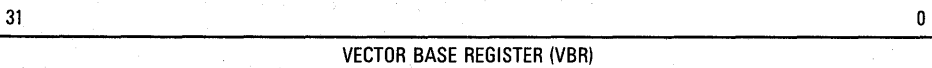


Figure 1-1. Loop Mode Instruction Sequence

### 1.1.3 Vector Base Register

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

The address of an interrupt exception vector is derived from an 8-bit vector number and the VBR. The vector numbers for some exceptions are obtained from an external device; other numbers are supplied automatically by the processor. The processor multiplies the vector number by four to calculate the vector offset, which is added to the VBR. The sum is the memory address of the vector. All exception vectors are located in supervisor data space, except the reset vector, which is located in supervisor program space. Only the initial reset vector is fixed in the processor's memory map; once initialization is complete, there are no fixed assignments. Since the VBR provides the base address of the vector table, the vector table can be located anywhere in memory; it can even be dynamically relocated for each task that is executed by an operating system. Details of exception processing are provided in **SECTION 6 EXCEPTION PROCESSING**.



### 1.1.4 Improved 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 instruction processing.

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 the exception stack frame. This information always includes the status register and PC context of the processor when the exception occurred. To support generic handlers, the processor places the vector offset in the exception stack frame. The processor also marks the frame with a frame format. The format field allows the return-from-exception (RTE) instruction to identify what information is on the stack so that it may 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 directly in memory; this flexibility eliminates the need for extra instructions to store register contents in memory.

The seven basic addressing modes are as follows:

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

Included in the register indirect addressing modes are the capabilities to postincrement, predecrement, and offset. The PC relative mode also has index and offset capabilities. In addition to these addressing modes, many instructions implicitly specify the use of the status register, SP, and/or PC. Addressing is explained fully in **SECTION 3 ADDRESSING MODES**. A summary of the 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 instructions found on the M68020 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, cpRESTORE, cpSAVE, cpScc, cpTRAPcc)
- PACK, UNPK — Pack, Unpack BCD Instructions

The CPU32 traps on unimplemented instructions or illegal effective addressing modes, allowing 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 core enhancements.

**1.1.6.1 TABLE LOOKUP AND INTERPOLATE INSTRUCTIONS.** To maximize throughput for real-time applications, reference data is often “precalculated” and stored in memory for quick access. The storage of each data point would require an inordinate amount of memory. The table instruction requires only a sample of data points stored in the array, reducing memory requirements. Intermediate values are recovered with this instruction via linear interpolation. The results are rounded (optional) with the round-to-nearest algorithm.

**1.1.6.2 LOW-POWER STOP INSTRUCTION.** In applications where power consumption is a consideration, the CPU32 forces the device into a low-power standby mode when immediate processing is not required. The low-power stop mode is entered by executing the LPSTOP instruction. The processor will remain in this mode until a user-specified (or higher) interrupt level or reset occurs.

## 1.1.7 Processing States

The processor is always in one of four processing states: normal, exception, halted, or background. The normal processing state is that associated with instruction execution; the bus is used to fetch instructions and operands and to store results. The exception processing state is associated with interrupts, trap instructions, tracing, and other exception conditions. The exception may

**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	Check Register Against Upper and Lower Bounds
CLR	Clear
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CMPM	Compare Memory to Memory
CMP2	Compare Register Against Upper and Lower Bounds
DBcc	Test Condition, Decrement and Branch
DIVS, DIVSL	Signed Divide
DIVU, DIVUL	Unsigned Divide
EOR	Logical Exclusive OR
EORI	Logical Exclusive OR Immediate
EXG	Exchange Registers
EXT, EXTB	Sign Extend
ILLEGAL	Take Illegal Instruction Trap
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link and Allocate
LPSTOP	Low-Power Stop
LSL, LSR	Logical Shift Left and Right

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



be internally generated explicitly by an instruction or by an unusual condition arising during the execution of an instruction. Externally, exception processing can be forced by an interrupt, a bus error, or a reset. The halted processing state is an indication of catastrophic hardware failure. For example, if during the exception processing of a bus error another bus error occurs, the processor assumes that the system is unusable and halts. The background processing state is initiated by breakpoints, execution of special instructions, or a double bus fault. Background processing allows interactive debugging of the system via a simple serial interface. Processing states are explained fully in **SECTION 5 PROCESSING STATES**.

### 1.1.8 Privilege States

The processor operates at one of two levels of privilege — user or supervisor. The supervisor level has higher privileges than the user level. Not all instructions are permitted to execute in the lower privileged user level, but all instructions are available at the supervisor level. This scheme allows a separation of supervisor and user levels so the supervisor can protect system resources from uncontrolled access. The processor uses the privilege level indicated by the S bit in the status register to select either the user or supervisor privilege level and either the user stack pointer (USP) or a supervisor stack pointer (SSP) for stack operations.

## 1.2 BLOCK DIAGRAM

A block diagram of the CPU32 is shown in Figure 1-2. The major blocks depicted operate in a highly independent fashion that maximizes concurrency of operation while managing the essential synchronization of instruction execution and bus operation. The bus controller loads instructions from the data bus into the decode unit. The sequencer and control unit provide overall chip control, managing the internal buses, registers, and functions of the execution unit.

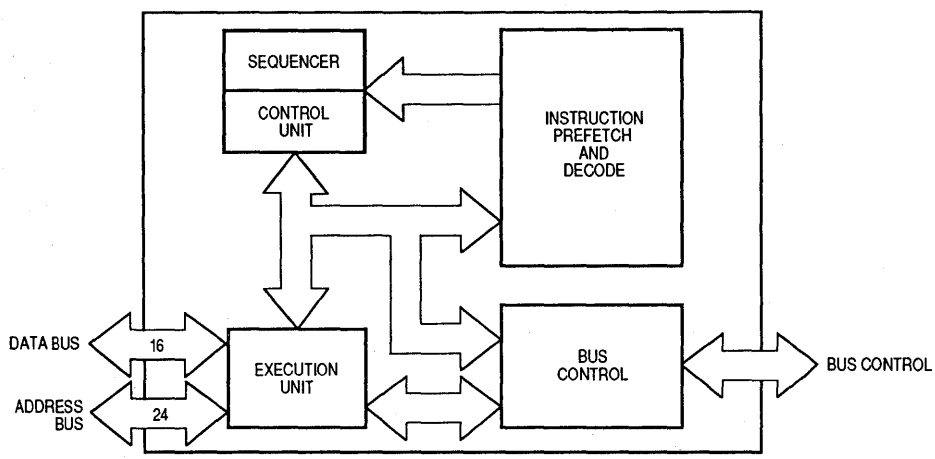


Figure 1-2. CPU32 Block Diagram



## SECTION 2

# ARCHITECTURE SUMMARY

The CPU32 architecture includes several important features that provide both power and versatility to the user. The CPU32 is source and object code compatible with the MC68000 and MC68010. All user state programs can be executed unchanged. The major CPU32 features are as follows:

- 32-Bit Internal Data Path and Arithmetic Hardware
- 24-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 programming model of the CPU32 consists of two groups of registers: user model and supervisor model that correspond to the user and supervisor privilege levels. Executing at the user privilege level, user programs can only use the registers of the user model. Executing at the supervisor level, system software uses the control registers of the supervisor level to perform supervisor functions.

As shown in the programming models (see Figures 2-1 and 2-2), the CPU32 has 16 32-bit general-purpose registers, a 32-bit program counter, one 32-bit supervisor stack pointer, a 16-bit status register, two alternate function

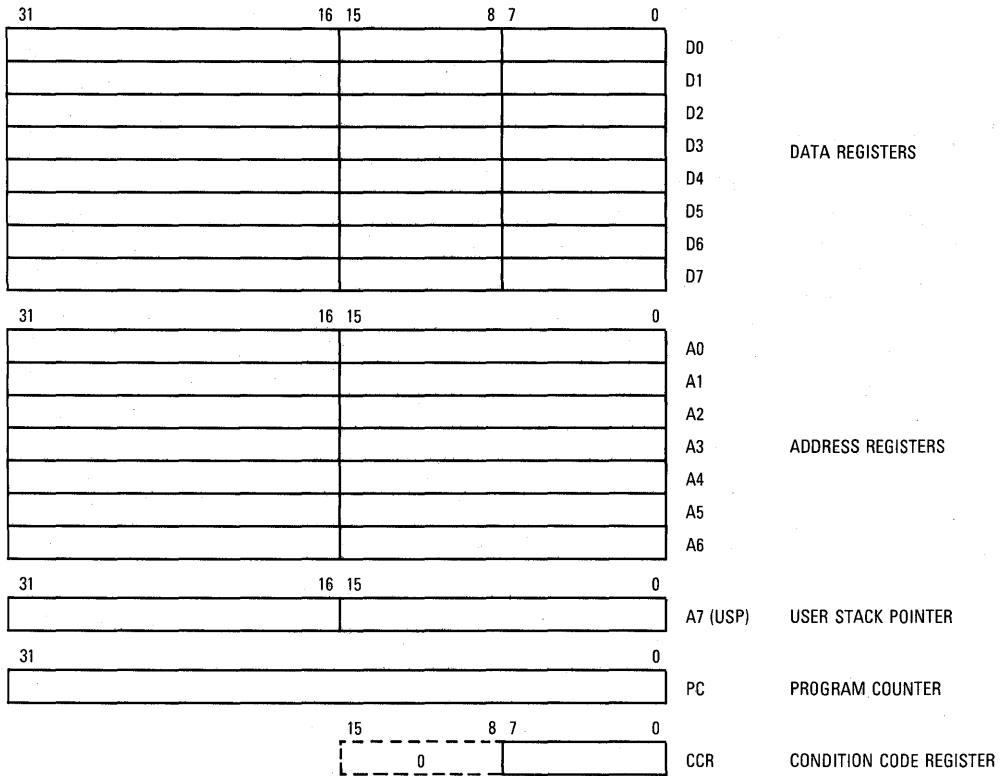


Figure 2-1. User Programming Model

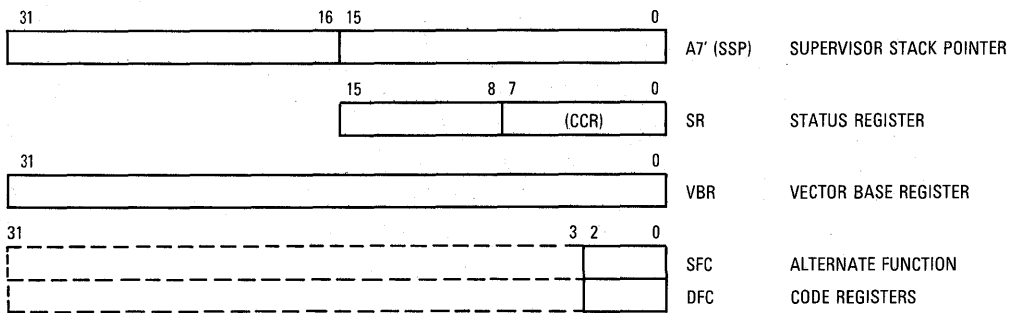


Figure 2-2. Supervisor Programming Model Supplement

code registers, and a 32-bit vector base register. The user programming model remains unchanged from previous M68000 Family microprocessors. The supervisor programming model, which supplements the user programming model, is used exclusively by the CPU32 system programmers who utilize the supervisor privilege level to implement sensitive operating system functions. The supervisor programming model contains all the controls to access and enable the special features of the CPU32. All application software, written to run at the nonprivileged user level, migrates to the CPU32 from any M68000 platform without modification.

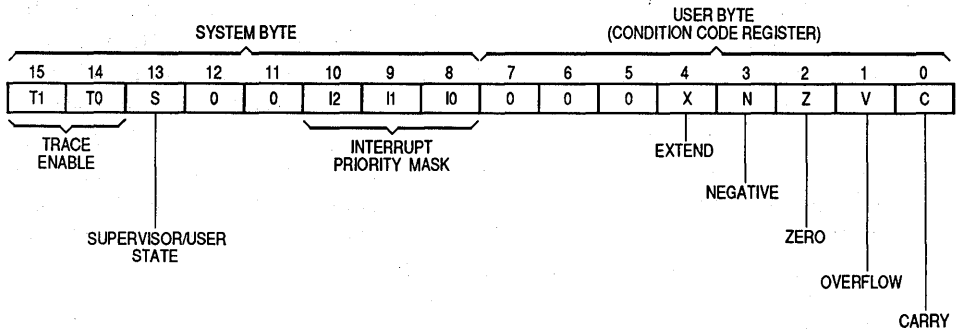
## 2.2 REGISTERS

Registers D7–D0 are used as data registers for bit (1 to 32 bits), byte (8 bit), word (16 bit), long-word (32 bit), and quad-word (64 bit) operations. Registers A6–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, the address registers may be used for word and long-word operations. All of the 16 general-purpose registers (D7–D0, A7–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) stores the processor status. It contains the condition codes that reflect the results of a previous operation and can be used for conditional instruction execution in a program. The condition codes are extend (X), negative (N), zero (Z), overflow (V), and carry (C). The user byte containing the condition codes is the only portion of the SR information available in the user privilege level; it is referenced as the condition code register (CCR) in user programs. In the supervisor privilege level, software can access the full status register, including the interrupt priority mask (three bits), as well as additional control bits. These bits put the processor in one of two trace modes (T1, T0) and in user or supervisor privilege level (S).

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.



**Figure 2-3. Status Register**

Alternate function code registers (SFC and DFC) contain 3-bit function codes. Function codes can be considered extensions of the 24-bit linear address that optionally provide as many as eight 16-Mbyte address spaces. Function codes are automatically generated by the processor to select address spaces for data and program at the user and supervisor privilege levels and to select a CPU address space used for processor functions (such as breakpoint and interrupt acknowledge cycles). Registers SFC and DFC are used by the MOVE instructions to explicitly specify the function codes of the memory address.

## 2.3 DATA TYPES

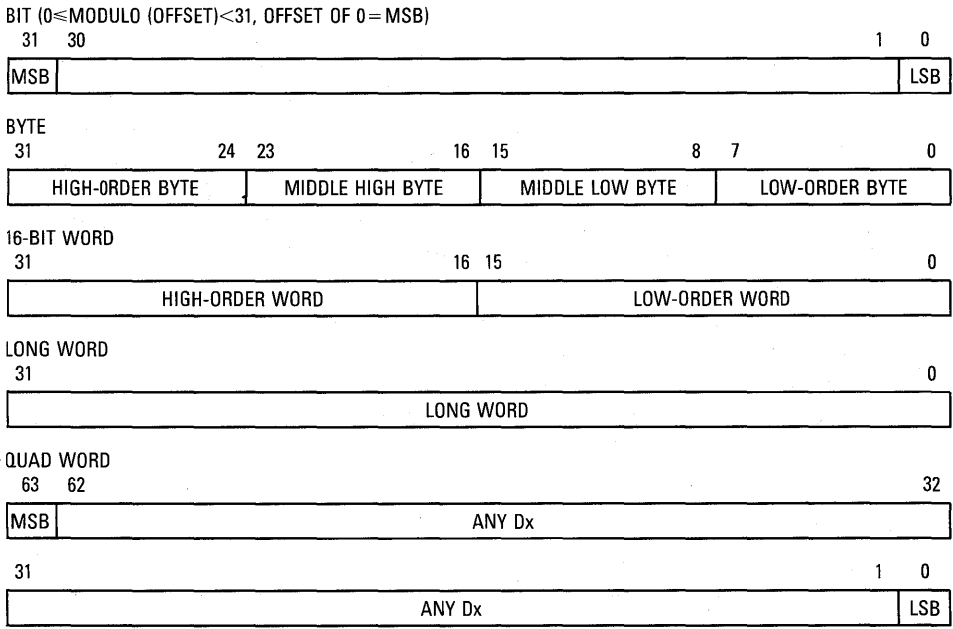
Six basic data types are supported:

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.



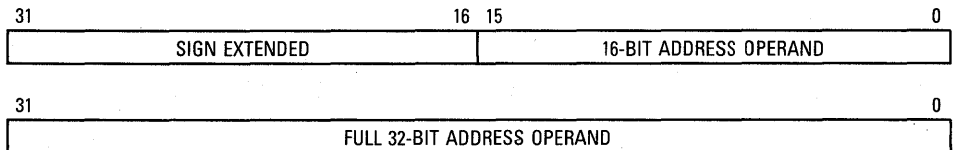
**Figure 2-4. Data Organization in Data Registers**

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. Although many BCD codes have been devised, the BCD instructions of the M68000 Family support formats in which the four LSBs consist of a binary number having the numeric value of the corresponding decimal number. In this BCD format, a byte contains one digit; the four LSBs contain the binary value, and the four MSBs are undefined. ABCD, SBCD, and NBCD operate on two BCD digits packed into a single byte.

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

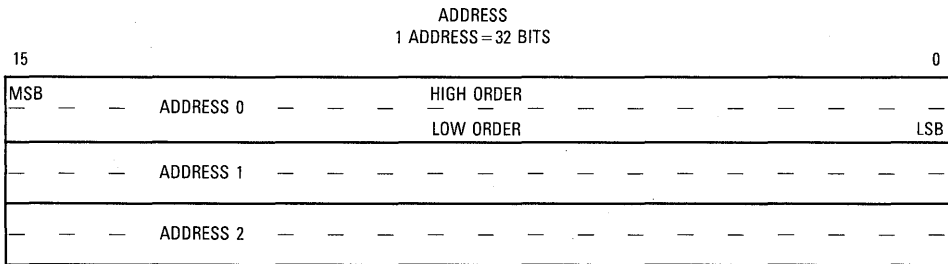
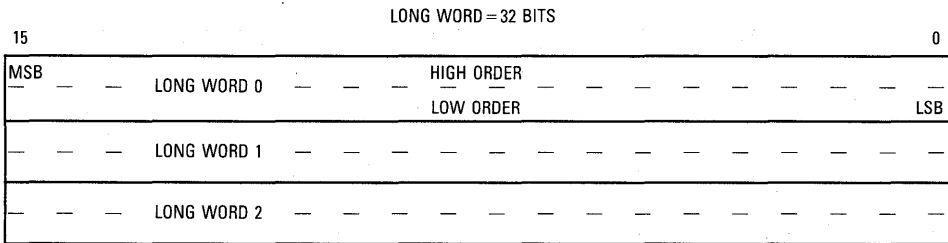
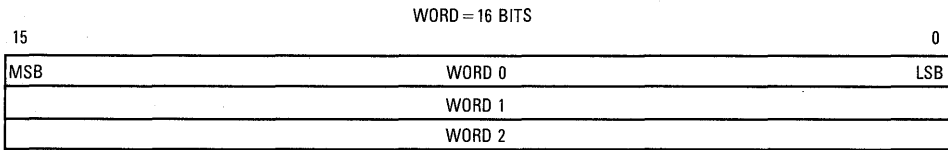
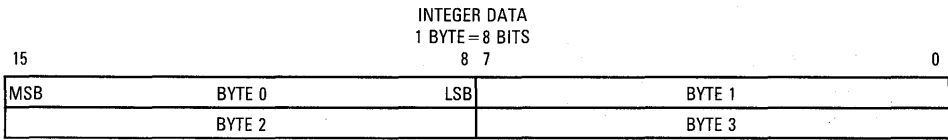
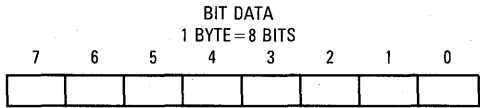
The SR shown in Figure 2-3 is 16 bits wide. Only 11 bits of the SR are defined; all undefined values are reserved by Motorola for future definition. The undefined bits are read as zeros and should be written as zeros for future compatibility. The lower byte of the SR is the CCR. Operations to the CCR

can be performed at the supervisor or user privilege level. All operations to the SR and CCR are word-size operations, but for all CCR operations, the upper byte is read as all zeros and is ignored when written, regardless of privilege level.

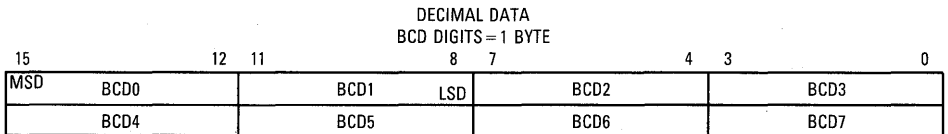
The alternate function code registers (SFC and DFC) are 32-bit registers with only bits 2:0 implemented that contain the address space values (FC2–FC0) for the read or write operand of the MOVES instruction. The MOVEC instruction is used to transfer values to and from the alternate function code registers. 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 in which lower addresses correspond to higher order bytes. The address,  $N$ , of a long-word data item corresponds to the address of the most significant byte of the highest order word. The lower order word is located at address  $N+2$ , leaving the least significant byte at address  $N+3$  (see Figure 2-1). The CPU32 requires long-word and word data as well as instruction words to be aligned on word boundaries (see Figure 2-6). Data misalignment is not supported.



MSB = Most Significant Bit  
LSB = Least Significant Bit



MSD = Most Significant Digit  
LSD = Least Significant Digit

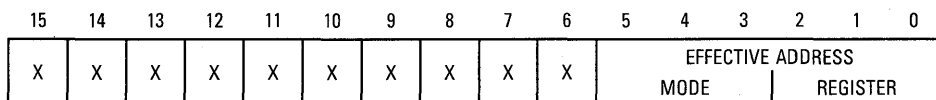
**Figure 2-6. Memory Operand Addressing**

## SECTION 3

# DATA ORGANIZATION AND ADDRESSING CAPABILITIES

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

Figure 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: mode field and register field. The value in the mode field selects one mode or a set of addressing modes. The register field specifies a register for the mode or a submode for modes that do not use registers.



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

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

The effective address field may require additional information to fully specify the operand address. This additional information, called the effective address extension, is contained in an additional word or words and is considered part of the instruction. Refer to **3.4.4 Effective Address Encoding Summary** for a description of the extension word formats.

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

## 3.1 PROGRAM AND DATA REFERENCES

An M68000 Family processor separates memory references into two classes, which creates two address spaces, each with a complete logical address range. The first class is program references, which includes primarily references to opcodes and extension words. The other class is data references. Operand reads are from the data space with two exceptions: 1) immediate operands embedded in the instruction stream and 2) operands addressed relative to the current program counter. Operands satisfying either of these two exceptions are classified as program space references. All operand writes are to data space.

# 3

## 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 — Denotes index register n (data or address), the index size (W for word, L for long word), and a scale factor (1, 2, 4, or 8 for no, word, long-word or 8 for quad-word scaling, respectively).

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

() — Identify 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 enumerates these instructions and the registers involved:

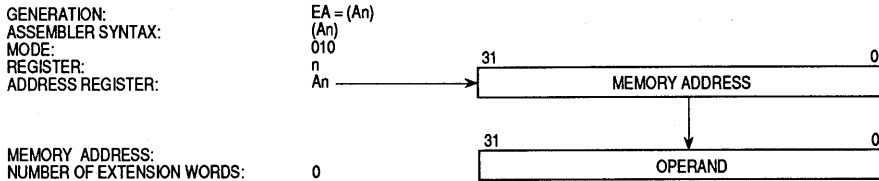
Instruction	Implicit Registers
ANDI to CCR	SR
ANDI to SR	SR
BRA	PC
BSR	PC, SP
CHK (exception)	SSP, SR
CHK2 (exception)	SSP, SR
DBcc	PC
DIVS (exception)	SSP, SR
DIVU (exception)	SSP, SR
EORI to CCR	SR
EORI to SR	SR
JMP	PC
JSR	PC, SP
LINK	SP
LPSTOP	SR
MOVE CCR	SR
MOVE SR	SR
MOVE USP	USP
ORI to CCR	SR
ORI to SR	SR
PEA	SP
RTD	PC, SP
RTE	PC, SP, SR
RTR	PC, SP, SR
RTS	PC, SP
STOP	SR
TRAP (exception)	SSP, SR
TRAPV (exception)	SSP, SR
UNLK	SP



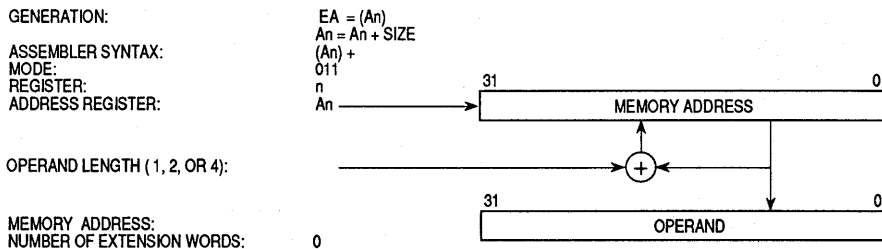
### 3.4.2 Memory Addressing Modes

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.

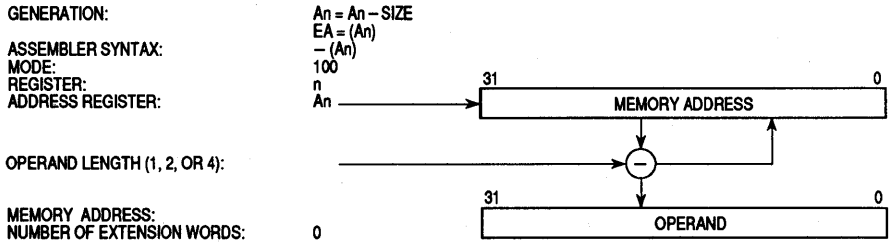


**3.4.2.2 ADDRESS REGISTER INDIRECT WITH POSTINCREMENT.** In the address register indirect with postincrement mode, the operand is in memory, and the address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two, or 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.

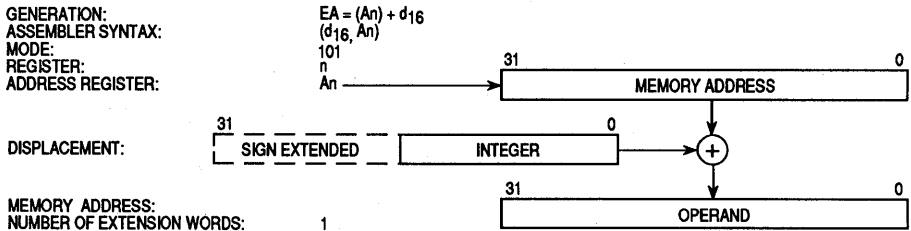




**3.4.2.3 ADDRESS REGISTER INDIRECT WITH PREDECREMENT.** In the address register indirect with predecrement mode, the operand is in memory, and the address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or 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 addressing mode requires one extension word that contains the index register indicator and an 8-bit displacement. The index register indicator includes size and scale information. In this mode, the operand is in memory. The address of the operand is the sum of the contents of the address register, the sign-extended displacement value in the low-order eight bits of the extension word, and the sign-extended contents of the index register (possibly scaled). The user must specify the displacement, the address register, and the index register in this mode.

GENERATION:  
 ASSEMBLER SYNTAX:  
 MODE:  
 REGISTER:  
 ADDRESS REGISTER:

$$EA = (An) + (Xn * SCALE) + dg$$

(dg, An, SIZE \* SCALE)

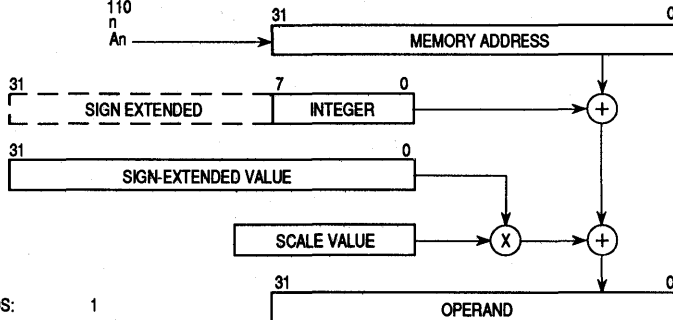
DISPLACEMENT:

INDEX REGISTER:

SCALE:

MEMORY ADDRESS:

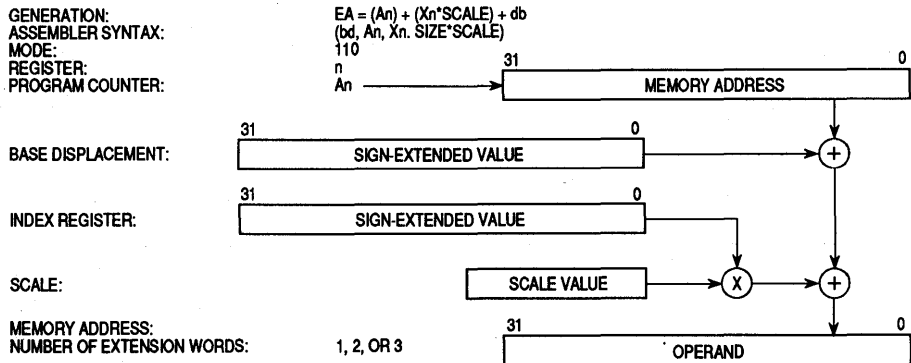
NUMBER OF EXTENSION WORDS: 1



This address mode uses two different formats of extension. The brief format provides fast indexed addressing; the full format provides a number of options in size of displacements. Both forms use an index operand. Notationally, this index operand is specified "Ri.sz\*scl". "Ri" selects one of the general data or address registers for the index register. The term "sz" refers to the index size and may be either: "W" or "L". The term "scl" refers to the index scale selection and may be any of 1, 2, 4, or 8. 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 is an address register if bit [15]=1. The register number of the index register is given by bits [14:12] of the extension word. The 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; if bit [11]=1, the index value is the long integer in the index register. Finally, the index value is scaled according to the scaling selection in bits [10:9] to derive the index operand. The scale selections 00, 01, 10, or 11 select scaling of the index value by 1, 2, 4, or 8, respectively. Brief format indexing requires one word of extension. 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.

**3.4.2.6 ADDRESS REGISTER INDIRECT WITH INDEX (BASE DISPLACEMENT).** This addressing mode requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The index register indicator includes size and 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.

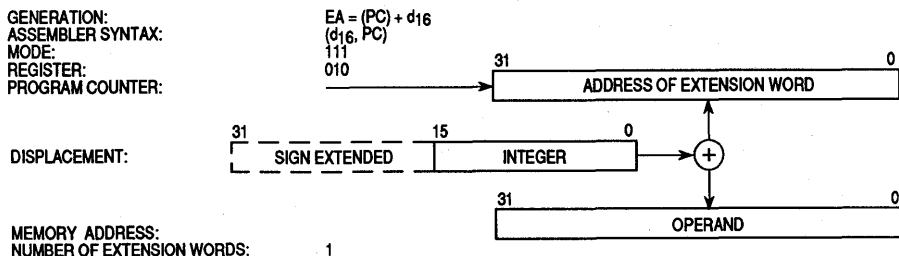
3



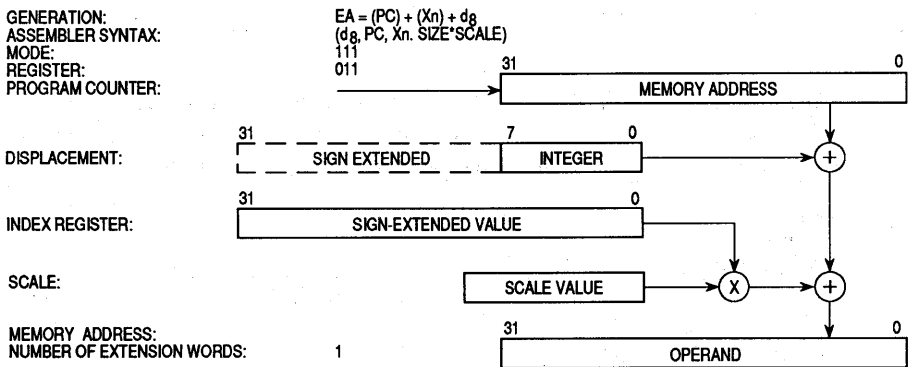
### 3.4.3 Special Addressing Modes

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.



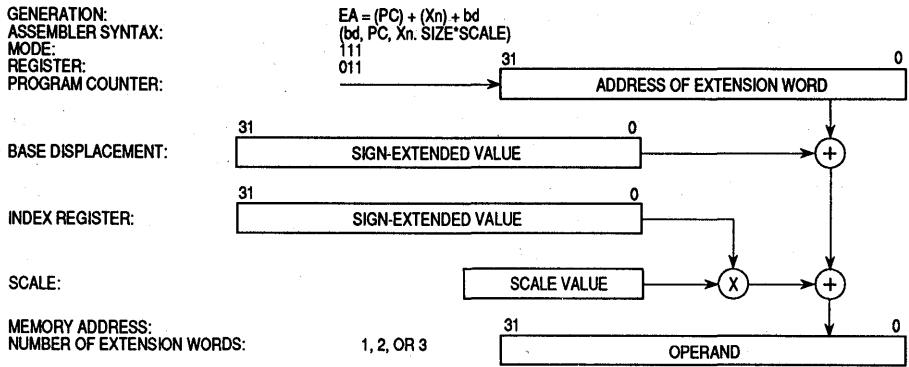
**3.4.3.2 PROGRAM COUNTER INDIRECT WITH INDEX (8-BIT DISPLACEMENT).** This mode is similar to the address register indirect with index (8-bit displacement) mode described in **3.4.2.5 ADDRESS REGISTER INDIRECT WITH INDEX (8-BIT DISPLACEMENT)**, but the program counter is used as the base register. The 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.



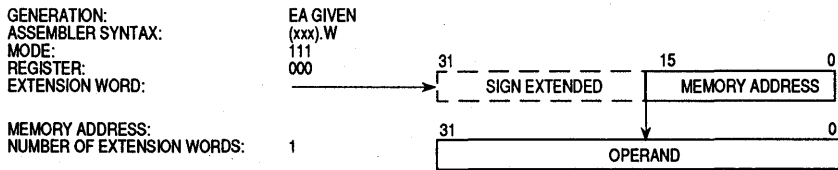
**3.4.3.3 PROGRAM COUNTER INDIRECT WITH INDEX (BASE DISPLACEMENT).** This mode is similar to the address register indirect with index (base displacement) mode described in **3.4.2.6 ADDRESS REGISTER INDIRECT WITH INDEX (BASE DISPLACEMENT)**, but the program counter is used as the base register. It requires 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

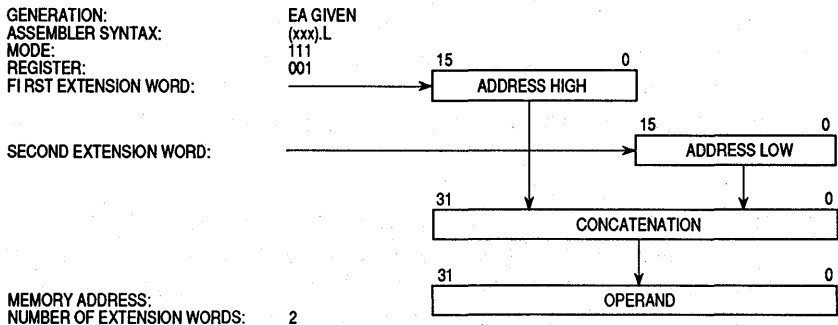
the program space with a data register indirect access by placing ZPC in the instruction and specifying a data register (Dn) as the index register.



**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

### 3.4.4 Effective Address Encoding Summary

Most of the 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 encoding of the mode field of this word selects the addressing mode. The register field contains the general register number or a value that selects the addressing mode when the mode field contains "111". 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 the source and destination EAs and with 32-bit base displacements for both addresses.

Grouped according to the use of the mode, EA modes can be classified as follows:

- Data A data addressing EA mode is one that refers to data operands.
- Memory A memory addressing EA mode is one that refers to memory operands.
- Alterable An alterable addressing EA mode is one that refers to alterable (writable) operands.
- Control A control addressing EA mode is one that refers to memory operands without an associated size.



### 3.5 PROGRAMMING VIEW OF ADDRESSING MODES

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

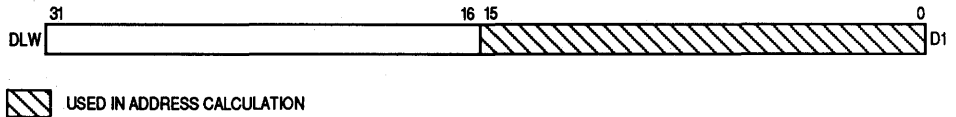
**Table 3-1. Effective Addressing Mode Categories**

Address Modes	Mode	Register	Data	Memory	Control	Alterable	Assembler Syntax
Data Register Direct	000	reg. no.	X	—	—	X	Dn
Address Register Direct	001	reg. no.	—	—	—	X	An
Address Register Indirect	010	reg. no.	X	X	X	X	(An)
Address Register Indirect with Postincrement	011	reg. no.	X	X	—	X	(An) +
Address Register Indirect with Predecrement	100	reg. no.	X	X	—	X	-(An)
Address Register Indirect with Displacement	101	reg. no.	X	X	X	X	(d <sub>16</sub> ,An)
Address Register Indirect with Index (8-Bit Displacement)	110	reg. no.	X	X	X	X	(d <sub>8</sub> ,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 <sub>16</sub> ,PC)
Program Counter Indirect with Index (8-Bit Displacement)	111	011	X	—	X	X	(d <sub>8</sub> ,PC,Xn)
Program Counter Indirect with Index (Base Displacement)	111	011	X	—	X	X	(bd,PC,Xn)
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. Since any of the data registers can be index registers, this provides a data register indirect form (Dn). Since either a data register or an address register can be used, the mode could be called register indirect (Rn). This addressing mode is an extension to the M68000 Family because the CPU32 can use both the data registers and the address registers to address memory. The capability of specifying the size and scale of an index register ( $Xn.SIZE*SCALE$ ) in these modes provides additional addressing flexibility. Using the SIZE parameter, either the entire contents of the index register can be used, or the least significant word can be sign extended to provide a 32-bit index value (refer to Figure 3-3).



**Figure 3-3. Using SIZE in the Index Selection**

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

The indirect suppressed index register mode (see Figure 3-5) 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.

Scaling provides an optional shifting of the value in an index register to the left by zero, one, two, or three bits before using it in the EA calculation (the actual value in the index register remains unchanged). This is equivalent to multiplying the register by one, two, four, or eight for direct subscripting into an array of elements of corresponding size using an arithmetic value residing

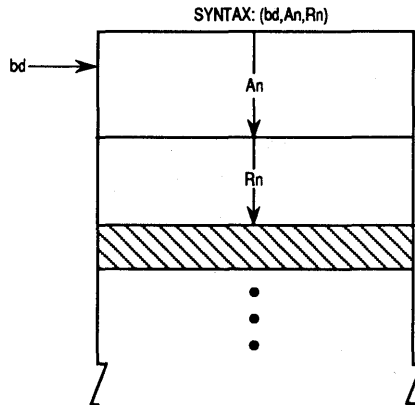


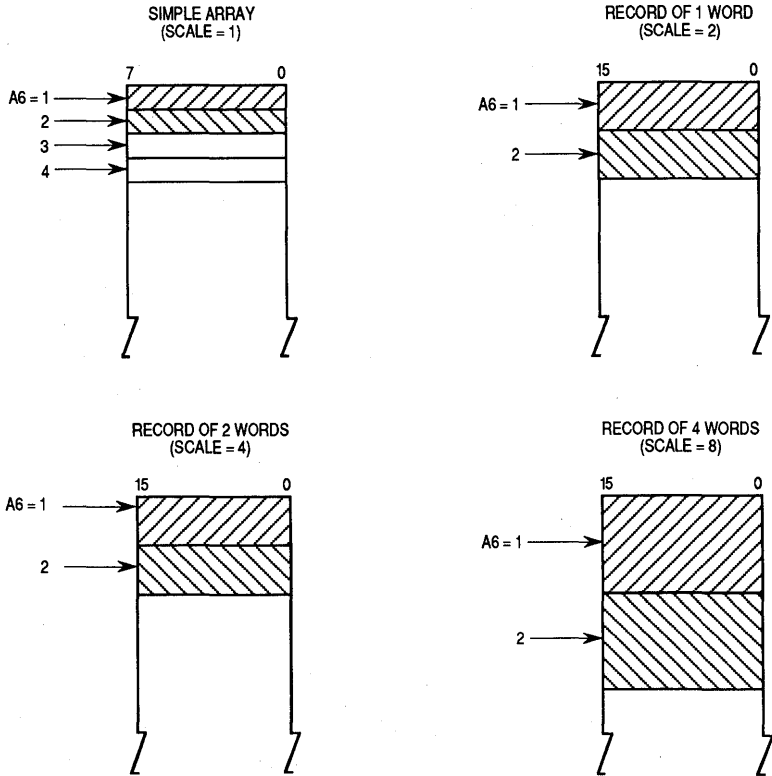
Figure 3-4. Using Absolute Address with Indexes

in any of the 16 general-purpose registers. 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.

### 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 the address register indirect if the register is an address register. If Rn is a data register, the assembler uses the address register indirect with index mode, using the data register as the indirect register, and suppresses the address register by setting the base suppress bit in the 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<sub>16</sub>,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.

SYNTAX: MOVE.W (A5,A6.L\*SCALE),(A7)  
WHERE:  
A5 = ADDRESS OF ARRAY STRUCTURE  
A6 = INDEX NUMBER OF ARRAY ITEM  
A7 = STACK POINTER



NOTE: Regardless of array structure, software increments indexed by the appropriate amount to point to next record.

Figure 3-5. Addressing Array Items

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 member in an upward compatible fashion. The user object code of each early member of the family is upward compatible with newer members and can be executed on the newer microprocessor without change. The address extension word(s) are encoded with information that allows the CPU32 to distinguish the new address extensions to the basic M68000 Family architecture. The address extension words for the early MC68000, MC68008, MC68010, and MC68020 microprocessors are shown in Figure 3-6. 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

**MC68000/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**

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.

The earlier microprocessors have no knowledge of the extension word formats implemented by newer processors, and, while they do detect illegal instructions, they do not decode invalid encodings of the extension words as exceptions.

## 3

### 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); 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.

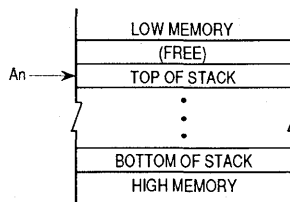
### 3.7.2 User Stacks

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

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

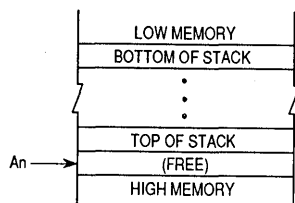
To implement stack growth from high to low memory, use  
 $-(An)$  to push data on the stack,  
 $(An)+$  to pull data from the stack.

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  
 $(A_n) +$  to push data on the stack,  
 $-(A_n)$  to pull data from the stack.

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

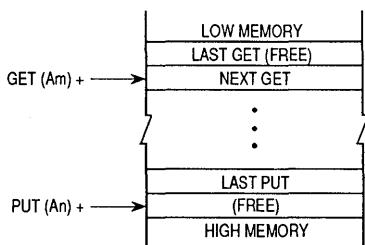


### 3.7.3 Queues

The user can implement queues with the address register indirect with post-increment or predecrement addressing modes. Using a pair of address registers (two of  $A_0$ – $A_6$ ), the user can implement a queue which is filled either from high to low memory or from low to high memory. Two registers are used because queues are pushed from one end and pulled from the other. One register,  $A_n$ , contains the “put” pointer; the other,  $A_m$ , the “get” pointer.

To implement growth of the queue from low to high memory, use  
 $(A_n) +$  to put data into the queue,  
 $(A_m) +$  to get data from the queue.

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

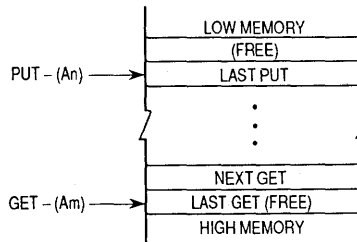


To implement the queue as a circular buffer, the relevant address register should be checked and adjusted, if necessary, before performing the "put" or "get" operation. The address register is adjusted by subtracting the buffer length (in bytes) from the register contents.

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" address register points to the last item placed in the queue, and the unchanged "get" address register points to the last item removed from the queue. After a "get" operation, the "get" address register points to the last item removed from the queue, and the unchanged "put" address register points to the last item placed in the queue, 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 adjusted, if necessary. 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 also included. After a summary of the instructions by category, a detailed description of the operation of each instruction is listed in alphabetical order. Programming information for specific instructions is included, followed by a description of condition code computation and an instruction format summary.

3

The CPU32 instructions form a set of tools which includes all the machine functions for 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 new instructions have been added to the M68000 instruction set for use in controller applications. They are low-power stop (LPSTOP) and table lookup and interpolate (TBL).

**4.1.1.1 LOW-POWER STOP (LPSTOP).** In applications where power consumption is a consideration, the CPU32 forces the device into a low-power standby mode when immediate processing is not required. The low-power stop mode is entered by executing the LPSTOP instruction. The processor remains in this mode until a user-specified or higher interrupt level or reset occurs.

**4.1.1.2 TABLE LOOKUP AND INTERPOLATE (TBL).** To maximize throughput for real-time applications, reference data is often precalculated and stored in memory for quick access. The storage of each data point would require an inordinate amount of memory. The TBL instruction, which requires only a sample of data points stored in the array, reduces memory requirements. Intermediate values are recovered with this instruction via linear interpolation.

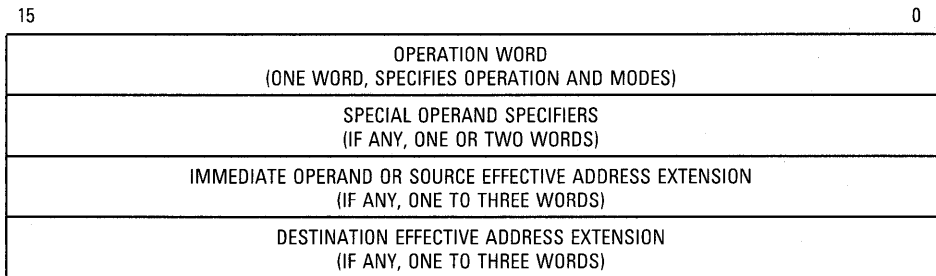
The CPU32 TBL instruction looks up the two table entries bounding the desired result and performs a linear interpolation between them. Byte, word, and long-word operand sizes are supported. The result is rounded according to the round-to-nearest algorithm. Optionally, byte and word results are left unrounded and returned along with the fractional portion of the calculated result. Software can make use of this extra "precision" to reduce the cumulative error in complex calculations. See **4.5 USING THE TABLE INSTRUCTION** for examples.

## 4.1.2 Unimplemented Instructions

Trap-on unimplemented instructions allow 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 have as many as seven words as shown in Figure 4-1. The first word of the instruction, called the operation word, specifies the length of the instruction and the operation to be performed. The remaining words, called extension words, further specify the instruction and operands. These words may be 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.



4

**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 selected is an address or data register and how the register is to be used. **SECTION 3 DATA ORGANIZATION AND ADDRESSING CAPABILITIES** contains detailed register information.

### 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
Logical	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.

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

An = any address register, A7–A0

Dn = any data register, D7–D0

Rn = any address or data register

CCR = condition code register (lower byte of status register)

cc = condition codes from CCR

SR = status register

SP = active stack pointer

USP = user stack pointer

SSP = supervisor stack pointer

DFC = destination function code register

SFC = source function code register

Rc = control register (VBR, SFC, DFC)

d = displacement;  $d_{16}$  is a 16-bit displacement

(ea) = effective address

list = list of registers (for example, D3–D0)

#(data) = immediate data; a literal integer

label = assembly program label

[7] = bit 7 of an operand

[31:24] = bits 31–24 of operand (high-order byte of a register)

X = extend (X) bit in CCR

N = negative (N) bit in CCR

V = overflow (V) bit in CCR

C = carry (C) bit in CCR

+ = arithmetic addition or postincrement

– = arithmetic subtraction or predecrement

\* = arithmetic multiplication

/ = arithmetic division or conjunction symbol

~ = invert; operand is logically complemented

$\wedge$  = logical AND

- V = logical OR
- ⊕ = logical exclusive OR
- Dc = data register D7–D0 used during compare
- Du = D7–D0 used during update
- Dr, Dq = data registers, remainder or quotient of divide
- Dh, Dl = data registers, high- or low-order 32 bits of product
- MSW = most significant word
- LSW = least significant word
- FC = function code
- {R/W} = read or write indicator
- [An] = address extensions

### 4.3.1 Data Movement Instructions

The MOVE instruction with its associated addressing modes 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 4-1 is a summary of the data movement operations.

**Table 4-1. Data Movement Operations**

Instruction	Operand Syntax	Operand Size	Operation
EXG	Rn, Rn	32	Rn ↔ Rn
LEA	<ea>, An	32	<ea> ↔ An
LINK	An, #(d)	16, 32	SP - 4 ↔ SP; An ↔ (SP); SP ↔ An, SP + d ↔ SP
MOVE MOVEA	<ea>, <ea> <ea>, An	8, 16, 32 16, 32 ↔ 32	source ↔ destination
MOVEM	list, <ea> <ea>, list	16, 32 16, 32 ↔ 32	listed registers ↔ destination source ↔ listed registers
MOVEP	Dn, (d <sub>16</sub> , An) (d <sub>16</sub> , 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

### 4.3.2 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, CPM, 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.

**Table 4-2. Integer Arithmetic Operations**

Instruction	Operand Syntax	Operand Size	Operation
ADD	Dn, (ea)	8, 16, 32	source + destination $\rightarrow$ destination
ADDA	(ea), Dn (ea), An	8, 16, 32 16, 32	
ADDI	#(data), (ea)	8, 16, 32	immediate data + destination $\rightarrow$ destination
ADDQ	#(data), (ea)	8, 16, 32	
ADDX	Dn, Dn -(An), -(An)	8, 16, 32 8, 16, 32	source + destination + X $\rightarrow$ destination
CLR	(ea)	8, 16, 32	0 $\rightarrow$ destination
CMP	(ea), Dn	8, 16, 32	destination — source
CMPA	(ea), An	16, 32	
CMPI	#(data), (ea)	8, 16, 32	destination — immediate data
CPM	(An)+, (An)+	8, 16, 32	destination — source
CMP2	(ea), Rn	8, 16, 32	lower bound $\leq$ Rn $\leq$ upper bound
DIVS/DIVU	(ea), Dn (ea), Dr:Dq	32/16 $\rightarrow$ 16:16 64/32 $\rightarrow$ 32:32	destination/source $\rightarrow$ destination (signed or unsigned)
DIVSL/DIVUL	(ea), Dq (ea), Dr:Dq	32/32 $\rightarrow$ 32 32/32 $\rightarrow$ 32:32	
EXT	Dn	8 $\rightarrow$ 16	sign extended destination $\rightarrow$ destination
EXTB	Dn Dn	16 $\rightarrow$ 32 8 $\rightarrow$ 32	
MULS/MULU	(ea), Dn (ea), Dl (ea), Dh:Dl	16 $\times$ 16 $\rightarrow$ 32 32 $\times$ 32 $\rightarrow$ 32 32 $\times$ 32 $\rightarrow$ 64	source * destination $\rightarrow$ destination (signed or unsigned)
NEG	(ea)	8, 16, 32	0 — destination $\rightarrow$ destination
NEGX	(ea)	8, 16, 32	0 — destination — X $\rightarrow$ destination
SUB	(ea), Dn	8, 16, 32	destination — source $\rightarrow$ destination
SUBA	Dn, (ea) (ea), An	8, 16, 32 16, 32	
SUBI	#(data), (ea)	8, 16, 32	destination — immediate data $\rightarrow$ destination
SUBQ	#(data), (ea)	8, 16, 32	
SUBX	Dn, Dn -(An), -(An)	8, 16, 32 8, 16, 32	destination — source — X $\rightarrow$ destination
TBLS/TBLU	(ea), Dx Dym:Dyn, Dx	8, 16, 32	Dyn — Dym $\rightarrow$ temp [temp * Dx (7:0)]/256 $\rightarrow$ temp Dym + temp $\rightarrow$ Dx

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 4-2 for a summary of the integer arithmetic operations.

### 4.3.3 Logical Instructions

The logical operation instructions (AND, OR, EOR, and NOT) perform logical operations with all sizes of integer data operands. A similar set of immediate instructions (ANDI, ORI, and EORI) provide these logical operations with all sizes of immediate data. The TST instruction arithmetically compares the operand with zero, placing the result in the condition code register. Table 4-3 summarizes the logical operations

**Table 4-3. Logical Operations**

Instruction	Operand Syntax	Operand Size	Operation
AND	$\langle ea \rangle, Dn$	8, 16, 32 $Dn, \langle ea \rangle$	source $\wedge$ destination $\blacktriangleright$ destination 8, 16, 32
ANDI	$\# \langle data \rangle, \langle ea \rangle$	8, 16, 32	immediate data $\wedge$ destination $\blacktriangleright$ destination
EOR	$Dn, \langle ea \rangle$	8, 16, 32	source $\oplus$ destination $\blacktriangleright$ destination
EORI	$\# \langle data \rangle, \langle ea \rangle$	8, 16, 32	immediate data $\oplus$ destination $\blacktriangleright$ destination
NOT	$\langle ea \rangle$	8, 16, 32	$\sim$ destination $\blacktriangleright$ destination
OR	$\langle ea \rangle, Dn$ $Dn, \langle ea \rangle$	8, 16, 32 8, 16, 32	source $\vee$ destination $\blacktriangleright$ destination
ORI	$\# \langle data \rangle, \langle ea \rangle$	8, 16, 32	immediate data $\vee$ destination $\blacktriangleright$ destination
TST	$\langle ea \rangle$	8, 16, 32	source $- 0$ to set condition codes



### 4.3.4 Shift and Rotate Instructions

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

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

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

**Table 4-4. Shift and Rotate Operations**

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

### 4.3.5 Bit Manipulation Instructions

Bit manipulation operations are accomplished using the following instructions: bit test (BTST), bit test and set (BSET), bit test and clear (BCLR), and bit test and change (BCHG). All bit manipulation operations can be performed on either registers or memory. The bit number is specified as immediate data or in a data register. Register operands are 32 bits long, and memory operands are 8 bits long. In Table 4-5, the summary of the bit manipulation operations, Z refers to bit 2, the zero bit of the status register.

Table 4-5. Bit Manipulation Operations

Instruction	Operand Syntax	Operand Size	Operation
BCHG	Dn,(ea) #(data),(ea)	8, 32 8, 32	$\sim$ ((bit number) of destination) $\blacktriangleright$ Z $\blacktriangleright$ bit of destination
BCLR	Dn,(ea) #(data),(ea)	8, 32 8, 32	$\sim$ ((bit number) of destination) $\blacktriangleright$ Z; 0 $\blacktriangleright$ bit of destination
BSET	Dn,(ea) #(data),(ea)	8, 32 8, 32	$\sim$ ((bit number) of destination) $\blacktriangleright$ Z; 1 $\blacktriangleright$ bit of destination
BTST	Dn,(ea) #(data),(ea)	8, 32 8, 32	$\sim$ ((bit number) of destination) $\blacktriangleright$ Z



### 4.3.6 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 4-6 is a summary of the BCD operations.

Table 4-6. Binary-Coded Decimal Operations

Instruction	Operand Syntax	Operand Size	Operation
ABCD	Dn,Dn -(An), -(An)	8 8	source <sub>10</sub> + destination <sub>10</sub> + X $\blacktriangleright$ destination
NBCD	(ea)	8	0 — destination <sub>10</sub> - X $\blacktriangleright$ destination
SBCD	Dn,Dn -(An), -(An)	8 8	destination <sub>10</sub> - source <sub>10</sub> - X $\blacktriangleright$ destination

## 4.3.7 Program Control Instructions

A set of subroutine call and return instructions and conditional and unconditional branch instructions perform program control operations. Table 4-7 summarizes these instructions.

**Table 4-7. Program Control Operations**

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

Letters cc in the instruction mnemonic opcodes specify testing one of the following condition codes:

CC — Carry clear	LS — Low or same
CS — Carry set	LT — Less than
EQ — Equal	MI — Minus
F — Never true*	NE — No equal
GE — Greater or equal	PL — Plus
GT — Greater than	T — Always true
HI — High	VC — Overflow clear
LE — Less or equal	VS — Overflow set

\*Not applicable to the Bcc instruction.

## 4.3.8 System Control Instructions

Privileged instructions, trapping instructions, and instructions that use or modify the condition code register provide system control operations. Table 4-8 summarizes these instructions. The preceding list of condition code

representations applies to the TRAPcc instruction. All of these instructions cause the processor to flush the instruction pipeline.

**Table 4-8. System Control Operations**

Instruction	Operand Syntax	Operand Size	Operation
<b>Privileged</b>			
ANDI	#{data},SR	16	immediate data $\wedge$ SR $\blacktriangleright$ SR
EORI	#{data},SR	16	immediate data $\oplus$ SR $\blacktriangleright$ SR
MOVE	(ea),SR SR,(ea)	16 16	source $\blacktriangleright$ SR SR $\blacktriangleright$ destination
MOVE	USP,An An,USP	32 32	USP $\blacktriangleright$ An An $\blacktriangleright$ USP
MOVEC	Rc,Rn Rn,Rc	32 32	Rc $\blacktriangleright$ Rn Rn $\blacktriangleright$ Rc
MOVES	Rn,(ea) (ea),Rn	8, 16, 32	Rn $\blacktriangleright$ destination using DFC source using SFC $\blacktriangleright$ Rn
ORI	#{data},SR	16	immediate data $\vee$ SR $\blacktriangleright$ SR
RESET	none	none	assert $\overline{\text{RESET}}$ line
RTE	none	none	(SP) $\blacktriangleright$ SR; SP + 2 $\blacktriangleright$ SP; (SP) $\blacktriangleright$ PC; SP + 4 $\blacktriangleright$ SP; restore stack according to format
STOP	#{data}	16	immediate data $\blacktriangleright$ SR; STOP
LPSTOP	#{data}	none	immediate data $\blacktriangleright$ SR; interrupt mask $\blacktriangleright$ EBI; STOP
<b>Trap Generating</b>			
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 $\blacktriangleright$ -(SSP); PC $\blacktriangleright$ -(SSP); SR $\blacktriangleright$ -(SSP); (vector) $\blacktriangleright$ 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 $\blacktriangleright$ SSP; vector offset $\blacktriangleright$ (SSP); SSP - 4 $\blacktriangleright$ SSP; PC $\blacktriangleright$ (SSP); SSP - 2 $\blacktriangleright$ SSP; SR $\blacktriangleright$ (SSP); illegal instruction vector address $\blacktriangleright$ PC
TRAP	#{data}	none	SSP - 2 $\blacktriangleright$ SSP; format and vector offset $\blacktriangleright$ (SSP) SSP - 4 $\blacktriangleright$ SSP; PC $\blacktriangleright$ (SSP); SR $\blacktriangleright$ (SSP); vector address $\blacktriangleright$ PC
TRAPcc	none #{data}	none 16, 32	if cc true, then TRAP exception
TRAPV	none	none	if V then take overflow TRAP exception
<b>Condition Code Register</b>			
ANDI	#{data},CCR	8	immediate data $\wedge$ CCR $\blacktriangleright$ CCR
EORI	#{data},CCR	8	immediate data $\oplus$ CCR $\blacktriangleright$ CCR
MOVE	(ea),CCR CCR,(ea)	16 16	source $\blacktriangleright$ CCR CCR $\blacktriangleright$ destination
ORI	#{data},CCR	8	immediate data $\vee$ CCR $\blacktriangleright$ CCR

## 4.4 INSTRUCTION DETAILS

The following paragraphs contain detailed information about each instruction in the CPU32 instruction set. First, the notation and the format of the instruction description is presented. Then each instruction is described in detail. The instruction descriptions are arranged alphabetically by instruction mnemonic.

### 4.4.1 Notation and Format

The instruction descriptions use notational conventions for the operands, the subfields and qualifiers, and the operations performed by the instructions. In the syntax descriptions, the left operand is the source operand, and the right operand is the destination operand. The notational conventions listed in **4.3 INSTRUCTION SUMMARY** apply. The following lists contain the additional notations used in the instruction descriptions.

Notation for operands:

PC — Program counter

SR — Status register

V — Overflow condition code

Immediate Data — Immediate data from the instruction

Source — Source contents

Destination — Destination contents

Vector — Location of exception vector

By convention, the destination operand is the operand on the right.

Notation for subfields and qualifiers:

$\langle \text{bit} \rangle$  of  $\langle \text{operand} \rangle$  — Selects a single bit of the operand

$\langle \langle \text{operand} \rangle \rangle$  — The contents of the referenced location

$\langle \text{operand} \rangle_{10}$  — The operand is binary-coded decimal; operations are performed in decimal

$\langle \langle \text{address register} \rangle \rangle$  — The register indirect operator, which indicates that  
–  $\langle \langle \text{address register} \rangle \rangle$  the operand register points to the memory location of the instruction operand. The optional mode qualifiers are –, +, (d), and (d,ix)  
 $\langle \langle \text{address register} \rangle \rangle +$

$\#xxx$  or  $\# \langle \text{data} \rangle$  — Immediate data that follows the instruction word(s)

Notations for operations that have two operands, written  $\langle \text{operand} \rangle \langle \text{op} \rangle \langle \text{operand} \rangle$ , where  $\langle \text{op} \rangle$  is one of the following:

- ↵ — The source operand is moved to the destination operand
- ↔ — The two operands are exchanged
- +
- The destination operand is subtracted from the source operand
- \*
- / — The source operand is divided by the destination operand
- < — Relational test, true if source operand is less than destination operand
- > — Relational test, true if source operand is greater than destination operand
- shifted by — The source operand is shifted or rotated by the number of positions specified by the second operand

4

Notation for single-operand operations:

- $\sim \langle \text{operand} \rangle$  — The operand is logically complemented
- $\langle \text{operand} \rangle \text{sign-extended}$  — The operand is sign extended; all bits of the upper portion are made equal to the high-order bit of the lower portion
- $\langle \text{operand} \rangle \text{tested}$  — The operand is compared to 0, and the condition codes are set appropriately

Notation for other operations:

- TRAP — Equivalent to Format/Offset Word ↵ (SSP); SSP - 2 ↵ SSP; PC ↵ (SSP); SSP - 4 ↵ SSP; SR ↵ (SSP); SSP - 2 ↵ SSP; (vector) ↵ PC
- STOP — Enter the stopped state; waiting for interrupts
- If  $\langle \text{condition} \rangle$  then  $\langle \text{operations} \rangle$  else  $\langle \text{operations} \rangle$  — The condition is tested. If true, the operations after "then" are performed. If the condition is false and the optional "else" clause is present, the operations after "else" are performed. If the condition is false and else is omitted, the instruction performs no operation. Refer to the description of Bcc instruction as an example.

## 4.4.2 Condition Code Register

The condition code register portion of the status register contains five bits:

X — Extend

V — Overflow

N — Negative

C — Carry

Z — Zero

The last four bits represent a condition of the result of a processor operation. Table 4-9 lists the effect of each instruction on these bits. The X bit is an operand for multiprecision computations; when it is used, it is set to the value of the carry bit. The carry bit and the multiprecision extend bit are separate in the M68000 Family to simplify programming techniques that use them. Refer to Table 4-4 as an example.

Program and system control instructions use certain combinations of these bits to control program and system flow. Table 4-9 lists the combinations of these bits and their interpretations.

**Table 4-9. Condition Code Computations (Sheet 1 of 2)**

Operations	X	N	Z	V	C	Special Definition
ABCD	*	U	?	U	?	C = Decimal Carry Z = $Z \wedge Rm \wedge \dots \wedge R0$
ADD, ADDI, ADDQ	*	*	*	?	?	V = $S\overline{m} \wedge Dm \wedge \overline{Rm} \vee S\overline{m} \wedge Dm \wedge Rm$ C = $S\overline{m} \wedge Dm \vee \overline{Rm} \wedge Dm \vee S\overline{m} \wedge Rm$
ADDX	*	*	?	?	?	V = $S\overline{m} \wedge Dm \wedge \overline{Rm} \vee S\overline{m} \wedge Dm \wedge Rm$ C = $S\overline{m} \wedge Dm \vee \overline{Rm} \wedge Dm \vee S\overline{m} \wedge Rm$ Z = $Z \wedge Rm \wedge \dots \wedge R0$
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) V (R = UB) C = (LB < = UB) $\wedge$ (IR < LB) V (R > UB) V (UB < LB) $\wedge$ (R > UB) $\wedge$ (R < LB)
SUB, SUBI, SUBQ	*	*	*	?	?	V = $S\overline{m} \wedge Dm \wedge \overline{Rm} \vee S\overline{m} \wedge Dm \wedge Rm$ C = $S\overline{m} \wedge Dm \vee Rm \wedge Dm \vee S\overline{m} \wedge Rm$
SUBX	*	*	?	?	?	V = $S\overline{m} \wedge Dm \wedge \overline{Rm} \vee S\overline{m} \wedge Dm \wedge Rm$ C = $S\overline{m} \wedge Dm \vee Rm \wedge Dm \vee S\overline{m} \wedge Rm$ Z = $Z \wedge Rm \wedge \dots \wedge R0$
CMP, CAMPI, CMPM	—	*	*	?	?	V = $S\overline{m} \wedge Dm \wedge \overline{Rm} \vee S\overline{m} \wedge Dm \wedge Rm$ C = $S\overline{m} \wedge Dm \vee Rm \wedge Dm \vee S\overline{m} \wedge Rm$
DIVS, DUVI	—	*	*	?	0	V = Division Overflow
MULS, MULU	—	*	*	?	0	V = Multiplication Overflow
SBCD, NBCD	*	U	?	U	?	C = Decimal Borrow Z = $Z \wedge Rm \wedge \dots \wedge R0$

**Table 4-9. Condition Code Computations (Sheet 2 of 2)**

Operations	X	N	Z	V	C	Special Definition
NEG	*	*	*	?	?	$V = Dm \wedge Rm$ $C = Dm \vee Rm$
NEGX	*	*	?	?	?	$V = Dm \wedge Rm$ $C = Dm \vee Rm$ $Z = Z \wedge Rm \wedge \dots \wedge R0$
ASL	*	*	*	?	?	$V = Dm \wedge (\overline{Dm-1} \vee \dots \vee \overline{Dm-r}) \vee \overline{Dm} \wedge$ $(\overline{Dm-1} \vee \dots \vee \overline{Dm-r})$ $C = \overline{Dm-r+1}$
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:**

— = Not Affected

U = Undefined, Result Meaningless

? = Other — See Special Definition

\* = General Case

X = C

N = Rm

Z =  $Rm \wedge \dots \wedge R0$

Sm = Source Operand — Most Significant Bit

Dm = Destination Operand — Most Significant Bit

Rm = Result Operand — Most Significant Bit

R = Register Tested

n = Bit Number

r = Shift Count

LB = Lower Bound

UB = Upper Bound

$\wedge$  = Boolean AND

$\vee$  = Boolean OR

$\overline{Rm}$  = NOT Rm



In the instruction set descriptions, the condition code register is shown as follows:

X	N	Z	V	C
*	U	*	U	*

where:

X (extend)

Set to the value of the C bit for many arithmetic operations. Otherwise not affected or set to a specified result.

N (negative)

Set if the most significant bit of the result is set. Cleared otherwise.

Z (zero)

Set if the result equals zero. Cleared otherwise.

V (overflow)

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

C (carry)

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

4

The following symbols are shown in the square representing each condition code:

\* = Set according to the result of the operation

— = Not affected by the operation

0 = Cleared

1 = Set

U = Undefined after the operation

### 4.4.3 Condition Tests

Table 4-10 lists the condition names, encodings, and tests for the condition branch and set instructions. The test associated with each condition is a logical formula using the current states of the condition codes. If this formula evaluates to one, the condition is true. If the formula evaluates to zero, the condition is false. For example, the T condition is always true, and the EQ condition is true only if the Z bit condition code is currently true.

**Table 4-10. Conditional Tests**

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

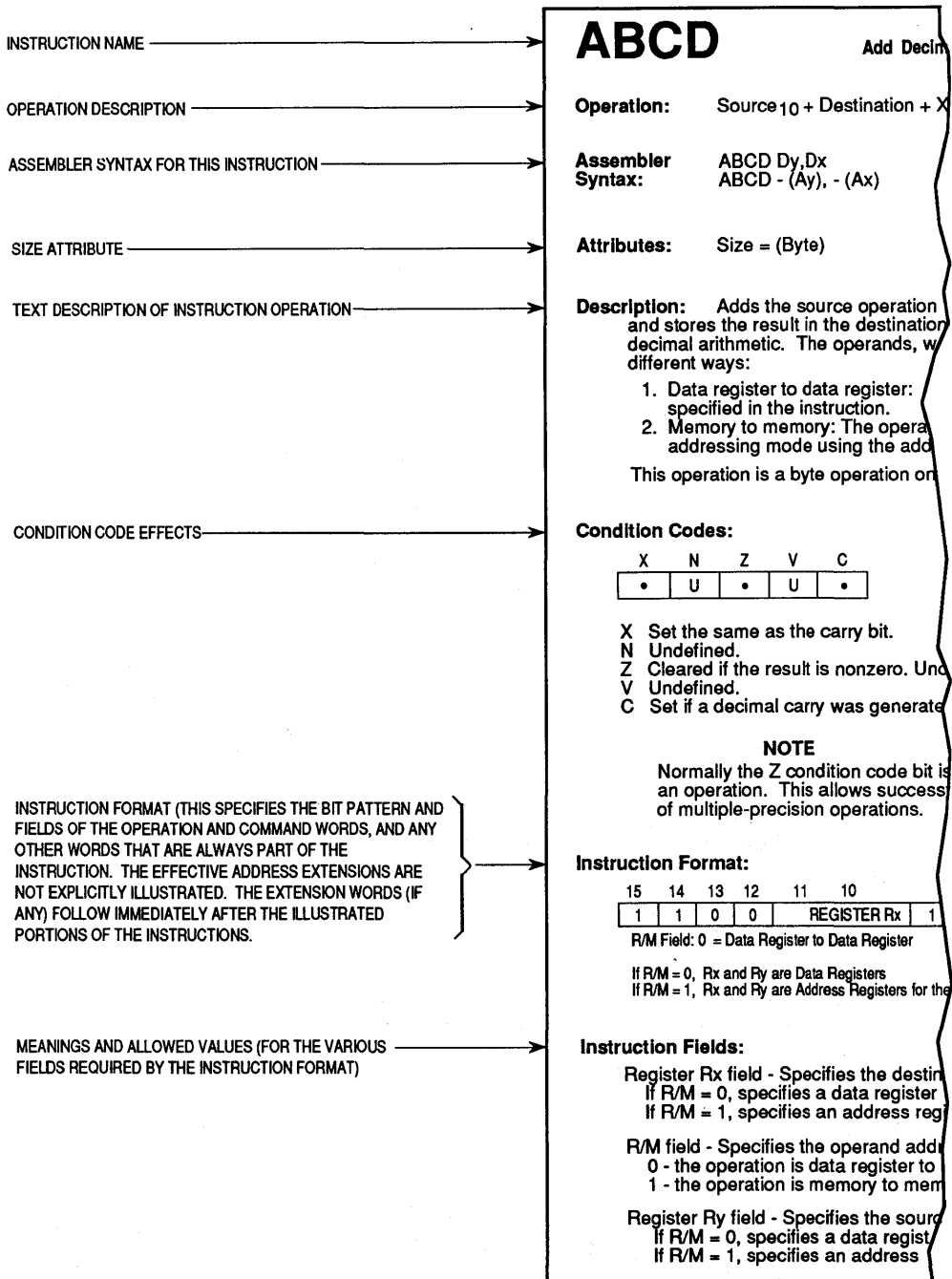
\*Not available for the Bcc instruction.

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

#### 4.4.4 Instruction Descriptions

Figure 4-2 shows the format of the instruction descriptions. 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 operands
- .W — Word operands
- .L — Long-word operands



**Figure 4-2. Instruction Description Format**

# ABCD

## Add Decimal with Extend

# ABCD

**Operation:** Source<sub>10</sub> + Destination<sub>10</sub> + X → Destination

**Assembler** ABCD D<sub>y</sub>,D<sub>x</sub>

**Syntax:** ABCD – (A<sub>y</sub>), – (A<sub>x</sub>)

**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: The operands are contained in the data registers specified in the instruction.
2. Memory to memory: The operands are addressed with the predecrement addressing mode using the address registers specified in the instruction.

This operation is a byte operation only.

### 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			

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Rx and Ry are Data Registers

If R/M = 1, Rx and Ry are Address Registers for the Predecrement Addressing Mode

### 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 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 Ry 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

# ADD

Add

# ADD

**Operation:** Source + Destination  $\rightarrow$  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 size of the operation may be specified as byte, word, or long. 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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	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> $\rightarrow$ <n>
100	101	110	<Dn> + <ea> $\rightarrow$ <ea>

# ADD

## Add

# ADD

Effective Address Field — Determines addressing mode:

- a. If the location specified is a source operand, all addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

\*Word and Long Only

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(d <sub>8</sub> ,PC,Xn)	111	011
(bd,PC,Xn)	111	011

4

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(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. 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

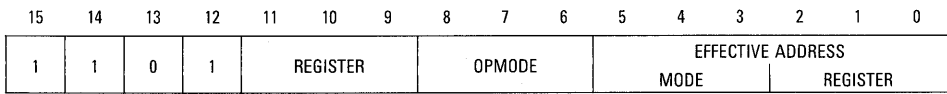
**Attributes:** Size = (Word, Long)

**Description:** Adds the source operand to the destination address register, and stores the result in the address register. The size of the operation may be specified as word or long. 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 the source operand. All addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

# ADDI

## Add Immediate

# ADDI

**Operation:** Immediate Data + Destination  $\rightarrow$  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 operation may be specified as byte, word, or long. The size of the immediate data matches 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:**

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

# ADDI

## Add Immediate

# ADDI

Effective Address field — Specifies the destination operand.

Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

4

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 of one to eight to the operand at the destination location. The size of the operation may be specified as byte, word, or long. Word and long operations are also 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:**

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		

**Instruction Fields:**

Data field — Three bits of immediate data, 7–0 (with the immediate value 0 representing a value of 8).

Size field — Specifies the size of the operation:

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

# ADDQ

## Add Quick

# ADDQ

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

\*Word and Long Only

# ADDX

## Add Extended

# ADDX

**Operation:** Source + Destination + X  $\rightarrow$  Destination

**Assembler:** ADDX Dy,Dx

**Syntax:** 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 different ways:

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

The size of the operation can be specified as byte, word, or long.

4

### 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		

### 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 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 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 the predecrement addressing mode.

# AND

## AND Logical

# AND

**Operation:** Source  $\wedge$  Destination  $\blacktriangleright$  Destination

**Assembler:** AND  $\langle ea \rangle, Dn$

**Syntax:** AND Dn,  $\langle ea \rangle$

**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 size of the operation can be specified as byte, word, or long. 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	$\langle ea \rangle \wedge \langle Dn \rangle \blacktriangleright Dn$
100	101	110	$\langle Dn \rangle \wedge \langle ea \rangle \blacktriangleright ea$



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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

4

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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  $\wedge$  Destination  $\rightarrow$  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 operation can be specified as byte, word, or long. The size of the immediate data matches the operation size.

**Condition Codes:**

X	N	Z	V	C
—	*	*	0	0

4

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

# ANDI

## AND Immediate

# ANDI

Effective Address field — Specifies the destination operand.

Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

4

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 Codes

# ANDI to CCR

**Operation:** SourceACCR  $\wedge$  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								BYTE DATA (8 BITS)							

# ANDI to SR

# ANDI to SR

## AND Immediate to the Status Register (Privileged Instruction)

**Operation:** If supervisor state  
then SourceASR  $\rightarrow$  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 (16 BITS)															

# ASL, ASR

## Arithmetic Shift

# ASL, ASR

**Operation:** Destination Shifted by <count> → Destination

**Assembler** ASd Dx,Dy

**Syntax:** ASd #(<data>),Dy

ASd <ea>

where d is direction, L or R

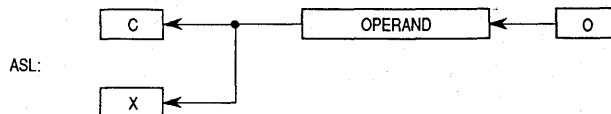
**Attributes:** Size = (Byte, Word, Long)

**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 the shifting of a register may be specified in two different ways:

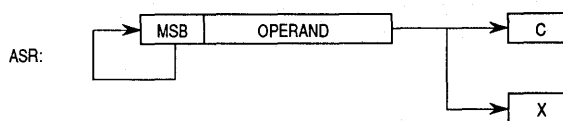
1. Immediate — The shift count is specified in the instruction (shift range, 8–1).
2. Register — The shift count is the value in the data register specified in instruction modulo 64.

The size of the operation can be specified as byte, word, or long. 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.



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.



# 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 at any time 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.

## 4

### 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 the 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—





# Bcc

## Branch Conditionally

# Bcc

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

**Assembler**

**Syntax:** Bcc (label)

**Attributes:** Size = (Byte, Word, Long)

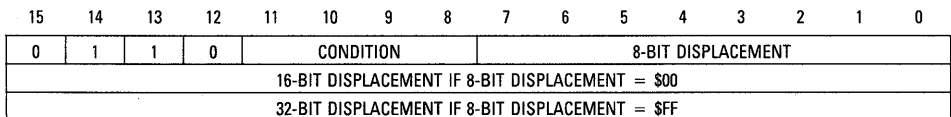
**Description:** If the specified condition is true, program execution continues at location (PC) + displacement. The PC contains the address of the instruction word of the Bcc instruction plus two. The displacement is a twos complement integer that represents the relative distance in bytes from the 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 code cc specifies one of the following conditions:

CC	carry clear	0100	$\overline{C}$	LS	low or same	0011	$C + Z$
CS	carry set	0101	C	LT	less than	1101	$N \cdot \overline{V} + \overline{N} \cdot V$
EQ	equal	0111	Z	MI	minus	1011	$\overline{N}$
GE	greater or equal	1100	$N \cdot V + \overline{N} \cdot \overline{V}$	NE	not equal	0110	$\overline{Z}$
GT	greater than	1110	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot V \cdot \overline{\overline{Z}}$	PL	plus	1010	$\overline{N}$
HI	high	0010	$\overline{C} \cdot \overline{Z}$	VC	overflow clear	1000	$\overline{V}$
LE	less or equal	1111	$Z + N \cdot \overline{V} + \overline{N} \cdot V$	VS	overflow set	1001	V

**Condition Codes:**

Not affected.

**Instruction Format:**



4

**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 the displacement when the 8-bit displacement field contains \$00.

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

**NOTE**

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

# BCHG

## Test a Bit and Change

# BCHG

**Operation:**  $\sim(\langle\text{number}\rangle \text{ of Destination}) \blacktriangleright Z;$   
 $\sim(\langle\text{number}\rangle \text{ of Destination}) \blacktriangleright \langle\text{bit number}\rangle \text{ of Destination}$

**Assembler:** BCHG Dn,(ea)

**Syntax:** BCHG #⟨data⟩,(ea)

**Attributes:** Size = (Byte, Long)

**Description:** Tests a bit in the destination operand and sets the Z condition code appropriately, then inverts the specified bit in the destination. 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 in 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 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
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

\*Long only; all others are byte only.



### 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							

### Instruction Fields (Bit Number Static):

Effective Address field — Specifies the destination location.

Only data alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

\*Long only; all others are byte only.

Bit Number field — Specifies the bit number.

# BCLR

## Test a Bit and Clear

# BCLR

**Operation:**  $\sim(\langle\text{bit number}\rangle \text{ of Destination}) \uparrow Z;$   
 $0 \uparrow \langle\text{bit number}\rangle \text{ of Destination}$

**Assembler** BCLR Dn,<ea>

**Syntax:** BCLR #<data>,<ea>

**Attributes:** Size = (Byte, Long)

**Description:** Tests a bit in the destination operand and sets the Z condition code appropriately, then clears the specified bit in the destination. 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 in 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 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	0	EFFECTIVE ADDRESS					
										MODE		REGISTER			

4

### 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 <sub>16</sub> ,An)	101	reg. number:An	(d <sub>16</sub> ,PC)	—	—
(dg,An,Xn)	110	reg. number:An	(dg,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—

\*Long only; all others are byte only.

4

### 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							

### Instruction Fields (Bit Number Static):

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 <sub>16</sub> ,An)	101	reg. number:An	(d <sub>16</sub> ,PC)	—	—
(dg,An,Xn)	110	reg. number:An	(dg,PC,Xn)	—	—
(bd,An,Xn)	110	reg. number:An	(bd,PC,Xn)	—	—

\*Long only; all others are byte only.

Bit Number field — Specifies the bit number.

# BGND

## Enter Background Mode

# BGND

**Operation:** IF (background mode enabled) THEN  
    Enter Background Mode  
ELSE  
    Format/Vector offset  $\blacktriangleright$  – (SSP)  
    PC  $\blacktriangleright$  – (SSP)  
    SR  $\blacktriangleright$  – (SSP)  
    (Vector)  $\blacktriangleright$  PC

### Assembler

**Syntax:** BGND

**Attributes:** Size = (Unsize)

4

**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 current 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 with the immediate data (value 0–7) on bits 2–4 of the address bus and zeros on bits 0 and 1 of the address bus.

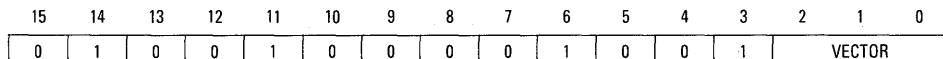
The breakpoint acknowledge cycle accesses the CPU space, addressing type 0, and provides the breakpoint number specified by the instruction on address lines A4–A2. If the external hardware terminates the cycle with  $\overline{DSACKx}$ s the data on the bus (an instruction word) is inserted into the instruction pipe and is executed after the breakpoint instruction. The breakpoint instruction requires a word to be transferred so if the first bus cycle accesses an 8-bit port, a second cycle is required. If the external logic terminates the breakpoint acknowledge cycle with  $\overline{BERR}$  (i.e., no instruction word available) the processor takes an illegal instruction exception. Refer to **7.2.5 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; that instruction is returned by the breakpoint acknowledge cycle.

**Condition Codes:**

Not affected.

**Instruction Format:**



**Instruction Fields:**

Vector field — Contains the immediate data, a value in the range of 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 the 8-bit displacement is equal to \$00.

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

## NOTE

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

# BSET

## Test a Bit and Set

# BSET

**Operation:**  $\sim\langle\text{bit number}\rangle$  of Destination  $\uparrow$  Z;  
 $1 \uparrow \langle\text{bit number}\rangle$  of Destination

**Assembler** BSET Dn,(ea)

**Syntax:** BSET #⟨data⟩,(ea)

**Attributes:** Size = (Byte, Long)

**Description:** Tests a bit in the destination operand and 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 either of two ways:

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

4

### 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 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
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

\*Long only; all others are byte only.

### 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	BIT NUMBER								

### Instruction Fields (Bit Number Static):

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

Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

\*Long only; all others are byte only.

Bit Number field — Specifies the bit number.

# BSR

## Branch to Subroutine

# BSR

**Operation:** SP - 4  $\blacktriangleright$  SP; PC  $\blacktriangleright$  (SP); PC + d  $\blacktriangleright$  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.

4

**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	1	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 the 8-bit displacement is equal to \$00.

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

**NOTE**

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

# BTST

## Test a Bit

# BTST

**Operation:** — ((bit number) of Destination)  $\nabla$  Z;

**Assembler** BTST Dn,(ea)

**Syntax:** 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 in a second word of the instruction.
2. Register — The specified data register contains the bit number.

4

### 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 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:

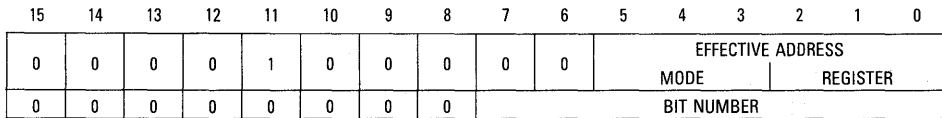
Addressing Mode	Mode	Register
Dn*	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

\*Long only; all others are byte only.



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



### Instruction Fields (Bit Number Static):

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

Bit Number field — Specifies the bit number.

# CHK

## Check Register Against Bounds

# CHK

**Operation:** If  $D_n < 0$  or  $D_n > \text{Source}$  then TRAP

**Assembler**

**Syntax:** CHK (ea),Dn

**Attributes:** Size = (Word, Long)

**Description:** Compares the value in the data register specified in 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  $D_n < 0$ ; cleared if  $D_n > \text{effective address operand}$ . Undefined otherwise.

Z Undefined.

V Undefined.

C Undefined.

**Instruction Format:**

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

**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 are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011



**Operation:** If  $R_n < \text{lower bound}$  or  
 $R_n > \text{upper bound}$   
 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 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. 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.



### 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			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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	111	010
(d <sub>8</sub> ,PC,Xn)	111	011
(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. The size of the operation may be specified as byte, word, or long.

**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			REGISTER		

**Instruction Fields:**

Size field — Specifies the size of the operation.

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

# CLR

## Clear an Operand

# CLR

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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 the condition codes according to the result; the data register is not changed. The size of the operation can be byte, word, or long.

**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:**

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			REGISTER		

**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
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(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

**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			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.

4

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011



# CMPI

## Compare Immediate

# CMPI

**Operation:** Destination – Immediate Data

**Assembler**

**Syntax:** CMPI #<data>,<ea>

**Attributes:** Size = (Byte, Word, Long)

**Description:** Subtracts the immediate data from the destination operand and sets the condition codes according to the result; the destination location is not changed. The size of the operation may be specified as byte, word, or long. The size of the immediate data matches 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:**

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)															

**Instruction Fields:**

Size field — Specifies the size of the operation:

00 — Byte operation

01 — Word operation

10 — Long operation

# CMPI

## Compare Immediate

# CMPI

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(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  $\blacktriangleright$  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 in the instruction. The size of the operation may be specified as byte, word, or long.

**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.

4

# CMP2

## Compare Register Against Bounds

# CMP2

**Operation:** Compare  $R_n < \text{lower-bound}$  or  
 $R_n > \text{upper-bound}$   
and Set Condition Codes

**Assembler**

**Syntax:** `CMP2 <ea>,Rn`

**Attributes:** Size = (Byte, Word, Long)

**Description:** Compares the value in  $R_n$  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  $R_n$  is a data register and the operation size is byte or word, only the appropriate low-order part of  $R_n$  is checked. If  $R_n$  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  $A_n$ .

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  $R_n$  is out of bounds.

**Condition Codes:**

X	N	Z	V	C
—	U	*	U	*

X Not affected.

N Undefined.

Z Set if  $R_n$  is equal to either bound. Cleared otherwise.

V Undefined.

C Set if  $R_n$  is out of bounds. Cleared otherwise.

### 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(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.

4

**Operation:** If condition false then  $(Dn - 1 \rightarrow Dn)$ ;  
If  $Dn \neq -1$  then  $PC + d \rightarrow PC$

**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.

4

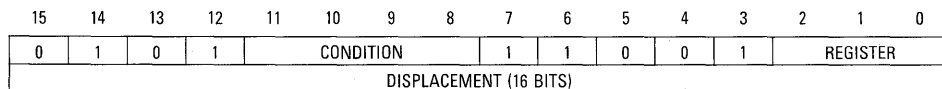
Condition code cc specifies one of the following conditions:

CC	carry clear	0100	$\bar{C}$	LS	low or same	0011	$C + Z$
CS	carry set	0101	C	LT	less than	1101	$N \cdot \bar{V} + \bar{N} \cdot V$
EQ	equal	0111	Z	MI	minus	1011	N
F	never equal	0001	0	NE	not equal	0110	$\bar{Z}$
GE	greater or equal	1100	$N \cdot V + \bar{N} \cdot \bar{V}$	PL	plus	1010	$\bar{N}$
GT	greater than	1110	$N \cdot V \cdot \bar{Z} + \bar{N} \cdot \bar{V} \cdot Z$	T	always true	0000	1
HI	high	0010	$\bar{C} \cdot \bar{Z}$	VC	overflow clear	1000	$\bar{V}$
LE	less or equal	1111	$Z + N \cdot \bar{V} + \bar{N} \cdot V$	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.

Register field — Specifies the data register used as the counter.

Displacement field — Specifies the number of bytes to branch.

**NOTES:**

1. The terminating condition is similar to the UNTIL loop clauses of high-level languages. For example: DBMI can be stated as "decrement and branch until minus".
2. Most assemblers accept DBRA for DBF for use when only 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 directly to the trailing DBcc instruction, the control count should equal the loop execution count. In this case, if a zero count occurs, the DBcc instruction does not branch, and the main loop is not executed.

# DIVS DIVSL

## Signed Divide

# DIVS DIVSL

**Operation:** Destination/Source  $\blacktriangleright$  Destination

**Assembler** DIVS.W (ea),Dn 32/16  $\blacktriangleright$  16r:16q

**Syntax:** DIVS.L (ea),Dq 32/32  $\blacktriangleright$  32q  
DIVS.L (ea),Dr:Dq 64/32  $\blacktriangleright$  32r:32q  
DIVSL.L (ea),Dr:Dq 32/32  $\blacktriangleright$  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 the remainder is in the upper word (most significant 16 bits) of the result. 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 and set before the instruction completes. If the instruction detects an overflow, it sets the overflow condition code, 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 the quotient is negative. Cleared otherwise. Undefined if overflow or divide by zero occurs.
- Z Set if the 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):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	REGISTER	1	1	1	EFFECTIVE ADDRESS							
									MODE	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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

## NOTE

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

### 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 <sub>16</sub> ,An)	101	reg. number:An	(d <sub>16</sub> ,PC)	111	010
(dg,An,Xn)	110	reg. number:An	(dg,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 comes 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.

# DIVU DIVUL

## Unsigned Divide

# DIVU DIVUL

**Operation:** Destination/Source  $\rightarrow$  Destination

**Assembler** DIVU.W  $\langle ea \rangle, Dn$  32/16  $\rightarrow$  16r:16q

**Syntax:** DIVU.L  $\langle ea \rangle, Dq$  32/32  $\rightarrow$  32q

DIVU.L  $\langle ea \rangle, Dr:Dq$  64/32  $\rightarrow$  32r:32q

DIVUL.L  $\langle ea \rangle, Dr:Dq$  32/32  $\rightarrow$  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 the remainder is in the upper word (most significant 16 bits) of the result.

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 and set before the instruction completes. If the instruction detects an overflow, it sets the overflow condition code, and the operands are unaffected.

### Condition Codes:

X	N	Z	V	C
—	*	*	*	0

X Not affected.

N Set if the quotient is negative. Cleared otherwise. Undefined if overflow or divide by zero occurs.

Z Set if the 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):

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			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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

### NOTE

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

### 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			0	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:

4

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d16,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d16,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

Register Dq field — Specifies a data register for the destination operand. The low-order 32 bits of the dividend comes 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 unsigned integer.

# EOR

## Exclusive OR Logical

# EOR

**Operation:** Source  $\oplus$  Destination  $\blacktriangleright$  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 size of the operation may be specified to be byte, word, or long. 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

4

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 (word form):**

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		REGISTER			

**Instruction Fields:**

Register field — Specifies any of the eight data registers.

Opmode field:

Byte	Word	Long	Operation
100	101	110	((ea) $\oplus$ (<Dn>) $\blacktriangleright$ <ea>)

# EOR

## Exclusive OR Logical

# EOR

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

4

### 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  $\oplus$  Destination  $\rightarrow$  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 operation may be specified as byte, word, or long. The size of the immediate data matches the operation size.

**Condition Codes:**

X	N	Z	V	C
—	*	*	0	0

4

- 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
Dn	000	reg. number:Dn
An	—	—
{An}	010	reg. number:An
{An}+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

4

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

# EORI to CCR

**Operation:** Source  $\oplus$  CCR  $\rightarrow$  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	0	BYTE DATA (8 BITS)						

4

# EORI to SR

Exclusive OR Immediate to the Status Register  
(Privileged Instruction)

# EORI to SR

**Operation:** If supervisor state  
then Source  $\oplus$  SR  $\rightarrow$  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.

4

## 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  $\rightarrow$  Destination

**Assembler Syntax:** EXT.W Dn extend byte to word  
EXT.L Dn extend word to long word  
EXTB.L Dn extend 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.

4

### 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:

- Opcode 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  $\blacktriangleright$  SSP; Vector Offset  $\blacktriangleright$  (SSP);  
SSP – 4  $\blacktriangleright$  SSP; PC  $\blacktriangleright$  (SSP);  
SSP – 2  $\blacktriangleright$  SSP; SR  $\blacktriangleright$  (SSP);  
Illegal Instruction Vector Address  $\blacktriangleright$  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

4

**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  $\blacktriangleright$  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:**

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		

**Instruction Fields:**

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

4

# JSR

## Jump to Subroutine

# JSR

**Operation:** SP – 4  $\blacktriangleright$  Sp; PC  $\blacktriangleright$  (SP)  
Destination Address  $\blacktriangleright$  PC

**Assembler**

**Syntax:** JSR <ea>

**Attributes:** Unsized

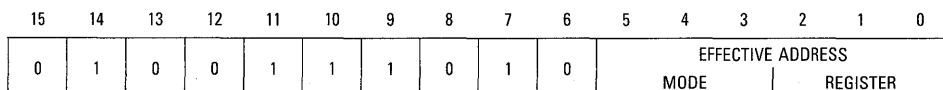
**Description:** Pushes the long word address of the instruction immediately following the JSR instruction onto the system stack. Program execution then continues at the address specified in the instruction.

**Condition Codes:**

Not affected.

4

**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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
–(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011



# LEA

## Load Effective Address

# LEA

**Operation:** (ea) ↗ An

**Assembler**

**Syntax:** LEA (ea),An

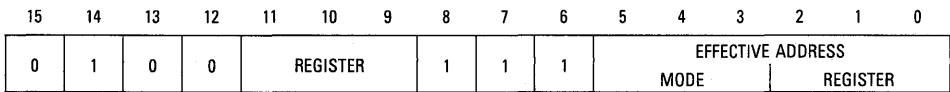
**Attributes:** Size = (Long)

**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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	111	010
(d <sub>8</sub> ,PC,Xn)	111	011
(bd,PC,Xn)	111	011

# LINK

## Link and Allocate

# LINK

**Operation:** Sp - 4  $\uparrow$  Sp; An  $\uparrow$  (SP);  
 SP  $\uparrow$  An; SP + d  $\uparrow$  SP

**Assembler**

**Syntax:** LINK An, #(displacement)

**Attributes:** Size = (Word, Long)

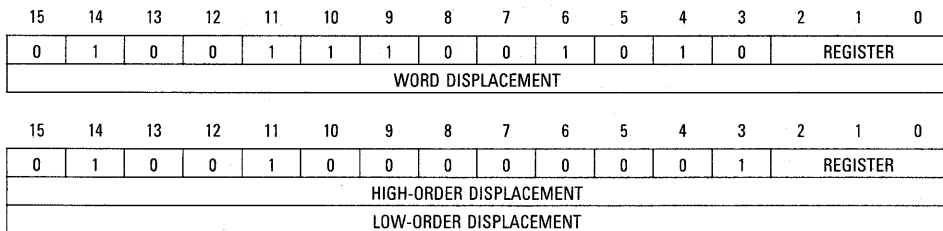
**Description:** Pushes the contents of the specified address register onto the stack. Then loads the updated stack pointer into the address register. Finally, adds the displacement value to the stack pointer. For word size operation, the displacement is the sign-extended word following the operation 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 in order to allocate stack area.

4

**Condition Codes:**

Not affected.

**Instruction Format:**



**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  
Immediate Data  $\blacktriangleright$  SR  
Interrupt Mask  $\blacktriangleright$  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 will occur if the trace state is on when the LPSTOP instruction is executed. If an interrupt request is asserted with a higher priority than 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 will cause 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** LSd Dx,Dy

**Syntax:** LSd #(data),Dy

LSd (ea)

where d is direction, L or R

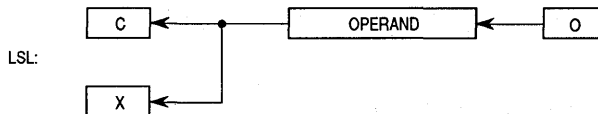
**Attributes:** Size = (Byte, Word, Long)

**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. The shift count for the shifting of a register is specified in two different ways:

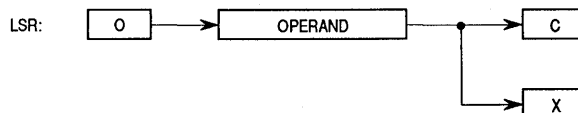
1. Immediate — The shift count (1-8) is specified in the instruction.
2. Register — The shift count is the value in the data register specified in the instruction modulo 64.

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 bit.



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 bit.



# 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 Field (Register Shifts):

Count/Register field:

If  $i/r = 0$ , this field contains the shift count. The values 1-7 represent shifts of 1-7; value of 0 specifies a shift count of 8.

If  $i/r = 1$ , the data register specified in this field 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An
([bd,An,Xn],od)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—
([bd,PC,Xn],od)	—	—

# MOVE

Move Data from Source to Destination

# MOVE

**Operation:** Source  $\rightarrow$  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. The size of the operation may be specified as byte, word, or long.

**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	0	SIZE		DESTINATION				SOURCE							
				REGISTER			MODE			MODE			REGISTER		

**Instruction Fields:**

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

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

4

# MOVE

## Move Data from Source to Destination

# MOVE

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d16,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d16,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d16,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d16,PC)	111	010
(dg,PC,Xn)	111	011
(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  $\rightarrow$  Destination

**Assembler**

**Syntax:** MOVEA <ea>,An

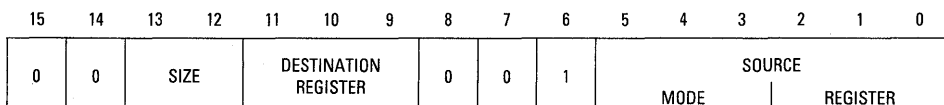
**Attributes:** Size = (Word, Long)

**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:**



**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 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
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(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)

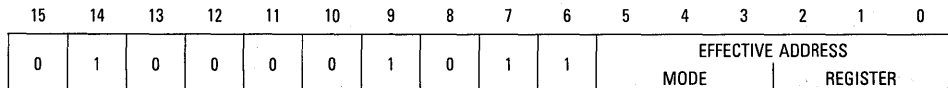
**Attributes:** Size = (Word)

**Description:** Moves the condition code bits (zero extended to word size) to the destination location. The operand size is a word. 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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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 Codes

# MOVE to CCR

**Operation:** Source  $\rightarrow$  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.

4

**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 Codes

# MOVE to CCR

### Instruction Fields:

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

4

### 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 is word length. 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	0	0	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

**Instruction Fields:**

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

**NOTE**

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

# MOVE to SR

# MOVE to SR

## Move to the Status Register (Privileged Instruction)

**Operation:** If supervisor state  
then Source  $\rightarrow$  SR  
else TRAP

**Assembler**

**Syntax:** MOVE <ea>,SR

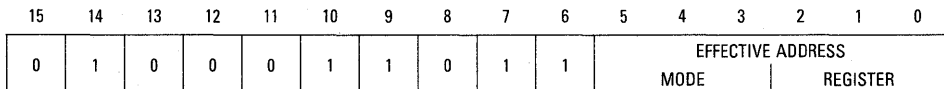
**Attributes:** Size = (Word)

**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 location of the source operand. Only data addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(d <sub>8</sub> ,PC,Xn)	111	011
(bd,PC,Xn)	111	011

# MOVE USP

Move User Stack Pointer  
(Privileged Instruction)

# MOVE USP

**Operation:** If supervisor state  
then USP  $\nabla$  An or An  $\nabla$  USP  
else TRAP

**Assembler** MOVE USP,An

**Syntax:** MOVE An,USP

**Attributes:** Size = (Long)

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

4

**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 (Privileged Instruction)

# MOVEC

**Operation:** If supervisor state  
 then Rc  $\rightarrow$  Rn or Rn  $\rightarrow$  Rc  
 else TRAP

**Assembler** MOVEC Rc,Rn

**Syntax:** 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. This 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  $\blacktriangleright$  Destination  
Source  $\blacktriangleright$  Registers

**Assembler** MOVEM register list,(ea)

**Syntax:** 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

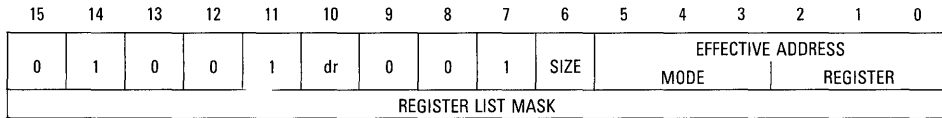
## Move Multiple Registers

# 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

4

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 the 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  $\nabla$  Destination

**Assembler** MOVEP Dx,(d,Ay)

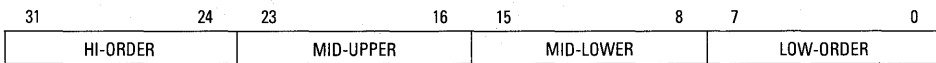
**Syntax:** MOVEP (d,Ay),Dx

**Attributes:** Size = (Word, Long)

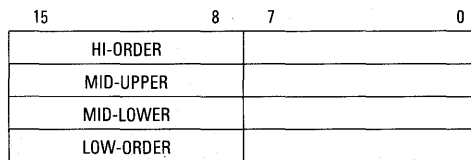
**Description:** Moves data between a data register and alternate bytes within the address space (typically assigned to a peripheral), starting at the location specified and incrementing by two. This instruction is designed for 8-bit peripherals on a 16-bit data bus. The high-order byte of the data register is transferred first and the low-order byte is transferred last. The memory address is specified in 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)



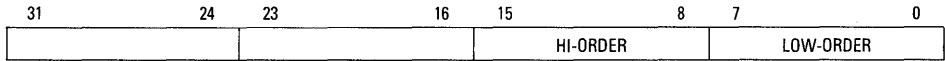
# MOVEP

## Move Peripheral Data

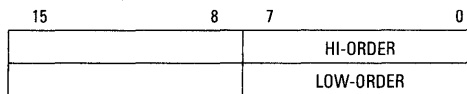
# MOVEP

Example: Word transfer to/from an odd address

### Byte Organization in Register



### Byte Organization in Memory (Low Address at Top)

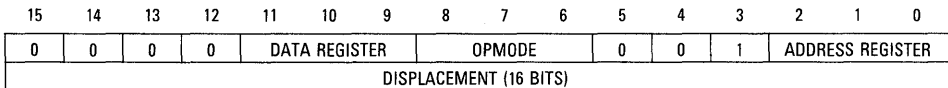


4

#### 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:**

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

**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 that 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 that 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				
A/D		REGISTER			dr	0	0	0	0	0	MODE		REGISTER		
										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 source or destination location within the alternate address space. Only memory alterable addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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 with the same address register as both source and destination

MOVES.x An,(An)+

MOVES.x An,-(An)

the value stored is undefined. The current implementations of the MC68010, CPU32, and MC68020 store the incremented or decremented value of An.



# MULS

## Signed Multiply

# MULS

**Operation:** Source \* Destination → Destination

**Assembler** MULS.W (ea),Dn 16x16 → 32

**Syntax:** MULS.L (ea),DI 32x32 → 32  
MULS.L (ea),Dh:DI 32 x 32 → 64

**Attributes:** Size = (Word, Long)

**Description:** Multiplies two signed operands yielding a signed result. This instruction has a word operand form and a long word operand form.

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.

### 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		REGISTER			

# MULS

## Signed Multiply

# MULS

### 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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

4

### 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 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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

# MULS

## Signed Multiply

# MULS

Register DI 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 DI.

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

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 = DI and Size is 1, the results of the operation are undefined. Otherwise, this field is unused.

# MULU

## Unsigned Multiply

# MULU

**Operation:** Source \* Destination  $\rightarrow$  Destination

**Assembler:** MULU.W (ea),Dn 16x16  $\rightarrow$  32

**Syntax:** MULU.L (ea),DI 32x32  $\rightarrow$  32

MULU.L (ea),Dh:DI 32x32  $\rightarrow$  64

**Attributes:** Size = (Word, Long)

**Description:** Multiplies two unsigned operands yielding an unsigned result. This instruction has a word operand form and a long word operand form.

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.

4

### 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):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	REGISTER				0	1	1	EFFECTIVE ADDRESS				
										MODE		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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

### 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 DI			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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

Register DI 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 DI.

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

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 = DI and Size is 1, the results of the operation are undefined. Otherwise, this field is unused.

# NBCD

## Negate Decimal with Extend

# NBCD

**Operation:** 0 – (Destination<sub>10</sub>) – X ↗ 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. This is a byte operation only.

### 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				

**Instruction Fields:**

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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. The size of the operation is specified as byte, word, or long.

**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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

# NEGX

Negate with Extend

# NEGX

**Operation:** 0 – (Destination) – X  $\blacktriangleright$  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. The size of the operation is specified as byte, word, or long.

**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 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	0	0	0	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

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

# NOP

None

# NOP

**Operation:** None

**Assembler**

**Syntax:** NOP

**Attributes:** Unsized

**Description:** Performs no operation. The processor state, other than the program counter, is 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.

4

**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:**     ~ Destination ♦ 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. The size of the operation is specified as byte, word, or long.

**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

**NOT****Logical Complement****NOT**

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

**4**

# OR

## Inclusive OR Logical

# OR

**Operation:** Source V Destination  $\blacktriangleright$  Destination

**Assembler** OR <ea>,Dn

**Syntax:** 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 size of the operation is specified as byte, word, or long. 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	0	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> V <Dn> $\blacktriangleright$ <Dn>
100	101	110	<Dn> V <ea> $\blacktriangleright$ <ea>



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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

4

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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

# ORI

**Operation:** Immediate Data V Destination  $\rightarrow$  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 operation is specified as byte, word, or long. The size of the immediate data matches 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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(bd,PC,Xn)	—	—

4

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 Codes

# ORI to CCR

**Operation:** Source V CCR  $\rightarrow$  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 one. Unchanged otherwise.
- N Set if bit 3 of immediate operand is one. Unchanged otherwise.
- Z Set if bit 2 of immediate operand is one. Unchanged otherwise.
- V Set if bit 1 of immediate operand is one. Unchanged otherwise.
- C Set 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	0	0	0	0	0	0	1	1	1	1	0	0
										BYTE DATA (8 BITS)					

# ORI to SR

Inclusive OR Immediate to the Status Register  
(Privileged Instruction)

# ORI to SR

**Operation:** If supervisor state  
then Source V SR  $\uparrow$  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.

4

**Condition Codes:**

X	N	Z	V	C
*	*	*	*	*

- X Set if bit 4 of immediate operand is one. Unchanged otherwise.
- N Set if bit 3 of immediate operand is one. Unchanged otherwise.
- Z Set if bit 2 of immediate operand is one. Unchanged otherwise.
- V Set if bit 1 of immediate operand is one. Unchanged otherwise.
- C Set 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	0	0	0	0	0	1	1	1	1	1	0	0
WORD DATA (16 BITS)															

# 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 is 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		

4

**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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

# RESET

Reset External Devices  
(Privileged Instruction)

# RESET

**Operation:** If supervisor state  
then Assert  $\overline{\text{RESET}}$  Line  
else TRAP

**Assembler  
Syntax:** RESET

**Attributes:** Unsized

**Description:** Asserts the  $\overline{\text{RESET}}$  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** R0d Dx,Dy

**Syntax:** R0d #<data>,Dy  
R0d (ea)  
where d is direction, L or R

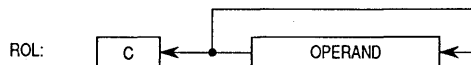
**Attributes:** Size = (Byte, Word, Long)

**Description:** Rotates the bits of the operand in the direction specified (L or R). The extend bit is not included in the rotation. The rotate count for the rotation of a register is specified in either of two ways:

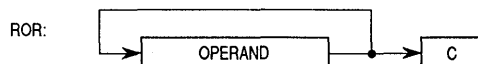
1. Immediate — The rotate count (1-8) is specified in the instruction.
2. Register — The rotate count is the value in the data register specified in the instruction, modulo 64.

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.

# 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

# ROXL, ROXR

Rotate with Extend

# ROXL, ROXR

**Operation:** Destination Rotated with X by <count> ↯ Destination

**Assembler** ROXd Dx,Dy

**Syntax:** ROXd #(data),Dy

ROXd (ea)

where d is direction, L or R

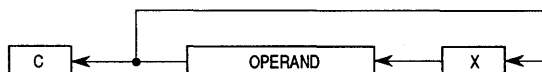
**Attributes:** Size = (Byte, Word, Long)

**Description:** Rotates the bits of the operand in the direction specified (L or R). The extend bit is included in the rotation. The rotate count for the rotation of a register is specified in either of two ways:

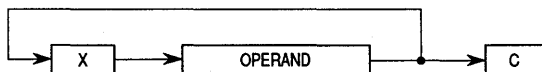
1. Immediate — The rotate count (1-8) is specified in the instruction.
2. Register — The rotate count is the value in the data register specified in the instruction, modulo 64.

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 the rotate 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. When the rotate count is zero, set to the value of the extend bit.

## 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.

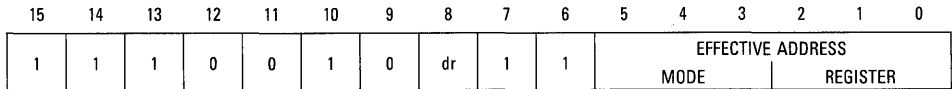
Register field — Specifies a data register to be rotated.

# ROXL, ROXR

Rotate with Extend

# ROXL, ROXR

## Instruction Format (Memory Rotate):



## 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

4

# RTD

## Return and Deallocate

# RTD

**Operation:** (SP)  $\downarrow$  PC; SP + 4 + d  $\downarrow$  SP

**Assembler**

**Syntax:** RTD #(displacement)

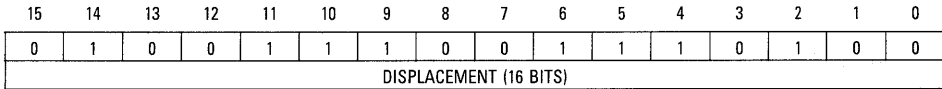
**Attributes:** Unsize

**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:**



4

**Instruction Field:**

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

**Operation:** If supervisor state  
 then (SP)  $\blacktriangleright$  SR; SP + 2  $\blacktriangleright$  SP; (SP)  $\blacktriangleright$  PC;  
 SP + 4  $\blacktriangleright$  SP;  
 restore state and deallocate stack according to (SP)  
 else TRAP

**Assembler**

**Syntax:** RTE

**Attributes:** Unsize

**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)  $\downarrow$  CCR; SP + 2  $\downarrow$  SP;  
(SP)  $\downarrow$  PC; SP + 4  $\downarrow$  SP

**Assembler**

**Syntax:** RTR

**Attributes:** Unsize

**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

4

# RTS

## Return from Subroutine

# RTS

**Operation:** (SP)  $\blacktriangleright$  PC; SP + 4  $\blacktriangleright$  SP

**Assembler**

**Syntax:** RTS

**Attributes:** Unsized

**Description:** Pulls the program counter value from the stack. 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	1

# SBCD

## Subtract Decimal with Extend

# SBCD

**Operation:** Destination<sub>10</sub> – Source<sub>10</sub> – X → Destination

**Assembler** SBCD Dx,Dy

**Syntax:** 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 in the instruction contain the operands.
2. Memory to memory: The address registers specified in the instruction access the operands from memory using the predecrement addressing mode.

This operation is a byte operation only.

### 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.

# SBCD

## Subtract Decimal with Extend

# SBCD

### 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

# Scc

**Operation:** If Condition True  
 then 1s → Destination  
 else 0s → Destination

**Assembler**

**Syntax:** Scc (ea)

**Attributes:** Size = (Byte)

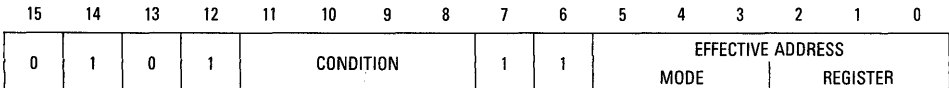
**Description:** Tests the specified condition code; if the condition is true, sets the byte specified by the effective address to TRUE (all ones). Otherwise, sets that byte to FALSE (all zeros). Condition code cc specifies one of the following conditions:

CC	carry clear	0100	$\bar{C}$	LS	low or same	0011	$C+Z$
CS	carry set	0101	C	LT	less than	1101	$N\cdot\bar{V} + \bar{N}\cdot V$
EQ	equal	0111	Z	MI	minus	1011	N
F	never true	0001	0	NE	not equal	0110	$\bar{Z}$
GE	greater or equal	1100	$N\cdot V + \bar{N}\cdot\bar{V}$	PL	plus	1010	$\bar{N}1$
GT	greater than	1110	$N\cdot V\cdot\bar{Z} + \bar{N}\cdot\bar{V}\cdot Z$	T	always true	0000	$\bar{V}$
HI	high	0010	$\bar{C}\cdot\bar{Z}$	VC	overflow clear	1000	V
LE	less or equal	1111	$Z + N\cdot\bar{V} + \bar{N}\cdot V$	VS	overflow set	1001	

**Condition Codes:**

Not affected.

**Instruction Format:**



### 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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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 (Privileged Instruction)

# STOP

**Operation:** If supervisor state  
then Immediate Data  $\blacktriangleright$  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  $\blacktriangleright$  Destination

**Assembler** SUB (ea),Dn

**Syntax:** 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 size of the operation is specified as byte, word, or long. 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 Format:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	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	((Dn)) – ((ea)) $\blacktriangleright$ (Dn)
100	101	110	((ea)) – ((Dn)) $\blacktriangleright$ (ea)



# SUB

## Subtract

# SUB

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
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

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

4

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(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  $\blacktriangleright$  Destination

**Assembler**

**Syntax:** SUBA (ea),An

**Attributes:** Size = (Word, Long)

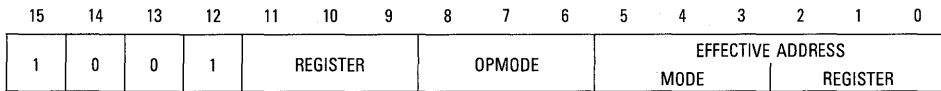
**Description:** Subtracts the source operand from the destination address register and stores the result in the address register. The size of the operation is specified as word or long. Word size source operands are sign extended to 32-bit quantities prior to the subtraction.

**Condition Codes:**

Not affected.

4

**Instruction Format:**



Opmode Field:

Word	Long	Operation
011	111	((An)) – ((ea)) $\blacktriangleright$ (An)

**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 the operation 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
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(bd,PC,Xn)	111	011

# SUBI

## Subtract Immediate SUBI

# 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 operation is specified as byte, word, or long. The size of the immediate data matches 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:**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(d <sub>8</sub> ,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(d <sub>8</sub> ,PC,Xn)	—	—
(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. The size of the operation is specified as byte, word, or long. 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:**

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

**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

# SUBQ

## Subtract Quick

# SUBQ

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#{data}	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

\*Word and Long Only

4

# SUBX

Subtract with Extend

# SUBX

**Operation:** Destination – Source – X  $\blacktriangleright$  Destination

**Assembler** SUBX Dx,Dy

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

The size of the operand is specified as byte, word, or long.

4

**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.

**Instruction Format:**

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



# SUBX

Subtract with Extend

# SUBX

## 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]  $\leftrightarrow$  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.

4

**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.

**Operation:**      Rounded:  $\text{ENTRY}(n) + \{(\text{ENTRY}(n+1) - \text{ENTRY}(n)) \cdot \text{Dx}[7:0]\} / 256 \blacktriangleright \text{Dx}$   
                          Unrounded:  $\text{ENTRY}(n) \cdot 256 + \{(\text{ENTRY}(n+1) - \text{ENTRY}(n)) \cdot \text{Dx}[7:0]\} \blacktriangleright \text{Dx}$

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

<b>Assembler</b>	TBLS.<size>	<ea>,Dx	*Result rounded
<b>Syntax:</b>	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 piecewise linear, compressed 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 linearized 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.

Signed table entries range from  $-2^{n-1}$  to  $2^{n-1}-1$ ; whereas, 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.

Rounding of the result is optionally selected via the "R" instruction field. If R=0 (TABLE), 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
$\leq -1/2$	-1
$> -1/2$ and $< 1/2$	+0
$\geq 1/2$	+1

4

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	Result
Word	Unaffected	Unaffected	Result	Result	Result
Long	Result	Result	Result	Result	Result

If R=1 (TABLENR), 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	Fraction
Word	Sign Extended	Result	Result	Fraction	Fraction
Long	Result	Result	Result	Fraction	Fraction

### NOTE

The long-word result contains only the least significant 24 bits of integer precision.

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 that 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  $0 \leq X \leq 255$ . 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 representation of a linearized function in which only every 256th value is actually stored in memory.

See **4.5 INSTRUCTION FORMAT SUMMARY** for examples on using the TBLS 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,  $-(2^{23}) \leq \text{Result} \leq (2^{23}) - 1$ . 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

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	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 destination location. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(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 the 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.

**Operation:**      Rounded:  
                           $ENTRY(n) + \{(ENTRY(n+1) - ENTRY(n)) * Dx[7:0]\} / 256 \blacktriangleright Dx$   
                          Unrounded:  
                           $ENTRY(n) * 256 + \{(ENTRY(n+1) - ENTRY(n)) * Dx[7:0]\} \blacktriangleright Dx$

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

<b>Assembler</b>	TBLU.<size>	<ea>,Dx	* Result rounded
<b>Syntax:</b>	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 unsigned table lookup and interpolate instruction, TBLU, allows the efficient use of piecewise linear, compressed 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 unsigned byte, word, or long-word table containing a linearized 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.

Signed table entries range from  $-2^{n-1}$  to  $2^{n-1}-1$ ; whereas, 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. The unsigned and unrounded table results will be zero extended instead of sign extended.

Rounding of the result is optionally selected via the "R" instruction field. If  $R=0$  (TABLE), 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
$\geq 1/2$	+1
$< 1/2$	+0

4

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	Result
Word	Unaffected	Unaffected	Result	Result	Result
Long	Result	Result	Result	Result	Result

If  $R=1$  (TBLUN), 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	Fraction
Word	Sign Extended	Result	Result	Fraction	Fraction
Long	Result	Result	Result	Fraction	Fraction

### NOTE

The long-word result contains only the least significant 24 bits of integer precision.



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 that 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  $0 \leq X \leq 255$ . 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 representation of a linearized function in which only every 256th value is actually stored in memory.

See **4.5 INSTRUCTION FORMAT SUMMARY** for examples on using the TBLU 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,  $-(2^{23}) \leq \text{Result} \leq (2^{23}) - 1$ . 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				0	R	0	1	Size	0	0	0	0	0	0	0

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	0	0	REGISTER Dym
0	REGISTER Dx				0	R	0	0	SIZE	0	0	0	0	REGISTER Dyn	

### Instruction Fields:

Effective address field (table lookup and interpolate mode only):

Specifies the destination location. Only control addressing modes are allowed as shown:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An)+	—	—
-(An)	—	—
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(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 the 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.

**Operation:** Destination Tested  $\blacktriangleright$  Condition Codes; 1  $\blacktriangleright$  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 or semaphore 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		

**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
Dn	000	reg. number:Dn
An	—	—
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	—	—
(d <sub>16</sub> ,PC)	—	—
(dg,PC,Xn)	—	—
(bd,PC,Xn)	—	—

# TRAP

## Trap

# TRAP

**Operation:** SSP - 2  $\blacktriangledown$  SSP; Format/Offset  $\blacktriangledown$  (SSP);  
 SSP - 4  $\blacktriangledown$  SSP; PC  $\blacktriangledown$  (SSP); SSP - 2  $\blacktriangledown$  SSP;  
 SR  $\blacktriangledown$  (SSP); Vector Address  $\blacktriangledown$  PC

**Assembler**

**Syntax:** TRAP #(<vector>)

**Attributes:** Unsized

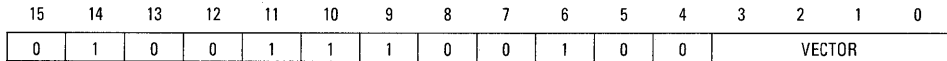
**Description:** Causes a TRAP #(<vector>) exception. The instruction adds the immediate operand (vector) of the instruction to 32 to obtain the vector number. The range of vector values is 0-15, which provides 16 vectors.

**Condition Codes:**

Not affected.



**Instruction Format:**



**Instruction Fields:**

Vector field — Specifies the trap vector to be taken.

# TRAPcc

Trap on Condition

# TRAPcc

**Operation:** If cc then TRAP

**Assembler** TRAPcc

**Syntax:** TRAPcc.W #(data)  
TRAPcc.L #(data)

**Attributes:** Unsized or Size = (Word, Long)

**Description:** If the specified condition is true, causes a TRAPcc exception. The vector number is 7. The processor pushes the address of the next instruction word (currently in the program counter) 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 should be placed in the next word(s) following the operation word and is available to the trap handler. Condition code cc specifies one of the following conditions.

CC	carry clear	0100	$\bar{C}$	LS	low or same	0011	$C+Z$
CS	carry set	0101	C	LT	less than	1101	$N\cdot\bar{V} + \bar{N}\cdot V$
EQ	equal	0111	Z	MI	minus	1011	$\bar{N}$
F	never true	0001	0	NE	not equal	0110	$\bar{Z}$
GE	greater or equal	1100	$N\cdot V + \bar{N}\cdot\bar{V}$	PL	plus	1010	$\bar{N}$
GT	greater than	1110	$N\cdot V\cdot Z + \bar{N}\cdot\bar{V}\cdot\bar{Z}$	T	always true	0000	1
HI	high	0010	$\bar{C}\cdot\bar{Z}$	VC	overflow clear	1000	$\bar{V}$
LE	less or equal	1111	$Z + N\cdot\bar{V} + \bar{N}\cdot V$	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:** Unsize

**Description:** If the overflow condition is set, causes a TRAPV exception (vector number 7). If the overflow condition 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  $\blacktriangleright$  Condition Codes

**Assembler**

**Syntax:** TST (ea)

**Attributes:** Size = (Byte, Word, Long)

**Description:** Compares the operand with zero and sets the condition codes according to the results of the test. The size of the operation is specified as byte, word, or long.

**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:**

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			

**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
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An)+	011	reg. number:An
-(An)	100	reg. number:An
(d <sub>16</sub> ,An)	101	reg. number:An
(dg,An,Xn)	110	reg. number:An
(bd,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#(data)	111	100
(d <sub>16</sub> ,PC)	111	010
(dg,PC,Xn)	111	011
(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 the 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	0	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	Coprocessor Operation

## CPU32 INSTRUCTIONS

### 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 (16 BITS)															

### 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

### Bit (Dynamic)

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

Type Field: 00=TST 10=CLR 01=CHG 11=SET

## MOVEP

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

## SUBI

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

## ADDI

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

### Bit (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	TYPE		EFFECTIVE ADDRESS					
										MODE			REGISTER		
0	0	0	0	0	0	0	0	BIT NUMBER							

Type Field: 00=TST 10=CLR 01=CHG 11=SET

### 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)															

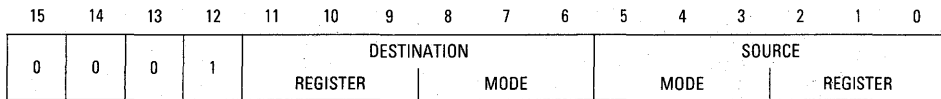
Size Field: 00=Byte 01=Word 10=Long

### 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

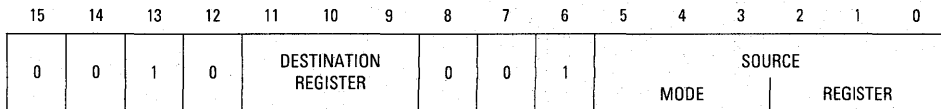
dr Field: 0=EA to Register 1=Register to EA

## MOVE Byte

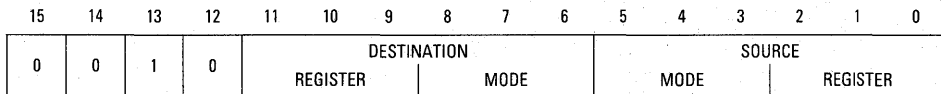


Note Register and Mode Locations

## MOVEA Long

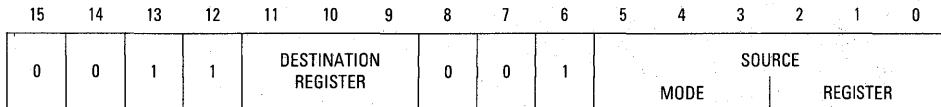


## MOVE Long

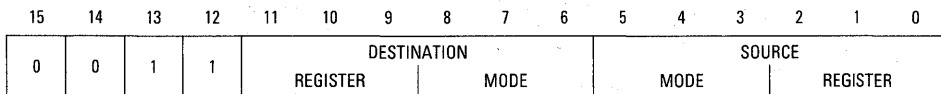


Note Register and Mode Locations

## MOVEA Word

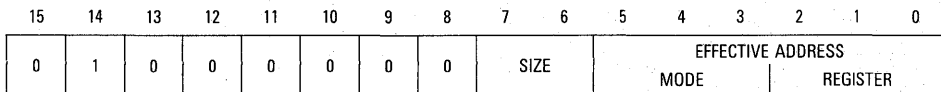


## MOVE Word



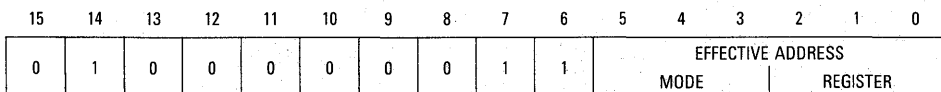
Note Register and Mode Locations

## NEGX

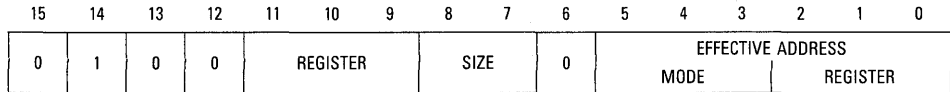


Size Field: 00=Byte 01=Word 10=Long

## MOVE from SR

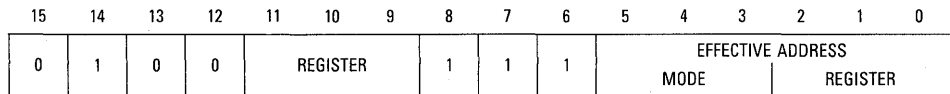


### CHK

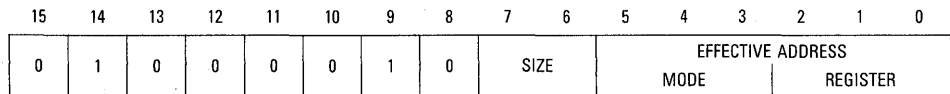


Size Field: 10=Long 11=Word

### LEA

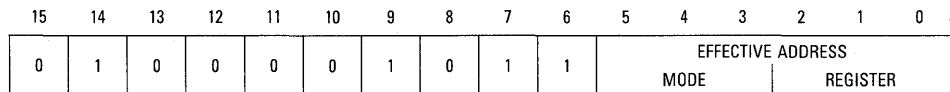


### CLR

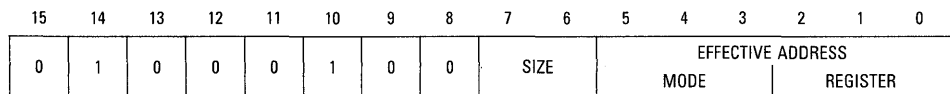


Size Field: 00=Byte 01=Word 10=Long

### MOVE from CCR

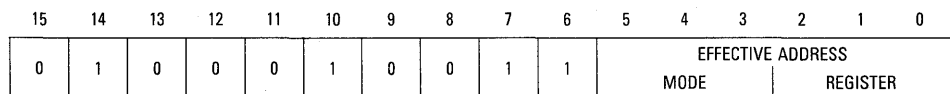


### NEG

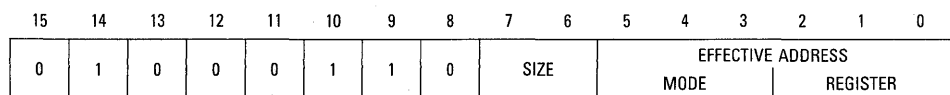


Size Field: 00=Byte 01=Word 10=Long

### MOVE to CCR



### NOT



Size Field: 00=Byte 01=Word 10=Long

### MOVE to SR

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

### NBCD

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				

### 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	OPCODE		0	0	0	REGISTER			

Opmode Field: 010=Extend Word 011=Extend Long 111=Extend Byte Long

### MOVEM Registers to EA

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: 0=Word Transfer 1=Long Transfer



## 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	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

## MULS/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 DI			TYPE	SIZE	0	0	0	0	0	0	0	REGISTER Dh		

Type Field: 0=MULU 1=MULS

Size Field: 0=Long-Word Product 1=Quad-Word Product

## DIVS/DIVU Long DIVUL/DIVSL

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			TYPE	SIZE	0	0	0	0	0	0	0	REGISTER Dr		

Type Field: 0=DIVU 1=DIVS

Size Field: 0=Long-Word Dividend 1=Quad-Word Dividend

## MOVEM EA to Registers

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: 0=Word Transfer 1=Long Transfer

**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 to 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	0	REGISTER		

**MOVE from 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	1	REGISTER		

**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

**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

### 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		

4

### 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; 1-7 represent immediate values of 1-7, and 0 represents 8.  
 Size Field: 00=Byte 01=Word 10=Long

### Scc

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

### DBcc

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 (16 BITS)															

### TRAPcc

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															

Opmode Field: 010=Word Operand 011=Long Operand 100=No Operand

### SUBQ

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

Data Field: Three bits of immediate data; 1-7 represent immediate values of 1-7, and 0 represents 8.  
 Size Field: 00=Byte 01=Word 10=Long

### Bcc

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

### BRA

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															

4

## BSR

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

## MOVEQ

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

Data Field: Data is sign extended to a long operand, and all 32 bits are transferred to the data register.

## OR

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

Opmode Field:

Byte	Word	Long	Operation
000	001	010	((ea))v((Dn)) ↯ (Dn)
100	101	110	((Dn))v((ea)) ↯ (ea)

## DIVS/DIVU Word

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

Type Field: 0 = DIVU 1 = DIVS

## SBCD

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		

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers

If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

## SUB

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

Opmode Field:

Byte	Word	Long	Operation
000	001	010	((Dn)) - ((ea)) ↯ (Dn)
100	101	110	((ea)) - ((Dn)) ↯ (ea)

## SUBA

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

Opmode Field:

Word	Long	Operation
011	111	((An) - ((ea)) $\downarrow$ (An)

## SUBX

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

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 are Data Registers

If R/M=1, Both Registers are Address Registers for the Predecrement Addressing Mode

## CMP

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			REGISTER		

Opmode Field:

Byte	Word	Long	Operation
000	001	010	((Dn) - ((ea))

## CMPA

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			REGISTER		

Opmode Field:

Word	Long	Operation
011	111	((An) - ((ea))

## EOR

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			REGISTER		

Opmode Field:

Byte	Word	Long	Operation
100	101	110	((ea) $\oplus$ ((Dn)) $\downarrow$ (ea)

## CMPM

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		

Size Field: 00=Byte 01=Word 10=Long

## AND

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		

Opmode Field:

Byte	Word	Long	Operation
000	001	010	$\{(\text{ea})\} \wedge \{(\text{Dn})\} \# (\text{Dn})$
100	101	110	$\{(\text{Dn})\} \wedge \{(\text{ea})\} \# (\text{ea})$

## MULS/MULU Word

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

Type Field: 0 = MULU 1 = MULS

## ABCD

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		

R/M Field: 0 = Data Register to Data Register 1 = Memory to Memory

If R/M = 0, Both Registers are Data Registers

If R/M = 1, Both Registers are Address Registers for the Predecrement Addressing Mode

## EXG Data Registers

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

## EXG Address Registers

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

## EXG Data Register and Address Register

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

## ADD

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

Opmode Field:

Byte	Word	Long	Operation
000	001	010	$\{(\text{ea})\} + \{(\text{Dn})\} \# (\text{Dn})$
100	101	110	$\{(\text{Dn})\} + \{(\text{ea})\} \# (\text{ea})$

## ADDA

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

Opmode Field:

Word	Long	Operation
011	111	((ea)) + ((An)) ↓ (An)

## ADDX

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		

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 are Data Registers

If R/M=1, Both Registers are Address Registers for the Predecrement Addressing Mode

4

## Shift/Rotate 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	TYPE		REGISTER		

Count/Register Field:

If I/R Field=0, Specifies Shift Count

If I/R Field=1, Specifies a Data Register That Contains the 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

Type Field: 00=Arithmetic Shift 01=Logical Shift 10=Rotate with Extend 11=Rotate

## Shift/Rotate Memory

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

Type Field: 00=Arithmetic Shift 01=Logical Shift 10=Rotate with Extend 11=Rotate

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															



## TBLS/TBLU (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	1	0	0	0	EFFECTIVE ADDRESS						
				REGISTER Dx		1	R	0	1	SIZE	0	0	0	0	0	0

Type Field: 0 = TBLS 1 = TBLU  
R Field: 0 = TBLS/TBLU 1 = TBLSN/TBLUN

## TBLS/TBLU (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	0	REGISTER Dym	
0	REGISTER Dx			TYPE	R	0	0	SIZE	0	0	0	REGISTER Dyn			

Type Field: 0 = TBLS 1 = TBLU  
R Field: 0 = TBLS/TBLU 1 = TBLSN/TBLUN

4

## 4.6 USING THE TABLE INSTRUCTION

The table lookup and interpolate instruction supports two variants: TBLS returns a rounded byte, word, or long-word signed result; TBLSN returns an unrounded byte, word, or long-word signed result; TBLU returns a rounded byte, word, or long-word unsigned result, and TBLUN returns an unrounded byte, word, or long-word unsigned result. All four variants support two locations for the interpolation data: an n-element table stored in memory, and a 2-element table stored in a pair of data registers. The latter form provides a means of calculating a surface interpolation between a pair of previously calculated linear interpolations.

The following examples provide some insight as to how the programmer may compress tables and/or force fewer interpolation levels between table entries. Example 1 (see Figure 4-3) demonstrates the table lookup and interpolation process for a 257-entry table allowing up to 256 interpolation levels between entries. Example 2 (see Figure 4-4) reduces the table length to four entries for the same set of data. Example 3 (see Figure 4-5) demonstrates how an 8-bit independent variable can be used with this instruction.

Two additional examples demonstrate how the programmer might use TBLSN to reduce cumulative errors in cases where several table lookup and interpolations are required in a single calculation. Example 4 demonstrates the case where the results from three table interpolations are added together to achieve the desired result. Example 5 illustrates the usefulness of TBLSN in doing surface (3D) interpolations.

## 4.6.1 Table Example 1: Standard Usage

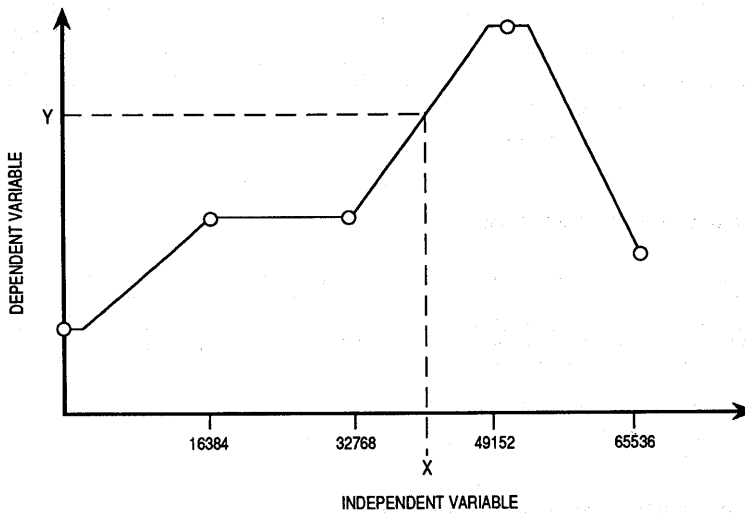


Figure 4-3. Table Example 1

The table consists of 257 word entries. The function is a straight line within the range  $32768 \leq X \leq 49152$  as shown on the plot. The table entries within the range of interest are as follows:

Entry No.	X Value	Y Value
128*	32768	1311
162	41472	1659
163	41728	1669
164	41984	1679
165	42240	1690
192*	49152	1966

\*For this example, these values have been chosen as the end points of the linear range. All table entries between these two points fall on the line.

The table instruction is executed with the following bit pattern in Dx:

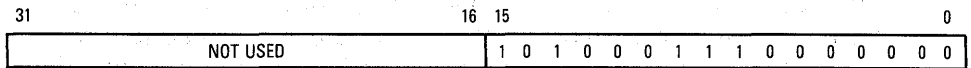


Table Entry Offset Dx  $\blacktriangledown$  (8:15) = \$A3 = 163

Interpolation Fraction  $\blacktriangledown$  Dx(0:7) = \$80 = 128

Using this information, the table instruction calculates the dependent variable, Y:

$$Y = 1669 + (128(1679 - 1669))/256 = 1674$$

4

### 4.6.2 Table Example 2: Compressed Table

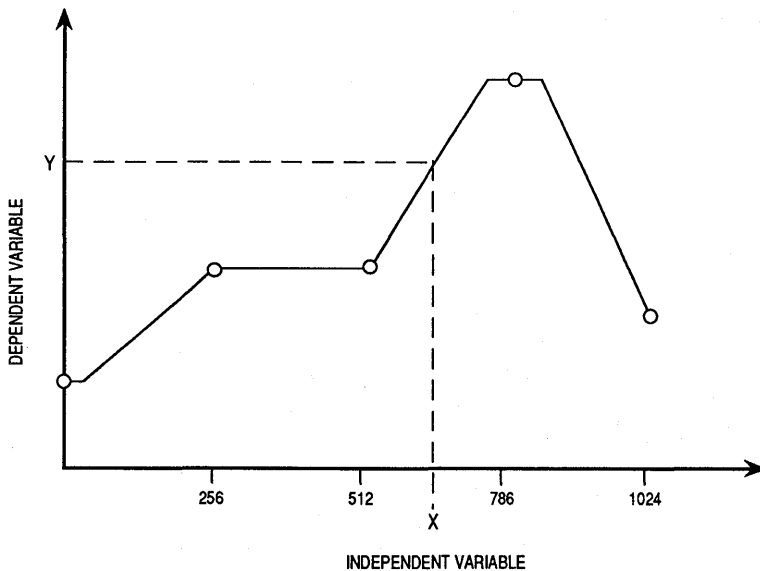


Figure 4-4. Table Example 2

In example 2, the data from Example 1 has been compressed into a 5-entry table by limiting the maximum value the independent value may take on. Instead of the normal range of  $0 \leq X = 65535$ ,  $X$  is limited to  $0 \leq X \leq 1023$ . Up to 256 levels of interpolation are still allowed between table entries.

### CAUTION

Only highly linear functions should compress the table to such extremes while continuing to allow so many levels of interpolation between entries. The table entries within the range of interest are as follows:

Entry No.	X Value	Y Value
2	512	1311
3	786	1966

Since the table was reduced from 257 to 5 entries, the 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:

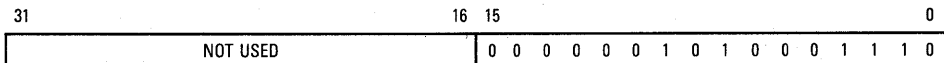


Table Entry Offset  $\blacktriangleright$   $Dx(8:15) = \$02 = 2$   
 Interpolation Fraction  $\blacktriangleright$   $Dx(0:7) = \$8E = 142$

Using this information, the table instruction calculates the dependent variable,  $Y$ :

$$Y = 1331 + (142(1966 - 1311))/256 = 1674$$

The chosen function was linear between the points entered into the table. Had another function been chosen, the interpolated values for Examples 1 and 2 might not have been identical.

### 4.6.3 Table Example 3: 8-Bit Independent Variable

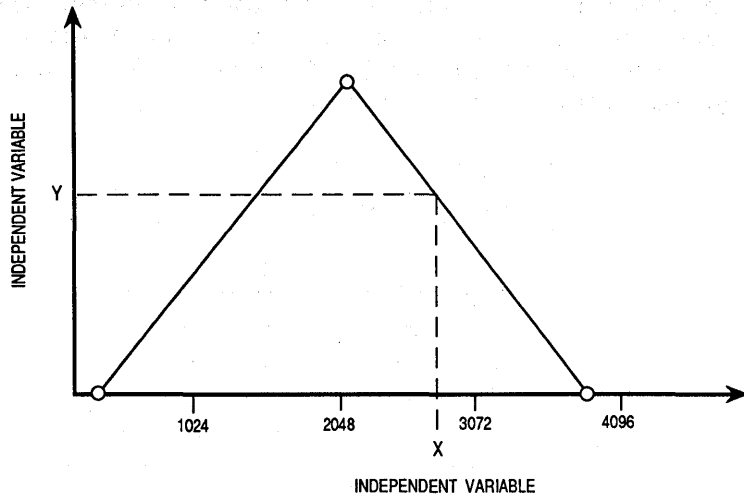


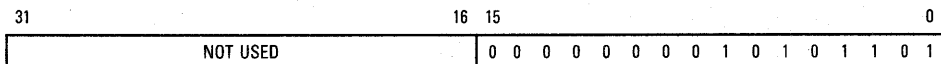
Figure 4-5. Table Example 3

In this example, the independent variable, X, is assumed to have been calculated as an 8-bit value, allowing 16 levels of interpolation on a 17-entry table. This value is passed to a subroutine that performs an interpolation using the following table values. An 8-bit result is returned. This example attempts to demonstrate the setup required to utilize the table instruction within the interpolation subroutine. The 17-entry table for the function plotted in Figure 4-5 contains the following data:

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

4

The first column represents the value passed to the subroutine, the second column represents the value expected by the table instruction, and the third column is the result to be returned. The following value has been calculated for the independent variable, X:



As an 8-bit value, using the upper four bits as the table offset and the lower four bits as the interpolation fraction, the following results should be obtained from a table lookup subroutine:

Table Entry Offset  $\blacktriangleright$   $Dx(4:7) = \$B = 11$   
 Interpolation Fraction  $\blacktriangleright$   $Dx(0:3) = \$D = 13$

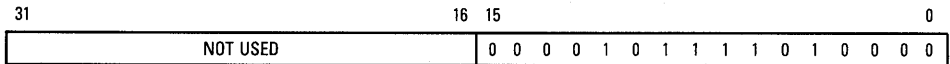
Thus, Y is calculated as follows:

$$Y = 80 + (13(64 - 80))/16 = 67$$

If the 8-bit value for X were to be used directly by the table instruction, the interpolation would incorrectly be between entries 0 and 1. To get the data in the correct format, it must be shifted to the left four places.

LSL.W #4,Dx

The new range for X is  $0 \leq X \leq 4096$ ; however, since the shift zero-filled the least significant digits, the interpolation fraction can only take on 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  $\blacktriangleright$   $Dx(8:15) = \$0B = 11$   
 Interpolation Fraction  $\blacktriangleright$   $Dx(0:7) = \$D0 = 208$

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 interpolations (TLIs) are performed and the results summed. The calculation is done twice; once with the result of each TLI rounded before the addition and again with only the final result

rounded. Assume that the result of the three interpolations are as follows. The "." indicates the binary radix point.

TLI #1	0100 0000 . 0111 0000
TLI #2	0011 1111 . 0111 0000
TLI #3	0000 0001 . 0111 0000

First, the results of each TLI are rounded using the round-to-nearest-even algorithm of TBLS. The following values represent the values that would be returned by TBLS:

TLI #1	0100 0000 .
TLI #2	0011 1111 .
TLI #3	0000 0001 .

Summing, the following result is obtained:

```

0100 0000 .
0011 1111 .
0000 0001 .
1000 0000 .

```

Using the same TLI results, the sum is calculated and then rounded according to the same algorithm.

```

0100 0000 . 0111 0000
0011 1111 . 0111 0000
0000 0001 . 0111 0000
1000 0001 0101 0000

```

Rounding yields:

```

1000 0001 .

```



Since the second result is the more desirable of the two, the following code sequence illustrates how the addition of a series of table interpolations could be performed without losing any precision in the intermediate results:

TBLSN.B	(ea),Dx	
TBLSN.B	(ea),Dm	
TBLSN.B	(ea),DI	
ADD.L	Dx,Dm	Long additions avoid
ADD.L	Dm,DI	problems with carry
ASR.L	8,DI	Move radix point
BCC.S	L1	Fraction MSB in carry
ADDQ	L 1,DI	Simple half round
L1 . . .		

## 4

### 4.6.5 Table Example 5: Surface Interpolations

The various forms of table can be used to perform surface (3D) TLLs. However, since the calculation must be split into a series of 2D TLLs, the possibility of losing precision in the intermediate results is possible. The following code sequence, incorporating both TBLS and TBLSN, eliminates this possibility.

MOVE.W	Dx, DI	Copy entry number and fraction #
TBLSN.B	(ea), Dx	
TBLSN.B	(ea), DI	
TBLS.W	Dx/DI, Dm	Surface interpolation w/Round
ASR.L	8, Dm	Read just the result
BCC.S	L1	No round necessary
ADDQ.<size>	#1, Dm	Half round up
L1 . . .		

At the beginning of this code sequence, Dx should contain the fraction and entry number for the two TLLs. Dm contains the fraction for the surface interpolation. The two (ea) fields in the calls to TBLSN point to consecutive columns in the 3D table. The size of TBLS must be word if the size of TBLSN was byte and long word if TBLSN was word. The increased size of TBLS is a result of increasing the number of significant digits to accommodate the scaled fractional result of the 2D TLLs.

## 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. Using this instruction in a series of subroutine calls results in a linked list of stack frames.

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

## 4.8 PIPELINE SYNCHRONIZATION WITH THE NOP INSTRUCTION

Although the no operation (NOP) instruction performs no visible operation, it forces synchronization of the instruction pipeline by waiting for all pending bus cycles to complete. All previous instructions complete execution before the NOP begins.



## SECTION 5

# PROCESSING STATES

This section describes the processing states of the CPU32. It describes the functions of the bits in the supervisor portion of the status register and the actions taken by the processor in response to exception conditions.

Unless the processor is halted, it is always in the normal, background, or exception processing state. Whenever the processor is executing instructions or fetching instructions or operands, it is in the normal processing state. The stopped state, which the processor enters when a STOP or LPSTOP instruction is executed, is a special case of the normal state in which no further bus cycles are generated.

### NOTE

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

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

A catastrophic system failure occurs whenever the processor receives a bus error or generates an address error while in the exception processing state. This type of failure halts the processor. For example, if during the exception processing of one bus error another bus error occurs, the CPU32 has not completed the transition to normal processing; therefore, the processor assumes that the system is not operational and halts. Only a reset external to

the CPU can restart a halted processor. (When the processor executes a STOP or LPSTOP instruction, it is in a special type of normal processing state — one without bus cycles. It is stopped, not halted.)

## 5.1 PRIVILEGE LEVELS

The processor operates at one of two levels of privilege — user or supervisor. The supervisor level has higher privileges than the user level. Not all instructions are permitted to execute in the lower privileged user level, but all instructions are available at the supervisor level. This scheme allows a separation of supervisor and user levels so the supervisor can protect system resources from uncontrolled access. The processor uses the privilege level indicated by the S bit in the status register to select either the user or supervisor privilege level and either the user stack pointer or a supervisor stack pointer for stack operations. The processor identifies a bus access (supervisor or user mode) via the function codes to maintain differentiation between supervisor and user levels.

In many systems, the majority of programs execute at the user level. User programs can access only their own code and data areas and can be restricted from accessing other information. The operating system, which typically 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

### 5.1.1 Supervisor Privilege Level

The supervisor level is the higher privilege level. For instruction execution, the supervisor state is determined by the S bit of the status register; if the S bit is set, the supervisor 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 FC2–FC0 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.

### 5.1.2 User Privilege Level

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

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

The bus cycles for an instruction executed at the user privilege level are classified as user references, and the values of the function codes on FC2–FC0 specify user address spaces. While the processor is at the user level, references to the system stack pointer implicitly, or to address register seven (A7) explicitly, refer to the user stack pointer (USP).

### 5.1.3 Changing Privilege Level

To change from the user to the supervisor privilege level, one of the conditions that causes the processor to perform exception processing must occur. Exception processing saves the current values of the S bit as well as the remainder of the status register on the supervisor stack, and then sets the S bit, forcing the processor into the supervisor privilege level. Execution of instructions continues at the supervisor level to process the exception condition.

To return to the user privilege level, a system routine must execute one of the following instructions: MOVE to SR, ANDI to SR, EORI to SR, ORI to SR, or RTE. These instructions execute at the supervisor privilege level and can modify the S bit of the status register. After these instructions execute, the instruction pipeline is flushed and is refilled from the appropriate address space.

The RTE instruction returns to the program that was executing when the exception occurred. It restores the exception stack frame saved on the supervisor stack. If the frame on top of the stack was generated by an interrupt, breakpoint, trap, or instruction exception, the RTE instruction restores the status register and program counter to the values saved on the supervisor

stack. The processor then continues execution at the restored program counter address and at the privilege level determined by the S bit of the restored status register. If the frame on top the stack was generated by a bus fault (bus error or address error exception), the RTE instruction restores the entire saved processor state from the stack.

## 5.2 ADDRESS SPACE TYPES

The processor specifies a target address space for every bus cycle with the function code signals according to the access required. In addition to distinguishing between supervisor/user and program/data, the processor can identify special processor cycles, such as the interrupt acknowledge cycle or the LPSTOP broadcast cycle. Table 5-1 lists the access types defined for the CPU32 and the corresponding values of function codes FC2–FC0.

The memory locations of user program and data accesses are not predefined. Neither are the locations of supervisor data space. During reset, the first two long words beginning at memory location zero in the supervisor program space are used for processor initialization. No other memory locations are explicitly defined by the CPU32.

A function code of \$7 (FC2:FC0 = 111) selects the CPU address space. This special address space does not contain instructions or operands but is reserved for special processor functions. The processor uses accesses in this space to communicate with external devices for special purposes. For example, all M68000 processors use the CPU space for interrupt acknowledge cycles. The CPU32 also generates CPU space accesses for breakpoint acknowledge and the LPSTOP broadcast.

**Table 5-1. Address Space Encodings**

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

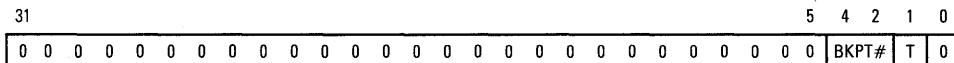
\*Address space 3 is reserved for user definition; 0 and 4 are reserved for future use by Motorola.

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

The following formats represent the information presented on the address bus for various CPU space transactions. The CPU space type field resides on bits A19–A16 and indicates the CPU space function the processor is performing. The functionality of address bits A5–A0 is particular to the type of function being performed. Since address bits A31–A20 are not present on all implementations of M68000 processors, they cannot be essential for decoding CPU space transactions and are insignificant. Currently, only five of the possible 16 encodings of A19–A16 are defined: 0000, 0001, 0010, 0011, and 1111. Of these, only 0000, 0011, and 1111 are supported by the CPU32.

### 5.2.1 Type 0000 — Breakpoint

This CPU space type is used as a breakpoint acknowledge.



The BKPT# field on A4–A2 indicates the breakpoint number. Software breakpoints will set this value to the number of the executing breakpoint instruction. Hardware breakpoints always set BKPT# to 7 (%111).

The T bit on A1 designates the type of breakpoints. T=0 indicates a software breakpoint; T=1 indicates a hardware breakpoint.

### 5.2.2 Type 0001 — MMU Access

This type of access is not supported by the CPU32 processor. This space is reserved for future use.

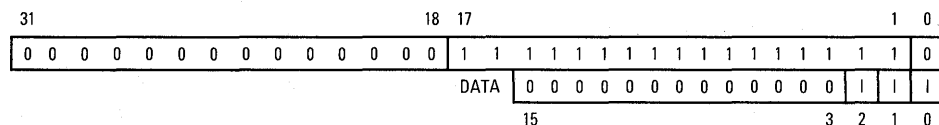
### 5.2.3 Type 0010 — Coprocessor Access

This type of access is not supported by the CPU32 processor. This space is reserved for future use.

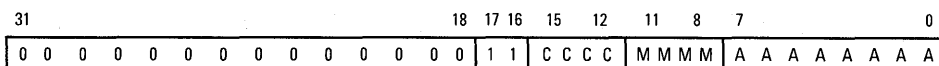


## 5.2.4 Type 0011 — Internal Register Access

This CPU space type is used to access certain critical system configuration or control registers. There is a single register on the CPU32 which resides in CPU space, the interrupt mask register in the external bus interface. This register is written to by the LPSTOP instruction to mask off external interrupts below the CPU mask level while in STOP mode. The levels on A3–A1 indicate the encoded CPU interrupt mask level.



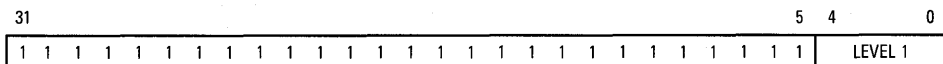
These registers will also reside in CPU space 3 and are only accessible through the MOVES command. The general format of this CPU space type is defined as follows:



A15–A12 is used as a 1 of 16 external chip select.  
 A11–A8 is used as a 1 of 16 internal module select.  
 A7–A0 is used as a 1 of 256 module register address.

## 5.2.5 Type 1111 — Interrupt Acknowledge

This CPU space type is used for interrupt acknowledge. The levels on A3–A1 indicate the encoded interrupt level being acknowledged.



## 5.3 EXCEPTION PROCESSING

An exception is defined as a special condition that pre-empts normal processing. Both internal and external conditions cause exceptions. External conditions that cause exceptions are interrupts from external devices, bus errors, breakpoint requests, and reset. Instructions, address errors, tracing, and breakpoints are internal conditions that cause exceptions. The TRAP, TRAPcc, TRAPV, CHK, CHK2, RTE, and DIV instructions can all generate exceptions as part of their normal execution. In addition, illegal instructions and privilege violations cause exceptions.

Exception processing, which is the transition from the normal processing of a program to the processing required for the exception condition, involves the exception vector table and an exception stack frame. The following paragraphs describe the exception vector table and a generalized exception stack frame. Exception processing is discussed in detail in **SECTION 6 EXCEPTION PROCESSING**.

### 5.3.1 Exception Vectors

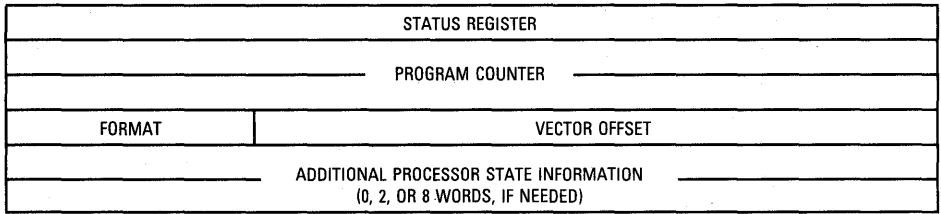
The vector base register (VBR) contains the base address of the 1024-byte exception vector table, which consists of 256 exception vectors. Exception vectors contain the memory addresses of routines that begin execution at the completion of exception processing. These routines perform a series of operations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

The address of an exception vector is derived from an 8-bit vector number and the VBR. The vector numbers for some exceptions are obtained from an external device; other numbers are supplied automatically by the processor. The processor multiplies the vector number by four to calculate the vector offset, which it adds to the VBR. The sum is the memory address of the vector. All exception vectors are located in supervisor data space, except the reset vector, which is located in supervisor program space. Only the initial reset vector is fixed in the processor's memory map; once initialization is complete, there are no fixed assignments. Since the VBR provides the base address of the vector table, the vector table can be located anywhere in memory; it can even be dynamically relocated for each task that is executed by an operating system. Details of exception processing are provided in **SECTION 6 EXCEPTION PROCESSING**, and Table 6-1 lists the exception vector assignments.

### 5.3.2 Exception Stack Frame

Exception processing saves the most volatile portion of the current processor context on the top of the supervisor stack. This context is organized in a format called the exception stack frame. This information always includes a copy of the status register, the program counter, the vector offset of the vector, and the frame format field. The frame format field identifies the type

of stack frame. The RTE instruction uses the value in the format field to properly restore the information stored in the stack frame and to deallocate the stack space. The general form of the exception stack frame is illustrated in Figure 5-1. Refer to **SECTION 6 EXCEPTION PROCESSING** for a complete list of exception stack frames.



**Figure 5-1. General Exception Stack Frame**

## SECTION 6

# EXCEPTION PROCESSING

Processing an exception occurs in four steps, with variations for different exception causes. During the first step, a temporary 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 switches to instruction processing.

### 6.1 EXCEPTION VECTORS

Exception vectors are memory locations from which the processor fetches the address of a routine to handle that exception. All exception vectors are one long word in length, except for the reset vector, which is two long words in length. All exception vectors lie in the supervisor data address space, except for the reset vector located in the supervisor program address space. A vector number is an 8-bit number. This vector number is multiplied by four to form the vector offset. The vector offset is added to the vector base register (VBR) during exception processing to arrive at a memory address. Vector numbers are generated internally or externally, depending on the cause of the exception.

As shown in Table 6-1, 192 vectors are reserved for user definition as interrupt vectors, and 64 vectors are defined by the processor. However, there is no protection on the first 64 vectors; therefore, external devices may use vectors reserved for internal purposes at the discretion of the systems designer. This practice, however, is strongly discouraged.

#### 6.1.1 Types of Exceptions

Exceptions can be generated by either internal or external causes. The externally generated exceptions are interrupts, bus errors, breakpoint, and reset requests. The interrupts are requests from peripheral devices for processor action, the breakpoints are requests from development equipment for processor action, and bus errors and reset are used for access control and

**Table 6-1. Exception Vector Assignments**

Vector Number(s)	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 Divide
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 for Coprocessor Protocol Violation)
14	56	038	SD	Format Error and
15	60	03C	SD	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 for Coprocessor) —
59–63	236 252	0EC 0FC	SD	(Unassigned, Reserved) —
64–255 1020	256	100 3FC	SD	User-Defined Vectors (192)

6

processor restart. The internally generated exceptions come from instructions, address errors, tracing, or breakpoint instructions. The TRAP, TRAPcc, TRAPV, BKPT, CHK, CHK2, RTE, and DIV instructions can generate exceptions as part of their normal execution. In addition, illegal instructions, instruction fetches from odd addresses, word or long-word operand accesses from odd addresses, and privilege violations cause exceptions.

## 6.1.2 Multiple Exceptions

The following paragraphs describe the processing that occurs when multiple exceptions arise simultaneously. Exceptions can be grouped according to their characteristics and priority as shown in Table 6-2.

The priority relationship between two exceptions determines which is processed first if both exceptions occur simultaneously. The term "process" in this context means the execution of the four steps previously defined:

1. Change processing states (if needed).
2. Determine exception vector.
3. Save old context.
4. Load new context and begin instruction fetches.

Process in this context does not include the execution of the routine pointed to by the exception vector. As soon as the CPU32 has completed processing for an exception, it is then ready to begin execution of the exception handler routine or begin exception processing for other pending exceptions. Also, it is possible for a higher priority exception to be processed before the completion of exception processing for lower priority exceptions. For example, if a bus error occurs during the processing for a trace exception, the bus error will be processed and handled before the trace exception processing is completed. However, most exceptions cannot occur during exception processing, and very few combinations of the exceptions shown in Table 6-2 can be pending simultaneously.

**Table 6-2. Exception 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, Divide 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 is executed.
4.1 4.2 4.3	Trace Hardware Breakpoint Interrupt	Exception processing begins when current instruction or previous exception processing is completed.

Group zero (0) is the highest priority; group four (4.3) is the lowest priority. This priority scheme is very important in determining the order in which exception handlers are executed in multiple-exception situations. As a general rule, the lower the priority of an exception, the more quickly the handler routine for that exception will be executed.

For example, consider the arrival of an interrupt during the execution of a TRAP instruction while tracing is enabled. The trap exception is processed first, followed immediately by exception processing for the trace and then the interrupt. Thus, when the processor resumes normal instruction execution, it is in the interrupt handler, which returns to the trace handler, which returns to the trap exception handler. An exception to this rule is the reset exception, which is the highest priority and the first exception handled, since all other exceptions are cleared by the reset condition.

### 6.1.3 Exception Stack Frame

Exception processing saves the most volatile portion of the current context on the top of the supervisor stack. This context is organized in a format called the exception stack frame. This information always includes the status register and program counter of the processor when the exception occurred. To support generic handlers, the processor places the vector offset in the exception stack frame. The processor also marks the frame with a frame format. The format field allows the RTE instruction to identify what information is on the stack so that it may 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, the format 0000 is always legal and indicates that just the first four words of the frame are present. See **6.4 CPU32 STACK FRAMES** for a complete list of CPU32 exception stack frames.

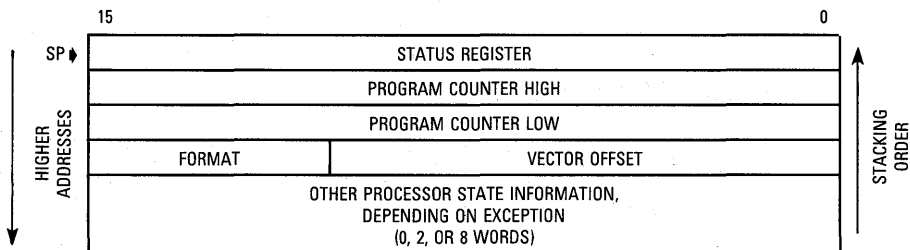


Figure 6-1. Exception Stack Frame

## 6.1.4 Exception Processing Sequence

Exception processing occurs in four identifiable steps. During the first step, an internal copy is made of the status register. After the copy is made, the processor state bits in the status register are changed as follows. The S bit is set, putting the processor into supervisor privilege state. Also, the T1 and T0 bits are cleared, allowing the exception handler to execute unhindered by tracing. For the reset and interrupt exceptions, the interrupt priority mask is also updated.

During the second step, the vector number of the exception is determined. For interrupts, the vector number is obtained by a processor read from CPU space \$F, defined as an interrupt acknowledge. For all other exceptions, internal logic provides the vector number. This vector number is then used to generate the address of the exception vector.

For exceptions other than reset, the third step is to save the current processor status. The exception stack frame is created and placed on the supervisor stack. The stacked information is dependent on the exception and the context in which it is being processed. All exception stack frames contain a copy of the status register (before the exception) and the program counter to return to following the RTE instruction. Additional information defining the current context is stacked for some instructions generated and for all bus error and address error exceptions.

The last step is the same for all exceptions. The exception vector offset is determined by multiplying the vector number by four. This offset is then added to the contents of the VBR to determine the memory address of the exception vector. The new program counter value (and supervisor stack pointer for the reset exception) is fetched from the exception vector. The processor then resumes instruction execution. The instruction at the address given in the exception vector is fetched, and normal instruction decoding and execution is started.

## 6.2 PROCESSING OF SPECIFIC EXCEPTIONS

Exceptions have a number of sources, and each exception has processing that is peculiar to it. This section details the sources of exceptions, how each arises, and how each is processed.



## 6.2.1 Reset

Assertion of RESET by external hardware or assertion of the internal  $\overline{\text{RESET}}$  signal by an internal module causes a reset exception. The reset exception has the highest priority of any exception; it provides for system initialization and recovery from catastrophic failure. When it is recognized, reset exception aborts any processing in progress, and that processing cannot be recovered. Figure 6-2 is a flowchart of the reset exception, which performs the following operations:

1. Clears both trace bits in the status register to disable tracing.
2. Places the processor in the supervisor privilege level by setting the supervisor bit in the status register.
3. Sets the processor interrupt priority mask to the highest priority level (level seven).
4. Initializes the vector base register to zero (\$00000000).
5. Generates a vector number to reference the reset exception vector (two long words) at offset zero in the supervisor program address space.
6. Loads the first long word of the reset exception vector into the interrupt stack pointer.
7. Loads the second long word of the reset exception vector into the program counter.

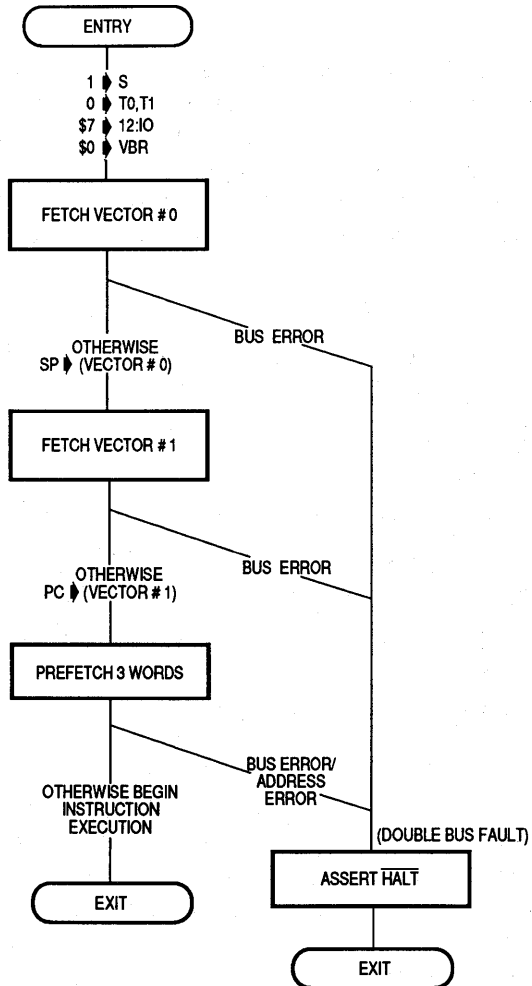
After the initial instruction prefetches, program execution begins at the address in the program counter. The reset exception does not save the value of either the program counter or the status register. If a bus error or address error occurs during the exception processing sequence for a reset, a double bus fault occurs. The processor halts, and the  $\overline{\text{HALT}}$  signal is asserted to indicate the halted condition. Execution of the reset instruction does not cause a reset exception or affect any internal registers, but it does cause the CPU32 to assert the  $\overline{\text{RESET}}$  signal, resetting all internal and external peripherals.

6

## 6.2.2 Bus Error

Bus error exceptions occur when an assertion of the  $\overline{\text{BERR}}$  signal is acknowledged. The  $\overline{\text{BERR}}$  signal is asserted by one of three sources:

1. External logic by assertion of the  $\overline{\text{BERR}}$  input pin, or
2. Direct assertion of the internal  $\overline{\text{BERR}}$  signal by an internal module, or
3. Direct assertion of the internal  $\overline{\text{BERR}}$  signal by the on-chip hardware watchdog after detecting a no-response condition.



**Figure 6-2. Reset Operation Flowchart**

The time at which the bus error is acknowledged differs for instruction and data accesses. If the aborted bus cycle is a data access, the processor immediately begins exception processing, 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. If the aborted bus cycle is an instruction prefetch, the processor will delay taking the exception until it attempts to use the prefetched information. For example, a branch instruction may flush an aborted prefetch and, since that word is not

accessed, no exception occurs. Because of instruction prefetch, exception processing of an aborted instruction fetch is delayed until the processor actually attempts to use the aborted information.

Exception processing for bus error exceptions follows the normal exception flow. The status register is copied, the supervisor state is entered, and the tracing is disabled. A vector number is generated to refer to the bus error vector. Since the processor was not necessarily between instructions when the bus error exception request was made, preserving the context of the processor is more involved than for other exceptions. Any registers altered within the faulted-instruction effective address calculation are restored to their initial values. Information identifying the faulted bus cycle is placed on the stack in a special status word (SSW). The SSW contains specific information about the aborted access: its size, whether it was a read or a write, the bus cycle type, and the function code when the bus error occurred. The fault address, bus error exception vector number, program counter, and copy of the status register are also saved. Several bus error stack format organizations are utilized to provide additional information regarding the nature of the fault. After stacking, the processor continues instruction processing at the address contained in the bus error exception vector (vector 2).

If a bus error occurs during the exception processing for a bus error, address error, or reset or while the processor is loading information from the stack during the execution of an RTE instruction, the processor halts. This cessation simplifies the detection of catastrophic system failure, since the processor removes itself from the system rather than modifying the current state of the stacks and memory. Only the assertion of RESET can restart a halted processor.

## 6

### 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 like an internally generated bus error initiating exception processing. After exception processing commences, the sequence is the same as that for bus error, except that the vector number refers to the address error vector. If an address error occurs during the exception processing for a bus error, address error, or reset, the processor is halted.

The time at which the address error is acknowledged is dependent on several factors. If the aborted bus cycle was a data space access, the processor begins exception processing when an attempt is made to use the data from the bus

cycle, or, in the case of a released write, at the next instruction boundary or attempted operand access. The prefetch mechanism delays exceptions until the processor actually attempts to use the aborted cycle's information. Therefore, an address error on a change in flow (e.g., a branch to an odd address) is delayed until the end of the instruction that affected the program counter. For this case, the fault address and return program counter will be the odd address. The current instruction program counter is that of the instruction which resulted in the exception.

## 6.2.4 Instruction Traps

Traps are exceptions caused by instructions. They arise either from processor recognition of abnormal conditions during instruction execution or from use of specific trapping instructions.

Some instructions are used specifically to generate traps. 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 an exception if the user program detects a run-time error, which may be an arithmetic overflow or a value out of bounds. The DIVS and DIVU instructions will force an exception if a division operation is attempted with a divisor of zero.

Exception processing for traps is straightforward. The status register is copied, the supervisor state is entered, and the trace state is turned off. Thus, if tracing was enabled when the instruction causing a trap began 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 is internally generated; for the TRAP instruction, part of the vector number comes from the instruction itself. The trap vector number, program counter, and copy of the status register are saved on the supervisor stack. The saved value of the program counter is the address of the instruction following the instruction which generated the trap. For all instruction traps other than TRAP, a pointer to the instruction causing the trap is also saved as the fifth and sixth words of the exception stack frame. Finally, instruction execution commences at the address contained in the exception vector.

## 6.2.5 Software Breakpoints

To use the CPU32 in a hardware emulator, the processor must provide a means of inserting breakpoints into the target code, giving a clear announcement when it has reached a breakpoint. For the MC68000 and MC68008, an illegal instruction can be inserted at the breakpoint and detected when the processor is fetched from the illegal instruction exception vector location. Since the VBR on the MC68010, MC68020, and CPU32 allows arbitrary relocation of the exception vectors, the exception vector address cannot serve as a reliable indication that the processor is taking the breakpoint. On the MC68010, MC68020, and CPU32, this function is provided by extending the functionality of a set of the illegal instructions (\$4848–\$484F) to serve as breakpoint instructions.

When a breakpoint instruction is executed, the CPU32 performs a read from CPU space \$0 at a location corresponding to the breakpoint number. If this bus cycle is terminated by BERR, the processor then proceeds to perform the illegal instruction exception processing. If the bus cycle is terminated by DSACK, the processor uses the data returned to replace the breakpoint in the internal instruction pipeline and begins execution of that instruction. See **5.2 ADDRESS SPACE TYPES** for detailed descriptions of the various CPU space operations.

## 6

## 6.2.6 Hardware Breakpoints

In addition to software breakpoints, the CPU32 also recognizes hardware breakpoint requests. Breakpoint requests arriving at the processor do not force immediate exception processing but are made pending. Pending breakpoints are detected between instruction executions and at the end of exception processing. When a hardware breakpoint is acknowledged, the CPU performs a read from CPU space \$0 at location \$1E (see **5.2 ADDRESS SPACE TYPES**). If the bus cycle is terminated normally, instruction execution continues with the next instruction as if a breakpoint request had not been received. On the other hand, if the bus cycle is terminated by BERR, the CPU begins exception processing. The status register is copied, the supervisor state is entered, and the trace state is turned off. The vector number is internally generated; vector 12 (offset \$30) is assigned to hardware breakpoints. The program counter of the currently executing instruction, program counter of the next instruction to execute, and copy of the status register are saved on the supervisor stack. Data returned during this bus cycle is always ignored.

## 6.2.7 Format Error

Just as the processor checks that the instruction words fetched are valid, the processor also performs some checks of data values for control operations. The RTE instruction checks the validity of the stack format code and, in the case of the bus cycle fault format, the validity of the data to be loaded into the various internal registers (i.e., the version number of the processor that generated the frame). This check ensures that the program is not making erroneous assumptions about internal state information in the stack frame.

If this check determines that 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

Illegal instruction is the term used to refer to any of the word bit patterns which 1) do not correspond to the bit pattern of the first word of a legal CPU32 instruction, 2) are an undefined register specification field in the first extension word of a MOVEC instruction, or 3) are an indexed addressing mode extension word with bits [5:4] = 00 or bits [3:0] ≠ 0000.

The word patterns with bits [15:12] equal to 1010 (referred to as A-line opcodes) are distinguished as unimplemented instructions, and a separate exception vector (vector 10, offset \$28) is given to this pattern to permit efficient emulation. If during instruction execution, such an illegal instruction is fetched, an illegal instruction exception occurs. This facility allows the operating system to detect program errors or to emulate unimplemented instructions in software.

The word patterns with bits [15:12] equal to 1111 (referred to as F-line opcodes) are used for instruction set extensions to the M68000 Family. They may generate an unimplemented instruction exception. This exception is caused by the first extension word of the instruction or by the addressing mode extension word. If the F-line instruction is an unimplemented instruction, then 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. After the instruction is fetched and decoding is attempted, the processor determines that execution of an illegal instruction is being attempted and starts exception processing before altering any registers. The status register is copied, the supervisor state is entered, and tracing is disabled. The vector number is generated to refer to the illegal instruction vector or, in the case of unimplemented instructions, to the corresponding emulation vector. The illegal instruction vector number, current program counter, and copy of the status register are saved on the supervisor stack, with the saved value of the program counter being the address of the illegal or unimplemented instruction. Finally, instruction execution commences at the address contained in the exception vector.

## 6.2.9 Privilege Violations

To provide system security, various instructions are privileged. An attempt to execute one of the privileged instructions while in the user privilege state 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. After the instruction is fetched and decoded and the processor determines that a privilege violation is being attempted, the processor starts exception processing before the instruction is executed. The status register is copied, the supervisor state is entered, and the trace state is turned off. The vector number is generated to reference the privilege violation vector; the privilege violation vector number, current program counter, and status register are saved on the supervisor stack. The saved value of the program counter is the address of the first word of the instruction causing the privilege violation. Finally, instruction execution commences at the address contained in the privilege violation exception vector.

### 6.2.10 Tracing

To aid in program development, the M68000 processors include a facility to allow instruction execution by instruction tracing. The CPU32 also allows instruction tracing to change program flow. In trace mode, a trace exception is generated after an instruction is executed, allowing a debugging program to monitor the execution of a program under test.

The trace facility uses the T1 and T0 bits in the supervisor portion of the status register. If both T bits are clear, tracing is disabled, and instruction execution proceeds normally (see Table 6-3).

If the T1 bit is clear and the T0 bit is set at the beginning of the execution of an instruction and if that instruction causes the program counter to be updated in a nonsequential manner, a trace exception is generated after the execution of that instruction has completed. Instructions that can be traced in this mode are all branches, jumps, subroutine calls, returns, and status register manipulations. If the branch is not taken, an exception is not generated.

If the T1 bit is set and the T0 bit is clear at the beginning of the execution of any instruction, a trace exception is generated after the execution of that

**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)



instruction has completed. If the instruction is not executed, either because an interrupt is taken or the instruction is illegal, unimplemented, or privileged, the trace exception does not occur. The trace exception also does not occur if the instruction is aborted by a reset, bus error, or address error exception. In those cases, the trace exception is deferred until after execution of the instruction has successfully completed. If the instruction is executed and an interrupt is pending on completion, the trace exception is processed before the interrupt exception. If, during the execution of the instruction, an exception is forced by that instruction, 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.

In general terms, 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 completed. If the instruction does not complete execution due to a bus error or address error exception, trace exception processing is deferred until after the execution of the suspended instruction is restarted (by the associated RTE) and completed normally. If the instruction is executed and an interrupt is pending on completion, the trace exception processing is completed before the interrupt exception processing starts. If, during the execution of the instruction, an exception is forced by that instruction, the forced exception is processed before the trace exception is processed.

6

If the processor is in trace mode when an attempt is made to execute an illegal, unimplemented, or privileged instruction, that instruction will not cause a trace since it is not executed. This is of particular importance to an instruction emulation routine that performs the instruction function, adjusts the stacked program counter to beyond the unimplemented instruction, and then returns. Before the return is executed, the status register on the stack should be checked to determine if tracing is on. If tracing is on, the trace exception processing should also be emulated for the trace exception handler to account for the emulated instruction.

The exception processing for trace starts at the end of normal processing for the traced instruction and before the start of the next instruction. An internal copy is made of the status register. The transition to supervisor state is made, and the T bits of the status register are cleared, disabling further tracing. 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 the 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. Instruction execution commences at the address contained in the trace exception vector.

Tracing affects the normal operation of two instructions. If the STOP or LPSTOP instructions begin execution with T1 set, a trace exception will be taken after the STOP (LPSTOP) 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 never enter the stopped condition. Also, an RTE from a bus error or address error will not be traced because of the possibility of continuing the instruction from the fault.

## 6.2.11 Interrupts

Seven levels of interrupt priorities are provided. Devices may be chained within interrupt priorities, allowing an unlimited number of peripheral devices to interrupt the processor. Interrupt recognition and subsequent processing are based on the internal interrupt request signals ( $\overline{\text{IRQ7}}\text{--}\overline{\text{IRQ1}}$ ) and the current processor priority set in the priority mask (I2, I1, I0) of the status register. Interrupt request level zero ( $\overline{\text{IRQ7}}\text{--}\overline{\text{IRQ1}}$  negated) indicates that no service is requested. When interrupt level one through six is requested via  $\overline{\text{IRQ6}}\text{--}\overline{\text{IRQ1}}$ , 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 processor priority. The exception is level seven, which is nonmaskable.

The CPU32 input synchronization circuitry for the  $\overline{\text{IRQ7}}\text{--}\overline{\text{IRQ1}}$  control lines samples these inputs on consecutive rising edges of the processor clock to synchronize and debounce these signals. An interrupt request held constant for two consecutive clock periods is considered a valid input; therefore, it is possible that an interrupt request is held for as short a period as two clock cycles could be recognized. Valid edges on level seven are latched until acknowledged.

Interrupt requests arriving at the processor do not force immediate exception processing but are made pending. Pending interrupts are detected between instruction executions and at the end of exception processing. If the priority of the pending interrupt is greater than the current processor priority, the exception processing sequence is started. First, a copy of the status register is saved, the privilege state is set to supervisor, tracing is suppressed, and

the processor priority level is set to the level of the interrupt being acknowledged. The processor fetches the vector number from the interrupting device, classifying the bus cycle as an interrupt acknowledge (CPU space \$F) and displaying the encoded level number of the acknowledged interrupt on the address bus.

If the interrupting device requests automatic vectoring, the processor internally generates a vector number 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 generated vector number references the spurious interrupt vector. The processor then proceeds with the usual exception processing, saving the exception vector number, program counter, and status register on the supervisor stack. The saved value of the program counter is the address of the instruction which would have been executed had the interrupt not occurred. The content of the interrupt vector, whose vector number was previously obtained, is fetched and loaded into the program counter, and normal instruction execution commences in the interrupt handler routine.

Priority level seven is a special case. Level-seven interrupts cannot be inhibited by the interrupt priority mask, thus providing a nonmaskable interrupt capability. Level-seven requests are necessarily edge triggered to eliminate continuous servicing and inevitable stack overflow. A level-seven interrupt is generated 1) each time the interrupt request level changes from some lower level to level seven (regardless of the processor priority mask level) or 2) if the request level remains at level seven and the processor priority mask is changed from level seven to a lower level.

Many M68000 peripherals provide for programmable interrupt vector numbers to be used in the interrupt request/acknowledge mechanism of the system. If this vector number is not initialized after reset and if the peripheral must acknowledge an interrupt request, the peripheral should return the vector number for uninitialized interrupt vector (vector 15).

See the system integration user's manual for detailed information on the interrupt acknowledge cycle operation.

## 6.2.12 Return from Exception

After exception stacking operations have completed for all pending exceptions, the processor resumes instruction execution at the address contained in the vector referenced by the last exception to be processed. Once the

exception handler has completed execution, the processor must return to the system context in existence prior to the exception (if possible). The mechanism to accomplish this action for any exception is the RTE instruction.

When the RTE instruction is executed, the processor examines the stack frame on top of the supervisor stack to determine if it is a valid frame and what type of context restoration should be performed. See **SECTION 7 DEVELOPMENT SUPPORT** for a description of each format type. For a normal four-word frame, the processor updates the status register and program counter with the data pulled from the stack, increments the supervisor stack pointer by eight, and resumes normal instruction execution. For the six-word frame, the status register and program counter are updated from the stack, the active supervisor stack pointer is incremented by 12, and normal instruction execution resumes.

For the bus fault frame, the format value on the stack is first checked for validity. In addition, the version number contained 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. A validity check is used to insure that the data in a multiple processor system will be properly interpreted by the RTE instruction. If the frame is invalid or inaccessible, a format error or a bus error exception is taken, respectively. Otherwise, the processor reads the entire frame into the proper internal registers, deallocates the stack (12 words), and resumes normal processing. Bus error frames for faults during exception processing require that the RTE instruction rewrite the faulted stack frame. If an error occurs during any of the bus cycles required to rewrite the frame, the processor will enter the halted state.

If a format error or bus error occurs during execution of an RTE instruction due to any of the errors previously described or an illegal format code, the processor will create a normal four-word or a bus-cycle fault stack frame below the frame that it was attempting to use. In this way, the faulty stack frame remains intact and may be examined and repaired by the format error or bus error exception handler or used by a different type processor (e.g., an MC68010, MC68020, or a future M68000 processor) in a multiprocessor system.

## 6.3 FAULT RECOVERY

There are four facets to recovery from the memory 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 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	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 (RMW)
- IN — Instruction/Other
- RW — Read/Write of Faulted Bus Cycle
- LG — Original Operand Size Was Long Word
- FUNC — Function Code of Faulted Bus Cycle
- SIZ — Remaining Size of Faulted Bus Cycle

6

The SSW defines the class of the faulted bus operation in the TP field. Two BERR exception frame types are defined to support faults on prefetch and operand accesses and 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 the bus fault. If a prefetch bus fault occurs while recovering from the MOVEM fault (e.g., when refetching the MOVEM opcode and extension word), both the MV and IN bits will be set in the stacked SSW.

- 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 the bus error exception was processed; thus, the instruction that generated the trace will not be restarted upon returning from the exception handler. This includes the MOVEM

and released write bus errors indicated by the assertion of either MV or RR in the stacked SSW.

- 0 — Trace not pending
- 1 — Trace pending

B1 indicates that a breakpoint exception was pending on channel 1 (external breakpoint source) when the bus error exception was processed. Pending breakpoint status is stacked, regardless of the type of bus error exception being processed.

- 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 being processed.

- 0 — Breakpoint not pending
- 1 — Breakpoint pending

RR will be set in the stacked SSW if the faulted bus cycle was a released write. If the write is completed (rerun) in the exception handler, the stacked RR bit should be cleared before executing the RTE. The bus cycle will be rerun if the stacked RR bit is set upon returning 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 in the stacked SSW. This bit 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 in the stacked SSW. 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 in the stacked SSW. Read bus cycles will set this bit, and write bus cycles will clear this bit in the stacked SSW. This 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

An original operand size of long word is conveyed in the LG bit in the stacked SSW. LG is cleared if the original size of the operand was byte or word; SIZ will indicate the original (and remaining) size. LG is set if the original size of the operand was long word; SIZ will indicate the remaining size at the time of the fault. LG is ignored during unstacking.

- 0 — Original operand size was byte or word
- 1 — Original operand size was long word

The operand size remaining when the fault was detected is available in the SIZ field of the stacked SSW. 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 at the external bus and is transparent to the CPU. The byte size is shown only when the original operand was a byte. This 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

### 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 the operand transfer portion of MOVEM, and faults on any other bus cycle.

**6.3.1.1 TYPE I: RELEASED WRITE FAULTS.** CPU32 instruction pipelining is the overlap of the final instruction write with the execution of the following instruction. These overlapped writes are referred to as released writes. Since the machine context is lost (for the instruction that queued the write) as soon as the following instruction starts, it would not be possible to restart the faulted instruction.

Released write faults are taken at the next instruction boundary, and the stacked program counter is that of the next unexecuted instruction. Should a subsequent instruction attempt an operand access with a released write

fault pending, the instruction will be aborted and the write fault acknowledged. This action prevents any possibility of stale data 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		

The trace and breakpoint pending bits (TR, B1, 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 LG, SIZ, and FUNC fields of SSW.

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 the execution of the MOVEP instruction (except the last write of a MOVEP Rn,(ea) or the last write of MOVEM, which is a type I fault). The TAS, MOVEP, and MOVEM instructions account for all operand writes not considered released.

All type II faults cause an immediate exception, resulting in the current instruction being aborted. 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 in the SSW for type II faults, since the instruction will be restarted upon returning from the handler. Saving the pending exception on the stack would have resulted in the trace exception being taken prior to restarting the instruction. Assuming that 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 returning from the handler. This stacking avoids problems with bus state analyzer equipment, which has been programmed to breakpoint only the first access to a specific location or which is counting accesses to the programmed location. If this response is not desired, the exception handler can clear the bits before returning. The RM, IN, RW, LG, FUNC, and SIZ fields all reflect the type of bus cycle causing the fault. If the bus cycle was an RMW, the RM bit will be set with the RW bit indicating whether the fault was on the read or write.

**6.3.1.3 TYPE III: FAULTS DURING MOVEM OPERAND TRANSFERS.** Bus faults occurring as a result of a MOVEM operand transfer are classified as type III faults. Instruction prefetch faults associated with MOVEM are type II faults.

Type III faults cause an immediate exception, resulting in the current instruction being aborted. None of the registers altered during execution of the faulted instruction are restored prior to executing the fault handler, including any register predecremented as a result of the effective address calculation or any registers overwritten during the course of instruction execution. Since postincremented registers are not updated until the end of the 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 in the stacked SSW, indicating that the MOVEM should be continued from the point of the fault upon returning from the exception handler. TR, B1, and B0 are set if the corresponding exception is pending when the BERR exception is taken. IN is set if a bus fault occurred during an attempt to restart the instruction while refetching the opcode or an extension word. RW, LG, SIZ, and FUNC all reflect the type of bus cycle which caused the fault. All write faults have the RR bit set, indicating that the write should be rerun upon returning from the exception handler.

The remainder of the stack frame contains the processor's internal state necessary to continue the MOVEM with the operand transfer following the faulted transfer. The next operand to be transferred, incremented or decremented by the 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 final instance in which a fault can occur is during exception processing. If the exception is another address or bus error, the machine halts with 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		

TR, B1, and B0 are set if the 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. From the format/vector word, the type of exception can be determined. If the faulted exception stack frame was a six-word frame, the program counter of the instruction causing the initial exception is stacked as well. This data is written on the stack in the format depicted in Figure 6-4. The stacked RTE return address is that to which the initial exception would have returned.

### 6.3.2 Correcting the Fault

Methods for correcting type I, II, III, and IV faults are discussed in the following paragraphs.

**6.3.2.1 COMPLETING RELEASED WRITES (TYPE I) VIA SOFTWARE.** There are two methods of completing a faulted released write bus cycle. The first method is to use a software handler; the second method involves having the bus cycle rerun by the RTE instruction. To complete the bus cycle in software, the handler must transfer the data in the stacked image of the data output buffer to the external location indicated by the image of the fault address in the address space defined by the function code field of the SSW.

The internal 16-bit data bus requires splitting long operands into two bus accesses. A fault on the second access of a long operand causes the LG bit in the SSW to be set with the SIZ field, indicating the operand size remaining at the time of the fault. If operand coherency is important, the complete operand should be rewritten, not just the portion remaining when the fault occurred. After completion of the bus cycle, the RR bit in the stacked SSW should be cleared. Failure to clear this bit may result in an attempt by the RTE to rerun the bus cycle. It is not necessary to adjust the program counter (or any other stack contents) before executing the RTE instruction.

**6.3.2.2 COMPLETING RELEASED WRITES (TYPE I) VIA RTE.** Type I faults need not be completed in a software exception handler since the RTE instruction can complete the faulted bus cycle. After correcting the cause of the fault, the handler should execute an RTE instruction. 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 the TR, B1, and B0 bits in the stacked SSW, are requeued during the restoration of SSW. As part of the unstacking operation, the RR bit in the SSW is checked. If set, the SSW, RW, FUNC, and SIZ fields are copied into the machine. Utilizing the restored state, the CPU32 initiates a bus cycle to rerun the released write cycle.

Long-word write operand coherency can be maintained only if the stack contents are adjusted prior to execution of the RTE. The fault address should be decremented by two if LG is set and SIZ indicates a remaining byte or word size. SIZ must also be set to long. All other fields should be unaltered. Utilizing this modified state, the bus controller initiates the bus cycle(s) to rerun the complete released write operand.

#### NOTE

Manipulating the stacked SSW value can cause unpredictable results. The RR bit is checked by RTE; however, the RW bit is not checked. Therefore, it is possible that the bus cycle rerun by the RTE instruction will not be a write or will not be to the same address space as the initial bus cycle. If the bus cycle is a read, the returned data will be ignored.

**6.3.2.3 CORRECTING TYPE II FAULTS VIA RTE.** Instructions aborted due to a type II fault are restarted upon returning from the exception handler. It is imperative that the fault handler perform corrective steps which allow the instruction to be safely executed when restarted. If the fault was due to a nonresident

page in a demand-paged virtual memory configuration, the fault address should be read from the stack, and the corresponding page should be swapped in. An RTE instruction terminates the exception handler. After unstacking the machine state, the instruction is refetched and restarted.

**6.3.2.4 CORRECTING TYPE III FAULTS VIA SOFTWARE.** MOVEM operand fault recovery is by one of three methods: completion of the instruction in software, converting the fault to type II and restarting the instruction via RTE, or continuing the instruction from the fault via RTE. The preferred method is the latter of the three. Sufficient information is contained in the stack frame to complete the instruction in software. After correcting the cause of the memory fault, the faulted bus cycle should be rerun. The following steps are required to complete the instruction through software:

1. The MOVEM opcode and extension are read from the location pointed to by the stacked PC and PC+2. The handler need not recalculate the effective address since the next operand address is saved in the stack frame; however, the effective address field in the opcode must be examined to determine what updates should be done to the address register and program counter at the completion of the instruction.
2. Before restarting the instruction, the mask must be adjusted to account for the operands already transferred. The stacked operand transfer count is subtracted from 16 to obtain the actual number of operands transferred. Using this count value, the mask should be scanned. Each time a set bit is found, it should be cleared, and the count should be decremented. When the count reaches zero, the mask is ready for continuation of the MOVEM.
3. The operand address must be adjusted. If the effective addressing mode is predecrement, subtract the operand size (i.e., four if the size is long) to the stacked value; otherwise, add the operand size. The instruction may now be continued.
4. Continue scanning the mask for set bits. As each bit is found, the selected register is read/written from/to the operand address.
5. As each operand is transferred, the mask bit is cleared and the operand address incremented (decremented if predecrement effective address). When all bits in the mask are cleared, all operands have been transferred.
6. If the addressing mode was predecrement or postincrement, the register should be updated to complete the execution of the instruction.

7. If the TR bit is set in the stacked SSW, a six-word stack frame should be created and the trace handler should be executed. Likewise, if either the B1 or B0 bit is set in the SSW, another six-word stack frame should be created, and the hardware breakpoint handler should be executed.
8. The stack is deallocated, and control is returned to the faulted program.

In some situations, it may be necessary to rerun all operand transfers instead of continuing from the faulted operand. Clearing the MV bit in the stacked SSW converts a type III fault into a type II fault. Consequently, MOVEM, as with all other type II exceptions, will be restarted upon returning from the exception handler. When the faulted operand was not the first transferred, operand transfers completed before the fault are not "undone." These memory locations are accessed a second time when the instruction is restarted. If any registers used in the effective addressing calculation were overwritten before the fault occurred, an incorrect effective address is calculated upon restarting the instruction.

**6.3.2.5 CORRECTING TYPE III FAULTS VIA RTE.** The preferred method for recovering from a MOVEM bus fault is to correct the cause of the fault and to execute an RTE instruction without altering the contents of the stack. The RTE recognizes that a MOVEM was in progress when the fault occurred, restores the appropriate machine state, refetches the instruction, and continues the instruction with the faulted transfer. This instruction is the only instruction continued upon returning from the 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. Therefore, modifying the code prior to the RTE causes unexpected results.

**6.3.2.6 CORRECTING TYPE IV FAULTS VIA SOFTWARE.** BERR exceptions potentially occur at two points during exception processing: while fetching the exception vector or while stacking. The same stack frame and SSW are used for both possibilities; the fault address distinguishes between the two. The format/vector word image in the BERR stack frame identifies the type of faulted exception and the contents of the remainder of the frame.

A fault address corresponding to the stacked format/vector word indicates that the error occurred while trying to acquire the address of the exception handler. After correcting the cause of the fault, the BERR exception handler should execute an RTE. The RTE restores the internal machine state, fetches the address of the original exception handler, creates the original exception stack frame, and resumes execution at the address of the exception handler.

If the memory fault is uncorrectable, the exception handler should rewrite the faulted exception stack frame at  $SP + \$14 + \$06$  and jump directly to the 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 of the current stack pointer may indicate that, although the first exception received a BERR while stacking, the BERR exception stacking completed successfully. While this practice is highly unlikely, it is a possibility supported by the CPU32. Once the exception handler is assured that the fault has been corrected, recovery can proceed as described previously. If the fault cannot be corrected, the supervisor stack should be moved to another area of memory, any valid stack frames copied to the new stack area, the faulted exception frame created on top of the stack, and execution resumed at the address of the exception handler.

## 6.4 CPU32 STACK FRAMES

The CPU32 generates three different stack frames. These frames consist of the normal four- and six-word stacks and the twelve-word BERR stack frame.

### 6.4.1 Normal Four-Word Stack Frame

This stack frame is created by noninstruction-related exceptions, including interrupts and format error. It is also used for TRAP #n instructions, illegal instructions, A-line and F-line emulator traps, and privilege violations. The program counter value contains the address of the next instruction to be executed or the instruction that caused the exception, depending on the exception type (see Figure 6-3).

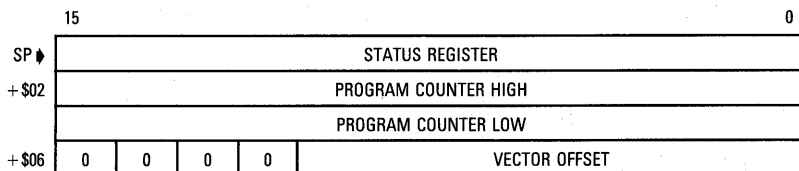


Figure 6-3. Format \$0 — Four-Word Stack Frame

## 6.4.2 Normal Six-Word Stack Frame

This stack frame is created by instruction-related traps, which include CHK, CHK2, TRAPcc, TRAPV, and zero divide. This frame is also used for trace exceptions. For these cases, the faulted instruction program counter value is the address of the instruction causing the exception. The current program counter value is the address of the next instruction to be executed and the address to which the RTE instruction returns.

Hardware breakpoints also utilize this format. The stacked faulted instruction program counter is the address of the instruction executing when the breakpoint was sensed. In most cases, this is the address of the instruction that caused the breakpoint; however, since the final operand write of an instruction is allowed to overlap into the next instruction(s), the faulted instruction program counter is not always that of the instruction causing the breakpoint. The current program counter value is the address of the next instruction to be executed and the address to which the RTE instruction returns (see Figure 6-4).

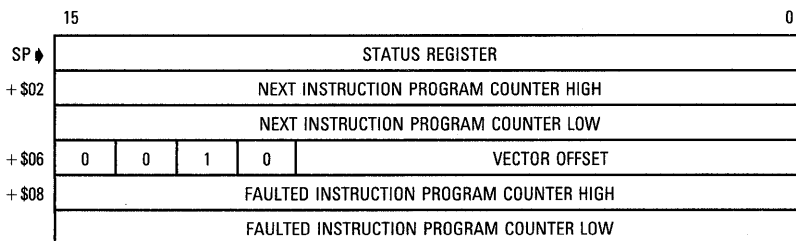
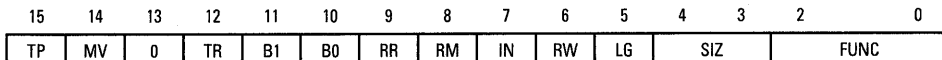


Figure 6-4. Format \$2 — Six-Word Stack Frame

## 6.4.3 BERR Stack Frame

This stack frame is created whenever a bus cycle fault is detected. The CPU32 BERR stack frame is quite different from the equivalent stack frame on any other M68000 Family member. 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 internal register. The bus operation in progress at the time of the fault is conveyed in the SSW.



The CPU32 BERR stack frame is 12 words in length. There are three variations of this frame, each distinguished by a different value in the TP:MV field of the SSW. Faults occurring during normal instruction execution (both pre-fetches and non-MOVEM operand accesses) utilize the SSW with TP:MV = 00. Figure 6-5 depicts this stack frame. An internal transfer count register appears at location SP + \$14 in all three bus error stack frames. The register contains an 8-bit microcode revision number, and, for type III faults, an 8-bit transfer count. The format of this register is depicted in Figure 6-6.

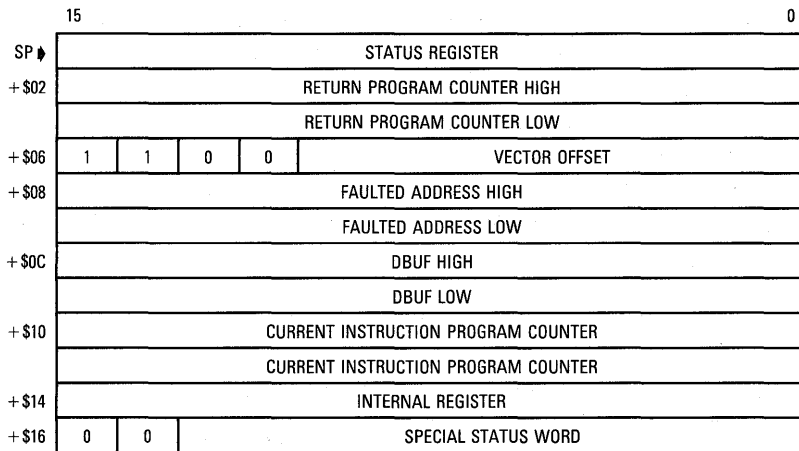


Figure 6-5. Format \$C — BERR Stack for Prefetches and Operands

6

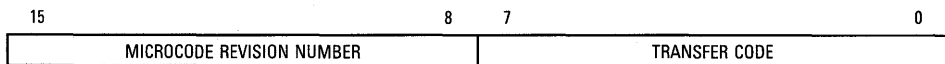


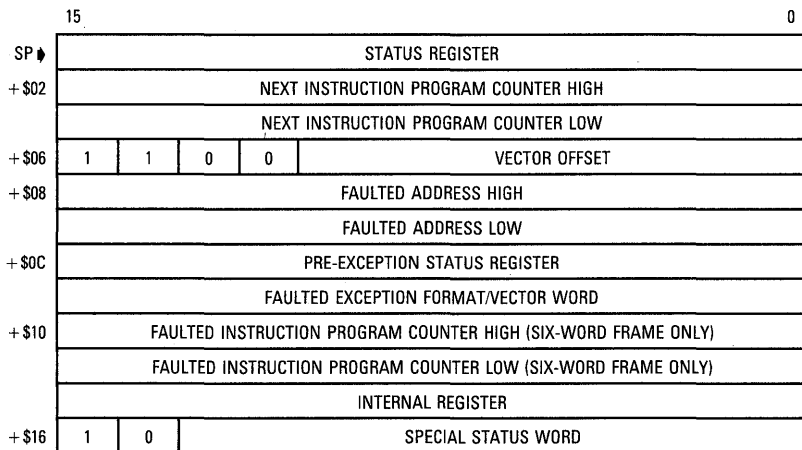
Figure 6-6. Internal Transfer Count Register



The microcode revision number is checked by the CPU when restoring a BERR stack frame via the RTE instruction. This precaution ensures that, should multiple processors be used in a system, the processor restarting from the stacked information is at the same revision level as the processor which created the stack frame. 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 the RTE instruction. An internal transfer count register appears at location SP+\$14 in all three BERR stack frames.

If a bus error occurs during exception processing, the SSW TP:MV field is set to TP:MV=10. The frame detailed in Figure 6-7 is written below the faulting frame. Stacking begins at the address pointed to by SP-6 (the SP value is the value before initial stacking on the faulted frame).

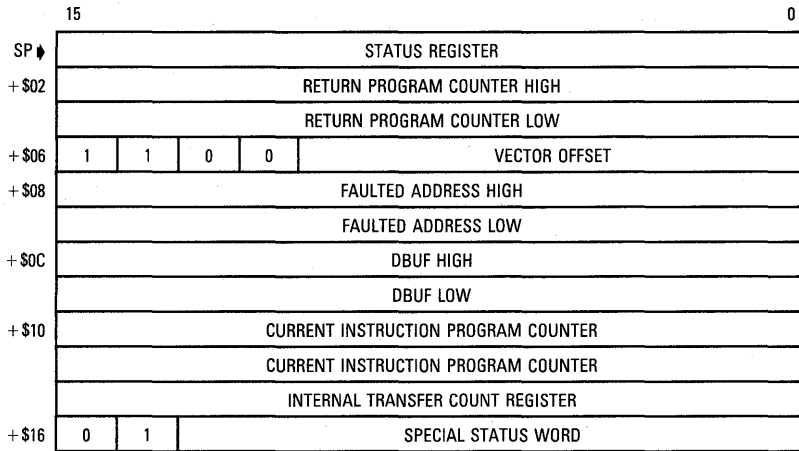
The fault address of a dynamically sized bus cycle is the upper byte. There is no indication which of the two bytes caused the BERR.



**Figure 6-7. Format \$C — BERR Stack During Four- or Six-Word Stack**

6

The third stack format is for faults that occur during the operand portion of the MOVEM instruction. This format is identified by TP:MV=01. Figure 6-8 details this stack frame.



**Figure 6-8. Format \$C — BERR Stack on MOVEM Operand**



## SECTION 7

# DEVELOPMENT SUPPORT

All M68000 Family members include the following to facilitate applications development:

**Trace on Instruction Execution** — M68000 processors include an instruction-by-instruction tracing facility as an aid to program development; however, the MC68020, MC68030, and CPU32 also allow tracing only those instructions causing a change in program flow. In the trace mode, a trace exception is generated after each instruction is executed, allowing a debugger program to monitor the execution of a program under test. See **6.2.10 Tracing** for more information.

**Breakpoint Instruction** — An emulator may insert software breakpoints into the target code to indicate when a breakpoint has occurred. 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.

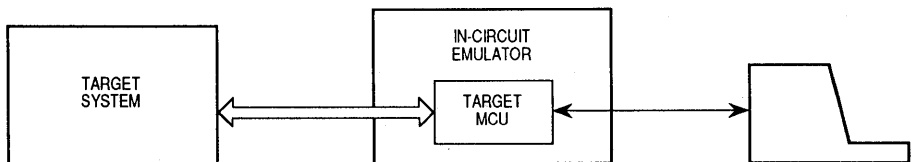
**Unimplemented Instruction Emulation** — When an attempt is made to execute an illegal instruction, an illegal instruction exception occurs. Unimplemented instructions (F-line, A-line, . . .) utilize separate exception vectors to permit efficient emulation of unimplemented instructions in software. See **6.2.8 Illegal or Unimplemented Instructions** for more information.

### 7.1 CPU32 INTEGRATED DEVELOPMENT SUPPORT

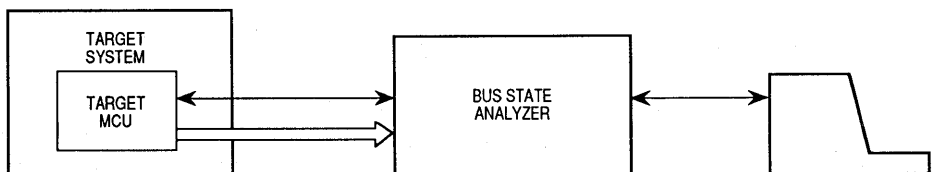
The CPU32 not only incorporates all the previous features but also provides additional features that aid development tools in advancing support for integrated system development. These additions include background debug mode, deterministic opcode tracking, hardware breakpoints, and internal visibility in the single-chip environment.

## 7.1.1 Background Debug Mode (BDM) Overview

Microcomputer systems generally provide a debugger, implemented in software, for system analysis at the lowest level. The BDM on the CPU32 is unique in that the debugger is implemented in CPU microcode. Registers can be viewed and/or altered, memory can be read or written to, and test features can be invoked. Incorporating these capabilities on-chip simplifies the environment in which an in-circuit emulator operates. The traditional in-circuit emulator configuration (see Figure 7-1) removes the MCU from the target, replacing it with hardware resident in the emulator. An expensive cable provides the communication path between the target system and the emulator. By contrast, with an integrated debugger, the traditional emulator configuration can be replaced by a bus state analyzer (BSA) (see Figure 7-2). The advantage of this configuration is twofold: 1) the processor remains in the target hardware, serving as its own emulation processor and 2) this integration reduces cost by eliminating the cable. The BSA provides a means for monitoring target processor operation; the on-chip debugger provides the mechanism for altering the operating environment. Many of the problems experienced with the classic emulator configuration are minimized: i.e., limitations on high-frequency operation, AC and DC parametric mismatches, and restrictions on cable length.



**Figure 7-1. Traditional In-Circuit Emulator Diagram**



**Figure 7-2. Bus State Analyzer Configuration**

## 7.1.2 Deterministic Opcode Tracking Overview

CPU32 function code outputs are augmented by two supplementary signals to monitor the instruction pipeline. The instruction pipe (IPIPE) output indicates the start of each new instruction and each mid-instruction pipeline advance. The instruction fetch (IFETCH) output identifies those bus cycles in which the operand data is loaded into the instruction pipeline. Pipeline flushes are also signaled with IFETCH. Monitoring these two signals allows a BSA to synchronize to the instruction stream and monitor the activity. Refer to **7.3 DETERMINISTIC OPCODE TRACKING** for a complete description.

## 7.1.3 On-Chip Hardware Breakpoint Overview

An external breakpoint input and on-chip hardware breakpoint allow a breakpoint trap on any memory access. Off-chip address comparators preclude breakpoints on internal accesses unless show cycles are enabled. Breakpoints on instruction prefetches, which are ultimately flushed from the instruction pipeline, are not acknowledged; operand breakpoints are always acknowledged. Acknowledged breakpoints optionally initiate exception processing or BDM. See **6.2.6 Hardware Breakpoints** for more information.

## 7.2 BACKGROUND DEBUG MODE (BDM)

BDM is an alternate CPU32 operating mode in which normal instruction execution is suspended while special microcode performs the functions of a debugger. BDM is initiated by one of several sources: externally generated breakpoints, internal peripheral breakpoints, the background (BGND) instruction, or catastrophic exception conditions. While in BDM, the CPU32 ceases fetching instructions via the parallel bus and, instead, accepts commands via a dedicated serial interface. A high-speed, SPI-type serial link provides BDM communication between the CPU and the development system. Figure 7-3 illustrates a block diagram of the BDM.

7

### 7.2.1 Enabling BDM

Accidentally entering BDM in a nondevelopment environment could inadvertently 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 (BKPT) signal. When BKPT is asserted (low) at the rising edge on RESET, BDM operation is enabled until the next system reset. A high BKPT

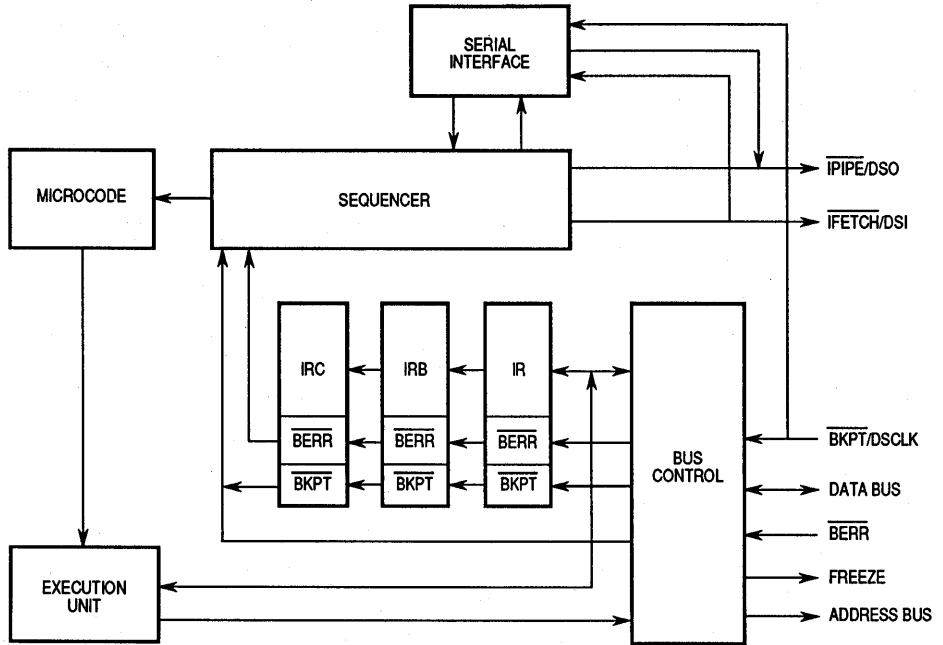


Figure 7-3. BDM Block Diagram

signal at the trailing edge of  $\overline{\text{RESET}}$  disables BDM and all sources of entry revert to their normal operation.  $\overline{\text{BKPT}}$  is related on each rising transition of  $\overline{\text{RESET}}$ .

$\overline{\text{BKPT}}$  is synchronized internally; therefore, the signal must be held low for at least two clock cycles prior to the negation of  $\overline{\text{RESET}}$ . Special care must be taken in the design of the BDM enable logic. If the hold time on  $\overline{\text{BKPT}}$  (after the trailing edge of  $\overline{\text{RESET}}$ ) extends into the first bus, the possibility exists that the bus cycle could be inadvertently tagged with a breakpoint. Refer to the system integration module user's manual for actual timing information.

## 7.2.2 BDM Sources

Once BDM has been enabled, any of several sources are capable of causing the transition from normal operation into BDM. These sources include 1) external breakpoint hardware, 2) the BGND instruction, 3) double bus faults,

and 4) internal peripheral breakpoints. If BDM is **not** enabled when the exception condition occurs, the exception is processed normally. Table 7-1 summarizes the processing of each source for both the enabled and disabled cases. As depicted in the table, the BKPT instruction never causes a transition into the BDM of operation.

**Table 7-1. BDM Source Summary**

Source	BDM Enabled	BDM Disabled
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 BKPT SIGNAL.** Once enabled, BDM is initiated whenever assertion of  $\overline{\text{BKPT}}$  is acknowledged. If BDM is disabled, a breakpoint exception (vector \$0C) is acknowledged. Timing on the  $\overline{\text{BKPT}}$  input is the same as that for read cycle data with respect to the trailing edge of data strobe. A breakpoint acknowledge bus cycle is **not** run when entering BDM.

**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 FAULTS.** Two bus faults in succession, or double bus faults, normally indicate that a catastrophic error has occurred within the system, resulting in the suspension of instruction execution. When this error condition occurs during initial system debug (e.g., a fault in the reset logic), further debugging is impossible until the situation is corrected. Through BDM, the fault can be bypassed temporarily, the cause of the problem can be determined, and the effects of the problem can be corrected. Should BDM be disabled, a double bus fault causes the processor to terminate instruction execution until reset.

**7.2.2.4 PERIPHERAL BREAKPOINTS.** Peripherals capable of requesting breakpoints do so by asserting the  $\overline{\text{BKPT}}$  signal. With respect to the CPU32, the



operation of peripheral breakpoints is identical to that of external breakpoints. Consult the appropriate peripheral user's manual for additional details on the generation of peripheral breakpoints.

### 7.2.3 Entering BDM

Upon detecting a breakpoint or double bus fault or upon decoding a BGND instruction, the processor suspends instruction execution and asserts the FREEZE output. This action is the first indication that the processor has entered BDM. Once FREEZE has been asserted, the CPU enables the serial communication hardware and awaits the first command.

As part of the process of entering BDM, the CPU writes a unique value into temporary register A (ATEMP), indicating the source that caused the transition. By issuing a read system register command as the initial command, the user can poll the register and determine the source (see Table 7-2).

**Table 7-2. Polling the BDM Entry Source**

Source	ATEMP [31:24]	ATEMP [23:0]
Double Bus Fault	SSW	FFFF
BGND Instruction	\$0000	\$0001
Hardware Breakpoint	\$0000	\$0000

ATEMP is used in most debugger commands for temporary storage; therefore, it is imperative that the read system register (RSREG) command be the first command issued after the transition into BDM.

A double bus fault during the initial stack pointer/program counter (SP/PC) fetch sequence is further distinguished by a value of \$00000001 in the current instruction PC. At no other time will the processor write an odd value into this register.

### 7.2.4 Command Execution

As each command is accumulated in the serial shifter, the microcode routine corresponding to that command is executed. If the command can complete without additional serial traffic, it does. However, if addresses or operands are required, the microcode reads each word as it is assembled by the serial

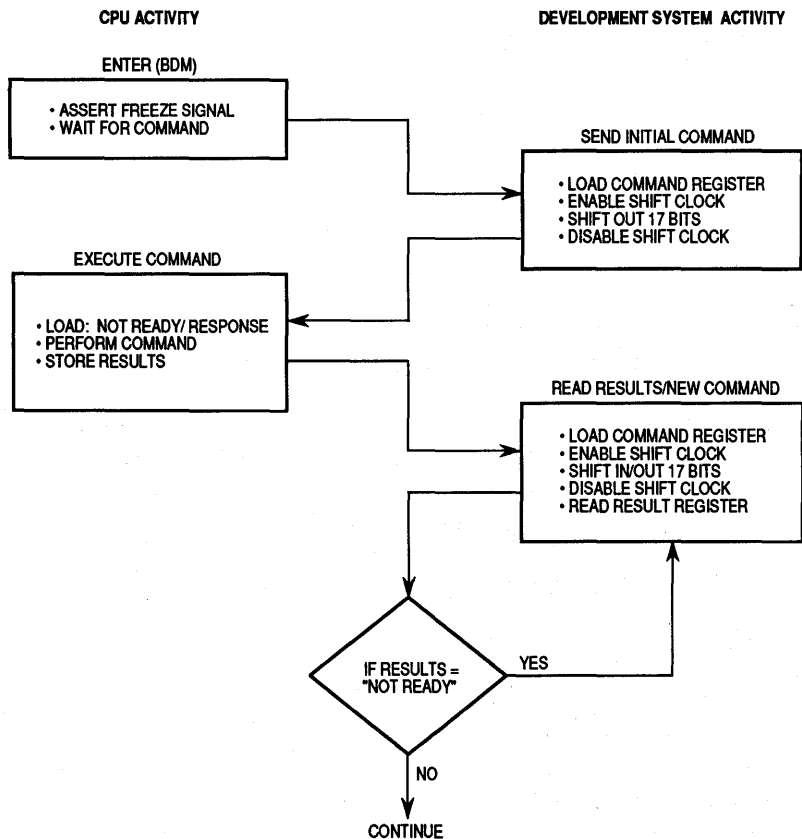


Figure 7-4. BDM Command Execution Flowchart

interface. The CPU then performs the desired operation, including any necessary memory or register accesses. 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 instruction until the CPU returns to the normal operating mode.

### 7.2.5 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 current PC. The current

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 IPIPE/IFETCH functionality.

## 7.2.6 Serial Interface

Communication with CPU32 during BDM sessions is via a dedicated serial interface, which shares pins with other development features. The BKPT signal becomes the serial clock (DSCLK); serial input data (DSI) is received on IFETCH, and serial output data (DSO) is transmitted on IPIPE.

The serial interface is 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. By deriving DSCLK from the CPU32 system clock, the design of the development system serial logic is unhindered by the operating frequency of the target MCU. Operable frequency range of the serial clock is from DC to one-half the MCU system clock frequency.

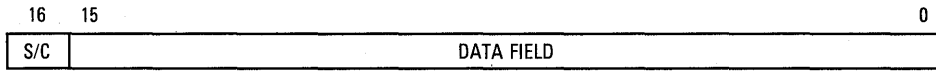
The serial interface operates in a full-duplex mode — that is, data is both transmitted and received simultaneously by the master and slave devices. In general, data transitions occur on the falling edge of the DSCLK and are stable by the following rising edge of DSCLK. Data is transmitted most significant bit 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. For CPU-generated messages, bit 17 indicates the message status as shown in Table 7-3.

Command and data transfers initiated by the development system should clear bit 17. The current implementation ignores this bit; however, Motorola reserves the right to use this bit for future enhancements.

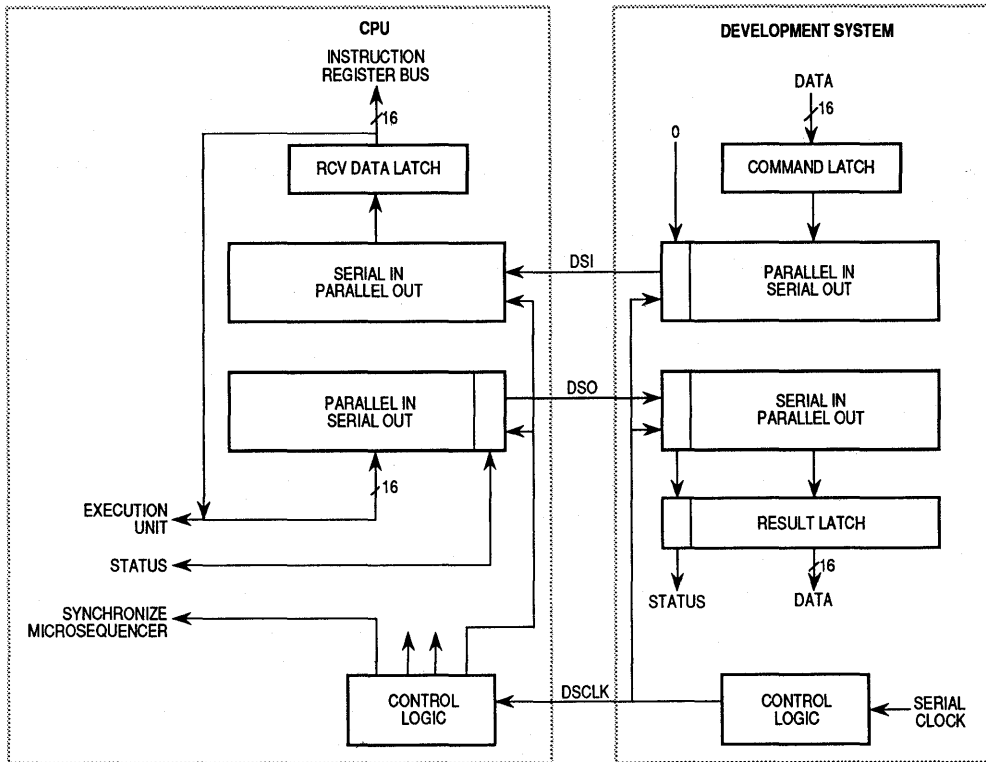
**7.2.6.1 CPU SERIAL LOGIC.** The CPU serial logic block diagram, pictured in the left-hand portion of Figure 7-5, consists of the transmit and receive shift registers and control logic containing synchronization logic, serial clock generation circuitry, and a received bit counter.

**Table 7-3. CPU-Generated Message Encoding**



↑  
STATUS/CONTROL

Bit 17	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



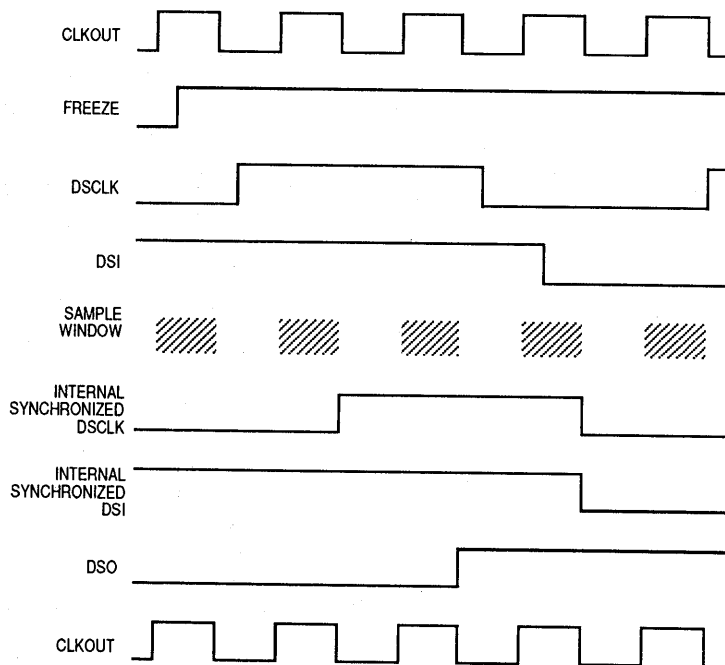
7

**Figure 7-5. Debug Serial I/O Block Diagram**

Both DSCLK and DSI are synchronized to the 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 the internal logic. The effect of this synchronization technique on the user is that the minimum hold time on DSI with respect to DSCLK is one full period of CLKOUT.

The serial state machine begins a sequence of events based on the rising edge of the synchronized DSCLK (see Figure 7-6). The 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.



**Figure 7-6. Serial Interface Timing Diagram**

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 may take advantage of the state change on DSO to signal hardware that the next serial transfer may begin. A timeout of sufficient length should also be incorporated into the design to trap error conditions that do not change the state of DSO. Hardware interlocks in the CPU prevent result data from corrupting serial transfers in progress.

**7.2.6.2 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 inadvertently interact if the dual operating modes are not properly considered when designing the clock generator.

Breakpoint requests are made by asserting  $\overline{\text{BKPT}}$  to the low state using one of two methods. The predominant method (one described thus far) is to assert  $\overline{\text{BKPT}}$  during the single bus cycle for which the exception is desired. A second method is to assert  $\overline{\text{BKPT}}$  and continue asserting 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 BKPT assertion method requires a slightly different approach in the design of the serial logic to avoid spurious serial clocks.

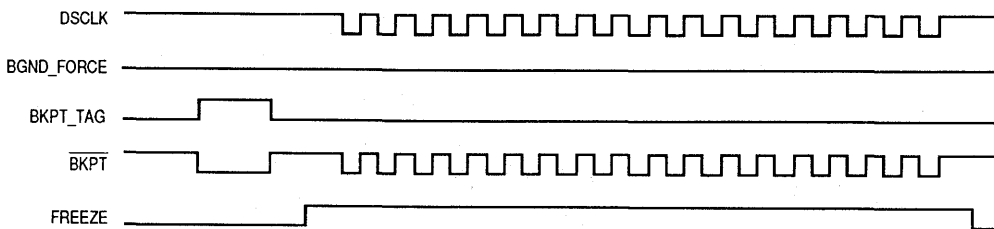


Figure 7-7. BKPT Timing for Single Bus Cycle

Figure 7-7 represents the timing required for asserting  $\overline{\text{BKPT}}$  during a single bus cycle. Figure 7-8 depicts the timing of the 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; whereas, the falling edge is effectively ignored.

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 BKPT low until the first DSCLK is applied.

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 returns high until the start of the next transmission. Clock frequency is implementation dependent and may range from DC to the maximum specified frequency.

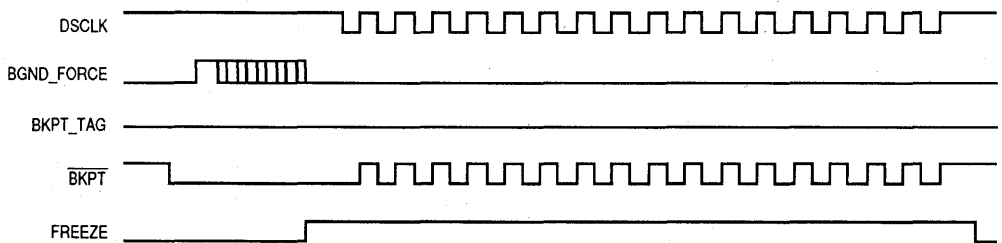


Figure 7-8. BKPT Timing for Forcing BDM

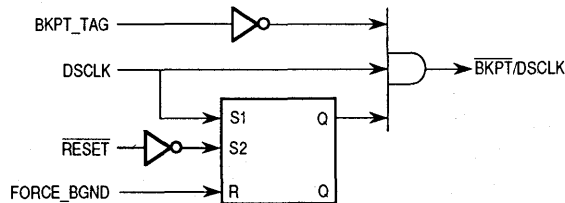


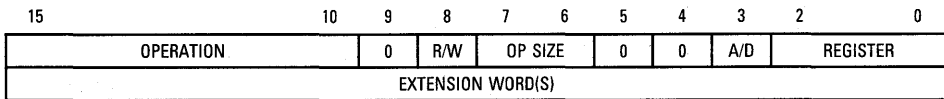
Figure 7-9. BKPT/DSCLK Logic Diagram

Although performance considerations might dictate a hardware implementation, software solutions are not precluded provided serial bus timing is maintained.

## 7.2.7 Command Set

Following is a description of the command set available in BDM.

**7.2.7.1 COMMAND FORMAT.** The following standard bit format is utilized by all BDM commands.



### Operation Field:

Commands are distinguished by the operation field. This 6-bit field provides for a maximum of 64 unique commands.

### R/W Field:

The direction of the operand transfer is specified in this field. When this bit is a one, the operation is from the CPU to the development system. When this bit is a zero, data is written into the CPU or 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:

Used by those commands that operate on address and data registers, the A/D field determines whether the register field specifies a data or address register. A one indicates an address register; a zero selects 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):

Some commands require immediate data or addresses in the form of extension words. Addresses require two extension words each since addressing capability is limited to absolute long. Immediate data can be either one or two words. Byte and word data each require a single extension word; long-word data requires two words. Operands and addresses are transferred most significant word first. At this time, no command requires an extension word to fully specify the operation to be performed (i.e., single-word commands only).

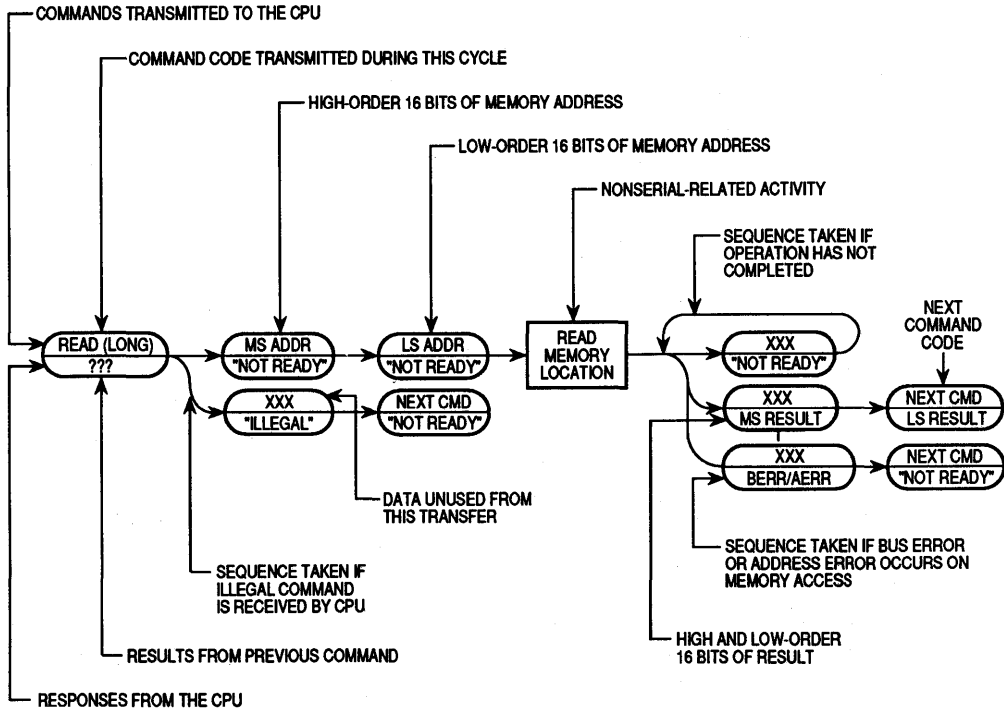
**7.2.7.2 COMMAND SEQUENCE DIAGRAMS.** A command sequence diagram illustrates the serial bus traffic for each command. Each bubble in the diagram represents a single 17-bit transfer across the bus. The top half in each diagram corresponds to the data transmitted by the development system to the CPU; likewise, 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.

The command sequence diagram in Figure 7-10 demonstrates the use of these diagrams. The cycle in which the command is issued contains the command mnemonic issued by the development system (in this example, read memory location). During the same cycle, the CPU is responding with either the lowest order results of the previous command or with a command complete status if no other results were required.

During the second cycle of the diagram, the development system supplies the high-order 16 bits of the memory address. The CPU returns the "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 except in those cases when a memory bus cycle is in progress. In all other cases, the CPU can accept a new serial transfer with eight system clock periods.



**Figure 7-10. Command-Sequence-Diagram Example**

In the third cycle, the development system supplies the low-order 16 bits of the memory address. The CPU always returns the "not ready" response in this cycle. At the completion of the third cycle, the CPU initiates the memory read operation. Any serial transfers that begin while the memory access is in progress will return the "not ready" response.

The results are returned in the two serial transfer cycles following the completion of the memory access. The data transmitted to the CPU during the final transfer is the opcode for the following command. Should the memory access generate either a bus or address error, an error status is returned in place of the result data.

**7.2.7.3 COMMAND SET SUMMARY.** The BDM command set is summarized in Table 7-4. Detailed descriptions of each command can be found in subsequent paragraphs.

**Table 7-4. BDM Command Summary**

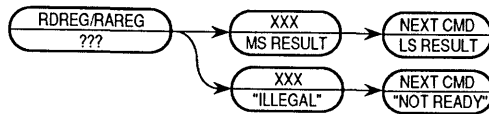
Command	Mnemonic	Description
Read A/D Register	RAREG/RDREG	Read the selected address or data register and return the results via the serial interface.
Write A/D Register	WAREG/WDREG	The data operand is written to the specified address or data register.
Read System Register	RSREG	The specified system 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 current 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 <b>not</b> 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.

**7.2.7.4 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.

**NOTE**

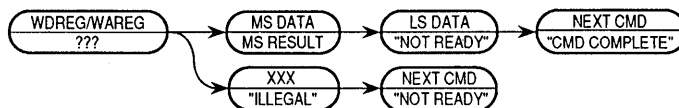
Accesses to register A7 follow the supervisor (S) bit at the time BDM was entered. If S=0, A7 corresponds to the user SP; if S=1, A7 corresponds to the supervisor SP. BDM writes to SR, which affect the S bit, have no effect on the selection of A7. Use the RSREG/WSREG commands to directly access a specific SP.

**7.2.7.5 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:

The long-word data is written into the specified address or data register. The data is supplied most significant word first.

Result Data:

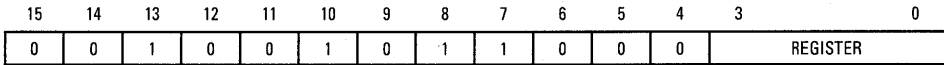
Command complete status (\$0FFFF) is returned when the register write has completed.

#### NOTE

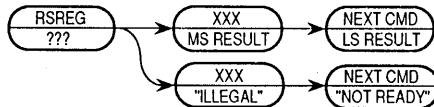
Accesses to register A7 follow the S bit at the time BDM was entered. If S=0, A7 corresponds to the user SP; if S=1, A7 corresponds to the supervisor SP. BDM writes to SR, which affect the S bit, have no effect on the selection of A7. Use the RSREG/WSREG commands to directly access a specific SP.

**7.2.7.6 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:



Command Sequence:



Operand Data:  
None

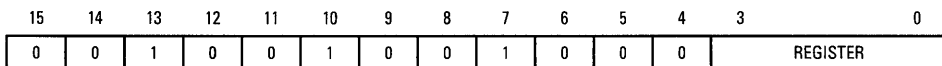
Result Data:  
Always returns 32 bits of data, regardless of the size of the register being read. If the register is less than 32 bits, the result is returned zero extended.

Register Field:  
The system control register is specified by the register field according to the following table:

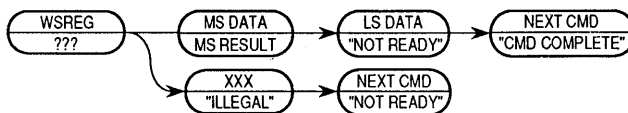
System Register	Select Code
Return Program Counter (RPC)	0000
Current Instruction Program Counter (PCC)	0001
Status Register (SR)	1011
User Stack Pointer (USP)	1100
Supervisor Stack Pointer (SSP)	1101
Source Function Code Register (SFC)	1110
Destination Function Code Register (DFC)	1111
Temporary Register A (ATEMP)	1000
Fault Address Register (FAR)	1001
Vector Base Register (VBR)	1010

**7.2.7.7 WRITE SYSTEM REGISTER (WSREG).** The 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:



Command Sequence:



Operand Data:

The data to be written into the register is always supplied as a 32-bit long word. If the written register is less than 16 bits, the least significant word is used.

Result Data:

"Command complete" status is returned when the register write is completed.

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 this register is ignored.

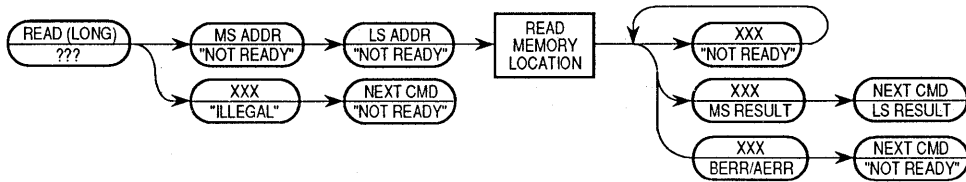
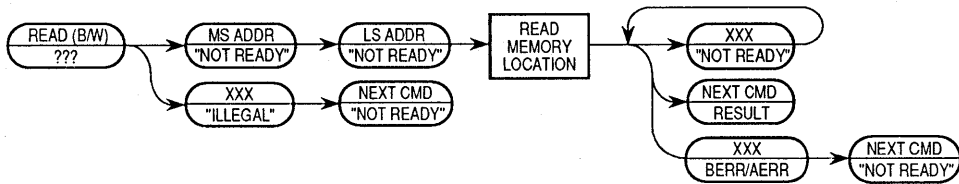
System Register	Select Code
Return Program Counter (RPC)	0000
Current Instruction Program Counter (PCC)	0001
Status Register (SR)	1011
User Stack Pointer (USP)	1100
Supervisor Stack Pointer (SSP)	1101
Source Function Code Register (SFC)	1110
Destination Function Code Register (DFC)	1111
Temporary Register A (ATEMP)	1000
Fault Address Register (FAR)	1001
Vector Base Register (VBR)	1010

**7.2.7.8 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	2	1	0
0	0	0	1	1	0	0	1	OP. SIZE	0	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. Word results return 16 bits of significant data; long-word results return 32 bits.

A successful read operation returns data bit 17 cleared; whereas, if a bus or address error is encountered, the returned data is \$10001.

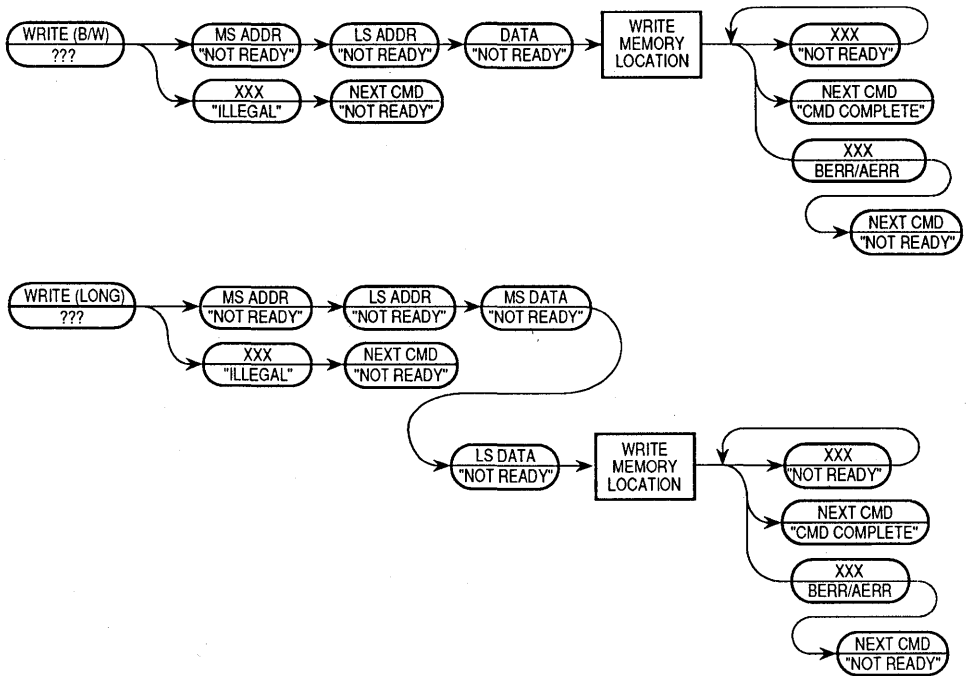


**7.2.7.9 WRITE MEMORY LOCATION (WRITE).** Write the operand data to the memory location specified by the long-word address. The destination function code (DFC) register determines the address space accessed. Only absolute addressing is supported. Valid data sizes include byte, word, and long word.

Command Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	0	OP. SIZE	0	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 specifying the location the operand data is to be written to. 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 17 in the status message and by a data pattern of \$0001.

**7.2.7.10 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**

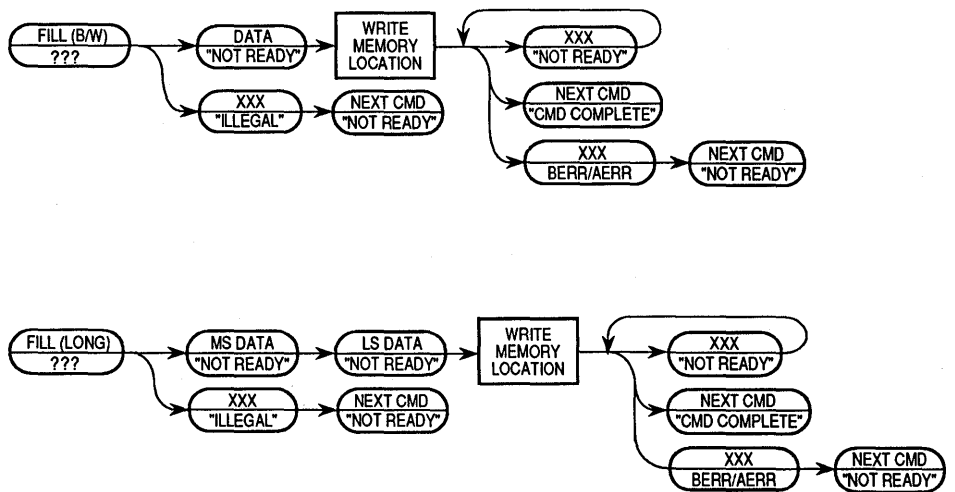
The DUMP command does not check to see that a valid address is present in the temporary register. Therefore, DUMP is a valid command only when preceded by another DUMP or by a READ command; otherwise, the results are undefined.

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	2	1	0
0	0	0	1	1	1	0	1	OP. SIZE	0	0	0	0	0	0	0

Command Sequence:



Operand Data:

None

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. 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.

**7.2.7.11 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**

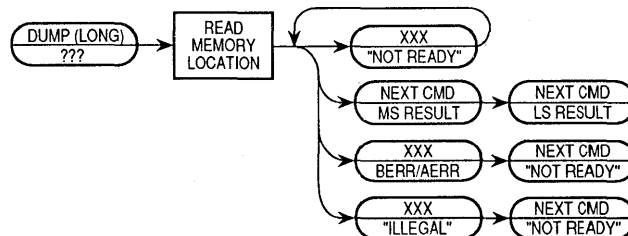
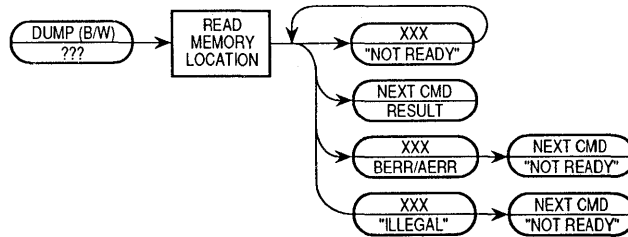
The FILL command does not check to see that a valid address is present in the temporary register. Therefore, FILL is a valid command only when preceded by another FILL or by a WRITE command; otherwise, the results are undefined.

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	2	1	0
0	0	0	1	1	1	0	0	OP. SIZE	0	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: \$0FFF for a successful operation and \$10001 for a bus or address error during write.

**7.2.7.12 RESUME EXECUTION (GO).** The pipeline is flushed and refilled before normal instruction execution is resumed. Prefetching begins at the current 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.

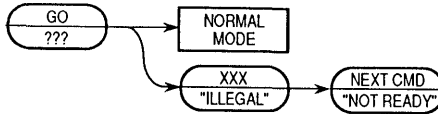
**NOTE**

A bus error or address error on the first instruction prefetch from the new PC allows BDM to exit and to be trapped as a normal mode exception. The stacked value of the current PC may or may not be valid in this case, depending on the state of the machine prior to entering BDM. In the case of an 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

**7.2.7.13 CALL USER CODE (CALL).** This instruction provides a convenient way to patch user code. The current PC is stacked at the location pointed to by the current SP (SP selected by S bit latched when BDM entered). The stacked PC serves as a return address to be restored by the return from subroutine (RTS), which terminates the patch routine. The 32-bit operand data is then loaded into the PC. The pipeline is flushed and refilled from the location pointed to by the new PC. BDM is exited, and instruction execution is initiated.

As an example, consider the following code segment that is supposed to output a character to an asynchronous communications interface adaptor. Note the missing check of 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 insert the missing code by observing the following sequence:

1. Breakpoint user program at CHKSTAT;
2. Enter BDM;
3. Execute CALL command to MISSING;  
Exit BDM;
4. Execute MISSING code; and
5. Return to user program.

#### NOTE

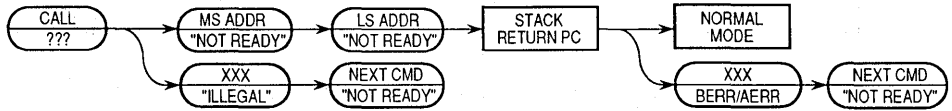
Bus errors or address errors that occur during stacking of the return address cause the CPU to return an error status via the serial interface and to remain in BDM. A bus error or address error on the first instruction prefetch from the new PC allows BDM to exit and to be trapped as a normal mode exception. The stacked value of the current PC may or may not be valid in this case, depending on the state of the machine prior to entering BDM. In the case of an address error, the return 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:

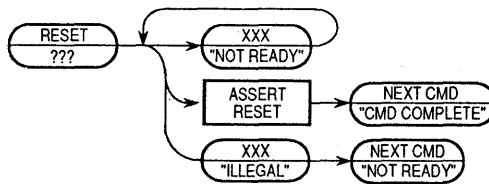
None

**7.2.7.14 RESET PERIPHERALS (RST).** RST asserts  $\overline{\text{RESET}}$  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	1	0	0	0	0	0	0	0	0	0	0	0

Command Sequence:



Operand Data:

None

Result Data:

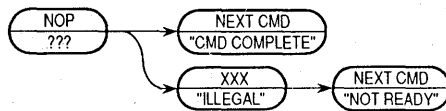
The "command complete" response (\$0FFFF) is loaded into the serial shifter after the negation of  $\overline{\text{RESET}}$ .

**7.2.7.15 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 loaded into the serial shifter.

**7.2.7.16 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.

## 7.3 DETERMINISTIC OPCODE TRACKING

The CPU32 utilizes deterministic opcode tracking to trace program execution. Two new signals, IPIPE and IFETCH, provide all the information required to analyze the operation of the instruction pipeline.

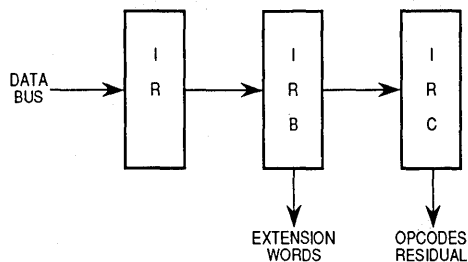
### 7.3.1 Instruction Fetch (IFETCH)

IFETCH indicates which bus cycles are accessing data to fill the instruction pipeline. IFETCH is pulse-width modulated to multiplex two indications on a single pin. Asserted for a single clock cycle, IFETCH indicates that the data from the current bus cycle is routed to the instruction pipeline. IFETCH held low for two clock cycles indicates that the instruction pipeline has been flushed. The operand of the bus cycle is used to begin filling the empty pipeline. Both user and supervisor mode fetches are signaled by IFETCH.

Proper tracking of bus cycles via the IFETCH signal on a fast bus requires a simple state machine. On a two-clock bus, IFETCH may signal a pipeline flush with associated prefetch and a consecutive prefetch. That is, 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 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 (IPIPE)

IPIPE signals the advances of the internal instruction pipeline (see Figure 7-11). The pipeline can be modeled as a three-stage FIFO in which data can be used out of both the second and third stages. The instruction register B (IRB) stage, which provides for initial decoding of the opcode and decoding of any extension words, is a source for immediate data. On the other hand, the IRC stage supplies residual decoding of the opcode during instruction execution. Assertion of IPIPE for a single clock cycle indicates the use of data out of the second stage (IRB). Regardless of the presence of valid data in the initial stage (IR), the contents of IRB are invalidated. If the IR stage contains valid data, the data is copied into IRB (IR  $\rightarrow$  IRB), and the IRB stage is revalidated.

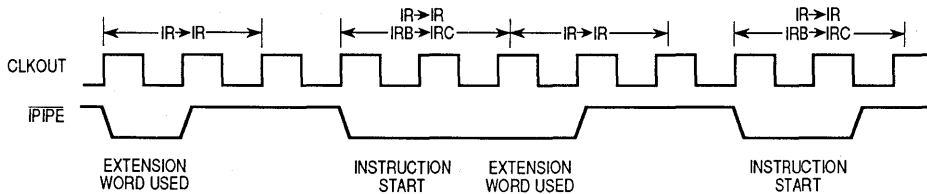


**Figure 7-11. Functional Model of Instruction Pipeline**

Assertion of  $\overline{\text{IPIPE}}$  for two clock cycles indicates the start of a new instruction and subsequent replacement of data in the final stage (IRC). This action causes a full advance of the pipeline:  $\text{IRB} \rightarrow \text{IRC}$  and  $\text{IR} \rightarrow \text{IRB}$ . IR is refilled during the next instruction fetch bus cycle. Data loaded into IR propagates through empty pipeline stages automatically, which implies that an accurate model of pipeline operation should include valid bits for the IR and IRB stages. Advancing the pipeline, either explicitly via  $\overline{\text{IPIPE}}$ , or implicitly by negated valid bits, should set the valid bit of the stage being loaded and negate the valid bit of the register supplying the data.

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.  $\overline{\text{IPIPE}}$  should be sampled on the falling edge of the clock. The assertion of  $\overline{\text{IPIPE}}$  for a single cycle after deassertion for one or more cycles indicates a use of the data in IRB (advance of IR into IRB). Assertion for two clock cycles indicates that a new instruction has started and both the  $\text{IR} \rightarrow \text{IRB}$  and  $\text{IRB} \rightarrow \text{IRC}$  transfers have occurred. Loading IRC always indicates that a new instruction is beginning execution. The opcode is the word loaded into IRC by the transfer.

In some cases, instructions using immediate addressing initiate the start of an instruction and a second pipeline advance. That is, the  $\overline{\text{IPIPE}}$  signal is not to 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.



**Figure 7-12. Instruction Pipeline Timing Diagram**

### 7.3.3 Opcode Tracking during Loop Mode

IPIPE and IFETCH continue to work normally during loop mode. IFETCH indicates all instruction fetches up through the point that data begins recirculating within the instruction pipeline. IPIPE continues to signal the start of instructions and the use of extension words even though data is being recirculated internally. IFETCH returns to normal operation with the first fetch after exiting loop mode.



## **SECTION 8**

# **INSTRUCTION EXECUTION TIMING**

This section, which describes the instruction execution timing of the CPU32 using external clock cycles, provides accurate execution and operation timing guidelines but not exact timings for every possible circumstance. This approach is used since exact execution time for an instruction or operation is highly dependent on concurrency of independently scheduled resources, memory speeds, and other variables. The timing numbers presented in this section allow the assembly language programmer or compiler writer to predict the performance of the CPU32. Additionally, the timings for exception processing are included so that designers of multitasking or real-time systems can predict task-switch overhead, maximum interrupt latency, and similar timing parameters. Instruction timings are given in clock cycles to eliminate clock frequency dependencies.

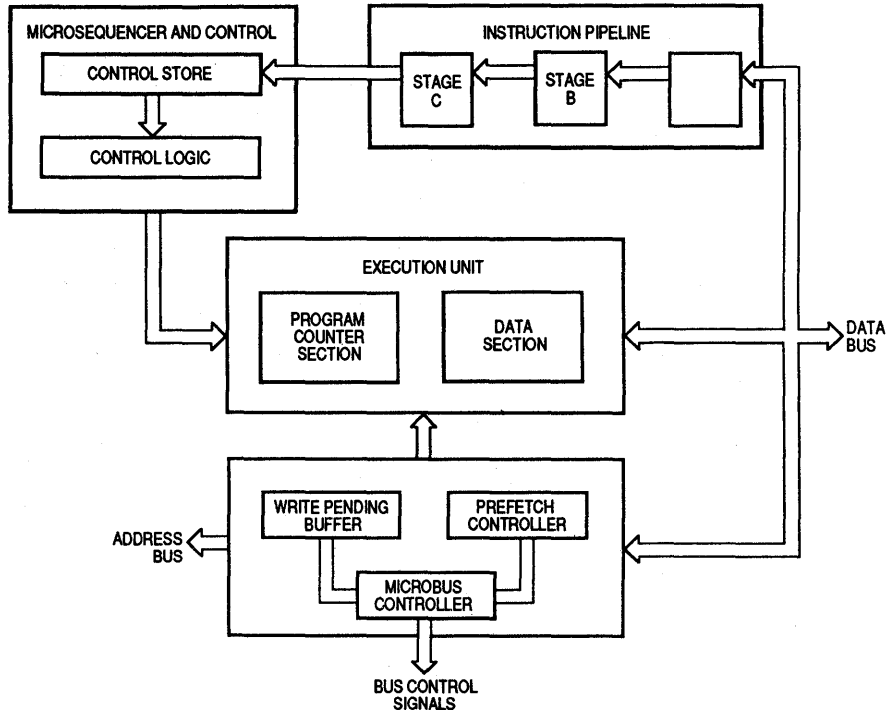
### **8.1 RESOURCE SCHEDULING**

Some of the variability in instruction execution timings results from the overlap of resource utilization. The processor can be viewed as consisting of several independently scheduled resources. Since little of the resource scheduling is directly related to instruction boundaries, it is impossible to make accurate estimates of the time required to execute a particular instruction without knowing the complete context within which the instruction is executing. The position of these resources within the CPU32 is shown in Figure 8-1.

#### **8.1.1 Microsequencer**

The microsequencer is either executing microinstructions or awaiting completion of accesses necessary to continue executing microcode. The microsequencer controls the bus controller, instruction execution, and internal processor operations such as calculation of effective address and setting of condition codes. The microsequencer initiates instruction word prefetches after a change of flow and controls the 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. As shown in Figure 8-1, instruction words (instruction operation words and all extension words) enter the pipeline at stage B. To reach stage C, an instruction word must have been completely decoded. Each of the pipeline stages has a status bit that reflects whether or not the word in that stage was loaded with data from a bus cycle which terminated abnormally. Stages of the pipeline are filled from an initial request by the microsequencer and are subsequently filled by the prefetch controller as they are emptied.

The instruction pipeline contains an additional stage which serves as a buffer. Prefetches completing on the bus before stage B of the instruction pipeline have been emptied are temporarily stored in this buffer.

### 8.1.3 Bus Controller Resources

The bus controller and microsequencer can operate concurrently. The bus controller can perform a read or write or schedule a prefetch while the microsequencer controls an effective address calculation or sets the condition codes. The microsequencer may also request a bus cycle that the bus controller cannot perform immediately. In this case, the bus cycle is queued, and the bus controller runs the cycle when the current cycle is complete.

The bus controller consists of the 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.

**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 requested by the prefetch controller whenever a pipeline stage is invalidated, either through completion of an instruction or use of extension words. Additional state information permits the controller to inhibit prefetch requests when a change in instruction flow (e.g., JMP) is anticipated. The prefetch occurs as soon as the bus is free of operand accesses already requested by the microsequencer.

For the typical program, a change of flow can be expected in approximately 10 to 25 percent of the instructions executed. Each time this happens, the instruction pipeline must be flushed and refilled from the new instruction stream. If priority were given to instruction prefetches rather than to operand accesses, it is likely that many instruction words would be flushed and never used, meanwhile delaying needed operand cycles. To maximize the available bus bandwidth, the CPU32 will schedule a prefetch only when the next instruction is not a change-of-flow instruction and when room exists in the pipeline for the prefetch.

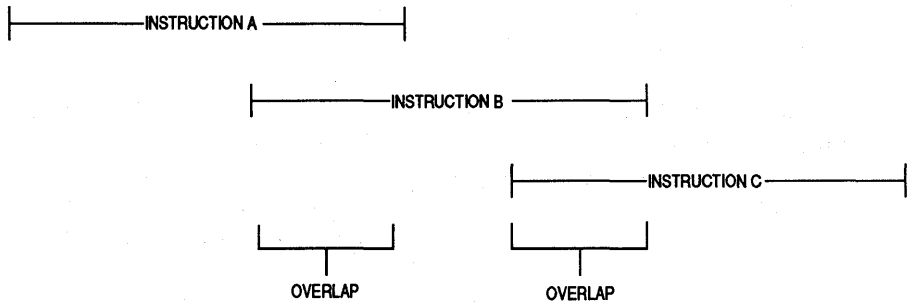
**8.1.3.2 WRITE-PENDING BUFFER.** The CPU32 incorporates a single-operand write-pending buffer, allowing the microsequencer to continue execution after the request for a write cycle is queued in the bus controller. The time occupied by the write at the end of an instruction can utilize the next instruction's head cycle time, thus reducing overall execution time. Interlocks prevent the microsequencer from overwriting this buffer.

**8.1.3.3 MICROBUS CONTROLLER.** The microbus controller performs the bus cycles issued to the bus controller by the microsequencer. Operand accesses always take 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 in the case of a word operand to a byte-sized external port. Long operands are accessed in two bus cycles, the most significant word first.

The goal of the bus controller is to maximize the useful bandwidth of the bus by not starting prefetches when those prefetches will be discarded due to a change of flow. Capable of recognizing certain instructions like branches and RTS, the instruction pipeline will inform the bus controller that no more prefetches are required.

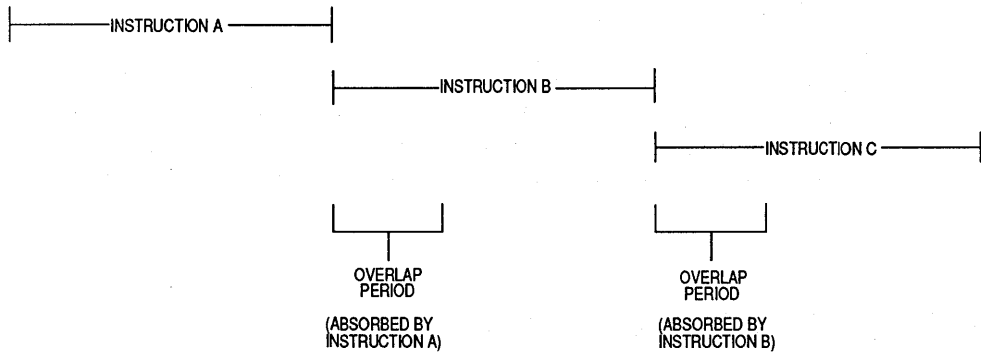
### 8.1.4 Instruction Execution Overlap

Overlap is the time, measured in clock cycles, that an instruction executes concurrently with the previous instruction. As illustrated in Figure 8-2, portions of instructions A and B execute simultaneously. The overlapped portion of instruction B is absorbed in the execution time of A. Similarly, the overlap time between instructions B and C reduces the overall execution time of the two instructions. Each instruction contributes to the total overlap time. The portion of the time at the end of the execution time of instruction A that can overlap at the beginning of instruction B is called the tail of instruction A. The portion of 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 consists of the lesser of the 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.



**Figure 8-3. Attributed Instruction Times**

### 8.1.5 Effects of Wait States

The CPU32 contains a small amount of on-chip memory with an access time of two clocks. While it is possible to get two-clock external accesses when the bus is operated in a synchronous mode, the typical external memory speed is three clocks or more.

All instruction times given in the timing tables in this section assume that both instruction fetches and operand cycles are to the two-clock memory and are for word access only (unless explicitly mentioned otherwise). Any time a long access is made, the 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 into that memory for each bus cycle.

A typical application will have a mixture of bus speeds: e.g., program executing from an off-chip ROM, the stack in the on-chip RAM, bulk storage of variables in a slower off-chip RAM, and 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 may be used to reduce the total number of cycles caused by a slower memory on the prefetch started by the previous instruction. For such cases, the increase in access time has no effect on the total execution time of that pair of instructions.

To trace the execution time of instructions by monitoring the external bus, note that the order of operand accesses is always the same for a particular instruction sequence, and, provided the bus speed is identical across those sequences, the interleaving of instruction prefetches with operands is also identical.

## 8.2 INSTRUCTION STREAM TIMING EXAMPLES

Some programming examples will allow a more detailed examination of these effects. For all examples, the memory access is from the internal two-clock memory or external synchronous memory, the bus is idle, and the instruction pipeline is full at the start.

### 8.2.1 Timing Example 1: Execution Overlap

The example shown in Figure 8-4 illustrates the overlapping of execution due to the bus controller's ability to execute bus cycles while the sequencer is calculating the next effective address. One clock is saved between each instruction since that is the minimum time of the individual head and tail numbers.

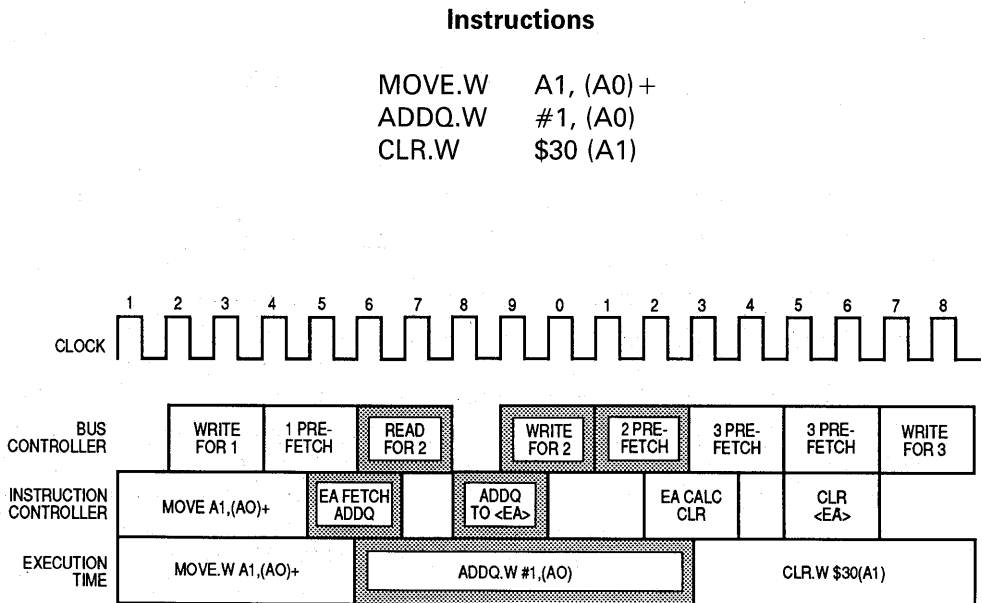


Figure 8-4. Example 1 — Instruction Stream

## 8.2.2 Timing Example 2: Branch Instructions

Example 2 shows what happens when a branch instruction is executed for both the taken and not-taken cases (see 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.L    D1, D0
BLE.B    NEXT
MOVE.L   D1, (A0)
    
```

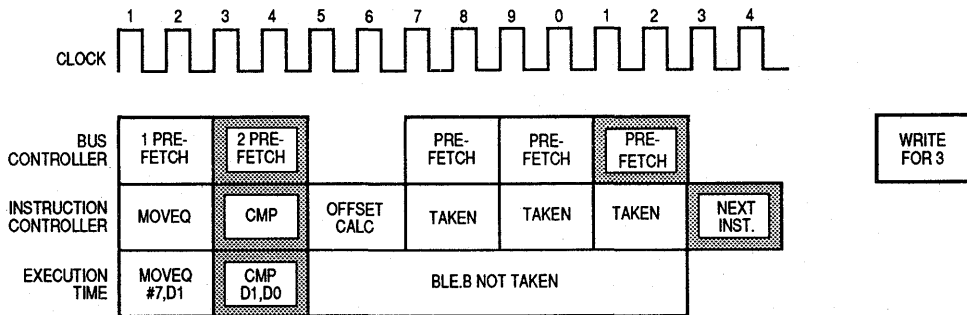


Figure 8-5. Example 2 — Branch Taken

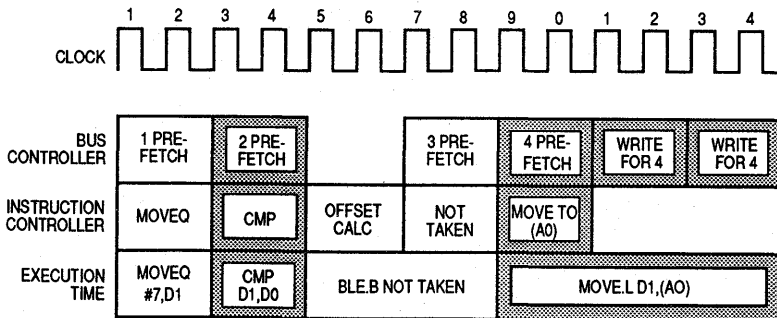


Figure 8-6. Example 2 — Branch Not Taken

### 8.2.3 Timing Example 3: Negative Tails

This example (see Figure 8-7) shows how to properly account for the negative tail figures for branches and other change-of-flow instructions. For this example, the bus speed is assumed to be four clocks per access. Instruction three is at the branch destination.

#### Instructions

```

MOVEQ    #7, D1
BRA.W    FARAWAY
MOVE.L   D1, D0
    
```

The CPU32 has a two-word instruction pipeline, but, due to internal delays, the minimum time for a branch instruction allows three bus cycles. The negative tail is intended to serve as a reminder that on a fast bus an extra two clocks are available for prefetching a third word, but that on a slower bus the third word is not forced to be fetched.

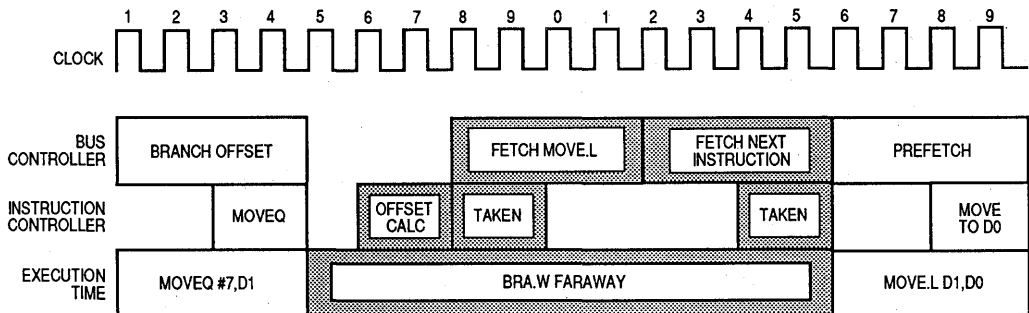


Figure 8-7. Example 3 — Branch Negative Tail

Example 3 actually illustrates three different considerations in calculating the time for an instruction. 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 be calculated for a three-word pipeline, and there is a one-clock delay caused by the displacement arriving late at the CPU.

The negative tail only needs to be calculated on changes of flow, but the concept can be generalized to any instruction so that only two words are required to be in the pipeline, but up to three words may be present. When there is an opportunity for the 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:

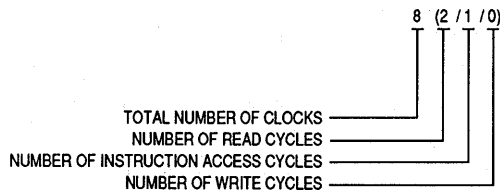
- A 16-bit data bus is used for all memory accesses.
- All memory accesses occur with two-clock bus cycles and no wait states.
- The instruction pipeline is full at the beginning of the instruction and is refilled by the end of the instruction.

Three values are listed for each instruction and addressing mode:

**Head** This value is the number of cycles at the beginning of the instruction available for the previous instruction's write to complete or for a prefetch to occur.

**Tail** This value is the number of cycles at the end of the instruction used by the instruction to complete a write.

**Cycles** This field contains four numbers per entry, three of which are contained in parenthesis. The outer number represents the minimum number of cycles required for the instruction to complete. Within the parenthesis, the numbers represent the number of bus accesses performed by the instruction. The first number inside the parenthesis 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 to keep the instruction and the instruction pipeline filled. The third number is the number of write accesses performed by the instruction.





The total number of bus-activity clocks and internal clocks (not overlapped by bus activity) of the instruction in this example are derived as follows:

$$\begin{aligned} & (2 \text{ reads} \times 2 \text{ clocks/read}) + \\ & (1 \text{ instruction access} \times 2 \text{ clocks/access}) + \\ & \underline{(0 \text{ writes} \times 2 \text{ clocks/write}) = 6 \text{ clocks of bus activity}} \\ & 8 \text{ clocks total} - 6 \text{ clocks bus activity} = 2 \text{ internal clocks} \end{aligned}$$

One example from the timing tables is the ADD.L (12, A3, D7.W • 4), D2 instruction, with the instructions and data from two-clock memory. The effective addressing mode is listed as head = 4, tail = 4, cycles = 10 (2/1/0). The difference from CEA timing table (see **8.3.2 Calculate Effective Address**) is because the table is listed for word accesses, and this example is for a long access. The instruction itself has a head = 0, tail = 0, and cycles = 2(0/1/0) from the arithmetic/logical timing table (see **8.3.5 Arithmetic/Logical Instructions**). Assuming no trailing write exists from the previous instruction, the execution time is calculated as follows:

The effective address calculation requires six clocks, with the replacement fetch for the effective address occurring during this time (leaving a head of four). If there had not been time in the head to perform the prefetch due to a previous trailing write, then time must be allotted in the middle of the instruction or after the tail to do the prefetches. The read of the memory 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 since the instruction has a head of zero. An additional two clocks are required for the actual ADD, which makes the total 6 + 4 + 2 = 12 clocks. If the bus cycles take more time (i.e., the memory is off-chip), then add the appropriate number of clocks to each memory access.

An example of the overlapped execution possible on the CPU32 is with the instruction sequence MOVE.L D0, (A0) followed by LSL.L #7, D2. The MOVE has a head of zero and a tail of four, since it is a long write. The LSL has a head of four; therefore, the trailing write from the MOVE will overlap the LSL completely. Thus, this two-instruction sequence has a head of zero and a tail of zero and a total execution of eight clocks instead of 12 clocks obtained by adding the individual cycle times.

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 may begin there in advance of the instruction needing the prefetch.
- 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
Dn	—	—	0(0/0/0)
An	—	—	0(0/0/0)
(An)	1	1	3(x/0/0)
(An) +	1	1	3(x/0/0)
– (An)	2	2	4(x/0/0)
(d <sub>16</sub> ,An) or (d <sub>16</sub> ,PC)	1	3	5(x/1/0)
(xxx).W	1	3	5(x/1/0)
(xxx).L	1	5	7(x/2/0)
#(data).W	1	1	3(0/1/0)
#(data).B	1	1	3(0/1/0)
#(data).L	1	3	5(0/2/0)
(d <sub>8</sub> ,An,Xn.Sz × Sc) or (d <sub>8</sub> ,PC,Xn.Sz × Sc)	4	2	8(x/1/0)
(O)	2	2	6(x/1/0)
(d <sub>16</sub> )	1	3	7(x/2/0)
(d <sub>32</sub> )	1	5	9(x/3/0)
(An)	1	1	5(x/1/0)
(Xm.Sz × Sc)	4	2	8(x/1/0)
(An,Xm.Sz × Sc)	4	2	8(x/1/0)
(d <sub>16</sub> ,An) or (d <sub>16</sub> ,PC)	1	3	7(x/2/0)
(d <sub>32</sub> ,An) or (d <sub>32</sub> ,PC)	1	5	9(x/3/0)
(d <sub>16</sub> ,An,Xm) or (d <sub>16</sub> ,PC,Xm)	2	2	8(x/2/0)
(d <sub>32</sub> ,An,Xm) or (d <sub>32</sub> ,PC,Xm)	1	3	9(x/3/0)
(d <sub>16</sub> ,An,Xm.Sz × Sc) or (d <sub>16</sub> ,PC,Xm.Sz × Sc)	2	2	8(x/2/0)
(d <sub>32</sub> ,An,Xm.Sz × Sc) or (d <sub>32</sub> ,PC,Xm.Sz × Sc)	1	3	9(x/3/0)

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.

#### NOTES:

1. Tail on reads is the minimum time a read or prefetch will take after the number of cycles for word operands.
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. For the indexed addressing modes, the prefetches to replace it are permitted to occur immediately so that the head is zero if the previous instruction had a tail smaller than the listed effective address head. This scheme allows a slower bus to work faster than might otherwise be calculated.

### 8.3.2 Calculate Effective Address

The calculate effective address table indicates the number of clock periods needed for the processor to calculate the 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
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 <sub>16</sub> ,An) or (d <sub>16</sub> ,PC)	1	1	3(0/1/0)
(xxx).W	1	1	3(0/1/0)
(xxx).L	1	3	5(0/2/0)
(dg,An,Xn.Sz×Sc) or (dg,PC,Xn.Sz×Sc)	4	0	6(0/1/0)
(O)	2	0	4(0/1/0)
(d <sub>16</sub> )	1	1	5(0/2/0)
(d <sub>32</sub> )	1	3	7(0/3/0)
(An)	1	0	4(0/1/0)
(Xm.Sz×Sc)	4	0	6(0/1/0)
(An,Xm.Sz×Sc)	4	0	6(0/1/0)
(d <sub>16</sub> ,An) or (d <sub>16</sub> ,PC)	1	1	5(0/2/0)
(d <sub>32</sub> ,An) or (d <sub>32</sub> ,PC)	1	3	7(0/3/0)
(d <sub>16</sub> ,An,Xm) or (d <sub>16</sub> ,PC,Xm)	2	0	6(0/2/0)
(d <sub>32</sub> ,An,Xm) or (d <sub>32</sub> ,PC,Xm)	1	1	7(0/3/0)
(d <sub>16</sub> ,An,Xm.Sz×Sc) or (d <sub>16</sub> ,PC,Xm.Sz×Sc)	2	0	6(0/2/0)
(d <sub>32</sub> ,An,Xm.Sz×Sc) or (d <sub>32</sub> ,PC,Xm.Sz×Sc)	1	1	7(0/3/0)

**NOTES:**

1. Tail on reads is the minimum time a read or prefetch will take after the number of cycles for word operands.
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. For the indexed addressing modes, the prefetches to replace it are permitted to occur immediately so that the head is zero, if the previous instruction had a tail smaller than the listed effective address head. This scheme allows a slower bus to work faster than might otherwise be calculated.

### 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 the MOVE or MOVEA instruction. The fetch effective address table is needed on most MOVE operations (source, destination dependent). The destination effective addresses are divided by their formats (refer to **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.

Instruction	Head	Tail	Cycles
MOVE Rn, Dn	0	0	2(0/1/0)
MOVE Rn, An	0	0	2(0/1/0)
MOVE <FEA>, An	0	0	2(0/1/0)
MOVE <FEA>, Dn	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 #, <CEA>	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 <FEA>, -(An)	2	2	6(0/1/x)
MOVE <CEA>, <CEA>	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.

\* = There should be an immediate effective address calculation also included for this instruction.

**NOTE:**

For instructions not explicitly listed (MOVE <CEA>, <CEA>), the source effective address is calculated by the calculate effective address table, and the destination effective address is calculated by the same table except the bus cycle is for the source effective address. If the prefetches for the addressing modes have been completed, then a maximum of three cycles are allotted for the memory read without affecting the total execution time.

### 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 the control registers or specified effective address. Footnotes indicate when to account for the appropriate effective address times. The total number of clock cycles is outside the parentheses. The 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	2	0	4(0/1/0)
MOVEC Cr, Rn	10	0	14(0/2/0)
MOVEC Rn, Cr	12	0	<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	$8 + n \cdot 4(n + 1, 2, 0)^*$
MOVEM.W RL, (CEA)	1	0	$8 + n \cdot 4(0, 2, n)^*$
MOVEM.L (CEA), RL	1	0	$12 + n \cdot 4(2n + 2, 2, 0)$
MOVEM.L RL, (CEA)	1	2	$10 + n \cdot 4(0, 2, 2n)$
MOVEP.W Dn, (d16, An)	2	0	10(0/2/2)
MOVEP.W (d16, An), Dn	1	2	11(2/2/0)
MOVEP.L Dn, (d16, An)	2	0	14(0/2/4)
MOVEP.L (d16, An), Dn	1	2	19(4/2/0)
MOVES (CEA), Rn	1/7	1	13(x/2/0)
MOVES Rn, (CEA)	1/9	2	14(0/2/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.

\* = Each bus cycle may take up to four clocks without increasing total execution time.

Cr = Control registers USP, VBR, SFC, DFC, CAAR, and CACR

n = Number of registers to transfer

RL = Register List

< = Maximum time is indicated; certain data or mode combinations may execute faster.

## 8.3.5 Arithmetic/Logical Instructions

The arithmetical/logical instruction table indicates the number of clock periods needed for the processor to perform the specified arithmetical/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)
AND Rn, <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)
R <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 Rn Rn, <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 <FEA>, Rn	1/2	0	<20(x/2/0)
MULsu.W <FEA>, Dn	0	0	26(0/1/0)
MULsu.L <FEA>, Dn	1/2	0	<52(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 <FEA>, Dn	1/2	0	<48(0/1/0)
DIVS.L <FEA>, Dn	1/2	0	<64(0/1/0)
TBLsu Dn/Dm, Dp	26	0	<30(0/2/0)
TBLsu <CEA>, Dn	1/6	0	<38(2x/2/0)
TBLNsu Dn/Dm, Dp	30	0	<34(0/2/0)
TBLNsu <CEA>, Dn	1/6	0	<42(2x/2/0)

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 is indicated; certain data or mode combinations will execute faster.

su = The execution time is identical for signed or unsigned operands.

### 8.3.6 Immediate Arithmetic/Logical Instructions

The immediate arithmetical/logical 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	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.

\* = Add immediate effective address.



### 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	Dn	0	0	2(0/1/0)
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
LSd Dn, Dm	-2*	0	(0/1/0)*
LSd #, Dm	4	0	6(0/1/0)
LSd <FEA>	0	2	6(0/1/1)
ASd Dn, Dm	-2*	0	(0/1/0)*
ASd #, Dm	4	0	6(0/1/0)
ASd <FEA>	0	2	6(0/1/1)
ROd Dn, Dm	-2*	0	(0/1/0)*
ROd #, Dm	4	0	6(0/1/0)
ROd <FEA>	0	2	6(0/1/1)
ROXd Dn, Dm	-2**	0	(0/1/0)**
ROXd #, Dm	-2***	0	(0/1/0)***
ROXd <FEA>	0	2	6(0/1/1)

d = Direction (left or right).

\* = Execution time is calculated by this formula:  $\max(3 + (n/4) + \text{mod}(n, 4) + \text{mod}((n/4) + \text{mod}(n, 4) + 1, 2), 6)$  or by the following table.

\*\* = Execution time is calculated by this formula (count  $\leq$  63):  $\max(3 + n + \text{mod}(n + 1, 2), 6)$ .

\*\*\* = Execution time is calculated by this formula (count  $\leq$  8):  $\max(2 + n + \text{mod}(n, 2), 6)$ .

Clocks	Shift Counts							
6	0	1	2	3	4	5	6	8
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
31	34	35	37	38	40	41	44	
39	42	43	45	46	48	49	52	
47	50	51	53	54	56	57	60	
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	4(0/1/0)

### 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(0/3/0)
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	6(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 <FEA>, Dn (no ex)	1/2	0	20(x/1/0)
CHK2 <FEA>, Dn (ex)	1/2	-2	54(x + 2/2/6)
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	-2	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.

### 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	0	-2	25(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.

### 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(12/2/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 will execute faster.

y = If a bus error occurred during a write cycle, the cycle is rerun by the RTE.





# APPENDIX A

## M68000 FAMILY SUMMARY

Appendix A summarizes the characteristics of the microprocessors in the M68000 Family. M68000 UM/AD, *M68000 User's Manual* Sixth Edition includes more detailed information about the MC68000 and MC68010 differences.

	MC68000	MC68010	CPU32	MC68020
Data Bus Size (Bits)	16	16	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)

**A**

### 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 = 7 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 $\overline{AS}$ Signal
MC68010	Use $\overline{AS}$ Signal
CPU32	Use $\overline{RMC}$ Signal
MC68020	Use $\overline{RMC}$ 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

## M68000 Instruction Set Extensions

Mnemonic	Description	CPU32	M68020
Bcc	Supports 32-Bit Displacements	✓	✓
BFxxxx	Bit Field Instructions (BFCHG, BFCLR, BFEXTS, BFEXTU, BFFFO, BFINS, BFSET, BFTST)		✓
BGND	Background Operation	✓	
BKPT	New Instruction Functionality	✓	✓
BRA	Supports 32-Bit Displacements	✓	✓
BSR	Supports 32-Bit Displacements	✓	✓
CALLM	New Instruction		✓
CAS, CAS2	New Instructions		✓
CHK	Supports 32-Bit Operands	✓	✓
CHK2	New Instruction	✓	✓
CMPI	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 3-Bit Displacements	✓	✓
LPSTOP	New Instruction	✓	
MOVEC	Supports New Control Registers	✓	✓
MULS/MULU	Supports 32-Bit Operands, 64-Bit Results	✓	✓
PACK	New Instruction		✓
RTM	New Instruction		✓
TABLE	New Instruction	✓	
TST	Supports Program Counter Relative, Immediate, and an Addressing	✓	✓
TRAPcc	New Instruction	✓	✓
UNPK	New Instruction		✓

## 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 Predecrement	-(An)	✓	✓	✓
Address Register Indirect with Displacement	(d <sub>16</sub> ,An)	✓	✓	✓
Address Register Indirect with Index (8-Bit Displacement)	(d <sub>8</sub> ,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	(d <sub>16</sub> ,PC)	✓	✓	✓
Program Counter Indirect with Index (8-Bit Displacement)	(d <sub>8</sub> ,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

## — A —

- Absolute Long Address Mode, 3-10
- Absolute Short Address Mode, 3-10
- AC Electrical Specifications,
  - See system integration user's manual
- Address Bus,
  - See system integration user's manual
- Address Error Exception, 6-8
- Address Register,
  - Direct Addressing Mode, 3-4
  - Indirect Displacement Mode, 3-6
  - Indirect Index (Base Displacement) Mode, 3-9
  - Indirect Index (8-Bit Displacement) Mode, 3-9
  - Indirect Addressing Mode, 3-4
  - Indirect Postincrement Addressing Mode, 3-5
  - Indirect Predecrement Addressing Mode, 3-6
- Address Registers, 2-6
- Address Space Types, 5-4
- Addressing,
  - Capabilities, 3-14
  - Compatibility, M68000, 3-17, A-4
  - Indexed, 3-6, 3-9
  - Indirect, 3-5ff.
  - Mode Enhancements, 1-5
  - Mode Summary, 3-15
- Addressing Modes,
  - Register Direct, 3-4
  - Memory, 3-5
  - Programming View, 3-13
  - Special, 3-8
- Architectural Comparisons (M68000), A-1
- Arithmetic/Logical Instruction,
  - Immediate, Timing Table, 8-17
  - Timing Table, 8-16
- Assignments, Exception Vector, 6-2
- Asynchronous Bus Operation,
  - See system integration user's manual

## — B —

- Background Debug Mode, 7-3
  - Command,
    - Execution, 7-6
    - Format, 7-14
    - Sequence Diagrams, 7-16

- Sequence Example, 7-17
- Set, 7-14
- Enabling, 7-3
  - Entering, 7-6
  - Returning from, 7-7
  - Sources, 7-4
- BGND Instruction, 7-5
- Binary-Coded Decimal and Extended Instructions, 8-18
- Binary-Coded Decimal Operations, 4-9
- Bit Manipulation Instructions, 8-21
- Bit Manipulation Operations, 4-9
- Block Diagram, 1-8
- Branch Instructions, 8-7
- Breakpoint Exception Processing, 7-5
- Breakpoints,
  - Hardware, 7-3, 6-10
  - On Data Accesses, 7-3
  - On Instructions, 7-3
  - Peripheral, 7-5
- Breakpoint Instruction, 7-1
- Breakpoint Pin, External, 7-5
- Bus Controller Resources, 8-3
- Bus Cycle Fault Stack Frame, 6-28
- Bus Error, 6-6, 6-28
- Bus Faults, Double, 7-5

## — C —

- Calculate Effective Address, 8-13
- Changing Privilege Level, 5-3
- Compatibility, M68000 Addressing, 3-14
- Condition Code,
  - Computations, 4-14
  - Register, 4-14
- Condition Tests, 4-17
- Conditional Branch Instructions, 8-22
- Control,
  - Instructions, 8-23
  - Registers, 2-6
- Conventions, Notation, 3-2
- Correcting Faults, 6-22
- CPU,
  - Serial Logic, 7-8
  - Space, 5-4
  - Space Address Encoding, 5-5

— D —

- Data,
  - Format, 7-11
  - Movement Instructions, 4-5
  - Register Direct Addressing, 3-4
  - Registers, 2-4
  - Structures, Other, 3-18
  - Types, 2-4
- Deterministic Opcode Tracking, 7-3, 7-33
- Development Features, Standard, 7-1
- Development Support, 7-1
- Development System Serial Logic, 7-11
- Double Bus Faults, 7-5
- Dump Memory Block (DUMP), 7-24
- Dynamic Bus Sizing, 6-20, 6-31

— E —

- Effective Address, 3-4
  - Calculate Table (CEA), 8-13
  - Encoding Summary, 3-11
  - Fetch Table (FEA), 8-12
- Enhanced Addressing Modes, 1-5
- Enhanced Instruction Set, 1-5
- Errors, Bus, 6-6
- Example, Table Instruction, 4-192
- Exception,
  - Address Error, 6-6
  - Breakpoint Instruction (BKPT), 6-10
  - Bus Error, 6-6
  - Format Error, 6-11
  - Handling, 1-4
  - Illegal Instruction, 6-11
  - Instruction Traps, 6-9
  - Interrupts, 6-15
  - Kinds of, 6-1
  - Multiple, 6-3
  - Priority, 6-3
  - Privilege Violation, 6-12
  - Processing, 5-6
    - Sequence, 6-5
    - State, 6-1
  - Reset, 6-6
  - Related Instructions and Operations, 8-24
  - Return from, 6-16
  - Stack Frame, 5-7, 6-4
  - Trace, 6-13
  - Unimplemented Instruction, 6-11
  - Vectors, 5-7, 6-1
- Execution Time Calculations, 8-9ff.
- Execution Overlap Example, 8-6

— F —

- Faults,
  - Correcting,
    - Type II via RTE, 6-24
    - Type III via RTE, 6-26
    - Type III via Software, 6-25
    - Type IV via Software, 6-26
  - Double Bus, 7-5
  - During,
    - Exception Processing, 6-23
    - MOVEM Operand Transfers, 6-22
  - Prefetch, Operand, RMW, and MOVEP, 6-21
  - Released Write, 6-20
  - Recovery, 6-18
  - Types of, 6-20
- Features, 1-2
- Fetch Effective Address, 8-12
- Fill Memory Block (FILL), 7-26
- Format Error, 6-11
- Four-Word Stack Frame, Normal, 6-27
- Function Code Registers, 2-4ff.
- Future Commands, 7-32

— G —

- General Description, 1-1

— H —

- Halt Operation,
  - See system integration user's manual

— I —

- Illegal or Unimplemented Instructions, 6-12
- Immediate,
  - Arithmetic/Logical Instructions, 8-17
  - Data Addressing, 3-11
- Implicit Reference, 3-3
- Indexed Addressing, 3-6, 3-9
- Indirect Addressing, 3-5ff.
- Instruction,
  - Descriptions, 4-18ff.
  - Details, 4-12
  - Execution Overlap, 8-4
  - Family Compatibility, 4-1
  - Fetch (IFETCH), 7-29

- Instruction,
  - Format, 4-3
  - Format Summary, 4-177
  - New, 1-6, 4-2
  - Pipe, 8-2
  - Pipe (IPIPE), 7-33
  - Set Enhancements, 1-5
  - Summary, 4-4
  - Timing Tables, 8-9
  - Traps, 6-9
- Instruction Set Extensions, A-3
- Instruction Stream Timing Examples, 8-6
- Instructions,
  - Binary-Coded Decimal (BCD), 4-9, 8-18
  - Bit Manipulation, 4-9, 8-21
  - Conditional Branch, 8-22
  - Data Movement, 4-5, 8-14
  - Exception Related, 8-24
  - Integer Arithmetic, 4-6, 8-16
  - Logical, 4-7, 8-16
  - Program Control, 4-10, 8-23
  - Shift and Rotate, 4-8, 8-20
  - Single Operand, 8-19
  - System Control, 4-10, 8-23
- Integer Arithmetic Operations, 4-6
- Interrupts, 6-15

— L —

- Logical Operations, 4-7
- Loop Mode Instruction Execution, 1-3
- Low-Power Stop (LPSTOP), 1-6, 4-2

— M —

- M68000 Family Addressing Capability, 3-17
- M68000 Family Compatibility, 4-1
- Memory,
  - Addressing Modes, 3-5
  - Data Organization, 2-7
  - Indirect Addressing, A-4
  - Virtual, 1-2
- Microbus Controller, 8-4
- Microsequencer, 8-1
- Model, Programming, 2-1
- Move Instruction, 8-14
- Move Instruction, Special Purpose, 8-15
- Multiple Exceptions, 6-3

— N —

- Negative Tails, 8-8
- No Operation (NOP), 7-32
- Notation and Format, 4-12

- Notation Conventions, 3-2
- Normal Processing State, 6-1

— O —

- Opcode Tracking during Loop Mode, 7-35
- Opcode Tracking, in Background Mode, 7-3, 7-35
- Organization,
  - Memory, 2-7
  - Registers, 2-4
- Overlap, 8-4

— P —

- Pipeline Sync with the NOP Instruction, 4-201
- Prefetch Controller, 8-3
- Priority Exception, 6-3
- Privilege Levels,
  - Changing, 5-3
  - States, 1-8
  - Supervisor, 5-2
  - User, 5-2
- Privilege Violations, 6-11
- Processing of Specific Exceptions, 6-5
- Processing States, 1-6
- Program and Data References, 3-2
- Program Control Instructions, 4-10
- Program Counter Indirect with Displacement, 3-8
  - Index (8-Bit Displacement), 3-9
  - Index (Base Displacement), 3-9
- Programmer's Model, 2-1
- Programmer's View of Addressing Modes, 3-13

— Q —

- Queues, 3-20

— R —

- Read,
  - Address/Data Register (RAREG/RDREG), 7-17
  - Memory Location (READ), 7-21
  - System Register (RSREG), 7-19
- Recovery,
  - Bus Fault, 6-24
  - RTE, 6-26
  - Software, 6-25
- References,
  - Data, 3-2
  - Implicit, 3-3
  - Program, 3-2
- Register Direct Addressing, 3-4



- Registers,
  - Address, 2-6
  - Condition Code, 4-14
  - Control, 2-6
  - Data, 2-4
  - Function Code, 2-4ff.
  - Organization, 2-3
  - Status, 2-3
  - Vector Base, 1-4
- Release Writes (Type I),
  - Completing via Software, 6-23
  - Completing via the RTE, 6-24
- Reset, 6-6
- Reset Peripherals (RST), 7-31
- Resource Scheduling, 8-1
- Return from Exception, 8-16
- Rotate Instructions, 4-8

— S —

- Save and Restore Operations, 8-25
- Serial Interface, 7-8
- Shift and Rotate Operations, 4-8
- Shift and Rotate Instructions, 8-20
- Single Operand Instructions, 8-19
- Six-Word Stack Frame, Normal, 6-28
- Sizing, Dynamic Bus, 6-20, 6-31
- Software Breakpoints, 6-10
- Software Bus Fault Recovery, 6-25
- Space Formats, 5-5
  - Type 0000—Breakpoint, 5-5
  - Type 0001—MMU Access, 5-5
  - Type 0010—Coprocessor Access, 5-6
  - Type 0011—Internal Register Access, 5-6
  - Type 1111—Interrupt Acknowledge, 5-6
- Special Address Modes, 3-8
- Special-Purpose MOVE Instruction, 8-15
- Stack,
  - Frames, 6-27, 5-7
  - User, 3-19
  - System, 3-18
- States, Processing, 5-1ff
- Status Register, 2-3
- Subroutine Calls, Nested, 4-201
- Supervisor Privilege Level, 5-2
- Surface Interpolations, 4-200
- System,
  - Control Instructions, 4-10
  - Stack, 3-18
- Synchronization, Pipeline with NOP, 4-201

— T —

- Table,
  - Examples,
    - Table Standard Usage, 4-193
    - Compressed Table, 4-194
    - 8-Bit Independent Variable, 4-196
    - Maintaining Precision, 4-198
    - Surface Interpolations, 4-200
  - Instruction, Using the, 4-192
  - Lookup and Interpolate (TBL), 1-6, 4-2
- Tests, Condition, 4-17
- Timing Examples,
  - Branch Instructions, 8-7
  - Execution Overlap, 8-6
  - Negative Tails, 8-8
  - See system integration user's manual
- Timing Tables, 8-9
- Trace on Instruction Execution, 6-13, 7-1

— U —

- Unimplemented Instruction Emulation, 7-1
- Unimplemented Instructions, 4-2
- User Privilege Level, 5-3
- User Stacks, 3-19

— V —

- Vector Base Register, 1-4
- Vectors, Exception, 6-2, 5-7
- Virtual Memory, 1-2

— W —

- Wait States, Effects of, 8-5
- Write,
  - Address/Data Register (WAREG/WDREG), 7-18
  - Memory Location (WRITE), 7-22
  - Pending Buffer, 8-3
  - System Register (WSREG), 7-20

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

**1** Overview

**2** Architecture Summary

**3** Data Organization and Addressing Capabilities

**4** Instruction Set

**5** Processing States

**6** Exception Processing

**7** Development Support

**8** Instruction Execution Timing

**A** M68000 Family Summary

**I** Index