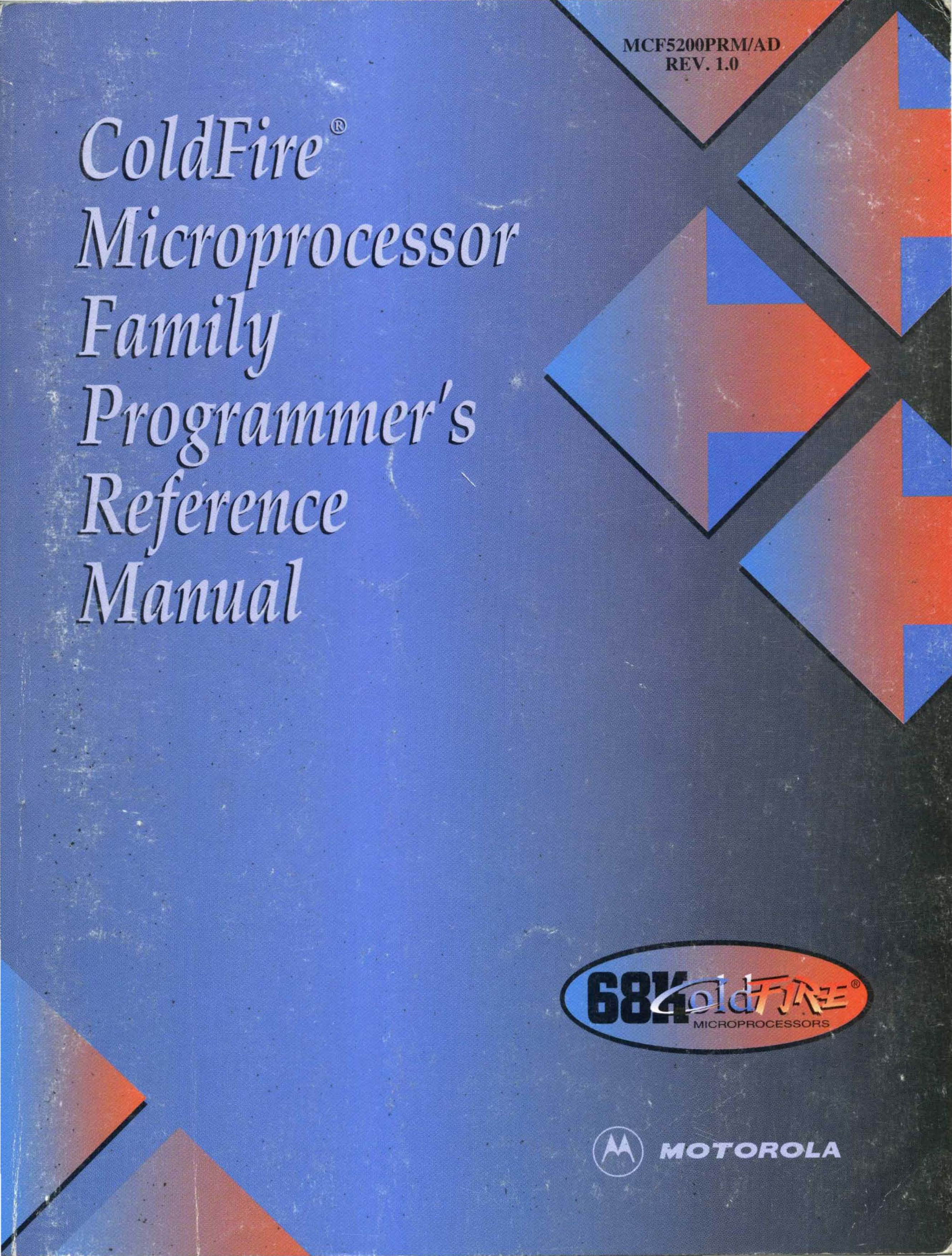


MCF5200PRM/AD
REV. 1.0

ColdFire® Microprocessor Family Programmer's Reference Manual

68K ColdFire®
MICROPROCESSORS

MOTOROLA

MOTOROLA

ColdFire® Microprocessor Family
Programmer's
Reference Manual
Revision 1.0

® ColdFire and Motorola are registered trademarks of Motorola, Inc.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not 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.

© MOTOROLA, 1997 All Rights Reserved

Documentation Feedback

For Nontechnical Comments on the Documentation	FAX 512-891-8593 http://www.mot.com/hpesd/docs_survey.html

The Information Development team welcomes your suggestions for improving our documentation and encourages you to complete the documentation feedback form at the World Wide Web address listed above. Your help helps us measure how well we are serving your information requirements.

The Information Development team also provides a fax number for you to submit any questions or comments about this document or how to order other documents. Please provide the part number and revision number (located in upper right-hand corner of the cover) and the title of the document. When referring to items in the manual, please reference by the page number, paragraph number, figure number, table number, and line number if needed. **Please do not fax technical questions to this number.**

When sending a fax, please provide your name, company, fax number, and phone number including area code.

For Internet Access	Web Only: http: // www.mot.com/aesop
For HotLine Questions	FAX (US or Canada): 1-800-248-8567

Applications and Technical Information

Technical Support For questions or comments pertaining to technical information, questions, and applications, please contact the sales offices nearest you.

Sales Offices Field applications engineering is available through all sales offices.

UNITED STATES	
ALABAMA, Huntsville	(205) 464-6800
ARIZONA, Tempe	(602) 897-5056
CALIFORNIA, Agoura Hills	(818) 706-1929
CALIFORNIA, Los Angeles	(310) 417-8848
CALIFORNIA, Irvine	(714) 753-7360
CALIFORNIA, Roseville	(916) 922-7152
CALIFORNIA, San Diego	(619) 541-2163
CALIFORNIA, Sunnyvale	(408) 749-0510
COLORADO, Colorado Springs	(719) 599-7497
COLORADO, Denver	(303) 337-3434
CONNECTICUT, Wallingford	(203) 949-4100
FLORIDA, Maitland	(407) 628-2636
FLORIDA, Pompano Beach/ Fort Lauderdale	(305) 486-9776
FLORIDA, Clearwater	(813) 538-7750
GEORGIA, Atlanta	(404) 729-7100
IDAHO, Boise	(208) 323-9413
ILLINOIS, Chicago/Hoffman Estates	(708) 490-9500
INDIANA, Fort Wayne	(219) 436-5818
INDIANA, Indianapolis	(317) 571-0400
INDIANA, Kokomo	(317) 457-6634
IOWA, Cedar Rapids	(319) 373-1328
KANSAS, Kansas City/Mission	(913) 451-8555
MARYLAND, Columbia	(410) 381-1570
MASSACHUSETTS, Marborough	(508) 481-8100
MASSACHUSETTS, Woburn	(617) 932-9700
MICHIGAN, Detroit	(313) 347-6800
MINNESOTA, Minnetonka	(612) 932-1500
MISSOURI, St. Louis	(314) 275-7380
NEW JERSEY, Fairfield	(201) 808-2400
NEW YORK, Fairport	(716) 425-4000
NEW YORK, Hauppauge	(516) 361-7000
NEW YORK, Poughkeepsie/Fishkill	(914) 473-8102
NORTH CAROLINA, Raleigh	(919) 870-4355
OHIO, Cleveland	(216) 349-3100
OHIO, Columbus/Worthington	(614) 431-8492
OHIO, Dayton	(513) 495-6800
OKLAHOMA, Tulsa	(800) 544-9496
OREGON, Portland	(503) 641-3681
PENNSYLVANIA, Colmar	(215) 997-1020
Philadelphia/Horsham	(215) 957-4100
TENNESSEE, Knoxville	(615) 584-4841
TEXAS, Austin	(512) 873-2000
TEXAS, Houston	(800) 343-2692
TEXAS, Plano	(214) 516-5100
VIRGINIA, Richmond	(804) 285-2100
WASHINGTON, Bellevue	(206) 454-4160
Seattle Access	(206) 622-9960
WISCONSIN, Milwaukee/Brookfield	(414) 792-0122
CANADA	
BRITISH COLUMBIA, Vancouver	(604) 293-7605
ONTARIO, Toronto	(416) 497-8181
ONTARIO, Ottawa	(613) 226-3491
QUEBEC, Montreal	(514) 731-6881
INTERNATIONAL	
AUSTRALIA, Melbourne	(61-3)887-0711
AUSTRALIA, Sydney	(61(2)906-3855
BRAZIL, Sao Paulo	55(11)815-4200
CHINA, Beijing	86 505-2180
FINLAND, Helsinki	358-0-35161191
Car Phone	358(49)211501
FRANCE, Paris/Vanves	33(1)40 955 900
GERMANY, Langenhagen/ Hanover	49(511)789911
GERMANY, Munich	49 89 92103-0
GERMANY, Nuremberg	49 911 64-3044
GERMANY, Sindelfingen	49 7031 69 910
GERMANY, Wiesbaden	49 611 761921
HONG KONG, Kwai Fong	852-4808333
Tai Po	852-6668333
INDIA, Bangalore	(91-812)627094
ISRAEL, Tel Aviv	972(3)753-8222
ITALY, Milan	39(2)82201
JAPAN, Aizu	81(241)272231
JAPAN, Atsugi	81(0462)23-0761
JAPAN, Kumagaya	81(0485)26-2600
JAPAN, Kyushu	81(092)771-4212
JAPAN, Mito	81(0292)26-2340
JAPAN, Nagoya	81(052)232-1621
JAPAN, Osaka	81(06)305-1801
JAPAN, Sendai	81(22)268-4333
JAPAN, Tachikawa	81(0425)23-6700
JAPAN, Tokyo	81(03)3440-3311
JAPAN, Yokohama	81(045)472-2751
KOREA, Pusan	82(51)4635-035
KOREA, Seoul	82(2)554-5188
MALAYSIA, Penang	60(4)374514
MEXICO, Mexico City	52(5)282-2864
MEXICO, Guadalajara	52(36)21-8977
Marketing	52(36)21-9023
Customer Service	52(36)669-9160
NETHERLANDS, Best	(31)49988 612 11
PUERTO RICO, San Juan	(809)793-2170
SINGAPORE	(65)2945438
SPAIN, Madrid	34(1)457-8204
or	34(1)457-8254
SWEDEN, Solna	46(8)734-8800
SWITZERLAND, Geneva	41(22)7991111
SWITZERLAND, Zurich	41(1)730 4074
TAIWAN, Taipei	886(2)717-7089
THAILAND, Bangkok	(66-2)254-4910
UNITED KINGDOM, Aylesbury	44(296)395-252
FULL LINE REPRESENTATIVES	
COLORADO, Grand Junction	
Cheryl Lee Whitely	(303) 243-9658
KANSAS, Wichita	
Melinda Shores/Kelly Greiving	(316) 838 0190
NEVADA, Reno	
Galena Technology Group	(702) 746 0642
NEW MEXICO, Albuquerque	
S&S Technologies, Inc.	(505) 298-7177
UTAH, Salt Lake City	
Utah Component Sales, Inc.	(801) 561-5099
WASHINGTON, Spokane	
Doug Kenley	(509) 924-2322
ARGENTINA, Buenos Aires	
Argonics, S.A.	(541) 343-1787
HYBRID COMPONENTS RESELLERS	
Elmo Semiconductor	(818) 768-7400
Minco Technology Labs Inc.	(512) 834-2022
Semi Dice Inc.	(310) 594-4631

PREFACE

Introduction

The *ColdFire® Microprocessor Family Programmer's Reference Manual Rev. 1.0* describes the programming, capabilities, and operation of the ColdFire Family processors.

Additional ColdFire Documentation

The following documents provide details on specific ColdFire Family devices:

- *MCF5202 User's Manual* (order *MCF5202UM/AD*)
- *MCF5204 User's Manual* (order *MCF5204UM/AD*)
- *MCF5206 User's Manual* (order *MCF5206UM/AD*)
- *MCF5202 Product Brief* (order *MCF5202/D*)
- *MCF5204 Product Brief* (order *MCF5204/D*)
- *MCF5206 Product Brief* (order *MCF5206/D*)
- *68K and ColdFire Product Portfolio Overview (updated quarterly)*

Be sure to check the website listed below for specific ColdFire device errata/addendums:
http://www.mot.com/ColdFire

Trademarks

All other trademarks and registered trademarks are the property of their respective owners. "ColdFire device(s)," "ColdFire Family," and other such references as used in this manual refer to the ColdFire® Microprocessor Family.

TABLE OF CONTENTS

Title **Page Number**

Section 1
Introduction

TABLE OF CONTENTS (Continued)

Title **Page Number**

Section 2
Addressing Capabilities

TABLE OF CONTENTS (Continued)

Title | Page Number

Section 3
Instruction Set Summary

TABLE OF CONTENTS (Continued)

Section 4
Integer Instructions

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Title | Page Number

Section 5
Supervisor (Privileged) Instructions

TABLE OF CONTENTS (Continued)

Section 6
Instruction Format Summary

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Title | Page Number

Section 7
Exception Processing

TABLE OF CONTENTS (Continued)

Title — Page Number

Section 8
S-Record Output Format

Section 9
Instruction Execution Timing (5200 Series Only)

TABLE OF CONTENTS (Continued)

Title | Page Number

Appendix A
Processor Instruction Summary

Appendix B
Multiply and Accumulate (MAC) Instructions

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Title	Page Number

TABLE OF CONTENTS (Continued)

Title **Page Number**

Section 1
Introduction

Overview

Introduction

This manual contains detailed information about software instructions used by ColdFire® microprocessors.

User and Supervisor Groups

The ColdFire Family programming model consists of two register groups:

1. User
2. Supervisor

Programs executing in the **user mode** use only the registers in the user group. System software executing in the **supervisor mode** can access all registers and use the control registers in the supervisor group to perform supervisor functions. The subsections that follow briefly describe the registers in the user and supervisor models as well as the data organization in the registers.

Processor Register Description

The following paragraphs describe the processor registers in the **user and supervisor programming models**. The appropriate programming model is selected based on the privilege level (user mode or supervisor mode) of the processor as defined by the S-bit of the status register.

User Programming Model

Introduction

Figure 1-1 illustrates the **user programming model.** The model is the same as for M68000 Family microprocessors, consisting of the following registers:

- 16 general-purpose 32-bit registers (D0–D7, A0–A7)
- 32-bit program counter (PC)
- 8-bit condition code register (CCR)

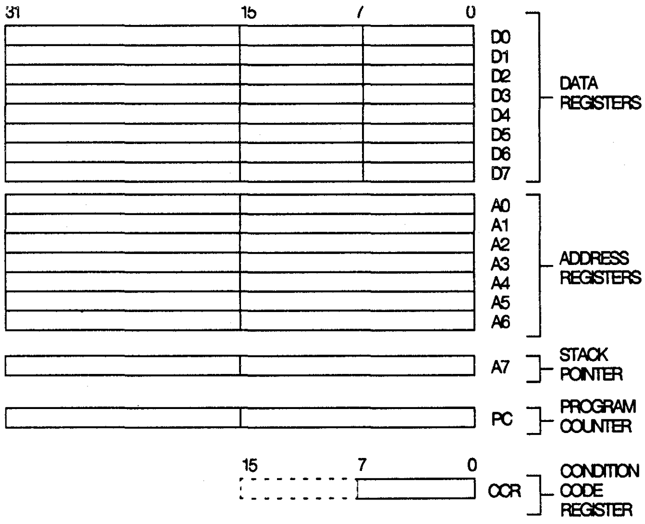
Data Registers (D0-D7)

Registers D0–D7 are used as data registers for bit (1 bit), byte (8 bit), word (16 bit) and longword (32 bit) operations and can also be used as index registers.

Address Registers (A0-A6)

These registers can be used as software stack pointers, index registers, or base address registers as well as for word and longword operations.

Figure 1-1: User Programming Model



User Programming Model, Continued

Stack Pointer (A7)

ColdFire supports a single hardware stack pointer (A7) for explicit references or implicit ones during stacking for subroutine calls and returns and exception handling. The initial value of A7 is loaded from the reset exception vector, address $0. The same register is used for both user and supervisor mode as well as word and longword operations.

A subroutine call saves the program counter (PC) on the stack and the return restores it from the stack. Both the PC and the SR are saved on the stack during the processing of exceptions and interrupts. The return from exception instruction restores the SR and PC values from the stack.

Program Counter (PC)

The PC contains the address of the currently executing instruction. 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. For some addressing modes, the PC can be used as a pointer for PC-relative operand addressing.

Condition Code Register (CCR)

The CCR is the least significant byte of the processor status register (SR), as shown below. Bits 4–0 represent indicator flags based on results generated by processor operations. Bit 4, the extend bit (X-bit), is also used as an input operand during multiprecision arithmetic computations.

4	3	2	1	0
X	N	Z	V	C

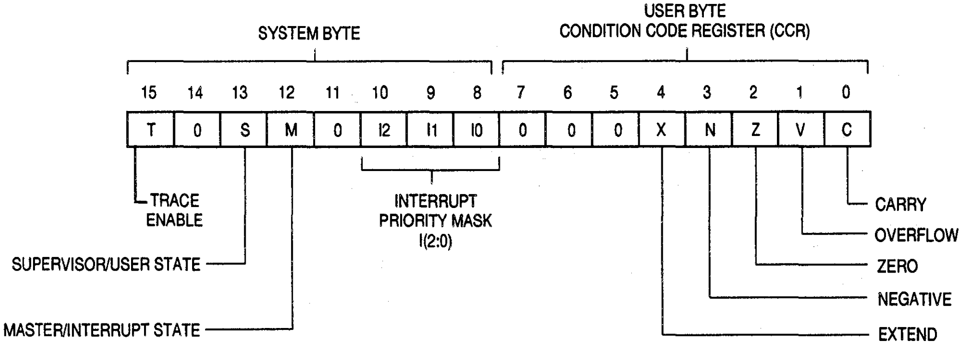
- X—extend condition code bit
- N—negative condition code bit; set if the most significant bit of the result is set; otherwise cleared
- Z—zero condition code bit; set if the result equals zero; otherwise cleared
- V—overflow condition code bit; set if an arithmetic overflow occurs implying that the result cannot be represented in the operand size; otherwise cleared
- C—carry condition code bit; set if a carryout of the operand MSB occurs for an addition, or if a borrow occurs in a subtraction; otherwise cleared; set to the value of the C-bit for arithmetic operations; otherwise not affected.

User Programming Mode, Continued

Status Register (SR)

Figure 1-2 illustrates the SR, which stores the processor status and contains the condition codes that reflect the results of a previous operation. In the supervisor mode, software can access the full SR, including the interrupt-priority mask and additional control bits. In user mode, only the lower 8 bits are accessible (CCR). These bits indicate the following states for the processor: trace mode (T), supervisor or user mode (S), and master or interrupt mode (M).

Figure 1-2: Status Register

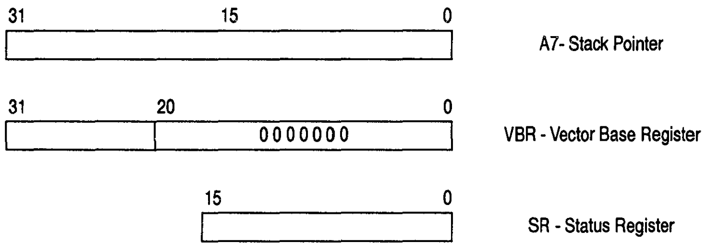


Supervisor Programming Model

Introduction

System programers use the **supervisor programming model** to implement sensitive operating system functions. The following paragraphs briefly describe the registers in the supervisor programming model. All accesses that affect the control features of ColdFire processors are in the supervisor programming model, which consists of the register available to users as well as the registers listed in Table 1-1.

Table 1-1: Supervisor Registers



Address Register 7 (A7)

ColdFire supports a single stack pointer (A7). The initial value of A7 is loaded from the reset exception vector, address offset 0. This is the same register as the stack pointer (A7) in the user programming model.

Status Register (SR)

See **User Programming Mode, Status Register**, page 1-4.

Vector Base Register (VBR)

The vector base register (VBR) contains the base address of the exception vector table in memory. The displacement of an exception vector adds to the value in this register, which accesses the vector table. The lower 20 bits of the VBR are filled with zeros.

Integer Data Format

The operand data formats are supported by the processor core, as listed in Table 1-2. Integer operands can reside in registers, memory, or instructions themselves. The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation.

Supervisor Programming Mode, Continued

Table 1-2: Integer Data Format

OPERAND DATA FORMAT	SIZE
Bit	1 Bit
Byte Integer	8 Bits
Word Integer	16 Bits
Longword Integer	32 Bits

Organization of Data in Registers

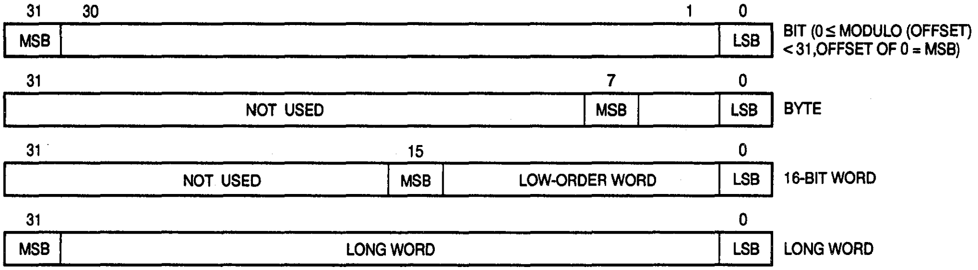
Introduction

The following paragraphs describe data organization within the data, address, and control registers.

Organization of Data Formats in Registers

Each data register is 32 bits wide. Byte and word operands occupy the lower 8- and 16-bit portions of integer data registers, respectively. Longword operands occupy the entire 32 bits of integer data registers. A data register that is either a source or destination operand only uses or changes the appropriate lower 8 or 16 bits (in byte or word operations, respectively).The address of the least significant bit (LSB) is at bit position 0 and the address of the most significant bit (MSB) is bit position 31. See Figure 1-3.

Figure 1-3: Organization of Integer Data Formats in Data Registers

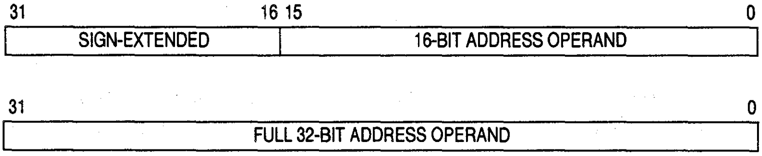


Integer Data Formats in Address Registers

Because address registers and stack pointers are 32 bits wide, address registers cannot be used for byte-size operands. When an address register is a source operand, either the low-order word or the entire longword operand is used, depending on the operation size. When an address register is the destination operand, the entire register becomes affected, despite the operation size. If the source operand is a word size, it is sign-extended to 32 bits and then used in the operation to an address-register destination. Address registers are primarily for addresses and address computation support. The instruction set explains how to add to, compare, and move the contents of address registers. Figure 1-4 illustrates the organization of addresses in address registers.

Organization of Data in Registers, Continued

Figure 1-4: Organization of Integer Data Formats in Address Registers



Undefined Bits in Control Registers

Control registers vary in size according to function. Some control registers have undefined bits reserved for future definition by Motorola. Those particular bits read as zeros and must be written as zeros for future compatibility.

SR and CCR Operations

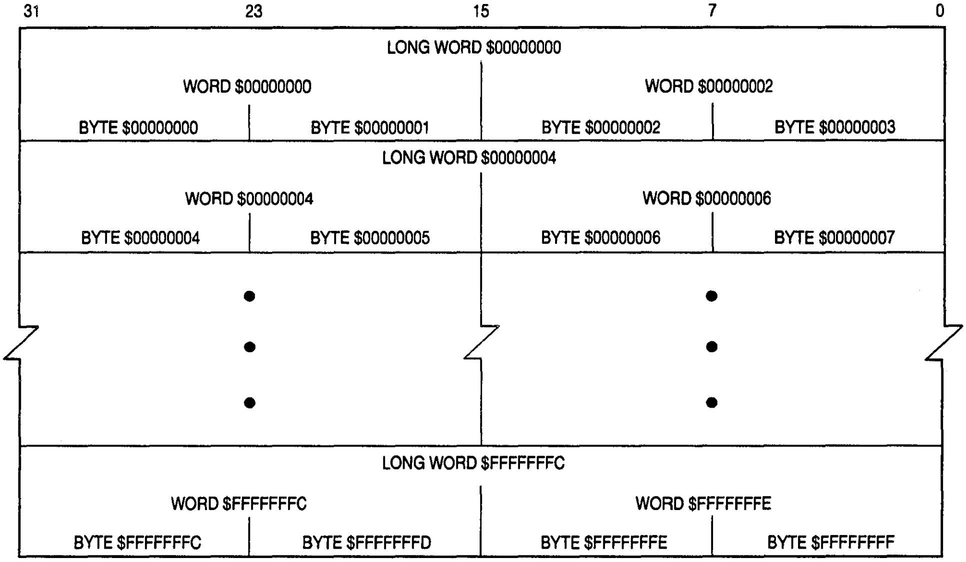
All operations to the SR and CCR are word-size operations. For all CCR operations, the upper byte is read as all zeros and is ignored when written, despite privilege mode. The write-only MOVEC instruction writes to the VBR. Other system control registers can be added depending on the implementation.

Organization of Integer Data Formats in Memory

The byte-addressable organization of memory allows lower addresses to correspond to higher order bytes. The address N of a longword data item corresponds to the address of the highest order word's MSB. The lower order word is located at address N + 2, leaving the LSB at address N + 3 (see Figure 1-5). The lowest address (nearest $00000000) is the location of the MSB, with each successive LSB located at the next address (N + 1, N + 2, etc.). The highest address (nearest $FFFFFFFF) is the location of the LSB.

Organization of Data in Registers, Continued

Figure 1-5: Memory Operand Addressing



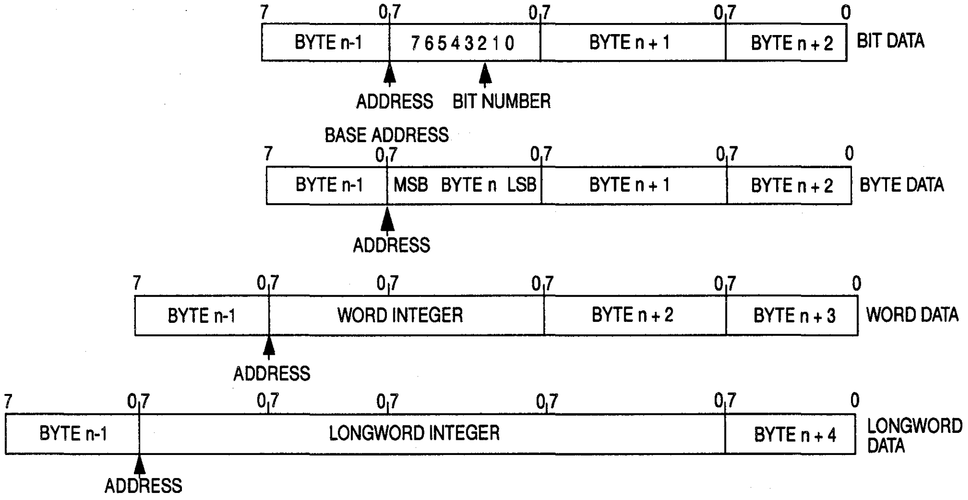
Organization of Integer Data Formats in Memory

Figure 1-6 illustrates the organization of integer data formats in memory. For RBFBRONC55 of bit data, a base address that selects one byte in memory—the base byte—specifies a bit number that selects one bit, the bit operand, in the base byte. The MSB of the byte is 7.

Continued on next page

Organization of Data in Registers, Continued

Figure 1-6: Memory Organization for Integer Operands



Section 2
Addressing Capabilities

Overview

Introduction

Most operations compute a source operand and destination operand then store the result in the destination location. Single-operand operations compute a destination operand then store the result in the destination location. External microprocessor references to memory are either program references that refer to program space, or data references that refer to data space. They access either instruction words or operands (data items) for an instruction.

Program space is the section of memory that contains the program instructions and any immediate data operands residing in the instruction stream.

Data space is the section of memory that contains the program data.

The program-counter relative addressing modes are classified as **data references.**

Instruction Format

ColdFire Family Instructions

ColdFire Family instructions consist of 1, 2, or 3 words. Figure 2-1 illustrates the general composition of an instruction. The first word of the instruction, called the **single effective address operation word**, specifies

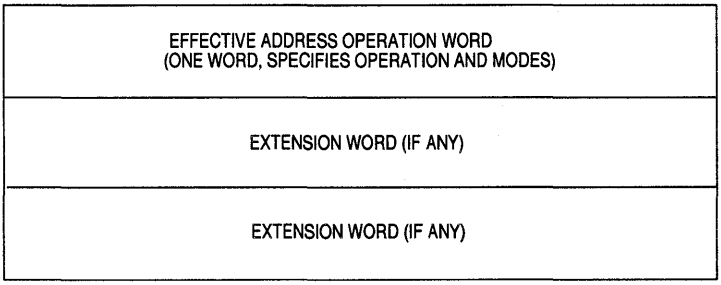
- Length of the instruction
- Effective addressing mode
- Operation to be performed

The remaining words further specify the instruction and operands. These words can be

- Immediate operands
- Extensions to the effective addressing mode specified in the simple effective address operation word
- Branch displacements
- Bit number or special register specifications
- Trap operands
- Argument counts

The ColdFire architecture instruction word length is limited to16, 32, or 48 bits.

Figure 2-1: Instruction Word General Format



Instruction-Specified Operand Location

An instruction specifies the function to be performed with an operation code and defines the location of every operand. Instructions specify an operand location by the following:

Continued on next page

Instruction Format, Continued

Instruction-Specified Operand Location (Continued)

- Register specification (the instruction's register field holds the register's number)
- Effective address (the instruction's effective address field contains addressing mode information)
- Implicit reference (the definition of the instruction implies the use of specific registers)

Instruction Word

The single effective address operation word format is the basic **instruction word** (see Figure 2-2). The encoding of the mode field 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 opcode 111. Some indexed or indirect addressing modes use a combination of the simple effective address operation word followed by an extension word. Figure 2-2 illustrates two formats used in an instruction word including the extension word format used for indexed addressing modes. Table 2-1 lists the field definitions.

Figure 2-2: Instruction Word Specification Formats

Single Effective Address Operation Word Format

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

Extension Word Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	REGISTER			W/L	SCALE		EV	DISPLACEMENT							

Continued on next page

Instruction Format, Continued

Table 2-1: Instruction Word Format Field Definitions

FIELD	DEFINITION
INSTRUCTION	
Mode	Addressing Mode
Register	General Register Number
EXTENSIONS	
D/A	Index Register Type 0 = Dn 1 = An
W/L	Word/Long-Word Index Size 0 = Address Error Exception 1 = Long Word
Scale	Scale Factor 00 = 1 01 = 2 10 = 4 11 = Address Error Exception
EV	Extension Word Valid 0 = Extension Word Valid 1 = Address Error Exception
Displacement	8 bit displacement (sign extended to 32 bits)

Effective Addressing Modes

Defining Operand Locations

Besides the operation code that specifies the function to be performed, an instruction defines the location of every operand for the function in 1 of 3 ways:

1. A register field within an instruction can specify the register to be used.
2. An instruction's effective address field can contain addressing mode information.
3. The instruction's definition can imply the use of a specific register.

Other fields within the instruction specify whether the register selected is an address or data register and how the register is to be used.

Instruction Addressing Mode

An instruction's addressing mode specifies

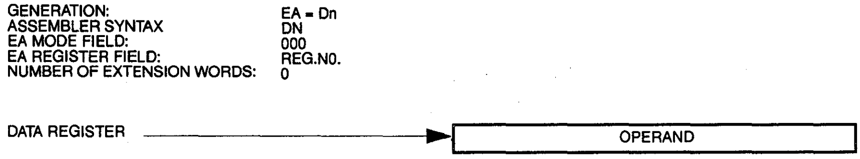
- The value of an operand
- A register that contains the operand, or
- How to derive the effective address of an operand in memory

Each addressing mode has an assembler syntax. Some instructions imply the addressing mode for an operand. These instructions include the appropriate fields for operands that use only one addressing mode.

Data Register Direct Mode

In the data register direct mode, the effective address field specifies the data register containing the operand (see Figure 2-3).

Figure 2-3: Data Register Direct Mode



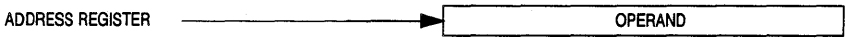
Effective Addressing Modes, Continued

Address Register Direct Mode

In the address register direct mode, the effective address field specifies the address register containing the operand (see Figure 2-4).

Figure 2-4: Address Register Direct Mode

GENERATION: EA = An
ASSEMBLER SYNTAX: An
EA MODE FIELD: 001
EA REGISTER FIELD: REG. NO.
NUMBER OF EXTENSION WORDS: 0

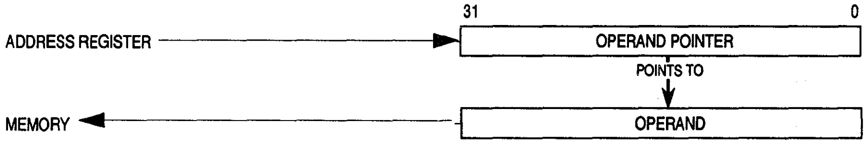


Address Register Indirect Mode

In the address register indirect mode, the operand is in memory. The effective address field specifies the address register containing the address of the operand in memory (see Figure 2-5).

Figure 2-5: Address Register Indirect Mode

GENERATION: EA = (An)
ASSEMBLER SYNTAX: (An)
EA MODE FIELD: 010
EA REGISTER FIELD: REG. NO.
NUMBER OF EXTENSION WORDS: 0



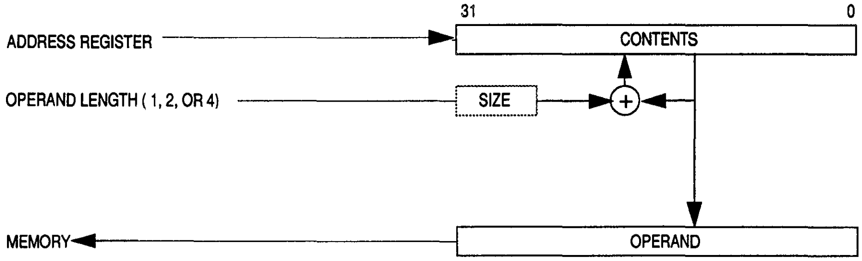
Address Register Indirect with Postincrement Mode

In the address register indirect with postincrement mode, the operand is in memory. The effective address field specifies the address register containing the address of the operand in memory. After the operand address is used, it is incremented by one, two, or four, depending on the size of the operand (i.e., byte, word, or longword, respectively). Note that the stack pointer (A7) is treated the same as other address registers (see Figure 2-6).

Effective Addressing Modes, Continued

Figure 2-6: Address Register Indirect with Postincrement Mode

GENERATION: EA = (An) + SIZE
ASSEMBLER SYNTAX: (An) +
EA MODE FIELD: 011
EA REGISTER FIELD: REG. NO.
NUMBER OF EXTENSION WORDS: 0

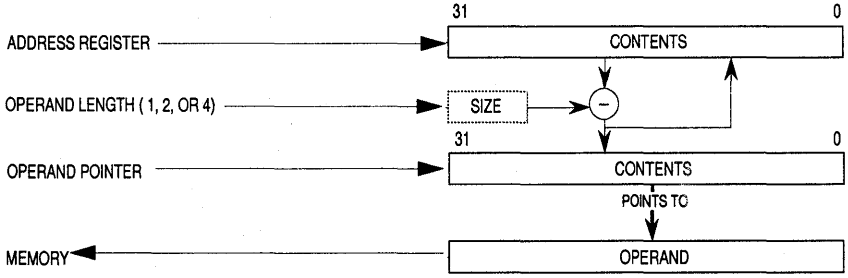


Address Register Indirect with Predecrement Mode

In the address register indirect with predecrement mode, the operand is in memory. The effective address field specifies the address register containing the address of the operand in memory. Before the operand address is used, it is decremented by one, two, or four depending on the operand size (i.e., byte, word, or longword, respectively). Note that the stack pointer (A7) is treated just like the other address registers (see Figure 2-7).

Figure 2-7: Address Register Indirect with Predecrement Mode

GENERATION: EA = (An)-SIZE
ASSEMBLER SYNTAX: - (An)
EA MODE FIELD: 100
EA REGISTER FIELD: REG. NO.
NUMBER OF EXTENSION WORDS: 0



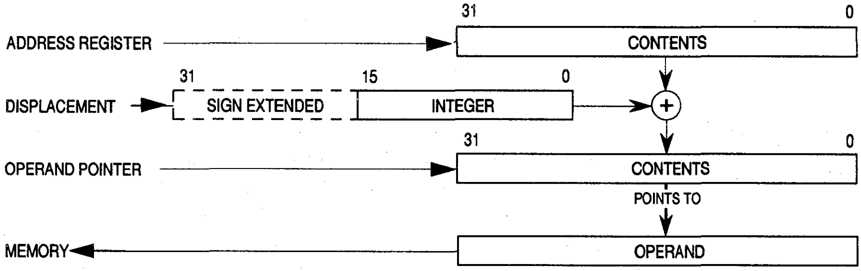
Effective Addressing Modes, Continued

Address Register Indirect with Displacement Mode

In the address register indirect with displacement mode, the operand is in memory. The operand address in memory consists of the sum of the address in the address register, which the effective address specifies, and the sign-extended 16-bit displacement integer in the extension word. Displacements are always sign-extended to 32 bits prior to being used in effective address calculations (see Figure 2-8).

Figure 2-8: Address Register Indirect with Displacement Mode

GENERATION:	EA = (An) + d_{16}
ASSEMBLER SYNTAX:	(d_{16},An)
EA MODE FIELD:	101
EA REGISTER FIELD:	REG. NO.
NUMBER OF EXTENSION WORDS:	1



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

This addressing mode requires one extension word that contains an 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 operand address is the sum of the

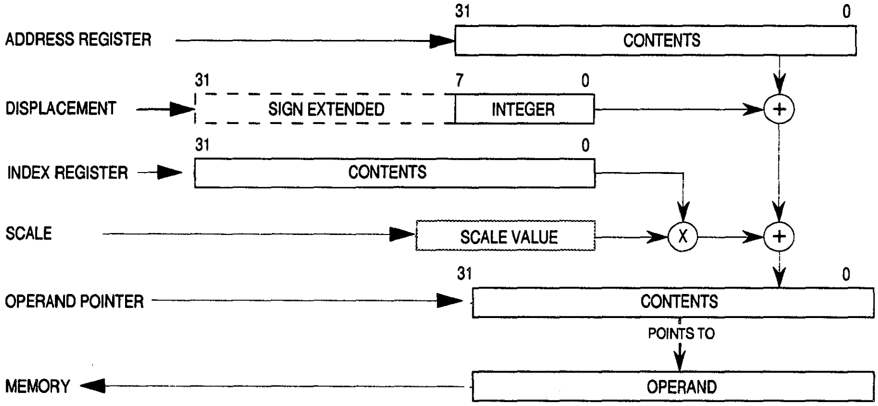
- Address register contents
- Sign-extended displacement value in the extension word's low-order 8 bits
- Index register's contents (possibly scaled)

You must specify the address register, the displacement, and the index register in this mode (see Figure 2-9).

Effective Addressing Modes, Continued

Figure 2-9: Address Register Indirect with Index (8-Bit Displacement) Mode

GENERATION: EA=(An) + (Xi) + d_8
ASSEMBLER SYNTAX: (d_8, An, Xi, + scale)
EA MODE FIELD: 110
EA REGISTER FIELD: REG. NO.
NUMBER OF EXTENSION WORDS: 1



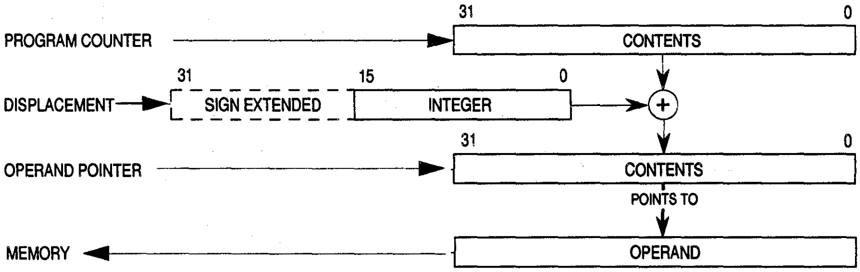
Program Counter Indirect with Displacement Mode

In this mode, the operand is in memory. The address of the operand is the sum of the address in the program counter (PC) and the sign-extended 16-bit displacement integer in the extension word. The value in the PC is the address of the extension word. This is a data reference allowed only for operand reads (see Figure 2-10).

Effective Addressing Modes, Continued

Figure 2-10: Program Counter Indirect with Displacement Mode

GENERATION: EA = (PC) + d_{16}
ASSEMBLER SYNTAX: (d_{16},PC)
EA MODE FIELD: 111
EA REGISTER FIELD: 010
NUMBER OF EXTENSION WORDS: 1



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

This mode is similar to the mode described in **Address Register Indirect with Index (8-Bit Displacement) Mode**, except the PC is the base register. The operand is in memory.

The operand address is the sum of the

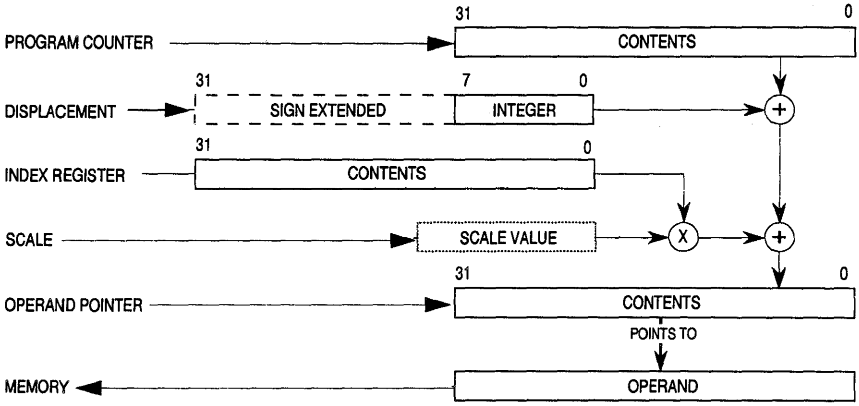
- Address in the PC
- Sign-extended displacement integer in the extension word's lower 8 bits
- Scaled index register

The value in the PC is the address of the extension word. This is a data reference allowed only for operand reads. You must include the displacement, the PC, and the index register when specifying this addressing mode (see Figure 2-11).

Effective Addressing Modes, Continued

Figure 2-11: Program Counter Indirect with Index (8-Bit Displacement) Model

GENERATION:	EA = (PC) + (XI) + d_8
ASSEMBLER SYNTAX:	(d_8, PC, Xi, + scale)
EA MODE FIELD:	111
EA REGISTER FIELD:	011
NUMBER OF EXTENSION WORDS:	1

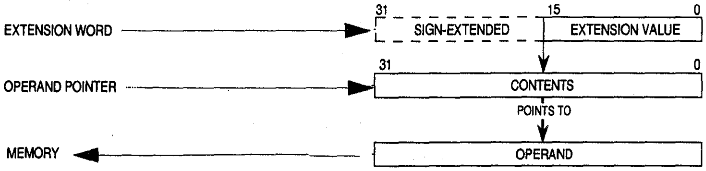


Absolute Short-Addressing Mode

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

Figure 2-12: Absolute Short Addressing Mode

GENERATION:	EA GIVEN
ASSEMBLER SYNTAX:	(xxx).W
EA MODE FIELD:	111
EA REGISTER FIELD:	000
NUMBER OF EXTENSION WORDS:	1



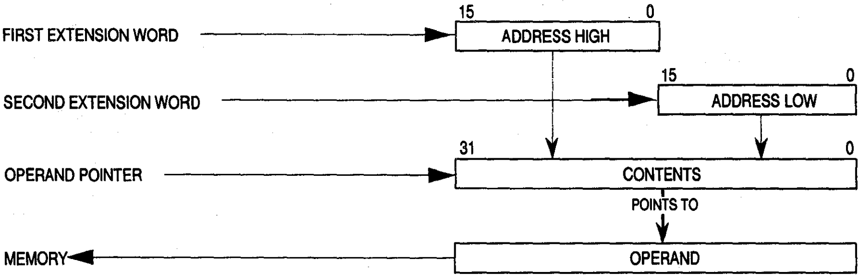
Effective Addressing Modes, Continued

Absolute Long Addressing Mode

In this addressing mode, the operand is in memory, and the operand address occupies the two extension words following the instruction word in memory. The first extension word contains the high-order part of the address; the second contains the low-order part of the address (see Figure 2-13).

Figure 2-13: Absolute Long Addressing Mode

GENERATION:	EA GIVEN
ASSEMBLER SYNTAX:	(xxx).L
EA MODE FIELD:	111
EA REGISTER FIELD:	001
NUMBER OF EXTENSION WORDS:	2



Immediate Data

In this addressing mode, the operand is in 1 or 2 extension words. Table 2-2 lists the location of the operand within the instruction word format.

Table 2-2: Immediate Operand Location

OPERATION LENGTH	LOCATION
Byte	Low-order byte of the extension word
Word	The entire extension word
Long Word	High-order word of the operand is in the first extension word; the low-order word is in the second extension word

Immediate Data Addressing Mode

GENERATION:	OPERAND GIVEN
ASSEMBLER SYNTAX:	#<xxx>
EA MODE FIELD:	111
EA REGISTER FIELD:	100
NUMBER OF EXTENSION WORDS:	1,2,4, OR 6, EXCEPT FOR PACKED DECIMAL REAL OPERANDS

Effective Addressing Modes, Continued

Effective Addressing Mode Summary

Effective addressing modes are grouped according to the mode use. Data-addressing modes refer to data operands. Memory-addressing modes refer to memory operands. Alterable addressing modes refer to alterable (writable) operands. Control-addressing modes refer to memory operands without an associated size.

Alterable Memory and Data Alterable

These categories sometimes combine to form new categories that are more restrictive: **alterable memory** (addressing modes that are both alterable and memory addresses), and **data alterable** (addressing modes that are both alterable and data). Table 2-3 lists a summary of effective addressing modes and their categories.

Table 2-3: Effective Addressing Modes and Categories

ADDRESSING MODES	SYNTAX	MODE FIELD	REG. FIELD	DATA	MEMORY	CONTROL	ALTERABLE
Register Direct							
Data	Dn	000	reg. no. "n"	X	—	—	X
Address	An	001	reg. no. "n"	—	—	—	X
Register Indirect							
Address	(An)	010	reg. no. "n"	X	X	X	X
Address with Postincrement	(An)+	011	reg. no. "n"	X	X	—	X
Address with Predecrement	–(An)	100	reg. no. "n"	X	X	—	X
Address with Displacement	(d_{16},An)	101	reg. no. "n"	X	X	X	X
Address Register Indirect with Index							
8-Bit Displacement	(d_8,An,Xi)	110	reg. no. "n"	X	X	X	X
Program Counter Indirect with Displacement	(d_{16},PC)	111	010	X	X	X	—
Program Counter Indirect with Index							
8-Bit Displacement	(d_8,PC,Xi)	111	011	X	X	X	—
Absolute Data Addressing							
Short	(xxx).W	111	000	X	X	X	—
Long	(xxx).L	111	000	X	X	X	—
Immediate	#<xxx>	111	100	X	X	—	—

Stack

Overview

Address register (A7) stacks exception frames, subroutine calls and returns, temporary variable storage, parameter passing, and is affected by instructions such as the LINK, UNLK, RTE, RTS, and PEA. To maximize performance, A7 must be longword-aligned at all times. Therefore, when modifying A7, be sure to do so in multiples of 4 to maintain alignment. To futher ensure alignment of A7 during exception handling, the ColdFire architecture implements a self-aligning stack when processing exceptions.

Implementing Other Stacks Using Other Address Registers

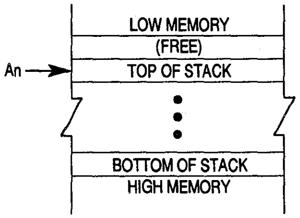
You can employ other address registers to implement other stacks using the address register indirect with postincrement and predecrement addressing modes. With an address register, you can implement a stack that fills either from high memory to low memory, or vice-versa. Regarding the following important considerations, you should

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

Implementing Stack Growth from High Memory to Low Memory

To implement stack growth from high memory to low memory, use -(An) to push data on the stack and (An) + to pop data from the stack. For this type of stack, after either a push or a pop operation, the address register points to the top item on the stack (see Figure 2-14).

Figure 2-14: Stack Growth from High Memory to Low Memory

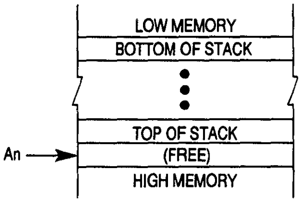


Stack, Continued

Implementing Stack Growth from Low Memory to High Memory

To implement stack growth from low memory to high memory, use (An) + to push data on the stack and -(An) to pop data from the stack. After either a push or pop operation, the address register points to the next available space on the stack (see Figure 2-15).

Figure 2-15: Stack Growth from Low Memory to High Memory



Section 3

Instruction Set Summary

Overview

Introduction

This section briefly describes the ColdFire Family instruction set using Motorola's assembly language syntax and notation. It includes instruction set details such as notation and format, selected instruction examples, and an integer condition code discussion.

The section concludes with a discussion of conditional test definitions, an explanation of the operation table, and postprocessing.

Instruction Summary

Tools for Specific Operations

Instructions form a set of tools that perform the following types of operations:

- Data Movement
- Integer Arithmetic
- Logical Operations
- Bit Manipulation
- Program Control
- System Control
- Shift Operations

The following paragraphs describe in detail the instruction for each type of operation. Table 3-1 lists the notations used throughout this manual. In the operand syntax statements of the instruction definitions, the operand on the right is the destination operand.

Instruction Summary, Continued

Table 3-1: Notational Conventions

SINGLE- AND DOUBLE OPERAND OPERATIONS	
+	Arithmetic addition or postincrement indicator
–	Arithmetic subtraction or predecrement indicator
¥	Arithmetic multiplication
Π	Arithmetic division or conjunction symbol
~	Invert; operand is logically complemented
L	Logical AND
V	Logical OR
≈	Logical exclusive OR
→	Source operand is moved to destination operand
←→	Two operands are exchanged
<op>	Any double-operand operation
<operand>tested	Operand is compared to zero and the condition codes are set appropriately
sign-extended	All bits of the upper portion are made equal to the high-order bit of the lower portion
OTHER OPERATIONS	
TRAP	SP - 4 Æ SP; PC Æ (SP); SP - 2 Æ SP; SR Æ (SP); SP - 2 Æ SP; FORMAT Æ (SP); (Vector) Æ PC
STOP	Enter the stopped state, waiting for interrupts
If <condition> then <operations> else <operations>	Test the condition. If true, the operations after "then"are performed. If the condition is false and the optional "else"clause is present, the operations after "else"are performed. If the condition is false and else is omitted, the instruction performs no operation. Refer to the Bcc instruction description as an example.
REGISTER SPECIFICATIONS	
An	Any Address Register n (example: A3 is address register 3)
Ay, Ax	Source and destination address registers, respectively
Dn	Any Data Register n (example: D5 is data register 5)
Dy, Dx	Source and destination data registers, respectively
MRn	Any Memory Register n
Rn	Any Address or Data Register
Rc	Any con trol register
Ry, Rx	Any source and destination registers, respectively
Xi	Index Register

Continued on next page

Instruction Summary, Continued

Table 3-1: Notational Conventions (Continued)

DATA FORMAT AND TYPE	
<fmt>	Operand Data Format: Byte (B), Word (W), Long (L)
B, W, L	Specifies an integer data type of byte, word, or longword size
SUBFIELDS AND QUALIFIERS	
#<xxx> or #<data>	Immediate data following the instruction word(s).
( )	Identifies an indirect address in a register, contents of memory location
d_n	Displacement Value, n Bits Wide (example: d_{16} is a 16-bit displacement)
LSB	Least Significant Bit
LSW	Least Significant Word
MSB	Most Significant Bit
MSW	Most Significant Word
REGISTER NAMES	
CCR	Condition Code Register (lower byte of Status Register)
IC, DC, IC/DC	Instruction, Data, or Both Caches
PC	Program Counter
SR	Status Register
VBR	Vector Base Register
REGISTER CODES	
*	General Case
C	Carry Bit in CCR
cc	Condition Codes from CCR (c, n, v, x, z)
	c = carry bit is set
	n = negative number
	v = overflow
	x = sign extended
	z = zero
N	Negative Bit in CCR
U	Undefined, Reserved for Motorola Use
V	Overflow Bit in CCR
X	Extend Bit in CCR
Z	Zero Bit in CCR
—	Not Affected or Applicable
MISCELLANEOUS	
<ea>y,<ea>x	Any source or destination effective address, respectively
<label>	Assembly Program Label
<list>	List of registers, for example D3–D0
m	Bit m of an Operand
m–n	Bits m through n of Operand

Instruction Summary, Continued

Data Movement Instructions

The MOVE instruction with its associated addressing mode is the basic means of transferring and storing addresses and data.

MOVE INSTRUCTIONS TRANSFER...	FROM...	TO...
byte, word, longword operands...	Memory	Memory
	Memory	Register
	Register	Memory
	Register	Register

MOVEA instructions transfer word and longword operands and ensure that only valid address manipulations are executed. In addition to the general MOVE instructions, there are several special data movement instructions: MOVEM, MOVEQ, LEA, PEA, LINK, and UNLK. See Table 3-2 for details.

Table 3-2: Data Movement Operation Format

INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
LEA	<ea>y,Ax	32	<ea> → An
LINK	Ax,#data	32	SP–4 → SP; Ax → (SP): SP → Ax; SP + d16→ SP
MOVE MOVEA	<ea>y,<ea>x <ea>y,Ax	8, 16, 32 16, 32 → 32	Source → Destination
MOVEM	list,<ea>x <ea>y,list	32 32	Listed Registers → Destination Source → Listed Registers
MOVEQ	#<data>,Dx	8 → 32	Sign-Extended Immediate Data → Destination
PEA	<ea>y	32	SP – 4 → SP; <ea>y → (SP)
UNLK	Ax	32	Ax → SP; (SP) → Ax; SP + 4 → SP

Integer Arithmetic Instructions

The integer arithmetic operations include 6 basic operations: ADD, SUB, MUL, CMP, CLR, and NEG. Most instructions support only longword operands. The CLR instruction applies to all sizes of data operands. Signed and unsigned MUL instructions include:

- Word multiply to produce a longword product
- Longword multiply to produce a longword product

Continued on next page

Instruction Summary, Continued

Integer Arithmetic Instructions (Continued)

A set of extended instructions provides multiprecision and mixed-size arithmetic: ADDX, SUBX, EXT, and NEGX. Refer to Table 3-3 for a summary of the integer arithmetic operations. In Table 3-3, X refers to the "extend" bit in the CCR.

Table 3-3: Integer Arithmetic Operations Format

INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
ADD	Dy,<ea>,x	32	Source + Destination → Destination
	<ea>y,Dx	32	
ADDA	<ea>y,Ax	32	
ADDI	#<data>,Dx	32	Immediate Data + Destination → Destination
ADDQ	#<data>,<ea>x	32	
ADDX	Dy,Dx	32	Source + Destination + X → Destination
CLR	<ea>x	8, 16, 32	0 → Destination
CMP	<ea>y,Dx	32	Destination – Source
CMPA	<ea>y,Ax	32	
CMPI	#<data>, Dx	32	Destination – Immediate Data
EXT	Dx	8 → 16	Sign-Extended Destination → Destination
	Dx	16 → 32	
EXTB	Dx	8 → 32	
MULS/MULU	<ea>y,Dx	16 x 16 → 32	Source x Destination → Destination
	<ea>y,Dl	32 x 32 → 32	(Signed or Unsigned)
NEG	<ea>x	32	0 – Destination → Destination
NEGX	<ea>x	32	0 – Destination – X → Destination
SUB	Dy,<ea>x	32	Destination - Source → Destination
	<ea>y,Dx	32	
SUBA	<ea>,Ax	32	
SUBI	#<data>, Dx	32	Destination – Immediate Data → Destination
SUBQ	#<data>,<ea>x	32	
SUBX	Dy,Dx	32	Destination – Source – X → Destination

Logic Instructions

The instructions AND, OR, EOR, and NOT perform logic operations with all sizes of integer data operands. A similar set of immediate instructions (ANDI, ORI, and EORI) provides these logic operations with all sizes of immediate data. Table 3-4 summarizes the logic operations.

Instruction Summary, Continued

Table 3-4: Logic Operation Format

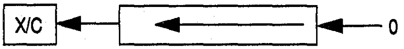
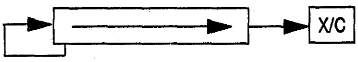
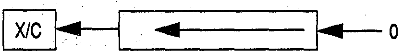
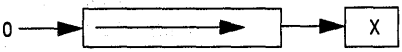
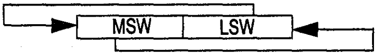
INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
AND	Dy, <ea>x <ea>y, Dx	32 32	Source L Destination → Destination
ANDI	#<data>, Dx	32	Immediate Data L Destination → Destination
EOR	Dy,<ea>x	32	Source ≈ Destination → Destination
EORI	#<data>, Dx	32	Immediate Data ≈ Destination → Destination
NOT	<ea>x	32	~ Destination → Destination
OR	Dy, <ea>x <ea>y, Dx	32	Source V Destination → Destination
ORI	#<data>, Dx	32	Immediate Data V Destination → Destination

Shift Instruction

The ASR, ASL, LSR, and LSL instructions provide shift operations in both directions. All shift operations can be performed on longword-sized data registers.The shift count can be specified in the instruction operation word (to shift from 1 – 8 places) or in a register (modulo 64 shift count).

The SWAP instruction exchanges the 16-bit halves of a register. Table 3-5 is a summary of the shift operations. In Table 3-5, C and X refer to the carry bit and extend bit in the CCR.

Table 3-5: Shift Operation Format

INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
ASL	Dx, Dy # <data>, Dx	32 32	
ASR	Dx, Dy #<data>, Dx	32 32	
LSL	Dx, Dy #<data>, Dx	32 32	
LSR	Dx, Dy #<data>, Dx	32 32	
SWAP	Dx	16	

NOTE: X indicates the extend bit and C the carry bit in the CCR.

Instruction Summary, Continued

Bit Manipulation Instructions

BTST, BSET, BCLR, and BCHG are bit manipulation instructions. All bit manipulation operations can be performed on either registers or memory. The bit number is specified either as immediate data or in the contents of a data register. Register operands are 32 bits long, and memory operands are 8 bits long. Table 3-6 summarizes bit manipulation operations; Z refers to the zero bit of the CCR.

Table 3-6: Bit Manipulation Operation Format

INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
BCHG	Dy,<ea>x #<data>,<ea>x	8, 32 8, 32	~ (<Bit Number> of Destination) → Z, Bit of Destination
BCLR	Dy,<ea>x #<data>,<ea>x	8, 32 8, 32	~ (<Bit Number> of Destination) → Z; 0 → Bit of Destination
BSET	Dy,<ea>x #<data>,<ea>x	8, 32 8, 32	~ (<Bit Number> of Destination) → Z; 1 → Bit of Destination
BTST	Dy,<ea>x #<data>,<ea>x	8, 32 8, 32	~ (<Bit Number> of Destination) → Z

Program Control Instructions

A set of subroutine call-and-return instructions and conditional and unconditional branch instructions perform program-control operations. Also included are test operand instructions (TST), which set the condition codes for use by other program- and system-control instructions. NOP forces synchronization of the internal pipelines. Table 3-7 summarizes these instructions.

Instruction Summary, Continued

Table 3-7: Program Control Operation Format

INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
			CONDITIONAL
Bcc	<label>	8, 16	If Condition True, Then PC + d_n → PC
Scc	Dx	8	If Condition True, Then 1's → Destination; Else 0's → Destination
			UNCONDITIONAL
BRA	<label>	8, 16	PC + d_n → PC
BSR	<label>	8, 16	SP – 4 → SP;Next PC → (SP); PC + d_n → PC
JMP	<ea>y	none	<ea>y → PC
JSR	<ea>y	none	SP – 4 → SP;Next PC → (SP); <ea>y → PC
NOP	none	none	PC + 2 → PC (Pipelines Synchronized)
TRAPF	none	none	PC + 2→ PC
TRAPF	# <data>	16	PC + 4 → PC
		32	PC + 6→ PC
			RETURNS
RTS	none	none	(SP) → PC; SP + 4 → SP
			TEST OPERAND
TST	<ea>y	8, 16, 32	Set Condition Codes

Note:

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

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

*Not applicable to the Bcc instructions

System Control Instructions

Introduction

Privileged and trap instructions as well as instructions that use or modify the CCR provide system control operations. Table 3-8 summarizes these instructions. See **Integer Unit Condition Code Computation** for more details on condition codes.

System Control Instructions, Continued

Table 3-8: System Control Operation Format

INSTRUCTION	OPERAND SYNTAX	OPERAND SIZE	OPERATION
PRIVILEGED			
MOVE to SR	Dy, SR, #<data>, SR	16	Source → SR
MOVE from SR	Dx	16	SR → Destination
MOVEC	Rn,Rc	32	Ry → Rc
RTE	none	none	2 (SP) → SR; 4 (SP) → PC; SP + 8 →SP Adjust Stack According to Format
STOP	#<data>	16	Immediate Data → SR; STOP
HALT	none	none	Halt the processor
WDEBUG	<ea>y	64	<ea>y → DEBUG; <ea>y+4 →DEBUG
PULSE	none	none	Generate unique PST value
WDDATA	<ea>y	8, 16, 32	(<ea>y) → DDATA port
TRAP GENERATING			
Illegal, Trap	none	none	SP – 4 → SP; PC → (SP); SP – 2 → SP; SR → (SP) SP – 2 → SP; Format/Vector → (SP) (Vector)→ PC
CONDITION CODE REGISTER			
MOVE to CCR	Dy, CCR #<data>,CCR	16	Source → CCR
MOVE from CCR	Dx	16	CCR → Destination

Integer Unit Condition Code Computation

Introduction

Many integer instructions affect the CCR to indicate the instruction's results. Program and system control instructions also use certain combinations of these bits to control program and system flow. The condition codes meet consistency criteria across instructions, uses, and instances. They also meet the criteria of meaningful results, where no change occurs unless it provides useful information.

Continued on next page

Integer Unit Condition Code Computation

Introduction (Continued)

Table 3-9 lists the integer condition code computations for instructions and Table 3-10 lists the condition names, encodings, and tests for the conditional branch and set instructions. The test associated with each condition is a logical formula using the current states of the condition codes. If this formula evaluates to one, the condition is true. If the formula evaluates to zero, the condition is false. For example, the T condition is always true, and the EQ condition is true only if the Z-bit condition code is currently true.

Table 3-9: Integer Unit Condition Code Computations

OPERATIONS	X	N	Z	V	C	SPECIAL DEFINITION
ADD, ADDI, ADDQ	*	*	*	?	?	$V = Sm\ L\ Dm\ L\ \overline{Rm}\ V\ \overline{Sm}\ L\ \overline{Dm}\ L\ Rm$ $C = Sm\ L\ Dm\ V\ \overline{Rm}\ L\ Dm\ V\ Sm\ L\ \overline{Rm}$
ADDX	*	*	?	?	?	$V = Sm\ L\ Dm\ L\ \overline{Rm}\ V\ \overline{Sm}\ L\ \overline{Dm}\ L\ Rm$ $C = Sm\ L\ Dm\ V\ \overline{Rm}\ L\ Dm\ V\ Sm\ L\ \overline{Rm}$ $Z = Z\ L\ \overline{Rm}\ L\ ...L\ \overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT, EXTB, NOT, TST	—	*	*	0	0	
SUB, SUBI, SUBQ	*	*	*	?	?	$V = \overline{Sm}\ L\ Dm\ L\ \overline{Rm}\ V\ Sm\ L\ \overline{Dm}\ L\ Rm$ $C = Sm\ L\ \overline{Dm}\ V\ Rm\ L\ \overline{Dm}\ V\ Sm\ L\ Rm$
SUBX	*	*	?	?	?	$V = \overline{Sm}\ L\ Dm\ L\ \overline{Rm}\ V\ Sm\ L\ \overline{Dm}\ L\ Rm$ $C = Sm\ L\ \overline{Dm}\ V\ Rm\ L\ \overline{Dm}\ V\ Sm\ L\ Rm$ $Z = Z\ L\ \overline{Rm}\ L\ ...L\ \overline{R0}$
CMP, CMPA, CMPI	—	*	*	?	?	$V = \overline{Sm}\ L\ Dm\ L\ \overline{Rm}\ V\ Sm\ L\ \overline{Dm}\ L\ Rm$ $C = Sm\ L\ \overline{Dm}\ V\ Rm\ L\ \overline{Dm}\ V\ Sm\ L\ Rm$
MULS, MULU	—	*	*	0	0	
NEG	*	*	*	?	?	$V = Dm\ L\ Rm$ $C = Dm\ V\ Rm$
NEGX	*	*	?	?	?	$V = Dm\ L\ Rm$ $C = Dm\ V\ Rm$ $Z = Z\ L\ \overline{Rm}\ L\ ...L\ \overline{R0}$
BTST, BCHG, BSET, BCLR	—	—	?	—	—	$Z = \overline{Dn}$
ASL	*	*	*	0	?	$C = \overline{Dm-}\ \overline{r+1}$
ASL (r = 0)	—	*	*	0	0	
LSL	*	*	*	0	?	$C = Dm - r + 1$
LSR (r = 0)	—	*	*	0	0	
ASR, LSR	*	*	*	0	?	$C = Dr - 1$
ASR, LSR (r = 0)	—	*	*	0	0	

Integer Unit Condition Code Computation, Continued

Notes

? = Other—See Special Definition
N = Result Operand (MSB)
Z = $\overline{Rm}$ L...L $\overline{R0}$
Sm = Source Operand (MSB)
Dm = Destination Operand (MSB)

Rm = Result Operand (MSB)
$\overline{Rm}$ = Not Result Operand (MSB)
R = Register Tested
r = Shift Count

Table 3-10: Conditional Tests

MNEMONIC	CONDITION	ENCODING	TEST
T*	True	0000	1
F*	False	0001	0
HI	High	0010	$\overline{C}$ V $\overline{Z}$
LS	Low or Same	0011	C V Z
CC(HI)	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 L V V $\overline{N}$ L $\overline{V}$
LT	Less Than	1101	N L $\overline{V}$ V $\overline{N}$ L V
GT	Greater Than	1110	N L V L $\overline{Z}$ V $\overline{N}$ L $\overline{V}$ L $\overline{Z}$
LE	Less or Equal	1111	Z V N L $\overline{V}$ V $\overline{N}$ L V

Notes

$\overline{N}$ = Logical Not N
$\overline{V}$ = Logical Not V
$\overline{Z}$ = Logical Not Z
**Not available for the Bcc instruction.*

Section 4
Integer Instructions

Overview

Introduction

This section describes the integer instructions for the ColdFire Family. A detailed discussion of each instruction description is arranged in alphabetical order by instruction mnemonic.

ADD (Add)

Operation:	Source + Destination → Destination
Assembler Syntax:	ADD < ea >y ,Dx; ADD Dy, < ea >x
Attributes:	Size = 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 is specified as a longword. 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

ADD, Continued

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 8 data registers

Opmode field:

LONG	OPERATION
010	< ea >y + Dx
110	Dy + < ea >x –> < ea >x

Effective Address field—determines addressing mode

a. If the location <ea>x specified is a source operand, use addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy	(xxx).W	111	000
Ay	001	reg. number:Ay	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	111	100
(Ay) +	011	reg. number:Ay			
– (Ay)	100	reg. number:Ay			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xi)	110	reg. number:Ay	(d_8,PC,Xi)	111	011

b. If the <ea>x location specified is a destination operand, use only memory alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx	—	—	(xxx).W	111	000
Ax	—	—	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

ADD, Continue

Note

The Dx mode is used when the destination is a data register; the destination < ea > mode is invalid for a data register.

ADDA is used when the destination is an address register. ADDI and ADDQ are used when the source is immediate data.

ADDA (Add Address)

Operation: Source + Destination → Destination

Assembler Syntax: ADDA <ea>y , Ax

Attributes: Size = Long

Description

Adds the source operand to the destination address register and stores result in the address register. Operation size is specified as a longword.

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	1	REGISTER			1	1	1	EFFECTIVE ADDRESS MODE			REGISTER		

Instruction Fields

Register field—specifies the destination address register, Ax.

Effective Address field—specifies the source operand; use addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy	(xxx).W	111	000
Ay	001	reg. number:Ay	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	111	100
(Ay) +	011	reg. number:Ay			
– (Ay)	100	reg. number:Ay			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xi)	110	reg. number:Ay	(d_8,PC,Xi)	111	011

ADD I (Add Immediate)

Operation: Immediate Data + Destination → Destination
Assembler
Syntax: ADDI # < data > , Dx
Attributes: Size = Long

Description

Adds the immediate data to the destination operand and stores the result in the destination location. The size of the operation is specified as longword.

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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Instruction Fields

Register field—specifies the destination data register, Dx

ADDQ (Add Quick)

Operation: Immediate Data + Destination → Destination
Assembler
Syntax: ADDQ # < data > , < ea >x
Attributes: Size = Long

ADDQ, Continued

Description

Adds an immediate value of 1 to 8 to the operand at the destination location. The size of the operation is specified as longword. If the destination is an address register, the condition codes are not affected.

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	1	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields

Data field—3 bits of immediate data representing 8 values (0 – 7), with the immediate value 0 representing a value of 8

Effective Address field—specifies the destination location; use only those alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dx	000	reg. number:Dx
Ax	001	reg. number:Ax
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

ADDX (Add Extended)

Operation:	Source + Destination + X → Destination
Assembler Syntax:	ADDX Dy,Dx
Attributes:	Size = Long

Description

Adds the source operand and the extend bit to the destination operand and stores the result in the destination location. The operands can be addressed from data register to data register where the data registers specified in the instruction contain the operands.

The size of the operation is specified as a longword.

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

Instruction Format

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

Instruction Fields

Register Dx field—specifies the destination data register

Register Dy field—specifies the source data register

AND (AND Logical)

Operation:	Source + Destination → Destination
Assembler Syntax:	AND Dy, < ea >x; AND < ea >y ,Dx
Attributes:	Size = Long

AND (AND Logical), Continued

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 is specified as a longword. Address register contents cannot 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

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="3" rowspan="2">REGISTER</td><td colspan="3" rowspan="2">OPMODE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Fields

Register field—Specifies any of the 8 data registers.

Opmode field:

LONG	OPERATION
010	< ea >y I Dx — Dx
110	Dy I< ea>x → < ea>x

Effective Address field—determines addressing mode.

a. If the location specified is a source operand, use only those data addressing modes listed in the following table:

Continued on next page

AND (AND Logical), Continued

Instruction Fields (Continued)

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
- (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xy)	111	011

b. If the location specified is a destination operand, use only those memory alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
- (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

ANDI (AND Immediate)

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ANDI # < data >, Dx

Attributes: Size = 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 is specified as a longword. The size of the immediate data is specified as a longword.

ANDI (AND Immediate), Continued

Condition Codes

X	N	Z	V	C
—	•	•	0	0

X—not affected
N—set if the most significant bit of the result is set; cleared otherwise
Z—set if the result is zero; cleared otherwise
V—always cleared
C—always cleared

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Instruction Fields *Register field* - specifies the destination data register, Dx

ASL, ASR (Arithmetic Shift)

Operation: Destination Shifted By Count → Destination
Assembler Syntax: ASd Dy,Dx; ASd # < data > , Dx where d is direction, L or R
Attributes: Size = Long

Description

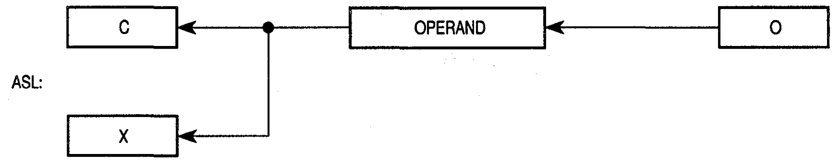
Arithmetically shifts the bits of the operand in the direction (L or R) specified. The carry bit receives the last bit shifted out of the operand. The shift count for the operation may be specified in two ways:

1. *Immediate*—The shift count is specified in the instruction (shift range,1 – 8).

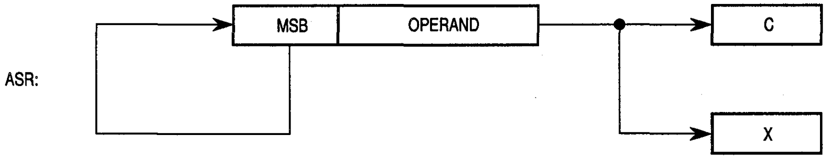
2. *Register*—The shift count is the value in the data register specified in instruction (modulo 64)For ASL, the operand is shifted left; the shift count equals the number of positions shifted. Bits shifted out of the high-order bit go to both the carry and the extend bits; zeros are shifted into the low-order bit. The overflow bit is always cleared.

ASL, ASR (Arithmetic Shift), Continued

Description
(Continued)



For ASR, the operand is shifted right; the number of positions shifted equals 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.



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 most significant bit of the result is set; cleared otherwise
Z—set if the result is zero; cleared otherwise
C—set according to the last bit shifted out of the operand; cleared for a shift count of zero
V— always cleared

Instruction Format

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

ASL, ASR Arithmetic Shift (Continued)

Instruction Fields

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

If i/r = 0, contains the shift count; values 1 – 7 represent counts of 1 –7; a value of zero represents a count of 8

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

dr field—specifies the direction of the shift:
0—shift right
1—shift left

i/r field
If i/r = 0, specifies immediate shift count
If i/r = 1, specifies register shift count

Register field—specifies the destination data register to be shifted, Dx

---

Bcc (Branch Conditionally)

Operation: If Condition True, Then $PC + d_n \rightarrow PC$

Assembler
Syntax: Bcc < label >

Attributes: Size = Byte, Word

Description

If the specified condition is true, program execution continues at location (PC) + displacement. The program counter contains the address of the instruction word for the Bcc instruction, plus two. The displacement is a two's-complement integer that represents the relative distance (in bytes) from the current program counter to the destination program counter. If the 8-bit displacement field in the instruction word is 0, a 16-bit displacement (the word immediately following the instruction) is used. Condition code CC specifies one of the following conditional tests:

MNEMONIC	CONDITION	MNEMONIC	CONDITION
CC(HI)	Carry Clear	LS	Low or Same
CS(LO)	Carry Set	LT	Less Than
EQ	Equal	MI	Minus
GE	Greater or Equal	NE	Not Equal
GT	Greater Than	PL	Plus
HI	High	VC	Overflow Clear
LE	Less or Equal	VS	Overflow Set

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	CONDITION				8-BIT DISPLACEMENT							
16-BIT DISPLACEMENT IF 8-BIT DISPLACEMENT = $00															

Bcc (Branch Conditionally), Continued

Instruction Fields

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

8-Bit Displacement field—two's 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

Note

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

BCHG (Test a Bit and Change)

Operation:	(< bit number > of Destination) → Z; (< bit number > of Destination) → < bit number > of Destination
Assembler Syntax:	BCHG Dy, < ea >x; BCHG # < data > , < ea >x
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:

Immediate— bit number is specified in a second word of the instruction

Register— specified data register contains the bit number

BCHG (Test a Bit and Change), Continued

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

Register field—specifies the data register that contains the bit number

Effective Address field—specifies the destination location; use only those data alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx	(xxx).W	111	000
Ax	—	—	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

*Longword only; all others are byte

Instruction Format (Bit Number Static, Specified as Immediate Data)

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

BCHG (Test a Bit and Change), Continued

Instruction Fields

Effective Address field—specifies the destination location; use only those data alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

ADDRESSING MODE	MODE	REGISTER
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

*Longword only; all others are byte

Bit Number field—specifies the bit number

BCLR (Test a Bit and Clear)

Operation: (< bit number > of Destination) → Z;
0 → < bit number > of Destination

Assembler Syntax: BCLR Dy, < ea >x; BCLR # < data >, < ea >x

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*—bit number is specified in a second word of the instruction

2. *Register*—specified data register contains the bit number

BCLR (Test a Bit and Clear), Continued

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		

Instruction Fields

Register field—specifies the data register that contains the bit number., Dy.

Effective Address field—specifies the destination location; use only those data alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx	(xxx).W	111	000
Ax	—	—	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

*Longword only; all others are byte

Instruction Format (Bit Number Static, Specified as Immediate Data)

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

BCLR (Test a Bit and Clear), Continued

Instruction Fields *Effective Address field*—specifies the destination location; use only those data alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx	(xxx).W	—	—
Ax	—	—	(xxx).L	—	—
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	—	—	(d_8,PC,Xn)	—	—

*Longword only; all others are byte

Bit Number field—specifies the bit number

BRA (Branch Always)

Operation: PC + d_n → PC

Assembler Syntax: BRA < label >

Attributes: Size = Byte, Word

Description Program execution continues at location (PC) + displacement. The program counter contains the address of the instruction word of the BRA instruction, plus two. The displacement is a two's-complement integer that represents the relative distance (in bytes) from the current program counter to the destination program counter. If the 8-bit displacement field in the instruction word is 0, a 16-bit displacement (the 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															

BRA (Branch Always), Continued

Instruction Fields

8-Bit Displacement field—two's-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

Note

A branch to the next immediate instruction requires the use of the 16-bit displacement format because the 8-bit displacement field contains $00 (zero offset).

BSET (Test a Bit and Set)

Operation:	TEST (< bit number > of Destination) → Z 1 → < bit number > of Destination
Assembler Syntax:	BSET Dy, < ea > x; BSET # < data >, < ea > x
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 0 refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

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

BSET (Test a Bit and Set), Continued

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

Register field—specifies the data register that contains the bit number, Dy

Effective Address field—specifies the destination location; use only those data alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx	(xxx).W	111	000
Ax	—	—	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
- (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

*Longword only; all others are byte

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								

BSET (Test a Bit and Set)

Instruction Fields *Effective Address field*—specifies the destination location; use only those data alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx	(xxx).W	—	—
Ax	—	—	(xxx).L	—	—
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	—	—	(d_8,PC,Xi)	—	—

*Longword only; all others are byte

Bit Number field—specifies the bit number

BSR (Branch to Subroutine)

Operation: SP – 4 → SP; Next PC → (SP); PC + d_n → PC
Assembler Syntax: BSR < label >
Attributes: Size = Byte, Word

Description Pushes the longword address of the instruction immediately following the BSR instruction onto the system stack. The program counter contains the address of the instruction word, plus two. Program execution then continues at location (PC) + displacement. The displacement is a two's-complement integer that represents the relative distance in bytes from the current program counter to the destination program counter. If the 8-bit displacement field in the instruction word is 0, a 16-bit displacement (the word immediately following the instruction) is used.

Condition Codes Not affected

BSR (Branch to Subroutine), Continued

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															

Instruction Fields

8-Bit Displacement field—two's-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

Note

A branch to the next immediate instruction requires the use of the 16-bit displacement format because the 8-bit displacement field contains $00 (zero offset).

BTST (Test a Bit)

Operation: (< bit number > of Destination) → Z

Assembler Syntax: BTST Dy, < ea >x; BTST # < data >, < ea >x

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 0 refers to the least significant bit. The bit number for this operation can be specified in either of two ways:

1. *Immediate*—bit number is specified in a second word of the instruction

2. *Register*—specified data register contains the bit number

BTST (Test a Bit), Continued

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

Register field—specifies the data register that contains the bit number, Dy

Effective Address field—specifies the destination location; use only those data addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx	(xxx).W	111	000
Ax	—	—	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	111	100
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	111	010
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	111	011

*Longword only; all others are byte

Instruction Format (Bit Number Static, Specified as Immediate Data)

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

BTST (Test a Bit), Continued

Instruction Fields

Effective Address field—specifies the destination location; use only those data addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dx*	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

ADDRESSING MODE	MODE	REGISTER
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

*Longword only; all others are byte

Bit Number field—specifies the bit number

CLR (Clear an Operand)

Operation: 0 → Destination

Assembler Syntax: CLR < ea > x

Attributes: Size = Byte, Word, Long

Description: Clears the destination operand. 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 word operation

Effective Address field—specifies the destination location; use only those data alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx	000	reg. number:Dx	(xxx).W	111	000
Ax	—	—	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax, Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

CMP (Compare)

Operation:	Destination – Source → cc
Assembler Syntax:	CMP < ea >y , Dx
Attributes:	Size = 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 is specified as a longword.

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			0	1	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields

Register field—specifies the destination data register

Effective Address field—specifies the source operand; use addressing modes as listed in the following table:

CMP (Compare), Continued

Instruction Fields (Continued)

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	001	reg. number:Ay
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

CMPA (Compare Address)

Operation: Destination – Source → cc

Assembler
Syntax: CMPA < ea >y , Ax

Attributes: Size = 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 is specified as a long word. 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			1	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields *Register field*—specifies the destination address register

Effective Address field—specifies the source operand; use addressing modes as listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	001	reg. number:Ay
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

CMPI (Compare Immediate)

Operation: Destination – Immediate Data → cc
Assembler
Syntax: CMPI # < data > , Dx
Attributes: Size = 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 is specified as a longword. The size of the immediate data is specified as a longword.

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

CMPI (Compare Immediate), Continued

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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Instruction Fields *Register field*—destination data register, Dx

DIVS, DIVSL (Signed Divide)

Operation: Destination ÷ Source → Destination
Assembler Syntax: DIVS.W < ea > ,Dn32/16 → 16r – 16q
*DIVS.L < ea > ,Dq32/32 → 32q
*DIVS.L < ea > ,Dr:Dq 64/32 → 32r – 32q
*DIVSL.L < ea > ,Dr:Dq32/32 → 32r – 32q
**Applies to MC68020, MC68030, MC68040, CPU32 only*
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 longword by a word. The result is a quotient in the lower word (least significant 16 bits) and a remainder in the upper word (most significant 16 bits). The sign of the remainder is the same as the sign of the dividend.

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

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

The **third long form** divides a longword by a longword. The result is a longword quotient and a longword 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.

Continued on next page

DIVS, DIVSL (Signed Divide), Continued

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

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 can be used as listed in the following tables:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	111	100
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011

MC68020, MC68030, and MC68040 only

(bd,An,Xn)*	110	reg. number:An	(bd,PC,Xn)*	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

*Can be used with CPU32.

DIVS, DIVSL (Signed Divide), Continued

Note

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

Instruction Format

Long

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

Instruction Fields

Effective Address field—Specifies the source operand. Only data alterable addressing modes can be used as listed in the following tables:

MC68020, MC68030, and MC68040 only

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_{16},An)	101	reg. number:An
(d_8,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_{16},PC)	111	010
(d_8,PC,Xn)	111	011
(bd,PC,Xn)	111	011

MC68020, MC68030, and MC68040 only

([bd,An,Xn],od)	110	reg. number:An
([bd,An],Xn,od)	110	reg. number:An

([bd,PC,Xn],od)	111	011
([bd,PC],Xn,od)	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

Continued on next page

DIVS, DIVSL (Signed Divide), Continued

Instruction Fields, (Continued)

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 the size field 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)

Operation:	Destination ÷ Source → Destination
Assembler Syntax:	DIVU.W < ea > ,Dn32/16 → 16r – 16q *DIVU.L < ea > ,Dq32/32 → 32q *DIVU.L < ea > ,Dr:Dq64/32 → 32r – 32q *DIVUL.L < ea > ,Dr:Dq 32/32 → 32r – 32q **Applies to MC68020, MC68030, MC68040, CPU32 only*
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 longword by a word. The result is a quotient in the lower word (least significant 16 bits) and a remainder in the upper word (most significant 16 bits).

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

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

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

Continued on next page

DIVU, DIVUL (Unsigned Divide), Continued

Description (Continued)

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; cleared otherwise; undefined if divide by zero occurs
C—Always cleared

Instruction Format

							Word								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										EFFECTIVE ADDRESS					
1	0	0	0	REGISTER			0	1	1	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 can be used as listed in the following tables:

Continued on next page

DIVU, DIVUL (Unsigned Divide), Continued

Instruction Fields (Continued)

MC68020, MC68030, and MC68040 only

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	111	100
(An) +	011	reg. number:An			
- (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011

MC68020, MC68030, and MC68040 only

(bd,An,Xn)*	110	reg. number:An	(bd,PC,Xn)*	111	011
([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	111	011

**Can be used with CPU32.

Note

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

Instruction Format

Long

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
<tr><td>0</td><td colspan="3">REGISTER Dq</td><td>0</td><td>SIZE</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="3">REGISTER Dr</td></tr>
</table>

Instruction Fields *Effective Address field*—Specifies the source operand. Only data addressing modes can be used as listed in the following tables:

Continued on next page

DIVU, DIVUL (Unsigned Divide), Continued

Instruction Fields (Continued)

MC68020, MC68030, and MC68040 only

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dn	000	reg. number:Dn	(xxx).W	111	000
An	—	—	(xxx).L	111	001
(An)	010	reg. number:An	#<data>	111	100
(An) +	011	reg. number:An			
– (An)	100	reg. number:An			
(d_{16},An)	101	reg. number:An	(d_{16},PC)	111	010
(d_8,An,Xn)	110	reg. number:An	(d_8,PC,Xn)	111	011
(bd,An,Xn)*	110	reg. number:An	(bd,PC,Xn)*	111	011

MC68020, MC68030, and MC68040 only

([bd,An,Xn],od)	110	reg. number:An	([bd,PC,Xn],od)	111	011
([bd,An],Xn,od)	110	reg. number:An	([bd,PC],Xn,od)	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 the size field 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)

Operation: Source ^ Destination → Destination
Assembler
Syntax: EOR Dy, < ea > x
Attributes: Size = Long

Description

Performs an exclusive OR operation on the destination operand using the source operand and stores the result in the destination location. Operation is specified as a longword. Source operand must be a data register. Destination operand is specified in the effective address field.

Condition Codes

X	N	Z	V	C
—	*	*	0	0

X—not affected
N—set if the most significant bit of the result is set; cleared otherwise
Z—set if the result is zero; cleared otherwise
V—always cleared
C—always cleared

Instruction Format

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

Instruction Fields

Register field—specifies the source data registers, Dy

Effective Address field—specifies the destination operand. Use only those data alterable addressing modes listed in the following table :

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx	000	reg. number:Dx	(xxx).W	111	000
Ax	—	—	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
- (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

EORI (Exclusive OR Immediate)

Operation: Immediate Data ⊕ Destination → Destination

Assembler Syntax: EORI # < data > , Dx

Attributes: Size = 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 operation size is specified as a longword.

Condition Codes

X	N	Z	V	C
—	*	*	0	0

X—not affected
N—set if the most significant bit of the result is set; cleared otherwise
Z—set if the result is zero; cleared otherwise
V—always cleared
C—always cleared

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Instruction Fields

Register field—destination data register, Dx

EXT, EXTB (Sign Extend)

Operation:	Destination Sign-Extended → Destination	
Assembler Syntax:	EXT.W	Dx extend byte to word
	EXT.L	Dx extend word to longword
	EXTB.L	Dx extend byte to longword
Attributes:	Size = Word, Long	

Description

Extends a byte in a data register to a word or a longword, or a word in a data register to a longword, by replicating the sign bit when the operation extends a byte to a word, bit 7 of the destination data register is copied to bits 15-8 of the data register. When the operation extends a word to a longword, bit 15 of the designated data register is copied to bits 31 – 16 of the data register. The EXTB form copies bit 7 of the designated register to bits 31 – 8 of the data register.

Condition Codes

X	N	Z	V	C
—	*	*	0	0

X—not affected
N—set if the result is negative; cleared otherwise
Z—set if the result is zero; cleared otherwise
V—always cleared
C—always cleared

Instruction Format

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

Instruction Fields

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

Register field—specifies that the data register is to be sign-extended, Dx

JMP (Jump)

Operation: Target Address → PC
Assembler
Syntax: JMP < ea > y
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 Field

Effective Address field—specifies the address of the next instruction; use only those control addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	—	—	(xxx).W	111	000
Ay	—	—	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	—	—
(Ay) +	—	—			
– (Ay)	—	—			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xi)	110	reg. number:Ay	(d_8,PC,Xi)	111	011

JSR (Jump to Subroutine)

Operation: SP – 4 → SP; Next PC → (SP); Target Address → PC
Assembler
Syntax: JSR < ea >y
Attributes: Unsized

Continued on next page

JSR (Jump to Subroutine), Continued

Description

Pushes the longword 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

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	0	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Field

Effective Address field—specifies the address of the next instruction; use only those control addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	—	—	(xxx).W	111	000
Ay	—	—	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	—	—
(Ay) +	—	—			
- (Ay)	—	—			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xi)	110	reg. number:Ay	(d_8,PC,Xi)	111	011

LEA (Load Effective Address)

Operation: Effective Address→ Destination

Assembler Syntax: LEA < ea >y, Ax

Attributes: Size = Long

Description Loads the effective address into the specified address register. This instruction affects all 32 bits of the address register.

Condition Codes Not affected

Instruction Format

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

Instruction Fields *Register field*—specifies the destination address register, Ax

Effective Address field—specifies the address to be loaded into the address register; use only those control addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	—	—	(xxx).W	111	000
Ay	—	—	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	—	—
(Ay) +	—	—			
- (Ay)	—	—			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xn)	110	reg. number:Ay	(d_8,PC,Xn)	111	011

LINK (Link and Allocate)

Operation: SP – 4 → SP; Ax → (SP); SP → Ax; SP + d_{16} → SP
Assembler Syntax: LINK Ax, # < displacement >
Attributes: Size = Word

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. The displacement is the sign-extended word following the operation word.

Condition Codes

Not affected

Instruction Format

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

Instruction Fields

Register field—specifies the address register for the link

Displacement field—specifies the two's-complement integer to be added to the stack pointer

LSR, LSL (Logical Shift Right, Left)

Operation: Destination Shifted By Count → Destination
Assembler Syntax: LSd Dy,Dx; LSd # < data > ,Dx
where d is direction, L or R
Attributes: Size = 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:

Continued on next page

LSR (Logical Shift), Continued

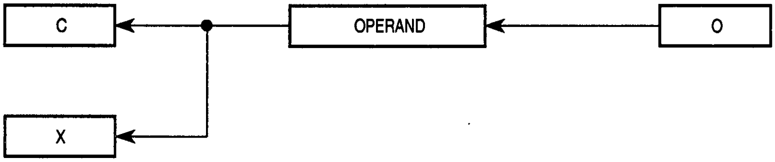
Description (Continued)

1. *Immediate*— shift count (1 – 8) is specified in the instruction

2. *Register*— shift count is the value in the data register specified in the instruction modulo 64

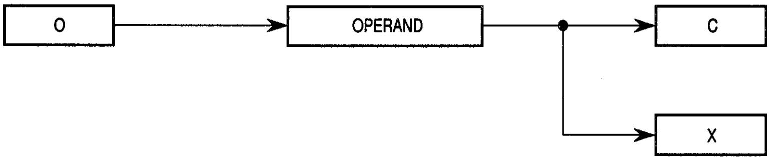
The LSL instruction shifts the operand to the left the number of positions specified as the shift count. Bits shifted out of the high-order bit go to both the carry and the extend bits; zeros are shifted into the low-order bit.

LSL:



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.

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

LSR (Logical Shift), Continued

Instruction Format

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

Instruction Fields

Count/Register field

If i/r = 0, this field contains the shift count; values 1 – 7 represent shifts of 1 – 7; value of 0 specifies shift count of 8

If i/r = 1, data register specified in this field contains shift count (modulo 64), Dy

dr field—specifies the direction of the shift:

0—shift right
1—shift left

i/r field

0—immediate shift count
1—register shift count

Register field—specifies the destination data register to be shifted,Dx

MOVE, MOVEA (Move Data from Source to Destination)

Operation:	Source → Destination
Assembler Syntax:	MOVE < ea >y , < ea >x; MOVEA <ea>y, Ax
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 longword.

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

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">0</td><td rowspan="2">0</td><td colspan="2" rowspan="2">SIZE</td><td colspan="6">DESTINATION</td><td colspan="6">SOURCE</td></tr>
<tr><td colspan="3">REGISTER</td><td colspan="3">MODE</td><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Fields

Size field—specifies the size of the operand to be moved:
01—byte operation
11—word operation
10—long operation

Destination Effective Address field—specifies the destination location; the possible data alterable addressing modes are listed in the table below. The ColdFire MOVE instruction has restrictions on combinations of source and destination addressing modes.

Continued on next page

MOVE, MOVEA (Move Data from Source to Destination),
Continued

Instruction Fields
(Continued)

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx	000	reg. number:Dx	(xxx).W	111	000
Ax*	001	reg. number: Ax	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

*If the destination is an address register, condition codes are unaffected. Some assemblers accept the MOVEA mneumonic to designate this slight difference.

Source Effective Address field—specifies the source operand; the possible addressing modes are listed in the table below. The ColdFire MOVE instruction has restrictions on combinations of source and destination addressing modes. The table shown below outlines the restrictions.

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy	(xxx).W	111	000
Ay	001	reg. number:Ay	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	111	100
(Ay) +	011	reg. number:Ay			
– (Ay)	100	reg. number:Ay			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xi)	110	reg. number:Ay	(d_8,PC,Xi)	111	011

Note

Most assemblers use MOVEA when the destination is an address register. Use MOVEQ to move an immediate 8-bit value to a data register. Not all combinations of source/destination addressing modes are possible. The next table shows the possible combinations.

Continued on next page

MOVE, MOVEA (Move Data from Source to Destination),
Continued

Note
(Continued)

SOURCE ADDRESSING MODE	DESTINATION ADDRESSING MODE
Dy, Ay, (Ay), (Ay)+,-(Ay)	All possible
(d_{16}, Ay), (d16, PC)	All possible except (d_8, Ay, Xi), (xxx).W, (xxx).L
(d8, Ay, Xi), (d8, PC, Xi), (xxx).W, (xxx).L, #<xxx>	All possible except (d_8, Ay, Xi), (d_{16}, Ay), (xxx).W, (xxx).L

Refer to the previous tables for valid source and destination addressing modes.

MOVE from CCR (Move from the Condition Code Register)

Operation: CCR → Destination
Assembler Syntax: MOVE CCR, Dx
Attributes: Size = Word

Description

Moves the condition code bits (zero-extended to word size) to the destination location. The operand size is a word.

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	1	0	1	1	0	0	0	REGISTER		

Instruction Fields

Register field—specifies the destination data register, Dx

MOVE to CCR (Move to Condition Code Register)

Operation: Source → CCR
Assembler
Syntax: MOVE, Dy ,CCR; MOVE #<data>, CCR
Attributes: Size = Word

Description

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

Condition Codes

X	N	Z	V	C
*	*	*	*	*

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

Instruction Format

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

Instruction Field

Effective Address field—specifies the location of the source operand; use only those data addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy	(xxx).W	—	—
Ay	—	—	(xxx).L	—	—
(Ay)	—	—	#<data>	111	100
(Ay) +	—	—			
- (Ay)	—	—			
(d_{16},Ay)	—	—	(d_{16},PC)	—	—
(d_8,Ay,Xi)	—	—	(d_8,PC,Xi)	—	—

MOVEM (Move Multiple Registers)

Operation:	Registers → Destination Source → Registers
Assembler Syntax:	MOVEM < list >, < ea >x; MOVEM < ea >y, < list >
Attributes:	Size = 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 registers are transferred starting at the specified address, and the address is incremented by the operand length (4) following each transfer. The order of the registers is from D0 to D7, then from A0 to A7.

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	dr	0	0	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		
REGISTER LIST MASK															

Instruction Fields

dr field—specifies the direction of the transfer:
0—register to memory
1—memory to register

Effective Address field—specifies the memory address for register-to-memory transfers

Continued on next page

MOVEM (Move Multiple Registers), Continued

Instruction Fields (Continued)

ADDRESSING MODE	MODE	REGISTER
Dy	—	—
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	—	—
- (Ay)	—	—
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

ADDRESSING MODE	MODE	REGISTER
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

For memory-to-register transfers, use addressing modes listed in the following tables:

ADDRESSING MODE	MODE	REGISTER
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	—	—
- (Ax)	—	—
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

ADDRESSING MODE	MODE	REGISTER
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

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. The mask correspondence is shown below.

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

MOVEQ (Move Quick)

Operation:	Immediate Data → Destination
Assembler Syntax:	MOVEQ # < data > ,Dx
Attributes:	Size = Long

Description

Moves a byte of immediate data to a 32-bit data register. The data in the 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 destination data register, Dx

Data field—8 bits of data, which are sign-extended to a long operand

MULS (Signed Multiply)

Operation:	Source x Destination → Destination
Assembler Syntax:	MULS.L < ea >y ,Dx 32 x 32 → 32 MULS.W < ea >y ,Dx 16 x 16 → 32
Attributes:	Size = Word, Long

Description

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

In the word form, the multiplier and multiplicand are both word operands, and the result is a longword 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 longword operands. The destination data register stores the low order 32-bits with the product. The upper 32 bits of the product are discarded.

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

Word

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		

Instruction Fields

Register field—specifies the destination data register, Dx.

Effective Address field—specifies the source operand; use only those data addressing modes listed in the following table:

Continued on next page

MULS (Signed Multiply), Continued

Instruction Fields (Continued)

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

Instruction Format

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 Dx			1	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields

Effective Address field—specifies the source operand; use only data addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

ADDRESSING MODE	MODE	REGISTER
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

Register Dx field—specifies the destination data register; the 32-bit multiplicand comes from this register, and the low-order 32 bits of the product are loaded into this register.

MULU (Unsigned Multiply)

Operation: Source x Destination → Destination

Assembler
Syntax: MULU.L < ea >y ,Dx 32 x 32 → 32
MULU.W < ea >y ,Dx 16 x 16 → 32

Attributes: Size = Word, Long

Description

Multiplies two unsigned operands yielding an unsigned result. This instruction has a word operand form and a long 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 longword operands, and the destination data register stores the low order 32 bits of the product. The upper 32 bits of the product are discarded.

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

Word

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

Instruction Fields

Register field—specifies the destination data register as the destination

Effective Address field—specifies the source operand; use only those data addressing modes listed in the following table:

Continued on next page

MULU (Unsigned Multiply), Continued

Instruction Fields (Continued)

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy	(xxx).W	111	000
Ay	—	—	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	111	100
(Ay) +	011	reg. number:Ay			
- (Ay)	100	reg. number:Ay			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xi)	110	reg. number:Ay	(d_8,PC,Xi)	111	011

Instruction Format

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			0	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields

Effective Address field—specifies the source operand; use only data addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy	(xxx).W	—	—
Ay	—	—	(xxx).L	—	—
(Ay)	010	reg. number:Ay	#<data>	—	—
(Ay) +	011	reg. number:Ay			
- (Ay)	100	reg. number:Ay			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	—	—
(d_8,Ay,Xi)	—	—	(d_8,PC,Xi)	—	—

Register Dx 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.

NEG (Negate)

Operation:	0 – Destination → Destination
Assembler Syntax:	NEG Dx
Attributes:	Size = Long

Description

Subtracts the destination operand from zero and stores the result in the destination location. The size of the operation is specified as a longword.

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	1	0	0	0	0	REGISTER		

Instruction Fields

Register field - specifies the destination data register, Dx

NEGX (Negate with Extend)

Operation:	0 – Destination – X → Destination
Assembler Syntax:	NEGX Dx
Attributes:	Size = 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 a longword.

NEGX (Negate with Extend), Continued

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

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	0	0	0	0	REGISTER		

Instruction Fields *Register field*—specifies the destination data register, Dx

NOP (No Operation)

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 have completed. This synchronizes the pipeline and prevents instruction overlap.

Condition Codes

Not affected

Instruction Format

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

NOT (Logical Complement)

Operation: ~ Destination → Destination
Assembler Syntax: NOT Dx
Attributes: Size = Long

Description

Calculates the logical complement of the destination operand and stores the result in the destination location. The size of the operation is specified as a longword.

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

NOT (Logical Complement), Continued

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	1	0	0	0	0	REGISTER		

Instruction Fields *Register field* - specifies destination data register, Dx

OR (Inclusive OR Logical)

Operation: Source | Destination → Destination

Assembler Syntax:
OR Dy, < ea > x
OR < ea >y ,Dx

Attributes: Size = 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 a longword. The contents of an address register cannot 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 8 data registers

Opmode field:

LONG	OPERATION
010	< ea >y | Dx → Dx
110	Dy | < ea >x → < ea >x

Effective Address field—if the location specified is a source operand, use only those data addressing modes listed in the following table:

Continued on next page

OR (Inclusive OR Logical), Continued

Instruction Fields (Continued)

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

If the location specified is a destination operand, use only those memory alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

Note

If the destination is a data register, specify using the destination Dx mode, not the destination < ea > mode.

ORI (Inclusive OR Immediate)

Operation:	Immediate Data | Destination → Destination
Assembler Syntax:	ORI # < data > , Dx
Attributes:	Size = 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 a longword. The size of the immediate data is specified as a longword.

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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Instruction Fields

Register field—destination data registers, Dx

PEA (Push Effective Address)

Operation: SP – 4 → SP; < ea >y → (SP)

Assembler Syntax: PEA < ea >y

Attributes: Size = Long

Description

Computes the effective address and pushes it onto the stack. The effective address is a long address.

Condition Codes

Not affected

Instruction Format

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

Instruction Field

Effective Address field—specifies the address to be pushed onto the stack; use only those control addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dy	—	—
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

RTS (Return from Subroutine)

Operation:	(SP) → PC; SP + 4 → SP
Assembler Syntax:	RTS
Attributes:	Unsized

Description

Pops 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

Scc (Set According to Condition)

Operation: If Condition True
Then 1s → Destination
Else 0s → Destination

Assembler Syntax: Scc Dx

Attributes: Size = Byte

Description

Tests the specified condition code; if the condition is true, sets the lowest byte of the destination data register to TRUE (all ones). Otherwise, sets that byte to FALSE (all zeros). Condition code cc specifies one of the following conditional tests:

MNEMONIC	CONDITION	MNEMONIC	CONDITION
CC(HI)	Carry Clear	LS	Low or Same
CS(LO)	Carry Set	LT	Less Than
EQ	Equal	MI	Minus
F	False	NE	Not Equal
GE	Greater or Equal	PL	Plus
GT	Greater Than	T	True
HI	High	VC	Overflow Clear
LE	Less or Equal	VS	Overflow Set

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	0	0	0	REGISTER		

Instruction Fields

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

Register field—specifies the destination data register, Dx

SUB (Subtract)

Operation: Destination – Source → Destination

Assembler Syntax: SUB Dy, < ea >x
SUB < ea >y ,Dx

Attributes: Size = 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 a long- word. The mode of the instruction indicates which operand is the source and which is the destination.

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

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">0</td><td rowspan="2">1</td><td colspan="3" rowspan="2">REGISTER</td><td colspan="3" rowspan="2">OPMODE</td><td colspan="6">EFFECTIVE ADDRESS</td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REGISTER</td></tr>
</table>

Instruction Fields

Register field—specifies any of the 8 data registers

Opmode field:

LONG	OPERATION
010	Dx – < ea >y → Dx
110	< ea >x – Dy → < ea >x

Effective Address field—Determines the addressing mode; if the location specified is a source operand, use addressing modes listed in the following table:

Continued on next page

SUB (Subtract), Continued

Instruction Fields
(Continued)

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	001	reg. number:Ay
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xi)	111	011

If the location specified is a destination operand, use only those memory alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

Note:

If the destination is a data register, it must be specified as a destination Dx address, not as a destination < ea >x address.

SUBA (Subract Address)

Operation: Destination – Source → Destination
Assembler Syntax: SUBA < ea >y ,Ax
Attributes: Size = Long

Continued on next page

SUBA (Subtract Address), Continued

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 a longword.

Condition Codes

Not affected

Instruction Format

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

Instruction Fields

Register field—specifies the destination address register, Ax

Effective Address field—specifies the source operand; use addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy
Ay	001	reg. number:Ay
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xn)	110	reg. number:Ay

ADDRESSING MODE	MODE	REGISTER
(xxx).W	111	000
(xxx).L	111	001
#<data>	111	100
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

SUBI (Subtract Immediate)

Operation:	Destination – Immediate Data → Destination
Assembler Syntax:	SUBI # < data > , Dx
Attributes:	Size = 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 a longword.

SUBI (Subract Immediate), Continued

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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Instruction Fields *Register field*—specifies the destination data register, Dx

SUBQ (Subtract Quick)

Operation: Destination – Immediate Data → Destination
Assembler
Syntax: SUBQ # < data > , < ea >x
Attributes: Size = Long

Description

Subtracts the immediate data (1 – 8) from the destination operand. The size of the operation is specified as a longword. When subtracting from address registers, the condition codes are not affected.

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

SUBQ (Subtract Quick), Continued

Instruction Format

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

Effective Address field—specifies the destination location; use only those alterable addressing modes listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dx	000	reg. number:Dx	(xxx).W	111	000
Ax	001	reg. number:Ax	(xxx).L	111	001
(Ax)	010	reg. number:Ax	#<data>	—	—
(Ax) +	011	reg. number:Ax			
– (Ax)	100	reg. number:Ax			
(d_{16},Ax)	101	reg. number:Ax	(d_{16},PC)	—	—
(d_8,Ax,Xi)	110	reg. number:Ax	(d_8,PC,Xi)	—	—

SUBX (Subtract with Extend)

Operation: Destination – Source – X → Destination
Assembler
Syntax: SUBX Dy,Dx
Attributes: Size = Long

Description

Subtracts the source operand and the extend bit from the destination operand and stores the result in the destination.

SUBX (Subtract with Extend), Continued

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 borrow occurs; cleared otherwise

Instruction Format

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

Instruction Fields

Dx field — specifies destination data register

Dy field — specifies source data register

SWAP (Swap Register Halves)

Operation: Register 31 – 16 ←→ Register 15 – 0
Assembler Syntax: SWAP Dx
Attributes: Size = Word

Description

Exchange the 16-bit words (halves) of a data register

Continued on next page

SWAP (Swap Register Halves), Continued

Condition Codes

X	N	Z	V	C
—	*	*	0	0

X—not affected
N—set if the most significant bit of the 32-bit result is set; cleared otherwise
Z—set if the 32-bit result is zero; cleared otherwise
V—always cleared
C—always cleared

Instruction Format

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

Instruction Field

Register field—specifies the destination data register to swap, Dx

TRAP (Trap)

Operation: 0 → T-Bit of SR
1 → S-Bit of SR
SP - 4 → SP; PC → (SP); SP - 2→ SP
SR → (SP); SSP – 2 → SP; Format/Vectort → (SP);
Vector → 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. The exception stack frame is stored at 0-modulo-4 memmory addresses. See Section 7 for more information about the operation of the self-aligning stack pointer.

Condition Codes

Not affected

Instruction Format

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

Instruction Fields

Vector field—specifies the trap vector to be taken

TRAPF (Trapf)

Operation: No operation

Assembler Syntax: TRAPF
TRAPF.W #<data>
TRAPF.L #<data>

Attributes: Unsized or Size = Word or Long

Continued on next page

TRAPF (Trapf), Continued

Description

This instruction performs no operation. It can be used to occupy 16, 32, or 48 bits in instruction space, and effectively provides a variable-length no operation 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	1	0	0	0	1	1	1	1	1	1	OPMODE		
OPTIONAL IMMEDIATE WORD															
OPTIONAL IMMEDIATE WORD															

Instruction Fields

OPMODE field:

010—instruction word followed by one extension word
011—instruction word followed by two extension words
100—instruction word without any additional extensions

TST (Test an Operand)

Operation: Destination Tested → Condition Codes
Assembler Syntax: TST < ea > y
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 longword.

Continued on next page

TST (Test an Operand), Continued

Condition Codes

X	N	Z	V	C
—	*	*	0	0

X—not affected
N—set if the operand is negative; cleared otherwise
Z—det 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—longword operation

Effective Address field—specifies the addressing mode for the destination operand as listed in the following table:

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dy	(xxx).W	111	000
Ay	001	reg. number:Ay	(xxx).L	111	001
(Ay)	010	reg. number:Ay	#<data>	111	100
(Ay) +	011	reg. number:Ay			
– (Ay)	100	reg. number:Ay			
(d_{16},Ay)	101	reg. number:Ay	(d_{16},PC)	111	010
(d_8,Ay,Xi)	110	reg. number:Ay	(d_8,PC,Xi)	111	011

*Word and longword operations only

UNLK (Unlink)

Operation: Ax → SP; (SP) → Ax; SP + 4 → SP
Assembler Syntax: UNLK Ax
Attributes: Unsized

Description

Loads the stack pointer from the specified address register, then loads the address register with the longword popped 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 Field

Register field—specifies the address register destination, Ax

Section 5
Supervisor (Privileged) Instructions

Overview

Introduction

This section contains information about the supervisor (privileged) instructions for the ColdFire Family. Each instruction is described in detail with the instruction descriptions arranged in alphabetical order by instruction mnemonic.

The supervisor instruction set has complete access to the user mode instructions in addition to those listed in Table 5-1.

OPCODE	SUPPORTED OPERAND SIZES	ADDRESSING MODES
CPUSHL	Unsized	Ay
HALT	Unsized	
MOVE from SR	Word	Dx
MOVE to SR	Word	Dy, SR #<data>, SR
MOVEC	Longword	Ry, Rc
RTE	Unsized	
STOP	Unsized	#<data>
WDEBUG	Longword	<ea>y

MOVEC Instruction

The MOVEC instruction providess access to the various control registers dealing with system-level functions. This includes all the configuration registers defining the address space as well as a single module base address register (MBAR) that provides the specification for the memory-mapped module configuration and control registers. The control register address, contained in bits [11:0] of the first extension word of the instruction, is defined in the next table.

Overview, Continued

Table 5-2: CPU Space Map

RC[11:0][1]	REGISTER DEFINITION
$002	Cache Control Register (CACR)
$004	Access Control Register 0 (ACR0)
$005	Access Control Register 1 (ACR1)
$08x [2]	Write the processor core address and data registers [2]
$18x [2]	Read the processor core address and data registers
$801	Vector Base Register (VBR)
$80E [2]	Status Register (SR) [2]
$80F	Program Counter(PC)
$C04	SRAM Base Address Register (RAMBAR)
$C0F	Module Base Address Register (MBAR)

[1] Any other address produces undefined results and should not be performed.

[2] Not accessible via MOVEC; accessible via the Debug interface, if present.

Note

The actual control registers in a given design are dependent of the on-chip memory and module configurations. In addition, a ColdFire processor only supports write access to all the control registers accessed by the MOVEC instruction.

CPUSHL (Push and Possibly Invalidate Cache)

Operation:

```
If Supervisor State, then if operand data, then
    push selected modified operand cache line
    if CPDI bit of CACR = 0, then
        invalidate selected cache line
    endif
else if instruction data, then
    If CPDI bit of CACR = 1, then
        invalidate selected cache line
    endif
endif
else TRAP
```

Assembler Syntax: CPUSHL , < ea >, (Ax)

Attributes: Unsized

Description

The CPUSHL instruction pushes modified cache lines and possibly invalidates the selected cache entries. If the addressed cache location contains modified data, the contents of the cache line are pushed to memory and the state of the line changed to simply "valid." For any execution of this instruction, the addressed cache entry is then invalidated if the CDPI bit of the CACR register is cleared. Otherwise, the selected cache entry is unchanged. The CACR is accessed using the MOVEC instruction.

Note

In all cases, the cache set is defined by bits[n:4] of the Ax value, where the exact value of "n" is cache-size dependent. Thus, the ColdFire version of this instruction addresses a specific cache location using the Ax register. The basic algorithm is (total cache capacity in bytes/ associativity/16 bytes/line) defines the required range. For an MCF5202 cache, the calculation would be:
(2048/4-way/16) =32 = 2^5 *-> so, address range is [8:4].*

Condition Codes Not affected

CPUSHL (Push and Possibly Invalidate Cache), Continued

Instruction Format

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

Register field—specifies the destination data register, Ax

HALT (Halt the CPU [Privileged])

Operation:	If Supervisor State Then Halt the Processor Core Else Privilege Violation Exception
Assembler Syntax:	HALT
Attributes:	Unsized

Description

The processor core is synchronized (meaning all previous instructions and bus cycles are completed), and then halts operation. The processor's halt status is signaled on the processor status output pins. If a "go" debug command is received, the processor resumes execution at the next instruction.

If bit 10 of the Debug module's configuration status register is asserted, execution of the HALT instruction in user mode is allowed.

Condition Codes

Not affected

Instruction Format

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

MOVEC (Move Control Register)

Operation: If Supervisor State
Then Ry → Rc
Else Privilege Violation Exception

Assembler Syntax: MOVEC Ry,Rc

Attributes: Size = Long

Description

Moves the contents of the 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.

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	1
A/D	REGISTER			CONTROL REGISTER											

Instruction Fields

A/D field—specifies the type of source register:
- 0—data register
- 1—address register

Actual control registers in a given design can vary. Only the VBR exists in all ColdFire designs. Do not attempt access to undefined control register space as it could yield undefined results. Access to unimplemented, but defined, control registers produces undefined results.

Register field—specifies the source register number, Ry

Control Register field—specifies the control register

MOVEC (Move Control Register)

Table 5-3: CPU Space Map

RC[11:0][1]	REGISTER DEFINITION
$002	Cache Control Register (CACR)
$004	Access Control Register 0 (ACR0)
$005	Access Control Register 1 (ACR1)
$08x [2]	Write the processor core address and data registers [2]
$18x [2]	Read the processor core address and data registers
$801	Vector Base Register (VBR)
$80E [2]	Status Register (SR) [2]
$80F	Program Counter(PC)
$C00	ROM Base Address Register (ROMBAR)
$C04	SRAM Base Address Register (RAMBAR)
$C0F	Module Base Address Register (MBAR)

[1] Any other address produces undefined results and should not be performed.

[2] Not accessible via MOVEC; accessible via the Debug interface, if present.

RTE (Return from Exception)

Operation:

```
If Supervisor state then
Test (SP);
(SP) → SR; 4 (SP) → PC; SP + 8→ SP
 Adjust stack according to format
Else
Privilege Violation Exception
```

Assembler Syntax: RTE

Attributes: Unsized

Description

Loads the processor state information stored in the exception stack frame located at the top of the stack into the processor. The instruction examines the stack format field in the format/offset word to determine how much information must be restored. If the format field is illegal, the processor generates a format-error exception.

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

MOVE from SR (Move from the Status Register)

Operation: If Supervisor State
Then SR → Destination
Else Privilege Violation Exception

Assembler
Syntax: MOVE SR, Dx
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	0	0	0	REGISTER		

Register field—specifies the destination data register, Dx.

MOVE to SR (Move to the Status Register)

Operation: If Supervisor State
Then Source → SR
Else Privilege Violation Exception

Assembler
Syntax: MOVE < ea >y, 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

MOVE to SR (Move to the Status Register), Continued

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	1	1	EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Field

Effective Address field—specifies the location of the source operand; use only those data addressing modes listed in the following table.

Table 5-4: Effective Data Addressing Modes

ADDRESSING MODE	MODE	REGISTER	ADDRESSING MODE	MODE	REGISTER
Dy	000	reg. number:Dn	(xxx).W	—	—
Ay	—	—	(xxx).L	—	—
(Ay)	—	—	# < data >	111	100
(Ay) +	—	—			
—(Ay)	—	—			
(d_{16},Ay)	—	—	(d_{16},PC)	—	—
(d_8,Ay,Xi)	—	—	(d_8,PC,Xi)	—	—

STOP (Load Status Register and Stop)

Operation: If Supervisor State
Then Immediate Data → SR; STOP the processor core
Else Privilege Violation Exception

Assembler Syntax: STOP # < data >

Attributes: Size = word

Description

1. Moves the immediate operand into the status register (both user and supervisor portions
2. Advances the program counter to point to the next instruction
3. Stops the fetching and executing of instructions

An interrupt or reset exception causes the processor to resume instruction 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. In the ColdFire processors, the STOP command places the processor in a low-power state.

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

WDEBUG (Write Debug Control Register)

Operation: If Supervisor State
Then Write Control Register
Command Executed in Debug Module
Else Privilege Violation Exception

Assembler Syntax: WDEBUG <ea>y

Attributes: Size = Long

Description

This instruction performs two functions. First, it fetches two consecutive long words from the memory location defined by the effective address. Second, it sends the operands to the ColdFire Debug module for execution as an instruction to write one of the Debug Control Registers (DRc). The memory location defined by the effective address must be on a longword address or the behavior of the operation is undefined. The debug command must be organized in memory as shown below.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0	1	0	0	0	DRc			
DATA[31:16]															
DATA[15:0]															
UNUSED															

where:

1. The first 16 bits define the "write debug register" command to the Debug module
2. The low-order 4 bits (DRc) define the specific control register being written
3. The 32-bit operand to be written is defined as data[31:0]
4. The lower 16 bits of the second longword of the instruction are unused

Condition Codes Not affected

Instruction Format

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

WDEBUG (Write Debug Control Register), Continued

Instruction Fields Effective Address field—determines the addressing mode for debug command location in memory.

ADDRESSING MODE	MODE	REGISTER
Dy		
Ay		
(Ay)	010	reg. number: Ay
(Ay) +	—	—
- (Ay)	—	—
(d_{16}, Ay)	101	reg. number: Ay
(d_8, Ay, Xi)	—	—

ADDRESSING MODE	MODE	REGISTER
(xxx).W		
(xxx).L		
#<data>		
(d_{16}, PC)		
(d_8, PC, Xi)		

Section 6
Instruction Format Summary

Overview

Introduction

This section contains a listing of the ColdFire Family instructions in binary format.

Instruction Format

Introduction

The following paragraphs present a summary of the binary encoding fields.

Effective Address Field

This field specifies which addressing mode is to be used. Some operations have hardware-enforced restrictions on the available addressing modes.

Shift Instruction

The following paragraphs define the fields used with the shift instructions.

Count Register Field

If i/r = 0, this field contains the shift count of 1 – 8 (a zero specifies 8). If i/r = 1, this field specifies a data register that contains the shift count.

The following shift fields are encoded as follows:

dr field:
- 0—shift right
- 1—shift left

i/r field:
- 0—immediate shift count
- 1—register shift count

Register Field

This field specifies a data register to be shifted.

Size Field

This field specifies the size of the operation and is encoded as follows:

- 00—byte operation
- 01—word operation
- 10—long operation

Opmode Field

Refer to the applicable instruction descriptions for the encoding of this field.

Instruction Format, Continued

Address/Data Field This field specifies the type of general register and is encoded as follows:

0—data register
1—address register

Operation Code Map

Introduction

Table 6-1 lists the encoding for bits 15 – 12 and the operation performed.

Table 6-1: Operation Code Map

BITS 15 - 12	OPERATION
0000	Bit Manipulation/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/Scc
0110	Bcc/BSR/BRA
0111	MOVEQ
1000	OR
1001	SUB/SUBX
1010	optional MAC opcodes
1011	CMP/EOR
1100	AND/MUL
1101	ADD/ADDX
1110	Shift
1111	CPUSHL/WDDATA/WDEBUG

Opcodes

The following opcodes are sorted by numeric value.

Continued on next page

Operation Code Map, Continued

1. 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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

2. BTST

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		

3. BCHG

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		

4. BCLR

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		

5. BSET

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		

6. 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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

7. 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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Operation Code Map, Continued

8. 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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

9. BTST

BIT NUMBER STATIC, SPECIFIED AS IMMEDIATE DATA

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

10. BCHG

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							

11. BCLR

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							

12. BSET

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								

13. 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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

Operation Code Map, Continued

14. 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	1	0	0	0	0	REGISTER		
UPPER WORD OF IMMEDIATE DATA															
LOWER WORD OF IMMEDIATE DATA															

15. MOVE

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		

16. NEGX

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	0	0	0	0	REGISTER		

17. MOVE from SR

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	0	0	0	REGISTER		

18. LEA

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

19. CLR

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		

20. MOVE from CCR

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		0	0	0	REGISTER		

21. NEG

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	0	0	0	0	REGISTER		

Operation Code Map, Continued

22. MOVE to CCR

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	0	0	0	REGISTER		

23. NOT

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	0	0	0	0	REGISTER		

24. 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		

25. 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		

26. 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		

27. EXT, EXTB

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

28. MOVEM

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

29. 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		

Operation Code Map, Continued

30. HALT

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	0	0	1	0	0	0

31. PULSE

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

32. 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

33. MULU.L

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 DX			0	0	0	0	0	0	0	0	0	0	0	0

34. MULS.L

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 Dx			1	0	0	0	0	0	0	0	0	0	0	0

35. 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			

36. LINK

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		
WORD DISPLACEMENT															

37. UNLINK

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		
WORD DISPLACEMENT															

Operation Code Map, Continued

38. 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

39. STOP

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

40. 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

41. 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

42. 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	1
A/D	REGISTER			CONTROL REGISTER											

43. 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		

44. 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		

45. ADDQ

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

Operation Code Map, Continued

46. 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		

47. TRAPF

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	MODE		

48. SUBQ

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

49. 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															

50. 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															

51. 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															

52. MOVEQ

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

53. 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		

Operation Code Map, Continued

54. 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		

55. SUBX

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

56. SUBA

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

57. CMP

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

58. EOR

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

59. CMPA

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

60. 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		

61. MULU.W

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		

Operation Code Map, Continued

62. MULS.W

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		

63. 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		

64. ADDX

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

65. ADDA

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

66. ASL, ASR

REGISTER SHIFT

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

67. LSL, LSR

REGISTER SHIFT

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

68. WDDATA

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

69. WDEBUG

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

Operation Code Map, Continued

70. CPUSHL

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

Section 7

Exception Processing

Overview

Introduction

Exception processing is the activity performed by the processor in preparing to execute a special routine for any condition that causes an exception. Exception processing does not include execution of the routine itself.

This section describes the processing for each type of integer unit exception, exception priorities, the return from an exception, and bus fault recovery. Also described are the formats of the exception stack frames.

Exception Processing Basics

Exception processing for ColdFire processors is streamlined for performance. Differences from previous 68000 Family processors include:

- A simplified exception vector table
- Reduced relocation capabilities using the vector base register
- A single exception stack frame format
- Use of a single self-aligning system stack

ColdFire 5200 processors use an instruction restart exception model but do require more software support to recover from certain access errors. See the subsection on **Access Error Exception** for details.

Four Steps of Exception Processing

Introduction

Exception processing is comprised of four major steps and can be defined as the time from the detection of the fault condition until the fetch of the first handler instruction has been initiated.

Step 1

First, the processor makes an internal copy of the SR and then enters supervisor mode by asserting the S-bit and disabling trace mode by negating the T-bit. The occurrence of an interrupt exception also forces the M-bit to be cleared and the interrupt priority mask to be set to the level of the current interrupt request.

Step 2

Second, the processor determines the exception vector number. For all faults *except* interrupts, the processor performs this calculation based on the exception type. For interrupts, the processor performs an interrupt-acknowledge (IACK) bus cycle to obtain the vector number from a peripheral device. The IACK cycle is mapped to a special acknowledge address space with the interrupt level encoded in the address.

Step 3

Third, the processor saves the current context by creating an exception stack frame on the system stack. ColdFire 5200 processors support a single stack pointer in the A7 address register; therefore, there is no notion of separate supervisor or user stack pointers. As a result, the exception stack frame is created at a 0-modulo-4 address on the top of the current system stack.

Additionally, the processor uses a simplified fixed-length stack frame for all exceptions. The exception type determines whether the program counter placed in the exception stack frame defines the location of the faulting instruction (fault) or the address of the next instruction to be executed (next).

Step 4

Fourth, the processor calculates the address of the first instruction of the exception handler. By definition, the exception vector table is aligned on a 1 Mbyte boundary. This instruction address is generated by fetching an exception vector from the table located at the address defined in the vector base register.

Continued on next page

Four Steps of Exception Processing, Continued

Step 4 (Continued)

The index into the exception table is calculated as (4 x vector_number). Once the exception vector has been fetched, the contents of the vector determine the address of the first instruction of the desired handler. After the instruction fetch for the first opcode of the handler has been initiated, exception processing terminates and normal instruction processing continues in the handler.

1024-Byte Vector Table

ColdFire 5200 processors support a 1024-byte vector table aligned on any 1 Mbyte address boundary (see Table 7-1). The table contains 256 exception vectors where the first 64 are defined by Motorola and the remaining 192 are user-defined interrupt vectors.

Table 7-1: Exception Vector Assignments

VECTOR NUMBER(S)	VECTOR OFFSET (HEX)	STACKED PROGRAM COUNTER	ASSIGNMENT
0	$000	-	Initial stack pointer
1	$004	-	Initial program counter
2	$008	Fault	Access error
3	$00C	Fault	Address error
4	$010	Fault	Illegal instruction
5-7	$014-$01C	-	Reserved
8	$020	Fault	Privilege violation
9	$024	Next	Trace
10	$028	Fault	Unimplemented line-a opcode
11	$02C	Fault	Unimplemented line-f opcode
12	$030	Next	Debug interrupt
13	$034	-	Reserved
14	$038	Fault	Format error
15	$03C	Next	Uninitialized interrupt
16-23	$040-$05C	-	Reserved
24	$060	Next	Spurious interrupt
25-31	$064-$07C	Next	Level 1-7 autovectored interrupts
32-47	$080-$0BC	Next	Trap # 0-15 instructions
48-63	$0C0-$0FC	-	Reserved
64-255	$100-$3FC	Next	User-defined interrupts

"Fault" refers to the PC of the instruction that caused the exception
"Next" refers to the PC of the next instruction that follows the instruction that caused the fault.

Interrupt Sampling and ColdFire 5200 Processors

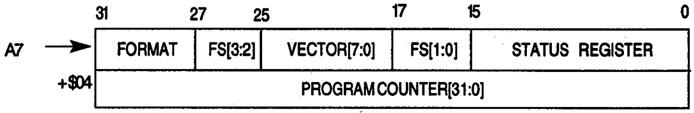
ColdFire 5200 processors inhibit sampling for interrupts during the first instruction of all exception handlers. This allows any handler to effectively disable interrupts, if necessary, by raising the interrupt mask level contained in the status register.

Exception Stack Frame Definition

Introduction

The exception stack frame is shown in Figure 7-1. The first longword of the exception stack frame contains the 16-bit format/vector word (F/V) and the 16-bit status register, and the second longword contains the 32-bit program counter address.

Figure 7-1: Exception Stack Frame Form



Three Unique Fields of the 16-Bit Format/ Vector Word

1. A 4-bit format field at the top of the system stack is always written with a value of {4,5,6,7} by the processor indicating a two-longword frame format. See the following table.

ORIGINAL A7 @ TIME OF EXCEPTION, BITS 1:0	A7 @ 1ST INSTRUCTION OF HANDLER	FORMAT FIELD
00	Original A7 - 8	4
01	Original A7 - 9	5
10	Original A7 - 10	6
11	Original A7 - 11	7

2. A 4-bit fault status field, FS[3:0], at the top of the system stack. This field is defined for access and address errors only and written as zeros for all other types of exceptions. See the following table.

FS[3:0]	DEFINITION
00xx	Reserved
0100	Error on instruction fetch
0101	Reserved
011x	Reserved
1000	Error on operand write
1001	Attempted write to write-protected space
101x	Reserved
1100	Error on operand read
1101	Reserved
111x	Reserved

Exception Stack Frame Definition, Continued

Three Unique Fields of the 16-Bit Format/ Vector Word (Continued)

3. The 8-bit vector number, vector[7:0], defines the exception type and is calculated by the processor for all internal faults and represents the value supplied by the peripheral in the case of an interrupt. Refer to Table 7-1.

Processor Exceptions

Access Error Exception: Instruction Fetch

The exact processor response to an access error depends on the type of memory reference being performed. For an **instruction fetch**, the processor postpones the error reporting until the faulted reference is needed by an instruction for execution. Therefore, faults that occur during instruction prefetches that are then followed by a change of instruction flow will not generate an exception.

Access Error Exception: Instruction with Faulted Opword

When the processor tries to execute an **instruction with a faulted opword** and/or extension words, the access error will be signaled and the instruction aborted. For this type of exception, the programming model has not been altered by the instruction generating the access error.

Access Error Exception: Operand Read

If the access error occurs on an **operand read**, the processor immediately aborts the current instruction's execution and initiates exception processing. In this situation, any address register updates attributable to the auto-addressing modes, {e.g., (An)+,-(An)}, will already have been performed.

So, the programming model contains the updated An value. In addition, if an access error occurs during the execution of a MOVEM instruction loading from memory, any registers already updated *before* the fault occurs will contain the operands from memory.

Access Error Exception: Operand Writes

The ColdFire processor uses an imprecise reporting mechanism for access errors on **operand writes**. Because the actual write cycle may be decoupled from the processor's issuing of the operation, the signaling of an access error appears to be decoupled from the instruction that generated the write. Accordingly, the PC contained in the exception stack frame merely represents the location in the program when the access error was signaled.

All programming model updates associated with the write instruction are completed. The NOP instruction can collect access errors for writes. This instruction delays its execution until all previous operations, including all pending write operations, are complete. If any previous write terminates with an access error, it is guaranteed to be reported on the NOP instruction.

Processor Exceptions, Continued

Address-Error Exception

Any attempted execution transferring control to an odd instruction address (i.e., if bit 0 of the target address is set) results in an **address-error exception**. Any attempted use of a word-sized index register (Xn.w) or a scale factor of 8 on an indexed effective addressing mode generates an address error as does an attempted execution of a full-format indexed addressing mode (refer to the *M68000 Programmer's Reference Manual* for information on 680X0 family opcodes).

Illegal Instruction Exception

The attempted execution of the $0000 and the $4AFC opwords generates an **illegal instruction exception**. Additionally, the attempted execution of any line A and most line F opcode generates their unique exception types, vector numbers 10 and 11, respectively. ColdFire 5200 processors do not provide illegal instruction detection on the extension words on any instruction, including MOVEC. If any other nonsupported opcode is executed, the resulting operation is undefined.

Privilege Violation

The attempted execution of a supervisor mode instruction while in user mode generates a **privilege violation** exception. This *ColdFire Programmer's Reference Manual* revision contains lists of supervisor- and user-mode instructions.

Trace Exception

To aid in program development, the ColdFire 5200 processors provide an instruction-by-instruction tracing capability. While in trace mode, indicated by the assertion of the T-bit in the status register (SR[15] = 1), the completion of an instruction execution signals a **trace exception**. This functionality lets a debugger monitor program execution.

The single exception to this definition is the STOP instruction. When the STOP opcode is executed, the processor core waits until an unmasked interrupt request is asserted, then aborts the pipeline and initiates interrupt exception processing.

Because ColdFire processors do not support any hardware stacking of multiple exceptions, it is the responsibility of the operating system to check for trace mode after processing other exception types. As an example, consider the execution of a TRAP instruction while in trace

Continued on next page

Processor Exceptions, Continued

Trace Exception (Continued)

mode. The processor will initiate the TRAP exception and then pass control to the corresponding handler. If the system requires that a trace exception be processed, it is the responsibility of the TRAP exception handler to check for this condition (SR[15] in the exception stack frame asserted) and pass control to the trace handler before returning from the original exception.

Debug Interrupt

This exception is generated in response to a hardware breakpoint register trigger. The processor does not generate an IACK cycle but rather calculates the vector number internally (vector number 12).

RTE and Format Error Exceptions

When an RTE instruction is executed, the processor first examines the 4-bit format field to validate the frame type. For a ColdFire 5200 processor, any attempted execution of an RTE where the format is not equal to {4,5,6,7} generates a **format error**. The exception stack frame for the format error is created without disturbing the original RTE frame and the stacked PC pointing to the RTE instruction.

The selection of the format value provides some limited debug support for porting code from 68000 applications. On 680x0 Family processors, the SR was located at the top of the stack. On those processors, bit[30] of the longword addressed by the system stack pointer is typically zero. Thus, if an RTE is attempted using this “old” format, it generates a format error on a ColdFire 5200 processor.

If the format field defines a valid type, the processor

1. Reloads the SR operand
2. fetches the second longword operand
3. Adjusts the stack pointer by adding the format value to the auto-incremented address after the fetch of the first longword
4. Transfers control to the instruction address defined by the second longword operand within the stack frame.

TRAP Instruction Exceptions

The TRAP #n instruction always forces an exception as part of its execution and is useful for implementing system calls.

Processor Exceptions, Continued

Interrupt Exception

The **interrupt exception** processing, with interrupt recognition and vector fetching, includes uninitialized and spurious interrupts as well as those where the requesting device supplies the 8-bit interrupt vector. Autovectoring may optionally be supported through the System Integration Module (SIM).

Fault-on-Fault Halt

If a ColdFire 5200 processor encounters any type of fault during the exception processing of another fault, the processor immediately halts execution with the catastrophic **fault-on-fault** condition. A reset is required to force the processor to exit this halted state.

Reset Exception

Asserting the reset input signal to the processor causes a **reset exception.** The reset exception has the highest priority of any exception; it provides for system initialization and recovery from catastrophic failure. Reset also aborts any processing in progress when the reset input is recognized. Processing cannot be recovered.

The reset exception places the processor in the supervisor mode by setting the S-bit and disables tracing by clearing the T-bit in the SR. This exception also clears the M-bit and sets the processor's interrupt priority mask in the SR to the highest level (level 7). Next, the VBR is initialized to zero ($00000000). The control registers specifying the operation of any memories (e.g., cache and/or RAM modules) connected directly to the processor are disabled.

Note

Other implementation-specific supervisor registers are also affected. Refer to the specific user's manual for details.

Reset Exception (Continued)

Once the processor is granted the bus and it does not detect any other alternate masters taking the bus, the core then performs two longword read bus cycles. The first longword at address 0 is loaded into the stack pointer and the second longword at address 4 is loaded into the program counter. After the initial instruction is fetched from memory, program execution begins at the address in the PC. If an access error or address error occurs before the first instruction is executed, the processor enters the fault-on-fault halted state.

Section 8
S-Record Output Format

Overview

Introduction

The S-record format for output modules is for encoding programs or data files in a printable format for transportation between computer systems. The transportation process can be visually monitored, and the S-records can be easily edited.

S-Record Content

Introduction

Visually, S-records are essentially character strings made of several fields that identify the record type, record length, memory address, code/data, and checksum. Each byte of binary data encodes as a two-character hexadecimal number: the first character represents the high-order four bits, and the second character represents the low-order four bits of the byte. Figure 8-1 illustrates the five fields that comprise an S-record. Table 8-1 lists the composition of each S-record field.

Figure 8-1: Five Fields of an S-Record

TYPE	RECORD LENGTH	ADDRESS	CODE/DATA	CHECKSUM

Table 8-1: Field Composition of an S-Record

FIELD	PRINTABLE CHARACTERS	CONTENTS
Type	2	S-record type—S0, S1, etc.
Record Length	2	The count of the character pairs in the record, excluding the type and record length.
Address	4, 6, or 8	The 2-, 3-, or 4-byte address at which the data field is to be loaded into memory.
Code/Data	0–2n	From 0 to n bytes of executable code, memory loadable data, or descriptive information. For compatibility with teletypewriters, some programs may limit the number of bytes to as few as 28 (56 printable characters in the S-record).
Checksum	2	The least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the record length, address, and the code/data fields.

Downloading S-Records

When downloading S-records, each must be terminated with a CR. Additionally, an S-record may have an initial field that fits other data such as line numbers generated by some time-sharing systems. The record length (byte count) and checksum fields ensure transmission accuracy.

S-Record Types

Types of S-Records

There are 8 types of S-records to accommodate the encoding, transportation, and decoding functions. The various Motorola record transportation control programs (e.g., upload, download, etc.), cross assemblers, linkers, and other file creating or debugging programs, only use S-records serving the program's purpose. For more information on support of specific S-records, refer to the user's manual for that program.

Types of S-Record Format Modules

An S-record format module may contain S-records of the following types:

S0—The header record for each block of S-records. The code/data field may contain any descriptive information identifying the following block of S-records. Under VERSAdos, the resident linker IDENT command can be used to designate module name, version number, revision number, and description information that will make up the header record. The address field is normally zeros.

S1—A record containing code/data and the 2-byte address at which the code/data is to reside.

S2—A record containing code/data and the 3-byte address at which the code/data is to reside.

S3—A record containing code/data and the 4-byte address at which the code/data is to reside.

S5—A record containing the number of S1, S2, and S3 records transmitted in a particular block. This count appears in the address field. There is no code/data field.

S7—A termination record for a block of S3 records. The address fieldmay optionally contain the 4-byte address of the instruction to which control is to be passed. There is no code /data field.

S8—A termination record for a block of S2 records. The address field may optionally contain the 3-byte address of the instruction to which control is to be passed. There is no code /data field.

Continued on next page

S-Record Types, Continued

Types of S-Record Format Modules (Continued)

S9—A termination record for a block of S1 records. The address field may optionally contain the 2-byte address of the instruction to which control is to be passed. Under VERSAdos, the resident linker ENTRY command can be used to specify this address. If this address is not specified, the first entry point specification encountered in the object module

Each block of S-records uses only one termination record. S7 and S8 records are only active when control passes to a 3- or 4-byte address; otherwise, an S9 is used for termination. Normally, there is only one header record, although it is possible for multiple header records to occur.

S-Record Creation

Introduction

Dump utilities, debuggers, a VERSAdos resident linkage editor, or cross assemblers and linkers produce S-record format programs. On VERSAdos systems, the build load module (MBLM) utility builds an executable load module from S-records. It has a counterpart utility in BUILDS that creates an S-record file from a load module.

Programs are available for downloading or uploading a file in S- record format from a host system to an 8- or 16-bit microprocessor- based system.

S-Record Format Module Example

A typical S-record format module is printed or displayed as follows:

```
S00600004844521B
S1130000285F245F2212226A000424290008237C2A
S11300100002000800082629001853812341001813
S113002041E900084E42234300182342000824A952
S107003000144ED492
S9030000FC
```

The module has an S0 record, four S1 records, and an S9 record. The following character pairs comprise the S-record format module.

S0 Record:

S0—S-record type S0, indicating that it is a header record
06—Hexadecimal 06 (decimal 6), indicating that six character pairs (or ASCII bytes) follow
0000—A 4-character, 2-byte address field; zeros in this example
48—ASCII H
44—ASCII D
52—ASCII R
1B—The checksum

First S1 Record:

S1—S-record type S1, indicating that it is a code/data record to be loaded/verified at a 2-byte address
13—Hexadecimal 13 (decimal 19), indicating that 19 character pairs, representing 19 bytes of binary data, follow *???*

Continued on next page

S-Record Creation, Continued

S-Record Format Module Example (Continued)

0000—A 4-character, 2-byte address field (hexadecimal address 0000) indicating where the data that follows is to be loaded

The next 16 character pairs of the first S1 record are the ASCII bytes of the actual program code/data. In this assembly language example, the program hexadecimal opcodes are sequentially written in the code/data fields of the S1 records.

OPCODE	INSTRUCTION	
285F	MOVE.L	(A7) +, A4
245F	MOVE.L	(A7) +, A2
2212	MOVE.L	(A2), D1
226A0004	MOVE.L	4(A2), A1
24290008	MOVE.L	FUNCTION(A1), D2
237C	MOVE.L	#FORCEFUNC, FUNCTION(A1)

The rest of this code continues in the remaining S1 record's code/data fields and stores in memory location 0010, etc.

2A—The checksum of the first S1 record.

The second and third S1 records also contain hexadecimal 13 (decimal 19) character pairs and end with checksums 13 and 52, respectively. The fourth S1 record contains 07 character pairs and has a checksum of 92.

S9 Record:

S9 —S-record type S9, indicating that it is a termination record
03 —Hexadecimal 03, indicating that three character pairs (3 bytes) follow
0000—The address field, zeros
FC—The checksum of the S9 record

Each printable character in an S-record encodes in hexadecimal (ASCII in this example) representation of the binary bits that transmit. Figure 8-2 illustrates the sending of the first S1 record. Table 8-2 lists the ASCII code for S-records.

S-Record Creation, Continued

Figure 8-2: ASCII Code for S-Records

TYPE				RECORD LENGTH				ADDRESS								CODE/DATA									CHECKSUM			
S		1		1		3		0		0		0		0		2		8		5		F		****	2		A	
5	3	3	1	3	1	3	3	3	0	3	0	3	0	3	0	3	2	3	8	3	5	4	6	****	3	2	4	1
0101	0011	0011	0001	0011	0001	0011	0011	0011	0000	0011	0000	0011	0000	0011	0000	0011	0010	0011	1000	0011	0101	0100	0110	****	0011	0010	0100	0001

Table 8-2: Transmission of an S1 Record

LEAST SIGNIFICANT DIGIT	MOST SIGNIFICANT DIGIT							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	–	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Section 9

Instruction Execution Timing (5200 Series Only)

Overview

Introduction

This section presents ColdFire 5200 Series processor instruction execution times in terms of processor core clock cycles. The number of operand references for each instruction is also included, enclosed in parentheses following the number of clock cycles. Each timing entry is presented as C(r/w) where:

C—The number of processor clock cycles, including all applicable operand fetches and writes, as well as all internal core cycles required to complete the instruction execution.

r/w—The number of operand reads (r) and writes (w) required by the instruction. An operation performing a read-modify-write function is denoted as (1/1).

This section includes assumptions concerning the timing values and the execution time details.

Timing Assumptions

Four Timing Assumptions

The timing data presented in this section have the following assumptions:

1. *The operand execution pipeline (OEP) is loaded with the opword and all required extension words at the beginning of each instruction execution.* This implies that the OEP doesn't wait for the instruction fetch pipeline (IFP) to supply opwords and/or extension words.

2. *The OEP does not experience any sequence-related pipeline stalls.* For the ColdFire processor, the most common example of this type of stall involves consecutive STORE operations, excluding the MOVEM instruction. For all STORE operations (except MOVEM), certain hardware resources within the ColdFire processor are marked as "busy" for two clock cycles after the final DSOC cycle of the STORE instruction. If a subsequent STORE instruction is encountered within this 2-cycle window, it will be stalled until the resource again becomes available. Thus, the maximum pipeline stall involving consecutive STORE operations is 2 cycles. The MOVEM instruction uses a different set of resources and this stall does not apply.

3. *The OEP completes all memory accesses without any stall conditions caused by the memory itself.* Thus, the timing details provided in this section assume an infinite zero-wait state memory is attached to the processor core.

4. *All operand data accesses are aligned on the same byte boundary as the operand size:* 16-bit operands aligned on 0-modulo-2 addresses, 32-bit operands aligned on 0-modulo-4 addresses.

If the operand alignment fails these guidelines, the optional hardware module that supports misaligned references is required. With the support this module provides, each misaligned reference requires a minimum of 2 additional clock cycles to process.

MOVE Instruction Execution Times

Introduction

The execution times for the MOVE.{B,W} instructions are shown in Table 9-1, while Table 9-2 provides the timing for MOVE.L.

For all tables in this section, the execution time (ET) of any instruction using the PC-relative effective addressing modes is exactly equivalent to the time using the comparable An-relative mode.

The nomenclature "xxx.wl" refers to both forms of absolute addressing, xxx.w and xxx.l.

Table 9-1: Move Byte and Word Execution Times

SOURCE	DESTINATION						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xn*SF)	xxx.wl
Dy	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
Ay	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
(Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
(Ay)+	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
-(Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)
(d16,Ay)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,Ay,Xn*SF)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
xxx.w	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
xxx.l	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
(d16,PC)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—
(d8,PC,Xn*SF)	4(1/0)	4(1/1)	4(1/1)	4(1/1)	—	—	—
#xxx	1(0/0)	3(0/1)	3(0/1)	3(0/1)	—	—	—

Table 9-2: Move Long Execution Times

SOURCE	DESTINATION						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xn*SF)	xxx.wl
Dy	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
Ay	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)
(Ay)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(Ay)+	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
-(Ay)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	3(1/1)	2(1/1)
(d16,Ay)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,Ay,Xn*SF)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
xxx.w	2(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
xxx.l	2(1/0)	2(1/1)	2(1/1)	2(1/1)	—	—	—
(d16,PC)	2(1/0)	2(1/1)	2(1/1)	2(1/1)	2(1/1)	—	—
(d8,PC,Xn*SF)	3(1/0)	3(1/1)	3(1/1)	3(1/1)	—	—	—
#xxx	1(0/0)	2(0/1)	2(0/1)	2(0/1)	—	—	—

Standard One Operand Instruction Execution Times

Table 9-3: One Operand Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn*SF)	xxx.wl	#xxx
CLR.B	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
CLR.W	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
CLR.L	<ea>	1(0/0)	1(0/1)	1(0/1)	1(0/1)	1(0/1)	2(0/1)	1(0/1)	—
EXT.W	Dx	1(0/0)	—	—	—	—	—	—	—
EXT.L	Dx	1(0/0)	—	—	—	—	—	—	—
EXTB.L	Dx	1(0/0)	—	—	—	—	—	—	—
NEG.L	Dx	1(0/0)	—	—	—	—	—	—	—
NEGX.L	Dx	1(0/0)	—	—	—	—	—	—	—
NOT.L	Dx	1(0/0)	—	—	—	—	—	—	—
SCC	Dx	1(0/0)	—	—	—	—	—	—	—
SWAP	Dx	1(0/0)	—	—	—	—	—	—	—
TST.B	<ea>	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
TST.W	<ea>	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
TST.L	<ea>	1(0/0)	2(1/0)	2(1/0)	2(1/0)	2(1/0)	3(1/0)	2(1/0)	1(0/0)

Standard Two Operand Instruction Execution Times

Table 9-4: Two Operand Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xn*SF) (d8,PC,Xn*SF)	xxx.wl	#xxx
ADD.L	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
ADD.L	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
ADDI.L	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
ADDQ.L	#imm,<ea>	1(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
ADDX.L	Dy,Dx	1(0/0)	—	—	—	—	—	—	—
AND.L	<ea>,Dn	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
AND.L	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
ANDI.L	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
ASL.L	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
ASR.L	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
BCHG	Dy,<ea>	2(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
BCHG	#imm,<ea>	2(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—	—
BCLR	Dy,<ea>	2(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
BCLR	#imm,<ea>	2(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—	—
BSET	Dy,<ea>	2(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
BSET	#imm,<ea>	2(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—	—
BTST	Dy,<ea>	2(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
BTST	#imm,<ea>	1(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	—	—	1(0/0)
CMP.L	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
CMPI.L	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
EOR.L	Dy,<ea>	1(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
EORI.L	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
LEA	<ea>,Ax	—	1(0/0)	—	—	1(0/0)	2(0/0)	1(0/0)	—
LSL.L	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
LSR.L	<ea>,Dx	1(0/0)	—	—	—	—	—	—	1(0/0)
MOVEQ	#imm,Dx	—	—	—	—	—	—	—	1(0/0)
MULS.W	<ea>,Dx	9(0/0)	11(1/0)	11(1/0)	11(1/0)	11(1/0)	12(1/0)	11(1/0)	9(0/0)
MULU.W	<ea>,Dx	9(0/0)	11(1/0)	11(1/0)	11(1/0)	11(1/0)	12(1/0)	11(1/0)	9(0/0)
MULS.L	<ea>,Dx	18(0/0)	20(1/0)	20(1/0)	20(1/0)	20(1/0)	—	—	—
MULU.L	<ea>,Dx	18(0/0)	20(1/0)	20(1/0)	20(1/0)	20(1/0)	—	—	—
OR.L	<ea>,Dn	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
OR.L	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
OR.L	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
SUB.L	<ea>,Rx	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	1(0/0)
SUB.L	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
SUBI.L	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
SUBQ.L	#imm,<ea>	1(0/0)	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
SUBX.L	Dy,Dx	1(0/0)	—	—	—	—	—	—	—

Miscellaneous Instruction Execution Times

Table 9-5: Miscellaneous Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn*SF)	xxx.wl	#xxx
LINK.W	Ay,#imm	2(0/1)	—	—	—	—	—	—	—
MOVE.W	CCR,Dx	1(0/0)	—	—	—	—	—	—	—
MOVE.W	<ea>,CCR	1(0/0)	—	—	—	—	—	—	1(0/0)
MOVE.W	SR,Dx	1(0/0)	—	—	—	—	—	—	—
MOVE.W	<ea>,SR	7(0/0)	—	—	—	—	—	—	7(0/0) [1]
MOVEC	Ry,Rc	9(0/1)	—	—	—	—	—	—	—
MOVEM.L	<ea>,&list	—	1+n(n/0)	—	—	1+n(n/0)	—	—	—
MOVEM.L	&list,<ea>	—	1+n(0/n)	—	—	1+n(0/n)	—	—	—
NOP		3(0/0)	—	—	—	—	—	—	—
PEA	<ea>	—	2(0/1)	—	—	2(0/1) [3]	3(0/1) [4]	2(0/1)	—
PULSE		1(0/0)	—	—	—	—	—	—	—
STOP	#imm	—	—	—	—	—	—	—	3(0/0) [2]
TRAP	#imm	—	—	—	—	—	—	—	15(1/2)
TPF		1(0/0)	—	—	—	—	—	—	—
TPF.W	#imm	1(0/0)	—	—	—	—	—	—	—
TPF.L	#imm	1(0/0)	—	—	—	—	—	—	—
UNLK	Ax	2(1/0)	—	—	—	—	—	—	—
WDDATA	<ea>	—	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	3(1/0)
WDEBUG	<ea>	—	5(2/0)	—	—	5(2/0)	—	—	—

n is the number of registers moved by the movem opcode.

[1] If a MOVE.W #imm,SR instruction is executed and imm[13] = 1, the execution time is 1(0/0).

[2] The execution time for STOP is the time required until the processor begins sampling continuously for interrupts.

[3] PEA execution times are the same for (d16,PC)

[4] PEA execution times are the same for (d8,PC,Xn*SF)

Branch Instruction Execution Times

Table 9-6: General Branch Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xi*SF)	xxx.wl	#xxx
BSR		—	—	—	—	3(0/1)	—	—	—
JMP	<ea>	—	3(0/0)	—	—	3(0/0)	4(0/0)	3(0/0)	—
JSR	<ea>	—	3(0/1)	—	—	3(0/1)	4(0/1)	3(0/1)	—
RTE		—	—	8(2/0)	—	—	—	—	—
RTS		—	—	5(1/0)	—	—	—	—	—

Table 9-7: BRA, Bcc Instruction Execution Times

OPCODE	FORWARD TAKEN	FORWARD NOT TAKEN	BACKWARD TAKEN	BACKWARD NOT TAKEN
BRA	2(0/0)	—	2(0/0)	—
Bcc	3(0/0)	1(0/0)	2(0/0)	3(0/0)

Appendix A
Processor Instruction Summary

Overview

Introduction

This appendix provides a quick reference of the ColdFire instructions. Table A-1 lists the ColdFire instructions by mnemonics, followed by the descriptive name

---

Table A-1: ColdFire Instruction Set

MNEMONIC	DESCRIPTION
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
BRA	Branch
BSET	Test Bit and Set
BSR	Branch to Subroutine
BTST	Test Bit
CLR	Clear
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CPUSHL	Cache Push (Line)
EOR	Logical Exclusive-OR
EORI	Logical Exclusive-OR Immediate
EXT, EXTB	Sign Extend
HALT	Halt CPU
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link and Allocate
LSL, LSR	Logical Shift Left and Right
MOVE	Move
MOVEA	Move Address

MNEMONIC	DESCRIPTION
MOVEC	Move Control Register
MOVE from CCR	Move from Condition Code Register
MOVE to CCR	Move to Condition Code Register
MOVEM	Move Multiple Registers
MOVE from SR	Move from the Status Register
MOVE to SR	Move to the Status Register
MOVEQ	Move Quick
MULS	Signed Multiply
MULU	Unsigned Multiply
NEG	Negate
NEGX	Negate with Extend
NOP	No Operation
NOT	Logical Complement
OR	Logical Inclusive-OR
ORI	Logical Inclusive-OR Immediate
PEA	Push Effective Address
PULSE	Generate Processor Status
RTE	Return from Exception
RTS	Return from Subroutine
SUB	Subtract
Scc	Set According to Condition
STOP	Load Status Register and Stop
SUBA	Subtract Address
SUBI	Subtract Immediate
SUBQ	Subtract Quick
SUBX	Subtract with Extend
SWAP	Swap Register Words
TRAP	Trap
TRAPF	No Operation
TST	Test Operand
UNLK	Unlink
WDEBUG	Write Debug Control Register

Appendix B
Multiply and Accumulate (MAC) Instructions

Introduction

Note

Not all ColdFire products will contain the optional MAC unit.

MAC (Multiply and Accumulate)

Operation: $ACC + ((Rw \times Rx)\{<< 1 \mid >> 1\}) \rightarrow ACC$

Assembler Syntax: MAC.<size> Ry.<ul>,Rx.
MAC.<size> Ry.<ul>,Rx.<ul>,<shift>

Attributes: size = (Word, Long)
ul = (Upper, Lower)
shift = (<<, >>)

Description

Multiply two 16- or 32-bit numbers to produce a 32-bit result, then add this product, optionally shifted left or right one bit, to the accumulator (ACC). The result is stored back into the accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified.

MAC Status Register

OMC	S/U	-	-	N	Z	V	C
-	-	0	0	*	*	*	0

OMC—not affected
S/U—not affected
N—set if the most significant bit of the result is set, otherwise cleared
Z—set if the result is zero, otherwise cleared
V—set if an overflow is generated, otherwise unchanged
C—always cleared

Continued on next page

MAC (Multiply and Accumulate), Continued

Processor Condition Codes Not affected

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	RY			0	0	RY	0	0	RX			
-				SZ	SF		0	U/LY	U/LX	-					

Instruction Fields

RY—Source Y field
Specifies a source register operand, where \$0 is D0,..., \$7 is D7, \$8 is A0,..., \$F is A7. Note that bit ordering is 6, 11, 10, 9 (MSB to LSB).

RX—Source X field
Specifies a source register operand, where \$0 is D0,..., \$7 is D7, \$8 is A0,..., \$F is A7. Note that bit ordering is 3, 2, 1, 0 (MSB to LSB).

SZ—Size field
0 = word-sized input operands
1 = long-sized input operands

SF—Scale Factor field
00 = none
01 = product << 1
10 = reserved
11 = product >> 1

U/LY—Source Y Word Select field
This bit determines which 16-bit operand of the source W register is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

U/LX—Source X Word Select field
This bit determines which 16-bit operand of the source X register is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

MACL (Multiply and Accumulate with Register Load)

Operation: ACC + ((Ry × Rx){<< 1 | >> 1}) → ACC
(<ea>{& MASK}) → Ry

Assembler Syntax: MACL.<size> Ry.<ul>, Rx.<ul>,<ea>,Rw
MACL.<size> Ry.<ul>, Rx.<ul>,<shift>,<ea>,Rw
MACL.<size> Ry.<ul>, Rx.<ul>,<shift>,<ea>&,Rw

Attributes: size = (Word, Long)
ul = (Upper, Lower)
shift = (<<, >>)
ea = Effective Address

Description

Multiply two 16- or 32-bit numbers to produce a 32-bit result, then add this product, optionally shifted left or right one bit, to the accumulator (ACC). The result is stored back into the accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified. In parallel with this operation, a 32-bit operand is fetched from the memory location defined by <ea> and loaded into the destination register, Rw. If the mask addressing mode is used, the low-order word of <ea> is ANDed with the mask register.

MAC Status Register

OMC	S/U	-	-	N	Z	V	C
-	-	0	0	*	*	*	0

OMC—not affected
S/U—not affected
N—set if the most significant bit of the result is set, otherwise cleared
Z—set if the result is zero, otherwise cleared
V—set if an overflow is generated, otherwise unchanged
C—always cleared

Processor Condition Codes

Not affected

Continued on next page

MACL (Multiply and Accumulate with Register Load), Continued

Instruction Format

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td colspan="3">RW</td><td>0</td><td>1</td><td>RW</td><td colspan="6"><EA>
MODE | REG</td></tr>
<tr><td colspan="4">RW</td><td>SZ</td><td colspan="2">SF</td><td>0</td><td>U/LY</td><td>U/LX</td><td>MAM</td><td>0</td><td colspan="4">RY</td></tr>
</table>

Instruction Fields

RY—Source Y field
Specifies a source register operand, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7. Note that bit ordering is 15, 14, 13, 12 (MSB to LSB).

RX—Source X field
Specifies a source register operand, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7. Note that bit ordering is 3, 2, 1, 0 (MSB to LSB).

RW—Destination field
Specifies a destination register operand, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7. Note that bit ordering is 6, 11, 10, 9 (MSB to LSB).

<*ea*>—Effective Address of Memory Operand field

ADDRESSING MODE	MODE	REGISTER
Dn	-	-
An	-	-
(An)	010	reg.num:An
(An)+	011	reg.num:An
-(An)	100	reg.num:An
(d16,An)	101	reg.num:An
(d8,An,Xn)	-	-

ADDRESSING MODE	MODE	REGISTER
(xxx).W	-	-
(xxx).L	-	-
#<data>	-	-
(d16,PC)	-	-
(d8,PC,Xn)	-	-

SZ—Size field
0 = word-sized input operands
1 = long-sized input operands

SF—Scale Factor field
00 = none
01 = product << 1
10 = reserved
11 = product >> 1

Continued on next page

MACL (Multiply and Accumulate with Register Load), Continued

Instruction Fields (Continued)

U/Ly—Source Y Word Select field
This bit determines which 16-bit operand of the source Y register is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

U/Lx—Source X Word Select field
This bit determines which 16-bit operand of the source X register is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

MAM—Mask Addressing Mode Modifier
This bit determines if the mask addressing mode should be used.
0 = normal addressing mode
1 = mask addressing mode

MSAC (Multiply and Subtract)

Operation:	ACC - ((Ry × Rx){<< 1 | >> 1}) → ACC
Assembler Syntax:	MSAC.<size> Ry. ,Rx. MSAC.<size> Ry. ,Rx. ,<shift>
Attributes:	size = (Word, Long) ul = (Upper, Lower) shift = (<<, >>)

Description

Multiply two 16- or 32-bit numbers to produce a 32-bit result, then subtract this product, optionally shifted left or right one bit, from the accumulator (ACC). The result is stored back into the accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified.

Continued on next page

MSAC (Multiply and Subtract), Continued

MAC Status Register

OMC	S/U	-	-	N	Z	V	C
-	-	0	0	*	*	*	0

OMC—not affected
S/U—not affected
N—set if the most significant bit of the result is set, otherwise cleared
Z—set if the result is zero, otherwise cleared
V—set if an overflow is generated, otherwise unchanged
C—always cleared

Processor Condition Codes

Not affected

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	RY			0	0	RY	0	0	RX			
-				SZ	SF		1	U/LY	U/LX						

Instruction Fields

RY—Operand Y field
Specifies a source register operand, where \$0 is D0,..., \$7 is D7, \$8 is A0,..., \$F is A7. Note that bit ordering is 6, 11, 10, 9 (MSB to LSB).

RX—Operand X field
Specifies a source register operand, where \$0 is D0,..., \$7 is D7, \$8 is A0,..., \$F is A7. Note that bit ordering is 3, 2, 1, 0 (MSB to LSB).

SZ—Size field
0 = word-sized input operands
1 = long-sized input operands

SF—Scale Factor field
00 = none
01 = product << 1
10 = reserved
11 = product >> 1

Continued on next page

Instruction Fields (Continued)

U/LY—Source Y Word Select field
This bit determines which 16-bit operand of the source Yregister is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

U/LX—Source X Word Select field
This bit determines which 16-bit operand of the source X register is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

MSACL (Multiply and Subtract with Register Load)

Operation: ACC - ((Ry × Rx){<< 1 | >> 1}) → ACC
(<ea>{& MASK}) → Ry

Assembler Syntax:
MSACL.<size> Ry.<ul>,Rx.<ul>,<ea>,Rw
MSACL.<size> Ry.<ul>,Rx.<ul>,<shift>,<ea>,Rw
MSACL.<size> Ry.<ul>,Rx.<ul>,<shift>,<ea>&,Rw

Attributes:
size = (Word, Long)
ul = (Upper, Lower)
shift = (<<, >>)
ea = Effective Address

Description

Multiply two 16- or 32-bit numbers to produce a 32-bit result, then subtract this product, optionally shifted left or right one bit, from the accumulator (ACC). The result is stored back into the accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified. In parallel with this operation, a 32-bit operand is fetched from the memory location defined by <ea> and loaded into the destination register, Ry. If the mask addressing mode is used, the low-order word of <ea> is ANDed with the mask register.

Continued on next page

MSACL (Multiply and Subtract with Register Load), Continued

MAC Status Register

OMC	S/U	-	-	N	Z	V	C
-	-	0	0	*	*	*	0

OMC—not affected
S/U—not affected
N—set if the most significant bit of the result is set, otherwise cleared
Z—set if the result is zero, otherwise cleared
V—set if an overflow is generated, otherwise unchanged
C—always cleared

Processor Condition Codes

Not affected

Instruction Format

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">0</td><td colspan="3" rowspan="2">RY</td><td rowspan="2">0</td><td rowspan="2">1</td><td rowspan="2">RY</td><td colspan="6"><EA></td></tr>
<tr><td colspan="3">MODE</td><td colspan="3">REG</td></tr>
<tr><td colspan="4">RW</td><td>SZ</td><td colspan="2">SF</td><td>1</td><td>U/LY</td><td>U/LX</td><td>MAM</td><td>0</td><td colspan="4">RX</td></tr>
</table>

Instruction Fields

RY—Source Y field
Specifies a source register operand, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7. Note that bit ordering is 15, 14, 13, 12 (MSB to LSB).

RX—Source X field
Specifies a source register operand, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7. Note that bit ordering is 3, 2, 1, 0 (MSB to LSB).

RW—Destination field
Specifies a destination register operand, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7. Note that bit ordering is 6, 11, 10, 9 (MSB to LSB).

<*ea*>—Effective Address of Memory Operand field

Continued on next page

Instruction Fields
(Continued)

ADDRESSING MODE	MODE	REGISTER		ADDRESSING MODE	MODE	REGISTER
Dn	-	-		(xxx).W	-	-
An	-	-		(xxx).L	-	-
(An)	010	reg.num:An		#<data>	-	-
(An)+	011	reg.num:An				
-(An)	100	reg.num:An				
(d16,An)	101	reg.num:An		(d16,PC)	-	-
(d8,An,Xn)	-	-		(d8,PC,Xn)	-	-

SZ—Size field
0 = word-sized input operands
1 = long-sized input operands

SF—Scale Factor field
00 = none
01 = product << 1
10 = reserved
11 = product >> 1

U/LY—Source Y Word Select field
This bit determines which 16-bit operand of the source Y register is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

U/LX—Source X Word Select field
This bit determines which 16-bit operand of the source X register is used in the operation for word-sized operations only.
0 = lower word
1 = upper word

MAM—Mask Addressing Mode Modifier
This bit determines if the mask addressing mode should be used.
0 = normal addressing mode
1 = mask addressing mode

New Register Instructions

This section describes the new register instructions. A detailed discussion of each instruction description is arranged in alphabetical order by instruction mnemonic.

MOVE from ACC (Move from Accumulator)

Operation: ACC → Rn
Assembler Syntax: MOVE.<size> ACC, Rn
Attributes: size = Long

Description

Move a 32-bit value from the accumulator (ACC) to a register. The size of the operation must be specified as long.

MAC Status Register

Not affected

Processor Condition Codes

Not affected

Instruction Format

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

Instruction Fields

Rn[3:0] specifies the destination register, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7.

MOVE from MACSR (Move from MAC Status Register)

Operation:	MACSR → Rn[7:0] 0 → Rn[31:8]
Assembler Syntax:	MOVE.<size> MACSR, Rn
Attributes:	size = Long

Description

Move the contents of the MAC status register (MACSR), zero-extended to long size, into a general-purpose register, Rx. The size of the operation must be specified as long.

MAC Status Register

Not affected

Processor Condition Codes

Not affected

Instruction Format

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

Instruction Fields

Rn[3:0] specifies the destination register, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7.

MOVE from MASK

Operation:	MASK → Rn[15:0] 0xFFFF → Rn[31:16]
Assembler Syntax:	MOVE.<size> MASK, Rn
Attributes:	size = Long

Continued on next page

MOVE from MASK, Continued

Description

Move a 32-bit value from the mask register (MASK), one-extended to long size, to a register. The size of the operation must be specified as long.

MAC Status Register

Not affected

Processor Condition Codes

Not affected

Instruction Format

<table>
<tr><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>0</th></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="2">RN</td></tr>
</table>

Instruction Fields

Rn[3:0] specifies the destination register, where $0 is D0,..., $7 is D7, $8 is A0,..., $F is A7.

MOVE to ACC (Move to Accumulator)

Operation: Source → ACC
Assembler
Syntax: MOVE.<size> <ea>, ACC
Attributes: size = Long

Description

Move a 32-bit value from a register or an immediate value into the accumulator (ACC). The size of the operation must be specified as long.

Continued on next page

MOVE to ACC (Move to Accumulator), Continued

MAC Status Register

OMC	S/U	-	-	N	Z	V	C
-	-	0	0	*	*	0	0

OMC—not affected
S/U—not affected
N—set if the most significant bit of the result is set, otherwise cleared
Z—set if the result is zero, otherwise cleared
V—always cleared
C—always cleared

Processor Condition Codes

Not affected

Instruction Format

15	14	13	12	11	10	9	8	7	6	5		0
										<EA>		
1	0	1	0	0	0	0	1	0	0	MODE		REG

Instruction Fields

<ea>—Effective Address

ADDRESSING MODE	MODE	REGISTER
Dn	000	reg.num:Dn
An	001	reg.num:An
(An)	-	-
(An)+	-	-
-(An)	-	-
(d16,An)	-	-
(d8,An,Xn)	-	-

ADDRESSING MODE	MODE	REGISTER
(xxx).W	-	-
(xxx).L	-	-
#<data>	111	100
(d16,PC)	-	-
(d8,PC,Xn)	-	-

MOVE to CCR (Move to Condition Code Register)

Operation: MACSR[4:0] → CCR[4:0]
Assembler Syntax: MOVE.<size> MACSR,CCR
Attributes: size = Long

Description

Move the indicator flags of the MAC status register (MACSR) into the processor's condition code cegister (CCR). The size of the operation must be specified as long.

MAC Status Register

Not affected

Processor Condition Codes

X	N	Z	V	C
-	*	*	*	*

X—not affected
N—set to the value of MACSR bit 3, N
Z—set to the value of MACSR bit 2, Z
V—set to the value of MACSR bit 1, V
C—set to the value of MACSR bit 0, C

Instruction Format

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

MOVE to MACSR (Move to MAC Status Register)

Operation: Source → MACSR
Assembler
Syntax: MOVE.<size> <ea>, MACSR
Attributes: size = Long

Description

Move the low-order byte of a 32-bit value from a register or an immediate value into the MAC status register (MACSR). The size of the operation must be specified as long.

MAC Status Register

OMC	S/U	-	-	N	Z	V	C
*	*	0	0	*	*	*	0

OMC—set to the value of bit 7 of the source operand
S/U—set to the value of bit 6 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—always cleared

Processor Condition Codes

Not affected

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	0
										<EA>	
1	0	1	0	1	0	0	1	0	0	MODE	REG

Continued on next page

MOVE to MACSR (Move to MAC Status Register), Continued

Instruction Fields <*ea*>—Effective Address

ADDRESSING MODE	MODE	REGISTER
Dn	000	reg.num:Dn
An	001	reg.num:An
(An)	-	-
(An)+	-	-
-(An)	-	-
(d16,An)	-	-
(d8,An,Xn)	-	-

ADDRESSING MODE	MODE	REGISTER
(xxx).W	-	-
(xxx).L	-	-
#<data>	111	100
(d16,PC)	-	-
(d8,PC,Xn)	-	-

MOVE to MASK (Move to Modulus Register)

Operation: Source → MASK
Assembler
Syntax: MOVE.<size> <ea>, MASK
Attributes: size = Long

Description Move the low-order word of a 32-bit value from a register or an immediate value into the mask register (MASK). The size of the operation must be specified as long.

MAC Status Register Not affected

Processor Condition Codes Not affected

Instruction Format

15	14	13	12	11	10	9	8	7	6	5	0
1	0	1	0	1	1	0	1	0	0	<EA> MODE	 REG

Continued on next page

MOVE to MASK (Move to Modulus Register), Continued

Instruction Fields <*ea*>—Effective Address

ADDRESSING MODE	MODE	REGISTER		ADDRESSING MODE	MODE	REGISTER
Dn	000	reg.num:Dn		(xxx).W	-	-
An	001	reg.num:An		(xxx).L	-	-
(An)	-	-		#<data>	111	100
(An)+	-	-				
-(An)	-	-				
(d16,An)	-	-		(d16,PC)	-	-
(d8,An,Xn)	-	-		(d8,PC,Xn)	-	-

Operation Code Map

All MAC instructions are mapped into line A, i.e. bits 15-12 of the instruction are 1010 ($A).

---

1. MAC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	RY			0	0	RY	0	0	RX			
-				SZ	SF		0	U/LY	U/LX	-					

2. MSAC

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	RY			0	0	RY	0	0	RX			
-				SZ	SF		1	U/LY	U/LX	-					

3. MACL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	RY			0	1	RY	<EA> MODE			REG		
RW				SZ	SF		0	U/LY	U/LX	MAM	0	RX			

4. MSACL

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	RY			0	1	RY	<EA> MODE			REG		
RW				SZ	SF		1	U/LY	U/LX	MAM	0	RX			

5. MOVE to ACC

15	14	13	12	11	10	9	8	7	6	5					0
1	0	1	0	0	0	0	1	0	0	<EA> MODE			REG		

6. MOVE to MACSR

15	14	13	12	11	10	9	8	7	6	5					0
1	0	1	0	1	0	0	1	0	0	<EA> MODE			REG		

7. MOVE to MASK

15	14	13	12	11	10	9	8	7	6	5	0
1	0	1	0	1	1	0	1	0	0	<EA> MODE	REG

8. MOVE from ACC

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

9. MOVE from MACSR

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

10. MOVE from MASK

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

11. MOVE to CCR

15	14	13	12	11	10	9	8	7	6	5	0
1	0	1	0	0	0	0	1	0	0	<EA> MODE	REG

Table B-1. MAC Instruction Execution Times

OPCODE	<EA>	EFFECTIVE ADDRESS							
		RN	(AN)	(AN)+	-(AN)	(D16, AN) (D16, PC)	(D8, AN, XN*SF) (D8, PC, XN*SF)	XXX.WL	#XXX
mac.w	RY, RX	1(0/0)	-	-	-	-	-	-	-
mac.l	RY, RX	3(0/0)	-	-	-	-	-	-	-
msac.w	RY, RX	1(0/0)	-	-	-	-	-	-	-
msac.l	RY, RX	3(0/0)	-	-	-	-	-	-	-
macl.w	RY, RX, <ea>, RW	-	2(1/0)	2(1/0)	2(1/0)	2(1/0)^	-	-	-
macl.l	RY, RX, <ea>, RW	-	4(1/0)	4(1/0)	4(1/0)	4(1.0)^	-	-	-
msacl.w	RY, RX, <ea>, RW	-	2(1/0)	2(1/0)	2(1/0)	2(1/0)^	-	-	-
msacl.l	RY, RX, <ea>, RW	-	4(1/0)	4(1/0)	4(1/0)	4(1/0)^	-	-	-

Note: ^Effective address of (d16, PC) not supported

B

Products powered by Motorola are fast becoming a way of life

POWERED BY MOTOROLA

How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution;
P.O. Box 20912; Phoenix, Arizona 85036, 1-800-441-2447 or 602-303-5454

MFAX: RMFAX0@email.sps.mot.com–TOUCHTONE 602-244-6609
INTERNET: http://Design–NET.com

JAPAN: Nippon Motorola Ltd.; 4-32-1, Tatsumi–SPD–JLDC, 6F Salbu–Butsuryu–Center,
3–14–2 Tatsumi Koto–Ku, Tokyo 135, Japan. 03–81–3521–8315

ASIA-PACIFIC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,
51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852–26629298