MOTOROLA

MCF5200

ColdFire™ Family

Programmer's

Reference Manual

68K FAX-IT

Documentation Comments

FAX 512-891-8593—Documentation Comments Only

The Motorola High-Performance Embedded Systems Technical Communications Department provides a fax number for you to submit any questions or comments about this document or how to order other documents. We welcome your suggestions for improving our documentation. Please do not fax technical questions.

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.

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

For Internet Access:

Telnet: pirs.aus.sps.mot.com (Login: pirs)
WWW: http: / / pirs.aus.sps.mot.com/aesop/hmpg.html
Query By Email: aesop_query@pirs.aus.sps.mot.com
(Type ‘‘HELP’’ in text body.)

For Dial-Up:

Phone: +1-512-891-3650
Phone (US or Canada): 1-800-843-3451
Connection Settings: N/8/1/F
Data Rate: < 14,400 bps
Terminal Emulation: VT100
Login: pirs

For AESOP Questions:

FAX: +1-512-891-8775
EMAIL: aesop_sysop@pirs.aus.sps.mot.com

For Hotline Questions:

FAX (US or Canada): 1-800-248-8567
EMAIL: aesop_support@pirs.aus.sps.mot.com

Applications and Technical Information

For questions or comments pertaining to technical information, questions, and applications, please contact one of the following sales offices nearest you.

— Sales Offices —

Field Applications Engineering 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, Rosevllle	(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 Whltely	(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

The *MCF5200 ColdFire Family Programmer's Reference Manual* describes the programming, capabilities, and operation of the ColdFire Family processors, while the *MC68000 Family Programmer's Reference Manual* provides instruction details for the EC000 core.

TRADEMARKS

All trademarks reside with their respective owners.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Section 4
Integer Instructions

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section 7

Exception Processing

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (Continued)

SECTION 1
INTRODUCTION

This manual contains detailed information about software instructions used by the ColdFire™ 5200 microprocessors.

The ColdFire Family programming model consists of two register groups: user and 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 following paragraphs provide a brief description of the registers in the user and supervisor models as well as the data organization in the registers.

1.1 INTEGER UNIT USER PROGRAMMING MODEL

Figure 1-1 illustrates the integer portion of the user programming model. It consists of the following registers:

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

1.1.1 Data Registers (D7 – D0)

These registers are for bit, byte (8 bits), word (16 bits), and long-word (32 bits) operations. They can also be used as index registers.

1.1.2 Address Registers (A7 – A0)

These registers serve as software stack pointers, index registers, or base address registers. The base address registers can be used for word and long-word operations. Register A7 functions as a hardware stack pointer during stacking for subroutine calls and exception handling.

1.1.3 Program Counter (PC)

The program counter (PC) contains the address of the instruction currently executing. During instruction execution and exception processing, the processor automatically increments the contents or places a new value in the PC. For some addressing modes, the PC can serve as a pointer for PC relative addressing.

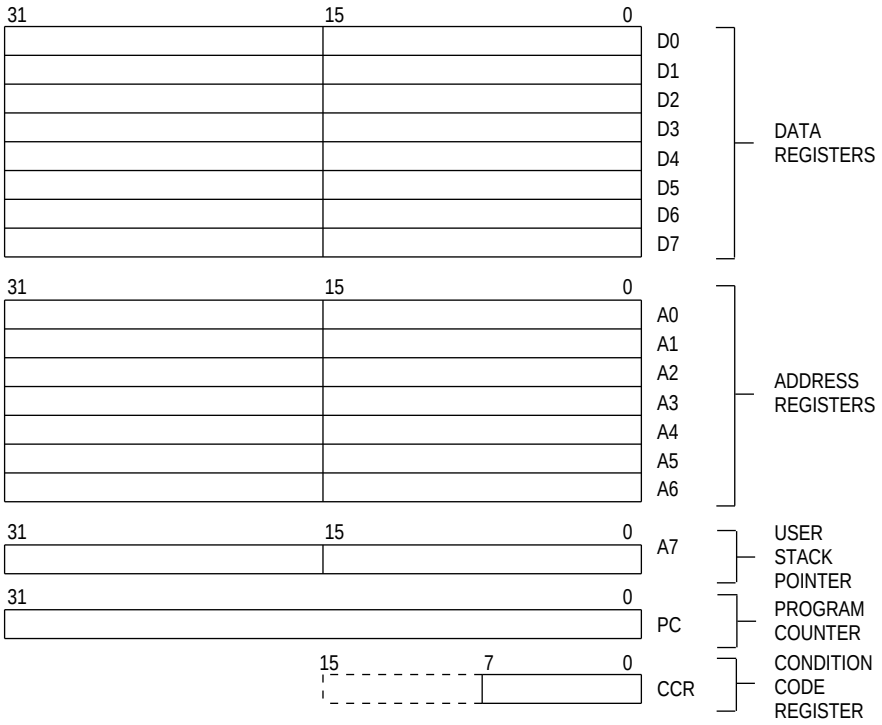


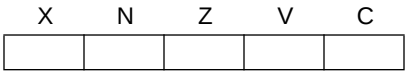
Figure 1-1. ColdFire Family User Programming Model

1.1.4 Condition Code Register (CCR)

Consisting of 5 bits, the condition code register (CCR)—the status register's lower byte—is the only portion of the (SR) available in the user mode. Many integer instructions affect the CCR, indicating the instruction's result. Program and system control instructions also use certain combinations of these bits to control program and system flow. The condition codes meet two criteria: consistency across instructions, uses, and instances and meaningful results with no change unless it provides useful information.

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

The first 4 bits represent a condition of the result generated by an operation. The fifth bit or the extend bit (X-bit) is an operand for multiprecision computations. In the instruction set definitions, the CCR is illustrated as follows:



X—Extend
Set to the value of the C-bit for arithmetic operations; otherwise not affected or set to a specified result

N—Negative
Set if the most significant bit of the result is set; otherwise clear

Z—Zero
Set if the result equals zero; otherwise clear

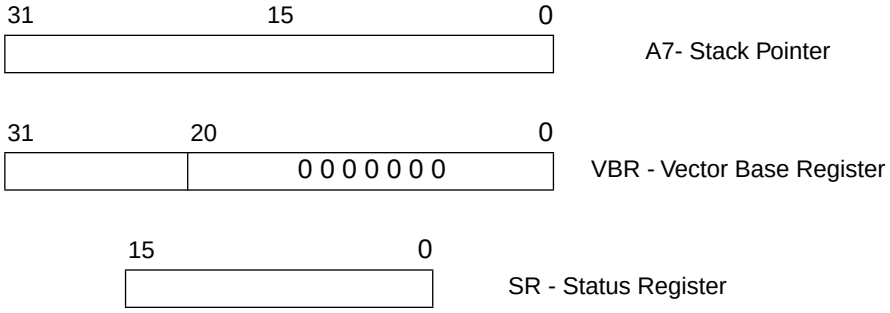
V—Overflow
Set if an arithmetic overflow occurs implying that the result cannot be represented in the operand size; otherwise clear

C—Carry
Set if a carryout of the most significant bit of the operand occurs for an addition, or if a borrow occurs in a subtraction; otherwise clear

1.2 SUPERVISOR PROGRAMMING MODEL

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 register listed in Table 1-1.

Table 1-1. Supervisor Registers



1.2.1 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.

1.2.2 Status Register

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).

.

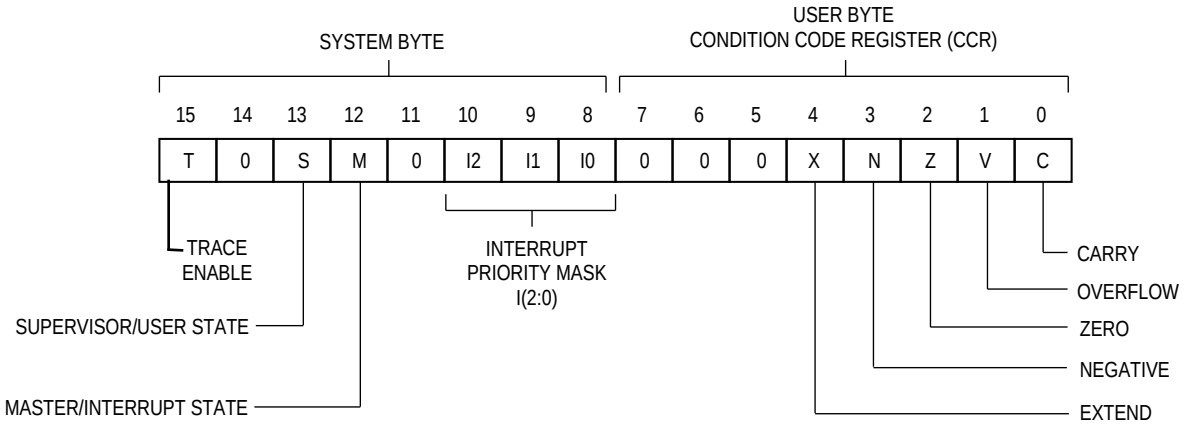


Figure 1-2. Status Register

1.2.3 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.

1.3 INTEGER DATA FORMATS

The operand data formats are supported by the integer unit, as listed in Table 1-2. Integer unit 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.

Table 1-2. Integer Data Formats

Operand Data Format	Size
Bit	1 Bit
Byte Integer	8 Bits
Word Integer	16 Bits
Long-Word Integer	32 Bits

1.4 ORGANIZATION OF DATA IN REGISTERS

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

1.4.1 Organization of Integer Data Formats in Registers

Each integer data register is 32 bits wide. Byte and word operands occupy the lower 8- and 16-bit portions of integer data registers, respectively. Long-word 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 remaining high-order portion does not change and must be sign-extended.The address of the least significant bit (LSB) of a long-word integer is zero, and the most significant bit (MSB) is 31.

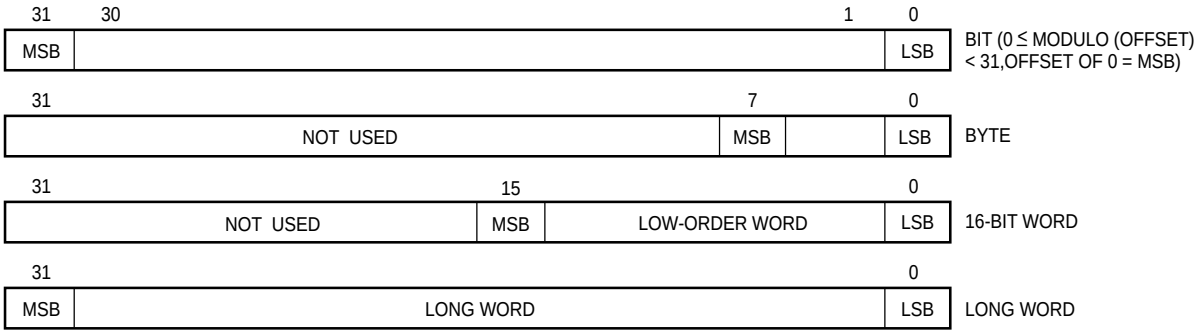


Figure 1-3. Organization of Integer Data Formats in Data Registers

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 long-word 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.

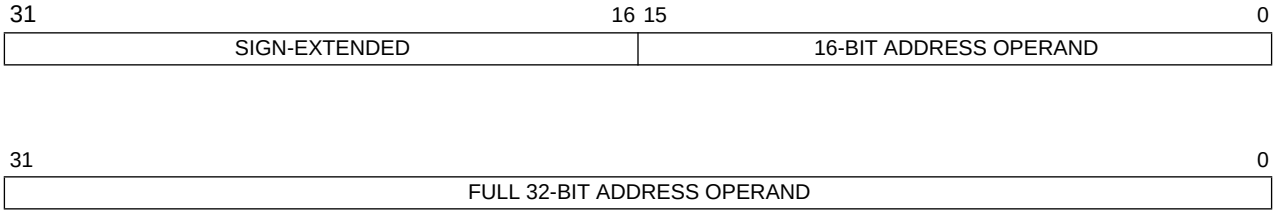


Figure 1-4. Organization of Integer Data Formats in Address Registers

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.

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 may be added depending on the implementation.

1.4.2 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 long-word data item corresponds to the address of the highest order words 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.

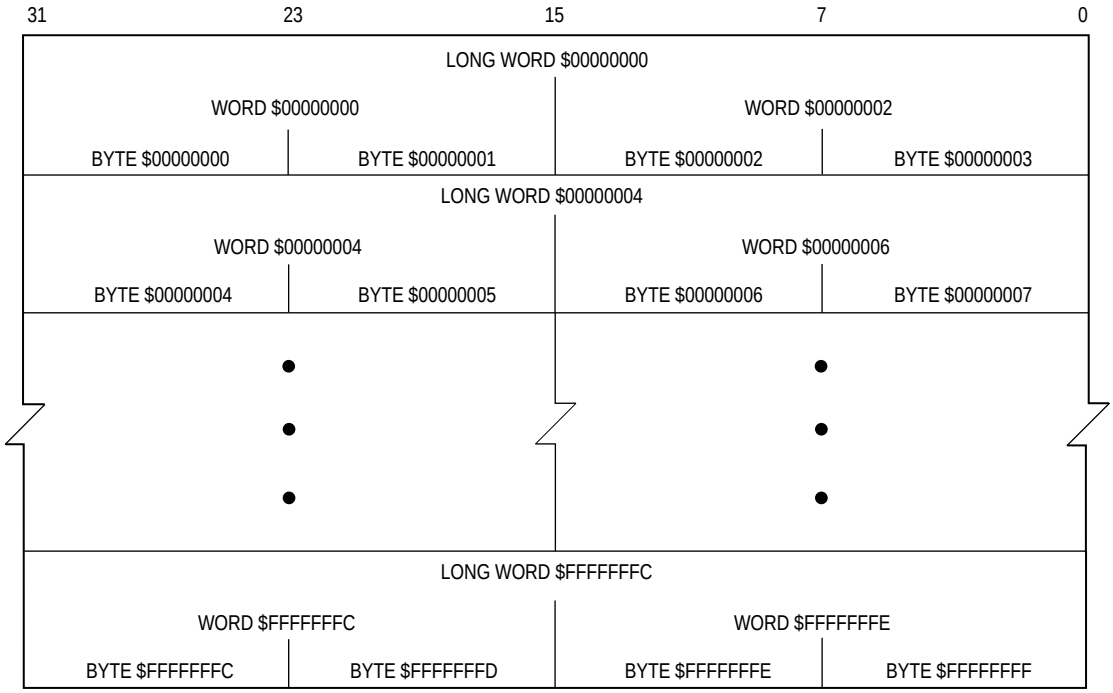


Figure 1-5. Memory Operand Addressing

Figure 1-6 illustrates the organization of IU data formats in memory. 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.

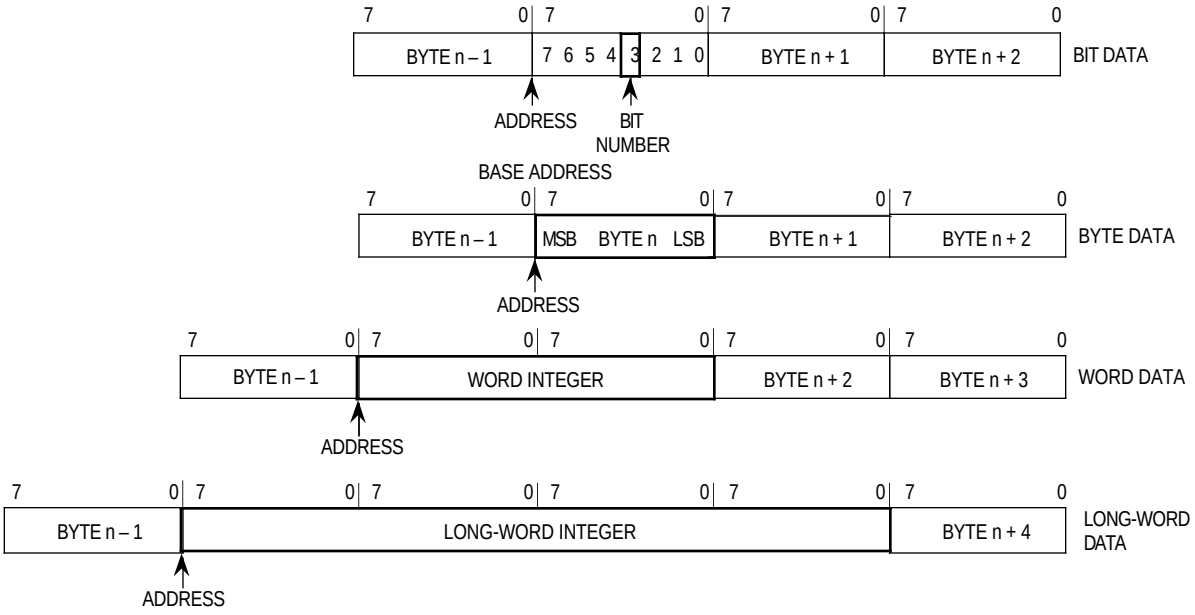


Figure 1-6. Memory Organization for Integer Operands

SECTION 2
ADDRESSING CAPABILITIES

Most operations compute a source operand and destination operand and store the result in the destination location. Single-operand operations compute a destination operand and 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 can be classified as data references.

2.1 INSTRUCTION FORMAT

ColdFire Family instructions consist of 1 to 3 words. Figure 2-1 illustrates the general composition of an instruction. The first word of the instruction, called the simple effective address operation word, specifies the length of the instruction, the effective addressing mode, and the operation to be performed. The remaining words further specify the instruction and operands. These words can be conditional predicates, 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, or argument counts. The ColdFire architecture instruction word length is limited to 3 sizes: 16, 32, or 48 bits.

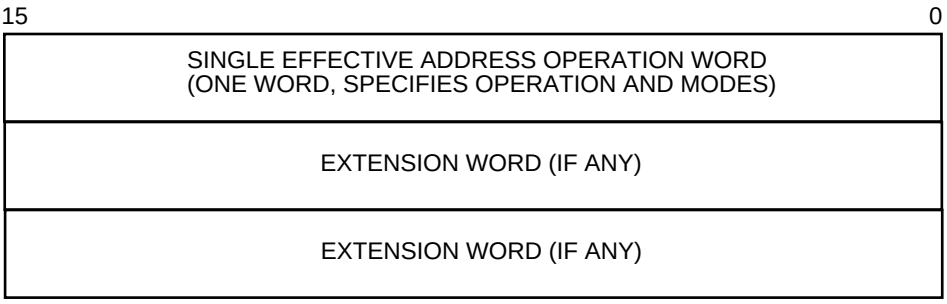
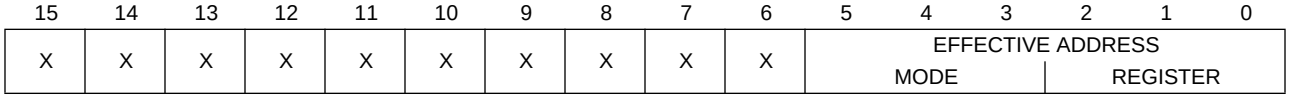


Figure 2-1. Instruction Word General Format

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 register specification (the instruction's register field holds the register's number), by effective address (the instruction's effective address field contains addressing mode information), or by implicit reference (the definition of the instruction implies the use of specific registers).

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;Table 2-1 lists the field definitions.

SINGLE EFFECTIVE ADDRESS OPERATION WORD FORMAT



EXTENSION WORD FORMAT

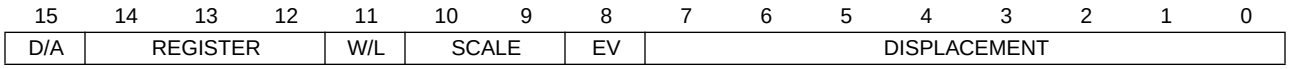


Figure 2-2. Instruction Word Specification Formats

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
	Extension Word Valid 0 = Extension Word Valid 1 = Address Error Exception

2.2 EFFECTIVE ADDRESSING MODES

Besides the operation code that specifies the function to be performed, an instruction defines the location of every operand for the function. Instructions specify an operand location 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;

or (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.

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.

2.2.1 Data Register Direct Mode

In the data register direct mode, the effective address field specifies the data register containing the operand.

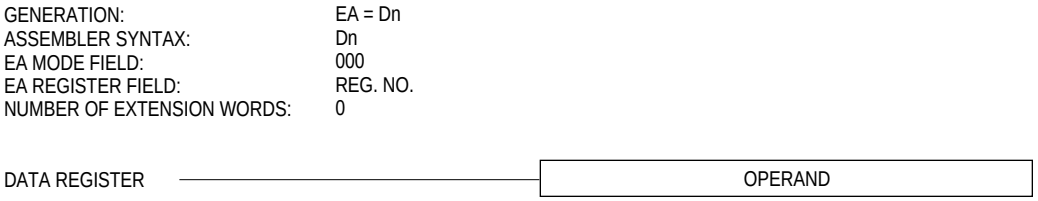


Figure 2-3. Data Register Direct Mode

2.2.2 Address Register Direct Mode

In the address register direct mode, the effective address field specifies the address register containing the operand.

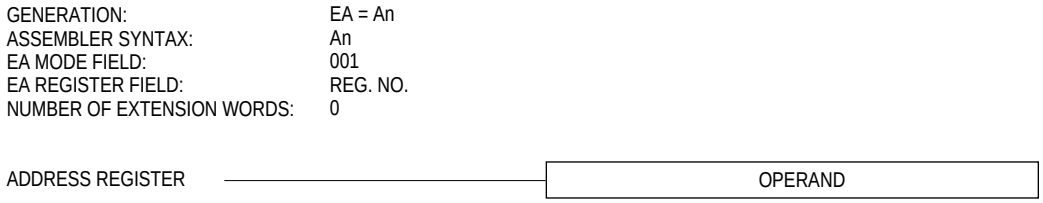


Figure 2-4. Address Register Direct Mode

2.2.3 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.

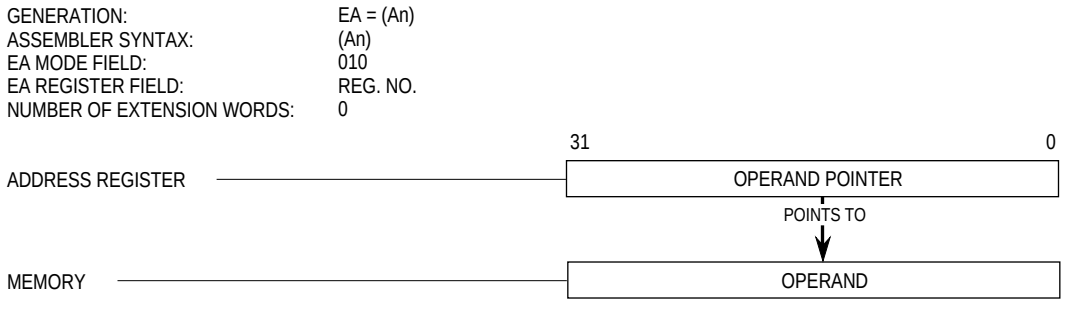


Figure 2-5. Address Register Indirect Mode

2.2.4 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 long word, respectively). Note that the stack pointer (A7) is treated exactly like any other address register.

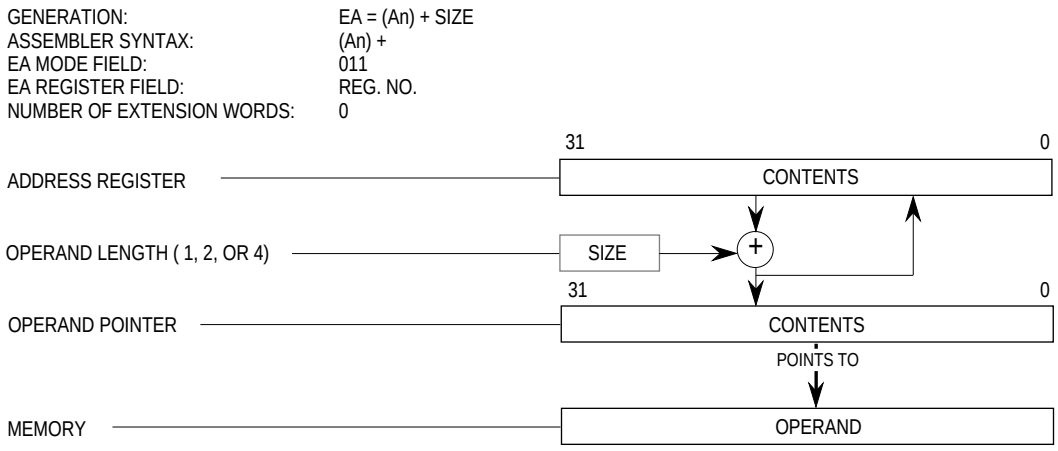


Figure 2-6. Address Register Indirect with Postincrement Mode

2.2.5 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 long word, respectively). Note that the stack pointer (A7) is treated just like the other address registers.

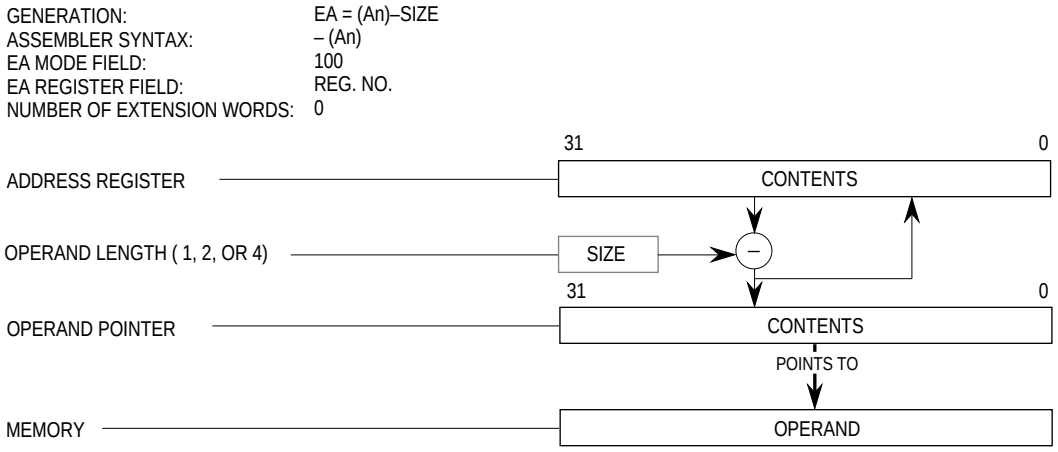


Figure 2-7. Address Register Indirect with Predecrement Mode

2.2.6 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.

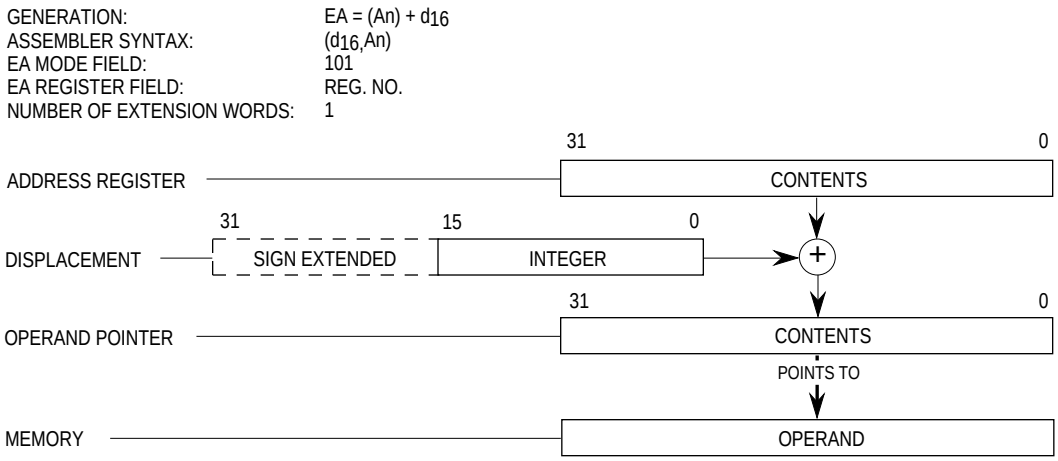


Figure 2-8. Address Register Indirect with Displacement Mode

2.2.7 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; the sign-extended displacement value in the extension word's low-order 8 bits; and the index register's sign-extended contents (possibly scaled). Users must specify the address register, the displacement, and the index register in this mode.

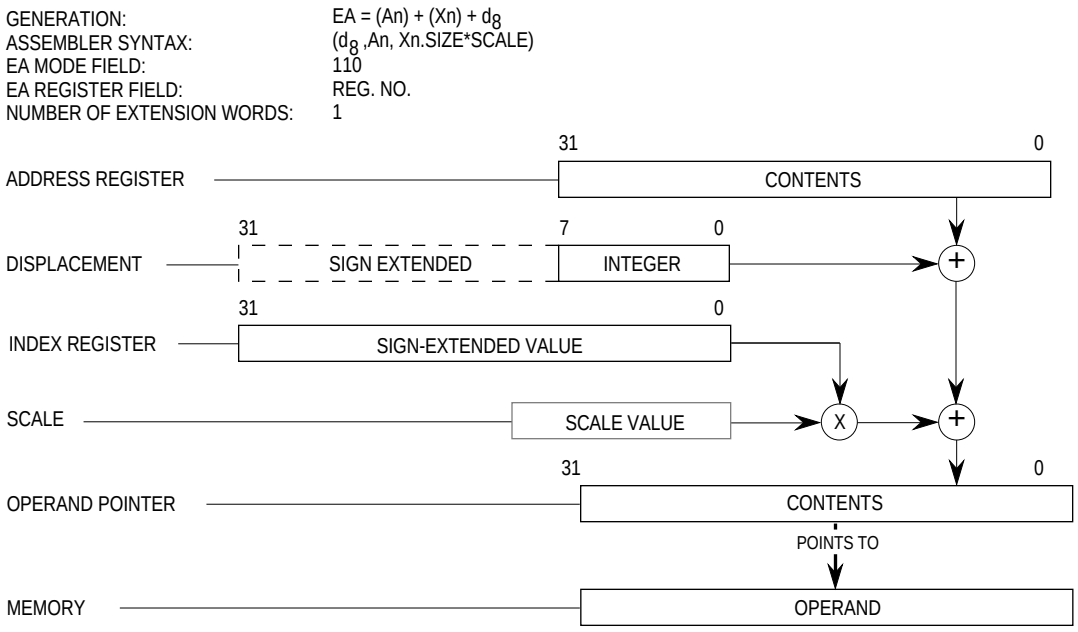


Figure 2-9. Address Register Indirect with Index (8-Bit Displacement) Mode

2.2.8 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 program reference allowed only for reads.

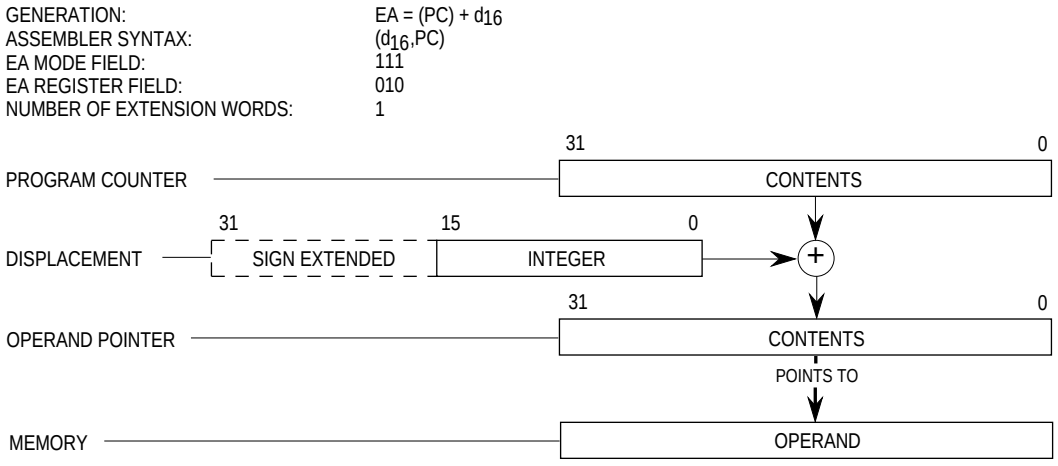


Figure 2-10. Program Counter Indirect with Displacement Mode

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

This mode is similar to the mode described in **2.2.7 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, the sign-extended displacement integer in the extension word's lower 8 bits, and the sized, scaled, and sign-extended index operand. The value in the PC is the address of the extension word. This is a program reference allowed only for reads. Users must include the displacement, the PC, and the index register when specifying this addressing mode.

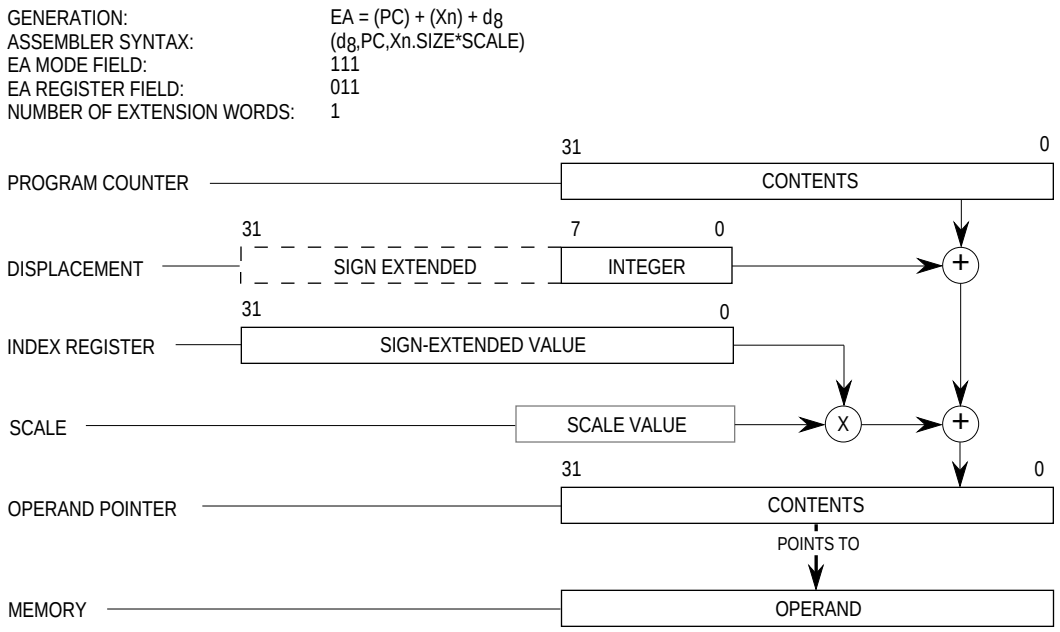


Figure 2-11. Program Counter Indirect with Index (8-Bit Displacement) Mode

2.2.10 Absolute Short-Addressing Mode

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

2.2.11 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.

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

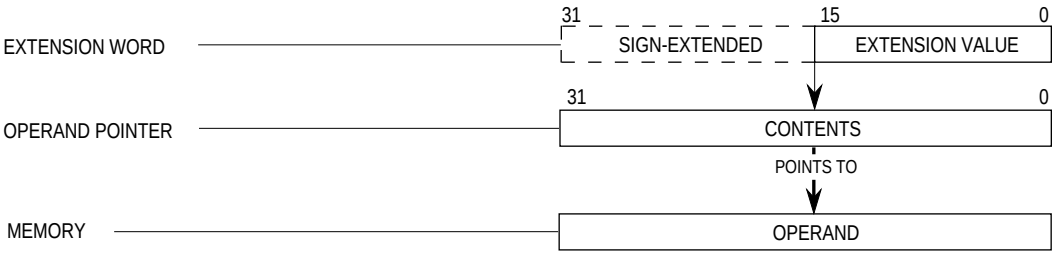


Figure 2-12. Absolute Short Addressing Mode

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

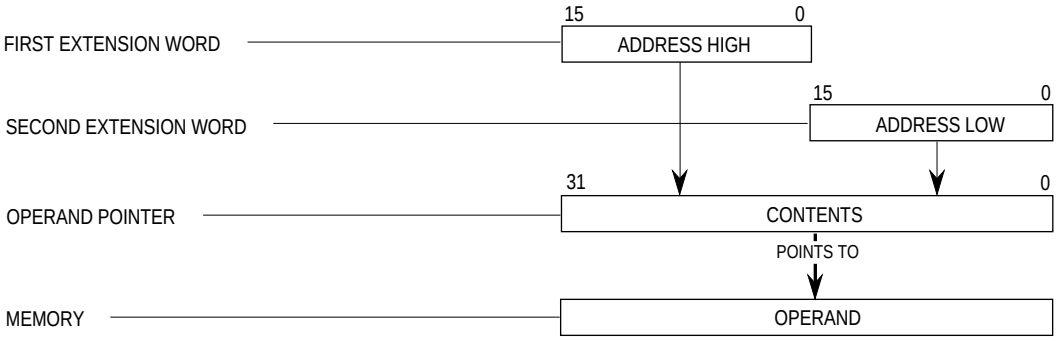


Figure 2-13. Absolute Long Addressing Mode

2.2.12 Immediate Data

In this addressing mode, the operand is in 1 or 2 extension words. Table 2-3 lists the location of the operand within the instruction word format. The immediate data format is as follows:

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.

GENERATION:	OPERAND GIVEN
ASSEMBLER SYNTAX:	#<xxx>
EA MODE FIELD:	111
EA REGISTER FIELD:	100
NUMBER OF EXTENSION WORDS:	1,2,

Figure 2-14. Immediate Data Addressing Mode

2.2.13 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.

These categories sometimes combine to form new categories that are more restrictive. Two combined classifications are alterable memory (addressing modes that are both alterable and memory addresses) and data alterable (addressing modes that are both alterable and data). Table 2-4 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.	X	—	—	X
Address	An	001	reg. no.	—	—	—	X
Register Indirect							
Address	(An)	010	reg. no.	X	X	X	X
Address with Postincrement	(An)+	011	reg. no.	X	X	—	X
Address with Predecrement	–(An)	100	reg. no.	X	X	—	X
Address with Displacement	(d_{16},An)	101	reg. no.	X	X	X	X
Address Register Indirect with Index							
8-Bit Displacement	(d_8,An,Xn)	110	reg. no.	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,Xn)	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	—	—

2.3 STACK

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, and PEA. To maximize performance, A7 must be long-word 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.

Users can employ other address registers to implement other stacks using the address register indirect with postincrement and predecrement addressing modes. With an address register, users can implement a stack that fills either from high memory to low memory or vice versa. Regarding the following important considerations, users 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 long-word items mix in these stacks

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

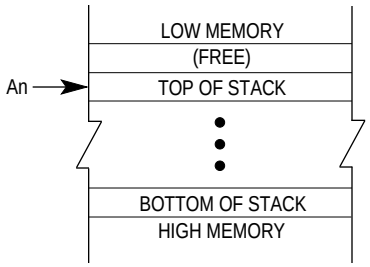


Figure 2-15. Stack Growth from High Memory to Low Memory

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

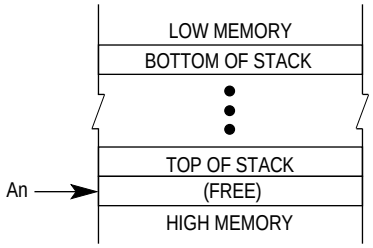


Figure 2-16. Stack Growth from Low Memory to High Memory

SECTION 3
INSTRUCTION SET SUMMARY

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.

3.1 INSTRUCTION SUMMARY

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.

Table 3-1. Notational Conventions

Single- And Double Operand Operations	
+	Arithmetic addition or postincrement indicator.
–	Arithmetic subtraction or predecrement indicator.
×	Arithmetic multiplication.
÷	Arithmetic division or conjunction symbol.
~	Invert; operand is logically complemented.
Λ	Logical AND
V	Logical OR
⊕	Logical exclusive OR
→	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 → SP; PC → (SP); SP - 2 → SP; SR → (SP); SP - 2 → SP; FORMAT → (SP); (Vector) → 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)
Ax, Ay	Source and destination address registers, respectively.
Dc	Data register D7–D0, used during compare.
Dh, Dl	Data register’s high- or low-order 32 bits of product.
Dn	Any Data Register n (example: D5 is data register 5)
Dr, Dq	Data register’s remainder or quotient of divide.
Du	Data register D7–D0, used during update.
Dx, Dy	Source and destination data registers, respectively.
MRn	Any Memory Register n.
Rn	Any Address or Data Register
Rx, Ry	Any source and destination registers, respectively.
Xn	Index Register

Table 3-1. Notational Conventions (Continued)

Data Format And Type	
<fmt>	Operand Data Format: Byte (B), Word (W), Long (L)
B, W, L	Specifies a signed integer data type (twos complement) of byte, word, or long word.
Subfields and Qualifiers	
#<xxx> or #<data>	Immediate data following the instruction word(s).
( )	Identifies an indirect address in a register.
[ ]	Identifies an indirect address in memory.
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
SIZE	The index register's size (W for word, L for long word).
Register Names	
CCR	Condition Code Register (lower byte of status register)
IC, DC, IC/DC	Instruction, Data, or Both Caches
PC	Program Counter
OC	An or Dn Register
SR	Status Register

Table 3-1. Notational Conventions (Concluded)

Register Codes	
*	General Case
C	Carry Bit in CCR
cc	Condition Codes from CCR
FC	Function Code
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>	Effective Address
<label>	Assemble Program Label
<list>	List of registers, for example D3–D0.
m	Bit m of an Operand
m–n	Bits m through n of Operand

3.1.1 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 byte, word, and long-word operands from memory to memory, memory to register, register to memory, and register to register. MOVE instructions transfer word and long-word operands and ensure that only valid address manipulations are executed. In addition to the general MOVE instructions, there are several special data movement instructions: MOVEM, MOVEQ, LEA, PEA, LINK, and UNLK.

Table 3-2. Data Movement Operation Format

Instruction	Operand Syntax	Operand Size	Operation
LEA	<ea>,An	32	<ea> → An
LINK	An,#<data>	16	SP – 4 → SP; An → (SP); SP → An, SP + D → SP
MOVE MOVEA	<ea>,<ea> <ea>,An	8, 16, 32 16, 32 → 32	Source → Destination
MOVEM	list,<ea> <ea>,list	32 32	Listed Registers → Destination Source → Listed Registers
MOVEQ	#<data>,Dn	8 → 32	Immediate Data → Destination
PEA	<ea>	32	SP – 4 → SP; <ea> → (SP)
UNLK	An	32	An → SP; (SP) → An; SP + 4 → SP

3.1.2 Integer Arithmetic Instructions

The integer arithmetic operations include 3 basic operations: ADD, SUB, and MUL. They also include CMP, CLR, and NEG. The instruction set includes ADD, CMP, and SUB instructions for both address and data operations with all operand sizes valid for data operations. Address operands consist of 16 or 32 bits. The CLR and NEG instructions apply to all sizes of data operands. Signed and unsigned MUL instruction includes

- word multiply to produce a long-word product
- long-word multiply to produce a long-word product

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 X-bit in the CCR.

Table 3-3. Integer Arithmetic Operation Format

Instruction	Operand Syntax	Operand Size	Operation
ADD ADDA	Dn,<ea> <ea>,Dn <ea>,An	32 32 32	Source + Destination → Destination
ADDI ADDQ	#<data>,Dn #<data>,<ea>	32 32	Immediate Data + Destination → Destination
ADDX	Dy,Dx	32	Source + Destination + X → Destination
CLR	<ea>	8, 16, 32	0 → Destination
CMP CMPA	<ea>,Dn <ea>,An	32 32	Destination – Source
CMPI	#<data>, Dn	8, 16, 32	Destination – Immediate Data
EXT EXTB	Dn Dn Dn	8 → 16 16 → 32 8 → 32	Sign-Extended Destination → Destination
MULS/MULU	<ea>,Dn <ea>,Dl	16 x 16 → 32 32 x 32 → 32	Source x Destination → Destination (Signed or Unsigned)
NEG	<ea>	32	0 – Destination → Destination
NEGX	<ea>	32	0 – Destination – X → Destination
SUB SUBA	<ea>,Dn Dn,<ea> <ea>,An	32 32 32	Destination = Source → Destination
SUBI SUBQ	#<data>, Dn #<data>,<ea>	32 32	Destination – Immediate Data → Destination
SUBX	Dy,Dx	32	Destination – Source – X → Destination

3.1.3 Logical Instructions

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

Table 3-4. Logical Operation Format

Instruction	Operand Syntax	Operand Size	Operation
AND	<ea>,Dn Dn,<ea>	32 32	Source Λ Destination $\rightarrow$ Destination
ANDI	#<data>, Dn	32	Immediate Data Λ Destination $\rightarrow$ Destination
EOR	Dn,<ea>	32	Source $\oplus$ Destination $\rightarrow$ Destination
EORI	#<data>, Dn	32	Immediate Data $\oplus$ Destination $\rightarrow$ Destination
NOT	<ea>	32	~ Destination $\rightarrow$ Destination
OR	<ea>,Dn Dn,<ea>	32	Source V Destination $\rightarrow$ Destination
ORI	#<data>, Dn	32	Immediate Data V Destination $\rightarrow$ Destination

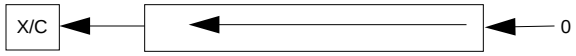
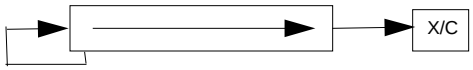
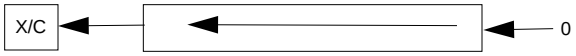
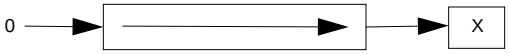
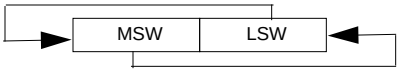
3.1.4 Shift Instruction

The ASR, ASL, LSR, and LSL instructions provide shift operations in both directions. All shift operations can be performed on either registers or memory.

Register shift operations shift all operand sizes. 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).

Memory shift operations shift word operands one bit position only. 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 C-bit and X- bit in the CCR.

Table 3–5. Shift Operation Format

Instruction	Operand Syntax	Operand Size	Operation
ASL	Dx, Dy # <data>, Dn	32 32	
ASR	Dx, Dy #<data>, Dn	32 32	
LSL	Dx, Dy #<data>, Dn	32 32	
LSR	Dx, Dy #<data>, Dn	32 32	
SWAP	Dn	16	

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

3.1.5 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	Dn,<ea> #<data>,<ea>	8, 32 8, 32	~ (<Bit Number> of Destination) → Z → Bit of Destination
BCLR	Dn,<ea> #<data>,<ea>	8, 32 8, 32	~ (<Bit Number> of Destination) → Z; 0 → Bit of Destination
BSET	Dn,<ea> #<data>,<ea>	8, 32 8, 32	~ (<Bit Number> of Destination) → Z; 1 → Bit of Destination
BTST	Dn,<ea> #<data>,<ea>	8, 32 8, 32	~ (<Bit Number> of Destination) → Z

3.1.6 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 integer condition codes for use by other program and system control instructions. NOP forces synchronization of the internal pipelines. Table 3-7 summarizes these instructions.

Table 3-7. Program Control Operation Format

Instruction	Operand Syntax	Operand Size	Operation
Integer Conditional			
Bcc	<label>	8, 16	If Condition True, Then PC + d_n → PC
Scc	Dn	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; PC → (SP); PC + d_n → PC
JMP	<ea>	none	Destination → PC
JSR	<ea>	none	SP – 4 → SP; PC → (SP); Destination → PC
NOP	none	none	PC + 2 → PC (Integer Pipeline Synchronized)
TRAPF TRAPF	none # <data>	none 16 32	PC + 2→ PC PC + 4 → PC PC + 6→ PC
Returns			
RTS	none	none	(SP) → PC; SP + 4 → SP
Test Operand			
TST	<ea>	8, 16, 32	Set Integer Condition Codes

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

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

*Not applicable to the Bcc instructions.

3.1.7 System Control Instructions

Privileged and trapping instructions as well as instructions that use or modify the CCR provide system control operations. The conditional trap instructions, which use the same conditional tests as their corresponding program control instructions, allow an optional 16- or 32-bit immediate operand to be included as part of the instruction for passing parameters to the operating system. These instructions cause the processor to flush the instruction pipe. Table 3-8 summarizes these instructions. See **3.2 Integer Unit Condition Code Computation** for more details on condition codes.

Table 3-8. System Control Operation Format

Instruction	Operand Syntax	Operand Size	Operation
Privileged			
MOVE to SR	Dn, SR, #<data>, SR	16	Source → SR Dn or #<data> source only
MOVE from SR	Dn	16	SR → Destination
MOVEC	Rn,Rc	32	Rn → 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	
WDEBUG	<ea-2>	16	
PULSE	none	none	
WDDATA	<ea>	8, 16, 32	Immediate Data → DDATA port
Trap Generating			
TRAP	none	none	SP – 4 → SP; PC → (SP); SP – 2 → SP; SR → (SP) SP – 2 → SP; Format → (SP) Vector Address → PC
Condition Code Register			
MOVE to SR	Dn, SR, SR #<data>	16	Source → CCR
MOVE from SR	Dn	16	CCR → Destination

3.2 INTEGER UNIT CONDITION CODE COMPUTATION

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.

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 Λ Dm Λ $\overline{Rm}$ V $\overline{Sm}$ Λ $\overline{Dm}$ Λ Rm C = Sm Λ Dm V $\overline{Rm}$ Λ Dm V Sm Λ $\overline{Rm}$
ADDX	*	*	?	?	?	V = Sm Λ Dm Λ $\overline{Rm}$ V $\overline{Sm}$ Λ $\overline{Dm}$ Λ Rm C = Sm Λ Dm V $\overline{Rm}$ Λ Dm V Sm Λ $\overline{Rm}$ Z = Z Λ $\overline{Rm}$ Λ …Λ $\overline{R0}$
AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT, EXTB, NOT, TST	—	*	*	0	0	
SUB, SUBI, SUBQ	*	*	*	?	?	V = $\overline{Sm}$ Λ Dm Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ Rm C = Sm Λ $\overline{Dm}$ V Rm Λ $\overline{Dm}$ V Sm Λ Rm
SUBX	*	*	?	?	?	V = $\overline{Sm}$ Λ Dm Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ Rm C = Sm Λ $\overline{Dm}$ V Rm Λ $\overline{Dm}$ V Sm Λ Rm Z = Z Λ $\overline{Rm}$ Λ …Λ $\overline{R0}$
CMP, CMPA, CMPI	—	*	*	?	?	V = $\overline{Sm}$ Λ Dm Λ $\overline{Rm}$ V Sm Λ $\overline{Dm}$ Λ Rm C = Sm Λ $\overline{Dm}$ V Rm Λ $\overline{Dm}$ V Sm Λ Rm
MULS, MULU	—	*	*	0	0	
NEG	*	*	*	?	?	V = Dm Λ Rm C = Dm V Rm
NEGX	*	*	?	?	?	V = Dm Λ Rm C = Dm V Rm Z = Z Λ $\overline{Rm}$ Λ …Λ $\overline{R0}$
BTST, BCHG, BSET, BCLR	—	—	?	—	—	Z = $\overline{Dn}$
ASL	*	*	*	0	?	C = $\overline{Dm-r+1}$
ASL (r = 0)	—	*	*	0	0	
LSL	*	*	*	0	?	C = Dm – r + 1

Table 3-9. Integer Unit Condition Code Computations (Continued)

Operations	X	N	Z	V	C	Special Definition
LSR (r = 0)	—	*	*	0	0	
ASR, LSR, ROXR	*	*	*	0	?	C = Dr – 1
ASR, LSR (r = 0)	—	*	*	0	0	

? = Other—See Special Definition
N = Result Operand (MSB)
Z = $\overline{Rm} \Lambda \ldots \Lambda \overline{R0}$
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	$\overline{C} \Lambda \overline{Z}$
LS	Low or Same	0011	C V Z
CC(HI)	Carry Clear	0100	C
CS(LO)	Carry Set	0101	C
NE	Not Equal	0110	Z
EQ	Equal	0111	Z
VC	Overflow Clear	1000	V
VS	Overflow Set	1001	V
PL	Plus	1010	N
MI	Minus	1011	N
GE	Greater or Equal	1100	$N \Lambda V V \overline{N} \Lambda \overline{V}$
LT	Less Than	1101	$N \Lambda \overline{V} V \overline{N} \Lambda V$
GT	Greater Than	1110	$N \Lambda V \Lambda \overline{Z} V \overline{N} \Lambda \overline{V} \Lambda \overline{Z}$
LE	Less or Equal	1111	$Z V N \Lambda \overline{V} V \overline{N} \Lambda V$

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

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 ADD

Operation: Source + Destination → Destination

Assembler Syntax: ADD < ea > ,Dn
ADD Dn, < ea >

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 may be specified as a long word. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X — set the same as the carry bit
N — set if the result is negative; cleared otherwise
Z — set if the result is zero; cleared otherwise
V — set if an overflow is generated; cleared otherwise
C — set if a carry is generated; cleared otherwise

Instruction Format:

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

ADD Add ADD

Instruction Fields:

Register field—specifies any of the 8 data registers.

Opmode field

Long	Operation
010	< ea > + Dn
110	Dn + < ea > → < ea >

Effective Address field—determines addressing mode

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

ADD Add ADD

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

Addressing Mode	Mode	Register
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

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

NOTE

The Dn 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 ADDA

Operation: Source + Destination → Destination

Assembler Syntax: ADDA < ea > , An

Attributes: Size = Long

Description: Adds the source operand to the destination address register and stores the result in the address register. The size of the operation is specified as a long word.

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 any of the 8 address registers (this is always the destination).

ADDA Add Address ADDA

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

ADDI — Add Immediate — ADDI

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ADDI # < data > , Dn

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 long word.

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.

ADDQ — Add Quick — ADDQ

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ADDQ # < data > , < ea >

Attributes: Size = Long

Description: Adds an immediate value of 1 to 8 to the operand at the destination location. The size of the operation is specified as long word. When adding to address registers, the condition codes are not altered and the entire destination address register is used.

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		

ADDQ Add Quick ADDQ

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
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

ADDX Add Extended ADDX

Operation: Source + Destination + X → Destination

Assembler 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 long word.

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

ADDX Add Extended ADDX

Instruction Format:

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

Instruction Fields:

Register Rx field—specifies the destination data register.

Register Ry field—specifies the source data register.

AND — AND Logical — AND

Operation: Source L Destination → Destination

Assembler Syntax: AND < ea > ,Dn
AND Dn, < ea >

Attributes: Size = Long

Description: Performs an AND operation of the source operand with the destination operand and stores the result in the destination location. The size of the operation is specified as a long word. Address register contents may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Fields:

Register field—Specifies any of the 8 data registers.

Opmode field

Long	Operation
010	< ea > Λ Dn → Dn
110	Dn Λ < ea > → < ea >

AND AND Logical AND

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:

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

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

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
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

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

ANDI AND Immediate ANDI

Operation: Immediate Data Λ Destination $\rightarrow$ Destination

Assembler Syntax: ANDI # < data > , Dn

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 long word. The size of the immediate data is specified as a long word.

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.

ASL, ASR Arithmetic Shift ASL, ASR

Operation: Destination Shifted By Count → Destination

Assembler Syntax:
ASd Dx,Dy
ASd # < data > ,Dy
where d is direction, L or R

Attributes: Size = Long

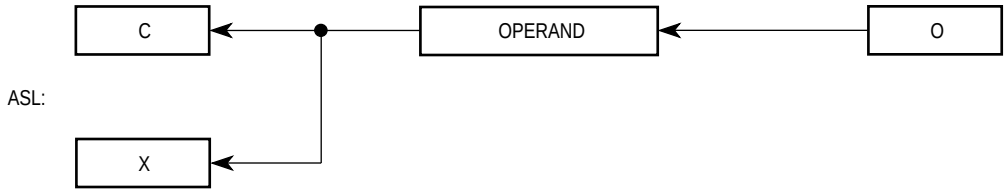
Description: Arithmetically shifts the bits of the operand in the direction (L or R) specified. The carry bit receives the last bit shifted out of the operand. The shift count for the shifting of a register may be specified in two different ways:

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)

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

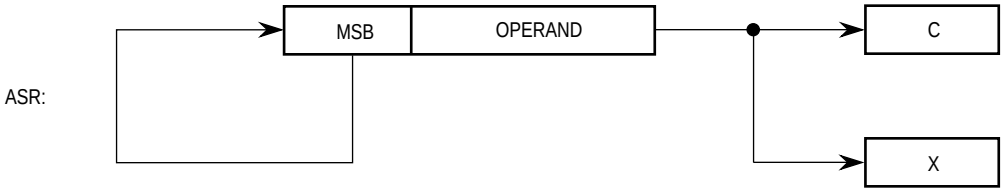
For ASL, the operand is shifted left; the shift count equals the number of positions shifted. Bits shifted out of the high-order bit go to both the carry and the extend bits; zeros are shifted into the low-order bit. The overflow bit is always zero.

.



ASL, ASR Arithmetic Shift ASL, ASR

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:

REGISTER SHIFTS

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

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).

ASL, ASR Arithmetic Shift ASL, ASR

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 a data register to be shifted.

Bcc Branch Conditionally Bcc

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

Assembler Syntax: Bcc < label >

Attributes: Size = Word or Long

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 (refer to Table 3-19 for more information on these conditional tests):

Mnemonic	Condition
CC(HI)	Carry Clear
CS(LO)	Carry Set
EQ	Equal
GE	Greater or Equal
GT	Greater Than
HI	High
LE	Less or Equal

Mnemonic	Condition
LS	Low or Same
LT	Less Than
MI	Minus
NE	Not Equal
PL	Plus
VC	Overflow Clear
VS	Overflow Set

Condition Codes:

Not affected

Bcc Branch Conditionally Bcc

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															

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 BCHG

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

Assembler Syntax: BCHG Dy, < ea >
BCHG # < data > , ea

Attributes: Size = Byte, Long

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

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

Condition Codes:

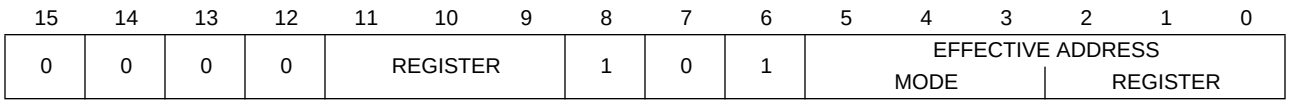
X	N	Z	V	C
—	—	*	—	—

X — not affected
N — not affected
Z — set if the bit tested is zero; cleared otherwise
V — not affected
C — not affected

BCHG Test a Bit and Change BCHG

Instruction Format:

BIT NUMBER DYNAMIC, SPECIFIED IN A 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
Dy*	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

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

*Long only; all others are byte

BCHG Test a Bit and Change BCHG

Instruction Format:

BIT NUMBER STATIC, SPECIFIED AS IMMEDIATE DATA

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

Instruction Fields:

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

Addressing Mode	Mode	Register
Dy*	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)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

*Long only; all others are byte

Bit Number field—specifies the bit number.

BCLR Test a Bit and Clear BCLR

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

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

Attributes: Size = Byte, Long

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

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

Condition Codes:

X	N	Z	V	C
—	—	*	—	—

X — not affected
N — not affected
Z — set if the bit tested is zero; cleared otherwise
V — not affected
C — not affected

BCLR Test a Bit and Clear BCLR

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.

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

Addressing Mode	Mode	Register
Dy*	000	reg. number:Dy
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)	—	—

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

*Word only; all others are byte

BCLR Test a Bit and Clear BCLR

Instruction Format:

BIT NUMBER STATIC, SPECIFIED AS IMMEDIATE DATA

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

Instruction Fields:

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

Addressing Mode	Mode	Register
Dy*	000	reg. number:Dy
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)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

*Long only; all others are byte

Bit Number field—specifies the bit number.

BRA Branch Always BRA

Operation: PC + $d_n \rightarrow$ 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															

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 automatically uses the 16-bit displacement format because the 8-bit displacement field contains $00 (zero offset).

BSET Test a Bit and Set BSET

Operation: TEST (< bit number > of Destination) → Z; 1 → < bit number > of Destination

Assembler Syntax:
BSET Dn, < ea >
BSET # < data > , < ea-1 >

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

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

BSET Test a Bit and Set BSET

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	EFFECTIVE ADDRESS 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
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)	—	—

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

*Long only; all others are byte

BSET Test a Bit and Set BSET

Instruction Format:

BIT NUMBER STATIC, SPECIFIED AS IMMEDIATE DATA

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

Instruction Fields:

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

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)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

*Long only; all others are byte

Bit Number field—specifies the bit number.

BSR Branch to Subroutine BSR

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

Assembler Syntax: BSR < label >

Attributes: Size = Byte, Word

Description: Pushes the word 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

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															

BSR Branch to Subroutine BSR

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 automatically uses the 16-bit displacement format because the 8-bit displacement field contains $00 (zero offset).

BTST Test a Bit BTST

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

Assembler Syntax: BTST Dn, < ea >
BTST # < data > , < ea >

Attributes: Size = Byte, Long

Description: Tests a bit in the destination operand and sets the Z-condition code appropriately. When a data register is the destination, any of the 32 bits can be specified by a modulo 32- bit number. When a memory location is the destination, the operation is a byte operation, and the bit number is modulo 8. In all cases, bit 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

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

BTST Test a Bit BTST

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.

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

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

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

*Long only; all others are byte

BTST Test a Bit BTST

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							

Instruction Fields:

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

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)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

Bit Number field—specifies the bit number.

CLR — Clear an Operand — CLR

Operation: 0 → Destination

Assembler Syntax: CLR < ea >

Attributes: Size = Byte, Word, Long

Description: Clears the destination operand to 0. 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		

CLR Clear an Operand CLR

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
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

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

CMP Compare CMP

Operation: Destination – Source → cc

Assembler Syntax: CMP < ea > , Dn

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 long word.

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.

CMP Compare CMP

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

CMPA Compare Address CMPA

Operation: Destination – Source → cc

Assembler Syntax: CMPA < ea > , An

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	EFFECTIVE ADDRESS: REGISTER

CMPA Compare Address CMPA

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
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

CMPI Compare Immediate CMPI

Operation: Destination – Immediate Data → cc

Assembler Syntax: CMPI # < data > , Dn

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 long word. The size of the immediate data is specified as a long word.

Condition Codes:

X	N	Z	V	C
—	*	*	*	*

X — not affected
N — set if the result is negative; cleared otherwise
Z — set if the result is zero; cleared otherwise
V — set if an overflow occurs; cleared otherwise
C — set if a borrow occurs; cleared otherwise

Instruction Format:

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

Instruction Fields:

Register field - destination data register.

EOR — Exclusive-OR Logical — EOR

Operation: Source ⊕ Destination → Destination

Assembler Syntax: EOR Dn, < ea >

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. The size of the operation is specified as a long word. The source operand must be a data register. The destination operand is specified in the effective address field.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Fields:

Register field—specifies any of the 8 data registers.

EOR Exclusive-OR Logical EOR

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

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

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

EORI Exclusive-OR Immediate EORI

Operation: Immediate Data ⊕ Destination → Destination

Assembler Syntax: EORI # < data > , Dn

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 size of the operation is specified as a long word.

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 IMMEDIATE DATA															
LOWER WORD IMMEDIATE DATA															

Instruction Fields:

Register field - destination data register.

EXT, EXTB Sign-Extend EXT, EXTB

Operation: Destination Sign-Extended → Destination

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

Attributes: Size = Word, Long

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

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Fields:

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

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

JMP Jump JMP

Operation: Destination Address → PC

Assembler Syntax: JMP < ea >

Attributes: Unsized

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

Condition Codes:

Not affected

Instruction Format:

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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

JSR — Jump to Subroutine — JSR

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

Assembler Syntax: JSR < ea >

Attributes: Unsized

Description: Pushes the long-word address of the instruction immediately following the JSR instruction onto the system stack. Program execution then continues at the address specified 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

LEA Load Effective Address LEA

Operation: < ea > → An

Assembler Syntax: LEA < ea > ,An

Attributes: Size = Long

Description: Loads the effective address into the specified address register. 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 address register to be updated with the effective address.

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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

LINK — Link and Allocate — LINK

Operation: SP – 4 → SP; An → (SP); SP → An; SP + d_n → SP

Assembler Syntax: LINK An, # < 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.

LSL, LSR Logical Shift LSL, LSR

Operation: Destination Shifted By Count → Destination

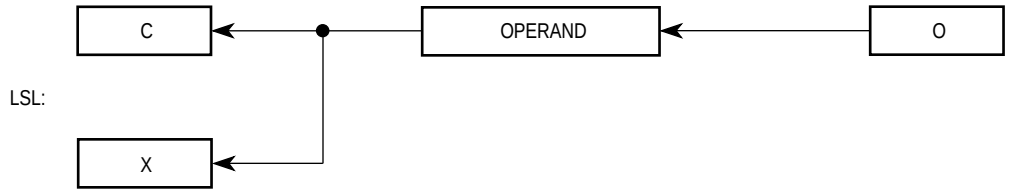
Assembler Syntax: LSd Dx,Dy
LSd # < data > ,Dy
where d is direction, L or R

Attributes: Size = Long

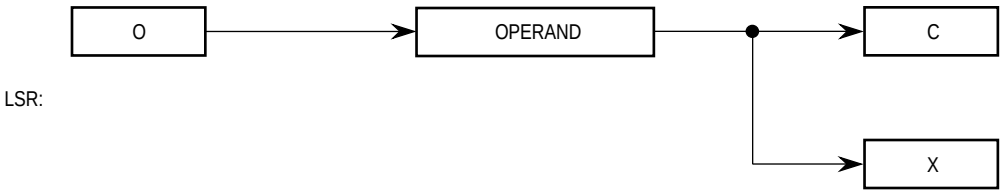
Description: Shifts the bits of the operand in the direction specified (L or R). The carry bit receives the last bit shifted out of the operand. The shift count for the shifting of a register is specified in two different ways:

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.



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



LSL, LSR — Logical Shift — LSL, LSR

Condition Codes:

X	N	Z	V	C
*	*	*	0	*

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

Instruction Format:

REGISTER SHIFTS

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

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

Register field—specifies a data register to be shifted.

I/R field
0 — immediate shift count
1 — register shift count

MOVE MOVEA — Move Data from Source to Destination — MOVE MOVEA

Operation: Source → Destination

Assembler Syntax:
MOVE < ea > , < ea >
MOVEA <ea>, An

Attributes: Size = Byte, Word, Long

Description: Moves the data at the source to the destination location and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or long word.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Fields:

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

MOVE MOVEA — Move Data from Source to Destination — MOVE MOVEA

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. The table shown on the bottom of page 4-55 outlines the restrictions.

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number: An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

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

MOVE MOVEA Move Data from Source to Destination MOVE MOVEA

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 on the bottom of this page outlines the restrictions.

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

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 table below shows the possible combinations.

Source Addressing Mode	Destination Addressing Mode
Dn, An, (An), (An)+,-(An)	All possible
(d_{16}, An), (d16, PC)	All possible except (d_8, An, Xn), (xxx).W, (xxx).L
(d8, An, Xn), (d8, PC, Xn), (xxx).W, (xxx).L, #<xxx>	All possible except (d_8, An, Xn), (d_{16}, An), (xxx).W, (xxx).L

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

MOVE from CCR

Move from the Condition Code Register

MOVE from CCR

Operation: CCR → Destination

Assembler Syntax: MOVE CCR, Dn

Attributes: Size = Word

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

Condition Codes:

Not affected

Instruction Format:

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 - destination data register.

MOVE to CCR — Move to Condition Code Register

MOVE to CCR

Operation: Source → CCR

Assembler Syntax:
MOVE, Dn ,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		

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

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
Dn	000	reg. number:Dn
An	—	—
(An)	—	—
(An) +	—	—
– (An)	—	—
(d_{16},An)	—	—
(d_8,An,Xn)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	111	100
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

MOVEM Move Multiple Registers MOVEM

Operation: Registers → Destination; Source → Registers

Assembler Syntax: MOVEM < list > , < ea >
MOVEM < ea > , < 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.

MOVEM Move Multiple Registers MOVEM

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.

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

MOVEM Move Multiple Registers MOVEM

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

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

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 MOVEQ

Operation: Immediate Data → Destination

Assembler Syntax: MOVEQ # < data > ,Dn

Attributes: Size = Long

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

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Fields:

Register field—specifies the data register to be loaded.

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

MULS Signed Multiply MULS

Operation: Source x Destination → Destination

Assembler Syntax: MULS.W < ea > ,Dn 16 x 16 → 32
MULS.L < ea > ,Dl 32 x 32 → 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 long-word operand. A register operand is the low-order word; the upper word of the register is ignored. All 32 bits of the product are saved in the destination data register.

In the long form, the multiplier and multiplicand are both long word operands. 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

MULS Signed Multiply MULS

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 a data register as the destination.

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

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

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

MULS Signed Multiply MULS

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			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
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)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

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

MULU — Unsigned Multiply — MULU

Operation: Source x Destination → Destination

Assembler Syntax: MULU.W < ea > ,Dn 16 x 16 → 32
MULU.L < ea > ,Dl 32 x 32 → 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 long- word 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

MULU Unsigned Multiply MULU

Instruction Format:

WORD

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

Instruction Fields:

Register field—specifies a data register as the destination.

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

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

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

MULU Unsigned Multiply MULU

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
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)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

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

NEG Negate NEG

Operation: 0 – Destination → Destination

Assembler Syntax: NEG Dn

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 long word.

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 data register used.

NEGX Negate with Extend NEGX

Operation: 0 – Destination – X → Destination

Assembler Syntax: NEGX Dn

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 long word.

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

NEGX Negate with Extend NEGX

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 data register used.

NOP — No Operation — NOP

Operation: None

Assembler Syntax: NOP

Attributes: Unsized

Description: Performs no operation. The processor state, other than the program counter, is unaffected. Execution continues with the instruction following the NOP instruction. The NOP instruction does not begin execution until all pending bus cycles 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 — NOT

Operation: ~ Destination → Destination

Assembler Syntax: NOT Dn

Attributes: Size = Long

Description: Calculates the ones complement of the destination operand and stores the result in the destination location. The size of the operation is specified as a long word.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Fields:

Register field - specifies data register used.

OR — Inclusive-OR Logical — OR

Operation: Source V Destination → Destination

Assembler Syntax: OR < ea > ,Dn
OR Dn, < ea >

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 long word. The contents of an address register may not be used as an operand.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Fields:

Register field—specifies any of the 8 data registers.

Opmode field

Long	Operation
010	< ea > V Dn → Dn
110	Dn V < ea > → < ea >

OR Inclusive-OR Logical OR

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

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

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

OR Inclusive-OR Logical OR

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

Addressing Mode	Mode	Register
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

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

NOTE

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

ORI — Inclusive-OR — ORI

Operation: Immediate Data V Destination → Destination

Assembler Syntax: ORI # < data > , Dn

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 long word. The size of the immediate data is specified as a long word.

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															

PEA Push Effective Address PEA

Operation: SP – 4 → SP; < ea > → (SP)

Assembler Syntax: PEA < ea >

Attributes: Size = Long

Description: Computes the effective address and pushes it onto the stack. The effective address is a long 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
Dn	—	—
An	—	—
(An)	010	reg. number:An
(An) +	—	—
– (An)	—	—
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	111	010
(d_8,PC,Xn)	111	011

RTS Return from Subroutine RTS

Operation: (SP) → PC; SP + 4 → SP

Assembler Syntax: RTS

Attributes: Unsized

Description: Pulls the program counter value from the stack. The previous 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 — Scc

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

Assembler Syntax: Scc Dn

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 (refer to Table 3-19 for more information on these 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

Scc — Set According to Condition — Scc

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 region.

SUB Subtract SUB

Operation: Destination – Source → Destination

Assembler Syntax: SUB < ea > ,Dn
SUB Dn, < ea >

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:

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	EFFECTIVE ADDRESS REGISTER

SUB Subtract SUB

Instruction Fields:

Register field—specifies any of the 8 data registers.

Opmode field

Long	Operation
010	Dn – < ea > → Dn
110	< ea > – Dn → < ea >

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

SUB Subtract SUB

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

Addressing Mode	Mode	Register
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

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

NOTE

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

SUBA Subtract Address SUBA

Operation: Destination – Source → Destination

Assembler Syntax: SUBA < ea > ,An

Attributes: Size = Long

Description: Subtracts the source operand from the destination address register and stores the result in the address register. The size of the operation is specified as a long word.

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, any of the 8 address registers.

SUBA Subtract Address SUBA

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

SUBI Subtract Immediate SUBI

Operation: Destination – Immediate Data → Destination

Assembler Syntax: SUBI # < data > , Dn

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 long word.

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.

SUBQ Subtract Quick SUBQ

Operation: Destination – Immediate Data → Destination

Assembler Syntax: SUBQ # < data > , < ea >

Attributes: Size = Long

Description: Subtracts the immediate data (1 – 8) from the destination operand. The size of the operation is specified as a long word. Operations do not affect the condition codes. When subtracting from address registers, the entire destination address register.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X — set to the value of the carry bit
N — set if the result is negative; cleared otherwise
Z — set if the result is zero; cleared otherwise
V — set if an overflow occurs; cleared otherwise
C — set if a borrow occurs; cleared otherwise

Instruction Format:

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

SUBQ Subtract Quick SUBQ

Instruction Fields:

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

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

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An*	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,An,Xn)	110	reg. number:An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

SUBX Subtract with Extend SUBX

Operation: Destination – Source – X → Destination

Assembler Syntax: SUBX Dx,Dy

Attributes: Size = Long

Description: Subtracts the source operand and the extend bit from the destination operand and stores the result in 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 — cleared if the result is nonzero; unchanged otherwise
V — set if an overflow occurs; cleared otherwise
C — set if a borrow occurs; cleared otherwise

SUBX Subtract with Extend SUBX

Instruction Format:

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

Instruction Fields:

Dy field — specifies destination data register.

Dx field — specifies source data register.

SWAP Swap Register Halves SWAP

Operation: Register 31 – 16 ⟷ Register 15 – 0

Assembler Syntax: SWAP Dn

Attributes: Size = Word

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

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:

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

Instruction Field:

Register field—specifies the data register to swap.

TRAP Trap TRAP

Operation: 1 → S-Bit of SR
SSP – 4 → SSP; PC → (SSP); SSP – 2 → SSP;
SR → (SSP); SSP – 2 → SSP; Format/Offset → (SSP);
Vector Address → PC

Assembler Syntax: TRAP # < vector >

Attributes: Unsized

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

Condition Codes:

Not affected

Instruction Format:

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 TRAPF

Operation: No operation

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

Attributes: Unsized or Size = Word or Long

Description: This instruction performs no operation. It can be used to occupy 16, 32, or 48 bits in instruction space.

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:

Vector field—specifies the trap vector to be taken.

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

TST — Test an Operand — TST

Operation: Destination Tested → Condition Codes

Assembler Syntax: TST < ea >

Attributes: Size = Byte, Word, Long

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

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		

TST Test an Operand TST

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 addressing mode for the destination operand as listed in the following table:

Addressing Mode	Mode	Register
Dn	000	reg. number:Dn
An	001	reg. number:An
(An)	010	reg. number:An
(An) +	011	reg. number:An
– (An)	100	reg. number:An
(d_{16},An)	101	reg. number:An
(d_8,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

UNLK Unlink UNLK

Operation: An → SP; (SP) → An; SP + 4 → SP

Assembler Syntax: UNLK An

Attributes: Unsized

Description: Loads the stack pointer from the specified address register, then loads the address register with the long word pulled from the top of the stack.

Condition Codes:

Not affected

Instruction Format:

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

Instruction Field:

Register field—specifies the address register for the instruction.

SECTION 5
SUPERVISOR (PRIVILEGED) INSTRUCTIONS

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.

Table 5-1. Supervisor Mode Instruction Summary

Opcode	Supported Operand Sizes	Addressing Modes
HALT	Unsized	
MOVE from SR	.W	Dn
MOVE to SR	.W	Dn, SR #<data>, SR
MOVEC	.L	Rn, Rc
RTE	Unsized	
STOP	Unsized	#<data>
WDEBUG	.L	<ea>
WDATA	.B, .W, .L	<ea>
PULSE	Unsized	

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 below.

Table 5-2. CPU Space Map

Rc[11:0][1]	Register Definition
$002	Cache Control Register (CACR)
$003	MMU Translation Control Register (TCR)
$004	Access Control Unit 2 (ACR2)
$005	Access Control Unit 3 (ACR3)
$006	Access Control Unit 0 (ACR0)
$007	Access Control Unit 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]
$C00	ROM Base Address Register (ROMBAR)
$C04	RAM Base Address Register 0 (RAMBAR0)
$C05	RAM Base Address Register 1 (RAMBAR1)
$C0F	Module Base Åddress Register (MBAR)

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

[2] Not accessible via MOVEC; only through the Debug interface, if implemented.

Note that 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.

HALT — Halt the CPU (Privileged) — HALT

Operation:
If Supervisor State
    Then Source Halt the Processor Core
Else TRAP

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.

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

PULSE Generate a Unique Processor Status PULSE

Operation: Generate a Unique Processor Status Encoding

Assembler Syntax: PULSE

Attributes: Unsized

Description: This instruction does not perform any explicit operation except for the generation of a unique encoding of the processor status output pins (PST = $4). This encoding is asserted by one processor clock cycle and is useful in providing a trigger to external logic during debug, performance characterization, etc.

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	1	0	0

WDDATA Write to Debug Data WDDATA

Operation: Source --> DDATA Signal Pins

Assembler Syntax: WDDATA <ea>

Attributes: Size = Byte, Word, Long Word

Description: This instruction fetches the operand defined by the effective address and captures the data in the ColdFire debug module for display on the DDATA output pins. The size of the operand determines the number of nibbles displayed on the DDATA output pins. The value of the debug module configuration/status register (CSR) does not affect the operation of this instruction.

The execution of this instruction generates a processor status encoding matching the PULSE instruction before the referenced operand is displayed on the DDATA outputs.

Condition Codes: Not affected

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	SIZE		EFFECTIVE ADDRESS					
										MODE			REGISTER		

Instruction Fields:

Size field—specifies the size of the operand data:
- 00—byte operation
- 01—word operation
- 10—long operation

Effective Address field—determines the addressing mode for the operand to be written to the DDATA signal pins; use only those memory alterable addressing modes listed in the following table:

Addressing Mode	Mode	Register
Dn	—	—
An	—	—
(An)	010	reg. number: An
(An) +	011	reg. number: An
-(An)	101	reg. number: An
(d_{16}, An)	101	reg. number: An
(d_8, An, Xn)	110	reg. number: An

Addressing Mode	Mode	Register
(xxx).W	111	000
(xxx).L	111	001
#<data>	—	—
(d_{16}, PC)	—	—
(d_8, PC, Xn)	—	—

WDEBUG Write Debug Control Register WDEBUG

Operation: If Supervisor State
Then Addressed Debug Write Control Register Command Executed
Else TRAP

Assembler Syntax: WDEBUG <ea>

Attributes: Size = Long

Description: This instruction does two things. 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 long word address or the behavior of the operation is undefined. The debug command must be organized in memory as follows:

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], and (4) the lower 16 bits of the second long word are unused.

Condition Codes: Not affected

Instruction Format:

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

Instruction Fields:

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

Addressing Mode	Mode	Register
Dn		
An		
(An)	010	reg. number: An
(An) +	—	—
- (An)	—	—
(d_{16}, An)	101	reg. number: An
(d_8, An, Xn)	—	—

Addressing Mode	Mode	Register
(xxx).W		
(xxx).L		
#<data>		
(d_{16}, PC)		
(d_8, PC, Xn)		

MOVE from SR — Move from the Status Register — MOVE from SR

Operation: If Supervisor State
Then SR → Destination
Else TRAP

Assembler Syntax: MOVE SR, Dn

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 destination data register.

MOVE to SR — Move to the Status Register — MOVE to SR

Operation:
If Supervisor State
Then Source → SR
Else TRAP

Assembler Syntax:
MOVE < ea >, SR

Attributes:
Size = Word

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

Condition Codes: Set according to the source operand

Instruction Format:

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		

MOVE to SR — Move to the Status Register — MOVE to SR

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
Dn	000	reg. number:Dn
An	—	—
(An)	—	—
(An) +	—	—
—(An)	—	—
(d_{16},An)	—	—
(d_8,An,Xn)	—	—

Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
# < data >	111	100
(d_{16},PC)	—	—
(d_8,PC,Xn)	—	—

MOVEC Move Control Register MOVEC

Operation: If Supervisor State
Then Rn → Rc
Else TRAP

Assembler Syntax: MOVEC Rn,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	dr
A/D	REGISTER			CONTROL REGISTER											

Instruction Fields:

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

Actual control registers in a given design can vary. Only the VBR exits in all ColdFire designs. Access to undefined control register space yields undefined results and must not be attempted. Access to unimplemented, but defined, control registers generally results in a virtual no-operation. The exception to this is the MBAR register. Access to an unimplemented MBAR may result in a system hang or bus error, depending on the implementation.

MOVEC Move Control Register MOVEC

Register field—specifies the register number.

Control Register field—specifies the control register.

Table 4-1. CPU Space Map

Rc[11:0]	Register Definition
$002	Cache Control Register (CACR)
$003	MMU Translation Control Register (TCR)
$004	Access Control Unit 2 (ACR2)
$005	Access Control Unit 3 (ACR3)
$006	Access Control Unit 0 (ACR0)
$007	Access Control Unit 1 (ACR1)
$801	Vector Base Register (VBR)
$C00	ROM Base Address Register (ROMBAR)
$C04	RAM Base Address Register 0 (RAMBAR0)
$C05	RAM Base Address Register 1 (RAMBAR1)
$COF	Module Base Address Register (MBAR)

RTE Return from Exception RTE

Operation: 2 (SP) → SR; 4 (SP) → PC; SP + 8→ SP
Adjust stack according to format

Assembler Syntax: RTE

Attributes: Unsized

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

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

Instruction Format:

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

STOP Load Status Register and Stop STOP

Operation: If Supervisor State
Then Immediate Data → SR; STOP
Else TRAP

Assembler Syntax: STOP # < data >

Attributes: Unsized

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, and (3) stops the fetching and executing of instructions. A trace, interrupt, or reset exception causes the processor to resume instruction execution. A trace exception occurs if instruction tracing is enabled (T0 = 1) when the STOP instruction begins execution. If an interrupt request is asserted with a priority higher than the priority level set by the new status register value, an interrupt exception occurs; otherwise, the interrupt request is ignored. External reset always initiates reset exception processing. 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.

SECTION 6
INSTRUCTION FORMAT SUMMARY

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

6.1 INSTRUCTION FORMAT

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

6.1.1 Effective Address Field

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

6.1.2 Shift Instruction

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

6.1.2.1 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

6.1.2.2 Register Field. This field specifies a data register to be shifted.

6.1.3 Size Field

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

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

6.1.4 Opmode Field

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

6.1.5 Address/Data Field

This field specifies the type of general register and is encoded as follows:

0 — data register
1 — address register

6.2 OPERATION CODE MAP

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/MOVEP/Immed iate
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	(Unassigned, Reserved)
1011	CMP/EOR
1100	AND/MUL /EXG
1101	ADD/ADDX
1110	Shift/Bit Field
1111	Unassigned reserved

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. ANDI

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															

3. 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															

4. 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															

5. 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															

6. 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															

7. 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							

8. 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							

9. 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							

10. 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								

11. 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		

12. 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		

13. 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		

14. 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		

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. 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		

17. MOVE from CCR

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		

18. 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		

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 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	EFFECTIVE ADDRESS					
										MODE			REGISTER		

21. 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		EFFECTIVE ADDRESS					
										MODE			REGISTER		

22. 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		

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. 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		

26. 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		

27. 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		

28. 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		

29. 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

30. 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

31. MULU

WORD

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		

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

32. MULS

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		

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			1	0	0	0	0	0	0	0	0	0	0	0

33. 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			

34. LINK

WORD

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

35. 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

36. 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															

37. 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

38. 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

39. MOVEC

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

40. 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		

41. 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		

42. 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															

43. 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		

44. 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		

45. 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		

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. 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															

48. 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															

49. 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															

50. MOVEQ

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

51. 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		

52. 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		

53. 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		

54. 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		

55. 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		

56. 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		

57. 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		

58. MULU

WORD

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		

59. MULS

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		

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. ADDX

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

62. 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		

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. 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		

65. 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		

66. 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		

67. 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

SECTION 7
EXCEPTION PROCESSING

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.

7.1 EXCEPTION PROCESSING OVERVIEW

Exception processing is the transition from the normal processing of a program to the processing required for any special internal or external condition that preempts normal processing. External conditions that cause exceptions are interrupts from external devices, bus errors, and resets. Internal conditions that cause exceptions are instructions, address errors, and tracing. In addition, some instructions such as the TRAP and RTE instructions can generate exceptions as part of their normal execution. Illegal instructions and privilege violations cause exceptions as well. Exception processing uses an exception vector table and an exception stack frame. The following paragraphs describe the vector table and a generalized exception stack frame.

The ColdFire processor uses an instruction restart exception model but does require additional software support to recover from certain types of exceptions, i.e., access and address errors.

Exception processing is defined as the time from the detection of the fault condition until the first instruction of the handler has been fetched. Figure 7-1 illustrates a general flowchart for the steps taken by the processor during exception processing.

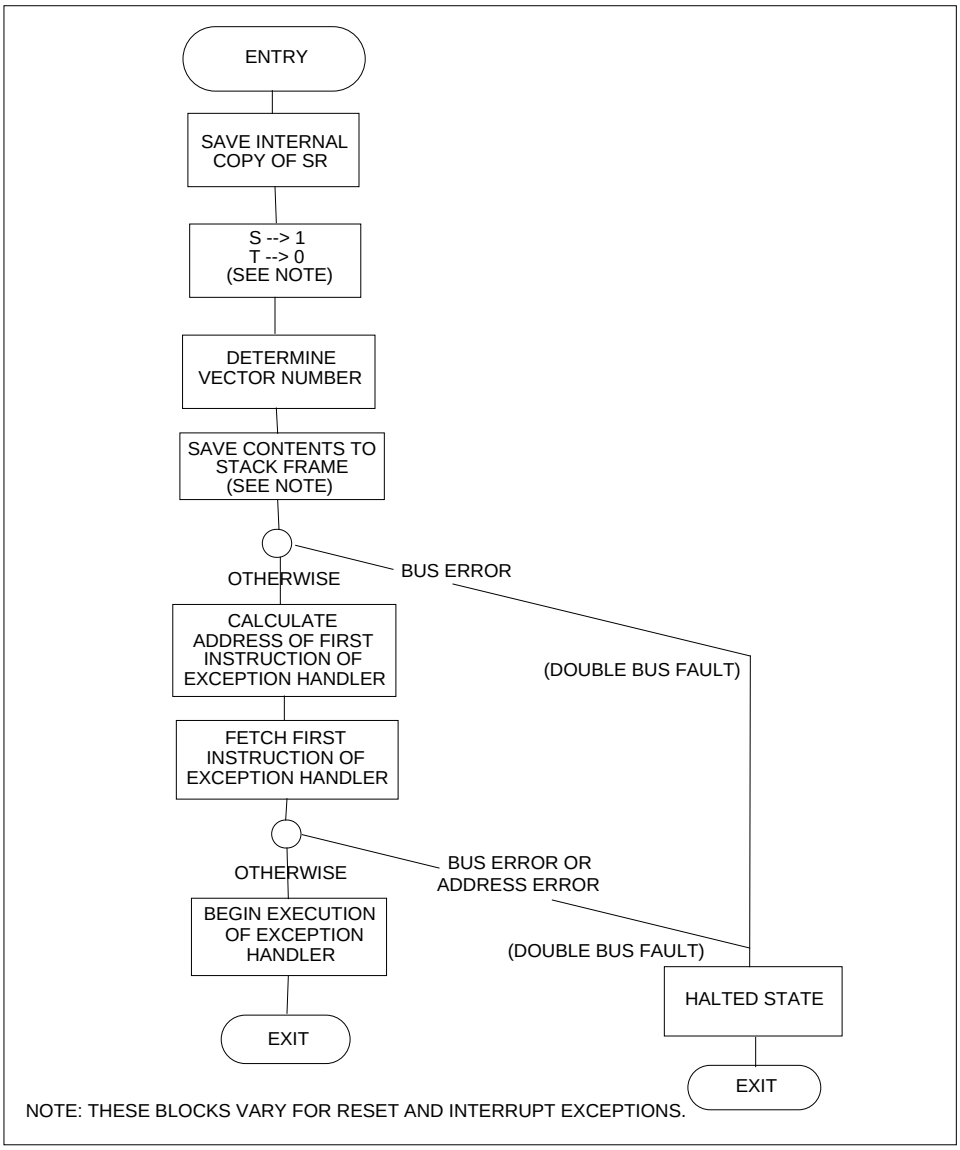


Figure 7-1. General Exception Processing Flowchart

First, the processor (a) makes an internal copy of the status register (SR), (b) enters supervisor mode by asserting the S-bit and (c) disables trace mode by negating the T-bit. Also, an interrupt exception forces an update of the interrupt priority mask.

Second, the processor determines the vector number of the exception. 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 transfer modifier.

Third, the processor saves the current context by creating an exception stack frame on the system stack. Note the ColdFire processor supports a single stack pointer in the A7 address register, i.e., there is no notion of a supervisor or user stack pointer. As a result, the excep-

tion stack frame is created on the top of the current system stack. The processor always forces the alignment of the exception stack frame to a 0-modulo-4 address, independent of the stack pointer value at the time of the exception. Additionally, the processor uses a simplified fixed-length stack frame for all exceptions, as shown in Figure 7-2. 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).

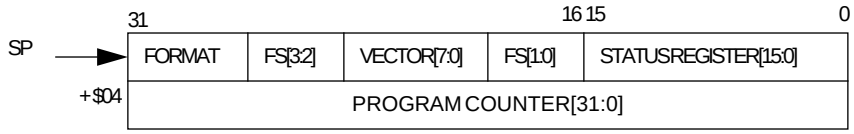


Figure 7-2. Exception Stack Frame Form

Fourth, the processor calculates the address of the first instruction of the exception handler. This address is generated by fetching an exception vector from the table located at the 0-modulo-1M address defined in the vector base register. For the ColdFire processor, relocating the exception vector table has been limited to 0-modulo-1M addresses. The index into the exception table is calculated as (4 x Vector_Number). Once the exception vector has been fetched, the vector contents determine the address of the first instruction of the desired handler. After fetching this initial handler instruction, exception processing terminates and normal instruction processing continues in the handler.

The ColdFire processor supports a 1024-byte vector table aligned on any 0-modulo-1M address (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. External devices can use vectors reserved for internal purposes at the discretion of the system designer. They can also supply vector numbers for some exceptions. External devices that cannot supply vector numbers use the autovector capability, which allows the processor to automatically generate a vector number.

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.

The ColdFire processor optionally supports autovectored interrupts. If the hardware module is included in the system, it converts an asserted $\overline{\textbf{AVEC}}$ signal into the appropriate vector for transmission to the processor core.

The ColdFire processor inhibits sampling for interrupts during the first instruction of all exception handlers. This allows any handler to effectively disable interrupts, if desired, by raising the interrupt mask level contained in the status register.

Normally, the end of an exception handler contains an RTE instruction. When the processor executes the RTE instruction, it examines the stack frame on top of the stack to determine if it is a valid frame. If the processor determines that it is a valid frame, the SR and PC fields are loaded from the exception frame and control is passed to the specified instruction address.

All exception vectors are located in the supervisor address space and are accessed using data references. Only the initial reset vector is fixed in the processor's memory map; once initialization is complete, there are no fixed assignments. Since the VBR provides the base address of the exception vector table, the exception vector table can be located anywhere in memory; it can even be dynamically relocated for each task that an operating system executes.

7.2 EXCEPTION STACK FRAME DEFINITION

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.

The 16-bit format/vector word contains 3 unique fields:

1. A 4-bit format field contained in bits[31:28] at the top of the system stack is always written by the processor with a value of {4,5,6,7} indicating a 2-longword frame format and the size of the stack pointer adjustment required to force 0-modulo-4 alignment. See Table 7-2.

Table 7-2. Format Field Encodings

Original A7 @ Time of Exception, bits 1:0	A7 @ 1st Instruction of Handler	Format Field
00	Original A7 - 8	0100
01	Original A7 - 9	0101
10	Original A7 - 10	0110
11	Original A7 - 11	0111

2. A 4-bit fault status field, FS[3:0], contained in bits[27:26, 17:16] 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 Table 7-3).

Table 7-3. Fault Status Encodings

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

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.

7.3 PROCESSOR EXCEPTIONS

The following paragraphs describe the external interrupt exceptions and the different types of exceptions generated internally by the integer unit. The following exceptions are discussed:

- Access Error
- Address Error
- Instruction Trap
- Illegal Instruction Exceptions
- Privilege Violation
- Trace
- Format Error
- Breakpoint Instruction
- Interrupt
- Reset

7.3.1 Access Error Exception

An access error exception in the ColdFire architecture occurs when a bus cycle terminates with an error condition such as $\overline{\text{TEA}}$. The exact response by the ColdFire processor to an access error depends on the type of bus cycle being performed.

If an instruction fetch results in an access error, the processor postpones error reporting until the faulted reference is needed by an instruction for execution. Thus, faults that occur during instruction prefetches which are then followed by a change of instruction flow will not generate an exception. 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.

If the access error occurs on an operand read, the processor immediately aborts the current instruction execution and initiates exception processing. In this situation, any address register updates due to the auto-addressing modes (e.g., (An)+, -(An)), will already have been performed. Thus, the programming model contains the updated An value.

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, an access error signal 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.

When the processor encounters an access error during the exception processing sequence of another exception, the processor enters a halted state.

7.3.2 Address Error Exception

Any attempted execution transferring control to an odd instruction address (i.e., if bit 0 of the target address is = 1) results in an address error exception.

The ColdFire processor default configuration supports word- and longword-sized operand references on 0-modulo-2 and 0-modulo-4 addresses, respectively. All other references are defined as misaligned accesses. Any attempt to access a misaligned operand generates an address-error exception, unless the optional hardware module for handling misalignment is present. This misalignment module converts any misaligned operand references into a series of aligned bus cycles to access the data. The existence of the misalignment module is implementation-dependent and is documented in the appropriate ColdFire user's manual.

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 attempted execution of a full-format indexed addressing mode.

Finally, when the processor encounters an address error during the exception processing sequence of another exception, the processor enters a halted state.

7.3.3 Trap Exception

The TRAP #n instruction always forces an exception and is useful for implementing system calls in user programs. Typically, passing a variable to the trap handler is done through registers. Otherwise, if passed through the stack, the stack frame format needs to be queried to determine the proper offset to the passed variables on the stack.

7.3.4 Illegal Instruction Exception

Attempting to execute the **$0000** and the **$4AFC** opwords generates an illegal instruction exception. Additionally, attempting to execute any line A and most line F opcode generates their unique exception types, vectors 10 and 11 respectively. The ColdFire processor does not provide illegal instruction detection on the extension words on any instruction, including **MOVEC**.

7.3.5 Privilege Violation

Any attempt to execute a supervisor mode instruction while in user mode will generate a privilege violation exception.

7.3.6 Trace Exception

To aid in program development, the ColdFire processor provides 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 allows a debugger to monitor execution of a program.

In general terms, a trace exception is an extension to the function of any traced instruction. The execution of a traced instruction is not complete until trace exception processing is complete. If an instruction does not complete because of an access error or address error exception, trace exception processing is deferred until the suspended instruction resumes execution. If an interrupt is pending at the completion of an instruction, trace exception processing occurs before interrupt exception processing starts. If an instruction forces an exception as part of its normal execution, the forced exception processing occurs before the trace exception is processed.

The T-bit in the supervisor portion of the SR controls tracing. The state of the T-bit when an instruction begins execution determines whether the instruction generates a trace exception after the instruction completes.

Note that if the processor is executing in trace mode when a group 2 exception is signaled, a trace exception will not be generated. This means that for the second example, as the TRAP exception handler completes its execution and performs its RTE, the next instruction of the program sequence will be executed before the next trace exception is performed (the MC68060 will not trace immediately after the TRAP). If tracing is required immediately following a group 2 exception, the SR contained in the exception stack frame should be checked before returning to the next instruction. If the stacked SR indicates that the processor was executing in trace mode, the trace handler should be executed to account for the instruction that initiated the exception.

7.3.7 Debug Interrupt

This special type of program interrupt is reserved for use with the Debug module. 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 $030).

7.3.8 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 the ColdFire 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 format value corresponds to the required system stack pointer adjustment to return it to the value at the time of the exception. This mechanism essentially negates the self-aligning operation to restore the stack pointer to its original value.

The selection of the format value provides some limited debug support for porting code from 68000 applications. On 680x0 family processors, the status register (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, an attempted RTE using this “old” format generates a format error on the ColdFire processor.

If the format field defines a valid type, the processor: (1) reloads the status register operand, (2) adjusts the stack pointer by adding the format value's two least significant bits to the auto-incremented value after the fetch of the first longword, and (3) transfers control to the instruction address defined by the program counter field within the stack frame.

In essence, the RTE restores the stack pointer to its original value, whether or not the stack frame was misaligned at the time of exception.

7.3.9 Interrupt Exception

When a peripheral device requires the services of the ColdFire processor or is ready to send information that the processor requires, it can signal the processor to take an interrupt exception. The exact hardware for reporting these interrupts is implementation-specific. The ColdFire architecture defines seven priority levels.

When an interrupt request has a priority higher than the value in the interrupt priority mask of the SR (bits 10–8), the processor makes the request a pending interrupt. Priority level 7, the nonmaskable interrupt, is a special case. The transition-sensitive Level 7 interrupts cannot be masked by the interrupt priority mask. The processor recognizes an interrupt request each time the external interrupt request level changes from some lower level to level 7, regardless of the value in the mask. Figure 7-3 shows two examples of interrupt recognitions, one for level 6 and one for level 7.

When the ColdFire processes a level 6 interrupt, the SR mask is automatically updated with a value of 6 before entering the handler routine so that subsequent level 6 interrupts and lower-level interrupts are masked. Provided no instruction that lowers the mask value is executed, the external request can be lowered to level 3 and then raised back to level 6 and a second level 6 interrupt is not processed.

However, if the processor is handling a level 7 interrupt (SR mask set to level 7) and the external request is lowered to level 3 and than raised back to level 7, a second level 7 interrupt is processed. The second level 7 interrupt is processed because the level 7 interrupt is transition-sensitive. A level comparison also generates a level 7 interrupt if the request level and mask level are at 7 and the priority mask is then set to a lower level (as

with the MOVE to SR or RTE instruction). The level 6 interrupt request and mask level example in Figure 7-3 is the same as for all interrupt levels except level 7.

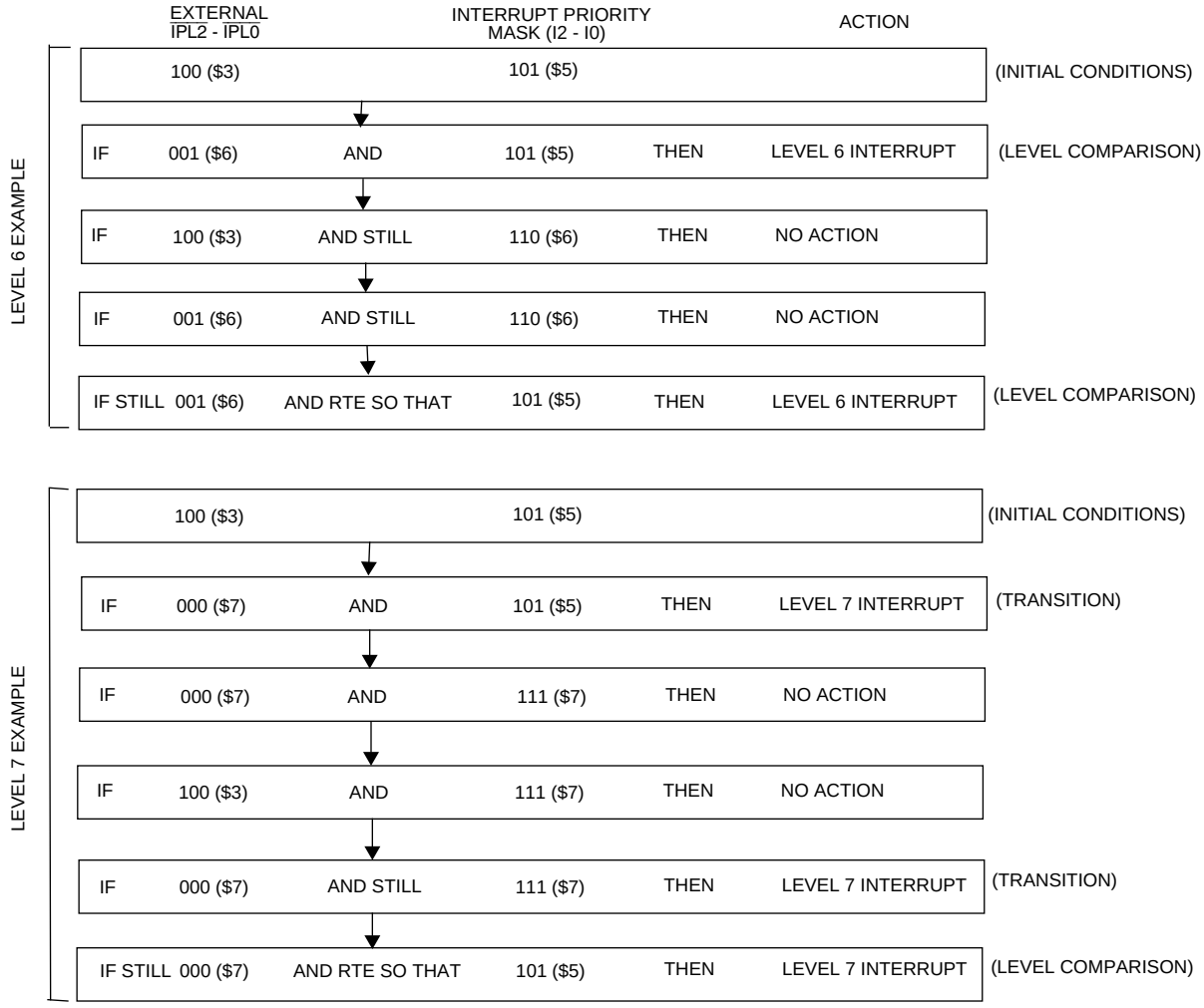


Figure 7-3. Interrupt Recognition Examples

Note that a mask value of 6 and a mask value of 7 both inhibit request levels of 1–6 from being recognized. In addition, neither masks an interrupt request level of 7. The only difference between mask values of 6 and 7 occurs when the interrupt request level is 7 and the mask value is 7. If the mask value is lowered to 6, a second level 7 interrupt is recognized.

External circuitry can chain or otherwise merge signals from devices at each level, allowing an unlimited number of devices to interrupt the processor. When several devices are connected to the same interrupt level, each device should hold its interrupt priority level constant until its corresponding interrupt acknowledge bus cycle ensures that all requests are processed. The interrupt acknowledge cycle is implementation-specific and is documented in the appropriate processor user's manual.

Figure 7-1 illustrates a flowchart for interrupt exception processing. When processing an interrupt exception, the processor first makes an internal copy of the SR, sets the mode to supervisor, suppresses tracing, and sets the processor interrupt mask level to the level of

the interrupt being serviced. The processor attempts to obtain a vector number from the interrupting device using an interrupt-acknowledge bus cycle with the interrupt level number output on the transfer modifier signals.

To support external devices that cannot provide an interrupt vector, the autovector signal must be asserted. The autovector feature of the ColdFire processor is implementation-dependent. If the specific implementation supports autovectoring, it is done through an optional module.

In the autovector case, the ColdFire processor uses an internally generated autovector (one of vector numbers 25–31) that corresponds to the interrupt level number. If external logic indicates a bus error during the interrupt acknowledge cycle, the interrupt is considered spurious, and the processor generates the spurious interrupt vector number, 24.

Interrupt sampling is deferred from the beginning of exception processing of any exception, up to and until the first instruction of the exception handler. This allows the first instruction of any exception handler to raise the interrupt mask level and execute the exception handler without interrupts (except level 7 interrupts).

Many M68000 Family peripherals use programmable interrupt vector numbers as part of the interrupt-acknowledge operation for the system. If this vector number is not initialized after reset and the peripheral must acknowledge an interrupt request, the peripheral usually returns the vector number for the uninitialized interrupt vector, 15.

7.3.10 Fault-on-Fault Halt

If the ColdFire processor encounters any access error or address error 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.

7.3.11 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 sets the processor’s interrupt priority mask in the SR to the highest level (level 7). Next, the VBR is initialized to zero ($00000000), and any caches in the specific ColdFire implementation are disabled.

Other implementation-specific supervisor registers are affected as well. Refer to the specific user’s manual for details.

If the processor is granted the bus, and it does not detect any other alternate masters taking the bus, the processor then performs two long-word-read bus cycles. The first long word, at address 0, is loaded into the stack pointer, and the second long word, at address 4, is loaded into the program counter. After the initial instruction is fetched from memory, program exe-

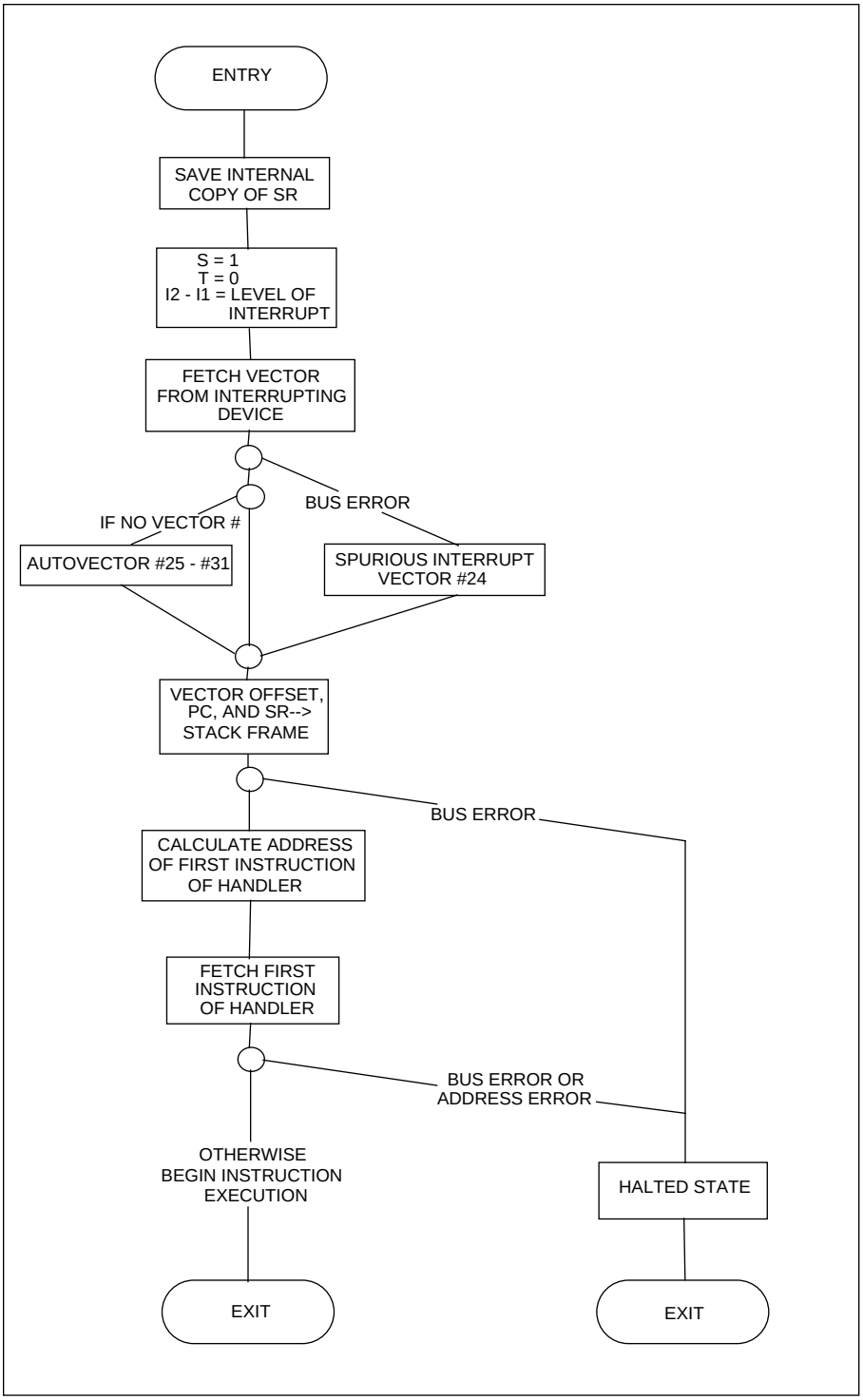


Figure 7-4. Interrupt Exception Processing Flowchart

cution begins at the address in the PC. If an access error or address error occurs before the first instruction is executed, the processor goes into a halted state.

7.4 EXCEPTION PRIORITIES

Exceptions can be divided into the five basic groups identified in Table 7-4. These groups are defined by specific characteristics and the order in which they are handled. Table 7-4 represents the priority used for simultaneous faults, as viewed by the ColdFire hardware. In Table 7-4, 0.0 represents the highest priority, while 3.1 is the lowest.

Table 7-4. Exception Priority Groups

Group.Priority	Exception and Relative Priority	Characteristics
0.0	Reset	The processor aborts all processing (instruction or exception) and does not save old context.
1.0 1.1 1.2	Address Error Instruction Access Error Data Access Error	The processor suspends processing and saves the processor context.
2.0 2.1 2.2 2.3	A-Line Unimplemented F-Line Unimplemented Illegal Instruction Privilege Violation	Exception processing begins before the instruction is executed.
3.0 3.1	Trace Interrupt	Exception processing begins when the current instruction is completed.

Within a ColdFire system, more than one exception can occur at the same time. The reset exception is unique; a reset overrides and clears all other exceptions that may have occurred at the same time. All other exceptions are handled according to the priority relationship defined in Table 7-4.

In general, when multiple exceptions are pending, the exception with the highest priority is processed first, and the remaining exceptions are regenerated when the original faulting instruction is restarted.

To illustrate the handling of multiple exceptions, consider first a pending interrupt being posted while a program is executing in trace mode (i.e., bit 15 of the SR is set).

Since the processor always samples for pending interrupts and traces at the conclusion of instruction execution, both the trace and the interrupt appear simultaneous to the processor. Because the trace has higher priority than the interrupt (3.0 versus 3.1), trace exception processing begins. After the first instruction of the trace exception handler has been executed, the processor again samples for pending interrupts. Providing the previous interrupt is still pending, the processor now begins interrupt-exception processing. As the interrupt handler completes execution, control returns to the trace handler. As the trace handler completes, control returns to the original program.

As a second example of the handling of multiple exceptions, consider the prior scenario (a pending interrupt being posted while a program is executing in trace mode) at the same time a TRAP instruction enters the execution unit.

As described before, because the processor always samples for pending interrupts and traces at the conclusion of instruction execution, both the trace and the interrupt appear simultaneous to the processor. Because the trace has higher priority than the interrupt, trace exception processing begins. After the first instruction of the trace exception handler has been executed, the processor again samples for pending interrupts. Providing the previous

interrupt is still pending, the processor begins interrupt exception processing. As the interrupt handler completes execution, control returns to the trace handler. As the trace handler completes, control returns to the original program, where the TRAP instruction is executed, causing that exception to occur.

Note that if the processor is executing in trace mode when a group 2 exception is signaled, a trace exception will not be generated. This means that for the second example, as the TRAP exception handler completes its execution and performs its RTE, the next instruction of the program sequence will be executed before the next trace exception is performed (the ColdFire processor will not trace immediately after the TRAP). If tracing is required immediately following a group 2 exception, the SR contained in the exception stack frame should be checked before returning to the next instruction. If the stacked SR indicates that the processor was executing in trace mode, the trace handler should be executed to account for the instruction that initiated the exception.

Considering the previous example, the TRAP handler should check the stacked SR, and because the processor was executing in trace mode, pass control to the trace handler. If this check is not made, the next trace exception will not occur until the instruction after the TRAP has completed execution.

SECTION 8
S-RECORD OUTPUT FORMAT

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.

8.1 S-RECORD CONTENT

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

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.

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.

8.2 S-RECORD TYPES

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 utilize 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.

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 field may 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.

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 input will be used. There is no code/data field.

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.

8.3 S-RECORD CREATION

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. 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
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.

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	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

Figure 8-2. Transmission of an S1 Record

Table 8-2. ASCII Code

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)

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.

9.1 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.

9.2 MOVE INSTRUCTION EXECUTION TIMES

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)	—	—	—

9.3 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)

9.4 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)	—	—	—	—	—	—	—

9.5 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)

9.6 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

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
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
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

Table A-1. ColdFire Instruction Set (Continued)

Mnemonic	Description
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
WDDATA	Write Data Control Register
WDEBUG	Write Debug Control Register