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

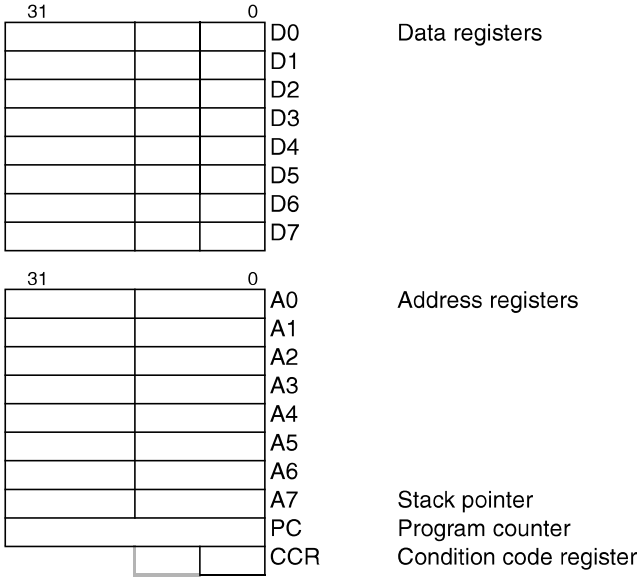


Figure 1-1. ColdFire Family User Programming Model

1.1.1 Data Registers (D0–D7)

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

1.1.2 Address Registers (A0–A7)

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

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 and indicate the

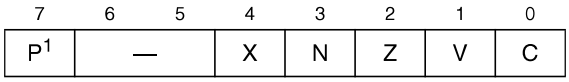
instruction's result. Program and system control instructions also use certain combinations of these bits to control program and system flow.

The condition codes meet two criteria:

1. Consistency across:
   — Instructions, meaning that all instructions that are special cases of more general instructions affect the condition codes in the same way;
   — Uses, meaning that conditional instructions test the condition codes similarly and provide the same results whether a compare, test, or move instruction sets the condition codes; and
   — Instances, meaning that all instances of an instruction affect the condition codes in the same way.
2. Meaningful results with no change unless it provides useful information.

Bits [3:0] represent a condition of the result generated by an operation. Bit 5, the extend bit, is an operand for multiprecision computations. Version 3 processors have an additional bit in the CCR: bit 7, the branch prediction bit.

The CCR is illustrated in Figure 1-2.



[1]The P bit is implemented only on the V3 core.

Figure 1-2. Condition Code Register (CCR)

Table 1-1 describes CCR bits.

Table 1-1. CCR Bit Descriptions

Bits	Field	Description
7	P	Branch prediction (Version 3 only). Alters the static prediction algorithm used by the branch acceleration logic in the instruction fetch pipeline on forward conditional branches. Refer to a V3 core or device user's manual for further information on this bit.
	—	Reserved; should be cleared (all other versions).
6–5	—	Reserved, should be cleared.
4	X	Extend. Set to the value of the C-bit for arithmetic operations; otherwise not affected or set to a specified result.
3	N	Negative. Set if the most significant bit of the result is set; otherwise cleared.
2	Z	Zero. Set if the result equals zero; otherwise cleared.
1	V	Overflow. Set if an arithmetic overflow occurs implying that the result cannot be represented in the operand size; otherwise cleared.
0	C	Carry. Set if a carry out of the most significant bit of the operand occurs for an addition, or if a borrow occurs in a subtraction; otherwise cleared.

1.2 Floating-Point Unit User Programming Model

The following paragraphs describe the registers for the optional ColdFire floating-point unit. Figure 1-3 illustrates the user programming model for the floating-point unit. It contains the following registers:

When used by an instruction, MASK is ANDed with the specified operand address. Thus, MASK allows an operand address to be effectively constrained within a certain range defined by the 16-bit value. This feature minimizes the addressing support required for filtering, convolution, or any routine that implements a data array as a circular queue using the (Ay)+ addressing mode.

For MAC with load operations, the MASK contents can optionally be included in all memory effective address calculations.

1.5 Supervisor Programming Model

System programmers use the supervisor programming model to implement operating system functions. All accesses that affect the control features of ColdFire processors must be made in supervisor mode. The following paragraphs briefly describe the supervisor registers, which can be accessed only by privileged instructions. The supervisor programming model consists of the registers available to users as well as the registers listed in Figure 1-14.

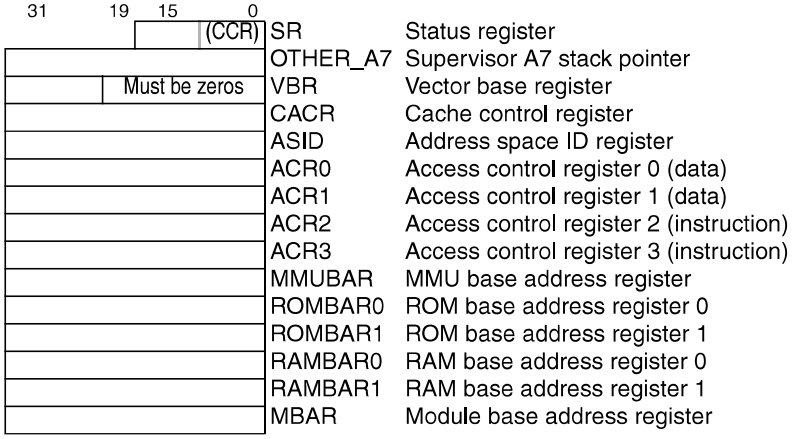


Figure 1-14. Supervisor Programming Model

Note that not all registers are implemented on every ColdFire device; refer to Table 1-6. Future devices may include registers not implemented on earlier devices.

Table 1-6. Implemented Supervisor Registers by Device

Name	V2	V3	V4	V5
SR	x	x	x	x
OTHER_A7	if ISA_A+	if ISA_A+	x	x
VBR	x	x	x	x
CACR	x	x	x	x
ASID			if MMU	if MMU
ACR0	x	x	x	x
ACR1	x	x	x	x
ACR2			x	x
ACR3			x	x
MMUBAR			if MMU	if MMU

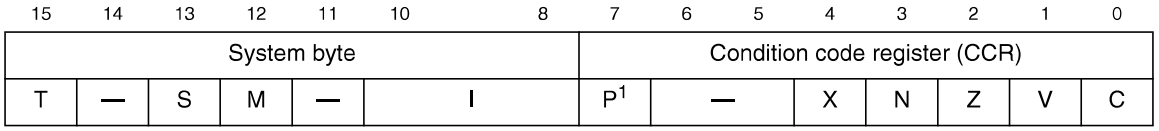
Table 1-6. Implemented Supervisor Registers by Device (Continued)

Name	V2	V3	V4	V5
ROMBAR0	DS	DS	DS	DS
ROMBAR1	DS	DS	DS	DS
RAMBAR0	DS	DS	DS	DS
RAMBAR1	DS	DS	DS	DS
MBAR	DS	DS	DS	DS

Note: “x” indicated the supervisor register is implemented. DS indicates the supervisor register is “device-specific”. Please consult the appropriate device reference manual to determine if the register is implemented. Certain supervisor registers are present only if the virtual memory management unit (MMU) is implemented (“if MMU”). Certain supervisor registers are present only if the implemented instruction set architecture is ISA_A+ (“if ISA_A+”).

1.5.1 Status Register (SR)

The SR, shown in Figure 1-15, stores the processor status, the interrupt priority mask, and other control bits. Supervisor software can read or write the entire SR; user software can read or write only SR[7–0], described in Section 1.1.4, “Condition Code Register (CCR).” The control bits indicate processor states: trace mode (T), supervisor or user mode (S), and master or interrupt state (M). SR is set to 0x27*xx* after reset.The SR register must be explicitly loaded after reset and before any compare, Bcc, or Scc instructions are executed.



[1]The P bit is implemented only on the V3 core.

Figure 1-15. Status Register (SR)

Table 1-7 describes SR fields.

Table 1-7. Status Field Descriptions

Bits	Name	Description
15	T	Trace enable. When T is set, the processor performs a trace exception after every instruction.
14	—	Reserved, should be cleared.
13	S	Supervisor/user state. Indicates whether the processor is in supervisor or user mode
12	M	Master/interrupt state. Cleared by an interrupt exception. It can be set by software during execution of the RTE or move to SR instructions so the OS can emulate an interrupt stack pointer.
11	—	Reserved, should be cleared.
10–8	I	Interrupt priority mask. Defines the current interrupt priority. Interrupt requests are inhibited for all priority levels less than or equal to the current priority, except the edge-sensitive level-7 request, which cannot be masked.
7–0	CCR	Condition code register (see Figure 1-2 and Table 1-1)

1.5.2 Supervisor/User Stack Pointers (A7 and OTHER_A7)

The ISA_A architectures support a single stack pointer (A7). The initial value of A7 is loaded from the reset exception vector, address offset 0.

All remaining ISA revisions support two independent stack pointer (A7) registers: the supervisor stack pointer (SSP) and the user stack pointer (USP). This support provides the required isolation between operating modes (supervisor and user).

The hardware implementation of these two programmable-visible 32-bit registers does not uniquely identify one as the SSP and the other as the USP. Rather, the hardware uses one 32-bit register as the currently active A7 and the other as OTHER_A7. Thus, the register contents are a function of the processor operating mode, as shown in the following:

```
if SR[S] = 1
        then
                A7 = Supervisor Stack Pointer
                other_A7 = User Stack Pointer
else
        A7 = User Stack Pointer
        other_A7 = Supervisor Stack Pointer
```

1.5.3 Vector Base Register (VBR)

The vector base register contains the 1 MByte-aligned 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. VBR[19–0] are filled with zeros.

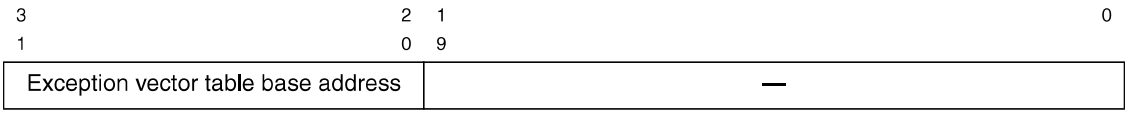


Figure 1-16. Vector Base Register (VBR)

1.5.4 Cache Control Register (CACR)

The CACR controls operation of both the instruction and data cache memory. It includes bits for enabling, locking, and invalidating cache contents. It also includes bits for defining the default cache mode and write-protect fields. Bit functions and positions may vary among ColdFire processor implementations. Refer to a specific device or core user's manual for further information.

1.5.5 Address Space Identifier (ASID)

Only the low-order 8 bits of the 32-bit ASID register are implemented. The ASID value is an 8-bit identifier assigned by the operating system to each process active in the system. It effectively serves as an extension to the 32-bit virtual address. Thus, the virtual reference now becomes a 40-bit value: the 8-bit ASID concatenated with the 32-bit virtual address. ASID is only available if a device has an MMU. Refer to a specific device or core user's manual for further information.

1.5.10 Module Base Address Register (MBAR)

The supervisor-level MBAR, Figure 1-18, specifies the base address and allowable access types for all internal peripherals. MBAR can be read or written through the debug module as a read/write register; only the debug module can read MBAR. All internal peripheral registers occupy a single relocatable memory block along 4-Kbyte boundaries. MBAR masks specific address spaces using the address space fields. Refer to a specific device or core user's manual for further information.

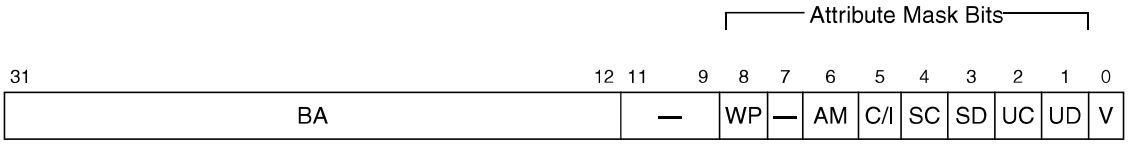


Figure 1-18. Module Base Address Register (MBAR)

Table 1-9 describes MBAR fields.

Table 1-9. MBAR Field Descriptions

Bits	Field	Description	
31–12	BA	Base address. Defines the base address for a 4-Kbyte address range.	
11–9	—	Reserved, should be cleared.	
8–1	AMB	Attribute mask bits	
8		WP	Write protect. Mask bit for write cycles in the MBAR-mapped register address range
7		—	Reserved, should be cleared.
6		AM	Alternate master mask
5		C/I	Mask CPU space and interrupt acknowledge cycles
4		SC	Setting masks supervisor code space in MBAR address range
3		SD	Setting masks supervisor data space in MBAR address range
2		UC	Setting masks user code space in MBAR address range
1		UD	Setting masks user data space in MBAR address range
0	V	Valid. Determines whether MBAR settings are valid.	

1.6 Integer Data Formats

The operand data formats are supported by the integer unit, as listed in Table 1-10. 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 1-10. Integer Data Formats

Operand Data Format	Size
Bit	1 bit
Byte integer	8 bits

Table 1-10. Integer Data Formats (Continued)

Operand Data Format	Size
Word integer	16 bits
Longword integer	32 bits

1.7 Floating-Point Data Formats

This section describes the optional FPU's operand data formats. The FPU supports three signed integer formats (byte, word, and longword) that are identical to those supported by the integer unit. The FPU also supports single- and double-precision binary floating-point formats that fully comply with the IEEE-754 standard.

1.7.1 Floating-Point Data Types

Each floating-point data format supports five unique data types: normalized numbers, zeros, infinities, NANs, and denormalized numbers. The normalized data type, Figure 1-19, never uses the maximum or minimum exponent value for a given format.

1.7.1.1 Normalized Numbers

Normalized numbers include all positive or negative numbers with exponents between the maximum and minimum values. For single- and double-precision normalized numbers, the implied integer bit is one and the exponent can be zero.

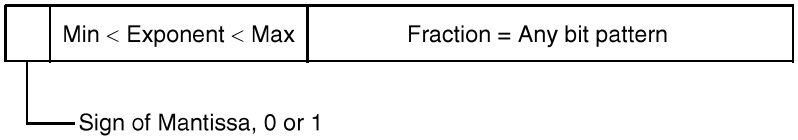


Figure 1-19. Normalized Number Format

1.7.1.2 Zeros

Zeros can be positive or negative and represent real values, + 0.0 and – 0.0. See Figure 1-20.

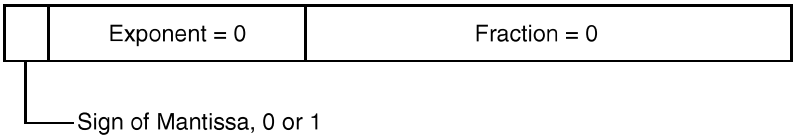


Figure 1-20. Zero Format

1.7.1.3 Infinities

Infinities can be positive or negative and represent real values that exceed the overflow threshold. A result's exponent greater than or equal to the maximum exponent value indicates an overflow for a given data format and operation. This overflow description ignores the effects of rounding and the user-selectable rounding models. For single- and double-precision infinities, the fraction is a zero. See Figure 1-21.

Chapter 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 operation word or opword, 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 operation word, branch displacements, bit number or special register specifications, trap operands, argument counts, or floating-point command words. The ColdFire architecture instruction word length is limited to 3 sizes: 16, 32, or 48 bits.

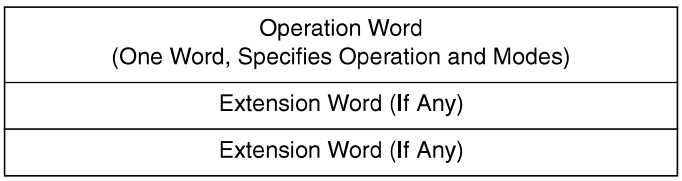


Figure 2-1. Instruction Word General Format

An instruction specifies the function to be performed with an operation code and defines the location of every operand. The 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 = 111. Some indexed or indirect addressing modes use a combination of the 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.

Operation Word Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	X	X	X	X	X	X	Effective Address					
										Mode			Register		

Extension Word Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D/A	Register			W/L	Scale		0	Displacement							

Figure 2-2. Instruction Word Specification Formats

Table 2-1 defines instruction word formats.

Table 2-1. Instruction Word Format Field Definitions

Field	Definition
Instruction	
Mode	Addressing mode (see Table 2-3)
Register	General register number (see Table 2-3)
Extensions	
D/A	Index register type 0 = D*n* 1 = A*n*
W/L	Word/longword index size 0 = Address Error Exception 1 = Longword
Scale	Scale factor 00 = 1 01 = 2 10 = 4 11 = 8 (supported only if FPU is present)

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 one of the three following ways:

- A register field within an instruction can specify the register to be used.
- An instruction's effective address field can contain addressing mode information.
- 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.

Generation	EA = Dn
Assembler Syntax	Dn
EA Mode Field	000
EA Register Field	Register number
Number of Extension Words	0

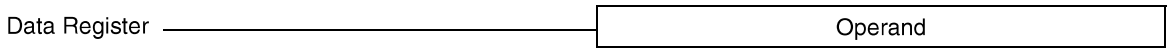


Figure 2-3. Data Register Direct

2.2.2 Address Register Direct Mode

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

Generation	EA = An
Assembler Syntax	An
EA Mode Field	001
EA Register Field	Register number
Number of Extension Words	0

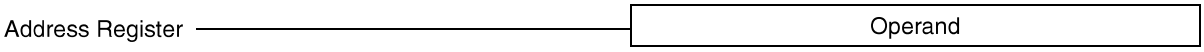


Figure 2-4. Address Register Direct

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.

Generation	EA = (An)
Assembler Syntax	(An)
EA Mode Field	010
EA Register Field	Register number
Number of Extension Words	0

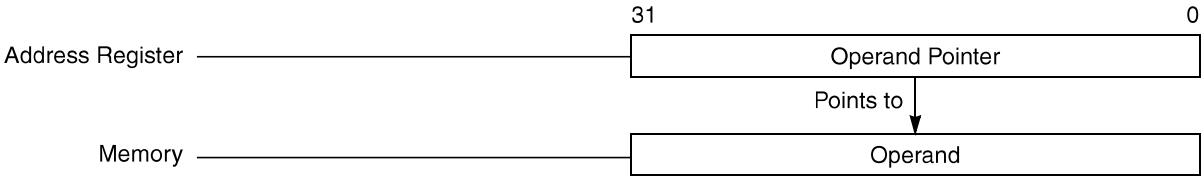


Figure 2-5. Address Register Indirect

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

Generation	EA = (An); An = An + Size
Assembler Syntax	(An)+
EA Mode Field	011
EA Register Field	Register number
Number of Extension Words	0

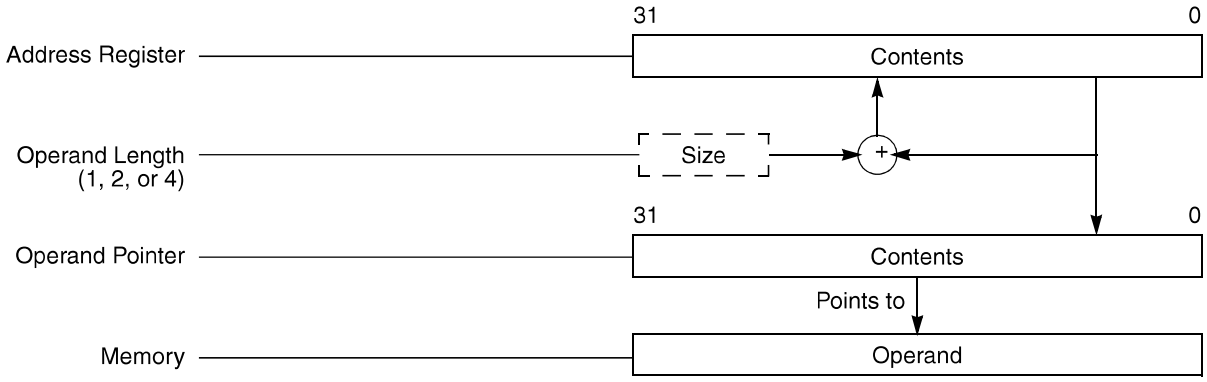


Figure 2-6. Address Register Indirect with Postincrement

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

Generation	EA = (An) - Size; An = An - Size;
Assembler Syntax	–(An)
EA Mode Field	100
EA Register Field	Register number
Number of Extension Words	0

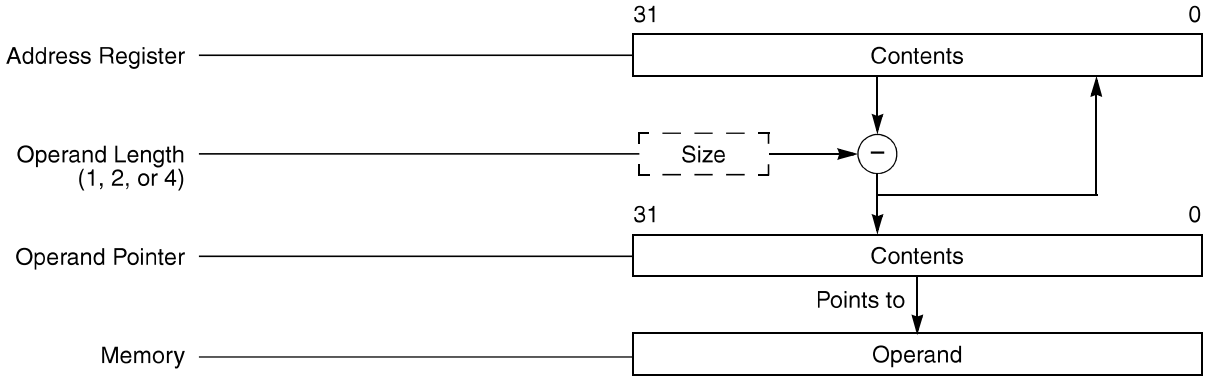


Figure 2-7. Address Register Indirect with Predecrement

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.

Generation	EA = (An) + d_{16}
Assembler Syntax	(d_{16},An)
EA Mode Field	101
EA Register Field	Register number
Number of Extension Words	1

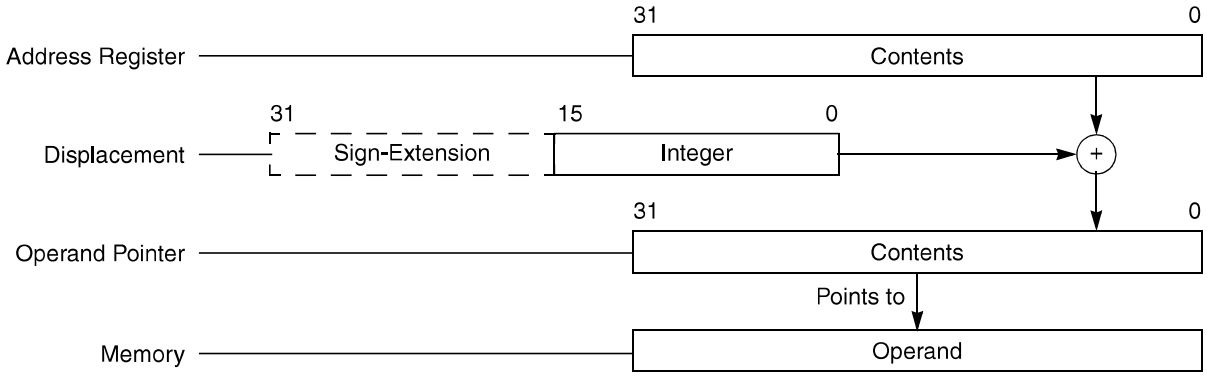


Figure 2-8. Address Register Indirect with Displacement

2.2.7 Address Register Indirect with Scaled Index and 8-Bit Displacement Mode

This addressing mode requires one extension word that contains an index register indicator, possibly scaled, 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 scaled index register's sign-extended contents. Users must specify the address register, the displacement, the scale factor and the index register in this mode.

Generation	EA = (An) + ((Xi) * ScaleFactor)) + Sign-extended d_8
Assembler Syntax	(d_8,An,Xi,Size*Scale)
EA Mode Field	110
EA Register Field	Register number
Number of Extension Words	1

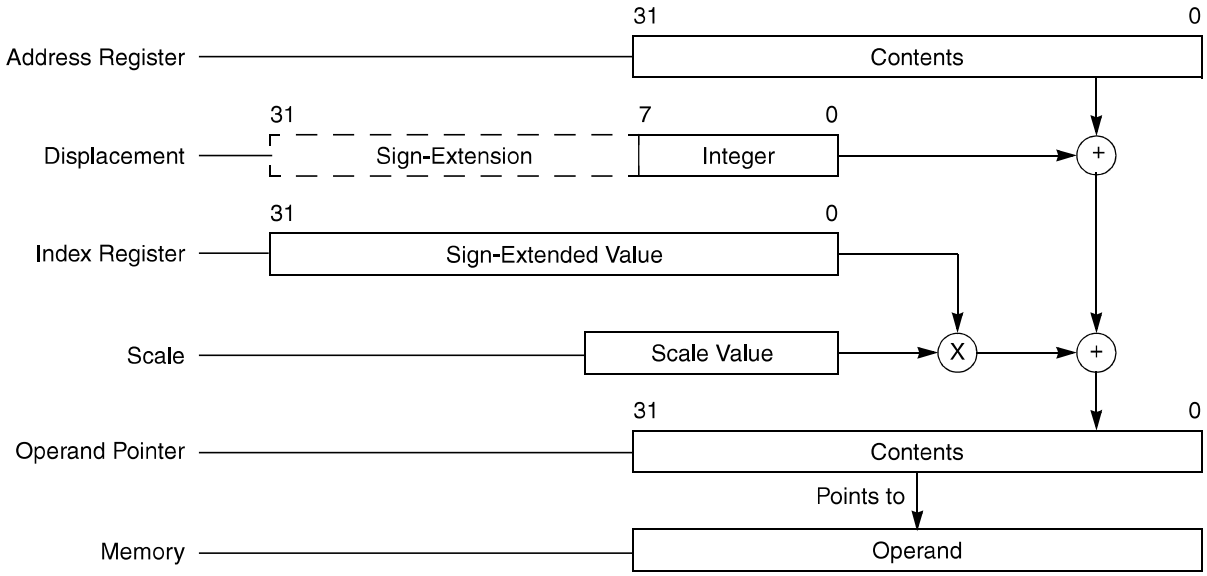


Figure 2-9. Address Register Indirect with Scaled Index and 8-Bit Displacement

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 at the time of address generation is PC+2, where PC is the address of the instruction operation word. This is a program reference allowed only for reads.

Generation	EA = (PC) + d_{16}
Assembler Syntax	(d_{16},PC)
EA Mode Field	111
EA Register Field	010
Number of Extension Words	1

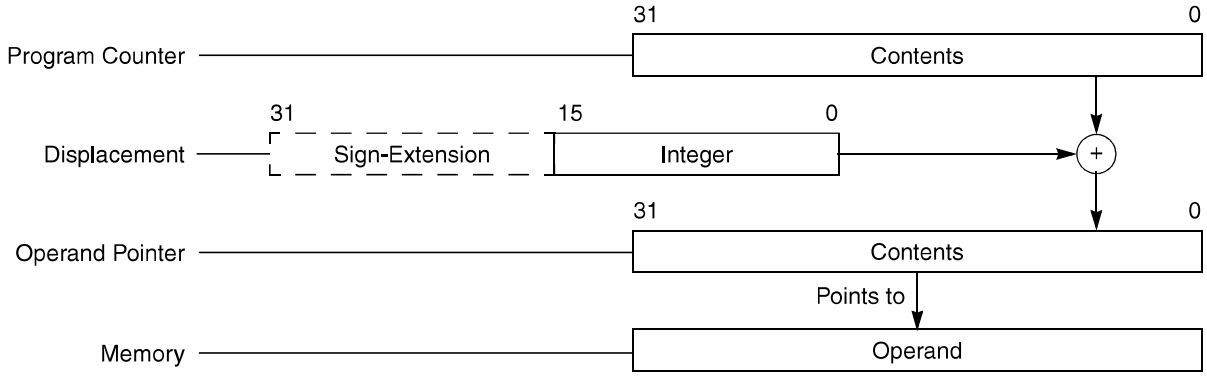


Figure 2-10. Program Counter Indirect with Displacement

2.2.9 Program Counter Indirect with Scaled Index and 8-Bit Displacement Mode

This mode is similar to the mode described in Section 2.2.7, "Address Register Indirect with Scaled Index and 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 at the time of address generation is PC+2, where PC is the address of the instruction operation word. This is a program reference allowed only for reads. Users must include the displacement, the scale, and the index register when specifying this addressing mode.

Generation	EA = (PC) + ((Xi) * ScaleFactor)) + Sign-extended d_8
Assembler Syntax	(d_8,PC,Xi,Size*Scale)
EA Mode Field	111
EA Register Field	011
Number of Extension Words	1

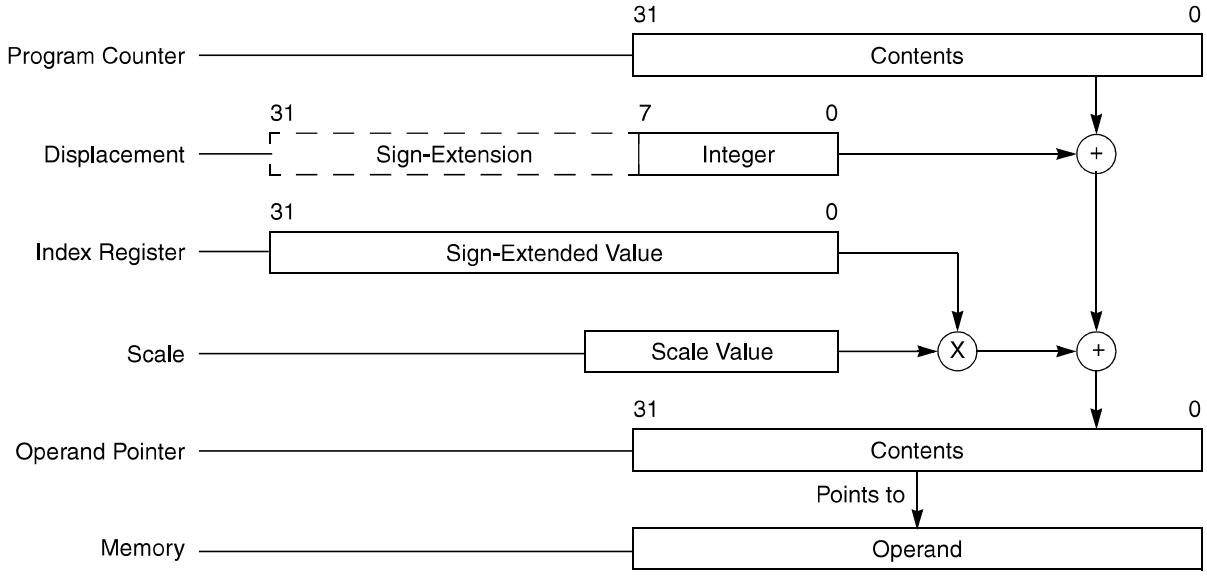


Figure 2-11. Program Counter Indirect with Scaled Index and 8-Bit Displacement

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.

Generation	EA Given
Assembler Syntax	(xxx).W
EA Mode Field	111
EA Register Field	000
Number of Extension Words	1

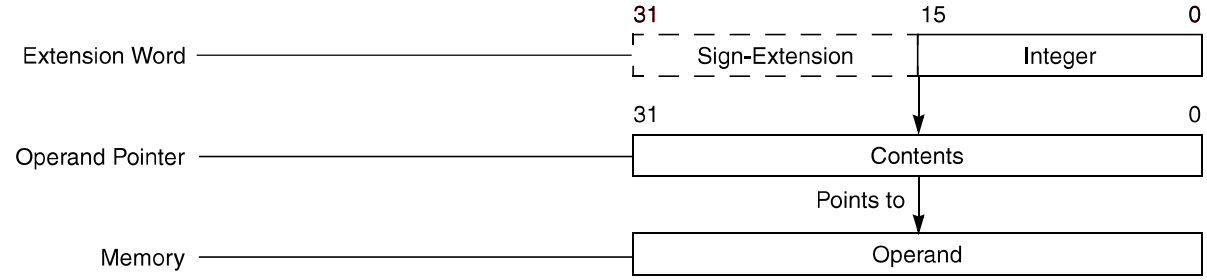


Figure 2-12. Absolute Short Addressing

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).L
EA Mode Field	111
EA Register Field	001
Number of Extension Words	2

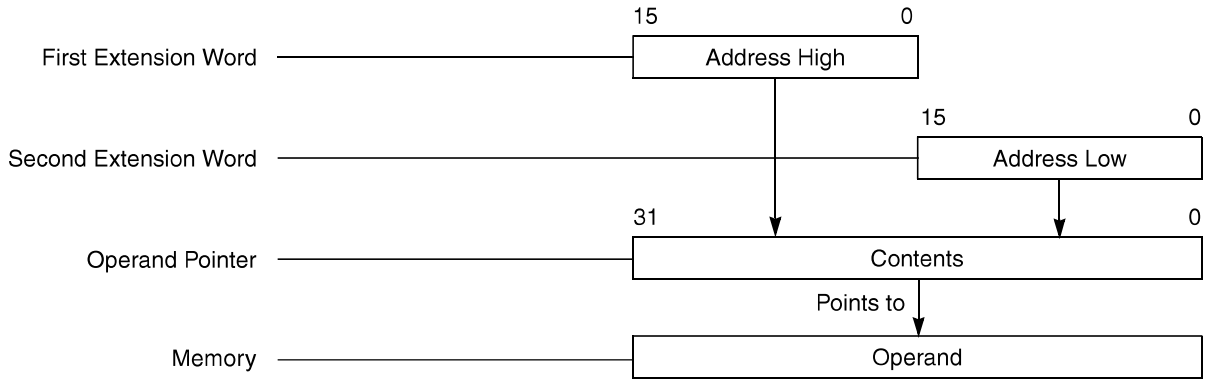


Figure 2-13. Absolute Long Addressing

2.2.12 Immediate Data

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

Figure 2-14. Immediate Data Addressing

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 2-3 lists a summary of effective addressing modes and their categories.

Table 2-3. Effective Addressing Modes and Categories

Addressing Modes	Syntax	Mode Field	Reg. Field	Data	Memory	Control	Alterable
Register Direct							
Data	Dn	000	reg. no.	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 Scaled Index and 8-Bit Displacement	(d_8,An,Xi*SF)	110	reg. no.	X	X	X	X
Program Counter Indirect with Displacement	(d_{16},PC)	111	010	X	X	X	—
Program Counter Indirect with Scaled Index and 8-Bit Displacement	(d_8,PC,Xi*SF)	111	011	X	X	X	—
Absolute Data Addressing							
Short	(xxx).W	111	000	X	X	X	—
Long	(xxx).L	111	001	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, and parameter passing and is affected by instructions such as the LINK, UNLK, RTE, and PEA. To maximize performance, A7 must be longword-aligned at all times. Therefore, when modifying A7, be sure to do so in multiples of 4 to maintain alignment. To further 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. Users should keep in mind these important directives:

- Use the predecrement mode to decrement the register before using its contents as the pointer to the stack.
- Use the postincrement mode to increment the register after using its contents as the pointer to the stack.

- Maintain the stack pointer correctly when byte, word, and longword items mix in these stacks.

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.

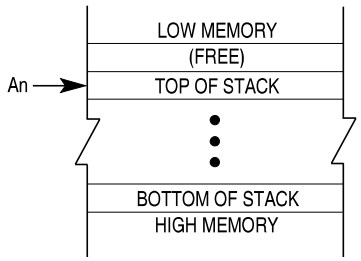


Figure 2-15. Stack Growth from High Memory to Low Memory

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.

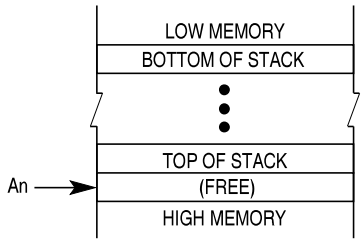


Figure 2-16. Stack Growth from Low Memory to High Memory

Chapter 3
Instruction Set Summary

This section briefly describes the ColdFire Family instruction set, using Freescale's assembly language syntax and notation. It includes instruction set details such as notation and format.

3.1 Instruction Summary

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

- Data movement
- Program control
- Integer arithmetic
- Floating-point arithmetic
- Logical operations
- Shift operations
- Bit manipulation
- System control
- Cache maintenance

NOTE

The DIVS/DIVU and RES/REMU instructions are not implemented on the earliest V2-based devices MCR5202, MCF5204 and MCF5206.

Although MCF5407 device implements ISA_B, support for the User Stack Pointer (Move to/from USP) is not provided.

Table 3-14 shows the entire user instruction set in alphabetical order. Table 3-15 shows the entire supervisor instruction set in alphabetical order.

The following paragraphs 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
~	Invert; operand is logically complemented
&	Logical AND
|	Logical OR
⊥	Logical exclusive OR
→	Source operand is moved to destination operand

Table 3-1. Notational Conventions (Continued)

←→	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	
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	
A*n*	Any address register *n* (example: A3 is address register 3)
Ax, Ay	Destination and source address registers, respectively
D*n*	Any data register *n* (example: D5 is data register 5)
Dx, Dy	Destination and source data registers, respectively
Dw	Data register containing a remainder
Rc	Control register
R*n*	Any address or data register
Rx, Ry	Any destination and source registers, respectively
Xi	Index register, can be any address or data register; all 32-bits are used.
Subfields and Qualifiers	
#<data>	Immediate data following the instruction word(s).
( )	Identifies an indirect address in a register.
d_n	Displacement value, *n* bits wide (example: d_{16} is a 16-bit displacement).
sz	Size of operation: Byte (B), Word (W), Longword (L)
lsb, msb	Least significant bit, most significant bit
LSW, MSW	Least significant word, most significant word
SF	Scale factor for an index register
Register Names	
CCR	Condition Code Register (lower byte of status register)
PC	Program Counter
SR	Status Register
USP	User Stack Pointer
ic, dc, bc	Instruction, data, or both caches (unified cache uses bc)
Condition Codes	

Table 3-1. Notational Conventions (Continued)

*	General case
C	Carry bit in CCR
cc	Condition codes from CCR
N	Negative bit in CCR
V	Overflow bit in CCR
X	Extend bit in CCR
Z	Zero bit in CCR
—	Not affected or applicable
Miscellaneous	
<ea>x, <ea>y	Destination and source effective address, respectively
<label>	Assembly program label
#list	List of registers, for example D3–D0
MAC Operations	
ACC, ACCx	MAC accumulator register, a specific EMAC accumulator register
ACCx, ACCy	Destination and source accumulators, respectively
ACCext01	Four extension bytes associated with EMAC accumulators 0 and 1
ACCext23	Four extension bytes associated with EMAC accumulators 2 and 3
EV	Extension overflow flag in MACSR
MACSR	MAC status register
MASK	MAC mask register
PAVx	Product/accumulation overflow flags in MACSR
RxSF	A register containing a MAC operand that is to be scaled
Rw	Destination register for a MAC with load operation
Floating-Point Operations	
fmt	Format of operation: Byte (B), Word (W), Longword (L), Single-precision (S), Double-precision(D)
+inf	Positive infinity
-inf	Negative infinity
FPx, FPy	Destination and source floating-point data registers, respectively
FPCR	Floating-point control register
FPIAR	Floating-point instruction address register
FPSR	Floating-point status register
NAN	Not-a-number

3.1.1 Data Movement Instructions

The MOVE and FMOVE instructions with their associated addressing modes are the basic means of transferring and storing addresses and data. MOVE instructions transfer byte, word, and longword operands from memory to memory, memory to register, register to memory, and register to register. MOVEA instructions transfer word and longword operands and ensure that only valid address manipulations are executed. In addition to the general MOVE instructions, there are several special data movement instructions: MOV3Q, MOVEM, MOVEQ, MVS, MVZ, LEA, PEA, LINK, and UNLK. MOV3Q, MVS, and MVZ are ISA_B additions to the instruction set.

The FMOVE instructions move operands into, out of, and between floating-point data registers. FMOVE also moves operands to and from the FPCR, FPIAR, and FPSR. For operands moved into a floating-point data register, FSMOVE and FDMOVE explicitly select single- and double-precision rounding of the result. FMOVEM moves any combination of floating-point data registers. Table 3-2 lists the general format of these integer and floating-point data movement instructions.

Table 3-2. Data Movement Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First Appeared: ISA, (E)MAC, or FPU
FDMOVE	FPy,FPx	D	Source → Destination; round destination to double	FPU
FMOVE	<ea>y,FPx FPy,<ea>x FPy,FPx FPcr,<ea>x <ea>y,FPcr	B,W,L,S,D B,W,L,S,D D L L	Source → Destination FPcr can be any floating-point control register: FPCR, FPIAR, FPSR	FPU
FMOVEM	#list,<ea>x <ea>y,#list	D	Listed registers → Destination Source → Listed registers	FPU
FSMOVE	<ea>y,FPx	B,W,L,S,D	Source → Destination; round destination to single	FPU
LEA	<ea>y,Ax	L	<ea>y → Ax	ISA_A
LINK	Ay,#<displacement>	W	SP – 4 → SP; Ay → (SP); SP → Ay, SP + d_n → SP	ISA_A
MOV3Q	#<data>,<ea>x	L	Immediate Data → Destination	ISA_B
MOVCLR	ACCy,Rx	L	Accumulator → Destination, 0 → Accumulator	EMAC
MOVE MOVE from CCR MOVE to CCR	<ea>y,<ea>x MACcr,Dx <ea>y,MACcr CCR,Dx <ea>y,CCR	B,W,L L L W W	Source → Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK	ISA_A[1] MAC[2] MAC[2] ISA_A ISA_A
MOVEA	<ea>y,Ax	W,L → L	Source → Destination	ISA_A
MOVEM	#list,<ea>x <ea>y,#list	L	Listed Registers → Destination Source → Listed Registers	ISA_A
MOVEQ	#<data>,Dx	B → L	Immediate Data → Destination	ISA_A
MVS	<ea>y,Dx	B,W	Source with sign extension → Destination	ISA_B
MVZ	<ea>y,Dx	B,W	Source with zero fill → Destination	ISA_B

Table 3-2. Data Movement Operation Format

PEA	<ea>y	L	SP – 4 → SP; <ea>y → (SP)	ISA_A
UNLK	Ax	none	Ax → SP; (SP) → Ax; SP + 4 → SP	ISA_A

[1] Support for certain effective addressing modes was introduced with ISA_B. See Table 3-16.

[2] Supported for certain control registers was introduced with the eEMAC instruction set. See Table 3-16

3.1.2 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 and FTST), which set the integer or floating-point condition codes for use by other program and system control instructions. NOP forces synchronization of the internal pipelines. TPF is a no-operation instruction that does not force pipeline synchronization. Table 3-3 summarizes these instructions.

Table 3-3. Program Control Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First Appeared: ISA, (E)MAC, or FPU
Conditional				
Bcc	<label>	B, W, L	If Condition True, Then PC + d_n → PC	ISA_A[1]
FBcc	<label>	W, L	If Condition True, Then PC + d_n → PC	FPU
Scc	Dx	B	If Condition True, Then 1s → Destination; Else 0s → Destination	ISA_A
Unconditional				
BRA	<label>	B, W, L	PC + d_n → PC	ISA_A[1]
BSR	<label>	B, W, L	SP – 4 → SP; nextPC → (SP); PC + d_n → PC	ISA_A[1]
FNOP	none	none	PC + 2 → PC (FPU pipeline synchronized)	FPU
JMP	<ea>y	none	Source Address → PC	ISA_A
JSR	<ea>y	none	SP – 4 → SP; nextPC → (SP); Source → PC	ISA_A
NOP	none	none	PC + 2 → PC (Integer Pipeline Synchronized)	ISA_A
TPF	none #<data> #<data>	none W L	IPC + 2→ PC PC + 4 → PC PC + 6→ PC	ISA_A
Returns				ISA_A
RTS	none	none	(SP) → PC; SP + 4 → SP	
Test Operand				
TAS	<ea>x	B	Destination Tested → CCR; 1 → bit 7 of Destination	ISA_B

Table 3-3. Program Control Operation Format (Continued)

FTST	<ea>y	B, W, L, S, D	Source Operand Tested → FPCC	FPU
TST	<ea>y	B, W, L	Source Operand Tested → CCR	ISA_A

[1] Support for certain operand sizes was introduced with ISA_B. See Table 3-16.

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[1]
EQ—Equal	HI—Higher
MI—Minus	VC—Overflow clear
F—Never true [1]	LE—Less than or equal
NE—Not equal	VS—Overflow set

[1] Not applicable to the Bcc instructions.

For the definition of cc for FBcc, refer to Section 7.2, "Conditional Testing."

3.1.3 Integer Arithmetic Instructions

The integer arithmetic operations include 5 basic operations: ADD, SUB, MUL, DIV, and REM. They also include CMP, CLR, and NEG. The instruction set includes ADD, CMP, and SUB instructions for both address and data operations. The CLR instruction applies to all sizes of data operands. Signed and unsigned MUL, DIV, and REM instructions include:

- word multiply to produce a longword product
- longword multiply to produce a longword product
- longword divided by a word with a word quotient and word remainder
- longword divided by a longword with a longword quotient
- longword divided by a longword with a longword remainder (REM)

A set of extended instructions provides multiprecision and mixed-size arithmetic: ADDX, SUBX, EXT, and NEGX. For devices with the optional MAC or EMAC unit, MAC and MSAC instructions are available. Refer to Table 3-4 for a summary of the integer arithmetic operations. In Table 3-4, X refers to the X-bit in the CCR.

Table 3-4. Integer Arithmetic Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
ADD ADDA	Dy,<ea>x <ea>y,Dx <ea>y,Ax	L L L	Source + Destination → Destination	ISA_A
ADDI ADDQ	#<data>,Dx #<data>,<ea>x	L L	Immediate Data + Destination → Destination	ISA_A

Table 3-4. Integer Arithmetic Operation Format (Continued)

ADDX	Dy,Dx	L	Source + Destination + CCR[X] → Destination	ISA_A
CLR	<ea>x	B, W, L	0 → Destination	ISA_A
CMP CMPA	<ea>y,Dx <ea>y,Ax	B, W, L W, L	Destination – Source → CCR	ISA_A[1]
CMPI	#<data>,Dx	B, W, L	Destination – Immediate Data → CCR	ISA_A[1]
DIVS/DIVU	<ea>y,Dx	W, L	Destination / Source → Destination (Signed or Unsigned)	ISA_A
EXT EXTB	Dx Dx Dx	B → W W → L B → L	Sign-Extended Destination → Destination	ISA_A
MAAAC	Ry, RxSF, ACCx, ACCw	W, L	ACCx + (Ry*Rx){<<l>>} SF → ACCx ACCw + (Ry*Rx){<<l>>} SF → ACCw	ISA_C EMAC_B
MAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw,ACCx	W, L W, L	ACCx + (Ry * Rx){<<l>>}SF → ACCx ACCx + (Ry * Rx){<<l>>}SF → ACCx; (<ea>y(&MASK)) → Rw	ISA_A
MASAC	Ry, RxSF, ACCx, ACCw	W, L	ACCx + (Ry*Rx){<<l>>} SF → ACCx ACCw - (Ry*Rx){<<l>>} SF → ACCw	ISA_C EMAC_B
MSAAC	Ry, RxSF, ACCx, ACCw	W, L	ACCx - (Ry*Rx){<<l>>} SF → ACCx ACCw + (Ry*Rx){<<l>>} SF → ACCw	ISA_C EMAC_B
MSAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw,ACCx	W, L W, L	ACCx - (Ry * Rx){<<l>>}SF → ACCx ACCx - (Ry * Rx){<<l>>}SF → ACCx; (<ea>y(&MASK)) → Rw	ISA_A
MSSAC	Ry, RxSF, ACCx, ACCw	W, L	ACCx - (Ry*Rx){<<l>>} SF → ACCx ACCw - (Ry*Rx){<<l>>} SF → ACCw	ISA_C EMAC_B
MULS/MULU	<ea>y,Dx	W * W → L L * L → L	Source * Destination → Destination (Signed or Unsigned)	ISA_A
NEG	Dx	L	0 – Destination → Destination	ISA_A
NEGX	Dx	L	0 – Destination – CCR[X] → Destination	ISA_A
REMS/REMU	<ea>y,Dw:Dx	L	Destination / Source → Remainder (Signed or Unsigned)	ISA_A

SATS	Dx	L	If CCR[V] == 1;
			then if Dx[31] == 0;
			then Dx[31:0] = 0x80000000;
			else Dx[31:0] = 0x7FFFFFFF;
			else Dx[31:0] is unchanged
				ISA_B

SUB SUBA	<ea>y,Dx Dy,<ea>x <ea>y,Ax	L L L	Destination - Source → Destination	ISA_A
SUBI SUBQ	#<data>,Dx #<data>,<ea>x	L L	Destination – Immediate Data → Destination	ISA_A
SUBX	Dy,Dx	L	Destination – Source – CCR[X] → Destination	ISA_A

[1] Support for certain operand sizes was introduced with ISA_B. See Table 3-16.

3.1.4 Floating-Point Arithmetic Instructions

The floating-point instructions are organized into two categories: dyadic (requiring two operands) and monadic (requiring one operand). The dyadic floating-point instructions provide several arithmetic functions such as FADD and FSUB. For these operations, the first operand can be located in memory, an integer data register, or a floating-point data register. The second operand is always located in a floating-point data register. The results of the operation are stored in the register specified as the second operand. All FPU arithmetic operations support all data formats. Results are rounded to either single- or double-precision format. Table 3-5 gives the general format for these dyadic instructions. Table 3-6 lists the available operations.

Table 3-5. Dyadic Floating-Point Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
F<dop>	<ea>y,FPx FPy,FPx	B, W, L, S, D	FPx <Function> Source → FPx	FPU

Table 3-6. Dyadic Floating-Point Operations

Instruction (F<dop>)	Operation
FADD, FSADD, FDADD	Add
FCMP	Compare
FDIV, FSDIV, FDDIV	Divide
FMUL, FSMUL, FDMUL	Multiply
FSUB, FSSUB, FDSUB	Subtract

The monadic floating-point instructions provide several arithmetic functions requiring one input operand such as FABS. Unlike the integer counterparts to these functions (e.g., NEG), a source and a destination can be specified. The operation is performed on the source operand and the result is stored in the destination, which is always a floating-point data register. All data formats are supported. Table 3-7 gives the general format for these monadic instructions. Table 3-8 lists the available operations.

Table 3-7. Monadic Floating-Point Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
F<mop>	<ea>y,FPx FPy,FPx FPx	B, W, L, S, D	Source → <Function> → FPx FPx → <Function> → FPx	FPU

Table 3-8. Monadic Floating-Point Operations

Instruction (F<mop>)	Operation
FABS, FSABS, FDABS	Absolute Value

Table 3-8. Monadic Floating-Point Operations

FINT	Extract Integer Part
FINTRZ	Extract Integer Part, Rounded to Zero
FNEG, FSNEG, FDNEG	Negate
FSQRT, FSSQRT, FDSQRT	Square Root

3.1.5 Logical Instructions

The instructions AND, OR, EOR, and NOT perform logical operations with longword integer data operands. A similar set of immediate instructions (ANDI, ORI, and EORI) provides these logical operations with longword immediate data. Table 3-9 summarizes the logical operations.

Table 3-9. Logical Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
AND	<ea>y,Dx Dy,<ea>x	L L	Source & Destination → Destination	ISA_A
ANDI	#<data>, Dx	L	Immediate Data & Destination → Destination	ISA_A
EOR	Dy,<ea>x	L	Source ^ Destination → Destination	ISA_A
EORI	#<data>,Dx	L	Immediate Data ^ Destination → Destination	ISA_A
NOT	Dx	L	~ Destination → Destination	ISA_A
OR	<ea>y,Dx Dy,<ea>x	L L	Source | Destination → Destination	ISA_A
ORI	#<data>,Dx	L	Immediate Data | Destination → Destination	ISA_A

3.1.6 Shift Instructions

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

Register shift operations shift longwords. The shift count can be specified in the instruction operation word (to shift from 1 to 8 places) or in a register (modulo 64 shift count).

The SWAP instruction exchanges the 16-bit halves of a register. Table 3-10 is a summary of the shift operations. In Table 3-10, C and X refer to the C-bit and X-bit in the CCR.

Table 3-10. Shift Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
ASL	Dy,Dx #<data>,Dx	L L	CCR[X,C] ← (Dx << Dy) ← 0 CCR[X,C] ← (Dx << #<data>) ← 0	ISA_A
ASR	Dy,Dx #<data>,Dx	L L	msb → (Dx >> Dy) → CCR[X,C] msb → (Dx >> #<data>) → CCR[X,C]	ISA_A
LSL	Dy,Dx #<data>,Dx	L L	CCR[X,C] ← (Dx << Dy) ← 0 CCR[X,C] ← (Dx << #<data>) ← 0	ISA_A
LSR	Dy,Dx #<data>,Dx	L L	0 → (Dx >> Dy) → CCR[X,C] 0 → (Dx >> #<data>) → CCR[X,C]	ISA_A
SWAP	Dx	W	MSW of Dx ↔ LSW of Dx	ISA_A

3.1.7 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. In addition, BITREV, BYTEREV and FF1 instructions provide additional functionality in this category and operate on 32-bit register data values. Table 3-11 summarizes bit manipulation operations.

Table 3-11. Bit Manipulation Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
BCHG	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z] → <bit number> of Destination	ISA_A
BCLR	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]; 0 →<bit number> of Destination	ISA_A
BITREV	Dx	L	Bit reversed Dx → Dx	ISA_A+, ISA_C
BSET	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]; 1 → <bit number> of Destination	ISA_A
BYTEREV	Dx	L	Byte reversed Dx → Dx	ISA_A+, ISA_C
BTST	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]	ISA_A
FF1	Dx	L	Bit offset of First Logical One “1” in Dx → Dx	ISA_A+, ISA_C

3.1.8 System Control Instructions

This type of instruction includes privileged and trapping instructions as well as instructions that use or modify the CCR. FSAVE and FRESTORE save and restore the nonuser visible portion of the FPU during context switches. Table 3-12 summarizes these instructions.

Table 3-12. System Control Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
Privileged				
FRESTORE	<ea>y	none	FPU State Frame → Internal FPU State	FPU
FSAVE	<ea>x	none	Internal FPU State → FPU State Frame	FPU
HALT	none	none	Halt processor core (synchronizes pipeline)	ISA_A
MOVE from SR	SR,Dx	W	SR → Destination	ISA_A
MOVE from USP	USP,Dx	L	USP → Destination	ISA_B
MOVE to SR	<ea>y,SR	W	Source → SR; Dy or #<data> source only (synchronizes pipeline)	ISA_A
MOVE to USP	Ay,USP	L	Source → USP	ISA_B
MOVEC	Ry,Rc	L	Ry → Rc (synchronizes pipeline)	ISA_A
RTE	none	none	2 (SP) → SR; 4 (SP) → PC; SP + 8 →SP Adjust stack according to format (synchronizes pipeline)	ISA_A
STOP	#<data>	none	Immediate Data → SR; STOP (synchronizes pipeline)	ISA_A
STLDSR	#<data>	W	SP – 4 → SP; zero-filled SR → (SP); Immediate Data → SR	ISA_A+, ISA_C
WDEBUG	<ea>y	L	Addressed Debug WDMREG Command Executed (synchronizes pipeline)	ISA_A
Debug Functions				
PULSE	none	none	Set PST = 0x4	ISA_A
WDDATA	<ea>y	B, W, L	Source → DDATA port	ISA_A
Trap Generating				
ILLEGAL	none	none	SP – 4 → SP; PC → (SP) → PC; SP – 2 → SP; SR → (SP); SP – 2 → SP; Vector Offset → (SP); (VBR + 0x10) → PC	ISA_A
TRAP	#<vector>	none	1 → S Bit of SR; SP – 4 → SP; nextPC → (SP); SP – 2 → SP; SR → (SP) SP – 2 → SP; Format/Offset → (SP) (VBR + 0x80 +4*n) → PC, where n is the TRAP number	ISA_A

Certain instructions perform a pipeline synchronization prior to their actual execution. For these opcodes, the instruction enters the OEP and then waits until the following conditions are met:

- The instruction cache is in a quiescent state with all outstanding cache misses completed.
- The data cache is in a quiescent state with all outstanding cache misses completed.
- The push/store buffer is empty.
- The execution of all previous instructions has completed.

Once all these conditions are satisfied, the instruction begins its actual execution. For the instruction timings listed in the timing data, the following assumptions are made for these pipeline synchronization instructions:

- The instruction cache is not processing any cache misses.
- The data cache is not processing any cache misses.
- The push/store buffer is empty.
- The OEP has dispatched an instruction or instruction-pair on the previous cycle.

The following instructions perform this pipeline synchronization:

- cpushl
- halt
- intouch
- move_to_sr
- movec
- nop
- rte
- stop
- wdebug

3.1.9 Cache Maintenance Instructions

The cache instructions provide maintenance functions for managing the caches. CPUSHL is used to push a specific cache line, and possibly invalidate it. INTOUCH can be used to load specific data into the cache. Both of these instructions are privileged instructions. Table 3-13 summarizes these instructions.

Table 3-13. Cache Maintenance Operation Format

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
CPUSHL	ic,(Ax) dc,(Ax) bc,(Ax)	none	If data is valid and modified, push cache line; invalidate line if programmed in CACR (synchronizes pipeline)	ISA_A
INTOUCH	Ay	none	Instruction fetch touch at (Ay) (synchronizes pipeline)	ISA_B

3.2 Instruction Set Summary

This section contains tables which summarize the ColdFire instruction set architecture.

Table 3-14 shows the entire user instruction set in alphabetical order. Table 3-15 shows the entire supervisor instruction set in alphabetical order. Recall the major ISA revisions are defined as:

- ISA_ A: Original ColdFire instruction set architecture.
- ISA_B: Improved data movement instructions, byte- and word-sized compares, and miscellaneous enhancements are added.
- ISA_C: Instructions are added for improved bit manipulation.
- FPU: Floating-Point Unit instructions.
- MAC: Multiply-Accumulate instructions.
- EMAC: Revised ISA for enhanced Multiply-Accumulate unit.
- EMAC_B: Instructions are added for dual-accumulation operations.

Table 3-14. ColdFire User Instruction Set Summary

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
ADD ADDA	Dy,<ea>x <ea>y,Dx <ea>y,Ax	L L L	Source + Destination → Destination	ISA_A
ADDI ADDQ	#<data>,Dx #<data>,<ea>x	L L	Immediate Data + Destination → Destination	ISA_A
ADDX	Dy,Dx	L	Source + Destination + CCR[X] → Destination	ISA_A
AND	<ea>y,Dx Dy,<ea>x	L L	Source & Destination → Destination	ISA_A
ANDI	#<data>, Dx	L	Immediate Data & Destination → Destination	ISA_A
ASL	Dy,Dx #<data>,Dx	L L	CCR[X,C] ← (Dx << Dy) ← 0 CCR[X,C] ← (Dx << #<data>) ← 0	ISA_A
ASR	Dy,Dx #<data>,Dx	L L	msb → (Dx >> Dy) → CCR[X,C] msb → (Dx >> #<data>) → CCR[X,C	ISA_A
Bcc	<label>	B, W	If Condition True, Then PC + d_n → PC	ISA_A
Bcc	<label>	L	If Condition True, Then PC + d_n → PC	ISA_B
BCHG	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z] → <bit number> of Destination	ISA_A
BCLR	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]; 0 →<bit number> of Destination	ISA_A
BITREV	Dx	L	Destination data register contents are bit-reversed	ISA_A+,ISA_C
BRA	<label>	B, W	PC + d_n → PC	ISA_A
BRA	<label>	L	PC + d_n → PC	ISA_B
BSET	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]; 1 → <bit number> of Destination	ISA_A
BSR	<label>	B, W	SP – 4 → SP; nextPC → (SP); PC + d_n → PC	ISA_A
BSR	<label>	L	SP – 4 → SP; nextPC → (SP); PC + d_n → PC	ISA_B

Table 3-14. ColdFire User Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
BTST	Dy,<ea>x #<data>,<ea>x	B, L B, L	~ (<bit number> of Destination) → CCR[Z]	ISA_A
BYTEREV	Dx	L	Destination data register contents are byte-reversed	ISA_A+,ISA_C
CLR	<ea>x	B, W, L	0 → Destination	ISA_A
CMP CMPA	<ea>y,Dx <ea>y,Ax	L L	Destination – Source → CCR	ISA_A
CMP CMPA	<ea>y,Dx <ea>y,Ax	B, W W	Destination – Source → CCR	ISA_B
CMPI	#<data>,Dx	L	Destination – Immediate Data → CCR	ISA_A
CMPI	#<data>,Dx	B, W	Destination – Immediate Data → CCR	ISA_B
DIVS/DIVU	<ea>y,Dx	W, L	Destination / Source → Destination (Signed or Unsigned)	ISA_A
EOR	Dy,<ea>x	L	Source ^ Destination → Destination	ISA_A
EORI	#<data>,Dx	L	Immediate Data ^ Destination → Destination	ISA_A
EXT EXTB	Dx Dx Dx	B → W W → L B → L	Sign-Extended Destination → Destination	ISA_A
FABS	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Absolute Value of Source → FPx Absolute Value of FPx → FPx	FPU
FADD	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source + FPx → FPx	FPU
FBcc	<label>	W, L	If Condition True, Then PC + d_n → PC	FPU
FCMP	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source	FPU
FDABS	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Absolute Value of Source → FPx; round destination to double Absolute Value of FPx → FPx; round destination to double	FPU
FDADD	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source + FPx → FPx; round destination to double	FPU
FDDIV	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx / Source → FPx; round destination to double	FPU
FDIV	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx / Source → FPx	FPU
FDMOVE	FPy,FPx	D	Source → Destination; round destination to double	FPU

Table 3-14. ColdFire User Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
FDMUL	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source * FPx → FPx; round destination to double	FPU
FDNEG	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	- (Source) → FPx; round destination to double - (FPx) → FPx; round destination to double	FPU
FDSQRT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Square Root of Source → FPx; round destination to double Square Root of FPx → FPx; round destination to double	FPU
FDSUB	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source → FPx; round destination to double	FPU
FF1	Dx	L	Bit offset of First Logical One in Register → Destination	ISA_A+,ISA_C
FINT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Integer Part of Source → FPx Integer Part of FPx → FPx	FPU
FINTRZ	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Integer Part of Source → FPx; round to zero Integer Part of FPx → FPx; round to zero	FPU
FMOVE	<ea>y,FPx FPy,<ea>x FPy,FPx FPcr,<ea>x <ea>y,FPcr	B,W,L,S,D B,W,L,S,D D L L	Source → Destination FPcr can be any floating-point control register: FPCR, FPIAR, FPSR	FPU
FMOVEM	#list,<ea>x <ea>y,#list	D	Listed registers → Destination Source → Listed registers	FPU
FMUL	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source * FPx → FPx	FPU
FNEG	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	- (Source) → FPx - (FPx) → FPx	FPU
FNOP	none	none	PC + 2 → PC (FPU Pipeline Synchronized)	FPU
FSABS	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Absolute Value of Source → FPx; round destination to single Absolute Value of FPx → FPx; round destination to single	FPU
FSADD	<ea>y,FPx FPy,FPx	B,W,L,S,D	Source + FPx → FPx; round destination to single	FPU
FSDIV	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx / Source → FPx; round destination to single	FPU

Table 3-14. ColdFire User Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
FSMOVE	<ea>y,FPx	B,W,L,S,D	Source → Destination; round destination to single	FPU
FSMUL	<ea>y,FPx FPy,FPx	B,W,L,S,D D	Source * FPx → FPx; round destination to single	FPU
FSNEG	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	- (Source) → FPx; round destination to single - (FPx) → FPx; round destination to single	FPU
FSQRT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Square Root of Source → FPx Square Root of FPx → FPx	FPU
FSSQRT	<ea>y,FPx FPy,FPx FPx	B,W,L,S,D D D	Square Root of Source → FPx; round destination to single Square Root of FPx → FPx; round destination to single	FPU
FSSUB	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source → FPx; round destination to single	FPU
FSUB	<ea>y,FPx FPy,FPx	B,W,L,S,D D	FPx - Source → FPx	FPU
FTST	<ea>y	B, W, L, S, D	Source Operand Tested → FPCC	FPU
ILLEGAL	none	none	SP – 4 → SP; PC → (SP) → PC; SP – 2 → SP; SR → (SP); SP – 2 → SP; Vector Offset → (SP); (VBR + 0x10) → PC	ISA_A
JMP	<ea>y	none	Source Address → PC	ISA_A
JSR	<ea>y	none	SP – 4 → SP; nextPC → (SP); Source → PC	ISA_A
LEA	<ea>y,Ax	L	<ea>y → Ax	ISA_A
LINK	Ay,#<displacement>	W	SP – 4 → SP; Ay → (SP); SP → Ay, SP + d_n → SP	ISA_A
LSL	Dy,Dx #<data>,Dx	L L	CCR[X,C] ← (Dx << Dy) ← 0 CCR[X,C] ← (Dx << #<data>) ← 0	ISA_A
LSR	Dy,Dx #<data>,Dx	L L	0 → (Dx >> Dy) → CCR[X,C] 0 → (Dx >> #<data>) → CCR[X,C]	ISA_A
MAAAC	Ry, RxSF, ACCx, ACCw	L	ACCx + (Ry*Rx){<<|>>} SF → ACCx ACCw + (Ry*Rx){<<|>>} SF → ACCw	ISA_C EMAC_B
MAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw, ACCx	W, L W, L	ACCx + (Ry * Rx){<<|>>}SF → ACCx ACCx + (Ry * Rx){<<|>>}SF → ACCx; (<ea>y(&MASK)) → Rw	MAC
MASAC	Ry, RxSF, ACCx, ACCw	L	ACCx + (Ry*Rx){<<|>>} SF → ACCx ACCw - (Ry*Rx){<<|>>} SF → ACCw	ISA_C EMAC_B
MOV3Q	#<data>,<ea>x	L	Immediate Data → Destination	ISA_B

Table 3-14. ColdFire User Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
MOVCLR	ACCy,Rx	L	Accumulator → Destination, 0 → Accumulator	EMAC
MOVE MOVE from CCR MOVE to CCR	<ea>y,<ea>x MACcr,Dx <ea>y,MACcr CCR,Dx <ea>y,CCR	B,W,L L L W W	Source → Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK	ISA_A MAC MAC ISA_A ISA_A
MOVE	#<data>, d16(Ax)	B,W	Immediate Data → Destination	ISA_B
MOVEA	<ea>y,Ax	W,L → L	Source → Destination	ISA_A
MOVEM	#list,<ea>x <ea>y,#list	L	Listed Registers → Destination Source → Listed Registers	ISA_A
MOVEQ	#<data>,Dx	B → L	Immediate Data → Destination	ISA_A
MSAAC	Ry, RxSF, ACCx, ACCw	L	ACCx - (Ry*Rx){<<|>>} SF → ACCx ACCw + (Ry*Rx){<<|>>} SF → ACCw	ISA_C EMAC_B
MSAC	Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw, ACCx	W, L W, L	ACCx - (Ry * Rx){<<|>>}SF → ACCx ACCx - (Ry * Rx){<<|>>}SF → ACCx; (<ea>y(&MASK)) → Rw	MAC
MSSAC	Ry, RxSF, ACCx, ACCw	L	ACCx - (Ry*Rx){<<|>>} SF → ACCx ACCw - (Ry*Rx){<<|>>} SF → ACCw	ISA_C EMAC_B
MULS/MULU	<ea>y,Dx	W * W → L L * L → L	Source * Destination → Destination (Signed or Unsigned)	ISA_A
MVS	<ea>y,Dx	B,W	Source with sign extension → Destination	ISA_B
MVZ	<ea>y,Dx	B,W	Source with zero fill → Destination	ISA_B
NEG	Dx	L	0 – Destination → Destination	ISA_A
NEGX	Dx	L	0 – Destination – CCR[X] → Destination	ISA_A
NOP	none	none	PC + 2 → PC (Integer Pipeline Synchronized)	ISA_A
NOT	Dx	L	~ Destination → Destination	ISA_A
OR	<ea>y,Dx Dy,<ea>x	L L	Source | Destination → Destination	ISA_A
ORI	#<data>,Dx	L	Immediate Data | Destination → Destination	ISA_A
PEA	<ea>y	L	SP – 4 → SP; <ea>y → (SP)	ISA_A
PULSE	none	none	Set PST = 0x4	ISA_A
REMS/REMU	<ea>y,Dw:Dx	L	Destination / Source → Remainder (Signed or Unsigned)	ISA_A
RTS	none	none	(SP) → PC; SP + 4 → SP	ISA_A

Table 3-14. ColdFire User Instruction Set Summary (Continued)

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU

SATS	Dx	L	If CCR[V] == 1;
			then if Dx[31] == 0;
			then Dx[31:0] = 0x80000000;
			else Dx[31:0] = 0x7FFFFFFF;
			else Dx[31:0] is unchanged
				ISA_B

Scc	Dx	B	If Condition True, Then 1s → Destination; Else 0s → Destination	ISA_A
SUB SUBA	<ea>y,Dx Dy,<ea>x <ea>y,Ax	L L L	Destination - Source → Destination	ISA_A
SUBI SUBQ	#<data>,Dx #<data>,<ea>x	L L	Destination – Immediate Data → Destination	ISA_A
SUBX	Dy,Dx	L	Destination – Source – CCR[X] → Destination	ISA_A
SWAP	Dx	W	MSW of Dx ↔ LSW of Dx	ISA_A
TAS	<ea>x	B	Destination Tested → CCR; 1 → bit 7 of Destination	ISA_B
TPF	none #<data> #<data>	none W L	PC + 2→ PC PC + 4 → PC PC + 6→ PC	ISA_A
TRAP	#<vector>	none	1 → S Bit of SR; SP – 4 → SP; nextPC → (SP); SP – 2 → SP; SR → (SP) SP – 2 → SP; Format/Offset → (SP) (VBR + 0x80 +4*n) → PC, where n is the TRAP number	ISA_A
TST	<ea>y	B, W, L	Source Operand Tested → CCR	ISA_A
UNLK	Ax	none	Ax → SP; (SP) → Ax; SP + 4 → SP	ISA_A
WDDATA	<ea>y	B, W, L	Source → DDATA port	ISA_A

Table 3-15. ColdFire Supervisor Instruction Set Summary

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
CPUSHL	ic,(Ax) dc,(Ax) bc,(Ax)	none	If data is valid and modified, push cache line; invalidate line if programmed in CACR (synchronizes pipeline)	ISA_A
FRESTORE	<ea>y	none	FPU State Frame → Internal FPU State	FPU
FSAVE	<ea>x	none	Internal FPU State → FPU State Frame	FPU

Table 3-15. ColdFire Supervisor Instruction Set Summary

Instruction	Operand Syntax	Operand Size	Operation	First appeared: ISA, (E)MAC or FPU
HALT	none	none	Halt processor core	ISA_A
INTOUCH	Ay	none	Instruction fetch touch at (Ay)	ISA_B
MOVE from SR	SR,Dx	W	SR → Destination	ISA_A
MOVE from USP	USP,Dx	L	USP → Destination	ISA_B
MOVE to SR	<ea>y,SR	W	Source → SR; Dy or #<data> source only	ISA_A
MOVE to USP	Ay,USP	L	Source → USP	ISA_B
MOVEC	Ry,Rc	L	Ry → Rc	ISA_A
RTE	none	none	2 (SP) → SR; 4 (SP) → PC; SP + 8 →SP Adjust stack according to format	ISA_A
STLDSR	#<data>	W	SP – 4 → SP; zero-filled SR → (SP); Immediate Data → SR	ISA_A+,ISA_C
STOP	#<data>	none	Immediate Data → SR; STOP	ISA_A
WDEBUG	<ea>y	L	Addressed Debug WDMREG Command Executed	ISA_A

3.3 ColdFire Core Summary

This chapter provides a quick reference of the entire ColdFire instruction set architecture and the appropriate revision. Table 3-16 provides an alphabetical list of the entire set of instruction mnemonics, and the associated instructino set revisions. For more detailed descriptions of the instructions, see Table 3-14 on page 3-13 and Table 3-15 on page 3-18.

The standard products available at the time of publication of this document and the cores and optional modules that they contain are shown in Table 3-16.

Table 3-16. ColdFire Instruction Set and Processor Cross-Reference

Mnemonic	Description	ISA_A	ISA_A+	ISA_B	ISA_C	FPU	MAC	EMAC	EMAC_B
ADD	Add	X	X	X	X				
ADDA	Add Address	X	X	X	X				
ADDI	Add Immediate	X	X	X	X				
ADDQ	Add Quick	X	X	X	X				
ADDX	Add with Extend	X	X	X	X				
AND	Logical AND	X	X	X	X				
ANDI	Logical AND Immediate	X	X	X	X				
ASL, ASR	Arithmetic Shift Left and Right	X	X	X	X				
Bcc.{B,W}	Branch Conditionally, Byte and Word	X	X	X	X				

Table 3-16. ColdFire Instruction Set and Processor Cross-Reference (Continued)

Mnemonic	Description	ISA_A	ISA_A+	ISA_B	ISA_C	FPU	MAC	EMAC	EMAC_B
Bcc.L	Branch Conditionally, Longword			X	X				
BCHG	Test Bit and Change	X	X	X	X				
BCLR	Test Bit and Clear	X	X	X	X				
BITREV	Bit Reverse		X		X				
BRA.{B,W}	Branch Always, Byte and Word	X	X	X	X				
BRA.L	Branch Always, Longword		X	X					
BSET	Test Bit and Set	X	X	X	X				
BSR.{B,W}	Branch to Subroutine, Byte and Word	X	X	X	X				
BSR.L	Branch to Subroutine, Longword			X	X				
BTST	Test a Bit	X	X	X	X				
BYTEREV	Byte Reverse		X		X				
CLR	Clear	X	X	X	X				
CMP.{B,W}	Compare, Byte and Word			X	X				
CMP.L	Compare, Longword	X	X	X	X				
CMPA.W	Compare Address, Word			X	X				
CMPA.L	Compare Address, Longword	X	X	X	X				
CMPI.{B,W}	Compare Immediate, Byte and Word			X	X				
CMPI.L	Compare Immediate, Longword	X	X	X	X				
CPUSHL	Push and Possibly Invalidate Cache	X	X	X	X				
DIVS	Signed Divide	X	X	X	X				
DIVU	Unsigned Divide	X	X	X	X				
EOR	Logical Exclusive-OR	X	X	X	X				
EORI	Logical Exclusive-OR Immediate	X	X	X	X				
EXT, EXTB	Sign Extend	X	X	X	X				
FABS, FSABS FDABS	Floating-Point Absolute Value					X			
FADD, FSADD, FDADD	Floating-Point Add					X			
FBcc	Floating-Point Branch Conditionally					X			
FCMP	Floating-Point Compare					X			
FDIV, FSDIV, FDDIV	Floating-Point Divide					X			
FF1	Find First One		X		X				

Table 3-16. ColdFire Instruction Set and Processor Cross-Reference (Continued)

Mnemonic	Description	ISA_A	ISA_A+	ISA_B	ISA_C	FPU	MAC	EMAC	EMAC_B
FFINT, FSINT, FDINT	Floating-Point Integer					X			
FINTRZ	Floating-Point Integer Round-to-Zero					X			
FMOVE, FSMOVE, FDMOVE	Move Floating-Point Data Register					X			
FMOVE from FPCR	Move from the Floating-Point Control Register					X			
FMOVE from FPIAR	Move from the Floating-Point Instruction Address Register					X			
FMOVE from FPSR	Move from the Floating-Point Status Register					X			
FMOVE to FPCR	Move to the Floating-Point Control Register					X			
FMOVE to FPIAR	Move to the Floating-Point Instruction Address Register					X			
FMOVE to FPSR	Move to the Floating-Point Status Register					X			
FMOVEM	Move Multiple Floating-Point Data Registers					X			
FMUL, FSMUL, FDMUL	Floating-Point Data Registers					X			
FNEG, FSNEG, FDNEG	Floating-Point Negate					X			
FNOP	Floating-Point No Operation					X			
FRESTORE	Restore Internal Floating-Point State					X			
FSAVE	Save Internal Floating-Point State					X			
FSQRT, FSSQRT, FDSQRT	Floating-Point Square Root					X			
FSUB	Floating-Point Subtract					X			
FTST	Test Floating-Point Operand					X			
HALT	Halt CPU	X	X	X	X				
ILLEGAL	Take Illegal Instruction Trap	X	X	X	X				
INTOUCH	Instruction Fetch Touch			X	X				
JMP	Jump	X	X	X	X				

Table 3-16. ColdFire Instruction Set and Processor Cross-Reference (Continued)

Mnemonic	Description	ISA_A	ISA_A+	ISA_B	ISA_C	FPU	MAC	EMAC	EMAC_B
JSR	Jump to Subroutine	X	X	X	X				
LEA	Load Effective Address	X	X	X	X				
LINK	Link and Allocate	X	X	X	X				
LSL, LSR	Logical Shift Left and Right	X	X	X	X				
MAAAC	Multiply and Add to 1st Accumulator, Add to 2nd Accumulator								X
MAC	Multiply and Accumulate						X	X	X
MASAC	Multiply and Add to 1st Accumulator, Subtract from 2nd Accumulator								X
MOV3Q	Move 3-Bit Data Quick			X	X				
MOVCLR	Move from Accumulator and Clear							X	X
MOVE	Move	X	X	X	X				
MOVEI	Move Immediate, Byte and Word to Ax with Displacement			X	X				
MOVE ACC to ACC	Copy Accumulator							X	X
MOVE from ACC	Move from Accumulator						X	X	X
MOVE from ACCext01	Move from Accumulator 0 and 1 Extensions							X	X
MOVE ACCext23	Move from Accumulator 2 and 3 Extensions							X	X
MOVE from CCR	Move from Condition Code Register	X	X	X	X				
MOVE from MACSR	Move from MAC Status Register						X	X	X
MOVE from MACSR tp CCR	Move from MAC Status Register to Condition Code Register						X	X	X
MOVE from MASK	Move from MAC Mask Register						X	X	X
MOVE from SR	Move from the Status Register	X	X	X	X				
MOVE from USP	Move from User Stack Pointer		X	X	X				
MOVE to ACC	Move to Accumulator						X	X	X
MOVE to ACCext01	Move to Accumulator 0 and 1 Extensions							X	X

Table 3-16. ColdFire Instruction Set and Processor Cross-Reference (Continued)

Mnemonic	Description	ISA_A	ISA_A+	ISA_B	ISA_C	FPU	MAC	EMAC	EMAC_B
MOVE ACCext23	Move to Accumulator 2 and 3 Extensions							X	X
MOVE to CCR	Move to Condition Code Register	X	X	X	X				
MOVE to MACSR	Move to MAC Status Register						X	X	X
MOVE to MASK	Move to MAC Mask Register						X	X	X
MOVE to SR	Move to the Status Register	X	X	X	X				
MOVE to USP	Move to User Stack Pointer		X	X	X				
MOVEA	Move Address	X	X	X	X				
MOVEC	Move Control Register	X	X	X	X				
MOVEM	Move Multiple Registers	X	X	X	X				
MOVEQ	Move Quick	X	X	X	X				
MSAAC	Multiply and Subtract to 1st Accumulator, Add to 2nd Accumulator								X
MSAC	Multiply and Subtract						X	X	X
MSSAC	Multiply and Subtract to 1st Accumulator, Subtract to 2nd Accumulator								X
MULS	Signed Multiply	X	X	X	X				
MULU	Unsigned Multiply	X	X	X	X				
MVS	Move with Sign Extend			X	X				
MVZ	Move with Zero-Fill			X	X				
NEG	Negate	X	X	X	X				
NEGX	Negate with Extend	X	X	X	X				
NOP	No Operation	X	X	X	X				
NOT	Logical Complement	X	X	X	X				
OR	Logical Inclusive-OR	X	X	X	X				
ORI	Logical Inclusive-OR Immediate	X	X	X	X				
PEA	Push Effective Address	X	X	X	X				
PULSE	Generate Processor Status	X	X	X	X				
REMS	Signed Divide Remainder	X	X	X	X				
REMU	Unsigned Divide Remainder	X	X	X	X				
RTE	Return from Exception	X	X	X	X				

Table 3-16. ColdFire Instruction Set and Processor Cross-Reference (Continued)

Mnemonic	Description	ISA_A	ISA_A+	ISA_B	ISA_C	FPU	MAC	EMAC	EMAC_B
RTS	Return from Subroutine	X	X	X	X				
SATS	Signed Saturate			X	X				
Scc	Set According to Condition	X	X	X	X				
STLDSR	Store and Load Status Register		X		X				
STOP	Load Status Register and Stop	X	X	X	X				
SUB	Subtract	X	X	X	X				
SUBA	Subtract Address	X	X	X	X				
SUBI	Subtract Immediate	X	X	X	X				
SUBQ	Subtract Quick	X	X	X	X				
SUBX	Subtract with Extend	X	X	X	X				
SWAP	Swap Register Words	X	X	X	X				
TAS	Test and Set and Operand			X	X				
TPF	Trap False	X	X	X	X				
TRAP	Trap	X	X	X	X				
TST	Test Operand	X	X	X	X				
UNLK	Unlink	X	X	X	X				
WDDATA	Write Data Control Register	X	X	X	X				
WDEBUG	Write Debug Control Register	X	X	X	X				

Chapter 4
Integer User Instructions

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

Not all instructions are supported by all ColdFire processors. See Chapter 3, “Instruction Set Summary for specific details on the instruction set definitions.

Add

First appeared in ISA_A

Operation: Source + Destination → Destination

Assembler Syntax: ADD.L <ea>y,Dx
ADD.L Dy,<ea>x

Attributes: Size = longword

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 only be specified as a longword. The mode of the instruction indicates which operand is the source and which is the destination as well as the operand size.

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

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

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 an carry is generated; cleared otherwise

Instruction Format:

15	14	13	12	11 10 9	8	7 6	5 4	3 2 1 0
1	1	0	1	Register	Opmode	Effective Address		
						Mode		Register

Instruction Fields:

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

Byte	Word	Longword	Operation
—	—	010	<ea>y + Dx → Dx
—	—	110	Dy + <ea>x → <ea>x

Instruction Fields (continued):

- Effective Address field—Determines addressing mode
  - For the source operand <ea>y, use addressing modes listed in the following table:

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

  - For the destination operand <ea>x, use addressing modes listed in the following table:

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

ADDA — Add Address — ADDA

First appeared in ISA_A

Operation: Source + Destination → Destination

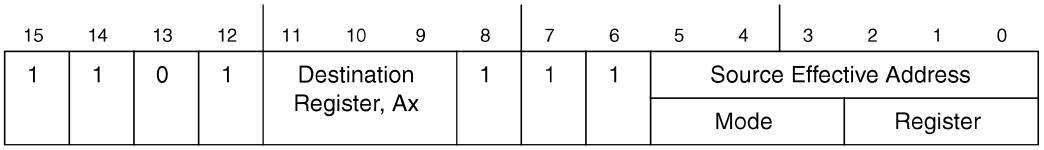
Assembler Syntax: ADDA.L <ea>y,Ax

Attributes: Size = longword

Description: Operates similarly to ADD, but is used when the destination register is an address register rather than a data register. 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 longword.

Condition Codes: Not affected

Instruction Format:



Instruction Fields:

- Destination Register field—Specifies the destination register, Ax.
- Source Effective Address field— Specifies the source operand; use addressing modes listed in the following table:

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

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

ADDI

Add Immediate

First appeared in ISA_A

ADDI

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ADDI.L #<data>,Dx

Attributes: Size = longword

Description: Operates similarly to ADD, but is used when the source operand is immediate data. Adds the immediate data to the destination operand and stores the result in the destination data register, Dx. The size of the operation is specified as longword. The size of the immediate data is specified as a longword. Note that the immediate data is contained in the two extension words, with the first extension word, bits [15:0], containing the upper word, and the second extension word, bits [15:0], containing the lower 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 an 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, Dx		
Upper Word of Immediate Data															
Lower Word of Immediate Data															

Instruction Fields:

- Destination Register field - Specifies the destination data register, Dx.

ADDQ — Add Quick — ADDQ

First appeared in ISA_A

Operation: Immediate Data + Destination → Destination

Assembler Syntax: ADDQ.L #<data>,<ea>x

Attributes: Size = longword

Description: Operates similarly to ADD, but is used when the source operand is immediate data ranging in value from 1 to 8. Adds the immediate value to the operand at the destination location. The size of the operation is specified as longword. The immediate data is zero-filled to a longword before being added to the destination. When adding to address registers, the condition codes are not altered.

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 an carry is generated; 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			0	1	0	Destination Effective Address					
										Mode			Register		

Instruction Fields:

- Data field—3 bits of immediate data representing 8 values (0 – 7), with 1-7 representing values of 1-7 respectively and 0 representing a value of 8.
- Destination Effective Address field—Specifies the destination location, <ea>x; use only those alterable addressing modes listed in the following table:

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

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

Add Extended

First appeared in ISA_A

ADDX

Operation: Source + Destination + CCR[X] → Destination

Assembler Syntax: ADDX.L Dy,Dx

Attributes: Size = longword

Description: Adds the source operand and CCR[X] to the destination operand and stores the result in the destination location. The size of the operation is specified as a longword.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit
N Set if the result is negative; cleared otherwise
Z Cleared if the result is non-zero; unchanged otherwise
V Set if an overflow is generated; cleared otherwise
C Set if an carry is generated; cleared otherwise

Normally CCR[Z] is set explicitly via programming before the start of an ADDX operation to allow successful testing for zero results upon completion of multiple-precision operations.

Instruction Format:

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

Instruction Fields:

- Register Dx field—Specifies the destination data register, Dx.
- Register Dy field—Specifies the source data register, Dy.

AND Logical

First appeared in ISA_A

Operation: Source & Destination → Destination

Assembler Syntax: AND.L <ea>y,Dx
AND.L Dy,<ea>x

Attributes: Size = longword

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

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

ANDI is used when the source is immediate data.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb 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	Data Register	Opmode	Effective Address	
						Mode	Register

Instruction Fields:

- Register field—Specifies any of the 8 data registers.
- Opmode field:

Byte	Word	Longword	Operation
—	—	010	<ea>y & Dx → Dx
—	—	110	Dy & <ea>x → <ea>x

Instruction Fields (continued):

- Effective Address field—Determines addressing mode.
  - For the source operand <ea>y, use addressing modes listed in the following table:

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

  - For the destination operand <ea>x, use addressing modes listed in the following table:

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

First appeared in ISA_A

Operation: Immediate Data & Destination → Destination

Assembler Syntax: ANDI.L #<data>,Dx

Attributes: Size = longword

Description: Performs an AND operation of the immediate data with the destination operand and stores the result in the destination data register, Dx. The size of the operation is specified as a longword. The size of the immediate data is specified as a longword. Note that the immediate data is contained in the two extension words, with the first extension word, bits [15:0], containing the upper word, and the second extension word, bits [15:0], containing the lower word.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb 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	Destination Register, Dx		
Upper Word of Immediate Data															
Lower Word of Immediate Data															

Instruction Fields:

- Destination Register field - specifies the destination data register, Dx.

Arithmetic Shift

First appeared in ISA_A

ASL, ASR

Operation: Destination Shifted By Count → Destination

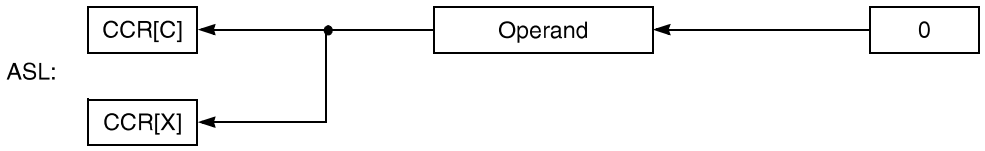
Assembler Syntax: ASd.L Dy,Dx
ASd.L #<data>,Dx
where d is direction, L or R

Attributes: Size = longword

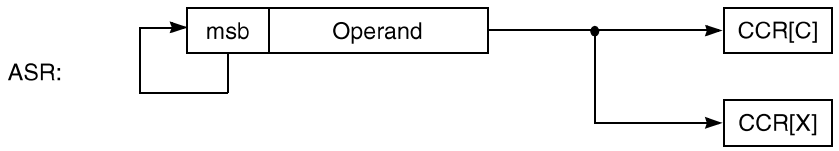
Description: Arithmetically shifts the bits of the destination operand, Dx, in the direction (L or R) specified. The size of the operand is a longword. CCR[C] receives the last bit shifted out of the operand. The shift count is the number of bit positions to shift the destination register and may be specified in two different ways:

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

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



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 msb of the result is set; cleared otherwise
Z Set if the result is zero; cleared otherwise
V Always cleared
C Set according to the last bit shifted out of the operand; cleared for a shift count of zero

Note that CCR[V] is always cleared by ASL and ASR, unlike on the 68K family processors.

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	1	1	0	Count or Register, Dy	dr	1	0	i/r	0	0	Register, Dx

Instruction Fields:

- Count or Register field—Specifies shift count or register, Dy, that contains the shift count:
  - If i/r = 0, this field 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, this field specifies the data register, Dy, that contains the shift count (modulo 64).
- 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, Dx, to be shifted.

Bcc

Branch Conditionally

First appeared in ISA_A

.L First appeared in ISA_B

Bcc

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

Assembler Syntax: Bcc.sz <label>

Attributes: Size = byte, word, longword (longword supported starting with ISA_B)

Description: If the condition is true, execution continues at (PC) + displacement. Branches can be forward, with a positive displacement, or backward, with a negative displacement. PC holds 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 PC to the destination PC. If the 8-bit displacement field is 0, a 16-bit displacement (the word after the instruction) is used. If the 8-bit displacement field is 0xFF, the 32-bit displacement (longword after the instruction) is used. A branch to the next immediate instruction uses 16-bit displacement because the 8-bit displacement field is 0x00.

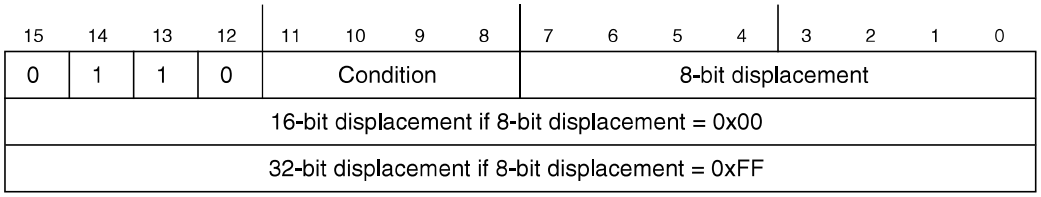
Condition code specifies one of the following tests, where C, N, V, and Z stand for the condition code bits CCR[C], CCR[N], CCR[V] and CCR[Z], respectively:

Code	Condition	Encod-ing	Test	Code	Condition	Encod-ing	Test
CC(HS)	Carry clear	0100	$\overline{C}$	LS	Lower or same	0011	C I Z
CS(LO)	Carry set	0101	C	LT	Less than	1101	N & $\overline{V}$ I $\overline{N}$ & V
EQ	Equal	0111	Z	MI	Minus	1011	N
GE	Greater or equal	1100	N & V I $\overline{N}$ & $\overline{V}$	NE	Not equal	0110	$\overline{Z}$
GT	Greater than	1110	N & V & $\overline{Z}$ I $\overline{N}$ & $\overline{V}$ & $\overline{Z}$	PL	Plus	1010	$\overline{N}$
HI	High	0010	$\overline{C}$ & $\overline{Z}$	VC	Overflow clear	1000	$\overline{V}$
LE	Less or equal	1111	Z I N & $\overline{V}$ I $\overline{N}$ & V	VS	Overflow set	1001	V

Condition Codes: Not affected

Bcc Branch Conditionally Bcc

Instruction Format:



Instruction Fields:

- Condition field—Binary encoding 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 and the next instruction to be executed if the condition is met.
- 16-bit displacement field—Used when the 8-bit displacement contains 0x00.
- 32-bit displacement field—Used when the 8-bit displacement contains 0xFF.

BCHG — Test a Bit and Change — BCHG

First appeared in ISA_A

Operation: ~ (<bit number> of Destination) → CCR[Z];
~ (<bit number> of Destination) → <bit number> of Destination

Assembler Syntax: BCHG.sz Dy,<ea>x
BCHG.sz #<data>,<ea>x

Attributes: Size = byte, longword

Description: Tests a bit in the destination operand and sets CCR[Z] 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

Bit Number Static, Specified as Immediate Data:

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	1	Destination Effective Address					
										Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

Instruction Fields:

- Destination Effective Address field—Specifies the destination location <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode, all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

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

- Bit Number field—Specifies the bit number.

Bit Number Dynamic, Specified in a Register:

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Data Register, Dy			1	0	1	Destination Effective Address					
										Mode			Register		

Instruction Fields:

- Data Register field—Specifies the data register, Dy, that contains the bit number.
- Destination Effective Address field—Specifies the destination location, <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode, all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

BCLR

Test a Bit and Clear

First appeared in ISA_A

BCLR

Operation: ~ (<bit number> of Destination) → CCR[Z];
0 → <bit number> of Destination

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

Attributes: Size = byte, longword

Description: Tests a bit in the destination operand and sets CCR[Z] 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

Bit Number Static, Specified as Immediate Data:

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	0	Destination Effective Address					
										Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

Instruction Fields:

- Destination Effective Address field—Specifies the destination location <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode, all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

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

- Bit Number field—Specifies the bit number.

Bit Number Dynamic, Specified in a Register:

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
0	0	0	0	Data Register, Dy	1	1	0	Destination Effective Address	
								Mode	Register

Instruction Fields:

- Data Register field—Specifies the data register, Dy, that contains the bit number.
- Destination Effective Address field—Specifies the destination location, <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode, all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

BITREV Bit Reverse Register BITREV

First appeared in ISA_C

Operation: Bit Reversed Dx → Dx

Assembler Syntax: BITREV.L Dx

Attributes: Size = longword

Description: The contents of the destination data register are bit-reversed, i.e., new Dx[31] = old Dx[0], new Dx[30] = old Dx[1], ..., new Dx[0] = old Dx[31].

Condition Codes: Not affected

Instruction Format:

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

Instruction Field:

Register field—Specifies the destination data register, Dx

BRA

Branch Always

First appeared in ISA_A
.L First appeared in ISA_B

BRA

Operation: $PC + d_n \rightarrow PC$

Assembler Syntax: BRA.sz <label>

Attributes: Size = byte, word, longword (longword supported starting with ISA_B)

Description: Program execution continues at location (PC) + displacement. Branches can be forward with a positive displacement, or backward with a negative displacement. The PC 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 PC to the destination PC. If the 8-bit displacement field in the instruction word is 0, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones (0xFF), the 32-bit displacement (longword immediately following the instruction) is used. A branch to the next immediate instruction automatically uses the 16-bit displacement format because the 8-bit displacement field contains 0x00 (zero offset).

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 = 0x00															
32-bit displacement if 8-bit displacement = 0xFF															

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 displacement when the 8-bit displacement contains 0x00.
- 32-bit displacement field—Used for displacement when the 8-bit displacement contains 0xFF.

BSET

Test a Bit and Set

First appeared in ISA_A

BSET

Operation: ~ (<bit number> of Destination) → CCR[Z];
1 → <bit number> of Destination

Assembler Syntax: BSET.sz Dy,<ea>x
BSET.sz #<data >,<ea>x

Attributes: Size = byte, longword

Description: Tests a bit in the destination operand and sets CCR[Z] 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

Bit Number Static, Specified as Immediate Data:

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	1	1	Destination Effective Address					
										Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

Instruction Fields:

- Destination Effective Address field—Specifies the destination location <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode; all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

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

- Bit Number field—Specifies the bit number.

Bit Number Dynamic, Specified in a Register:

Instruction Format:

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
0	0	0	0	Data Register, Dy	1	1	1	Destination Effective Address	
								Mode	Register

Instruction Fields:

- Data Register field—Specifies the data register, Dy, that contains the bit number.
- Destination Effective Address field—Specifies the destination location, <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode, all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

BSR

Branch to Subroutine

First appeared in ISA_A
.L First appeared in ISA_B

BSR

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

Assembler Syntax: BSR.sz <label>

Attributes: Size = byte, word, longword (longword supported starting with ISA_B)

Description: Pushes the longword address of the instruction immediately following the BSR instruction onto the system stack. Branches can be forward with a positive displacement, or backward with a negative displacement.The PC 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 PC to the destination PC. If the 8-bit displacement field in the instruction word is 0, a 16-bit displacement (the word immediately following the instruction) is used. If the 8-bit displacement field in the instruction word is all ones (0xFF), the 32-bit displacement (longword immediately following the instruction) is used. A branch to the next immediate instruction automatically uses the 16-bit displacement format because the 8-bit displacement field contains 0x00 (zero offset).

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 = 0x00															
32-bit displacement if 8-bit displacement = 0xFF															

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 displacement when the 8-bit displacement contains 0x00.
- 32-bit displacement field—Used for displacement when the 8-bit displacement contains 0xFF.

BTST — Test a Bit — BTST

First appeared in ISA_A

Operation: ~ (<bit number> of Destination) → CCR[Z]

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

Attributes: Size = byte, longword

Description: Tests a bit in the destination operand and sets CCR[Z] 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

Bit Number Static, Specified as Immediate Data:

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	Destination Effective Address					
										Mode			Register		
0	0	0	0	0	0	0	0	Bit Number							

Instruction Fields:

- Destination Effective Address field—Specifies the destination location <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode, all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

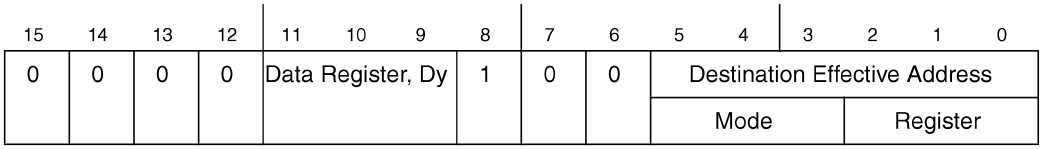
Addressing Mode	Mode	Register
(xxx).W	—	—
(xxx).L	—	—
#<data>	—	—
(d_{16},PC)	—	—
(d_8,PC,Xi)	—	—

Instruction Fields (continued):

- Bit Number field—Specifies the bit number.

Bit Number Dynamic, Specified in a Register:

Instruction Format:



Instruction Fields:

- Data Register field—Specifies the data register, Dy, that contains the bit number.
- Destination Effective Address field—Specifies the destination location, <ea>x; use only those data alterable addressing modes listed in the following table. Note that longword is allowed only for the Dx mode, all others are byte only.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

BYTEREV Byte Reverse Register BYTEREV

First appeared in ISA_C

Operation: Byte Reversed Dx → Dx

Assembler Syntax: BYTEREV.L Dx

Attributes: Size = longword

Description: The contents of the destination data register are byte-reversed as defined below:

Table 1:

new Dx[31:24]	= old Dx[7:0]
new Dx[23:16]	= old Dx[15:8]
new Dx[15:8]	= old Dx[23:16]
new Dx[7:0]	= old Dx[31:24]

Condition Codes: Not affected

Instruction Format:

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

Instruction Field:

- Register field—Specifies the destination data register, Dx.

CLR — Clear an Operand — CLR

First appeared in ISA_A

Operation: 0 → Destination

Assembler Syntax: CLR.sz <ea>x

Attributes: Size = byte, word, longword

Description: Clears the destination operand to 0. The size of the operation may be specified as byte, word, or longword.

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		Destination Effective Address					
										Mode		Register			

Instruction Fields:

- Size field—Specifies the size of the operation
  - — 00 byte operation
  - — 01 word operation
  - — 10 longword operation
  - — 11 reserved
- Effective Address field—Specifies the destination location, <ea>x; use only those data alterable addressing modes listed in the following table:

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

CMP

Compare

First appeared in ISA_A
.B and .W First appeared in ISA_B

CMP

Operation: Destination – Source → cc

Assembler Syntax: CMP.sz <ea>y,Dx

Attributes: Size = byte, word, longword (byte, word supported starting with ISA_B)

Description: Subtracts the source operand from the destination operand in the data register and sets condition codes according to the result; the data register is unchanged. The operation size may be a byte, word, or longword. CMPA is used when the destination is an address register; CMPI is used when the source is immediate data.

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	Destination Register, Dx	Opmode	Source Effective Address: Mode	Source Effective Address: Register

Instruction Fields:

- Register field—Specifies the destination register, Dx.
- Opmode field:

Byte	Word	Longword	Operation
000	001	010	Dx - <ea>y

- Source Effective Address field— Specifies the source operand, <ea>y; use addressing modes listed in the following table:

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

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

CMPA — Compare Address — CMPA

First appeared in ISA_A
.W First appeared in ISA_B

Operation: Destination – Source → cc

Assembler Syntax: CMPA.sz <ea>y, Ax

Attributes: Size = word, longword (word supported starting with ISA_B)

Description: Operates similarly to CMP, but is used when the destination register is an address register rather than a data register. The operation size can be word or longword. 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 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	Address Register, Ax	Opmode	Source Effective Address	
						Mode	Register

Instruction Fields:

- Address Register field—Specifies the destination register, Ax.
- Opmode field:

Byte	Word	Longword	Operation
—	011	111	Ax - <ea>y

- Source Effective Address field specifies the source operand, <ea>y; use addressing modes in the following table:

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

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

CMPI

Compare Immediate

First appeared in ISA_A
.B and .W First appeared in ISA_B

CMPI

Operation: Destination – Immediate Data→ cc

Assembler Syntax: CMPI.sz #<data>,Dx

Attributes: Size = byte, word, longword (byte, word supported starting with ISA_B)

Description: Operates similarly to CMP, but is used when the source operand is immediate data. The operation size can be byte, word, or longword. The size of the immediate data matches the operation size. Note that if size = byte, the immediate data is contained in bits [7:0] of the single extension word. If size = word, the immediate data is contained in the single extension word, bits [15:0]. If size = longword, the immediate data is contained in the two extension words, with the first extension word, bits [15:0], containing the upper word, and the second extension word, bits [15:0], containing the lower 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	Size		0	0	0	Register, Dx		
Upper Word of Immediate Data															
Lower Word of Immediate Data															

Instruction Fields:

- Register field—Specifies the destination register, Dx.
- Size field—Specifies the size of the operation
  - — 00 byte operation
  - — 01 word operation
  - — 10 longword operation
  - — 11 reserved

DIVS — Signed Divide — DIVS

First appeared in ISA_A
Not implemented in MCF5202, MCF5204 and MCF5206

Operation: Destination/Source → Destination

Assembler Syntax: DIVS.W <ea>y,Dx 32-bit Dx/16-bit <ea>y Æ (16r:16q) in Dx
DIVS.L <ea>y,Dx 32-bit Dx/32-bit <ea>y Æ 32q in Dx
where q indicates the quotient, and r indicates the remainder

Attributes: Size = word, longword

Description: Divide the signed destination operand by the signed source and store the signed result in the destination. For a word-sized operation, the destination operand is a longword and the source is a word; the 16-bit quotient is in the lower word and the 16-bit remainder is in the upper word of the destination. Note that the sign of the remainder is the same as the sign of the dividend. For a longword-sized operation, the destination and source operands are both longwords; the 32-bit quotient is stored in the destination. To determine the remainder on a longword-sized operation, use the REMS instruction.

An attempt to divide by zero results in a divide-by-zero exception and no registers are affected. The resulting exception stack frame points to the offending divide opcode. If overflow is detected, the destination register is unaffected. An overflow occurs if the quotient is larger than a 16-bit (.W) or 32-bit (.L) signed integer.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- X Not affected
- N Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive
- Z Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero
- V Set if an overflow occurs; cleared otherwise
- C Always cleared

Instruction Format: (Word)

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
1	0	0	0	Register, Dx	1	1	1	Source Effective Address	
								Mode	Register

DIVS — Signed Divide — DIVS

First appeared in ISA_A

Instruction Fields (Word):

- Register field—Specifies the destination register, Dx.
- Source Effective Address field specifies the source operand, <ea>y; use addressing modes in the following table:

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

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

Instruction Format: (Longword)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	Source Effective Address					
										Mode			Register		
0	Register, Dx			1	0	0	0	0	0	0	0	0	Register, Dx		

Instruction Fields (Longword):

- Register field—Specifies the destination register, Dx. Note that this field appears twice in the instruction format.
- Source Effective Address field— Specifies the source operand, <ea>y; use addressing modes in the following table:

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

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

DIVU — Unsigned Divide — DIVU

First appeared in ISA_A
Not implemented in MCF5202, MCF5204 and MCF5206

Operation: Destination/Source → Destination

Assembler Syntax: DIVU.W <ea>y,Dx 32-bit Dx/16-bit <ea>y Æ (16r:16q) in Dx
DIVU.L <ea>y,Dx 32-bit Dx/32-bit <ea>y Æ 32q in Dx
where q indicates the quotient, and r indicates the remainder

Attributes: Size = word, longword

Description: Divide the unsigned destination operand by the unsigned source and store the unsigned result in the destination. For a word-sized operation, the destination operand is a longword and the source is a word; the 16-bit quotient is in the lower word and the 16-bit remainder is in the upper word of the destination. For a longword-sized operation, the destination and source operands are both longwords; the 32-bit quotient is stored in the destination. To determine the remainder on a longword-sized operation, use the REMU instruction.

An attempt to divide by zero results in a divide-by-zero exception and no registers are affected. The resulting exception stack frame points to the offending divide opcode. If overflow is detected, the destination register is unaffected. An overflow occurs if the quotient is larger than a 16-bit (.W) or 32-bit (.L) unsigned integer.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- X Not affected
- N Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive
- Z Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero
- V Set if an overflow occurs; cleared otherwise
- C Always cleared

Instruction Format: (Word)

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
1	0	0	0	Register, Dx	0	1	1	Source Effective Address	
								Mode	Register

Instruction Fields (Word):

- Register field—Specifies the destination register, Dx.
- Source Effective Address field specifies the source operand, <ea>y; use addressing modes in the following table:

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

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

Instruction Format: (Longword)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	1	Source Effective Address					
										Mode			Register		
0	Register, Dx			0	0	0	0	0	0	0	0	0	Register, Dx		

Instruction Fields (Longword):

- Register field—Specifies the destination register, Dx. Note that this field appears twice in the instruction format.
- Source Effective Address field— Specifies the source operand, <ea>y; use addressing modes in the following table:

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

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

Exclusive-OR Logical

First appeared in ISA_A

EOR

Operation: Source ^ Destination → Destination

Assembler Syntax: EOR.L Dy,<ea>x

Attributes: Size = longword

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 longword. The source operand must be a data register. The destination operand is specified in the effective address field. EORI is used when the source is immediate data.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb 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, Dy	1	1	0	Destination Effective Address	
								Mode	Register

Instruction Fields:

- Register field—Specifies any of the 8 data registers for the source operand, Dy.
- Destination Effective Address field—Specifies the destination operand, <ea>x; use addressing modes in the following table:

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

EORI

Exclusive-OR Immediate

First appeared in ISA_A

EORI

Operation: Immediate Data ^ Destination → Destination

Assembler Syntax: EORI.L #<data>,Dx

Attributes: Size = longword

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 data register, Dx. The size of the operation is specified as a longword. Note that the immediate data is contained in the two extension words, with the first extension word, bits [15:0], containing the upper word, and the second extension word, bits [15:0], containing the lower word.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb 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, Dx		
Upper Word of Immediate Data															
Lower Word of Immediate Data															

Instruction Fields:

- Register field - Destination data register, Dx.

Sign-Extend

First appeared in ISA_A

EXT, EXTB

Operation: Destination Sign-Extended → Destination

Assembler Syntax:

EXT.W Dx	extend byte to word
EXT.L Dx	extend word to longword
EXTB.L Dx	extend byte to longword

Attributes: Size = word, longword

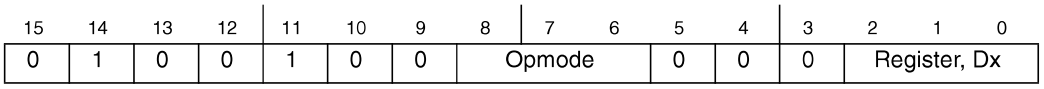
Description: Extends a byte in a data register, Dx, to a word or a longword, or a word in a data register to a longword, by replicating the sign bit to the left. When the EXT operation extends a byte to a word, bit 7 of the designated data register is copied to bits 15 – 8 of the data register. When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

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format:



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 longword
  - — 111 sign-extend low-order byte of data register to longword
- Register field—Specifies the data register, Dx, to be sign-extended.

FF1 Find First One in Register FF1

First appeared in ISA_C

Operation: Bit Offset of the First Logical One in Register → Destination

Assembler Syntax: FF1.L Dx

Attributes: Size = longword

Description: The data register, Dx, is scanned, beginning from the most-significant bit (Dx[31]) and ending with the least-significant bit (Dx[0]), searching for the first set bit. The data register is then loaded with the offset count from bit 31 where the first set bit appears, as shown below. If the source data is zero, then an offset of 32 is returned.

Table 0–1

Old Dx[31:0]	New Dx[31:0]
0x8000 0000	0x0000 0000
0x4000 0000	0x0000 0001
0x2000 0000	0x0000 0002
...	...
0x0000 0002	0x0000 001E
0x0000 0001	0x0000 001F
0x0000 0000	0x0000 0020

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb of the source operand is set; cleared otherwise
Z Set if the source 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	0	0	0	0	1	0	0	1	1	0	0	0	Destination Register, Dx

ILLEGAL Take Illegal Instruction Trap ILLEGAL

First appeared in ISA_A

Operation:
SP - 4 → SP; PC → (SP) (forcing stack to be longword aligned)
SP - 2 → SP; SR → (SP)
SP - 2 → SP; Vector Offset → (SP)
(VBR + 0x10) → PC

Assembler Syntax: ILLEGAL

Attributes: Unsized

Description: Execution of this instruction causes an illegal instruction exception. The opcode for ILLEGAL is 0x4AFC.

Starting with ISA_B (for devices which have an MMU), the Supervisor Stack Pointer (SSP) is used for this 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	1	1	1	1	0	0

Jump

First appeared in ISA_A

Operation: Destination Address → PC

Assembler Syntax: JMP <ea>y

Attributes: Unsized

Description: Program execution continues at the effective address specified by 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	1	Source Effective Address					
										Mode			Register		

Instruction Field:

- Source Effective Address field—Specifies the address of the next instruction, <ea>y; use the control addressing modes in the following table:

Addressing Mode	Mode	Register
Dy	—	—
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

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

JSR — Jump to Subroutine — JSR

First appeared in ISA_A

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

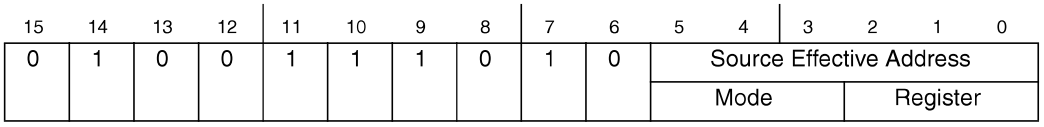
Assembler Syntax: JSR <ea>y

Attributes: Unsized

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

Condition Codes: Not affected

Instruction Format:



Instruction Field:

- Source Effective Address field—Specifies the address of the next instruction, <ea>y; use the control addressing modes in the following table:

Addressing Mode	Mode	Register
Dy	—	—
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

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

LEA — Load Effective Address — LEA

First appeared in ISA_A

Operation: <ea>y → Ax

Assembler Syntax: LEA.L <ea>y,Ax

Attributes: Size = longword

Description: Loads the effective address into the specified address register, Ax.

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, Ax	1	1	1	Source Effective Address	
								Mode	Register

Instruction Fields:

- Register field—Specifies the address register, Ax, to be updated with the effective address.
- Source Effective Address field—Specifies the address to be loaded into the destination address register; use the control addressing modes in the following table:

Addressing Mode	Mode	Register
Dy	—	—
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

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

LINK

Link and Allocate

First appeared in ISA_A

LINK

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

Assembler Syntax: LINK.W Ay,#<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. Note that although LINK is a word-sized instruction, most assemblers also support an unsized LINK.

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, Ay		
Word Displacement															

Instruction Fields:

- Register field—Specifies the address register, Ay, for the link.
- Displacement field—Specifies the two's complement integer to be added to the stack pointer.

Logical Shift

First appeared in ISA_A

LSL, LSR

Operation: Destination Shifted By Count → Destination

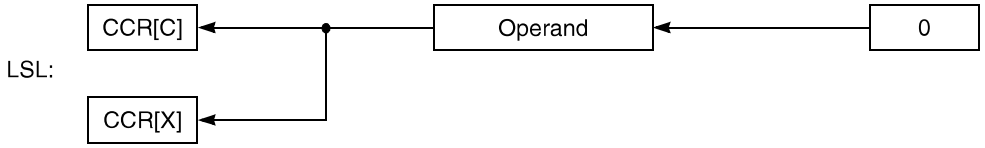
Assembler Syntax: LSd.L Dy,Dx
LSd.L #<data>,Dx
where d is direction, L or R

Attributes: Size = longword

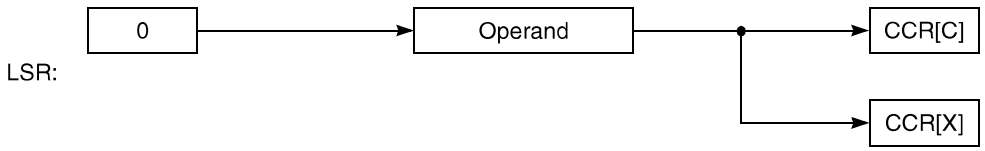
Description: Shifts the bits of the destination operand, Dx, in the direction (L or R) specified. The size of the operand is a longword. CCR[C] receives the last bit shifted out of the operand. The shift count is the number of bit positions to shift the destination register and may be specified in two different ways:

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

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 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:

15	14	13	12	11 10 9	8	7	6	5	4	3	2 1 0
1	1	1	0	Count or Register, Dy	dr	1	0	i/r	0	1	Register, Dx

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, Dy, specified in this field contains shift count (modulo 64)
- dr field—Specifies the direction of the shift:
  - 0 shift right
  - 1 shift left
- i/r field
  - 0 immediate shift count
  - 1 register shift count
- Register field—Specifies a data register, Dx, to be shifted.

MOV3Q

Move 3-Bit Data Quick
First appeared in ISA_B

MOV3Q

Operation: 3-bit Immediate Data → Destination

Assembler Syntax: MOV3Q.L #<data>,<ea>x

Attributes: Size = longword

Description: Move the immediate data to the operand at the destination location. The data range is from -1 to 7, excluding 0. The 3-bit immediate operand is sign extended to a longword operand and all 32 bits are transferred to the destination location.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if 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
1	0	1	0	Immediate Data	1	0	1	Destination Effective Address	
								Mode	Register

Instruction Fields:

- Immediate data field—3 bits of data having a range {-1,1-7} where a data value of 0 represents -1.
- Destination Effective Address field—Specifies the destination operand, <ea>x; use only data addressing modes listed in the following table:

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

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

MOVE — Move Data from Source to Destination — MOVE

First appeared in ISA_A

Operation: Source → Destination

Assembler Syntax: MOVE.sz <ea>y,<ea>x

Attributes: Size = byte, word, longword

Description: Moves the data at the source to the destination location and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or longword. MOVEA is used when the destination is an address register. MOVEQ is used to move an immediate 8-bit value to a data register. MOV3Q (supported starting with ISA_B) is used to move a 3-bit immediate value to any effective destination address.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if 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 Effective Address		Source Effective Address
			Register	Mode	Mode	Register

Instruction fields:

- Size field—Specifies the size of the operand to be moved:
  - — 01 byte operation
  - — 11 word operation
  - — 10 longword operation
- Destination Effective Address field—Specifies destination location, <ea>x; the table below lists possible data alterable addressing modes. The restrictions on combinations of source and destination addressing modes are listed in the table at the bottom of the next page.

Addressing Mode	Mode	Register
Dx	000	reg. number:Dx
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

Instruction fields (continued):

- Source Effective Address field—Specifies source operand, <ea>y; the table below lists possible addressing modes. The restrictions on combinations of source and destination addressing modes are listed in the table at the bottom of the next page.

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

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

NOTE

Not all combinations of source/destination addressing modes are possible. The table below shows the possible combinations. Starting with ISA_B, the combination of #<xxx>,d_{16}(Ax) can be used with MOVE.B and MOVE.W opcodes.

Source Addressing Mode	Destination Addressing Mode
Dy, Ay, (Ay), (Ay)+,-(Ay)	All possible
(d_{16}, Ay), (d16, PC)	All possible except (d_8, Ax, Xi), (xxx).W, (xxx).L
(d8, Ay, Xi), (d8, PC, Xi), (xxx).W, (xxx).L, #<xxx>	All possible except (d_{16}, Ax), (d_8, Ax, Xi), (xxx).W, (xxx).L

MOVEA Move Address from Source to Destination MOVEA

First appeared in ISA_A

Operation: Source → Destination

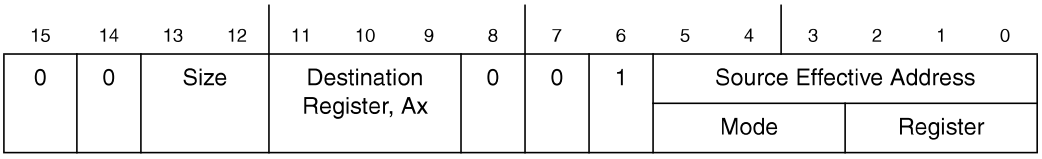
Assembler Syntax: MOVEA.sz <ea>y,Ax

Attributes: Size = word, longword

Description: Moves the address at the source to the destination address register. The size of the operation may be specified as word or longword. Word size source operands are sign extended to 32-bit quantities before the operation is done.

Condition Codes: Not affected

Instruction Format:



Instruction fields:

- Size field—Specifies the size of the operand to be moved:
  - 0x reserved
  - 11 word operation
  - 10 longword operation
- Destination Register field — Specifies the destination address register, Ax.
- Source Effective Address field—Specifies the source operand, <ea>y; the table below lists possible modes.

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

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

MOVEM — Move Multiple Registers — MOVEM

First appeared in ISA_A

Operation: Registers → Destination;
Source → Registers

Assembler Syntax: MOVEM.L #list,<ea>x
MOVEM.L <ea>y,#list

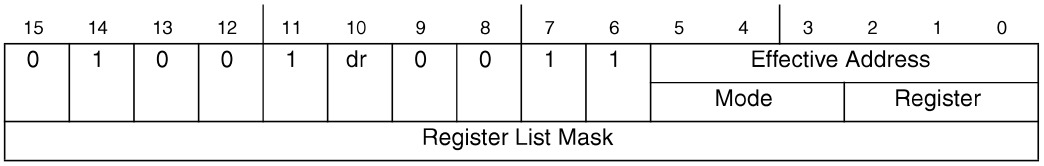
Attributes: Size = longword

Description: Moves the contents of selected registers to or from consecutive memory locations starting at the location specified by the effective address. A register is selected if the bit in the mask field corresponding to that register is set.

The registers are transferred starting at the specified address, and the address is incremented by the operand length (4) following each transfer. The order of the registers is from D0 to D7, then from A0 to A7.

Condition Codes: Not affected

Instruction Format:



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 the data transfer. For register-to-memory transfers, use the following table for <ea>x.

Addressing Mode	Mode	Register
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	—	—
– (Ax)	—	—
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	—	—

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

Instruction Fields (continued):

- Effective Address field (continued)—For memory-to-register transfers, use the following table for <ea>y.

Addressing Mode	Mode	Register
Dy	—	—
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

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

- 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

Move Quick

MOVEQ

First appeared in ISA_A

Operation: Immediate Data → Destination

Assembler Syntax: MOVEQ.L #<data>,Dx

Attributes: Size = longword

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

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if 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, Dx	0	Immediate Data

Instruction Fields:

- Register field—Specifies the data register, Dx, to be loaded.
- Data field—8 bits of data, which are sign-extended to a longword operand.

MOVE from CCR

Move from the Condition Code Register

First appeared in ISA_A

MOVE from CCR

Operation: CCR → Destination

Assembler Syntax: MOVE.W CCR,Dx

Attributes: Size = Word

Description: Moves the condition code bits (zero-extended to word size) to the destination location, Dx. 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, Dx		

Instruction Field:

- Register field - Specifies destination data register, Dx.

MOVE to CCR

Move to the Condition Code Register

First appeared in ISA_A

Operation: Source → CCR

Assembler Syntax: MOVE.B Dy,CCR
MOVE.B #<data>,CCR

Attributes: Size = Byte

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	Source Effective Address					
										Mode		Register			

Instruction Field:

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

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	—	—
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	—	—
(d_8,Ay,Xi)	—	—

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

MULS

Signed Multiply

First appeared in ISA_A

MULS

Operation: Source * Destination → Destination

Assembler Syntax:

MULS.W <ea>y,Dx	16 x 16 → 32
MULS.L <ea>y,Dx	32 x 32 → 32

Attributes: Size = word, longword

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

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

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

Note that CCR[V] is always cleared by MULS, unlike the 68K family processors.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format: (Word)

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
1	1	0	0	Register, Dx	1	1	1	Source Effective Address	
								Mode	Register

Instruction Fields (Word):

- Register field—Specifies the destination data register, Dx.
- Effective Address field—Specifies the source operand, <ea>y; use only those data addressing modes listed in the following table:

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

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

MULS Signed Multiply MULS

Instruction Format: (Longword)

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	Source Effective Address					
										Mode			Register		
0	Register, Dx			1	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields (Longword):

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

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

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

- Register field—Specifies a data register, Dx, 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

First appeared in ISA_A

MULU

Operation: Source * Destination → Destination

Assembler Syntax:

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

Attributes: Size = word, longword

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

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

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

Note that CCR[V] is always cleared by MULU, unlike the 68K family processors.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

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

Instruction Format: (Word)

15	14	13	12	11 10 9	8	7	6	5 4	3 2 1 0
1	1	0	0	Register, Dx	0	1	1	Source Effective Address	
								Mode	Register

Instruction Fields (Word):

- Register field—Specifies the destination data register, Dx.
- Effective Address field—Specifies the source operand, <ea>y; use only those data addressing modes listed in the following table:

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

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

Instruction Format: (Longword)

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	Source Effective Address					
										Mode			Register		
0	Register, Dx			0	0	0	0	0	0	0	0	0	0	0	0

Instruction Fields (Longword):

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

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

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

- Register field—Specifies a data register, Dx, 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.

MVS

Move with Sign Extend

First appeared in ISA_B

MVS

Operation: Source with sign extension → Destination

Assembler Syntax: MVS.sz <ea>y,Dx

Attributes: Size = byte, word

Description: Sign-extend the source operand and move to the destination register. For the byte operation, bit 7 of the source is copied to bits 31–8 of the destination. For the word operation, bit 15 of the source is copied to bits 31-16 of the destination.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if 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, Dx			1	0	Size	Source Effective Address					
										Mode		Register			

Instruction Fields:

- Register field—Specifies the destination data register, Dx.
- Size field—Specifies the size of the operation
  - — 0 byte operation
  - — 1 word operation
- Source Effective Address field—specifies the source operand, <ea>y; use only data addressing modes from the following table:

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

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

MVZ

Move with Zero-Fill

First appeared in ISA_B

MVZ

Operation: Source with zero fill → Destination

Assembler Syntax: MVZ.sz <ea>y,Dx

Attributes: Size = byte, word

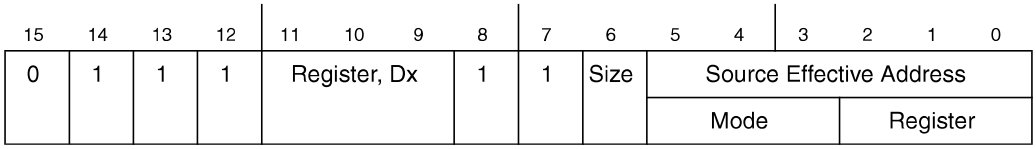
Description: Zero-fill the source operand and move to the destination register. For the byte operation, the source operand is moved to bits 7–0 of the destination and bits 31–8 are filled with zeros. For the word operation, the source operand is moved to bits 15–0 of the destination and bits 31–16 are filled with zeros.

Condition Codes:

X	N	Z	V	C
—	0	*	0	0

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

Instruction Format:



Instruction Fields:

- Register field—Specifies the destination data register, Dx.
- Size field—Specifies the size of the operation
  - — 0 byte operation
  - — 1 word operation
- Source Effective Address field—Specifies the source operand, <ea>y; use the following data addressing modes:

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

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

Negate

NEG

First appeared in ISA_A

Operation: 0 – Destination → Destination

Assembler Syntax: NEG.L Dx

Attributes: Size = longword

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

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit
N Set if the result is negative; cleared otherwise
Z Set if the result is zero; cleared otherwise
V Set if an overflow is generated; 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, Dx		

Instruction Fields:

- Register field - Specifies data register, Dx.

NEGX

Negate with Extend

First appeared in ISA_A

NEGX

Operation: 0 – Destination – CCR[X] → Destination

Assembler Syntax: NEGX.L Dx

Attributes: Size = longword

Description: Subtracts the destination operand and CCR[X] from zero. Stores the result in the destination location. The size of the operation is specified as a longword.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit
N Set if the result is negative; cleared otherwise
Z Cleared if the result is nonzero; unchanged otherwise
V Set if an overflow is generated; cleared otherwise
C Set if a borrow occurs; cleared otherwise

Normally CCR[Z] is set explicitly via programming before the start of an NEGX operation to allow successful testing for zero results upon completion of multiple-precision operations.

Instruction Format:

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

Instruction Fields:

- Register field - Specifies data register, Dx.

NOP

No Operation

First appeared in ISA_A

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, synchronizing the pipeline and preventing instruction overlap.

Because the NOP instruction is specified to perform a pipeline synchronization in addition to performing no operation, the execution time is multiple cycles. In cases where only code alignment is desired, it is preferable to use the TPF instruction, which operates as a 1-cycle no operation instruction. The opcode for NOP is 0x4E71.

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

First appeared in ISA_A

Operation: ~ Destination → Destination

Assembler Syntax: NOT.L Dx

Attributes: Size = longword

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 longword.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if 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, Dx		

Instruction Fields:

- Register field — Specifies data register, Dx.

Inclusive-OR Logical

First appeared in ISA_A

OR

Operation: Source | Destination → Destination

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

Attributes: Size = longword

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

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

In addition, ORI is used when the source is immediate data.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb 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 the data register.
- Opmode field:

Byte	Word	Longword	Operation
—	—	010	<ea>y I Dx → Dx
—	—	110	Dy I <ea>x → <ea>x

Instruction Fields (continued):

- Effective Address field—Determines addressing mode
  - For the source operand <ea>y, use addressing modes listed in the following table:

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

  - For the destination operand <ea>x, use addressing modes listed in the following table:

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

First appeared in ISA_A

Operation: Immediate Data | Destination → Destination

Assembler Syntax: ORI.L #<data>,Dx

Attributes: Size = longword

Description: Performs an inclusive-OR operation on the immediate data and the destination operand and stores the result in the destination data register, Dx. The size of the operation is specified as a longword. The size of the immediate data is specified as a longword. Note that the immediate data is contained in the two extension words, with the first extension word, bits [15:0], containing the upper word, and the second extension word, bits [15:0], containing the lower word.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb 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, Dx		
Upper Word of Immediate Data															
Lower Word of Immediate Data															

Instruction Fields:

- Destination register field - Specifies the destination data register, Dx.

PEA — Push Effective Address — PEA

First appeared in ISA_A

Operation: SP – 4 → SP; <ea>y → (SP)

Assembler Syntax: PEA.L <ea>y

Attributes: Size = longword

Description: Computes the effective address and pushes it onto the stack. The effective address is a longword 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	Source Effective Address					
										Mode			Register		

Instruction Field:

- Effective Address field—Specifies the address, <ea>y, to be pushed onto the stack; use only those control addressing modes listed in the following table:

Addressing Mode	Mode	Register
Dy	—	—
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	110	reg. number:Ay

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

PULSE Generate Unique Processor Status PULSE

First appeared in ISA_A

Operation: Set PST = 0x4

Assembler Syntax: PULSE

Attributes: Unsized

Description: Performs no operation. The processor state, other than the program counter, is unaffected. However, PULSE generates a special encoding of the Processor Status (PST) output pins, making it very useful for external triggering purposes. The opcode for PULSE is 0x4ACC.

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

REMS — Signed Divide Remainder — REMS

First appeared in ISA_A
Not implemented in MCF5202, MCF5204 and MCR5206

Operation: Destination/Source → Remainder

Assembler Syntax: REMS.L <ea>y,Dw:Dx 32-bit Dx/32-bit <ea>y Æ 32r in Dw
where r indicates the remainder

Attributes: Size = longword

Description: Divide the signed destination operand by the signed source and store the signed remainder in another register. If Dw is specified to be the same register as Dx, the DIVS instruction is executed rather than REMS. To determine the quotient, use DIVS.

An attempt to divide by zero results in a divide-by-zero exception and no registers are affected. The resulting exception stack frame points to the offending REMS opcode. If overflow is detected, the destination register is unaffected. An overflow occurs if the quotient is larger than a 32-bit signed integer.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- X Not affected
- N Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive
- Z Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero
- V Set if an overflow occurs; cleared otherwise
- 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	1	0	0	0	1	Source Effective Address					
										Mode			Register		
0	Register Dx			1	0	0	0	0	0	0	0	0	Register Dw		

Instruction Fields:

- Register Dx field—Specifies the destination register, Dx.
- Source Effective Address field— Specifies the source operand, <ea>y; use addressing modes in the following table:

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

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

- Register Dw field—Specifies the remainder register, Dw.

REMU Unsigned Divide Remainder REMU

First appeared in ISA_A
Not implemented in MCF5202, MCF5204 and MCF5206

Operation: Destination/Source → Remainder

Assembler Syntax: REMU.L <ea>y,Dw:Dx 32-bit Dx/32-bit <ea>y Æ 32r in Dw
where r indicates the remainder

Attributes: Size = longword

Description: Divide the unsigned destination operand by the unsigned source and store the unsigned remainder in another register. If Dw is specified to be the same register as Dx, the DIVU instruction is executed rather than REMU. To determine the quotient, use DIVU.

An attempt to divide by zero results in a divide-by-zero exception and no registers are affected. The resulting exception stack frame points to the offending REMU opcode. If overflow is detected, the destination register is unaffected. An overflow occurs if the quotient is larger than a 32-bit signed integer.

Condition Codes:

X	N	Z	V	C
—	*	*	*	0

- X Not affected
- N Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive
- Z Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero
- V Set if an overflow occurs; cleared otherwise
- 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	1	0	0	0	1	Source Effective Address					
										Mode			Register		
0	Register Dx			0	0	0	0	0	0	0	0	0	Register Dw		

Instruction Fields:

- Register Dx field—Specifies the destination register, Dx.
- Source Effective Address field— Specifies the source operand, <ea>y; use addressing modes in the following table:

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	010	reg. number:Ay
(Ay) +	011	reg. number:Ay
– (Ay)	100	reg. number:Ay
(d_{16},Ay)	101	reg. number:Ay
(d_8,Ay,Xi)	—	—

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

- Register Dw field—Specifies the remainder register, Dw.

RTS

Return from Subroutine

First appeared in ISA_A

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. The opcode for RTS is 0x4E75.

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

SATS

Signed Saturate

First appeared in ISA_B

SATS

Operation:

```
If CCR[V] == 1,
                then if Dx[31] == 0,
                         then Dx[31:0] = 0x80000000
                         else Dx[31:0] = 0x7FFFFFFF
                else Dx[31:0] is unchanged
```

Assembler Syntax: SATS.L Dx

Attributes: Size = longword

Description: Update the destination register only if the overflow bit of the CCR is set. If the operand is negative, then set the result to greatest positive number; otherwise, set the result to the largest negative value. The condition codes are set according to the result.

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	1	0	0	1	0	0	0	0	Register, Dx		

Instruction Fields:

- Register field—Specifies the destination data register, Dx.

Scc — Set According to Condition — Scc

First appeared in ISA_A

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

Assembler Syntax: Scc.B Dx

Attributes: Size = byte

Description: Tests the specified condition code; if the condition is true, sets the lowest byte of the destination data register to TRUE (all ones). Otherwise, sets that byte to FALSE (all zeros). Condition code cc specifies one of the following conditional tests, where C, N, V, and Z represent CCR[C], CCR[N], CCR[V], and CCR[Z], respectively:

Code	Condition	Encod-ing	Test	Code	Condition	Encod-ing	Test
CC(HS)	Carry clear	0100	$\overline{C}$	LS	Lower or same	0011	C I Z
CS(LO)	Carry set	0101	C	LT	Less than	1101	N & $\overline{V}$ I $\overline{N}$ & V
EQ	Equal	0111	Z	MI	Minus	1011	N
F	False	0001	0	NE	Not equal	0110	$\overline{Z}$
GE	Greater or equal	1100	N & V I $\overline{N}$ & $\overline{V}$	PL	Plus	1010	$\overline{N}$
GT	Greater than	1110	N & V & $\overline{Z}$ I $\overline{N}$ & $\overline{V}$ & $\overline{Z}$	T	True	0000	1
HI	High	0010	$\overline{C}$ & $\overline{Z}$	VC	Overflow clear	1000	$\overline{V}$
LE	Less or equal	1111	Z I N & $\overline{V}$ I $\overline{N}$ & V	VS	Overflow set	1001	V

Condition Codes: Not affected

Instruction Format:

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

Instruction Fields:

- Condition field—Binary code for one of the conditions listed in the table.
- Register field —Specifies the destination data register, Dx.

SUB — Subtract — SUB

First appeared in ISA_A

Operation: Destination – Source → Destination

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

Attributes: Size = longword

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 longword. The mode of the instruction indicates which operand is the source and which is the destination.

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

In addition, SUBA is used when the destination is an address register. SUBI and SUBQ are used when the source is immediate data.

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 an carry is generated; cleared otherwise

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	Register				Opmode		Effective Address					
										Mode		Register			

Instruction Fields:

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

Byte	Word	Longword	Operation
—	—	010	Dx - <ea>y → Dx
—	—	110	<ea>x - Dy → <ea>x

Instruction Fields (continued):

- Effective Address field—Determines addressing mode
  - For the source operand <ea>y, use addressing modes listed in the following table:

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

  - For the destination operand <ea>x, use addressing modes listed in the following table:

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

SUBA — Subtract Address — SUBA

First appeared in ISA_A

Operation: Destination - Source → Destination

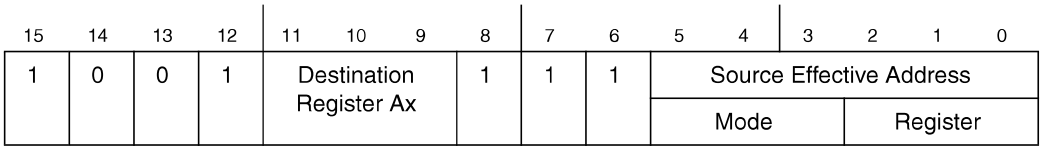
Assembler Syntax: SUBA.L <ea>y,Ax

Attributes: Size = longword

Description: Operates similarly to SUB, but is used when the destination is an address register rather than a data register. Subtracts the source operand from the destination address register and stores the result in the address register. The size of the operation is specified as a longword.

Condition Codes: Not affected

Instruction Format:



Instruction Fields:

- Destination Register field—Specifies the destination address register, Ax.
- Source Effective Address field— Specifies the source operand, <ea>y; use addressing modes listed in the following table:

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

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

SUBI — Subtract Immediate — SUBI

First appeared in ISA_A

Operation: Destination - Immediate Data → Destination

Assembler Syntax: SUBI.L #<data>,Dx

Attributes: Size = longword

Description: Operates similarly to SUB, but is used when the source operand is immediate data. Subtracts the immediate data from the destination operand and stores the result in the destination data register, Dx. The size of the operation is specified as longword. Note that the immediate data is contained in the two extension words, with the first extension word, bits [15:0], containing the upper word, and the second extension word, bits [15:0], containing the lower 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 an 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	0	0	1	0	0	0	0	Register, Dx		
Upper Word of Immediate Data															
Lower Word of Immediate Data															

Instruction Fields:

- Destination Register field—Specifies the destination data register, Dx.

SUBQ — Subtract Quick — SUBQ

First appeared in ISA_A

Operation: Destination - Immediate Data → Destination

Assembler Syntax: SUBQ.L #<data>,<ea>x

Attributes: Size = longword

Description: Operates similarly to SUB, but is used when the source operand is immediate data ranging in value from 1 to 8. Subtracts the immediate value from the operand at the destination location. The size of the operation is specified as longword. The immediate data is zero-filled to a longword before being subtracted from the destination. When adding to address registers, the condition codes are not altered.

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 an carry is generated; 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	Destination Effective Address	
								Mode	Register

Instruction Fields:

- Data field—3 bits of immediate data representing 8 values (0 – 7), with the immediate values 1-7 representing values of 1-7 respectively and 0 representing a value of 8.
- Destination Effective Address field—specifies the destination location; use only those alterable addressing modes listed in the following table:

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

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

SUBX — Subtract Extended — SUBX

First appeared in ISA_A

Operation: Destination - Source - CCR[X] → Destination

Assembler Syntax: SUBX.L Dy,Dx

Attributes: Size = longword

Description: Subtracts the source operand and CCR[X] from the destination operand and stores the result in the destination location. The size of the operation is specified as a longword.

Condition Codes:

X	N	Z	V	C
*	*	*	*	*

X Set the same as the carry bit
N Set if the result is negative; cleared otherwise
Z Cleared if the result is non-zero; unchanged otherwise
V Set if an overflow is generated; cleared otherwise
C Set if an carry is generated; cleared otherwise

Normally CCR[Z] is set explicitly via programming before the start of an SUBX operation to allow successful testing for zero results upon completion of multiple-precision operations.

Instruction Format:

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

Instruction Fields:

- Register Dx field—Specifies the destination data register, Dx.
- Register Dy field—Specifies the source data register, Dy.

SWAP — Swap Register Halves — SWAP

First appeared in ISA_A

Operation: Register[31:16] ↔ Register[15:0]

Assembler Syntax: SWAP.W Dx

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 msb 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	1	0	0	1	0	0	0	0	1	0	0	0	Register, Dx		

Instruction Fields:

- Register field—Specifies the destination data register, Dx.

TAS

Test and Set an Operand

First appeared in ISA_B

TAS

Operation: Destination Tested → CCR; 1 → bit 7 of Destination

Assembler Syntax: TAS.B <ea>x

Attributes: Size = byte

Description: Tests and sets the byte operand addressed by the effective address field. The instruction tests the current value of the operand and sets CCR[N] and CCR[Z] appropriately. TAS also sets the high-order bit of the operand. The operand uses a read-modify-write memory cycle that completes the operation without interruption. This instruction supports use of a flag or semaphore to coordinate several processors. Note that, unlike 68K Family processors, Dx is not a supported addressing mode.

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the msb of the operand was set; cleared otherwise
Z Set if the operand was zero; cleared otherwise
V Always cleared
C Always cleared

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	1	1	Destination Effective Address					
										Mode			Register		

Instruction Fields:

- Destination Effective Address field—Specifies the destination location, <ea>x; the possible data alterable addressing modes are listed in the table below.

Addressing Mode	Mode	Register
Dx	—	—
Ax	—	—
(Ax)	010	reg. number:Ax
(Ax) +	011	reg. number:Ax
– (Ax)	100	reg. number:Ax
(d_{16},Ax)	101	reg. number:Ax
(d_8,Ax,Xi)	110	reg. number:Ax

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

TPF — Trap False — TPF

First appeared in ISA_A

Operation: No Operation

Assembler Syntax:

TPF	PC + 2 → PC
TPF.W #<data>	PC + 4 → PC
TPF.L #<data>	PC + 6 → PC

Attributes: Size = unsized, word, longword

Description: Performs no operation. TPF can occupy 16, 32, or 48 bits in instruction space, effectively providing a variable-length, single-cycle, no operation instruction. When code alignment is desired, TPF is preferred over the NOP instruction, as the NOP instruction also synchronizes the processor pipeline, resulting in multiple-cycle operation.

TPF.{W,L} can be used for elimination of unconditional branches, for example:

```
      if (a == b)
              z = 1;
              else
                                                    z = 2;
```

which typically compiles to:

```
        cmp.l   d0,d1              ; compare a == b
        beq.b   label0             ; branch if equal
        movq.l  #2,d2              ; z = 2
        bra.b   label1             ; continue
label0:
        movq.l  #1,d2              ; z = 1
label1:
```

For this type of sequence, the BRA.B instruction can be replaced with a TPF.W or TPF.L opcode (depending on the length of the instruction at label0 - in this case, a TPF.W opcode would be applicable). The instruction(s) at the first label effectively become packaged as extension words of the TPF instruction, and the branch is completely eliminated.

Condition Codes: Not affected

Instruction Format:

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

Instruction Fields:

- Opmode field—Specifies the number of optional extension words.
  - 010 one extension word
  - 011 two extension words
  - 100 no extension words

TRAP — Trap — TRAP

First appeared in ISA_A

Operation: 1 → S-Bit of SR
SP – 4 → SP; nextPC → (SP); SP – 2 → SP;
SR → (SP); SP – 2 → SP; Format/Offset → (SP);
(VBR + 0x80 + 4*n) → PC
where n is the TRAP vector number

Assembler Syntax: TRAP #<vector>

Attributes: Unsized

Description: Causes a TRAP #<vector> exception. The TRAP vector field is multiplied by 4 and then added to 0x80 to form the exception address. The exception address is then added to the VBR to index into the exception vector table. The vector field value can be 0 – 15, providing 16 vectors.

Note when SR is copied onto the exception stack frame, it represents the value at the beginning of the TRAP instruction's execution. At the conclusion of the exception processing, the SR is updated to clear the T bit and set the S bit.

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.

Test an Operand

First appeared in ISA_A

Operation: Source Operand Tested → CCR

Assembler Syntax: TST.sz <ea>y

Attributes: Size = byte, word, longword

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

Condition Codes:

X	N	Z	V	C
—	*	*	0	0

X Not affected
N Set if the operand Is negative; cleared otherwise
Z Set if the operand was zero; cleared otherwise
V Always cleared
C Always cleared

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	0	1	0	Size		Destination Effective Address					
										Mode			Register		

Instruction Fields:

- Size field—Specifies the size of the operation:
  - — 00 byte operation
  - — 01 word operation
  - — 10 longword operation
  - — 11 word operation
- Destination Effective Address field—Specifies the addressing mode for the destination operand, <ea>x, as listed in the following table:

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

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

* The Ax addressing mode is allowed only for word and longword operations.

UNLK — Unlink — UNLK

First appeared in ISA_A

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

Assembler Syntax: UNLK Ax

Attributes: Unsized

Description: Loads the stack pointer from the specified address register, then loads the address register with the longword 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, Ax		

Instruction Field:

- Register field—Specifies the address register, Ax, for the instruction.

Chapter 8
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. Supervisor instructions for optional core modules (for example, the floating-point unit) are also detailed in this section.

Not all instructions are supported by all ColdFire processors. See Chapter 3, "Instruction Set Summary for specific details on the instruction set definitions.

MOVE from SR

Move from the Status Register

First appeared in ISA_A

MOVE from SR

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

Assembler Syntax: MOVE.W SR,Dx

Attributes: Size = word

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

Condition Codes: Not affected

Instruction Format:

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

Instruction Field:

- Register field—Specifies the destination data register, Dx.

MOVE from USP — Move from User Stack Pointer — MOVE from USP

First appeared in ISA_B

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

Assembler Syntax: MOVE.L USP,Ax

Attributes: Size = longword

Description: Moves the contents of the user stack pointer to the specified address register. If execution of this instruction is attempted on a processor implementing ISA_A, or on the MCF5407, an illegal instruction exception will be taken. For all processors, this instruction executes correctly if CACR[EUSP] is set.

Condition Codes: Not affected

Instruction Format:

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

Instruction Field:

- Register field—Specifies the destination address register, Ax.

MOVE to SR

Move to the Status Register

First appeared in ISA_A

MOVE to SR

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

Assembler Syntax: MOVE.W <ea>y,SR

Attributes: Size = word

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

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	1	0	1	1	Source Effective Address					
										Mode		Register			

Instruction Field:

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

Addressing Mode	Mode	Register
Dy	000	reg. number:Dy
Ay	—	—
(Ay)	—	—
(Ay) +	—	—
– (Ay)	—	—
(d_{16},Ay)	—	—
(d_8,Ay,Xi)	—	—

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

MOVE to USP

Move to User Stack Pointer

First appeared in ISA_B

MOVE to USP

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

Assembler Syntax: MOVE.L Ay,USP

Attributes: Size = longword

Description: Moves the contents of an address register to the user stack pointer. If execution of this instruction is attempted on a processor implementing ISA_A, or on the MCF5407, an illegal instruction exception will be taken. For all processors, this instruction executes correctly if CACR[EUSP] is set.

Condition Codes: Not affected

Instruction Format:

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

Instruction Field:

- Register field—Specifies the source address register, Ay.

RTE — Return from Exception — RTE

First appeared in ISA_A

Operation:

If Supervisor State
    Then 2 + (SP) → SR; 4 + (SP) → PC; SP + 8 → SP
    Adjust stack according to format
Else Privilege Violation Exception

Assembler Syntax: RTE

Attributes: Unsized

Description: Loads the processor state information stored in the exception stack frame located at the top of the stack into the processor. The instruction examines the stack format field in the format/offset word to determine how much information must be restored. Upon returning from exception, the processor is in user mode if SR[S]=0 when it is loaded from memory; otherwise, the processor remains in supervisor mode. Note that this instruction synchronizes the pipeline.

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

Chapter 11
Exception Processing

This chapter describes exception processing for the ColdFire family.

11.1 Overview

Exception processing for ColdFire processors is streamlined for performance. Differences from previous M68000 Family processors include the following:

- A simplified exception vector table
- Reduced relocation capabilities using the vector base register
- A single exception stack frame format
- Use of a single, self-aligning stack pointer (for ISA_A implementations only)

Beginning with th eV4 core, support for an optional virtual memory management unit is provided. For devices containing an MMU, the exception processing is slightly modified. Differences from previous ColdFire Family processors are related to the instruction restart model for translation (TLB miss) and access faults. This functionality extends the original ColdFire access error fault vector and exception stack frames.

Earlier ColdFire processors (V2 and V3) use an instruction restart exception model but require additinoal software support to recover from certain access errors.

Exception processing can be defined as the time from the detection of the fault condition until the fetch of the first handler instruction has been initiated. It consists of the following four major steps:

1. The processor makes an internal copy of the status register (SR) and then enters supervisor mode by setting SR[S] and disabling trace mode by clearing SR[T]. The occurrence of an interrupt exception also clears SR[M] and sets the interrupt priority mask, SR[I] to the level of the current interrupt request.
2. The processor determines the exception vector number. For all faults except interrupts, the processor bases this calculation on exception type. For interrupts, the processor performs an interrupt acknowledge (IACK) bus cycle to obtain the vector number from peripheral. The IACK cycle is mapped to a special acknowledge address space with the interrupt level encoded in the address.
3. The processor saves the current context by creating an exception stack frame on the system stack. Processors implementing ISA_A support a single stack pointer in the A7 address register; therefore, there is no notion of separate supervisor and user stack pointers. As a result, the exception stack frame is created at a 0-modulo-4 address on top of the current system stack. For processors implementing all other ISA revisions and supporting 2 stack pointers, the exception stack frame is created at a 0-modulo-4 address on opt of the system stack pointed to by the Supervisor Stack Pointer (SSP). All ColdFire processors use a simplified fixed-length stack frame, shown in Figure 11-1, for all exceptions. In addition, processor cores supporting an MMU use the same fixed-length stack frame with additional fault status (FS) encodings to support the MMU. In some exception types, the program counter (PC) in the exception stack frame contains the address of the faulting instruction (fault); in others, the PC contains the next instruction to be executed (next).

If the exception is caused by an FPU instruction, the PC contains the address of either the next floating-point instruction (nextFP) if the exception is pre-instruction, or the faulting instruction (fault) if the exception is post-instruction.

4. The processor acquires the address of the first instruction of the exception handler. The instruction address is obtained by fetching a value from the exception table at the address in the vector base register. The index into the table is calculated as 4 x vector_number. When the index value is generated, the vector table contents determine the address of the first instruction of the desired handler. After the fetch of the first opcode of the handler is initiated, exception processing terminates and normal instruction processing continues in the handler.

The vector base register described in Section 1.5.3, "Vector Base Register (VBR)," holds the base address of the exception vector table in memory. The displacement of an exception vector is added to the value in this register to access the vector table. VBR[19–0] are not implemented and are assumed to be zero, forcing the vector table to be aligned on a 0-modulo-1-Mbyte boundary.

ColdFire processors support a 1024-byte vector table as shown in Table 11-1. The table contains 256 exception vectors, the first 64 of which are defined by Motorola. The rest are user-defined interrupt vectors.

Table 11-1. Exception Vector Assignments

Vector Numbers	Vector Offset (Hex)	Stacked Program Counter[1]	Assignment
0	000	—	Initial stack pointer (SSP for cores with dual stack pointers)
1	004	—	Initial program counter
2	008	Fault	Access error
3	00C	Fault	Address error
4	010	Fault	Illegal instruction
5[2]	014	Fault	Divide by zero
6–7	018–01C	—	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[3]	030	Next	Non-PC breakpoint debug interrupt
13[3]	034	Next	PC breakpoint debug interrupt
14	038	Fault	Format error
15	03C	Next	Uninitialized interrupt
16–23	040–05C	—	Reserved
24	060	Next	Spurious interrupt
25–31[4]	064–07C	Next	Level 1–7 autovectored interrupts
32–47	080–0BC	Next	Trap #0–15 instructions

Table 11-1. Exception Vector Assignments (Continued)

Vector Numbers	Vector Offset (Hex)	Stacked Program Counter[1]	Assignment
48[5]	0C0	Fault	Floating-point branch on unordered condition
49[5]	0C4	NextFP or Fault	Floating-point inexact result
50[5]	0C8	NextFP	Floating-point divide-by-zero
51[5]	0CC	NextFP or Fault	Floating-point underflow
52[5]	0D0	NextFP or Fault	Floating-point operand error
53[5]	0D4	NextFP or Fault	Floating-point overflow
54[5]	0D8	NextFP or Fault	Floating-point input not-a-number (NAN)
55[5]	0DC	NextFP or Fault	Floating-point input denormalized number
56–60	0E0–0F0	—	Reserved
61[6]	0F4	Fault	Unsupported instruction
62–63	0F8–0FC	—	Reserved
64–255	100–3FC	Next	User-defined interrupts

1 'Fault' refers to the PC of the faulting instruction. 'Next' refers to the PC of the instruction immediately after the faulting instruction. 'NextFP' refers to the PC of the next floating-point instruction.

2 If the divide unit is not present (5202, 5204, 5206), vector 5 is reserved.

3 On V2 and V3, all debug interrupts use vector 12; vector 13 is reserved.

4 Support for autovectored interrupts is dependent on the interrupt controller implementation. Consult the specific device reference manual for additional details.

5 If the FPU is not present, vectors 48 - 55 are reserved.

6 Some devices do not support this exception; refer to Table 11-3.

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

11.1.1 Supervisor/User Stack Pointers (A7 and OTHER_A7)

Some ColdFire cores support two unique stack pointer (A7) registers: the supervisor stack pointer (SSP) and the user stack pointer (USP). This support provides the required isolation between operating modes. Note that only the SSP is used during creation of the exception stack frame.

The hardware implementation of these two program-visible 32-bit registers does not uniquely identify one as the SSP and the other as the USP. Rather, the hardware uses one 32-bit register as the currently-active A7 and the other as OTHER_A7. Thus, the register contents are a function of the processor operating mode:

```
if SR[S] = 1
        then
                A7 = Supervisor Stack Pointer
                other_A7 = User Stack Pointer
else
        A7 = User Stack Pointer
        other_A7 = Supervisor Stack Pointer
```

The BDM programming model supports reads and writes to A7 and OTHER_A7 directly. It is the responsibility of the external development system to determine the mapping of A7 and OTHER_A7 to the two program-visible definitions (SSP and USP), based on the setting of SR[S]. This functionality is enabled by setting by the dual stack pointer enable bit CACR[DSPE]. If this bit is cleared, only the stack pointer, A7 (defined for previous ColdFire versions), is available. DSPE is zero at reset.

If DSPE is set, the appropriate stack pointer register (SSP or USP) is accessed as a function of the processor's operating mode. To support dual stack pointers, the following two privileged MC680x0 instructions to load/store the USP are added to the ColdFire instruction set architecture as part of ISA_B:

```
mov.l Ay,USP # move to USP: opcode = 0x4E6(0xxx)

mov.l USP,Ax # move from USP: opcode = 0x4E6(1xxx)
```

The address register number is encoded in the low-order three bits of the opcode.

11.1.2 Exception Stack Frame Definition

The first longword of the exception stack frame, Figure 11-1, holds the 16-bit format/vector word (F/V) and 16-bit status register. The second holds the 32-bit program counter address.

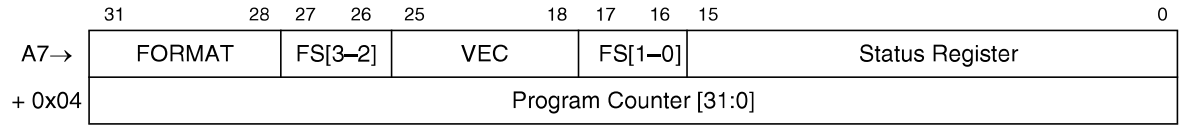


Figure 11-1. Exception Stack Frame

Table 11-2 describes F/V fields. FS encodings added to support the MMU are noted.

Table 11-2. Format/Vector Word

Bits	Field	Description
31–28	FORMAT	Format field. Written with a value of {4,5,6,7} by the processor indicating a 2-longword frame format. FORMAT records any longword stack pointer misalignment when the exception occurred.

A7 at Time of Exception, Bits[1:0]	A7 at First Instruction of Handler	FORMAT
00	Original A7—8	0100
01	Original A7—9	0101
10	Original A7—10	0110
11	Original A7—11	0111

Table 11-2. Format/Vector Word (Continued)

Bits	Field	Description
27–26	FS[3–2]	Fault status. Defined for access and address errors and for interrupted debug service routines. 0000 Not an access or address error nor an interrupted debug service routine 0001 Reserved 0010 Interrupt during a debug service routine for faults other than access errors. (V4 and beyond, if MMU)[1] 0011 Reserved 0100 Error (for example, protection fault) on instruction fetch 0101 TLB miss on opword of instruction fetch (V4 and beyond, if MMU) 0110 TLB miss on extension word of instruction fetch (V4 and beyond, if MMU) 0111 IFP access error while executing in emulator mode (V4 and beyond, if MMU) 1000 Error on data write 1001 Error on attempted write to write-protected space 1010 TLB miss on data write (V4 and beyond, if MMU) 1011 Reserved 1100 Error on data read 1101 Attempted read, read-modify-write of protected space(V4 and beyond, if MMU) 1110 TLB miss on data read, or read-modify-write (V4 and beyond, if MMU) 1111 OEP access error while executing in emulator mode (V4 and beyond, if MMU)
25–18	VEC	Vector number. Defines the exception type. It is calculated by the processor for internal faults and is supplied by the peripheral for interrupts. See Table 11-1.
17–16	FS[1–0]	See bits 27–26.

[1] This generally refers to taking an I/O interrupt while in a debug service routine but also applies to other fault types. If an access error occurs during a debug service routine, FS is set to 0111 if it is due to an instruction fetch or to 1111 for a data access. This applies only to access errors with the MMU present. If an access error occurs without an MMU, FS is set to 0010.

11.1.3 Processor Exceptions

Table 11-3 describes ColdFire core exceptions. Note that if a ColdFire processor encounters any fault while processing another fault, it immediately halts execution with a catastrophic fault-on-fault condition. A reset is required to force the processor to exit this halted state.

11.1.4 Floating-Point Arithmetic Exceptions

This section describes floating-point arithmetic exceptions; Table 11-3 lists these exceptions in order of priority:

Table 11-3. Exception Priorities

Priority	Exception
1	Branch/set on unordered (BSUN)
2	Input Not-a-Number (INAN)
3	Input denormalized number (IDE)
4	Operand error (OPERR)
5	Overflow (OVFL)
6	Underflow (UNFL)

Field	Description
10 CEIB	Cache enable non-cacheable instruction bursting. Setting this bit enables the line-fill buffer to be loaded with burst transfers under control of CLNF[1:0] for non-cacheable accesses. Non-cacheable accesses are never written into the memory array. 0 Disable burst fetches on non-cacheable accesses 1 Enable burst fetches on non-cacheable accesses
8 DBWE	Default buffered write enable. This bit defines the default value for enabling buffered writes. If DBWE = 0, the termination of an operand write cycle on the processor's local bus is delayed until the external bus cycle is completed. If DBWE = 1, the write cycle on the local bus is terminated immediately and the operation buffered in the bus controller. In this mode, operand write cycles are effectively decoupled between the processor's local bus and the external bus. Generally, enabled buffered writes provide higher system performance but recovery from access errors can be more difficult. For the ColdFire core, reporting access errors on operand writes is always imprecise and enabling buffered writes further decouples the write instruction and the signaling of the fault 0 Disable buffered writes 1 Enable buffered writes
4 EUSP	Enable user stack pointer. See Section 3.2.3, “Supervisor/User Stack Pointers (A7 and OTHER_A7)”for more information on the dual stack pointer implementation. 0 Disable the processor’s use of the User Stack Pointer 1 Enable the processor’s use of the User Stack Pointer
1–0 CLNF	Cache line fill. These bits control the size of the memory request the cache issues to the bus controller for different initial instruction line access offsets.

3.3.5 Instruction Execution Timing

This section presents processor instruction execution times in terms of processor-core clock cycles. The number of operand references for each instruction is enclosed in parentheses following the number of processor clock cycles. Each timing entry is presented as C(R/W) where:

- C is the number of processor clock cycles, including all applicable operand fetches and writes, and all internal core cycles required to complete the instruction execution.
- R/W is 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 the assumptions concerning the timing values and the execution time details.

3.3.5.1 Timing Assumptions

For the timing data presented in this section, these assumptions apply:

1. The OEP is loaded with the opword and all required extension words at the beginning of each instruction execution. This implies that the OEP does not wait for the IFP to supply opwords and/or extension words.
2. The OEP does not experience any sequence-related pipeline stalls. The most common example of stall involves consecutive store operations, excluding the MOVEM instruction. For all STORE operations (except MOVEM), certain hardware resources within the processor are marked as busy for two clock cycles after the final decode and select/operand fetch cycle (DSOC) of the store instruction. If a subsequent STORE instruction is encountered within this 2-cycle window, it is

stalled until the resource again becomes available. Thus, the maximum pipeline stall involving consecutive STORE operations is two 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 that 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; for example, 16-bit operands aligned on 0-modulo-2 addresses, 32-bit operands aligned on 0-modulo-4 addresses.

   The processor core decomposes misaligned operand references into a series of aligned accesses as shown in Table 3-11.

Table 3-11. Misaligned Operand References

address[1:0]	Size	Bus Operations	Additional C(R/W)
01 or 11	Word	Byte, Byte	2(1/0) if read 1(0/1) if write
01 or 11	Long	Byte, Word, Byte	3(2/0) if read 2(0/2) if write
10	Long	Word, Word	2(1/0) if read 1(0/1) if write

3.3.5.2 MOVE Instruction Execution Times

Table 3-12 lists execution times for MOVE.{B,W} instructions; Table 3-13 lists timings for MOVE.L.

NOTE

For all tables in this section, the execution time of any instruction using the PC-relative effective addressing modes is the same for the comparable An-relative mode.

ET with {<ea> = (d16,PC)}	equals ET with {<ea> = (d16,An)}
ET with {<ea> = (d8,PC,Xi*SF)}	equals ET with {<ea> = (d8,An,Xi*SF)}

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

Table 3-12. MOVE Byte and Word Execution Times

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*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)

Table 3-12. MOVE Byte and Word Execution Times (continued)

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*SF)	xxx.wl
(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,Xi*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,Xi*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 3-13. MOVE Long Execution Times

Source	Destination						
	Rx	(Ax)	(Ax)+	-(Ax)	(d16,Ax)	(d8,Ax,Xi*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,Xi*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,Xi*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)	—	—	—

3.3.5.3 Standard One Operand Instruction Execution Times

Table 3-14. One Operand Instruction Execution Times

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn*SF)	xxx.wl	#xxx
BITREV	Dx	1(0/0)	—	—	—	—	—	—	—
BYTEREV	Dx	1(0/0)	—	—	—	—	—	—	—

Table 3-14. One Operand Instruction Execution Times (continued)

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)	—	—	—	—	—	—	—
FF1	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)

3.3.5.4 Standard Two Operand Instruction Execution Times

Table 3-15. 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>,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)
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)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—

Table 3-15. Two Operand Instruction Execution Times (continued)

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xn*SF) (d8,PC,Xn*SF)	xxx.wl	#xxx
BCHG	#imm,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—	—
BCLR	Dy,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
BCLR	#imm,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—	—
BSET	Dy,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	5(1/1)	4(1/1)	—
BSET	#imm,<ea>	2(0/0)	4(1/1)	4(1/1)	4(1/1)	4(1/1)	—	—	—
BTST	Dy,<ea>	2(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	4(1/0)	3(1/0)	—
BTST	#imm,<ea>	1(0/0)	3(1/0)	3(1/0)	3(1/0)	3(1/0)	—	—	—
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)	—	—	—	—	—	—	—
DIVS.W	<ea>,Dx	20(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	24(1/0)	23(1/0)	20(0/0)
DIVU.W	<ea>,Dx	20(0/0)	23(1/0)	23(1/0)	23(1/0)	23(1/0)	24(1/0)	23(1/0)	20(0/0)
DIVS.L	<ea>,Dx	≤35(0/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
DIVU.L	<ea>,Dx	≤35(0/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
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.L	#imm,Dx	—	—	—	—	—	—	—	1(0/0)
OR.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)
OR.L	Dy,<ea>	—	3(1/1)	3(1/1)	3(1/1)	3(1/1)	4(1/1)	3(1/1)	—
ORI.L	#imm,Dx	1(0/0)	—	—	—	—	—	—	—
REMS.L	<ea>,Dx	≤35(0/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
REMU.L	<ea>,Dx	≤35(0/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	≤38(1/0)	—	—	—
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)	—	—	—	—	—	—	—

3.3.5.5 Miscellaneous Instruction Execution Times

Table 3-16. Miscellaneous Instruction Execution Times

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn*SF)	xxx.wl	#xxx
CPUSHL	(Ax)	—	11(0/1)	—	—	—	—	—	—
LINK.W	Ay,#imm	2(0/1)	—	—	—	—	—	—	—
MOVE.L	Ay,USP	3(0/0)	—	—	—	—	—	—	—
MOVE.L	USP,Ax	3(0/0)	—	—	—	—	—	—	—
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) [2]
MOVEC	Ry,Rc	9(0/1)	—	—	—	—	—	—	—
MOVEM.L	<ea>, and list	—	1+n(n/0)	—	—	1+n(n/0)	—	—	—
MOVEM.L	and list,<ea>	—	1+n(0/n)	—	—	1+n(0/n)	—	—	—
NOP		3(0/0)	—	—	—	—	—	—	—
PEA	<ea>	—	2(0/1)	—	—	2(0/1) [4]	3(0/1) [5]	2(0/1)	—
PULSE		1(0/0)	—	—	—	—	—	—	—
STLDSR	#imm	—	—	—	—	—	—	—	5(0/1)
STOP	#imm	—	—	—	—	—	—	—	3(0/0) [3]
TRAP	#imm	—	—	—	—	—	—	—	15(1/2)
TPF		1(0/0)	—	—	—	—	—	—	—
TPF.W		1(0/0)	—	—	—	—	—	—	—
TPF.L		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)	—
WDEBUG	<ea>	—	5(2/0)	—	—	5(2/0)	—	—	—

[1]The n is the number of registers moved by the MOVEM opcode.

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

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

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

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

3.3.5.6 EMAC Instruction Execution Times

Table 3-17. EMAC Instruction Execution Times

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An)	(d8,An, Xn*SF)	xxx.wl	#xxx
MAC.L	Ry, Rx, Raccx	1(0/0)	—	—	—	—	—	—	—
MAC.L	Ry, Rx, <ea>, Rw, Raccx	—	(1/0)	(1/0)	(1/0)	(1/0)[1]	—	—	—
MAC.W	Ry, Rx, Raccx	1(0/0)	—	—	—	—	—	—	—
MAC.W	Ry, Rx, <ea>, Rw, Raccx	—	(1/0)	(1/0)	(1/0)	(1/0)[1]	—	—	—
MOVE.L	<ea>y, Raccx	1(0/0)	—	—	—	—	—	—	1(0/0)
MOVE.L	Raccy, Raccx	1(0/0)	—	—	—	—	—	—	—
MOVE.L	<ea>y, MACSR	5(0/0)	—	—	—	—	—	—	5(0/0)
MOVE.L	<ea>y, Rmask	4(0/0)	—	—	—	—	—	—	4(0/0)
MOVE.L	<ea>y,Raccext01	1(0/0)	—	—	—	—	—	—	1(0/0)
MOVE.L	<ea>y,Raccext23	1(0/0)	—	—	—	—	—	—	1(0/0)
MOVE.L	Raccx, <ea>x	1(0/0)[2]	—	—	—	—	—	—	—
MOVE.L	MACSR, <ea>x	1(0/0)	—	—	—	—	—	—	—
MOVE.L	Rmask, <ea>x	1(0/0)	—	—	—	—	—	—	—
MOVE.L	Raccext01,<ea.x	1(0/0)	—	—	—	—	—	—	—
MOVE.L	Raccext23,<ea>x	1(0/0)	—	—	—	—	—	—	—
MSAC.L	Ry, Rx, Raccx	1(0/0)	—	—	—	—	—	—	—
MSAC.W	Ry, Rx, Raccx	1(0/0)	—	—	—	—	—	—	—
MSAC.L	Ry, Rx, <ea>, Rw, Raccx	—	(1/0)	(1/0)	(1/0)	(1/0)[1]	—	—	—
MSAC.W	Ry, Rx, <ea>, Rw, Raccx	—	(1/0)	(1/0)	(1/0)	(1/0)[1]	—	—	—
MULS.L	<ea>y, Dx	4(0/0)	(1/0)	(1/0)	(1/0)	(1/0)	—	—	—
MULS.W	<ea>y, Dx	4(0/0)	(1/0)	(1/0)	(1/0)	(1/0)	(1/0)	(1/0)	4(0/0)
MULU.L	<ea>y, Dx	4(0/0)	(1/0)	(1/0)	(1/0)	(1/0)	—	—	—
MULU.W	<ea>y, Dx	4(0/0)	(1/0)	(1/0)	(1/0)	(1/0)	(1/0)	(1/0)	4(0/0)

1 Effective address of (d16,PC) not supported

2 Storing an accumulator requires one additional processor clock cycle when saturation is enabled, or fractional rounding is performed (MACSR[7:4] equals 1---, -11-, --11)

NOTE

The execution times for moving the contents of the Racc, Raccext[01,23], MACSR, or Rmask into a destination location <ea>x shown in this table represent the best-case scenario when the store instruction is executed and there are no load or M{S}AC instructions in the EMAC execution pipeline. In general, these store operations require only a single cycle for execution, but if preceded immediately by a load, MAC, or MSAC instruction, the depth of the EMAC pipeline is exposed and the execution time is four cycles.

3.3.5.7 Branch Instruction Execution Times

Table 3-18. General Branch Instruction Execution Times

Opcode	<EA>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xi*SF) (d8,PC,Xi*SF)	xxx.wl	#xxx
BRA		—	—	—	—	2(0/1)	—	—	—
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		—	—	10(2/0)	—	—	—	—	—
RTS		—	—	5(1/0)	—	—	—	—	—

Table 3-19. Bcc Instruction Execution Times

Opcode	Forward Taken	Forward Not Taken	Backward Taken	Backward Not Taken
Bcc	3(0/0)	1(0/0)	2(0/0)	3(0/0)