## Freescale Semiconductor，Inc．

# ColdFire® ${ }^{\circledR}$ Family Programmer＇s Reference Manual 

（A）MOTOROLA

# Freescale Semiconductor, Inc. 

ColdFire is a registered trademark and DigitalDNA is a trademark of Motorola, Inc.
$I^{2} \mathrm{C}$ is a registered trademark of Philips Semiconductors


#### Abstract

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and $(\mathbb{A})$ are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.


How to reach us:
USA/EUROPE/Locations Not Listed: Motorola Literature Distribution; P.O. Box 5405, Denver, Colorado 80217. 1-303-675-2140 or 1-800-441-2447
JAPAN: Motorola Japan Ltd.; SPS, Technical Information Center, 3-20-1, Minami-Azabu. Minato-ku, Tokyo 106-8573 Japan.
81-3-3440-3569
ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852-26668334

Technical Information Center: 1-800-521-6274

HOME PAGE: http://www.motorola.com/semiconductors

Document Comments: FAX (512) 9335-2625, Attn: RISC Applications Engineering
World Wide Web Addresses: http://www.motorola.com/PowerPC
http://www.motorola.com/NetComm
http://www.motorola.com/ColdFire

MAC User Instructions

FPU User Instructions

Supervisor Instructions

Instruction Format Summary

Exception Processing

Addressing Capabilities

Instruction Set Summary

Integer User Instructions

EMAC User Instructions

7
FPU User Instructions

Supervisor Instructions

9
Instruction Format Summary

PST/DDATA Encodings

11 Exception Processing

Processor Instruction Summary

S-Record Output Format

## CONTENTS

| Paragraph <br> Number | Title |
| :--- | :---: |
|  | Chapter 1 <br> Introduction |

1.1 Integer Unit User Programming Model ..... 1-1
1.1.1 Data Registers (D0-D7) ..... 1-2
1.1.2 Address Registers (A0-A7) ..... 1-2
1.1.3 Program Counter (PC) ..... 1-2
1.1.4 Condition Code Register (CCR) ..... 1-2
1.2 Floating-point Unit User Programming Model. ..... 1-4
1.2.1 Floating-Point Data Registers (FP0-FP7) ..... 1-4
1.2.1.1 Floating-Point Control Register (FPCR) ..... 1-4
1.2.2 Floating-Point Status Register (FPSR) ..... 1-5
1.2.3 Floating-Point Instruction Address Register (FPIAR) ..... 1-6
1.3 MAC User Programming Model ..... 1-7
1.3.1 MAC Status Register (MACSR) ..... 1-7
1.3.2 MAC Accumulator (ACC). ..... 1-8
1.3.3 MAC Mask Register (MASK) ..... 1-8
1.4 EMAC User Programming Model ..... 1-8
1.4.1 MAC Status Register (MACSR) ..... 1-8
1.4.2 MAC Accumulators (ACC[0:3]) ..... 1-9
1.4.3 Accumulator Extensions (ACCext01, ACCext23) ..... 1-11
1.4.4 MAC Mask Register (MASK) ..... 1-11
1.5 Supervisor Programming Model. ..... 1-11
1.5.1 Status Register (SR) ..... 1-12
1.5.2 Supervisor/User Stack Pointers (A7 and OTHER_A7) ..... 1-13
1.5.3 Vector Base Register (VBR). ..... 1-14
1.5.4Cache Control Register (CACR)1-14
1.5.5 Address Space Identifier (ASID) ..... 1-14
1.5.6 Access Control Registers (ACR0-ACR3) ..... 1-14
1.5.7 MMU Base Address Register (MMUBAR) ..... 1-14
1.5.8 RAM Base Address Registers (RAMBAR0/RAMBAR1) ..... 1-15
1.5.9 ROM Base Address Registers (ROMBAR0/ROMBAR1) ..... 1-15
1.5.10 Module Base Address Register (MBAR) ..... 1-15
1.6 Integer Data Formats. ..... 1-16
1.7 Floating-Point Data Formats ..... 1-16
1.7.1 Floating-Point Data Types ..... 1-17

## Freescale Semiconductor, Inc.

## CONTENTS

| Paragraph | Page |  |
| :--- | :--- | ---: |
| Number | Title | Number |

## Page Number

1.7.1.1 Normalized Numbers ..... 1-17
1.7.1.2 Zeros ..... 1-17
1.7.1.3 Infinities ..... 1-17
1.7.1.4 Not-A-Number ..... 1-18
1.7.1.5 Denormalized Numbers ..... 1-18
1.7.2 FPU Data Format and Type Summary ..... 1-18
1.8 Multiply Accumulate Data Formats. ..... 1-20
1.9 Organization of Data in Registers ..... 1-20
1.9.1 Organization of Integer Data Formats in Registers ..... 1-20
1.9.2 Organization of Integer Data Formats in Memory ..... 1-22
Chapter 2
Addressing Capabilities
2.1 Instruction Format. ..... 2-1
2.2 Effective Addressing Modes ..... 2-2
2.2.1 Data Register Direct Mode ..... 2-3
2.2.2 Address Register Direct Mode ..... 2-3
2.2.3 Address Register Indirect Mode ..... 2-3
2.2.4 Address Register Indirect with Postincrement Mode ..... 2-4
2.2.5 Address Register Indirect with Predecrement Mode ..... 2-4
2.2.6 Address Register Indirect with Displacement Mode ..... 2-5
2.2.7 Address Register Indirect with Scaled Index and 8-Bit Displacement Mode ..... 2-6
2.2.8 Program Counter Indirect with Displacement Mode ..... 2-6
2.2.9 Program Counter Indirect with Scaled Index and 8-Bit Displacement Mode ..... 2-7
2.2.10 Absolute Short Addressing Mode ..... 2-8
2.2.11 Absolute Long Addressing Mode ..... 2-9
2.2.12 Immediate Data ..... 2-9
2.2.13 Effective Addressing Mode Summary ..... 2-10
2.3 Stack ..... 2-10
Chapter 3 Instruction Set Summary
3.1 Instruction Summary ..... 3-1
3.1.1 Data Movement Instructions ..... 3-4
3.1.2 Integer Arithmetic Instructions ..... 3-5
3.1.3 Logical Instructions ..... 3-7
3.1.4 Shift Instructions ..... 3-7
3.1.5 Bit Manipulation Instructions ..... 3-8
3.1.6 Program Control Instructions. ..... 3-8
3.1.7 System Control Instructions. ..... 3-10

# Freescale Semiconductor, Inc. CONTENTS 

ParagraphTitlePage Number
3.1.8 Cache Maintenance Instructions ..... 3-10
3.1.9 Floating Point Arithmetic Instructions ..... 3-11
3.2 Instruction Set Additions ..... 3-12
Chapter 4
Integer User Instructions
Chapter 5Multiply-Accumulate Unit (MAC) User Instructions
Chapter 6Enhanced Multiply-Accumulate Unit (EMAC) User Instructions
Chapter 7Floating-Point Unit (FPU) User Instructions
7.1 Floating-Point Status Register (FPSR) ..... 7-1
7.2 Conditional Testing ..... 7-3
7.3 Instruction Results when Exceptions Occur ..... 7-6
7.4 Instruction Descriptions ..... 7-7
Chapter 8
Supervisor (Privileged) Instructions
Chapter 9Instruction Format Summary
9.1 Operation Code Map ..... 9-1
Chapter 10PST/DDATA Encodings
10.1 User Instruction Set. ..... 10-1
10.2 Supervisor Instruction Set ..... 10-7
Chapter 11
Exception Processing
11.1 Overview ..... 11-1
11.1.1 Supervisor/User Stack Pointers (A7 and OTHER_A7) ..... 11-4

## Freescale Semiconductor, Inc.

## CONTENTS

Paragraph Title Page Number Number
11.1.2 Exception Stack Frame Definition ..... 11-4
11.1.3 Processor Exceptions ..... 11-5
11.1.4 Floating-Point Arithmetic Exceptions ..... 11-9
11.1 .5 Branch/Set on Unordered (BSUN) ..... 11-11
11.1.6 Input Not-A-Number (INAN) ..... 11-11
11.1.7 Input Denormalized Number (IDE) ..... 11-11
11.1.8 Operand Error (OPERR) ..... 11-12
11.1.9 Overflow (OVFL) ..... 11-13
11.1 .10 Underflow (UNFL) ..... 11-13
11.1.11 Divide-by-Zero (DZ) ..... 11-14
11.1.12 Inexact Result (INEX) ..... 11-14
11.1.13 V4 Changes to the Exception Processing Model ..... 11-15
Chapter 12
Processor Instruction Summary
Appendix A
S-Record Output Format
A. 1 S-Record Content ..... A-1
A. 2 S-Record Types ..... A-2
A. 3 S-Record Creation ..... A-3

## Freescale Semiconductor, Inc.

## ILLUSTRATIONS

Figure Number

## Title

Page Number
1-1 ColdFire Family User Programming Model ..... 1-2
1-2 Condition Code Register (CCR) ..... 1-3
1-3 ColdFire Family Floating-point Unit User Programming Model ..... 1-4
1-4 Floating-Point Control Register (FPCR) ..... 1-4
1-5 Floating-Point Status Register (FPSR) ..... 1-5
1-6 MAC Unit Programming Model ..... 1-7
1-7 MAC Status Register (MACSR) ..... 1-7
1-8 EMAC Programming Model ..... 1-8
1-9 MAC Status Register (MACSR) ..... 1-9
1-10 EMAC Fractional Alignment ..... 1-10
1-11 EMAC Signed and Unsigned Integer Alignment ..... 1-10
1-12 Accumulator 0 and 1 Extensions (ACCext01) ..... 1-11
1-13 Accumulator 2 and 3 Extensions (ACCext01) ..... 1-11
1-14 Supervisor Programming Model ..... 1-12
1-15 Status Register (SR) ..... 1-13
1-16 Vector Base Register (VBR) ..... 1-14
1-17 MMU Base Address Register (MMUBAR) ..... 1-15
1-18 Module Base Address Register (MBAR) ..... 1-16
1-19 Normalized Number Format ..... 1-17
1-20 Zero Format ..... 1-17
1-21 Infinity Format ..... 1-17
1-22 Not-a-Number Format ..... 1-18
1-23 Denormalized Number Format ..... 1-18
1-24 Two's Complement, Signed Fractional Equation ..... 1-20
1-25 Organization of Integer Data Format in Data Registers ..... 1-21
1-26 Organization of Addresses in Address Registers ..... 1-21
1-27 Memory Operand Addressing ..... 1-22
1-28 Memory Organization for Integer Operands ..... 1-22
2-1 Instruction Word General Format ..... 2-1
2-2 Instruction Word Specification Formats ..... 2-2
2-3 Data Register Direct ..... 2-3
2-4 Address Register Direct ..... 2-3
2-5 Address Register Indirect ..... 2-4
2-6 Address Register Indirect with Postincrement ..... 2-4
2-7 Address Register Indirect with Predecrement ..... 2-5
2-8 Address Register Indirect with Displacement ..... 2-5

## Freescale Semiconductor, Inc.

## ILLUSTRATIONS

Figure
Title Page Number
2-9 Address Register Indirect with Scaled Index and 8-Bit Displacement ..... 2-6
2-10 Program Counter Indirect with Displacement ..... 2-7
2-11 Program Counter Indirect with Scaled Index and 8-Bit Displacement ..... 2-8
2-12 Absolute Short Addressing ..... 2-8
2-13 Absolute Long Addressing ..... 2-9
2-14 Immediate Data Addressing ..... 2-9
2-15 Stack Growth from High Memory to Low Memory ..... 2-11
2-16 Stack Growth from Low Memory to High Memory ..... 2-11
7-1 Floating-Point Status Register (FPSR) ..... 7-1
11-1 Exception Stack Frame ..... 11-5

## Freescale Semiconductor, Inc.

## TABLES

| Table | Pitle | Page |
| :--- | ---: | ---: |
| Number |  | Number |

Title

1-1 CCR Bit Descriptions ..... 1-3
1-2 FPCR Field Descriptions ..... 1-5
1-3 FPSR Field Descriptions ..... 1-5
1-4 MACSR Field Descriptions ..... 1-7
1-5 MACSR Field Descriptions ..... 1-9
1-6 Implemented Supervisor Registers by Device ..... 1-12
1-7 Status Field Descriptions ..... 1-13
1-8 MMU Base Address Register Field Descriptions ..... 1-15
1-9 MBAR Field Descriptions ..... 1-16
1-10 Integer Data Formats ..... 1-16
1-11 Real Format Summary ..... 1-19
2-1 Instruction Word Format Field Definitions ..... 2-2
2-2 Immediate Operand Location ..... 2-9
2-3 Effective Addressing Modes and Categories ..... 2-10
3-1 Notational Conventions ..... 3-2
3-2 Data Movement Operation Format ..... 3-5
3-3 Integer Arithmetic Operation Format ..... 3-6
3-4 Logical Operation Format ..... 3-7
3-5 Shift Operation Format ..... 3-8
3-6 Bit Manipulation Operation Format ..... 3-8
3-7 Program Control Operation Format ..... 3-9
3-8 System Control Operation Format ..... 3-10
3-9 Cache Maintenance Operation Format ..... 3-11
3-10 Dyadic Floating-Point Operation Format ..... 3-11
3-11 Dyadic Floating-Point Operations ..... 3-11
3-12 Monadic Floating-Point Operation Format ..... 3-12
3-13 Monadic Floating-Point Operations ..... 3-12
3-14 ColdFire User Instruction Set Summary ..... 3-12
3-15 ColdFire Supervisor Instruction Set Summary ..... 3-17
3-16 ColdFire ISA_B Additions Summary ..... 3-18
3-17 MAC Instruction Set Summary ..... 3-19
3-18 EMAC Instruction Set Enhancements Summary ..... 3-19
3-19 Floating-Point Instruction Set Summary ..... 3-20
7-1 FPSR Field Descriptions ..... 7-1
7-2 FPSR EXC Bits ..... 7-3
7-3 FPCC Encodings ..... 7-4

## Freescale Semiconductor, Inc.

## TABLES

| Table | Page |  |
| :--- | :--- | ---: |
| Number | Title | Number |

7-4 Floating-Point Conditional Tests ..... 7-5
7-5 FPCR EXC Byte Exception Enabled/Disabled Results ..... 7-6
7-6 Data Format Encoding ..... 7-8
8-1 State Frames ..... 8-3
8-2 State Frames ..... 8-5
8-3 ColdFire CPU Space Assignments ..... 8-14
9-1 Operation Code Map ..... 9-1
10-1 PST/DDATA Specification for User-Mode Instructions ..... 10-2
10-2 PST/DDATA Values for User-Mode Multiply-Accumulate Instructions ..... 10-5
10-3 PST/DDATA Values for User-Mode Floating-Point Instructions ..... 10-6
10-4 Data Markers and FPU Operand Format Specifiers ..... 10-7
10-5 PST/DDATA Specifications for Supervisor-Mode Instructions ..... 10-7
11-1 Exception Vector Assignments ..... 11-2
11-2 Format/Vector Word ..... 11-5
11-3 Exceptions ..... 11-6
11-4 Exception Priorities ..... 11-9
11-5 BSUN Exception Enabled/Disabled Results ..... 11-11
11-6 INAN Exception Enabled/Disabled Results ..... 11-11
11-7 IDE Exception Enabled/Disabled Results ..... 11-12
11-8 Possible Operand Errors ..... 11-12
11-9 OPERR Exception Enabled/Disabled Results ..... 11-12
11-10 OVFL Exception Enabled/Disabled Results ..... 11-13
11-11 UNFL Exception Enabled/Disabled Results ..... 11-14
11-12 DZ Exception Enabled/Disabled Results ..... 11-14
11-13 Inexact Rounding Mode Values ..... 11-14
11-14 INEX Exception Enabled/Disabled Results ..... 11-15
11-15 OEP EX Cycle Operations ..... 11-16
12-1 Standard Products ..... 12-1
12-2 ColdFire Instruction Set and Processor Cross-Reference ..... 12-2
12-3 ColdFire MAC and EMAC Instruction Sets ..... 12-4
12-4 ColdFire FPU Instruction Set ..... 12-5

## Freescale Semiconductor, Inc.

## Chapter 1 Introduction

This manual contains detailed information about software instructions used by the Version 2 (V2), Version 3 (V3), and Version 4 (V4) ColdFire ${ }^{\circledR}$ microprocessors.

The ColdFire Family programming model consists of two register groups: user and supervisor. Programs executing in the user mode use only the registers in the user group. System software executing in the supervisor mode can access all registers and use the control registers in the supervisor group to perform supervisor functions. The following paragraphs provide a brief description of the registers in the user and supervisor models as well as the data organization in the registers.

### 1.1 Integer Unit User Programming Model

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

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


Data registers

Address registers

Stack pointer
Program counter
Condition code register

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.

## Freescale Semiconductor, Inc.

Integer Unit User Programming Model
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 <br> branch acceleration logic in the instruction fetch pipeline on forward conditional <br> branches. Refer to a V3 core or device user's manual for further information on this bit. |
|  | - | Reserved, should be cleared (Versions 2 and 4). |
| $6-5$ | - | Reserved, should be cleared. |
| 4 | X | Extend. Set to the value of the C-bit for arithmetic operations; otherwise not affected or <br> set to a specified result. |
| 3 | N | Negative. Set if the most significant bit of the result is set; otherwise cleared. <br> 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 <br> 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, <br> 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:

- 864-bit floating-point data registers (FP0-FP7)
- 32-bit floating-point control register (FPCR)
- 32-bit floating-point status register (FPSR)
- 32-bit floating-point instruction address register (FPIAR)


Floating-point data registers

Floating-point control register
Floating-point status register
Floating-point instruction address register

Figure 1-3. ColdFire Family Floating-Point Unit User Programming Model

### 1.2.1 Floating-Point Data Registers (FP0-FP7)

Floating-point data registers are analogous to the integer data registers for the $68 \mathrm{~K} /$ ColdFire family. The 64-bit floating-point data registers always contain numbers in double-precision format. All external operands, regardless of the source data format, are converted to double-precision values before being used in any calculation or being stored in a floating-point data register. A reset or a null-restore operation sets FP0-FP7 to positive, nonsignaling not-a-numbers (NANs).

### 1.2.1.1 Floating-Point Control Register (FPCR)

The FPCR, Figure 1-4, contains an exception enable byte (EE) and a mode control byte (MC). The user can read or write to FPCR using FMOVE or FRESTORE. A processor reset or a restore operation of the null state clears the FPCR. When this register is cleared, the FPU never generates exceptions.


Figure 1-4. Floating-Point Control Register (FPCR)
Table 1-2 describes FPCR fields.

# Freescale Semiconductor, Inc. 

Floating-Point Unit User Programming Model
Table 1-2. FPCR Field Descriptions


### 1.2.2 Floating-Point Status Register (FPSR)

The FPSR, Figure 1-5, contains a floating-point condition code byte (FPCC), a floating-point exception status byte (EXC), and a floating-point accrued exception byte (AEXC). The user can read or write all FPSR bits. Execution of most floating-point instructions modifies FPSR. FPSR is loaded by using FMOVE or FRESTORE. A processor reset or a restore operation of the null state clears the FPSR.


Figure 1-5. Floating-Point Status Register (FPSR)
Table 1-3 describes FPSR fields.
Table 1-3. FPSR Field Descriptions


Freescale Semiconductor, Inc.

Table 1-3. FPSR Field Descriptions (Continued)

| Bits | Field | Description |  |
| :---: | :---: | :---: | :---: |
| 26 | FPPC | Z | Zero |
| 25 | (c) | 1 | Infinity |
| 24 |  | NAN | Not-a-number |
| 23-16 | - | Reserved, should be cleared. |  |
| 15-8 | EXC | Exception status byte. Contains a bit for each floating-point exception that might have occurred during the most recent arithmetic instruction or move operation. |  |
| 15 |  | BSUN | Branch/set on unordered |
| 14 |  | INAN | Input not-a-number |
| 13 |  | OPERR | Operand error |
| 12 |  | OVFL | Overflow |
| 11 |  | UNFL | Underflow |
| 10 |  | DZ | Divide by zero |
| 9 |  | INEX | Inexact operation |
| 8 |  | IDE | Input denormalization |
| 7-0 | AEXC | Accrued exception byte. Contains 5 exception bits the IEEE 754 standard requires for exception-disabled operations. These exceptions are logical combinations of bits in the EXC byte. AEXC records all floating-point exceptions since the user last cleared AEXC. |  |
| 7 |  | IOP | Invalid operation |
| 6 |  | OVFL | Underflow |
| 5 |  | UNFL | Divide By Zero |
| 4 |  | DZ | Inexact Operation |
| 3 |  | INEX | Input Denormalization |
| 2-0 | - | Reserved, should be cleared. |  |

### 1.2.3 Floating-Point Instruction Address Register (FPIAR)

The ColdFire operand execution pipeline can execute integer and floating-point instructions simultaneously. As a result, the PC value stacked by the processor in response to a floating-point exception trap may not point to the instruction that caused the exception.

For those FPU instructions that can generate exception traps, the 32-bit FPIAR is loaded with the instruction PC address before the FPU begins execution. In case of an FPU exception, the trap handler can use the FPIAR contents to determine the instruction that generated the exception. FMOVE to/from the FPCR, FPSR, or FPIAR and FMOVEM instructions cannot generate floating-point exceptions and so do not modify FPIAR. A reset or a null-restore operation clears FPIAR.

## Freescale Semiconductor, Inc.

### 1.3 MAC User Programming Model

The following paragraphs describe the registers for the optional ColdFire MAC unit. Figure 1-6 illustrates the user programming model for the MAC unit. It contains the following registers:

- 32-bit MAC status register (MACSR)
- 32-bit accumulator register (ACC)
- 32-bit MAC mask register (MASK)


Figure 1-6. MAC Unit Programming Model

### 1.3.1 MAC Status Register (MACSR)

The MACSR, shown in Figure 1-7, contains an operational mode field and a set of flags.

| 31 | $7{ }^{7}$ |  |  |  | 3 |  |  | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | eratio | al Mo |  |  |  |  |  |
| - | OMC | S/U | F/I | R/T | N | Z | V | C |

Figure 1-7. MAC Status Register (MACSR)
Table 1-4 describes MACSR fields.
Table 1-4. MACSR Field Descriptions

| Bits | Field | Description |  |
| :---: | :---: | :---: | :---: |
| 31-8 | - | Reserved, should be cleared. |  |
| 7-4 | OMF | Operational mode field. Defines the operating configuration of the MAC unit. |  |
| 7 |  | OMC | Overflow/saturation mode |
| 6 |  | S/U | Signed/unsigned operations |
| 5 |  | F/I | Fraction/integer mode |
| 4 |  | R/T | Round/truncate mode |
| 3-0 | Flags | Flags. Contains indicator flags from the last MAC instruction execution. |  |
| 3 |  | N | Negative |
| 2 |  | Z | Zero |
| 1 |  | V | Overflow |
| 0 |  | C | Carry. This field is always zero. |

### 1.3.2 MAC Accumulator (ACC)

This 32-bit register contains the results of MAC operations.

### 1.3.3 MAC Mask Register (MASK)

The mask register (MASK) is 32 bits of which only the low-order 16 bits are implemented. When MASK is loaded, the low-order 16 bits of the source operand are loaded into the register. When it is stored, the upper 16 bits are forced to all ones.

When used by an instruction, this register 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.

### 1.4 EMAC User Programming Model

The following paragraphs describe the registers for the optional ColdFire EMAC unit. Figure 1-8 illustrates the user programming model for the EMAC unit. It contains the following registers:

- One 32-bit MAC status register (MACSR) including four indicator bits signaling product or accumulation overflow (one for each accumulator: PAV0-PAV3)
- Four 32-bit accumulators ( $\mathrm{ACCx}=\mathrm{ACC} 0, \mathrm{ACC} 1, \mathrm{ACC} 2, \mathrm{ACC} 3)$
- Eight 8-bit accumulator extensions (two per accumulator), packaged as two 32-bit values for load and store operations (ACCext01, ACCext23)
- One 32-bit mask register (MASK)


Figure 1-8. EMAC Programming Model

### 1.4.1 MAC Status Register (MACSR)

Figure 1-9 shows the EMAC MACSR, which contains an operational mode field and two sets of flags.

# Freescale Semiconductor, Inc. 

EMAC User Programming Model


Figure 1-9. MAC Status Register (MACSR)
Table 1-5 describes EMAC MACSR fields.
Table 1-5. MACSR Field Descriptions

| Bits | Field | Description |  |
| :---: | :---: | :---: | :---: |
| 31-12 | - | Reserved, should be cleared. |  |
| 11-8 | PAV $x$ | Product/accumulation overflow flags, one per accumulator |  |
| 7-4 | OMF | Operational mode field. Defines the operating configuration of the EMAC unit. |  |
| 7 |  | OMC | Overflow/saturation mode |
| 6 |  | S/U | Signed/unsigned operations |
| 5 |  | F/I | Fraction/integer mode |
| 4 |  | R/T | Round/truncate mode |
| 3-0 | Flags | Flags. Contains indicator flags from the last MAC instruction execution. |  |
| 3 |  | N | Negative |
| 2 |  | Z | Zero |
| 1 |  | V | Overflow |
| 0 |  | C | Carry. This field is always zero. |

### 1.4.2 MAC Accumulators (ACC[0:3])

The EMAC implements four 48 -bit accumulators. The 32 -bit ACCx registers, along with the accumulator extension words, contain the accumulator data. Figure 1-10 shows the data contained by the accumulator and accumulator extension words when the EMAC is operating in fractional mode. The upper 8 bits of the extended product are sign-extended from the 40 -bit result taken from the product.
motorola

Freescale Semiconductor, Inc.


Figure 1-10. EMAC Fractional Alignment
Figure 1-11 shows the data contained by the accumulator and accumulator extension words when the EMAC is operating in signed or unsigned integer mode. In signed mode, the upper 8 bits of the extended product are sign extended from the 40 -bit result taken from the product. In unsigned mode, the upper 8 bits of the extended product are all zeros.


Figure 1-11. EMAC Signed and Unsigned Integer Alignment

## Freescale Semiconductor, Inc.

### 1.4.3 Accumulator Extensions (ACCext01, ACCext23)

The 32-bit accumulator extension registers (ACCext01, ACCext23) allow the complete contents of the 48-bit accumulator to be saved and restored on context switches. Figure 1-12 shows how the ACC 0 and ACC 1 data is stored when loaded into a register. Refer to Figure 1-10 and Figure 1-11 for information on the data contained in the extension bytes.

| 2423 |  | 1615 |  | 7 |
| :---: | :---: | :---: | :---: | :---: |
| ACC1 Upper <br> Extension Byte | ACC1 Lower <br> Extension Byte | ACC0 Upper <br> Extension Byte | ACC0 Lower <br> Extension Byte |  |

Figure 1-12. Accumulator 0 and 1 Extensions (ACCext01)
Figure 1-13 shows how the ACC 2 and ACC 3 data is stored when loaded into a register. Refer to Figure 1-10 and Figure 1-11 for information on the data contained in the extension bytes.

|  | 2423 |  | 16 |  | 7 |
| :---: | :---: | :---: | :---: | :---: | :---: |
| ACC3 Upper <br> Extension Byte | ACC3 Lower <br> Extension Byte | ACC2 Upper <br> Extension Byte | ACC2 Lower <br> Extension Byte |  |  |

Figure 1-13. Accumulator 2 and 3 Extensions (ACCext01)

### 1.4.4 MAC Mask Register (MASK)

Only the low-order 16 bits of the 32 -bit mask register (MASK) are implemented. When MASK is loaded, the low-order 16 bits of the source operand are loaded into the register. When it is stored, the upper 16 bits are forced to all ones.

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.

Freescale Semiconductor, Inc.

| 31 | 1915 |  |  |
| :---: | :---: | :---: | :---: |
|  | (CCR) | SR | Status register |
|  |  | OTHER_A7 | Supervisor A7 stack pointer |
|  | Must be zeros | VBR | Vector base register |
|  |  | CACR | Cache control register |
|  |  | ASID | Address space ID register |
|  |  | ACR0 | Access control register 0 (data) |
|  |  | ACR1 | Access control register 1 (data) |
|  |  | ACR2 | Access control register 2 (instruction) |
|  |  | ACR3 | Access control register 3 (instruction) |
|  |  | MMUBAR | MMU base address register |
|  |  | ROMBAR0 | ROM base address register 0 |
|  |  | ROMBAR1 | ROM base address register 1 |
|  |  | RAMBAR0 | RAM base address register 0 |
|  |  | RAMBAR1 | RAM base address register 1 |
|  |  | MBAR | Module base address register |

Figure 1-14. Supervisor Programming Model
Note that not all registers are implemented on every ColdFire device; refer to Table 1-6. Future devices will implement registers that are not implemented on current devices.

Table 1-6. Implemented Supervisor Registers by Device

| Name | $\mathbf{5 2 0 2}$ | $\mathbf{5 2 0 4}$ | $\mathbf{5 2 0 6}$ <br> $\mathbf{5 2 0 6 e}$ | $\mathbf{5 2 7 2}$ | $\mathbf{5 3 0 7}$ | $\mathbf{5 4 0 7}$ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| SR | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| OTHER_A7 |  |  |  |  |  |  |
| VBR | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| CACR | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| ASID |  |  |  |  |  |  |
| ACR0 | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| ACR1 | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| ACR2 |  |  |  |  |  | $\checkmark$ |
| ACR3 |  |  |  |  |  | $\checkmark$ |
| MMUBAR |  |  |  |  |  |  |
| ROMBAR0 |  |  |  |  |  |  |
| ROMBAR1 |  |  |  |  |  |  |
| RAMBAR0 |  | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ | $\checkmark$ |
| RAMBAR1 |  |  |  |  |  | $\checkmark$ |
| MBAR |  | $\checkmark$ | $\checkmark$ | $\checkmark$ | $V$ | $\checkmark$ |

### 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 0x27xx after reset.

## Freescale Semiconductor, Inc.

Supervisor Programming Model

${ }^{1}$ The P bit is implemented only on the V 3 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 <br> 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 <br> priority levels less than or equal to the current priority, except the edge-sensitive level-7 request, <br> 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

The V2 and V3 architectures support a single stack pointer (A7). The initial value of A7 is loaded from the reset exception vector, address offset 0 .

The V4 architecture supports 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 as dictated by the virtual memory management scheme provided by the memory management unit (MMU). (Note that a device without an MMU, such as V2 and V3, has a single stack pointer.)

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
```

Freescale Semiconductor, Inc.

### 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, freezing, 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.

### 1.5.6 Access Control Registers (ACRO-ACR3)

The access control registers (ACR[0:3]) define attributes for four user-defined memory regions. ACR0 and ACR1 control data memory space and ACR2 and ACR3 control instruction memory space. Attributes include definition of cache mode, write protect, and buffer write enables. Not all ColdFire processors implement all four ACRs. Bit functions and positions may vary among ColdFire processor implementations. Refer to a specific device or core user's manual for further information.

### 1.5.7 MMU Base Address Register (MMUBAR)

MMUBAR, shown in Figure 1-17, defines a memory-mapped, privileged data-only space with the highest priority in effective address attribute calculation for the data internal memory bus (that is, the MMUBAR has priority over RAMBAR0). If virtual mode is enabled, any normal mode access that does not hit in the MMUBAR, RAMBARs, ROMBARs, or ACRs is considered a normal-mode, virtual address request and generates its access attributes from the MMU. MMUBAR is only available if a device has an MMU.

Refer to a specific device or core user's manual for further information.
BA -

Figure 1-17. MMU Base Address Register
Table 1-8 describes MMU base address register fields.
Table 1-8. MMU Base Address Register Field Descriptions

| Bits | Name | Description |
| :---: | :---: | :--- |
| $31-16$ | BA | Base address. Defines the base address for the 64-Kbyte address space mapped to the MMU. |
| $15-1$ | - | Reserved, should be cleared. |
| 0 | V | Valid |

### 1.5.8 RAM Base Address Registers (RAMBAR0/RAMBAR1)

RAMBAR registers determine the base address of the internal SRAM modules and indicate the types of references mapped to each. Each RAMBAR includes a base address, write-protect bit, address space mask bits, and an enable bit. RAM base address alignment is implementation specific. A specific ColdFire processor may implement 2 , 1 , or 0 RAMBARs. Bit functions and positions can vary among ColdFire processor implementations. Refer to a specific device or core user's manual for further information.

### 1.5.9 ROM Base Address Registers (ROMBAR0/ROMBAR1)

ROMBAR registers determine the base address of the internal ROM modules and indicate the types of references mapped to each. Each ROMBAR includes a base address, write-protect bit, address space mask bits, and an enable bit. ROM base address alignment is implementation specific. A specific ColdFire processor may implement 2 , 1 , or 0 ROMBARs. Bit functions and positions can vary among ColdFire processor implementations. 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.

## Freescale Semiconductor, Inc.

| 31 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| BA | - | WP | - | AM | C/I | SC | SD | UC | UD | V |  |  |

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 | ddress. Defines the base address for a 4-Kbyte address range. |
| 11-9 | - | Rese | ed, should be cleared. |
| 8-1 | AMB | Attrib | 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 |
| 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.


Figure 1-21. Infinity Format

## Freescale Semiconductor, Inc.

### 1.7.1.4 Not-A-Number

When created by the FPU, NANs represent the results of operations having no mathematical interpretation, such as infinity divided by infinity. Operations using a NAN operand as an input return a NAN result. User-created NANs can protect against uninitialized variables and arrays or can represent user-defined data types. See Figure 1-22.


Figure 1-22. Not-a-Number Format
If an input operand to an operation is a NAN, the result is an FPU-created default NAN. When the FPU creates a NAN, the NAN always contains the same bit pattern in the mantissa: all mantissa bits are ones and the sign bit is zero. When the user creates a NAN, any nonzero bit pattern can be stored in the mantissa and the sign bit.

### 1.7.1.5 Denormalized Numbers

Denormalized numbers represent real values near the underflow threshold. Denormalized numbers can be positive or negative. For denormalized numbers in single- and double-precision, the implied integer bit is a zero. See Figure 1-23.


Figure 1-23. Denormalized Number Format
Traditionally, the detection of underflow causes floating-point number systems to perform a flush-to-zero. The IEEE-754 standard implements gradual underflow: the result mantissa is shifted right (denormalized) while the result exponent is incremented until reaching the minimum value. If all the mantissa bits of the result are shifted off to the right during this denormalization, the result becomes zero.

Denormalized numbers are not supported directly in the hardware of this implementation but can be handled in software if needed (software for the input denorm exception could be written to handle denormalized input operands, and software for the underflow exception could create denormalized numbers). If the input denorm exception is disabled, all denormalized numbers are treated as zeros.

### 1.7.2 FPU Data Format and Type Summary

Table 1-11 summarizes the data type specifications for byte, word, longword, single-, and double-precision data formats.

## Freescale Semiconductor, Inc.

Table 1-11. Real Format Summary


| Signed Zeros |  |  |
| :--- | :---: | :---: |
| Biased exponent format minimum | $0(0 \times 00)$ | $0(0 \times 00)$ |
| Mantissa | $0 . f=0.0$ | $0 . f=0.0$ |
| Signed Infinities |  |  |
| Biased exponent format maximum | $255(0 x F F)$ | $2047(0 x 7 F F)$ |
| Mantissa | $0 . f=0.0$ | $0 . f=0.0$ |
|  | NANs | 0 or 1 |
| Sign | Don't care | $255(0 \times 7 F F)$ |
| Biased exponent format maximum | $255(0 x F F)$ | Nonzero |
| Fraction | Nonzero |  |

Freescale Semiconductor, Inc.

Table 1-11. Real Format Summary (Continued)

| Parameter | Single-Precision | Double-Precision |
| :--- | :---: | :---: |
| Representation of fraction <br> Nonzero bit pattern created by user <br> Fraction when created by FPU | xxxxx...xxxx <br> $11111 \ldots 1111$ | xxxxx...xxxx <br> $11111 \ldots 1111$ |
| Approximate Ranges |  |  |
| Maximum positive normalized | $3.4 \times 10^{38}$ | $1.8 \times 10^{308}$ |
| Minimum positive normalized | $1.2 \times 10^{-38}$ | $2.2 \times 10^{-308}$ |
| Minimum positive denormalized | $1.4 \times 10^{-45}$ | $4.9 \times 10^{-324}$ |

### 1.8 Multiply Accumulate Data Formats

The MAC and EMAC units support 16- or 32-bit input operands of the following formats:

- Two's complement signed integers: In this format, an N -bit operand value lies in the range $-2^{(N-1)} \leq$ operand $\leq 2^{(N-1)}-1$. The binary point is right of the lsb.
- Unsigned integers: In this format, an N -bit operand value lies in the range $0 \leq$ operand $\leq 2^{\mathrm{N}}-1$. The binary point is right of the lsb.
- Two's complement, signed fractionals: In an N-bit number, the first bit is the sign bit. The remaining bits signify the first $\mathrm{N}-1$ bits after the binary point. Given an N -bit number, $a_{N-1} a_{N-2} a_{N-3 \ldots} a_{2} a_{1} a_{0}$, its value is given by the equation in Figure 1-24.

$$
\text { value }=-\left(1 \cdot a_{N-1}\right)+\sum_{i=0}^{N-2} 2^{(i+1-N)} \cdot a i
$$

Figure 1-24. Two's Complement, Signed Fractional Equation
This format can represent numbers in the range $-1 \leq$ operand $\leq 1-2^{(N-1)}$.
For words and longwords, the largest negative number that can be represented is -1 , whose internal representation is $0 x 8000$ and $0 \times 8000 \_0000$, respectively. The largest positive word is $0 \times 7 \mathrm{FFF}$ or $\left(1-2^{-15}\right)$; the most positive longword is $0 \times 7 \mathrm{FFF}$ _FFFF or $\left(1-2^{-31}\right)$.

### 1.9 Organization of Data in Registers

This section describes data organization within the data, address, and control registers.

### 1.9.1 Organization of Integer Data Formats in Registers

Each integer data register is 32 bits wide. Byte and word operands occupy the lower 8- and 16-bit portions of integer data registers, respectively. Longword operands occupy entire data registers. A data register that is either a source or destination operand only uses or

## Freescale Semiconductor, Inc. <br> Organization of Data in Registers

changes the appropriate lower 8 or 16 bits (in byte or word operations, respectively). The remaining high-order portion does not change and is unused and unchanged. The address of the least significant bit (lsb) of a longword integer is zero, and the most significant bit $(\mathrm{msb})$ is 31 . Figure 1-25 illustrates the organization of integer data in data registers.


Figure 1-25. Organization of Integer Data Format in Data Registers
Because address registers and stack pointers are 32 bits wide, address registers cannot be used for byte-size operands. When an address register is a source operand, either the low-order word or the entire longword operand is used, depending on the operation size. When an address register is the destination operand, the entire register becomes affected, despite the operation size. If the source operand is a word size, it is sign-extended to 32 bits and then used in the operation to an address-register destination. Address registers are primarily for addresses and address computation support. The instruction set explains how to add to, compare, and move the contents of address registers. Figure 1-26 illustrates the organization of addresses in address registers.


Figure 1-26. Organization of Addresses in Address Registers

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

All operations to the SR and CCR are word-size operations. For all CCR operations, the upper byte is read as all zeros and is ignored when written, despite privilege mode. The write-only MOVEC instruction writes to the system control registers (VBR, CACR, etc.).

## Freescale Semiconductor, Inc.

### 1.9.2 Organization of Integer Data Formats in Memory

The byte-addressable organization of memory allows lower addresses to correspond to higher order bytes. The address N of a longword data item corresponds to the address of the MSB of the highest order word. The lower order word is located at address $\mathrm{N}+2$, leaving the LSB at address $\mathrm{N}+3$ (see Figure 1-27). The lowest address (nearest 0x00000000) is the location of the MSB, with each successive LSB located at the next address $(\mathrm{N}+1, \mathrm{~N}+2$, etc.). The highest address (nearest 0xFFFFFFFF) is the location of the LSB.


Figure 1-27. Memory Operand Addressing
Figure 1-28 illustrates the organization of data formats in memory. A base address that selects one byte in memory-the base byte-specifies a bit number that selects one bit, the bit operand, in the base byte. The msb of the byte is 7 .


Figure 1-28. Memory Organization for Integer Operands

## 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.

| Operation Word <br> (One Word, Specifies Operation and Modes) |
| :---: |
| Extension Word (If Any) |
| Extension Word (If Any) |

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

## Freescale Semiconductor, Inc.

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) |
| Exions  <br> D/A Index register type <br>  $0=$ Dn <br>  $1=$ An <br> W/L Word/longword index size <br>  $0=$ Address Error Exception <br>  $1=$ Longword <br> Scale Scale factor <br>  $00=1$ <br>  $01=2$ <br>  $10=4$ <br>  $11=8$ (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.


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 $=(\mathrm{An})$ |
| :--- | :--- |
| 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.


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 Assembler Syntax EA Mode Field EA Register Field Number of Extension Words
$E A=(A n)-$ Size; An = An - Size;
-(An)
100
Register number
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 $=(A n)+d_{16}$ |
| :--- | :--- |
| Assembler Syntax | $\left(d_{16}, A n\right)$ |
| 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
Assembler Syntax
EA Mode Field
EA Register Field
Number of Extension Words

```
EA = (An) + ((Xi) * ScaleFactor)) + Sign-extended d
(d
110
Register number
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 is the address of the extension word ( $\mathrm{PC}+2$ ). This is a program reference allowed only for reads.


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 is the address of the extension word $(\mathrm{PC}+2)$. 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 $=(\mathrm{PC})+\left((\mathrm{Xi}){ }^{*}\right.$ ScaleFactor $\left.)\right)+$ Sign-extended $\mathrm{d}_{8}$ |
| :--- | :--- |
| Assembler Syntax | $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}, \mathrm{Size}^{*}\right.$ 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 <br> low-order word is in the second extension word. |

Generation
Assembler Syntax
EA Mode Field
EA Register Field
Number of Extension Words

Operand given
\#<xxx>
111
100
1 or 2

Figure 2-14. Immediate Data Addressing

MOTOROLA

## Freescale Semiconductor, Inc.

### 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. <br> Field | Data | Memory | Control | Alterable |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Register Direct Data Address | $\begin{aligned} & \text { Dn } \\ & \text { An } \end{aligned}$ | $\begin{aligned} & 000 \\ & 001 \end{aligned}$ | reg. no. reg. no. | X | - | - | $\begin{aligned} & X \\ & X \end{aligned}$ |
| Register Indirect <br> Address <br> Address with Postincrement Address with Predecrement Address with Displacement | $\begin{gathered} (\mathrm{An}) \\ (\mathrm{An})+ \\ -(\mathrm{An}) \\ \left(\mathrm{d}_{16}, \mathrm{An}\right) \end{gathered}$ | $\begin{aligned} & 010 \\ & 011 \\ & 100 \\ & 101 \end{aligned}$ | reg. no. reg. no. reg. no. reg. no. | $\begin{aligned} & X \\ & X \\ & X \\ & X \end{aligned}$ | $\begin{aligned} & X \\ & X \\ & X \\ & X \\ & X \end{aligned}$ | $\frac{x}{\bar{x}}$ | $\begin{aligned} & X \\ & X \\ & X \\ & X \\ & X \end{aligned}$ |
| Address Register Indirect with Scaled Index and 8-Bit Displacement | $\left(\mathrm{d}_{8}, \mathrm{An}, \mathrm{Xi}^{*} \mathrm{SF}\right)$ | 110 | reg. no. | X | X | X | X |
| Program Counter Indirect with Displacement | $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 | X | X | X | - |
| Program Counter Indirect with Scaled Index and 8-Bit Displacement | ( $\left.\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}^{*} \mathrm{SF}\right)$ | 111 | 011 | X | X | X | - |
| Absolute Data Addressing Short Long | $\begin{gathered} (x x x) . W \\ (x x x) . \mathrm{L} \end{gathered}$ | $\begin{aligned} & 111 \\ & 111 \end{aligned}$ | $\begin{aligned} & 000 \\ & 001 \end{aligned}$ | $\begin{aligned} & X \\ & X \end{aligned}$ | $\begin{aligned} & X \\ & X \end{aligned}$ | $\begin{aligned} & X \\ & X \end{aligned}$ | - |
| 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 $-(\mathrm{An})$ 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

## Freescale Semiconductor, Inc.

## Chapter 3 Instruction Set Summary

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

### 3.1 Instruction Summary

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

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

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.

## Freescale Semiconductor, Inc.

Instruction Summary
Table 3-1. Notational Conventions

| Single- And Double Operand Operations |  |
| :---: | :---: |
| + | Arithmetic addition or postincrement indicator |
| - | Arithmetic subtraction or predecrement indicator |
| * | Arithmetic multiplication |
| 1 | Arithmetic division |
| ~ | Invert; operand is logically complemented. |
| \& | Logical AND |
| \| | Logical OR |
| $\wedge$ | Logical exclusive OR |
| $\rightarrow$ | Source operand is moved to destination operand. |
| $\leftarrow \rightarrow$ | 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 |  |
| An | Any address register $n$ (example: A3 is address register 3) |
| Ax, Ay | Destination and source address registers, respectively |
| Dn | 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 |
| $\mathrm{R} n$ | 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. |
| $\mathrm{d}_{\mathrm{n}}$ | Displacement value, $n$ bits wide (example: $\mathrm{d}_{16}$ is a 16-bit displacement). |
| sz | Size of operation: Byte (B), Word (W), Longword (L) |
| Isb, msb | Least significant bit, most significant bit |
| LSW, MSW | Least significant word, most significant word |
| SF | Scale factor for an index register |

Freescale Semiconductor, Inc.

Table 3-1. Notational Conventions (Continued)

| 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 |  |
| * | 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 |

Freescale Semiconductor, Inc.

Table 3-1. Notational Conventions (Continued)

| Floating-Point Operations |  |
| :---: | :--- |
| fmt | Format of operation: Byte (B), Word (W), Longword (L), Single-precision (S), <br> 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.

# Freescale Semiconductor, Inc. 

Table 3-2. Data Movement Operation Format

| Instruction | Operand Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :---: |
| FDMOVE | FPy,FPx | D | Source $\rightarrow$ Destination; round destination to double |
| FMOVE | $\begin{gathered} <e a>y, F P x \\ \text { FPy, }<e a>x \\ \text { FPy,FPx } \\ \text { FPcr, }<e a>x \\ <e a>y, F P c r \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \text { B,W,L,S,D } \\ \text { D } \\ \text { L } \\ \text { L } \end{gathered}$ | Source $\rightarrow$ Destination <br> FPcr can be any floating-point control register: FPCR, FPIAR, FPSR |
| FMOVEM | \#list,<ea>x <ea>y,\#list | D | Listed registers $\rightarrow$ Destination Source $\rightarrow$ Listed registers |
| FSMOVE | <ea>y,FPx | B,W,L,S,D | Source $\rightarrow$ Destination; round destination to single |
| LEA | <ea>y,Ax | L | <ea>y $\rightarrow$ Ax |
| LINK | Ay,\#<displacement> | W | SP - $4 \rightarrow$ SP; Ay $\rightarrow$ (SP); SP $\rightarrow$ Ay, SP + $\mathrm{d}_{\mathrm{n}} \rightarrow$ SP |
| MOV3Q ${ }^{1}$ | \#<data>,<ea>x | L | Immediate Data $\rightarrow$ Destination |
| MOVCLR ${ }^{2}$ | ACCy,Rx | L | Accumulator $\rightarrow$ Destination, $0 \rightarrow$ Accumulator |
| $\mathrm{MOVE}^{3}$ <br> MOVE from CCR MOVE to CCR | $\begin{gathered} <e a>y,<e a>x \\ \text { MACcr,Dx } \\ <e a>y, M A C c r \\ \text { CCR,Dx } \\ \text { <ea>y,CCR } \end{gathered}$ | $\begin{gathered} B, W, L \\ L \\ L \\ W \\ W \end{gathered}$ | Source $\rightarrow$ Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK |
| MOVEA | <ea>y,Ax | W, L $\rightarrow$ L | Source $\rightarrow$ Destination |
| MOVEM | \#list,<ea>x <ea>y,\#list | L | Listed Registers $\rightarrow$ Destination Source $\rightarrow$ Listed Registers |
| MOVEQ | \#<data>,Dx | $B \rightarrow L$ | Immediate Data $\rightarrow$ Destination |
| MVS ${ }^{1}$ | <ea>y,Dx | B,W | Source with sign extension $\rightarrow$ Destination |
| MVZ ${ }^{1}$ | <ea>y,Dx | B,W | Source with zero fill $\rightarrow$ Destination |
| PEA | <ea>y | L | SP - $4 \rightarrow$ SP; <ea>y $\rightarrow$ (SP) |
| UNLK | Ax | none | $A x \rightarrow S P ;(S P) \rightarrow A x ; S P+4 \rightarrow S P$ |
| 1 Supported starting with V4 <br> 2 EMAC instruction <br> 3 Additional addressing modes supported starting with V4 |  |  |  |

### 3.1.2 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)


## Freescale Semiconductor, Inc.

## Instruction Summary

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 3-3 for a summary of the integer arithmetic operations. In Table 3-3, X refers to the X-bit in the CCR.

Table 3-3. Integer Arithmetic Operation Format

| Instruction | Operand Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :---: |
| $\begin{aligned} & \text { ADD } \\ & \text { ADDA } \end{aligned}$ | Dy,<ea>x $<e a>y, D x$ <ea>y,Ax | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \\ & \mathrm{~L} \end{aligned}$ | Source + Destination $\rightarrow$ Destination |
| $\begin{gathered} \text { ADDI } \\ \text { ADDQ } \end{gathered}$ | $\begin{gathered} \text { \#<data>,Dx } \\ \text { \#<data>,<ea>x } \end{gathered}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | Immediate Data + Destination $\rightarrow$ Destination |
| ADDX | Dy, Dx | L | Source + Destination + CCR $[\mathrm{X}] \rightarrow$ Destination |
| CLR | <ea>x | B, W, L | $0 \rightarrow$ Destination |
| CMP CMPA | $\begin{aligned} & <e a>y, D x \\ & <e a>y, A x \end{aligned}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}^{1} \\ \mathrm{~W}, \mathrm{~L}^{2} \end{gathered}$ | Destination - Source $\rightarrow$ CCR |
| CMPI | \#<data>,Dx | B, W, L ${ }^{1}$ | Destination - Immediate Data $\rightarrow$ CCR |
| DIVS/DIVU ${ }^{3}$ | <ea>y,Dx | W, L | Destination / Source $\rightarrow$ Destination (Signed or Unsigned) |
| $\begin{aligned} & \text { EXT } \\ & \text { EXTB } \end{aligned}$ | $\begin{aligned} & D x \\ & D x \\ & D x \end{aligned}$ | $\begin{aligned} & B \rightarrow W \\ & W \rightarrow L \\ & B \rightarrow L \end{aligned}$ | Sign-Extended Destination $\rightarrow$ Destination |
| MAC | $\begin{gathered} R y, R x S F, A C C x^{4} \\ R y, R x S F,<e a>y, R w, A C C x^{4} \end{gathered}$ | $\begin{aligned} & \text { W, L } \\ & \text { W, L } \end{aligned}$ | $\begin{aligned} & \text { ACCx }+(\text { Ry * Rx })\{\ll \mid \gg\} S F \rightarrow \text { ACCx } \\ & \text { ACCx }+(\text { Ry * Rx) }\{\ll \mid \gg\} S F \rightarrow \text { ACCx; } \\ & (<e a>y(\& M A S K)) \rightarrow R w \end{aligned}$ |
| MSAC | $\begin{gathered} R y, R x S F, A C C x^{4} \\ R y, R x S F,<e a>y, R w, A C C x^{4} \end{gathered}$ | $\begin{aligned} & \text { W, L } \\ & \text { W, L } \end{aligned}$ | $\begin{aligned} & \text { ACCx }-\left(\mathrm{Ry}^{*} \mathrm{Rx}\right)\{\ll \mid \gg\} S F \rightarrow \text { ACCx } \\ & \text { ACCx - (Ry * Rx) }\{\ll \mid \gg\} S F \rightarrow \mathrm{ACCx} ; \\ & (<\mathrm{ea}>\mathrm{y}(\& \mathrm{MASK})) \rightarrow \mathrm{Rw} \end{aligned}$ |
| MULS/MULU | <ea>y,Dx | $\begin{gathered} W^{*} \mathrm{~W} \rightarrow \mathrm{~L} \\ L^{*} \mathrm{~L} \rightarrow \mathrm{~L} \end{gathered}$ | Source * Destination $\rightarrow$ Destination (Signed or Unsigned) |
| NEG | Dx | L | 0 - Destination $\rightarrow$ Destination |
| NEGX | Dx | L | 0 - Destination - CCR $[\mathrm{X}] \rightarrow$ Destination |
| REMS/REMU ${ }^{3}$ | <ea>y,Dw:Dx | L | Destination / Source $\rightarrow$ Remainder (Signed or Unsigned) |
| SATS ${ }^{5}$ | 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``` |
| SUB <br> SUBA | <ea>y,Dx Dy,<ea>x $<e a>y, A x$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \\ & \mathrm{~L} \end{aligned}$ | Destination - Source $\rightarrow$ Destination |

# Freescale Semiconductor, Inc. 

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

| SUBI <br> SUBQ | \#<data>,Dx <br> \#<data>,<ea $>x$ | L <br> L | Destination - Immediate Data $\rightarrow$ Destination |
| :---: | :---: | :---: | :--- |
| SUBX | Dy,Dx | L | Destination - Source - CCR $[\mathrm{X}] \rightarrow$ Destination |

1 Byte and word supported starting with V4
2 Word supported starting with V4
3 Supported starting with the 5206e
4 The accumulator does not need to be specified on the original MAC unit
5 Supported starting with V4

### 3.1.3 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 3-4 summarizes the logical operations.

Table 3-4. Logical Operation Format

| Instruction | Operand Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :--- |
| AND | $<e a>y, D x$ <br> Dy, <ea $>x$ | L | Source \& Destination $\rightarrow$ Destination |
| ANDI | \#<data>, Dx | L | Immediate Data \& Destination $\rightarrow$ Destination |
| EOR | Dy,<ea $>x$ | L | Source $^{\wedge}$ Destination $\rightarrow$ Destination |
| EORI | \#<data>,Dx | L | Immediate Data ^ Destination $\rightarrow$ Destination |
| NOT | Dx | L | $\sim$ Destination $\rightarrow$ Destination |
| OR | $<e a>y, D x$ | L | Source \| Destination $\rightarrow$ Destination |
| ORI | \#<data>,Dx | L | Immediate Data \| Destination $\rightarrow$ Destination |

### 3.1.4 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 3-5 is a summary of the shift operations. In Table 3-5, C and X refer to the C-bit and X-bit in the CCR.

Table 3-5. Shift Operation Format

| Instruction | Operand <br> Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :--- |
| ASL | Dy,Dx <br> \#<data $>, D x$ | L | $\mathrm{CCR}[\mathrm{X}, \mathrm{C}] \leftarrow(\mathrm{Dx} \ll \mathrm{Dy}) \leftarrow 0$ |
| $\mathrm{CCR}[\mathrm{X}, \mathrm{C}] \leftarrow(\mathrm{Dx} \ll$ \#<data $>) \leftarrow 0$ |  |  |  |

### 3.1.5 Bit Manipulation Instructions

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

Table 3-6. Bit Manipulation Operation Format

| Instruction | Operand Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :--- |
| BCHG | Dy, $<$ ea $>x$ | B, L | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z] |
|  | \#<data $>,<e a>x$ | B, L | $\rightarrow<$ bit number> of Destination |
| BCLR | Dy,<ea $>x$ | B, L | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z]; |
|  | \#<data $>,<e a>x$ | B, L | $0 \rightarrow<$ bit number> of Destination |
| BSET | Dy, $<$ ea $>x$ | B, L | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z]; |
|  | \#<data $>,<e a>x$ | B, L | $1 \rightarrow<$ bit number> of Destination |
| BTST | Dy,<ea $>x$ | B, L | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z] |
|  | \#<data $>,<e a>x$ | B, L |  |

### 3.1.6 Program Control Instructions

A set of subroutine call-and-return instructions and conditional and unconditional branch instructions perform program control operations. Also included are test operand instructions (TST 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 3-7 summarizes these instructions.

# Freescale Semiconductor, Inc. 

Table 3-7. Program Control Operation Format

| Instruction | Operand Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :---: |
| Conditional |  |  |  |
| Bcc | <label> | B, W, L ${ }^{1}$ | If Condition True, Then PC $+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ |
| FBcc | <label> | W, L | If Condition True, Then PC $+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ |
| Scc | Dx | B | If Condition True, Then 1s $\rightarrow$ Destination; Else 0s $\rightarrow$ Destination |
| Unconditional |  |  |  |
| BRA | <label> | B, W, L ${ }^{1}$ | $\mathrm{PC}+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ |
| BSR | <label> | B, W, L ${ }^{1}$ | SP - $4 \rightarrow$ SP; nextPC $\rightarrow$ (SP); PC + $\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ |
| FNOP | none | none | $\mathrm{PC}+2 \rightarrow \mathrm{PC}$ (FPU pipeline synchronized) |
| JMP | <ea>y | none | Source Address $\rightarrow$ PC |
| JSR | <ea>y | none | SP - $4 \rightarrow$ SP; nextPC $\rightarrow$ (SP); Source $\rightarrow$ PC |
| NOP | none | none | $\mathrm{PC}+2 \rightarrow \mathrm{PC}$ (Integer pipeline synchronized) |
| TPF | none <br> \#<data> <br> \#<data> | none W L | $\begin{aligned} & \mathrm{PC}+2 \rightarrow \mathrm{PC} \\ & \mathrm{PC}+4 \rightarrow \mathrm{PC} \\ & \mathrm{PC}+6 \rightarrow \mathrm{PC} \\ & \text { (Pipeline not synchronized) } \end{aligned}$ |
| Returns |  |  |  |
| RTS | none | none | $(\mathrm{SP}) \rightarrow \mathrm{PC} ; \mathrm{SP}+4 \rightarrow \mathrm{SP}$ |
| Test Operand |  |  |  |
| TAS ${ }^{2}$ | <ea>x | B | Destination Tested $\rightarrow$ CCR; $1 \rightarrow$ bit 7 of Destination |
| FTST | <ea>y | B, W, L, S, D | Source Operand Tested $\rightarrow$ FPCC |
| TST | <ea>y | B, W, L | Source Operand Tested $\rightarrow$ CCR |

1 Longword supported starting with V4
2 Supported starting with V4
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 | 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."

## Freescale Semiconductor, Inc.

### 3.1.7 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 3-8 summarizes these instructions.

Table 3-8. System Control Operation Format

| Instruction | Operand Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :---: |
| Privileged |  |  |  |
| FRESTORE | <ea>y | none | FPU State Frame $\rightarrow$ Internal FPU State |
| FSAVE | <ea>x | none | Internal FPU State $\rightarrow$ FPU State Frame |
| HALT | none | none | Halt processor core (synchronizes pipeline) |
| MOVE from SR | SR,Dx | W | SR $\rightarrow$ Destination |
| MOVE from USP ${ }^{1}$ | USP,Dx | L | USP $\rightarrow$ Destination |
| MOVE to SR | <ea>y,SR | W | Source $\rightarrow$ SR; Dy or \#<data> source only (synchronizes pipeline) |
| MOVE to USP ${ }^{1}$ | Ay,USP | L | Source $\rightarrow$ USP |
| MOVEC | Ry,Rc | L | $\mathrm{Ry} \rightarrow \mathrm{Rc}$ (synchronizes pipeline) |
| RTE | none | none | $2(\mathrm{SP}) \rightarrow \mathrm{SR} ; 4(\mathrm{SP}) \rightarrow \mathrm{PC} ; \mathrm{SP}+8 \rightarrow \mathrm{SP}$ <br> Adjust stack according to format (synchronizes pipeline) |
| STOP | \#<data> | none | Immediate Data $\rightarrow$ SR; STOP (synchronizes pipeline) |
| WDEBUG | <ea>y | L | Addressed Debug WDMREG Command Executed (synchronizes pipeline) |
| Debug Functions |  |  |  |
| PULSE | none | none | Set PST $=0 \times 4$ |
| WDDATA | <ea>y | B, W, L | Source $\rightarrow$ DDATA port |
| Trap Generating |  |  |  |
| ILLEGAL | none | none | $\begin{aligned} & S P-4 \rightarrow \text { SP; PC } \rightarrow(S P) \rightarrow P C ; S P-2 \rightarrow S P ; \\ & S R \rightarrow(S P) ; S P-2 \rightarrow \text { SP; Vector Offset } \rightarrow(S P) ; \\ & (V B R+0 \times 10) \rightarrow P C \end{aligned}$ |
| TRAP | \#<vector> | none | $\begin{aligned} & 1 \rightarrow \text { S Bit of SR; SP }-4 \rightarrow \text { SP; nextPC } \rightarrow(S P) ; \\ & S P-2 \rightarrow S P ; S R \rightarrow(S P) \\ & S P-2 \rightarrow \text { SP; Format/Offset } \rightarrow(S P) \\ & \left(V B R+0 \times 80+4^{*} n\right) \rightarrow P C \text {, where } n \text { is the TRAP number } \end{aligned}$ |

1 Supported starting with V4 on devices containing an MMU.

### 3.1.8 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 3-9 summarizes these instructions.

Table 3-9. Cache Maintenance Operation Format

| Instruction | Operand <br> Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :--- |
| CPUSHL | ic,(Ax) <br> dc,(Ax) <br> bc,(Ax) | none | If data is valid and modified, push cache line; invalidate <br> line if programmed in CACR (synchronizes pipeline) |
| INTOUCH $^{1}$ | Ay | none | Instruction fetch touch at (Ay) (synchronizes pipeline) |

1 Supported starting with V4

### 3.1.9 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 3-10 gives the general format for these dyadic instructions. Table 3-11 lists the available operations.

Table 3-10. Dyadic Floating-Point Operation Format

| Instruction | Operand <br> Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :---: |
| F<dop> | $<e a>y, F P x$ <br> FPy,FPx | B, W, L, S, D | FPx <Function $>$ Source $\rightarrow$ FPx |

Table 3-11. 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 3-12 gives the general format for these monadic instructions. Table 3-13 lists the available operations.

Table 3-12. Monadic Floating-Point Operation Format

| Instruction | Operand <br> Syntax | Operand Size | Operation |
| :---: | :---: | :---: | :---: |
| F<mop> | $<$ ea>y,FPx <br> FPy,FPx <br> FPx | B, W, L, S, D | Source $\rightarrow<$ Function $>\rightarrow$ FPx |
|  |  | FPx $\rightarrow<$ Function $>\rightarrow$ FPx |  |

## Table 3-13. Monadic Floating-Point Operations

| Instruction (F<mop>) | Operation |
| :--- | :--- |
| FABS, FSABS, FDABS | Absolute Value |
| FINT | Extract Integer Part |
| FINTRZ | Extract Integer Part, Rounded to Zero |
| FNEG, FSNEG, FDNEG | Negate |
| FSQRT, FSSQRT, FDSQRT | Square Root |

### 3.2 Instruction Set Additions

This section contains tables which summarize the baseline instruction set as well as the instructions that are added through ISA_B and the optional MAC, EMAC, and Floating-Point Units.

Table 3-14 shows the entire user instruction set in alphabetical order. Table 3-15 shows the entire supervisor instruction set in alphabetical order. In these tables, the ISA column has the following definitions:

- ISA_A: Part of the original ColdFire instruction set architecture
- ISA_B: Added with V4. ISA_B also contains all ISA_A instructions.
- MAC: Part of the original ColdFire MAC instruction set
- EMAC: Additional MAC instructions included in the EMAC
- FPU: Floating-Point Unit instructions

Table 3-14. ColdFire User Instruction Set Summary

| Instruction | Operand Syntax | Operand <br> Size | Operation | ISA |
| :---: | :---: | :---: | :--- | :---: |
| ADD | Dy, <ea>x <br> $<e a>y, D x$ <br> $<e a>y, A x$ | L <br> ADDA | Source + Destination $\rightarrow$ Destination | ISA_A |
| ADDI | \#<data>,Dx <br> \#<data>,<ea>x | L | Immediate Data + Destination $\rightarrow$ Destination | ISA_A |
| ADDQ | Dy,Dx | L | Source + Destination + CCR[X] $\rightarrow$ Destination | ISA_A |
| ADDX | $<e a>y, D x$ <br> Dy, $<e a>x$ | L | Source \& Destination $\rightarrow$ Destination | ISA_A |
| AND | \#<data>, Dx | L | Immediate Data \& Destination $\rightarrow$ Destination | ISA_A |
| ANDI |  |  |  |  |

## Freescale Semiconductor, Inc.

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

| Instruction | Operand Syntax | Operand Size | Operation | ISA |
| :---: | :---: | :---: | :---: | :---: |
| ASL | $\begin{gathered} \text { Dy,Dx } \\ \text { \#<data>,Dx } \end{gathered}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | $\begin{aligned} & \text { CCR }[\mathrm{X}, \mathrm{C}] \leftarrow(\mathrm{Dx} \ll \mathrm{Dy}) \leftarrow 0 \\ & \mathrm{CCR}[\mathrm{X}, \mathrm{C}] \leftarrow(\mathrm{Dx} \ll \text { \#<data }) \leftarrow 0 \end{aligned}$ | ISA_A |
| ASR | $\begin{gathered} \mathrm{Dy}, \mathrm{Dx} \\ \text { \#<data>,Dx } \end{gathered}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | $\begin{aligned} & \mathrm{msb} \rightarrow(\mathrm{Dx} \gg \mathrm{Dy}) \rightarrow \mathrm{CCR}[\mathrm{X}, \mathrm{C}] \\ & \mathrm{msb} \rightarrow(\mathrm{Dx} \gg \text { \#<data>}) \rightarrow \mathrm{CCR}[\mathrm{X}, \mathrm{C} \end{aligned}$ | ISA_A |
| Bcc | <label> | B, W, L ${ }^{1}$ | If Condition True, Then PC $+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ | ISA_A |
| BCHG | $\begin{gathered} \text { Dy, }<e a>x \\ \text { \#<data>, }<\text { ea }>x \end{gathered}$ | $\begin{aligned} & \mathrm{B}, \mathrm{~L} \\ & \mathrm{~B}, \mathrm{~L} \end{aligned}$ | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z] $\rightarrow$ <bit number> of Destination | ISA_A |
| BCLR | $\begin{gathered} \text { Dy, }<e a>x \\ \text { \#<data>,<ea>x } \end{gathered}$ | $\begin{aligned} & \mathrm{B}, \mathrm{~L} \\ & \mathrm{~B}, \mathrm{~L} \end{aligned}$ | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z]; $0 \rightarrow<$ bit number $>$ of Destination | ISA_A |
| BRA | <label> | B, W, L ${ }^{1}$ | $\mathrm{PC}+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ | ISA_A |
| BSET | $\begin{gathered} \text { Dy, }<e a>x \\ \text { \#<data>, }<\text { ea }>x \end{gathered}$ | $\begin{aligned} & \mathrm{B}, \mathrm{~L} \\ & \mathrm{~B}, \mathrm{~L} \end{aligned}$ | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z]; <br> $1 \rightarrow$ <bit number> of Destination | ISA_A |
| BSR | <label> | B, W, L ${ }^{1}$ | SP - $4 \rightarrow$ SP; nextPC $\rightarrow$ (SP); PC $+d_{n} \rightarrow$ PC | ISA_A |
| BTST | $\begin{gathered} \text { Dy,<ea>x } \\ \text { \#<data>,,<ea>x } \end{gathered}$ | $\begin{aligned} & \mathrm{B}, \mathrm{~L} \\ & \mathrm{~B}, \mathrm{~L} \end{aligned}$ | $\sim$ (<bit number> of Destination) $\rightarrow$ CCR[Z] | ISA_A |
| CLR | <ea>x | B, W, L | $0 \rightarrow$ Destination | ISA_A |
| $\begin{aligned} & \text { CMP } \\ & \text { CMPA } \end{aligned}$ | $\begin{aligned} & <e a>y, D x \\ & <e a>y, A x \end{aligned}$ | $\begin{gathered} B, W, L^{2} \\ W, L^{3} \end{gathered}$ | Destination - Source $\rightarrow$ CCR | ISA_A |
| CMPI | \#<data>,Dx | B, W, L ${ }^{1}$ | Destination - Immediate Data $\rightarrow$ CCR | ISA_A |
| DIVS/DIVU ${ }^{4}$ | <ea>y,Dx | W, L | Destination / Source $\rightarrow$ Destination (Signed or Unsigned) | ISA_A |
| EOR | Dy, <ea>x | L | Source ^ Destination $\rightarrow$ Destination | ISA_A |
| EORI | \#<data>,Dx | L | Immediate Data ^ Destination $\rightarrow$ Destination | ISA_A |
| EXT <br> EXTB | $\begin{aligned} & \mathrm{Dx} \\ & \mathrm{Dx} \\ & \mathrm{Dx} \end{aligned}$ | $\begin{aligned} & B \rightarrow W \\ & W \rightarrow L \\ & B \rightarrow L \end{aligned}$ | Sign-Extended Destination $\rightarrow$ Destination | ISA_A |
| FABS | $\begin{gathered} <e a>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Absolute Value of Source $\rightarrow$ FPx <br> Absolute Value of $\mathrm{FPx} \rightarrow \mathrm{FPx}$ | FPU |
| FADD | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source + FPx $\rightarrow$ FPx | FPU |
| FBcc | <label> | W, L | If Condition True, Then PC $+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ | FPU |
| FCMP | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx - Source | FPU |
| FDABS | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Absolute Value of Source $\rightarrow \mathrm{FPx}$; round destination to double <br> Absolute Value of FPx $\rightarrow \mathrm{FPx}$; round destination to double | FPU |
| FDADD | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source + FPx $\rightarrow$ FPx; round destination to double | FPU |
| FDDIV | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx / Source $\rightarrow$ FPx; round destination to double | FPU |

## Freescale Semiconductor, Inc.

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

| Instruction | Operand Syntax | Operand Size | Operation | ISA |
| :---: | :---: | :---: | :---: | :---: |
| FDIV | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \mathrm{D} \end{gathered}$ | FPx / Source $\rightarrow$ FPx | FPU |
| FDMOVE | FPy,FPx | D | Source $\rightarrow$ Destination; round destination to double | FPU |
| FDMUL | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ D \end{gathered}$ | Source * FPx $\rightarrow$ FPx; round destination to double | FPU |
| FDNEG | $\begin{gathered} \text { <ea }>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | - (Source) $\rightarrow$ FPx; round destination to double <br> - (FPx) $\rightarrow$ FPx; round destination to double | FPU |
| FDSQRT | $\begin{gathered} \text { <ea }>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Square Root of Source $\rightarrow \mathrm{FPx}$; round destination to double <br> Square Root of FPx $\rightarrow$ FPx; round destination to double | FPU |
| FDSUB | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx - Source $\rightarrow$ FPx; round destination to double | FPU |
| FINT | $\begin{gathered} \text { <ea }>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Integer Part of Source $\rightarrow$ FPx Integer Part of FPx $\rightarrow$ FPx | FPU |
| FINTRZ | $\begin{gathered} <e a>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Integer Part of Source $\rightarrow \mathrm{FPx}$; round to zero Integer Part of FPx $\rightarrow \mathrm{FPx}$; round to zero | FPU |
| FMOVE | $\begin{gathered} <e a>y, \text { FPx } \\ \text { FPy, <ea>x } \\ \text { FPy,FPx } \\ \text { FPcr, }<e a>x \\ <e a>y, F P c r \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \text { B,W,L,S,D } \\ \text { D } \\ \text { L } \\ \text { L } \end{gathered}$ | Source $\rightarrow$ Destination <br> FPcr can be any floating-point control register: FPCR, FPIAR, FPSR | FPU |
| FMOVEM | \#list,<ea>X <ea>y,\#list | D | Listed registers $\rightarrow$ Destination Source $\rightarrow$ Listed registers | FPU |
| FMUL | $\begin{aligned} & \text { <ea>y,FPx } \\ & \text { FPy,FPx } \end{aligned}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source * FPx $\rightarrow$ FPx | FPU |
| FNEG | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | $\begin{aligned} & -(\text { Source }) \rightarrow F P x \\ & -(F P x) \rightarrow F P x \end{aligned}$ | FPU |
| FNOP | none | none | PC + $2 \rightarrow$ PC (FPU Pipeline Synchronized) | FPU |
| FSABS | $\begin{gathered} <e a>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Absolute Value of Source $\rightarrow \mathrm{FPx}$; round destination to single Absolute Value of FPx $\rightarrow$ FPx; round destination to single | FPU |
| FSADD | $\begin{aligned} & \text { <ea>y,FPx } \\ & \text { FPy,FPx } \end{aligned}$ | B,W,L,S,D | Source + FPx $\rightarrow$ FPx; round destination to single | FPU |
| FSDIV | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx / Source $\rightarrow$ FPx; round destination to single | FPU |
| FSMOVE | $<e a>y, F P x$ | B,W,L,S,D | Source $\rightarrow$ Destination; round destination to single | FPU |
| FSMUL | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \mathrm{D} \end{gathered}$ | Source * FPx $\rightarrow$ FPx; round destination to single | FPU |

## Freescale Semiconductor, Inc.

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

| Instruction | Operand Syntax | $\begin{aligned} & \text { Operand } \\ & \text { Size } \end{aligned}$ | Operation | ISA |
| :---: | :---: | :---: | :---: | :---: |
| FSNEG | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} B, W, L, S, D \\ D \\ D \end{gathered}$ | - (Source) $\rightarrow \mathrm{FPx}$; round destination to single <br> - (FPx) $\rightarrow$ FPx; round destination to single | FPU |
| FSQRT | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} B, W, L, S, D \\ D \\ D \end{gathered}$ | Square Root of Source $\rightarrow$ FPx Square Root of $\mathrm{FPx} \rightarrow \mathrm{FPx}$ | FPU |
| FSSQRT | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ D \\ D \end{gathered}$ | Square Root of Source $\rightarrow$ FPx; round destination to single <br> Square Root of $\mathrm{FPx} \rightarrow \mathrm{FPx}$; round destination to single | FPU |
| FSSUB | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \mathrm{D} \end{gathered}$ | FPx - Source $\rightarrow$ FPx; round destination to single | FPU |
| FSUB | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx - Source $\rightarrow$ FPx | FPU |
| FTST | <ea>y | B, W, L, S, D | Source Operand Tested $\rightarrow$ FPCC | FPU |
| ILLEGAL | none | none | $\begin{aligned} & \mathrm{SP}-4 \rightarrow \mathrm{SP} ; \mathrm{PC} \rightarrow(\mathrm{SP}) \rightarrow \mathrm{PC} ; \mathrm{SP}-2 \rightarrow \mathrm{SP} ; \\ & \mathrm{SR} \rightarrow(\mathrm{SP}) ; \mathrm{SP}-2 \rightarrow \mathrm{SP} ; \text { Vector Offset } \rightarrow(\mathrm{SP}) ; \\ & (\mathrm{VBR}+0 \times 10) \rightarrow \mathrm{PC} \end{aligned}$ | ISA_A |
| JMP | <ea>y | none | Source Address $\rightarrow$ PC | ISA_A |
| JSR | <ea>y | none | SP - $4 \rightarrow$ SP; nextPC $\rightarrow$ (SP); Source $\rightarrow$ PC | ISA_A |
| LEA | <ea>y,Ax | L | <ea>y $\rightarrow$ Ax | ISA_A |
| LINK | Ay,\#<displacement> | W | SP - 4 $\rightarrow$ SP; Ay $\rightarrow$ (SP); SP $\rightarrow$ Ay, SP + $\mathrm{d}_{\mathrm{n}} \rightarrow$ SP | ISA_A |
| LSL | $\begin{gathered} \mathrm{Dy}, \mathrm{Dx} \\ \text { \#<data>,Dx } \end{gathered}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | $\begin{aligned} & \text { CCR }[\mathrm{X}, \mathrm{C}] \leftarrow(\mathrm{Dx} \ll \text { Dy }) \leftarrow 0 \\ & \text { CCR }[\mathrm{X}, \mathrm{C}] \leftarrow(\text { Dx } \ll \text { \#<data }) \leftarrow 0 \end{aligned}$ | ISA_A |
| LSR | $\begin{gathered} \mathrm{Dy}, \mathrm{Dx} \\ \text { \#<data>,Dx } \end{gathered}$ | $\begin{aligned} & \hline \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | $\begin{aligned} & 0 \rightarrow(\text { Dx } \gg \text { Dy }) \rightarrow \text { CCR[X,C] } \\ & 0 \rightarrow(D x \gg \text { \#<data> }) \rightarrow \text { CCR[ } X, C] \end{aligned}$ | ISA_A |
| MAC | Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw, ACCx | $\begin{aligned} & \text { W, L } \\ & \text { W, L } \end{aligned}$ | $\begin{aligned} & \text { ACCx }+(\mathrm{Ry} \text { * } \mathrm{Rx})\{\ll \mid \gg\} S F \rightarrow \text { ACCx } \\ & \text { ACCx }+(\mathrm{Ry} \text { * } \mathrm{Rx})\{\ll \mid \gg\} S F \rightarrow \text { ACCx; } \\ & (<e \mathrm{e}>\mathrm{y}(\& M A S K)) \rightarrow \mathrm{Rw} \end{aligned}$ | MAC |
| MOV3Q | \#<data>,<ea>x | L | Immediate Data $\rightarrow$ Destination | ISA_B |
| MOVCLR | ACCy,Rx | L | Accumulator $\rightarrow$ Destination, $0 \rightarrow$ Accumulator | EMAC |
| MOVE <br> MOVE from CCR MOVE to CCR | $\begin{gathered} <e a>y,<e a>x \\ \text { MACcr,Dx } \\ <e a>y, M A C c r \\ \text { CCR,Dx } \\ \text { <ea>y,CCR } \end{gathered}$ | $\begin{gathered} \text { B,W,L } \\ L \\ L \\ W \\ W \end{gathered}$ | Source $\rightarrow$ Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK | ISA_A <br> MAC <br> MAC <br> ISA A <br> ISA_A |
| MOVEA | <ea>y,Ax | $\mathrm{W}, \mathrm{L} \rightarrow \mathrm{L}$ | Source $\rightarrow$ Destination | ISA_A |
| MOVEM | \#list,<ea>x <ea>y,\#list | L | Listed Registers $\rightarrow$ Destination Source $\rightarrow$ Listed Registers | ISA_A |
| MOVEQ | \#<data>,Dx | $B \rightarrow L$ | Immediate Data $\rightarrow$ Destination | ISA_A |
| MSAC | Ry,RxSF,ACCx Ry,RxSF,<ea>y,Rw, ACCx | $\begin{aligned} & \text { W, L } \\ & \text { W, L } \end{aligned}$ | $\begin{aligned} & \text { ACCx - (Ry * Rx) }\{\ll \mid \gg\} S F \rightarrow \text { ACCx } \\ & \text { ACCx }-(R y * R x)\{\ll \mid \gg\} S F \rightarrow \text { ACCx; } \\ & (<e a>y(\& M A S K)) \rightarrow \mathrm{Rw} \end{aligned}$ | MAC |

## Freescale Semiconductor, Inc.

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

| Instruction | Operand Syntax | Operand Size | Operation | ISA |
| :---: | :---: | :---: | :---: | :---: |
| MULS/MULU | <ea>y,Dx | $\begin{gathered} W * W \rightarrow L \\ L^{*} L \rightarrow L \end{gathered}$ | Source * Destination $\rightarrow$ Destination (Signed or Unsigned) | ISA_A |
| MVS | <ea>y, Dx | B,W | Source with sign extension $\rightarrow$ Destination | ISA_B |
| MVZ | <ea>y,Dx | B,W | Source with zero fill $\rightarrow$ Destination | ISA_B |
| NEG | Dx | L | 0 - Destination $\rightarrow$ Destination | ISA_A |
| NEGX | Dx | L | 0 - Destination - CCR $[\mathrm{X}] \rightarrow$ Destination | ISA_A |
| NOP | none | none | PC + $2 \rightarrow$ PC (Integer Pipeline Synchronized) | ISA_A |
| NOT | Dx | L | $\sim$ Destination $\rightarrow$ Destination | ISA_A |
| OR | $\begin{aligned} & <e a>y, D x \\ & D y,<e a>x \end{aligned}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | Source \| Destination $\rightarrow$ Destination | ISA_A |
| ORI | \#<data>,Dx | L | Immediate Data \| Destination $\rightarrow$ Destination | ISA_A |
| PEA | <ea>y | L | SP - $4 \rightarrow$ SP; <ea>y $\rightarrow$ (SP) | ISA_A |
| PULSE | none | none | Set PST $=0 \times 4$ | ISA_A |
| REMS/REMU ${ }^{4}$ | <ea>y,Dw:Dx | L | Destination / Source $\rightarrow$ Remainder (Signed or Unsigned) | ISA_A |
| RTS | none | none | $(\mathrm{SP}) \rightarrow \mathrm{PC} ; \mathrm{SP}+4 \rightarrow \mathrm{SP}$ | ISA_A |
| SATS | Dx | L | ```If CCR[V] == 1; then if Dx[31] == 0; then Dx[31:0] = 0x80000000; else Dx[31:0] = 0x7FFFFFFFF; else Dx[31:0] is unchanged``` | ISA_B |
| Scc | Dx | B | If Condition True, Then 1s $\rightarrow$ Destination; Else 0s $\rightarrow$ Destination | ISA_A |
| SUB <br> SUBA | $\begin{aligned} & <e a>y, D x \\ & D y,<e a>x \\ & <e a>y, A x \end{aligned}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \\ & \mathrm{~L} \end{aligned}$ | Destination - Source $\rightarrow$ Destination | ISA_A |
| $\begin{aligned} & \text { SUBI } \\ & \text { SUBQ } \end{aligned}$ | $\begin{gathered} \text { \#<data>,Dx } \\ \text { \#<data>,<ea>x } \end{gathered}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | Destination - Immediate Data $\rightarrow$ Destination | ISA_A |
| SUBX | Dy, Dx | L | Destination - Source - CCR[X] $\rightarrow$ Destination | ISA_A |
| SWAP | Dx | W | MSW of Dx $\leftrightarrow$ LSW of Dx | ISA_A |
| TAS | <ea>x | B | Destination Tested $\rightarrow$ CCR; $1 \rightarrow$ bit 7 of Destination | ISA_B |
| TPF | none <br> \#<data> <br> \#<data> | none <br> W <br> L | $\begin{aligned} & \mathrm{PC}+2 \rightarrow \mathrm{PC} \\ & \mathrm{PC}+4 \rightarrow \mathrm{PC} \\ & \mathrm{PC}+6 \rightarrow \mathrm{PC} \end{aligned}$ | ISA_A |
| TRAP | \#<vector> | none | $1 \rightarrow$ S Bit of SR; SP $-4 \rightarrow$ SP; nextPC $\rightarrow$ (SP); SP-2 $\rightarrow$ SP; SR $\rightarrow$ (SP) SP - $2 \rightarrow$ SP; Format/Offset $\rightarrow$ (SP) $\left(\mathrm{VBR}+0 \times 80+4^{*} \mathrm{n}\right) \rightarrow \mathrm{PC}$, where n is the TRAP number | ISA_A |
| TST | <ea>y | B, W, L | Source Operand Tested $\rightarrow$ CCR | ISA_A |

## Freescale Semiconductor, Inc.

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

| Instruction | Operand Syntax | Operand <br> Size | Operation | ISA |
| :---: | :---: | :---: | :--- | :---: |
| UNLK | Ax | none | $\mathrm{Ax} \rightarrow \mathrm{SP} ;(\mathrm{SP}) \rightarrow \mathrm{Ax} ; \mathrm{SP}+4 \rightarrow \mathrm{SP}$ | ISA_A |
| WDDATA | <ea $>y$ | $\mathrm{~B}, \mathrm{~W}, \mathrm{~L}$ | Source $\rightarrow$ DDATA port | ISA_A |

1 Longword supported starting with V4
2 Byte and word supported starting with V4
3 Word supported starting with V4
4 Supported starting with the 5206e
Table 3-15. ColdFire Supervisor Instruction Set Summary

| Instruction | Operand Syntax | Operand Size | Operation | ISA |
| :---: | :---: | :---: | :---: | :---: |
| 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 $\rightarrow$ Internal FPU State | FPU |
| FSAVE | <ea>x | none | Internal FPU State $\rightarrow$ FPU State Frame | 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 $\rightarrow$ Destination | ISA_A |
| MOVE from USP ${ }^{1}$ | USP,Dx | L | USP $\rightarrow$ Destination | ISA_B |
| MOVE to SR | <ea>y,SR | W | Source $\rightarrow$ SR; Dy or \#<data> source only | ISA_A |
| MOVE to USP ${ }^{1}$ | Ay,USP | L | Source $\rightarrow$ USP | ISA_B |
| MOVEC | Ry,Rc | L | Ry $\rightarrow$ Rc | ISA_A |
| RTE | none | none | $2(\mathrm{SP}) \rightarrow \mathrm{SR} ; 4(\mathrm{SP}) \rightarrow \mathrm{PC} ; \mathrm{SP}+8 \rightarrow \mathrm{SP}$ <br> Adjust stack according to format | ISA_A |
| STOP | \#<data> | none | Immediate Data $\rightarrow$ SR; STOP | ISA_A |
| WDEBUG | <ea>y | L | Addressed Debug WDMREG Command Executed | ISA_A |

1 Supported starting with V4 on devices containing an MMU.

## Freescale Semiconductor, Inc.

Table 3-16 summarizes the additional instructions for the ISA_B instruction set.
Table 3-16. ColdFire ISA_B Additions Summary

| Instruction | Operand Syntax | Operand Size | Operation | Super/ User |
| :---: | :---: | :---: | :---: | :---: |
| Bcc | <label> | B, W, L ${ }^{1}$ | If Condition True, Then PC $+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ | User |
| BRA | <label> | B, W, L ${ }^{1}$ | $\mathrm{PC}+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ | User |
| BSR | <label> | $B, W, L^{1}$ | SP - $4 \rightarrow$ SP; PC $\rightarrow$ (SP); PC + $\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ | User |
| $\begin{gathered} \text { CMP } \\ \text { CMPA } \end{gathered}$ | $\begin{aligned} & <e a>y, D x \\ & <e a>y, A x \end{aligned}$ | $\begin{gathered} B, W, L^{2} \\ W, L^{3} \end{gathered}$ | Destination - Source $\rightarrow$ cc | User |
| CMPI | \#<data>,Dx | B, W, L ${ }^{1}$ | Destination - Immediate Data $\rightarrow$ cc | User |
| CPUSHL | $\begin{aligned} & \mathrm{ic},(\mathrm{Ax}) \\ & \mathrm{dc},(\mathrm{Ax}) \\ & \mathrm{bc},(\mathrm{Ax}) \end{aligned}$ | none | If data is valid and modified, push cache line; invalidate line if programmed in CACR | Super |
| INTOUCH | Ax | none | Instruction fetch touch at (Ax) | Super |
| MOV3Q | \#<data>,<ea>x | L | Immediate Data $\rightarrow$ Destination | User |
| MOVE ${ }^{4}$ | <ea>y,<ea>x | B,W,L | Source $\rightarrow$ Destination | User |
| MOVE from USP ${ }^{5}$ | USP,Dx | L | USP $\rightarrow$ Destination | Super |
| MOVE to USP ${ }^{5}$ | Ay,USP | L | Source $\rightarrow$ USP | Super |
| MOVEA | <ea>y,Ax | W,L $\rightarrow$ L | Source $\rightarrow$ Destination | User |
| MVS | <ea>y,Dx | B,W | Source with sign extension $\rightarrow$ Destination | User |
| MVZ | <ea>y,Dx | B,W | Source with zero fill $\rightarrow$ Destination | User |
| SATS | Dx | L | ```If CCR[V] == 1; then if Dx[31] == 0; then Dx[31:0] = 0x80000000; else Dx[31:0] = 0x7FFFFFFFF; else Dx[31:0] is unchanged``` | User |
| TAS | <ea>x | B | Destination Tested $\rightarrow$ Condition Codes; $1 \rightarrow$ bit 7 of Destination | User |

1 Longword supported starting with V4
2 Byte and word supported starting with V4
3 Word supported starting with V4
4 Additional addressing modes supported starting with V4
5 Supported starting with V4 on devices containing an MMU.

## Freescale Semiconductor, Inc.

Table 3-17 summarizes the instruction set supported by the original MAC unit.
Table 3-17. MAC Instruction Set Summary

| Instruction | Operand Syntax | Operand Size | Operation | Super/ User |
| :---: | :---: | :---: | :---: | :---: |
| MAC | Ry,RxSF,ACC Ry,RxSF,<ea>y,Rw, ACC | W, L W, L | $\begin{aligned} & \text { ACC + (Ry * Rx) }\{\ll \mid \gg\} S F \rightarrow \text { ACC } \\ & \text { ACC + (Ry *Rx) }\{\ll \mid \gg\} S F \rightarrow A C C ; \\ & (<e a>y(\& M A S K)) \rightarrow R w \end{aligned}$ | User |
| MOVE | $\begin{gathered} \text { MACcr,Dx } \\ <e \mathrm{ea>y}, \mathrm{MACcr} \end{gathered}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | Source $\rightarrow$ Destination where MACcr can be any MAC control register: ACC, MACSR, MASK | User |
| MSAC | Ry,RxSF,ACC Ry,RxSF,<ea>y,Rw, ACC | $\begin{aligned} & \text { W, L } \\ & \text { W, L } \end{aligned}$ | $\begin{aligned} & \text { ACC }-\left(\mathrm{Ry}^{*} \mathrm{Rx}\right)\{\ll \mid \gg\} S F \rightarrow \mathrm{ACC} \\ & \text { ACC }-\left(\mathrm{Ry}{ }^{*} \mathrm{Rx}\right)\{\ll \mid \gg\} S F \rightarrow \mathrm{ACC} ; \\ & (<\mathrm{ea}>\mathrm{y}(\& \mathrm{MASK})) \rightarrow \mathrm{Rw} \end{aligned}$ | User |

Table 3-18 summarizes the changes to the instruction set due to the enhancements in the EMAC unit.

Table 3-18. EMAC Instruction Set Enhancements Summary

| Instruction | Operand Syntax | Operand Size | Operation | Super/ User |
| :---: | :---: | :---: | :---: | :---: |
| MAC ${ }^{1}$ | Ry,RxSF,ACCx $R y, R x S F,<e a>y, R w$, ACCx | $\begin{aligned} & \text { W, L } \\ & \text { W, L } \end{aligned}$ | $\begin{aligned} & \text { ACCx }+\left(\mathrm{Ry}{ }^{*} \mathrm{Rx}\right)\{\ll \mid \gg\} S F \rightarrow \text { ACCx } \\ & \text { ACCx }+(\mathrm{Ry} \text { * } \mathrm{Rx})\{\ll \mid \gg\} S F \rightarrow \mathrm{ACCx} ; \\ & (<e \mathrm{e}>\mathrm{y}(\& \mathrm{MASK})) \rightarrow \mathrm{Rw} \end{aligned}$ | User |
| MOVCLR | ACCy,Rx | L | Accumulator $\rightarrow$ Destination, $0 \rightarrow$ Accumulator | User |
| MOVE | $\begin{gathered} \text { MACcr,Dx } \\ \text { <ea>y,MACcr } \end{gathered}$ | $\begin{aligned} & \mathrm{L} \\ & \mathrm{~L} \end{aligned}$ | Source $\rightarrow$ Destination where MACcr can be any MAC control register: ACCx, ACCext01, ACCext23, MACSR, MASK | User |
| MSAC | Ry,RxSF,ACCx $R y, R x S F,<e a>y, R w$, ACCx | W, L W, L | $\begin{aligned} & \text { ACCx }-(\text { Ry * Rx })\{\ll \mid \gg\} S F \rightarrow \text { ACCx } \\ & \text { ACCx }-(R y \text { * } R x)\{\ll \mid \gg\} S F \rightarrow \text { ACCx; } \\ & (<e a>y(\& M A S K)) \rightarrow R w \end{aligned}$ | User |

1 The EMAC has 4 accumulators

## Freescale Semiconductor, Inc.

## Instruction Set Additions

Table 3-19 summarizes the instruction set supported by the floating-point unit.
Table 3-19. Floating-Point Instruction Set Summary

| Instruction | Operand Syntax | Operand Size | Operation | Super/ User |
| :---: | :---: | :---: | :---: | :---: |
| FABS | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Absolute Value of Source $\rightarrow$ FPx <br> Absolute Value of FPx $\rightarrow$ FPx | User |
| FADD | $\begin{aligned} & \text { <ea>y,FPx } \\ & \text { FPy,FPx } \end{aligned}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source + FPx $\rightarrow$ FPx | User |
| FBcc | <label> | W, L | If Condition True, Then PC $+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$ | User |
| FCMP | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \mathrm{D} \end{gathered}$ | FPx - Source | User |
| FDABS | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Absolute Value of Source $\rightarrow \mathrm{FPx}$; round destination to double <br> Absolute Value of $\mathrm{FPx} \rightarrow \mathrm{FPx}$; round destination to double | User |
| FDADD | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source + FPx $\rightarrow$ FPx; round destination to double | User |
| FDDIV | $\begin{aligned} & \text { <ea>y,FPx } \\ & \text { FPy,FPx } \end{aligned}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \mathrm{D} \end{gathered}$ | FPx / Source $\rightarrow$ FPx; round destination to double | User |
| FDIV | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx / Source $\rightarrow$ FPx | User |
| FDMOVE | FPy,FPx | D | Source $\rightarrow$ Destination; round destination to double | User |
| FDMUL | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source * FPx $\rightarrow$ FPx; round destination to double | User |
| FDNEG | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | - (Source) $\rightarrow$ FPx; round destination to double <br> - (FPx) $\rightarrow$ FPx; round destination to double | User |
| FDSQRT | $\begin{gathered} <e a>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Square Root of Source $\rightarrow \mathrm{FPx}$; round destination to double <br> Square Root of $\mathrm{FPx} \rightarrow \mathrm{FPx}$; round destination to double | User |
| FDSUB | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx - Source $\rightarrow$ FPx; round destination to double | User |
| FINT | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Integer Part of Source $\rightarrow$ FPx <br> Integer Part of FPx $\rightarrow$ FPx | User |
| FINTRZ | $\begin{gathered} <e a>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Integer Part of Source $\rightarrow$ FPx; round to zero Integer Part of FPx $\rightarrow$ FPx; round to zero | User |
| FMOVE | $\begin{gathered} <e a>y, F P x \\ \text { FPy, }<e a>x \\ \text { FPy,FPx } \\ \text { FPcr, }<e a>x \\ <e a>y, F P c r \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \text { B,W,L,S,D } \\ \text { D } \\ \text { L } \\ \text { L } \end{gathered}$ | Source $\rightarrow$ Destination <br> FPcr can be any floating-point control register: FPCR, FPIAR, FPSR | User |
| FMOVEM | \#list, <ea>x <ea>y,\#list | D | Listed registers $\rightarrow$ Destination Source $\rightarrow$ Listed registers | User |

## Freescale Semiconductor, Inc.

Table 3-19. Floating-Point Instruction Set Summary

| Instruction | Operand Syntax | Operand Size | Operation | Super/ User |
| :---: | :---: | :---: | :---: | :---: |
| FMUL | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source * FPx $\rightarrow$ FPx | User |
| FNEG | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \text { D } \\ \text { D } \end{gathered}$ | $\begin{aligned} & -(\text { Source }) \rightarrow F P x \\ & -(F P x) \rightarrow F P x \end{aligned}$ | User |
| FNOP | none | none | PC + $2 \rightarrow$ PC (FPU Pipeline Synchronized) | User |
| FRESTORE | <ea>y | none | FPU State Frame $\rightarrow$ Internal FPU State | Super |
| FSABS | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ D \\ D \end{gathered}$ | Absolute Value of Source $\rightarrow \mathrm{FPx}$; round destination to single Absolute Value of FPx $\rightarrow$ FPx; round destination to single | User |
| FSADD | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source + FPx $\rightarrow$ FPx; round destination to single | User |
| FSAVE | <ea>x | none | Internal FPU State $\rightarrow$ FPU State Frame | Super |
| FSDIV | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx / Source $\rightarrow$ FPx; round destination to single | User |
| FSMOVE | <ea>y,FPx | B,W,L,S,D | Source $\rightarrow$ Destination; round destination to single | User |
| FSMUL | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | Source * FPx $\rightarrow$ FPx; round destination to single | User |
| FSNEG | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ D \\ D \end{gathered}$ | - (Source) $\rightarrow$ FPx; round destination to single <br> - (FPx) $\rightarrow$ FPx; round destination to single | User |
| FSQRT | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} \text { B,W,L,S,D } \\ \text { D } \\ \text { D } \end{gathered}$ | Square Root of Source $\rightarrow$ FPx <br> Square Root of $\mathrm{FPx} \rightarrow \mathrm{FPx}$ | User |
| FSSQRT | $\begin{gathered} <e a>y, \text { FPx } \\ \text { FPy,FPx } \\ \text { FPx } \end{gathered}$ | $\begin{gathered} B, W, L, S, D \\ D \\ D \end{gathered}$ | Square Root of Source $\rightarrow$ FPx; round destination to single <br> Square Root of $\mathrm{FPx} \rightarrow \mathrm{FPx}$; round destination to single | User |
| FSSUB | $\begin{gathered} \text { <ea>y,FPx } \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx - Source $\rightarrow$ FPx; round destination to single | User |
| FSUB | $\begin{gathered} <e a>y, F P x \\ \text { FPy,FPx } \end{gathered}$ | $\begin{gathered} \mathrm{B}, \mathrm{~W}, \mathrm{~L}, \mathrm{~S}, \mathrm{D} \\ \mathrm{D} \end{gathered}$ | FPx - Source $\rightarrow$ FPx | User |
| FTST | <ea>y | B, W, L, S, D | Source Operand Tested $\rightarrow$ FPCC | User |

## Freescale Semiconductor, Inc.

## 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. DIVS/U and REMS/U are supported starting with the 5206e. The original ColdFire Instruction Set Architecture, ISA_A, is supported by V2 and V3 cores. The V4 core supports ISA_B, which encompasses all of ISA_A, extends the functionality of some ISA_A instructions, and adds several new instructions. These extensions can be identified by a table which appears at the end of each instruction description where there are ISA_B differences.

# Freescale Semiconductor, Inc. 

ADD
Add
ADD
(All ColdFire Processors)
Operation: $\quad$ Source + Destination $\rightarrow$ Destination
Assembler Syntax: ADD.L <ea>y,Dx
ADD.L Dy, <ea>x
Attributes: $\quad$ 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.


Instruction
Format:


## Instruction Fields:

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

| Byte | Word | Longword | Operation |
| :---: | :---: | :---: | :---: |
| - | - | 010 | $<e a>y+D x \rightarrow D x$ |
| - | - | 110 | $\mathrm{Dy}+<\mathrm{e} a>x \rightarrow<\mathrm{ea}>\mathrm{x}$ |

## 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 |
| :---: | :---: | :---: |
| Dy | 000 | reg. number:Dy |
| Ay | 001 | reg. number:Ay |
| (Ay) | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(d_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | 111 | 011 |

- For the destination operand $\langle\mathrm{ea}>\mathrm{x}$, use addressing modes listed in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | - | - |
| $A x$ | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Operation: $\quad$ Source + Destination $\rightarrow$ Destination

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

## Attributes: $\quad$ 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 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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

Operation: $\quad$ Immediate Data + Destination $\rightarrow$ Destination
Assembler Syntax: ADDI.L \#<data>,Dx
Attributes: $\quad$ 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.


| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 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.

MOTOROLA

ADDQ
Add Quick
ADDQ
(All ColdFire Processors)
Operation: $\quad$ Immediate Data + Destination $\rightarrow$ Destination
Assembler Syntax: ADDQ.L \#<data>, <ea>x

## Attributes: $\quad$ 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.



## 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 |
| $A x$ | 001 | reg. number:Ax |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

## Add Extended

ADDX
(All ColdFire Processors)
Operation: $\quad$ Source + Destination + CCR $[\mathrm{X}] \rightarrow$ Destination

## Assembler Syntax: ADDX.L Dy,Dx

Attributes: $\quad$ 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 | X | N | Z | V | C | X | Set the same as the carry bit |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Codes: | * | * | * | * | * | N | Set if the result is negative; cleared otherwise |
|  |  |  |  |  |  | Z | Cleared if the result is non-zero; unchanged otherwise |
|  |  |  |  |  |  | $\checkmark$ | Set if an overflow is generated; cleared otherwise |
|  |  |  |  |  |  |  | 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 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 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.
motorola


## Freescale Semiconductor, Inc.

AND
AND Logical
(All ColdFire Processors)
Operation: $\quad$ Source \& Destination $\rightarrow$ Destination
Assembler Syntax: AND.L <ea>y,Dx
AND.L Dy, <ea>x
Attributes: $\quad$ 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.


| Instruction | 15 | 14 | 13 | 12 | 11 | 109 | 8 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 0 | 0 | Data Register |  | Opmode |  |  |  |  | ctiv |  |  |
|  |  |  |  |  |  |  |  | Mod |  |  |  |

## Instruction Fields:

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

| Byte | Word | Longword | Operation |
| :---: | :---: | :---: | :---: |
| - | - | 010 | $<e a>y \& D x \rightarrow D x$ |
| - | - | 110 | Dy \& <ea $>x \rightarrow<e a>x$ |

## Freescale Semiconductor, Inc.

AND Logical

Instruction Fields (continued):

- Effective Address field-Determines addressing mode.
- For the source operand $\langle e a\rangle \mathrm{y}$, use addressing modes listed in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | 000 | reg. number:Dy |
| Ay | - | - |
| (Ay) | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

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

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | - | - |
| Ax | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## AND Immediate

(All ColdFire Processors)
Operation: $\quad$ Immediate Data \& Destination $\rightarrow$ Destination
Assembler Syntax: ANDI.L \#<data>,Dx
Attributes: $\quad$ 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.


[^0]| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 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.


## Freescale Semiconductor, Inc.

Operation: $\quad$ Destination Shifted By Count $\rightarrow$ Destination

## Assembler Syntax: ASd.L Dy,Dx

ASd.L \#<data>,Dx

$$
\text { where } \mathrm{d} \text { is direction, } \mathrm{L} \text { or } \mathrm{R}
$$

Attributes: $\quad$ 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.

ASR:


## Freescale Semiconductor, Inc.

ASL, ASR


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 68 K family processors.

| Instruction | 15 | 14 | 13 | 12 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 0 | Count or <br> Register, $D y$ | dr | 1 | 0 | $\mathrm{i} / \mathrm{r}$ | 0 | 0 | Register, Dx |  |  |  |

## Instruction Fields:

- Count or Register field—Specifies shift count or register, Dy, that contains the shift count:
- If $\mathrm{i} / \mathrm{r}=0$, this field contains the shift count; values $1-7$ represent counts of 1 7 ; a value of zero represents a count of 8 .
- If $\mathrm{i} / \mathrm{r}=1$, this field specifies the data register, $D y$, that contains the shift count (modulo 64).
- dr field—specifies the direction of the shift:
- 0 shift right
- 1 shift left
- $\mathrm{i} / \mathrm{r}$ field
- If $\mathrm{i} / \mathrm{r}=0$, specifies immediate shift count
- If $\mathrm{i} / \mathrm{r}=1$, specifies register shift count
- Register field—Specifies a data register, Dx, to be shifted.


## Branch Conditionally

Bcc
(All ColdFire Processors; .L supported starting with V4)

Operation: $\quad$| If Condition True |
| :---: |
| Then PC $+d_{n}$ |$\rightarrow P C$

Assembler Syntax: Bcc.sz <label>
Attributes: $\quad$ Size $=$ byte, word, longword (longword supported starting with V4)
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 0 xFF , 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 $0 x 00$.

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

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

Condition Codes: Not affected

| Format: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 0 | 1 | 1 | 0 | Condition |  |  |  | 8-bit displacement |  |  |  |  |  |  |  |
|  | 16-bit displacement if 8-bit displacement $=0 \times 00$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 32 -bit displacement if 8 -bit displacement $=0 \times \mathrm{FF}$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## 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 $0 x 00$.
- 32-bit displacement field—Used when the 8-bit displacement contains 0xFF.

| Bcc | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | Yes | Yes |
| Operand sizes supported | B,W | B,W,L |

(All ColdFire Processors)
Operation: $\quad \sim(<$ bit number $>$ of Destination $) \rightarrow$ CCR[Z];
$\sim$ (<bit number> of Destination) $\rightarrow$ <bit number> of Destination
Assembler Syntax: BCHG.sz Dy, <ea>x
BCHG.sz \#<data>,<ea>x
Attributes: $\quad$ 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.


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:

|  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Instruction Format: | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Destination Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mode |  | Regist |  |
|  | 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 | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

- 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 |

## 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 | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Test a Bit and Clear

(All ColdFire Processors)
Operation: $\quad \sim(<$ bit number $>$ of Destination $) \rightarrow$ CCR[Z];
$0 \rightarrow$ <bit number> of Destination
Assembler Syntax: BCLR.sz Dy, <ea>x
BCLR.sz \#<data>,<ea>x
Attributes: $\quad$ 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.


X Not affected
N Not affected
Z Set if the bit tested is zero; cleared otherwise
$\checkmark$ Not affected
C Not affected

## Bit Number Static, Specified as Immediate Data:



## 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 | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

- 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 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

## 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 | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Branch Always

(All ColdFire Processors; .L supported starting with V4)
Operation: $\quad \mathrm{PC}+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$
Assembler Syntax: BRA.sz <label>
Attributes: $\quad$ Size $=$ byte, word, longword (longword supported starting with V4)
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 ( $0 x \mathrm{xF}$ ), 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 $0 \times 00$ (zero offset).
Condition codes: Not affected

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

## 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.

| BRA | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | Yes | Yes |
| Operand sizes supported | B,W | B,W,L |

# Freescale Semiconductor, Inc. 

BSET
Test a Bit and Set
BSET
(All ColdFire Processors)
Operation: $\quad \sim(<$ bit number $>$ of Destination $) \rightarrow$ CCR[Z];
$1 \rightarrow$ <bit number> of Destination
Assembler Syntax: BSET.sz Dy, <ea>x
BSET.sz \#<data >,<ea>x
Attributes: $\quad$ 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.


Bit Number Static, Specified as Immediate Data:

|  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Instruction | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |  | stinat | n | ve Addr |  |
| Format: |  |  |  |  |  |  |  |  |  |  |  | Mode |  | Regis |  |
|  | 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 |
| $A x$ | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

- Bit Number field-Specifies the bit number.


## Bit Number Dynamic, Specified in a Register:

Instruction Format:

| 15 | 14 | 13 | 12 | 11 | $10 \quad 9$ | 8 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | 0 | 0 | Data Register, Dy |  | 1 | 1 | 1 |  | stinat |  |  |  |
|  |  |  |  |  |  |  |  |  | Mode |  | gis |  |

## 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 | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Branch to Subroutine

(All ColdFire Processors; .L supported starting with V4)
Operation: $\quad$ SP $-4 \rightarrow$ SP; nextPC $\rightarrow$ (SP); PC $+\mathrm{d}_{\mathrm{n}} \rightarrow \mathrm{PC}$
Assembler Syntax: BSR.sz <label>

$$
\text { Attributes: } \quad \text { Size }=\text { byte, word, longword (longword supported starting with V4) }
$$

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 ( 0 xFF ), 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 $0 \times 00$ (zero offset).

Condition Codes: Not affected
Instruction
Format:


## 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.

| BSR | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | Yes | Yes |
| Operand sizes supported | B,W | B,W,L |

(All ColdFire Processors)
Operation: $\quad \sim(\langle$ bit number $\rangle$ of Destination $) \rightarrow$ CCR[Z]

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

Attributes: $\quad$ 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.


## Bit Number Static, Specified as Immediate Data:

Instruction
Format:


## 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 | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  | - |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Test a Bit

## Instruction Fields (continued):

- Bit Number field-Specifies the bit number.


## Bit Number Dynamic, Specified in a Register:

| Instruction Format: | 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 | 6 | 5 | 4 | 3 |  | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 0 | 0 | 0 | 0 | Data | egister, Dy | 1 | 0 | 0 |  | stina | E | ve |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | gis |  |

## 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 | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, \mathrm{Xi}\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

Operation: $\quad 0 \rightarrow$ Destination
Assembler Syntax: CLR.sz <ea>x
Attributes: $\quad$ Size $=$ byte, word, longword
Description: Clears the destination operand to 0 . The size of the operation may be specified as byte, word, or longword.


| X | Not affected |
| :--- | :--- |
| N | Always cleared |
| Z | Always set |
| V | Always cleared |
| C | Always cleared |

Instruction

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

## 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 |
| $A x$ | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

(All ColdFire Processors; .B and .W supported starting with V4)
Operation: $\quad$ Destination - Source $\rightarrow \mathrm{cc}$
Assembler Syntax: CMP.sz <ea>y,Dx

## Attributes:

Size $=$ byte, word, longword (byte, word supported starting with V4)
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.


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 |
| :--- |
| 1 |

## 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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| CMP | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | Yes | Yes |
| Operand sizes supported | L | $\mathrm{B}, \mathrm{W}, \mathrm{L}$ |

## Freescale Semiconductor, Inc.

CMPA
Compare Address
CMPA
(All ColdFire Processors; .W supported starting with V4)
Operation: $\quad$ Destination - Source $\rightarrow \mathrm{cc}$
Assembler Syntax: CMPA.sz <ea>y, Ax

## Attributes: $\quad$ Size $=$ word, longword (word supported starting with V4)

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.


| 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 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 1 | Address Register, Ax |  |  |  | Opmode |  |  | Source Effective Address |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mod |  | gis |  |

## Instruction Fields:

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

| Byte | Word | Longword | Operation |
| :---: | :---: | :---: | :---: |
| - | 011 | 111 | $\mathrm{Ax}-<\mathrm{ea}>\mathrm{y}$ |

- 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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| CMPA | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | Yes | Yes |
| Operand sizes supported | L | W,L |

## Freescale Semiconductor, Inc.

(All ColdFire Processors; .B and .W supported starting with V4)
Operation: Destination - Immediate Data $\rightarrow \mathrm{cc}$
Assembler Syntax: CMPI.sz \#<data>,Dx
Attributes: $\quad$ Size $=$ byte, word, longword (byte, word supported starting with V4)
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 | X | N | Z | V |  |
| :--- | :---: | :---: | :---: | :---: | :---: |
| Codes: | - | $*$ | $*$ | $*$ | $*$ |
|  |  |  |  |  |  |

$\begin{array}{ll}\mathrm{X} & \text { Not affected } \\ \mathrm{N} & \text { Set if the result is negative; cleared otherwise } \\ \mathrm{Z} & \text { Set if the result is zero; cleared otherwise } \\ \mathrm{V} & \text { Set if an overflow occurs; cleared otherwise } \\ \mathrm{C} & \text { Set if a borrow occurs; cleared otherwise }\end{array}$

Instruction
Format:


## 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

| CMPI | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | Yes | Yes |
| Operand sizes supported | L | $\mathrm{B}, \mathrm{W}, \mathrm{L}$ |

## Signed Divide

(All ColdFire Processors Starting with MCF5206e)
Operation: $\quad$ Destination/Source $\rightarrow$ Destination
Assembler Syntax: DIVS.W <ea>y,Dx $\quad$ 32-bit Dx/16-bit <ea>y $\rightarrow$ (16r:16q) in Dx DIVS.L <ea>y,Dx $\quad 32$-bit Dx/32-bit <ea>y $\rightarrow 32 q$ in Dx where $q$ indicates the quotient, and $r$ indicates the remainder

Attributes: $\quad$ 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 | N | Not affected <br> Cleared if overflow is detected; otherwise set if the quotient is negative, cleared if positive |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | - | * | * | * | 0 |  |  |
|  |  |  |  |  |  | z | Cleared if overflow is detected; otherwise set if the quotient is zero, cleared if nonzero |
|  |  |  |  |  |  | V | Set if an overflow occurs; cleared otherwise |
|  |  |  |  |  |  |  | Always cleared |



## Signed Divide

## 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 |
| $A y$ | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| Instruction Format: <br> (Longword) | 150 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  |  |  |
|  | 0 | Register, Dx |  |  | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |

## 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Unsigned Divide

(All ColdFire Processors Starting with MCF5206e)
Operation: $\quad$ Destination/Source $\rightarrow$ Destination
Assembler Syntax: DIVU.W <ea>y,Dx $\quad 32$-bit Dx/16-bit <ea>y $\rightarrow$ (16r:16q) in Dx DIVU.L <ea>y,Dx 32-bit Dx/32-bit <ea>y $\rightarrow 32 q$ in Dx where $q$ indicates the quotient, and $r$ indicates the remainder

Attributes: $\quad$ 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 | $\checkmark$ | C | XN | Not affected |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | - | * | * | * | 0 |  | Not affected <br> Cleared if overflow is detected; otherwise set if the |
|  |  |  |  |  |  | Z | quotient is negative, cleared if positive <br> 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 <br> Format: <br> (Word) | 15 | 14 | 13 | 12 | 1110 | 9 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 0 | 0 | 0 | Register, Dx | 0 | 1 | 1 |  | Source | ff | Addres |  |
|  |  |  |  |  |  |  |  |  |  | Mode |  | Regist |  |

## 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 | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| Instruction Format: (Longword) | 0 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  |  |  |
|  | 0 | Register, Dx |  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |

## 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

EOR

## Exclusive-OR Logical

(All ColdFire Processors)
Operation: $\quad$ Source ${ }^{\wedge}$ Destination $\rightarrow$ Destination

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

## Attributes: $\quad$ 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 | X | N | z | v | c |
| :---: | :---: | :---: | :---: | :---: | :---: |
|  | - | * | * | 0 | 0 |

Instruction
Format:


## 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 | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

EORI
Exclusive-OR Immediate
EORI
(All ColdFire Processors)
Operation: $\quad$ Immediate Data ${ }^{\wedge}$ Destination $\rightarrow$ Destination
Assembler Syntax: EORI.L \#<data>,Dx
Attributes: $\quad$ 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.


| 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:


## Instruction Fields:

- Register field - Destination data register, Dx.


## Freescale Semiconductor, Inc.

Sign-Extend
EXT, EXTB
(All ColdFire Processors)
Operation: $\quad$ Destination Sign-Extended $\rightarrow$ Destination

Assembler Syntax: EXT.W Dx
EXT.L Dx
EXTB.L Dx
extend byte to word
extend word to longword
extend byte to longword

Attributes: $\quad$ 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.


Instruction
Format:

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

## 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.


## Freescale Semiconductor, Inc.

## ILLEGAL

Take Illegal Instruction Trap
ILLEGAL
(All ColdFire Processors)
Operation: $\quad \mathrm{SP}-4 \rightarrow \mathrm{SP} ; \mathrm{PC} \rightarrow(\mathrm{SP})$ (forcing stack to be longword aligned)
SP - $2 \rightarrow$ SP; SR $\rightarrow$ (SP)
SP - $2 \rightarrow$ SP; Vector Offset $\rightarrow$ (SP)
$(\mathrm{VBR}+0 \mathrm{x} 10) \rightarrow \mathrm{PC}$
Assembler Syntax: ILLEGAL
Attributes: Unsized
Description: Execution of this instruction causes an illegal instruction exception. The opcode for ILLEGAL is $0 x 4 \mathrm{AFC}$.

Starting with V4 (for devices which have an MMU), the Supervisor Stack Pointer (SSP) is used for this instruction.

Condition Codes: Not affected.
Instruction

| 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 |

## Freescale Semiconductor, Inc.

## Jump

(All ColdFire Processors)
Operation: $\quad$ Destination Address $\rightarrow \mathrm{PC}$
Assembler Syntax: JMP <ea>y
Attributes: Unsized
Description: Program execution continues at the effective address specified by the instruction.

Condition Codes: Not affected.
Instruction

| 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 |  |  |  |  |

## 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 | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(d_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

## Jump to Subroutine

Operation: $\quad \mathrm{SP}-4 \rightarrow \mathrm{SP}$; nextPC $\rightarrow(\mathrm{SP}) ;$ Destination Address $\rightarrow \mathrm{PC}$
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 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regis |  |

## 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

## Freescale Semiconductor, Inc.

LEA
Load Effective Address
LEA
(All ColdFire Processors)
Operation: $\quad<e a>y \rightarrow A x$
Assembler Syntax: LEA.L <ea>y,Ax
Attributes: $\quad$ Size $=$ longword
Description: Loads the effective address into the specified address register, Ax.
Condition Codes: Not affected


## 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

## Link and Allocate

LINK
(All ColdFire Processors)
Operation: $\quad$ SP $-4 \rightarrow$ SP; Ay $\rightarrow$ (SP); SP $\rightarrow$ Ay; SP $+d_{n} \rightarrow$ SP
Assembler Syntax: LINK.W Ay,\#<displacement>
Attributes: $\quad$ 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.


# Freescale Semiconductor, Inc. 

(All ColdFire Processors)
Operation: $\quad$ Destination Shifted By Count $\rightarrow$ Destination

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

Attributes: $\quad$ 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.


## Freescale Semiconductor, Inc.



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

## Instruction Fields:

- Count/Register field
- If $\mathrm{i} / \mathrm{r}=0$, this field contains the shift count; values $1-7$ represent shifts of $1-7$; value of 0 specifies shift count of 8
— If $\mathrm{i} / \mathrm{r}=1$, 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
- $\quad i / r$ field
- 0 immediate shift count
- 1 register shift count
- Register field—Specifies a data register, Dx, to be shifted.


# Freescale Semiconductor, Inc. 

## Move 3-Bit Data Quick

(Supported starting with V4)
Operation: $\quad$ 3-bit Immediate Data $\rightarrow$ Destination
Assembler Syntax: MOV3Q.L \#<data>, <ea>x

## Attributes: $\quad$ 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.


Instruction Format:


## 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 |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |


| MOV3Q | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | L |

Operation: $\quad$ Source $\rightarrow$ Destination
Assembler Syntax: MOVE.sz <ea>y, <ea>x

## Attributes: $\quad$ 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 V4) is used to move a 3-bit immediate value to any effective destination address.

| Condition | X | N | Z | V | C |
| :--- | :---: | :---: | :---: | :---: | :---: |
|  |  | - | $*$ | $*$ | 0 |
| Codes: |  |  |  |  |  |

$\begin{array}{ll}\mathrm{X} & \text { Not affected } \\ \mathrm{N} & \text { Set if result is negative; cleared otherwise } \\ \mathrm{Z} & \text { Set if the result is zero; cleared otherwise } \\ \mathrm{V} & \text { Always cleared } \\ \mathrm{C} & \text { Always cleared }\end{array}$

Instruction
Format:

| 15 | 14 | 3 | 1110 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | Size | Destination Effective Address |  |  |  |  |  | Source Effective Address |  |  |  |
|  |  |  | Register |  |  | Mode |  |  | Mode |  | gis |  |

## Instruction fields:

- Size field—Specifies the size of the operand to be moved:
- 01 byte operation
- 11 word operation
- 10 longword operation
- 11 reserved
- 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 | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

## 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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

## NOTE:

Not all combinations of source/destination addressing modes are possible. The table below shows the possible combinations. Starting with V4, the combination of \#<xxx>, $d_{16}(A x)$ can be used with MOVE.B and MOVE.W opcodes.

| Source Addressing Mode | Destination Addressing Mode |
| :--- | :--- |
| Dy, Ay, (Ay), (Ay)+,-(Ay) | All possible |
| $\left(d_{16}, A y\right),(d 16, P C)$ | All possible except ( $\left.d_{8}, A x, X i\right),(x x x) . W,(x x x) . L$ |
| $(d 8, A y, X i),(d 8, P C, X i),(x x x) . W,(x x x) . L, \#<x x x>$ | All possible except $\left(d_{8}, A x, X i\right),(x x x) . W,(x x x) . L$ |


| MOVE | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | Yes | Yes |
| Operand sizes supported | B,W,L <br> except <br> MOVE.W,L <br> M<data>, $d_{16}(A x)$ | including <br> MOVE. $\mathrm{B}, \mathrm{W}\}$ \# $<$ data>, $\mathrm{d}_{16}(\mathrm{Ax})$ |

## Freescale Semiconductor, Inc.

MOVEA
Move Address from Source to Destination
MOVEA
(All ColdFire Processors)
Operation: $\quad$ Source $\rightarrow$ Destination
Assembler Syntax: MOVEA.sz <ea>y,Ax

## Attributes: $\quad$ 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 | 15 | 14 | 13 | 11 | $10 \quad 9$ | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 0 | Size | Destination Register, Ax |  | 0 | 0 | 1 |  | Sourc | Eff | A | re |  |
|  |  |  |  |  |  |  |  |  | Mode |  |  | gis |  |

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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

Operation: $\quad$ Registers $\rightarrow$ Destination;
Source $\rightarrow$ Registers

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

Attributes: $\quad$ 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 $\langle\mathrm{ea}\rangle \mathrm{x}$.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | - | - |
| Ax | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | - | - |
| $-(\mathrm{Ax})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

## Move Multiple Registers

## Instruction Fields (continued):

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

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| $A y$ | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | - | - |
| $-(A y)$ | - | - |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

- 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 |

# Freescale Semiconductor, Inc. 

## Move Quick

MOVEQ
(All ColdFire Processors)
Operation: $\quad$ Immediate Data $\rightarrow$ Destination
Assembler Syntax: MOVEQ.L \#<data>,Dx
Attributes: $\quad$ 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 | x | N | z | v | C | X | Not affected |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Codes: | - | * | * | 0 | 0 | N | Set if result is negative; cleared otherwise |
| Codes. |  |  |  |  |  | V | Set if the result is zero; cleared otherwise Always cleared |
|  |  |  |  |  |  | C | Always cleared |


| Instruction | 15 | 14 | 13 | 12 | 110 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 1 | 1 | Register, Dx | 0 |  |  |  | ed | D |  |  |  |

## 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 <br> from CCR

## MOVE <br> from CCR

 Condition Code Register(All ColdFire Processors)
Operation: $\quad$ CCR $\rightarrow$ Destination
Assembler Syntax: MOVE.W CCR,Dx
Attributes: $\quad$ 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 | 15 | 14 | 13 | 12 | ${ }^{11}$ | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 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 the

## Condition Code Register

(All ColdFire Processors)

Operation: $\quad$ Source $\rightarrow$ CCR
Assembler Syntax: MOVE.B Dy,CCR
MOVE.B \#<data>,CCR
Attributes: $\quad$ 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 | X | N | Z | V | C |
| :--- | :---: | :---: | :---: | :---: | :---: |
|  |  | $*$ | $*$ | $*$ | $*$ |
| Codes: |  |  |  |  |  |
|  |  |  |  |  |  |

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 |
| $A y$ | - | - |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(d_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

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

Assembler Syntax: MULS.W <ea>y,Dx $16 \times 16 \rightarrow 32$
MULS.L<ea>y,Dx $\quad 32 \times 32 \rightarrow 32$

Attributes: $\quad$ 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 68 K family processors.


Instruction Format:
(Word)

| 15 | 14 | 13 | 12 | 11109 | 8 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 0 | 0 | Register, Dx | 1 | 1 | 1 |  | Source Effective Address |  |  |  |
|  |  |  |  |  |  |  |  |  | Mode |  | Regis |  |

## 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: <br> (Longword) | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
|  | 0 |  | er |  | 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

- 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.


# Freescale Semiconductor, Inc. 

Operation: $\quad$ Source * Destination $\rightarrow$ Destination
Assembler Syntax: MULU.W <ea>y,Dx $\quad 16 \times 16 \rightarrow 32$
MULU.L <ea>y,Dx $\quad 32 \times 32 \rightarrow 32$

Attributes: $\quad$ 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 68 K family processors.


Instruction Format:
(Word)


## 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: <br> (Longword) | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
|  | 0 |  | er |  | 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

- 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.


# Freescale Semiconductor, Inc. 

## Move with Sign Extend

MVS
(Supported starting with V4)
Operation: $\quad$ Source with sign extension $\rightarrow$ Destination
Assembler Syntax: MVS.sz <ea>y,Dx

## Attributes: $\quad$ 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.


Instruction Format:

| 14 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | 0

## 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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . W$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(d_{16}, P C\right)$ | 111 | 010 |
| $\left(d_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| MVS | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | B,W |

# Freescale Semiconductor, Inc. 

(Supported starting with V4)
Operation: $\quad$ Source with zero fill $\rightarrow$ Destination
Assembler Syntax: MVZ.sz <ea>y,Dx

## Attributes: $\quad$ 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.


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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |


| MVZ | V2, V3 Core | V4 Core |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | B, W |

# Freescale Semiconductor, Inc. 

NEG
Negate
NEG
(All ColdFire Processors)
Operation: $\quad 0$ - Destination $\rightarrow$ Destination

## Assembler Syntax: NEG.L Dx

Attributes: $\quad$ 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 |  | Set the same as the carry bit |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | * | * | * | * | * | N | Set if the result is negative; cleared otherwise |
|  |  |  |  |  |  | V | Set if the result is zero; cleared otherwise <br> Set if an overflow is generated; cleared otherwise |
|  |  |  |  |  |  |  | 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.


# Freescale Semiconductor, Inc. 

Negate with Extend
NEGX
(All ColdFire Processors)
Operation: $\quad 0$ - Destination - CCR[X] $\rightarrow$ Destination
Assembler Syntax: NEGX.L Dx
Attributes: $\quad$ 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.


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

Instruction Fields:

- Register field - Specifies data register, Dx.


## No Operation

(All ColdFire Processors)
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 $0 x 4 \mathrm{E} 71$.

Condition Codes: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

## Freescale Semiconductor, Inc.

## Logical Complement

NOT

## (All ColdFire Processors)

Operation: $\quad \sim$ Destination $\rightarrow$ Destination
Assembler Syntax: NOT.L Dx
Attributes: $\quad$ 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.


Instruction

| 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.


## Freescale Semiconductor, Inc.

OR

## Inclusive-OR Logical

(All ColdFire Processors)
Operation: $\quad$ Source $\mid$ Destination $\rightarrow$ Destination

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

OR.L Dy, <ea>x
Attributes: $\quad$ 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.


Instruction
Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |  | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 0 | 0 | Register |  |  |  | Opmode |  |  |  | ctiv | dr |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | gis |  |

## Instruction Fields:

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

| Byte | Word | Longword | Operation |
| :---: | :---: | :---: | :---: |
| - | - | 010 | $<e a>y \mid \mathrm{Dx} \rightarrow \mathrm{Dx}$ |
| - | - | 110 | $\mathrm{Dy} \mid<\mathrm{ea}>\mathrm{x} \rightarrow<\mathrm{ea}>\mathrm{x}$ |

## Freescale Semiconductor, Inc.

OR

## Inclusive-OR Logical

OR

## Instruction Fields (continued):

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

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | 000 | reg. number:Dy |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

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

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | - | - |
| $A x$ | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

ORI

## Inclusive-OR

(All ColdFire Processors)
Operation: $\quad$ Immediate Data $\mid$ Destination $\rightarrow$ Destination
Assembler Syntax: ORI.L \#<data>,Dx
Attributes: $\quad$ 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.


| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 21 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 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.

Operation: $\quad \mathrm{SP}-4 \rightarrow \mathrm{SP} ;<\mathrm{ea}>\mathrm{y} \rightarrow(\mathrm{SP})$
Assembler Syntax: PEA.L <ea>y
Attributes: $\quad$ 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:


## 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 | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | 111 | 011 |

Operation: $\quad$ Set PST $=0 \mathrm{x} 4$

## 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 $0 \times 4 \mathrm{ACC}$.

Condition Codes: Not affected

| struction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

Operation: $\quad$ Destination/Source $\rightarrow$ Remainder
Assembler Syntax: REMS.L <ea>y,Dw:Dx $\quad 32$-bit Dx/32-bit <ea>y $\rightarrow 32 \mathrm{r}$ in Dw where $r$ indicates the remainder

Attributes: $\quad$ 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.


| 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:


## 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 |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

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

Operation: $\quad$ Destination/Source $\rightarrow$ Remainder
Assembler Syntax: REMU.L <ea>y,Dw:Dx 32 -bit Dx/32-bit <ea>y $\rightarrow 32$ r in Dw where $r$ indicates the remainder

Attributes: $\quad$ Size $=$ longword
Description: Divide the unsigned destination operand by the unsigned source and store the unsigned remainder in another register. If $D w$ is specified to be the same register as $D x$, 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.


| 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:


## 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 |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

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


## Return from Subroutine

(All ColdFire Processors)
Operation: $\quad(\mathrm{SP}) \rightarrow \mathrm{PC} ; \mathrm{SP}+4 \rightarrow \mathrm{SP}$
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 $0 \times 4 \mathrm{E} 75$.

Condition Codes: Not affected
Instruction

| 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 |

## 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: $\quad$ 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.


| 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

| 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.

| SATS | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | L |

Operation: If Condition True
Then 1s $\rightarrow$ Destination
Else 0s $\rightarrow$ Destination
Assembler Syntax: Scc.B Dx
Attributes: $\quad$ 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 $\mathrm{C}, \mathrm{N}, \mathrm{V}$, and Z represent $\mathrm{CCR}[\mathrm{C}], \mathrm{CCR}[\mathrm{N}], \mathrm{CCR}[\mathrm{V}]$, and $\mathrm{CCR}[\mathrm{Z}]$, respectively:

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

Condition Codes: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 0 | 1 |  | Cond |  |  | 1 | 1 | 0 | 0 | 0 |  | ster |  |

## Instruction Fields:

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

Operation: $\quad$ Destination - Source $\rightarrow$ Destination

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

SUB.L Dy,<ea>x
Attributes: $\quad$ 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.


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
$\checkmark$ 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 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

## Instruction Fields:

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

| Byte | Word | Longword | Operation |
| :---: | :---: | :---: | :---: |
| - | - | 010 | $\mathrm{Dx}-<\mathrm{ea}>\mathrm{y} \rightarrow \mathrm{Dx}$ |
| - | - | 110 | $<\mathrm{ea}>\mathrm{x}-\mathrm{Dy} \rightarrow<\mathrm{ea}>\mathrm{x}$ |

## Freescale Semiconductor, Inc.

SUB
Subtract
SUB

## Instruction Fields (continued):

- Effective Address field-Determines addressing mode
- For the source operand $<e a>y$, 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 |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

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

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | - | - |
| $A x$ | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

Operation: $\quad$ Destination - Source $\rightarrow$ Destination

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

## Attributes: $\quad$ 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 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 |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(d_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

## Subtract Immediate

Operation: $\quad$ Destination - Immediate Data $\rightarrow$ Destination
Assembler Syntax: SUBI.L \#<data>,Dx

## Attributes: $\quad$ 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.


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
$\checkmark$ Set if an overflow is generated; cleared otherwise
C Set if an carry is generated; cleared otherwise

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 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.

MOTOROLA

Operation: $\quad$ Destination - Immediate Data $\rightarrow$ Destination
Assembler Syntax: SUBQ.L \#<data>, <ea>x

## Attributes: $\quad$ 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.


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 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |  | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 0 | 1 |  | Data |  | 1 | 1 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | gist |  |

## 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 |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Subtract Extended

Operation: $\quad$ Destination - Source - CCR $[\mathrm{X}] \rightarrow$ Destination
Assembler Syntax: SUBX.L Dy,Dx
Attributes: $\quad$ 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 | x | N | z | v | C | X | Set the same as the carry bit |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Codes: | * | * | * | * | * | N | Set if the result is negative; cleared otherwise |
|  |  |  |  |  |  | V | Cleared if the result is non-zero; unchanged otherwise Set if an overflow is generated; cleared otherwise |
|  |  |  |  |  |  |  | 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

## Instruction Fields:

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

MOTOROLA

## Freescale Semiconductor, Inc.

Swap Register Halves
SWAP
(All ColdFire Processors)
Operation: $\quad$ Register[31:16] $\leftrightarrow$ Register[15:0]
Assembler Syntax: SWAP.W Dx
Attributes: $\quad$ Size $=$ Word
Description: Exchange the 16-bit words (halves) of a data register.


| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Format: | 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.

Operation: $\quad$ Destination Tested $\rightarrow \mathrm{CCR} ; 1 \rightarrow$ bit 7 of Destination
Assembler Syntax: TAS.B <ea>x
Attributes: $\quad$ 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 68 K Family processors, Dx is not a supported addressing mode.


[^1]| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 |  | 4 | 3 |  | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regis |  |

## 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 | - | - |
| $A x$ | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, \mathrm{Xi}^{\prime}\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |


| TAS | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | B |

## Trap False

(All ColdFire Processors)
Operation: No Operation

| Assembler Syntax: | TPF | $\mathrm{PC}+2 \rightarrow \mathrm{PC}$ |
| :--- | :--- | :--- |
| TPF.W \#<data> | $\mathrm{PC}+4 \rightarrow \mathrm{PC}$ |  |
| TPF.L \#<data> | $\mathrm{PC}+6 \rightarrow \mathrm{PC}$ |  |

Attributes: $\quad$ 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. $\{\mathrm{W}, \mathrm{L}\}$ 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
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 Fields:

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

Trap
(All ColdFire Processors)
Operation: $\quad 1 \rightarrow$ S-Bit of SR
SP - $4 \rightarrow$ SP; nextPC $\rightarrow$ (SP); SP $-2 \rightarrow \mathrm{SP}$;
$\mathrm{SR} \rightarrow$ (SP); SP - $2 \rightarrow$ SP; Format/Offset $\rightarrow$ (SP);
$(\mathrm{VBR}+0 \times 80+4 * \mathrm{n}) \rightarrow \mathrm{PC}$
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 $0 \times 80$ 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.

Note also that for processors beginning with V4 (for devices which have an MMU), the SSP is used for this operation.

Condition Codes: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :--- |
| Format: | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |  | Vector |  |  |

## Instruction Fields:

- Vector field-Specifies the trap vector to be taken.


## Freescale Semiconductor, Inc.

## Test an Operand

(All ColdFire Processors)
Operation: $\quad$ Source Operand Tested $\rightarrow$ CCR
Assembler Syntax: TST.sz <ea>y
Attributes: $\quad$ 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.


Instruction
Format:


## 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, $\langle e a\rangle x$, as listed in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | 000 | reg. number:Dx |
| $A x^{*}$ | 001 | reg. number:Ax |
| $(A x)$ | 010 | reg. number:Ax |
| $(A x)+$ | 011 | reg. number:Ax |
| $-(A x)$ | 100 | reg. number:Ax |
| $\left(d_{16}, A x\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | 110 | reg. number:Ax |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(x x x) . \mathrm{L}$ | 111 | 001 |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | 111 | 011 |

*The Ax addressing mode is allowed only for word and longword operations.
(All ColdFire Processors)
Operation: $\quad \mathrm{Ax} \rightarrow \mathrm{SP} ;(\mathrm{SP}) \rightarrow \mathrm{Ax} ; \mathrm{SP}+4 \rightarrow \mathrm{SP}$

## 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 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 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.

Assembler Syntax: WDDATA.sz <ea>y

## Attributes: $\quad$ Size = byte, word, longword

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

The execution of this instruction generates a processor status encoding matching the PULSE instruction ( $0 \times 4$ ) before the referenced operand is displayed on the DDATA outputs.

Condition Codes: Not affected
Instruction
Format:


## Instruction Fields:

- Size field-specifies the size of the operand data
- 00 byte operation
- 01 word operation
- 10 longword operation
- 11 reserved
- Source Effective Address field—Determines the addressing mode for the operand, <ea>y, to be written to the DDATA signal pins; use only those memory alterable addressing modes listed in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| $A y$ | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | 110 | reg. number:Ay |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | 111 | 000 |
| $(\mathrm{xxx}) . \mathrm{L}$ | 111 | 001 |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

# Chapter 5 <br> Multiply-Accumulate Unit (MAC) User Instructions 

This chapter describes the user instructions for the optional multiply-accumulate (MAC) unit in the ColdFire family of processors. A detailed discussion of each instruction description is arranged in alphabetical order by instruction mnemonic.

For instructions implemented by the Enhanced Multiply-Accumulate Unit (EMAC), refer to Chapter 6, "Enhanced Multiply-Accumulate Unit (EMAC) User Instructions."

Operation:

$$
\mathrm{ACC}+(\mathrm{Ry} * \mathrm{Rx})\{\ll \mid \gg\} \mathrm{SF} \rightarrow \mathrm{ACC}
$$

Assembler syntax: MAC.sz Ry. $\{\mathrm{U}, \mathrm{L}\}, \mathrm{Rx} .\{\mathrm{U}, \mathrm{L}\}$ SF
Attributes:
Size = word, longword
Description: Multiply two 16- or 32 -bit numbers to yield a 32 -bit result, then add this product, shifted as defined by the scale factor, to the accumulator. If 16 -bit operands are used, the upper or lower word of each register must be specified.

| Condition | N | Z | v |  | wise |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Codes | * | * | * | Z | Set if the result is zero; cleared otherwise | (MACSR):

Instruction
Format:

| 15 | 14 | 13 | 12 | 11 | $10 \quad 9$ | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rx |  | 0 | 0 | Rx | 0 | 0 | Register, Ry |  |  |  |
| - | - | - | - | sz | Scale Factor | 0 | U/Lx | U/Ly | - | - | - | - | - | - |

## Instruction Fields:

- Register Rx[6,11-9] field- Specifies a source register operand, where $0 \times 0$ is D0,..., $0 \times 7$ is $\mathrm{D} 7,0 \times 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A7. Note that bit 6 of the operation word is the msb of the register number field.
- Register Ry[3-0] field - Specifies a source register operand, where 0x0 is D0,..., 0 x 7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A7.
- sz field-Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field -Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product << 1
- 10 reserved
- 11 product >> 1


# Freescale Semiconductor, Inc. 

MAC

## Multiply Accumulate

## Instruction Fields (continued):

- U/Lx-Specifies which 16-bit operand of the source register, Rx , is used for a word-sized operation.
- 0 lower word
- 1 upper word
- U/Ly-Specifies which 16-bit operand of the source register, Ry, is used for a word-sized operation.
- 0 lower word
- 1 upper word

Operation:
$\mathrm{ACC}+(\mathrm{Ry} * \mathrm{Rx})\{\ll \mid \gg\} \mathrm{SF} \rightarrow \mathrm{ACC}$ $(<e a>y) \rightarrow R w$

Assembler syntax: MAC.sz Ry.\{U,L\},Rx.\{U,L\}SF,<ea>y\&,Rw where \& enables the use of the MASK

Attributes: $\quad$ Size $=$ word, longword
Description: Multiply two 16- or 32 -bit numbers to yield a 32 -bit result, then add this product, shifted as defined by the scale factor, to the accumulator. If 16 -bit operands are used, the upper or lower word of each register must be specified. In parallel with this operation, a 32 -bit operand is fetched from the memory location defined by $\langle e a\rangle y$ and loaded into the destination register, Rw. If the MASK register is specified to be used, the <ea>y operand is ANDed with MASK prior to being used by the instruction.

| Condition | N | Z | V | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Codes | * | * | * | Z | Set if the result is zero; cleared otherwise |
| (MACSR): |  |  |  |  | Set if an overflow is generated; unchanged otherwis |

Instruction
Format:


## Instruction Fields:

- Register Rw[6,11-9] field- Specifies the destination register, Rw, where 0x0 is $\mathrm{D} 0, \ldots, 0 \mathrm{x} 7$ is $\mathrm{D} 7,0 \times 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A7. Note that bit 6 of the operation word is the msb of the register number field.
- Source Effective Address field specifies the source operand, <ea>y; use addressing modes in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## MAC

## Multiply Accumulate with Load

## Instruction Fields (continued):

- Register Rx field - Specifies a source register operand, where $0 x 0$ is $D 0, \ldots, 0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .
- sz field—Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field - Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product $\ll 1$
- 10 reserved
- 11 product >> 1
- U/Lx, U/Ly—Specifies which 16-bit operand of the source register, Rx/Ry, is used for a word-sized operation.
- 0 lower word
- 1 upper word
- Mask field - Specifies whether or not to use the MASK register in generating the source effective address, <ea>y.
- 0 do not use MASK
- 1 use MASK
- Register Ry field - Specifies a source register operand, where $0 x 0$ is $D 0, \ldots, 0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## Freescale Semiconductor, Inc.

## MOVE <br> from ACC

## MOVE

Move from
Accumulator

Operation: $\quad$ Accumulator $\rightarrow$ Destination
Assembler syntax: MOVE.L ACC,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32-bit value from the accumulator into a general-purpose register, Rx. When operating in fractional mode ( $\operatorname{MACSR}[\mathrm{F} / \mathrm{I}]=1$ ), if MACSR[S/U] is set, the accumulator contents are rounded to a 16-bit value and stored in the lower 16-bits of the destination register Rx. The upper 16 bits of the destination register are zero-filled. The value of the accumulator is not affected by this rounding operation.

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |  | Register, Rx |  |  |

Instruction Field:

- Register Rx field - Specifies a destination register operand, where 0x0 is D0,..., 0x7 is $\mathrm{D} 7,0 \times 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


# MOVE <br> from MACSR 

Move from the
MACSR

Operation: $\quad$ MACSR $\rightarrow$ Destination
Assembler Syntax: MOVE.L MACSR,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves the MACSR register contents into a general-purpose register, Rx. Rx[31:8] are cleared.

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |  | Regis |  |  |

Instruction Field:

- Register Rx field - Specifies a source register operand, where $0 x 0$ is $D 0, \ldots, 0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## Move from the <br> MAC MASK Register

MOVE
from MASK

Operation: $\quad$ MASK $\rightarrow$ Destination
Assembler Syntax: MOVE.L MASK,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves the MASK register contents into a general-purpose register, Rx. $\operatorname{Rx}[31: 16]$ are set to $0 x F F F F$.

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | ${ }^{11}$ | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 21 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |  | Register, Rx |  |

Instruction Field:

- Register Rx field - Specifies a source register operand, where $0 x 0$ is $D 0, \ldots, 0 \times 7$ is D7, 0 x 8 is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## Freescale Semiconductor, Inc.

# MOVE MACSR to CCR 

## Move from the MACSR to the CCR

Operation: $\quad$ MACSR $\rightarrow$ CCR
Assembler Syntax: MOVE.L MACSR,CCR
Attributes: $\quad$ Size $=$ longword
Description: Moves the MACSR condition codes into the Condition Code Register. The opcode for MOVE MACSR to CCR is $0 \times \mathrm{A} 9 \mathrm{C} 0$.

MACSR: Not affected


Instruction
Format:

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

# Freescale Semiconductor, Inc. 

## Accumulator

Operation: $\quad$ Source $\rightarrow$ Accumulator
Assembler syntax: MOVE.L Ry,ACC
MOVE.L \#<data>,ACC
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32 -bit value from a register or an immediate operand into the accumulator.


Instruction


## Instruction Fields:

- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

Move to the
MOVE
to MACSR

Operation: $\quad$ Source $\rightarrow$ MACSR
Assembler Syntax: MOVE.L Ry,MACSR
MOVE.L \#<data>,MACSR
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32-bit value from a register or an immediate operand into the MACSR.

MACSR:

Source <ea> bit:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| - | - | - | - | - | - | - | - | OMC | S/U | F/I | R/T | N | Z | V | - |
| - | - | - | - | - | - | - | - | [7] | [6] | [5] | [4] | [3] | [2] | [1] | - |

Instruction
Format:


## Instruction Fields:

- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

Operation: $\quad$ Source $\rightarrow$ MASK
Assembler Syntax: MOVE.L Ry,MASK
MOVE.L \#<data>,MASK
Attributes: $\quad$ Size $=$ longword
Description: Moves a 16-bit value from the lower word of a register or an immediate operand into the MASK register.

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regis |  |

## Instruction Fields:

- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Operation: $\quad$ ACC $-(\mathrm{Ry} * \mathrm{Rx})\{\ll \mid \gg\} \mathrm{SF} \rightarrow \mathrm{ACC}$
Assembler syntax: MSAC.sz Ry. $\{\mathrm{U}, \mathrm{L}\}, \mathrm{Rx} .\{\mathrm{U}, \mathrm{L}\}$ SF
Attributes: $\quad$ Size $=$ word, longword
Description: Multiply two 16- or 32-bit numbers to yield a 32 -bit result, then subtract this product, shifted as defined by the scale factor, from the accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified.
 (MACSR):

Instruction
Format:

| 15 | 14 | 13 | 12 | 11 | $10 \quad 9$ | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rx |  | 0 | 0 | Rx | 0 | 0 | Register, Ry |  |  |  |
| - | - | - | - | sz | Scale Factor | 1 | U/Lx | U/Ly | - | - | - | - | - | - |

## Instruction Fields:

- Register Rx[6,11-9] field- Specifies a source register operand, where $0 \times 0$ is D0,..., $0 \times 7$ is $\mathrm{D} 7,0 \times 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A7. Note that bit 6 of the operation word is the msb of the register number field.
- Register Ry[3-0] field - Specifies a source register operand, where 0x0 is D0,..., 0 x 7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A7.
- sz field—Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field -Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product << 1
- 10 reserved
- 11 product >> 1


# Freescale Semiconductor, Inc. 

Multiply Subtract
MSAC

## Instruction Fields (continued):

- U/Lx—Specifies which 16-bit operand of the source register, Rx , is used for a word-sized operation.
- 0 lower word
- 1 upper word
- U/Ly-Specifies which 16-bit operand of the source register, Ry, is used for a word-sized operation.
- 0 lower word
- 1 upper word


## MSAC

## Multiply Subtract with Load

Operation:
ACC - (Ry * Rx $)\{\ll \mid \gg\}$ SF $\rightarrow$ ACC
(<ea>y) $\rightarrow$ Rw
Assembler syntax: MSAC.sz Ry. $\{\mathrm{U}, \mathrm{L}\}, \mathrm{Rx} .\{\mathrm{U}, \mathrm{L}\}$ SF, <ea>y\&,Rw where \& enables the use of the MASK

Attributes: $\quad$ Size $=$ word, longword
Description: Multiply two 16- or 32-bit numbers to yield a 32-bit result, then subtract this product, shifted as defined by the scale factor, from the accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified. In parallel with this operation, a 32-bit operand is fetched from the memory location defined by <ea>y and loaded into the destination register, Rw. If the MASK register is specified to be used, the <ea>y operand is ANDed with MASK prior to being used by the instruction.

| Condition | N | Z | V | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Codes | * | * | * | Z | Set if the result is zero; cleared otherwise |
| (MACSR): |  |  |  |  | Set if an overflow is generated; unchanged otherwis |

Instruction
Format:


## Instruction Fields:

- Register Rw[6,11-9] field- Specifies the destination register, Rw, where 0x0 is $\mathrm{D} 0, \ldots, 0 \times 7$ is $\mathrm{D} 7,0 \times 8$ is $\mathrm{A} 0, \ldots, 0 \times F$ is A 7 . Note that bit 6 of the operation word is the msb of the register number field.
- Source Effective Address field specifies the source operand, <ea>y; use addressing modes in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Instruction Fields (continued):

- Register Rx field - Specifies a source register operand, where $0 x 0$ is $D 0, \ldots, 0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .
- sz field—Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field - Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product $\ll 1$
- 10 reserved
- 11 product >> 1
- U/Lx, U/Ly—Specifies which 16-bit operand of the source register, Rx/Ry, is used for a word-sized operation.
- 0 lower word
- 1 upper word
- Mask field - Specifies whether or not to use the MASK register in generating the source effective address, <ea>y.
- 0 do not use MASK
- 1 use MASK
- Register Ry field - Specifies a source register operand, where $0 x 0$ is $D 0, \ldots, 0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


# Chapter 6 Enhanced Multiply-Accumulate Unit (EMAC) User Instructions 

This chapter describes the user instructions for the optional enhanced multiply-accumulate (EMAC) unit in the ColdFire family of processors. A detailed discussion of each instruction description is arranged in alphabetical order by instruction mnemonic.

For instructions implemented by the Multiply-Accumulate Unit (MAC), refer to Chapter 5, "Multiply-Accumulate Unit (MAC) User Instructions."

Operation: $\quad \mathrm{ACCx}+(\mathrm{Ry} * \mathrm{Rx})\{\ll \mid \gg\} \mathrm{SF} \rightarrow \mathrm{ACCx}$
Assembler syntax: MAC.sz Ry.\{U,L\},Rx.\{U,L\}SF,ACCx
Attributes: $\quad$ Size $=$ word, longword
Description: Multiply two 16- or 32-bit numbers to yield a 40-bit result, then add this product, shifted as defined by the scale factor, to an accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified.

| Condition Codes <br> (MACSR): | N | z | v | PAV | EV | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | * | * | * | * | * | z | Set if the result is zero; cleared otherwis |
|  |  |  |  |  |  |  | Set if a product or accumulation overflow is generated or PAV $\mathrm{x}=1$; cleared otherwise |
|  |  |  |  |  |  | PAVx | Set if a product or accumulation overflow is generated; unchanged otherwise |
|  |  |  |  |  |  | EV | Set if accumulation overflows lower 32 bits in integer mode or lower 40 bits in fractional mode; cleared otherwise |

Instruction
Format:

| 15 | 14 | 13 | 12 | 11 | $10 \quad 9$ | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rx |  | 0 | $\begin{gathered} \text { ACC } \\ \text { Isb } \end{gathered}$ | $\begin{gathered} \mathrm{Rx} \\ \mathrm{msb} \end{gathered}$ | 0 | 0 | Register, Ry |  |  |  |
| - | - | - | - | SZ | Scale <br> Factor | 0 | U/Lx | U/Ly | - | ACC <br> msb | - | - | - | - |

## Instruction Fields:

- Register Rx[6,11-9] field- Specifies a source register operand, where $0 x 0$ is D0,..., $0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 . Note that bit 6 of the operation word is the msb of the register number field.
- ACC field—Specifies the destination accumulator, ACCx. Bit 4 of the extension word is the msb and bit 7 of the operation word is the lsb. The value of these two bits specify the accumulator number as shown in the following table:

| Ext <br> word [4] | Op word <br> [7] | Accumulator |
| :---: | :---: | :---: |
| 0 | 0 | ACC0 |
| 0 | 1 | ACC1 |
| 1 | 0 | ACC2 |
| 1 | 1 | ACC3 |

- Register Ry[3-0] field - Specifies a source register operand, where 0x0 is D0,..., 0 x 7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## Multiply Accumulate

## Instruction Fields (continued):

- sz field—Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field - Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product << 1
- 10 reserved
- 11 product >> 1
- U/Lx—Specifies which 16-bit operand of the source register, Rx , is used for a word-sized operation.
- 0 lower word
- 1 upper word
- U/Ly—Specifies which 16-bit operand of the source register, Ry, is used for a word-sized operation.
- 0 lower word
- 1 upper word

Operation: $\quad \mathrm{ACCx}+(\mathrm{Ry} * \mathrm{Rx})\{\ll \mid \gg\} \mathrm{SF} \rightarrow \mathrm{ACCx}$ $(<e a>y) \rightarrow R w$

Assembler syntax: MAC.sz Ry. $\{\mathrm{U}, \mathrm{L}\}, R x .\{\mathrm{U}, \mathrm{L}\}$ SF, <ea>y\&,Rw,ACCx where \& enables the use of the MASK

Attributes: $\quad$ Size $=$ word, longword
Description: Multiply two 16- or 32 -bit numbers to yield a 40 -bit result, then add this product, shifted as defined by the scale factor, to an accumulator. If 16 -bit operands are used, the upper or lower word of each register must be specified. In parallel with this operation, a 32 -bit operand is fetched from the memory location defined by <ea>y and loaded into the destination register, Rw. If the MASK register is specified to be used, the <ea>y operand is ANDed with MASK prior to being used by the instruction.

| Condition Codes <br> (MACSR): | N | Z | V | PAV | EV | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | * | * | * | * | * | Z | Set if the result is zero; cleared otherwise |
|  |  |  |  |  |  | V | Set if a product or accumulation overflow is generated or PAV $\mathrm{x}=1$; cleared otherwise |
|  |  |  |  |  |  |  | Set if a product or accumulation overflow is generated; unchanged otherwise |
|  |  |  |  |  |  | EV | Set if accumulation overflows lower 32 bits (integer) |



## Instruction Fields:

- Register Rw[6,11-9] field- Specifies the destination register, Rw, where 0x0 is $\mathrm{D} 0, \ldots, 0 \times 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 . Note that bit 6 of the operation word is the msb of the register number field.
- ACC field—Specifies the destination accumulator, ACCx. Bit 4 of the extension word is the msb and bit 7 of the operation word is the inverse of the lsb (unlike the MAC instruction without a load). The value of these two bits specify the accumulator number as shown in the following table:

| Ext word [4] | Op word [7] | Accumulator |
| :---: | :---: | :---: |
| 0 | 1 | ACC0 |
| 0 | 0 | ACC1 |
| 1 | 1 | ACC2 |
| 1 | 0 | ACC3 |

## Instruction Fields (continued):

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

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

- Register Rx field - Specifies a source register operand, where $0 \times 0$ is D0,..., $0 \times 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .
- sz field—Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field -Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product $\ll 1$
- 10 reserved
- 11 product >> 1
- U/Lx, U/Ly—Specifies which 16-bit operand of the source register, Rx/Ry, is used for a word-sized operation.
- 0 lower word
- 1 upper word
- Mask field - Specifies whether or not to use the MASK register in generating the source effective address, <ea>y.
- 0 do not use MASK
- 1 use MASK
- Register Ry field - Specifies a source register operand, where $0 x 0$ is D0,..., $0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## MOVCLR

## Move from Accumulator and Clear

Operation: $\quad$ Accumulator $\rightarrow$ Destination; $0 \rightarrow$ Accumulator
Assembler syntax: MOVCLR.L ACCy,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32 -bit accumulator value into a general-purpose register, Rx. The selected accumulator is cleared after the store to the Rx register is complete. This clearing operation also affects the accumulator extension bytes and the product/accumulation overflow indicator. The store accumulator function is quite complex, and a function of the EMAC configuration defined by the MACSR. The following pseudocode defines its operation; in this description, ACC[47:0] represents the concatenation of the 32-bit accumulator and the 16 -bit extension word.

```
if MACSR[S/U,F/I] == 00 /* signed integer mode
    if MACSR[OMC] == 0
        then ACC[31:0] -> Rx /* saturation disabled
        else if ACC[47:31] == 0x0000_0 or 0xFFFF_1
            then ACC[31:0] }->\mathrm{ Rx
            else if ACC[47] == 0
                then 0x7FFF_FFFF -> Rx
            else 0x8000_0000 -> Rx
if MACSR[S/U,F/I] == 10 /* unsigned integer mode
    if MACSR[OMC] == 0
        then ACC[31:0] -> Rx /* saturation disabled
        else if ACC[47:32] == 0x0000
            then ACC[31:0] }->\textrm{Rx
            else 0xFFFF_FFFF -> Rx
if MACSR[F/I] == 1 /* signed fractional mode
    if MACSR[OMC,S/U,R/T] == 000 /* no saturation, no 16-bit rnd, no 32-bit rnd
        then ACC[39:8] }->\mathrm{ Rx
    if MACSR[OMC,S/U,R/T] == 001 /* no saturation, no 16-bit rnd, 32-bit rnd
        then ACC[39:8] rounded by contents of [7:0] }->\textrm{Rx
    if MACSR[OMC,S/U] == 01 /* no saturation, 16-bit rounding
        then 0 }->\mathrm{ Rx[31:16]
            ACC[39:24] rounded by contents of [23:0] }->\textrm{Rx}[15:0
    if MACSR[OMC,S/U,R/T] == 100 /* saturation, no 16-bit rnd, no 32-bit rnd
        if ACC[47:39] == 0x00_0 or 0xFF_1
            then ACC[39:8] }->\textrm{Rx
            else if ACC[47] == 0
            then 0x7FFF_FFFFF->Rx
            else 0x8000_0000 -> Rx
    if MACSR[OMC,S/U,R/T] == 101 /* saturation, no 16-bit rnd, 32-bit rounding
        Temp[47:8] = ACC[47:8] rounded by contents of [7:0]
        if Temp[47:39] == 0x00_0 or 0xFF_1
            then Temp[39:8] }->\mathrm{ Rx
            else if Temp[47] == 0
                    then 0x7FFF_FFFFF -> Rx
            else 0x8000_0000 -> Rx
```


## Move from <br> Accumulator and Clear

```
    if MACSR[OMC,S/U] == 11/* saturation, 16-bit rounding
    Temp[47:24] = ACC[47:24] rounded by the contents of [23:0]
    if Temp[47:39] == 0x00_0 or 0xFF_1
        then 0 }->\mathrm{ Rx[31:16]
            Temp[39:24] }->\textrm{Rx}[15:0
        else if Temp[47] == 0
            then 0x0000_7FFF -> Rx
            else 0x0000_8000 -> Rx
0 -> ACCx, ACCextx, MACSR[PAVx]
```

| Condition | N | Z | V | PAV | E | N | Not affected |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Codes | - | - | - | 0 |  | Z | Not affected |
| (MACSR): |  |  |  |  |  |  | Not affected Cleared |

Instruction

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

## Instruction Fields:

- ACC-Specifies the destination accumulator. The value of bits [10:9] specify the accumulator number.
- Register Rx field - Specifies a destination register operand, where 0x0 is D0,..., 0x7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .

Move from
Accumulator

Operation: $\quad$ Accumulator $\rightarrow$ Destination
Assembler syntax: MOVE.L ACCy,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32 -bit value from an accumulator into a general-purpose register, Rx.

The store accumulator function is quite complex, and a function of the EMAC configuration defined by the MACSR. The following pseudocode defines its operation; in this description, ACC [47:0] represents the concatenation of the 32-bit accumulator and the 16-bit extension word.

```
if MACSR[S/U,F/I] == 00 /* signed integer mode
    if MACSR[OMC] == 0
        then ACC[31:0] -> Rx /* saturation disabled
        else if ACC[47:31] == 0x0000_0 or 0xFFFF_1
            then ACC[31:0] }->\mathrm{ Rx
            else if ACC[47] == 0
                then 0x7FFF_FFFF -> Rx
                else 0x8000_0000 }->\textrm{Rx
if MACSR[S/U,F/I] == 10 /* unsigned integer mode
    if MACSR[OMC] == 0
        then ACC[31:0] -> Rx /* saturation disabled
        else if ACC[47:32] == 0x0000
            then ACC[31:0] }->\mathrm{ Rx
            else 0xFFFF_FFFF -> Rx
if MACSR[F/I] == 1 /* signed fractional mode
    if MACSR[OMC,S/U,R/T] == 000 /* no saturation, no 16-bit rnd, no 32-bit rnd
        then ACC[39:8] }->\textrm{Rx
    if MACSR[OMC,S/U,R/T] == 001 /* no saturation, no 16-bit rnd, 32-bit rnd
        then ACC[39:8] rounded by contents of [7:0] }->\textrm{Rx
    if MACSR[OMC,S/U] == 01 /* no saturation, 16-bit rounding
        then 0 }->\mathrm{ Rx[31:16]
            ACC[39:24] rounded by contents of [23:0] }->\mathrm{ Rx[15:0]
    if MACSR[OMC,S/U,R/T] == 100 /* saturation, no 16-bit rnd, no 32-bit rnd
        if ACC[47:39] == 0x00_0 or 0xFF_1
            then ACC[39:8] }->\mathrm{ Rx
            else if ACC[47] == 0
            then 0x7FFF_FFFF }->\mathrm{ Rx
            else 0x8000_0000 -> Rx
    if MACSR[OMC,S/U,R/T] == 101 /* saturation, no 16-bit rnd, 32-bit rounding
        Temp[47:8] = ACC[47:8] rounded by contents of [7:0]
        if Temp[47:39] == 0x00_0 or 0xFF_1
            then Temp[39:8] }->\mathrm{ Rx
            else if Temp[47] == 0
            then 0x7FFF_FFFF }->\textrm{Rx
            else 0x8000_0000 -> Rx
```

Move from an

## Accumulator

```
    if MACSR[OMC,S/U] == 11/* saturation, 16-bit rounding
    Temp[47:24] = ACC[47:24] rounded by the contents of [23:0]
    if Temp[47:39] == 0x00_0 or 0xFF_1
        then 0 }->\mathrm{ Rx[31:16]
            Temp[39:24] }->\textrm{Rx[15:0]
        else if Temp[47] == 0
            then 0x0000_7FFF -> Rx
            else 0x0000_8000 -> Rx
0 -> ACCx, ACCextx, MACSR[PAVx]
```



Instruction

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

## Instruction Fields:

- ACC—Specifies the destination accumulator. The value of bits [10:9] specify the accumulator number.
- Register Rx field - Specifies a destination register operand, where 0x0 is D0,..., 0x7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


# MOVE from ACCext01 

## MOVE from <br> ACCext01

 0 and 1 ExtensionsOperation: $\quad$ Accumulator 0 and 1 extension words $\rightarrow$ Destination
Assembler syntax: MOVE.L ACCext01,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves the contents of the four extension bytes associated with accumulators 0 and 1 into a general-purpose register. The accumulator extension bytes are stored as shown in the following table. Note the position of the LSB of the extension within the combined 48 -bit accumulation logic is dependent on the operating mode of the EMAC (integer versus fractional).

| Accumulator Extension <br> Byte | Destination <br> Data Bits |
| :---: | :---: |
| ACCext1[15:8] | $[31: 24]$ |
| ACCext1[7:0] | $[23: 16]$ |
| ACCext0[15:8] | $[15: 8]$ |
| ACCext0[7:0] | $[7: 0]$ |

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |  | Register, Rx |  |

Instruction Field:

- Register Rx field - Specifies a source register operand, where 0x0 is D0,..., 0x7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A7.


# MOVE from ACCext23 

## MOVE from <br> ACCext23

 2 and 3 ExtensionsOperation: $\quad$ Accumulator 2 and 3 extension words $\rightarrow$ Destination
Assembler syntax: MOVE.L ACCext23,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves the contents of the four extension bytes associated with accumulators 2 and 3 into a general-purpose register. The accumulator extension bytes are stored as shown in the following table. Note the position of the LSB of the extension within the combined 48 -bit accumulation logic is dependent on the operating mode of the EMAC (integer versus fractional).

| Accumulator Extension <br> Byte | Destination <br> Data Bits |
| :---: | :---: |
| ACCext3[15:8] | $[31: 24]$ |
| ACCext3[7:0] | $[23: 16]$ |
| ACCext2[15:8] | $[15: 8]$ |
| ACCext2[7:0] | $[7: 0]$ |

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |  | Register, Rx |  |

Instruction Field:

- Register Rx field - Specifies a source register operand, where 0x0 is D0,..., 0x7 is D7, $0 \times 8$ is A0,..., $0 \times \mathrm{FF}$ is A7.


## Freescale Semiconductor, Inc.

## MOVE <br> from MACSR

Move from the
MACSR

MOVE
from MACSR

Operation: $\quad$ MACSR $\rightarrow$ Destination
Assembler Syntax: MOVE.L MACSR,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves the MACSR register contents into a general-purpose register, Rx. $\operatorname{Rx}[31: 12]$ are cleared.

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 21 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |  | Register, Rx |  |

## Instruction Field:

- Register Rx field - Specifies a source register operand, where 0x0 is D0,..., $0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## Move from the <br> MAC MASK Register

MOVE
from MASK

Operation: $\quad$ MASK $\rightarrow$ Destination
Assembler Syntax: MOVE.L MASK,Rx
Attributes: $\quad$ Size $=$ longword
Description: Moves the MASK register contents into a general-purpose register, Rx. $\operatorname{Rx}[31: 16]$ are set to $0 x F F F F$.

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 21 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |  | Register, Rx |  |

Instruction Field:

- Register Rx field - Specifies a source register operand, where $0 x 0$ is $D 0, \ldots, 0 \times 7$ is D7, $0 \times 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## Freescale Semiconductor, Inc.

## Copy an

MOVE

Accumulator
Operation: $\quad$ Source Accumulator $\rightarrow$ Destination Accumulator
Assembler syntax: MOVE.L ACCy,ACCx
Attributes: $\quad$ Size $=$ longword
Description: Moves the 48-bit source accumulator contents and its associated PAV flag into the destination accumulator. This operation is fully pipelined within the EMAC so no pipeline stalls are associated with it. This instruction provides better performance than the two-step process of moving an accumulator to a general-purpose register Rn , then moving Rn into the destination accumulator.

| Condition | N | z | v | PAVx | EV | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Codes | * | * | * | * | * | Z | Set if the result is zero; cleared otherwise |
| (MACSR): |  |  |  |  |  | P PAV | Set if PAV $\mathrm{P}=1$; cleared otherwise Set to the value of the source PAVy flag |
|  |  |  |  |  |  | EV | Set if the source accumulator overflows lower 32 bits in integer mode or lower 40 bits in fractional mode; cleared otherwise |


| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 0 | ACCX | 1 | 0 | 0 | 0 | 1 | 0 | 0 | ACCy |  |  |

## Instruction Fields:

- ACCx—Specifies the destination accumulator. The value of bits [10:9] specify the accumulator number.
- ACCy-Specifies the source accumulator. The value of bits [1:0] specify the accumulator number.


## Freescale Semiconductor, Inc.

# MOVE MACSR to CCR 

## Move from the MACSR to the CCR

Operation: $\quad$ MACSR $\rightarrow$ CCR
Assembler Syntax: MOVE.L MACSR,CCR
Attributes: $\quad$ Size $=$ longword
Description: Moves the MACSR condition codes into the Condition Code Register. The opcode for MOVE MACSR to CCR is $0 \times \mathrm{A} 9 \mathrm{C} 0$.

MACSR: Not affected


Instruction
Format:

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

Operation: $\quad$ Source $\rightarrow$ Accumulator
Assembler syntax: MOVE.L Ry,ACCx
MOVE.L \#<data>,ACCx
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32 -bit value from a register or an immediate operand into an accumulator. If the EMAC is operating in signed integer mode (MACSR[6:5] $=00$ ), the 16 -bit accumulator extension is loaded with the sign-extension of bit 31 of the source operand, while operation in unsigned integer mode (MACSR[6:5] = 10) clears the entire 16 -bit field. If operating in fractional mode (MACSR[5] $=1$, the upper 8 bits of the accumulator extension are loaded with the sign-extension of bit 31 of the source operand, while the low-order 8 -bits of the extension are cleared. The appropriate product/accumulation overflow bit is cleared.

| Condition | N | Z | V | PAV | EV | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Codes | * | * | 0 | 0 | 0 | Z | Set if the result is zero; cleared otherwise |
| (MACSR): |  |  |  |  |  | PAVx | Always cleared |
|  |  |  |  |  |  | EV | Always cleared |



## Instruction Fields:

- ACC—Specifies the destination accumulator. The value of bits [10:9] specify the accumulator number.
- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

# MOVE <br> to ACCext01 

## MOVE <br> to ACCext01

Move to Accumulator 0 and 1 Extensions

Operation: $\quad$ Source $\rightarrow$ Accumulator 0 and 1 extension words
Assembler syntax: MOVE.L Ry,ACCext01
MOVE.L \#<data>,ACCext01
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32-bit value from a register or an immediate operand into the four extension bytes associated with accumulators 0 and 1 . The accumulator extension bytes are loaded as shown in the following table. Note the position of the LSB of the extension within the combined 48 -bit accumulation logic is dependent on the operating mode of the EMAC (integer versus fractional).

| Source Data <br> Bits | Accumulator Extension <br> Affected |
| :---: | :---: |
| $[31: 24]$ | ACCext1[15:8] |
| $[23: 16]$ | ACCext1[7:0] |
| $[15: 8]$ | ACCext0[15:8] |
| $[7: 0]$ | ACCext0[7:0] |

MACSR:
Not affected
Instruction


## Instruction Fields:

- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

# MOVE <br> to ACCext23 

Move to Accumulator

Operation: $\quad$ Source $\rightarrow$ Accumulator 2 and 3 extension words
Assembler syntax: MOVE.L Ry,ACCext23
MOVE.L \#<data>,ACCext23
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32-bit value from a register or an immediate operand into the four extension bytes associated with accumulators 2 and 3. The accumulator extension bytes are loaded as shown in the following table. Note the position of the LSB of the extension within the combined 48-bit accumulation logic is dependent on the operating mode of the EMAC (integer versus fractional).

| Source Data <br> Bits | Accumulator Extension <br> Affected |
| :---: | :---: |
| $[31: 24]$ | ACCext3[15:8] |
| $[23: 16]$ | ACCext3[7:0] |
| $[15: 8]$ | ACCext2[15:8] |
| $[7: 0]$ | ACCext2[7:0] |

MACSR:
Not affected
Instruction

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

## Instruction Fields:

- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

Move to the
MOVE
to MACSR

Operation: $\quad$ Source $\rightarrow$ MACSR
Assembler Syntax: MOVE.L Ry,MACSR
MOVE.L \#<data>,MACSR
Attributes: $\quad$ Size $=$ longword
Description: Moves a 32-bit value from a register or an immediate operand into the MACSR.

MACSR:

Source <ea> bit:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| - | - | - | - | PAV3 | PAV2 | PAV1 | PAVO | OMC | S/U | F/I | R/T | N | Z | V | EV |
| - | - | - | - | [11] | [10] | [9] | [8] | [7] | [6] | [5] | [4] | [3] | [2] | [1] | [0] |

Instruction
Format:


## Instruction Fields:

- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

Operation: $\quad$ Source $\rightarrow$ MASK
Assembler Syntax: MOVE.L Ry,MASK
MOVE.L \#<data>,MASK
Attributes: $\quad$ Size $=$ longword
Description: Moves a 16-bit value from the lower word of a register or an immediate operand into the MASK register.

MACSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regis |  |

## Instruction Fields:

- 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 |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(d_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(d_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(\mathrm{~d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Operation: $\quad$ ACCx $-(R y * R x)\{\ll \mid \gg\} S F \rightarrow A C C x$
Assembler syntax: MSAC.sz Ry.\{U,L\},Rx.\{U,L\}SF,ACCx

## Attributes: $\quad$ Size $=$ word, longword

Description: Multiply two 16- or 32 -bit numbers to yield a 40 -bit result, then subtract this product, shifted as defined by the scale factor, from an accumulator. If 16 -bit operands are used, the upper or lower word of each register must be specified.

| Condition Codes <br> (MACSR): | N | z | v | PAV | EV | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | * | * | * | * | * | z | Set if the result is zero; cleared otherwis |
|  |  |  |  |  |  |  | Set if a product or accumulation overflow is generated or PAV $\mathrm{x}=1$; cleared otherwise |
|  |  |  |  |  |  | PAVx | Set if a product or accumulation overflow is generated; unchanged otherwise |
|  |  |  |  |  |  | EV | Set if accumulation overflows lower 32 bits in integer mode or lower 40 bits in fractional mode; cleared otherwise |

Instruction
Format:

| 15 | 14 | 13 | 12 | 11 | 19 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rx |  | 0 | $\begin{gathered} \mathrm{ACC} \\ \text { Isb } \end{gathered}$ | $\begin{array}{\|c} \hline \mathrm{Rx} \\ \mathrm{msb} \end{array}$ | 0 | 0 | Register, Ry |  |  |  |
| - | - | - | - | sz | Scale <br> Factor | 1 | U/Lx | U/Ly | - | $\begin{aligned} & \mathrm{ACC} \\ & \mathrm{msb} \end{aligned}$ | - | - | - | - |

## Instruction Fields:

- Register Rx[6,11-9] field- Specifies a source register operand, where $0 \times 0$ is D0,..., 0 x 7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A7. Note that bit 6 of the operation word is the msb of the register number field.
- ACC field-Specifies the destination accumulator, ACCx. Bit 4 of the extension word is the msb and bit 7 of the operation word is the lsb. The value of these two bits specify the accumulator number as shown in the following table.

| Ext <br> word [4] | Op word <br> [7] | Accumulator |
| :---: | :---: | :---: |
| 0 | 0 | ACC0 |
| 0 | 1 | ACC1 |
| 1 | 0 | ACC2 |
| 1 | 1 | ACC3 |

- Register Ry[3-0] field - Specifies a source register operand, where 0x0 is D0,..., 0 x 7 is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


# Freescale Semiconductor, Inc. 

## Instruction Fields (continued):

- sz field—Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field - Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product $\ll 1$
- 10 reserved
- 11 product >> 1
- U/Lx-Specifies which 16-bit operand of the source register, Rx , is used for a word-sized operation.
- 0 lower word
- 1 upper word
- U/Ly—Specifies which 16-bit operand of the source register, Ry, is used for a word-sized operation.
- 0 lower word
- 1 upper word

Operation: $\quad \mathrm{ACCx}-(\mathrm{Ry} * \mathrm{Rx})\{\ll \mid \gg\} \mathrm{SF} \rightarrow \mathrm{ACCx}$
(<ea>y) $\rightarrow$ Rw
Assembler syntax: MAC.sz Ry. $\{\mathrm{U}, \mathrm{L}\}, \mathrm{Rx} .\{\mathrm{U}, \mathrm{L}\}$ SF, <ea>y\&,Rw,ACCx where \& enables the use of the MASK

Attributes: $\quad$ Size $=$ word, longword
Description: Multiply two 16- or 32-bit numbers to yield a 40-bit result, then subtract this product, shifted as defined by the scale factor, from an accumulator. If 16-bit operands are used, the upper or lower word of each register must be specified. In parallel with this operation, a 32-bit operand is fetched from the memory location defined by <ea>y and loaded into the destination register, Rw. If the MASK register is specified to be used, the <ea>y operand is ANDed with MASK prior to being used by the instruction.

| Condition Codes <br> (MACSR): | N | Z | V | PAV | EV | N | Set if the msb of the result is set; cleared otherwise |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | * | * | * | * | * | Z | Set if the result is zero; cleared otherwise |
|  |  |  |  |  |  | V | Set if a product or accumulation overflow is generated or PAV $\mathrm{x}=1$; cleared otherwise |
|  |  |  |  |  |  |  | Set if a product or accumulation overflow is generated; unchanged otherwise |
|  |  |  |  |  |  | EV | Set if accumulation overflows lower 32 bits (integer) |



## Instruction Fields:

- Register Rw[6,11-9] field- Specifies the destination register, Rw, where 0x0 is $\mathrm{D} 0, \ldots, 0 \times 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 . Note that bit 6 of the operation word is the msb of the register number field.
- ACC field—Specifies the destination accumulator, ACCx. Bit 4 of the extension word is the msb and bit 7 of the operation word is the inverse of the lsb (unlike the MSAC without load). The value of these two bits specify the accumulator number as shown in the following table:

| Ext word [4] | Op word [7] | Accumulator |
| :---: | :---: | :---: |
| 0 | 1 | ACC0 |
| 0 | 0 | ACC1 |
| 1 | 1 | ACC2 |
| 1 | 0 | ACC3 |

## Instruction Fields (continued):

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

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| Ay | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

- Register Rx field - Specifies a source register operand, where $0 \times 0$ is D0,..., $0 \times 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .
- sz field—Specifies the size of the input operands
- 0 word
- 1 longword
- Scale Factor field -Specifies the scale factor. This field is ignored when using fractional operands.
- 00 none
- 01 product $\ll 1$
- 10 reserved
- 11 product >> 1
- U/Lx, U/Ly—Specifies which 16-bit operand of the source register, $\mathrm{Rx} / \mathrm{Ry}$, is used for a word-sized operation.
- 0 lower word
- 1 upper word
- Mask field - Specifies whether or not to use the MASK register in generating the source effective address, <ea>y.
- 0 do not use MASK
- 1 use MASK
- Register Ry field - Specifies a source register operand, where $0 x 0$ is D0,..., $0 x 7$ is $\mathrm{D} 7,0 \mathrm{x} 8$ is $\mathrm{A} 0, \ldots, 0 \mathrm{xF}$ is A 7 .


## Chapter 7 Floating-Point Unit (FPU) User Instructions

This chapter contains the instruction descriptions implemented in the optional floating-point unit (FPU). Common information on the effects on the floating-point status register (FPSR) and conditional testing has been consolidated in the front of the chapter.

### 7.1 Floating-Point Status Register (FPSR)

The FPSR, Figure 7-1, contains a floating-point condition code byte (FPCC), a floating-point exception status byte (EXC), and a floating-point accrued exception byte (AEXC). The user can read or write all FPSR bits. Execution of most floating-point instructions modifies FPSR.


Figure 7-1. Floating-Point Status Register (FPSR)
Table 7-1 describes FPSR fields.
Table 7-1. FPSR Field Descriptions

| Bits | Field | Description |  |
| :---: | :---: | :---: | :---: |
| 31-24 | FPCC | Floating-point condition code byte. Contains four condition code bits that are set after completion of all arithmetic instructions involving the floating-point data registers. The floating-point store operation, FMOVEM, and move system control register instructions do not affect the FPCC. |  |
| 31-28 |  | Reserved, should be cleared. |  |
| 27 |  | N | Negative |
| 26 |  | Z | Zero |
| 25 |  | I | Infinity |
| 24 |  | NAN | Not-a-number |
| 23-16 | - | Reserved, should be cleared. |  |

Freescale Semiconductor, Inc.

Table 7-1. FPSR Field Descriptions (Continued)

| Bits | Field | Description |  |
| :---: | :---: | :---: | :---: |
| 15-8 | EXC | Exception status byte. Contains a bit for each floating-point exception that might have occurred during the most recent arithmetic instruction or move operation. |  |
| 15 |  | BSUN | Branch/set on unordered |
| 14 |  | INAN | Input not-a-number |
| 13 |  | OPERR | Operand error |
| 12 |  | OVFL | Overflow |
| 11 |  | UNFL | Underflow |
| 10 |  | DZ | Divide by zero |
| 9 |  | INEX | Inexact result |
| 8 |  | IDE | Input is denormalized |
| 7-0 | AEXC | Accrued exception byte. At the end of arithmetic operations, EXC bits are logically combined to form an AEXC value that is logically ORed into the existing AEXC byte (FBcc only updates IOP). This operation creates sticky floating-point exception bits in AEXC that the user can poll only at the end of a series of floating-point operations. A sticky bit is one that remains set until the user clears it. |  |
| 7 |  | IOP | Invalid operation |
| 6 |  | OVFL | Overflow |
| 5 |  | UNFL | Underflow |
| 4 |  | DZ | Divide by zero |
| 3 |  | INEX | Inexact result |
| 2-0 |  | - | Reserved, should be cleared. |

For AEXC[OVFL], AEXC[DZ], and AEXC[INEX], the next value is determined by ORing the current AEXC value with the EXC equivalent, as shown in the following:

- Next AEXC[OVFL] = Current AEXC[OVFL] |EXC[OVFL]
- Next AEXC[DZ] = Current AEXC[DZ] |EXC[DZ]
- Next AEXC[INEX] = Current AEXC[INEX] |EXC[INEX]

For AEXC[IOP] and AEXC[UNFL], the next value is calculated by ORing the current AEXC value with EXC bit combinations, as follows:

- Next AEXC[IOP] = Current AEXC[IOP] | EXC[BSUN | INAN | OPERR]
- Next AEXC[UNFL] = Current AEXC[UNFL]|EXC[UNFL \& INEX]

Table 7-2 shows how the FPSR EXC bits are affected by instruction execution.

Table 7-2. FPSR EXC Bits

| EXC Bit | Description |
| :---: | :---: |
| BSUN | Branch/set on unordered. Set on FBcc if the NAN bit is set and the condition selected is an IEEE nonaware test; cleared otherwise. |
| INAN | Input not-a-number. Set if either input operand is a NAN; cleared otherwise. |
| IDE | Input denormalized number. Set if either input operand is a denormalized number; cleared otherwise. |
| OPERR | Operand error. Set under the following conditions: <br> FADD <br> FDIV $\begin{aligned} & {[(+\infty)+(-\infty)] \text { or }[(-\infty)+(+\infty)]} \\ & (0 \div 0) \text { or }(\infty \div \infty) \end{aligned}$ <br> FMOVE OUT (to B,W,L) Integer overflow, source is NAN or $\pm \infty$ <br> FMUL <br> Source is $<0$ or $-\infty$ <br> FSQRT <br> One operand is 0 and the other is $\pm \infty$ <br> FSUB $[(+\infty)-(+\infty)] \text { or }[(-\infty)-(-\infty)]$ <br> Cleared otherwise. |
| OVFL | Overflow. Set during arithmetic operations if the destination is a floating-point data register or memory when the intermediate result's exponent is greater than or equal to the maximum exponent value of the selected rounding precision. Cleared otherwise. Overflow occurs only when the destination is S- or D-precision format; overflows for other formats are handled as operand errors. |
| UNFL | Underflow. Set if the intermediate result of an arithmetic instruction is too small to be represented as a normalized number in a floating-point register or memory using the selected rounding precision, that is, when the intermediate result exponent is less than or equal to the minimum exponent value of the selected rounding precision. Cleared otherwise. Underflow can only occur when the destination format is single or double precision. When the destination is byte, word, or longword, the conversion underflows to zero without causing an underflow or an operand error. |
| DZ | Set if a FDIV instruction is attempted with a zero divisor; cleared otherwise. |
| INEX | Set under the following conditions: <br> - If the infinitely-precise mantissa of a floating-point intermediate result has more significant bits than can be represented exactly in the selected rounding precision or in the destination format <br> - If an input operand is a denormalized number and the input denorm exception <br> (IDE) is disabled <br> - An overflowed result <br> - An underflowed result with the underflow exception disabled Cleared otherwise. |

### 7.2 Conditional Testing

Unlike operation-dependent integer condition codes, an instruction either always sets FPCC bits in the same way or does not change them at all. Therefore, instruction descriptions do not include FPCC settings. This section describes how FPCC bits are set.

FPCC bits differ slightly from integer condition codes. An FPU operation's final result sets or clears FPCC bits accordingly, independent of the operation itself. Integer condition codes bits CCR[N] and CCR[Z] have this characteristic, but CCR[V] and CCR[C] are set differently for different instructions. Table 7-3 lists FPCC settings for each data type. Loading FPCC with another combination and executing a conditional instruction can produce an unexpected branch condition.

## Freescale Semiconductor, Inc.

Table 7-3. FPCC Encodings

| Data Type | $\mathbf{N}$ | $\mathbf{Z}$ | $\mathbf{I}$ | NAN |
| :--- | :---: | :---: | :---: | :---: |
| + Normalized or Denormalized | 0 | 0 | 0 | 0 |
| - Normalized or Denormalized | 1 | 0 | 0 | 0 |
| +0 | 0 | 1 | 0 | 0 |
| -0 | 1 | 1 | 0 | 0 |
| + Infinity | 0 | 0 | 1 | 0 |
| - Infinity | 1 | 0 | 1 | 0 |
| + NAN | 0 | 0 | 0 | 1 |
| - NAN | 1 | 0 | 0 | 1 |

The inclusion of the NAN data type in the IEEE floating-point number system requires each conditional test to include FPCC[NAN] in its boolean equation. Because it cannot be determined whether a NAN is bigger or smaller than an in-range number (that is, it is unordered), the compare instruction sets FPCC[NAN] when an unordered compare is attempted. All arithmetic instructions that result in a NAN also set the NAN bit. Conditional instructions interpret NAN being set as the unordered condition.

The IEEE-754 standard defines the following four conditions:

- Equal to (EQ)
- Greater than (GT)
- Less than (LT)
- Unordered (UN)

The standard requires only the generation of the condition codes as a result of a floating-point compare operation. The FPU can test for these conditions and 28 others at the end of any operation affecting condition codes. For floating-point conditional branch instructions, the processor logically combines the 4 bits of the FPCC condition codes to form 32 conditional tests, 16 of which cause an exception if an unordered condition is present when the conditional test is attempted (IEEE nonaware tests). The other 16 do not cause an exception (IEEE-aware tests). The set of IEEE nonaware tests is best used in one of the following cases:

- When porting a program from a system that does not support the IEEE standard to a conforming system
- When generating high-level language code that does not support IEEE floating-point concepts (that is, the unordered condition).

An unordered condition occurs when one or both of the operands in a floating-point compare operation is a NAN. The inclusion of the unordered condition in floating-point branches destroys the familiar trichotomy relationship (greater than, equal, less than) that exists for integers. For example, the opposite of floating-point branch greater than (FBGT)

## Freescale Semiconductor, Inc.

is not floating-point branch less than or equal (FBLE). Rather, the opposite condition is floating-point branch not greater than (FBNGT). If the result of the previous instruction was unordered, FBNGT is true; whereas, both FBGT and FBLE would be false because unordered fails both of these tests (and sets BSUN). Because it is common for compilers to invert the sense of conditions, compiler code generators should be particularly careful of the lack of trichotomy in the floating-point branches.

When using the IEEE nonaware tests, the user receives a BSUN exception if a branch is attempted and FPCC[NAN] is set, unless the branch is an FBEQ or an FBNE. If the BSUN exception is enabled in FPCR, the exception takes a BSUN trap. Therefore, the IEEE nonaware program is interrupted if an unexpected condition occurs. Users knowledgeable of the IEEE-754 standard should use IEEE-aware tests in programs that contain ordered and unordered conditions. Because the ordered or unordered attribute is explicitly included in the conditional test, EXC[BSUN] is not set when the unordered condition occurs. Table $7-4$ summarizes conditional mnemonics, definitions, equations, predicates, and whether EXC[BSUN] is set for the 32 floating-point conditional tests. The equation column lists FPCC bit combinations for each test in the form of an equation. Condition codes with an overbar indicate cleared bits; all other bits are set.

Table 7-4. Floating-Point Conditional Tests

| Mnemonic | Definition | Equation | Predicate ${ }^{1}$ | EXC[BSUN] Set |
| :---: | :---: | :---: | :---: | :---: |
| IEEE Nonaware Tests |  |  |  |  |
| EQ | Equal | Z | 000001 | No |
| NE | Not equal | $\bar{Z}$ | 001110 | No |
| GT | Greater than | $\overline{\text { NAN }\|Z\| N}$ | 010010 | Yes |
| NGT | Not greater than | NAN \\| $\mathrm{C} \mid \mathrm{N}$ | 011101 | Yes |
| GE | Greater than or equal | Z \| ( $\overline{\text { AN } \mid N}$ ) | 010011 | Yes |
| NGE | Not greater than or equal | NAN \| ( N \& $\overline{\mathrm{Z}}$ ) | 011100 | Yes |
| LT | Less than | N \& ( $\overline{\text { AN }} \mid \bar{Z})$ | 010100 | Yes |
| NLT | Not less than | NAN \| (Z| $\overline{\mathrm{N}})$ | 011011 | Yes |
| LE | Less than or equal | $\mathrm{Z} \mid$ ( N \& $\overline{\mathrm{NAN}}$ ) | 010101 | Yes |
| NLE | Not less than or equal | NAN \| ( $\overline{\text { \| }} \mathbf{Z}$ ) | 011010 | Yes |
| GL | Greater or less than | $\overline{\text { NAN }}$ \|Z | 010110 | Yes |
| NGL | Not greater or less than | NAN \| Z | 011001 | Yes |
| GLE | Greater, less or equal | $\overline{\text { NAN }}$ | 010111 | Yes |
| NGLE | Not greater, less or equal | NAN | 011000 | Yes |
| IEEE-Aware Tests |  |  |  |  |
| EQ | Equal | Z | 000001 | No |
| NE | Not equal | $\overline{\mathrm{Z}}$ | 001110 | No |
| OGT | Ordered greater than | $\overline{\text { NAN }} \mathrm{Z} \mid \mathrm{N}$ | 000010 | No |

Freescale Semiconductor, Inc. Instruction Results when Exceptions Occur

Table 7-4. Floating-Point Conditional Tests (Continued)

| Mnemonic | Definition | Equation | Predicate ${ }^{1}$ | EXC[BSUN] Set |
| :---: | :---: | :---: | :---: | :---: |
| ULE | Unordered or less or equal | NAN \| Z | N | 001101 | No |
| OGE | Ordered greater than or equal | Z \| ( $\overline{\mathrm{NAN} \mid \mathrm{N}})$ | 000011 | No |
| ULT | Unordered or less than | NAN \| (N \& $\overline{\mathrm{Z}})$ | 001100 | No |
| OLT | Ordered less than | N \& ( $\overline{\mathrm{NAN}} \mid \overline{\mathrm{Z}})$ | 000100 | No |
| UGE | Unordered or greater or equal | NAN \| (Z| $\overline{\mathrm{N}})$ | 001011 | No |
| OLE | Ordered less than or equal | Z \| ( N \& $\overline{\text { NAN }}$ ) | 000101 | No |
| UGT | Unordered or greater than | NAN \| ( $\overline{\mathrm{N} \mid \mathrm{Z}})$ | 001010 | No |
| OGL | Ordered greater or less than | $\overline{\text { NAN \| }} \mathbf{Z}$ | 000110 | No |
| UEQ | Unordered or equal | NAN \| Z | 001001 | No |
| OR | Ordered | $\overline{\text { NAN }}$ | 000111 | No |
| UN | Unordered | NAN | 001000 | No |
| Miscellaneous Tests |  |  |  |  |
| F | False | False | 000000 | No |
| T | True | True | 001111 | No |
| SF | Signaling false | False | 010000 | Yes |
| ST | Signaling true | True | 011111 | Yes |
| SEQ | Signaling equal | Z | 010001 | Yes |
| SNE | Signaling not equal | Z | 011110 | Yes |

### 7.3 Instruction Results when Exceptions Occur

Instruction execution results may be different depending on whether exceptions are enabled in the FPCR, as shown in Table 7-5. An exception is enabled when the value of the EXC bit is 1 , disabled when the value is 0 . Note that if an exception is enabled and occurs on a FMOVE OUT, the destination is unaffected.

Table 7-5. FPCR EXC Byte Exception Enabled/Disabled Results

| EXC Bit | Exception | Description |
| :---: | :---: | :--- |
| BSUN | Disabled | The floating-point condition is evaluated as if it were the equivalent IEEE-aware conditional <br> predicate. No exceptions are taken. |
|  | Enabled | The processor takes a floating-point pre-instruction exception. |
| INAN | Disabled | If the destination data format is single- or double-precision, a NAN is generated with a <br> mantissa of all ones and a sign of zero transferred to the destination. If the destination data <br> format is B, W, or L, a constant of all ones is written to the destination. |
|  | Enabled | The result written to the destination is the same as the exception disabled case unless the <br> exception occurs on a FMOVE OUT, in which case the destination is unaffected. |

## Freescale Semiconductor, Inc.

Table 7-5. FPCR EXC Byte Exception Enabled/Disabled Results (Continued)

| EXC Bit | Exception | Description |
| :---: | :---: | :---: |
| IDE | Disabled | The operand is treated as zero, INEX is set, and processing continues. |
|  | Enabled | If an operand is denormalized, an IDE exception is taken but INEX is not set so that the handler can set INEX appropriately. The destination is overwritten with the same value as if IDE were disabled unless the exception occurred on a FMOVE OUT, in which case the destination is unaffected. |
| OPERR | Disabled | When the destination is a floating-point data register, the result is a double-precision NAN, with its mantissa set to all ones and the sign set to zero (positive). <br> For a FMOVE OUT instruction with the format S or D, an OPERR is impossible. With the format $\mathrm{B}, \mathrm{W}$, or L , an OPERR is possible only on a conversion to integer overflow, or if the source is either an infinity or a NAN. On integer overflow and infinity source cases, the largest positive or negative integer that can fit in the specified destination format (B, W, or L) is stored. In the NAN source case, a constant of all ones is written to the destination. |
|  | Enabled | The result written to the destination is the same as for the exception disabled case unless the exception occurred on a FMOVE OUT, in which case the destination is unaffected. |
| OVFL | Disabled | The values stored in the destination based on the rounding mode defined in FPCR[MODE]. <br> RN Infinity, with the sign of the intermediate result. <br> RZ Largest magnitude number, with the sign of the intermediate result. <br> RM For positive overflow, largest positive normalized number <br> For negative overflow, $-\infty$. <br> RP For positive overflow, $+\infty$ <br> For negative overflow, largest negative normalized number. |
|  | Enabled | The result written to the destination is the same as for the exception disabled case unless the exception occurred on a FMOVE OUT, in which case the destination is unaffected. |
| UNFL | Disabled | The stored result is defined below. UNFL also sets INEX if the UNFL exception is disabled. <br> RN Zero, with the sign of the intermediate result. <br> RZ Zero, with the sign of the intermediate result. <br> RM For positive underflow, +0 <br> For negative underflow, smallest negative normalized number. <br> RP For positive underflow, smallest positive normalized number <br> For negative underflow, - 0 |
|  | Enabled | The result written to the destination is the same as for the exception disabled case unless the exception occurs on a FMOVE OUT, in which case the destination is unaffected. |
| DZ | Disabled | The destination floating-point data register is written with infinity with the sign set to the exclusive OR of the signs of the input operands. |
|  | Enabled | The destination floating-point data register is written as in the exception is disabled case. |
| INEX | Disabled | The result is rounded and then written to the destination. |
|  | Enabled | The result written to the destination is the same as for the exception disabled case unless the exception occurred on a FMOVE OUT, in which case the destination is unaffected. |

### 7.4 Instruction Descriptions

This section describes floating-point instructions in alphabetical order by mnemonic. Operation tables list results for each situation that can be encountered in each instruction. The top and left side of each table represent possible operand inputs, both positive and negative; results are shown in other entries. In most cases, results are floating-point values (numbers, infinities, zeros, or NANs), but for FCMP and FTST, the only result is the setting of condition code bits. When none is stated, no condition code bits are set. Note that if a

## Freescale Semiconductor, Inc.

PC-relative effective address is specified for an FPU instruction, the PC always holds the address of the 16 -bit operation word plus 2.

To understand the results of floating-point instructions under exceptional conditions (overflow, NAN operand, etc.), refer to Table 7-5.

Table 7-6 shows data format encoding used for source data and for destination data for FMOVE register-to-memory operations.

Table 7-6. Data Format Encoding

| Source Data Format | Description |
| :---: | :--- |
| 000 | Longword integer (L) |
| 001 | Single-precision real (S) |
| 100 | Word integer (W) |
| 101 | Double-precision real (D) |
| 110 | Byte integer (B) |

## Freescale Semiconductor, Inc.

Operation: $\quad$ Absolute value of source $\rightarrow \mathrm{FPx}$
Assembler Syntax: FABS.fmt
<ea>y,FPx
FABS.D FPy,FPx
FABS.D FPx
FrABS.fmt <ea>y,FPx
FrABS.D FPy,FPx
FrABS.D FPx
where $r$ is rounding precision, S or D

Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and stores its absolute value in the destination floating-point data register.

FABS rounds the result to the precision selected in FPCR. FSABS and FDABS round to single- or double-precision, respectively, regardless of the rounding precision selected in FPCR.

| Operation Table: | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | + | In Range | + | + | Zero | - | + | Infinity | - |
|  | Result |  | Absolute Value |  |  | Absolute Value |  |  | Absolute Value |  |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
FPSR[FPCC]: See Section 7.2, "Conditional Testing."

| FPSR |  |  |  |  |  |  |  |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| [EXC]: | BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
|  | 0 | See Table 7-2 |  | 0 | 0 | 0 | 0 | 0 |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"


## Instruction fields:

- Source Effective Address field-Determines the addressing mode for external operands.
- If $\mathrm{R} / \mathrm{M}=1$, this field specifies the location of the source operand, <ea>y. Only the addressing modes listed in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $\mathrm{Dy}^{1}$ | 000 | reg. number:Dy |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Only if format is byte, word, longword or single-precision.

- If $R / M=0$, this field is unused and must be all zeros.
- R/M field-Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
-0 : The operation is register to register.
- Source Specifier field-Specifies the source register or data format.

If $R / M=1$, specifies the source data format. Table $7-6$ shows source data format encoding.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination Register field-Specifies the destination floating-point data register, FPx.
- Opmode field-Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0011000 | FABS | Rounding precision specified by FPCR |
| 1011000 | FSABS | Single-precision rounding |
| 1011100 | FDABS | Double-precision rounding |

## Freescale Semiconductor, Inc.

Operation: $\quad$ Source $+\mathrm{FPx} \rightarrow \mathrm{FPx}$
Assembler Syntax: FADD.fmt <ea>y,FPx
FADD.D FPy,FPx
FrADD.fmt <ea>y,FPx
FrADD.D FPy,FPx
where $r$ is rounding precision, $S$ or $D$
Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and adds that number to the number in the destination floating-point data register. Stores the result in the destination floating-point data register.

FADD rounds the result to the precision selected in FPCR. FSADD and FDADD round the result to single- or double-precision, respectively, regardless of the rounding precision selected in FPCR.

| Operation Table: | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | + In Range | - | + | Zero | - | + | Infinity | - |
|  | In Range + | Add |  |  | Add |  | +inf |  | -inf |
|  | Zero + | Add |  | $\begin{aligned} & +0.0 \\ & 0.0^{2} \end{aligned}$ |  | $\begin{gathered} 0.0^{2} \\ -0.0 \end{gathered}$ | +inf |  | -inf |
|  | Infinity $\quad \begin{aligned} & \text { + } \\ & -\end{aligned}$ | $\begin{aligned} & + \text { inf } \\ & -\mathrm{inf} \end{aligned}$ |  |  | $\begin{aligned} & +\mathrm{inf} \\ & -\mathrm{inf} \end{aligned}$ |  | $\begin{aligned} & \text { +inf } \\ & \text { NAN } 3 \end{aligned}$ |  | $\begin{aligned} & \text { NAN }^{3} \\ & \quad-\mathrm{inf} \end{aligned}$ |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
2 Returns +0.0 in rounding modes RN, RZ, and RP; returns -0.0 in RM.
3 Sets the OPERR bit in the FPSR exception byte.
FPSR[FPCC]: See Section 7.2, "Conditional Testing."

| FPSR |  |  |  |  |  |  |  |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| [EXC]: | BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
|  | 0 | See Table 7-2 |  | Set if source and destination <br> are opposite-signed infinities; <br> cleared otherwise. | See Table 7-2 | 0 | See <br> Table 7-2 |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"

## Freescale Semiconductor, Inc.



## Instruction fields:

- Source Effective Address field—Determines the addressing mode.
- If $R / M=1$, this field specifies the location of the source operand, <ea>y. Only the addressing modes listed in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy $^{1}$ | 000 | reg. number:Dy |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

${ }^{1}$ Only if format is byte, word, longword or single-precision.

- If $\mathrm{R} / \mathrm{M}=0$, this field is unused and must be all zeros.
- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0 : The operation is register to register.
- Source Specifier field-Specifies the source register or data format.
- If $R / M=1$, specifies the source data format. See Table 7-6.
- If $\mathrm{R} / \mathrm{M}=0$, specifies the source floating-point data register, FPy.
- Destination Register field-Specifies the destination floating-point register, FPx.
- Opmode field-Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0100010 | FADD | Rounding precision specified by FPCR |
| 1100010 | FSADD | Single-precision rounding |
| 1100110 | FDADD | Double-precision rounding |

Operation: | If Condition True |
| :---: |
| Then PC $+d_{n} \rightarrow P C$ |

Assembler Syntax: FBcc.fmt <label>
Attributes: $\quad$ Format $=$ word, longword
Description: If the specified condition is met, execution continues at (PC) + displacement, a 2's-complement integer that counts relative distance in bytes. The PC value determining the destination is the branch address plus 2 . For word displacement, a 16 -bit value is stored in the word after the instruction operation word. For longword displacement, a 32-bit value is stored in the longword after the instruction operation word. The specifier cc selects a test described in Section 7.2, "Conditional Testing."

FPSR[FPCC]: Not affected.
FPSR [EXC]:

| BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Set it the NAN bit is set and the <br> condition selected is an IEEE <br> nonaware test. | Not affected |  |  |  |  |  |  |

FPSR
[AEXC]:

| IOP | OVFL | UNFL | DZ | INEX |
| :---: | :---: | :---: | :---: | :---: |
| Set if EXC[BSUN] is set. | Not affected |  |  |  |

Instruction

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

## Instruction fields:

- Size field-Specifies the size of the signed displacement.
- If size $=1$, displacement is 32 bits.
- If size $=0$, displacement is 16 bits and is sign-extended before use.
- Conditional predicate field-Specifies a conditional test defined in Table 7-4.


## NOTE:

A BSUN exception causes a pre-instruction exception to be taken. If the handler does not update the stack frame PC image to point to the instruction after FBcc, it must clear the NAN bit or disable the BSUN trap, or the exception recurs on returning.

Operation: $\quad \mathrm{FPx}-$ Source
Assembler Syntax: FCMP.fmt <ea>y,FPx FCMP.D FPy,FPx

Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and subtracts the operand from the destination floating-point data register. The result of the subtraction is not retained but is used to set floating-point condition codes as described in Section 7.2, "Conditional Testing."

Note that if either operand is denormalized, it is treated as zero. Thus, two denormalized operands will compare as equal (set FPCC[Z]) even if they are not identical. This situation can be detected with INEX or IDE.

The entries in this table differ from those for most floating-point instructions. For each combination of input operand types, condition code bits that may be set are indicated. If a condition code bit name is given and is not enclosed in brackets, it is always set. If the name is enclosed in brackets, the bit is set or cleared, as appropriate. If the name is not given, the operation always clears the bit. FCMP always clears the infinity bit because it is not used by any conditional predicate equations.

| Operation | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Table: |  | + | In Range | + | Zero | - | + | Infinity | - |
|  | In Range + | $\left\lvert\, \begin{aligned} & \{N Z\} \\ & N \end{aligned}\right.$ | none \{NZ\} | none |  | none N | $\stackrel{N}{N}$ |  | none none |
|  | Zero + | $\begin{aligned} & N \\ & N \end{aligned}$ | none none | $\mathrm{Z} \mathrm{Z}$ |  | $\mathrm{NZ}$ | $\begin{aligned} & N \\ & N \end{aligned}$ |  | none none |
|  | Infinity + | none <br> N | none | none <br> N |  | none | $\mathrm{Z}$ |  | $\begin{gathered} \text { none } \\ \text { NZ } \end{gathered}$ |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."

## NOTE:

The NAN bit is not shown because NANs are always handled in the same manner (see Section 1.7.1.4, "Not-A-Number).

FPSR[FPCC]: See preceding operation table.

| FPSR |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| [EXC]: |$\quad$| BSUN | INAN | IDE | OPERR | OVFL | UNFL |
| :---: | :---: | :---: | :---: | :---: | :---: | DZ | INEX |
| :---: |
| 0 | See Table 7-2

# Freescale Semiconductor, Inc. 

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
|  | 0 | R/M | 0 | Source Specifier |  |  | Destination Register, FPx |  |  | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

## Instruction fields:

- Effective Address field-Specifies the addressing mode for external operands. If $R / M=1$, this field specifies the location of the source operand, $\langle e a>y$. Only the addressing modes listed in the following table can be used:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy $^{1}$ | 000 | reg. number:Dy |
| Ay | - | - |
| (Ay) | 010 | reg. number:Ay |
| (Ay)+ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(d_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Only if format is byte, word, longword or single-precision.
If $R / M=0$, this field is unused and must be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0 : The operation is register to register.
- Source specifier field-Specifies the source register or data format.

If $\mathrm{R} / \mathrm{M}=1$, specifies the source data format. See Table 7-6.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination register field-Specifies the destination floating-point register, FPx. FCMP does not overwrite the register specified by this field.


## FDIV

Operation: $\quad$ FPx / Source $\rightarrow$ FPx
Assembler Syntax: FDIV.fmt <ea>y,FPx
FDIV.D FPy,FPx
FrDIV.fmt <ea>y,FPx
FrDIV.D FPy,FPx
where $r$ is rounding precision, S or D
Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and divides it into the number in the destination floating-point data register. Stores the result in the destination floating-point data register.

FDIV rounds the result to the precision selected in FPCR. FSDIV and FDDIV round the result to single- or double-precision, respectively, regardless of the rounding precision selected in FPCR.

| Operation Table: | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | In Range | - | + | Zero | - | + | Infinity | - |
|  | In Range + |  | Divide |  | $\begin{aligned} & +\mathrm{inf}^{2} \\ & -\mathrm{inf}^{2} \end{aligned}$ |  | $\begin{aligned} & -\mathrm{inf}^{2} \\ & +\mathrm{inf}^{2} \end{aligned}$ | $\left\lvert\, \begin{aligned} & +0.0 \\ & -0.0 \end{aligned}\right.$ |  | $\begin{aligned} & -0.0 \\ & +0.0 \end{aligned}$ |
|  | Zero + | $\begin{aligned} & +0.0 \\ & -0.0 \end{aligned}$ |  | -0.0 +0.0 |  | NAN ${ }^{3}$ |  | $\left\lvert\, \begin{aligned} & +0.0 \\ & -0.0 \end{aligned}\right.$ |  | $\begin{gathered} -0.0 \\ +0.0 \end{gathered}$ |
|  | Infinity $\quad+$ | $+\inf _{\text {inf }}$ |  | $\begin{aligned} & \text {-inf } \\ & + \text { inf } \end{aligned}$ | $\begin{aligned} & +\mathrm{inf} \\ & -\mathrm{inf} \end{aligned}$ |  | $\begin{gathered} \hline-\mathrm{inf} \\ +\mathrm{inf} \end{gathered}$ |  | NAN ${ }^{3}$ |  |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
2 Sets the DZ bit in the FPSR exception byte.
3 Sets the OPERR bit in the FPSR exception byte.
FPSR[FPCC]: See Section 7.2, "Conditional Testing."
FPSR [EXC]:

| BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| :---: | :--- | :---: | :---: | :---: | :---: | :--- | :--- |
| 0 | See Table 7-2 | Set for $0 \div 0$ <br> or $\infty \div \infty ;$ <br> cleared <br> otherwise. | See Table 7-2 | Set if source is 0 <br> and destination is in <br> range; cleared <br> otherwise. | See <br> Table 7-2 |  |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"

## Freescale Semiconductor, Inc.

## Instruction fields:

- Effective Address field—Specifies the addressing mode for external operands.

If $R / M=1$, this field specifies the location of the source operand, <ea>y. Only the addressing modes listed in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy $^{1}$ | 000 | reg. number:Dy |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(d_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Only if format is byte, word, longword or single-precision.
If $\mathrm{R} / \mathrm{M}=0$, this field is unused and must be all zeros.

- $\mathrm{R} / \mathrm{M}$ field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0 : The operation is register to register.
- Source specifier field—Specifies the source register or data format.

If $\mathrm{R} / \mathrm{M}=1$, specifies the source data format. See Table 7-6.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination register field—Specifies the destination floating-point register, FPx.
- Opmode field-Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0100000 | FDIV | Rounding precision specified by FPCR |
| 1100000 | FSDIV | Single-precision rounding |
| 1100100 | FDDIV | Double-precision rounding |

Operation: $\quad$ Integer Part of Source $\rightarrow \mathrm{FPx}$
Assembler Syntax: FINT.fmt <ea>y,FPx
FINT.D FPy,FPx
FINT.D FPx
Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary), extracts the integer part, and converts it to a double-precision value. Stores the result in the destination floating-point data register. The integer part is extracted by rounding the double-precision number to an integer using the current rounding mode selected in the FPCR mode control byte. Thus, the integer part returned is the number to the left of the radix point when the exponent is zero after rounding. For example, the integer part of 137.57 is 137.0 for round-to-zero and round-to-negative infinity modes and 138.0 for round-to-nearest and round-to-positive infinity modes. Note that the result of this operation is a floating-point number.

| Operation <br> Table: | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  | + | In Range |  | Zero |  | + | Infinity | - |
|  | Result |  | Integer | +0.0 |  | -0.0 | +inf |  | -inf |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
FPSR[FPCC]: See Section 7.2, "Conditional Testing."

| FPSR | BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| [EXC]: | 0 | See Table 7-2 | 0 | 0 | 0 | 0 | See Table 7-2 |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
|  | 0 | R/M | 0 | Source Specifier |  |  | Destination Register, FPx |  |  | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Floating-Point Integer

## Instruction fields:

- Source Effective Address field-Determines the addressing mode for external operands.

If $R / M=1$, this field specifies the location of the source operand <ea>y. Only the addressing modes the following table can be used.

| Addressing Mode | Mode | Register | Addressing Mode | Mode | Register |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Dy ${ }^{1}$ | 000 | reg. number:Dy | ( $\mathrm{x} x \mathrm{x}$ ).W | - | - |
| Ay | - | - | (xxx).L | - | - |
| (Ay) | 010 | reg. number:Ay | \# <data> | - | - |
| (Ay)+ | 011 | reg. number:Ay |  |  |  |
| -(Ay) | 100 | reg. number:Ay |  |  |  |
| ( $\mathrm{d}_{16}, \mathrm{Ay}$ ) | 101 | reg. number:Ay | ( $\left.\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| ( $\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}$ ) | - | - | ( $\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}$ ) | - | - |

Only if format is byte, word, longword or single-precision.
If $R / M=0$, this field is unused and should be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
-0 : The operation is register to register.
- Source specifier field-Specifies the source register or data format.

If $R / M=1$, specifies the source data format. See Table 7-6.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination register field-Specifies the destination floating-point register, FPx.

If $R / M=0$ and the source and destination fields are equal, the input operand is taken from the specified floating-point data register, and the result is written into the same register. If the single register syntax is used, Motorola assemblers set the source and destination fields to the same value.

## FINTRZ

Floating-Point Integer Round-to-Zero

Operation: $\quad$ Integer Part of Source $\rightarrow \mathrm{FPx}$
Assembler Syntax: FINTRZ.fmt <ea>y,FPx
FINTRZ.D FPy,FPx
FINTRZ.D FPx
Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and extracts the integer part and converts it to a double-precision number. Stores the result in the destination floating-point data register. The integer part is extracted by rounding the double-precision number to an integer using the round-to-zero mode, regardless of the rounding mode selected in the FPCR mode control byte (making it useful for FORTRAN assignments). Thus, the integer part returned is the number that is to the left of the radix point when the exponent is zero. For example, the integer part of 137.57 is 137.0 . Note the result of this operation is a floating-point number.

Operation
Table:

| Destination | Source $^{\mathbf{1}}$ |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :--- | :---: | :--- | :--- | :--- | :--- |
|  | + | In Range | - | + | Zero | - | Infinity | - |
| Result | Integer, Forced to <br> Round to Zero | +0.0 |  | -0.0 | + inf |  | - inf |  |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
FPSR[FPCC]: See Section 7.2, "Conditional Testing."

| FPSR | BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| [EXC]: | 0 | See Table 7-2 | 0 | 0 | 0 | 0 | See Table 7-2 |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
|  | 0 | R/M | 0 | Source Specifier |  |  | Destination <br> Register, FPx |  |  | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

## Freescale Semiconductor, Inc.

## FINTRZ Floating-Point Integer Round-to-Zero

## Instruction fields:

- Effective Address field—Determines the addressing mode for external operands.

If $\mathrm{R} / \mathrm{M}=1$, this field specifies the location of the source operand, <ea>y. Only the addressing modes listed in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $D^{1}$ | 000 | reg. number:Dy |
| $A y$ | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

${ }^{1}$ Only if format is byte, word, longword or single-precision.
If $R / M=0$, this field is unused and should be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0 : The operation is register to register.
- Source specifier field-Specifies the source register or data format.

If $R M=1$, specifies the source data format. See Table 7-6.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination register field—Specifies the destination floating-point register, FPx. If $R / M=0$ and the source and destination fields are equal, the input operand is taken from the specified floating-point data register and the result is written into the same register. If the single register syntax is used, Motorola assemblers set the source and destination fields to the same value.


# FMOVE 

Operation: $\quad$ Source $\rightarrow$ Destination

Assembler Syntax: FMOVE.fmt <ea>y,FPx<br>FMOVE.fmt FPy,<ea>x<br>FMOVE.D FPy,FPx<br>FrMOVE.fmt <ea>y, FPx<br>FrMOVE.D FPy, FPx<br>where r is rounding precision, S or D

Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Moves the contents of the source operand to the destination operand. Although the primary function of FMOVE is data movement, it is considered an arithmetic instruction because conversions from the source operand format to the destination operand format occur implicitly. Also, the source operand is rounded according to the selected rounding precision and mode.

Unlike MOVE, FMOVE does not support a memory-to-memory format. For such transfers, MOVE is much faster than FMOVE to transfer floating-point data. FMOVE supports memory-to-register, register-to-register, and register-to-memory operations (memory here can include an integer data register if the format is byte, word, longword, or single-precision). Memory- and register-to-register operations use a command word encoding different from that used by the register-to-memory operation; these two operation classes are described separately.

Memory- and register-to-register operations (<ea>y,FPx; FPy,FPx): Converts the source operand to a double-precision number (if necessary) and stores it in the destination floating-point data register, FPx. FMOVE rounds the result to the precision selected in FPCR. FSMOVE and FDMOVE round the result to single- and double-precision, regardless of the rounding selected in FPCR. Note that if the source format is longword or double precision, inexact results may be created when rounding to single precision. All other combinations of source formats and rounding precision produce an exact result.

FPSR[FPCC]: See Section 7.2, "Conditional Testing."
FPSR
[EXC]:

| BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | See Table 7-2 |  | 0 | 0 | 0 | 0 | See Table 7-2 |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"

## Freescale Semiconductor, Inc.



Instruction fields:

- Effective address field—Determines the addressing mode for external operands. If $\mathrm{R} / \mathrm{M}=1$, this field specifies the location of the source operand. Only the addressing modes listed in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy $^{1}$ | 000 | reg. number:Dy |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(d_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Only if format is byte, word, longword or single-precision.
If $\mathrm{R} / \mathrm{M}=0$, this field is unused and must be all zeros.

- $\mathrm{R} / \mathrm{M}$ field—Specifies the source operand address mode.

If $R / M=0$ the operation is register to register.
If $R / M=1$ the operation is <ea>y to register.

- Source specifier field-Specifies the source register or data format.

If $R / M=0$, specifies the source floating-point data register, FPy.
If $R / M=1$, specifies the source data format. See Table 7-6.

- Destination register field—Specifies the destination floating-point register, FPx.
- Opmode field-Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0000000 | FMOVE | Rounding precision specified by the FPCR |
| 1000000 | FSMOVE | Single-precision rounding |
| 1000100 | FDMOVE | Double-precision rounding |

## FMOVE

Move Floating-Point Data Register
FMOVE

Register-to-memory operation ( $\mathbf{F P y},<\mathbf{e a}>\mathbf{x}$ ): Rounds the source operand to the specified destination format and stores it at the destination effective address, <ea>x. Note that the rounding mode in FPCR is ignored for register-to-memory operations.

FPSR[FPCC]: Not affected.

| FPSR <br> [EXC]: <br> format $=. \mathrm{B}, . \mathrm{W}$, <br> or .L | BSUN | InAN | IDE | OPERR | OVFL | UNFL | Dz | INEX |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 0 | See Table 7-2 |  | Set if source operand is $\infty$ or if destination size is exceeded after conversion and rounding | 0 | 0 | 0 | See <br> Table 7-2 |
| $\begin{aligned} & \text { format }=. S \text { or } \\ & . D \end{aligned}$ |  |  |  | 0 |  | 7-2 | 0 |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"
Instruction Format FPy,<ea>x:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
| 0 | 1 | 1 | Destination Format |  |  | Source Register, FPy |  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Instruction fields:

- Destination Effective Address field-Specifies the destination location, <ea>x. Only modes in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $\mathrm{Dx}^{1}$ | 000 | reg. number:Dx |
| Ax | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

1 Only if format is byte, word, longword or single-precision.

- Destination Format field—Specifies the data format of the destination operand. See Table 7-6.
- Source Register field—Specifies the source floating-point data register, FPy.


# FMOVE <br> from FPCR 

 Point Control RegisterOperation: $\quad$ FPCR $\rightarrow$ Destination

## Assembler syntax: FMOVE.L FPCR,<ea>x

Attributes: $\quad$ Format $=$ longword
Description: Moves the contents of the FPCR to an effective address. A 32-bit transfer is always performed, even though the FPCR does not have 32 implemented bits. Unimplemented bits of a control register are read as zeros. Exceptions are not taken upon execution of this instruction.

FPSR: Not affected

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

## Instruction field:

- Effective Address field-Specifies the addressing mode, <ea>x, shown in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | 000 | reg. number:Dx |
| Ax | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

# FMOVE from FPIAR Move from the Floating from FPIAR Point Instruction Address Register 

Operation: $\quad$ FPIAR $\rightarrow$ Destination

Assembler syntax: FMOVE.L FPIAR, <ea>x
Attributes: $\quad$ Format $=$ longword
Description: Moves the contents of the floating-point instruction address register to an effective address. Exceptions are not taken upon execution of this instruction.

FPSR: Not affected

| Instruction | 15 | 14 | 4 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 |  | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mod |  |  | gis |  |
|  | 1 | 0 | 0 |  | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Instruction field:

- Effective Address field-Specifies the addressing mode, <ea>x, shown in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | 000 | reg. number:Dx |
| Ax | 001 | reg. number:Ax |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})_{+}$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(d_{8}, A x, X i\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

## FMOVE <br> from FPSR

Operation: $\quad$ FPSR $\rightarrow$ Destination
Assembler syntax: FMOVE.L FPSR, <ea>x
Attributes: $\quad$ Format $=$ longword
Description: Moves the contents of the FPCR to an effective address. A 32-bit transfer is always performed, even though the FPSR does not have 32 implemented bits. Unimplemented bits of a control register are read as zeros. Exceptions are not taken upon execution of this instruction.

FPSR: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  | Mod |  |  | gis |  |
|  | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Instruction field:

- Effective Address field-Specifies the addressing mode, $\langle e a>x$, shown in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | 000 | reg. number:Dx |
| $A x$ | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | 011 | reg. number:Ax |
| $-(\mathrm{Ax})$ | 100 | reg. number:Ax |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## FMOVE <br> to FPCR

## FMOVE to FPCR

## Move to the Floating

Operation: $\quad$ Source $\rightarrow$ FPCR

## Assembler syntax: FMOVE.L <ea>y,FPCR

Attributes: $\quad$ Format $=$ longword
Description: Loads the FPCR from an effective address. A 32-bit transfer is always performed, even though the FPCR does not have 32 implemented bits. Unimplemented bits are ignored during writes (must be zero for compatibility with future devices). Exceptions are not taken upon execution of this instruction.

FPSR: Not affected.

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
|  | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Instruction field:

- Effective Address field-Specifies the addressing mode, <ea>y, shown in the following table:

| Addressing Mode | Mode | Register | Addressing Mode | Mode | Register |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Dy | 000 | reg. number:Dy | (xxx).W | - | - |
| Ay | - | - | (xxx).L | - | - |
| (Ay) | 010 | reg. number:Ay | \# <data> | - | - |
| (Ay)+ | 011 | reg. number:Ay |  |  |  |
| -(Ay) | 100 | reg. number:Ay |  |  |  |
| ( $\mathrm{d}_{16}, \mathrm{Ay}$ ) | 101 | reg. number:Ay | ( $\left.\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| ( $\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}$ ) | - | - | ( $\left.\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

## FMOVE <br> to FPIAR

Move to the Floating
FMOVE to FPIAR Point Instruction Address Register

Operation: $\quad$ Source $\rightarrow$ FPIAR
Assembler syntax: FMOVE.L <ea>y,FPIAR
Attributes: $\quad$ Format $=$ longword
Description: Loads the floating-point instruction address register from an effective address. Exceptions are not taken upon execution of this instruction.

FPSR: Not affected.


## Instruction field:

- Effective Address field-Specifies the addressing mode, <ea>y, shown in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | 000 | reg. number:Dy |
| Ay | 001 | reg. number:Ay |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## FMOVE to FPSR

Operation: $\quad$ Source $\rightarrow$ FPSR

## Assembler syntax: FMOVE.L <ea>y,FPSR

Attributes: $\quad$ Format $=$ longword
Description: Loads the FPSR from an effective address. A 32-bit transfer is always performed, even though the FPSR does not have 32 implemented bits. Unimplemented bits are ignored during writes (must be zero for compatibility with future devices). Exceptions are not taken upon execution of this instruction.

FPSR:
All bits are modified to reflect the source operand value.

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
|  | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Instruction field:

- Effective Address field-Specifies the addressing mode, <ea>y, shown in the following table:

| Addressing Mode | Mode | Register | Addressing Mode | Mode | Register |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Dy | 000 | reg. number:Dy | ( xxx ).W | - | - |
| Ay | - | - | (xxx).L | - | - |
| (Ay) | 010 | reg. number:Ay | \# <data> | - | - |
| (Ay)+ | 011 | reg. number:Ay |  |  |  |
| -(Ay) | 100 | reg. number:Ay |  |  |  |
| ( $\mathrm{d}_{16}, \mathrm{Ay}$ ) | 101 | reg. number:Ay | ( $\left.\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| ( $\left.\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - | ( $\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}$ ) | - | - |

# FMOVEM 

Move Multiple Floating-Point Data Registers

Operation: $\quad$ Register List $\rightarrow$ Destination Source $\rightarrow$ Register List<br>Assembler syntax: FMOVEM.D \#list, <ea>x<br>FMOVEM.D <ea>y,\#list

Attributes: $\quad$ Format $=$ double-precision
Description: Moves one or more double-precision numbers to or from a list of floating-point data registers. No conversion or rounding is performed during this operation, and the FPSR is not affected by the instruction. Exceptions are not taken upon execution of this instruction. Any combination of the eight floating-point data registers can be transferred, with selected registers specified by a user-supplied mask. This mask is an 8-bit number, where each bit corresponds to one register; if a bit is set in the mask, that register is moved. Note that a null register list (all zeros) generates a line F exception.

FMOVEM allows two addressing modes: address register indirect and base register plus 16-bit displacement, where the base is an address register, or for loads only, the program counter. In all cases, the processor calculates the starting address and then increments by 8 bytes for each register moved. The transfer order is always FP0-FP7.

## NOTE:

FMOVEM offers the only way to move floating-point data between the FPU and memory without converting data or affecting condition code and exception status bits.

FPSR: Not affected.

| Instruction Format: | 15 | 14 |  | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |  | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 |  | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |  | Effective Address |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | Mode |  | Register |  |
|  | 1 |  | 1 | dr | 1 | 0 | 0 | 0 | 0 | Register List |  |  |  |  |  |  |

## FMOVEM Move Multiple Floating-Point Data Registers FMOVEM

## Instruction fields:

- Effective address field—Specifies the addressing mode. For memory-to-register the allowed $\langle\mathrm{e} a\rangle$ y modes are shown in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| Ay | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

- Effective address field—Specifies the addressing mode. For register-to-memory the allowed $\langle\mathrm{e} \mathrm{a}\rangle \mathrm{x}$ modes are shown in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | - | - |
| Ax | - | - |
| $(\mathrm{Ax})$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})_{+}$ | - | - |
| $-(\mathrm{Ax})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

- dr field—Specifies the direction of the transfer.
- 0: Move the listed registers from memory to the FPU.
- 1: Move the listed registers from the FPU to memory.
- Register list field-Contains the register select mask. If a register is to be moved, the corresponding mask bit is set as shown below; otherwise it is zero.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| FP0 | FP1 | FP2 | FP3 | FP4 | FP5 | FP6 | FP7 |

Operation: $\quad$ Source * FPx $\rightarrow$ FPx
Assembler syntax: FMUL.fmt <ea>y,FPx
FMUL.D FPy,FPx
FrMUL.fmt <ea>y,FPx
FrMUL.D FPy,FPx where $r$ is rounding precision, $S$ or $D$

Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision Description: Converts source operand to double-precision (if necessary) and multiplies that number by the number in destination floating-point data register. Stores result in the destination floating-point data register.

FMUL rounds the result to the precision selected in FPCR. FSMUL and FDMUL round the result to single- or double-precision, respectively, regardless of the rounding precision selected in FPCR.

| Operation Table: | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  |  |  | In Range | - | + | Zero | - | + | Infinity | - |
|  | In Range + |  | Multiply |  | $\begin{aligned} & +0.0 \\ & -0.0 \end{aligned}$ |  | $\begin{aligned} & -0.0 \\ & +0.0 \end{aligned}$ | $\begin{aligned} & + \text { +inf } \\ & -i n f \end{aligned}$ |  | $\begin{gathered} \text {-inf } \\ +\mathrm{inf} \end{gathered}$ |
|  | Zero | +0.0 -0.0 |  | -0.0 +0.0 | $\begin{aligned} & +0.0 \\ & -0.0 \end{aligned}$ |  | $\begin{aligned} & -0.0 \\ & +0.0 \end{aligned}$ |  | NAN ${ }^{2}$ |  |
|  | Infinity + | +inf |  | -inf <br> + inf |  | NAN ${ }^{2}$ |  | $+\operatorname{linf}_{\text {inf }}$ |  | $\begin{aligned} & \text {-inf } \\ & + \text { inf } \end{aligned}$ |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
2 Sets the OPERR bit in the FPSR exception byte.
FPSR[FPCC]: See Section 7.2, "Conditional Testing."
FPSR [EXC]:

| BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| INEX |  |  |  |  |  |  |
| 0 | See Table 7-2 | Set for $0 \mathrm{x} \infty ;$ cleared <br> otherwise. | See Table 7-2 | 0 | See <br> Table 7-2 |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"


## FMUL

## Instruction fields:

- Effective address field—Determines the addressing mode for external operands. If $\mathrm{R} / \mathrm{M}=1$, this field specifies the location of the source operand. Only the addressing modes listed in the following table can be used.

| Addressing Mode | Mode | Register | Addressing Mode | Mode | Register |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Dy ${ }^{1}$ | 000 | reg. number:Dy | ( $x$ xx).W | - | - |
| Ay | - | - | (xxx).L | - | - |
| (Ay) | 010 | reg. number:Ay | \# <data> | - | - |
| (Ay)+ | 011 | reg. number:Ay |  |  |  |
| -(Ay) | 100 | reg. number:Ay |  |  |  |
| ( $\mathrm{d}_{16}, \mathrm{Ay}$ ) | 101 | reg. number:Ay | $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| ( $\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}$ ) | - | - | ( $\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}$ ) | - | - |

Only if format is byte, word, longword or single-precision.
If $R / M=0$, this field is unused and must be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0 : The operation is register to register.
- Source specifier field—Specifies the source register or data format.

If $\mathrm{R} / \mathrm{M}=1$, specifies the source data format. See Table 7-6.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination register field—Specifies the destination floating-point register, FPx.
- Opmode field—Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0100011 | FMUL | Rounding precision specified by the FPCR |
| 1100011 | FSMUL | Single-precision rounding |
| 1100111 | FDMUL | Double-precision rounding |

## Freescale Semiconductor, Inc.

Floating-Point Negate

Operation: $\quad-($ Source $) \rightarrow \mathrm{FPx}$
Assembler syntax: FNEG.fmt <ea>y,FPx
FNEG.D FPy,FPx
FNEG.D FPx
FrNEG.fmt <ea>y,FPx
FrNEG.D FPy,FPx
FrNEG.D FPx
where r is rounding precision, S or D
Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and inverts the sign of the mantissa. Stores the result in the destination floating-point data register, FPx.

FNEG rounds the result to the precision selected in the FPCR. FSNEG and FDNEG round the result to single- or double-precision, respectively, regardless of the rounding precision selected in the FPCR.

Operation

| Destination | Source $^{\mathbf{1}}$ |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | ---: | ---: |
|  | + | In Range | $-\quad+$ | Zero | - | + | Infinity | - |
| Result | Negate | -0.0 |  | +0.0 | - inf |  | + inf |  |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
FPSR[FPCC]: See Section 7.2, "Conditional Testing."
FPSR
[EXC]:

| BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | See Table 7-2 | 0 | 0 | 0 | 0 | 0 |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"


## Instruction fields:

- Effective Address field—Determines the addressing mode for external operands.

If $R / M=1$, this field specifies the location of the source operand. Only modes in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy $^{1}$ | 000 | reg. number:Dy |
| Ay | - | - |
| (Ay) | 010 | reg. number:Ay |
| (Ay)+ | 011 | reg. number:Ay |
| $-(\mathrm{Ay})$ | 100 | reg. number:Ay |
| $\left(d_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Only if format is byte, word, longword or single-precision.
If $R / M=0$, this field is unused and must be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0 : The operation is register to register.
- Source specifier field-Specifies the source register or data format.

If $\mathrm{R} / \mathrm{M}=1$, specifies the source data format. See Table 7-6.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination register field—Specifies the destination floating-point register, FPx. If $R / M=0$ and the source and destination fields are equal, the input operand is taken from the specified floating-point data register and the result is written into the same register. If the single register syntax is used, Motorola assemblers set the source and destination fields to the same value.
- Opmode field—Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0011010 | FNEG | Rounding precision specified by the FPCR |
| 1011010 | FSNEG | Single-precision rounding |
| 1011110 | FDNEG | Double-precision rounding |

Operation: None

## Assembler syntax: FNOP

Attributes: Unsized
Description: FNOP performs no explicit operation. It is used to synchronize the FPU with an integer unit or to force processing of pending exceptions. For most floating-point instructions, the integer unit can continue executing the next instruction once the FPU has any operands needed for an operation, thus supporting concurrent execution of integer and floating-point instructions. FNOP causes the integer unit to wait for all previous floating-point instructions to complete. It also forces any exceptions pending from the execution of a previous floating-point instruction to be processed as a pre-instruction exception. The opcode for FNOP is $0 x F 2800000$.

FPSR: Not affected.

| Instruction Format: | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

NOTE:
FNOP uses the same opcode as the FBcc.W <label> instruction, with $\mathrm{cc}=\mathrm{F}$ (nontrapping false) and <label> $=+2$ (which results in a displacement of 0 ).

Operation: $\quad$ Square Root of Source $\rightarrow \mathrm{FPx}$
Assembler syntax: FSQRT.fmt <ea>y,FPx
FSQRT.D FPy,FPx
FSQRT.D FPx
FrSQRT.fmt <ea>y,FPx
FrSQRT.D FPy,FPx
FrSQRT.D FPx
where $r$ is rounding precision, S or D
Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and calculates the square root of that number. Stores the result in the destination floating-point data register, FPx. This function is not defined for negative operands.

FSQRT rounds the result to the precision selected in the FPCR. FSFSQRT and FDFSQRT round the result to single- or double-precision, respectively, regardless of the rounding precision selected in the FPCR.

| Operation | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Table: |  |  | In Range |  | Zero | - | + | Infinity | - |
|  | Result | $\sqrt{x}$ | NAN ${ }^{2}$ | +0.0 |  | -0.0 | +inf |  | NAN ${ }^{2}$ |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
2 Sets the OPERR bit in the FPSR exception byte.
FPSR[FPCC]: See Section 7.2, "Conditional Testing."

| FPSR |  |  |  |  |  |  |  |  |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| [EXC]: | BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| 0 | See Table 7-2 | Set if the source operand is <br> not 0 and is negative; cleared <br> otherwise. | 0 | 0 | 0 | See <br> Table 7-2 |  |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"


## FSQRT

## Floating-Point Square Root

## Instruction fields:

- Effective address field—Specifies the addressing mode for external operands. If $R / M=1$, this field specifies the location of the source operand, <ea>y. Only modes in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $D^{1}$ | 000 | reg. number:Dy |
| $A y$ | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

Only if format is byte, word, longword or single-precision.
If $\mathrm{R} / \mathrm{M}=0$, this field is unused and must be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0: The operation is register to register.
- Source specifier field-Specifies the source register or data format.

If $\mathrm{R} / \mathrm{M}=1$, specifies the source data format. See Table 7-6.
If $\mathrm{R} / \mathrm{M}=0$, specifies the source floating-point data register, FPy.

- Destination register field—Specifies the destination floating-point register, FPx. If $R / M=0$ and source and destination fields are equal, the input operand comes from the specified floating-point data register, and the result is written into the same register. If single register syntax is used, Motorola assemblers set the source and destination fields to the same value.
- Opmode field—Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0000100 | FSQRT | Rounding precision specified by the FPCR |
| 1000001 | FSSQRT | Single-precision rounding |
| 1000101 | FDSQRT | Double-precision rounding |

Operation: $\quad \mathrm{FPx}-$ Source $\rightarrow \mathrm{FPx}$
Assembler syntax: FSUB.fmt <ea>y,FPx
FSUB.D FPy,FPx
FrSUB.fmt <ea>y,FPx
FrSUB.D FPy,FPx where $r$ is rounding precision, $S$ or $D$

Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and subtracts it from the number in the destination floating-point data register. Stores the result in the destination floating-point data register.


1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
2 Returns +0.0 in rounding modes RN, RZ, and RP; returns -0.0 in RM.
3 Sets the OPERR bit in the FPSR exception byte.
FPSR[FPCC]: See Section 7.2, "Conditional Testing."

| FPSR |
| :--- | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| [EXC]: |$\quad$| BSUN | INAN | IDE | OPERR | OVFL |
| :---: | :---: | :---: | :---: | :---: |
| 0 | See Table 7-2 |  | Set if source and destination <br> are like-signed infinities; <br> cleared otherwise. | See Table 7-2 |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)."


## FSUB

## Instruction fields:

- Effective address field—Determines the addressing mode for external operands. If $R / M=1$, this field specifies the location of the source operand, <ea>y. Only the addressing modes listed in the following table can be used.

| Addressing Mode | Mode | Register | Addressing Mode | Mode | Register |
| :---: | :---: | :---: | :---: | :---: | :---: |
| Dy ${ }^{1}$ | 000 | reg. number:Dy | ( xxx ).W | - | - |
| Ay | - | - | (xxx).L | - | - |
| (Ay) | 010 | reg. number:Ay | \# <data> | - | - |
| (Ay)+ | 011 | reg. number:Ay |  |  |  |
| -(Ay) | 100 | reg. number:Ay |  |  |  |
| ( $\mathrm{d}_{16}, \mathrm{Ay}$ ) | 101 | reg. number:Ay | $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| ( $\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}$ ) | - | - | ( $\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}$ ) | - | - |

Only if format is byte, word, longword or single-precision.
If $R / M=0$, this field is unused and must be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is $\langle\mathrm{ea}\rangle \mathrm{y}$ to register.
- 0 : The operation is register to register.
- Source Specifier field—Specifies the source register or data format.

If $\mathrm{R} / \mathrm{M}=1$, specifies the source data format. See Table 7-6.
If $R / M=0$, specifies the source floating-point data register, FPy.

- Destination register field—Specifies the destination floating-point register, FPx.
- Opmode field—Specifies the instruction and rounding precision.

| Opmode | Instruction | Rounding Precision |
| :---: | :---: | :---: |
| 0101000 | FSUB | Rounding precision specified by the FPCR |
| 1101000 | FSSUB | Single-precision rounding |
| 1101100 | FDSUB | Double-precision rounding |

motorola

Test Floating-Point Operand

Operation: $\quad$ Source Operand Tested $\rightarrow$ FPCC

## Assembler syntax: FTST.fmt <ea>y FTST.D FPy

Attributes: $\quad$ Format $=$ byte, word, longword, single-precision, double-precision
Description: Converts the source operand to double-precision (if necessary) and sets the condition code bits according to the data type of the result. Note that for denormalized operands, FPCC[Z] is set because denormalized numbers are normally treated as zero. When Z is set, INEX is set if the operand is a denormalized number (and IDE is disabled). INEX is cleared if the operand is exactly zero.

| Operation | Destination | Source ${ }^{1}$ |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Table: |  | + | In Range | - | + | Zero | - | + | Infinity | - |
|  | Result | none |  | N | Z |  | NZ | I |  | NI |

1 If the source operand is a NAN, refer to Section 1.7.1.4, "Not-A-Number."
Note that the operation table differs from other operation tables. A letter in a table entry indicates that FTST always sets the designated condition code bit. All unspecified condition code bits are cleared during the operation.

FPSR[FPCC]: See Section 7.2, "Conditional Testing."
FPSR [EXC]:

| BSUN | INAN | IDE | OPERR | OVFL | UNFL | DZ | INEX |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | See Table 7-2 | 0 | 0 | 0 | 0 | Set if denormalized and IDE is <br> disabled; cleared otherwise |  |

FPSR[AEXC]: See Section 7.1, "Floating-Point Status Register (FPSR)"


## FTST

## Test Floating-Point Operand

## Instruction fields:

- Effective address field—Determines the addressing mode for external operands. If $\mathrm{R} / \mathrm{M}=1$, this field specifies the source operand location, <ea>y. Only modes in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy $^{1}$ | 000 | reg. number:Dy |
| Ay | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | 011 | reg. number:Ay |
| $-(A y)$ | 100 | reg. number:Ay |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{xi}\right)$ | - | - |

Only if format is byte, word, longword or single-precision.
If $R / M=0$, this field is unused and must be all zeros.

- R/M field—Specifies the source operand address mode.
- 1: The operation is <ea>y to register.
- 0 : The operation is register to register.
- Source specifier field-Specifies the source register or data format.
- If $\mathrm{R} / \mathrm{M}=1$, specifies the source data format. See Table 7-6.
— If $R / M=0$, specifies the source floating-point data register, FPx .
- Destination register field-FTST uses the command word format used by all FPU arithmetic instructions but ignores and does not overwrite the register specified by this field. This field should be cleared for compatibility with future devices; however, because this field is ignored for the FTST instruction, the FPU does not signal an exception if the field is not zero.
motorola


## Freescale Semiconductor, Inc.

## 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. The original ColdFire Instruction Set Architecture, ISA_A, is supported by V2 and V3 cores. The V4 core supports ISA_B, which encompasses all of ISA_A, extends the functionality of some ISA_A instructions, and adds several new instructions. These extensions can be identified by a table which appears at the end of each instruction description where there are ISA_B differences.

# Push and Possibly Invalidate Cache 

(All ColdFire Processors)

## Operation: If Supervisor State

Then if Data Valid and Modified
Push Cache Line
Then Invalidate Line if Programmed in CACR
Else Privilege Violation Exception

Assembler Syntax: CPUSHL dc,(Ax)
CPUSHL ic,(Ax) CPUSHL bc,(Ax)
data cache
instruction cache both caches or unified cache

## Attributes: Unsized

Description: Pushes a specified cache line if modified and invalidates it if programmed to do so by CACR[DPI]. Care should be exercised when clearing lines from both caches if the sizes of the caches are different. For example, using a device with a 16 K instruction cache and an 8 K data cache, an address of $0 \times 800$ applied to both caches is referencing cache address $0 \times 80$ of the instruction cache, but address $0 \times 00$ of the data cache. Note that this instruction synchronizes the pipeline.

Condition Codes: Not affected
Instruction
Format:

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

## Instruction Fields:

- Cache - Specifies the affected cache as follows:
- 00 reserved
- 01 data cache (dc)
- 10 instruction cache (ic)
- 11 both caches or unified cache (bc); also use this encoding for a device which has an instruction cache, but not a data cache
- Register, Ax - Specifies the address register defining the line within the cache to be pushed or invalidated. Ax should be programmed as follows:
- Ax[4] is the lsb for the address field, which extends upward as required by the given cache size. The algorithm for the size of the address field is as follows:

Range $=$ Cache size in bytes $/($ Associativity $* 16)$
Using a $16 \mathrm{~K}, 4$ way set-associative cache as an example:

$$
\text { Range }=16384 /\left(4^{*} 16\right)=256=2^{8}
$$

Thus, the address range for this cache would be $\operatorname{Ax}[11: 4]$

- $\operatorname{Ax}[1: 0]$ specify the cache way or level where the line is located.


## Operation: If in Supervisor State

Then FPU State Frame $\rightarrow$ Internal State
Else Privilege Violation Exception
Assembler syntax: FRESTORE <ea>y

## Attributes: Unsized

Description: Aborts any floating-point operation and loads a new FPU internal state from the state frame at the effective address. The frame format is specified in the byte at <ea>y, and an internal exception vector is contained in the byte at <ea>y+1. If the frame format is invalid, FRESTORE aborts and a format exception is generated (vector 14). If the format is valid, the frame is loaded into the FPU, starting at the specified location and proceeding through higher addresses.

FRESTORE ignores the vector specified in the byte at $\langle\mathrm{ea}>\mathrm{y}+1$ because all vectors are generated from FPCR and FPSR exception bits. This vector is provided for the handler.

FRESTORE does not normally affect the FPU programming model except the NULL state frame. It is generally used with FMOVEM to fully restore the FPU context including floating-point data and system control registers. For complete restoration, FMOVEM first loads the data registers, then FRESTORE loads the internal state, FPCR, and FPSR. Table 8-1 lists supported state frames. If the frame format is not $0 x 00,0 x 05$, or $0 x E 5$, the processor responds with a format error exception, vector 14, and the internal FPU state is unaffected.

Table 8-1. State Frames

| State | Format | Description |
| :---: | :---: | :--- |
| NULL | $0 \times 00$ | FRESTORE of this state frame is like a hardware reset of the FPU. The programmer's model enters <br> reset state, with NANs in floating-point data registers and zeros in FPCR, FPSR, and FPIAR. |
| IDLE | $0 \times 05$ | A FRESTORE of the IDLE or EXCP state frame yields the same results. The FPU is restored to idle <br> state, waiting for initiation of the next instruction, with no exceptions pending. However, if an |
| EXCP | $0 \times E 5$ | FPSR[EXC] bit and corresponding FPCR enable bit are set, the FPU enters exception state. In this <br> state, initiating a floating-point instruction other than FSAVE, FMOVEM, FMOVE of system <br> registers, or another FRESTORE causes a pending exception. The programmer's model is <br> unaffected by loading this type of state frame (except FPSR and FPCR are loaded from the state <br> frame). |

## FPSR:

FPCR:
FPIAR:

Cleared if NULL frame format; otherwise, loaded from state frame.
Cleared if NULL frame format; otherwise, loaded from state frame.
Cleared if NULL frame format; otherwise unchanged.

## FRESTORE

## Restore Internal Floating-Point State

Floating-point data registers: Set to NANs if NULL frame format; otherwise, unaffected.


## Instruction field:

- Source Effective Address field-Specifies the addressing mode, <ea>y, for the state frame. Only modes in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| $A y$ | - | - |
| $(\mathrm{Ay})$ | 010 | reg. number:Ay |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | 101 | reg. number:Ay |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | 111 | 010 |
| $\left(\mathrm{~d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Save Internal

Operation: If in Supervisor State<br>Then FPU Internal State $\rightarrow\langle e a>x$<br>Else Privilege Violation Exception

Assembler syntax: FSAVE <ea>x
Attributes: Unsized
Description: After allowing completion of any floating-point operation in progress, FSAVE saves the FPU internal state in a frame at the effective address. After a save operation, FPCR is cleared and the FPU is in idle state until the next instruction executes. The first longword written to the state frame includes the format field data. Floating-point operations in progress when an FSAVE is encountered complete before FSAVE executes, which then creates an IDLE state frame if no exceptions occurred; otherwise, an EXCP state frame is created. State frames in Table 8-2 apply.

Table 8-2. State Frames

| State | Description |
| :---: | :--- |
| NULL | An FSAVE generating this state frame indicates the FPU state was not modified because the last processor <br> reset or FRESTORE with a NULL state frame. This indicates that the programmer's model is in reset state, <br> with NANs in floating-point data registers and zeros in FPCR, FPSR, and FPIAR. Stores of the system <br> registers, FSAVE, and FMOVEM stores do not cause the FPU change from NULL to another state. |
| IDLE | An FSAVE that generates this state frame indicates the FPU finished in an idle condition and is without <br> pending exceptions waiting for the initiation of the next instruction. |
| EXCP | An FSAVE generates this state frame if any FPSR[EXC] bits and corresponding FPCR exception enable bits <br> are set. This state typically indicates the FPU encountered an exception while attempting to complete <br> execution of a previous floating-point instruction. |

FSAVE does not save FPU programming model registers. It can be used with FMOVEM to perform a full context save of the FPU that includes floating-point data and system control registers. For a complete context save, first execute FSAVE to save the internal state, then execute the appropriate FMOVEM to store the data registers. FPCR and FPSR are saved as part of the FSAVE state frame. Furthermore, FPCR is cleared at the end of the FSAVE, preventing further exceptions if the handler includes floating-point instructions.

## FPSR: Not affected

FPCR: Cleared

## Freescale Semiconductor, Inc.

## Save Internal Floating-Point State

Instruction
Format:

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

## Instruction field:

- Effective address field-Specifies the addressing mode, <ea>x for the state frame. Only modes in the following table can be used.

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dx | - | - |
| $A x$ | - | - |
| $(A x)$ | 010 | reg. number:Ax |
| $(\mathrm{Ax})+$ | - | - |
| $-(\mathrm{Ax})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ax}\right)$ | 101 | reg. number:Ax |
| $\left(\mathrm{d}_{8}, \mathrm{Ax}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \# <data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

HALT

Operation: If Supervisor State
Then Halt the Processor Core
Else Privilege Violation Exception

Assembler Syntax: HALT
Attributes: Unsized
Description: The processor core is synchronized (meaning all previous instructions and bus cycles are completed) and then halts operation. The processor's halt status is signaled on the processor status output pins ( $\mathrm{PST}=0 \mathrm{xF}$ ). If a GO debug command is received, the processor resumes execution at the next instruction. Note that this instruction synchronizes the pipeline. The opcode for HALT is $0 x 4 \mathrm{AC} 8$.

Note that setting CSR[UHE] through the debug module allows HALT to be executed in user mode.

Condition Codes: Not affected
Instruction

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

## Freescale Semiconductor, Inc.

(Supported Starting with V4)
Operation: If Supervisor State
then Instruction Fetch Touch at (Ay)
else Privilege Violation Exception

## Assembler Syntax: INTOUCH (Ay)

## Attributes: Unsized

Description: Generates an instruction fetch reference at address (Ay). If the referenced address space is a cacheable region, this instruction can be used to prefetch a 16-byte packet into the processor's instruction cache. If the referenced instruction address is a non-cacheable space, the instruction effectively performs no operation. Note that this instruction synchronizes the pipeline.

The INTOUCH instruction can be used to prefetch, and with the later programming of CACR, lock specific memory lines in the processor's instruction cache. This function may be desirable in systems where deterministic real-time performance is critical.

Condition Codes: Not affected.

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Format: | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Register, Ay |  |  |

## Instruction Fields:

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

| INTOUCH | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | - |

## MOVE <br> from SR

Move from the Status Register

## MOVE <br> from SR

(All ColdFire Processors)
Operation: If Supervisor State
Then SR $\rightarrow$ Destination
Else Privilege Violation Exception
Assembler Syntax: MOVE.W SR,Dx
Attributes: $\quad$ 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 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Format: | 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.


## Freescale Semiconductor, Inc.

(Supported Starting with V4)
Operation: If Supervisor State
Then USP $\rightarrow$ Destination
Else Privilege Violation Exception
Assembler Syntax: MOVE.L USP,Ax
Attributes: $\quad$ 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 V2 or V3 device, or on the MCF5407, an illegal instruction exception will be taken. This instruction will execute correctly on other V4 devices 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 from USP | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | L |

Operation:

## Move to the Status Register

(All ColdFire Processors)

If Supervisor State
Then Source $\rightarrow$ SR
Else Privilege Violation Exception
Assembler Syntax: MOVE.W <ea>y,SR
Attributes: $\quad$ 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.


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

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |  | Source Effective Address |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  | gis |  |

## 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 | - | - |
| $(\mathrm{Ay})$ | - | - |
| $(\mathrm{Ay})+$ | - | - |
| $-(\mathrm{Ay})$ | - | - |
| $\left(\mathrm{d}_{16}, \mathrm{Ay}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{Ay}, \mathrm{Xi}\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(x x x) . \mathrm{W}$ | - | - |
| $(x x x) . \mathrm{L}$ | - | - |
| \#<data> | 111 | 100 |
|  |  |  |
|  |  |  |
| $\left(d_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Move to User Stack Pointer

(Supported Starting with V4)
Operation:
If Supervisor State
Then Source $\rightarrow$ USP
Else Privilege Violation Exception
Assembler Syntax: MOVE.L Ay,USP
Attributes: $\quad$ Size $=$ longword
Description: Moves the contents of an address register to the user stack pointer. If execution of this instruction is attempted on a V2 or V3 device, or on the MCF5407, an illegal instruction exception will be taken. This instruction will execute correctly on other V4 devices 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.

| MOVE to USP | V2, V3 Core (ISA_A) | V4 Core (ISA_B) |
| :---: | :---: | :---: |
| Opcode present | No | Yes |
| Operand sizes supported | - | L |

Operation: If Supervisor State
Then Ry $\rightarrow$ Rc
Else Privilege Violation Exception
Assembler Syntax: MOVEC.L Ry,Rc
Attributes: $\quad$ Size $=$ longword
Description: Moves the contents of the general-purpose register to the specified control register. This transfer is always 32 bits even though the control register may be implemented with fewer bits. Note that the control registers are write only. The on-chip debug module can be used to read control registers. Note that this instruction synchronizes the pipeline.

Not all control registers are implemented in every ColdFire processor design. Refer to the user's manual for a specific device to find out which registers are implemented. Attempted access to undefined or unimplemented control register space produces undefined results.

Condition Codes: Not affected
Instruction
Format:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| A/D | Register, Ry |  |  | Control Register, Rc |  |  |  |  |  |  |  |  |  |  |  |

## Instruction Fields:

- A/D field—Specifies the type of source register, Ry:
- 0 data register
- 1 address register
- Register Ry field—Specifies the source register, Ry.
- Control Register Rc field-Specifies the control register affected using the values shown in Table 8-3.


## Freescale Semiconductor, Inc.

Move Control Register
MOVEC

Table 8-3. ColdFire CPU Space Assignments

| Name | CPU Space Assignment | Register Name |
| :---: | :---: | :---: |
| Memory Management Control Registers |  |  |
| CACR | 0x002 | Cache control register |
| ASID | 0x003 | Address space identifier register |
| ACR0 | 0x004 | Access control registers 0 |
| ACR1 | 0x005 | Access control registers 1 |
| ACR2 | 0x006 | Access control registers 2 |
| ACR3 | 0x007 | Access control registers 3 |
| MMUBAR | 0x008 | MMU base address register |
| Processor Miscellaneous Registers |  |  |
| VBR | 0x801 | Vector base register |
| PC | 0x80F | Program counter |
| Local Memory and Module Control Registers |  |  |
| ROMBAR0 | 0xC00 | ROM base address register 0 |
| ROMBAR1 | 0xC01 | ROM base address register 1 |
| RAMBAR0 | 0xC04 | RAM base address register 0 |
| RAMBAR1 | 0xC05 | RAM base address register 1 |
| MPCR | 0xC0C | Multiprocessor control register ${ }^{1}$ |
| EDRAMBAR | 0xC0D | Embedded DRAM base address register ${ }^{1}$ |
| SECMBAR | 0xC0E | Secondary module base address register ${ }^{1}$ |
| MBAR | 0xC0F | Primary module base address register |
| Local Memory Address Permutation Control Registers ${ }^{1}$ |  |  |
| PCR1U0 | 0xD02 | 32 msbs of RAM 0 permutation control register 1 |
| PCR1L0 | 0xD03 | 32 Isbs of RAM 0 permutation control register 1 |
| PCR2U0 | 0xD04 | 32 msbs of RAM 0 permutation control register 2 |
| PCR2L0 | 0xD05 | 32 Isbs of RAM 0 permutation control register 2 |
| PCR3U0 | 0xD06 | 32 msbs of RAM 0 permutation control register 3 |
| PCR3L0 | 0xD07 | 32 Isbs of RAM 0 permutation control register 3 |
| PCR1U1 | 0xD0A | 32 msbs of RAM 1 permutation control register 1 |
| PCR1L1 | 0xD0B | 32 Isbs of RAM 1 permutation control register 1 |
| PCR2U1 | 0xD0C | 32 msbs of RAM 1 permutation control register 2 |
| PCR2L1 | 0xD0D | 32 Isbs of RAM 1 permutation control register 2 |
| PCR3U1 | 0xD0E | 32 msbs of RAM 1 permutation control register 3 |
| PCR3L1 | 0xD0F | 32 Isbs of RAM 1 permutation control register 3 |

1 Field definitions for these optional registers are implementation-specific.

RTE

Operation: If Supervisor State
Then $2+(\mathrm{SP}) \rightarrow \mathrm{SR} ; 4+(\mathrm{SP}) \rightarrow \mathrm{PC} ; \mathrm{SP}+8 \rightarrow \mathrm{SP}$
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 $\operatorname{SR}[S]=0$ when it is loaded from memory; otherwise, the processor remains in supervisor mode. Note that this instruction synchronizes the pipeline. The opcode for RTE is 0 x 4 E 73 .

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

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

(All ColdFire Processors)

Operation: If Supervisor State
Then Immediate Data $\rightarrow$ SR; STOP
Else Privilege Violation Exception
Assembler Syntax: STOP \#<data>

## Attributes: Unsized

Description: Moves the immediate word operand into the status register (both user and supervisor portions), advances the program counter to point to the next instruction, and stops the fetching and executing of instructions. A trace, interrupt, or reset exception causes the processor to resume instruction execution. A trace exception occurs if instruction tracing is enabled $(\mathrm{T} 0=1)$ when the STOP instruction begins execution, or if bit 15 of the immediate operand is a 1 . If an interrupt request is asserted with a priority higher than the priority level set by the new status register value, an interrupt exception occurs; otherwise, the interrupt request is ignored. External reset always initiates reset exception processing. The STOP command places the processor in a low-power state. Note that this instruction synchronizes the pipeline. The opcode for STOP is $0 \times 4 \mathrm{E} 72$, followed by the immediate data.

| Condition | X | N | z | v | c |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Codes: | * | * | * | * |  |  |

X Set to the value of bit 4 of the immediate data
N Set to the value of bit 3 of the immediate data
Z Set to the value of bit 2 of the immediate data
$V$ Set to the value of bit 1 of the immediate data
C Set to the value of bit 0 of the immediate data

Instruction
Format:

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

## Instruction Field:

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

Operation: If Supervisor State
Then Write Control Register Command Executed in Debug Module
Else Privilege Violation Exception
Assembler Syntax: WDEBUG.L <ea>y
Attributes: $\quad$ Size $=$ longword
Description: Fetches two consecutive longwords from the memory location defined by the effective address. These operands are used by the ColdFire debug module to write one of the debug control registers (DRc). Note that this instruction synchronizes the pipeline. The memory location defined by the effective address must be longword aligned; otherwise undefined operation results. The debug command must be organized in memory as shown on the next page.

Condition Codes: Not affected

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| Format: | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

## Instruction Field:

- Source Effective Address field—Specifies the address, <ea>y, for the operation; use only the addressing modes listed in the following table:

| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| Dy | - | - |
| Ay | - | - |
| $(A y)$ | 010 | reg. number:Ay |
| $(A y)+$ | - | - |
| $-(A y)$ | - | - |
| $\left(d_{16}, A y\right)$ | 101 | reg. number:Ay |
| $\left(d_{8}, A y, X i\right)$ | - | - |


| Addressing Mode | Mode | Register |
| :---: | :---: | :---: |
| $(\mathrm{xxx}) . \mathrm{W}$ | - | - |
| $(\mathrm{xxx}) . \mathrm{L}$ | - | - |
| \#<data> | - | - |
|  |  |  |
|  |  |  |
| $\left(\mathrm{d}_{16}, \mathrm{PC}\right)$ | - | - |
| $\left(\mathrm{d}_{8}, \mathrm{PC}, \mathrm{Xi}\right)$ | - | - |

## Freescale Semiconductor, Inc.

## WDEBUG

Debug Command Organization in Memory:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |  |  | DRc |  |  |
| Data[31:16] |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Data[15:0] |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Unused |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

where:

- Bits [15:4] of the first word define the WDREG command to the debug module.
- Bits [3:0] of the first word define the specific control register, DRc, to write. The table below contains DRc definitions. Note that some cores implement a subset of the debug registers. Refer to a specific device or core user's manual for more information.

| DRc[4-0] | Register Name |
| :---: | :--- |
| $0 \times 00$ | Configuration/status register |
| $0 \times 01-0 \times 0$ | Reserved |
| $0 \times 04$ | PC breakpoint ASID control |
| $0 \times 05$ | BDM address attribute register |
| $0 \times 06$ | Address attribute trigger register |
| $0 \times 07$ | Trigger definition register |
| $0 \times 08$ | Program counter breakpoint register |
| $0 \times 09$ | Program counter breakpoint mask <br> register |
| $0 \times 0 \mathrm{~A}-0 \times 0 \mathrm{~B}$ | Reserved |
| $0 \times 0 \mathrm{C}$ | Address breakpoint high register |
| $0 \times 0 \mathrm{D}$ | Address breakpoint low register |
| $0 \times 0 \mathrm{E}$ | Data breakpoint register |
| $0 \times 0 \mathrm{~F}$ | Data breakpoint mask register |


| DRc[4-0] | Register Name |
| :---: | :--- |
| $0 \times 10-0 \times 1$ | Reserved |
| $0 \times 14$ | PC breakpoint ASID register |
| $0 \times 15$ | Reserved |
| $0 \times 16$ | Address attribute trigger register 1 |
| $0 \times 17$ | Extended trigger definition register |
| $0 \times 18$ | Program counter breakpoint 1 register |
| $0 \times 19$ | Reserved |
| $0 \times 1 \mathrm{~A}$ | Program counter breakpoint register 2 |
| $0 \times 1 \mathrm{~B}$ | Program counter breakpoint register 3 |
| $0 \times 1 \mathrm{C}$ | Address high breakpoint register 1 |
| $0 \times 1 \mathrm{D}$ | Address low breakpoint register 1 |
| $0 \times 1 \mathrm{E}$ | Data breakpoint register 1 |
| $0 \times 1 \mathrm{~F}$ | Data breakpoint mask register 1 |

- Data[31:0] is the 32 -bit operand to be written.
- The fourth word is unused.


## Chapter 9 Instruction Format Summary

This chapter contains a numerical listing of the ColdFire family instructions in binary format. Wherever the binary encoding for an entire nibble of an instruction is predefined, the hex value for that nibble appears on the right side of the page, otherwise a dash ( - ) is used to show that it is variable.

### 9.1 Operation Code Map

Table 9-1 lists the encoding for bits 15-12 and the operation performed.
Table 9-1. Operation Code Map

| Bits 15-12 | Hex | Operation |
| :--- | :---: | :--- |
| 0000 | 0 | Bit Manipulation/Immediate |
| 0001 | 1 | Move Byte |
| 0010 | 2 | Move Longword |
| 0011 | 3 | Move Word |
| 0100 | 4 | Miscellaneous |
| 0101 | 5 | ADDQ/SUBQ/Scc/TPF |
| 0110 | 6 | Bcc/BSR/BRA |
| 0111 | 7 | MOVEQ/MVS/MVZ |
| 1000 | 8 | OR/DIV |
| 1001 | 9 | SUB/SUBX |
| 1010 | A | MAC/EMAC instructions/MOV3Q |
| 1011 | B | CMP/EOR |
| 1100 | D | AND/MUL |
| 1101 | E | Shift |
| 1110 | F | Floating-Point/Debug/Cache Instructions |
| 1111 |  |  |

## Freescale Semiconductor, Inc.

## Operation Code Map



## BTST

$0 \times 0-$
Bit number dynamic, specified in a register

| 15 | 14 | 13 | 12 | 11 | 109 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | 0 | 0 | Data Register, Dy |  | 1 | 0 | 0 |  | Destination Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  | Mode |  |  | gis |  |

## BCHG

$0 \times 0$
Bit number dynamic, specified in a register

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 |  |  |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

## BCLR

$0 \times 0-$
Bit number dynamic, specified in a register

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

BSET $0 \times 0$ —
Bit number dynamic, specified in a register

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

## ANDI

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Register, Dx |  |  |
| Upper Word of Immediate Data |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Lower Word of Immediate Data |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## SUBI

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Register, Dx |  |  |
| Upper Word of Immediate Data |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Lower Word of Immediate Data |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Freescale Semiconductor, Inc.
Operation Code Map


BCHG 0x08-00—
Bit number static, specified as immediate data


## BCLR

0x08- $00-$
Bit number static, specified as immediate data

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

## BSET

Bit number static, specified as immediate data


## EORI

0xOA8-


## CMPI <br> $0 \times 0 \mathrm{C}-$

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

MOVE $0 x —$


MOVEA
$0 x$


## NEGX

0x408-

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

## MOVE from SR

0x40C-

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

$$
\text { LEA } 0 \times 4-
$$



## CLR

0x42-

| 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 |  |  | Regis |  |

## MOVE from CCR

| 14 | 14 | 13 | 12 | 10 | 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 |  |  |


| 15 | 14 | 13 | 12 | 10 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |  | 2 | 1 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | 0

MOVE to CCR


NOT
0x468-

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

MOVE to SR 0x46


SWAP
0x484-

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

## PEA

0x48-

| 15 | 14 | 13 | 12 | 10 | 10 | 9 |  |  |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |

## EXT, EXTB

$0 \times 4$

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

MOVEM 0x4——


Freescale Semiconductor, Inc.
Operation Code Map
TST
$0 \times 4 \mathrm{~A}-$

| 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 |  |  | Regis |  |

## TAS

$0 x 4 \mathrm{~A}-$


HALT
0x4AC8

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

## PULSE <br> 0x4ACC

| 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 |

## ILLEGAL

0x4AFC

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

MULU.L
$0 \times 4 \mathrm{C}-\mathbf{0 0 0}$

| 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 |

MULS.L
$0 \times 4 C--800$

| 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 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
| 0 | Register, Dx |  |  | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## DIVU.L

$0 \times 4 \mathrm{C}-\mathbf{- 0 0}$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |  | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mod |  |  |  |
| 0 | Register, Dx |  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |  |

Freescale Semiconductor, Inc.

REMU.L
$0 \times 4 \mathrm{C}-\mathrm{OO}-$


DIVS.L
$0 \times 4 \mathrm{C}-80-$


REMS.L
$0 \times 4 \mathrm{C}-80-$


| 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 |  |  |

TRAP

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

LINK
0x4E5-

| 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 |  | ster |  |
| Word Displacement |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

UNLK
0x4E5-

| 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 |  |  |

MOVE to USP
0x4E6-

| 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 |  |  |

## Freescale Semiconductor, Inc.

MOVE from USP
0x4E6-

| 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 |  |  |

NOP
0x4E71

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

STOP
0x4E72

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| Immediate Data |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## RTE

0x4E73

| 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 |

## RTS

0x4E75

| 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 |

MOVEC
0x4E7B

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| D/A | Register, Ry |  |  | Control Register, Rc |  |  |  |  |  |  |  |  |  |  |  |

JSR


## JMP

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regis |  |


| 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 |  |  | Regis |  |

Freescale Semiconductor, Inc.

Scc

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

SUBQ 0x5——

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

TPF 0x51F-

BRA 0x60—

BSR 0x61—

| 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 $=0 \times 00$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 32 -bit displacement if 8 -bit displacement $=0 \times \mathrm{FF}$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Bcc
$0 \times 6$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | 1 | 1 | 0 | Condition |  |  |  | 8 -bit displacement |  |  |  |  |  |  |  |
| 16 -bit displacement if 8 -bit displacement $=0 \times 00$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 32 -bit displacement if 8 -bit displacement $=0 \times$ FF |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| MOVEQ 0x7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 1 | 1 |  | ister, Dx |  | 0 | Immediate Data |  |  |  |  |  |  |  |

Freescale Semiconductor, Inc.
Operation Code Map

| MVS |  |  |  |  |  |  |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 0 |
| 0 | 1 | 1 | 1 | Register, Dx |  | 1 | 0 | Size |  | Source Effective Address |  |  |  |
|  |  |  |  |  |  |  |  |  | Mode |  |  |  |

$$
\text { MVZ } 0 x 7-
$$


OR
$0 \times 8-$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 0 | 0 | Register |  |  | Opmode |  |  |  |  | ctiv | dd |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regis |  |

## DIVU.W

$0 \times 8$ -

$\left.\begin{array}{|l|l|l|l|l|l|l|l|l|l|l|l|l|}\hline 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3\end{array}\right) 2$| 1 |
| :--- |
| 1 |

## DIVS.W

$0 \times 8$ -

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | 22 | 1 |
| :--- |
| 1 |

SUB
$0 \times 9$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 0 | 1 | Register |  |  | Opmode |  |  |  |  | cti | ddr |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mod |  |  | Regi |  |

SUBX 0x9——

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

## SUBA

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 0 | 1 | Destination Register, Ax |  |  | 1 | 1 | 1 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regi |  |


| MAC | A |  |  |  |  |  |  |  |  |  |  |  |  | - |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1 | 0 | 1 | 0 |  | gister, Rx | 0 | 0 | Rx | 0 | 0 |  | Regi | , Ry |  |
| - | - | - | - | sz | Scale Factor | 0 | U/Lx | U/Ly | - | - | - | - | - | - |

MAC (EMAC) 0xA——

| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rx |  | 0 | $\begin{gathered} \text { ACC } \\ \text { Isb } \end{gathered}$ | $\begin{gathered} \mathrm{Rx} \\ \mathrm{msb} \end{gathered}$ | 0 | 0 | Register, Ry |  |  |  |
| - | - | - | - | sz | Scale Factor | 0 | U/Lx | U/Ly | - | $\begin{aligned} & \mathrm{ACC} \\ & \mathrm{msb} \end{aligned}$ | - | - | - | - |

## MAC with load (MAC)

$0 \times A-$

| 15 | 14 | 13 | 12 | 11 | $10 \quad 9$ | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rw |  | 0 | 1 | Rw | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  | od |  |  | Registe |  |
| Register, Rx |  |  |  | sz | Scale Factor |  | 0 | U/Lx | U/Ly | Mask | 0 |  | Reg | ter, Ry |  |

MAC with load (EMAC) OxA——

MSAC (MAC) 0xA-

| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rx |  | 0 | 0 | Rx | 0 | 0 | Register, Ry |  |  |  |
| - | - | - | - | sz | Scale Factor | 1 | U/Lx | U/Ly | - | - | - | - | - | - |
| MSAC (EMAC) 0xA |  |  |  |  |  |  |  |  |  |  |  |  |  |  |


| 15 | 14 | 13 | 12 | 11 | 10 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | Register, Rx |  | 0 | $\begin{gathered} \text { ACC } \\ \text { Isb } \end{gathered}$ | Rx msb | 0 | 0 | Register, Ry |  |  |  |
| - | - | - | - | sz | Scale Factor | 1 | U/Lx | U/Ly | - | $\begin{aligned} & \text { ACC } \\ & \mathrm{msb} \end{aligned}$ | - | - | - | - |

Freescale Semiconductor, Inc.

## MSAC with load (MAC)

$0 \times A-$


## MSAC with load (EMAC)

$0 \times A-$


MOVE to ACC (MAC)
0xA1-



MOVE ACC to ACC (EMAC)
0xA-1-

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

MOVE from ACC (MAC)
0xA18-

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |  |  |
| Register, Rx |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

MOVE from ACC (EMAC)
0xA-8-

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | 1

MOVCLR (EMAC)
$0 \times A-C-$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | 1

Freescale Semiconductor, Inc.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |  |  |
| Register, Rx |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

MOVE to MACSR 0xA9—


MOVE from MACSR to CCR
0xA9C0

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

MOVE to ACCext01 (EMAC)
0xAB-


MOVE from ACCext01 (EMAC)
0xAB8-

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

MOVE to MASK
0xAD-


MOVE from MASK
0xAD8-

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

MOVE to ACCext23 (EMAC)
0xAF-


MOVE from ACCext23 (EMAC)
0xAF8-

| 15 | 14 | ${ }^{13}$ | 12 | ${ }^{11}$ | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |  | Regis |  |  |

Freescale Semiconductor, Inc.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | 2 | 1 |
| :--- |
| 1 |

## CMP

$0 \times B-$

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

## CMPA

$0 \times B-$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 0 | 1 | 1 | Destination Register, Ax |  |  | Opmode |  |  |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  | Mode |  |  | Regi |  |

EOR 0xB——

| 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 |  |  | Regis |  |

AND
$0 \times C-$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | 2 | 1 |
| :--- |
| 1 |

MULU.W
$0 \times C-$

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

MULS.W
$0 \times C-$

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



Freescale Semiconductor, Inc.

ADDX

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |$\quad 1$| R |
| :--- |

## ADDA

0xD

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 0 | 1 | Destination Register, Ax |  |  | 1 | 1 | 1 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regis |  |

## ASL, ASR 0xE—

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

LSL, LSR 0xE——

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

FMOVE
0xF2-
Memory- and register-to-register operation (<ea>y,FPx; FPy,FPx)


## FMOVE <br> $0 \times 52-0$

Register-to-memory operation (FPy,<ea>x)

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
| 0 | 1 | 1 | Destination Format |  |  | Source <br> Register, FPy |  |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FINT
$0 \times 52-1$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
| 0 | R/M | 0 | Source Specifier |  |  | Destination Register, FPx |  |  | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Freescale Semiconductor, Inc.
Operation Code Map

FINTRZ
$0 \times F 2-$ - 3


## FSQRT

$0 \times F 2-$

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

FABS 0xF2—


## FNEG

$0 \times F 2-$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mod |  |  | Regist |  |
| 0 | R/M | 0 | Source Specifier |  |  | Destination Register, FPx |  |  |  | $\begin{gathered} \text { Opmode } \\ (0011010,1011010, \text { or } 1011110) \end{gathered}$ |  |  |  |  |  |

FDIV 0xF2—

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | 1

FADD 0xF2—
$\left.\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2\end{array} 1 \begin{array}{l}\text { Register }\end{array}\right]$

Freescale Semiconductor, Inc.

FMUL
$0 \times F 2-$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mod |  |  | Regi |  |
| 0 | R/M | 0 | Source Specifier |  |  | Destination Register, FPx |  |  |  |  | $111$ |  |  | $0111$ |  |

FSUB
$0 \times F 2-$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | 1

FCMP 0xF2———8

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Eff |  |  | e |  |  |
|  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
| 0 | R/M | 0 | Source Specifier |  |  | Destination Register, FPx |  |  | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

## FTST

$0 \times F 2-1$

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |  |  | Eff | A |  |  |
|  |  |  |  |  |  |  |  |  |  | Mode |  |  | Register |  |  |
| 0 | R/M | 0 | Source Specifier |  |  | Destination Register, FPx |  |  | 0 | 1 | 1 | 1 | 0 | 1 | 0 |

FBcc
0xF2-

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | Size |  | Conditional Predicate |  |  |  |  |
| 16-bit displacement or most significant word of 32-bit displacement |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Least significant word of 32-bit displacement (if needed) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

FMOVE to FPIAR
0xF2— 8400

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

## Freescale Semiconductor, Inc.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Source Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## FMOVE to FPCR

0xF2-9000

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

## FMOVE from FPIAR

0xF2- A400

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FMOVE from FPSR
0xF2-A800

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## FMOVE from FPCR

0xF2- B000

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | Destination Effective Address |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | od |  |  | gis |  |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FMOVEM
0xF2- 0 -


FNOP
0xF280 0000

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

FSAVE
0xF3-


## FRESTORE

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |  | Source Effective Address |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  | Mode |  |  | Regi |  |

## INTOUCH

0xF42-

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

CPUSHL 0xF4—

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

## WDDATA

$0 x F B-$


## WDEBUG

0xFB— 0003

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

## Freescale Semiconductor, Inc.

## Chapter 10 PST/DDATA Encodings

This chapter specifies the ColdFire processor and debug module's generation of the processor status (PST) and debug data (DDATA) output on an instruction basis. In general, the PST/DDATA output for an instruction is defined as follows:

$$
\text { PST }=0 \times 1,\{\operatorname{PST}=\{0 \times 8,0 \times 9,0 \times B\}, \text { DDATA }=\text { operand }\}
$$

where the $\{\ldots\}$ definition is optional operand information defined by the setting of the CSR.
The CSR provides capabilities to display operands based on reference type (read, write, or both). A PST value $\{0 \mathrm{x} 8,0 \mathrm{x} 9$, or 0 xB$\}$ identifies the size and presence of valid data to follow on the DDATA output $\{1,2$, or 4 bytes $\}$. Additionally, for certain change-of-flow branch instructions, CSR[BTB] provides the capability to display the target instruction address on the DDATA output $\{2,3$, or 4 bytes $\}$ using a PST value of $\{0 \mathrm{x} 9,0 \mathrm{xA}$, or 0 xB$\}$.

For V2 and V3 devices, PST and DDATA are separate ports; and real-time trace information is displayed on both ports concurrently. Starting with V4, the PST and DDATA outputs are combined into a single port. Real-time trace information appears as a sequence of 4-bit data values with no alignment restrictions; that is, the processor status (PST) values and operands (DDATA) may appear on either nibble of PSTDDATA[7:0]. The upper nibble (PSTDDATA[7:4]) is the most significant and yields values first. Note that the combined PSTDDATA output still displays processor status and debug data in a manner that is compatible with the displays generated with the separate PST and DDATA outputs. For further information, refer to the debug section of a device or core user's manual.

Note that not all instructions are implemented on all cores and devices. Refer to Chapter 12, "Processor Instruction Summary," for further information.

### 10.1 User Instruction Set

Table 10-1 shows the PST/DDATA specification for user-mode instructions. Rn represents any $\{D n, A n\}$ register. The ' $y$ ' suffix denotes the source and ' $x$ ' denotes the destination operand. For a given instruction, the optional operand data is displayed only for those effective addresses referencing memory. The 'DD' nomenclature refers to the DDATA outputs.

## Freescale Semiconductor, Inc.

## User Instruction Set

Table 10-1. PST/DDATA Specification for User-Mode Instructions

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :---: |
| add.I | <ea>y,Dx | PST $=0 \times 1,\{\mathrm{PST}=0 \times B, \mathrm{DD}=$ source operand $\}$ |
| add.I | Dy,<ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| adda.l | <ea>y,Ax | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source operand $\}$ |
| addi.I | \#<data>,Dx | PST $=0 \times 1$ |
| addq. ${ }^{\text {l }}$ | \#<data>,<ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| addx.l | Dy, Dx | PST = 0x1 |
| and.l | <ea>y, Dx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| and.l | Dy,<ea>x | PST = 0x1, PST = 0xB, DD = source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| andi.I | \#<data>,Dx | PST $=0 \times 1$ |
| asl.l | \{Dy,\#<data>\},Dx | PST $=0 \times 1$ |
| asr.l | \{Dy,\#<data>\},Dx | PST = 0x1 |
| bcc. $\{\mathrm{b}, \mathrm{w}, 1\}$ |  | if taken, then PST $=0 \times 5$, else PST $=0 \times 1$ |
| bchg.\{b,l\} | \#<data>,<ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ destination $\}$ |
| bchg.\{b,l\} | Dy,<ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ destination $\}$ |
| bclr.\{b,l\} | \#<data>,<ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ destination $\}$ |
| bclr.\{b,l\} | Dy, <ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ destination $\}$ |
| bra.\{b,w,l\} |  | PST $=0 \times 5$ |
| bset.\{b,l\} | \#<data>,<ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ destination $\}$ |
| bset.\{b,l\} | Dy, <ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ destination $\}$ |
| bsr.\{b,w,l\} |  | PST $=0 \times 5,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination operand $\}$ |
| btst. $\{\mathrm{b}, \mathrm{l}\}$ | \#<data>,<ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source operand $\}$ |
| btst.\{b,l\} | Dy, <ea>x | PST $=0 \times 1,\{$ PST $=0 \times 8, \mathrm{DD}=$ source operand $\}$ |
| clr.b | <ea>x | PST $=0 \times 1,\{$ PST $=0 \times 8, \mathrm{DD}=$ destination operand $\}$ |
| clr.l | <ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination operand $\}$ |
| clr.w | <ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times 9, \mathrm{DD}=$ destination operand $\}$ |
| cmp.b | <ea>y,Dx | PST $=0 \times 1$, $\{0 \times 8$, source operand $\}$ |
| cmp.l | <ea>y,Dx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| cmp.w | <ea>y,Dx | PST $=0 \times 1,\{0 \times 9$, source operand $\}$ |
| cmpa.l | <ea>y,Ax | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| cmpa.w | <ea>y,Ax | PST $=0 \times 1,\{0 \times 9$, source operand $\}$ |
| cmpi.b | \#<data>,Dx | PST $=0 \times 1$ |
| cmpi.l | \#<data>,Dx | PST $=0 \times 1$ |
| cmpi.w | \#<data>,Dx | PST $=0 \times 1$ |
| divs.I | <ea>y,Dx | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source operand $\}$ |
| divs.w | <ea>y,Dx | PST $=0 \times 1,\{$ PST $=0 \times 9, \mathrm{DD}=$ source operand $\}$ |
| divu.I | <ea>y,Dx | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source operand $\}$ |

## Freescale Semiconductor, Inc.

Table 10-1. PST/DDATA Specification for User-Mode Instructions (Continued)

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :---: |
| divu.w | <ea>y,Dx | PST $=0 \times 1,\{\mathrm{PST}=0 \times 9, \mathrm{DD}=$ source operand $\}$ |
| eor.l | Dy, <ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| eori.l | \#<data>,Dx | PST = 0x1 |
| ext.l | Dx | PST = 0x1 |
| ext.w | Dx | PST $=0 \times 1$ |
| extb.l | Dx | PST = 0x1 |
| illegal |  | PST $=0 \times 1{ }^{1}$ |
| jmp | <ea>y | PST $=0 \times 5,\{\mathrm{PST}=\{0 \times 9,0 \times \mathrm{A}, 0 \times \mathrm{B}\}, \mathrm{DD}=\text { target address }\}^{2}$ |
| jsr | <ea>y | $\begin{aligned} \mathrm{PST}=0 \times 5,\{\mathrm{PST} & =\{0 \times 9,0 \times \mathrm{A}, 0 \times \mathrm{B}\}, \mathrm{DD}=\text { target address }\}, \\ \{\mathrm{PST} & =0 \times B, \mathrm{DD}=\text { destination operand }\}^{2} \end{aligned}$ |
| lea.l | <ea>y,Ax | PST $=0 \times 1$ |
| link.w | Ay,\#<displacement> | PST $=0 \times 1,\{\mathrm{PST}=0 \times B, \mathrm{DD}=$ destination operand $\}$ |
| Is..I | \{Dy,\#<data>\},Dx | PST = 0x1 |
| Isr.I | \{Dy,\#<data>\},Dx | PST $=0 \times 1$ |
| mov3q.1 | \#<data>,<ea>x | PST $=0 \times 1$, \{0xB, destination operand $\}$ |
| move.b | <ea>y,<ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ destination $\}$ |
| move.l | <ea>y,<ea>x | PST = 0x1, \{PST = 0xB, DD = source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| move.w | <ea>y,<ea>x | PST = 0x1, \{PST = 0x9, DD = source $\},\{\mathrm{PST}=0 \times 9, \mathrm{DD}=$ destination $\}$ |
| move.w | CCR,Dx | PST = 0x1 |
| move.w | \{Dy,\#<data>\},CCR | PST $=0 \times 1$ |
| movea.l | <ea>y,Ax | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source $\}$ |
| movea.w | <ea>y,Ax | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times 9, \mathrm{DD}=$ source $\}$ |
| movem.l | \#list, <ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}, \ldots{ }^{3}$ |
| movem.l | <ea>y,\#list | PST $=0 \times 1,\{\mathrm{PST}=0 \times B, \mathrm{DD}=$ source $\}, \ldots{ }^{3}$ |
| moveq. 1 | \#<data>,Dx | PST $=0 \times 1$ |
| muls. 1 | <ea>y, Dx | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source operand $\}$ |
| muls.w | <ea>y,Dx | PST $=0 \times 1,\{\mathrm{PST}=0 \times 9, \mathrm{DD}=$ source operand $\}$ |
| mulu.l | <ea>y,Dx | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source operand $\}$ |
| mulu.w | $<e a>y, D x$ | PST $=0 \times 1,\{\mathrm{PST}=0 \times 9, \mathrm{DD}=$ source operand $\}$ |
| mvs.b | <ea>y, Dx | PST $=0 \times 1,\{0 \times 8$, source operand $\}$ |
| mvs.w | <ea>y,Dx | PST $=0 \times 1,\{0 \times 9$, source operand $\}$ |
| mvz.b | <ea>y, Dx | PST $=0 \times 1,\{0 \times 8$, source operand $\}$ |
| mvz.w | <ea>y,Dx | PST $=0 \times 1,\{0 \times 9$, source operand $\}$ |
| neg.l | Dx | PST $=0 \times 1$ |
| negx.l | Dx | PST $=0 \times 1$ |
| nop |  | PST $=0 \times 1$ |

## Freescale Semiconductor, Inc.

## User Instruction Set

Table 10-1. PST/DDATA Specification for User-Mode Instructions (Continued)

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :---: |
| not.l | Dx | PST $=0 \times 1$ |
| or.l | <ea>y,Dx | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source operand $\}$ |
| or.l | Dy,<ea>x | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| ori.l | \#<data>,Dx | PST $=0 \times 1$ |
| pea.l | <ea>y | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination operand $\}$ |
| pulse |  | PST $=0 \times 4$ |
| rems.I | <ea>y,Dw:Dx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| remu.l | <ea>y,Dw:Dx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| rts |  | ```PST = 0x1, PST = 0x5, {{0x9,0\timesA,0xB}, target address } PST = 0x1, {PST = 0xB, DD = source operand}, PST = 0x5, {PST = {0x9,0xA,0xB }, DD = target address }``` |
| sats.I | Dx | PST $=0 \times 1$ |
| scc.b | Dx | PST $=0 \times 1$ |
| sub.l | <ea>y,Dx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| sub.l | Dy,<ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| suba.l | <ea>y,Ax | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| subi.l | \#<data>,Dx | PST $=0 \times 1$ |
| subq.I | \#<data>,<ea>x | PST $=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source $\},\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ destination $\}$ |
| subx.l | Dy,Dx | PST $=0 \times 1$ |
| swap.w | Dx | PST $=0 \times 1$ |
| tas.b | <ea>x | PST $=0 \times 1,\{0 \times 8$, source $\},\{0 \times 8$, destination $\}$ |
| tpf |  | PST $=0 \times 1$ |
| tpf.l | \#<data> | PST $=0 \times 1$ |
| tpf.w | \#<data> | PST = 0x1 |
| trap | \#<data> | PST $=0 \times 1^{1}$ |
| tst.b | <ea>x | PST $=0 \times 1,\{$ PST $=0 \times 8, \mathrm{DD}=$ source operand $\}$ |
| tst.I | <ea>y | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| tst.w | <ea>y | PST $=0 \times 1,\{$ PST $=0 \times 9$, DD $=$ source operand $\}$ |
| unlk | Ax | PST $=0 \times 1$, PPST $=0 \times B, \mathrm{DD}=$ destination operand $\}$ |
| wddata.b | <ea>y | PST $=0 \times 4,\{\mathrm{PST}=0 \times 8, \mathrm{DD}=$ source operand |
| wddata.I | <ea>y | PST $=0 \times 4,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=$ source operand |
| wddata.w | <ea>y | PST $=0 \times 4,\{\mathrm{PST}=0 \times 9, \mathrm{DD}=$ source operand |

## Freescale Semiconductor, Inc.

1 During normal exception processing, the PST output is driven to a $0 \times \mathrm{C}$ indicating the exception processing state. The exception stack write operands, as well as the vector read and target address of the exception handler may also be displayed.
Exception Processing $P S T=0 x C$, $\{P S T=0 x B, D D=$ destination\}, // stack frame
$\{P S T=0 x B, D D=$ destination\}, // stack frame
$\{\mathrm{PST}=0 x B, D D=$ source\}, // vector read
PST $=0 \times 5, \quad\{\mathrm{PST}=[0 \mathrm{x} 9 \mathrm{AB}], \mathrm{DD}=$ target $\} \quad / /$ handler PC
The PST/DDATA specification for the reset exception is shown below:
Exception Processing PST $=0 x C$,
PST $=0 \times 5, \quad\{\mathrm{PST}=$ [0x9AB],DD $=$ target $\} \quad / /$ handler PC
The initial references at address 0 and 4 are never captured nor displayed since these accesses are treated as instruction fetches.
For all types of exception processing, the PST $=0 \times \mathrm{C}$ value is driven at all times, unless the PST output is needed for one of the optional marker values or for the taken branch indicator ( $0 \times 5$ ).
2 For JMP and JSR instructions, the optional target instruction address is displayed only for those effective address fields defining variant addressing modes. This includes the following <ea>x values: (An), (d16,An), (d8,An,Xi), (d8,PC,Xi).
3 For Move Multiple instructions (MOVEM), the processor automatically generates line-sized transfers if the operand address reaches a 0-modulo-16 boundary and there are four or more registers to be transferred. For these line-sized transfers, the operand data is never captured nor displayed, regardless of the CSR value.
The automatic line-sized burst transfers are provided to maximize performance during these sequential memory access operations.

Table 10-2 shows the PST specification for multiply-accumulate instructions.

## Table 10-2. PST/DDATA Values for User-Mode Multiply-Accumulate Instructions

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :---: |
| mac.I | Ry,Rx | PST = 0x1 |
| mac.l | Ry,Rx,<ea>y,Rw,ACCx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| mac.l | Ry,Rx,ACCx | PST = 0x1 |
| mac.l | Ry,Rx,ea,Rw | PST $=0 \times 1,\{\mathrm{PST}=0 \times B, \mathrm{DD}=$ source operand $\}$ |
| mac.w | Ry, Rx | PST $=0 \times 1$ |
| mac.w | Ry,Rx, <ea>y,Rw,ACCx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| mac.w | Ry,Rx,ACCx | PST = 0x1 |
| mac.w | Ry,Rx,ea,Rw | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| move.l | \{Ry,\#<data>\},ACCext01 | PST = 0x1 |
| move.l | \{Ry,\#<data>\},ACCext23 | PST = 0x1 |
| move.l | \{Ry,\#<data>\},ACCx | PST = 0x1 |
| move.l | \{Ry,\#<data>\},MACSR | PST = 0x1 |
| move.l | \{Ry,\#<data>\},MASK | $\mathrm{PST}=0 \times 1$ |
| move.l | ACCext01,Rx | PST = 0x1 |
| move.l | ACCext23,Rx | PST = 0x1 |
| move.l | ACCy,ACCx | $\mathrm{PST}=0 \times 1$ |
| move.l | ACCy,Rx | PST = 0x1 |

motorola

## Freescale Semiconductor, Inc.

## User Instruction Set

Table 10-2. PST/DDATA Values for User-Mode
Multiply-Accumulate Instructions (Continued)

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :---: |
| move.l | MACSR,CCR | PST $=0 \times 1$ |
| move.l | MACSR,Rx | PST = 0x1 |
| move.l | MASK,Rx | PST $=0 \times 1$ |
| msac.l | Ry,Rx | PST $=0 \times 1$ |
| msac.l | $R y, R x,<e a>y, R w, A C C x$ | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| msac.l | Ry,Rx,ACCx | PST $=0 \times 1$ |
| msac.l | Ry, Rx, <ea>y, Rw | $\begin{aligned} & \mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times \mathrm{B}, \mathrm{DD}=\text { source }\}, \\ & \{\mathrm{PST}=0 \times B, \mathrm{DD}=\text { destination }\} \end{aligned}$ |
| msac.w | Ry,Rx | PST $=0 \times 1$ |
| msac.w | Ry,Rx, <ea>y,Rw,ACCx | PST $=0 \times 1,\{$ PST $=0 \times B$, DD $=$ source operand $\}$ |
| msac.w | Ry,Rx,ACCx | PST $=0 \times 1$ |
| msac.w | Ry, Rx, <ea>y,Rw | $\begin{aligned} & \text { PST }=0 \times 1,\{\mathrm{PST}=0 \times B, \mathrm{DD}=\text { source }\}, \\ & \{\mathrm{PST}=0 \times B, \mathrm{DD}=\text { destination }\} \end{aligned}$ |

Table 10-3 shows the PST/DDATA specification for floating-point instructions; note that <ea>y includes FPy, Dy, Ay, and <mem>y addressing modes. The optional operand capture and display applies only to the <mem>y addressing modes. Note also that the PST/DDATA values are the same for a given instruction, regardless of explicit rounding precision.

## Table 10-3. PST/DDATA Values for User-Mode Floating-Point Instructions

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :---: |
| fabs.sz | <ea>y,FPx | PST $=0 \times 1$, [89B], source $\}$ |
| fadd.sz | <ea>y,FPx | PST = 0x1, [89B], source |
| fbcc. $\{\mathrm{w}, \mathrm{l}\}$ | <label> | if taken, then PST $=5$, else PST $=0 \times 1$ |
| fcmp.sz | <ea>y,FPx | PST $=0 \times 1$, [89B], source $\}$ |
| fdiv.sz | <ea>y,FPx | PST $=0 \times 1$, [89B], source $\}$ |
| fint.sz | <ea>y,FPx | PST = 0x1, [89B], source $\}$ |
| fintrz.sz | <ea>y,FPx | PST $=0 \times 1$, [89B], source $\}$ |
| fmove.sz | <ea>y,FPx | PST = 0x1, [89B], source $\}$ |
| fmove.sz | FPy,<ea>x | PST $=0 \times 1$, [89B], destination $\}$ |
| fmove.l | $<e a>y, \mathrm{FP}^{*} \mathrm{R}^{1}$ | PST $=0 \times 1$, B, source $\}$ |
| fmove.l | FP*R, <ea>x ${ }^{1}$ | PST $=0 \times 1, \mathrm{~B}$, destination $\}$ |
| fmovem | <ea>y,\#list | PST $=0 \times 1$ |
| fmovem | \#list, <ea>x | PST $=0 \times 1$ |
| fmul.sz | <ea>y,FPx | PST = 0x1, [89B], source $\}$ |

## Freescale Semiconductor, Inc.

Table 10-3. PST/DDATA Values for User-Mode Floating-Point Instructions (Continued)

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :---: |
| fneg.sz | <ea>y,FPx | PST = 0x1, [89B], source $\}$ |
| fnop |  | PST $=0 \times 1$ |
| fsqrt.sz | <ea>y,FPx | PST = 0x1, [89B], source $\}$ |
| fsub.sz | <ea>y,FPx | PST = 0x1, [89B], source $\}$ |
| ftst.sz | <ea>y | PST = 0x1, [89B], source $\}$ |

Depending on the size of any external memory operand specified by the $\mathrm{f}<\mathrm{op}>$.fmt field, the data marker is defined as shown in Table 10-4

## Table 10-4. Data Markers and FPU Operand Format Specifiers

| Format Specifier | Data Marker |
| :---: | :---: |
| . b | 8 |
| .$w$ | 9 |
| . I | B |
| .$s$ | B |
| .$d$ | Never captured |

### 10.2 Supervisor Instruction Set

The supervisor instruction set has complete access to the user mode instructions plus the opcodes shown below. The PST/DDATA specification for these opcodes is shown in Table 10-5.

Table 10-5. PST/DDATA Specifications for Supervisor-Mode Instructions

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :--- |
| cpushl | $\mathrm{dc},(\mathrm{Ax})$ <br> $\mathrm{ic},(\mathrm{Ax})$ <br> $\mathrm{bc},(\mathrm{Ax})$ | $\mathrm{PST}=0 \times 1$ |
| frestore | <ea>y | $\mathrm{PST}=0 \times 1$ |
| fsave | <ea>x | $\mathrm{PST}=0 \times 1$ |
| halt |  | PST $=0 \times 1$, <br> $\mathrm{PST}=0 \times F$ |
| intouch | (Ay) | $\mathrm{PST}=0 \times 1$ |
| move.I | Ay,USP | $\mathrm{PST}=0 \times 1$ |
| move.l | USP,Ax | $\mathrm{PST}=0 \times 1$ |

Table 10-5. PST/DDATA Specifications for Supervisor-Mode Instructions (Continued)

| Instruction | Operand Syntax | PST/DDATA |
| :---: | :---: | :--- |
| move.w | SR,Dx | PST $=0 \times 1$ |
| move.w | $\{\mathrm{Dy}, \#<$ data $>\}, \mathrm{SR}$ | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times 3\}$ |
| movec.l | Ry,Rc | $\mathrm{PST}=0 \times 1,\{8, \mathrm{ASID}\}$ |
| re |  | $\mathrm{PST}=0 \times 7,\{\mathrm{PST}=0 \times B, \mathrm{DD}=$ source operand $\},\{\mathrm{PST}=3\}$ <br> $\{\mathrm{PST}=0 \times B, \mathrm{DD}=$ source operand $\},\{\mathrm{DD}\}$, <br> $\mathrm{PST}=0 \times 5,\{[\mathrm{PST}=0 \times 9 \mathrm{AB}], \mathrm{DD}=$ target address $\}$ |
| stop | \#<data $>$ | $\mathrm{PST}=0 \times 1$, <br> $\mathrm{PST}=0 \times \mathrm{E}$ |
| wdebug.l | <ea>y | $\mathrm{PST}=0 \times 1,\{\mathrm{PST}=0 \times B, \mathrm{DD}=$ source, PST $=0 \times B$, <br> $\mathrm{DD}=$ source $\}$ |

The move-to-SR and RTE instructions include an optional PST $=0 \times 3$ value, indicating an entry into user mode. Additionally, if the execution of a RTE instruction returns the processor to emulator mode, a multiple-cycle status of 0 xD is signaled.

Similar to the exception processing mode, the stopped state $(\mathrm{PST}=0 \mathrm{xE})$ and the halted state $(\mathrm{PST}=0 \mathrm{xF})$ display this status throughout the entire time the ColdFire processor is in the given mode.

## 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

Because the V4 core can implement an MMU, exception processing for devices containing an MMU is slightly modified. Differences from previous ColdFire Family processors include the following:

- An instruction restart model for translation (TLB miss) and access faults. This new functionality extends the existing ColdFire access error fault vector and exception stack frames.
- Use of separate system stack pointers for user and supervisor modes.

Previous ColdFire processors (V2 and V3) use an instruction restart exception model but require additional 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, $\mathrm{SR}[I]$ to the level of the current interrupt request.

## Freescale Semiconductor, Inc.

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. V2 and V3 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. Because V4 supports a supervisor stack pointer (SSP), the exception stack frame is created at a 0 -modulo- 4 address on top of the system stack pointed to by the SSP.
V2 and V3 use a simplified fixed-length stack frame, shown in Figure 11-1, for all exceptions. V4 uses 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 | Assignment |
| :---: | :---: | :---: | :--- |
| 0 | 000 | - | Initial stack pointer (SSP for V4e) |
| 1 | 004 | - | Initial program counter |
| 2 | 008 | Fault | Access error |

Table 11-1. Exception Vector Assignments (Continued)

| Vector Numbers | Vector Offset (Hex) | Stacked Program Counter ${ }^{1}$ | Assignment |
| :---: | :---: | :---: | :---: |
| 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 | 064-07C | Next | Level 1-7 autovectored interrupts |
| 32-47 | 080-0BC | Next | Trap \#0-15 instructions |
| $48^{4}$ | 0C0 | Fault | Floating-point branch on unordered condition |
| $49^{4}$ | 0C4 | NextFP or Fault | Floating-point inexact result |
| $50^{4}$ | 0 C 8 | NextFP | Floating-point divide-by-zero |
| $51^{4}$ | OCC | NextFP or Fault | Floating-point underflow |
| $52^{4}$ | 0D0 | NextFP or Fault | Floating-point operand error |
| $53^{4}$ | 0D4 | NextFP or Fault | Floating-point overflow |
| $54^{4}$ | 0D8 | NextFP or Fault | Floating-point input not-a-number (NAN) |
| $55^{4}$ | ODC | NextFP or Fault | Floating-point input denormalized number |
| 56-60 | 0E0-0F0 | - | Reserved |
| $61^{5}$ | 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 If the FPU is not present, vectors $48-55$ are reserved.
5 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.

## Freescale Semiconductor, Inc.

### 11.1.1 Supervisor/User Stack Pointers (A7 and OTHER_A7)

The V4 architecture supports 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 as dictated by the virtual memory management scheme provided by the MMU. 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 |
| 27-26 | FS[3-2] | Fault status. Defined for access and address errors and for interrupted debug service routines. <br> 0000 Not an access or address error nor an interrupted debug service routine <br> 0001 Reserved <br> 0010 Interrupt during a debug service routine for faults other than access errors. (New in V4) ${ }^{1}$ <br> 0011 Reserved <br> 0100 Error (for example, protection fault) on instruction fetch <br> 0101 TLB miss on opword of instruction fetch (New in V4, MMU only) <br> 0110 TLB miss on extension word of instruction fetch (New in V4, MMU only) <br> 0111 IFP access error while executing in emulator mode (New in V4, MMU only) <br> 1000 Error on data write <br> 1001 Error on attempted write to write-protected space <br> 1010 TLB miss on data write (New in V4, MMU only) <br> 1011 Reserved <br> 1100 Error on data read <br> 1101 Attempted read, read-modify-write of protected space (New in V4, MMU only) <br> 1110 TLB miss on data read, or read-modify-write (New in V4, MMU only) <br> 1111 OEP access error while executing in emulator mode (New in V4, MMU only) |  |  |
| 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.

Table 11-3. Exceptions

| Vector <br> Number | Type | Description |
| :---: | :---: | :---: |
| 0,1 | Reset | Asserting the reset input signal ( $\overline{\mathrm{RSTI}})$ causes a reset exception, which has the highest exception priority and provides for system initialization and recovery from catastrophic failure. When assertion of $\overline{\mathrm{RSTI}}$ is recognized, current processing is aborted and cannot be recovered. The reset exception places the processor in supervisor mode by setting $\operatorname{SR}[\mathrm{S}]$ and disables tracing by clearing $\operatorname{SR}[T]$. It clears $\operatorname{SR}[M]$ and sets $\operatorname{SR}[1]$ to the highest level (0b111, priority level 7). Next, VBR is cleared. Configuration registers controlling operation of all on-chip memories are invalidated, disabling the memories. Note: Implementation-specific supervisor registers are also affected at reset. After RSTI is negated, the processor waits a number of cycles before beginning the reset exception process. During this time, certain events are sampled, including the assertion of the debug breakpoint signal. If the processor is not halted, it initiates the reset exception by performing two longword read bus cycles. The longword at address 0 is loaded into the stack pointer and the longword at address 4 is loaded into the PC. After the initial instruction is fetched from memory, program execution begins at the address in the PC. If an access error or address error occurs before the first instruction executes, the processor enters a fault-on-fault halted state. |
| 2 | Access error | Caused by an error when accessing memory. ColdFire cores handle access errors differently: |
|  |  | V2: For an access error on an instruction fetch, the processor postpones the error reporting until the instruction at the faulted reference is executed. Thus, faults that occur during instruction prefetches that are followed by a change of instruction flow do not generate an exception. When the processor attempts to execute an instruction with a faulted opword or extension word, the access error is signaled, and the instruction is aborted; the programming model is not altered by the faulted instruction. <br> If an access error occurs on an operand read, the processor immediately aborts the current instruction execution and initiates exception processing. Any address register changes caused by the auto-addressing modes, (An)+ and -(An), have already occurred. In addition, if the error occurs during the execution of a MOVEM instruction loading from memory, registers may contain memory operands. <br> Due to the processor pipeline implementation, a write cycle may be decoupled from the execution of the instruction causing the write. Thus, if an access error occurs on an operand write, the signaling of the error is imprecise. Accordingly, the PC contained in the exception stack frame represents the location in the program when the access error is signaled, not necessarily the instruction causing the fault. All programming model updates associated with the write instruction are complete. The NOP instruction can be used to help identify write access errors. A NOP is not executed until all previous operations, including any pending writes, are complete. Thus, if any previous write terminates with an access error, it is guaranteed to be reported on the NOP. |

V3: Access errors are reported only in conjunction with an attempted store to write-protected memory. Thus, access errors associated with instruction fetch or operand read accesses are not possible.

V4: If the MMU is disabled, access errors are reported only in conjunction with an attempted store to write-protected memory. Thus, access errors associated with instruction fetch or operand read accesses are not possible. The condition code register is updated if a write-protect error occurs during a CLR or MOV3Q operation to memory.
Internal memory accesses that fault generate an access error exception. MMU TLB misses and access violations use the same fault. If the MMU is enabled, all TLB misses and protection violations generate an access error exception. To quickly determine if a fault is due to a TLB miss or another type of access error, new FS encodings (described in Table 11-2) signal TLB misses on instruction fetch, instruction extension fetch, data read, and data write.

Table 11-3. Exceptions (Continued)

| Vector Number | Type | Description |
| :---: | :---: | :---: |
| 3 | Address error | An address error is caused by an attempted execution transferring control to an odd instruction address (that is, if bit 0 of the target address is set), an attempted use of a word-sized index register (Xi.w), or by an attempted execution of an instruction with a full-format indexed addressing mode. <br> If an address error occurs on a JSR instruction, the V4 processor first pushes the return address onto the stack and then calculates the target address. On V2 and V3 processors, the target address is calculated, then the return address is pushed on stack. <br> If an address error occurs on an RTS instruction, the V4 processor preserves the original return PC and writes the exception stack frame above this value. On V2 and V3 processors, the faulting return PC is overwritten by the address error stack frame. |
| 4 | Illegal instruction | On V2, only some illegal opcodes ( $0 \times 0000$ and 0x4AFC) are decoded and generate an illegal instruction exception. Additionally, attempting to execute an illegal line A or line F opcode generates unique exception types: vectors 10 and 11, respectively. If any other nonsupported opcode is executed, the resulting operation is undefined. V3 and V4 decode the complete 16-bit opcode, and this exception is generated if execution of an unsupported instruction is attempted. In addition, the illegal opcodes above, line A and line F, also generate this exception. <br> ColdFire processors do not provide illegal instruction detection on extension words of any instruction, including MOVEC. Attempting to execute an instruction with an illegal extension word causes undefined results. |
| 5 | Divide-by-zero | Attempting to divide by zero causes an exception (vector 5, offset $=0 \times 014$ ). Note that this exception cannot be generated unless the device has a divide unit. |
| 8 | Privilege violation | Caused by attempted execution of a supervisor mode instruction while in user mode. |
| 9 | Trace | Trace mode, which allows instruction-by-instruction tracing, is enabled by setting SR[T]. If SR[T] is set, instruction completion (for all but the STOP instruction) signals a trace exception. The STOP instruction has the following effects: <br> 1 The instruction before the STOP executes and then generates a trace exception. In the exception stack frame, the PC points to the STOP opcode. <br> 2 When the trace handler is exited, the STOP instruction is executed, loading the SR with the immediate operand from the instruction. <br> 3 The processor then generates a trace exception. The PC in the exception stack frame points to the instruction after STOP, and the SR reflects the value loaded in the previous step. <br> If the processor is not in trace mode and executes a STOP instruction where the immediate operand sets SR[T], hardware loads the SR and generates a trace exception. The PC in the exception stack frame points to the instruction after STOP, and the SR reflects the value loaded in step 2 . Note that because ColdFire processors do not support hardware stacking of multiple exceptions, it is the responsibility of the operating system to check for trace mode after processing other exception types. For example, when a TRAP instruction executes in trace mode, the processor initiates the TRAP exception and passes control to the corresponding handler. If the system requires a trace exception, the TRAP exception handler must check for this condition (SR[15] in the exception stack frame set) and pass control to the trace handler before returning from the original exception. |

Table 11-3. Exceptions (Continued)

| Vector <br> Number | Type | Description |
| :---: | :---: | :---: |
| 10 | Unimplemented line-a opcode | A line-a opcode results when bits [15:12] of the opword are 1010. This exception is generated by the attempted execution of an undefined line-a opcode as well as under the following conditions: <br> - On an early V2 core or device $(5202,5204,5206)$ when attempting to execute a MAC or EMAC instruction. <br> - On a later V2 core or device $(5206 e, 5272)$ when attempting to execute an EMAC instruction. <br> - On an early V3 core or device (5307) when attempting to execute an EMAC instruction. |
| 11 | Unimplemented line-f opcode | A line-f opcode results when bits [15:12] of the opword are 1111. This exception is generated under the following conditions: <br> - When attempting to execute an undefined line-f opcode. <br> - When attempting to execute an FPU instruction when the FPU is not present. <br> - When attempting to execute an FPU instruction when the FPU is present but has been disabled in the CACR. |
| 12,13 | Debug | The debug interrupt exception is caused by a hardware breakpoint register trigger. Rather than generating an IACK cycle, the processor internally calculates the vector number (12 for V2 and V3; 12 or 13, depending on the type of breakpoint trigger for V4). Additionally, SR[M,I] are unaffected by the interrupt. <br> On V4, separate exception vectors are provided for PC breakpoints (vector 13) and for address/data breakpoints (vector 12). In the case of a two-level trigger, the last breakpoint determines the vector. The two unique entries occur when a PC breakpoint generates the $0 \times 034$ vector. |
| 14 | Format error | When an RTE instruction executes, the processor first examines the 4-bit format field to validate the frame type. For a ColdFire processor, attempted execution of an RTE where the format is not equal to $\{4,5,6,7\}$ generates a format error. The exception stack frame for the format error is created without disturbing the original exception frame and the stacked PC points to RTE. The selection of the format value provides limited debug support for porting code from M68000 applications. On M68000 Family processors, the SR was at the top of the stack. Bit 30 of the longword addressed by the system stack pointer is typically zero. Attempting an RTE using this old format generates a format error on a ColdFire processor. If the format field defines a valid type, the processor does the following: <br> 1 Reloads the SR operand. <br> 2 Fetches the second longword operand. <br> 3 Adjusts the stack pointer by adding the format value to the auto-incremented address after the first longword fetch. <br> 4 Transfers control to the instruction address defined by the second longword operand in the stack frame. <br> When the processor executes a FRESTORE instruction, if the restored FPU state frame contains a nonsupported value, execution is aborted and a format error exception is generated. |
| $\begin{gathered} 15, \\ 24-31, \\ 64-255 \end{gathered}$ | Interrupt | Interrupt exception processing, with interrupt recognition and vector fetching, includes uninitialized and spurious interrupts as well as those where the requesting device supplies the 8-bit interrupt vector. |
| 32-47 | Trap | Executing a Trap instruction always forces an exception and is useful for implementing system calls. The trap instruction may be used to change from user to supervisor mode. |

Table 11-3. Exceptions (Continued)

| Vector <br> Number | Type | Description |
| :---: | :---: | :--- |
| $48-55$ | Floating-point | See Section 11.1.4, "Floating-Point Arithmetic Exceptions." |
| 61 | Unsupported <br> instruction | Executing a valid DIV, MAC, or EMAC instruction when the required optional hardware <br> module is not present can generate a non-supported instruction exception. Control is <br> then passed to an exception handler that can then process the opcode as required by <br> the system. This exception can be generated by the attempted execution of DIV, MAC, <br> or EMAC instructions as follows: <br> 1 On newer V2 cores without DIV, MAC, or EMAC units. <br> 2 On newer V3 cores without a MAC or EMAC unit. (The divide unit is not optional on <br> V3.) <br> 3 On newer V4 cores attempting EMAC instructions without an EMAC unit. (The MAC <br> and divide units are not optional on V4, although the MAC unit can be replaced with an <br> EMAC.) <br> Note that this exception will never be generated by the current ColdFire standard <br> products. The 5202, 5204, and 5206 do not support this exception. The 5206e, 5272, <br> 5307, and 5407 all have divide and MAC units. All of these devices will generate an <br> unimplemented line-a exception if an EMAC instruction is attempted. |

### 11.1.4 Floating-Point Arithmetic Exceptions

This section describes floating-point arithmetic exceptions; Table 11-4 lists these exceptions in order of priority:

Table 11-4. 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) |
| 7 | Divide-by-zero (DZ) |
| 8 | Inexact (INEX) |

Most floating-point exceptions are taken when the next floating-point arithmetic instruction is encountered (this is called a pre-instruction exception). Exceptions set during a floating-point store to memory or to an integer register are taken immediately (post-instruction exception).

Note that FMOVE is considered an arithmetic instruction because the result is rounded. Only FMOVE with any destination other than a floating-point register (sometimes called FMOVE OUT) can generate post-instruction exceptions. Post-instruction exceptions never write the destination. After a post-instruction exception, processing continues with the next instruction.

## Freescale Semiconductor, Inc.

A floating-point arithmetic exception becomes pending when the result of a floating-point instruction sets an FPSR[EXC] bit and the corresponding FPCR[ENABLE] bit is set. A user write to the FPSR or FPCR that causes the setting of an exception bit in FPSR[EXC] along with its corresponding exception enabled in FPCR, leaves the FPU in an exception-pending state. The corresponding exception is taken at the start of the next arithmetic instruction as a pre-instruction exception.

Executing a single instruction can generate multiple exceptions. When multiple exceptions occur with exceptions enabled for more than one exception class, the highest priority exception is reported and taken. It is up to the exception handler to check for multiple exceptions. The following multiple exceptions are possible:

- Operand error (OPERR) and inexact result (INEX)
- Overflow (OVFL) and inexact result (INEX)
- Underflow (UNFL) and inexact result (INEX)
- Divide-by-zero (DZ) and inexact result (INEX)
- Input denormalized number (IDE) and inexact result (INEX)
- Input not-a-number (INAN) and input denormalized number (IDE)

In general, all exceptions behave similarly. If the exception is disabled when the exception condition exists, no exception is taken, a default result is written to the destination (except for BSUN exception, which has no destination), and execution proceeds normally.

If an enabled exception occurs, the same default result above is written for pre-instruction exceptions but no result is written for post-instruction exceptions.

An exception handler is expected to execute FSAVE as its first floating-point instruction. This also clears FPCR, which keeps exceptions from occurring during the handler. Because the destination is overwritten for floating-point register destinations, the original floating-point destination register value is available for the handler on the FSAVE state frame. The address of the instruction that caused the exception is available in the FPIAR. When the handler is done, it should clear the appropriate FPSR exception bit on the FSAVE state frame, then execute FRESTORE. If the exception status bit is not cleared on the state frame, the same exception occurs again.

Alternatively, instead of executing FSAVE, an exception handler could simply clear appropriate FPSR exception bits, optionally alter FPCR, and then return from the exception. Note that exceptions are never taken on FMOVE to or from the status and control registers and FMOVEM to or from the floating-point data registers.

At the completion of the exception handler, the RTE instruction must be executed to return to normal instruction flow.

### 11.1.5 Branch/Set on Unordered (BSUN)

A BSUN results from performing an IEEE nonaware conditional test associated with the FBcc instruction when an unordered condition is present. Any pending floating-point exception is first handled by a pre-instruction exception, after which the conditional instruction restarts. The conditional predicate is evaluated and checked for a BSUN exception before executing the conditional instruction. A BSUN exception occurs if the conditional predicate is an IEEE non-aware branch and FPCC[NAN] is set. When this condition is detected, FPSR[BSUN] is set. Table 11-5 shows the results when the exception is enabled or disabled.

Table 11-5. BSUN Exception Enabled/Disabled Results

| Condition | BSUN |  |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | The floating-point condition is evaluated as if it were the equivalent IEEE-aware conditional <br> predicate. No exceptions are taken. |
| Exception <br> Enabled | 1 | The processor takes a floating-point pre-instruction exception. <br> The BSUN exception is unique in that the exception is taken before the conditional predicate is <br> evaluated. If the user BSUN exception handler fails to update the PC to the instruction after the <br> excepting instruction when returning, the exception executes again. Any of the following actions <br> prevent taking the exception again: <br> - Clearing FPSR[NAN] <br> - Disabling FPCR[BSUN] <br> - Incrementing the stored PC in the stack bypasses the conditional instruction. This applies to <br> situations where fall-through is desired. Note that to accurately calculate the PC increment <br> requires knowledge of the size of the bypassed conditional instruction. |

### 11.1.6 Input Not-A-Number (INAN)

The INAN exception is a mechanism for handling a user-defined, non-IEEE data type. If either input operand is a NAN, FPSR[INAN] is set. By enabling this exception, the user can override the default action taken for NAN operands. Because FMOVEM, FMOVE FPCR, and FSAVE instructions do not modify status bits, they cannot generate exceptions. Therefore, these instructions are useful for manipulating INANs. See Table 11-6.

Table 11-6. INAN Exception Enabled/Disabled Results

| Condition | INAN | Description |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | If the destination data format is single- or double-precision, a NAN is generated with a mantissa <br> of all ones and a sign of zero transferred to the destination. If the destination data format is B, <br> W, or L, a constant of all ones is written to the destination. |
| Exception <br> enabled | 1 | The result written to the destination is the same as the exception disabled case, unless the <br> exception occurs on a FMOVE OUT, in which case the destination is unaffected. |

### 11.1.7 Input Denormalized Number (IDE)

The input denorm bit, FPCR[IDE], provides software support for denormalized operands. When the IDE exception is disabled, the operand is treated as zero, FPSR[INEX] is set, and the operation proceeds. When the IDE exception is enabled and an operand is

## Freescale Semiconductor, Inc.

denormalized, an IDE exception is taken but FPSR[INEX] is not set to allow the handler to set it appropriately. See Table 11-7.

Note that the FPU never generates denormalized numbers. If necessary, software can create them in the underflow exception handler.

Table 11-7. IDE Exception Enabled/Disabled Results

| Condition | IDE | Description |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | Any denormalized operand is treated as zero, FPSR[INEX] is set, and the operation proceeds. |
| Exception <br> enabled | 1 | The result written to the destination is the same as the exception disabled case unless the <br> exception occurs on a FMOVE OUT, in which case the destination is unaffected. FPSR[INEX] is <br> not set to allow the handler to set it appropriately. |

### 11.1.8 Operand Error (OPERR)

The operand error exception encompasses problems arising in a variety of operations, including errors too infrequent or trivial to merit a specific exceptional condition. Basically, an operand error occurs when an operation has no mathematical interpretation for the given operands. Table 11-8 lists possible operand errors. When one occurs, FPSR[OPERR] is set.

Table 11-8. Possible Operand Errors

| Instruction | Condition Causing Operand Error |
| :--- | :--- |
| FADD | $[(+\infty)+(-\infty)]$ or $[(-\infty)+(+\infty)]$ |
| FDIV | $(0 \div 0)$ or $(\infty \div \infty)$ |
| FMOVE OUT (to B, W, or L) | Integer overflow, source is NAN or $\pm \infty$ |
| FMUL | One operand is 0 and the other is $\pm \infty$ |
| FSQRT | Source is $<0$ or $-\infty$ |
| FSUB | $[(+\infty)-(+\infty)]$ or $[(-\infty)-(-\infty)]$ |

Table 11-9 describes results when the exception is enabled and disabled.
Table 11-9. OPERR Exception Enabled/Disabled Results

| Condition | OPERR | Description |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | When the destination is a floating-point data register, the result is a double-precision NAN, <br> with its mantissa set to all ones and the sign set to zero (positive). <br> For a FMOVE OUT instruction with the format S or D, an OPERR exception is impossible. With <br> the format B, W, or L, an OPERR exception is possible only on a conversion to integer <br> overflow, or if the source is either an infinity or a NAN. On integer overflow and infinity source <br> cases, the largest positive or negative integer that can fit in the specified destination size (B, <br> W, or L) is stored. In the NAN source case, a constant of all ones is written to the destination. |
| Exception <br> enabled | 1 | The result written to the destination is the same as for the exception disabled case unless the <br> exception occurred on a FMOVE OUT, in which case the destination is unaffected. If desired, <br> the user OPERR handler can overwrite the default result. |

## Freescale Semiconductor, Inc.

### 11.1.9 Overflow (OVFL)

An overflow exception is detected for arithmetic operations in which the destination is a floating-point data register or memory when the intermediate result's exponent is greater than or equal to the maximum exponent value of the selected rounding precision. Overflow occurs only when the destination is S- or D-precision format; overflows for other formats are handled as operand errors. At the end of any operation that could potentially overflow, the intermediate result is checked for underflow, rounded, and then checked for overflow before it is stored to the destination. If overflow occurs, FPSR[OVFL,INEX] are set.

Even if the intermediate result is small enough to be represented as a double-precision number, an overflow can occur if the magnitude of the intermediate result exceeds the range of the selected rounding precision format. See Table 11-10.

Table 11-10. OVFL Exception Enabled/Disabled Results

| Condition | OVFL | Description |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | The values stored in the destination based on the rounding mode defined in FPCR[MODE]. <br> RN <br> RZ <br> Infinity, with the sign of the intermediate result. <br> Largest magnitude number, with the sign of the intermediate result. |
| RMFor positive overflow, largest positive normalized number <br> For negative overflow, $-\infty$. |  |  |
| RPFor positive overflow, + <br> For negative overflow, largest negative normalized number. |  |  |
| Exception <br> enabled | 1 | The result written to the destination is the same as for the exception disabled case unless the <br> exception occurred on a FMOVE OUT, in which case the destination is unaffected. If desired, the <br> user OVFL handler can overwrite the default result. |

### 11.1.10 Underflow (UNFL)

An underflow exception occurs when the intermediate result of an arithmetic instruction is too small to be represented as a normalized number in a floating-point register or memory using the selected rounding precision, that is, when the intermediate result exponent is less than or equal to the minimum exponent value of the selected rounding precision. Underflow can only occur when the destination format is single or double precision. When the destination is byte, word, or longword, the conversion underflows to zero without causing an underflow or an operand error. At the end of any operation that could underflow, the intermediate result is checked for underflow, rounded, and checked for overflow before it is stored in the destination. FPSR[UNFL] is set if underflow occurs. If the underflow exception is disabled, FPSR[INEX] is also set.

Even if the intermediate result is large enough to be represented as a double-precision number, an underflow can occur if the magnitude of the intermediate result is too small to be represented in the selected rounding precision. Table 11-11 shows results when the exception is enabled or disabled.

Table 11-11. UNFL Exception Enabled/Disabled Results

| Condition | UNFL | Description |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | The stored result is defined below. The UNFL exception also sets FPSR[INEX] if the UNFL <br> exception is disabled. <br> RN |
| Zero, with the sign of the intermediate result |  |  |
| RZZero, with the sign of the intermediate result <br> RM <br> For positive underflow, + 0 <br> For negative underflow, smallest negative normalized number <br> RP <br> For positive underflow, smallest positive normalized number <br> For negative underflow, - 0 |  |  |
| Exception <br> enabled | 1 | The result written to the destination is the same as for the exception disabled case, unless the <br> exception occurs on a FMOVE OUT, in which case the destination is unaffected. If desired, the <br> user UNFL handler can overwrite the default result. The UNFL exception does not set <br> FPSR[INEX] if the UNFL exception is enabled so the exception handler can set FPSR[INEX] <br> based on results it generates. |

### 11.1.11 Divide-by-Zero (DZ)

Attempting to use a zero divisor for a divide instruction causes a divide-by-zero exception. When a divide-by-zero is detected, FPSR[DZ] is set. Table 11-12 shows results when the exception is enabled or disabled.

Table 11-12. DZ Exception Enabled/Disabled Results

| Condition | $\mathbf{D Z}$ | Description |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | The destination floating-point data register is written with infinity with the sign set to the exclusive <br> OR of the signs of the input operands. |
| Exception <br> enabled | 1 | The destination floating-point data register is written as in the exception is disabled case. |

### 11.1.12 Inexact Result (INEX)

An INEX exception condition exists when the infinitely precise mantissa of a floating-point intermediate result has more significant bits than can be represented exactly in the selected rounding precision or in the destination format. If this condition occurs, FPSR[INEX] is set and the infinitely precise result is rounded according to Table 11-13.

## Table 11-13. Inexact Rounding Mode Values

| Mode | Result |
| :---: | :--- |
| RN | The representable value nearest the infinitely precise intermediate value is the result. If the two nearest <br> representable values are equally near, the one whose Isb is 0 (even) is the result. This is sometimes called <br> round-to-nearest-even. |
| RZ | The result is the value closest to and no greater in magnitude than the infinitely precise intermediate result. <br> This is sometimes called chop-mode, because the effect is to clear bits to the right of the rounding point. |
| RM | The result is the value closest to and no greater than the infinitely precise intermediate result (possibly $-\infty$ ). |
| RP | The result is the value closest to and no less than the infinitely precise intermediate result (possibly $+\infty$ ). |

## Freescale Semiconductor, Inc.

FPSR[INEX] is also set for any of the following conditions:

- If an input operand is a denormalized number and the IDE exception is disabled
- An overflowed result
- An underflowed result with the underflow exception disabled

Table 11-14 shows results when the exception is enabled or disabled.
Table 11-14. INEX Exception Enabled/Disabled Results

| Condition | INEX | Description |
| :---: | :---: | :--- |
| Exception <br> disabled | 0 | The result is rounded and then written to the destination. |
| Exception <br> enabled | 1 | The result written to the destination is the same as for the exception disabled case, unless the <br> exception occurred on a FMOVE OUT, in which case the destination is unaffected. If desired, the <br> user INEX handler can overwrite the default result. |

### 11.1.13 V4 Changes to the Exception Processing Model

When an MMU is present in a ColdFire device, all memory references require support for precise, recoverable faults. This section details the changes in the ColdFire exception processing model due to the presence of an MMU.

The ColdFire instruction restart mechanism ensures that a faulted instruction restarts from the beginning of execution; that is, no internal state information is saved when an exception occurs and none is restored when the handler ends. Given the PC address defined in the exception stack frame, the processor reestablishes program execution by transferring control to the given location as part of the RTE (return from exception) instruction.

The instruction restart recovery model requires program-visible register changes made during execution to be undone if that instruction subsequently faults.

The V4 Operand Execution Pipeline (OEP) structure naturally supports this concept for most instructions; program-visible registers are updated only in the final OEP stage when fault collection is complete. If any type of exception occurs, pending register updates are discarded.

For V4 cores, most single-cycle instructions already support precise faults and instruction restart. Some complex instructions do not. Consider the following memory-to-memory move:

$$
\text { mov.l }(A y)+,(A x)+\quad \# \text { copy } 4 \text { bytes from source to destination }
$$

On a V4 processor, this instruction takes 1 cycle to read the source operand (Ay) and 1 to write the data into (Ax). Both the source and destination address pointers are updated as part of execution. Table 11-15 lists the operations performed in execute stage (EX).

Table 11-15. OEP EX Cycle Operations

| EX Cycle | Operations |
| :---: | :---: |
| 1 | Read source operand from memory @ (Ay), update Ay, new Ay = old Ay + 4 |
| 2 | Write operand into destination memory @ (Ax), update Ax, new Ax = old Ax + 4, update CCR |

A fault detected with the destination memory write is reported during the second cycle. At this point, operations performed in the first cycle are complete, so if the destination write takes any type of access error, Ay is updated. After the access error handler executes and the faulting instruction restarts, the processor's operation is incorrect because the source address register has an incorrect (post-incremented) value.

To recover the original state of the programming model for all instructions, the V4 core adds the needed hardware to support full register recovery. This hardware allows program-visible registers to be restored to their original state for multi-cycle instructions so that the instruction restart mechanism is supported. Memory-to-memory moves and move multiple loads are representative of the complex instructions needing the special recovery support.

## Chapter 12 Processor Instruction Summary

This chapter provides a quick reference of the ColdFire instructions. Table 12-2 lists the ColdFire instructions by mnemonic, the descriptive name, and the cores that support them. The Version 2 and 3 cores (V2 and V3) support ISA_A, and the Version 4 core (V4) supports ISA_B.

Table 12-3 lists the instructions supported by the optional MAC unit (both fractional and integer only) and the optional enhanced MAC unit (EMAC). Table 12-4 lists the instructions supported by the optional floating-point unit (FPU).

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 12-1.

Table 12-1. Standard Products

| Standard Product | Core/ISA | Optional Modules |
| :---: | :---: | :---: |
| 5202 | V2, ISA_A |  |
| 5204 | V2, ISA_A |  |
| 5206 | V2, ISA_A |  |
| 5206 e | V2, ISA_A | Divide, MAC |
| 5272 | V2, ISA_A | Divide, MAC $^{2}$ |
| 5307 | V3, ISA_A | MAC (fractional) $^{1}$ |
| 5407 | V4, ISA_B | MAC (fractional) $^{1}$ |

1 Divide is a required module for V 3 and V 4 .

## Freescale Semiconductor, Inc.

Table 12-2. ColdFire Instruction Set and Processor Cross-Reference

| Mnemonic | Description | V2 | V3 | V4 |
| :---: | :---: | :---: | :---: | :---: |
| ADD | Add | X | X | X |
| ADDA | Add Address | X | X | X |
| ADDI | Add Immediate | X | X | X |
| ADDQ | Add Quick | X | X | X |
| ADDX | Add with Extend | X | X | X |
| AND | Logical AND | X | X | X |
| ANDI | Logical AND Immediate | X | X | X |
| ASL, ASR | Arithmetic Shift Left and Right | X | X | X |
| Bcc. $\{\mathrm{B}, \mathrm{W}\}$ | Branch Conditionally, Byte and Word | X | X | X |
| Bcc.L | Branch Conditionally, Longword |  |  | X |
| BCHG | Test Bit and Change | X | X | X |
| BCLR | Test Bit and Clear | X | X | X |
| BRA.\{B,W $\}$ | Branch Always, Byte and Word | X | X | X |
| BRA.L | Branch Always, Longword |  |  | X |
| BSET | Test Bit and Set | X | X | X |
| BSR.\{B,W\} | Branch to Subroutine, Byte and Word | X | X | X |
| BSR.L | Branch to Subroutine, Longword |  |  | X |
| BTST | Test a Bit | X | X | X |
| CLR | Clear | X | X | X |
| CMP.\{B,W\} | Compare, Byte and Word |  |  | X |
| CMP.L | Compare, Longword | X | X | X |
| CMPA.W | Compare Address, Word |  |  | X |
| CMPA.L | Compare Address, Longword | X | X | X |
| CMPI. $\{\mathrm{B}, \mathrm{W}\}$ | Compare Immediate, Byte and Word |  |  | X |
| CMPI.L | Compare Immediate, Longword | X | X | X |
| CPUSHL | Push and Possibly Invalidate Cache | X | X | X |
| DIVS | Signed Divide | $X^{1}$ | X | X |
| DIVU | Unsigned Divide | $\mathrm{X}^{1}$ | X | X |
| EOR | Logical Exclusive-OR | X | X | X |
| EORI | Logical Exclusive-OR Immediate | X | X | X |
| EXT, EXTB | Sign Extend | X | X | X |
| HALT | Halt CPU | X | X | X |
| ILLEGAL | Take Illegal Instruction Trap | X | X | X |
| INTOUCH | Instruction Fetch Touch |  |  | X |
| JMP | Jump | X | X | X |
| JSR | Jump to Subroutine | X | X | X |

## Freescale Semiconductor, Inc.

Table 12-2. ColdFire Instruction Set and Processor Cross-Reference (Continued)

| Mnemonic | Description | V2 | V3 | V4 |
| :---: | :---: | :---: | :---: | :---: |
| LEA | Load Effective Address | X | X | X |
| LINK | Link and Allocate | X | X | X |
| LSL, LSR | Logical Shift Left and Right | X | X | X |
| MOV3Q | Move 3-Bit Data Quick |  |  | X |
| MOVE | Move | X | X | $\mathrm{X}^{2}$ |
| MOVE from CCR | Move from Condition Code Register | X | X | X |
| MOVE from SR | Move from the Status Register | X | X | X |
| MOVE from USP | Move from User Stack Pointer |  |  | $\mathrm{X}^{3}$ |
| MOVE to CCR | Move to Condition Code Register | X | X | X |
| MOVE to SR | Move to the Status Register | X | X | X |
| MOVE to USP | Move to User Stack Pointer |  |  | $\mathrm{X}^{3}$ |
| MOVEA | Move Address | X | X | X |
| MOVEC | Move Control Register | X | X | X |
| MOVEM | Move Multiple Registers | X | X | X |
| MOVEQ | Move Quick | X | X | X |
| MULS | Signed Multiply | X | X | X |
| MULU | Unsigned Multiply | X | X | X |
| MVS | Move with Sign Extend |  |  | X |
| MVZ | Move with Zero-Fill |  |  | X |
| NEG | Negate | X | X | X |
| NEGX | Negate with Extend | X | X | X |
| NOP | No Operation | X | X | X |
| NOT | Logical Complement | X | X | X |
| OR | Logical Inclusive-OR | X | X | X |
| ORI | Logical Inclusive-OR Immediate | X | X | X |
| PEA | Push Effective Address | X | X | X |
| PULSE | Generate Processor Status | X | X | X |
| REMS | Signed Divide Remainder | $X^{1}$ | X | X |
| REMU | Unsigned Divide Remainder | $\mathrm{X}^{1}$ | X | X |
| RTE | Return from Exception | X | X | X |
| RTS | Return from Subroutine | X | X | X |
| SATS | Signed Saturate |  |  | X |
| Scc | Set According to Condition | X | X | X |
| STOP | Load Status Register and Stop | X | X | X |
| SUB | Subtract | X | X | X |
| SUBA | Subtract Address | X | X | X |

## Freescale Semiconductor, Inc.

Table 12-2. ColdFire Instruction Set and Processor Cross-Reference (Continued)

| Mnemonic | Description | V2 | V3 | V4 |
| :---: | :--- | :---: | :---: | :---: |
| SUBI | Subtract Immediate | X | X | X |
| SUBQ | Subtract Quick | X | X | X |
| SUBX | Subtract with Extend | X | X | X |
| SWAP | Swap Register Words | X | X | X |
| TAS | Test and Set and Operand |  |  | X |
| TPF | Trap False | X | X | X |
| TRAP | Trap | X | X | X |
| TST | Test Operand | X | X | X |
| UNLK | Unlink | X | X | X |
| WDDATA | Write Data Control Register | X | X | X |
| WDEBUG | Write Debug Control Register | X | X | X |

1 The 5202, 5204, and 5206 do not support this instruction.
2 V4 and V4e additionally support the MOVE.\{B,W\} \#<data>, $\mathrm{d}_{16}$ (Ax)
3 The 5407 does not have an MMU and therefore does not support this instruction.
Table 12-3. ColdFire MAC and EMAC Instruction Sets

| Mnemonic | Description | MAC | EMAC |
| :---: | :--- | :---: | :---: |
| MAC | Multiply and Accumulate | X | X |
| MOVCLR | Move from Accumulator and Clear |  | X |
| MOVE ACC to ACC | Copy Accumulator | X |  |
| MOVE from ACC | Move from Accumulator | X | X |
| MOVE from ACCext01 | Move from Accumulator 0 and 1 <br> Extensions | X |  |
| MOVE from ACCext23 | Move from Accumulator 2 and 3 <br> Extensions | X | X |
| MOVE from MACSR | Move from MAC Status Register | X | X |
| MOVE from MACSR to | Move from MAC Status Register to <br> Condition Code Register | X | X |
| MOVE from MASK | Move from MAC Mask Register | X | X |
| MOVE to ACC | Move to Accumulator | X | X |
| MOVE to ACCext01 | Move to Accumulator 0 and 1 <br> Extensions | X |  |
| MOVE to ACCext23 | Move to Accumulator 2 and 3 <br> Extensions | X | X |
| MOVE to MACSR | Move to MAC Status Register | X | X |
| MOVE to MASK | Move to MAC Mask Register | X | X |
| MSAC | Multiply and Subtract | X | X |

## Freescale Semiconductor, Inc.

Table 12-4. ColdFire FPU Instruction Set

| Mnemonic | Description |
| :---: | :---: |
| FABS, FSABS, FDABS | Floating-Point Absolute Value |
| FADD, FSADD, FDADD | Floating-Point Add |
| FBcc | Floating-Point Branch Conditionally |
| FCMP | floating-Point Compare |
| FDIV, FSDIV, FDDIV | Floating-Point Divide |
| FINT, FSINT, FDINT | Floating-Point Integer |
| FINTRZ | Floating-Point Integer Round-to-Zero |
| FMOVE, FSMOVE, FDMOVE | Move Floating-Point Data Register |
| FMOVE from FPCR | Move from the Floating-Point Control Register |
| FMOVE from FPIAR | Move from the Floating-Point Instruction Address Register |
| FMOVE from FPSR | Move from the Floating-Point Status Register |
| FMOVE to FPCR | Move to the Floating-Point Control Register |
| FMOVE to FPIAR | Move to the Floating-Point Instruction Address Register |
| FMOVE to FPSR | Move to the Floating-Point Status Register |
| FMOVEM | Move Multiple Floating-Point Data Registers |
| FMUL, FSMUL, FDMUL | Floating-Point Multiply |
| FNEG, FSNEG, FDNEG | Floating-Point Negate |
| FNOP | No Operation |
| FRESTORE | Restore Internal Floating-Point State |
| FSAVE | Save Internal Floating-Point State |
| FSQRT, FSSQRT, FDSQRT | Floating-Point Square Root |
| FSUB | Floating-Point Subtract |
| FTST | Test Floating-Point Operand |

## Freescale Semiconductor, Inc.

## Appendix A S-Record Output Format

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

## A. 1 S-Record Content

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

| Type | Record Length | Address | Code/Data | Checksum |
| :---: | :---: | :---: | :---: | :---: |

Figure A-1. Five Fields of an S-Record
Table A-1. Field Composition of an S-Record

| Field | Printable <br> Characters | Contents |
| :---: | :---: | :--- |
| Type | 2 | S-record type-S0, S1, etc. |
| Record Length | 2 | The count of the character pairs in the record, excluding the type and <br> record length. |
| Address | 4,6, or 8 | The 2-, 3-, or 4-byte address at which the data field is to be loaded into <br> memory. |
| Code/Data | $0-2 \mathrm{n}$ | From 0 to n bytes of executable code, memory loadable data, or <br> descriptive information. Some programs may limit the number of bytes <br> to as few as 28 (56 printable characters in the S-record). |
| Checksum | 2 | The least significant byte of the one's complement of the sum of the <br> values represented by the pairs of characters making up the record <br> length, address, and the code/data fields. |

## Freescale Semiconductor, Inc.

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

## A. 2 S-Record Types

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

An S-record format module may contain S-records of the following types:
S0 The header record for each block of S-records. The code/data field may contain any descriptive information identifying the following block of S-records. The header record can be used to designate module name, version number, revision number, and description information. The address field is normally zeros.

S1 A record containing code/data and the 2-byte address at which the code/data is to reside.
S2 A record containing code/data and the 3-byte address at which the code/data is to reside.

S3 A record containing code/data and the 4-byte address at which the code/data is to reside.
S5 A record containing the number of S1, S2, and S3 records transmitted in a particular block. This count appears in the address field. There is no code/data field.

S7 A termination record for a block of S3 records. The address field may optionally contain the 4-byte address of the instruction to which control is to be passed. There is no code/data field.
S8 A termination record for a block of S 2 records. The address field may optionally contain the 3-byte address of the instruction to which control is to be passed. There is no code/data field.
S9 A termination record for a block of S1 records. The address field may optionally contain the 2-byte address of the instruction to which control is to be passed. If this address is not specified, the first entry point specification encountered in the object module input will be used. There is no code/data field.

Each block of S-records uses only one termination record. S7 and S8 records are only active when control passes to a 3- or 4-byte address; otherwise, an S 9 is used for termination.

Normally, there is only one header record, although it is possible for multiple header records to occur.

## A. 3 S-Record Creation

Dump utilities, debuggers, or cross assemblers and linkers produce S-record format programs. Programs are available for downloading or uploading a file in S- record format from a host system to a microprocessor-based system.

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

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

The module has an S0 record, four S1 records, and an S9 record. The following character pairs comprise the S-record format module.
S0 Record:
S0 S-record type S 0 , indicating that it is a header record
06 Hexadecimal 06 (decimal 6), indicating that six character pairs (or ASCII bytes) follow
0000 A 4-character, 2-byte address field; zeros in this example
48 ASCII H
44 ASCII D
52 ASCII R
1B The checksum
First S1 Record:
S1 S-record type S1, indicating that it is a code/data record to be loaded/verified at a 2-byte address
13 Hexadecimal 13 (decimal 19), indicating that 19 character pairs, representing 19 bytes of binary data, follow
0000 A 4-character, 2-byte address field (hexadecimal address 0000) indicating where the data that follows is to be loaded.

The next 16 character pairs of the first S1 record are the ASCII bytes of the actual program code/data. In this assembly language example, the program hexadecimal opcodes are sequentially written in the code/data fields of the S 1 records.

## Freescale Semiconductor, Inc.

## S-Record Creation

| Opcode | Instruction |
| :---: | :--- |
| 285 F | MOVE.L (A7) +, A4 |
| $245 F$ | MOVE.L (A7) +, A2 |
| 2212 | MOVE.L (A2), D1 |
| 226 A0004 | MOVE.L 4(A2), A1 |
| 24290008 | MOVE.L FUNCTION(A1), D2 |
| $237 C$ | MOVE.L \#FORCEFUNC, FUNCTION(A1) |

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

2A The checksum of the first S1 record.
The second and third S1 records also contain hexadecimal 13 (decimal 19) character pairs and end with checksums 13 and 52, respectively. The fourth S1 record contains 07 character pairs and has a checksum of 92 .

S9 Record:
S9 S-record type S 9 , indicating that it is a termination record
03 Hexadecimal 03, indicating that three character pairs (3 bytes) follow

0000 Address field, zeros
FC Checksum of the S9 record
Each printable character in an S-record encodes in hexadecimal (ASCII in this example) representation of the binary bits that transmit. Figure A-2 illustrates the sending of the first S1 record. Table A-2 lists the ASCII code for S-records.


Figure A-2. Transmission of an S1 Record

## Freescale Semiconductor, Inc.

Table A-2. ASCII Code

| Least Significant Digit | Most Significant Digit |  |  |  |  |  |  |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | NUL | DLE | SP | 0 | @ | P | - | p |
| 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | STX | DC2 | " | 2 | B | R | b | $r$ |
| 3 | ETX | DC3 | \# | 3 | C | S | C | S |
| 4 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 5 | ENQ | NAK | \% | 5 | E | U | e | u |
| 6 | ACK | SYN | \& | 6 | F | V | f | v |
| 7 | BEL | ETB | , | 7 | G | W | g | w |
| 8 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | HT | EM | ) | 9 | 1 | Y | i | y |
| A | LF | SUB | * | : | J | Z | J | z |
| B | VT | ESC | + | ; | K | [ | k | \{ |
| C | FF | FS | , | < | L | 1 | I | \| |
| D | CR | GS | - | = | M | ] | m | \} |
| E | SO | RS | . | > | N | $\wedge$ | n | $\sim$ |
| F | SI | US | 1 | ? | O | - | 0 | DEL |

## Freescale Semiconductor, Inc.

## INDEX

## A

Access control registers (ACR0-ACR3), 1-14
Accumulator
EMAC, 1-9
extensions (ACCext01, ACCext23), 1-11
Address register
direct mode, 2-3
general (A0-A7), 1-2
indirect mode
displacement, 2-5
postincrement, 2-4
predecrement, 2-4
regular, 2-3
scaled index and 8-bit displacement, 2-6
Address space identifier (ASID), 1-14

## B

Bit manipulation instructions, 3-8
Branch/set on unordered (BSUN), 11-11

## C

Cache
control register (CACR), 1-14
maintenance instructions, 3-10
Condition code register (CCR), 1-2
Conditional testing, 7-3

## D

Data formats
and type summary, 1-18
multiply accumulate, 1-20
Data movement instructions, 3-4
Data register
direct mode, 2-3
general (D0-D7), 1-2
Data, immediate, 2-9
Divide-by-zero (DZ), 11-14

## E

EMAC
accumulators, 1-9
user instructions, 6-1-6-24
user programming model, 1-8
Exception processing model V4 changes, 11-15
Exception stack frame definition, 11-4

Exceptions
floating-point arithmetic, 11-9
processor, 11-5

## F

Floating-point
arithmetic exceptions, 11-9
arithmetic instructions, 3-11
control register (FPCR), 1-4
data formats, 1-16
data registers (FP0-FP7), 1-4
data types
denormalized numbers, 1-18
not-a-number, 1-18
zeros, 1-17
instruction
address register (FPIAR), 1-6
descriptions, 7-9-7-43
status register (FPSR), 1-5, 7-1
Formats
floating-point data, 1-16
integer data, 1-16
FPU user programming model, 1-4

## I

Inexact result (INEX), 11-14
Infinities, 1-17
Input
denormalized number, 11-11
not-a-number (INAN), 11-11
Instructions
bit manipulation, 3-8
cache maintenance, 3-10
data movement, 3-4
descriptions, 7-7
floating-point arithmetic, 3-11
format, 2-1
integer arithmetic, 3-5
logical, 3-7
processor summary, 12-1
program control, 3-8
results, exceptions, 7-6
set, 12-2, 12-5
additions, 3-12
shift, 3-7
summary, 3-1
system control, 3-10

Integer arithmetic instructions, 3-5
Integer data formats
general, 1-16
in memory, 1-22
in registers, 1-20
Integer unit user programming model, 1-1
Integer user instructions, 4-1-4-84

## L

Logical instructions, 3-7

## M

MAC
accumulator (ACC), 1-8
mask register (MASK)
EMAC, 1-11
MAC, 1-8
status register (MACSR) EMAC, 1-8
MAC, 1-7
user instructions, 5-1-5-16
user programming model, 1-7
Memory integer data formats, 1-22
MMU base address register (MMUBAR), 1-14
Modes
address register
indirect
postincrement, 2-4
regular, 2-6
indirect with displacement, 2-5
addressing
absolute long, 2-9
absolute short, 2-8
direct
address register, 2-3
data register, 2-3
effective addressing, 2-2, 2-10
indirect
address register, 2-3
predecrement address register, 2-4
program counter, 2-6, 2-7
Module base address register (MBAR), 1-15
Multiply accumulate data formats, 1-20

## N

Normalized numbers, 1-17

## 0

Operand error (OPERR), 11-12
Operation code map, 9-1
Organization of data in registers, 1-20
Overflow (OVFL), 11-13

## P

Processor
cross-reference, 12-2-12-5
exceptions, 11-5
Program control instructions, 3-8
Program counter (PC)
general, 1-2
indirect
displacement, 2-6
scaled index and 8-bit displacement, 2-7
Programming model
EMAC user, 1-8
FPU user, 1-4
integer unit user, 1-1
MAC user, 1-7
supervisor, 1-11

## R

RAM base address registers
(RAMBAR0/RAMBAR1), 1-15
Registers
ABLR/ABHR, 8-18
access control (ACR0-ACR3), 1-14
address (A0-A7), 1-2
cache control (CACR), 1-14
condition code (CCR), 1-2
data (D0-D7), 1-2
data organization, 1-20
DBR/DBMR, 8-18
floating-point
control (FPCR), 1-4
data (FP0-FP7), 1-4
instruction address (FPIAR), 1-6
status, 7-1
status (FPSR), 1-5
integer data formats, 1-20
MAC mask (MASK)
EMAC, 1-11
MAC, 1-8
MAC status (MACSR)
EMAC, 1-8
MAC, 1-7
MMU base address (MMUBAR), 1-14
module base address (MBAR), 1-15
RAM base (RAMBAR0/RAMBAR1), 1-15
ROM base address (ROMBAR0/ROMBAR1), 1-15
status (SR), 1-12
vector base (VBR), 1-14, 11-2
ROM base address registers
(ROMBAR0/ROMBAR1), 1-15

## Freescale Semiconductor, Inc.

## INDEX

## S

Shift instructions, 3-7
S-record
content, A-1
creation, A-3
types, A-2
Stack, 2-10
Stack pointers supervisor/user, 1-13, 11-4
Status register (SR), 1-12
Supervisor
instruction descriptions, 8-1-8-18
instruction set, 10-7
programming model, 1-11
Supervisor/user stack pointers, 1-13, 11-4
System control instructions, 3-10

## U

Underflow (UNFL), 11-13
User instruction set, 10-1-10-7

## V

Vector base register, 1-14, 11-2

## Freescale Semiconductor, Inc.

## INDEX

MAC User Instructions

FPU User Instructions

Supervisor Instructions

Instruction Format Summary

Exception Processing

Addressing Capabilities

Instruction Set Summary

Integer User Instructions

EMAC User Instructions

7
FPU User Instructions

Supervisor Instructions

9
Instruction Format Summary

PST/DDATA Encodings

11 Exception Processing

Processor Instruction Summary

S-Record Output Format


[^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

[^1]:    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

