

**Freescale Semiconductor, Inc.**

# **RCPU RISC CENTRAL PROCESSING UNIT REFERENCE MANUAL**

Revision 1

Revised 1 February 1999

**Freescale Semiconductor, Inc.**



© Copyright 2001 MOTOROLA; All Rights Reserved

**For More Information On This Product,  
Go to: [www.freescale.com](http://www.freescale.com)**

**Freescale Semiconductor, Inc.**

**Freescale Semiconductor, Inc.**

**For More Information On This Product,  
Go to: [www.freescale.com](http://www.freescale.com)**

Paragraph Number	TABLE OF CONTENTS	Page Number
	<b>PREFACE</b>	
	<b>Section 1</b>	
	<b>OVERVIEW</b>	
1.1	RCPU Overview . . . . .	1-1
1.1.1	RCPU Features . . . . .	1-2
1.1.2	RCPU Block Diagram . . . . .	1-3
1.1.3	Instruction Sequencer . . . . .	1-5
1.1.4	Independent Execution Units . . . . .	1-6
1.1.4.1	Branch Processing Unit (BPU) . . . . .	1-7
1.1.4.2	Integer Unit (IU) . . . . .	1-7
1.1.4.3	Floating-Point Unit (FPU) . . . . .	1-8
1.1.4.4	Load/Store Unit (LSU) . . . . .	1-8
1.1.5	Instruction Cache . . . . .	1-9
1.1.6	Instruction Pipeline . . . . .	1-9
1.1.7	Development Support . . . . .	1-10
1.2	Levels of the PowerPC Architecture . . . . .	1-11
1.3	The RCPU as a PowerPC Implementation . . . . .	1-11
1.3.1	PowerPC Registers and Programming Model . . . . .	1-11
1.3.1.1	General-Purpose Registers (GPRs) . . . . .	1-12
1.3.1.2	Floating-Point Registers (FPRs) . . . . .	1-12
1.3.1.3	Condition Register (CR) . . . . .	1-12
1.3.1.4	Floating-Point Status and Control Register (FPSCR) . . . . .	1-12
1.3.1.5	Machine State Register (MSR) . . . . .	1-12
1.3.1.6	Special-Purpose Registers (SPRs) . . . . .	1-12
1.3.1.7	User-Level SPRs . . . . .	1-12
1.3.1.8	Supervisor-Level SPRs . . . . .	1-13
1.3.2	Instruction Set and Addressing Modes . . . . .	1-13
1.3.2.1	PowerPC Instruction Set . . . . .	1-14
1.3.2.2	PowerPC Addressing Modes . . . . .	1-15
1.3.2.3	RCPU Instruction Set . . . . .	1-15
1.3.3	PowerPC Exception Model . . . . .	1-16
	<b>Section 2</b>	
	<b>REGISTERS</b>	
2.1	Programming Models . . . . .	2-1
2.2	PowerPC UISA Register Set . . . . .	2-3
2.2.1	General Purpose Registers (GPRs) . . . . .	2-3
2.2.2	Floating-Point Registers (FPRs) . . . . .	2-3
2.2.3	Floating-Point Status and Control Register (FPSCR) . . . . .	2-4
2.2.4	Condition Register (CR) . . . . .	2-8
RCPU	TABLE OF CONTENTS	MOTOROLA
REFERENCE MANUAL	Rev. 1 February 1999	iii
	<b>For More Information On This Product,</b>	
	<b>Go to: <a href="http://www.freescale.com">www.freescale.com</a></b>	

Paragraph Number	Page Number
2.2.4.1 Condition Register CR0 Field Definition . . . . .	2-9
2.2.4.2 Condition Register CR1 Field Definition . . . . .	2-9
2.2.4.3 Condition Register CR $n$ Field — Compare Instruction . . . . .	2-10
2.2.5 Integer Exception Register (XER) . . . . .	2-10
2.2.6 Link Register (LR) . . . . .	2-11
2.2.7 Count Register (CTR) . . . . .	2-12
2.3 PowerPC VEA Register Set — Time Base . . . . .	2-12
2.4 PowerPC OEA Register Set . . . . .	2-13
2.4.1 Machine State Register (MSR) . . . . .	2-13
2.4.2 DAE/Source Instruction Service Register (DSISR) . . . . .	2-16
2.4.3 Data Address Register (DAR) . . . . .	2-16
2.4.4 Time Base Facility (TB) — OEA . . . . .	2-16
2.4.5 Decrementer Register (DEC) . . . . .	2-17
2.4.6 Machine Status Save/Restore Register 0 (SRR0) . . . . .	2-18
2.4.7 Machine Status Save/Restore Register 1 (SRR1) . . . . .	2-19
2.4.8 General SPRs (SPRG0–SPRG3) . . . . .	2-19
2.4.9 Processor Version Register (PVR) . . . . .	2-20
2.4.10 Implementation-Specific SPRs . . . . .	2-20
2.4.10.1 EIE, EID, and NRI Special-Purpose Registers . . . . .	2-20
2.4.10.2 Instruction-Cache Control Registers . . . . .	2-21
2.4.10.3 Development Support Registers . . . . .	2-21
2.4.10.4 Floating-Point Exception Cause Register (FPECR) . . . . .	2-22

## Section 3 OPERAND CONVENTIONS

3.1 Data Alignment and Memory Organization . . . . .	3-1
3.2 Byte Ordering . . . . .	3-2
3.2.1 Structure Mapping Examples . . . . .	3-3
3.2.1.1 Big-Endian Mapping . . . . .	3-3
3.2.1.2 Little-Endian Mapping . . . . .	3-4
3.2.2 Data Memory in Little-Endian Mode . . . . .	3-4
3.2.2.1 Aligned Scalars . . . . .	3-4
3.2.2.2 Misaligned Scalars . . . . .	3-6
3.2.2.3 String Operations . . . . .	3-7
3.2.2.4 Load and Store Multiple Instructions . . . . .	3-8
3.2.3 Instruction Memory Addressing in Little-Endian Mode . . . . .	3-8
3.2.4 Input/Output in Little-Endian Mode . . . . .	3-10
3.3 Floating-Point Data . . . . .	3-10
3.3.1 Floating-Point Data Format . . . . .	3-10
3.3.2 Value Representation . . . . .	3-12
3.3.3 Normalized Numbers ( $\pm$ NORM) . . . . .	3-13
3.3.4 Zero Values ( $\pm 0$ ) . . . . .	3-14
3.3.5 Denormalized Numbers ( $\pm$ DENORM) . . . . .	3-14

<b>Paragraph Number</b>	<b>Page Number</b>
3.3.6 Infinities ( $\pm\infty$ ) . . . . .	3-15
3.3.7 Not a Numbers (NaNs) . . . . .	3-15
3.3.8 Sign of Result . . . . .	3-16
3.3.9 Normalization and Denormalization . . . . .	3-17
3.3.10 Data Handling and Precision . . . . .	3-17
3.3.11 Rounding . . . . .	3-19
3.4 Floating-Point Execution Models . . . . .	3-21
3.4.1 Execution Model for IEEE Operations . . . . .	3-22
3.4.2 Execution Model for Multiply-Add Type Instructions . . . . .	3-24
3.4.3 Non-IEEE Operation . . . . .	3-25
3.4.4 Working Without the Software Envelope . . . . .	3-26

## **Section 4**

### **ADDRESSING MODES AND INSTRUCTION SET SUMMARY**

4.1 Memory Addressing . . . . .	4-1
4.1.1 Memory Operands . . . . .	4-2
4.1.2 Addressing Modes and Effective Address Calculation . . . . .	4-2
4.2 Classes of Instructions . . . . .	4-3
4.2.1 Definition of Boundedly Undefined . . . . .	4-3
4.2.2 Defined Instruction Class . . . . .	4-3
4.2.3 Illegal Instruction Class . . . . .	4-4
4.2.4 Reserved Instruction Class . . . . .	4-4
4.3 Integer Instructions . . . . .	4-4
4.3.1 Integer Arithmetic Instructions . . . . .	4-5
4.3.2 Integer Compare Instructions . . . . .	4-11
4.3.3 Integer Logical Instructions . . . . .	4-12
4.3.4 Integer Rotate and Shift Instructions . . . . .	4-14
4.3.4.1 Integer Rotate Instructions . . . . .	4-16
4.3.4.2 Integer Shift Instructions . . . . .	4-17
4.4 Floating-Point Instructions . . . . .	4-19
4.4.1 Floating-Point Arithmetic Instructions . . . . .	4-19
4.4.2 Floating-Point Multiply-Add Instructions . . . . .	4-22
4.4.3 Floating-Point Rounding and Conversion Instructions . . . . .	4-25
4.4.4 Floating-Point Compare Instructions . . . . .	4-27
4.4.5 Floating-Point Status and Control Register Instructions . . . . .	4-28
4.5 Load and Store Instructions . . . . .	4-30
4.5.1 Integer Load and Store Address Generation . . . . .	4-30
4.5.1.1 Register Indirect with Immediate Index Addressing . . . . .	4-30
4.5.1.2 Register Indirect with Index Addressing . . . . .	4-31
4.5.1.3 Register Indirect Addressing . . . . .	4-32
4.5.2 Integer Load Instructions . . . . .	4-33
4.5.3 Integer Store Instructions . . . . .	4-36
4.5.4 Integer Load and Store with Byte Reversal Instructions . . . . .	4-37

Paragraph Number	Page Number
4.5.5 Integer Load and Store Multiple Instructions . . . . .	4-38
4.5.6 Integer Move String Instructions . . . . .	4-39
4.5.7 Floating-Point Load and Store Address Generation . . . . .	4-41
4.5.7.1 Register Indirect with Immediate Index Addressing . . . . .	4-41
4.5.7.2 Register Indirect with Index Addressing . . . . .	4-41
4.5.8 Floating-Point Load Instructions . . . . .	4-42
4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions . . . . .	4-43
4.5.8.2 Floating-Point Load Single Operands . . . . .	4-44
4.5.9 Floating-Point Store Instructions . . . . .	4-44
4.5.9.1 Double-Precision Conversion for Floating-Point Store Instructions . . . . .	4-46
4.5.9.2 Floating-Point Store-Single Operands . . . . .	4-47
4.5.10 Floating-Point Move Instructions . . . . .	4-47
4.6 Flow Control Instructions . . . . .	4-48
4.6.1 Branch Instruction Address Calculation . . . . .	4-49
4.6.1.1 Branch Relative Address Mode . . . . .	4-49
4.6.1.2 Branch Conditional Relative Address Mode . . . . .	4-50
4.6.1.3 Branch to Absolute Address Mode . . . . .	4-51
4.6.1.4 Branch Conditional to Absolute Address Mode . . . . .	4-52
4.6.1.5 Branch Conditional to Link Register Address Mode . . . . .	4-52
4.6.1.6 Branch Conditional to Count Register . . . . .	4-53
4.6.2 Conditional Branch Control . . . . .	4-54
4.6.2.1 BO Operand and Branch Prediction . . . . .	4-54
4.6.2.2 BI Operand . . . . .	4-56
4.6.2.3 Simplified Mnemonics for Conditional Branches . . . . .	4-56
4.6.3 Branch Instructions . . . . .	4-56
4.6.4 Condition Register Logical Instructions . . . . .	4-57
4.6.5 System Linkage Instructions . . . . .	4-58
4.6.6 Simplified Mnemonics for Branch and Flow Control Instructions . . . . .	4-59
4.6.7 Trap Instructions . . . . .	4-59
4.7 Processor Control Instructions . . . . .	4-60
4.7.1 Move to/from Machine State Register and Condition Register Instructions . . . . .	4-60
4.7.2 Move to/from Special Purpose Register Instructions . . . . .	4-61
4.7.3 Move from Time Base Instruction . . . . .	4-64
4.8 Memory Synchronization Instructions . . . . .	4-65
4.9 Memory Control Instructions . . . . .	4-68
4.10 Recommended Simplified Mnemonics . . . . .	4-68

## Section 5 INSTRUCTION CACHE

5.1 Instruction Cache Organization . . . . .	5-1
5.2 Programming Model . . . . .	5-3
5.2.1 I-Cache Control and Status Register (ICCST) . . . . .	5-3
5.2.2 I-Cache Address Register (ICADR) . . . . .	5-5

<b>Paragraph Number</b>	<b>Page Number</b>
5.2.3 I-Cache Data Register (ICDAT) .....	5-5
5.3 Instruction Cache Operation .....	5-5
5.3.1 Cache Hit .....	5-6
5.3.2 Cache Miss .....	5-6
5.3.3 Instruction Fetch on a Predicted Path .....	5-6
5.4 Cache Commands .....	5-7
5.4.1 Instruction Cache Block Invalidate .....	5-7
5.4.2 Invalidate All .....	5-8
5.4.3 Load and Lock .....	5-8
5.4.4 Unlock Line .....	5-9
5.4.5 Unlock All .....	5-9
5.4.6 Cache Enable .....	5-9
5.4.7 Cache Disable .....	5-9
5.4.8 Cache Inhibit .....	5-9
5.4.9 Cache Read .....	5-10
5.5 I-Cache and On-Chip Memories with Zero Wait States .....	5-11
5.6 Cache Coherency .....	5-11
5.7 Updating Code and Attributes of Memory Regions .....	5-11
5.8 Reset Sequence .....	5-11
5.9 Debugging Support .....	5-12
5.9.1 Running a Debug Routine from the I-Cache .....	5-12
5.9.2 Instruction Fetch from the Development Port .....	5-12

## Section 6 EXCEPTIONS

6.1 Exception Classes .....	6-2
6.1.1 Ordered and Unordered Exceptions .....	6-2
6.1.2 Synchronous, Precise Exceptions .....	6-2
6.1.3 Asynchronous Exceptions .....	6-4
6.1.3.1 Asynchronous, Maskable Exceptions .....	6-5
6.1.3.2 Asynchronous, Non-Maskable Exceptions .....	6-5
6.2 Exception Vector Table .....	6-5
6.3 Precise Exception Model Implementation .....	6-7
6.4 Implementation of Asynchronous Exceptions .....	6-8
6.5 Recovery from Exceptions .....	6-9
6.5.1 Recovery from Ordered Exceptions .....	6-9
6.5.2 Recovery from Unordered Exceptions .....	6-9
6.5.3 Commands to Alter MSR[EE] and MSR[RI] .....	6-10
6.6 Exception Order and Priority .....	6-10
6.7 Ordering of Synchronous, Precise Exceptions .....	6-12
6.8 Exception Processing .....	6-13
6.8.1 Enabling and Disabling Exceptions .....	6-16
6.8.2 Steps for Exception Processing .....	6-16

Paragraph Number	Page Number
6.8.3 DAR, DSISR, and BAR Operation .....	6-17
6.8.4 Returning from Supervisor Mode .....	6-18
6.9 Process Switching .....	6-18
6.10 Exception Timing .....	6-18
6.11 Exception Definitions .....	6-20
6.11.1 Reset Exception (0x0100) .....	6-20
6.11.2 Machine Check Exception (0x00200) .....	6-21
6.11.2.1 Machine Check Exception Enabled .....	6-21
6.11.2.2 Checkstop State .....	6-22
6.11.2.3 Machine-Check Exceptions and Debug Mode .....	6-22
6.11.3 External Interrupt (0x00500) .....	6-22
6.11.4 Alignment Exception (0x00600) .....	6-23
6.11.4.1 Interpretation of the DSISR as Set by an Alignment Exception .....	6-24
6.11.5 Program Exception (0x00700) .....	6-26
6.11.6 Floating-Point Unavailable Exception (0x00800) .....	6-28
6.11.7 Decrementer Exception (0x00900) .....	6-29
6.11.8 System Call Exception (0x00C00) .....	6-29
6.11.9 Trace Exception (0x00D00) .....	6-30
6.11.10 Floating-Point Assist Exception (0x00E00) .....	6-31
6.11.10.1 Floating-Point Software Envelope .....	6-31
6.11.10.2 Floating-Point Assist for Denormalized Operands .....	6-32
6.11.10.3 Synchronized Ignore Exceptions (SIE) Mode .....	6-34
6.11.10.4 Floating-Point Exception Cause Register .....	6-34
6.11.10.5 Floating-Point Enabled Exceptions .....	6-36
6.11.10.6 Invalid Operation Exception Conditions .....	6-42
6.11.10.7 Zero Divide Exception Condition .....	6-43
6.11.10.8 Overflow Exception Condition .....	6-44
6.11.10.9 Underflow Exception Condition .....	6-44
6.11.10.10 Inexact Exception Condition .....	6-45
6.11.11 Software Emulation Exception (0x01000) .....	6-46
6.11.12 Data Breakpoint Exception (0x01C00) .....	6-47
6.11.13 Instruction Breakpoint Exception (0x01D00) .....	6-48
6.11.14 Maskable External Breakpoint Exception (0x01E00) .....	6-49
6.11.15 Non-Maskable External Breakpoint Exception (0x01F00) .....	6-49

## Section 7 INSTRUCTION TIMING

7.1 Instruction Flow .....	7-1
7.1.1 Instruction Sequencer Data Path .....	7-2
7.1.2 Instruction Issue .....	7-3
7.1.3 Basic Instruction Pipeline .....	7-3
7.2 Execution Unit Timing Details .....	7-5
7.2.1 Integer Unit (IU) .....	7-5



**Freescale Semiconductor, Inc.**

<b>Paragraph Number</b>	<b>Page Number</b>
7.2.1.1 Update of the XER During Divide Instructions . . . . .	7-6
7.2.2 Floating Point Unit (FPU) . . . . .	7-6
7.2.3 Load/Store Unit (LSU) . . . . .	7-6
7.2.3.1 Load/Store Instruction Issue . . . . .	7-7
7.2.3.2 Load/Store Synchronizing Instructions . . . . .	7-7
7.2.3.3 Load/Store Instruction Timing Summary . . . . .	7-7
7.2.3.4 Bus Cycles for String Instructions . . . . .	7-8
7.2.3.5 Stalls During Floating-Point Store Instructions . . . . .	7-8
7.2.4 Branch Processing Unit (BPU) . . . . .	7-9
7.3 Serialization . . . . .	7-9
7.3.1 Execution Serialization . . . . .	7-9
7.3.2 Fetch Serialization . . . . .	7-10
7.4 Context Synchronization . . . . .	7-10
7.5 Implementation of Special-Purpose Registers . . . . .	7-10
7.6 Instruction Execution Timing . . . . .	7-11
7.7 Instruction Execution Timing Examples . . . . .	7-16
7.7.1 Load from Internal Memory Example . . . . .	7-16
7.7.2 Write-Back Arbitration Examples . . . . .	7-17
7.7.3 Load with Private Write-Back Bus . . . . .	7-18
7.7.4 Fastest External Load Example . . . . .	7-19
7.7.5 History Buffer Full Example . . . . .	7-20
7.7.6 Store and Floating-Point Example . . . . .	7-21
7.7.7 Branch Folding Example . . . . .	7-22
7.7.8 Branch Prediction Example . . . . .	7-23

## Section 8

### DEVELOPMENT SUPPORT

8.1 Program Flow Tracking . . . . .	8-1
8.1.1 Indirect Change-of-Flow Cycles . . . . .	8-2
8.1.1.1 Marking the Indirect Change-of-Flow Attribute . . . . .	8-3
8.1.1.2 Sequential Instructions with the Indirect Change-of-Flow Attribute . . . . .	8-3
8.1.2 Instruction Fetch Show Cycle Control . . . . .	8-4
8.1.3 Program Flow-Tracking Pins . . . . .	8-5
8.1.3.1 Instruction Queue Status Pins . . . . .	8-5
8.1.3.2 History Buffer Flush Status Pins . . . . .	8-6
8.1.3.3 Flow-Tracking Status Pins in Debug Mode . . . . .	8-6
8.1.3.4 Cycle Type, Write/Read, and Address Type Pins . . . . .	8-7
8.1.4 External Hardware During Program Trace . . . . .	8-8
8.1.4.1 Back Trace . . . . .	8-8
8.1.4.2 Window Trace . . . . .	8-8
8.1.4.3 Synchronizing the Trace Window to Internal CPU Events . . . . .	8-8
8.1.4.4 Detecting the Trace Window Starting Address . . . . .	8-10
8.1.4.5 Detecting the Assertion or Negation of VSYNC . . . . .	8-10

Paragraph Number	Page Number
8.1.4.6 Detecting the Trace Window Ending Address . . . . .	8-10
8.1.5 Compress . . . . .	8-11
8.2 Watchpoint and Breakpoint Support . . . . .	8-11
8.2.1 Watchpoints . . . . .	8-12
8.2.1.1 Restrictions on Watchpoint Detection . . . . .	8-13
8.2.1.2 Byte and Half-Word Working Modes . . . . .	8-14
8.2.1.3 Generating Six Compare Types . . . . .	8-15
8.2.1.4 I-Bus Support Detailed Description . . . . .	8-16
8.2.1.5 L-Bus Support Detailed Description . . . . .	8-17
8.2.1.6 Treating Floating-Point Numbers . . . . .	8-19
8.2.2 Internal Breakpoints . . . . .	8-20
8.2.2.1 Breakpoint Counters . . . . .	8-20
8.2.2.2 Trap-Enable Programming . . . . .	8-20
8.2.2.3 Ignore First Match . . . . .	8-21
8.2.3 External Breakpoints . . . . .	8-21
8.2.4 Breakpoint Masking . . . . .	8-21
8.3 Development Port . . . . .	8-22
8.3.1 Development Port Signals . . . . .	8-23
8.3.1.1 Development Serial Clock . . . . .	8-23
8.3.1.2 Development Serial Data In . . . . .	8-24
8.3.1.3 Development Serial Data Out . . . . .	8-25
8.3.2 Development Port Registers . . . . .	8-25
8.3.2.1 Development Port Shift Register . . . . .	8-26
8.3.2.2 Trap Enable Control Register . . . . .	8-26
8.3.3 Development Port Clock Mode Selection . . . . .	8-26
8.3.4 Development Port Transmissions . . . . .	8-31
8.3.5 Trap-Enable Input Transmissions . . . . .	8-32
8.3.6 CPU Input Transmissions . . . . .	8-32
8.3.7 Serial Data Out of Development Port — Non-Debug Mode . . . . .	8-33
8.3.8 Serial Data Out of Development Port — Debug Mode . . . . .	8-33
8.3.8.1 Valid Data Output . . . . .	8-34
8.3.8.2 Sequencing Error Output . . . . .	8-34
8.3.8.3 CPU Exception Output . . . . .	8-35
8.3.8.4 Null Output . . . . .	8-35
8.3.9 Use of the Ready Bit . . . . .	8-35
8.4 Debug Mode Functions . . . . .	8-36
8.4.1 Enabling Debug Mode . . . . .	8-36
8.4.2 Entering Debug Mode . . . . .	8-37
8.4.3 Debug Mode Operation . . . . .	8-38
8.4.4 Freeze Function . . . . .	8-39
8.4.5 Exiting Debug Mode . . . . .	8-39
8.4.6 Checkstop State and Debug Mode . . . . .	8-39
8.5 Development Port Transmission Sequence . . . . .	8-40

**Freescale Semiconductor, Inc.**

<b>Paragraph Number</b>	<b>Page Number</b>
8.5.1 Port Usage in Debug Mode . . . . .	8-40
8.5.2 Debug Mode Sequence Diagram . . . . .	8-42
8.5.3 Port Usage in Normal (Non-Debug) Mode . . . . .	8-43
8.6 Examples of Debug Mode Sequences . . . . .	8-44
8.6.1 Prologue Instruction Sequence . . . . .	8-44
8.6.2 Epilogue Instruction Sequence . . . . .	8-44
8.6.3 Peek Instruction Sequence . . . . .	8-45
8.6.4 Poke Instruction Sequence . . . . .	8-45
8.7 Software Monitor Support . . . . .	8-46
8.8 Development Support Registers . . . . .	8-47
8.8.1 Register Protection . . . . .	8-48
8.8.2 Comparator A–D Value Registers (CMPA–CMPD) . . . . .	8-50
8.8.3 Comparator E–F Value Registers . . . . .	8-50
8.8.4 Comparator G–H Value Registers (CMPG–CMPH) . . . . .	8-51
8.8.5 I-Bus Support Control Register . . . . .	8-51
8.8.6 L-Bus Support Control Register 1 . . . . .	8-53
8.8.7 L-Bus Support Control Register 2 . . . . .	8-55
8.8.8 Breakpoint Counter A Value and Control Register . . . . .	8-57
8.8.9 Breakpoint Counter B Value and Control Register . . . . .	8-58
8.8.10 Exception Cause Register (ECR) . . . . .	8-58
8.8.11 Debug Enable Register (DER) . . . . .	8-60

## Section 9 INSTRUCTION SET

9.1 Instruction Formats . . . . .	9-1
9.1.1 Split Field Notation . . . . .	9-1
9.1.2 Instruction Fields . . . . .	9-1
9.1.3 Notation and Conventions . . . . .	9-3
9.2 RCPU Instruction Set . . . . .	9-6

## Appendix A INSTRUCTION SET LISTINGS

## Appendix B MULTIPLE-PRECISION SHIFTS

## Appendix C FLOATING-POINT MODELS AND CONVERSIONS

C.1 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word . . . . .	C-1
C.2 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word . . . . .	C-1
C.3 Floating-Point Models . . . . .	C-1
C.3.1 Floating-Point Round to Single-Precision Model . . . . .	C-1
C.3.2 Floating-Point Convert to Integer Model . . . . .	C-5
C.4 Floating-Point Convert from Integer Model . . . . .	C-8

## Appendix D SYNCHRONIZATION PROGRAMMING EXAMPLES

D.1 General Information . . . . .	D-1
D.2 Synchronization Primitives . . . . .	D-2
D.2.1 Fetch and No-Op . . . . .	D-2
D.2.2 Fetch and Store . . . . .	D-2
D.3 Fetch and Add . . . . .	D-2
D.3.1 Fetch and AND . . . . .	D-3
D.3.2 Test and Set . . . . .	D-3
D.4 Compare and Swap . . . . .	D-3
D.5 List Insertion . . . . .	D-4

## Appendix E SIMPLIFIED MNEMONICS

E.1 Symbols . . . . .	E-1
E.2 Simplified Mnemonics for Subtract Instructions . . . . .	E-2
E.2.1 Subtract Immediate . . . . .	E-2
E.2.2 Subtract . . . . .	E-2
E.3 Simplified Mnemonics for Compare Instructions . . . . .	E-2
E.4 Simplified Mnemonics for Rotate and Shift Instructions . . . . .	E-3
E.5 Simplified Mnemonics for Branch Instructions . . . . .	E-4
E.5.1 BO and BI Fields . . . . .	E-5
E.5.2 Basic Branch Mnemonics . . . . .	E-5
E.5.3 Branch Mnemonics Incorporating Conditions . . . . .	E-9
E.5.4 Branch Prediction . . . . .	E-10
E.6 Simplified Mnemonics for Condition Register Logical Instructions . . . . .	E-11
E.7 Simplified Mnemonics for Trap Instructions . . . . .	E-12
E.8 Simplified Mnemonics for Special-Purpose Registers . . . . .	E-15
E.9 Recommended Simplified Mnemonics . . . . .	E-16
E.9.1 No-Op . . . . .	E-16
E.9.2 Load Immediate . . . . .	E-16
E.9.3 Load Address . . . . .	E-16
E.9.4 Move Register . . . . .	E-17
E.9.5 Complement Register . . . . .	E-17
E.9.6 Move to Condition Register . . . . .	E-17

## GLOSSARY OF TERMS AND ABBREVIATIONS

## SUMMARY OF CHANGES

## INDEX

Online publishing by JABIS™, <http://www.jabis.com>

**LIST OF FIGURES**

<b>Figure</b>	<b>Title</b>	<b>Page</b>
1-1	RCPU-Based Microcontroller .....	1-2
1-2	RCPU Block Diagram .....	1-4
1-3	RCPU Instruction Flow .....	1-6
1-4	Basic Instruction Pipeline .....	1-10
2-1	RCPU Programming Model .....	2-2
3-1	Big-Endian Byte Ordering .....	3-2
3-2	Big-Endian Mapping of Structure S .....	3-4
3-3	Little-Endian Mapping of Structure S .....	3-4
3-4	PowerPC Little-Endian Structure S in Memory .....	3-5
3-5	PowerPC Little-Endian Structure S as Seen by Processor .....	3-6
3-6	PowerPC Little-Endian Mode, Word Stored at Address 5 .....	3-6
3-7	Word Stored at Little-Endian Address 5 as Seen by Big-Endian Addressing .....	3-7
3-8	PowerPC Big-Endian Instruction Sequence as Seen by Processor .....	3-9
3-9	PowerPC Little-Endian Instruction Sequence as Seen by Processor .....	3-9
3-10	Floating-Point Single-Precision Format .....	3-10
3-11	Floating-Point Double-Precision Format .....	3-11
3-12	Biased Exponent Format .....	3-12
3-13	Approximation to Real Numbers .....	3-13
3-14	Format for Normalized Numbers .....	3-14
3-15	Format for Zero Numbers .....	3-14
3-16	Format for Denormalized Numbers .....	3-14
3-17	Format for Positive and Negative Infinities .....	3-15
3-18	Format for NaNs .....	3-15
3-19	Representation of QNaN .....	3-16
3-20	Single-Precision Representation in an FPR .....	3-19
3-21	Rounding Flow Diagram .....	3-20
3-22	Relation of Z1 and Z2 .....	3-21
4-1	Register Indirect with Immediate Index Addressing .....	4-31
4-2	Register Indirect with Index Addressing .....	4-32
4-3	Register Indirect Addressing .....	4-33
4-4	Register Indirect with Immediate Index Addressing .....	4-41
4-5	Register Indirect with Index Addressing .....	4-42
4-6	Branch Relative Addressing .....	4-50
4-7	Branch Conditional Relative Addressing .....	4-51
4-8	Branch to Absolute Addressing .....	4-51

Figure	Title	Page
4-9	Branch Conditional to Absolute Addressing .....	4-52
4-10	Branch Conditional to Link Register Addressing .....	4-53
4-11	Branch Conditional to Count Register Addressing .....	4-54
5-1	Instruction Cache Organization .....	5-2
5-2	Instruction Cache Data Path .....	5-3
6-1	History Buffer Queue .....	6-8
6-2	RCPU Floating-Point Architecture .....	6-32
6-3	Real Numbers Axis for Denormalized Operands .....	6-33
7-1	Instruction Flow .....	7-2
7-2	Instruction Sequencer Data Path .....	7-3
7-3	Basic Instruction Pipeline .....	7-5
7-4	Number of Bus Cycles Needed for String Instruction Execution .....	7-8
7-5	Load from Internal Memory Example .....	7-17
7-6	Write-Back Arbitration Example I .....	7-17
7-7	Write-Back Arbitration Example II .....	7-18
7-8	Load with Private Write-Back Bus Example .....	7-19
7-9	External Load Example .....	7-20
7-10	History Buffer Full Example .....	7-21
7-11	Store and Floating-Point Example .....	7-22
7-12	Branch Folding Example .....	7-23
7-13	Branch Prediction Example .....	7-24
8-1	Watchpoint and Breakpoint Support in the RCPU .....	8-12
8-2	Partially Supported Watchpoint/Breakpoint Example .....	8-15
8-3	I-Bus Support General Structure .....	8-16
8-4	L-Bus Support General Structure .....	8-18
8-5	Development Port Support Logic .....	8-23
8-6	Development Port Registers and Data Paths .....	8-25
8-7	Enabling Clock Mode Following Reset .....	8-28
8-8	Asynchronous Clocked Serial Communications .....	8-29
8-9	Synchronous Clocked Serial Communications .....	8-30
8-10	Synchronous Self-Clocked Serial Communications .....	8-31
8-11	Enabling Debug Mode at Reset .....	8-37
8-12	Entering Debug Mode Following Reset .....	8-38
8-13	General Port Usage Sequence Diagram .....	8-43
8-14	Debug Mode Logic .....	8-47
9-1	Instruction Description .....	9-6

## LIST OF TABLES

Table	Title	Page
1-1	RCPU Execution Units.....	1-7
2-1	FPSCR Bit Categories.....	2-5
2-2	FPSCR Bit Settings .....	2-5
2-3	Floating-Point Result Flags in FPSCR.....	2-8
2-4	Bit Settings for CR0 Field of CR.....	2-9
2-5	Bit Settings for CR1 Field of CR.....	2-9
2-6	CR $n$ Field Bit Settings for Compare Instructions .....	2-10
2-7	Integer Exception Register Bit Definitions .....	2-11
2-8	Time Base Field Definitions .....	2-13
2-9	Machine State Register Bit Settings .....	2-15
2-10	Floating-Point Exception Mode Bits.....	2-16
2-11	Time Base Field Definitions .....	2-17
2-12	Uses of SPRG0–SPRG3 .....	2-20
2-13	Processor Version Register Bit Settings.....	2-20
2-14	EIE, EID, AND NRI Registers.....	2-21
2-15	Instruction Cache Control Registers.....	2-21
2-16	Development Support Registers.....	2-22
3-1	Memory Operands .....	3-1
3-2	EA Modifications .....	3-5
3-3	Load/Store String Instructions .....	3-7
3-4	Load/Store Multiple Instructions .....	3-8
3-5	IEEE Floating-Point Fields.....	3-11
3-6	Recognized Floating-Point Numbers .....	3-13
3-7	FPSCR Bit Settings — RN Field.....	3-20
3-8	Interpretation of G, R, and X Bits.....	3-23
3-9	Location of the Guard, Round and Sticky Bits .....	3-23
4-1	Integer Arithmetic Instructions .....	4-6
4-2	Integer Compare Instructions .....	4-12
4-3	Integer Logical Instructions.....	4-13
4-4	Rotate and Shift Operations .....	4-15
4-5	Integer Rotate Instructions.....	4-16
4-6	Integer Shift Instructions .....	4-18
4-7	Floating-Point Arithmetic Instructions .....	4-19
4-8	Floating-Point Multiply-Add Instructions .....	4-22
4-9	Floating-Point Rounding and Conversion Instructions.....	4-26
4-10	CR Bit Settings .....	4-27
4-11	Floating-Point Compare Instructions .....	4-28
4-12	Floating-Point Status and Control Register Instructions .....	4-29

**Freescale Semiconductor, Inc.**

<b>Table</b>	<b>Title</b>	<b>Page</b>
4-13	Integer Load Instructions .....	4-34
4-14	Integer Store Instructions.....	4-37
4-15	Integer Load and Store with Byte Reversal Instructions.....	4-38
4-16	Integer Load and Store Multiple Instructions .....	4-39
4-17	Integer Move String Instructions .....	4-40
4-18	Floating-Point Load Instructions .....	4-42
4-19	Floating-Point Store Instructions.....	4-45
4-20	Floating-Point Move Instructions .....	4-48
4-21	BO Operand Encodings.....	4-55
4-22	Branch Instructions .....	4-56
4-23	Condition Register Logical Instructions .....	4-58
4-24	System Linkage Instructions.....	4-59
4-25	Trap Instructions .....	4-60
4-26	TO Operand Bit Encoding.....	4-60
4-27	Move to/from Machine State Register/Condition Register Instructions .....	4-61
4-28	Move to/from Special Purpose Register Instructions.....	4-62
4-29	User-Level SPR Encodings .....	4-62
4-30	Supervisor-Level SPR Encodings.....	4-63
4-31	Development Support SPR Encodings.....	4-64
4-32	Move from Time Base Instruction .....	4-65
4-33	User-Level TBR Encodings .....	4-65
4-34	Memory Synchronization Instructions.....	4-67
4-35	Instruction Cache Management Instruction .....	4-68
5-1	Instruction Cache Programming Model .....	5-3
5-2	ICGST Bit Settings.....	5-4
5-3	ICADR Bit Settings .....	5-5
5-4	ICDAT Bit Settings.....	5-5
5-5	ICADR Bits Function for the Cache Read Command .....	5-10
5-6	ICDAT Layout During a Tag Read.....	5-11
6-1	RCPU Exception Classes .....	6-2
6-2	Handling of Precise Exceptions .....	6-4
6-3	Exception Vectors and Conditions.....	6-6
6-4	Manipulating EE and RI Bits .....	6-10
6-5	Exception Priorities.....	6-11
6-6	Detection Order of Synchronous Exceptions.....	6-13
6-7	Machine State Register Bit Settings .....	6-15
6-8	Floating-Point Exception Mode Bits.....	6-16
6-9	MSR Setting Due to Exception .....	6-17
6-10	DAR, BAR, and DSISR Values in Exception Processing .....	6-18



Table	Title	Page
6-11	Exception Latency .....	6-19
6-12	Settings Caused by Reset .....	6-20
6-13	Machine Check Exception Processor Actions .....	6-21
6-14	Register Settings Following a Machine Check Exception.....	6-22
6-15	Register Settings Following External Interrupt.....	6-23
6-16	Register Settings for Alignment Exception .....	6-24
6-17	DSISR[15:21] Settings.....	6-25
6-18	Register Settings Following Program Exception.....	6-28
6-19	Register Settings Following a Floating-Point Unavailable Exception .....	6-28
6-20	Register Settings Following a Decrementer Exception.....	6-29
6-21	Register Settings Following a System Call Exception .....	6-30
6-22	Register Settings Following a Trace Exception .....	6-30
6-23	Register Settings Following a Floating-Point Assist Exception.....	6-31
6-24	Software/Hardware Partitioning in Operands Treatment.....	6-33
6-25	FPECR Bit Settings .....	6-35
6-26	FPSCR Bit Settings .....	6-36
6-27	Floating-Point Result Flags in FPSCR.....	6-38
6-28	Floating-Point Exception Mode Bits.....	6-41
6-29	Register Settings Following a Software Emulation Exception .....	6-47
6-30	Register Settings Following Data Breakpoint Exception.....	6-48
6-31	Register Settings Following an Instruction Breakpoint Exception.....	6-48
6-32	Register Settings Following a Maskable External Breakpoint Exception.....	6-49
6-33	Register Settings Following a Non-Maskable External Breakpoint Exception .....	6-50
7-1	Load/Store Instructions Timing .....	7-8
7-2	Encodings of External-to-the-Processor SPRs.....	7-11
7-3	Instruction Execution Timing.....	7-12
7-4	Control Registers and Serialized Access.....	7-15
8-1	Program Trace Cycle Attribute Encodings.....	8-3
8-2	Fetch Show Cycles Control .....	8-4
8-3	VF Pins Instruction Encodings.....	8-5
8-4	VF Pins Queue Flush Encodings.....	8-6
8-5	VFLS Pin Encodings.....	8-6
8-6	Cycle Type Encodings .....	8-7
8-7	Detecting the Trace Buffer Starting Point .....	8-10
8-8	I-bus Watchpoint Programming Options.....	8-17
8-9	L-Bus Data Events.....	8-19
8-10	L-Bus Watchpoints Programming Options.....	8-19

# Freescale Semiconductor, Inc.

Table	Title	Page
8-11	Trap Enable Data Shifted Into Development Port Shift Register .....	8-32
8-12	Breakpoint Data Shifted Into Development Port Shift Register .....	8-32
8-13	CPU Instructions/Data Shifted into Shift Register.....	8-32
8-14	Status Shifted Out of Shift Register — Non-Debug Mode .....	8-33
8-15	Status/Data Shifted Out of Shift Register .....	8-34
8-16	Sequencing Error Activity .....	8-35
8-17	Checkstop State and Debug Mode.....	8-40
8-18	Debug Mode Development Port Usage .....	8-41
8-19	Non-Debug Mode Development Port Usage .....	8-44
8-20	Prologue Events .....	8-44
8-21	Epilogue Events.....	8-45
8-22	Peek Instruction Sequence.....	8-45
8-23	Poke Instruction Sequence.....	8-46
8-24	Development Support Programming Model.....	8-48
8-25	Development Support Registers Read Access Protection .....	8-49
8-26	Development Support Registers Write Access Protection.....	8-49
8-27	CMPE-CMPD Bit Settings .....	8-50
8-28	CMPE-CMPF Bit Settings.....	8-50
8-29	CMPG-CMPH Bit Settings.....	8-51
8-30	ICTRL Bit Settings .....	8-52
8-31	LCTRL1 Bit Settings .....	8-54
8-32	LCTRL2 Bit Settings .....	8-55
8-33	Breakpoint Counter A Value and Control Register (COUNTA).....	8-57
8-34	Breakpoint Counter B Value and Control Register (COUNTB).....	8-58
8-35	ECR Bit Settings.....	8-59
8-36	DER Bit Settings.....	8-61
9-1	Instruction Formats .....	9-2
9-2	RTL Notation and Conventions.....	9-4
9-3	Precedence Rules .....	9-5
9-4	Simplified Mnemonics for addi Instruction .....	9-10
9-5	Simplified Mnemonics for addic Instruction .....	9-11
9-6	Simplified Mnemonics for addic. Instruction .....	9-12
9-7	Simplified Mnemonics for addis Instruction .....	9-13
9-8	Simplified Mnemonics for bc, bca, bcl, and bcla Instructions .....	9-22
9-9	Simplified Mnemonics for bcctr and bcctrl Instructions.....	9-26
9-10	Simplified Mnemonics for bclr and bclrl Instructions .....	9-28
9-11	Simplified Mnemonics for cmp Instruction .....	9-30

**Freescale Semiconductor, Inc.**

<b>Table</b>	<b>Title</b>	<b>Page</b>
9-12	Simplified Mnemonics for cmpi Instruction .....	9-31
9-13	Simplified Mnemonics for cmpl Instruction .....	9-32
9-14	Simplified Mnemonics for cmpli Instruction.....	9-33
9-15	Simplified Mnemonics for creqv Instruction .....	9-37
9-16	Simplified Mnemonics for crnr Instruction.....	9-39
9-17	Simplified Mnemonics for cror Instruction.....	9-40
9-18	Simplified Mnemonics for crxor Instruction .....	9-42
9-19	Simplified Mnemonics for mfspr Instruction .....	9-114
9-20	TBR Encodings for mftb.....	9-115
9-21	Simplified Mnemonics for mfspr Instruction .....	9-116
9-22	Simplified Mnemonics for mtrcf Instruction .....	9-117
9-23	Simplified Mnemonics for mtspr Instruction .....	9-124
9-24	Simplified Mnemonics for nor Instruction.....	9-131
9-25	Simplified Mnemonics for or Instruction.....	9-132
9-26	Simplified Mnemonics for ori Instruction .....	9-134
9-27	Simplified Mnemonics for rlwimi Instruction.....	9-137
9-28	Simplified Mnemonics for rlwinm Instruction.....	9-139
9-29	Simplified Mnemonics for rlwnm Instruction .....	9-140
9-30	Simplified Mnemonics for subf Instruction .....	9-173
9-31	Simplified Mnemonics for subfc Instruction .....	9-174
9-32	Simplified Mnemonics for tw Instruction .....	9-180
9-33	Simplified Mnemonics for twi Instruction.....	9-181
A-1	Complete Instruction List Sorted by Mnemonic.....	A-1
E-1	Condition Register CR Field Bit Symbols.....	E-1
E-2	Word Compare Simplified Mnemonics .....	E-3
E-3	Word Rotate and Shift Instructions.....	E-4
E-4	BO Operand Encodings .....	E-5
E-5	Simplified Branch Mnemonics .....	E-6
E-6	Operands for Simplified Branch Mnemonics .....	E-7
E-7	Simplified Branch Mnemonics with Comparison Conditions .....	E-9
E-8	Operands for Simplified Branch Mnemonics with Comparison Conditions .....	E-9
E-9	Condition Register Logical Mnemonics .....	E-12
E-10	Trap Mnemonics Encoding.....	E-13
E-11	Trap Mnemonics.....	E-14
E-12	TO Operand Bit Encoding .....	E-15
E-13	SPR Simplified Mnemonics .....	E-15

Table	Title	Page
-------	-------	------

## PREFACE

This manual defines the functionality of the RCPU for use by software and hardware developers. The RCPU is a PowerPC-based processor used in the Motorola MPC500 family of microcontrollers.

### Audience

This manual is intended for system software and hardware developers and applications programmers who want to develop products for RCPU-based microcontroller systems. It is assumed that the reader understands operating systems, microprocessor and microcontroller system design, and the basic principles of RISC processing.

### Additional Reading

This section lists additional reading that provides background to or supplements the information in this manual.

- John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA
- *PowerPC Microprocessor Family: the Programming Environments*, MPCFPE/AD (Motorola order number)
- *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Standard 754-1985), published by the Institute of Electrical and Electronics Engineers, Inc., New York, NY
- Motorola technical summaries and device manuals for individual RCPU-based microcontrollers; and module reference manuals (such as this manual) that describe the operation of the individual modules in RCPU-based MCUs in detail. Refer to <http://www.mcu.motsp.com> for a comprehensive listing of available documentation.

### Conventions

This document uses the following notational conventions:

ACTIVE_HIGH	Names for signals that are active high are shown in uppercase text without an overbar. Signals that are active high are referred to as asserted when they are high and negated when they are low.
<u>ACTIVE_LOW</u>	A bar over a signal name indicates that the signal is active low. Active-low signals are referred to as asserted (active) when they are low and negated when they are high.
<b>mnemonics</b>	Instruction mnemonics are shown in lowercase bold.
<i>italics</i>	Italics indicate variable command parameters, for example,

# Freescale Semiconductor, Inc.

	<b>bcctrx</b>
0x0F	Hexadecimal numbers
0b0011	Binary numbers
rA 0	The contents of a specified GPR or the value 0.
REG[FIELD]	Abbreviations or acronyms for registers are shown in uppercase text. Specific bit fields or ranges are shown in brackets.
x	In certain contexts, such as a signal encoding, this indicates a don't care. For example, if a field is binary encoded 0bx001, the state of the first bit is a don't care.

## Nomenclature

**Logic level one** is the voltage that corresponds to Boolean true (1) state.

**Logic level zero** is the voltage that corresponds to Boolean false (0) state.

To **set** a bit or bits means to establish logic level one on the bit or bits.

To **clear** a bit or bits means to establish logic level zero on the bit or bits.

A signal that is **asserted** is in its active logic state. An active low signal changes from logic level one to logic level zero when asserted, and an active high signal changes from logic level zero to logic level one.

A signal that is **negated** is in its inactive logic state. An active low signal changes from logic level zero to logic level one when negated, and an active high signal changes from logic level one to logic level zero.

**LSB** means least significant bit or bits. **MSB** means most significant bit or bits. References to low and high bytes are spelled out.

## SECTION 1 OVERVIEW

This section provides an overview of RCPU features and the PowerPC Architecture™ and summarizes the operation of the RCPU as a PowerPC™ implementation.

### 1.1 RCPU Overview

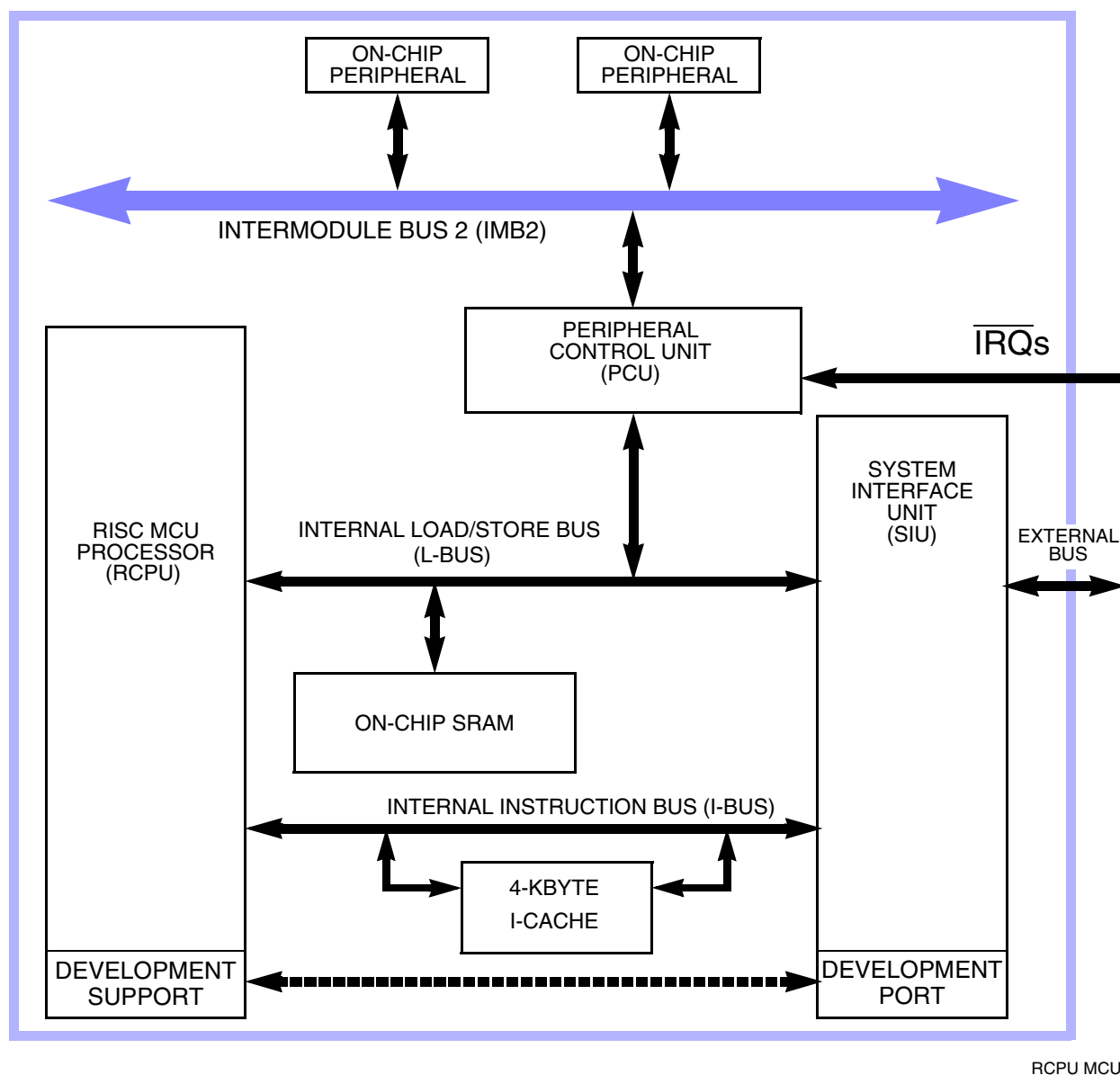
The RCPU is a single-issue, 32-bit implementation of the PowerPC architecture. The processor provides 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of 32 and 64 bits.

The RCPU integrates four execution units: an integer unit (IU), a load/store unit (LSU), a branch processing unit (BPU), and a floating-point unit (FPU). The RCPU can issue one sequential (non-branch) instruction per clock cycle. In addition, the processor attempts to evaluate branch conditions ahead of time and execute branch instructions simultaneously with sequential instructions, often resulting in zero-cycle execution time for branch instructions. Instructions can complete out of order for increased performance; however, the processor makes execution appear sequential.

RCPU-based microcontrollers (MCUs) include an on-chip, four-Kbyte, two-way set-associative instruction cache (I-cache). The I-cache uses a least recently used (LRU) replacement algorithm.

RCPU-based MCUs include a number of features to aid in system debugging. Features implemented in the silicon include internal breakpoint comparators, internal bus visibility, program flow tracking, and a development port.

**Figure 1-1** is a simplified block diagram of an RCPU-based MCU.



**Figure 1-1 RCPU-Based Microcontroller**

The following subsections describe the features of the RCPU, provide a block diagram showing the major functional units, and give an overview of RCPU operation.

### 1.1.1 RCPU Features

Major features of the RCPU are as follows:

- High-performance microprocessor
  - Single clock-cycle execution for most instructions
- Four independent execution units
  - Independent load/store unit (LSU) for load and store operations
  - Branch processor unit (BPU) featuring static branch prediction



## Freescale Semiconductor, Inc.

- A 32-bit integer unit (IU)
- Floating-point unit (FPU) for both single- and double-precision operations (fully IEEE 754-compliant when used with software envelope)
- 32 general-purpose registers (GPRs) for integer operands
- 32 floating-point registers (FPRs) for single- or double-precision operands
- Facilities for enhanced system performance
  - Programmable big- and little-endian byte ordering
  - Atomic memory references
- In-system testability and debugging features through boundary-scan capability
- High instruction and data throughput
  - Condition register (CR) look-ahead operations performed by BPU
  - Branch-folding capability during execution (zero-cycle branch execution time)
  - Programmable static branch prediction on unresolved conditional branches
  - A pre-fetch queue that can hold up to four instructions, providing look-ahead capability
  - Interlocked pipelines with feed-forwarding that control data dependencies in hardware
  - Four-Kbyte instruction cache: two-way set-associative, LRU replacement algorithm
  - Programmable static branch prediction on conditional branches

### 1.1.2 RCPU Block Diagram

**Figure 1-2** provides a block diagram of the RCPU.

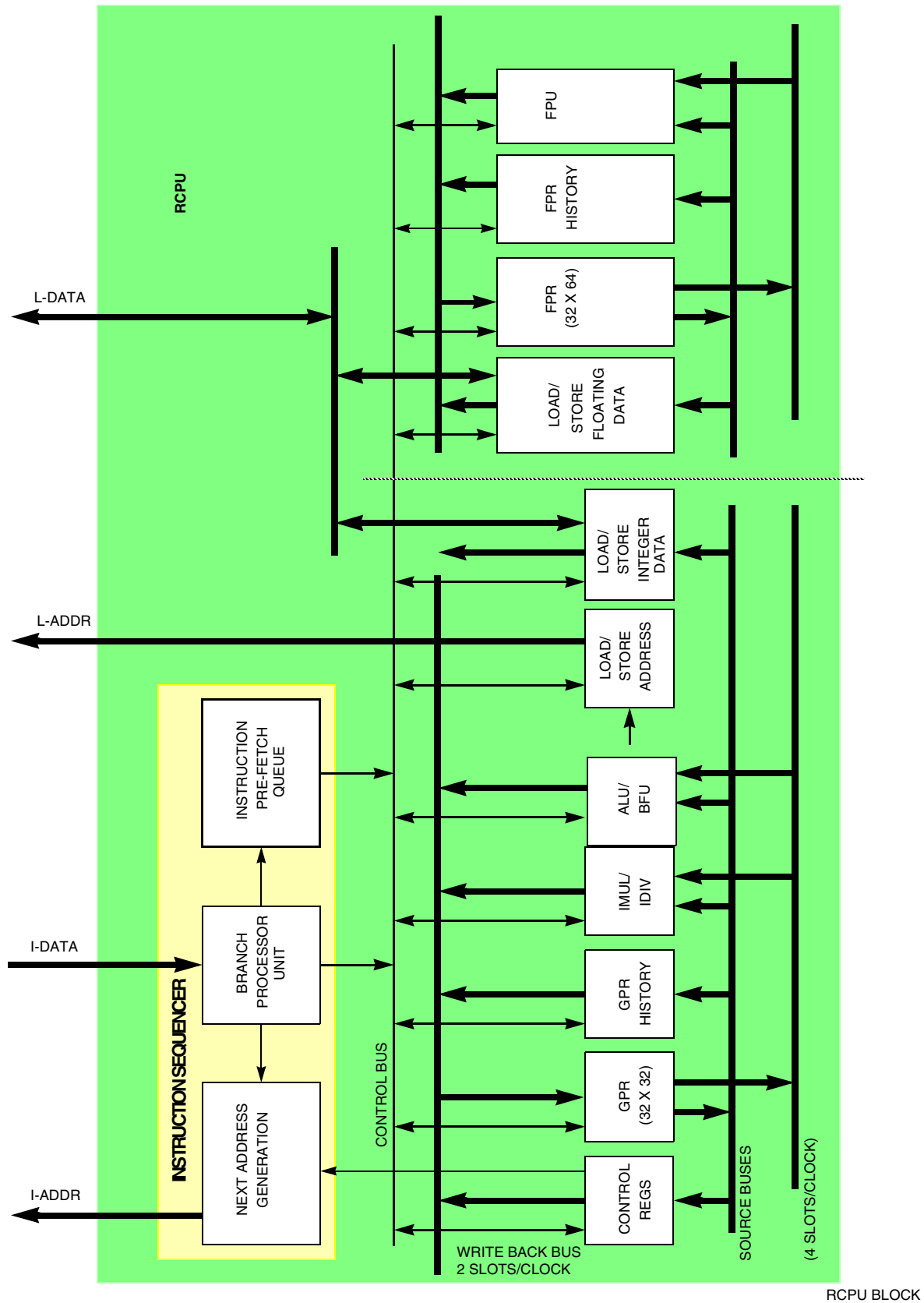


Figure 1-2 RCPU Block Diagram

## 1.1.3 Instruction Sequencer

The instruction sequencer provides centralized control over data flow between execution units and register files. The sequencer implements the basic instruction pipeline, fetches instructions from the memory system, issues them to available execution units, and maintains a state history so it can back the machine up in the event of an exception.

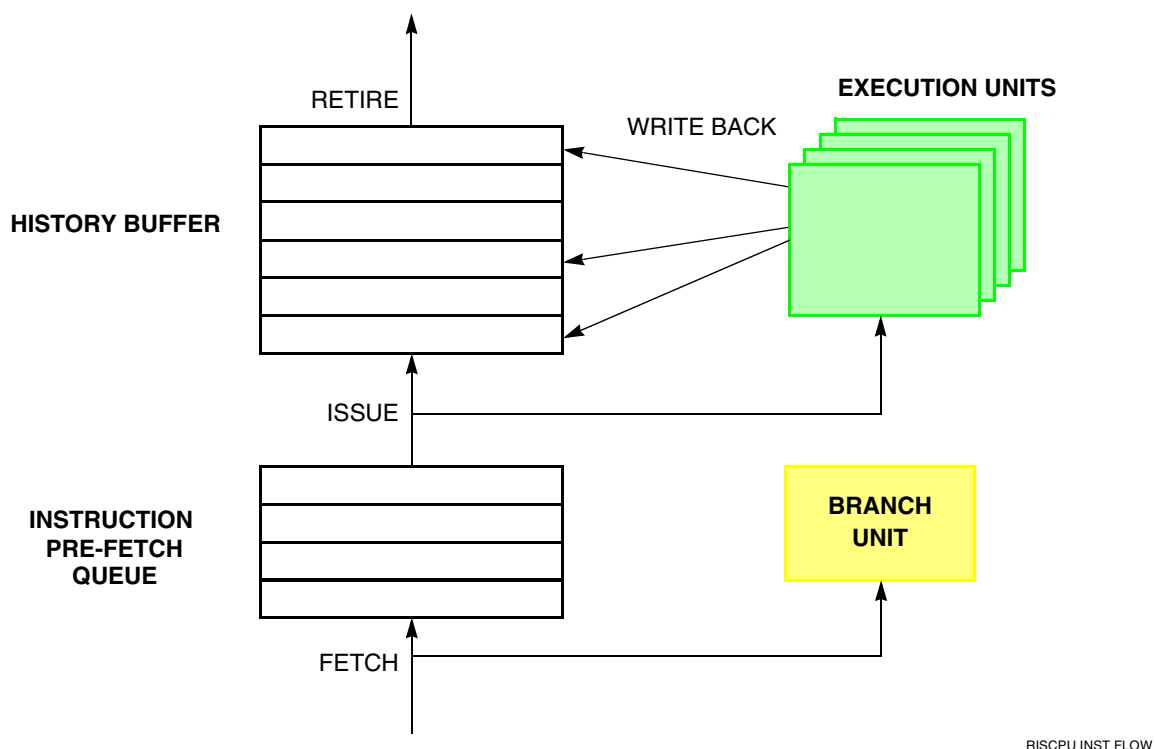
In addition, the sequencer implements all branch processor instructions, which include flow control and condition register instructions. Refer to [1.1.4.1 Branch Processing Unit \(BPU\)](#) for more details on the branch processing unit within the instruction sequencer.

The instruction sequencer fetches the instructions from the instruction cache into the instruction pre-fetch queue, which can hold up to four instructions. The processor uses branch folding (a technique of removing the branch instructions from the pre-fetch queue) in order to execute branches in parallel with execution of sequential instructions. Sequential (non-branch) instructions reaching the top of the instruction pre-fetch queue are issued to the execution units. Instructions may be flushed from the queue when an external interrupt is detected, a previous instruction causes an exception, or a branch prediction turns out to be incorrect.

All instructions, including branches, enter the history buffer along with processor state information that may be affected by the instruction's execution. This information is used to enable out-of-order completion of instructions together with the handling of precise exceptions. Instructions may be flushed from the machine when an exception is taken. Refer to [6.3 Precise Exception Model Implementation](#) and to [7.1 Instruction Flow](#) for additional information.

An instruction retires from the machine after it finishes execution without exception and all preceding instructions have already retired from the machine.

**Figure 1-3** illustrates the instruction flow in the RCPU.



**Figure 1-3 RCPU Instruction Flow**

#### 1.1.4 Independent Execution Units

The RCPU supports independent floating-point, integer, load/store, and branch processing execution units, making it possible to implement advanced features such as look-ahead operations. For example, since branch instructions do not depend on GPRs or FPRs, branches can often be resolved early, eliminating stalls caused by taken branches.

**Table 1-1** summarizes the RCPU execution units.

Table 1-1 RCPU Execution Units

Unit	Description
Branch processor unit (BPU)	Includes the implementation of all branch instructions.
Load/store unit (LSU)	Includes implementation of all load and store instructions, whether defined as part of the integer processor or the floating-point processor.
Integer unit (IU)	Includes implementation of all integer instructions except load/store instructions. This module includes the GPRs (including GPR history and scoreboard) and the following subunits: <ul style="list-style-type: none"> <li>The IMUL-IDIV unit includes the implementation of the integer multiply and divide instructions.</li> <li>The ALU-BFU unit includes implementation of all integer logic, add and subtract instructions, and bit field instructions.</li> </ul>
Floating-point unit (FPU)	Includes the FPRs (including FPR history and scoreboard) and the implementation of all floating-point instructions except load and store floating-point instructions.

#### 1.1.4.1 Branch Processing Unit (BPU)

The branch processor unit executes all branch instructions defined in the PowerPC architecture, including flow control and condition register instructions.

The BPU is implemented as part of the instruction sequencer. The BPU performs condition register look-ahead operations on conditional branches. The BPU looks through the instruction queue for a conditional branch instruction and attempts to resolve it early, achieving the effect of a zero-cycle branch in many cases.

The BPU uses a bit in the instruction encoding to predict the direction of the conditional branch. (Refer to the discussion of the BO field in [4.6 Flow Control Instructions](#).) Therefore, when an unresolved conditional branch instruction is encountered, the processor pre-fetches instructions from the predicted target stream until the conditional branch is resolved.

The BPU contains an adder to compute branch target addresses and three special-purpose, user-accessible registers: the link register (LR), the count register (CTR), and the condition register (CR). The BPU calculates the return pointer for subroutine calls and saves it into the LR. The LR also contains the branch target address for the branch conditional to link register (**bclrx**) instruction. The CTR contains the branch target address for the branch conditional to count register (**bcctrx**) instruction. The contents of the LR and CTR can be copied to or from any GPR. Because the BPU uses dedicated registers rather than general-purpose or floating-point registers, execution of branch instructions is independent from execution of integer and floating-point instructions.

#### 1.1.4.2 Integer Unit (IU)

The integer unit executes all fixed-point (integer) processor instructions defined by

## Freescal Semiconductor, Inc.

the PowerPC architecture, except for fixed-point load and store instructions, which are implemented by the load/store unit. The IU consists of two execution units:

- The IMUL-IDIV executes the integer multiply and divide instructions.
- The ALU-BFU unit executes all integer logic, add, and subtract instructions, and bit-field instructions.

The IU includes the integer exception register (XER) and the general-purpose register file. These registers are described in [SECTION 2 REGISTERS](#).

### 1.1.4.3 Floating-Point Unit (FPU)

The floating-point unit executes all the floating-point processor instructions defined by the PowerPC architecture, except for floating-point load and store instructions, which are executed by the load/store unit.

The floating-point unit contains a double-precision multiply array, the floating-point status and control register (FPSCR), and the FPRs. The multiply-add array allows the processor to efficiently implement floating-point operations such as multiply, multiply-add, and divide.

The RCPU depends on a software envelope to fully implement the IEEE floating-point specification. Overflows, underflows, NaNs, and denormalized numbers cause floating-point assist exceptions that invoke a software routine to deliver (with hardware assistance) the correct IEEE result. Refer to [6.11.10 Floating-Point Assist Exception \(0x00E00\)](#) for additional information.

To accelerate time-critical operations and make them more deterministic, the RCPU provides a mode of operation that avoids invoking the software envelope and attempts to deliver results in hardware that are adequate for most applications, if not in strict conformance with IEEE standards. In this mode, denormalized numbers, NaNs, and IEEE-invalid operations are treated as legitimate, returning default results rather than causing floating-point assist exceptions.

### 1.1.4.4 Load/Store Unit (LSU)

The load/store unit handles all integer and floating-point load and store instructions, including unaligned and string accesses. LSU instructions transfer data between the integer and floating-point register files and the chip-internal load/store bus (L-bus). The load/store unit is implemented as an independent execution unit so that stalls in the memory pipeline do not cause the master instruction pipeline to stall (unless there is a data dependency). The unit is fully pipelined so that memory instructions of any size may be issued on back-to-back cycles.

There is a 32-bit wide data path between the load/store unit and the integer register file and a 64-bit wide data path between the load/store unit and the floating-point register file.

The LSU interfaces with the external bus interface for all instructions that access memory. Addresses are formed by adding the source one register operand speci-

fied by the instruction (or zero) to either a source two register operand or to a 16-bit, immediate value embedded in the instruction.

### 1.1.5 Instruction Cache

RCPU-based MCUs contain a 4-Kbyte, two-way set associative instruction cache. The cache is organized into 128 sets, two lines per set, and four words per line. Cache lines are aligned on four-word boundaries in memory.

A cache access cycle begins with an instruction request from the instruction unit in the processor. In case of a cache hit, the instruction is delivered to the instruction unit. In the case of a cache miss, the cache initiates a burst read cycle on the I-bus with the address of the requested instruction. The first word received from the bus is the requested instruction. The cache forwards this instruction to the instruction unit of the CPU as soon as it is received from the I-bus. A cache line is then selected to receive the data which will be coming from the bus. An LRU replacement algorithm is used to select a line when no empty lines are available.

Each cache line can be used as an SRAM, thus allowing the application to lock critical code segments that need fast and deterministic execution time.

The instruction cache is described in [SECTION 5 INSTRUCTION CACHE](#).

### 1.1.6 Instruction Pipeline

The RCPU is a pipelined processor. A pipelined processor is one in which the processing of an instruction is broken down into discrete stages; in the RCPU, these stages are dispatch, execute, writeback, and retirement (described below). Because instruction processing is broken into a series of steps, an instruction does not require the entire resources of the processor. For example, after an instruction completes the decode stage, it can pass on to the next stage, while the subsequent instruction can advance into the decode stage.

The RCPU instruction pipeline has four stages:

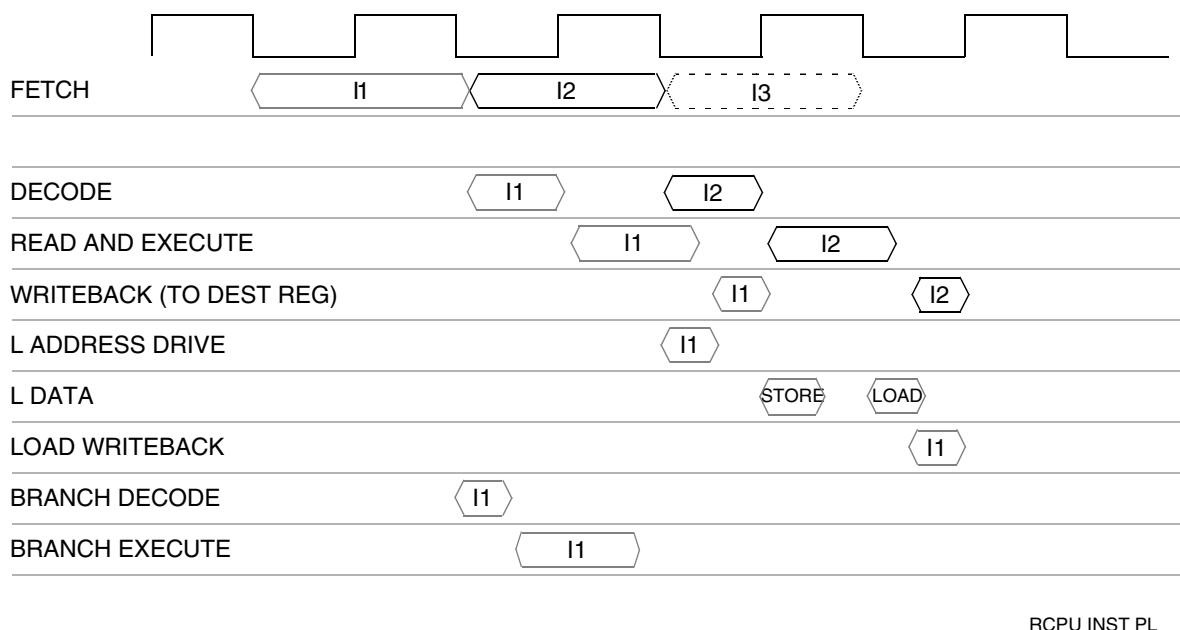
1. The dispatch stage is implemented using a distributed mechanism. The central dispatch unit broadcasts the instruction to all units. In addition, scoreboard information (regarding data dependencies) is broadcast to each execution unit. Each execution unit decodes the instruction. If the instruction is not implemented, a program exception is taken. If the instruction is legal and no data dependency is found, the instruction is accepted by the appropriate execution unit, and the data found in the destination register is copied to the history buffer. If a data dependency exists, the machine is stalled until the dependency is resolved.
2. In the execute stage, each execution unit that has an executable instruction executes the instruction (perhaps over multiple cycles).
3. In the writeback stage, the execution unit writes the result to the destination register and reports to the history buffer that the instruction is completed.
4. In the retirement stage, the history buffer retires instructions in architectural order. An instruction retires from the machine if it completes execution with

# Freescale Semiconductor, Inc.

no exceptions and if all instructions preceding it in the instruction stream have finished execution with no exceptions. As many as six instructions can be retired in one clock.

The history buffer maintains the correct architectural machine state. An exception is taken only when the instruction is ready to be retired from the machine. When an exception is taken, all instructions following the excepting instruction are canceled, (i.e., the values of the affected destination registers are restored using the values saved in the history buffer during the dispatch stage).

**Figure 1-4** illustrates the basic instruction pipeline timing. Refer to **SECTION 7 INSTRUCTION TIMING** for more detailed timing illustrations.



**Figure 1-4 Basic Instruction Pipeline**

## 1.1.7 Development Support

Development tools are used by a microcomputer system developer to debug the hardware and software of a target system. These tools are used to give the developer some control over the execution of the target program and to allow the user to debug the program by observing its execution. In-circuit emulators, bus state analyzers, and software monitors are the most frequently used debugging tools. The RCPu supports the use of development tools by providing internal breakpoint comparators, internal bus visibility, program flow tracking, and a development port.

For details on development support, refer to **SECTION 8 DEVELOPMENT SUPPORT**.



## 1.2 Levels of the PowerPC Architecture

The PowerPC architecture consists of three layers. Adherence to the PowerPC architecture can be measured in terms of which of the following levels of the architecture are implemented:

- PowerPC user instruction set architecture (UISA) — Defines the base user-level instruction set, user-level registers, data types, floating-point exception model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment.
- PowerPC virtual environment architecture (VEA) — Describes the memory model for a multiprocessor environment, defines cache control instructions, and describes other aspects of virtual environments. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.
- PowerPC operating environment architecture (OEA) — Defines the memory management model, supervisor-level registers, synchronization requirements, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA.

## 1.3 The RCPU as a PowerPC Implementation

This subsection describes the RCPU as a member of the PowerPC processor family.

### 1.3.1 PowerPC Registers and Programming Model

The PowerPC architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction opcode. The three-register instruction format allows specification of a target register distinct from the two source operands. Load and store instructions transfer data between memory and on-chip registers.

PowerPC processors have two levels of privilege: supervisor mode of operation (typically used by the operating system) and user mode of operation (used by the application software). The programming models incorporate 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each PowerPC processor also has its own unique set of implementation-specific registers.

The RCPU is a 32-bit implementation of the PowerPC architecture. In the RCPU, the time base and FPRs are 64 bits; all other registers are 32 bits.

The following sections summarize the PowerPC registers that are implemented in the RCPU. Refer to [SECTION 2 REGISTERS](#) for detailed descriptions of PowerPC registers. In addition, for descriptions of the I-cache control registers, refer to [SECTION 5 INSTRUCTION CACHE](#). For details on development-support registers, refer to [SECTION 8 DEVELOPMENT SUPPORT](#).

## 1.3.1.1 General-Purpose Registers (GPRs)

The processor provides 32 user-level, general-purpose registers (GPRs). The GPRs serve as the data source or destination for all integer instructions and provide addresses for all memory-access instructions.

## 1.3.1.2 Floating-Point Registers (FPRs)

The processor also provides 32 user-level 64-bit floating-point registers. The FPRs serve as the data source or destination for floating-point instructions. These registers can contain data objects of either single- or double-precision floating-point formats. The floating-point register file can only be accessed by the FPU.

## 1.3.1.3 Condition Register (CR)

The CR is a 32-bit user-level register that consists of eight four-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, and provide a mechanism for testing and branching.

## 1.3.1.4 Floating-Point Status and Control Register (FPSCR)

The floating-point status and control register (FPSCR) is a user-level register that contains all exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard.

## 1.3.1.5 Machine State Register (MSR)

The machine state register (MSR) is a supervisor-level register that defines the state of the processor. The contents of this register are saved when an exception is taken and restored when the exception handling completes.

## 1.3.1.6 Special-Purpose Registers (SPRs)

The processor provides several special-purpose registers that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. Some SPRs are accessed implicitly as part of executing certain instructions. All SPRs can be accessed by using the move to/from special-purpose register instructions, **mtspr** and **mfspir**.

## 1.3.1.7 User-Level SPRs

The following SPRs are accessible by user-level software:

- The link register (LR) can be used to provide the branch target address and to hold the return address after branch and link instructions.
- The count register (CTR) is decremented and tested automatically as a result of branch-and-count instructions.
- The integer exception register (XER) contains the integer carry and overflow bits and two fields for the load string and compare byte indexed (**lscbx**) instruction. The XER is 32 bits wide in all implementations.

- The time base (TB) can be read at the user privilege level. A separate SPR number is provided for writing to the time base. Writes to the time base can occur only at the supervisor privilege level.

## NOTE

While these registers are defined as SPRs and can be accessed by using the **mtspr** and **mfspr** instructions, they (except for the time base) are typically accessed implicitly.

### 1.3.1.8 Supervisor-Level SPRs

The processor also contains SPRs that can be accessed only by supervisor-level software. These registers consist of the following:

- The data access exception (DAE)/source instruction service register (DSISR) defines the cause of data access and alignment exceptions.
- The data address register (DAR) holds the address of an access after an alignment or data access exception.
- Decrementer register (DEC) is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. The DEC frequency is provided as a subdivision of the processor clock frequency.
- The machine status save/restore register 0 (SRR0) is used by the processor to save the address of the instruction that caused the exception, and the address to return to when a return from interrupt (**rfi**) instruction is executed.
- The machine status save/restore register 1 (SRR1) is used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed.
- General SPRs, SPRG[0:3], are provided for operating system use.
- The processor version register (PVR) is a read-only register that identifies the version (model) and revision level of the PowerPC processor.
- The time base (TB) can be written to only at the supervisor privilege level. Separate SPR numbers are provided for reading and writing to the time base.

The following supervisor-level SPRs are implementation-specific to the RCPU:

- The EIE, EID, and NRI are provided to facilitate exception processing.
- Cache control SPRs allow system software to control the operation of the instruction cache.
- Development support SPRs allow development-system software control over the on-chip development support.
- The floating-point exception cause register (FPECR) is a 32-bit internal status and control register used to assist the software emulation of floating-point operations.

### 1.3.2 Instruction Set and Addressing Modes

The following subsections describe the PowerPC instruction set and addressing modes and summarize the instructions implemented in the RCPU.

### 1.3.2.1 PowerPC Instruction Set

All PowerPC instructions are encoded as single-word (32-bit) opcodes. Instruction formats are consistent among all instruction types, permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplify instruction pipelining. In addition, each instruction is defined in a way that simplifies pipelined implementations and allows maximum realization of instruction-level parallelism.

The PowerPC instructions are divided into the following categories:

- Integer instructions. These include computational and logical instructions.
  - Integer arithmetic instructions
  - Integer compare instructions
  - Integer logical instructions
  - Integer rotate and shift instructions
- Floating-point instructions. These include floating-point computational instructions, as well as instructions that affect the floating-point status and control register (FPSCR).
  - Floating-point arithmetic instructions
  - Floating-point multiply-add instructions
  - Floating-point rounding and conversion instructions
  - Floating-point compare instructions
  - Floating-point status and control instructions
- Load/store instructions. These include integer and floating-point load and store instructions.
  - Integer load and store instructions
  - Integer load and store multiple instructions
  - Floating-point load and store
  - Floating-point move instructions
- Flow control instructions. These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
  - Branch and trap instructions
  - Condition register logical instructions
- Processor control instructions. These instructions are used for synchronizing memory accesses and cache management.
  - Move to/from special-purpose register instructions
  - Synchronize
  - Instruction synchronize
- Memory control instructions. These instructions provide control of the instruction cache.
  - Instruction cache block invalidate

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). It also pro-

vides for word and double-word operand loads and stores between memory and a set of 32 floating-point registers (FPRs).

Computational instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location with distinct instructions.

## 1.3.2.2 PowerPC Addressing Modes

The effective address (EA) is the 32-bit address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction.

The PowerPC architecture supports two simple memory addressing modes:

- $EA = (rA|0) + \text{offset}$  (register indirect with immediate index)
- $EA = (rA|0) + rB$  (register indirect with index)

Note that with register indirect with immediate index addressing, the offset can be equal to zero.

Refer to [4.1.2 Addressing Modes and Effective Address Calculation](#) for additional information.

## 1.3.2.3 RCPU Instruction Set

The RCPU instruction set is defined as follows:

- The RCPU supports all 32-bit PowerPC UISA required instructions.
- The RCPU supports the following PowerPC VEA instructions: **eieio**, **icbi**, **isync**, and **mftb**.
- The RCPU supports the following PowerPC OEA instructions: **mfmsr**, **mf spr**, **mtmsr**, **mts pr**, **r fi**, and **sc**. (Note that **mts pr**, **mf spr**, and **sc** are also defined in the UISA architecture.)
- The RCPU does not provide any implementation-specific instructions not defined in the PowerPC architecture.
- The RCPU implements the following instruction which is defined as optional in the PowerPC architecture: **stfiwx**.

An attempt to execute a PowerPC optional instruction that is not implemented in hardware causes the RCPU to take the implementation dependent software emulation exception.

For additional information on the PowerPC architecture, refer to *PowerPC Microprocessor Family: the Programming Environments*, MPCFPE/AD (Motorola order number).

## 1.3.3 PowerPC Exception Model

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, the address of the instruction to be executed after control is returned to the original program and the contents of the machine state register are saved to the save/restore registers (SRR0 and SRR1). Program control then passes from user to supervisor level, and software continues execution at an address (exception vector) predetermined for each exception.

Although multiple exception conditions can map to a single exception vector, the specific condition can be determined by examining a register associated with the exception — for example, the DAE/DSISR and the FPSCR. Specific exception conditions can be explicitly enabled or disabled by software.

While exception conditions may be recognized out of order, they are handled strictly in order. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream are allowed to complete. Any exceptions caused by those instructions are handled in order.

Unless a catastrophic condition causes a non-maskable exception, only one exception is handled at a time. If, for example, a single instruction encounters multiple exception conditions, those conditions are encountered sequentially. After the exception handler handles an exception, the instruction execution continues until the next exception condition is encountered. This method of recognizing and handling exception conditions sequentially guarantees that the processor can recover the machine state following an exception.

For additional information on exception handling, refer to **SECTION 6 EXCEPTIONS**.

## SECTION 2 REGISTERS

This section describes the RCPU register organization as defined by the three levels of the PowerPC architecture: the user instruction set architecture (UISA), the virtual environment architecture (VEA), and the operating environment architecture (OEA), as well as the RCPU's implementation-specific registers.

### 2.1 Programming Models

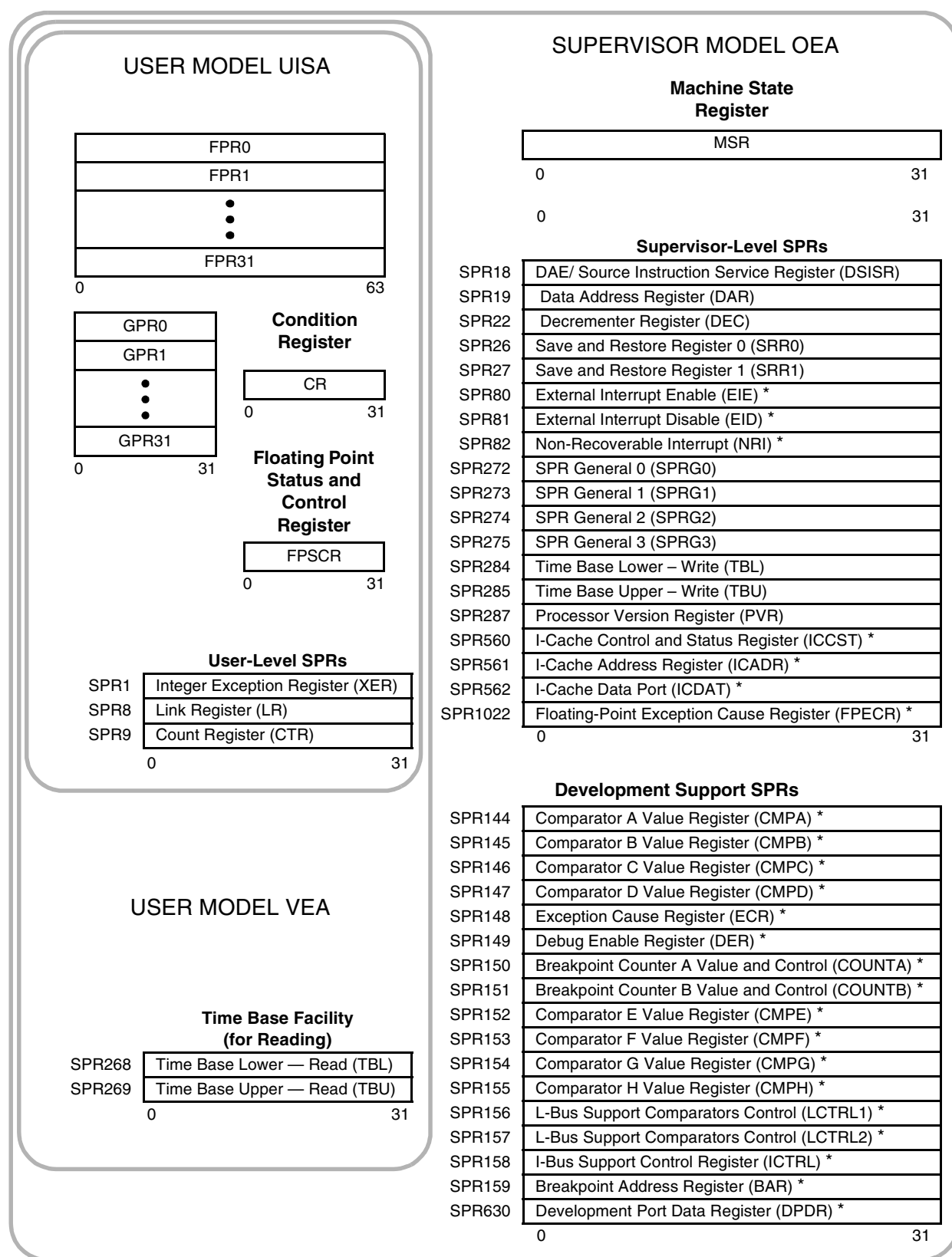
The processor operates at one of two privilege levels: supervisor level (typically used by the operating environment) or user level (used by the application software). This division allows the operating system to control the application environment, protecting operating-system and critical machine resources. Instructions that control the state of the processor and supervisor registers can be executed only when the processor is operating at the supervisor level.

Supervisor-level access is provided through the processor's exception mechanism. That is, when an exception is taken, either due to an error or problem that needs to be serviced or deliberately through the use of a trap instruction, the processor begins operating in supervisor mode. The level of access is indicated by the privilege-level (PR) bit in the machine state register (MSR).

**Figure 2-1** shows the user-level and supervisor-level RCPU programming models and also illustrates the three levels of the PowerPC architecture. The numbers to the left of the SPRs indicate the decimal number that is used in the syntax of the instruction operands to access the register.

#### NOTE

Registers such as the general-purpose registers (GPRs) and floating-point registers (FPRs) are accessed through operands that are part of the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as move to special-purpose register (**mtspr**) and move from special-purpose register (**mfspr**) instructions) or implicitly as the part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.



\* Implementation-specific to the RCPU

RMCU CPU REG MA

Figure 2-1 RCPU Programming Model



Where not otherwise noted, reserved fields in registers are ignored when written and return zero when read. An exception to this rule is XER[16:23]. These bits are set to the value written to them and return that value when read.

## 2.2 PowerPC UISA Register Set

The PowerPC UISA registers can be accessed by either user- or supervisor-level instructions. The general-purpose registers and floating-point registers are accessed through instruction operands. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as the **mtspr** and **mf-spr** instructions) or implicit as part of the execution (or side effect) of an instruction. Some registers are accessed both explicitly and implicitly.

### 2.2.1 General Purpose Registers (GPRs)

Integer data is manipulated in the integer unit's thirty-two 32-bit GPRs, shown below. These registers are accessed as source and destination registers through operands in the instruction syntax.

#### GPRs — General Purpose Registers

0		31
	GPR0	
	GPR1	
	...	
	...	
	GPR31	

RESET: UNCHANGED

### 2.2.2 Floating-Point Registers (FPRs)

The PowerPC architecture provides thirty-two 64-bit FPRs. These registers are accessed as source and destination registers through operands in floating-point instructions. Each FPR supports the double-precision, floating-point format. Every instruction that interprets the contents of an FPR as a floating-point value uses the double-precision floating-point format for this interpretation.

All floating-point arithmetic instructions operate on data located in FPRs and, with the exception of the compare instructions (which update the CR), place the result into an FPR. Information about the status of floating-point operations is placed into the floating-point status and control register (FPSCR) and in some cases, into the CR, after the completion of the operation's writeback stage. For information on how the CR is affected by floating-point operations, see [2.2.4 Condition Register \(CR\)](#).

Load and store double instructions transfer 64 bits of data between memory and the FPRs in the floating-point processor with no conversion. Load single instruc-

tions transfer and convert floating-point values in single-precision floating-point format from memory to the same value in double-precision floating-point format in the FPRs. Store single instructions are provided to read a double-precision floating-point value from a floating-point register, convert it to single-precision floating-point format, and place it in the target memory location.

Single- and double-precision arithmetic instructions accept values from the FPRs in double-precision format. For single-precision arithmetic instructions, all input values must be representable in single-precision format; otherwise, the result placed into the target FPR and the setting of status bits in the FPSCR and in the condition register are undefined.

The floating-point arithmetic instructions produce intermediate results that may be regarded as infinitely precise. After normalization or denormalization, if the precision of the intermediate result cannot be represented in the destination format (either 32-bit or 64-bit) then it must be rounded. The final result is then placed into the FPR in the double-precision format.

## FPRs — Floating-Point Registers

0	63
FPR0	
FPR1	
...	
...	
FPR31	

RESET: UNCHANGED

### 2.2.3 Floating-Point Status and Control Register (FPSCR)

The FPSCR controls the handling of floating-point exceptions and records status resulting from the floating-point operations. FPSCR[0:23] are status bits. FPSCR[24:31] are control bits.

FPSCR[0:12] and FPSCR[21:23] are floating-point exception condition bits. These bits are sticky, except for the floating-point enabled exception summary (FEX) and floating-point invalid operation exception summary (VX). Once set, sticky bits remain set until they are cleared by an **mcrfs**, **mtfsfi**, **mtfsf**, or **mtfsb0** instruction.

**Table 2-1** summarizes which bits in the FPSCR are sticky bits, which are normal status bits, and which are control bits.

Table 2-1 FPSCR Bit Categories

Bits	Type
[0], [3:12], [21:23]	Status, sticky
[1:2], [13:20]	Status, not sticky
[24:31]	Control

FEX and VX are the logical ORs of other FPSCR bits. Therefore these two bits are not listed among the FPSCR bits directly affected by the various instructions.

### FPSCR — Floating-Point Status and Control Register

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FX	FEX	VX	OX	UX	ZX	XX	VXS- NAN	VXSI	VXDI	VXZDZ	VXIMZ	VXVC	FR	FI	FPRF0

RESET: UNCHANGED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
FPRF[16:19]				0	VX- SOFT	VX- SQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN	

RESET: UNCHANGED

A listing of FPSCR bit settings is shown in [Table 2-2](#).

Table 2-2 FPSCR Bit Settings

Bit(s)	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to change from zero to one. The <b>mcrfs</b> instruction implicitly clears FPSCR[FX] if the FPSCR field containing FPSCR[FX] is copied. The <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions can set or clear FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked with their respective enable bits. The <b>mcrfs</b> instruction implicitly clears FPSCR[FEX] if the result of the logical OR described above becomes zero. The <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions cannot set or clear FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exceptions. The <b>mcrfs</b> instruction implicitly clears FPSCR[VX] if the result of the logical OR described above becomes zero. The <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions cannot set or clear FPSCR[VX] explicitly. This is not a sticky bit.

## Table 2-2 FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
3	OX	Floating-point overflow exception. This is a sticky bit. See <a href="#">6.11.10.8 Overflow Exception Condition</a> .
4	UX	Floating-point underflow exception. This is a sticky bit. See <a href="#">6.11.10.9 Underflow Exception Condition</a> .
5	ZX	Floating-point zero divide exception. This is a sticky bit. See <a href="#">6.11.10.7 Zero Divide Exception Condition</a> .
6	XX	Floating-point inexact exception. This is a sticky bit. See <a href="#">6.11.10.10 Inexact Exception Condition</a> .
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
8	VXISI	Floating-point invalid operation exception for $\infty-\infty$ . This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
9	VXIDI	Floating-point invalid operation exception for $\infty/\infty$ . This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
10	VXZDZ	Floating-point invalid operation exception for 0/0. This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
11	VXIMZ	Floating-point invalid operation exception for $x*0$ . This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
13	FR	Floating-point fraction rounded. The last floating-point instruction that potentially rounded the intermediate result incremented the fraction. (See <a href="#">3.3.11 Rounding</a> .) This bit is not sticky.
14	FI	Floating-point fraction inexact. The last floating-point instruction that potentially rounded the intermediate result produced an inexact fraction or a disabled exponent overflow. (See <a href="#">3.3.11 Rounding</a> .) This bit is not sticky.
[15:19]	FPRF	<p>Floating-point result flags. This field is based on the value placed into the target register even if that value is undefined. Refer to <a href="#">Table 2-3</a> for specific bit settings.</p> <p>15 Floating-point result class descriptor (C). Floating-point instructions other than the compare instructions may set this bit with the FPCC bits, to indicate the class of the result.</p> <p>[16:19] Floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Other floating-point instructions may set the FPCC bits with the C bit, to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.</p> <p>16 Floating-point less than or negative (FL or &lt;)</p> <p>17 Floating-point greater than or positive (FG or &gt;)</p> <p>18 Floating-point equal or zero (FE or =)</p> <p>19 Floating-point unordered or NaN (FU or ?)</p>
20	—	Reserved

## Freescale Semiconductor, Inc.

Table 2-2 FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
21	VXSOF	Floating-point invalid operation exception for software request. This bit can be altered only by the <b>mcrfs</b> , <b>mtfsfi</b> , <b>mtfsf</b> , <b>mtfsb0</b> , or <b>mtfsb1</b> instructions. The purpose of VXSOF is to allow software to cause an invalid operation condition for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root if the source operand is negative. This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit. This guarantees that software can simulate <b>fsqrt</b> and <b>frsqrite</b> , and to provide a consistent interface to handle exceptions caused by square-root operations. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
24	VE	Floating-point invalid operation exception enable. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
25	OE	Floating-point overflow exception enable. See <a href="#">6.11.10.8 Overflow Exception Condition</a> .
26	UE	Floating-point underflow exception enable. This bit should not be used to determine whether denormalization should be performed on floating-point stores. See <a href="#">6.11.10.9 Underflow Exception Condition</a> .
27	ZE	Floating-point zero divide exception enable. See <a href="#">6.11.10.7 Zero Divide Exception Condition</a> .
28	XE	Floating-point inexact exception enable. See <a href="#">6.11.10.10 Inexact Exception Condition</a> .
29	NI	Non-IEEE mode bit. See <a href="#">3.4.3 Non-IEEE Operation</a> .
[30:31]	RN	Floating-point rounding control. See <a href="#">3.3.11 Rounding</a> . 00 = Round to nearest 01 = Round toward zero 10 = Round toward +infinity 11 = Round toward -infinity

**Table 2-3** illustrates the floating-point result flags that correspond to FPSCR bits [15:19].

Table 2-3 Floating-Point Result Flags in FPSCR

Result Flags (Bits [15:19]) C<=>=?	Result value class
10001	Quiet NaN
01001	– Infinity
01000	– Normalized number
11000	– Denormalized number
10010	– Zero
00010	+ Zero
10100	+ Denormalized number
00100	+Normalized number
00101	+Infinity

### 2.2.4 Condition Register (CR)

The condition register (CR) is a 32-bit register that reflects the result of certain operations and provides a mechanism for testing and branching. The bits in the CR are grouped into eight 4-bit fields, CR0 to CR7.

#### CR — Condition Register

0	3	4	7	8	11	12	15	16	19	20	23	24	27	28	31
CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7								

RESET: UNCHANGED

The CR fields can be set in the following ways:

- Specified fields of the CR can be set by a move instruction (**mtrcf**) to the CR from a GPR.
- Specified fields of the CR can be moved from one CRx field to another with the **mcrf** instruction.
- A specified field of the CR can be set by a move instruction (**mcrfs**) to the CR from the FPSCR.
- A specified field of the CR can be set by a move instruction (**mcrxr**) to the CR from the XER.
- Condition register logical instructions can be used to perform logical operations on specified bits in the condition register.
- CR0 can be the implicit result of an integer operation.
- CR1 can be the implicit result of a floating-point operation.
- A specified CR field can be the explicit result of either an integer or floating-point compare instruction.

Instructions are provided to test individual CR bits.

### 2.2.4.1 Condition Register CR0 Field Definition

In most integer instructions, when the CR is set to reflect the result of the operation (that is, when  $R_c = 1$ ), and for **addic.**, **andi.**, and **andis.**, the first three bits of CR0 are set by an algebraic comparison of the result to zero; the fourth bit of CR0 is copied from XER[SO]. For integer instructions, CR[0:3] are set to reflect the result as a signed quantity. The result as an unsigned quantity or a bit string can be deduced from the EQ bit.

The CR0 bits are interpreted as shown in [Table 2-4](#). If any portion of the result (the 32-bit value placed into the destination register) is undefined, the value placed in the first three bits of CR0 is undefined.

**Table 2-4 Bit Settings for CR0 Field of CR**

CR0 Bit	Description
0	Negative (LT) — This bit is set when the result is negative.
1	Positive (GT) — This bit is set when the result is positive (and not zero).
2	Zero (EQ) — This bit is set when the result is zero.
3	Summary overflow (SO) — This is a copy of the final state of XER[SO] at the completion of the instruction.

### 2.2.4.2 Condition Register CR1 Field Definition

In all floating-point instructions when the CR is set to reflect the result of the operation (that is, when  $R_c = 1$ ), the CR1 field (bits 4 to 7 of the CR) is copied from FPSCR[0:3] to indicate the floating-point exception status. For more information about the FPSCR, see [2.2.3 Floating-Point Status and Control Register \(FPSCR\)](#). The bit settings for the CR1 field are shown in [Table 2-5](#).

**Table 2-5 Bit Settings for CR1 Field of CR**

CR1 Bit	Description
0	Floating-point exception (FX) — This is a copy of the final state of FPSCR[FX] at the completion of the instruction.
1	Floating-point enabled exception (FEX) — This is a copy of the final state of FPSCR[FEX] at the completion of the instruction.
2	Floating-point invalid exception (VX) — This is a copy of the final state of FPSCR[VX] at the completion of the instruction.
3	Floating-point overflow exception (OX) — This is a copy of the final state of FPSCR[OX] at the completion of the instruction.

### 2.2.4.3 Condition Register CR<sub>n</sub> Field — Compare Instruction

When a specified CR field is set by a compare instruction, the bits of the specified field are interpreted as shown in [Table 2-6](#). A condition register field can also be accessed by the **mfcrr**, **mcrf**, and **mtcrf** instructions.

### Table 2-6 CR<sub>n</sub> Field Bit Settings for Compare Instructions

CRn Bit <sup>1</sup>	Description
0	Less than, floating-point less than (LT, FL). For integer compare instructions, ( <b>rA</b> ) < SIMM, UIMM, or ( <b>rB</b> ) (algebraic comparison) or ( <b>rA</b> ) SIMM, UIMM, or ( <b>rB</b> ) (logical comparison). For floating-point compare instructions, ( <b>frA</b> ) < ( <b>frB</b> ).
1	Greater than, floating-point greater than (GT, FG). For integer compare instructions, ( <b>rA</b> ) > SIMM, UIMM, or ( <b>rB</b> ) (algebraic comparison) or ( <b>rA</b> ) SIMM, UIMM, or ( <b>rB</b> ) (logical comparison). For floating-point compare instructions, ( <b>frA</b> ) > ( <b>frB</b> ).
2	Equal, floating-point equal (EQ, FE). For integer compare instructions, ( <b>rA</b> ) = SIMM, UIMM, or ( <b>rB</b> ). For floating-point compare instructions, ( <b>frA</b> ) = ( <b>frB</b> ).
3	Summary overflow, floating-point unordered (SO, FU). For integer compare instructions, this is a copy of the final state of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of ( <b>frA</b> ) and ( <b>frB</b> ) is not a number (NaN).

NOTES:

1. Here, the bit indicates the bit number in any one of the four-bit subfields, CR[0:7]

### 2.2.5 Integer Exception Register (XER)

The integer exception register (XER) is a user-level, 32-bit register.

## XER — Integer Exception Register

## SPR 1

0	1	2	3																			24	25												31	
SO	OV	CA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	BYTES														

RESET:UNCHANGED

The SPR number for the XER is one. The bit definitions for XER, shown in [Table 2-7](#), are based on the operation of an instruction considered as a whole, not on intermediate results. For example, the result of the subtract from carrying (**subfcx**) instruction is specified as the sum of three values. This instruction sets bits in the XER based on the entire operation, not on an intermediate sum.

In most cases, reserved fields in registers are ignored when written and return zero when read. However, XER[16:23] are set to the value written to them and return that value when read.



Table 2-7 Integer Exception Register Bit Definitions

Bit(s)	Name	Description
0	SO	Summary Overflow (SO) — The summary overflow bit is set whenever an instruction sets the overflow bit (OV) to indicate overflow and remains set until software clears it. It is not altered by compare instructions or other instructions that cannot overflow.
1	OV	Overflow (OV) — The overflow bit is set to indicate that an overflow has occurred during execution of an instruction. Integer and subtract instructions having OE = 1 set OV if the carry out of bit 0 is not equal to the carry out of bit 1, and clear it otherwise. The OV bit is not altered by compare instructions or other instructions that cannot overflow.
2	CA	Carry (CA) — In general, the carry bit is set to indicate that a carry out of bit 0 occurred during execution of an instruction. Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA to one if there is a carry out of bit 0, and clear it otherwise. The CA bit is not altered by compare instructions or other instructions that cannot carry, except that shift right algebraic instructions set the CA bit to indicate whether any '1' bits have been shifted out of a negative quantity.
[3:24]	—	Reserved
[25:31]	BYTES	This field specifies the number of bytes to be transferred by a load string word indexed ( <b>lswx</b> ) or store string word indexed ( <b>stswx</b> ) instruction.

### 2.2.6 Link Register (LR)

The 32-bit link register supplies the branch target address for the branch conditional to link register (**bclrx**) instruction, and can be used to hold the logical address of the instruction that follows a branch and link instruction.

#### LR — Link Register

**SPR 8**

0

31

Branch Address
----------------

RESET:UNCHANGED

#### NOTE

Although the two least-significant bits can accept any values written to them, they are ignored when the LR is used as an address. The link register can be accessed by the **mtspr** and **mfspir** instructions using the SPR number eight. Prefetching instructions along the target path (loaded by an **mtspr** instruction) is possible provided the link register is loaded sufficiently ahead of the branch instruction. It is usually possible to prefetch along a target path loaded by a branch and link instruction.

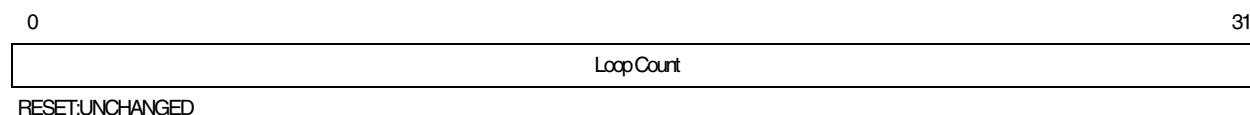
Both conditional and unconditional branch instructions include the option of placing the effective address of the instruction following the branch instruction in the LR. This is done regardless of whether the branch is taken.

## 2.2.7 Count Register (CTR)

The count register (CTR) is a 32-bit register for holding a loop count that can be decremented during execution of branch instructions that contain an appropriately coded BO field. If the value in CTR is zero before being decremented, it is negative one afterward. The count register provides the branch target address for the branch conditional to count register (**bcctrx**) instruction.

### CTR — Count Register

SPR 9



Prefetching instructions along the target path is also possible provided the count register is loaded sufficiently ahead of the branch instruction.

The count register can be accessed by the **mtspr** and **mfscr** instructions by specifying SPR 9. In branch conditional instructions, the BO field specifies the conditions under which the branch is taken. The first four bits of the BO field specify how the branch is affected by or affects the condition register and the count register. The encoding for the BO field is shown in [Table 4-21](#) in [SECTION 4 ADDRESSING MODES AND INSTRUCTION SET SUMMARY](#).

## 2.3 PowerPC VEA Register Set — Time Base

The PowerPC virtual environment architecture (VEA) defines registers in addition to those in the UISA register set. The PowerPC VEA register set can be accessed by all software with either user- or supervisor-level privileges.

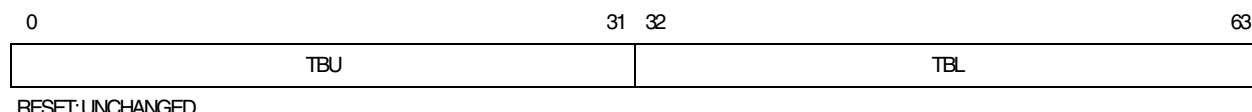
The PowerPC VEA includes the time base facility (TB), a 64-bit structure that contains a 64-bit unsigned integer that is incremented periodically. The frequency at which the counter is updated is implementation-dependent and need not be constant over long periods of time.

The TB consists of two 32-bit registers: time base upper (TBU) and time base lower (TBL). In the context of the VEA, user-level applications are permitted read-only access to the TB. The OEA defines supervisor-level access to the TB for writing values to the TB. Different SPR encodings are provided for reading and writing the time base.

Refer to [2.4 PowerPC OEA Register Set](#) for more information on writing to the TB. Refer to [4.7.2 Move to/from Special Purpose Register Instructions](#) for simplified mnemonics for reading and writing to the time base. For information on the time base clock source, refer to the [System Interface Unit Reference Manual \(SI-URM/AD\)](#).

**TB — Time Base (Reading)**

**SPR 268, 269**



**Table 2-8 Time Base Field Definitions**

Bits	Name	Description
[0:31]	TBU	Time Base (Upper) — The high-order 32 bits of the time base
[32:63]	TBL	Time Base (Lower) — The low-order 32 bits of the time base

In 32-bit PowerPC implementations such as the RCPU, it is not possible to read the entire 64-bit time base in a single instruction. The **mftb** simplified mnemonic copies the lower half of the time base register (TBL) to a GPR, and the **mftbu** simplified mnemonic copies the upper half of the time base (TBU) to a GPR.

Because of the possibility of a carry from TBL to TBU occurring between reads of the TBL and TBU, a sequence such as the following example is necessary to read the time base on RCPU-based systems.

```

loop:
    mftbu    rx      #load from TBU
    mftb     ry      #load from TBL
    mftbu    rz      #load from TBU
    cmpw     rz,rx    #see if 'old'='new'
    bne      loop    #loop if carry occurred
    
```

The comparison and loop are necessary to ensure that a consistent pair of values has been obtained.

## 2.4 PowerPC OEA Register Set

The PowerPC operating environment architecture (OEA) includes a number of SPRs and other registers that are accessible only by supervisor-level instructions. Some SPRs are RCPU-specific; some RCPU SPRs may not be implemented in other PowerPC processors, or may not be implemented in the same way.

### 2.4.1 Machine State Register (MSR)

The machine state register is a 32-bit register that defines the state of the processor. When an exception occurs, the current contents of the MSR are loaded into SRR1, and the MSR is updated to reflect the new machine state. The MSR can also be modified by the **mtmsr**, **sc**, and **rfi** instructions. It can be read by the **mfmshr** instruction.

MSR — Machine State Register

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

RESET:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
EE	PR	FP	ME	FE0	SE	BE	FE1	0	IP	RESERVED				RI	LE

RESET:

0	0	0	U	0	0	0	0	0	*	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

\* Reset value of this bit depends on the value of the internal reset configuration word. Refer to the *System Interface Unit Reference Manual* (SIURM/AD) for more information.

Table 2-13 shows the bit definitions for the MSR.

## Table 2-9 Machine State Register Bit Settings

Bit(s)	Name	Description
[0:14]	—	Reserved
15	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. 0 = Processor runs in big-endian mode during exception processing. 1 = Processor runs in little-endian mode during exception processing.
16	EE	External interrupt enable 0 = The processor delays recognition of external interrupts and decrementer exception conditions. 1 = The processor is enabled to take an external interrupt or the decrementer exception.
17	PR	Privilege level 0 = The processor can execute both user- and supervisor-level instructions. 1 = The processor can only execute user-level instructions.
18	FP	Floating-point available 0 = The processor prevents dispatch of floating-point instructions, including floating-point loads, stores and moves. Floating-point enabled program exceptions can still occur and the FPRs can still be accessed. 1 = The processor can execute floating-point instructions, and can take floating-point enabled exception type program exceptions.
19	ME	Machine check enable 0 = Machine check exceptions are disabled. 1 = Machine check exceptions are enabled.
20	FE0	Floating-point exception mode zero (See <a href="#">Table 2-10</a> .)
21	SE	Single-step trace enable 0 = The processor executes instructions normally. 1 = The processor generates a single-step trace exception upon the successful execution of the next instruction. When this bit is set, the processor dispatches instructions in strict program order. Successful execution means the instruction caused no other exception. Single-step tracing may not be present on all implementations.
22	BE	Branch trace enable 0 = No trace exception occurs when a branch instruction is completed 1 = Trace exception occurs when a branch instruction is completed
23	FE1	Floating-point exception mode 1 (See <a href="#">Table 2-10</a> .)
24	—	Reserved.
25	IP	Exception prefix. The setting of this bit specifies the location of the exception vector table. 0 = Exception vector table starts at the physical address 0x0000 0000. 1 = Exception vector table starts at the physical address 0xFFFF 0000.
[26:29]	—	Reserved
30	RI	Recoverable exception (for machine check and non-maskable breakpoint exceptions) 0 = Machine state is not recoverable. 1 = Machine state is recoverable. Refer to <a href="#">SECTION 6 EXCEPTIONS</a> for more information.
31	LE	Little-endian mode 0 = Processor operates in big-endian mode during normal processing. 1 = Processor operates in little-endian mode during normal processing.

# Freescale Semiconductor, Inc.

The floating-point exception mode bits are interpreted as shown in [Table 2-10](#). For further details see [6.11.10.5 Floating-Point Enabled Exceptions](#).

**Table 2-10 Floating-Point Exception Mode Bits**

FE[0:1]	Mode
00	Ignore exceptions mode — Floating-point exceptions do not cause the floating-point assist error handler to be invoked.
01, 10, 11	Floating-point precise mode — The system floating-point assist error handler is invoked precisely at the instruction that caused the enabled exception.

## 2.4.2 DAE/Source Instruction Service Register (DSISR)

The 32-bit DSISR identifies the cause of data access and alignment exceptions.

**DSISR** — DAE/Source Instruction Service Register

**SPR 18**

0	31
DSISR	

RESET: UNCHANGED

For information about bit settings, see [6.11.4 Alignment Exception \(0x00600\)](#).

## 2.4.3 Data Address Register (DAR)

After an alignment exception, the DAR is set to the effective address of a load or store element. For information, see [6.11.4 Alignment Exception \(0x00600\)](#).

**DAR** — Data Address Register

**SPR 19**

0	31
Data Address	

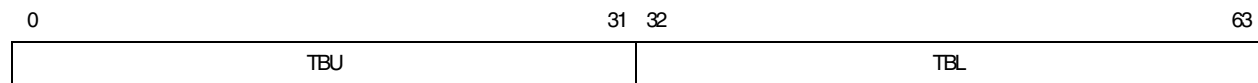
RESET: UNCHANGED

## 2.4.4 Time Base Facility (TB) — OEA

As described in [2.3 PowerPC VEA Register Set — Time Base](#), the time base (TB) provides a 64-bit incrementing counter. The VEA defines user-level, read-only access to the TB. Writing to the TB is reserved for supervisor-level applications such as operating systems and bootstrap routines. The OEA defines supervisor-level write access to the TB.

**TB — Time Base (Writing)**

**SPR 284, 285**



RESET: UNCHANGED

**Table 2-11 Time Base Field Definitions**

Bits	Name	Description
[0:31]	TBU	Time Base (Upper) — The high-order 32 bits of the time base
[32:63]	TBL	Time Base (Lower) — The low-order 32 bits of the time base

The TB can be written at the supervisor privilege level only. The **mttbl** and **mttbu** simplified mnemonics write the lower and upper halves of the TB, respectively. The **mtspr**, **mttbl**, and **mttbu** instructions treat TBL and TBU as separate 32-bit registers; setting one leaves the other unchanged. It is not possible to write the entire 64-bit time base in a single instruction.

The TB can be written by a sequence such as the following:

```

lwz      rx,upper      # load 64-bit value for
lwz      ry,lower      # TB into rx and ry
li       rz,0
mttbl    rz             # force TBL to 0
mttbu    rx             # set TBU
mttbl    ry             # set TBL
    
```

Loading zero into TBL prevents the possibility of a carry from TBL to TBU while the time base is being initialized.

For information about reading the time base, refer to [2.3 PowerPC VEA Register Set — Time Base](#).

## 2.4.5 Decrementer Register (DEC)

The DEC is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay. The DEC frequency is based on a subdivision of the processor clock. Refer to the [System Interface Unit Reference Manual \(SIURM/AD\)](#) for information on the clock source for the decrementer.

## DEC — Decrementer Register

SPR 22



The DEC counts down, causing an exception (unless masked) when it passes through zero. The DEC satisfies the following requirements:

- Loading a GPR from the DEC has no effect on the DEC.
- Storing a GPR to the DEC replaces the value in the DEC with the value in the GPR.
- Whenever bit 0 of the DEC changes from zero to one, a decrementer exception request is signaled. Multiple DEC exception requests may be received before the first exception occurs; however, any additional requests are canceled when the exception occurs for the first request. Refer to [6.11.7 Decrementer Exception \(0x00900\)](#) for additional information.
- If the DEC is altered by software and the content of bit 0 is changed from zero to one, an exception request is signaled.

The content of the DEC can be read or written using the **mf spr** and **mt spr** instructions. Using a simplified mnemonic for the **mt spr** instruction, the DEC can be written from GPR **rA** with the following:

**mtdec rA**

If the execution of this instruction causes bit 0 of the DEC to change from zero to one, an exception request is signaled. The DEC can be read into GPR **rA** with the following instruction:

**mfdec rA**

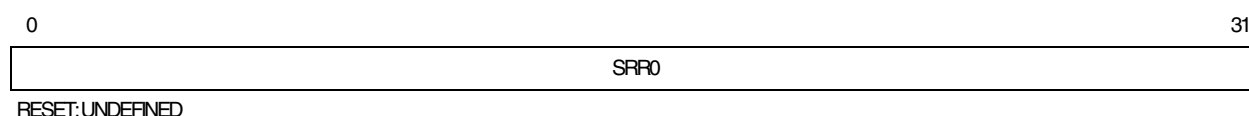
Copying the DEC to a GPR does not affect the DEC content or the exception mechanism.

### 2.4.6 Machine Status Save/Restore Register 0 (SRR0)

The machine status save/restore register 0 (SRR0) is a 32-bit register that identifies where instruction execution should resume when an **rfi** instruction is executed following an exception. It also holds the effective address of the instruction that follows the system call (**sc**) instruction.

## SRR0 — Machine Status Save/Restore Register 0

SPR 26





When an exception occurs, SRR0 is set to point to an instruction such that all prior instructions have completed execution and no subsequent instruction has begun execution. The instruction addressed by SRR0 may not have completed execution, depending on the exception type. SRR0 addresses either the instruction causing the exception or the immediately following instruction. The instruction addressed can be determined from the exception type and status bits.

For information on how specific exceptions affect SRR0, refer to the descriptions of individual exceptions in [SECTION 6 EXCEPTIONS](#).

## 2.4.7 Machine Status Save/Restore Register 1 (SRR1)

SRR1 is a 32-bit register used to save machine status on exceptions and to restore machine status when an **rfi** instruction is executed.

### SRR1 — Machine Status Save/Restore Register 1

**SPR 27**

0	31
SRR1	

RESET: UNDEFINED

In general, when an exception occurs, SRR1[0:15] are loaded with exception-specific information and of MSR[16:31] are placed into SRR1[16:31].

For information on how specific exceptions affect SRR1, refer to the individual exceptions in [SECTION 6 EXCEPTIONS](#).

## 2.4.8 General SPRs (SPRG0–SPRG3)

SPRG0 through SPRG3 are 32-bit registers provided for general operating system use, such as performing a fast state save and for supporting multiprocessor implementations. SPRG0–SPRG3 are shown below.

### SPRG0–SPRG3 — General Special-Purpose Registers 0–3

**SPR 272 – SPR 275**

0	31
SPRG0	
SPRG1	
SPRG2	
SPRG3	

RESET: UNCHANGED

Uses for SPRG0–SPRG3 are shown in [Table 2-12](#).

Table 2-12 Uses of SPRG0–SPRG3

Register	Description
SPRG0	Software may load a unique physical address in this register to identify an area of memory reserved for use by the exception handler. This area must be unique for each processor in the system.
SPRG1	This register may be used as a scratch register by the exception handler to save the content of a GPR. That GPR then can be loaded from SPRG0 and used as a base register to save other GPRs to memory.
SPRG2	This register may be used by the operating system as needed.
SPRG3	This register may be used by the operating system as needed.

### 2.4.9 Processor Version Register (PVR)

The PVR is a 32-bit, read-only register that identifies the version and revision level of the PowerPC processor. The contents of the PVR can be copied to a GPR by the **mfspvr** instruction. Read access to the PVR is available in supervisor mode only; write access is not provided.

**PVR** — Processor Version Register

**SPR 287**

0	15	16	31
VERSION		REVISION	

RESET: UNCHANGED

Table 2-13 Processor Version Register Bit Settings

Bit(s)	Name	Description
[0:15]	VERSION	A 16-bit number that identifies the version of the processor and of the PowerPC architecture
[16:31]	REVISION	A 16-bit number that distinguishes between various releases of a particular version

### 2.4.10 Implementation-Specific SPRs

The RCPU includes several implementation-specific SPRs that are not defined by the PowerPC architecture. These registers can be accessed by supervisor-level instructions only.

#### 2.4.10.1 EIE, EID, and NRI Special-Purpose Registers

The RCPU includes three implementation-specific SPRs to facilitate the software manipulation of the MSR[RI] and MSR[EE] bits. Issuing the **mtspr** instruction with one of these registers as an operand causes the RI and EE bits to be set or cleared as shown in [Table 2-14](#).

A read (**mf spr**) of any of these locations is treated as an unimplemented instruction, resulting in a software emulation exception.

**Table 2-14 EIE, EID, AND NRI Registers**

SPR Number (Decimal)	Mnemonic	MSR[EE]	MSR[RI]
80	EIE	1	1
81	EID	0	1
82	NRI	0	0

Refer to **SECTION 6 EXCEPTIONS** for more information on these registers.

## 2.4.10.2 Instruction-Cache Control Registers

The implementation-specific supervisor-level SPRs shown in **Table 2-15** control the operation of the instruction cache.

**Table 2-15 Instruction Cache Control Registers**

SPR Number (Decimal)	Name	Description
560	ICCST	I-cache control and status register
561	ICADR	I-cache address register
562	ICDAT	I-cache data port (read only)

Refer to **SECTION 5 INSTRUCTION CACHE** for details on these registers.

## 2.4.10.3 Development Support Registers

**Table 2-16** lists the implementation-specific RCPU registers provided for development support.

**Table 2-16 Development Support Registers**

SPR Number (Decimal)	Mnemonic	Name
144	CMPA	Comparator A Value Register
145	CMPB	Comparator B Value Register
146	CMPC	Comparator C Value Register
147	CMPD	Comparator D Value Register
148	ECR	Exception Cause Register
149	DER	Debug Enable Register
150	COUNTA	Breakpoint Counter A Value and Control Register
151	COUNTB	Breakpoint Counter B Value and Control Register
152	CMPE	Comparator E Value Register
153	CMPF	Comparator F Value Register
154	CMPG	Comparator G Value Register
155	CMPH	Comparator H Value Register
156	LCTRL1	L-Bus Support Control Register 1
157	LCTRL2	L-Bus Support Control Register 2
158	ICTRL	I-Bus Support Control Register
159	BAR	Breakpoint Address Register
630	DPDR	Development Port Data Register

Refer to [SECTION 8 DEVELOPMENT SUPPORT](#) for details about these registers.

#### **2.4.10.4 Floating-Point Exception Cause Register (FPECR)**

The FPECR is a 32-bit supervisor-level internal status and control register used by the floating-point assist software envelope. Refer to [6.11.10 Floating-Point Assist Exception \(0x00E00\)](#) for more information on this register.

## SECTION 3

### OPERAND CONVENTIONS

This section describes the conventions used for storing values in registers and memory, accessing PowerPC registers, and representing data in these registers.

#### 3.1 Data Alignment and Memory Organization

Bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte.

Memory operands can be bytes, half words, words, or double words, or, for the load/store multiple and move assist instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned.

Operands for single-register memory access instructions have the characteristics shown in [Table 3-1](#). (Although not permitted as memory operands, quad words are shown because quad-word alignment is desirable for certain memory operands.)

**Table 3-1 Memory Operands**

Operand	Length	ADDR[28:31] if aligned
Byte	8 bits	xxxx <sup>1</sup>
Half word	2 bytes	xxx0 <sup>1</sup>
Word	4 bytes	xx00 <sup>1</sup>
Double word	8 bytes	x000 <sup>1</sup>
Quad word	16 bytes	0000

**NOTES:**

1. An “x” in an address bit position indicates that the bit can be zero or one independent of the state of other bits in the address.

The concept of alignment is also applied more generally to data in memory. For example, 12 bytes of data are said to be word-aligned if the address of the lowest-numbered byte is a multiple of four.

Some instructions require their memory operands to have certain alignments. In addition, alignment may affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned. Additional effects of data placement on performance are described in [SECTION 7 INSTRUCTION TIMING](#).

Instructions are four bytes long and word-aligned.

### 3.2 Byte Ordering

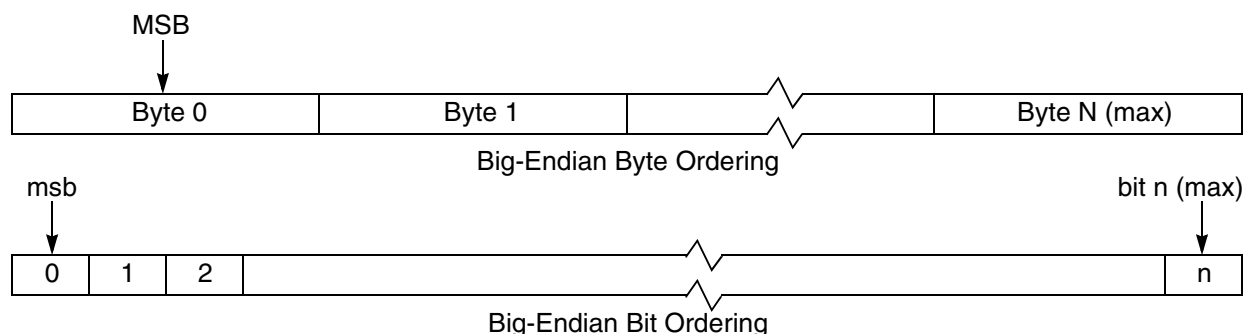
There are two practical ways to order the four bytes in a word: big-endian and little-endian. The PowerPC architecture supports both these formats.

Big-endian ordering assigns the lowest address to the highest-order eight bits of the scalar. This is called big-endian because the big end of the scalar, considered as a binary number, comes first in memory.

Little-endian byte ordering assigns the lowest address to the lowest-order (right-most) eight bits of the scalar. The little end of the scalar, considered as a binary number, comes first in memory.

Two bits in the MSR specify byte ordering: LE (little-endian mode) and ILE (exception little-endian mode). The LE bit specifies the endian mode in which the processor is currently operating, and ILE specifies the mode to be used when the system error handler is invoked. That is, when an exception occurs, the ILE bit (as set for the interrupted process) is copied into MSR[LE] to select the endian mode for the context established by the exception. For both bits, a value of zero specifies big-endian mode and a value of one specifies little-endian mode.

The default byte and bit ordering is big-endian, as shown in [Figure 3-1](#). After a hard reset, the hard reset handler (using the **mtspr** instruction) can select little-endian mode for normal operation and exception processing by setting the LE and ILE bits, respectively, in the MSR.



**Figure 3-1 Big-Endian Byte Ordering**

If individual data items were indivisible, the concept of byte ordering would be unnecessary. The order of bits or groups of bits within the smallest addressable unit of memory is irrelevant, because nothing can be observed about such order. Order matters only when scalars, which the processor and programmer regard as indivisible quantities, can be made up of more than one addressable units of memory.

For a device in which the smallest addressable unit is the 64-bit double word, there is no question of the order of bytes within double words. All transfers of individual scalars between registers and system memory are of double words. A subset of the 64-bit scalar (for example, a byte) is not addressable in memory. As a result, to access any subset of the bits of a scalar, the entire 64-bit scalar must be accessed, and when a memory location is read, the 64-bit value returned is the 64-bit value last written to that location.

For PowerPC processors, the smallest addressable memory unit is the byte (8 bits), and scalars are composed of one or more sequential bytes. When a 32-bit scalar is moved from a register to memory, it occupies four consecutive byte addresses, and a decision must be made regarding the order of these bytes in these four addresses.

### 3.2.1 Structure Mapping Examples

The following C programming example contains an assortment of scalars and one character string. The value presumed to be in each structure element is shown in hexadecimal in the comments and are used below to show how the bytes that comprise each structure element are mapped into memory.

```
struct {
    int      a;          /* 0x1112_1314          word          */
    double   b;          /* 0x2122_2324_2526_2728 doubleword */
    char *    c;          /* 0x3132_3334          word          */
    char      d[7];       /* 'A','B','C','D','E','F','G' array of bytes */
    short     e;          /* 0x5152               halfword     */
    int       f;          /* 0x6162_6364          word          */
} s;
```

Note that the C structure mapping introduces padding (skipped bytes) in the map in order to align the scalars on their proper boundaries — four bytes between *a* and *b*, one byte between *d* and *e*, and two bytes between *e* and *f*. Both big- and little-endian mappings use the same amount of padding.

#### 3.2.1.1 Big-Endian Mapping

The big-endian mapping of a structure *S* is shown in [Figure 3-2](#). Addresses are shown in hexadecimal at the left of each double word and in small figures below each byte. The content of each byte, as shown in the preceding C programming example, is shown in hexadecimal as characters for the elements of the string.

# Freescale Semiconductor, Inc.

00	11 00	12 01	13 02	14 03	04	05	06	07
08	21 08	22 09	23 0A	24 0B	25 0C	26 0D	27 0E	28 0F
10	31 10	32 11	33 12	34 13	'A' 14	'B' 15	'C' 16	'D' 17
18	'E' 18	'F' 19	'G' 1A	1B	51 1C	52 1D	1E	1F
20	61 20	62 21	63 22	64 23				

**Figure 3-2 Big-Endian Mapping of Structure S**

## 3.2.1.2 Little-Endian Mapping

**Figure 3-3** shows the structure, S, using little-endian mapping. Double words are laid out from right to left.

07	06	05	04	11 03	12 02	13 01	14 00
21 0F	22 0E	23 0D	24 0C	25 0B	26 0A	27 09	28 08
'D' 17	'C' 16	'B' 15	'A' 14	31 13	32 12	33 11	34 10
1F	1E	51 1D	52 1C	1B	'G' 1A	'F' 19	'E' 18
				61 23	62 22	63 21	64 20

**Figure 3-3 Little-Endian Mapping of Structure S**

## 3.2.2 Data Memory in Little-Endian Mode

This section describes how data in memory is stored and accessed in little-endian mode.

### 3.2.2.1 Aligned Scalars

For load and store instructions, the effective address is computed as specified in the instruction descriptions in **SECTION 4 ADDRESSING MODES AND INSTRUCTION SET SUMMARY**. The effective address is modified as shown in **Table 3-2** before it is used to access memory.



Table 3-2 EA Modifications

Data Width (Bytes)	EA Modification
8	No change
4	XOR with 0b100
2	XOR with 0b110
1	XOR with 0b111

The modified EA is passed to the main memory and the specified width of the data is transferred between a GPR or FPR and the addressed memory locations (as modified). The effective address modification makes it appear to the processor that individual aligned scalars are stored as little-endian, when in fact they are stored as big-endian but in different bytes within double words from the order in which they are stored in big-endian mode.

Taking into account the preceding description of EA modifications, in little-endian mode structure *S* is placed in memory as shown in [Figure 3-4](#).

00	00	01	02	03	11	12	13	14
					04	05	06	07
08	21	22	23	24	25	26	27	28
	08	09	0A	0B	0C	0D	0E	0F
10	'D'	'C'	'B'	'A'	31	32	33	34
	10	11	12	13	14	15	16	17
18	18	19	51	52	1C	'G'	'F'	'E'
			1A	1B		1D	1E	1F
20	20	21	22	23	61	62	63	64
					24	25	26	27

Figure 3-4 PowerPC Little-Endian Structure *S* in Memory

Because of the modifications on the EA, the same structure *S* appears to the processor to be mapped into memory this way when LM = 1 (little-endian enabled). This is shown in [Figure 3-5](#).

# Freescale Semiconductor, Inc.

07	06	05	04	11 03	12 02	13 01	14 00
21 0F	22 0E	23 0D	24 0C	25 0B	26 0A	27 09	28 08
'D' 17	'C' 16	'B' 15	'A' 14	31 13	32 12	33 11	34 10
1F	1E	51 1D	52 1C	1B	'G' 1A	'F' 19	'E' 18
				61 23	62 22	63 21	64 20

**Figure 3-5 PowerPC Little-Endian Structure S as Seen by Processor**

Note that as seen by the program executing in the processor, the mapping for the structure S is identical to the little-endian mapping shown in [Figure 3-3](#). From outside of the processor, the addresses of the bytes making up the structure S are as shown in [Figure 3-4](#). These addresses match neither the big-endian mapping of [Figure 3-2](#) or the little-endian mapping of [Figure 3-3](#). This must be taken into account when performing I/O operations in little-endian mode; this is discussed in [3.2.4 Input/Output in Little-Endian Mode](#).

## 3.2.2.2 Misaligned Scalars

Performing an XOR operation on the low-order bits of the address of a scalar requires the scalar to be aligned on a boundary equal to a multiple of its length. When executing in little-endian mode (LM = 1), the RCPU takes an alignment exception whenever a load or store instruction is issued with a misaligned EA, regardless of whether such an access could be handled without causing an exception in big-endian mode (LM = 0).

The PowerPC architecture defines that half words, words, and double words be placed in memory such that the little-endian address of the lowest-order byte is the EA computed by the load or store instruction; the little-endian address of the next-lowest-order byte is one greater, and so on. [Figure 3-6](#) shows a four-byte word stored at little-endian address 5. The word is presumed to contain the binary representation of 0x1112 1314.

12 07	13 06	14 05	04	03	02	01	00	00
0F	0E	0D	0C	0B	0A	09	11 08	08

**Figure 3-6 PowerPC Little-Endian Mode, Word Stored at Address 5**

**Figure 3-7** shows the same word stored by a little-endian program, as seen by the memory system (assuming big-endian mode).

00	12 00	13 01	14 02	03	04	05	06	07
08	08	09	0A	0B	0C	0D	0E	11 0F

**Figure 3-7 Word Stored at Little-Endian Address 5 as Seen by Big-Endian Addressing**

#### NOTE

The misaligned word in this example spans two double words. The two parts of the misaligned word are not contiguous in the big-endian addressing space.

An implementation may choose to support only a subset of misaligned little-endian memory accesses. For example, misaligned little-endian accesses contained within a single double word may be supported, while those that span double words may cause alignment exceptions.

### 3.2.2.3 String Operations

The load and store string instructions, listed in **Table 3-3**, cause alignment exceptions when they are executed in little-endian mode.

**Table 3-3 Load/Store String Instructions**

Mnemonic	Description
<b>lswi</b>	Load String Word Immediate
<b>lswx</b>	Load String Word Indexed
<b>stswi</b>	Store String Word Immediate
<b>stswx</b>	Store String Word Indexed
<b>lscbx</b>	Load String and Compare Byte Indexed

String accesses are inherently misaligned; they transfer word-length quantities between memory and registers, but the quantities are not necessarily aligned on word boundaries.

#### NOTE

The system software must determine whether to emulate the excepting instruction or treat it as an illegal operation.

### 3.2.2.4 Load and Store Multiple Instructions

The load and store multiple instructions shown in [Table 3-4](#) cause alignment exceptions when executed in little-endian mode.

**Table 3-4 Load/Store Multiple Instructions**

Mnemonic	Instruction
<b>lmw</b>	Load Multiple Word
<b>stmw</b>	Store Multiple Word

Although the words addressed by these instructions are on word boundaries, each word is in the half of its containing double word opposite from where it would be in big-endian mode. Note that the system software must determine whether to emulate the excepting instruction or treat it as an illegal operation.

### 3.2.3 Instruction Memory Addressing in Little-Endian Mode

Each PowerPC instruction occupies 32 bits (one word) of memory. PowerPC processors fetch and execute instructions as if the current instruction address had been advanced one word for each sequential instruction. When operating in little-endian mode, the address is modified according to the little-endian rule for fetching word-length scalars; that is, it is XORed with 0b100. A program is thus an array of little-endian words with each word fetched and executed in order (not including branches).

Consider the following example:

```

loop:
    cmplwi    r5, 0
    beq       done
    lwzux     r4, r5, r6
    add       r7, r7, r4
    subi      r5, 1
    b         loop
done:
    stw       r7, total
  
```

Assuming the program starts at address 0, these instructions are mapped into memory for big-endian execution as shown in [Figure 3-8](#).

# Freescale Semiconductor, Inc.

00	loop: cmplwi r5, 8	beq done
	00 01 02 03	04 05 06 07
08	lwzux r4, r5, r6	add r7, r7, r4
	08 09 0A 0B	0C 0D 0E 0F
10	subi r5, 1	b loop
	10 11 12 13	14 15 16 17
18	done: stw r7, total	
	18 19 1A 1B	1C 1D 1E 1F

**Figure 3-8 PowerPC Big-Endian Instruction Sequence as Seen by Processor**

If this same program is assembled for and executed in little-endian mode, the mapping seen by the processor appears as shown in [Figure 3-9](#).

Each machine instruction appears in memory as a 32-bit integer containing the value described in the instruction description, regardless of whether the processor is operating in big- or little-endian mode. This is because scalars are always mapped in memory in big-endian byte order.

beq done	loop: cmplwi	00
07 06 05 04	03 02 01 00	
add r7, r7, r4	lwzux r4, r5, r6	08
0F 0E 0D 0C	0B 0A 09 08	
b loop	subi r5, 1	10
17 16 15 14	13 12 11 10	
	done: stw r7, total	18
1F 1E 1D 1C	1B 1A 19 18	

**Figure 3-9 PowerPC Little-Endian Instruction Sequence as Seen by Processor**

When little-endian mapping is used, all references to the instruction stream must follow little-endian addressing conventions, including addresses saved in system registers when the exception is taken, return addresses saved in the link register, and branch displacements and addresses.

- An instruction address placed in the link register by branch and link, or an instruction address saved in an SPR when an exception is taken is the address that a program executing in little-endian mode would use to access the instruction as a word of data using a load instruction.
- An offset in a relative branch instruction reflects the difference between the addresses of the instructions, where the addresses used are those that a pro-

gram executing in little-endian mode would use to access the instructions as data words using a load instruction.

- A target address in an absolute branch instruction is the address that a program executing in little-endian mode would use to access the target instruction as a word of data using a load instruction.

### 3.2.4 Input/Output in Little-Endian Mode

Input/output operations transfer a byte stream on both big- and little-endian systems. For a PowerPC system running in big-endian mode, both the processor and the memory subsystem recognize the same byte as byte 0. However, this is not true for a PowerPC system running in little-endian mode because of the modification of the three low-order bits when the processor accesses memory.

In order for I/O transfers in little-endian mode to appear to transfer bytes properly, they must be performed as if the bytes transferred were accessed one at a time, using the little-endian address modification appropriate for the single-byte transfers (XOR the bits with 0b111). This does not mean that I/O on little-endian PowerPC machines must be done using only one-byte-wide transfers. Data transfers can be as wide as desired, but the order of the bytes within double words must be as if they were fetched or stored one at a time.

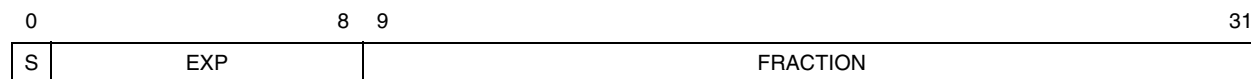
## 3.3 Floating-Point Data

This subsection describes how floating-point data is represented in floating-point registers and in memory.

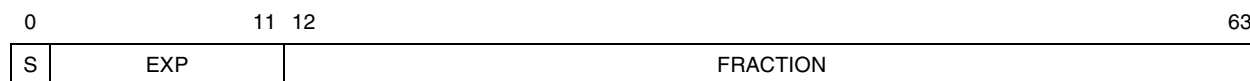
### 3.3.1 Floating-Point Data Format

The PowerPC architecture defines the representation of a floating-point value in two different binary, fixed-length formats: a 32-bit format for a single-precision floating-point value or a 64-bit format for a double-precision floating-point value. Data in memory may use either the single-precision or double-precision format. Floating-point registers use the double-precision format.

The length of the exponent and the fraction fields differ between these two precision formats. The structure of the single-precision format is shown in [Figure 3-10](#); the structure of the double-precision format is shown in [Figure 3-11](#).



**Figure 3-10 Floating-Point Single-Precision Format**



**Figure 3-11 Floating-Point Double-Precision Format**

Values in floating-point format consist of three fields:

- S (sign bit).
- EXP (exponent + bias)
- FRACTION (fraction)

If only a portion of a floating-point data item in memory is accessed, as with a load or store instruction for a byte or half word (or word in the case of floating-point double-precision format), the value affected depends on whether the PowerPC system is using big- or little-endian byte ordering, which is described in [3.2 Byte Ordering](#). Big-endian mode is the default.

The significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit is a one for normalized numbers and a zero for denormalized numbers in the unit bit position (that is, the first bit to the left of the binary point). Parameters for the two floating-point formats are listed in [Table 3-5](#).

**Table 3-5 IEEE Floating-Point Fields**

Parameter	Single-Precision	Double-Precision
Exponent bias	+127	+1023
Maximum exponent (unbiased)	+127	+1023
Minimum exponent	-126	-1022
Format width	32 bits	64 bits
Sign width	1 bit	1 bit
Exponent width	8 bits	11 bits
Fraction width	23 bits	52 bits
Significand width	24 bits	53 bits

The exponent is expressed as an 8-bit value for single-precision numbers or an 11-bit value for double-precision numbers. These bits hold the biased exponent; the true value of the exponent can be determined by subtracting 127 for single-precision numbers and 1023 for double-precision values. This is shown in [Figure 3-12](#). Note that using a bias eliminates the need for a sign bit. The highest-order bit is

## Freescale Semiconductor, Inc.

used both to generate the number, and is an implicit sign bit. Note also that two values are reserved — all bits set indicates that the number is an infinity or NaN and all bits cleared indicates that the number is either zero or denormalized.

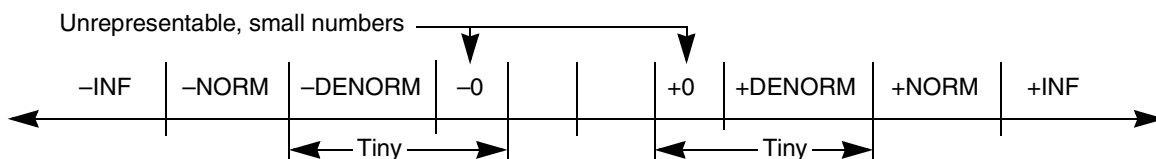
### 3.3.2 Value Representation

The PowerPC architecture defines numerical and non-numerical values representable within single- and double-precision formats. The numerical values are approximations to the real numbers and include the normalized numbers, denormalized numbers, and zero values. The non-numerical values representable are the positive and negative infinities and the NaNs. The positive and negative infinities are adjoined to the real numbers but are not numbers themselves, and the standard rules of arithmetic do not hold when they appear in an operation. They are related to the real numbers by “order” alone. It is possible, however, to define restricted operations among numbers and infinities as defined in the following paragraphs. The relative location on the real number line for each of the defined entities is shown in **Figure 3-13**.

Biased Exponent (binary)		Single-Precision (unbiased)	Double-Precision (unbiased)
Positive	11 . . . . 11	Reserved for Infinities and NaNs	
	11 . . . . 10	+127	+1023
	11 . . . . 01	+126	+1022
	.	.	.
	.	.	.
Zero	10 . . . . 00	1	1
	01 . . . . 11	0	0
Negative	01 . . . . 10	-1	-1
	.	.	.
	.	.	.
	.	.	.
	00 . . . . 01	-126	-1022
00 . . . . 00		Reserved for Zeros and Denormalized Numbers	

**Figure 3-12 Biased Exponent Format**





**Figure 3-13 Approximation to Real Numbers**

The positive and negative NaNs are not related to the numbers or  $\pm x$  by order or value, but they are encodings that convey diagnostic information such as the representation of uninitialized variables.

**Table 3-6** describes each of the floating-point formats.

**Table 3-6 Recognized Floating-Point Numbers**

Sign Bit	Exponent (Biased)	Leading Bit	Mantissa	Value
0	Maximum	x	Non-zero	+NaN
0	Maximum	x	Zero	+Infinity
0	0 < Exponent < Maximum	1	Non-zero	+Normalized
0	0	0	Non-zero	+Denormalized
0	0	0	Zero	+0
1	0	0	Zero	−0
1	0	0	Non-zero	−Denormalized
1	0 < Exponent < Maximum	1	Non-zero	−Normalized
1	Maximum	x	Zero	−Infinity
1	Maximum	x	Non-zero	−NaN

### 3.3.3 Normalized Numbers ( $\pm$ NORM)

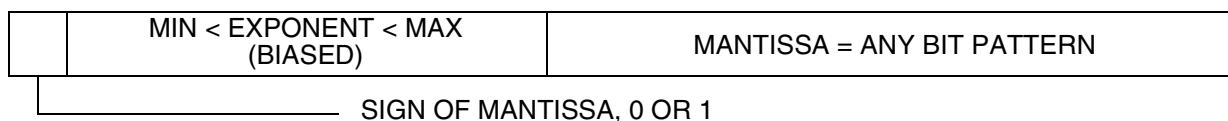
The values for normalized numbers have a biased exponent value in the range:

- 1–254 in single-precision format
- 1–2046 in double-precision format

The implied unit bit is one. Normalized numbers are interpreted as follows:

$$\text{NORM} = (-1)^s \times 2^E \times (1.\text{fraction})$$

where (s) is the sign, (E) is the unbiased exponent and (1.fraction) is the significand composed of a leading unit bit (implied bit) and a fractional part. The format for normalized numbers is shown in **Figure 3-14**.



**Figure 3-14 Format for Normalized Numbers**

The ranges covered by the magnitude (M) of a normalized floating-point number are approximately equal to the following:

**Single-precision format:**

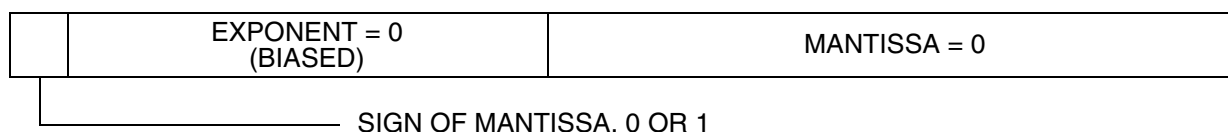
$$1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$$

**Double-precision format:**

$$2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$$

### 3.3.4 Zero Values ( $\pm 0$ )

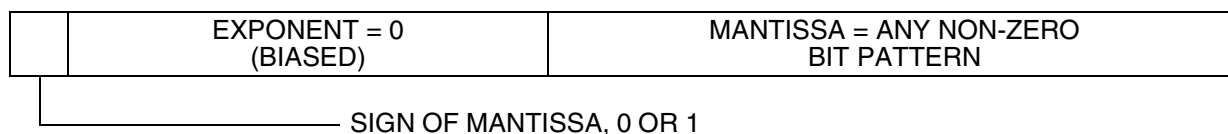
Zero values have a biased exponent value of zero and a fraction value of zero. This is shown in [Figure 3-15](#). Zeros can have a positive or negative sign. The sign of zero is ignored by comparison operations (that is, comparison regards +0 as equal to -0).



**Figure 3-15 Format for Zero Numbers**

### 3.3.5 Denormalized Numbers ( $\pm \text{DENORM}$ )

Denormalized numbers have a biased exponent value of zero and a non-zero fraction value. The format for denormalized numbers is shown in [Figure 3-16](#).



**Figure 3-16 Format for Denormalized Numbers**

Denormalized numbers are non-zero numbers smaller in magnitude than the representable normalized numbers. They are values in which the implied unit bit is zero. Denormalized numbers are interpreted as follows:

# Freescale Semiconductor, Inc.

$$\text{DENORM} = (-1)^s \times 2^{\text{Emin}} \times (0.\text{fraction})$$

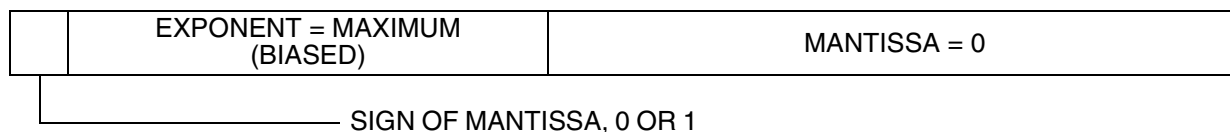
Emin is the minimum representable exponent value (that is, -126 for single-precision, -1022 for double-precision).

## 3.3.6 Infinities (±x)

Positive and negative infinities have the maximum biased exponent value:

- 255 in the single-precision format
- 2047 in the double-precision format

The format for infinities is shown in [Figure 3-17](#).



**Figure 3-17 Format for Positive and Negative Infinities**

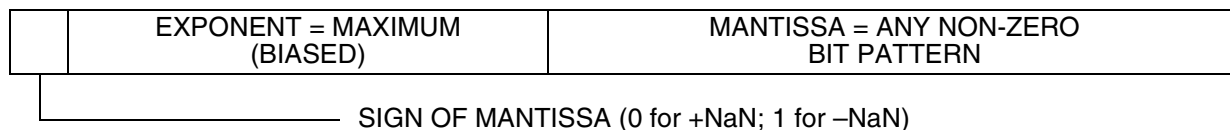
The fraction value is zero. Infinities are used to approximate values greater in magnitude than the maximum normalized value. Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined between numbers and infinities. Infinities and the reals can be related as follows:

$$-x < \text{every finite number} < +x$$

Arithmetic using infinite numbers is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in [6.11.10.6 Invalid Operation Exception Conditions](#).

## 3.3.7 Not a Numbers (NaNs)

NaNs have the maximum biased exponent value and a non-zero fraction value. The format for NaNs is shown in [Figure 3-18](#). The sign bit of NaNs is ignored (that is, NaNs are neither positive nor negative). If the high-order bit of the fraction field is a zero, the NaN is a signaling NaN (SNaN); otherwise it is a quiet NaN (QNaN).



**Figure 3-18 Format for NaNs**

Signaling NaNs signal exceptions when they are specified as arithmetic operands.

Quiet NaNs represent the results of certain invalid operations, such as invalid arith-

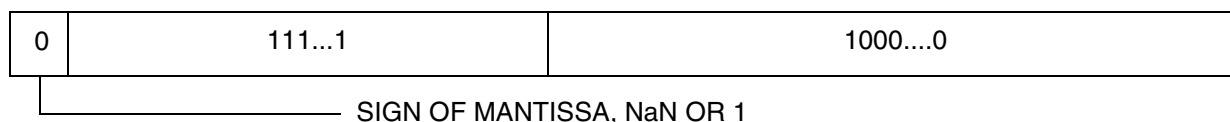
metic operations on infinities or on NaNs, when the invalid operation exception is disabled (FPSCR[VE] = 0). Quiet NaNs propagate through all operations, except ordered comparison, floating round to single precision, and conversion to integer operations. Quiet NaNs do not signal exceptions, except during ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of operations and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN results from an operation because an operand is a NaN or because a QNaN is generated due to a disabled invalid operation exception, the following rule is applied to determine the QNaN with the high-order fraction bit set to one that is to be stored as the result:

```

If (frA) is a NaN
Then frD ← (frA)
  Else if (frB) is a NaN
    Then frD ← (frB)
  Else if (frC) is a NaN
    Then frD ← (frC)
  Else if generated QNaN
    Then frD ← generated QNaN
    
```

If the operand specified by **frA** is a NaN, that NaN is stored as the result. Otherwise, if the operand specified by **frB** is a NaN (if the instruction specifies an **frB** operand), that NaN is stored as the result. Otherwise, if the operand specified by **frC** is a NaN (if the instruction specifies an **frC** operand), that NaN is stored as the result. Otherwise, if a QNaN is generated by a disabled invalid operation exception, that QNaN is stored as the result. If a QNaN is to be generated as a result, the QNaN generated has a sign bit of zero, an exponent field of all ones, and a high-order fraction bit of one with all other fraction bits zero. An instruction that generates a QNaN as the result of a disabled invalid operation generates this QNaN. This is shown in **Figure 3-19**.



**Figure 3-19 Representation of QNaN**

### 3.3.8 Sign of Result

The following rules govern the sign of the result of an arithmetic operation, when the operation does not yield an exception. These rules apply even when the operands or results are  $\pm 0$  or  $\pm x$ .

The sign of the result of an addition operation is the sign of the source operand having the larger absolute value. The sign of the result of the subtraction operation,  $x - y$ , is the same as the sign of the result of the addition operation,  $x + (-y)$ .

When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except round toward negative infinity ( $-\infty$ ), in which case the sign is negative.

- The sign of the result of a multiplication or division operation is the exclusive OR of the signs of the source operands.
- The sign of the result of a round to single-precision or convert to/from integer operation is the sign of the source operand.

For multiply-add instructions, these rules are applied first to the multiplication operation and then to the addition or subtraction operation (one of the source operands to the addition or subtraction operation is the result of the multiplication operation).

## 3.3.9 Normalization and Denormalization

When an arithmetic operation produces an intermediate result, consisting of a sign bit, an exponent, and a non-zero significand with a zero leading bit, the result is not a normalized number and must be normalized before it is stored.

A number is normalized by shifting its significand left while decrementing its exponent by one for each bit shifted, until the leading significand bit becomes one. The guard bit and the round bit participate in the shift with zeros shifted into the round bit; see [3.4.1 Execution Model for IEEE Operations](#).

During normalization, the exponent is regarded as if its range were unlimited. If the resulting exponent value is less than the minimum value that can be represented in the format specified for the result, the intermediate result is said to be “tiny” and the stored result is determined by the rules described in [6.11.10.9 Underflow Exception Condition](#). The sign of the number does not change.

When an arithmetic operation produces a non-zero intermediate result whose exponent is less than the minimum value that can be represented in the format specified, the stored result may need to be denormalized. The result is determined by the rules described in [6.11.10.9 Underflow Exception Condition](#).

A number is denormalized by shifting its significand to the right while incrementing its exponent by one for each bit shifted until the exponent equals the format's minimum value. If any significant bits are lost in this shifting process, a loss of accuracy has occurred, and an underflow exception is signaled. The sign of the number does not change.

When denormalized numbers are operands of multiply and divide operations, operands are prenormalized internally before the operations are performed.

## 3.3.10 Data Handling and Precision

There are specific instructions for moving floating-point data between the FPRs and memory. Data in double-precision format is not altered during the move. Sin-

## Freescale Semiconductor, Inc.

gle-precision data is converted to double-precision format when loaded from memory into an FPR. A format conversion from double- to single-precision is performed when data from an FPR is stored. Floating-point exceptions cannot occur during these operations.

All arithmetic operations use floating-point double-precision format.

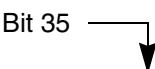
Floating-point single-precision formats are used by the following four types of instructions:

- **Load Floating-Point Single-Precision (lfs)** — This instruction accesses a single-precision operand in single-precision format in memory, converts it to double-precision, and loads it into an FPR. Exceptions are not detected during the load operation.
- **Round to floating-point single-precision** — If the operand is not already in single-precision range, the floating round to single-precision instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective enable bits in the FPSCR. The instruction places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions and by single-precision loads, this operation does not alter the value.
- **Single-precision arithmetic instructions** — These instructions take operands from the FPRs in double-precision format, perform the operation as if it produced an intermediate result correct to infinite precision and with unbounded range, and then force this intermediate result to fit in single-precision format. Status bits in the FPSCR and in the condition register are set to reflect the single-precision result. The result is then converted to double-precision format and placed into an FPR. The result falls within the range supported by the single format.

For single-precision operations, source operands must be representable in single-precision format. If they are not, the result placed into the target FPR, and the setting of status bits in the FPSCR and in the condition register, are undefined.

- **Store Floating-Point Single-Precision (stfs)** — This form of instruction converts a double-precision operand to single-precision format and stores that operand into memory. If the operand requires denormalization in order to fit in single-precision format, it is automatically denormalized prior to being stored. No exceptions are detected on the store operation (the value being stored is effectively assumed to be the result of an instruction of one of the preceding three types).

When the result of a load floating-point single-precision (**lfs**), floating-point round to single-precision (**frspx**), or single-precision arithmetic instruction is stored in an FPR, the low-order 29 fraction bits are zero. This is shown in [Figure 3-20](#).



### Figure 3-20 Single-Precision Representation in an FPR

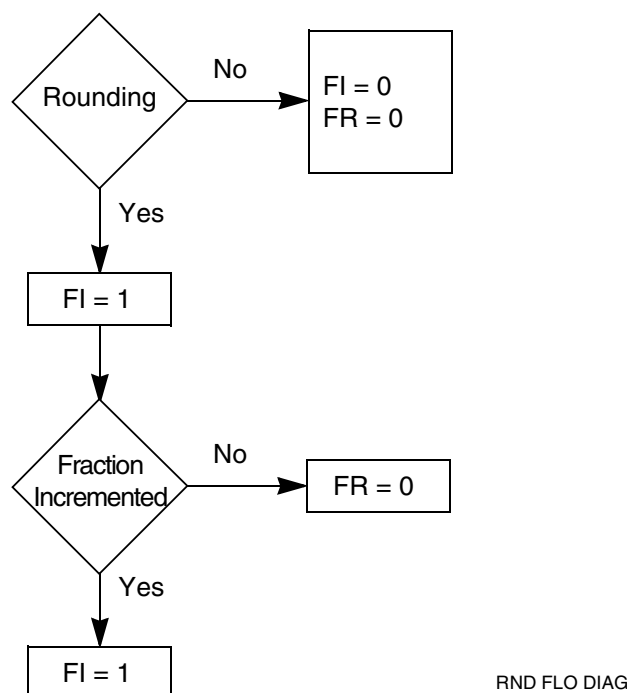
The floating-point round to single-precision (**frpsx**) instruction allows conversion from double to single precision with appropriate exception checking and rounding. This instruction should be used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions) to single-precision values before storing them into single-format memory elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions can be stored directly, or used directly as operands for single-precision arithmetic instructions, without preceding the store, or the arithmetic instruction, by **frspx**.

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value can be represented in single-precision format. Some implementations may execute single-precision arithmetic instructions faster than double-precision arithmetic instructions. Therefore, if double-precision accuracy is not required, using single-precision data and instructions can speed operations.

### 3.3.11 Rounding

All arithmetic instructions defined by the PowerPC architecture produce an intermediate result considered infinitely precise. This result must then be written with a precision of finite length into an FPR. After normalization or denormalization, if the infinitely precise intermediate result cannot be represented in the precision required by the instruction, it is rounded before being placed into the target FPR.

The instructions that potentially round their result are the arithmetic, multiply-add, and rounding and conversion instructions. As shown in **Figure 3-21**, whether rounding occurs depends on the source values.



**Figure 3-21 Rounding Flow Diagram**

Each of these instructions sets FPSCR bits FR and FI, according to whether rounding occurs (FI) and whether the fraction was incremented (FR). If rounding occurs, FI is set to one and FR may be either zero or one. If rounding does not occur, both FR and FI are cleared. Other floating-point instructions do not alter FR and FI. Four modes of rounding are provided that are user-selectable through the floating-point rounding control field in the FPSCR. These are encoded as follows in [Table 3-7](#).

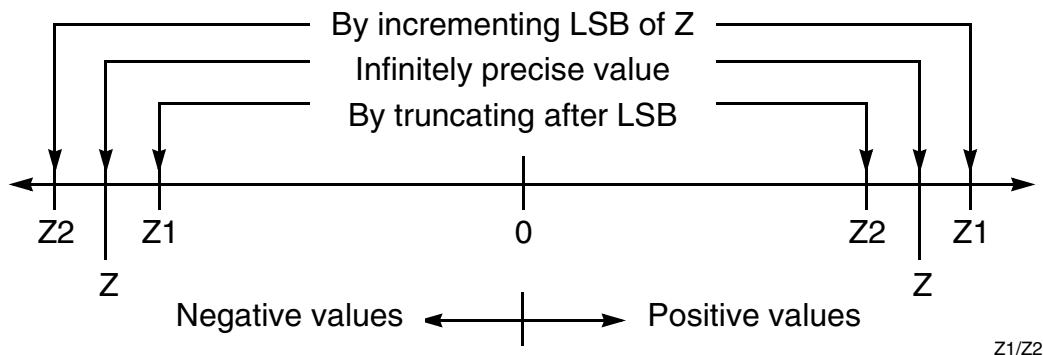
**Table 3-7 FPSCR Bit Settings — RN Field**

RN	Rounding Mode
00	Round to nearest
01	Round toward zero
10	Round toward +infinity
11	Round toward -infinity

Let Z be the infinitely precise intermediate arithmetic result or the operand of a conversion operation. If Z can be represented exactly in the target format, no rounding occurs and the result in all rounding modes is equivalent to truncation of Z. If Z cannot be represented exactly in the target format, let Z1 and Z2 be the next larger and next smaller numbers representable in the target format that bound Z; then Z1 or Z2 can be used to approximate the result in the target format.



**Figure 3-22** shows a graphical representation of Z, Z1, and Z2.



**Figure 3-22 Relation of Z1 and Z2**

Rounding follows the four following rules:

- Round to nearest — Choose the best approximation (Z1 or Z2). In case of a tie, choose the one which is even (i.e., with least significant bit equal to zero). Refer to [3.4.1 Execution Model for IEEE Operations](#) for details on how the processor selects the best approximation.
- Round toward zero — Choose the smaller in magnitude (Z1 or Z2).
- Round toward +infinity — Choose Z1.
- Round toward -infinity — Choose Z2.

If Z is to be rounded up and Z1 does not exist (that is, if there is no number larger than Z that is representable in the target format), then an overflow exception occurs if Z is positive and an underflow exception occurs if Z is negative. Similarly, if Z is to be rounded down and Z2 does not exist, then an overflow exception occurs if Z is negative and an underflow exception occurs if Z is positive. The results in these cases are defined in [6.11.10 Floating-Point Assist Exception \(0x00E00\)](#).

### 3.4 Floating-Point Execution Models

The following paragraphs describe the floating-point execution models for IEEE operations, as well as that for a special multiply-add type of instruction. In addition, the execution model for non-IEEE compliant operation, used to accelerate time-critical operations, is described.

The IEEE-754 standard includes 32-bit and 64-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The PowerPC architecture follows these guidelines:

- Double-precision arithmetic instructions can have operands of either or both precisions.

## Freescal Semiconductor, Inc.

- Single-precision arithmetic instructions require all operands to be single-precision.
- Double-precision arithmetic instructions produce double-precision values.
- Single-precision arithmetic instructions produce single-precision values.

For arithmetic instructions, conversions from double- to single-precision must be done explicitly by software, while conversions from single- to double-precision are done implicitly.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following cases when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized factor.
- Overflow during division using a denormalized divisor.

### 3.4.1 Execution Model for IEEE Operations

The following description uses 64-bit arithmetic as an example. Thirty-two-bit arithmetic is similar except that the fraction field is a 23-bit field and the single-precision guard, round, and sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION (or mantissa) field.

The bits and fields for the IEEE 64-bit execution model are defined as follows:

- The S bit is the sign bit.
- The C bit is the carry bit that captures the carry out of the significand.
- The L bit is the leading unit bit of the significand which receives the implicit bit from the operands.
- The FRACTION is a 52-bit field that accepts the fraction (mantissa) of the operands.
- The guard (G), round (R), and sticky (X) bits are extensions to the low-order bits of the accumulator. The G and R bits are required for post normalization of the result. The G, R, and X bits are required during rounding to determine if the intermediate result is equally near the two nearest representable values. The X bit serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, either due to shifting the accumulator right or other generation of low-order result bits. The G and R bits participate in the left shifts with zeros being shifted into the R bit. **Table 3-8** shows the relationship among the G, R, and X bits, the intermediate result (IR), the next lower in magnitude representable number (NL), and the next higher in magnitude representable number (NH).

Table 3-8 Interpretation of G, R, and X Bits

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	IR closer to NL
0	1	0	
0	1	1	
1	0	0	IR midway between NL and NH
1	0	1	IR closer to NH
1	1	0	
1	1	1	

The significand of the intermediate result is made up of the L bit, the FRACTION, and the G, R, and X bits.

The infinitely precise intermediate result of an operation is the result normalized in bits L, FRACTION, G, R, and X of the floating-point accumulator.

Before results are stored into an FPR, the significand is rounded if necessary, using the rounding mode specified by FPSCR[RN]. If rounding causes a carry into C, the significand is shifted right one position and the exponent is incremented by one. This may yield an inexact result and possibly exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR, and low-order bit positions, if any, are set to zero.

Four rounding modes are provided which are user-selectable through FPSCR[RN] as described in [3.3.11 Rounding](#). For rounding, the conceptual guard, round, and sticky bits are defined in terms of accumulator bits.

**Table 3-9** shows the positions of the guard, round, and sticky bits for double-precision and single-precision floating-point numbers.

Table 3-9 Location of the Guard, Round and Sticky Bits

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	26–52 G,R,X

Rounding can be treated as though the significand were shifted right, if required, until the least significant bit to be retained is in the low-order bit position of the FRACTION. If any of the guard, round, or sticky bits are non-zero, the result is inexact.

## Freescale Semiconductor, Inc.

Z1 and Z2, defined in [3.3.11 Rounding](#), can be used to approximate the result in the target format when one of the following rules is used:

- Round to nearest
  - Guard bit = 0: The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011))
  - Guard bit = 1: Depends on round and sticky bits:
    - Case a: If the round or sticky bit is one (inclusive), the result is incremented. (result closest to next higher value in magnitude (GRX = 101, 110, or 111))
    - Case b: If the round and sticky bits are zero (i.e., the result is midway between the closest representable values), the result is rounded to an even value. That is, if the low-order bit of the result is one, the result is incremented. If the low-order bit of the result is zero, the result is truncated.
- If during the round to nearest process, truncation of the unrounded number produces the maximum magnitude for the specified precision, the following action is taken:
  - Guard bit = 1: Store infinity with the sign of the unrounded result.
  - Guard bit = 0: Store the truncated (maximum magnitude) value.
- Round toward zero — Choose the smaller in magnitude of Z1 or Z2. If the guard, round, or sticky bit is non-zero, the result is inexact.
- Round toward +infinity  
Choose Z1.
- Round toward -infinity  
Choose Z2.

Where the result is to have fewer than 53 bits of precision because the instruction is a floating round to single-precision or single-precision arithmetic instruction, the intermediate result either is normalized or is placed in correct denormalized form before the result is potentially rounded.

### 3.4.2 Execution Model for Multiply-Add Type Instructions

The PowerPC architecture makes use of a special form of instruction that performs up to three operations in one instruction (a multiply, an add, and a negate). With this added capability is the special feature of being able to produce a more exact intermediate result as an input to the rounder. The 32-bit arithmetic is similar except that the fraction field is smaller.

#### NOTE

The rounding occurs only after add; therefore, the computation of the sum and product together are infinitely precise before the final result is rounded to a representable format.

The first part of the operation is a multiply. The multiply has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, placing the L bit into the most significant bit of the FRACTION and placing the C bit into the L bit. All 106 bits (L bit plus the frac-

tion) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the add is then normalized, with all bits of the add result, except the X' bit, participating in the shift. The normalized result provides an intermediate result as input to the rounder that conforms to the model described in **3.4.1 Execution Model for IEEE Operations**, where:

- The guard bit is bit 53 of the intermediate result.
- The round bit is bit 54 of the intermediate result.
- The sticky bit is the OR of all remaining bits to the right of bit 55, inclusive.

If the instruction is floating negative multiply-add or floating negative multiply-subtract, the final result is negated.

Status bits are set to reflect the result of the entire operation: for example, no status is recorded for the result of the multiplication part of the operation.

### 3.4.3 Non-IEEE Operation

The RCPU depends on a software envelope to fully implement the IEEE-754 floating-point specification. Even when all exceptions are disabled (i.e., when exception enable bits in the FPSCR are cleared), tiny results and denormalized operands cause FPU exceptions that invoke a software routine to deliver (with hardware assistance) the correct IEEE result.

To accelerate time-critical operations and make them more deterministic, the RCPU provides a non-IEEE mode of operation. In this mode, whenever a tiny result is detected and floating-point underflow exception is disabled (FPSCR[UE] = 0), the hardware delivers a correctly signed zero instead of invoking the floating-point assist exception handler.

Non-IEEE mode is entered by setting the NI (non-IEEE enable) bit in the FPSCR.

Denormalized numbers are never generated in non-IEEE mode. Therefore, when denormalized operands are detected, they are treated exactly as they are in IEEE mode. Refer to **6.11.10 Floating-Point Assist Exception (0x00E00)** for more information.

The hardware never asserts the FPSCRXX (inexact) bit on an underflow condition; it is done as a part of the floating-point assist interrupt handler. Therefore, in non-IEEE mode, FPSCRXX cannot be depended upon to be a complete accumulation of all inexact conditions.

## 3.4.4 Working Without the Software Envelope

Even when the processor is operating in non-IEEE mode, the software envelope may be invoked when denormalized numbers are used as the input to the calculation or when an enabled IEEE exception is detected. To ensure that the software envelope is never invoked, the user needs to do the following:

- Set the NI bit in the FPSCR to enable non-IEEE mode.
- Disable all floating-point exceptions.
- Avoid using denormalized numbers as inputs to floating-point calculations.

## SECTION 4

### ADDRESSING MODES AND INSTRUCTION SET SUMMARY

This section describes instructions and address modes supported by the RCPU. These instructions are divided into the following categories:

- Integer instructions — These include computational and logical instructions.
- Floating-point instructions — These include floating-point computational instructions, as well as instructions that affect the floating-point status and control register.
- Load/store instructions — These include integer and floating-point load and store instructions.
- Flow control instruction — These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
- Processor control instruction — These instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs), and to read from the time base register (TBU or TBL).
- Memory synchronization instructions — These instructions are used for synchronizing memory.
- Memory control instructions — These instructions provide control of the I-cache.

Notice that this grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions. This information is provided in [SECTION 9 INSTRUCTION SET](#).

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC architecture uses instructions that are four bytes long and word-aligned. It provides for byte, half-word, and word operand fetches and stores between memory and a set of 32 general-purpose registers (GPRs). It also provides for word and double-word operand fetches and stores between memory and a set of 32 floating-point registers (FPRs).

Arithmetic and logical instructions do not modify memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location.

#### 4.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next sequential instruction.



#### 4.1.1 Memory Operands

Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The address of a memory operand is the address of its lowest-numbered byte. Operand length is implicit for each instruction. The PowerPC architecture supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian; see [3.2 Byte Ordering](#) for more information.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. For a detailed discussion of memory operands, see [SECTION 3 OPERAND CONVENTIONS](#).

#### 4.1.2 Addressing Modes and Effective Address Calculation

A program references memory using the effective address (EA) computed by the processor when it executes a memory access or branch instruction, or when it fetches the next sequential instruction.

The effective address is the 32-bit address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the storage operand is considered to wrap around from the maximum effective address to effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

Load and store operations have three categories of effective address generation:

- Register indirect with immediate index mode. The d operand is added to the contents of the GPR specified by the rA operand to generate the effective address.
- Register indirect with index mode. The contents of the GPR specified by rB operand are added to the contents of the GPR specified by the rA operand to generate the effective address.
- Register indirect mode. The contents of the GPR specified by the rA operand are used as the effective address.

Branch instructions have three categories of effective address generation:

- Immediate addressing. The BD or LI operands are sign extended with the two low-order bits cleared to zero to generate the branch effective address.
- Link register indirect. The contents of the link register with the two low-order bits cleared to zero are used as the branch effective address.
- Counter register indirect. The contents of the counter register with the two low-order bits cleared to zero are used as the branch effective address.

Branch instructions can optionally load the link register with the next sequential instruction address (current instruction address + 4).



## 4.2 Classes of Instructions

PowerPC instructions belong to one of three classes:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among the PowerPC processors, the assignment of these classifications is not. For example, an instruction that is specific to 64-bit implementations is considered defined for 64-bit implementations but illegal for 32-bit implementations such as the RCPU.

The class is determined by examining the primary opcode and the extended opcode, if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or a reserved instruction, the instruction is illegal.

In future versions of the PowerPC architecture, instruction codings that are now illegal may become defined (by being added to the architecture) or reserved (by being assigned to one of the special purposes). Likewise, reserved instructions may become defined.

### 4.2.1 Definition of Boundedly Undefined

The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary sequence of instructions, starting in the state the machine was in before executing the given instruction. Boundedly undefined results for a given instruction may vary between implementations and between execution attempts on the same implementation.

### 4.2.2 Defined Instruction Class

Defined instructions include all the instructions defined in the PowerPC UISA, VEA, and OEA. Defined instructions can be required or optional. The RCPU supports the following defined instructions:

- All 32-bit PowerPC UISA required instructions
- The following PowerPC VEA instructions: **eieio**, **icbi**, **isync**, and **mftb**
- The following PowerPC OEA instructions: **mfmsr**, **mf spr**, **mtmsr**, **mts pr**, **r fi**, and **sc**.
- The following optional instruction: **stfiwx**

A defined instruction may have an instruction form that is invalid if one or more operands, excluding opcodes, are coded incorrectly in a manner that can be deduced by examining only the instruction encoding (primary and extended opcodes). For example, an invalid form results when a reserved bit (shown as “0” in the instruction descriptions in **SECTION 9 INSTRUCTION SET**) is set to one.

Attempting to execute an invalid form of a defined instruction either invokes the software emulation instruction error handler or yields boundedly undefined results. Where not otherwise noted in the individual instruction descriptions in **SECTION 9 INSTRUCTION SET** for individual instruction descriptions, attempting to execute

an instruction in which a reserved bit is set to one yields the same result as executing the instruction with the reserved bit cleared to zero.

Attempting to execute a defined PowerPC instruction, including an optional instruction, that is not implemented in hardware causes the RCPU to take the implementation dependent software emulation exception.

## NOTE

Other PowerPC implementations invoke the program exception handler in this case. Refer to [6.11.11 Software Emulation Exception \(0x01000\)](#) for additional information.

### 4.2.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions that are not implemented in the PowerPC architecture. These opcodes are available for future extensions of the PowerPC architecture; that is, future versions of the PowerPC architecture may define any of these instructions to perform new functions.
- Instructions that are implemented in the PowerPC architecture but are not implemented in a specific PowerPC implementation. For example, instructions that can be executed on 64-bit PowerPC processors are considered illegal for 32-bit processors.
- All unused extended opcodes are illegal.
- An instruction consisting entirely of zeros is guaranteed to be an illegal instruction.

An attempt to execute an illegal instruction invokes the software emulation error handler. Notice that in other PowerPC implementations, the program exception handler may be invoked in this case.

### 4.2.4 Reserved Instruction Class

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC architecture. Attempting to execute an unimplemented reserved instruction causes the RCPU to take the implementation dependent software emulation exception.

## NOTE

Other PowerPC implementations invoke the program exception handler in this case. Refer to [6.11.11 Software Emulation Exception \(0x01000\)](#) for additional information.

### 4.3 Integer Instructions

This section describes the integer instructions. These consist of the following:

- Integer arithmetic instructions
- Integer compare instructions

## Freescale Semiconductor, Inc.

- Integer rotate and shift instructions
- Integer logical instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs, into the integer exception register (XER), and into condition register fields.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as an unsigned operation or an address conversion.

The integer instructions that update the condition register (i.e., those with a mnemonic ending in a period) set condition register field CR0 (bits [0:3]) to characterize the result of the operation. These instructions include those with the Rc bit equal to one and the **addic.**, **andi.**, and **andis.** integer logical and arithmetic instructions. The condition register field CR0 is set as if the result were compared algebraically to zero.

The following integer arithmetic instructions always set XER[CA] to reflect the carry out of bit 0: **addic**, **addic.**, **subfic**, **addc**, **subfc**, **adde**, **subfe**, **addme**, **subfme**, **addze**, and **subfze**. Integer arithmetic instructions with the overflow enable (OE) bit set cause XER[SO] and XER[OV] to be set to reflect overflow of the 32-bit result.

Unless otherwise noted, when condition register field CR0 and the XER are affected, they reflect the value placed in the target register.

The RCPU performs best for aligned load and store operations. See [6.11.4 Alignment Exception \(0x00600\)](#) for scenarios that cause an alignment exception.

### 4.3.1 Integer Arithmetic Instructions

[Table 4-1](#) lists the integer arithmetic instructions.

## Table 4-1 Integer Arithmetic Instructions

Name	Mnemonic	Operand Syntax	Operation
Add Immediate	<b>addi</b>	rD,rA,SIMM	The sum (rA 0) + SIMM is placed into register rD.
Add Immediate Shifted	<b>addis</b>	rD,rA,SIMM	The sum (rA 0) + (SIMM    0x0000) is placed into register rD.
Add	<b>add</b> <b>add.</b> <b>addo</b> <b>addo.</b>	rD,rA,rB	<p>The sum (rA) + (rB) is placed into register rD.</p> <p><b>add</b>        <b>Add</b>  <b>add.</b>       <b>Add</b> with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>addo</b>       <b>Add</b> with Overflow Enabled. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>addo.</b>       <b>Add</b> with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Subtract from	<b>subf</b> <b>subf.</b> <b>subfo</b> <b>subfo.</b>	rD,rA,rB	<p>The sum <math>\neg(rA) + (rB) + 1</math> is placed into rD.</p> <p><b>subf</b>       Subtract from</p> <p><b>subf.</b>       Subtract from with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>subfo</b>       Subtract from with Overflow Enabled. The o suffix enables the overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>subfo.</b>       Subtract from with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Add Immediate Carrying	<b>addic</b>	rD,rA,SIMM	The sum (rA) + SIMM is placed into register rD.
Add Immediate Carrying and Record	<b>addic.</b>	rD,rA,SIMM	The sum (rA) + SIMM is placed into rD. The condition register is updated.
Subtract from Immediate Carrying	<b>subfic</b>	rD,rA,SIMM	The sum $\neg(rA) + SIMM + 1$ is placed into register rD.
Add Carrying	<b>addc</b> <b>addc.</b> <b>addco</b> <b>addco.</b>	rD,rA,rB	<p>The sum (rA) + (rB) is placed into register rD.</p> <p><b>addc</b>       Add Carrying</p> <p><b>addc.</b>       Add Carrying with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>addco</b>       Add Carrying with Overflow Enabled. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>addco.</b>       Add Carrying with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>

## Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Subtract from Carrying	<b>subfc</b> <b>subfc.</b> <b>subfco</b> <b>subfco.</b>	rD,rA,rB	<p>The sum <math>\neg(rA) + (rB) + 1</math> is placed into register rD.</p> <p><b>subfc</b> Subtract from Carrying</p> <p><b>subfc.</b> Subtract from Carrying with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>subfco</b> Subtract from Carrying with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>subfco.</b> Subtract from Carrying with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Add Extended	<b>adde</b> <b>adde.</b> <b>addeo</b> <b>addeo.</b>	rD,rA,rB	<p>The sum <math>(rA) + (rB) + XER(CA)</math> is placed into register rD.</p> <p><b>adde</b> Add Extended</p> <p><b>adde.</b> Add Extended with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>addeo</b> Add Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>addeo.</b> Add Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Subtract from Extended	<b>subfe</b> <b>subfe.</b> <b>subfeo</b> <b>subfeo.</b>	rD,rA,rB	<p>The sum <math>\neg(rA) + (rB) + XER(CA)</math> is placed into register rD.</p> <p><b>subfe</b> Subtract from Extended</p> <p><b>subfe.</b> Subtract from Extended with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>subfeo</b> Subtract from Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>subfeo.</b> Subtract from Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow (OV) bit in the XER.</p>
Add to Minus One Extended	<b>addme</b> <b>addme.</b> <b>addmeo</b> <b>addmeo.</b>	rD,rA	<p>The sum <math>(rA) + XER(CA) + 0xFFFF FFFF</math> is placed into register rD.</p> <p><b>addme</b> Add to Minus One Extended</p> <p><b>addme.</b> Add to Minus One Extended with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>addmeo</b> Add to Minus One Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>addmeo.</b> Add to Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow (OV) bit in the XER.</p>
Subtract from Minus One Extended	<b>subfme</b> <b>subfme.</b> <b>subfmeo</b> <b>subfmeo.</b>	rD,rA	<p>The sum <math>\neg(rA) + XER(CA) + 0xFFFF FFFF</math> is placed into register rD.</p> <p><b>subfme</b> Subtract from Minus One Extended</p> <p><b>subfme.</b> Subtract from Minus One Extended with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>subfmeo</b> Subtract from Minus One Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>subfmeo.</b> Subtract from Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>

## Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Add to Zero Extended	<b>addze</b> <b>addze.</b> <b>addzeo</b> <b>addzeo.</b>	rD,rA	<p>The sum <math>(rA) + XER(CA)</math> is placed into register rD.</p> <p><b>addze</b> Add to Zero Extended</p> <p><b>addze.</b> Add to Zero Extended with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>addzeo</b> Add to Zero Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>addzeo.</b> Add to Zero Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Subtract from Zero Extended	<b>subfze</b> <b>subfze.</b> <b>subfzeo</b> <b>subfzeo.</b>	rD,rA	<p>The sum <math>\neg(rA) + XER(CA)</math> is placed into register rD.</p> <p><b>subfze</b> Subtract from Zero Extended</p> <p><b>subfze.</b> Subtract from Zero Extended with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>subfzeo</b> Subtract from Zero Extended with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>subfzeo.</b> Subtract from Zero Extended with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Negate	<b>neg</b> <b>neg.</b> <b>nego</b> <b>nego.</b>	rD,rA	<p>The sum <math>\neg(rA) + 1</math> is placed into register rD.</p> <p><b>neg</b> Negate</p> <p><b>neg.</b> Negate with CR Update. The dot suffix enables the update of the condition register.</p> <p><b>nego</b> Negate with Overflow. The o suffix enables the overflow bit (OV) in the XER.</p> <p><b>nego.</b> Negate with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Multiply Low Immediate	<b>mulli</b>	rD,rA,SIMM	<p>The low-order 32 bits of the 48-bit product <math>(rA) * SIMM</math> are placed into register rD. The low-order 32 bits of the product are the correct 32-bit product. The low-order bits are independent of whether the operands are treated as signed or unsigned integers. However, XER[OV] is set based on the result interpreted as a signed integer.</p> <p>The high-order bits are lost. This instruction can be used with <b>mulhwx</b> to calculate a full 64-bit product.</p>

## Freescale Semiconductor, Inc.

Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Multiply Low	<b>mullw</b> <b>mullw.</b> <b>mullwo</b> <b>mullwo.</b>	rD,rA,rB	<p>The low-order 32 bits of the 64-bit product (rA) * (rB) are placed into register rD. The low-order 32 bits of the product are the correct 32-bit product. The low-order bits are independent of whether the operands are treated as signed or unsigned integers. However, XER[OV] is set based on the result interpreted as a signed integer.</p> <p>The high-order bits are lost. This instruction can be used with <b>mulhwx</b> to calculate a full 64-bit product. Some implementations may execute faster if rB contains the operand having the smaller absolute value.</p> <p><b>mullw</b> Multiply Low  <b>mullw.</b> Multiply Low with CR Update. The dot suffix enables the update of the condition register.  <b>mullwo</b> Multiply Low with Overflow. The o suffix enables the overflow bit (OV) in the XER.  <b>mullwo.</b> Multiply Low with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>
Multiply High Word	<b>mulhw</b> <b>mulhw.</b>	rD,rA,rB	<p>The contents of rA and rB are interpreted as 32-bit signed integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into rD.</p> <p>Both operands and the product are interpreted as signed integers.</p> <p>This instruction may execute faster if rB contains the operand having the smaller absolute value.</p> <p><b>mulhw</b> Multiply High Word  <b>mulhw.</b> Multiply High Word with CR Update. The dot suffix enables the update of the condition register.</p>
Multiply High Word Unsigned	<b>mulhwu</b> <b>mulhwu.</b>	rD,rA,rB	<p>The contents of rA and of rB are extracted and interpreted as 32-bit unsigned integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into rD.</p> <p>Both operands and the product are interpreted as unsigned integers.</p> <p>This instruction may execute faster if rB contains the operand having the smaller absolute value.</p> <p><b>mulhwu</b> Multiply High Word Unsigned  <b>mulhwu.</b> Multiply High Word Unsigned with CR Update. The dot suffix enables the update of the condition register.</p>

## Freescale Semiconductor, Inc.

Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Divide Word	<b>divw</b> <b>divw.</b> <b>divwo</b> <b>divwo.</b>	<b>rD,rA,rB</b>	<p>The dividend is the signed value of (<b>rA</b>). The divisor is the signed value of (<b>rB</b>). The 64-bit quotient is formed. The low-order 32 bits of the 64-bit quotient are placed into <b>rD</b>. The remainder is not supplied as a result.</p> <p>Both operands are interpreted as signed integers. The quotient is the unique signed integer that satisfies the following:</p> <p>dividend = (quotient times divisor) + r  where <math>0 \leq r &lt;  \text{divisor} </math> if the dividend is non-negative, and  <math>- \text{divisor}  &lt; r \leq 0</math> if the dividend is negative.</p> <p>If an attempt is made to perform any of the divisions</p> <p>0x8000 0000 / -1</p> <p>or</p> <p>&lt;anything&gt; / 0</p> <p>the contents of register <b>rD</b> are undefined, as are the contents of the LT, GT, and EQ bits of the condition register field CR0 if the instruction has condition register updating enabled. In these cases, if instruction overflow is enabled, then XER[OV] is set.</p> <p>The 32-bit signed remainder of dividing (<b>rA</b>) by (<b>rB</b>) can be computed as follows, except in the case that (<b>rA</b>) = <math>-2^{31}</math> and (<b>rB</b>) = -1:</p> <p><b>divw</b> <b>rD,rA,rB</b>    <b>rD</b> = quotient  <b>mull</b> <b>rD,rD,rB</b>    <b>rD</b> = quotient*divisor  <b>subf</b> <b>rD,rD,rA</b>    <b>rD</b> = remainder</p> <p><b>divw</b>        Divide Word  <b>divw.</b>       Divide Word with CR Update. The dot suffix enables the update of the condition register.  <b>divwo</b>       Divide Word with Overflow. The o suffix enables the overflow bit (OV) in the XER.  <b>divwo.</b>      Divide Word with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>



Table 4-1 Integer Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Divide Word Unsigned	<b>divwu</b> <b>divwu.</b> <b>divwuo</b> <b>divwuo.</b>	<b>rD,rA,rB</b>	<p>The dividend is the value of (<b>rA</b>). The divisor is the value of (<b>rB</b>). The 32-bit quotient is placed into <b>rD</b>. The remainder is not supplied as a result.</p> <p>Both operands are interpreted as unsigned integers. The quotient is the unique unsigned integer that satisfies the following:</p> $\text{dividend} = (\text{quotient times divisor}) + r$ <p>where <math>0 \leq r &lt; \text{divisor}</math>.</p> <p>If an attempt is made to perform the division</p> $\text{<anything>} / 0$ <p>the contents of register <b>rD</b> are undefined, as are the contents of the LT, GT, and EQ bits of the condition register field CR0 if the instruction has the condition register updating enabled. In these cases, if instruction overflow is enabled, then XER[OV] is set.</p> <p>The 32-bit unsigned remainder of dividing (<b>rA</b>) by (<b>rB</b>) can be computed as follows:</p> <p><b>divwu</b> <b>rD,rA,rB</b>    <b>rD</b> = quotient  <b>mull</b> <b>rD,rD,rB</b>    <b>rD</b> = quotient*divisor  <b>subf</b> <b>rD,rD,rA</b>    <b>rD</b> = remainder</p> <p><b>divwu</b>    Divide Word Unsigned  <b>divwu.</b>    Divide Word Unsigned with CR Update. The dot suffix enables the update of the condition register.  <b>divwuo</b>    Divide Word Unsigned with Overflow. The o suffix enables the overflow bit (OV) in the XER.  <b>divwuo.</b>    Divide Word Unsigned with Overflow and CR Update. The o. suffix enables the update of the condition register and enables the overflow bit (OV) in the XER.</p>

See [E.2 Simplified Mnemonics for Subtract Instructions](#) for information on simplified mnemonics.

### 4.3.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **rA** with either the **UIMM** operand, the **SIMM** operand or the contents of register **rB**. Algebraic comparison compares two signed integers. Logical comparison compares two unsigned numbers. [Table 4-2](#) summarizes the RCPU integer compare instructions.

Table 4-2 Integer Compare Instructions

Name	Mnemonic	Operand Syntax	Operation
Compare Immediate	<b>cmpi</b>	crfD,L,rA,SIMM	The contents of register <b>rA</b> is compared with the sign-extended value of the SIMM operand, treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand <b>crfD</b> .
Compare	<b>cmp</b>	crfD,L,rA,rB	The contents of register <b>rA</b> is compared with register <b>rB</b> , treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand <b>crfD</b> .
Compare Logical Immediate	<b>cmpli</b>	crfD,L,rA,UIMM	The contents of register <b>rA</b> is compared with 0x0000    UIMM, treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand <b>crfD</b> .
Compare Logical	<b>cmpl</b>	crfD,L,rA,rB	The contents of register <b>rA</b> is compared with register <b>rB</b> , treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand <b>crfD</b> .

While the PowerPC architecture specifies that the value in the L field specifies whether the operands are treated as 32- or 64-bit values, the RCPU ignores the value in the L field and treats the operands as 32-bit values.

The **crfD** field can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in the instruction **crfD** field, using one of the CR field symbols (CR0 to CR7) or an explicit field number. Refer to [Table E-2](#) for the list of CR field symbols and to [E.3 Simplified Mnemonics for Compare Instructions](#) for simplified mnemonics.

#### 4.3.3 Integer Logical Instructions

The logical instructions shown in [Table 4-4](#) perform bit-parallel operations. Logical instructions with Rc = 1 and instructions **andi.** and **andis.** set condition register field CR0 to characterize the result of the logical operation. These fields are set as if the sign-extended low-order 32 bits of the result were algebraically compared to zero. The remaining logical instructions do not modify the condition register. Logical instructions do not change the SO, OV, or CA bits in the XER.

**Table 4-3 Integer Logical Instructions**

Name	Mnemonic	Operand Syntax	Operation
AND Immediate	<b>andi.</b>	<b>rA,rS,UIMM</b>	The contents of <b>rS</b> is ANDed with 0x0000    UIMM and the result is placed into <b>rA</b> .
AND Immediate Shifted	<b>andis.</b>	<b>rA,rS,UIMM</b>	The contents of <b>rS</b> is ANDed with UIMM    0x0000 and the result is placed into <b>rA</b> .
OR Immediate	<b>ori</b>	<b>rA,rS,UIMM</b>	The contents of <b>rS</b> is ORed with 0x0000    UIMM and the result is placed into <b>rA</b> . The preferred no-op is <b>ori 0,0,0</b>
OR Immediate Shifted	<b>oris</b>	<b>rA,rS,UIMM</b>	The contents of <b>rS</b> is ORed with UIMM    0x0000 and the result is placed into <b>rA</b> .
XOR Immediate	<b>xori</b>	<b>rA,rS,UIMM</b>	The contents of <b>rS</b> is XORed with 0x0000    UIMM and the result is placed into <b>rA</b> .
XOR Immediate Shifted	<b>xoris</b>	<b>rA,rS,UIMM</b>	The contents of <b>rS</b> is XORed with UIMM    0x0000 and the result is placed into <b>rA</b> .
AND	<b>and</b> <b>and.</b>	<b>rA,rS,rB</b>	The contents of <b>rS</b> is ANDed with the contents of register <b>rB</b> and the result is placed into <b>rA</b> .  <b>and</b> AND <b>and.</b> AND with CR Update. The dot suffix enables the update of the condition register.
OR	<b>or</b> <b>or.</b>	<b>rA,rS,rB</b>	The contents of <b>rS</b> is ORed with the contents of <b>rB</b> and the result is placed into <b>rA</b> .  <b>or</b> OR <b>or.</b> OR with CR Update. The dot suffix enables the update of the condition register.
XOR	<b>xor</b> <b>xor.</b>	<b>rA,rS,rB</b>	The contents of <b>rS</b> is XORed with the contents of <b>rB</b> and the result is placed into register <b>rA</b> .  <b>xor</b> XOR <b>xor.</b> XOR with CR Update. The dot suffix enables the update of the condition register.
NAND	<b>nand</b> <b>nand.</b>	<b>rA,rS,rB</b>	The contents of <b>rS</b> is ANDed with the contents of <b>rB</b> and the one's complement of the result is placed into register <b>rA</b> .  <b>nand</b> NAND <b>nand.</b> NAND with CR Update. The dot suffix enables the update of the condition register. NAND with <b>rS</b> = <b>rB</b> can be used to obtain the one's complement.
NOR	<b>nor</b> <b>nor.</b>	<b>rA,rS,rB</b>	The contents of <b>rS</b> is ORed with the contents of <b>rB</b> and the one's complement of the result is placed into register <b>rA</b> .  <b>nor</b> NOR <b>nor.</b> NOR with CR Update. The dot suffix enables the update of the condition register. NOR with <b>rS</b> = <b>rB</b> can be used to obtain the one's complement.

**Table 4-3 Integer Logical Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Equivalent	<b>eqv</b> <b>eqv.</b>	rA,rS,rB	The contents of rS is XORed with the contents of rB and the complemented result is placed into register rA.  <b>eqv</b> Equivalent <b>eqv.</b> Equivalent with CR Update. The dot suffix enables the update of the condition register.
AND with Complement	<b>andc</b> <b>andc.</b>	rA,rS,rB	The contents of rS is ANDed with the complement of the contents of rB and the result is placed into rA.  <b>andc</b> AND with Complement <b>andc.</b> AND with Complement with CR Update. The dot suffix enables the update of the condition register.
OR with Complement	<b>orc</b> <b>orc.</b>	rA,rS,rB	The contents of rS is ORed with the complement of the contents of rB and the result is placed into rA.  <b>orc</b> OR with Complement <b>orc.</b> OR with Complement with CR Update. The dot suffix enables the update of the condition register.
Extend Sign Byte	<b>extsb</b> <b>extsb.</b>	rA,rS	The contents of rS[24:31] are placed into rA[24:31]. Bit 24 of rS is placed into rA[0:23].  <b>extsb</b> Extend Sign Byte <b>extsb.</b> Extend Sign Byte with CR Update. The dot suffix enables the update of the condition register.
Extend Sign Half Word	<b>extsh</b> <b>extsh.</b>	rA,rS	The contents of rS[16:31] are placed into rA[16:31]. Bit 16 of rS is placed into rA[0:15].  <b>extsh</b> Extend Sign Half Word <b>extsh.</b> Extend Sign Half Word with CR Update. The dot suffix enables the update of the condition register.
Count Leading Zeros Word	<b>cntlzw</b> <b>cntlzw.</b>	rA,rS	A count of the number of consecutive zero bits of rS is placed into rA. This number ranges from 0 to 32, inclusive.  <b>cntlzw</b> Count Leading Zeros Word <b>cntlzw.</b> Count Leading Zeros Word with CR Update. The dot suffix enables the update of the condition register.  When the Count Leading Zeros Word instruction has condition register updating enabled, the LT field is cleared to zero in CR0.

#### 4.3.4 Integer Rotate and Shift Instructions

Rotate and shift instructions provide powerful and general ways to manipulate register contents. [Table 4-4](#) shows the types of rotate and shift operations provided by the RCPu.

Table 4-4 Rotate and Shift Operations

Operation	Description
Extract	Select a field of $n$ bits starting at bit position $b$ in the source register, right or left justify this field in the target register, and clear all other bits of the target register to zero.
Insert	Select a left- or right-justified field of $n$ bits in the source register, insert this field starting at bit position $b$ of the target register, and leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a left-justified field when operating on double-words; such an insertion requires more than one instruction.)
Rotate	Rotate the contents of a register right or left $n$ bits without masking.
Shift	Shift the contents of a register right or left $n$ bits, clearing vacated bits to zero (logical shift).
Clear	Clear the leftmost or rightmost $n$ bits of a register to zero.
Clear left and shift left	Clear the leftmost $b$ bits of a register, then shift the register left by $n$ bits. This operation can be used to scale a known non-negative array index by the width of an element.

The IU performs rotation operations on data from a GPR and returns the result, or a portion of the result, to a GPR. Rotation operations rotate a 32-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 31. A rotate right operation can be accomplished by specifying a rotation of  $32-n$  bits, where  $n$  is the right rotation amount.

Rotate and shift instructions use a mask generator. The mask is 32 bits long and consists of 1-bits from a start bit, **MB**, through and including a stop bit, **ME**, and 0-bits elsewhere. The values of **MB** and **ME** range from zero to 31. If **MB** > **ME**, the 1-bits wrap around from position 31 to position 0. Thus the mask is formed as follows:

if **MB**  $\leq$  **ME** then

mask[mstart:mstop] = ones  
mask[all other bits] = zeros

else

mask[mstart:31] = ones  
mask[0:mstop] = ones  
mask[all other bits] = zeros

It is not possible to specify an all-zero mask. The use of the mask is described in the following sections.

If condition register updating is enabled, rotate and shift instructions set condition register field CR0 according to the contents of **rA** at the completion of the instruction. Rotate and shift instructions do not change the values of XER[OV] or XER[SO] bits. Rotate and shift instructions, except algebraic right shifts, do not change the XER[CA] bit.

Simplified mnemonics allow simpler coding of often-used functions such as clear-

ing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts. Some of these are shown as examples with the rotate instructions. In addition, [E.4 Simplified Mnemonics for Rotate and Shift Instructions](#) provides a list of these mnemonics.

#### 4.3.4.1 Integer Rotate Instructions

Integer rotate instructions rotate the contents of a register. The result of the rotation is inserted into the target register under control of a mask (if a mask bit is one the associated bit of the rotated data is placed into the target register, and if the mask bit is zero the associated bit in the target register is unchanged), or ANDed with a mask before being placed into the target register.

Rotate left instructions allow right-rotation of the contents of a register to be performed by a left-rotation of  $32 - n$ , where  $n$  is the number of bits by which to rotate right.

The integer rotate instructions are summarized in [Table 4-5](#).

**Table 4-5 Integer Rotate Instructions**

Name	Mnemonic	Operand Syntax	Operation
Rotate Left Word Immediate then AND with Mask	<b>rlwinm</b> <b>rlwinm.</b>	$rA, rS, SH, MB, ME$	<p>The contents of register <math>rS</math> are rotated left by the number of bits specified by operand <math>SH</math>. A mask is generated having 1-bits from the bit specified by operand <math>MB</math> through the bit specified by operand <math>ME</math> and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register <math>rA</math>.</p> <p><b>rlwinm</b> Rotate Left Word Immediate then AND with Mask  <b>rlwinm.</b> Rotate Left Word Immediate then AND with Mask with CR Update. The dot suffix enables the update of the condition register.</p> <p>Simplified mnemonics:  <b>extlwi</b> <math>rA, rS, n, b, r, lwinm</math> <math>rA, rS, b, 0, n-1</math>  <b>srwi</b> <math>rA, rS, n, r, lwinm</math> <math>rA, rS, 32-n, n, 31</math>  <b>clrrwi</b> <math>rA, rS, n, r, lwinm</math> <math>rA, rS, 0, 0, 31-n</math></p> <p>Note: The <b>rlwinm</b> instruction can be used for extracting, clearing and shifting bit fields using the methods shown below:</p> <p>To extract an <math>n</math>-bit field that starts at bit position <math>b</math> in register <math>rS</math>, right-justified into <math>rA</math> (clearing the remaining <math>32 - n</math> bits of <math>rA</math>), set <math>SH = b + n</math>, <math>MB = 32 - n</math>, and <math>ME = 31</math>.</p> <p>To extract an <math>n</math>-bit field that starts at bit position <math>b</math> in <math>rS</math>, left-justified into <math>rA</math>, set <math>SH = b</math>, <math>MB = 0</math>, and <math>ME = n - 1</math>.</p> <p>To rotate the contents of a register left (right) by <math>n</math> bits, set <math>SH = (32 - n)</math>, <math>MB = 0</math>, and <math>ME = 31</math>.</p> <p>To shift the contents of a register right by <math>n</math> bits, set <math>SH = 32 - n</math>, <math>MB = n</math>, and <math>ME = 31</math>.</p> <p>To clear the high-order <math>b</math> bits of a register and then shift the result left by <math>n</math> bits, set <math>SH = n</math>, <math>MB = b - n</math> and <math>ME = 31 - n</math>.</p> <p>To clear the low-order <math>n</math> bits of a register, set <math>SH = 0</math>, <math>MB = 0</math>, and <math>ME = 31 - n</math>.</p>

**Table 4-5 Integer Rotate Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Rotate Left Word then AND with Mask	<b>rlwnm</b> <b>rlwnm.</b>	<b>rA,rS,rB,MB,ME</b>	<p>The contents of <b>rS</b> are rotated left by the number of bits specified by <b>rB[27:31]</b>. A mask is generated having 1-bits from the bit specified by operand <b>MB</b> through the bit specified by operand <b>ME</b> and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into <b>rA</b>.</p> <p><b>rlwnm</b> Rotate Left Word then AND with Mask  <b>rlwnm.</b> Rotate Left Word then AND with Mask with CR Update. The dot suffix enables the update of the condition register.</p> <p>Simplified mnemonics:  <b>rotlw rA,rS,rBrlwnm rA,rS,rB,0,31</b></p> <p>Note: The <b>rlwnm</b> instruction can be used to extract and rotate bit fields using the methods shown below:</p> <p>To extract an <math>n</math>-bit field that starts at the variable bit position <math>b</math> in the register specified by operand <b>rS</b>, right-justified into <b>rA</b> (clearing the remaining <math>32-n</math> bits of <b>rA</b>), set <b>rB[27:31] = b + n</b>, <b>MB = 32 - n</b>, and <b>ME = 31</b>.</p> <p>To extract an <math>n</math>-bit field that starts at variable bit position <math>b</math> in the register specified by operand <b>rS</b>, left-justified into <b>rA</b> (clearing the remaining <math>32 - n</math> bits of <b>rA</b>), set <b>rB[27:31] = b</b>, <b>MB = 0</b>, and <b>ME = n - 1</b>.</p> <p>To rotate the contents of the low-order 32 bits of a register left (right) by variable <math>n</math> bits, set <b>rB[27:31] = n (32 - n)</b>, <b>MB = 0</b>, and <b>ME = 31</b>.</p>
Rotate Left Word Immediate then Mask Insert	<b>rlwimi</b> <b>rlwimi.</b>	<b>rA,rS,SH,MB,ME</b>	<p>The contents of <b>rS</b> are rotated left by the number of bits specified by operand <b>SH</b>. A mask is generated having 1-bits from the bit specified by <b>MB</b> through the bit specified by <b>ME</b> and 0-bits elsewhere. The rotated data is inserted into <b>rA</b> under control of the generated mask.</p> <p><b>rlwimi</b> Rotate Left Word Immediate then Mask  <b>rlwimi.</b> Rotate Left Word Immediate then Mask Insert with CR Update. The dot suffix enables the update of the condition register.</p> <p>Simplified mnemonic:  <b>inslw rA,rS,n,brlwim rA,rS,32-b,b,b+n-1</b></p> <p>Note: The opcode <b>rlwimi</b> can be used to insert a bit field into the contents of register specified by operand <b>rA</b> using the methods shown below:</p> <p>To insert an <math>n</math>-bit field that is left-justified in <b>rS</b> into <b>rA</b> starting at bit position <math>b</math>, set <b>SH = 32 - b</b>, <b>MB = b</b>, and <b>ME = (b + n) - 1</b>.</p> <p>To insert an <math>n</math>-bit field that is right-justified in <b>rS</b> into <b>rA</b> starting at bit position <math>b</math>, set <b>SH = 32 - (b + n)</b>, <b>MB = b</b>, and <b>ME = (b + n) - 1</b>.</p> <p>Simplified mnemonics are provided for both of these methods.</p>

#### 4.3.4.2 Integer Shift Instructions

The instructions in this section perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics are provided to make coding of



such shifts simpler and easier to understand.

Any shift right algebraic instruction, followed by **addze**, can be used to divide quickly by  $2^n$ .

Multiple-precision shifts can be programmed as shown in **APPENDIX B MULTIPLE-PRECISION SHIFTS**.

The integer shift instructions are summarized in **Table 4-6**.

**Table 4-6 Integer Shift Instructions**

Name	Mnemonic	Operand Syntax	Operation
Shift Left Word	<b>slw</b> <b>slw.</b>	<b>rA,rS,rB</b>	<p>The contents of <b>rS</b> are shifted left the number of bits specified by <b>rB[26:31]</b>. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into <b>rA</b>.</p> <p>If <b>rB[26] = 1</b>, then <b>rA</b> is filled with zeros.</p> <p><b>slw</b>            Shift Left Word  <b>slw.</b>          Shift Left Word with CR Update. The dot suffix enables the update of the condition register.</p>
Shift Right Word	<b>srw</b> <b>srw.</b>	<b>rA,rS,rB</b>	<p>The contents of <b>rS</b> are shifted right the number of bits specified by <b>rB[26:31]</b>. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into <b>rA</b>.</p> <p>If <b>rB[26]=1</b>, then <b>rA</b> is filled with zeros.</p> <p><b>srw</b>            Shift Right Word  <b>srw.</b>          Shift Right Word with CR Update. The dot suffix enables the update of the condition register.</p>
Shift Right Algebraic Word Immediate	<b>srawi</b> <b>srawi.</b>	<b>rA,rS,SH</b>	<p>The contents of <b>rS</b> are shifted right the number of bits specified by operand <b>SH</b>. Bits shifted out of position 31 are lost. The 32-bit result is sign extended and placed into <b>rA</b>. <b>XER[CA]</b> is set if <b>rS</b> contains a negative number and any 1-bits are shifted out of position 31; otherwise <b>XER(CA)</b> is cleared. An operand <b>SH</b> of zero causes <b>rA</b> to be loaded with the contents of <b>rS</b> and <b>XER[CA]</b> to be cleared to zero.</p> <p><b>srawi</b>          Shift Right Algebraic Word Immediate  <b>srawi.</b>        Shift Right Algebraic Word Immediate with CR Update. The dot suffix enables the update of the condition register.</p>
Shift Right Algebraic Word	<b>sraw</b> <b>sraw.</b>	<b>rA,rS,rB</b>	<p>The contents of <b>rS</b> are shifted right the number of bits specified by <b>rB[26:31]</b>. The 32-bit result is placed into <b>rA</b>. <b>XER[CA]</b> is set to one if <b>rS</b> contains a negative number and any 1-bits are shifted out of position 31; otherwise <b>XER[CA]</b> is cleared to zero. An operand (<b>rB</b>) of zero causes <b>rA</b> to be loaded with the contents of <b>rS</b>, and <b>XER[CA]</b> to be cleared to zero. If <b>rB[26] = 1</b>, then <b>rA</b> is filled with 32 sign bits (bit 0) from <b>rS</b>. If <b>rB[26] = 0</b>, then <b>rA</b> is filled from the left with sign bits. Condition register field <b>CR0</b> is set based on the value written into <b>rA</b>.</p> <p><b>sraw</b>            Shift Right Algebraic Word  <b>sraw.</b>          Shift Right Algebraic Word with CR Update. The dot suffix enables the update of the condition register.</p>



## 4.4 Floating-Point Instructions

This section describes the floating-point instructions, which include the following:

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions

Floating-point loads and stores are discussed in [4.5 Load and Store Instructions](#).

### 4.4.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in [Table 4-7](#).

**Table 4-7 Floating-Point Arithmetic Instructions**

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Add	<b>fadd</b> <b>fadd.</b>	<b>frD,frA,frB</b>	<p>The floating-point operand in register <b>frA</b> is added to the floating-point operand in register <b>frB</b>. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added algebraically to form an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.</p> <p>If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fadd</b> Floating-Point Add  <b>fadd.</b> Floating-Point Add with CR Update. The dot suffix enables the update of the condition register.</p>

**Table 4-7 Floating-Point Arithmetic Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Add Single-Precision	<b>fadds</b> <b>fadds.</b>	<b>frD,frA,frB</b>	<p>The floating-point operand in register <b>frA</b> is added to the floating-point operand in register <b>frB</b>. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added algebraically to form an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.</p> <p>If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fadds</b> Floating-Point Single-Precision  <b>fadds.</b> Floating-Point Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Subtract	<b>fsub</b> <b>fsub.</b>	<b>frD,frA,frB</b>	<p>The floating-point operand in register <b>frB</b> is subtracted from the floating-point operand in register <b>frA</b>. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>The execution of the Floating-Point Subtract instruction is identical to that of Floating-Point Add, except that the contents of register <b>frB</b> participates in the operation with its sign bit (bit 0) inverted.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fsub</b> Floating-Point Subtract  <b>fsub.</b> Floating-Point Subtract with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Subtract Single-Precision	<b>fsubs</b> <b>fsubs.</b>	<b>frD,frA,frB</b>	<p>The floating-point operand in register <b>frB</b> is subtracted from the floating-point operand in register <b>frA</b>. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>The execution of the Floating-Point Subtract instruction is identical to that of Floating-Point Add, except that the contents of register <b>frB</b> participates in the operation with its sign bit (bit 0) inverted.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fsubs</b> Floating-Point Subtract Single-Precision  <b>fsubs.</b> Floating-Point Subtract Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>

**Table 4-7 Floating-Point Arithmetic Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Multiply	<b>fmul</b> <b>fmul.</b>	<b>frD,frA,frC</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>.</p> <p>If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>Floating-point multiplication is based on exponent addition and multiplication of the significands.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fmul</b> Floating-Point Multiply <b>fmul.</b> Floating-Point Multiply with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Multiply Single-Precision	<b>fmuls</b> <b>fmuls.</b>	<b>frD,frA,frC</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>Floating-point multiplication is based on exponent addition and multiplication of the significands.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fmuls</b> Floating-Point Multiply Single-Precision <b>fmuls.</b> Floating-Point Multiply Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Divide	<b>fdiv</b> <b>fdiv.</b>	<b>frD,frA,frB</b>	<p>The floating-point operand in register <b>frA</b> is divided by the floating-point operand in register <b>frB</b>. No remainder is preserved.</p> <p>If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>Floating-point division is based on exponent subtraction and division of the significands.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE]=1.</p> <p><b>fdiv</b> Floating-Point Divide <b>fdiv.</b> Floating-Point Divide with CR Update. The dot suffix enables the update of the condition register.</p>

Table 4-7 Floating-Point Arithmetic Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Divide Single-Precision	<b>fdivs</b> <b>fdivs.</b>	<b>frD,frA,frB</b>	<p>The floating-point operand in register <b>frA</b> is divided by the floating-point operand in register <b>frB</b>. No remainder is preserved.</p> <p>If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>Floating-point division is based on exponent subtraction and division of the significands.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.</p> <p><b>fdivs</b> Floating-Point Divide Single-Precision  <b>fdivs.</b> Floating-Point Divide Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>

#### 4.4.2 Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The fractional part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

The floating-point multiply-add instructions are summarized in [Table 4-8](#).

Table 4-8 Floating-Point Multiply-Add Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Multiply-Add	<b>fmadd</b> <b>fmadd.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is added to this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fmadd</b> Floating-Point Multiply-Add  <b>fmadd.</b> Floating-Point Multiply-Add with CR Update. The dot suffix enables the update of the condition register.</p>

## Table 4-8 Floating-Point Multiply-Add Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Multiply-Add Single-Precision	<b>fmadds</b> <b>fmadds.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is added to this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fmadds</b> Floating-Point Multiply-Add Single-Precision  <b>fmadds.</b> Floating-Point Multiply-Add Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Multiply-Subtract	<b>fmsub</b> <b>fmsub.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is subtracted from this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fmsub</b> Floating-Point Multiply-Subtract  <b>fmsub.</b> Floating-Point Multiply-Subtract with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Multiply-Subtract Single-Precision	<b>fmsubs</b> <b>fmsubs.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is subtracted from this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register <b>frD</b>.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fmsubs</b> Floating-Point Multiply-Subtract Single-Precision  <b>fmsubs.</b> Floating-Point Multiply-Subtract Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>

**Table 4-8 Floating-Point Multiply-Add Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Negative Multiply-Add	<b>fnmadd</b> <b>fnmadd.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is added to this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register <b>frD</b>.</p> <p>This instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions:</p> <ul style="list-style-type: none"> <li>• QNaNs propagate with no effect on their sign bit.</li> <li>• QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.</li> <li>• SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.</li> </ul> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fnmadd</b> Floating-Point Negative Multiply-Add  <b>fnmadd.</b> Floating-Point Negative Multiply-Add with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Negative Multiply-Add Single-Precision	<b>fnmadds</b> <b>fnmadds.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is added to this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register <b>frD</b>.</p> <p>This instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions:</p> <ul style="list-style-type: none"> <li>• QNaNs propagate with no effect on their sign bit.</li> <li>• QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.</li> <li>• SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.</li> </ul> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fnmadds</b> Floating-Point Negative Multiply-Add Single-Precision  <b>fnmadds.</b> Floating-Point Negative Multiply-Add Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>

Table 4-8 Floating-Point Multiply-Add Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Negative Multiply-Subtract	<b>fnmsub</b> <b>fnmsub.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is subtracted from this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register <b>frD</b>.</p> <p>This instruction produces the same result as would be obtained by using the floating-point multiply-subtract instruction and then negating the result, with the following exceptions:</p> <ul style="list-style-type: none"> <li>• QNaNs propagate with no effect on their sign bit.</li> <li>• QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.</li> <li>• SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.</li> </ul> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fnmsub</b> Floating-Point Negative Multiply-Subtract  <b>fnmsub.</b> Floating-Point Negative Multiply-Subtract with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Negative Multiply-Subtract Single-Precision	<b>fnmsubs</b> <b>fnmsubs.</b>	<b>frD,frA,frC,frB</b>	<p>The floating-point operand in register <b>frA</b> is multiplied by the floating-point operand in register <b>frC</b>. The floating-point operand in register <b>frB</b> is subtracted from this intermediate result.</p> <p>If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into register <b>frD</b>.</p> <p>This instruction produces the same result as would be obtained by using the floating-point multiply-subtract instruction and then negating the result, with the following exceptions:</p> <ul style="list-style-type: none"> <li>• QNaNs propagate with no effect on their sign bit.</li> <li>• QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.</li> <li>• SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.</li> </ul> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>fnmsubs</b> Floating-Point Negative Multiply-Subtract Single-Precision  <b>fnmsubs.</b> Floating-Point Negative Multiply-Subtract Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>

#### 4.4.3 Floating-Point Rounding and Conversion Instructions

The floating-point rounding instruction is used to produce a 32-bit single-precision number from a 64-bit double-precision floating-point number. The floating-point convert instructions convert 64-bit double-precision floating point numbers to 32-bit signed integer numbers.



Examples of uses of these instructions to perform various conversions can be found in [APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS](#).

**Table 4-9 Floating-Point Rounding and Conversion Instructions**

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Round to Single-Precision	<b>frsp</b> <b>frsp.</b>	<b>frD,frB</b>	<p>If it is already in single-precision range, the floating-point operand in register <b>frB</b> is placed into register <b>frD</b>. Otherwise the floating-point operand in register <b>frB</b> is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into register <b>frD</b>.</p> <p>The rounding is described fully in <a href="#">APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS</a>.</p> <p>FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.</p> <p><b>frsp</b> Floating-Point Round to Single-Precision  <b>frsp.</b> Floating-Point Round to Single-Precision with CR Update. The dot suffix enables the update of the condition register.</p>
Floating-Point Convert to Integer Word	<b>fctiw</b> <b>fctiw.</b>	<b>frD,frB</b>	<p>The floating-point operand in register <b>frB</b> is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in <b>frD</b>[32:63]. <b>frD</b>[0:31] are undefined.</p> <p>If the operand in register <b>frB</b> is greater than <math>2^{31}-1</math>, <b>frD</b>[32:63] are set to 0x7FFF FFFF.</p> <p>If the operand in register <b>frB</b> is less than <math>-2^{31}</math>, <b>frD</b>[32:63] are set to 0x8000 0000.</p> <p>The conversion is described fully in <a href="#">APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS</a>.</p> <p>Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.</p> <p><b>fctiw</b> Floating-Point Convert to Integer Word  <b>fctiw.</b> Floating-Point Convert to Integer Word with CR Update. The dot suffix enables the update of the condition register.</p>



Table 4-9 Floating-Point Rounding and Conversion Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Convert to Integer Word with Round	<b>fctiwz</b> <b>fctiwz.</b>	<b>frD,frB</b>	<p>The floating-point operand in register <b>frB</b> is converted to a 32-bit signed integer, using the rounding mode Round toward Zero, and placed in <b>frD[32:63]</b>. <b>frD[0:31]</b> are undefined.</p> <p>If the operand in <b>frB</b> is greater than <math>2^{31} - 1</math>, <b>frD[32:63]</b> are set to 0x7FFF FFFF.</p> <p>If the operand in register <b>frB</b> is less than <math>-2^{31}</math>, <b>frD[32:63]</b> are set to 0x8000 0000.</p> <p>The conversion is described fully in <a href="#">APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS</a>.</p> <p>Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.</p> <p><b>fctiwz</b> Floating-Point Convert to Integer Word with Round Toward Zero</p> <p><b>fctiwz.</b> Floating-Point Convert to Integer Word with Round Toward Zero with CR Update. The dot suffix enables the update of the condition register.</p>

#### 4.4.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two floating-point registers and the comparison ignores the sign of zero (that is  $+0 = -0$ ). The comparison can be ordered or unordered. The comparison sets one bit in the designated CR field and clears the other three bits. The FPCC bits (FPSCR[16:19]) are set in the same way.

The CR field and the FPCC are interpreted as shown in [Table 4-10](#).

Table 4-10 CR Bit Settings

Bit	Name	Description
0	FL	( <b>frA</b> ) < ( <b>frB</b> )
1	FG	( <b>frA</b> ) > ( <b>frB</b> )
2	FE	( <b>frA</b> ) = ( <b>frB</b> )
3	FU	( <b>frA</b> ) ? ( <b>frB</b> ) (unordered)

On floating-point compare unordered (**fcmpu**) and floating-point compare ordered (**fcmpo**) instructions with condition register updating enabled, the PowerPC architecture defines CR1 and the CR field specified by operand **crfD** as undefined.

The floating-point compare instructions are summarized in [Table 4-11](#).

Table 4-11 Floating-Point Compare Instructions

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Compare Unordered	<b>fcmpu</b>	<b>crfD,frA,frB</b>	<p>The floating-point operand in register <b>frA</b> is compared to the floating-point operand in register <b>frB</b>. The result of the compare is placed into CR field <b>crfD</b> and the FPCC.</p> <p>If an operand is a NaN, either quiet or signaling, CR field <b>crfD</b> and the FPCC are set to reflect unordered. If an operand is a Signaling NaN, VXSNAN is set.</p>
Floating-Point Compare Ordered	<b>fcmpo</b>	<b>crfD,frA,frB</b>	<p>The floating-point operand in register <b>frA</b> is compared to the floating-point operand in register <b>frB</b>. The result of the compare is placed into CR field <b>crfD</b> and the FPCC.</p> <p>If an operand is a NaN, either quiet or signaling, CR field <b>crfD</b> and the FPCC are set to reflect unordered. If an operand is a Signaling NaN, VXSNAN is set, and if invalid operation is disabled (VE = 0) then VXVC is set. Otherwise, if an operand is a Quiet NaN, VXVC is set.</p>

#### 4.4.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed. In particular:

- All exceptions caused by the previously initiated instructions are recorded in the FPSCR before the FPSCR instruction is initiated.
- All invocations of the floating-point exception handler caused by the previously initiated instructions have occurred before the FPSCR instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits appears to be initiated until the FPSCR instruction has completed.

Floating-point memory access instructions are not affected.

The floating-point status and control register instructions are summarized in [Table 4-12](#).

**Table 4-12 Floating-Point Status and Control Register Instructions**

Name	Mnemonic	Operand Syntax	Operation
Move from FPSCR	<b>mffs</b> <b>mffs.</b>	<b>frD</b>	The contents of the FPSCR are placed into <b>frD</b> [32:63].  <b>mffs</b> Move from FPSCR <b>mffs.</b> Move from FPSCR with CR Update. The dot suffix enables the update of the condition register.
Move to Condition Register from FPSCR	<b>mcrfs</b>	<b>crfD,crfS</b>	The contents of FPSCR field specified by operand <b>crfS</b> are copied to the CR field specified by operand <b>crfD</b> . All exception bits copied are cleared to zero in the FPSCR.
Move to FPSCR Field Immediate	<b>mtfsfi</b> <b>mtfsfi.</b>	<b>crfD,IMM</b>	The value of the IMM field is placed into FPSCR field <b>crfD</b> . All other FPSCR fields are unchanged.  <b>mtfsfi</b> Move to FPSCR Field Immediate <b>mtfsfi.</b> Move to FPSCR Field Immediate with CR Update. The dot suffix enables the update of the condition register.  When FPSCR[0:3] is specified, bits 0 (FX) and 3 (OX) are set to the values of IMM[0] and IMM[3] (i.e., even if this instruction causes OX to change from zero to one, FX is set from IMM[0] and not by the usual rule that FX is set to one when an exception bit changes from zero to one). Bits 1 and 2 (FEX and VX) are set according to the usual rule described in <a href="#">2.2.3 Floating-Point Status and Control Register (FPSCR)</a> , and not from IMM[1:2].
Move to FPSCR Fields	<b>mtfsf</b> <b>mtfsf.</b>	<b>FM,frB</b>	<b>frB</b> [32:63] are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let <i>i</i> be an integer in the range 0-7. If FM = 1 then FPSCR field <i>i</i> (FPSCR bits 4* <i>i</i> through 4* <i>i</i> +3) is set to the contents of the corresponding field of the low-order 32 bits of register <b>frB</b> .  <b>mtfsf</b> Move to FPSCR Fields <b>mtfsf.</b> Move to FPSCR Fields with CR Update. The dot suffix enables the update of the condition register.  When FPSCR[0:3] is specified, bits 0 (FX) and 3 (OX) are set to the values of <b>frB</b> [32] and <b>frB</b> [35] (i.e., even if this instruction causes OX to change from zero to one, FX is set from <b>frB</b> [32] and not by the usual rule that FX is set to one when an exception bit changes from zero to one). Bits 1 and 2 (FEX and VX) are set according to the usual rule described in <a href="#">2.2.3 Floating-Point Status and Control Register (FPSCR)</a> , and not from <b>frB</b> [33:34].
Move to FPSCR Bit 0	<b>mtfsb0</b> <b>mtfsb0.</b>	<b>crbD</b>	The bit of the FPSCR specified by operand <b>crbD</b> is cleared to zero. Bits 1 and 2 (FEX and VX) cannot be explicitly reset.  <b>mtfsb0</b> Move to FPSCR Bit 0 <b>mtfsb0.</b> Move to FPSCR Bit 0 with CR Update. The dot suffix enables the update of the condition register.
Move to FPSCR Bit 1	<b>mtfsb1</b> <b>mtfsb1.</b>	<b>crbD</b>	The bit of the FPSCR specified by operand <b>crbD</b> is set to one. Bits 1 and 2 (FEX and VX) cannot be reset explicitly.  <b>mtfsb1</b> Move to FPSCR Bit 1 <b>mtfsb1.</b> Move to FPSCR Bit 1 with CR Update. The dot suffix enables the update of the condition register.

## 4.5 Load and Store Instructions

The RCPU supports the following types of load and store instructions:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte reversal instructions
- Integer load and store multiple instructions
- Floating-point load instructions
- Floating-point store instructions
- Floating-point move instructions
- Memory synchronization instructions (described in [4.8 Memory Synchronization Instructions](#))

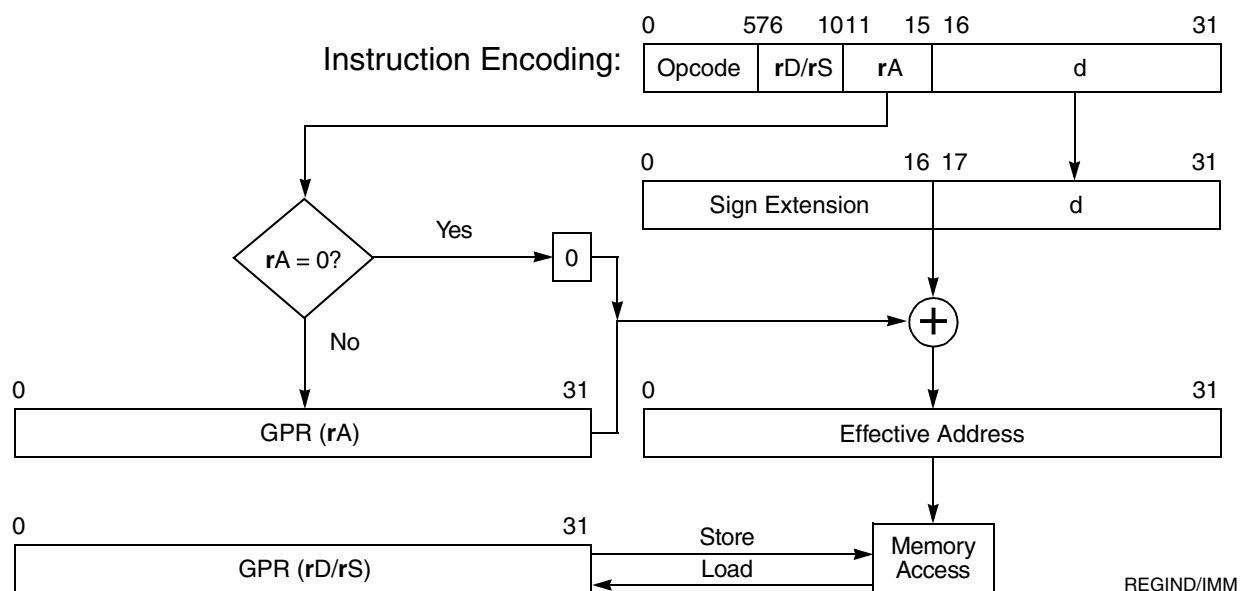
### 4.5.1 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode or register indirect mode.

#### 4.5.1.1 Register Indirect with Immediate Index Addressing

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended to 32 bits, and added to the contents of a general purpose register specified in the instruction (rA operand) to generate the effective address. A zero in place of the rA operand causes a zero to be added to the immediate index (d operand). The option to specify rA or zero is shown in the instruction descriptions as (rA|0).

**Figure 4-1** shows how an effective address is generated when using register indirect with immediate index addressing.

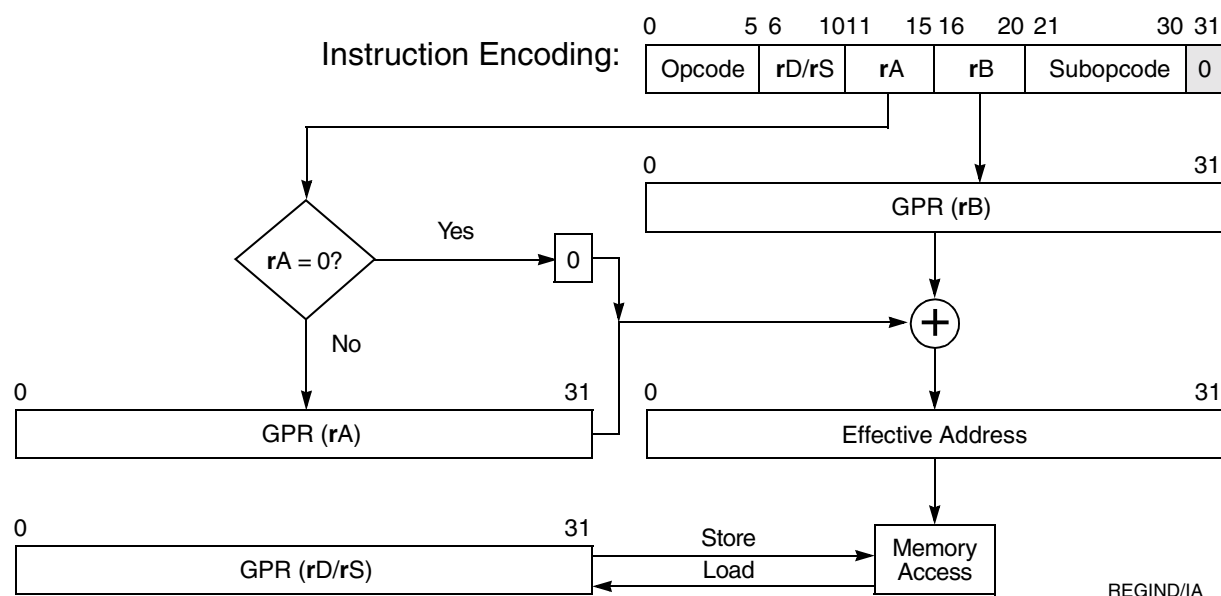


**Figure 4-1 Register Indirect with Immediate Index Addressing**

#### 4.5.1.2 Register Indirect with Index Addressing

Instructions using this addressing mode cause the contents of two general purpose registers (specified as operands **rA** and **rB**) to be added in the generation of the effective address. A zero in place of the **rA** operand causes a zero to be added to the contents of the general-purpose register specified in operand **rB**. The option to specify **rA** or zero is shown in the instruction descriptions as (**rA**|0).

**Figure 4-2** shows how an effective address is generated when using register indirect with index addressing.

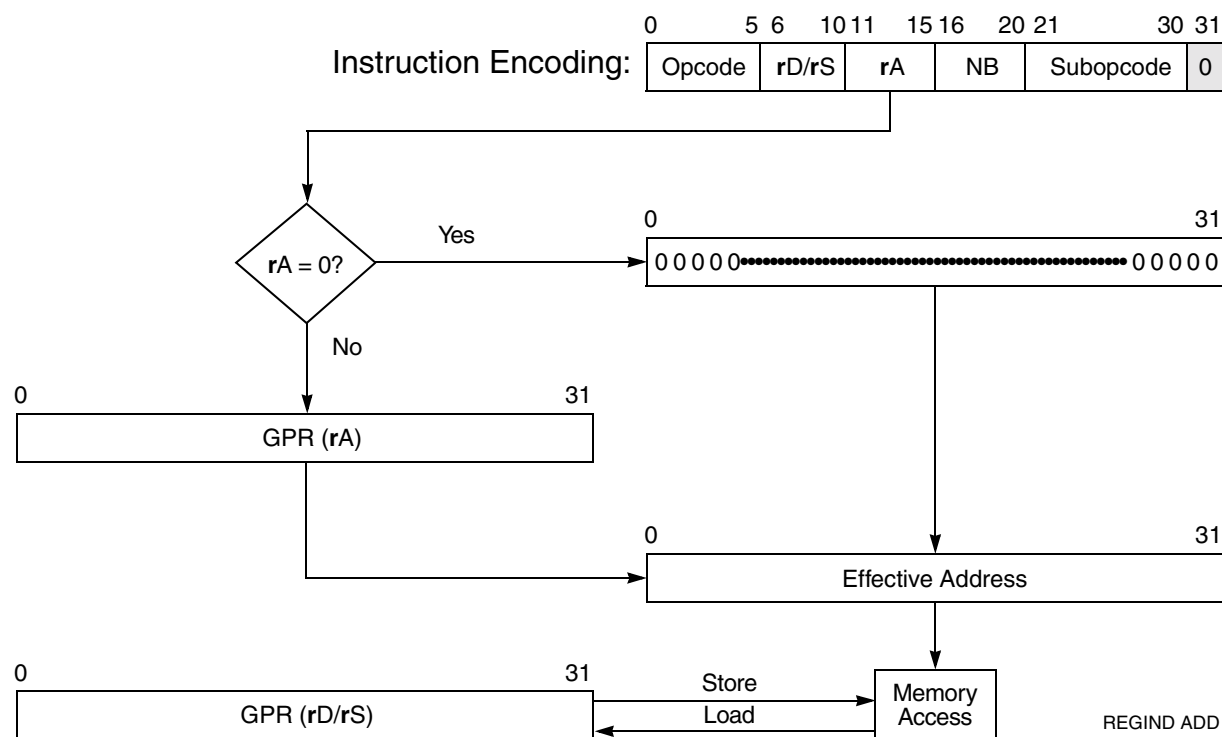


**Figure 4-2 Register Indirect with Index Addressing**

#### 4.5.1.3 Register Indirect Addressing

Instructions using this addressing mode use the contents of the general purpose register specified by the **rA** operand as the effective address. A zero in the **rA** operand causes an effective address of zero to be generated. The option to specify **rA** or zero is shown in the instruction descriptions as **(rA|0)**.

**Figure 4-3** shows how an effective address is generated when using register indirect addressing.



**Figure 4-3 Register Indirect Addressing**

### 4.5.2 Integer Load Instructions

For load instructions, the byte, half-word, word, or double-word addressed by EA is loaded into rD. Many integer load instructions have an update form, in which rA is updated with the generated effective address. For these forms, if  $rA \neq 0$  and  $rA \neq rD$ , the effective address is placed into rA and the memory element (byte, half-word, or word) addressed by EA is loaded into rD.

The PowerPC architecture defines load with update instructions with  $rA = 0$  or  $rA = rD$  as an invalid form. In the RCPU, however, if  $rA = 0$  then the EA is written into R0. If  $rA = rD$  then rA is loaded from memory location  $MEM(rA, N)$  where N is determined by the instruction operand size.

**Table 4-13** summarizes the RCPU load instructions.

**Table 4-13 Integer Load Instructions**

Name	Mnemonic	Operand Syntax	Operation
Load Byte and Zero	<b>lbz</b>	<b>rD,d(rA)</b>	The effective address is the sum $(rA 0) + d$ . The byte in memory addressed by the EA is loaded into register <b>rD</b> [24:31]. The remaining bits in register <b>rD</b> are cleared to zero.
Load Byte and Zero Indexed	<b>lbzx</b>	<b>rD,rA,rB</b>	The effective address is the sum $(rA 0) + (rB)$ . The byte in memory addressed by the EA is loaded into register <b>rD</b> [24:31]. The remaining bits in register <b>rD</b> are cleared to zero.
Load Byte and Zero with Update	<b>lbzu</b>	<b>rD,d(rA)</b>	The effective address (EA) is the sum $(rA 0) + d$ . The byte in memory addressed by the EA is loaded into register <b>rD</b> [24:31]. The remaining bits in register <b>rD</b> are cleared to zero. The EA is placed into register <b>rA</b> .  The PowerPC architecture defines load with update instructions with <b>rA</b> = 0 or <b>rA</b> = <b>rD</b> as invalid forms. In the RCPU, however, if <b>rA</b> = 0 then the EA is written into R0. If <b>rA</b> = <b>rD</b> then <b>rA</b> is loaded from memory location <b>MEM(rA, N)</b> where N is determined by the instruction operand size.
Load Byte and Zero with Update Indexed	<b>lbzux</b>	<b>rD,rA,rB</b>	The effective address (EA) is the sum $(rA 0) + (rB)$ . The byte addressed by the EA is loaded into register <b>rD</b> [24:31]. The remaining bits in register <b>rD</b> are cleared to zero. The EA is placed into register <b>rA</b> .  The PowerPC architecture defines load with update instructions with <b>rA</b> = 0 or <b>rA</b> = <b>rD</b> as invalid forms. In the RCPU, however, if <b>rA</b> = 0 then the EA is written into R0. If <b>rA</b> = <b>rD</b> then <b>rA</b> is loaded from memory location <b>MEM(rA, N)</b> where N is determined by the instruction operand size.
Load Half Word and Zero	<b>lhz</b>	<b>rD,d(rA)</b>	The effective address is the sum $(rA 0) + d$ . The half-word in memory addressed by the EA is loaded into register <b>rD</b> [16:31]. The remaining bits in <b>rD</b> are cleared to zero.
Load Half Word and Zero Indexed	<b>lhzx</b>	<b>rD,rA,rB</b>	The effective address is the sum $(rA 0) + (rB)$ . The half-word in memory addressed by the EA is loaded into register <b>rD</b> [16:31]. The remaining bits in register <b>rD</b> are cleared.
Load Half Word and Zero with Update	<b>lhzu</b>	<b>rD,d(rA)</b>	The effective address is the sum $(rA 0) + d$ . The half-word in memory addressed by the EA is loaded into register <b>rD</b> [16:31]. The remaining bits in register <b>rD</b> are cleared.  The EA is placed into register <b>rA</b> .  The PowerPC architecture defines load with update instructions with <b>rA</b> = 0 or <b>rA</b> = <b>rD</b> as invalid forms. In the RCPU, however, if <b>rA</b> = 0 then the EA is written into R0. If <b>rA</b> = <b>rD</b> then <b>rA</b> is loaded from memory location <b>MEM(rA, N)</b> where N is determined by the instruction operand size.



**Table 4-13 Integer Load Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Load Half Word and Zero with Update Indexed	<b>lhzux</b>	<b>rD,rA,rB</b>	<p>The effective address is the sum <math>(rA 0) + (rB)</math>. The half-word in memory addressed by the EA is loaded into register <b>rD</b>[16:31]. The remaining bits in register <b>rD</b> are cleared. The EA is placed into register <b>rA</b>.</p> <p>The PowerPC architecture defines load with update instructions with <b>rA = 0</b> or <b>rA = rD</b> as invalid forms. In the RCP, however, if <b>rA = 0</b> then the EA is written into R0. If <b>rA = rD</b> then <b>rA</b> is loaded from memory location <b>MEM(rA, N)</b> where N is determined by the instruction operand size.</p>
Load Half Word Algebraic	<b>lha</b>	<b>rD,d(rA)</b>	<p>The effective address is the sum <math>(rA) + d</math>. The half-word in memory addressed by the EA is loaded into register <b>rD</b>[16:31]. The remaining bits in register <b>rD</b> are filled with a copy of bit 0 of the loaded half-word.</p>
Load Half Word Algebraic Indexed	<b>lhax</b>	<b>rD,rA,rB</b>	<p>The effective address is the sum <math>(rA 0) + (rB)</math>. The half-word in memory addressed by the EA is loaded into register <b>rD</b>[16:31]. The remaining bits in register <b>rD</b> are filled with a copy of bit 0 of the loaded half-word.</p>
Load Half Word Algebraic with Update	<b>lhau</b>	<b>rD,d(rA)</b>	<p>The effective address is the sum <math>(rA 0) + d</math>. The half-word in memory addressed by the EA is loaded into register <b>rD</b>[16:31]. The remaining bits in register <b>rD</b> are filled with a copy of bit 0 of the loaded half-word. The EA is placed into register <b>rA</b>.</p> <p>The PowerPC architecture defines load with update instructions with <b>rA = 0</b> or <b>rA = rD</b> as invalid forms. In the RCP, however, if <b>rA = 0</b> then the EA is written into R0. If <b>rA = rD</b> then <b>rA</b> is loaded from memory location <b>MEM(rA, N)</b> where N is determined by the instruction operand size.</p>
Load Half Word Algebraic with Update Indexed	<b>lhaux</b>	<b>rD,rA,rB</b>	<p>The effective address is the sum <math>(rA 0) + (rB)</math>. The half-word in memory addressed by the EA is loaded into register <b>rD</b>[16:31]. The remaining bits in register <b>rD</b> are filled with a copy of bit 0 of the loaded half-word. The EA is placed into register <b>rA</b>.</p> <p>The PowerPC architecture defines load with update instructions with <b>rA = 0</b> or <b>rA = rD</b> as invalid forms. In the RCP, however, if <b>rA = 0</b> then the EA is written into R0. If <b>rA = rD</b> then <b>rA</b> is loaded from memory location <b>MEM(rA, N)</b> where N is determined by the instruction operand size.</p>
Load Word and Zero	<b>lwz</b>	<b>rD,d(rA)</b>	<p>The effective address is the sum <math>(rA 0) + d</math>. The word in memory addressed by the EA is loaded into register <b>rD</b>[0:31].</p>
Load Word and Zero Indexed	<b>lwzx</b>	<b>rD,rA,rB</b>	<p>The effective address is the sum <math>(rA 0) + (rB)</math>. The word in memory addressed by the EA is loaded into register <b>rD</b>[0:31].</p>
Load Word and Zero with Update	<b>lwzu</b>	<b>rD,d(rA)</b>	<p>The effective address is the sum <math>(rA 0) + d</math>. The word in memory addressed by the EA is loaded into register <b>rD</b>[0:31]. The EA is placed into register <b>rA</b>.</p> <p>The PowerPC architecture defines load with update instructions with <b>rA = 0</b> or <b>rA = rD</b> as invalid forms. In the RCP, however, if <b>rA = 0</b> then the EA is written into R0. If <b>rA = rD</b> then <b>rA</b> is loaded from memory location <b>MEM(rA, N)</b> where N is determined by the instruction operand size.</p>

Table 4-13 Integer Load Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Load Word and Zero with Update Indexed	<b>lwzux</b>	<b>rD,rA,rB</b>	<p>The effective address is the sum (<math>rA 0</math>) + (<math>rB</math>). The word in memory addressed by the EA is loaded into register <math>rD[0:31]</math>. The EA is placed into register <math>rA</math>.</p> <p>The PowerPC architecture defines load with update instructions with <math>rA = 0</math> or <math>rA = rD</math> as invalid forms. In the RCPU, however, if <math>rA = 0</math> then the EA is written into R0. If <math>rA = rD</math> then <math>rA</math> is loaded from memory location <math>MEM(rA, N)</math> where N is determined by the instruction operand size.</p>

### 4.5.3 Integer Store Instructions

For integer store instructions, the contents of register  $rS$  are stored into the byte, half-word, word or double-word in memory addressed by EA. Many store instructions have an update form, in which register  $rA$  is updated with the effective address. For these forms, the following rules apply:

- If  $rA \neq 0$ , the effective address is placed into register  $rA$ .
- If  $rA = 0$ , the effective address is written into R0. (Although the PowerPC architecture defines store with update instructions with  $rA = 0$  as invalid forms, the RCPU does not.)
- If  $rS = rA$ , the contents of register  $rS$  are copied to the target memory element, then the generated EA is placed into  $rA$ .

A summary of the RCPU integer store instructions is shown in [Table 4-14](#).

**Table 4-14 Integer Store Instructions**

Name	Mnemonic	Operand Syntax	Operation
Store Byte	<b>stb</b>	<b>rS,d(rA)</b>	The effective address is the sum (rA 0) + d. Register rS[24:31] is stored into the byte in memory addressed by the EA.
Store Byte Indexed	<b>stbx</b>	<b>rS,rA,rB</b>	The effective address is the sum (rA 0) + (rB). rS[24:31] is stored into the byte in memory addressed by the EA.
Store Byte with Update	<b>stbu</b>	<b>rS,d(rA)</b>	The effective address is the sum (rA 0) + d. rS[24:31] is stored into the byte in memory addressed by the EA. The EA is placed into register rA.
Store Byte with Update Indexed	<b>stbux</b>	<b>rS,rA,rB</b>	The effective address is the sum (rA 0) + (rB). rS[24:31] is stored into the byte in memory addressed by the EA. The EA is placed into register rA.
Store Half Word	<b>sth</b>	<b>rS,d(rA)</b>	The effective address is the sum (rA 0) + d. rS[16:31] is stored into the half-word in memory addressed by the EA.
Store Half Word Indexed	<b>sthx</b>	<b>rS,rA,rB</b>	The effective address (EA) is the sum (rA 0) + (rB). rS[16:31] is stored into the half-word in memory addressed by the EA.
Store Half Word with Update	<b>sthu</b>	<b>rS,d(rA)</b>	The effective address is the sum (rA 0) + d. rS[16:31] is stored into the half-word in memory addressed by the EA. The EA is placed into register rA.
Store Half Word with Update Indexed	<b>sthux</b>	<b>rS,rA,rB</b>	The effective address is the sum (rA 0) + (rB). rS[16:31] is stored into the half-word in memory addressed by the EA. The EA is placed into register rA.
Store Word	<b>stw</b>	<b>rS,d(rA)</b>	The effective address is the sum (rA 0) + d. Register rS is stored into the word in memory addressed by the EA.
Store Word Indexed	<b>stwx</b>	<b>rS,rA,rB</b>	The effective address is the sum (rA 0) + (rB). rS is stored into the word in memory addressed by the EA.
Store Word with Update	<b>stwu</b>	<b>rS,d(rA)</b>	The effective address is the sum (rA 0) + d. Register rS is stored into the word in memory addressed by the EA. The EA is placed into register rA.
Store Word with Update Indexed	<b>stwux</b>	<b>rS,rA,rB</b>	The effective address is the sum (rA 0) + (rB). Register rS is stored into the word in memory addressed by the EA. The EA is placed into register rA.

#### 4.5.4 Integer Load and Store with Byte Reversal Instructions

**Table 4-15** describes the integer load and store with byte reversal instructions.

**Table 4-15 Integer Load and Store with Byte Reversal Instructions**

Name	Mnemonic	Operand Syntax	Operation
Load Half Word Byte-Reverse Indexed	<b>lhbrx</b>	<b>rD,rA,rB</b>	The effective address is the sum (rA 0) + (rB). Bits 0 to 7 of the half-word in memory addressed by the EA are loaded into rD[24:31]. Bits 8 to 15 of the half-word in memory addressed by the EA are loaded into rD[16:23]. The rest of the bits in rD are cleared to zero.
Load Word Byte-Reverse Indexed	<b>lwbrx</b>	<b>rD,rA,rB</b>	The effective address is the sum (rA 0)+(rB). Bits 0–7 of the word in memory addressed by the EA are loaded into rD[24:31]. Bits 8 to 15 of the word in memory addressed by the EA are loaded into rD[16:23]. Bits 16 to 23 of the word in memory addressed by the EA are loaded into rD[8:15]. Bits 24 to 31 of the word in memory addressed by the EA are loaded into rD[0:7].
Store Half Word Byte-Reverse Indexed	<b>sthbrx</b>	<b>rS,rA,rB</b>	The effective address is the sum (rA 0)+(rB). rS[24:31] are stored into bits 0 to 7 of the half-word in memory addressed by the EA. rS[16:23] are stored into bits 8 to 15 of the half-word in memory addressed by the EA.
Store Word Byte-Reverse Indexed	<b>stwbrx</b>	<b>rS,rA,rB</b>	The effective address is the sum (rA 0)+(rB). rS[24:31] are stored into bits 0 to 7 of the word in memory addressed by EA. Register rS[16:23] are stored into bits 8 to 15 of the word in memory addressed by the EA. Register rS[8:15] are stored into bits 16 to 23 of the word in memory addressed by the EA. rS[0:7] are stored into bits 24 to 31 of the word in memory addressed by the EA.

#### 4.5.5 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs.

The PowerPC architecture defines the load multiple instruction (**lmw**) with **rA** in the range of registers to be loaded as an invalid form. In the RCPu, however, if **rA** is in the range of registers to be loaded, the instruction completes normally, and **rA** is loaded from the memory location as follows:

$$rA \leftarrow \text{MEM}(\text{EA} + (rA - rS) * 4, 4)$$

For integer load and store multiple instructions, the effective address must be a multiple of four. If not, a system alignment exception is generated.

**Table 4-16 Integer Load and Store Multiple Instructions**

Name	Mnemonic	Operand Syntax	Operation
Load Multiple Word	<b>lmw</b>	<b>rD,d(rA)</b>	The effective address is the sum $(rA 0)+d$ . $n = 32 - rD$ . $n$ consecutive words starting at EA are loaded into GPRs rD through 31. If the EA is not a multiple of four the alignment exception handler is invoked.
Store Multiple Word	<b>stmw</b>	<b>rS,d(rA)</b>	The effective address is the sum $(rA 0)+d$ . $n = (32 - rS)$ . $n$ consecutive words starting at the EA are stored from GPRs rS through 31. If the EA is not a multiple of four the alignment exception handler is invoked.

#### 4.5.6 Integer Move String Instructions

The integer move string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields.

Load/store string indexed instructions of zero length have no effect, except that load string indexed instructions of zero length may set register rD to an undefined value.

The PowerPC architecture defines the load string instructions with rA in the range of registers to be loaded as an invalid form. In the RCPu, however, if rA is in the range of registers to be loaded, the instruction completes normally, and rA is loaded from memory.

**Table 4-17 Integer Move String Instructions**

Name	Mnemonic	Operand Syntax	Operation
Load String Word Immediate	<b>lswi</b>	<b>rD,rA,NB</b>	<p>The EA is (rA 0).</p> <p>Let <math>n = NB</math> if <math>NB \neq 0</math>, <math>n = 32</math> if <math>NB = 0</math>; <math>n</math> is the number of bytes to load. Let <math>nr = (n/4)</math>; <math>nr</math> is the number of registers to receive data.</p> <p><math>n</math> consecutive bytes starting at the EA are loaded into GPRs rD through rD + nr - 1. Bytes are loaded left to right in each register. The sequence of registers wraps around to r0 if required. If the four bytes of register rD + nr - 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared to zero.</p> <p>The PowerPC architecture defines the load string instructions with rA in the range of registers to be loaded as an invalid form. In the RCP, however, if rA is in the range of registers to be loaded, the instruction completes normally, and rA is loaded from memory.</p>
Load String Word Indexed	<b>lswx</b>	<b>rD,rA,rB</b>	<p>The EA is the sum (rA 0)+(rB).</p> <p>Let <math>n = XER[25:31]</math>; <math>n</math> is the number of bytes to load.</p> <p>Let <math>nr = CEIL(n/4)</math>; <math>nr</math> is the number of registers to receive data.</p> <p>If <math>n &gt; 0</math>, <math>n</math> consecutive bytes starting at the EA are loaded into registers rD through rD+nr-1.</p> <p>Bytes are loaded left to right in each register. The sequence of registers wraps around to r0 if required. If the four bytes of register rD + nr - 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared to zero.</p> <p>If <math>n = 0</math>, the contents of register rD is undefined.</p> <p>The PowerPC architecture defines the load string instructions with rA in the range of registers to be loaded as an invalid form. In the RCP, however, if rA is in the range of registers to be loaded, the instruction completes normally, and rA is loaded from memory.</p>
Store String Word Immediate	<b>stswi</b>	<b>rS,rA,NB</b>	<p>The EA is (rA 0).</p> <p>Let <math>n = NB</math> if <math>NB \neq 0</math>, <math>n = 32</math> if <math>NB = 0</math>; <math>n</math> is the number of bytes to store. Let <math>nr = CEIL(n/4)</math>; <math>nr</math> is the number of registers to supply data.</p> <p><math>n</math> consecutive bytes starting at the EA are stored from register rS through rS+nr-1.</p> <p>Bytes are stored left to right from each register. The sequence of registers wraps around through r0 if required.</p>
Store String Word Indexed	<b>stswx</b>	<b>rS,rA,rB</b>	<p>The effective address is the sum (rA 0)+(rB).</p> <p>Let <math>n = XER[25:31]</math>; <math>n</math> is the number of bytes to store.</p> <p>Let <math>nr = CEIL(n/4)</math>; <math>nr</math> is the number of registers to supply data.</p> <p><math>n</math> consecutive bytes starting at the EA are stored from register rS through rS+nr-1.</p> <p>Bytes are stored left to right from each register. The sequence of registers wraps around through r0 if required.</p>

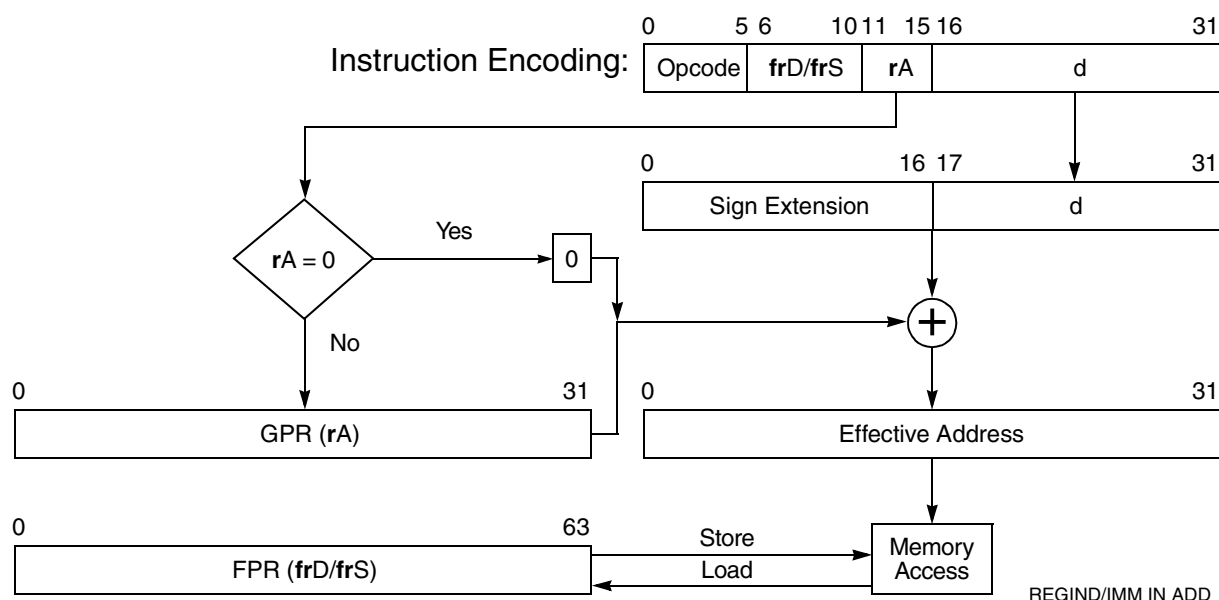
## 4.5.7 Floating-Point Load and Store Address Generation

Floating point load and store operations generate effective addresses using the register indirect with immediate index mode and register indirect with index mode, the details of which are described below.

### 4.5.7.1 Register Indirect with Immediate Index Addressing

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended to 32 bits, and added to the contents of a general purpose register specified in the instruction (rA operand) to generate the effective address. A zero in the rA operand causes a zero to be added to the immediate index (d operand). This is shown in the instruction descriptions as (rA|0).

**Figure 4-4** shows how an effective address is generated when using register indirect with immediate index addressing.

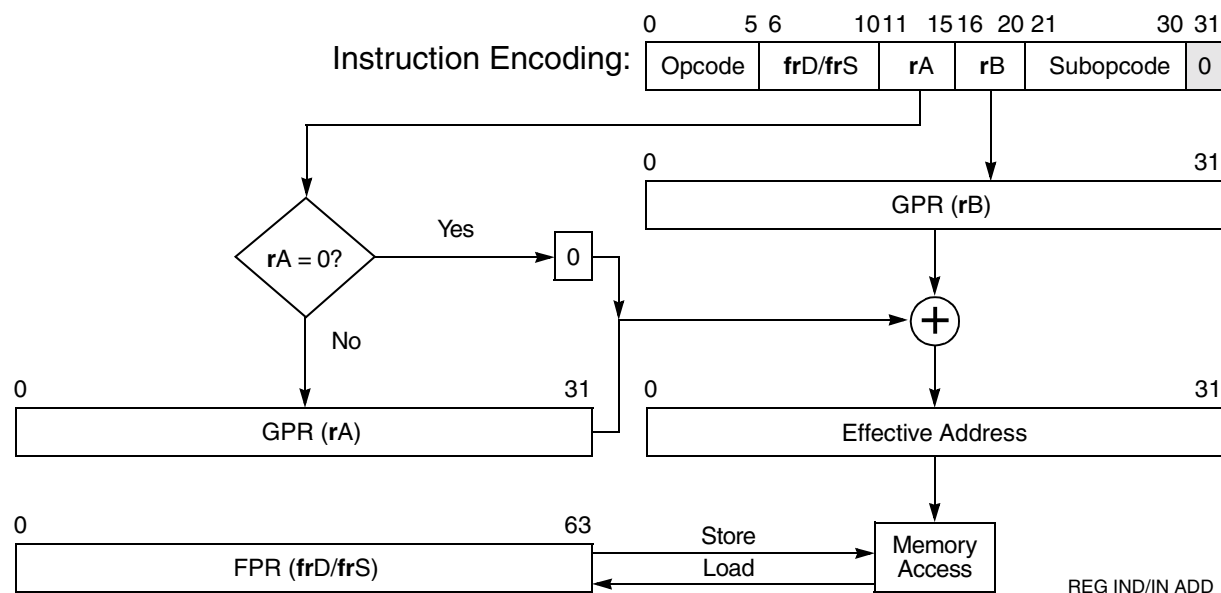


**Figure 4-4 Register Indirect with Immediate Index Addressing**

### 4.5.7.2 Register Indirect with Index Addressing

Instructions using this addressing mode add the contents of two general-purpose registers (specified in operands rA and rB) to generate the effective address. A zero in the rA operand causes a zero to be added to the contents of general-purpose register specified in operand rB. This is shown in the instruction descriptions as (rA|0).

**Figure 4-5** shows how an effective address is generated when using register indirect with index addressing.



**Figure 4-5 Register Indirect with Index Addressing**

### 4.5.8 Floating-Point Load Instructions

There are two basic forms of floating-point load instruction: single-precision and double-precision formats. Because the FPRs support only floating-point, double-precision format, single-precision floating-point load instructions convert single-precision data to double-precision format before loading the operands into the target FPR. This conversion is described in [4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions](#). [Table 4-18](#) provides a summary of the floating-point load instructions.

**Table 4-18 Floating-Point Load Instructions**

Name	Mnemonic	Operand Syntax	Operation
Load Floating-Point Single-Precision	<b>lfs</b>	<b>frD,d(rA)</b>	The effective address is the sum $(rA 0)+d$ . The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into register <b>frD</b> .
Load Floating-Point Single-Precision Indexed	<b>lfsx</b>	<b>frD,rA,rB</b>	The effective address is the sum $(rA 0)+(rB)$ . The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision and placed into register <b>frD</b> .



**Table 4-18 Floating-Point Load Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Load Floating-Point Single-Precision with Update	<b>lfsu</b>	<b>frD,d(rA)</b>	<p>The effective address is the sum <math>(rA 0)+d</math>.</p> <p>The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see <a href="#">4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions</a>) and placed into register <b>frD</b>.</p> <p>The EA is placed into the register specified by <b>rA</b>.</p>
Load Floating-Point Single-Precision with Update Indexed	<b>lfsux</b>	<b>frD,rA,rB</b>	<p>The effective address is the sum <math>(rA 0)+(rB)</math>.</p> <p>The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see <a href="#">4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions</a>) and placed into register <b>frD</b>.</p> <p>The EA is placed into the register specified by <b>rA</b>.</p>
Load Floating-Point Double-Precision	<b>lfd</b>	<b>frD,d(rA)</b>	<p>The effective address is the sum <math>(rA 0)+d</math>.</p> <p>The double-word in memory addressed by the EA is placed into register <b>frD</b>.</p>
Load Floating-Point Double-Precision Indexed	<b>lfdx</b>	<b>frD,rA,rB</b>	<p>The effective address is the sum <math>(rA 0)+(rB)</math>.</p> <p>The double-word in memory addressed by the EA is placed into register <b>frD</b>.</p>
Load Floating-Point Double-Precision with Update	<b>lfd u</b>	<b>frD,d(rA)</b>	<p>The effective address is the sum <math>(rA 0)+d</math>.</p> <p>The double-word in memory addressed by the EA is placed into register <b>frD</b>.</p> <p>The EA is placed into the register specified by <b>rA</b>.</p>
Load Floating-Point Double-Precision with Update Indexed	<b>lfd u x</b>	<b>frD,rA,rB</b>	<p>The effective address is the sum <math>(rA 0)+(rB)</math>.</p> <p>The double-word in memory addressed by the EA is placed into register <b>frD</b>.</p> <p>The EA is placed into the register specified by <b>rA</b>.</p>

#### 4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions

The steps for converting from single- to double-precision and loading are as follows:

WORD[0:31] is the floating-point, single-precision operand accessed from memory.

**Normalized Operand**

If WORD[1:8] > 0 and WORD[1:8] < 255

```
frD[0:1] < WORD[0:1]
frD[2] < ¬WORD[1]
frD[3] < ¬WORD[1]
frD[4] < ¬WORD[1]
frD[5:63] < WORD[2:31] || 290b0
```

**Denormalized Operand**

If WORD[1:8] = 0 and WORD[9:31] ≠ 0

```
sign < WORD[0]
exp < -126
frac[0:52] < 0b0 || WORD[9:31] || 20b0
normalize the operand
Do while frac 0 = 0
    frac < frac[1:52] || 0b0
    exp < exp - 1
End
frD[0] < sign
frD[1:11] < exp + 1023
frD[12:63] < frac[1:52]
```

**Infinity / QNaN / SNaN / Zero**

If WORD[1:8] = 255 or WORD[1:31] = 0

```
frD[1:1] < WORD[0:1]
frD[2] < WORD[1]
frD[3] < WORD[1]
frD[4] < WORD[1]
frD[5:63] < WORD[2:31] || 290b0
```

For double-precision floating-point load instructions, no conversion is required as the data from memory is copied directly into the FPRs.

Many floating-point load instructions have an update form in which register **rA** is updated with the EA. For these forms, the effective address is placed into register **rA** and the memory element (word or double-word) addressed by the EA is loaded into the floating-point register specified by operand **frD**.

**4.5.8.2 Floating-Point Load Single Operands**

If the operand falls in the range of a single denormalized number, the floating-point assist exception handler is invoked. Refer to [6.11.10 Floating-Point Assist Exception \(0x00E00\)](#) for additional information.

**4.5.9 Floating-Point Store Instructions**

There are two basic forms of the floating-point store instruction: single- and double-

precision. Because the FPRs support only floating-point, double-precision format, single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands. The conversion steps are described in [4.5.9.1 Double-Precision Conversion for Floating-Point Store Instructions](#). Table 4-19 is a summary of the floating-point store instructions.

**Table 4-19 Floating-Point Store Instructions**

Name	Mnemonic	Operand Syntax	Operation
Store Floating-Point Single-Precision	<b>stfs</b>	<b>frS,d(rA)</b>	The EA is the sum $(rA 0)+d$ . The contents of register <b>frS</b> is converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single-Precision Indexed	<b>stfsx</b>	<b>frS,rA,rB</b>	The EA is the sum $(rA 0)+(rB)$ . The contents of register <b>frS</b> is converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single-Precision with Update	<b>stfsu</b>	<b>frS,d(rA)</b>	The EA is the sum $(rA 0)+d$ . The contents of register <b>frS</b> is converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into the register specified by operand <b>rA</b> .
Store Floating-Point Single-Precision with Update Indexed	<b>stfsux</b>	<b>frS,rA,rB</b>	The EA is the sum $(rA 0)+(rB)$ . The contents of register <b>frS</b> is converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into the register specified by operand <b>rA</b> .
Store Floating-Point Double-Precision	<b>stfd</b>	<b>frS,d(rA)</b>	The effective address is the sum $(rA 0)+d$ . The contents of register <b>frS</b> is stored into the double-word in memory addressed by the EA.
Store Floating-Point Double-Precision Indexed	<b>stfdx</b>	<b>frS,rA,rB</b>	The EA is the sum $(rA 0)+(rB)$ . The contents of register <b>frS</b> is stored into the double-word in memory addressed by the EA.
Store Floating-Point Double-Precision with Update	<b>stfdu</b>	<b>frS,d(rA)</b>	The effective address is the sum $(rA 0)+d$ . The contents of register <b>frS</b> is stored into the double-word in memory addressed by the EA. The EA is placed into register <b>rA</b> .

Table 4-19 Floating-Point Store Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Store Floating-Point Double-Precision with Update Indexed	<b>stfdux</b>	<b>frS,rA,rB</b>	The EA is the sum (rA 0)+(rB). The contents of register <b>frS</b> is stored into the double-word in memory addressed by EA. The EA is placed into register <b>rA</b> .
Store Floating-Point as Integer Word	<b>stfiwx</b>	<b>frS,rA,rB</b>	The EA is the sum (rA 0)+(rB). The contents of the low-order 32 bits of register <b>frS</b> are stored, without conversion, into the word in memory addressed by EA.

#### 4.5.9.1 Double-Precision Conversion for Floating-Point Store Instructions

The steps for converting single- to double-precision for floating-point store instructions are as follows:

Let WORD[0:31] be the word written in memory.

##### No Denormalization Required

If  $\text{frS}[1:11] > 896$  or  $\text{frS}[1:63] = 0$

WORD[0:1] < frS[0:1]

WORD[2:31] < frS[5:34]

##### Denormalization Required

If  $874 \leq \text{frS}[1:11] \leq 896$

sign < frS[0]

exp < frS[1:11] - 1023

frac < 0b1 || frS[12:63]

Denormalize operand

Do while exp < -126

frac < 0b0 || frac[0:62]

exp < exp + 1

End

WORD0 < sign

WORD[1:8] < 0x00

WORD[9:31] < frac[1:23]

For double-precision floating-point store instructions, no conversion is required as the data from the FPRs is copied directly into memory. Many floating-point store instructions have an update form, in which register **rA** is updated with the effective address. For these forms, if operand **rA**  $\neq 0$ , the effective address is placed into register **rA**.

Floating-point store instructions are listed in [Table 4-19](#). Recall that **rA**, **rB**, and **rD** denote GPRs, while **frA**, **frB**, **frC**, **frS** and **frD** denote FPRs.

## 4.5.9.2 Floating-Point Store-Single Operands

If the operand falls in the range of a single denormalized number, the floating-point assist exception handler is invoked.

If the operand is zero, it is converted to the correct signed zero in single-precision format.

If the operand is between the range of single denormalized and double denormalized, it is considered a programming error. The hardware handles this case as if the operand were single denormalized.

If the operand falls in the range of double denormalized numbers, it is considered a programming error. The hardware handles this case as if the operand were zero.

The following check is done on the stored operand in order to determine whether it is a denormalized single-precision operand and invoke the floating-point assist exception handler:

$$(\text{FRS}[1:11]) \neq 0 \text{ AND } \text{FRS}[1:11] \neq 896$$

Refer to [6.11.10 Floating-Point Assist Exception \(0x00E00\)](#) for a complete description of handling denormalized floating-point numbers.

## 4.5.10 Floating-Point Move Instructions

Floating-point move instructions copy data from one floating-point register to another with data modifications as described for each instruction. These instructions do not modify the FPSCR. The condition register update option in these instructions controls the placing of result status into condition register field CR1. If the condition register update option is enabled, then CR1 is set, otherwise CR1 is unchanged. Floating-point move instructions are listed in [Table 4-20](#).

**Table 4-20 Floating-Point Move Instructions**

Name	Mnemonic	Operand Syntax	Operation
Floating-Point Move Register	<b>fmr</b> <b>fmr.</b>	<b>frD,frB</b>	The contents of register <b>frB</b> is placed into <b>frD</b> .  <b>fmr</b> Floating-Point Move Register <b>fmr.</b> Floating-Point Move Register with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Negate	<b>fneg</b> <b>fneg.</b>	<b>frD,frB</b>	The contents of register <b>frB</b> with bit 0 inverted is placed into register <b>frD</b> .  <b>fneg</b> Floating-Point Negate <b>fneg.</b> Floating-Point Negate with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Absolute Value	<b>fabs</b> <b>fabs.</b>	<b>frD,frB</b>	The contents of <b>frB</b> with bit 0 cleared to zero is placed into <b>frD</b> .  <b>fabs</b> Floating-Point Absolute Value <b>fabs.</b> Floating-Point Absolute Value with CR Update. The dot suffix enables the update of the condition register.
Floating-Point Negative Absolute Value	<b>fnabs</b> <b>fnabs.</b>	<b>frD,frB</b>	The contents of <b>frB</b> with bit 0 set to one is placed into <b>frD</b> .  <b>fnabs</b> Floating-Point Negative Absolute Value <b>fnabs.</b> Floating-Point Negative Absolute Value with CR Update. The dot suffix enables the update of the condition register.

## 4.6 Flow Control Instructions

Branch instructions are executed by the BPU. Some of these instructions can redirect instruction execution conditionally based on the value of bits in the condition register. When the branch processor encounters one of these instructions, it scans the instructions being processed by the various execution units to determine whether an instruction in progress may affect the particular condition register bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the condition register and taking the action defined for the branch instruction.

If an interlock is detected, the branch is considered unresolved and the direction of the branch is predicted using static branch prediction as described in [Table 4-21](#). The interlock is monitored while instructions are fetched for the predicted branch. When the interlock is cleared, the branch processor determines whether the prediction was correct based on the value of the condition register bit. If the prediction is correct, the branch is considered completed and instruction fetching continues. If the prediction is incorrect, the prefetched instructions are purged, and instruction fetching continues along the alternate path.

When the branch instructions contain immediate addressing operands, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be prefetched along the target path. If the branch instructions use the link and count registers, instructions along the target path can be prefetched if the link or count register is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and the return address can optionally be provided. If the return address is to be provided, the effective address of the instruction following the branch instruction is placed in the link register after the branch target address has been computed. This is done regardless of whether the branch is taken.

## 4.6.1 Branch Instruction Address Calculation

Branch instructions can change the sequence of instruction execution. Instruction addresses are always assumed to be on word boundaries; therefore the processor ignores the two low-order bits of the generated branch target address.

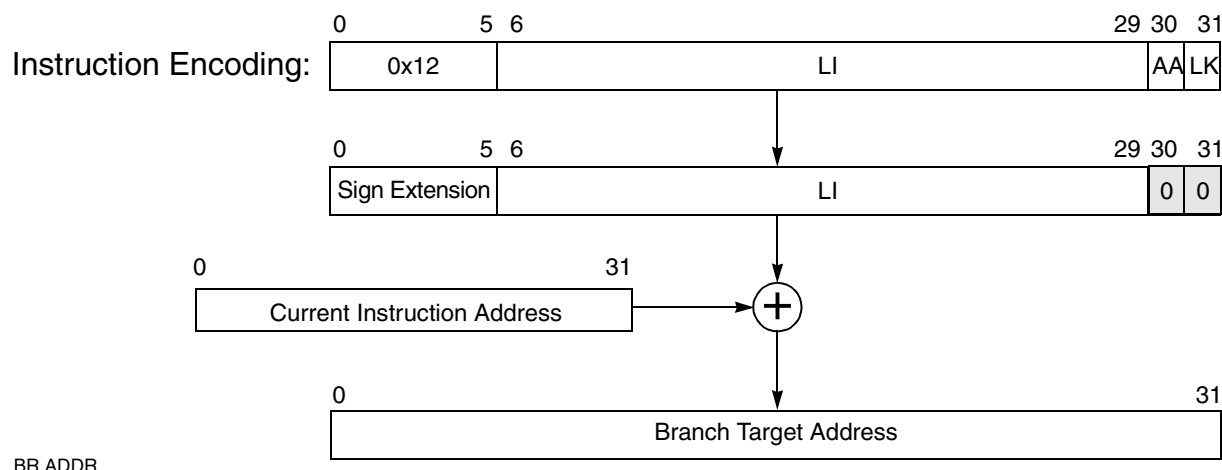
Branch instructions compute the effective address (EA) of the next instruction address using the following addressing modes:

- Branch relative
- Branch to absolute address
- Branch conditional to relative address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

### 4.6.1.1 Branch Relative Address Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand (LI) and adding the resultant value to the current instruction address. Branches using this address mode have the absolute addressing option disabled (AA, bit 30 in the instruction encoding, equals zero). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

**Figure 4-6** shows how the branch target address is generated when using the branch relative addressing mode.



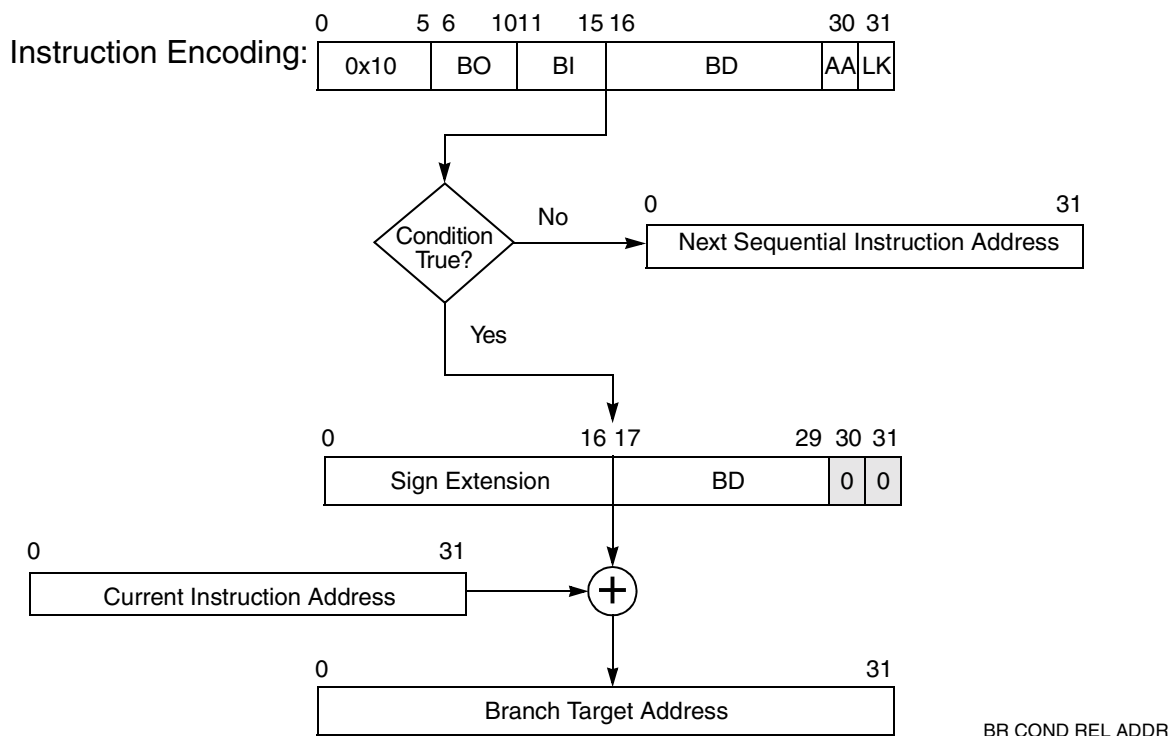
**Figure 4-6 Branch Relative Addressing**

#### 4.6.1.2 Branch Conditional Relative Address Mode

If the branch conditions are met, instructions that use the branch conditional relative address mode generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this address mode have the absolute addressing option disabled (AA, bit 30 in the instruction encoding, equals zero). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

**Figure 4-7** shows how the branch target address is generated when using the branch conditional relative addressing mode.



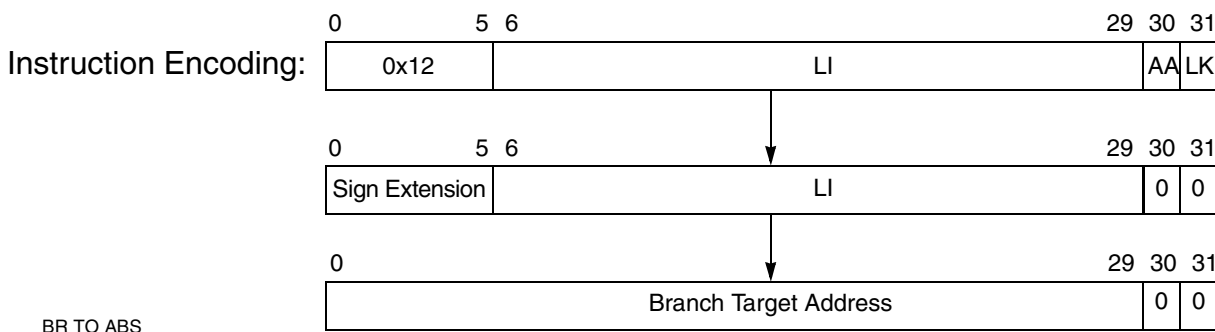


**Figure 4-7 Branch Conditional Relative Addressing**

#### 4.6.1.3 Branch to Absolute Address Mode

Instructions that use branch to absolute address mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this address mode have the absolute addressing option enabled (AA, bit 30 in the instruction encoding, equals one). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

**Figure 4-8** shows how the branch target address is generated when using the branch to absolute address mode.

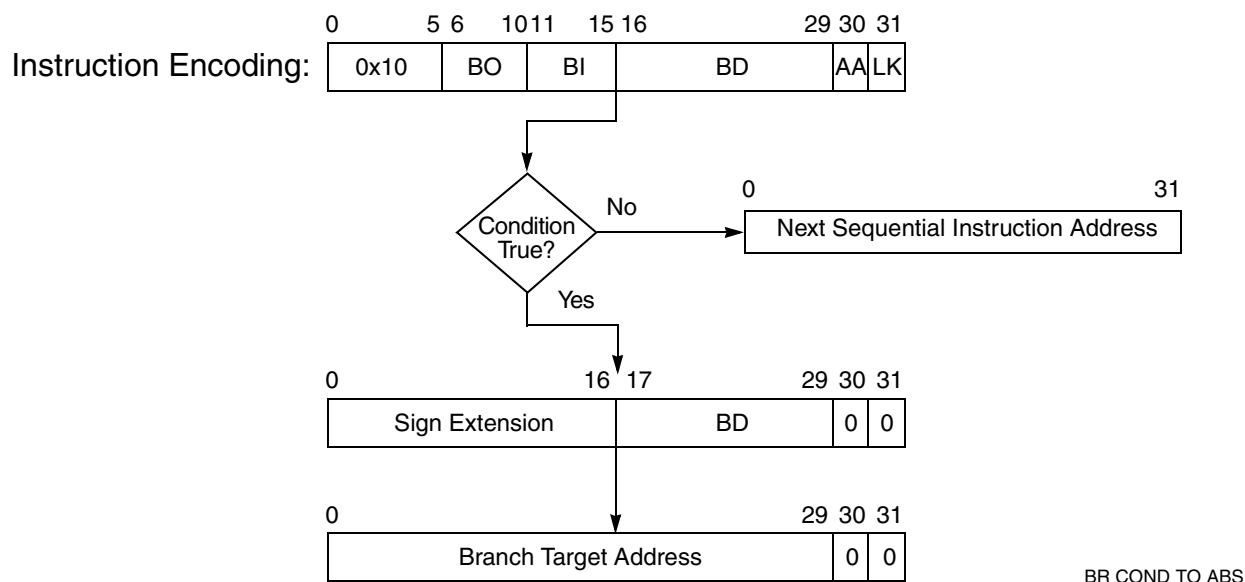


**Figure 4-8 Branch to Absolute Addressing**

#### 4.6.1.4 Branch Conditional to Absolute Address Mode

If the branch conditions are met, instructions that use the branch conditional to absolute address mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this address mode have the absolute addressing option enabled (AA, bit 30 in the instruction encoding, equals one). If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

**Figure 4-9** shows how the branch target address is generated when using the branch conditional to absolute address mode.

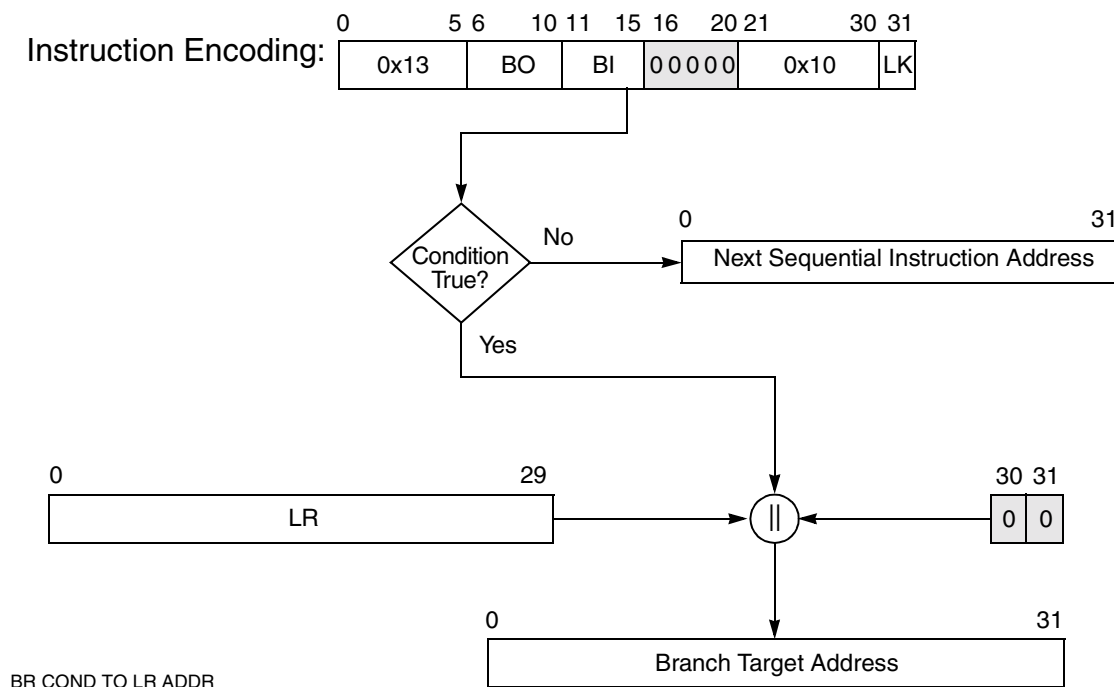


**Figure 4-9 Branch Conditional to Absolute Addressing**

#### 4.6.1.5 Branch Conditional to Link Register Address Mode

If the branch conditions are met, the branch conditional to link register instruction generates the next instruction address by fetching the contents of the link register and clearing the two low order bits to zero. If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

**Figure 4-10** shows how the branch target address is generated when using the branch conditional to link register address mode.

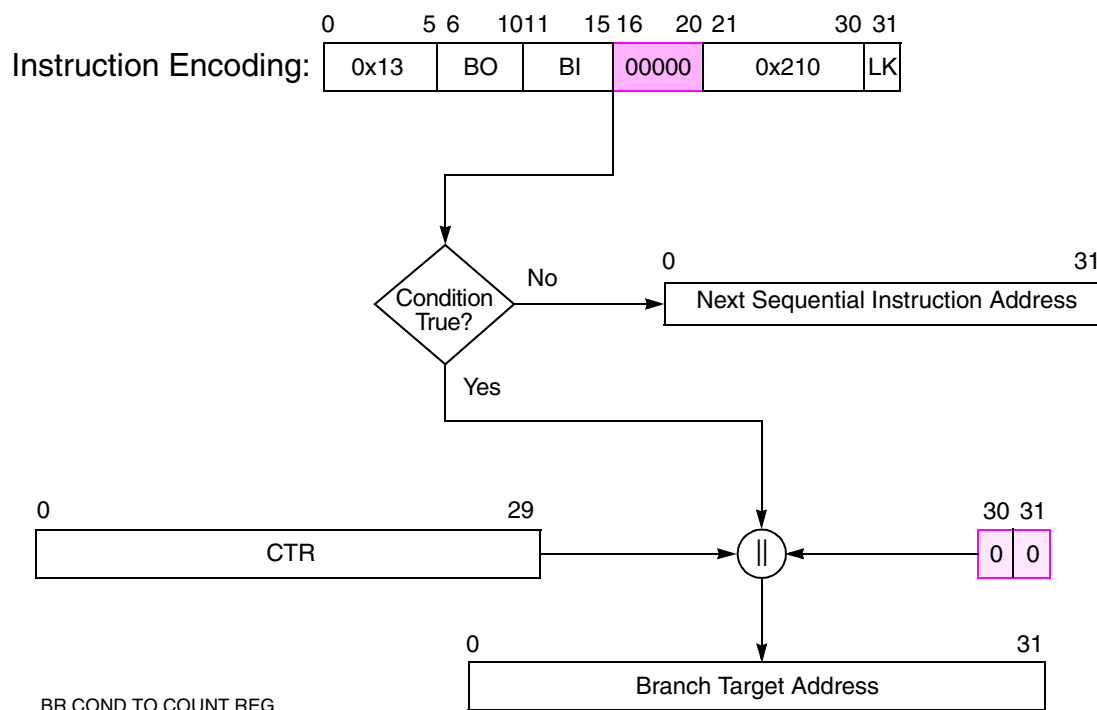


**Figure 4-10 Branch Conditional to Link Register Addressing**

#### 4.6.1.6 Branch Conditional to Count Register

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register and clearing the two low order bits to zero. If the link register update option is enabled (LK, bit 31 in the instruction encoding, equals one), the effective address of the instruction following the branch instruction is placed in the link register.

**Figure 4-11** shows how the branch target address is generated when using the branch conditional to count register address mode.



**Figure 4-11 Branch Conditional to Count Register Addressing**

## 4.6.2 Conditional Branch Control

For branch conditional instructions, the BO and BI operands specify the conditions under which the branch is taken.

### 4.6.2.1 BO Operand and Branch Prediction

The encodings for the BO operand are shown in [Table 4-21](#).

Table 4-21 BO Operand Encodings

BO <sup>1</sup>	Description
0000y	Decrement the CTR, then branch if the decremented CTR $\neq$ 0 and the condition is FALSE.
0001y	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement the CTR, then branch if the decremented CTR $\neq$ 0 and the condition is TRUE.
0101y	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement the CTR, then branch if the decremented CTR $\neq$ 0.
1z01y	Decrement the CTR, then branch if the decremented CTR = 0.
1z1zz	Branch always.

## NOTES:

1. The z indicates a bit that must be zero; otherwise, the instruction form is invalid. The y bit provides a hint about whether a conditional branch is likely to be taken.

The first four bits of the BO operand specify how the branch is affected by or affects the condition and count registers. The fifth bit, shown in [Table 4-21](#) as having the value y, is used for branch prediction. The branch always encoding of the BO operand does not have a y bit.

Clearing the y bit to zero indicates that the following behavior is likely:

- For **bcx** with a negative value in the displacement operand, the branch is taken.
- In all other cases (**bcx** with a non-negative value in the displacement operand, **bclrx**, or **bcctrx**), the branch is not taken.

Setting the y bit to one reverses the preceding indications.

Note that branch prediction occurs for branches to the LR or CTR only if the target address is ready.

The sign of the displacement operand is used as described above even if the target is an absolute address. The default value for the y bit should be zero, and should only be set to one if software has determined that the prediction corresponding to y = one is more likely to be correct than the prediction corresponding to y = zero. Software that does not compute branch predictions should set the y bit to zero.

For all three of the branch conditional instructions, the branch should be predicted to be taken if the value of the following expression is one, and to fall through if the value is zero.

$$((BO[0] \& BO[2]) \mid S) \oplus BO[4]$$

In the expression above, S (bit 16 of the branch conditional instruction coding) is

## Freescale Semiconductor, Inc.

the sign bit of the displacement operand if the instruction has a displacement operand and is zero if the operand is reserved. BO[4] is the y bit, or zero for the branch always encoding of the BO operand. (Advantage is taken of the fact that, for **bclrx** and **bcctrx**, bit 16 of the instruction is part of a reserved operand and therefore must be zero.)

### 4.6.2.2 BI Operand

The 5-bit BI operand in branch conditional instructions specifies which of the 32 bits in the CR represents the condition to test.

### 4.6.2.3 Simplified Mnemonics for Conditional Branches

To provide a simplified mnemonic for every possible combination of BO and BI fields would require  $2^{10} = 1024$  mnemonics, most of which would be only marginally useful. The abbreviated set found in [E.5 Simplified Mnemonics for Branch Instructions](#) is intended to cover the most useful cases. Unusual cases can be coded using a basic branch conditional mnemonic (**bc**, **bclr**, **bcctr**) with the condition to be tested specified as a numeric operand.

### 4.6.3 Branch Instructions

[Table 4-22](#) describes the RCPU branch instructions.

**Table 4-22 Branch Instructions**

Name	Mnemonic	Operand Syntax	Operation
Branch	<b>b</b>	imm_addr	Branch. Branch to the address computed as the sum of the immediate address and the address of the current instruction.
	<b>ba</b>		Branch Absolute. Branch to the absolute address specified.
	<b>bl</b>		Branch then Link. Branch to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the link register (LR).
	<b>bla</b>		Branch Absolute then Link. Branch to the absolute address specified. The instruction address following this instruction is placed into the link register (LR).

**Table 4-22 Branch Instructions (Continued)**

Name	Mnemonic	Operand Syntax	Operation
Branch Conditional	<b>bc</b> <b>bca</b> <b>bcl</b> <b>bcla</b>	BO, BI, target_addr	<p>The BI operand specifies the bit in the condition register (CR) to be used as the condition of the branch. The BO operand is used as described in <a href="#">Table 4-21</a>.</p> <p><b>bc</b> Branch Conditional. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction.</p> <p><b>bca</b> Branch Conditional Absolute. Branch conditionally to the absolute address specified.</p> <p><b>bcl</b> Branch Conditional then Link. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the link register.</p> <p><b>bcla</b> Branch Conditional Absolute then Link. Branch conditionally to the absolute address specified. The instruction address following this instruction is placed into the link register.</p>
Branch Conditional to Link Register	<b>bclr</b> <b>bclrl</b>	BO, BI	<p>The BI operand specifies the bit in the condition register to be used as the condition of the branch. The BO operand is used as described in <a href="#">Table 5-21</a>.</p> <p><b>bclr</b> Branch Conditional to Link Register. Branch conditionally to the address in the link register.</p> <p><b>bclrl</b> Branch Conditional to Link Register then Link. Branch conditionally to the address specified in the link register. The instruction address following this instruction is then placed into the link register.</p>
Branch Conditional to Count Register	<b>bcctr</b> <b>bcctrl</b>	BO, BI	<p>The BI operand specifies the bit in the condition register to be used as the condition of the branch. The BO operand is used as described in <a href="#">Table 5-21</a>.</p> <p><b>bcctr</b> Branch Conditional to Count Register. Branch conditionally to the address specified in the count register.</p> <p><b>bcctrl</b> Branch Conditional to Count Register then Link. Branch conditionally to the address specified in the count register. The instruction address following this instruction is placed into the link register.</p> <p><b>Note:</b> If the “decrement and test CTR” option is specified (BO[2]=0), the instruction form is invalid.</p>

#### 4.6.4 Condition Register Logical Instructions

Similar to the system call (**sc**) instruction, condition register logical instructions, shown in [Table 4-23](#), and the move condition register field (**mcrf**) instruction are defined as flow control instructions, although they are executed by the IU.

Note that if the link register update option (LR) is enabled for any of these instructions, the PowerPC architecture defines these forms of the instructions as invalid.

Table 4-23 Condition Register Logical Instructions

Name	Mnemonic	Operand Syntax	Operation
Condition Register AND	<b>crand</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is ANDed with the bit in the condition register specified by <b>crbB</b> . The result is placed into the condition register bit specified by <b>crbD</b> .
Condition Register OR	<b>cror</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is ORed with the bit in the condition register specified by <b>crbB</b> . The result is placed into the condition register bit specified by <b>crbD</b> .
Condition Register XOR	<b>crxor</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is XORed with the bit in the condition register specified by <b>crbB</b> . The result is placed into the condition register bit specified by <b>crbD</b> .
Condition Register NAND	<b>crnand</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is ANDed with the bit in the condition register specified by <b>crbB</b> . The complemented result is placed into the condition register bit specified by <b>crbD</b> .
Condition Register NOR	<b>crnor</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is ORed with the bit in the condition register specified by <b>crbB</b> . The complemented result is placed into the condition register bit specified by <b>crbD</b> .
Condition Register Equivalent	<b>creqv</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is XORed with the bit in the condition register specified by <b>crbB</b> . The complemented result is placed into the condition register bit specified by <b>crbD</b> .
Condition Register AND with Complement	<b>crandc</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is ANDed with the complement of the bit in the condition register specified by <b>crbB</b> and the result is placed into the condition register bit specified by <b>crbD</b> .
Condition Register OR with Complement	<b>crorc</b>	<b>crbD,crbA,crbB</b>	The bit in the condition register specified by <b>crbA</b> is ORed with the complement of the bit in the condition register specified by <b>crbB</b> and the result is placed into the condition register bit specified by <b>crbD</b> .
Move Condition Register Field	<b>mcrf</b>	<b>crfD,crfS</b>	The contents of <b>crfS</b> are copied into <b>crfD</b> . No other condition register fields are changed.

Refer to [E.6 Simplified Mnemonics for Condition Register Logical Instructions](#) for simplified mnemonics.

#### 4.6.5 System Linkage Instructions

This section describes the system linkage instructions (see [Table 4-29](#)). The system call (**sc**) instruction permits a program to call on the system to perform a service and the system to return from performing a service or from processing an exception.



Table 4-24 System Linkage Instructions

Name	Mnemonic	Operand Syntax	Operand Syntax
System Call	<b>sc</b>	—	<p>When executed, the effective address of the instruction following the <b>sc</b> instruction is placed into SRR0. MSR[16:31] are placed into SRR1[16:31], and SRR1[0:15] are set to undefined values. Then a system call exception is generated.</p> <p>The exception causes the next instruction to be fetched from offset 0xC00 from the base physical address indicated by the new setting of MSR[IP]. Refer to <a href="#">6.11.8 System Call Exception (0x00C00)</a> for more information.</p> <p>This instruction is context synchronizing.</p>
Return from Interrupt	<b>rfi</b>	—	<p>SRR1[16:31] are placed into MSR[16:31], then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0:29]    0b00.</p> <p>This is a supervisor-level, context-synchronizing instruction.</p>

#### 4.6.6 Simplified Mnemonics for Branch and Flow Control Instructions

To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, trap, and certain other instructions; for more information, see [APPENDIX E SIMPLIFIED MNEMONICS](#).

Mnemonics are provided so that branch conditional instructions can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. Some of these are shown as examples with the branch instructions.

PowerPC-compliant assemblers provide the mnemonics and symbols listed here and possibly others.

#### 4.6.7 Trap Instructions

The trap instructions shown in [Table 4-25](#) are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally.

Table 4-25 Trap Instructions

Name	Mnemonic	Operand Syntax	Operand Syntax
Trap Word Immediate	<b>twi</b>	TO,rA,SIMM	The contents of rA is compared with the sign-extended SIMM operand. If any bit in the TO operand is set to one and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.
Trap Word	<b>tw</b>	TO,rA,rB	The contents of rA is compared with the contents of rB. If any bit in the TO operand is set to one and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

The trap instructions evaluate a trap condition as follows:

The contents of register rA is compared with either the sign-extended SIMM field or with the contents of register rB, depending on the trap instruction. The comparison results in five conditions which are ANDed with operand TO. If the result is not zero, the trap exception handler is invoked. These conditions are provided in [Table 4-26](#).

Table 4-26 TO Operand Bit Encoding

TO Bit	ANDed with Condition
0	Less than
1	Greater than
2	Equal
3	Logically less than
4	Logically greater than

A standard set of codes has been adopted for the most common combinations of trap conditions. Refer to [E.7 Simplified Mnemonics for Trap Instructions](#) for a description of these codes and of simplified mnemonics employing them.

## 4.7 Processor Control Instructions

Processor control instructions are used to read from and write to the machine state register (MSR), condition register (CR), and special purpose registers (SPRs).

### 4.7.1 Move to/from Machine State Register and Condition Register Instructions

[Table 4-27](#) summarizes the instructions for reading from or writing to the machine state register and the condition register.

Table 4-27 Move to/from Machine State Register/Condition Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move to Condition Register Fields	<b>mtcrf</b>	CRM,rS	The contents of rS are placed into the condition register under control of the field mask specified by operand CRM. The field mask identifies the 4-bit fields affected. Let $i$ be an integer in the range 0-7. If CRM( $i$ ) = 1, then CR field $i$ (CR bits $4*i$ through $4*i+3$ ) is set to the contents of the corresponding field of rS.
Move to Condition Register from XER	<b>mcrxr</b>	crfD	The contents of XER[0:3] are copied into the condition register field designated by <b>crfD</b> . All other fields of the condition register remain unchanged. XER[0:3] is cleared to zero.
Move from Condition Register	<b>mfcr</b>	rD	The contents of the condition register are placed into rD.
Move to Machine State Register	<b>mtmsr</b>	rS	The contents of rS are placed into the MSR. This instruction is a supervisor-level instruction and is context synchronizing.
Move from Machine State Register	<b>mfmsr</b>	rD	The contents of the MSR are placed into rD. This is a supervisor-level instruction.

#### 4.7.2 Move to/from Special Purpose Register Instructions

Simplified mnemonics are provided for the **mtspr** and **mf spr** instructions so they can be coded with the SPR name as part of the mnemonic rather than as a numeric operand. Some of these are shown as examples with the two instructions. (See [Table 4-28](#).) Refer to [E.8 Simplified Mnemonics for Special-Purpose Registers](#) for a complete list of these mnemonics.

Table 4-28 Move to/from Special Purpose Register Instructions

Name	Mnemonic	Operand Syntax	Operation
Move to Special Purpose Register	<b>mtspr</b>	SPR,rS	<p>The SPR field denotes a special purpose register, encoded as shown in <a href="#">Table 4-29</a> and <a href="#">Table 4-30</a> below. The contents of rS are placed into the designated SPR.</p> <p>Simplified mnemonic examples:</p> <p><b>mtxer rA    mtspr 1,rA</b>  <b>mtlr rA    mtspr 8,rA</b>  <b>mtctr rA    mtspr 9,rA</b></p>
Move from Special Purpose Register	<b>mfspir</b>	rD,SPR	<p>The SPR field denotes a special purpose register, encoded as shown in <a href="#">Table 4-29</a> and <a href="#">Table 4-30</a> below. The contents of the designated SPR are placed into rD.</p> <p>Simplified mnemonic examples:</p> <p><b>mfxer rA    mfspr rA,1</b>  <b>mflr rA    mfspr rA,8</b>  <b>mfctr rA    mfspr rA,9</b></p>

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits [16:20] of the instruction and the low-order five bits in bits [11:15].

[Table 4-29](#) summarizes SPR encodings to which the RCPU permits user-level access.

Table 4-29 User-Level SPR Encodings

Decimal Value in rD	SPR[0:4] SPR[5:9]	Register Name	Description
1	0b00001 00000	XER	Integer exception register
8	0b01000 00000	LR	Link register
9	0b01001 00000	CTR	Count register
268	0b01100 01000	TBL	Time base — lower (read only)
269	0b01101 01000	TBU	Time base — upper (read only)

Table 4-30 summarizes SPR encodings that the RCPU permits at the supervisor level.

Table 4-30 Supervisor-Level SPR Encodings

Decimal Value in rD <sup>1</sup>	SPR[0:4] SPR[5:9]	Register Name	Description
18	0b10010 00000	DSISR	DAE/source instruction service register
19	0b10011 00000	DAR	Data address register
22	0b10110 00000	DEC	Decrementer register
26	0b11010 00000	SRR0	Save and restore register 0
27	0b11011 00000	SRR1	Save and restore register 1
80	0b10000 00010	EIE	External interrupt enable (write only)
81	0b10001 00010	EID	External interrupt disable (write only)
82	0b10010 00010	NRI	Non-recoverable exception
272	0b10000 01000	SPRG0	SPR general 0
273	0b10001 01000	SPRG1	SPR general 1
274	0b10010 01000	SPRG2	SPR general 2
275	0b10011 01000	SPRG3	SPR general 3
284	0b11100 01000	TBL <sup>2</sup>	Time base — lower (write only)
285	0b11101 01000	TBU <sup>2</sup>	Time base — upper (write only)
287	0b11111 01000	PVR	Processor version register (read only)
560	0b10000 10001	ICCST	I-Cache Control and Status Register
561	0b10001 10001	ICADR	I-cache address register
562	0b10010 10001	ICDAT	I-cache data port
1022	0b11110 11111	FPECR	Floating-point exception cause register

NOTES:

1. If the SPR field contains any value other than one of the values shown in [Table 4-30](#), the instruction form is invalid. For an invalid instruction form in which SPR[0]=1, either a privileged instruction type program exception or software emulation exception is generated if the instruction is executed by a user-level program. (Refer to the discussion of these two exception types in [SECTION 6 EXCEPTIONS](#) for more information.) If the instruction is executed by a supervisor-level program, the software emulation exception handler is invoked.  
SPR[0] = 1 if and only if writing the register is supervisor-level. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = 1 results in a privileged instruction type program exception.
2. The PowerPC architecture defines the encodings as TBRs, although it is the same as the SPR encodings. Moving to the time base is performed by the **mtspr** instruction, and moving from the time base is performed by the **mftb** instruction.

[Table 4-31](#) summarizes SPR encodings that the RCPU permits in debug mode, or in supervisor mode when debug mode is not enabled out of reset.

Table 4-31 Development Support SPR Encodings

Decimal Value in rD	SPR[0:4] SPR[5:9]	Register Name	Description
144	0b10000 00010	CMPA	Comparator A Value Register
145	0b10001 00010	CMPB	Comparator B Value Register
146	0b10010 00010	CMPC	Comparator C Value Register
147	0b10011 00010	CMPD	Comparator D Value Register
148	0b10100 00010	ECR	Exception Cause Register
149	0b10101 00010	DER	Debug Enable Register
150	0b10110 00010	COUNTA	Breakpoint Counter A Value and Control
151	0b10111 00010	COUNTB	Breakpoint Counter B Value and Control
152	0b11000 00010	CMPE	Comparator E Value Register
153	0b11001 00010	CMPF	Comparator F Value Register
154	0b11010 00010	CMPG	Comparator G Value Register
155	0b11011 00010	CMPH	Comparator H Value Register
156	0b11100 00010	LCTRL1	L-Bus Support Comparators Control 1
157	0b11101 00010	LCTRL2	L-Bus Support Comparators Control 2
158	0b11110 00010	ICTRL	I-Bus Support Control
159	0b11111 00010	BAR	Breakpoint Address Register
630	0b10110 10011	DPDR	Development Port Data Register

#### 4.7.3 Move from Time Base Instruction

The **mftb** instruction is used to read from the time base register. The instruction is permitted at the user or supervisor privilege level.

Simplified mnemonics for the **mftb** instruction allow it to be coded with the TBR name as part of the mnemonic. Refer to [E.8 Simplified Mnemonics for Special-Purpose Registers](#) for details. Notice that the simplified mnemonics for move from time base and move from time base upper are variants of the **mftb** instruction rather than of **mfspr**. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form and an **mftb** mnemonic with one operand as the simplified form.

Table 4-32 Move from Time Base Instruction

Name	Mnemonic	Operand Syntax	Operation
Move from Time Base	<b>mftb</b>	rD,TBR	The TBR field denotes either the time base lower (TBL) or time base upper (TBU), encoded as shown in <a href="#">Table 4-33</a> . The contents of the designated register are copied to rD.

[Table 4-33](#) summarizes the time base (TBL/TBU) register encodings to which user-level access read access (using the **mftb** instruction) is permitted.

Table 4-33 User-Level TBR Encodings

Decimal Value in rD	SPR[0:4] SPR[5:9]	Register Name	Description
268	0b01100 01000	TBL	Time base lower (read only)
269	0b01101 01000	TBU	Time base upper (read only)

Writing to the time base is permitted at the supervisor privilege level only and is accomplished with the **mtspr** instruction (see [4.7.2 Move to/from Special Purpose Register Instructions](#)) or the **mttb** simplified mnemonic (see [E.8 Simplified Mnemonics for Special-Purpose Registers](#)).

#### 4.8 Memory Synchronization Instructions

Memory synchronization instructions can control the order in which memory operations are completed with respect to asynchronous events and the order in which memory operations are seen by other processors and by other mechanisms that access memory.

The synchronize (**sync**) instruction delays execution of subsequent instructions until all previous instructions have completed (i.e., all internal pipeline stages and instruction buffers have emptied), all previous memory accesses are performed globally, and the **sync** or **eieio** operation is broadcast onto the external bus interface. This set of conditions is referred to as execution serialization (or simply serialization).

The enforce in-order execution of I/O (**eieio**) instruction serializes load/store instructions. No load or store instruction following **eieio** is issued until all loads and stores preceding **eieio** have completed execution.

The instruction synchronize (**isync**) instruction causes the RCPU to halt instruction fetch until all instructions currently in the processor have completed execution, i.e., all issued instructions as well as the pre-fetched instructions waiting to be issued. This condition is referred to as fetch serialization.

## Freescale Semiconductor, Inc.

The proper use of the load word and reserve indexed (**lwarx**) and store word conditional indexed (**stwcx**.) instructions allows programmers to emulate common semaphore operations such as “test and set”, “compare and swap”, “exchange memory”, and “fetch and add”. Examples of these semaphore operations can be found in **APPENDIX D SYNCHRONIZATION PROGRAMMING EXAMPLES**. The **lwarx** instruction must be paired with an **stwcx** instruction with the same effective address used for both instructions of the pair. The reservation granularity is 32 bytes.

The concept behind the use of the **lwarx** and **stwcx** instructions is that a processor may load a semaphore from memory, compute a result based on the value of the semaphore, and conditionally store it back to the same location. The conditional store is performed based on the existence of a reservation established by the preceding **lwarx**. If the reservation exists when the store is executed, the store is performed and a bit is set to one in the condition register. If the reservation does not exist when the store is executed, the target memory location is not modified and a bit is set to zero in the condition register.

The **lwarx** and **stwcx** primitives allow software to read a semaphore, compute a result based on the value of the semaphore, store the new value back into the semaphore location only if that location has not been modified since it was first read, and determine if the store was successful. If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, other processors may have read from the location during this operation.

The **lwarx** and **stwcx** instructions require the EA to be aligned. Exception handling software should not attempt to emulate a misaligned **lwarx** or **stwcx** instruction, because there is no correct way to define the address associated with the reservation.

In general, the **lwarx** and **stwcx** instructions should be used only in system programs, which can be invoked by application programs as needed.

At most one reservation exists at a time on a given processor. The address associated with the reservation can be changed by a subsequent **lwarx** instruction. The conditional store is performed based on the existence of a reservation established by the preceding **lwarx** regardless of whether the address generated by the **lwarx** matches that generated by the **stwcx**. A reservation held by the processor is cleared by any of the following:

- execution of an **stwcx** instruction to any address
- execution of an **sc** instruction
- execution of an instruction that causes an exception
- occurrence of an asynchronous exception
- attempt by some other device to modify a location in the reservation granularity (32 bytes)



When an **lwarx** instruction is executed, the load/store unit issues a cycle to the load/store bus with a special attribute.

In case of an external memory access, this attribute causes the external bus interface (EBI) to set a storage reservation on the cycle address. The EBI must either snoop the external bus or receive some indication from external snoop logic in case the storage reservation is broken by some other processor accessing the same location. When an **stwcx.** instruction to external memory is executed, the EBI checks if the reservation was lost. If so, the cycle is blocked from going to the external bus, and the EBI notifies the LSU that the **stwcx.** instruction did not complete.

The RCPU memory synchronization instructions are summarized in [Table 4-34](#).

**Table 4-34 Memory Synchronization Instructions**

Name	Mnemonic	Operand Syntax	Operation
Enforce In-Order Execution of I/O	<b>eieio</b>	—	The <b>eieio</b> instruction provides an ordering function for the effects of load and store instructions executed by a given processor. Executing an <b>eieio</b> instruction ensures that all memory accesses previously initiated by the given processor are complete with respect to main memory before allowing any memory accesses subsequently initiated by the given processor to access main memory.
Instruction Synchronize	<b>isync</b>	—	This instruction causes instruction fetch to be halted until all instructions currently in the processor have completed execution, i.e., all issued instructions as well as the pre-fetched instructions waiting to be issued.  This instruction has no effect on other processors or on their caches.
Load Word and Reserve Indexed	<b>lwarx</b>	rD,rA,rB	The effective address is the sum (rA 0) + (rB). The word in memory addressed by the EA is loaded into register rD.  This instruction creates a reservation for use by an <b>stwcx.</b> instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation.  The EA must be a multiple of four. If it is not, the alignment exception handler is invoked.
Store Word Conditional Indexed	<b>stwcx.</b>	rS,rA,rB	The effective address is the sum (rA 0) + (rB).  If a reservation exists, register rS is stored into the word in memory addressed by the EA and the reservation is cleared.  If a reservation does not exist, the instruction completes without altering memory.  The EQ bit in the condition register field CR0 is modified to reflect whether the store operation was performed (i.e., whether a reservation existed when the <b>stwcx.</b> instruction began execution). If the store was completed successfully, the EQ bit is set to one.  The EA must be a multiple of four; otherwise, the alignment exception handler is invoked.

Table 4-34 Memory Synchronization Instructions (Continued)

Name	Mnemonic	Operand Syntax	Operation
Synchronize	<b>sync</b>	—	Executing a <b>sync</b> instruction ensures that all instructions previously initiated by the given processor appear to have completed before any subsequent instructions are initiated by the given processor. When the <b>sync</b> instruction completes, all memory accesses initiated by the given processor prior to the <b>sync</b> will have been performed with respect to all other mechanisms that access memory. The <b>sync</b> instruction can be used to ensure that the results of all stores into a data structure, performed in a critical section of a program, are seen by other processors before the data structure is seen as unlocked.

#### 4.9 Memory Control Instructions

This section describes memory control instructions. In the RCPU, only one such instruction is supported: Instruction cache block invalidate (**icbi**).

Table 4-35 Instruction Cache Management Instruction

Name	Mnemonic	Operand Syntax	Operation
Instruction Cache Block Invalidate	<b>icbi</b>	rA,rB	<p>The effective address is the sum (rA 0) + (rB).</p> <p>This instruction causes any subsequent fetch request for an instruction in the block to not find the block in the cache and to be sent to storage. The instruction causes the target block in the instruction cache of the executing processor to be marked invalid. If the target block is not accessible to the program for loads, the system data storage error handler may be invoked.</p> <p>This is a supervisor-level instruction.</p>

#### 4.10 Recommended Simplified Mnemonics

To simplify assembly language programs, a set of simplified mnemonics is provided for some of the most frequently used instructions such as no-op, load immediate, load address, move register, and complement register). PowerPC compliant assemblers provide the simplified mnemonics listed in [E.9 Recommended Simplified Mnemonics](#). Programs written to be portable across the various assemblers for the PowerPC architecture should not assume the existence of mnemonics not defined in this manual.

For a complete list of simplified mnemonics, see [APPENDIX E SIMPLIFIED MNEMONICS](#).

## **SECTION 5 INSTRUCTION CACHE**

The instruction cache (I-cache) is a 4-Kbyte, 2-way set associative cache. The cache is organized into 128 sets, with two lines per set and four words per line. Cache lines are aligned on 4-word boundaries in memory.

A cache access cycle begins with an instruction request from the CPU instruction unit. In case of a cache hit, the instruction is delivered to the instruction unit. In case of a cache miss, the cache initiates a burst read cycle (four beats per burst, one word per beat) on the instruction bus (I-bus) with the address of the requested instruction. The first word received from the bus is the requested instruction. The cache forwards this instruction to the instruction unit as soon as it is received from the I-bus. A cache line is then selected to receive the data that will be coming from the bus. A least-recently-used (LRU) replacement algorithm is used to select a line when no empty lines are available.

Each cache line can be used as an SRAM, allowing the application to lock critical code segments that need fast and deterministic execution time.

Cache coherency in a multiprocessor environment is maintained by software and supported by a fast hardware invalidation capability.

### **5.1 Instruction Cache Organization**

**Figure 5-1** illustrates the I-cache organization.

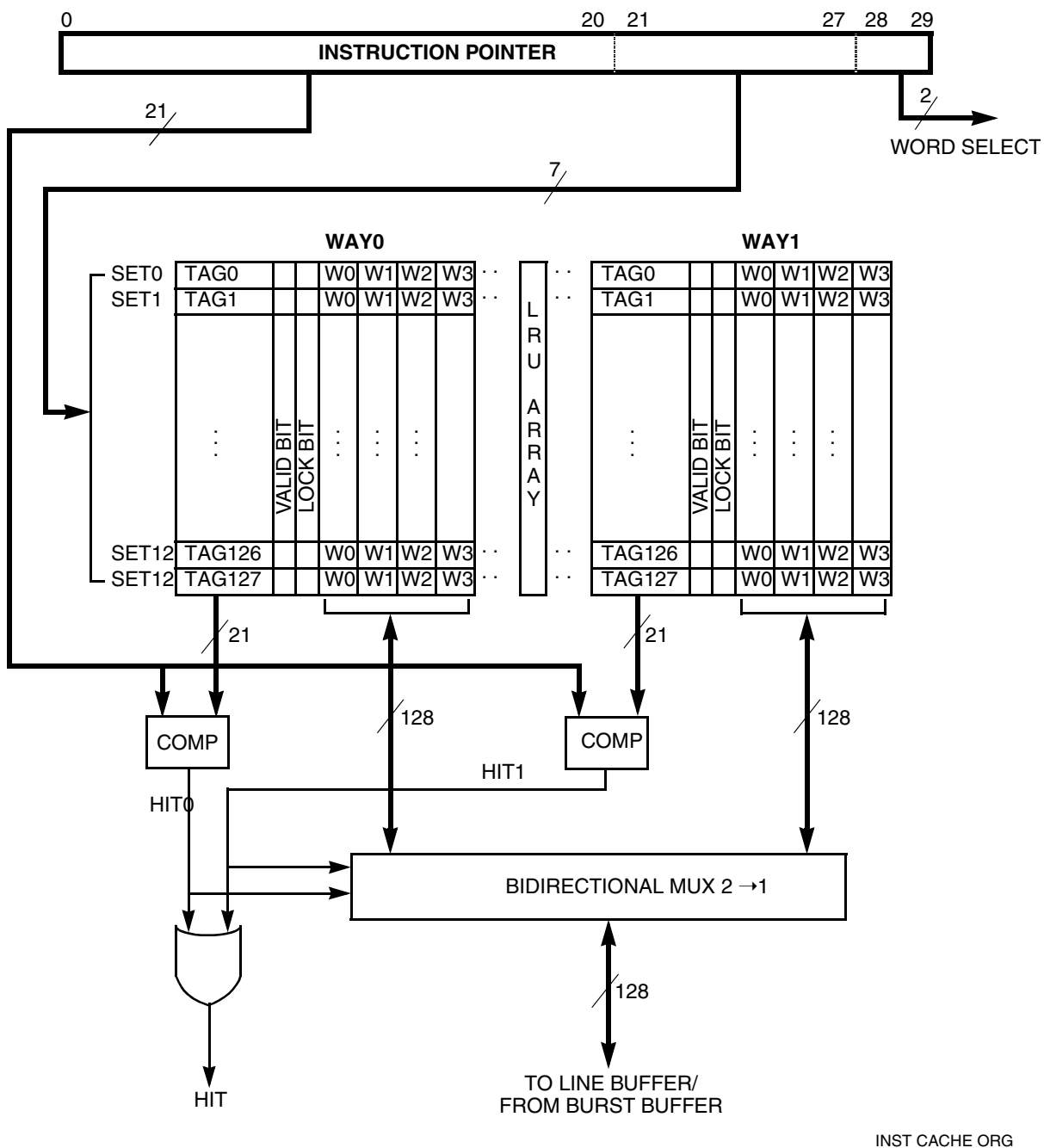


Figure 5-1 Instruction Cache Organization

Figure 5-2 illustrates the data path of the I-cache.

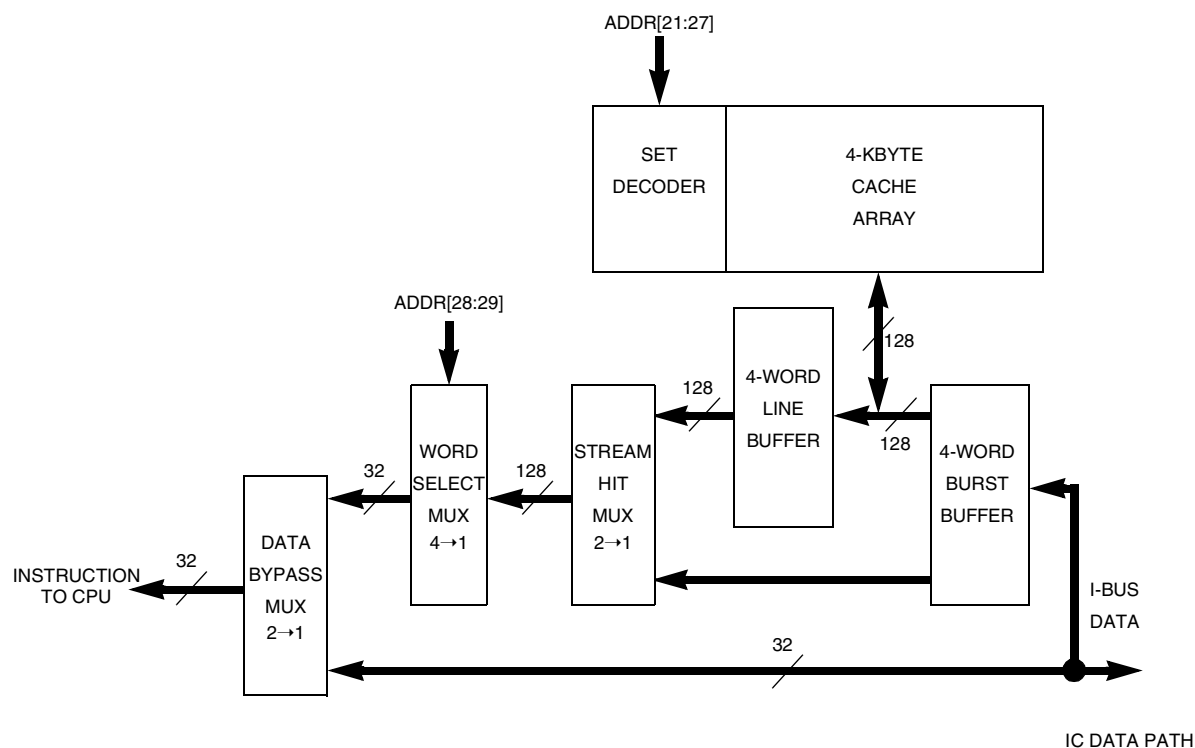


Figure 5-2 Instruction Cache Data Path

## 5.2 Programming Model

**Table 5-1** lists the special purpose registers (SPRs) that control the operation of the I-cache.

Table 5-1 Instruction Cache Programming Model

SPR Number (Decimal)	Name	Description
560	ICCST	I-cache control and status register
561	ICADR	I-cache address register
562	ICDAT	I-cache data port (read only)

These registers are privileged; attempting to access them when the CPU is operating at the user privilege level results in a program interrupt.

### 5.2.1 I-Cache Control and Status Register (ICCST)

The ICCST contains control bits for enabling the I-cache and executing I-cache commands and status bits to indicate error conditions.

## ICCST — I-Cache Control and Status Register

SPR 560

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

RESET:

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

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED															

RESET:

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

**Table 5-2 ICCST Bit Settings**

Bits	Mnemonic	Description
0	IEN	I-cache enable status bit. This bit is a read-only bit. Any attempt to write it is ignored 0 = I-cache is disabled 1 = I-cache is enabled
[1:3]	—	Reserved
[4:6]	CMD	I-Cache Command 000 = No command 001 = <b>Cache enable</b> 010 = <b>Cache disable</b> 011 = <b>Load &amp; lock</b> 100 = <b>Unlock line</b> 101 = <b>Unlock all</b> 110 = <b>Invalidate all</b> 111 = Reserved
[7:9]	—	Reserved
10	CCER1	I-Cache Error Type 1 (sticky bit) 0 = No error 1 = Error
11	CCER2	I-Cache Error Type 2 (sticky bit) 0 = No error 1 = Error
12	CCER3	I-Cache Error Type 3 (sticky bit) 0 = No error 1 = Error
[13:31]	—	Reserved

### 5.2.2 I-Cache Address Register (ICADR)

Writing to the ICADR assigns the address that will be used by subsequent I-cache commands that are programmed in the ICCST.

#### ICADR — I-Cache Address Register

**SPR 561**

0	31
ADR	
RESET: UNDEFINED	

**Table 5-3 ICADR Bit Settings**

Bits	Mnemonic	Description
[0:31]	ADR	The address to be used in the command programmed in the control and status register

### 5.2.3 I-Cache Data Register (ICDAT)

The ICDAT register contains the data received when the I-cache tag array is read.

#### ICDAT — I-Cache Data Register

**SPR 562**

0	31
DAT	
RESET: UNDEFINED	

**Table 5-4 ICDAT Bit Settings**

Bits	Mnemonic	Description
[0:31]	DAT	The data received when reading information from the I-cache

## 5.3 Instruction Cache Operation

On an instruction fetch, bits 21 to 27 of the instruction's address are used as an index into the cache to retrieve the tags and data of one set. The tags from both accessed lines are then compared to bits 0 to 20 of the instruction's address. If a match is found and the matched entry is valid, then the access is a cache hit.

If neither tag matches or if the matched tag is not valid, the access is a cache miss.

The I-cache includes one burst buffer that holds the last line received from the bus, and one line buffer that holds the last line received from the cache array. If the requested data is found in one of these buffers, the access is considered a cache hit.

To minimize power consumption, the I-cache attempts to make use of data stored in one of its internal buffers. Using a special indication from the CPU, it is also possible, in some cases, to detect that the requested data is in one of the buffers early enough so the cache array is not activated at all.

## 5.3.1 Cache Hit

On a cache hit, bits 28 to 29 of the instruction's address are used to select one word from the cache line whose tag matched. In the same clock cycle, the instruction is transferred to the instruction unit of the processor.

## 5.3.2 Cache Miss

On a cache miss, the address of the missed instruction is driven on the I-bus with a four-word burst transfer read request. A cache line is then selected to receive the data that will be coming from the bus. The selection algorithm gives first priority to invalid lines. If neither of the two candidate lines in the selected set are invalid, then the least recently used line is selected for replacement. Locked lines are never replaced.

The transfer begins with the word requested by the instruction unit (critical word first), followed by any remaining words of the line, then by any remaining words at the beginning of the line (wrap around). As the missed instruction is received from the bus, it is immediately delivered to the instruction unit and also written to the burst buffer.

As subsequent instructions are received from the bus they are also written into the burst buffer and, if needed, delivered to the instruction unit (stream hit) either directly from the bus or from the burst buffer. When the entire line resides in the burst buffer, it is written to the cache array if the cache array is not busy with an instruction unit request.

If a bus error is encountered on the access to the requested instruction, a machine check exception is taken. If a bus error occurs on any access to other words in the line, the burst buffer is marked invalid and the line is not written to the array. If no bus error is encountered, the burst buffer is marked valid and eventually is written to the array.

Together with the missed word, an indication may arrive from the I-bus that the memory device is non-cacheable. If such an indication is received, the line is written only to the burst buffer and not to the cache. Instructions stored in the burst buffer that originated in a cache-inhibited memory region are used only once before being refetched. Refer to [5.4.8 Cache Inhibit](#) for more information.

## 5.3.3 Instruction Fetch on a Predicted Path

The processor implements branch prediction to allow branches to issue as early as possible. This mechanism allows instruction pre-fetch to continue while an unresolved branch is being computed and the condition is being evaluated. Instructions fetched following unresolved branches are said to be fetched on a predicted path.



These instructions may be discarded later if it turns out that the machine has followed the wrong path.

To minimize power consumption, the I-cache does not initiate a miss sequence in most cases when the instruction is inside a predicted path. The I-cache evaluates fetch requests to determine whether they are inside a predicted path. If a hit is detected, the requested data is delivered to the processor. However, on a cache miss, in most cases the cache-miss sequence is not initiated until the processor finishes the branch evaluation.

## 5.4 Cache Commands

The instruction cache supports the PowerPC instruction cache block invalidate (**icbi**) instruction together with some additional commands that help control the cache and debug the information stored in it. The additional commands are implemented using the three special purpose control registers ICCST, ICADR, and IC-DAT.

Most of the commands are executed immediately after the control register is written and cannot generate any errors. When these commands are executed, there is no need to check the error status in the ICCST.

The **load & lock** command may take longer and may generate errors. When executing this command, the user needs to insert an **isync** instruction immediately after the I-cache command and check the error status in the ICCST after the **isync** instruction. The error type bits in the ICCST are sticky, allowing the user to perform a series of I-cache commands before checking the termination status. These bits are set by hardware and cleared by software.

Only commands that are not executed immediately need to be followed by an **isync** instruction for the hardware to perform them correctly. However, all commands need to be followed by **isync** in order to make sure all fetches of instructions that follow the I-cache command in the program stream are affected by the I-cache command.

Because the ICCST is a supervisor-level register, cache commands that require setting bits in this register are accessible only at the supervisor privilege level (MSR[PR] = 0). Attempting to write this register at the user privilege level results in a program exception.

The CPU **icbi** instruction (discussed below) can be performed at the user privilege level.

### 5.4.1 Instruction Cache Block Invalidate

The PowerPC instruction cache block invalidate (**icbi**) instruction invalidates the cache block indicated by the effective address in the instruction. The RCPU implements this instruction as if it pertains only to the on-chip instruction cache. This instruction does not broadcast on the external bus, and the RCPU does not snoop this instruction if broadcast by other masters.

This command is not privileged and has no error cases that the user needs to check.

The I-cache performs this instruction in one clock cycle. In order to calculate the latency of this instruction accurately, bus latency should be taken into account.

## 5.4.2 Invalidate All

To invalidate the whole cache, set the **invalidate all** command in the ICCST. This command has no error cases that the user needs to check.

When the command is invoked, if MSR[PR] = 0, all valid lines in the cache, except lines that are locked, are made invalid. As a result of this command, the LRU of all lines points to an unlocked way or to way zero if both lines are not locked. This last feature is useful in order to initialize the I-cache out of reset.

The I-cache performs this instruction in one clock cycle. In order to calculate the latency of this instruction accurately, bus latency should be taken into account.

## 5.4.3 Load and Lock

The **load & lock** operation is used to lock critical code segments in the cache. The **load & lock** operation is performed on a single cache line. After a line is locked it operates as a regular instruction SRAM; it will not be replaced during future misses and will not be affected by invalidate commands.

The following sequence loads and locks one line:

1. Read error type bits in the ICCST in order to clear them
2. Write the address of the line to be locked to the ICADR
3. Set the **load & lock** command in the ICCST
4. Issue the **isync** instruction
5. Return to step 2 to load and lock more lines
6. Read the error type bits in the ICCST to determine whether the operation completed properly

After the **load & lock** command is written to the ICCST, the cache checks if the line containing the byte addressed by the ICADR is in the cache. If it is, the line is locked and the command terminates with no exception. If the line is not in the cache a regular miss sequence is initiated. After the whole line is placed in the cache the line is locked.

The user needs to check the error type bits in the ICCST to determine if the operation completed properly or not. The **load & lock** command can generate two errors:

- Type 1 — bus error in one of the cycles that fetches the line
- Type 2 — no place to lock. It is the responsibility of the user to make sure that there is at least one unlocked way in the appropriate set.

## 5.4.4 Unlock Line

The **unlock line** operation is used to unlock locked cache lines. The **unlock line** operation is performed on a single cache line. If the line is found in the cache (cache hit), it is unlocked and starts to operate as a regular valid cache line. If the line is not found in the cache (cache miss), no operation is performed, and the command terminates with no exception.

The following sequence unlocks one cache line:

1. Write the address of the line to be unlocked to the ICADR
2. Set the **unlock line** command in the ICCST

This command has no error cases that the user needs to check.

The I-cache performs this instruction in one clock cycle. To calculate the latency of this instruction accurately, bus latency should be taken into account.

## 5.4.5 Unlock All

The **unlock all** operation is used to unlock the whole cache. This operation is performed on all cache lines. If a line is locked it is unlocked and starts to operate as a regular valid cache line. If a line is not locked or if it is invalid no operation is performed.

In order to unlock the whole cache set the **unlock all** command in the ICCST.

This command has no error cases that the user needs to check.

The I-cache performs this instruction in one clock cycle. To calculate the latency of this instruction accurately, bus latency should be taken into account.

## 5.4.6 Cache Enable

To enable the cache, set the **cache enable** command in the ICCST. This operation can be performed only at the supervisor privilege level. The **cache enable** command has no error cases that the user needs to check.

## 5.4.7 Cache Disable

To disable the cache, set the **cache disable** command in the ICCST. This operation can be performed only at the supervisor privilege level. The cache disable command has no error cases that the user needs to check.

## 5.4.8 Cache Inhibit

A memory region can be programmed in the chip select logic to be cache inhibited. When an instruction is fetched from a cache-inhibited region, the full line is brought to the internal burst buffer. Instructions stored in the burst buffer that originated from a cache-inhibited region may be sent to the processor no more than once before being re-fetched.

## Freescale Semiconductor, Inc.

When changing a memory region (by writing to the appropriate chip-select registers) from a cache-enabled to a cache-inhibited region, the user must do the following:

1. Unlock all locked lines containing code that originated in this memory region
2. Invalidate all lines containing code that originated in this memory region
3. Execute an **isync** instruction

If these steps are not followed, code from a cache-inhibited region could be left inside the cache, and a reference to a cache-inhibited region could result in a cache hit. When a reference to a cache-inhibited region results in a cache hit, the data is delivered to the processor from the cache, not from memory.

When the FREEZE signal is asserted, indicating that the processor is under debug, all fetches from the cache are treated as if they were from cache-inhibited regions.

### 5.4.9 Cache Read

The user can read all data stored in the I-cache, including the data stored in the tags array.

To read the data that is stored in the I-cache,

1. Write the address of the data to be read to the ICADR. Note that it is also possible to read this register for debugging purposes.
2. Read the ICDAT

So that all parts of the I-cache can be accessed, the ICADR is divided into the following fields:

**Table 5-5 ICADR Bits Function for the Cache Read Command**

[0:17]	18	19	20	[21:27]	[28:29]	[30:31]
Reserved	0 = tag 1 = data	0 = way 0 1 = way 1	Reserved	Set select	Word select (used only for data array)	Reserved

When the data array is read from, the 32 bits of the word selected by the ICADR are placed in the target general-purpose register.

When the tag array is read, the 21 bits of the tag selected by the ICADR, along with additional information, are placed in the target general-purpose register. [Table 5-6](#) illustrates the bits layout of the I-cache data register when a tag is read.

Table 5-6 ICDAT Layout During a Tag Read

[0:20]	21	22	23	24	[25:31]
Tag value	Reserved	0 = not valid 1 = valid	0 = not locked 1 = locked	LRU bit	Reserved

### 5.5 I-Cache and On-Chip Memories with Zero Wait States

On-chip memories on the I-bus are considered to be cache-inhibited memory regions.

Performing a **load & lock** with such an on-chip memory is not advised. In most cases the instruction will still be fetched from the on-chip memory, even though it is also present in the I-cache.

### 5.6 Cache Coherency

Cache coherency in a multi-processor environment is maintained by software and supported by the invalidation mechanisms described in [5.4 Cache Commands](#). All instruction storage is considered to be in “coherency not required” mode.

### 5.7 Updating Code and Attributes of Memory Regions

When updating code or when changing the attributes of memory regions (by writing to chip-select registers), the user must perform the following actions:

1. Update code or change memory region programming in the chip-select logic.
2. Execute the **sync** instruction to ensure the update or change operation finished.
3. Unlock all locked lines containing code that was updated.
4. Invalidate all lines containing code that was updated.
5. Execute the **isync** instruction.

### 5.8 Reset Sequence

To simplify system debugging, the I-cache is forced to be disabled only during reset (ICCST[EN] = 0). This feature enables the user to investigate the exact state of the I-cache prior to the event that asserted the reset.

In order to ensure proper operation of the I-cache after reset, the following actions must be performed:

1. **unlock all**
2. **invalidate all**
3. **cache enable**

## 5.9 Debugging Support

The processor can be debugged either in debug mode or by a software monitor debugger. In both cases the processor asserts the internal FREEZE signal. For a detailed description of RCPU debugging support, refer to **SECTION 8 DEVELOPMENT SUPPORT**.

When FREEZE is asserted, the I-cache treats all misses as if they were from cache-inhibited regions. That is, the misses are loaded only to the burst buffer and the cache state therefore remains exactly the same (assuming the debug routine is not in the I-cache). Notice that when FREEZE is asserted, cache hits are still read from the array, and therefore the LRU bits are updated.

### 5.9.1 Running a Debug Routine from the I-Cache

It may be desirable, in some cases, to be able to run a debug routine from the I-cache (e.g., for performance reasons). The following steps could be used to run the debug routine from the I-cache:

1. Save both ways of the sets that are needed for the debug routine by reading the tag value, LRU bit value, valid bit value, and lock bit value.
2. **Unlock** the locked ways in the selected sets.
3. Use **load & lock** to load and lock the debug routine into the I-cache (**load & lock** operates the same when FREEZE is asserted).
4. Run the debug routine. All accesses to it will result in hits.

After the debug routine is finished, the old state of the I-cache can be restored by following these steps:

1. **Unlock** and **invalidate** all the sets that are used by the debug routine (both ways).
2. Use **load & lock** to restore the old sets.
3. **Unlock** the ways that were not locked before.
4. In order to restore the old state of the LRU, make sure the last access (**load & lock** or **unlock**) is performed the MRU way (not the LRU way).

### 5.9.2 Instruction Fetch from the Development Port

When the processor is in debug mode, all instructions are fetched from the development port, regardless of the address generated by the processor. The I-cache is therefore bypassed in debug mode.

## **SECTION 6 EXCEPTIONS**

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to certain registers, and the processor begins execution at an address predetermined for each exception. Processing of exceptions occurs in supervisor mode.

Although multiple exception conditions can map to a single exception vector, the specific condition can be determined by examining a register associated with the exception — for example, the DAE/source instruction service register (DSISR) and the floating-point status and control register (FPSCR). Additionally, specific exception conditions can be explicitly enabled or disabled by software.

The PowerPC architecture requires that exceptions be handled in program order; therefore, while exception conditions may be recognized out of order, they are handled strictly in order. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream are required to complete before the exception is taken. An instruction is said to have “completed” when the results of that instruction’s execution have been committed to the appropriate registers (i.e., following the writeback stage). If a single instruction encounters multiple exception conditions, those exceptions are taken and handled sequentially.

Asynchronous exceptions (exceptions not associated with a specific instruction) are recognized when they occur, but are not handled until all completed instructions have retired and the instruction remaining at the head of the history buffer is ready to retire.

In many cases, after an exception handler handles an exception, there is an attempt to execute the instruction that caused the exception. Instruction execution continues until the next exception condition is encountered. This method of recognizing and handling exception conditions sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

Exception handlers should save the information saved in SRR0 and SRR1 soon after the exception is taken to prevent this information from being lost due to another exception being taken. The information should be saved before enabling any exception that is automatically disabled when an exception is taken.

### **NOTE**

If debug mode is enabled and the appropriate bit in the debug enable register (DER) is set, recognition of an exception results in debug-mode processing rather than normal exception processing. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for details.

## 6.1 Exception Classes

Exception classes are shown in [Table 6-1](#). These classes are described in the following paragraphs.

**Table 6-1 RCPU Exception Classes**

Type	Exception
Asynchronous, unordered (non-maskable)	System reset Non-maskable data or instruction breakpoint Non-maskable external breakpoint
Asynchronous, ordered (maskable)	External interrupt Decrementer Maskable external breakpoint
Synchronous (precise), ordered	Instruction-caused exceptions (except machine check)
Synchronous (precise), unordered	Machine check

### 6.1.1 Ordered and Unordered Exceptions

An exception is said to be ordered if, when it is taken, it is guaranteed that no program state is lost (provided proper procedures are followed in the exception handlers). In the RCPU implementation of the PowerPC architecture, all exceptions are ordered except for the following:

- Reset
- Machine check
- Non-maskable internal (instruction and data) breakpoints
- Non-maskable external breakpoints

Unordered exceptions may be reported at any time and are not guaranteed to preserve program state information. The processor can never recover from a reset exception. It can recover from other unordered exceptions in most cases. However, if an unordered exception occurs during the servicing of a previous exception, the machine state information in SRR0 and SRR1 (and, in some cases, the DAR and DSISR) may not be recoverable; the processor may be in the process of saving or restoring these registers.

To determine whether the machine state is recoverable, the user can read the RI (recoverable exception) bit in SRR1. Refer to [6.5 Recovery from Exceptions](#) for details.

### 6.1.2 Synchronous, Precise Exceptions

Synchronous exceptions are caused by instructions. They are said to be either precise or imprecise. In the RCPU implementation of the PowerPC architecture, all synchronous exceptions are precise.

When a precise exception occurs, the processor backs the machine up to the in-



## Freescale Semiconductor, Inc.

struction causing the exception. This ensures that the machine is in its correct architecturally-defined state. The following conditions exist at the point a precise exception occurs:

1. Architecturally, no instruction following the faulting instruction in the code stream has begun execution.
2. All instructions preceding the faulting instruction appear to have completed with respect to the executing processor.
3. SRR0 addresses either the instruction causing the exception or the immediately following instruction. Which instruction is addressed can be determined from the exception type and the status bits.
4. Depending on the type of exception, the instruction causing the exception may not have begun execution, may have partially completed, or may have completed execution. Refer to [Table 6-2](#) for details.

The precise exception model can simplify and speed up exception processing because software does not have to save the machine's internal pipeline states, unwind the pipelines, and cleanly terminate the faulting instruction, nor does it have to reverse the process to resume execution of the faulting instruction stream.

### NOTE

In the RCPU implementation of the PowerPC architecture, the machine-check exception is synchronous, (i.e., it is assigned to the instruction that caused it). In other PowerPC implementations, this exception may be asynchronous.

[Table 6-2](#) shows which precise exceptions are taken before the excepting instruction is executed, which are taken after, and which are taken after the instruction is partially executed.

**Table 6-2 Handling of Precise Exceptions**

Exception Type	Instruction Type	Before/After	Contents of SRR0
Machine check	Any	Before	Faulting instruction
Alignment	Multiple	Partially	Faulting instruction
	Others	Before	
Floating-point enabled	Move to MSR, <b>rfi</b>	After	Next instruction to execute
Floating-point enabled	Move to FPSCR	After	Faulting instruction
Privileged instruction, trap, floating-point unavailable	Multiple	Before	Faulting instruction
System call	<b>sc</b>	After	Next instruction to execute
Trace	Any	After	Next instruction to execute
Debug I-breakpoint	Any	Before	Faulting instruction
Debug L-breakpoint	Load/store	After	Faulting instruction + 4
Software emulation	NA	Before	Faulting instruction
Floating-point assist	Floating point	Before	Faulting instruction

### 6.1.3 Asynchronous Exceptions

Asynchronous exceptions are not caused by instructions and are thus not synchronized to internal processor events. When an asynchronous exception occurs, the following conditions exist at the exception point:

- SRR0 addresses the instruction that would have completed if the exception had not occurred.
- An exception is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing processor.

Asynchronous exceptions can be either ordered or unordered, depending on whether they are maskable.

Maskable exceptions are considered ordered because, if proper software procedures are followed, they are never recognized while the processor is saving or restoring the machine state during a previous exception. Thus, the processor can always recover from one of these exceptions.

Asynchronous, non-maskable exceptions can occur while other exceptions are being processed. If one of these exceptions occurs immediately after another exception, the state information saved by the first exception may be overwritten when the second exception occurs. These exceptions are thus considered unordered. For additional information, refer to [6.5.2 Recovery from Unordered Exceptions](#).

### 6.1.3.1 Asynchronous, Maskable Exceptions

The RCPU supports the following asynchronous, maskable exceptions: external interrupts, decremter interrupts, and maskable internal and external breakpoint exceptions.

External and decremter interrupts are masked by the external interrupt enable (EE) bit in the MSR. When MSR[EE] = 0, these exception conditions are latched and are not recognized until MSR[EE] is set. MSR[EE] is cleared automatically when an exception is taken to delay recognition of external and decremter interrupts.

Maskable internal or external breakpoint exceptions are recognized only when the RI (recoverable exception) bit in the MSR = 1. This ensures that (with proper software safeguards) the processor can always recover from one of these exceptions.

Refer to **SECTION 8 DEVELOPMENT SUPPORT** for details on maskable and non-maskable internal and external breakpoints.

### 6.1.3.2 Asynchronous, Non-Maskable Exceptions

Asynchronous, non-maskable exceptions include reset and non-maskable internal and external breakpoint exceptions. These exceptions have the highest priority and can occur while other exceptions are being processed. Because these exceptions are non-maskable, they are never delayed; therefore, if an asynchronous, non-maskable exception occurs immediately after another exception, the state information saved by the first exception may be overwritten when the second exception occurs.

For additional information, refer to **6.5.2 Recovery from Unordered Exceptions**. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for details on maskable and non-maskable internal and external breakpoints.

## 6.2 Exception Vector Table

The setting of the exception prefix (IP) bit in the MSR determines how exceptions are vectored. If the bit is cleared, exceptions are vectored to the physical address 0x0000 0000 plus the vector offset; if IP is set, exceptions are vectored to the physical address 0xFFFF0 0000 plus the vector offset. **Table 6-3** shows the exception vector offset of the first instruction of the exception handler routine for each exception type.

### NOTE

The exception vectors shown in **Table 6-3**, up to and including the floating-point assist exception (vector offset 0x00E00), are defined by the PowerPC architecture. Exception vectors beginning with offset 0x01000 (software emulation exception in the RCPU) are reserved in the PowerPC architecture for implementation-specific exceptions.

## Table 6-3 Exception Vectors and Conditions

Exception Type	Vector Offset	Causing Conditions
Reserved	0x00000	Reserved
System reset	0x00100	A reset exception results when the $\overline{\text{RESET}}$ input to the processor is asserted.
Machine check	0x00200	A machine check exception results when the $\overline{\text{TEA}}$ signal is asserted internally or externally.
—	0x00300	Reserved. (In the PowerPC architecture, this exception vector is reserved for data access exceptions.)
—	0x00400	Reserved. (In the PowerPC architecture, this exception vector is reserved for instruction access exceptions.)
External interrupt	0x00500	An external interrupt occurs when the RCPU $\overline{\text{IRQ}}$ input signal is asserted.
Alignment	0x00600	An alignment exception is caused when the processor cannot perform a memory access for one of the following reasons: <ul style="list-style-type: none"> <li>The operand of a floating-point load or store is not word-aligned.</li> <li>The operand of a load- or store-multiple instruction is not word-aligned.</li> <li>The operand of <b>lwarx</b> or <b>stwcx</b> is not word-aligned.</li> <li>In little-endian mode, an operand is not properly aligned.</li> <li>In little-endian mode, the processor attempts to execute a multiple or string instruction.</li> </ul>
Program	0x00700	A program exception is caused by one of the following exception conditions: <ul style="list-style-type: none"> <li>Floating-point enabled exception — A floating-point enabled program exception condition is generated when the following condition is met as a result of a move to FPSCR instruction, move to MSR instruction, or return from interrupt instruction: (MSR[FE0]   MSR[FE1]) &amp; FPSCR[FEX] = 1.</li> <li>Privileged instruction — A privileged instruction type program exception is generated when the execution of a privileged instruction is attempted and the MSR register user privilege bit, MSR[PR], is set. This exception is also generated for <b>mtspr</b> or <b>mfspr</b> with an invalid SPR field if SPR[0]=1 and MSR[PR]=1.</li> <li>Trap — A trap type program exception is generated when any of the conditions specified in a trap instruction is met.</li> </ul>
Floating-point unavailable	0x00800	A floating-point unavailable exception is caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit is disabled, MSR[FP]=0.
Decrementer	0x00900	The decrementer exception occurs when the most significant bit of the decrementer (DEC) register changes from zero to one.
Reserved	0x00A00	—
Reserved	0x00B00	—
System call	0x00C00	A system call exception occurs when a system call ( <b>sc</b> ) instruction is executed.
Trace	0x00D00	A trace exception occurs if MSR[SE] = 1 and any instruction other than <b>rfi</b> is successfully completed, or if MSR[BE] = 1 and a branch is completed.

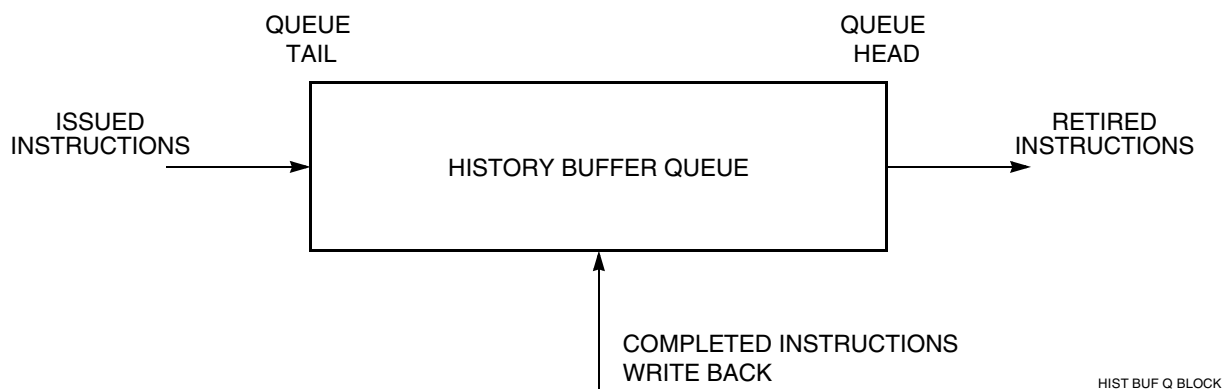
**Table 6-3 Exception Vectors and Conditions (Continued)**

Exception Type	Vector Offset	Causing Conditions
Floating-point assist	0x00E00	<p>A floating-point assist exception occurs in the following cases:</p> <ul style="list-style-type: none"> <li>When the following condition is true (except in the cases mentioned above for program exceptions): (MSR[FE0]   MSR[FE1]) &amp; FPSCR[FEX] = 1</li> <li>When a tiny result is detected and the floating-point underflow exception is disabled (FPSCR[UE] = 0)</li> <li>In some cases when at least one of the source operands is denormalized.</li> </ul>
Software emulation	0x01000	An implementation-dependent software emulation exception occurs when an attempt is made to execute an unimplemented instruction, or to execute a <b>mtspr</b> or <b>mfspr</b> instruction that specifies an unimplemented register.
Data breakpoint	0x01C00	An implementation-dependent data breakpoint exception occurs when an internal breakpoint match occurs on the load/store bus.
Instruction breakpoint	0x01D00	An implementation-dependent instruction breakpoint exception occurs when an internal breakpoint match occurs on the instruction bus.
Maskable external breakpoint	0x01E00	An implementation-dependent maskable external breakpoint occurs when an external device or on-chip peripheral generates a maskable breakpoint.
Non-maskable external breakpoint	0x01F00	An implementation-dependent non-maskable external breakpoint occurs when an external breakpoint is input to the serial interface of the development port.

### 6.3 Precise Exception Model Implementation

In order to achieve maximum performance, the RCPU processes many pieces of the instruction stream concurrently. Instructions execute in parallel and may complete out of order. The processor is designed to ensure that this out of order operation never has an effect different from that specified by the program. This requirement is most difficult to ensure when an exception occurs after instructions that logically follow the faulting instruction have already completed. When an exception occurs, the machine state becomes visible to other processes and therefore must be in its correct architecturally specified condition. The processor takes care of this in hardware by automatically backing the machine up to the instruction that caused the interrupt. The processor is therefore said to implement a precise exception model.

To enable the processor to recover from an exception, a history buffer is used. This buffer is a FIFO queue which records relevant machine state at the time of each instruction issue. Instructions are placed on the tail of the queue when they are issued and percolate to the head of the queue while they are in execution. Instructions remain in the queue until they complete execution (i.e., have completed the writeback stage) and all preceding instructions have completed as well. In this way, when an exception occurs, the machine state necessary to recover the architectural state is available. As instructions complete execution, they are retired from the queue, and the buffer storage is reclaimed for new instructions entering the queue.



**Figure 6-1 History Buffer Queue**

An exception can be detected at any time during instruction execution and is recorded in the history buffer when the instruction finishes execution. The exception is not recognized until the faulting instruction reaches the head of the history queue. When the exception is recognized, exception processing begins. The queue is reversed, and the machine is restored to its state at the time the instruction was issued. Machine state is restored at a maximum rate of two floating-point and two integer instructions per clock cycle.

To correctly restore the architectural state, the history buffer must record the value of the destination before the instruction is executed. The destination of a store instruction, however, is in memory. It is not practical for the processor to always read memory before writing it. Therefore, stores issue immediately to store buffers, but do not update memory until all previous instructions have completed execution without exception, i.e., until the store has reached the head of the history buffer.

The history buffer has enough storage to hold a total of six instructions. Of these, a maximum of four can be integer instructions (including integer load or store instructions), and a maximum of three can be floating-point instructions (including floating-point loads or stores). If the buffer includes an instruction with long latency, it is possible (if a data dependency does not occur first) for issued instructions to fill up the history buffer. If so, instruction issue halts until the long-latency operation retires (along with any instructions following it that are ready to retire). Instructions that can cause the history buffer to fill up include floating-point arithmetic instructions, integer divide instructions, and instructions that affect or use resources external to the processor (e.g., load/store instructions).

## 6.4 Implementation of Asynchronous Exceptions

When an enabled asynchronous exception is detected, the processor attempts to retire as many instructions as possible. That is, all instructions that have completed the writeback stage without generating exceptions are allowed to retire, provided all instructions ahead of them in the history buffer have also completed the writeback stage without generating exceptions.

## Freescale Semiconductor, Inc.

The asynchronous exception is then assigned to the instruction at the head of the history buffer, which has not yet completed (otherwise, it would have been retired). If this instruction is one of the following, it is allowed to complete execution and retire:

- **mtspr**, **mtmsr**, or **rfi** instruction
- Memory reference that is already on the bus (other than a load or store multiple or string instruction)
- Cache control instruction.

In this case, the exception is assigned to the next instruction in the history buffer. Notice that if the instruction at the head of the history buffer generates an exception before it retires, that exception is treated before the asynchronous exception.

If the instruction is not one of those listed above, it and all subsequent instructions are flushed from the buffer as if they were never executed at all.

### 6.5 Recovery from Exceptions

The processor should always be able to recover from an ordered exception. Provided no machine state information is lost, the processor can recover from unordered exceptions, except reset, as well.

#### 6.5.1 Recovery from Ordered Exceptions

The RCPU can always recover from an ordered exception, provided the exception-handling software follows proper procedures. Exception handlers must ensure that no exception-generating instruction is executed during the prologue (before appropriate registers are saved) or epilogue (between restore of these registers and the execution of the **rfi** instruction). Registers that need to be saved are the machine status save/restore registers (SRR0 and SRR1) and, for certain exceptions, the DAR (data address register) and DSISR (data storage interrupt status register).

Hardware automatically clears MSR[EE] during exception processing in order to disable external and decrementer interrupts. If desired, the user can set this bit at the end of the exception handler prologue, after saving the machine state. In this case, the user must clear the bit (along with the RI bit) before the start of the exception handler epilog. Refer to [6.5.3 Commands to Alter MSR\[EE\] and MSR\[RI\]](#) for instructions on altering these bits.

#### 6.5.2 Recovery from Unordered Exceptions

Unless it is in the process of saving or restoring machine state, the processor can recover from the following unordered exceptions:

- Machine check
- Non-maskable external breakpoint
- Non-maskable internal instruction or data breakpoint

The RI bit (recoverable exception) in the MSR and its shadow in SRR1 enable an exception handler to determine whether the processor can recover from an exception. During exception processing, the RI bit in the MSR is copied to SRR1; the bit



in the MSR is then cleared. Each exception handler should set the RI bit in the MSR (using the **mtmsr** instruction) at the end of its prologue, after saving the program state (SRR0, SRR1, and, in some cases, DSISR and DAR). At the start of its epilogue (before saving the machine state), each exception handler should clear the RI bit in the MSR.

In this way, the exception handler for an unordered exception can read the RI bit in SRR1 to determine whether the processor can recover from the exception. If the exception occurs while the machine state is being saved or restored during the processing of a previous exception, the RI bit in SRR1 will be cleared, indicating that the processor cannot recover from the exception. If the exception occurs at any other time, the RI bit in SRR1 will be set, indicating the processor can recover from the exception.

In critical code sections where MSR[EE] is negated but SRR0 and SRR1 are not busy, MSR[RI] should be left asserted. In these cases if an exception occurs, the processor can be restarted.

### 6.5.3 Commands to Alter MSR[EE] and MSR[RI]

The processor includes special commands to facilitate the software manipulation of the MSR[RI] and MSR[EE] bits. These commands are executed by issuing the **mtspr** instruction with one of the pseudo-SPRs shown in [Table 6-4](#). Writing any data to one of these locations performs the operation specified in the table. A read (**mfspr**) of any of these locations is treated as an unimplemented instruction, resulting in a software emulation exception.

**Table 6-4 Manipulating EE and RI Bits**

SPR # (Decimal)	Mnemonic	MSR[EE]	MSR[RI]	Use
80	EIE	1	1	External Interrupt Enable: <ul style="list-style-type: none"> <li>End of exception handler's prologue, to enable nested external interrupts;</li> <li>End of critical code segment in which external interrupts were disabled</li> </ul>
81	EID	0	1	External Interrupt Disable, but other interrupts are recoverable: <ul style="list-style-type: none"> <li>End of exception handler's prologue, to keep external nested interrupts disabled;</li> <li>Start of critical code segment in which external interrupts are disabled</li> </ul>
82	NRI	0	0	Non-Recoverable Interrupt: <ul style="list-style-type: none"> <li>Start of exception handler's epilogue</li> </ul>

### 6.6 Exception Order and Priority

When multiple conditions that can cause an exception are present, the highest-priority exception is taken. Exceptions are roughly prioritized by exception class, as follows:



## Freescale Semiconductor, Inc.

1. Asynchronous, non-maskable exceptions have priority over all other exceptions. These exceptions cannot be delayed and do not wait for the completion of any precise exception handling.
2. Synchronous exceptions are caused by instructions and are handled in strict program order.
3. Asynchronous, maskable exceptions (external interrupt, decrements exceptions, and maskable breakpoint exceptions) are delayed until exceptions caused by the instruction at the head of the history buffer (after instructions that have already completed have retired) are taken.

The exceptions are listed in [Table 6-5](#) in order of highest to lowest priority.

**Table 6-5 Exception Priorities**

Class	Priority	Exception
Asynchronous, non-maskable	1	Non-maskable external breakpoint — This exception has the highest priority and is taken immediately, regardless of other pending exceptions or whether the machine state is recoverable.
	2	Reset —The reset exception has the second-highest priority and is taken immediately, regardless of other pending exceptions (except for the non-maskable external breakpoint exception) or whether the machine state is recoverable.
Synchronous	3	Instruction dependent — When an instruction causes an exception, the exception mechanism waits for any instructions prior to the exception instruction in the instruction stream to execute. Any exceptions caused by these instructions are handled first. It then generates the appropriate exception if no higher priority exception exists when the exception is to be generated.

Table 6-5 Exception Priorities (Continued)

Class	Priority	Exception
Asynchronous, maskable	4	Peripheral or external maskable breakpoint request — When this exception type occurs, the processor retires as many instructions as possible (i.e., all instructions that have completed the writeback stage without generating an instruction, provided all instructions ahead of it in the history buffer have also completed the writeback stage without generating an exception). Then, depending on the instruction currently at the head of the history buffer, the processor either flushes the history buffer or allows the instruction at the head of the buffer to retire before generating an exception. Refer to <a href="#">6.4 Implementation of Asynchronous Exceptions</a> .
	5	External interrupt — When this exception type occurs, the processor retires as many instructions as possible (i.e., all instructions that have completed the writeback stage without generating an instruction, provided all instructions ahead of it in the history buffer have also completed the writeback stage without generating an exception). Then, depending on the instruction currently at the head of the history buffer, the processor either flushes the history buffer or allows the instruction at the head of the buffer to retire before generating an exception (provided a higher priority exception does not exist). Refer to <a href="#">6.4 Implementation of Asynchronous Exceptions</a> . This exception is delayed if MSR[EE] is cleared.
	6	Decrementer — This exception is the lowest priority exception. When this exception type occurs, the processor retires as many instructions as possible (i.e., all instructions that have completed the writeback stage without generating an instruction, provided all instructions ahead of it in the history buffer have also completed the writeback stage without generating an exception). Then, depending on the instruction currently at the head of the history buffer, the processor either flushes the history buffer or allows the instruction at the head of the buffer to retire before generating an exception (provided a higher priority exception does not exist). Refer to <a href="#">6.4 Implementation of Asynchronous Exceptions</a> . This exception is delayed if MSR[EE] is cleared.

## 6.7 Ordering of Synchronous, Precise Exceptions

Synchronous exceptions are handled in strict program order, even though instructions can execute and exceptions can be detected out of order. Therefore, before the RCPU processes an instruction-caused exception, it executes all instructions and handles any resulting exceptions that appear earlier in the instruction stream.

Only one synchronous, precise exception can be reported at a time. If single instructions generate multiple exception conditions, the processor handles the exception it encounters first; then the execution of the excepting instruction continues until the next excepting condition is encountered. [Table 6-6](#) lists the order in which synchronous exceptions are detected.

Table 6-6 Detection Order of Synchronous Exceptions

Order of Detection	Exception Type
1	Trace <sup>1</sup>
2	Machine check during instruction fetch
3	I-bus breakpoint
4	Software emulation exception
5	Floating-point unavailable
6 <sup>2</sup>	Privileged instruction
	Alignment exception
	Floating-point enabled exception
	System call
	Trap
7	Floating-point assist exception detected by floating-point unit, or by load/store unit during a store
8	Machine check during load or store
9	Floating-point assist exception detected by load/store unit during a load
10	L-bus breakpoint

## NOTES:

1. The trace mechanism is implemented by letting one instruction complete as if no trace were enabled and then trapping the second instruction. Trace has the highest priority of exceptions associated with this second instruction.
2. All of these cases are mutually exclusive for any one instruction.

## 6.8 Exception Processing

When an exception is taken, the processor uses the save/restore registers, SRR0 and SRR1, to save the contents of the machine state register and to identify where instruction execution should resume after the exception is handled.

When an exception occurs, SRR0 is set to point to the instruction at which instruction processing should resume when the exception handler returns control to the interrupted process. All instructions in the program flow preceding this one will have completed, and no subsequent instruction will have completed. The address may be of the instruction that caused the exception or of the next one (as in the case of a system call exception, for example). The instruction addressed can be determined from the exception type and status bits.

SRR1 is a 32-bit register used to save machine status (the contents of the MSR) on exceptions and to restore machine status when **rfi** is executed.

The data address register (DAR) is a 32-bit register used by alignment exceptions to identify the address of a memory element.

# Freescale Semiconductor, Inc.

When an exception occurs, SRR1[0:15] are loaded with exception-specific information and bits SRR1[16:31] are loaded with the corresponding bits of the MSR. The machine state register is shown below.

## MSR — Machine State Register

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

RESET:

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

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
EE	PR	FP	ME	FE0	SE	BE	FE1	0	IP	IR	DR	0		RI	LE

RESET:

0 0 0 U 0 0 0 0 0 \* 0 0 0 0 0 0

\*Reset value of this bit on value of internal data bus configuration word at reset. Refer to the *System Interface Unit Reference Manual* (SIURM/AD).

**Table 6-7** shows the bit definitions for the MSR.

## Table 6-7 Machine State Register Bit Settings

Bit(s)	Name	Description
[0:14]	—	Reserved
15	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. 0 Processor runs in big-endian mode during exception processing. 1 Processor runs in little-endian mode during exception processing.
16	EE	External interrupt enable 0 The processor delays recognition of external interrupts and decremter exception conditions. 1 The processor is enabled to take an external interrupt or the decremter exception.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores and moves. Floating-point enabled program exceptions can still occur and the FPRs can still be accessed. 1 The processor can execute floating-point instructions, and can take floating-point enabled exception type program exceptions.
19	ME	Machine check enable 0 Machine check exceptions are disabled. 1 Machine check exceptions are enabled.
20	FE0	Floating-point exception mode 0 (See <a href="#">Table 6-8.</a> )
21	SE	Single-step trace enable 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of the next instruction. When this bit is set, the processor dispatches instructions in strict program order. Successful execution means the instruction caused no other exception. Single-step tracing may not be present on all implementations.
22	BE	Branch trace enable
23	FE1	Floating-point exception mode 1 (See <a href="#">Table 6-8.</a> )
24	—	Reserved
25	IP	Exception prefix. The setting of this bit determines the location of the exception vector table. 0 Exceptions are vectored to the physical address 0x0000 0000 plus vector offset. 1 Exceptions are vectored to the physical address 0xFFFF0 0000 plus vector offset.
[26:29]	—	Reserved
30	RI	Recoverable exception 0 Exception is not recoverable. 1 Exception is recoverable.
31	LE	Little-endian mode 0 Processor operates in big-endian mode during normal processing. 1 Processor operates in little-endian mode during normal processing.

**Table 6-8 Floating-Point Exception Mode Bits**

FE[0:1]	Mode
00	Floating-point exceptions disabled
01, 10, 11	Floating-point precise mode

MSR[16:31] are guaranteed to be written to SRR1 when the first instruction of the exception handler is encountered.

### 6.8.1 Enabling and Disabling Exceptions

When a condition exists that causes an exception to be generated, the processor must determine whether the exception is enabled for that condition.

- Floating-point enabled exceptions (a type of program exception) can be disabled by clearing both MSR[FE0] and MSR[FE1]. If either or both of these bits are set, all floating-point exceptions are taken and cause a program exception. Bits in the FPSCR can enable and disable individual conditions that can generate floating-point exceptions.
- External and decrements interrupts are enabled by setting the MSR[EE] bit. When MSR[EE] = 0, recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an exception is taken to delay recognition of conditions causing those exceptions.
- A machine check exception can only occur if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine-check exception condition occurs.
- System reset and non-maskable external breakpoint exceptions cannot be masked.
- Internal data and instruction breakpoints are specified as maskable or non-maskable by the BRKNOMSK bit in LCTRL2.
- Maskable internal (data and instruction) and external breakpoints are recognized only when MSR[RI] = 1.

### 6.8.2 Steps for Exception Processing

After determining that the exception can be taken (by confirming that any instruction-caused exceptions occurring earlier in the instruction stream have been handled, and by confirming that the exception is enabled for the exception condition), the processor does the following:

1. Loads the machine status save/restore register 0 (SRR0) with an instruction address that depends on the type of exception. See the individual exception description for details about how this register is used for specific exceptions.
2. Loads SRR1[0:15] with information specific to the exception type.
3. Loads SRR1[16:31] with a copy of MSR[16:31].
4. Sets the MSR as described in [Table 6-9](#). The new values take effect beginning with the fetching of the first instruction of the exception-handler routine located at the exception vector address.

5. Resumes fetching and executing instructions, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector (see [Table 6-3](#)) to the base address determined by MSR[IP]. If IP is cleared, exceptions are vectored beginning at the physical address 0x0000 0000. If IP is set, exceptions are vectored beginning at 0xFFFF0 0000. For a machine check exception that occurs when MSR[ME] = 0 (machine check exceptions are disabled), the checkstop state is entered (the machine stops executing instructions).
6. The **lwarx** and **stwx** instructions require special handling if a reservation is still set when an exception occurs.

**Table 6-9** shows the MSR bit settings when the processor changes to supervisor mode.

**Table 6-9 MSR Setting Due to Exception**

Exception Type	MSR Bit										
	EE 16	PR 17	FP 18	ME 19	FE0 20	SE 21	BE 22	FE1 23	IP 25	RI 30	LE 31
Reset	0	0	0	— <sup>1</sup>	0	0	0	0	† <sup>2</sup>	0	0
All others	0	0	0	—	0	0	0	0	—	0	† <sup>3</sup>

NOTES:

1. — indicates that bit is not altered.
2. Depends on value of internal data bus configuration word at reset.
3. Contains value of MSR[ILE] prior to exception.

### 6.8.3 DAR, DSISR, and BAR Operation

The load/store unit keeps track of all instructions and bus cycles. In case of a bus error, the data address register (DAR) is loaded with the effective address (EA) of the cycle. In case of a multi-cycle instruction, the effective address of the first of-fending cycle is loaded.

The data storage and interrupt status register (DSISR) identifies the cause of the error in case of an exception caused by a load or a store. The DSISR is loaded with the instruction information as described in [6.11.4 Alignment Exception \(0x00600\)](#).

The breakpoint address register (BAR) indicates the address on which an L-bus breakpoint occurs. For multi-cycle instructions, the BAR contains the address of the first cycle associated with the breakpoint. The BAR has a valid value only when a data breakpoint exception is taken; at all other times its value is boundedly unde-fined.

**Table 6-10** summarizes the values in DAR, BAR, and DSISR following an excep-tion.

Table 6-10 DAR, BAR, and DSISR Values in Exception Processing

Exception Type	DAR Value	DSISR Value	BAR Value
Alignment exception	Data EA	Instruction information	Undefined
L-bus breakpoint exception	Unchanged	Unchanged	Cycle EA
Floating-Point Assist Exception	Unchanged	Unchanged	Undefined
Machine-check exception	Cycle EA	Instruction information	Undefined
Implementation-dependent software emulation exception	Unchanged	Unchanged	Undefined
Floating-point unavailable exception	Unchanged	Unchanged	Undefined
Program exception	Unchanged	Unchanged	Unchanged

#### 6.8.4 Returning from Supervisor Mode

The return from interrupt (**rfi**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to user mode. Execution of the **rfi** instruction ensures the following:

- All previous instructions have been retired.
- Previous instructions complete execution in the context (privilege and protection) under which they were issued.
- The instructions following this instruction execute in the context established by this instruction.

#### 6.9 Process Switching

The operating system should execute the following when processes are switched:

- The **sync** instruction, to resolve any data dependencies between the processes and to synchronize the use of SPRs.
- The **isync** instruction, to ensure that undispatched instructions not in the new process are not used by the new process.
- The **stwcx.** instruction, to clear any outstanding reservations, which ensures that an **lwarx** instruction in the old process is not paired with an **stwcx.** in the new process.

Note that if an exception handler is used to emulate an instruction that is not implemented, the exception handler must report in SRR0 the EA computed by the instruction being emulated and not one used to emulate the instruction.

#### 6.10 Exception Timing

Table 6-11 illustrates the significant events in exception processing.



Table 6-11 Exception Latency

Time	Fetch	Issue	Instruction Complete	Kill Pipeline
A		Faulting instruction issue		
B			Instruction complete and all previous instructions complete	
				Kill pipeline
C	Start fetch handler			
$D \leq B + 3$ clocks				
E		1st instruction of handler issued		

At time-point A the excepting instruction issues and begins execution. During the interval A-B previously issued instructions are finishing execution. The interval A-B is equivalent to the time required for all instructions currently in progress to complete, (i.e., the time to serialize the machine).

At time-point B the excepting instruction has reached the head of the history queue, implying that all instructions preceding it in the code stream have finished execution without generating any exception. In addition, the excepting instruction itself has completed execution. At this time the exception is recognized, and exception processing begins. If at this point the instruction had not generated an exception, it would have been retired.

During the interval B-D the machine state is being restored. This can take up to three clock cycles.

At time-point C the processor starts fetching the first instruction of the exception handler.

By time-point D the state of the machine prior to the issue of the excepting instruction has been restored. During interval D-E, the machine is saving context information in SRR0 and SRR1, disabling interrupts, placing the machine in privileged mode, and may continue the process of fetching the first instructions of the interrupt handler from the vector table.

At time-point E the MSR and instruction pointer of the executing process have been saved and control has been transferred to the exception handler routine.

The interval D-E requires a minimum of one clock cycle. The interval C-E depends on the memory system. This interval is the time it takes to fetch the first instruction of the exception handler. For a full history buffer, it is no less than two clocks.

## 6.11 Exception Definitions

The following paragraphs describe each type of exception supported by the RCPU.

### 6.11.1 Reset Exception (0x0100)

The reset exception is a non-maskable, asynchronous exception signaled to the processor by the assertion of the internal reset input signal (RESET). The system interface unit asserts this signal in response to either the assertion of the external  $\overline{\text{RESET}}$  pin or an internal reset request, such as from the software watchdog timer. Refer to the *System Interface Unit Reference Manual* (SIURM/AD) for a description of sources within the SIU that can cause the RESET input to the processor to be asserted.

A reset operation should be performed on power-on to appropriately reset the processor. The assertion of RESET causes the reset exception to be taken. The physical address of the handler is 0xFFFF 0100 or 0x0000 0100, depending on the value of the internal data bus configuration word during reset. Refer to the *System Interface Unit Reference Manual* (SIURM/AD) for additional information on the data bus configuration word and system configuration during reset.

**Table 6-12** shows the state of the machine just before it fetches the first instruction after reset. Registers not listed are not affected by reset.

**Table 6-12 Settings Caused by Reset**

Register	Setting
MSR	IP depends on internal data bus configuration word ME is unchanged All other bits are cleared
SRR0	Undefined
SRR1	Undefined
FPECR	0x0000 0000
ICTRL	0x0000 0000
LCTRL1	0x0000 0000
LCTRL2	0x0000 0000
COUNTA[16:31]	0x0000 0000
COUNTB[16:31]	0x0000 0000
DMCR	0x0000 0000
DMMR[2,4,28:31]	Set to one
ICCST	0x0000 0000
ICADR, ICDAT	Undefined

### 6.11.2 Machine Check Exception (0x00200)

The processor conditionally initiates a machine-check exception after detecting the assertion of the TEA signal, indicating the occurrence of a bus error. The TEA signal can be asserted either externally (by an external device asserting the  $\overline{\text{TEA}}$  pin), or internally by the SIU chip-select logic. The processor receives notification of the exception from either the I-bus (if the exception is caused during the instruction phase) or the L-bus (if the exception is caused during the data phase).

Machine check exceptions are unordered. The machine-state exception handler must read the SRR1[RI] bit to determine whether the processor can recover from a machine-check exception. For additional information, refer to [6.5.2 Recovery from Unordered Exceptions](#).

A machine-check exception is assumed to be caused by one of the following conditions:

- The accessed address does not exist.
- A data error was detected.
- A storage protection violation was detected by chip-select logic (either on-chip or external).

When a machine-check exception occurs, the processor does one of the following:

- Takes a machine check exception;
- Enters the checkstop state; or
- Enters debug mode.

Which action is taken depends on the value of the MSR[ME] bit, whether or not debug mode was enabled at reset, and (if debug mode is enabled) the values of the CHSTPE (checkstop enable) and MCIE (machine check enable) bits in the debug enable register (DER). [Table 6-13](#) summarizes the possibilities.

**Table 6-13 Machine Check Exception Processor Actions**

MSR[ME]	Debug Mode Enable	CHSTPE	MCIE	Action Performed when Exception Detected
0	0	X	X	Enter checkstop state
1	0	X	X	Branch to machine-check exception handler
0	1	0	X	Enter checkstop state
0	1	1	X	Enter debug mode
1	1	X	0	Branch to machine-check exception handler
1	1	X	1	Enter debug mode

#### 6.11.2.1 Machine Check Exception Enabled

A machine check exception is taken when MSR[ME] is set and either debug mode is disabled or DER[MCIE] is cleared. When a machine check exception is taken, registers are updated as shown in [Table 6-14](#).

**Table 6-14 Register Settings Following a Machine Check Exception**

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the interrupt.	
SRR1	0	Cleared
	1	Set for instruction-fetch related errors, cleared for load-store related errors
	[2:15]	Cleared
	[16:31]	Loaded from MSR[16:31].
MSR	IP	No change
	ME	Cleared to zero
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared
DSISR (L-bus case only)	[15:16]	Set to bits [29:30] of the instruction if X-form Set to 0b00 if D-form
	17	Set to bit 25 of the instruction if X-form Set to bit 5 of the instruction if D-form
	[22:31]	Set to bits [6:15] of the instruction
DAR (L-bus case only)	Set to the effective address of the data access that caused the exception.	

When a machine check exception is taken, instruction execution resumes at offset 0x00200 from the physical base address indicated by MSR[IP].

### 6.11.2.2 Checkstop State

The processor enters the checkstop state when a machine check exception occurs, MSR[ME] equals zero, and either debug mode is disabled or DER[CHSTPE] is cleared. When the processor is in the checkstop state, instruction processing is suspended and generally cannot be restarted without resetting the processor. The contents of all latches (except any associated with the bus clock) are frozen within two cycles upon entering checkstop state so that the state of the processor can be analyzed.

### 6.11.2.3 Machine-Check Exceptions and Debug Mode

The processor enters debug mode when a machine check exception occurs, debug mode is enabled, and either MSR[ME] = 0 and DER[CHSTPE] = 1, or MSR[ME] = 1 and DER[MCIE] = 1. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information.

### 6.11.3 External Interrupt (0x00500)

The interrupt controller in the on-chip peripheral control unit signals an external interrupt by asserting the  $\overline{\text{IRQ}}$  input to the processor. The interrupt may be caused by the assertion of an external IRQ pin, by the periodic interrupt timer, or by an on-chip peripheral. Refer to *System Interface Unit Reference Manual (SIURM/AD)* for more information on the interrupt controller.

The interrupt may be delayed by other higher priority exceptions or if the MSR[EE]

## Freescale Semiconductor, Inc.

bit is cleared when the exception occurs. MSR[EE] is automatically cleared by hardware to disable external interrupts when any exception is taken.

Upon detecting an external interrupt, the processor assigns it to the instruction at the head of the history buffer (after retiring all instructions that are ready to retire). Refer to [6.4 Implementation of Asynchronous Exceptions](#) for more information.

The register settings for the external interrupt exception are shown in [Table 6-15](#).

**Table 6-15 Register Settings Following External Interrupt**

Register	Setting Description	
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.	
SRR1	[0:15]	Cleared
	[16:31]	Loaded from bits [16:31] of the MSR
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared

When an external interrupt is taken, instruction execution resumes at offset 0x00500 from the physical base address indicated by MSR[IP].

### 6.11.4 Alignment Exception (0x00600)

The following conditions cause an alignment exception:

- The operand of a floating-point load or store instruction is not word-aligned.
- The operand of a load or store multiple instruction is not word-aligned.
- The operand of **lwarx** or **stwcx.** is not word-aligned.
- The operand of a load or store instruction is not naturally aligned, and MSR[LE] = 1 (little-endian mode).
- The processor attempts to execute a multiple or string instruction when MSR[LE] = 1 (little-endian mode).

Alignment exceptions use the SRR0 and SRR1 to save the machine state and the DSISR to determine the source of the exception.

The register settings for alignment exceptions are shown in [Table 6-16](#).

Table 6-16 Register Settings for Alignment Exception

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception.	
SRR1	[0:15] [16:31]	Cleared Loaded from MSR[16:31]
MSR	IP ME LE Other bits	No change No change Set to value of ILE bit prior to the exception Cleared
DSISR	[0:11] [12:13] 14 [15:16]  17  [18:21]  [22:26] [27:31]	Cleared Cleared Cleared For instructions that use register indirect with index addressing, set to bits [29:30] of the instruction. For instructions that use register indirect with immediate index addressing, cleared. For instructions that use register indirect with index addressing, set to bit 25 of the instruction. For instructions that use register indirect with immediate index addressing, set to bit 5 of the instruction. For instructions that use register indirect with index addressing, set to bits [21:24] of the instruction. For instructions that use register indirect with immediate index addressing, set to bits [1:4] of the instruction. Set to bits [6:10] (source or destination) of the instruction. Set to bits [11:15] of the instruction (rA). Set to either bits [11:15] of the instruction or to any register number not in the range of registers loaded by a valid form instruction, for <b>lmw</b> , <b>lswi</b> , and <b>lswx</b> instructions. Otherwise undefined.  Note that for load or store instructions that use register indirect with index addressing, the DSISR can be set to the same value that would have resulted if the corresponding instruction uses register indirect with immediate index addressing had caused the exception. Similarly, for load or store instructions that use register indirect with immediate index addressing, DSISR can hold a value that would have resulted from an instruction that uses register indirect with index addressing. (If there is no corresponding instruction, no alternative value can be specified.)

When an alignment exception is taken, instruction execution resumes at offset 0x00600 from the physical base address indicated by MSR[IP].

#### 6.11.4.1 Interpretation of the DSISR as Set by an Alignment Exception

For most alignment exceptions, an exception handler may be designed to emulate the instruction that causes the exception. To do this, it needs the following characteristics of the instruction:

- Load or store
- Length (half word, word, or double word)
- String, multiple, or normal load/store

- Integer or floating-point
- Whether the instruction performs update
- Whether the instruction performs byte reversal

The PowerPC architecture provides this information implicitly, by setting opcode bits in the DSISR that identify the excepting instruction type. The exception handler does not need to load the excepting instruction from memory. The mapping for all exception possibilities is unique except for the few exceptions discussed below.

**Table 6-17** shows how the DSISR bits identify the instruction that caused the exception.

**Table 6-17 DSISR[15:21] Settings**

DSISR[15:21]	Instruction
00 0 0000	lwarx, lwz, reserved
00 0 0010	stw
00 0 0100	lhz
00 0 0101	lha
00 0 0110	sth
00 0 0111	lmw
00 0 1000	lfs
00 0 1001	lfd
00 0 1010	stfs
00 0 1011	stfd
00 1 0000	lwzu
00 1 0010	stwu
00 1 0100	lhzu
00 1 0101	lhau
00 1 0110	sthu
00 1 0111	stmw
00 1 1000	lfsu
00 1 1001	lfdu
00 1 1010	stfsu
00 1 1011	stfdu
01 0 1000	lswx
01 0 1001	lswi
01 0 1010	stswx

Table 6-17 DSISR[15:21] Settings (Continued)

DSISR[15:21]	Instruction
01 0 1011	stswi
01 1 0101	lwaux
10 0 0010	stwcx.
10 0 1000	lwbrx
10 0 1010	stwbrx
10 0 1100	lhbrx
10 0 1110	sthbrx
11 0 0000	lwzx
11 0 0010	stwx
11 0 0100	lhzx
11 0 0101	lhax
11 0 0110	sthx
11 0 1000	lfsx
11 0 1001	lfdx
11 0 1010	stfsx
11 0 1011	stfdx
11 1 0000	lwzux
11 1 0010	stwux
11 1 0100	lhzux
11 1 0101	lhaux
11 1 0110	sthux
11 1 1000	lfsux
11 1 1001	lfdux
11 1 1010	stfsux
11 1 1011	stfdux

## NOTES:

1. The instructions **lwz** and **lwarx** give the same DSISR bits (all zero). But if **lwarx** causes an alignment exception, it is an invalid form, so it need not be emulated in any precise way. It is adequate for the alignment exception handler to simply emulate the instruction as if it were an **lwz**. It is important that the emulator use the address in the DAR, rather than computing it from rA/rB/D, because **lwz** and **lwarx** use different addressing modes.

## 6.11.5 Program Exception (0x00700)

A program exception occurs when no higher priority exception exists and one or more of the following exception conditions, which correspond to bit settings in SRR1, occur during execution of an instruction:



## Freescale Semiconductor, Inc.

- System floating-point enabled exception — A system floating-point enabled exception is generated when the following condition is met as a result of a move to FPSCR instruction, move to MSR (**mtmsr**) instruction, or return from interrupt (**rfi**) instruction:

$$(\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) \& \text{FPSCR}[\text{FEX}] = 1.$$

Notice that in the RCPU implementation of the PowerPC architecture, a program interrupt is not generated by a floating-point arithmetic instruction that results in the condition shown above; a floating-point assist exception is generated instead.

- Privileged instruction — A privileged instruction type program exception is generated by any of the following conditions:
  - The execution of a privileged instruction (**mfmsr**, **mtmsr**, or **rfi**) is attempted and the processor is operating at the user privilege level ( $\text{MSR}[\text{PR}] = 1$ ).
  - The execution of **mtspr** or **mfmspr** where  $\text{SPR0} = 1$  in the instruction encoding (indicating a supervisor-access register) and  $\text{MSR}[\text{PR}] = 1$  (indicating the processor is operating at the user privilege level), provided the SPR instruction field encoding represents either:
    - a valid internal-to-the-processor special-purpose register; or
    - an external-to-the-processor special-purpose register (either valid or invalid).
  - Refer to **7.5 Implementation of Special-Purpose Registers** for a discussion of internal- and external-to-the-processor SPRs.
- Trap — A trap type program exception is generated when any of the conditions specified in a trap instruction is met. Trap instructions are described in **SECTION 4 ADDRESSING MODES AND INSTRUCTION SET SUMMARY**.

Notice that, in contrast to some other PowerPC processors, the RCPU generates a software emulation exception, rather than a program exception, when an attempt is made to execute any unimplemented instruction. This includes all illegal instructions and optional instructions not implemented in the RCPU.

The register settings are shown in **Table 6-18**.

**Table 6-18 Register Settings Following Program Exception**

Register	Setting Description	
SRR0	Contains the effective address of the excepting instruction	
SRR1	[0:10]	Cleared
	11	Set for a floating-point enabled program exception; otherwise cleared.
	12	Cleared.
	13	Set for a privileged instruction program exception; otherwise cleared.
	14	Set for a trap program exception; otherwise cleared.
	15	Cleared if SRR0 contains the address of the instruction causing the exception, and set if SRR0 contains the address of a subsequent instruction.
	[16:31]	Loaded from MSR[16:31].
	Note that only one of bits 11, 13, and 14 can be set.	
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared to zero

When a program exception is taken, instruction execution resumes at offset 0x00700 from the physical base address indicated by MSR[IP].

#### 6.11.6 Floating-Point Unavailable Exception (0x00800)

A floating-point unavailable exception occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, and move instructions), and the floating-point available bit in the MSR is disabled, (MSR[FP] = 0).

The register settings for floating-point unavailable exceptions are shown in [Table 6-19](#).

**Table 6-19 Register Settings Following a Floating-Point Unavailable Exception**

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception.	
SRR1	[0:15]	Cleared
	[16:31]	Loaded from MSR[16:31]
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared

When a floating-point unavailable exception is taken, instruction execution resumes at offset 0x00800 from the physical base address indicated by MSR[IP].

### 6.11.7 Decrementer Exception (0x00900)

A decrementer exception occurs when no higher priority exception exists, the decrementer register has completed decrementing, and MSR[EE] = 1. The decrementer exception request is canceled when the exception is handled. The decrementer register counts down, causing an exception (unless masked) when passing through zero. The decrementer implementation meets the following requirements:

- Loading a GPR from the decrementer does not affect the decrementer.
- Storing a GPR value to the decrementer replaces the value in the decrementer with the value in the GPR.
- Whenever bit 0 of the decrementer changes from zero to one, an exception request is signaled. If multiple decrementer exception requests are received before the first can be reported, only one exception is reported. The occurrence of a decrementer exception cancels the request.
- If the decrementer is altered by software and if bit 0 is changed from zero to one, an interrupt request is signaled.

The register settings for the decrementer exception are shown in [Table 6-20](#).

**Table 6-20 Register Settings Following a Decrementer Exception**

Register	Setting Description	
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.	
SRR1	[0:15]	Cleared
	[16:31]	Loaded from MSR[16:31]
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared to zero

When a decrementer exception is taken, instruction execution resumes at offset 0x00900 from the physical base address indicated by MSR[IP].

### 6.11.8 System Call Exception (0x00C00)

A system call exception occurs when a system call instruction is executed. The effective address of the instruction following the **sc** instruction is placed into SRR0. MSR[16:31] are placed into SRR1[16:31], and SRR1[0:15] are set to undefined values. Then a system call exception is generated.

The system call instruction is context synchronizing. That is, when a system call exception occurs, instruction dispatch is halted and the following synchronization is performed:

1. The exception mechanism waits for all instructions in execution to complete to a point where they report all exceptions they will cause.
2. The processor ensures that all instructions in execution complete in the con-

# Freescale Semiconductor, Inc.

text in which they began execution.

- Instructions dispatched after the exception is processed are fetched and executed in the context established by the exception mechanism.

Register settings are shown in [Table 6-22](#).

**Table 6-21 Register Settings Following a System Call Exception**

Register	Setting Description	
SRR0	Set to the effective address of the instruction following the System Call instruction	
SRR1	[0:15]	Undefined
	[16:31]	Loaded from MSR[16:31]
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared to zero

When a system call exception is taken, instruction execution resumes at offset 0x00C00 from the physical base address indicated by MSR[IP].

## 6.11.9 Trace Exception (0x00D00)

A trace exception occurs if MSR[SE] = 1 and any instruction other than **rfi** is successfully completed, or if MSR[BE] = 1 and a branch is completed. Notice that the trace exception does not occur after an instruction that causes an exception.

A monitor or debugger software needs to change the vectors of other possible exception addresses in order to single-step such instructions. If this is not desirable, other debugging features can be used. Refer to [SECTION 8 DEVELOPMENT SUPPORT](#) for more information.

Register settings are shown in [Table 6-22](#).

**Table 6-22 Register Settings Following a Trace Exception**

Register	Setting Description	
SRR0	Set to the effective address of the instruction following the executed instruction	
SRR1	[0:15]	Cleared to zero
	[16:31]	Loaded from MSR[16:31]
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared to zero

When a trace exception is taken, execution resumes at offset 0x00D00 from the base address indicated by MSR[IP].

#### 6.11.10 Floating-Point Assist Exception (0x00E00)

A floating point assist exception occurs in the following cases:

- When the following conditions are true:
  - A floating-point enabled exception condition is detected;
  - the corresponding floating-point enable bit in the FPSCR (floating point status and control register) is set (exception enabled); and
  - $\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}] = 1$

These conditions are summarized in the following equation:

$$(\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) \& \text{FPSCR}[\text{FEX}] = 1$$

Note that when  $((\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) \& \text{FPSCR}[\text{FEX}])$  is set as a result of move to FPSCR, move to MSR or **rfi**, a program exception is generated, rather than a floating-point assist exception.

- When a tiny result is detected and the floating point underflow exception is disabled ( $\text{FPSCR}[\text{UE}] = 0$ )
- In some cases when at least one of the source operands is denormalized (refer to [6.11.10.2 Floating-Point Assist for Denormalized Operands](#))

The register settings for floating-point assist exceptions are shown in [Table 6-22](#). In addition, floating-point enabled exceptions affect the FPSCR as shown in [Table 6-26](#).

**Table 6-23 Register Settings Following a Floating-Point Assist Exception**

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception	
SRR1	[0:15]	Cleared to zero
	[16:31]	Loaded from bits [16:31] of the MSR
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared to zero

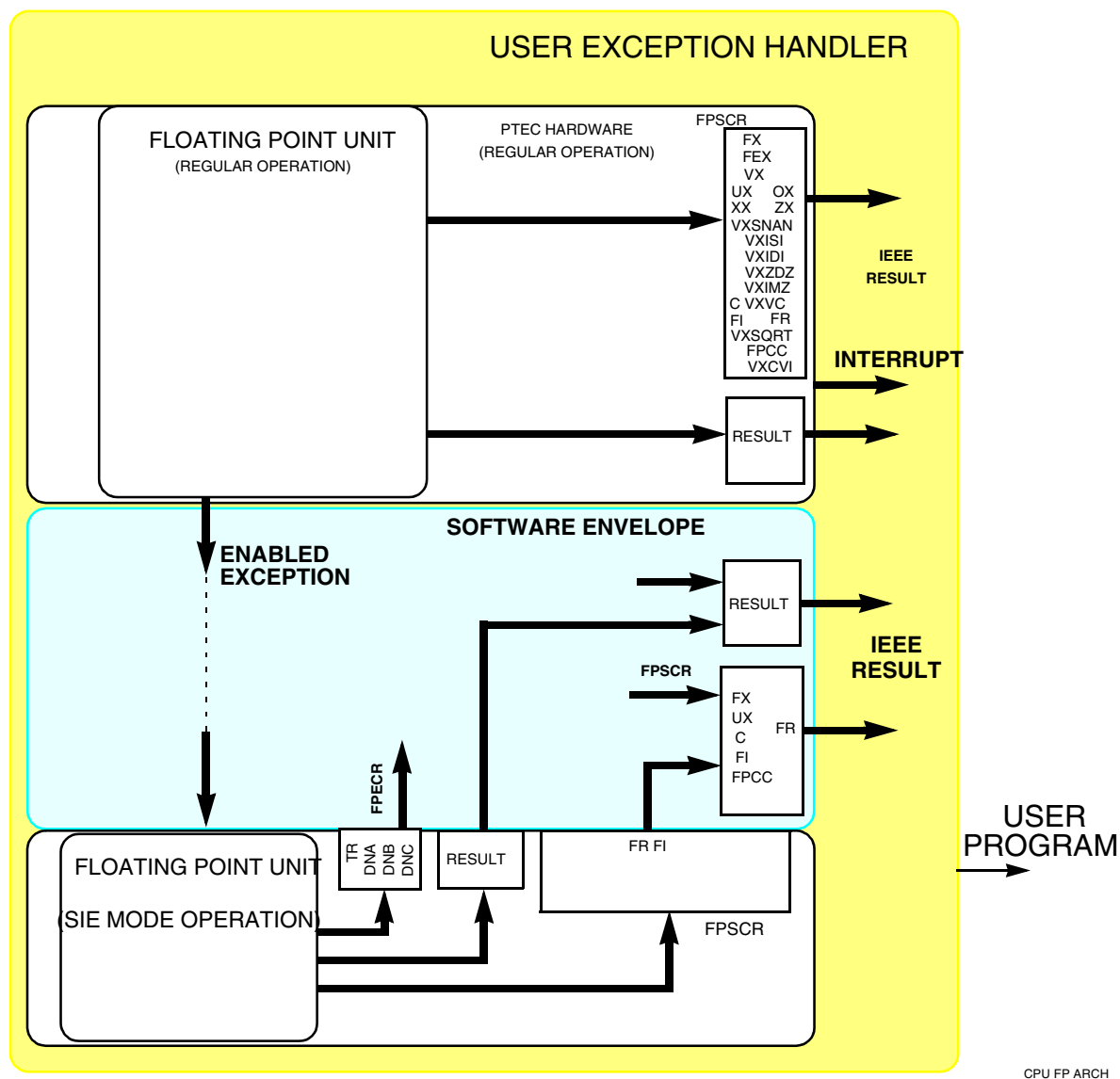
When a floating-point assist exception is taken, execution resumes at offset 0x00E00 from the base address indicated by  $\text{MSR}[\text{IP}]$ .

##### 6.11.10.1 Floating-Point Software Envelope

The floating-point assist software envelope is an exception handler for floating-point assist exceptions. Use of this exception handler guarantees that results of floating-point operations are in compliance with IEEE standards.

In most cases, floating-point operations are implemented in hardware in the RCPU. For cases in which the hardware needs software assistance, the software envelope emulates the instruction using a special synchronized ignore exceptions (SIE) hardware mode. This mode is useful only for emulating an instruction executed in the floating-point unit, not an instruction executed by the load/store unit. SIE mode is described in [6.11.10.3 Synchronized Ignore Exceptions \(SIE\) Mode](#).

Execution of floating-point instructions is illustrated in **Figure 6-2**. This process shows the execution of all floating-point instructions except the floating-point move to FPSCR type instructions.

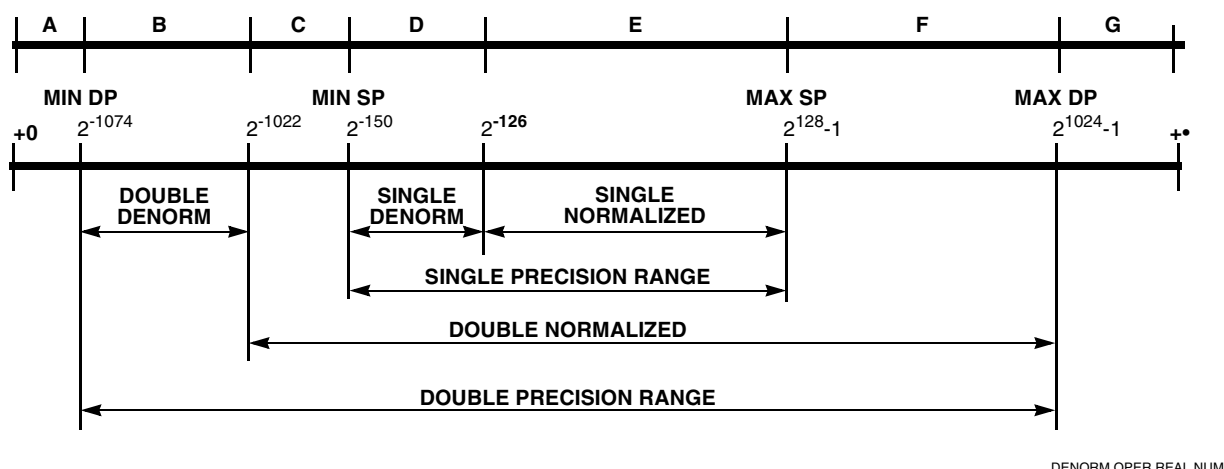


**Figure 6-2 RCPU Floating-Point Architecture**

#### 6.11.10.2 Floating-Point Assist for Denormalized Operands

When a denormalized operand is detected there are some cases in which the processor needs the assistance of the software to perform the operation. In these cases the software envelope is invoked. **Table 6-24** summarizes the hardware/software partitioning in handling denormalized operands in the input stage of the execution units. The ranges referred to in the table are defined in **Figure 6-3**.

# Freescale Semiconductor, Inc.



**Figure 6-3 Real Numbers Axis for Denormalized Operands**

**Table 6-24 Software/Hardware Partitioning in Operands Treatment**

Instruction	Range B	Range C	Range D	Range E	Range F
Load single	NA	NA	Floating-Point Assist	Hardware	NA
Load double	Hardware	Hardware	Hardware	Hardware	Hardware
Store single	Programming error <sup>1</sup>	Programming error <sup>1</sup>	Floating-Point Assist	Hardware <sup>2</sup>	Programming error
Store double	Hardware	Hardware	Hardware	Hardware	Hardware
FP arithmetic & Multiply-add single	Programming error <sup>1</sup>	Programming error <sup>1</sup>	Hardware <sup>2</sup>	Hardware <sup>2</sup>	Programming error <sup>1</sup>
FP arithmetic & Multiply-add double	Floating-Point Assist	Hardware	Hardware	Hardware	Hardware
Round to single	Floating-Point Assist	Hardware <sup>3</sup>	Hardware <sup>3</sup>	Hardware	Hardware
FP compare, FP move, convert to integer & FPSCR instr.	Hardware	Hardware	Hardware	Hardware	Hardware

**NOTES:**

1. The results in all cases of programming errors are boundedly undefined.
2. When used by a single precision instruction, generates correct result only if bits [35:63] of the operand equal zero, otherwise it is a programming error.
3. Since the result is tiny, a floating-point assist exception is taken at the end of instruction execution.

## 6.11.10.3 Synchronized Ignore Exceptions (SIE) Mode

The software envelope uses SIE mode to emulate instructions executed by the floating-point unit. (This mode is not used to emulate floating-point instructions executed by the load/store unit.) In SIE mode the floating-point unit does the following:

- Re-executes the instruction (without generating a floating-point assist exception a second time)
- Generates default results in hardware
- Updates the FPSCR
- Updates the floating-point exceptions cause register (FPECR).

The FPECR is a special-purpose register used by the software envelope. It contains four status bits indicating whether the result of the operation is tiny and whether any of three source operands are denormalized. In addition, it contains one control bit to enable or disable SIE mode. This register must not be accessed by user code. Refer to [6.11.10.4 Floating-Point Exception Cause Register](#) for more information.

If as a result of the operation performed in SIE mode,  $((\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) \& \text{FPSCR}[\text{FEX}])$  is set, a program exception is taken. It is the responsibility of the software envelope to make sure that when executing an instruction in SIE mode  $((\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) = 0)$ .

Except when the result is tiny or when denormalized operands are detected, the results generated by the hardware in SIE mode are practically all that is needed in order to complete the operation according to the IEEE standard. Therefore, in most cases after executing the instruction in SIE mode all that is needed by the software is to issue **rfi**. Upon execution of the **rfi**, the hardware restores the previous value of the MSR, as it was saved in SRR1. If as a result  $((\text{MSR}[\text{FE0}] \mid \text{MSR}[\text{FE1}]) \& \text{FPSCR}[\text{FEX}])$  is set, a program exception is generated.

When the result is tiny and the floating-point underflow exception is disabled ( $\text{FPSCR}[\text{UE}] = 0$ ), the hardware in SIE mode delivers the same result as when the exception is enabled ( $\text{FPSCR}[\text{UE}] = 1$ ), (i.e., rounded mantissa with exponent adjusted by adding 192 for single precision or 1536 for double precision). This intermediate result simplifies the task of the emulation routine that finishes the instruction execution and delivers the correct IEEE result. In this case the software envelope is responsible for updating the floating-point underflow exception bit ( $\text{FPSCR}[\text{UX}]$ ) as well.

When at least one of the source operands is denormalized and the hardware can not complete the operation, the destination register value is unchanged. In this case, the software emulation routine must execute the instruction in software, deliver the result to the destination register, and update the FPSCR.

## 6.11.10.4 Floating-Point Exception Cause Register

The FPECR is a special-purpose register used by the software envelope. It con-



# Freescale Semiconductor, Inc.

tains four status bits indicating whether the result of the operation is tiny and whether any of three source operands are denormalized. In addition, it contains one control bit to enable or disable SIE mode. This register must not be accessed by user code.

## FPECR — Floating-Point Exception Cause Register

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

RESET:

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

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED												DNC	DNB	DNA	TR

RESET:

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

A listing of FPECR bit settings is shown in [Table 6-26](#).

**Table 6-25 FPECR Bit Settings**

Bit(s)	Name	Description
0	SIE	SIE mode control bit 0 Disable SIE mode 1 Enable SIE mode
[1:27]	—	Reserved
28	DNC	Source operand C denormalized status bit 0 Source operand C is not denormalized 1 Source operand C is denormalized
29	DNB	Source operand B denormalized status bit 0 Source operand B is not denormalized 1 Source operand B is denormalized
30	DNA	Source operand A denormalized status bit 0 Source operand A is not denormalized 1 Source operand A is denormalized
31	TR	Floating-point tiny result 0 Floating-point result is not tiny 1 Floating-point result is tiny

### NOTE

Software must insert a **sync** instruction before reading the FPECR.

### 6.11.10.5 Floating-Point Enabled Exceptions

Floating-point exceptions are signaled by condition bits set in the floating-point status and control register (FPSCR). They can cause the system floating-point enabled exception error handler to be invoked. All floating-point exceptions are handled precisely. The FPSCR is shown below.

#### FPSCR — Floating-Point Status and Control Register

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FX	FEX	VX	OX	UX	ZX	XX	VXS-NAN	VXISI	VXIDI	VXZDZ	VXIMZ	VXVC	FR	FI	FPRF0

RESET:

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

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
FPRF[16:19]				0	VX-SOFT	VX-SQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN	

RESET:

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

A listing of FPSCR bit settings is shown in [Table 6-26](#).

**Table 6-26 FPSCR Bit Settings**

Bit(s)	Name	Description
0	FX	Floating-point exception summary (FX). Every floating-point instruction implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to change from zero to one. The <b>mcrfs</b> instruction implicitly clears FPSCR[FX] if the FPSCR field containing FPSCR[FX] is copied. The <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions can set or clear FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary (FEX). This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked with their respective enable bits. The <b>mcrfs</b> instruction implicitly clears FPSCR[FEX] if the result of the logical OR described above becomes zero. The <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions cannot set or clear FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary (VX). This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exceptions. The <b>mcrfs</b> instruction implicitly clears FPSCR[VX] if the result of the logical OR described above becomes zero. The <b>mtfsf</b> , <b>mtfsfi</b> , <b>mtfsb0</b> , and <b>mtfsb1</b> instructions cannot set or clear FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception (OX). This is a sticky bit. See <a href="#">6.11.10.8 Overflow Exception Condition</a> .
4	UX	Floating-point underflow exception (UX). This is a sticky bit. See <a href="#">6.11.10.9 Underflow Exception Condition</a> .

## Table 6-26 FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
5	ZX	Floating-point zero divide exception (ZX). This is a sticky bit. See <a href="#">6.11.10.7 Zero Divide Exception Condition</a> .
6	XX	Floating-point inexact exception (XX). This is a sticky bit. See <a href="#">6.11.10.10 Inexact Exception Condition</a> .
7	VXSNAN	Floating-point invalid operation exception for SNaN (VXSNAN). This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
8	VXISI	Floating-point invalid operation exception for $\times \times$ (VXISI). This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
9	VXIDI	Floating-point invalid operation exception for $\times / \times$ (VXIDI). This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
10	VXZDZ	Floating-point invalid operation exception for $0/0$ (VXZDZ). This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
11	VXIMZ	Floating-point invalid operation exception for $\times * 0$ (VXIMZ). This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
12	VXVC	Floating-point invalid operation exception for invalid compare (VXVC). This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
13	FR	Floating-point fraction rounded (FR). The last floating-point instruction that potentially rounded the intermediate result incremented the fraction. (See <a href="#">3.3.11 Rounding</a> .) This bit is not sticky.
14	FI	Floating-point fraction inexact (FI). The last floating-point instruction that potentially rounded the intermediate result produced an inexact fraction or a disabled exponent overflow. (See <a href="#">3.3.11 Rounding</a> .) This bit is not sticky.
[15:19]	FPRF	<p>Floating-point result flags (FPRF). This field is based on the value placed into the target register even if that value is undefined. Refer to <a href="#">Table 6-27</a> for specific bit settings.</p> <p>15 Floating-point result class descriptor (C). Floating-point instructions other than the compare instructions may set this bit with the FPCC bits, to indicate the class of the result.</p> <p>[16:19] Floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Other floating-point instructions may set the FPCC bits with the C bit, to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero.</p> <p>16 Floating-point less than or negative (FL or &lt;)</p> <p>17 Floating-point greater than or positive (FG or &gt;)</p> <p>18 Floating-point equal or zero (FE or =)</p> <p>19 Floating-point unordered or NaN (FU or ?)</p>
20	—	Reserved
21	VXSOFT	Floating-point invalid operation exception for software request (VXSOFT). This bit can be altered only by the <b>mcrfs</b> , <b>mtfsfi</b> , <b>mtfsf</b> , <b>mtfsb0</b> , or <b>mtfsb1</b> instructions. The purpose of VXSOFT is to allow software to cause an invalid operation condition for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root if the source operand is negative. This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
22	VXSQRT	Floating-point invalid operation exception for invalid square root (VXSQRT). This is a sticky bit. This guarantees that software can simulate <b>fsqrt</b> and <b>frsqrite</b> , and to provide a consistent interface to handle exceptions caused by square-root operations. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .

# Freescale Semiconductor, Inc.

## Table 6-26 FPSCR Bit Settings (Continued)

Bit(s)	Name	Description
23	VXCVI	Floating-point invalid operation exception for invalid integer convert (VXCVI). This is a sticky bit. See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
24	VE	Floating-point invalid operation exception enable (VE). See <a href="#">6.11.10.6 Invalid Operation Exception Conditions</a> .
25	OE	Floating-point overflow exception enable (OE). See <a href="#">6.11.10.8 Overflow Exception Condition</a> .
26	UE	Floating-point underflow exception enable (UE). This bit should not be used to determine whether denormalization should be performed on floating-point stores. See <a href="#">6.11.10.9 Underflow Exception Condition</a> .
27	ZE	Floating-point zero divide exception enable (ZE). See <a href="#">6.11.10.7 Zero Divide Exception Condition</a> .
28	XE	Floating-point inexact exception enable (XE). See <a href="#">6.11.10.10 Inexact Exception Condition</a> .
29	NI	Non-IEEE mode bit. See <a href="#">3.4.3 Non-IEEE Operation</a> .
[30:31] ]	RN	Floating-point rounding control (RN). See <a href="#">3.3.11 Rounding</a> . 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward -infinity

**Table 6-27** illustrates the floating-point result flags that correspond to FPSCR[15:19].

## Table 6-27 Floating-Point Result Flags in FPSCR

Result Flags FPSCR[15:19] C<>=?	Result Value Class
10001	Quiet NaN
01001	-Infinity
01000	-Normalized number
11000	-Denormalized number
10010	-Zero
00010	+ Zero
10100	+ Denormalized number
00100	+Normalized number
00101	+Infinity

The following conditions cause floating-point assist exceptions when the corresponding enable bit in the FPSCR is set and the FE field in the MSR has a nonzero value (enabling floating-point exceptions). These conditions may occur during execution of floating-point arithmetic instructions. The corresponding status bits in the

FPSCR are indicated in parentheses.

- Invalid floating-point operation exception condition (VX)
  - SNaN condition (VXSNAN)
  - Infinity–infinity condition (VXISI)
  - Infinity/infinity condition (VXIDI)
  - Zero/zero condition (VXZDZ)
  - Infinity\*zero condition (VXIMZ)
  - Illegal compare condition (VXVC)
 These exception conditions are described in [6.11.10.6 Invalid Operation Exception Conditions](#).
- Software request condition (VXSOFT). These exception conditions are described in [6.11.10.6 Invalid Operation Exception Conditions](#).
- Illegal integer convert condition (VXCVI). These exception conditions are described in [6.11.10.6 Invalid Operation Exception Conditions](#).
- Zero divide exception condition (ZX). These exception conditions are described in [6.11.10.7 Zero Divide Exception Condition](#).
- Overflow Exception Condition (OX). These exception conditions are described in [6.11.10.8 Overflow Exception Condition](#).
- Underflow Exception Condition (UX). These exception conditions are described in [6.11.10.9 Underflow Exception Condition](#).
- Inexact Exception Condition (XX). These exception conditions are described in [6.11.10.10 Inexact Exception Condition](#).

Each floating-point exception condition and each category of illegal floating-point operation exception condition have a corresponding exception bit in the FPSCR. In addition, each floating-point exception has a corresponding enable bit in the FPSCR. The exception bit indicates the occurrence of the corresponding condition. If a floating-point exception occurs, the corresponding enable bit governs the result produced by the instruction and, in conjunction with bits FE0 and FE1, whether and how the system floating-point enabled exception error handler is invoked. (The “enabling” specified by the enable bit is of invoking the system error handler, not of permitting the exception condition to occur. The occurrence of an exception condition depends only on the instruction and its inputs, not on the setting of any control bits.)

The floating-point exception summary bit (FX) in the FPSCR is set when any of the exception condition bits transitions from a zero to a one or when explicitly set by software. The floating-point enabled exception summary bit (FEX) in the FPSCR is set when any of the exception condition bits is set and the exception is enabled (enable bit is one).

A single instruction may set more than one exception condition bit in the following cases:

- The inexact exception condition bit may be set with overflow exception condition.
- The inexact exception condition bit may be set with underflow exception condition.
- The illegal floating-point operation exception condition bit (SNaN) may be set

## Freescal Semiconductor, Inc.

with illegal floating-point operation exception condition ( $\times 0$ ) for multiply-add instructions.

- The illegal operation exception condition bit (SNaN) may be set with illegal floating-point operation exception condition (illegal compare) for compare ordered instructions.
- The illegal floating-point operation exception condition bit (SNaN) may be set with illegal floating-point operation exception condition (illegal integer convert) for convert to integer instructions.

When an exception occurs, the instruction execution may be suppressed or a result may be delivered, depending on the exception condition.

Instruction execution is suppressed for the following kinds of exception conditions, so that there is no possibility that one of the operands is lost:

- Enabled illegal floating-point operation
- Enabled zero divide

For the remaining kinds of exception conditions, a result is generated and written to the destination specified by the instruction causing the exception. The result may be a different value for the enabled and disabled conditions for some of these exception conditions. The kinds of exception conditions that deliver a result are the following:

- Disabled illegal floating-point operation
- Disabled zero divide
- Disabled overflow
- Disabled underflow
- Disabled inexact
- Enabled overflow
- Enabled underflow
- Enabled inexact

Subsequent sections define each of the floating-point exception conditions and specify the action taken when they are detected.

The IEEE standard specifies the handling of exception conditions in terms of traps and trap handlers. In the PowerPC architecture, setting an FPSCR exception enable bit causes generation of the result value specified in the IEEE standard for the trap enabled case — the expectation is that the exception is detected by software, which will revise the result. An FPSCR exception enable bit of zero causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case — the expectation is that the exception will not be detected by software, which will simply use the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs, which is to generate a default value and not to notify software, is obtained by clearing all FPSCR exception enable bits and using ignore exceptions mode (see [Table 6-8](#)). In this case the system floating-point assist error handler is not invoked, even if floating-point excep-

tions occur. If necessary, software can inspect the FPSCR exception bits to determine whether exceptions have occurred.

If the program exception handler notifies software that a given exception condition has occurred, the corresponding FPSCR exception enable bit must be set and a mode other than ignore exceptions mode must be used. In this case the system floating-point assist error handler is invoked if an enabled floating-point exception condition occurs.

Whether the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs is controlled by MSR bits FE0 and FE1 as shown in [Table 6-8](#). (The system floating-point enabled exception error handler is never invoked because of a disabled floating-point exception.)

**Table 6-28 Floating-Point Exception Mode Bits**

FE[0:1]	Mode
00	Ignore exceptions mode — Floating-point exceptions do not cause the floating-point assist error handler to be invoked.
01, 10, 11	Floating-point precise mode — The system floating-point assist error handler is invoked precisely at the instruction that caused the enabled exception.

Whenever the system floating-point enabled exception error handler is invoked, the processor ensures that all instructions logically residing before the excepting instruction have completed, and no instruction after that instruction has been executed.

If exceptions are ignored, an FPSCR instruction can be used to force any exceptions caused by instructions initiated before the FPSCR instruction to be recorded in the FPSCR. A **sync** instruction can also be used to force exceptions, but is likely to degrade performance more than an FPSCR instruction.

For the best performance across the widest range of implementations, the following guidelines should be considered:

- If the IEEE default results are acceptable to the application, FE0 and FE1 should be cleared (ignore exceptions mode). All FPSCR exception enable bits should be cleared.
- For even faster operation, non-IEEE can be selected by setting the NI bit in the FPSCR. To ensure that the software envelope is never invoked, select non-IEEE mode, disable all floating-point exceptions, and avoid using denormalized numbers as input to floating-point calculations. Refer to [3.4.3 Non-IEEE Operation](#) and [3.4.4 Working Without the Software Envelope](#) for more information.
- Ignore exceptions mode should not, in general, be used when any FPSCR exception enable bits are set.

- Precise mode may degrade performance in some implementations, perhaps substantially, and therefore should be used only for debugging and other specialized applications.

## 6.11.10.6 Invalid Operation Exception Conditions

An invalid operation exception occurs when an operand is invalid for the specified operation. The invalid operations are as follows:

- Any operation except load, store, move, select, or **mtfsf** on a signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities ( $\infty - \infty$ )
- Division of infinity by infinity ( $\infty / \infty$ )
- Division of zero by zero ( $0/0$ )
- Multiplication of infinity by zero ( $\infty * 0$ )
- Ordered comparison involving a NaN (invalid compare)
- Square root or reciprocal square root of a negative, non-zero number (invalid square root)
- Integer convert involving a number that is too large to be represented in the format, an infinity, or a NaN (invalid integer convert)

FPSCR[VXSOFT] allows software to cause an invalid operation exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root if the source operand is negative. This facilitates the emulation of PowerPC instructions not implemented in the RCPU.

When an invalid-operation exception occurs, the action to be taken depends on the setting of the invalid operation exception enable bit of the FPSCR. When invalid operation exception is enabled (FPSCR[VE] = 1) and invalid operation occurs or software explicitly requests the exception, the following actions are taken:

- The following status bits are set in the FPSCR:
  - VXSNAN (if SNaN)
  - VXISI (if  $\infty - \infty$ )
  - VXIDI (if  $\infty / \infty$ )
  - VXZDZ (if  $0/0$ )
  - VXIMZ (if  $\infty * 0$ )
  - VXVC (if invalid comparison)
  - VXSOFT (if software request)
  - VXCVI (if invalid integer convert)
- If the operation is an arithmetic or convert-to-integer operation,
  - the target FPR is unchanged
  - FPSCR[FR] and FPSCR[FI] are cleared
  - FPSCR[FPRF] is unchanged
- If the operation is a compare,
  - the FR, FI, and C bits in the FPSCR are unchanged
  - FPSCR[FPCC] is set to reflect unordered
- If software explicitly requests the exception, FPSCR[FR FI FPRF] are as set by the **mtfsfi**, **mtfsf**, or **mtfsb1** instruction



## Freescal Semiconductor, Inc.

When invalid operation exception condition is disabled ( $\text{FPSCRVE} = 0$ ) and invalid operation occurs or software explicitly requests the exception, the following actions are taken:

- The same status bits are set in the FPSCR as when the exception is enabled.
- If the operation is an arithmetic operation,
  - the target FPR is set to a quiet NaN
  - $\text{FPSCR}[\text{FR}]$  and  $\text{FPSCR}[\text{FI}]$  are cleared
  - $\text{FPSCR}[\text{FPRF}]$  is set to indicate the class of the result (quiet NaN)
- If the operation is a convert to 32-bit integer operation, the target FPR is set as follows:
  - $\text{FRT}[0:31] = \text{undefined}$
  - $\text{FRT}[32:63] = \text{most negative 32-bit integer}$
  - $\text{FPSCR}[\text{FR}]$  and  $\text{FPSCR}[\text{FI}]$  are cleared
  - $\text{FPSCR}[\text{FPRF}]$  is undefined
- If the operation is a convert to 64-bit integer operation, the target FPR is set as follows:
  - $\text{FRT}[0:63] = \text{most negative 64-bit integer}$
  - $\text{FPSCR}[\text{FR}]$  and  $\text{FPSCR}[\text{FI}]$  are cleared
  - $\text{FPSCR}[\text{FPRF}]$  is undefined
- If the operation is a compare,
  - The FR, FI, and C bits in the FPSCR are unchanged
  - $\text{FPSCR}[\text{FPCC}]$  is set to reflect unordered
- If software explicitly requests the exception, the FR, FI and FPRF fields in the FPSCR are as set by the **mtfsfi**, **mtfsf**, or **mtfsb1** instruction.

### 6.11.10.7 Zero Divide Exception Condition

A zero divide exception condition occurs when a divide instruction is executed with a zero divisor value and a finite, non-zero dividend value.

When a zero divide exception occurs, the action to be taken depends on the setting of the zero divide exception condition enable bit of the FPSCR. When the zero divide exception condition is enabled ( $\text{FPSCR}[\text{ZE}] = 1$ ) and a zero divide condition occurs, the following actions are taken:

- Zero divide exception condition bit is set:  $\text{FPSCR}[\text{ZX}] = 1$
- The target FPR is unchanged
- $\text{FPSCR}[\text{FR}]$  and  $\text{FPSCR}[\text{FI}]$  are cleared
- $\text{FPSCR}[\text{FPRF}]$  is unchanged

When zero divide exception condition is disabled ( $\text{FPSCR}[\text{ZE}] = 0$ ) and zero divide occurs, the following actions are taken:

- Zero divide exception condition bit is set:  $\text{FPSCR}[\text{ZX}] = 1$
- The target FPR is set to a  $\pm\text{infinity}$ , where the sign is determined by the XOR of the signs of the operands
- $\text{FPSCR}[\text{FR}]$  and  $\text{FPSCR}[\text{FI}]$  are cleared
- $\text{FPSCR}[\text{FPRF}]$  is set to indicate the class and sign of the result ( $\pm\text{infinity}$ )

### 6.11.10.8 Overflow Exception Condition

Overflow occurs when the magnitude of what would have been the rounded result if the exponent range were unbounded exceeds that of the largest finite number of the specified result precision.

The action to be taken depends on the setting of the overflow exception condition enable bit of the FPSCR. When the overflow exception condition is enabled (FPSCR[OE] = 1) and an exponent overflow condition occurs, the following actions are taken:

- Overflow exception condition bit is set: FPSCR[OX] = 1.
- For double-precision arithmetic instructions, the exponent of the normalized intermediate result is adjusted by subtracting 1536.
- For single-precision arithmetic instructions and the floating round to single-precision instruction, the exponent of the normalized intermediate result is adjusted by subtracting 192.
- The adjusted rounded result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result ( $\pm$ normal number).

When the overflow exception condition is disabled (FPSCR[OE] = 0) and an overflow condition occurs, the following actions are taken:

- Overflow exception condition bit is set: FPSCR[OX] = 1
- Inexact exception condition bit is set: FPSCR[XX] = 1
- The result is determined by the rounding mode (FPSCR[RN]) and the sign of the intermediate result as follows:
  - Round to nearest  
Store  $\pm$  infinity, where the sign is the sign of the intermediate result
  - Round toward zero  
Store the format's largest finite number with the sign of the intermediate result
  - Round toward +infinity  
For negative overflows, store the format's most negative finite number; for positive overflows, store +infinity
  - Round toward -infinity  
For negative overflows, store -infinity; for positive overflows, store the format's largest finite number
- The result is placed into the target FPR
- FPSCR[FR FI] are cleared
- FPSCR[FPRF] is set to indicate the class and sign of the result ( $\pm$ infinity or  $\pm$ normal number)

### 6.11.10.9 Underflow Exception Condition

The underflow exception condition is defined separately for the enabled and disabled states:

- Enabled — Underflow occurs when the intermediate result is tiny.
- Disabled — Underflow occurs when the intermediate result is tiny and there is loss of accuracy.

## Freescal Semiconductor, Inc.

A tiny result is detected before rounding, when a non-zero result value computed as though the exponent range were unbounded would be less in magnitude than the smallest normalized number.

If the intermediate result is tiny and the underflow exception condition enable bit is cleared (FPSCR[UE] = 0), the intermediate result is denormalized.

Loss of accuracy is detected when the delivered result value differs from what would have been computed were both the exponent range and precision unbounded.

When an underflow exception occurs, the action to be taken depends on the setting of the underflow exception condition enable bit of the FPSCR.

When the underflow exception condition is enabled (FPSCR[UE] = 1) and an exponent underflow condition occurs, the following actions are taken:

- Underflow exception condition bit is set: FPSCR[UX] = 1.
- For double-precision arithmetic and conversion instructions, the exponent of the normalized intermediate result is adjusted by adding 1536.
- For single-precision arithmetic instructions and the floating round to single-precision instruction, the exponent of the normalized intermediate result is adjusted by adding 192.
- The adjusted rounded result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result ( $\pm$ normalized number).

The FR and FI bits in the FPSCR allow the system floating-point enabled exception error handler, when invoked because of an underflow exception condition, to simulate a trap disabled environment. That is, the FR and FI bits allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

When the underflow exception condition is disabled (FPSCR[UE] = 0) and an underflow condition occurs, the following actions are taken:

- Underflow exception condition enable bit is set: FPSCR[UX] = 1
- The rounded result is placed into the target FPR
- FPSCR[FPRF] is set to indicate the class and sign of the result ( $\pm$ denormalized number or  $\pm$ zero)

### 6.11.10.10 Inexact Exception Condition

The inexact exception condition occurs when one of two conditions occur during rounding:

- The rounded result differs from the intermediate result assuming the intermediate result exponent range and precision to be unbounded.
- The rounded result overflows and overflow exception condition is disabled.

When the inexact exception condition occurs, regardless of the setting of the inexact exception condition enable bit of the FPSCR, the following actions are taken:

## Freescall Semiconductor, Inc.

- Inexact exception condition enable bit in the FPSCR is set: FPSCR[XX] = 1.
- The rounded or overflowed result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result.

### 6.11.11 Software Emulation Exception (0x01000)

An implementation-dependent software emulation exception occurs in the following cases:

- An attempt is made to execute an instruction that is not implemented in the RCPU. This includes all illegal and optional instructions. Since an RCPU-based MCU does not contain a data cache, segment registers, or a translation lookaside buffer, the following optional PowerPC instructions cause the RCPU to generate a software emulation exception:
  - Data cache instructions (**dcbt**, **dcbtst**, **dcbz**, **dcbst**, **dcbf**, **dcbi**)
  - Instructions to access segment registers (**mtsr**, **mfsr**, **mtsrin**, **mfsrin**)
  - Instructions to manage translation lookaside buffers (**tlbei**, **tlbiex**, **tlbsync**, **tlbie**, **tlbia**)
- An attempt is made to execute an **mtspr** or **mfspr** instruction that specifies an unimplemented internal-to-the-processor SPR. (This exception is taken regardless of the value of the SPR0 bit of the instruction. That is, if the SPR0 bit of the instruction equals one, indicating a privileged register, and the processor is operating in user mode, this exception is taken rather than a program exception.)

Refer to [7.5 Implementation of Special-Purpose Registers](#) for an explanation of internal- and external-to-the-processor SPRs.

- An attempt is made to execute a **mtspr** or **mfspr** instruction that specifies an unimplemented external-to-the-processor register, and either SPR0 = 0 or MSR[PR] = 0 (no program exception condition).

Register settings after a software emulation exception is taken are shown in [Table 6-22](#).

**Table 6-29 Register Settings Following a Software Emulation Exception**

Register	Setting Description	
SRR0	Set to the effective address of the instruction that caused the exception	
SRR1	[0:15]	Cleared to zero
	[16:31]	Loaded from MSR[16:31]
MSR	IP	No change
	ME	No change
	LE	Set to value of ILE bit prior to the exception
	Other bits	Cleared

When a software emulation exception is taken, execution resumes at offset 0x01000 from the base address indicated by MSR[IP].

#### 6.11.12 Data Breakpoint Exception (0x01C00)

An implementation-dependent data (L-bus) breakpoint occurs when an internal breakpoint match occurs on the load/store bus.

The processor can be programmed to recognize a data breakpoint at all times (non-masked mode), or only when the MSR[RI] bit is set (masked mode). When operating in non-masked mode, the processor enters a non-restartable state if it recognizes an internal breakpoint when MSR[RI] is cleared.

In order to enable the user to use the breakpoint features without adding restrictions on the software, the address of the load/store cycle that generated the data breakpoint is not stored in the DAR (data address register), as with other exceptions that occur during loads or stores. Instead, the address of the load/store cycle that generated the breakpoint is stored in an implementation dependent register called the breakpoint address register (BAR).

Register settings after a data breakpoint exception is taken are shown in [Table 6-22](#).

**Table 6-30 Register Settings Following Data Breakpoint Exception**

Register	Setting Description
SRR0	Set to the effective address of the instruction following the instruction that caused the exception
SRR1	[0:15] Cleared to zero [16:31] Loaded from bits MSR[16:31] If development port request is asserted at reset, the value of SRR1 is undefined.
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared
BAR	Set to the effective address of the data access as computed by the instruction that caused the exception.
DSISR, DAR	No change

When a data breakpoint exception is taken, execution resumes at offset 0x01C00 from the base address indicated by MSR[IP].

Refer to **SECTION 8 DEVELOPMENT SUPPORT** for additional information on data breakpoints.

#### 6.11.13 Instruction Breakpoint Exception (0x01D00)

An implementation-dependent instruction (I-bus) breakpoint occurs when an internal breakpoint match occurs on the instruction bus.

The processor can be programmed to recognize a data breakpoint at all times (non-masked mode), or only when the MSR[RI] bit is set (masked mode). When operating in non-masked mode, the processor enters a non-restartable state if it recognizes an internal breakpoint when MSR[RI] is cleared.

Register settings after an instruction breakpoint exception is taken are shown in **Table 6-22**.

**Table 6-31 Register Settings Following an Instruction Breakpoint Exception**

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the exception
SRR1	[0:15] Cleared to zero [16:31] Loaded fromMSR[16:31] If development port request is asserted at reset, the value of SRR1 is undefined.
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared

When an instruction breakpoint exception is taken, execution resumes at offset 0x01D00 from the base address indicated by MSR[IP].

Refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information on instruction breakpoint exceptions.

#### 6.11.14 Maskable External Breakpoint Exception (0x01E00)

An implementation-dependent maskable external breakpoint can be generated by any of the peripherals of the system, including those found on the L-bus, I-bus, IMB2 and external bus, and also by an external development system. Peripherals found on the external bus use the serial interface of the development port to assert the external breakpoint. Breakpoints are generated by the development port from the associated bits of the trap enable control register.

Maskable external breakpoint exceptions are asynchronous and ordered. The processor does not take the exception if the RI (recoverable exception) bit in the MSR is cleared. Refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information.

**Table 6-32 Register Settings Following a Maskable External Breakpoint Exception**

Register	Setting Description
SRR0	Set to the effective address of the instruction that caused the exception
SRR1	[0:15] Cleared to zero [16:31] Loaded fromMSR[16:31] If development port request is asserted at reset, the value of SRR1 is undefined.
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared to zero

When a maskable external breakpoint exception is taken, execution resumes at offset 0x01E00 from the base address indicated by MSR[IP].

**6.11.15 Non-Maskable External Breakpoint Exception (0x01F00)**

An implementation-dependent non-maskable external breakpoint exception is generated by the development port from the associated bits of the trap enable mode serial communications.

This exception is asynchronous and unordered. The exception is not referenced to any particular instruction. The processor stops instruction execution and either begins exception processing or enters debug mode as soon as possible after detecting the breakpoint exception.

The non-maskable external breakpoint exception causes the processor to stop without regard to the state of the MSR[RI] bit. If the processor is in a non-recoverable state when the exception occurs, the state of the SRR0, SRR1, DAR, and DSISR registers may have been overwritten. In this case, it is not possible to restart the processor since the restarting address and MSR context are saved in the SRR0 and SRR1 registers.

This exception allows the user to stop the processor in cases in which it would otherwise not stop, but with the penalty that the processor may not be restartable. The value of the MSR[RI] bit, as saved in the SRR1 register, indicates whether the processor stopped in a recoverable state or not.

**Table 6-33 Register Settings Following a  
Non-Maskable External Breakpoint Exception**

Register	Setting Description
SRR0	Set to the effective address of the instruction that would have been executed next if no exception had occurred. If the development port request is asserted at reset, the value of SRR0 is undefined.
SRR1	[0:15] Cleared to zero [16:31] Loaded from bits [16:31] of the MSR If development port request is asserted at reset, the value of SRR1 is undefined.
MSR	IP No change ME No change LE Set to value of ILE bit prior to the exception Other bits Cleared to zero

When a non-maskable external breakpoint exception is taken, execution resumes at offset 0x01000 from the base address indicated by MSR[IP].



## SECTION 7

### INSTRUCTION TIMING

This section describes instruction flow and the basic instruction pipeline in the RCPU, provides details of execution timing for each execution unit, defines the concepts of serialization and synchronization, provides timing information for each RCPU instruction, and provides timing examples for different types of instructions.

#### 7.1 Instruction Flow

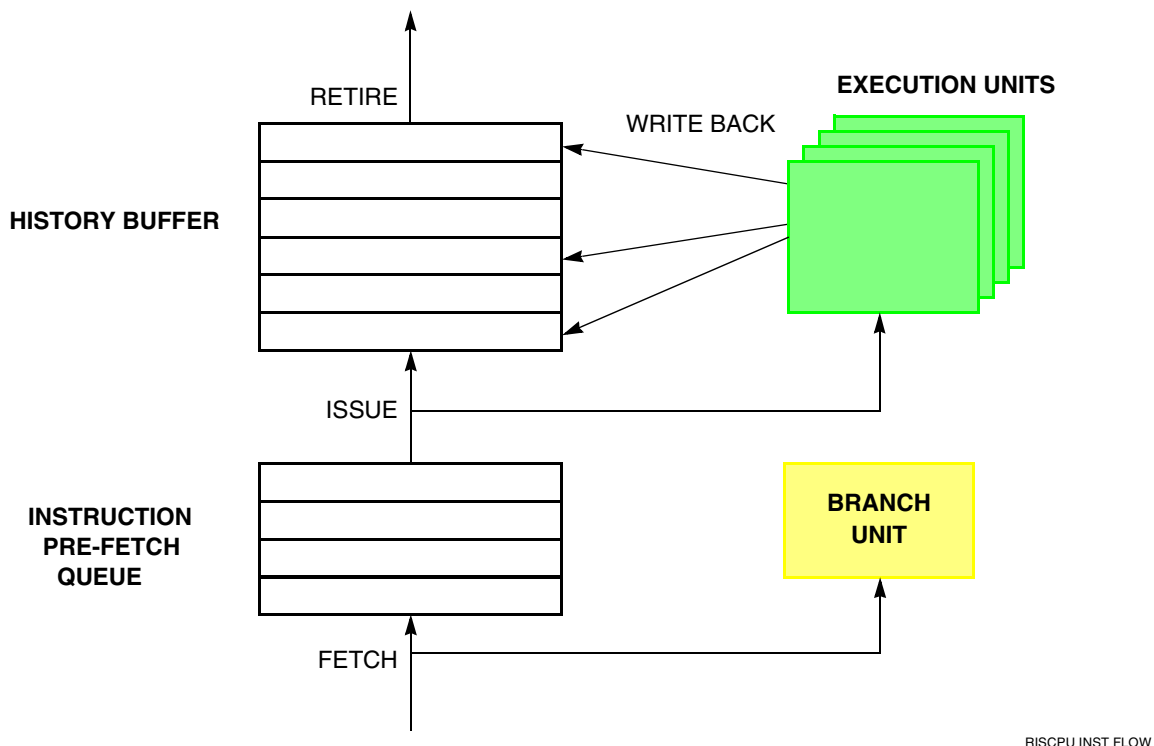
The instruction sequencer provides centralized control over data flow between execution units and register files. The sequencer implements the basic instruction pipeline, fetches instructions from the memory system, issues them to available execution units, and maintains a state history so it can back the machine up in the event of an exception.

The instruction sequencer fetches the instructions from the instruction cache into the instruction pre-fetch queue. The processor uses branch folding (a technique of removing the branch instructions from the pre-fetch queue) in order to execute branches in parallel with execution of sequential instructions. Sequential (non-branch) instructions reaching the top of the instruction queue are issued to the execution units. Instructions may be flushed from the instruction queue when an external interrupt is detected, a previous instruction causes an exception, or a branch prediction turns out to be incorrect.

All instructions, including branches, enter the history buffer along with processor state information that may be affected by the instruction's execution. This information is used to enable out of order completion of instructions together with precise exceptions handling. Instructions may be flushed from the machine when an exception is taken. The instruction queue is always flushed when recovery of the history buffer takes place. Refer to [6.3 Precise Exception Model Implementation](#) for additional information.

An instruction retires from the machine after it finishes execution without exception and all preceding instructions have already retired from the machine.

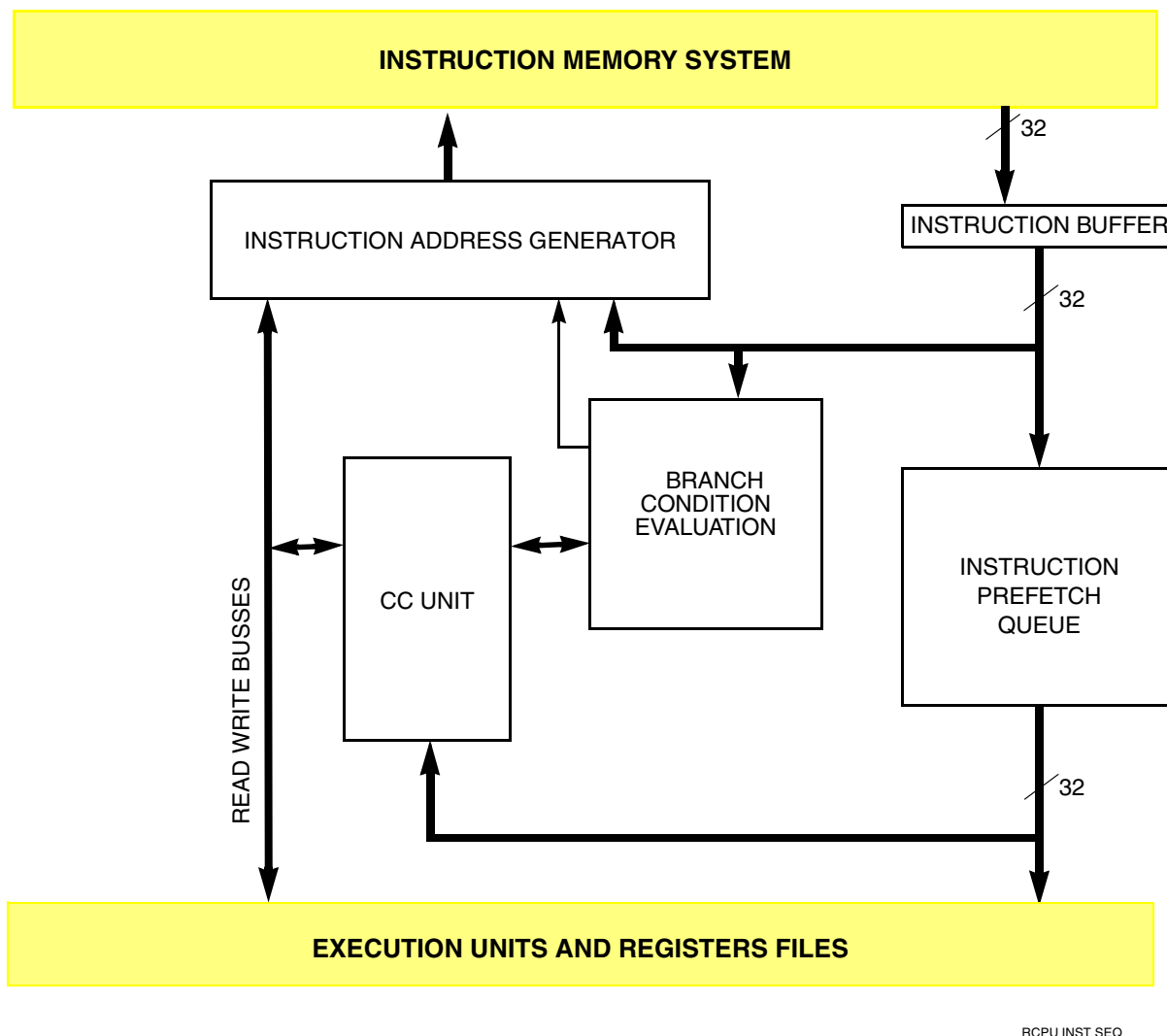
**Figure 7-1** illustrates the instruction flow in the RCPU.



**Figure 7-1 Instruction Flow**

### 7.1.1 Instruction Sequencer Data Path

**Figure 7-2** illustrates the instruction sequencer data path.



**Figure 7-2 Instruction Sequencer Data Path**

### 7.1.2 Instruction Issue

The sequencer attempts to issue a sequential (non-branch) instruction on each clock, if possible. In order for an instruction to be issued, the execution unit must be available and it must determine that the required source data is available and that no other instruction still in execution targets the same destination register. The sequencer broadcasts the presence of the instruction on the instruction bus, and each execution unit decodes the instruction. The execution unit responsible for executing the instruction determines whether the operands and target registers are free and informs the sequencer that it accepts the instruction for execution.

### 7.1.3 Basic Instruction Pipeline

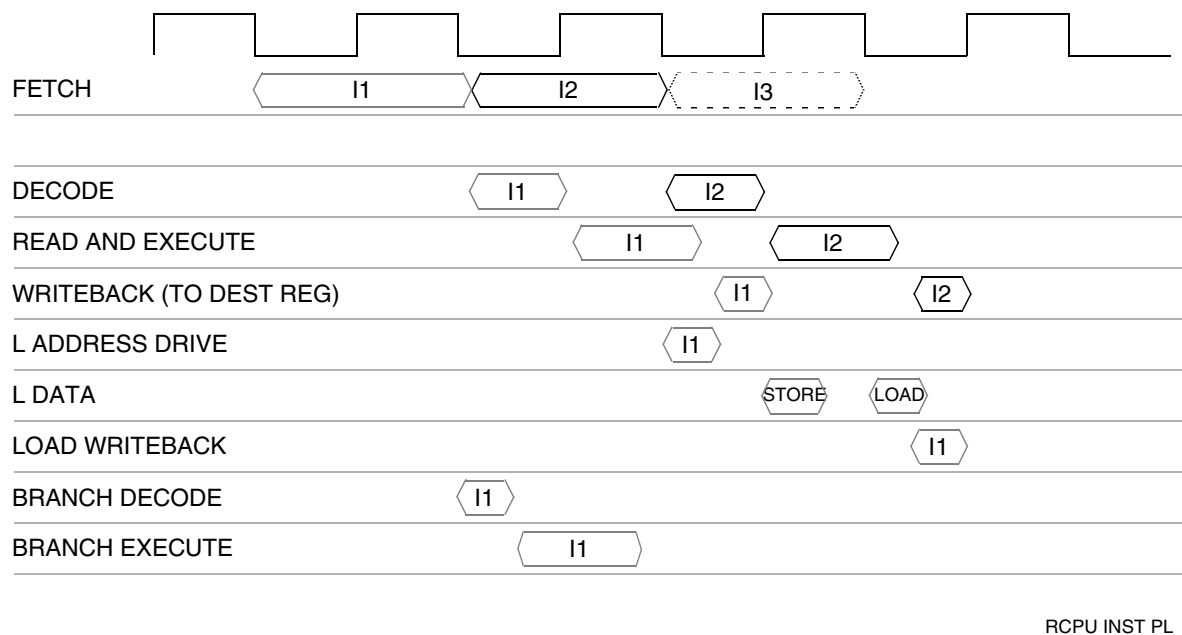
The RCPU instruction pipeline has four stages:

**Freescale Semiconductor, Inc.**

1. The dispatch stage is implemented using a distributed mechanism. The central dispatch unit broadcasts the instruction to all units. In addition, scoreboard information (regarding data dependencies) is broadcast to each execution unit. Each execution unit decodes the instruction. If the instruction is not implemented, a program exception is taken. If the instruction is legal and no data dependency is found, the instruction is accepted by the appropriate execution unit, and the data found in the destination register is copied to the history buffer. If a data dependency exists, the machine is stalled until the dependency is resolved.
2. In the execute stage, each execution unit that has an executable instruction executes the instruction (perhaps over multiple cycles).
3. In the writeback stage, the execution unit writes the result to the destination register and reports to the history buffer that the instruction is completed.
4. In the retirement stage, the history buffer retires instructions in architectural order. An instruction retires from the machine if it completes execution with no exceptions and if all instructions preceding it in the instruction stream have finished execution with no exceptions. As many as six instructions can be retired in one clock.

The history buffer maintains the correct architectural machine state. An exception is taken only when the instruction is ready to be retired from the machine (i.e., after all previously-issued instructions have already been retired from the machine). When an exception is taken, all instructions following the excepting instruction are canceled, (i.e., the values of the affected destination registers are restored using the values saved in the history buffer during the dispatch stage).

**Figure 7-3** illustrates the basic instruction pipeline timing.



**Figure 7-3 Basic Instruction Pipeline**

## 7.2 Execution Unit Timing Details

The following sections describe instruction timing considerations within each RCPU execution unit.

### 7.2.1 Integer Unit (IU)

The integer unit executes all integer processor instructions, except the integer storage access instructions, which are implemented by the load/store unit. The IU consists of two execution units:

- The IMUL-IDIV executes the integer multiply and divide instructions.
- The ALU-BFU unit executes all integer logic, add, and subtract instructions, and bit-field instructions.

All instructions executed by the ALU-BFU, except for integer trap instructions, have a latency of one clock cycle. Instructions executed by the IMUL-IDIV unit have latencies of more than one clock cycle. The IMUL-IDIV unit is pipelined for multiply instructions, but not for divide instructions. Therefore, the instruction sequencer can issue one instruction to the IU each clock cycle, except when an integer divide instruction is preceded or followed by an integer divide or multiply instruction.

## 7.2.1.1 Update of the XER During Divide Instructions

Integer divide instructions have a relatively long latency. However, these instructions can update XER[OV], the overflow bit in the integer exception register, after one cycle. Data dependency on the XER is therefore limited to one cycle although the latency of an integer divide instruction can be up to eleven clock cycles.

## 7.2.2 Floating Point Unit (FPU)

The floating-point unit contains a double-precision multiply array, the floating-point status and control register (FPSCR), and the FPRs. The multiply-add array allows the processor to efficiently implement floating-point operations such as multiply, multiply-add, and divide.

The RCPU depends on a software envelope to fully implement the IEEE floating-point specification. Overflows, underflows, NaNs, and denormalized numbers cause floating-point assist exceptions that invoke a software routine to deliver (with hardware assistance) the correct IEEE result. Refer to [6.11.10 Floating-Point Assist Exception \(0x00E00\)](#) for additional information.

To accelerate time-critical operations and make them more deterministic, the RCPU provides a mode of operation that avoids invoking the software envelope and attempts to deliver results in hardware that are adequate for most applications, if not in strict conformance with IEEE standards. In this mode, denormalized numbers, NaNs, and IEEE invalid operations are treated as legitimate, returning default results rather than causing floating-point assist exceptions.

## 7.2.3 Load/Store Unit (LSU)

The load-store unit handles all data transfer between the integer and floating-point register files and the chip-internal load/store bus (L-bus). The load/store unit is implemented as an independent execution unit so that stalls in the memory pipeline do not cause the master instruction pipeline to stall (unless there is a data dependency). The unit is fully pipelined so that memory instructions of any size may be issued on back-to-back cycles.

There is a 32-bit wide data path between the load/store unit and the integer register file and a 64-bit wide data path between the load/store unit and the floating-point register file.

Single-word accesses to on-chip data RAM require one clock cycle, resulting in two clock cycles latency. Double-word accesses require two clock cycles, resulting in three clock cycles latency. Since the L-bus is 32 bits wide, double-word transfers require two bus accesses.

The LSU interfaces with the external bus interface for all instructions that access memory. Addresses are formed by adding the source one register operand specified by the instruction (or zero) to either a source two register operand or to a 16-bit, immediate value embedded in the instruction.

## 7.2.3.1 Load/Store Instruction Issue

When a load or store instruction is encountered, the LSU checks the scoreboard to determine if all the operands are available. These operands include:

- Address registers operands
- Source data register operands (for store instructions)
- Destination data register operands (for load instructions)
- Destination address register operands (for load/store with update instructions)

If all operands are available, the LSU takes the instruction and enables the sequencer to issue a new instruction. Using a dedicated interface, the LSU notifies the IU to calculate the effective address.

All load and store instructions are executed and terminated in order. If there are no prior instructions waiting in the address queue, the load or store instruction is issued to the L-bus as soon as the instruction is taken. Otherwise, if there are still prior instructions whose address are still to be issued to the L-bus, the instruction is inserted into the address queue, and data (for store instructions) is inserted into the respective store data queue. Note that for load/store with update instructions, the destination address register is written back on the following clock cycle, regardless of the state of the address queue.

A new store instruction is not issued to the L-bus until all prior instructions have terminated without an exception. This is done in order to implement the PowerPC precise exception model. In case of a load instruction followed by a store instruction, a delay of one clock cycle is inserted between the termination of the load bus cycle and the issuing of the store cycle.

## 7.2.3.2 Load/Store Synchronizing Instructions

For certain LSU instructions, the instruction is not taken (as defined in the glossary) until all previous instructions have terminated. These instructions are:

- Load/Store Multiple instructions — **lmw**, **stmw**
- Storage Synchronization instructions — **lwarx**, **stwcx**, **sync**
- String instructions — **lswi**, **lswx**, **stswi**, **stswx**
- Move to internal special registers and move to external-to-processor special purpose registers

Issuing of further instructions is stalled until the following load/store instructions terminate:

- Load/Store Multiple instructions — **lmw**, **stmw**
- Storage Synchronization instructions — **lwarx**, **stwcx**, **sync**
- String instructions — **lswi**, **lswx**, **stswi**, **stswx**

## 7.2.3.3 Load/Store Instruction Timing Summary

**Table 7-1** summarizes the timing of load/store instructions, assuming a parked bus and zero wait state memory references. The parameter “N” denotes the number of registers transferred.

Table 7-1 Load/Store Instructions Timing

Instruction Type	Latency		Cleared from Load/Store Unit	
	Internal Memory	External Memory	Internal Memory	External Memory
Fixed-Point Single Target Register Load Floating-Point Single-Precision Load	2 clocks	4 clocks	2 clocks	4 clocks
Fixed-Point Single Target Register Store Floating-Point Single-Precision Store	1 clock	1 clock	2clock	4 clock
Floating-Point Double-Precision Load <sup>1</sup>	3 clocks	5 clocks	3 clocks	5 clocks
Floating-Point Double-Precision Store <sup>1</sup>	1 clock	1 clock	3 clocks	5 clocks
Load Multiple	1 + N	3 + N + [(N + 1)/3]	1 + N	3 + N + [(N + 1)/3]

## NOTES:

1. Double-precision load and store instructions are pipelined on the bus.

## 7.2.3.4 Bus Cycles for String Instructions

String instructions are broken into a series of aligned bus accesses. **Figure 7-4** illustrates the maximum number of bus cycles needed for string instruction execution. This is the case where the beginning and end of the string are unaligned.

0x00	00	01	02	03
0x04	04	05	06	07
0x08	08	09	0a	0b
0x0C	0c	0d	0e	0f
0x10	10	11	12	13
0x14	14	15	16	17
0x18	18	19	1a	1b

2 bus cycles

word transfers  
3 bus cycles

2 bus cycles

BUS CYC/STR EX

Figure 7-4 Number of Bus Cycles Needed for String Instruction Execution

## 7.2.3.5 Stalls During Floating-Point Store Instructions

In the following sequence there is a delay of one clock cycle before the second floating-point store instruction is executed:

1. Load instruction
2. First floating-point store instruction
3. Second floating-point store instruction



If the accesses are to zero-wait-state L-bus memory and the instructions are issued on consecutive clock cycles, the second floating-point store instruction is stalled for one clock cycle.

#### **7.2.4 Branch Processing Unit (BPU)**

The sequencer maintains a prefetch queue that can hold up to four instructions. This prefetch queue enables branches to be issued in parallel with sequential instructions. In the ideal case, a sequential instruction is issued every clock cycle, even when branches are present in the code. This feature is possible because of branch folding, the removal of branch instructions from the pre-fetch queue.

All instructions are fetched into the instruction prefetch queue, but only sequential instructions are issued to the execution units upon reaching the head of the queue. (Branches are placed into the instruction prefetch queue to enable watchpoint marking — refer to **SECTION 8 DEVELOPMENT SUPPORT** for more information.) Since branches do not prevent the issue of sequential instructions unless they come in pairs, the performance impact of entering branches in the instruction prefetch queue is negligible.

In addition to branch folding, the RCPU implements a branch reservation station and static branch prediction to allow branches to issue as early as possible. The reservation station allows a branch instruction to be issued even before its condition is ready. With the branch issued and out of the way, instruction pre-fetch can continue while the branch operand is being computed and the condition is being evaluated. Static branch prediction is used to determine which instruction stream is pre-fetched while the branch is being resolved. When the branch operand becomes available, it is forwarded to the branch unit and the condition is evaluated.

Refer to **4.6.2 Conditional Branch Control** for more information on static branch prediction.

### **7.3 Serialization**

The RCPU has multiple execution units, each of which may be executing different instructions at the same time. This concurrence is normally transparent to the user program. In certain circumstances, however (e.g., debugging, I/O control, and multi-processor synchronization), it may be necessary to force the machine to *serialize*.

Two types of serialization are defined for the RCPU: *execution serialization* and *fetch serialization*.

#### **7.3.1 Execution Serialization**

Execution serialization (also referred to as *serialization* or *execution synchronization*) causes the issue of subsequent instructions to be halted until all instructions currently in progress have completed execution, (i.e., all internal pipeline stages and instruction buffers have emptied and all outstanding memory transactions are completed).

An attempt to issue a serializing instruction causes the machine to serialize before the instruction issues. Notice that only the **sync** instruction guarantees serialization across PowerPC implementations.

## 7.3.2 Fetch Serialization

Fetch serialization (also referred to as “fetch synchronization”) causes instruction fetch to be halted until all instructions currently in the processor (i.e., all issued instructions as well as the pre-fetched instructions waiting to be issued) have completed execution.

Fetch of an **isync** instruction causes fetch serialization. This means that no instructions following **isync** in the instruction stream are pre-fetched until **isync** and all previous instructions have completed execution. In addition, when the SER (serialize mode) bit in the ICTRL is asserted, or when the processor is in debug mode, all instructions cause fetch serialization.

## 7.4 Context Synchronization

The system call (**sc**) and return from interrupt (**rfi**) instructions are context-synchronizing. Execution of one of these instructions ensures the following:

- No higher priority exception exists (**sc**).
- All previous instructions have completed to a point where they can no longer cause an exception.
- Previous instructions complete execution in the context (privilege and protection) under which they were issued.
- The instructions following the context-synchronizing instruction execute in the context established by the instruction.

## 7.5 Implementation of Special-Purpose Registers

Most special-purpose registers supported by the RCPU are physically implemented within the processor. The following SPRs, however, are physically implemented outside of the processor (i.e., in another module, such as the system interface unit, of the microcontroller):

- Instruction cache control registers (ICCST, ICADR, and IDDAT)
- Time base (TB) and decremter (DEC)
- Development port data register (DPDR)

These registers are read or written with the **mtspr** and **mfspr** instructions. The registers are physically accessed, however, via the internal L-bus or I-bus as appropriate.

The following encodings are reserved in the RCPU for SPRs not located within the processor:

Table 7-2 Encodings of External-to-the-Processor SPRs

SPR Instruction Field Encoding		Reserved for
SPR[5:9]	SPR[0:4]	
100xx	xxxxx	External-to-the-processor SPRs
100xx	x0xxx	System interface unit (SIU) route from L-bus to I-bus/internal SIU registers
100xx	x1xxx	Peripherals control unit registers
10011	x0xxx	SIU internal registers
0xxxx	xxxxx	DEC or TB, if this encoding appears on the L-bus
10000	x0xxx	Reserved for IBAT
10000	x1xxx	Reserved for DBAT
10001	x00xx	I-cache registers
10001	x1xxx	Reserved for D-cache

Many of the encodings in [Table 7-2](#) are not used in the RCPU. If the processor attempts to access to an unimplemented external-to-the-processor SPR, or if an error occurs during an access of an external-to-the-processor SPR, an implementation-dependent software emulation exception is taken (rather than a program exception).

An **mtspr** instruction to an external-to-the-processor register is not taken until all preceding instructions have terminated. Refer to [7.6 Instruction Execution Timing](#) for more information.

## 7.6 Instruction Execution Timing

[Table 7-3](#) lists the instruction execution timing in terms of latency and blockage of the appropriate execution unit. Latency refers to the interval from the time an instruction begins execution until it produces a result that is available for use by a subsequent instruction. Blockage refers to the interval from the time an instruction begins execution until its execution unit is available for a subsequent instruction. Note that a serializing instruction has the effect of blocking all execution units.

**Table 7-3 Instruction Execution Timing**

Instructions	Latency	Blockage	Execution Unit	Serializing Instruction
Branch Instructions: <b>b, ba, bl, bla, bc, bca, bcl, bcla, bclr, bclrl, bcctr, bcctl</b>	Taken 2	2	BPU	No
	Not taken 1	1		
<b>sc, rfi</b>	Serialize + 2	Serialize + 2	BPU	Yes
CR logical instructions: <b>crand, crxor, cror, crnand, crnor, crandc, creqv, crorc, mcrf</b>	1	1	BPU	No
Fixed-point trap instructions: <b>twi, tw</b>	Taken Serialize + 3	Serialize + 3	ALU/BFU	After
	Not taken 1	1		No
<b>mtspr</b> to LR, CTR	1	1	BPU	No
<b>mtspr</b> to XER, external-to-the-processor SPRs <sup>1</sup>	Serialize + 1	Serialize + 1	LSU	Refer to <a href="#">Table 7-4</a>
<b>mtspr</b> (to other registers)	Serialize + 1	Serialize + 1	BPU	Refer to <a href="#">Table 7-4</a>
<b>mfspr</b> from external-to-the-processor SPRs <sup>1</sup>	Serialize + load latency	Serialize + 1	LSU	No
<b>mfspr</b> (from other registers)	1	1	BPU	Refer to <a href="#">Table 7-4</a>
<b>mftb, mftbu</b>	Serialize + load latency	Serialize + 1	LSU	No
<b>mtcrf, mtmsr</b>	Serialize + 1	Serialize + 1	BPU	Yes
<b>mfcrr, mfmsr</b>	Serialize + 1	Serialize + 1	BPU	No
<b>mffs[.]</b>	1	1	FPU	No
<b>mcrxr</b>	Serialize + 1	Serialize + 1	LSU, BPU	Yes
<b>mcrfs</b>	Serialize + 1	Serialize + 1	FPU, BPU	Yes
Other move FPSCR: <b>mtfsfi[.], mtfsf[.], mtfsb0[.], mtfsb1[.]</b>	Serialize + 1	Serialize + 1	FPU	Yes
<b>mcrxr</b>	Serialize + 1	Serialize + 1	LSU	Yes (Before)
Integer arithmetic: <b>addi, add[o][.], addis, subf[o][.], addic, subfic, addic., addc[o][.], adde[o][.], subfc[o][.], subfe[o][.], addme[o][.], addze[o][.], subfme[o][.], subfze[o][.], neg[o][.]</b>	1	1	ALU/BFU	No
Integer arithmetic (divide instructions): <b>divw[o][.], divwu[o][.]</b>	Min 2 Max 11 <sup>2</sup>	Min 2 Max 11 <sup>3</sup>	IMUL/IDIV	No
Integer arithmetic (multiply instructions): <b>mulli, mull[o][.], mulhw[.], mulhwu[.]</b>	2	1-2 <sup>4</sup>	IMUL/IDIV	No

## Table 7-3 Instruction Execution Timing (Continued)

Instructions	Latency	Blockage	Execution Unit	Serializing Instruction
Integer compare: <b>cmpi</b> , <b>cmp</b> , <b>cmpli</b> , <b>cmpl</b>	1	1	ALU/BFU	No
Integer logical: <b>andi</b> ., <b>andis</b> ., <b>ori</b> , <b>oris</b> , <b>xori</b> , <b>xoris</b> , <b>and</b> [], <b>or</b> [], <b>xor</b> [], <b>nand</b> [], <b>nor</b> [], <b>eqv</b> [], <b>andc</b> [], <b>orc</b> [], <b>extsb</b> [], <b>extsh</b> [], <b>cntlzw</b> []	1	1	ALU/BFU	No
Integer rotate and shift: <b>rlwinm</b> [], <b>rlwnm</b> [], <b>rlwimi</b> [], <b>slw</b> [], <b>srw</b> [], <b>srawi</b> [], <b>sraw</b> []	1	1	ALU/BFU	No
Floating point move: <b>fmr</b> [], <b>fneg</b> [], <b>fabs</b> [], <b>fnabs</b> []	1	1	FPU	No
Floating point add/subtract: <b>fadd</b> [], <b>fadds</b> [], <b>fsub</b> [], <b>fsubs</b> []	4	4	FPU	No
Floating point multiply single: <b>fmuls</b> []	4	4	FPU	No
Floating point multiply double: <b>fmul</b> []	5	5	FPU	No
Floating point divide single: <b>fdivs</b> []	10	10	FPU	No
Floating point divide double: <b>fdiv</b> []	17	17	FPU	No
Floating point multiply-add single: <b>fmadds</b> [], <b>fmsubs</b> [], <b>fnmadds</b> [], <b>fnmsubs</b> []	6	6	FPU	No
Floating point multiply-add double: <b>fmadd</b> [], <b>fmsub</b> [], <b>fnmadd</b> [], <b>fnmsub</b> []	7	7	FPU	No
Floating round to single-precision: <b>frsp</b> []	2	2	FPU	No
Floating convert to integer: <b>fctiw</b> [], <b>fctiwz</b> []	3	3	FPU	No
Floating point compare: <b>fcmpu</b> , <b>fcmpo</b>	1	1	FPU	No
Integer load instructions: <b>lbz</b> , <b>lbzu</b> , <b>lbzx</b> , <b>lbzux</b> , <b>lhz</b> , <b>lhzu</b> , <b>lhzx</b> , <b>lhzux</b> , <b>lha</b> , <b>lhau</b> , <b>lhax</b> , <b>lhaux</b> , <b>lwz</b> , <b>lwzu</b> , <b>lwzx</b> , <b>lwzux</b> , <b>lhrx</b> , <b>lwbrx</b> , <b>lhrx</b>	2 <sup>5</sup>	1	LSU	No
Integer store instructions: <b>stb</b> , <b>stbu</b> , <b>stbx</b> , <b>stbux</b> , <b>sth</b> , <b>sthu</b> , <b>sthx</b> , <b>sthux</b> , <b>stw</b> , <b>stwu</b> , <b>stwbrx</b>	1 <sup>6</sup>	1	LSU	No
Integer load and store multiple instructions: <b>lmw</b> , <b>smw</b>	Serialize + 1 + Number of registers	Serialize + 1 + Number of registers	LSU	Yes
Synchronize: <b>sync</b>	Serialize + 1	Serialize + 1	LSU	Yes
Order storage access: <b>eieio</b>	Load/Store Serialize + 1	1	LSU	No

Table 7-3 Instruction Execution Timing (Continued)

Instructions	Latency	Blockage	Execution Unit	Serializing Instruction
Storage synchronization instructions: <b>lwarx, stwcx.</b>	Serialize + 2	Serialize + 2	LSU	Yes
Floating-point load single instructions: <b>lfs, lfsu, lfsx, lfsux</b>	2	1	LSU	No
Floating-point load double instructions: <b>lfd, lfd, lfdx, lfdx</b>	3	1	LSU	No
Floating-point store single instructions: <b>stfs, stfsu, stfsx, stfsux, stfiwx</b>	1	1	LSU	No
Floating-point store double instructions: <b>stfd, stfd, stfdx, stfdx</b>	1	1	LSU	No
String instructions: <b>lswi, lswx, stswi, stswx</b>	Serialize + 1 + Number of words accessed	Serialize + 1 + Number of words accessed	LSU	Yes
Storage control instructions: <b>isync</b>	serialize	serialize	BPU	Yes
<b>eieio</b>	1	1	LSU	Next load or store is serialized relative to all prior load or store
Cache control: <b>icbi</b>	1	1	LSU, I-cache	No

## NOTES:

1. SPRs that are physically implemented outside of the RCPU are the time base, decremter, ICCST, ICADR, IC-DAT, AND DPDR.

$$2. \quad \text{DivisionLatency} = \begin{cases} \text{NoOverflow} \Rightarrow 3 + \left\lfloor \frac{34 - \text{divisorLength}}{4} \right\rfloor \\ \text{Overflow} \Rightarrow 2 \end{cases}$$

$$\text{Where:} \quad \text{Overflow} = \left( \frac{x}{0} \right) \text{ or } \left( \frac{\text{MaxNegativeNumber}}{-1} \right)$$

3. DivisionBlockage = DivisionLatency
4. Blockage of the multiply instruction is dependent on the subsequent instruction  
for subsequent multiply instruction the blockage is one clock.  
for subsequent divide instruction the blockage is two clocks.
5. Assuming non-speculative aligned access, on chip memory and available bus.
6. Although stores issued to the LSU buffers free the CPU pipeline, next load or store will not actually be performed on the bus until the bus is free.

## Table 7-4 Control Registers and Serialized Access

SPR Number (Decimal)	Name	Serialize Access
1	XER	Write: full serialization Read: serialization relative to load/store operations
8	LR	No
9	CTR	No
18	DSISR	Write: full serialization Read: serialization relative to load/store operations
19	DAR	Write: full serialization Read: serialization relative to load/store operations
22	DEC	Write
26	SRR0	Write
27	SRR1	Write
80	EIE	Write
81	EID	Write
82	NRI	Write
144 – 147	CMPA – CMPD	Fetch serialized on write
148	ECR	Fetch serialized on write
149	DER	Fetch serialized on write
150	COUNTA	Fetch serialized on write
151	COUNTB	Fetch serialized on write
152 – 155	CMPE – CMPH	Write: fetch serialized Read: serialized relative to load/store operations
156	LCTRL1	Write: fetch serialized Read: serialized relative to load/store operations
157	LCTRL2	Write: fetch serialized Read: serialized relative to load/store operations
158	ICTRL	Fetch serialized on write
159	BAR	Write: fetch serialized Read: serialized relative to load/store operations
268	TBL read <sup>1</sup>	Write — as a store
269	TBU read <sup>1</sup>	Write — as a store
272 – 275	SPRG0 – SPRG3	Write
284	TBL write <sup>2</sup>	Write — as a store
285	TBU write <sup>2</sup>	Write — as a store
287	PVR	No (read only register)

Table 7-4 Control Registers and Serialized Access (Continued)

SPR Number (Decimal)	Name	Serialize Access
560	ICCST	Write — as a store
561	ICADR	Write — as a store
562	ICDAT	Write — as a store
630	DPDR	Read and write
1022	FPECR	Write
—	MSR	Fetch serialized on write
—	CR	Serialized for <b>mtcrf</b> only

## NOTES:

1. Any write (**mtspr**) to this address results in an implementation-dependent software emulation exception.
2. Any read (**mftb**) of this address results in an implementation-dependent software emulation exception.

## 7.7 Instruction Execution Timing Examples

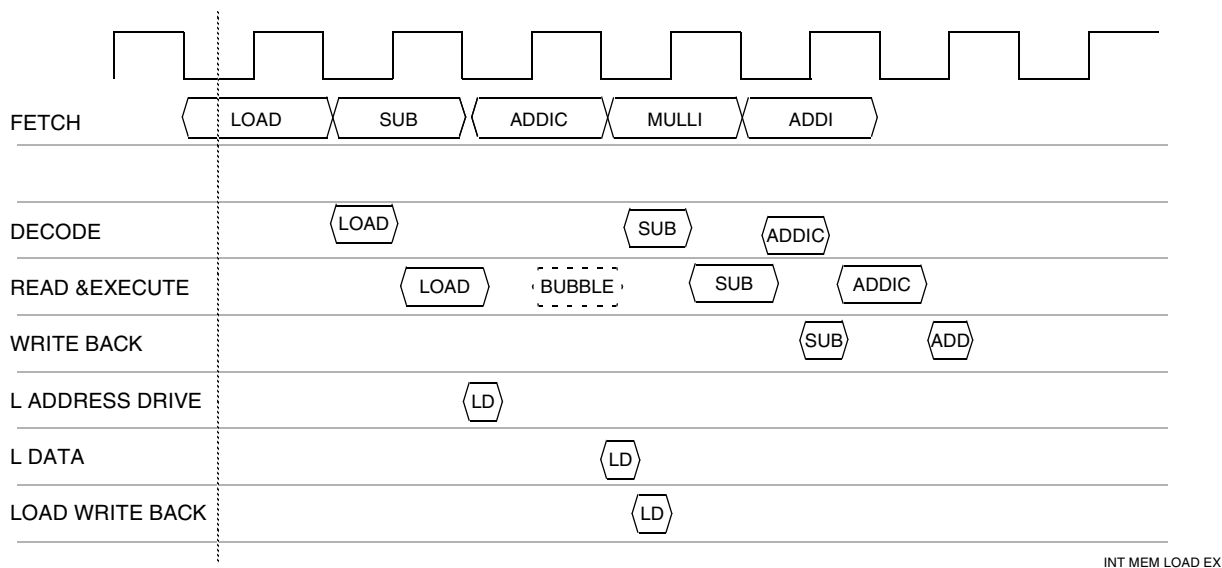
This section contains a number of examples illustrating the operation of the instruction pipeline. All examples assume an instruction cache hit.

### 7.7.1 Load from Internal Memory Example

This is an example of a load from an internal memory module with zero wait states. The **subf** instruction is dependent on the value loaded by the **lwz** to r12. This causes one bubble to occur in the instruction stream. See [7.7.3 Load with Private Write-Back Bus](#) for an example in which no such dependency exists.

```
lwz          r12, 64(r0)
subf         r3, r12, r3
addic       r4, r14, 1
mulli      r5, r3, 3
addi       r4, r0, 3
```



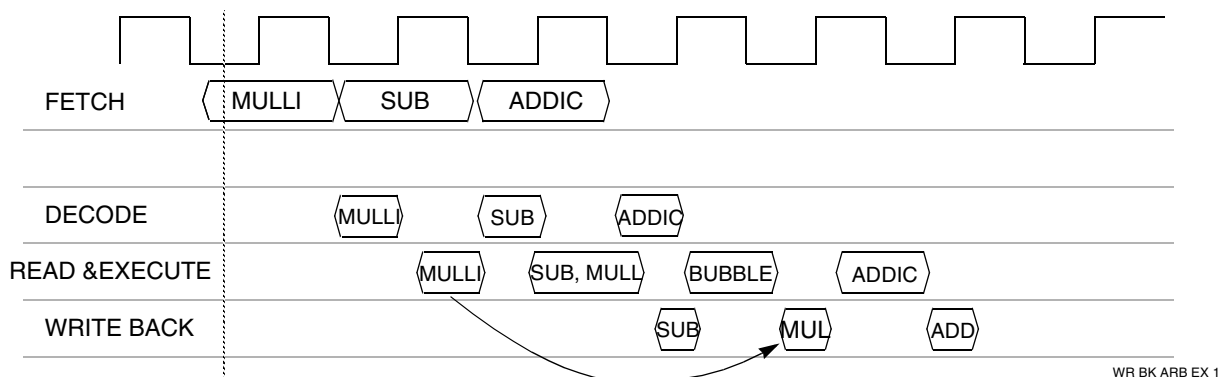


**Figure 7-5 Load from Internal Memory Example**

### 7.7.2 Write-Back Arbitration Examples

In the first example, the **addic** is dependent on the **mulli** result. Since the single cycle instruction **subf** has priority on the writeback bus over the **mulli**, the **mulli** write back is delayed one clock and causes a bubble in the execution stream.

```
mulli      r12, r4, 3
subf       r3, r15, r3
addic      r4, r12, 1
```



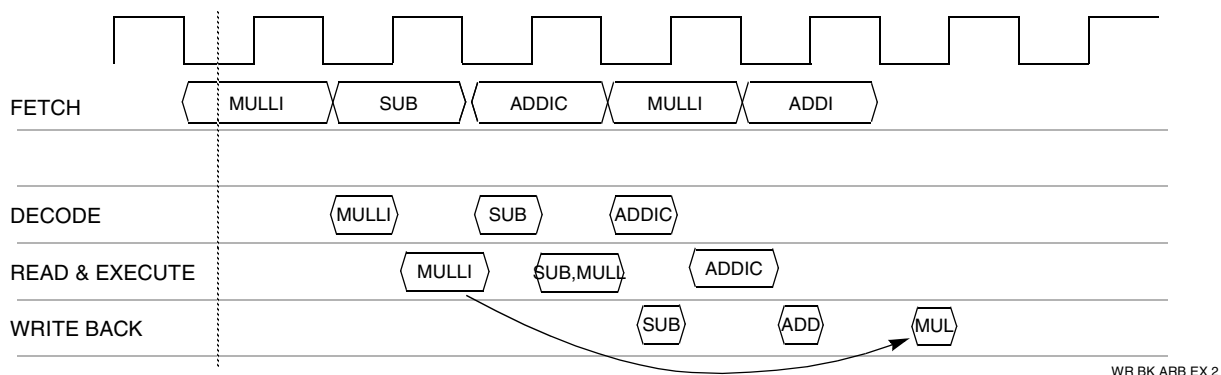
**Figure 7-6 Write-Back Arbitration Example I**

In the following example, the **addic** is dependent on the **subf** rather than on the **mulli**. Although the write back of the **mulli** is delayed two clocks, there is no bubble in the execution stream.

# Freescale Semiconductor, Inc.

```

mulli      r12,r4,3
subf       r3,r15,r3
addic      r4,r3,1
    
```



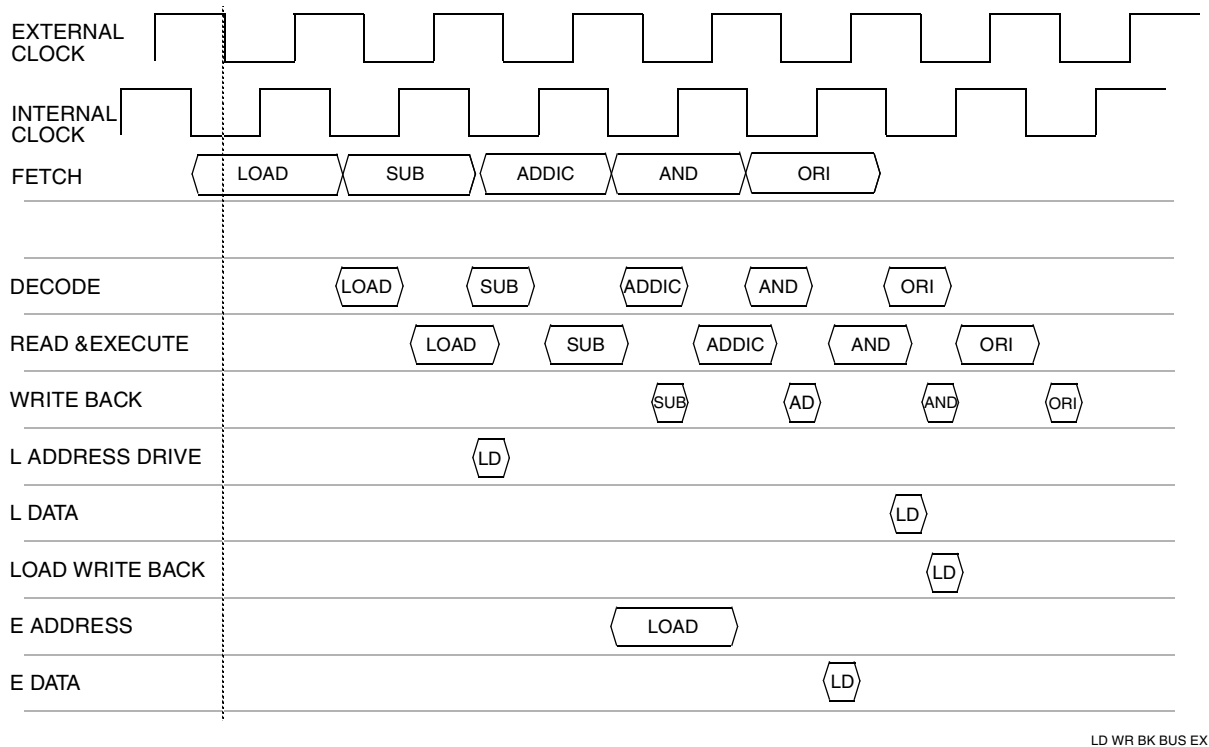
**Figure 7-7 Write-Back Arbitration Example II**

## 7.7.3 Load with Private Write-Back Bus

In this example, the **load** and the **and** write back in the same clock cycle, since they use the writeback bus during separate ticks.

```

lwz        r12,64(r0)
subf       r5,r3,r5
addic      r4,r14,1
and        r3,r4,r5
or         r6,r12,r3
    
```



**Figure 7-8 Load with Private Write-Back Bus Example**

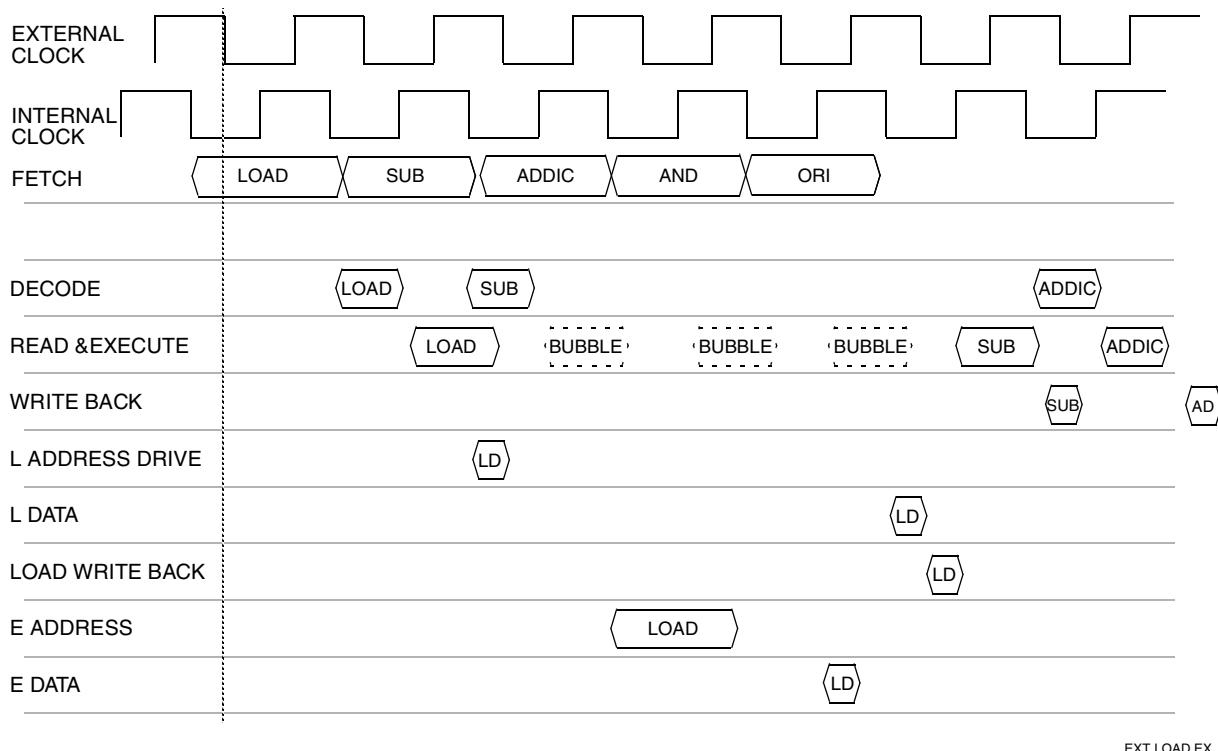
### 7.7.4 Fastest External Load Example

In this example, the **subf** is dependent on the value read by the **load**. It causes three bubbles in the instruction execution stream.

#### NOTE

The external clock is shifted 90° relative to the internal clock.

lwz	r12,64(r0)
subf	r3,r12,r3
addic	r4,r14,1



**Figure 7-9 External Load Example**

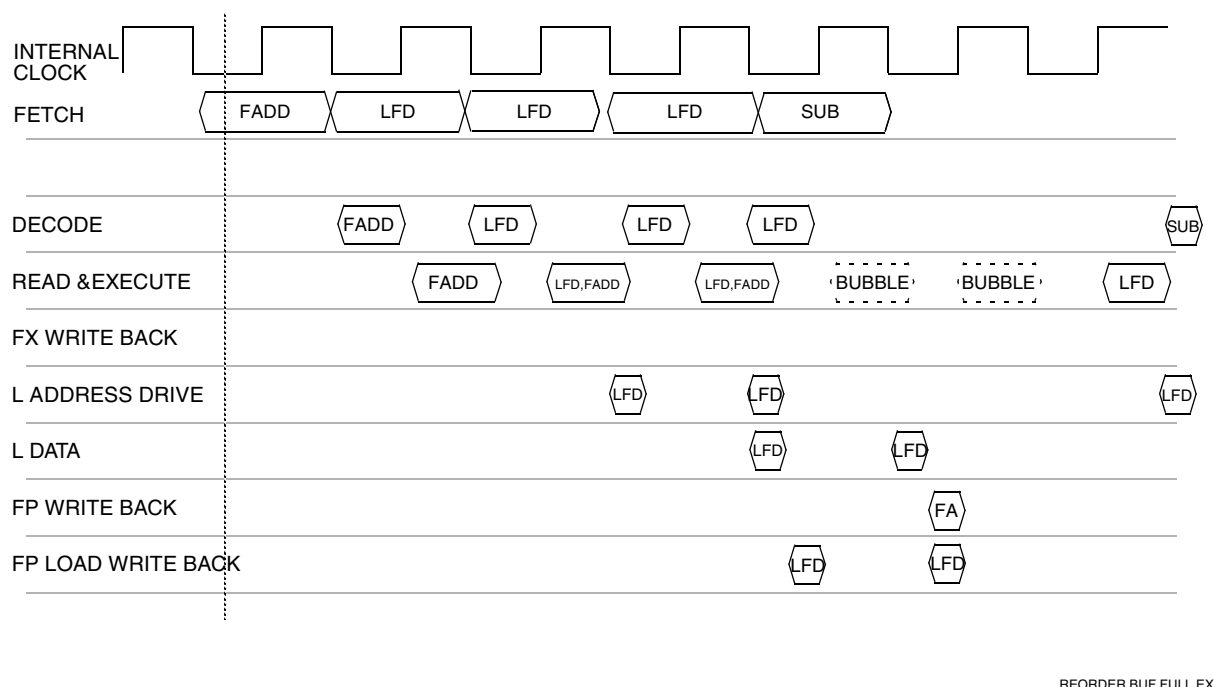
### 7.7.5 History Buffer Full Example

This example demonstrates the condition of a full history buffer. The floating-point history buffer is full by the **fadd** and two of the three **lfd**s.

#### NOTE

Following writeback of the **fadd** instruction, one additional bubble is required before instruction issue resumes. During this bubble, the history buffer retires the **fadd** instruction (as well as the two **lfd** instructions).

<b>fadd</b>	<b>fr5, fr6, fr7</b>
<b>lfd</b>	<b>fr12, 0(r2)</b>
<b>lfd</b>	<b>fr13, 8(r2)</b>
<b>lfd</b>	<b>fr14, 16(r2)</b>
<b>subf</b>	<b>r5, r3, r5</b>



**Figure 7-10 History Buffer Full Example**

### 7.7.6 Store and Floating-Point Example

In this example the **stw** access on the L-bus is delayed until the **fadd** instruction is written back.

#### NOTE

In contrast to full serialization cases, the issue and execution of following instructions continue unaffected.

fadd	fr5, fr6, fr7
stw	r12, 64 (SP)
subf	r5, r5, r3
addic	r4, r14, 1
fmul	fr3, fr4, fr5
or	r6, r12, r3

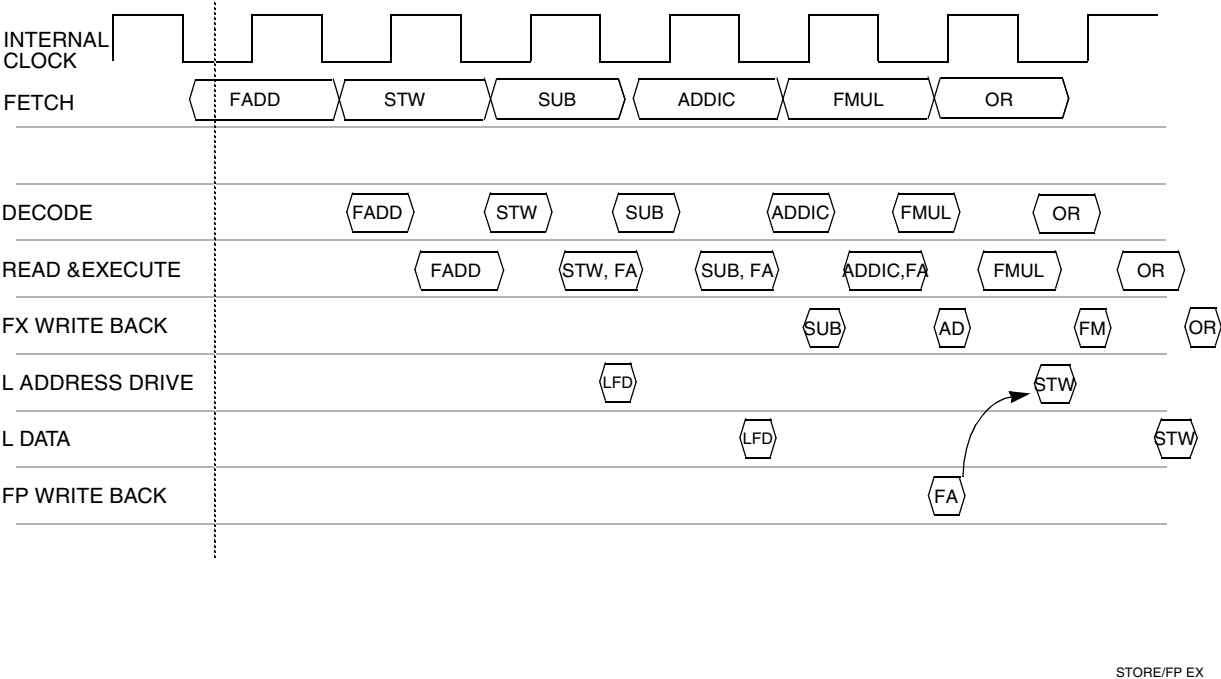


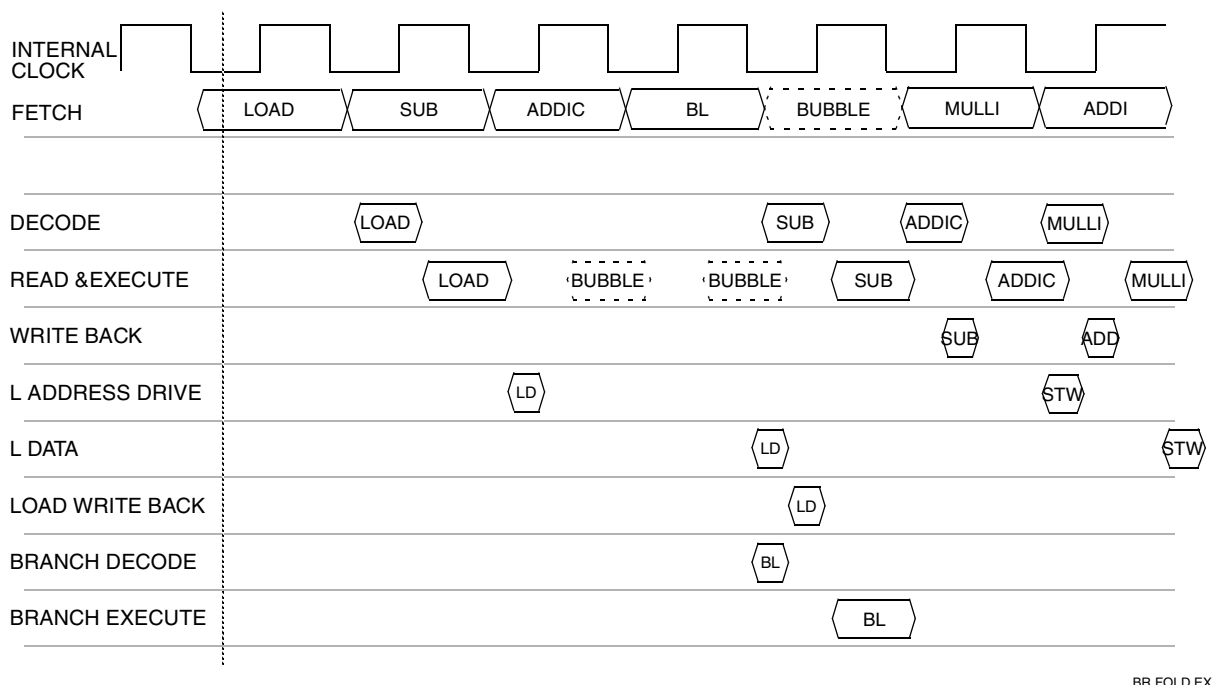
Figure 7-11 Store and Floating-Point Example

### 7.7.7 Branch Folding Example

In this example, the **lwz** instruction accesses internal storage with one wait state. The instruction prefetch queue and the parallel operation of the branch unit allow the two bubbles caused by the **bl** issue and execution to overlap the two bubbles caused by the **lwz** instruction.

```

        lwz          r12,64(SP)
        subf         r3,r12,r3
        addic        r4,r14,1
        bl           func
        ...
func:    mulli       r5,r3,3
        addi         r4,r0,3
    
```



**Figure 7-12 Branch Folding Example**

### 7.7.8 Branch Prediction Example

In this example the **blt** instruction is dependent on the **cmpi**. The branch unit still predicts the correct path and allows the bubbles caused by the **blt** instruction to overlap the bubbles caused by the **ld** instruction, as in the previous example.

When the **cmpi** instruction is written back, the branch unit re-evaluates the decision. If the branch was correctly predicted, execution continues without interruption. The fetched instructions on the predicted path are not allowed to execute before the condition is finally resolved. Instead, they are stacked in the instruction prefetch queue.

```
while:    mulli      r3,r12,4
          addi       r4,r0,3
          ...
          lwz        r12,64(r2)
          cmpi       r12,3
          addic      r6,r5,1
          blt        while
          ...
```

# Freescale Semiconductor, Inc.

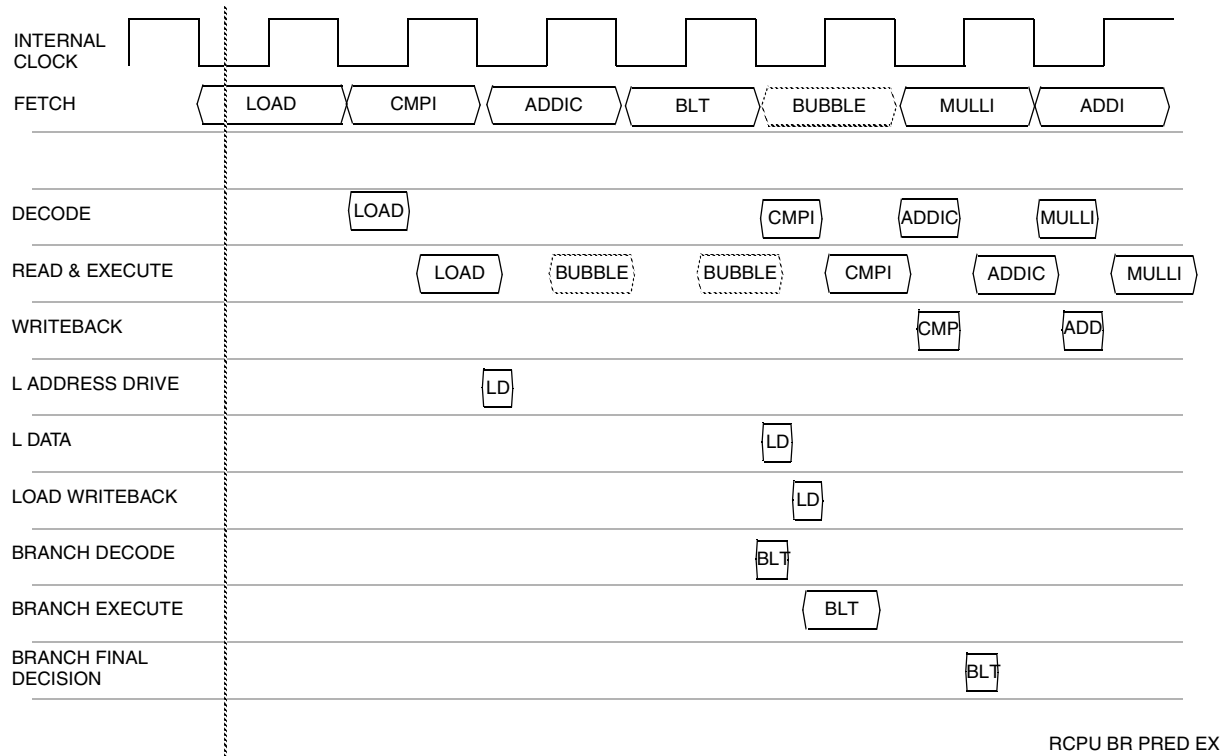


Figure 7-13 Branch Prediction Example



## **SECTION 8 DEVELOPMENT SUPPORT**

Development tools are used by a microcomputer system developer to debug the hardware and software of a target system. These tools are used to give the developer some control over the execution of the target program. In-circuit emulators and bus state analyzers are the most frequently used debugging tools. In order for these tools to function properly, they must have full visibility of the microprocessor's buses.

Visibility extends beyond the address and data portions of the buses and includes attribute and handshake signals. In some cases it may also include bus arbitration signals and signals which cause processor exceptions such as interrupts and resets. The visibility requirements of emulators and bus analyzers are in opposition to the trend of modern microcomputers and microprocessors where the CPU bus may be hidden behind a memory management unit or cache or where bus cycles to internal resources are not visible externally.

The development tool visibility requirements may be reduced if some of the development support functions are included in the silicon. For example, if the bus comparator part of a bus analyzer or breakpoint generator is included on the chip, it is not necessary for the entire bus to be visible at all times. In many cases the visibility requirements may be reduced to instruction fetch cycles for tracking program execution. If some additional status information is also available to assist in execution tracking and the development tool has access to the source code, then the only need for bus visibility is often the destination address of indirect change-of-flow instructions (return from subroutine, return from interrupt, and indexed branches and jumps).

Since full bus visibility reduces available bus bandwidth and processor performance, certain development support functions have been included in the MCU. These functions include the following:

- Controls to limit which internal bus cycles are reflected on the external bus (show cycles)
- CPU status signals to allow instruction execution tracking with minimal visibility of the instructions being fetched
- Watchpoint comparators that can generate breakpoints or signal an external bus analyzer
- A serial development port for general emulation control

### **8.1 Program Flow Tracking**

The exact program flow is visible on the external bus only when the processor is programmed to show all fetch cycles on the external bus. This mode is selected by

programming the ISCTL (instruction fetch show cycle control) field in the I-bus support control register (ICTRL), as shown in [Table 8-2](#). In this mode, the processor is fetch serialized, and all internal fetch cycles appear on the external bus. Processor performance is therefore much lower than when working in regular mode.

The mechanism described below allows tracking of the program instructions flow with almost no performance degradation. The information provided externally may be captured and compressed and then parsed by a post-processing program using the microarchitecture defined below.

The RCPU implements a prefetch queue combined with parallel, out of order, pipelined execution. Instructions progress inside the processor from fetch to retire. An instruction retires from the machine only after it, and all preceding instructions, finish execution with no exception. Therefore only retired instructions can be considered architecturally executed.

These features, together with the fact that most fetch cycles are performed internally (e.g., from the I-cache), increase performance but make it very difficult to provide the user with the real program trace.

In order to reconstruct a program trace, the program code and the following additional information from the MCU are needed:

- A description of the last fetched instruction (stall, sequential, branch not taken, branch direct taken, branch indirect taken, exception taken).
- The addresses of the targets of all indirect flow change. Indirect flow changes include all branches using the link and count registers as the target address, all exceptions, and **rfi** and **mtmsr** because they may cause a context switch.
- The number of instructions canceled each clock.

Reporting on program trace during retirement would significantly complicate the visibility support and increase the die size. (Complications arise because more than one instruction can retire in a clock cycle, and because it is harder to report on indirect branches during retirement.) Therefore, program trace is reported during fetch. Since not all fetched instructions eventually retire, an indication on canceled instructions is reported.

Instructions are fetched sequentially until branches (direct or indirect) or exceptions appear in the program flow or some stall in execution causes the machine not to fetch the next address. Instructions may be architecturally executed, or they may be canceled in some stage of the machine pipeline.

The following sections define how this information is generated and how it should be used to reconstruct the program trace. The issue of data compression that could reduce the amount of memory needed by the debug system is also mentioned.

### 8.1.1 Indirect Change-of-Flow Cycles

An *indirect change-of-flow* attribute is attached to all fetch cycles that result from indirect flow changes. Indirect flow changes include the following types of instructions or events:

- Assertion or negation of VSYNC.
- Exception taken.
- Indirect branch taken.
- Execution of the following sequential instructions: **rfi**, **isync**, **mtmsr**, and **mtspr** to CMPA–CMPF, ICTRL, ECR, and DER.

When a program trace recording is needed, the user can ensure that cycles that result from an indirect change-of-flow are visible on the external bus. The user can do this in one of two ways: by setting the VSYNC bit, or by programming the ISCTL bits in the I-bus support control register. Refer to [8.1.2 Instruction Fetch Show Cycle Control](#) for more information.

When the processor is programmed to generate show cycles on the external bus resulting from indirect change-of-flow, these cycles can generate regular bus cycles (address phase and data phase) when the instructions reside in one of the external devices, or they can generate address-only show cycles for instructions that reside in an internal device such as I-cache or internal ROM.

#### 8.1.1.1 Marking the Indirect Change-of-Flow Attribute

When an instruction fetch cycle that results from an indirect change-of-flow is an internal access (e.g., access to an internal memory location, or a cache hit during an access to an external memory address), the indirect change-of-flow attribute is indicated by the assertion (low) of the  $\overline{WR}$  pin during the external bus show cycle.

When an instruction fetch cycle that results from an indirect change-of-flow is an access to external memory not resulting in a cache hit, the indirect change-of-flow attribute is indicated by the value 0001 on the CT[0:3] pins.

**Table 8-1** summarizes the encodings that represent the indirect change-of-flow attribute. In all cases the AT1 pin is asserted (high), indicating the cycle is an instruction fetch cycle.

**Table 8-1 Program Trace Cycle Attribute Encodings**

CT[0:3]	AT1	$\overline{WR}$	Type of Bus Cycle
0001	1	1	External bus cycle
01xx, 10xx, 110x	1	0	Show cycle on the external bus reflecting an access to internal register or memory or a cache hit

Refer to [8.1.3 Program Flow-Tracking Pins](#) for more information on the use of these pins for program flow tracking.

#### 8.1.1.2 Sequential Instructions with the Indirect Change-of-Flow Attribute

Because certain sequential instructions (**rfi**, **isync**, **mtmsr**, and **mtspr** to CMPA – CMPF, ICTRL, ECR, and DER) affect the machine in a manner similar to indirect

branch instructions, the processor marks these instructions as indirect branch instructions (VF = 101, see [Table 8-3](#)) and marks the subsequent instruction address with the indirect change-of-flow attribute, as if it were an indirect branch target. Therefore, when the processor detects one of these instructions, the address of the following instruction is visible externally. This enables the reconstructing software to correctly evaluate the effect of these instructions.

### 8.1.2 Instruction Fetch Show Cycle Control

Instruction fetch show cycles are controlled by the bits in the ICTRL and the state of VSYNC, as illustrated in [Table 8-2](#).

**Table 8-2 Fetch Show Cycles Control**

VSYNC	ISCTL (Instruction Fetch Show Cycle Control Bits)	Show Cycles Generated
X	00	All fetch cycles
X	01	All change-of-flow (direct & indirect)
X	10	All indirect change-of-flow
0	11	No show cycles are performed
1	11	All indirect change-of-flow

Note that when the value of the ISCTL field is changed (with the **mtspr** instruction), the new value does not take effect until two instructions after the **mtspr** instruction. The instruction immediately following **mtspr** is under control of the old ISCTL value.

In order to keep the pin count of the chip as low as possible, VSYNC is not implemented as an external pin; rather, it is asserted and negated using the development port serial interface. For more information on this interface refer to [8.3.5 Trap-Enable Input Transmissions](#).

The assertion and negation of VSYNC forces the machine to synchronize and the first fetch after this synchronization to be marked as an indirect change-of-flow cycle and to be visible on the external bus. This enables the external hardware to synchronize with the internal activity of the processor.

When either VSYNC is asserted or the ISCTL bits in the I-bus control register are programmed to a value of 0b10, cycles resulting from an indirect change-of-flow are shown on the external bus. By programming the ISCTL bits to show all indirect flow changes, the user can thus ensure that the processor maintains exactly the same behavior when VSYNC is asserted as when it is negated. The loss of performance the user can expect from the additional external bus cycles is minimal.

For additional information on the ISCTL bits and the ICTRL register, refer to [8.8 Development Support Registers](#). For more information on the use of VSYNC during program trace, refer to [8.1.4 External Hardware During Program Trace](#).

### 8.1.3 Program Flow-Tracking Pins

The following sets of pins are used in program flow tracking:

- Instruction queue status pins (VF[0:2]) denote the type of the last fetched instruction or how many instructions were flushed from the instruction queue.
- History buffer flushes status pins (VFLS [0:1]) denote how many instructions were flushed from the history buffer during the current clock cycle.
- Address type pin 1 (AT1) indicates whether the cycle is transferring an instruction or data.
- The write/read pin ( $\overline{WR}$ ), when asserted during an instruction fetch show cycle, indicates the current cycle results from an indirect change-of-flow.
- Cycle type pins (CT[0:3]) indicate the type of bus cycle and are used to determine the address of an internal memory or register that is being accessed.

#### 8.1.3.1 Instruction Queue Status Pins

Instruction queue status pins VF[0:2] indicate the type of the last fetched instruction or how many instructions were flushed from the instruction queue. These status pins are used for both functions because queue flushes occur only during clock cycles in which there is no fetch type information to be reported.

**Table 8-3** shows the possible instruction types.

**Table 8-3 VF Pins Instruction Encodings**

VF[0:2]	Instruction Type	VF Next Clock Will Hold
000	None	More instruction type information
001	Sequential	More instruction type information
010	Branch (direct or indirect) <b>not</b> taken	More instruction type information
011	VSYNCR was asserted/negated and therefore the next instruction will be marked with the indirect change-of-flow attribute	More instruction type information
100	Exception taken — the target will be marked with the program trace cycle attribute	Queue flush information <sup>1</sup>
101	Branch indirect taken, <b>rfi</b> , <b>mtmsr</b> , <b>isync</b> and in some cases <b>mtspr</b> to CMPA-F, ICTRL, ECR, or DER — the target will be marked with the indirect change-of-flow attribute <sup>2</sup>	Queue flush information <sup>1</sup>
110	Branch direct taken	Queue flush information <sup>1</sup>
111	Branch (direct or indirect) <b>not</b> taken	Queue flush information <sup>1</sup>

NOTES:

1. Unless next clock VF=111. See below.
2. The sequential instructions listed here affect the machine in a manner similar to indirect branch instructions. Refer to **8.1.1.2 Sequential Instructions with the Indirect Change-of-Flow Attribute**.

**Table 8-4** shows VF[0:2] encodings for instruction queue flush information.

**Table 8-4 VF Pins Queue Flush Encodings**

VF[0:2]	Queue Flush Information
000	0 instructions flushed from instruction queue
001	1 instruction flushed from instruction queue
010	2 instructions flushed from instruction queue
011	3 instructions flushed from instruction queue
100	4 instructions flushed from instruction queue
101	5 instructions flushed from instruction queue
110	Reserved
111	Instruction type information <sup>1</sup>

## NOTES:

1. Refer to [Table 8-3](#).

There is one special case in which although queue flush information is expected on the VF[0:2] pins (according to the immediately preceding value on these pins), regular instruction type information is reported. The only instruction type information that can appear in this case is VF[0:2] = 111, indicating branch (direct or indirect) not taken. Since the maximum queue flushes possible is five, identifying this special case is not a problem.

**8.1.3.2 History Buffer Flush Status Pins**

History buffer flush status pins VFLS[0:1] indicate how many instructions are flushed from the history buffer this clock. [Table 8-4](#) shows VFLS encodings.

**Table 8-5 VFLS Pin Encodings**

VFLS[0:1]	History Buffer Flush Information
00	0 instructions flushed from history queue
01	1 instruction flushed from history queue
10	2 instructions flushed from history queue
11	Used for debug mode indication (FREEZE). Program trace external hardware should ignore this setting.

**8.1.3.3 Flow-Tracking Status Pins in Debug Mode**

When the processor is in debug mode, the VF[0:2] signals are low (000) and the VFLS[0:1] signals are high (11).

If VSYNC is asserted or negated while the processor is in debug mode, this information is reported as the first VF pins report when the processor returns to regular

mode. If VSYNC was not changed while the processor is in debug mode, the first VF pins report is of an indirect branch taken (VF[0:2] = 101), appropriate for the **rfi** instruction that is being issued. In both cases, the first instruction fetch after debug mode is marked with the program trace cycle attribute and therefore is visible externally.

## 8.1.3.4 Cycle Type, Write/Read, and Address Type Pins

Cycle type pins (CT[0:3]) indicate the type of bus cycle being performed. During show cycles, these pins are used to determine the internal address being accessed. **Table 8-6** summarizes cycle type encodings.

**Table 8-6 Cycle Type Encodings**

CT[0:3]	Description
0000	Normal external bus cycle
0001	If address type is data (AT1 = 0), this is a data access to the external bus and the start of a reservation. If address type is instruction (AT1=1), this cycle type indicates that an external address is the destination of an indirect change-of-flow.
0010	External bus cycle to emulation memory replacing internal I-bus or L-bus memory. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0011	Normal external bus cycle access to a port replacement chip used for emulation support.
0100	Access to internal I-bus memory. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0101	Access to internal L-bus memory. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0110	Cache hit on external memory address not controlled by chip selects. An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
0111	Access to an internal register.
1000	Cache hit on external memory address controlled by $\overline{CSBOOT}$ .
1001	Cache hit on external memory address controlled by $\overline{CS1}$ .
1010	Cache hit on external memory address controlled by $\overline{CS2}$ .
1011	Cache hit on external memory address controlled by $\overline{CS3}$ .
1100	Cache hit on external memory address controlled by $\overline{CS4}$ .
1101	Cache hit on external memory address controlled by $\overline{CS5}$ .
	An instruction access (AT1 = 1) with an address that is the target of an indirect change-of-flow is indicated as a logic level zero on the $\overline{WR}$ output.
1110	Reserved
1111	



Notice in [Table 8-6](#) that during an instruction fetch ( $AT1 = 1$ ) to internal memory or to external memory resulting in a cache hit, a logic level of zero on the  $\overline{WR}$  pin indicates that the cycle is the result of an indirect change-of-flow. The indirect change-of-flow attribute is also indicated by a cycle type encoding of 0001 when  $AT1 = 1$ . Refer to [8.1.1.1 Marking the Indirect Change-of-Flow Attribute](#) for additional information.

## 8.1.4 External Hardware During Program Trace

When program trace is needed, external hardware needs to record the status pins ( $VF[0:2]$  and  $VFLS[0:1]$ ) of each clock and record the address of all cycles marked with the indirect change-of-flow attribute.

Program trace can be used in various ways. Two types of traces that can be implemented are the back trace and the window trace.

### 8.1.4.1 Back Trace

A back trace provides a record of the program trace *before* some event occurred. An example of such an event is some system failure.

When a back trace is needed, the external hardware should start sampling the status pins and the address of all cycles marked with the indirect change-of-flow attribute immediately after reset is negated. Since the ISCTL field in the ICTRL has a value of 0b00 (show all cycles) out of reset, all cycles marked with the indirect change-of-flow attribute are visible on the external bus. VSYNC should be asserted sometime after reset and negated when the programmed event occurs. VSYNC must be asserted before the ISCTL encoding is changed to 0b11 (no show cycles), if such an encoding is selected.

Note that in case the timing of the programmed event is unknown, it is possible to use cyclic buffers.

After VSYNC is negated, the trace buffer will contain the program flow trace of the program executed before the programmed event occurred.

### 8.1.4.2 Window Trace

Window trace provides a record of the program trace *between* two events. VSYNC should be asserted between these two events.

After VSYNC is negated, the trace buffer will contain information describing the program trace of the program executed between the two events.

### 8.1.4.3 Synchronizing the Trace Window to Internal CPU Events

In order to synchronize the assertion or negation of VSYNC to an event internal to the processor, internal breakpoints can be used together with debug mode. This method is available only when debug mode is enabled. (Refer to [8.4 Debug Mode Functions](#).)

The following steps enable the user to synchronize the trace window to events in-



ternal to the processor:

1. Enter debug mode, either immediately out of reset or using the debug mode request.
2. Program the hardware to break on the event that marks the start of the trace window using the control registers defined in **8.8 Development Support Registers**.
3. Enable debug mode entry for the programmed breakpoint in the debug enable register (DER).
4. Return to the regular code run.
5. The hardware generates a breakpoint when the programmed event is detected, and the machine enters debug mode.
6. Program the hardware to break on the event that marks the end of the trace window.
7. Assert VSYNC.
8. Return to the regular code run. The first report on the VF pins is a VSYNC (VF[0:2] = 011).
9. The external hardware starts sampling the program trace information upon the report on the VF pins of VSYNC.
10. The hardware generates a breakpoint when the programmed event is detected, and the machine enters debug mode.
11. Negate VSYNC.
12. Return to the regular code run. The first report on the VF pins is a VSYNC (VF[0:2] = 011).
13. The external hardware stops sampling the program trace information upon the report on the VF pins of VSYNC.

A second method allows the trace window to be synchronized to internal processor events without stopping execution and entering debug mode at the two events.

1. Enter debug mode, either immediately out of reset or using the debug mode request.
2. Program a watchpoint for the event that marks the start of the trace window using the control registers defined in **8.8 Development Support Registers**.
3. Program a second watchpoint for the event that marks the end of the trace window.
4. Return to regular code execution by exiting debug mode.
5. The watchpoint logic signals the starting event by asserting the appropriate watchpoint pin.
6. Upon detecting the first watchpoint, assert VSYNC using the development port serial interface.
7. The external program trace hardware starts sampling the program trace information upon the report on the VF pins of VSYNC.
8. The watchpoint logic signals the ending event by asserting the appropriate watchpoint pin.
9. Upon detecting the second watchpoint, negate VSYNC using the development port serial interface.
10. The external program trace hardware stops sampling the program trace information upon the report on VF[0:1] of VSYNC.

The second method is not as precise as the first method because of the delay between the assertion of the watchpoint pins and the assertion or negation of VSYNC using the development port serial interface. It has the advantage, however, of allowing the program to run in quasi-real time (slowed only by show cycles on the external bus), instead of stopping execution at the starting and ending events.

#### 8.1.4.4 Detecting the Trace Window Starting Address

For a back trace, the value of the status pins (VF[0:2] and VFLS[0:1]) and the address of the cycles marked with the indirect change-of-flow attribute should be latched starting immediately after the negation of reset. The starting address is the first address in the program trace cycle buffer.

For a window trace, the value of the status pins and the address of the cycles marked with the indirect change-of-flow attribute should be latched beginning immediately after the first VSYNC is reported on the VF pins. The starting address of the trace window should be calculated according to the first two VF pin reports.

Assume VF1 and VF2 are the two first VF pin reports and T1 and T2 are the addresses of the first two cycles marked with the indirect change-of-flow attribute that were latched in the trace buffer. Use [Table 8-7](#) to calculate the trace window starting address.

**Table 8-7 Detecting the Trace Buffer Starting Point**

VF1	VF2	Starting Point	Description
011 VSYNC	001 Sequential	T1	VSYNC asserted followed by a sequential instruction. The starting address is T1.
011 VSYNC	110 Branch direct taken	$T1 - 4 + \text{offset}(T1 - 4)$	VSYNC asserted followed by a taken direct branch. The starting address is the target of the direct branch.
011 VSYNC	101 Branch indirect taken	T2	VSYNC asserted followed by a taken indirect branch. The starting address is the target of the indirect branch.

#### 8.1.4.5 Detecting the Assertion or Negation of VSYNC

Since the VF pins are used for reporting both instruction type information and queue flush information, the external hardware must take special care when trying to detect the assertion or negation of VSYNC. A VF[0:2] encoding of 011 indicates the assertion or negation of VSYNC only if the previous VF[0:2] pin values were 000, 001, or 010.

#### 8.1.4.6 Detecting the Trace Window Ending Address

The information on the VF and VFLS status pins changes every clock. Cycles marked with the indirect change-of-flow are generated on the external bus only when possible (when the SIU wins the arbitration over the external bus). Therefore,

there is some delay between the time it is reported on the status pins that a cycle marked as program trace cycle will be performed on the external bus and the actual time that this cycle can be detected on the external bus.

When the user negates VSYNC, the processor delays the report of the assertion or negation of VSYNC on the VF pins until all addresses marked with the indirect change-of-flow attribute have been made visible externally. Therefore, the external hardware should stop sampling the value of the status pins (VF and VFLS) and the address of the cycles marked with the program trace cycle attribute immediately after the VSYNC report on the VF pins.

### CAUTION

The last two instructions reported on the VF pins are not always valid. Therefore, at the last stage of the reconstruction software, the last two instructions should be ignored.

#### 8.1.5 Compress

In order to store all the information generated on the pins during program trace (5 bits per clock + 30 bits per show cycle) a large memory buffer may be needed. However, since this information includes events that were canceled, compression can be very effective. External hardware can be added to eliminate all canceled instructions and report only on branches (taken and not taken), indirect flow change, and the number of sequential instructions after the last flow change.

#### 8.2 Watchpoint and Breakpoint Support

The RCPU provides the ability to detect specific bus cycles, as defined by a user (watchpoints). It also provides the ability to conditionally respond to these watchpoints by taking an exception (internal breakpoints). Breakpoints can also be caused by an event or state in a peripheral or through the development port (external breakpoints, (i.e., breakpoints external to the processor)).

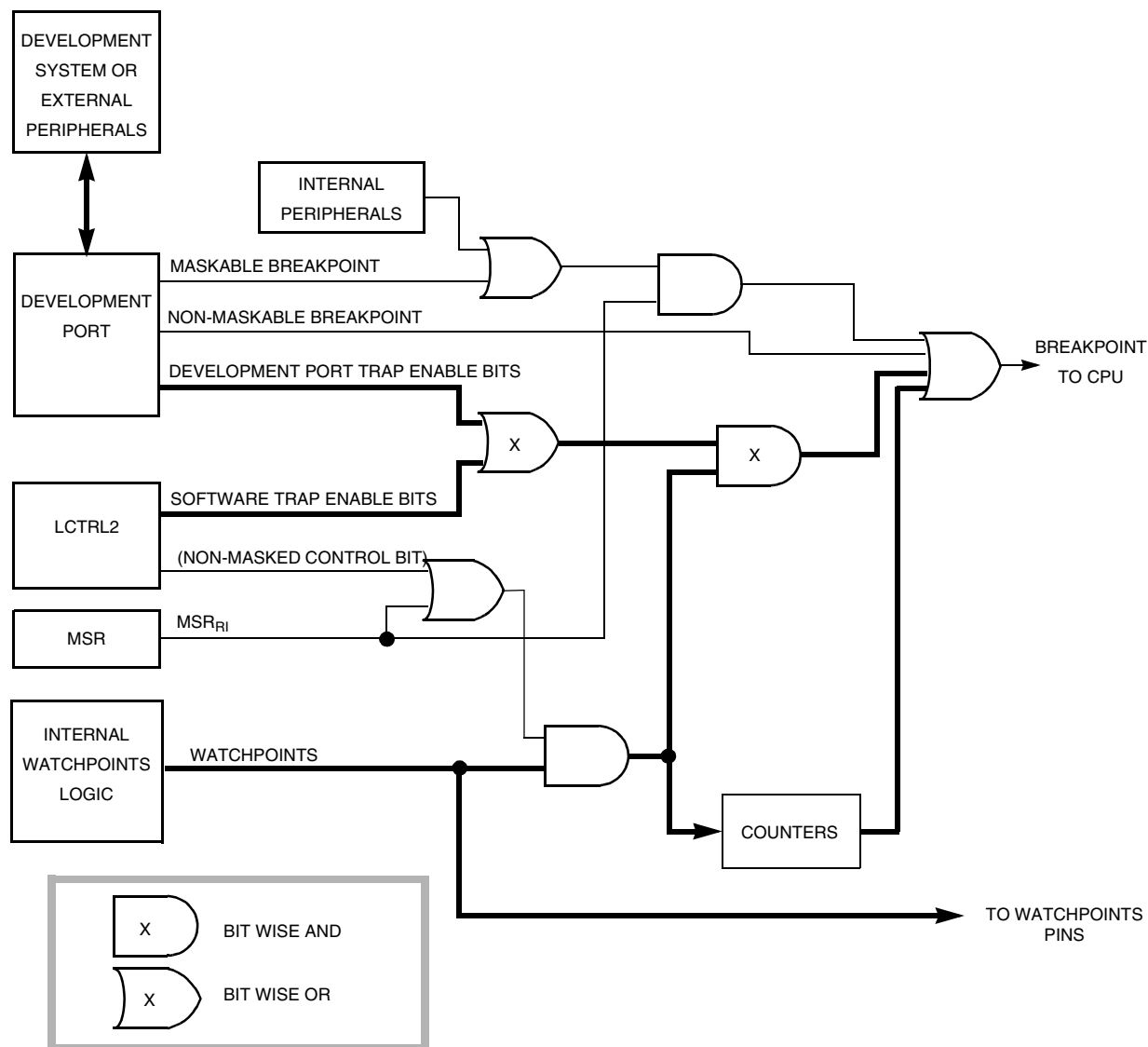
When a watchpoint is detected, it is reported to external hardware on dedicated pins. Watchpoints do not change the timing or flow of the processor. Because bus cycles on the internal MCU buses are not necessarily visible on the external bus, the watchpoints are a convenient way to signal an external instrument (such as a bus state analyzer or oscilloscope) that the internal bus cycle occurred.

An internal breakpoint occurs when a particular watchpoint is enabled to generate a breakpoint. A watchpoint may be enabled to generate a breakpoint from a software monitor or by using the development port serial interface. A watchpoint output may also be counted. When the counter reaches zero, an internal breakpoint is generated.

An external breakpoint occurs when a development system or external peripheral requests a breakpoint through the development port serial interface. In addition, if an on-chip peripheral requests a breakpoint, an external breakpoint is generated.

All internal breakpoints are masked by the MSR[RI] bit unless the non-masked control bit (BRKNOMSK) in LCTRL2 is set. The development port maskable breakpoint and breakpoints from internal peripherals are masked by the MSR[RI] bit. The development port non-maskable breakpoint is *not* masked by this bit.

**Figure 8-1** is a diagram of watchpoint and breakpoint support in the RCPU.



WATCH/BREAK SUPPORT

**Figure 8-1 Watchpoint and Breakpoint Support in the RCPU**

## 8.2.1 Watchpoints

Watchpoints are based on eight comparators on the I-bus and L-bus, two counters, and two AND-OR logic structures. There are four comparators on the instruction address bus (I-address), two comparators on the load/store address bus (L-address), and two comparators on the load/store data bus (L-data).

The comparators are able to detect the following conditions: equal, not equal, greater than, and less than. Greater than or equal and less than or equal are easily obtained from these four conditions. (For more information refer to **8.2.1.3 Generating Six Compare Types**.) Using the AND-OR logic structures, in range and out of range detection (on address and on data) are supported. Using the counters, it is possible to program a breakpoint to be generated after an event is detected a predefined number of times.

The L-data comparators can operate on integer data, floating-point single-precision data, and the integer value stored using the **stfiwx** instruction. Integer comparisons can be performed on bytes, half words, and words. The operands can be treated as signed or unsigned values.

The comparators generate match events. The I-bus match events enter the I-bus AND-OR logic, where the I-bus watchpoints and breakpoint are generated. When asserted, the I-bus watchpoints may generate the I-bus breakpoint. Two of them may decrement one of the counters. When a counter that is counting one of the I-bus watchpoints expires, the I-bus breakpoint is asserted.

The I-bus watchpoints and the L-bus match events (address and data) enter the L-bus AND-OR logic where the L-bus watchpoints and breakpoint are generated. When asserted, the L-bus watchpoints may generate the L-bus breakpoint, or they may decrement one of the counters. When a counter that is counting one of the L-bus watchpoints expires, the L-bus breakpoint is asserted.

L-bus watchpoints can be qualified by I-bus watchpoints. If qualified, the L-bus watchpoint occurs only if the L-bus cycle was the result of executing an instruction that caused the qualifying I-bus watchpoint.

A watchpoint progresses in the machine along with the instruction that caused it (fetch or load/store cycle). Watchpoints are reported on the external pins when the associated instruction is retired.

### **8.2.1.1 Restrictions on Watchpoint Detection**

There are cases when the same watchpoint can be detected more than once during the execution of a single instruction. For example, the processor may detect an L-bus watchpoint on more than one transfer when executing a load/store multiple or string instruction or may detect an L-bus watchpoint on more than one byte when working in byte mode. In these cases only one watchpoint of the same type is reported for a single instruction. Similarly, only one watchpoint of the same type can be counted in the counters for a single instruction.

Since watchpoint events are reported upon the retirement of the instruction that caused the event, and more than one instruction can retire from the machine in one clock, separate watchpoint events may be reported in the same clock. Moreover, the same event, if detected on more than one instruction (e.g., tight loops, range detection), in some cases is reported only once. However, the internal counters still count correctly.

## 8.2.1.2 Byte and Half-Word Working Modes

Watchpoint and breakpoint support enables the user to detect matches on bytes and half words even when accessed using a load/store instruction of larger data widths, e.g. when loading a table of bytes using a series of load word instructions.

To use this feature the user needs to program the byte mask for each of the L-data comparators and to write the needed match value to the correct half word of the data comparator when working in half word mode and to the correct bytes of the data comparator when working in byte mode.

Since bytes and half words can be accessed using a larger data width instruction, the user cannot predict the exact value of the L-address lines when the requested byte or half word is accessed. For example, if the matched byte is byte two of the word and it is accessed using a load word instruction, the L-address value will be of the word (byte zero). Therefore the processor masks the two least significant bits of the L-address comparators whenever a word access is performed and the least significant bit whenever a half word access is performed. Address range is supported only when aligned according to the access size.

The following examples illustrate how to detect matches on bytes and half words.

1. A fully supported scenario:

Looking for:

Data size: Byte

Address: 0x0000 0003

Data value: greater than 0x07 and less than 0x0C

Programming option:

One L-address comparator = 0x0000 0003 and program for equal

One L-data comparator = 0xFFFF XXX7 and program for greater than

One L-data comparator = 0xFFFF XXXC and program for less than

Both byte masks = 0b0001

Both L-data comparators program to byte mode

Result: The event will be detected regardless of the instruction the compiler chooses for this access

2. A fully supported scenario:

Looking for:

Data size: Half word

Address: greater than 0x0000 0000 and less than 0x0000 000C

Data value: greater than 0x4E20 and less than 0x9C40

Programming option:

One L-address comparator = 0x0000 0000 and program for greater than

One L-address comparator = 0x0000 000C and program for less than

One L-data comparator = 0x4E20 4E20 and program for greater than

One L-data comparator = 0x9C40 9C40 and program for less than

Both byte masks = 0b1111

Both L-data comparators program to half word mode

Result: The event will be detected correctly provided that the compiler does not use a load/store instruction with data size of byte.

## 3. A partially supported scenario:

Looking for:

Data size: Half word

Address: greater than 0x0000 0002 and less than 0x0000 000E

Data value: greater than 0x4E20 and less than 0x9C40

Programming option:

One L-address comparator = 0x0000 0002 and program for greater than

One L-address comparator = 0x0000 000E and program for less than

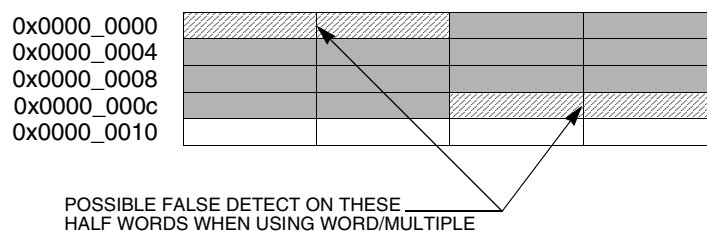
One L-data comparator = 0x4E20 4E20 and program for greater than

One L-data comparator = 0x9C40 9C40 and program for less than

Both byte masks = 0b1111

Both L-data comparators program to half word mode or to word mode

Result: The event will be detected correctly if the compiler chooses a load/store instruction with data size of half word. If the compiler chooses load/store instructions with data size greater than half word (word, multiple), there might be some false detections. These can be ignored only by the software that handles the break-points. **Figure 8-2** illustrates this partially supported scenario.



WATCH/BREAK EXAMPLE

**Figure 8-2 Partially Supported Watchpoint/Breakpoint Example****8.2.1.3 Generating Six Compare Types**

Using the four basic compare types (equal, not equal, greater than, less than), it is possible to generate two additional compare types: “greater than or equal” and “less than or equal.”

The “greater than or equal” compare type can be generated using the greater than compare type and programming the comparator to the needed value minus one.

The “less than or equal” compare type can be generated using the less than compare type and programming the comparator to the needed value plus one.

This method does not work for the following boundary cases:

- Less than or equal of the largest unsigned number (1111...1)
- Greater than or equal of the smallest unsigned number (0000...0)

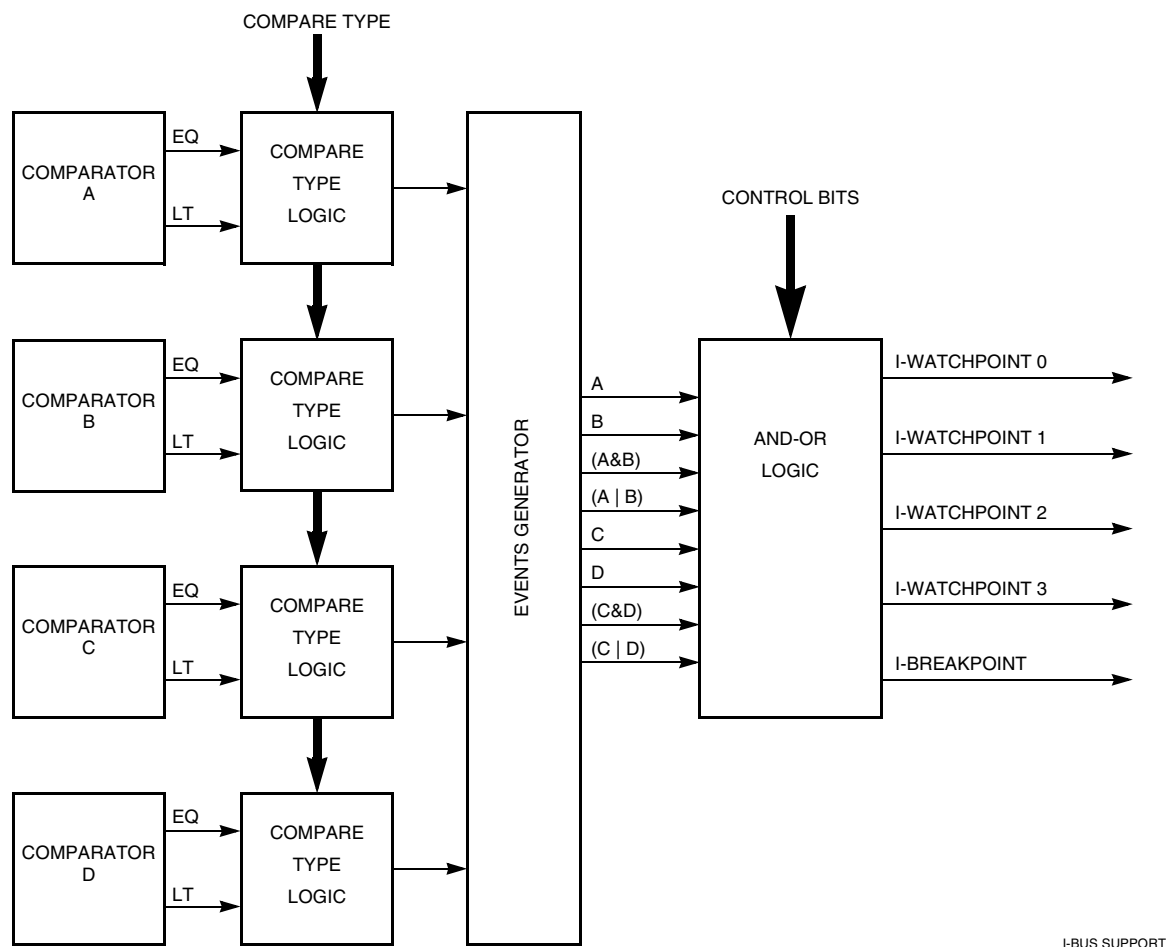


- Less than or equal of the maximum positive number when in signed mode (0111...1)
- Greater than or equal of the maximum negative number when in signed mode (1000...)

These boundary cases need no special support because they all mean “always true” and can be programmed using the ignore option of the L-bus watchpoint programming (refer to [8.8 Development Support Registers](#)).

## 8.2.1.4 I-Bus Support Detailed Description

There are four I-bus address comparators (comparators A,B,C,D). Each is 30 bits long and generates two output signals: equal and less than. These signals are used to generate one of the following four events: equal, not equal, greater than, less than. [Figure 8-3](#) shows the general structure of I-bus support.



**Figure 8-3 I-Bus Support General Structure**

The I-bus watchpoints and breakpoint are generated using these events and according to the user's programming of the CMPA, CMPB, CMPC, CMPD, and IC-



TRL registers. [Table 8-8](#) shows how watchpoints are determined from the programming options. Note that using the OR option enables “out of range” detection.

**Table 8-8 I-bus Watchpoint Programming Options**

Name	Description	Programming Options
IW0	First I-bus watchpoint	Comparator A Comparators (A&B)
IW1	Second I-bus watchpoint	Comparator B Comparator (A   B)
IW2	Third I-bus watchpoint	Comparator C Comparators (C&D)
IW3	Fourth I-bus watchpoint	Comparator D Comparator (C   D)

#### 8.2.1.5 L-Bus Support Detailed Description

There are two L-bus address comparators (comparators E and F). Each compares the 32 address bits and the cycle’s read/write attribute. The two least significant bits are masked (ignored) whenever a word is accessed, and the least significant bit is masked whenever a half word is accessed. (For more information refer to [8.2.1.2 Byte and Half-Word Working Modes](#)). Each comparator generates two output signals: equal and less than. These signals are used to generate one of the following four events (one from each comparator): equal, not equal, greater than, less than.

There are two L-bus data comparators (comparators G and H). Each is 32 bits wide and can be programmed to treat numbers either as signed values or as unsigned values. Each data comparator operates as four independent byte comparators. Each byte comparator has a mask bit and generates two output signals, equal and less than, if the mask bit is not set. Therefore, each 32-bit comparator has eight output signals.

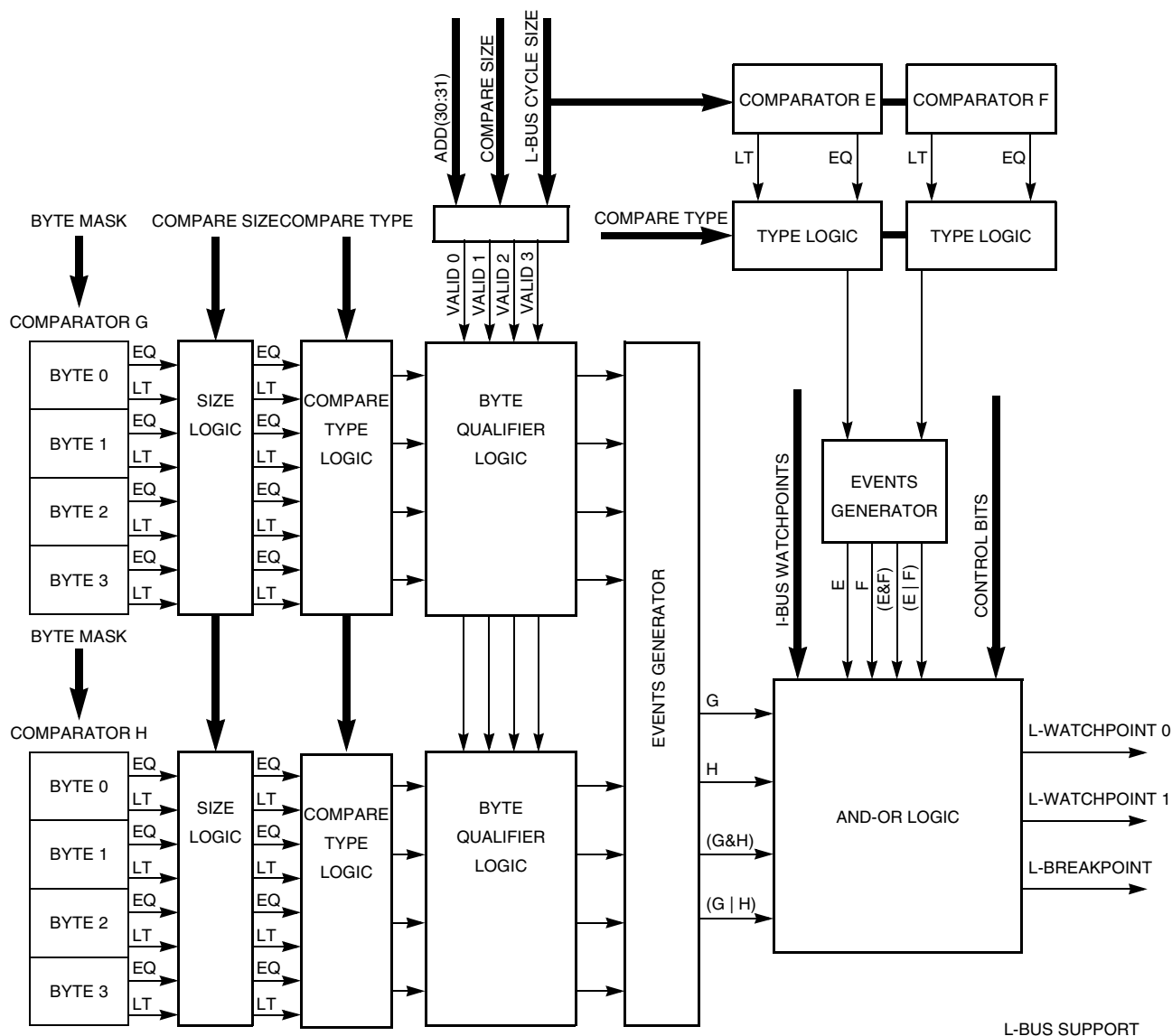
These signals are used to generate the “equal and less than” signals according to the compare size programmed by the user (byte, half word, word). In byte mode all signals are significant. In half word mode only four signals from each 32-bit comparator are significant. In word mode only two signals from each 32-bit comparator are significant.

From the new “equal and less than” signals, depending on the compare type programmed by the user, one of the following four match events is generated: equal, not equal, greater than, less than. Therefore from the two 32-bit comparators, eight match indications are generated: Gmatch[0:3], Hmatch[0:3].

According to the lower bits of the address and the size of the cycle, only match indications that were detected on bytes that have valid information are validated; the

rest are negated. Note that if the cycle executed has a smaller size than the compare size (e.g., a byte access when the compare size is word or half word) no match indication is asserted.

**Figure 8-4** shows the general structure of L-bus support.



**Figure 8-4 L-Bus Support General Structure**

Using the match indication signals, four L-bus data events are generated as shown in **Table 8-9**.

**Table 8-9 L-Bus Data Events**

Event name	Event Function <sup>1</sup>
G	(Gmatch0   Gmatch1   Gmatch2   Gmatch3)
H	(Hmatch0   Hmatch1   Hmatch2   Hmatch3)
(G&H)	((Gmatch0 & Hmatch0)   (Gmatch1 & Hmatch1)   (Gmatch2 & Hmatch2)   (Gmatch3 & Hmatch3))
(G   H)	((Gmatch0   Hmatch0)   (Gmatch1   Hmatch1)   (Gmatch2   Hmatch2)   (Gmatch3   Hmatch3))

NOTES:

1. '&' denotes a logical AND, '|' denotes a logical OR

The four L-bus data events together with the match events of the L-bus address comparators and the l-bus watchpoints are used to generate the L-bus watchpoints and breakpoint according to the user's programming of the CMPE, CMPF, CMPG, CMPH, LCTRL1, and LCTRL2 registers. **Table 8-10** shows how the watchpoints are determined from the programming options.

**Table 8-10 L-Bus Watchpoints Programming Options**

Name	Description	I-bus events programming options	L-address events programming options	L-data events programming options
LW0	First L-bus watchpoint	IW0, IW1, IW2, IW3 or don't care	Comparator E Comparator F Comparators (E&F) Comparators (E   F) or don't care	Comparator G Comparator H Comparators (G&H) Comparators (G   H) or don't care
LW1	Second L-bus watchpoint	IW0, IW1, IW2, IW3 or don't care	Comparator E Comparator F Comparators (E&F) Comparators (E   F) or don't care	Comparator G Comparator H Comparators (G&H) Comparators (G   H) or don't care

### 8.2.1.6 Treating Floating-Point Numbers

The data comparators can detect match events on floating-point single precision values in floating point load/store instructions. When floating point values are compared, the comparators must be programmed to operate in signed word mode.

During the execution of a load/store instruction of a floating-point double operand, the L-data comparators never generate a match. If L-data events are programmed for don't care (i.e., LCTRL2[LWOLADC] = 0), L-bus watchpoint and breakpoint events can be generated from the L-address events, even if the instruction is a load/store double instruction.

## 8.2.2 Internal Breakpoints

Internal breakpoints are generated from the watchpoints. The user may enable a watchpoint to create a breakpoint by setting the associated software trap enable bit in the ICTRL or LCTRL2 register. This can be done by a software monitor program executed by the MCU. An external development tool can also enable internal breakpoints from watchpoints by setting the associated development port trap enable bit using the development port serial interface.

Internal breakpoints can also be generated by assigning a breakpoint counter to a particular watchpoint. The counter counts down for each watchpoint, and a breakpoint is generated when the counter reaches zero.

An internal breakpoint progresses in the machine along with the instruction that caused it (fetch or load/store cycle). When a breakpoint reaches the top of the history buffer, the machine processes the breakpoint exception.

An instruction that causes an I-bus breakpoint is not retired. The processor branches to the breakpoint exception routine *before* it executes the instruction. An instruction that causes an L-bus breakpoint is executed. The processor branches to the breakpoint exception routine *after* it executes the instruction. The address of the load/store cycle that generated the L-bus breakpoint is stored in the breakpoint address register (BAR).

### 8.2.2.1 Breakpoint Counters

There are two 16-bit down counters. Each counter is able to count one of the I-bus watchpoints or one of the L-bus watchpoints. Both generate the corresponding breakpoint when they reach zero. If the instruction associated with the watchpoint is not retired, the counter is adjusted back so that it reflects actual execution.

In the masked mode, the counters do not count watchpoints detected when  $MSR[RI] = 0$ . See [8.2.4 Breakpoint Masking](#).

When counting watchpoints programmed on the actual instructions that alter the counters, the counters will have unpredictable values. A **sync** instruction should be inserted before a read of an active counter.

### 8.2.2.2 Trap-Enable Programming

The trap enable bits can be programmed by regular, supervisor-level software (by writing to the ICTRL or LCTRL2 with the **mtspr** instruction) or “on the fly” using the development port interface. For more information on the latter method, refer to [8.3.5 Trap-Enable Input Transmissions](#).

The value used by the breakpoints generation logic is the bit-wise OR of the software trap enable bits (the bits written using the **mtspr**) and the development port trap enable bits (the bits serially shifted using the development port).

All bits, the software trap-enable bits and the development port trap enable bits, can be read from ICTRL and the LCTRL2 using **mfspir**. For the exact bits placement refer to [Table 8-30](#) and [Table 8-32](#).

### 8.2.2.3 Ignore First Match

In order to facilitate the debugger utilities of “continue” and “go from x”, the option to ignore the first match is supported for the I-bus breakpoints. When an I-bus breakpoint is first enabled (as a result of the first write to the I-bus support control register or as a result of the assertion of the MSR[RI] bit in masked mode), the first instruction will not cause an I-bus breakpoint if the IFM (ignore first match) bit in the I-bus support control register (ICTRL) is set (used for “continue”). This allows the processor to be stopped at a breakpoint and then later to “continue” from that point without the breakpoint immediately stopping the processor again before executing the first instruction.

When the IFM bit is cleared, every matched instruction can cause an I-bus breakpoint (used for “go from x,” where x is an address that would not cause a breakpoint).

The IFM bit is set by the software and cleared by the hardware after the first I-bus breakpoint match is ignored.

Since L-bus breakpoints are treated after the instruction is executed, L-bus breakpoints and counter-generated I-bus breakpoints are not affected by this mode.

### 8.2.3 External Breakpoints

Breakpoints external to the processor can come from either an on-chip peripheral or from the development port. For additional information on breakpoints from on-chip peripherals, consult the user’s manual for the microcontroller of interest or the reference manual for the peripheral of interest.

The development port serial interface can be used to assert either a maskable or non-maskable breakpoint. Refer to **8.3.5 Trap-Enable Input Transmissions** for more information about generating breakpoints from the development port. The development port breakpoint bits remain asserted until they are cleared; however, they cause a breakpoint only when they change from cleared to set. If they remain set, they do not cause an additional breakpoint until they are cleared and set again.

External breakpoints are not referenced to any particular instruction; they are referenced to the current or following L-bus cycle. The breakpoint is taken as soon as the processor completes an instruction that uses the L-bus.

### 8.2.4 Breakpoint Masking

The processor responds to two different types of breakpoints. The maskable breakpoint is taken only if the processor is in a recoverable state. This means that taking the breakpoint will not destroy any of the internal machine context. The processor is defined to be in a recoverable state when the MSR[RI] (recoverable exception) bit is set. Maskable breakpoints are generated by the internal breakpoint logic, modules on the IMB2, and the development port.

Non-maskable breakpoints cause the processor to stop without regard to the state of the MSR[RI] bit. If the processor is in a non-recoverable state when the break-

## Freescale Semiconductor, Inc.

point occurs, the state of the SRR0, SRR1, and the DAR may have been overwritten by the breakpoint. It will not be possible to restart the processor, since the restart address and MSR context may not be available in SRR0 and SRR1.

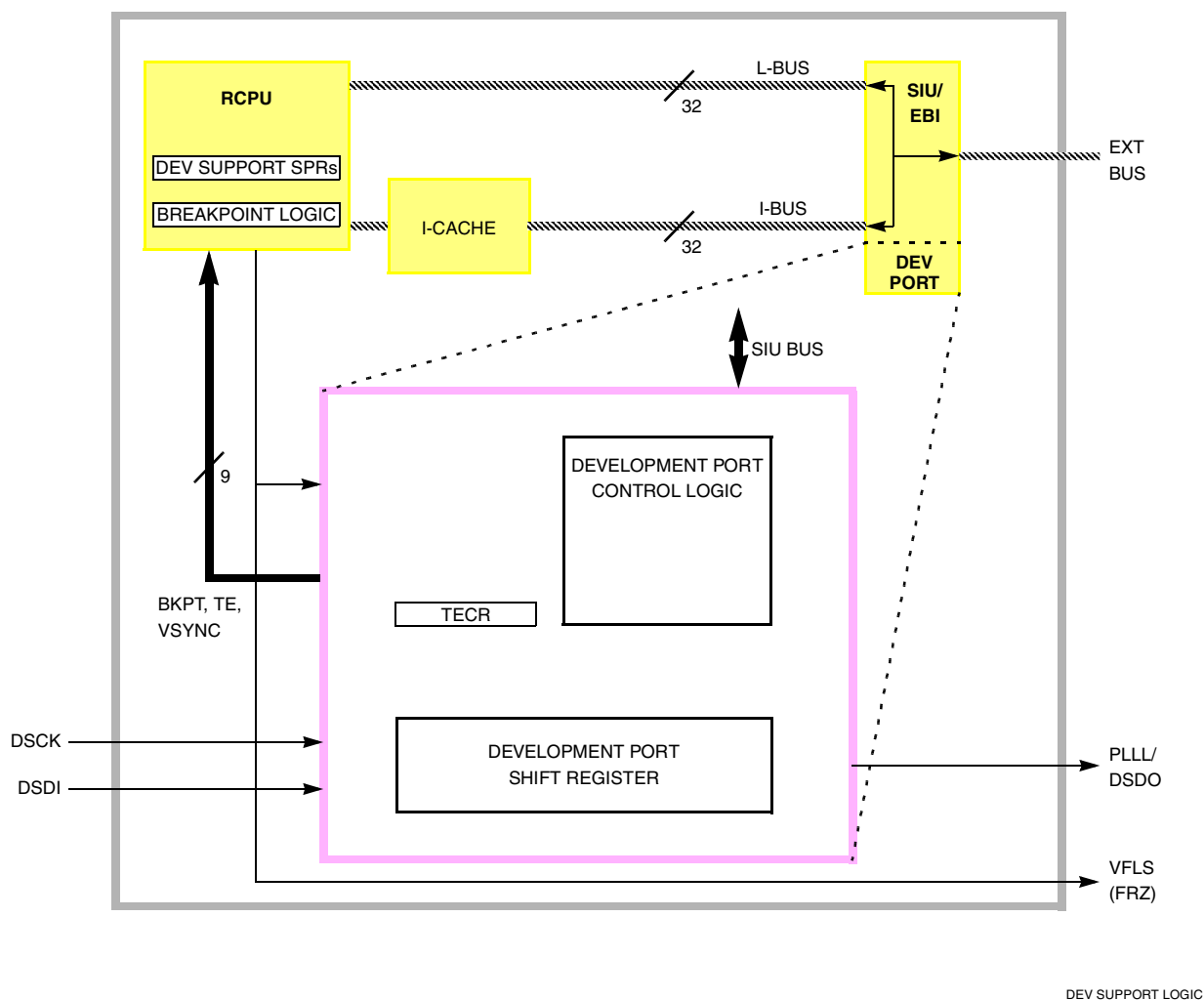
Only the development port and the internal breakpoint logic are capable of generating a non-maskable breakpoint. This allows the user to stop the processor in cases where it would otherwise not stop, but with the penalty that it may not be restartable. The value of the MSR[RI] bit as saved in the SRR1 register indicates whether the processor stopped in a recoverable state or not.

Internal breakpoints are made maskable or non-maskable by clearing or setting the BRKNOMSK bit of the LCTRL2 register. Refer to [8.8.7 L-Bus Support Control Register 2](#).

### 8.3 Development Port

The development port provides a full duplex serial interface for communications between the internal development support logic, including debug mode, and an external development tool.

The relationship of the development support logic to the rest of the MCU is shown in [Figure 8-5](#). Although the development port is implemented as part of the system interface unit (SIU), it is used in conjunction with RCPU development support features and is therefore described in this section.



**Figure 8-5 Development Port Support Logic**

### 8.3.1 Development Port Signals

The following development port signals are provided:

- Development serial clock (DSCK)
- Development serial data in (DSDI)
- Development serial data out (DSDO)

The development port signal DSDO shares a pin with the PLLL signal.

#### 8.3.1.1 Development Serial Clock

In clocked mode (see [8.3.3 Development Port Clock Mode Selection](#)), the development serial clock (DSCK) is used to shift data into and out of the development

port shift register. The DSCK and DSDI inputs are synchronized to the on-chip system clock, thereby minimizing the chance of propagating metastable states into the serial state machine. The values of the pins are sampled during the low phase of the system clock. At the rising edge of the system clock, the sampled values are latched internally. One quarter clock later, the latched values are made available to the development support logic.

In clocked mode, detection of the rising edge of the synchronized clock causes the synchronized data from the DSDI pin to be loaded into the least significant bit of the shift register. This transfer occurs one quarter clock after the next rising edge of the system clock. At the same time, the new most significant bit of the shift register is presented at the PLLL/DSDO pin. Future references to the DSCK signal imply the internal synchronized value of the clock. The DSCK input must be driven either high or low at all times and not allowed to float. A typical target environment would pull this input low with a resistor.

To allow the synchronizers to operate correctly, the development serial clock frequency must not exceed one half of the system clock frequency. The clock may be implemented as a free-running clock. The shifting of data is controlled by ready and start signals so the clock does not need to be gated with the serial transmissions. (Refer to [8.3.5 Trap-Enable Input Transmissions](#) and [8.3.6 CPU Input Transmissions](#).)

The DSCK pin is also used during reset to enable debug mode and immediately following reset to optionally cause immediate entry into debug mode following reset. This is described in section [8.4.1 Enabling Debug Mode](#) and [8.4.2 Entering Debug Mode](#).

### 8.3.1.2 Development Serial Data In

Data to be transferred into the development port shift register is presented at the development serial data in (DSDI) pin by external logic. To be sure that the correct value is used internally, transitions on the DSDI pin should occur at least a setup time ahead of the rising edge of the DSCK signal (if in clocked mode) or a setup time ahead of the rising edge of the system clock, whichever is greater. This will allow operation of the development port either asynchronously or synchronously with the system clock. The DSDI input must be driven either high or low at all times and not allowed to float. A typical target environment would pull this input low with a resistor.

When the processor is not in debug mode (freeze not indicated on VFLS[0:1] pins) the data received on the DSDI pin is transferred to the trap enable control register. When the processor is in debug mode, the data received on the DSDI pin is provided to the debug mode interface. Refer to [8.3.5 Trap-Enable Input Transmissions](#) and [8.3.6 CPU Input Transmissions](#) for additional information.

The DSDI pin is also used at reset to control overall chip reset configuration and immediately following reset to determine the development port clock mode. See [8.3.3 Development Port Clock Mode Selection](#) for more information.



When the processor is not in reset, the development port shifts data out of the development port shift register using the development serial data out (PLLL/DSDO) pin. When the processor is in reset, the PLL/DSDO pin indicates the state of lock of the system clock phase-locked loop. This can be used to determine when a reset is caused by a loss of lock on the system clock PLL.

The development port consists of two registers: the development port shift register and the trap enable control register. These registers are described in the following paragraphs. **Figure 8-6** illustrates the development port registers and data paths.



## 8.3.2.1 Development Port Shift Register

The development port shift register is a 35-bit shift register. Instructions and data are shifted into it serially from DSDI. These instructions or data are then transferred in parallel to the processor or the trap enable control register (TECR).

When the processor enters debug mode, it fetches instructions from the development port shift register. These instructions are serially loaded into the shift register from DSDI.

When the processor is in debug mode, data is transferred to the CPU by shifting it into the shift register. The processor then reads the data as the result of executing a “move from special purpose register DPDR” (development port data register) instruction.

In debug mode, data is also parallel loaded into the development port shift register from the CPU by executing a “move to special purpose register DPDR” instruction. It is then shifted out serially to PLLL/DSDO.

## 8.3.2.2 Trap Enable Control Register

The trap enable control register (TECR) is a nine-bit register that is loaded from the development port shift register. The contents of the TECR are used to drive the six trap enable signals, the two breakpoint signals, and the VSYNC signal to the processor. Trap-enable transmissions to the development port cause the appropriate bits of the development port shift register to be transferred to the control register.

## 8.3.3 Development Port Clock Mode Selection

All of the development port serial transmissions are clocked transmissions. The transmission clock can be either synchronous or asynchronous with the system clock (CLKOUT). The development port supports three methods for clocking the serial transmissions. The first method allows the transmission to occur without being externally synchronized with CLKOUT but at more restricted data rates. The two faster communication methods require the clock and data to be externally synchronized with CLKOUT.

The first clock mode is called *asynchronous clocked* since the input clock (DSCK) is asynchronous with CLKOUT. The input synchronizers on the DSCK and DSDI pins sample the inputs to ensure that the signals used internally have no metastable oscillations. To be sure that data on DSDI is sampled correctly, transitions on DSDI must occur a setup time ahead of the rising edge of DSCK. Data on DSDI must also be held for one CLKOUT cycle plus one hold time after the rising edge of DSCK. This ensures that after the signals have passed through the input synchronizers, the data will be valid at the rising edge of the serial clock even if DSCK and DSDI do not meet the setup and hold time requirements of the pins.

Asynchronous clocked mode allows communications with the port from a development tool that does not have access to the CLKOUT signal or where the CLKOUT signal has been delayed or skewed. Because of the asynchronous nature of the inputs and the setup and hold time requirements on DSDI, this clock mode must be clocked at a frequency less than or equal to one third of CLKOUT.

## Freescale Semiconductor, Inc.

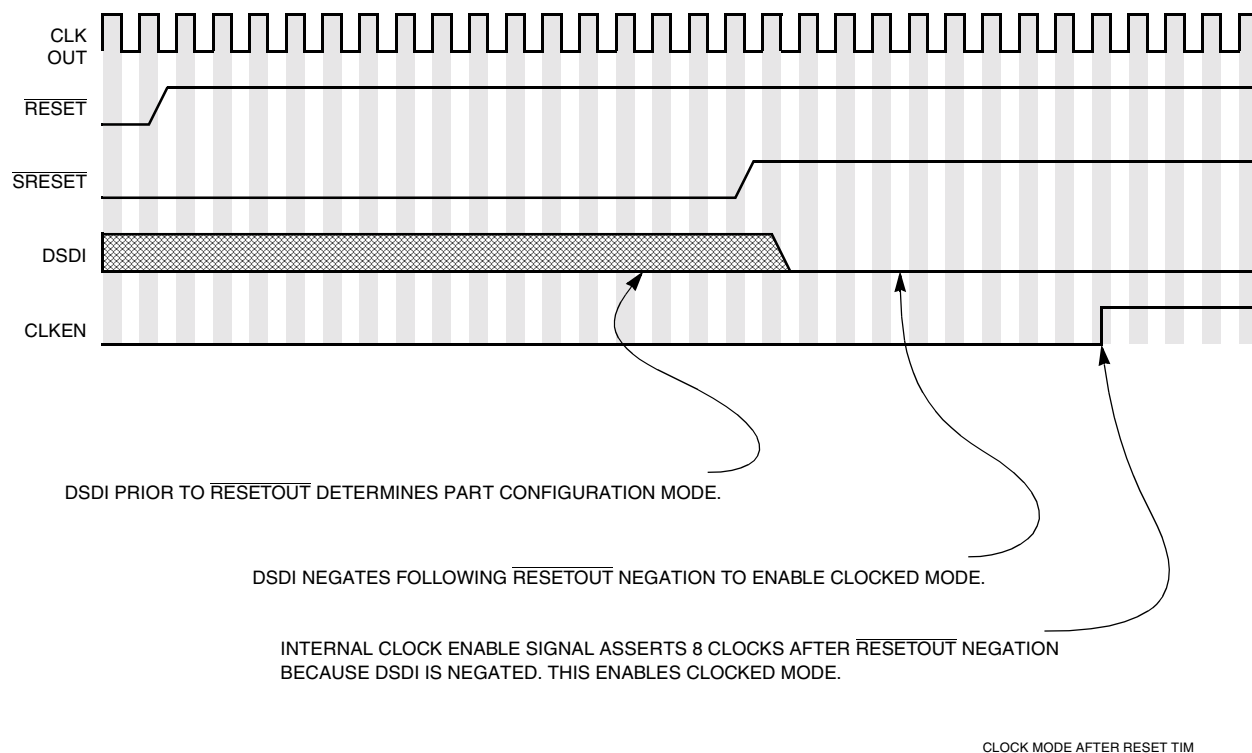
The second clock mode is called *synchronous clocked* because the input clock and input data meet all setup and hold time requirements with respect to CLKOUT. Since the input synchronizers must sample the input clock in both the high and low state, DSCK cannot be faster than one half of CLKOUT.

The third clock mode is called *synchronous self-clocked* because it does not require an input clock. Instead, the port is clocked by the system clock. The DSDI input is required to meet all setup and hold time requirements with respect to CLKOUT. The data rate for this mode is always the same as the system clock rate, which is at least twice as fast as in synchronous clocked mode. In this mode, an undelayed CLKOUT signal must be available to the development tool, and extra care must be taken to avoid noise and crosstalk on the serial lines.

The selection of clocked or self-clocked mode is made immediately following reset. The state of the DSDI input is latched eight clocks after  $\overline{\text{RESETOUT}}$  is negated. If it is latched low, external clocked mode is enabled. If it is latched high then self clocked mode is enabled. When external clocked mode is enabled, the use of asynchronous or synchronous mode is determined by the design of the external development tool.

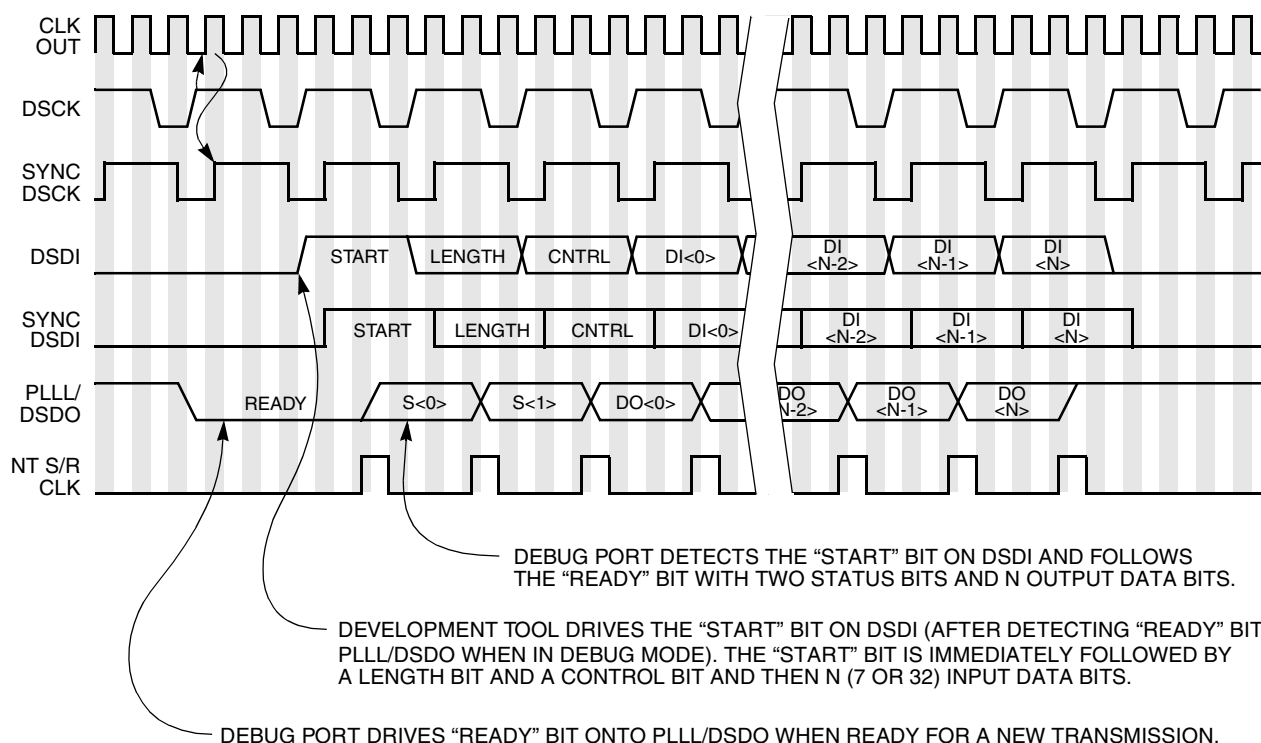
Since DSDI is used during reset to configure the MCU and to select the development port clocking scheme, it is necessary to prevent any transitions on DSDI during this time from being recognized as the start of a serial transmission. The port does not begin scanning for the start bit of a serial transmission until 16 clocks after the negation of  $\overline{\text{RESETOUT}}$ . If DSDI is asserted 16 clocks after  $\overline{\text{RESETOUT}}$  negation, the port will wait until DSDI is negated to begin scanning for the start bit.

The selection of clocked/self clocked mode is shown in [Figure 8-7](#). The timing diagrams in [Figure 8-8](#), [Figure 8-9](#), and [Figure 8-10](#) show the serial communications for both trap enable mode and debug mode for all clocking schemes.



**Figure 8-7 Enabling Clock Mode Following Reset**

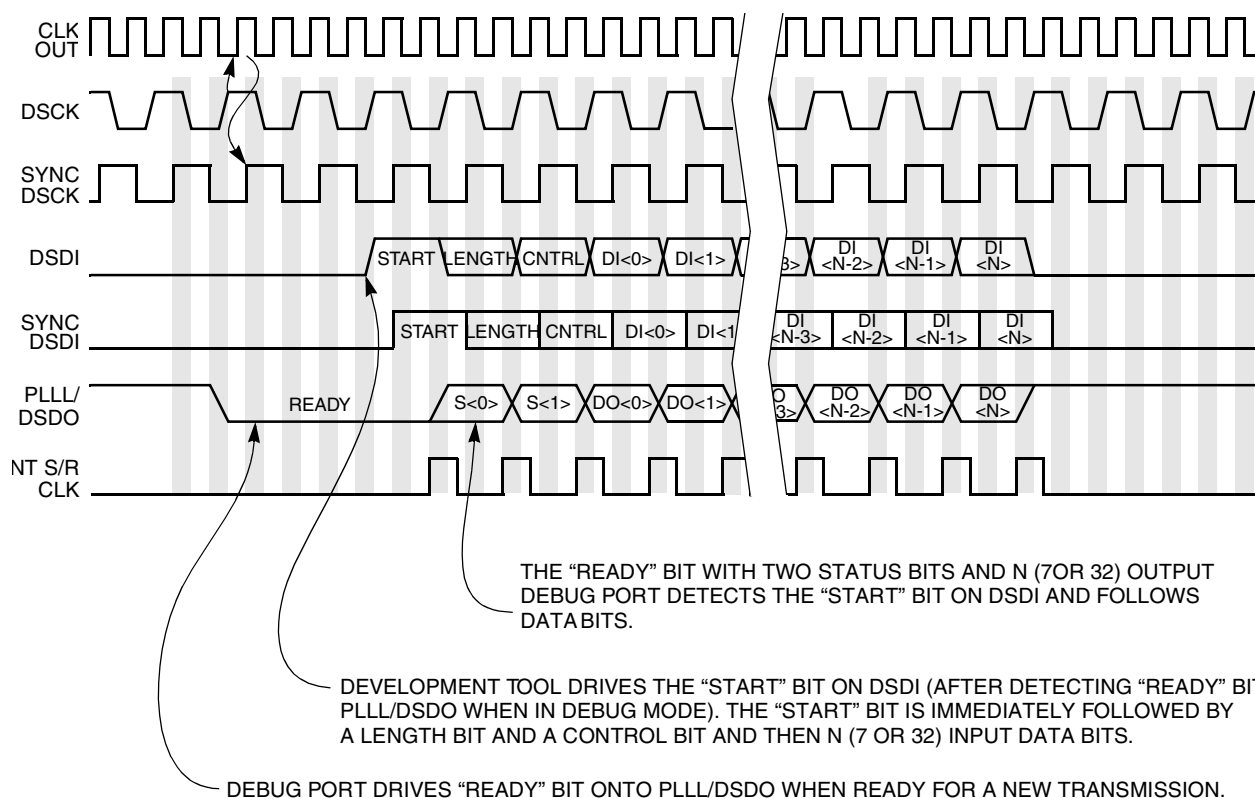
Examples of serial communications using the three clock modes are shown in [Figure 8-8](#), [Figure 8-9](#), and [Figure 8-10](#).



ASYNCR SER COM TIM

**Figure 8-8 Asynchronous Clocked Serial Communications**

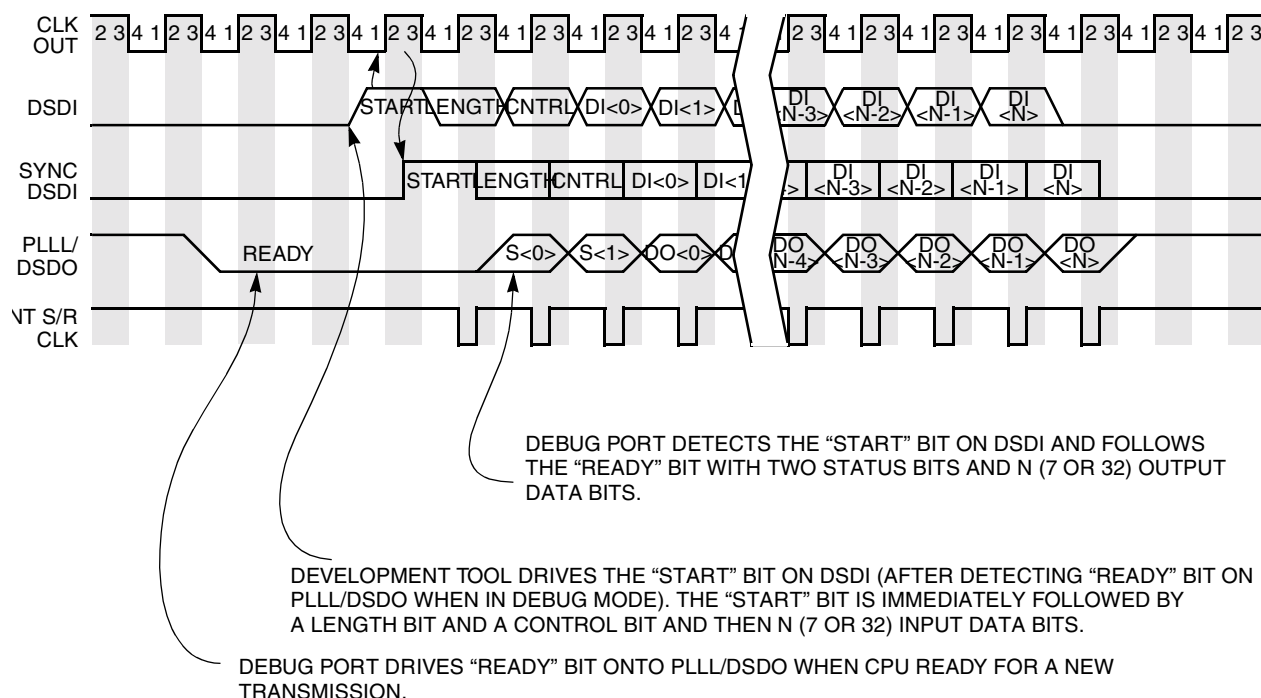
In **Figure 8-8**, the frequency on the DSCCK pin is equal to CLKOUT frequency divided by three. This is the maximum frequency allowed for the asynchronous clocked mode. DSCCK and DSDI transitions are not required to be synchronous with CLKOUT.



SYNC SER COM TIM

**Figure 8-9 Synchronous Clocked Serial Communications**

In **Figure 8-9**, the frequency on the DSK pin is equal to CLKOUT frequency divided by two. DSDI and DSK transitions must meet setup and hold timing requirements with respect to CLKOUT.



SYNC S SER COM 1

**Figure 8-10 Synchronous Self-Clocked Serial Communications**

In [Figure 8-10](#), the DSCK pin is not used, and the transmission is clocked by CLK-OUT. DSDI transitions must meet setup and hold timing requirements with respect to CLKOUT.

### 8.3.4 Development Port Transmissions

The development port starts communications by setting PLL/DSDO (the *ready* bit, or MSB of the 35 bit development port shift register) low to indicate that all activity related to the previous transmission is complete and that a new transmission may begin. The start of a serial transmission from an external development tool to the development port is signaled by a *start* bit on the DSDI pin.

The start bit also signals the development port that it can begin driving data on the DSDO pin. While data is shifting into the LSB of the shift register from the DSDI pin, it is simultaneously shifting out of the MSB of the shift register onto the DSDO pin.

A *length* bit defines the transmission as being to either the trap-enable register (length bit = 1, indicating 7 data bits) or the CPU (length bit = 0, indicating 32 data bits). Transmissions of data and instructions to the CPU are allowed only when the processor is in debug mode. The two types of transmissions are discussed in [8.3.5 Trap-Enable Input Transmissions](#) and [8.3.6 CPU Input Transmissions](#).

### 8.3.5 Trap-Enable Input Transmissions

If the length bit is set, the input transmission will only be 10 bits long. These trap-enable transmissions into the development port include a start bit, a length bit, a control bit, and seven data bits. Only the seven data bits are shifted into the 35-bit shift register. These seven bits are then latched into the TECR. The control bit determines whether the data is latched into the trap enable and VSYNC bits of the TECR or into the breakpoints bits of the TECR, as shown in [Table 8-11](#) and [Table 8-12](#).

**Table 8-11 Trap Enable Data Shifted Into Development Port Shift Register**

Start	Length	Control	1st	2nd	3rd	4th	1st	2nd	VSYNC	Usage
			I-bus				L-bus			
			Watchpoint Trap Enables							
1	1	0	0 = disabled; 1 = enabled							Input data for trap enable control register

**Table 8-12 Breakpoint Data Shifted Into Development Port Shift Register**

Start	Length	Control	Non-Maskable	Maskable	Reserved bits					Usage
			Breakpoints							
1	1	1	0 = negate; 1 = assert		1	1	1	1	1	Input data for trap enable control register

### 8.3.6 CPU Input Transmissions

If the length bit in the serial input sequence is cleared, the transmission is an input to the CPU. This transmission type is legal only when the processor is in debug mode.

For transmissions to the CPU, the 35 bits of the development port shift register are interpreted as a start bit, a length bit, a control bit, and 32 bits of instructions or data. The encoding of data shifted into the development port shift register (through the DSDI pin) is shown in [Table 8-13](#).

**Table 8-13 CPU Instructions/Data Shifted into Shift Register**

Start	Length	Control	Instruction/Data (32 Bits)	Usage
1	0	0	CPU Instruction	Input instruction for the CPU
1	0	1	CPU Data	Input data for the CPU



## Freescale Semiconductor, Inc.

The control bit differentiates between instructions and data and allows the development port to detect that an instruction was entered when the CPU was expecting data and vice versa. If this occurs, a sequence error indication is shifted out in the next serial transmission.

### 8.3.7 Serial Data Out of Development Port — Non-Debug Mode

The encoding of data shifted out of the development port shift register when the processor is not in debug mode is shown in [Table 8-14](#).

**Table 8-14 Status Shifted Out of Shift Register — Non-Debug Mode**

Ready	Status [0:1]		Data (7 or 32 Bits <sup>1</sup> )	Indication
(0)	0	1	Ones	Sequencing Error
(0)	1	1	Ones	Null

**NOTES:**

1. Depending on input mode.

When the processor is not in debug mode, the sequencing error encoding indicates that the transmission from the external development tool was a transmission to the CPU (length = 0). When a sequencing error occurs, the development port ignores the data being shifted in while the sequencing error is shifting out.

The null output encoding is used to indicate that the previous transmission did not have any associated errors.

When the processor is not in debug mode, the ready bit is asserted at the end of each transmission. If debug mode is not enabled and transmission errors can be guaranteed not to occur, the status output is not needed, and the DSDO pin can be used for untimed I/O.

### 8.3.8 Serial Data Out of Development Port — Debug Mode

The encoding of data shifted out of the development port shift register when the processor is in debug mode is shown in [Table 8-14](#).

Table 8-15 Status/Data Shifted Out of Shift Register

Ready	Status [0:1]		Data (7 or 32 Bits <sup>1</sup> )	Indication
(0)	0	0	Data	Valid Data from CPU
(0)	0	1	Ones	Sequencing Error
(0)	1	0	Ones	CPU Exception
(0)	1	1	Ones	Null

NOTES:

1. Depending on input mode.

### 8.3.8.1 Valid Data Output

The *valid data* encoding is used when data has been transferred from the CPU to the development port shift register. This is the result of executing an instruction in debug mode to move the contents of a general purpose register to the development port data register (DPDR).

The valid data encoding has the highest priority of all status outputs and is reported even if an exception occurs at the same time. Any exception that is recognized during the transmission of valid data is not related to the execution of an instruction. Therefore, a status of valid data is output and the CPU exception status is saved for the next transmission. Since it is not possible for a sequencing error to occur and for valid data to be received on the same transmission, there is no conflict between a valid data status and the sequencing error status.

### 8.3.8.2 Sequencing Error Output

The *sequencing error* encoding indicates that the inputs from the external development tool are not what the development port or the CPU was expecting. Two cases could cause this error:

- 1) the processor was trying to read instructions and data was shifted into the development port, or
- 2) the processor was trying to read data and an instruction was shifted into the development port.

When a sequencing error occurs, the port terminates the CPU read or fetch cycle with a bus error. This bus error causes the CPU to signal the development port that an exception occurred. Since a status of sequencing error has a higher priority than a status of exception, the port reports the sequencing error. The development port ignores the data being shifted in while the sequencing error is shifting out. The next transmission to the port should be a new instruction or trap enable data.

**Table 8-16** illustrates a typical sequence of events when a sequencing error occurs. This example begins with CPU data being shifted into the shift register (control bit = 1) when the processor is expecting an instruction. During the next

transmission, a sequencing error is shifted out of the development port, and the data shifted into the shift register is thrown away. During the third transmission, the “CPU exception” status is output, and again the data shifted into the shift register is thrown away. During the fourth transmission, an instruction is again shifted into the development port and fetched by the CPU for execution. Notice in this example that the development port throws away the first two input transmissions following the one causing the sequencing error.

**Table 8-16 Sequencing Error Activity**

Trans #	Input to Development Port	Output from Development Port	Port Action	CPU Action
1	CPU Data (Control bit = 1)	Depends on previous transmissions	Cause bus error, set sequence error latch	Fetch instruction, take exception because of bus error
2	X (Thrown away)	Sequencing Error	Set exception latch, clear sequencing error latch	Signal exception to port, begin new fetch from port
3	X (Thrown away)	CPU Exception	Clear exception latch	Continue to wait for instruction from port
4	CPU instruction	Null	Send instruction to CPU at end of transmission	Fetch instruction from port

### 8.3.8.3 CPU Exception Output

The *CPU exception* encoding is used to indicate that the CPU encountered an exception during the execution of the previous instruction in debug mode. Exceptions may occur as the result of instruction execution (such as unimplemented opcode or arithmetic error), because of a memory access fault, or from an external interrupt. The exception is recognized only if the associated bit in the DER is set. When an exception occurs, the development port ignores the data being shifted in while the CPU exception status is shifting out. The port terminates the current CPU access with a bus error. The next transmission to the port should be a new instruction or trap enable data.

### 8.3.8.4 Null Output

Finally, the *null* encoding is used to indicate that no data has been transferred from the CPU to the development port shift register. It also indicates that the previous transmission did not have any associated errors.

### 8.3.9 Use of the Ready Bit

To minimize the overhead required to detect and correct errors, the external development system should wait for the ready bit on DSDO before beginning each input transmission. This ensures that all CPU activity (if any) relative to the previous transmission has been completed and that any errors have been reported.

When the ready bit is used to pace the transmissions, the error status is reported during the transmission following the error. Since any transmission into the port which occurs while shifting out an error status is ignored by the port, the error handler in the external development tool does not need to undo the effects of an intervening instruction.

To improve system performance, however, an external development system may begin transmissions before the ready bit is asserted. If the next transmission does not wait until the port indicates ready, the port will not assert ready again until this next transmission completes and all activity associated with it has finished. Transmissions that begin before ready is asserted on DSDI are subject to the following limitations and problems.

First, if the previous transmission results in a sequence error, or the CPU reports an exception, that status may not be reported until two transmissions after the transmission that caused the error. (When the ready bit is used, the status is reported in the following transmission.) This is because an error condition which occurs after the start of a transmission cannot be reported until the next transmission.

Second, if a transmitted instruction causes the CPU to write to the DPDR and the transmission that follows does not wait for the assertion of ready, the CPU data may not be latched into the development port shift register, and the valid data status is not output. Despite this, no error is indicated in the status outputs. To ensure that the CPU has had enough time to write to the DPDR, there must be at least four CLKOUT cycles between when the last bit of the instruction (move to SPR) is clocked into the port and the time the start bit for the next transmission is clocked into the port.

## **8.4 Debug Mode Functions**

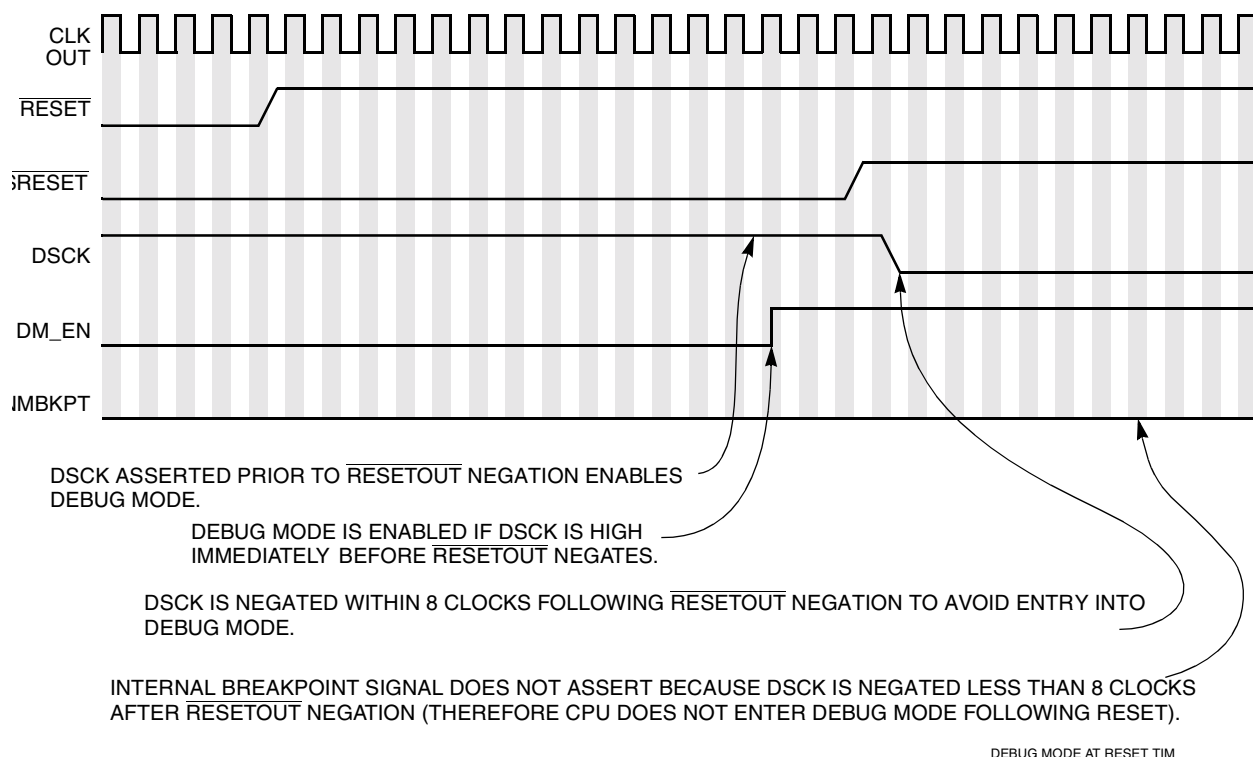
In debug mode, the CPU fetches all instructions from the development port. In addition, data can be read from and written to the development port. This allows memory and registers to be read and modified by an external development tool (emulator) connected to the development port.

### **8.4.1 Enabling Debug Mode**

Debug mode is enabled by asserting the DSCK pin during reset. The state of this pin is sampled immediately before the negation of RESETOUT. If the DSCK pin is sampled low, debug mode is disabled until a subsequent reset when the DSCK pin is sampled high. When debug mode is disabled, the internal watchpoint/breakpoint hardware is still operational and can be used by a software monitor program for debugging purposes.

The DSCK pin is sampled again eight clock cycles following the negation of RESETOUT. If DSCK is negated following reset, the processor jumps to the reset vector and begins normal execution. If DSCK is asserted following reset and debug mode is enabled, the processor enters debug mode before executing any instructions.

A timing diagram for enabling debug mode is shown in [Figure 8-11](#).



**Figure 8-11 Enabling Debug Mode at Reset**

#### 8.4.2 Entering Debug Mode

Debug mode is entered whenever debug mode is enabled, an exception occurs, and the corresponding bit is set in the debug enable register (DER). The processor performs normal exception processing, i.e., saving the next instruction address and the current state of MSR in SRR0 and SRR1 and modifying the contents of the MSR. The processor then enters debug mode and fetches the next instruction from the development port instead of from the vector address. The exception cause register (ECR) shows which event caused entry into debug mode. The freeze indication is encoded on the VFLS pins to show that the CPU is in debug mode.

Debug mode may also be entered immediately following reset. If the DSCK pin continues to be asserted following reset (after debug mode is enabled), the processor takes a breakpoint exception and enters debug mode directly after fetching (but not executing) the reset vector. To avoid entering debug mode following reset, the DSCK pin must be negated no later than seven clock cycles after  $\overline{\text{RESETOUT}}$  is negated.

A timing diagram for entering debug mode following reset is shown in [Figure 8-12](#).

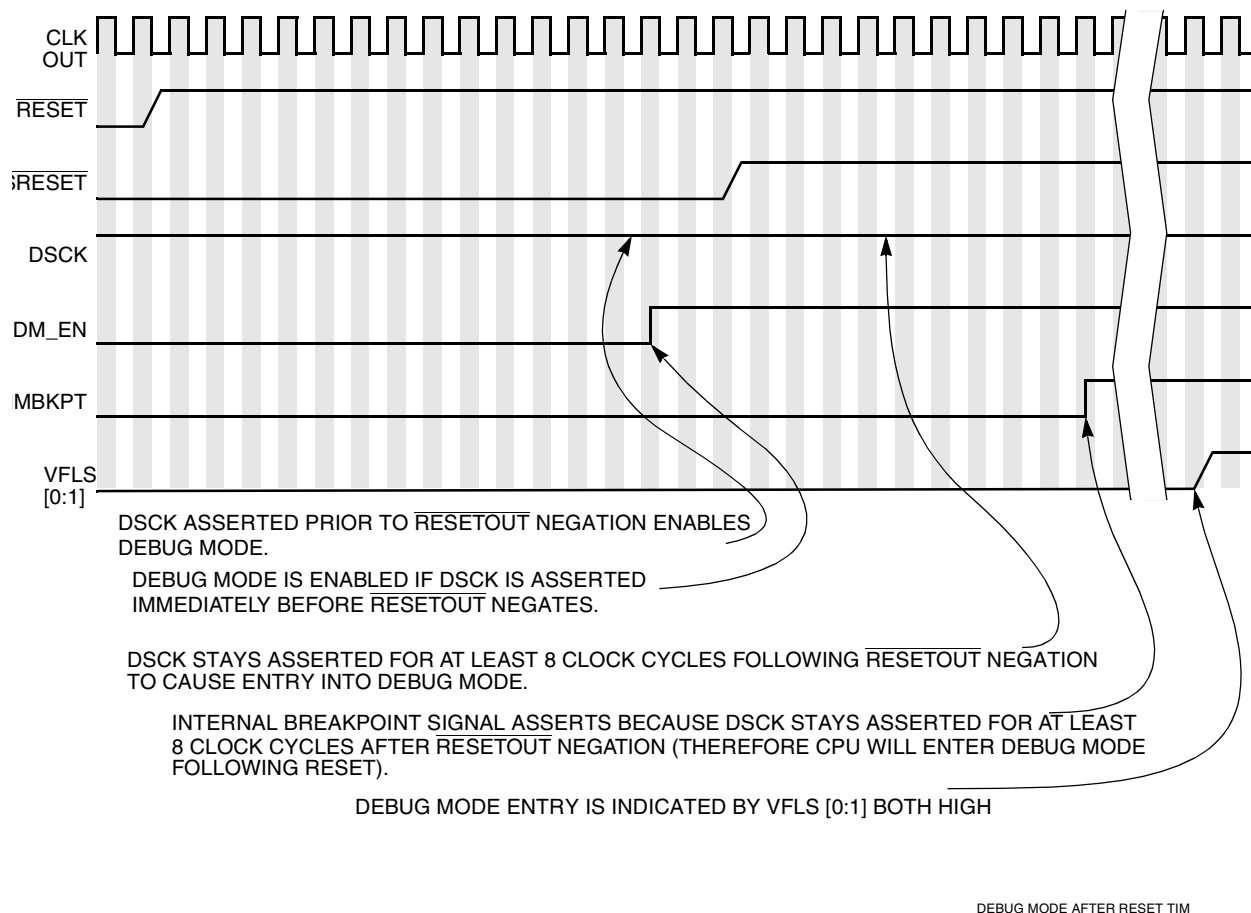


Figure 8-12 Entering Debug Mode Following Reset

### 8.4.3 Debug Mode Operation

In debug mode, the CPU fetches instructions from the development port. It can also read and write data at the development port. In debug mode the prefetch mechanism in the CPU is disabled. This forces all data accesses to the development port to occur immediately following the fetch of the associated instruction.

In debug mode, if an exception occurs during the execution of an instruction, normal exception processing does not result. (That is, the processor does not save the MSR and instruction address and does not branch to the exception handler.) Instead, a flag is set that results in a CPU exception status indication in the data shifted out of the development port shift register. The same thing happens if the processor detects an external interrupt. (This can occur only when the associated DER bit is clear and MSR[EE] is set.) When the data in the development port shift register is shifted out, the exception status is detected by the external development tool. The cause of the exception can be determined by reading the ECR.

#### 8.4.4 Freeze Function

While the processor is in debug mode, the freeze indication is broadcast throughout the MCU. This signal is generated by the CPU when debug mode is entered, or when a software debug monitor program is entered as the result of an exception and the associated bit in the DER is set. The software monitor can only assert freeze when debug mode is not enabled. Refer to [8.7 Software Monitor Support](#) for more information.

Freeze is indicated by the value 11 on the VFLS[0:1] pins. This encoding is not used for pipeline tracking and is left on the VFLS[0:1] pins when the processor is in debug mode. [Figure 8-14](#) shows how the internal freeze signal is generated.

#### 8.4.5 Exiting Debug Mode

Executing the **rfi** instruction in debug mode causes the processor to leave debug mode and return to normal execution. The freeze indication on the VFLS pins is negated to indicate that the CPU has exited debug mode.

Software must read the ECR (to clear it) before executing the **rfi** instruction. Otherwise, if a bit in the ECR is asserted and its corresponding enable bit in the DER is also asserted, the processor re-enters debug mode and re-asserts the freeze signal immediately after executing the **rfi** instruction.

#### 8.4.6 Checkstop State and Debug Mode

When debug mode is disabled, the processor enters the checkstop state if, when a machine check exception is detected, the machine check exception is disabled (MSR[ME] = 0). However, when debug mode is enabled, if a machine check exception is detected when MSR[ME] = 0 and the checkstop enable bit in the DER is set, the processor enters debug mode rather than the checkstop state. This allows the user to determine why the checkstop state was entered. [Table 8-17](#) shows what happens when a machine check exception occurs under various conditions.

**Table 8-17 Checkstop State and Debug Mode**

MSR[ME]	Debug Mode Enable	CHSTPE <sup>1</sup>	MCIE <sup>2</sup>	Action Performed when CPU Detects a Machine Check Interrupt	ECR Value
0	0	X	X	Enter the checkstop state	0x2000 0000
0	1	0	X	Enter the checkstop state	0x2000 0000
0	1	1	X	Enter debug mode	0x2000 0000
1	0	X	X	Take machine check exception	0x1000 0000
1	1	X	0	Take machine check exception	0x1000 0000
1	1	X	1	Enter debug mode	0x1000 0000

NOTES:

1. Checkstop enable bit in the DER
2. Machine check interrupt enable bit in the DER

## 8.5 Development Port Transmission Sequence

The following sections describe the sequence of events for communication with the development port in both debug and normal mode and provide specific sequences for prologues, epilogues, and poking and peeking operations.

### 8.5.1 Port Usage in Debug Mode

The sequence of events for communication with the development port in debug mode (freeze is indicated on the VFLS pins) is shown in [Table 8-18](#). The sequence starts with the processor trying to read an instruction in step one. The sequence ends when the processor is ready to read the next instruction. Reading an instruction is the first action the processor takes after entering debug mode. The processor and development port activity is determined by the instruction or data shifted into the shift register. The instruction or data shifted into the shift register also determines the status shifted out during the next transmission. The next step column indicates which step has the appropriate status response.



## Table 8-18 Debug Mode Development Port Usage

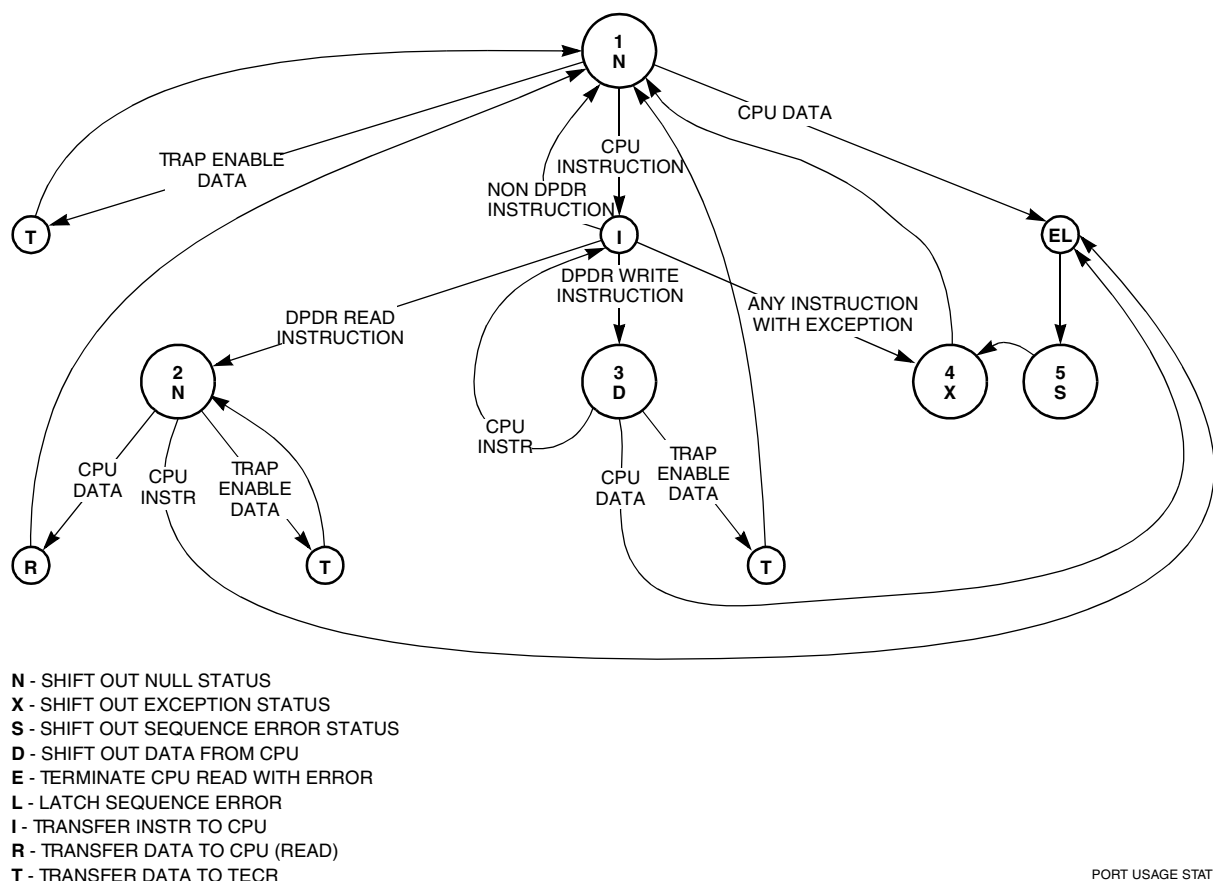
This Step	Serial Data Shifted In (DSDO indicates "READY")	Shifted Out This Transmission	Development Port Activity; Processor Activity	Next Step
1	CPU instruction (non-DPDR)	Null	Port transfers instruction to CPU; CPU executes instruction, fetches next instruction	1
	CPU instruction (DPDR read)		Port transfers instruction to CPU; CPU executes instruction, reads DPDR	2
	CPU instruction (DPDR write)		Port transfers instruction to CPU; CPU writes DPDR, fetches next instruction	3
	CPU instruction (instruction execution causes exception)		Port transfers instruction to CPU; CPU signals exception to port, fetches next instruction	4
	Data for CPU		Port ignores data, terminates fetch with error, latches sequence error; CPU signals exception to port, fetches next instruction	5
	Data for Trap Enable Control Register		Port updates Trap Enable Control Register; CPU waits (continues fetch)	1
2	Any CPU instruction	Null	Port ignores data, terminates DPDR read with error; latches sequence error; CPU signals exception to port, fetches next instruction	5
	Data for CPU		Port transfers data to CPU; CPU reads data from DPDR, fetches next instruction	1
	Data for trap enable control register		Port updates TECR; CPU waits (continue data read)	2

**Table 8-18 Debug Mode Development Port Usage (Continued)**

This Step	Serial Data Shifted In (DSDO indicates "READY")	Shifted Out This Transmission	Development Port Activity; Processor Activity	Next Step
3	CPU instruction (non-DPDR)	CPU data	Port transfers instruction to CPU; CPU executes instruction, fetches next instruction	1
	CPU instruction (DPDR read)		Port transfers instruction to CPU; CPU executes instruction, reads DPDR	2
	CPU instruction (DPDR write)		Port transfers instruction to CPU; CPU writes DPDR, fetches next instruction	3
	CPU instruction (with exception)		Port transfers instruction to CPU; CPU signals exception to port, fetches next instruction	4
	Data for CPU		Port ignores data, terminates fetch with error, latches sequence error; CPU signals exception to port, fetches next instruction	5
	Data for trap enable control register	7 MSB of CPU data	Port updates TECR; CPU waits (continues fetch)	1
4	Any (ignored by port)	Exception	Port ignores data; CPU waits (continues fetch)	1
5	Any (ignored by port)	Sequence Error	Port ignores data; CPU waits (continues fetch)	4

### 8.5.2 Debug Mode Sequence Diagram

The sequence of activity shown in [Table 8-18](#) is summarized below in [Figure 8-13](#). The numbers in the large circles correspond to the steps in [Table 8-18](#). The letters in the large circles indicate the status that will be shifted out during the transmission. The letters in the small circles show the activity of the development port and the CPU as a result of the transmission.



**Figure 8-13 General Port Usage Sequence Diagram**

### 8.5.3 Port Usage in Normal (Non-Debug) Mode

The sequence of events for communication with the development port when the CPU is not in debug mode (freeze is not indicated on the VFLS pins) is shown below in [Table 8-18](#). Note that any instructions or data for the CPU result in a sequence error status response when the processor is not in debug mode. Only data for the trap enable control register is allowed.

**Table 8-19 Non-Debug Mode Development Port Usage**

This Step	Serial Data Shifted Into DPDI (not in Debug Mode)	Shifted Out Of DPDO This Transmission	Development Port Activity	Next Step
6	Any CPU instruction or data	Null	Port ignores data and latches sequence error	7
	Data for trap enable control register		Port updates trap enable control register	6
7	Any (ignored by port)	Sequence Error	Port ignores data	6

## 8.6 Examples of Debug Mode Sequences

The tables that follow show typical sequences of instructions that are used in a development activity. They assume that no bus errors or sequence errors occur and that no writes occur to the trap enable control register.

### 8.6.1 Prologue Instruction Sequence

The prologue sequence of instructions is used to unload the machine context when entering debug mode. The sequence starts by unloading two general-purpose registers (R0 and R1) to be used as a data transfer register and an address pointer. Since SRR0 and SRR1 are not changed while in debug mode except by explicitly writing to them, there is no need to save and restore these registers. Finally, the ECR is unloaded to determine the cause of entry into debug mode. Any registers that will be used while in debug mode in addition to R0 and R1 will also need to be saved.

**Table 8-20 Prologue Events**

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction	<b>mtspr</b> DPDR, R0	Transfer R0 to DPDR	Save R0 so the register can be used
Shift out R0 data, shift in instruction	<b>mfscr</b> R0, ECR	Transfer ECR to R0	Read the debug mode cause register
Shift in instruction	<b>mtspr</b> DPDR, R0	Transfer from R0 to DPDR	Output reason for debug mode entry
Shift out stop cause data, shift in instruction	<b>mtspr</b> DPDR, R1	Transfer R1 to DPDR	Save R1 so the register can be used
Shift out R1 data, shift in instruction	First instruction of next sequence	Execute next instruction	Continue instruction processing

### 8.6.2 Epilogue Instruction Sequence

The epilogue sequence of instructions is used to restore the machine context when

leaving debug mode. It restores the two general-purpose registers and then issues the **rfi** instruction. If additional registers were used while in debug mode, they also need to be restored before the **rfi** instruction is executed.

**Table 8-21 Epilogue Events**

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction, shift in saved R0	<b>mfscr</b> R0, DPDR	Transfer from DPDR to R0	Restores value of R0 when stopped
Shift in instruction, shift in saved R1	<b>mfscr</b> R1, DPDR	Transfer from DPDR to R1	Restores value of R1 when stopped
Shift in instruction	<b>rfi</b>	Return from exception	Restart execution

## 8.6.3 Peek Instruction Sequence

The peek sequence of instructions is used to read a memory location and transfer the data to the development port. It starts by moving the memory address into R1 from the development port. Next the location is read and the data loaded into R0. Finally, R0 is transferred to the development port.

**Table 8-22 Peek Instruction Sequence**

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction	<b>mfscr</b> R1,DPDR	Transfer address from DPDR to R1	Point to memory address
Shift in instruction	<b>lwz</b> R0,D(R1)	Load data from memory address (R1) into R0	Read data from memory
Shift in instruction	<b>mtscr</b> DPDR,R0	Transfer data from R0 to DPDR	Write memory data to the port
Shift in instruction, shift out memory data	First instruction of next sequence	Execute next instruction	Output memory data

## 8.6.4 Poke Instruction Sequence

The poke sequence of instructions is used to write data entered at the development serial port to a memory location. It starts by moving the memory address into R1 from the development port. Next the data is moved into R0 from the development port. Finally, R0 is written to the address in R1.

Table 8-23 Poke Instruction Sequence

Development Port Activity	Instruction	Processor Activity	Purpose
Shift in instruction	<b>mfspir</b> R1,DPDR	Transfer address from DPDR to R1	Point to memory address
Shift in instruction, shift in memory data	<b>mfspir</b> R0, DPDR	Transfer data from DPDR to R0	Read memory data from the port
Shift in instruction	<b>stwu</b> R0,D(R1)	Store data from R0 to memory address (R1)	Write data to memory

## 8.7 Software Monitor Support

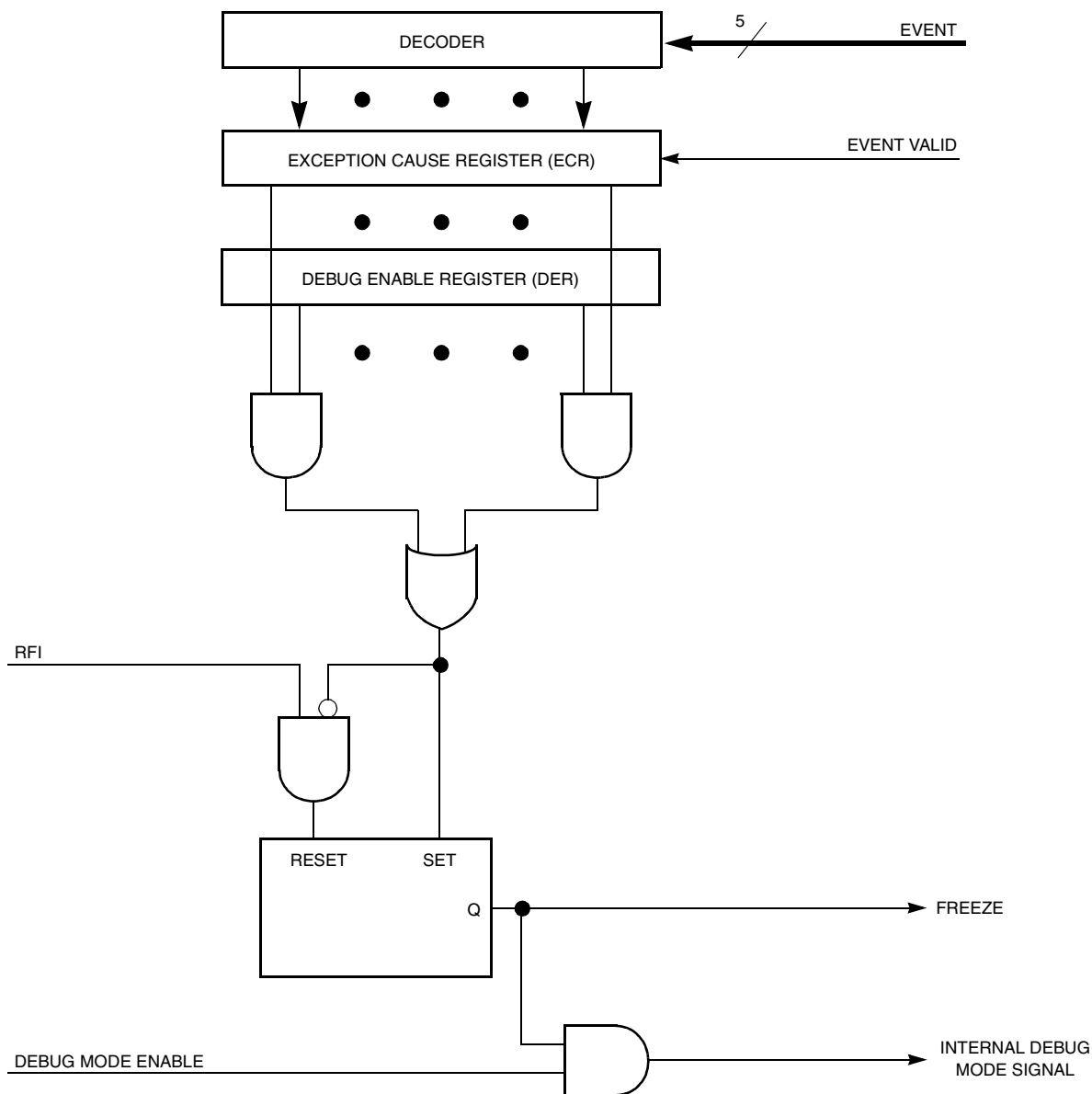
When debug mode is disabled, a software monitor debugger can make use of all of the processor's development support features. With debug mode disabled, all events result in regular exception handling, (i.e., the processor resumes execution in the appropriate exception handler). The ECR and the DER only influence the assertion and negation of the freeze indication.

The internal freeze signal is connected to all relevant internal modules. These modules can be programmed to stop all operations in response to the assertion of the freeze signal. In order to enable a software monitor debugger to broadcast the fact that the debug software is now executing, it is possible to assert and negate the internal freeze signal when debug mode is disabled. (The freeze signal can be asserted externally only when the processor enters debug mode.)

The internal freeze signal is asserted whenever an enabled event occurs, regardless of whether debug mode is enabled or disabled. To enable an event to cause freeze assertion, software needs to set the relevant bit in the DER. To clear the freeze signal, software needs to read the ECR to clear the register and then perform an **rfi** instruction.

If the ECR is not cleared before the **rfi** instruction is executed, freeze is not negated. It is therefore possible to nest inside a software monitor debugger without affecting the value of the freeze signal, even though **rfi** is performed. Only before the last **rfi** does the software need to clear the ECR.

**Figure 8-14** shows how the ECR and DER control the assertion and negation of the freeze signal and the internal debug mode signal.



RMCU DEBUG LOGIC

**Figure 8-14 Debug Mode Logic**

## 8.8 Development Support Registers

**Table 8-24** lists the registers used for development support. The registers are accessed with the **mtspr** and **mfspir** instructions.

**Table 8-24 Development Support Programming Model**

SPR Number (Decimal)	Mnemonic	Name
144	CMPA	Comparator A Value Register
145	CMPB	Comparator B Value Register
146	CMPC	Comparator C Value Register
147	CMPD	Comparator D Value Register
148	ECR	Exception Cause Register
149	DER	Debug Enable Register
150	COUNTA	Breakpoint Counter A Value and Control Register
151	COUNTB	Breakpoint Counter B Value and Control Register
152	CMPE	Comparator E Value Register
153	CMPF	Comparator F Value Register
154	CMPG	Comparator G Value Register
155	CMPH	Comparator H Value Register
156	LCTRL1	L-Bus Support Control Register 1
157	LCTRL2	L-Bus Support Control Register 2
158	ICTRL	I-Bus Support Control Register
159	BAR	Breakpoint Address Register
630	DPDR	Development Port Data Register

### 8.8.1 Register Protection

**Table 8-25** and **Table 8-26** summarize protection features of development support registers during read and write accesses, respectively.



## Table 8-25 Development Support Registers Read Access Protection

MSR[PR]	Debug Mode Enable	In Debug Mode	Result
0	0	X	Read is performed. ECR is cleared when read. Reading DPDR yields indeterminate data.
0	1	0	Read is performed. ECR is <i>not</i> cleared when read. Reading DPDR yields indeterminate data.
0	1	1	Read is performed. ECR is cleared when read.
1	X	X	Program exception is generated. Read is not performed. ECR is <i>not</i> cleared when read.

## Table 8-26 Development Support Registers Write Access Protection

MSR[PR]	Debug Mode Enable	In Debug Mode	Result
0	0	X	Write is performed. Write to ECR is ignored. Writing to DPDR is ignored.
0	1	0	Write is <i>not</i> performed. Writing to DPDR is ignored.
0	1	1	Write is performed. Write to ECR is ignored.
1	X	X	Write is <i>not</i> performed. Program exception is generated.

# Freescale Semiconductor, Inc.

## 8.8.2 Comparator A–D Value Registers (CMPA–CMPD)

### CMPA–CMPD — Comparator A–D Value Register

SPR 144 – SPR 147

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

RESET: UNDEFINED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CMPAD														RESERVED	

RESET: UNDEFINED

**Table 8-27 CMPA-CMPD Bit Settings**

Bits	Mnemonic	Description
0:29	CMPAD	Address bits to be compared
30:31	—	Reserved

The reset state of these registers is undefined.

## 8.8.3 Comparator E–F Value Registers

### CMPE–CMPF — Comparator E–F Value Registers

SPR 152, 153

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CMPEF																															

RESET: UNDEFINED

**Table 8-28 CMPE-CMPF Bit Settings**

Bits	Mnemonic	Description
[0:31]	CMPV	Address bits to be compared

The reset state of these registers is undefined.

## 8.8.4 Comparator G–H Value Registers (CMPG–CMPH)

## CMPG–CMPH — Comparator G–H Value Registers

SPR 154, 155

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CMPGH																															
RESET: UNDEFINED																															

Table 8-29 CMPG-CMPH Bit Settings

Bits	Mnemonic	Description
[0:31]	CMPGH	Data bits to be compared

The reset state of these registers is undefined.

## 8.8.5 I-Bus Support Control Register

## ICTRL — I-Bus Support Control Register

SPR 158

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CTA			CTB			CTC			CTD			IW0		IW1	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
IW2		IW3		SIW0 EN	SIW1 EN	SIW2 EN	SIW3 EN	DIW0 EN	DIW1 EN	DIW2 EN	DIW3 EN	IIFM	SER	ISCTL	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

**Table 8-30 ICTRL Bit Settings**

Bits	Mnemonic	Description	Function
[0:2]	CTA	Compare type of comparator A	0xx = not active (reset value) 100 = equal 101 = less than 110 = greater than 111 = not equal
[3:5]	CTB	Compare type of comparator B	
[6:8]	CTC	Compare type of comparator C	
[9:11]	CTD	Compare type of comparator D	
[12:13]	IW0	I-bus 1st watchpoint programming	0x = not active (reset value) 10 = match from comparator A 11 = match from comparators (A&B)
[14:15]	W1	I-bus 2nd watchpoint programming	0x = not active (reset value) 10 = match from comparator B 11 = match from comparators (A   B)
[16:17]	IW2	I-bus 3rd watchpoint programming	0x = not active (reset value) 10 = match from comparator C 11 = match from comparators (C&D)
[18:19]	IW3	I-bus 4th watchpoint programming	0x = not active (reset value) 10 = match from comparator D 11 = match from comparators (C   D)
20	SIW0EN	Software trap enable selection of the 1st I-bus watchpoint	0 = trap disabled (reset value) 1 = trap enabled
21	SIW1EN	Software trap enable selection of the 2nd I-bus watchpoint	
22	SIW2EN	Software trap enable selection of the 3rd I-bus watchpoint	
23	SIW3EN	Software trap enable selection of the 4th I-bus watchpoint	
24	DIW0EN	Development port trap enable selection of the 1st I-bus watchpoint (read only bit)	0 = trap disabled (reset value) 1 = trap enabled
25	DIW1EN	Development port trap enable selection of the 2nd I-bus watchpoint (read only bit)	
26	DIW2EN	Development port trap enable selection of the 3rd I-bus watchpoint (read only bit)	
27	DIW3EN	Development port trap enable selection of the 4th I-bus watchpoint (read only bit)	
28	IIFM	Ignore first match, only for I-bus breakpoints	0 = Do not ignore first match, used for “go to x” (reset value) 1 = Ignore first match (used for “continue”)
29	SER	Serialize	0 = Fetch serialize the machine 1 = Normal operation

Table 8-30 ICTRL Bit Settings (Continued)

Bits	Mnemonic	Description	Function
[30:31]	ISCTL	Instruction fetch show cycle control	00 = Show cycle will be performed for all fetched instructions (reset value). When in this mode, the machine is fetch serialized. 01 = Show cycle will be performed for all changes in the program flow. 10 = Show cycle will be performed for all indirect changes in the program flow. 11 = No show cycles will be performed for fetched instructions When the value of this field is changed (with the <b>mtspr</b> instruction), the new value does not take effect until two instructions after the <b>mtspr</b> instruction. The instruction immediately following <b>mtspr</b> is under control of the old ISCTL value.

The ICTRL is cleared following reset. Note that the machine is fetch serialized whenever SER = 0b0 or ISCTL = 0b00.

### 8.8.6 L-Bus Support Control Register 1

#### LCTRL1 — L-Bus Support Control Register 1

**SPR 156**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CTE			CTF			CTG			CTH			CRWE		CRWF	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
CSG		CSH		SUSG	SUSH	CGBMSK				CHBMSK				UNUSED	
RESET:															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

**Table 8-31 LCTRL1 Bit Settings**

Bits	Mnemonic	Description	Function
[0:2]	CTE	Compare type, comparator E	0xx = not active (reset value) 100 = equal 101 = less than 110 = greater than 111 = not equal
[3:5]	CTF	Compare type, comparator F	
[6:8]	CTG	Compare type, comparator G	
[9:11]	CTH	Compare type, comparator H	
[12:13]	CRWE	Select match on read/write of comparator E	0X = don't care (reset value) 10 = match on read 11 = match on write
[14:15]	CRWF	Select match on read/write of comparator F	
[16:17]	CSG	Compare size, comparator G	00 = reserved 01 = word 10 = half word 11 = byte (Must be programmed to word for floating point compares)
[18:19]	CSH	Compare size, comparator H	
20	SUSG	Signed/unsigned operating mode for comparator G	0 = unsigned 1 = signed (Must be programmed to signed for floating point compares)
21	SUSH	Signed/unsigned operating mode for comparator H	
[22:25]	CGBMSK	Byte mask for 1st L-data comparator	0000 = all bytes are <b>not</b> masked 0001 = the last byte of the word is masked . . . 1111 = all bytes are masked
[26:29]	CHBMSK	Byte mask for 2nd L-data comparator	
[30:31]	—	Reserved	—

LCTRL1 is cleared following reset.

## 8.8.7 L-Bus Support Control Register 2

### LCTRL2 — L-Bus Support Control Register 2

**SPR 157**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LW0EN	LW0IA	LW0IADC	LW0LA	LW0LADC	LW0LDDC	LW0LDDC	LW1EN	LW1IA	LW1IADC	LW1LA					

RESET:

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

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
LW1LADC	LW1LD	LW1LDDC	BRK NOM- SK	RESERVED								SLW0 EN	SLW1 EN	DLW0 EN	DLW1 EN

RESET:

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

**Table 8-32 LCTRL2 Bit Settings**

Bits	Mnemonic	Description	Function
0	LW0EN	1st L-bus watchpoint enable bit	0 = watchpoint not enabled (reset value) 1 = watchpoint enabled
[1:2]	LW0IA	1st L-bus watchpoint I-addr watchpoint selection	00 = first I-bus watchpoint 01 = second I-bus watchpoint 10 = third I-bus watchpoint 11 = fourth I-bus watchpoint
3	LW0IADC	1st L-bus watchpoint care/don't care I-addr events	0 = Don't care 1 = Care
[4:5]	LW0LA	1st L-bus watchpoint L-addr events selection	00 = match from comparator E 01 = match from comparator F 10 = match from comparators (E&F) 11 = match from comparators (E   F)
6	LW0LADC	1st L-bus watchpoint care/don't care L-addr events	0 = Don't care 1 = Care
[7:8]	LW0LDDC	1st L-bus watchpoint L-data events selection	00 = match from comparator G 01 = match from comparator H 10 = match from comparators (G&H) 11 = match from comparators (G   H)
9	LW0LDDC	1st L-bus watchpoint care/don't care L-data events	0 = Don't care 1 = Care
10	LW1EN	2nd L-bus watchpoint enable bit	0 = watchpoint not enabled (reset value) 1 = watchpoint enabled

**Table 8-32 LCTRL2 Bit Settings (Continued)**

Bits	Mnemonic	Description	Function
[11:12]	LW1IA	2nd L-bus watchpoint I-addr watchpoint selection	00 = first I-bus watchpoint 01 = second I-bus watchpoint 10 = third I-bus watchpoint 11 = fourth I-bus watchpoint
13	LW1IADC	2nd L-bus watchpoint care/don't care I-addr events	0 = Don't care 1 = Care
[14:15]	LW1LA	2nd L-bus watchpoint L-addr events selection	00 = match from comparator E 01 = match from comparator F 10 = match from comparators (E&F) 11 = match from comparators (E   F)
16	LW1LADC	2nd L-bus watchpoint care/don't care L-addr events	0 = Don't care 1 = Care
[17:18]	LW1LD	2nd L-bus watchpoint L-data events selection	00 = match from comparator G 01 = match from comparator H 10 = match from comparators (G&H) 11 = match from comparator (G   H)
19	LW1LDDC	2nd L-bus watchpoint care/don't care L-data events	0 = Don't care 1 = Care
20	BRKNOMSK	Internal breakpoints non-mask bit	0 = masked mode; breakpoints are recognized only when MSR[RI]=1 (reset value) 1 = non-masked mode; breakpoints are always recognized
[21:27]	—	Reserved	—
28	SLW0EN	Software trap enable selection of the 1st L-bus watchpoint	0 = trap disabled (reset value) 1 = trap enabled
29	SLW1EN	Software trap enable selection of the 2nd L-bus watchpoint	
30	DLW0EN	Development port trap enable selection of the 1st L-bus watchpoint (read only bit)	
31	DLW1EN	Development port trap enable selection of the 2nd L-bus watchpoint (read only bit)	

LCTRL2 is cleared following reset.

For each watchpoint, three control register fields (LWxIA, LWxLA, LWxLD) must be programmed. For a watchpoint to be asserted, all three conditions must be detected.



# Freescale Semiconductor, Inc.

## 8.8.8 Breakpoint Counter A Value and Control Register

### COUNTA — Breakpoint Counter A Value and Control Register

SPR 150

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

RESET: UNDEFINED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED														CNTC	

RESET:

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

**Table 8-33 Breakpoint Counter A Value and Control Register (COUNTA)**

Bit(s)	Name	Description
[0:15]	CNTV	Counter preset value
[16:29]	—	Reserved
[30:31]	CNTC	Counter source select 00 = not active (reset value) 01 = I-bus first watchpoint 10 = L-bus first watchpoint 11 = Reserved

COUNTA[16:31] are cleared following reset; COUNTA[0:15] are undefined.

## 8.8.9 Breakpoint Counter B Value and Control Register

**COUNTB** — Breakpoint Counter B Value and Control Register**SPR 151**

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

RESET: UNDEFINED

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESERVED														CNTC	

RESET:

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

**Table 8-34 Breakpoint Counter B Value and Control Register (COUNTB)**

Bit(s)	Name	Description
[0:15]	CNTV	Counter preset value
[16:29]	—	Reserved
[30:31]	CNTC	Counter source select 00 = not active (reset value) 01 = I-bus second watchpoint 10 = L-bus second watchpoint 11 = Reserved

COUNTB[16:31] are cleared following reset; COUNTB[0:15] are undefined.

## 8.8.10 Exception Cause Register (ECR)

The ECR indicates the cause of entry into debug mode. All bits are set by the hardware and cleared when the register is read when debug mode is disabled, or if the processor is in debug mode. Attempts to write to this register are ignored. When the hardware sets a bit in this register, debug mode is entered only if debug mode is enabled and the corresponding mask bit in the DER is set.

All bits are cleared to zero following reset.

## ECR — Exception Cause Register

SPR 148

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RESERVED	CHST P	MCE	DSE	ISE	EXTI	ALE	PRE	FPUV E	DECE	RESERVED	SYSE	TR	FPAS E		

RESET:

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

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	SEE	RESERVED										LBRK	IBRK	EBRK D	DPI

RESET:

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

Table 8-35 ECR Bit Settings

Bit(s)	Name	Description
[0:1]	—	Reserved
2	CHSTP	Checkstop bit. Set when the processor enters checkstop state.
3	MCE	Machine check interrupt bit. Set when a machine check exception (other than one caused by a data storage or instruction storage error) is asserted.
4	DSE	Data storage exception bit. Set when a machine check exception caused by a data storage error is asserted.
5	ISE	Instruction storage exception bit. Set when a machine check exception caused by an instruction storage error is asserted.
6	EXTI	External interrupt bit. Set when the external interrupt is asserted.
7	ALE	Alignment exception bit. Set when the alignment exception is asserted.
8	PRE	Program exception bit. Set when the program exception is asserted.
9	FPUVE	Floating point unavailable exception bit. Set when the program exception is asserted.
10	DECE	Decrementer exception bit. Set when the decrementer exception is asserted.
[11:12]	—	Reserved
13	SYSE	System call exception bit. Set when the system call exception is asserted.
14	TR	Trace exception bit. Set when in single-step mode or when in branch trace mode.
15	FPASE	Floating point assist exception bit. Set when the floating-point assist exception is asserted.
16	—	Reserved
17	SEE	Software emulation exception. Set when the software emulation exception is asserted.
[18:27]		Reserved

# Freescale Semiconductor, Inc.

## Table 8-35 ECR Bit Settings (Continued)

Bit(s)	Name	Description
28	LBRK	L-bus breakpoint exception bit. Set when an L-bus breakpoint is asserted.
29	IBRK	I-bus breakpoint exception bit. Set when an I-bus breakpoint is asserted.
30	EBRK	External breakpoint exception bit. Set when an external breakpoint is asserted (by an on-chip IMB or L-bus module, or by an external device or development system through the development port).
31	DPI	Development port interrupt bit. Set by the development port as a result of a debug station non-maskable request or when debug mode is entered immediately out of reset.

### 8.8.11 Debug Enable Register (DER)

This register enables the user to selectively mask the events that may cause the processor to enter into debug mode.

#### DER — Debug Enable Register

**SPR 149**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
RESERVED	CHST PE	MCEE	DSEE	ISEE	EXTIE	ALEE	PREE	FPU- VEE	DE- CEE	RESERVED	SY- SEE	TRE	FPA- SEE		

RESET:

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

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	SEEE	RESERVED										LBRK E	IBRKE	EBRK E	DPIE

RESET:

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

**Table 8-36 DER Bit Settings**

Bit(s)	Name	Description
[0:1]	—	Reserved
2	CHSTPE	Checkstop enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
3	MCEE	Machine check exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
4	DSEE	Data storage exception (type of machine check exception) enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
5	ISEE	Instruction storage exception (type of machine check exception) enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
6	EXTIE	External interrupt enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
7	ALEE	Alignment exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
8	PREE	Program exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
9	FPUVEE	Floating point unavailable exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
10	DECEE	Decrementer exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
[11:12] ]	—	Reserved
13	SYSEE	System call exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
14	TRE	Trace exception enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
15	FPASEE	Floating point assist exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
16	—	Reserved

**Table 8-36 DER Bit Settings (Continued)**

Bit(s)	Name	Description
17	SEEE	Software emulation exception enable bit 0 = Debug mode entry disabled (reset value) 1 = Debug mode entry enabled
[18:27] ]	—	Reserved
28	LBRKE	L-bus breakpoint exception enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
29	IBRKE	I-bus breakpoint exception enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
30	EBRKE	External breakpoint exception enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)
31	DPIE	Development port interrupt enable bit 0 = Debug mode entry disabled 1 = Debug mode entry enabled (reset value)

## SECTION 9 INSTRUCTION SET

This section describes individual instructions, including a description of instruction formats and notation and an alphabetical listing of RCPU instructions by mnemonic.

### 9.1 Instruction Formats

Instructions are four bytes long and word-aligned, so when instruction addresses are presented to the processor (as in branch instructions) the two low-order bits are ignored. Similarly, whenever the processor develops an instruction address, its two low-order bits are zero.

Bits 0 to 5 always specify the primary opcode. Many instructions also have a secondary opcode. The remaining bits of the instruction contain one or more fields for the different instruction formats.

Some instruction fields are reserved or must contain a predefined value as shown in the individual instruction layouts. If a reserved field does not have all bits set to zero, or if a field that must contain a particular value does not contain that value, the instruction form is invalid.

#### 9.1.1 Split Field Notation

Some instruction fields occupy more than one contiguous sequence of bits or occupy a contiguous sequence of bits used in permuted order. Such a field is called a split field. In the format diagrams and in the individual instruction layouts, the name of a split field is shown in small letters, once for each of the contiguous sequences. In the pseudocode description of an instruction having a split field and in some places where individual bits of a split field are identified, the name of the field in small letters represents the concatenation of the sequences from left to right. Otherwise, the name of the field is capitalized and represents the concatenation of the sequences in some order, which need not be left to right, as described for each affected instruction.

#### 9.1.2 Instruction Fields

**Table 9-1** describes the instruction fields used in the various instruction formats.

## Table 9-1 Instruction Formats

Field	Bits	Description
AA	30	<p>Absolute address bit</p> <p>0 The immediate field represents an address relative to the current instruction address. The effective address of the branch is either the sum of the LI field sign-extended to 32 bits and the address of the branch instruction or the sum of the BD field sign-extended to 32 bits and the address of the branch instruction.</p> <p>1 The immediate field represents an absolute address. The effective address of the branch is the LI field sign-extended to 32 bits or the BD field sign-extended to 32 bits.</p>
crbA	11:15	Field used to specify a bit in the CR to be used as a source.
crbB	16:20	Field used to specify a bit in the CR to be used as a source.
BD	16:29	Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 32 bits.
crfD	6:8	Field used to specify one of the CR fields or one of the FPSCR fields as a destination.
crfS	11:13	Field used to specify one of the CR fields or one of the FPSCR fields as a source.
BI	11:15	Field used to specify a bit in the CR to be used as the condition of a branch conditional instruction.
BO	6:10	Field used to specify options for the branch conditional instructions. The encoding is described in <a href="#">4.6 Flow Control Instructions</a> .
crbD	6:10	Field used to specify a bit in the CR or in the FPSCR as the destination of the result of an instruction.
CRM	12:19	Field mask used to identify the CR fields that are to be updated by the <b>mtcrf</b> instruction.
d	16:31	Immediate field specifying a 16-bit signed two's complement integer that is sign-extended to 32 bits.
FM	7:14	Field mask used to identify the FPSCR fields that are to be updated by the <b>mtfsf</b> instruction.
frA	11:15	Field used to specify an FPR as a source of an operation.
frB	16:20	Field used to specify an FPR as a source of an operation.
frC	21:25	Field used to specify an FPR as a source of an operation.
frS	6:10	Field used to specify an FPR as a source of an operation.
frD	6:10	Field used to specify an FPR as the destination of an operation.
IMM	16:19	Immediate field used as the data to be placed into a field in the FPSCR.
LI	6:29	Immediate field specifying a 24-bit, signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 32 bits.
LK	31	<p>Link bit.</p> <p>0 Does not update the link register.</p> <p>1 Updates the link register. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed into the link register.</p>
MB, M	21:25, 26:30	Fields used in rotate instructions to specify a 32-bit mask consisting of 1-bits from bit MB+32 through bit ME+32 inclusive, and 0-bits elsewhere, as described in <a href="#">4.3.4 Integer Rotate and Shift Instructions</a> .



**Table 9-1 Instruction Formats (Continued)**

Field	Bits	Description
NB	16:20	Field used to specify the number of bytes to move in an immediate string load or store.
opcode	0:5	Primary opcode field.
OE	21	Used for extended arithmetic to enable setting OV and SO in the XER.
rA	11:15	Field used to specify a GPR to be used as a source or as a destination.
rB	16:20	Field used to specify a GPR to be used as a source.
Rc	31	Record bit 0 Does not update the condition register. 1 Updates the condition register (CR) to reflect the result of the operation. For integer instructions, CR[0:3] are set to reflect the result as a signed quantity. The result as an unsigned quantity or a bit string can be deduced from the EQ bit. For floating-point instructions, CR[4:7] are set to reflect floating-point exception, floating-point enabled exception, floating-point invalid operation exception, and floating-point overflow exception.
rS	6:10	Field used to specify a GPR to be used as a source.
rD	6:10	Field used to specify a GPR to be used as a destination.
SH	16:20	Field used to specify a shift amount.
SIMM	16:31	Immediate field used to specify a 16-bit signed integer.
SPR	11:20	Field used to specify a special purpose register for the <b>mtspr</b> and <b>mfspr</b> instructions. The encoding is described in <a href="#">4.7.2 Move to/from Special Purpose Register Instructions</a> .
TO	6:10	Field used to specify the conditions on which to trap. The encoding is described in <a href="#">4.6.7 Trap Instructions</a> .
UIMM	16:31	Immediate field used to specify a 16-bit unsigned integer.
XO	21:30, 22:30, 26:30, or 30	Secondary opcode field.

### 9.1.3 Notation and Conventions

The operation of some instructions is described by a register transfer language (RTL). See [Table 9-2](#) for a list of RTL notation and conventions used throughout this chapter.

**Table 9-2 RTL Notation and Conventions**

Notation/Convention	Meaning
$\leftarrow$	Assignment
$\neg$	NOT logical operator
$*$	Multiplication
$\div$	Division (yielding quotient)
$+$	Two's-complement addition
$-$	Two's-complement subtraction, unary minus
$=, \neq$	Equals and Not Equals relations
$<, \leq, >, \geq$	Signed comparison relations
$<U, >U$	Unsigned comparison relations
$?$	Unordered comparison relation
$\&,  $	AND, OR logical operators
$  $	Used to describe the concatenation of two values (i.e., 010    111 is the same as 010111)
$\oplus, \equiv$	Exclusive-OR, Equivalence logical operators ( $(a \equiv b) = (a \oplus \neg b)$ )
$0bnnnn$	A number expressed in binary format
$0xnnnn$	A number expressed in hexadecimal format
$(rA 0)$	The contents of $rA$ if the $rA$ field has the value 1–31, or the value 0 if the $rA$ field is 0
$\cdot$ (period)	As the last character of an instruction mnemonic, a period ( $\cdot$ ) means that the instruction updates the condition register field.
$CEIL(x)$	Least integer $\geq x$
$DOUBLE(x)$	Result of converting $x$ from floating-point single format to floating-point double format.
$EXTS(x)$	Result of extending $x$ on the left with sign bits
$GPR(x)$	General Purpose Register $x$
$MASK(x, y)$	Mask having ones in positions $x$ through $y$ (wrapping if $x > y$ ) and 0's elsewhere
$MEM(x, y)$	Contents of $y$ bytes of memory starting at address $x$
$ROTL[32](x, y)$	Result of rotating the 64-bit value $x$ left $y$ positions, where $x$ is 32 bits long
$SINGLE(x)$	Result of converting $x$ from floating-point double format to floating-point single format.
$SPR(x)$	Special Purpose Register $x$
$x(n)$	$x$ is raised to the $n$ th power
$(n)x$	The replication of $x$ , $n$ times (i.e., $x$ concatenated to itself $n-1$ times). $(n)0$ and $(n)1$ are special cases
$x[n]$	$n$ is a bit or field within $x$ , where $x$ is a register
TRAP	Invoke the system trap handler

**Table 9-2 RTL Notation and Conventions (Continued)**

Notation/Convention	Meaning
undefined	An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.
characterization	Reference to the setting of status bits, in a standard way that is explained in the text
CIA	Current instruction address, which is the 32-bit address of the instruction being described by a sequence of pseudocode. Used by relative branches to set the next instruction address (NIA). Does not correspond to any architected register.
NIA	Next instruction address, which is the 32-bit address of the next instruction to be executed (the branch destination) after a successful branch. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions which do not branch, the next instruction address is CIA +4.
if...then...else...	Conditional execution, indenting shows range, else is optional
do	Do loop, indenting shows range. To and/or by clauses specify incrementing an iteration variable, and while and/or until clauses give termination conditions, in the usual manner.
leave	Leave innermost do loop, or do loop described in leave statement

Precedence rules for RTL operators are summarized in [Table 9-3](#).

**Table 9-3 Precedence Rules**

Operators	Associativity
$x[n]$ , function evaluation	Left to right
$(n)x$ or replication, $x(n)$ or exponentiation	Right to left
unary $-$ , $\neg$	Right to left
$*$ , $\div$	Left to right
$+$ , $-$	Left to right
$\parallel$	Left to right
$=, !, <, \leq, >, \geq, <U, >U, ?$	Left to right
$\&, \oplus, \equiv$	Left to right
$ $	Left to right
$-$ (range)	None
$\leftarrow$	None

Note that operators higher in [Table 9-3](#) are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown.

The remainder of this chapter lists and describes the RCPU instruction set. The instructions are listed in alphabetical order by mnemonic. **Figure 9-1** shows the format for each instruction description page.



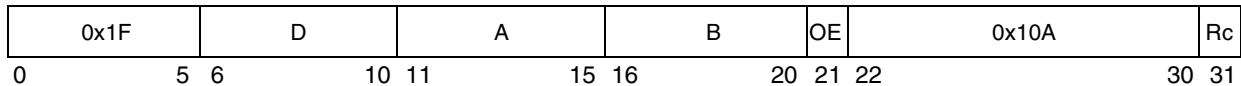
add<sub>x</sub>

Add

add<sub>x</sub>

Integer Unit

<b>add</b>	<b>rD,rA,rB</b>	<b>(OE=0 Rc=0)</b>
<b>add.</b>	<b>rD,rA,rB</b>	<b>(OE=0 Rc=1)</b>
<b>addo</b>	<b>rD,rA,rB</b>	<b>(OE=1 Rc=0)</b>
<b>addo.</b>	<b>rD,rA,rB</b>	<b>(OE=1 Rc=1)</b>



$rD \leftarrow (rA) + (rB)$   
The sum (rA) + (rB) is placed into rD.  
Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

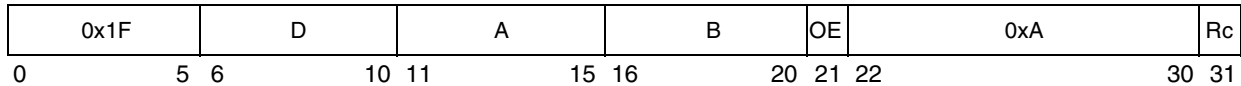
addc<sub>x</sub>

Add Carrying

addc<sub>x</sub>

Integer Unit

addc	rD,rA,rB	(OE=0 Rc=0)
addc.	rD,rA,rB	(OE=0 Rc=1)
addco	rD,rA,rB	(OE=1 Rc=0)
addco.	rD,rA,rB	(OE=1 Rc=1)



$rD \leftarrow (rA) + (rB)$   
The sum (rA) + (rB) is placed into rD.  
Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: CA  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

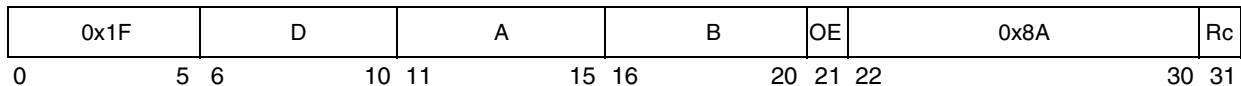
adde<sub>x</sub>

Add Extended

adde<sub>x</sub>

Integer Unit

adde	rD,rA,rB	(OE=0 Rc=0)
adde.	rD,rA,rB	(OE=0 Rc=1)
addeo	rD,rA,rB	(OE=1 Rc=0)
addeo.	rD,rA,rB	(OE=1 Rc=1)



$rD \leftarrow (rA) + (rB) + XER[CA]$   
The sum  $(rA) + (rB) + XER[CA]$  is placed into rD.  
Other registers altered:

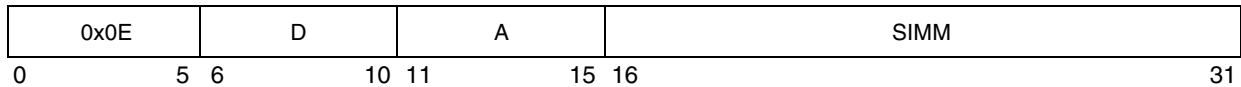
- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: CA  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

addi
Add Immediate

addi
Integer Unit

addi rD,rA,SIMM



if rA=0 then
rD←EXTS(SIMM)
else
rD←(rA)+EXTS(SIMM)
The sum (rA| 0) + SIMM is placed into rD.
Other registers altered:
• None

Table 9-4 Simplified Mnemonics for addi Instruction

Table with 3 columns: Simplified Mnemonic, Operands, and Equivalent To. It lists three mnemonics: la, li, and subi, along with their operands and the equivalent addi instruction.

This instruction is defined by the PowerPC UISA.



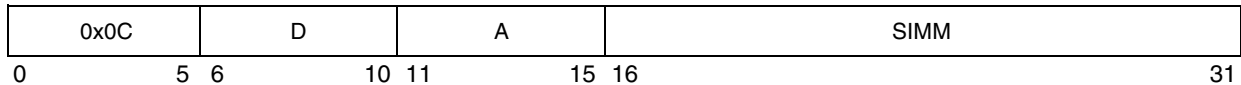
# addic

Add Immediate Carrying

# addic

Integer Unit

**addic**            **rD,rA,SIMM**



$rD \leftarrow (rA) + EXTS(SIMM)$

The sum (rA) + SIMM is placed into rD.

Other registers altered:

- XER:  
Affected: CA

**Table 9-5 Simplified Mnemonics for addic Instruction**

Simplified Mnemonic	Operands	Equivalent To
subic	rD,rA,value	addic rD,rA,-value

This instruction is defined by the PowerPC UISA.

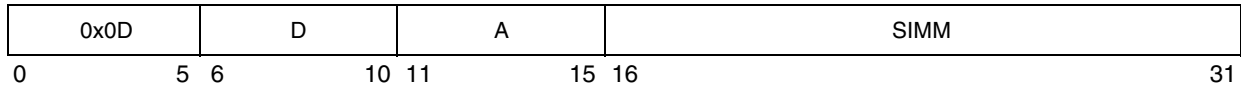
# addic.

Add Immediate Carrying and Record

# addic.

Integer Unit

**addic.**            **rD,rA,SIMM**



$$rD \leftarrow (rA) + \text{EXTS}(\text{SIMM})$$

The sum (rA) + SIMM is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO
- XER:  
Affected: CA

**Table 9-6 Simplified Mnemonics for addic. Instruction**

Simplified Mnemonic	Operands	Equivalent To
subic.	rD,rA,value	addic. rD,rA,-value

This instruction is defined by the PowerPC UISA.

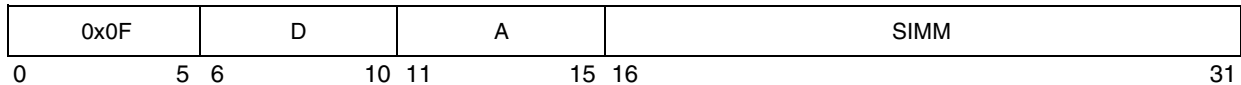
# addis

Add Immediate Shifted

# addis

Integer Unit

**addis**            **rD,rA,SIMM**



if **rA**=0 then  
     **rD**←(SIMM || (16)0)  
 else  
     **rD**←(**rA**)+(SIMM || (16)0)

The sum (**rA** | 0) + (SIMM || 0x0000) is placed into **rD**.

Other registers altered:

- None

**Table 9-7 Simplified Mnemonics for addis Instruction**

Simplified Mnemonic	Operands	Equivalent To
lis	rA,value	addi rA,0,value
subis	rD,rA,value	addis rD,rA,-value

This instruction is defined by the PowerPC UISA.

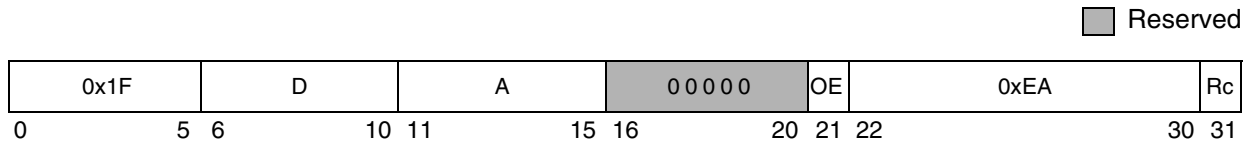
# addmex

Add to Minus One Extended

# addmex

Integer Unit

<b>addme</b>	<b>rD,rA</b>	(OE=0 Rc=0)
<b>addme.</b>	<b>rD,rA</b>	(OE=0 Rc=1)
<b>addmeo</b>	<b>rD,rA</b>	(OE=1 Rc=0)
<b>addmeo.</b>	<b>rD,rA</b>	(OE=1 Rc=1)



$rD \leftarrow (rA) + XER[CA] - 1$   
 The sum (rA)+XER[CA]+0xFFFF FFFF is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
 Affected: CA  
 Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

# addze<sub>x</sub>

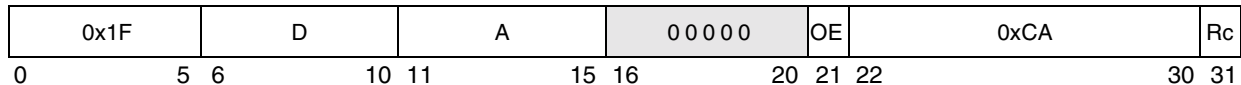
Add to Zero Extended

# addze<sub>x</sub>

Integer Unit

<b>addze</b>	<b>rD,rA</b>	(OE=0 Rc=0)
<b>addze.</b>	<b>rD,rA</b>	(OE=0 Rc=1)
<b>addzeo</b>	<b>rD,rA</b>	(OE=1 Rc=0)
<b>addzeo.</b>	<b>rD,rA</b>	(OE=1 Rc=1)

Reserved



$$rD \leftarrow (rA) + XER[CA]$$

The sum (rA)+XER[CA] is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: CA  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

**and<sub>x</sub>**

AND

**and<sub>x</sub>**

Integer Unit

**and**                      **rA,rS,rB**                      (Rc=0)  
**and.**                      **rA,rS,rB**                      (Rc=1)

0x1F				S				A				B				0x1C																Rc
0				5	6			10	11			15	16			20	21														30	31

$rA \leftarrow (rS) \& (rB)$

The contents of **rS** is ANDed with the contents of **rB** and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

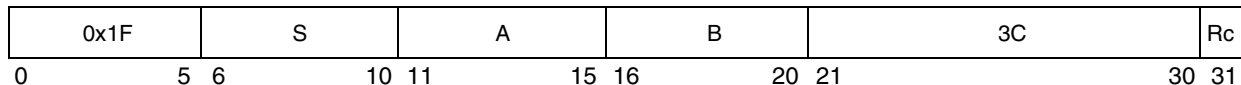
# andc<sub>x</sub>

AND with Complement

# andc<sub>x</sub>

Integer Unit

**andc**                      **rA,rS,rB**                      (Rc=0)  
**andc.**                      **rA,rS,rB**                      (Rc=1)



$$rA \leftarrow (rS) \& \neg (rB)$$

The contents of **rS** is ANDed with the one's complement of the contents of **rB** and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

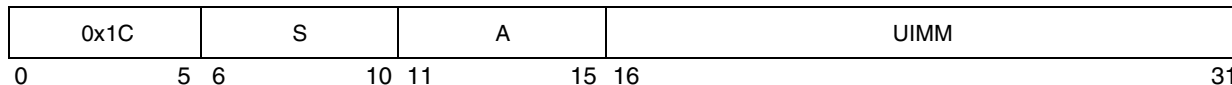
**andi.**

AND Immediate

**andi.**

Integer Unit

**andi.**            **rA,rS,UIMM**



$rA \leftarrow (rS) \& ((16)0 \parallel UIMM)$

The contents of **rS** are ANDed with 0x0000 || UIMM and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.



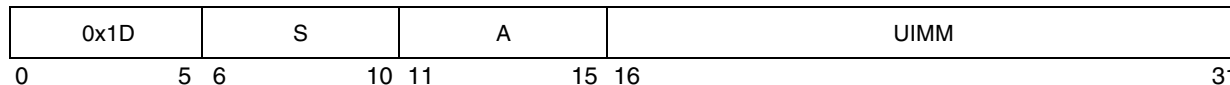
**andis.**

AND Immediate Shifted

**andis.**

Integer Unit

**andis.**      **rA,rS,UIMM**



$$rA \leftarrow (rS) + (UIMM \parallel (16)0)$$

The contents of **rS** are ANDed with **UIMM**  $\parallel$  0x0000 and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.

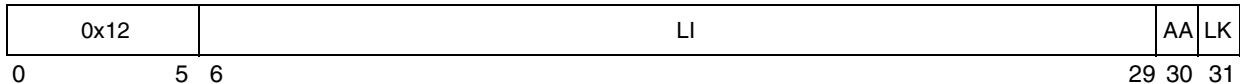
**b<sub>x</sub>**

Branch

**b<sub>x</sub>**

Branch Processing Unit

<b>b</b>	target_addr	(AA=0 LK=0)
<b>ba</b>	target_addr	(AA=1 LK=0)
<b>bl</b>	target_addr	(AA=0LK=1)
<b>bla</b>	target_addr	(AA=1 LK=1)



```

if AA then
    NIA←EXTS(LI || 0b00)
else
    NIA←CIA+EXTS(LI || 0b00)
if LK, then
    LR←CIA+4
    
```

target\_addr specifies the branch target address.

If AA=0, then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction.

If AA=1, then the branch target address is the value LI || 0b00 sign-extended.

If LK=1, then the effective address of the instruction following the branch instruction is placed into the link register.

Other registers affected:

Link Register (LR) (if LK=1)

This instruction is defined by the PowerPC UISA.

# bc<sub>x</sub>

Branch Conditional

# bc<sub>x</sub>

Branch Processing Unit

<b>bc</b>	BO,BI,target_addr	(AA=0 LK=0)
<b>bca</b>	BO,BI,target_addr	(AA=1 LK=0)
<b>bcl</b>	BO,BI,target_addr	(AA=0 LK=1)
<b>bcla</b>	BO,BI,target_addr	(AA=1 LK=1)

0x10	BO	BI	BD	AA	LK
0	5 6	10 11	15 16	29 30	31

```

if ¬ BO[2], then CTR ← CTR-1
ctr_ok ← BO[2] | ((CTR[0] ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
    if AA then
        NIA ← EXTS(BD || 0b00)
    else
        NIA ← CIA+EXTS(BD || 0b00)
if LK, then
    LR ← CIA+4
    
```

The BI field specifies the bit in the Condition Register (CR) to be used as the condition of the branch. The BO field is used as described above.

target\_addr specifies the branch target address.

If AA=0, the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction.

If AA=1, the branch target address is the value BD || 0b00 sign-extended.

If LK=1, the effective address of the instruction following the branch instruction is placed into the link register.

Other registers affected:

Count Register (CTR)	(if BO[2]=0)
Link Register (LR)	(if LK=1)

This instruction is defined by the PowerPC UISA.

**Table 9-8 Simplified Mnemonics for  
bc, bca, bcl, and bcla Instructions**

Operation	Simplified Mnemonic <sup>1</sup>	Equivalent To
Decrement CTR, branch if CTR non-zero	<b>bdnz</b> target	<b>bc 16,0,target</b>
Decrement CTR, branch absolute if CTR non-zero	<b>bdnza</b> target	<b>bca 16,0,target</b>
Decrement CTR, branch and update LR if CTR non-zero	<b>bdnzi</b> target	<b>bcl 16,0,target</b>
Decrement CTR, branch absolute and update LR if CTR non-zero	<b>bdnzla</b> target	<b>bcla 16,0,target</b>
Decrement CTR, branch if false and CTR non-zero	<b>bdnzf</b> BI,target	<b>bc 0,BI,target</b>
Decrement CTR, branch absolute if false and CTR non-zero	<b>bdnzfa</b> BI,target	<b>bca 0,BI,target</b>
Decrement CTR, branch and update LR if false and CTR non-zero	<b>bdnzfi</b> BI,target	<b>bcl 0,BI,target</b>
Decrement CTR, branch absolute and update LR if false and CTR non-zero	<b>bdnzfla</b> BI,target	<b>bcla 0,BI,target</b>
Decrement CTR, branch if true and CTR non-zero	<b>bdnzt</b> BI,target	<b>bc 8,BI,target</b>
Decrement CTR, branch absolute if true and CTR non-zero	<b>bdnzta</b> BI,target	<b>bca 8,BI,target</b>
Decrement CTR, branch and update LR if true and CTR non-zero	<b>bdnzti</b> BI,target	<b>bcl 8,BI,target</b>
Decrement CTR, branch absolute and update LR if true and CTR non-zero	<b>bdnztia</b> BI,target	<b>bcla 8,BI,target</b>
Decrement CTR, branch if CTR zero	<b>bdz</b> target	<b>bc 18,0,target</b>
Decrement CTR, branch absolute if CTR zero	<b>bdza</b> target	<b>bca 18,0,target</b>
Decrement CTR, branch and update LR if CTR zero	<b>bdzi</b> target	<b>bcl 18,0,target</b>
Decrement CTR, branch absolute and update LR if CTR zero	<b>bdzla</b> target	<b>bcla 18,0,target</b>
Decrement CTR, branch if false and CTR zero	<b>bdzf</b> BI,target	<b>bc 2,BI,target</b>
Decrement CTR, branch absolute if false and CTR zero	<b>bdzfa</b> BI,target	<b>bca 2,BI,target</b>
Decrement CTR, branch and update LR if false and CTR zero	<b>bdzfi</b> BI,target	<b>bcl 2,BI,target</b>
Decrement CTR, branch absolute and update LR if false and CTR zero	<b>bdzfla</b> BI,target	<b>bcla 2,BI,target</b>
Decrement CTR, branch if true and CTR zero	<b>bdzt</b> BI,target	<b>bc 10,BI,target</b>
Decrement CTR, branch absolute if true and CTR zero	<b>bdzta</b> BI,target	<b>bca 10,BI,target</b>
Decrement CTR, branch and update LR if true and CTR zero	<b>bdzti</b> BI,target	<b>bcl 10,BI,target</b>

## Table 9-8 Simplified Mnemonics for bc, bca, bcl, and bcla Instructions (Continued)

Operation	Simplified Mnemonic <sup>1</sup>	Equivalent To
Decrement CTR, branch absolute and update LR if true and CTR zero	<b>bdztla</b> Bl,target	<b>bcla 10,Bl,target</b>
Branch if equal	<b>beq</b> crX,target	<b>bc 12, 4*crX+2,target</b>
Branch absolute if equal	<b>beqa</b> crX,target	<b>bca 12, 4*crX+2,target</b>
Branch and update LR if equal	<b>beql</b> crX,target	<b>bcl 12, 4*crX+2,target</b>
Branch absolute and update LR if equal	<b>beqla</b> crX,target	<b>bcla 12, 4*crX+2,target</b>
Branch if false	<b>bf</b> Bl,target	<b>bc 4,Bl,target</b>
Branch absolute if false	<b>bfa</b> Bl,target	<b>bca 4,Bl,target</b>
Branch and update LR if false	<b>bfl</b> Bl,target	<b>bcl 4,Bl,target</b>
Branch absolute and update LR if false	<b>bfla</b> Bl,target	<b>bcla 4,Bl,target</b>
Branch if greater than or equal to	<b>bge</b> crX,target	<b>bc 4,4*crX,target</b>
Branch absolute if greater than or equal to	<b>bgea</b> crX,target	<b>bca 4,4*crX,target</b>
Branch and update LR if greater than or equal to	<b>bgel</b> crX,target	<b>bcl 4,4*crX,target</b>
Branch absolute and update LR if greater than or equal to	<b>bgela</b> crX,target	<b>bcla 4,4*crX,target</b>
Branch if greater than	<b>bgt</b> crX,target	<b>bc 12,4*crX+1,target</b>
Branch absolute if greater than	<b>bgta</b> crX,target	<b>bca 12,4*crX+1,target</b>
Branch and update LR if greater than	<b>bgtl</b> crX,target	<b>bcl 12,4*crX+1,target</b>
Branch absolute and update LR if greater than	<b>bgtla</b> crX,target	<b>bcla 12,4*crX+1,target</b>
Branch if less than or equal to	<b>ble</b> crX,target	<b>bc 4,4*crX+1,target</b>
Branch absolute if less than or equal to	<b>blea</b> crX,target	<b>bca 4,4*crX+1,target</b>
Branch and update LR if less than or equal to	<b>blel</b> crX,target	<b>bcl 4,4*crX+1,target</b>
Branch absolute and update LR if less than or equal to	<b>blela</b> crX,target	<b>bcla 4,4*crX+1,target</b>
Branch if less than	<b>blt</b> crX,target	<b>bc 12,4*crX,target</b>
Branch absolute if less than	<b>blta</b> crX,target	<b>bca 12,4*crX,target</b>
Branch and update LR if less than	<b>bltl</b> crX,target	<b>bcl 12,4*crX,target</b>
Branch absolute and update LR if less than	<b>bltla</b> crX,target	<b>bcla 12,4*crX,target</b>
Branch if not equal to	<b>bne</b> crX,target	<b>bc 4,4*crX+2,target</b>
Branch absolute if not equal to	<b>bnea</b> crX,target	<b>bca 4,4*crX+2,target</b>
Branch and update LR if not equal to	<b>bnel</b> crX,target	<b>bcl 4,4*crX+2,target</b>
Branch absolute and update LR if not equal to	<b>bnela</b> crX,target	<b>bcla 4,4*crX+2,target</b>
Branch if not greater than	<b>bng</b> crX,target	<b>bc 4,4*crX+1,target</b>

**Table 9-8 Simplified Mnemonics for  
bc, bca, bcl, and bcla Instructions (Continued)**

Operation	Simplified Mnemonic <sup>1</sup>	Equivalent To
Branch absolute if not greater than	<b>bnga crX,target</b>	<b>bca 4,4*crX+1,target</b>
Branch and update LR if not greater than	<b>bngl crX,target</b>	<b>bcl 4,4*crX+1,target</b>
Branch absolute and update LR if not greater than	<b>bngla crX,target</b>	<b>bcla 4,4*crX+1,target</b>
Branch if not less than	<b>bnl crX,target</b>	<b>bc 4,4*crX,target</b>
Branch absolute if not less than	<b>bnla crX,target</b>	<b>bca 4,4*crX,target</b>
Branch and update LR if not less than	<b>bnll crX,target</b>	<b>bcl 4,4*crX,target</b>
Branch absolute and update LR if not less than	<b>bnlla crX,target</b>	<b>bcla 4,4*crX,target</b>
Branch if not summary overflow	<b>bns crX,target</b>	<b>bc 4,4*crX+3,target</b>
Branch absolute if not summary overflow	<b>bnsa crX,target</b>	<b>bca 4,4*crX+3,target</b>
Branch and update LR if not summary overflow	<b>bnsi crX,target</b>	<b>bcl 4,4*crX+3,target</b>
Branch absolute and update LR if not summary overflow	<b>bnsia crX,target</b>	<b>bcla 4,4*crX+3,target</b>
Branch if not unordered	<b>bnu crX,target</b>	<b>bc 4,4*crX+3,target</b>
Branch absolute if not unordered	<b>bnu a crX,target</b>	<b>bca 4,4*crX+3,target</b>
Branch and update LR if not unordered	<b>bnul crX,target</b>	<b>bcl 4,4*crX+3,target</b>
Branch absolute and update LR if not unordered	<b>bnula crX,target</b>	<b>bcla 4,4*crX+3,target</b>
Branch if summary overflow	<b>bs o crX,target</b>	<b>bc 12,4*crX+3,target</b>
Branch absolute if summary overflow	<b>bs o a crX,target</b>	<b>bca 12,4*crX+3,target</b>
Branch and update LR if summary overflow	<b>bs ol crX,target</b>	<b>bcl 12,4*crX+3,target</b>
Branch absolute and update LR if summary overflow	<b>bs ola crX,target</b>	<b>bcla 12,4*crX+3,target</b>
Branch if true	<b>bt BI,target</b>	<b>bc 12,BI,target</b>
Branch absolute if true	<b>bta BI,target</b>	<b>bca 12,BI,target</b>
Branch and update LR if true	<b>btl BI,target</b>	<b>bcl 12,BI,target</b>
Branch absolute and update LR if true	<b>btla BI,target</b>	<b>bcla 12,BI,target</b>
Branch if unordered	<b>bun crX,target</b>	<b>bc 12,4*crX+3,target</b>
Branch absolute if unordered	<b>buna crX,target</b>	<b>bca 12,4*crX+3,target</b>
Branch and update LR if unordered	<b>bunl crX,target</b>	<b>bcl 12,4*crX+3,target</b>
Branch and update LR if unordered	<b>bunla crX,target</b>	<b>bcla 12,4*crX+3,target</b>

**NOTES:**

1. If **crX** is not included in the operand list (for operations that use a **cr** field), **cr0** is assumed.

Refer to **APPENDIX E SIMPLIFIED MNEMONICS** for more information on simplified mnemonics.

## Branch Processing Unit

<b>bcctr</b>	BO,BI	(LK=0)
<b>bcctrl</b>	BO,BI	(LK=1)

0x13					BO					BI					00000					0x210										LK																					
0					5					6					10					11					15					16					20					21					30					31	

```

cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if cond_ok then
    NIA ← CTR[0:29] || 0b00
    if LK then
        LR ← CIA+4

```

**Table 9-9** provides simplified mnemonics for the **bcctr** and **bcctrl** instructions. Refer to **APPENDIX E SIMPLIFIED MNEMONICS** for more information on simplified mnemonics.

**Table 9-9 Simplified Mnemonics for  
bcctr and bcctrl Instructions**

Operation	Simplified Mnemonic <sup>1</sup>	Equivalent To
Branch to CTR	<b>bctr</b>	<b>bcctr 20,0</b>
Branch to CTR and update LR	<b>bcctl</b>	<b>bcctrl 20,0</b>
Branch if equal to CTR	<b>beqctr crX</b>	<b>bcctr 12, 4*crX+2</b>
Branch if equal to CTR, update LR	<b>beqctrl crX</b>	<b>bcctrl 12, 4*crX+2</b>
Branch if false to CTR	<b>bfctr BI</b>	<b>bcctr 4,BI</b>
Branch if false to CTR, update LR	<b>bfctrl BI</b>	<b>bcctrl 4,BI</b>
Branch to CTR if greater than or equal to	<b>bgectr crX</b>	<b>bcctr 4,4*crX</b>
Branch to CTR if greater than or equal to, update LR	<b>bgectrl crX</b>	<b>bcctrl 4,4*crX</b>
Branch to CTR if greater than	<b>bgtctr crX</b>	<b>bcctr 12,4*crX+1</b>
Branch to CTR if greater than, update LR	<b>bgtctrl crX</b>	<b>bcctrl 12,4*crX+1</b>
Branch to CTR if less than or equal to	<b>blectr crX</b>	<b>bcctr 4,4*crX+1</b>
Branch to CTR if less than or equal to, update LR	<b>blectrl crX</b>	<b>bcctrl 4,4*crX+1</b>
Branch to CTR if less than	<b>bltctr crX</b>	<b>bcctr 12,4*crX</b>
Branch to CTR if less than, update LR	<b>bltctrl crX</b>	<b>bcctrl 12,4*crX</b>
Branch to CTR if not equal to	<b>bnctr crX</b>	<b>bcctr 4,4*crX+2</b>
Branch to CTR if not equal to, update LR	<b>bnctrl crX</b>	<b>bcctrl 4,4*crX+2</b>
Branch to CTR if not greater than	<b>bngctr crX</b>	<b>bcctr 4,4*crX+1</b>
Branch to CTR if not greater than, update LR	<b>bngctrl crX</b>	<b>bcctrl 4,4*crX+1</b>
Branch to CTR if not less than	<b>bnlctr crX</b>	<b>bcctr 4,4*crX</b>
Branch to CTR if not less than, update LR	<b>bnlctrl crX</b>	<b>bcctrl 4,4*crX</b>
Branch to CTR if not summary overflow	<b>bsnctr crX</b>	<b>bcctr 4,4*crX+3</b>
Branch to CTR if not summary overflow, update LR	<b>bsnctrl crX</b>	<b>bcctrl 4,4*crX+3</b>
Branch to CTR if not unordered	<b>bnuctr crX</b>	<b>bcctr 4,4*crX+3</b>
Branch to CTR if not unordered, update LR	<b>bnuctrl crX</b>	<b>bcctrl 4,4*crX+3</b>
Branch to CTR if summary overflow	<b>bsoctr crX</b>	<b>bcctr 12,4*crX+3</b>
Branch to CTR if summary overflow, update LR	<b>bsoctrl crX</b>	<b>bcctrl 12,4*crX+3</b>
Branch to CTR if true	<b>btctr BI</b>	<b>bcctr 12,BI</b>
Branch to CTR if true, update LR	<b>btctrl BI</b>	<b>bcctrl 12,BI</b>
Branch to CTR if unordered	<b>bunctr crX</b>	<b>bcctr 12,4*crX+3</b>
Branch to CTR if unordered, update LR	<b>bunctrl crX</b>	<b>bcctrl 12,4*crX+3</b>

**NOTES:**

1. If **crX** is not included in the operand list (for operations that use a **cr** field), **cr0** is assumed.



# bclr<sub>x</sub>

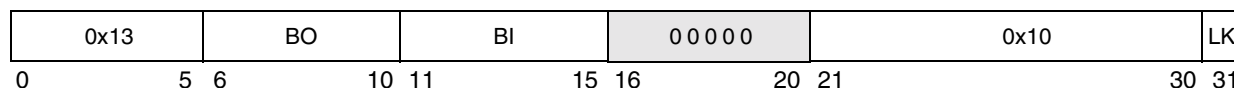
Branch Conditional to Link Register

# bclr<sub>x</sub>

Branch Processing Unit

**bclr** BO,BI (LK=0)  
**bclrl** BO,BI (LK=1)

Reserved



```

if ¬ BO[2] then
    CTR ← CTR-1
ctr_ok ← BO[2] | ((CTR[0] ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
    NIA ← LR[0:29] || 0b00
if LK then
    LR ← CIA+4
    
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is used as described above, and the branch target address is LR[0:29] || 0b00.

If LK=1 then the effective address of the instruction following the branch instruction is placed into the link register.

Other registers affected:

Count Register (CTR) (if BO[2]=0)  
 Link Register (LR) (if LK=1)

This instruction is defined by the PowerPC UISA.

## Table 9-10 Simplified Mnemonics for bclr and bclrl Instructions

Operation	Simplified Mnemonic <sup>1</sup>	Equivalent To
Decrement CTR, branch to LR if false and CTR non-zero	<b>bdnzflr</b> BI	<b>bclr</b> 0,BI
Decrement CTR, branch to LR if false and CTR non-zero, update LR	<b>bdnzflrl</b> BI	<b>bclrl</b> 0,BI
Decrement CTR, branch to LR if CTR non-zero	<b>bdnzlr</b>	<b>bclr</b> 16,0
Decrement CTR, branch to LR if CTR non-zero, update LR	<b>bdnzlrl</b>	<b>bclrl</b> 16,0
Decrement CTR, branch to LR if true and CTR non-zero	<b>bdnztlr</b> BI	<b>bclr</b> 8,BI
Decrement CTR, branch to LR if true and CTR non-zero, update LR	<b>bdnztlrl</b> BI	<b>bclrl</b> 8,BI
Decrement CTR, branch to LR if false and CTR zero	<b>bdzflr</b> BI	<b>bclr</b> 2,BI
Decrement CTR, branch to LR if false and CTR zero, update LR	<b>bdzflrl</b> BI	<b>bclrl</b> 2,BI
Decrement CTR, branch to LR if CTR zero	<b>bdzlr</b>	<b>bclr</b> 18,0
Decrement CTR, branch to LR if CTR zero, update LR	<b>bdzlrl</b>	<b>bclrl</b> 18,0
Decrement CTR, branch to LR if true and CTR zero	<b>bdztlr</b> BI	<b>bclr</b> 10,BI
Decrement CTR, branch to LR if true and CTR zero, update LR	<b>bdztlrl</b> BI	<b>bclrl</b> 10,BI
Branch to LR if equal	<b>beqlr</b> crX	<b>bclr</b> 12, 4*crX+2
Branch to LR if equal, update LR	<b>beqlrl</b> crX	<b>bclrl</b> 12, 4*crX+2
Branch to LR if false	<b>bflr</b> BI	<b>bclr</b> 4,BI
Branch to LR if false, update LR	<b>bflrl</b> BI	<b>bclrl</b> 4,BI
Branch to LR if greater than or equal to	<b>bgehr</b> crX	<b>bclr</b> 4,4*crX
Branch to LR if greater than or equal to, update LR	<b>bgehlrl</b> crX	<b>bclrl</b> 4,4*crX
Branch to LR if greater than	<b>bgtlr</b> crX	<b>bclr</b> 12,4*crX+1
Branch to LR if greater than, update LR	<b>bgtlrl</b> crX	<b>bclrl</b> 12,4*crX+1
Branch to LR if less than or equal to	<b>blehr</b> crX	<b>bclr</b> 4,4*crX+1
Branch to LR if less than or equal to, update LR	<b>blehlrl</b> crX	<b>bclrl</b> 4,4*crX+1
Branch to LR	<b>blr</b>	<b>bclr</b> 20,0
Branch to LR, update LR	<b>blrl</b>	<b>bclrl</b> 20,0
Branch to LR if less than	<b>bltlr</b> crX	<b>bclr</b> 12,4*crX
Branch to LR if less than, update LR	<b>bltlrl</b> crX	<b>bclrl</b> 12,4*crX

**Table 9-10 Simplified Mnemonics for  
bclr and bclrl Instructions (Continued)**

Operation	Simplified Mnemonic <sup>1</sup>	Equivalent To
Branch to LR if not equal to	<b>bnelr crX</b>	<b>bclr 4,4*crX+2</b>
Branch to LR if not equal to, update LR	<b>bnelrl crX</b>	<b>bclrl 4,4*crX+2</b>
Branch to LR if not greater than	<b>bnlgr crX</b>	<b>bclr 4,4*crX+1</b>
Branch to LR if not greater than, update LR	<b>bnlgrl crX</b>	<b>bclrl 4,4*crX+1</b>
Branch to LR if not less than	<b>bnllr crX</b>	<b>bclr 4,4*crX</b>
Branch to LR if not less than, update LR	<b>bnllrl crX</b>	<b>bclrl 4,4*crX</b>
Branch to LR if not summary overflow	<b>bnslr crX</b>	<b>bclr 4,4*crX+3</b>
Branch to LR if not summary overflow, update LR	<b>bnsrl crX</b>	<b>bclrl 4,4*crX+3</b>
Branch to LR if not unordered	<b>bnulr crX</b>	<b>bclr 4,4*crX+3</b>
Branch to LR if not unordered, update LR	<b>bnulrl crX</b>	<b>bclrl 4,4*crX+3</b>
Branch to LR if summary overflow	<b>bsolr crX</b>	<b>bclr 12,4*crX+3</b>
Branch to LR if summary overflow, update LR	<b>bsolrl crX</b>	<b>bclrl 12,4*crX+3</b>
Branch to LR if true	<b>btlr BI</b>	<b>bclr 12,BI</b>
Branch to LR if true, update LR	<b>btirl BI</b>	<b>bclrl 12,BI</b>
Branch to LR if unordered	<b>bunlr crX</b>	<b>bclr 12,4*crX+3</b>
Branch to LR if unordered, update LR	<b>bunlrl crX</b>	<b>bclrl 12,4*crX+3</b>

**NOTES:**

1. If **crX** is not included in the operand list (for operations that use a **cr** field), **cr0** is assumed.

Refer to **APPENDIX E SIMPLIFIED MNEMONICS** for more information on simplified mnemonics.

# cmp

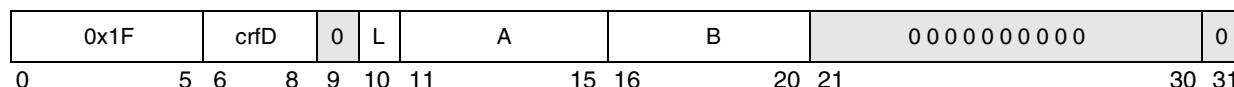
Compare

# cmp

Integer Unit

**cmp**                      **crfD,L,rA,rB**

Reserved



```

a ← (rA)
b ← (rB)
if a < b then
    c ← 0b100
else
    if a > b then
        c ← 0b010
    else
        c ← 0b001
CR[4*crfD:4*crfD+3] ← c || XER[SO]
    
```

The contents of **rA** are compared with the contents of **rB**, treating the operands as signed integers. The result of the comparison is placed into CR Field **crfD**.

The **L** operand controls whether **rA** and **rB** are treated as 32-bit operands (**L**=0) or 64-bit operands (**L**=1). For 32-bit PowerPC implementations such as the RCPU, if **L**=1, the instruction form is invalid.

Other registers altered:

- Condition Register (CR Field specified by operand **crfD**):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.

**Table 9-11 Simplified Mnemonics for cmp Instruction**

Operation	Simplified Mnemonic	Equivalent To
Compare word	<b>cmpw crfD, rA,rB</b> <b>cmp crfD, rA,rB</b>	<b>cmp crfD, 0, rA,rB</b>
Compare word, place result in CR0	<b>cmpw rA,rB</b> <b>cmp rA,rB</b>	<b>cmp 0, 0, rA,rB</b>

# cmpi

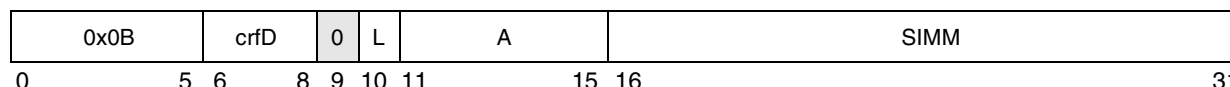
Compare Immediate

# cmpi

Integer Unit

**cmpi**      **crfD,L,rA,SIMM**

Reserved



```

a ← (rA)
if a < EXTS(SIMM) then
    c ← 0b100
else
    if a > EXTS(SIMM) then
        c ← 0b010
    else
        c ← 0b001
CR[4*crfD:4*crfD+3] ← c || XER[SO]
    
```

The contents of **rA** are compared with the sign-extended value of the SIMM field, treating the operands as signed integers. The result of the comparison is placed into CR Field **crfD**.

The L operand controls whether **rA** and **rB** are treated as 32-bit operands (L=0) or 64-bit operands (L=1). For 32-bit PowerPC implementations such as the RCPU, if L=1, the instruction form is invalid.

Other registers altered:

- Condition Register (CR Field specified by operand **crfD**):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.

**Table 9-12 Simplified Mnemonics for cmpi Instruction**

Operation	Simplified Mnemonic	Equivalent To
Compare word immediate	<b>cmpwi crf,rA,value</b> <b>cmpi crfD, rA,value</b>	<b>cmpi crfD, 0, rA,value</b>
Compare word immediate, place result in CR0	<b>cmpwi rA,value</b> <b>cmpi rA,value</b>	<b>cmpi 0, 0, rA,value</b>

# cmpl

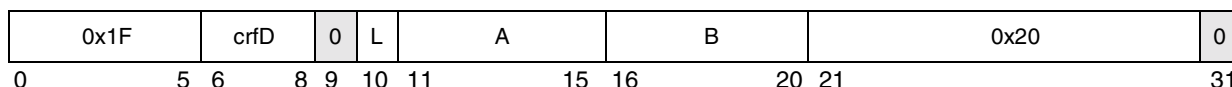
Compare Logical

# cmpl

Integer Unit

**cmpl**            **crfD,L,rA,rB**

Reserved



```

a ← (rA)
b ← (rB)
if a < U b then
    c ← 0b100
else
    if a >U b then
        c ← 0b010
    else
        c ← 0b001
CR[4*crfD:4*crfD+3] ← c || XER[SO]
    
```

The contents of **rA** are compared with the contents of **rB**, treating the operands as unsigned integers. The result of the comparison is placed into CR Field **crfD**.

The **L** operand controls whether **rA** and **rB** are treated as 32-bit operands (**L**=0) or 64-bit operands (**L**=1). For 32-bit PowerPC implementations such as the RCPu, if **L**=1, the instruction form is invalid.

Other registers altered:

- Condition Register (CR Field specified by operand **crfD**):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.

**Table 9-13 Simplified Mnemonics for cmpl Instruction**

Operation	Simplified Mnemonic	Equivalent To
Compare word logical	<b>cmplw crfD, rA,rB</b> <b>cmpl crfD, rA,rB</b>	<b>cmpl crfD, 0, rA,rB</b>
Compare word logical, place result in CR0	<b>cmplw rA,rB</b> <b>cmpl rA,rB</b>	<b>cmpl 0, 0, rA,rB</b>

# cmpli

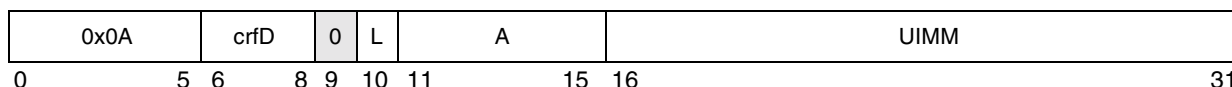
Compare Logical Immediate

# cmpli

Integer Unit

**cmpli**      **crfD,L,rA,UIMM**

Reserved



```

a ← (rA)
b ← (rB)
if a <U (0x0000 || UIMM) then
    c ← 0b100
else
    if a >U (0x0000 || UIMM) then
        c ← 0b010
    else
        c ← 0b001
CR[4*crfD:4*crfD+3] ← c || XER[SO]
    
```

The contents of **rA** are compared with 0x0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into CR Field **crfD**.

The L operand controls whether **rA** and **rB** are treated as 32-bit operands (L=0) or 64-bit operands (L=1). For 32-bit PowerPC implementations such as the RCPU, if L=1, the instruction form is invalid.

Other registers altered:

- Condition Register (CR Field specified by operand **crfD**):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.

**Table 9-14 Simplified Mnemonics for cmpli Instruction**

Operation	Simplified Mnemonic	Equivalent To
Compare word logical immediate	<b>cmplwi</b> crfD,rA,value <b>cmpli</b> crfD,rA,value	<b>cmpli</b> crfD,0,rA,value
Compare word logical immediate, place result in CR0	<b>cmplwi</b> rA,value <b>cmpli</b> rA,value	<b>cmpli</b> 0,0,rA,value

# cntlzw<sub>x</sub>

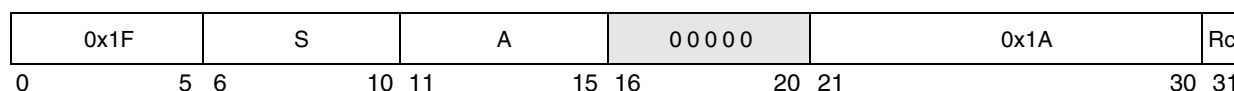
Count Leading Zeros Word

# cntlzw<sub>x</sub>

Integer Unit

**cntlzw**                      rA,rS                      (Rc=0)  
**cntlzw.**                      rA,rS                      (Rc=1)

Reserved



```

n ← 0
do while n < 32
    if rS[n]=1 then leave
    n ← n+1
rA ← n
    
```

A count of the number of consecutive zero bits starting at bit 0 of rS is placed into rA. This number ranges from 0 to 32, inclusive.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO                      (if Rc=1)

For count leading zeros instructions, if Rc=1 then LT is cleared to zero in the CR0 field.

This instruction is defined by the PowerPC UISA.



# crand

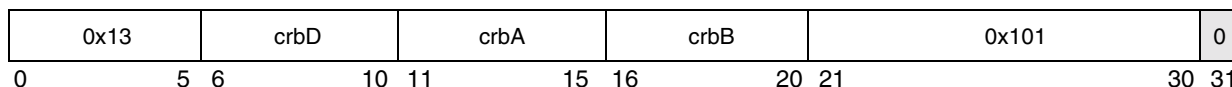
Condition Register AND

# crand

Branch Processor Unit

**crand**     **crbD,crbA,crbB**

Reserved



$$CR[crbD] \leftarrow CR[crbA] \& CR[crbB]$$

The bit in the condition register specified by **crbA** is ANDed with the bit in the condition register specified by **crbB**. The result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by operand **crbD**

This instruction is defined by the PowerPC UISA.

# crandc

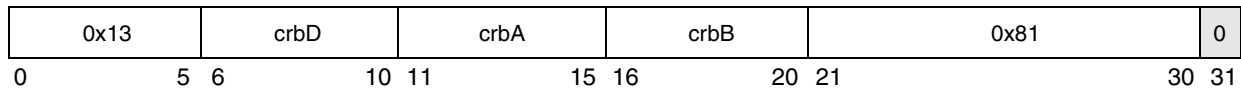
Condition Register AND with Complement

# crandc

Branch Processor Unit

**crandc**    **crbD,crbA,crbB**

 Reserved



$$CR[crbD] \leftarrow CR[crbA] \& \neg CR[crbB]$$

The bit in the condition register specified by **crbA** is ANDed with the complement of the bit in the condition register specified by **crbB** and the result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by operand **crbD**

This instruction is defined by the PowerPC UISA.

# creqv

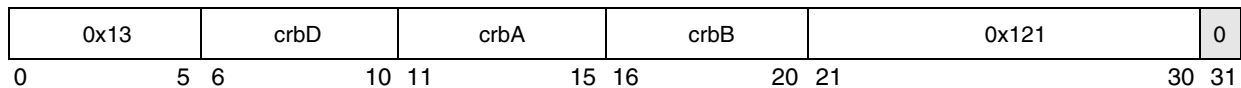
Condition Register Equivalent

# creqv

Branch Processor Unit

**creqv      crbD,crbA,crbB**

 Reserved



$$CR[crbD] \leftarrow CR[crbA] \oplus CR[crbB]$$

The bit in the condition register specified by **crbA** is XORed with the bit in the condition register specified by **crbB** and the complemented result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by operand **crbD**

This instruction is defined by the PowerPC UISA.

**Table 9-15 Simplified Mnemonics for creqv Instruction**

Operation	Simplified Mnemonic	Equivalent To
Condition register set	<b>crset crbD</b>	<b>creqv crbD,crbD,crbD</b>

# crnand

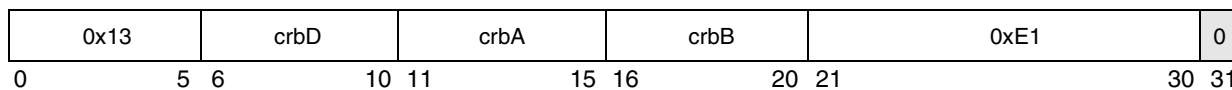
Condition Register NAND

# crnand

Branch Processor Unit

**crnand**    **crbD,crbA,crbB**

Reserved



$$CR[crbD] \leftarrow \neg (CR[crbA] \& CR[crbB])$$

The bit in the condition register specified by **crbA** is ANDed with the bit in the condition register specified by **crbB** and the complemented result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by operand **crbD**

This instruction is defined by the PowerPC UISA.

# crnor

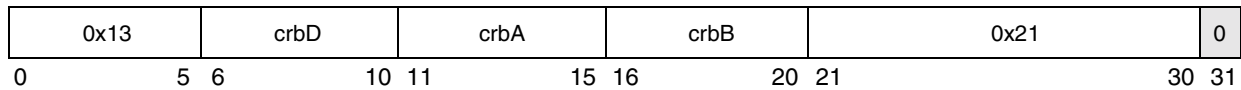
Condition Register NOR

# crnor

Branch Processor Unit

**crnor**      **crbD,crbA,crbB**

Reserved



$$CR[crbD] \leftarrow \neg (CR[crbA] \mid CR[crbB])$$

The bit in the condition register specified by **crbA** is ORed with the bit in the condition register specified by **crbB** and the complemented result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by operand **crbD**

This instruction is defined by the PowerPC UISA.

**Table 9-16 Simplified Mnemonics for crnor Instruction**

Operation	Simplified Mnemonic	Equivalent To
Condition register NOT	<b>crnot crbD, crbA</b>	<b>crnor crbD,crbA,crbA</b>

**cror**

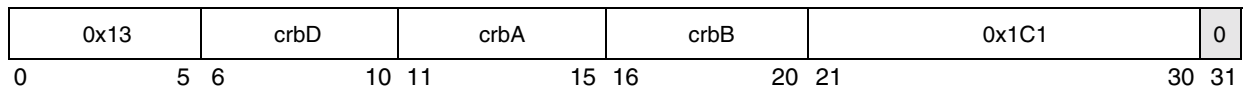
Condition Register OR

**cror**

Branch Processor Unit

**cror**      **crbD,crbA,crbB**

 Reserved



$$CR[crbD] \leftarrow CR[crbA] \mid CR[crbB]$$

The bit in the condition register specified by **crbA** is ORed with the bit in the condition register specified by **crbB**. The result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by operand **crbD**

This instruction is defined by the PowerPC UISA.

**Table 9-17 Simplified Mnemonics for cror Instruction**

Operation	Simplified Mnemonic	Equivalent To
Condition register move	<b>crmove crbD, crbA</b>	<b>cror crbD,crbA,crbA</b>

# crorc

Condition Register OR with Complement

# crorc

Branch Processor Unit

**crorc**      **crbD,crbA,crbB**

Reserved

0x13	crbD	crbA	crbB	0x1A1	0
0	5 6	10 11	15 16	20 21	30 31

$$CR[crbD] \leftarrow CR[crbA] \mid \neg CR[crbB]$$

The bit in the condition register specified by **crbA** is ORed with the complement of the condition register bit specified by **crbB** and the result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by operand **crbD**

This instruction is defined by the PowerPC UISA.

# crxor

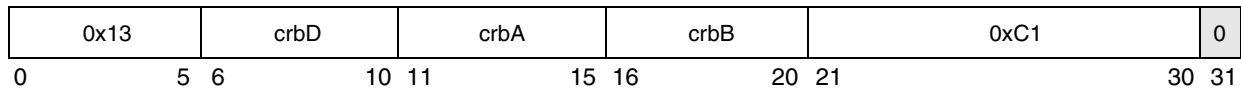
Condition Register XOR

# crxor

Branch Processor Unit

**crxor      crbD,crbA,crbB**

Reserved



$$CR[crbD] \leftarrow CR[crbA] \oplus CR[crbB]$$

The bit in the condition register specified by **crbA** is XORed with the bit in the condition register specified by **crbB** and the result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register:  
Affected: Bit specified by **crbD**

This instruction is defined by the PowerPC UISA.

**Table 9-18 Simplified Mnemonics for crxor Instruction**

Operation	Simplified Mnemonic	Equivalent To
Condition register clear	crclr crbD	crxor crbD,crbD,crbD



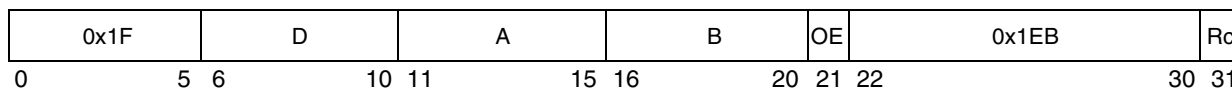
# divwx

Divide Word

# divwx

Integer Unit

<b>divw</b>	<b>rD,rA,rB</b>	(OE=0 Rc=0)
<b>divw.</b>	<b>rD,rA,rB</b>	(OE=0 Rc=1)
<b>divwo</b>	<b>rD,rA,rB</b>	(OE=1 Rc=0)
<b>divwo.</b>	<b>rD,rA,rB</b>	(OE=1 Rc=1)



dividend ← (rA)  
divisor ← (rB)  
rD ← dividend ÷ divisor

Register **rA** is the 32-bit dividend. Register **rB** is the 32-bit divisor. A 32-bit quotient is formed and placed into **rD**. The remainder is not supplied as a result.

Both operands are interpreted as signed integers. The quotient is the unique signed integer that satisfies the following:

$$\text{dividend} = (\text{quotient times divisor}) + r$$

where

$$0 \leq r < |\text{divisor}|$$

if the dividend is non-negative, and

$$-|\text{divisor}| < r \leq 0$$

if the dividend is negative.

If an attempt is made to perform any of the divisions

$$0x8000\ 0000 / -1$$

$$<\text{anything}> / 0$$

then the following conditons result:

- The contents of **rD** are undefined.
- If **Rc** = 1, the contents of the LT, GT, and EQ bits of the CR0 field are undefined.
- If **OE** = 1, then **OV** is set to 1.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if **Rc**=1)

- XER:

Affected: SO, OV

(if OE=1)

The 32-bit signed remainder of dividing **rA** by **rB** can be computed as follows, except in the case that **rA**= 0x8000 0000 and **rB**=-1:

**divw**      **rD,rA,rB**      # **rD**=quotient

**mull**      **rD,rD,rB**      # **rD**=quotient\*divisor

**subf**      **rD,rD,rA**      # **rD**=remainder

This instruction is defined by the PowerPC UISA.

## Integer Unit

MOTOROLA  
9-45

# eieio

Enforce In-Order Execution of I/O

# eieio

Load/Store Unit

Reserved

0x1F	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0x356	0
0	5 6	10 11	15 16	20 21	30 31

The **eieio** instruction provides an ordering function for the effects of load and store instructions executed by a given processor. Executing an **eieio** instruction ensures that all memory accesses previously initiated by the given processor are complete with respect to main memory before any memory accesses subsequently initiated by the given processor access main memory.

The **eieio** instruction orders loads from cache-inhibited memory.

Other registers altered:

- None

The **eieio** instruction is intended for use only in performing memory-mapped I/O operations and to prevent load/store combining operations in main memory. It can be thought of as placing a barrier into the stream of memory accesses issued by a processor, such that any given memory access appears to be on the same side of the barrier to both the processor and the I/O device.

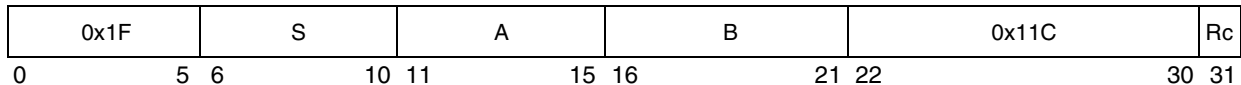
The **eieio** instruction may complete before previously initiated memory accesses have been performed with respect to other processors and mechanisms.

This instruction is defined by the PowerPC VEA.

**eqvX**  
Equivalent

**eqvX**  
Integer Unit

**eqv**                      **rA,rS,rB**                      (Rc=0)  
**eqv.**                      **rA,rS,rB**                      (Rc=1)



$rA \leftarrow ((rS) \equiv (rB))$

The contents of **rS** are XORed with the contents of **rB** and the complemented result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

# extsbx

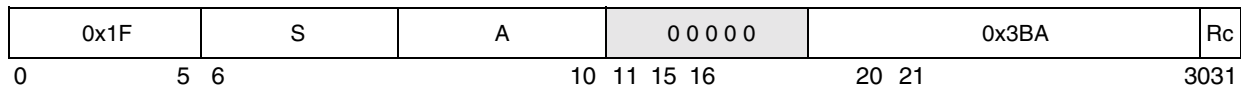
Extend Sign Byte

# extsbx

Integer Unit

**extsb**                      **rA,rS**                      (Rc=0)  
**extsb.**                      **rA,rS**                      (Rc=1)

 Reserved



$S \leftarrow rS[24]$   
 $rA[24:31] \leftarrow rS[24:31]$   
 $rA[0:23] \leftarrow (24)S$

The contents of **rS[24:31]** are placed into **rA[24:31]**. Bit 24 of **rS** is placed into **rA[0:23]**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

# extshx

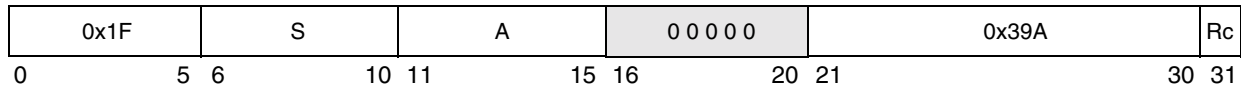
Extend Sign Half Word

# extshx

Integer Unit

**extsh**                      **rA,rS**                      (Rc=0)  
**extsh.**                      **rA,rS**                      (Rc=1)

 Reserved



$S \leftarrow rS[16]$   
 $rA[16:31] \leftarrow rS[16:31]$   
 $rA[0:15] \leftarrow (16)S$

The contents of **rS[16:31]** are placed into **rA[16:31]**. Bit 16 of **rS** is placed into **rA[0:15]**.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

# fabsx

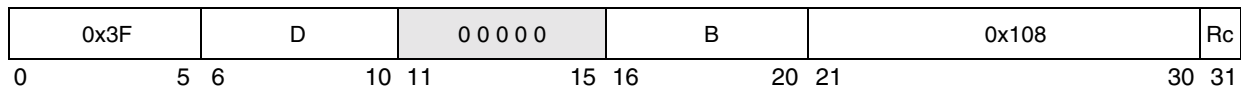
Floating-Point Absolute Value

# fabsx

Floating-Point Unit

**fabs**                      **frD,frB**                      (Rc=0)  
**fabs.**                      **frD,frB**                      (Rc=1)

Reserved



The contents of **frB** with bit 0 cleared to zero are placed into **frD**.

Other registers altered:

- Condition Register (CR1 Field):  
     Affected: FX, FEX, VX, OX                      (if Rc=1)

This instruction is defined by the PowerPC UISA.



# faddx

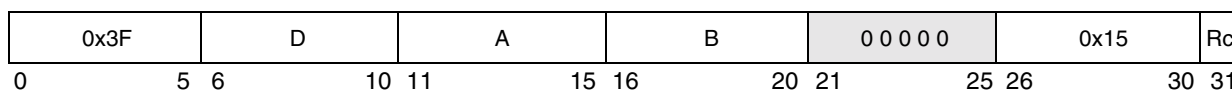
Floating-Point Add

# faddx

Floating-Point Unit

**fadd**                      **frD,frA,frB**                      (Rc=0)  
**fadd.**                      **frD,frA,frB**                      (Rc=1)

Reserved



The floating-point operand in **frA** is added to the floating-point operand in **frB**. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

This instruction is defined by the PowerPC UISA.

# faddsx

Floating-Point Add (Single-Precision)

# faddsx

Floating-Point Unit

**fadds**                      **frD,frA,frB**                      (Rc=0)

**fadds.**                      **frD,frA,frB**                      (Rc=1)

Reserved

0x3B	D	A	B	0 0 0 0 0	0x15	Rc
0                      5   6	10 11	15 16	20 21	25 26	30 31	

The floating-point operand in **frA** is added to the floating-point operand in **frB**. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands, to form an intermediate sum. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

This instruction is defined by the PowerPC UISA.

**fcmpo**

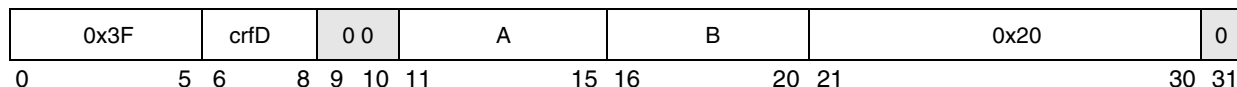
Floating-Point Compare Ordered

**fcmpo**

Floating-Point Unit

**fcmpo****crfD,frA,frB**

Reserved



```

if (frA) is a NaN or (frB) is a NaN
    then c ← 0b001
else if (frA) < (frB) then c ← 0b1000
else if (frA) > (frB) then c ← 0b0100
else c ← 0b0010
FPSCR[FPCC] ← c
CR[4*crfD: 4*crfD+3] ← c
if (frA) is an SNaN or (frB) is an SNaN
    then FPSCR[VXSNAN] ← 1
    if VE=0 then FPSCR[VXVC] ← 1
else if (frA) is a QNaN or (frB) is a QNaN
    then FPSCR[VXVC] ← 1

```

The floating-point operand in **frA** is compared to the floating-point operand in **frB**. The result of the compare is placed into CR Field **crfD** and FPSCR[FPCC].

If at least one of the operands is a NaN, either quiet or signaling, then CR Field **crfD** and FPSCR[FPCC] are set to reflect unordered. If at least one of the operands is a signaling NaN, then FPSCR[VXSNAN] is set, and if invalid operation is disabled (FPSCR[VE]=0) then FPSCR[VXVC] is set. If neither operand is a signaling NaN, but at least one is a QNaN, then FPSCR[VXVC] is set.

Other registers altered:

- Condition Register (CR Field specified by operand **crfD**):  
Affected: FX, FEX, VX, OX
- Floating-Point Status and Control Register:  
Affected: FPCC, FX, VXSNAN, VXVC

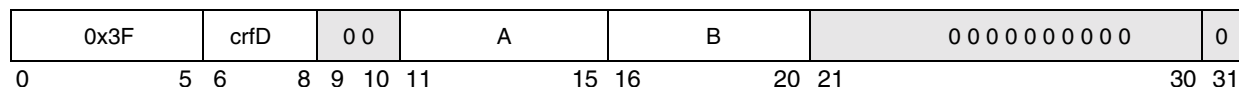
This instruction is defined by the PowerPC UISA.

**fcmpu**

Floating-Point Compare Unordered

**fcmpu**

Floating-Point Unit

**fcmpu****crfD,frA,frB** Reserved


```

if (frA) is a NaN or (frB) is a NaN
    then c ← 0b001
else if (frA) < (frB) then c ← 0b1000
else if (frA) > (frB) then c ← 0b0100
else c ← 0b0010
FPSCR[FPCC] ← c
CR[4*crfD: 4*crfD+3] ← c
if (frA) is an SNaN or (frB) is an SNaN
    then FPSCR[VXSNAN] ← 1

```

The floating-point operand in register **frA** is compared to the floating-point operand in register **frB**. The result of the compare is placed into CR Field **crfD** and into FPSCR[FPCC].

If at least one of the operands is a NaN, either quiet or signaling, then CR Field **crfD** and FPSCR[FPCC] are set to reflect unordered. If at least one of the operands is a signaling NaN, then FPSCR[VXSNAN] is set.

Other registers altered:

- Condition Register (CR Field specified by operand **crfD**):  
Affected: FX, FEX, VX, OX
- Floating-Point Status and Control Register:  
Affected: FPCC, FX, VXSNAN

This instruction is defined by the PowerPC UISA.

# fctiwX

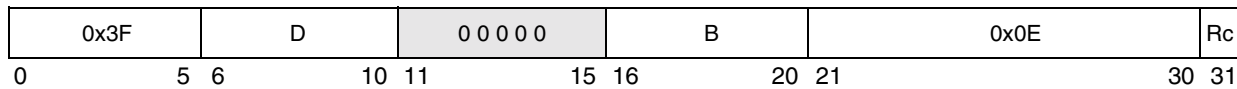
Floating-Point Convert to Integer Word

# fctiwX

Floating-Point Unit

**fctiw**                      **frD,frB**                      (Rc=0)  
**fctiw.**                      **frD,frB**                      (Rc=1)

Reserved



The floating-point operand in register **frB** is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in of **frD**[32:63]. **frD**[0:31] are undefined.

If the contents of **frB** is greater than  $2^{31}-1$ , **frD**[32:63] are set to 0x7FFF FFFF.

If the contents of **frB** is less than  $-2^{31}$ , **frD**[32:63] are set to 0x8000 0000.

The conversion is described fully in [APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS](#).

Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Other registers altered:

- Condition Register (CR1 Field):  
     Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
     Affected: FPRF (undefined), FR, FI, FX, XX, VXSNaN, VXCvI

This instruction is defined by the PowerPC UISA.

# fctiwzx

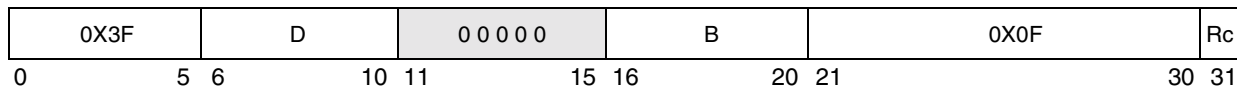
Floating-Point Convert to Integer Word with Round toward Zero

# fctiwzx

Floating-Point Unit

**fctiwz**                      **frD,frB**                      (Rc=0)  
**fctiwz.**                      **frD,frB**                      (Rc=1)

Reserved



The floating-point operand in register **frB** is converted to a 32-bit signed integer, using the rounding mode round toward zero, and placed in bits 32:63 of **frD**. **frD**[0:31] are undefined.

If the operand in **frB** is greater than  $2^{31}-1$ , **frD**[32:63] are set to 0x7FFF FFFF.

If the operand in **frB** is less than  $-2^{31}$ , **frD**[32:63] are set to 0x8000 0000.

The conversion is described fully in [APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS](#).

Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Other registers altered:

- Condition Register (CR1 Field):  
     Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
     Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

This instruction is defined by the PowerPC UISA.

# fdivx

Floating-Point Divide

# fdivx

Floating-Point Unit

**fdiv**                      **frD,frA,frB**                      (Rc=0)  
**fdiv.**                      **frD,frA,frB**                      (Rc=1)

Reserved

0x3F	D	A	B	0 0 0 0 0	0x12	Rc
0                      5   6	10 11	15 16	20 21	25 26	30 31	

The floating-point operand in register **frA** is divided by the floating-point operand in register **frB**. No remainder is preserved.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1 and zero divide exceptions when FPSCR[ZE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNaN, VXIDI, VXZDZ

This instruction is defined by the PowerPC UISA.

# fdivsx

Floating-Point Divide Single-Precision

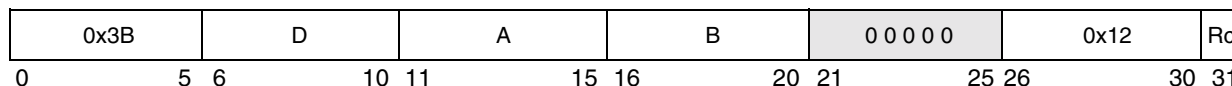
# fdivsx

Floating-Point Unit

**fdivs**                      **frD,frA,frB**                      (Rc=0)

**fdivs.**                      **frD,frA,frB**                      (Rc=1)

Reserved



The floating-point operand in register **frA** is divided by the floating-point operand in register **frB**. No remainder is preserved.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1 and zero divide exceptions when FPSCR[ZE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNaN, VXIDI, VXZDZ

This instruction is defined by the PowerPC UISA.



# fmaddx

Floating-Point Multiply-Add

# fmaddx

Floating-Point Unit

**fmadd**      **frD,frA,frC,frB**      (Rc=0)

**fmadd.**      **frD,frA,frC,frB**      (Rc=1)

0x3F	D	A	B	C	0x1D	Rc
0 5 6	10 11	15 16	20 21	25 26	30 31	

The following operation is performed:

$$\text{frD} \leftarrow [(\text{frA}) * (\text{frC})] + (\text{frB})$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

# fmaddsx

Floating-Point Multiply-Add Single-Precision

# fmaddsx

Floating-Point Unit

**fmadds**     **frD,frA,frC,frB**                      (Rc=0)

**fmadds.**     **frD,frA,frC,frB**                      (Rc=1)

0x3B	D	A	B	C	0x1D	Rc
0 5 6	10 11	15 16	20 21	25 26	30 31	

The following operation is performed:

$$\mathbf{frD} \leftarrow [(\mathbf{frA}) * (\mathbf{frC})] + (\mathbf{frB})$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

fmr<sub>x</sub>

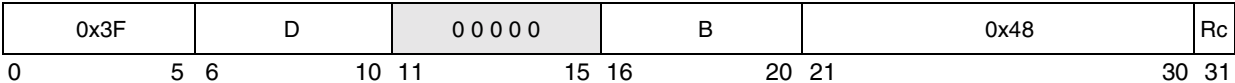
Floating-Point Move Register

fmr                      frD,frB                      (Rc=0)  
fmr.                    frD,frB                      (Rc=1)

fmr<sub>x</sub>

Floating-Point Unit

 Reserved



The contents of register **frB** are placed into **frD**.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

# fmsubx

Floating-Point Multiply-Subtract

# fmsubx

Floating-Point Unit

**fmsub**      **frD,frA,frC,frB**      (Rc=0)

**fmsub.**     **frD,frA,frC,frB**      (Rc=1)

0x3F	D	A	B	C	0x1C	Rc
0 5 6	10 11	15 16	20 21	25 26	30 31	

The following operation is performed:

$$\text{frD} \leftarrow [(\text{frA}) * (\text{frC})] - (\text{frB})$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

**fmsubsx**

Floating-Point Multiply-Subtract (Single-Precision)

**fmsubsx**

Floating-Point Unit

**fmsubs**     **frD,frA,frC,frB**                      (Rc=0)**fmsubs.**     **frD,frA,frC,frB**                      (Rc=1)

0x3B	D	A	B	C	0x1C	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

The following operation is performed:

$$\mathbf{frD} \leftarrow [(\mathbf{frA}) * (\mathbf{frC})] - (\mathbf{frB})$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

# fmulx

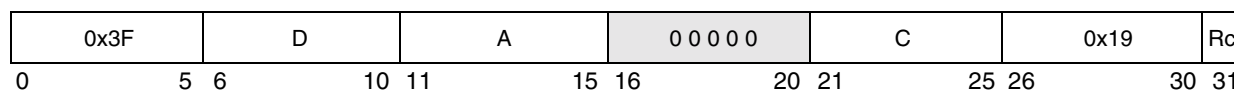
Floating-Point Multiply

# fmulx

Floating-Point Unit

**fmul**                      **frD,frA,frC**                      (Rc=0)  
**fmul.**                      **frD,frA,frC**                      (Rc=1)

Reserved



The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
     Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
     Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXIMZ

This instruction is defined by the PowerPC UISA.

# fmulsx

Floating-Point Multiply Single-Precision

# fmulsx

Floating-Point Unit

**fmuls**                      **frD,frA,frC**                      (Rc=0)  
**fmuls.**                      **frD,frA,frC**                      (Rc=1)

Reserved

0x3B	D	A	0 0 0 0 0	C	0x19	Rc
0                      5   6	10 11	15 16	20 21	25 26	30 31	

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**.

If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
     Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
     Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXIMZ

This instruction is defined by the PowerPC UISA.

# fnabsx

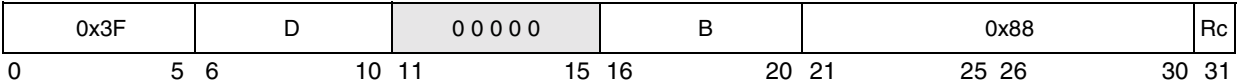
Floating-Point Negative Absolute Value

# fnabsx

Floating-Point Unit

fnabs frD,frB (Rc=0)  
 fnabs. frD,frB (Rc=1)

 Reserved



The contents of register frB, with bit 0 set to one, are placed into frD.

Other registers altered:

- Condition Register (CR1 Field):  
 Affected: FX, FEX, VX, OX (if Rc=1)

This instruction is defined by the PowerPC UISA.



fnegx

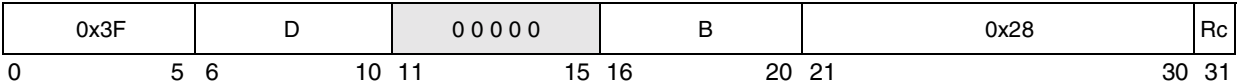
Floating-Point Negate

fnegx

Floating-Point Unit

fneg frD,frB (Rc=0)  
fneg. frD,frB (Rc=1)

Reserved



The contents of register frB, with bit 0 inverted, are placed into frD.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX (if Rc=1)

This instruction is defined by the PowerPC UISA.

**fnmaddx**

Floating-Point Negative Multiply-Add

**fnmaddx**

Floating-Point Unit

**fnmadd**     **frD,frA,frC,frB**                    (Rc=0)**fnmadd.**    **frD,frA,frC,frB**                    (Rc=1)

0x3F	D	A	B	C	0x1F	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

The following operation is performed:

$$\text{frD} \leftarrow -([\text{frA}] * [\text{frC}]) + [\text{frB}]$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result. If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                    (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

**fnmaddsx**

Floating-Point Negative Multiply-Add Single-Precision

**fnmaddsx**

Floating-Point Unit

**fnmadds** **frD,frA,frC,frB** (Rc=0)**fnmadds.** **frD,frA,frC,frB** (Rc=1)

0x3B	D	A	B	C	0x1F	Rc
0 5 6	10 11	15 16	20 21	25 26	30 31	

The following operation is performed:

$$\text{frD} \leftarrow -([\text{frA}] * [\text{frC}]) + [\text{frB}]$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result. If an operand is a denormalized number then it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result as would be obtained by using the floating-point multiply-add instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

# fnmsubx

Floating-Point Negative Multiply-Subtract

# fnmsubx

Floating-Point Unit

**fnmsub**     **frD,frA,frC,frB**                      (Rc=0)

**fnmsub.**    **frD,frA,frC,frB**                      (Rc=1)

0x3F						D					A					B					C					0x1E					Rc	
0						5	6				10	11				15	16				20	21				25	26				30	31

The following operation is performed:

$$\text{frD} \leftarrow -([\text{frA} * (\text{frC})] - (\text{frB}))$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If an operand is a denormalized number, it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result obtained by negating the result of a floating multiply-subtract instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field)  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

# fnmsubsx

Floating-Point Negative Multiply-Subtract Single-Precision

# fnmsubsx

Floating-Point Unit

**fnmsubs**    **frD,frA,frC,frB**                      (Rc=0)

**fnmsubs.**   **frD,frA,frC,frB**                      (Rc=1)

0x3B		D		A		B		C		0x1E		Rc													
0		5		6		10		11		15		16		20		21		25		26		30		31	

The following operation is performed:

$$\text{frD} \leftarrow -([\text{frA} * (\text{frC})] - (\text{frB}))$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If an operand is a denormalized number, it is prenormalized before the operation is started. If the most significant bit of the resultant significand is not one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result obtained by negating the result of a floating multiply-subtract instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field)  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

This instruction is defined by the PowerPC UISA.

# frsp<sub>x</sub>

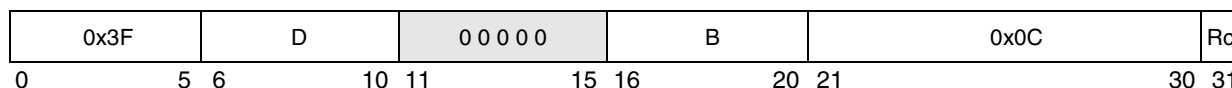
Floating-Point Round to Single-Precision

# frsp<sub>x</sub>

Floating-Point Unit

**frsp**                      **frD,frB**                      (Rc=0)  
**frsp.**                      **frD,frB**                      (Rc=1)

Reserved



If it is already in single-precision range, the floating-point operand in register **frB** is placed into **frD**. Otherwise the floating-point operand in register **frB** is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into **frD**.

The rounding is described fully in [APPENDIX C FLOATING-POINT MODELS AND CONVERSIONS](#).

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
     Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
     Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN

This instruction is defined by the PowerPC UISA.

# fsubx

Floating-Point Subtract

# fsubx

Floating-Point Unit

**fsub**                      **frD,frA,frB**                      (Rc=0)  
**fsub.**                      **frD,frA,frB**                      (Rc=1)

☐ Reserved

0x3F	D	A	B	0 0 0 0 0	0x14	Rc
0                      5   6	10   11	15   16	20   21	25   26	30   31	

The floating-point operand in register **frB** is subtracted from the floating-point operand in register **frA**. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

The execution of the floating-point subtract instruction is identical to that of floating-point add, except that the contents of **frB** participates in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

This instruction is defined by the PowerPC UISA.

# fsubsx

Floating-Point Subtract Single-Precision

# fsubsx

Floating-Point Unit

**fsubs**                      **frD,frA,frB**                      (Rc=0)

**fsubs.**                      **frD,frA,frB**                      (Rc=1)

☐ Reserved

0x3B	D	A	B	0 0 0 0 0	0x14	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

The floating-point operand in register **frB** is subtracted from the floating-point operand in register **frA**. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

The execution of the floating-point subtract instruction is identical to that of floating-point add, except that the contents of **frB** participates in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE]=1.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

This instruction is defined by the PowerPC UISA.



# icbi

Instruction Cache Block Invalidate

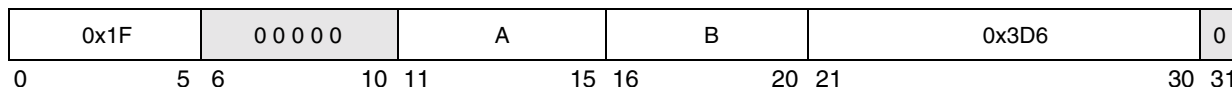
# icbi

Load/Store Unit

**icbi**

**rA,rB**

☐ Reserved



EA is the sum (rA|0)+(rB).

If a block containing the byte addressed by EA is in the instruction cache of this processor, the block is made invalid in the processor. Subsequent references cause the block to be refetched.

## NOTE

According to the PowerPC architecture, if the addressed block is in coherency-required mode, the block is made invalid in all affected processors. In the RCPU, however, all instruction memory is considered to be in coherency-not-required mode.

Other registers altered:

- None

This instruction is defined by the PowerPC VEA.

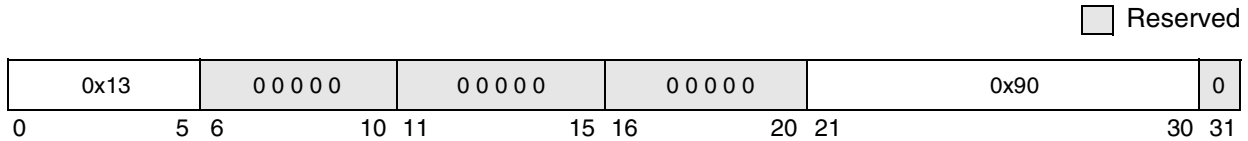
# isync

Instruction Synchronize

# isync

Branch Processor Unit

## isync



Fetch of an **isync** instruction causes fetch serialization: instruction fetch is halted until all instructions currently in the processor (i.e., all issued instructions as well as the pre-fetched instructions waiting to be issued) have completed execution. This instruction causes subsequent instructions to execute in the context established by the previous instructions.

This instruction has no effect on other processors or on their caches.

Other registers altered:

- None

This instruction is defined by the PowerPC VEA.

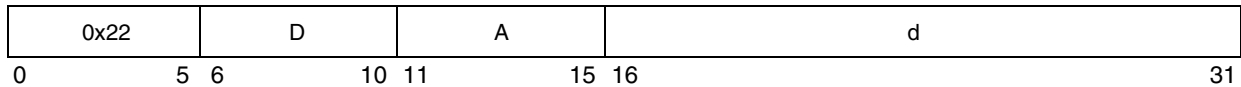
# lbz

Load Byte and Zero

# lbz

Load/Store Unit

**lbz**                      **rD,d(rA)**



```

if rA=0 then b ← 0
else b ← (rA)
EA ← b+EXTS(d)
rD ← (24)0 || MEM(EA, 1)
    
```

The effective address is the sum (rA|0) + d. The byte in memory addressed by EA is loaded into rD[24:31]. Bits rD[0:23] are cleared to zero.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

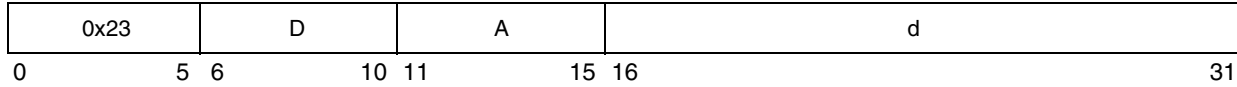
# lbzu

Load Byte and Zero with Update

# lbzu

Load/Store Unit

**lbzu**                      **rD,d(rA)**



$EA \leftarrow (rA) + EXT(S(d))$   
 $rD \leftarrow (24)0 \parallel MEM(EA, 1)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + d$ . The byte in memory addressed by EA is loaded into  $rD[24:31]$ . Bits  $rD[0:23]$  are cleared to zero.

EA is placed into  $rA$ .

If  $rA=0$  or  $rA=rD$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lbzux

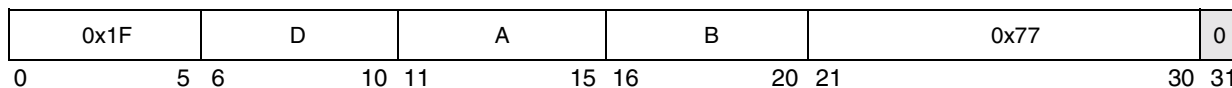
Load Byte and Zero with Update Indexed

# lbzux

Load/Store Unit

**lbzux**                      **rD,rA,rB**

Reserved



$EA \leftarrow (rA) + (rB)$   
 $rD \leftarrow (24)0 \parallel \text{MEM}(EA, 1)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + (rB)$ . The byte addressed by EA is loaded into  $rD[24:31]$ . Bits  $rD[0:23]$  are cleared to zero.

EA is placed into  $rA$ .

If  $rA=0$  or  $rA=rD$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

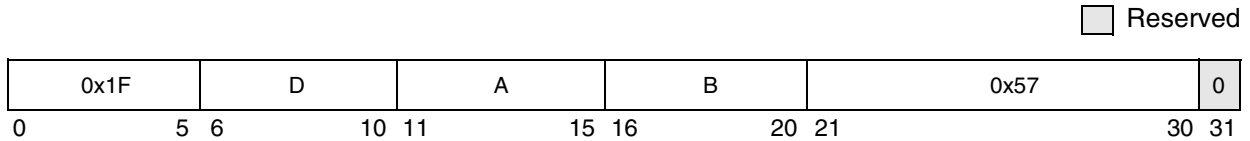
# lbzx

Load Byte and Zero Indexed

# lbzx

Load/Store Unit

**lbzx**                      rD,rA,rB



```

if rA=0 then b ← 0
else b ← (rA)
EA ← b+(rB)
rD ← (24)0 || MEM(EA, 1)

```

EA is the sum (rA|0) + (rB). The byte in memory addressed by EA is loaded into rD[24:31].

Bits rD[0:23] are cleared to zero.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

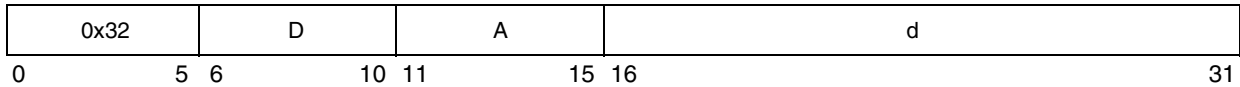
# lfd

Load Floating-Point Double-Precision

# lfd

Load/Store Unit

**lfd**                      **frD,d(rA)**



```

if rA=0 then b ← 0
else b ← (rA)
EA ← b+EXTS(d)
frD ← MEM(EA, 8)
    
```

EA is the sum (rA|0) + d.

The double word in memory addressed by EA is placed into **frD**.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

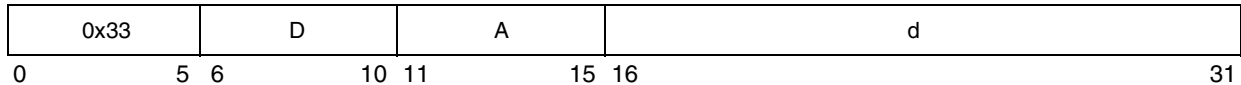
# lfd

Load Floating-Point Double-Precision with Update

# lfd

Load/Store Unit

**lfd**                      **frD,d(rA)**



EA ← (rA)+EXTS(d)  
frD ← MEM(EA, 8)  
rA ← EA

EA is the sum (rA|0) + d.

The double word in memory addressed by EA is placed into **frD**.

EA is placed into rA.

If rA=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.



# lfdux

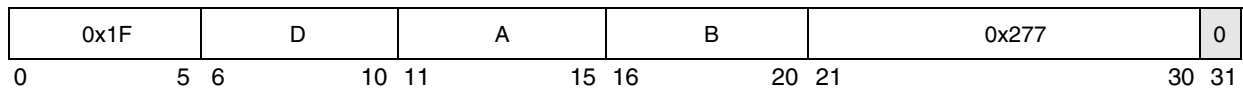
Load Floating-Point Double-Precision with Update Indexed

# lfdux

Load/Store Unit

**lfdux**                      **frD,rA,rB**

Reserved



$EA \leftarrow (rA) + (rB)$   
 $frD \leftarrow MEM(EA, 8)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + (rB)$ .

The double word in memory addressed by EA is placed into **frD**.

EA is placed into **rA**.

If **rA**=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

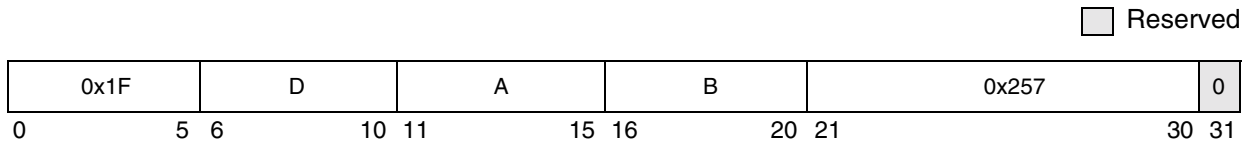
lfdx

Load Floating-Point Double-Precision Indexed

lfdx

Load/Store Unit

lfdx                      frD,rA,rB



if rA=0 then b ← 0

else b ← (rA)

EA ← b+(rB)

frD ← MEM(EA, 8)

EA is the sum (rA|0) + (rB).

The double word in memory addressed by EA is placed into frD.

Other registers altered:

• None

This instruction is defined by the PowerPC UISA.

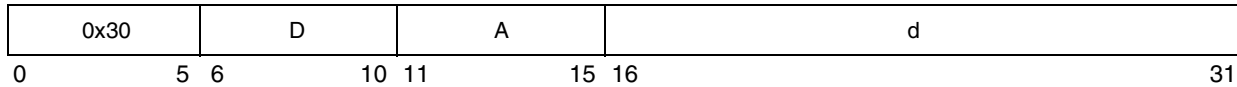
# lfs

Load Floating-Point Single-Precision

# lfs

Integer Unit

**lfs**                      **frD,d(rA)**



```

if rA=0 then b ← 0
else b ← (rA)
EA ← b+EXTS(d)
frD ← DOUBLE(MEM(EA, 4))
    
```

EA is the sum (rA|0) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see [4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions](#)) and placed into **frD**.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

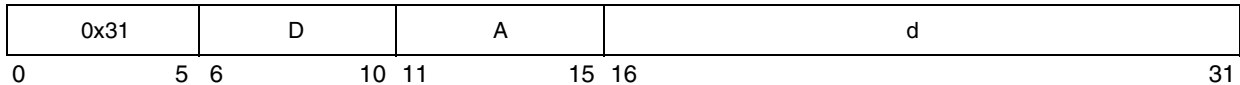
# lfsu

Load Floating-Point Single-Precision with Update

# lfsu

Integer Unit

**lfsu**                      **frD,d(rA)**



EA ← (rA)+EXTS(d)  
frD ← DOUBLE(MEM(EA, 4))  
rA ← EA

EA is the sum (rA|0) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see [4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions](#)) and placed into **frD**.

EA is placed into **rA**.

If **rA**=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lfsux

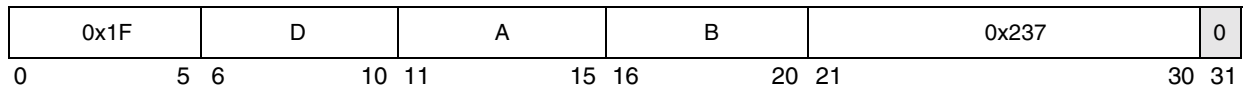
Load Floating-Point Single-Precision with Update Indexed

# lfsux

Load/Store Unit

**lfsux**                      **frD,rA,rB**

Reserved



$EA \leftarrow (rA)+(rB)$   
 $frD \leftarrow \text{DOUBLE}(\text{MEM}(EA, 4))$   
 $rA \leftarrow EA$

EA is the sum (rA|0) + (rB).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see [4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions](#)) and placed into frD.

EA is placed into rA.

If rA=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lfsx

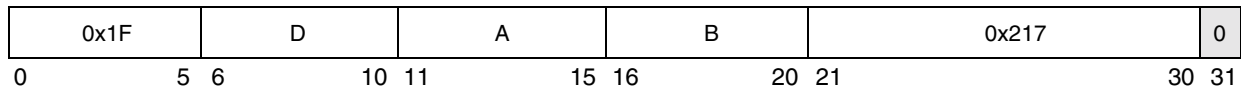
Load Floating-Point Single-Precision Indexed

# lfsx

Load/Store Unit

**lfsx**                      **frD,rA,rB**

Reserved



```

if rA=0 then b ← 0
else b ← (rA)
EA ← b+(rB)
frD ← DOUBLE(MEM(EA, 4))
    
```

EA is the sum (rA|0) + (rB).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see [4.5.8.1 Double-Precision Conversion for Floating-Point Load Instructions](#)) and placed into frD.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

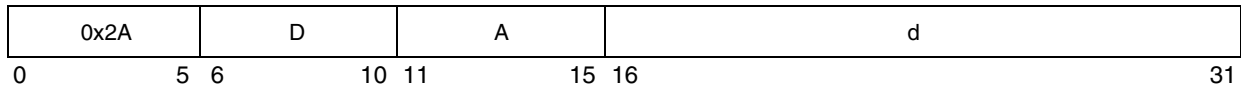
# lha

Load Half Word Algebraic

# lha

Load/Store Unit

**lha**                      **rD,d(rA)**



```

if rA=0 then b ← 0
else b ← (rA)
EA ← b+EXTS(d)
rD ← EXTS(MEM(EA, 2))

```

EA is the sum (rA[0] + d. The half word in memory addressed by EA is loaded into rD[16:31]. Bits rD[0:15] are filled with a copy of bit 0 of the loaded half word.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

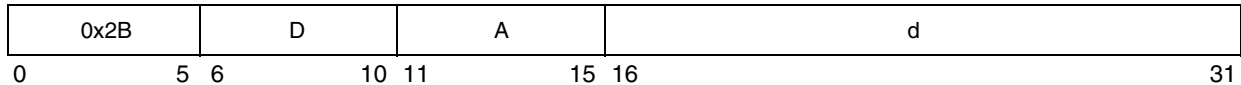
# lhau

Load Half Word Algebraic with Update

# lhau

Load/Store Unit

**lhau**                      **rD,d(rA)**



$EA \leftarrow (rA) + EXTS(d)$   
 $rD \leftarrow EXTS(MEM(EA, 2))$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + d$ . The half word in memory addressed by EA is loaded into  $rD[16:31]$ .

Bits  $rD[0:15]$  are filled with a copy of bit 0 of the loaded half word.

EA is placed into  $rA$ .

If  $rA=0$  or  $rA=rD$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.



# lhaux

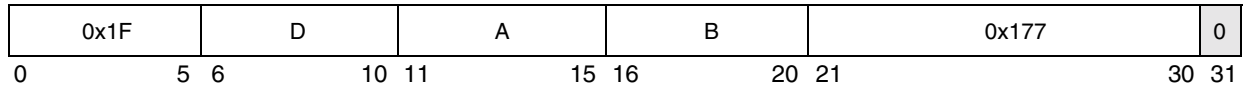
Load Half Word Algebraic with Update Indexed

# lhaux

Load/Store Unit

**lhaux**                      **rD,rA,rB**

Reserved



$EA \leftarrow (rA) + (rB)$   
 $rD \leftarrow \text{EXTS}(\text{MEM}(EA, 2))$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + (rB)$ . The half word in memory addressed by EA is loaded into  $rD[16:31]$ . Bits  $rD[0:15]$  are filled with a copy of bit 0 of the loaded half word.

EA is placed into **rA**.

If  $rA=0$  or  $rA=rD$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

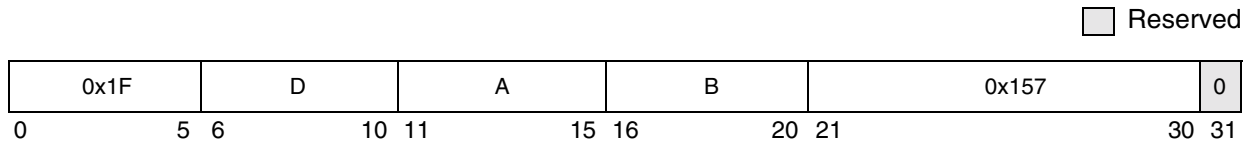
# lhax

Load Half Word Algebraic Indexed

# lhax

Load/Store Unit

lhax                      rD,rA,rB



if rA=0 then b ← 0  
else b ← (rA)  
EA ← b+(rB)  
rD ← EXTS(MEM(EA, 2))

EA is the sum (rA|0) + (rB). The half word in memory addressed by EA is loaded into rD[16:31]. Bits rD[0:15] are filled with a copy of bit 0 of the loaded half word.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lhbrx

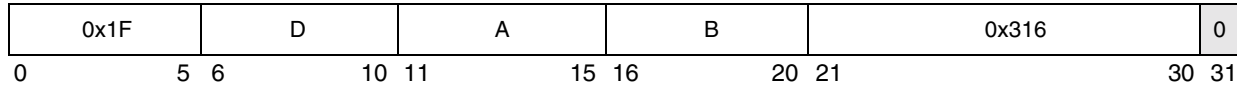
Load Half Word Byte-Reverse Indexed

# lhbrx

Load/Store Unit

**lhbrx**                      **rD,rA,rB**

Reserved



```

if rA=0 then b ← 0
else b ← (rA)
EA ← b+(rB)
rD ← (16)0 || MEM(EA+1, 1) || MEM(EA,1)
    
```

EA is the sum  $(rA|0) + (rB)$ . Bits 0:7 of the half word in memory addressed by EA are loaded into **rD**[24:31]. Bits 8:15 of the half word in memory addressed by EA are loaded into **rD**[16:23]. Bits **rD**[0:15] are cleared to zero.

Some PowerPC implementations may run the **lhbrx** instructions with greater latency than other types of load instructions. This is not the case in the RCPU. This instruction operates with the same latency as other load instructions.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

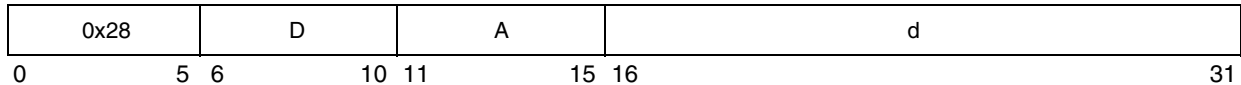
# lhz

Load Half Word and Zero

# lhz

Load/Store Unit

**lhz**                      **rD,d(rA)**



if  $rA=0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + EXTS(d)$   
 $rD \leftarrow (16)0 \parallel MEM(EA, 2)$

EA is the sum  $(rA|0) + d$ . The half word in memory addressed by EA is loaded into  $rD[16:31]$ . Bits  $rD[0:15]$  are cleared to zero.

Other registers altered:

- None

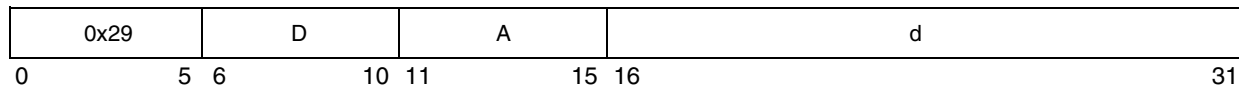
This instruction is defined by the PowerPC UISA.

# lhzu

Load Half Word and Zero with Update

# lhzu

Load/Store Unit

**lhzu****rD,d(rA)**

$EA \leftarrow (rA) + \text{EXTS}(d)$   
 $rD \leftarrow (16)0 \parallel \text{MEM}(EA, 2)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + d$ . The half word in memory addressed by EA is loaded into  $rD[16:31]$ . Bits  $rD[0:15]$  are cleared to zero.

EA is placed into  $rA$ .

If  $rA=0$  or  $rA=rD$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lhzux

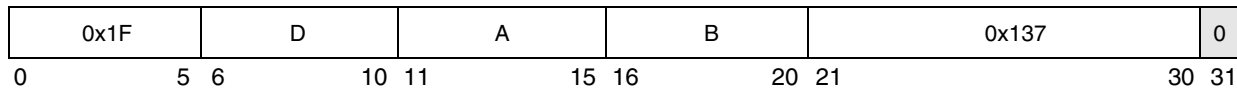
Load Half Word and Zero with Update Indexed

# lhzux

Load/Store Unit

**lhzux**                      **rD,rA,rB**

Reserved



$EA \leftarrow (rA|0) + (rB)$   
 $rD \leftarrow (16)0 \parallel \text{MEM}(EA, 2)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + (rB)$ . The half word in memory addressed by EA is loaded into  $rD[16:31]$ . Bits  $rD[0:15]$  are cleared to zero.

EA is placed into  $rA$ .

If  $rA=0$  or  $rA=rD$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lhzx

Load Half Word and Zero Indexed

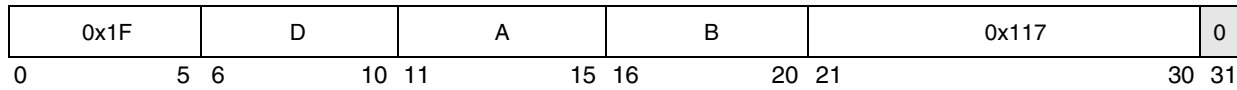
# lhzx

Load/Store Unit

**lhzx**

**rD,rA,rB**

Reserved



```

if rA=0 then b←0
else b←(rA)
EA←b+(rB)
rD←(16)0 || MEM(EA, 2)
    
```

The effective address is the sum ( $rA|0$ ) + ( $rB$ ). The half word in memory addressed by EA is loaded into  $rD[16:31]$ . Bits  $rD[0:15]$  are cleared to zero.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

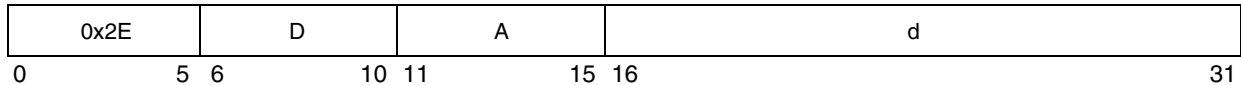
# lmw

Load Multiple Word

# lmw

Load/Store Unit

**lmw**                      **rD,d(rA)**



```

if rA=0 then b←0
else b←(rA)
EA←b+EXTS(d)
r←rD
do while r ≤ 31
    GPR(r)← MEM(EA, 4)
    r←r+1
    EA←EA+4
EA is the sum (rA|0) + d.
n=(32-rD).
n consecutive words starting at EA are loaded into the 32 bits of GPRs rD through r31.
EA must be a multiple of four; otherwise, the system alignment exception handler is in-
voked.
If rA is in the range of registers specified to be loaded, including the case in which rA = 0,
the instruction form is invalid.
Other registers altered:
    • None
This instruction is defined by the PowerPC UISA.
    
```




# lswi

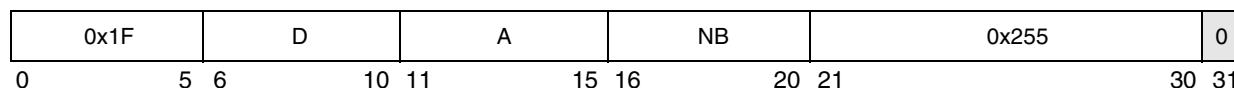
Load String Word Immediate

# lswi

Load/Store Unit

**lswi**                      **rD,rA,NB**

 Reserved



```

if rA=0 then EA←0
else EA←(rA)
if NB=0 then n←32
else n←NB
r←rD - 1
i←32
do while n Š 0
    if i=32 then
        r←r+1 (mod 32)
        GPR(r)←0
        GPR(r)[i:i+7]←MEM(EA, 1)
        i←i+8
        EA←EA+1
        n←n-1
    
```

The EA is (rA|0).

Let  $n=NB$  if  $NB \neq 0$ ,  $n=32$  if  $NB=0$ ;  $n$  is the number of bytes to load. Let  $nr=CEIL(n/4)$ ;  $nr$  is the number of registers to be loaded with data.

$n$  consecutive bytes starting at the EA are loaded into GPRs **rD** through **rD+nr-1**. Bytes are loaded left to right in each register. The sequence of registers wraps around to **r0** if required. If the four bytes of register **rD+nr-1** are only partially filled, the unfilled low-order byte(s) of that register are cleared to zero.

If **rA** is in the range of registers specified to be loaded, including the case in which **rA** = 0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

**lswx**

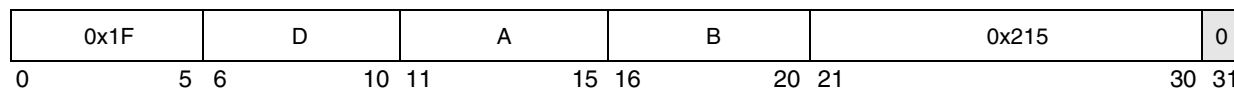
Load String Word Indexed

**lswx**

Load/Store Unit

**lswx****rD,rA,rB**

Reserved



```

if rA=0 then b←0
else b←(rA)
EA←b+(rB)
n←XER[25:31]
r←rD - 1
i←32
do while n Š 0
    if i=32 then
        r←r+1 (mod 32)
        GPR(r)←0
        GPR(r)[i:i+7]←MEM(EA, 1)
        i←i+8
        EA←EA+1
        n←n-1

```

EA is the sum  $(rA|0)+(rB)$ . Let  $n=XER[25:31]$ ;  $n$  is the number of bytes to load. Let  $nr=CEIL(n/4)$ ;  $nr$  is the number of registers to receive data.

If  $n>0$ ,  $n$  consecutive bytes starting at EA are loaded into GPRs rD through  $rD+nr-1$ .

Bytes are loaded left to right in each register. The sequence of registers wraps around to r0 if required. If the bytes of  $rD+nr-1$  are only partially filled, the unfilled low-order byte(s) of that register are cleared to zero.

If  $n=0$ , the content of rD is undefined.

If rA or rB is in the range of registers specified to be loaded, including the case in which  $rA = 0$ , either the system illegal instruction error handler is invoked or the results are boundedly undefined.

If  $rD = rA$  or  $rD = rB$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

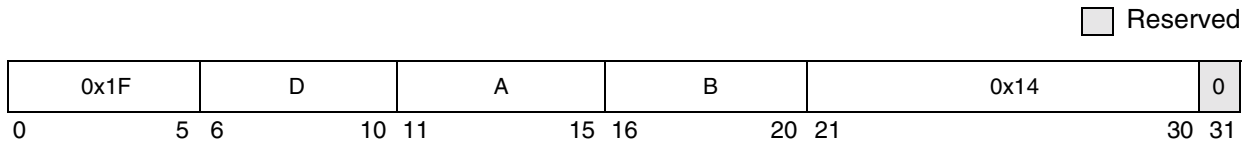
# lwarx

Load Word and Reserve Indexed

# lwarx

Load/Store Unit

**lwarx**                      **rD,rA,rB**



```

if rA=0 then b←0
else b←(rA)
EA←b+(rB)
RESERVE←1
RESERVE_ADDR←func(EA)
rD←MEM(EA,4)
    
```

EA is the sum (rA|0) + (rB). The word in memory addressed by EA is loaded into rD.

This instruction creates a reservation for use by a store word conditional instruction. An address computed from the EA is associated with the reservation, and replaces any address previously associated with the reservation: the manner in which the address to be associated with the reservation is computed from the EA is described in [4.1.2 Addressing Modes and Effective Address Calculation](#).

If the EA is not a multiple of four, the alignment exception handler is invoked.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lwbrx

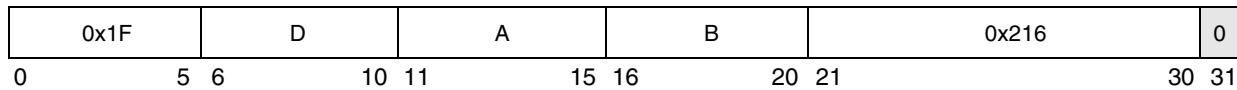
Load Word Byte-Reverse Indexed

# lwbrx

Load/Store Unit

**lwbrx**                      **rD,rA,rB**

Reserved



```

if rA=0 then b←0
else b←(rA)
EA←b+(rB)
rD←MEM(EA+3, 1) || MEM(EA+2, 1) || MEM(EA+1, 1) || MEM(EA, 1)
    
```

EA is the sum (rA|0)+(rB). Bits 0:7 of the word in memory addressed by EA are loaded into rD[24:31]. Bits 8:15 of the word in memory addressed by EA are loaded into rD[16:23]. Bits 16:23 of the word in memory addressed by EA are loaded into rD[8:15]. Bits 24:31 of the word in memory addressed by EA are loaded into rD[0:7].

Some PowerPC implementations may run the **lwbrx** instructions with greater latency than other types of load instructions. This is not the case in the RCPU. This instruction operates with the same latency as other load instructions.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

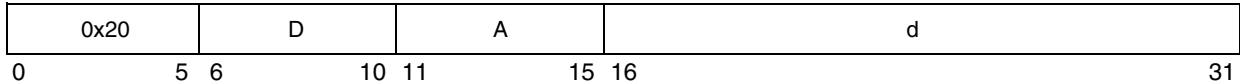
# lwz

Load Word and Zero

# lwz

Load/Store Unit

**lwz**                      **rD,d(rA)**



if  $rA=0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + EXTS(d)$   
 $rD \leftarrow MEM(EA, 4)$

EA is the sum  $(rA|0) + d$ . The word in memory addressed by EA is loaded into rD.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

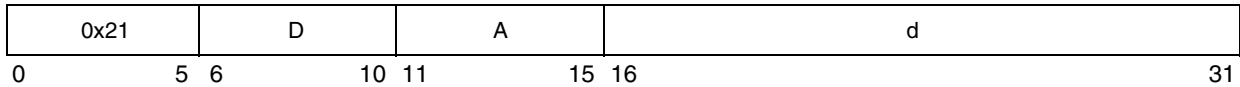
# lwzu

Load Word and Zero with Update

# lwzu

Load/Store Unit

lwzu                      rD,d(rA)



EA ← (rA)+EXTS(d)  
rD←MEM(EA, 4)  
rA←EA

EA is the sum (rA|0) + d. The word in memory addressed by EA is loaded into rD.

EA is placed into rA.

If rA=0 or rA=rD, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lwzux

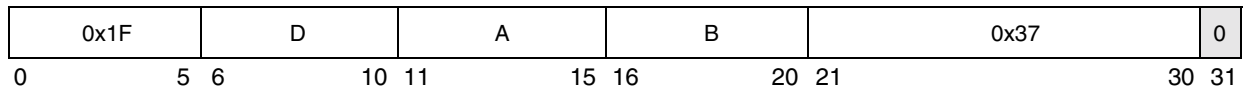
Load Word and Zero with Update Indexed

# lwzux

Load/Store Unit

**lwzux**                      **rD,rA,rB**

Reserved



$EA \leftarrow (rA) + (rB)$   
 $rD \leftarrow MEM(EA, 4)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0) + (rB)$ . The word in memory addressed by EA is loaded into rD.

EA is placed into rA.

If  $rA=0$  or  $rA=rD$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# lwzx


Load Word and Zero Indexed

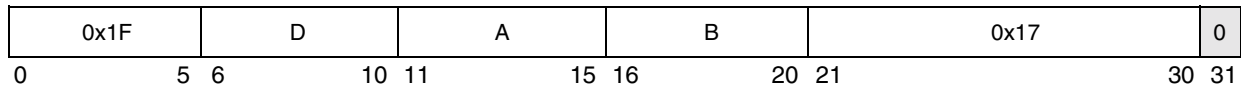
# lwzx

Load/Store Unit

**lwzx**

**rD,rA,rB**

 Reserved



if  $rA=0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + (rB)$   
 $rD \leftarrow \text{MEM}(EA, 4)$

EA is the sum  $(rA|0) + (rB)$ . The word in memory addressed by EA is loaded into rD.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.



# mcrf

Move Condition Register Field

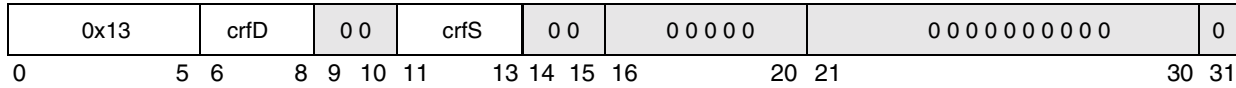
# mcrf

Branch Processor Unit

**mcrf**

**crfD,crfS**

Reserved



$$CR[4*crfD:4*crfD+3] \leftarrow CR[4*crfS:4*crfS+3]$$

The contents of condition register field **crfS** are copied into condition register field **crfD**. All other condition register fields remain unchanged.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.

# mcrfs

# mcrfs

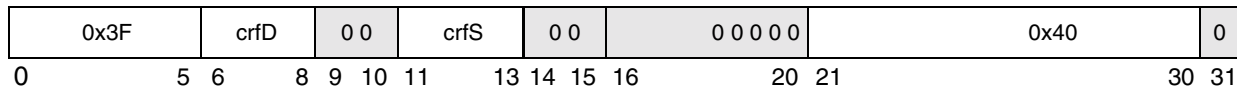
Move to Condition Register from FPSCR

Floating Point and Branch Processor Units

**mcrfs**

**crfD,crfS**

Reserved



The contents of FPSCR field **crfS** are copied to CR Field **crfD**. All other CR fields are unchanged. All exception bits copied except FEX and VX are cleared in the FPSCR.

Other registers altered:

- Condition Register (CR Field specified by operand **crfS**):
  - Affected: FX, OX (if **crfS**=0)
  - Affected: UX, ZX, XX, VXSNNAN (if **crfS**=1)
  - Affected: VXISI, VXIDI, VXZDZ, VXIMZ (if **crfS**=2)
  - Affected: VXVC (if **crfS**=3)
  - Affected: VXSOFT, VXSQRT, VXCVI (if **crfS**=5)

This instruction is defined by the PowerPC UISA.

# mcrxr


Move to Condition Register from XER

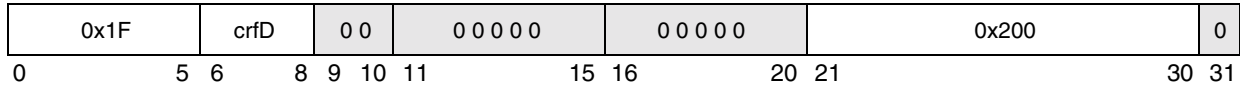
# mcrxr

Load/Store and Branch Processor Units

**mcrxr**

**crfD**

 Reserved



$CR[4*crfD:4*crfD+3] \leftarrow XER[0:3]$

$XER[0:3] \leftarrow 0b0000$

The contents of XER[0:3] are copied into the condition register field designated by **crfD**. All other fields of the condition register remain unchanged. XER[0:3] is cleared to zero.

Other registers altered:

- Condition Register (CR Field specified by **crfD** operand):  
Affected: LT, GT, EQ, SO
- XER[0:3]

This instruction is defined by the PowerPC UISA.

# mfcrr

Move from Condition Register

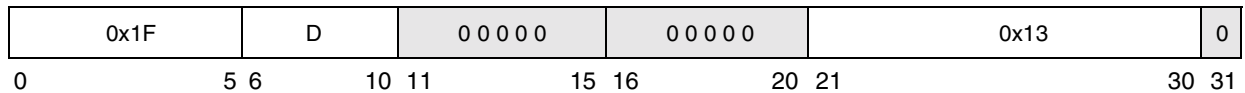
# mfcrr

Branch Processor Unit

mfcrr

rD

 Reserved



$$rD \leftarrow CR$$

The contents of the condition register are placed into rD.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

**mffs<sub>x</sub>**

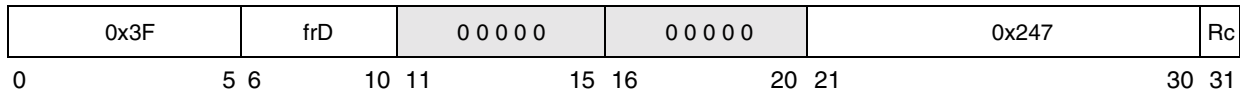
Move from FPSCR

**mffs<sub>x</sub>**

Floating-Point Unit

<b>mffs</b>	<b>frD</b>	(Rc=0)
<b>mffs.</b>	<b>frD</b>	(Rc=1)

☐ Reserved



The contents of the FPSCR are placed into **frD**[32:63]. **frD**[0:31] are undefined.

Other registers altered:

- Condition Register (CR1 Field):  
     Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

# mfmsr


Move from Machine State Register

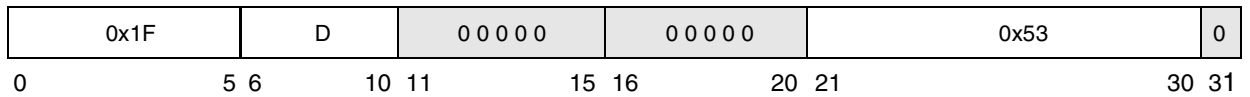
# mfmsr

Branch Processor Unit

**mfmsr**

**rD**

 Reserved



$rD \leftarrow \text{MSR}$

The contents of the MSR are placed into rD.

This is a supervisor-level instruction.

Other registers altered:

- None

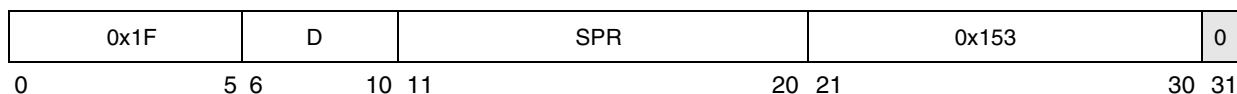
This instruction is defined by the PowerPC OEA.

**mf spr**

Move from Special Purpose Register

**mf spr**

All Units

**mf spr****rD, SPR** Reserved


$$n \leftarrow \text{SPR}[5:9] \parallel \text{SPR}[0:4]$$

$$\text{rD} \leftarrow \text{SPR}(n)$$

The SPR field denotes a special purpose register, encoded as shown in [Table 4-29](#), [Table 4-30](#), and [Table 4-31](#). The contents of the designated special purpose register are placed into rD.

For **mt spr** and **mf spr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16 to 20 of the instruction and the low-order 5 bits in bits 11 to 15.

If the SPR field contains any value other than one of the values shown in one of the tables listed above, one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The system software emulation exception handler is invoked.

The value of SPR[0] is one if and only if reading the register is at the supervisor level. Execution of this instruction specifying a supervisor-level register when MSR[PR]=1 will result in a supervisor-level instruction type program exception or a software emulation exception. Refer to [SECTION 6 EXCEPTIONS](#) for details.

If the SPR field contains a value that is not valid for the RCPU, the instruction form is invalid. For an invalid instruction form in which SPR[0]=1, if MSR[PR]=1 a supervisor-level instruction type program exception will occur instead of a no-op.

The execution unit that executes the **mf spr** instruction depends on the SPR. Moves from the XER and from SPRs that are physically implemented outside the processor are handled by the LSU. Moves from the FPSCR and FPECR are executed by the FPU. In all other cases, the BPU executes the **mf spr** instruction.

Other registers altered:

- None

Table 9-19 Simplified Mnemonics for mfspr Instruction

Operation	Simplified Mnemonic	Equivalent To
Move from XER	mfxer rD	mfspr rD,1
Move from LR	mflr rD	mfspr rD,8
Move from CTR	mfctr rD	mfspr rD,9
Move from DSISR	mfdsisr rD	mfspr rD,18
Move from DAR	mfdar rD	mfspr rD,19
Move from DEC	mfdec rD	mfspr rD,22
Move from SRR0	mfsrr0 rD	mfspr rD,26
Move from SRR1	mfsrr1 rD	mfspr rD,27
Move from SPRG	mfsprg rD, <i>n</i>	mfspr rD,272+ <i>n</i>
Move from TBL	mftb rD	mftb rD,268
Move from TBU	mftbu rD	mftb rD,269
Move from PVR	mfpvr rD	mfspr rD,287

This instruction is defined by the PowerPC UISA.



**mftb**

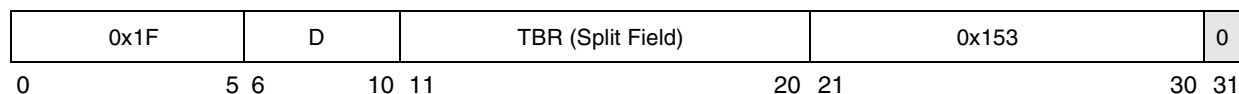
Move from Time Base

**mftb**

Load/Store Unit

**mftb****rD**,TBR

Reserved



```

n ← TBR[5:9] || TBR[0:4]
if n = 268 then
    rD ← TBL
else if n = 269 then
    rD ← TBU

```

The TBR field denotes either the time base lower (TBL) or time base upper (TBU), encoded as shown in [Table 9-20](#). Notice that the order of the two 5-bit halves of the TBR number is reversed in the instruction. The contents of the designated register are copied into **rD**.

**Table 9-20 TBR Encodings for mftb**

Decimal	TBR[5:9]	TBR[0:4]	Register Name	Access
268	01000	01100	TBL	User
269	01000	01101	TBU	User

If the TBR field contains any value other than one of the values shown in [Table 9-20](#), one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The results are boundedly undefined

Note that **mftb** serves as both a basic and a simplified mnemonic. The assembler recognized an **mftb** mnemonic with two operands as the basic form and an **mftb** mnemonic with one operand as the simplified form. If **mftb** is coded with one operand, then that operand is assumed to be **rD**, and TBR defaults to the value corresponding to TBL.

Other registers altered:

- None

This instruction is defined by the PowerPC VEA.

**Table 9-21 Simplified Mnemonics for mfspr Instruction**

Operation	Simplified Mnemonic	Equivalent To
Move from TBL	mftb rD	mftb rD,268
Move from TBU	mftbu rD	mftb rD,269

# mtcrf

Move to Condition Register Fields

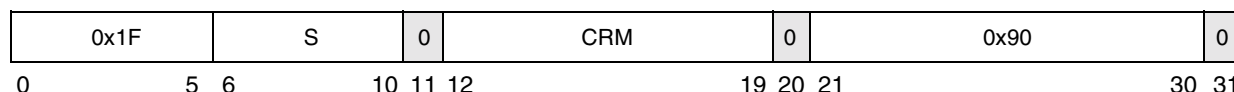
# mtcrf

Branch Processor Unit

**mtcrf**

CRM,rS

Reserved



$mask \leftarrow (4)(CRM[0]) \parallel (4)(CRM[1]) \parallel \dots (4)(CRM[7])$

$CR \leftarrow ((rS) \& mask) \mid (CR \& \neg mask)$

The contents of rS are placed into the condition register under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If CRM(i) = 1, CR Field i (CR bits 4\*i through 4\*i+3) is set to the contents of the corresponding field of rS.

Other registers altered:

- CR fields selected by mask

**Table 9-22 Simplified Mnemonics for mtcrr Instruction**

Operation	Simplified Mnemonic	Equivalent To
Move to condition register	mtcr rS	mtcrf 0xFF,rS

This instruction is defined by the PowerPC UISA.

# mtfsb0<sub>x</sub>

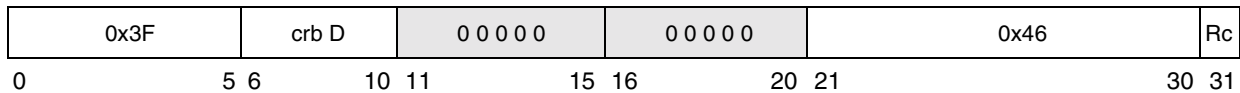
Move to FPSCR Bit 0

# mtfsb0<sub>x</sub>

Floating-Point Unit

mtfsb0                      crbD                      (Rc=0)  
 mtfsb0.                    crbD                      (Rc=1)

☐ Reserved



Bit **crbD** of the FPSCR is cleared to zero. All other bits of the FPSCR are unchanged.

Other registers altered:

- Condition Register (CR1 Field):  
 Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
 Affected: FPSCR bit **crbD**  
**Note:** Bits 1 and 2 (FEX and VX) cannot be explicitly reset.

This instruction is defined by the PowerPC UISA.

mtfsb1<sub>x</sub>

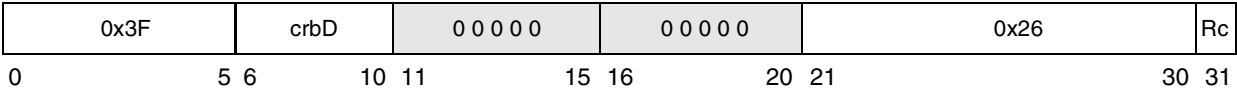
Move to FPSCR Bit 1

mtfsb1<sub>x</sub>

Floating-Point Unit

mtfsb1                      crbD                      (Rc=0)  
mtfsb1.                      crbD                      (Rc=1)

☐ Reserved



Bit **crbD** of the FPSCR is set to one. All other bits of the FPSCR are unchanged.

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
FPSCR bit **crbD** and FX  
**Note:** Bits 1 and 2 (FEX and VX) cannot be explicitly set.

This instruction is defined by the PowerPC UISA.

# mtfsf<sub>x</sub>

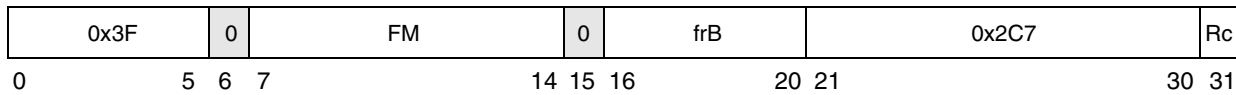
Move to FPSCR Fields

# mtfsf<sub>x</sub>

Floating-Point Unit

**mtfsf** FM,frB (Rc=0)  
**mtfsf.** FM,frB (Rc=1)

☐ Reserved



**frB**[32:63] are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0–7. If FM(*i*)=1, FPSCR Field *i* (FPSCR bits 4\**i* through 4\**i*+3) is set to the contents of the corresponding field of the low-order 32 bits of register **frB**.

FPSCR[FX] is altered only if FM[0]=1.

In some PowerPC implementations, updating fewer than all eight fields of the FPSCR may have substantially poorer performance than updating all the fields. This is not the case with the RCPU.

When FPSCR[0:3] is specified, bits 0 (FX) and 3 (OX) are set to the values of **frB**[32] and **frB**[35] (i.e., even if this instruction causes OX to change from zero to one, FX is set from **frB**[32] and not by the usual rule that FX is set to one when an exception bit changes from zero to one). Bits 1 and 2 (FEX and VX) are set according to the usual rule and not from **frB**[33:34].

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX (if Rc=1)
- Floating-point Status and Control Register:  
FPSCR fields selected by mask

This instruction is defined by the PowerPC UISA.

# mtfsfi<sub>x</sub>

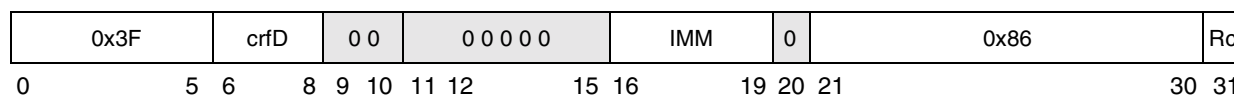
Move to FPSCR Field Immediate

# mtfsfi<sub>x</sub>

Floating-Point Unit

**mtfsfi**                      **crfD**,IMM                      (Rc=0)  
**mtfsfi.**                      **crfD**,IMM                      (Rc=1)

☐ Reserved



The value of the IMM field is placed into FPSCR field **crfD**.

FPSCR[FX] is altered only if **crfD** = 0.

When FPSCR[0:3] is specified, bits 0 (FX) and 3 (OX) are set to the values of IMM[0] and IMM[3] (i.e., even if this instruction causes OX to change from zero to one, FX is set from IMM[0] and not by the usual rule that FX is set to one when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule, given in [2.2.3 Floating-Point Status and Control Register \(FPSCR\)](#) and not from IMM[1:2].

Other registers altered:

- Condition Register (CR1 Field):  
Affected: FX, FEX, VX, OX                      (if Rc=1)
- Floating-point Status and Control Register:  
FPSCR field **crfD**

This instruction is defined by the PowerPC UISA.

# mtmsr

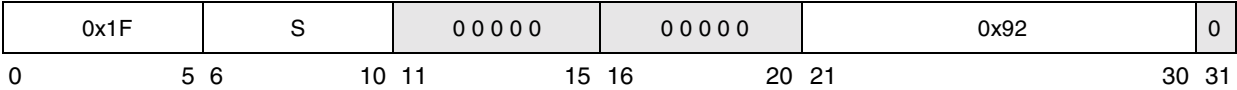
Move to Machine State Register

# mtmsr

Branch Processor Unit

mtmsr                      rS

 Reserved



$$MSR \leftarrow rS$$

The contents of rS are placed into the MSR.

This is a supervisor-level, executing-synchronizing instruction.

Other registers altered:

- MSR

This instruction is defined by the PowerPC OEA.

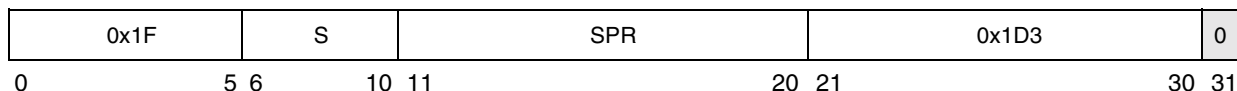


**mtspr**

Move to Special Purpose Register

**mtspr**

All Units

**mtspr****SPR,rS** Reserved


$$n = \text{SPR}[5:9] \parallel \text{SPR}[0:4]$$

$$\text{SPREG}(n) \leftarrow (rS)$$

The SPR field denotes a special purpose register, encoded as shown in [Table 4-29](#), [Table 4-30](#), and [Table 4-31](#). The contents of **rS** are placed into the designated special purpose register.

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16 to 20 of the instruction and the low-order 5 bits in bits 11 to 15.

If the SPR field contains any value other than one of the values shown in one of the tables listed above, one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The software emulation exception handler is invoked.

Note that, for this instruction, SPRs TBL and TBU are treated as separate registers; setting one leaves the other unaltered.

The value of SPR[0] is one if and only if the register is read at the supervisor-level. Execution of this instruction specifying a supervisor-level register when MSR[PR]=1 results in a supervisor-level instruction type program exception or software emulation exception.

If the SPR field contains a value that is not valid for the RCPU, the instruction form is invalid. For an invalid instruction form in which SPR[0]=1, if MSR[PR]=1 a supervisor-level instruction type program exception will occur instead of a no-op.

The execution unit that executes the **mtspr** instruction depends on the SPR. Moves to the XER and to SPRs that are physically implemented outside the processor are handled by the LSU. Moves to the FPSCR and FPECR are executed by the FPU. In all other cases, the BPU executes the **mtspr** instruction.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# Freescale Semiconductor, Inc.

**Table 9-23 Simplified Mnemonics for mtspr Instruction**

Operation	Simplified Mnemonic	Equivalent To
Move to XER	mtxr rS	mtspr 1,rS
Move to LR	mtlr rS	mtspr 8,rS
Move to CTR	mtctr rS	mtspr 9,rS
Move to DSISR	mtdsisr rS	mtspr 18,rS
Move to DAR	mtdar rS	mtspr 19,rS
Move to DEC	mtdec rS	mtspr 22,rS
Move to SRR0	mtsrr0 rS	mtspr 26,rS
Move to SRR1	mtsrr1 rS	mtspr 27,rS
Move to SPRG	mtsprg <i>n</i> , rS	mtspr 272+ <i>n</i> ,rS
Move to TBL	mttbl rS	mtspr 284,rS
Move to TBU	mttbu rS	mtspr 285,rS

# mulhw<sub>x</sub>

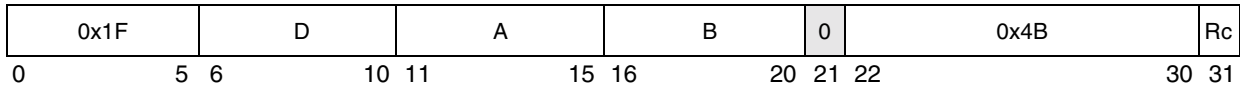
Multiply High Word

# mulhw<sub>x</sub>

Integer Unit

**mulhw**                    **rD,rA,rB**                    (Rc=0)  
**mulhw.**                   **rD,rA,rB**                    (Rc=1)

Reserved



prod[0:63]←(rA)\*(rB)  
rD←prod[0:31]

The contents of **rA** and of **rB** are interpreted as 32-bit signed integers. They are multiplied to form a 64-bit signed integer product. The high-order 32 bits of the 64-bit product are placed into **rD**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                    (if Rc=1)

This instruction is defined by the PowerPC UISA.

# mulhwu<sub>x</sub>

Multiply High Word Unsigned

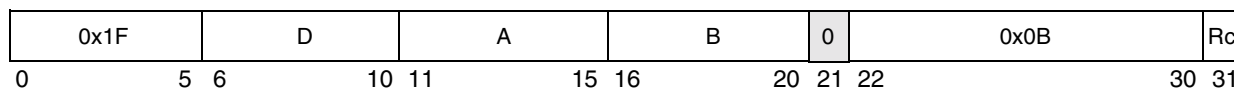
# mulhwu<sub>x</sub>

Integer Unit

**mulhwu**                      **rD,rA,rB**                      (Rc=0)

**mulhwu.**                      **rD,rA,rB**                      (Rc=1)

☐ Reserved



$prod[0:63] \leftarrow (rA) * (rB)$

$rD \leftarrow prod[0:31]$

The contents of **rA** and of **rB** are interpreted as 32-bit unsigned integers. They are multiplied to form a 64-bit unsigned integer product. The high-order 32 bits of the 64-bit product are placed into **rD**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

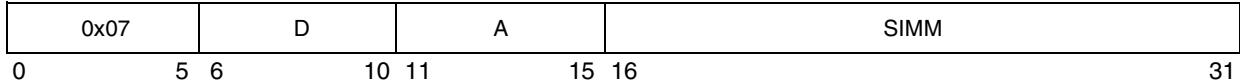
# mulli

Multiply Low Immediate

# mulli

Integer Unit

**mulli**                    **rD,rA,SIMM**



```
prod[0:47]←rA*SIMM
rD←prod[16:47]
```

The low-order 32 bits of the 48-bit product (**rA**)\*SIMM are placed into **rD**. The low-order bits are calculated independently of whether the operands are treated as signed or unsigned 32-bit integers.

This instruction can be used with **mulhwx** to calculate a full 64-bit product.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

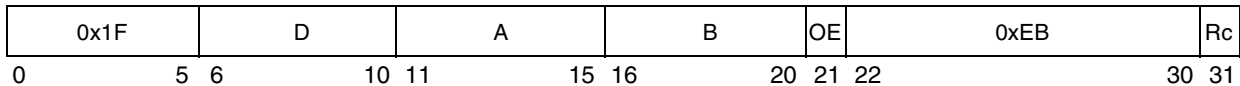
# mullw<sub>x</sub>

Multiply Low

# mullw<sub>x</sub>

Integer Unit

<b>mullw</b>	<b>rD,rA,rB</b>	(OE=0 Rc=0)
<b>mullw.</b>	<b>rD,rA,rB</b>	(OE=0 Rc=1)
<b>mullwo</b>	<b>rD, rA,rB</b>	(OE=1 Rc=0)
<b>mullwo.</b>	<b>rD,rA,rB</b>	(OE=1 Rc=1)



```
prod[0:63]←(rA)*(rB)
rD←prod[32:63]
```

The low-order 32 bits of the 64-bit product (rA)\*(rB) are placed into rD. The low-order bits are calculated independently of whether the operands are treated as signed or unsigned integers. However, OV is set based on the result interpreted as a signed integer.

If OE=1, then OV is set to one if the product cannot be represented in 32 bits.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

# nand<sub>x</sub>

NAND

# nand<sub>x</sub>

Integer Unit

**nand**                      **rA,rS,rB**                      (**Rc=0**)

**nand.**                      **rA,rS,rB**                      (**Rc=1**)

0x1F						S					A					B					0x1DC												Rc
0		5		6		10		11		15		16		20		21		30												31			

$$rA \leftarrow \neg ((rS) \& (rB))$$

The contents of **rS** are ANDed with the contents of **rB**, and the complemented result is placed into **rA**.

**nand** with **rS = rB** can be used to obtain the one's complement.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)

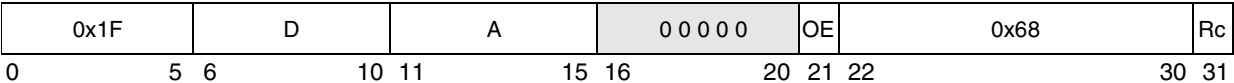
This instruction is defined by the PowerPC UISA.

**neg<sub>x</sub>**  
Negate

**neg<sub>x</sub>**  
Integer Unit

<b>neg</b>	<b>rD,rA</b>	(OE=0 Rc=0)
<b>neg.</b>	<b>rD,rA</b>	(OE=0 Rc=1)
<b>nego</b>	<b>rD,rA</b>	(OE=1 Rc=0)
<b>nego.</b>	<b>rD,rA</b>	(OE=1 Rc=1)

Reserved



$$rD \leftarrow \neg(rA) + 1$$

The sum  $\neg(rA) + 1$  is placed into **rD**.

If **rA** contains the most negative 32-bit number (0x8000 0000), the result is the most negative 32-bit number, and if OE=1, OV is set.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: SO OV (if OE=1)

This instruction is defined by the PowerPC UISA.



**nor<sub>x</sub>**

NOR

**nor<sub>x</sub>**

Integer Unit

**nor**                      **rA,rS,rB**                      (Rc=0)

**nor.**                      **rA,rS,rB**                      (Rc=1)

0x1F	S	A	B	0x7C	Rc
0	5 6	10 11	15 16	20 21	30 31

$$rA \leftarrow \neg ((rS) | (rB))$$

The contents of **rS** are ORed with the contents of **rB**, and the one's complement of the result is placed into **rA**.

**nor** with **rS=rB** can be used to obtain the one's complement.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

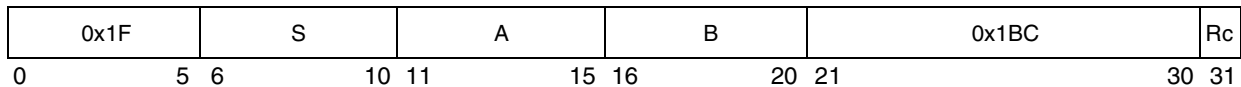
**Table 9-24 Simplified Mnemonics for nor Instruction**

Operation	Simplified Mnemonic	Equivalent To
Complement register	<b>not rA, rS</b> <b>not. rA, rS</b>	<b>nor rA,rS,rS</b> <b>nor. rA,rS,rS</b>

**or<sub>x</sub>**  
OR

**or<sub>x</sub>**  
Integer Unit

**or**                      **rA,rS,rB**                      (Rc=0)  
**or.**                      **rA,rS,rB**                      (Rc=1)



$$rA \leftarrow (rS) | (rB)$$

The contents of **rS** is ORed with the contents of **rB** and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

**Table 9-25 Simplified Mnemonics for or Instruction**

Operation	Simplified Mnemonic	Equivalent To
Move register	<b>mr rA, rS</b> <b>mr. rA, rS</b>	<b>or rA,rS,rS</b> <b>or. rA,rS,rS</b>

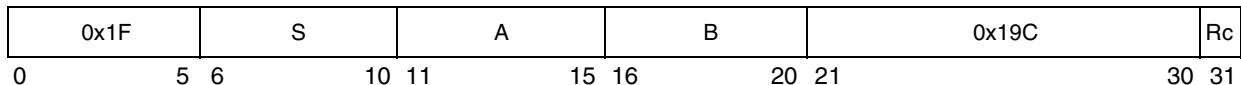
# orc<sub>x</sub>

OR with Complement

# orc<sub>x</sub>

Integer Unit

**orc**                      **rA,rS,rB**                      (Rc=0)  
**orc.**                      **rA,rS,rB**                      (Rc=1)



$$rA \leftarrow (rS) \mid \neg (rB)$$

The contents of **rS** is ORed with the complement of the contents of **rB** and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

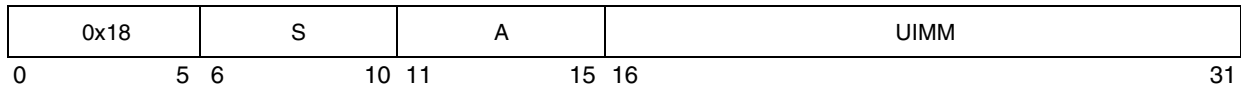
# ori

OR Immediate

# ori

Integer Unit

**ori**                      **rA,rS,UIMM**



$$rA \leftarrow (rS) \mid ((16)0 \parallel UIMM)$$

The contents of **rS** is ORed with 0x0000 || UIMM and the result is placed into **rA**.

The preferred no-op is:

**ori** 0,0,0

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

**Table 9-26 Simplified Mnemonics for ori Instruction**

Operation	Simplified Mnemonic	Equivalent To
No operation	nop	ori 0,0,0

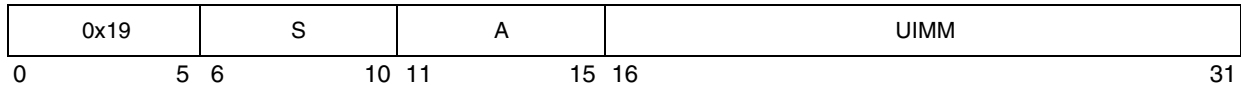
# oris

OR Immediate Shifted

# oris

Integer Unit

**oris**                    **rA,rS,UIMM**



$$rA \leftarrow (rS) \mid (UIMM \mid (16)0)$$

The contents of **rS** is ORed with **UIMM** || 0x0000 and the result is placed into **rA**.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

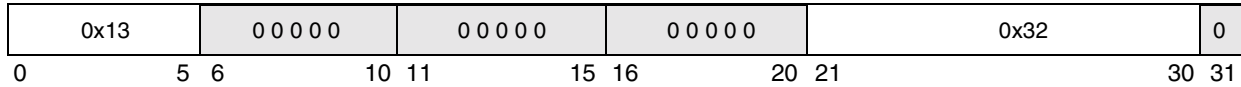
rfi

Return from Interrupt

rfi

Branch Processor Unit

☐ Reserved



$MSR[16:31] \leftarrow SRR1[16:31]$

$NIA \leftarrow SRR0[0:29] \parallel 0b00$

SRR1[16:31] are placed into MSR[16:31]. If the new MSR value does not enable any pending exceptions, then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0:29] || 0b00. If the new MSR value enables one or more pending exceptions, the exception associated with the highest priority pending exception is generated; in this case the value placed into SRR0 by the exception processing mechanism is the address of the instruction that would have been executed next had the exception not occurred.

This is a supervisor-level, context-synchronizing instruction.

Other registers altered:

- MSR

This instruction is defined by the PowerPC OEA.

**rlwimi<sub>x</sub>**

Rotate Left Word Immediate then Mask Insert

**rlwimi<sub>x</sub>**

Integer Unit

**rlwimi**    **rA,rS,SH,MB,ME**                      (**Rc=0**)**rlwimi.**   **rA,rS,SH,MB,ME**                      (**Rc=1**)

0x14		S		A		SH		MB		ME		Rc
0	5	6	10	11	15	16	20	21	25	26	30	31

 $n \leftarrow \text{SH}$  $r \leftarrow \text{ROTL}(rS, n)$  $m \leftarrow \text{MASK}(\text{MB}, \text{ME})$  $rA \leftarrow (r \& m) \mid (rA \& \neg m)$ 

The contents of **rS** are rotated left **SH** bits. A mask is generated having 1-bits from bit **MB** through bit **ME** and 0-bits elsewhere. The rotated data is inserted into **rA** under control of the generated mask.

Note that **rlwimi** can be used to insert a bit field into the contents of **rA** using the methods shown below:

- To insert an  $n$ -bit field that is left-justified in **rS** into **rA** starting at bit position  $b$ , set  $\text{SH} = 32 - b$ ,  $\text{MB} = b$ , and  $\text{ME} = b + n - 1$
- To insert an  $n$ -bit field that is right-justified in **rS** into **rA** starting at bit position  $b$ , set  $\text{SH} = 32 - (b + n)$ ,  $\text{MB} = b$ , and  $\text{ME} = b + n - 1$

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if **Rc=1**)

This instruction is defined by the PowerPC UISA.

**Table 9-27 Simplified Mnemonics for rlwimi Instruction**

Operation	Simplified Mnemonic	Equivalent To
Insert from left immediate	<b>inslwi</b> <b>rA,rS,n,b</b> <b>inslwi.</b> <b>rA,rS,n,b</b>	<b>rlwimi</b> <b>rA,rS,32-b,b,b+n-1</b> <b>rlwimi.</b> <b>rA,rS,32-b,b,b+n-1</b>
Insert from right immediate	<b>insrwi</b> <b>rA,rS,n,b</b> <b>insrwi.</b> <b>rA,rS,n,b</b>	<b>rlwimi</b> <b>rA,rS,32-(b+n),b,b+n-1</b> <b>rlwimi.</b> <b>rA,rS,32-(b+n),b,b+n-1</b>

# rlwinm<sub>x</sub>

Rotate Left Word Immediate then AND with Mask

# rlwinm<sub>x</sub>

Integer Unit

**rlwinm** **rA,rS,SH,MB,ME** (Rc=0)

**rlwinm.** **rA,rS,SH,MB,ME** (Rc=1)

0x15	S	A	SH	MB	ME	Rc
0 5 6	10 11	15 16	20 21	25 26	30 31	

$n \leftarrow SH$   
 $r \leftarrow ROTL(rS, n)$   
 $m \leftarrow MASK(MB, ME)$   
 $rA \leftarrow r \& m$

The contents of **rS** are rotated left **SH** bits. A mask is generated having 1-bits from bit **MB** through bit **ME** and 0-bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **rA**.

Note that **rlwinm** can be used to extract, rotate, or clear bit fields using the following methods:

- To extract an *n*-bit field that starts at bit position *b* in **rS**[0:31], right-justified into **rA** (clearing the remaining 32-*n* bits of **rA**), set **SH**=*b*+*n*, **MB**=32-*n*, and **ME**=31.
- To extract an *n*-bit field that starts at bit position *b* in **rS**[0-31], left-justified into **rA** (clearing the remaining 32-*n* bits of **rA**), set **SH**=*b*, **MB**=0, and **ME**=*n*-1.
- To rotate the contents of a register left (or right) by *n* bits, set **SH**=*n* (32-*n*), **MB**=0, and **ME**=31.
- To shift the contents of a register right by *n* bits, set **SH**=32-*n*, **MB**=*n*, and **ME**=31.
- To clear the high-order *b* bits of a register and then shift the result left by *n* bits, set **SH**=*n*, **MB**=*b*-*n* and **ME**=31-*n*.
- To clear the low-order *n* bits of a register, set **SH**=0, **MB**=0, and **ME**=31-*n*.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)

This instruction is defined by the PowerPC UISA.



**Table 9-28 Simplified Mnemonics for rlwinm Instruction**

Operation	Simplified Mnemonic	Equivalent To
Extract and left justify immediate	<b>extlwi</b> $rA, rS, n, b$ ( $n > 0$ ) <b>extlwi.</b> $rA, rS, n, b$ ( $n > 0$ )	<b>rlwinm</b> $rA, rS, b, 0, n-1$ <b>rlwinm.</b> $rA, rS, b, 0, n-1$
Extract and right justify immediate	<b>extrwi</b> $rA, rS, n, b$ ( $n > 0$ ) <b>extrwi.</b> $rA, rS, n, b$ ( $n > 0$ )	<b>rlwinm</b> $rA, rS, b + n, 32 - n, 31$ <b>rlwinm.</b> $rA, rS, b + n, 32 - n, 31$
Rotate left immediate	<b>rotlwi</b> $rA, rS, n$ <b>rotlwi.</b> $rA, rS, n$	<b>rlwinm</b> $rA, rS, n, 0, 31$ <b>rlwinm.</b> $rA, rS, n, 0, 31$
Rotate right immediate	<b>rotrwi</b> $rA, rS, n$ <b>rotrwi.</b> $rA, rS, n$	<b>rlwinm</b> $rA, rS, 32 - n, 0, 31$ <b>rlwinm.</b> $rA, rS, 32 - n, 0, 31$
Shift left immediate	<b>srwi</b> $rA, rS, n$ ( $n < 32$ ) <b>srwi.</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, n, 0, 31-n$ <b>rlwinm.</b> $rA, rS, n, 0, 31-n$
Shift right immediate	<b>srwi</b> $rA, rS, n$ ( $n < 32$ ) <b>srwi.</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, 32-n, n, 31$ <b>rlwinm.</b> $rA, rS, 32-n, n, 31$
Clear left immediate	<b>clrlwi</b> $rA, rS, n$ ( $n < 32$ ) <b>clrlwi.</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, 0, n, 31$ <b>rlwinm.</b> $rA, rS, 0, n, 31$
Clear right immediate	<b>clrrwi</b> $rA, rS, n$ ( $n < 32$ ) <b>clrrwi.</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, 0, 0, 31-n$ <b>rlwinm.</b> $rA, rS, 0, 0, 31-n$
Clear left and shift left immediate	<b>clrlslwi</b> $rA, rS, b, n$ ( $n \neq b \neq 31$ ) <b>clrlslwi.</b> $rA, rS, b, n$ ( $n \neq b \neq 31$ )	<b>rlwinm</b> $rA, rS, n, b-n, 31-n$ <b>rlwinm.</b> $rA, rS, n, b-n, 31-n$

## Integer Unit

**rlwnm.      rA,rS,rB,MB,ME      (Rc=1)**

```

n ← rB[27:31]
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m

```

Note that **rlwnm** can be used to extract and rotate bit fields using the following methods:

- To extract an  $n$ -bit field that starts at variable bit position  $b$  in  $rS[0:31]$ , right-justified into  $rA$  (clearing the remaining  $32-n$  bits of  $rA$ ), set  $rB[27:31]=b+n$ ,  $MB=32-n$ , and  $ME=31$ .
- To extract an  $n$ -bit field, that starts at variable bit position  $b$  in  $rS[0:31]$ , left-justified into  $rA$  (clearing the remaining  $32-n$  bits of  $rA$ ), set  $rB[27:31]=b$ ,  $MB=0$ , and  $ME=n-1$ .
- To rotate the contents of a register left (or right) by variable  $n$  bits, set  $rB[27:31]=n$  ( $32-N$ ),  $MB=0$ , and  $ME=31$ .

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)

This instruction is defined by the PowerPC UISA.

### Table 9-29 Simplified Mnemonics for rlwnm Instruction

Operation	Simplified Mnemonic	Equivalent To
Rotate left	<b>rotlw</b> rA,rS,rB <b>rotlw.</b> rA,rS,rB	<b>rlwnm</b> rA,rS,rB,0,31 <b>rlwnm.</b> rA,rS,rB,0,31

SC

System Call

SC

Branch Processor Unit

☐ Reserved

0x11	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1	0
0	5 6	10 11	15 16	29 30	31

This instruction calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers depends on the register conventions used by the program providing the system service.

The effective address of the instruction following the system call instruction is placed into SRR0. MSR[16:31] are placed into SRR1[16:31], and SRR1[0:15] are set to undefined values.

Then a system call exception is generated. The exception causes the MSR to be altered as described in [6.11.8 System Call Exception \(0x00C00\)](#).

The exception causes the next instruction to be fetched from offset 0xC00 from the physical base address indicated by the new setting of MSR[IP]. This instruction is context-synchronizing.

Other registers altered:

- Dependent on the system service
- SRR0
- SRR1
- MSR

This instruction is defined by the PowerPC UISA.

# slw<sub>x</sub>

Shift Left Word

# slw<sub>x</sub>

Integer Unit

**slw**                      **rA,rS,rB**                      (Rc=0)  
**slw.**                      **rA,rS,rB**                      (Rc=1)

0x1F	S	A	B	0x18	Rc
0	5 6	10 11	15 16	20 21	30 31

```

n ← rB[27:31]
r ← ROTL((rS), n)
if rB[26]=0 then
    m ← MASK(0,31-n)
else
    m ← (32)0
rA ← r&m
    
```

If **rB[26]=0**, the contents of **rS** are shifted left the number of bits specified by **rB[27:31]**. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into **rA**. If **rB[26]=1**, 32 zeros are placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)

This instruction is defined by the PowerPC UISA.

# sraw<sub>x</sub>

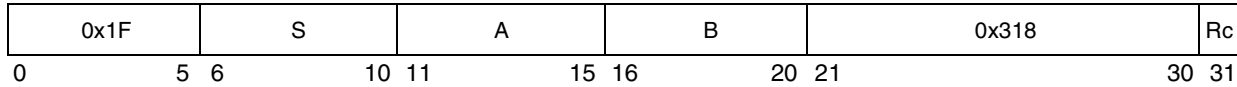
Shift Right Algebraic Word

# sraw<sub>x</sub>

Integer Unit

**sraw**                      **rA,rS,rB**                      (Rc=0)

**sraw.**                      **rA,rS,rB**                      (Rc=1)



```

n ← rB[27:31]
r ← ROTL((rS), 32-n)
if rB[26]=0 then
    m ← MASK(n,31)
else
    m ← (32)0
s ← rS[0]
rA ← r & m | (32)s & ¬ m
XER[CA] ← s & ((r & ¬ m);0)
    
```

If **rB[26]=0**, then the contents of **rS** are shifted right the number of bits specified by **rB[27:31]**. Bits shifted out of position 31 are lost. The result is padded on the left with sign bits before being placed into **rA**. If **rB[26]=1**, then **rA** is filled with 32 sign bits (bit 0) from **rS**. CR0 is set based on the value written into **rA**.

**XER[CA]** is set to one if **rS** contains a negative number and any 1-bits are shifted out of position 31; otherwise **XER[CA]** is cleared to zero. A shift amount of zero causes **XER[CA]** to be cleared.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)
- XER:  
Affected: CA

This instruction is defined by the PowerPC UISA.

# srawi<sub>x</sub>

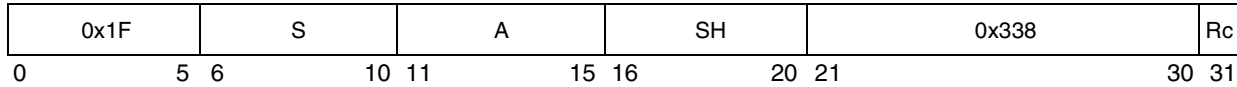
Shift Right Algebraic Word Immediate

# srawi<sub>x</sub>

Integer Unit

**srawi**                      **rA,rS,SH**                      (Rc=0)

**srawi.**                      **rA,rS,SH**                      (Rc=1)



```

n ← SH
r ← ROTL((rS), 32-n)
m ← MASK(n,31)
s ← rS[0]
rA ← r & m | (32)s & ¬ m
XER[CA] ← s & ((r & ¬ m);0)
    
```

The contents of **rS** are shifted right **SH** bits. Bits shifted out of position 31 are lost. The shifted value is sign extended before being placed in **rA**. The 32-bit result is placed into **rA**. **XER[CA]** is set to one if **rS** contains a negative number and any 1-bits are shifted out of position 31; otherwise **XER[CA]** is cleared to zero. A shift amount of zero causes **XER[CA]** to be cleared to zero.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)
- XER:  
Affected: CA

This instruction is defined by the PowerPC UISA.

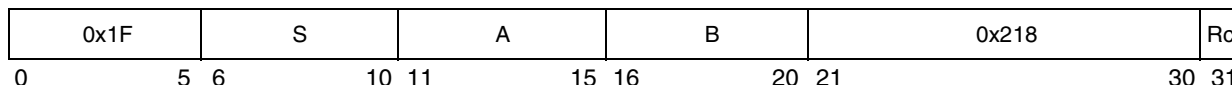
# srw<sub>x</sub>

Shift Right Word

# srw<sub>x</sub>

Integer Unit

**srw**                      **rA,rS,rB**                      (**Rc=0**)  
**srw.**                      **rA,rS,rB**                      (**Rc=1**)



```

n ← rB[27:31]
r ← ROTL((rS), 32-n)
if rB[26]=0 then
    m ← MASK(n,31)
else
    m ← (32)0
rA ← r & m
    
```

If **rB[26]=0**, the contents of **rA** are shifted right the number of bits specified by **rA[27:31]**. Bits shifted out of position 31 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into **rA**.

If **rB[26]=1**, then **rA** is filled with zeros.

Other registers altered:

- Condition Register (CR0 Field):  
 Affected: LT, GT, EQ, SO                      (if **Rc=1**)

This instruction is defined by the PowerPC UISA.

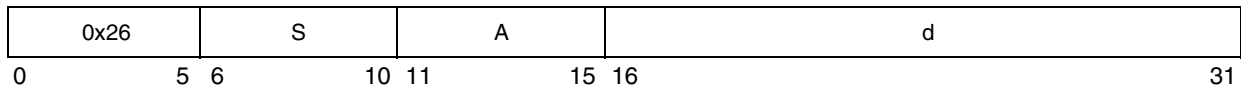
# stb

Store Byte

# stb

Load/Store Unit

**stb**                      **rS,d(rA)**



if  $rA = 0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + \text{EXTS}(d)$   
 $\text{MEM}(EA, 1) \leftarrow rS[24:31]$

EA is the sum  $(rA|0)+d$ . The contents of  $rS[24:31]$  are stored into the byte in memory addressed by EA. Register rS is unchanged.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.



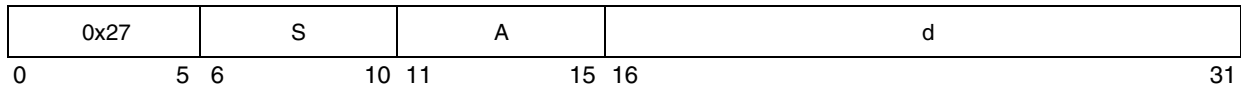
# stbu

Store Byte with Update

# stbu

Load/Store Unit

**stbu**                      **rS,d(rA)**



EA ← (rA) + EXTS(d)  
MEM(EA, 1) ← rS[24:31]  
rA ← EA

EA is the sum (rA|0)+d. The contents of rS[24:31] are stored into the byte in memory addressed by EA.

EA is placed into rA.

If rA=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stbux

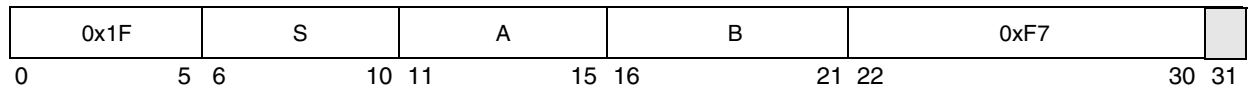
Store Byte with Update Indexed

# stbux

Load/Store Unit

**stbux**                      **rS,rA,rB**

 Reserved



EA ← (rA) + (rB)  
 MEM(EA, 1) ← rS[24:31]  
 rA ← EA

EA is the sum (rA|0)+(rB). The contents of rS[24:31] is stored into the byte in memory addressed by EA.

EA is placed into rA.

If rA=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stbx

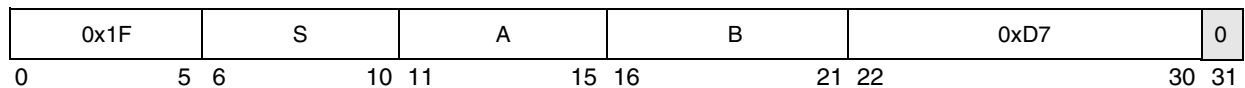
Store Byte Indexed

# stbx

Load/Store Unit

**stbx**                      **rS,rA,rB**

☐ Reserved



if **rA** = 0 then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + (rB)$   
 $EM(EA, 1) \leftarrow rS[24:31]$

EA is the sum  $(rA|0)+(rB)$ .

The contents of **rS**[24:31] is stored into the byte in memory addressed by EA. Register **rS** is unchanged.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

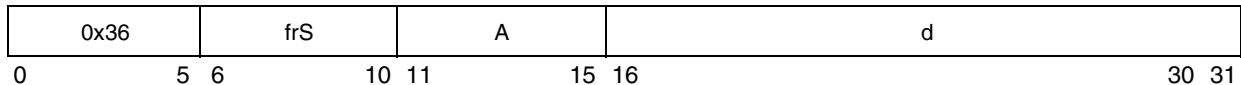
# stfd

Store Floating-Point Double-Precision

# stfd

Floating-Point Unit

**stfd**                      **frS,d(rA)**



```

if rA = 0 then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 8) ← (frS)
    
```

EA is the sum (rA|0)+d.

The contents of **frS** are stored into the double word in memory addressed by EA.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

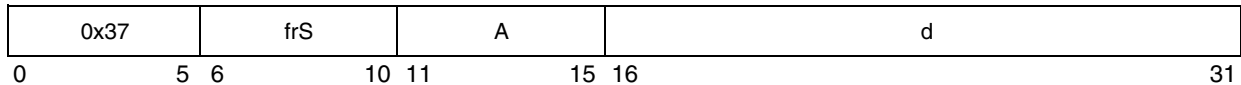
# stfdu

Store Floating-Point Double-Precision with Update

# stfdu

Load/Store Unit

**stfdu**                      **frS,d(rA)**



EA ← (rA) + EXTS(d)  
MEM(EA, 8) ← (frS)  
rA ← EA

EA is the sum (rA|0)+d.

The contents of frS are stored into the double word in memory addressed by EA.

EA is placed into rA.

If rA=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stfdux

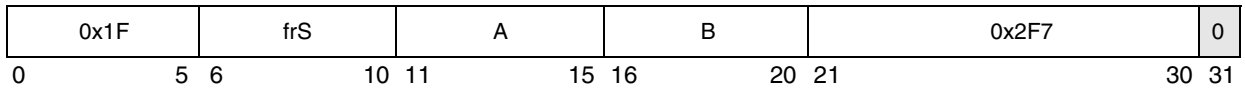
Store Floating-Point Double-Precision with Update Indexed

# stfdux

Load/Store Unit

**stfdux**                      **frS,rA,rB**

Reserved



$EA \leftarrow (rA) + (rB)$   
 $MEM(EA, 8) \leftarrow (frS)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0)+(rB)$ .

The contents of **frS** are stored into the double word in memory addressed by EA.

EA is placed into **rA**.

If **rA**=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

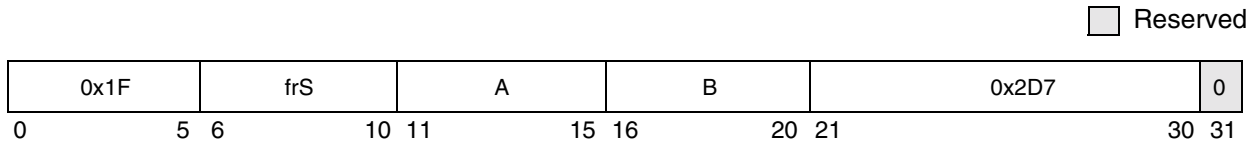
# stfdx

Store Floating-Point Double-Precision Indexed

# stfdx

Load/Store Unit

stfdx                      frS,rA,rB



if rA + 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
MEM(EA, 8) ← (frS)

EA is the sum (rA|0)+(rB).

The contents of frS are stored into the double word in memory addressed by EA.

Other registers altered:

• None

This instruction is defined by the PowerPC UISA.

# stfiwx

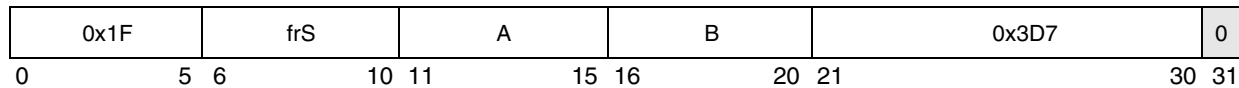
Store Floating-Point as Integer Word Indexed

# stfiwx

Load/Store Unit

**stfiwx**                      **frS,rA,rB**

Reserved



```

if rA =0 then b ←0
else b←(rA)
EA←b + (rB)
MEM(EA, 4)←frS[32:63]

```

EA is the sum (rA|0)+(rB).

The low-order 32 bits of **frS** are stored, without conversion, into the word in memory addressed by EA.

If the contents of **frS** were produced, either directly or indirectly, by an **lfs** instruction, a single-precision arithmetic instruction, or **frsp**, then the value stored is undefined. The contents of **frS** are produced directly by such an instruction if **frS** is the target register for the instruction. The contents of **frS** are produced indirectly by such an instruction if **frS** is the final target register of a sequence of one or more floating-point move instructions, with the input to the sequence having been produced directly by such an instruction.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.



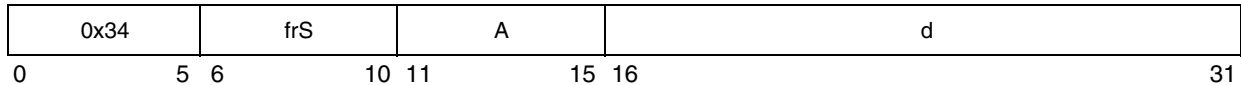
# stfs

Store Floating-Point Single-Precision

# stfs

Load/Store Unit

**stfs**                      **frS,d(rA)**



```

if rA = 0 then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)

```

EA is the sum (rA|0)+d.

The contents of **frS** are converted to single-precision and stored into the word in memory addressed by EA.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

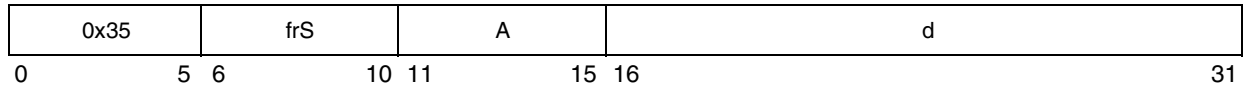
# stfsu

Store Floating-Point Single-Precision with Update

# stfsu

Integer and Floating-Point Units

**stfsu**                      **frS,d(rA)**



```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (rA|0)+d.

The of **frS** are converted to single-precision and stored into the word in memory addressed by EA.

EA is placed into **rA**.

If **rA**=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stfsux

# stfsux

Store Floating-Point Single-Precision with Update Indexed Integer/Floating-Point Units

**stfsux**      **frS,rA,rB** Reserved

0x1F	frS	A	B	0x2B7	0
0	5 6	10 11	15 16	20 21	30 31

$EA \leftarrow (rA) + (rB)$   
 $MEM(EA, 4) \leftarrow SINGLE(frS)$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0)+(rB)$ .

The contents of **frS** are converted to single-precision and stored into the word in memory addressed by EA.

EA is placed into **rA**.

If **rA**=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stfsx

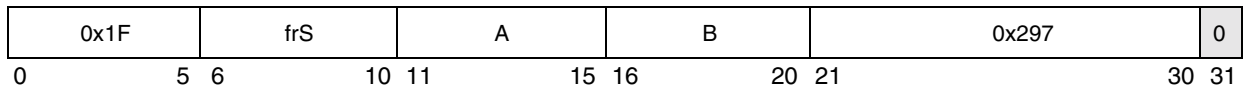
Store Floating-Point Single-Precision Indexed

# stfsx

Load/Store Unit

**stfsx**                      **frS,rA,rB**

Reserved



if **rA**=0 then **b**←0  
 else **b**←(**rA**)  
**EA**←**b** + (**rB**)  
**MEM**(**EA**, 4)←**SINGLE**(**frS**)

**EA** is the sum (**rA**|0)+(**rB**).

The contents of **frS** are converted to single-precision and stored into the word in memory addressed by **EA**.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

**sth**

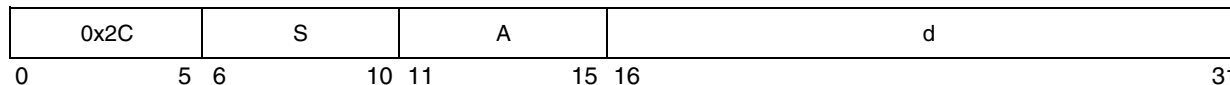
Store Half Word

**sth**

Load/Store Unit

**sth**

**rS,d(rA)**



if  $rA = 0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + \text{EXTS}(d)$   
 $\text{MEM}(EA, 2) \leftarrow rS[16:31]$

EA is the sum  $(rA|0)+d$ .

The contents of  $rS[16:31]$  are stored into the half word in memory addressed by EA.

Other registers altered:

- None

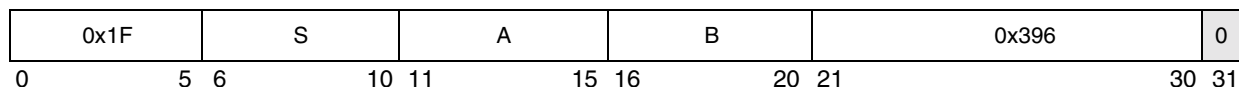
This instruction is defined by the PowerPC UISA.

**sthbrx**

Store Half Word Byte-Reverse Indexed

**sthbrx**

Load/Store Unit

**sthbrx**      **rS,rA,rB** Reserved


if  $rA = 0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + (rB)$   
 $MEM(EA, 2) \leftarrow rS[24:31] \parallel rS[16:23]$

EA is the sum  $(rA|0) + (rB)$ .

The contents of  $rS[24:31]$  are stored into bits 0:7 of the half word in memory addressed by EA. Bits  $rS[16:23]$  are stored into bits 8:15 of the half word in memory addressed by EA.

Other registers altered:

- None

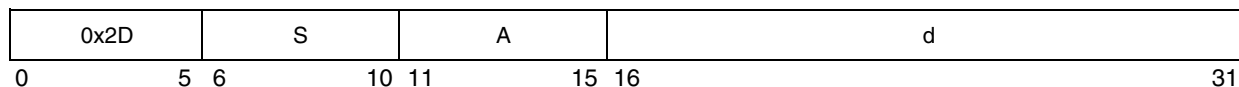
This instruction is defined by the PowerPC UISA.

**sth**

Store Half Word with Update

**sth**

Load/Store Unit

**sth**                      **rS,d(rA)**

$EA \leftarrow (rA) + \text{EXTS}(d)$   
 $\text{MEM}(EA, 2) \leftarrow rS[16:31]$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0)+d$ .

The contents of  $rS[16:31]$  are stored into the half word in memory addressed by EA.

EA is placed into  $rA$ .

If  $rA=0$ , the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# sthux

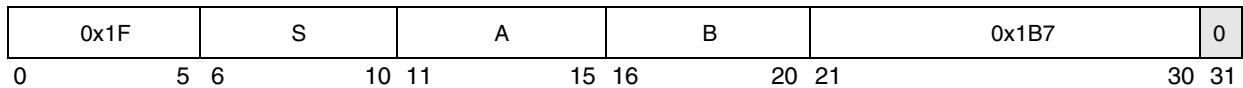
Store Half Word with Update Indexed

# sthux

Load/Store Unit

**sthux**                      rS,rA,rB

☐ Reserved



```
EA← (rA) + (rB)
MEM(EA, 2)←rS[16:31]
rA←EA
```

EA is the sum (rA|0)+(rB).

The contents of rS[16:31] are stored into the half word in memory addressed by EA.

EA is placed into rA.

If rA=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.



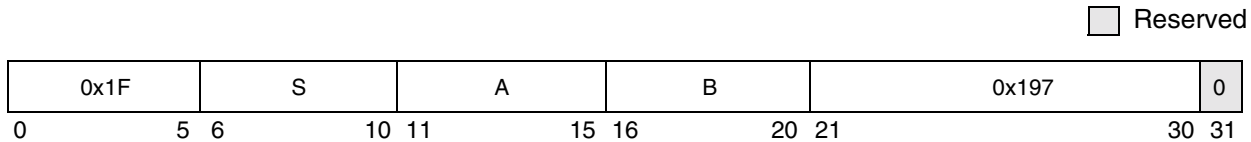
# sthx

Store Half Word Indexed

# sthx

Load/Store Unit

**sthx**                      rS,rA,rB



if rA = 0 then b←0
else b←(rA)
EA←b + (rB)
MEM(EA, 2)←rS[16:31]

EA is the sum (rA|0)+(rB).

The contents of rS[16:31] are stored into the half word in memory addressed by EA.

Other registers altered:

None

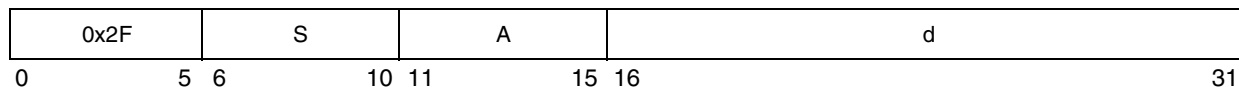
This instruction is defined by the PowerPC UISA.

**stmw**

Store Multiple Word

**stmw**

Load/Store Unit

**stmw**                      **rS,d(rA)**

```

if rA = 0 then b ← 0
else b ← (rA)
EA ← b + EXTS(d)
r ← rS
do while r ≤ 31
    MEM(EA, 4) ← GPR(r)
    r ← r + 1
    EA ← EA + 4

```

EA is the sum  $(rA|0)+d$ .

$n = (32 - rS)$ .

$n$  consecutive words starting at EA are stored from the GPRs rS through 31. For example, if rS=30, two words are stored.

EA must be a multiple of four; otherwise, the system alignment error handler is invoked.

Other registers altered:

- None

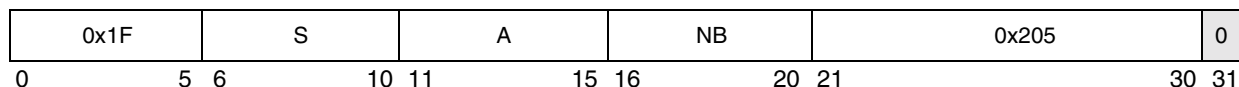
This instruction is defined by the PowerPC UISA.

**stswi**

Store String Word Immediate

**stswi**

Load/Store Unit

**stswi**                    **rS,rA,NB** Reserved


```

if rA = 0 then EA ← 0
else EA ← (rA)
if NB = 0 then n ← 32
else n ← NB
r ← rS-1
i ← 0
do while n > 0
    if i = 0 then r ← r+1 (mod 32)
    MEM(EA, 1) ← GPR(r)[i:i+7]
    i ← i+8
    if i = 32 then i ← 0
    EA ← EA+1
    n ← n-1

```

EA is (rA|0). Let  $n = NB$  if  $NB \neq 0$ ,  $n = 32$  if  $NB=0$ ;  $n$  is the number of bytes to store. Let  $nr = \text{CEIL}(n/4)$ :  $nr$  is the number of registers to supply data.

$n$  consecutive bytes starting at EA are stored from GPRs rS through rS+nr-1.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR0 if required.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stswx

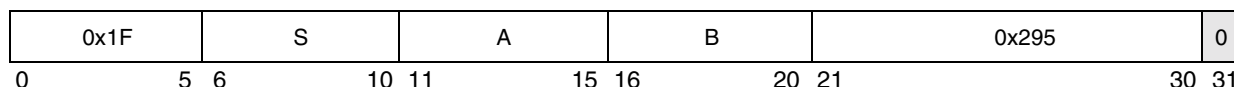
Store String Word Indexed

# stswx

Load/Store Unit

**stswx**                      rS,rA,rB

Reserved



```

if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
n ← XER[25:31]
r ← rS - 1
i ← 0
do while n > 0
    if i = 0 then r ← r + 1 (mod 32)
    MEM(EA, 1) ← GPR(r)[i:i+7]
    i ← i + 8
    if i = 32 then i ← 0
    EA ← EA + 1
    n ← n - 1
    
```

EA is the sum (rA|0)+(rB). Let  $n = \text{XER}[25:31]$ ;  $n$  is the number of bytes to store.

Let  $nr = \text{CEIL}(n/4)$ ;  $nr$  is the number of registers to supply data.

$n$  consecutive bytes starting at EA are stored from GPRs rS through rS+nr-1. If  $n = 0$ , no bytes are stored.

Bytes are stored left to right from each register. The sequence of registers wraps around to GPR0 if required.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

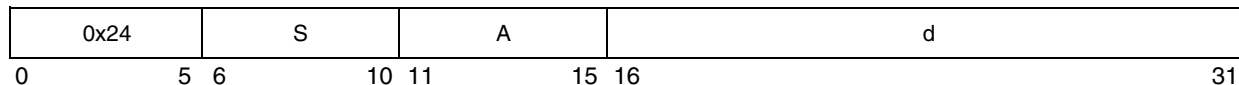
# stw

Store Word

# stw

Load/Store Unit

**stw**                      **rS,d(rA)**



if  $rA = 0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + \text{EXTS}(d)$   
 $\text{MEM}(EA, 4) \leftarrow rS$

EA is the sum  $(rA|0)+d$ .

The contents of **rS** are stored into the word in memory addressed by EA.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stwbrx

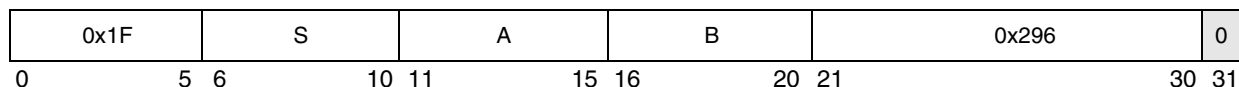
Store Word Byte-Reverse Indexed

# stwbrx

Load/Store Unit

**stwbrx**                      **rS,rA,rB**

Reserved



if  $rA = 0$  then  $b \leftarrow 0$   
 else  $b \leftarrow (rA)$   
 $EA \leftarrow b + (rB)$   
 $MEM(EA, 4) \leftarrow rS[24:31] \parallel rS[16:23] \parallel rS[8:15] \parallel rS[0:7]$

EA is the sum  $(rA|0)+(rB)$ .

The contents of  $rS[24:31]$  are stored into bits 0:7 of the word in memory addressed by EA. Bits  $rS[16:23]$  are stored into bits 8:15 of the word in memory addressed by EA. Bits  $rS[8:15]$  are stored into bits 16:23 of the word in memory addressed by EA. Bits  $rS[0:7]$  are stored into bits 24:31 of the word in memory addressed by EA.

Other registers altered:

- None

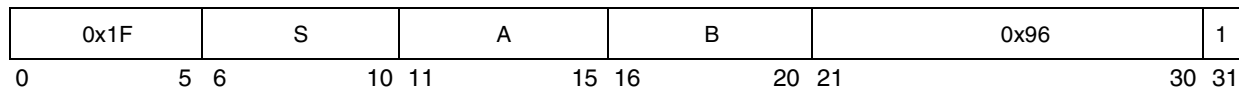
This instruction is defined by the PowerPC UISA.

**stwcx.**

Store Word Conditional Indexed

**stwcx.**

Load/Store Unit

**stwcx.**                    **rS,rA,rB**

```

if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
if RESERVE then
    MEM(EA, 4) ← rS
    RESERVE ← 0
    CR0 ← 0b00 || 0b1 || XER[SO]
else
    CR0 ← 0b00 || 0b0 || XER[SO]

```

EA is the sum  $(rA|0) + (rB)$ .

If a reservation exists, the contents of **rS** are stored into the word in memory addressed by EA and the reservation is cleared. If no reservation exists, the instruction completes without altering memory.

CR0 Field is set to reflect whether the store operation was performed (i.e., whether a reservation existed when the **stwcx.** instruction commenced execution) as follows.

$CR0[LT\ GT\ EQ\ SO] \leftarrow 0b00 \ ||\ store\_performed \ ||\ XER[SO]$

The EQ bit in the condition register field CR0 is modified to reflect whether the store operation was performed (i.e., whether a reservation existed when the **stwcx.** instruction began execution). If the store was completed successfully, the EQ bit is set to one.

EA must be a multiple of four; otherwise, the system alignment error handler is invoked.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO

This instruction is defined by the PowerPC UISA.

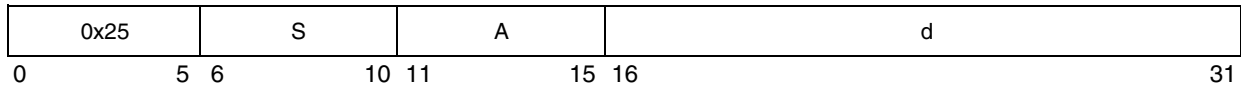
# stwu

Store Word with Update

# stwu

Load/Store Unit

**stwu**                      **rS,d(rA)**



EA ← (rA) + EXT5(d)  
 MEM(EA, 4) ← rS  
 rA ← EA

EA is the sum (rA|0)+d.  
 The contents of rS are stored into the word in memory addressed by EA.  
 EA is placed into rA.  
 If rA=0, the instruction form is invalid.  
 Other registers altered:

- None

This instruction is defined by the PowerPC UISA.



# stwux

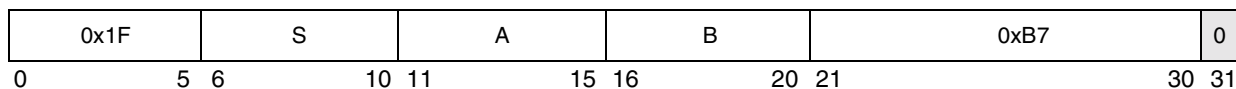
Store Word with Update Indexed

# stwux

Load/Store Unit

**stwux**                      **rS,rA,rB**

☐ Reserved



$EA \leftarrow (rA) + (rB)$   
 $MEM(EA, 4) \leftarrow rS$   
 $rA \leftarrow EA$

EA is the sum  $(rA|0)+(rB)$ .

The contents of **rS** are stored into the word in memory addressed by EA.

EA is placed into **rA**.

If **rA**=0, the instruction form is invalid.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# stwx

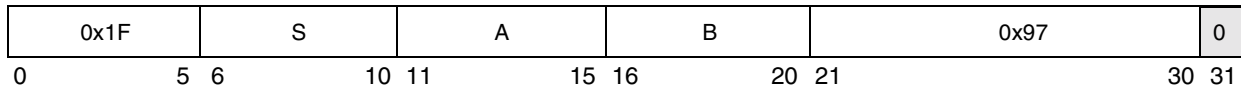
Store Word Indexed

# stwx

Load/Store Unit

**stwx**                      **rS,rA,rB**

Reserved



if **rA** = 0 then  $b \leftarrow 0$   
 else  $b \leftarrow (\mathbf{rA})$   
 $EA \leftarrow b + (\mathbf{rB})$   
 $MEM(EA, 4) \leftarrow \mathbf{rS}$

EA is the sum  $(\mathbf{rA}|0) + (\mathbf{rB})$ . The contents of **rS** are stored into the word in memory addressed by EA.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

# subf<sub>x</sub>

Subtract from

# subf<sub>x</sub>

Integer Unit

**subf**                    rD,rA,rB            (OE=0 Rc=0)  
**subf.**                  rD,rA,rB            (OE=0 Rc=1)  
**subfo**                  rD,rA,rB            (OE=1 Rc=0)  
**subfo.**                  rD,rA,rB            (OE=1 Rc=1)

0x1F	D	A	B	OE	0x28	Rc
0	5 6	10 11	15 16	20 21 22		30 31

$$rD \leftarrow \neg (rA) + (rB) + 1$$

The sum  $\neg (rA) + (rB) + 1$  is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

**Table 9-30 Simplified Mnemonics for subf Instruction**

Operation	Simplified Mnemonic	Equivalent To
Subtract	<b>sub</b> rD,rA,rB <b>sub.</b> rD,rA,rB <b>subo</b> rD,rA,rB <b>subo.</b> rD,rA,rB	<b>subf</b> rD,rB,rA <b>subf.</b> rD,rB,rA <b>subfo</b> rD,rB,rA <b>subfo.</b> rD,rB,rA

# subfc<sub>x</sub>

Subtract from Carrying

# subfc<sub>x</sub>

Integer Unit

**subfc**                      rD,rA,rB                      (OE=0 Rc=0)  
**subfc.**                     rD,rA,rB                     (OE=0 Rc=1)  
**subfco**                    rD,rA,rB                    (OE=1 Rc=0)  
**subfco.**                   rD,rA,rB                   (OE=1 Rc=1)

0x1F	D	A	B	OE	0x08	Rc
0	5 6	10 11	15 16	20 21 22		30 31

$$rD \leftarrow \neg (rA) + (rB) + 1$$

The sum  $\neg (rA) + (rB) + 1$  is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: CA  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

**Table 9-31 Simplified Mnemonics for subfc Instruction**

Operation	Simplified Mnemonic	Equivalent To
Subtract	<b>subc</b> rD,rA,rB <b>subc.</b> rD,rA,rB <b>subco</b> rD,rA,rB <b>subco.</b> rD,rA,rB	<b>subfc</b> rD,rB,rA <b>subfc.</b> rD,rB,rA <b>subfco</b> rD,rB,rA <b>subfco.</b> rD,rB,rA

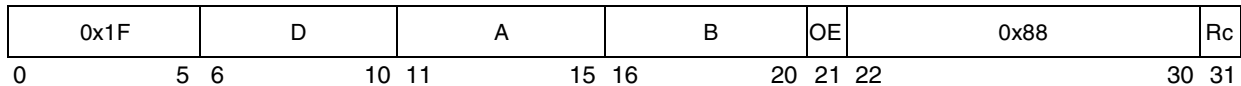
subfe<sub>x</sub>

Subtract from Extended

subfe	rD,rA,rB	(OE=0 Rc=0)
subfe.	rD,rA,rB	(OE=0 Rc=1)
subfeo	rD,rA,rB	(OE=1 Rc=0)
subfeo.	rD,rA,rB	(OE=1 Rc=1)

subfe<sub>x</sub>

Integer Unit



$rD \leftarrow \neg (rA) + (rB) + XER[CA]$

The sum  $\neg (rA)+(rB)+XER[CA]$  is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: CA  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

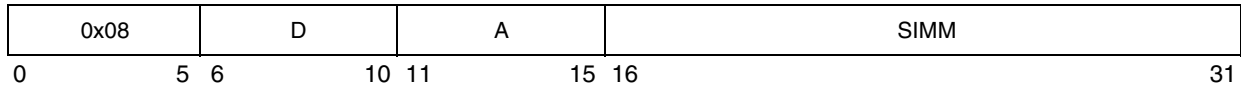
# subfic

Subtract from Immediate Carrying

# subfic

Integer Unit

**subfic**            rD,rA,SIMM



$rD \leftarrow \neg (rA) + \text{EXTS}(\text{SIMM}) + 1$   
 The sum  $\neg (rA)+\text{EXTS}(\text{SIMM})+1$  is placed into rD.  
 Other registers altered:  
 • XER:  
     Affected: CA

This instruction is defined by the PowerPC UISA.

# subfme<sub>x</sub>

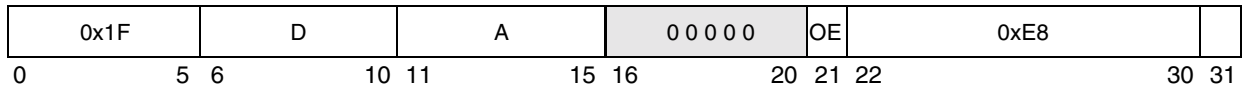
Subtract from Minus One Extended

# subfme<sub>x</sub>

Integer Unit

<b>subfme</b>	<b>rD,rA</b>	(OE=0 Rc=0)
<b>subfme.</b>	<b>rD,rA</b>	(OE=0 Rc=1)
<b>subfmeo</b>	<b>rD,rA</b>	(OE=1 Rc=0)
<b>subfmeo.</b>	<b>rD,rA</b>	(OE=1 Rc=1)

Reserved



$$rD \leftarrow \neg (rA) + XER[CA] - 1$$

The sum  $\neg (rA) + XER[CA] + 0xFFFF\_FFFF$  is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: CA  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.

# subfze<sub>x</sub>

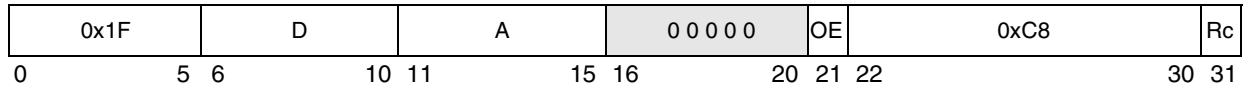
Subtract from Zero Extended

# subfze<sub>x</sub>

Integer Unit

<b>subfze</b>	<b>rD,rA</b>	(OE=0 Rc=0)
<b>subfze.</b>	<b>rD,rA</b>	(OE=0 Rc=1)
<b>subfzeo</b>	<b>rD,rA</b>	(OE=1 Rc=0)
<b>subfzeo.</b>	<b>rD,rA</b>	(OE=1 Rc=1)

Reserved



$$rD \leftarrow \neg (rA) + XER[CA]$$

The sum  $\neg (rA)+XER[CA]$  is placed into rD.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO (if Rc=1)
- XER:  
Affected: CA  
Affected: SO, OV (if OE=1)

This instruction is defined by the PowerPC UISA.



# sync

Synchronize

# sync

Load/Store Unit

Reserved

0x1F	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0x256	0
0	5 6	10 11	15 16	20 21	30 31

The **sync** instruction provides an ordering function for the effects of all instructions executed by a given processor. Executing a **sync** instruction ensures that all instructions previously initiated by the given processor appear to have completed before any subsequent instructions are initiated by the given processor. When the **sync** instruction completes, all external accesses initiated by the given processor prior to the **sync** will have been performed with respect to all other mechanisms that access memory.

The **sync** instruction can be used to ensure that the results of all stores into a data structure, performed in a “critical section” of a program, are seen by other processors before the data structure is seen as unlocked.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

**tw**

Trap Word

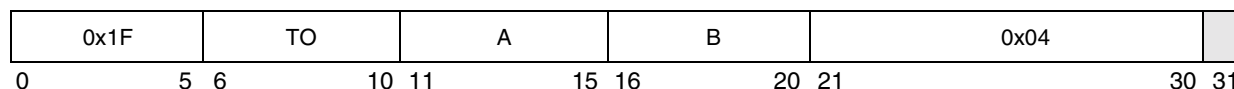
**tw**

Integer Unit

**tw**

TO,rA,rB

 Reserved



```

a ← (rA)
b ← (rB)
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <U b) & TO[3] then TRAP
if (a >U b) & TO[4] then TRAP
    
```

The contents of **rA** are compared with the contents of **rB**. If any bit in the **TO** field is set to one and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

**Table 9-32 Simplified Mnemonics for tw Instruction**

Operation	Operands	Equivalent To
Trap unconditionally	<b>trap</b>	<b>tw 31,0,0</b>
<b>Trap if equal</b>	<b>tw eq rA,rB</b>	<b>tw 4,rA,rB</b>
<b>Trap if greater than or equal to</b>	<b>tw ge rA,rB</b>	<b>tw 12,rA,rB</b>
Trap if greater than	<b>tw gt rA,rB</b>	<b>tw 8,rA,rB</b>
Trap if less than or equal to	<b>tw le rA,rB</b>	<b>tw 20,rA,rB</b>
Trap if logically greater than or equal to	<b>tw lge rA,rB</b>	<b>tw 5,rA,rB</b>
Trap if logically greater than	<b>tw lgt rA,rB</b>	<b>tw 1,rA,rB</b>
Trap if logically less than or equal to	<b>tw lle rA,rB</b>	<b>tw 6,rA,rB</b>
Trap if logically less than	<b>tw llt rA,rB</b>	<b>tw 2,rA,rB</b>
Trap if logically not greater than	<b>tw lng rA,rB</b>	<b>tw 6,rA,rB</b>
Trap if logically not less than	<b>tw lnl rA,rB</b>	<b>tw 5,rA,rB</b>
Trap if less than	<b>tw lt rA,rB</b>	<b>tw 16,rA,rB</b>
Trap if not equal to	<b>tw ne rA,rB</b>	<b>tw 24,rA,rB</b>
Trap if not greater than	<b>tw ng rA,rB</b>	<b>tw 20,rA,rB</b>
Trap if not less than	<b>tw nl rA,rB</b>	<b>tw 12,rA,rB</b>

**twi**

Trap Word Immediate

**twi**

Integer Unit

**twi**

TO,rA,SIMM

0x03	TO	A	SIMM
0	5 6	10 11	15 16 31

```

a ← (rA)
if (a < EXTS(SIMM)) & TO[0] then TRAP
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP

```

The contents of **rA** are compared with the sign-extended **SIMM** field. If any bit in the **TO** field is set to one and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

**Table 9-33 Simplified Mnemonics for twi Instruction**

Operation	Operands	Equivalent To
Trap if equal	<b>tweqi</b> rA,value	<b>twi 4</b> ,rA,value
Trap if greater than or equal to	<b>twgei</b> rA,value	<b>twi 12</b> ,rA,value
Trap if greater than	<b>twgti</b> rA,value	<b>twi 8</b> ,rA,value
Trap if less than or equal to	<b>twlei</b> rA,value	<b>twi 20</b> ,rA,value
Trap if logically greater than or equal to	<b>twlgei</b> rA,value	<b>twi 5</b> ,rA,value
Trap if logically greater than	<b>twlgti</b> rA,value	<b>twi 1</b> ,rA,value
Trap if logically less than or equal to	<b>twllei</b> rA,value	<b>twi 6</b> ,rA,value
Trap if logically less than	<b>twlhti</b> rA,value	<b>twi 2</b> ,rA,value
Trap if logically not greater than	<b>twlngi</b> rA,value	<b>twi 6</b> ,rA,value
Trap if logically not less than	<b>twlnli</b> rA,value	<b>twi 5</b> ,rA,value
Trap if less than	<b>twlti</b> rA,value	<b>twi 16</b> ,rA,value
Trap if not equal to	<b>twnei</b> rA,value	<b>twi 24</b> ,rA,value
Trap if not greater than	<b>twngi</b> rA,value	<b>twi 20</b> ,rA,value
Trap if not less than	<b>twnli</b> rA,value	<b>twi 12</b> ,rA,value

# **Xor<sub>x</sub>**

XOR

# **Xor<sub>x</sub>**

Integer Unit

**xor**                      **rA,rS,rB**                      (**Rc=0**)  
**xor.**                      **rA,rS,rB**                      (**Rc=1**)

0x1F				S				A				B				0x13C												Rc	
0				5	6			10	11			15	16			20	21											30	31

$$rA \leftarrow (rS) \oplus (rB)$$

The contents of **rA** is XORed with the contents of **rB** and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 Field):  
Affected: LT, GT, EQ, SO                      (if Rc=1)

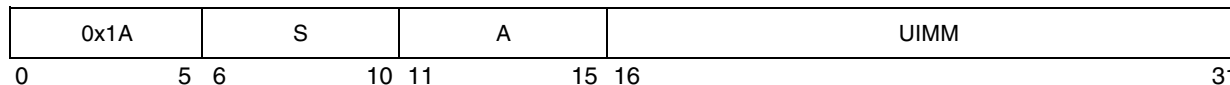
This instruction is defined by the PowerPC UISA.

**xori**

XOR Immediate

**xori**

Integer Unit

**xori**      **rA,rS,UIMM**

$$rA \leftarrow (rS) \oplus ((16)0 \parallel UIMM)$$

The contents of **rS** is XORed with 0x0000 || UIMM and the result is placed into **rA**.

Other registers altered:

- None

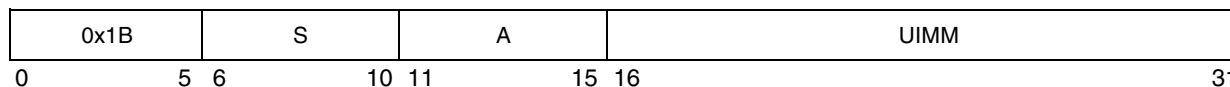
This instruction is defined by the PowerPC UISA.

**xoris**

XOR Immediate Shifted

**xoris**

Integer Unit

**xoris**      **rA,rS,UIMM**

$$rA \leftarrow (rS) \oplus (UIMM \parallel (16)0)$$

The contents of **rS** is XORed with UIMM || 0x0000 and the result is placed into **rA**.

Other registers altered:

- None

This instruction is defined by the PowerPC UISA.

## APPENDIX A INSTRUCTION SET LISTINGS

This appendix lists the instruction set implemented in the RCPU, sorted by mnemonic. Reserved bits are shaded.

**Table A-1 Complete Instruction List Sorted by Mnemonic**

Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
<b>addx</b>	0x1F				D				A				B				OE		0x10A						Rc							
<b>addcx</b>	0x1F				D				A				B				OE		0xA						Rc							
<b>addex</b>	0x1F				D				A				B				OE		0x8A						Rc							
<b>addi</b>	0x0E				D				A				SIMM																			
<b>addic</b>	0x0C				D				A				SIMM																			
<b>addic.</b>	0x0D				D				A				SIMM																			
<b>addis</b>	0x0F				D				A				SIMM																			
<b>addmex</b>	0x1F				D				A				0 0 0 0 0				OE		0xEA						Rc							
<b>addzex</b>	0x1F				D				A				0 0 0 0 0				OE		0xCA						Rc							
<b>andx</b>	0x1F				S				A				B				0x1C						Rc									
<b>andcx</b>	0x1F				S				A				B				0x3C						Rc									
<b>andi.</b>	0x1C				S				A				UIMM																			
<b>andis.</b>	0x1D				S				A				UIMM																			
<b>bx</b>	0x12				LI																										AA	LK
<b>bcx</b>	0x10				BO				BI				BD												AA	LK						
<b>bcctrx</b>	0x13				BO				BI				0 0 0 0 0				0x210						LK									
<b>bclrx</b>	0x13				BO				BI				0 0 0 0 0				0x10						LK									
<b>cmp</b>	0x1F				crfD		0	L	A				B				0 0 0 0 0 0 0 0 0 0												0			
<b>cmpi</b>	0x0B				crfD		0	L	A				SIMM																			
<b>cmpl</b>	0x1F				crfD		0	L	A				B				0x20						0									
<b>cmpli</b>	0x0A				crfD		0	L	A				UIMM																			
<b>cntlzwx</b>	0x1F				S				A				0 0 0 0 0				0x1A						Rc									
<b>crand</b>	0x13				crbD				crbA				crbB				0x101						0									
<b>crandc</b>	0x13				crbD				crbA				crbB				0x81						0									

# Freescal Semiconductor, Inc.

Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
<b>creqv</b>	0x13							crbD					crbA				crbB									0x121						0
<b>crnand</b>	0x13							crbD					crbA				crbB									0xE1						0
<b>crnor</b>	0x13							crbD					crbA				crbB									0x13						0
<b>cror</b>	0x13							crbD					crbA				crbB									0x1C1						0
<b>crorc</b>	0x13							crbD					crbA				crbB									0x1A1						0
<b>crxor</b>	0x13							crbD					crbA				crbB									0xC1						0
<b>divwx</b>	0x1F							D					A				B				OE					0x1EB						Rc
<b>divwux</b>	0x1F							D					A				B				OE					0x1CB						Rc
<b>eieio</b>	0x1F							0 0 0 0 0					0 0 0 0 0				0 0 0 0 0									0x356						0
<b>eqvx</b>	0x1F							S					A				B									0x11C						Rc
<b>extsbx</b>	0x1F							S					A				0 0 0 0 0									0x3BA						Rc
<b>extshx</b>	0x1F							S					A				0 0 0 0 0									0x39A						Rc
<b>fabsx</b>	0x3F							D					0 0 0 0 0				B									0x108						Rc
<b>faddx</b>	0x3F							D					A				B								0 0 0 0 0		0x15					Rc
<b>faddsx</b>	0x3B							D					A				B								0 0 0 0 0		0x15					Rc
<b>fcmpo</b>	0x3F							crfD		0 0			A				B									0x20						0
<b>fcmpu</b>	0x3F							crfD		0 0			A				B									0 0 0 0 0 0 0 0 0 0						0
<b>fctiw</b>	0x3F							D					0 0 0 0 0				B									0x0E						Rc
<b>fctiwz</b>	0x3F							D					0 0 0 0 0				B									0x0F						Rc
<b>fdivx</b>	0x3F							D					A				B								0 0 0 0 0		0x12					Rc
<b>fdivsx</b>	0x3B							D					A				B								0 0 0 0 0		0x12					Rc
<b>fmaddx</b>	0x3F							D					A				B								C		0x1D					Rc
<b>fmaddsx</b>	0x3B							D					A				B								C		0x1D					Rc
<b>fmr</b>	0x3F							D					0 0 0 0 0				B									0x48						Rc
<b>fmsubx</b>	0x3F							D					A				B								C		0x1C					Rc
<b>fmsubsx</b>	0x3B							D					A				B								C		0x1C					Rc
<b>fmulx</b>	0x3F							D					A				0 0 0 0 0								C		0x19					Rc
<b>fmulsx</b>	0x3B							D					A				0 0 0 0 0								C		0x19					Rc
<b>fnabsx</b>	0x3F							D					0 0 0 0 0				B									0x88						Rc
<b>fnegx</b>	0x3F							D					0 0 0 0 0				B									0x28						Rc
<b>fnmaddx</b>	0x3F							D					A				B								C		0x1F					Rc
<b>fnmaddsx</b>	0x3B							D					A				B								C		0x1F					Rc
<b>fnmsubx</b>	0x3F							D					A				B								C		0x1E					Rc



# Freescale Semiconductor, Inc.

Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
<b>fnmsubsx</b>	0x3B							D					A				B					C				0x1E						Rc
<b>frsp<sub>x</sub></b>	0x3F							D				0 0 0 0 0					B					0x0C										Rc
<b>fsub<sub>x</sub></b>	0x3F							D					A				B					0 0 0 0 0			0x14							Rc
<b>fsubsx</b>	0x3B							D					A				B					0 0 0 0 0			0x14							Rc
<b>icbi</b>	0x1F						0 0 0 0 0						A				B					0x3D6										0
<b>isync</b>	0x13						0 0 0 0 0					0 0 0 0 0				0 0 0 0 0						0x90										0
<b>lbz</b>	0x22							D					A									d										
<b>lbzu</b>	0x23							D					A									d										
<b>lbzux</b>	0x1F							D					A				B					0x77										0
<b>lbzx</b>	0x1F							D					A				B					0x57										0
<b>lfd</b>	0x32							D					A									d										
<b>lfd<sub>u</sub></b>	0x33							D					A									d										
<b>lfd<sub>ux</sub></b>	0x1F							D					A				B					0x277										0
<b>lfd<sub>x</sub></b>	0x1F							D					A				B					0x257										0
<b>lfs</b>	0x30							D					A									d										
<b>lfs<sub>u</sub></b>	0x31							D					A									d										
<b>lfs<sub>ux</sub></b>	0x1F							D					A				B					0x237										0
<b>lfs<sub>x</sub></b>	0x1F							D					A				B					0x217										0
<b>lha</b>	0x2A							D					A									d										
<b>lh<sub>u</sub></b>	0x2B							D					A									d										
<b>lh<sub>aux</sub></b>	0x1F							D					A				B					0x177										0
<b>lh<sub>ax</sub></b>	0x1F							D					A				B					0x157										0
<b>lhbr<sub>x</sub></b>	0x1F							D					A				B					0x316										0
<b>lhz</b>	0x28							D					A									d										
<b>lhzu</b>	0x29							D					A									d										
<b>lhz<sub>ux</sub></b>	0x1F							D					A				B					0x137										0
<b>lhz<sub>x</sub></b>	0x1F							D					A				B					0x117										0
<b>lmw</b>	0x2E							D					A									d										
<b>lsw<sub>i</sub></b>	31							D					A				NB					0x255										0
<b>lsw<sub>x</sub></b>	31							D					A				B					0x215										0
<b>lwar<sub>x</sub></b>	31							D					A				B					0x14										0
<b>lwbr<sub>x</sub></b>	31							D					A				B					0x216										0
<b>lwz</b>	32							D					A									d										

# Freescale Semiconductor, Inc.

Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lwzu	33				D				A				d																			
lwzux	31				D				A				B				0x37								0							
lwzx	31				D				A				B				0x17								0							
mcrf	19				crfD		0 0		crfS		0 0		0 0 0 0 0				0 0 0 0 0 0 0 0 0 0												0			
mcrfs	63				crfD		0 0		crfS		0 0		0 0 0 0 0				0x40								0							
mcrxr	31				crfS		0 0		0 0 0 0 0				0 0 0 0 0				0x200								0							
mfcrr	31				D				0 0 0 0 0				0 0 0 0 0				0x13								0							
mffsx	63				D				0 0 0 0 0				0 0 0 0 0				0x247								Rc							
mfmsr	31				D				0 0 0 0 0				0 0 0 0 0				0x53								0							
mfspr	31				D				SPR												0x153								0			
mftb	31				D				TBR												0x173								0			
mtcrf	31				S				0	CRM								0	0x90								0					
mtfsb0x	63				crbD				0 0 0 0 0				0 0 0 0 0				0x46								Rc							
mtfsb1x	63				crbD				0 0 0 0 0				0 0 0 0 0				0x26								Rc							
mtfsfx	31				0	FM								0	frB				0x2C7								Rc					
mtfsfix	63				crbD		0 0		0 0 0 0 0				IMM				0	0x86								Rc						
mtmsr	31				S				0 0 0 0 0				0 0 0 0 0				0x92								0							
mtspr	31				D				SPR												0x1D3								0			
mulhw	0x1F				D				A				B				0	0x4B								Rc						
mulhwu	0x1F				D				A				B				0	0x0B								Rc						
mullw	0x1F				D				A				B				OE	0xEB								Rc						
mulli	0x07				D				A				SIMM																			
nand	0x1F				S				A				B				0x1DC								Rc							
neg	0x1F				D				A				0 0 0 0 0				OE	0x68								Rc						
nor	0x1F				S				A				B				0x7C								Rc							
or	0x1F				S				A				B				0x1BC								Rc							
orc	0x1F				S				A				B				0x19C								Rc							
ori	0x18				S				A				UIMM																			
oris	0x19				S				A				UIMM																			
rfi	0x13				0 0 0 0 0				0 0 0 0 0				0 0 0 0 0				0x32								0							
rlwim	0x14				S				A				SH				MB				ME				Rc							
rlwinm	0x15				S				A				SH				MB				ME				Rc							
rlwnm	0x17				S				A				B				MB				ME				Rc							

# Freescale Semiconductor, Inc.

Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
<b>sc</b>	0x11						0 0 0 0 0					0 0 0 0 0																			1	0
<b>slwx</b>	0x1F						S					A					B															Rc
<b>srawx</b>	0x1F						S					A					B															Rc
<b>srawix</b>	0x1F						S					A					SH															Rc
<b>srwx</b>	0x1F						S					A					B															Rc
<b>stb</b>	0x26						S					A																				
<b>stbu</b>	0x27						S					A																				
<b>stbux</b>	0x1F						S					A					B															0
<b>stbx</b>	0x1F						S					A					B															0
<b>stfd</b>	0x36						frS					A																				
<b>stfdu</b>	0x37						frS					A																				
<b>stfdx</b>	0x1F						frS					A					B															0
<b>stfdx</b>	0x1F						frS					A					B															0
<b>stfiwx</b>	0x1F						frS					A					B															0
<b>stfs</b>	0x34						frS					A																				
<b>stfsu</b>	0x35						frS					A																				
<b>stfsux</b>	0x1F						frS					A					B															0
<b>stfsx</b>	0x1F						frS					A					B															0
<b>sth</b>	0x2C						S					A																				
<b>sthbrx</b>	0x1F						S					A					B															0
<b>sthu</b>	0x2D						S					A																				
<b>sthux</b>	0x1F						S					A					B															0
<b>sthx</b>	0x1F						S					A					B															0
<b>stmw</b>	0x2F						S					A																				
<b>stswi</b>	0x1F						S					A					NB															0
<b>stswx</b>	0x1F						S					A					B															0
<b>stw</b>	0x24						S					A																				
<b>stwbrx</b>	0x1F						S					A					B															0
<b>stwcx.</b>	31						S					A					B															1
<b>stwu</b>	0x25						S					A																				
<b>stwux</b>	0x1F						S					A					B															0
<b>stwx</b>	0x1F						S					A					B															0
<b>subfx</b>	0x1F						D					A					B					OE										Rc

# Freescale Semiconductor, Inc.

Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
<b>subfcx</b>	0x1F							D					A				B				OE				0x08							Rc
<b>subfex</b>	0x1F							D					A				B				OE				0x88							Rc
<b>subfic</b>	0x08							D					A				SIMM															
<b>subfmex</b>	0x1F							D					A				0 0 0 0 0				OE				0xE8							Rc
<b>subfzex</b>	0x1F							D					A				0 0 0 0 0				OE				0xC8							Rc
<b>sync</b>	0x1F						0 0 0 0 0					0 0 0 0 0					0 0 0 0 0								0x256							0
<b>tw</b>	0x1F							TO					A				B								0x04							0
<b>twi</b>	0x03							TO					A				SIMM															
<b>xorx</b>	0x1F							S					A				B								0x13C							Rc
<b>xori</b>	0x1A							S					A				UIMM															
<b>xoris</b>	0x1B							S					A				UIMM															

## APPENDIX B

### MULTIPLE-PRECISION SHIFTS

This appendix gives examples of how multiple precision shifts can be programmed. A multiple-precision shift is initially defined to be a shift of an  $n$ -word quantity, where  $n > 1$ . The quantity to be shifted is contained in  $n$  registers. The shift amount is specified either by an immediate value in the instruction or by bits 27 to 31 of a register.

The examples shown below distinguish between the cases  $n = 2$  and  $n > 2$ . If  $n = 2$ , the shift amount may be in the range 0 to 63, which are the maximum ranges supported by the shift instructions used. However if  $n > 2$ , the shift amount must be in the range 0 to 31, for the examples to yield the desired result. The specific instance shown for  $n > 2$  is  $n = 3$ : extending those instruction sequences to larger  $n$  is straightforward, as is reducing them to the case  $n = 2$  when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts only the case  $n = 3$  is shown, because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers. In all cases, for both input and result, the lowest-numbered register contains the highest-order part of the data and highest-numbered register contains the lowest-order part. For non-immediate shifts, the shift amount is assumed to be in bits 27 to 31 (32-bit mode) of GPR6. For immediate shifts, the shift amount is assumed to be greater than zero. GPRs 0 to 31 are used as scratch registers. For  $n > 2$ , the number of instructions required is  $2N-1$  (immediate shifts) or  $3N-1$  (non-immediate shifts).

In the following examples, let  $n$  be the number of words to be shifted.

#### Shift Left Immediate, $n = 3$ (Shift Amount $< 32$ )

```
rlwinm    r2,r2,SH,0,31-SH
rlwimi    r2,r3,SH,32-SH,31
rlwinm    r3,r3,SH,0,31-SH
rlwimi    r3,r4,SH,32-SH,31
rlwinm    r4,r4,SH,0,31-SH
```

#### Shift Left, $n = 2$ (Shift Amount $< 64$ )

```
subfic    r31,r6,32
slw       r2,r2,r6
srw       r0,r3,r31
or        r2,r2,r0
addic     r31,r6,r6
slw       r0,r3,r31
or        r2,r2,r0
slw       r3,r3,r6
```

## Shift Left, $n = 3$ (Shift Amount < 32)

```
subfic    r31,r6,32
slw       r2,r2,r6
srw       r0,r3,r31
or        r2,r2,r0
slw       r3,r3,6
srw       r0,r4,r31
or        r3,r3,r0
slw       r4,r4,r6
```

## Shift Right Immediate, $n = 3$ (Shift Amount < 32)

```
rlwinm    r4,r4,32-SH,SH,31
rlwimi    r4,r3,32-SH,0,SH-1
rlwinm    r3,r3,32-SH,SH,31
rlwimi    r3,r2,32-SH,0,SH-1
rlwinm    r2,r2,32-SH,SH,31
```

## Shift Right, $n = 2$ (Shift Amount < 64)

```
subfic    r31,r6,32
srw       r3,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
addic     r31,r6,-32
srw       r0,r2,r31
or        r3,r3,r0
srw       r2,r2,r6
```

## Shift Right, $n = 3$ (Shift Amount < 32)

```
subfic    r31,r6,32
srw       r4,r4,r6
slw       r0,r2,r31
or        r4,r4,r0
srw       r31,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
srw       r2,r2,r6
```

## Shift Right Algebraic Immediate, $n = 3$ (Shift Amount < 32)

```
rlwinm    r4,r4,32-SH,SH,31
rlwimi    r4,r3,32-SH,0,SH-1
rlwinm    r3,r3,32-SH,SH,31
rlwimi    r3,r2,32-SH,0,SH-1
srawi     r2,r2,SH
```

## Shift Right Algebraic, $n = 2$ (Shift Amount < 64)

```
subfic    r31,r6,32
srw       r3,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
addic     r31,r6,-32
sraw      r0,r2,r31
ble       $+8
ori       r3,r0,0
sraw      r2,r2,r6
```

## Shift Right Algebraic, $n = 3$ (Shift Amount < 32)

```
subfic    r31,r6,32
srw       r4,r4,r6
slw       r0,r3,r31
or        r4,r4,r0
srw       r3,r3,r6
slw       r0,r2,r31
or        r3,r3,r0
sraw      r2,r2,r6
```

## APPENDIX C

### FLOATING-POINT MODELS AND CONVERSIONS

This appendix gives examples of how the floating-point conversion instructions can be used to perform various conversions.

#### C.1 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full convert to signed fixed-point integer word function can be implemented with the sequence shown below, assuming that the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement "disp" from the address in GPR1 can be used as scratch space.

```
fctiw[z]    f2,f1           #convert to fx int
stfd        f2,disp(r1)     #store float
lwz         r3,disp+4(r1)    #load word and zero
```

#### C.2 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

The full convert to unsigned fixed-point integer word function can be implemented with the sequence shown below, assuming that the floating-point value to be converted is in FPR1, the value 0 is in FPR0, the value  $2^{32}$  is in FPR3, the value 0x0000 0000 7FFF FFFF is in FPR4, the value  $2^{31}$  is in FPR5 and GPR5, the result is returned in GPR3, and a double word at displacement "disp" from the address in GPR1 can be used as scratch space.

```
fmr         f2,f0           #use 0 if < 0
fcmpu       cr2,f1,f0
bl          cr2,store
fmr         f2,f4           #use max if > max
fcmpu       cr2,f1,f3
bgt         cr2,store
fsub        f2,f1,f5         #subtract 2**31
fcmpu       cr2,f1,f5         #use diff if S 2**31
bnl         cr2,$+8
fmr         f2,f1
fctiw[z]    f2,f2           #convert to fx int store-
stfd        f2,disp(r1)     #store float
lwz         r3,disp+4(r1)    #load word
bl          cr2,$+8         #add 2**31 if input
add         r3,r3,r5         #was S 2**31
```

#### C.3 Floating-Point Models

This section describes models for floating-point instructions.

##### C.3.1 Floating-Point Round to Single-Precision Model

The following algorithm describes the operation of the floating-point round to single-precision (**frsp**) instruction.

# Freescale Semiconductor, Inc.

```

If FRB[1:11]<897 and FRB[1:63]>0 then
Do
    If FPSCR[UE]=0 then goto Disabled Exponent Underflow
    If FPSCR[UE]=1 then goto Enabled Exponent Underflow
End
If FRB[1:11]>1150 and FRB[1:11]<2047 then
Do
    If FPSCR[OE]=0 then goto Disabled Exponent Overflow
    If FPSCR[OE]=1 then goto Enabled Exponent Overflow
End
If FRB[1:11]>896 and FRB[1:11]<1151 then goto Normal Operand
If FRB[1:63]=0 then goto Zero Operand
If FRB[1:11]=2047 then
Do
    If FRB[12:63]=0 then goto Infinity Operand
    If FRB[12]=1 then goto QNaN Operand
    If FRB[12]=0 and FRB[13:63]>0 then goto SNaN Operand
End

```

## Disabled Exponent Underflow:

```

sign ← FRB0
If FRB[1:11]=0 then
Do
    exp ← -1022
    frac ← 0b0 || FRB[12:63]
End
If FRB[1:11]>0 then
Do
    exp ← FRB[1:11] - 1023
    frac ← 0b1 || FRB[12:63]
End
Denormalize operand:
G || R || X ← 0b000
Do while exp<-126
    exp ← exp + 1
    frac || G || R || X ← 0b0 || frac || G || (R | X)
End
FPSCR[UX] < frac[24:52] || G || R || X>0
If frac[24:52] || G || R || X>0 then FPSCR[XX] ← 1
Round single(sign,exp,frac,G,R,X)
If frac=0 then
Do
    FRT00 ← sign
    FRT0[1:63] ← 0
    If sign=0 then FPSCR[FPRF] ← "+zero"
    If sign=1 then FPSCR[FPRF] ← "-zero"
End
If frac>0 then
Do
    If frac[0]=1 then
Do
        If sign=0 then FPSCR[FPRF] ← "+normal number"
        If sign=1 then FPSCR[FPRF] ← "-normal number"
End
    If frac[0]=0 then
Do
        If sign=0 then FPSCR[FPRF] ← "+denormalized number"
        If sign=1 then FPSCR[FPRF] ← "-denormalized number"
End
    Normalize operand-
    Do while frac[0]=0
        exp ← exp-1
        frac || G || R ← frac[1:52] || G || R || 0b0
    End

```



```

        End
        FRT[0] ← sign
        FRT[1:11] ← exp + 1023
        FRT[12:63] ← frac[1:23] || 0b 0 0000 0000 0000 0000 0000 0000 0000
    End
Done

```

## Enabled Exponent Underflow

```

FPSCR[UX] ← 1
sign ← FRB[0]
If FRB[1:11]=0 then
    Do
        exp ← -1022
        frac ← 0b0 || FRB[12:63]
    End
    If FRB[1:11]>0 then
        Do
            exp ← FRB[1:11] - 1023
            frac ← 0b1 || FRB[12:63]
        End
    End
    Normalize operand-
    Do while frac[0]=0
        exp ← exp - 1
        frac ← frac[1:52] || 0b0
    End
    If frac[24:52]>0 then FPSCR[XX] ← 1
    Round single(sign,exp,frac,0,0,0)
    exp ← exp + 192
    FRT[0] ← sign
    FRT[1:11] ← exp + 1023
    FRT[12:63] ← frac[1:23] || 0b0 0000 0000 0000 0000 0000 0000 0000
    If sign=0 then FPSCR[FPRF] ← "+normal number"
    If sign=1 then FPSCR[FPRF] ← "-normal number"
Done

```

## Disabled Exponent Overflow

```

inc ← 0
FPSCR[OX] ← 1
FPSCR[XX] ← 1
If FPSCR[RN]= 0b00 then /* Round to Nearest */
    Do
        inc ← 0
        If FRB[0]=0 then FRT ← 0x7FF0 0000 0000 0000
        If FRB[0]=1 then FRT ← 0xFFFF 0000 0000 0000
        If FRB[0]=0 then FPSCR[FPRF] ← "+infinity"
        If FRB[0]=1 then FPSCR[FPRF] ← "-infinity"
    End
    If FPSCR[RN]= 0b01 then /* Round Truncate */
        Do
            If (0b0 || FRB[1:63]) < 0x047EF FFFF E000 0000 then inc ← 0
            If FRB[0]=0 then FRT ← 0x47EF FFFF E000 0000
            If FRB[0]=1 then FRT ← 0xC7EF FFFF E000 0000
            If FRB[0]=0 then FPSCR[FPRF] ← "+normal number"
            If FRB[0]=1 then FPSCR[FPRF] ← "-normal number"
        End
    End
    If FPSCR[RN]= 0b10 then /* Round to +Infinity */
        Do
            If FRB[0]=0 then inc ← 0
            If (FRB[0]=1 & (FRB > 0xC7EF FFFF E000 0000 then inc ← 1)
            If FRB[0]=0 then FRT ← 0x7FF0 0000 0000 0000
            If FRB[0]=1 then FRT ← 0xC7EF FFFF E000 0000
            If FRB[0]=0 then FPSCR[FPRF] ← "+infinity"
            If FRB[0]=1 then FPSCR[FPRF] ← "-normal number"
        End
    End
    If FPSCR[RN]= 0b11 then /* Round to -Infinity */

```

```

Do
  (If FRB[0]=0 & FRB < 0x47EF FFFF E000 0000) then inc ← 1
  If FRB[0]= 1 then inc ← 1
  If FRB[0]=0 then FRT ← 0x47EF FFFF E000 0000
  If FRB[0]=1 then FRT ← 0xFFFF0 0000 0000 0000
  If FRB[0]=0 then FPSCR[FPRF] ← "+normal number"
  If FRB[0]=1 then FPSCR[FPRF] ← "-infinity"
End
FPSCR[FR] ← inc
FPSCR[FI] ← 1
Done

```

### Enabled Exponent Overflow

```

sign ← FRB[0]
exp ← FRB[1:11] - 1023
frac ← 0b1 || [12:63]
If frac[24:52]>0 then FPSCR[XX] ← 1
Round single(sign,exp,frac,0,0,0)
Enabled Overflow
FPSCR[OX] ← 1
exp ← exp - 192
FRT[0] ← sign
FRT[1:11] ← exp + 1023
FRT[12:63] ← frac[1:23] || 0b0 0000 0000 0000 0000 0000 0000 0000
If sign=0 then FPSCR[FPRF] ← "+normal number"
If sign=1 then FPSCR[FPRF] ← "-normal number"
Done

```

### Zero Operand

```

FRT ← FRB
If FRB[0]=0 then FPSCR[FPRF] ← "+zero"
If FRB[0]=1 then FPSCR[FPRF] ← "-zero"
FPSCR[FR FI] ← 0b00
Done

```

### Infinity Operand

```

FRT ← FRB
If FRB[0]=0 then FPSCR[FPRF] ← "+infinity"
If FRB[0]=1 then FPSCR[FPRF] ← "-infinity" Done
QNaN Operand-
FRT ← FRB[0:34] || 0b0 0000 0000 0000 0000 0000 0000 0000
FPSCR[FPRF] ← "QNaN"
FPSCR[FR FI] ← 0b00
Done

```

### QNaN Operand

```

FRT ← FRB[0:34] || 0b0 0000 0000 0000 0000 0000 0000 0000
FPSCR[FPRF] ← "QNaN"
FPSCR[FR FI] ← 0b00
Done

```

### SNaN Operand

```

FPSCR[VXSNAN] ← 1
If FPSCR[VE]=0 then
  Do
    FRT[0:11] ← FRB[0:11]
    FRT[12] ← 1
    FRT[13:63] ← FRB[13:34] || 0b0 0000 0000 0000 0000 0000 0000 0000
    FPSCR[FPRF] ← "QNaN"
  End
FPSCR[FR FI] ← 0b00
Done

```

## Normal Operand

```

sign ← FRB[0]
exp ← FRB[1:11] - 1023
frac ← 0b1 || FRB[12:63]
If frac[24:52] > 0 then FPSCR[XX] ← 1
Round single(sign, exp, frac, 0, 0, 0)
If exp > +127 and FPSCR[OE] = 0 then go to Disabled Exponent Overflow
If exp > +127 and FPSCR[OE] = 1 then go to Enabled Overflow
FRT[0] ← sign
FRT[1:11] ← exp + 1023
FRT[12:63] ← frac[1:23] || 0b0 0000 0000 0000 0000 0000 0000 0000
If sign = 0 then FPSCR[FPRF] ← "+normal number"
If sign = 1 then FPSCR[FPRF] ← "-normal number"
Done

```

## Round Single (sign, exp, frac, G, R, X)

```

inc ← 0
lsb ← frac[23]
gbit ← frac[24]
rbit ← frac[25]
xbit ← (frac[26:52] || G || R || X) | 0
If FPSCR[RN] = 0b00 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b11uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If FPSCR[RN] = 0b10 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b 0u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
    End
If FPSCR[RN] = 0b11 then
    Do
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
frac[0:23] ← frac[0:23] + inc
If carry_out = 1 then
    Do
        frac[0:23] ← 0b1 || frac[0:22]
        exp ← exp + 1
    End
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return

```

## C.3.2 Floating-Point Convert to Integer Model

The following algorithm describes the operation of the floating-point convert to integer instructions. In this example, u represents an undefined hexadecimal digit.

```

If Floating Convert to Integer Word
Then Do
    Then round_mode ← FPSCR[RN]
    tgt_precision ← "32-bit integer"
End
If Floating Convert to Integer Word with round toward Zero
Then Do
    round_mode ← 0b01
    tgt_precision ← "32-bit integer"
End
If Floating Convert to Integer Doubleword

```

```

Then Do
    round_mode ← FPSCR[RN]
    tgt_precision ← "64-bit integer"
End
If Floating Convert to Integer Doubleword with round toward Zero
Then Do
    round_mode ← 0b01
    tgt_precision ← "64-bit integer"
End
If FRB[1:11]=2047 and FRB[12:63]=0 then goto Infinity Operand
If FRB[1:11]=2047 and FRB12=0 then goto SNaN Operand
If FRB[1:11]=2047 and FRB12=1 then goto QNaN Operand
If FRB[1:11]>1086 then goto Large Operand

sign ← FRB0
If FRB[1:11]>0 then exp ← FRB[1:11] - 1023 /* exp - bias */
If FRB[1:11]=0 then exp ← -1022
If FRB[1:11]>0 then frac[0:64]←0b01 ||FRB[12:63]||0b000000000000 /
*normal*/
If FRB[1:11]=0 then frac[0:64]←b'00' ||FRB[12:63]||0b000000000000 /
*denormal*/

gbit || rbit || xbit ← 0b000
Do i=1,63-exp
    frac[0:64] || gbit || rbit || xbit ← 0b0 || frac[0:64] || gbit ||
(rbit|xbit)
End

If gbit | rbit | xbit then FPSCR[XX] ← 1

```

## Round Integer (frac,gbit,rbit,xbit,round\_mode)

In this example, u represents an undefined hexadecimal digit. Comparisons ignore the u bits.

```

If sign=1 then frac[0:64] ← -frac[0:64] + 1
If tgt_precision="32-bit integer" and frac[0:64]>+2(31)-1
    then goto Large Operand
If tgt_precision="64-bit integer" and frac[0:64]>+2(63)-1
    then goto Large Operand
If tgt_precision="32-bit integer" and frac[0:64]<-2(31) then goto Large
Operand
If tgt_precision="64-bit integer" and frac[0:64]<-2(63) then goto Large
Operand
If tgt_precision="32-bit integer"
    then FRT ← 0x xuuuuuuu || frac[33:64]
If tgt_precision="64-bit integer" then FRT ← frac[1:64]
FPSCR[FPRF] ← undefined
Done

```

## Round Integer (frac,gbit,rbit,xbit,round\_mode)

In this example, u represents an undefined hexadecimal digit. Comparisons ignore the u bits.

```
inc ← 0
If round_mode= 0b00 then
  Do
    If sign || frac[64] || gbit || rbit || xbit = 0b11ux then inc ← 1
    If sign || frac[64] || gbit || rbit || xbit = 0b011x then inc ← 1
    If sign || frac64 || gbit || rbit || xbit = 0b01u1 then inc ← 1
  End
If round_mode= 0b10 then
  Do
    If sign || frac64 || gbit || rbit || xbit = 0b0u1ux then inc ← 1
    If sign || frac64 || gbit || rbit || xbit = 0b0uu1x then inc ← 1
    If sign || frac64 || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
  End
If round_mode = 0b11 then
  Do
    If sign || frac64 || gbit || rbit || xbit = 0b1u1ux then inc ← 1
    If sign || frac64 || gbit || rbit || xbit = 0b1uu1x then inc ← 1
    If sign || frac64 || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
  End
frac[0:64] ← frac[0:64] + inc
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

## Infinity Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE]=0 then Do
  If tgt_precision="32-bit integer" then
    Do
      If sign=0 then FRT ← 0xuuuu uuuu 7FFF FFFF
      If sign=1 then FRT ← 0xuuuu uuuu 8000 0000
    End
  Else
    Do
      If sign=0 then FRT ← 0x7FFF FFFF FFFF FFFF
      If sign=1 then FRT ← 0x8000 0000 0000 0000
    End
  End
  FPSCR[FPRF] < undefined
End
Done
```

## SNaN Operand

```
FPSCR[FR FI VXCVI VXSNaN] ← 0b0011
If FPSCR[VE]=0 then
  Do
    If tgt_precision="32-bit integer"
      then FRT ← 0xuuuu uuuu 8000 0000
    If tgt_precision="64-bit integer"
      then FRT ← 0x8000 0000 0000 0000
    FPSCR[FPRF] ← undefined
  End
Done
```

## QNaN Operand

```

FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE]=0 then
  Do
    If tgt_precision="32-bit integer" then FRT ← 0xuuuu uuuu 8000 0000
    If tgt_precision="64-bit integer" then FRT ← 0x8000 0000 0000 0000
    FPSCR[FPRF] < undefined
  End
Done

```

## Large Operand

```

FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE]=0 then Do
  If tgt_precision="32-bit integer" then
    Do
      If sign=0 then FRT ← 0xuuuu uuuu 7FFF FFFF
      If sign=1 then FRT ← 0xuuuu uuuu 8000 0000
    End
  Else
    Do
      If sign=0 then FRT ← 0x7FFF FFFF FFFF FFFF
      If sign=1 then FRT ← 0x8000 0000 0000 0000
    End
  FPSCR[FPRF] ← undefined
End
Done

```

## C.4 Floating-Point Convert from Integer Model

The following algorithm describes the operation of the floating-point convert from integer instructions.

```

sign ← FRB[0]
exp ← 63
frac ← FRB

If frac=0 then go to Zero Operand
If sign=1 then frac ← ¬frac + 1

Do until frac[0]=1
  frac ← frac[1:63] || 0b0
  exp ← exp - 1
End

```

### Round Float (sign,exp,frac,FPSCR[RN])

```

If sign=1 then FPSCR[FPRF] ← "-normal number"
If sign=0 then FPSCR[FPRF] ← "+normal number"
FRT[0] ← sign
FRT[1:11] ← exp + 1023 /* exp + bias */
FRT[12:63] ← frac[1:52]
Done

```

### Zero Operand

```

FPSCR[FR FI] ← 0b00
FPSCR[FPRF] ← "+zero"
FRT ← 0x0000 0000 0000 0000
Done

```

## Round Float (sign,exp,frac,round\_mode)

In this example, the bits designated as u are ignored in comparisons.

```

inc ← 0
lsb ← frac[52]
gbit ← frac[53]
rbit ← frac[54]
xbit ← frac[55-63] > 0
If round_mode=0b00 then
  Do
    If sign || lsb || gbit || rbit || xbit = 0b11uu then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b011u then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b01u1 then inc ← 1
  End
If round_mode= 0b10 then
  Do
    If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
  End
If round_mode= 0b11 then
  Do
    If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
    If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
  End
frac[0:52] ← frac[0:52] + inc
If carry_out=1 then exp ← exp + 1
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
If (gbit | rbit | xbit) then FPSCR[XX] ← 1
Return
  
```





## APPENDIX D

### SYNCHRONIZATION PROGRAMMING EXAMPLES

The examples in this appendix show how synchronization instructions can be used to emulate various synchronization primitives and how to provide more complex forms of synchronization.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization of the accessed data.

#### D.1 General Information

The following points provide general information about the **lwarx** and **stwcx** instructions:

- In general, **lwarx** and **stwcx** instructions should be paired, with the same effective address used for both. The exception is an isolated **stwcx** instruction that is used to clear any existing reservation on the processor, for which there is no paired **lwarx** and for which any (scratch) effective address can be used.
- It is acceptable to execute an **lwarx** instruction for which no **stwcx** instruction is executed. For example, such a dangling **lwarx** instruction occurs if the value loaded in the test and set sequence shown in [D.3.2 Test and Set](#) is not zero.
- To increase the likelihood that forward progress is made, it is important that looping on **lwarx/stwcx** pairs be minimized. For example, in the sequence shown above for test and set, this is achieved by testing the old value before attempting the store — were the order reversed, more **stwcx** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx** instructions.
- The manner in which **lwarx** and **stwcx** are communicated to other processors and mechanisms and between levels of the memory subsystem within a given processor is implementation-dependent. In some implementations performance may be improved by minimizing looping on an **lwarx** instruction that fails to return a desired value. For example, in the test and set example shown above, to stay in the loop until the word loaded is zero, the programmer could change the **bne S+ 12** to **bne loop**. However, in some implementations better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows:

```

loop:  lwz      r5,0(r3)      #load the word
        cmpwi   r5,0         #loop back if word
        bne     loop        #not equal to 0
        lwarx   r5,0,r3      #try again, reserving
        cmpwi   r5,0         #(likely to succeed)
        bne     loop        #try to store nonzero
        stwcx.  r4,0,r3      #loop if lost reservation
        bne     loop

```

- In a multiprocessor, livelock is possible if a loop containing an **lwarx/stwcx.** pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule of the reservation. For example, the first code sequence shown in [D.5 List Insertion](#) can cause livelock if two list elements have next element pointers in the same reservation granule.

## D.2 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to emulate various synchronization primitives. The sequences used to emulate the various primitives consist primarily of a loop using **lwarx** and **stwcx.**. Additional synchronization is unnecessary, because the **stwcx.** will fail, clearing the EQ bit, if the word loaded by **lwarx** has changed before the **stwcx.** is executed.

### D.2.1 Fetch and No-Op

The fetch and no-op primitive atomically loads the current value in a word in memory. In this example it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:    lwarx      r4,0,r3      #load and reserve
         stwcx.    r4,0,r3      #store old value if still reserved
         bne      loop          #loop if lost reservation
```

The **stwcx.**, if it succeeds, stores to the destination location the same value that was loaded by the preceding **lwarx**. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the **lwarx** was the current value (that is, the source of the value loaded by the **lwarx** was the last store to the location that preceded the **stwcx.** in the coherence order for the location).

### D.2.2 Fetch and Store

The fetch and store primitive atomically loads and replaces a word in memory.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx      r5,0,r3      #load and reserve
         stwcx.    r4,0,r3      #store new value if still reserved
         bne      loop          #loop if lost reservation
```

### D.3 Fetch and Add

The fetch and add primitive atomically increments a word in memory.

In this example it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx      r5,0,r3      #load and reserve
         add       ra,r4,r5      #increment word
         stwcx.    ra,0,r3      #store new value if still reserved
         bne      loop          #loop if lost reservation
```

## D.3.1 Fetch and AND

The fetch and AND primitive atomically performs a logical AND of a value and a word in memory.

In this example it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx      r5,0,r3      #load and reserve
         and       ra,r4,r5      #AND word
         stwcx.    ra,0,r3      #store new value if still reserved
         bne      loop          #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory, simply by changing the AND instruction to the desired Boolean instruction (OR, XOR, etc.).

## D.3.2 Test and Set

The test and set primitive atomically loads a word from memory, ensures that the word in memory contains a non-zero value, and sets the EQ bit of CR field 0 according to whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR3, the new value (non-zero) is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx      r5,0,r3      #load and reserve
         cmpwi     r5,0          #done if word
         bne      $+12          #not equal to 0
         stwcx.    r4,0,r3      #try to store nonzero
         bne      loop          #loop if lost reservation
```

Test and set is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx.** Test and set does not scale well. Using test and set before a critical section allows only one process to execute in the critical section at a time. Using **lwarx** and **stwcx.** to bracket the critical section allows many processes to execute in the critical section at once, but at most one will succeed in exiting from the section with its results stored.

Depending on the application, if test and set fails (that is, clears the EQ bit of CR field 0) it may be appropriate to re-execute the test and set.

## D.4 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory. If they are equal, it stores the value from a second register into the word in memory. If they are unequal, it loads the word from memory into the first register, and sets the EQ bit of the CR0 field to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR3, the comparand is in GPR4, the new value is in GPR5, and the old value is returned in GPR6.

```
lwarx      r6,0,r3      #load and reserve
cmpw      r4,r6         #first 2 operands equal ?
bne       $+8          #skip if not
stwcx.    r5,0,r3       #store new value if still reserved
```

Compare and swap is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx**. A major weakness of typical compare and swap instructions is that they permit spurious success if the word being tested has changed and then changed back to its old value: the sequence shown above does not have this weakness.

Depending on the application, if compare and swap fails (that is, clears the EQ bit of CR0) it may be appropriate to recompute the value potentially to be stored and then re-execute the compare and swap.

## D.5 List Insertion

The following example shows how the **lwarx** and **stwcx** instructions can be used to implement simple LIFO (last-in-first-out) insertion into a singly-linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below, and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list: this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset 0 from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:      lwarx      r2,0,r3      #get next pointer
           stw       r2,0(r4)     #store in new element
           sync                      #let store settle (can omit if not
MP)                                     MP)
           stwcx. r 4, a, r3      #add new element to list
           bne      loop          #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, livelock can occur. (Livelock is a state in which processors interact in a way such that no processor makes progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, code sequence.

**Freescale Semiconductor, Inc.**

```
loop1:  lwz      r2,0(r3)    #get next pointer
        mr      r5,r2      #keep a copy
        stw     r2,0(r4)    #store in new element
        sync                    #let store settle
loop2:  lwarx   rZ,0,r3     #get it again
        cmpw   r2,r5       #loop if changed (someone
        bne    loop1       #else progressed)
        stwcx. r4,0,r3     #add new element to list
        bne    loop2       #loop if failed
```



## APPENDIX E

### SIMPLIFIED MNEMONICS

This appendix is provided in order to simplify writing and comprehending assembly language programs. Included are a set of simplified mnemonics and symbols that define the simple shorthand used for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions.

#### E.1 Symbols

The symbols in [Table E-1](#) are defined for use in instructions (basic or simplified mnemonics) that specify a condition register (CR) field or a bit in the CR.

**Table E-1 Condition Register CR Field Bit Symbols**

Symbol	Value	Bit Field Range	Description
<b>lt</b>	0	—	Less than. Identifies a bit number within a CR field.
<b>gt</b>	1	—	Greater than. Identifies a bit number within a CR field.
<b>eq</b>	2	—	Equal. Identifies a bit number within a CR field.
<b>so</b>	3	—	Summary overflow. Identifies a bit number within a CR field.
<b>un</b>	3	—	Unordered (after floating-point comparison). Identifies a bit number within a CR field.
<b>cr0</b>	0	0:3	CR0 field.
<b>cr1</b>	1	4:7	CR1 field.
<b>cr2</b>	2	8:11	CR2 field.
<b>cr3</b>	3	12:15	CR3 field.
<b>cr4</b>	4	16:19	CR4 field.
<b>cr5</b>	5	20:23	CR5 field.
<b>cr6</b>	6	24:27	CR6 field.
<b>cr7</b>	7	28:31	CR7 field.

The simplified mnemonics in [E.5 Simplified Mnemonics for Branch Instructions](#) and [E.6 Simplified Mnemonics for Condition Register Logical Instructions](#) require identification of a CR bit. If one of the CR field symbols is used, it must be multiplied by four and added to a symbol or value (zero to three) representing the bit number within the CR field.

The simplified mnemonics in [E.5.3 Branch Mnemonics Incorporating Condi-](#)

tions and [E.3 Simplified Mnemonics for Compare Instructions](#) require identification of a CR field. If one of the CR field symbols is used, it must *not* be multiplied by four. Refer to each of these sections for examples that use the symbols in [Table E-1](#).

## E.2 Simplified Mnemonics for Subtract Instructions

This section discusses simplified mnemonics for the subtract instructions.

### E.2.1 Subtract Immediate

Although there is not a “subtract immediate” instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation, making the intent of the computation clearer. In these examples, the immediate operand “value” is subtracted from the value in **rA** and the result placed in **rD**.

<b>subi</b>	<b>rD,rA,value</b>	(equivalent to <b>addi rD,rA,-value</b> )
<b>subis</b>	<b>rD,rA,value</b>	(equivalent to <b>addis rD,rA,-value</b> )
<b>subic</b>	<b>rD,rA,value</b>	(equivalent to <b>addic rD,rA,-value</b> )
<b>subic.</b>	<b>rD,rA,value</b>	(equivalent to <b>addic. rD,rA,-value</b> )

### E.2.2 Subtract

The “subtract-from” instructions subtract the second operand (**rA**) from the third (**rB**). Simplified mnemonics are provided in which the third operand is subtracted from the second. Both these mnemonics can be coded with a final ‘o’ or ‘.’ (or both) to cause the OE or Rc bit, respectively, to be set in the underlying instruction. In these examples, the value in **rB** is subtracted from the value in **rA** and the result placed in **rD**.

<b>sub</b>	<b>rD,rA,rB</b>	(equivalent to <b>subf rD,rB,rA</b> )
<b>subc</b>	<b>rD,rA,rB</b>	(equivalent to <b>subfc rD,rB,rA</b> )

## E.3 Simplified Mnemonics for Compare Instructions

The instructions listed in [Table 4-3](#) are simplified mnemonics that provide compare word capability for 32-bit operands. These instructions correctly clear the L value in the instruction (specifying a 32-bit operand; refer to [4.3.2 Integer Compare Instructions](#)) rather than requiring it to be coded as a numeric operand.

The **crfD** field can be omitted if the result of the comparison is to be placed into the CR0 field. Otherwise, the target CR field must be specified as the first operand. The CR field symbols defined in [E.1 Symbols](#) can be used to identify the condition register field.



## CAUTION

If the **crfD** field is omitted from a compare mnemonic, the L field must also be omitted. That is, when the assembler encounters a compare instruction with three operands, it interprets the first operand to be the **crfD** field.

### Table E-2 Word Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Word Immediate	<b>cmpwi crfD,rA,SIMM</b> <b>cmpi crfD,rA,SIMM</b>	<b>cmpi crfD,0,rA,SIMM</b>
Compare Word	<b>cmpw crfD,rA,rB</b> <b>cmp crfD,rA,rB</b>	<b>cmp crfD,0,rA,rB</b>
Compare Logical Word Immediate	<b>cmplwi crfD,rA,UIMM</b> <b>cmpli crfD,rA,UIMM</b>	<b>cmpli crfD,0,rA,UIMM</b>
Compare Logical Word	<b>cmplw crfD,rA,rB</b> <b>cmpl crfD,rA,rB</b>	<b>cmpl crfD,0,rA,rB</b>

The following examples demonstrate the use of the word compare mnemonics:

11. Compare 32 bits in register **rA** with immediate value 100 and place result in condition register field CR0.

**cmpwi**    rA,100                      (equivalent to **cmpi** 0,0,rA,100)

12. Same as (1), but place results in condition register field CR4.

**cmpwi cr4,rA,100** (equivalent to **cmpi 4,0,rA,100**)

13. Compare registers **rA** and **rB** as logical 32-bit quantities and place result in condition register field **CR0**.

**cmplw**    rA,rB                    (equivalent to **cmpl 0,0,rA,rB**)

14. Same as (3), but place result in condition register field CR4.

**cmplw**    **cr4,rA,rB**                    (equivalent to **cmpl 4,0,rA,rB**)

## E.4 Simplified Mnemonics for Rotate and Shift Instructions

The rotate and shift instructions provide powerful and general ways to manipulate register contents but can be difficult to understand. Simplified mnemonics, which allow some of the simpler operations to be coded easily, are provided for the following types of operations:

- **Extract** — Select a field of  $n$  bits starting at bit position  $b$  in the source register; left or right justify this field in the target register; clear all other bits of the target register.
- **Insert** — Select a left-justified or right-justified field of  $n$  bits in the source register; insert this field into the target register, clearing all other bits of the target register.

ister; insert this field starting at bit position  $b$  of the target register; leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a left-justified field when operating on double words, because such an insertion requires more than one instruction.)

- Rotate — Rotate the contents of a register right or left  $n$  bits without masking.
- Shift — Shift the contents of a register right or left  $n$  bits, clearing vacated bits (logical shift).
- Clear — Clear the leftmost or rightmost  $n$  bits of a register.
- Clear left and shift left — Clear the leftmost  $b$  bits of a register, then shift the register left by  $n$  bits. This operation can be used to scale a (known non-negative) array index by the width of an element.

The word rotate and shift operations shown in **Table E-3** are available in all implementations. All these mnemonics can be coded with a final ‘.’ to cause the Rc bit to be set in the underlying instruction.

**Table E-3 Word Rotate and Shift Instructions**

Operation	Simplified Mnemonic	Equivalent to
Extract and left justify immediate	<b>extlwi</b> $rA, rS, n, b$ ( $n > 0$ )	<b>rlwinm</b> $rA, rS, b, 0, n-1$
Extract and right justify immediate	<b>extrwi</b> $rA, rS, n, b$ ( $n > 0$ )	<b>rlwinm</b> $rA, rS, b + n, 32 - n, 31$
Insert from left immediate	<b>inslwi</b> $rA, rS, n, b$	<b>rlwimi</b> $rA, rS, 32-b, b, b+n-1$
Insert from right immediate	<b>insrwi</b> $rA, rS, n, b$	<b>rlwimi</b> $rA, rS, 32-(b+n), b, b+n-1$
Rotate left immediate	<b>rotlwi</b> $rA, rS, n$	<b>rlwinm</b> $rA, rS, n, 0, 31$
Rotate right immediate	<b>rotrwi</b> $rA, rS, n$	<b>rlwinm</b> $rA, rS, 32 - n, 0, 31$
Rotate left	<b>rotlw</b> $rA, rS, rB$	<b>rlwnm</b> $rA, rS, rB, 0, 31$
Shift left immediate	<b>srwi</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, n, 0, 31-n$
Shift right immediate	<b>srwi</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, 32-n, n, 31$
Clear left immediate	<b>clrlwi</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, 0, n, 31$
Clear right immediate	<b>clrrwi</b> $rA, rS, n$ ( $n < 32$ )	<b>rlwinm</b> $rA, rS, 0, 0, 31-n$
Clear left and shift left immediate	<b>clrlslwi</b> $rA, rS, b, n$ ( $n \neq b \neq 31$ )	<b>rlwinm</b> $rA, rS, n, b-n, 31-n$

The following examples illustrate the use of these mnemonics.

1. Extract the sign bit (bit 32) of  $rS$  and place the result right-justified into  $rA$ .  
**extrwi**  $rA, rS, 1, 0$  (equivalent to: **rlwinm**  $rA, rS, 1, 31, 31$ )
2. Insert the bit extracted in (1) into the sign bit (bit 32) of  $rB$ .  
**insrwi**  $rB, rA, 1, 0$  (equivalent to: **rlwimi**  $rB, rA, 31, 0, 0$ )
3. Shift the contents of  $rA$  left 8 bits, clearing the high-order 32 bits.  
**slwi**  $rA, rA, 8$  (equivalent to: **rlwinm**  $rA, rA, 8, 0, 23$ )

## E.5 Simplified Mnemonics for Branch Instructions

Mnemonics are provided so that branch conditional instructions can be coded with the condition as part of the instruction mnemonic rather than as a numeric operand. The mnemonics discussed in this section are variations of the branch conditional instructions.

### E.5.1 BO and BI Fields

The 5-bit BO field in branch conditional instructions encodes the following operations:

- Decrement count register (CTR)
- Test CTR equal to zero
- Test CTR not equal to zero
- Test condition true
- Test condition false
- Branch prediction (taken, fall through)

**Table E-4 BO Operand Encodings**

BO	Description
0000y	Decrement the CTR, then branch if the decremented CTR $\neq 0$ and the condition is FALSE.
0001y	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement the CTR, then branch if the decremented CTR $\neq 0$ and the condition is TRUE.
0101y	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement the CTR, then branch if the decremented CTR $\neq 0$ .
1z01y	Decrement the CTR, then branch if the decremented CTR = 0.
1z1zz	Branch always.

The z indicates a bit that must be zero; otherwise, the instruction form is invalid.

The y bit provides a hint about whether a conditional branch is likely to be taken.

The 5-bit BI field in branch conditional instructions specifies which of the 32 bits in the CR represents the condition to test.

To provide a simplified mnemonic for every possible combination of BO and BI fields would require  $2^{10} = 1024$  mnemonics, most of which would be only marginally useful. The abbreviated set found in **E.5.2 Basic Branch Mnemonics** is intended to cover the most useful cases. Unusual cases can be coded using a basic branch conditional mnemonic (**bc**, **bclr**, **bcctr**) with the condition to be tested specified as a numeric operand.

### E.5.2 Basic Branch Mnemonics

**Table E-5** provides the simplified mnemonics for the most commonly performed conditional branches. These mnemonics allow all the BO operand encodings shown in **Table E-4** to be specified as part of the mnemonic, along with the absolute address (AA) and set link register (LK) bits. (The y bit in the BO operand is always cleared in these simplified mnemonics.)

Notice that there are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

**Table E-5 Simplified Branch Mnemonics**

	LK Bit Not Set (LR Update Not Enabled)				LK Bit Set (LR Update Enabled)			
Branch Semantics	bc Relative	bca Absolute	bclr to LR	bcctr to CTR	bcl Relative	bcla Absolute	bclrl to LR	bcctrl to CTR
Branch unconditionally	b <sup>1</sup>	ba <sup>1</sup>	blr	bctr	bl <sup>1</sup>	bla <sup>1</sup>	blrl	bctrl
Branch if condition true <sup>2</sup>	bt	bta	btlr	btctr	btl	btla	btlrl	btctrl
Branch if condition false <sup>2</sup>	bf	bfa	bflr	bfctr	bfl	bfla	bflrl	bfctrl
Decrement CTR, branch if CTR non-zero	bdnz	bdnza	bdnzlr	—	bdnzl	bdnzla	bdnzlrl	—
Decrement CTR, branch if CTR non-zero AND condition true	bdnzt	bdnzta	bdnztlr	—	bdnztl	bdnztla	bdnztlrl	—
Decrement CTR, branch if CTR non-zero AND condition false	bdnzf	bdnzfa	bdnzflr	—	bdnzfl	bdnzfla	bdnzflrl	—
Decrement CTR, branch if CTR zero	bdz	bdza	bdzlr	—	bdzl	bdzla	bdzlrl	—
Decrement CTR, branch if CTR zero AND condition true	bdzt	bdzta	bdztlr	—	bdztl	bdztla	bdztlrl	—
Decrement CTR, branch if CTR zero AND condition false	bdzf	bdzfa	bdzflr	—	bdzfl	bdzfla	bdzflrl	—

**NOTES:**

1. These are basic mnemonics, not simplified mnemonics.
2. Refer to [Table E-7](#) for an expanded set of simplified mnemonics for “branch if condition true” and “branch if condition false.” This expanded set of simplified mnemonics incorporates the condition being tested as part of the mnemonic.

[Table E-6](#) provides the operands for the simplified mnemonics in [Table E-5](#), as well as the operands of the corresponding basic branch instruction.

**Table E-6 Operands for Simplified Branch Mnemonics**

Branch Type	Simplified Mnemonic		Equivalent to:	
	Mnemonic	Operands	Mnemonic	Operands
Branch unconditionally	<b>blr</b>	None	<b>bclr</b>	<b>20,0</b>
	<b>bctr</b>	None	<b>bcctr</b>	<b>20,0</b>
	<b>blrl</b>	None	<b>bclrl</b>	<b>20,0</b>
	<b>bctrl</b>	None	<b>bcctrl</b>	<b>20,0</b>
Branch if true	<b>bt</b>	BI,target	<b>bc</b>	<b>12,BI,target</b>
	<b>bta</b>	BI,target	<b>bca</b>	<b>12,BI,target</b>
	<b>btlr</b>	BI	<b>bclr</b>	<b>12,BI</b>
	<b>btctr</b>	BI	<b>bcctr</b>	<b>12,BI</b>
	<b>btl</b>	BI,target	<b>bcl</b>	<b>12,BI,target</b>
	<b>btla</b>	BI,target	<b>bcla</b>	<b>12,BI,target</b>
	<b>btlrl</b>	BI	<b>bclrl</b>	<b>12,BI</b>
	<b>btctrl</b>	BI	<b>bcctrl</b>	<b>12,BI</b>
Branch if false	<b>bf</b>	BI,target	<b>bc</b>	<b>4,BI,target</b>
	<b>bfa</b>	BI,target	<b>bca</b>	<b>4,BI,target</b>
	<b>bflr</b>	BI	<b>bclr</b>	<b>4,BI</b>
	<b>bfctr</b>	BI	<b>bcctr</b>	<b>4,BI</b>
	<b>bfl</b>	BI,target	<b>bcl</b>	<b>4,BI,target</b>
	<b>bfla</b>	BI,target	<b>bcla</b>	<b>4,BI,target</b>
	<b>bflrl</b>	BI	<b>bclrl</b>	<b>4,BI</b>
	<b>bfctrl</b>	BI	<b>bcctrl</b>	<b>4,BI</b>
Decrement CTR, branch if CTR non-zero	<b>bdnz</b>	target	<b>bc</b>	<b>16,0,target</b>
	<b>bdnza</b>	target	<b>bca</b>	<b>16,0,target</b>
	<b>bdnzlr</b>	None	<b>bclr</b>	<b>16,0</b>
	<b>bdnzl</b>	target	<b>bcl</b>	<b>16,0,target</b>
	<b>bdnzla</b>	target	<b>bcla</b>	<b>16,0,target</b>
	<b>bdnzlrl</b>	None	<b>bclrl</b>	<b>16,0</b>
Decrement CTR, branch if CTR non-zero AND condition true	<b>bdnzt</b>	BI,target	<b>bc</b>	<b>8,BI,target</b>
	<b>bdnzta</b>	BI,target	<b>bca</b>	<b>8,BI,target</b>
	<b>bdnztlr</b>	BI	<b>bclr</b>	<b>8,BI</b>
	<b>bdnztl</b>	BI,target	<b>bcl</b>	<b>8,BI,target</b>
	<b>bdnztla</b>	BI,target	<b>bcla</b>	<b>8,BI,target</b>
	<b>bdnztlrl</b>	BI	<b>bclrl</b>	<b>8,BI</b>
Decrement CTR, branch if CTR non-zero AND condition false	<b>bdnzf</b>	BI,target	<b>bc</b>	<b>0,BI,target</b>
	<b>bdnzfa</b>	BI,target	<b>bca</b>	<b>0,BI,target</b>
	<b>bdnzflr</b>	BI	<b>bclr</b>	<b>0,BI</b>
	<b>bdnzfl</b>	BI,target	<b>bcl</b>	<b>0,BI,target</b>
	<b>bdnzfla</b>	BI,target	<b>bcla</b>	<b>0,BI,target</b>
	<b>bdnzflrl</b>	BI	<b>bclrl</b>	<b>0,BI</b>

Table E-6 Operands for Simplified Branch Mnemonics (Continued)

Branch Type	Simplified Mnemonic		Equivalent to:	
	Mnemonic	Operands	Mnemonic	Operands
Decrement CTR, branch if CTR zero	<b>bdz</b>	target	<b>bc</b>	18,0,target
	<b>bdza</b>	target	<b>bca</b>	18,0,target
	<b>bdzlr</b>	None	<b>bclr</b>	18,0
	<b>bdzl</b>	target	<b>bcl</b>	18,0,target
	<b>bdzla</b>	target	<b>bcla</b>	18,0,target
	<b>bdzlrl</b>	None	<b>bclrl</b>	18,0
Decrement CTR, branch if CTR zero AND condition true	<b>bdzt</b>	BI,target	<b>bc</b>	10,BI,target
	<b>bdzta</b>	BI,target	<b>bca</b>	10,BI,target
	<b>bdztlr</b>	BI	<b>bclr</b>	10,BI
	<b>bdztl</b>	BI,target	<b>bcl</b>	10,BI,target
	<b>bdztla</b>	BI,target	<b>bcla</b>	10,BI,target
	<b>bdztlrl</b>	BI	<b>bclrl</b>	10,BI
Decrement CTR, branch if CTR zero AND condition false	<b>bdzf</b>	BI,target	<b>bc</b>	2,BI,target
	<b>bdzfa</b>	BI,target	<b>bca</b>	2,BI,target
	<b>bdzflr</b>	BI	<b>bclr</b>	2,BI
	<b>bdzfl</b>	BI,target	<b>bcl</b>	2,BI,target
	<b>bdzfla</b>	BI,target	<b>bcla</b>	2,BI,target
	<b>bdzflrl</b>	BI	<b>bclrl</b>	2,BI

Instructions using a mnemonic from [Table E-5](#) that test a condition specify the condition (bit in the condition register) as the first (BI) operand of the instruction. The symbols defined in [E.1 Symbols](#) can be used in this operand. If one of the CR field symbols is used, it must be multiplied by four and added to a symbol or value (zero to three) representing the bit number within the CR field.

The simplified mnemonics found in [Table E-5](#) are illustrated in the following examples:

- Decrement CTR and branch if it is still non-zero (closure of a loop controlled by a count loaded into CTR).  
**bdnz**      target      (equivalent to **bc 16,0, target**)
- Same as (1) but branch only if CTR is non-zero and condition in CR0 is "equal."  
**bdnzt**    **eq**, target      (equivalent to **bc 8,2,target**)
- Same as (2), but "equal" condition is in CR5.  
**bdnzt**    **4 \* cr5+eq**,target      (equivalent to **bc 8,22,target**)
- Branch if bit 27 of CR is false.  
**bf**      **27**,target      (equivalent to **bc 4,27,target**)
- Same as (4), but set the link register. This is a form of conditional "call."  
**bfl**      **27**,target      (equivalent to **bcl 4,27,target**)

### E.5.3 Branch Mnemonics Incorporating Conditions

The mnemonics defined in [Table E-7](#) are variations of the “branch if condition true” and “branch if condition false” BO encodings, with the most common values of the BI operand represented in the mnemonic rather than specified as a numeric operand.

**Table E-7 Simplified Branch Mnemonics with Comparison Conditions**

	LK Bit Not Set (LR Update Not Enabled)				LK Bit Set (LR Update Enabled)			
Branch Semantics	bc Relative	bca Absolute	bclr to LR	bcctr to CTR	bcl Relative	bcla Absolute	bclrl to LR	bcctrl to CTR
Branch if less than	blt	blta	bltlr	bltctr	bltl	bltla	bltlrl	bltctrl
Branch if less than or equal	ble	blea	blelr	blectr	blel	blela	blelrl	blectrl
Branch if equal	beq	beqa	beqlr	beqctr	beql	beqla	beqlrl	beqctrl
Branch if greater than	bge	bgea	bgehr	bgectr	bgehr	bgeha	bgehrhl	bgectrl
Branch if greater than	bgt	bgtla	bgtlr	bgtctr	bgtl	bgtla	bgtlrl	bgtctrl
Branch if not less than	bhl	bhla	bhlhr	bhlctr	bhl	bhla	bhlhrhl	bhlctrl
Branch if not equal	bne	bnea	bnelr	bnectr	bnel	bnela	bnelrl	bnectrl
Branch if not greater than	bng	bnga	bnglr	bngctr	bngl	bngla	bnglrl	bngctrl
Branch if summary overflow	bso	bsoa	bsolr	bsoctr	bsol	bsola	bsolrl	bsoctrl
Branch if not summary overflow	bns	bnsa	bnslr	bnsctr	bns	bnsa	bnsrl	bnsctrl
Branch if unordered	bun	buna	bunlr	bunctr	bun	buna	bunrl	bunctrl
Branch if not unordered	bnu	bnu	bnulr	bnuctr	bnu	bnu	bnulrl	bnuctrl

[Table E-8](#) shows the operands used with the simplified branch mnemonics in [Table E-7](#). The examples provided are for the first column of [Table E-7](#) (simplified forms of the **bc** instruction), but all entries within a row in [Table E-7](#) use the same operands (except that branches to the LR or CTR do not require a “target” operand). [Table E-8](#) also indicates the operands used with the corresponding basic branch mnemonic.

**Table E-8 Operands for Simplified Branch Mnemonics with Comparison Conditions**

Branch	Simplified Mnemonics Example	Equivalent to
Branch if less than	<b>blt</b> crfD,target	<b>bc</b> 12,4*crfD,target
Branch if less than or equal	<b>ble</b> crfD,target	<b>bc</b> 4,4*crfD+1,target
Branch if equal	<b>beq</b> crfD,target	<b>bc</b> 12, 4*crfD+2,target
Branch if greater than	<b>bgt</b> crfD,target	<b>bc</b> 12,4*crfD+1,target

**Table E-8 Operands for Simplified Branch Mnemonics with Comparison Conditions (Continued)**

Branch	Simplified Mnemonics Example	Equivalent to
Branch if greater than or equal	<b>bge crfD</b> ,target	<b>bc 4,4*crfD</b> ,target
Branch if not less than	<b>bnl crfD</b> ,target	<b>bc 4,4*crfD</b> ,target <sup>1</sup>
Branch if not equal	<b>bne crfD</b> ,target	<b>bc 4,4*crfD+2</b> ,target
Branch if not greater than	<b>bng crfD</b> ,target	<b>bc 4,4*crfD+1</b> ,target <sup>2</sup>
Branch if summary overflow	<b>bsu crfD</b> ,target	<b>bc 12,4*crfD+3</b> ,target
Branch if not summary overflow	<b>bns crfD</b> ,target	<b>bc 4,4*crfD+3</b> ,target
Branch if unordered	<b>bun crfD</b> ,target	<b>bc 12,4*crfD+3</b> ,target
Branch if not unordered	<b>bnu crfD</b> ,target	<b>bc 4,4*crfD+3</b> ,target

## NOTES:

1. Same as “branch if greater than or equal.”
2. Same as “branch if less than or equal.”

Instructions using the mnemonics in [Table E-7](#) specify the condition register field in an optional first operand. If the CR field being tested is CR0, this operand need not be specified. Otherwise, one of the CR field symbols defined in [E.1 Symbols](#) can be used for this operand.

If one of the CR field symbols is used, it must *not* be multiplied by four. The bit number within the CR field is part of the simplified mnemonic. The CR field is identified, and the assembler does the multiplication and addition required to produce a CR bit number for the BI field of the underlying basic mnemonic.)

The simplified mnemonics found in **Table E-7** are used in the following examples:

- [illegible]

### E.5.4 Branch Prediction

In branch conditional instructions that are not always taken, the low-order bit (y bit)



## Freescale Semiconductor, Inc.

of the BO field provides a hint about whether the branch is likely to be taken. See [4.6.2 Conditional Branch Control](#) for more information on the y bit.

Assemblers should clear this bit unless otherwise directed. This default action indicates the following:

- A branch conditional with a negative displacement field is predicted to be taken.
- A branch conditional with a non-negative displacement field is predicted not to be taken (fall through).
- A branch conditional to an address in the LR or CTR is predicted not to be taken (fall through).

If the likely outcome (branch or fall through) of a given branch conditional instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the y bit. That is, '+' indicates that the branch is to be taken and '-' indicates that the branch is not to be taken. Such a suffix can be added to any branch conditional mnemonic, either basic or simplified.

For relative and absolute branches (**bc[l][a]**), the setting of the y bit depends on whether the displacement field is negative or non-negative. For negative displacement fields, coding the suffix '+' causes the bit to be cleared, and coding the suffix '-' causes it to be set. For non-negative displacement fields, coding the suffix '+' causes the bit to be set, and coding the suffix '-' causes the bit to be cleared.

For branches to an address in the LR or CTR (**bcclr[l]** or **bcctr[l]**), coding the suffix '+' causes the y bit to be set, and coding the suffix '-' causes the bit to be cleared.

Examples of branch prediction follow:

1. Branch if CR0 reflects condition "less than," specifying that the branch should be predicted to be taken.  
**blt+**      target
2. Same as (1), but target address is in the LR and the branch should be predicted not to be taken.  
**btlr-**

### E.6 Simplified Mnemonics for Condition Register Logical Instructions

The condition register logical instructions are used to set, clear, copy, or invert a given condition register bit. The simplified mnemonics shown in [Table E-9](#) allow these operations to be coded easily.

Table E-9 Condition Register Logical Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Condition register set	<b>crset bx</b>	<b>creqv bx,bx,bx</b>
Condition register clear	<b>crclr bx</b>	<b>crxor bx,bx,bx</b>
Condition register move	<b>crmove bx,by</b>	<b>cror bx,by,by</b>
Condition register NOT	<b>crnot bx,by</b>	<b>crnor bx,by,by</b>

The symbols defined in [E.1 Symbols](#) can be used to identify the condition register bit. If one of the CR field symbols is used, it must be multiplied by four and added to a symbol or value (zero to three) representing the bit number within the CR field.

The following examples illustrate the condition register logical mnemonics:

1. Set CR bit 25.  
**crset 25** (equivalent to **creqv 25,25,25**)
2. Clear the SO bit of CR0.  
**clclr so** (equivalent to **crxor 3,3,3**)
3. Same as (2), but SO bit to be cleared is in CR3.  
**clclr 4 \* cr3 + so** (equivalent to **crxor 15,15,15**)
4. Invert the EQ bit.  
**crnot eq,eq** (equivalent to **crnor 2,2,2**)
5. Same as (4), but EQ bit to be inverted is in CR4, and the result is to be placed into the EQ bit of CR5.  
**crnot 4\*cr5+eq,4\*cr4+eq** (equivalent to **crnor 22,18,18**)

## E.7 Simplified Mnemonics for Trap Instructions

A standard set of codes, shown in [Table E-10](#), has been adopted for the most common combinations of trap conditions.

Table E-10 Trap Mnemonics Encoding

Code	Meaning	TO Operand Encoding	<	>	=	<U <sup>1</sup>	>U <sup>2</sup>
<b>lt</b>	Less than	16	1	0	0	0	0
<b>le</b>	Less than or equal	20	1	0	1	0	0
<b>eq</b>	Equal	4	0	0	1	0	0
<b>ge</b>	Greater than or equal	12	0	1	1	0	0
<b>gt</b>	Greater than	8	0	1	0	0	0
<b>nl</b>	Not less than	12	0	1	1	0	0
<b>ne</b>	Not equal	24	1	1	0	0	0
<b>ng</b>	Not greater than	20	1	0	1	0	0
<b>llt</b>	Logically less than	2	0	0	0	1	0
<b>lle</b>	Logically less than or equal	6	0	0	1	1	0
<b>lge</b>	Logically greater than or equal	5	0	0	1	0	1
<b>lgt</b>	Logically greater than	1	0	0	0	0	1
<b>lnl</b>	Logically not less than	5	0	0	1	0	1
<b>lng</b>	Logically not greater than	6	0	0	1	1	0
(none)	Unconditional	31	1	1	1	1	1

## NOTES:

1. The symbol '<U' indicates an unsigned "less than" evaluation will be performed.
2. The symbol '>U' indicates an unsigned "greater than" evaluation will be performed.

The mnemonics defined in [Table E-11](#) are variations of the trap instructions, with the most useful values of the trap instruction TO operand represented as a mnemonic rather than specified as a numeric operand.

Table E-11 Trap Mnemonics

Trap Semantics	32-Bit Comparison	
	twi Immediate	tw Register
Trap unconditionally	—	trap
Trap if less than	twlti	twlt
Trap if less than or equal	twlei	twle
Trap if equal	tweqi	tweq
Trap if greater than or equal	twgei	twge
Trap if greater than	twgti	twgt
Trap if not less than	twnli	twnl
Trap if not equal	twnei	twne
Trap if logically less than	twllti	twllt
Trap if logically less than or equal	twlle	twlle
Trap if logically greater than or equal	twllgi	twllg
Trap if logically greater than	twllgi	twllg
Trap if logically not less than	twlnli	twlnl

The following examples illustrate the use of simplified mnemonics for trap instructions:

1. Trap if Rx, considered as a 32-bit quantity, is logically greater than 0x7FF.

**twlg**      rA, 0x7FF      (equivalent to **twi 1,rA, 0x7FF**)

2. Trap unconditionally.

**trap**      (equivalent to **tw 31,0,0**)

Trap instructions evaluate a trap condition as follows: the contents of register rA are compared with either the sign-extended SIMM field or the contents of register rB, depending on the trap instruction.

The comparison results in five conditions which are ANDed with operand TO. If the result is not zero, the trap exception handler is invoked. See [Table E-12](#) for these conditions.

**Table E-12 TO Operand Bit Encoding**

TO Bit	ANDed with Condition
0	Less than, using signed comparison
1	Greater than, using signed comparison
2	Equal
3	Less than, using unsigned comparison
4	Greater than, using unsigned comparison

## E.8 Simplified Mnemonics for Special-Purpose Registers

The **mtspr** and **mfspir** instructions specify an SPR as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as an operand. **Table E-13** below specifies the simplified mnemonics provided for SPR operations.

**Table E-13 SPR Simplified Mnemonics**

Special Purpose Register	Move to SPR Simplified Mnemonic	Move to SPR Instruction	Move from SPR Simplified Mnemonic	Move from SPR Instruction <sup>1</sup>
Integer unit exception register	<b>mtxer rS</b>	<b>mtspr 1,rS</b>	<b>mfixer rD</b>	<b>mfspir rD,1</b>
Link register	<b>mtlr rS</b>	<b>mtspr 8,rS</b>	<b>mflr rD</b>	<b>mfspir rD,8</b>
Count register	<b>mtctr rS</b>	<b>mtspr 9,rS</b>	<b>mfctr rD</b>	<b>mfspir rD,9</b>
DAE/source instruction service register	<b>mtdsisr rS</b>	<b>mtspr 18,rS</b>	<b>mfdsisr rD</b>	<b>mfspir rD,18</b>
Data address register	<b>mtdar rS</b>	<b>mtspr 19,rS</b>	<b>mfdar rD</b>	<b>mfspir rD,19</b>
Decrementer	<b>mtdec rS</b>	<b>mtspr 22,rS</b>	<b>mfdec rD</b>	<b>mfspir rD,22</b>
Status save/restore register 0	<b>mtsrr0 rS</b>	<b>mtspr 26,rS</b>	<b>mfssrr0 rD</b>	<b>mfspir rD,26</b>
Status save/restore register 1	<b>mtsrr1 rS</b>	<b>mtspr 27,rS</b>	<b>mfssrr1 rD</b>	<b>mfspir rD,27</b>
General special purpose registers G0 through G3	<b>mtsprg n,rS</b>	<b>mtspr 272+n,rS</b>	<b>mfspirg rD,n</b>	<b>mfspir rD,272+n</b>
Time base (lower)	<b>mttbl rS</b>	<b>mtspr 284,rS</b>	<b>mftb rD</b>	<b>mftb rD,268</b>
Time base (upper)	<b>mttbu rS</b>	<b>mtspr 285,rS</b>	<b>mftbu rD</b>	<b>mftb rD,269</b>
Processor version register	—	—	<b>mfpvr rD</b>	<b>mfspir rD,287</b>

NOTES:

1. Except for **mftb** and **mftbu**

## Freescale Semiconductor, Inc.

The following examples illustrate the use of SPR simplified mnemonics.

1. Copy the contents of the low-order 32 bits of **rS** to the XER.  
**mtxer rS** (equivalent to **mtspr 1,rS**)
2. Copy the contents of the LR to **rS**.  
**mflr rS** (equivalent to **mtspr rS,8**)
3. Copy the contents of **rS** to the CTR.  
**mtctr rS** (equivalent to **mtspr 9,rS**)

### E.9 Recommended Simplified Mnemonics

This section describes some of the most commonly-used operations: no-op, load immediate, load address, move register, complement register, and move to condition register.

#### E.9.1 No-Op

Many PowerPC instructions can be coded in a way such that, effectively, no operation is performed. An additional mnemonic is provided for the preferred form of no-op.

**nop** (equivalent to **ori 0,0,0**)

#### E.9.2 Load Immediate

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

The following instruction loads a 16-bit signed immediate value into **rA**:

**li rA,value** (equivalent to **addi rA,0,value**)

The following instruction loads a 16-bit signed immediate value, shifted left by 16 bits, into **rA**:

**lis rA,value** (equivalent to **addi rA,0,value**)

#### E.9.3 Load Address

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction which normally requires a separate register and immediate operands.

**la rD,SIMM(rA)** (equivalent to **addi rD,rA,SIMM**)

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the

displacement. If the variable *v* is located at offset `SIMMv` bytes from the address in register *rV*, and the assembler has been told to use register *rV* as a base for references to the data structure containing *v*, then the following line causes the address of *v* to be loaded into register *rD*.

**la**      **rD,v**      (equivalent to **addi rD,rA,SIMMv**)

### E.9.4 Move Register

Several PowerPC instructions can be coded to simply copy the contents of one register to another. An extended mnemonic is provided to move data from one register to another with no computational activity.

The following instruction copies the contents of register **rS** into register **rA**. This mnemonic can be coded with a **.** to cause the condition register update option to be specified in the underlying instruction.

**mr**      **rA,rS**      (equivalent to **or rA,rS,rB**)

### E.9.5 Complement Register

Several PowerPC instructions can be coded to complement the contents of one register and place the result in another register. A simplified mnemonic is provided that complements the contents of **rS** and places the results into register **rA**. This mnemonic can be coded with a **'** to cause the condition register update option to be specified in the underlying instruction.

**not**  $r_A, r_S$  (equivalent to **nor**  $r_A, r_S, r_B$ )

### E.9.6 Move to Condition Register

This mnemonic permits copying the contents of a GPR to the condition register, using the same style as the **mfcrr** instruction.

**mtcr**      **rS**      (equivalent to **mtcrf** 0xFF,rS)





## **GLOSSARY OF TERMS AND ABBREVIATIONS**

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from *IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

- A**      **Atomic.** A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The processor initiates the read and write separately, but signals the L-bus or external bus interface that it is attempting an atomic operation. If the operation fails, status is kept so that the processor can try again. The processor implements atomic accesses through the **lwarx/stwcx**. instruction pair.
- B**      **Beat.** A single state on the external bus interface that may extend across multiple bus cycles. An RCPU transaction can be composed of multiple address or data beats.
- Biased Exponent.** The sum of the exponent and a constant (bias) chosen to make the biased exponent's range non-negative.
- Big-Endian.** A byte-ordering method in memory where the address *n* of a word corresponds to the most significant byte. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most significant byte.
- Blockage.** The number of clock cycles between the time an instruction begins execution and the time its execution unit is available for a subsequent instruction.
- Boundedly Undefined.** The results of attempting to execute a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction. Boundedly undefined results for a given instruction may vary between implementations, and between execution attempts in the same implementation.
- Branch Folding.** A technique of removing the branch instruction from the instruction sequence.
- Breakpoint.** An event that, when detected, forces the machine to branch to a breakpoint exception routine.

**Burst.** A multiple beat data transfer.

**Bus Master.** The owner of the address or data bus; the device that initiates or requests the transaction.

**C** **Cache Coherency.** Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor's cache.

**Context Synchronization.** All instructions in execution complete past the point where they can produce an exception; all instructions in execution complete in the context in which they began execution; all subsequent instructions are fetched and executed in the new context.

**D** **Denormalized Number.** A non-zero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

**E** **Exception.** An unusual or error condition encountered by the processor that results in special processing.

**Exception Handler.** A software routine that executes when an exception occurs. Normally, the exception handler corrects the condition that caused the exception, or performs some other meaningful task (such as aborting the program that caused the exception). The addresses of the exception handlers are defined by a two-word exception vector that is branched to automatically when an exception occurs.

**Execution Serialization.** During execution serialization, instruction issue is halted until all instructions currently in the pipeline (i.e., all instructions that have been issued but have not completed) complete execution.

**Exponent.** The component of a binary floating-point number that normally signifies the integer power to which two is raised in determining the value of the represented number. Occasionally the exponent is called the signed or unbiased exponent.

**F** **Fetch Serialization.** During fetch serialization, instruction fetch is halted until all instructions currently in the processor (i.e., in the pipeline or in the pre-fetch queue) have completed. Following fetch serialization, the machine is said to be completely synchronized.

**Floating-Point Unit.** The functional unit in the RCPUs responsible for executing all floating-point arithmetic instructions.

**Flow-Control Instruction.** One of the following: **b**, **br**, **bcr**, **bcc**, **rfi**, **sc**, or (in some cases) **isync**.

**Fraction.** The field of the significand that lies to the right of its implied binary point.

## G

**General-Purpose Registers.** Any of the 32 registers in the MPC601 register file. These registers provide the source operands and destination results for all MPC601 data manipulation instructions. Load instructions move data from memory to registers, and store instructions move data from registers to memory.

## I

**IEEE 754.** A standard written by the Institute of Electrical and Electronics Engineers that defines operations of binary floating-point arithmetic and representations of binary floating-point numbers.

**I-Bus.** Internal instruction bus connecting the processor to instruction memory.

**Implementation Specific.** An RCPU register, exception, or other feature is said to be implementation specific if it is not part of the PowerPC architecture.

**Instruction Completion.** Completion of the instruction issue, execution, and writeback stages. An instruction is ready to be retired if it completes without generating an exception and all instructions ahead of it in the history buffer have completed without generating an exception.

**Instruction Execution Time.** The number of clock cycles between the time an instruction is taken and the time it is completed.

**Instruction Fetch.** The process of reading the instruction data received from the instruction memory.

**Instruction Issue.** The process of driving valid instruction bits inside the processor. The instruction is decoded by each execution unit, and the appropriate execution unit prepares to execute the instruction during the next clock cycle.

**Instruction Taken.** An instruction is taken after it has been issued and recognized by the appropriate execution unit. All resources to perform the instruction are ready, and the processor begins to execute it.

**Instruction Unit.** The functional unit in the RCPU that fetches all instructions from memory and performs the initial stages of instruction decoding. The instruction unit also contains the branch processing unit and performs all instruction address calculations (including branch address calculations).

**Integer Unit.** The functional unit in the RCPU responsible for executing all integer arithmetic instructions.

# Freescale Semiconductor, Inc.

**Instruction Cache.** High-speed memory containing recently accessed instructions (subset of main memory).

**Interrupt.** An external signal that causes the processor to suspend current execution and take a predefined exception.

## L

**L-Bus.** Internal load/store bus connecting the processor to internal modules and data memory and to the external bus interface.

**Latency.** The number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

**Little-Endian.** A byte-ordering method in memory where the address *n* of a word corresponds to the least significant byte. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the most significant byte.

## N

**NaN.** Not a number; a symbolic entity encoded in floating-point format. There are two types of NaNs — signaling NaNs and quiet NaNs.

**No-Op.** No-operation. A single-cycle operation that does not affect registers or generate bus activity.

## O

**Overflow.** An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are added, the sum may require 33 bits due to carry. Since the 32-bit registers of the MPC601 cannot represent this sum, an overflow condition occurs.

## P

**Park.** The act of allowing a bus master to maintain mastership of the bus without having to arbitrate.

**Pipelining.** A technique that breaks instruction execution into distinct steps so that multiple steps can be performed at the same time.

**Precise Exceptions.** The pipeline can be stopped so the instructions that preceded the faulting instruction can complete, and subsequent instructions can be executed from their beginning.

## Q

**Quiet NaNs.** Propagate through almost every arithmetic operation without signaling exceptions. These are used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid.

## S

**Sequential Instruction.** Any instruction other than a flow-control instruction or *isync*.

**Show Cycle.** An internal access (e.g., to an internal memory) reflected on the external bus using a special cycle (marked with a dedicated transfer code). For an internal memory “hit,” an address-only bus cycle is generated; for an internal memory “miss,” a complete bus cycle is generated.

**Signaling NaNs.** Signal the invalid operation exception when they are specified as arithmetic operands.

**Significand.** The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

**Slave.** The device addressed by a master device. The slave is identified in the address tenure and is responsible for supplying or latching the requested data for the master during the data tenure.

**Snooping.** Monitoring addresses driven by a bus master to detect the need for coherency actions.

**Static Branch Prediction.** Mechanism by which software (for example, compilers) can give a hint to the machine hardware about the direction the branch is likely to take.

**Supervisor Mode.** The privileged operation state of the RCPU. In supervisor mode, software can access all control registers and can access the supervisor memory space, among other privileged operations.

**T** **Tiny Result.** A tiny result is detected before rounding when a non-zero result value, computed as though the exponent range were unbounded, would be smaller in magnitude than the smallest normalized number.

**U** **Underflow.** An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result is a single-precision number. The result may require a larger exponent and/or mantissa than the single-precision format makes available. In other words, the result is too small to be represented accurately.

**User Mode.** The unprivileged operating state of the RCPU. In user mode, software can only access certain control registers and can only access user memory space. No privileged operations can be performed.

**W** **Watchpoint.** An event that, when detected, is reported but does not change the timing of the machine.



**SUMMARY OF CHANGES**

This is a complete revision, with complete reprint. All known errors in the publication have been corrected. The following summary lists significant changes.

**Section 2 Registers**

Page 2-5	Added Table 2-1 FPSCR Control, Status, and Sticky Bits.
Page 2-14	Added to the description of the BE bit in Table 2-8 Machine State Register Bit Settings.

**Section 4**

Page 4-62	Updated Table 4-30 Supervisor-Level SPR Encodings.
-----------	--

**Section 6 Exceptions**

Page 6-7	Modified Figure 6-1.
Page 6-14	Added to the description of the BE bit in Table 6-7 Machine State Register Bit Settings.
Page 6-19	Changed MSR[RI] to SRR1[RI] in second paragraph under 6.11.2.

**Section 7 Instruction Timing**

7-16	Corrected the syntax in the programming example in Section 7.7.1.
7-17	Corrected the syntax in the programming example in Section 7.7.2.
7-18	Corrected the syntax in the programming example in Section 7.7.3.
7-20	Corrected the syntax in the programming example in Section 7.7.4.
7-21	Corrected the syntax in the programming example in Section 7.7.5.
7-22	Corrected the syntax in the programming example in Section 7.7.6.
7-23	Corrected the syntax in the programming example in Section 7.7.7.

7-24 Corrected the syntax in the programming example in Section 7.7.8.



## Section 8 Development Support

Page 8-37 Revised the second paragraph of section 8.4.2.

Page 8-51 Updated Table 8-30 ICTRL Bit Settings to include the SER bit.

Page 8-57 Corrected the reset value for CHSTPE in Table 8-36 DER Bit Settings.

## Section 9 Instruction Set

Page 9-17 Corrected the RTL.

Page 9-18 Corrected the RTL.

Page 9-21 Made RTL consistent with RTL on pages 9 -25 and 9-27.

Page 9-22 to 9-24 Corrected Table 9-8 Simplified Mnemonics for bc, bca, bcl, and bcla Instructions.

Page 9-25 to 9-26 Corrected the RTL. Corrected Table 9-9 Simplified Mnemonics for bcctr and bcctrl Instructions.

Page 9-27 Corrected the RTL.

Page 9-28 Corrected Table 9-10 Simplified Mnemonics for bclr and bclrl Instructions.

Page 9-30 Revised the second paragraph of text. Updated Table 9-11 Simplified Mnemonics for cmp Instruction.

Page 9-31 Updated Table 9-12 Simplified Mnemonics for cmpi Instruction.

Page 9-32 Corrected the RTL. Updated Table 9-13 Simplified Mnemonics for cmpl Instruction.

Page 9-33 Corrected the RTL. Revised the second paragraph of text. Updated Table 9-14 Simplified Mnemonics for cmpli Instruction.

Page 9-34 Corrected the RTL.

Page 9-43 Updated text regarding setting OV bit.

Page 9-44 Updated text regarding division by 0.

Page 9-45 Updated text regarding setting LT, GT, and EQ bits.

Page 9-47 Corrected the RTL.

Page 9-53 Revised the second paragraph of text. Corrected Other Reg-



**Freescale Semiconductor, Inc.**

isters Altered. Added RTL.

Page 9-54	Revised the second text paragraph of text. Corrected Other Registers Altered.
Page 9-78	Corrected the RTL. Revised the third paragraph of text.
Page 9-79	Corrected the RTL. Revised the third paragraph of text.
Page 9-82	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-83	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-86	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-87	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-90	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-91	Corrected the RTL. Revised the third paragraph of text.
Page 9-94	Corrected the RTL.
Page 9-95	Corrected the RTL. Revised the third paragraph of text.
Page 9-96	Corrected the RTL. Revised the third paragraph of text.
Page 9-97	Corrected the RTL.
Page 9-98	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-99	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-100	Corrected the RTL.
Page 9-101	Corrected the RTL.
Page 9-102	Corrected the RTL.
Page 9-103	Corrected the RTL.
Page 9-104	Corrected the RTL. Revised the third paragraph of text.
Page 9-105	Corrected the RTL. Revised the third paragraph of text.
Page 9-106	Corrected the RTL.
Page 9-108	Revised the first paragraph of text.
Page 9-113	Corrected the RTL.
Page 9-115	Corrected the RTL.
Page 9-117	Corrected the RTL. Added Table 9-22 Simplified Mnemonics for mtrcf Instruction.

**Freescale Semiconductor, Inc.**

Page 9-121	Revised the first paragraph of text.
Page 9-123	Corrected the RTL and instruction encoding.
Page 9-125	Corrected the RTL.
Page 9-126	Corrected the RTL.
Page 9-127	Corrected the RTL.
Page 9-128	Corrected the RTL. Revised the first paragraph of text.
Page 9-130	Revised the first paragraph of text.
Page 9-142	Corrected the RTL. Revised the first paragraph of text.
Page 9-143	Corrected the RTL.
Page 9-144	Corrected the RTL.
Page 9-145	Corrected the RTL.
Page 9-146	Corrected the RTL.
Page 9-147	Corrected the RTL. Revised the third paragraph of text.
Page 9-148	Corrected the RTL. Revised the third paragraph of text.
Page 9-151	Corrected the RTL. Revised the third paragraph of text.
Page 9-152	Corrected the RTL. Revised the third paragraph of text.
Page 9-154	Corrected the RTL. Revised the second paragraph of text.
Page 9-156	Corrected the RTL. Revised the third paragraph of text.
Page 9-157	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-161	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-162	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-165	Corrected the RTL. Revised the third paragraph of text.
Page 9-166	Corrected the RTL. Revised the third and fourth paragraphs of text.
Page 9-170	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-171	Corrected the RTL. Revised the fourth paragraph of text.
Page 9-180	Corrected the RTL. Corrected Table 9-31 Simplified Mnemonics for Instructions.
Page 9-181	Corrected the RTL.

# **Freescale Semiconductor, Inc.**

## **Appendix A Instruction Set Listings**

A-1 to A-6	Corrected Table A-1 Complete Instruction List Sorted by Mnemonic.
------------	---

## **Appendix E Simplified Mnemonics**

E-2	Added caution note to section E.3.
E-9	Corrected Table E-8 Operands for Simplified Branch Mnemonics with Comparison Conditions.
E-14	Corrected encoding for no-op instruction.



## INDEX

## -A-

AA operand 4-49, 4-50, 4-51, 4-52, 9-2

add 4-6, 9-7

addc 4-6, 9-8

adde 4-7, 9-9

addi 4-6, 9-10, E-16

addic 4-6, 9-11, 9-12

addic. 4-6

addis 4-6, 9-13, E-16

addme 4-7, 9-14

Address calculation

branch instructions 4-49

Addressing

branch conditional relative 4-50

branch conditional to absolute 4-52

branch conditional to count register 4-53

branch conditional to link register 4-52

branch relative 4-49

branch to absolute 4-51

immediate index, floating-point 4-41

register indirect with immediate index, integer 4-30

register indirect with index, integer 4-31

register indirect, floating-point 4-41

register indirect, integer 4-32

Addressing modes 4-1

addze 4-8, 9-15

ADR field (in ICADR) 5-5

ALE 8-59

ALEE 8-61

Alignment exception 6-23

ALU-BFU 1-8, 7-5

and 4-13, 9-16

andc 4-14, 9-17

andi 4-13, 9-18

andis 4-13

andis. 9-19

Arithmetic instructions

floating point 4-19

integer 4-5

Asynchronous exceptions 6-4, 6-8

Atomic memory references

lwarx 9-101

stwcx. 4-65, 9-169, D-1

Atomic memory references

lwarx D-1

## -B-

b 4-56, 9-20

ba 4-56

Back trace 8-8

bc 4-57, 9-21

bca 4-57

bcctr 4-57, 9-25

bcctrl 4-57

bcl 4-57

bcla 4-57

bclr 4-57, 9-27

bclrl 4-57

BD operand 4-50, 9-2

BE bit 2-15, 6-15

BI operand 4-56, 9-2, E-5

Big-endian mode 3-2, 3-3

bl 4-56

bla 4-56

Blockage 7-11

BO operand 9-2, E-5

BO operand encodings 4-54

BPU 1-7

Branch

prediction 4-54

Branch folding 1-5, 7-1

example 7-22

Branch instructions 4-48

address calculation 4-49

condition register logical 4-57

description 4-56

EA generation 4-2

simplified mnemonics 4-59, E-4

Branch prediction E-10

and instruction cache 5-6

example 7-23

Branch processing unit 1-7

Branch trace enable 2-15, 6-15

Breakpoint counter A value and control register 8-57

Breakpoint counter B value and control register 8-58

Breakpoints 8-11

data 6-47

instruction 6-48

maskable external 6-49

non-maskable external 6-49

BRKNOMSK 8-56

Bus error 6-21

Byte ordering 3-2

BYTES field 2-11

## -C-

C bit 2-6

CA bit 2-11, 4-5

Cache, instruction. See Instruction cache

Carry 2-11

## Freescale Semiconductor, Inc.

Carry bit 4-5  
 CCER 5-4  
 CGBMSK 8-54  
 CHBMSK 8-54  
 Checkstop enable 6-21  
 Checkstop state 6-22  
     and debug mode 8-39  
 CHSTP bit 8-59  
 CHSTPE 8-61  
 CHSTPE bit 6-21  
 Clock mode, development port 8-26  
 CMD field 5-4  
 cmp 4-12, 9-30  
 CMPA–CMPD 8-50  
 CMPE–CMPF 8-50  
 CMPG–CMPH 8-51  
 cmpi 4-12, 9-31  
 cmpl 4-12, 9-32  
 cmpli 4-12, 9-33  
 CNTC 8-57  
 cntlzw 4-14, 9-34  
 CNTV 8-57  
 Comparator A–D value registers 8-50  
 Comparator E–F value registers 8-50  
 Comparator G–H value registers 8-51  
 Compare and swap D-3  
 Compare instructions  
     and condition register 2-10  
     floating point 4-27  
     integer 4-11  
     simplified mnemonics E-2  
 Compare size 8-54  
 Compare type 8-52, 8-54  
 Compare types 8-15  
 Complement register, simplified mnemonic E-17  
 Completed instructions 6-1  
 Condition register 1-12, 2-8  
     and compare instructions 2-10  
     logical instructions, simplified mnemonics E-11  
 Context synchronization 7-10  
 Conversions, floating point 4-25, C-1  
 Count register 2-12  
 COUNTA 8-57  
 COUNTB 8-58  
 CPU exception encoding 8-35  
 CR 1-12, 2-8, 2-12  
     and compare instructions 2-10  
     Move to/from 4-60  
 CR0 field 2-9  
 CR1 field 2-9  
 crand 4-58, 9-35  
 crandc 4-58, 9-36  
 crbA 9-2  
 crbB 9-2  
 crbD 9-2  
 creqv 4-58, 9-37  
 crfD 9-2  
 crfS 9-2  
 CRM 9-2  
 crnand 4-58, 9-38

crnor 4-58, 9-39  
 cror 4-58, 9-40  
 crorc 4-58, 9-41  
 CRWE 8-54  
 CRWF 8-54  
 crxor 4-58, 9-42  
 CSG 8-54  
 CSH 8-54  
 CTA 8-52  
 CTB 8-52  
 CTC 8-52  
 CTD 8-52  
 CTE 8-54  
 CTF 8-54  
 CTG 8-54  
 CTH 8-54

## –D–

DAE/source instruction service register 2-16, 6-24  
     settings for alignment exception 6-25  
 DAR 2-16  
 DAT field (in ICSDAT) 5-5  
 Data address register 2-16  
 Data alignment 3-1  
 Data breakpoint exception 6-47  
 Debug enable register 6-1, 8-60  
 Debug mode 6-1, 8-36  
     checkstop state 8-39  
     enabling 8-36  
     entering 8-37  
     exiting 8-39  
     program trace 8-6  
 DEC 2-17  
 DECE 8-59  
 DECEE 8-61  
 Decrementer exception 6-29  
 Decrementer register 2-17  
 Denormalization 3-17  
 Denormalized numbers 3-14  
 DER 6-1, 8-60  
 Development Port  
     trap enable selection 8-52  
 Development port 8-22  
     clock mode selection 8-26  
     input transmissions 8-32  
     ready bit 8-35  
     registers 8-25  
     serial data out 8-33  
     shift register 8-26  
     signals 8-23  
     transmission sequence 8-40  
     transmissions 8-31  
 Development serial clock 8-23  
 Development serial data in 8-24  
 Development serial data out 8-25  
 Development Support  
     SPRs 4-64  
 Development support 8-1  
     I-bus support 8-16  
     instruction cache 5-12

# Freescale Semiconductor, Inc.

L-bus support 8-17  
 registers 8-46  
 Dispatch stage 1-9, 7-4  
 divw 4-10, 9-43  
 divwu 4-11, 9-45  
 DIW0EN 8-52  
 DIW1EN 8-52  
 DIW2EN 8-52  
 DIW3EN 8-52  
 DLW0EN 8-56  
 DLW1EN 8-56  
 DPI 8-60  
 DPIR/DPDR input transmissions 8-32  
 DSCK 8-23  
 DSDI 8-24  
 DSDO 8-25  
 DSE 8-59  
 DSEE 8-61  
 DSISR 2-16, 6-24  
 settings for alignment exception 6-25

## –E–

EA 4-2  
 EBRK 8-60  
 ECR 8-58  
 EE bit 2-15, 2-20, 6-5, 6-10, 6-15  
 Effective address calculation 4-2  
 branches 4-49  
 loads and stores 4-30, 4-41  
 EID 2-21, 6-10  
 EIE 2-21, 6-10  
 eieio 4-65, 4-67, 9-46  
 Enabling debug mode 8-36  
 Entering debug mode 8-37  
 eqv 4-14, 9-47  
 Exception cause register 8-58  
 Exception little endian mode 6-15  
 Exception prefix 2-15, 6-5, 6-15  
 Exceptions 1-16, 6-1  
 alignment 6-23  
 asynchronous 6-4, 6-8  
 classes 6-2  
 data breakpoint 6-47  
 decremter 6-29  
 enabling and disabling 6-16  
 external interrupt 6-22  
 floating-point assist 6-31  
 floating-point unavailable 6-28  
 instruction breakpoint 6-48  
 little endian mode 2-15  
 machine check 6-21  
 maskable 6-5, 6-16  
 external breakpoint 6-49  
 non-maskable 6-5  
 external breakpoint 6-49  
 order and priority 6-10  
 ordered and unordered 6-2  
 precise 6-7  
 processing 6-13  
 program 6-26

recovery from 6-9  
 reset 6-20  
 software emulation 6-46  
 synchronous and precise 6-2  
 system call 6-29  
 timing 6-18  
 trace 6-30  
 vector table 6-5  
 Execute stage 1-9, 7-4  
 Execution serialization 7-9  
 Execution units 1-6  
 Exiting debug mode 8-39  
 EXP field 3-11  
 External interrupt 6-22  
 disable 2-21, 6-10  
 enable 2-15, 2-20, 2-21, 6-5, 6-10, 6-15  
 EXTI 8-59  
 EXTIE 8-61  
 extsb 4-14, 9-48  
 extsh 4-14, 9-49

## –F–

fabs 4-48, 9-50  
 fadd 4-19, 9-51, 9-52  
 fadds 4-20  
 fcmpo 4-28, 9-53  
 fcmpu 4-28, 9-54  
 fctiw 4-26, 9-55  
 fctiwz 4-27, 9-56  
 fdiv 4-21, 9-57, 9-58  
 fdivs 4-22  
 FE bits 2-15, 2-16, 6-15, 6-16  
 FE flag 2-6  
 Fetch and add D-2  
 Fetch and AND D-3  
 Fetch and no-op D-2  
 Fetch and store D-2  
 Fetch serialization 7-10  
 Fetch serialized 8-2  
 FEX bit 2-5, 6-36  
 FG bit 2-6  
 FI bit 2-6, 3-20, 6-37  
 FL bit 2-6  
 Floating-point  
 arithmetic instructions 4-19  
 assist exception 6-31, 6-32  
 available 2-15, 6-15  
 compare instructions 4-27  
 condition code 2-6  
 conversions C-1  
 data 3-10  
 data handling and precision 3-17  
 denormalized numbers 3-14  
 enabled exception summary 2-5, 6-36  
 enabled exceptions 6-36  
 equal or zero 2-6  
 exception mode 2-15, 2-16, 6-15, 6-16  
 exception summary 2-5, 6-36  
 exceptions cause register 6-34  
 execution models 3-21

+



I-bus  
     watchpoint programming 8-52  
 I-bus support 8-16  
     control register 8-51  
 I-cache. See Instruction cache  
 ICADR 2-21, 5-3, 5-5, 5-10  
 icbi 4-68, 5-7, 9-75  
 ICCST 2-21, 5-3  
 ICDAT 2-21, 5-3, 5-5, 5-11  
 ICTRL 8-51  
 IEEE operations, floating point 3-22  
 IEN bit 5-4  
 Ignore first match 8-52  
 IIFM 8-52  
 ILE bit 2-15, 6-15  
 IMM 9-2  
 IMUL-IDIV 1-8, 7-5  
 Inexact bit 3-25  
 Infinities 3-15  
 Input/output in little-endian mode 3-10  
 Instruction 9-1  
     blockage 7-11  
     breakpoint exception 6-48  
     cache 1-9  
         management instruction 4-68  
     completed 6-1  
     fields 9-1  
     flow 1-5, 7-1  
     issue 7-3  
     latency 7-11  
     memory addressing in little-endian mode 3-8  
     pipeline 1-9, 7-3  
     queue status pins 8-5  
     retired 1-5, 1-9, 6-7, 7-1  
     sequencer 1-5, 7-1  
     set 1-14  
     set summary 4-1  
     timing 1-9, 7-1, 7-3  
 Instruction cache 5-1  
     address register 2-21, 5-3, 5-5, 5-10  
     and branch prediction 5-6  
     and zero-wait-state memories 5-11  
     block invalidate 5-7  
     coherency 5-11  
     command field 5-4  
     commands 5-7  
     control and status register 2-21, 5-3  
     data path 5-3  
     data port 2-21, 5-3  
     data register 5-5, 5-11  
     debugging support 5-12  
     disable command 5-9  
     enable command 5-9  
     enable status bit 5-4  
     error types 5-4  
     hit 5-6  
     invalidate all 5-8  
     load & lock 5-8  
     miss 5-6  
     operation 5-5  
     organization 5-1  
     reading 5-10  
     reset sequence 5-11  
     SPRs 5-3  
     unlock all 5-9  
     unlock line 5-9  
 Instruction fetch  
     show cycle control 8-2, 8-53  
     show cycles 8-4  
 Instructions 4-58, 9-1  
     add 4-6, 9-7  
     addc 4-6, 9-8  
     adde 4-7, 9-9  
     addi 4-6, 9-10, E-16  
     addic 4-6, 9-11, 9-12  
     addic. 4-6  
     addis 4-6, 9-13, E-16  
     addme 4-7, 9-14  
     addze 4-8, 9-15  
     and 4-13, 9-16  
     andc 4-14, 9-17  
     andi 4-13, 9-18  
     andis 4-13  
     andis. 9-19  
     b 4-56, 9-20  
     ba 4-56  
     bc 4-57, 9-21  
     bca 4-57  
     bcctr 4-57, 9-25  
     bcctrl 4-57  
     bcl 4-57  
     bcla 4-57  
     bclr 4-57, 9-27  
     bclrl 4-57  
     bl 4-56  
     bla 4-56  
     branch 4-56  
     cmp 4-12, 9-30  
     cmpi 4-12, 9-31  
     cmpl 4-12, 9-32  
     cmpli 4-12, 9-33  
     cntlzw 4-14, 9-34  
     condition register logical 4-57  
     crand 4-58, 9-35  
     crandc 4-58, 9-36  
     creqv 4-58, 9-37  
     crnand 4-58, 9-38  
     crnor 4-58, 9-39  
     cror 4-58, 9-40  
     crorc 4-58, 9-41  
     crxor 9-42  
     divw 4-10, 9-43  
     divwu 4-11, 9-45  
     eieio 4-65, 4-67, 9-46  
     eqv 4-14, 9-47  
     extsb 4-14, 9-48  
     extsh 4-14, 9-49  
     fabs 4-48, 9-50  
     fadd 4-19, 9-51, 9-52  
     fadds 4-20

## Freescale Semiconductor, Inc.

fcmpo 4-28, 9-53  
 fcmppu 4-28, 9-54  
 fctiw 4-26, 9-55  
 fctiwz 4-27, 9-56  
 fdiv 4-21, 9-57, 9-58  
 fdivs 4-22  
 floating-point  
     arithmetic 4-19  
     compare 4-27  
     double-precision conversion, store 4-46  
     FP status and control register 4-28  
     move 4-47  
     multiply-add 4-22  
     rounding and conversion 4-25  
 flow control 4-48  
 fmadd 4-22, 9-59, 9-60  
 fmadds 4-23  
 fmr 4-48, 9-61  
 fmsub 4-23, 9-62, 9-63  
 fmsubs 4-23  
 fmul 4-21, 9-64, 9-65  
 fmuls 4-21  
 fnabs 4-48, 9-66  
 fneg 4-48, 9-67  
 fnmadd 4-24, 9-68, 9-69  
 fnmadds 4-24  
 fnmsub 4-25, 9-70, 9-71  
 fnmsubs 4-25  
 frsp 4-26, 9-72  
 fsub 4-20, 9-73, 9-74  
 fsubs 4-20  
 icbi 4-68, 9-75  
 instruction cache management 4-68  
 integer 4-4  
     compare 4-11  
     load 4-33  
     logical 4-12  
     move string 4-39  
     rotate and shift 4-14  
     store 4-36  
 integer arithmetic 4-5  
 isync 4-65, 4-67, 9-76  
 lbz 4-34, 9-77  
 lbzu 4-34, 9-78  
 lbzux 4-34, 9-79  
 lbzx 4-34, 9-80  
 lfd 4-43, 9-81  
 lfdx 4-43, 9-82  
 lfdux 4-43, 9-83  
 lfdx 4-43, 9-84  
 lfs 4-42, 9-85  
 lfsu 4-43, 9-86  
 lfsux 4-43, 9-87  
 lfsx 4-42, 9-88  
 lha 4-35, 9-89  
 lhau 4-35, 9-90  
 lhaux 4-35, 9-91  
 lhax 4-35, 9-92  
 lhbrx 4-38, 9-93  
 lhz 4-34, 9-94  
 lhzx 4-34, 9-95  
 lhzux 4-35, 9-96  
 lhzx 4-34, 9-97  
 lmw 4-39, 9-98  
 load  
     floating-point 4-42  
 load/store 4-30  
     address generation, floating-point 4-41  
     address generation, integer 4-30  
     integer load 4-33  
     integer multiple 4-38  
     with byte reversal 4-37  
 lswi 4-40, 9-99  
 lswx 4-40, 9-100  
 lwarx 4-67, 9-101  
 lwbrx 4-38, 9-102  
 lwz 4-35, 9-103  
 lwzu 4-35, 9-104  
 lwzux 4-36, 9-105  
 lwzx 4-35, 9-106  
 mcrf 4-58, 9-107  
 mcrfs 4-29, 9-108  
 mcrxr 4-61, 9-109  
 memory control 4-68  
 memory synchronization 4-65  
 mfcrr 4-61, 9-110  
 mffs 4-29, 9-111  
 mfmsr 4-61, 9-112  
 mfspr 4-62, 9-113  
 move to/from MSR and CR 4-60  
 move to/from SPR 4-61  
 mtrcrf 4-61, 9-115, 9-117  
 mtfbsb0 4-29, 9-118  
 mtfbsb1 4-29, 9-119  
 mtfbsf 4-29, 9-120, 9-137  
 mtfbsfi 4-29, 9-121  
 mtmsr 4-61, 9-122  
 mtspr 4-62, 9-123  
 mulhw 4-9, 9-125  
 mulhwu 4-9, 9-126  
 mulli 4-8, 9-127  
 mullw 4-9, 9-128  
 nand 4-13, 9-129  
 neg 4-8, 9-130  
 no-op E-16  
 nor 4-13, 9-131  
 or 4-13, 9-132  
 orc 4-14, 9-133  
 ori 4-13, 9-134  
 oris 4-13, 9-135  
 processor control 4-60  
 rfi 4-59, 9-136  
 rlwimi 4-17  
 rlwinm 4-16, 9-138  
 rlwnm 9-140  
 sc 4-59, 9-141  
 slw 4-18, 9-142  
 saw 4-18, 9-143  
 sawi 4-18, 9-144  
 srw 4-18, 9-145

- stb 4-37, 9-146
- stbu 4-37, 9-147
- stbux 4-37, 9-148
- stbx 4-37, 9-149
- stfd 9-150
- stfdu 9-151
- stfdx 9-152
- stfdx 9-153, 9-154
- stfs 9-155
- stfsu 9-156
- stfsux 9-157
- stfsx 9-158
- sth 4-37, 9-159
- sthbrx 4-38, 9-160
- sthu 4-37, 9-161
- sthux 4-37, 9-162
- sthx 4-37, 9-163
- stmw 4-39, 9-164
- store
  - floating-point 4-44
- stswi 4-40, 9-165
- stswx 4-40, 9-166
- stw 4-37, 9-167
- stwbrx 4-38, 9-168
- stwcx 4-67
- stwcx. 4-66, 9-169
- stwu 4-37, 9-170
- stwux 4-37, 9-171
- stwx 4-37, 9-172
- subf 4-6, 9-173
- subfc 4-7, 9-174
- subfe 4-7, 9-175
- subfic 4-6, 9-176
- subfme 4-7, 9-177
- subfze 4-8, 9-178
- sync 4-65, 4-68, 9-179
- system linkage 4-58
- trap 4-59
- tw 4-60, 9-180
- twi 4-60, 9-181
- xor 4-13, 9-182
- xori 4-13, 9-183
- xoris 4-13, 9-184
- Integer
  - instructions 4-4
    - arithmetic 4-5
    - compare 4-11
    - load 4-33
    - load/store 4-33
    - logical 4-12
    - move string 4-39
    - rotate and shift 4-14
    - store 4-36
- Integer exception register 2-10
- Invalidate all 5-8
- IP bit 2-15, 6-5, 6-15
- IRQ 6-22
- ISCTL 8-2, 8-53
- ISE 8-59
- ISEE 8-61
- Issued instructions 7-3
- isync 4-65, 4-67, 9-76
- IW 8-52
- L-
- Latency 7-11
- LBRK 8-60
- L-bus support 8-17
  - control register 1 8-53
  - control register 2 8-55
- lbz 4-34, 9-77
- lbzu 4-34, 9-78
- lbzux 4-34, 9-79
- lbzx 4-34, 9-80
- LCTRL1 8-53
- LCTRL2 8-55
- LE bit 2-15, 6-15
- lfd 4-43, 9-81
- lfdx 4-43, 9-82
- lfdx 4-43, 9-83
- lfdx 4-43, 9-84
- lfs 3-18, 4-42, 9-85
- lfsu 4-43, 9-86
- lfsux 4-43, 9-87
- lfsx 4-42, 9-88
- lha 4-35, 9-89
- lhau 4-35, 9-90
- lhaux 4-35, 9-91
- lhax 4-35, 9-92
- lhbrx 4-38, 9-93
- lhz 4-34, 9-94
- lhzu 4-34, 9-95
- lhux 4-35, 9-96
- lhux 4-34, 9-97
- LI 9-2
- LI operand 4-49
- Link register 2-11
- List insertion D-4
- Little endian mode 2-15
- Little-endian mode 3-2, 6-15
  - input/output 3-10
  - instruction memory addressing 3-8
  - load and store multiple instructions 3-8
  - misaligned operands 3-6
  - misaligned scalars 3-6
  - string operations 3-7
- LK 9-2
- lmw 4-39, 9-98
- Load
  - address, simplified mnemonic E-16
  - immediate, simplified mnemonic E-16
  - instructions 4-30
    - floating point 4-42
    - integer 4-33
- Load & lock 5-8
- Load floating-point single-precision 3-18
- Load/store
  - address generation 4-30
  - integer 4-30
  - byte reverse instructions 4-37

## Freescale Semiconductor, Inc.

multiple instructions, integer 4-38  
 Load/store multiple instructions  
   in little-endian mode 3-8  
 Loadstore  
   address generation 4-2  
 Logical instructions, integer 4-12  
 LR 2-11  
 lswi 4-40, 9-99  
 lswx 4-40, 9-100  
 LW0EN 8-55  
 LW0IA 8-55  
 LW0IADC 8-55  
 LW0LA 8-55  
 LW0LADC 8-55  
 LW0LD 8-55  
 LW0LDDC 8-55  
 LW1EN 8-55  
 LW1IA 8-56  
 LW1IADC 8-56  
 LW1LA 8-56  
 LW1LADC 8-56  
 LW1LD 8-56  
 LW1LDDC 8-56  
 lwarx 4-67, 9-101, D-1  
 lwbrx 4-38, 9-102  
 lwz 4-35, 9-103  
 lwzu 4-35, 9-104  
 lwzux 4-36, 9-105  
 lwzx 4-35, 9-106

## -M-

Machine check  
   enable 2-15, 6-15, 6-21  
   exception 6-21  
 machine check  
   exception  
     enable 6-21  
 Machine state register 1-12, 2-13, 6-14  
 Machine status save/restore register 0 2-18  
 Machine status save/restore register 1 2-19  
 Maskable  
   exceptions 6-5  
   external breakpoints 6-49  
 Maskable exceptions 6-16  
 MB 9-2  
 MCE 8-59  
 MCEE 8-61  
 MCIE bit 6-21  
 mcrf 4-58, 9-107  
 mcrfs 4-29, 9-108  
 mcrxr 4-61, 9-109  
 ME bit 2-15, 6-15, 6-21, 9-2  
 Memory  
   control instructions 4-68  
   operands 4-2  
   organization 3-1  
   synchronization  
     eieio 4-65  
     isync 4-65  
     stwcx. 4-66

sync 4-65  
 mfcrr 4-61, 9-110  
 mffs 4-29, 9-111  
 mfmsr 4-61, 9-112  
 mfspr 4-62, 9-113  
 Misaligned operands  
   little-endian mode 3-6  
 Miss, instruction cache 5-6  
 Move  
   instructions, floating point 4-47  
   register, simplified mnemonic E-17  
   string instructions, integer 4-39  
   to/from SPR Instructions 4-61  
 MSR 1-12, 2-13, 6-14  
   Move to/from 4-60  
 mtrcrf 4-61, 9-115, 9-117  
 mtfsb0 4-29, 9-118  
 mtfsb1 4-29, 9-119  
 mtfsf 4-29, 9-120, 9-137  
 mtfsfi 4-29, 9-121  
 mtmsr 4-61, 9-122  
 mtspr 4-62, 9-123  
 mulhw 4-9, 9-125  
 mulhwu 4-9, 9-126  
 mulli 4-8, 9-127  
 mullw 4-9, 9-128  
 Multiple-precision shifts B-1  
 Multiply-add instructions, floating point 3-24, 4-22

## -N-

nand 4-13, 9-129  
 NaNs 3-15  
 NB 9-3  
 neg 4-8, 9-130  
 NI bit 2-7, 3-25, 6-38  
 Non-IEEE floating-point operation 2-7, 3-25, 6-38  
 Non-maskable  
   exceptions 6-5  
     external breakpoint 6-49  
 Non-recoverable interrupt 2-21, 6-10  
 No-op E-16  
 nor 4-13, 9-131  
 Normalized numbers 3-13, 3-17  
 Not a Numbers 3-15  
 NRI 2-21, 6-10  
 Null output encoding 8-35

## -O-

OE bit 2-7, 6-38, 9-3  
 or 4-13, 9-132  
 orc 4-14, 9-133  
 Ordered exceptions 6-2  
 ori 4-13, 9-134  
 oris 4-13, 9-135  
 OV (overflow) bit 2-11, 4-5  
 OX bit 2-6, 6-36

## Freescale Semiconductor, Inc.

## -P-

Pipeline, instruction 1-9, 7-3  
 PR 2-1  
 PR bit 2-15, 6-15  
 PRE 8-59  
 Precise exceptions 6-2, 6-7  
 PREE 8-61  
 Privilege level 2-1  
 Privilege levels 2-15, 6-15  
 Process switching 6-18  
 Processor control instructions 4-60  
 Processor version register 2-20  
 Program 6-26  
   exception 6-26  
   flow tracking 8-1  
   flow-tracking  
     status pins 8-5  
   trace  
     back 8-8  
     in debug mode 8-6  
     window 8-8  
 Programming models 2-1  
 PVR 2-20

## -R-

R bit 3-22  
 rA 9-3  
 rB 9-3  
 Rc 9-3  
 RCPU  
   execution units 1-6  
   registers 2-1  
 rD 9-3  
 RE bit 6-15  
 Ready bit, development port 8-35  
 Recoverable exception 2-15, 2-20, 6-5, 6-9, 6-10, 6-15  
 Register transfer language 9-3  
 Registers 2-1  
   CMPA-CMPD 8-50  
   CMPE-CMPF 8-50  
   CMPG-CMPH 8-51  
   COUNTA 8-57  
   COUNTB 8-58  
   CR 2-8, 2-12  
   DAR 2-16  
   DEC 2-17  
   DER 6-1, 8-60  
   development port 8-25  
   development support 8-46  
   development support shift register 8-26  
   DSISR 6-24  
   ECR 8-58  
   FPRs 2-3  
   FPSCR 2-4, 6-36  
   GPRs 2-3  
   ICADR 2-21, 5-3, 5-10  
   ICCST 2-21, 5-3  
   ICDAT 2-21, 5-3, 5-11  
   ICTRL 8-51

LCTRL1 8-53  
 LCTRL2 8-55  
 LR 2-11  
 MSR 2-13, 6-14  
 PVR 2-20  
 SPRGs 2-19  
 SRR0 2-18  
 SRR1 2-19  
 supervisor level 2-13  
 TB 2-12  
 TECR 8-26  
 user level 2-3  
 XER 2-10

Reset exception 6-20  
   and instruction cache 5-11  
 Retired instructions 1-5, 1-9, 6-7, 7-1  
 Retirement stage 1-9, 7-4  
 rfi 4-59, 6-18, 9-136  
 RI bit 2-15, 2-20, 6-5, 6-9, 6-10  
 rlwimi 4-17  
 rlwinm 4-16, 9-138  
 rlwnm 9-140  
 RN field 2-7, 6-38  
 Rotate and shift instructions 4-14  
   simplified mnemonics E-3  
 Round bit 3-22  
 Round to floating-point single-precision 3-18  
 Rounding, floating point 3-19, 4-25  
 rS 9-3  
 RTL 9-3

## -S-

S (sign bit) 3-11  
 sc 4-59, 9-141  
 SE bit 2-15, 6-15  
 SEE 8-59  
 Sequencer 1-5, 7-1  
 Sequencing error encoding 8-34  
 Serialization 7-9  
   execution 7-9  
   fetch 7-10, 8-2  
 SH 9-3  
 Shift instructions, integer 4-14  
 Shifts, multiple precision B-1  
 Show cycles 8-4  
 SIE mode 6-34  
 SIMM 9-3  
 Simplified mnemonics 4-68, E-1  
   branch instructions E-4  
   compare instructions E-2  
   condition register logical instructions E-11  
   miscellaneous 4-68  
   rotate and shift instructions E-3  
   special-purpose registers E-15  
   SPRs E-15  
   subtract instructions E-2  
   trap instructions E-12  
 Single-precision floating point 3-18  
 Single-step trace enable 2-15, 6-15  
 SIW0EN 8-52

- SIW1EN 8-52
- SIW2EN 8-52
- SIW3EN 8-52
- slw 4-18, 9-142
- SLW0EN 8-56
- SLW1EN 8-56
- SO bit 2-11, 4-5
- Software emulation exception 6-46
- Software envelope 3-25, 6-31
- Software monitor support 8-46
- Software trap enable selection 8-52
- Special-purpose registers 1-12
  - general 2-19
  - simplified mnemonics E-15
  - supervisor level 1-13, 4-63
  - user level 1-12, 4-62
- Split field notation 9-1
- SPR 9-3
- SPRG0–SPRG3 2-19
- SPRGs 2-19
- SPRs
  - development support 4-64
  - general 2-19
  - instruction cache 5-3
  - simplified mnemonics E-15
- SPRs. See Special-purpose registers
- sraw 4-18, 9-143
- srawi 4-18, 9-144
- SRR0 2-18
- SRR1 2-19
- srw 4-18, 9-145
- stb 4-37, 9-146
- stbu 4-37, 9-147
- stbux 4-37, 9-148
- stbx 4-37, 9-149
- stfd 9-150
- stfdu 9-151
- stfdx 9-152
- stfdx 9-153, 9-154
- stfs 3-18, 9-155
- stfsu 9-156
- stfsux 9-157
- stfsx 9-158
- sth 4-37, 9-159
- sthbrx 4-38, 9-160
- sthu 4-37, 9-161
- sthux 4-37, 9-162
- sthx 4-37, 9-163
- Sticky bit 3-22
- stmw 4-39, 9-164
- Store
  - floating-point single-precision 3-18
- Store instructions 4-30
  - floating point 4-44
  - integer 4-36
- String instructions, timing 7-8
- String operations
  - in little-endian mode 3-7
- stswi 4-40, 9-165
- stswx 4-40, 9-166
- stw 4-37, 9-167
- stwbrx 4-38, 9-168
- stwcx 4-67
- stwcx. 4-66, 9-169, D-1
- stwu 4-37, 9-170
- stwux 4-37, 9-171
- stwx 4-37, 9-172
- subf 4-6, 9-173
- subfc 4-7, 9-174
- subfe 4-7, 9-175
- subfic 4-6, 9-176
- subfme 4-7, 9-177
- subfze 4-8, 9-178
- Subtract instructions, simplified mnemonics E-2
- Summary overflow 2-11, 4-5
- Supervisor
  - privilege level 2-1
- Supervisor level
  - registers 2-13
  - returning from 6-18
  - SPRs 1-13
- Supervisor-Level
  - SPRs 4-63
- SUSG 8-54
- SUSH 8-54
- sync 4-65, 4-68, 9-179
- Synchronization
  - context 7-10
  - primitives D-2
  - programming examples D-1
- Synchronized ignore exceptions 6-34
- Synchronous exceptions 6-2
  - ordering 6-12
- SYSE 8-59
- SYSEE 8-61
- System call exception 6-29
- System linkage instructions 4-58

-T-

- TB 2-12
- TBL 2-13, 2-17
- TBU 2-13, 2-17
- TEA 6-21
- TECR 8-26
- Test and set D-3
- Time base 2-12
- Timing, instruction 1-9
- TO 9-3
- TO operand 4-60
- TR 8-59
- Trace 8-8
  - window 8-8
- Trace exception 6-30
- Trap enable
  - input transmissions 8-32
  - programming 8-20
- Trap enable control register 8-26
- Trap instructions 4-59
  - simplified mnemonics E-12
- TRE 8-61

**Freescale Semiconductor, Inc.**

tw 4-60, 9-180

twi 4-60, 9-181

**-U-**

UIMM 9-3

UISA register set 2-3

Unlock all 5-9

Unlock line 5-9

Unordered exceptions 6-2

User

privilege level 2-1

User level

registers 2-3

SPRs 1-12, 4-62

UX bit 2-6, 6-36

**-V-**

Valid data encoding 8-34

VE bit 2-7, 6-38

Vector table, exceptions 6-5

VF signals 8-5

VFLS signals 8-6

VSYNC 8-10

VX 6-36

VX bit 2-5

VXCVI bit 2-7, 6-38

VXIDI 2-6

VXIDI bit 6-37

VXIMZ bit 2-6, 6-37

VXISI 2-6

VXISI bit 6-37

VXSNAN 2-6

VXSNAN bit 6-37

VXSOF bit 2-7, 6-37

VXSQRT bit 2-7, 6-37

VXVC bit 2-6, 6-37

VXZDZ bit 2-6, 6-37

**-W-**

Watchpoints 8-11

Window trace 8-8

Writeback stage 1-9, 6-1, 7-4

**-X-**

X bit 3-22

XE bit 2-7, 6-38

XER 2-10

XO 9-3

xor 4-13, 9-182

xori 4-13, 9-183

xoris 4-13, 9-184

XX bit 2-6, 6-37

**-Z-**

ZE bit 2-7, 6-38

Zero values 3-14

ZX bit 2-6, 6-37





**Freescale Semiconductor, Inc.**

**Freescale Semiconductor, Inc.**

**For More Information On This Product,  
Go to: [www.freescale.com](http://www.freescale.com)**

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

**To obtain additional product information:**

**USA/EUROPE:**

Motorola Literature Distribution;

P.O. Box 20912; Phoenix, Arizona 85036. 1-800-441-2447

**JAPAN:**

Nippon Motorola Ltd.; Tatsumi-SPD-JLDC, Toshikatsu Otsuki,

6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 03-3521-8315

**HONG KONG:**

Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,

51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298

**MFAX:**

RMFAX0@email.sps.mot.com - TOUCHTONE (602) 244-6609

**INTERNET:** <http://www.mot.com>



**MOTOROLA**

**For More Information On This Product,  
Go to: [www.freescale.com](http://www.freescale.com)**