



SiFive S76 Core Complex Manual

21G3.02.00

Copyright © 2019–2022 by SiFive, Inc. All rights reserved.

SiFive S76 Core Complex Manual

Proprietary Notice

Copyright © 2019–2022 by SiFive, Inc. All rights reserved.

SiFive S76 Core Complex Manual by SiFive, Inc. is licensed under Attribution-NonCommercialNoDerivatives 4.0 International. To view a copy of this license, visit: <http://creativecommons.org/licenses/by-nc-nd/4.0>

Information in this document is provided “as is,” with all faults.

SiFive expressly disclaims all warranties, representations, and conditions of any kind, whether express or implied, including, but not limited to, the implied warranties or conditions of merchantability, fitness for a particular purpose and non-infringement.

SiFive does not assume any liability rising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation indirect, incidental, special, exemplary, or consequential damages.

SiFive reserves the right to make changes without further notice to any products herein.

Contents

| | |
|--|----|
| List of Tables | 10 |
| List of Figures | 15 |
| 1 Introduction | 19 |
| 1.1 About this Document | 19 |
| 1.2 About this Release | 20 |
| 1.3 S76 Core Complex Overview | 20 |
| 1.4 S7 RISC-V Core | 21 |
| 1.5 Memory System..... | 22 |
| 1.6 Interrupts | 22 |
| 1.7 Debug Support | 22 |
| 1.8 Compliance | 22 |
| 2 List of Abbreviations and Terms | 24 |
| 3 S7 RISC-V Core | 26 |
| 3.1 Supported Privilege Modes | 26 |
| 3.2 Instruction Memory System..... | 26 |
| 3.2.1 Execution Memory Space | 27 |
| 3.2.2 L1 Instruction Cache..... | 27 |
| 3.2.3 Cache Maintenance..... | 27 |
| 3.2.4 Instruction Tightly-Integrated Memory (ITIM) | 27 |
| 3.2.5 Instruction Fetch Unit..... | 28 |
| 3.2.6 Branch Prediction | 28 |
| 3.3 Execution Pipeline | 29 |
| 3.4 Data Memory System..... | 31 |
| 3.4.1 L1 Data Cache | 31 |
| 3.4.2 Cache Maintenance Operations..... | 31 |

| | | |
|----------|--|-----------|
| 3.4.3 | Data Local Store (DLS) | 32 |
| 3.5 | Fast I/O | 32 |
| 3.6 | Atomic Memory Operations | 32 |
| 3.7 | Floating-Point Unit (FPU) | 32 |
| 3.8 | Physical Memory Protection (PMP) | 33 |
| 3.8.1 | PMP Functional Description | 33 |
| 3.8.2 | PMP Region Locking | 34 |
| 3.8.3 | PMP Registers | 34 |
| 3.8.4 | PMP and PMA | 36 |
| 3.8.5 | PMP Programming Overview | 36 |
| 3.8.6 | PMP and Paging | 38 |
| 3.8.7 | PMP Limitations | 38 |
| 3.8.8 | Behavior for Regions without PMP Protection | 38 |
| 3.8.9 | Cache Flush Behavior on PMP Protected Region | 39 |
| 3.9 | Hardware Performance Monitor | 39 |
| 3.9.1 | Performance Monitoring Counters Reset Behavior | 39 |
| 3.9.2 | Fixed-Function Performance Monitoring Counters | 39 |
| 3.9.3 | Event-Programmable Performance Monitoring Counters | 40 |
| 3.9.4 | Event Selector Registers | 40 |
| 3.9.5 | Event Selector Encodings | 42 |
| 3.9.6 | Counter-Enable Registers | 44 |
| 3.10 | Ports | 44 |
| 3.10.1 | Front Port | 44 |
| 3.10.2 | Memory Port | 45 |
| 3.10.3 | Peripheral Port | 45 |
| 3.10.4 | System Port | 45 |
| 4 | Physical Memory Attributes and Memory Map | 46 |
| 4.1 | Physical Memory Attributes Overview | 46 |
| 4.2 | Memory Map | 47 |
| 5 | Programmer's Model | 49 |
| 5.1 | Base Instruction Formats | 49 |

| | | |
|--------|--|----|
| 5.2 | RV64I Base Integer Instruction Set | 50 |
| 5.2.1 | R-Type (Register-Based) Integer Instructions | 51 |
| 5.2.2 | I-Type Integer Instructions | 52 |
| 5.2.3 | I-Type Load Instructions | 53 |
| 5.2.4 | S-Type Store Instructions | 54 |
| 5.2.5 | Unconditional Jumps | 55 |
| 5.2.6 | Conditional Branches | 56 |
| 5.2.7 | Upper-Immediate Instructions | 57 |
| 5.2.8 | Memory Ordering Operations | 57 |
| 5.2.9 | Environment Call and Breakpoints | 58 |
| 5.2.10 | NOP Instruction | 58 |
| 5.3 | M Extension: Multiplication Operations | 58 |
| 5.3.1 | Division Operations | 59 |
| 5.4 | A Extension: Atomic Operations | 59 |
| 5.4.1 | Atomic Load-Reserve and Store-Conditional Instructions | 59 |
| 5.4.2 | Atomic Memory Operations (AMOs) | 60 |
| 5.5 | F Extension: Single-Precision Floating-Point Instructions | 61 |
| 5.5.1 | Floating-Point Control and Status Registers | 61 |
| 5.5.2 | Rounding Modes | 62 |
| 5.5.3 | Single-Precision Floating-Point Load and Store Instructions | 62 |
| 5.5.4 | Single-Precision Floating-Point Computational Instructions | 63 |
| 5.5.5 | Single-Precision Floating-Point Conversion and Move Instructions | 63 |
| 5.5.6 | Single-Precision Floating-Point Compare Instructions | 66 |
| 5.6 | D Extension: Double-Precision Floating-Point Instructions | 67 |
| 5.6.1 | Double-Precision Floating-Point Load and Store Instructions | 67 |
| 5.6.2 | Double-Precision Floating-Point Computational Instructions | 68 |
| 5.6.3 | Double-Precision Floating-Point Conversion and Move Instructions | 69 |
| 5.6.4 | Double-Precision Floating-Point Compare Instructions | 72 |
| 5.6.5 | Double-Precision Floating-Point Classify Instruction | 73 |
| 5.7 | C Extension: Compressed Instructions | 73 |
| 5.7.1 | Compressed 16-bit Instruction Formats | 73 |
| 5.7.2 | Stack-Pointed-Based Loads and Stores | 74 |
| 5.7.3 | Register-Based Loads and Stores | 75 |

| | | |
|--------|--|-----|
| 5.7.4 | Control Transfer Instructions..... | 76 |
| 5.7.5 | Integer Computational Instructions..... | 77 |
| 5.8 | Zba Extension: Address Calculation Instructions..... | 80 |
| 5.8.1 | Address Calculation Instructions..... | 80 |
| 5.8.2 | Add/Shift with Prefix Zero-Extend Instructions..... | 81 |
| 5.9 | Zbb Extension: Basic Bit Manipulation Instructions..... | 81 |
| 5.9.1 | Count Leading/Trailing Zeroes Instructions..... | 81 |
| 5.9.2 | Count Population Instructions..... | 82 |
| 5.9.3 | Logic-With-Negate Instructions..... | 82 |
| 5.9.4 | Comparison Instructions..... | 83 |
| 5.9.5 | Sign-Extend and Zero-Extend Instructions..... | 83 |
| 5.9.6 | Bitwise Rotation Instructions..... | 84 |
| 5.9.7 | OR Combine Instruction..... | 85 |
| 5.9.8 | Byte-Reverse Instruction..... | 85 |
| 5.10 | Zicsr Extension: Control and Status Register Instructions..... | 85 |
| 5.10.1 | Control and Status Registers..... | 87 |
| 5.10.2 | Defined CSRs..... | 87 |
| 5.10.3 | CSR Access Ordering..... | 90 |
| 5.10.4 | SiFive RISC-V Implementation Version Registers..... | 90 |
| 5.10.5 | Custom CSRs..... | 93 |
| 5.11 | Base Counters and Timers..... | 93 |
| 5.11.1 | Timer Register..... | 94 |
| 5.11.2 | Timer API..... | 94 |
| 5.12 | Privileged Instructions..... | 95 |
| 5.12.1 | Machine-Mode Privileged Instructions..... | 95 |
| 5.13 | ABI - Register File Usage and Calling Conventions..... | 96 |
| 5.13.1 | RISC-V Assembly..... | 98 |
| 5.13.2 | Assembler to Machine Code..... | 98 |
| 5.13.3 | Calling a Function (Calling Convention)..... | 100 |
| 5.14 | Memory Ordering - FENCE Instructions..... | 102 |
| 5.15 | Boot Flow..... | 103 |
| 5.16 | Linker File..... | 104 |
| 5.16.1 | Linker File Symbols..... | 105 |

| | | |
|----------|--|------------|
| 5.17 | RISC-V Compiler Flags | 107 |
| 5.17.1 | arch, abi, and mtune..... | 107 |
| 5.18 | Compilation Process | 110 |
| 5.19 | Large Code Model Workarounds | 110 |
| 5.19.1 | RISC-V Code Model Summary | 111 |
| 5.19.2 | Enabling the Compact Code Model | 111 |
| 5.20 | Pipeline Hazards..... | 113 |
| 5.20.1 | Read-After-Write Hazards | 113 |
| 5.20.2 | Write-After-Write Hazards..... | 113 |
| 5.21 | Reading CSRs..... | 113 |
| 6 | Custom Instructions and CSRs | 115 |
| 6.1 | SiFive Custom Instructions | 115 |
| 6.1.1 | CFLUSH.D.L1 | 115 |
| 6.1.2 | CDISCARD.D.L1..... | 115 |
| 6.1.3 | CEASE..... | 116 |
| 6.1.4 | PAUSE..... | 116 |
| 6.2 | SiFive Custom CSRs..... | 117 |
| 6.2.1 | Branch Prediction Mode CSR | 117 |
| 6.2.2 | SiFive Feature Disable CSR..... | 117 |
| 6.2.3 | Power Dial CSR | 119 |
| 6.3 | Other Custom Instructions and CSRs | 119 |
| 7 | Interrupts and Exceptions | 120 |
| 7.1 | Interrupt Concepts | 120 |
| 7.2 | Exception Concepts | 120 |
| 7.3 | Trap Concepts | 122 |
| 7.4 | Interrupt Block Diagram | 123 |
| 7.5 | Local Interrupts..... | 123 |
| 7.6 | Interrupt Operation..... | 124 |
| 7.6.1 | Interrupt Entry and Exit | 124 |
| 7.7 | Interrupt Control and Status Registers | 125 |
| 7.7.1 | Machine Status Register (mstatus)..... | 125 |

| | | |
|----------|---|------------|
| 7.7.2 | Machine Trap Vector (mtvec)..... | 125 |
| 7.7.3 | Machine Interrupt Enable (mie)..... | 127 |
| 7.7.4 | Machine Interrupt Pending (mip)..... | 127 |
| 7.7.5 | Machine Cause (mcause) | 127 |
| 7.7.6 | Minimum Interrupt Configuration | 128 |
| 7.8 | Interrupt Priorities | 129 |
| 7.9 | Interrupt Latency..... | 129 |
| 7.10 | Non-Maskable Interrupt | 129 |
| 7.10.1 | Handler Addresses | 129 |
| 7.10.2 | RNMI CSRs | 130 |
| 7.10.3 | MNRET Instruction | 131 |
| 7.10.4 | RNMI Operation | 131 |
| 8 | Core-Local Interruptor (CLINT)..... | 132 |
| 8.1 | CLINT Priorities and Preemption | 132 |
| 8.2 | CLINT Vector Table | 133 |
| 8.3 | CLINT Interrupt Sources | 135 |
| 8.4 | CLINT Interrupt Attribute..... | 135 |
| 8.5 | CLINT Memory Map | 135 |
| 8.6 | Register Descriptions | 136 |
| 8.6.1 | MSIP Registers | 136 |
| 8.6.2 | Timer Registers..... | 136 |
| 9 | Platform-Level Interrupt Controller (PLIC) | 137 |
| 9.1 | Memory Map | 137 |
| 9.2 | Interrupt Sources | 138 |
| 9.3 | Interrupt Priorities | 139 |
| 9.4 | Interrupt Pending Bits..... | 139 |
| 9.5 | Interrupt Enables | 140 |
| 9.6 | PLIC Clock Gate Disable | 141 |
| 9.7 | Priority Thresholds | 142 |
| 9.8 | Interrupt Claim Process | 142 |
| 9.9 | Interrupt Completion..... | 142 |

| | | |
|-----------|--|------------|
| 9.10 | Example PLIC Interrupt Handler | 143 |
| 10 | TileLink Error Device | 145 |
| 11 | Power Management..... | 146 |
| 11.1 | Power Modes | 146 |
| 11.2 | Run Mode | 146 |
| 11.2.1 | Power Control | 146 |
| 11.3 | WFI Clock Gate Mode | 147 |
| 11.3.1 | WFI Wake Up..... | 147 |
| 11.4 | CEASE Instruction for Power Down | 148 |
| 11.5 | Subsystem LowPower Controller (SLPC) | 148 |
| 11.6 | Composable Cache Clock Gating | 149 |
| 11.7 | Hardware Reset..... | 150 |
| 11.8 | Early Boot Flow..... | 151 |
| 11.9 | Interrupt State During Early Boot | 151 |
| 11.10 | Other Boot Time Considerations | 152 |
| 11.11 | Power-Down Flow | 152 |
| 12 | Debug | 154 |
| 12.1 | Debug Module | 154 |
| 12.2 | Debug and Trigger Registers..... | 157 |
| 12.2.1 | Debug Control and Status Register (dcsr) | 157 |
| 12.2.2 | Debug PC (dpc) | 158 |
| 12.2.3 | Debug Scratch (dscratch)..... | 158 |
| 12.2.4 | Trigger Select Register (tselect) | 159 |
| 12.2.5 | Trigger Data Registers (tdata1-3)..... | 159 |
| 12.3 | Breakpoints | 160 |
| 12.3.1 | Breakpoint Match Control Register (mcontrol016)..... | 160 |
| 12.3.2 | Breakpoint Match Address Register (maddress) | 163 |
| 12.3.3 | Breakpoint Execution | 163 |
| 12.3.4 | Sharing Breakpoints Between Debug and Machine Mode | 164 |
| 12.4 | Debug Memory Map..... | 164 |

| | | |
|----------|---|------------|
| 12.4.1 | Debug RAM and Program Buffer (0x300–0x3FF) | 164 |
| 12.4.2 | Debug ROM (0x800–0xFFF) | 164 |
| 12.4.3 | Debug Flags (0x100–0x110, 0x400–0x7FF) | 164 |
| 12.4.4 | Safe Address | 165 |
| 12.5 | Debug Module Interface..... | 165 |
| 12.5.1 | Debug Module Status Register (dmstatus) | 166 |
| 12.5.2 | Debug Module Control Register (dmcontrol)..... | 168 |
| 12.5.3 | Hart Info Register (hartinfo)..... | 169 |
| 12.5.4 | Abstract Control and Status Register (abstractcs)..... | 171 |
| 12.5.5 | Abstract Command Register (command) | 172 |
| 12.5.6 | Abstract Command Autoexec Register (abstractauto)..... | 172 |
| 12.5.7 | Debug Module Control and Status 2 Register (dmcs2)..... | 172 |
| 12.5.8 | Abstract Commands | 173 |
| 12.5.9 | System Bus Access | 175 |
| 12.6 | Debug Module Operational Sequences | 175 |
| 12.6.1 | Entering Debug Mode | 175 |
| 12.6.2 | Exiting Debug Mode | 176 |
| A | SiFive Core Complex Configuration Options..... | 177 |
| A.1 | S7 Series..... | 177 |
| B | SiFive RISC-V Implementation Registers..... | 181 |
| B.1 | Machine Architecture ID Register (marchid) | 181 |
| B.2 | Machine Implementation ID Register (mimpid) | 182 |
| C | SiFive Custom CSRs | 183 |
| D | Floating-Point Unit Instruction Timing | 184 |
| D.1 | S7 Floating-Point Instruction Timing | 184 |
| E | Revision History | 187 |
| F | Knowledge Base Articles | 188 |

References 189

Tables

| | | |
|-----------------|--|----|
| Table 1 | S76 Core Complex Feature Set | 19 |
| Table 2 | RISC-V Specification Compliance | 23 |
| Table 3 | Abbreviations and Terms..... | 25 |
| Table 4 | S7 Feature Set..... | 26 |
| Table 5 | Executable Memory Regions for the S76 Core Complex | 27 |
| Table 6 | S7 Instruction Latency | 30 |
| Table 7 | pmpXcfg Bitfield Description | 35 |
| Table 8 | pmpaddrX Encoding Examples for A=NAPOT..... | 36 |
| Table 9 | mhpmeventX Register Bit Layout for Overflow and Filtering | 41 |
| Table 10 | mhpmevent Register | 43 |
| Table 11 | Physical Memory Attributes for External Regions..... | 47 |
| Table 12 | Physical Memory Attributes for Internal Regions..... | 47 |
| Table 13 | S76 Core Complex Memory Map. Physical Memory Attributes: R –Read, W –Write, X –Execute, I –Instruction Cacheable, D –Data Cacheable, A –Atomics..... | 48 |
| Table 14 | Base Instruction Formats | 49 |
| Table 15 | R-Type Integer Instructions..... | 51 |
| Table 16 | R-Type Integer Instruction Description | 51 |
| Table 17 | I-Type Integer Instructions | 52 |
| Table 18 | I-Type Integer Instruction Description | 53 |
| Table 19 | I-Type Load Instructions | 54 |
| Table 20 | I-Type Load Instruction Description | 54 |
| Table 21 | S-Type Store Instructions | 55 |
| Table 22 | S-Type Store Instruction Description | 55 |
| Table 23 | J-Type Instruction Description..... | 56 |
| Table 24 | B-Type Instructions..... | 56 |
| Table 25 | B-Type Instruction Description | 56 |
| Table 26 | RISC-V Base Instruction to Assembly Pseudoinstruction Example | 57 |
| Table 27 | Multiplication Operation Description | 58 |
| Table 28 | Division Operation Description..... | 59 |

| | | |
|-----------------|---|----|
| Table 29 | Atomic Load-Reserve and Store-Conditional Instruction Description..... | 60 |
| Table 30 | Atomic Memory Operation Description..... | 61 |
| Table 31 | Accrued Exception Flags..... | 61 |
| Table 32 | Floating-Point Rounding Modes | 62 |
| Table 33 | Single-Precision FP Load and Store Instructions Description | 62 |
| Table 34 | Single-Precision FP Computational Instructions Description | 63 |
| Table 35 | Single-Precision FP Conversion Instructions Description | 64 |
| Table 36 | Single-Precision FP to FP Sign-Injection Instructions Description | 65 |
| Table 37 | RISC-V Base Instruction to Assembly Pseudoinstruction Example | 65 |
| Table 38 | Single-Precision FP Move Instructions Description | 65 |
| Table 39 | Single-Precision FP Compare Instructions Description | 66 |
| Table 40 | Single-Precision FP Classify Instruction Description | 66 |
| Table 41 | Floating-Point Number Classes..... | 67 |
| Table 42 | Double-Precision FP Load and Store Instructions Description..... | 67 |
| Table 43 | Double-Precision FP Computational Instructions Description | 68 |
| Table 44 | Double-Precision FP Conversion Instructions Description | 70 |
| Table 45 | Double-Precision FP to FP Sign-Injection Instructions Description | 71 |
| Table 46 | RISC-V Base Instruction to Assembly Pseudoinstruction Example | 71 |
| Table 47 | Double-Precision FP Move Instructions Description | 72 |
| Table 48 | Double-Precision FP Compare Instructions Description..... | 72 |
| Table 49 | Double-Precision FP Classify Instruction Description | 73 |
| Table 50 | Stack-Pointed-Based Load Instruction Description..... | 74 |
| Table 51 | Stack-Pointed-Based Store Instruction Description | 75 |
| Table 52 | Register-Based Load Instruction Description | 75 |
| Table 53 | Register-Based Store Instruction Description | 76 |
| Table 54 | Unconditional Jump Instruction Description..... | 76 |
| Table 55 | Unconditional Control Transfer Instruction Description | 76 |
| Table 56 | Conditional Control Transfer Instruction Description..... | 77 |
| Table 57 | Integer Constant-Generation Instruction Description | 77 |
| Table 58 | Integer Register-Immediate Operation Description..... | 78 |
| Table 59 | Integer Register-Immediate Operation Description (cont.) | 78 |
| Table 60 | Integer Register-Immediate Operation Description (cont.) | 78 |
| Table 61 | Integer Register-Immediate Operation Description (cont.) | 78 |

| | | |
|-----------------|--|-----|
| Table 62 | Integer Register-Immediate Operation Description (cont.) | 79 |
| Table 63 | Integer Register-Register Operation Description | 79 |
| Table 64 | Integer Register-Register Operation Description (cont.)..... | 79 |
| Table 65 | Address Calculation Instructions Description | 80 |
| Table 66 | Add/Shift with Prefix Zero-Extend Instructions Description | 81 |
| Table 67 | Count Leading/Trailing Zeroes Instructions Description | 82 |
| Table 68 | Count Population Instructions Description | 82 |
| Table 69 | Logic-With-Negate Instructions Description | 83 |
| Table 70 | Comparison Instructions Description | 83 |
| Table 71 | Sign- and Zero-Extend Instructions | 84 |
| Table 72 | Bitwise Rotation Instructions Description | 85 |
| Table 73 | OR Combine Instruction Description..... | 85 |
| Table 74 | Byte-Reverse Instruction Description | 85 |
| Table 75 | Control and Status Register Instruction Description | 86 |
| Table 76 | CSR Reads and Writes | 87 |
| Table 77 | User Mode CSRs | 88 |
| Table 78 | Machine Mode CSRs..... | 89 |
| Table 79 | Debug Mode Registers | 90 |
| Table 80 | Core Generator Encoding of marchid..... | 91 |
| Table 81 | Generator Release Encoding of mimpid..... | 92 |
| Table 82 | Timer and Counter Pseudoinstruction Description..... | 93 |
| Table 83 | Timer and Counter CSRs | 94 |
| Table 84 | RISC-V Registers | 97 |
| Table 85 | RISC-V Assembly and C Examples..... | 98 |
| Table 86 | RISC-V Code Model Table | 111 |
| Table 87 | Branch Prediction Mode CSR | 117 |
| Table 88 | SiFive Feature Disable CSR..... | 118 |
| Table 89 | SiFive Feature Disable CSR Usage..... | 119 |
| Table 90 | Exception Priority | 121 |
| Table 91 | Summary of Exception and Interrupt CSRs | 122 |
| Table 92 | Machine Status Register (partial) | 125 |
| Table 93 | Machine Trap Vector Register..... | 126 |
| Table 94 | Encoding of mtvec.MODE | 126 |

| | | |
|------------------|---|-----|
| Table 95 | Machine Interrupt Enable Register | 127 |
| Table 96 | Machine Interrupt Pending Register | 127 |
| Table 97 | Machine Cause Register | 128 |
| Table 98 | mcause Exception Codes..... | 128 |
| Table 99 | RNMI CSRs | 130 |
| Table 101 | mnstatus CSR Fields | 130 |
| Table 102 | S76 Core Complex Interrupt IDs | 135 |
| Table 103 | CLINT Memory Map..... | 136 |
| Table 104 | PLIC Memory Map..... | 138 |
| Table 105 | Mapping of global_interrupts Signal Bits to PLIC Interrupt ID | 139 |
| Table 106 | PLIC Interrupt Priority Register | 139 |
| Table 107 | PLIC Interrupt Pending Register 1 | 140 |
| Table 108 | PLIC Interrupt Pending Register 4 | 140 |
| Table 109 | PLIC Interrupt Enable Register 1 for Hart 0 M-Mode | 141 |
| Table 110 | PLIC Interrupt Enable Register 4 for Hart 0 M-Mode | 141 |
| Table 111 | PLIC Clock Gate Disable Register..... | 141 |
| Table 112 | PLIC Interrupt Priority Threshold Register for Hart 0 M-Mode..... | 142 |
| Table 113 | PLIC Claim/Complete Register for Hart 0 M-Mode | 143 |
| Table 114 | Power Dial CSR | 147 |
| Table 115 | CorePower State Memory Map | 148 |
| Table 116 | CorePower State Register | 148 |
| Table 117 | DisableClockGate Memory Map..... | 149 |
| Table 118 | DisableClockGate Register..... | 150 |
| Table 119 | Debug Module Memory Map Seen from the Debug Module Interface | 155 |
| Table 120 | Debug Module Memory Map from the Perspective of the Core..... | 156 |
| Table 121 | Debug and Trigger Registers | 157 |
| Table 122 | Debug Control and Status Register | 158 |
| Table 123 | Trigger Select Register..... | 159 |
| Table 124 | Trigger Data Register 1 | 159 |
| Table 125 | Trigger Data Registers 2 and 3 | 160 |
| Table 126 | Trigger CSRs When Used as Breakpoints..... | 160 |
| Table 127 | Breakpoint Match Control Register | 161 |
| Table 128 | NAPOT Size Encoding..... | 162 |

| | | |
|------------------|--|-----|
| Table 129 | Debug Module Interface Signals | 166 |
| Table 130 | Debug Module Status Register | 167 |
| Table 131 | Debug Module Control Register | 169 |
| Table 132 | Hart Info Register | 170 |
| Table 133 | Abstract Control and Status Register | 171 |
| Table 134 | Abstract Command Register | 172 |
| Table 135 | Abstract Command Autoexec Register | 172 |
| Table 136 | Debug Module Control and Status 2 Register | 173 |
| Table 137 | Debug Abstract Commands | 174 |
| Table 138 | Abstract Command Example for 32-bit Block Write | 175 |
| Table 139 | System Bus vs. Program Buffer Comparison | 175 |
| Table 140 | Core Generator Encoding of marchid | 181 |
| Table 141 | Generator Release Encoding of mimpid | 182 |
| Table 142 | SiFive Custom CSRs | 183 |
| Table 143 | S7 Single-Precision FPU Instruction Latency and Repeat Rates | 185 |
| Table 144 | S7 Double-Precision FPU Instruction Latency and Repeat Rates | 186 |
| Table 145 | S76 Core Complex Manual Revision History | 187 |

Figures

| | | |
|------------------|------------------------------------|----|
| Figure 1 | S7 Series Block Diagram..... | 21 |
| Figure 2 | S7 Execution Pipeline | 29 |
| Figure 3 | RV64 pmpcfg0 Register | 34 |
| Figure 4 | RV64 pmpcfg2 Register | 34 |
| Figure 5 | RV64 pmpXcfg bitfield | 34 |
| Figure 6 | RV64 pmpaddrX Register..... | 36 |
| Figure 7 | PMP Example Block Diagram | 37 |
| Figure 8 | Event Selector Fields | 40 |
| Figure 9 | R-Type | 49 |
| Figure 10 | I-Type | 50 |
| Figure 11 | S-Type..... | 50 |
| Figure 12 | B-Type..... | 50 |
| Figure 13 | U-Type..... | 50 |
| Figure 14 | J-Type | 50 |
| Figure 15 | ADD Instruction Example..... | 51 |
| Figure 16 | ADDI Instruction Example..... | 53 |
| Figure 17 | LW Instruction Example | 54 |
| Figure 18 | Store Instructions..... | 54 |
| Figure 19 | SW Instruction Example | 55 |
| Figure 20 | JAL Instruction..... | 55 |
| Figure 21 | JALR Instruction | 55 |
| Figure 22 | Branch Instructions | 56 |
| Figure 23 | Upper-Immediate Instructions | 57 |
| Figure 24 | FENCE Instructions | 57 |
| Figure 25 | NOP Instructions | 58 |
| Figure 26 | Multiplication Operations | 58 |
| Figure 27 | Division Operations..... | 59 |
| Figure 28 | Atomic Operations | 59 |
| Figure 29 | Atomic Memory Operations..... | 60 |

| | | |
|------------------|--|----|
| Figure 30 | Floating-Point Control and Status Register | 61 |
| Figure 31 | Single-Precision FP Load Instruction | 62 |
| Figure 32 | Single-Precision FP Store Instruction | 62 |
| Figure 33 | Single-Precision FP Computational Instructions | 63 |
| Figure 34 | Single-Precision FP Fused Computational Instructions | 63 |
| Figure 35 | Single-Precision FP to Integer and Integer to FP Conversion Instructions | 63 |
| Figure 36 | Single-Precision FP to FP Sign-Injection Instructions | 64 |
| Figure 37 | Single-Precision FP Move Instructions | 65 |
| Figure 38 | Single-Precision FP Compare Instructions | 66 |
| Figure 39 | Single-Precision FP Classify Instruction | 66 |
| Figure 40 | Double-Precision FP Load Instruction | 67 |
| Figure 41 | Double-Precision FP Store Instruction | 67 |
| Figure 42 | Double-Precision FP Computational Instructions..... | 68 |
| Figure 43 | Double-Precision FP Fused Computational Instructions..... | 68 |
| Figure 44 | Double-Precision FP to Integer and Integer to FP Conversion Instructions | 69 |
| Figure 45 | Double-Precision to Single-Precision and Single-Precision to Double-Precision FP Conversion Instructions | 69 |
| Figure 46 | Double-Precision FP to FP Sign-Injection Instructions | 71 |
| Figure 47 | Double-Precision FP Move Instructions..... | 71 |
| Figure 48 | Double-Precision FP Compare Instructions | 72 |
| Figure 49 | Double-Precision FP Classify Instruction..... | 73 |
| Figure 50 | CR Format - Register | 73 |
| Figure 51 | CI Format - Immediate | 73 |
| Figure 52 | CSS Format - Stack-relative Store..... | 73 |
| Figure 53 | CIW Format - Wide Immediate | 73 |
| Figure 54 | CL Format - Load..... | 74 |
| Figure 55 | CS Format - Store..... | 74 |
| Figure 56 | CA Format - Arithmetic..... | 74 |
| Figure 57 | CJ Format - Jump | 74 |
| Figure 58 | Stack-Pointed-Based Loads..... | 74 |
| Figure 59 | Stack-Pointed-Based Stores | 74 |
| Figure 60 | Register-Based Loads..... | 75 |
| Figure 61 | Register-Based Stores | 75 |

| | | |
|------------------|--|----|
| Figure 62 | Unconditional Jump Instructions..... | 76 |
| Figure 63 | Unconditional Control Transfer Instructions | 76 |
| Figure 64 | Conditional Control Transfer Instructions..... | 77 |
| Figure 65 | Integer Constant-Generation Instructions | 77 |
| Figure 66 | Integer Register-Immediate Operations..... | 77 |
| Figure 67 | Integer Register-Immediate Operations (cont.) | 78 |
| Figure 68 | Integer Register-Immediate Operations (cont.) | 78 |
| Figure 69 | Integer Register-Immediate Operations (cont.) | 78 |
| Figure 70 | Integer Register-Immediate Operations (cont.) | 79 |
| Figure 71 | Integer Register-Register Operations..... | 79 |
| Figure 72 | Integer Register-Register Operations (cont.)..... | 79 |
| Figure 73 | Defined Illegal Instruction | 80 |
| Figure 74 | Address Calculation Instructions | 80 |
| Figure 75 | Address Calculation Instructions (cont.) | 80 |
| Figure 76 | Add with Prefix Zero-Extend Instruction | 81 |
| Figure 77 | Shift with Prefix Zero-Extend Instruction..... | 81 |
| Figure 78 | Count Leading/Trailing Zeroes Instructions..... | 81 |
| Figure 79 | Count Leading/Trailing Zeroes Instructions (cont.) | 81 |
| Figure 80 | Count Population Instruction | 82 |
| Figure 81 | Count Population Instruction (cont.)..... | 82 |
| Figure 82 | Logic-With-Negate Instructions | 82 |
| Figure 83 | Comparison Instructions..... | 83 |
| Figure 84 | Sign-Extend Instructions..... | 83 |
| Figure 85 | Zero-Extend Instruction | 83 |
| Figure 86 | Bitwise Rotation Instructions | 84 |
| Figure 87 | Bitwise Rotation Instructions (cont.)..... | 84 |
| Figure 88 | Bitwise Rotation Instructions (cont.)..... | 84 |
| Figure 89 | Bitwise Rotation Instructions (cont.)..... | 84 |
| Figure 90 | Zicsr Instructions | 86 |
| Figure 91 | Timer and Counter Pseudoinstructions | 93 |
| Figure 92 | ECALL and EBREAK Instructions..... | 95 |
| Figure 93 | Wait for Interrupt Instruction | 96 |
| Figure 94 | RISC-V Assembly Example | 98 |

| | | |
|-------------------|---|-----|
| Figure 95 | RISC-V Assembly to Machine Code | 99 |
| Figure 96 | One RISC-V Instruction | 100 |
| Figure 97 | Stack Memory during Function Calls..... | 102 |
| Figure 98 | S76 Core Complex Interrupt Architecture Block Diagram | 123 |
| Figure 99 | CLINT Block Diagram..... | 132 |
| Figure 100 | CLINT Interrupts and Vector Table | 133 |
| Figure 101 | CLINT Vector Table Example | 134 |
| Figure 102 | CLINT Interrupt Attribute Example | 135 |

Chapter 1

Introduction

SiFive's S76 Core Complex is a high performance implementation of the RISC-V *RV64GC_Zba_Zbb_Sscopmf* architecture. The SiFive S76 Core Complex is guaranteed to be compatible with all applicable RISC-V standards, and this document should be read together with the official RISC-V user-level, privileged, and external debug architecture specifications.



A summary of features in the S76 Core Complex can be found in Table 1.

| S76 Core Complex Feature Set | |
|---------------------------------|--|
| Feature | Description |
| Number of Harts | 1 Hart. |
| S7 Core | 1 × S7 RISC-V core. |
| PLIC Interrupts | 127 Interrupt signals, which can be connected to off-core-complex devices. |
| PLIC Priority Levels | The PLIC supports 7 priority levels. |
| Hardware Breakpoints | 4 hardware breakpoints. |
| Physical Memory Protection Unit | PMP with 8 regions and a minimum granularity of 64 bytes. |

Table 1: S76 Core Complex Feature Set

The S76 Core Complex also has a number of on-core-complex configurability options, allowing one to tune the design to a specific application. The configurable options are described in Appendix A.

1.1 About this Document

This document describes the functionality of the S76 Core Complex 21G3.02.00. To learn more about the production deliverables of the S76 Core Complex, consult the S76 Core Complex User Guide.

1.2 About this Release

This is a general release of the S76 Core Complex 21G3.02.00, with a supported life cycle of two years from the release date. Contact support@sifive.com if you have any questions.

1.3 S76 Core Complex Overview

The S76 Core Complex includes 1 × S7 64-bit RISC-V core, along with the necessary functional units required to support the core. These units include a Core-Local Interruptor (CLINT) to support local interrupts, a Platform-Level Interrupt Controller (PLIC) to support platform interrupts, physical memory protection, a Debug unit to support a JTAG-based debugger host connection, and a local crossbar that integrates the various components together.

The S76 Core Complex memory system consists of a Data Cache, Data Local Store (DLS), Instruction Cache, and Instruction Tightly-Integrated Memory (ITIM). The S76 Core Complex also includes a Front Port, which allows external masters to be coherent with the L1 memory system and access to the TIMs, thereby removing the need to maintain coherence in software for any external agents.

An overview of the SiFive S7 Series is shown in Figure 1. Refer to the `docs/core_complex_configuration.txt` file for a comprehensive summary of the S76 Core Complex configuration.

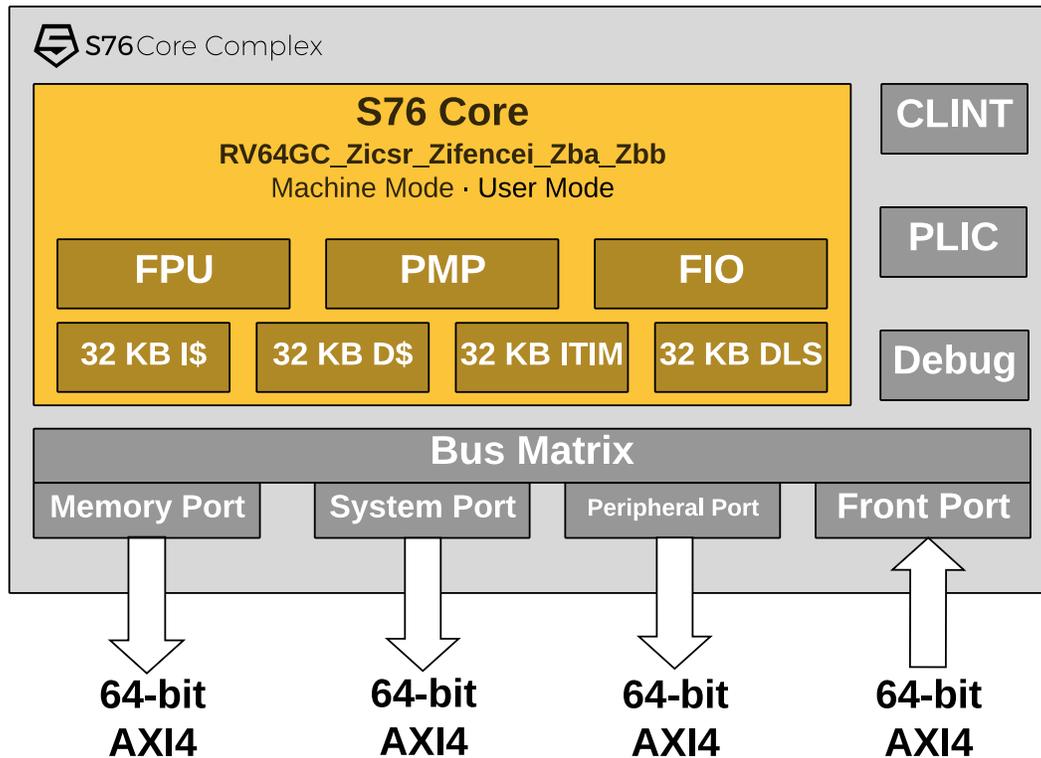


Figure 1: S7 Series Block Diagram

The S76 Core Complex memory map is detailed in Section 4.2, and the interfaces are described in full in the S76 Core Complex User Guide.

1.4 S7 RISC-V Core

The S76 Core Complex includes a 64-bit S7 RISC-V core, which has a dual-issue, in-order execution pipeline, with a peak execution rate of two instructions per clock cycle. The SiFive S7 core is guaranteed to be compatible with all applicable RISC-V standards.

The S7 core is configured to support the RV64I base ISA, as well as the Multiply (M), Atomic (A), Single-Precision Floating Point (F), Double-Precision Floating Point (D), Compressed (C), CSR Instructions (Zicsr), Instruction-Fetch Fence (Zifencei), Address Calculation (Zba), Basic Bit Manipulation (Zbb), and Count Overflow and Mode-Based Filtering (Sscofpmf) RISC-V extensions. This is captured by the RISC-V extension string: *RV64GC_Zba_Zbb_Sscofpmf*. The base ISA and instruction extensions are described in Chapter 5.

The S7 also supports machine and user privilege modes, in conjunction with Physical Memory Protection (PMP), thereby allowing System-on-Chip (SoC) implementations to make the right area, power, and feature trade-offs.

The S7 core is designed to be feature rich, providing a very flexible memory system that includes L1 caches, Tightly-Integrated Memory (TIM), standards-based configurable bus interfaces, and memory maps that provide a lot of flexibility for SoC integration.

The microarchitecture also incorporates a branch prediction unit that is composed of a 4-entry Branch Target Buffer (BTB), a 1.3 KiB-entry Branch History Table (BHT), a 3-entry Return Address Stack (RAS), 2-entry Indirect Jump Target Predictor (IJTP), and a 16-entry Return Instruction Predictor.

The S7 includes an IEEE 754-2008 compliant Floating-Point Unit.

The S7 core is described in more detail in Chapter 3.

1.5 Memory System

The S76 Core Complex memory system has a Level 1 memory system optimized for high performance. The instruction subsystem consists of a 32 KiB, 2-way instruction cache. The S76 Core Complex also includes a 32 KiB Instruction Tightly Integrated Memory (ITIM). The data subsystem is comprised of a high performance 32 KiB, 4-way L1 data cache. The memory system is described in more detail in Chapter 3.

1.6 Interrupts

The S76 Core Complex provides the standard RISC-V M-mode timer and software interrupts via the Core-Local Interruptor (CLINT).

The S76 Core Complex also includes a RISC-V standard Platform-Level Interrupt Controller (PLIC), which supports 127 global interrupts with 7 priority levels.

Interrupts are described in Chapter 7. The CLINT is described in Chapter 8. The PLIC is described in Chapter 9.

1.7 Debug Support

The S76 Core Complex provides external debugger support over an industry-standard JTAG port, including 4 hardware-programmable breakpoints per hart.

Debug support is described in detail in Chapter 12, and the debug interface is described in the S76 Core Complex User Guide.

1.8 Compliance

The S76 Core Complex is compliant to the following versions of the various RISC-V specifications:

| ISA | Version | Status |
|---|----------------|---------------|
| RV64I Base Integer Instruction Set | 2.1 | Ratified |
| Extensions | Version | Status |
| M Standard Extension for Integer Multiplication and Division | 2.0 | Ratified |
| A Standard Extension for Atomic Instruction | 2.1 | Ratified |
| F Standard Extension for Single-Precision Floating-Point | 2.2 | Ratified |
| D Standard Extension for Double-Precision Floating-Point | 2.2 | Ratified |
| C Standard Extension for Compressed Instruction | 2.0 | Ratified |
| Zicsr Standard Extension for Control and Status Register (CSR) Instructions | 2.0 | Ratified |
| Zifencei Standard Extension for Instruction-Fetch Fence | 2.0 | Ratified |
| Zba Standard Extension for Address Calculation | 1.0 | Ratified |
| Zbb Standard Extension for Basic Bit Manipulation | 1.0 | Ratified |
| Sscofpmf Standard Extension for Count Overflow and Mode-Based Filtering | 0.1 | Ratified |
| Privilege Mode | Version | Status |
| Machine-Level ISA | 1.11 | Ratified |
| User-Level ISA | 1.11 | Ratified |
| Devices | Version | Status |
| The RISC-V Debug Specification | 1.0 | Frozen |
| RISC-V Platform-Level Interrupt Controller (PLIC) Specification | — | — |

Table 2: RISC-V Specification Compliance

Chapter 2

List of Abbreviations and Terms

| Term | Definition |
|-----------------|--|
| BHT | Branch History Table |
| BTB | Branch Target Buffer |
| CLIC | Core-Local Interrupt Controller. Configures priorities and levels for core-local interrupts. |
| CLINT | Core-Local Interruptor. Generates per hart software and timer interrupts. |
| DTIM | Data Tightly-Integrated Memory |
| Hart | HARdware Thread |
| IJTP | Indirect-Jump Target Predictor |
| ITIM | Instruction Tightly-Integrated Memory |
| JTAG | Joint Test Action Group |
| LIM | Loosely-Integrated Memory. Used to describe memory space delivered in a SiFive Core Complex that is not tightly integrated to a CPU core. |
| PLIC | Platform-Level Interrupt Controller. The global interrupt controller in a RISC-V system. |
| PMC | Power Management Controller |
| PMP | Physical Memory Protection |
| RAS | Return-Address Stack |
| RO | Used to describe a Read-Only register field |
| RS | Read/Set field. A register field that cannot be cleared by software, only reset will clear. |
| RW | Used to describe a Read/Write register field |
| RW1C | Used to describe a Read/Write-1-to-Clear register field |
| TileLink | A free and open interconnect standard originally developed at UC Berkeley |
| W1C | Used to describe a Write-1-to-Clear register field |
| WARL | Write-Any, Read-Legal field. A register field that can be written with any value, but returns only supported values when read. |
| WIRI | Writes-Ignored, Reads-Ignore field. A read-only register field reserved for future use. Writes to the field are ignored and reads should ignore the value returned. |
| WLRL | Write-Legal, Read-Legal field. A register field that should only be written with legal values and that only returns legal value if last written with a legal value. |
| WO | Used to describe a Write-Only register field |
| WPRI | Writes-Preserve, Reads-Ignore field. A register field that might contain unknown information. Reads should ignore the value returned, but writes to the whole register should preserve the original value. |

Table 3: Abbreviations and Terms

Chapter 3

S7 RISC-V Core

This chapter describes the 64-bit S7 RISC-V processor core, instruction fetch and execution unit, L1 memory system, Physical Memory Protection unit, Hardware Performance Monitor, and external interfaces.

The S7 feature set is summarized in Table 4.

| Feature | Description |
|--|--|
| ISA | <i>RV64GC_Zba_Zbb_Sscofpmf</i> |
| SiFive Custom Instruction Extension (SCIE) | Not Present |
| Privilege Modes | M, U |
| L1 Instruction Cache | 32 KiB 2-way instruction cache |
| Instruction Tightly-Integrated Memory (ITIM) | 32 KiB ITIM |
| L1 Data Cache | 32 KiB 4-way data cache |
| Data Local Store (DLS) | 32 KiB DLS with 1 bank |
| Fast I/O | Present |
| Physical Memory Protection | 8 regions with a granularity of 64 bytes |

Table 4: S7 Feature Set

3.1 Supported Privilege Modes

The S7 supports the RISC-V user mode, providing two levels of privilege: machine (M) and user (U). U-mode provides a mechanism to isolate application processes from each other and from trusted code running in M-mode.

See *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* for more information on the privilege modes.

3.2 Instruction Memory System

This section describes the instruction memory system of the S7 core.

3.2.1 Execution Memory Space

The regions of executable memory consist of all directly addressable memory in the system. The memory includes any volatile or non-volatile memory located off the Core Complex ports, and includes the on-core-complex ITIM.

Table 5 shows the executable regions of the S76 Core Complex.

| Base | Top | Description |
|-------------|-------------|---------------------------|
| 0x0180_0000 | 0x0180_7FFF | ITIM |
| 0x2000_0000 | 0x3FFF_FFFF | Peripheral Port (512 MiB) |
| 0x4000_0000 | 0x5FFF_FFFF | System Port (512 MiB) |
| 0x7000_0000 | 0x7000_7FFF | Data Local Store |
| 0x8000_0000 | 0x9FFF_FFFF | Memory Port (512 MiB) |

Table 5: Executable Memory Regions for the S76 Core Complex

All executable regions, except the ITIM, are treated as instruction cacheable. There is no method to disable this behavior.

Trying to execute an instruction from a non-executable address results in an instruction access trap.

3.2.2 L1 Instruction Cache

The L1 instruction cache is a 32 KiB 2-way set-associative cache. It has a line size of 64 bytes and is read/write-allocate with a random replacement policy. A cache line fill triggers a burst access outside of the Core Complex, starting with the first address of the cache line. There are no write-backs to memory from the instruction cache and it is not kept coherent with rest of the platform memory system.

Out of reset, all blocks of the instruction cache are invalidated. The access latency of the cache is one clock cycle. There is no way to disable the instruction cache and cache allocations begin immediately out of reset.

3.2.3 Cache Maintenance

The instruction cache supports the `FENCE.I` instruction, which invalidates the entire instruction cache, as described in Section 5.14. Writes to instruction memory from the core or another master must be synchronized with the instruction fetch stream by executing `FENCE.I`.

3.2.4 Instruction Tightly-Integrated Memory (ITIM)

The S7 includes a 32 KiB ITIM in addition to the L1 instruction cache. ITIM accesses have the same performance as instruction cache hits, but can never suffer a miss. This makes the ITIM useful for storing code that benefits from deterministic execution, such as interrupt handlers.

3.2.5 Instruction Fetch Unit

The S7 instruction fetch unit is responsible for keeping the pipeline fed with instructions from memory. The instruction fetch unit delivers up to 8 bytes of instructions per clock cycle to support superscalar instruction execution. Fetches are always word-aligned and there is a one-cycle penalty for branching to a 32-bit instruction that is not word-aligned.

The S7 implements the standard Compressed (C) extension to the RISC-V architecture, which allows for 16-bit RISC-V instructions. As four 16-bit instructions can be fetched per cycle, the instruction fetch unit can be idle when executing programs comprised mostly of compressed 16-bit instructions. This reduces memory accesses and power consumption.

All branches must be aligned to half-word addresses. Otherwise, the fetch generates an instruction address misaligned trap. Trying to fetch from a non-executable or unimplemented address results in an instruction access trap.

3.2.6 Branch Prediction

The S7 instruction fetch unit contains sophisticated predictive hardware to mitigate the performance impact of control hazards within the instruction stream. The instruction fetch unit is decoupled from the execution unit, so that correctly predicted control-flow events usually do not result in execution stalls.

- A 4-entry branch target buffer (BTB), which predicts the target of taken branches and direct jumps;
- A 1.3 KiB branch history table (BHT), which predicts the direction of conditional branches;
- A 2-entry indirect-jump target predictor (IJTP);
- A 3-entry return-address stack (RAS), which predicts the target of procedure returns.

The BHT is a correlating predictor that supports long branch histories. The BTB has one-cycle latency, so that correctly predicted branches and direct jumps result in no penalty, provided the target is 8-byte aligned.

Direct jumps that miss in the BTB result in a one-cycle fetch bubble. This event might not result in any execution stalls if the fetch queue is sufficiently full.

The BHT, IJTP, and RAS take precedence over the BTB. If these structures' predictions disagree with the BTB's prediction, a one-cycle fetch bubble results. Similar to direct jumps that miss in the BTB, the fetch bubble might not result in an execution stall.

Mispredicted branches usually incur a four-cycle penalty, but sometimes the branch resolves later in the execution pipeline and incurs a six-cycle penalty instead. Mispredicted indirect jumps incur a six-cycle penalty.

Branch prediction is enabled out of reset and cannot be disabled. However, instruction speculation, fetching before a prediction is confirmed, must be enabled in the Feature Disable CSR, described in Chapter 6.

As instruction speculation can occur at any point after it has been enabled, data cacheable regions of memory (i.e., DDR) must be able to respond to instruction fetches immediately after instruction speculation is enabled. If DDR initialization is not completed before instruction speculation is enabled, the memory system must return a decode error (DECERR) for accesses made to DDR. The fetch unit will ignore errors associated with speculative accesses and continue to operate normally.

The Branch Prediction Mode CSR, also described in Chapter 6, provides a means to customize the branch predictor behavior to trade average performance for more predictable execution time.

3.3 Execution Pipeline

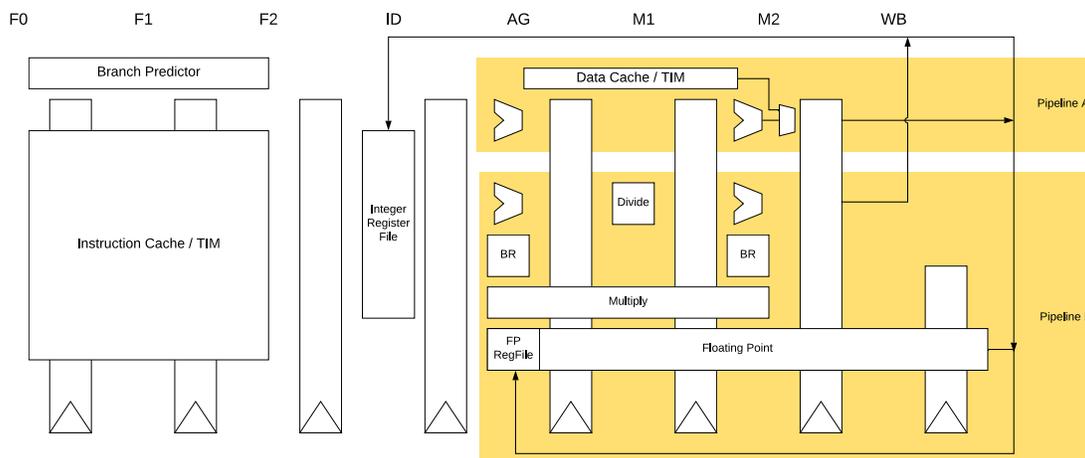


Figure 2: S7 Execution Pipeline

The S7 execution unit is a dual-issue, in-order pipeline. The pipeline comprises eight stages: three stages of instruction fetch (F0-F2), instruction decode (ID), address generation (AG), two stages of data memory access (M1 and M2), and register write-back (WB). The pipeline has a peak execution rate of two instructions per clock cycle, and is fully bypassed so that most instructions have a one-cycle result latency:

- Integer arithmetic and branch instructions can execute in either the AG or M2 pipeline stage. If such an instruction's operands are available when the instruction enters the AG stage, then it executes in AG; otherwise, it executes in M2.
- Loads produce their result in the M2 stage. There is no load-use delay for most integer instructions. However, effective addresses for memory accesses are always computed in the AG stage. Hence, loads, stores, and indirect jumps require their address operands to be ready when the instruction enters AG. If an address-generation operation depends upon a load from memory, then the load-use delay is two cycles.
- Integer multiplication instructions consume their operands in the AG stage and produce their results in the M2 stage. The integer multiplier is fully pipelined.

- Integer division instructions consume their operands in the AG stage. These instructions have between a six-cycle and 68-cycle result latency, depending on the operand values.
- CSR accesses execute in the M2 stage. CSR read data can be bypassed to most integer instructions with no delay. Most CSR writes flush the pipeline, which is a seven-cycle penalty.

| Instruction | Latency |
|---|---|
| LW | Three-cycle latency, assuming cache hit ¹ |
| LH, LHU, LB, LBU | Three-cycle latency, assuming cache hit ¹ |
| CSR Reads | One-cycle latency ² |
| MUL, MULH, MULHU, MULHSU | Three-cycle latency |
| DIV, DIVU, REM, REMU | Between six-cycle to 68-cycle latency, depending on operand values ³ |
| ¹ Effective address not ready in AG stage. Load-to-use latency = load-to-use delay + 1 ² cycle latency = cycle delay + 1 ³ The latency of DIV, DIVU, REM, and REMU instructions can be determined by calculating: Latency = 2 cycles + $\log_2(\text{dividend}) - \log_2(\text{divisor}) + 1$ cycle if the input is negative + 1 cycle if the output is negative | |

Table 6: S7 Instruction Latency

The pipeline has some register dependencies, where it interlocks on read-after-write and write-after-write hazards, so instructions may be scheduled to avoid stalls. Otherwise, the processor can have multiple outstanding memory-mapped I/O accesses, even to the same address. The pipeline implements a flexible dual-instruction-issue scheme. Provided there are no data hazards between a pair of instructions, the two instructions may issue in the same cycle, provided the following constraints are met:

- At most one instruction accesses data memory.
- At most one instruction is a branch or jump.
- At most one instruction is a floating-point arithmetic operation.
- At most one instruction is an integer multiplication or division operation.
- Neither instruction explicitly accesses a CSR.

See Appendix D for a complete list of floating-point unit instruction timings.

3.4 Data Memory System

The data memory system consists of on-core-complex data and the ports in the S76 Core Complex memory map, shown in Section 4.2. The on-core-complex data memory consists of a 32 KiB L1 data cache. A design cannot have both data cache and DTIM.

Data accesses are classified as cacheable, for those targeting the Memory Port; or non-cacheable, for those targeting any other port in the Core Complex. Non-cacheable data accesses are collectively called memory-mapped I/O accesses, or MMIOs.

The S7 pipeline allows for multiple outstanding memory accesses, but only allows one outstanding cache line fill. The memory system includes the Fast I/O feature, described in Section 3.5, which improves the throughput of MMIOs. The number of outstanding MMIOs are implementation dependent. Misaligned accesses are not allowed to any memory region and result in a trap to allow for software emulation.

3.4.1 L1 Data Cache

The L1 data cache is a 32 KiB 4-way set-associative cache. It has a line size of 64 bytes and is read/write-allocate with a PLRU replacement policy. The cache operates in write-back mode; this means that if a cache line is dirty, it is written back to memory when evicted. Out of reset, all lines of the cache are invalidated.

The Memory Port address range is the only cacheable region of memory. A cache line fill triggers a burst access starting with the first address of the cache line. On a cache hit, the access latency is two clock cycles for words and double-words, and three clock cycles for smaller quantities. Stores are pipelined and commit on cycles where the data memory system is otherwise idle. Pending stores are stored in a buffer, which drains whenever there is an idle cycle or another store. Loads to addresses currently in the store pipeline result in a five-cycle penalty.

The data cache supports only one outstanding line fill. Once a cacheable access is made that misses, another cannot be issued until the line fill completes. However, other MMIOs can be issued before or after the line fill as long as there are no address or register hazards.

The data cache cannot be disabled and the properties of the Memory Port cannot be modified to prevent cacheable accesses.

3.4.2 Cache Maintenance Operations

The data cache supports `CFLUSH.D.L1` and `CDISCARD.D.L1`. The instruction `CFLUSH.D.L1` cleans and invalidates the specified line or all cache lines. The instruction `CDISCARD.D.L1` invalidates the specified line or all cache lines.

These custom instructions are further described in Chapter 6.

3.4.3 Data Local Store (DLS)

The S7 includes an additional fast, local memory called the Data Local Store (DLS). The DLS is 32 KiB in size, has 1 bank, and is directly addressable, as shown in Section 4.2. Accesses to the DLS have a fixed, five-cycle load-to-use latency, which makes it ideal for holding data that requires deterministic access time.

3.5 Fast I/O

The Fast I/O feature improves the performance of the memory-mapped I/O (MMIO) subsystem. This is achieved by predicting whether an access is I/O or not by examining the base address of a read or write.

Fast I/O enables a sustained rate of one MMIO operation per clock cycle. By contrast, when this feature is excluded, MMIO loads can only sustain half that rate. Fast I/O also decouples the MMIO load response from the cache-hit path. This way, MMIO requests and responses can happen on the same cycle, doubling the peak load throughput.

Note

Fast I/O is NOT an I/O port.

3.6 Atomic Memory Operations

The S7 core supports the RISC-V standard Atomic (A) extension on the Memory Port, Peripheral Port, and internal memory regions.

Atomic instructions that target the Memory Port are implemented in the data cache and are not observable on the external data bus. The load-reserved (LR) and store-conditional (SC) instructions are special atomic instructions that are only supported in data cacheable regions. They will generate a precise access exception if targeted at uncacheable data regions.

Atomic memory operations are not supported on the System Port. Atomic operations that target the System Port will generate a precise access exception.

See Section 5.4 for more information on the instructions added by this extension.

3.7 Floating-Point Unit (FPU)

The S7 FPU provides full hardware support for the IEEE 754-2008 floating-point standard for 32-bit single-precision and 64-bit double-precision arithmetic. The FPU includes a fully pipelined fused-multiply-add unit and an iterative divide and square-root unit, magnitude comparators, and float-to-integer conversion units, all with full hardware support for subnormals and all IEEE default values.

Section 5.5 describes the 32-bit single-precision instructions. Section 5.6 describes the 64-bit double-precision instructions.

The FPU comes up disabled on reset. First initialize `fcsr` and `mstatus.FS` prior to executing any floating-point instructions. In the `freedom-meta1` startup code, write `mstatus.FS[1:0]` to `0x1`.

3.8 Physical Memory Protection (PMP)

Machine mode is the highest privilege level and by default has read, write, and execute permissions across the entire memory map of the device. However, privilege levels below machine mode do not have read, write, or execute permissions to any region of the device memory map unless it is specifically allowed by the PMP. For the lower privilege levels, the PMP may grant permissions to specific regions of the device's memory map, but it can also revoke permissions when in machine mode.

When programmed accordingly, the PMP will check every access when the hart is operating in user mode. For machine mode, PMP checks do not occur unless the lock bit (L) is set in the `pmpcfgY` CSR for a particular region.

PMP checks also occur on loads and stores when the machine previous privilege level is user (`mstatus.MPP=0x0`), and the Modify Privilege bit is set (`mstatus.MPRV=1`).

The S7 PMP supports 8 regions with a minimum region size of 64 bytes.

This section describes how PMP concepts in the RISC-V architecture apply to the S7. For additional information on the PMP refer to *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

3.8.1 PMP Functional Description

The S7 PMP unit has 8 regions and a minimum granularity of 64 bytes. Access to each region is controlled by an 8-bit `pmpXcfg` field and a corresponding `pmpaddrX` register. Overlapping regions are permitted, where the lower numbered `pmpXcfg` and `pmpaddrX` registers take priority over higher numbered regions. The S7 PMP unit implements the architecturally defined `pmpcfgY` CSR `pmpcfg0`, supporting 8 regions. `pmpcfg2` is implemented, but hardwired to zero. Access to `pmpcfg1` or `pmpcfg3` results in an illegal instruction exception.

The PMP registers may only be programmed in M-mode. Ordinarily, the PMP unit enforces permissions on U-mode accesses. However, locked regions (see Section 3.8.2) additionally enforce their permissions on M-mode.

3.8.2 PMP Region Locking

The PMP allows for region locking whereby, once a region is locked, further writes to the configuration and address registers are ignored. Locked PMP entries may only be unlocked with a system reset. A region may be locked by setting the L bit in the pmpXcfg register.

In addition to locking the PMP entry, the L bit indicates whether the R/W/X permissions are enforced on machine mode accesses. When the L bit is clear, the R/W/X permissions apply only to U-mode.

3.8.3 PMP Registers

Each PMP region is described by an 8-bit pmpXcfg field, used in association with a 64-bit pmpaddrX register that holds the base address of the protected region. The range of each region depends on the Addressing (A) mode described in the next section. The pmpXcfg fields reside within 64-bit pmpcfgY CSRs.

Each 8-bit pmpXcfg field includes a read, write, and execute bit, plus a two bit address-matching field A, and a Lock bit, L. Overlapping regions are permitted, where the lowest numbered PMP entry wins for that region.

PMP Configuration Registers

For RV64 architectures, pmpcfg1 and pmpcfg3 are not implemented. This reduces the footprint since pmpcfg2 already contains configuration fields pmp8cfg through pmp11cfg for both RV32 and RV64.

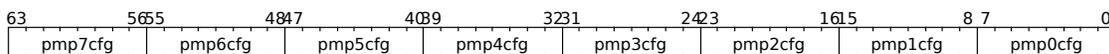


Figure 3: RV64 pmpcfg0 Register

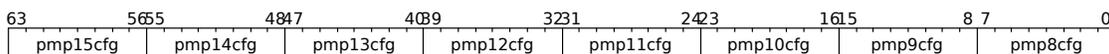


Figure 4: RV64 pmpcfg2 Register

The pmpcfgY and pmpaddrX registers are only accessible via CSR specific instructions such as csrr for reads, and csrw for writes.

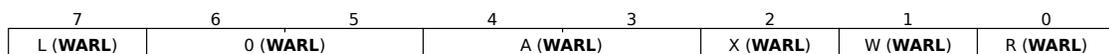


Figure 5: RV64 pmpXcfg bitfield

| Bits | Description |
|---|---|
| 0 | R: Read Permissions <ul style="list-style-type: none"> • 0x0 - No read permissions for this region • 0x1 - Read permission granted for this region |
| 1 | W: Write Permissions <ul style="list-style-type: none"> • 0x0 - No write permissions for this region • 0x1 - Write permission granted for this region |
| 2 | X: Execute permissions <ul style="list-style-type: none"> • 0x0 - No execute permissions for this region • 0x1 - Execute permission granted for this region |
| [4:3] | A: Address matching mode <ul style="list-style-type: none"> • 0x0 - PMP Entry disabled. No PMP protection applied for any privilege level. • 0x1 - Top of range (TOR) region defined by two adjacent pmpaddr registers. The upper limit of region X is defined by pmpaddrX, and the base of the region is defined by pmpaddr(X-1). Address 'a' matches the region if $[pmpaddr(X-1) \leq a < pmpaddrX]$. If pmp0cfg defines a TOR region, then the base address of that region is 0x0, and pmpaddr0 defines the upper limit. Supports only a four byte granularity. • 0x2 - Naturally aligned four-byte region (NA4). Supports only a four-byte region with four byte granularity. • 0x3 - Naturally aligned power-of-two region (NAPOT), ≥ 8 bytes. When this setting is programmed, the low bits of the pmpaddrX register encode the size, while the upper bits encode the base address right shifted by two. There is a zero bit in between, we will refer to as the least significant zero bit (LSZB). |
| 7 | L: Lock Bit <ul style="list-style-type: none"> • 0x0 - PMP Entry Unlocked, no permission restrictions applied to machine mode. PMP entry only applies to S and U modes. • 0x1 - PMP Entry Locked, permissions enforced for all privilege levels including machine mode. Writes to pmpXcfg and pmpcfgY are ignored and can only be cleared with system reset. |
| Note: The combination of R=0 and W=1 is not currently implemented. | |

Table 7: pmpXcfg Bitfield Description

Out of reset, the PMP register fields A and L are set to 0. All other hart state is unspecified by *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

Some examples follow using NAPOT address mode.

| Base Address | Region Size* | LSZB Position | pmpaddrX Value |
|----------------------------------|--------------|---------------|--|
| 0x4000_0000 | 8 B | 0 | (0x1000_0000 1'b0) |
| 0x4000_0000 | 32 B | 2 | (0x1000_0000 3'b011) |
| 0x4000_0000 | 4 KB | 9 | (0x1000_0000 10'b01_1111_1111) |
| 0x4000_0000 | 64 KB | 13 | (0x1000_0000 14'b01_1111_1111_1111) |
| 0x4000_0000 | 1 MB | 17 | (0x1000_0000 18'b01_1111_1111_1111_1111) |
| *Region size is $2^{(LSZB+3)}$. | | | |

Table 8: pmpaddrX Encoding Examples for A=NAPOT

PMP Address Registers

The PMP has 8 address registers. Each address register pmpaddrX correlates to the respective pmpXcfg field. Each address register contains the base address of the protected region right shifted by two, for a minimum 4-byte alignment.

The maximum encoded address bits per *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* are [55:2].

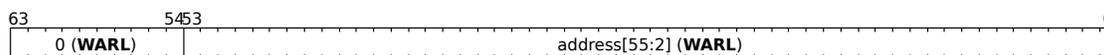


Figure 6: RV64 pmpaddrX Register

3.8.4 PMP and PMA

The PMP values are used in conjunction with the Physical Memory Attributes (PMAs) described in Section 4.1. Since the PMAs are static and not configurable, the PMP can only revoke read, write, or execute permissions to the PMA regions if those permissions already apply statically.

3.8.5 PMP Programming Overview

The PMP registers can only be programmed in machine mode. The pmpaddrX register should be first programmed with the base address of the protected region, right shifted by two. Then, the pmpcfgY register should be programmed with the properly configured 64-bit value containing each properly aligned 8-bit pmpXcfg field. Fields that are not used can be simply written to 0, marking them unused.

PMP Programming Example

The following example shows a machine mode only configuration where PMP permissions are applied to three regions of interest, and a fourth region covers the remaining memory map. Recall that lower numbered pmpXcfg and pmpaddrX registers take priority over higher numbered regions. This rule allows higher numbered PMP registers to have blanket coverage over the entire memory map while allowing lower numbered regions to apply permissions to specific regions of interest. The following example shows a 64 KB Flash region at base address 0x0, a

32 KB RAM region at base address 0x2000_0000, and finally a 4 KB peripheral region at base address base 0x3000_0000. The rest of the memory map is reserved space.

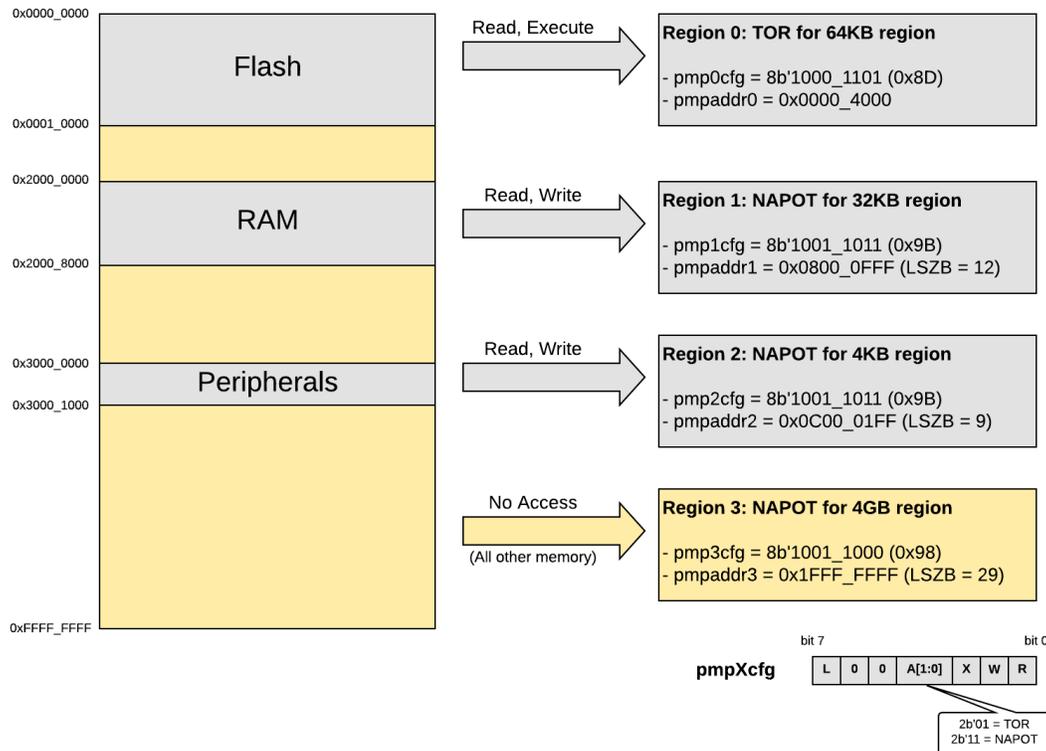


Figure 7: PMP Example Block Diagram

PMP Access Scenarios

The L, R, W, and X bits only determine if an access succeeds if all bytes of that access are covered by that PMP entry. For example, if a PMP entry is configured to match the four-byte range 0xC–0xF, then an 8-byte access to the range 0x8–0xF will fail, assuming that PMP entry is the highest-priority entry that matches those addresses.

While operating in machine mode when the lock bit is clear (L=0), if a PMP entry matches all bytes of an access, the access succeeds. If the lock bit is set (L=1) while in machine mode, then the access depends on the permissions set for that region. Similarly, while in Supervisor mode, the access depends on permissions set for that region.

Failed read or write accesses generate a load or store access exception, and an instruction access fault would occur on a failed instruction fetch. When an exception occurs while attempting to execute from a region without execute permissions, the fault occurs on the fetch and not the branch, so the mepc CSR will reflect the value of the targeted protected region, and not the address of the branch.

It is possible for a single instruction to generate multiple accesses, which may not be mutually atomic. If at least one access generated by an instruction fails, then an exception will occur. It might be possible that other accesses from a single instruction will succeed, with visible side effects. For example, references to virtual memory may be decomposed into multiple accesses.

On some implementations, misaligned loads, stores, and instruction fetches may also be decomposed into multiple accesses, some of which may succeed before an access exception occurs. In particular, a portion of a misaligned store that passes the PMP check may become visible, even if another portion fails the PMP check. The same behavior may manifest for floating-point stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

3.8.6 PMP and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*. When paging is enabled, instructions that access virtual memory may result in multiple physical-memory accesses, including implicit references to the page tables. The PMP checks apply to all of these accesses. The effective privilege mode for implicit page-table accesses is supervisor mode.

Implementations with virtual memory are permitted to perform address translations speculatively and earlier than required by an explicit virtual-memory access. The PMP settings for the resulting physical address may be checked at any point between the address translation and the explicit virtual-memory access. A mis-predicted branch to a non-executable address range does not generate a trap. Hence, when the PMP settings are modified in a manner that affects either the physical memory that holds the page tables or the physical memory to which the page tables point, M-mode software must synchronize the PMP settings with the virtual memory system. This is accomplished by executing an SFENCE.VMA instruction with $rs1=x0$ and $rs2=x0$, after the PMP CSRs are written.

If page-based virtual memory is not implemented, or when it is disabled, memory accesses check the PMP settings synchronously, so no fence is needed.

3.8.7 PMP Limitations

In a system containing multiple harts, each hart has its own PMP device. The PMP permissions on a hart cannot be applied to accesses from other harts in a multi-hart system. In addition, SiFive designs may contain a Front Port to allow external bus masters access to the full memory map of the system. The PMP cannot prevent access from external bus masters on the Front Port.

3.8.8 Behavior for Regions without PMP Protection

If a non-reserved region of the memory map does not have PMP permissions applied, then by default, supervisor or user mode accesses will fail, while machine mode access will be allowed.

Access to reserved regions within a device's memory map (an interrupt controller for example) will return 0x0 on reads, and writes will be ignored. Access to reserved regions outside of a device's memory map without PMP protection will result in a bus error.

3.8.9 Cache Flush Behavior on PMP Protected Region

When a line is brought into cache and the PMP is set up with the lock (L) bit asserted to protect a part of that line, a data cache flush instruction will generate a store access fault exception if the flush includes any part of the line that is protected. The cache flush instruction does an invalidate and write-back, so it is essentially trying to write back to the memory location that is protected. If a cache flush occurs on a part of the line that was not protected, the flush will succeed and not generate an exception. If a data cache flush is required without a write-back, use the cache discard instruction instead, as this will invalidate but not write back the line.

3.9 Hardware Performance Monitor

The S7 processor core supports a basic hardware performance monitoring (HPM) facility. The performance monitoring facility is divided into two classes of counters: fixed-function and event-programmable counters. These classes consist of a set of fixed counters and their counter-enable registers, as well as a set of event-programmable counters and their event selector registers. The registers are available to control the behavior of the counters. Performance monitoring can be useful for multiple purposes, from optimization to debug.

3.9.1 Performance Monitoring Counters Reset Behavior

The `instret` and `cycle` counters are initialized to zero on system reset. The hardware performance monitor event counters are not initialized on system reset, and thus have an arbitrary value. Users can write desired values to the counter control and status registers (CSRs) to start counting at a given, known value.

3.9.2 Fixed-Function Performance Monitoring Counters

A fixed-function performance monitor counter is hardware wired to only count one specific event type. That is, they cannot be reconfigured with respect to the event type(s) they count. The only modification to the fixed-function performance monitoring counters that can be done is to enable or disable counting, and write the counter value itself.

The S7 processor core contains two fixed-function performance monitoring counters.

Fixed-Function Cycle Counter (`mcycle`)

The fixed-function performance monitoring counter `mcycle` holds a count of the number of clock cycles the hart has executed since some arbitrary time in the past. The `mcycle` counter is read-write and 64 bits wide. Reads of `mcycle` return all 64 bits of the `mcycle` CSR.

Fixed-Function Instructions-Retired Counter (`minstret`)

The fixed-function performance monitoring counter `minstret` holds a count of the number of instructions the hart has retired since some arbitrary time in the past. The `minstret` counter is read-write and 64 bits wide. Reads of `minstret` return all 64 bits of the `minstret` CSR.

3.9.3 Event-Programmable Performance Monitoring Counters

Complementing the fixed-function counters are a set of programmable event counters. The S7 HPM includes two additional event counters, `mhpmcounter3` and `mhpmcounter4`. These programmable event counters are read-write and 64 bits wide. The hardware counters themselves are implemented as 40-bit counters on the S7 core series. These hardware counters can be written to in order to initialize the counter value.

3.9.4 Event Selector Registers

To control the event type to count, event selector CSRs `mhpmevent3` and `mhpmevent4` are used to program the corresponding event counters. These event selector CSRs are 64-bit **WARL** registers.

The event selectors are partitioned into three fields; the lower 8 bits select an event class, a middle set of bits that form a mask of events in that class, with the upper 8-bits used for counter overflow and event filtering.

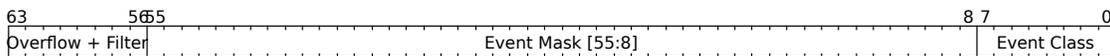


Figure 8: Event Selector Fields

The counter increments if the event corresponding to any set mask bit occurs. For example, if `mhpmevent3` is set to `0x4200`, then `mhpmcounter3` will increment when either a load instruction or a conditional branch instruction retires. An event selector of 0 means "count nothing".

Counter Overflow and Event Filters

The upper 8-bits of the `mhpmevent` register are used for controlling a counter overflow interrupt, as well as mode-based event filtering. The bit layout is shown below:

| Machine Hardware Performance Monitor Event Register | | | |
|---|----------|------|--|
| Bits | Name | Attr | Description |
| [7:0] | Class | WARL | Selects the Event Class to make available for counting |
| [55:8] | EventSel | WARL | Bit-mask of Event(s) to count |
| [57:56] | Reserved | — | |
| 58 | VUINH | WARL | Reserved |
| 59 | VSINH | WARL | Reserved |
| 60 | UINH | WARL | If set, counting of events in U-mode is inhibited |
| 61 | SINH | WARL | Reserved |
| 62 | MINH | RW | If set, then counting of events in M-mode is inhibited |
| 63 | OF | RW | Overflow status and interrupt disable bit. Set by hardware when counter overflows. |

Table 9: mhpmeventX Register Bit Layout for Overflow and Filtering

Performance Counter Overflow Operation

Each of the five xINH bits inhibits counting of event when the core is in privilege mode x. All zeroes in these fields results in counting of events in all modes.

The OF bit is set by hardware when the corresponding hpmcounterX overflows and remains set until written by software. Since hpmcounter values are unsigned, overflow is defined as incrementing from all-ones to all-zeroes. Note that there is no loss of information after an overflow since the counter wraps around and keeps counting while the sticky OF bit remains set.

If an hpmcounter overflows while the associated OF bit is zero, then a *count overflow interrupt request* is generated. If an hpmcounter overflows while the associated OF bit is one, then no interrupt is generated. Consequently, the OF bit also functions as a count overflow interrupt disable for the associated hpmcounter.

Count overflow never results from writes to the mhpcounterX or mhpmeventX registers. Overflow occurs only as a result of an event causing a counter to increment.

The counter overflow interrupt is a standard local interrupt that corresponds to bit 13 in the mip and mie registers. The mip LCOFIP bit is the interrupt-pending bit for this interrupt and the mie LCOFIE bit is the interrupt-enable bit for this interrupt. LCOFI represents Local Count Overflow Interrupt. If S-mode is present, sip and sie include the corresponding bits for supervisor interrupt control and status.

Generation of a count overflow interrupt request by an hpmcounter sets the LCOFIP bit in the mip register and sets the associated OF bit. If S-mode is present, the mideleg register controls the delegation of this interrupt to S-mode, which sets the LCOFIP bit in the sip register. The LCOFIP bit is cleared by software after servicing the count overflow interrupt.

Multiple simultaneous interrupts destined for the same privilege mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

Note that there are not separate overflow status and overflow interrupt enable bits. In practice, enabling overflow interrupt generation by clearing the OF bit is done in conjunction with initializing the counter to a starting value. Once a counter has overflowed, it and the OF bit must be reinitialized before another overflow interrupt can be generated.

Software can distinguish newly-overflowed counters which have yet to be serviced by an overflow interrupt handler from overflowed counters that have already been serviced (or that are configured not to generate an interrupt on overflow by maintaining a bit mask reflecting which counters are active and due to overflow eventually).

Disabling Counters in Debug Mode

When set, the `dcsr.stopcount` bit stops counters while in debug mode. This is especially important for `mcycle` and `minstret` counters, since in debug mode, the core is executing ROM instructions in a tight loop. The Freedom Studio Performance Monitor View automatically sets the `dcsr.stopcount` bit.

3.9.5 Event Selector Encodings

Table 10 describes the event selector encodings available. Events are categorized into classes based on the Event Class field encoded in `mhpmeventx[7:0]`. One or more events can be programmed by setting the respective Event Mask bit for a given event class. An event selector encoding of 0 means "count nothing". Multiple events will cause the counter to increment any time any of the selected events occur.

| Machine Hardware Performance Monitor Event Register | |
|--|--|
| Instruction Commit Events, mhpmeventX[7:0]=0x0 | |
| Bits | Description |
| 8 | Exception taken |
| 9 | Integer load instruction retired |
| 10 | Integer store instruction retired |
| 11 | Atomic memory operation retired |
| 12 | System instruction retired |
| 13 | Integer arithmetic instruction retired |
| 14 | Conditional branch retired |
| 15 | JAL instruction retired |
| 16 | JALR instruction retired |
| 17 | Integer multiplication instruction retired |
| 18 | Integer division instruction retired |
| 19 | Floating-point load instruction retired |
| 20 | Floating-point store instruction retired |
| 21 | Floating-point addition retired |
| 22 | Floating-point multiplication retired |
| 23 | Floating-point fused multiply-add retired |
| 24 | Floating-point division or square-root retired |
| 25 | Other floating-point instruction retired |
| Microarchitectural Events, mhpmeventX[7:0]=0x1 | |
| Bits | Description |
| 8 | Address-generation interlock |
| 9 | Long-latency interlock |
| 10 | CSR read interlock |
| 11 | Instruction cache/ITIM busy |
| 12 | Data cache/DTIM busy |
| 13 | Branch direction misprediction |
| 14 | Branch/jump target misprediction |
| 15 | Pipeline flush from CSR write |
| 16 | Pipeline flush from other event |
| 17 | Integer multiplication interlock |
| 18 | Floating-point interlock |
| Memory System Events, mhpmeventX[7:0]=0x2 | |
| Bits | Description |
| 8 | Instruction cache miss |
| 9 | Data cache miss or memory-mapped I/O access |
| 10 | Data cache write-back |

Table 10: mhpmevent Register

Event mask bits that are writable for any event class are writable for all classes. Setting an event mask bit that does not correspond to an event defined in Table 10 has no effect for current

implementations. However, future implementations may define new events in that encoding space, so it is not recommended to program unsupported values into the `mhpmevent` registers.

Combining Events

It is common usage to directly count each respective event. Additionally, it is possible to use combinations of these events to count new, unique events. For example, to determine the average cycles per load from a data memory subsystem, program one counter to count "Data cache/DTIM busy" and another counter to count "Integer load instruction retired". Then, simply divide the "Data cache/DTIM busy" cycle count by the "Integer load instruction retired" instruction count and the result is the average cycle time for loads in cycles per instruction.

It is important to be cognizant of the event types being combined; specifically, event types counting occurrences and event types counting cycles.

3.9.6 Counter-Enable Registers

The 32-bit counter-enable register `mcounteren` controls the availability of the hardware performance-monitoring counters to the next-lowest privileged mode.

The settings in these registers only control accessibility. The act of reading or writing these enable registers does not affect the underlying counters, which continue to increment when not accessible.

When any bit in the `mcounteren` register is clear, attempts to read the cycle, time, instruction retire, or `hpmcounterX` register while executing in U-mode will cause an illegal instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode, U-mode.

`mcounteren` is a **WARL** register. Any of the bits may contain a hardwired value of zero, indicating reads to the corresponding counter will cause an illegal instruction exception when executing in a less-privileged mode.

3.10 Ports

This section describes the Port interfaces to the S7 core.

3.10.1 Front Port

The Front Port can be used by external masters to read from and write into the memory system utilizing any Memory Port in the Core Complex. The ITIM can also be accessed through the Front Port. Transactions cannot be routed from the Front Port to any System or Peripheral Ports.

If a Front Port access targets the Memory Port, a coherency manager is responsible for maintaining coherency with the L1 data cache. A read access can be returned directly from the

cache without generating an external bus access. If a write from the Front Port targets a location allocated in the cache, it results in the line being evicted and invalidated. The write will then proceed to external memory.

Any Front Port access that targets the Memory Port and results in a cache miss will result in an external memory access.

The S76 Core Complex User Guide describes the implementation details of the Front Port.

3.10.2 Memory Port

The Memory Port is used to interface with memory that offers the highest performance for the S76 Core Complex, such as DDR. It supports cacheable accesses for data and instructions.

Consult Section 4.1 for further information about the Memory Port and its Physical Memory Attributes.

See the S76 Core Complex User Guide for a description of the Memory Port implementation in the S76 Core Complex.

3.10.3 Peripheral Port

The Peripheral Port is used to interface with lower speed peripherals and also supports code execution. When a device is attached to the Peripheral Port, it is expected that there are no other masters connected to that device.

Consult Section 4.1 for further information about the Peripheral Port and its Physical Memory Attributes.

See the S76 Core Complex User Guide for a description of the Peripheral Port implementation in the S76 Core Complex.

3.10.4 System Port

The System Port is used to interface with lower performance memory, like SRAM, memory-mapped I/O (MMIO), and higher speed peripherals. The System Port also supports code execution.

Consult Section 4.1 for further information about the System Port and its Physical Memory Attributes.

See the S76 Core Complex User Guide for a description of the System Port implementation in the S76 Core Complex.

Chapter 4

Physical Memory Attributes and Memory Map

This chapter describes the S76 Core Complex physical memory attributes and memory map.

4.1 Physical Memory Attributes Overview

The memory map is divided into different regions covering on-core-complex memory, system memory, peripherals, and empty holes. Physical memory attributes (PMAs) describe the properties of the accesses that can be made to each region in the memory map. These properties encompass the type of access that may be performed: execute, read, or write. As well as other optional attributes related to the access, such as supported access size, alignment, atomic operations, and cacheability.

RISC-V utilizes a simpler approach than other processor architectures in defining the attributes of memory accesses. Instead of defining access characteristics in page table descriptors or memory protection logic, the properties are fixed for memory regions or may only be modified in platform-specific control registers. As most systems don't require the ability to modify PMAs, SiFive cores only support fixed PMAs, which are set at design time. This results in a simpler design with lower gate count and power savings, and an easier programming interface.

External memory map regions are accessed through a specific port type and that port type is used to define the PMAs. The port types are Memory, Peripheral, and System. Memory map regions defined for internal memory and internal control regions also have a predefined PMA based on the underlying contents of the region.

The assigned PMA properties and attributes for S76 Core Complex memory regions are shown in Table 11 and Table 12 for external and internal regions, respectively.

The configured memory regions of the S76 Core Complex are listed with their attributes in Table 13.

| Port Type | Access Properties | Attributes |
|-----------------|----------------------|---|
| Memory Port | Read, Write, Execute | Atomics+LR/SC, Data Cacheable, Instruction Cacheable, Instruction Speculation |
| Peripheral Port | Read, Write, Execute | Atomics, Instruction Cacheable |
| System Port | Read, Write, Execute | Instruction Cacheable |

Table 11: Physical Memory Attributes for External Regions

| Region | Access Properties | Attributes |
|------------------|----------------------|----------------------------------|
| CLINT | Read, Write | Atomics |
| Data Local Store | Read, Write, Execute | Atomics, Instruction Cacheable |
| Debug | None | N/A |
| Error Device | Read, Write, Execute | Atomics |
| ITIM | Read, Write, Execute | Atomics, Instruction Speculation |
| PLIC | Read, Write | Atomics |
| Reserved | None | N/A |
| SLPC | Read, Write | Atomics |

Table 12: Physical Memory Attributes for Internal Regions

All memory map regions support word, half-word, and byte size data accesses.

Atomic access support enables the RISC-V standard Atomic (A) Extension for atomic instructions. These atomic instructions are further documented in Section 3.6 for the S7 core. The load-reserved (LR) and store-conditional (SC) instructions are only supported on the data cacheable region, marked in Table 11 with "Atomics+LR/SC".

No region supports unaligned accesses. An unaligned access will generate the appropriate trap: instruction address misaligned, load address misaligned, or store/AMO address misaligned.

The Physical Memory Protection unit is capable of controlling access properties based on address ranges, not ports. It has no control over the attributes of an address range, however.

Note

The Debug and Error Device regions have special behavior. The Debug region is reserved for use from a Debugger, and all accesses to it from the core in non-Debug mode will trap. The Error Device will also trap all accesses, as described in Chapter 10.

4.2 Memory Map

The memory map of the S76 Core Complex is shown in Table 13.

| Base | Top | PMA | Description |
|-------------|-------------|--------|---------------------------|
| 0x0000_0000 | 0x0000_0FFF | | Debug |
| 0x0000_1000 | 0x0000_2FFF | | Reserved |
| 0x0000_3000 | 0x0000_3FFF | RWX A | Error Device |
| 0x0000_4000 | 0x017F_FFFF | | Reserved |
| 0x0180_0000 | 0x0180_7FFF | RWX A | ITIM |
| 0x0180_8000 | 0x01FF_FFFF | | Reserved |
| 0x0200_0000 | 0x0200_FFFF | RW A | CLINT |
| 0x0201_0000 | 0x0300_7FFF | | Reserved |
| 0x0300_8000 | 0x0300_8FFF | RW A | SLPC |
| 0x0300_9000 | 0x0BFF_FFFF | | Reserved |
| 0x0C00_0000 | 0x0FFF_FFFF | RW A | PLIC |
| 0x1000_0000 | 0x1FFF_FFFF | | Reserved |
| 0x2000_0000 | 0x3FFF_FFFF | RWXI A | Peripheral Port (512 MiB) |
| 0x4000_0000 | 0x5FFF_FFFF | RWXI | System Port (512 MiB) |
| 0x6000_0000 | 0x6FFF_FFFF | | Reserved |
| 0x7000_0000 | 0x7000_7FFF | RWXI A | Data Local Store |
| 0x7000_8000 | 0x7FFF_FFFF | | Reserved |
| 0x8000_0000 | 0x9FFF_FFFF | RWXIDA | Memory Port (512 MiB) |
| 0xA000_0000 | 0xFFFF_FFFF | | Reserved |

Table 13: S76 Core Complex Memory Map. Physical Memory Attributes: **R**–Read, **W**–Write, **X**–Execute, **I**–Instruction Cacheable, **D**–Data Cacheable, **A**–Atomics

Chapter 5

Programmer's Model

The S76 Core Complex implements the 64-bit RISC-V architecture. The following chapter provides a reference for programmers and an explanation of the extensions supported by *RV64GC_Zba_Zbb_Sscofpmf*.

This chapter contains a high-level discussion of the RISC-V instruction set architecture and additional resources which will assist software developers working with RISC-V products. The S76 Core Complex is an implementation of the RISC-V *RV64GC_Zba_Zbb_Sscofpmf* architecture, and is guaranteed to be compatible with all applicable RISC-V standards. *RV64GC_Zba_Zbb_Sscofpmf* can emulate almost any other RISC-V ISA extension.

5.1 Base Instruction Formats

RISC-V base instructions are fixed to 32 bits in length and must be aligned on a four-byte boundary in memory. RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding, with the exception of the 5-bit immediates used in CSR instructions.

The various formats are described in Table 14 below.

| Format | Description |
|--------|---|
| R | Format for register-register arithmetic/logical operations. |
| I | Format for register-immediate ALU operations and loads. |
| S | Format for stores. |
| B | Format for branches. |
| U | Format for 20-bit upper immediate instructions. |
| J | Format for jumps. |

Table 14: Base Instruction Formats

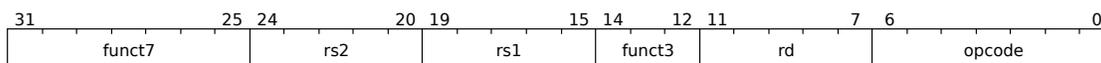


Figure 9: R-Type

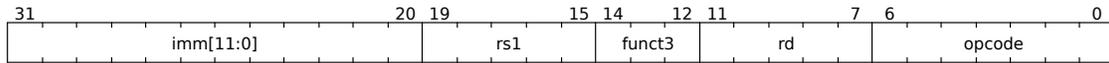


Figure 10: I-Type

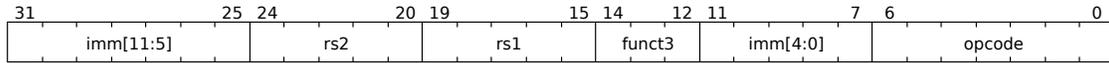


Figure 11: S-Type

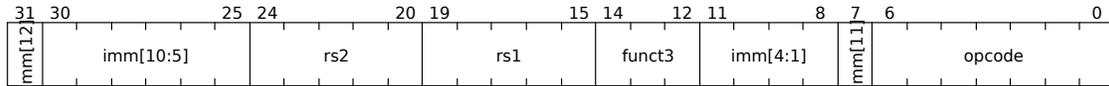


Figure 12: B-Type

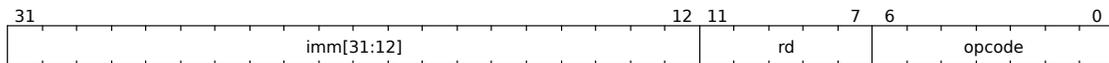


Figure 13: U-Type

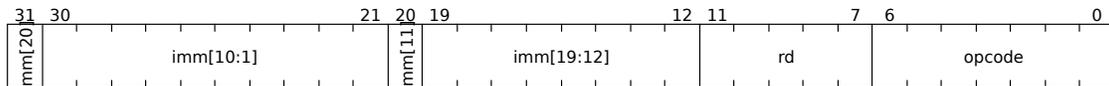


Figure 14: J-Type

The **opcode** field partially specifies an instruction, combined with **funct7 + funct3** which describe what operation to perform. Each register field (**rs1**, **rs2**, **rd**) holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0 - x31). Sign-extension is one of the most critical operations on immediates (particularly for XLEN>32), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

5.2 RV64I Base Integer Instruction Set

This section discusses the standard integer instructions supported by RISC-V. Integer computational instructions don't cause arithmetic exceptions.

5.2.2 I-Type Integer Instructions

For I-Type integer instruction, one field is different from R-format. `rs2` and `funct7` are replaced by the 12-bit signed immediate, `imm[11:0]`, which can hold values in range `[-2048, +2047]`. The immediate is always sign-extended to 32-bits before being used in an arithmetic operation. Bits `[31:12]` receive the same value as bit 11.

| imm | | | func3 | | opcode | Instruction |
|-----------|--------|-----|-------|----|---------|-------------|
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 00000000 | shamnt | rs1 | 001 | rd | 0010011 | SLLI |
| 00000000 | shamnt | rs1 | 101 | rd | 0010011 | SRLI |
| 01000000 | shamnt | rs1 | 001 | rd | 0010011 | SRAI |

Table 17: I-Type Integer Instructions

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI).

| Instruction | Description |
|-------------|--|
| ADDI | Adds the sign-extended 12-bit immediate to register <i>rs1</i> . Arithmetic overflow is ignored and the result is simply the low 64-bits of the result. <code>ADDI rd, rs1, 0</code> is used to implement the <code>MV rd, rs1</code> assembler pseudoinstruction. |
| SLTI | Set less than immediate. Places the value 1 in register <i>rd</i> if register <i>rs1</i> is less than the sign extended immediate when both are treated as signed numbers, else 0 is written to <i>rd</i> . |
| SLTIU | Compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 64-bits then treated as an unsigned number). Note: <code>SLTIU rd, rs1, 1</code> sets <i>rd</i> to 1 if <i>rs1</i> equals zero, otherwise sets <i>rd</i> to 0 (assembler pseudo instruction <code>SEQZ rd, rs</code>). |
| XORI | Bitwise XOR on register <i>rs1</i> and the sign-extended 12-bit immediate and place the result in <i>rd</i> . |
| ORI | Bitwise OR on register <i>rs1</i> and the sign-extended 12-bit immediate and place the result in <i>rd</i> . |
| ANDI | Bitwise AND on register <i>rs1</i> and the sign-extended 12-bit immediate and place the result in <i>rd</i> . |
| SLLI | Shift Left Logical. The operand to be shifted is in <i>rs1</i> , and the shift amount is encoded in the lower 5 bits of the I-immediate field. |
| SRLI | Shift Right Logical. The operand to be shifted is in <i>rs1</i> , and the shift amount is encoded in the lower 5 bits of the I-immediate field. |
| SRAI | Shift Right Arithmetic. The operand to be shifted is in <i>rs1</i> , and the shift amount is encoded in the lower 5 bits of the I-immediate field (the original sign bit is copied into the vacated upper bits). |

Table 18: I-Type Integer Instruction Description

Shift-by-immediate instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions).

Below is an example of an ADDI instruction.

addi x15, x1, -50

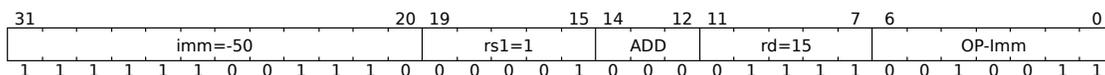


Figure 16: ADDI Instruction Example

5.2.3 I-Type Load Instructions

For I-Type load instructions, a 12-bit signed immediate is added to the base address in register *rs1* to form the memory address. In Table 19 below, **funct3** field encodes size and signedness of load data.

| imm | | | func3 | imm | opcode | Instruction |
|-----------|-----|-----|-------|----------|----------|-------------|
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 01000011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 01000011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 01000011 | SW |

Table 21: S-Type Store Instructions

| Instruction | Description |
|---------------------------|---|
| SB rs2, imm[11:0](rs1) | Store 8-bit value from the low bits of register rs2 to memory. |
| SH rs2, imm[11:0](rs1) | Store 16-bit value from the low bits of register rs2 to memory. |
| SW rs2, imm[11:0](rs1) | Store 32-bit value from the low bits of register rs2 to memory. |

Table 22: S-Type Store Instruction Description

Below is an example SW instruction.

sw x14, 8(x2)

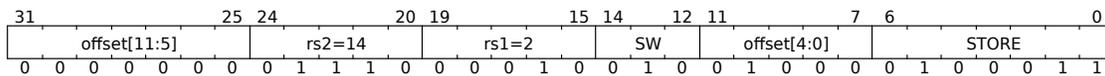


Figure 19: SW Instruction Example

5.2.5 Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ± 1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

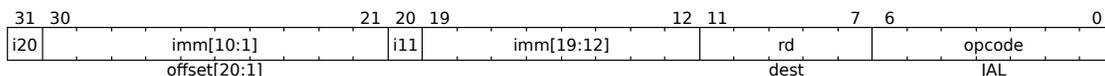


Figure 20: JAL Instruction

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.

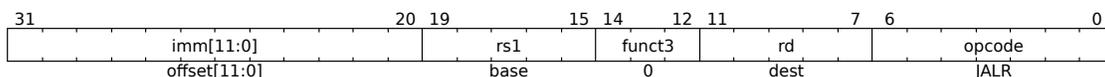


Figure 21: JALR Instruction

Both JAL and JALR instructions will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary.

| Instruction | Description |
|-------------------------|------------------------|
| JAL rd, imm[20:1] | Jump and link |
| JALR rd, rs1, imm[11:0] | Jump and link register |

Table 23: J-Type Instruction Description

5.2.6 Conditional Branches

All branch instructions use the B-Type instruction format. The 12-bit immediate represents values -4096 to +4094 in 2-byte increments. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ± 4 KiB.

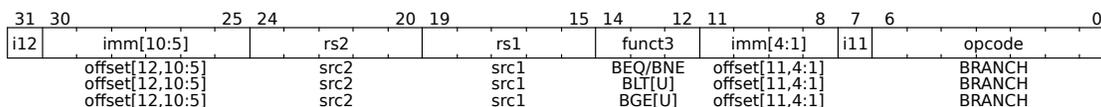


Figure 22: Branch Instructions

| imm | rs2 | rs1 | funct3 | imm | opcode | Instruction |
|--------------|-----|-----|--------|-------------|--------|-------------|
| imm[12,10:5] | rs2 | rs1 | 000 | imm[4:1,11] | 110011 | BEQ |
| imm[12,10:5] | rs2 | rs1 | 001 | imm[4:1,11] | 110011 | BNE |
| imm[12,10:5] | rs2 | rs1 | 100 | imm[4:1,11] | 110011 | BLT |
| imm[12,10:5] | rs2 | rs1 | 101 | imm[4:1,11] | 110011 | BGE |
| imm[12,10:5] | rs2 | rs1 | 110 | imm[4:1,11] | 110011 | BLTU |
| imm[12,10:5] | rs2 | rs1 | 111 | imm[4:1,11] | 110011 | BGEU |

Table 24: B-Type Instructions

| Instruction | Description |
|--------------------------|--|
| BEQ rs1, rs2, imm[12:1] | Take the branch if registers rs1 and rs2 are equal. |
| BNE rs1, rs2, imm[12:1] | Take the branch if registers rs1 and rs2 are unequal. |
| BLT rs1, rs2, imm[12:1] | Take the branch if rs1 is less than rs2. |
| BGE rs1, rs2, imm[12:1] | Take the branch if rs1 is greater than or equal to rs2. |
| BLTU rs1, rs2, imm[12:1] | Take the branch if rs1 is less than rs2 (unsigned). |
| BGEU rs1, rs2, imm[12:1] | Take the branch if rs1 is greater than or equal to rs2 (unsigned). |

Table 25: B-Type Instruction Description

| ISA Base Instruction | Pseudoinstruction | Description |
|---------------------------|------------------------|--|
| BEQ <i>rs, x0, offset</i> | BEQZ <i>rs, offset</i> | Take the branch if <i>rs</i> is equal to zero. |

Table 26: RISC-V Base Instruction to Assembly Pseudoinstruction Example

Note

Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

5.2.7 Upper-Immediate Instructions

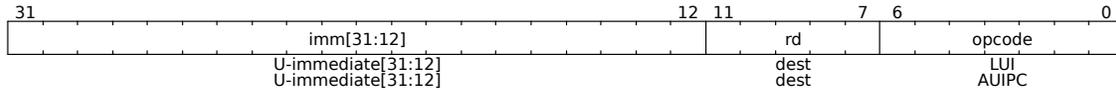


Figure 23: Upper-Immediate Instructions

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros. Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

For example:

LUI x10, 0x87654 # x10 = 0x8765_4000

ADDI x10, x10, 0x321 # x10 = 0x8765_4321

AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, and adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

5.2.8 Memory Ordering Operations

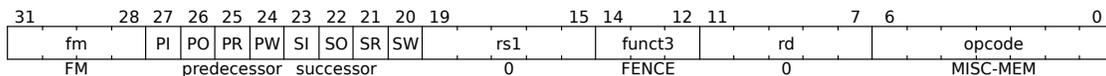


Figure 24: FENCE Instructions

The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device

output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. These operations are discussed further in Section 5.14.

5.2.9 Environment Call and Breakpoints

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions.

5.2.10 NOP Instruction

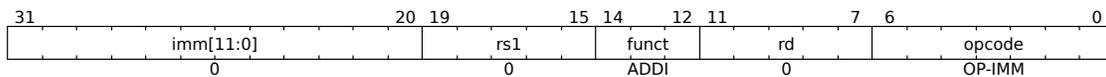


Figure 25: NOP Instructions

The NOP instruction does not change any architecturally visible state, except for advancing the pc and incrementing any applicable performance counters. NOP is encoded as **ADDI x0, x0, 0**.

5.3 M Extension: Multiplication Operations

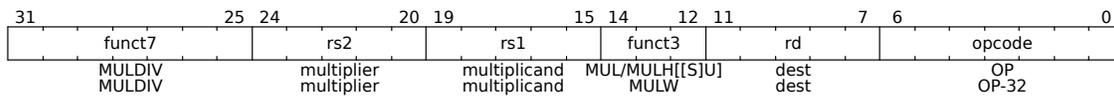


Figure 26: Multiplication Operations

| Instruction | Description |
|---------------------|--|
| MUL rd, rs1, rs2 | Multiplication of rs1 by rs2 and places the lower 64-bits in the destination register. |
| MULH rd, rs1, rs2 | Multiplication that return the upper 64-bits of the full 2×64-bit product. |
| MULHU rd, rs1, rs2 | Unsigned multiplication that return the upper 64-bits of the full 2×64-bit product. |
| MULHSU rd, rs1, rs2 | Signed rs1 multiple unsigned rs2 that return the upper 64-bits of the full 2×64-bit product. |
| MULW rd, rs1, rs2 | RV64 instruction that multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register. |

Table 27: Multiplication Operation Description

Combining MUL and MULH together creates one multiplication operation.

5.3.1 Division Operations

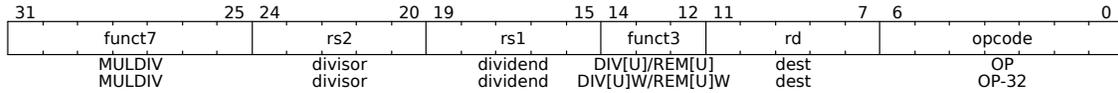


Figure 27: Division Operations

| Instruction | Description |
|--------------------|---|
| DIV rd, rs1, rs2 | 64-bits by 64-bits signed division of r1 by rs2 rounding towards zero. |
| DIVU rd, rs1, rs2 | 64-bits by 64-bits unsigned division of r1 by rs2 rounding towards zero. |
| REM rd, rs1, rs2 | Remainder of the corresponding division. |
| REMU rd, rs1, rs2 | Unsigned remainder of the corresponding division. |
| DIVW rd, rs1, rs2 | RV64 instruction. Signed divide the lower 32 bits of rs1 by the lower 32 bits of rs2. |
| DIVUW rd, rs1, rs2 | RV64 instruction. Unsigned divide the lower 32 bits of rs1 by the lower 32 bits of rs2. |
| REMW rd, rs1, rs2 | Singed remainder. |
| REMUW rd, rs1, rs2 | Unsigned remainder sign-extend the 32-bit result to 64 bits, including on a divide by zero. |
| MULDIV rd, rs1, rs | Multiply Divide. |

Table 28: Division Operation Description

Combining DIV and REM together creates one division operation.

5.4 A Extension: Atomic Operations

Atomic operations are defined as operations that automatically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space.

5.4.1 Atomic Load-Reserve and Store-Conditional Instructions

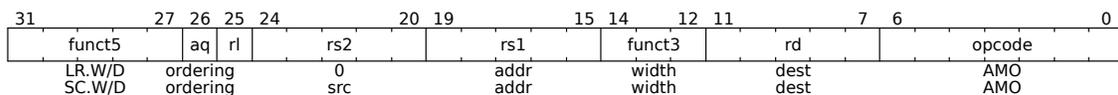


Figure 28: Atomic Operations

| Instruction | Description |
|-------------|--|
| LR.W | Load Reserve. Loads a word from the address in <i>rs1</i> , places the sign-extended value in <i>rd</i> , and registers a reservation set—a set of bytes that subsumes the bytes in the addressed word. |
| SC.W | Store Conditional Conditionally writes a word in <i>rs2</i> to the address in <i>rs1</i> : the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the SC.W succeeds, the instruction writes the word in <i>rs2</i> to memory, and it writes zero to <i>rd</i> . If the SC.W fails, the instruction does not write to memory, and it writes a nonzero value to <i>rd</i> . Executing an SC.W instruction invalidates any reservation held by this hart. |
| LR.D | RV64 - Loads doubleword. |
| SC.D | RV64 - Stores doubleword. |

Table 29: Atomic Load-Reserve and Store-Conditional Instruction Description

For RV64, the sign-extended value of LR.W and SC.W is placed in *rd*.

Note

Only cores with data caches support the LR/SC instructions used by the A-Extension. Cores with DTIMs will *NOT*.

5.4.2 Atomic Memory Operations (AMOs)

The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization. These AMO instructions atomically load a data value from the address in *rs1*, place the value into register *rd*, apply a binary operator to the loaded value and the original value in *rs2*, then store the result back to the address in *rs1*.

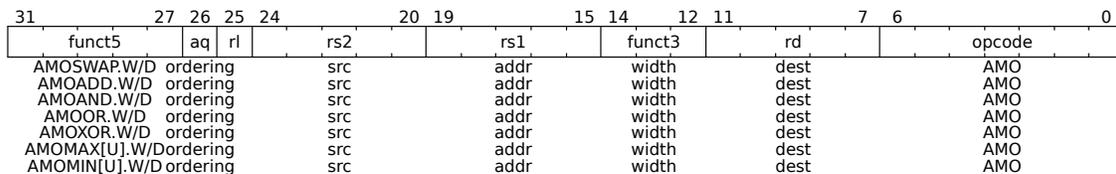


Figure 29: Atomic Memory Operations

| Instruction | Description |
|-------------|-------------------------------------|
| AMOSWAP.W/D | Word / doubleword swap. |
| AMOADD.W/D | Word / doubleword add. |
| AMOAND.W/D | Word / doubleword and. |
| AMOOR.W/D | Word / doubleword or. |
| AMOXOR.W/D | Word / doubleword xor. |
| AMOMIN.W/D | Word / doubleword minimum. |
| AMOMINU.W/D | Unsigned word / doubleword minimum. |
| AMOMAX.W/D | Word / doubleword maximum. |
| AMOMAXU.W/D | Unsigned word / doubleword maximum. |

Table 30: Atomic Memory Operation Description

For RV64, 32-bit AMOs always sign-extend the value placed in rd.

5.5 F Extension: Single-Precision Floating-Point Instructions

The F Extension implements single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The F Extension adds 32 floating-point registers, f0–f31, each 32 bits wide, and a floating-point control and status register fcsr. Floating-point load and store instructions transfer floating-point values between registers and memory, and instructions to transfer values to and from the integer register file are also provided.

5.5.1 Floating-Point Control and Status Registers

Floating-Point Control and Status Register, fcsr, is a RISC-V control and status register (CSR). The register selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags.

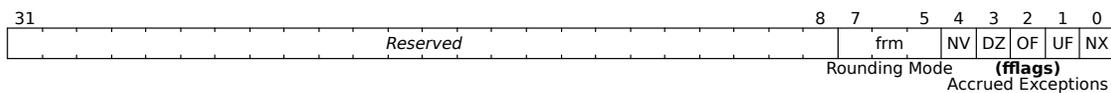


Figure 30: Floating-Point Control and Status Register

| Flag Mnemonic | Flag Meaning |
|---------------|-------------------|
| NV | Invalid Operation |
| DZ | Divide by Zero |
| OF | Overflow |
| UF | Underflow |
| NX | Inexact |

Table 31: Accrued Exception Flags

The `fcsr` register can be read and written with the `FRCSR` and `FSCSR` instructions. The `FRRM` instruction reads the Rounding Mode field `frm`. `FRRM` swaps the value in `frm` with an integer register. `FRFLAGS` and `FSFLAGS` are defined analogously for the Accrued Exception Flags field `fflags`.

5.5.2 Rounding Modes

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. A value of 111 in the instruction's `rm` field selects the dynamic rounding mode held in `frm`. If `frm` is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will raise an illegal instruction exception. Some instructions, including widening conversions, have the `rm` field, but are nevertheless unaffected by the rounding mode. Software should set their `rm` field to `RNE` (000).

| Mode | Mnemonic | Meaning |
|------|----------|--|
| 000 | RNE | Round to Nearest, ties to Even. |
| 001 | RTZ | Round towards Zero. |
| 010 | RDN | Round Down (towards $-\infty$). |
| 011 | RUP | Round Up (towards $+\infty$). |
| 100 | RMM | Round to Nearest, ties to Max Magnitude. |
| 101 | | <i>Invalid. Reserved for future use.</i> |
| 110 | | <i>Invalid. Reserved for future use.</i> |
| 111 | DYN | In instruction's <code>rm</code> field, selects dynamic rounding mode; In Rounding Mode register, <i>Invalid</i> . |

Table 32: Floating-Point Rounding Modes

5.5.3 Single-Precision Floating-Point Load and Store Instructions

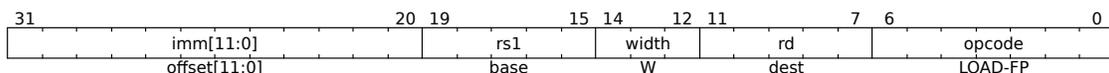


Figure 31: Single-Precision FP Load Instruction

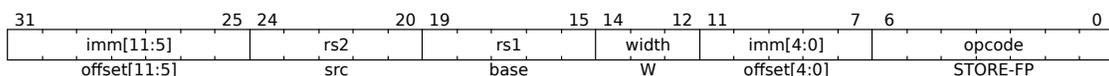


Figure 32: Single-Precision FP Store Instruction

| Instruction | Operation | Description |
|--------------------------------|---|--|
| FLW <code>rd, rs1, imm</code> | $f[rd] = M[x[rs1] + sext(offset)][31:0]$ | Loads a single-precision floating-point value from memory into floating-point register <code>rd</code> . |
| FSW <code>imm, rs1, rs2</code> | $M[x[rs1] + sext(offset)] = f[rs2][31:0]$ | Stores a single-precision value from floating-point register <code>rs2</code> to memory. |

Table 33: Single-Precision FP Load and Store Instructions Description

5.5.4 Single-Precision Floating-Point Computational Instructions

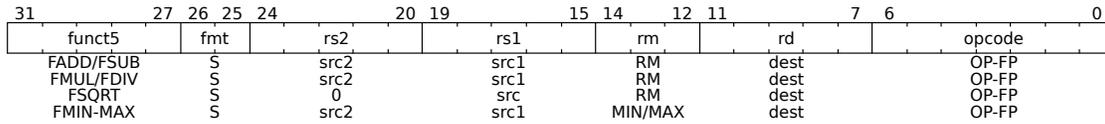


Figure 33: Single-Precision FP Computational Instructions

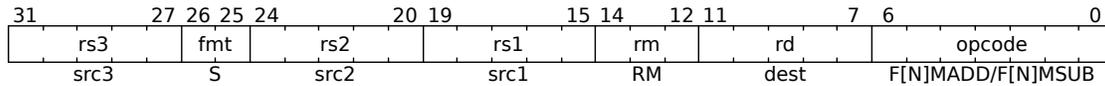


Figure 34: Single-Precision FP Fused Computational Instructions

| Instruction | Operation | Description |
|-------------------------|--|---|
| FADD.S rd,rs1,rs2 | $f[rd] = f[rs1] + f[rs2]$ | Single-precision floating-point addition. |
| FSUB.S rd,rs1,rs2 | $f[rd] = f[rs1] - f[rs2]$ | Single-precision floating-point subtraction. |
| FMUL.S rd,rs1,rs2 | $f[rd] = f[rs1] \times f[rs2]$ | Single-precision floating-point multiplication. |
| FDIV.S rd,rs1,rs2 | $f[rd] = f[rs1] \div f[rs2]$ | Single-precision floating-point division. |
| FSQRT.S rd,rs1 | $f[rd] = \sqrt{f[rs1]}$ | Single-precision floating-point square root. |
| FMIN.S rd,rs1,rs2 | $f[rd] = \min(f[rs1], f[rs2])$ | Single-precision floating-point minimum-number. |
| FMAX.S rd,rs1,rs2 | $f[rd] = \max(f[rs1], f[rs2])$ | Single-precision floating-point maximum-number. |
| FMADD.S rd,rs1,rs2,rs3 | $f[rd] = (f[rs1] \times f[rs2]) + f[rs3]$ | Single-precision floating-point multiply and add. |
| FMSUB.S rd,rs1,rs2,rs3 | $f[rd] = (f[rs1] \times f[rs2]) - f[rs3]$ | Single-precision floating-point multiply and subtract. |
| FNMADD.S rd,rs1,rs2,rs3 | $f[rd] = -(f[rs1] \times f[rs2]) + f[rs3]$ | Single-precision floating-point multiply, negate, and add. |
| FNMSUB.S rd,rs1,rs2,rs3 | $f[rd] = -(f[rs1] \times f[rs2]) - f[rs3]$ | Single-precision floating-point multiply, negate, and subtract. |

Table 34: Single-Precision FP Computational Instructions Description

5.5.5 Single-Precision Floating-Point Conversion and Move Instructions

Single-Precision Floating-Point Conversion Instructions

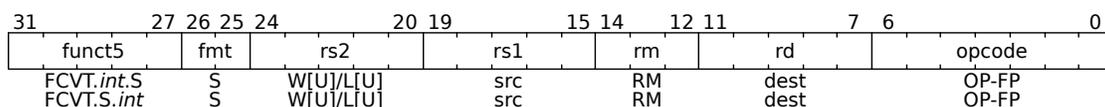


Figure 35: Single-Precision FP to Integer and Integer to FP Conversion Instructions

| Instruction | Operation | Description |
|-------------------|--|--|
| FCVT.W.S rd, rs1 | $x[rd] = \text{sext}(s32_{f32}(f[rs1]))$ | Converts a single-precision floating-point number to a signed 32-bit integer. Sign-extends the 32-bit result to the destination register width. |
| FCVT.S.W rd, rs1 | $f[rd] = f32_{s32}(x[rs1])$ | Converts a signed 32-bit integer to a single-precision floating-point number. |
| FCVT.WU.S rd, rs1 | $x[rd] = \text{sext}(u32_{f32}(f[rs1]))$ | Converts a single-precision floating-point number to an unsigned 32-bit integer. Sign-extends the 32-bit result to the destination register width. |
| FCVT.S.WU rd, rs1 | $f[rd] = f32_{u32}(x[rs1])$ | Converts an unsigned 32-bit integer to a single-precision floating-point number. |
| FCVT.L.S rd, rs1 | $x[rd] = s64_{f32}(f[rs1])$ | Converts a single-precision floating-point number to a signed 64-bit integer. |
| FCVT.S.L rd, rs1 | $f[rd] = f32_{s64}(x[rs1])$ | Converts a signed 64-bit integer to a single-precision floating-point number. |
| FCVT.LU.S rd, rs1 | $x[rd] = u64_{f32}(f[rs1])$ | Converts a single-precision floating-point number to an unsigned 64-bit integer. |
| FCVT.S.LU rd, rs1 | $f[rd] = f32_{u64}(x[rs1])$ | Converts an unsigned 64-bit integer to a single-precision floating-point number. |

Table 35: Single-Precision FP Conversion Instructions Description

If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set.

Single-Precision Floating-Point-to-Floating-Point Sign-Injection Instructions

The floating-point-to-floating-point sign-injection instructions produce a result that takes all bits except the sign bit from rs1. The sign-injection instructions provide floating-point MV, ABS and NEG.

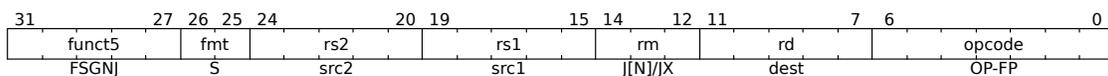


Figure 36: Single-Precision FP to FP Sign-Injection Instructions

| Instruction | Operation | Description |
|-----------------------|--|--|
| FSGNJ.S rd, rs1, rs2 | $f[rd] = \{f[rs2][31], f[rs1][30:0]\}$ | Produces a result that takes all bits except the sign bit from rs1. The result's sign bit is rs2's sign bit. |
| FSGNJS rd, rs1, rs2 | $f[rd] = \{-f[rs2][31], f[rs1][30:0]\}$ | Produces a result that takes all bits except the sign bit from rs1. The result's sign bit is the opposite of rs2's sign bit. |
| FSGNJX.S rd, rs1, rs2 | $f[rd] = \{f[rs1][31] \wedge f[rs2][31], f[rs1][30:0]\}$ | Produces a result that takes all bits except the sign bit from rs1. The sign bit is the XOR of the sign bits of rs1 and rs2. |

Table 36: Single-Precision FP to FP Sign-Injection Instructions Description

| ISA Base Instruction | Pseudoinstruction | Description |
|----------------------|-------------------|---------------------------------------|
| FSGNJ.S rx, ry, ry | FMV.S rx, ry | Moves ry to rx. |
| FSGNJS rx, ry, ry | FNEG.S rx, ry | Moves the negation of ry to rx. |
| FSGNJX.S rx, ry, ry | FABS.S rx, ry | Moves the absolute value of ry to rx. |

Table 37: RISC-V Base Instruction to Assembly Pseudoinstruction Example

Single-Precision Floating-Point Move Instructions

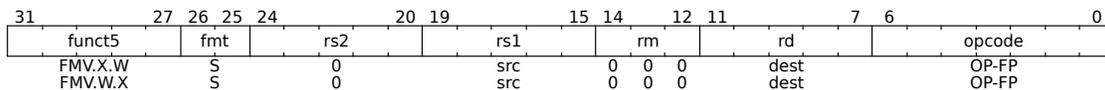


Figure 37: Single-Precision FP Move Instructions

| Instruction | Operation | Description |
|-----------------|------------------------------|---|
| FMV.X.W rd, rs1 | $x[rd] = sext(f[rs1][31:0])$ | Moves the single-precision value in floating-point register rs1 represented in IEEE 754-2008 encoding to the lower 32 bits of integer register rd. The higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit. |
| FMV.W.X rd, rs1 | $f[rd] = x[rs1][31:0]$ | Moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register rs1 to the floating-point register rd. |

Table 38: Single-Precision FP Move Instructions Description

5.5.6 Single-Precision Floating-Point Compare Instructions

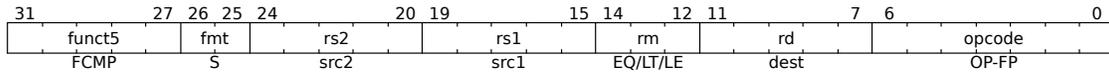


Figure 38: Single-Precision FP Compare Instructions

| Instruction | Operation | Description |
|------------------|------------------------------|--|
| FEQ.S rd,rs1,rs2 | $x[rd] = f[rs1] == f[rs2]$ | Writes 1 to the integer register rd if rs1 is equal to rs2, 0 otherwise. Performs a quiet comparison; only sets the invalid operation exception flag if either input is a signaling NaN. |
| FLT.S rd,rs1,rs2 | $x[rd] = f[rs1] < f[rs2]$ | Writes 1 to the integer register rd if rs1 less than rs2, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN. |
| FLE.S rd,rs1,rs2 | $x[rd] = f[rs1] \leq f[rs2]$ | Writes 1 to the integer register rd if rs1 less than or equal to rs2, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN. |

Table 39: Single-Precision FP Compare Instructions Description

Single-Precision Floating-Point Classify Instruction

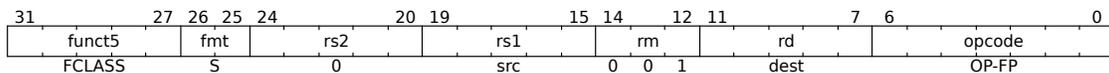


Figure 39: Single-Precision FP Classify Instruction

| Instruction | Operation | Description |
|-----------------|-------------------------------------|--|
| FCLASS.S rd,rs1 | $x[rd] = \text{classify}_s(f[rs1])$ | Examines the value in floating-point register rs1 and writes to integer register rd a 10-bit mask that indicates the class of the floating-point number. |

Table 40: Single-Precision FP Classify Instruction Description

| rd bit | Meaning |
|--------|------------------------------------|
| 0 | rs1 is $-\infty$ |
| 1 | rs1 is negative normal number |
| 2 | rs1 is a negative subnormal number |
| 3 | rs1 is -0 |
| 4 | rs1 is $+0$ |
| 5 | rs1 is a positive subnormal number |
| 6 | rs1 is a positive normal number |
| 7 | rs1 is $+\infty$ |
| 8 | rs1 is a signaling NaN |
| 9 | rs1 is a quiet NaN |

Table 41: Floating-Point Number Classes

5.6 D Extension: Double-Precision Floating-Point Instructions

The D extension widens the 32 floating-point registers, f_0 – f_{31} , to 64 bits. The f registers can now hold either 32-bit or 64-bit floating-point values. When multiple floating-point precisions are supported, then valid values of narrower n-bit types, $n < \text{FLEN}$, are represented in the lower n bits of an FLEN-bit. Any operation that writes a narrower result to an f register must write all 1s to the uppermost FLEN-n bits to yield a legal NaN-boxed value. Floating-point n-bit transfer operations move external values held in IEEE standard formats into and out of the f registers, and comprise floating-point loads and stores and floating-point move instructions.

5.6.1 Double-Precision Floating-Point Load and Store Instructions

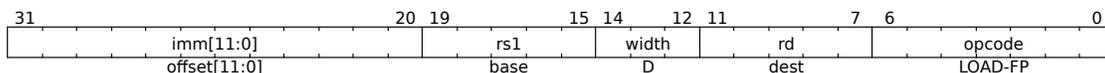


Figure 40: Double-Precision FP Load Instruction

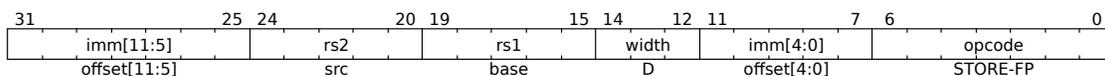


Figure 41: Double-Precision FP Store Instruction

| Instruction | Operation | Description |
|-------------------|---|--|
| FLD rd, rs1, imm | $f[\text{rd}] = M[x[\text{rs1}] + \text{sext}(\text{offset})][63:0]$ | Loads a double-precision floating-point value from memory into floating-point register rd. |
| FSD imm, rs1, rs2 | $M[x[\text{rs1}] + \text{sext}(\text{offset})] = f[\text{rs2}][63:0]$ | Stores a double-precision value from the floating-point register rs2 to memory. |

Table 42: Double-Precision FP Load and Store Instructions Description

FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and $XLEN \geq 64$. These instructions do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

5.6.2 Double-Precision Floating-Point Computational Instructions

The double-precision floating-point computational instructions are defined analogously to their single-precision counterparts, but operate on double-precision operands and produce double-precision results.

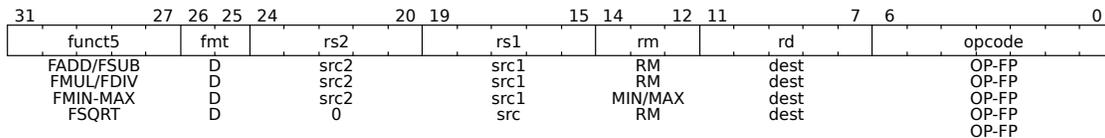


Figure 42: Double-Precision FP Computational Instructions

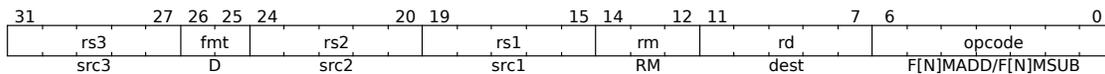


Figure 43: Double-Precision FP Fused Computational Instructions

| Instruction | Operation | Description |
|-------------------------|--|---|
| FADD.D rd,rs1,rs2 | $f[rd] = f[rs1] + f[rs2]$ | Double-precision floating-point addition. |
| FSUB.D rd,rs1,rs2 | $f[rd] = f[rs1] - f[rs2]$ | Double-precision floating-point subtraction. |
| FMUL.D rd,rs1,rs2 | $f[rd] = f[rs1] \times f[rs2]$ | Double-precision floating-point multiplication. |
| FDIV.D rd,rs1,rs2 | $f[rd] = f[rs1] \div f[rs2]$ | Double-precision floating-point division. |
| FSQRT.D rd,rs1 | $f[rd] = \sqrt{f[rs1]}$ | Double-precision floating-point square root. |
| FMIN.D rd,rs1,rs2 | $f[rd] = \min(f[rs1], f[rs2])$ | Double-precision floating-point minimum-number. |
| FMAX.D rd,rs1,rs2 | $f[rd] = \max(f[rs1], f[rs2])$ | Double-precision floating-point maximum-number. |
| FMADD.D rd,rs1,rs2,rs3 | $f[rd] = (f[rs1] \times f[rs2]) + f[rs3]$ | Double-precision floating-point multiply and add. |
| FMSUB.D rd,rs1,rs2,rs3 | $f[rd] = (f[rs1] \times f[rs2]) - f[rs3]$ | Double-precision floating-point multiply and subtract. |
| FNMADD.D rd,rs1,rs2,rs3 | $f[rd] = -(f[rs1] \times f[rs2]) + f[rs3]$ | Double-precision floating-point multiply, negate, and add. |
| FNMSUB.D rd,rs1,rs2,rs3 | $f[rd] = -(f[rs1] \times f[rs2]) - f[rs3]$ | Double-precision floating-point multiply, negate, and subtract. |

Table 43: Double-Precision FP Computational Instructions Description

5.6.3 Double-Precision Floating-Point Conversion and Move Instructions

Double-Precision Floating-Point Conversion Instructions

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field.

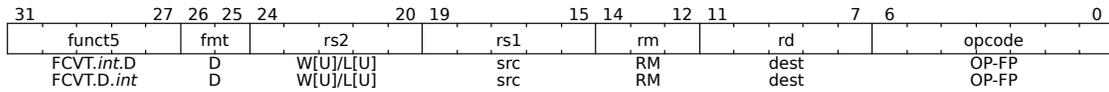


Figure 44: Double-Precision FP to Integer and Integer to FP Conversion Instructions

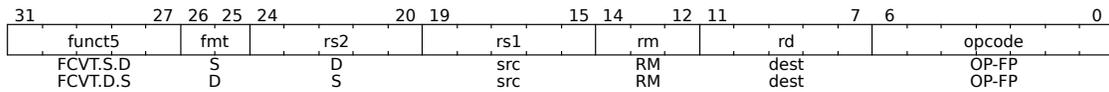


Figure 45: Double-Precision to Single-Precision and Single-Precision to Double-Precision FP Conversion Instructions

| Instruction | Operation | Description |
|-------------------|--|--|
| FCVT.W.D rd, rs1 | $x[rd] = \text{sext}(s32_{f64}(f[rs1]))$ | Converts a double-precision floating-point number to a signed 32-bit integer. Sign-extends the 32-bit result to the destination register width. |
| FCVT.D.W rd, rs1 | $f[rd] = f64_{s32}(x[rs1])$ | Converts a signed 32-bit integer to a double-precision floating-point number. Always produces an exact result and is unaffected by rounding mode. |
| FCVT.WU.D rd, rs1 | $x[rd] = \text{sext}(u32_{f64}(f[rs1]))$ | Converts a double precision floating-point number to an unsigned 32-bit integer. Sign-extends the 32-bit result to the destination register width. |
| FCVT.D.WU rd, rs1 | $f[rd] = f64_{u32}(x[rs1])$ | Converts an unsigned 32-bit integer to a double-precision floating-point number. Always produces an exact result and is unaffected by rounding mode. |
| FCVT.L.D rd, rs1 | $x[rd] = s64_{f64}(f[rs1])$ | Converts a double-precision floating-point number to a signed 64-bit integer. |
| FCVT.D.L rd, rs1 | $f[rd] = f64_{s64}(x[rs1])$ | Converts a signed 64-bit integer to a double-precision floating-point number. |
| FCVT.LU.D rd, rs1 | $x[rd] = u64_{f64}(f[rs1])$ | Converts a double-precision floating-point number to an unsigned 64-bit integer. |
| FCVT.D.LU rd, rs1 | $f[rd] = f64_{u64}(x[rs1])$ | Converts an unsigned 64-bit integer to a double-precision floating-point number. |
| FCVT.S.D rd, rs1 | $f[rd] = f32_{f64}(f[rs1])$ | Converts a double-precision floating-point number to a single-precision floating-point number. |
| FCVT.D.S rd, rs1 | $f[rd] = f64_{f32}(f[rs1])$ | Converts a single-precision floating-point number to a double-precision floating-point number. |

Table 44: Double-Precision FP Conversion Instructions Description

Double-Precision Floating-Point-to-Floating-Point Sign-Injection Instructions

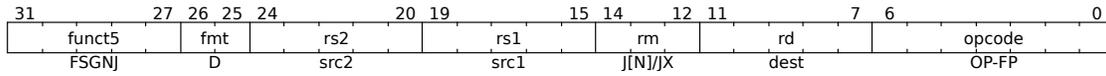


Figure 46: Double-Precision FP to FP Sign-Injection Instructions

| Instruction | Operation | Description |
|-----------------------|--|--|
| FSGNJ.D rd, rs1, rs2 | $f[rd] = \{f[rs2][63], f[rs1][62:0]\}$ | Produces a result that takes all bits except the sign bit from rs1. The result's sign bit is rs2's sign bit. |
| FSGNJD.D rd, rs1, rs2 | $f[rd] = \{-f[rs2][63], f[rs1][62:0]\}$ | Produces a result that takes all bits except the sign bit from rs1. The result's sign bit is the opposite of rs2's sign bit. |
| FSGNJD.D rd, rs1, rs2 | $f[rd] = \{f[rs1][63] \wedge f[rs2][63], f[rs1][62:0]\}$ | Produces a result that takes all bits except the sign bit from rs1. The sign bit is the XOR of the sign bits of rs1 and rs2. |

Table 45: Double-Precision FP to FP Sign-Injection Instructions Description

| ISA Base Instruction | Pseudoinstruction | Description |
|----------------------|-------------------|---------------------------------------|
| FSGNJ.D rx, ry, ry | FMV.D rx, ry | Moves ry to rx. |
| FSGNJD.D rx, ry, ry | FNEG.D rx, ry | Moves the negation of ry to rx. |
| FSGNJD.D rx, ry, ry | FABS.D rx, ry | Moves the absolute value of ry to rx. |

Table 46: RISC-V Base Instruction to Assembly Pseudoinstruction Example

Double-Precision Floating-Point Move Instructions

The RV64 architecture provides instructions to move bit patterns between the floating-point and integer registers.

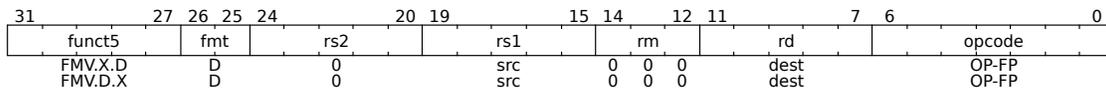


Figure 47: Double-Precision FP Move Instructions

| Instruction | Operation | Description |
|-----------------|------------------------|--|
| FMV.X.D rd, rs1 | $x[rd] = f[rs1][63:0]$ | Moves the double-precision value in floating-point register rs1 to a representation in IEEE 754-2008 standard encoding in integer register rd. |
| FMV.D.X rd, rs1 | $f[rd] = x[rs1][63:0]$ | Moves the double-precision value encoded in IEEE 754-2008 standard encoding from the integer register rs1 to the floating-point register rd. |

Table 47: Double-Precision FP Move Instructions Description

FMV.X.D and FMV.D.X do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

5.6.4 Double-Precision Floating-Point Compare Instructions

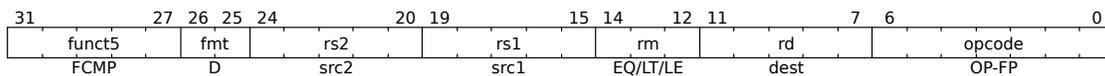


Figure 48: Double-Precision FP Compare Instructions

| Instruction | Operation | Description |
|--------------------|------------------------------|--|
| FEQ.D rd, rs1, rs2 | $x[rd] = f[rs1] == f[rs2]$ | Writes 1 to the integer register rd if rs1 is equal to rs2, 0 otherwise. Performs a quiet comparison; only sets the invalid operation exception flag if either input is a signaling NaN. |
| FLT.D rd, rs1, rs2 | $x[rd] = f[rs1] < f[rs2]$ | Writes 1 to the integer register rd if rs1 less than rs2, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN. |
| FLE.D rd, rs1, rs2 | $x[rd] = f[rs1] \leq f[rs2]$ | Writes 1 to the integer register rd if rs1 less than or equal to rs2, 0 otherwise. Performs signaling comparisons; sets the invalid operation exception flag if either input is NaN. |

Table 48: Double-Precision FP Compare Instructions Description

5.6.5 Double-Precision Floating-Point Classify Instruction

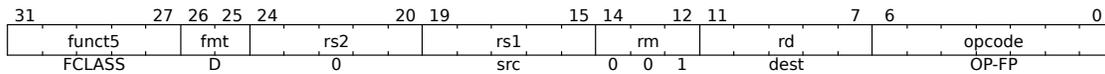


Figure 49: Double-Precision FP Classify Instruction

| Instruction | Operation | Description |
|------------------|-------------------------------------|--|
| FCLASS.D rd, rs1 | $x[rd] = \text{classify}_d(f[rs1])$ | Examines the value in floating-point register rs1 and writes to integer register rd a 10-bit mask that indicates the class of the floating-point number. |

Table 49: Double-Precision FP Classify Instruction Description

5.7 C Extension: Compressed Instructions

The C Extension reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs (RV32, RV64, RV128), and we use the generic term "RVC" to cover any of these. Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction. The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions. It is important to note that the C extension is not designed to be a stand-alone ISA, and is meant to be used alongside a base ISA. The compressed 16-bit instruction format is designed around the assumption that x1 is the return address register and x2 is the stack pointer.

5.7.1 Compressed 16-bit Instruction Formats

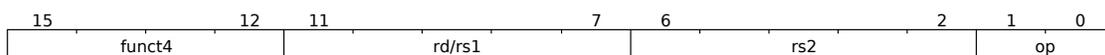


Figure 50: CR Format - Register



Figure 51: CI Format - Immediate

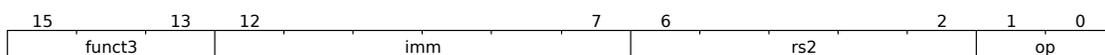


Figure 52: CSS Format - Stack-relative Store



Figure 53: CIW Format - Wide Immediate

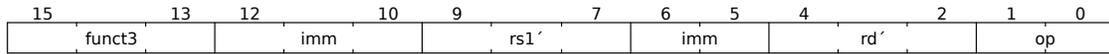


Figure 54: CL Format - Load

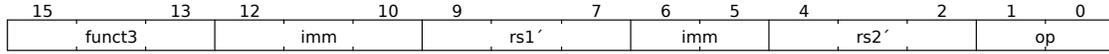


Figure 55: CS Format - Store

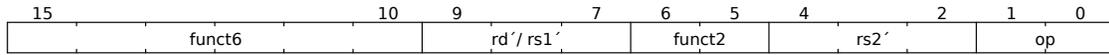


Figure 56: CA Format - Arithmetic

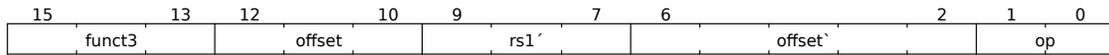


Figure 57: CJ Format - Jump

5.7.2 Stack-Pointed-Based Loads and Stores

The compressed load instructions are expressed in CI format.

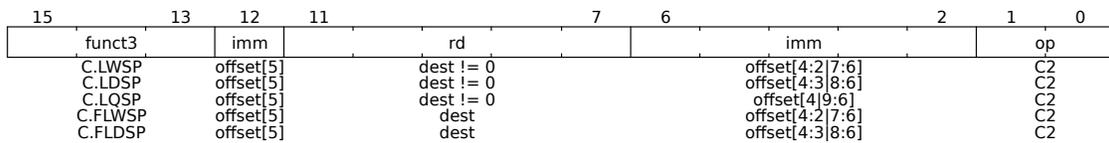


Figure 58: Stack-Pointed-Based Loads

| Instruction | Description |
|-------------|---|
| C.LWSP | Loads a 32-bit value from memory into register rd. |
| C.LDSP | RV64C Instruction which loads a 64-bit value from memory into register rd. |
| C.LQSP | RV128C loads a 128-bit value from memory into register rd. |
| C.FLWSP | RV32FC Instruction that loads a single-precision floating-point value from memory into floating-point register rd. |
| C.FLDSP | RV32DC/RV64DC Instruction that loads a double-precision floating-point value from memory into floating-point register rd. |

Table 50: Stack-Pointed-Based Load Instruction Description

The compressed store instructions are expressed in CSS format.

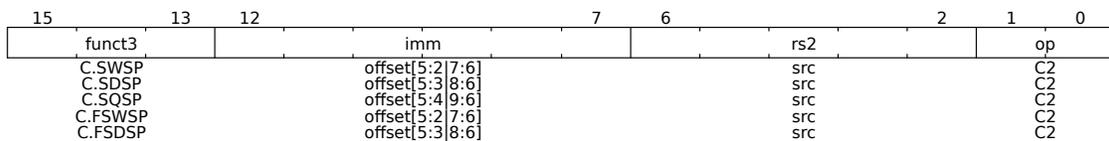


Figure 59: Stack-Pointed-Based Stores

| Instruction | Description |
|-------------|---|
| C.LWSP | Loads a 32-bit value from memory into register rd. |
| C.SWSP | Stores a 32-bit value in register rs2 to memory. |
| C.SDSP | RV64C/RV128C instruction that stores a 64-bit value in register rs2 to memory. |
| C.SQSP | RV128C instruction that stores a 128-bit value in register rs2 to memory. |
| C.FSWSP | RV32FC instruction that stores a single-precision floating-point value in floating-point register rs2 to memory. |
| C.FSDSP | RV32DC/RV64DC instruction that stores a double-precision floating-point value in floating-point register rs2 to memory. |

Table 51: Stack-Pointed-Based Store Instruction Description

5.7.3 Register-Based Loads and Stores

The compressed register-based load instructions are expressed in CL format.

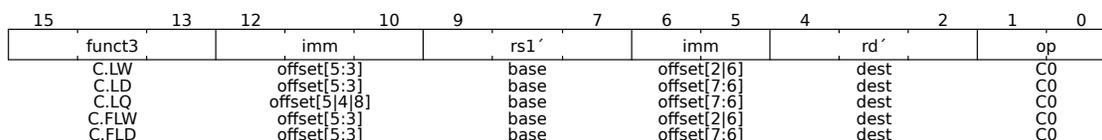


Figure 60: Register-Based Loads

| Instruction | Description |
|-------------|--|
| C.LW | Loads a 32-bit value from memory into register rd. |
| C.LD | RV64C/RV128C-only instruction that loads a 64-bit value from memory into register rd. |
| C.LQ | RV128C-only instruction that loads a 128-bit value from memory into register rd. |
| C.FLW | RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register rd. |
| C.FLD | RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register rd. |

Table 52: Register-Based Load Instruction Description

The compressed register-based store instructions are expressed in CS format.

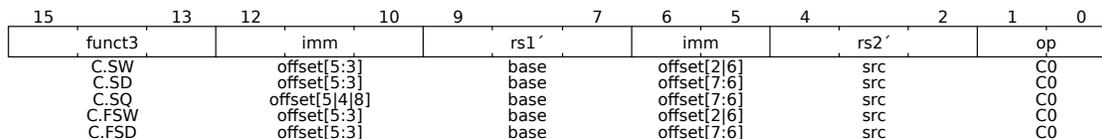


Figure 61: Register-Based Stores

| Instruction | Description |
|-------------|---|
| C.SW | Stores a 32-bit value in register rs2 to memory. |
| C.SD | RV64C/RV128C instruction that stores a 64-bit value in register rs2 to memory. |
| C.SQ | RV128C instruction that stores a 128-bit value in register rs2 to memory. |
| C.FSW | RV32FC instruction that stores a single-precision floating-point value in floating-point register rs2 to memory. |
| C.FSD | RV32DC/RV64DC instruction that stores a double-precision floating-point value in floating-point register rs2 to memory. |

Table 53: Register-Based Store Instruction Description

5.7.4 Control Transfer Instructions

RVC provides unconditional jump instructions and conditional branch instructions.

The unconditional jump instructions are expressed in CJ format.

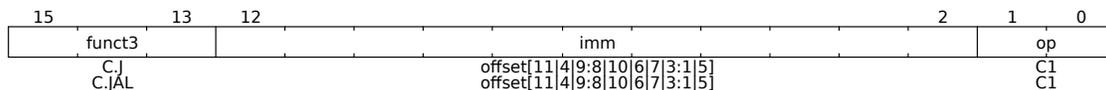


Figure 62: Unconditional Jump Instructions

| Instruction | Description |
|-------------|---|
| C.J | Unconditional control transfer. |
| C.JAL | RV32C instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1. |

Table 54: Unconditional Jump Instruction Description

The unconditional control transfer instructions are expressed in CR format.

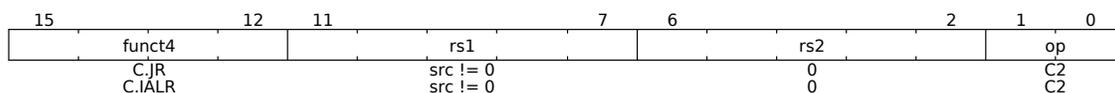


Figure 63: Unconditional Control Transfer Instructions

| Instruction | Description |
|-------------|---|
| C.JR | Performs an unconditional control transfer to the address in register rs1. |
| C.JALR | Performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1. |

Table 55: Unconditional Control Transfer Instruction Description

The conditional control transfer instructions are expressed in CB format.

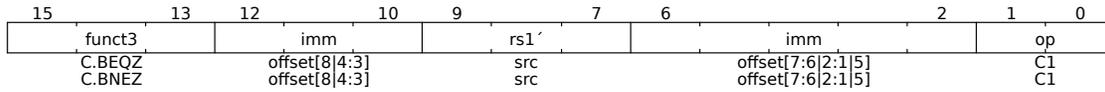


Figure 64: Conditional Control Transfer Instructions

| Instruction | Description |
|-------------|---|
| C.BEQZ | Conditional control transfers. Takes the branch if the value in register <code>rs1'</code> is zero. |
| C.BNEZ | Conditional control transfers. Takes the branch if <code>rs1'</code> contains a nonzero value. |

Table 56: Conditional Control Transfer Instruction Description

5.7.5 Integer Computational Instructions

Integer Constant-Generation Instructions

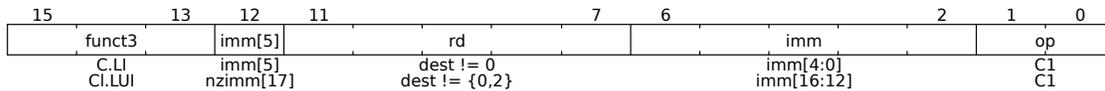


Figure 65: Integer Constant-Generation Instructions

| Instruction | Description |
|-------------|--|
| C.LI | Loads the sign-extended 6-bit immediate, <code>imm</code> , into register <code>rd</code> . |
| C.LUI | Loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination |

Table 57: Integer Constant-Generation Instruction Description

Integer Register-Immediate Operations

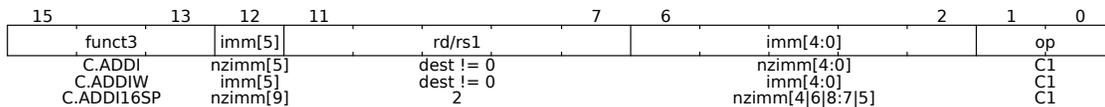


Figure 66: Integer Register-Immediate Operations

| Instruction | Description |
|-------------|--|
| C.ADDI | Adds the non-zero sign-extended 6-bit immediate to the value in register <i>rd</i> then writes the result to <i>rd</i> . |
| C.ADDIW | RV64C/RV128C instruction that performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits. |
| C.ADDI16SP | Adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (<i>sp=x2</i>), where the immediate is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. |

Table 58: Integer Register-Immediate Operation Description

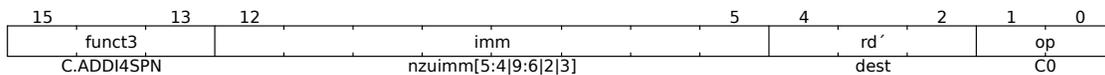


Figure 67: Integer Register-Immediate Operations (cont.)

| Instruction | Description |
|-------------|---|
| C.ADDI4SPN | Adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, <i>x2</i> , and writes the result to <i>rd'</i> . |

Table 59: Integer Register-Immediate Operation Description (cont.)

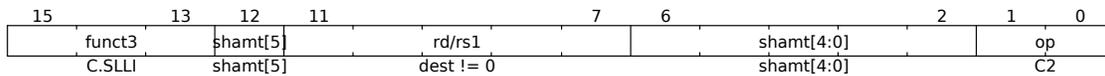


Figure 68: Integer Register-Immediate Operations (cont.)

| Instruction | Description |
|-------------|---|
| C.SLLI | Performs a logical left shift of the value in register <i>rd</i> then writes the result to <i>rd</i> . The shift amount is encoded in the <i>shamt</i> field. |

Table 60: Integer Register-Immediate Operation Description (cont.)

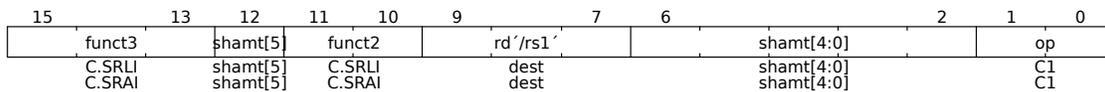


Figure 69: Integer Register-Immediate Operations (cont.)

| Instruction | Description |
|-------------|--|
| C.SRLI | Logical right shift of the value in register <i>rd'</i> then writes the result to <i>rd'</i> . The shift amount is encoded in the <i>shamt</i> field. |
| C.SRAI | Arithmetic right shift of the value in register <i>rd'</i> then writes the result to <i>rd'</i> . The shift amount is encoded in the <i>shamt</i> field. |

Table 61: Integer Register-Immediate Operation Description (cont.)

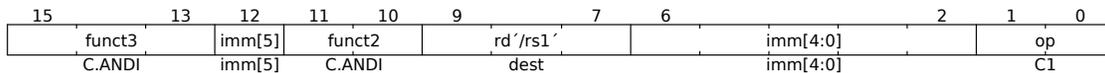


Figure 70: Integer Register-Immediate Operations (cont.)

| Instruction | Description |
|-------------|---|
| C.ANDI | Computes the bitwise AND of the value in register rd' and the sign-extended 6-bit immediate, then writes the result to rd'. |

Table 62: Integer Register-Immediate Operation Description (cont.)

Integer Register-Register Operations

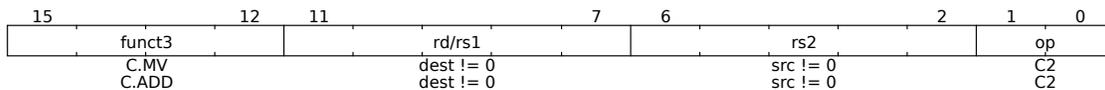


Figure 71: Integer Register-Register Operations

| Instruction | Description |
|-------------|---|
| C.MV | Copies the value in register rs2 into register rd. |
| C.ADD | Adds the values in registers rd and rs2 and writes the result to register rd. |

Table 63: Integer Register-Register Operation Description

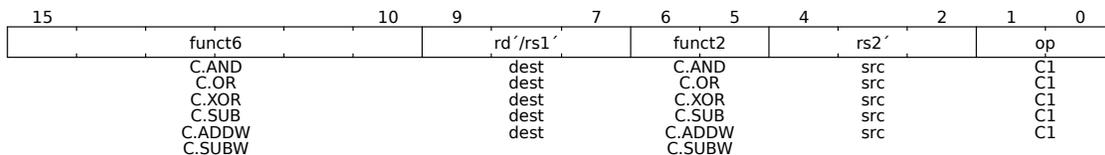


Figure 72: Integer Register-Register Operations (cont.)

| Instruction | Description |
|-------------|---|
| C.AND | Computes the bitwise AND of the values in registers rd' and rs2'. |
| C.OR | Computes the bitwise OR of the values in registers rd' and rs2'. |
| C.XOR | Computes the bitwise XOR of the values in registers rd' and rs2'. |
| C.SUB | Subtracts the value in register rs2' from the value in register rd'. |
| C.ADDW | RV64C/RV128C-only instruction that adds the values in registers rd' and rs2', then sign-extends the lower 32 bits of the sum before writing the result to register rd. |
| C.SUBW | RV64C/RV128C-only instruction that subtracts the value in register rs2' from the value in register rd', then sign-extends the lower 32 bits of the difference before writing the result to register rd. |

Table 64: Integer Register-Register Operation Description (cont.)

Defined Illegal Instruction

A 16-bit instruction with all bits zero is permanently reserved as an illegal instruction.

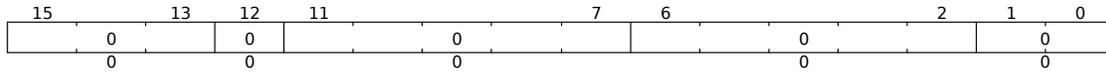


Figure 73: Defined Illegal Instruction

5.8 Zba Extension: Address Calculation Instructions

The Zba instructions are used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and 64-sized indices; that is, a shifted index is added to a base address.

5.8.1 Address Calculation Instructions

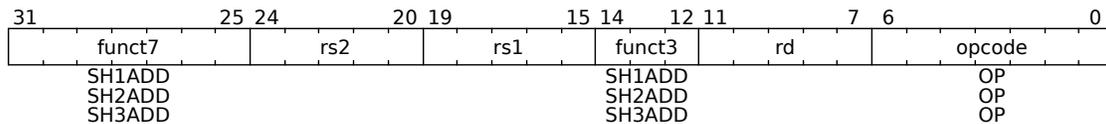


Figure 74: Address Calculation Instructions

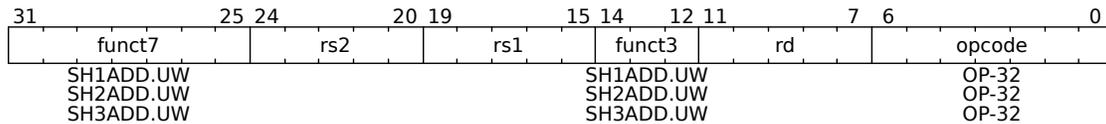


Figure 75: Address Calculation Instructions (cont.)

| Instruction | Description |
|------------------------|---|
| SH1ADD rd, rs1, rs2 | Shifts rs1 by 1 bit, then adds the result to rs2 |
| SH2ADD rd, rs1, rs2 | Shifts rs1 by 2 bits, then adds the result to rs2 |
| SH3ADD rd, rs1, rs2 | Shifts rs1 by 3 bits, then adds the result to rs2 |
| SH1ADD.UW rd, rs1, rs2 | Performs an 64-wide addition of rs2, and the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 1 bit |
| SH2ADD.UW rd, rs1, rs2 | Performs an 64-wide addition of rs2, and the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 2 bits |
| SH3ADD.UW rd, rs1, rs2 | Performs an 64-wide addition of rs2, and the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 3 bits |

Table 65: Address Calculation Instructions Description

5.8.2 Add/Shift with Prefix Zero-Extend Instructions

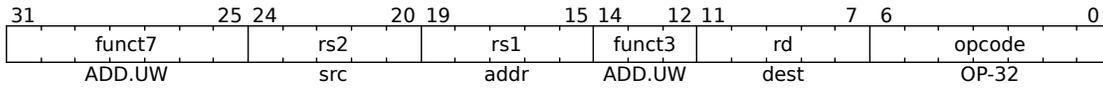


Figure 76: Add with Prefix Zero-Extend Instruction

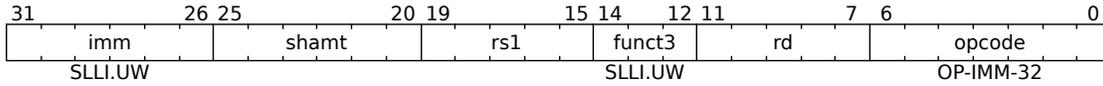


Figure 77: Shift with Prefix Zero-Extend Instruction

| Instruction | Description |
|------------------------|---|
| ADD.UW rd, rs1, rs2 | Performs a 64-wide addition between rs2 and the zero-extended least-significant word of rs1 |
| SLLI.UW rd, rs1, shamt | Takes the least-significant word of rs1, zero-extends it, and shifts it left by the immediate |

Table 66: Add/Shift with Prefix Zero-Extend Instructions Description

5.9 Zbb Extension: Basic Bit Manipulation Instructions

The Zbb instructions are used for basic bit manipulation.

5.9.1 Count Leading/Trailing Zeroes Instructions

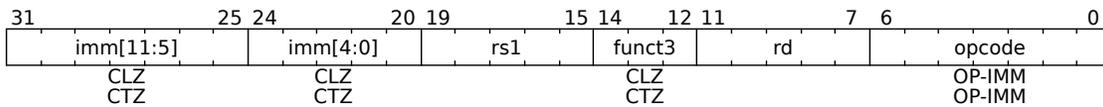


Figure 78: Count Leading/Trailing Zeroes Instructions

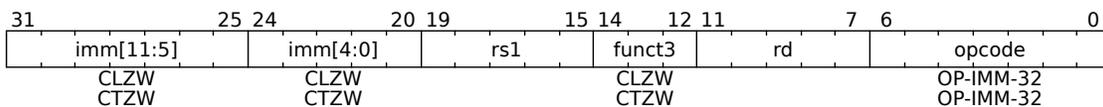


Figure 79: Count Leading/Trailing Zeroes Instructions (cont.)

| Instruction | Description |
|-------------|--|
| CLZ rd, rs | Counts the number of 0 bits before the first 1 bit, starting at the most-significant bit and progressing to bit 0. If the input is 0, the output is 64. If the most-significant bit of the input is 1, the output is 0. |
| CTZ rd, rs | Counts the number of 0 bits before the first 1 bit, starting at the least-significant bit and progressing to the most-significant bit. If the input is 0, the output is 64. If the least-significant bit of the input is 1, the output is 0. |
| CLZW rd, rs | Counts the number of 0 bits before the first 1 bit, starting at bit 31 and progressing to bit 0. If the least-significant word is 0, the output is 32. If the most-significant bit of the word is 1, the output is 0. |
| CTZW rd, rs | Counts the number of 0 bits before the first 1 bit, starting at the least-significant bit and progressing to the most-significant word. If the least-significant word is 0, the output is 32. If the least-significant bit of the input is 1, the output is 0. |

Table 67: Count Leading/Trailing Zeroes Instructions Description

5.9.2 Count Population Instructions

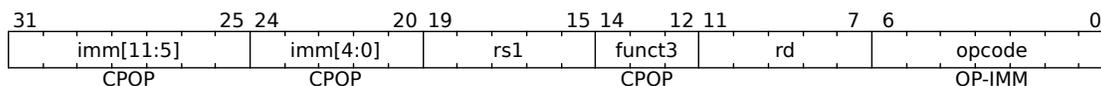


Figure 80: Count Population Instruction

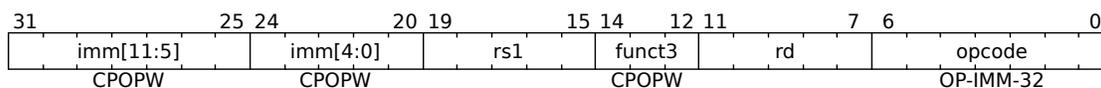


Figure 81: Count Population Instruction (cont.)

| Instruction | Description |
|--------------|--|
| CPOP rd, rs | Counts the number of 1 bits in the source register |
| CPOPW rd, rs | Counts the number of 1 bits in the least-significant word of the source register |

Table 68: Count Population Instructions Description

5.9.3 Logic-With-Negate Instructions

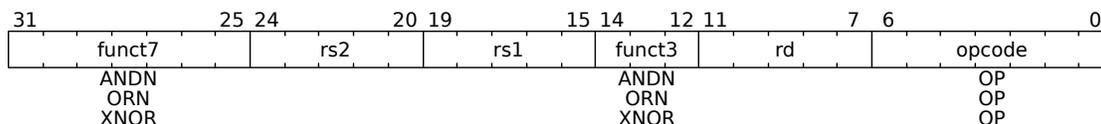


Figure 82: Logic-With-Negate Instructions

| Instruction | Description |
|-------------------|---|
| ANDN rd, rs1, rs2 | Performs bitwise logical AND between rs1 and the bitwise inversion of rs2 |
| ORN rd, rs1, rs2 | Performs bitwise logical OR between rs1 and the bitwise inversion of rs2 |
| XNOR rd, rs1, rs2 | Performs bitwise exclusive-NOR on rs1 and rs2 |

Table 69: Logic-With-Negate Instructions Description

5.9.4 Comparison Instructions

These instructions are arithmetic R-type instructions that return the smaller or larger value of two operands.

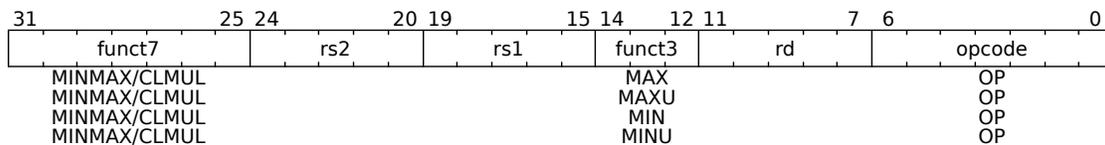


Figure 83: Comparison Instructions

| Instruction | Description |
|-------------------|--|
| MIN rd, rs1, rs2 | Returns the smaller of two signed integers |
| MINU rd, rs1, rs2 | Returns the smaller of two unsigned integers |
| MAX rd, rs1, rs2 | Returns the larger of two signed integers |
| MAXU rd, rs1, rs2 | Returns the larger of two unsigned integers |

Table 70: Comparison Instructions Description

5.9.5 Sign-Extend and Zero-Extend Instructions

These instructions perform the sign-extension or zero-extension of the least-significant 8 or 16 bits of the source register.

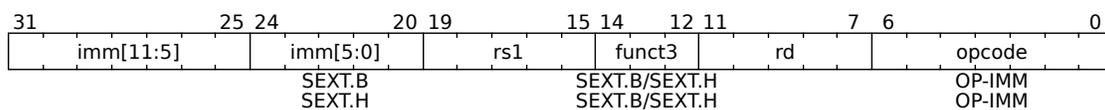


Figure 84: Sign-Extend Instructions

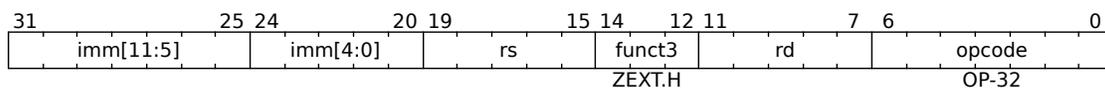


Figure 85: Zero-Extend Instruction

| Instruction | Description |
|---------------|---|
| SEXT.B rd, rs | Sign-extends the least-significant byte in the source to 64 by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits |
| SEXT.H rd, rs | Sign-extends the least-significant halfword in <i>rs</i> to 64 by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits |
| ZEXT.H rd, rs | Zero-extends the least-significant halfword of the source to 64 by inserting 0's into all of the bits more significant than 15 |

Table 71: Sign- and Zero-Extend Instructions

5.9.6 Bitwise Rotation Instructions

Bitwise rotation instructions are similar to the shift-logical operations from the base ISA specification. However, where the shift-logical instructions shift in zeros, the rotate instructions shift in the bits that were shifted out of the other side of the values.

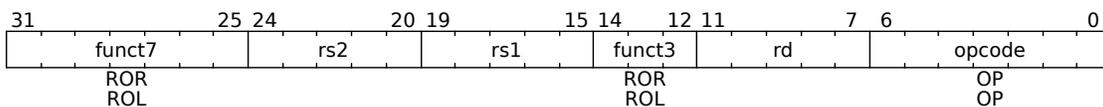


Figure 86: Bitwise Rotation Instructions

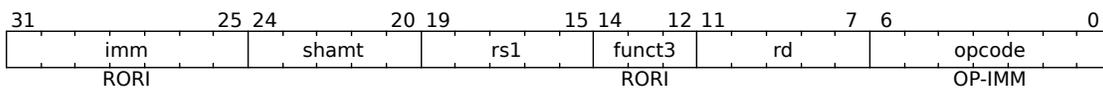


Figure 87: Bitwise Rotation Instructions (cont.)

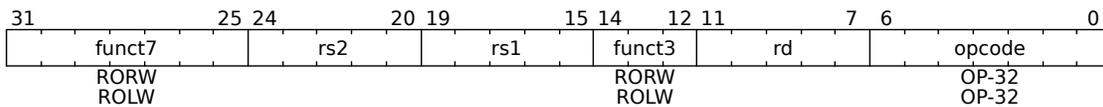


Figure 88: Bitwise Rotation Instructions (cont.)

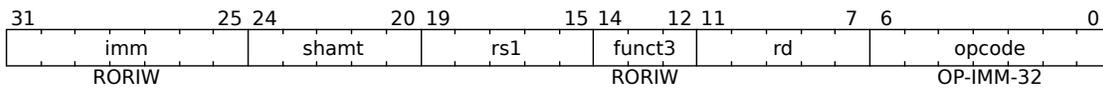


Figure 89: Bitwise Rotation Instructions (cont.)

| Instruction | Description |
|---------------------|---|
| ROR rd, rs1, rs2 | Performs a rotate right shift of rs1 by the amount in the least-significant 6 bits of rs2 |
| ROL rd, rs1, rs2 | Performs a rotate left shift of rs1 by the amount in the least-significant 6 bits of rs2 |
| RORI rd, rs1, shamt | Performs a rotate right shift of rs1 by the amount in the least significant 6 bits of shamt |
| RORW rd, rs1, rs2 | Performs a rotate right shift of the least-significant word of rs1 by the amount in the least-significant 5 bits of rs2. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits. |
| ROLW rd, rs1, rs2 | Performs a rotate left shift on the least-significant word of rs1 by the amount in the least-significant 5 bits of rs2. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits. |
| RORIW rd, rs1, imm | Performs a rotate right shift on the least-significant word of rs1 by the amount in the least-significant 6 bits of shamt. The resulting word value is sign-extended by copying bit 31 to all of the more significant bits. |

Table 72: Bitwise Rotation Instructions Description

5.9.7 OR Combine Instruction

| Instruction | Description |
|--------------|--|
| ORC.B rd, rs | Combines the bits within every byte through a reciprocal bitwise logical OR. This sets the bits of each byte in the result rd to all zeros if no bit within the respective byte of rs is set, otherwise it sets the bits to all ones if any bit within the respective byte of rs is set. |

Table 73: OR Combine Instruction Description

5.9.8 Byte-Reverse Instruction

| Instruction | Description |
|-------------|---|
| REV8 rd, rs | Reverses the order of the bytes in a register |

Table 74: Byte-Reverse Instruction Description

5.10 Zicsr Extension: Control and Status Register Instructions

RISC-V defines a separate address space of 4096 Control and Status registers associated with each hart. The defined instructions access counter, timers and floating-point status registers.

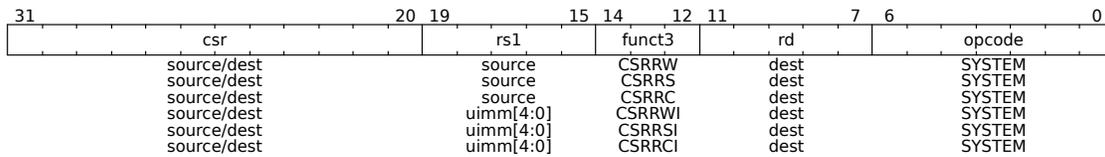


Figure 90: Zicsr Instructions

| Instruction | Description |
|--------------------|--|
| CSRRW rd, rs1 csr | Instruction atomically swaps values in the CSRs and integer registers. |
| CSRRS rd, rs1 csr | Instruction reads the value of the CSR, zero-extends the value to 64-bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. |
| CSRRC rd, rs1 csr | Instruction reads the value of the CSR, zero-extends the value to 64-bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. |
| CSRRWI rd, rs1 csr | Update the CSR using an 64-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. |
| CSRRSI rd, rs1 csr | Update the CSR using an 64-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. |
| CSRRCI rd, rs1 csr | If the uimm[4:0] field is zero, then these instructions will not write to the CSR. |

Table 75: Control and Status Register Instruction Description

The CSRRWI, CSRRSI, and CSRRCI instructions are similar in kind to CSRRW, CSRRS, and CSRRC respectively, except in that they update the CSR using an 64-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field instead of a value from an integer register. For CSRRSI and CSRRCI, these instructions will not write to the CSR if the uimm[4:0] field is zero, and they shall not cause any of the size effects that might otherwise occur on a CSR write. For CSRRWI, if rd = x0, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read. Both CSRRSI and CSRRCI will always read the CSR and cause any read side effects regardless of the rd and rs1 fields.

Table 76 shows if a CSR reads or writes given a particular CSR.

| Register Operand | | | | |
|-------------------|-----|------|-----------|------------|
| Instruction | rd | rs1 | read CSR? | write CSR? |
| CSRRW | x0 | - | no | yes |
| CSRRW | !x0 | - | yes | yes |
| CSRRS/C | - | x0 | yes | no |
| CSRRS/C | - | !x0 | yes | yes |
| Immediate Operand | | | | |
| Instruction | rd | uimm | read CSR? | write CSR? |
| CSRRWI | x0 | - | no | yes |
| CSRRWI | !x0 | - | yes | yes |
| CSRRS/CI | - | 0 | yes | no |
| CSRRS/CI | - | !0 | yes | yes |

Table 76: CSR Reads and Writes

5.10.1 Control and Status Registers

The control and status registers (CSRs) are only accessible using variations of the CSRR (Read) and CSRRW (Write) instructions. Only the CPU executing the csr instruction can read or write these registers, and they are not visible by software outside of the core they reside on. The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs. Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction. A read/write register might also contain some bits that are read-only, in which case, writes to the read-only bits are ignored. Each core functionality has its own control and status registers which are described in the corresponding section.

5.10.2 Defined CSRs

The following tables describe the currently defined CSRs, categorized by privilege level. The usage of the CSRs below is implementation specific. CSRs are only accessible when operating within a specific access mode (user mode, debug mode, supervisor mode, or machine mode). Therefore, attempts to access a non-existent CSR raise an illegal instruction exception, and attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.

| Number | Privilege | Name | Description |
|---------------------------------|-----------|--------------|--|
| User Trap Setup | | | |
| 0x000 | RW | ustatus | User status register. |
| 0x004 | RW | uie | User interrupt-enable register. |
| 0x005 | RW | utvec | User trap handler base address. |
| User Trap Handling | | | |
| 0x040 | RW | uscratch | Scratch register for use trap handlers. |
| 0x041 | RW | uepc | User exception program counter. |
| 0x042 | RW | ucause | User trap cause. |
| 0x043 | RW | ubadaddr | User bad address. |
| 0x044 | RW | uip | User interrupt pending. |
| User Floating-Point CSRs | | | |
| 0x001 | RW | fflags | Floating-Point Accrued Exceptions. |
| 0x002 | RW | frm | Floating-Point Dynamic Rounding Mode. |
| 0x003 | RW | fcsr | Floating-Point Control and Status Register (frm + fflags). |
| User Counter/Timers | | | |
| 0xC00 | RO | cycle | Cycle counter for RDCYCLE instruction. |
| 0xC01 | RO | time | Timer for RDTIME instruction. |
| 0xC02 | RO | instret | Instructions-retired counter for RDINSTRET instruction. |
| 0xC03 | RO | hpmcounter3 | Performance-monitoring counter. |
| 0xC04 | RO | hpmcounter4 | Performance-monitoring counter. |
| | | ... | |
| 0xC1F | RO | hpmcounter31 | Performance-monitoring counter. |

Table 77: User Mode CSRs

| Number | Privilege | Name | Description |
|--|-----------|---------------|--|
| Machine Information Registers | | | |
| 0xF11 | RO | mvendorid | Vendor ID. |
| 0xF12 | RO | marchid | Architecture ID. |
| 0xF13 | RO | mimpid | Implementation ID. |
| 0xF14 | RO | mhartid | Hardware thread ID. |
| Machine Trap Setup | | | |
| 0x300 | RW | mstatus | Machine status register. |
| 0x301 | RW | misa | ISA and extensions. |
| 0x302 | RW | medeleg | Machine exception delegation register. |
| 0x303 | RW | mideleg | Machine interrupt delegation register. |
| 0x304 | RW | mie | Machine interrupt-enable register. |
| 0x305 | RW | mtvec | Machine trap-handler base address. |
| 0x306 | RW | mcounteren | Machine counter enable. |
| Machine Trap Handling | | | |
| 0x340 | RW | mscratch | Scratch register for machine trap handlers. |
| 0x341 | RW | mepc | Machine exception program counter. |
| 0x342 | RW | mcause | Machine trap cause. |
| 0x343 | RW | mtval | Machine bad address or instruction. |
| 0x344 | RW | mip | Machine interrupt pending. |
| Machine Memory Protection | | | |
| 0x3A0 | RW | pmpcfg0 | Physical memory protection configuration. |
| 0x3A1 | RW | pmpcfg1 | Physical memory protection configuration, RV32 only. |
| 0x3A2 | RW | pmpcfg2 | Physical memory protection configuration. |
| 0x3A3 | RW | pmpcfg3 | Physical memory protection configuration, RV32 only. |
| 0x3B0 | RW | pmpaddr0 | Physical memory protection address register. |
| 0x3B1 | RW | pmpaddr1 | Physical memory protection address register. |
| | | ... | |
| 0x3BF | RW | pmpaddr15 | Physical memory protection address register. |
| Machine Counter/Timers | | | |
| 0xB00 | RW | mcycle | Machine cycle counter. |
| 0xB02 | RW | minstret | Machine instruction-retired counter. |
| Machine Counter Setup | | | |
| 0x320 | RW | mcountinhibit | Machine counter-inhibit register. |
| 0x323 | RW | mhpmevent3 | Machine performance-monitoring event selector. |
| 0x324 | RW | mhpmevent4 | Machine performance-monitoring event selector. |
| | | ... | |
| 0x33F | RW | mhpmevent31 | Machine performance-monitoring event selector. |
| Debug/Trace Register (shared with Debug Mode) | | | |
| 0x7A0 | RW | tselect | Debug/Trace trigger register select. |

Table 78: Machine Mode CSRs

| Number | Privilege | Name | Description |
|--------|-----------|--------|---|
| 0x7A1 | RW | tdata1 | First Debug/Trace trigger data register. |
| 0x7A2 | RW | tdata2 | Second Debug/Trace trigger data register. |
| 0x7A3 | RW | tdata3 | Third Debug/Trace trigger data register. |

Table 78: Machine Mode CSRs

| Number | Privilege | Name | Description |
|--------|-----------|----------|------------------------------------|
| 0x7B0 | RW | dcsr | Debug control and status register. |
| 0x7B1 | RW | dpc | Debug PC. |
| 0x7B2 | RW | dscratch | Debug scratch register. |

Table 79: Debug Mode Registers

5.10.3 CSR Access Ordering

On a given hart, explicit and implicit CSR access are performed in program order with respect to those instructions whose execution behavior is affected by the state of the accessed CSR. In particular, a CSR access is performed after the execution of any prior instructions in program order whose behavior modifies or is modified by the CSR state and before the execution of any subsequent instructions in program order whose behavior modifies or is modified by the CSR state.

Furthermore, a CSR read access instruction returns the accessed CSR state before the execution of the instruction, while a CSR write access instruction updates the accessed CSR state after the execution of the instruction. Where the above program order does not hold, CSR accesses are weakly ordered, and the local hart or other harts may observe the CSR accesses in an order different from program order. In addition, CSR accesses are not ordered with respect to explicit memory accesses, unless a CSR access modifies the execution behavior of the instruction that performs the explicit memory access or unless a CSR access and an explicit memory access are ordered by either the syntactic dependencies defined by the memory model or the ordering requirements defined by the Memory-Ordering PMAs. To enforce ordering in all other cases, software should execute a FENCE instruction between the relevant accesses. For the purposes of the FENCE instruction, CSR read accesses are classified as device input (I), and CSR write accesses are classified as device output (O). For more about the FENCE instructions, see Section 5.14. For CSR accesses that cause side effects, the above ordering constraints apply to the order of the initiation of those side effects but does not necessarily apply to the order of the completion of those side effects.

5.10.4 SiFive RISC-V Implementation Version Registers

`mvendorid`

The value in `mvendorid` is 0x489, corresponding to SiFive's JEDEC number.

marchid

The value in `marchid` indicates the overall microarchitecture of the core and at SiFive we use this to distinguish between core generators. The RISC-V standard convention separates `marchid` into open-source and proprietary namespaces using the most-significant bit (MSB) of the `marchid` register; where if the MSB is clear, the `marchid` is for an open-source core, and if the MSB is set, then `marchid` is a proprietary microarchitecture. The open-source namespace is managed by the RISC-V Foundation and the proprietary namespace is managed by SiFive.

SiFive's E3 and S5 cores are based on the open-source 3/5-Series microarchitecture, which has a Foundation-allocated `marchid` of 1. Our other generators are numbered according to the core series.

| Value | Core Generator |
|--------------|--------------------------------|
| 0x8000_0007 | 6/7/P200/X200-Series Processor |

Table 80: Core Generator Encoding of `marchid`

mimpid

The value in `mimpid` holds an encoded value that uniquely identifies the version of the generator used to build this implementation. If your release version is not included in Table 81, contact your SiFive account manager for more information.

| Value | Generator Release Version |
|-------------|---|
| 0x0000_0000 | Pre-19.02 |
| 0x2019_0228 | 19.02 |
| 0x2019_0531 | 19.05 |
| 0x2019_0919 | 19.08p0p0 / 19.08.00 |
| 0x2019_1105 | 19.08p1p0 / 19.08.01.00 |
| 0x2019_1204 | 19.08p2p0 / 19.08.02.00 |
| 0x2020_0423 | 19.08p3p0 / 19.08.03.00 |
| 0x0120_0626 | 19.08p4p0 / 19.08.04.00 |
| 0x0220_0515 | koala.00.00-preview and koala.01.00-preview |
| 0x0220_0603 | koala.02.00-preview |
| 0x0220_0630 | 20G1.03.00 / koala.03.00-general |
| 0x0220_0710 | 20G1.04.00 / koala.04.00-general |
| 0x0220_0826 | 20G1.05.00 / koala.05.00-general |
| 0x0320_0908 | kiwi.00.00-preview |
| 0x0220_1013 | 20G1.06.00 / koala.06.00-general |
| 0x0220_1120 | 20G1.07.00 / koala.07.00-general |
| 0x0421_0205 | llama.00.00-preview |
| 0x0421_0324 | 21G1.01.00 / llama.01.00-general |
| 0x0421_0427 | 21G1.02.00 / llama.02.00-general |
| 0x0521_0528 | mongoose.00.00-preview |
| 0x0521_0714 | 21G2.01.00 / mongoose.01.00-general |
| 0x0521_1008 | 21G2.02.00 / mongoose.02.00-general |
| 0x0621_1027 | narwhal.00.00-preview |
| 0x0621_1203 | narwhal.01.00-preview |
| 0x0621_1222 | 21G3.02.00 / narwhal.02.00-general |

Table 81: Generator Release Encoding of mimpid

Reading Implementation Version Registers

To read the mvendorid, marchid, and mimpid registers, simply replace mimpid with mvendorid or marchid as needed.

In C:

```
uintptr_t mimpid;
__asm__ volatile("csrr %0, mimpid" : "=r"(mimpid));
```

In Assembly:

```
csrr a5, mimpid
```

5.10.5 Custom CSRs

SiFive implements some custom CSRs that are specific to the implementation. For these CSRs, including the Feature Disable CSR, consider Chapter 6.

5.11 Base Counters and Timers

RISC-V ISAs provide a set of up to 32×64-bit performance counters and timers that are accessible via unprivileged 64-bit read-only CSR registers 0xC00–0xC1F. The first three of these (CYCLE, TIME, and INSTRET) have dedicated functions; while the remaining counters, if implemented, provide programmable event counting.

The S76 Core Complex implements `mcycle`, `mtime`, and `minstret` counters, which have dedicated functions: cycle count, real-time clock, and instructions-retired, respectively. The timer functionality is based on the `mtime` register. Additionally, the S76 Core Complex implements event counters in the form of `mhpmcounter`, which is used to monitor user requested events.

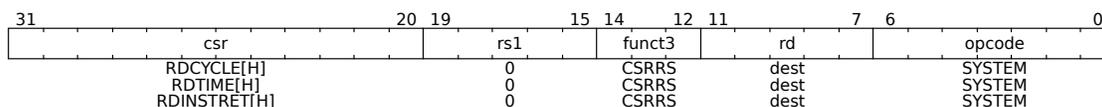


Figure 91: Timer and Counter Pseudoinstructions

| Instruction | Description |
|--------------|--|
| RDCYCLE rd | Reads the 64-bits of the cycle CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. |
| RDTIME rd | Generates an illegal instruction exception. The <code>mtime</code> register is memory mapped to the CLINT register space and can be read using a regular load instruction. |
| RDINSTRET rd | Reads the 64-bits of the instret CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past. |

Table 82: Timer and Counter Pseudoinstruction Description

RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions read the full 64 bits of the `cycle`, `time`, and `instret` counters. The RDCYCLE pseudoinstruction reads the low 64-bits of the cycle CSR (`mcycle`), which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. The RDTIME pseudoinstruction reads the low 64-bits of the time CSR (`mtime`), which counts wall-clock real time that has passed from an arbitrary start time in the past. The RDINSTRET pseudoinstruction reads the low 64-bits of the instret CSR (`minstret`), which counts the number of instructions retired by this hart from some arbitrary start point in the past. The rate at which the cycle counter advances is `rtc_clock`. To determine the current rate (cycles per second) of instruction execution, call the `metal_timer_get_timebase_frequency` API. The

`metal_timer_get_timebase_frequency` and additional APIs are described in Section 5.11.2 below.

| Number | Privilege | Name | Description |
|--------|-----------|---------|---|
| 0xC00 | RO | cycle | Cycle counter for RDCYCLE instruction |
| 0xC01 | RO | time | Timer for RDTIME instruction |
| 0xC02 | RO | instret | Instruction-retired counter for RDINSTRET instruction |

Table 83: Timer and Counter CSRs

5.11.1 Timer Register

`mtime` is a 64-bit read-write register that contains the number of cycles counted from the `rtc_toggle` signal described in the S76 Core Complex User Guide. On reset, `mtime` is cleared to zero.

5.11.2 Timer API

The APIs below are used for reading and manipulating the machine timer. Other APIs are described in more detail within the Freedom Metal documentation. <https://sifive.github.io/freedom-metal-docs/>

Functions

`int metal_timer_get_cyclecount(int hartid, unsigned long long *cyclecount)`

Read the machine cycle count.

Return

0 upon success

Parameters

- `hartid`: The hart ID to read the cycle count of
- `cyclecount`: The variable to hold the value

`int metal_timer_get_timebase_frequency(int hartid, unsigned long long *timebase)`

Get the machine timebase frequency.

Return

0 upon success

Parameters

- `hartid`: The hart ID to read the cycle count of
- `timebase`: The variable to hold the value

`int metal_timer_set_tick(int hartid, int second)`

Set the machine timer tick interval in seconds.

Return

0 upon success

Parameters

- `hartid`: The hart ID to read the cycle count of
- `second`: The number of seconds to set the tick interval to

5.12 Privileged Instructions

The RISC-V architecture implements privileged instructions that can only be executed when the S76 Core Complex is operating in a privileged mode. The SYSTEM major opcode is used to encode all of the privileged instructions.

5.12.1 Machine-Mode Privileged Instructions

Environment Call and Breakpoint

These ECALL and EBREAK instructions cause a precise requested trap to the supporting execution environment. The ECALL instruction is used to make a service request to the execution environment. The EBREAK instruction is used to return control to a debugging environment.

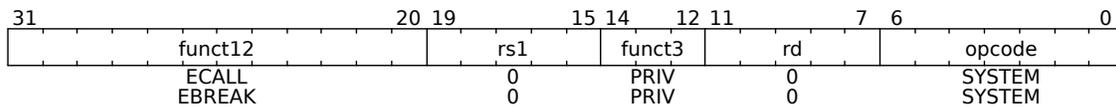


Figure 92: ECALL and EBREAK Instructions

Trap-Return Instructions

To return after handling a trap, there are separate trap return instructions per privilege level: MRET and SRET. MRET is always provided, while SRET is provided if the respective privilege mode is supported. An xRET instruction can be executed in privilege mode x or higher, where executing a lower-privilege xRET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack.

In addition to manipulating the privilege stack, xRET instructions sets the pc to the value stored in the corresponding xepc register.

Wait for Interrupt

The Wait for Interrupt (WFI) instruction provides a hint to the S76 Core Complex that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart.

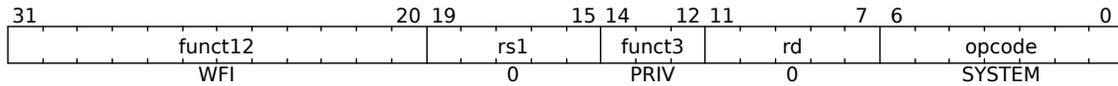


Figure 93: Wait for Interrupt Instruction

If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt exception will be taken on the following instruction, i.e., execution resumes in the trap handler and $mepc = pc + 4$. The WFI instruction can also be executed when interrupts are disabled. The operation of WFI must be unaffected by the global interrupt bits in `mstatus` (MIE/SIE) (i.e., the hart must resume if a locally enabled interrupt becomes pending), but should honor the individual interrupt enables (e.g., MTIE). WFI is also required to resume execution for locally enabled interrupts pending at any privilege level, regardless of the global interrupt enable at each privilege level. If the event that causes the hart to resume execution does not cause an interrupt to be taken, execution will resume at $pc + 4$, and software must determine what action to take, including looping back to repeat the WFI if there was no actionable event.

The suggested way to call WFI is inside an infinite loop as described below.

```
while (1) {
    __asm__ volatile ("wfi");
}
```

In SiFive's implementation of WFI, the WFI instruction is issued and the core goes into an internal clock gating state.

5.13 ABI - Register File Usage and Calling Conventions

`RV64GC_Zba_Zbb_Sscopmf` has 32 x registers that are each 64 bits wide.

| Register | ABI Name | Description | Saver |
|--------------------------|----------|-------------------------------------|--------|
| x0 | zero | Hard-wired zero | - |
| x1 | ra | Return address | Caller |
| x2 | sp | Stack pointer | Callee |
| x3 | gp | Global pointer | - |
| x4 | tp | Thread pointer | - |
| x5 | t0 | Temporary / alternate link register | Caller |
| x6-7 | t1-2 | Temporaries | Caller |
| x8 | s0/fp | Saved-register / frame-pointer | Callee |
| x9 | s1 | Saved register | Callee |
| x10-11 | a0-1 | Function arguments / return values | Caller |
| x12-17 | a2-7 | Function arguments | Caller |
| x18-27 | s2-11 | Saved registers | Callee |
| x28-31 | t3-6 | Temporaries | Caller |
| Floating-Point Registers | | | |
| f0-7 | ft0-7 | FP temporaries | Caller |
| f8-9 | fs0-1 | FP saved registers | Callee |
| f10-11 | fa0-1 | FP arguments / return values | Caller |
| f12-17 | fa2-7 | FP arguments | Caller |
| f18-27 | fa2-11 | FP saved registers | Callee |
| f28-31 | ft8-11 | FP temporaries | Caller |

Table 84: RISC-V Registers

The programmer counter PC hold the address of the current instruction.

- x1 / ra - holds the return address for a call.
- x2 / sp - stack pointer, points to the current routine stack.
- x8 / fp / s0 - frame pointer, points to the bottom of the top stack frame.
- x3 / gp - global pointer, points into the middle of the global data section.
 The common definition is: .data + 0x800. RISC-V immediate values are 12-bit signed values, which is +/- 2048 in decimal or +/- 0x800 in hex. So that global pointer relative accesses can reach their full extent, the global pointer point + 0x800 into the data section. The linker can then relax LUI+LW, LUI+SW into gp-relative LW or SW, i.e., shorter instruction sequences and access most global data using LW at gp +/- offset

```
LW t0 , 0x800(gp)
LW t1 , 0x7FF(gp)
```

- x4 / tp - thread pointer, point to thread-local storage (TLS-mostly used in Linux and RTOS).
 If you create a variable in TLS, every thread has its own copy of the variable, i.e., changes to the variable are local to the thread. This is a static area of memory that gets copied for each thread in a program. It is also used to create libraries that have thread-safe functions,

because of the fact that each call to a function has its copy of the same global data, so it's safe.

5.13.1 RISC-V Assembly

RISC-V instructions have opcodes and operands.

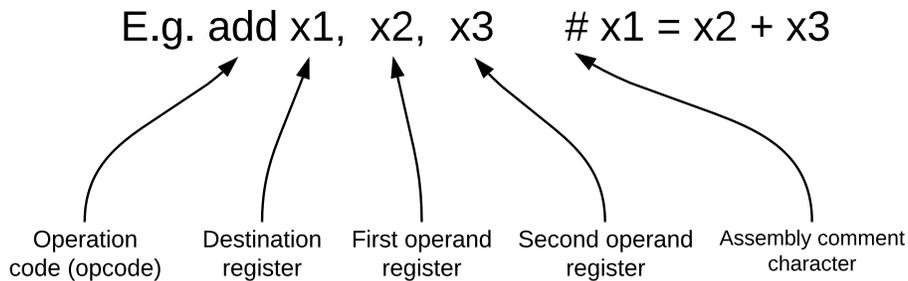


Figure 94: RISC-V Assembly Example

| Assembly | C | Description |
|---|---|---|
| add x1, x2, x3 | a = b + c | a=x1, b=x2, c=x3 |
| sub x3, x4, x5 | d = e - f | d=x3, e=x4, f=x5 |
| add x0, x0, x0 | NOP | Writes to x0 are always ignored |
| add x3, x4, x0 | f = g | f=x3, g=x4 |
| addi x3, x4, -10 | f = g - 10 | f=x3, g=x4 |
| lw x10, 12(x13) # 12 = 3x4 add x11, x12, x10 | int A[100]; g = h + A[3]; | Reg x10 gets A[3] g=x11, h=x12 |
| lw x10, 12(x13) # 12 = 3x4 add x10, x12, x10 sw x10, 40(x13) # 40 = 10x4 | int A[100]; A[10] = h + A[3]; | Reg x10 gets A[3] h=x12 Reg x10 gets h + A[3] |
| bne x13, x14, done add x10, x11, x12 done: | if (i == j) f = g + h; | f=x10, g=x11, h=x12, i=x13, j=x14 |
| bne x10, x14, else add x10, x11, x12 j done else: sub x10, x11, x12 done: | if (i == j) f = g + h; else f = g - h; | f=x10, g=x11, h=x12, i=x13, j=x14 |

Table 85: RISC-V Assembly and C Examples

5.13.2 Assembler to Machine Code

The following flowchart describes how the assembler converts the RISC-V assembly code to machine code.

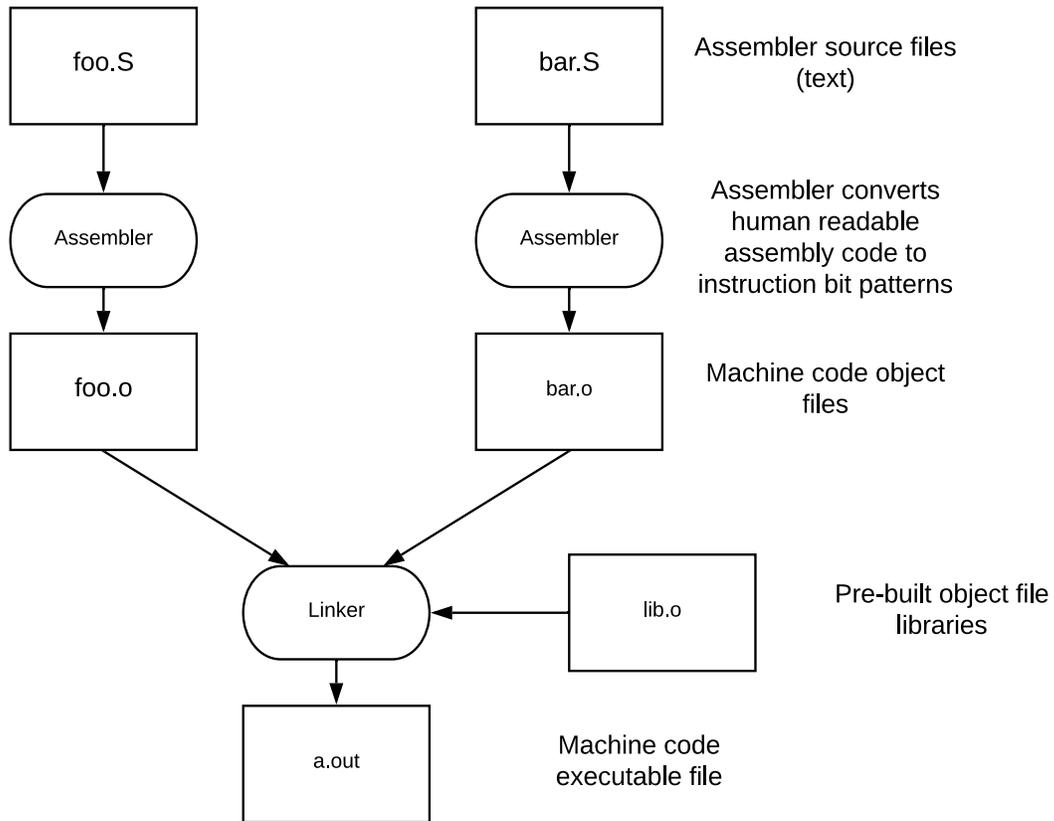


Figure 95: RISC-V Assembly to Machine Code

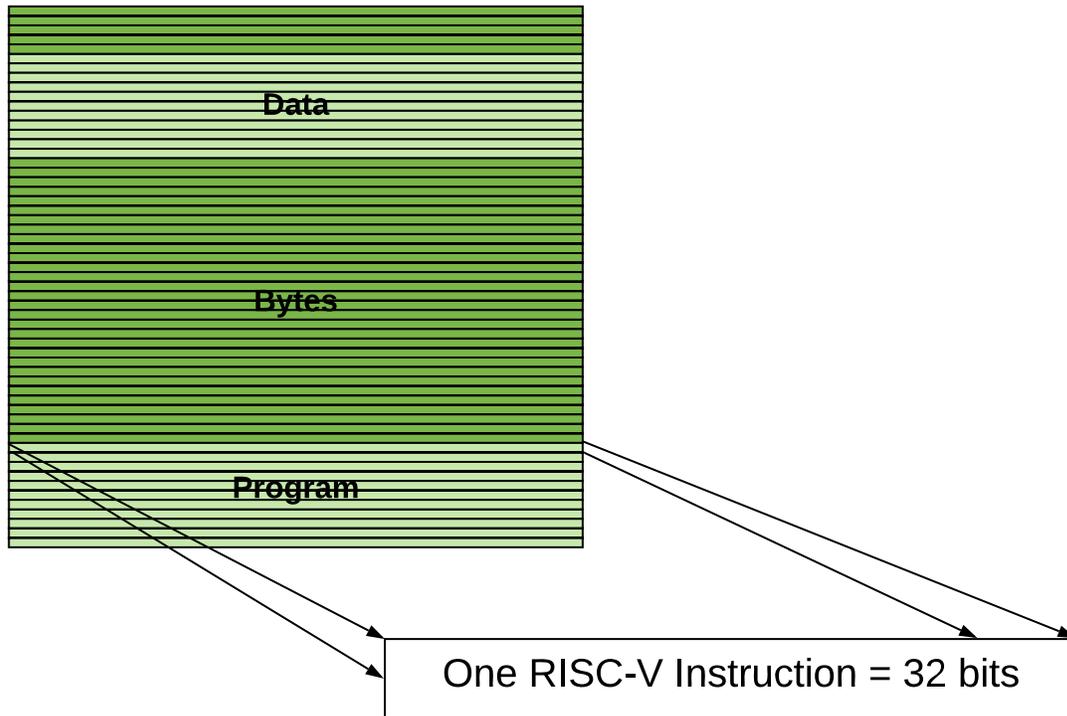


Figure 96: One RISC-V Instruction

5.13.3 Calling a Function (Calling Convention)

1. Put parameters in place where function can access them.
2. Transfer control to function.
3. Acquire local resources needed for function.
4. Perform function task.
5. Place result values where calling code can access and restore any registers might have used.
6. Return control to original caller.

Caller-saved The function invoked can do whatever it likes with the registers. Callee-saved If a function wants to use registers it needs to store and restore them.

Take, for example, the following function:

```
int leaf(int g, int h, int i, int j) {  
    int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

In this function above, arguments are passed in a0, a1, a2 and a3. The return value is returned in a0.

```
addi sp, sp, -8    # adjust stack for 2 items
sw s1, 4(sp)      # save s1 for use afterwards
sw s0, 0(sp)      # save s0 for use afterwards

add s0,a0,a1      # s0 = g + h
add s1,a2,a3      # s1 = i + j
sub a0,s0,s1      # return value (g + h) - (i + j)

lw s0, 0(sp)      # restore register s0 for caller
lw s1, 4(sp)      # restore register s1 for caller
addi s1, 4(sp)    # adjust stack to delete 2 items
jr ra            # jump back to calling routine
```

In the assembly above, notice that the stack pointer was decremented by 8 to make room to save the registers. Also, s1 and s0 are saved and will be stored at the end.

Nested Functions

In the case of nested function calls, values held in a0-7 and ra will be clobbered.

Take, for example, the following function:

```
int sumSquare(int x, int y) {
    return mult(x,x) + y;
}
```

In the function above, a function called sumSquare is calling mult. To execute the function, there's a value in ra that sumSquare wants to jump back to, but this value will be overwritten by the call to mult.

To avoid this, the sumSquare return address must be saved before the call to mult. To save the return address of sumSquare, the function can utilize stack memory. The user can use stack memory to preserve automatic (local) variables that don't fit within the registers.

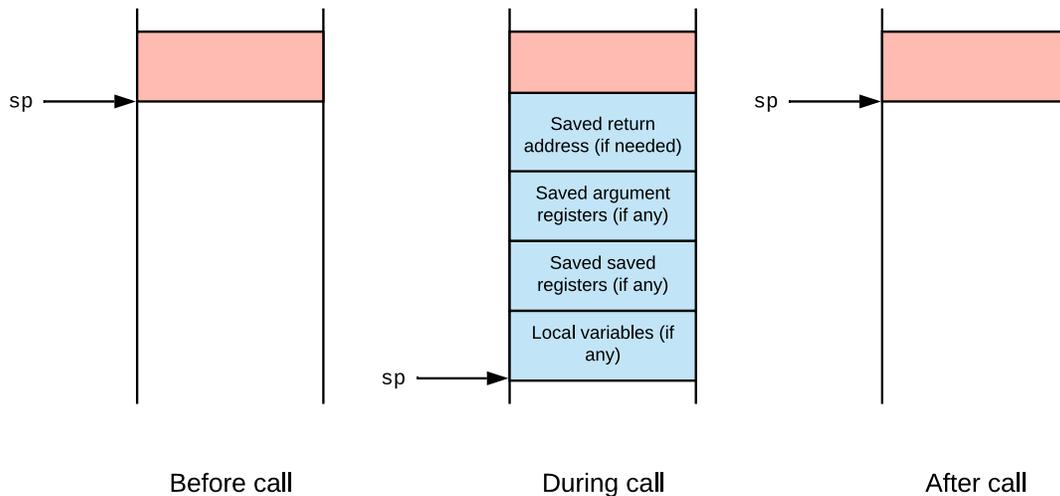


Figure 97: Stack Memory during Function Calls

Consider the assembly for sumSquare below:

```
sumSquare:
addi sp,sp,-8      # reserve space on stack
sw ra, 4(sp)      # save return address
sw a1, 0(sp)      # save y
mv a1,a0          # mult(x,x)
jal mult          # call mult
lw a1, 0(sp)      # restore y
add a0,a0,a1      # mult()+y
lw ra, 4(sp)      # get return address
addi sp,sp,8      # restore stack
mult:...
```

5.14 Memory Ordering - FENCE Instructions

In the RISC-V ISA, each thread, referred to as a hart, observes its own memory operations as if they executed sequentially in program order. RISC-V also has a relaxed memory model, which requires explicit FENCE instructions to guarantee the ordering of memory operations.

The FENCE instructions include FENCE and FENCE.I. The FENCE instruction simply ensures that the memory access instructions before the FENCE instruction get committed before the FENCE instruction is committed. It does not guarantee that those memory access instructions have actually completed. For example, a load instruction before a FENCE instruction can commit without waiting for its value to come back from the memory system. FENCE.I functions the same as FENCE, as well as flushes the instruction cache.

For example, without FENCE instructions:

Hart 1 executes:

Load X
Store Y
Store Z

Because of relaxed memory model, Hart 2 could see stores/loads arranged in any order:

Store Z
Load X
Store Y

With FENCE instructions:

Hart 1 executes:

Load X
Store Y
FENCE
Store Z

Hart 2 sees:

Store Y
Load X
Store Z

With FENCE instructions, Hart 2 is forced to see the Load X and the Store Y prior to the Store Z, but could arbitrarily see Store Y before Load X or Load X before Store Y. Functionally, FENCE instructions order the completion of older memory accesses prior to newer accesses. However, unnecessary FENCE instructions slow processes and can hide bugs, so it is essential to identify where and when FENCE should be used.

5.15 Boot Flow

This process is managed as part of the Freedom Metal source code. The freedom-metal boot code supports single core boot or multi-core boot, and contains all the necessary initialization code to enable every core in the system.

1. ENTRY POINT: File: freedom-metal/src/entry.S, label: `_enter`.
2. Write `mtvec` register with `early_trap_vector` as default exception handler.
3. Initialize global pointer `gp` register using the generated symbol `__global_pointer$`.
4. Clear feature disable CSR `0x7c1`.
5. Read `mhartid` into register `a0` and call `_start`, which exists in `crt0.S`.
6. We now transition to File: freedom-metal/gloss/crt0.S, label: `_start`.
7. Initialize stack pointer, `sp`, with `_sp` generated symbol. Harts with `mhartid` of one or larger are offset by $(_sp + _stack_size \times mhartid)$. The `__stack_size` field is generated in the linker file.

8. Check if `mhartid == __metal_boot_hart` and run the init code if they are equal. All other harts skip init and go to the Post-Init Flow, step #15.
9. Boot Hart Init Flow begins here.
10. Init data section to destination in defined RAM space.
11. Copy ITIM section, if ITIM code exists, to destination.
12. Zero out bss section.
13. Call `atexit` library function that registers the `libc` and `freedom-metal` destructors to run after `main` returns.
14. Call the `__libc_init_array` library function, which runs all functions marked with `__attribute__((constructor))`.
 - a. For example, PLL, UART, hardware prefetcher, L2, L2 prefetcher, and/or L3, if they exist in the design. This method provides full early initialization prior to entering the main application. Prefetchers are initialized to a known value at startup, but not tuned for any specific workload. Refer to the prefetcher software example that demonstrates this tuning.
15. Post-Init Flow Begins Here.
16. Call the C routine `__metal_synchronize_harts`, where hart 0 will release all harts once their individual `msip` bits are set. The `msip` bit is typically used to assert a software interrupt on individual harts, however interrupts are not yet enabled, so `msip` in this case is used as a gatekeeping mechanism.
17. Check `misa` register to see if floating-point hardware is part of the design, and set up `mstatus` accordingly.
18. Single or multi-hart design redirection step.
 - a. If design is a single hart only, or a multi-hart design without a C-implemented function `secondary_main`, ONLY the boot hart will continue to `main()`.
 - b. For multi-hart designs, all other CPUs will enter sleep via WFI instruction via the weak `secondary_main` label in `crt0.S`, while boot hart runs the application program.
 - c. In a multi-hart design which includes a C-defined `secondary_main` function, all harts will enter `secondary_main` as the primary C function.

5.16 Linker File

The linker file generates important symbols that are used in the boot code. The linker file options are found in the `freedom-e-sdk/bsp` path.

There are usually three different linker file options:

- `metal.default.lds` — Use flash and RAM sections

- `metal.ramrodata.lds` — Place read only data in RAM for better performance
- `metal.scratchpad.lds` — Places all code + data sections into available RAM location

Each linker option can be selected by specifying `LINK_TARGET` on the command line.

For example:

```
make PROGRAM=hello TARGET=design-rtl CONFIGURATION=release LINK_TARGET=scratchpad
software
```

The `metal.default.lds` linker file is selected by default when `LINK_TARGET` is not specified. If there is a scenario where a custom linker is required, one of the supplied linker files can be copied and renamed and used for the build. For example, if a new linker file named `metal.newmap.lds` was generated, this can be used at build time by specifying `LINK_TARGET=newmap` on the command line.

5.16.1 Linker File Symbols

The linker file generates symbols that are used by the startup code, so that software can use these symbols to assign the stack pointer, initialize or copy certain RAM sections, and provide the boot hart information. These symbols are made visible to software using the `PROVIDE` keyword.

For example:

```
__stack_size = DEFINED(__stack_size) ? __stack_size : 0x400;
PROVIDE(__stack_size = __stack_size);
```

Generated Linker Symbols

A description list of the generated linker symbols is shown below.

`__metal_boot_hart`

This is an integer number to describe which hart runs the main init flow. The `mhartid` CSR contains the integer value for each hart. For example, hart 0 has `mhartid==0`, hart 1 has `mhartid==1`, and so on. An assembly example is shown below, where `a0` already contains the `mhartid` value.

```
/* If we're not hart 0, skip the initialization work */
la t0, __metal_boot_hart
bne a0, t0, _skip_init
```

An example on how to use this symbol in C code is shown below.

```
extern int __metal_boot_hart;
int boot_hart = (int)&__metal_boot_hart;
```

Additional linker file generated symbols, along with descriptions are shown below.

__metal_chicken_bit

Status bit to tell startup code to zero out the Feature Disable CSR. Details of this register are internal use only.

__global_pointer\$

Static value used to write the gp register at startup.

_sp

Address of the end of stack for hart 0, used to initialize the beginning of the stack since the stack grows lower in memory. On a multi-hart system, the start address of the stack for each hart is calculated using $(_sp + _stack_size \times mhartid)$

metal_segment_bss_target_start

metal_segment_bss_target_end

Used to zero out global data mapped to .bss section.

- Only `__metal_boot_hart` runs this code.

metal_segment_data_source_start

metal_segment_data_target_start

metal_segment_data_target_end

Used to copy data from image to its destination in RAM.

- Only `__metal_boot_hart` runs this code.

metal_segment_itim_source_start

metal_segment_itim_target_start

metal_segment_itim_target_end

Code or data can be placed in itim sections using the `__attribute__((section(".itim")))`.

- When this attribute is applied to code or data, the `metal_segment_itim_source_start`, `metal_segment_itim_target_start`, and `metal_segment_itim_target_end` symbols get updated accordingly, and these symbols allow the startup code to copy code and data into the ITIM area.
 - Only `__metal_boot_hart` runs this code.

Note

At the time of this writing, the boot flow does not support C++ projects

5.17 RISC-V Compiler Flags

5.17.1 arch, abi, and mtune

RISC-V targets are described using three arguments:

1. `-march=ISA`: selects the architecture to target.
2. `-mabi=ABI`: selects the ABI to target.
3. `-mtune=CODENAME`: selects the microarchitecture to target.

`-march`

This argument controls which instructions and registers are available for the compiler, as defined by the RISC-V user-level ISA specification.

The RISC-V ISA with 32, 32-bit integer registers and the instructions for multiplication would be denoted as RV32IM. Users can control the set of instructions that GCC uses when generating assembly code by passing the lower-case ISA string to the `-march` GCC argument; for example, `-march=rv32im`. On RISC-V systems that don't support particular operations, emulation routines may be used to provide the missing functionality.

Example:

```
double dmul(double a, double b) {  
    return a * b;  
}
```

will compile directly to a FP multiplication instruction when compiled with the D extension:

```
$ riscv64-unknown-elf-gcc test.c -march=rv64imafdc -mabi=lp64d -o- -S -O3  
dmul:  
    fmul.d    fa0,fa0,fa1  
    ret
```

but will compile to an emulation routine without the D extension:

```
$ riscv64-unknown-elf-gcc test.c -march=rv64i -mabi=lp64 -o- -S -O3  
dmul:  
    add      sp,sp,-16  
    sd      ra,8(sp)  
    call    __muldf3  
    ld      ra,8(sp)  
    add     sp,sp,16  
    jr      ra
```

Similar emulation routines exist for the C intrinsics that are trivially implemented by the M and F extensions.

-mabi

`-mabi` selects the ABI to target. This controls the calling convention (which arguments are passed in which registers) and the layout of data in memory. The `-mabi` argument to GCC specifies both the integer and floating-point ABIs to which the generated code complies. Much like how the `-march` argument specifies which hardware generated code can run on, the `-mabi` argument specifies which software-generated code can link against. We use the standard naming scheme for integer ABIs (`ilp32` or `lp64`), with an argumental single letter appended to select the floating-point registers used by the ABI (`ilp32` vs. `ilp32f` vs. `ilp32d`). In order for objects to be linked together, they must follow the same ABI.

RISC-V defines two integer ABIs and three floating-point ABIs.

- `ilp32`: `int`, `long`, and pointers are all 32-bits long. `long long` is a 64-bit type, `char` is 8-bit, and `short` is 16-bit.
- `lp64`: `long` and pointers are 64-bits long, while `int` is a 32-bit type. The other types remain the same as `ilp32`.

The floating-point ABIs are a RISC-V specific addition:

- `""` (the empty string): No floating-point arguments are passed in registers.
- `f`: 32-bit and smaller floating-point arguments are passed in registers. This ABI requires the F extension, as without F there are no floating-point registers.
- `d`: 64-bit and smaller floating-point arguments are passed in registers. This ABI requires the D extension.

arch/abi Combinations

- `march=rv32imafdc -mabi=ilp32d`: Hardware floating-point instructions can be generated and floating-point arguments are passed in registers. This is like the `-mfloat-abi=hard` argument for the Arm® architecture GCC.
- `march=rv32imac -mabi=ilp32`: No floating-point instructions can be generated and no floating-point arguments are passed in registers. This is like the `-mfloat-abi=soft` argument for the Arm architecture GCC.
- `march=rv32imafdc -mabi=ilp32`: Hardware floating-point instructions can be generated, but no floating-point arguments will be passed in registers. This is like the `-mfloat-abi=softfp` argument for the Arm architecture GCC, and is usually used when interfacing with soft-float binaries on a hard-float system.
- `march=rv32imac -mabi=ilp32d`: Illegal, as the ABI requires floating-point arguments are passed in registers but the ISA defines no floating-point registers to pass them in.

Example:

```
double dmul(double a, double b) {  
    return b * a;  
}
```

If neither the ABI nor ISA contains the concept of floating-point hardware then the C compiler cannot emit any floating-point-specific instructions. In this case, emulation routines are used to perform the computation and the arguments are passed in integer registers:

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imac -mabi=ilp32 -o- -S -O3  
dmul:  
    mv     a4,a2  
    mv     a5,a3  
    add    sp,sp,-16  
    mv     a2,a0  
    mv     a3,a1  
    mv     a0,a4  
    mv     a1,a5  
    sw     ra,12(sp)  
    call   __muldf3  
    lw     ra,12(sp)  
    add    sp,sp,16  
    jr     ra
```

The second case is the exact opposite of this one: everything is supported in hardware. In this case we can emit a single `fmul.d` instruction to perform the computation.

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32d -o- -S -O3  
dmul:  
    fmul.d fa0,fa1,fa0  
    ret
```

The third combination is for users who may want to generate code that can be linked with code designed for systems that don't subsume a particular extension while still taking advantage of the extra instructions present in a particular extension. This is a common problem when dealing with legacy libraries that need to be integrated into newer systems. For this purpose, the compiler arguments and multilib paths designed to cleanly integrate with this workflow. The generated code is essentially a mix between the two above outputs: the arguments are passed in the registers specified by the `ilp32` ABI (as opposed to the `ilp32d` ABI, which could pass these arguments in registers) but then once inside the function the compiler is free to use the full power of the `RV32IMAFDC` ISA to actually compute the result. While this is less efficient than the code the compiler could generate if it was allowed to take full advantage of the `D`-extension registers, it's a lot more efficient than computing the floating-point multiplication without the `D`-extension instructions

```
$ riscv64-unknown-elf-gcc test.c -march=rv32imafdc -mabi=ilp32 -o- -S -O3
    dmul:
    add    sp,sp,-16
    sw     a0,8(sp)
    sw     a1,12(sp)
    fld    fa5,8(sp)
    sw     a2,8(sp)
    sw     a3,12(sp)
    fld    fa4,8(sp)
    fmul.d fa5,fa5,fa4
    fsd    fa5,8(sp)
    lw     a0,8(sp)
    lw     a1,12(sp)
    add    sp,sp,16
    jr     ra
```

5.18 Compilation Process

GCC driver script is actually running the preprocessor, then the compiler, then the assembler and finally the linker. If the user runs GCC with the `--save-temps` argument, several intermediate files will be generated.

```
$ riscv64-unknown-linux-gnu-gcc relocation.c -o relocation -O3 --save-temps
```

- `relocation.i`: The preprocessed source, which expands any preprocessor directives (things like `#include` or `#ifdef`).
- `relocation.s`: The output of the actual compiler, which is an assembly file (a text file in the RISC-V assembly format).
- `relocation.o`: The output of the assembler, which is an un-linked object file (an ELF file, but not an executable ELF).
- `relocation`: The output of the linker, which is a linked executable (an executable ELF file).

5.19 Large Code Model Workarounds

RISC-V software currently requires that linked symbols reside within a 32-bit range. There are two types of code models defined for RISC-V, **medlow** and **medany**. The **medany** code model generates `auipc/ld` pairs to refer to global symbols, which allows the code to be linked at any address, while **medlow** generates `lui/ld` pairs to refer to global symbols, which restricts the code to be linked around address zero. They both generate 32-bit signed offsets for referring to symbols, so they both restrict the generated code to being linked within a 2 GiB window. When building software, the code model parameter is passed into the RISC-V toolsuite and it defines a method to generate the necessary instruction combinations to access global symbols within the software program. This is done using `-mcmmodel=medany/medlow`. For 32-bit architectures, we use the **medlow** code model, while **medany** is used for 64-bit architectures. This is controlled within the 'setting.mk' file in the `freedom-e-sdk/bsp` folder.

The real problem occurs when:

1. Total program size exceeds 2 GiB, which is rare
2. When global symbols within a single compiled image are required to reside in a region outside of the 32-bit space

Example for symbols within 32-bit address space:

```
MEMORY
{
ram (wxa!ri) : ORIGIN = 0x80000000, LENGTH = 0x4000
flash (rxai!w) : ORIGIN = 0x20400000, LENGTH = 0x1fc00000
}
```

Example for symbols outside 32-bit address space:

```
MEMORY
{
ram (wxa!ri) : ORIGIN = 0x100000000, LENGTH = 0x4000 /* Updated ORIGIN from
0x80000000 */
flash (rxai!w) : ORIGIN = 0x20400000, LENGTH = 0x1fc00000
}
```

If a software example uses the above memory map, and uses either medlow or medany code models, it will not link successfully. Generated errors will generally contain the following phrase:

```
relocation truncated to fit:
```

A workaround for the linker error “relocation truncated to fit:” is to use `LINK_TARGET=scratchpad` since both the code and data sections get placed into the ram section, as defined by the linker script. Note that this doesn’t always solve the problem, since some designs do not have enough memory allocated to the ram section to fit the compiled software example. To solve these cases, SiFive provides support for the compact code model.

5.19.1 RISC-V Code Model Summary

| | Medlow | Medany | Compact |
|-----------------|----------------|-------------------|----------------|
| Code | Small | Small | Small |
| Data | Small | Small | Small |
| Distance | < 4GB | < 2GB (PC to GOT) | No limitation |
| Address | 4GB (absolute) | No limitation | No limitation |

Table 86: RISC-V Code Model Table

As shown in the above table, the compact code model option has no limits on the base address, or the distance between, the code and data sections.

5.19.2 Enabling the Compact Code Model

To enable the large code model, follow the steps below:

1. Enable compact code model in settings.mk file to use `RISCV_CMODEL = compact` instead of `RISCV_CMODEL = medany`.
2. Update assembly files to use new instructions if `__riscv_cmodel_compact` is defined by the toolsuite.
3. Update linker alignment.
4. Use the latest GCC+LLVM toolsuite from SiFive, starting with 2021.06.x.

Makefile Update

To enable the toolsuite to generate the proper code sequences for the compact code model, first update settings.mk file, which can be found in the board support package (BSP) path, similar to `freedom-e-sdk/bsp/design-rtl`.

Change: `RISCV_CMODEL = medany` ⇒ `RISCV_CMODEL = compact`

The `RISCV_CMODEL` definition gets passed into the toolsuite using the `-mcmode1` switch.

Note

For 32-bit designs, which do not require the use of the compact code model, you will find `RISCV_CMODEL = medlow` in settings.mk.

This `-mcmode1=compact` option will enable the symbol `__riscv_cmodel_compact` to be visible within the code, and can be used to determine the correct code sequences to use within assembly files.

Assembly File Updates

Assembly files may need to be hand-edited to support the compact code model if they reference a global symbol. The following freedom-metal source files now support the compact code model option:

1. `freedom-metal/src/entry.S`
2. `freedom-metal/src/scrub.S`
3. `freedom-metal/gloss/crt0.S`

Linker Alignment

The global pointer alignment is required to be: `PROVIDE(__global_pointer$ = ALIGN . + 0x800), 16;`

See `metal.default.lds`, or the `*.lds` you plan to use.

5.20 Pipeline Hazards

The pipeline only interlocks on read-after-write and write-after-write hazards, so instructions may be scheduled to avoid stalls.

5.20.1 Read-After-Write Hazards

Read-after-Write (RAW) hazards occur when an instruction tries to read a register before a preceding instruction tries to write to it. This hazard describes a situation where an instruction refers to a result that has not been calculated or retrieved. This situation is possible because even though an instruction was executed after a prior instruction, the prior instruction may only have processed partly through the core pipeline.

Example:

- Instruction 1: $x1 + x3$ is saved in $x2$
- Instruction 2: $x2 + x3$ is saved in $x4$

The first instruction is calculating a value ($x1 + x3$) to be saved in $x2$. The second instruction is going to use the value of $x2$ to compute a result to be saved in $x4$. However, in the core pipeline, when operations are fetched for the second operation, the results from the first operation have not yet been saved.

5.20.2 Write-After-Write Hazards

Write-after-write (WAW) hazards occur when an instruction tries to write an operand before it is written by a preceding instruction.

Example:

- Instruction 1: $x4 + x7$ is saved in $x2$
- Instruction 2: $x1 + x3$ is saved in $x2$

Write-back of instruction 2 must be delayed until instruction 1 finishes executing.

In general, MMIO accesses stall when there is a hazard on the result caused by either RAW or WAW. So, instructions may be scheduled to avoid stalls.

5.21 Reading CSRs

There are several methods for reading the CSRs that are implemented in the S76 Core Complex. A full list of the defined RISC-V CSRs are described in Section 5.10.2.

1. Inline assembly using `csrr` instruction and the register name. For example, reading the `misalr` CSR:

```
int misa;  
__asm__ volatile("csrr %0, misa" : "=r" (misa));
```

2. Using the Freedom Metal API `METAL_CPU_GET_CSR`. Again, reading the `misa` CSR:

```
int misa_value;  
METAL_CPU_GET_CSR(misa,misa_value);
```

In the second method, the first argument is the register name and the second is the variable to store the result in.

Both inline assembly and Freedom Metal API methods can receive the CSR number instead of its name. For example:

```
int mscratch;  
METAL_CPU_GET_CSR(0x340, mscratch_value); // reading mscratch csr
```

Note

Accessing CSRs has to be according to the privilege level you are in. Attempting to access a CSR in a privilege level higher than the current level of operation will result in an exception.

To access a privileged CSR, the user must switch to the appropriate privilege level. This can be done using the following Freedom Metal API:

```
metal_privilege_drop_to_mode(METAL_PRIVILEGE_USER,  
                             my_regfile,  
                             user_mode_entry_point);
```

The Freedom Metal API routines and more examples located in `freedom-e-sdk/software` directory.

Chapter 6

Custom Instructions and CSRs

This chapter describes some of the custom instructions and CSRs configured in the S76 Core Complex.

6.1 SiFive Custom Instructions

These custom instructions use the SYSTEM instruction encoding space, which is the same as the custom CSR encoding space, but with `funct3=0`.

6.1.1 CFLUSH.D.L1

- Implemented as state machine in L1 data cache, for cores with data caches.
- Only available in M-mode. Execution from modes other than M-mode raises an illegal-instruction exception.
- Opcode `0xFC000073`, with optional `rs1` field in bits [19:15].
- When `rs1 = x0`, `CFLUSH.D.L1` writes back and invalidates all lines in the L1 data cache.
- When `rs1 ≠ x0`, `CFLUSH.D.L1` writes back and invalidates the L1 data cache line containing the virtual address in integer register `rs1`.
- If the effective privilege mode does not have write permissions to the address in `rs1`, then a store access or store page-fault exception is raised.
- If the address in `rs1` is in an uncacheable region with write permissions, the instruction has no effect but raises no exceptions.
- If the address in `rs1` is in an uncacheable region, cache line misaligned accesses may raise an access exception.
- Note that if the PMP scheme write-protects only part of a cache line, then using a value for `rs1` in the write-protected region will cause an exception, whereas using a value for `rs1` in the write-permitted region will write back the entire cache line.

6.1.2 CDISCARD.D.L1

- Implemented as state machine in L1 data cache, for cores with data caches.

- Only available in M-mode. Execution from modes other than M-mode raises an illegal-instruction exception.
- Opcode `0xFC200073`, with optional `rs1` field in bits [19:15].
- When `rs1 = x0`, `CDISCARD.D.L1` invalidates, but does not write back, all lines in the L1 data cache. Dirty data within the cache is lost.
- When `rs1 ≠ x0`, `CDISCARD.D.L1` invalidates, but does not write back, the L1 data cache line containing the virtual address in integer register `rs1`. Dirty data within the cache line is lost.
- If the effective privilege mode does not have write permissions to the address in `rs1`, then a store access or store page-fault exception is raised.
- If the address in `rs1` is in an uncacheable region with write permissions, the instruction has no effect but raises no exceptions.
- If the address in `rs1` is in an uncacheable region, cache line misaligned accesses may raise an access exception.
- Note that if the PMP scheme write-protects only part of a cache line, then using a value for `rs1` in the write-protected region will cause an exception, whereas using a value for `rs1` in the write-permitted region will invalidate and discard the entire cache line.

6.1.3 CEASE

- Privileged instruction only available in M-mode.
- Opcode `0x30500073`.
- After retiring `CEASE`, hart will not retire another instruction until reset.
- Instigates power-down sequence, which will eventually raise the `cease_from_tile_N` signal to the outside of the Core Complex, indicating that it is safe to power down.
- `CEASE` has no effect on System Bus Access.
- Debug `haltreq` will not work after a `CEASE` instruction has retired.

6.1.4 PAUSE

- Opcode `0x0100000F`, which is a `FENCE` instruction with predecessor set `W` and null successor set. Therefore, `PAUSE` is a `HINT` instruction that executes as a no-op on all RISC-V implementations.
- This instruction may be used for more efficient idling in spin-wait loops.
- This instruction causes a stall of up to 32 cycles or until a cache eviction occurs, whichever comes first.

6.2 SiFive Custom CSRs

These custom CSRs use the custom CSR encoding space. Refer to Appendix C for an exhaustive list of the custom CSRs available on the S76 Core Complex.

6.2.1 Branch Prediction Mode CSR

This SiFive custom extension adds an M-mode CSR to control the current branch prediction mode, `mbpm` at CSR `0x7C0`.

The S76 Core Complex's branch prediction system includes a Return Address Stack (RAS), a Branch Target Buffer (BTB), and a Branch History Table (BHT). While branch predictors are essential to achieve high performance in pipelined processors, they can also cause undesirable timing variability for hard real-time systems. The `mbpm` register provides a means to customize the branch predictor behavior to trade average performance for a more predictable execution time.

| Branch Prediction Mode CSR | | | | |
|----------------------------|---------------------|-------|------|-----------------------------|
| CSR | 0x7C0 | | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | <code>static</code> | RW | 0x0 | Branch-Direction Prediction |
| [7:1] | Reserved | RO | 0x0 | |

Table 87: Branch Prediction Mode CSR

The `static` bit determines the value returned by the BHT component of the branch prediction system. A zero value indicates dynamic direction prediction, and a non-zero value indicates static-taken direction prediction. The BTB is cleared on any write to `static`, and the RAS is unaffected by writes to `static`.

6.2.2 SiFive Feature Disable CSR

The SiFive custom M-mode Feature Disable CSR is provided to enable or disable certain microarchitectural features. In the S76 Core Complex, CSR `0x7C1` has been allocated for this purpose. These features are described in Table 88.

Warning

The features that can be controlled by this CSR are subject to change or removal in future releases. It is not advised to depend on this CSR for development.

A feature is fully enabled when the associated bit is zero. If a particular core does not support the disabling of a feature, the corresponding bit is hardwired to zero.

On reset, all implemented bits are set to 1, disabling all features. The bootloader is responsible for turning on all required features and can simply write zero to turn on the maximal set of features. SiFive's Freedom Metal bootloader handles turning on these features; when using a custom bootloader, clearing the Feature Disable CSR must be implemented.

Note that arbitrary toggling of the Feature Disable CSR bits is neither recommended nor supported; they are only intended to be set from 1 to 0. A particular Feature Disable CSR bit is only to be used in a very limited number of situations, as detailed in the **Example Usage** entry in Table 89.

| SiFive Feature Disable CSR | | | | |
|----------------------------|--------------------------------|-------|------|--|
| CSR | 0x7C1 | | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | disableDCacheClockGate | RW | 0x1 | Disable data cache clock gating |
| 1 | disableICacheClockGate | RW | 0x1 | Disable instruction cache clock gating |
| 2 | disableCoreClockGate | RW | 0x1 | Disable core clock gating |
| 3 | disableSpeculativeICacheRefill | RW | 0x1 | Disable speculative instruction cache refill |
| [6:4] | Reserved | RO | 0x0 | |
| 7 | disableTileClockGate | RW | 0x1 | Disable tile clock gating |
| 8 | Reserved | RO | 0x0 | |
| 9 | suppressCorruptOnGrantData | RW | 0x1 | Suppress corrupt signal on GrantData messages |
| [15:10] | Reserved | RO | 0x0 | |
| 16 | branchpredicationdisable | RW | 0x1 | Disable short forward branch optimization |
| 17 | prefetchdisable | RO | 0x0 | Disable instruction cache next-line prefetcher |
| [31:18] | Reserved | RO | 0x0 | |

Table 88: SiFive Feature Disable CSR

| Feature Disable CSR Usage | |
|---------------------------|---|
| Bit | Description / Usage |
| 3 | <p>Disable speculative instruction cache refill</p> <p>Example Usage A particular integration might require that execution from the System Port range be disallowed. Startup code would first configure PMP to prevent execution from the System Port range, followed by clearing bit 3 of the Feature Disable CSR. This would enable speculative instruction cache refill accesses, without allowing those to access the System Port range because PMP would prohibit such accesses.</p> |
| 9 | <p>Suppress corrupt signal on GrantData messages</p> <p>Example Usage 1 When running in debug mode on configurations having both ECC and a BEU, setting bit 9 of the Feature Disable CSR will suppress debug mode errors.</p> <p>Example Usage 2 Startup code could scrub errors present in RAMs at power-on, followed by clearing bit 9 of the Feature Disable CSR to allow normal operation.</p> |

Table 89: SiFive Feature Disable CSR Usage

6.2.3 Power Dial CSR

The Power Dial CSR, at 0x7C8, provides a method of scaling down dynamic power in a core in order to limit maximum power without frequency changes. This CSR is further described in Section 11.2.1.

6.3 Other Custom Instructions and CSRs

Other custom instructions and CSRs may be implemented, but their functionality is not documented further here, and they should not be used in this version of the S76 Core Complex.

Chapter 7

Interrupts and Exceptions

This chapter describes how interrupt and exception concepts in the RISC-V architecture apply to the S76 Core Complex.

7.1 Interrupt Concepts

Interrupts are *asynchronous* events that cause program execution to change to a specific location in the software application to handle the interrupting event. When processing of the interrupt is complete, program execution resumes back to the original program execution location. For example, a timer that triggers every 10 milliseconds will cause the CPU to branch to the interrupt handler, acknowledge the interrupt, and set the next 10 millisecond interval.

The S76 Core Complex supports machine mode interrupts.

The Core Complex also has support for the following types of RISC-V interrupts: local and global. Local interrupts are signaled directly to an individual hart with a dedicated interrupt exception code and fixed priority. This allows for reduced interrupt latency as no arbitration is required to determine which hart will service a given request and no additional memory accesses are required to determine the cause of the interrupt. Software and timer interrupts are local interrupts generated by the Core-Local Interruptor (CLINT). The S76 Core Complex contains no other local interrupt sources.

Global interrupts are routed through a Platform-Level Interrupt Controller (PLIC), which can direct interrupts to any hart in the system via the external interrupt. Decoupling global interrupts from the hart allows the design of the PLIC to be tailored to the platform, permitting a broad range of attributes like the number of interrupts and the prioritization and routing schemes.

Chapter 8 describes the CLINT. Chapter 9 describes the global interrupt architecture and the PLIC design.

7.2 Exception Concepts

Exceptions are different from interrupts in that they typically occur *synchronously* to the instruction execution flow, and most often are the result of an unexpected event that results in the program to enter an exception handler. For example, if a hart is operating in supervisor mode and attempts to access a machine mode only Control and Status Register (CSR), it will immediately

enter the exception handler and determine the next course of action. The exception code in the `mstatus` register will hold a value of `0x2`, showing that an illegal instruction exception occurred. Based on the requirements of the system, the supervisor mode application may report an error and/or terminate the program entirely.

There are no specific enable bits to allow exceptions to occur since they are always enabled by default. However, early in the boot flow, software should set up `mtvec.BASE` to a defined value, which contains the base address of the default exception handler. All exceptions will trap to `mtvec.BASE`. Software must read the `mcause` CSR to determine the source of the exception, and take appropriate action.

Synchronous exceptions that occur from within an interrupt handler will immediately cause program execution to abort the interrupt handler and enter the exception handler. Exceptions within an interrupt handler are usually the result of a software bug and should generally be avoided since `mepc` and `mcause` CSRs will be overwritten from the values captured in the original interrupt context.

The RISC-V defined synchronous exceptions have a priority order which may need to be considered when multiple exceptions occur simultaneously from a single instruction. Table 90 describes the synchronous exception priority order.

| Priority | Exception Code | Description |
|----------------|----------------|-----------------------------------|
| <i>Highest</i> | 3 | Instruction address breakpoint |
| | 12 | Instruction page fault |
| | 1 | Instruction access fault |
| | 2 | Illegal instruction |
| | 0 | Instruction address misaligned |
| | 8, 9, 11 | Environment call |
| | 3 | Environment break |
| | 3 | Load/Store/AMO address breakpoint |
| | 6 | Store/AMO address misaligned |
| | 4 | Load address misaligned |
| | 15 | Store/AMO page fault |
| <i>Lowest</i> | 13 | Load page fault |
| | 7 | Store/AMO access fault |
| | 5 | Load access fault |

Table 90: Exception Priority

Refer to Table 98 for the full list of exception codes.

Data address breakpoints (watchpoints), Instruction address breakpoints, and environment break exceptions (EBREAK) all have the same exception code (3), but different priority, as shown in the table above.

Instruction address misaligned exceptions (0) have lower priority than other instruction address exceptions because they are the result of control-flow instructions with misaligned targets, rather than from instruction fetch.

Some of the helpful CSRs for debugging exceptions and interrupts are described below:

| CSR | Description |
|-----------|--|
| exception | SiFive Scope signal. Indicates the moment that an exception occurs in the write-back (commit) stage. |
| mcause | Contains the cause value of the exception/interrupt. See Section 7.7.5 for more description. |
| mepc | Contains the pc where the exception occurs. |
| mtval | If the cause is a load/store fault, this register has the value of the problematic address. If it is an invalid instruction, it provides the instruction that the core tried to execute. |
| mstatus | Contains the interrupt enables, privilege modes, and general status of execution. See Section 7.7.1 for more description. |
| mtvec | Contains the vector that the core will jump to when an exception occurs. If this is not a valid executable value, you may get a double exception when jumping to the exception handler, so it is important to look at all these registers when the exception FIRST occurs. See Section 7.7.2 for more description. |

Table 91: Summary of Exception and Interrupt CSRs

7.3 Trap Concepts

The term trap describes the transfer of control in a software application, where trap handling typically executes in a more privileged environment. For example, a particular hart contains three privilege modes: machine, supervisor, and user. Each privilege mode has its own software execution environment including a dedicated stack area. Additionally, each privilege mode contains separate control and status registers (CSRs) for trap handling. While operating in user mode, a context switch is required to handle an event in supervisor mode. The software sets up the system for a context switch and then an ECALL instruction is executed, which synchronously switches control to the environment-call-from-user-mode exception handler.

The default mode out of reset is machine mode. Software begins execution at the highest privilege level, which allows all CSRs and system resources to be initialized before any privilege level changes. The steps below describe the required steps necessary to change privilege mode from machine to user mode, on a particular design that also includes supervisor mode.

1. Interrupts should first be disabled globally by writing `mstatus.MIE` to 0, which is the default reset value.
2. Write `mtvec` CSR with the base address of the machine mode exception handler. This is a required step in any boot flow.
3. Write `mstatus.MPP` to 0 to set the previous mode to user, which allows us to *return* to that mode.

4. Setup the Physical Memory Protection (PMP) regions to grant the required regions to user and supervisor mode, and optionally, revoke permissions from machine mode.
5. Write `stvec` CSR with the base address of the supervisor mode exception handler.
6. Write `medeleg` register to delegate exceptions to supervisor mode. Consider ECALL and page fault exceptions.
7. Write `mstatus.FS` to enable floating-point (if supported).
8. Store machine mode user registers to stack or to an application-specific frame pointer.
9. Write `mepc` with the entry point of user mode software
10. Execute `mret` instruction to enter user mode.

Note

There is only one set of user registers (x1-x31) that are used across all privilege levels, so application software is responsible for saving and restoring state when entering and exiting different levels.

7.4 Interrupt Block Diagram

The S76 Core Complex interrupt architecture is depicted in Figure 98.

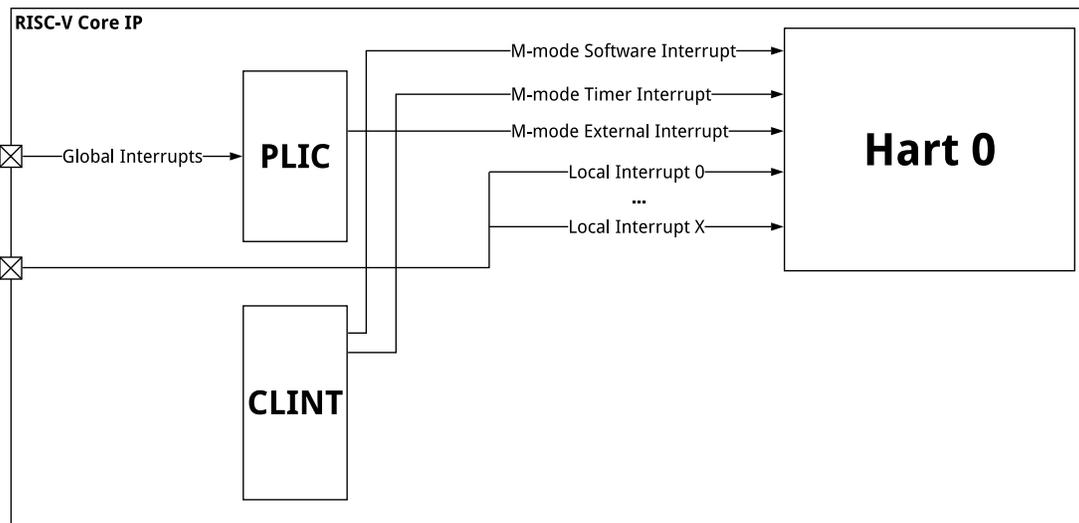


Figure 98: S76 Core Complex Interrupt Architecture Block Diagram

7.5 Local Interrupts

Software interrupts (Interrupt ID #3) are triggered by writing the memory-mapped interrupt pending register `msip` for a particular hart. The `msip` register is described in Table 96.

Timer interrupts (Interrupt ID #7) are triggered when the memory-mapped register `mtime` is greater than or equal to the global timebase register `mtimecmp`, and both registers are part of the CLINT memory map. The `mtime` and `mtimecmp` registers are generally only available in machine mode, unless the PMP grants user mode access to the memory-mapped region in which they reside.

Global interrupts are usually first routed to the PLIC, then into the hart using external interrupts (Interrupt ID #11).

7.6 Interrupt Operation

If the global interrupt-enable `mstatus.MIE` is clear, then no interrupts will be taken. If `mstatus.MIE` is set, then pending-enabled interrupts at a higher interrupt level will preempt current execution and run the interrupt handler for the higher interrupt level.

When an interrupt or synchronous exception is taken, the privilege mode is modified to reflect the new privilege mode. The global interrupt-enable bit of the handler's privilege mode is cleared.

7.6.1 Interrupt Entry and Exit

When an interrupt occurs:

- The value of `mstatus.MIE` is copied into `mcause.MPIE`, and then `mstatus.MIE` is cleared, effectively disabling interrupts.
- The privilege mode prior to the interrupt is encoded in `mstatus.MPP`.
- The current `pc` is copied into the `mepc` register, and then `pc` is set to the value specified by `mtvec` as defined by the `mtvec.MODE` described in Table 94.

At this point, control is handed over to software in the interrupt handler with interrupts disabled. When an `mret` instruction is executed, the following occurs:

- The privilege mode is set to the value encoded in `mstatus.MPP`.
- The global interrupt enable, `mstatus.MIE`, is set to the value of `mcause.MPIE`.
- The `pc` is set to the value of `mepc`.

At this point, control is handed over to software.

At the software level, interrupt attributes can be applied to interrupt processing functions, as described in Section 8.4.

The Control and Status Registers (CSRs) involved in handling RISC-V interrupts are described in Section 7.7.

7.7 Interrupt Control and Status Registers

The S76 Core Complex specific implementation of interrupt CSRs is described below. For a complete description of RISC-V interrupt behavior and how to access CSRs, please consult *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

7.7.1 Machine Status Register (mstatus)

The mstatus register keeps track of and controls the hart's current operating state, including whether or not interrupts are enabled. A summary of the mstatus fields related to interrupts in the S76 Core Complex is provided in Table 92. Note that this is not a complete description of mstatus as it contains fields unrelated to interrupts. For the full description of mstatus, please consult *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

| Machine Status Register (mstatus) | | | |
|-----------------------------------|------------|-------|-----------------------------------|
| CSR | 0x300 | | |
| Bits | Field Name | Attr. | Description |
| [2:0] | Reserved | WPRI | |
| 3 | MIE | RW | Machine Interrupt Enable |
| [6:4] | Reserved | WPRI | |
| 7 | MPIE | RW | Machine Previous Interrupt Enable |
| [10:8] | Reserved | WPRI | |
| [12:11] | MPP[1:0] | RW | Machine Previous Privilege Mode |

Table 92: Machine Status Register (partial)

Interrupts are enabled by setting the MIE bit in mstatus. Prior to writing mstatus.MIE=1, it is recommended to first enable interrupts in mie.

7.7.2 Machine Trap Vector (mtvec)

The mtvec register has two main functions: defining the base address of the trap vector, and setting the mode by which the S76 Core Complex will process interrupts. For Direct and Vectored modes, the interrupt processing mode is defined in the MODE field of the mtvec register. The mtvec register is described in Table 93, and the mtvec.MODE field is described in Table 94.

| Machine Trap Vector Register (mtvec) | | | |
|--------------------------------------|------------|-------|--|
| CSR | 0x305 | | |
| Bits | Field Name | Attr. | Description |
| [1:0] | MODE | WARL | MODE Sets the interrupt processing mode. The encoding for the S76 Core Complex supported modes is described in Table 94. |
| [63:2] | BASE[63:2] | WARL | Interrupt Vector Base Address. Operating in Direct Mode requires 4-byte alignment. Operating in Vectored Mode requires 256-byte alignment. |

Table 93: Machine Trap Vector Register

| MODE Field Encoding mtvec.MODE | | |
|--------------------------------|----------|--|
| Value | Mode | Description |
| 0x0 | Direct | All asynchronous interrupts and synchronous exceptions set pc to BASE. |
| 0x1 | Vectored | Exceptions set pc to BASE, interrupts set pc to BASE + 4 × mcause.EXCCODE. |
| ≥0x2 | Reserved | |

Table 94: Encoding of mtvec.MODE

Mode Direct

When operating in direct mode, all interrupts and exceptions trap to the mtvec.BASE address. Inside the trap handler, software must read the mcause register to determine what triggered the trap. The mcause register is described in Table 97.

When operating in Direct Mode, BASE must be 4-byte aligned.

Mode Vectored

While operating in vectored mode, interrupts set the pc to mtvec.BASE + 4 × exception code (mcause.EXCCODE). For example, if a machine timer interrupt is taken, the pc is set to mtvec.BASE + 0x1C. Typically, the trap vector table is populated with jump instructions to transfer control to interrupt-specific trap handlers.

In vectored interrupt mode, BASE must be 256-byte aligned.

All machine external interrupts (global interrupts) are mapped to exception code 11. Thus, when interrupt vectoring is enabled, the pc is set to address mtvec.BASE + 0x2C for any global interrupt.

7.7.3 Machine Interrupt Enable (mie)

Individual interrupts are enabled by setting the appropriate bit in the mie register. The mie register is described in Table 95.

| Machine Interrupt Enable Register (mie) | | | |
|---|------------|-------|-----------------------------------|
| CSR | 0x304 | | |
| Bits | Field Name | Attr. | Description |
| [2:0] | Reserved | WPRI | |
| 3 | MSIE | RW | Machine Software Interrupt Enable |
| [6:4] | Reserved | WPRI | |
| 7 | MTIE | RW | Machine Timer Interrupt Enable |
| [10:8] | Reserved | WPRI | |
| 11 | MEIE | RW | Machine External Interrupt Enable |
| [63:12] | Reserved | WPRI | |

Table 95: Machine Interrupt Enable Register

7.7.4 Machine Interrupt Pending (mip)

The machine interrupt pending (mip) register indicates which interrupts are currently pending. The mip register is described in Table 96.

| Machine Interrupt Pending Register (mip) | | | |
|--|------------|-------|------------------------------------|
| CSR | 0x344 | | |
| Bits | Field Name | Attr. | Description |
| [2:0] | Reserved | WIRI | |
| 3 | MSIP | RO | Machine Software Interrupt Pending |
| [6:4] | Reserved | WIRI | |
| 7 | MTIP | RO | Machine Timer Interrupt Pending |
| [10:8] | Reserved | WIRI | |
| 11 | MEIP | RO | Machine External Interrupt Pending |
| [63:12] | Reserved | WIRI | |

Table 96: Machine Interrupt Pending Register

7.7.5 Machine Cause (mcause)

When a trap is taken in machine mode, mcause is written with a code indicating the event that caused the trap. When the event that caused the trap is an interrupt, the most-significant bit of mcause is set to 1, and the least-significant bits indicate the interrupt number, using the same encoding as the bit positions in mip. For example, a Machine Timer Interrupt causes mcause to be set to 0x8000_0000_0000_0007. mcause is also used to indicate the cause of synchronous exceptions, in which case the most-significant bit of mcause is set to 0.

See Table 97 for more details about the mcause register. Refer to Table 98 for a list of synchronous exception codes.

| Machine Cause Register (mcause) | | | |
|---------------------------------|------------|-------|---|
| CSR | 0x342 | | |
| Bits | Field Name | Attr. | Description |
| [9:0] | EXCCODE | WLRL | A code identifying the last exception. |
| [62:10] | Reserved | WLRL | |
| 63 | Interrupt | WARL | 1, if the trap was caused by an interrupt; 0 otherwise. |

Table 97: Machine Cause Register

| Interrupt | Exception Code | Description |
|-----------|----------------|--------------------------------|
| 1 | 0–2 | Reserved |
| 1 | 3 | Machine software interrupt |
| 1 | 4–6 | Reserved |
| 1 | 7 | Machine timer interrupt |
| 1 | 8–10 | Reserved |
| 1 | 11 | Machine external interrupt |
| 1 | 12–13 | Reserved |
| 1 | 14 | Debug interrupt |
| 1 | ≥15 | Reserved |
| 0 | 0 | Instruction address misaligned |
| 0 | 1 | Instruction access fault |
| 0 | 2 | Illegal instruction |
| 0 | 3 | Breakpoint |
| 0 | 4 | Load address misaligned |
| 0 | 5 | Load access fault |
| 0 | 6 | Store/AMO address misaligned |
| 0 | 7 | Store/AMO access fault |
| 0 | 8 | Environment call from U-mode |
| 0 | 9–10 | Reserved |
| 0 | 11 | Environment call from M-mode |
| 0 | 12–13 | Reserved |
| 0 | 14 | Debug exception |
| 0 | ≥15 | Reserved |

Table 98: mcause Exception Codes

Note that there are scenarios where a misaligned load or store will generate an access exception instead of an address-misaligned exception. The access exception is raised when the misaligned access should not be emulated in a trap handler, e.g., emulating an access in an I/O region, as such emulation could cause undesirable side-effects.

7.7.6 Minimum Interrupt Configuration

The minimum configuration needed to configure an interrupt is shown below.

- Write `mtvec` to configure the interrupt mode and the base address for the interrupt vector table.
- Enable interrupts in memory-mapped PLIC register space. The CLINT does not contain interrupt enable bits.
- Write `mie` CSR to enable the software, timer, and external interrupt enables for each privilege mode.
- Write `mstatus` to enable interrupts globally for each supported privilege mode.

7.8 Interrupt Priorities

Individual priorities of global interrupts are determined by the PLIC, as discussed in Chapter 9.

S76 Core Complex interrupts are prioritized as follows, in decreasing order of priority:

- Machine external interrupts
- Machine software interrupts
- Machine timer interrupts

7.9 Interrupt Latency

Interrupt latency for the S76 Core Complex is four `external_source_for_core_N_clock` cycles, as counted by the number of cycles it takes from signaling of the interrupt to the hart to the first instruction fetch of the handler.

Global interrupts routed through the PLIC incur additional latency of three clock cycles, where the PLIC is clocked by `clock`. This means that the total latency, in cycles, for a global interrupt is: $4 + 3 \times (\text{external_source_for_core_N_clock Hz} \div \text{clock Hz})$. This is a best-case cycle count and assumes the handler is cached or located in ITIM. It does not take into account additional latency from a peripheral source.

7.10 Non-Maskable Interrupt

The `rnmi` (resumable non-maskable interrupt) interrupt signal is a level-sensitive input to the hart. Non-maskable interrupts have higher priority than any other interrupt or exception on the hart and cannot be disabled by software. Specifically, they are not disabled by clearing the `mstatus.mie` register.

7.10.1 Handler Addresses

The NMI has an associated exception trap handler address. This address is set by external input signals, described in the S76 Core Complex User Guide.

7.10.2 RNMI CSRs

These M-mode CSRs enable a resumable non-maskable interrupt (RNMI).

| Number | Name | Description |
|--------|----------|---|
| 0x350 | mnsratch | Resumable Non-maskable scratch register |
| 0x351 | mnepc | Resumable Non-maskable EPC value |
| 0x352 | mncause | Resumable Non-maskable cause value |
| 0x353 | mnstatus | Resumable Non-maskable status |

Table 99: RNMI CSRs

- The `mnsratch` CSR holds a 64-bit read-write register, which enables the NMI trap handler to save and restore the context that was interrupted.
- The `mnepc` CSR is a 64-bit read-write register, which, on entry to the NMI trap handler, holds the PC of the instruction that took the interrupt. The lowest bit of `mnepc` is hardwired to zero.
- The `mncause` CSR holds the reason for the NMI, with bit 63 set to 1, and the NMI cause encoded in the least-significant bits, or zero if NMI causes are not supported. The lower bits of `mncause`, defined as the `exception_code`, are as follows:

| mncause | NMI Cause | Function |
|---------|----------------|------------------------------------|
| 1 | Reserved | Reserved |
| 2 | RNMI input pin | External <code>rnmi_N</code> input |
| 3 | Reserved | Reserved |

Table 100: `mncause.exception_code` Fields

- The `mnstatus` CSR holds a two-bit field `mnpp` encoded in the same manner as `mstatus.mpp`, which, on entry to the trap handler, holds the privilege mode of the interrupted context.
- `mnstatus` also hold the one-bit field `mnie` indicating whether NMIs are currently enabled. This bit can only be cleared by hardware, but can be set by software to indicate a further NMI can be taken.

| Bits | Field Name | Description |
|-------|-------------------|------------------------------|
| 3 | <code>mnie</code> | RMNI interrupt enable |
| 7 | <code>mpv</code> | Hardwired to zero |
| 12:11 | <code>mnpp</code> | RMNI previous priority level |

Table 101: `mnstatus` CSR Fields

7.10.3 MNRET Instruction

This M-mode only instruction uses the values in `mnepc` and `mnstatus` to return to the program counter and privileged mode of the interrupted context. This instruction also sets the internal `rnmie` state bits.

Encoding is same as MRET except with bit 30 set (i.e., `funct7=0111000`). For example:

```
.word 0x70200073    // opcode for MNRET (return from RNMI)
```

7.10.4 RNMI Operation

When an RNMI interrupt is detected, the interrupted PC is written to the `mnepc` CSR, the type of RNMI to the `mncause` CSR, and the privilege mode of the interrupted context to the `mnstatus` CSR. An internal microarchitectural state bit, `rnmie`, is cleared to indicate that the processor is in an RNMI handler and cannot take a new RNMI interrupt. When clear, the internal `rnmie` bit also disables all other interrupts.

Note

These interrupts are called non-maskable because software cannot mask the interrupts. However, for correct operation, other instances of the same interrupt must be held off until the handler is completed, hence the internal state bit.

The RNMI handler can resume original execution using the MNRET instruction (described in Section 7.10.3), which restores the PC from `mnepc`, the privilege mode from `mnstatus`, and also sets the internal `rnmie` state bit, which re-enables other interrupts.

If the hart encounters an exception while the `rnmie` bit is clear, the exception state is written to `mepc` and `mcause`, `mstatus.mpp` is set to M-mode, and the hart jumps to the RNMI exception handler address.

Note

Traps in the RNMI handler can only be resumed if they occur while the handler was servicing an interrupt that occurred outside of machine mode.

Chapter 8

Core-Local Interruptor (CLINT)

This chapter describes the operation of the Core-Local Interruptor (CLINT). The S76 Core Complex CLINT complies with *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11*.

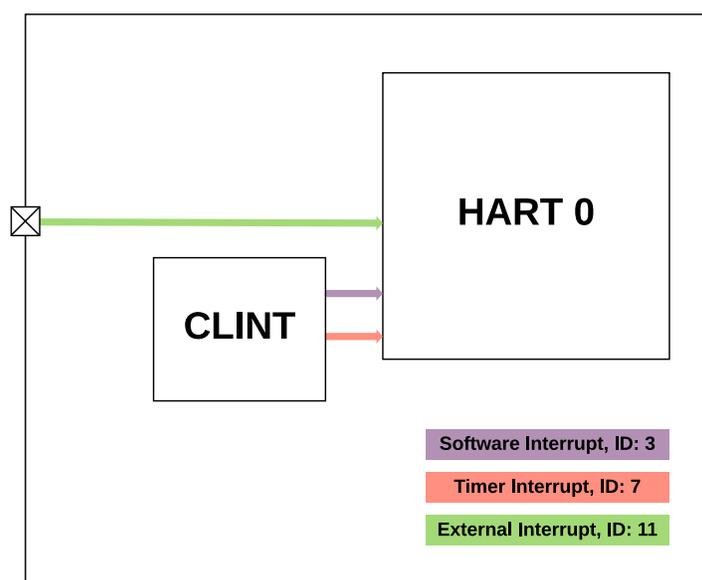


Figure 99: CLINT Block Diagram

The CLINT has a small footprint and provides software, timer, and external interrupts directly to the hart. The CLINT block also holds memory-mapped control and status registers associated with software and timer interrupts.

8.1 CLINT Priorities and Preemption

The CLINT has a fixed priority scheme, described in Section 7.8, and nested interrupts (pre-emption) within a given privilege level is not supported. Higher privilege levels may preempt lower privilege levels, however. The CLINT offers two modes of operation, Direct mode and Vectored mode.

In Direct mode, all interrupts and exceptions trap to `mtvec.BASE`. In Vectored mode, exceptions trap to `mtvec.BASE`, but interrupts will jump directly to their vector table index. See Section 7.7.2 for more information about `mtvec.BASE`.

8.2 CLINT Vector Table

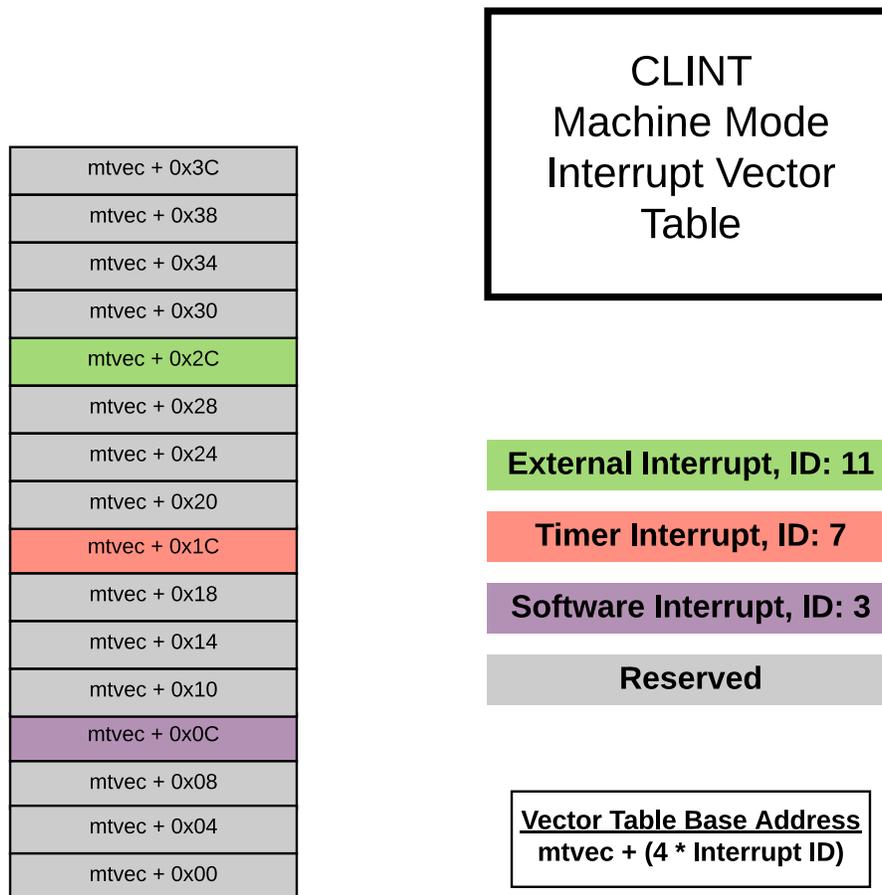


Figure 100: CLINT Interrupts and Vector Table

The CLINT vector table is populated with jump instructions, since hardware jumps to the index in the vector table first, then subsequently jumps to the handler. All exception types trap to the first entry in the table, which is `mtvec.BASE`.

An example CLINT vector table is shown below.

```
.weak default_exception_handler
.balign 4, 0
.global default_exception_handler

.weak software_handler
.balign 4, 0
.global software_handler

.weak timer_handler
.balign 4, 0
.global timer_handler

.weak external_handler
.balign 4, 0
.global external_handler

.option norvc
.weak __mtvec_clint_vector_table
#if __riscv_xlen == 32
.balign 128, 0
#else
.balign 256, 0
#endif
.global __mtvec_clint_vector_table
__mtvec_clint_vector_table:

IRQ_0:
    j default_exception_handler
IRQ_1:
    j default_vector_handler
IRQ_2:
    j default_vector_handler
IRQ_3:
    j software_handler
IRQ_4:
    j default_vector_handler
IRQ_5:
    j default_vector_handler
IRQ_6:
    j default_vector_handler
IRQ_7:
    j timer_handler
IRQ_8:
    j default_vector_handler
IRQ_9:
    j default_vector_handler
IRQ_10:
    j default_vector_handler
IRQ_11:
    j external_handler
IRQ_12:
    j default_vector_handler
IRQ_13:
    j default_vector_handler
IRQ_14:
    j default_vector_handler
IRQ_15:
    j default_vector_handler
```

Figure 101: CLINT Vector Table Example

8.3 CLINT Interrupt Sources

The S76 Core Complex supports the standard RISC-V software, timer, and external interrupts.

CLINT Interrupt IDs are provided in Table 102.

| S76 Core Complex Interrupt IDs | | |
|--------------------------------|-----------|----------------------------|
| ID | Interrupt | Notes |
| 0–2 | Reserved | |
| 3 | msip | Machine Software Interrupt |
| 4–6 | Reserved | |
| 7 | mtip | Machine Timer Interrupt |
| 8–10 | Reserved | |
| 11 | meip | Machine External Interrupt |

Table 102: S76 Core Complex Interrupt IDs

8.4 CLINT Interrupt Attribute

To help with efficiency of save and restore context, interrupt attributes can be applied to functions used for interrupt handling.

```
void __attribute__((interrupt))
software_handler (void) {
    // handler code
}
```

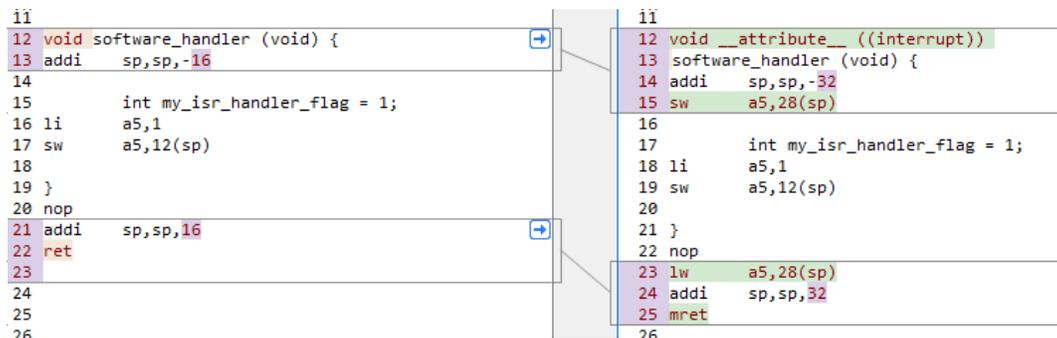


Figure 102: CLINT Interrupt Attribute Example

This attribute will save and restore registers that are used within the handler, and insert an `mret` instruction at the end of the handler.

8.5 CLINT Memory Map

Table 103 shows the memory map for CLINT on the S76 Core Complex. Note that there are no enable bits for specific interrupts within the CLINT memory map, as the enables for these inter-

rupts reside in the `mie` CSR for each interrupt, and the `mstatus.mie` CSR bit, which enables all machine interrupts globally. See Section 7.7.3 for a description of the interrupt enable bits in the `mie` CSR, and Section 7.7.4 for a description of the interrupt pending bits in the `mip` CSR.

| Address | Width | Attr. | Description | Notes |
|-------------|-------|-------|----------------------------------|----------------------------|
| 0x0200_0000 | 4B | RW | <code>msip</code> for hart 0 | MSIP Register (1-bit wide) |
| 0x0200_0004 | | | Reserved | |
| ... | | | | |
| 0x0200_3FFF | | | | |
| 0x0200_4000 | 8B | RW | <code>mtimecmp</code> for hart 0 | MTIMECMP Register |
| 0x0200_4008 | | | Reserved | |
| ... | | | | |
| 0x0200_BFF7 | | | | |
| 0x0200_BFF8 | 8B | RW | <code>mtime</code> | Timer Register |
| 0x0200_C000 | | | Reserved | |

Table 103: CLINT Memory Map

8.6 Register Descriptions

This section describes the functionality of the memory-mapped registers in the CLINT.

8.6.1 MSIP Registers

Machine mode software interrupts are generated by writing to the memory-mapped control register `msip`. The `msip` register is a 32-bit wide **WARL** register, where the upper 31 bits are tied to 0. The least-significant bit is reflected in the MSIP bit of the `mip` CSR. Other bits in the `msip` registers are hardwired to zero. On reset, each `msip` register is cleared to zero.

Software interrupts are most useful for interprocessor communication in multi-hart systems, as harts may write each other's `msip` bits to effect interprocessor interrupts.

8.6.2 Timer Registers

`mtime` is a 64-bit read-write register that contains the number of cycles counted from the `rtc_toggle` signal, which is described in the S76 Core Complex User Guide. A timer interrupt is pending whenever `mtime` is greater than or equal to the value in the `mtimecmp` register. The timer interrupt is reflected in the `mtip` bit of the `mip` register, described in Chapter 7.

On reset, `mtime` is cleared to zero. The `mtimecmp` registers are not reset.

Note that `mtime` is volatile and may be masked in IP-XACT.

Chapter 9

Platform-Level Interrupt Controller (PLIC)

This chapter describes the operation of the Platform-Level Interrupt Controller (PLIC) on the S76 Core Complex. The PLIC complies with *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* and can support a maximum of 127 external interrupt sources with 7 priority levels.

The S76 Core Complex PLIC resides in the `c1ock` timing domain, allowing for relaxed timing requirements. The latency of global interrupts, as perceived by a hart, increases with the ratio of the `external_source_for_core_N_c1ock` frequency and the `c1ock` frequency.

9.1 Memory Map

The memory map for the S76 Core Complex PLIC control registers is shown in Table 104. The PLIC memory map only supports aligned 32-bit memory accesses.

| Address | Width | Attr. | Description | Notes |
|-------------|-------|-------|--|--------------------------------------|
| 0x0C00_0000 | | | Reserved | |
| 0x0C00_0004 | 4B | RW | Source 1 priority | See Section 9.3 for more information |
| ... | | | | |
| 0x0C00_01FC | 4B | RW | Source 127 priority | |
| 0x0C00_0200 | | | Reserved | |
| ... | | | | |
| 0x0C00_1000 | 4B | RO | Start of pending array | See Section 9.4 for more information |
| ... | | | | |
| 0x0C00_100C | 4B | RO | Last word of pending array | |
| 0x0C00_1010 | | | Reserved | |
| ... | | | | |
| 0x0C00_2000 | 4B | RW | Start Hart 0 M-Mode interrupt enables | See Section 9.5 for more information |
| ... | | | | |
| 0x0C00_200C | 4B | RW | End Hart 0 M-Mode interrupt enables | |
| 0x0C00_2010 | | | Reserved | |
| ... | | | | |
| 0x0C1F_F000 | 1B | RW | PLIC global clock gating disable feature | See Section 9.6 for more information |
| 0x0C1F_F001 | | | Reserved | |
| ... | | | | |
| 0x0C20_0000 | 4B | RW | Hart 0 M-Mode priority threshold | See Section 9.7 for more information |
| 0x0C20_0004 | 4B | RW | Hart 0 M-Mode claim/complete | See Section 9.8 for more information |
| 0x0C20_0008 | | | Reserved | |
| ... | | | | |
| 0x1000_0000 | | | End of PLIC Memory Map | |

Table 104: PLIC Memory Map

9.2 Interrupt Sources

The S76 Core Complex has a total of 127 external global interrupt sources, in addition to the local interrupts described in Table 102.

Note

In the *RISC-V Platform-Level Interrupt Controller Specification*, interrupt source 0 (ID 0) is unused, so the first usable PLIC Interrupt ID has a value of 1.

Table 105 describes the mapping of external global interrupts to its corresponding top-level `global_interrupts` signal bit. This signal is positive-level triggered and not configurable. See the S76 Core Complex User Guide for further description of `global_interrupts`.

| global_interrupts Signal | PLIC Interrupt ID | PLIC Pending / Enable Register |
|---|--------------------------|---|
| <code>global_interrupts[0]</code> | 1 | <code>pending1[1] / enable1[1]*</code> |
| <code>global_interrupts[1]</code> | 2 | <code>pending1[2] / enable1[2]</code> |
| <code>global_interrupts[2]</code> | 3 | <code>pending1[3] / enable1[3]</code> |
| ... | | |
| <code>global_interrupts[126]</code> | 127 | <code>pending4[31] / enable4[31]</code> |
| * <code>pending1[0]</code> and <code>enable1[0]</code> are unused | | |

Table 105: Mapping of `global_interrupts` Signal Bits to PLIC Interrupt ID

9.3 Interrupt Priorities

Each PLIC interrupt source can be assigned a priority by writing to its 32-bit memory-mapped priority register. The S76 Core Complex supports 7 levels of priority. A priority value of 0 is reserved to mean "never interrupt" and effectively disables the interrupt. Priority 1 is the lowest active priority, and priority 7 is the highest. Ties between global interrupts of the same priority are broken by the Interrupt ID; interrupts with the lowest ID have the highest effective priority. See Table 106 for the detailed register description.

| PLIC Interrupt Priority Register | | | | |
|---|-------------------|---|-------------|---------------------------|
| Base Address | | $0x0C00_0000 + 4 \times \text{Interrupt ID}$ | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [2:0] | Priority | RW | X | Global interrupt priority |
| [31:3] | Reserved | RO | 0x0 | |

Table 106: PLIC Interrupt Priority Register

9.4 Interrupt Pending Bits

The current status of the interrupt source pending bits in the PLIC core can be read from the pending array, organized as 4 words of 32 bits. The pending bit for interrupt ID N is stored in bit $(N \bmod 32)$ of word $(N / 32)$. As such, the S76 Core Complex has four interrupt pending registers. Bit 0 of word 0, which represents the non-existent interrupt source 0, is hardwired to zero.

A pending bit in the PLIC core can be cleared by setting the associated enable bit then performing a claim as described in Section 9.8.

| PLIC Interrupt Pending Register 1 | | | | |
|-----------------------------------|----------------------|-------------|------|--|
| Base Address | | 0x0C00_1000 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | Interrupt 0 Pending | RO | X | Non-existent global interrupt 0 is hardwired to zero |
| 1 | Interrupt 1 Pending | RO | X | Pending bit for global interrupt 1 |
| 2 | Interrupt 2 Pending | RO | X | Pending bit for global interrupt 2 |
| ... | | | | |
| 31 | Interrupt 31 Pending | RO | X | Pending bit for global interrupt 31 |

Table 107: PLIC Interrupt Pending Register 1

| PLIC Interrupt Pending Register 4 | | | | |
|-----------------------------------|-----------------------|-------------|------|--------------------------------------|
| Base Address | | 0x0C00_100C | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | Interrupt 96 Pending | RO | X | Pending bit for global interrupt 96 |
| ... | | | | |
| 31 | Interrupt 127 Pending | RO | X | Pending bit for global interrupt 127 |

Table 108: PLIC Interrupt Pending Register 4

9.5 Interrupt Enables

Each global interrupt can be enabled by setting the corresponding bit in the enable registers. The enable registers are accessed as a contiguous array of 4×32 -bit words, packed the same way as the pending bits. Bit 0 of enable word 0 represents the non-existent interrupt ID 0 and is hardwired to 0.

64-bit and 32-bit word accesses are supported by the enables array in SiFive RV64 systems.

| PLIC Interrupt Enable Register 1 for Hart 0 M-Mode | | | | |
|--|---------------------|-------------|------|--|
| Base Address | | 0x0C00_2000 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | Interrupt 0 Enable | RO | 0x0 | Non-existent global interrupt 0 is hardwired to zero |
| 1 | Interrupt 1 Enable | RW | X | Enable bit for global interrupt 1 |
| 2 | Interrupt 2 Enable | RW | X | Enable bit for global interrupt 2 |
| ... | | | | |
| 31 | Interrupt 31 Enable | RW | X | Enable bit for global interrupt 31 |

Table 109: PLIC Interrupt Enable Register 1 for Hart 0 M-Mode

| PLIC Interrupt Enable Register 4 for Hart 0 M-Mode | | | | |
|--|----------------------|-------------|------|-------------------------------------|
| Base Address | | 0x0C00_200C | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | Interrupt 96 Enable | RW | X | Enable bit for global interrupt 96 |
| ... | | | | |
| 31 | Interrupt 127 Enable | RW | X | Enable bit for global interrupt 127 |

Table 110: PLIC Interrupt Enable Register 4 for Hart 0 M-Mode

9.6 PLIC Clock Gate Disable

The PLIC implements a clock gating feature to gate the module clock node when not active. PLIC clock gating is disabled out of reset and should be enabled in startup code, unless otherwise specified by SiFive erratum. Once enabled, clock is only available when there is activity on the PLIC control bus or on any interrupt line when the corresponding interrupt is not in flight. Clock gating is further described in the S76 Core Complex User Guide.

| PLIC Clock Gate Disable Register | | | | |
|----------------------------------|-----------------------------|-------------|------|--|
| Base Address | | 0x0C1F_F000 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | disablePlicClockGateFeature | RW | 0x1 | Used to enable/disable PLIC clock gating feature. Clear to enable. |
| [7:1] | Reserved | RO | 0x0 | |

Table 111: PLIC Clock Gate Disable Register

9.7 Priority Thresholds

The S76 Core Complex supports setting of an interrupt priority threshold via the `threshold` register. The `threshold` is a **WARL** field, where the S76 Core Complex supports a maximum threshold of 7.

The S76 Core Complex masks all PLIC interrupts of a priority less than or equal to `threshold`. For example, a `threshold` value of zero permits all interrupts with non-zero priority, whereas a value of 7 masks all interrupts. If the `threshold` register contains a value of 5, all PLIC interrupt configured with priorities from 1 through 5 will not be allowed to propagate to the CPU.

| PLIC Interrupt Priority Threshold Register for Hart 0 M-Mode | | | | |
|--|------------|-------------|------|-----------------------------|
| Base Address | | 0x0C20_0000 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [2:0] | Threshold | WARL | X | Sets the priority threshold |
| [31:3] | Reserved | RO | 0x0 | |

Table 112: PLIC Interrupt Priority Threshold Register for Hart 0 M-Mode

9.8 Interrupt Claim Process

A S76 Core Complex hart can perform an interrupt claim by reading the `claim_complete` register (Table 113), which returns the ID of the highest-priority pending interrupt or zero if there is no pending interrupt. A successful claim also atomically clears the corresponding pending bit on the interrupt source.

A S76 Core Complex hart can perform a claim at any time, even if the `MEIP` bit in its `mip` (Table 96) register is not set.

The claim operation is not affected by the setting of the priority threshold register.

9.9 Interrupt Completion

A S76 Core Complex hart signals it has completed executing an interrupt handler by writing the interrupt ID it received from the claim to the `claim_complete` register (Table 113). The PLIC does not check whether the completion ID is the same as the last claim ID for that target. If the completion ID does not match an interrupt source that is currently enabled for the target, the completion is silently ignored.

| PLIC Claim/Complete Register for Hart 0 M-Mode | | | | |
|--|--|-------------|------|---|
| Base Address | | 0x0C20_0004 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [31:0] | Interrupt Claim/Complete for Hart 0 M-Mode | RW | X | A read of zero indicates that no interrupts are pending. A non-zero read contains the ID of the highest pending interrupt. A write to this register signals completion of the interrupt ID written. Consecutive reads will not return the same result. Note that this field is volatile and may be masked in IP-XACT. |

Table 113: PLIC Claim/Complete Register for Hart 0 M-Mode

The PLIC cannot forward a new interrupt to a hart that has claimed an interrupt, but has not yet finished the complete step of the interrupt handler. Thus, the PLIC does not support preemption of global interrupts to an individual hart.

Interrupt IDs for global interrupts routed through the PLIC are independent of the interrupt IDs for local interrupts. The PLIC handler may check for additional pending global interrupts once the initial claim/complete process has finished, prior to exiting the handler. This method could save additional PLIC save/restore context for global interrupts.

9.10 Example PLIC Interrupt Handler

Since the PLIC interfaces with the CPU through external interrupt #11, the external handler must contain an additional claim/complete step that is used to handshake with the PLIC logic.

```
void external_handler() {
    //get the highest priority pending PLIC interrupt
    uint32_t int_num = plic.claim_complete;

    //branch to handler
    plic_handler[int_num]();

    //complete interrupt by writing interrupt number back to PLIC
    plic.claim_complete = int_num;

    // Add additional checks for PLIC pending here, if desired
}
```

If a CPU reads `claim_complete` and it returns 0, the interrupt does not require processing, and thus write-back of the claim/complete is not necessary.

The `plic_handler[]()` routine shown above demonstrates one method to implement a software table where the offset of the function that resides within the table is determined by the

PLIC interrupt ID. The PLIC interrupt ID is unique to the PLIC, in that it is completely independent of the interrupt IDs of local interrupts.

Chapter 10

TileLink Error Device

The Error Device is a TileLink slave that responds to all requests with a TileLink denied error and all reads with a corrupt error. It has no registers. The entire memory range discards writes and returns zeros on read. Both operation acknowledgements carry an error indication.

The Error Device serves a dual role. Internally, it is used as a landing pad for illegal off-chip requests. However, it is also useful for testing software handling of bus errors.

Chapter 11

Power Management

The following chapter describes power modes and establishes flows for powering up, powering down, and resetting the hardware of the S76 Core Complex.

11.1 Power Modes

Power modes include normal run mode with the Power Dial option and wait-for-interrupt clock gating mode using the `wFI` instruction. Additionally, there is a full power down mode supported via the `CEASE` instruction. These modes are covered in detail below.

11.2 Run Mode

The hart is fully operational in run mode, and SiFive designs include the option to include coarse-grained architectural clock gating. When this feature is enabled in the hart, configured instruction cache, data cache, integer pipeline, Debug Logic, and Floating-Point Unit (FPU) modules each contain their own clock gate. The clock gating feature will enable automatic clock gating of functional units when they are inactive and allow the hart to gate its own clock(s) based on activity.

11.2.1 Power Control

Core Clock

To further reduce power while in run mode, users may choose to reduce `external_source_for_core_N_clock`, which is required to be changed synchronously to the rest of the clocks in the system. It is important to note that the clock relationships with the rest of the system must still be maintained if `external_source_for_core_N_clock` is reduced. `external_source_for_core_N_clock` and all other clock signals are described in the S76 Core Complex User Guide.

Power Dial

To limit maximum power without frequency changes, Power Dial provides a method of scaling down dynamic power in a core. Power is reduced by restricting cycles allowed to advance

instructions into the execution pipeline. This feature can typically be used for throttling high CPU usage applications.

| Power Dial CSR | | | | |
|----------------|------------|-------|------|---|
| CSR | 0x7C8 | | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [3:0] | dutycycle | RW | 0x0 | (1 - value/16) portion of the peak instruction throughput (i.e., value=0 is no reduction) |
| [7:4] | Reserved | RO | 0x0 | |

Table 114: Power Dial CSR

The Power Dial Register may only be programmed in M-mode. When written with a non-zero value, the register restricts peak instruction throughput to the indicated rate. A value of 0 has no effect on instruction throughput. The rate is calculated per 256-cycle period, so a Power Dial value of 1 restricts instruction throughput at a common point in the pipeline to 240 cycles of each 256-cycle period. Reducing the peak rate reduces the worst-case power while minimizing the impact on performance.

11.3 WFI Clock Gate Mode

WFI clock gating mode can be entered by executing the WFI instruction. The assembly-level instruction is simply `wfi` and executing the C method using the GCC compiler can be accomplished with `asm("WFI")`.

11.3.1 WFI Wake Up

Wake up from a WFI occurs when the hart receives any interrupt. Depending on the software configuration, the hart will either immediately enter the interrupt handler, or resume execution on the instruction immediately after the WFI.

If interrupts are enabled and `mstatus.MIE=1`, then the hart will wake when an interrupt is enabled and becomes pending, and immediately enter the interrupt handler. Upon exit from the interrupt handler, program execution will resume at the instruction following the WFI.

If interrupts are enabled but `mstatus.MIE=0`, then the hart will wake when an interrupt is enabled and becomes pending but will not enter the interrupt handler. It will simply resume at the instruction immediately after the WFI in this case.

To prevent an interrupt source from waking a hart, the enable bit for that interrupt must be written to 0 prior to executing the WFI instruction. If any interrupts are pending upon executing a WFI instruction, then the WFI is effectively treated as a NOP instruction.

Refer to Chapter 7 for more detail on interrupt configuration.

11.4 CEASE Instruction for Power Down

To fully power down, follow the steps described in Section 11.11, where the last step is to execute a CEASE instruction. Once the CEASE instruction is executed, the core will not retire another instruction until reset. The CEASE opcode is 0x30500073 and can be implemented in either assembly or C. To create an assembly-level function using GCC, consider the following example.

```
.global _cease
.type      _cease, @function
_cease:
    .word 0x30500073
    ret
```

The next example demonstrates how to implement the CEASE instruction within a function in C.

```
static inline void cease()
{
    __asm__ __volatile__ (".word 0x30500073" : : : "memory"); // CEASE
}
```

11.5 Subsystem LowPower Controller (SLPC)

The Subsystem LowPower Controller (SLPC) is a unit inside the uncore that contains power management-related logic.

It includes the CorePowerState register that indicates the individual WFI and CEASE status for up to 16 Tiles in the core-complex.

It is used by core-complex power gating, but can be present when core-complex power gating is not enabled or supported.

Below are the memory map and bitfield definitions for the SLPC:

| Location | Description |
|--------------|--------------------------|
| base + 0x000 | CorePower State Register |
| base + 0x008 | Reserved |

Table 115: CorePower State Memory Map

| CorePower State Register | | |
|--------------------------|-----------|--|
| Bits | Field | Description |
| [15:0] | wfi_<x> | WFI indication from Tile-X, where x is the Tile Index. WFI status for up to 16 Tiles can be captured here. |
| [31:16] | cease_<x> | CEASE indication from Tile-X, where x is the Tile Index. CEASE status for up to 16 Tiles can be captured here. |

Table 116: CorePower State Register

11.6 Composable Cache Clock Gating

The Composable Cache implements two levels of architectural clock gating to gate clock nodes when not active, reducing its clock tree and dynamic power.

1. The first level is a trunk clock gate on the entire Composable Cache Wrapper; that is, the clock to the entire module and all its flops is gated. This feature is disabled (i.e., the clock is always enabled) out of reset.

DisableClockGate register bit 0 (DisableWFICTrunkClockGate) enables the feature during start-up code. The clock gate only becomes active when all the cores in the Core IP subsystem are in WFI mode and the Composable Cache is idle. This saves internal clock tree and Flop's dynamic power in WFI mode. When any of the core is out of WFI mode, the clock is always enabled.

If the feature is enabled, in WFI mode, the clock is only available under the following conditions:

- Activity on the ingress memory bus initiated by any bus master
- Activity on the LIM bus
- Activity on the Composable Cache TileLink control bus

The clock remains available until all the above inflight transactions are finished at the Composable Cache Wrapper boundary.

2. The second level a regional clock gate that gates the clock of some the Composable Cache's major units. This feature is disabled (i.e., the clock is always enabled) out of reset.

DisableClockGate register bit 1 (DisableCCRegionalClockGate) enables the feature during start-up code.

Below are the memory map and bitfield definitions for the DisableClockGate register:

| Location | Description |
|---------------|---------------------------|
| base + 0x1000 | DisableClockGate Register |

Table 117: DisableClockGate Memory Map

| DisableClockGate Register | | | | |
|---------------------------|--------------------------------|------|-------------|---|
| Bits | Field | Attr | Reset State | Description |
| 2 | DisableCCRegionalClockGateSlow | RW | 0x1 | When set, disables regional clock gating "slow" feature. When clear, if regional clock gating is also enabled, the aggressive clock gating feature is enabled. This clock gater imposes a 1-cycle exit penalty from the gated state (on the first TL channel-A request), but saves more power than the no-penalty regional clock gate option. |
| 1 | DisableCCRegionalClockGate | RW | 0x1 | When set, disables all regional clock gating. When clear, non-aggressive clock gating is enabled. This clock gater has no penalty for exit, but less power savings than regional "slow" clock gating. |
| 0 | DisableWFICCTrunkClockGate | RW | 0x1 | When set, disables trunk clock gating feature. When clear, dynamic gating of the Ccache clock trunk is enabled. This clock gater incurs a 2-cycle penalty when exiting the WFI clock gated state and saves the most power. All cores in a core-complex must be in WFI before this state is entered. |

Table 118: DisableClockGate Register

Note

To enable the regional clock gating "slow" feature, both bits 1 and 2 need to be cleared.

11.7 Hardware Reset

The following list summarizes the hardware reset values required by *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.11* and applies to all SiFive designs.

1. Privilege mode is set to machine mode.

2. `mstatus.MIE` and `mstatus.MPRV` are required to be 0.
3. The `misa` register holds the full set of supported extensions for that implementation, and `misa.MXL` defaults to the widest supported ISA available, referred to as `MXLEN`.
4. The `pc` is set to the implementation specific reset vector.
5. The `mcause` register is set to 0x0 at reset.
6. The PMP configuration fields for address matching mode (A) and Lock (L) are set to 0, which defaults to no protection for any privilege level.

The internal state of the rest of the system should be completed by software early in the boot flow.

11.8 Early Boot Flow

For the early stages of boot, some of the first things software must consider are listed below:

- The global pointer (`gp` or `x3`) user register should be initialized to the `__global_pointer$` linker generated symbol and not changed at any point in the application program.
- The stack pointer (`sp` or `x2`) user register should be also set up as a standard part of the boot flow.
- All other user registers (`x1`, `x4` - `x31`) can be written to 0 upon initial power-on.
- The `mtvec` register holds the default exception handler base address, so it is important to set up this register early in the boot flow, so it points to a properly aligned, valid exception handler location.
- Zero out the `bss` section and copy data sections into RAM areas as needed.

11.9 Interrupt State During Early Boot

Since `mstatus.MIE` defaults to 0, all interrupts are disabled globally out of reset. Prior to enabling interrupts globally through `mstatus.MIE`, consider the following:

- Ensure no timer interrupts are pending by checking the `mip.MTIP` bit. The `mtime` register is 0 out of reset and starts running immediately. However, the `mtimecmp` register does not have a reset value.

If no timer interrupt is required, leave `mie.MTIE` equal to 0 prior to enabling global interrupt with `mstatus.MIE`.

If the application requires a timer interrupt, write `mtimecmp` to a value in the future for the next timer interrupt before enabling `mstatus.MIE`.

- Write the remaining bits in the `mie` CSR to the desired value to enable interrupts based on the requirements of the system. This register is not defined to have a reset value.

- Each `msip` register in the Core-Local Interruptor (CLINT) or Core-Local Interrupt Controller (CLIC) address space is reset to 0, so no specific initialization is required for local software interrupts.

Since `msip` is memory-mapped, any hart in the system may trigger a software interrupt on another hart, so this should be considered during the boot flow on a multi-hart system.

- If a Platform-Level Interrupt Controller (PLIC) exists, check the PLIC pending status. The PLIC memory mapped pending bits are read-only, so the pending status should be cleared at the source if they reset to a non-zero status. Then, enable the PLIC interrupts as required by the system prior to enabling interrupts in the system via `mstatus.MIE`.

11.10 Other Boot Time Considerations

- Write 0 to enable the appropriate bits in the Feature Disable CSR as described in Section 6.2.2.
- Ensure the remaining bits in the `mstatus` CSR are written to the desired application specific configuration at boot time.
- If a design includes user and supervisor privilege levels, initialize `medeleg` and `mideleg` registers to 0 until supervisor-level trap handling is set up correctly using `stvec`.
- The `mcause`, `mepc`, and `mtval` registers hold important information in the event of a synchronous exception. If the synchronous exception handler forces reset in the application, the contents of these registers can be checked to understand root cause.
- The PMP address and configuration CSRs are required to be initialized if user or supervisor privilege levels are part of the design. By default, user and supervisor modes have no permissions to the memory map unless explicitly granted by the PMP.
- The `mcycle` CSR is a 64-bit counter on both RV32 and RV64 systems, and it counts the number of cycles executed by the hart. It has an arbitrary value after reset and can be written as needed by the application.
- Instructions retired can be counted by the `minstret` register, and this also has an arbitrary value after reset. This can be written to any given value.
- The `mhpmeventX` CSR selects which hardware events to count, where the count is reflected in `mhpmpcounterX`. At any point, the `mhpmpcounterX` registers can be directly written to reset their value when the `mhpmeventX` register has the proper event selected.
- There is no requirement for boot time initialization to any of the registers within the Debug Module, unless there is an application specific reason to do so.
- All other CSRs during boot time initialization should be considered based on system and application requirements.

11.11 Power-Down Flow

For SiFive Core IP, coordination with an "External Agent" is required.

1. External Agent: Wait for communication from the core to initiate the following steps:
 - a. Stop sending inbound traffic (both transactions and interrupts) into the Core Complex.
 - b. Wait until all outstanding requests to the Core Complex are completed, then
 - c. Wait until `cease_from_tile_N` is high for the core.
 - d. Once `cease_from_tile_N` is high for the core, apply reset to the entire Core Complex.
2. Core:
 - a. The following sequence should be executed in machine mode and NOT out of a remote ITIM/DTIM.
 - b. Communicate with external agent to initiate cease power-down sequence.
 - c. Poll external agent until steps 1.a and 1.b are completed.
 - d. Disable all interrupts except those related to bus errors/memory corruption.
 - i. Copy contents of any TIMs/LIMs into external memory.
 - ii. If there is an L2 cache, flush it.
 - iii. If there is no L2 cache, but there is a data cache, flush it.
 - e. Disable all interrupts.
 - f. Execute CEASE instruction.

Chapter 12

Debug

This chapter describes the operation of SiFive debug hardware, which follows *The RISC-V Debug Specification, Version 1.0*. Currently, only interactive debug and hardware breakpoints are supported.

12.1 Debug Module

The Debug Module (DM) handles nearly all of the functions related to debugging. It is a slave to both the Debug Module Interface (DMI) coming from the probe, and a TileLink bus coming from the core. From the perspective of the core, the DM appears as a 4KiB block in the memory map. The DM memory map as seen from the perspective of the core is shown in Table 120, and the register map from the perspective of the DMI is shown in Table 119.

Most of the DM is clocked by `debug_clock`. The `dmcontrol` register is accessible when `debug_clock` is not running, mainly to be able to write to `haltreq` while the core is in reset due to `ndreset`. Doing so generates a debug interrupt and will interrupt the selected core immediately once it is out of reset or during a WFI instruction.

| DMI Address | Name | Description |
|-------------|--------------------|---|
| 0x04-0x0F | data0-data11 | Read/Write DATA registers. 32-bit SiFive cores have 1 data register, 64-bit cores have 2. Note that these registers are volatile and may be masked in IP-XACT. |
| 0x10 | dmcontrol | Debug Module Control. See Table 131 for more information. |
| 0x11 | dmstatus | Debug Module Status. See Table 130 for more information. |
| 0x12 | hartinfo | Hart Information. See Table 132 for more information. |
| 0x13 | haltsum1 | Read-only. Halt Summary 1. Only present on systems with >32 harts. Not used by SiFive. |
| 0x16 | abstractcs | Abstract Control and Status. See Table 133 for more information. |
| 0x17 | command | Initiate abstract command. See Table 134 for more information. |
| 0x18 | abstractauto | Selects whether access to particular DATA or PROGBUF locations will re-execute the last command. Used for block transfers or other repeating commands. See Table 135 for more information. |
| 0x20-0x2F | progbuf0-progbuf15 | Read/Write PROGBUF registers. |
| 0x32 | dmcs2 | Fields to set up and read back Halt Group or Resume Group configuration. Present by default on systems with more than 1 hart or with any external triggers. See Table 136 for more information. |
| 0x38 | sbc | System Bus Access Control and Status |
| 0x39 | sbaddress0 | System Bus Address 31:0 |
| 0x3A | sbaddress1 | System Bus Address 63:32. |
| 0x3C | sbdata0 | System Bus Data 31:0 |
| 0x3D | sbdata1 | System Bus Data 63:32 |
| 0x40 | haltsum0 | Read-only. Halt Summary 0. Bit n reads 1 if hart n is halted. Note that this register is volatile and may be masked in IP-XACT. |

Table 119: Debug Module Memory Map Seen from the Debug Module Interface

From the point of view of the core, the DM appears as a 4KiB block of memory. It is mapped into low memory so that memory references can use addresses relative to the \$zero register.

| TL Address | Name | Attr. | Description |
|-------------|-----------|-------|---|
| 0x100 | HALTED | WO | Written with hartid by ROM code when hart gets a debug interrupt or reenters ROM due to EBREAK. Sets halted[hartid]. If an abstract command was running, writing this also clears busy. |
| 0x104 | GOING | WO | Written by ROM code when it begins executing a command started by FLAGS[hartid].go. Clears FLAGS[hartid].go. |
| 0x108 | RESUMING | WO | Written with hartid by hart when it is about to resume. Sets resumeack[hartid] and clears halted[hartid] and FLAGS[hartid].resume. |
| 0x10C | EXCEPTION | WO | Written by hart when it encounters an exception in debug mode. Sets cmderr to "exception". |
| 0x300 | WHERE TO | RO | JAL to ABSTRACT. This opcode is constructed by DM hardware and is needed because ABSTRACT is not a fixed address (depends on number of PROGBUF words selected in the configuration). Note that ABSTRACT is volatile and may be masked in IP-XACT. |
| contiguous | ABSTRACT | RO | 2 words constructed by DM hardware based on abstract command written from DTM. <ul style="list-style-type: none"> • 0 - If transfer set, construct instruction to load/store specific register to/from DATA[0] (32 bits) or DATA[1:0] (64 bits), else NOP. • 4 - If postexec set, then NOP to fall thru and execute PROGBUF, else EBREAK to return to ROM park loop. <p>Note that this register is volatile and may be masked in IP-XACT.</p> |
| contiguous | PROGBUF | RW | Configurable number (typically 16, max 16) of R/W words to be filled in by debugger and executed by hart. Note that this register is volatile and may be masked in IP-XACT. |
| 0x380-0x3BF | DATA | RW | Configurable number (1 for 32-bit or 2 for 64-bit, max 12) of R/W words intended for use for data transfer between debugger and hart. Since it is contiguous with PROGBUF, the debugger may use DATA as an extension of PROGBUF. Note that these registers are volatile and may be masked in IP-XACT. |
| 0x400-0x7FF | FLAGS | RO | One byte flag per hart. <ul style="list-style-type: none"> • Bit 0 (go): Set by writing an abstract command, cleared by ROM write to GOING. ROM will jump to WHERE TO. |

Table 120: Debug Module Memory Map from the Perspective of the Core

| TL Address | Name | Attr. | Description |
|-------------|------|-------|---|
| | | | <ul style="list-style-type: none"> Bit 1 (resume): Set by writing 1 to resumereq[hartid]. Cleared by ROM write of hartid to RESUMING. ROM restores s0 then executes dret. <p>Note that these registers are volatile and may be masked in IP-XACT.</p> |
| 0x800-0xFFF | ROM | RO | <p>Debug interrupt or EBREAK enters at 0x800, saves s0, writes hartid to HALTED, then busy-waits for FLAGS[hartid] > 0.</p> <p>If FLAGS[hartid].go, write 0 to GOING, then jump to WHERETO.</p> <p>Else write hartid to RESUMING, then execute dret to return to user program.</p> <p>ROM Source Code: https://github.com/chipsalliance/rocket-chip/blob/master/scripts/debug_rom/debug_rom.S</p> |

Table 120: Debug Module Memory Map from the Perspective of the Core

12.2 Debug and Trigger Registers

This section describes the per hart debug and trigger registers, which are mapped into the CSR space as follows:

| CSR | Name | Allowed Access Mode | Description |
|-------|-----------|---------------------|---|
| 0x7B0 | dcsr | Debug | Debug Control and Status Register |
| 0x7B1 | dpc | Debug | Debug PC. Stores execution address just before debug exception and to return to at dret. |
| 0x7B2 | dscratch0 | Debug | Debug Scratch Register 0 |
| 0x7A0 | tselect | Debug, Machine | Trigger Select. Most configs implement 2, 4, or 8 triggers. Triggers are all type 6 (address/data). |
| 0x7A1 | tdata1 | Debug, Machine | Trigger Data 1, mcontrol6 |
| 0x7A2 | tdata2 | Debug, Machine | Trigger Data 2, the address for comparison |
| 0x7A3 | tdata3 | Debug, Machine | Trigger Data 3 |

Table 121: Debug and Trigger Registers

12.2.1 Debug Control and Status Register (dcsr)

This register gives information about debug capabilities and status. Its detailed functionality is described in *The RISC-V Debug Specification, Version 1.0*.

| Debug Control and Status Register (dcsr) | | | |
|--|------------|-------|---|
| CSR | 0x7B0 | | |
| Bits | Field Name | Attr. | Description |
| [1:0] | prv | RW | Privilege level of processor prior to debug exception and to return to at dret. |
| 2 | step | RW | Set to 0x1 to single-step. |
| 3 | nmip | RO | Non-maskable interrupt pending. Not used by SiFive. |
| 4 | mprven | WARL | Not used by SiFive. |
| 5 | Reserved | | |
| [8:6] | cause | RO | Indicates cause of most recent debug exception. |
| 9 | stoptime | WARL | 0x1 will stop timers in debug mode. Not used by SiFive (timers continue). |
| 10 | stopcount | WARL | 0x1 will stop counters in debug mode. Not used by SiFive (counters continue). |
| 11 | stepie | WARL | Enable interrupts when stepping. Not used by SiFive (interrupts disabled). |
| 12 | ebreaku | RW | EBREAK instructions in U-mode enter debug mode (vs. breakpoint exception). |
| 13 | ebreaks | RW | EBREAK instructions in S-mode enter debug mode. |
| 14 | Reserved | | |
| 15 | ebreakm | RW | EBREAK instructions in M-mode enter debug mode. |
| [27:16] | Reserved | | |
| [31:28] | xdebugver | RO | Version |

Table 122: Debug Control and Status Register

12.2.2 Debug PC (dpc)

When entering debug mode, the current PC is copied here. When leaving debug mode, execution resumes at this PC.

12.2.3 Debug Scratch (dscratch)

This register is generally reserved for use by Debug ROM in order to save registers needed by the code in Debug ROM. The debugger may use it as described in *The RISC-V Debug Specification, Version 1.0*.

12.2.4 Trigger Select Register (tselect)

To support a large and variable number of triggers for tracing and breakpoints, they are accessed through one level of indirection where the `tselect` register selects which bank of three `tdata1-3` registers are accessed via the other three addresses.

The `tselect` register has the format shown below:

| Trigger Select Register (tselect) | | | |
|-----------------------------------|------------|-------|-----------------------------|
| CSR | 0x7A0 | | |
| Bits | Field Name | Attr. | Description |
| [63:0] | index | WARL | Selection index of triggers |

Table 123: Trigger Select Register

The `index` field is a **WARL** field that does not hold indices of unimplemented triggers. Even if `index` can hold a trigger index, it does not guarantee the trigger exists. The `type` field of `tdata1` must be inspected to determine whether the trigger exists.

12.2.5 Trigger Data Registers (tdata1-3)

The `tdata1-3` registers are 64-bit read/write registers selected from a larger underlying bank of triggers by the `tselect` register.

| Trigger Data Register 1 (tdata1) | | | |
|----------------------------------|-----------------------|-------|---|
| CSR | 0x7A1 | | |
| Bits | Field Name | Attr. | Description |
| [58:0] | Trigger-Specific Data | | |
| 59 | dmode | WARL | Selects between debug mode (dmode=1) and machine mode (dmode=0) views of the registers, where only debug mode code can access the debug mode view of the triggers |
| [63:60] | type | WARL | The type of trigger selected by <code>tselect</code> <ul style="list-style-type: none"> • 0x0 - No such trigger • 0x1-0x5 - Reserved • 0x6 - Address/Data Match Trigger • ≥0x7 - Reserved |

Table 124: Trigger Data Register 1

| Trigger Data Registers 2 and 3 (tdata2/3) | | | |
|---|-----------------------|-------|-------------|
| CSR | 0x7A2 - 0x7A3 | | |
| Bits | Field Name | Attr. | Description |
| [63:0] | Trigger-Specific Data | | |

Table 125: Trigger Data Registers 2 and 3

Any attempt to read/write the tdata1-3 registers in machine mode when TSELECT.dmode=1 raises an illegal-instruction exception.

12.3 Breakpoints

The S76 Core Complex supports four hardware breakpoint registers per hart, which can be flexibly shared between debug mode and machine mode.

When a breakpoint register is selected with tselect, the other CSRs access the following information for the selected breakpoint:

| CSR Name | Breakpoint Alias | Description |
|----------|------------------|----------------------------|
| tselect | tselect | Breakpoint selection index |
| tdata1 | mcontrol6 | Breakpoint match control |
| tdata2 | maddress | Breakpoint match address |
| tdata3 | N/A | Reserved |

Table 126: Trigger CSRs When Used as Breakpoints

12.3.1 Breakpoint Match Control Register (mcontrol6)

Each breakpoint control register is a read/write register laid out in Table 127. This register is accessible as tdata1 when type is 0x2.

| Breakpoint Match Control Register (mcontrol16) | | | | |
|--|------------|-------|------|--|
| CSR | 0x7A1 | | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | R | WARL | 0x0 | Address match on load |
| 1 | W | WARL | 0x0 | Address match on store |
| 2 | X | WARL | 0x0 | Address match on instruction fetch |
| 3 | U | WARL | X | Address match on user mode |
| 4 | S | WARL | X | Address match on supervisor mode |
| 5 | Reserved | WPRI | X | |
| 6 | M | WARL | X | Address match on machine mode |
| [10:7] | match | WARL | X | Breakpoint Address Match type <ul style="list-style-type: none"> • 0x0 - Single address • 0x1 - Power-of-2 range, limited to 64 bytes in SiFive implementations • 0x2 - ≥ address • 0x3 - < address • Others not supported by SiFive |
| 11 | chain | WARL | 0x0 | Chain adjacent conditions. When set, this trigger and the next must match at the same time to fire. Typically used for a range breakpoint using 2 triggers, one with match=0x2 and one with match=0x3. This is not a sequential trigger. |
| [15:12] | action | WARL | 0x0 | Breakpoint action to take |
| [19:16] | size | WARL | 0x0 | Size of the breakpoint. Fixed at 0, meaning accesses of any size that cover any part of the trigger address range will fire. |
| 20 | timing | WARL | 0x0 | Timing of the breakpoint. Fixed at 0, meaning breaks happen just before the event. |
| 21 | select | WARL | 0x0 | Perform match on address or data. Fixed at 0, meaning all triggers compare addresses only (no data value). |
| [58:22] | Reserved | | | |
| 59 | dmode | RW | 0x0 | Debug-only access mode |
| [63:60] | type | RO | 0x6 | Address/Data match type, always 0x6 |

Table 127: Breakpoint Match Control Register

The type field is a 4-bit read-only field holding the value 0x2 to indicate this is a breakpoint containing address match logic.

The action field is a 4-bit read-write **WARL** field that specifies the available actions when the address match is successful. The value 0 generates a breakpoint exception. The value 1 enters debug mode. Other actions are not implemented.

The R/W/X bits are individual **WARL** fields, and if set, indicate an address match should only be successful for loads, stores, and instruction fetches, respectively. All combinations of implemented bits must be supported.

The M/S/U bits are individual **WARL** fields, and if set, indicate that an address match should only be successful in the machine, supervisor and user modes, respectively. All combinations of implemented bits must be supported.

The match field is a 4-bit read-write **WARL** field that encodes the type of address range for breakpoint address matching. Three different match settings are currently supported: exact, NAPOT, and arbitrary range. A single breakpoint register supports both exact address matches and matches with address ranges that are naturally aligned powers-of-two (NAPOT) in size. Breakpoint registers can be paired to specify arbitrary exact ranges, with the lower-numbered breakpoint register giving the byte address at the bottom of the range and the higher-numbered breakpoint register giving the address 1 byte above the breakpoint range and using the chain bit to indicate both must match for the action to be taken.

NAPOT ranges make use of low-order bits of the associated breakpoint address register to encode the size of the range as follows:

| address | Match Type and Size |
|------------|----------------------------|
| a...aaaaaa | Exactly 1 byte |
| a...aaaaa0 | 2-byte NAPOT range |
| a...aaaa01 | 4-byte NAPOT range |
| a...aaa011 | 8-byte NAPOT range |
| a...aa0111 | 16-byte NAPOT range |
| a...a01111 | 32-byte NAPOT range |
| | ... |
| a01...1111 | 2^{31} -byte NAPOT range |

Table 128: NAPOT Size Encoding

maskmax6 determines the largest supported NAPOT range. The value of maskmax6 is the logarithm base 2 of the number of bytes in the largest supported NAPOT range. A value of 0 indicates that only exact address matches are supported (1-byte range). A value of 31 corresponds to the maximum NAPOT range, which is 2^{31} bytes in size. The largest range is encoded in address with the 30 least-significant bits set to 1, bit 30 set to 0, and bit 31 holding the only address bit considered in the address comparison.

The value of `maskmax6` is not directly observable, but can be determined via the following sequence:

1. Set `match` to 1 to select NAPOT mode
2. Read `match`. If the returned value is not 1, then NAPOT matching is not supported.
3. Write all ones to `tdata2`
4. Read `tdata2`. The value of `maskmax6` is one more than the index of the most-significant zero bit. For example, if the read value is `0xFFFF_FFF7`, bit 3 is zero, so `maskmax6` is 4.

To provide breakpoints on an exact range, two neighboring breakpoints can be combined with the `chain` bit. The first breakpoint can be set to match on an address using `action` of 2 (greater than or equal). The second breakpoint can be set to match on address using `action` of 3 (less than). Setting the `chain` bit on the first breakpoint prevents the second breakpoint from firing unless they both match.

Note

When a chain includes both instruction trigger and data address trigger, the breakpoint does not fire. To work around this limitation, set a data trigger on any access to the data item. Then, in the GDB breakpoint command script, check whether the PC is the one you want and restart if not.

12.3.2 Breakpoint Match Address Register (`maddress`)

Each breakpoint match address register is a 64-bit read/write register used to hold significant address bits for address matching and also the unary-encoded address masking information for NAPOT ranges.

12.3.3 Breakpoint Execution

Breakpoint traps are taken precisely. Implementations that emulate misaligned accesses in software will generate a breakpoint trap when either half of the emulated access falls within the address range. Implementations that support misaligned accesses in hardware must trap if any byte of an access falls within the matching range.

Debug mode breakpoint traps jump to the debug trap vector without altering machine mode registers.

Machine mode breakpoint traps jump to the exception vector with "Breakpoint" set in the `mcause` register and with `badaddr` holding the instruction or data address that caused the trap.

12.3.4 Sharing Breakpoints Between Debug and Machine Mode

When debug mode uses a breakpoint register, it is no longer visible to machine mode (that is, the `tdrtype` will be 0). Typically, a debugger will leave the breakpoints alone until it needs them, either because a user explicitly requested one or because the user is debugging code in ROM.

12.4 Debug Memory Map

This section describes the Debug Module's memory map when accessed via the regular system interconnect. The Debug Module is only accessible to debug code running in debug mode on a hart (or via a Debug Transport Module). The following addresses are offsets from the base address of the Debug Module. Note that the PMP must allow M-mode access to the Debug Module address range for debugging to be possible.

12.4.1 Debug RAM and Program Buffer (0x300–0x3FF)

The S76 Core Complex has 16 32-bit words of program buffer for the debugger to direct a hart to execute arbitrary RISC-V code. Its location in memory can be determined by executing `aiupc` instructions and storing the result into the program buffer.

The S76 Core Complex has two 32-bit words of debug data RAM. Its location can be determined by reading the `DM.hartinfo` register, as described in *The RISC-V Debug Specification, Version 1.0*. This RAM space is used to pass data for the Access Register abstract command, as described in *The RISC-V Debug Specification, Version 1.0*. The S76 Core Complex supports only general-purpose register access when harts are halted. All other commands must be implemented by executing from the debug program buffer.

In the S76 Core Complex, both the program buffer and debug data RAM are general-purpose RAM and are mapped contiguously in the Core Complex memory space. Therefore, additional data can be passed in the program buffer, and additional instructions can be stored in the debug data RAM.

Debuggers must not execute program buffer programs that access any Debug Module memory except defined program buffer and debug data addresses.

12.4.2 Debug ROM (0x800–0xFFF)

This ROM region holds the debug routines on SiFive systems. The actual total size may vary between implementations.

12.4.3 Debug Flags (0x100–0x110, 0x400–0x7FF)

The flag registers in the Debug Module are used for the Debug Module to communicate with each hart. These flags are set and read used by the debug ROM and should not be accessed by any program buffer code. The specific behavior of the flags is not further documented here.

12.4.4 Safe Address

In the S76 Core Complex, the Debug Module contains the Debug Module address range in the memory map. Memory accesses to these addresses raise access exceptions, unless the hart is in debug mode. This property allows a "safe" location for unprogrammed parts, as the default `mtvec` location is `0x0`.

12.5 Debug Module Interface

The SiFive Debug Module (DM) conforms to *The RISC-V Debug Specification, Version 1.0*. A debug probe or agent connects to the Debug Module through the Debug Module Interface (DMI). The following sections describe notable spec options used in the implementation and should be read in conjunction with *The RISC-V Debug Specification, Version 1.0*.

DMI is a simple read/write bus whose master is the DTM (if it exists, otherwise DMI passes through to customer logic) and whose slave is the Debug Module. The master sends a request to the slave and the slave responds with a response. A request is considered sent if `req_ready=1` indicating the master is sending a request and `req_valid=1` indicating the slave is accepting the request on this cycle. Similarly, the response is sent when both `resp_valid=1` indicating the slave is sending a response and `resp_ready=1` indicating the master is accepting it.

Note

It is the responsibility of the debugger to simulate virtual address accesses by accessing the page tables directly, then sending the translated physical address to hardware when doing the access.

Note

The Debug Module registers are not directly accessible from the core.

| Group | Signal | Source | Description |
|--------------|------------|--------|--|
| System | clock | system | All signals timed to this clock. With JTAG DTM, this clock is the JTAG TCK. |
| | reset | system | Synchronous reset. Generated by power-on reset circuit. |
| Request Bus | req_ready | slave | Slave ready to receive request. |
| | req_valid | master | Master's request valid. |
| | req_addr | master | Configurable width address bus. 0x7 for SiFive. |
| | req_data | master | 32-bit write data bus. |
| | req_op | master | <ul style="list-style-type: none"> • 0x0 = None • 0x1 = Read • 0x2 = Write • 0x3 = Reserved |
| Response Bus | resp_ready | master | Master is ready to receive response. |
| | resp_valid | slave | Slave response is valid. |
| | resp_data | slave | 32-bit read data bus. |
| | resp_op | slave | <ul style="list-style-type: none"> • 0x0 = Success • 0x1 = Failure • 0x2 = Not used • 0x3 = Reserved |

Table 129: Debug Module Interface Signals

12.5.1 Debug Module Status Register (dmstatus)

dmstatus holds the DM version number and other implementation information. Most importantly, it contains status bits that indicate the current state of the selected hart(s).

| Debug Module Status Register (dmstatus) | | | | |
|---|-----------------|-------|------|--|
| DMI Address | | 0x11 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [3:0] | version | RO | 0x3 | Implementation version number |
| 4 | Reserved | RO | 0x0 | |
| 5 | hasresethaltreq | RO | 0x1 | 1 if resethaltreq exists |
| [7:6] | Reserved | RO | 0x0 | |
| 8 | anyhalted | RO | 0x0 | Any currently selected hart is halted. Note that this field is volatile and may be masked in IP-XACT. |
| 9 | allhalted | RO | 0x0 | All currently selected harts are halted. Note that this field is volatile and may be masked in IP-XACT. |
| 10 | anyrunning | RO | 0x1 | Any currently selected hart is running. Note that this field is volatile and may be masked in IP-XACT. |
| 11 | allrunning | RO | 0x1 | All currently selected harts are running. Note that this field is volatile and may be masked in IP-XACT. |
| 12 | anyunavail | RO | 0x0 | Any currently selected hart is not available (i.e., is powered down). DM supports it, but not currently used by SiFive cores. Note that this field is volatile and may be masked in IP-XACT. |
| 13 | allunavail | RO | 0x0 | All currently selected harts are not available (i.e., is powered down). DM supports it, but not currently used by SiFive cores. Note that this field is volatile and may be masked in IP-XACT. |
| 14 | anynonexistent | RO | 0x0 | Any currently selected hart does not exist in the system. Note that this field is volatile and may be masked in IP-XACT. |
| 15 | allnnonexistent | RO | 0x0 | All currently selected harts do not exist in the system. Note that this field is volatile and may be masked in IP-XACT. |
| 16 | anyresumeack | RO | 0x1 | Any currently selected hart has resumed execution. Note that this field is volatile and may be masked in IP-XACT. |
| 17 | allresumeack | RO | 0x1 | All currently selected harts have resumed execution. Note that this field |

Table 130: Debug Module Status Register

| | | | | |
|---------|--------------|----|-----|---|
| | | | | is volatile and may be masked in IP-XACT. |
| 18 | anyhavereset | RO | 0x0 | Any currently selected hart has been reset, but reset has not been acknowledged. Note that this field is volatile and may be masked in IP-XACT. |
| 19 | allhavereset | RO | 0x0 | All currently selected harts have been reset, but reset has not been acknowledged. Note that this field is volatile and may be masked in IP-XACT. |
| [21:20] | Reserved | RO | 0x0 | |
| 22 | impebreak | RO | 0x0 | 1 if PROGBUF is followed by implicit EBREAK. Generally, 1 for E2 cores, 0 otherwise. |
| [31:23] | Reserved | RO | 0x0 | |

Table 130: Debug Module Status Register

12.5.2 Debug Module Control Register (`dmcontrol`)

A debugger performs most hart controls through the `dmcontrol` register.

| Debug Module Control Register (dmcontrol) | | | | |
|---|-----------------|-------|------|--|
| DMI Address | | 0x10 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | dmactive | RW | 0x0 | 0 disables the DM and sets DMI registers to their reset state, 1 puts the DM in operational mode. Drives dmactive output that could be used by a system power controller to maintain power to the DM while it is being used. When 1, dmcontrol should be read back until dmactive=1, which indicates that the Debug Module is fully operational. When 0, the DM TileLink clock is gated off to save power. |
| 1 | ndmreset | RW | 0x0 | Write 1 to reset system (assert ndreset output). Write 0 to operate normally. |
| 2 | clrresethaltreq | WO | 0x0 | Write 1 to clear the reset-halt-request bit |
| 3 | setresethaltreq | WO | 0x0 | When written to 1, the core will halt upon the next deassertion of its reset |
| [27:4] | Reserved | RW | 0x0 | |
| 28 | ackhavereset | WO | 0x0 | Write 1 to acknowledge that a reset occurred on the selected hart |
| 29 | Reserved | RO | 0x0 | |
| 30 | resumereq | WO | 0x0 | Write 1 to request selected hart to resume, cleared to 0 automatically when hart resumes |
| 31 | haltreq | RW | 0x0 | Write 1 to request selected hart to halt. Generates debug interrupt to the core. Write 0 once halted has been set by the DM. |

Table 131: Debug Module Control Register

12.5.3 Hart Info Register (hartinfo)

hartinfo contains information about the currently selected hart.

| Hart Info Register (hart.info) | | | | |
|--------------------------------|------------|-------|-------|--|
| DMI Address | | 0x12 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [11:0] | dataaddr | RO | 0x380 | Address of DATA registers in hart memory map. 0x380 for SiFive. |
| [15:12] | datasize | RO | 0x2 | Number of DATA registers. 0x1 for 32-bit, 0x2 for 64-bit SiFive cores. |
| 16 | dataaccess | RO | 0x1 | DATA registers are shadowed in the hart memory map. 1 for SiFive. |
| [19:17] | Reserved | RO | 0x0 | |
| [23:20] | nscratch | RO | 0x1 | Number of dscratch registers available for debugger. 1 for SiFive. |
| [31:24] | Reserved | RO | 0x0 | |

Table 132: Hart Info Register

12.5.4 Abstract Control and Status Register (abstractcs)

| Abstract Control and Status Register (abstractcs) | | | | |
|---|-------------|-------|------|---|
| DMI Address | | 0x16 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [3:0] | datacount | RO | 0x2 | Number of DATA registers. 0x1 for 32-bit, 0x2 for 64-bit SiFive cores. |
| [7:4] | Reserved | RO | 0x0 | |
| [10:8] | cmderr | RW1C | 0x0 | <p>Non-zero value indicates an abstract command error. Remains set until cleared by writing all ones. If set, no abstract commands are accepted.</p> <ul style="list-style-type: none"> • 0x0 - No error • 0x1 - Busy. Abstract command or register was accessed while command was running. • 0x2 - Not supported. Abstract command type not supported by hardware was attempted. • 0x3 - Exception. An exception occurred during execution of an abstract command. • 0x4 - Halt/resume. Abstract command attempted while hart was running or unavailable. • 0x5 - Bus. Bus error occurred during abstract command. Not used by SiFive. • 0x7 - Other. Abstract command failed for another reason. Not used by SiFive. <p>Note that this field is volatile and may be masked in IP-XACT.</p> |
| 11 | Reserved | RO | 0x0 | |
| 12 | busy | RO | 0x0 | Reads as 1 while Abstract command is running, 0 if not. Note that this field is volatile and may be masked in IP-XACT. |
| [23:13] | Reserved | RO | 0x0 | |
| [28:24] | progbufsize | RO | 0x10 | Number of 32-bit words in PROGBUF. S76 Core Complex has 16 words. |
| [31:29] | Reserved | RO | 0x0 | |

Table 133: Abstract Control and Status Register

12.5.5 Abstract Command Register (command)

| Abstract Command Register (command) | | | | |
|-------------------------------------|------------------|-------|--|--|
| DMI Address | | 0x17 | | |
| Bits | Field Name | Attr. | Description | |
| [15:0] | regno | RW | Select which register to read/write. SiFive only supports GPRs: 0x1000-0x101F. | |
| 16 | write | RW | 1=write register, 0=read register. Only done if transfer=1. | |
| 17 | transfer | RW | 1=do the register read/write, 0=don't. | |
| 18 | postexec | RW | 1=execute PROGBUF after the command, 0=don't. | |
| 19 | aarpostincrement | RW | Not supported by SiFive. | |
| [22:20] | aarsize | RW | 0x2, 0x3, 0x4 select 32, 64, 128 bits, respectively. | |
| 23 | Reserved | RO | | |
| [31:24] | cmdtype | RW | 0=Access Register is the only type supported by SiFive. | |

Table 134: Abstract Command Register

Note that this register is volatile and may be masked in IP-XACT.

12.5.6 Abstract Command Autoexec Register (abstractauto)

| Abstract Command Autoexec Register (abstractauto) | | | | |
|---|-----------------|-------|------|---|
| DMI Address | | 0x18 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| [11:0] | autoexecdata | RW | 0x0 | Bitmap of DATA registers [11:0]. 1 indicates DATA access initiates command. |
| [15:12] | Reserved | RO | 0x0 | |
| [31:16] | autoexecprogbuf | RW | 0x0 | Bitmap of PROGBUF words [15:0]. 1 indicates PROGBUF access initiates command. |

Table 135: Abstract Command Autoexec Register

12.5.7 Debug Module Control and Status 2 Register (dmcs2)

Table 136 describes the Debug Module Control and Status 2 Register dmcs2. If halt/resume groups are not implemented, then group will always read back as 0. The Debug Module external triggers may be allocated as needed between halt and resume groups.

| Debug Module Control and Status 2 Register (dmcs2) | | | | |
|--|------------|-------|------|--|
| DMI Address | | 0x32 | | |
| Bits | Field Name | Attr. | Rst. | Description |
| 0 | hgselect | RW | 0x0 | 0=operate on harts, 1=operate on external triggers. |
| 1 | hgwrite | WO | X | When written with 1, the selected harts or external trigger is assigned to group group. |
| [6:2] | group | RW | 0x0 | Specify the halt group or resume group number that the selected harts or external triggers will be assigned to. Note that this field is volatile and may be masked in IP-XACT. |
| [10:7] | Reserved | RO | 0x0 | |
| 11 | grouptype | RW | 0x0 | 0=operate on Halt Group configuration, 1=operate on Resume Group configuration. |
| [31:12] | Reserved | RO | 0x0 | |

Table 136: Debug Module Control and Status 2 Register

12.5.8 Abstract Commands

Abstract commands provide a debugger with a path to read and write processor state and are used for extracting and modifying processor state such as registers and memory. Register `s0` is saved by the ROM and is available for use by the abstract command code. An abstract command is started by the debugger writing to `command`. In `command`, the debugger selects whether to load/store a register, execute `PROGBUF`, or both. Only GPR register transfers are supported currently. Many aspects of Abstract Commands are optional in *The RISC-V Debug Specification, Version 1.0* and are implemented as described below.

| cmdtype | Feature | Support |
|-----------------|----------------------|--|
| Access Register | GPR registers | Access Register command, register number 0x1000 - 0x101F |
| | CSR registers | Not supported. CSRs are accessed using the Program Buffer. |
| | FPU registers | Not supported. FPU registers are accessed using the Program Buffer. |
| | Autoexec | Both autoexecprogbuf and autoexecdata are supported. |
| | Post-increment | Not supported. |
| | Core Register Access | Not supported. |
| Quick Access | | Not supported. |
| Access Memory | | Not supported. Memory access is accomplished using the Program Buffer. |

Table 137: Debug Abstract Commands

The use of abstract commands is outlined in the following example, describing how to read a word of target memory:

1. The debugger writes opcodes to PROGBUF to accomplish the desired function.
2. The debugger writes the desired memory address to DATA[0].
3. The debugger requests an abstract command specifying to load s0 from DATA[0], then execute PROGBUF. Writing to command while hart n is selected has the side effect of setting FLAGS[n].go. Writing to command also sets busy which is readable from the debugger, and indicates that an abstract command is in progress.
4. The ROM busy-wait loop being executed by hart n sees FLAGS[n].go set.
5. ROM code writes 0 to GOING which has the effect of clearing FLAGS[n].go.
6. ROM code jumps to WHERETO, then ABSTRACT which contains the opcode lw s0, 0(DATA) to load s0 from DATA[0]. Opcodes in ABSTRACT are constructed by DM hardware from command. If command.transfer=0, no register transfer is done and instead ABSTRACT[0] reads as NOP.
7. If a register read/write is all that is needed, the debugger would set command.postexec to 0. ABSTRACT[1] would then read as EBREAK.
8. If command.postexec=1, ABSTRACT[1] reads as NOP and execution falls through to PROGBUF which will have been previously written by the debugger with the opcodes lw s0, 0(s0), then sw s0, DATA(zero), then EBREAK.
9. EBREAK reenters ROM at address 0x800. ROM writes hartid to HALTED which has the side effect of clearing busy, telling the debugger that the abstract command is finished.
10. The debugger reads the result from DATA[0].

The autoexec feature of Abstract Commands is supported by SiFive hardware (and is used by OpenOCD for memory block read and write). Once an abstract command has been completed, the debugger can read or write a particular DATA or PROGBUF location to run the command again. For example, fast download can be accomplished by setting up PROGBUF for memory write, then repeatedly writing words to DATA[0]. Each write re-executes the register transfer and PROGBUF to store the word into memory. For a 32-bit block write, the abstract command would be set up like this:

| | |
|----------|---|
| ABSTRACT | <pre>regno=s1, write=1, transfer=1, postexec=1. DM constructs the instructions lw s1,0(DATA) // load s1 from debugger NOP // fall thru to PROGBUF</pre> |
| PROGBUF | <pre>sw s1, 0(s0) // store s1 to memory addi s0, s0, 4 // increment memory pointer ebreak // done</pre> |

Table 138: Abstract Command Example for 32-bit Block Write

12.5.9 System Bus Access

System Bus Access (SBA) provides an alternative method to access memory. SBA operation conforms to *The RISC-V Debug Specification, Version 1.0* and its description is not duplicated here. It implements a bus master that connects with the bus crossbar to allow access to the device’s physical address space without involving a hart to perform accesses. SBA is controlled from the DMI using registers in the range 0x37 - 0x3F. By default, the maximum bus width supported by SBA is 64. Comparing Program Buffer memory access and SBA:

| Program Buffer Memory Access | SBA Memory Access |
|---|-------------------------------|
| Physical Address | Physical Address |
| Subject to Physical Memory Protection (PMP) | Not subject to PMP |
| Cache coherent | Cache coherent |
| Hart must be halted | Hart may be halted or running |

Table 139: System Bus vs. Program Buffer Comparison

12.6 Debug Module Operational Sequences

The sections below describe the flow for entering into and exiting from debug mode. The user can halt and resume more than one hart at a time using the hart array mask.

12.6.1 Entering Debug Mode

To use debug mode, the DM must be enabled by writing 0x0000_0001 to dmcontrol.

The debugger can request a halt by writing 0x8000_0001 to dmcontrol to set haltreq. This generates a debug interrupt to the core.

The core enters debug mode and jumps to the debug interrupt handler located at 0x800 and serviced from the DM.

ROM code at 0x800 writes `hartid` into the `HALTED` register which has the effect of setting the halted bit for this hart. Halted bits are readable from the debugger and generally will be continually polled to check for breakpoints when a hart is running.

ROM code then busy-waits checking its hart-specific `FLAGS` register.

12.6.2 Exiting Debug Mode

The debugger writes 1 to `resumereq` in the `dmcontrol` register to restart execution. This clears `resumeack` and sets bit 1 of the `FLAGS` register for the selected hart.

The ROM busy-wait loop being executed by hart `n` sees `FLAGS[n].resume` set.

ROM code writes `hartid` to `RESUMING`, which has the effect of clearing `FLAGS[n].resume`, setting `resumeack`, and clearing `halted` for the hart.

ROM code then executes `dret` which returns to user code at the address currently in `dpc`.

The debugger sees `resumeack` and knows the resume was successful.

Appendix A

SiFive Core Complex Configuration Options

This section provides a reference of the key configuration options of the SiFive S7 Series cores and the larger Core Complex. The file `docs/core_complex_configuration.txt` lists the features and options configured in the S76 Core Complex.

A.1 S7 Series

The S7 Series comes with the following set of configuration options. Note that the configuration may be limited to a fixed set of discrete options.

Modes and ISA:

- Configurable number of Cores (1 to 8). In the case where more than one core is selected, all cores are configured the same.
- Optional support for RISC-V user mode
- Optional M, F, D, B, and Zfh extensions
 - If M extension, configurable performance (1-cycle or 4-cycle)
- Optional SiFive Custom Instruction Extension (SCIE)

On-Chip Memory:

- Optional Instruction Cache with configurable size (4 KiB to 64 KiB) and associativity (2-, 4-, or 8-way)
- Optional Instruction-Tightly Integrated Memory (ITIM) with configurable size (4 KiB to 256 KiB) and base address
- Data Tightly-Integrated Memory (DTIM) or Data Cache:
 - If DTIM, then configurable size (4 KiB to 256 KiB) and base address
 - If Data Cache, then configurable size (4 KiB to 256 KiB) and associativity (2-, 4-, 8-, or 16-way)

- Optional Data Local Store (DLS) with the following options:
 - Configurable size (4 KiB to 8 MiB)
 - Configurable base address
 - Configurable pipeline depth (0, 1, or 3 additional stages)
 - Configurable number of banks (1 to 64)
- Optional L2 Cache with the following options:
 - Configurable size (128 KiB to 4 MiB), associativity (2-, 4-, 8-, 16-, or 32-way), and banks (1, 2, or 4)
 - Configurable number of L2 Hardware Prefetcher streams (4, 8, or 16) and queue size (4, 8, 12, or 16)
 - Configurable L1 to L2 bus width (64-, 128-, or 256-bit)
- Optional Fast I/O
- Optional Address Remapper with the following options:
 - Configurable number of entries (4, 8, 16, 32, or 64)
 - "From" region with configurable size (power of 2 up to 64) and base address
 - "To" region with configurable size (power of 2 up to 64) and base address
 - Configurable remap entry size (4 B to "From" region size)
- Configurable number of MMIO registers (4 to 24)

Error Handling:

- Optional Bus-Error Unit (BEU)
- Optional ECC support

Ports:

- Optional Memory Port, System Port, Peripheral Port, Front Port, Core Local Port, and Core Local Front Port
 - Each port has a configurable base address, width (32-, 64-, or 128-bit), size (64 KiB to 2 GiB), and protocol (AHB, AHB-Lite, APB, AXI4)
 - If AXI4 protocol, configurable AXI ID width (4, 8, or 16). Front, Memory, and System Ports only.
- Optional Front Port Passthrough

Security:

- Optional Physical Memory Protection (PMP), configurable up to 16 regions

- Optional Disable Debug Input
- Optional Password-protected Debug
- Optional Public key based secure Debug
- Optional Hardware Cryptographic Accelerator (HCA) with the following options:
 - Configurable base address
 - Optional AES-128/192/256
 - Optional AES-MAC
 - Optional SHA-224/256/384/512
 - Optional True Random Number Generator (TRNG)
 - Optional Public Key Accelerator (PKA) with the following parameters:
 - Configurable PKA operation maximum width (256 or 384 bits)

SiFive Insight Debug and Trace:

- Optional Debug Module with the following options:
 - Configurable base address
 - Configurable debug interface (JTAG, cJTAG, or APB)
 - Configurable number of Hardware Breakpoints (0 to 16) and External Triggers (0 to 16)
 - Optional System Bus Access
- Configurable number of performance counters (0 to 8)
- Optional Raw Instruction Trace Port
- Optional Nexus Trace Encoder with the following options:
 - Optional Event Trace
 - Configurable Trace Encoder Format (BTM or HTM)
 - Trace Sink (SRAM, ATB Bridge, SWT, System Memory, and/or PIB)
 - If SRAM Sink, configurable Trace Buffer size (256 B to 64 KiB)
 - If PIB Sink, configurable width (1-, 2-, 3-, 5-, or 9-bit) and optional PIB clock input
 - Optional Timestamp capabilities with configurable width (40, 48, or 56 bits) and source (Bus Clock, Core Clock, or External)
 - External Trigger Inputs (0 to 8) and Outputs (0 to 8)
 - Optional Instrumentation Trace Component (ITC)
 - Optional PC Sampling

Interrupts:

- Optional Platform-Level Interrupt Controller (PLIC) with the following parameters:
 - Priority Levels (1 to 7)
 - Number of interrupts (1 to 511)
- A configurable number of Core-Local Interruptor (CLINT) interrupts (0 to 16)

Design For Test:

- Configurable SRAM user-defined inputs (0 to 1024)
- Configurable SRAM user-defined outputs (0 to 1024)
- Optional SRAM Macro Extraction
- Optional Clock Gate Extraction
- Optional Grouping and Wrapping of extracted macros

Note that the SRAM user-defined feature is mutually exclusive to the macro extraction features.

Clocks and Reset:

- Optional Clock Gating
- Configurable Reset Scheme (Synchronous, Asynchronous, Full Asynchronous)

Branch Prediction:

- Configurable Branch Prediction (Area- or Performance-Optimized)

RTL Options:

- Optional custom RTL module name prefix

WorldGuard:

- Optional WorldGuard support with the following options:
 - Configurable number of worlds (2 to 32) and base address
 - Optional, configurable WorldGuard PMPs, filters, markers, and ROM for various component

Appendix B

SiFive RISC-V Implementation Registers

This section provides a reference to the SiFive RISC-V implementation version registers `marchid` and `mimpid`.

B.1 Machine Architecture ID Register (`marchid`)

| Value | Core Generator |
|-------------|--------------------------------|
| 0x8000_0007 | 6/7/P200/X200-Series Processor |

Table 140: Core Generator Encoding of `marchid`

B.2 Machine Implementation ID Register (mimpid)

| Value | Generator Release Version |
|-------------|---|
| 0x0000_0000 | Pre-19.02 |
| 0x2019_0228 | 19.02 |
| 0x2019_0531 | 19.05 |
| 0x2019_0919 | 19.08p0p0 / 19.08.00 |
| 0x2019_1105 | 19.08p1p0 / 19.08.01.00 |
| 0x2019_1204 | 19.08p2p0 / 19.08.02.00 |
| 0x2020_0423 | 19.08p3p0 / 19.08.03.00 |
| 0x0120_0626 | 19.08p4p0 / 19.08.04.00 |
| 0x0220_0515 | koala.00.00-preview and koala.01.00-preview |
| 0x0220_0603 | koala.02.00-preview |
| 0x0220_0630 | 20G1.03.00 / koala.03.00-general |
| 0x0220_0710 | 20G1.04.00 / koala.04.00-general |
| 0x0220_0826 | 20G1.05.00 / koala.05.00-general |
| 0x0320_0908 | kiwi.00.00-preview |
| 0x0220_1013 | 20G1.06.00 / koala.06.00-general |
| 0x0220_1120 | 20G1.07.00 / koala.07.00-general |
| 0x0421_0205 | llama.00.00-preview |
| 0x0421_0324 | 21G1.01.00 / llama.01.00-general |
| 0x0421_0427 | 21G1.02.00 / llama.02.00-general |
| 0x0521_0528 | mongoose.00.00-preview |
| 0x0521_0714 | 21G2.01.00 / mongoose.01.00-general |
| 0x0521_1008 | 21G2.02.00 / mongoose.02.00-general |
| 0x0621_1027 | narwhal.00.00-preview |
| 0x0621_1203 | narwhal.01.00-preview |
| 0x0621_1222 | 21G3.02.00 / narwhal.02.00-general |

Table 141: Generator Release Encoding of mimpid

Appendix C

SiFive Custom CSRs

This section provides a reference for the custom RISC-V CSRs configured in the S76 Core Complex.

| CSR | Name | Notes |
|-------|----------------------------|---|
| 0x7C0 | Branch Prediction Mode CSR | See Section 6.2.1 for more information |
| 0x7C1 | SiFive Feature Disable CSR | See Section 6.2.2 for more information |
| 0x7C8 | Power Dial CSR | See Section 11.2.1 for more information |

Table 142: SiFive Custom CSRs

Appendix D

Floating-Point Unit Instruction Timing

This section provides a reference for the instruction timings of the single- and double-precision floating-point unit in the S76 Core Complex.

D.1 S7 Floating-Point Instruction Timing

Single-precision floating-point unit instruction latency and repeat rates are described in Table 143.

| Assembly | Operation | Latency | Repeat Rate |
|----------------------------|--|---------|-------------|
| Sign Inject | | | |
| fabs.s rd, rs1 | $f[rd] = f[rs1] $ | 2 | 1 |
| fsgnj.s rd, rs1, rs2 | $f[rd] = \{f[rs2][31], f[rs1][30:0]\}$ | 2 | 1 |
| fsgnjn.s rd, rs1, rs2 | $f[rd] = \{-f[rs2][31], f[rs1][30:0]\}$ | 2 | 1 |
| fsgnjx.s rd, rs1, rs2 | $f[rd] = \{f[rs1][31] \wedge f[rs2][31], f[rs1][30:0]\}$ | 2 | 1 |
| Arithmetic | | | |
| fadd.s rd, rs1, rs2 | $f[rd] = f[rs1] + f[rs2]$ | 5 | 1 |
| fsub.s rd, rs1, rs2 | $f[rd] = f[rs1] - f[rs2]$ | 5 | 1 |
| fdiv.s rd, rs1, rs2 | $f[rd] = f[rs1] \div f[rs2]$ | 9–36 | 8–33 |
| fmul.s rd, rs1, rs2 | $f[rd] = f[rs1] \times f[rs2]$ | 5 | 1 |
| fsqrt.s rd, rs1 | $f[rd] = \sqrt{f[rs1]}$ | 9–28 | 8–33 |
| fmadd.s rd, rs1, rs2, rs3 | $f[rd] = (f[rs1] \times f[rs2]) + f[rs3]$ | 5 | 1 |
| fmsub.s rd, rs1, rs2, rs3 | $f[rd] = (f[rs1] \times f[rs2]) - f[rs3]$ | 5 | 1 |
| Negate Arithmetic | | | |
| fneg.s rd, rs1 | $f[rd] = -f[rs1]$ | 2 | 1 |
| fnmadd.s rd, rs1, rs2, rs3 | $f[rd] = -(f[rs1] \times f[rs2]) - f[rs3]$ | 5 | 1 |
| fnmsub.s rd, rs1, rs2, rs3 | $f[rd] = -(f[rs1] \times f[rs2]) + f[rs3]$ | 5 | 1 |
| Compare | | | |
| feq.s rd, rs1, rs2 | $x[rd] = f[rs1] == f[rs2]$ | 4 | 1 |
| fle.s rd, rs1, rs2 | $x[rd] = f[rs1] \leq f[rs2]$ | 4 | 1 |
| flt.s rd, rs1, rs2 | $x[rd] = f[rs1] < f[rs2]$ | 4 | 1 |
| fmax.s rd, rs1, rs2 | $f[rd] = \max(f[rs1], f[rs2])$ | 2 | 1 |
| fmin.s rd, rs1, rs2 | $f[rd] = \min(f[rs1], f[rs2])$ | 2 | 1 |
| Categorize | | | |
| fclass.s rd, rs1 | $x[rd] = \text{classify}_s(f[rs1])$ | 4 | 1 |
| Convert Data Type | | | |
| fcvt.w.s rd, rs1 | $x[rd] = \text{sext}(s32_{f32}(f[rs1]))$ | 4 | 1 |
| fcvt.l.s rd, rs1 | $x[rd] = s64_{f32}(f[rs1])$ | N/A | N/A |
| fcvt.s.w rd, rs1 | $f[rd] = f32_{s32}(x[rs1])$ | 2 | 1 |
| fcvt.s.l rd, rs1 | $f[rd] = f32_{s64}(x[rs1])$ | N/A | N/A |
| fcvt.wu.s rd, rs1 | $x[rd] = \text{sext}(u32_{f32}(f[rs1]))$ | 4 | 1 |
| fcvt.lu.s rd, rs1 | $x[rd] = u64_{f32}(f[rs1])$ | N/A | N/A |
| fcvt.s.wu rd, rs1 | $f[rd] = f32_{u32}(x[rs1])$ | 2 | 1 |
| fcvt.s.lu rd, rs1 | $f[rd] = f32_{u64}(x[rs1])$ | N/A | N/A |
| Move | | | |
| fmv.s rd, rs1 | $f[rd] = f[rs1]$ | 2 | 1 |
| fmv.w.x rd, rs1 | $f[rd] = x[rs1][31:0]$ | 1 | 1 |
| fmv.x.w rd, rs1 | $x[rd] = \text{sext}(f[rs1][31:0])$ | 1 | 1 |
| Load/Store | | | |
| flw rd, offset(rs1) | $f[rd] = M[x[rs1] + \text{sext}(\text{offset})][31:0]$ | 1 | 1 |
| fsw rs2, offset(rs1) | $M[x[rs1] + \text{sext}(\text{offset})] = f[rs2][31:0]$ | 1 | 1 |

Table 143: S7 Single-Precision FPU Instruction Latency and Repeat Rates

Double-precision floating-point unit latency and repeat rates are described in Table 144.

| Assembly | Operation | Latency | Repeat Rate |
|----------------------------|--|---------|-------------|
| Sign Inject | | | |
| fabs.d rd, rs1 | $f[rd] = f[rs1] $ | 2 | 1 |
| fsgnj.d rd, rs1, rs2 | $f[rd] = \{f[rs2][63], f[rs1][62:0]\}$ | 2 | 1 |
| fsgnjn.d rd, rs1, rs2 | $f[rd] = \{-f[rs2][63], f[rs1][62:0]\}$ | 2 | 1 |
| fsgnjx.d rd, rs1, rs2 | $f[rd] = \{f[rs1][63] \wedge f[rs2][63], f[rs1][62:0]\}$ | 2 | 1 |
| Arithmetic | | | |
| fadd.d rd, rs1, rs2 | $f[rd] = f[rs1] + f[rs2]$ | 7 | 1 |
| fsub.d rd, rs1, rs2 | $f[rd] = f[rs1] - f[rs2]$ | 7 | 1 |
| fdiv.d rd, rs1, rs2 | $f[rd] = f[rs1] \div f[rs2]$ | 9–58 | 8–58 |
| fmul.d rd, rs1, rs2 | $f[rd] = f[rs1] \times f[rs2]$ | 7 | 1 |
| fsqrt.d rd, rs1 | $f[rd] = \sqrt{f[rs1]}$ | 9–57 | 8–58 |
| fmadd.d rd, rs1, rs2, rs3 | $f[rd] = (f[rs1] \times f[rs2]) + f[rs3]$ | 7 | 1 |
| fmsub.d rd, rs1, rs2, rs3 | $f[rd] = (f[rs1] \times f[rs2]) - f[rs3]$ | 7 | 1 |
| Negate Arithmetic | | | |
| fneg.d rd, rs1 | $f[rd] = -f[rs1]$ | 2 | 1 |
| fnmadd.d rd, rs1, rs2, rs3 | $f[rd] = -(f[rs1] \times f[rs2]) - f[rs3]$ | 7 | 1 |
| fnmsub.d rd, rs1, rs2, rs3 | $f[rd] = -(f[rs1] \times f[rs2]) + f[rs3]$ | 7 | 1 |
| Compare | | | |
| feq.d rd, rs1, rs2 | $x[rd] = f[rs1] == f[rs2]$ | 4 | 1 |
| fle.d rd, rs1, rs2 | $x[rd] = f[rs1] \leq f[rs2]$ | 4 | 1 |
| flt.d rd, rs1, rs2 | $x[rd] = f[rs1] < f[rs2]$ | 4 | 1 |
| fmax.d rd, rs1, rs2 | $f[rd] = \max(f[rs1], f[rs2])$ | 2 | 1 |
| fmin.d rd, rs1, rs2 | $f[rd] = \min(f[rs1], f[rs2])$ | 2 | 1 |
| Categorize | | | |
| fclass.d rd, rs1 | $x[rd] = \text{classify}_d(f[rs1])$ | 4 | 1 |
| Convert Data Type | | | |
| fcvt.w.d rd, rs1 | $x[rd] = \text{sext}(s32_{f64}(f[rs1]))$ | 4 | 1 |
| fcvt.l.d rd, rs1 | $x[rd] = s64_{f64}(f[rs1])$ | N/A | N/A |
| fcvt.d.w rd, rs1 | $f[rd] = f64_{s32}(x[rs1])$ | 2 | 1 |
| fcvt.d.l rd, rs1 | $f[rd] = f64_{s64}(x[rs1])$ | N/A | N/A |
| fcvt.wu.d rd, rs1 | $x[rd] = \text{sext}(u32_{f64}(f[rs1]))$ | 4 | 1 |
| fcvt.lu.d rd, rs1 | $x[rd] = u64_{f64}(f[rs1])$ | N/A | N/A |
| fcvt.d.wu rd, rs1 | $f[rd] = f64_{u32}(x[rs1])$ | 2 | 1 |
| fcvt.d.lu rd, rs1 | $f[rd] = f64_{u64}(x[rs1])$ | N/A | N/A |
| fcvt.s.d rd, rs1 | $f[rd] = f32_{f64}(f[rs1])$ | 2 | 1 |
| fcvt.d.s rd, rs1 | $f[rd] = f64_{f32}(f[rs1])$ | 2 | 1 |
| Move | | | |
| fmv.d rd, rs1 | $f[rd] = f[rs1]$ | 2 | 1 |
| fmv.d.x rd, rs1 | $f[rd] = x[rs1][63:0]$ | N/A | N/A |
| fmv.x.d rd, rs1 | $x[rd] = f[rs1][63:0]$ | N/A | N/A |
| Load/Store | | | |
| fld rd, offset(rs1) | $f[rd] = M[x[rs1] + \text{sext}(\text{offset})][63:0]$ | 1 | 1 |
| fsd rs2, offset(rs1) | $M[x[rs1] + \text{sext}(\text{offset})] = f[rs2][63:0]$ | 1 | 1 |

Table 144: S7 Double-Precision FPU Instruction Latency and Repeat Rates

*Instruction and data are in the ITIM and DTIM, respectively.

Appendix E

Revision History

This section describes the changes in this document between release versions.

| Version | Date | Document Changes |
|------------|-------------------|---|
| 21G3.02.00 | December 22, 2021 | <ul style="list-style-type: none">Initial release |

Table 145: S76 Core Complex Manual Revision History

Appendix F

Knowledge Base Articles

The SiFive support team provides access to an index of Knowledge Base articles in order to further assist users developing with and integrating their RISC-V IP. These articles focus on topics ranging from architectural design and software development to board bring-up and implementation.

To view more than 100 articles in the SiFive Knowledge Base, access the link below:

<https://sifive.atlassian.net/servicedesk/customer/portal/47/article/465732086>

References

Visit the SiFive forums for support and answers to frequently asked questions:
<https://forums.sifive.com>

[1] A. Waterman and K. Asanovic, Eds., The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2, June 2019. [Online]. Available: <https://riscv.org/specifications/>

[2] ———, The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Version 1.11, June 2019. [Online]. Available: <https://riscv.org/specifications/privileged-isa/>

[3] SiFive TileLink Specification Version 1.8.0, August 2019. [Online]. Available: <https://sifive.com/documentation/tilelink/tilelink-spec>

[4] A. Chang, D. Barbier, and P. Dabbelt, RISC-V Platform-Level Interrupt Controller (PLIC) Specification. [Online]. Available: <https://github.com/riscv/riscv-plic-spec>

[5] E. Edgar and T. Newsome, Eds., RISC-V Debug Support, Version 1.0, May 2021. [Online]. Available: <https://github.com/riscv/riscv-debug-spec>

[6] RISC-V Bit-Manipulation ISA-extensions, Version 1.0, June 2021. [Online]. Available: <https://github.com/riscv/riscv-bitmanip/releases/tag/1.0.0>