Slicudis. RISC. Machine. Manual

ISA version 5.4

Written by Santiago Licudis

Contents

Recent changelog 2
Introduction 3
Syntax specifications 3
Instruction Set naming convention 4
SRM32 Base instruction set 5
Formats 5
Arithmetic instructions 5
Arithmetic instructions with immediate values 6
LUI (Load upper immediate) 8
Memory storing 8
Memory loading 9
JAL (Jump and link) 9
JALR (Jump to register and link) 10
Conditional jumping 10
Pseudo-instructions 11
Macro-instructions 13
Registers 16
Register file 16
Special purpose internal registers 17
Standard SRM32 extensions 18
Base system extension 18
Register reduction extension (H) 22
Multiplication/Division extension (M) 22
Atomic extension (A) 24
SRM64: The standard 64-bit extension 33
Base instruction set 33
Multiplication / Division extension 37
Atomic extension 39
Control and Status Registers (CSRs) 43
Machine level CSRs 44
Supervisor level CSRs 44
Privileged instructions 45
Interrupt and exception handling 46
Interrupt cause IDs 47

Recent changelog

- Reformatted the **fence** instruction from the **system extension.**
- Implemented the **fence.inst** instruction as part of the **system extension.**
- Implemented fence suffixes to the atomic instructions from the **atomic extension.**
- Re-defined and re-formatted the arithmetic-logic atomic instructions from the **atomic extension.**
- Removed the macro-instructions from the **multiplication and division extension.**

Introduction

SRM is an open source 32-bit little endian reduced instruction set architecture designed by Santiago Licudis with self-educational purposes and the potential of being used in real life applications and education in institutions.

Syntax specifications

This manual uses **System Verilog** and **C-like** syntax for the symbology, operators and concatenations.

The operand “**rd**” is defined as the destination register used by an instruction. “**rs1**” and “**rs2**” are defined as the source registers.

The operand “**imm**” is defined as an immediate value. All the immediate values are **sign-extended** unless it’s specified that it’s unsigned in the instruction definitions / notes.

The operand “**opcode**” indicates the main opcode of an instruction.

The operands “**fn4**” and “**fn7**” are **secondary opcodes** used by the instructions.

The symbol “$” is the position of the instruction/label that uses it.

Instruction Set naming convention

A specific syntax system is used for naming variations of the instruction set. The names start with “SRM”. Then the register size is specified (Example: SRM32). After that the extensions are specified (Example: SRM32SMA = 32-bit SRM with the Multiplication/Division and Atomic extensions)

Instruction extensions:

H - Register reduction extension
S - System extension
M - Multiplication and division extension
A - Atomic extension
F- Single precision Floating point extension (Work in progress)
D - Double precision Floating point extension (Work in progress)
L - Hypervisor extension (Work in progress)
V - Vectorial extension (Work in progress)
Zvm - Protected/Virtual memory extension (Work in progress)
_(Custom) - Custom extensions

Bit size extensions:

SRM32 - Standard 32-bit SRM
SRM64 - 64-bit extension

Examples:

Useful for small microcontrollers: SRM32H
Useful for a graphing calculator: SRM32MF
Technically the most advanced SRM CPU: SRM64SMAFDLZvm

SRM32 Base instruction set

Formats

<table>
<tr><th>FMT</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>DI</td><td colspan="21">imm [20:0]</td><td colspan="5">rd</td><td colspan="6">opcode</td></tr>
<tr><td>DSS</td><td colspan="7">fn7</td><td colspan="5">rs2</td><td colspan="4">fn4</td><td colspan="5">rs1</td><td colspan="5">rd</td><td colspan="6">opcode</td></tr>
<tr><td>DSI</td><td colspan="12">imm [11:0]</td><td colspan="4">fn4</td><td colspan="5">rs1</td><td colspan="5">rd</td><td colspan="6">opcode</td></tr>
<tr><td>SSI</td><td colspan="7">imm [6:0]</td><td colspan="5">rs2</td><td colspan="4">fn4</td><td colspan="5">rs1</td><td colspan="5">imm [11:7]</td><td colspan="6">opcode</td></tr>
</table>

Arithmetic instructions

Opcode: 0x0 / 0b000000
Format: DSS
FN7: 0x0

ADD: (Add)
Description: Adds registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: add {rd}, {rs1}, {rs2}
FN4: 0x0

SUB: (Subtract)
Description: Subtracts register **rs1** by register **rs2**, and stores the result in register **rd**.
Syntax: sub {rd}, {rs1}, {rs2}
FN4: 0x1

AND: (Bitwise AND)
Description: Bitwise AND between registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: and {rd}, {rs1}, {rs2}
FN4: 0x2

OR: (Bitwise OR)
Description: Bitwise OR between registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: or {rd}, {rs1}, {rs2}
FN4: 0x3

XOR: (Bitwise XOR)
Description: Bitwise XOR between registers **rs1** and **rs2**, and stores the result in register **rd**.
Syntax: xor {rd}, {rs1}, {rs2}
FN4: 0x4

SHR: (Logical right shift)
Description: Shifts register **rs1** by register **rs2** to the right, and stores the result in register **rd**.
Syntax: shr {rd}, {rs1}, {rs2}
FN4: 0x5

ASR: (Arithmetic right shift)
Description: Shifts register **rs1** by register **rs2** to the right. The MSBs are set to the sign of register **rs1**. The result is stored in register **rd**.
Syntax: asr {rd}, {rs1}, {rs2}
FN4: 0x6

SHL: (Logical left shift)
Description: Shifts register **rs1** by register **rs2** to the left, and stores the result in register **rd**.
Syntax: shl {rd}, {rs1}, {rs2}
FN4: 0x7

CCH: (Addition carry check)
Description: Sets register **rd** to bit 32 of an addition between register **rs1** and register **rs2**
Syntax: cch {rd}, {rs1}, {rs2}
FN4: 0x8

BCH: (Subtraction borrow check)
Description: Sets register **rd** to bit 32 of a subtraction between register **rs1** and register **rs2**
Syntax: bch {rd}, {rs1}, {rs2}
FN4: 0x9

SLT: (Set if less than)
Description: if register **rs1** is less than register **rs2** in the context of signed values, set register **rd** to 1. Else, set register **rd** to 0
Syntax: sslt {rd}, {rs1}, {rs2}
FN4: 0xA

Arithmetic instructions with immediate values

Opcode: 0x1 / 0b000001
Format: DSI

ADDI: (Add immediate)
Description: Adds registers **rs1** by **imm**, and stores the result in register **rd**.
Syntax: addi {rd}, {rs1}, {imm}
FN4: 0x0

ANDI: (Bitwise AND immediate)
Description: Bitwise AND between register **rs1** and **imm**, and stores the result in register **rd**.
Syntax: andi {rd}, {rs1}, {imm}
FN4: 0x2

ORI: (Bitwise OR immediate)
Description: Bitwise OR between register **rs1** and **imm**, and stores the result in register **rd**.
Syntax: ori {rd}, {rs1}, {imm}
FN4: 0x3

XORI: (Bitwise XOR immediate)
Description: Bitwise XOR between register **rs1** and **imm**, and stores the result in register **rd**.
Syntax: xori {rd}, {rs1}, {imm}
FN4: 0x4

SHRI: (Logical right shift immediate)
Description: Shifts register **rs1** by **imm** to the right, and stores the result in register **rd**.
Syntax: shri {rd}, {rs1}, {imm}
FN4: 0x5

ASRI: (Arithmetic right shift immediate)
Description: Shifts register **rs1** by **imm** to the right. The MSBs are set to the sign of register **rs1**. The result is stored in register **rd**.
Syntax: asri {rd}, {rs1}, {imm}
FN4: 0x6

SHLI: (Logical left shift immediate)
Description: Shifts register **rs1** by **imm** to the left, and stores the result in register **rd**.
Syntax: shli {rd}, {rs1}, {imm}
FN4: 0x7

CCHI: (Addition carry check immediate)
Description: Sets register **rd** to bit 32 of an addition between register **rs1** and register **imm**.
Syntax: cchi {rd}, {rs1}, {imm}
FN4: 0x8

BCHI: (Subtraction borrow check immediate)
Description: Sets register **rd** to bit 32 of a subtraction between register **rs1** and register **imm.**
Syntax: bchi {rd}, {rs1}, {imm}
FN4: 0x9

SLTI: (Set if less than immediate)
Description: if register **rs1** is less than **imm** in the context of signed values, set register **rd** to 1. Else, set register **rd** to 0
Syntax: sslt {rd}, {rs1}, {imm}
FN4: 0xA

LUI (Load upper immediate)

Opcode: 0x2 / 0b000010
Format: DI
Description: Register **rd** is set to (**imm** << 11).
Syntax: lui {rd}, {imm}

Memory storing

Opcode: 0x3 / 0b000011
Format: SSI
Note: The data format is big endian.

STB: (Store byte)
Description: memory**[rs1** + **imm**] = **rs2**[7:0].
Syntax: stb {rs2}, [{rs1}, {imm}]
FN4: 0x0

STW: (Store word)
Description: memory**[rs1** + **imm**] = **rs2**[15:0]. Address[0] is padded with 0.
Syntax: stw {rs2}, [{rs1}, {imm}]
FN4: 0x1

STD: (Store dword)
Description: memory**[rs1** + **imm**] = **rs2**[31:0]. Address[1:0] is padded with 0s.
Syntax: std {rs2}, [{rs1}, {imm}]
FN4: 0x2

Memory loading

Opcode: 0x4 / 0b000100
Format: DSI
Note: The data format is big endian.

LDB: (Load byte)
Description: **rd** = memory[**rs1** + **imm**][7:0]. Zero-extends
Syntax: ldb {rd}, [{rs1}, {imm}]
FN4: 0x0

LDSB: (Load signed byte)
Description: **rd** = memory[**rs1** + **imm**][7:0]. Sign-extends.
Syntax: ldsb {rd}, [{rs1}, {imm}]
FN4: 0x4

LDW: (Load word)
Description: **rd** = memory[**rs1** + **imm**][15:0]. Zero-extends and address[0] is padded with 0.
Syntax: ldw {rd}, [{rs1}, {imm}]
FN4: 0x1

LDSW: (Load signed word)
Description: **rd** = memory[**rs1** + **imm**][15:0]. Sign-extends and address[0] is padded with 0.
Syntax: ldsw {rd}, [{rs1}, {imm}]
FN4: 0x5

LDD: (Load dword)
Description: **rd** = memory[**rs1** + **imm**][31:0]. Zero-extends and address[1:0] is padded with 0s.
Syntax: ldw {rd}, [{rs1}, {imm}]
FN4: 0x2

JAL (Jump and link)

Opcode: 0x5 / 0b000101
Format: DI
Description: IP is set to (IP + **imm**) and the old value of IP+4 is stored to register **rd**.
Syntax: jal {rd}, {label} (**imm** is the relative position between IP and the label's address).

JALR (Jump to register and link)

Opcode: 0x6 / 0b000110
Format: DSI
FN4: 0x0
Description: IP is set to (**rs1** + **imm**) and the old value of IP+4 is stored to register **rd**.
Syntax: jalr {rd}, {rs1}, {imm}

Conditional jumping

Opcode: 0x7 / 0b000111
Format: SSI

JEQ: (Jump if equal)
Description: IP is set to (IP + **imm**) only if (**rs1** == **rs2**). If not, no operation happens.
Syntax: jeq {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x0

JLT: (Jump if less than)
Description: IP is set to (IP + **imm**) only if (**rs1** < **rs2**). If not, no operation happens.
Syntax: jlt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x1

JSLT: (Jump if signed less than)
Description: IP is set to (IP + **imm**) only if signed (**rs1** < **rs2**). If not, no operation happens.
Syntax: jslt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x2

JNE: (Jump if not equal)
Description: IP is set to (IP + **imm**) only if (**rs1** != **rs2**). If not, no operation happens.
Syntax: jeq {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x4

JGE: (Jump if greater of equal than)
Description: IP is set to (IP + **imm**) only if (**rs1** >= **rs2**). If not, no operation happens.
Syntax: jlt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x5

JSGE: (Jump if signed greater or equal than)
Description: IP is set to (IP + **imm**) only if signed (**rs1** >= **rs2)**. If not, no operation happens.
Syntax: jslt {rs1}, {rs2}, {label} (**imm** is the relative position between IP and the label's address).
FN1: 0x6

Pseudo-instructions

Pseudo-instructions are virtual instructions (used by assemblers) that can be represented by one real instruction.

NOP: (No operation)
Description: No operation. | Conversion: add zr, zr, zr | Syntax: nop

NOT: (Bitwise NOT)
Description: **rd** = ~**rs1** | Conversion: xori {rd}, {rs1}, -1| Syntax: not {rd}, {rs1}

INC: (Increment)
Description: **rd** = **rs1**++ | Conversion: addi {rd}, {rs1}, 1 | Syntax: inc {rd}, {rs1}

DEC: (Decrement)
Description: **rd** = **rs1**- - | Conversion: addi {rd}, {rs1}, -1 | Syntax: dec {rd}, {rs1}

MOV: (Move)
Description: Move **rs1** to **rd** | Conversion: add {rd}, {rs1}, zr | Syntax: mov {rd}, {rs1}

Move IP:
Description: Move IP + 4 to **rd** | Conversion: jalr zr, {rd}, $+4 | Syntax: mov {rd}, ip

RET: (Return)
Description: Return from a function call | Conversion: jalr zr, rp, 0 | Syntax: mov {rd}, ip

CLR: (Clear)
Description: Clear **rd** | Conversion: xor {rd}, {rd}, {rd} | Syntax: clr {rd}

NEG: (Negate)
Description: Negate **rd** | Conversion: sub {rd}, zr, {rd} | Syntax: clr {rd}

SLTU: (Set if unsigned less than)
Description: **rd** = (**rs1** < **rs2**) ? 1 : 0 | Conversion: bch {rd}, {rs1}, {rs2}
Syntax: sltu {rd}, {rs1}, {rs2}

JLE: (Jump if less or equal)
Description: Jump if (**rs1** <= **rs2**) | Conversion: jge {rs2}, {rs1}, {label}
Syntax: jle {rs1}, {rs2}, {label}

JGT: (Jump if greater than)
Description: Jump if (**rs1** > **rs2**) | Conversion: jlt {rs2}, {rs1}, {label}
Syntax: jgt {rs1}, {rs2}, {label}

JMP: (Jump)
Description: Jump | Conversion: jal zr, {label} | Syntax: jmp {label}

SGT: (Set if greater than)
Description: **rd** = (**rs1** > **rs2**) ? 1 : 0 | Conversion: slt {rd}, {rs2}, {rs1}
Syntax: sgt {rd}, {rs1}, {rs2}

SGTU: (Set if unsigned greater than)
Description: **rd** = (**rs1** > **rs2**) ? 1 : 0 | Conversion: bch {rd}, {rs2}, {rs1}
Syntax: ssgt {rd}, {rs1}, {rs2}

JZ: (Jump if zero)
Description: Jump if (**rs1** == 0) | Conversion: jeq {rs1}, zr, {label}
Syntax: jz {rs1}, {label}

JNZ: (Jump if not zero)
Description: Jump if (**rs1** != 0) | Conversion: jne {rs1}, zr, {label}
Syntax: jnz {rs1}, {label}

JLZ: (Jump if less than zero)
Description: Jump if (**rs1** < 0) | Conversion: jslt {rs1}, zr, {label}
Syntax: jlz {rs1}, {label}

JGZ: (Jump if greater than zero)
Description: Jump if (**rs1** > 0) | Conversion: jslt zr, {rs1}, {label}
Syntax: jgz {rs1}, {label}

LEA: (Load effective address)
Description: Load **rd** with the effective address of [**rs1**, **imm**]
Conversion: addi {rd}, {rs1}, {imm}
Syntax: lea {rd}, [{rs1}, {imm}]

Macro-instructions

Macro-instructions are virtual instructions (used by assemblers) that can be represented by one real instruction. They take more than one instruction to recreate and sometimes require conditional compilation systems, like LDI.

LDI: (Load immediate)
Description: Load **rd** with an immediate value
Syntax: ldi {rd}, {imm}
Conversion:

```
if (imm fits in 12 bits):
        place: addi {rd}, zr, {imm}
else if (imm is a multiple of 0x2000)
        place: lui {rd}, {imm}[32:12]
else:
        place: lui {rd}, {imm}[32:10]
        place addi {rd}, {rd}, {imm}[9:0]
```

PUSH: (Push)
Description: Push **rs1** onto the stack
Syntax: push {rs1}
Conversion:

```
//p is the list of push instructions in a row
//y is the number of those pushes in a row
for (x = 0; x < y; x++):
        place: std {rs1 of p[x]}, [sp, x*-4]
place: addi sp, sp, y*-4
```

POP: (Pop)
Description: Pop to **rd** from the stack
Syntax: pop {rd}
Conversion:

```
//p is the list of pop instructions in a row
//y is the number of those pushes in a row
for (x = 0; x =< y; x = x++):
        place: ldb {rd of ps[x]}, [sp, x*4]
place: addi sp, sp, y*4
```


XCHG: (Exchange)
Description: Exchange between **rs1** and **rs2**
Syntax: xchg {rs1}, {rs2}
Conversion:
xor {rs1}, {rs1}, {rs2}
xor {rs2}, {rs1}, {rs2}
xor {rs1}, {rs1}, {rs2}

LD(B/SB/W/SW/D)UPDT: (Load byte/signed byte/word/signed word/dword)
Description: Load **rd** and load **rs1** with the effective address
Syntax: ld(b/sb/w/sw/d)updt {rd}, [{rs1}, {imm}] (Example: ldbupdt s0, [t0, 1])
Conversion:
ld(b/sb/w/sw/d) {rd}, [{rs1}, {imm}]
addi {rs1}, {rs1}, {imm}

ST(B/SB/W/SW/D)UPDT: (Store byte/word/dword)
Description: Store **rs2** and load **rs1** with the effective address
Syntax: st(b/sb/w/sw/d)updt {rs2}, [{rs1}, {imm}] (Example: stwupdt s0, [t0, 1])
Conversion:
st(b/sb/w/sw/d) {rs2}, [{rs1}, {imm}]
addi {rs1}, {rs1}, {imm}

SSLT: (Set if signed less than)
Description: rd = signed (rs1 < rs2) ? 1 : 0
Syntax: sslt {rd}, {rs1}, {rs2}
Conversion:
sub {rd}, {rs1}, {rs2}
shri {rd}, {rd}, 31

BST: (Bit set)
Description: rd[imm] = 1
Syntax: bst {rd}, {imm}
Conversion:
```
if (imm < 12):
        place: ori {rd}, {rd}, 1 << {imm}
else:
        place: ldi at, 1 << {imm}
        place: or {rd}, {rd}, at
```


BCL: (Bit clear)
Description: rd[imm] = 0
Syntax: bcl {rd}, {imm}
Conversion:
```
if (imm < 12):
        place: andi {rd}, {rd}, -1*(1 << {imm})
else:
        place: ldi at, -1*(1 << {imm})
        place: and {rd}, {rd}, at
```

BFL: (Bit flip)
Description: rd[imm] = 0
Syntax: bfl {rd}, {imm}
Conversion:
```
if (imm < 12):
        place: xori {rd}, {rd}, (1 << {imm})
else:
        place: ldi at, (1 << {imm})
        place: xor{rd}, {rd}, at
```

Registers

Register file

The register files in SRM cores consist of 3 special purpose registers and 29 general purpose registers. The H extension removes r16-31.

Register	Name	Function	Saver
r0	zr	Constant 0	-
r1	at	Assembler temporary	Caller
r2	gp	Global pointer	-
r3	tp	Thread pointer	-
r4	rp	Return pointer	Callee
r5	sp	Stack pointer	Callee
r6	fp	Frame pointer	Callee
r7-9	a0-2	Function arguments/return values	Caller
r10-12	s0-2	Saved registers	Callee
r13-20	t0-7	Temporaries	Caller
r21-28	s3-8	Saved registers	Callee
r28-31	a3-a7	Function arguments	Caller

Each register has their assigned application for function calls. **ZR** is always 0 and can't be modified, **AT** is a temporary used by assemblers for macro-instructions, **RP** contains the return address for function calls, **SP** points at the top-most value of the stack, FP is used to delineate the boundary between two stack frames, **GP** is used for fast access to global variables and data structures, **TP** points at the thread local memory, **A0-5** are used as function arguments and their return values, **S0-8** are used for local variables that are expected to keep their values after function calls, and **T0-7** are used for local variables that become garbage after function calls. The conventions for r1-31 aren't enforced but are highly recommended to follow.

Special purpose internal registers

These registers are outside of the register file and are used for specific applications.

IP: Instruction pointer register; points at the memory location of the current instruction being executed and its reset value is 0x0.

CSRs: Control and Status registers; they contain control and status values/flags. They are from the S (System) extension.

Standard SRM32 extensions

These extensions expand the instruction set and the new instructions are assigned to specific opcodes to maintain global compatibility.

Base system extension

This extension implements system instructions, privilege modes (User, Kernel and Machine), control and status registers (CSRs) which are mainly for handling interrupts, and instructions for IO.

New formats:

<table>
<tr><th>FMT</th><th>31</th><th>30</th><th>29</th><th>28</th><th>27</th><th>26</th><th>25</th><th>24</th><th>23</th><th>22</th><th>21</th><th>20</th><th>19</th><th>18</th><th>17</th><th>16</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>CSB</td><td colspan="12">imm [11:0]</td><td colspan="4">fn4</td><td colspan="5">uimm[4:0]</td><td colspan="5">rd</td><td colspan="6">opcode</td></tr>
<tr><td>FNC</td><td colspan="7">fn7</td><td>-</td><td>I</td><td>O</td><td>R</td><td>W</td><td colspan="4">fn4</td><td colspan="10">reserved</td><td colspan="6">opcode</td></tr>
</table>

Opcode: 0x8 / 0b001000

SYSCALL: (System call)
Format: DSS
FN4: 0x0
FN7: 0x0
Description: Transfers control to a higher privilege level (see the **Interrupt and exception handling** section for more details).
Syntax: syscall

SYSBREAK: (System break)
Format: DSS
FN4: 0x0
FN7: 0x1
Description: Transfers control to a higher privilege level for debugging purposes (see the **Interrupt and exception handling** section for more details).
Syntax: syscall

MRET: (Machine return)
Format: DSS
FN4: 0x1
FN7: 0x3
Description: IP is set to the contents of the **mra** CSR and the privilege mode is set to the contents of **mpriv**. This instruction can only be executed on privilege level 3.
Syntax: mret

SYSRET: (System return)
Format: DSS
FN4: 0x1
FN7: 0x1
Description: IP is set to the contents of the **kra** CSR and the privilege mode is set to the contents of **kpriv**. This instruction can only be executed on privilege level 1+.
Syntax: sysret

CSRR: (CSR read)
Format: DSI
FN4: 0x2
Description: Register **rd** is set to csr[imm] if the privilege mode allows the program to write to the selected CSR (If not, the value returned is 0).
Syntax: csrr {rd}, {csr}

CSRW: (CSR write)
Format: DSI
FN4: 0x3
Description: csr[imm] is set to register **rs1** if the privilege mode allows the program to write to the selected CSR.
Syntax: csrr {rs1}, {csr}

CSRS: (CSR bit set)
Format: DSI
FN4: 0x4
Description: csr[**imm**][**rs1**] is set to 1. if the privilege mode allows the program to write/read to the selected CSR.
Syntax: csrs {csr}, {position}

CSRC: (CSR bit clear)
Format: DSI
FN4: 0x5
Description: csr[**imm**][**rs1**] is set to 0. if the privilege mode allows the program to write/read to the selected CSR.
Syntax: csrs {csr}, {position}

CSRRW: (CSR read-write)
Format: DSI
FN4: 0x6
Description: register **rd** is set to csr[**imm**] and csr[**imm**] is set to register **rs1** if the privilege mode allows the program to write/read to the selected CSR (If not, the value returned is 0).
Syntax: csrr {rs1}, {csr}

CSRRS: (CSR read-set)
Format: DSI
FN4: 0x7
Description: register **rd** is set to csr[**imm**] and csr[**imm**][**rs1**] is set to 1 if the privilege mode allows the program to write/read to the selected CSR (If not, the value returned is 0).
Syntax: csrr {rs1}, {csr}

CSRRC: (CSR read-clear)
Format: DSI
FN4: 0x8
Description: register **rd** is set to csr[**imm**] and csr[**imm**][**rs1**] is set to 0 if the privilege mode allows the program to write/read to the selected CSR (If not, the value returned is 0).
Syntax: csrr {rs1}, {csr}

CSRSI: (CSR bit set immediate)
Format: CSB
FN4: 0x9
Description: csr[**imm**][**uimm**] is set to 1. if the privilege mode allows the program to write/read to the selected CSR.
Syntax: csrs {csr}, {position}

CSRCI: (CSR bit clear immediate)
Format: CSB
FN4: 0xA
Description: csr[**imm**][**uimm**] is set to 0. if the privilege mode allows the program to write/read to the selected CSR.
Syntax: csrs {csr}, {position}

CSRRSI: (CSR read-set immediate)
Format: CSB
FN4: 0xB
Description: register **rd** is set to csr[**imm**] and csr[**imm**][**uimm**] is set to 1 if the privilege mode allows the program to write/read to the selected CSR (If not, the value returned is 0).
Syntax: csrr {rs1}, {csr}

CSRRCI: (CSR read-clear immediate)
Format: CSB
FN4: 0xC
Description: register **rd** is set to csr[**imm**] and csr[**imm**][**uimm**] is set to 0 if the privilege mode allows the program to write/read to the selected CSR (If not, the value returned is 0).
Syntax: csrr {rs1}, {csr}

IN: (Input)
Format: DSI
FN4: 0xD
Description: Register **rd** is set to the input data of port [**rs1** + imm] using big endian.
Syntax: in {rd}, [{rs1}, {imm}]

OUT: (Output)
Format: SSI
FN4: 0xE
Description: port [**rs1**] is set to the contents of register **rs2** using the big endian data format.
Syntax: out {rs2}, [{rs1}, {imm}]

FENCE: (Fence)
Format: FNC
FN4: 0xF
FN5: 0x0
Description: Enforce an ordering constraint on memory and IO operations issued before and after the fence instruction. The fence type is defined with the I, O, R, W bits,
Syntax: fence.{fence type} (Ex: fence iorw) (Ex: fence iw)
Fence type order: IORW

FENCE.INST: (Instruction fence)
Format: FNC
FN4: 0xF
FN5: 0x1
Description: Flush the internal CPU pipeline and the L1 instruction cache.
Syntax: fence.inst

Register reduction extension (H)

This extension removes registers r16-31. SRM cores without this extension are software compatible with the software for SRM CPUs with only 16 registers.

Multiplication/Division extension (M)

This extension implements support for integer multiplication, division and modulo (both signed and unsigned).

Opcode: 0x0 / 0b000000
FN7: 0x1
Format: DSS

MUL: (Multiply)
FN4: 0x0
Description: Register **rs1** is multiplied by **rs2** and the lower 32 bits of the result are stored in register **rd**.
Syntax: mul {rd}, {rs1}, {rs2}

MULH: (Multiply high)
FN4: 0x2
Description: Register **rs1** is multiplied by **rs2** and result [63:32] is stored in **rd**.
Syntax: mulh {rd}, {rs1}, {rs2}

SMUL: (Signed multiply high)
FN4: 0x3
Description: **rd** = (signed **rs1** * signed **rs2**)[63:32].
Syntax: smulh {rd}, {rs1}, {rs2}

DIV: (Divide)
FN4: 0x4
Description: Register **rs1** is divided by **rs2** and the quotient of the result is stored in **rd**.
Syntax: div {rd}, {rs1}, {rs2}

SDIV: (Signed divide)
FN4: 0x5
Description: **rd** = signed **rs1** / signed **rs2**
Syntax: sdiv {rd}, {rs1}, {rs2}

REM: (Remainder)
Description: Register **rs1** is divided by **rs2** and the remainder of the result is stored in **rd**.
Syntax: mod {rd}, {rs1}, {rs2}
FN4: 0x6

SREM: (Signed remainder)
Description: Register **rs1** is divided by **rs2** and the remainder of the result is stored in **rd**. The operation is performed in a signed context
Syntax: smod {rd}, {rs1}, {rs2}
FN4: 0x7

Atomic extension (A)

The atomic extension implements a total of 54 atomic instructions (64 in SRM64) that can be used for thread synchronization. AQ (Acquire) fences memory read instructions and RL (Release) fences memory write instructions. Its syntax is {instruction}.(aq)(rl) (ex: ll.d.aqrl). The prefix isn't used when no fence is used.

FMT	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ATM	fn5					aq	rl	rs2					fn4				rs1					rd					opcode					

<u>Opcode:</u> 0x9 / 0b001001
<u>Format:</u> ATM

Load and Link:

<u>FN7: 0x0</u>
<u>General description:</u> Set register **rd** to memory[**rs1**][bitwidth-1:0] and link memory[**rs1**]. Only one link can be done at a time (New links overwrite old links). The link will always be broken if the contents of the linked address are modified.

LL.B(.{fence}): (Load and link byte)
<u>FN4:</u> 0x0
<u>Description:</u> Set register **rd** to memory[**rs1**][7:0] and link memory[**rs1**]. Zero-extends.
<u>Syntax:</u> ll.b(.{fence}) {rd}, [{rs1}]

LL.W(.{fence}): (Load and link word)
<u>FN4:</u> 0x1
<u>Description:</u> Set register **rd** to memory[**rs1**][15:0] and link memory[**rs1**]. Zero-extends.
<u>Syntax:</u> ll.w(.{fence}) {rd}, [{rs1}]

LL.D(.{fence}): (Load and link dword)
<u>FN4:</u> 0x2
<u>Description:</u> Set register **rd** to memory[**rs1**][31:0] and link memory[**rs1**]. Zero-extends.
<u>Syntax:</u> ll.d(.{fence}) {rd}, [{rs1}]

LL.SB(.{fence}): (Load and link signed byte)
<u>FN4:</u> 0x0
<u>Description:</u> Set register **rd** to memory[**rs1**][7:0] and link memory[**rs1**]. Sign-extends.
<u>Syntax:</u> ll.sb(.{fence}) {rd}, [{rs1}]

LL.SW(.{fence}): (Load and link signed word)
<u>FN4:</u> 0x1
<u>Description:</u> Set register **rd** to memory[**rs1**][15:0] and link memory[**rs1**]. Sign-extends.
<u>Syntax:</u> ll.sw(.{fence}) {rd}, [{rs1}]

Store Conditional:

FN7: 0x1
General description: If memory[**rs1**] is reserved, store register **rs2** [bitwidth-1:0] into memory[**rs1**] and set register **rd** to 1 to indicate success. Else, only set register **rd** to 0 to indicate failure.

SC.B(.{fence}): (Store conditional byte)
FN4: 0x0
Description: Store conditional to memory[**rs1**][7:0]
Syntax: sc.b(.{fence}) {rd}, [{rs1}]

SC.W(.{fence}): (Store conditional word)
FN4: 0x1
Description: Store conditional to memory[**rs1**][15:0]
Syntax: sc.w(.{fence}) {rd}, [{rs1}]

SC.D(.{fence}): (Store conditional dword)
FN4: 0x2
Description: Store conditional to memory[**rs1**][31:0]
Syntax: sc.d(.{fence}) {rd}, [{rs1}]

Atomic compare and exchange:

FN7: 0x2
General description: If the contents of register **rd** are equal to the contents of memory[**rs1**], memory[**rs1**] is set to register **rs2**. In all cases, **rd** is set to the old contents of memory[**rs1**].

ACMPXCHG.B(.{fence}): (Atomic compare and exchange byte)
FN4: 0x0
Description: Atomic compare and exchange using register **rd** [7:0], memory[rs1][7:0] and register **rs2** [7:0]
Syntax: acmpxchg.b(.{fence}) {rd}, {rs2}, [{rs1}]

ACMPXCHG.W(.{fence}): (Atomic compare and exchange word)
FN4: 0x1
Description: Atomic compare and exchange using register **rd** [15:0], memory[rs1][15:0] and register **rs2** [15:0]
Syntax: acmpxchg.b(.{fence}) {rd}, {rs2}, [{rs1}]

ACMPXCHG.D(.{fence}): (Atomic compare and exchange dword)
FN4: 0x2
Description: Atomic compare and exchange using register **rd** [31:0], memory[rs1][31:0] and register **rs2** [31:0]
Syntax: acmpxchg.b(.{fence}) {rd}, {rs2}, [{rs1}]

Atomic exchange and add:

FN7: 0x3
General description: The addition between memory[**rs1**][bitwidth-1:0] and register **rs2** is stored into memory[**rs1**][bitwidth-1:0]. The old contents of memory[**rs1**][bitwidth-1:0] are stored into register **rd**.

AXADD.B(.{fence}): (Atomic exchange and add byte)
FN4: 0x0
Description: Atomic exchange and add using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Zero-extends for memory[**rs1**][7:0] stored into register **rd**.
Syntax: axadd.b(.{fence}) {rd}, {rs2}, [{rs1}]

AXADD.W(.{fence}): (Atomic exchange and add word)
FN4: 0x1
Description: Atomic exchange and add using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Zero-extends for memory[**rs1**][15:0] stored into register **rd**.
Syntax: axadd.w(.{fence}) {rd}, {rs2}, [{rs1}]

AXADD.D(.{fence}): (Atomic exchange and add dword)
FN4: 0x2
Description: Atomic exchange and add using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Zero-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axadd.d(.{fence}) {rd}, {rs2}, [{rs1}]

AXADD.SB(.{fence}): (Atomic exchange and add signed byte)
FN4: 0x4
Description: Atomic exchange and add using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Sign-extends for memory[**rs1**][7:0] stored into register **rd**.
Syntax: axadd.sb(.{fence}) {rd}, {rs2}, [{rs1}]

AXADD.SW(.{fence}): (Atomic exchange and add signed word)
FN4: 0x5
Description: Atomic exchange and add using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Sign-extends for memory[**rs1**][15:0] stored into register **rd**.
Syntax: axadd.sw(.{fence}) {rd}, {rs2}, [{rs1}]

Atomic exchange and subtract:

FN7: 0x4

General description: The subtraction between memory[**rs1**][bitwidth-1:0] and register **rs2** is stored into memory[**rs1**][bitwidth-1:0]. The old contents of memory[**rs1**][bitwidth-1:0] are stored into register **rd**.

AXSUB.B(.{fence}): (Atomic exchange and subtract byte)

FN4: 0x0

Description: Atomic exchange and subtract using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Zero-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axsub.b(.{fence}) {rd}, {rs2}, [{rs1}]

AXSUB.W(.{fence}): (Atomic exchange and subtract word)

FN4: 0x1

Description: Atomic exchange and subtract using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Zero-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axsub.w(.{fence}) {rd}, {rs2}, [{rs1}]

AXSUB.D(.{fence}): (Atomic exchange and subtract dword)

FN4: 0x2

Description: Atomic exchange and subtract using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Zero-extends for memory[**rs1**][31:0] stored into register **rd**.

Syntax: axsub.d(.{fence}) {rd}, {rs2}, [{rs1}]

AXSUB.SB(.{fence}): (Atomic exchange and subtract signed byte)

FN4: 0x4

Description: Atomic exchange and subtract using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Sign-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axsub.sb(.{fence}) {rd}, {rs2}, [{rs1}]

AXSUB.SW(.{fence}): (Atomic exchange and subtract signed word)

FN4: 0x5

Description: Atomic exchange and subtract using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Sign-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axsub.sw(.{fence}) {rd}, {rs2}, [{rs1}]

Atomic exchange and AND:

FN7: 0x5

General description: The result of a bitwise AND operation between memory[**rs1**][bitwidth-1:0] and register **rs2** is stored into memory[**rs1**][bitwidth-1:0]. The old contents of memory[**rs1**][bitwidth-1:0] are stored into register **rd**.

AXAND.B(.{fence}): (Atomic exchange and AND byte)

FN4: 0x0

Description: Atomic exchange and AND using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Zero-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axand.b(.{fence}) {rd}, {rs2}, [{rs1}]

AXAND.W(.{fence}): (Atomic exchange and AND word)

FN4: 0x1

Description: Atomic exchange and AND using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Zero-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axand.w(.{fence}) {rd}, {rs2}, [{rs1}]

AXAND.D(.{fence}): (Atomic exchange and AND dword)

FN4: 0x2

Description: Atomic exchange and AND using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Zero-extends for memory[**rs1**][31:0] stored into register **rd**.

Syntax: axand.d(.{fence}) {rd}, {rs2}, [{rs1}]

AXAND.SB(.{fence}): (Atomic exchange and AND signed byte)

FN4: 0x4

Description: Atomic exchange and AND using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Sign-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axand.sb(.{fence}) {rd}, {rs2}, [{rs1}]

AXAND.SW(.{fence}): (Atomic exchange and AND signed word)

FN4: 0x5

Description: Atomic exchange and AND using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Sign-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axand.sw(.{fence}) {rd}, {rs2}, [{rs1}]

Atomic exchange and OR:

FN7: 0x6

General description: The result of a bitwise OR operation between memory[**rs1**][bitwidth-1:0] and register **rs2** is stored into memory[**rs1**][bitwidth-1:0]. The old contents of memory[**rs1**][bitwidth-1:0] are stored into register **rd**.

AXIOR.B(.{fence}): (Atomic exchange and OR byte)

FN4: 0x0

Description: Atomic exchange and OR using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Zero-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axior.b(.{fence}) {rd}, {rs2}, [{rs1}]

AXIOR.W(.{fence}): (Atomic exchange and OR word)

FN4: 0x1

Description: Atomic exchange and OR using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Zero-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axior.w(.{fence}) {rd}, {rs2}, [{rs1}]

AXIOR.D(.{fence}): (Atomic exchange and OR dword)

FN4: 0x2

Description: Atomic exchange and OR using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Zero-extends for memory[**rs1**][31:0] stored into register **rd**.

Syntax: axior.d(.{fence}) {rd}, {rs2}, [{rs1}]

AXIOR.SB(.{fence}): (Atomic exchange and OR signed byte)

FN4: 0x4

Description: Atomic exchange and OR using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Sign-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axior.sb(.{fence}) {rd}, {rs2}, [{rs1}]

AXIOR.SW(.{fence}): (Atomic exchange and OR signed word)

FN4: 0x5

Description: Atomic exchange and OR using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Sign-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axior.sw(.{fence}) {rd}, {rs2}, [{rs1}]

Atomic exchange and XOR:

FN7: 0x7

General description: The result of a bitwise XOR operation between memory[**rs1**][bitwidth-1:0] and register **rs2** is stored into memory[**rs1**][bitwidth-1:0]. The old contents of memory[**rs1**][bitwidth-1:0] are stored into register **rd**.

AXXOR.B(.{fence}): (Atomic exchange and XOR byte)

FN4: 0x0

Description: Atomic exchange and XOR using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Zero-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axxor.b(.{fence}) {rd}, {rs2}, [{rs1}]

AXXOR.W(.{fence}): (Atomic exchange and XOR word)

FN4: 0x1

Description: Atomic exchange and XOR using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Zero-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axxor.w(.{fence}) {rd}, {rs2}, [{rs1}]

AXXOR.D(.{fence}): (Atomic exchange and XOR dword)

FN4: 0x2

Description: Atomic exchange and XOR using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Zero-extends for memory[**rs1**][31:0] stored into register **rd**.

Syntax: axxor.d(.{fence}) {rd}, {rs2}, [{rs1}]

AXXOR.SB(.{fence}): (Atomic exchange and XOR signed byte)

FN4: 0x4

Description: Atomic exchange and XOR using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Sign-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axxor.sb(.{fence}) {rd}, {rs2}, [{rs1}]

AXXOR.SW(.{fence}): (Atomic exchange and XOR signed word)

FN4: 0x5

Description: Atomic exchange and XOR using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Sign-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axxor.sw(.{fence}) {rd}, {rs2}, [{rs1}]

Atomic exchange and MAX:

FN7: 0x7
General description: The lowest value (in a signed context) between memory[**rs1**][bitwidth-1:0] and register **rs2** is stored into memory[**rs1**][bitwidth-1:0]. The old contents of memory[**rs1**][bitwidth-1:0] are stored into register **rd**.

AXMAX.B(.{fence}): (Atomic exchange and MAX byte)
FN4: 0x0
Description: Atomic exchange and MAX using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Zero-extends for memory[**rs1**][7:0] stored into register **rd**.
Syntax: axmax.b(.{fence}) {rd}, {rs2}, [{rs1}]

AXMAX.W(.{fence}): (Atomic exchange and MAX word)
FN4: 0x1
Description: Atomic exchange and MAX using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Zero-extends for memory[**rs1**][15:0] stored into register **rd**.
Syntax: axmax.w(.{fence}) {rd}, {rs2}, [{rs1}]

AXMAX.D(.{fence}): (Atomic exchange and MAX dword)
FN4: 0x2
Description: Atomic exchange and MAX using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Zero-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axmax.d(.{fence}) {rd}, {rs2}, [{rs1}]

AXMAX.SB(.{fence}): (Atomic exchange and MAX signed byte)
FN4: 0x4
Description: Atomic exchange and MAX using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Sign-extends for memory[**rs1**][7:0] stored into register **rd**.
Syntax: axmax.sb(.{fence}) {rd}, {rs2}, [{rs1}]

AXMAX.SW(.{fence}): (Atomic exchange and MAX signed word)
FN4: 0x5
Description: Atomic exchange and MAX using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Sign-extends for memory[**rs1**][15:0] stored into register **rd**.
Syntax: axmax.sw(.{fence}) {rd}, {rs2}, [{rs1}]

Atomic exchange and MIN:

FN7: 0x7

General description: The lowest value (in a signed context) between memory[**rs1**][bitwidth-1:0] and register **rs2** is stored into memory[**rs1**][bitwidth-1:0]. The old contents of memory[**rs1**][bitwidth-1:0] are stored into register **rd**.

AXMIN.B(.{fence}): (Atomic exchange and MIN byte)

FN4: 0x0

Description: Atomic exchange and MIN using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Zero-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axmin.b(.{fence}) {rd}, {rs2}, [{rs1}]

AXMIN.W(.{fence}): (Atomic exchange and MIN word)

FN4: 0x1

Description: Atomic exchange and MIN using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Zero-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axmin.w(.{fence}) {rd}, {rs2}, [{rs1}]

AXMIN.D(.{fence}): (Atomic exchange and MIN dword)

FN4: 0x2

Description: Atomic exchange and MIN using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Zero-extends for memory[**rs1**][31:0] stored into register **rd**.

Syntax: axmin.d(.{fence}) {rd}, {rs2}, [{rs1}]

AXMIN.SB(.{fence}): (Atomic exchange and MIN signed byte)

FN4: 0x4

Description: Atomic exchange and MIN using register **rd** [7:0], memory[rs1][7:0] and register **rs2**. Sign-extends for memory[**rs1**][7:0] stored into register **rd**.

Syntax: axmin.sb(.{fence}) {rd}, {rs2}, [{rs1}]

AXMIN.SW(.{fence}): (Atomic exchange and MIN signed word)

FN4: 0x5

Description: Atomic exchange and MIN using register **rd** [15:0], memory[rs1][15:0] and register **rs2.** Sign-extends for memory[**rs1**][15:0] stored into register **rd**.

Syntax: axmin.sw(.{fence}) {rd}, {rs2}, [{rs1}]

SRM64: The standard 64-bit extension

The 64-bit extension extends the base instruction set and the other extensions to support 64-bit operations while having backwards compatibility with SRM32 software.

Base instruction set

Arithmetic instructions from the SRM32 Base instruction set will be executed in a 32-bit context (The operation will be performed with the lower 32 bits of the registers and the result will be truncated to 32 bits). The 32-bit result will be sign-extended to 64 bits.

Memory:

STQ: (Store qword)
Opcode: 0x3 / 0b000011
FN4: 0x3
Description: memory[**rs1** + **imm**] = **rs2**[63:0].
Syntax: stq {rs2}, [{rs1}, {imm}]

LDSD: (Load signed dword)
Opcode: 0x4 / 0b000100
FN4: 0x6
Description: **rd** = memory[**rs1** + **imm**][31:0]. Sign-extends.
Syntax: ldsd {rd}, [{rs1}, {imm}]

LDQ: (Load qword)
Opcode: 0x4 / 0b000100
FN4: 0x3
Description: **rd** = memory[**rs1** + **imm**][63:0]. Zero-extends.
Syntax: ldq {rd}, [{rs1}, {imm}]

Arithmetic:

Opcode: 0x0 / 0b000000
Format: DSS
FN7: 0x2

ADD.Q: (Add qword)
Description: Adds registers **rs1** and **rs2**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: add.q {rd}, {rs1}, {rs2}
FN4: 0x0

SUB.Q: (Subtract qword)
Description: Subtracts register **rs1** by register **rs2**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: sub.d {rd}, {rs1}, {rs2}
FN4: 0x1

AND.Q: (Bitwise AND qword)
Description: Bitwise AND between registers **rs1** and **rs2**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: and.q {rd}, {rs1}, {rs2}
FN4: 0x2

OR.Q: (Bitwise OR qword)
Description: Bitwise OR between registers **rs1** and **rs2**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: or.q {rd}, {rs1}, {rs2}
FN4: 0x3

XOR.Q: (Bitwise XOR qword)
Description: Bitwise XOR between registers **rs1** and **rs2**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: xor.q {rd}, {rs1}, {rs2}
FN4: 0x4

SHR.Q: (Logical right shift qword)
Description: Shifts register **rs1** by register **rs2** to the right, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: shr.q {rd}, {rs1}, {rs2}
FN4: 0x5

ASR.Q: (Arithmetic right shift qword)
Description: Shifts register **rs1** by register **rs2** to the right. The MSBs are set to the sign of register **rs1**. The result is stored in register **rd**. The operation is performed in a 64-bit context.
Syntax: asr.q {rd}, {rs1}, {rs2}
FN4: 0x6

SHL.Q: (Logical left shift qword)
Description: Shifts register **rs1** by register **rs2** to the left, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: shl.q {rd}, {rs1}, {rs2}
FN4: 0x7

CCH.Q: (Addition carry check qword)
Description: Sets register **rd** to the carry-out bit of an addition between register **rs1** and register **rs2**. The operation is performed in a 64-bit context.
Syntax: cch.q {rd}, {rs1}, {rs2}
FN4: 0x8

BCH.Q: (Subtraction borrow check qword)
Description: Sets register **rd** to the borrow-out of a subtraction between register **rs1** and register **rs2**. The operation is performed in a 64-bit context.
Syntax: bch.q {rd}, {rs1}, {rs2}
FN4: 0x9

SLT.Q: (Set if less than qword)
Description: if register **rs1** is less than register **rs2** in the context of signed values, set register **rd** to 1. Else, set register **rd** to 0. The operation is performed in a 64-bit context.
Syntax: slt.q {rd}, {rs1}, {rs2}
FN4: 0xA

Arithmetic-immediate:

Opcode: 0xA / 0b001010
Format: DSI

ADDI.Q: (Add immediate qword)
Description: Adds registers **rs1** by **imm**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: addi.q {rd}, {rs1}, {imm}
FN4: 0x0

ANDI.Q: (Bitwise AND immediate qword)
Description: Bitwise AND between register **rs1** and **imm**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: andi.q {rd}, {rs1}, {imm}
FN4: 0x2

ORI.Q: (Bitwise OR immediate qword)
Description: Bitwise OR between register **rs1** and **imm**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: ori.q {rd}, {rs1}, {imm}
FN4: 0x3

XORI.Q: (Bitwise XOR immediate qword)
Description: Bitwise XOR between register **rs1** and **imm**, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: xori.q {rd}, {rs1}, {imm}
FN4: 0x4

SHRI.Q: (Logical right shift immediate qword)
Description: Shifts register **rs1** by **imm** to the right, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: shri.q {rd}, {rs1}, {imm}
FN4: 0x5

ASRI.Q: (Arithmetic right shift immediate qword)
Description: Shifts register **rs1** by **imm** to the right. The MSBs are set to the sign of register **rs1**. The result is stored in register **rd**. The operation is performed in a 64-bit context.
Syntax: asri.q {rd}, {rs1}, {imm}
FN4: 0x6

SHLI.Q: (Logical left shift immediate dword)
Description: Shifts register **rs1** by **imm** to the left, and stores the result in register **rd**. The operation is performed in a 64-bit context.
Syntax: shli.q {rd}, {rs1}, {imm}
FN4: 0x7

CCHI.Q: (Addition carry check immediate qword)
Description: Sets register **rd** to bit 64 of an addition between register **rs1** and register **imm**. The operation is performed in a 64-bit context.
Syntax: cchi.q {rd}, {rs1}, {imm}
FN4: 0x8

BCHI.Q: (Subtraction borrow check immediate qword)
Description: Sets register **rd** to bit 64 of a subtraction between register **rs1** and register **imm.** The operation is performed in a 64-bit context.
Syntax: bchi.q {rd}, {rs1}, {imm}
FN4: 0x9

SLTI.Q: (Set if less than immediate qword)
Description: if register **rs1** is less than **imm** in the context of signed values, set register **rd** to 1. Else, set register **rd** to 0. The operation is performed in a 64-bit context.
Syntax: slti.q {rd}, {rs1}, {rs2}
FN4: 0xA

Multiplication / Division extension

Classical multiplication/division instructions (without the .q suffix) will be executed on a 32-bit context and sign-extended to 64 bits.

Opcode: 0x0 / 0b000000
FN7: 0x1
Format: DSS

MUL.Q: (Multiply qword)
FN4: 0x8
Description: Register **rs1** is multiplied by **rs2** and result [63:0] is stored in **rd**. The operation is performed in a 64-bit context.
Syntax: mul.q {rd}, {rs1}, {rs2}

MULH.Q: (Multiply high qword)
FN4: 0xA
Description: Register **rs1** is multiplied by **rs2** and result [127:63] is stored in **rd**. The operation is performed in a 64-bit context.
Syntax: mulh.q {rd}, {rs1}, {rs2}

SMULH.Q: (Signed multiply high qword)
FN4: 0xB
Description: Register **rs1** is multiplied by **rs2** and result [127:63] is stored in **rd**. The operation is performed in a 64-bit context.
Syntax: smulh.q {rd}, {rs1}, {rs2}

DIV.Q: (Divide qword)
FN4: 0xC
Description: Register **rs1** is divided by **rs2** and the quotient is stored in **rd**. The operation is performed in a 64-bit context.
Syntax: div.q {rd}, {rs1}, {rs2}

SDIV.Q: (Signed divide qword)
FN4: 0xD
Description: Register **rs1** is divided by **rs2** and the quotient is stored in **rd**. The operation is performed in a signed context. The operation is performed in a 64-bit context.
Syntax: sdiv.q {rd}, {rs1}, {rs2}

REM.Q: (Remainder qword)
FN4: 0xE
Description: Register **rs1** is divided by **rs2** and the remainder is stored in **rd**. The operation is performed in a 64-bit context.
Syntax: mod.q {rd}, {rs1}, {rs2}

SREM.Q: (Signed remainder qword)
FN4: 0xE
Description: Register **rs1** is divided by **rs2** and the remainder is stored in **rd**. The operation is performed in a 64-bit context.
Syntax: srem.q {rd}, {rs1}, {rs2}

Atomic extension

64-bit support is implemented to the A extension in the SRM64 extension by implementing the .sd and .q suffixes.

Opcode: 0xA / 0b001010

LL.Q(.{fence}): (Load and link qword)
FN4: 0x3
FN7: 0x0
Description: Set register **rd** to memory[**rs1**][63:0] and link memory[**rs1**].
Syntax: ll.q {rd}, [{rs1}]

LL.SD(.{fence}): (Load and link signed dword)
FN4: 0x6
FN7: 0x0
Description: Set register **rd** to memory[**rs1**][31:0] and link memory[**rs1**]. Sign-extends.
Syntax: ll.d {rd}, [{rs1}]

SC.Q(.{fence}): (Store conditional qword)
FN4: 0x3
FN7: 0x1
Description: Store conditional to memory[**rs1**][63:0]
Syntax: sc.q {rd}, [{rs1}]

ACMPXCHG.Q(.{fence}): (Atomic compare and exchange qword)
FN4: 0x3
Description: Atomic compare and exchange using register **rd** [63:0], memory[rs1][63:0] and register **rs2** [63:0]
Syntax: acmpxchg.b {rd}, {rs2}, [{rs1}]

AXADD.Q(.{fence}): (Atomic exchange and add qword)
FN7: 0x3
FN4: 0x3
Description: Atomic exchange and add using register **rd** [63:0], memory[rs1][63:0] and register **rs2.** Zero-extends for memory[**rs1**][63:0] stored into register **rd**.
Syntax: axadd.q {rd}, {rs2}, [{rs1}]

AXADD.SD(.{fence}): (Atomic exchange and add signed dword)
FN7: 0x3
FN4: 0x6
Description: Atomic exchange and add using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Sign-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axadd.sd {rd}, {rs2}, [{rs1}]

AXSUB.Q(.{fence}): (Atomic exchange and subtract qword)
FN7: 0x4
FN4: 0x3
Description: Atomic exchange and subtract using register **rd** [63:0], memory[rs1][63:0] and register **rs2.** Zero-extends for memory[**rs1**][63:0] stored into register **rd**.
Syntax: axadd.q {rd}, {rs2}, [{rs1}]

AXSUB.SD(.{fence}): (Atomic exchange and subtract signed dword)
FN7: 0x4
FN4: 0x6
Description: Atomic exchange and subtract using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Sign-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axadd.sd {rd}, {rs2}, [{rs1}]

AXAND.Q(.{fence}): (Atomic exchange and AND qword)
FN7: 0x5
FN4: 0x3
Description: Atomic exchange and AND using register **rd** [63:0], memory[rs1][63:0] and register **rs2.** Zero-extends for memory[**rs1**][63:0] stored into register **rd**.
Syntax: axand.q {rd}, {rs2}, [{rs1}]

AXAND.SD(.{fence}): (Atomic exchange and AND signed dword)
FN7: 0x5
FN4: 0x6
Description: Atomic exchange and AND using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Sign-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axand.sd {rd}, {rs2}, [{rs1}]

AXIOR.Q(.{fence}): (Atomic exchange and OR qword)
FN7: 0x6
FN4: 0x3
Description: Atomic exchange and OR using register **rd** [63:0], memory[rs1][63:0] and register **rs2.** Zero-extends for memory[**rs1**][63:0] stored into register **rd**.
Syntax: axior.q {rd}, {rs2}, [{rs1}]

AXIOR.SD: (Atomic exchange and OR signed dword)
FN7: 0x6
FN4: 0x6
Description: Atomic exchange and OR using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Sign-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axior.d {rd}, {rs2}, [{rs1}]

AXXOR.Q: (Atomic exchange and XOR qword)
FN7: 0x7
FN4: 0x3
Description: Atomic exchange and XOR using register **rd** [63:0], memory[rs1][63:0] and register **rs2.** Zero-extends for memory[**rs1**][63:0] stored into register **rd**.
Syntax: axxor.q {rd}, {rs2}, [{rs1}]

AXXOR.SD: (Atomic exchange and XOR signed dword)
FN7: 0x7
FN4: 0x6
Description: Atomic exchange and XOR using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Sign-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axxor.sd {rd}, {rs2}, [{rs1}]

AXMAX.Q: (Atomic exchange and MAX qword)
FN7: 0x8
FN4: 0x3
Description: Atomic exchange and MAX using register **rd** [63:0], memory[rs1][63:0] and register **rs2.** Zero-extends for memory[**rs1**][63:0] stored into register **rd**.
Syntax: axmax.q {rd}, {rs2}, [{rs1}]

AXMAX.SD: (Atomic exchange and MAX signed dword)
FN7: 0x8
FN4: 0x6
Description: Atomic exchange and MAX using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Sign-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axmax.sd {rd}, {rs2}, [{rs1}]

AXMIN.Q: (Atomic exchange and MIN qword)
FN7: 0x9
FN4: 0x3
Description: Atomic exchange and MIN using register **rd** [63:0], memory[rs1][63:0] and register **rs2.** Zero-extends for memory[**rs1**][63:0] stored into register **rd**.
Syntax: axmin.q {rd}, {rs2}, [{rs1}]

AXMIN.SD: (Atomic exchange and MIN signed dword)
FN7: 0x9
FN4: 0x6
Description: Atomic exchange and MIN using register **rd** [31:0], memory[rs1][31:0] and register **rs2.** Sign-extends for memory[**rs1**][31:0] stored into register **rd**.
Syntax: axmin.sd {rd}, {rs2}, [{rs1}]

Control and Status Registers (CSRs)

The **System extension** allows SRM Central Processing Units to support up to 4096 Control and Status Registers (CSRs). The base CSRs included are:

Address	Label	Function	Privilege
0x800	SSTATUS	Supervisor status	SRW
0x801	SISRA	Supervisor Interrupt Routine Service address	SRW
0x802	SIRA	Supervisor interrupt return address	SRW
0x803	SCAUSE	Supervisor interrupt/exception cause	SRW
0x804	SEINST	Supervisor exception instruction	SRW
0x805	SERIOA	Supervisor exception requested address	SRW
0x806	ST	Supervisor timer	SRW
0x807	STL	Supervisor timer's limit	SRW
0x808	STEMP	Supervisor temporary	SRW
0xC00	MSTATUS	Machine status	MRW
0xC01	MISRA	Machine Interrupt Routine Service address	MRW
0xC02	MIRA	Machine interrupt return address	MRW
0xC03	MCAUSE	Machine interrupt/exception cause	MRW
0xC04	MEINST	Machine exception instruction	MRW
0xC05	MERIOA	Machine exception requested address	MRW
0xC06	MT	Machine timer	MRW
0xC07	MTL	Machine timer's limit	MRW
0xC08	MTEMP	Machine temporary	MRW
0xF00	MISA	Machine ISA type	MRO
0xF01	MHTID	Machine hardware thread ID	MRO

PRIVILEGE LEVELS:

0. USER: Restricted access to memory and no access to IO; Privilege level of user programs.
1. SUPERVISOR: Limited access to IO and unrestricted access to user-level memory; Privilege level of kernels and drivers
2. *RESERVED*
3. MACHINE: Unrestricted access to all IO and memory; Privilege level of firmware programs (Example: BIOS).

Machine level CSRs

MSTATUS: (Machine status register)

Position	Function
0-1	Current privilege mode.
1-2	Saved privilege mode.
4	Enable machine-level interrupts.
5	Enable memory and IO protection from lower privilege modes.

MT: (Machine timer)
Counter that increments every clock cycle. when its value is greater than or equal to the contents of **mtl**, **mt** is reset to 0 and an exception with ID 0x8 is raised. If **mtl** is 0, machine timer exceptions will be disabled.

Supervisor level CSRs

SSTATUS: (Supervisor status register)

Position	Function
0	Enable supervisor-level interrupts.
1	Enable memory and IO protection from user mode.

ST: (Supervisor timer)
Counter that increments every clock cycle if the current privilege level is user mode. when its value is greater than or equal to the contents of **stl**, **st** is reset to 0 and an exception with ID 0x6 is raised. If **stl** is 0, supervisor timer exceptions will be disabled.

Privileged instructions

Executing privileged instructions at a lower privilege level than their minimum privilege level will trigger an exception type 0x9 (Illegal instruction) and the illegal instruction will be copied into the **seinst** or **mesinst** CSRs (Depending on the privilege mode in which the instruction was executed).

INSTRUCTION	EXTENSION	MINIMUM PRIVILEGE
MRET	SYSTEM	MACHINE
SYSRET	SYSTEM	SUPERVISOR
IN	SYSTEM	SUPERVISOR
OUT	SYSTEM	SUPERVISOR

Interrupt and exception handling

Definition of interrupt: *An interrupt is a request for the core to interrupt currently executing code (when permitted), so that the event can be processed in a timely manner. If the request is accepted, the processor will suspend its current activities, save its state, and execute a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is often temporary, allowing the software to resume normal activities after the interrupt handler finishes.*

Definition of exception: *Type of interrupt triggered when an internal error is detected inside the core.*

The process of triggering interrupts/exceptions varies depending on the privilege mode that the core was in when the interrupt/exception was triggered.

MACHINE LEVEL (LEVEL 3):

1. If **mstatus**[4] is 0, the external interrupt will be ignored or placed in a queue (chosen by the manufacturer). In case of an exception, the checking process is skipped.
2. IP is copied to the **mira** CSR.
3. IP is set to the contents of the **misra** CSR

SUPERVISOR LEVEL (LEVEL 1):

1. If **mstatus**[4] is 0, the external interrupt is ignored or placed in a queue (chosen by the manufacturer). In case of an exception, the checking process is skipped.
2. IP is copied to the **mira** CSR.
3. IP is set to the contents of the **misra** CSR.
4. The privilege mode is saved into **mstatus**[2:1].
5. The privilege mode is set to Machine mode.

USER LEVEL (LEVEL 0):

1. If **sstatus**[0] is 0 and it’s an user-level interrupt, the external interrupt is ignored or placed in a queue (chosen by the manufacturer). In case of an exception, the checking process is skipped.
2. IP is copied to the **sisra** CSR.
3. IP is set to the contents of the **sisra** CSR.
4. The privilege mode is set to Supervisor mode.

Interrupt cause IDs

When interrupts are triggered, **mcause** or **kcause** (depending on the level on which the interrupt was triggered) are updated with the type of interrupt that was triggered. The most significant bit of the ID must be 0 if it's an exception and 1 if it's an interrupt.

The standard IDs used are:

Cause ID	Cause
0x0	System call from user mode
0x1	System call from supervisor mode
0x2	*Reserved*
0x3	System call from machine mode
0x4	System break
0x5	*Reserved*
0x6	Supervisor timer interrupt (**st** == **stl**)
0x7	*Reserved*
0x8	Machine timer interrupt (**mt** == **mtl**)
0x9	Illegal instruction
0xA	Illegal memory address
0xB	Illegal IO address