

Slicudis. RISC. Machine. Manual



ISA version 5.3 Pre-release 2
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).....	24
Multiplication/Division extension (M).....	24
Atomic extension (A).....	26
SRM64: The standard 64-bit extension.....	35
Base instruction set.....	35
Multiplication / Division extension.....	39
Atomic extension.....	41
Interrupt and exception handling.....	45
Interrupt cause IDs.....	46

Recent changelog

- Changed the data format to little endian.
- Implemented the **fence** instruction as part of the base **Base system extension** instead of implementing it into the **Zmo extension** (which was now removed).
- Joined all the opcodes of the **Base system extension** into a single opcode.
- Changed the mnemonic names for IO instructions from the **Base system extension**.
- Removed the Q bit from the **64-bit extension**.
- Implemented 32-bit-specific arithmetic instructions as part of the 64-bit extension.
- Moved the instructions from the **M extension** to opcode 0x0.
- Implemented **slt** and **slti** as part of the base instruction set
- Implemented the **A (Atomic) extension**.
- Separated **SRM64** from the common standard extensions.

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:

S - Base system extension

M - Multiplication and division extension

A - Atomic extension

H - Register reduction extension

L - Hypervisor extension (Work in progress)

F - Single precision Floating point extension (Work in progress)

D - Double precision Floating point extension (Work in progress)

Zvm - Protected/Virtual memory extension (Work in progress)

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: SRM64SMALFZvm

SRM32 Base instruction set

Formats

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
DI	imm [20:0]																					rd				opcode						
DSS	fn7							rs2				fn4				rs1				rd				opcode								
DSI	imm [11:0]												fn4				rs1				rd				opcode							
SSI	imm [6:0]							rs2				fn4				rs1				imm [11:7]				opcode								

Arithmetic instructions

Opcode: 0x0 / 0b0000000

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: slt {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: $\text{memory}[\text{rs1} + \text{imm}] = \text{rs2}[7:0]$.

Syntax: `stb {rs2}, [{rs1}], {imm}`

FN4: 0x0

STW: (Store word)

Description: $\text{memory}[\text{rs1} + \text{imm}] = \text{rs2}[15:0]$. Address[0] is padded with 0.

Syntax: `stw {rs2}, [{rs1}], {imm}`

FN4: 0x1

STD: (Store dword)

Description: $\text{memory}[\text{rs1} + \text{imm}] = \text{rs2}[31:0]$. 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: jne {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: jge {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 $(rs1 \leq rs2)$ | 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 $(rs1 \neq 0)$ | 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 is 0):

place: clr {rd}

else if (imm is positive):

if (imm[10:0] != 0):

place: addi {rd}, zr, {imm}[10:0]

if (imm[31:11] != 0):

place: lui {rd}, zr, {imm}[31:11]

else if (imm is negative):

if (imm[10:0] != 0x7ff):

place: addi {rd}, zr, -1*{imm}[10:0] (positive)}

if (imm[31:11] != 0x1ffff):

place: lui {rd}, zr, {imm}[31:11]

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: $rd = \text{signed}(rs1 < rs2) ? 1 : 0$

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.

The System extension supports up to 4096 CSRs.

Register	Label	Function	Read privilege	Write privilege
csr0	mstatus	Machine status register	Machine	Machine
csr1	miva	Machine interrupt vector address	Machine	Machine
csr2	mra	Machine interrupt return address	Machine	Machine
csr3	mcause	Machine interrupt cause	Machine	Machine
csr4	mrpiv	Machine interrupt saved privilege mode	Machine	Machine
csr5-1023	<i>reserved</i>	<i>reserved</i>	Machine	Machine
csr6-2047	<i>reserved</i>	<i>reserved</i>	<i>reserved</i>	<i>reserved</i>
csr2048	kstatus	Kernel status register	Kernel	Kernel
csr2049	kiva	Kernel interrupt vector address	Kernel	Kernel
csr2050	kra	Kernel interrupt return address	Kernel	Kernel
csr2051	kcause	Kernel interrupt cause	Kernel	Kernel
csr2052	kpriv	Kernel interrupt saved privilege mode	Kernel	Kernel
csr2053-3071	<i>reserved</i>	<i>reserved</i>	Kernel	Kernel
csr3072-4095	<i>reserved</i>	<i>reserved</i>	User	User

MSTATUS: (Machine status register)

Bit position	Function
0	Enable external interrupts at machine level
1 - 2	Current privilege mode

KSTATUS: (Kernel status register)

Bit position	Function
0	Enable external interrupts at kernel level

PRIVILEGE LEVELS:

- 0 - USER (Least privileged): Restricted hardware and memory access
- 1 - KERNEL: Unrestricted access to every user-level process and almost all IO
- 2 - *RESERVED*
- 3 - MACHINE (Most privileged)

New formats:

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
CSB	imm [11:0]												fn4				reserved				sel				opcode							
FNC	fn7								-	I	O	R	W	fn4				reserved								opcode						

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: DSIFN4: 0x2Description: 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: DSIFN4: 0x3Description: 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: CSBFN4: 0x4Description: csr[imm][sel] is set to 1 (rd and rs1 are just the operands, not the selected registers by them).Syntax: csrs {csr}, {position}**CSRC: (CSR bit clear)**Format: CSBFN4: 0x5Description: csr[imm][sel] is set to 0 (rd and rs1 are just the operands, not the selected registers by them).Syntax: csrs {csr}, {position}**IN.B: (Input byte)**Format: DSIFN4: 0x6FN7: 0x0Description: Register **rd** is set to the input data of port [rs1][7:0]. Zero-extends. This instruction can only be executed on a privilege level equal or higher than level 1.Syntax: inb {rd}, [{rs1}, {imm}]**IN.W: (Input word)**Format: DSIFN4: 0x6FN7: 0x1Description: Register **rd** is set to the input data of port [rs1][15:0]. Zero-extends and the lowest bit of the address is padded with a 0. This instruction can only be executed on a privilege level equal or higher than level 1.Syntax: inw {rd}, [{rs1}, {imm}]

IN.D: (Input dword)

Format: DSI

FN4: 0x6

FN7: 0x2

Description: Register **rd** is set to the input data of port [**rs1**][31:0]. Zero-extends and the lowest 2 bits of the address are padded with a 0s. This instruction can only be executed on a privilege level equal or higher than level 1.

Syntax: ind {rd}, [{rs1}, {imm}]

IN.B: (Input signed byte)

Format: DSI

FN4: 0x6

FN7: 0x4

Description: Register **rd** is set to the input data of port [**rs1**][7:0]. Sign-extends and the lowest 2 bits of the address are padded with a 0s. This instruction can only be executed on a privilege level equal or higher than level 1.

Syntax: insb {rd}, [{rs1}, {imm}]

IN.SW: (Input signed word)

Format: DSI

FN4: 0x6

FN7: 0x5

Description: Register **rd** is set to the input data of port [**rs1**][15:0]. Sign-extends and the lowest 2 bits of the address are padded with a 0s. This instruction can only be executed on a privilege level equal or higher than level 1.

Syntax: insw {rd}, [{rs1}]

OUT.B: (Output byte)

Format: SSI

FN4: 0x7

FN7: 0x0

Description: port [**rs1**] is set to the contents of register **rs2** [7:0] (Zero-extends). This instruction can only be executed on a privilege level equal or higher than level 1.

Syntax: outb {rs2}, [{rs1}]

OUT.W: (Output word)

Format: SSI

FN4: 0x7

FN7: 0x1

Description: port [**rs1**] is set to the contents of register **rs2** [15:0] (Zero-extends). This instruction can only be executed on a privilege level equal or higher than level 1.

Syntax: outw {rs2}, [{rs1}]

OUT.D: (Output dword)Format: SSIFN4: 0x7FN7: 0x2Description: port [**rs1**] is set to the contents of register **rs2** [31:0] (Zero-extends). This instruction can only be executed on a privilege level equal or higher than level 1.Syntax: outd {rs2}, [{rs1}]**FENCE: (Fence)**Format: FNCFN4: 0x8FN7: 0x0Description: 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)

Type syntax	I bit	O bit	R bit	W bit
W	0	0	0	1
R	0	0	1	0
RW	0	0	1	1
O	0	1	0	0
OW	0	1	0	1
OR	0	1	1	0
ORW	0	1	1	1
I	1	0	0	0
IW	1	0	0	1
IR	1	0	1	0
IRW	1	0	1	1
IO	1	1	0	0
IOW	1	1	0	1
IOR	1	1	1	0
IORW / (Empty)	1	1	1	1

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 / 0b0000000

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}

MOD: (Modulo)

Description: Register **rs1** is divided by **rs2** and the remainder of the result is stored in **rd**.

Syntax: mod {rd}, {rs1}, {rs2}

FN4: 0x6

SMOD: (Signed modulo)

Description: **rd** = signed **rs1** % signed **rs2**

Syntax: smod {rd}, {rs1}, {rs2}

FN4: 0x7

M macro-instructions:

(MUL/SMUL/MULH/SMULH/DIV/SDIV/MOD/SMOD)I: ([...] Immediate)

Description: **rd** = **rs1** [...] **imm**

Syntax: (mul/smul/mulh/smulh/div/sdiv/mod/smod)i {rd}, {rs1}, {imm}

Conversion:

ldi at, {imm}

(mul/smul/mulh/smulh/div/sdiv/mod/smod) {rd}, {rs1}, at

MADD/SMADD: (Multiply / Signed multiply and accumulate)

Description: **rd** += **rs1** * **rs2**

Syntax: madd/smadd {rd}, {rs1}, {imm}

Conversion:

mul at, {rs1}, {rs2}

add {rd}, {rs}, at

(MADD/SMADD)I: (Multiply / Signed multiply immediate and accumulate)

Description: **rd** += **rs1** * **rs2**

Syntax: (madd/smadd)i {rd}, {rs1}, {imm}

Conversion:

ldi at, {imm}

(s)mul at, {rs1}, at

add {rd}, {rd}, at

Atomic extension (A)

The atomic extension implements a total of 54 atomic instructions (64 in SRM64) that can be used for thread synchronization.

Opcode: 0x9 / 0b001001

Format: DSS

Load and Link:

FN7: 0x0

General description: 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: (Load and link byte)

FN4: 0x0

Description: Set register **rd** to memory[**rs1**][7:0] and link memory[**rs1**]. Zero-extends.

Syntax: ll.b {rd}, [{rs1}]

LL.W: (Load and link word)

FN4: 0x1

Description: Set register **rd** to memory[**rs1**][15:0] and link memory[**rs1**]. Zero-extends.

Syntax: ll.w {rd}, [{rs1}]

LL.D: (Load and link dword)

FN4: 0x2

Description: Set register **rd** to memory[**rs1**][31:0] and link memory[**rs1**]. Zero-extends.

Syntax: ll.d {rd}, [{rs1}]

LL.SB: (Load and link signed byte)

FN4: 0x0

Description: Set register **rd** to memory[**rs1**][7:0] and link memory[**rs1**]. Sign-extends.

Syntax: ll.sb {rd}, [{rs1}]

LL.SW: (Load and link signed word)

FN4: 0x1

Description: Set register **rd** to memory[**rs1**][15:0] and link memory[**rs1**]. Sign-extends.

Syntax: ll.sw {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: (Store conditional byte)FN4: 0x0

Description: Store conditional to memory[**rs1**][7:0]

Syntax: sc.b {rd}, [{rs1}]

SC.W: (Store conditional word)FN4: 0x1

Description: Store conditional to memory[**rs1**][15:0]

Syntax: sc.w {rd}, [{rs1}]

SC.D: (Store conditional dword)FN4: 0x2

Description: Store conditional to memory[**rs1**][31:0]

Syntax: sc.d {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: (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 {rd}, {rs2}, [{rs1}]

ACMPXCHG.W: (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.w {rd}, {rs2}, [{rs1}]

ACMPXCHG.D: (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.d {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 register **rd**. The old contents of register **rd** [bitwidth-1:0] are stored into memory[**rs1**]

AXADD.B: (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 {rd}, {rs2}, [{rs1}]

AXADD.W: (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 {rd}, {rs2}, [{rs1}]

AXADD.D: (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 {rd}, {rs2}, [{rs1}]

AXADD.SB: (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 {rd}, {rs2}, [{rs1}]

AXADD.SW: (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 {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 register **rd**. The old contents of register **rd** [bitwidth-1:0] are stored into memory[**rs1**]

AXSUB.B: (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 {rd}, {rs2}, [{rs1}]

AXSUB.W: (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 {rd}, {rs2}, [{rs1}]

AXSUB.D: (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 {rd}, {rs2}, [{rs1}]

AXSUB.SB: (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 {rd}, {rs2}, [{rs1}]

AXSUB.SW: (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 {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 register **rd**. The old contents of register **rd** [bitwidth-1:0] are stored into memory[**rs1**]

AXAND.B: (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 {rd}, {rs2}, [{rs1}]

AXAND.W: (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 {rd}, {rs2}, [{rs1}]

AXAND.D: (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 {rd}, {rs2}, [{rs1}]

AXAND.SB: (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 {rd}, {rs2}, [{rs1}]

AXAND.SW: (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 {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 register **rd**. The old contents of register **rd** [bitwidth-1:0] are stored into memory[**rs1**]

AXIOR.B: (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 {rd}, {rs2}, [{rs1}]

AXIOR.W: (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 {rd}, {rs2}, [{rs1}]

AXIOR.D: (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 {rd}, {rs2}, [{rs1}]

AXIOR.SB: (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 {rd}, {rs2}, [{rs1}]

AXIOR.SW: (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 {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 register **rd**. The old contents of register **rd** [bitwidth-1:0] are stored into memory[**rs1**]

AXXOR.B: (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 {rd}, {rs2}, [{rs1}]

AXXOR.W: (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 {rd}, {rs2}, [{rs1}]

AXXOR.D: (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 {rd}, {rs2}, [{rs1}]

AXXOR.SB: (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 {rd}, {rs2}, [{rs1}]

AXXOR.SW: (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 {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 register **rd**. The old contents of register **rd** [bitwidth-1:0] are stored into memory[**rs1**]

AXMAX.B: (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 {rd}, {rs2}, [{rs1}]

AXMAX.W: (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 {rd}, {rs2}, [{rs1}]

AXMAX.D: (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 {rd}, {rs2}, [{rs1}]

AXMAX.SB: (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 {rd}, {rs2}, [{rs1}]

AXMAX.SW: (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 {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 register **rd**. The old contents of register **rd** [bitwidth-1:0] are stored into memory[**rs1**]

AXMIN.B: (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 {rd}, {rs2}, [{rs1}]

AXMIN.W: (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 {rd}, {rs2}, [{rs1}]

AXMIN.D: (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 {rd}, {rs2}, [{rs1}]

AXMIN.SB: (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 {rd}, {rs2}, [{rs1}]

AXMIN.SW: (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 {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: $\text{memory}[\text{rs1} + \text{imm}] = \text{rs2}[63:0]$.

Syntax: stq {rs2}, [{rs1}, {imm}]

LDSD: (Load signed dword)

Opcode: 0x4 / 0b000100

FN4: 0x6

Description: $\text{rd} = \text{memory}[\text{rs1} + \text{imm}][31:0]$. Sign-extends.

Syntax: ldsd {rd}, [{rs1}, {imm}]

LDQ: (Load qword)

Opcode: 0x4 / 0b000100

FN4: 0x3

Description: $\text{rd} = \text{memory}[\text{rs1} + \text{imm}][63:0]$. Zero-extends.

Syntax: ldq {rd}, [{rs1}, {imm}]

Arithmetic:

Opcode: 0x0 / 0b0000000

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 / 0b0000000

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.d {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.d {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.d {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.d {rd}, {rs1}, {rs2}

MOD.Q: (Modulo 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.d {rd}, {rs1}, {rs2}

SMOD.Q: (Signed modulo 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: smod.d {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: (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: (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: (Store conditional qword)

FN4: 0x3

FN7: 0x1

Description: Store conditional to memory[**rs1**][63:0]

Syntax: sc.q {rd}, [{rs1}]

ACMPXCHG.Q: (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: (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: (Atomic exchange and add signed dword)FN7: 0x3FN4: 0x6Description: 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: (Atomic exchange and subtract qword)**FN7: 0x4FN4: 0x3Description: 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: (Atomic exchange and subtract signed dword)**FN7: 0x4FN4: 0x6Description: 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: (Atomic exchange and AND qword)**FN7: 0x5FN4: 0x3Description: 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: (Atomic exchange and AND signed dword)**FN7: 0x5FN4: 0x6Description: 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: (Atomic exchange and OR qword)**FN7: 0x6FN4: 0x3Description: 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: 0x6FN4: 0x6Description: 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: 0x7FN4: 0x3Description: 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: 0x7FN4: 0x6Description: 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: 0x8FN4: 0x3Description: 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: 0x8FN4: 0x6Description: 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: 0x9FN4: 0x3Description: 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}]

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 varies depending on the privilege mode that the core was in when the interrupt was triggered.

MACHINE LEVEL (LEVEL 3):

1. If **mstatus**[0] is 0, the external interrupt is ignored or placed in a queue (chosen by the manufacturer). In case of an internal interrupt, the checking process is skipped.
2. If **mstatus**[0] (Unless it's an internal interrupt) is 1, the following processes happen:
3. IP is copied to the **mra** CSR.
4. IP is set to the contents of the **miva** CSR
- 5.

KERNEL LEVEL (LEVEL 1):

1. If **kstatus**[0] is 0, the external interrupt is ignored or placed in a queue (chosen by the manufacturer). In case of an internal interrupt, the checking process is skipped.
2. If **kstatus**[0] (Unless it's an internal interrupt) is 1, the following processes happen:
3. IP is copied to the **mra** CSR.
4. IP is set to the contents of the **miva** CSR.
5. The privilege mode is saved to the **mpriv** CSR.
6. The privilege mode is set to level 3 (Machine mode).

USER LEVEL (LEVEL 0):

1. IP is copied to the **kra** CSR.
2. IP is set to the contents of the **kiva** CSR..
3. The privilege mode is saved to the **kpriv** CSR.
4. The privilege mode is set to level 1 (Kernel 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 standard IDs used are:

0x0: System call from user mode
 0x1: System call from kernel mode
 0x2: *Reserved*
 0x3: System call from machine mode
 0x4: System break

EXAMPLE:

Using a table of addresses for each type of interrupt

```
Handling_section:
;Context switch logic here...
csrr kcause, t0          ;Get the interrupt cause
shli t0, t0, 2           ;Adjust
ldi t1, TableBaseAddress ;Get the interrupt table address
add t0, t0, t1           ;Calculate the address
ldd t0, [t0, 0]          ;Fetch the vector
jalr zr, t0, 0           ;Jump to the selected routine
;...
```