RISC-V Instruction Set Summary

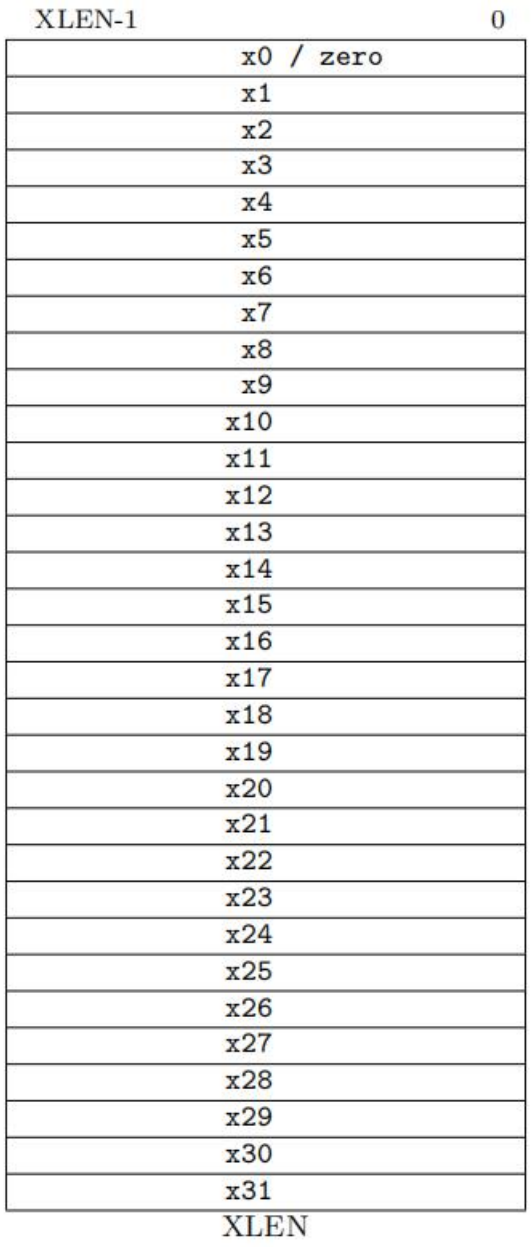
CONTENT

Revision History

Version Number	Date	Author	Summary
0.1	2021/03/01	Haitao Xie (@Wechat:xie22289255) (@E-mail:tjxht1994@163.com)	Initial revision

1 Register File

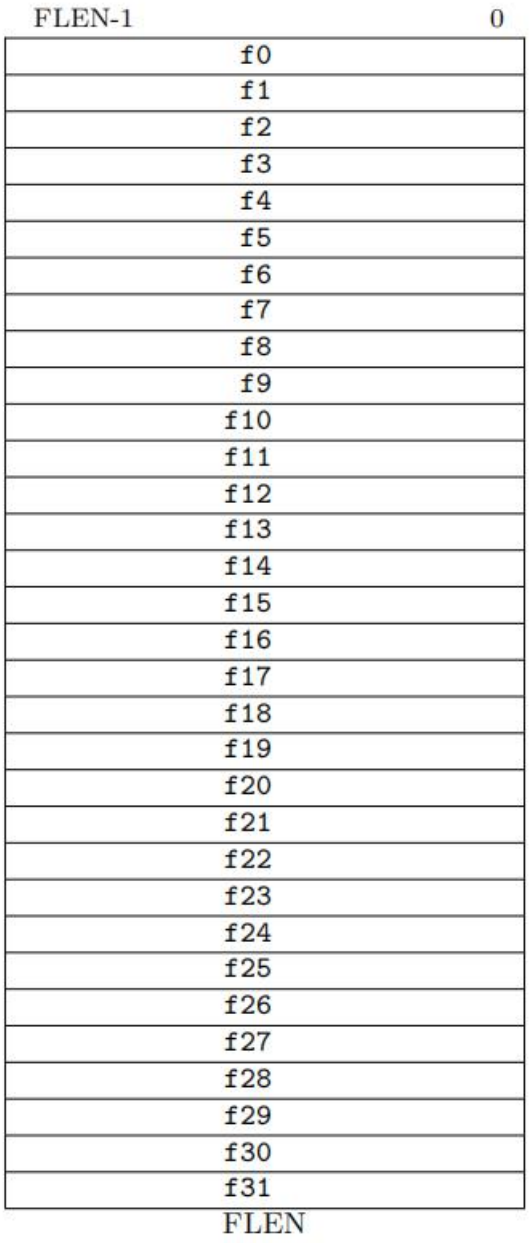
1.1 GPR



Register	ABI	Description
x0	zero	Zero value
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/Alternate Link Register
x6-7	t1-2	Temporary register

x8	s0/fp	Reserved register/frame pointer
x9	s1	Reserved register
x10-11	a0-a1	Function parameter/return value
x12-17	a2-a7	Function parameters
x18-27	s2-s11	Reserved register
x28-31	t3-t6	Temporary register

1.2 FGPR



Register	ABI	Description
f0~f7	ft0-7	Floating-point temporary register
f8~f9	fs0-1	Floating-point retention register

f10~f11	fa0-1	Floating point parameter/return value register
f12~f17	fa2-7	Floating point parameter register
f18~f27	fs2-11	Floating-point retention register
f28~f31	ft8-11	Floating-point temporary register

1.3 Vector Registers

1.3.1 Vector CSRs

The vector extension adds 32 vector registers, and seven unprivileged CSRs (vstart, vxsat, vxrm, vcsr, vtype, vl, vlenb) to a base scalar RISC-V ISA.

Table 1-1 New Vector CSRs

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start position
0x009	URW	vxsat	Fixed-Point Saturate Flag
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

1.3.1.1 Vector type register (vtype)

The read-only XLEN-wide vector type CSR, vtype provides the default type used to interpret the contents of the vector register file, and can only be updated by vsetvl{i} instructions. The vector type also determines the organization of elements in each vector register, and how multiple vector registers are grouped.

In the base vector extension, the vtype register has three fields, vill, vsew[2:0], and vlmul[1:0].

Table 3-2 Vtype Register Layout

Bits	Name	Description
XLEN-1:5		Reserved
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

1.3.1.1.1 Vector standard element width (vsew)

The value in vsew sets the dynamic standard element width (SEW). By default, a vector register is viewed as being divided into VLEN / SEW standard-width elements. In the base vector extension, only SEW up to max(XLEN,FLEN) are required to be supported.

Table 3-3 Vsew[2:0] (Standard Element Width) Encoding

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

Table 3-4 Example VLEN = 256 bits

SEW	Elements per vector register
64	4
32	8
16	16
8	32

1.3.1.1.2 Vector Register Grouping (vlmul)

Multiple vector registers can be grouped together, so that a single vector instruction can operate on multiple vector registers. Vector register groups allow double-width or larger elements to be operated on with the same vector length as standard-width elements. Vector register groups also provide greater execution efficiency for longer application vectors.

The term vector register group is used herein to refer to one or more vector registers used as a single operand to a vector instruction. The number of vector registers in a group, LMUL, is an integer power of two set by the vlmul field in vtype (LMUL = $2^{vlmul[1:0]}$).

The derived value VLMAX = LMUL*VLEN/SEW represents the maximum number of elements that can be operated on with a single vector instruction given the current SEW and LMUL settings.

Table 3-5 Vector Register Grouping

vlmul		LMUL	#groups	VLMAX	Grouped registers
0	0	1	32	VLEN/SEW	vn (single register in group)

0	1	2	16	2*VLEN/SEW	vn, vn+1
1	0	4	8	4*VLEN/SEW	vn, ⋯, vn+3
1	1	8	4	8*VLEN/SEW	vn, ⋯, vn+7

When vlmul=01, then vector operations on register v n also operate on vector register v n+1, giving twice the vector length in bits. Instructions specifying a vector operand with an odd-numbered vector register will raise an illegal instruction exception.

Similarly, when vlmul=10, vector instructions operate on four vector registers at a time, and instructions specifying vector operands using vector register numbers that are not multiples of four will raise an illegal instruction exception. When vlmul=11, operations operate on eight vector registers at a time, and instructions specifying vector operands using register numbers that are not multiples of eight will raise an illegal instruction exception.

Mask register instructions always operate on a single vector register, regardless of LMUL setting.

1.3.1.2 Vector Length Register (vl)

The XLEN-bit-wide read-only vl CSR can only be updated by the vsetvli and vsetvl instructions.

The vl register holds an unsigned integer specifying the number of elements to be updated by a vectorinstruction. Elements in any destination vector register group with indices $\geqslant$ vl are unmodi ed during execution of a vector instruction. When vstart $\geqslant$ vl, no elements are updated in any destination vector register group.

1.3.2 Vector General Purpose Registers (VGPR)

The vector extension adds 32 architectural vector registers, v0-v31 to the base scalar RISC-V ISA. Each vector register has a fixed VLEN bits of state.For programmer, VGRF is reconfigurable, and variable-length.

1.3.3 Mapping of Vector Elements to Vector Register State

The following diagrams illustrate how different width elements are packed into the bytes of a vector register depending on the current SEW and LMUL settings, as well as implementation ELEN and VLEN. Elements are packed into each vector register with the least-signi cant byte in the lowest-numbered bits.

1.3.3.1 Mapping with LMUL=1

When LMUL=1, elements are simply packed in order from the least-signi cant to most-signi cant bits of the vector register

The element index is given in hexadecimal and is shown placed at the least-significant

```
VLEN=32b

Byte          3 2 1 0

SEW=8b        3 2 1 0
SEW=16b         1   0
SEW=32b             0


VLEN=64b

Byte        7 6 5 4 3 2 1 0

SEW=8b      7 6 5 4 3 2 1 0
SEW=16b       3   2   1   0
SEW=32b           1       0
SEW=64b                   0

VLEN=128b

Byte        F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=8b      F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW=16b       7   6   5   4   3   2   1   0
SEW=32b           3       2       1       0
SEW=64b                   1               0
SEW=128b                                  0


VLEN=256b
 Byte     1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F  E  D  C  B  A  9  8  7  6  5  4  3  2  1  0
 SEW=8b   1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F  E  D  C  B  A  9  8  7  6  5  4  3  2  1  0
 SEW=16b    F    E    D    C    B    A    9    8    7    6    5    4    3    2    1    0
 SEW=32b         7         6         5         4         3         2         1         0
 SEW=64b                   3                   2                   1                   0
 SEW=128b                                      1                                       0
```

1.3.3.2 Mapping with LMUL > 1

When vector registers are grouped, the elements of the vector register group are striped across

the constituent vector registers.

```
Example 1: VLEN=32b, SEW=16b, LMUL=2

Byte          3 2 1 0
v2*n            1   0

v2*n+1          3   2

Example 2: VLEN=64b, SEW=32b, LMUL=2

Byte          7 6 5 4 3 2 1 0
v2*n                1       0

v2*n+1              3       2

Example 3: VLEN=128b, SEW=32b, LMUL=2

Byte         F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n               3       2       1       0

v2*n+1             7       6       5       4

Example 4: VLEN=256b, SEW=32b, LMUL=2

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n             B       A       9       8       3       2       1       0

v2*n+1           F       E       D       C       7       6       5       4
```

1.3.3.3 Mask Register Layout

A vector mask occupies only one vector register regardless of SEW and LMUL. The mask bits that are used for each vector operation depends on the current SEW and LMUL setting.

The maximum number of elements in a vector operand is:

VLMAX = LMUL * VLEN/SEW

A mask is allocated for each element by dividing the mask register into VLEN/VLMAX fields. The size of each mask element in bits, MLEN, is:

MLEN = VLEN/VLMAX
= VLEN/(LMUL * VLEN/SEW)
= SEW/LMUL

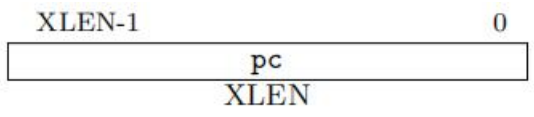
The size of MLEN varies from ELEN (SEW=ELEN, LMUL=1) down to 1 (SEW=8b,LMUL=8), and hence a single vector register can always hold the entire mask register.

The mask bits for element i are located in bits [MLEN*i+(MLEN-1) : MLEN*i] of the mask register. When a mask element is written by a compare instruction, the low bit in the mask element is written with the compare result and the upper bits of the mask element are zeroed. Destination mask elements past the end of the current vector length are unchanged. When a value is read as a mask, only the least-significant bit of the mask element is used to control masking and the upper bits are ignored.

The pattern is such that for constant SEW/LMUL values, the effective predicate bits are located in the same bit of the mask vector register, which simpli es use of masking in loops with mixed-width elements.

```
VLEN=32b

          Byte     3    2    1    0
LMUL=1,SEW=8b
                   3    2    1    0  Element
                 [24][16][08][00] Mask bit position in decimal

 LMUL=2,SEW=16b
                              1    0
                            [08]  [00]
                              3    2
                            [24]  [16]


LMUL=4,SEW=32b
                                  0
                                 [00]
                                 1
                                 [08]
                                 2
                                 [16]
                                 3
                                 [24]
```

1.4 PC



1.5 CSRs

Number	Privilege	Name	Description
		User Trap Setup	
0x000	URW	ustatus	User status register.
0x004	URW	uie	User interrupt-enable register.
0x005	URW	utvec	User trap handler base address.
		User Trap Handling	
0x040	URW	uscratch	Scratch register for user trap handlers.
0x041	URW	uepc	User exception program counter.
0x042	URW	ucause	User trap cause.
0x043	URW	utval	User bad address or instruction.
0x044	URW	uip	User interrupt pending.
		User Floating-Point CSRs	
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
		User Counter/Timers	
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		⋮	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	URO	timeh	Upper 32 bits of time, RV32I only.
0xC82	URO	instreth	Upper 32 bits of instret, RV32I only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32I only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32I only.
		⋮	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32I only.

Number	Privilege	Name	Description
		Supervisor Trap Setup	
0x100	SRW	sstatus	Supervisor status register.
0x102	SRW	sedeleg	Supervisor exception delegation register.
0x103	SRW	sideleg	Supervisor interrupt delegation register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
		Supervisor Trap Handling	
0x140	SRW	sscratch	Scratch register for supervisor trap handlers.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor bad address or instruction.
0x144	SRW	sip	Supervisor interrupt pending.
		Supervisor Protection and Translation	
0x180	SRW	satp	Supervisor address translation and protection.

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
Machine Trap Handling			
0x340	MRW	mscratch	Scratch register for machine trap handlers.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine bad address or instruction.
0x344	MRW	mip	Machine interrupt pending.
Machine Protection and Translation			
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		⋮	
0x3BF	MRW	pmpaddr15	Physical memory protection address register.

Number	Privilege	Name	Description
Machine Counter/Timers			
0xB00	MRW	mcycle	Machine cycle counter.
0xB02	MRW	minstret	Machine instructions-retired counter.
0xB03	MRW	mhpmcounter3	Machine performance-monitoring counter.
0xB04	MRW	mhpmcounter4	Machine performance-monitoring counter.
		⋮	
0xB1F	MRW	mhpmcounter31	Machine performance-monitoring counter.
0xB80	MRW	mcycleh	Upper 32 bits of mcycle, RV32I only.
0xB82	MRW	minstreth	Upper 32 bits of minstret, RV32I only.
0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter3, RV32I only.
0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter4, RV32I only.
		⋮	
0xB9F	MRW	mhpmcounter31h	Upper 32 bits of mhpmcounter31, RV32I only.
Machine Counter Setup			
0x323	MRW	mhpmevent3	Machine performance-monitoring event selector.
0x324	MRW	mhpmevent4	Machine performance-monitoring event selector.
		⋮	
0x33F	MRW	mhpmevent31	Machine performance-monitoring event selector.
Debug/Trace Registers (shared with Debug Mode)			
0x7A0	MRW	tselect	Debug/Trace trigger register select.
0x7A1	MRW	tdata1	First Debug/Trace trigger data register.
0x7A2	MRW	tdata2	Second Debug/Trace trigger data register.
0x7A3	MRW	tdata3	Third Debug/Trace trigger data register.
Debug Mode Registers			
0x7B0	DRW	dcsr	Debug control and status register.
0x7B1	DRW	dpc	Debug PC.
0x7B2	DRW	dscratch	Debug scratch register.

2 RISCV32-IMACF Instructions

2.1 RISCV32-I

2.1.1 Load and Store Instructions

2.1.1.1 lb rd, offset(rs1)

- x[rd] = sext(M[x[rs1] + sext(offset)][7:0])
- Load Byte.
- I-type, RV32I and RV64I.
- Read one byte from the address x[rs1] + sign-extend(offset), and write x[rd] after the sign bit extension.

31	20 19	15 14	12 11	7 6	0

offset[11:0]	rs1	000	rd	0000011

2.1.1.2 lbu rd, offset(rs1)

- x[rd] = M[x[rs1] + sext(offset)][7:0]
- Load Byte, Unsigned.
- I-type, RV32I and RV64I.
- Read one byte from the address x[rs1] + sign-extend(offset) and write it to x[rd] after zero extension.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	100	rd	0000011

2.1.1.3 lh rd, offset(rs1)

- x[rd] = sext(M[x[rs1] + sext(offset)][15:0])
- Load Halfword.
- I-type, RV32I and RV64I.
- Read two bytes from the address x[rs1] + sign-extend(offset), and write x[rd] after the sign bit extension.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	001	rd	0000011

2.1.1.4 lhu rd, offset(rs1)

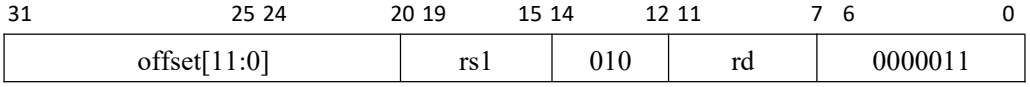
- x[rd] = M[x[rs1] + sext(offset)][15:0]
- Load Halfword, Unsigned.
- I-type, RV32I and RV64I.
- Read two bytes from address x[rs1] + sign-extend(offset) and write x[rd] after zero extension.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	101	rd	0000011

2.1.1.5 lw rd, offset(rs1)

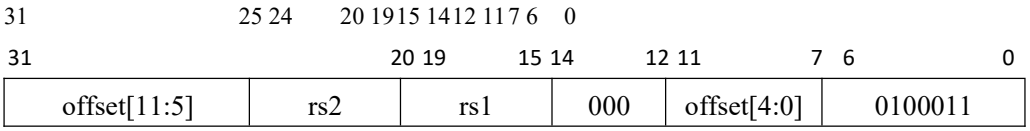
- x[rd] = sext(M[x[rs1] + sext(offset)][31:0])
- Load Word.
- I-type, RV32I and RV64I.

- Read four bytes from the address x[rs1] + sign-extend(offset) and write x[rd]. For RV64I, the result is sign extended.
- Compressed form： c.lwsp rd, offset; c.lw rd, offset(rs1)



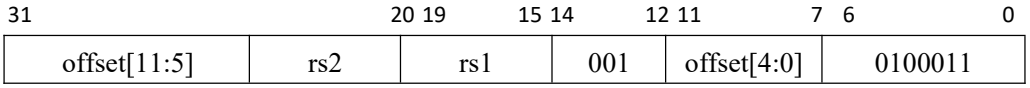
2.1.1.6 sb rs2, offset(rs1)

- M[x[rs1] + sext(offset)] = x[rs2][7: 0]
- Store Byte.
- S-type, RV32I and RV64I.
- Store the low byte of x[rs2] into the memory address x[rs1]+sign-extend(offset).



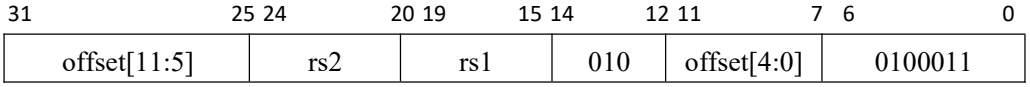
2.1.1.7 sh rs2, offset(rs1)

- M[x[rs1] + sext(offset)] = x[rs2][15: 0]
- Store Halfword.
- S-type, RV32I and RV64I.
- Store the lower 2 bytes of x[rs2] into the memory address x[rs1]+sign-extend(offset).



2.1.1.8 sw rs2, offset(rs1)

- M[x[rs1] + sext(offset)] = x[rs2][31: 0]
- Store Word.
- S-type, RV32I and RV64I.
- Store the lower 4 bytes of x[rs2] into the memory address x[rs1]+sign-extend(offset).
- Compressed form： c.swsp rs2, offset; c.sw rs2, offset(rs1)



2.1.2 Integer Computational Instructions

2.1.2.1 sll rd, rs1, rs2

- x[rd] = x[rs1] ≪ x[rs2]
- Shift Left Logical.
- R-type, RV32I and RV64I.
- Shift the register x[rs1] to the left by x[rs2] bits, fill the vacant position with 0, and write the result to x[rd]. The lower 5 bits of x[rs2] (or the lower 6 bits in the case of RV64I) represent the number of shift bits, and the higher bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	001	rd	0110011

2.1.2.2 slli rd, rs1, shamt

- x[rd] = x[rs1] ≪ shamt
- Shift Left Logical Immediate.
- I-type, RV32I and RV64I.
- Shift the register x[rs1] to the left by the shamt bit, fill the vacant position with 0, and write the result to x[rd]. For RV32I, the instruction is valid only when shamt[5]=0.
- Compressed form： c.slli rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	001	rd	0010011

2.1.2.3 srl rd, rs1, rs2

- x[rd] = (x[rs1] $\gg_u$ x[rs2])
- Shift Right Logical.
- R-type, RV32I and RV64I.
- Shift the register x[rs1] to the right by x[rs2] bits, fill the vacant position with 0, and write the result to x[rd]. The lower 5 bits of x[rs2] (or the lower 6 bits in the case of RV64I) represent the number of shift bits, and the higher bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	101	rd	0110011

2.1.2.4 srli rd, rs1, shamt

- x[rd] = (x[rs1] $\gg_u$ shamt)
- Shift Right Logical Immediate.
- I-type, RV32I and RV64I.
- Shift the register x[rs1] to the right by the shamt bit, fill the vacant position with 0, and write the result to x[rd]. For RV32I, the instruction is valid only when shamt[5]=0.
- Compressed form： c.srli rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	101	rd	0010011

2.1.2.5 sra rd, rs1, rs2

- x[rd] = (x[rs1] $\gg_s$ x[rs2])
- Shift Right Arithmetic.
- R-type, RV32I and RV64I.
- Shift the register x[rs1] to the right by x[rs2] bits, fill the empty bits with the highest bit of x[rs1], and write the result to x[rd]. The lower 5 bits of x[rs2] (or the lower 6 bits in the case of RV64I) are the shift bits, and the higher bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0110011

2.1.2.6 srai rd, rs1, shamt

- x[rd] = (x[rs1] $\gg_s$ shamt)
- Shift Right Arithmetic Immediate.
- I-type, RV32I and RV64I.
- Shift the register x[rs1] right by shamt bits, fill the empty bits with the highest bit of x[rs1], and write the result to x[rd]. For RV32I, the instruction is valid only when shamt[5]=0.
- Compressed form： c.srai rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
010000	shamt	rs1	101	rd	0010011

2.1.2.7 add rd, rs1, rs2

- x[rd] = x[rs1] + x[rs2]
- Add.

- R-type, RV32I and RV64I.
- Add register x[rs2] to register x[rs1], and write the result to x[rd]. Ignore arithmetic overflow.
- Compressed form： c.add rd, rs2; c.mv rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0110011

2.1.2.8 addi rd, rs1, immediate

- x[rd] = x[rs1] + sext(immediate)
- Add Immediate.
- I-type, RV32I and RV64I.
- Add the sign-extended immediate value to the register x[rs1], and write the result to x[rd]. Ignore arithmetic overflow.
- Compressed form： c.li rd, imm; c.addi rd, imm; c.addi16sp imm; c.addi4spn rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0010011

2.1.2.9 sub rd,rs1, rs2

- x[rd] = x[rs1] − x[rs2]
- Substract.
- R-type, RV32I and RV64I.
- Subtract x[rs2] from x[rs1] and write the result to x[rd]. Ignore arithmetic overflow.
- Compressed form： c.sub rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0110011

2.1.2.10 lui rd, immediate

- x[rd] = sext(immediate[31:12] << 12)
- Load Upper Immediate.
- U-type, RV32I and RV64I.
- Shift the sign-extended 20-bit immediate data by 12 bits to the left, and write the low 12 bits to x[rd].
- Compressed form： c.lui rd, imm

31 12	11 7	6 0
immediate[31:12]	rd	0110111

2.1.2.11 auipc rd, immediate

- x[rd] = pc + sext(immediate[31:12] << 12)
- Add Upper Immediate to PC.
- U-type, RV32I and RV64I.
- Add the signed 20-bit (left shifted 12-bit) immediate value to the pc, and write the result to x[rd].

31 12	11 7	6 0
immediate[31:12]	rd	0010111

2.1.2.12 and rd, rs1, rs2

- x[rd] = x[rs1] & x[rs2]
- And.
- R-type, RV32I and RV64I.
- Take the bitwise AND of register x[rs1] and register x[rs2],and write the result to x[rd].
- Compressed form： c.and rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	111	rd	0110011

2.1.2.13 andi rd, rs1, immediate

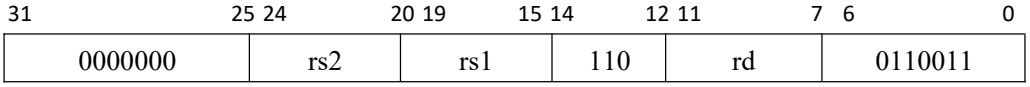
- x[rd] = x[rs1] & sext(immediate)
- And Immediate.
- I-type, RV32I and RV64I.
- Take the bitwise AND of register x[rs1] and the sign-extended immediate value,and write the result to x[rd].
- Compressed form： c.andi rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	111	rd	0010011

2.1.2.14 or rd, rs1, rs2

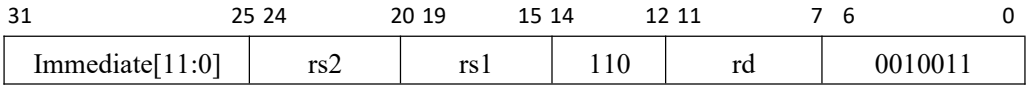
- x[rd] = x[rs1] | x[rs2]
- OR.
- R-type, RV32I and RV64I.
- Take the bitwise OR of register x[rs1] and register x[rs2], and write the result to x[rd].

- Compressed form：c.or rd, rs2



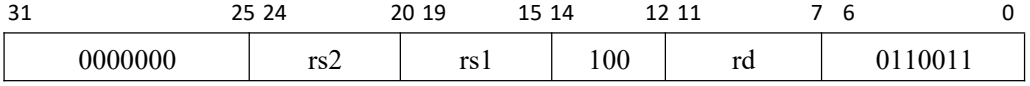
2.1.2.15 ori rd, rs1, immediate

- x[rd] = x[rs1] | sext(immediate)
- OR Immediate.
- R-type, RV32I and RV64I.
- Take the bitwise OR of register x[rs1] and the sign-extended immediate value,and write the result to x[rd].
- Compressed form：c.or rd, rs2



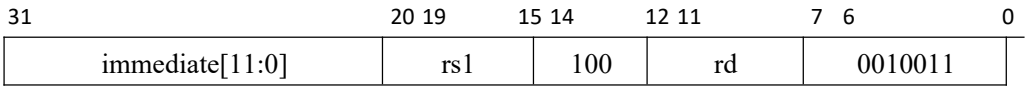
2.1.2.16 xor rd, rs1, rs2

- x[rd] = x[rs1] ^ x[rs2]
- Exclusive-OR.
- R-type, RV32I and RV64I.
- Take the bitwise XOR of register x[rs1] and register x[rs2], and write the result to x[rd].
- Compressed form：c.xor rd, rs2



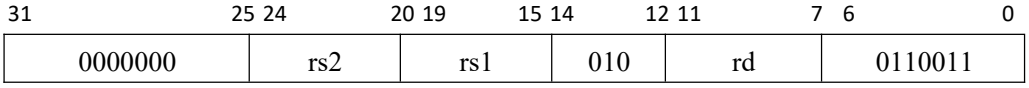
2.1.2.17 xori rd, rs1, immediate

- x[rd] = x[rs1] ^ sext(immediate)
- Exclusive-OR Immediate.
- I-type, RV32I and RV64I.
- Take the bitwise XOR of register x[rs1] and the sign-extended immediate value,and write the result to x[rd].
- Compressed form：c.xor rd, rs2



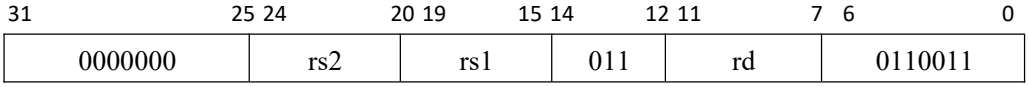
2.1.2.18 slt rd, rs1, rs2

- x[rd] = (x[rs1] $<_s$ x[rs2])
- Set if Less Than.
- R-type, RV32I and RV64I.
- Compare the numbers in x[rs1] and x[rs2], if x[rs1] is smaller, write 1 to x[rd], otherwise write 0.



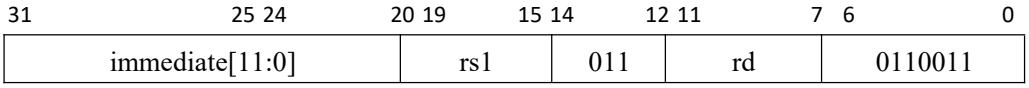
2.1.2.19 sltu rd, rs1, rs2

- x[rd] = (x[rs1] $<_u$ x[rs2])
- Set if Less Than, Unsigned.
- R-type, RV32I and RV64I.
- Compare x[rs1] and x[rs2] and treat them as unsigned numbers. If x[rs1] is smaller, write 1 to x[rd], otherwise write 0.



2.1.2.20 slti rd, rs1, immediate

- x[rd] = (x[rs1] $<_s$ sext(immediate))
- Set if Less Than Immediate.
- I-type, RV32I and RV64I.
- Compare x[rs1] and sign-extended immediate. If x[rs1] is smaller, write 1 to x[rd], otherwise write 0.



2.1.2.21 sltiu rd, rs1, immediate

- x[rd] = (x[rs1] $<_u$ sext(immediate))
- Set if Less Than Immediate, Unsigned.
- I-type, RV32I and RV64I.
- Compare x[rs1] with signed-extended immediate, and treat it as an unsigned number during the comparison. If x[rs1] is smaller, write 1 to x[rd], otherwise write 0.

31 25	24 20	19 15	14 12	11 7	6 0
immediate[11:0]		rs1	011	rd	0010011

2.1.3 Control Transfer Instructions

2.1.3.1 beq rs1, rs2, offset

- if (rs1 == rs2) pc += sext(offset)
- Branch if Equal.
- B-type, RV32I and RV64I.
- If the values of register x[rs1] and register x[rs2] are equal, the value of pc is set to the current value plus the sign-extension offset.
- Compressed form： c.beqz rs1, offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	000	offset[4:1|11]	1100011

2.1.3.2 bne rs1, rs2, offset

- if (rs1 ≠ rs2) pc += sext(offset)
- Branch if Not Equal.
- B-type, RV32I and RV64I.
- If the values of register x[rs1] and register x[rs2] are not equal, the value of pc is set to the current value plus the sign-extended offset.
- Compressed form： c.bnez rs1, offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	001	offset[4:1|11]	1100011

2.1.3.3 blt rs1, rs2, offset

- if (rs1 $<_s$ rs2) pc += sext(offset)
- Branch if Less Than.
- B-type, RV32I and RV64I.
- If the value of register x[rs1] is smaller than the value of register x[rs2] (both regarded as twos complement), the value of pc is set to the current value plus the sign-extension offset.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	100	offset[4:1|11]	1100011

2.1.3.4 bge rs1, rs2, offset

- if (rs1 $\geqslant_s$ rs2) pc += sext(offset)
- Branch if Greater Than or Equal.
- B-type, RV32I and RV64I.
- If the value of register x[rs1] is greater than or equal to the value of register x[rs2] (all regarded as twos complement), the value of pc is set to the current value plus the sign-extension offset.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	101	offset[4:1|11]	1100011

2.1.3.5 bltu rs1, rs2, offset

- if (rs1 $<_u$ rs2) pc += sext(offset)
- Branch if Less Than, Unsigned.
- B-type, RV32I and RV64I.
- If the value of register x[rs1] is smaller than the value of register x[rs2] (all regarded as unsigned numbers), the value of pc is set to the current value plus the sign-extension offset.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	110	offset[4:1|11]	1100011

2.1.3.6 bgeu rs1, rs2, offset

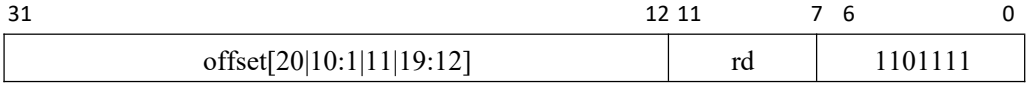
- if (rs1 $\geqslant_u$ rs2) pc += sext(offset)
- Branch if Greater Than or Equal, Unsigned.
- B-type, RV32I and RV64I.
- If the value of register x[rs1] is greater than or equal to the value of register x[rs2] (all regarded as unsigned numbers), the value of pc is set to the current value plus the sign-extension offset.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	111	offset[4:1|11]	1100011

2.1.3.7 jal rd, offset

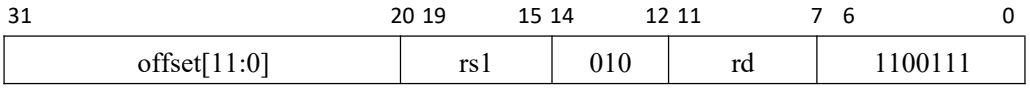
- x[rd] = pc+4; pc += sext(offset)
- Jump and Link.
- J-type, RV32I and RV64I.

- Set the address of the next instruction (pc+4), and then set pc to the current value plus the sign-extended offset. rd defaults to x1.
- Compressed form： c.j offset; c.jal offset



2.1.3.8 jalr rd, offset(rs1)

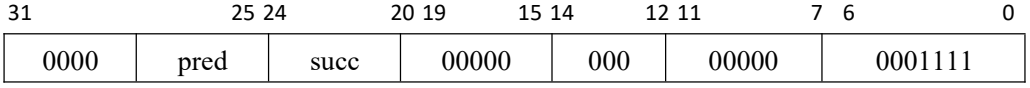
- t =pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t
- Jump and Link Register.
- I-type, RV32I and RV64I.
- Set pc to x[rs1] + sign-extend(offset), set the least significant bit of the calculated address to 0, and write the original pc+4 value to f[rd]. rd defaults to x1.
- Compressed form： c.jr rs1; c.jalr rs1



2.1.4 Misc-mem instructions

2.1.4.1 fence pred, succ

- Fence(pred, succ)
- Fence Memory and I/O.
- I-type, RV32I and RV64I.
- Before the memory and I/O accesses in subsequent instructions are visible to the outside (such as other threads), make the memory and I/O accesses before this instruction visible to the outside. Bits 3, 2, 1, and 0 of the bits correspond to device input, device output, and memory read and write respectively. For example, fence r, rw, sort the previous reads and the subsequent reads and writes, using pred = 0010 and succ = 0011 for encoding. If the parameter is omitted, it means fence iorw, iorw, that is, sort all memory fetch requests.



2.1.4.2 fence.i

- Fence(Store, Fetch)
- Fence Instruction Stream.
- I-type, RV32I and RV64I.

- Make the read and write to the memory instruction area visible to subsequent instruction fetches.

31 20	19 15	14 12	11 7	6 0
000000000000	00000	001	00000	0001111

2.1.4.3 sfence.vma rs1, rs2

- Fence(Store, AddressTranslation)
- Fence Virtual Memory.
- R-type, RV32I and RV64I , Privilege architecture
- Sort the previous page table storage according to the subsequent virtual address translation. When rs2=0, the translation of all address spaces will be affected; otherwise, only the translation of the address space identified by x[rs2] will be sorted. When rs1=0, the translations of all virtual addresses in the selected address space are sorted; otherwise, only the page address translations containing the virtual address x[rs1] are sorted.

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	000	00000	1110011

2.1.5 Control and Status Register Instructions

2.1.5.1 csrrw rd, csr, zimm[4:0]

- t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t
- Control and Status Register Read and Write.
- I-type, RV32I and RV64I.
- Mark the value in the control status register csr as t. Write the value of register x[rs1] into csr, and then write t into x[rd].

31 20	19 15	14 12	11 7	6 0
csr	rs1	001	rd	1110011

2.1.5.2 csrrs rd, csr, rs1

- t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t
- Control and Status Register Read and Set.
- I-type, RV32I and RV64I.
- Mark the value in the control status register csr as t. Write the result of bitwise OR of t and register x[rs1] to csr, then write t into x[rd].

31 20	19 15	14 12	11 7	6 0
csr	rs1	010	rd	1110011

2.1.5.3 csrrc rd, csr, rs1

- t = CSRs[csr]; CSRs[csr] = t &~x[rs1]; x[rd] = t
- Control and Status Register Read and Clear.
- I-type, RV32I and RV64I.
- Mark the value in the control status register csr as t. Write the result of bitwise AND of t and register x[rs1] into csr, and then write t into x[rd].

31 20	19 15	14 12	11 7	6 0
csr	rs1	011	rd	1110011

2.1.5.4 csrrwi rd, csr, zimm[4:0]

- x[rd] = CSRs[csr]; CSRs[csr] = zimm
- Control and Status Register Read and Write Immediate.
- I-type, RV32I and RV64I.
- Copy the value in the control status register csr to x[rd], and then write the value of the five-bit zero-extended immediate zimm to csr.

31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	101	rd	1110011

2.1.5.5 csrrsi rd, csr, zimm[4:0]

- t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t
- Control and Status Register Read and Set Immediate.
- I-type, RV32I and RV64I.
- Mark the value in the control status register csr as t. Write the bitwise OR result of t and the five-digit zero-extended immediate value zimm into csr, and then write t into x[rd] (the 5th and higher bits of the csr register remain unchanged).

31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	110	rd	1110011

2.1.5.6 csrrci rd, csr, zimm[4:0]

- t = CSRs[csr]; CSRs[csr] = t &~zimm; x[rd] = t
- Control and Status Register Read and Clear Immediate.
- I- type, RV32I and RV64I.
- Mark the value in the control status register csr as t. Write the bitwise AND result of t and the five-digit zero-extended immediate value zimm into csr, and then write t into x[rd] (the 5th and higher bits of the csr register remain unchanged).

31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	111	rd	1110011

2.1.6 Environment Call and Breakpoints

2.1.6.1 ecall

- RaiseException(EnvironmentCall)
- Environment Call.
- I-type, RV32I and RV64I.
- Request the execution environment by raising an environment call exception.

31 20	19 15	14 12	11 7	6 0
000000000000	00000	000	00000	1110011

2.1.6.2 ebreak

- RaiseException(Breakpoint)
- Environment Breakpoint.
- I-type, RV32I and RV64I.
- Request the debugger by throwing a breakpoint exception.

31 20	19 15	14 12	11 7	6 0
000000000001	00000	000	00000	1110011

2.1.7 Trap-Return Instructions

2.1.7.1 mret

- ExceptionReturn(Machine)

- Machine-mode Exception Return.
- R-type, RV32I and RV64I, Privilege architecture
- Return from the machine mode exception handler. Set pc to CSRs[mepc], set the privilege level to CSRs[mstatus].MPP, CSRs[mstatus].MIE to CSRs[mstatus].MPIE, and set CSRs[mstatus].MPIE to 1; and if To support user mode, set CSR [mstatus].MPP to 0.

31 25	24 20	19 15	14 12	11 7	6 0
0011000	00010	00000	000	00000	1110011

2.1.7.2 sret

- ExceptionReturn(Supervisor)
- Supervisor-mode Exception Return.
- R-type, RV32I and RV64I, Privilege architecture
- Return from the exception handler in the administrator mode, set pc to CSRs[spec], permission mode to CSRs[sstatus].SPP, CSRs[sstatus].SIE to CSRs[sstatus].SPIE, and CSRs[sstatus].SPIE to 1. CSRs[sstatus].spp is 0.

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00010	00000	000	00000	1110011

2.1.8 Wait for Interrupt

2.1.8.1 wfi

- while (noInterruptPending) idle wait for Interrupt.
- R-type, RV32I and RV64I, Privilege architecture
- If there are no pending interrupts, the processor is placed in an idle state.

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00101	00000	000	00000	1110011

2.2 RISCV32-M

2.2.1 mul rd, rs1, rs2

- x[rd] = x[rs1] × x[rs2]
- Multiply.
- R-type, RV32M and RV64M.
- Multiply the register x[rs2] to the register x[rs1], and write the product to x[rd]. Ignore

arithmetic overflow.

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	000	rd	0110011

2.2.2 mulh rd, rs1, rs2

- x[rd] = (x[rs1] $_s\times_s$ x[rs2]) $\gg_s$ XLEN
- Multiply High.
- R-type, RV32M and RV64M.
- Multiplying the register x[rs2] to the register x[rs1] is regarded as 2's complement, and the high bit of the product is written to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	001	rd	0110011

2.2.3 mulhsu rd,rs1, rs2

- x[rd] = (x[rs1] $_s\times_u$ x[rs2]) $\gg_s$ XLEN
- Multiply High Signed-Unsigned.
- R-type, RV32M and RV64M.
- Multiply the register x[rs2] to the register x[rs1], x[rs1] is 2's complement, x[rs2] is an unsigned number, write the high bit of the product to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	010	rd	0110011

2.2.4 mulhu rd, rs1, rs2

- x[rd] = (x[rs1] $_u\times_u$ x[rs2]) $\gg_u$ XLEN
- Multiply High Unsigned.
- R-type, RV32M and RV64M.
- Multiply the register x[rs2] to the register x[rs1]. Both x[rs1] and x[rs2] are unsigned numbers. Write the high bit of the product to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	011	rd	0110011

2.2.5 div rd, rs1, rs2

- x[rd] = x[rs1] $\div_s$ x[rs2]

- Divide.
- R-type, RV32M and RV64M.
- Divide the value of register x[rs1] by the value of register x[rs2], round to zero, treat these numbers as two's complement, and write the quotient to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	100	rd	0110011

2.2.6 divu rd, rs1, rs2

- x[rd] = x[rs1] $\div_u$ x[rs2]
- Divide, Unsigned.
- R-type, RV32M and RV64M.
- Divide the value of register x[rs1] by the value of register x[rs2], round to zero, treat these numbers as unsigned numbers, and write the quotient to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0110011

2.2.7 rem rd, rs1, rs2

- x[rd] = x[rs1] $\%_s$ x[rs2]
- Remainder.
- R-type, RV32M and RV64M.
- Dividing x[rs1] by x[rs2] and rounding to 0 are regarded as 2's complement, and the remainder is written into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	110	rd	0110011

2.2.8 remu rd, rs1, rs2

- x[rd] = x[rs1] $\%_u$ x[rs2]
- Remainder, Unsigned.
- R-type, RV32M and RV64M.
- Dividing x[rs1] by x[rs2] and rounding to 0 are regarded as unsigned numbers, and the remainder is written into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	111	rd	0110011

2.3 RISCV32-C

2.3.1 Load and Store Instructions

2.3.1.1 c.lw rd', uimm(rs1')

- x[8+rd’] = sext(M[x[8+rs1’] + uimm][31:0])
- Load Word.
- RV32IC and RV64IC.
- The extended form is lw rd, uimm(rs1), where rd=8+rd’, rs1=8+rs1’.

15 13	12	11 7	6 5	4 2	1 0
010	uimm[5:3]	rs1’	uimm[2|6]	rd’	00

2.3.1.2 c.sw rs2', uimm(rs1')

- M[x[8+rs1’] + uimm][31:0] = x[8+rs2’]
- Store Word.
- RV32IC and RV64IC.
- The extended form is sw rs2, uimm(rs1), where rs2=8+rs2’, rs1=8+rs1’.

15 13	12	11 7	6 5	4 2	1 0
110	uimm[5:3]	rs1’	uimm[2|6]	rd’	00

2.3.1.3 c.lwsp rd, uimm(x2)

- x[rd] = sext(M[x[2] + uimm][31:0])
- Load Word, Stack-Pointer Relative.
- RV32IC and RV64IC.
- The extended form is lw rd, uimm(x2). It is illegal when rd=x0.

15 13	12	11 7	6 2	1 0
010	uimm[5]	rd	uimm[4:2|7:6]	10

2.3.1.4 c.swsp rs2, uimm(x2)

- M[x[2] + uimm][31:0] = x[rs2]
- Store Word, Stack-Pointer Relative.

- RV32IC and RV64IC.
- The extended form is sw rs2, uimm(x2).

15 13	12 11 7	6 2	1 0
110	uimm[5:2|7:6]	rs2	10

2.3.2 Control Transfer Instructions

2.3.2.1 c.beqz rs1', offset

- if (x[8+rs1'] == 0) pc += sext(offset)
- Branch if Equal to Zero.
- RV32IC and RV64IC.
- The extended form is beq rs1, x0, offset, where rs1=8+rs1'.

15 13	12 10	9 7	6 2	1 0
110	offset[8|4:3]	rs1'	offset[7:6|2:1|5]	01

2.3.2.2 c.bnez rs1', offset

- if (x[8+rs1'] ≠ 0) pc += sext(offset)
- Branch if Not Equal to Zero.
- RV32IC and RV64IC.
- The extended form is bne rs1, x0, offset, where rs1=8+rs1'.

15 13	12 10	9 7	6 2	1 0
111	offset[8|4:3]	rs1'	offset[7:6|2:1|5]	01

2.3.2.3 c.j offset

- pc += sext(offset)
- Jump.
- RV32IC and RV64IC.
- The extended form is jal x0, offset.

15 13	12 2	1 0
101	offset[11|4|9:8|10|6|7|3:1|5]	01

2.3.2.4 c.jr rs1

- pc = x[rs1]
- Jump Register.
- RV32IC and RV64IC.
- The extended form is jalr x0, 0(rs1). It is illegal when rs1=x0.

15 13	12	11 7	6 2	1 0
100	0	rs1	00000	10

2.3.2.5 c.jal offset

- x[1] = pc+2; pc += sext(offset)
- Jump and Link.
- RV32IC.
- The extended form is jal x1, offset.

15 13	12 2	1 0
001	offset[11|4|9:8|10|6|7|3:1|5]	01

2.3.2.6 c.jalr rs1

- t = pc+2; pc = x[rs1]; x[1] = t
- Jump and Link Register.
- RV32IC and RV64IC.
- The extended form is jalr x1, 0(rs1). It is illegal when rs1=x0.

15 13	12	11 7	6 2	1 0
100	1	rs1	00000	10

2.3.2.7 c.ebreak

- RaiseException(Breakpoint)
- Environment Breakpoint.
- RV32IC and RV64IC.
- The extended form is ebreak.

15 13	12	11 7	6 2	1 0
100	1	00000	00000	10

2.3.3 Integer Computational Instructions

2.3.3.1 c.add rd, rs2

- x[rd] = x[rd] + x[rs2]
- Add.
- RV32IC and RV64IC.
- The extended form is add rd, rd, rs2. It is illegal when rd=x0 or rs2=x0.

15 13	12	11 7	6 2	1 0
100	1	rd	rs2	10

2.3.3.2 c.addi rd, imm

- x[rd] = x[rd] + sext(imm)
- Add Immediate. RV32IC and RV64IC.
- The extended form is addi rd, rd, imm.

15 13	12	11 7	6 2	1 0
000	imm[5]	rd	imm[4:0]	01

2.3.3.3 c.sub rd', rs2'

- x[8+rd’] = x[8+rd’] - x[8+rs2’]
- Subtract.
- RV32IC and RV64IC.
- The extended form is sub rd, rd, rs2. Among them, rd=8+rd’, rs2=8+rs2’.

15 10	9 7	6 5	4 2	1 0
100011	rd’	00	rs2’	01

2.3.3.4 c.addi16sp imm

- x[2] = x[2] + sext(imm)
- Add Immediate, Scaled by 16, to Stack Pointer.
- RV32IC and RV64IC.
- The extended form is addi x2, x2, imm. It is illegal when imm=0.

15 13	12	11 7	6 2	1 0

011	imm[9]	00010	imm[4|6|8:7|5]	01

2.3.3.5 c.addi4spn rd', uimm

- x[8+rd’] = x[2] + uimm
- Add Immediate, Scaled by 4, to Stack Pointer, Nondestructive.
- RV32IC and RV64IC.
- The extended form is addi rd, x2, uimm, where rd=8+rd’. It is illegal when uimm=0.

15 13	12 5	4 2	1 0
000	uimm[5:4|9:6|2|3]	rd’	00

2.3.3.6 c.and rd', rs2'

- x[8+rd’] = x[8+rd’] & x[8+rs2’]
- AND.
- RV32IC and RV64IC.
- The extended form is and rd, rd, rs2, where rd=8+rd’, rs2=8+rs2’.

15 10	9 7	6 5	4 2	1 0
100011	rd’	11	rs2’	01

2.3.3.7 c.andi rd', imm

- x[8+rd’] = x[8+rd’] & sext(imm)
- AND Immediate.
- RV32IC and RV64IC.
- The extended form is andi rd, rd, imm, where rd=8+rd’.

15 13	12	11 10	9 7	6 2	1 0
100	imm[5]	10	rd’	imm[4:0]	01

2.3.3.8 c.or rd', rs2'

- x[8+rd’] = x[8+rd’] | x[8+rs2’]
- OR.
- RV32IC and RV64IC.
- The extended form is or rd, rd, rs2, where rd=8+rd’, rs2=8+rs2’.

15 10	9 7	6 5	4 2	1 0
100011	rd’	10	rs2’	01

2.3.3.9 c.xor rd’, rs2’

- x[8+rd’] = x[8+rd’] ^ x[8+rs2’]
- Exclusive-OR.
- RV32IC and RV64IC.
- The extended form is xor rd, rd, rs2, where rd=8+rd’, rs2=8+rs2’.

15 10	9 7	6 5	4 2	1 0
100011	rd’	01	rs2’	01

2.3.3.10 c.slli rd, uimm

- x[rd] = x[rd] << uimm
- Shift Left Logical Immediate.
- RV32IC and RV64IC.
- The extended form is slli rd, rd, uimm.

15 13	12	11 7	6 2	1 0
000	uimm[5]	rd	uimm[4:0]	10

2.3.3.11 c.srli rd’, uimm

- x[8+rd’] = x[8+rd’] >>u uimm
- Shift Right Logical Immediate.
- RV32IC and RV64IC.
- The extended form is srli rd, rd, uimm, where rd=8+rd’.

15 13	12	11 10	9 7	6 2	1 0
100	uimm[5]	00	rd’	uimm[4:0]	01

2.3.3.12 c.srai rd’, uimm

- x[8+rd’] = x[8+rd’] >>s uimm
- Shift Right Arithmetic Immediate.
- RV32IC and RV64IC.
- The extended form is srai rd, rd, uimm, where rd=8+rd’.

15 13	12	11 10	9 7	6 2	1 0
100	uimm[5]	01	rd’	uimm[4:0]	01

2.3.3.13 c.mv rd, rs2

- x[rd] = x[rs2]
- Move.
- RV32IC and RV64IC.
- The extended form is add rd, x0, rs2. It is illegal when rs2=x0.

15 13	12	11 7	6 2	1 0
100	0	rd	rs2	10

2.3.3.14 c.li rd, imm

- x[rd] = sext(imm)
- Load Immediate.
- RV32IC and RV64IC.
- The extended form is addi rd, x0, imm.

15 13	12	11 7	6 2	1 0
010	imm[5]	rd	imm[4:0]	01

2.3.3.15 c.lui rd, imm

- x[rd] = sext(imm[17:12] << 12)
- Load Upper Immediate.
- RV32IC and RV64IC.
- The extended form is lui rd, imm. It is illegal when rd=x2 or imm=0.

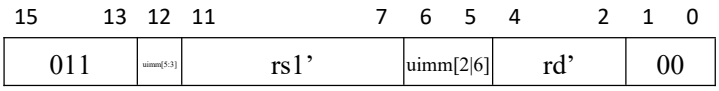
15 13	12	11 7	6 2	1 0
011	imm[17]	rd	imm[16:12]	01

2.3.4 Floating Point Instructructions

2.3.4.1 c.flw rd’, uimm(rs1’)

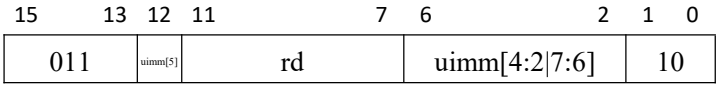
- f[8+rd’] = M[x[8+rs1’] + uimm][31:0]
- Floating-point Load Word.

- RV32FC.
- The extended form is flw rd, uimm(rs1), where rd=8+rd’, rs1=8+rs1’.



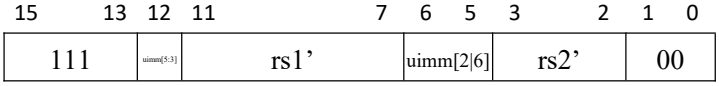
2.3.4.2 c.flwsp rd, uimm(x2)

- f[rd] = M[x[2] + uimm][31:0]
- Floating-point Load Word, Stack-Pointer Relative.
- RV32FC.
- The extended form is flw rd, uimm(x2).



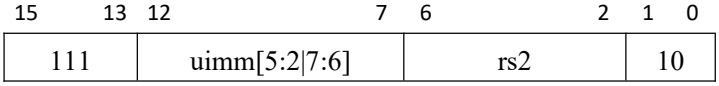
2.3.4.3 c.fsw rs2’, uimm(rs1’)

- M[x[8+rs1’] + uimm][31:0] = f[8+rs2’]
- Floating-point Store Word.
- RV32FC.
- The extended form is fsw rs2, uimm(rs1), where rs2=8+rs2’, rs1=8+rs1’.



2.3.4.4 c.fswsp rs2, uimm(x2)

- M[x[2] + uimm][31:0] = f[rs2]
- Floating-point Store Word, Stack-Pointer Relative.
- RV32FC.
- The extended form is fsw rs2, uimm(x2).

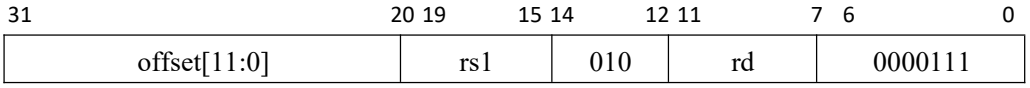


2.4 RISCV-F

2.4.1 Single-Precision Floating-Point Load and Store Instructions

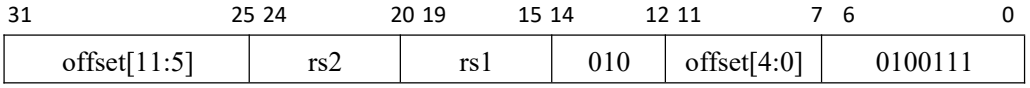
2.4.1.1 flw rd, offset(rs1)

- f[rd] = M[x[rs1] + sext(offset)][31:0]
- Floating-point Load Word.
- I-type, RV32F and RV64F.
- Load a single-precision floating-point number from the memory address x[rs1] + sign-extend(offset) and write it to f[rd].
- Compressed form： c.flwsp rd, offset; c.flw rd, offset(rs1)



2.4.1.2 fsw rs2, offset(rs1)

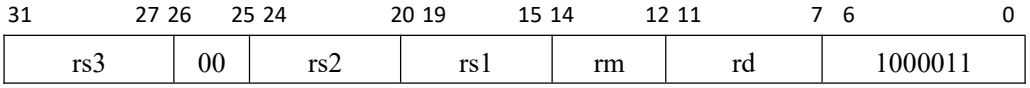
- M[x[rs1] + sext(offset)] = f[rs2][31:0]
- Floating-point Store Word.
- S-type, RV32F and RV64F.
- Store the single-precision floating-point number in the register f[rs2] into the memory address x[rs1] + sign-extend(offset).
- Compressed form： c.fswsp rs2, offset; c.fsw rs2, offset(rs1)



2.4.2 Single-Precision Floating-Point Computational Instructions

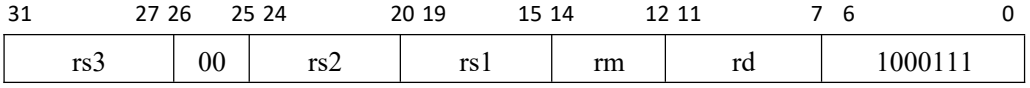
2.4.2.1 fmadd.s rd, rs1, rs2, rs3

- f[rd] = f[rs1]×f[rs2]+f[rs3]
- Floating-point Fused Multiply-Add, Single-Precision.
- R4-type, RV32F and RV64F.
- Multiply the single-precision floating-point numbers in the registers f[rs1] and f[rs2], and add the unrounded product to the single-precision floating-point number in the register f[rs3] to add the rounded single-precision floating-point numbers. The number of points is written into f[rd].



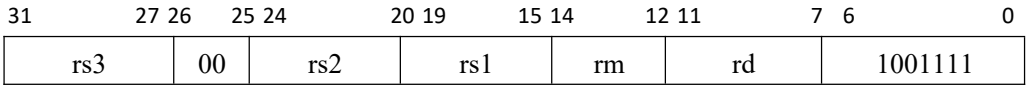
2.4.2.2 fmsub.s rd, rs1, rs2, rs3

- f[rd] = f[rs1]×f[rs2]-f[rs3]
- Floating-point Fused Multiply-Subtarct, Single-Precision.
- R4-type, RV32F and RV64F.
- Multiply the single-precision floating-point numbers in the registers f[rs1] and f[rs2], and subtract the unrounded product from the single-precision floating-point number in the register f[rs3], and write the rounded single-precision floating-point number into f[rd].



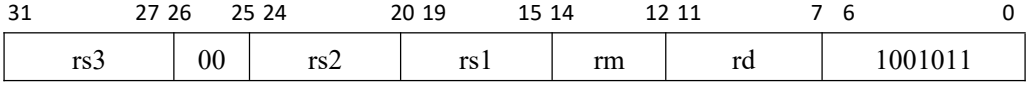
2.4.2.3 fnmadd.s rd, rs1, rs2, rs3

- f[rd] = -f[rs1]_f[rs2]-f[rs3]
- Floating-point Fused Negative Multiply-Add, Single-Precision.
- R4-type, RV32F and RV64F.
- Multiply the single-precision floating-point numbers in the register f[rs1] and f[rs2], invert the result, and add the unrounded product to the single-precision floating-point number in the register f[rs3], and it will be rounded The following single-precision floating-point number is written into f[rd].



2.4.2.4 fnmsub.s rd, rs1, rs2, rs3

- f[rd] = -f[rs1]×f[rs2]+f[rs3]
- Floating-point Fused Negative Multiply-Subtract, Single-Precision.
- R4- type, RV32F and RV64F.
- Multiply the single-precision floating-point numbers in register f[rs1] and f[rs2], invert the result, and subtract the unrounded product from the single-precision floating-point number in register f[rs3]. After rounding the single-precision floating-point number is written to f[rd].



2.4.2.5 fadd.s rd, rs1, rs2

- f [rd] = f [rs1] + f [rs2]
- Floating-point Add, Single-Precision.
- R-type, RV32F and RV64F.
- Add the single-precision floating-point numbers in the registers f[rs1] and f[rs2], and write the rounded sum to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	rm	rd	1010011

2.4.2.6 fsub.s rd,rs1, rs2

- f[rd] = f[rs1] - f[rs2]
- Floating-point Subtract, Single-Precision.
- R-type, RV32F and RV64F.
- Subtract the single-precision floating-point numbers in registers f[rs1] and f[rs2], and write the rounded difference to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000100	rs2	rs1	rm	rd	1010011

2.4.2.7 fmul.s rd, rs1, rs2

- f[rd] = f[rs1] × f[rs2]
- Floating-point Multiply, Single-Precision.
- R-type, RV32F and RV64F.
- Multiply the single-precision floating-point numbers in the registers f[rs1] and f[rs2], and write the rounded single-precision result into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0001000	rs2	rs1	rm	rd	1010011

2.4.2.8 fdiv.s rd, rs1, rs2

- f[rd] = f[rs1] ÷ f[rs2]
- Floating-point Divide, Single-Precision.
- R-type, RV32F and RV64F.
- Divide the single-precision floating-point numbers in registers f[rs1] and f[rs2], and write the rounded quotient to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0001100	rs2	rs1	rm	rd	1010011

2.4.2.9 fsqrt.s rd, rs1, rs2

- f[rd] = √ f[rs1]
- Floating-point Square Root, Single-Precision.
- R-type, RV32F and RV64F.
- Round the square root of the single-precision floating-point number in f[rs1] and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0101100	00000	rs1	rm	rd	1010011

2.4.2.10 fmin.s rd,rs1, rs2

- f[rd] = min(f[rs1], f[rs2])
- Floating-point Minimum, Single-Precision.
- R-type, RV32F and RV64F.
- Write the smaller value of the single-precision floating-point number in the registers f[rs1] and f[rs2] into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010100	rs2	rs1	000	rd	1010011

2.4.2.11 fmax.s rd,rs1, rs2

- f[rd] = max(f[rs1], f[rs2])
- Floating-point Maximum, Single-Precision.
- R-type, RV32F and RV64F.
- Write the larger value of the single-precision floating-point numbers in registers f[rs1] and f[rs2] into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010100	rs2	rs1	001	rd	1010011

2.4.3 Single-Precision Floating-Point Conversion and Move Instructions

2.4.3.1 fcvt.w.s rd, rs1, rs2

- x[rd] = sext(s32f32(f[rs1]))
- Floating-point Convert to Word from Single.
- R-type, RV32F and RV64F.
- Convert the single-precision floating-point number in the register f[rs1] into a 32-bit two's complement integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00000	rs1	rm	rd	1010011

2.4.3.2 fcvt.wu.s rd, rs1, rs2

- x[rd] = sext(u32f32(f[rs1]))
- Floating-point Convert to Unsigned Word from Single.
- R-type, RV32F and RV64F.
- Convert the single-precision floating-point number in the register f[rs1] into a 32-bit unsigned integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00001	rs1	rm	rd	1010011

2.4.3.3 fcvt.s.w rd, rs1, rs2

- f[rd] = f32s32(x[rs1])
- Floating-point Convert to Single from Word.
- R-type, RV32F and RV64F.
- Convert the 32-bit two's complement integer represented by the register x[rs1] into a single-precision floating-point number, and then write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00000	rs1	rm	rd	1010011

2.4.3.4 fcvt.s.wu rd, rs1, rs2

- f[rd] = f32u32(x[rs1])
- Floating-point Convert to Single from Unsigned Word.
- R-type, RV32F and RV64F.

- Convert the 32-bit unsigned integer in the register x[rs1] into a single-precision floating-point number, and then write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00001	rs1	rm	rd	1010011

2.4.3.5 fsgnj.s rd, rs1, rs2

- f[rd] = {f[rs2][31], f[rs1][30:0]}
- Floating-point Sign Inject, Single-Precision.
- R-type, RV32F and RV64F.
- Construct a new single-precision floating-point number with the exponent and significant number of f[rs1] and the sign bit of the sign of f[rs2], and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	000	rd	1010011

2.4.3.6 fsgnjn.s rd, rs1, rs2

- f[rd] = {~f[rs2][31], f[rs1][30:0]}
- Floating-point Sign Inject-Negate, Single-Precision.
- R-type, RV32F and RV64F.
- Construct a new single-precision floating-point number with the exponent and significant number of f[rs1] and the inverse of the sign bit of the f[rs2], and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	001	rd	1010011

2.4.3.7 fsgnjx.s rd, rs1, rs2

- f[rd] = {f[rs1][31] ^ f[rs2][31], f[rs1][30:0]}
- Floating-point Sign Inject-XOR, Single-Precision.
- R-type, RV32F and RV64F.
- Construct a new single-precision floating-point number with the exponent and significant number of f[rs1] and the XOR of the sign bit of the f[rs2] and the sign bit of the f[rs1], and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	010	rd	1010011

2.4.3.8 fmv.x.w rd, rs1, rs2

- x[rd] = sext(f[rs1][31:0])
- Floating-point Move Word to Integer.
- R-type, RV32F and RV64F.
- Copy the single-precision floating-point number in the register f[rs1] to x[rd]. For RV64F, sign-extend the result.

31 25	24 20	19 15	14 12	11 7	6 0
1110000	00000	rs1	000	rd	1010011

2.4.3.9 fmv.w.x rd, rs1, rs2

- f[rd] = x[rs1][31:0]
- Floating-point Move Word from Integer.
- R-type, RV32F and RV64F.
- Copy the single-precision floating-point number in the register x[rs1] to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1111000	00000	rs1	000	rd	1010011

2.4.4 Single-Precision Floating-Point Compare Instructions

2.4.4.1 feq.s rd, rs1, rs2

- x[rd] = f[rs1] == f[rs2]
- Floating-point Equals, Single-Precision.
- R-type, RV32F and RV64F.
- If the single-precision floating-point numbers in the registers f[rs1] and f[rs2] are equal, write 1 in x[rd], otherwise write 0.

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	010	rd	1010011

2.4.4.2 flt.s rd, rs1, rs2

- x[rd] = f[rs1] < f[rs2]
- Floating-point Less Than, Single-Precision.
- R-type, RV32F and RV64F.
- If the single-precision floating-point number in register f[rs1] is less than the single-precision

floating-point number in f[rs2], write 1 in x[rd], otherwise write 0.

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	001	rd	1010011

2.4.4.3 fle.s rd, rs1, rs2

- x[rd] = f[rs1] ⩽ f[rs2]
- Floating-point Less Than or Equal, Single-Precision.
- R-type, RV32F and RV64F.
- If the single-precision floating-point number in register f[rs1] is less than or equal to the single-precision floating-point number in f[rs2], write 1 in x[rd], otherwise write 0.

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	000	rd	1010011

2.4.5 Single-Precision Floating-Point Classify Instruction

2.4.5.1 fclass.s rd, rs1

- x[rd] = classifys(f[rs1])
- Floating-point Classify, Single-Precision.
- R-type, RV32F and RV64F.
- Write a mask representing the category of single-precision floating-point numbers in register f[rs1] into x[rd]. One and only one bit in x[rd] is set up, see the table below.

x[rd] bit	Meaning
0	f [rs1] is −∞.
1	f [rs1] is a negative normalized number.
2	f [rs1] is a negative denormalized number.
3	f [rs1] is -0.
4	f [rs1] is +0.
5	f [rs1] is a positive denormalized number.
6	f [rs1] is a positive normalized number.
7	f [rs1] is +∞.
8	f [rs1] is the signaling NaN.
9	f [rs1] is a quiet NaN.

31 25	24 20	19 15	14 12	11 7	6 0

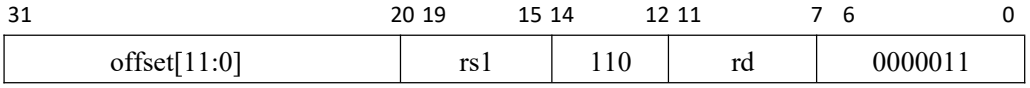
1110000	00000	rs1	001	rd	1010011

3 RISCV64-IMACD Instructions

3.1 RISCV64-I

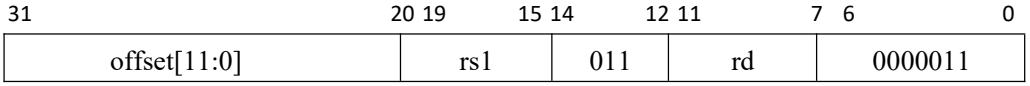
3.1.1 lwu rd, offset(rs1)

- x[rd] = M[x[rs1] + sext(offset)][31:0]
- Load Word, Unsigned.
- I-type, RV64I.
- Read four bytes from address x[rs1] + sign-extend(offset), write x[rd] after zero extension.



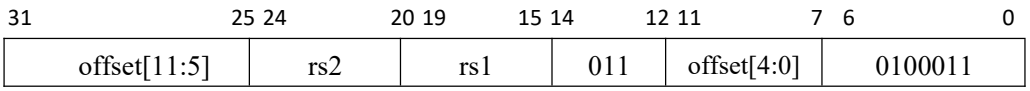
3.1.2 ld rd, offset(rs1)

- x[rd] = M[x[rs1] + sext(offset)][63:0]
- Load Doubleword.
- I-type, RV32I and RV64I.
- Read eight bytes from address x[rs1] + sign-extend(offset) and write x[rd].
- Compressed form：c.ldsp rd, offset; c.ld rd, offset(rs1)



3.1.3 sd rs2, offset(rs1)

- M[x[rs1] + sext(offset) = x[rs2][63: 0]
- Store Doubleword.
- S-type, RV64I only.
- Store the 8 bytes in x[rs2] into the memory address x[rs1]+sign-extend(offset).
- Compressed form：c.sdsp rs2, offset; c.sd rs2, offset(rs1)



3.1.4 addw rd, rs1, rs2

- x[rd] = sext((x[rs1] + x[rs2])[31:0])
- Add Word.
- R-type, RV64I.
- The register x[rs2] is added to the register x[rs1], the result is truncated to 32 bits, and the result of the sign bit extension is written to x[rd]. Ignore arithmetic overflow.
- Compressed form： c.addw rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0111011

3.1.5 addiw rd, rs1, immediate

- x[rd] = sext((x[rs1] + sext(immediate))[31:0])
- Add Word Immediate.
- I-type, RV64I.
- Add the sign bit extended immediate value to x[rs1], truncate the result to 32 bits, and write the sign bit extended result to x[rd]. Ignore arithmetic overflow.
- Compressed form： c.addiw rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0011011

3.1.6 subw rd, rs1, rs2

- x[rd] = sext((x[rs1] − x[rs2])[31: 0])
- Substract Word.
- R-type, RV64I only.
- Subtract x[rs2] from x[rs1], and the result is truncated to 32 bits, signed and extended and written into x[rd]. Ignore arithmetic overflow.
- Compressed form： c.subw rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0111011

3.1.7 sllw rd, rs1, rs2

- x[rd] = sext((x[rs1] ≪ x[rs2][4: 0])[31: 0])
- Shift Left Logical Word.
- R-type, RV64I only.

- Shift the lower 32 bits of the register x[rs1] to the left by x[rs2] bits, fill the vacant position with 0, and write the result to x[rd] after signed extension. The low 5 bits of x[rs2] represent the number of shift bits, and the high bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	001	rd	0111011

3.1.8 slliw rd, rs1, shamt

- x[rd] = sext((x[rs1] ≪ shamt)[31: 0])
- Shift Left Logical Word Immediate.
- I-type, RV64I only.
- Shift the register x[rs1] to the left by shamt bits, fill the vacant position with 0, and the result will be truncated to 32 bits, and then write to x[rd] after signed extension. The instruction is valid only when shamt[5]=0.

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	001	rd	0011011

3.1.9 sraw rd,rs1, rs2

- x[rd] = sext(x[rs1][31: 0] $\gg_s$ x[rs2][4: 0])
- Shift Right Arithmetic Word.
- R-type, RV64I only.
- Shift the lower 32 bits of the register x[rs1] to the right by x[rs2] bits, fill the empty bits with x[rs1][31], and write the result to x[rd] after signed extension. The lower 5 digits of x[rs2] are shift digits, and the higher digits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0111011

3.1.10 sraiw rd,rs1, shamt

- x[rd] = sext(x[rs1][31: 0] $\gg_s$ shamt)
- Shift Right Arithmetic Word Immediate.
- I-type, RV64I only.
- Shift the lower 32 bits of the register x[rs1] to the shamt bit to the right, fill the empty bits with x[rs1][31], and write the result to x[rd] after signed extension. The instruction is valid only when shamt[5]=0.
- Compressed form： c.srai rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0

010000	shamt	rs1	101	rd	0011011

3.1.11 srliw rd, rs1, shamt

- x[rd] = sext(x[rs1][31: 0] $\gg_u$ shamt)
- Shift Right Logical Word Immediate.
- I-type, RV64I only.
- Shift the register x[rs1] to the right by shamt bits, fill the vacant position with 0, and the result will be truncated to 32 bits, and then write x[rd] after signed extension. The instruction is valid only when shamt[5]=0.

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	101	rd	0011011

3.1.12 srlw rd, rs1, rs2

- x[rd] = sext(x[rs1][31: 0] $\gg_u$ x[rs2][4: 0])
- Shift Right Logical Word.
- R-type, RV64I only.
- Shift the lower 32 bits of the register x[rs1] to the right by x[rs2] bits, fill the vacant position with 0, and write the result to x[rd] after signed extension. The low 5 bits of x[rs2] represent the number of shift bits, and the high bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	101	rd	0111011

3.2 RISCV64-M

3.2.1 mulw rd, rs1, rs2

- x[rd] = sext((x[rs1] × x[rs2])[31: 0])
- Multiply Word.
- R-type, RV64M only.
- Multiply the register x[rs2] to the register x[rs1], the product is truncated to 32 bits, and then signed extension is written to x[rd]. Ignore arithmetic overflow.

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	000	rd	0111011

3.2.2 divw rd, rs1, rs2

- x[rd] = sext(x[rs1][31:0] ÷s x[rs2][31:0])
- Divide Word.
- R-type, RV64M.
- Divide the lower 32 bits of the register x[rs1] by the lower 32 bits of the register x[rs2], round to zero, treat these numbers as two's complement, and write the sign-extended 32-bit quotient to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	100	rd	0111011

3.2.3 divuw rd, rs1, rs2

- x[rd] = sext(x[rs1][31:0] ÷u x[rs2][31:0])
- Divide Word, Unsigned.
- R-type, RV64M.
- Divide the lower 32 bits of the register x[rs1] by the lower 32 bits of the register x[rs2], round to zero, treat these numbers as unsigned numbers, and write the signed 32-bit quotient to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0111011

3.2.4 remw rd, rs1, rs2

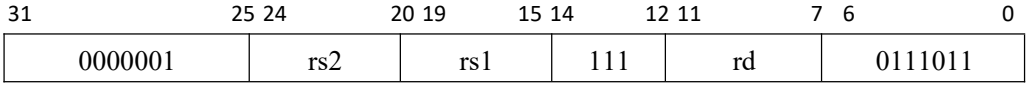
- x[rd] = sext(x[rs1][31: 0] % x[rs2][31: 0])
- Remainder Word.
- R-type, RV64M only.
- Dividing the low 32 bits of x[rs1] by the low 32 bits of x[rs2], rounding to 0, is regarded as 2's complement, and the signed extension of the remainder is written to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	110	rd	0111011

3.2.5 remuw rd, rs1, rs2

- x[rd] = sext(x[rs1][31: 0] % x[rs2][31: 0])
- Remainder Word, Unsigned.
- R-type, RV64M only.
- Dividing the lower 32 bits of x[rs1] by the lower 32 bits of x[rs2] and rounding to 0 are

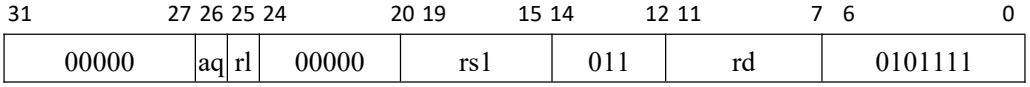
regarded as unsigned numbers, and the signed extension of the remainder is written to x[rd].



3.3 RISCV64-A

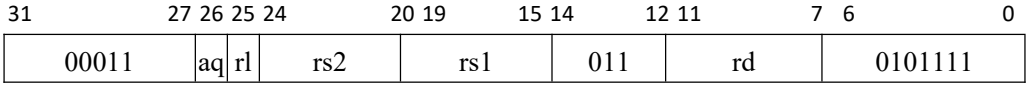
3.3.1 lr.d rd, (rs1)

- x[rd] = LoadReserved64(M[x[rs1]])
- Load-Reserved Doubleword.
- R-type, RV64A.
- Load eight bytes from the memory address x[rs1], write x[rd], and register and reserve this memory double word.



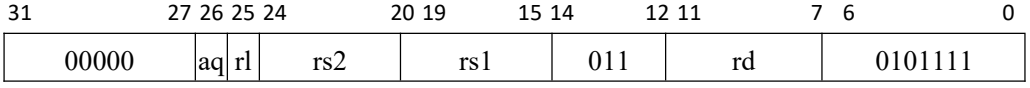
3.3.2 sc.d rd, rs2, (rs1)

- x[rd] = StoreConditonal64(M[x[rs1], x[rs2])
- Store-Conditional Doubleword.
- R-type, RV64A only.
- If there is a load reservation at the memory address x[rs1], store the 8-byte number in the x[rs2] register into this address. If the save is successful, save 0 into the register x[rd], otherwise save a non-zero error code.



3.3.3 amoadd.d rd, rs2, (rs1)

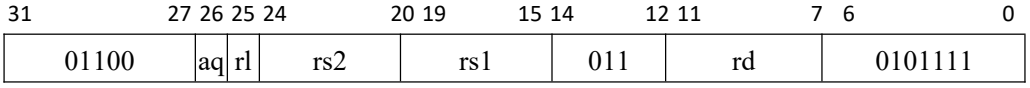
- x[rd] = AMO64(M[x[rs1]] + x[rs2])
- Atomic Memory Operation: Add Doubleword.
- R-type, RV64A.
- Perform the following atomic operation: record the double word in memory at address x[rs1] as t, change this double word to t+x[rs2], and set x[rd] to t.



3.3.4 amoand.d rd, rs2, (rs1)

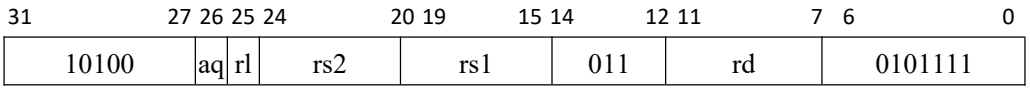
- x[rd] = AMO64(M[x[rs1]] & x[rs2])
- Atomic Memory Operation: AND Doubleword.

- R-type, RV64A.
- Perform the following atomic operation: record the double word in memory at address x[rs1] as t, turn this double word into the result of the bit AND of t and x[rs2], and set x[rd] to t.



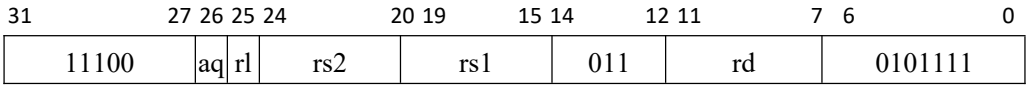
3.3.5 amomax.d rd, rs2, (rs1)

- x[rd] = AMO64(M[x[rs1]] MAX x[rs2])
- Atomic Memory Operation: Maximum Doubleword.
- R-type, RV64A.
- Perform the following atomic operation: record the double word in memory at address x[rs1] as t, change this double word into the larger one of t and x[rs2] (compare with two's complement), and set x [rd] is set to t.



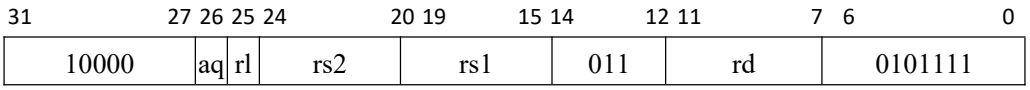
3.3.6 amomaxu.d rd, rs2, (rs1)

- x[rd] = AMO64(M[x[rs1]] MAXU x[rs2])
- Atomic Memory Operation: Maximum Doubleword, Unsigned.
- R-type, RV64A.
- Perform the following atomic operation: record the double word in memory at address x[rs1] as t, change this double word to the larger one of t and x[rs2] (using an unsigned comparison), and change x[rd] is set to t.



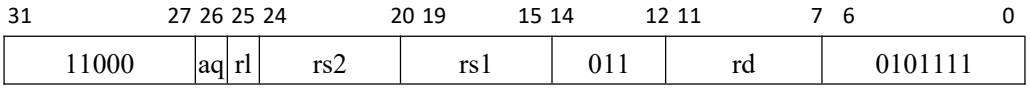
3.3.7 amomin.d rd, rs2, (rs1)

- x[rd] = AMO64(M[x[rs1]] MIN x[rs2])
- Atomic Memory Operation: Minimum Doubleword.
- R-type, RV64A.
- Perform the following atomic operation: record the double word in memory at address x[rs1] as t, change this double word into the smaller one of t and x[rs2] (compare with two's complement), and set x [rd] is set to t.



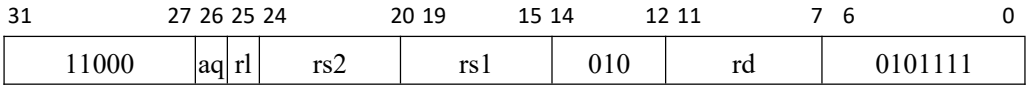
3.3.8 amominu.d rd, rs2,(rs1)

- x[rd] = AMO64(M[x[rs1]] MINU x[rs2])
- Atomic Memory Operation: Minimum Doubleword, Unsigned).
- R-type, RV64A.
- Perform the following atomic operation: record the double word in memory at address x[rs1] as t, change this double word into the smaller one of t and x[rs2] (using an unsigned comparison), and put x[rd] is set to t.



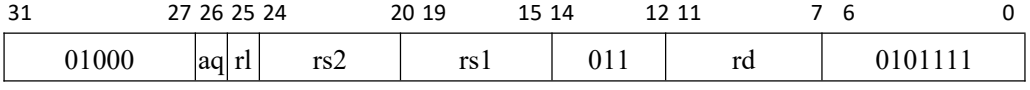
3.3.9 amominu.w rd, rs2, (rs1)

- x[rd] = AMO32(M[x[rs1]] MINU x[rs2])
- Atomic Memory Operation: Minimum Word, Unsigned.
- R-type, RV32A and RV64A.
- Perform the following atomic operation: record the word in memory at address x[rs1] as t, change this word to the smaller one of t and x[rs2] (using an unsigned comparison), and set x[rd] Set to t with sign bit extension.



3.3.10 amoor.d rd, rs2, (rs1)

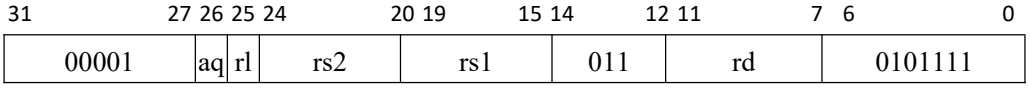
- x[rd] = AMO64(M[x[rs1]] | x[rs2])
- Atomic Memory Operation: OR Doubleword.
- R-type, RV64A.
- Perform the following atomic operation: record the double word in the memory at address x[rs1] as t, turn this double word into the result of the bit OR of t and x[rs2], and set x[rd] to t.



3.3.11 amoswap.d rd, rs2, (rs1)

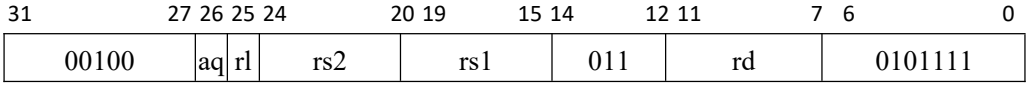
- x[rd] = AMO64(M[x[rs1]] SWAP x[rs2])

- Atomic Memory Operation: Swap Doubleword.
- R-type, RV64A.
- Perform the following atomic operation: record the double word in memory at address x[rs1] as t, change this double word to the value of x[rs2], and set x[rd] to t.



3.3.12 amoxor.d rd, rs2, (rs1)

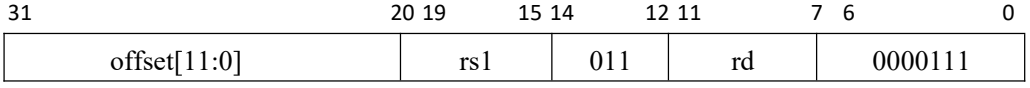
- x[rd] = AMO64(M[x[rs1]] ^ x[rs2])
- Atomic Memory Operation: XOR Doubleword.
- R-type, RV64A.
- Perform the following atomic operation: record the double word in the memory at address x[rs1] as t, turn this double word into the result of the bitwise XOR of t and x[rs2], and set x[rd] to t .



3.4 RISCV-D

3.4.1 fld rd, offset(rs1)

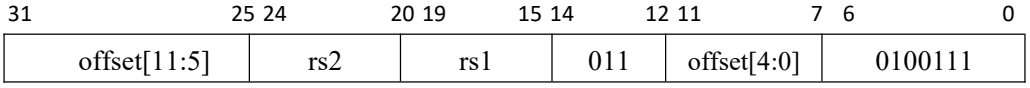
- f[rd] = M[x[rs1] + sext(offset)][63:0]
- Floating-point Load Doubleword.
- I-type, RV32D and RV64D.
- Take a double-precision floating-point number from the memory address x[rs1] + sign-extend(offset) and write it to f[rd].
- Compressed form： c.fldsp rd, offset; c.fld rd, offset(rs1)



3.4.2 fsd rs2, offset(rs1)

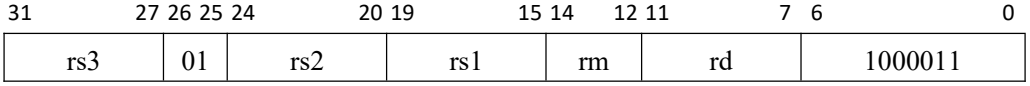
- M[x[rs1] + sext(offset)] = f[rs2][63:0]
- Floating-point Store Doubleword.
- S-type, RV32D and RV64D.

- Store the double-precision floating-point number in the register f[rs2] into the memory address x[rs1] + sign-extend(offset).
- Compressed form： c.fsdsp rs2, offset; c.fsd rs2, offset(rs1)



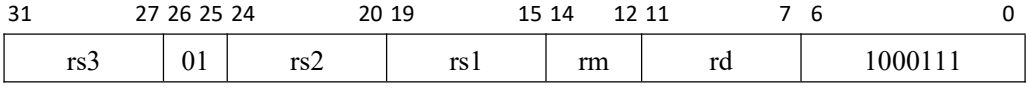
3.4.3 fmadd.d rd, rs1, rs2, rs3

- f[rd] = f[rs1]×f[rs2]+f[rs3]
- Floating-point Fused Multiply-Add, Double-Precision.
- R4-type, RV32D and RV64D.
- Multiply the double-precision floating-point numbers in the register f[rs1] and f[rs2], and add the unrounded product to the double-precision floating-point number in the register f[rs3], and the rounded double-precision floating-point number The points are written into f[rd].



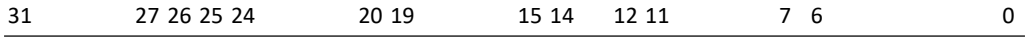
3.4.4 fmsub.d rd, rs1, rs2, rs3

- f[rd] = f[rs1]×f[rs2]-f[rs3]
- Floating-point Fused Multiply-Subtract, Double-Precision.
- R4-type, RV32D and RV64D.
- Multiply the double-precision floating-point numbers in the registers f[rs1] and f[rs2], and subtract the double-precision floating-point number in the register f[rs3] from the unrounded product, and the rounded double-precision floating-point number Write f[rd].



3.4.5 fnmsub.d rd, rs1, rs2, rs3

- f[rd] = -f[rs1]_f[rs2]+f[rs3]
- Floating-point Fused Negative Multiply-Subtract, Double-Precision). R4- type, RV32D and RV64D.
- Multiply the double-precision floating-point numbers in the registers f[rs1] and f[rs2], invert the result, and subtract the double-precision floating-point numbers in the register f[rs3] from the unrounded product. After rounding The double-precision floating-point number is written to f[rd].



rs3	01	rs2	rs1	rm	rd	1001011

3.4.6 fnmadd.d rd, rs1, rs2, rs3

- f[rd] = f[rs1]×f[rs2]+f[rs3]
- Floating-point Fused Negative Multiply-Add, Double-Precision.
- R4-type, RV32D and RV64D.
- Multiply the double-precision floating-point number in the register f[rs1] and f[rs2], invert the result, and add the unrounded product to the double-precision floating-point number in the register f[rs3], and it will be rounded The following double-precision floating-point number is written into f[rd].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1001111

3.4.7 fadd.d rd, rs1, rs2

- f[rd] = f[rs1] + f[rs2]
- Floating-point Add, Double-Precision.
- R-type, RV32D and RV64D.
- Add the double-precision floating-point numbers in the registers f[rs1] and f[rs2], and write the rounded sum to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	rm	rd	1010011

3.4.8 fsub.d rd, rs1, rs2

- f[rd] = f[rs1] - f[rs2]
- Floating-point Subtract, Double-Precision.
- R-type, RV32D and RV64D.
- Subtract the double-precision floating-point numbers in the registers f[rs1] and f[rs2], and write the rounded difference to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0000101	rs2	rs1	rm	rd	1010011

3.4.9 fmul.d rd, rs1, rs2

- f[rd] = f[rs1] × f[rs2]

- Floating-point Multiply, Double-Precision.
- R-type, RV32D and RV64D.
- Multiply the double-precision floating-point numbers in the registers f[rs1] and f[rs2], and write the rounded double-precision result into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	rm	rd	1010011

3.4.10 fdiv.d rd, rs1, rs2

- f[rd] = f[rs1] ÷ f[rs2]
- Floating-point Divide, Double-Precision.
- R-type, RV32D and RV64D.
- Divide the double-precision floating-point numbers in registers f[rs1] and f[rs2], and write the rounded quotient to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0001101	rs2	rs1	rm	rd	1010011

3.4.11 fsqrt.d rd, rs1, rs2

- f[rd] = √ f[rs1]
- Floating-point Square Root, Double-Precision.
- R-type, RV32D and RV64D.
- Round the square root of the double-precision floating-point number in f[rs1] and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0101101	rs2	rs1	rm	rd	1010011

3.4.12 fsgnj.d rd, rs1, rs2

- f[rd] = {f[rs2][63], f[rs1][62:0]}
- Floating-point Sign Inject, Double-Precision.
- R-type, RV32D and RV64D.
- Construct a new double-precision floating-point number with the exponent and significant number of f[rs1] and the sign bit of the sign of f[rs2], and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	000	rd	1010011

3.4.13 fsgnjn.d rd, rs1, rs2

- f[rd] = {~f[rs2][63], f[rs1][62:0]}
- Floating-point Sign Inject-Negate, Double-Precision.
- R-type, RV32D and RV64D.
- Use the exponent of f[rs1] and the significant number and the sign bit of the sign of f[rs2] to be inverted to construct a new double-precision floating-point number and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	001	rd	1010011

3.4.14 fsgnjx.d rd, rs1, rs2

- f[rd] = {f[rs1][63] ^ f[rs2][63], f[rs1][62:0]}
- Floating-point Sign Inject-XOR, Double-Precision.
- R-type, RV32D and RV64D.
- Use f[rs1] exponent and significant number and the sign bit of f[rs1] and f[rs2] to construct a new double-precision floating-point number and write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	010	rd	1010011

3.4.15 fmin.d rd, rs1, rs2

- f[rd] = min(f[rs1], f[rs2])
- Floating-point Minimum, Double-Precision.
- R-type, RV32D and RV64D.
- Write the smaller value of the double-precision floating-point number in the registers f[rs1] and f[rs2] into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0010101	rs2	rs1	000	rd	1010011

3.4.16 fmax.d rd, rs1, rs2

- f[rd] = max(f[rs1], f[rs2])
- Floating-point Maximum, Double-Precision.
- R-type, RV32D and RV64D.
- Write the larger value of the double-precision floating-point number in the registers f[rs1] and f[rs2] into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0

0010101	rs2	rs1	001	rd	1010011

3.4.17 fcvt.s.d rd, rs1, rs2

- f[rd] = $f32_{f64}$(f[rs1])
- Floating-point Convert to Single from Double.
- R-type, RV32D and RV64D.
- Convert the double-precision floating-point number in the register f[rs1] into a single-precision floating-point number, and then write it into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	rm	rd	1010011

3.4.18 fcvt.d.s rd, rs1, rs2

- f[rd] = $f64_{f32}$(f[rs1])
- Floating-point Convert to Double from Single.
- R-type, RV32D and RV64D.
- Convert the single-precision floating-point number in the register f[rs1] into a double-precision floating-point number, and then write it into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
0100001	rs2	rs1	rm	rd	1010011

3.4.19 fcvt.w.d rd, rs1, rs2

- x[rd] = sext($s32_{f64}$(f[rs1]))
- Floating-point Convert to Word from Double.
- R-type, RV32D and RV64D.
- Convert the double-precision floating-point number in the register f[rs1] into a 32-bit two's complement integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	rs2	rs1	rm	rd	1010011

3.4.20 fcvt.wu.d rd, rs1, rs2

- x[rd] = sext($u32_{f64}$(f[rs1]))
- Floating-point Convert to Unsigned Word from Double.
- R-type, RV32D and RV64D.

- Convert the double-precision floating-point number in the register f[rs1] into a 32-bit unsigned integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00001	rs1	rm	rd	1010011

3.4.21 fcvt.d.w rd, rs1, rs2

- f[rd] = $f64_{s32}$(x[rs1])
- Floating-point Convert to Double from Word.
- R-type, RV32D and RV64D.
- Convert the 32-bit two's complement integer represented by the register x[rs1] into a double-precision floating-point number, and then write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00000	rs1	rm	rd	1010011

3.4.22 fcvt.d.wu rd, rs1, rs2

- f[rd] = $f64_{u32}$(x[rs1])
- Floating-point Convert to Double from Unsigned Word.
- R-type, RV32D and RV64D.
- Convert the 32-bit unsigned integer in the register x[rs1] into a double-precision floating-point number, and then write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00001	rs1	rm	rd	1010011

3.4.23 feq.d rd, rs1, rs2

- x[rd] = f[rs1] == f[rs2]
- Floating-point Equals, Double-Precision.
- R-type, RV32D and RV64D.
- If the double-precision floating-point numbers in the registers f[rs1] and f[rs2] are equal, write 1 in x[rd], otherwise write 0.

31 25	24 20	19 15	14 12	11 7	6 0
1101001	rs2	rs1	010	rd	1010011

3.4.24 flt.d rd, rs1, rs2

- x[rd] = f[rs1] ⩽ f[rs2]
- Floating-point Less Than or Equal, Double-Precision).
- R-type, RV32D and RV64D.
- If the double-precision floating-point number in register f[rs1] is less than or equal to the double-precision floating-point number in f[rs2], write 1 in x[rd], otherwise write 0.

31 25	24 20	19 15	14 12	11 7	6 0
1101001	rs2	rs1	000	rd	1010011

3.4.25 fle.d rd, rs1, rs2

- x[rd] = f[rs1] < f[rs2]
- Floating-point Less Than, Double-Precision.
- R-type, RV32D and RV64D.
- If the double-precision floating-point number in register f[rs1] is less than the double-precision floating-point number in f[rs2], write 1 in x[rd], otherwise write 0.

31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	001	rd	1010011

3.4.26 fmv.d.x rd, rs1, rs2

- f[rd] = x[rs1][63:0]
- Floating-point Move Doubleword from Integer.
- R-type, RV64D.
- Copy the double-precision floating-point number in the register x[rs1] to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1111001	00000	rs1	000	rd	1010011

3.4.27 fmv.x.d rd, rs1, rs2

- x[rd] = f[rs1][63:0]
- Floating-point Move Doubleword to Integer.
- R-type, RV64D.
- Copy the double-precision floating-point number in the register f[rs1] to x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1110001	00000	rs1	000	rd	1010011

3.4.28 fcvt.d.l rd, rs1, rs2

- f[rd] = $f64_{s64}$(x[rs1])
- Floating-point Convert to Double from Long.
- R-type, RV64D.
- Convert the 64-bit two's complement integer represented by the register x[rs1] into a double-precision floating-point number, and then write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00010	rs1	rm	rd	1010011

3.4.29 fcvt.d.lu rd, rs1, rs2

- f[rd] = $f64_{u64}$(x[rs1])
- Floating-point Convert to Double from Unsigned Long.
- R- type, RV64D.
- Convert the 64-bit unsigned integer in the register x[rs1] into a double-precision floating-point number, and then write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00011	rs1	rm	rd	1010011

3.4.30 fcvt.l.d rd, rs1, rs2

- x[rd] = $s64_{f64}$(f[rs1])
- Floating-point Convert to Long from Double.
- R-type, RV64D.
- Convert the double-precision floating-point number in the register f[rs1] into a 64-bit two's complement integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00010	rs1	rm	rd	1010011

3.4.31 fcvt.l.s rd, rs1, rs2

- x[rd] = $s64_{f32}$(f[rs1])
- Floating-point Convert to Long from Single.
- R-type, RV64F.
- Convert the single-precision floating-point number in the register f[rs1] into a 64-bit two's complement integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0

1100001	00010	rs1	rm	rd	1010011

3.4.32 fcvt.lu.d rd, rs1, rs2

- x[rd] = $u64_{f64}$(f[rs1])
- Floating-point Convert to Unsigned Long from Double).
- R- type, RV64D.
- Convert the double-precision floating-point number in the register f[rs1] into a 64-bit unsigned integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00011	rs1	rm	rd	1010011

3.4.33 fcvt.lu.s rd, rs1, rs2

- x[rd] = $u64_{f32}$(f[rs1])
- Floating-point Convert to Unsigned Long from Single.
- R-type, RV64F.
- Convert the single-precision floating-point number in the register f[rs1] into a 64-bit two's complement integer, and then write it into x[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00011	rs1	rm	rd	1010011

3.4.34 fcvt.s.l rd, rs1, rs2

- f[rd] = $f32_{s64}$(x[rs1])
- Floating-point Convert to Single from Long.
- R-type, RV64F.
- Convert the 64-bit two's complement integer represented by the register x[rs1] into a single-precision floating-point number, and then write it into f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00010	rs1	rm	rd	1010011

3.4.35 fcvt.s.lu rd, rs1, rs2

- f[rd] = $f32_{u64}$(x[rs1])
- Floating-point Convert to Single from Unsigned Long.
- R-type, RV64F.

- Convert the 64-bit unsigned integer in the register x[rs1] into a single-precision floating-point number, and then write it to f[rd].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00011	rs1	rm	rd	1010011

3.4.36 fclass.d rd, rs1, rs2

- x[rd] = classifyd(f[rs1])
- Floating-point Classify, Double-Precision.
- R-type, RV32D and RV64D.
- Write a mask representing the category of double-precision floating-point numbers in register f[rs1] into x[rd]. For how to interpret the value written in x[rd], please refer to the description of the instruction fclass.s.

31 25	24 20	19 15	14 12	11 7	6 0
1110001	00000	rs1	001	rd	1010011

3.5 RISCV64-C

3.5.1 c.ld rd', uimm(rs1')

- x[8+rd’] = M[x[8+rs1’] + uimm][63:0]
- Load Doubleword.
- RV64IC.
- The extended form is ld rd, uimm(rs1), where rd=8+rd’, rs1=8+rs1’.

15 13	12 11 10	9 7	6 5	4 2	1 0
011	imm[5:3]	rs1’	uimm[7:6]	rd’	00

3.5.2 c.ldsp rd, uimm(x2)

- x[rd] = M[x[2] + uimm][63:0]
- Load Doubleword, Stack-Pointer Relative.
- RV64IC.
- The extended form is ld rd, uimm(x2). rd=x0 is illegal.

15 13	12	11 7	6 2	1 0
011	imm[5]	rd	uimm[4:3|8:6]	10

3.5.3 c.sd rs2', uimm(rs1')

- M[x[8+rs1’] + uimm][63:0] = x[8+rs2’]
- Store Doubleword.
- RV64IC.
- The extended form is sd rs2, uimm(rs1), where rs2=8+rs2’, rs1=8+rs1’.

15 13	12 11 10	9 7	6 5	4 2	1 0
111	imm[5:3]	rs1’	uimm[7:6]	rs2’	00

3.5.4 c.sdsp rs2, uimm(x2)

- M[x[2] + uimm][63:0] = x[rs2]
- Store Doubleword, Stack-Pointer Relative.
- RV64IC.
- The extended form is sd rs2, uimm(x2).

15 13	12 7	6 2	1 0
111	imm[5:3|8:6]	rs2	00

3.5.5 c.addw rd', rs2'

- x[8+rd’] = sext((x[8+rd’] + x[8+rs2’])[31:0])
- Add Word.
- RV64IC.
- The extended form is addw rd, rd, rs2, where rd=8+rd’, rs2=8+rs2’.

15 10	9 7	6 5	4 2	1 0
100111	rd’	01	rs2’	01

3.5.6 c.addiw rd, imm

- x[rd] = sext((x[rd] + sext(imm))[31:0])
- Add Word Immediate.
- RV64IC.
- The extended form is addiw rd, rd, imm. rd=x0 is illegal.

15 13	12	11 7	6 2	1 0
001	imm[5]	rd	imm[4:0]	01

3.5.7 c.subw rd', rs2'

- x[8+rd'] = sext((x[8+rd'] - x[8+rs2']))[31:0])
- Subtract Word.
- RV64IC.
- The extended form is subw rd, rd, rs2. where rd=8+rd', rs2=8+rs2'.

15 10	9 7	6 5	4 2	1 0
100111	rd'	00	rs2'	01

3.5.8 c.fld rd', uimm(rs1')

- f[8+rd'] = M[x[8+rs1'] + uimm][63:0]
- Floating-point Load Doubleword.
- RV32DC and RV64DC.
- The extended form is fld rd, uimm(rs1), where rd=8+rd', rs1=8+rs1'.

15 13	12 11 10	9 7	6 5	4 2	1 0
001	uimm[5:3]	rs1'	uimm[7:6]	rd'	00

3.5.9 c.fldsp rd, uimm(x2)

- f[rd] = M[x[2] + uimm][63:0]
- Floating-point Load Doubleword, Stack-Pointer Relative.
- RV32DC and RV64DC.
- The extended form is fld rd, uimm(x2).

15 13	12	11 7	6 2	1 0
001	uimm[5]	rd	uimm[4:3|8:6]	10

3.5.10 c.fsd rs2', uimm(rs1')

- M[x[8+rs1'] + uimm][63:0] = f[8+rs2']
- Floating-point Store Doubleword.
- RV32DC and RV64DC.
- The extended form is fsd rs2, uimm(rs1), where rs2=8+rs2', rs1=8+rs1'.

15 13	12 11 10	9 7	6 5	4 2	1 0
101	imm[5:3]	rs1'	uimm[7:6]	rs2'	00

3.5.11 c.fsdsp rs2, uimm(x2)

- M[x[2] + uimm][63:0] = f[rs2]
- Floating-point Store Doubleword, Stack-Pointer Relative.
- RV32DC and RV64DC.
- The extended form is fsd rs2, uimm(x2).

15 13	12 7	6 2	1 0
101	imm[5:3|8:6]	rs2	10

4 RISCV Pseudo-Instructions

4.1.1 nop

- Nothing
- No operation.
- Pseudoinstruction, RV32I and RV64I.
- Advance pc to the next instruction.
- Actually expanded to : **addi** x0, x0, 0.

4.1.2 neg rd, rs2

- x[rd] = −x[rs2]
- Negate.
- Pseudoinstruction, RV32I and RV64I.
- Write the twos complement of register x[rs2] to x[rd].
- Actually expanded to : **sub** rd, x0, rs2.

4.1.3 negw rd, rs2

- x[rd] = sext((−x[rs2])[31: 0])
- Negate Word.
- Pseudoinstruction, RV64I only.
- Calculate the 2's complement of the register x[rs2], the result is truncated to 32 bits, sign extended and written into x[rd].
- Actually expanded to : **subw** rd, x0, rs2.

4.1.4 snez rd, rs2

- x[rd] = (x[rs2] $\neq$ 0)
- Set if Not Equal to Zero.
- Pseudoinstruction, RV32I and RV64I.
- If x[rs1] is not equal to 0, write 1 to x[rd], otherwise write 0.
- Actually expanded to : **sltu** rd, x0, rs2.

4.1.5 sltz rd, rs1

- x[rd] = (x[rs1] $<_s$ 0)
- Set if Less Than to Zero.
- Pseudoinstruction, RV32I and RV64I.

- If x[rs1] is less than 0, write 1 to x[rd], otherwise write 0.
- Actually expanded to : **slt** rd, rs1, x0.

4.1.6 sgtz rd, rs2

- x[rd] = (x[rs1] $>_s$ 0)
- Set if Greater Than Zero.
- Pseudoinstruction, RV32I and RV64I.
- If x[rs2] is greater than 0, write 1 to x[rd], otherwise write 0.
- Actually expanded to : **slt** rd, x0, rs2.

4.1.7 beqz rs1, offset

- if (rs1 == 0) pc += sext(offset)
- Branch if Equal to Zero.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **beq** rs1, x0, offset.

4.1.8 bnez rs1, offset

- if (rs1 $\neq$ 0) pc += sext(offset)
- Branch if Not Equal to Zero.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **bne** rs1, x0, offset.

4.1.9 blez rs2, offset

- if (rs2 $\leqslant_s$ 0) pc += sext(offset)
- Branch if Less Than or Equal to Zero.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **bge** x0, rs2, offset.

4.1.10 bgez rs1, offset

- if (rs1 $\geqslant_s$ 0) pc += sext(offset)
- Branch if Greater Than or Equal to Zero.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **bge** rs1, x0, offset.

4.1.11 bltz rs2, offset

- if (rs1 $<_s$ 0) pc += sext(offset)
- Branch if Less Than Zero.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **blt** rs1, x0, offset.

4.1.12 bgtz rs1, offset

- if (rs2 $>_s$ 0) pc += sext(offset)
- Branch if Greater Than Zero.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **blt** x0, rs2, offset.

4.1.13 j offset

- pc += sext(offset)
- Jump.
- Pseudoinstruction, RV32I and RV64I.
- Set pc to the current value plus the sign extended offset.
- Actually expanded to : **jal** x0, offset.

4.1.14 jr rs1

- pc = x[rs1]
- Jump Register.
- Pseudoinstruction, RV32I and RV64I.
- Set pc to x[rs1].
- Actually expanded to : **jalr** x0, 0(rs1).

4.1.15 ret

- pc = x[1]
- Return.
- Pseudoinstruction, RV32I and RV64I.
- Return from subroutine.
- Actually expanded to : **jalr** x0, 0(x1).

4.1.16 tail symbol

- pc = &symbol; clobber x[6]
- Tail call.
- Pseudoinstuction, RV32I and RV64I.
- Set pc to symbol and overwrite x[6] at the same time.
- Actually expanded to : **auipc** x6, offsetHi and **jalr** x0, offsetLo(x6).

4.1.17 rdcycle rd

- x[rd] = CSRS[cycle]
- Read Cycle Counter.
- Pseudoinstruction, RV32I and RV64I.
- Write the number of cycles to x[rd].
- Actually expanded to : **csrrs** rd, cycle, x0.

4.1.18 rdcycleh rd

- x[rd] = CSRs[cycleh]
- Read Cycle Counte High.
- Pseudoinstruction, RV32I only.
- Shift the number of cycles to the right by 32 bits and write x[rd].
- Actually expanded to : **csrrs** rd, cycleh, x0.

4.1.19 rdinstret rd

- x[rd] = CSRs[instret]
- Read Instruction-Retired Counter.
- Pseudoinstruction, RV32I and RV64I.
- Write the number of completed instructions to x[rd].
- Actually expanded to : **csrrs** rd, instret, x0.

4.1.20 rdinstreth rd

- x[rd] = CSRs[instreth]
- Read Instruction-Retired Counter High.
- Pseudoinstruction, RV32I only.
- Shift the number of completed instructions to the right by 32 bits and write x[rd].
- Actually expanded to : **csrrs** rd, instreth, x0.

4.1.21 rdtime rd

- x[rd] = CSRs[time]
- Read Time.
- Pseudoinstruction, RV32I and RV64I.
- Write the current time into x[rd], the time frequency is related to the platform.
- Actually expanded to : **csrrs** rd, time, x0.

4.1.22 rdtimeh rd

- x[rd] = CSRs[timeh]
- Read Time High.
- Pseudoinstruction, RV32I only.
- Shift the current time to the right by 32 bits and write it into x[rd]. The time and frequency are related to the platform.
- Actually expanded to : **csrrs** rd, timeh, x0.

4.1.23 csrr rd, csr

- x[rd] = CSRs[csr]
- Control and Status Register Read.
- Pesudoinstruction, RV32I and RV64I.
- Write the value of the control status register csr into x[rd].
- Actually expanded to : **csrrs** rd, csr, x0.

4.1.24 csrc csr, rs1

- CSRs[csr] &= ~x[rs1]
- Control and Status Register Clear.
- Pesudoinstruction, RV32I and RV64I.
- For each bit of 1 in x[rs1], clear the corresponding bit of the control status register csr;
- Actually expanded to : **csrrc** x0, csr, rs1.

4.1.25 csrci csr, zimm[4:0]

- CSRs[csr] &= ~zimm
- Control and Status Register Clear Immediate.
- Pesudoinstruction, RV32I and RV64I.

- For each bit of the five-bit zero-extended immediate value that is 1, clear the corresponding bit of the control status register csr.
- Actually expanded to : **csrrci** x0, csr, zimm.

4.1.26 csrs csr, rs1

- CSRs[csr] |= x[rs1]
- Control and Status Register Set.
- Pesudoinstruction, RV32I and RV64I.
- For each bit of 1 in x[rs1], set the corresponding bit of the control status register csr.
- Actually expanded to : **csrrs** x0, csr, rs1.

4.1.27 csrsi csr, zimm[4:0]

- CSRs[csr] |= zimm
- Control and Status Register Set Immediate.
- Pesudoinstruction, RV32I and RV64I.
- For each bit of the five-bit zero-extended immediate value that is 1, clear the corresponding bit of the control status register csr.
- Actually expanded to : **csrrsi** x0, csr, zimm.

4.1.28 csrw csr, rs1

- CSRs[csr] = x[rs1]
- Control and Status Register Set.
- Pesudoinstruction, RV32I and RV64I.
- For each bit of 1 in x[rs1], set the corresponding bit of the control status register csr.
- Actually expanded to : **csrrs** x0, csr, rs1.

4.1.29 csrwi csr, zimm[4:0]

- CSRs[csr] = zimm
- Control and Status Register Write Immediate.
- Pesudoinstruction, RV32I and RV64I.
- Write the value of the five-bit zero-extended immediate value into the control status register csr.
- Actually expanded to : **csrrwi** x0, csr, zimm.

4.1.30 frcsr rd

- x[rd] = CSRs[fcsr]

- Floating-point Read Control and Status Register.
- Pseudoinstruction, RV32F and RV64F.
- Write the value of the floating-point control status register to x[rd].
- Actually expanded to : **csrrs** rd, fcsr, x0.

4.1.31 fscsr rd, rs1

- t = CSRs[fcsr]; CSRs[fcsr] = x[rs1]; x[rd] = t
- Floating-point Swap Control and Status Register.
- Pseudoinstruction, RV32F and RV64F.
- Write the value of register x[rs1] into the floating-point control status register, and write the original value of the floating-point control status register to x[rd].
- Actually expanded to : **csrrw** rd, fcsr, rs1.rd defaults to x0.

4.1.32 frrm rd

- x[rd] = CSRs[frm]
- Floating-point Read Rounding Mode.
- Pseudoinstruction, RV32F and RV64F.
- Write the value of floating-point rounding mode to x[rd].
- Actually expanded to : **csrrs** rd, frm, x0.

4.1.33 fsrm rd, rs1

- t = CSRs[frm]; CSRs[frm] = x[rs1]; x[rd] = t
- Floating-point Swap Rounding Mode.
- Pseudoinstruction, RV32F and RV64F.
- Write the value of register x[rs1] into the floating-point rounding mode register, and write the original value of the floating-point rounding mode register to x[rd].
- Actually expanded to : **csrrw** rd, frm, rs1.rd defaults to x0.

4.1.34 frflags rd

- x[rd] = CSRs[fflags]
- Floating-point Read Exception Flags.
- Pseudoinstruction, RV32F and RV64F.
- Write the value of the floating-point exception flag to x[rd].
- Actually expanded to : **csrrs** rd, fflags, x0.

4.1.35 fsflags rd, rs1

- t = CSRs[fflags]; CSRs[fflags] = x[rs1]; x[rd] = t
- Floating-point Swap Exception Flags.
- Pseudoinstruction, RV32F and RV64F.
- Write the value of register x[rs1] into the floating-point exception flag register, and write the original value of the floating-point exception flag register to x[rd].
- Actually expanded to : **csrrw** rd, fflags, rs1.rd defaults to x0.

4.1.36 lla rd, symbol

- x[rd] = &symbol
- Load Local Address.
- Pseudoinstruction, RV32I and RV64I.
- Load the address of symbol into x[rd].
- Actually expanded to : **auipc** rd, offsetHi，then **addi** rd, rd, offsetLo.

4.1.37 la rd, symbol

- x[rd] = &symbol
- Load Address.
- Pseudoinstruction, RV32I and RV64I.
- Load the address of symbol into x[rd].
- When compiling position-independent code, it will be expanded to load the Global Offset Table. For RV32I, it is equivalent to executing **auipc** rd, offsetHi, then **lw** rd, offsetLo(rd); for RV64I, it is equivalent to **auipc** rd, offsetHi and **ld** rd, offsetLo(rd). If the offset is too large, the initial instructions for calculating the load address will become two, first **auipc** rd, offsetHi and then **addi** rd, rd, offsetLo.

4.1.38 li rd, immediate

- x[rd] = immediate
- Load Immediate.
- Pseudoinstruction, RV32I and RV64I.
- Use as few instructions as possible to load constants into x[rd].
- In RV32I, it is equivalent to executing **lui and/or addi**; for RV64I, it will be expanded to this kind of instruction sequence **lui, addi, slli, addi, slli, addi, slli, addi**.

4.1.39 mv rd, rs1

- x[rd] = x[rs1]

- Move.
- Pseudoinstruction, RV32I and RV64I.
- Copy the register x[rs1] to x[rd].
- Actually expanded to : **addi** rd, rs1, 0

4.1.40 not rd, rs1

- x[rd] = ~x[rs1]
- NOT.
- Pseudoinstruction, RV32I and RV64I.
- Write the 1's complement of register x[rs1] (that is, the bit-wise inverted value) into x[rd].
- Actually expanded to : **xori** rd, rs1, -1.

4.1.41 sext.w rd, rs1

- x[rd] = sext(x[rs1][31: 0])
- Sign-extend Word.
- Pseudoinstruction, RV64I only.
- Read in the lower 32 bits of x[rs1] with sign extension, and write the result to x[rd].
- Actually expanded to : **addiw** rd, rs1, 0.

4.1.42 seqz rd, rs1

- x[rd] = (x[rs1] == 0)
- Set if Equal to Zero.
- Pseudoinstruction, RV32I and RV64I.
- If x[rs1] is equal to 0, write 1 to x[rd], otherwise write 0.
- Actually expanded to : **sltiu** rd, rs1, 1.

4.1.43 fmv.s rd, rs1

- f[rd] = f[rs1]
- Floating-point Move.
- Pesudoinstruction, RV32F and RV64F.
- Copy the single-precision floating-point number in the register f[rs1] to f[rd].
- Actually expanded to : **fsgnj.s** rd, rs1, rs1.

4.1.44 fmv.d rd, rs1

- f[rd] = f[rs1]
- Floating-point Move.

- Pesudoinstruction, RV32D and RV64D.
- Copy the double-precision floating-point number in the register f[rs1] to f[rd].
- Actually expanded to : **fsgnj.d** rd, rs1, rs1.

4.1.45 fneg.d rd, rs1

- f[rd] = -f[rs1]
- Floating-point Negate.
- Pesudoinstruction, RV32D and RV64D.
- Invert the double-precision floating-point number in register f[rs1] and write it into f[rd].
- Actually expanded to : **fsgnjn.d** rd, rs1, rs1.

4.1.46 fneg.s rd, rs1

- f[rd] = -f[rs1]
- Floating-point Negate.
- Pesudoinstruction, RV32F and RV64F.
- Invert the single-precision floating-point number in register f[rs1] and write it into f[rd].
- Actually expanded to : **fsgnjn.s** rd, rs1, rs1.

4.1.47 fabs.d rd, rs1

- f[rd] = |f[rs1]|
- Floating-point Absolute Value.
- Pesudoinstruction, RV32D and RV64D.
- Write the absolute value of the double-precision floating-point number f[rs1] to f[rd].
- Actually expanded to : **fsgnjx.d** rd, rs1, rs1.

4.1.48 fabs.s rd, rs1

- f[rd] = |f[rs1]|
- Pesudoinstruction, RV32F and RV64F.
- Write the absolute value of the single-precision floating-point number f[rs1] to f[rd].
- Actually expanded to : **fsgnjx.s** rd, rs1, rs1.

4.1.49 bgt rs1, rs2, offset

- if (rs1 >s rs2) pc += sext(offset)
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **blt** rs2, rs1, offset.

4.1.50 bgtu rs1, rs2, offset

- if (rs1 >u rs2) pc += sext(offset)
- Branch if Greater Than, Unsigned.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **bltu** rs2, rs1, offset.

4.1.51 ble rs1, rs2, offset

- if (rs1 ⩽s rs2) pc += sext(offset)
- Branch if Less Than or Equal.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **bge** rs2, rs1, offset.

4.1.52 bleu rs1, rs2, offset

- if (rs1 ⩽u rs2) pc += sext(offset)
- Branch if Less Than or Equal, Unsigned.
- Pesudoinstruction, RV32I and RV64I.
- Actually expanded to : **bgeu** rs2, rs1, offset.

4.1.53 call rd, symbol

- x[rd] = pc+8; pc = &symbol
- Call.
- Pesudoinstruction, RV32I and RV64I.
- Write the address of the next instruction (pc+8) into x[rd], and then set pc to symbol.
- Actually expanded to : **auipc** rd, offestHi,then **jalr** rd, offsetLo(rd). If rd is omitted, the default is x1.

5 RISCV-V Instructions

5.1 Configuration-Setting Instructions

5.1.1 vsetvli rd, rs1, vtypei

- x[rd] = new vl, x[rs1] = AVL, vtypei = new vtype setting
- The new vtype setting is encoded in the immediate fields of vsetvli and in the rs2 register for vsetvl.
- The new vector length setting is based on the requested application vector length (AVL), which is encoded in the rs1 and rd fields as follows:

Table 7. AVL used in vsetvli and vsetvl instructions

rd	rs1	AVL value	Description/Usage
0	0	Value in vl register	Change vtype without changing vl
!0	0	~0	Set vl to VLMAX
-	!0	Value in x[rs1]	Normal stripmining

  - When rs1 is not x0, the AVL is an unsigned integer held in the x register speci ed by rs1, and the new vl value is also written to the x register speci ed by rd.
  - When rs1=x0 but rd!=x0, the maximum unsigned integer value (~0) is used as the AVL, and the resulting VLMAX is written to vl and also to the x register speci ed by rd.
  - When rs1=x0 and rd=x0, the current vector length in vl is used as the AVL, and the resulting value is only written to vl.

Table 8. vtype register layout

Bits	Name	Description
XLEN-1:5		Reserved
4:2	vsew[2:0]	Standard element width (SEW) setting
1:0	vlmul[1:0]	Vector register group multiplier (LMUL) setting

- Constraints on Setting vl

  The vsetvl{i} instructions first set VLMAX according to the vtype argument, then set vl obeying the following constraints:

  1. vl = AVL if AVL $\leqslant$ VLMAX
  2. ceil(AVL / 2) $\leqslant$ vl $\leqslant$ VLMAX if AVL < (2 * VLMAX)
  3. vl = VLMAX if AVL $\geqslant$ (2 * VLMAX)
  4. Deterministic on any given implementation for same input AVL and VLMAX values
  5. These speci c properties follow from the prior rules:
     a. vl = 0 if AVL = 0

b. vl > 0 if AVL > 0

c. vl $\leqslant$ VLMAX

d. vl $\leqslant$ AVL

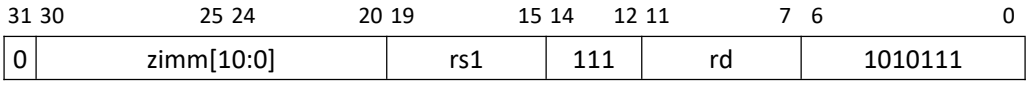
e. a value read from vl when used as the AVL argument to vsetvl{i} results in the same value in vl, provided the resultant VLMAX equals the value of VLMAX at the time that vl was read.

- Examples

The SEW and LMUL settings can be changed dynamically to provide high throughput on mixed-width operations in a single loop.

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result # right by 3, store 32b values.

# Loop using only widest elements:

loop:
    vsetvli a3, a0, e32,m8      # Use only 32-bit elements
    vlh.v v8, (a1)    # Sign-extend 16b load values to 32b elements
    sll t1, a3, 1       # Multiply length by two bytes/element
    add a1, a1, t1   # Bump pointer
    vmul.vx    v8, v8, x10       # 32b multiply result vsrl.vi
    v8, v8, 3   # Shift elements
    vsw.v v8, (a2)   # Store vector of 32b results
    sll t1, a3, 2       # Multiply length by four bytes/element
    add a2, a2, t1   # Bump pointer
    sub a0, a0, a3   # Decrement count
    bnez a0, loop   # Any more?

# Alternative loop that switches element widths.

loop:
    vsetvli a3, a0, e16,m4      # vtype = 16-bit integer vectors
    vlh.v v4, (a1)    # Get 16b vector
    slli t1, a3, 1       # Multiply length by two bytes/element
    add a1, a1, t1   # Bump pointer
    vwmul.vx v8, v4, x10       # 32b in <v8--v15>

    vsetvli x0, a0, e32,m8      # Operate on 32b values vsrl.vi v8, v8, 3
    vsw.v v8, (a2)   # Store vector of 32b
    slli t1, a3, 2       # Multiply length by four bytes/element
    add a2, a2, t1   # Bump pointer
    sub a0, a0, a3   # Decrement count
    bnez a0, loop   # Any more?
```

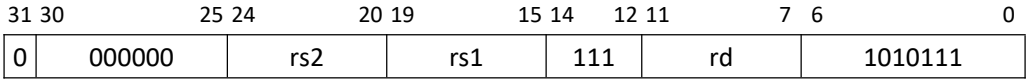
The second loop is more complex but will have greater performance on machines where 16b widening multiplies are faster than 32b integer multiplies, and where 16b vector load can run faster due to the narrower writes to the vector regfile.



31	30 25	24 20	19 15	14 12	11 7	6 0
0	zimm[10:0]		rs1	111	rd	1010111

5.1.2 vsetvl rd, rs1, rs2

- x[rd] = new vl, x[rs1] = AVL, vtypei = new vtype setting
- The vsetvl variant operates similarly to vsetvli except that it takes a vtype value from rs2 and can be used for context restore, and when the vtypei eld is too small to hold the desired setting.



31	30 25	24 20	19 15	14 12	11 7	6 0
0	000000	rs2	rs1	111	rd	1010111

5.2 Vector Loads and Stores Instructions

Vector Load/Store Addressing Modes

The base vector extension supports unit-stride, strided, and indexed (scatter/gather) addressing modes. Vector load/store base registers and strides are taken from the GPR x registers. The base effective address for all vector accesses is given by the contents of the x register named in rs1.

Vector unit-stride operations access elements stored contiguously in memory starting from the base effective address.

Vector strided operations access the first memory element at the base effective address, and then access subsequent elements at address increments given by the byte offset contained in the x register specified by rs2.

Vector indexed operations add the contents of each element of the vector offset operand speci ed by vs2 to the base effective address to give the effective address of each element. The vector offset operand is treated as a vector of byte offsets. If the vector offset elements are narrower than XLEN, they are zero- extended to XLEN before adding to the base effective address. If the vector offset elements are wider than XLEN, the least-signi cant XLEN bits are used in the

address calculation.

Vector Load/Store Width Encoding

Three of the width types encode vector loads and stores that move fixed-size memory elements of 8 bits, 16 bits, or 32 bits, while the fourth encoding moves SEW-bit memory elements.

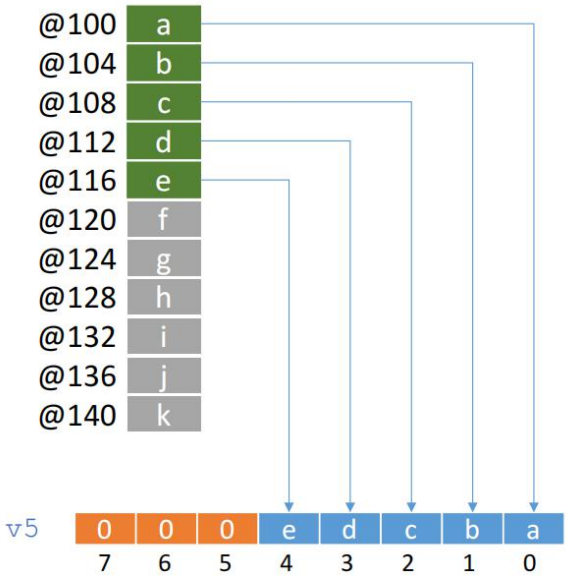
	Width [2:0]			Mem bits	Reg bits	Opcode
Vector byte	0	0	0	vl*8	vl*SEW	VxB
Vector halfword	1	0	1	vl*16	vl*SEW	VxH
Vector word	1	1	0	vl*32	vl*SEW	VxW
Vector element	1	1	1	vl*SEW	vl*SEW	VxE

Mem bits is the size of element accessed in memory
Reg bits is the size of element accessed in register

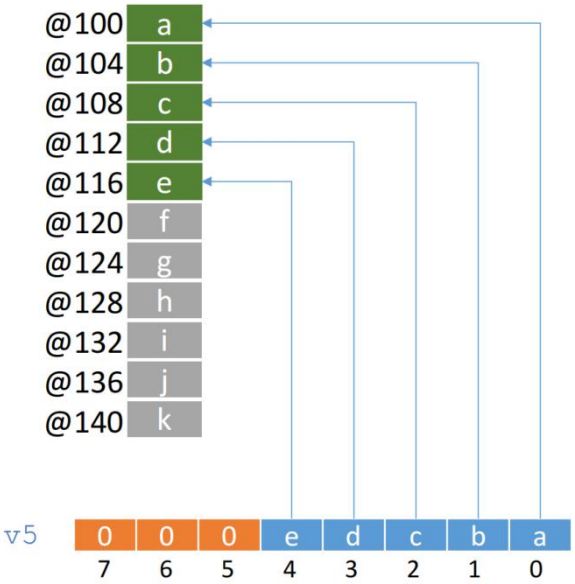
Fixed-sized vector loads can optionally sign or zero-extend their memory element into the destination register element if the register element is wider than the memory element. A fixed-size vector load raises an illegal instruction exception if the destination register element is narrower than the memory element. The variable-sized load is encoded as if a zero-extended load, with what would be the sign-extended encoding of a variable-sized load currently reserved.

Fixed-size vector stores take their operand from the least-significant bits of the register element if the register element is wider than the memory element. Fixed-sized vector stores raise an illegal instruction exception if the memory element is wider than the register element.

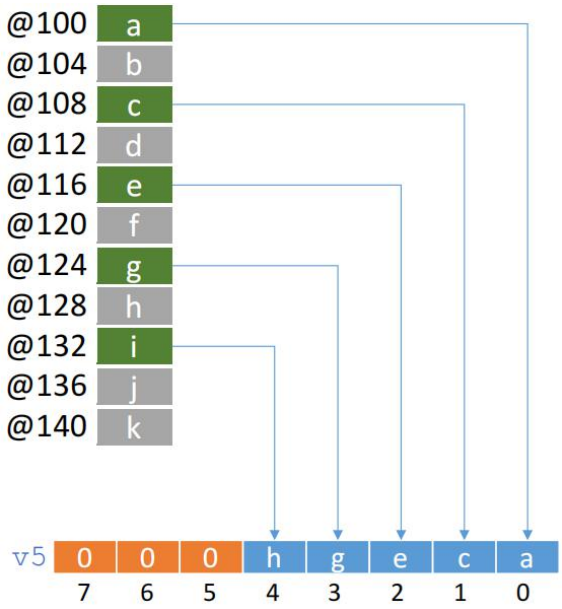
Vector load unit-stride



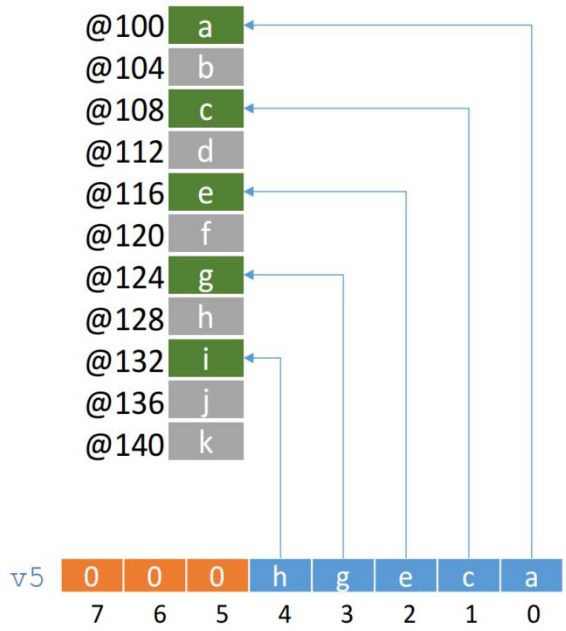
Vector store unit-stride



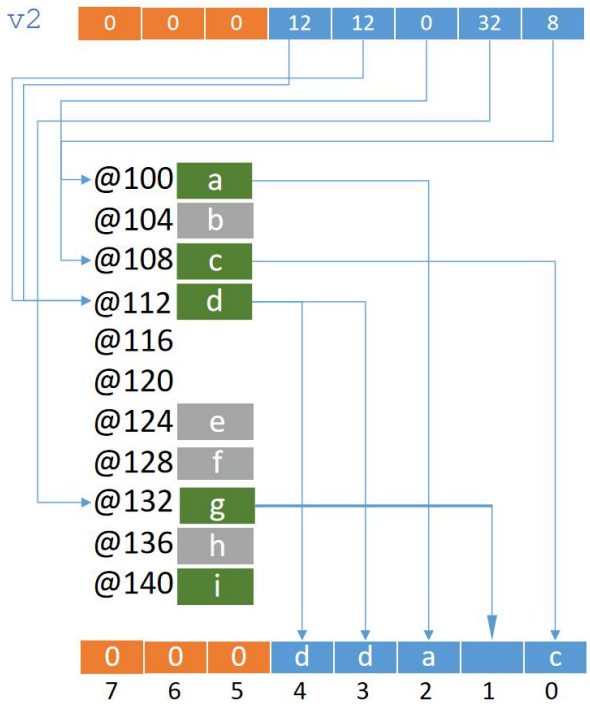
Vector load strided



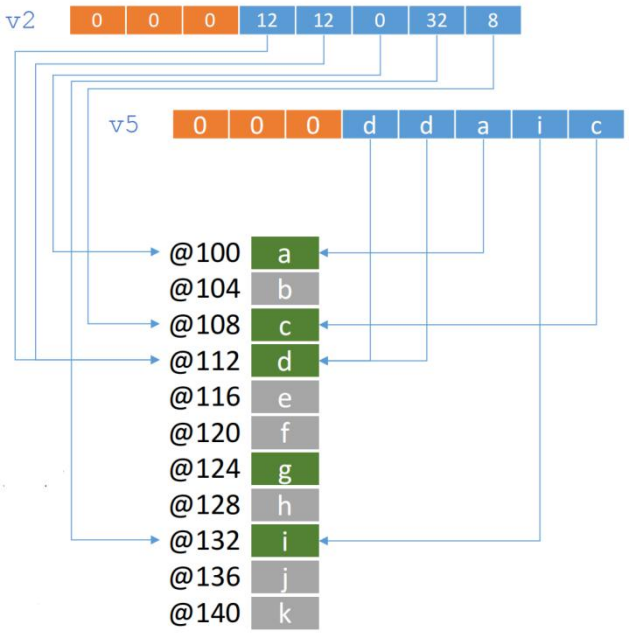
Vector store strided



Vector load indexed



Vector store indexed



5.2.1 Unit-stride

5.2.1.1 vlb.v vd, (rs1), vm

- vd[i] = sext(M[x[rs1] + i * 1][7:0])
- Vector Load Byte, unit-stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	100	vm	00000	rs1	000	rd	0000111

5.2.1.2 vlh.v vd, (rs1), vm

- vd[i] = sext(M[x[rs1] + i * 2][15:0])
- Vector Load Halfword, unit-stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	100	vm	00000	rs1	101	rd	0000111

5.2.1.3 vlw.v vd, (rs1), vm

- vd[i] = sext(M[x[rs1] + i * 4][31:0])
- Vector Load Word, unit-stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	100	vm	00000	rs1	110	rd	0000111

5.2.1.4 vlbu.v vd, (rs1), vm

- vd[i] = usext(M[x[rs1] + i * 1][7:0])
- Vector Load Byte, unsigned, unit-stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	000	rd	0000111

5.2.1.5 vlhu.v vd, (rs1), vm

- vd[i] = usext(M[x[rs1] + i * 2][15:0])
- Vector Load Halfword, unsigned, unit-stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	101	rd	0000111

5.2.1.6 vlwu.v vd, (rs1), vm

- vd[i] = usext(M[x[rs1] + i * 4][31:0])
- Vector Load Word, unsigned, unit-stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	110	rd	0000111

5.2.1.7 vle.v vd, (rs1), vm

- vd[i] = M[x[rs1] + i * SEW/8]
- Vector Load Element, unit-stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	111	rd	0000111

5.2.1.8 vsb.v vs3, (rs1), vm

- M[x[rs1] + i * 1] = vs3[i][7: 0]
- Vector Store Byte, unit stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	000	vs3	0100111

5.2.1.9 vsh.v vs3, (rs1), vm

- M[x[rs1] + i * 2] = vs3[i][15: 0]
- Vector Store Halfword, unit stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	101	vs3	0100111

5.2.1.10 vsw.v vs3, (rs1), vm

- M[x[rs1] + i * 4] = vs3[i][31: 0]
- Vector Store Word, unit stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	110	vs3	0100111

5.2.1.11 vse.v vs3, (rs1), vm

- M[x[rs1] + i * SEW/8] = vs3[i]
- Vector Store Element, unit stride.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	000	vm	00000	rs1	111	vs3	0100111

5.2.2 Strided

5.2.2.1 vlsb.v vd, (rs1), rs2, vm

- vd[i] = sext(M[x[rs1] + x[rs2] + i * 1][7:0])
- Vector Load Byte, strided.

31 29	28 26	25 24	20 19	15 14	12 11	7 6	0

000	110	vm	rs2	rs1	000	rd	0000111

5.2.2.2 vlsh.v vd, (rs1), rs2, vm

- vd[i] = sext(M[x[rs1] + x[rs2] + i * 2][15:0])
- Vector Load Halfword, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	110	vm	rs2	rs1	101	rd	0000111

5.2.2.3 vlsw.v vd, (rs1), rs2, vm

- vd[i] = sext(M[x[rs1] + x[rs2] + i * 4][31:0])
- Vector Load Word, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	110	vm	rs2	rs1	110	rd	0000111

5.2.2.4 vlsbu.v vd, (rs1), rs2, vm

- vd[i] = usext(M[x[rs1] + x[rs2] + i * 1][7:0])
- Vector Load Byte, unsigned, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	010	vm	rs2	rs1	000	rd	0000111

5.2.2.5 vlshu.v vd, (rs1), rs2, vm

- vd[i] = usext(M[x[rs1] + x[rs2] + i * 2][15:0])
- Vector Load Halfword, unsigned, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	010	vm	rs2	rs1	101	rd	0000111

5.2.2.6 vlswu.v vd, (rs1), rs2, vm

- vd[i] = usext(M[x[rs1] + x[rs2] + i * 4][31:0])
- Vector Load Word, unsigned, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0

000	010	vm	rs2	rs1	110	rd	0000111

5.2.2.7 vlse.v vd, (rs1), rs2, vm

- vd[i] = M[x[rs1] + x[rs2] + i * SEW/8]
- Vector Load Element, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	010	vm	rs2	rs1	111	rd	0000111

5.2.2.8 vssb.v vs3, (rs1), rs2, vm

- M[x[rs1] + x[rs2] + i * 1] = vs3[i][7: 0]
- Vector Store Byte, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	010	vm	rs2	rs1	000	vs3	0100111

5.2.2.9 vssh.v vs3, (rs1), rs2, vm

- M[x[rs1] + x[rs2] + i * 2] = vs3[i][15: 0]
- Vector Store Halfword, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	010	vm	rs2	rs1	101	vs3	0100111

5.2.2.10 vssw.v vs3, (rs1), rs2, vm

- M[x[rs1] + x[rs2] + i * 4] = vs3[i][31: 0]
- Vector Store Word, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
000	010	vm	rs2	rs1	110	vs3	0100111

5.2.2.11 vsse.v vs3, (rs1), rs2, vm

- M[x[rs1] + x[rs2] + i * SEW/8] = vs3[i]
- Vector Store Element, strided.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0

000	010	vm	rs2	rs1	111	vs3	0100111

5.2.3 Indexed

5.2.3.1 vlxb.v vd, (rs1), vs2, vm

- vd[i] = sext(M[x[rs1] +vs2[i]][7:0])
- Vector Load Byte, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
111	110	vm	rs2	rs1	000	rd	0000111

5.2.3.2 vlxh.v vd, (rs1), vs2, vm

- vd[i] = sext(M[x[rs1] +vs2[i]][15:0])
- Vector Load Halfword, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
111	110	vm	rs2	rs1	101	rd	0000111

5.2.3.3 vlxw.v vd, (rs1), vs2, vm

- vd[i] = sext(M[x[rs1] +vs2[i]][31:0])
- Vector Load Word, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
111	110	vm	rs2	rs1	110	rd	0000111

5.2.3.4 vlxbu.v vd, (rs1), vs2, vm

- vd[i] = usext(M[x[rs1] +vs2[i]][7:0])
- Vector Load Byte, unsigned, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
011	010	vm	rs2	rs1	000	rd	0000111

5.2.3.5 vlxhu.v vd, (rs1), vs2, vm

- vd[i] = usext(M[x[rs1] +vs2[i]][15:0])
- Vector Load Halfword, unsigned, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
011	010	vm	rs2	rs1	101	rd	0000111

5.2.3.6 vlxwu.v vd, (rs1), vs2, vm

- vd[i] = usext(M[x[rs1] +vs2[i]][31:0])
- Vector Load Word, unsigned, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
011	010	vm	rs2	rs1	110	rd	0000111

5.2.3.7 vlxe.v vd, (rs1), vs2, vm

- vd[i] = M[x[rs1] +vs2[i]]
- Vector Load Element, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
011	010	vm	rs2	rs1	111	rd	0000111

5.2.3.8 vsxb.v vs3, (rs1), vs2, vm

- M[x[rs1] +vs2[i]] = vs3[i][7: 0]
- Vector Store Byte, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
011	010	vm	rs2	rs1	000	vs3	0100111

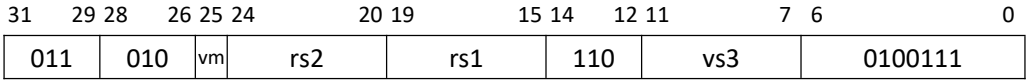
5.2.3.9 vsxh.v vs3, (rs1), vs2, vm

- M[x[rs1] +vs2[i]] = vs3[i][15: 0]
- Vector Store Halfword, indexed.

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
011	010	vm	rs2	rs1	101	vs3	0100111

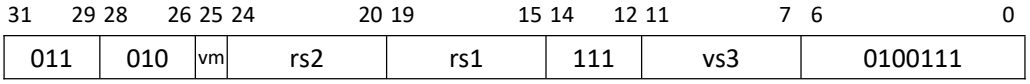
5.2.3.10 vsxw.v vs3, (rs1), vs2, vm

- M[x[rs1] +vs2[i]] = vs3[i][31: 0]
- Vector Store Word, indexed.



5.2.3.11 vsxe.v vs3, (rs1), vs2, vm

- M[x[rs1] +vs2[i]] = vs3[i]
- Vector Store Element, indexed.

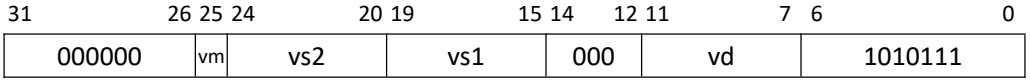


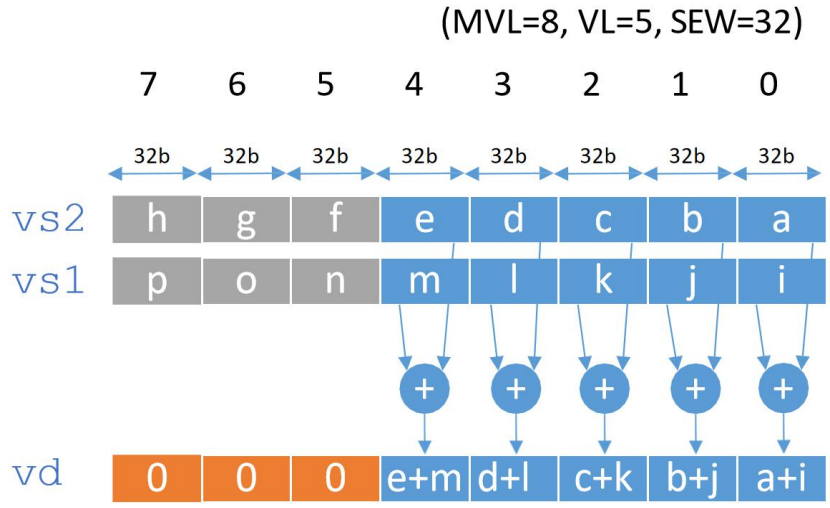
5.3 Vector Integer Arithmetic Instructions

5.3.1 Single-width integer add and subtract

5.3.1.1 vadd.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] + vs1[i]
- Vector Single-Width Integer Add

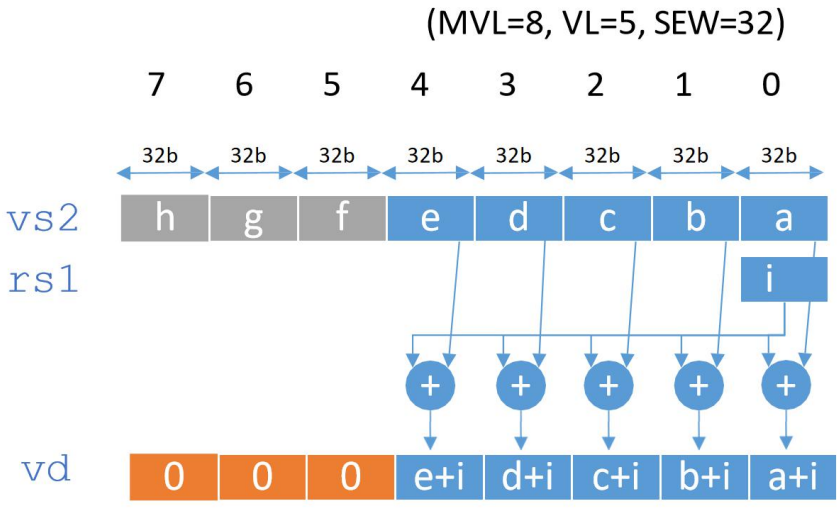




5.3.1.2 vadd.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] + x[rs1]
- Vector Single-Width Integer Add

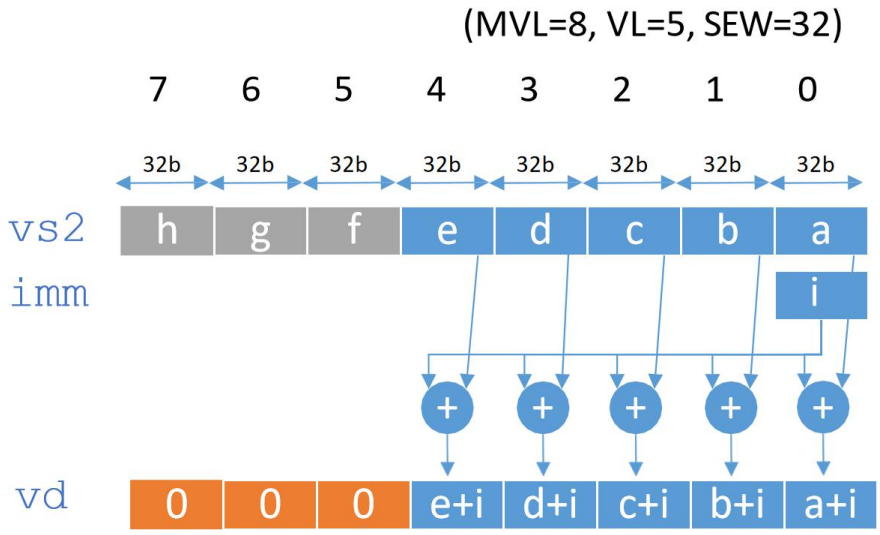
31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs2	rs1	100	vd	1010111



5.3.1.3 vadd.vi vd, vs2, imm, vm

- vd[i] = vs2[i] + imm
- Vector Single-Width Integer Add

31 26	25	24 20	19 15	14 12	11 7	6 0
000000	vm	vs2	simm5	011	vd	1010111



5.3.1.4 vsub.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] - vs1[i]
- Vector Single-Width Integer Subtract

31 26	25	24 20	19 15	14 12	11 7	6 0
000010	vm	vs2	vs1	000	vd	1010111

5.3.1.5 vsub.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] - x[rs1]
- Vector Single-Width Integer Subtract

31 26	25	24 20	19 15	14 12	11 7	6 0
000010	vm	vs2	rs1	100	vd	1010111

5.3.1.6 vrsub.vx vd, vs2, rs1, vm

- vd[i] = x[rs1] - vs2[i]
- Vector Single-Width Integer Reverse Subtract

31 26	25	24 20	19 15	14 12	11 7	6 0
000011	vm	vs2	vs1	000	vd	1010111

5.3.1.7 vrsub.vi vd, vs2, imm, vm

- vd[i] = imm - vs2[i]
- Vector Single-Width Integer Reverse Subtract

31 26	25	24 20	19 15	14 12	11 7	6 0
000011	vm	vs2	simm5	011	vd	1010111

5.3.2 Bitwise logical

5.3.2.1 vand.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] & vs1[i]
- Vector Bitwise Logical AND Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0
001001	vm	vs2	vs1	000	vd	1010111

5.3.2.2 vand.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] & x[rs1]
- Vector Bitwise Logical AND Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0
000010	vm	vs2	rs1	100	vd	1010111

5.3.2.3 vand.vi vd, vs2, imm, vm

- vd[i] = vs2[i] & sext(imm)
- Vector Bitwise Logical AND Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0
000010	vm	vs2	simm5	011	vd	1010111

5.3.2.4 vor.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] | vs1[i]
- Vector Bitwise Logical OR Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0

001010	vm	vs2	vs1	000	vd	1010111

5.3.2.5 vor.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] | x[rs1]
- Vector Bitwise Logical OR Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0
001010	vm	vs2	rs1	100	vd	1010111

5.3.2.6 vor.vi vd, vs2, imm, vm

- vd[i] = vs2[i] | sext(imm)
- Vector Bitwise Logical OR Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0
001010	vm	vs2	simm5	011	vd	1010111

5.3.2.7 vxor.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] ^ vs1[i]
- Vector Bitwise Logical XOR Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0
001011	vm	vs2	vs1	000	vd	1010111

5.3.2.8 vxor.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] ^ x[rs1]
- Vector Bitwise Logical XOR Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0
001011	vm	vs2	rs1	100	vd	1010111

5.3.2.9 vxor.vi vd, vs2, imm, vm

- vd[i] = vs2[i] ^ sext(imm)
- Vector Bitwise Logical XOR Instruction

31 26	25	24 20	19 15	14 12	11 7	6 0

001011	vm	vs2	simm5	011	vd	1010111

5.3.3 Single-width bit shift

5.3.3.1 vsll.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] << vs1[i]
- Vector Single-Width Logic Shift Left Instruction
- Only the low lg2(SEW) bits are read to obtain the shift amount from a register value.

31 26	25	24 20	19 15	14 12	11 7	6 0
100101	vm	vs2	vs1	000	vd	1010111

5.3.3.2 vsll.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] << x[rs1]
- Vector Single-Width Logic Shift Left Instruction
- Only the low lg2(SEW) bits are read to obtain the shift amount from a register value.

31 26	25	24 20	19 15	14 12	11 7	6 0
100101	vm	vs2	rs1	100	vd	1010111

5.3.3.3 vsll.vi vd, vs2, uimm, vm

- vd[i] = vs2[i] << uimm
- Vector Single-Width Logic Shift Left Instruction
- The immediate is treated as an unsigned shift amount, with a maximum shift amount of 31.

31 26	25	24 20	19 15	14 12	11 7	6 0
100101	vm	vs2	simm5	011	vd	1010111

5.3.3.4 vsrl.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] $>>_u$ vs1[i]
- Vector Single-Width Logic Shift Right Instruction
- Only the low lg2(SEW) bits are read to obtain the shift amount from a register value.

31 26	25	24 20	19 15	14 12	11 7	6 0
101000	vm	vs2	vs1	000	vd	1010111

5.3.3.5 vsrl.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] $>>_u$ x[rs1]
- Vector Single-Width Logic Shift Right Instruction
- Only the low lg2(SEW) bits are read to obtain the shift amount from a register value.

31 26	25	24 20	19 15	14 12	11 7	6 0
101000	vm	vs2	rs1	100	vd	1010111

5.3.3.6 vsrl.vi vd, vs2, uimm, vm

- vd[i] = vs2[i] $>>_u$ uinn
- Vector Single-Width Logic Shift Right Instruction
- The immediate is treated as an unsigned shift amount, with a maximum shift amount of 31.

31 26	25	24 20	19 15	14 12	11 7	6 0
101000	vm	vs2	simm5	011	vd	1010111

5.3.3.7 vsra.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] $>>_s$ vs1[i]
- Vector Single-Width Arithmetic Shift Right Instruction
- Only the low lg2(SEW) bits are read to obtain the shift amount from a register value.

31 26	25	24 20	19 15	14 12	11 7	6 0
101001	vm	vs2	vs1	000	vd	1010111

5.3.3.8 vsra.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] $>>_s$ x[rs1]
- Vector Single-Width Arithmetic Shift Right Instruction
- Only the low lg2(SEW) bits are read to obtain the shift amount from a register value.

31 26	25	24 20	19 15	14 12	11 7	6 0
101001	vm	vs2	rs1	100	vd	1010111

5.3.3.9 vsra.vi vd, vs2, uimm, vm

- vd[i] = vs2[i] $>>_s$ uimm
- Vector Single-Width Arithmetic Shift Right Instruction

- The immediate is treated as an unsigned shift amount, with a maximum shift amount of 31.

31 26	25	24 20	19 15	14 12	11 7	6 0
101001	vm	vs2	simm5	011	vd	1010111

5.3.4 Integer comparison

5.3.4.1 vmseq.vv vd, vs2, vs1, vm

- if(vs2[i] == vs1[i]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if equal
- The integer compare instructions write 1 to the destination mask register element if the comparison evaluates to true, and 0 otherwise. The destination mask vector is always held in a single vector register, with a layout of elements as described in Section Mask Register Layout.

31 26	25	24 20	19 15	14 12	11 7	6 0
011000	vm	vs2	vs1	000	vd	1010111

5.3.4.2 vmseq.vx vd, vs2, rs1, vm

- if(vs2[i] == x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if equal

31 26	25	24 20	19 15	14 12	11 7	6 0
011000	vm	vs2	rs1	100	vd	1010111

5.3.4.3 vmseq.vi vd, vs2, imm, vm

- if(vs2[i] == sext(imm)) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if equal

31 26	25	24 20	19 15	14 12	11 7	6 0
011000	vm	vs2	simm5	011	vd	1010111

5.3.4.4 vmsne.vv vd, vs2, vs1, vm

- if(vs2[i] != vs1[i]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if not equal

31 26	25	24 20	19 15	14 12	11 7	6 0

011001	vm	vs2	vs1	000	vd	1010111

5.3.4.5 vmsne.vx vd, vs2, rs1, vm

- if(vs2[i] != x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if not equal

31 26	25	24 20	19 15	14 12	11 7	6 0
011001	vm	vs2	rs1	100	vd	1010111

5.3.4.6 vmsne.vi vd, vs2, imm, vm

- if(vs2[i] != sext(imm)) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if not equal

31 26	25	24 20	19 15	14 12	11 7	6 0
011001	vm	vs2	simm5	011	vd	1010111

5.3.4.7 vmsltu.vv vd, vs2, vs1, vm

- if(vs2[i] < vs1[i]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
011010	vm	vs2	vs1	000	vd	1010111

5.3.4.8 vmsltu.vx vd, vs2, rs1, vm

- if(vs2[i] < x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
011010	vm	vs2	rs1	100	vd	1010111

5.3.4.9 vmslt.vv vd, vs2, vs1, vm

- if(vs2[i] < vs1[i]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than, signed

31 26	25	24 20	19 15	14 12	11 7	6 0

011011	vm	vs2	vs1	000	vd	1010111

5.3.4.10 vmslt.vx vd, vs2, rs1, vm

- if(vs2[i] < x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than, signed

31 26	25	24 20	19 15	14 12	11 7	6 0
011011	vm	vs2	rs1	100	vd	1010111

5.3.4.11 vmsleu.vv vd, vs2, vs1, vm

- if(vs2[i] <= vs1[i]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than or equal, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
011100	vm	vs2	vs1	000	vd	1010111

5.3.4.12 vmsleu.vx vd, vs2, rs1, vm

- if(vs2[i] <= x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than or equal, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
011100	vm	vs2	rs1	100	vd	1010111

5.3.4.13 vmsleu.vi vd, vs2, imm, vm

- if(vs2[i] <= usext(imm)) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than or equal, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
011100	vm	vs2	simm5	011	vd	1010111

5.3.4.14 vmsle.vv vd, vs2, vs1, vm

- if(vs2[i] <= rs1) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than or equal, signed

31 26	25	24 20	19 15	14 12	11 7	6 0

011101	vm	vs2	vs1	000	vd	1010111

5.3.4.15 vmsle.vx vd, vs2, rs1, vm

- if(vs2[i] <= x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than or equal, signed

31 26	25	24 20	19 15	14 12	11 7	6 0
011101	vm	vs2	rs1	100	vd	1010111

5.3.4.16 vmsle.vi vd, vs2, imm, vm

- if(vs2[i] <= sext(imm)) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if less than or equal, signed

31 26	25	24 20	19 15	14 12	11 7	6 0
011101	vm	vs2	simm5	011	vd	1010111

5.3.4.17 vmsgtu.vx vd, vs2, rs1, vm

- if(vs2[i] > x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if greater than, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
011110	vm	vs2	rs1	100	vd	1010111

5.3.4.18 vmsgtu.vi vd, vs2, imm, vm

- if(vs2[i] > usext(imm)) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if greater than, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
011110	vm	vs2	rs1	100	vd	1010111

5.3.4.19 vmsgt.vx vd, vs2, rs1, vm

- if(vs2[i] > x[rs1]) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if greater than, signed

31 26	25	24 20	19 15	14 12	11 7	6 0

011111	vm	vs2	rs1	100	vd	1010111

5.3.4.20 vmsgt.vi vd, vs2, imm, vm

- if(vs2[i] > sext(imm)) set vd[i] = 1; else set vd[i] = 0
- Vector Integer Instruction,Set if greater than, signed

31 26	25	24 20	19 15	14 12	11 7	6 0
011111	vm	vs2	simm5	011	vd	1010111

5.3.5 Integer min/max

5.3.5.1 vminu.vv vd, vs2, vs1, vm

- vd[i] = minu(vs2[i], vs1[i])
- Vector Integer Min Instruction, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
000100	vm	vs2	vs1	000	vd	1010111

5.3.5.2 vminu.vx vd, vs2, rs1, vm

- vd[i] = minu(vs2[i], x[rs1])
- Vector Integer Min Instruction, unsigned

31 26	25	24 20	19 15	14 12	11 7	6 0
000100	vm	vs2	rs1	100	vd	1010111

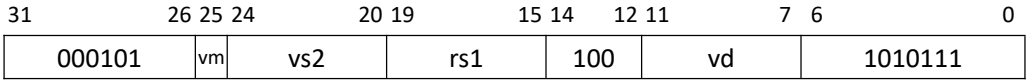
5.3.5.3 vmin.vv vd, vs2, vs1, vm

- vd[i] = min(vs2[i], vs1[i])
- Vector Integer Min Instruction, signed

31 26	25	24 20	19 15	14 12	11 7	6 0
000101	vm	vs2	vs1	000	vd	1010111

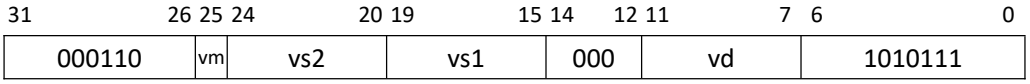
5.3.5.4 vmin.vx vd, vs2, rs1, vm

- vd[i] = min(vs2[i], x[rs1])
- Vector Integer Min Instruction, signed



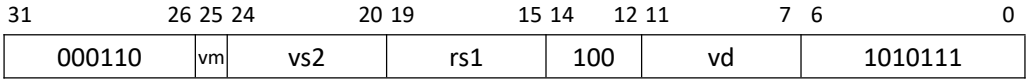
5.3.5.5 vmaxu.vv vd, vs2, vs1, vm

- vd[i] = maxu(vs2[i], vs1[i])
- Vector Integer Min Instruction, unsigned



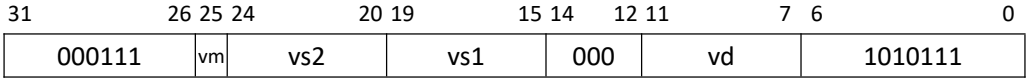
5.3.5.6 vmaxu.vx vd, vs2, rs1, vm

- vd[i] = maxu(vs2[i], x[rs1])
- Vector Integer Min Instruction, unsigned



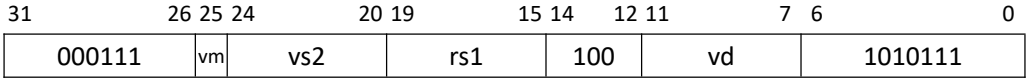
5.3.5.7 vmax.vv vd, vs2, vs1, vm

- vd[i] = max(vs2[i], vs1[i])
- Vector Integer Min Instruction, signed



5.3.5.8 vmax.vx vd, vs2, rs1, vm

- vd[i] = max(vs2[i],x[rs1])
- Vector Integer Min Instruction, signed



5.3.6 Single-width integer multiply

5.3.6.1 vmul.vv vd, vs2, vs1, vm

- vd[i] = vs2[i] * vs1[i]
- Vector Single-Width Integer Multiply Instruction
- Signed multiply, returning low bits of product

31 26	25	24 20	19 15	14 12	11 7	6 0
100101	vm	vs2	vs1	010	vd	1010111

5.3.6.2 vmul.vx vd, vs2, rs1, vm

- vd[i] = vs2[i] * x[rs1]
- Vector Single-Width Integer Multiply Instruction
- Signed multiply, returning low bits of product

31 26	25	24 20	19 15	14 12	11 7	6 0
100101	vm	vs2	rs1	110	vd	1010111

5.3.6.3 vmulh.vv vd, vs2, vs1, vm

- vd[i] = (vs2[i] * vs1[i]) $>>_s$ SEW
- Vector Single-Width Integer Multiply Instruction
- Signed multiply, returning high bits of product

31 26	25	24 20	19 15	14 12	11 7	6 0
100111	vm	vs2	vs1	010	vd	1010111

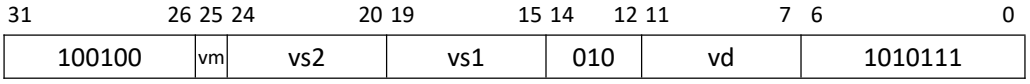
5.3.6.4 vmulh.vx vd, vs2, rs1, vm

- vd[i] = (vs2[i] * x[rs1]) $>>_s$ SEW
- Vector Single-Width Integer Multiply Instruction
- Signed multiply, returning high bits of product

31 26	25	24 20	19 15	14 12	11 7	6 0
100111	vm	vs2	rs1	110	vd	1010111

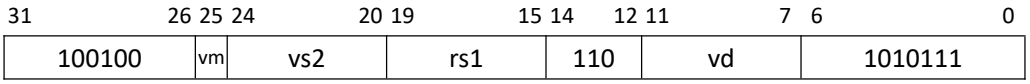
5.3.6.5 vmulhu.vv vd, vs2, vs1, vm

- vd[i] = (vs2[i] * vs1[i]) >>$_u$ SEW
- Vector Single-Width Integer Multiply Instruction
- Unsigned multiply, returning high bits of product



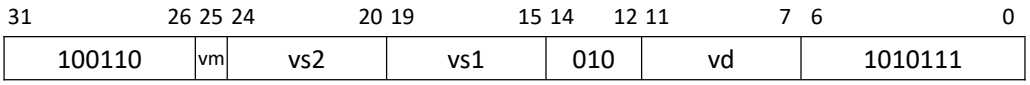
5.3.6.6 vmulhu.vx vd, vs2, rs1, vm

- vd[i] = (vs2[i] * x[rs1]) >>$_u$ SEW
- Vector Single-Width Integer Multiply Instruction
- Unsigned multiply, returning high bits of product



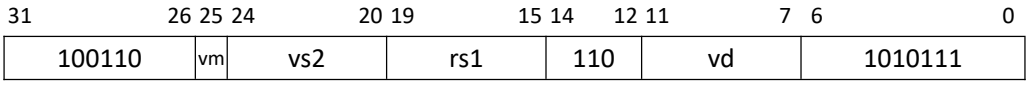
5.3.6.7 vmulhsu.vv vd, vs2, vs1, vm

- vd[i] = (vs2[i] * vs1[i]) >>$_s$ SEW
- Vector Single-Width Integer Multiply Instruction
- Signed(vs2)-Unsigned multiply, returning high bits of product



5.3.6.8 vmulhsu.vx vd, vs2, rs1, vm

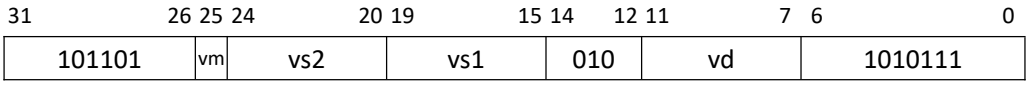
- vd[i] = (vs2[i] * x[rs1]) >>$_s$ SEW
- Vector Single-Width Integer Multiply Instruction
- Signed(vs2)-Unsigned multiply, returning high bits of product



5.3.7 Single-width integer multiply-add

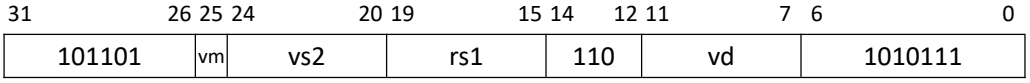
5.3.7.1 vmacc.vv vd, vs1, vs2, vm

- vd[i] = +(vs1[i] * vs2[i]) + vd[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-add, overwrite addend



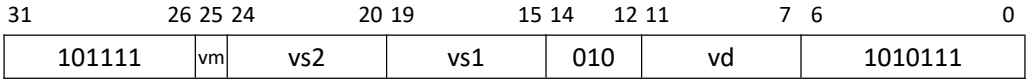
5.3.7.2 vmacc.vx vd, rs1, vs2, vm

- vd[i] = +(x[rs1] * vs2[i]) + vd[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-add, overwrite addend



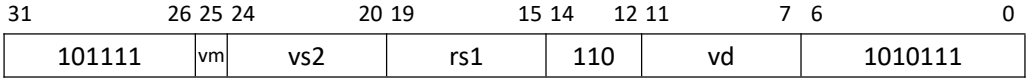
5.3.7.3 vnmsac.vv vd, vs1, vs2, vm

- vd[i] = -(vs1[i] * vs2[i]) + vd[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-sub, overwrite minuend



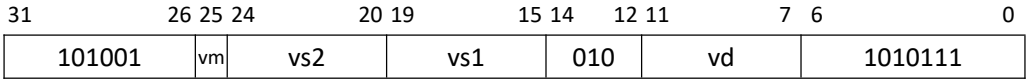
5.3.7.4 vnmsac.vx vd, rs1, vs2, vm

- vd[i] = -(x[rs1] * vs2[i]) + vd[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-sub, overwrite minuend



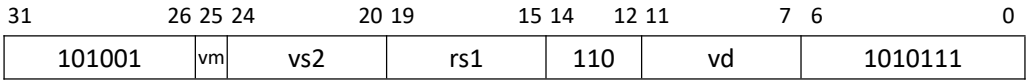
5.3.7.5 vmadd.vv vd, vs1, vs2, vm

- vd[i] = +(vs1[i] * vs2[i]) + vd[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-add, overwrite multiplicand



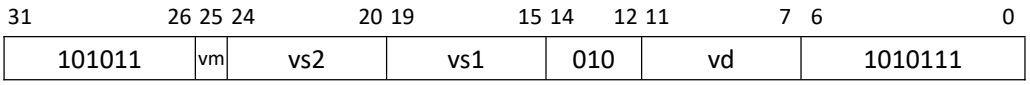
5.3.7.6 vmadd.vx vd, rs1, vs2, vm

- vd[i] = +(x[rs1] * vs2[i]) + vd[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-add, overwrite multiplicand



5.3.7.7 vnmsub.vv vd, vs1, vs2, vm

- vd[i] = -(vs1[i] * vd[i]) + vs2[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-sub, overwrite multiplicand



5.3.7.8 vnmsub.vx vd, rs1, vs2, vm

- vd[i] = -(x[rs1] * vd[i]) + vs2[i]
- Vector Single-Width Integer Multiply-Add Instructions
- Integer multiply-sub, overwrite multiplicand