RISC-V Reference

James Zhu <jameszhu@berkeley.edu>

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

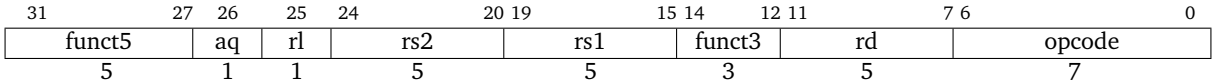
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
`add`	ADD	R	0110011	000	0000000	`rd = rs1 + rs2`	
`sub`	SUB	R	0110011	000	0100000	`rd = rs1 - rs2`	
`xor`	XOR	R	0110011	100	0000000	`rd = rs1 ^ rs2`	
`or`	OR	R	0110011	110	0000000	`rd = rs1 | rs2`	
`and`	AND	R	0110011	111	0000000	`rd = rs1 & rs2`	
`sll`	Shift Left Logical	R	0110011	001	0000000	`rd = rs1 << rs2`	
`srl`	Shift Right Logical	R	0110011	010	0000000	`rd = rs1 >> rs2`	
`sra`	Shift Right Arith*	R	0110011	011	0100000	`rd = rs1 >> rs2`	msb-extends
`slt`	Set Less Than	R	0110011	010		`rd = (rs1 < rs2)?1:0`	
`sltu`	Set Less Than (U)	R	0110011	011		`rd = (rs1 < rs2)?1:0`	zero-extends
`addi`	ADD Immediate	I	0010011	000	0000000	`rd = rs1 + imm`	
`xori`	XOR Immediate	I	0010011	100	0000000	`rd = rs1 ^ imm`	
`ori`	OR Immediate	I	0010011	110	0000000	`rd = rs1 | imm`	
`andi`	AND Immediate	I	0010011	111	0000000	`rd = rs1 & imm`	
`slli`	Shift Left Logical Imm	I	0010011	001	0000000	`rd = rs1 << imm`	
`srli`	Shift Right Logical Imm	I	0010011	001	0000000	`rd = rs1 >> imm`	
`srai`	Shift Right Arith Imm	I	0010011	011	0100000	`rd = rs1 >> imm`	msb-extends
`slti`	Set Less Than Imm	I	0010011	010		`rd = (rs1 < imm)?1:0`	
`sltiu`	Set Less Than Imm (U)	I	0010011	011		`rd = (rs1 < imm)?1:0`	zero-extends
`lb`	Load Byte	I	0000011	000		`rd = M[rs1+imm][0:7]`	
`lh`	Load Half	I	0000011	001		`rd = M[rs1+imm][0:15]`	
`lw`	Load Word	I	0000011	010		`rd = M[rs1+imm][0:31]`	
`lbu`	Load Byte (U)	I	0000011	100		`rd = M[rs1+imm][0:7]`	zero-extends
`lhu`	Load Half (U)	I	0000011	101		`rd = M[rs1+imm][0:15]`	zero-extends
`sb`	Store Byte	S	0100011	000		`M[rs1+imm][0:7] = rs2[0:7]`	
`sh`	Store Half	S	0100011	001		`M[rs1+imm][0:15] = rs2[0:15]`	
`sw`	Store Word	S	0100011	010		`M[rs1+imm][0:31] = rs2[0:31]`	
`beq`	Branch ==	B	1100011	000		`if(rs1 == rs2) PC += imm`	
`bne`	Branch !=	B	1100011	001		`if(rs1 != rs2) PC += imm`	
`blt`	Branch <	B	1100011	100		`if(rs1 < rs2) PC += imm`	
`bge`	Branch ≥	B	1100011	101		`if(rs1 >= rs2) PC += imm`	
`bltu`	Branch < (U)	B	1100011	110		`if(rs1 < rs2) PC += imm`	zero-extends
`bgeu`	Branch ≥ (U)	B	1100011	111		`if(rs1 >= rs2) PC += imm`	zero-extends
`jal`	Jump And Link	J	1101111			`rd = PC+4; PC += imm`	
`jalr`	Jump And Link Reg	I	1100111	000		`rd = PC+4; PC = rs1 + imm`	
`lui`	Load Upper Imm	U	0110111			`rd = imm << 12`	
`auipc`	Add Upper Imm to PC	U	0010111			`rd = PC + (imm << 12)`	
`ecall`	Environment Call	I	1110011	000	0000000	`Transfer control to OS`	imm: 0x000
`ebreak`	Environment Break	I	1110011	000	0000000	`Transfer control to debugger`	imm: 0x001

Standard Extensions

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	0110011	000	0000001	`rd = (rs1 * rs2)[31:0]`
`mulh`	MUL High	R	0110011	001	0000001	`rd = (rs1 * rs2)[63:32]`
`mulsu`	MUL High (S) (U)	R	0110011	010	0000001	`rd = (rs1 * rs2)[63:32]`
`mulu`	MUL High (U)	R	0110011	011	0000001	`rd = (rs1 * rs2)[63:32]`
`div`	DIV	R	0110011	100	0000001	`rd = rs1 / rs2`
`divu`	DIV (U)	R	0110011	101	0000001	`rd = rs1 / rs2`
`rem`	Remainder	R	0110011	110	0000001	`rd = rs1 % rs2`
`remu`	Remainder (U)	R	0110011	111	0000001	`rd = rs1 % rs2`

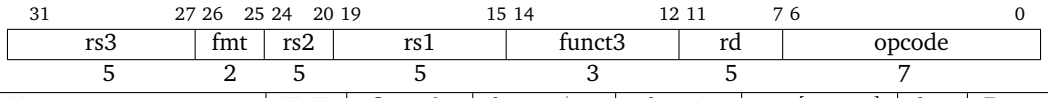
RV32A Atomic Extension



Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
`lr.w`	Load Reserved	R	0101111	010	00010	`rd = M[rs1], reserve M[rs1]`
`sc.w`	Store Conditional	R	0101111	010	00011	`if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }`
`amoswap.w`	Atomic Swap	R	0101111	010	00001	`rd = M[rs1]; swap(rd, rs2); M[rs1] = rd`
`amoadd.w`	Atomic ADD	R	0101111	010	00000	`rd = M[rs1] + rs2; M[rs1] = rd`
`amoand.w`	Atomic AND	R	0101111	010	01100	`rd = M[rs1] & rs2; M[rs1] = rd`
`amoor.w`	Atomic OR	R	0101111	010	01010	`rd = M[rs1] | rs2; M[rs1] = rd`
`amoxor.w`	Atomix XOR	R	0101111	010	00100	`rd = M[rs1] ^ rs2; M[rs1] = rd`
`amomax.w`	Atomic MAX	R	0101111	010	10100	`rd = max(M[rs1], rs2); M[rs1] = rd`
`amomin.w`	Atomic MIN	R	0101111	010	10000	`rd = min(M[rs1], rs2); M[rs1] = rd`

RV32F Single-Precision Floating-Point Extension

R4-type instructions



Inst	Name	FMT	Opcode	funct3/rm	funct7	rs2 [20...24]	fmt	Description (C)
`flw`	Flt Load Word	I	0000111	010				`rd = M[rs1 + imm]`
`fsw`	Flt Store Word	I	0100111	010				`M[rs1 + imm] = rs2`
`fadd.s`	Flt Add	R	1010011		0000000			`rd = rs1 + rs2`
`fsub.s`	Flt Sub	R	1010011		0000100			`rd = rs1 - rs2`
`fmul.s`	Flt Mul	R	1010011		0001000			`rd = rs1 * rs2`
`fdiv.s`	Flt Div	R	1010011		0001100			`rd = rs1 / rs2`
`fsgnj.s`	Flt Sign Injection	R	1010011	000	0010000			`rd = abs(rs1) * sgn(rs2)`
`fsgnjn.s`	Flt Sign Neg Injection	R	1010011	001	0010000			`rd = abs(rs1) * -sgn(rs2)`
`fsgnjx.s`	Flt Sign Xor Injection	R	1010011	010	0010000			`rd = rs1 * sgn(rs2)`
`fmin.s`	Flt Minimum	R	1010011		0010100			`rd = min(rs1, rs2)`
`fmax.s`	Flt Maximum	R	1010011		0010100			`rd = max(rs1, rs2)`
`fsqrt.s`	Flt Square Root	R	1010011		0101100	00000		`rd = sqrt(rs1)`
`fle.s`	Float Less / Equal	R	1010011	000	1010000			`rd = (rs1 <= rs2) ? 1 : 0`
`flt.s`	Float Less Than	R	1010011	001	1010000			`rd = (rs1 < rs2) ? 1 : 0`
`feq.s`	Float Equality	R	1010011	010	1010000			`rd = (rs1 == rs2) ? 1 : 0`
`fcvt.w.s`	Flt Convert to Int	R	1010011		1100000	00000		`rd = (int32_t) rs1`
`fcvt.wu.s`	Flt Convert to Int	R	1010011		1100000	00001		`rd = (uint32_t) rs1`
`fmv.x.w`	Move Float to Int	R	1010011	000	1110000	00000		`rd = *((int*) &rs1)`
`fclass.s`	Float Classify	R	1010011	001	1110000	00000		`rd = 0..9`
`fcvt.s.w`	Flt Conv from Sign Int	R	1010011		1101000	00000		`rd = (float) rs1`
`fcvt.s.wu`	Flt Conv from Uns Int	R	1010011		1101000	00001		`rd = (float) rs1`
`fmv.w.x`	Move Int to Float	R	1010011	000	1111000	00000		`rd = *((float*) &rs1)`
`fmadd.s`	Flt Fused Mul-Add	R4	1000011				00	`rd = rs1 * rs2 + rs3`
`fmsub.s`	Flt Fused Mul-Sub	R4	1000111				00	`rd = rs1 * rs2 - rs3`
`fnmadd.s`	Flt Neg Fused Mul-Add	R4	1001011				00	`rd = -rs1 * rs2 + rs3`
`fnmsub.s`	Flt Neg Fused Mul-Sub	R4	1001111				00	`rd = -rs1 * rs2 - rs3`

RV32D Double-Precision Floating-Point Extension

Inst	Name	FMT	Opcode	funct3/rm	funct7	rs2 [20...24]	fmt	Description (C)
fld	Flt Load Word	I	0000111	011				rd = M[rs1 + imm]
fsd	Flt Store Word	I	0100111	011				M[rs1 + imm] = rs2
fadd.d	Flt Add	R	1010011		0000001			rd = rs1 + rs2
fsub.d	Flt Sub	R	1010011		0000101			rd = rs1 - rs2
fmul.d	Flt Mul	R	1010011		0001001			rd = rs1 * rs2
fdiv.d	Flt Div	R	1010011		0001101			rd = rs1 / rs2
fsgnj.d	Flt Sign Injection	R	1010011	000	0010001			rd = abs(rs1) * sgn(rs2)
fsgnjn.d	Flt Sign Neg Injection	R	1010011	001	0010001			rd = abs(rs1) * -sgn(rs2)
fsgnjx.d	Flt Sign Xor Injection	R	1010011	010	0010001			rd = rs1 * sgn(rs2)
fmin.d	Flt Minimum	R	1010011		0010101			rd = min(rs1, rs2)
fmax.d	Flt Maximum	R	1010011		0010101			rd = max(rs1, rs2)
fsqrt.d	Flt Square Root	R	1010011		0101101	00000		rd = sqrt(rs1)
fle.d	Float Less / Equal	R	1010011	000	1010001			rd = (rs1 <= rs2) ? 1 : 0
flt.d	Float Less Than	R	1010011	001	1010001			rd = (rs1 < rs2) ? 1 : 0
feq.d	Float Equality	R	1010011	010	1010001			rd = (rs1 == rs2) ? 1 : 0
fcvt.w.d	Flt Convert to Int	R	1010011		1100001	00000		rd = (int32_t) rs1
fcvt.wu.d	Flt Convert to Int	R	1010011		1100001	00001		rd = (uint32_t) rs1
fcvt.s.d	Double Flt to Single	R	1010011	000	0100000	00001		rd = *((float*) &rs1)
fmv.x.d	Move Int to Float	R	1010011	000	1111001	00000		rd = *((float*) &rs1)
fclass.d	Float Classify	R	1010011	001	1110001	00000		rd = 0..9
fcvt.d.w	Flt Conv from Sign Int	R	1010011		1101001	00000		rd = (float) rs1
fcvt.d.wu	Flt Conv from Uns Int	R	1010011		1101001	00001		rd = (float) rs1
fcvt.d.s	Single Flt to Double	R	1010011	000	0100001	00001		rd = *((float*) &rs1)
fmadd.d	Flt Fused Mul-Add	R4	1000011				01	rd = rs1 * rs2 + rs3
fmsub.d	Flt Fused Mul-Sub	R4	1000111				01	rd = rs1 * rs2 - rs3
fnmadd.d	Flt Neg Fused Mul-Add	R4	1001011				01	rd = -rs1 * rs2 + rs3
fnmsub.d	Flt Neg Fused Mul-Sub	R4	1001111				01	rd = -rs1 * rs2 - rs3

RV32Q Quad-Precision Floating-Point Extension

Inst	Name	FMT	Opcode	funct3/rm	funct7	rs2 [20...24]	fmt	Description (C)
flq	Flt Load Word	I	0000111	100				rd = M[rs1 + imm]
fsq	Flt Store Word	I	0100111	100				M[rs1 + imm] = rs2
fadd.q	Flt Add	R	1010011		0000011			rd = rs1 + rs2
fsub.q	Flt Sub	R	1010011		0000111			rd = rs1 - rs2
fmul.q	Flt Mul	R	1010011		0001011			rd = rs1 * rs2
fdiv.q	Flt Div	R	1010011		0001111			rd = rs1 / rs2
fsgnj.q	Flt Sign Injection	R	1010011	000	0010011			rd = abs(rs1) * sgn(rs2)
fsgnjn.q	Flt Sign Neg Injection	R	1010011	001	0010011			rd = abs(rs1) * -sgn(rs2)
fsgnjx.q	Flt Sign Xor Injection	R	1010011	010	0010011			rd = rs1 * sgn(rs2)
fmin.q	Flt Minimum	R	1010011		0010111			rd = min(rs1, rs2)
fmax.q	Flt Maximum	R	1010011		0010111			rd = max(rs1, rs2)
fsqrt.q	Flt Square Root	R	1010011		0101111	00000		rd = sqrt(rs1)
fle.q	Float Less / Equal	R	1010011	000	1010011			rd = (rs1 <= rs2) ? 1 : 0
flt.q	Float Less Than	R	1010011	001	1010011			rd = (rs1 < rs2) ? 1 : 0
feq.q	Float Equality	R	1010011	010	1010011			rd = (rs1 == rs2) ? 1 : 0
fcvt.w.q	Flt Convert to Int	R	1010011		1100011	00000		rd = (int32_t) rs1
fcvt.wu.q	Flt Convert to Int	R	1010011		1100011	00001		rd = (uint32_t) rs1
fcvt.s.q	Quad Flt to Single	R	1010011	000	0100000	00001		rd = *((float*) &rs1)
fcvt.d.q	Quad Flt to Double	R	1010011	000	0100001	00001		rd = *((float*) &rs1)
fmv.x.q	Move Int to Float	R	1010011	000	1111011	00000		rd = *((float*) &rs1)
fclass.q	Float Classify	R	1010011	001	1110011	00000		rd = 0..9
fcvt.q.w	Flt Conv from Sign Int	R	1010011		1101011	00000		rd = (float) rs1
fcvt.q.wu	Flt Conv from Uns Int	R	1010011		1101011	00001		rd = (float) rs1
fcvt.q.s	Single Flt to Quad	R	1010011	000	0100011	00000		rd = *((float*) &rs1)
fcvt.q.d	Double Flt to Quad	R	1010011	000	0100011	00001		rd = *((float*) &rs1)
fmadd.q	Flt Fused Mul-Add	R4	1000011				11	rd = rs1 * rs2 + rs3
fmsub.q	Flt Fused Mul-Sub	R4	1000111				11	rd = rs1 * rs2 - rs3
fnmadd.q	Flt Neg Fused Mul-Add	R4	1001011				11	rd = -rs1 * rs2 + rs3
fnmsub.q	Flt Neg Fused Mul-Sub	R4	1001111				11	rd = -rs1 * rs2 - rs3

RV32C Compressed Extension

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
funct4		rd/rs1		rs2		op	CR-type
funct3	imm	rd/rs1		imm		op	CI-type
funct3	imm			rs2		op	CSS-type
funct3	imm				rd’	op	CIW-type
funct3	imm	rs1’		imm	rd’	op	CL-type
funct3	imm	rd’/rs1’		imm	rs2’	op	CS-type
funct3	imm	rs1’		imm		op	CB-type
funct3	offset					op	CJ-type

Inst	Name	FMT	OP	funct4	funct3	[12]	[11..10]	[6..5]	Description
`c.addi4spn`	ADD Imm * 4 + SP	CIW	00		000				`addi rd’, sp, 4*imm`
`c.fld`	Flt Load Double	CL	00		001				`flw rd’, (8*imm)(rs1’)`
`c.lw`	Load Word	CL	00		010				`lw rd’, (4*imm)(rs1’)`
`c.flw`	Flt Load Word	CL	00		011				`flw rd’, (4*imm)(rs1’)`
`c.fsd`	Flt Store Double	CS	00		101				`fsd rs1’, (8*imm)(rs2’)`
`c.sw`	Store Word	CS	00		110				`sw rs1’, (4*imm)(rs2’)`
`c.fsw`	Flt Store Word	CS	00		111				`fsw rs1’, (4*imm)(rs2’)`
`c.addi`	ADD Immediate	CI	01		000				`addi rd, rd, imm`
`c.jal`	Jump And Link	CJ	01		001				`jal ra, 2*offset`
`c.li`	Load Immediate	CI	01		010				`addi rd, x0, imm`
`c.lui`	Load Upper Imm	CI	01		011				`lui rd, imm`
`c.addi16sp`	ADD Imm * 16 to SP	CI	01		011				`addi sp, sp, 16*imm`
`c.srli`	Shift Right Logical Imm	CB	01		100		00		`srli rd’, rd’, imm`
`c.srai`	Shift Right Arith Imm	CB	01		100		01		`srai rd’, rd’, imm`
`c.andi`	AND Imm	CB	01		100		10		`andi rd’, rd’, imm`
`c.sub`	SUB	CS	01		100	0	11	00	`sub rd’, rd’, rs2’`
`c.xor`	XOR	CS	01		100	0	11	01	`xor rd’, rd’, rs2’`
`c.or`	OR	CS	01		100	0	11	10	`or rd’, rd’, rs2’`
`c.and`	AND	CS	01		100	0	11	11	`and rd’, rd’, rs2’`
`c.subw`	SUB Word	CS	01		100	1	11	00	`subw rd’, rd’, rs2’`
`c.addw`	ADD Word	CS	01		100	1	11	01	`addw rd’, rd’, rs2’`
`c.j`	Jump	CJ	01		101				`jal x0, 2*offset`
`c.beqz`	Branch == 0	CB	01		110				`beq rs’, x0, 2*imm`
`c.bnez`	Branch != 0	CB	01		111				`bne rs’, x0, 2*imm`
`c.slli`	Shift Left Logical Imm	CI	10		000				`slli rd, rd, imm`
`c.lwsp`	Load Word from SP	CI	10		010				`lw rd, (4*imm)(sp)`
`c.fldsp`	Flt Load Double from SP	CI	10		001				`flw rd, (8*imm)(sp)`
`c.flwsp`	Flt Load Word from SP	CI	10		011				`flw rd, (4*imm)(sp)`
`c.fswsp`	Flt Store Word to SP	CSS	10		111				`fsw rs2, (4*imm)(sp)`
`c.mv`	MoVe	CR	10	1000					`add rd, x0, rs2`
`c.add`	ADD	CR	10	1001					`add rd, rd, rs2`
`c.fsdsp`	Flt Store Double to SP	CSS	10		101				`fsd rs2, (8*imm)(sp)`
`c.swsp`	Store Word to SP	CSS	10		110				`sw rs2, (4*imm)(sp)`

RV64I Base Integer Instructions (in addition to RV32I)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
addw	Add registers	R	0111011	000	0000000	
subw	Subtract registers	R	0111011	000	0100000	
sllw	Shift left logical by register	R	0111011	001	0000000	
srlw	Shift right logical by register	R	0111011	101	0000000	
sraw	Shift right arithmetic by register	R	0111011	101	0100000	
addiw	Add immediate	I	0011011	000		
slliw	Shift left logical by immediate	I	0011011	001	0000000	
srliw	Shift right logical by immediate	I	0011011	101	0000000	
sraiw	Shift right arithmetic by immediate	I	0011011	101	0100000	
lwu	Load word, unsigned	I	0000011	110		
ld	Load double-word	I	0000011	000		
sd	Store double-word	S	0100011	011		

RV64M Multiply Extension (in addition to RV32M)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mulw	Multiply	R	0111011	000	0000001	
divw	Signed division	R	0111011	100	0000001	
divuw	Unsigned division	R	0111011	101	0000001	
remw	Signed remainder	R	0111011	110	0000001	
remuw	Unsigned remainder	R	0111011	111	0000001	

RV64A Atomic Extension (in addition to RV32A)

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
lr.d	Load reserved	R	0101111	011	00010	
sc.d	Store conditional	R	0101111	011	00011	
amoswap.d	Atomic swap	R	0101111	011	00001	
amoadd.d	Atomic addition	R	0101111	011	00000	
amoxor.d	Atomic bitwise XOR	R	0101111	011	00100	
amoand.d	Atomic bitwise AND	R	0101111	011	01100	
amoor.d	Atomic bitwise OR	R	0101111	011	01000	
amomin.d	Atomic two's complement minimum	R	0101111	011	10000	
amomax.d	Atomic two's complement maximum	R	0101111	011	10100	
amominu.d	Atomic unsigned minimum	R	0101111	011	11000	
amomaxu.d	Atomic unsigned maximum	R	0101111	011	11100	

RV64F Single-Precision Floating-Point Extension (in addition to RV32F)

Inst	Name	FMT	Opcode	rs2	funct7	Description (C)
fcvt.l.s	Convert to signed 64-bit integer	R	1010011	00010	1100000	
fcvt.lu.s	Convert to unsigned 64-bit integer	R	1010011	00011	1100000	
fcvt.s.l	Convert from signed 64-bit integer	R	1010011	00010	1101000	
fcvt.s.lu	Convert from unsigned 64-bit integer	R	1010011	00011	1101000	

RV64D Double-Precision Floating-Point Extension (in addition to RV32D)

Inst	Name	FMT	Opcode	funct3/rm	rs2	funct7	Description (C)
fmv.x.d	Move from F.P. to integer register	R	1010011	000	00000	1110001	
fmv.d.x	Move from integer to F.P. register	R	1010011	000	00000	1111001	
fcvt.l.d	Convert to signed 64-bit integer	R	1010011		00010	1100001	
fcvt.lu.d	Convert to unsigned 64-bit integer	R	1010011		00011	1100001	
fcvt.d.l	Convert from signed 64-bit integer	R	1010011		00010	1101001	
fcvt.d.lu	Convert from unsigned 64-bit integer	R	1010011		00011	1101001	

RV64Q Quad-Precision Floating-Point Extension (in addition to RV32Q)

Inst	Name	FMT	Opcode	funct3/rm	rs2	funct7	Description (C)
`fcvt.l.q`	Convert to signed 64-bit integer	R	1010011		00010	1100011	
`fcvt.lu.q`	Convert to unsigned 64-bit integer	R	1010011		00011	1100011	
`fcvt.q.l`	Convert from signed 64-bit integer	R	1010011		00010	1101011	
`fcvt.q.lu`	Convert from unsigned 64-bit integer	R	1010011		00011	1101011	

RV64C Compressed Extension (in addition to RV32C)

Inst	Name	FMT	OP	funct4	funct3	[12]	[11..10]	[6..5]	Description
`c.ld`	Load Double	CL	00		011				`ld rd', (8*imm)(rs1')`
`c.sd`	Store Double	CS	00		111				`sd rs1', (8*imm)(rs2')`
`c.addiw`	ADD Immediate Word	CI	01		001				`addiw rd, rd, imm`
`c.ldsp`	Load Double from SP	CI	10		011				`ld rd, (8*imm)(sp)`
`c.sdsp`	Store Double to SP	CSS	10		111				`sd rs2, (8*imm)(sp)`

Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	—
x3	gp	Global pointer	—
x4	tp	Thread pointer	Callee
x5	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller