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

31 28	27 24	23 20	19 15	14 12	11 7	6 0	
fm	pred	succ	rs1	funct3	rd	opcode	FENCE

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-ext
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	signed
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	unsigned
addi	ADD (Immediate)	I	0010011	0x0		rd = rs1 + imm	
xori	XOR (Immediate)	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR (Immediate)	I	0010011	0x6		rd = rs1 | imm	
andi	AND (Immediate)	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[11:5]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[11:5]=0x00	rd = rs1 >> imm[0:4]	zero-ext
srai	Shift Right Arith Imm	I	0010011	0x5	imm[11:5]=0x20	rd = rs1 >> imm[0:4]	msb-ext
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	signed
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	unsigned
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-ext
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-ext
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	signed
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	signed
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	unsigned
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	unsigned
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		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	0x0	imm=0x0	Context switch to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Context switch to debugger	
fence	Memory fence	FENCE	0001111	0x0		Order mem/io accesses	

Standard Extensions

RV32M Multiply Extension

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulhsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulhu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1]; reserved=1;

sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0; }
						else { rd = 1; }; reserved=0

amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; M[rs1] = rs2
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1]; M[rs1] = rd + rs2
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1]; M[rs1] = rd & rs2
amoor.w	Atomic OR	R	0101111	0x2	0x08	rd = M[rs1]; M[rs1] = rd | rs2
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1]; M[rs1] = rd ˆ rs2
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = M[rs1]; M[rs1] = max(rd, rs2)
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = M[rs1]; M[rs1] = min(rd, rs2)
amomaxu.w	Atomic MAX (U)	R	0101111	0x2	0x1c	rd = M[rs1]; M[rs1] = max(rd, rs2)
amominu.w	Atomic MIN (U)	R	0101111	0x2	0x18	rd = M[rs1]; M[rs1] = min(rd, rs2)

RV32F / D Floating-Point Extensions

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	width	rd	opcode
offset[11:0]	base	W	dest	LOAD-FP

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode
offset[11:5]	src	base	W	offset[4:0]	STORE-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
FADD/FSUB	S	src2	src1	RM	dest	OP-FP (R)

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)	Note
flw	Flt Load Word	I				rd = M[rs1 + imm]	
fsw	Flt Store Word	S				M[rs1 + imm] = rs2	
fmadd.s	Flt Fused Mul-Add	R				rd = rs1 * rs2 + rs3	
fmsub.s	Flt Fused Mul-Sub	R				rd = rs1 * rs2 - rs3	
fnmadd.s	Flt Neg [Fused Mul-Add]	R				rd = -(rs1 * rs2) - rs3	[sic]
fnmsub.s	Flt Neg [Fused Mul-Sub]	R				rd = -(rs1 * rs2) + rs3	[sic]
fadd.s	Flt Add	R				rd = rs1 + rs2	
fsub.s	Flt Sub	R				rd = rs1 - rs2	
fmul.s	Flt Mul	R				rd = rs1 * rs2	
fdiv.s	Flt Div	R				rd = rs1 / rs2	
fsqrt.s	Flt Square Root	R				rd = sqrt(rs1)	
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)	
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)	
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)	
fmin.s	Flt Minimum	R				rd = min(rs1, rs2)	
fmax.s	Flt Maximum	R				rd = max(rs1, rs2)	
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1	
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1	
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1	
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1	
fmv.x.w	Move Float to Int	*				rd = *((int*) &rs1)	
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)	
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0	
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0	
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0	
fclass.s	Float Classify	*				rd = 0..9	

RV32C Compressed Extension

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

Inst	Name	FMT	OP	Funct	Description
`c.lwsp`	Load Word from SP	CI	`10`	`010`	`lw rd, (4*imm)(sp)`
`c.swsp`	Store Word to SP	CSS	`10`	`110`	`sw rs2, (4*imm)(sp)`
`c.lw`	Load Word	CL	`00`	`010`	`lw rd’, (4*imm)(rs1’)`
`c.sw`	Store Word	CS	`00`	`110`	`sw rs1’, (4*imm)(rs2’)`
`c.j`	Jump	CJ	`01`	`101`	`jal x0, 2*offset`
`c.jal`	Jump And Link	CJ	`01`	`001`	`jal ra, 2*offset`
`c.jr`	Jump Reg	CR	`10`	`1000`	`jalr x0, rs1, 0`
`c.jalr`	Jump And Link Reg	CR	`10`	`1001`	`jalr ra, rs1, 0`
`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.li`	Load Immediate	CI	`01`	`010`	`addi rd, x0, imm`
`c.lui`	Load Upper Imm	CI	`01`	`011`	`lui rd, imm`
`c.addi`	ADD Immediate	CI	`01`	`000`	`addi rd, rd, imm`
`c.addi16sp`	ADD Imm * 16 to SP	CI	`01`	`011`	`addi sp, sp, 16*imm`
`c.addi4spn`	ADD Imm * 4 + SP	CIW	`00`	`000`	`addi rd’, sp, 4*imm`
`c.slli`	Shift Left Logical Imm	CI	`10`	`000`	`slli rd, rd, imm`
`c.srli`	Shift Right Logical Imm	CB	`01`	`100x00`	`srli rd’, rd’, imm`
`c.srai`	Shift Right Arith Imm	CB	`01`	`100x01`	`srai rd’, rd’, imm`
`c.andi`	AND Imm	CB	`01`	`100x10`	`andi rd’, rd’, imm`
`c.mv`	MoVe	CR	`10`	`1000`	`add rd, x0, rs2`
`c.add`	ADD	CR	`10`	`1001`	`add rd, rd, rs2`
`c.and`	AND	CS	`01`	`10001111`	`and rd’, rd’, rs2’`
`c.or`	OR	CS	`01`	`10001110`	`or rd’, rd’, rs2’`
`c.xor`	XOR	CS	`01`	`10001101`	`xor rd’, rd’, rs2’`
`c.sub`	SUB	CS	`01`	`10001100`	`sub rd’, rd’, rs2’`
`c.nop`	No OPeration	CI	`01`	`000`	`addi x0, x0, 0`
`c.ebreak`	Environment BREAK	CR	`10`	`1001`	`ebreak`

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 ≠ 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 ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ 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 ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, 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

General Purpose Registers

Register	ABI Name	Description	Saver
`x0`	`zero`	Zero constant	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5-x7`	`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

Floating Point Registers

`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