RISC-V Reference

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`	`0x0`	`0x00`	`rd = rs1 + rs2`	
`sub`	SUB	R	`0110011`	`0x0`	`0x20`	`rd = rs1 - rs2`	
`xor`	XOR	R	`0110011`	`0x4`	`0x00`	`rd = rs1 ^ 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-extends
`slt`	Set Less Than	R	`0110011`	`0x2`	`0x00`	`rd = (rs1 < rs2)?1:0`	
`sltu`	Set Less Than (U)	R	`0110011`	`0x3`	`0x00`	`rd = (rs1 < rs2)?1:0`	zero-extends
`addi`	ADD Immediate	I	`0010011`	`0x0`		`rd = rs1 + imm`	
`xori`	XOR Immediate	I	`0010011`	`0x4`		`rd = rs1 ^ 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[5:11]=0x00`	`rd = rs1 << imm[0:4]`	
`srli`	Shift Right Logical Imm	I	`0010011`	`0x5`	`imm[5:11]=0x00`	`rd = rs1 >> imm[0:4]`	
`srai`	Shift Right Arith Imm	I	`0010011`	`0x5`	`imm[5:11]=0x20`	`rd = rs1 >> imm[0:4]`	msb-extends
`slti`	Set Less Than Imm	I	`0010011`	`0x2`		`rd = (rs1 < imm)?1:0`	
`sltiu`	Set Less Than Imm (U)	I	`0010011`	`0x3`		`rd = (rs1 < imm)?1:0`	zero-extends
`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-extends
`lhu`	Load Half (U)	I	`0000011`	`0x5`		`rd = M[rs1+imm][0:15]`	zero-extends
`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`	
`bge`	Branch ≥	B	`1100011`	`0x5`		`if(rs1 >= rs2) PC += imm`	
`bltu`	Branch < (U)	B	`1100011`	`0x6`		`if(rs1 <  rs2) PC += imm`	zero-extends
`bgeu`	Branch ≥ (U)	B	`1100011`	`0x7`		`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`	`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`	`Transfer control to OS`	
`ebreak`	Environment Break	I	`1110011`	`0x0`	`imm=0x1`	`Transfer control to debugger`	

Standard Extensions

RV32M Multiply Extension

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]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	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], reserve M[rs1]
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RV32F / D Floating-Point Extensions

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				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	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				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

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	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]`	Load address
	`addi rd, rd, symbol[11:0]`	
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]`	Load global
	`l{b|h|w|d} rd, symbol[11:0](rd)`	
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`	Store global
	`s{b|h|w|d} rd, symbol[11:0](rt)`	
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`	Floating-point load global
	`fl{w|d} rd, symbol[11:0](rt)`	
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`	Floating-point store global
	`fs{w|d} rd, symbol[11:0](rt)`	
`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]`	Call far-away subroutine
	`jalr x1, x1, offset[11:0]`	
`tail offset`	`auipc x6, offset[31:12]`	Tail call far-away subroutine
	`jalr x0, x6, offset[11:0]`	
`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	Callee
`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
`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

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base		+RV{64,128}	
Loads	Load Byte	I	`LB`	`rd,rs1,imm`		
	Load Halfword	I	`LH`	`rd,rs1,imm`		
	Load Word	I	`LW`	`rd,rs1,imm`	`L{D|Q}`	`rd,rs1,imm`
	Load Byte Unsigned	I	`LBU`	`rd,rs1,imm`		
	Load Half Unsigned	I	`LHU`	`rd,rs1,imm`	`L{W|D}U`	`rd,rs1,imm`
Stores	Store Byte	S	`SB`	`rs1,rs2,imm`		
	Store Halfword	S	`SH`	`rs1,rs2,imm`		
	Store Word	S	`SW`	`rs1,rs2,imm`	`S{D|Q}`	`rs1,rs2,imm`
Shifts	Shift Left	R	`SLL`	`rd,rs1,rs2`	`SLL{W|D}`	`rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI`	`rd,rs1,shamt`	`SLLI{W|D}`	`rd,rs1,shamt`
	Shift Right	R	`SRL`	`rd,rs1,rs2`	`SRL{W|D}`	`rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI`	`rd,rs1,shamt`	`SRLI{W|D}`	`rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA`	`rd,rs1,rs2`	`SRA{W|D}`	`rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI`	`rd,rs1,shamt`	`SRAI{W|D}`	`rd,rs1,shamt`
Arithmetic	ADD	R	`ADD`	`rd,rs1,rs2`	`ADD{W|D}`	`rd,rs1,rs2`
	ADD Immediate	I	`ADDI`	`rd,rs1,imm`	`ADDI{W|D}`	`rd,rs1,imm`
	SUBtract	R	`SUB`	`rd,rs1,rs2`	`SUB{W|D}`	`rd,rs1,rs2`
	Load Upper Imm	U	`LUI`	`rd,imm`		
	Add Upper Imm to PC	U	`AUIPC`	`rd,imm`		
Logical	XOR	R	`XOR`	`rd,rs1,rs2`		
	XOR Immediate	I	`XORI`	`rd,rs1,imm`		
	OR	R	`OR`	`rd,rs1,rs2`		
	OR Immediate	I	`ORI`	`rd,rs1,imm`		
	AND	R	`AND`	`rd,rs1,rs2`		
	AND Immediate	I	`ANDI`	`rd,rs1,imm`		
Compare	Set <	R	`SLT`	`rd,rs1,rs2`		
	Set < Immediate	I	`SLTI`	`rd,rs1,imm`		
	Set < Unsigned	R	`SLTU`	`rd,rs1,rs2`		
	Set < Imm Unsigned	I	`SLTIU`	`rd,rs1,imm`		
Branches	Branch =	SB	`BEQ`	`rs1,rs2,imm`		
	Branch ≠	SB	`BNE`	`rs1,rs2,imm`		
	Branch <	SB	`BLT`	`rs1,rs2,imm`		
	Branch ≥	SB	`BGE`	`rs1,rs2,imm`		
	Branch < Unsigned	SB	`BLTU`	`rs1,rs2,imm`		
	Branch ≥ Unsigned	SB	`BGEU`	`rs1,rs2,imm`		
Jump & Link	J&L	UJ	`JAL`	`rd,imm`		
	Jump & Link Register	UJ	`JALR`	`rd,rs1,imm`		
Synch	Synch thread	I	`FENCE`			
	Synch Instr & Data	I	`FENCE.I`			
System	System CALL	I	`SCALL`			
	System BREAK	I	`SBREAK`			
Counters	ReaD CYCLE	I	`RDCYCLE`	`rd`		
	ReaD CYCLE upper Half	I	`RDCYCLEH`	`rd`		
	ReaD TIME	I	`RDTIME`	`rd`		
	ReaD TIME upper Half	I	`RDTIMEH`	`rd`		
	ReaD INSTR RETired	I	`RDINSTRET`	`rd`		
	ReaD INSTR upper Half	I	`RDINSTRETH`	`rd`		

RV Privileged Instructions

Category	Name	RV mnemonic	
CSR Access	Atomic R/W	`CSRRW`	`rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS`	`rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC`	`rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI`	`rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI`	`rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI`	`rd,csr,imm`
Change Level	Env. Call	`ECALL`	
	Environment Breakpoint	`EBREAK`	
	Environment Return	`ERET`	
Trap Redirect	to Supervisor	`MRTS`	
	Redirect Trap to Hypervisor	`MRTH`	
	Hypervisor Trap to Supervisor	`HRTS`	
Interrupt	Wait for Interrupt	`WFI`	
MMU	Supervisor FENCE	`SFENCE.VM`	`rs1`

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC		RVI equivalent	
Loads	Load Word	CL	`C.LW`	`rd′,rs1′,imm`	`LW`	`rd′,rs1′,imm*4`
	Load Word SP	CI	`C.LWSP`	`rd,imm`	`LW`	`rd,sp,imm*4`
	Load Double	CL	`C.LD`	`rd′,rs1′,imm`	`LD`	`rd′,rs1′,imm*8`
	Load Double SP	CI	`C.LDSP`	`rd,imm`	`LD`	`rd,sp,imm*8`
	Load Quad	CL	`C.LQ`	`rd′,rs1′,imm`	`LQ`	`rd′,rs1′,imm*16`
	Load Quad SP	CI	`C.LQSP`	`rd,imm`	`LQ`	`rd,sp,imm*16`
Stores	Store Word	CS	`C.SW`	`rs1′,rs2′,imm`	`SW`	`rs1′,rs2′,imm*4`
	Store Word SP	CSS	`C.SWSP`	`rs2,imm`	`SW`	`rs2,sp,imm*4`
	Store Double	CS	`C.SD`	`rs1′,rs2′,imm`	`SD`	`rs1′,rs2′,imm*8`
	Store Double SP	CSS	`C.SDSP`	`rs2,imm`	`SD`	`rs2,sp,imm*8`
	Store Quad	CS	`C.SQ`	`rs1′,rs2′,imm`	`SQ`	`rs1′,rs2′,imm*16`
	Store Quad SP	CSS	`C.SQSP`	`rs2,imm`	`SQ`	`rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD`	`rd,rs1`	`ADD`	`rd,rd,rs1`
	ADD Word	CR	`C.ADDW`	`rd,rs1`	`ADDW`	`rd,rd,imm`
	ADD Immediate	CI	`C.ADDI`	`rd,imm`	`ADDI`	`rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW`	`rd,imm`	`ADDIW`	`rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP`	`x0,imm`	`ADDI`	`sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN`	`rd',imm`	`ADDI`	`rd',sp,imm*4`
	Load Immediate	CI	`C.LI`	`rd,imm`	`ADDI`	`rd,x0,imm`
	Load Upper Imm	CI	`C.LUI`	`rd,imm`	`LUI`	`rd,imm`
	MoVe	CR	`C.MV`	`rd,rs1`	`ADD`	`rd,rs1,x0`
	SUB	CR	`C.SUB`	`rd,rs1`	`SUB`	`rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI`	`rd,imm`	`SLLI`	`rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ`	`rs1′,imm`	`BEQ`	`rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ`	`rs1′,imm`	`BNE`	`rs1',x0,imm`
Jump	Jump	CJ	`C.J`	`imm`	`JAL`	`x0,imm`
	Jump Register	CR	`C.JR`	`rd,rs1`	`JALR`	`x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL`	`imm`	`JAL`	`ra,imm`
	Jump & Link Register	CR	`C.JALR`	`rs1`	`JALR`	`ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`		`EBREAK`	

32-bit Instruction Formats

	31	30 — 25	24 — 21	20	19 — 15	14 — 12	11 — 8	7	6 — 0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[11:0]				rs1	funct3	rd		opcode
S	imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

16-bit (RVC) Instruction Formats

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

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	`MUL`	`rd,rs1,rs2`	`MUL{W|D}`	`rd,rs1,rs2`
	MULtiply upper Half	R	`MULH`	`rd,rs1,rs2`		
	MULtiply Half Sign/Uns	R	`MULHSU`	`rd,rs1,rs2`		
	MULtiply upper Half Uns	R	`MULHU`	`rd,rs1,rs2`		
Divide	DIVide	R	`DIV`	`rd,rs1,rs2`	`DIV{W|D}`	`rd,rs1,rs2`
	DIVide Unsigned	R	`DIVU`	`rd,rs1,rs2`		
Remainder	REMainder	R	`REM`	`rd,rs1,rs2`	`REM{W|D}`	`rd,rs1,rs2`
	REMainder Unsigned	R	`REMU`	`rd,rs1,rs2`	`REMU{W|D}`	`rd,rs1,rs2`

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	`LR.W`	`rd,rs1`	`LR.{D|Q}`	`rd,rs1`
Store	Store Conditional	R	`SC.W`	`rd,rs1,rs2`	`SC.{D|Q}`	`rd,rs1,rs2`
Swap	SWAP	R	`AMOSWAP.W`	`rd,rs1,rs2`	`AMOSWAP.{D|Q}`	`rd,rs1,rs2`
Add	ADD	R	`AMOADD.W`	`rd,rs1,rs2`	`AMOADD.{D|Q}`	`rd,rs1,rs2`
Logical	XOR	R	`AMOXOR.W`	`rd,rs1,rs2`	`AMOXOR.{D|Q}`	`rd,rs1,rs2`
	AND	R	`AMOAND.W`	`rd,rs1,rs2`	`AMOAND.{D|Q}`	`rd,rs1,rs2`
	OR	R	`AMOOR.W`	`rd,rs1,rs2`	`AMOOR.{D|Q}`	`rd,rs1,rs2`
Min/Max	MINimum	R	`AMOMIN.W`	`rd,rs1,rs2`	`AMOMIN.{D|Q}`	`rd,rs1,rs2`
	MAXimum	R	`AMOMAX.W`	`rd,rs1,rs2`	`AMOMAX.{D|Q}`	`rd,rs1,rs2`
	MINimum Unsigned	R	`AMOMINU.W`	`rd,rs1,rs2`	`AMOMINU.{D|Q}`	`rd,rs1,rs2`
	MAXimum Unsigned	R	`AMOMAXU.W`	`rd,rs1,rs2`	`AMOMAXU.{D|Q}`	`rd,rs1,rs2`

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)		+RV{64,128}	
Move	Move from Integer	R	`FMV.{H|S}.X`	`rd,rs1`	`FMV.{D|Q}.X`	`rd,rs1`
	Move to Integer	R	`FMV.X.{H|S}`	`rd,rs1`	`FMV.X.{D|Q}`	`rd,rs1`
Convert	Convert from Int	R	`FCVT.{H|S|D|Q}.W`	`rd,rs1`	`FCVT.{H|S|D|Q}.{L|T}`	`rd,rs1`
	Convert from Int Unsigned	R	`FCVT.{H|S|D|Q}.WU`	`rd,rs1`	`FCVT.{H|S|D|Q}.{L|T}U`	`rd,rs1`
	Convert to Int	R	`FCVT.W.{H|S|D|Q}`	`rd,rs1`	`FCVT.{L|T}.{H|S|D|Q}`	`rd,rs1`
	Convert to Int Unsigned	R	`FCVT.WU.{H|S|D|Q}`	`rd,rs1`	`FCVT.{L|T}U.{H|S|D|Q}`	`rd,rs1`
Load	Load	I	`FL{W,D,Q}`	`rd,rs1,imm`		
Store	Store	S	`FS{W,D,Q}`	`rs1,rs2,imm`		
Arithmetic	ADD	R	`FADD.{S|D|Q}`	`rd,rs1,rs2`		
	SUBtract	R	`FSUB.{S|D|Q}`	`rd,rs1,rs2`		
	MULtiply	R	`FMUL.{S|D|Q}`	`rd,rs1,rs2`		
	DIVide	R	`FDIV.{S|D|Q}`	`rd,rs1,rs2`		
	SQuare RooT	R	`FSQRT.{S|D|Q}`	`rd,rs1`		
Mul-Add	Multiply-ADD	R	`FMADD.{S|D|Q}`	`rd,rs1,rs2,rs3`		
	Multiply-SUBtract	R	`FMSUB.{S|D|Q}`	`rd,rs1,rs2,rs3`		
	Negative Multiply-SUBtract	R	`FNMSUB.{S|D|Q}`	`rd,rs1,rs2,rs3`		
	Negative Multiply-ADD	R	`FNMADD.{S|D|Q}`	`rd,rs1,rs2,rs3`		
Sign Inject	SiGN source	R	`FSGNJ.{S|D|Q}`	`rd,rs1,rs2`		
	Negative SiGN source	R	`FSGNJN.{S|D|Q}`	`rd,rs1,rs2`		
	Xor SiGN source	R	`FSGNJX.{S|D|Q}`	`rd,rs1,rs2`		
Min/Max	MINimum	R	`FMIN.{S|D|Q}`	`rd,rs1,rs2`		
	MAXimum	R	`FMAX.{S|D|Q}`	`rd,rs1,rs2`		
Compare	Compare Float =	R	`FEQ.{S|D|Q}`	`rd,rs1,rs2`		
	Compare Float <	R	`FLT.{S|D|Q}`	`rd,rs1,rs2`		
	Compare Float ≤	R	`FLE.{S|D|Q}`	`rd,rs1,rs2`		
Categorization	Classify Type	R	`FCLASS.{S|D|Q}`	`rd,rs1`		
Configuration	Read Status	R	`FRCSR`	`rd`		
	Read Rounding Mode	R	`FRRM`	`rd`		
	Read Flags	R	`FRFLAGS`	`rd`		
	Swap Status Reg	R	`FSCSR`	`rd,rs1`		
	Swap Rounding Mode	R	`FSRM`	`rd,rs1`		
	Swap Flags	R	`FSFLAGS`	`rd,rs1`		
	Swap Rounding Mode Imm	I	`FSRMI`	`rd,imm`		
	Swap Flags Imm	I	`FSFLAGSI`	`rd,imm`		

RISC-V Calling Convention

Register	ABI Name	Saver	Description
`x0`	`zero`	---	Hard-wired zero
`x1`	`ra`	Caller	Return address
`x2`	`sp`	Callee	Stack pointer
`x3`	`gp`	---	Global pointer
`x4`	`tp`	---	Thread pointer
`x5-7`	`t0-2`	Caller	Temporaries
`x8`	`s0/fp`	Callee	Saved register/frame pointer
`x9`	`s1`	Callee	Saved register
`x10-11`	`a0-1`	Caller	Function arguments/return values
`x12-17`	`a2-7`	Caller	Function arguments
`x18-27`	`s2-11`	Callee	Saved registers
`x28-31`	`t3-t6`	Caller	Temporaries
`f0-7`	`ft0-7`	Caller	FP temporaries
`f8-9`	`fs0-1`	Callee	FP saved registers
`f10-11`	`fa0-1`	Caller	FP arguments/return values
`f12-17`	`fa2-7`	Caller	FP arguments
`f18-27`	`fs2-11`	Callee	FP saved registers
`f28-31`	`ft8-11`	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` *means set, so* `L{D|Q}` *is both* `LD` *and* `LQ`. *See risc.org. (8/21/15 revision)*