RISC-V Instruction Set Summary

31:27	26:25	24:20	19:15	14:12	11:7	6:0	
funct7		rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$			rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$		rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$		rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$					rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$					rd	op	**J-Type**
fs3	funct2	fs2	fs1	funct3	fd	op	**R4-Type**
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure B.1 RISC-V 32-bit instruction formats

- `imm`: signed immediate in $\mathbf{imm}_{11:0}$
- `uimm`: 5-bit unsigned immediate in $\mathbf{imm}_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $\mathbf{imm}_{31:12}$
- `Address`: memory address: **rs1** + SignExt($\mathbf{imm}_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$\mathbf{imm}_{12:1}$, 1'b0})
- `JTA`: jump target address: PC + SignExt({$\mathbf{imm}_{20:1}$, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits
- `csr`: control and status register

Table B.1 RV32I: RISC-V integer instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	[Address]$_{15:0}$ = rs2$_{15:0}$
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	[Address]$_{31:0}$ = rs2
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	rd = rs1 – rs2
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	rd = rs1 & rs2
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	`jal rd, label`	jump and link	PC = JTA, rd = PC + 4

*Encoded in instr$_{31:25}$, the upper seven bits of the immediate field

Table B.2 **RV64I: Extra integer instructions**

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	011	–	I	ld rd, imm(rs1)	load double word	rd=[Address]$_{63:0}$
0000011 (3)	110	–	I	lwu rd, imm(rs1)	load word unsigned	rd=ZeroExt([Address]$_{31:0}$)
0011011 (27)	000	–	I	addiw rd, rs1, imm	add immediate word	rd=SignExt((rs1+SignExt(imm))$_{31:0}$)
0011011 (27)	001	0000000	I	slliw rd, rs1, uimm	shift left logical immediate word	rd=SignExt((rs1$_{31:0}$ << uimm)$_{31:0}$)
0011011 (27)	101	0000000	I	srliw rd, rs1, uimm	shift right logical immediate word	rd=SignExt((rs1$_{31:0}$ >> uimm)$_{31:0}$)
0011011 (27)	101	0100000	I	sraiw rd, rs1, uimm	shift right arith. immediate word	rd=SignExt((rs1$_{31:0}$ >>> uimm)$_{31:0}$)
0100011 (35)	011	–	S	sd rs2, imm(rs1)	store double word	[Address]$_{63:0}$=rs2
0111011 (59)	000	0000000	R	addw rd, rs1, rs2	add word	rd=SignExt((rs1+rs2)$_{31:0}$)
0111011 (59)	000	0100000	R	subw rd, rs1, rs2	subtract word	rd=SignExt((rs1−rs2)$_{31:0}$)
0111011 (59)	001	0000000	R	sllw rd, rs1, rs2	shift left logical word	rd=SignExt((rs1$_{31:0}$ << rs2$_{4:0}$)$_{31:0}$)
0111011 (59)	101	0000000	R	srlw rd, rs1, rs2	shift right logical word	rd=SignExt((rs1$_{31:0}$ >> rs2$_{4:0}$)$_{31:0}$)
0111011 (59)	101	0100000	R	sraw rd, rs1, rs2	shift right arithmetic word	rd=SignExt((rs1$_{31:0}$ >>> rs2$_{4:0}$)$_{31:0}$)

In RV64I, registers are 64 bits, but instructions are still 32 bits. The term "word" generally refers to a 32-bit value. In RV64I, immediate shift instructions use 6-bit immediates: uimm$_{5:0}$; but for word shifts, the most significant bit of the shift amount (uimm$_5$) must be 0. Instructions ending in "w" (for "word") operate on the lower half of the 64-bit registers. Sign- or zero-extension produces a 64-bit result.

Table B.3 **RVF/D: RISC-V single- and double-precision floating-point instructions**

op	funct3	funct7	rs2	Type	Instruction	Description	Operation
1000011 (67)	rm	fs3, fmt	–	R4	fmadd fd,fs1,fs2,fs3	multiply-add	fd = fs1 * fs2 + fs3
1000111 (71)	rm	fs3, fmt	–	R4	fmsub fd,fs1,fs2,fs3	multiply-subtract	fd = fs1 * fs2 − fs3
1001011 (75)	rm	fs3, fmt	–	R4	fnmsub fd,fs1,fs2,fs3	negate multiply-add	fd = −(fs1 * fs2 + fs3)
1001111 (79)	rm	fs3, fmt	–	R4	fnmadd fd,fs1,fs2,fs3	negate multiply-subtract	fd = −(fs1 * fs2 - fs3)
1010011 (83)	rm	00000, fmt	–	R	fadd fd,fs1,fs2	add	fd = fs1 + fs2
1010011 (83)	rm	00001, fmt	–	R	fsub fd,fs1,fs2	subtract	fd = fs1 − fs2
1010011 (83)	rm	00010, fmt	–	R	fmul fd,fs1,fs2	multiply	fd = fs1 * fs2
1010011 (83)	rm	00011, fmt	–	R	fdiv fd,fs1,fs2	divide	fd = fs1 / fs2
1010011 (83)	rm	01011, fmt	00000	R	fsqrt fd,fs1	square root	fd = sqrt(fs1)
1010011 (83)	000	00100, fmt	–	R	fsgnj fd,fs1,fs2	sign injection	fd = fs1, sign = sign(fs2)
1010011 (83)	001	00100, fmt	–	R	fsgnjn fd,fs1,fs2	negate sign injection	fd = fs1, sign = −sign(fs2)
1010011 (83)	010	00100, fmt	–	R	fsgnjx fd,fs1,fs2	xor sign injection	fd = fs1, sign = sign(fs2) ^ sign(fs1)
1010011 (83)	000	00101, fmt	–	R	fmin fd,fs1,fs2	min	fd = min(fs1, fs2)
1010011 (83)	001	00101, fmt	–	R	fmax fd,fs1,fs2	max	fd = max(fs1, fs2)
1010011 (83)	010	10100, fmt	–	R	feq rd,fs1,fs2	compare =	rd = (fs1 == fs2)
1010011 (83)	001	10100, fmt	–	R	flt rd,fs1,fs2	compare <	rd = (fs1 < fs2)
1010011 (83)	000	10100, fmt	–	R	fle rd,fs1,fs2	compare ≤	rd = (fs1 ≤ fs2)
1010011 (83)	001	11100, fmt	00000	R	fclass rd,fs1	classify	rd = classification of fs1
RVF only							
0000111 (7)	010	–	–	I	flw fd, imm(rs1)	load float	fd = [Address]$_{31:0}$
0100111 (39)	010	–	–	S	fsw fs2,imm(rs1)	store float	[Address]$_{31:0}$ = fd
1010011 (83)	rm	1100000	00000	R	fcvt.w.s rd, fs1	convert to integer	rd = integer(fs1)
1010011 (83)	rm	1100000	00001	R	fcvt.wu.s rd, fs1	convert to unsigned integer	rd = unsigned(fs1)
1010011 (83)	rm	1101000	00000	R	fcvt.s.w fd, rs1	convert int to float	fd = float(rs1)
1010011 (83)	rm	1101000	00001	R	fcvt.s.wu fd, rs1	convert unsigned to float	fd = float(rs1)
1010011 (83)	000	1110000	00000	R	fmv.x.w rd, fs1	move to integer register	rd = fs1
1010011 (83)	000	1111000	00000	R	fmv.w.x fd, rs1	move to f.p. register	fd = rs1
RVD only							
0000111 (7)	011	-	–	I	fld fd, imm(rs1)	load double	fd = [Address]$_{63:0}$
0100111 (39)	011	–	–	S	fsd fs2,imm(rs1)	store double	[Address]$_{63:0}$ = fd
1010011 (83)	rm	1100001	00000	R	fcvt.w.d rd, fs1	convert to integer	rd = integer(fs1)
1010011 (83)	rm	1100001	00001	R	fcvt.wu.d rd, fs1	convert to unsigned integer	rd = unsigned(fs1)
1010011 (83)	rm	1101001	00000	R	fcvt.d.w fd, rs1	convert int to double	fd = double(rs1)
1010011 (83)	rm	1101001	00001	R	fcvt.d.wu fd, rs1	convert unsigned to double	fd = double(rs1)
1010011 (83)	rm	0100000	00001	R	fcvt.s.d fd, fs1	convert double to float	fd = float(fs1)
1010011 (83)	rm	0100001	00000	R	fcvt.d.s fd, fs1	convert float to double	fd = double(fs1)

fs1, fs2, fs3, fd: floating-point registers. **fs1, fs2**, and **fd** are encoded in fields **rs1, rs2**, and **rd**; only R4-type also encodes **fs3**. **fmt**: precision of computational instruction (single=00_2, double=01_2, quad=11_2). **rm**: rounding mode (0=to nearest, 1=toward zero, 2=down, 3=up, 4=to nearest (max magnitude), 7=dynamic). sign(**fs1**): the sign of **fs1**.

Table B.4 Register names and numbers

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

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		rs1'	imm	rs2'	op	CS-Type
funct6			rd'/rs1'	funct2	rs2'	op	CS'-Type
funct3	imm		rs1'	imm		op	CB-Type
funct3	imm	funct	rd'/rs1'	imm		op	CB'-Type
funct3	imm					op	CJ-Type
funct3	imm			rs2		op	CSS-Type
funct3	imm				rd'	op	CIW-Type
funct3	imm		rs1'	imm	rd'	op	CL-Type
3 bits	3 bits		3 bits	2 bits	3 bits	2 bits	

Figure B.2 RISC-V compressed (16-bit) instruction formats

Table B.5 RVM: RISC-V multiply and divide instructions

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000001	R	`mul rd, rs1, rs2`	multiply	rd = (rs1 * rs2)$_{31:0}$
0110011 (51)	001	0000001	R	`mulh rd, rs1, rs2`	multiply high signed signed	rd = (rs1 * rs2)$_{63:32}$
0110011 (51)	010	0000001	R	`mulhsu rd, rs1, rs2`	multiply high signed unsigned	rd = (rs1 * rs2)$_{63:32}$
0110011 (51)	011	0000001	R	`mulhu rd, rs1, rs2`	multiply high unsigned unsigned	rd = (rs1 * rs2)$_{63:32}$
0110011 (51)	100	0000001	R	`div rd, rs1, rs2`	divide (signed)	rd = rs1 / rs2
0110011 (51)	101	0000001	R	`divu rd, rs1, rs2`	divide unsigned	rd = rs1 / rs2
0110011 (51)	110	0000001	R	`rem rd, rs1, rs2`	remainder (signed)	rd = rs1 % rs2
0110011 (51)	111	0000001	R	`remu rd, rs1, rs2`	remainder unsigned	rd = rs1 % rs2

Table B.6 RVC: RISC-V compressed (16-bit) instructions

op	instr$_{15:10}$	funct2	Type	RVC Instruction	32-Bit Equivalent
00 (0)	000–––	–	CIW	`c.addi4spn rd', sp, imm`	`addi rd', sp, ZeroExt(imm)*4`
00 (0)	001–––	–	CL	`c.fld fd', imm(rs1')`	`fld fd', (ZeroExt(imm)*8)(rs1')`
00 (0)	010–––	–	CL	`c.lw rd', imm(rs1')`	`lw rd', (ZeroExt(imm)*4)(rs1')`
00 (0)	011–––	–	CL	`c.flw fd', imm(rs1')`	`flw fd', (ZeroExt(imm)*4)(rs1')`
00 (0)	101–––	–	CS	`c.fsd fs2', imm(rs1')`	`fsd fs2', (ZeroExt(imm)*8)(rs1')`
00 (0)	110–––	–	CS	`c.sw rs2', imm(rs1')`	`sw rs2', (ZeroExt(imm)*4)(rs1')`
00 (0)	111–––	–	CS	`c.fsw fs2', imm(rs1')`	`fsw fs2', (ZeroExt(imm)*4)(rs1')`
01 (1)	000000	–	CI	`c.nop (rs1=0,imm=0)`	`nop`
01 (1)	000–––	–	CI	`c.addi rd, imm`	`addi rd, rd, SignExt(imm)`
01 (1)	001–––	–	CJ	`c.jal label`	`jal ra, label`
01 (1)	010–––	–	CI	`c.li rd, imm`	`addi rd, x0, SignExt(imm)`
01 (1)	011–––	–	CI	`c.lui rd, imm`	lui rd, {14{imm$_5$}, imm}
01 (1)	011–––	–	CI	`c.addi16sp sp, imm`	`addi sp, sp, SignExt(imm)*16`
01 (1)	100–00	–	CB'	`c.srli rd', imm`	`srli rd', rd', imm`
01 (1)	100–01	–	CB'	`c.srai rd', imm`	`srai rd', rd', imm`
01 (1)	100–10	–	CB'	`c.andi rd', imm`	`andi rd', rd', SignExt(imm)`
01 (1)	100011	00	CS'	`c.sub rd', rs2'`	`sub rd', rd', rs2'`
01 (1)	100011	01	CS'	`c.xor rd', rs2'`	`xor rd', rd', rs2'`
01 (1)	100011	10	CS'	`c.or rd', rs2'`	`or rd', rd', rs2'`
01 (1)	100011	11	CS'	`c.and rd', rs2'`	`and rd', rd', rs2'`
01 (1)	101–––	–	CJ	`c.j label`	`jal x0, label`
01 (1)	110–––	–	CB	`c.beqz rs1', label`	`beq rs1', x0, label`
01 (1)	111–––	–	CB	`c.bnez rs1', label`	`bne rs1', x0, label`
10 (2)	000–––	–	CI	`c.slli rd, imm`	`slli rd, rd, imm`
10 (2)	001–––	–	CI	`c.fldsp fd, imm`	`fld fd, (ZeroExt(imm)*8)(sp)`
10 (2)	010–––	–	CI	`c.lwsp rd, imm`	`lw rd, (ZeroExt(imm)*4)(sp)`
10 (2)	011–––	–	CI	`c.flwsp fd, imm`	`flw fd, (ZeroExt(imm)*4)(sp)`
10 (2)	1000––	–	CR	`c.jr rs1 (rs1≠0,rs2=0)`	`jalr x0, rs1, 0`
10 (2)	1000––	–	CR	`c.mv rd, rs2 (rd ≠0,rs2≠0)`	`add rd, x0, rs2`
10 (2)	1001––	–	CR	`c.ebreak (rs1=0,rs2=0)`	`ebreak`
10 (2)	1001––	–	CR	`c.jalr rs1 (rs1≠0,rs2=0)`	`jalr ra, rs1, 0`
10 (2)	1001––	–	CR	`c.add rd, rs2 (rs1≠0,rs2≠0)`	`add rd, rd, rs2`
10 (2)	101–––	–	CSS	`c.fsdsp fs2, imm`	`fsd fs2, (ZeroExt(imm)*8)(sp)`
10 (2)	110–––	–	CSS	`c.swsp rs2, imm`	`sw rs2, (ZeroExt(imm)*4)(sp)`
10 (2)	111–––	–	CSS	`c.fswsp fs2, imm`	`fsw fs2, (ZeroExt(imm)*4)(sp)`

rs1', rs2', rd': 3-bit register designator for registers 8–15: 000$_2$ = x8 or f8, 001$_2$ = x9 or f9, etc.