

RISC-V Instruction Set Summary

| 31:25 | 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 _{20,10,11,19,12} | | | | rd | op | J-Type |
| fs3 | fmt | fs2 | fs1 | funct3 | fd | RA-Type |
| 5 bits | 2 bits | 5 bits | 5 bits | 3 bits | 5 bits | 7 bits |

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

- XLEN: 32 for RV32 / 64 for RV64
- imm: signed immediate in imm_{11,0}
- uimm: 5/6-bit unsigned immediate in imm_{5/4,0}
- upimm: 20 upper bits of a 32-bit immediate, in imm_{31,12}
- Address: memory address: rs1 + SignExt(imm_{11,0})
- [Address]: data at memory location Address
- BTA: branch target address: PC + SignExt((imm_{12:1}, 1'b0))
- JTA: jump target address: PC + SignExt((imm_{20:1}, 1'b0))
- label: text indicating instruction address
- SignExt: value sign-extended to XLEN
- ZeroExt: value zero-extended to XLEN
- csr: control and status register

Table I.1 RV32/64I: RISC-V integer instructions

| op | funct3/ imm _{14,12} | funct7/ instr _{31,25} | Type | Instruction | Description | Operation |
|---------|---------------------------------|-----------------------------------|------|----------------------|-------------------------------|---|
| 0000011 | 000 | imm _{11,5} | I | lb rd, imm(rs1) | load byte | rd = SignExt([Address] _{7:0}) |
| 0000011 | 001 | imm _{11,5} | I | lh rd, imm(rs1) | load half | rd = SignExt([Address] _{15:0}) |
| 0000011 | 010 | imm _{11,5} | I | lw rd, imm(rs1) | load word | rd = SignExt([Address] _{31:0}) |
| 0000011 | 100 | imm _{11,5} | I | lbu rd, imm(rs1) | load byte unsigned | rd = ZeroExt([Address] _{7:0}) |
| 0000011 | 101 | imm _{11,5} | I | lhu rd, imm(rs1) | load half unsigned | rd = ZeroExt([Address] _{15:0}) |
| 0010011 | 000 | imm _{11,5} | I | addi rd, rs1, imm | add immediate | rd = rs1 + SignExt(imm) |
| 0010011 | 001 | 000000* | I | slli rd, rs1, uimm | shift left logical immediate | rd = rs1 << uimm |
| 0010011 | 010 | imm _{11,5} | I | slti rd, rs1, imm | set less than immediate | rd = (rs1 < SignExt(imm)) |
| 0010011 | 011 | imm _{11,5} | I | sltiu rd, rs1, imm | set less than imm. unsigned | rd = (rs1 < SignExt(imm)) |
| 0010011 | 100 | imm _{11,5} | I | xori rd, rs1, imm | xor immediate | rd = rs1 ^ SignExt(imm) |
| 0010011 | 101 | 000000* | I | srlr rd, rs1, uimm | shift right logical immediate | rd = rs1 >> uimm |
| 0010011 | 101 | 010000* | I | srair rd, rs1, uimm | shift right arithmetic imm. | rd = rs1 >>> uimm |
| 0010011 | 110 | imm _{11,5} | I | ori rd, rs1, imm | or immediate | rd = rs1 SignExt(imm) |
| 0010011 | 111 | imm _{11,5} | I | andi rd, rs1, imm | and immediate | rd = rs1 & SignExt(imm) |
| 0010111 | imm _{14,12} | imm _{31,25} | U | auipc rd, upimm | add upper immediate to PC | rd = {upimm, 12'b0} + PC |
| 0100011 | 000 | imm _{11,5} | S | sb rs2, imm(rs1) | store byte | [Address] _{7:0} = rs2 _{7:0} |
| 0100011 | 001 | imm _{11,5} | S | sh rs2, imm(rs1) | store half | [Address] _{15:0} = rs2 _{15:0} |
| 0100011 | 010 | imm _{11,5} | S | sw rs2, imm(rs1) | store word | [Address] _{31:0} = rs2 _{31:0} |
| 0110011 | 000 | 0000000 | R | add rd, rs1, rs2 | add | rd = rs1 + rs2 |
| 0110011 | 000 | 0100000 | R | sub rd, rs1, rs2 | sub | rd = rs1 - rs2 |
| 0110011 | 001 | 0000000 | R | sll rd, rs1, rs2 | shift left logical | rd = rs1 << rs2 _{5/4:0} (RV64/32) |
| 0110011 | 010 | 0000000 | R | slt rd, rs1, rs2 | set less than | rd = (rs1 < rs2) |
| 0110011 | 011 | 0000000 | R | sltu rd, rs1, rs2 | set less than unsigned | rd = (rs1 < rs2) |
| 0110011 | 100 | 0000000 | R | xor rd, rs1, rs2 | xor | rd = rs1 ^ rs2 |
| 0110011 | 101 | 0000000 | R | srl rd, rs1, rs2 | shift right logical | rd = rs1 >> rs2 _{5/4:0} (RV64/32) |
| 0110011 | 101 | 0100000 | R | sra rd, rs1, rs2 | shift right arithmetic | rd = rs1 >>> rs2 _{5/4:0} (RV64/32) |
| 0110011 | 110 | 0000000 | R | or rd, rs1, rs2 | or | rd = rs1 rs2 |
| 0110011 | 111 | 0000000 | R | and rd, rs1, rs2 | and | rd = rs1 & rs2 |
| 0110111 | imm _{14,12} | imm _{31,25} | U | lui rd, upimm | load upper immediate | rd = {upimm, 12'b0} |
| 1100011 | 000 | imm _{12,10,5} | B | beq rs1, rs2, label | branch if = | if (rs1 == rs2) PC = BTA |
| 1100011 | 001 | imm _{12,10,5} | B | bne rs1, rs2, label | branch if ≠ | if (rs1 ≠ rs2) PC = BTA |
| 1100011 | 100 | imm _{12,10,5} | B | blt rs1, rs2, label | branch if < | if (rs1 < rs2) PC = BTA |
| 1100011 | 101 | imm _{12,10,5} | B | bge rs1, rs2, label | branch if ≥ | if (rs1 ≥ rs2) PC = BTA |
| 1100011 | 110 | imm _{12,10,5} | B | bltu rs1, rs2, label | branch if < unsigned | if (rs1 < rs2) PC = BTA |
| 1100011 | 111 | imm _{12,10,5} | B | bgeu rs1, rs2, label | branch if ≥ unsigned | if (rs1 ≥ rs2) PC = BTA |
| 1100111 | 000 | imm _{11,5} | I | jalr rd, rs1, imm | jump and link register | PC = rs1 + SignExt(imm), rd = PC + 4 |
| 1101111 | imm _{14,12} | imm _{20,10,5} | J | jal rd, label | jump and link | PC = JTA, rd = PC + 4 |

* = 0 for RV32, uimm₅ for RV64.

Table I.2 RV64I: Extra integer instructions

| op | funct3 | funct7/ imm _{11:5} | Type | Instruction | Description | Operation |
|---------|--------|--------------------------------|------|---------------------|------------------------------------|--|
| 0000011 | 011 | imm _{11:5} | I | ld rd, imm(rs1) | load double word | rd = [Address] _{63:0} |
| 0000011 | 110 | imm _{11:5} | I | lwu rd, imm(rs1) | load word unsigned | rd = ZeroExt([Address] _{31:0}) |
| 0011011 | 000 | imm _{11:5} | I | addiw rd, rs1, imm | add immediate word | rd = SignExt((rs1 + SignExt(imm)) _{31:0}) |
| 0011011 | 001 | 0000000 | I | slliw rd, rs1, uimm | shift left logical immediate word | rd = SignExt((rs1 _{31:0} << uimm) _{31:0}) |
| 0011011 | 101 | 0000000 | I | srliw rd, rs1, uimm | shift right logical immediate word | rd = SignExt((rs1 _{31:0} >> uimm) _{31:0}) |
| 0011011 | 101 | 0100000 | I | sraiw rd, rs1, uimm | shift right arith. immediate word | rd = SignExt((rs1 _{31:0} >>> uimm) _{31:0}) |
| 0100011 | 011 | imm _{11:5} | S | sd rs2, imm(rs1) | store double word | [Address] _{63:0} = rs2 |
| 0111011 | 000 | 0000000 | R | addw rd, rs1, rs2 | add word | rd = SignExt((rs1 + rs2) _{31:0}) |
| 0111011 | 000 | 0100000 | R | subw rd, rs1, rs2 | subtract word | rd = SignExt((rs1 - rs2) _{31:0}) |
| 0111011 | 001 | 0000000 | R | sllw rd, rs1, rs2 | shift left logical word | rd = SignExt((rs1 _{31:0} << rs2 _{4:0}) _{31:0}) |
| 0111011 | 101 | 0000000 | R | srlw rd, rs1, rs2 | shift right logical word | rd = SignExt((rs1 _{31:0} >> rs2 _{4:0}) _{31:0}) |
| 0111011 | 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 (uimm₅) 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 I.3 RVF/D/Q/Zfh: RISC-V single-, double-, quad-, and half-precision floating-point instructions

| op | funct3 | funct7/ instr _{31:25} | rs2/ imm _{4:0} | Type | Flags NV DZ OF UF NX | Instruction | Description | Operation |
|----------|--------|-----------------------------------|----------------------------|------|-------------------------|----------------------------|-----------------------|--|
| 1000011 | rm | fs3, ffmt | fs2 | R4 | x - x x x | fadd.* fd, fs1, fs2, fs3 | multiply-add | fd = fs1 * fs2 + fs3 |
| 1000111 | rm | fs3, ffmt | fs2 | R4 | x - x x x | fmsub.* fd, fs1, fs2, fs3 | multiply-subtract | fd = fs1 * fs2 - fs3 |
| 1001011 | rm | fs3, ffmt | fs2 | R4 | x - x x x | fnmsub.* fd, fs1, fs2, fs3 | negate multiply-sub | fd = -(fs1 * fs2 - fs3) |
| 1001111 | rm | fs3, ffmt | fs2 | R4 | x - x x x | fnmadd.* fd, fs1, fs2, fs3 | negate multiply-add | fd = -(fs1 * fs2 + fs3) |
| 1010011 | rm | 00000, ffmt | fs2 | R | x - x - x | fadd.* fd, fs1, fs2 | add | fd = fs1 + fs2 |
| 1010011 | rm | 00001, ffmt | fs2 | R | x - x - x | fsub.* fd, fs1, fs2 | subtract | fd = fs1 - fs2 |
| 1010011 | rm | 00010, ffmt | fs2 | R | x - x x x | fmul.* fd, fs1, fs2 | multiply | fd = fs1 * fs2 |
| 1010011 | rm | 00011, ffmt | fs2 | R | x x x x x | fdiv.* fd, fs1, fs2 | divide | fd = fs1 / fs2 |
| 1010011 | rm | 01011, ffmt | 00000 | I | x - - - x | fsqrt.* fd, fs1 | square root | fd = sqrt(fs1) |
| 1010011 | 000 | 00100, ffmt | fs2 | R | - - - - - | fsgnj.* fd, fs1, fs2 | sign injection | fd = fs1, sign = sign(fs2) |
| 1010011 | 001 | 00100, ffmt | fs2 | R | - - - - - | fsgnjn.* fd, fs1, fs2 | negate sign injection | fd = fs1, sign = -sign(fs2) |
| 1010011 | 010 | 00100, ffmt | fs2 | R | - - - - - | fsgnjx.* fd, fs1, fs2 | xor sign injection | fd = fs1, sign = sign(fs2) ^ sign(fs1) |
| 1010011 | 000 | 00101, ffmt | fs2 | R | x - - - - | fmin.* fd, fs1, fs2 | min | fd = min(fs1, fs2) |
| 1010011 | 001 | 00101, ffmt | fs2 | R | x - - - - | fmax.* fd, fs1, fs2 | max | fd = max(fs1, fs2) |
| 1010011 | 010 | 10100, ffmt | fs2 | R | x - - - - | feq.* rd, fs1, fs2 | compare = | rd = (fs1 == fs2) |
| 1010011 | 001 | 10100, ffmt | fs2 | R | x - - - - | flt.* rd, fs1, fs2 | compare < | rd = (fs1 < fs2) |
| 1010011 | 000 | 10100, ffmt | fs2 | R | x - - - - | fle.* rd, fs1, fs2 | compare ≤ | rd = (fs1 ≤ fs2) |
| 1010011 | 001 | 11100, ffmt | 00000 | I | - - - - - | fclass.* rd, fs1 | classify | rd = classification of fs1 |
| 0000111 | 010 | imm _{11:5} | imm _{4:0} | I | - - - - - | flw fd, imm(rs1) | load float | fd = [Address] _{31:0} |
| 0100111 | 010 | imm _{11:5} | fs2 | S | - - - - - | fsw fs2, imm(rs1) | store float | [Address] _{31:0} = fd |
| 1010011 | rm | 11000, ffmt | 000, lu | I | x - x - x | fcvt.dst.src rd, fs1 | convert fp to int | rd = dst(fs1) |
| 1010011 | rm | 11010, ffmt | 000, lu | I | - - x - x | fcvt.dst.src fd, rs1 | convert int to fp | fd = dst(rs1) |
| 1010011 | rm | 01000, dfmt | 000, sfmt | I | x - x x x | fcvt.dst.src fd, fs1 | convert fp to fp | fd = dst(fs1) |
| 1010011 | 000 | 11100, ffmt | 00000 | I | - - - - - | fmv.x.{h/w/d} rd, fs1 | move fp to int | rd = fs1 |
| 1010011 | 000 | 11110, ffmt | 00000 | I | - - - - - | fmv.{h/w/d}.x fd, rs1 | move int to fp | fd = rs1 |
| RVD only | | | | | | | | |
| 0000111 | 011 | imm _{11:5} | imm _{4:0} | I | - - - - - | fld fd, imm(rs1) | load double | fd = [Address] _{63:0} |
| 0100111 | 011 | imm _{11:5} | fs2 | S | - - - - - | fsd fs2, imm(rs1) | store double | [Address] _{63:0} = fs2 |
| RVQ only | | | | | | | | |
| 0000111 | 100 | imm _{11:5} | imm _{4:0} | I | - - - - - | flq fd, imm(rs1) | load quad | fd = [Address] _{127:0} |
| 0100111 | 100 | imm _{11:5} | fs2 | S | - - - - - | fsq fs2, imm(rs1) | store quad | [Address] _{127:0} = fs2 |
| RVH only | | | | | | | | |
| 0000111 | 001 | imm _{11:5} | imm _{4:0} | I | - - - - - | flh fd, imm(rs1) | load half | fd = [Address] _{15:0} |
| 0100111 | 001 | imm _{11:5} | fs2 | S | - - - - - | fsh fs2, imm(rs1) | store half | [Address] _{15:0} = fs2 |

* = {h/s/d/q}. 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. ffmt: precision of computational instruction (half=10₂, single=00₂, double=01₂, quad=11₂). sfmt: source format. rm: rounding mode (0=toward zero, 1=toward zero, 2=down, 3=up, 4=toward nearest (max magnitude), 7=dynamic). sign(fs1): the sign of fs1. dst or src = {h/s/d/q} for fp, {w/wu/lu} for 32/64 bit signed/unsigned ints. {w/wu} = 32-bit signed/unsigned int. {l/lu} = 64-bit (long) signed/unsigned int. For fp move instructions (fmv), {h/w/d} denotes a half-/single-/double-precision fp number held in a fp register, and x is a fp number (held in an integer register), with the same-precision as the fp source/destination. Floating-point exception flags are: NV: invalid operation, DZ: divide by zero, OF: overflow, UF: underflow, and NX: inexact.

Table I.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 |

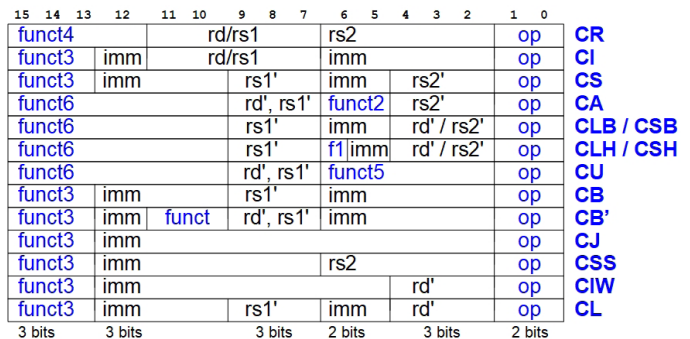


Figure I.2. RISC-V compressed (16-bit) instruction formats

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

| op | funct3 | funct7 | Type | Instruction | Description | Operation |
|---------|--------|---------|------|---------------------|---------------------------------|------------------------------------|
| 0110011 | 000 | 0000001 | R | mul rd, rs1, rs2 | multiply | $rd = (rs1 * rs2)_{XLEN-1:0}$ |
| 0110011 | 001 | 0000001 | R | mulh rd, rs1, rs2 | multiply high signed signed | $rd = (rs1 * rs2)_{2*XLEN-1:XLEN}$ |
| 0110011 | 010 | 0000001 | R | mulhsu rd, rs1, rs2 | multiply high signed unsigned | $rd = (rs1 * rs2)_{2*XLEN-1:XLEN}$ |
| 0110011 | 011 | 0000001 | R | mulhu rd, rs1, rs2 | multiply high unsigned unsigned | $rd = (rs1 * rs2)_{2*XLEN-1:XLEN}$ |
| 0110011 | 100 | 0000001 | R | div rd, rs1, rs2 | divide (signed) | $rd = rs1 / rs2$ |
| 0110011 | 101 | 0000001 | R | divu rd, rs1, rs2 | divide unsigned | $rd = rs1 / rs2$ |
| 0110011 | 110 | 0000001 | R | rem rd, rs1, rs2 | remainder (signed) | $rd = rs1 \% rs2$ |
| 0110011 | 111 | 0000001 | R | remu rd, rs1, rs2 | remainder unsigned | $rd = rs1 \% rs2$ |

RV64M adds mulw, divw, remw, divuw, remuw with op = 59. These operate on only the lower 32 bits of a register.

Table I.6 RVC/Zca: RISC-V compressed (16-bit) integer instructions

| op | instr _{15:10} | funct2 | Registers | Type | Compressed Instruction | 32-Bit Equivalent |
|--|------------------------|--------|---------------------|------|-------------------------|--------------------------------------|
| C/Zca: Compressed instructions excluding floating-point loads and stores | | | | | | |
| 00 | 000--- | - | imm ≠ 0 | CIW | c.addi4spn rd', sp, imm | addi rd', sp, ZeroExt(imm)*4 |
| 00 | 010--- | - | | CL | c.lw rd', imm(rs1') | lw rd', (ZeroExt(imm)*4)(rs1') |
| 00 | 110--- | - | | CS | c.sw rs2', imm(rs1') | sw rs2', (ZeroExt(imm)*4)(rs1') |
| 01 | 000000 | - | rs1 = 0, imm = 0 | CI | c.nop | addi x0, x0, 0 |
| 01 | 000--- | - | rd ≠ 0, imm ≠ 0 | CI | c.addi rd, imm | addi rd, rd, SignExt(imm) |
| 01 | 010--- | - | rd ≠ 0 | CI | c.li rd, imm | addi rd, x0, SignExt(imm) |
| 01 | 011--- | - | rd ≠ {0,2}, imm ≠ 0 | CI | c.lui rd, imm | lui rd, {14{imm ₆ }, imm} |
| 01 | 011--- | - | rd = 2, imm ≠ 0 | CI | c.addi16sp sp, imm | addi sp, sp, SignExt(imm)*16 |
| 01 | 100-00 | - | | CB' | c.srli rd', imm | srli rd', rd', imm |
| 01 | 100-01 | - | | CB' | c.srai rd', imm | srai rd', rd', imm |
| 01 | 100-10 | - | | CB' | c.andi rd', imm | andi rd', rd', SignExt(imm) |
| 01 | 100011 | 00 | | CA | c.sub rd', rs2' | sub rd', rd', rs2' |
| 01 | 100011 | 01 | | CA | c.xor rd', rs2' | xor rd', rd', rs2' |
| 01 | 100011 | 10 | | CA | c.or rd', rs2' | or rd', rd', rs2' |
| 01 | 100011 | 11 | | CA | c.and rd', rs2' | and rd', rd', rs2' |
| 01 | 101--- | - | | CJ | c.j label | jal x0, label |
| 01 | 110--- | - | | CB | c.beqz rs1', label | beq rs1', x0, label |
| 01 | 111--- | - | | CB | c.bnez rs1', label | bne rs1', x0, label |
| 10 | 000--- | - | rd ≠ 0 | CI | c.slli rd, imm | slli rd, rd, imm |
| 10 | 010--- | - | rd ≠ 0 | CI | c.lwsp rd, imm | lw rd, (ZeroExt(imm)*4)(sp) |
| 10 | 1000-- | - | rs1 ≠ 0, rs2 = 0 | CR | c.jr rs1 | jalr x0, rs1, 0 |
| 10 | 1000-- | - | rd ≠ 0, rs2 ≠ 0 | CR | c.mv rd, rs2 | add rd, x0, rs2 |
| 10 | 1001-- | - | rs1 = 0, rs2 = 0 | CR | c.ebreak | ebreak |
| 10 | 1001-- | - | rs1 ≠ 0, rs2 = 0 | CR | c.jalr rs1 | jalr ra, rs1, 0 |
| 10 | 1001-- | - | rd ≠ 0, rs2 ≠ 0 | CR | c.add rd, rs2 | add rd, rd, rs2 |
| 10 | 110--- | - | | CSS | c.swsp rs2, imm | sw rs2, (ZeroExt(imm)*4)(sp) |
| C/Zca (RV32 only) | | | | | | |
| 01 | 001--- | - | | CJ | c.jal label | jal ra, label |
| C/Zca (RV64 only, reuses encodings for c.jal above and c.flw, c.fsw, c.flwsp, and c.fswsp from Table I.22) | | | | | | |
| 00 | 011--- | - | | CL | c.ld rd', imm(rs1') | ld rd', (ZeroExt(imm)*8)(rs1') |
| 00 | 111--- | - | | CS | c.sd rs2', imm(rs1') | sd rs2, (ZeroExt(imm)*8)(rs1') |
| 01 | 001--- | - | rd ≠ 0 | CI | c.addiw rd, imm | addiw rd, rd, (SignExt(imm)) |
| 01 | 100111 | 00 | | CA | c.subw rd', rs2' | subw rd', rd', rs2' |
| 01 | 100111 | 01 | | CA | c.addw rd', rs2' | addw rd', rd', rs2' |
| 10 | 011--- | - | rd ≠ 0 | CI | c.ldsp rd, imm | ld rd, (ZeroExt(imm)*8)(sp) |
| 10 | 111--- | - | | CSS | c.sdsp rs2, imm | sd rs2, (ZeroExt(imm)*8)(sp) |

rs1', rs2', rd': 3-bit register designator for registers 8–15: 000₂ = x8 or f8, ..., 111₂ = x15 or f15. Table I.9 lists additional RISC-V compressed instructions.

Table I.7 Common RISC-V pseudoinstructions

| Pseudoinstruction | RISC-V Instructions | Description | Operation |
|----------------------------|---|------------------------------------|---------------------------------------|
| nop | addi x0, x0, 0 | no operation | |
| li rd, imm _{11:0} | addi rd, x0, imm _{11:0} | load 12-bit immediate | rd = SignExtend(imm _{11:0}) |
| li rd, imm _{31:0} | lui rd, imm _{31:12} [*] addi rd, rd, imm _{11:0} | load 32-bit immediate | rd = imm _{31:0} |
| mv rd, rs1 | addi rd, rs1, 0 | move (also called “register copy”) | rd = rs1 |
| not rd, rs1 | xori rd, rs1, -1 | one’s complement | rd = ~rs1 |
| neg rd, rs2 | sub rd, x0, rs2 | two’s complement | rd = -rs1 |
| seqz rd, rs1 | sltiu rd, rs1, 1 | set if = 0 | rd = (rs1 == 0) |
| snez rd, rs2 | sltu rd, x0, rs2 | set if ≠ 0 | rd = (rs1 ≠ 0) |
| sltz rd, rs1 | slt rd, rs1, x0 | set if < 0 | rd = (rs1 < 0) |
| sgtz rd, rs2 | slt rd, x0, rs2 | set if > 0 | rd = (rs1 > 0) |
| beqz rs1, label | beq rs1, x0, label | branch if = 0 | if (rs1 == 0) PC = label |
| bnez rs1, label | bne rs1, x0, label | branch if ≠ 0 | if (rs1 ≠ 0) PC = label |
| blez rs2, label | bge x0, rs2, label | branch if ≤ 0 | if (rs1 ≤ 0) PC = label |
| bgez rs1, label | bge rs1, x0, label | branch if ≥ 0 | if (rs1 ≥ 0) PC = label |
| bltz rs1, label | blt rs1, x0, label | branch if < 0 | if (rs1 < 0) PC = label |
| bgtz rs2, label | blt x0, rs2, label | branch if > 0 | if (rs1 > 0) PC = label |
| ble rs1, rs2, label | bge rs2, rs1, label | branch if ≤ | if (rs1 ≤ rs2) PC = label |
| bgt rs1, rs2, label | blt rs2, rs1, label | branch if > | if (rs1 > rs2) PC = label |
| bleu rs1, rs2, label | bgeu rs2, rs1, label | branch if ≤ (unsigned) | if (rs1 ≤ rs2) PC = label |
| bgtu rs1, rs2, label | bltu rs2, rs1, offset | branch if > (unsigned) | if (rs1 > rs2) PC = label |
| j label | jal x0, label | jump | PC = label |
| jal label | jal ra, label | jump and link | PC = label, ra = PC + 4 |
| jr rs1 | jalr x0, rs1, 0 | jump register | PC = rs1 |
| jalr rs1 | jalr ra, rs1, 0 | jump and link register | PC = rs1, ra = PC + 4 |
| ret | jalr x0, ra, 0 | return from function | PC = ra |
| call label | jal ra, label | call nearby function | PC = label, ra = PC + 4 |
| call label | auipc ra, offset _{31:12} [*] jalr ra, ra, offset _{11:0} | call far away function | PC = PC + offset, ra = PC + 4 |
| la rd, symbol | auipc rd, symbol _{31:12} [*] addi rd, rd, symbol _{11:0} | load address of global variable | rd = PC + symbol |
| l{b h w} rd, symbol | auipc rd, symbol _{31:12} [*] l{b h w} rd, symbol _{11:0} (rd) | load global variable | rd = [PC + symbol] |
| s{b h w} rs2, symbol, rs1 | auipc rs1, symbol _{31:12} [*] s{b h w} rs2, symbol _{11:0} (rs1) | store global variable | [PC + symbol] = rs2 |
| crrr rd, csr | csrrs rd, csr, x0 | read CSR | rd = csr |
| crrw csr, rs1 | csrrw x0, csr, rs1 | write CSR | csr = rs1 |
| csrs/csrc csr, rs1 | csrrs/csrrc x0, csr, rs1 | set/clear bits in CSR | csr = csr rs1 / csr = csr & ~rs1 |

^{*} If bit 11 of the immediate/offset/symbol is 1, the upper immediate is incremented by 1. offset/symbol are the 32-bit PC-relative addresses of a label/global variable.

Table I.8 Privileged / CSR / fence instructions

| op | funct3 | imm _{1:0} /funct7 | Registers | Type | Instruction | Description | Operation |
|---------|--------|----------------------------|--------------|------|--------------------|----------------------------------|-----------------------------|
| 1110011 | 000 | 000000000000 | rs1,rd=0 | I | ecall | transfer control to OS | |
| 1110011 | 000 | 000000000001 | rs1,rd=0 | I | ebreak | transfer control to debugger | |
| 1110011 | 000 | 000100000010 | rs1,rd=0 | I | sret | return from supervisor exception | PC = sepc |
| 1110011 | 000 | 001100000010 | rs1,rd=0 | I | mret | return from machine exception | PC = mepc |
| 1110011 | 000 | 000100000101 | rs1,rd=0 | I | wfi | wait for interrupt | |
| 0001111 | 000 | 000000110011 | rs1,rd=0 | I | fence | synchronize loads and stores | |
| 0001111 | 001 | 000000000000 | rs1,rd=0 | I | fence.i | synchronize instruction memory | |
| 1110011 | 000 | 0001001 | rs1,rs2,rd=0 | R | sfence.vma | synchronize page table | |
| 1110011 | 001 | csr | | I | csrrw rd,csr,rs1 | CSR read/write | rd=csr,csr = rs1 |
| 1110011 | 010 | csr | | I | csrrs rd,csr,rs1 | CSR read/set | rd=csr,csr = rs1 |
| 1110011 | 011 | csr | | I | csrrc rd,csr,rs1 | CSR read/clear | rd=csr,csr &=~rs1 |
| 1110011 | 101 | csr | rs1=uimm | I | csrrwi rd,csr,uimm | CSR read/write immediate | rd=csr,csr = ZeroExt(uimm) |
| 1110011 | 110 | csr | rs1=uimm | I | csrrsi rd,csr,uimm | CSR read/set immediate | rd=csr,csr = ZeroExt(uimm) |
| 1110011 | 111 | csr | rs1=uimm | I | csrrci rd,csr,uimm | CSR read/clear immediate | rd=csr,csr &=~ZeroExt(uimm) |