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

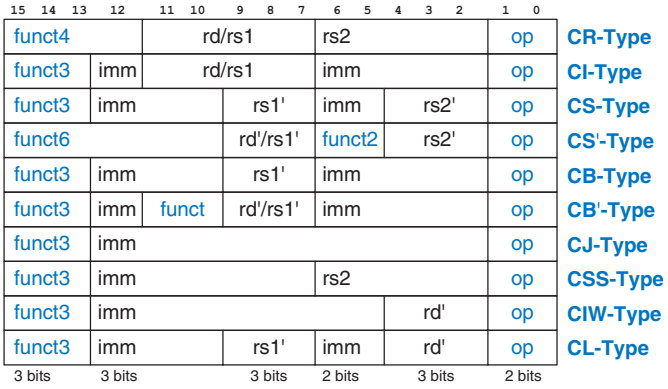

Figure B.2 RISC-V compressed (16-bit) instruction formats

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.

Table B.7 RISC-V pseudoinstructions

Pseudoinstruction	RISC-V Instructions	Description	Operation
`nop`	`addi x0, x0, 0`	no operation	
`li rd, imm11:0`	`addi rd, x0, imm11:0`	load 12-bit immediate	`rd = SignExtend(imm11:0)`
`li rd, imm31:0`	`lui rd, imm31:12`* `addi rd, rd, imm11:0`	load 32-bit immediate	`rd = imm31:0`
`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, rs1`	`sub rd, x0, rs1`	two's complement	`rd = −rs1`
`seqz rd, rs1`	`sltiu rd, rs1, 1`	set if = 0	`rd = (rs1 == 0)`
`snez rd, rs1`	`sltu rd, x0, rs1`	set if ≠ 0	`rd = (rs1 ≠ 0)`
`sltz rd, rs1`	`slt rd, rs1, x0`	set if < 0	`rd = (rs1 < 0)`
`sgtz rd, rs1`	`slt rd, x0, rs1`	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 rs1, label`	`bge x0, rs1, 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 rs1, label`	`blt x0, rs1, 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, offset31:12`* `jalr ra, ra, offset11:0`	call far away function	`PC = PC + offset, ra = PC + 4`
`la rd, symbol`	`auipc rd, symbol31:12`* `addi rd, rd, symbol11:0`	load address of global variable	`rd = PC + symbol`
`l{b|h|w} rd, symbol`	`auipc rd, symbol31:12`* `l{b|h|w} rd, symbol11:0(rd)`	load global variable	`rd = [PC + symbol]`
`s{b|h|w} rs2, symbol, rs1`	`auipc rs1, symbol31:12`* `s{b|h|w} rs2, symbol11:0(rs1)`	store global variable	`[PC + symbol] = rs2`
`csrr rd, csr`	`csrrs rd, csr, x0`	read CSR	`rd = csr`
`csrw csr, rs1`	`csrrw x0, csr, rs1`	write CSR	`csr = rs1`

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

Table B.8 Privileged / CSR instructions

op	funct3	Type	Instruction	Description	Operation
1110011 (115)	000	I	`ecall`	transfer control to OS (imm=0)	
1110011 (115)	000	I	`ebreak`	transfer control to debugger (imm=1)	
1110011 (115)	000	I	`uret`	return from user exception (rs1=0,rd=0,imm=2)	`PC = uepc`
1110011 (115)	000	I	`sret`	return from supervisor exception (rs1=0,rd=0,imm=258)	`PC = sepc`
1110011 (115)	000	I	`mret`	return from machine exception (rs1=0,rd=0,imm=770)	`PC = mepc`
1110011 (115)	001	I	`csrrw rd,csr,rs1`	CSR read/write (imm=CSR number)	`rd = csr, csr = rs1`
1110011 (115)	010	I	`csrrs rd,csr,rs1`	CSR read/set (imm=CSR number)	`rd = csr, csr = csr | rs1`
1110011 (115)	011	I	`csrrc rd,csr,rs1`	CSR read/clear (imm=CSR number)	`rd = csr, csr = csr & ~rs1`
1110011 (115)	101	I	`csrrwi rd,csr,uimm`	CSR read/write immediate (imm=CSR number)	`rd = csr, csr = ZeroExt(uimm)`
1110011 (115)	110	I	`csrrsi rd,csr,uimm`	CSR read/set immediate (imm=CSR number)	`rd = csr, csr = csr | ZeroExt(uimm)`
1110011 (115)	111	I	`csrrci rd,csr,uimm`	CSR read/clear immediate (imm=CSR number)	`rd = csr, csr = csr & ~ZeroExt(uimm)`

For privileged / CSR instructions, the 5-bit unsigned immediate, uimm, is encoded in the **rs1** field.