RISC-V Instruction Listings

(borrowed from “The RISC-V Reader: An Open Architecture Atlas”
Korean free version:
http://riscvbook.com/korean/risc-v-reader-korean-v1p0.pdf)

CS6640: Operating System Implementation

add rd, rs1, rs2 `x[rd] = x[rs1] + x[rs2]`

Add. R-type, RV32I and RV64I.

Adds register x[*rs2*] to register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

Compressed forms: **c.add** rd, rs2; **c.mv** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0110011

addi rd, rs1, immediate `x[rd] = x[rs1] + sext(immediate)`

Add Immediate. I-type, RV32I and RV64I.

Adds the sign-extended *immediate* to register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

Compressed forms: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0010011

addiw rd, rs1, immediate `x[rd] = sext((x[rs1] + sext(immediate))[31:0])`

Add Word Immediate. I-type, RV64I only.

Adds the sign-extended *immediate* to x[*rs1*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

Compressed form: **c.addiw** rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0011011

addw rd, rs1, rs2 `x[rd] = sext((x[rs1] + x[rs2])[31:0])`

Add Word. R-type, RV64I only.

Adds register x[*rs2*] to register x[*rs1*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

Compressed form: **c.addw** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0111011

amoadd.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] + x[rs2])`

Atomic Memory Operation: Add Doubleword. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to *t* + x[*rs2*]. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00000	aq	rl	rs2	rs1	011	rd	0101111

amoadd.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] + x[rs2])`

Atomic Memory Operation: Add Word. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to *t* + x[*rs2*]. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00000	aq	rl	rs2	rs1	010	rd	0101111

amoand.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] & x[rs2])`

Atomic Memory Operation: AND Doubleword. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the bitwise AND of *t* and x[*rs2*]. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01100	aq	rl	rs2	rs1	011	rd	0101111

amoand.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] & x[rs2])`

Atomic Memory Operation: AND Word. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to the bitwise AND of *t* and x[*rs2*]. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01100	aq	rl	rs2	rs1	010	rd	0101111

amomax.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MAX x[rs2])`

Atomic Memory Operation: Maximum Doubleword. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the larger of *t* and x[*rs2*], using a two's complement comparison. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10100	aq	rl	rs2	rs1	011	rd	0101111

amomax.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MAX x[rs2])`

Atomic Memory Operation: Maximum Word. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to the larger of *t* and x[*rs2*], using a two's complement comparison. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10100	aq	rl	rs2	rs1	010	rd	0101111

amomaxu.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MAXU x[rs2])`

Atomic Memory Operation: Maximum Doubleword, Unsigned. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the larger of *t* and x[*rs2*], using an unsigned comparison. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11100	aq	rl	rs2	rs1	011	rd	0101111

amomaxu.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MAXU x[rs2])`

Atomic Memory Operation: Maximum Word, Unsigned. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to the larger of *t* and x[*rs2*], using an unsigned comparison. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11100	aq	rl	rs2	rs1	010	rd	0101111

amomin.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MIN x[rs2])`

Atomic Memory Operation: Minimum Doubleword. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the smaller of *t* and x[*rs2*], using a two's complement comparison. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10000	aq	rl	rs2	rs1	011	rd	0101111

amomin.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MIN x[rs2])`

Atomic Memory Operation: Minimum Word. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to the smaller of *t* and x[*rs2*], using a two's complement comparison. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
10000	aq	rl	rs2	rs1	010	rd	0101111

amominu.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] MINU x[rs2])`

Atomic Memory Operation: Minimum Doubleword, Unsigned. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the smaller of *t* and x[*rs2*], using an unsigned comparison. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11000	aq	rl	rs2	rs1	011	rd	0101111

amominu.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] MINU x[rs2])`

Atomic Memory Operation: Minimum Word, Unsigned. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to the smaller of *t* and x[*rs2*], using an unsigned comparison. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
11000	aq	rl	rs2	rs1	010	rd	0101111

amoor.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] | x[rs2])`

Atomic Memory Operation: OR Doubleword. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the bitwise OR of *t* and x[*rs2*]. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01000	aq	rl	rs2	rs1	011	rd	0101111

amoor.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] | x[rs2])`

Atomic Memory Operation: OR Word. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to the bitwise OR of *t* and x[*rs2*]. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
01000	aq	rl	rs2	rs1	010	rd	0101111

amoswap.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] SWAP x[rs2])`

Atomic Memory Operation: Swap Doubleword. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to x[*rs2*]. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00001	aq	rl	rs2	rs1	011	rd	0101111

amoswap.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] SWAP x[rs2])`

Atomic Memory Operation: Swap Word. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to x[*rs2*]. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00001	aq	rl	rs2	rs1	010	rd	0101111

amoxor.d rd, rs2, (rs1) `x[rd] = AMO64(M[x[rs1]] ^ x[rs2])`

Atomic Memory Operation: XOR Doubleword. R-type, RV64A only.

Atomically, let *t* be the value of the memory doubleword at address x[*rs1*], then set that memory doubleword to the bitwise XOR of *t* and x[*rs2*]. Set x[*rd*] to *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00100	aq	rl	rs2	rs1	011	rd	0101111

amoxor.w rd, rs2, (rs1) `x[rd] = AMO32(M[x[rs1]] ^ x[rs2])`

Atomic Memory Operation: XOR Word. R-type, RV32A and RV64A.

Atomically, let *t* be the value of the memory word at address x[*rs1*], then set that memory word to the bitwise XOR of *t* and x[*rs2*]. Set x[*rd*] to the sign extension of *t*.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00100	aq	rl	rs2	rs1	010	rd	0101111

and rd, rs1, rs2

`x[rd] = x[rs1] & x[rs2]`

AND. R-type, RV32I and RV64I.

Computes the bitwise AND of registers x[*rs1*] and x[*rs2*] and writes the result to x[*rd*].

Compressed form: **c.and** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	111	rd	0110011

andi rd, rs1, immediate

`x[rd] = x[rs1] & sext(immediate)`

AND Immediate. I-type, RV32I and RV64I.

Computes the bitwise AND of the sign-extended *immediate* and register x[*rs1*] and writes the result to x[*rd*].

Compressed form: **c.andi** rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	111	rd	0010011

auipc rd, immediate

`x[rd] = pc + sext(immediate[31:12] << 12)`

Add Upper Immediate to PC. U-type, RV32I and RV64I.

Adds the sign-extended 20-bit *immediate*, left-shifted by 12 bits, to the *pc*, and writes the result to x[*rd*].

31 12	11 7	6 0
immediate[31:12]	rd	0010111

beq rs1, rs2, offset

`if (rs1 == rs2) pc += sext(offset)`

Branch if Equal. B-type, RV32I and RV64I.

If register x[*rs1*] equals register x[*rs2*], set the *pc* to the current *pc* plus the sign-extended *offset*.

Compressed form: **c.beqz** rs1, offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	000	offset[4:1|11]	1100011

beqz rs1, offset

if (rs1 == 0) pc += sext(offset)

Branch if Equal to Zero. Pseudoinstruction, RV32I and RV64I.

Expands to **beq** rs1, x0, offset.

---

bge rs1, rs2, offset

if (rs1 $\geq_s$ rs2) pc += sext(offset)

Branch if Greater Than or Equal. B-type, RV32I and RV64I.

If register x[*rs1*] is at least x[*rs2*], treating the values as two's complement numbers, set the *pc* to the current *pc* plus the sign-extended *offset*.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	101	offset[4:1|11]	1100011

---

bgeu rs1, rs2, offset

if (rs1 $\geq_u$ rs2) pc += sext(offset)

Branch if Greater Than or Equal, Unsigned. B-type, RV32I and RV64I.

If register x[*rs1*] is at least x[*rs2*], treating the values as unsigned numbers, set the *pc* to the current *pc* plus the sign-extended *offset*.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	111	offset[4:1|11]	1100011

---

bgez rs1, offset

if (rs1 $\geq_s$ 0) pc += sext(offset)

Branch if Greater Than or Equal to Zero. Pseudoinstruction, RV32I and RV64I.

Expands to **bge** rs1, x0, offset.

---

bgt rs1, rs2, offset

if (rs1 $>_s$ rs2) pc += sext(offset)

Branch if Greater Than. Pseudoinstruction, RV32I and RV64I.

Expands to **blt** rs2, rs1, offset.

---

bgtu rs1, rs2, offset

if (rs1 $>_u$ rs2) pc += sext(offset)

Branch if Greater Than, Unsigned. Pseudoinstruction, RV32I and RV64I.

Expands to **bltu** rs2, rs1, offset.

---

bgtz rs2, offset

if (rs2 $>_s$ 0) pc += sext(offset)

Branch if Greater Than Zero. Pseudoinstruction, RV32I and RV64I.

Expands to **blt** x0, rs2, offset.

---

ble rs1, rs2, offset

if (rs1 $\leq_s$ rs2) pc += sext(offset)

Branch if Less Than or Equal. Pseudoinstruction, RV32I and RV64I.

Expands to **bge** rs2, rs1, offset.

---

bleu rs1, rs2, offset

if (rs1 $\leq_u$ rs2) pc += sext(offset)

Branch if Less Than or Equal, Unsigned. Pseudoinstruction, RV32I and RV64I.

Expands to **bgeu** rs2, rs1, offset.

---

blez rs2, offset

if (rs2 $\leq_s$ 0) pc += sext(offset)

Branch if Less Than or Equal to Zero. Pseudoinstruction, RV32I and RV64I.

Expands to **bge** x0, rs2, offset.

---

blt rs1, rs2, offset

if (rs1 $<_s$ rs2) pc += sext(offset)

Branch if Less Than. B-type, RV32I and RV64I.

If register x[*rs1*] is less than x[*rs2*], treating the values as two's complement numbers, set the *pc* to the current *pc* plus the sign-extended *offset*.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	100	offset[4:1|11]	1100011

---

bltz rs1, offset

if (rs1 $<_s$ 0) pc += sext(offset)

Branch if Less Than Zero. Pseudoinstruction, RV32I and RV64I.

Expands to **blt** rs1, x0, offset.

---

bltu rs1, rs2, offset

if (rs1 $<_u$ rs2) pc += sext(offset)

Branch if Less Than, Unsigned. B-type, RV32I and RV64I.

If register x[*rs1*] is less than x[*rs2*], treating the values as unsigned numbers, set the *pc* to the current *pc* plus the sign-extended *offset*.

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	110	offset[4:1|11]	1100011

bne rs1, rs2, offset

if (rs1 $\neq$ rs2) pc += sext(offset)

Branch if Not Equal. B-type, RV32I and RV64I.

If register x[*rs1*] does not equal register x[*rs2*], set the *pc* to the current *pc* plus the sign-extended *offset*.

Compressed form: **c.bnez** rs1, offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	001	offset[4:1|11]	1100011

bnez rs1, offset

if (rs1 $\neq$ 0) pc += sext(offset)

Branch if Not Equal to Zero. Pseudoinstruction, RV32I and RV64I.

Expands to **bne** rs1, x0, offset.

c.add rd, rs2

x[rd] = x[rd] + x[rs2]

Add. RV32IC and RV64IC.

Expands to **add** rd, rd, rs2. Invalid when rd=x0 or rs2=x0.

15 13	12	11 7	6 2	1 0
100	1	rd	rs2	10

c.addi rd, imm

`x[rd] = x[rd] + sext(imm)`

Add Immediate. RV32IC and RV64IC.

Expands to **addi** rd, rd, imm.

15 13	12	11 7	6 2	1 0
000	imm[5]	rd	imm[4:0]	01

c.addi16sp imm

`x[2] = x[2] + sext(imm)`

Add Immediate, Scaled by 16, to Stack Pointer. RV32IC and RV64IC.

Expands to **addi** x2, x2, imm. Invalid when imm=0.

15 13	12	11 7	6 2	1 0
011	imm[9]	00010	imm[4|6|8:7|5]	01

c.addi4spn rd′, uimm

`x[8+rd′] = x[2] + uimm`

Add Immediate, Scaled by 4, to Stack Pointer, Nondestructive. RV32IC and RV64IC.

Expands to **addi** rd, x2, uimm, where rd=8+rd′. Invalid when uimm=0.

15 13	12 5	4 2	1 0
000	uimm[5:4|9:6|2|3]	rd′	00

c.addiw rd, imm

`x[rd] = sext((x[rd] + sext(imm))[31:0])`

Add Word Immediate. RV64IC only.

Expands to **addiw** rd, rd, imm. Invalid when rd=x0.

15 13	12	11 7	6 2	1 0
001	imm[5]	rd	imm[4:0]	01

c.and rd′, rs2′

`x[8+rd′] = x[8+rd′] & x[8+rs2′]`

AND. RV32IC and RV64IC.

Expands to **and** rd, rd, rs2, where rd=8+rd′ and rs2=8+rs2′.

15 10	9 7	6 5	4 2	1 0
100011	rd′	11	rs2′	01

c.addw rd′, rs2′

`x[8+rd′] = sext((x[8+rd′] + x[8+rs2′])[31:0])`

Add Word. RV64IC only.

Expands to **addw** rd, rd, rs2, where rd=8+rd′ and rs2=8+rs2′.

15 10	9 7	6 5	4 2	1 0
100111	rd′	01	rs2′	01

c.andi rd′, imm

`x[8+rd′] = x[8+rd′] & sext(imm)`

AND Immediate. RV32IC and RV64IC.

Expands to **andi** rd, rd, imm, where rd=8+rd′.

15 13	12	11 10	9 7	6 2	1 0
100	imm[5]	10	rd′	imm[4:0]	01

c.beqz rs1′, offset

`if (x[8+rs1′] == 0) pc += sext(offset)`

Branch if Equal to Zero. RV32IC and RV64IC.

Expands to **beq** rs1, x0, offset, where rs1=8+rs1′.

15 13	12 10	9 7	6 2	1 0
110	offset[8|4:3]	rs1′	offset[7:6|2:1|5]	01

c.bnez rs1′, offset `if (x[8+rs1′] ≠ 0) pc += sext(offset)`

Branch if Not Equal to Zero. RV32IC and RV64IC.

Expands to **bne** rs1, x0, offset, where rs1=8+rs1′.

15 13	12 10	9 7	6 2	1 0
111	offset[8|4:3]	rs1′	offset[7:6|2:1|5]	01

---

c.ebreak `RaiseException(Breakpoint)`

Environment Breakpoint. RV32IC and RV64IC.

Expands to **ebreak**.

15 13	12	11 7	6 2	1 0
100	1	00000	00000	10

---

c.fld rd′, uimm(rs1′) `f[8+rd′] = M[x[8+rs1′] + uimm][63:0]`

Floating-point Load Doubleword. RV32DC and RV64DC.

Expands to **fld** rd, uimm(rs1), where rd=8+rd′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00

---

c.fldsp rd, uimm(x2) `f[rd] = M[x[2] + uimm][63:0]`

Floating-point Load Doubleword, Stack-Pointer Relative. RV32DC and RV64DC.

Expands to **fld** rd, uimm(x2).

15 13	12	11 7	6 2	1 0
001	uimm[5]	rd	uimm[4:3|8:6]	10

---

c.flw rd′, uimm(rs1′)

`f[8+rd′] = M[x[8+rs1′] + uimm][31:0]`

Floating-point Load Word. RV32FC only.

Expands to **flw** rd, uimm(rs1), where rd=8+rd′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00

c.flwsp rd, uimm(x2)

`f[rd] = M[x[2] + uimm][31:0]`

Floating-point Load Word, Stack-Pointer Relative. RV32FC only.

Expands to **flw** rd, uimm(x2).

15 13	12	11 7	6 2	1 0
011	uimm[5]	rd	uimm[4:2|7:6]	10

c.fsd rs2′, uimm(rs1′)

`M[x[8+rs1′] + uimm][63:0] = f[8+rs2′]`

Floating-point Store Doubleword. RV32DC and RV64DC.

Expands to **fsd** rs2, uimm(rs1), where rs2=8+rs2′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00

c.fsdsp rs2, uimm(x2)

`M[x[2] + uimm][63:0] = f[rs2]`

Floating-point Store Doubleword, Stack-Pointer Relative. RV32DC and RV64DC.

Expands to **fsd** rs2, uimm(x2).

15 13	12 7	6 2	1 0
101	uimm[5:3|8:6]	rs2	10

c.fsw rs2′, uimm(rs1′)

`M[x[8+rs1′] + uimm][31:0] = f[8+rs2′]`

Floating-point Store Word. RV32FC only.

Expands to **fsw** rs2, uimm(rs1), where rs2=8+rs2′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00

c.fswsp rs2, uimm(x2)

`M[x[2] + uimm][31:0] = f[rs2]`

Floating-point Store Word, Stack-Pointer Relative. RV32FC only.

Expands to **fsw** rs2, uimm(x2).

15 13	12 7	6 2	1 0
111	uimm[5:2|7:6]	rs2	10

c.j offset

`pc += sext(offset)`

Jump. RV32IC and RV64IC.

Expands to **jal** x0, offset.

15 13	12 2	1 0
101	offset[11|4|9:8|10|6|7|3:1|5]	01

c.jal offset

`x[1] = pc+2; pc += sext(offset)`

Jump and Link. RV32IC only.

Expands to **jal** x1, offset.

15 13	12 2	1 0
001	offset[11|4|9:8|10|6|7|3:1|5]	01

c.jalr rs1

`t = pc+2; pc = x[rs1]; x[1] = t`

Jump and Link Register. RV32IC and RV64IC.

Expands to **jalr** x1, 0(rs1). Invalid when rs1=x0.

15 13	12	11 7	6 2	1 0
100	1	rs1	00000	10

c.jr rs1

`pc = x[rs1]`

Jump Register. RV32IC and RV64IC.

Expands to **jalr** x0, 0(rs1). Invalid when rs1=x0.

15 13	12	11 7	6 2	1 0
100	0	rs1	00000	10

c.ld rd′, uimm(rs1′)

`x[8+rd′] = M[x[8+rs1′] + uimm][63:0]`

Load Doubleword. RV64IC only.

Expands to **ld** rd, uimm(rs1), where rd=8+rd′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00

c.ldsp rd, uimm(x2)

`x[rd] = M[x[2] + uimm][63:0]`

Load Doubleword, Stack-Pointer Relative. RV64IC only.

Expands to **ld** rd, uimm(x2). Invalid when rd=x0.

15 13	12	11 7	6 2	1 0
011	uimm[5]	rd	uimm[4:3|8:6]	10

c.li rd, imm `x[rd] = sext(imm)`

Load Immediate. RV32IC and RV64IC.

Expands to **addi** rd, x0, imm.

15 13	12	11 7	6 2	1 0
010	imm[5]	rd	imm[4:0]	01

c.lui rd, imm `x[rd] = sext(imm[17:12] << 12)`

Load Upper Immediate. RV32IC and RV64IC.

Expands to **lui** rd, imm. Invalid when rd=x2 or imm=0.

15 13	12	11 7	6 2	1 0
011	imm[17]	rd	imm[16:12]	01

c.lw rd′, uimm(rs1′) `x[8+rd′] = sext(M[x[8+rs1′] + uimm][31:0])`

Load Word. RV32IC and RV64IC.

Expands to **lw** rd, uimm(rs1), where rd=8+rd′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00

c.lwsp rd, uimm(x2) `x[rd] = sext(M[x[2] + uimm][31:0])`

Load Word, Stack-Pointer Relative. RV32IC and RV64IC.

Expands to **lw** rd, uimm(x2). Invalid when rd=x0.

15 13	12	11 7	6 2	1 0
010	uimm[5]	rd	uimm[4:2|7:6]	10

c.mv rd, rs2 `x[rd] = x[rs2]`

Move. RV32IC and RV64IC.

Expands to **add** rd, x0, rs2. Invalid when rs2=x0.

15 13	12	11 7	6 2	1 0
100	0	rd	rs2	10

c.or rd′, rs2′

`x[8+rd′] = x[8+rd′] | x[8+rs2′]`

OR. RV32IC and RV64IC.

Expands to **or** rd, rd, rs2, where rd=8+rd′ and rs2=8+rs2′.

15 10	9 7	6 5	4 2	1 0
100011	rd′	10	rs2′	01

c.sd rs2′, uimm(rs1′)

`M[x[8+rs1′] + uimm][63:0] = x[8+rs2′]`

Store Doubleword. RV64IC only.

Expands to **sd** rs2, uimm(rs1), where rs2=8+rs2′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
111	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00

c.sdsp rs2, uimm(x2)

`M[x[2] + uimm][63:0] = x[rs2]`

Store Doubleword, Stack-Pointer Relative. RV64IC only.

Expands to **sd** rs2, uimm(x2).

15 13	12 7	6 2	1 0
111	uimm[5:3|8:6]	rs2	10

c.slli rd, uimm

`x[rd] = x[rd] << uimm`

Shift Left Logical Immediate. RV32IC and RV64IC.

Expands to **slli** rd, rd, uimm.

15 13	12	11 7	6 2	1 0
000	uimm[5]	rd	uimm[4:0]	10

c.srai rd′, uimm

`x[8+rd′] = x[8+rd′]` $>>_s$ `uimm`

Shift Right Arithmetic Immediate. RV32IC and RV64IC.

Expands to **srai** rd, rd, uimm, where rd=8+rd′.

15 13	12	11 10	9 7	6 2	1 0
100	uimm[5]	01	rd′	uimm[4:0]	01

c.srli rd′, uimm

`x[8+rd′] = x[8+rd′] >>`$_u$` uimm`

Shift Right Logical Immediate. RV32IC and RV64IC.

Expands to **srli** rd, rd, uimm, where rd=8+rd′.

15 13	12	11 10	9 7	6 2	1 0
100	uimm[5]	00	rd′	uimm[4:0]	01

c.sub rd′, rs2′

`x[8+rd′] = x[8+rd′] - x[8+rs2′]`

Subtract. RV32IC and RV64IC.

Expands to **sub** rd, rd, rs2, where rd=8+rd′ and rs2=8+rs2′.

15 10	9 7	6 5	4 2	1 0
100011	rd′	00	rs2′	01

c.subw rd′, rs2′

`x[8+rd′] = sext((x[8+rd′] - x[8+rs2′])[31:0])`

Subtract Word. RV64IC only.

Expands to **subw** rd, rd, rs2, where rd=8+rd′ and rs2=8+rs2′.

15 10	9 7	6 5	4 2	1 0
100111	rd′	00	rs2′	01

c.sw rs2′, uimm(rs1′)

`M[x[8+rs1′] + uimm][31:0] = x[8+rs2′]`

Store Word. RV32IC and RV64IC.

Expands to **sw** rs2, uimm(rs1), where rs2=8+rs2′ and rs1=8+rs1′.

15 13	12 10	9 7	6 5	4 2	1 0
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00

c.swsp rs2, uimm(x2)

`M[x[2] + uimm][31:0] = x[rs2]`

Store Word, Stack-Pointer Relative. RV32IC and RV64IC.

Expands to **sw** rs2, uimm(x2).

15 13	12 7	6 2	1 0
110	uimm[5:2|7:6]	rs2	10

c.xor rd′, rs2′

`x[8+rd′] = x[8+rd′] ^ x[8+rs2′]`

Exclusive-OR. RV32IC and RV64IC.

Expands to **xor** rd, rd, rs2, where rd=8+rd′ and rs2=8+rs2′.

15 10	9 7	6 5	4 2	1 0
100011	rd′	01	rs2′	01

call rd, symbol

`x[rd] = pc+8; pc = &symbol`

Call. Pseudoinstruction, RV32I and RV64I.

Writes the address of the next instruction (*pc*+8) to x[*rd*], then sets the *pc* to *symbol*. Expands to **auipc** rd, offsetHi then **jalr** rd, offsetLo(rd). If *rd* is omitted, x1 is implied.

csrr rd, csr

`x[rd] = CSRs[csr]`

Control and Status Register Read. Pseudoinstruction, RV32I and RV64I.

Copies control and status register *csr* to x[*rd*]. Expands to **csrrs** rd, csr, x0.

csrc csr, rs1

`CSRs[csr] &= ∼x[rs1]`

Control and Status Register Clear. Pseudoinstruction, RV32I and RV64I.

For each bit set in x[*rs1*], clear the corresponding bit in control and status register *csr*. Expands to **csrrc** x0, csr, rs1.

csrci csr, zimm[4:0]

`CSRs[csr] &= ∼zimm`

Control and Status Register Clear Immediate. Pseudoinstruction, RV32I and RV64I.

For each bit set in the five-bit zero-extended immediate, clear the corresponding bit in control and status register *csr*. Expands to **csrrci** x0, csr, zimm.

csrrc rd, csr, rs1 `t = CSRs[csr]; CSRs[csr] = t&∼x[rs1]; x[rd] = t`

Control and Status Register Read and Clear. I-type, RV32I and RV64I.

Let *t* be the value of control and status register *csr*. Write the bitwise AND of *t* and the ones' complement of x[*rs1*] to the *csr*, then write *t* to x[*rd*].

31 20	19 15	14 12	11 7	6 0
csr	rs1	011	rd	1110011

csrrci rd, csr, zimm[4:0] `t = CSRs[csr]; CSRs[csr] = t&∼zimm; x[rd] = t`

Control and Status Register Read and Clear Immediate. I-type, RV32I and RV64I.

Let *t* be the value of control and status register *csr*. Write the bitwise AND of *t* and the ones' complement of the five-bit zero-extended immediate *zimm* to the *csr*, then write *t* to x[*rd*]. (Bits 5 and above in the *csr* are not modified.)

31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	111	rd	1110011

csrrs rd, csr, rs1 `t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t`

Control and Status Register Read and Set. I-type, RV32I and RV64I.

Let *t* be the value of control and status register *csr*. Write the bitwise OR of *t* and x[*rs1*] to the *csr*, then write *t* to x[*rd*].

31 20	19 15	14 12	11 7	6 0
csr	rs1	010	rd	1110011

csrrsi rd, csr, zimm[4:0] `t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t`

Control and Status Register Read and Set Immediate. I-type, RV32I and RV64I.

Let *t* be the value of control and status register *csr*. Write the bitwise OR of *t* and the five-bit zero-extended immediate *zimm* to the *csr*, then write *t* to x[*rd*]. (Bits 5 and above in the *csr* are not modified.)

31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	110	rd	1110011

csrrw rd, csr, rs1 `t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t`

Control and Status Register Read and Write. I-type, RV32I and RV64I.

Let *t* be the value of control and status register *csr*. Copy x[*rs1*] to the *csr*, then write *t* to x[*rd*].

31 20	19 15	14 12	11 7	6 0
csr	rs1	001	rd	1110011

csrrwi rd, csr, zimm[4:0] `x[rd] = CSRs[csr]; CSRs[csr] = zimm`

Control and Status Register Read and Write Immediate. I-type, RV32I and RV64I.

Copies the control and status register *csr* to x[*rd*], then writes the five-bit zero-extended immediate *zimm* to the *csr*.

31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	101	rd	1110011

csrs csr, rs1 `CSRs[csr] |= x[rs1]`

Control and Status Register Set. Pseudoinstruction, RV32I and RV64I.

For each bit set in x[*rs1*], set the corresponding bit in control and status register *csr*. Expands to **csrrs** x0, csr, rs1.

csrsi csr, zimm[4:0] `CSRs[csr] |= zimm`

Control and Status Register Set Immediate. Pseudoinstruction, RV32I and RV64I.

For each bit set in the five-bit zero-extended immediate, set the corresponding bit in control and status register *csr*. Expands to **csrrsi** x0, csr, zimm.

---

csrw csr, rs1 `CSRs[csr] = x[rs1]`

Control and Status Register Write. Pseudoinstruction, RV32I and RV64I.

Copies x[*rs1*] to control and status register *csr*. Expands to **csrrw** x0, csr, rs1.

---

csrwi csr, zimm[4:0] `CSRs[csr] = zimm`

Control and Status Register Write Immediate. Pseudoinstruction, RV32I and RV64I.

Copies the five-bit zero-extended immediate to control and status register *csr*. Expands to **csrrwi** x0, csr, zimm.

---

div rd, rs1, rs2 x[rd] = x[rs1] $\div_s$ x[rs2]

Divide. R-type, RV32M and RV64M.

Divides x[*rs1*] by x[*rs2*], rounding towards zero, treating the values as two's complement numbers, and writes the quotient to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	100	rd	0110011

---

divu rd, rs1, rs2 x[rd] = x[rs1] $\div_u$ x[rs2]

Divide, Unsigned. R-type, RV32M and RV64M.

Divides x[*rs1*] by x[*rs2*], rounding towards zero, treating the values as unsigned numbers, and writes the quotient to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0110011

---

divuw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\div_u$ x[rs2][31:0])

Divide Word, Unsigned. R-type, RV64M only.

Divides the lower 32 bits of x[*rs1*] by the lower 32 bits of x[*rs2*], rounding towards zero, treating the values as unsigned numbers, and writes the sign-extended 32-bit quotient to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0111011

divw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\div_s$ x[rs2][31:0])

Divide Word. R-type, RV64M only.

Divides the lower 32 bits of x[*rs1*] by the lower 32 bits of x[*rs2*], rounding towards zero, treating the values as two's complement numbers, and writes the sign-extended 32-bit quotient to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	100	rd	0111011

ebreak

RaiseException(Breakpoint)

Environment Breakpoint. I-type, RV32I and RV64I.

Makes a request of the debugger by raising a Breakpoint exception.

31 20	19 15	14 12	11 7	6 0
000000000001	00000	000	00000	1110011

ecall

RaiseException(EnvironmentCall)

Environment Call. I-type, RV32I and RV64I.

Makes a request of the execution environment by raising an Environment Call exception.

31 20	19 15	14 12	11 7	6 0
000000000000	00000	000	00000	1110011

fabs.d rd, rs1 `f[rd] = |f[rs1]|`

Floating-point Absolute Value. Pseudoinstruction, RV32D and RV64D.

Writes the absolute value of the double-precision floating-point number in f[*rs1*] to f[*rd*]. Expands to **fsgnjx.d** rd, rs1, rs1.

---

fabs.s rd, rs1 `f[rd] = |f[rs1]|`

Floating-point Absolute Value. Pseudoinstruction, RV32F and RV64F.

Writes the absolute value of the single-precision floating-point number in f[*rs1*] to f[*rd*]. Expands to **fsgnjx.s** rd, rs1, rs1.

---

fadd.d rd, rs1, rs2 `f[rd] = f[rs1] + f[rs2]`

Floating-point Add, Double-Precision. R-type, RV32D and RV64D.

Adds the double-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] and writes the rounded double-precision sum to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	rm	rd	1010011

---

fadd.s rd, rs1, rs2 `f[rd] = f[rs1] + f[rs2]`

Floating-point Add, Single-Precision. R-type, RV32F and RV64F.

Adds the single-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] and writes the rounded single-precision sum to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	rm	rd	1010011

---

fclass.d rd, rs1 `x[rd] = classify`$_d$`(f[rs1])`

Floating-point Classify, Double-Precision. R-type, RV32D and RV64D.

Writes to x[*rd*] a mask indicating the class of the double-precision floating-point number in f[*rs1*]. See the description of **fclass.s** for the interpretation of the value written to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1110001	00000	rs1	001	rd	1010011

fclass.s rd, rs1 `x[rd] = classify`$_s$`(f[rs1])`

Floating-point Classify, Single-Precision. R-type, RV32F and RV64F.

Writes to x[*rd*] a mask indicating the class of the single-precision floating-point number in f[*rs1*]. Exactly one bit in x[*rd*] is set, per the following table:

x[*rd*] bit	Meaning
0	f[*rs1*] is $-\infty$.
1	f[*rs1*] is a negative normal number.
2	f[*rs1*] is a negative subnormal number.
3	f[*rs1*] is -0.
4	f[*rs1*] is $+0$.
5	f[*rs1*] is a positive subnormal number.
6	f[*rs1*] is a positive normal number.
7	f[*rs1*] is $+\infty$.
8	f[*rs1*] is a signaling NaN.
9	f[*rs1*] is a quiet NaN.

31 25	24 20	19 15	14 12	11 7	6 0
1110000	00000	rs1	001	rd	1010011

fcvt.d.l rd, rs1

$\texttt{f[rd] = f64}_{s64}\texttt{(x[rs1])}$

Floating-point Convert to Double from Long. R-type, RV64D only.

Converts the 64-bit two's complement integer in x[*rs1*] to a double-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00010	rs1	rm	rd	1010011

fcvt.d.lu rd, rs1

$\texttt{f[rd] = f64}_{u64}\texttt{(x[rs1])}$

Floating-point Convert to Double from Unsigned Long. R-type, RV64D only.

Converts the 64-bit unsigned integer in x[*rs1*] to a double-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00011	rs1	rm	rd	1010011

fcvt.d.s rd, rs1

$\texttt{f[rd] = f64}_{f32}\texttt{(f[rs1])}$

Floating-point Convert to Double from Single. R-type, RV32D and RV64D.

Converts the single-precision floating-point number in f[*rs1*] to a double-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0100001	00000	rs1	rm	rd	1010011

fcvt.d.w rd, rs1

$\texttt{f[rd] = f64}_{s32}\texttt{(x[rs1])}$

Floating-point Convert to Double from Word. R-type, RV32D and RV64D.

Converts the 32-bit two's complement integer in x[*rs1*] to a double-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00000	rs1	rm	rd	1010011

fcvt.d.wu rd, rs1

`f[rd] = f64`$_{u32}$`(x[rs1])`

Floating-point Convert to Double from Unsigned Word. R-type, RV32D and RV64D.

Converts the 32-bit unsigned integer in x[*rs1*] to a double-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101001	00001	rs1	rm	rd	1010011

fcvt.l.d rd, rs1

`x[rd] = s64`$_{f64}$`(f[rs1])`

Floating-point Convert to Long from Double. R-type, RV64D only.

Converts the double-precision floating-point number in register f[*rs1*] to a 64-bit two's complement integer and writes it to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00010	rs1	rm	rd	1010011

fcvt.l.s rd, rs1

`x[rd] = s64`$_{f32}$`(f[rs1])`

Floating-point Convert to Long from Single. R-type, RV64F only.

Converts the single-precision floating-point number in register f[*rs1*] to a 64-bit two's complement integer and writes it to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00010	rs1	rm	rd	1010011

fcvt.lu.d rd, rs1

`x[rd] = u64`$_{f64}$`(f[rs1])`

Floating-point Convert to Unsigned Long from Double. R-type, RV64D only.

Converts the double-precision floating-point number in register f[*rs1*] to a 64-bit unsigned integer and writes it to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00011	rs1	rm	rd	1010011

fcvt.lu.s rd, rs1

$\texttt{x[rd] = u64}_{f32}\texttt{(f[rs1])}$

Floating-point Convert to Unsigned Long from Single. R-type, RV64F only.

Converts the single-precision floating-point number in register f[*rs1*] to a 64-bit unsigned integer and writes it to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00011	rs1	rm	rd	1010011

fcvt.s.d rd, rs1

$\texttt{f[rd] = f32}_{f64}\texttt{(f[rs1])}$

Floating-point Convert to Single from Double. R-type, RV32D and RV64D.

Converts the double-precision floating-point number in f[*rs1*] to a single-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0100000	00001	rs1	rm	rd	1010011

fcvt.s.l rd, rs1

$\texttt{f[rd] = f32}_{s64}\texttt{(x[rs1])}$

Floating-point Convert to Single from Long. R-type, RV64F only.

Converts the 64-bit two's complement integer in x[*rs1*] to a single-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00010	rs1	rm	rd	1010011

fcvt.s.lu rd, rs1

$\texttt{f[rd] = f32}_{u64}\texttt{(x[rs1])}$

Floating-point Convert to Single from Unsigned Long. R-type, RV64F only.

Converts the 64-bit unsigned integer in x[*rs1*] to a single-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00011	rs1	rm	rd	1010011

fcvt.s.w rd, rs1 `f[rd] = f32`$_{s32}$`(x[rs1])`

Floating-point Convert to Single from Word. R-type, RV32F and RV64F.

Converts the 32-bit two's complement integer in x[*rs1*] to a single-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00000	rs1	rm	rd	1010011

fcvt.s.wu rd, rs1 `f[rd] = f32`$_{u32}$`(x[rs1])`

Floating-point Convert to Single from Unsigned Word. R-type, RV32F and RV64F.

Converts the 32-bit unsigned integer in x[*rs1*] to a single-precision floating-point number and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1101000	00001	rs1	rm	rd	1010011

fcvt.w.d rd, rs1 `x[rd] = sext(s32`$_{f64}$`(f[rs1]))`

Floating-point Convert to Word from Double. R-type, RV32D and RV64D.

Converts the double-precision floating-point number in register f[*rs1*] to a 32-bit two's complement integer and writes the sign-extended result to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00000	rs1	rm	rd	1010011

fcvt.wu.d rd, rs1 `x[rd] = sext(u32`$_{f64}$`(f[rs1]))`

Floating-point Convert to Unsigned Word from Double. R-type, RV32D and RV64D.

Converts the double-precision floating-point number in register f[*rs1*] to a 32-bit unsigned integer and writes the sign-extended result to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100001	00001	rs1	rm	rd	1010011

fcvt.w.s rd, rs1

$\texttt{x[rd] = sext(s32}_{f32}\texttt{(f[rs1]))}$

Floating-point Convert to Word from Single. R-type, RV32F and RV64F.

Converts the single-precision floating-point number in register f[*rs1*] to a 32-bit two's complement integer and writes the sign-extended result to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00000	rs1	rm	rd	1010011

fcvt.wu.s rd, rs1

$\texttt{x[rd] = sext(u32}_{f32}\texttt{(f[rs1]))}$

Floating-point Convert to Unsigned Word from Single. R-type, RV32F and RV64F.

Converts the single-precision floating-point number in register f[*rs1*] to a 32-bit unsigned integer and writes the sign-extended result to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1100000	00001	rs1	rm	rd	1010011

fdiv.d rd, rs1, rs2

$\texttt{f[rd] = f[rs1]} \div \texttt{f[rs2]}$

Floating-point Divide, Double-Precision. R-type, RV32D and RV64D.

Divides the double-precision floating-point number in register f[*rs1*] by f[*rs2*] and writes the rounded double-precision quotient to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0001101	rs2	rs1	rm	rd	1010011

fdiv.s rd, rs1, rs2

$\texttt{f[rd] = f[rs1]} \div \texttt{f[rs2]}$

Floating-point Divide, Single-Precision. R-type, RV32F and RV64F.

Divides the single-precision floating-point number in register f[*rs1*] by f[*rs2*] and writes the rounded single-precision quotient to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0001100	rs2	rs1	rm	rd	1010011

fence pred, succ

`Fence(pred, succ)`

Fence Memory and I/O. I-type, RV32I and RV64I.

Renders preceding memory and I/O accesses in the *pred*ecessor set observable to other threads and devices before subsequent memory and I/O accesses in the *succ*essor set become observable. Bits 3, 2, 1, and 0 in these sets correspond to device **i**nput, device **o**utput, memory **r**eads, and memory **w**rites, respectively. The instruction **fence** r,rw, for example, orders older reads with younger reads and writes, and is encoded with *pred*=0010 and *succ*=0011. If the arguments are omitted, a full **fence** iorw, iorw is implied.

31 28	27 24	23 20	19 15	14 12	11 7	6 0
0000	pred	succ	00000	000	00000	0001111

fence.i

`Fence(Store, Fetch)`

Fence Instruction Stream. I-type, RV32I and RV64I.

Renders stores to instruction memory observable to subsequent instruction fetches.

31 20	19 15	14 12	11 7	6 0
000000000000	00000	001	00000	0001111

feq.d rd, rs1, rs2

`x[rd] = f[rs1] == f[rs2]`

Floating-point Equals, Double-Precision. R-type, RV32D and RV64D.

Writes 1 to x[*rd*] if the double-precision floating-point number in f[*rs1*] equals the number in f[*rs2*], and 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	010	rd	1010011

feq.s rd, rs1, rs2

`x[rd] = f[rs1] == f[rs2]`

Floating-point Equals, Single-Precision. R-type, RV32F and RV64F.

Writes 1 to x[*rd*] if the single-precision floating-point number in f[*rs1*] equals the number in f[*rs2*], and 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	010	rd	1010011

fld rd, offset(rs1) `f[rd] = M[x[rs1] + sext(offset)][63:0]`

Floating-point Load Doubleword. I-type, RV32D and RV64D.

Loads a double-precision floating-point number from memory address x[*rs1*] + *sign-extend*(*offset*) and writes it to f[*rd*].

Compressed forms: **c.fldsp** rd, offset; **c.fld** rd, offset(rs1)

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	011	rd	0000111

---

fle.d rd, rs1, rs2 `x[rd] = f[rs1]` $\leq$ `f[rs2]`

Floating-point Less Than or Equal, Double-Precision. R-type, RV32D and RV64D.

Writes 1 to x[*rd*] if the double-precision floating-point number in f[*rs1*] is less than or equal to the number in f[*rs2*], and 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	000	rd	1010011

---

fle.s rd, rs1, rs2 `x[rd] = f[rs1]` $\leq$ `f[rs2]`

Floating-point Less Than or Equal, Single-Precision. R-type, RV32F and RV64F.

Writes 1 to x[*rd*] if the single-precision floating-point number in f[*rs1*] is less than or equal to the number in f[*rs2*], and 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	000	rd	1010011

---

flt.d rd, rs1, rs2 `x[rd] = f[rs1] < f[rs2]`

Floating-point Less Than, Double-Precision. R-type, RV32D and RV64D.

Writes 1 to x[*rd*] if the double-precision floating-point number in f[*rs1*] is less than the number in f[*rs2*], and 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
1010001	rs2	rs1	001	rd	1010011

---

flt.s rd, rs1, rs2

`x[rd] = f[rs1] < f[rs2]`

Floating-point Less Than, Single-Precision. R-type, RV32F and RV64F.

Writes 1 to x[*rd*] if the single-precision floating-point number in f[*rs1*] is less than the number in f[*rs2*], and 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
1010000	rs2	rs1	001	rd	1010011

flw rd, offset(rs1)

`f[rd] = M[x[rs1] + sext(offset)][31:0]`

Floating-point Load Word. I-type, RV32F and RV64F.

Loads a single-precision floating-point number from memory address x[*rs1*] + *sign-extend*(*offset*) and writes it to f[*rd*].

Compressed forms: **c.flwsp** rd, offset; **c.flw** rd, offset(rs1)

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	010	rd	0000111

fmadd.d rd, rs1, rs2, rs3

`f[rd] = f[rs1]`$\times$`f[rs2]+f[rs3]`

Floating-point Fused Multiply-Add, Double-Precision. R4-type, RV32D and RV64D.

Multiplies the double-precision floating-point numbers in f[*rs1*] and f[*rs2*], adds the unrounded product to the double-precision floating-point number in f[*rs3*], and writes the rounded double-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1000011

fmadd.s rd, rs1, rs2, rs3

`f[rd] = f[rs1]`$\times$`f[rs2]+f[rs3]`

Floating-point Fused Multiply-Add, Single-Precision. R4-type, RV32F and RV64F.

Multiplies the single-precision floating-point numbers in f[*rs1*] and f[*rs2*], adds the unrounded product to the single-precision floating-point number in f[*rs3*], and writes the rounded single-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1000011

fmax.d rd, rs1, rs2

`f[rd] = max(f[rs1], f[rs2])`

Floating-point Maximum, Double-Precision. R-type, RV32D and RV64D.

Copies the larger of the double-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010101	rs2	rs1	001	rd	1010011

fmax.s rd, rs1, rs2

`f[rd] = max(f[rs1], f[rs2])`

Floating-point Maximum, Single-Precision. R-type, RV32F and RV64F.

Copies the larger of the single-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010100	rs2	rs1	001	rd	1010011

fmin.d rd, rs1, rs2

`f[rd] = min(f[rs1], f[rs2])`

Floating-point Minimum, Double-Precision. R-type, RV32D and RV64D.

Copies the smaller of the double-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010101	rs2	rs1	000	rd	1010011

fmin.s rd, rs1, rs2

`f[rd] = min(f[rs1], f[rs2])`

Floating-point Minimum, Single-Precision. R-type, RV32F and RV64F.

Copies the smaller of the single-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010100	rs2	rs1	000	rd	1010011

fmsub.d rd, rs1, rs2, rs3

`f[rd] = f[rs1]×f[rs2]-f[rs3]`

Floating-point Fused Multiply-Subtract, Double-Precision. R4-type, RV32D and RV64D.

Multiplies the double-precision floating-point numbers in f[*rs1*] and f[*rs2*], subtracts the double-precision floating-point number in f[*rs3*] from the unrounded product, and writes the rounded double-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1000111

fmsub.s rd, rs1, rs2, rs3

`f[rd] = f[rs1]×f[rs2]-f[rs3]`

Floating-point Fused Multiply-Subtract, Single-Precision. R4-type, RV32F and RV64F.

Multiplies the single-precision floating-point numbers in f[*rs1*] and f[*rs2*], subtracts the single-precision floating-point number in f[*rs3*] from the unrounded product, and writes the rounded single-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1000111

fmul.d rd, rs1, rs2

`f[rd] = f[rs1] × f[rs2]`

Floating-point Multiply, Double-Precision. R-type, RV32D and RV64D.

Multiplies the double-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] and writes the rounded double-precision product to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	rm	rd	1010011

fmul.s rd, rs1, rs2

`f[rd] = f[rs1] × f[rs2]`

Floating-point Multiply, Single-Precision. R-type, RV32F and RV64F.

Multiplies the single-precision floating-point numbers in registers f[*rs1*] and f[*rs2*] and writes the rounded single-precision product to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0001000	rs2	rs1	rm	rd	1010011

fmv.d rd, rs1

`f[rd] = f[rs1]`

Floating-point Move. Pseudoinstruction, RV32D and RV64D.

Copies the double-precision floating-point number in f[*rs1*] to f[*rd*]. Expands to **fsgnj.d** rd, rs1, rs1.

fmv.d.x rd, rs1

`f[rd] = x[rs1][63:0]`

Floating-point Move Doubleword from Integer. R-type, RV64D only.

Copies the double-precision floating-point number in register x[*rs1*] to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1111001	00000	rs1	000	rd	1010011

fmv.s rd, rs1

`f[rd] = f[rs1]`

Floating-point Move. Pseudoinstruction, RV32F and RV64F.

Copies the single-precision floating-point number in f[*rs1*] to f[*rd*]. Expands to **fsgnj.s** rd, rs1, rs1.

fmv.w.x rd, rs1

`f[rd] = x[rs1][31:0]`

Floating-point Move Word from Integer. R-type, RV32F and RV64F.

Copies the single-precision floating-point number in register x[*rs1*] to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1111000	00000	rs1	000	rd	1010011

fmv.x.d rd, rs1

`x[rd] = f[rs1][63:0]`

Floating-point Move Doubleword to Integer. R-type, RV64D only.

Copies the double-precision floating-point number in register f[*rs1*] to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
1110001	00000	rs1	000	rd	1010011

fmv.x.w rd, rs1 `x[rd] = sext(f[rs1][31:0])`

Floating-point Move Word to Integer. R-type, RV32F and RV64F.

Copies the single-precision floating-point number in register f[*rs1*] to x[*rd*], sign-extending the result for RV64F.

31 25	24 20	19 15	14 12	11 7	6 0
1110000	00000	rs1	000	rd	1010011

fneg.d rd, rs1 `f[rd] = -f[rs1]`

Floating-point Negate. Pseudoinstruction, RV32D and RV64D.

Writes the opposite of the double-precision floating-point number in f[*rs1*] to f[*rd*]. Expands to **fsgnjn.d** rd, rs1, rs1.

fneg.s rd, rs1 `f[rd] = -f[rs1]`

Floating-point Negate. Pseudoinstruction, RV32F and RV64F.

Writes the opposite of the single-precision floating-point number in f[*rs1*] to f[*rd*]. Expands to **fsgnjn.s** rd, rs1, rs1.

fnmadd.d rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]-f[rs3]`

Floating-point Fused Negative Multiply-Add, Double-Precision. R4-type, RV32D and RV64D.

Multiplies the double-precision floating-point numbers in f[*rs1*] and f[*rs2*], negates the result, subtracts the double-precision floating-point number in f[*rs3*] from the unrounded product, and writes the rounded double-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1001111

fnmadd.s rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]-f[rs3]`

Floating-point Fused Negative Multiply-Add, Single-Precision. R4-type, RV32F and RV64F.
Multiplies the single-precision floating-point numbers in f[*rs1*] and f[*rs2*], negates the result, subtracts the single-precision floating-point number in f[*rs3*] from the unrounded product, and writes the rounded single-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1001111

fnmsub.d rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]+f[rs3]`

Floating-point Fused Negative Multiply-Subtract, Double-Precision. R4-type, RV32D and RV64D.
Multiplies the double-precision floating-point numbers in f[*rs1*] and f[*rs2*], negates the result, adds the unrounded product to the double-precision floating-point number in f[*rs3*], and writes the rounded double-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	01	rs2	rs1	rm	rd	1001011

fnmsub.s rd, rs1, rs2, rs3 `f[rd] = -f[rs1]×f[rs2]+f[rs3]`

Floating-point Fused Negative Multiply-Subtract, Single-Precision. R4-type, RV32F and RV64F.
Multiplies the single-precision floating-point numbers in f[*rs1*] and f[*rs2*], negates the result, adds the unrounded product to the single-precision floating-point number in f[*rs3*], and writes the rounded single-precision result to f[*rd*].

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	00	rs2	rs1	rm	rd	1001011

frcsr rd `x[rd] = CSRs[fcsr]`

Floating-point Read Control and Status Register. Pseudoinstruction, RV32F and RV64F.
Copies the floating-point control and status register to x[*rd*]. Expands to **csrrs** rd, fcsr, x0.

frflags rd

`x[rd] = CSRs[fflags]`

Floating-point Read Exception Flags. Pseudoinstruction, RV32F and RV64F.
Copies the floating-point exception flags to x[*rd*]. Expands to **csrrs** rd, fflags, x0.

---

frrm rd

`x[rd] = CSRs[frm]`

Floating-point Read Rounding Mode. Pseudoinstruction, RV32F and RV64F.
Copies the floating-point rounding mode to x[*rd*]. Expands to **csrrs** rd, frm, x0.

---

fscsr rd, rs1

`t = CSRs[fcsr]; CSRs[fcsr] = x[rs1]; x[rd] = t`

Floating-point Swap Control and Status Register. Pseudoinstruction, RV32F and RV64F.
Copies x[*rs1*] to the floating-point control and status register, then copies the previous value of the floating-point control and status register to x[*rd*]. Expands to **csrrw** rd, fcsr, rs1. If *rd* is omitted, x0 is assumed.

---

fsd rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = f[rs2][63:0]`

Floating-point Store Doubleword. S-type, RV32D and RV64D.
Stores the double-precision floating-point number in register f[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).
Compressed forms: **c.fsdsp** rs2, offset; **c.fsd** rs2, offset(rs1)

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	011	offset[4:0]	0100111

---

fsflags rd, rs1

`t = CSRs[fflags]; CSRs[fflags] = x[rs1]; x[rd] = t`

Floating-point Swap Exception Flags. Pseudoinstruction, RV32F and RV64F.
Copies x[*rs1*] to the floating-point exception flags register, then copies the previous floating-point exception flags to x[*rd*]. Expands to **csrrw** rd, fflags, rs1. If *rd* is omitted, x0 is assumed.

---

fsgnj.d rd, rs1, rs2 `f[rd] = {f[rs2][63], f[rs1][62:0]}`

Floating-point Sign Inject, Double-Precision. R-type, RV32D and RV64D.

Constructs a new double-precision floating-point number from the exponent and significand of f[*rs1*], taking the sign from f[*rs2*], and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	000	rd	1010011

fsgnj.s rd, rs1, rs2 `f[rd] = {f[rs2][31], f[rs1][30:0]}`

Floating-point Sign Inject, Single-Precision. R-type, RV32F and RV64F.

Constructs a new single-precision floating-point number from the exponent and significand of f[*rs1*], taking the sign from f[*rs2*], and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	000	rd	1010011

fsgnjn.d rd, rs1, rs2 `f[rd] = {∼f[rs2][63], f[rs1][62:0]}`

Floating-point Sign Inject-Negate, Double-Precision. R-type, RV32D and RV64D.

Constructs a new double-precision floating-point number from the exponent and significand of f[*rs1*], taking the opposite sign of f[*rs2*], and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	001	rd	1010011

fsgnjn.s rd, rs1, rs2 `f[rd] = {∼f[rs2][31], f[rs1][30:0]}`

Floating-point Sign Inject-Negate, Single-Precision. R-type, RV32F and RV64F.

Constructs a new single-precision floating-point number from the exponent and significand of f[*rs1*], taking the opposite sign of f[*rs2*], and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	001	rd	1010011

fsgnjx.d rd, rs1, rs2 `f[rd] = {f[rs1][63] ^ f[rs2][63], f[rs1][62:0]}`

Floating-point Sign Inject-XOR, Double-Precision. R-type, RV32D and RV64D.

Constructs a new double-precision floating-point number from the exponent and significand of f[*rs1*], taking the sign from the XOR of the signs of f[*rs1*] and f[*rs2*], and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010001	rs2	rs1	010	rd	1010011

fsgnjx.s rd, rs1, rs2 `f[rd] = {f[rs1][31] ^ f[rs2][31], f[rs1][30:0]}`

Floating-point Sign Inject-XOR, Single-Precision. R-type, RV32F and RV64F.

Constructs a new single-precision floating-point number from the exponent and significand of f[*rs1*], taking the sign from the XOR of the signs of f[*rs1*] and f[*rs2*], and writes it to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	010	rd	1010011

fsqrt.d rd, rs1 `f[rd] =` $\sqrt{\texttt{f[rs1]}}$

Floating-point Square Root, Double-Precision. R-type, RV32D and RV64D.

Computes the square root of the double-precision floating-point number in register f[*rs1*] and writes the rounded double-precision result to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0101101	00000	rs1	rm	rd	1010011

fsqrt.s rd, rs1 `f[rd] =` $\sqrt{\texttt{f[rs1]}}$

Floating-point Square Root, Single-Precision. R-type, RV32F and RV64F.

Computes the square root of the single-precision floating-point number in register f[*rs1*] and writes the rounded single-precision result to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0101100	00000	rs1	rm	rd	1010011

fsrm rd, rs1

`t = CSRs[frm]; CSRs[frm] = x[rs1]; x[rd] = t`

Floating-point Swap Rounding Mode. Pseudoinstruction, RV32F and RV64F.

Copies x[*rs1*] to the floating-point rounding mode register, then copies the previous floating-point rounding mode to x[*rd*]. Expands to **csrrw** rd, frm, rs1. If *rd* is omitted, x0 is assumed.

---

fsub.d rd, rs1, rs2

`f[rd] = f[rs1] - f[rs2]`

Floating-point Subtract, Double-Precision. R-type, RV32D and RV64D.

Subtracts the double-precision floating-point number in register f[*rs2*] from f[*rs1*] and writes the rounded double-precision difference to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000101	rs2	rs1	rm	rd	1010011

---

fsub.s rd, rs1, rs2

`f[rd] = f[rs1] - f[rs2]`

Floating-point Subtract, Single-Precision. R-type, RV32F and RV64F.

Subtracts the single-precision floating-point number in register f[*rs2*] from f[*rs1*] and writes the rounded single-precision difference to f[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000100	rs2	rs1	rm	rd	1010011

---

fsw rs2, offset(rs1)

`M[x[rs1] + sext(offset)] = f[rs2][31:0]`

Floating-point Store Word. S-type, RV32F and RV64F.

Stores the single-precision floating-point number in register f[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).

Compressed forms: **c.fswsp** rs2, offset; **c.fsw** rs2, offset(rs1)

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100111

---

j offset `pc += sext(offset)`

Jump. Pseudoinstruction, RV32I and RV64I.

Sets the *pc* to the current *pc* plus the sign-extended *offset*. Expands to **jal** x0, offset.

---

jal rd, offset `x[rd] = pc+4; pc += sext(offset)`

Jump and Link. J-type, RV32I and RV64I.

Writes the address of the next instruction (*pc*+4) to x[*rd*], then set the *pc* to the current *pc* plus the sign-extended *offset*. If *rd* is omitted, x1 is assumed.

Compressed forms: **c.j** offset; **c.jal** offset

31 12	11 7	6 0
offset[20|10:1|11|19:12]	rd	1101111

---

jalr rd, offset(rs1) `t=pc+4; pc=(x[rs1]+sext(offset))&∼1; x[rd]=t`

Jump and Link Register. I-type, RV32I and RV64I.

Sets the *pc* to x[*rs1*] + *sign-extend*(*offset*), masking off the least-significant bit of the computed address, then writes the previous *pc*+4 to x[*rd*]. If *rd* is omitted, x1 is assumed.

Compressed forms: **c.jr** rs1; **c.jalr** rs1

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	000	rd	1100111

---

jr rs1 `pc = x[rs1]`

Jump Register. Pseudoinstruction, RV32I and RV64I.

Sets the *pc* to x[*rs1*]. Expands to **jalr** x0, 0(rs1).

---

la rd, symbol `x[rd] = &symbol`

Load Address. Pseudoinstruction, RV32I and RV64I.

Loads the address of *symbol* into x[*rd*]. When assembling position-independent code, it expands into a load from the Global Offset Table: for RV32I, **auipc** rd, offsetHi then **lw** rd, offsetLo(rd); for RV64I, **auipc** rd, offsetHi then **ld** rd, offsetLo(rd). Otherwise, it expands into **auipc** rd, offsetHi then **addi** rd, rd, offsetLo.

---

lb rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][7:0])`

Load Byte. I-type, RV32I and RV64I.

Loads a byte from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes it to x[*rd*], sign-extending the result.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	000	rd	0000011

lbu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][7:0]`

Load Byte, Unsigned. I-type, RV32I and RV64I.

Loads a byte from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes it to x[*rd*], zero-extending the result.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	100	rd	0000011

ld rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][63:0]`

Load Doubleword. I-type, RV64I only.

Loads eight bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*].

Compressed forms: **c.ldsp** rd, offset; **c.ld** rd, offset(rs1)

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	011	rd	0000011

lh rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][15:0])`

Load Halfword. I-type, RV32I and RV64I.

Loads two bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], sign-extending the result.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	001	rd	0000011

lhu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][15:0]`

Load Halfword, Unsigned. I-type, RV32I and RV64I.

Loads two bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], zero-extending the result.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	101	rd	0000011

li rd, immediate `x[rd] = immediate`

Load Immediate. Pseudoinstruction, RV32I and RV64I.

Loads a constant into x[*rd*], using as few instructions as possible. For RV32I, it expands to **lui** and/or **addi**; for RV64I, it's as long as **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

lla rd, symbol `x[rd] = &symbol`

Load Local Address. Pseudoinstruction, RV32I and RV64I.

Loads the address of *symbol* into x[*rd*]. Expands into **auipc** rd, offsetHi then **addi** rd, rd, offsetLo.

lr.d rd, (rs1) `x[rd] = LoadReserved64(M[x[rs1]])`

Load-Reserved Doubleword. R-type, RV64A only.

Loads the eight bytes from memory at address x[*rs1*], writes them to x[*rd*], and registers a reservation on that memory doubleword.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00010	aq	rl	00000	rs1	011	rd	0101111

lr.w rd, (rs1) `x[rd] = LoadReserved32(M[x[rs1]])`

Load-Reserved Word. R-type, RV32A and RV64A.

Loads the four bytes from memory at address x[*rs1*], writes them to x[*rd*], sign-extending the result, and registers a reservation on that memory word.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00010	aq	rl	00000	rs1	010	rd	0101111

lw rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

Load Word. I-type, RV32I and RV64I.

Loads four bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*]. For RV64I, the result is sign-extended.

Compressed forms: **c.lwsp** rd, offset; **c.lw** rd, offset(rs1)

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	010	rd	0000011

lwu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][31:0]`

Load Word, Unsigned. I-type, RV64I only.

Loads four bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], zero-extending the result.

31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	110	rd	0000011

lui rd, immediate `x[rd] = sext(immediate[31:12] << 12)`

Load Upper Immediate. U-type, RV32I and RV64I.

Writes the sign-extended 20-bit *immediate*, left-shifted by 12 bits, to x[*rd*], zeroing the lower 12 bits.

Compressed form: **c.lui** rd, imm

31 12	11 7	6 0
immediate[31:12]	rd	0110111

mret

`ExceptionReturn(Machine)`

Machine-mode Exception Return. R-type, RV32I and RV64I privileged architectures.

Returns from a machine-mode exception handler. Sets the *pc* to CSRs[mepc], the privilege mode to CSRs[mstatus].MPP, CSRs[mstatus].MIE to CSRs[mstatus].MPIE, and CSRs[mstatus].MPIE to 1; and, if user mode is supported, sets CSRs[mstatus].MPP to 0.

31 25	24 20	19 15	14 12	11 7	6 0
0011000	00010	00000	000	00000	1110011

mul rd, rs1, rs2

`x[rd] = x[rs1]` $\times$ `x[rs2]`

Multiply. R-type, RV32M and RV64M.

Multiplies x[*rs1*] by x[*rs2*] and writes the product to x[*rd*]. Arithmetic overflow is ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	000	rd	0110011

mulh rd, rs1, rs2

`x[rd] = (x[rs1]` $_s\times_s$ `x[rs2])` $>>_s$ `XLEN`

Multiply High. R-type, RV32M and RV64M.

Multiplies x[*rs1*] by x[*rs2*], treating the values as two's complement numbers, and writes the upper half of the product to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	001	rd	0110011

mulhsu rd, rs1, rs2

`x[rd] = (x[rs1]` $_s\times_u$ `x[rs2])` $>>_s$ `XLEN`

Multiply High Signed-Unsigned. R-type, RV32M and RV64M.

Multiplies x[*rs1*] by x[*rs2*], treating x[rs1] as a two's complement number and x[rs2] as an unsigned number, and writes the upper half of the product to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	010	rd	0110011

mulhu rd, rs1, rs2

`x[rd] = (x[rs1]` $_u\times_u$ `x[rs2]) >>`$_u$ `XLEN`

Multiply High Unsigned. R-type, RV32M and RV64M.

Multiplies x[*rs1*] by x[*rs2*], treating the values as unsigned numbers, and writes the upper half of the product to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	011	rd	0110011

mulw rd, rs1, rs2

`x[rd] = sext((x[rs1]` $\times$ `x[rs2])[31:0])`

Multiply Word. R-type, RV64M only.

Multiplies x[*rs1*] by x[*rs2*], truncates the product to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	000	rd	0111011

mv rd, rs1

`x[rd] = x[rs1]`

Move. Pseudoinstruction, RV32I and RV64I.

Copies register x[*rs1*] to x[*rd*]. Expands to **addi** rd, rs1, 0.

neg rd, rs2

`x[rd] = -x[rs2]`

Negate. Pseudoinstruction, RV32I and RV64I.

Writes the two's complement of x[*rs2*] to x[*rd*]. Expands to **sub** rd, x0, rs2.

negw rd, rs2

`x[rd] = sext((-x[rs2])[31:0])`

Negate Word. Pseudoinstruction, RV64I only.

Computes the two's complement of x[*rs2*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Expands to **subw** rd, x0, rs2.

nop

Nothing

No operation. Pseudoinstruction, RV32I and RV64I.

Merely advances the *pc* to the next instruction. Expands to **addi** x0, x0, 0.

not rd, rs1

`x[rd] = ∼x[rs1]`

NOT. Pseudoinstruction, RV32I and RV64I.

Writes the ones' complement of x[*rs1*] to x[*rd*]. Expands to **xori** rd, rs1, -1.

---

or rd, rs1, rs2

`x[rd] = x[rs1] | x[rs2]`

OR. R-type, RV32I and RV64I.

Computes the bitwise inclusive-OR of registers x[*rs1*] and x[*rs2*] and writes the result to x[*rd*].

Compressed form: **c.or** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	110	rd	0110011

---

ori rd, rs1, immediate

`x[rd] = x[rs1] | sext(immediate)`

OR Immediate. I-type, RV32I and RV64I.

Computes the bitwise inclusive-OR of the sign-extended *immediate* and register x[*rs1*] and writes the result to x[*rd*].

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	110	rd	0010011

---

rdcycle rd

`x[rd] = CSRs[cycle]`

Read Cycle Counter. Pseudoinstruction, RV32I and RV64I.

Writes the number of cycles that have elapsed to x[*rd*]. Expands to **csrrs** rd, cycle, x0.

---

rdcycleh rd

`x[rd] = CSRs[cycleh]`

Read Cycle Counter High. Pseudoinstruction, RV32I only.

Writes the number of cycles that have elapsed, shifted right by 32 bits, to x[*rd*]. Expands to **csrrs** rd, cycleh, x0.

---

rdinstret rd

x[rd] = CSRs[instret]

Read Instructions-Retired Counter. Pseudoinstruction, RV32I and RV64I.

Writes the number of instructions that have retired to x[*rd*]. Expands to **csrrs** rd, instret, x0.

---

rdinstreth rd

x[rd] = CSRs[instreth]

Read Instructions-Retired Counter High. Pseudoinstruction, RV32I only.

Writes the number of instructions that have retired, shifted right by 32 bits, to x[*rd*]. Expands to **csrrs** rd, instreth, x0.

---

rdtime rd

x[rd] = CSRs[time]

Read Time. Pseudoinstruction, RV32I and RV64I.

Writes the current time to x[*rd*]. The timer frequency is platform-dependent. Expands to **csrrs** rd, time, x0.

---

rdtimeh rd

x[rd] = CSRs[timeh]

Read Time High. Pseudoinstruction, RV32I only.

Writes the current time, shifted right by 32 bits, to x[*rd*]. The timer frequency is platform-dependent. Expands to **csrrs** rd, timeh, x0.

---

rem rd, rs1, rs2

x[rd] = x[rs1] $\%_s$ x[rs2]

Remainder. R-type, RV32M and RV64M.

Divides x[*rs1*] by x[*rs2*], rounding towards zero, treating the values as two's complement numbers, and writes the remainder to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	110	rd	0110011

---

remu rd, rs1, rs2

x[rd] = x[rs1] $\%_u$ x[rs2]

Remainder, Unsigned. R-type, RV32M and RV64M.

Divides x[*rs1*] by x[*rs2*], rounding towards zero, treating the values as unsigned numbers, and writes the remainder to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	111	rd	0110011

remuw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\%_u$ x[rs2][31:0])

Remainder Word, Unsigned. R-type, RV64M only.

Divides the lower 32 bits of x[*rs1*] by the lower 32 bits of x[*rs2*], rounding towards zero, treating the values as unsigned numbers, and writes the sign-extended 32-bit remainder to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	111	rd	0111011

remw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] $\%_s$ x[rs2][31:0])

Remainder Word. R-type, RV64M only.

Divides the lower 32 bits of x[*rs1*] by the lower 32 bits of x[*rs2*], rounding towards zero, treating the values as two's complement numbers, and writes the sign-extended 32-bit remainder to x[*rd*].

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	110	rd	0111011

ret

pc = x[1]

Return. Pseudoinstruction, RV32I and RV64I.

Returns from a subroutine. Expands to **jalr** x0, 0(x1).

sb rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][7:0]`

Store Byte. S-type, RV32I and RV64I.

Stores the least-significant byte in register x[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	000	offset[4:0]	0100011

sc.d rd, rs2, (rs1) `x[rd] = StoreConditional64(M[x[rs1]], x[rs2])`

Store-Conditional Doubleword. R-type, RV64A only.

Stores the eight bytes in register x[*rs2*] to memory at address x[*rs1*], provided there exists a load reservation on that memory address. Writes 0 to x[*rd*] if the store succeeded, or a nonzero error code otherwise.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00011	aq	rl	rs2	rs1	011	rd	0101111

sc.w rd, rs2, (rs1) `x[rd] = StoreConditional32(M[x[rs1]], x[rs2])`

Store-Conditional Word. R-type, RV32A and RV64A.

Stores the four bytes in register x[*rs2*] to memory at address x[*rs1*], provided there exists a load reservation on that memory address. Writes 0 to x[*rd*] if the store succeeded, or a nonzero error code otherwise.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
00011	aq	rl	rs2	rs1	010	rd	0101111

sd rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][63:0]`

Store Doubleword. S-type, RV64I only.

Stores the eight bytes in register x[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).

Compressed forms: **c.sdsp** rs2, offset; **c.sd** rs2, offset(rs1)

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	011	offset[4:0]	0100011

seqz rd, rs1 `x[rd] = (x[rs1] == 0)`

Set if Equal to Zero. Pseudoinstruction, RV32I and RV64I.

Writes 1 to x[*rd*] if x[*rs1*] equals 0, or 0 if not. Expands to **sltiu** rd, rs1, 1.

---

sext.w rd, rs1 `x[rd] = sext(x[rs1][31:0])`

Sign-extend Word. Pseudoinstruction, RV64I only.

Reads the lower 32 bits of x[*rs1*], sign-extends them, and writes the result to x[*rd*]. Expands to **addiw** rd, rs1, 0.

---

sfence.vma rs1, rs2 `Fence(Store, AddressTranslation)`

Fence Virtual Memory. R-type, RV32I and RV64I privileged architectures.

Orders preceding stores to the page tables with subsequent virtual-address translations. When *rs2*=0, translations for all address spaces are affected; otherwise, only translations for address space identified by x[*rs2*] are ordered. When *rs1*=0, translations for all virtual addresses in the selected address spaces are ordered; otherwise, only translations for the page containing virtual address x[*rs1*] in the selected address spaces are ordered.

31 25	24 20	19 15	14 12	11 7	6 0
0001001	rs2	rs1	000	00000	1110011

---

sgtz rd, rs2 x[rd] = (x[rs2] $>_s$ 0)

Set if Greater Than to Zero. Pseudoinstruction, RV32I and RV64I.

Writes 1 to x[*rd*] if x[*rs2*] is greater than 0, or 0 if not. Expands to **slt** rd, x0, rs2.

---

sh rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][15:0]`

Store Halfword. S-type, RV32I and RV64I.

Stores the two least-significant bytes in register x[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	001	offset[4:0]	0100011

---

sw rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][31:0]`

Store Word. S-type, RV32I and RV64I.

Stores the four least-significant bytes in register x[*rs2*] to memory at address x[*rs1*] + *sign-extend*(*offset*).

Compressed forms: **c.swsp** rs2, offset; **c.sw** rs2, offset(rs1)

31 25	24 20	19 15	14 12	11 7	6 0
offset[11:5]	rs2	rs1	010	offset[4:0]	0100011

sll rd, rs1, rs2 `x[rd] = x[rs1] << x[rs2]`

Shift Left Logical. R-type, RV32I and RV64I.

Shifts register x[*rs1*] left by x[*rs2*] bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. The least-significant five bits of x[*rs2*] (or six bits for RV64I) form the shift amount; the upper bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	001	rd	0110011

slli rd, rs1, shamt `x[rd] = x[rs1] << shamt`

Shift Left Logical Immediate. I-type, RV32I and RV64I.

Shifts register x[*rs1*] left by *shamt* bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. For RV32I, the instruction is only legal when *shamt*[5]=0.

Compressed form: **c.slli** rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	001	rd	0010011

slliw rd, rs1, shamt `x[rd] = sext((x[rs1] << shamt)[31:0])`

Shift Left Logical Word Immediate. I-type, RV64I only.

Shifts x[*rs1*] left by *shamt* bit positions. The vacated bits are filled with zeros, the result is truncated to 32 bits, and the sign-extended 32-bit result is written to x[*rd*]. The instruction is only legal when *shamt*[5]=0.

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	001	rd	0011011

sllw rd, rs1, rs2 `x[rd] = sext((x[rs1] << x[rs2][4:0])[31:0])`

Shift Left Logical Word. R-type, RV64I only.

Shifts the lower 32 bits of x[*rs1*] left by x[*rs2*] bit positions. The vacated bits are filled with zeros, and the sign-extended 32-bit result is written to x[*rd*]. The least-significant five bits of x[*rs2*] form the shift amount; the upper bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	001	rd	0111011

slt rd, rs1, rs2 x[rd] = x[rs1] $<_s$ x[rs2]

Set if Less Than. R-type, RV32I and RV64I.

Compares x[*rs1*] and x[*rs2*] as two's complement numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	010	rd	0110011

slti rd, rs1, immediate x[rd] = x[rs1] $<_s$ sext(immediate)

Set if Less Than Immediate. I-type, RV32I and RV64I.

Compares x[*rs1*] and the sign-extended *immediate* as two's complement numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	010	rd	0010011

sltiu rd, rs1, immediate

x[rd] = x[rs1] $<_u$ sext(immediate)

Set if Less Than Immediate, Unsigned. I-type, RV32I and RV64I.

Compares x[*rs1*] and the sign-extended *immediate* as unsigned numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	011	rd	0010011

sltu rd, rs1, rs2

x[rd] = x[rs1] $<_u$ x[rs2]

Set if Less Than, Unsigned. R-type, RV32I and RV64I.

Compares x[*rs1*] and x[*rs2*] as unsigned numbers, and writes 1 to x[*rd*] if x[*rs1*] is smaller, or 0 if not.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	011	rd	0110011

sltz rd, rs1

x[rd] = (x[rs1] $<_s$ 0)

Set if Less Than to Zero. Pseudoinstruction, RV32I and RV64I.

Writes 1 to x[*rd*] if x[*rs1*] is less than zero, or 0 if not. Expands to **slt** rd, rs1, x0.

snez rd, rs2

x[rd] = (x[rs2] $\neq$ 0)

Set if Not Equal to Zero. Pseudoinstruction, RV32I and RV64I.

Writes 0 to x[*rd*] if x[*rs2*] equals 0, or 1 if not. Expands to **sltu** rd, x0, rs2.

sra rd, rs1, rs2

x[rd] = x[rs1] $>>_s$ x[rs2]

Shift Right Arithmetic. R-type, RV32I and RV64I.

Shifts register x[*rs1*] right by x[*rs2*] bit positions. The vacated bits are filled with copies of x[*rs1*]'s most-significant bit, and the result is written to x[*rd*]. The least-significant five bits of x[*rs2*] (or six bits for RV64I) form the shift amount; the upper bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0110011

srai rd, rs1, shamt

x[rd] = x[rs1] >>$_s$ shamt

Shift Right Arithmetic Immediate. I-type, RV32I and RV64I.

Shifts register x[*rs1*] right by *shamt* bit positions. The vacated bits are filled with copies of x[*rs1*]'s most-significant bit, and the result is written to x[*rd*]. For RV32I, the instruction is only legal when *shamt*[5]=0.

Compressed form: **c.srai** rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
010000	shamt	rs1	101	rd	0010011

sraiw rd, rs1, shamt

x[rd] = sext(x[rs1][31:0] >>$_s$ shamt)

Shift Right Arithmetic Word Immediate. I-type, RV64I only.

Shifts the lower 32 bits of x[*rs1*] right by *shamt* bit positions. The vacated bits are filled with copies of x[*rs1*][31], and the sign-extended 32-bit result is written to x[*rd*]. The instruction is only legal when *shamt*[5]=0.

31 26	25 20	19 15	14 12	11 7	6 0
010000	shamt	rs1	101	rd	0011011

sraw rd, rs1, rs2

x[rd] = sext(x[rs1][31:0] >>$_s$ x[rs2][4:0])

Shift Right Arithmetic Word. R-type, RV64I only.

Shifts the lower 32 bits of x[*rs1*] right by x[*rs2*] bit positions. The vacated bits are filled with x[*rs1*][31], and the sign-extended 32-bit result is written to x[*rd*]. The least-significant five bits of x[*rs2*] form the shift amount; the upper bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0111011

sret

`ExceptionReturn(Supervisor)`

Supervisor-mode Exception Return. R-type, RV32I and RV64I privileged architectures.

Returns from a supervisor-mode exception handler. Sets the *pc* to CSRs[sepc], the privilege mode to CSRs[sstatus].SPP, CSRs[sstatus].SIE to CSRs[sstatus].SPIE, CSRs[sstatus].SPIE to 1, and CSRs[sstatus].SPP to 0.

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00010	00000	000	00000	1110011

srl rd, rs1, rs2

`x[rd] = x[rs1]` $>>_u$ `x[rs2]`

Shift Right Logical. R-type, RV32I and RV64I.

Shifts register x[*rs1*] right by x[*rs2*] bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. The least-significant five bits of x[*rs2*] (or six bits for RV64I) form the shift amount; the upper bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	101	rd	0110011

srli rd, rs1, shamt

`x[rd] = x[rs1]` $>>_u$ `shamt`

Shift Right Logical Immediate. I-type, RV32I and RV64I.

Shifts register x[*rs1*] right by *shamt* bit positions. The vacated bits are filled with zeros, and the result is written to x[*rd*]. For RV32I, the instruction is only legal when *shamt*[5]=0.

Compressed form: **c.srli** rd, shamt

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	101	rd	0010011

srliw rd, rs1, shamt

`x[rd] = sext(x[rs1][31:0] >>`$_u$` shamt)`

Shift Right Logical Word Immediate. I-type, RV64I only.

Shifts the lower 32 bits of x[*rs1*] right by *shamt* bit positions. The vacated bits are filled with zeros, and the sign-extended 32-bit result is written to x[*rd*]. The instruction is only legal when *shamt*[5]=0.

31 26	25 20	19 15	14 12	11 7	6 0
000000	shamt	rs1	101	rd	0011011

srlw rd, rs1, rs2

`x[rd] = sext(x[rs1][31:0] >>`$_u$` x[rs2][4:0])`

Shift Right Logical Word. R-type, RV64I only.

Shifts the lower 32 bits of x[*rs1*] right by x[*rs2*] bit positions. The vacated bits are filled with zeros, and the sign-extended 32-bit result is written to x[*rd*]. The least-significant five bits of x[*rs2*] form the shift amount; the upper bits are ignored.

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	101	rd	0111011

sub rd, rs1, rs2

`x[rd] = x[rs1] - x[rs2]`

Subtract. R-type, RV32I and RV64I.

Subtracts register x[*rs2*] from register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

Compressed form: **c.sub** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0110011

subw rd, rs1, rs2 `x[rd] = sext((x[rs1] - x[rs2])[31:0])`

Subtract Word. R-type, RV64I only.

Subtracts register x[*rs2*] from register x[*rs1*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

Compressed form: **c.subw** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0111011

tail symbol `pc = &symbol; clobber x[6]`

Tail call. Pseudoinstruction, RV32I and RV64I.

Sets the *pc* to *symbol*, overwriting x[6] in the process. Expands to **auipc** x6, offsetHi then **jalr** x0, offsetLo(x6).

wfi `while (noInterruptsPending) idle`

Wait for Interrupt. R-type, RV32I and RV64I privileged architectures.

Idles the processor to save energy if no enabled interrupts are currently pending.

31 25	24 20	19 15	14 12	11 7	6 0
0001000	00101	00000	000	00000	1110011

xor rd, rs1, rs2 `x[rd] = x[rs1] ^ x[rs2]`

Exclusive-OR. R-type, RV32I and RV64I.

Computes the bitwise exclusive-OR of registers x[*rs1*] and x[*rs2*] and writes the result to x[*rd*].

Compressed form: **c.xor** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	100	rd	0110011

xori rd, rs1, immediate `x[rd] = x[rs1] ^ sext(immediate)`

Exclusive-OR Immediate. I-type, RV32I and RV64I.

Computes the bitwise exclusive-OR of the sign-extended *immediate* and register x[*rs1*] and writes the result to x[*rd*].

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	100	rd	0010011