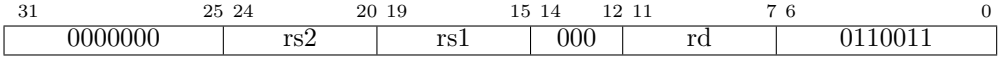
add rd, rs1, rs2 `x[rd] = x[rs1] + x[rs2]`

加。R 型，在 RV32I 和 RV64I 中。

将 x[*rs2*] 与 x[*rs1*] 相加，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.add** rd, rs2; **c.mv** rd, rs2



---

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

加立即数。I 型，在 RV32I 和 RV64I 中。

对 *immediate* 符号扩展后与 x[*rs1*] 相加，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm

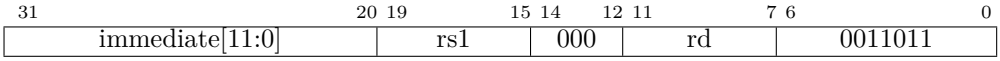
---

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

加立即数字。I 型，仅在 RV64I 中。

对 *immediate* 符号扩展后与 x[*rs1*] 相加，结果截为 32 位，符号扩展后写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.addiw** rd, imm



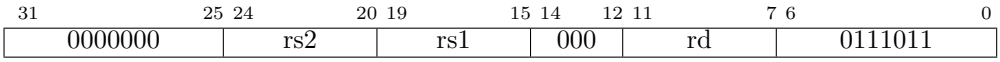
---

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

加字。R 型，仅在 RV64I 中。

将 x[*rs2*] 与 x[*rs1*] 相加，结果截为 32 位，符号扩展后写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.addw** rd, rs2

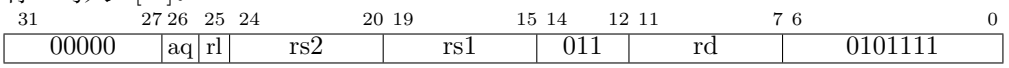


---

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

原子加双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 $t + $ x[*rs2*] 写入该地址，并将 t 写入 x[*rd*]。



---

amoadd.w rd, rs2, (rs1)

`x[rd] = AMO32(M[x[rs1]] + x[rs2])`

原子加字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t + x[*rs2*] 写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

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])`

原子与双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 t 和 x[*rs2*] 的按位与结果写入该地址，并将 t 写入 x[*rd*]。

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])`

原子与字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t 和 x[*rs2*] 的按位与结果写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

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])`

原子最大双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 t 和 x[*rs2*] 中较大者（补码比较）写入该地址，并将 t 写入 x[*rd*]。

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])`

原子最大字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t 和 x[*rs2*] 中较大者（补码比较）写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

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])`

原子无符号最大双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 t 和 x[*rs2*] 中较大者（无符号比较）写入该地址，并将 t 写入 x[*rd*]。

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])`

原子无符号最大字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t 和 x[*rs2*] 中较大者（无符号比较）写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

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])`

原子最小双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 t 和 x[*rs2*] 中较小者（补码比较）写入该地址，并将 t 写入 x[*rd*]。

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])`

原子最小字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t 和 x[*rs2*] 中较小者（补码比较）写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

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])`

原子无符号最小双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 t 和 x[*rs2*] 中较小者（无符号比较）写入该地址，并将 t 写入 x[*rd*]。

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])`

原子无符号最小字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t 和 x[*rs2*] 中较小者（无符号比较）写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

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])`

原子或双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 t 和 x[*rs2*] 的按位或结果写入该地址，并将 t 写入 x[*rd*]。

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])`

原子或字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t 和 x[*rs2*] 的按位或结果写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

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])`

原子交换双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 x[*rs2*] 写入该地址，并将 t 写入 x[*rd*]。

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])`

原子交换字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 x[*rs2*] 写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。

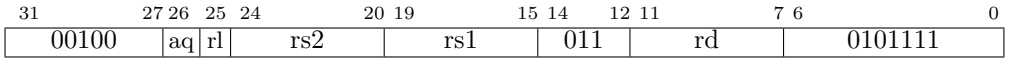
31–27	26	25	24–20	19–15	14–12	11–7	6–0
00001	aq	rl	rs2	rs1	010	rd	0101111

amoxor.d rd, rs2, (rs1)

`x[rd] = AMO64(M[x[rs1]] ^ x[rs2])`

原子异或双字。R 型，仅在 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的双字记为 t，将 t 和 x[*rs2*] 的按位异或结果写入该地址，并将 t 写入 x[*rd*]。

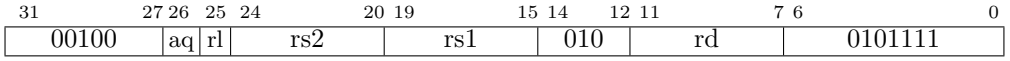


amoxor.w rd, rs2, (rs1)

`x[rd] = AMO32(M[x[rs1]] ^ x[rs2])`

原子异或字。R 型，在 RV32A 和 RV64A 中。

进行如下原子操作：将内存地址为 x[*rs1*] 的字记为 t，将 t 和 x[*rs2*] 的按位异或结果写入该地址，并将 t 的符号扩展结果写入 x[*rd*]。



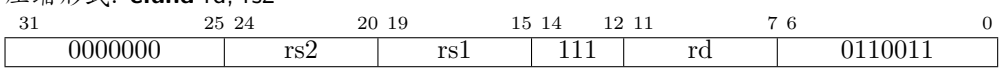
and rd, rs1, rs2

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

与。R 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 和 x[*rs2*] 的按位与结果写入 x[*rd*]。

压缩形式：**c.and** rd, rs2



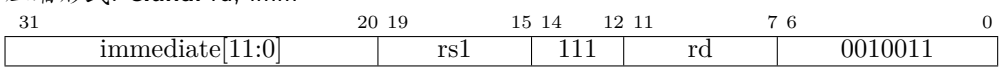
andi rd, rs1, immediate

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

与立即数。I 型，在 RV32I 和 RV64I 中。

对 *immediate* 符号扩展后和 x[*rs1*] 进行按位与，结果写入 x[*rd*]。

压缩形式：**c.andi** rd, imm

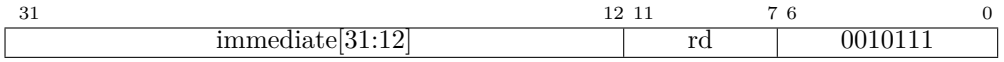


auipc rd, immediate

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

PC 加高位立即数。U 型，在 RV32I 和 RV64I 中。

对 20 位 *immediate* 符号扩展后左移 12 位，与 *pc* 相加，结果写入 x[*rd*]。



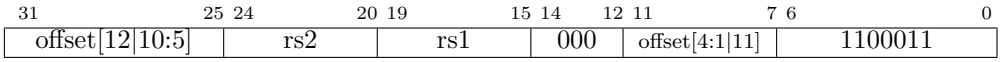
beq rs1, rs2, offset

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

相等时分支。B 型，在 RV32I 和 RV64I 中。

若 x[*rs1*] 和 x[*rs2*] 相等，将 *pc* 设为当前 *pc* 加上符号扩展后的 *offset*。

压缩形式：**c.beqz** rs1, offset



---

beqz rs1, offset

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

等于零时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **beq** rs1, x0, offset。

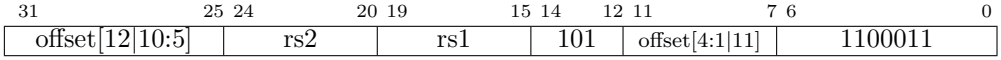
---

bge rs1, rs2, offset

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

大于等于时分支。B 型，在 RV32I 和 RV64I 中。

若 x[*rs1*] 大于等于 x[*rs2*]（补码比较），将 *pc* 设为当前 *pc* 加上符号扩展后的 *offset*。



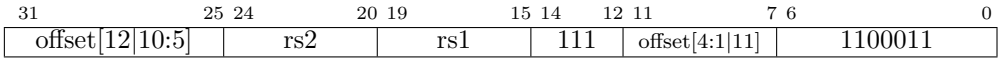
---

bgeu rs1, rs2, offset

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

无符号大于等于时分支。B 型，在 RV32I 和 RV64I 中。

若 x[*rs1*] 大于等于 x[*rs2*]（无符号比较），将 *pc* 设为当前 *pc* 加上符号扩展后的 *offset*。



---

bgez rs1, offset

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

大于等于零时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **bge** rs1, x0, offset。

---

bgt rs1, rs2, offset

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

大于时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **blt** rs2, rs1, offset。

---

bgtu rs1, rs2, offset `if (rs1 >u rs2) pc += sext(offset)`

无符号大于时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **bltu** rs2, rs1, offset。

---

bgtz rs2, offset `if (rs2 >s 0) pc += sext(offset)`

大于零时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **blt** x0, rs2, offset。

---

ble rs1, rs2, offset `if (rs1 ≤s rs2) pc += sext(offset)`

小于等于时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **bge** rs2, rs1, offset。

---

bleu rs1, rs2, offset `if (rs1 ≤u rs2) pc += sext(offset)`

无符号小于等于时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **bgeu** rs2, rs1, offset。

---

blez rs2, offset `if (rs2 ≤s 0) pc += sext(offset)`

小于等于零时分支。伪指令，在 RV32I 和 RV64I 中。

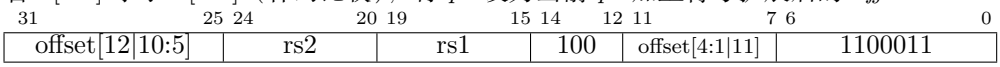
展开为 **bge** x0, rs2, offset。

---

blt rs1, rs2, offset `if (rs1 <s rs2) pc += sext(offset)`

小于时分支。B 型，在 RV32I 和 RV64I 中。

若 x[*rs1*] 小于 x[*rs2*]（补码比较），将 *pc* 设为当前 *pc* 加上符号扩展后的 *offset*。



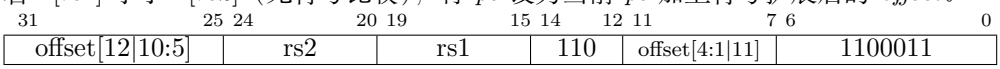
31–25	24–20	19–15	14–12	11–7	6–0
offset[12|10:5]	rs2	rs1	100	offset[4:1|11]	1100011

---

bltu rs1, rs2, offset `if (rs1 <u rs2) pc += sext(offset)`

无符号小于时分支。B 型，在 RV32I 和 RV64I 中。

若 x[*rs1*] 小于 x[*rs2*]（无符号比较），将 *pc* 设为当前 *pc* 加上符号扩展后的 *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

---

bltz rs1, offset

`if (rs1 <s 0) pc += sext(offset)`

小于零时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **blt** rs1, x0, offset。

---

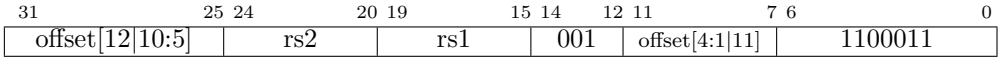
bne rs1, rs2, offset

`if (rs1 ≠ rs2) pc += sext(offset)`

不相等时分支。B 型，在 RV32I 和 RV64I 中。

若 x[*rs1*] 和 x[*rs2*] 不相等，将 *pc* 设为当前 *pc* 加上符号扩展后的 *offset*。

压缩形式：**c.bnez** rs1, offset



---

bnez rs1, offset

`if (rs1 ≠ 0) pc += sext(offset)`

不等于零时分支。伪指令，在 RV32I 和 RV64I 中。

展开为 **bne** rs1, x0, offset。

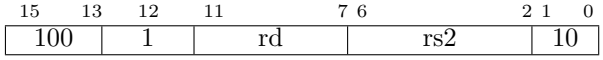
---

c.add rd, rs2

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

加。在 RV32IC 和 RV64IC 中。

展开为 **add** rd, rd, rs2。当 rd=x0 或 rs2=x0 时非法。



---

c.addi rd, imm

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

加立即数。在 RV32IC 和 RV64IC 中。

展开为 **addi** rd, rd, imm。

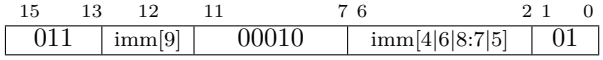
---

c.addi16sp imm

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

栈指针加 *16* 倍立即数。在 RV32IC 和 RV64IC 中。

展开为 **addi** x2, x2, imm。当 imm=0 时非法。



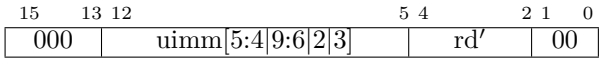
---

c.addi4spn rd′, uimm

`x[8+rd′] = x[2] + uimm`

栈指针无损加 4 倍立即数[1]。在 RV32IC 和 RV64IC 中。

展开为 **addi** rd, x2, uimm，其中 rd=8+rd′。当 uimm=0 时非法。



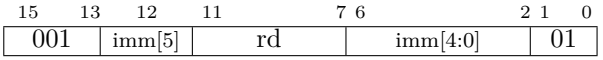
---

c.addiw rd, imm

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

加立即数字。仅在 RV64IC 中。

展开为 **addiw** rd, rd, imm。当 rd=x0 时非法。



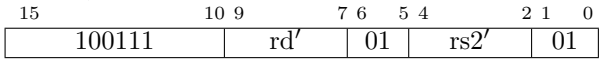
---

c.addw rd′, rs2′

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

加字。仅在 RV64IC 中。

展开为 **addw** rd, rd, rs2，其中 rd=8+rd′ 且 rs2=8+rs2′。



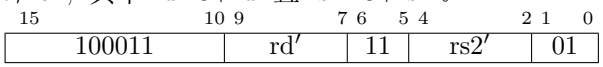
---

c.and rd′, rs2′

`x[8+rd′] = x[8+rd′] & x[8+rs2′]`

与。在 RV32IC 和 RV64IC 中。

展开为 **and** rd, rd, rs2，其中 rd=8+rd′ 且 rs2=8+rs2′。



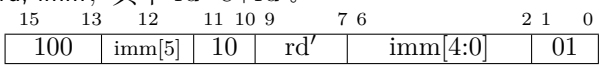
---

c.andi rd′, imm

`x[8+rd′] = x[8+rd′] & sext(imm)`

与立即数。在 RV32IC 和 RV64IC 中。

展开为 **andi** rd, rd, imm，其中 rd=8+rd′。



---

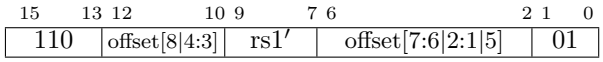
[1]译者注：“无损”指该指令不会破坏 sp 寄存器。

c.beqz rs1′, offset

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

等于零时分支。在 RV32IC 和 RV64IC 中。

展开为 **beq** rs1, x0, offset，其中 rs1=8+rs1′。



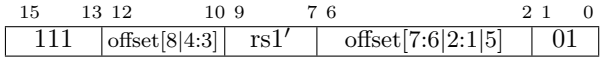
---

c.bnez rs1′, offset

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

不等于零时分支。在 RV32IC 和 RV64IC 中。

展开为 **bne** rs1, x0, offset，其中 rs1=8+rs1′。



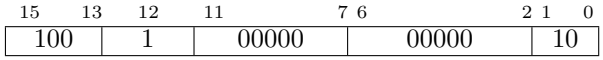
---

c.ebreak

`RaiseException(Breakpoint)`

环境断点。在 RV32IC 和 RV64IC 中。

展开为 **ebreak**。



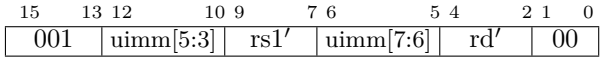
---

c.fld rd′, uimm(rs1′)

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

取浮点双字。在 RV32DC 和 RV64DC 中。

展开为 **fld** rd, uimm(rs1)，其中 rd=8+rd′ 且 rs1=8+rs1′。



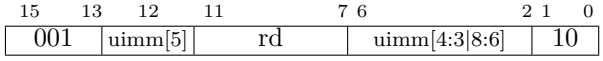
---

c.fldsp rd, uimm

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

相对栈指针取浮点双字。在 RV32DC 和 RV64DC 中。

展开为 **fld** rd, uimm(x2)。



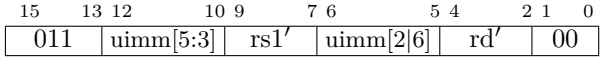
---

c.flw rd′, uimm(rs1′)

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

取浮点字。仅在 RV32FC 中。

展开为 **flw** rd, uimm(rs1)，其中 rd=8+rd′ 且 rs1=8+rs1′。



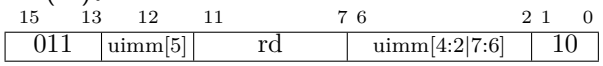
---

c.flwsp rd, uimm

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

相对栈指针取浮点字。仅在 RV32FC 中。

展开为 **flw** rd, uimm(x2)。



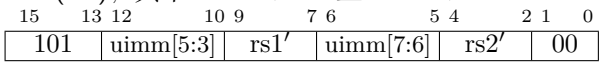
---

c.fsd rs2′, uimm(rs1′)

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

存浮点双字。在 RV32DC 和 RV64DC 中。

展开为 **fsd** rs2, uimm(rs1)，其中 rs2=8+rs2′ 且 rs1=8+rs1′。



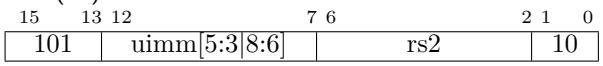
---

c.fsdsp rs2, uimm

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

相对栈指针存浮点双字。在 RV32DC 和 RV64DC 中。

展开为 **fsd** rs2, uimm(x2)。



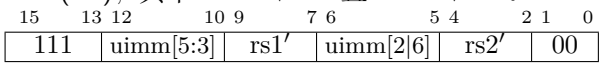
---

c.fsw rs2′, uimm(rs1′)

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

存浮点字。仅在 RV32FC 中。

展开为 **fsw** rs2, uimm(rs1)，其中 rs2=8+rs2′ 且 rs1=8+rs1′。



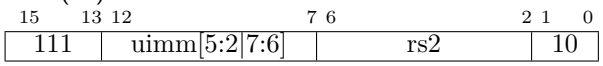
---

c.fswsp rs2, uimm

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

相对栈指针存浮点字。仅在 RV32FC 中。

展开为 **fsw** rs2, uimm(x2)。



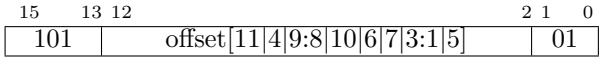
---

c.j offset

`pc += sext(offset)`

跳转。在 RV32IC 和 RV64IC 中。

展开为 **jal** x0, offset。



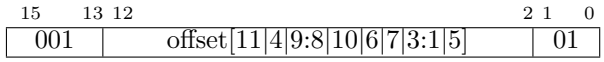
---

c.jal offset

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

跳转并链接。仅在 RV32IC 中。

展开为 **jal** x1, offset。



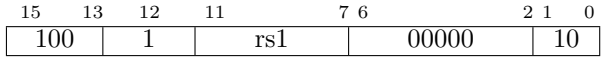
---

c.jalr rs1

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

寄存器跳转并链接。在 RV32IC 和 RV64IC 中。

展开为 **jalr** x1, 0(rs1)。当 rs1=x0 时非法。



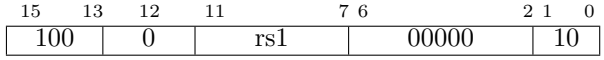
---

c.jr rs1

`pc = x[rs1]`

寄存器跳转。在 RV32IC 和 RV64IC 中。

展开为 **jalr** x0, 0(rs1)。当 rs1=x0 时非法。



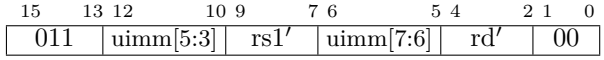
---

c.ld rd′, uimm(rs1′)

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

取双字。仅在 RV64IC 中。

展开为 **ld** rd, uimm(rs1)，其中 rd=8+rd′ 且 rs1=8+rs1′。



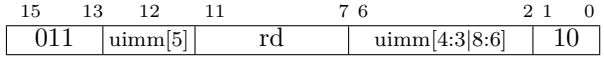
---

c.ldsp rd, uimm

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

相对栈指针取双字。仅在 RV64IC 中。

展开为 **ld** rd, uimm(x2)。当 rd=x0 时非法。



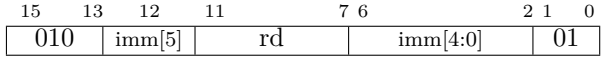
---

c.li rd, imm

`x[rd] = sext(imm)`

装入立即数。在 RV32IC 和 RV64IC 中。

展开为 **addi** rd, x0, imm。



---

c.lui rd, imm

`x[rd] = sext(imm[17:12] << 12)`

装入高位立即数。在 RV32IC 和 RV64IC 中。

展开为 **lui** rd, imm。当 rd=x2 或 imm=0 时非法。

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])`

取字。在 RV32IC 和 RV64IC 中。

展开为 **lw** rd, uimm(rs1)，其中 rd=8+rd′ 且 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

`x[rd] = sext(M[x[2] + uimm][31:0])`

相对栈指针取字。在 RV32IC 和 RV64IC 中。

展开为 **lw** rd, uimm(x2)。当 rd=x0 时非法。

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]`

数据传送。在 RV32IC 和 RV64IC 中。

展开为 **add** rd, x0, rs2。当 rs2=x0 时非法。

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′]`

或。在 RV32IC 和 RV64IC 中。

展开为 **or** rd, rd, rs2，其中 rd=8+rd′ 且 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′]`

存双字。仅在 RV64IC 中。

展开为 **sd** rs2, uimm(rs1)，其中 rs2=8+rs2′ 且 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

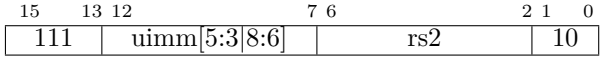
---

c.sdsp rs2, uimm

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

相对栈指针存双字。仅在 RV64IC 中。

展开为 **sd** rs2, uimm(x2)。



---

c.slli rd, uimm

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

逻辑左移立即数。在 RV32IC 和 RV64IC 中。

展开为 **slli** rd, rd, uimm。

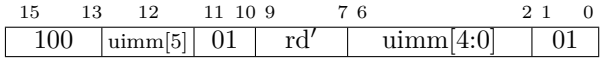
---

c.srai rd′, uimm

`x[8+rd′] = x[8+rd′] >>`$_s$ `uimm`

算术右移立即数。在 RV32IC 和 RV64IC 中。

展开为 **srai** rd, rd, uimm，其中 rd=8+rd′。



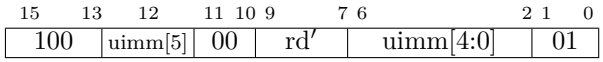
---

c.srli rd′, uimm

`x[8+rd′] = x[8+rd′] >>`$_u$ `uimm`

逻辑右移立即数。在 RV32IC 和 RV64IC 中。

展开为 **srli** rd, rd, uimm，其中 rd=8+rd′。



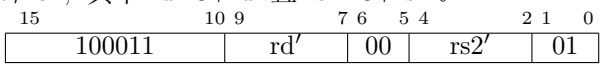
---

c.sub rd′, rs2′

`x[8+rd′] = x[8+rd′] - x[8+rs2′]`

减。在 RV32IC 和 RV64IC 中。

展开为 **sub** rd, rd, rs2，其中 rd=8+rd′ 且 rs2=8+rs2′。



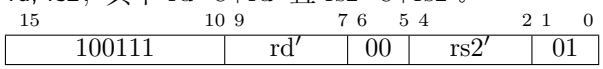
---

c.subw rd′, rs2′

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

减字。仅在 RV64IC 中。

展开为 **subw** rd, rd, rs2，其中 rd=8+rd′ 且 rs2=8+rs2′。

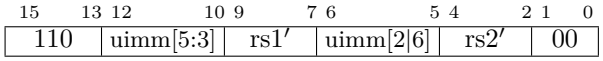


---

c.sw rs2′, uimm(rs1′) `M[x[8+rs1′] + uimm][31:0] = x[8+rs2′]`

存字。在 RV32IC 和 RV64IC 中。

展开为 **sw** rs2, uimm(rs1)，其中 rs2=8+rs2′ 且 rs1=8+rs1′。

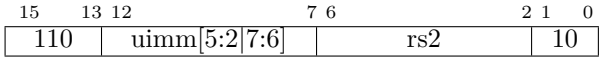


---

c.swsp rs2, uimm `M[x[2] + uimm][31:0] = x[rs2]`

相对栈指针存字。在 RV32IC 和 RV64IC 中。

展开为 **sw** rs2, uimm(x2)。

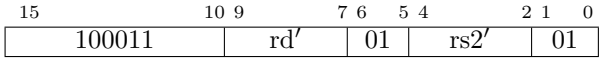


---

c.xor rd′, rs2′ `x[8+rd′] = x[8+rd′] ^ x[8+rs2′]`

异或。在 RV32IC 和 RV64IC 中。

展开为 **xor** rd, rd, rs2，其中 rd=8+rd′ 且 rs2=8+rs2′。



---

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

调用。伪指令，在 RV32I 和 RV64I 中。

将下一条指令的地址（pc+8）写入 x[rd]，然后将 pc 设为 $symbol$。展开为 **auipc** rd, offsetHi 和 **jalr** rd, offsetLo(rd)。若省略 rd，则默认为 x1。

---

csrc csr, rs1 `CSRs[csr] &= ~x[rs1]`

控制状态寄存器清位。伪指令，在 RV32I 和 RV64I 中。

对于 x[$rs1$] 中每一个为 1 的位，将控制状态寄存器 csr 的对应位清零。展开为 **csrrc** x0, csr, rs1。

---

csrci csr, zimm[4:0] `CSRs[csr] &= ~zimm`

控制状态寄存器清位立即数。伪指令，在 RV32I 和 RV64I 中。

对于 5 位立即数零扩展后中每一个为 1 的位，将控制状态寄存器 csr 的对应位清零。展开为 **csrrci** x0, csr, zimm。

---

csrr rd, csr `x[rd] = CSRs[csr]`

控制状态寄存器读。伪指令，在 RV32I 和 RV64I 中。

将控制状态寄存器 *csr* 写入 x[*rd*]。展开为 **csrrs** rd, csr, x0。

---

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

控制状态寄存器读后清位。I 型，在 RV32I 和 RV64I 中。

记控制状态寄存器 *csr* 的值为 *t*。将 x[*rs1*] 的反码和 *t* 按位与，结果写入 *csr*，再将 *t* 写入 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`

控制状态寄存器读后清位立即数。I 型，在 RV32I 和 RV64I 中。

记控制状态寄存器 *csr* 的值为 *t*。将 5 位立即数 *zimm* 零扩展后的反码和 *t* 按位与，结果写入 *csr*，再将 *t* 写入 x[*rd*]（*csr* 中的第 5 及更高的位不变）。

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`

控制状态寄存器读后置位。I 型，在 RV32I 和 RV64I 中。

记控制状态寄存器 *csr* 的值为 *t*。将 *t* 和 x[*rs1*] 的按位或结果写入 *csr*，再将 *t* 写入 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`

控制状态寄存器读后置位立即数。I 型，在 RV32I 和 RV64I 中。

记控制状态寄存器 *csr* 的值为 *t*。将 5 位立即数 *zimm* 零扩展后，和 *t* 按位或，结果写入 *csr*，再将 *t* 写入 x[*rd*]（*csr* 中的第 5 及更高的位不变）。

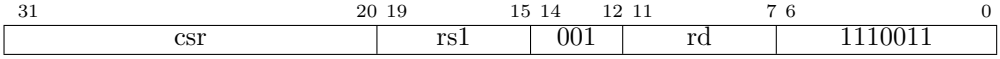
31 20	19 15	14 12	11 7	6 0
csr	zimm[4:0]	110	rd	1110011

---

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

控制状态寄存器读后写。I 型，在 RV32I 和 RV64I 中。

记控制状态寄存器 *csr* 的值为 *t*。将 x[*rs1*] 写入 *csr*，再将 *t* 写入 x[*rd*]。

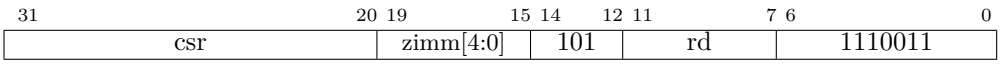


---

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

控制状态寄存器读后写立即数。I 型，在 RV32I 和 RV64I 中。

将控制状态寄存器 *csr* 的值复制到 x[*rd*] 中，再将 5 位立即数 *zimm* 的零扩展结果写入 *csr*。



---

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

控制状态寄存器置位。伪指令，在 RV32I 和 RV64I 中。

对于 x[*rs1*] 中每一个为 1 的位，将控制状态寄存器 *csr* 的对应位置 1。展开为 **csrrs** x0, csr, rs1。

---

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

控制状态寄存器置位立即数。伪指令，在 RV32I 和 RV64I 中。

对于 5 位立即数 *zimm* 的零扩展结果中每一个为 1 的位，将控制状态寄存器 *csr* 的对应位置 1。展开为 **csrrsi** x0, csr, zimm。

---

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

控制状态寄存器写。伪指令，在 RV32I 和 RV64I 中。

将 x[*rs1*] 的值写入控制状态寄存器 *csr*。展开为 **csrrw** x0, csr, rs1。

---

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

控制状态寄存器写立即数。伪指令，在 RV32I 和 RV64I 中。

将 5 位立即数 *zimm* 的零扩展结果写入控制状态寄存器 *csr*。展开为 **csrrwi** x0, csr, zimm。

---

div rd, rs1, rs2

`x[rd] = x[rs1] ÷s x[rs2]`

除。R 型，在 RV32M 和 RV64M 中。

x[*rs1*] 除以 x[*rs2*]（补码除法），结果向零舍入，将商写入 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] ÷u x[rs2]`

无符号除。R 型，在 RV32M 和 RV64M 中。

x[*rs1*] 除以 x[*rs2*]（无符号除法），结果向零舍入，将商写入 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] ÷u x[rs2][31:0])`

无符号除字。R 型，仅在 RV64M 中。

x[*rs1*] 的低 32 位除以 x[*rs2*] 的低 32 位（无符号除法），结果向零舍入，将 32 位商的符号扩展结果写入 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] ÷s x[rs2][31:0])`

除字。R 型，仅在 RV64M 中。

x[*rs1*] 的低 32 位除以 x[*rs2*] 的低 32 位（补码除法），结果向零舍入，将 32 位商的符号扩展结果写入 x[*rd*]。

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

ebreak

`RaiseException(Breakpoint)`

环境断点。I 型，在 RV32I 和 RV64I 中。

通过抛出断点异常调用调试器。

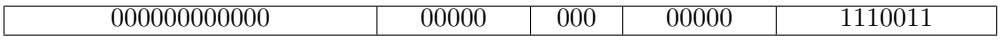
31 20	19 15	14 12	11 7	6 0
000000000001	00000	000	00000	1110011

ecall

`RaiseException(EnvironmentCall)`

环境调用。I 型，在 RV32I 和 RV64I 中。

通过抛出环境调用异常调用执行环境。



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

---

fabs.d rd, rs1

`f[rd] = |f[rs1]|`

浮点数绝对值。伪指令，在 RV32D 和 RV64D 中。

将双精度浮点数 f[*rs1*] 的绝对值写入 f[*rd*]。展开为 **fsgnjx.d** rd, rs1, rs1。

---

fabs.s rd, rs1

`f[rd] = |f[rs1]|`

浮点数绝对值。伪指令，在 RV32F 和 RV64F 中。

将单精度浮点数 f[*rs1*] 的绝对值写入 f[*rd*]。展开为 **fsgnjx.s** rd, rs1, rs1。

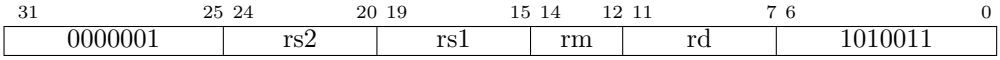
---

fadd.d rd, rs1, rs2

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

双精度浮点加。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相加，将舍入后的双精度结果写入 f[*rd*]。



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

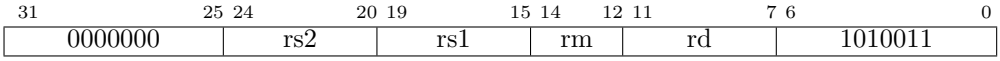
---

fadd.s rd, rs1, rs2

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

单精度浮点加。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相加，将舍入后的单精度结果写入 f[*rd*]。



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

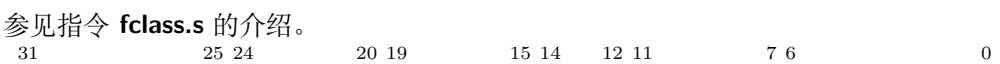
---

fclass.d rd, rs1

`x[rd] = classify`$_d$`(f[rs1])`

双精度浮点分类。R 型，在 RV32D 和 RV64D 中。

将一个表示 f[*rs1*] 中双精度浮点数类别的掩码写入 x[*rd*]。关于写入 x[*rd*] 的值的含义，参见指令 **fclass.s** 的介绍。



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

---

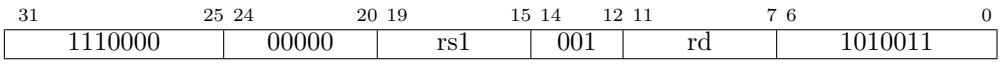
fclass.s rd, rs1

`x[rd] = classify`$_s$`(f[rs1])`

单精度浮点分类。R 型，在 RV32F 和 RV64F 中。

将一个表示 f[*rs1*] 中单精度浮点数类别的掩码写入 x[*rd*]。x[*rd*] 中有且仅有一位被置 1，见下表：

x[*rd*] 位	含义
0	f[*rs1*] 为 $-\infty$
1	f[*rs1*] 为规格化负数
2	f[*rs1*] 为非规格化负数
3	f[*rs1*] 为 -0
4	f[*rs1*] 为 $+0$
5	f[*rs1*] 为非规格化正数
6	f[*rs1*] 为规格化正数
7	f[*rs1*] 为 $+\infty$
8	f[*rs1*] 为发信号（signaling）NaN
9	f[*rs1*] 为不发信号（quiet）NaN

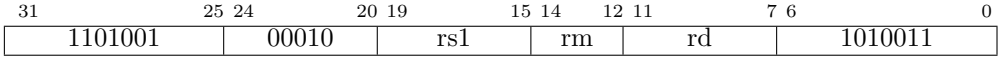


fcvt.d.l rd, rs1

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

长字转换为双精度浮点。R 型，仅在 RV64D 中。

将 x[*rs1*] 中的 64 位有符号整数（补码表示）转换为双精度浮点数，并写入 f[*rd*]。

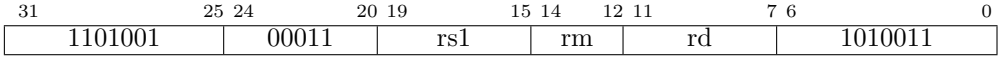


fcvt.d.lu rd, rs1

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

无符号长字转换为双精度浮点。R 型，仅在 RV64D 中。

将 x[*rs1*] 中的 64 位无符号整数转换为双精度浮点数，并写入 f[*rd*]。

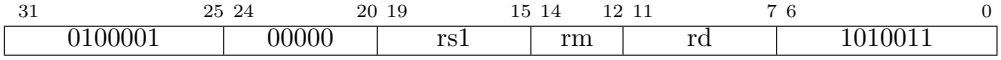


fcvt.d.s rd, rs1

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

单精度浮点转换为双精度浮点。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 中的单精度浮点数转换为双精度浮点数，并写入 f[*rd*]。

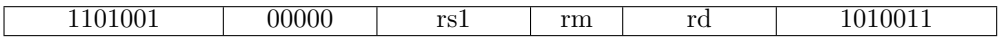


fcvt.d.w rd, rs1

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

字转换为双精度浮点。R 型，在 RV32D 和 RV64D 中。

将 x[*rs1*] 中的 32 位有符号整数（补码表示）转换为双精度浮点数，并写入 f[*rd*]。



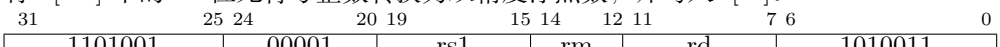
31–25	24–20	19–15	14–12	11–7	6–0
1101001	00000	rs1	rm	rd	1010011

fcvt.d.wu rd, rs1

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

无符号字转换为双精度浮点。R 型，在 RV32D 和 RV64D 中。

将 x[*rs1*] 中的 32 位无符号整数转换为双精度浮点数，并写入 f[*rd*]。



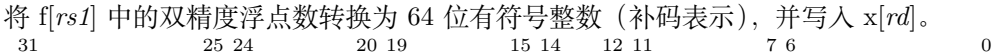
31–25	24–20	19–15	14–12	11–7	6–0
1101001	00001	rs1	rm	rd	1010011

fcvt.l.d rd, rs1

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

双精度浮点转换为长字。R 型，仅在 RV64D 中。

将 f[*rs1*] 中的双精度浮点数转换为 64 位有符号整数（补码表示），并写入 x[*rd*]。



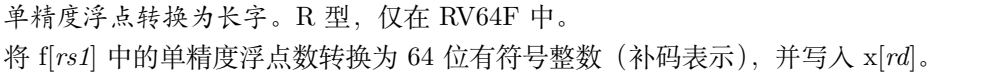
31–25	24–20	19–15	14–12	11–7	6–0
1100001	00010	rs1	rm	rd	1010011

fcvt.l.s rd, rs1

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

单精度浮点转换为长字。R 型，仅在 RV64F 中。

将 f[*rs1*] 中的单精度浮点数转换为 64 位有符号整数（补码表示），并写入 x[*rd*]。



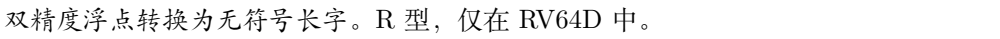
31–25	24–20	19–15	14–12	11–7	6–0
1100000	00010	rs1	rm	rd	1010011

fcvt.lu.d rd, rs1

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

双精度浮点转换为无符号长字。R 型，仅在 RV64D 中。

将 f[*rs1*] 中的双精度浮点数转换为 64 位无符号整数，并写入 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

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

单精度浮点转换为无符号长字。R 型，仅在 RV64F 中。

将 f[*rs1*] 中的单精度浮点数转换为 64 位无符号整数，并写入 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

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

双精度转换为单精度浮点。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 中的双精度浮点数转换为单精度浮点数，并写入 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

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

长字转换为单精度浮点转换。R 型，仅在 RV64F 中。

将 x[*rs1*] 中的 64 位有符号整数（补码表示）转换为单精度浮点数，并写入 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

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

无符号长字转换为单精度浮点。R 型，仅在 RV64F 中。

将 x[*rs1*] 中的 64 位无符号整数转换为单精度浮点数，并写入 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])`

字转换为单精度浮点。R 型，在 RV32F 和 RV64F 中。

将 x[*rs1*] 中的 32 位有符号整数（补码表示）转换为单精度浮点数，并写入 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])`

无符号字转换为单精度浮点。R 型，在 RV32F 和 RV64F 中。

将 x[*rs1*] 中的 32 位无符号整数转换为单精度浮点数，并写入 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]))`

双精度浮点转换为字。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 中的双精度浮点数转换为 32 位有符号整数（补码表示），符号扩展后写入 x[*rd*]。

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

---

fcvt.w.s rd, rs1

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

单精度浮点转换为字。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 中的单精度浮点数转换为 32 位有符号整数（补码表示），符号扩展后写入 x[*rd*]。

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

---

fcvt.wu.d rd, rs1

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

双精度浮点转换为无符号字。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 中的双精度浮点数转换为 32 位无符号整数，符号扩展后写入 x[*rd*]。

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

---

fcvt.wu.s rd, rs1

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

单精度浮点转换为无符号字。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 中的单精度浮点数转换为 32 位无符号整数，符号扩展后写入 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

`f[rd] = f[rs1] ÷ f[rs2]`

双精度浮点除法。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相除，并将舍入后的商写入 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

`f[rd] = f[rs1] ÷ f[rs2]`

单精度浮点除。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相除，并将舍入后的商写入 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)`

内存和 *I/O* 屏障。I 型，在 RV32I 和 RV64I 中。

在后继集合（*succ*）中的内存和 I/O 访问对其他线程和设备可见之前，保证前驱集合（*pred*）中的内存及 I/O 访问对其他线程和设备可见。上述集合中的第 3、2、1 和 0 位分别表示设备输入、设备输出、内存读、内存写。如 **fence** r,rw（通过 *pred*=0010 和 *succ*=0011 编码）用于将旧的读操作与新的读写操作定序。若省略参数，则默认为 **fence** iorw, iorw。

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)`

指令流屏障。I 型，在 RV32I 和 RV64I 中。

使内存指令区域的写入对后续取指操作可见。

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]`

双精度浮点相等。R 型，在 RV32D 和 RV64D 中。

若 f[*rs1*] 和 f[*rs2*] 中的双精度浮点数相等，则向 x[*rd*] 写 1，否则写 0。

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]`

单精度浮点相等。R 型，在 RV32F 和 RV64F 中。

若 f[*rs1*] 和 f[*rs2*] 中的单精度浮点数相等，则向 x[*rd*] 写 1，否则写 0。

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]`

取浮点双字。I 型，在 RV32D 和 RV64D 中。

从内存地址 x[*rs1*] + *sign-extend*(*offset*) 中读取双精度浮点数，并写入 f[*rd*]。

压缩形式：**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]`

双精度浮点小于等于。R 型，在 RV32D 和 RV64D 中。

若 f[*rs1*] 中的双精度浮点数小于等于 f[*rs2*]，则向 x[*rd*] 中写 1，否则写 0。

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]`

单精度浮点小于等于。R 型，在 RV32F 和 RV64F 中。

若 f[*rs1*] 中的单精度浮点数小于等于 f[*rs2*]，则向 x[*rd*] 中写 1，否则写 0。

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]`

双精度浮点小于。R 型，在 RV32D 和 RV64D 中。

若 f[*rs1*] 中的双精度浮点数小于 f[*rs2*]，则向 x[*rd*] 写 1，否则写 0。

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]`

单精度浮点小于。R 型，在 RV32F 和 RV64F 中。

若 f[*rs1*] 中的单精度浮点数小于 f[*rs2*]，则向 x[*rd*] 写 1，否则写 0。

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

---

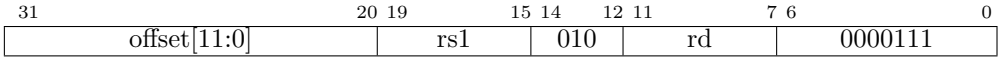
flw rd, offset(rs1)

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

取浮点字。I 型，在 RV32F 和 RV64F 中。

从内存地址 x[*rs1*] + *sign-extend*(*offset*) 中读取单精度浮点数，并写入 f[*rd*]。

压缩形式：**c.flwsp** rd, offset; **c.flw** rd, offset(rs1)



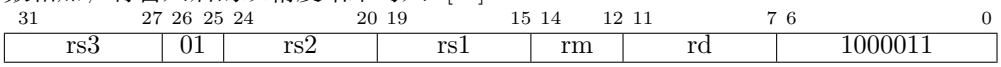
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
offset[11:0]	rs1	010	rd	0000111

fmadd.d rd, rs1, rs2, rs3

`f[rd] = f[rs1]×f[rs2]+f[rs3]`

双精度浮点乘加。R4 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相乘，并将未舍入的积与 f[*rs3*] 中的双精度浮点数相加，将舍入后的双精度结果写入 f[*rd*]。



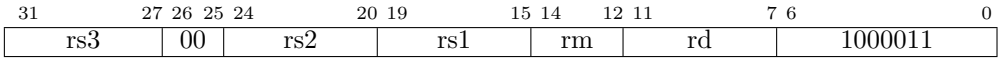
31 — 27	26 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
rs3	01	rs2	rs1	rm	rd	1000011

fmadd.s rd, rs1, rs2, rs3

`f[rd] = f[rs1]×f[rs2]+f[rs3]`

单精度浮点乘加。R4 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相乘，并将未舍入的积与 f[*rs3*] 中的单精度浮点数相加，将舍入后的单精度结果写入 f[*rd*]。



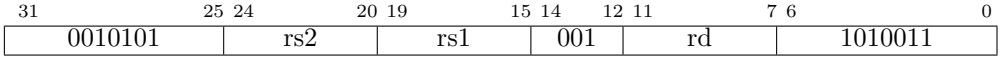
31 — 27	26 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
rs3	00	rs2	rs1	rm	rd	1000011

fmax.d rd, rs1, rs2

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

双精度浮点最大值。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 和 f[*rs2*] 中的双精度浮点数较大者写入 f[*rd*]。



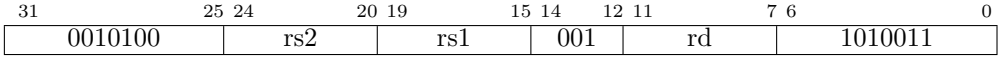
31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
0010101	rs2	rs1	001	rd	1010011

fmax.s rd, rs1, rs2

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

单精度浮点最大值。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 和 f[*rs2*] 中的单精度浮点数较大者写入 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])`

双精度浮点最小值。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 和 f[*rs2*] 中的双精度浮点数较小者写入 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])`

单精度浮点最小值。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 和 f[*rs2*] 中的单精度浮点数较小者写入 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]`

双精度浮点乘减。R4 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相乘，并将未舍入的积与 f[*rs3*] 中的双精度浮点数相减，将舍入后的双精度结果写入 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]`

单精度浮点乘减。R4 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相乘，并将未舍入的积与 f[*rs3*] 中的单精度浮点数相减，将舍入后的单精度结果写入 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]`

双精度浮点乘。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相乘，将舍入后的双精度结果写入 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]`

单精度浮点乘。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相乘，将舍入后的单精度结果写入 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]`

双精度浮点数据传送。伪指令，在 RV32D 和 RV64D 中。

将 f[*rs1*] 中的双精度浮点数复制到 f[*rd*]。展开为 **fsgnj.d** rd, rs1, rs1。

---

fmv.d.x rd, rs1 `f[rd] = x[rs1][63:0]`

从整数传送双字到浮点。R 型，仅在 RV64D 中。

将 x[*rs1*] 中的双精度浮点数复制到 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]`

单精度浮点数据传送。伪指令，在 RV32F 和 RV64F 中。

将 f[*rs1*] 中的单精度浮点数复制到 f[*rd*]。展开为 **fsgnj.s** rd, rs1, rs1。

---

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

从整数传送字到浮点。R 型，在 RV32F 和 RV64F 中。

将 x[*rs1*] 中的单精度浮点数复制到 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]`

从浮点传送双字到整数。R 型，仅在 RV64D 中。

将 f[*rs1*] 中的双精度浮点数复制到 x[*rd*]。

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

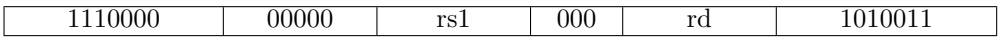
---

fmv.x.w rd, rs1

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

从浮点传送字到整数。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 中的单精度浮点数复制到 x[*rd*]，在 RV64F 中额外对结果进行符号扩展。



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]`

双精度浮点取负。伪指令，在 RV32D 和 RV64D 中。

将 f[*rs1*] 中的双精度浮点数取负后写入 f[*rd*]。展开为 **fsgnjn.d** rd, rs1, rs1。

fneg.s rd, rs1

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

单精度浮点取负。伪指令，在 RV32F 和 RV64F 中。

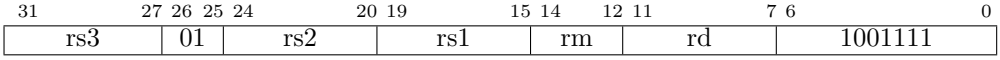
将 f[*rs1*] 中的单精度浮点数取负后写入 f[*rd*]。展开为 **fsgnjn.s** rd, rs1, rs1。

fnmadd.d rd, rs1, rs2, rs3

`f[rd] = -f[rs1]×f[rs2]-f[rs3]`

双精度浮点乘加取负。R4 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相乘，结果取负，并将未舍入的积与 f[*rs3*] 中的双精度浮点数相减，将舍入后的双精度结果写入 f[*rd*]。



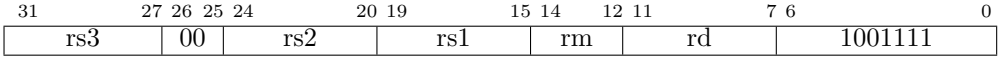
31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	01	rs2	rs1	rm	rd	1001111

fnmadd.s rd, rs1, rs2, rs3

`f[rd] = -f[rs1]×f[rs2]-f[rs3]`

单精度浮点乘加取负。R4 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相乘，结果取负，并将未舍入的积与 f[*rs3*] 中的单精度浮点数相减，将舍入后的单精度结果写入 f[*rd*]。



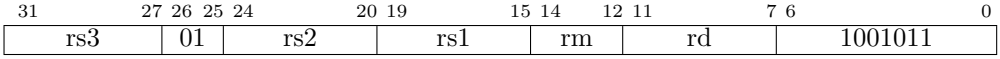
31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	00	rs2	rs1	rm	rd	1001111

fnmsub.d rd, rs1, rs2, rs3

`f[rd] = -f[rs1]×f[rs2]+f[rs3]`

双精度浮点乘减取负。R4 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相乘，结果取负，并将未舍入的积与 f[*rs3*] 中的双精度浮点数相加，将舍入后的双精度结果写入 f[*rd*]。

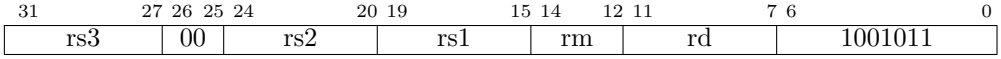


31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	01	rs2	rs1	rm	rd	1001011

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

单精度浮点乘减取负。R4 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相乘，结果取负，并将未舍入的积与 f[*rs3*] 中的单精度浮点数相加，将舍入后的单精度结果写入 f[*rd*]。



---

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

读浮点控制状态寄存器。伪指令，在 RV32F 和 RV64F 中。

将浮点控制状态寄存器写入 x[*rd*]。展开为 **csrrs** rd, fcsr, x0。

---

frflags rd `x[rd] = CSRs[fflags]`

读浮点异常标志。伪指令，在 RV32F 和 RV64F 中。

将浮点异常标志写入 x[*rd*]。展开为 **csrrs** rd, fflags, x0。

---

frrm rd `x[rd] = CSRs[frm]`

读浮点舍入模式。伪指令，在 RV32F 和 RV64F 中。

将浮点舍入模式写入 x[*rd*]。展开为 **csrrs** rd, frm, x0。

---

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

交换浮点控制状态寄存器。伪指令，在 RV32F 和 RV64F 中。

将 x[*rs1*] 写入浮点控制状态寄存器，并将浮点控制状态寄存器的原值写入 x[*rd*]。展开为 **csrrw** rd, fcsr, rs1。若省略 *rd*，则默认为 x0。

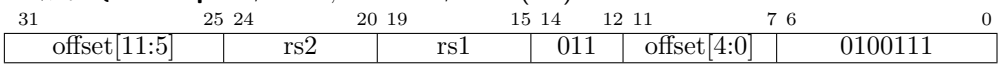
---

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

存浮点双字。S 型，在 RV32D 和 RV64D 中。

将 f[*rs2*] 中的双精度浮点数写入内存地址 x[*rs1*] + *sign-extend*(*offset*) 中。

压缩形式：**c.fsdsp** rs2, offset; **c.fsd** rs2, offset(rs1)



---

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

交换浮点异常标志。伪指令，在 RV32F 和 RV64F 中。

将 x[*rs1*] 写入浮点异常标志寄存器，并将浮点异常标志寄存器的原值写入 x[*rd*]。展开为 csrrw rd, fflags, rs1。若省略 *rd*，则默认为 x0。

---

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

双精度浮点符号注入。R 型，在 RV32D 和 RV64D 中。

用 f[*rs1*] 的阶码和尾数，以及 f[*rs2*] 的符号位，组成一个新的双精度浮点数，并将其写入 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]}`

单精度浮点符号注入。R 型，在 RV32F 和 RV64F 中。

用 f[*rs1*] 的阶码和尾数，以及 f[*rs2*] 的符号位，组成一个新的单精度浮点数，并将其写入 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]}`

双精度浮点符号取反注入。R 型，在 RV32D 和 RV64D 中。

用 f[*rs1*] 的阶码和尾数，以及 f[*rs2*] 的符号位取反，组成一个新的双精度浮点数，并将其写入 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]}`

单精度浮点符号取反注入。R 型，在 RV32F 和 RV64F 中。

用 f[*rs1*] 的阶码和尾数，以及 f[*rs2*] 的符号位取反，组成一个新的单精度浮点数，并将其写入 f[*rd*]。

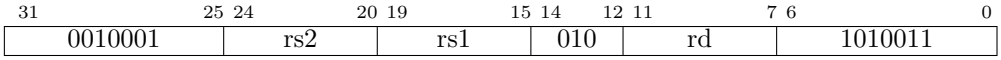
31 25	24 20	19 15	14 12	11 7	6 0
0010000	rs2	rs1	001	rd	1010011

---

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

双精度浮点符号异或注入。R 型，在 RV32D 和 RV64D 中。

用 f[*rs1*] 的阶码和尾数，以及 f[*rs1*] 和 f[*rs2*] 符号位的异或，组成一个新的双精度浮点数，并将其写入 f[*rd*]。



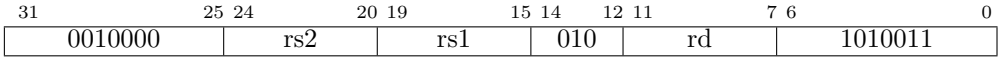
31–25	24–20	19–15	14–12	11–7	6–0
0010001	rs2	rs1	010	rd	1010011

---

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

单精度浮点符号异或注入。R 型，在 RV32F 和 RV64F 中。

用 f[*rs1*] 的阶码和尾数，以及 f[*rs1*] 和 f[*rs2*] 符号位的异或，组成一个新的单精度浮点数，并将其写入 f[*rd*]。



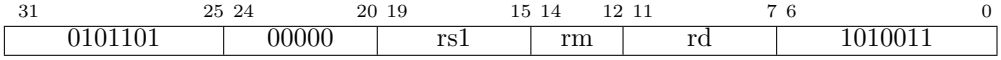
31–25	24–20	19–15	14–12	11–7	6–0
0010000	rs2	rs1	010	rd	1010011

---

fsqrt.d rd, rs1 `f[rd] = √f[rs1]`

双精度浮点求平方根。R 型，在 RV32D 和 RV64D 中。

计算 f[*rs1*] 中的双精度浮点数的平方根，将舍入后的双精度结果写入 f[*rd*]。



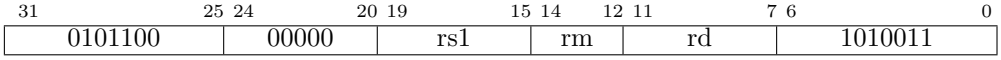
31–25	24–20	19–15	14–12	11–7	6–0
0101101	00000	rs1	rm	rd	1010011

---

fsqrt.s rd, rs1 `f[rd] = √f[rs1]`

单精度浮点求平方根。R 型，在 RV32F 和 RV64F 中。

计算 f[*rs1*] 中的单精度浮点数的平方根，将舍入后的单精度结果写入 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`

交换浮点舍入模式。伪指令，在 RV32F 和 RV64F 中。

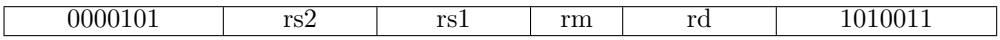
将 x[*rs1*] 写入浮点舍入模式寄存器，并将浮点舍入模式寄存器的原值写入 x[*rd*]。展开为 **csrrw** rd, frm, rs1。若省略 *rd*，则默认为 x0。

---

fsub.d rd, rs1, rs2 `f[rd] = f[rs1] - f[rs2]`

双精度浮点减。R 型，在 RV32D 和 RV64D 中。

将 f[*rs1*] 与 f[*rs2*] 中的双精度浮点数相减，将舍入后的双精度结果写入 f[*rd*]。



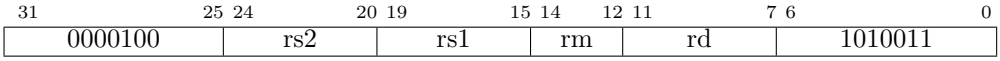
31–25	24–20	19–15	14–12	11–7	6–0
0000101	rs2	rs1	rm	rd	1010011

---

fsub.s rd, rs1, rs2 `f[rd] = f[rs1] - f[rs2]`

单精度浮点减。R 型，在 RV32F 和 RV64F 中。

将 f[*rs1*] 与 f[*rs2*] 中的单精度浮点数相减，将舍入后的单精度结果写入 f[*rd*]。



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

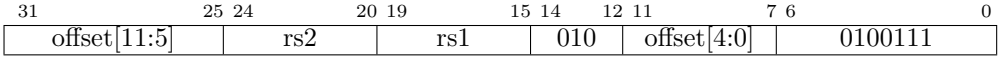
---

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

存浮点字。S 型，在 RV32F 和 RV64F 中。

将 f[*rs2*] 中的单精度浮点数写入内存地址 x[*rs1*] + *sign-extend*(*offset*) 中。

压缩形式：**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)`

跳转。伪指令，在 RV32I 和 RV64I 中。

将 *pc* 设为当前 *pc* 加上符号扩展后的 *offset*。展开为 **jal** x0, offset。

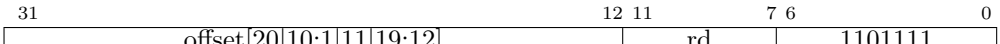
---

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

跳转并链接。J 型，在 RV32I 和 RV64I 中。

将下一条指令的地址（*pc*+4）写入 x[*rd*]，然后将 *pc* 设为当前 *pc* 加上符号扩展后的 *offset*。若省略 *rd*，则默认为 x1。

压缩形式：**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`

寄存器跳转并链接。I 型，在 RV32I 和 RV64I 中。

将 *pc* 设为 x[*rs1*] + *sign-extend*(*offset*)，将跳转地址的最低位清零，并将原 *pc*+4 写入 x[*rd*]。若省略 *rd*，则默认为 x1。

压缩形式：**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]`

寄存器跳转。伪指令，在 RV32I 和 RV64I 中。

将 *pc* 设为 x[*rs1*]。展开为 **jalr** x0, 0(rs1)。

---

la rd, symbol

`x[rd] = &symbol`

装入地址。伪指令，在 RV32I 和 RV64I 中。

将 *symbol* 的地址装入 x[*rd*]。当汇编位置无关代码时，它展开为对全局偏移量表（Global Offset Table）的读入操作：在 RV32I 中展开为 **auipc** rd, offsetHi 和 **lw** rd, offsetLo(rd)；在 RV64I 中则展开为 **auipc** rd, offsetHi 和 **ld** rd, offsetLo(rd)。否则，它展开为 **auipc** rd, offsetHi 和 **addi** rd, rd, offsetLo。

---

lb rd, offset(rs1)

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

取字节。I 型，在 RV32I 和 RV64I 中。

从地址 x[*rs1*] + *sign-extend*(*offset*) 读取 1 字节，符号扩展后写入 x[*rd*]。

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]`

取无符号字节。I 型，在 RV32I 和 RV64I 中。

从地址 x[*rs1*] + *sign-extend*(*offset*) 读取 1 字节，零扩展后写入 x[*rd*]。

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

---

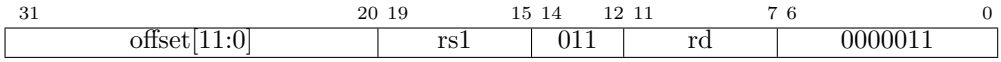
ld rd, offset(rs1)

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

取双字。I 型，仅在 RV64I 中。

从地址 x[*rs1*] + *sign-extend*(*offset*) 读取 8 字节，写入 x[*rd*]。

压缩形式：**c.ldsp** rd, offset; **c.ld** rd, offset(rs1)



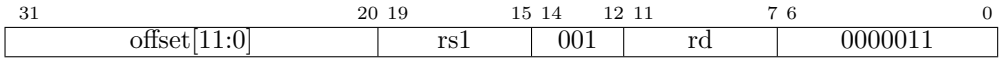
---

lh rd, offset(rs1)

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

取半字。I 型，在 RV32I 和 RV64I 中。

从地址 x[*rs1*] + *sign-extend*(*offset*) 读取 2 字节，符号扩展后写入 x[*rd*]。



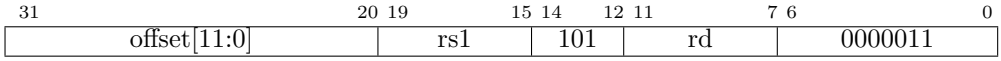
---

lhu rd, offset(rs1)

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

取无符号半字。I 型，在 RV32I 和 RV64I 中。

从地址 x[*rs1*] + *sign-extend*(*offset*) 读取 2 字节，零扩展后写入 x[*rd*]。



---

li rd, immediate

`x[rd] = immediate`

装入立即数。伪指令，在 RV32I 和 RV64I 中。

使用尽可能少的指令将常量装入 x[*rd*]。在 RV32I 中，它展开为 **lui** 和/或 **addi**；在 RV64I 中的展开结果较长：**lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**。

---

lla rd, symbol

`x[rd] = &symbol`

装入本地地址。伪指令，在 RV32I 和 RV64I 中。

将 *symbol* 的地址装入 x[*rd*]。展开为 **auipc** rd, offsetHi 和 **addi** rd, rd, offsetLo。

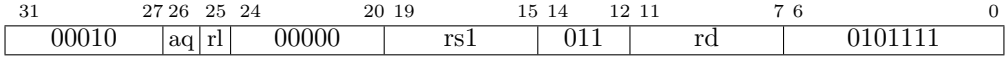
---

lr.d rd, (rs1)

`x[rd] = LoadReserved64(M[x[rs1]])`

预订取双字。R 型，仅在 RV64A 中。

从地址 x[*rs1*] 读取 8 字节，写入 x[*rd*]，并预订该内存双字。



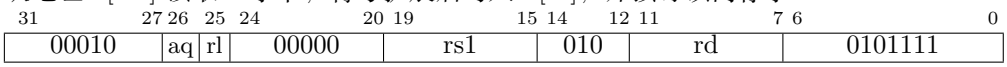
---

lr.w rd, (rs1)

`x[rd] = LoadReserved32(M[x[rs1]])`

预订取字。R 型，在 RV32A 和 RV64A 中。

从地址 x[*rs1*] 读取 4 字节，符号扩展后写入 x[*rd*]，并预订该内存字。



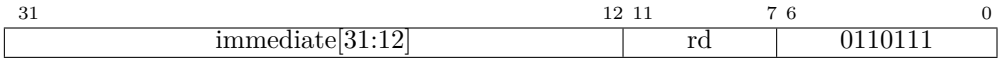
lui rd, immediate

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

装入高位立即数。U 型，在 RV32I 和 RV64I 中。

将 20 位 *immediate* 符号扩展后左移 12 位，并将低 12 位置零，结果写入 x[*rd*]。

压缩形式：**c.lui** rd, imm



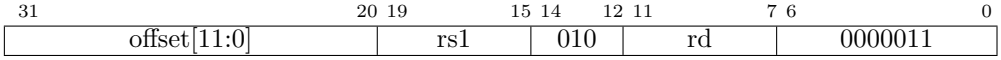
lw rd, offset(rs1)

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

取字。I 型，在 RV32I 和 RV64I 中。

从地址 x[*rs1*] + *sign-extend*(*offset*) 读取 4 字节，写入 x[*rd*]。在 RV64I 中，结果要进行符号扩展。

压缩形式：**c.lwsp** rd, offset; **c.lw** rd, offset(rs1)

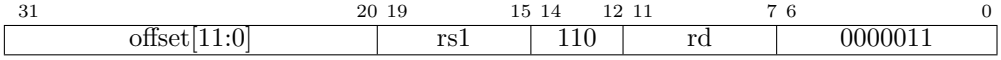


lwu rd, offset(rs1)

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

取无符号字。I 型，仅在 RV64I 中。

从地址 x[*rs1*] + *sign-extend*(*offset*) 读取 4 字节，零扩展后写入 x[*rd*]。

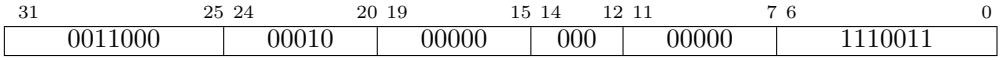


mret

`ExceptionReturn(Machine)`

机器模式异常返回。R 型，在 RV32I 和 RV64I 的特权架构中。

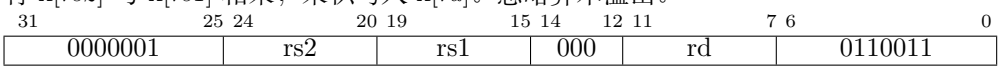
从机器模式的异常处理程序返回。将 *pc* 设为 CSRs[mepc]，将特权级设为 CSRs[mstatus].MPP，CSRs[mstatus].MIE 设为 CSRs[mstatus].MPIE，并将 CSRs[mstatus].MPIE 设为 1。若支持用户模式，则将 CSR[mstatus].MPP 设为 0。



mul rd, rs1, rs2 `x[rd] = x[rs1] × x[rs2]`

乘。R 型，在 RV32M 和 RV64M 中。

将 x[*rs2*] 与 x[*rs1*] 相乘，乘积写入 x[*rd*]。忽略算术溢出。



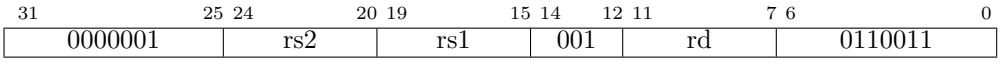
31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	000	rd	0110011

---

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

高位乘。R 型，在 RV32M 和 RV64M 中。

将 x[*rs2*] 与 x[*rs1*] 视为补码并相乘，乘积的高位写入 x[*rd*]。



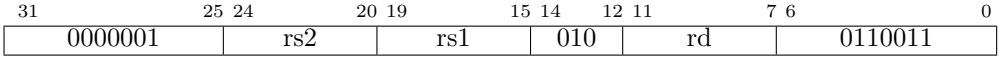
31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	001	rd	0110011

---

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

高位有符号-无符号乘。R 型，在 RV32M 和 RV64M 中。

将 x[*rs1*]（视为补码）与 x[*rs1*]（视为无符号数）相乘，乘积的高位写入 x[*rd*]。



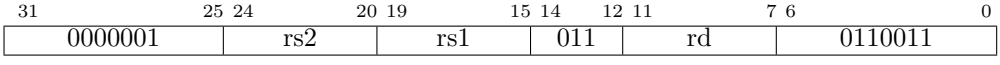
31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	010	rd	0110011

---

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

高位无符号乘。R 型，在 RV32M 和 RV64M 中。

将 x[*rs2*] 与 x[*rs1*] 视为无符号数并相乘，乘积的高位写入 x[*rd*]。



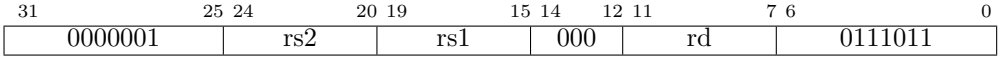
31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	011	rd	0110011

---

mulw rd, rs1, rs2 `x[rd] = sext((x[rs1] × x[rs2])[31:0])`

乘字。R 型，仅在 RV64M 中。

将 x[*rs2*] 与 x[*rs1*] 相乘，乘积截为 32 位，符号扩展后写入 x[*rd*]。忽略算术溢出。



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]`

数据传送。伪指令，在 RV32I 和 RV64I 中。

将 x[*rs1*] 复制到 x[*rd*] 中。展开为 **addi** rd, rs1, 0。

---

neg rd, rs2

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

取负。伪指令，在 RV32I 和 RV64I 中。

将 x[*rs2*] 的相反数写入 x[*rd*]。展开为 **sub** rd, x0, rs2。

---

negw rd, rs2

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

取负字。伪指令，仅在 RV64I 中。

将 x[*rs2*] 的相反数截为 32 位，符号扩展后写入 x[*rd*]。展开为 **subw** rd, x0, rs2。

---

nop

Nothing

空操作。伪指令，在 RV32I 和 RV64I 中。

仅让 *pc* 指向下一条指令。展开为 **addi** x0, x0, 0。

---

not rd, rs1

`x[rd] = ∼x[rs1]`

取反。伪指令，在 RV32I 和 RV64I 中。

将 x[*rs1*] 按位取反后写入 x[*rd*]。展开为 **xori** rd, rs1, -1。

---

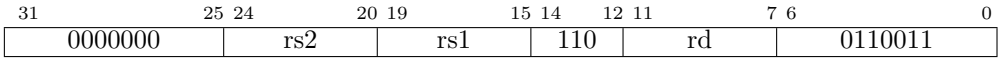
or rd, rs1, rs2

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

或。R 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 和 x[*rs2*] 按位或的结果写入 x[*rd*]。

压缩形式：**c.or** rd, rs2



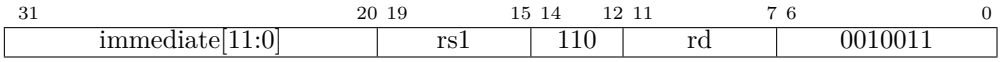
---

ori rd, rs1, immediate

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

或立即数。I 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 和符号扩展后的 *immediate* 按位或的结果写入 x[*rd*]。



---

rdcycle rd

`x[rd] = CSRs[cycle]`

读周期计数器。伪指令，在 RV32I 和 RV64I 中。

将经过的周期数写入 x[*rd*]。展开为 **csrrs** rd, cycle, x0。

---

rdcycleh rd

`x[rd] = CSRs[cycleh]`

读周期计数器高位。伪指令，仅在 RV32I 中。

将经过的周期数右移 32 位后写入 x[*rd*]。展开为 **csrrs** rd, cycleh, x0。

---

rdinstret rd

`x[rd] = CSRs[instret]`

读已提交指令计数器。伪指令，在 RV32I 和 RV64I 中。

将已提交指令数写入 x[*rd*]。展开为 **csrrs** rd, instret, x0。

---

rdinstreth rd

`x[rd] = CSRs[instreth]`

读已提交指令计数器高位。伪指令，仅在 RV32I 中。

将已提交指令数右移 32 位后写入 x[*rd*]。展开为 **csrrs** rd, instreth, x0。

---

rdtime rd

`x[rd] = CSRs[time]`

读时间。伪指令，在 RV32I 和 RV64I 中。

将当前时间写入 x[*rd*]，时钟频率与平台相关。展开为 **csrrs** rd, time, x0。

---

rdtimeh rd

`x[rd] = CSRs[timeh]`

读时间高位。伪指令，仅在 RV32I 中。

将当前时间右移 32 位后写入 x[*rd*]，时间频率与平台相关。展开为 **csrrs** rd, timeh, x0。

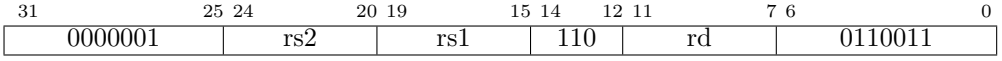
---

rem rd, rs1, rs2

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

求余数。R 型，在 RV32M 和 RV64M 中。

将 x[*rs1*] 和 x[*rs2*] 视为补码并相除，向 0 舍入，将余数写入 x[*rd*]。



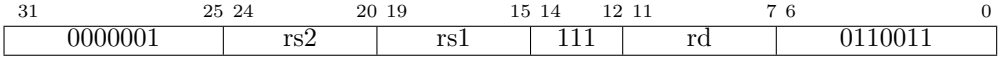
---

remu rd, rs1, rs2

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

求无符号余数。R 型，在 RV32M 和 RV64M 中。

将 x[*rs1*] 和 x[*rs2*] 视为无符号数并相除，向 0 舍入，将余数写入 x[*rd*]。

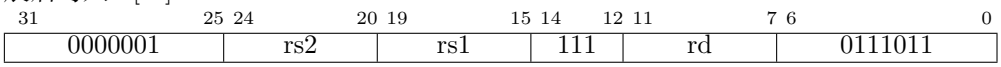


---

remuw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] %u x[rs2][31:0])`

求无符号余数字。R 型，仅在 RV64M 中。

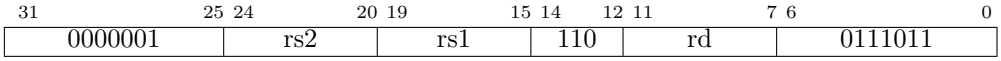
将 x[*rs1*] 和 x[*rs2*] 的低 32 位视为无符号数并相除，向 0 舍入，将 32 位余数符号扩展后写入 x[*rd*]。



remw rd, rs1, rs2 `x[rd] = sext(x[rs1][31:0] %s x[rs2][31:0])`

求余数字。R 型，仅在 RV64M 中。

将 x[*rs1*] 和 x[*rs2*] 的低 32 位视为补码并相除，向 0 舍入，将 32 位余数符号扩展后写入 x[*rd*]。



ret `pc = x[1]`

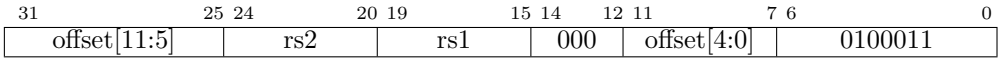
返回。伪指令，在 RV32I 和 RV64I 中。

从子过程返回。展开为 **jalr** x0, 0(x1)。

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

存字节。S 型，在 RV32I 和 RV64I 中。

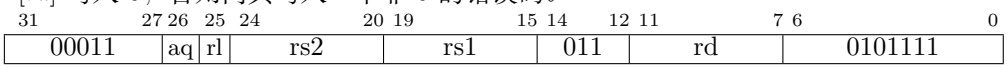
将 x[*rs2*] 的最低字节写入内存地址 x[*rs1*]+*sign-extend*(*offset*)。



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

条件存双字。R 型，仅在 RV64A 中。

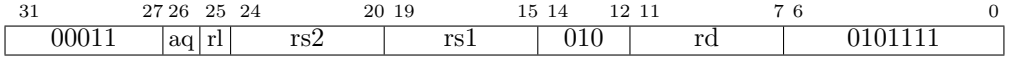
若内存地址 x[*rs1*] 被预订，则将 x[*rs2*] 中的 8 字节写入该地址。若写入成功，则向 x[*rd*] 写入 0，否则向其写入一个非 0 的错误码。



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

条件存字。R 型，在 RV32A 和 RV64A 中。

若内存地址 x[*rs1*] 被预订，则将 x[*rs2*] 中的 4 字节写入该地址。若写入成功，则向 x[*rd*] 写入 0，否则向其写入一个非 0 的错误码。



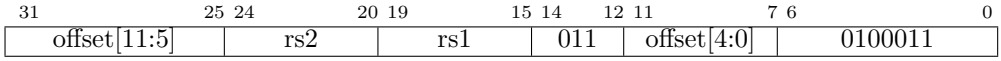
---

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

存双字。S 型，仅在 RV64I 中。

将 x[*rs2*] 中的 8 字节写入内存地址 x[*rs1*]+*sign-extend*(*offset*)。

压缩形式: **c.sdsp** rs2, offset; **c.sd** rs2, offset(rs1)



---

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

等于零时置位。伪指令，在 RV32I 和 RV64I 中。

若 x[*rs1*] 等于 0，则向 x[*rd*] 写入 1，否则写入 0。展开为 **sltiu** rd, rs1, 1。

---

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

符号扩展字。伪指令，仅在 RV64I 中。

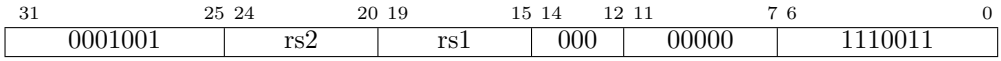
将 x[*rs1*] 低 32 位的符号扩展结果写入 x[*rd*]。展开为 **addiw** rd, rs1, 0。

---

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

虚拟内存屏障。R 型，在 RV32I 和 RV64I 的特权架构中。

将前驱的页表写入操作与后续虚拟地址翻译过程定序。当 *rs2*=0 时，将影响所有地址空间的翻译；否则，仅标识为 x[*rs2*] 的地址空间的翻译需要定序。当 *rs1*=0 时，指定地址空间中的所有虚拟地址的翻译都需要定序；否则仅其中包含虚拟地址 x[*rs1*] 的页面的翻译需要定序。



---

sgtz rd, rs2 `x[rd] = (x[rs2] >s 0)`

大于零则置位。伪指令，在 RV32I 和 RV64I 中。

若 x[*rs2*] 大于 0，则向 x[*rd*] 写入 1，否则写入 0。展开为 **slt** rd, x0, rs2。

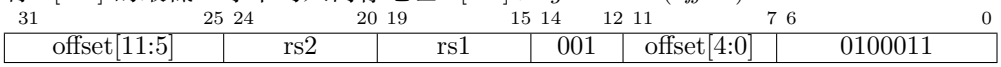
---

sh rs2, offset(rs1)

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

存半字。S 型，在 RV32I 和 RV64I 中。

将 x[*rs2*] 的最低 2 字节写入内存地址 x[*rs1*]+*sign-extend*(*offset*)。

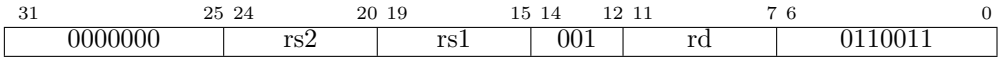


sll rd, rs1, rs2

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

逻辑左移。R 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 左移 x[*rs2*] 位，空位补零，结果写入 x[*rd*]。x[*rs2*] 的低 5 位（在 RV64I 中是低 6 位）为移位位数，高位忽略。



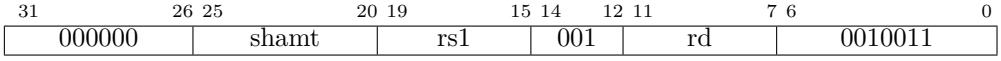
slli rd, rs1, shamt

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

逻辑左移立即数。I 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 左移 *shamt* 位，空位补零，结果写入 x[*rd*]。在 RV32I 中，仅当 *shamt*[5]=0 时该指令合法。

压缩形式：**c.slli** rd, shamt

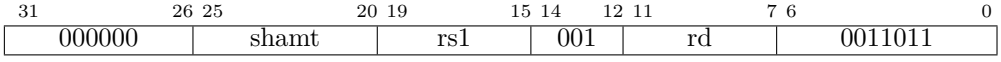


slliw rd, rs1, shamt

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

逻辑左移立即数字。I 型，仅在 RV64I 中。

将 x[*rs1*] 左移 *shamt* 位，空位补零，结果截为 32 位，符号扩展后写入 x[*rd*]。仅当 *shamt*[5]=0 时该指令合法。

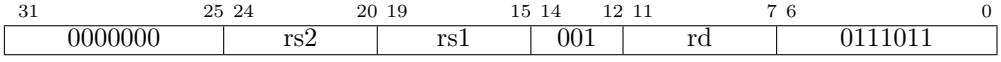


sllw rd, rs1, rs2

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

逻辑左移字。R 型，仅在 RV64I 中。

将 x[*rs1*] 的低 32 位左移 x[*rs2*] 位，空位补零，符号扩展后写入 x[*rd*]。x[*rs2*] 的低 5 位为移位位数，高位忽略。



slt rd, rs1, rs2

`x[rd] = x[rs1] <`$_s$ `x[rs2]`

小于则置位。R 型，在 RV32I 和 RV64I 中。

比较 x[*rs1*] 和 x[*rs2*]（视为补码），若 x[*rs1*] 更小，则向 x[*rd*] 写入 1，否则写入 0。

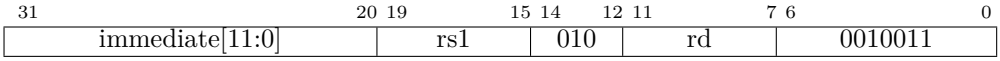
---

slti rd, rs1, immediate

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

小于立即数则置位。I 型，在 RV32I 和 RV64I 中。

比较 x[*rs1*] 和符号扩展后的 *immediate*（视为补码），若 x[*rs1*] 更小，则向 x[*rd*] 写入 1，否则写入 0。



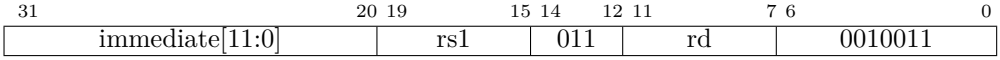
---

sltiu rd, rs1, immediate

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

无符号小于立即数则置位。I 型，在 RV32I 和 RV64I 中。

比较 x[*rs1*] 和符号扩展后的 *immediate*（视为无符号数），若 x[*rs1*] 更小，则向 x[*rd*] 写入 1，否则写入 0。



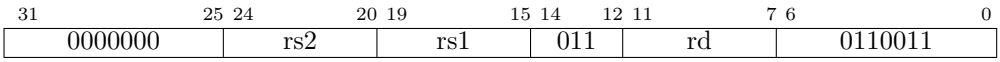
---

sltu rd, rs1, rs2

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

无符号小于则置位。R 型，在 RV32I 和 RV64I 中。

比较 x[*rs1*] 和 x[*rs2*]（视为无符号数），若 x[*rs1*] 更小，则向 x[*rd*] 写入 1，否则写入 0。



---

sltz rd, rs1

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

小于零则置位。伪指令，在 RV32I 和 RV64I 中。

若 x[*rs1*] 小于 0，则向 x[*rd*] 写入 1，否则写入 0。展开为 **slt** rd, rs1, x0。

---

snez rd, rs2

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

不等于零则置位。伪指令，在 RV32I 和 RV64I 中。

若 x[*rs2*] 不等于 0，则向 x[*rd*] 写入 1，否则写入 0。展开为 **sltu** rd, x0, rs2。

---

sra rd, rs1, rs2

`x[rd] = x[rs1] >>s x[rs2]`

算术右移。R 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 右移 x[*rs2*] 位，空位用 x[*rs1*] 的最高位填充，结果写入 x[*rd*]。x[*rs2*] 的低 5 位（在 RV64I 中是低 6 位）为移位位数，高位忽略。

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`

算术右移立即数。I 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 右移 *shamt* 位，空位用 x[*rs1*] 的最高位填充，结果写入 x[*rd*]。在 RV32I 中，仅当 *shamt*[5]=0 时该指令合法。

压缩形式：**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)`

算术右移立即数字。I 型，仅在 RV64I 中。

将 x[*rs1*] 的低 32 位右移 *shamt* 位，空位用 x[*rs1*][31] 填充，将 32 位结果符号扩展后写入 x[*rd*]。仅当 *shamt*[5]=0 时该指令合法。

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])`

算术右移字。R 型，仅在 RV64I 中。

将 x[*rs1*] 的低 32 位右移 x[*rs2*] 位，空位用 x[*rs1*][31] 填充，将 32 位结果符号扩展后写入 x[*rd*]。x[*rs2*] 的低 5 位为移位位数，高位忽略。

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	101	rd	0111011

sret

`ExceptionReturn(Supervisor)`

监管模式异常返回。R 型，在 RV32I 和 RV64I 的特权架构中。

从监管模式的异常处理程序返回。将 *pc* 设为 CSRs[sepc]，将特权模式设为 CSRs[sstatus].SPP，将 CSRs[sstatus].SIE 设为 CSRs[sstatus].SPIE，将 CSRs[sstatus].SPIE 设为 1，将 CSRs[sstatus].SPP 设为 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]`

逻辑右移。R 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 右移 x[*rs2*] 位，空位补零，结果写入 x[*rd*]。x[*rs2*] 的低 5 位（在 RV64I 中是低 6 位）为移位位数，高位忽略。

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`

逻辑右移立即数。I 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 右移 *shamt* 位，空位补零，结果写入 x[*rd*]。在 RV32I 中，仅当 *shamt*[5]=0 时该指令合法。

压缩形式：**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)`

逻辑右移立即数字。I 型，仅在 RV64I 中。

将 x[*rs1*] 的低 32 位右移 *shamt* 位，空位补零，结果截为 32 位，符号扩展后写入 x[*rd*]。仅当 *shamt*[5]=0 时该指令合法。

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])`

逻辑右移字。R 型，仅在 RV64I 中。

将 x[*rs1*] 的低 32 位右移 x[*rs2*] 位，空位补零，符号扩展后写入 x[*rd*]。x[*rs2*] 的低 5 位为移位位数，高位忽略。

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]`

减。R 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 减去 x[*rs2*]，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**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])`

减字。R 型，仅在 RV64I 中。

将 x[*rs1*] 减去 x[*rs2*]，结果截为 32 位，符号扩展后写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.subw** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0100000	rs2	rs1	000	rd	0111011

---

sw rs2, offset(rs1)

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

存字。S 型，在 RV32I 和 RV64I 中。

将 x[*rs2*] 的最低 4 字节写入内存地址 x[*rs1*]+*sign-extend*(*offset*)。

压缩形式：**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

---

tail symbol

`pc = &symbol; clobber x[6]`

尾调用。伪指令，在 RV32I 和 RV64I 中。

将 *pc* 设为 *symbol*，x[6] 将被覆盖。展开为 **auipc** x6, offsetHi 和 **jalr** x0, offsetLo(x6)。

---

wfi

`while (noInterruptsPending) idle`

等待中断。R 型，在 RV32I 和 RV64I 的特权架构中。

若无中断请求，则使处理器空闲以节省能耗。

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]`

异或。R 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 和 x[*rs2*] 按位异或，结果写入 x[*rd*]。

压缩形式：**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)`

异或立即数。I 型，在 RV32I 和 RV64I 中。

将 x[*rs1*] 和符号扩展后的 *immediate* 按位异或，结果写入 x[*rd*]。