RISC-V Instruction Set Specifications

Table of Contents

7.11. fsgnjn.s

7.12. fsgnjx.s

7.13. fmin.s

7.14. fmax.s

7.15. fcvt.w.s

7.16. fcvt.wu.s

7.17. fmv.x.w

7.18. feq.s

7.19. flt.s

7.20. fle.s

7.21. fclass.s

7.22. fcvt.s.w

7.23. fcvt.s.wu

7.24. fmv.w.x

7.25. fmadd.d

7.26. fmsub.d

7.27. fnmsub.d

7.28. fnmadd.d

7.29. fadd.d

7.30. fsub.d

7.31. fmul.d

7.32. fdiv.d

7.33. fsqrt.d

7.34. fsgnj.d

7.35. fsgnjn.d

7.36. fsgnjx.d

7.37. fmin.d

7.38. fmax.d

7.39. fcvt.s.d

7.40. fcvt.d.s

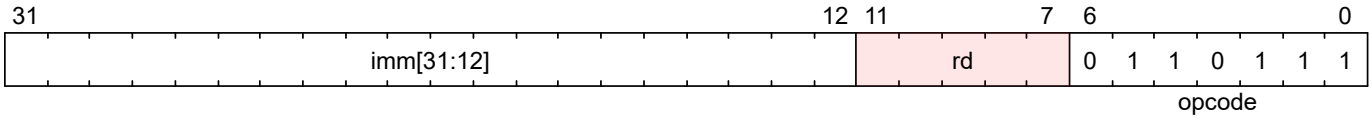
---

1. RV32I, RV64I Instructions

1.1. lui

load upper immediate.

Encoding



Format

```
lui rd,imm
```

Description

Build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.

Implementation

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

1.2. auipc

add upper immediate to pc

Encoding

Format

```
auipc rd,imm
```

Description

Build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.

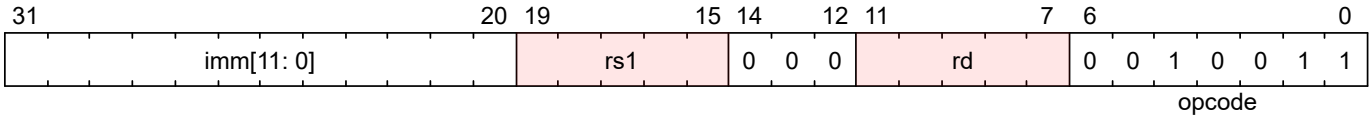
Implementation

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

1.3. addi

add immediate

Encoding



Format

```
addi rd,rs1,imm
```

Description

Adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.

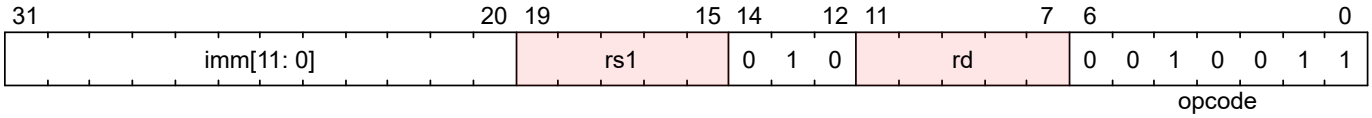
Implementation

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

1.4. slti

set less than immediate

Encoding



Format

```
slti rd,rs1,imm
```

Description

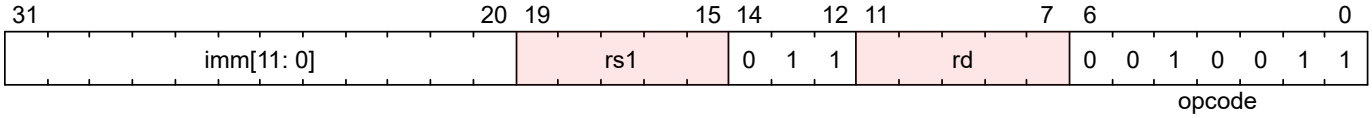
Place the value 1 in register rd if register rs1 is less than the signextended immediate when both are treated as signed numbers, else 0 is written to rd.

Implementation

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

1.5. sltiu

Encoding



Format

```
sltiu rd,rs1,imm
```

Description

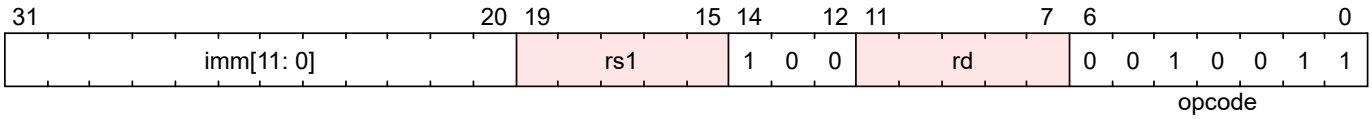
Place the value 1 in register rd if register rs1 is less than the immediate when both are treated as unsigned numbers, else 0 is written to rd.

Implementation

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

1.6. xori

Encoding



Format

```
xori rd,rs1,imm
```

Description

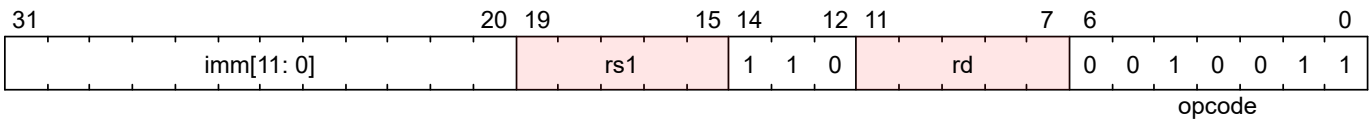
Performs bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd Note, "XORI rd, rs1, -1" performs a bitwise logical inversion of register rs1(assembler pseudo-instruction NOT rd, rs)

Implementation

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

1.7. ori

Encoding



Format

```
ori rd,rs1,imm
```

Description

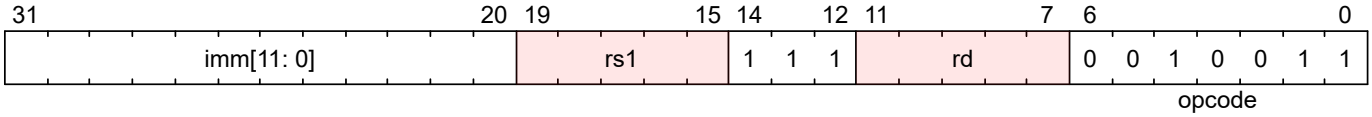
Performs bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd

Implementation

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

1.8. andi

Encoding



Format

```
andi rd,rs1,imm
```

Description

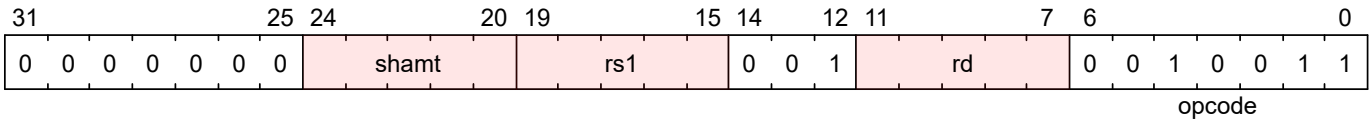
Performs bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd

Implementation

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

1.9. slli

Encoding



Format

```
slli rd,rs1,shamt
```

Description

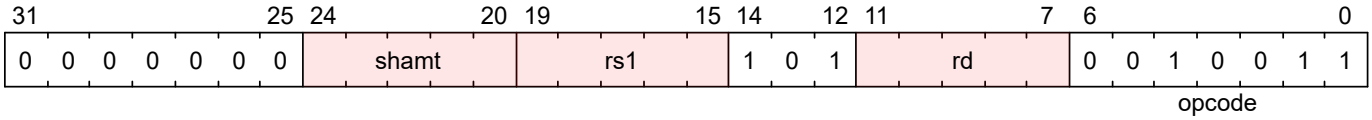
Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate In RV64, bit-25 is used to shamt[5].

Implementation

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

1.10. srli

Encoding



Format

```
srli rd,rs1,shamt
```

Description

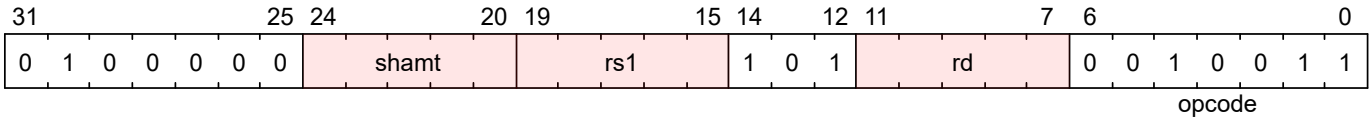
Performs logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate In RV64, bit-25 is used to shamt[5].

Implementation

```
x[rd] = x[rs1] >>u shamt
```

1.11. srai

Encoding



Format

```
srai rd,rs1,shamt
```

Description

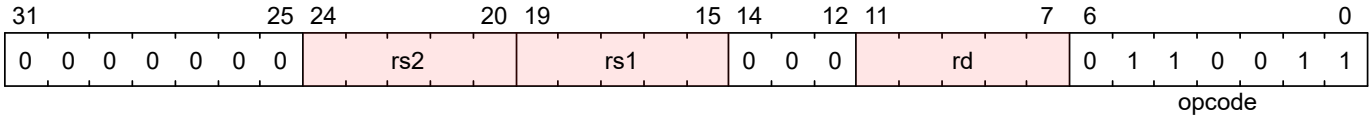
Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate In RV64, bit-25 is used to shamt[5].

Implementation

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

1.12. add

Encoding



Format

```
add rd,rs1,rs2
```

Description

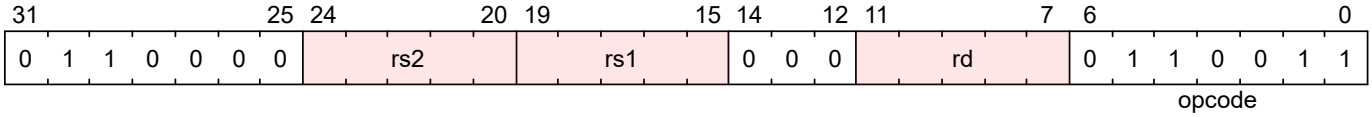
Adds the registers rs1 and rs2 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

Implementation

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

1.13. sub

Encoding



Format

```
sub rd,rs1,rs2
```

Description

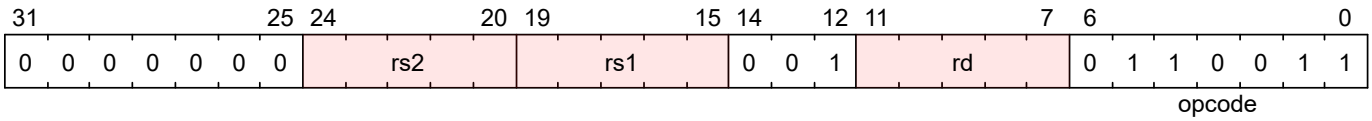
Subs the register rs2 from rs1 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

Implementation

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

1.14. sll

Encoding



Format

```
sll rd,rs1,rs2
```

Description

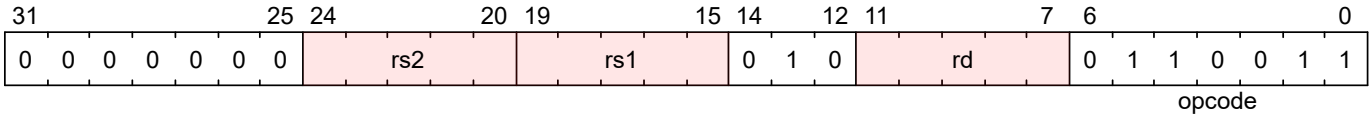
Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

Implementation

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

1.15. slt

Encoding



Format

```
slt rd,rs1,rs2
```

Description

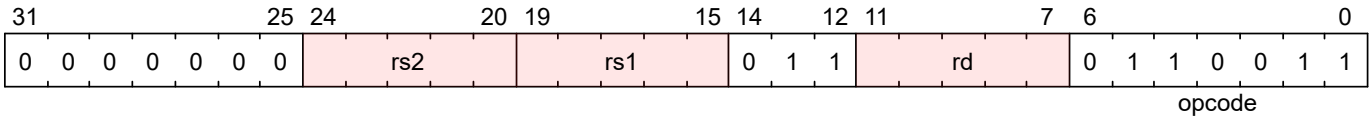
Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as signed numbers, else 0 is written to rd.

Implementation

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

1.16. sltu

Encoding



Format

```
sltu rd,rs1,rs2
```

Description

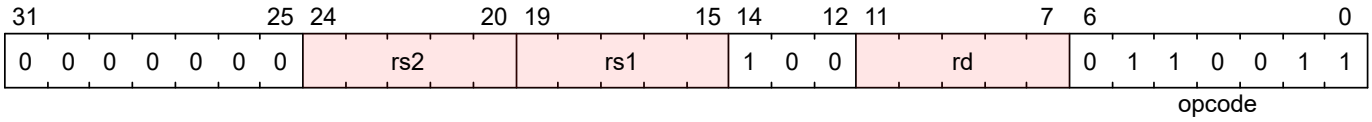
Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as unsigned numbers, else 0 is written to rd.

Implementation

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

1.17. xor

Encoding



Format

```
xor rd,rs1,rs2
```

Description

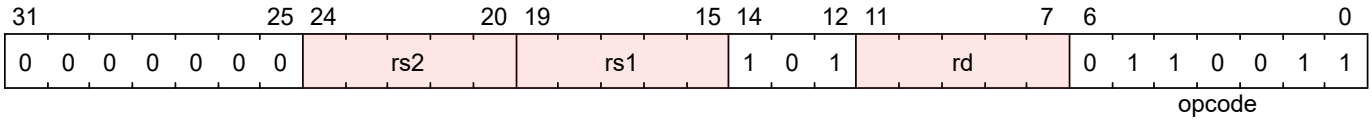
Performs bitwise XOR on registers rs1 and rs2 and place the result in rd

Implementation

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

1.18. srl

Encoding



Format

```
srl rd,rs1,rs2
```

Description

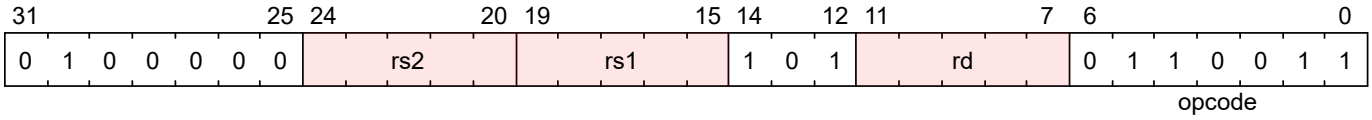
Logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2

Implementation

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

1.19. sra

Encoding



Format

```
sra rd,rs1,rs2
```

Description

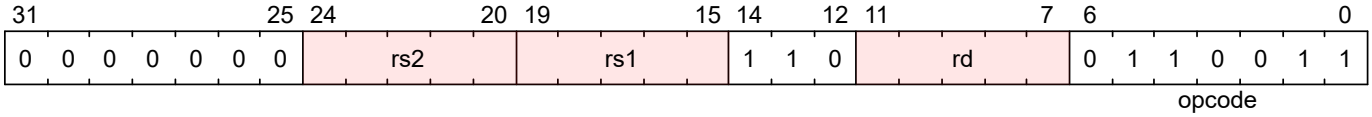
Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2

Implementation

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

1.20. or

Encoding



Format

```
or rd,rs1,rs2
```

Description

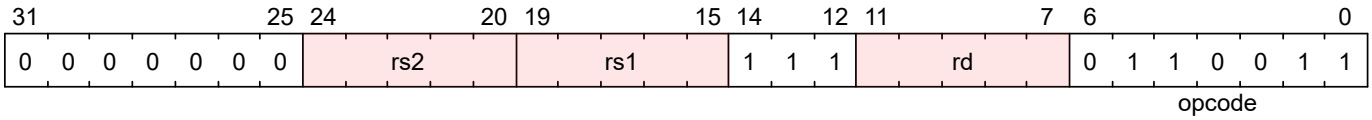
Performs bitwise OR on registers rs1 and rs2 and place the result in rd

Implementation

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

1.21. and

Encoding



Format

```
and rd,rs1,rs2
```

Description

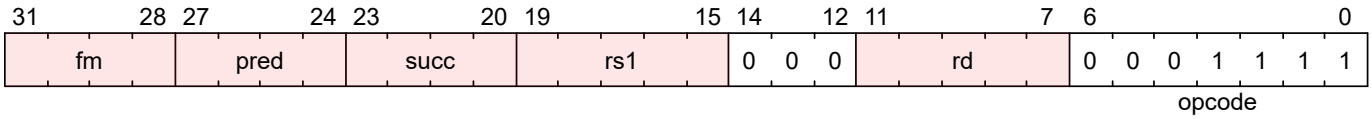
Performs bitwise AND on registers rs1 and rs2 and place the result in rd

Implementation

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

1.22. fence

Encoding



Format

```
fence pred, succ
```

Description

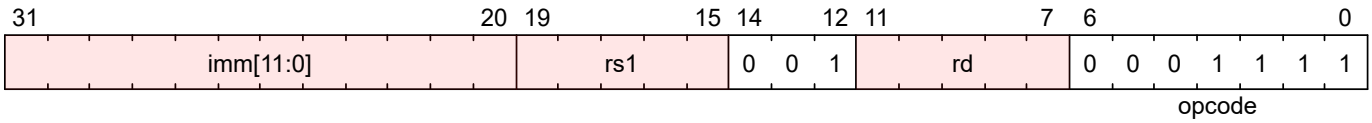
Used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads ®, and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the successor set following a FENCE before any operation in the predecessor set preceding the FENCE.

Implementation

```
Fence(pred, succ)
```

1.23. fence.i

Encoding



Format

```
fence.i
```

Description

Provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart.

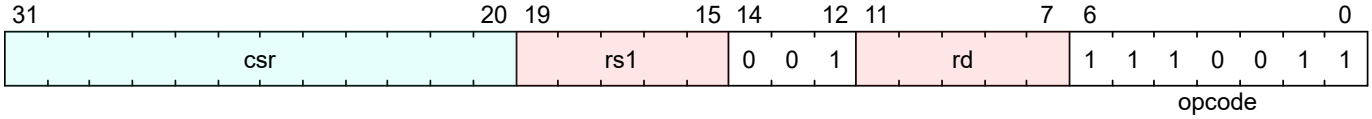
Implementation

```
Fence(Store, Fetch)
```

1.24. csrrw

atomic read/write CSR.

Encoding



Format

```
csrrw rd,offset,rs1
```

Description

Atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register rd. The initial value in rs1 is written to the CSR. If rd=x0, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read.

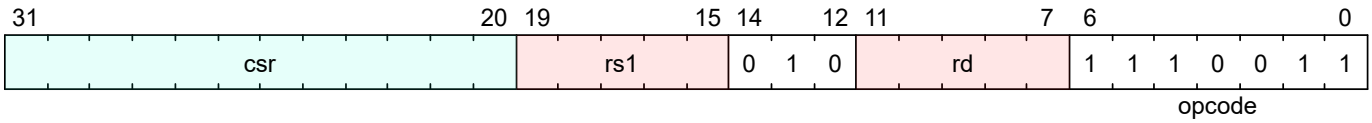
Implementation

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

1.25. csrrs

atomic read and set bits in CSR.

Encoding



Format

```
csrrs rd,offset,rs1
```

Description

Reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if

that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).

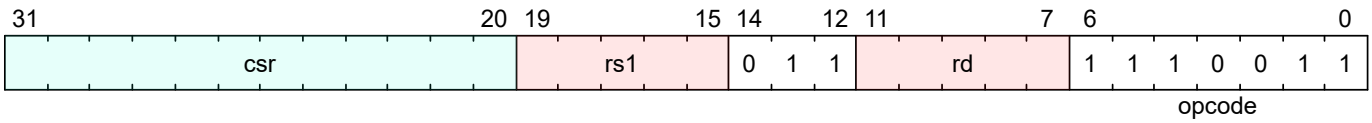
Implementation

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

1.26. csrrc

atomic read and clear bits in CSR.

Encoding



Format

```
csrrc rd,offset,rs1
```

Description

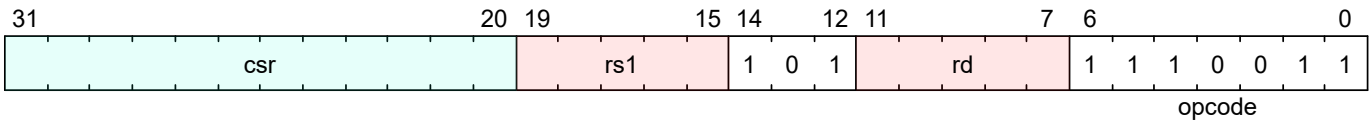
Reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

Implementation

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

1.27. csrrwi

Encoding



Format

```
csrrwi rd,offset,uimm
```

Description

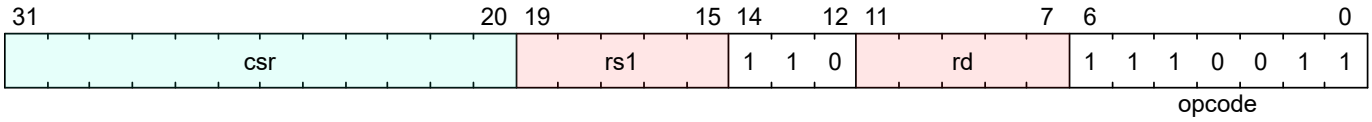
Update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.

Implementation

```
x[rd] = CSRs[csr]; CSRs[csr] = zimm
```

1.28. csrrsi

Encoding



Format

```
csrrsi rd,offset,uimm
```

Description

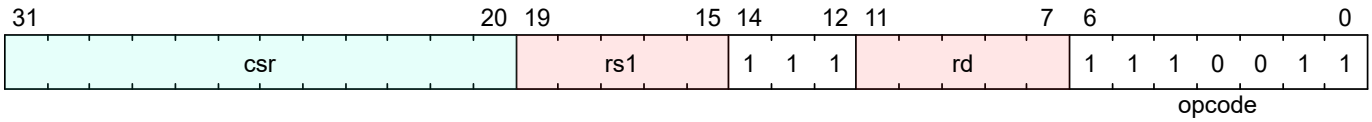
Set CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.

Implementation

```
t = CSRs[csr]; CSRs[csr] = t | zimm; x[rd] = t
```

1.29. csrrci

Encoding



Format

```
csrrci rd,offset,uimm
```

Description

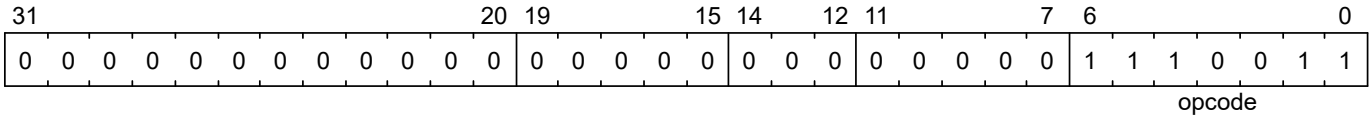
Clear CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.

Implementation

```
t = CSRs[csr]; CSRs[csr] = t &~zimm; x[rd] = t
```

1.30. ecall

Encoding



Format

```
ecall
```

Description

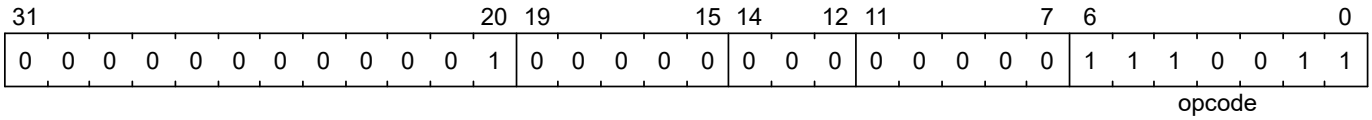
Make a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception, environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation.

Implementation

```
RaiseException(EnvironmentCall)
```

1.31. ebreak

Encoding



Format

```
ebreak
```

Description

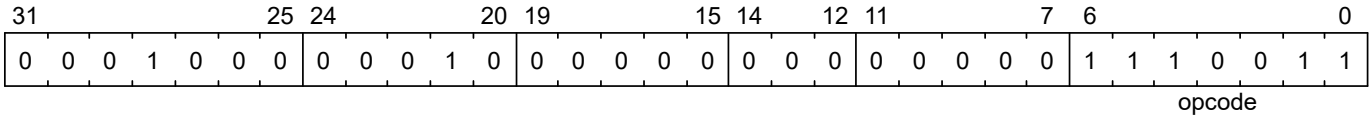
Used by debuggers to cause control to be transferred back to a debugging environment. It generates a breakpoint exception and performs no other operation.

Implementation

```
RaiseException(Breakpoint)
```

1.32. sret

Encoding



Format

```
sret
```

Description

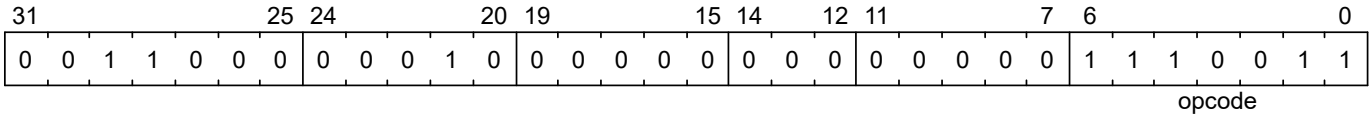
Return from traps in S-mode, and SRET copies SPIE into SIE, then sets SPIE.

Implementation

```
ExceptionReturn(User)
```

1.33. mret

Encoding



Format

```
mret
```

Description

Return from traps in M-mode, and MRET copies MPIE into MIE, then sets MPIE.

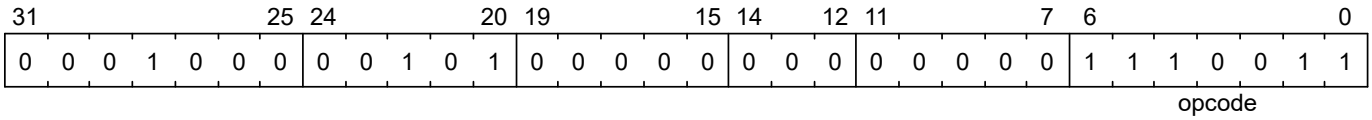
Implementation

```
ExceptionReturn(Machine)
```

1.34. wfi

wait for interrupt.

Encoding



Format

```
wfi
```

Description

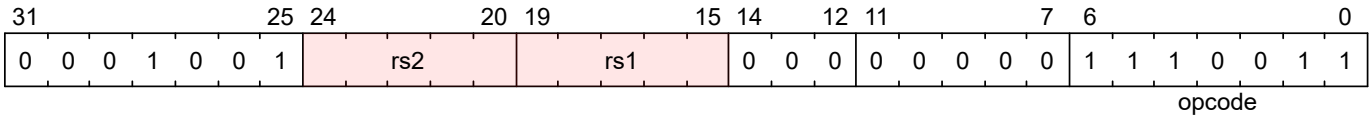
Provides a hint to the implementation that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart. WFI is available in all privileged modes, and optionally available to U-mode. This instruction may raise an illegal instruction exception when TW=1 in mstatus.

Implementation

```
while (noInterruptsPending) idle
```

1.35. sfence.vma

Encoding



Format

```
sfence.vma rs1,rs2
```

Description

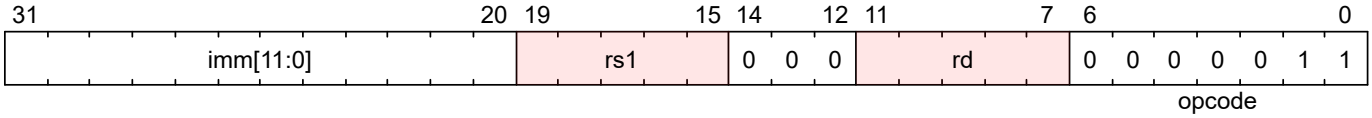
Guarantees that any previous stores already visible to the current RISC-V hart are ordered before all subsequent implicit references from that hart to the memory-management data structures. The SFENCE.VMA is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. SFENCE.VMA is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

Implementation

```
Fence(Store, AddressTranslation)
```

1.36. lb

Encoding



Format

```
lb rd,offset(rs1)
```

Description

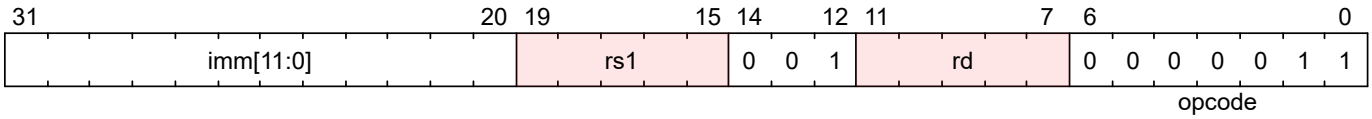
Loads a 8-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.37. lh

Encoding



Format

```
lh rd,offset(rs1)
```

Description

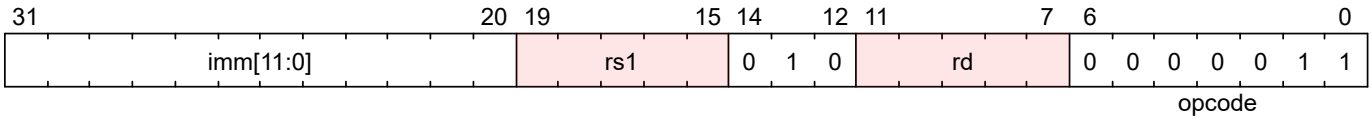
Loads a 16-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.38. lw

Encoding



Format

```
lw rd,offset(rs1)
```

Description

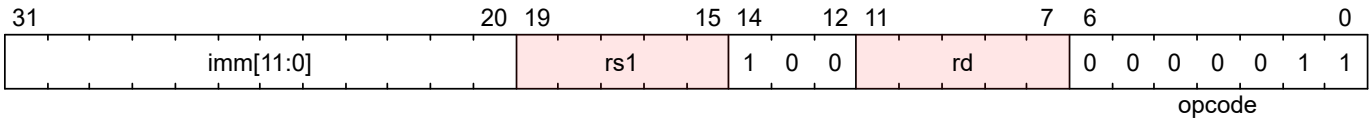
Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.39. lbu

Encoding



Format

```
lbu rd,offset(rs1)
```

Description

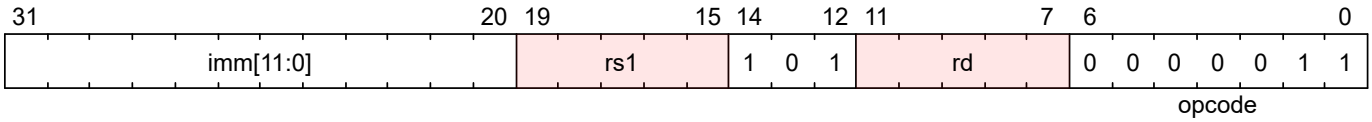
Loads a 8-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.40. lhu

Encoding



Format

```
lhu rd,offset(rs1)
```

Description

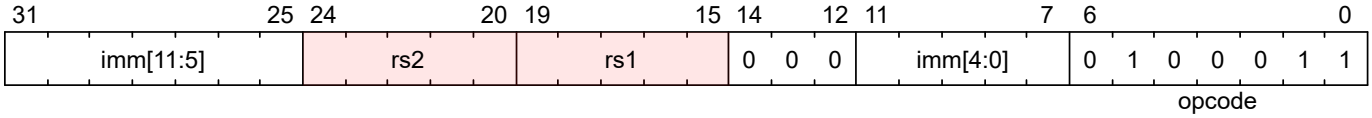
Loads a 16-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.41. sb

Encoding



Format

```
sb rs2,offset(rs1)
```

Description

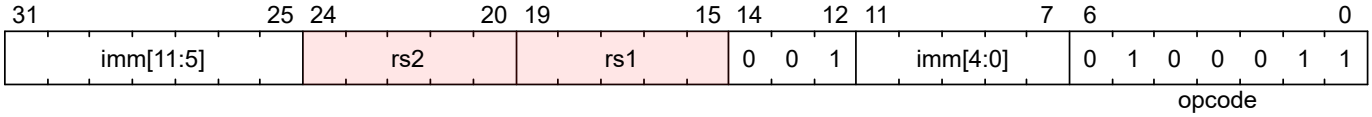
Store 8-bit, values from the low bits of register rs2 to memory.

Implementation

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

1.42. sh

Encoding



Format

```
sh rs2,offset(rs1)
```

Description

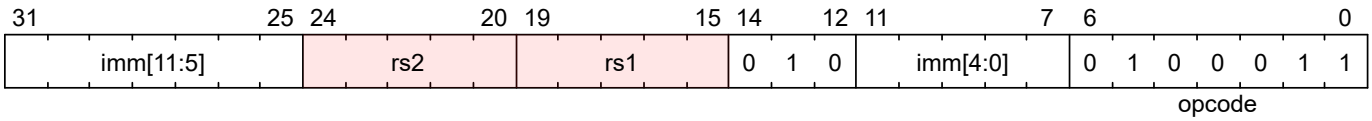
Store 16-bit, values from the low bits of register rs2 to memory.

Implementation

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

1.43. sw

Encoding



Format

```
sw rs2,offset(rs1)
```

Description

Store 32-bit, values from the low bits of register rs2 to memory.

Implementation

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

1.44. jal

Encoding

Format

```
jal rd,offset
```

Description

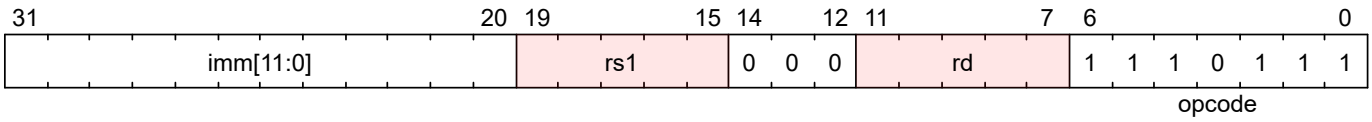
Jump to address and place return address in rd.

Implementation

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

1.45. jalr

Encoding



Format

```
jalr rd,rs1,offset
```

Description

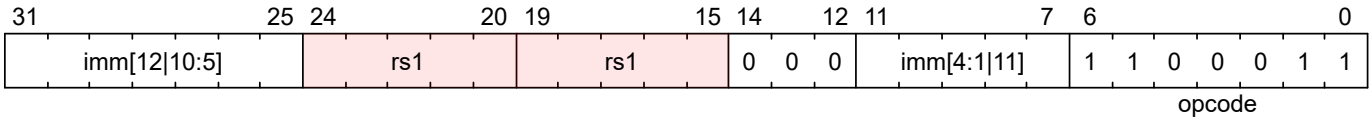
Jump to address and place return address in rd.

Implementation

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

1.46. beq

Encoding



Format

```
beq rs1,rs2,offset
```

Description

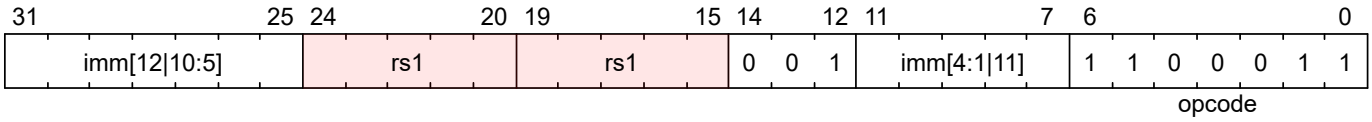
Take the branch if registers rs1 and rs2 are equal.

Implementation

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

1.47. bne

Encoding



Format

```
bne rs1,rs2,offset
```

Description

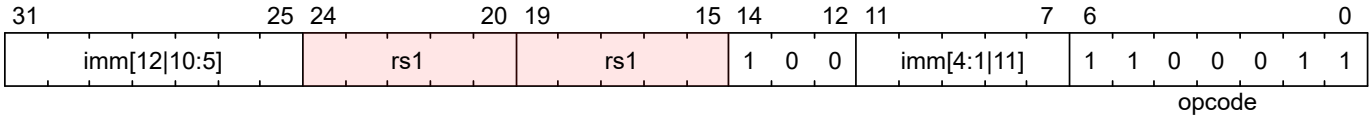
Take the branch if registers rs1 and rs2 are not equal.

Implementation

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

1.48. blt

Encoding



Format

```
blt rs1,rs2,offset
```

Description

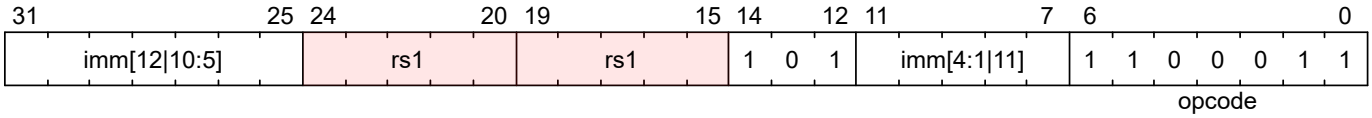
Take the branch if registers rs1 is less than rs2, using signed comparison.

Implementation

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

1.49. bge

Encoding



Format

```
bge rs1,rs2,offset
```

Description

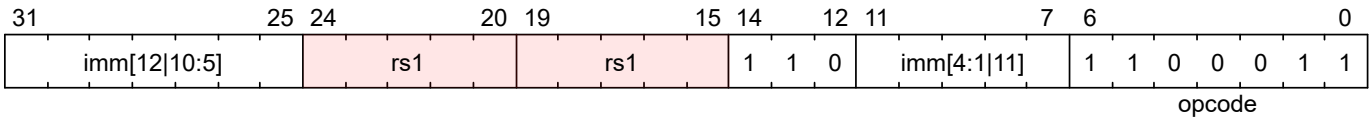
Take the branch if registers rs1 is greater than rs2, using signed comparison.

Implementation

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

1.50. bltu

Encoding



Format

```
bltu rs1,rs2,offset
```

Description

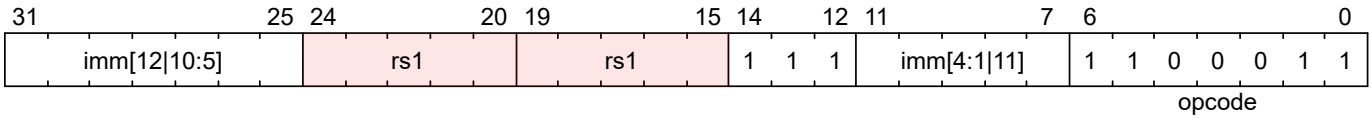
Take the branch if registers rs1 is less than rs2, using unsigned comparison.

Implementation

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

1.51. bgeu

Encoding



Format

```
bgeu rs1,rs2,offset
```

Description

Take the branch if registers rs1 is greater than rs2, using unsigned comparison.

Implementation

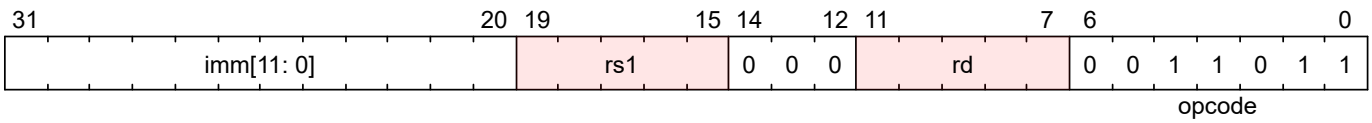
```
if (rs1 >=u rs2) pc += sext(offset)
```

---

2. RV64I Instructions

2.1. addiw

Encoding



Format

```
addiw rd,rs1,imm
```

Description

Adds the sign-extended 12-bit immediate to register rs1 and produces the proper sign-extension of a 32-bit result in rd. Overflows are ignored and the result is the low 32 bits of the result sign-

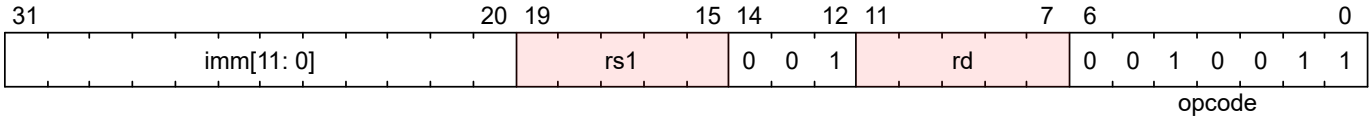
extended to 64 bits. Note, ADDIW rd, rs1, 0 writes the sign-extension of the lower 32 bits of register rs1 into register rd (assembler pseudoinstruction SEXT.W).

Implementation

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

2.2. slliw

Encoding



Format

```
slliw rd,rs1,shamt
```

Description

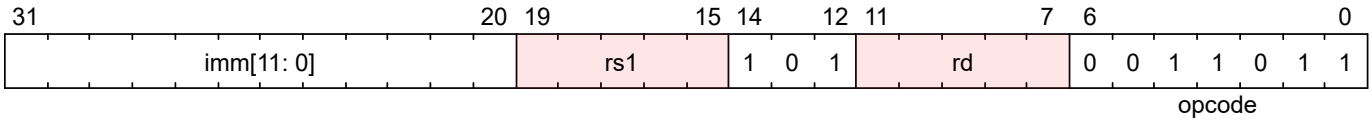
Performs logical left shift on the 32-bit of value in register rs1 by the shift amount held in the lower 5 bits of the immediate. Encodings with $imm[5] neq 0$ are reserved.

Implementation

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

2.3. srliw

Encoding



Format

```
srliw rd,rs1,shamt
```

Description

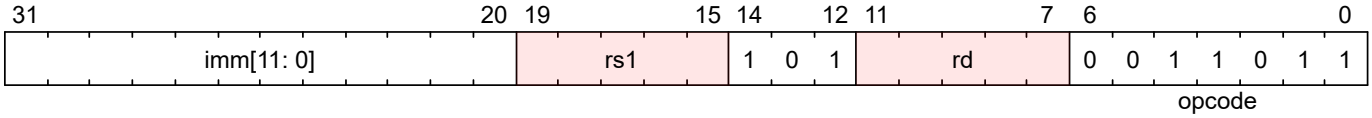
Performs logical right shift on the 32-bit of value in register rs1 by the shift amount held in the lower 5 bits of the immediate. Encodings with $imm[5] neq 0$ are reserved.

Implementation

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

2.4. sraiw

Encoding



Format

```
sraiw rd,rs1,shamt
```

Description

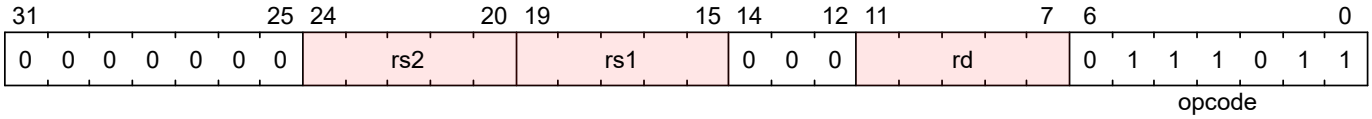
Performs arithmetic right shift on the 32-bit of value in register rs1 by the shift amount held in the lower 5 bits of the immediate. Encodings with $imm[5] neq 0$ are reserved.

Implementation

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

2.5. addw

Encoding



Format

```
addw rd,rs1,rs2
```

Description

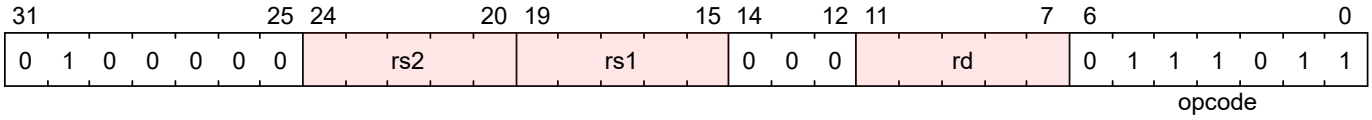
Adds the 32-bit of registers rs1 and 32-bit of register rs2 and stores the result in rd. Arithmetic overflow is ignored and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

Implementation

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

2.6. subw

Encoding



Format

```
subw rd,rs1,rs2
```

Description

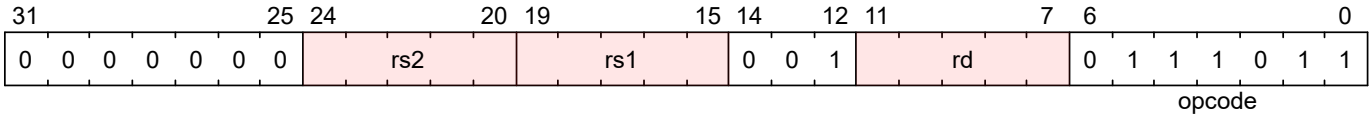
Subtract the 32-bit of registers rs1 and 32-bit of register rs2 and stores the result in rd. Arithmetic overflow is ignored and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

Implementation

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

2.7. sllw

Encoding



Format

```
sllw rd,rs1,rs2
```

Description

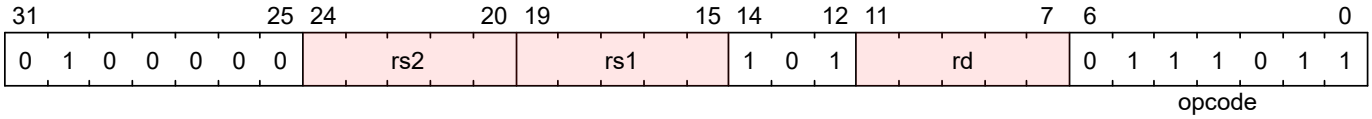
Performs logical left shift on the low 32-bits value in register rs1 by the shift amount held in the lower 5 bits of register rs2 and produce 32-bit results and written to the destination register rd.

Implementation

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

2.8. srlw

Encoding



Format

```
srlw rd,rs1,rs2
```

Description

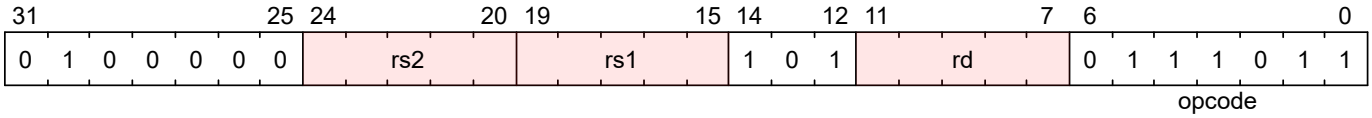
Performs logical right shift on the low 32-bits value in register rs1 by the shift amount held in the lower 5 bits of register rs2 and produce 32-bit results and written to the destination register rd.

Implementation

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

2.9. sraw

Encoding



Format

```
sraw rd,rs1,rs2
```

Description

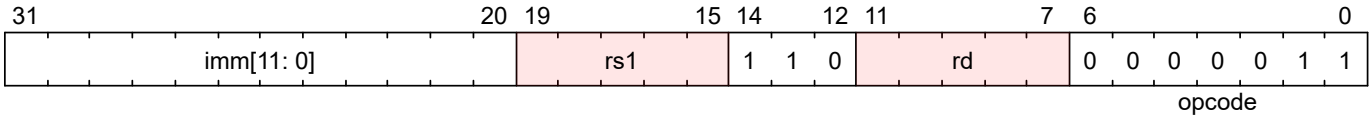
Performs arithmetic right shift on the low 32-bits value in register rs1 by the shift amount held in the lower 5 bits of register rs2 and produce 32-bit results and written to the destination register rd.

Implementation

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

2.10. lwu

Encoding



Format

```
lwu rd,offset(rs1)
```

Description

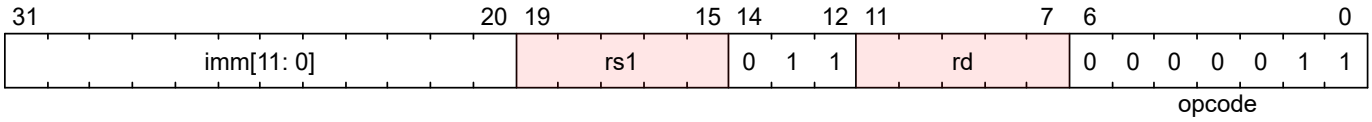
Loads a 32-bit value from memory and zero-extends this to 64 bits before storing it in register rd.

Implementation

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

2.11. ld

Encoding



Format

```
ld rd,offset(rs1)
```

Description

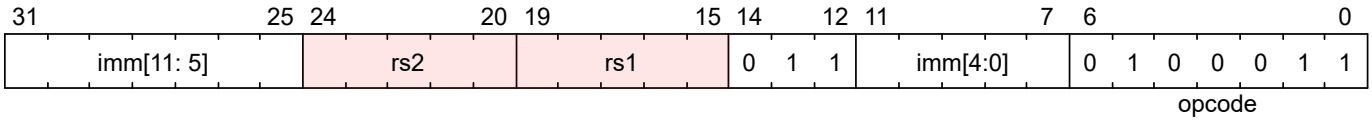
Loads a 64-bit value from memory into register rd for RV64I.

Implementation

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

2.12. sd

Encoding



Format

```
sd rs2,offset(rs1)
```

Description

Store 64-bit, values from register rs2 to memory.

Implementation

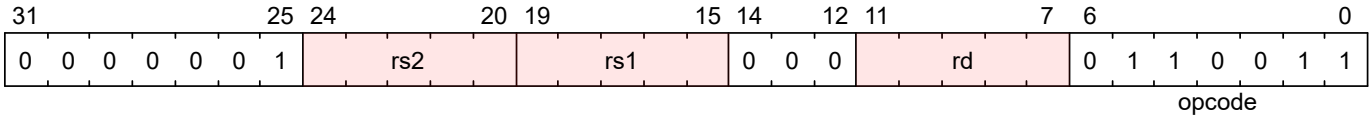
```
M[x[rs1] + sext(offset)] = x[rs2][63:0]
```

---

3. RV32M, RV64M Instructions

3.1. mul

Encoding



Format

```
mul rd,rs1,rs2
```

Description

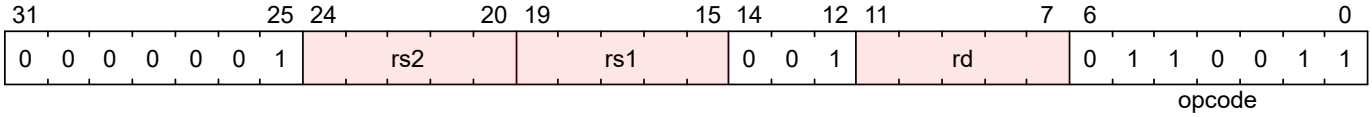
performs an XLEN-bit \(\times\) XLEN-bit multiplication of signed rs1 by signed rs2 and places the lower XLEN bits in the destination register.

Implementation

```
x[rd] = x[rs1] × x[rs2]
```

3.2. mulh

Encoding



Format

```
mulh rd,rs1,rs2
```

Description

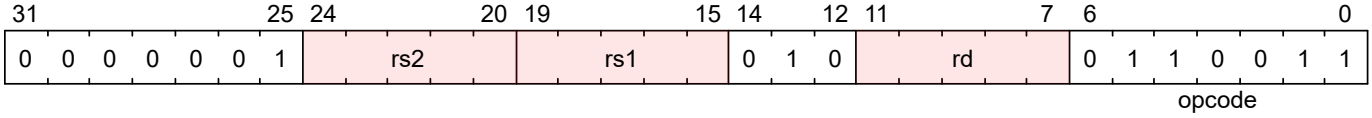
performs an XLEN-bit \(\times\) XLEN-bit multiplication of signed rs1 by signed rs2 and places the upper XLEN bits in the destination register.

Implementation

```
x[rd] = (x[rs1] s×s x[rs2]) >>s XLEN
```

3.3. mulhsu

Encoding



Format

```
mulhsu rd,rs1,rs2
```

Description

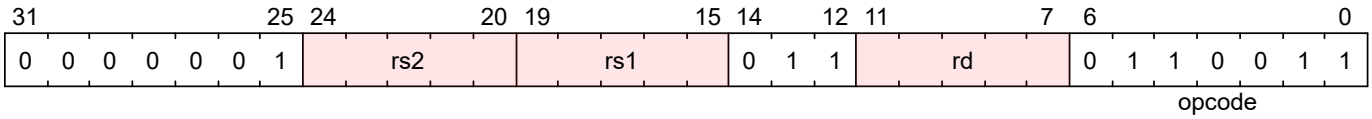
performs an XLEN-bit \(\times\) XLEN-bit multiplication of signed rs1 by unsigned rs2 and places the upper XLEN bits in the destination register.

Implementation

```
x[rd] = (x[rs1] s latexmath:[$\times$] x[rs2]) >>s XLEN
```

3.4. mulhu

Encoding



Format

```
mulhu rd,rs1,rs2
```

Description

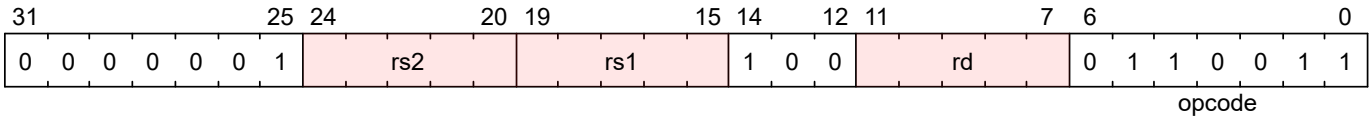
performs an XLEN-bit \(\times\) XLEN-bit multiplication of unsigned rs1 by unsigned rs2 and places the upper XLEN bits in the destination register.

Implementation

```
x[rd] = (x[rs1] u latexmath:[$\times$] x[rs2]) >>u XLEN
```

3.5. div

Encoding



Format

```
div rd,rs1,rs2
```

Description

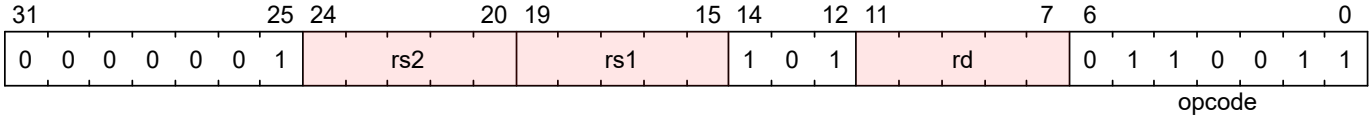
perform an XLEN bits by XLEN bits signed integer division of rs1 by rs2, rounding towards zero.

Implementation

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

3.6. divu

Encoding



Format

```
divu rd,rs1,rs2
```

Description

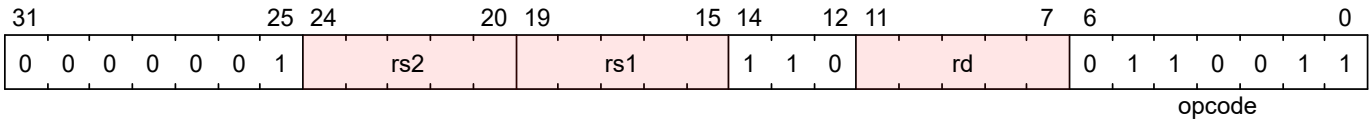
perform an XLEN bits by XLEN bits unsigned integer division of rs1 by rs2, rounding towards zero.

Implementation

```
x[rd] = x[rs1] /u x[rs2]
```

3.7. rem

Encoding



Format

```
rem rd,rs1,rs2
```

Description

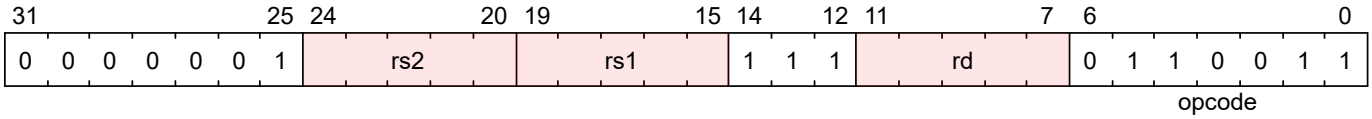
perform an XLEN bits by XLEN bits signed integer reminder of rs1 by rs2.

Implementation

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

3.8. remu

Encoding



Format

```
remu rd,rs1,rs2
```

Description

perform an XLEN bits by XLEN bits unsigned integer reminder of rs1 by rs2.

Implementation

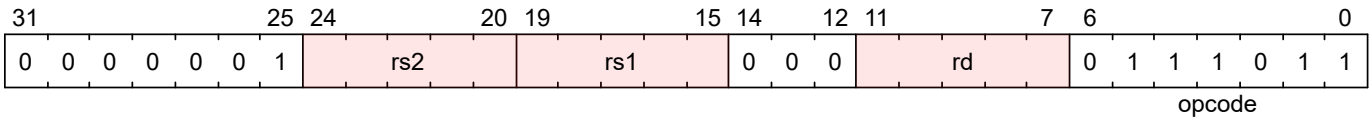
```
x[rd] = x[rs1] %u x[rs2]
```

---

4. RV64M Instructions

4.1. mulw

Encoding



Format

```
mulw rd,rs1,rs2
```

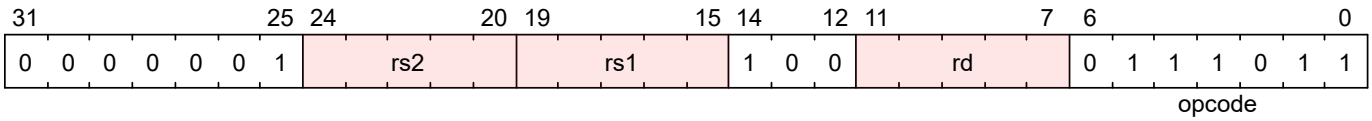
Description

Implementation

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

4.2. divw

Encoding



Format

```
divw rd,rs1,rs2
```

Description

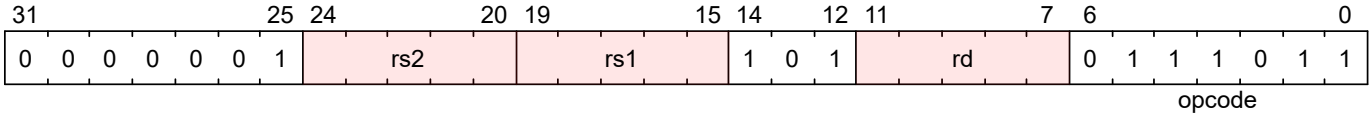
perform an 32 bits by 32 bits signed integer division of rs1 by rs2.

Implementation

```
x[rd] = sext(x[rs1][31:0] /s x[rs2][31:0]
```

=== divuw

Encoding



Format

```
divuw rd,rs1,rs2
```

Description

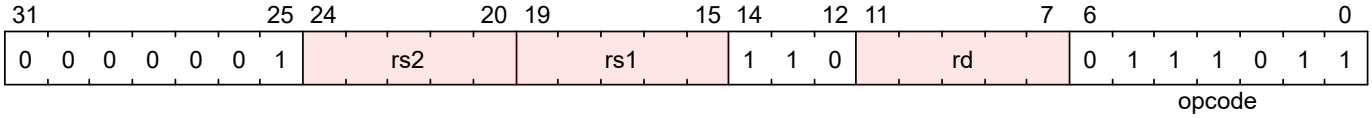
perform an 32 bits by 32 bits unsigned integer division of rs1 by rs2.

Implementation

```
x[rd] = sext(x[rs1][31:0] /u x[rs2][31:0])
```

4.3. remw

Encoding



Format

```
remw rd,rs1,rs2
```

Description

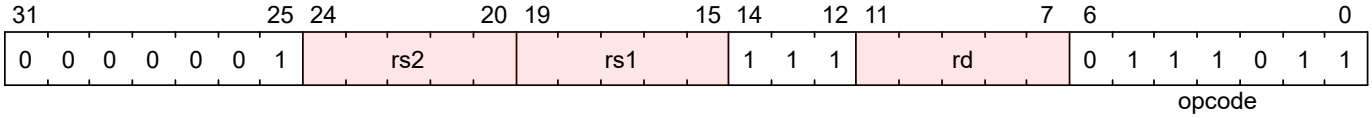
perform an 32 bits by 32 bits signed integer reminder of rs1 by rs2.

Implementation

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

=== remuw

Encoding



Format

```
remuw rd,rs1,rs2
```

Description

perform an 32 bits by 32 bits unsigned integer reminder of rs1 by rs2.

Implementation

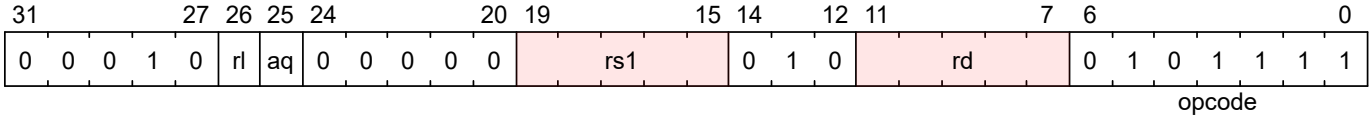
```
x[rd] = sext(x[rs1][31:0] %u x[rs2][31:0])
```

---

5. RV32A, RV64A Instructions

5.1. lr.w

Encoding



Format

```
lr.w rd,rs1
```

Description

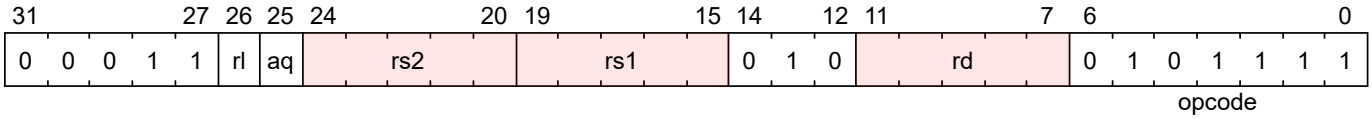
load a word from the address in rs1, places the sign-extended value in rd, and registers a reservation on the memory address.

Implementation

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

5.2. sc.w

Encoding



Format

```
sc.w rd,rs1,rs2
```

Description

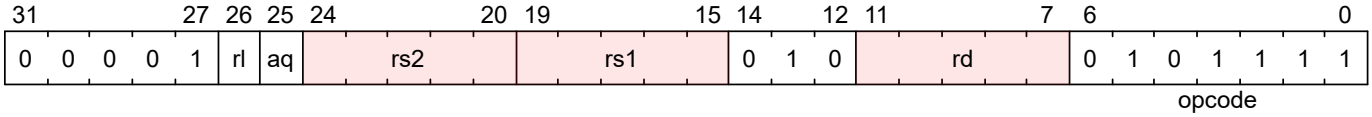
write a word in rs2 to the address in rs1, provided a valid reservation still exists on that address. SC writes zero to rd on success or a nonzero code on failure.

Implementation

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

5.3. amoswap.w

Encoding



Format

```
amoswap.w rd,rs2,(rs1)
```

Description

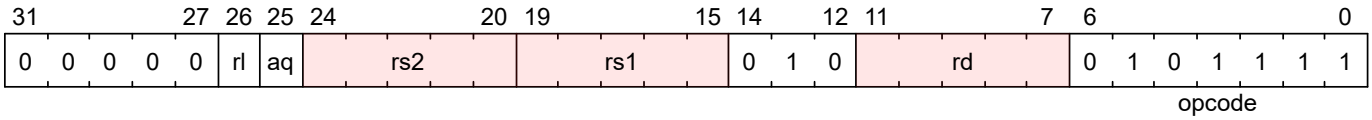
atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, swap the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.4. amoadd.w

Encoding



Format

```
amoadd.w rd,rs2,(rs1)
```

Description

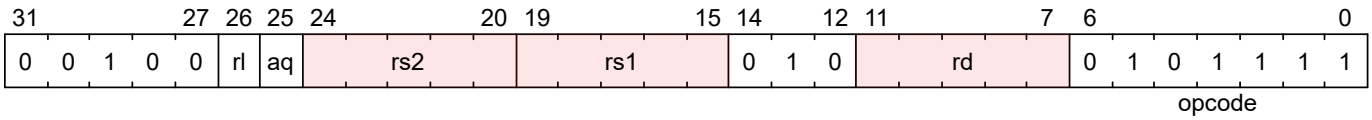
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply add the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.5. amoxor.w

Encoding



Format

```
amoxor.w rd,rs2,(rs1)
```

Description

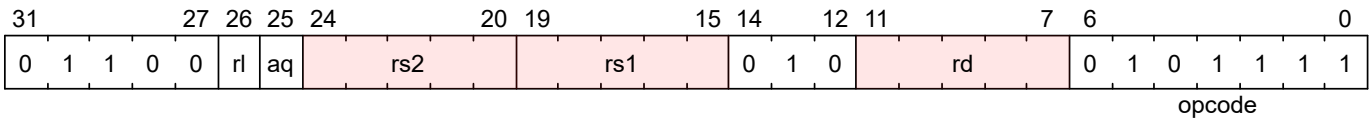
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply exclusive or the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.6. amoand.w

Encoding



Format

```
amoand.w rd,rs2,(rs1)
```

Description

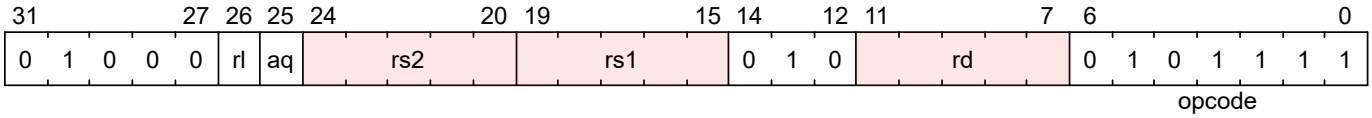
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply and the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.7. amoor.w

Encoding



Format

```
amoor.w rd,rs2,(rs1)
```

Description

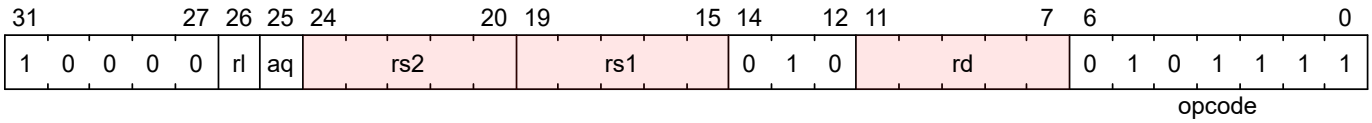
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply or the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.8. amomin.w

Encoding



Format

```
amomin.w rd,rs2,(rs1)
```

Description

Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply min operator the loaded value and the original 32-bit signed value in rs2, then store the

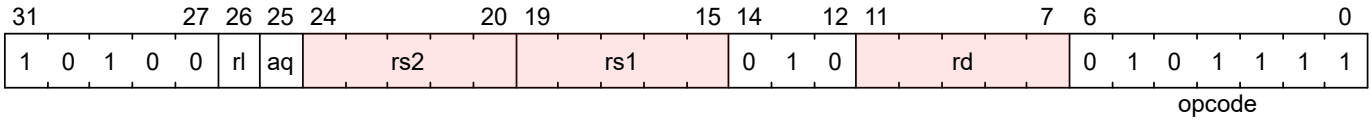
result back to the address in rs1.

Implementation

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

5.9. amomax.w

Encoding



Format

```
amomax.w rd,rs2,(rs1)
```

Description

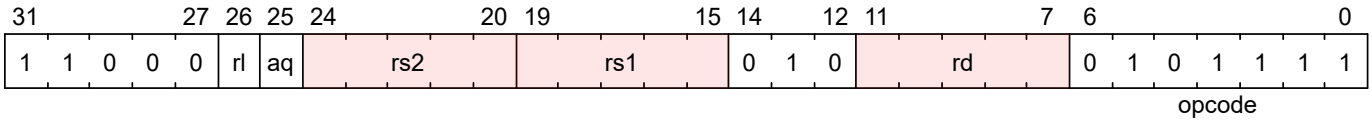
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply max operator the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.10. amominu.w

Encoding



Format

```
amominu.w rd,rs2,(rs1)
```

Description

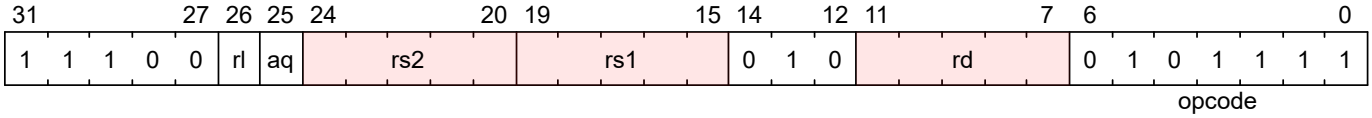
Atomically load a 32-bit unsigned data value from the address in rs1, place the value into register rd, apply unsigned min the loaded value and the original 32-bit unsigned value in rs2, then store the result back to the address in rs1.

Implementation

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

5.11. amomaxu.w

Encoding



Format

```
amomaxu.w rd,rs2,(rs1)
```

Description

Atomically load a 32-bit unsigned data value from the address in rs1, place the value into register rd, apply unsigned max the loaded value and the original 32-bit unsigned value in rs2, then store the result back to the address in rs1.

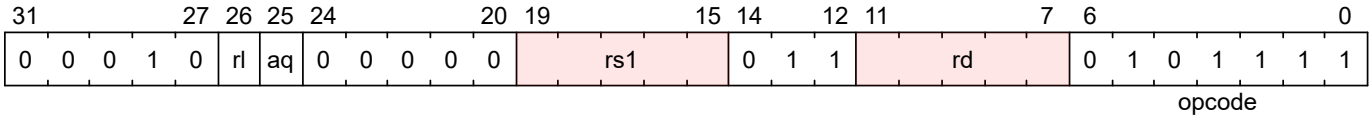
Implementation

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

6. RV64A Instructions

6.1. lr.d

Encoding



Format

```
lr.d rd,rs1
```

Description

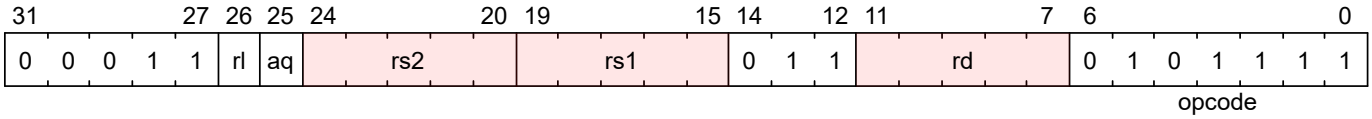
load a 64-bit data from the address in rs1, places value in rd, and registers a reservation on the memory address.

Implementation

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

6.2. sc.d

Encoding



Format

```
sc.d rd,rs1,rs2
```

Description

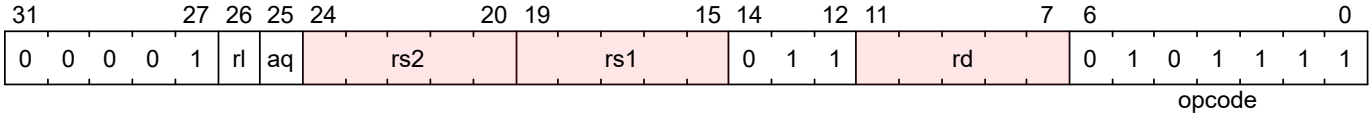
write a 64-bit data in rs2 to the address in rs1, provided a valid reservation still exists on that address. SC writes zero to rd on success or a nonzero code on failure.

Implementation

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

6.3. amoswap.d

Encoding



Format

```
amoswap.d rd,rs2,(rs1)
```

Description

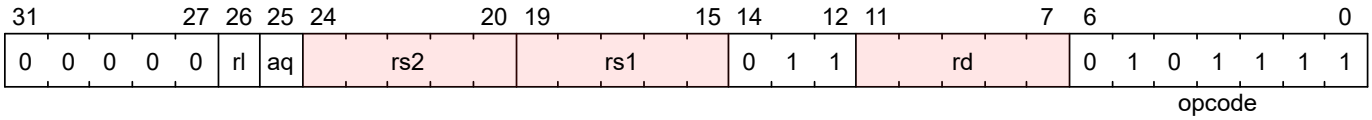
atomically load a 64-bit data value from the address in rs1, place the value into register rd, swap the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.4. amoadd.d

Encoding



Format

```
amoadd.d rd,rs2,(rs1)
```

Description

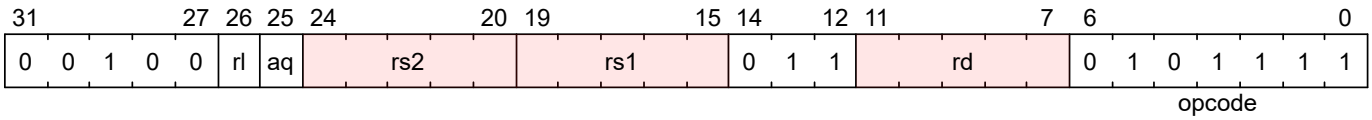
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply add the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.5. amoxor.d

Encoding



Format

```
amoxor.d rd,rs2,(rs1)
```

Description

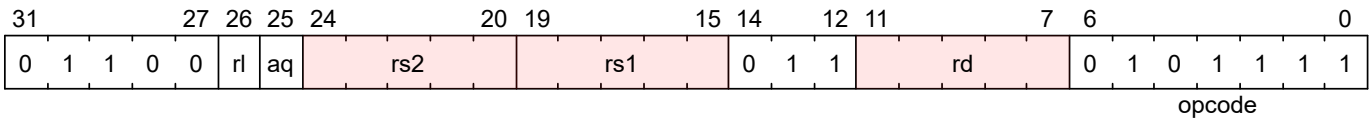
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply xor the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.6. amoand.d

Encoding



Format

```
amoand.d rd,rs2,(rs1)
```

Description

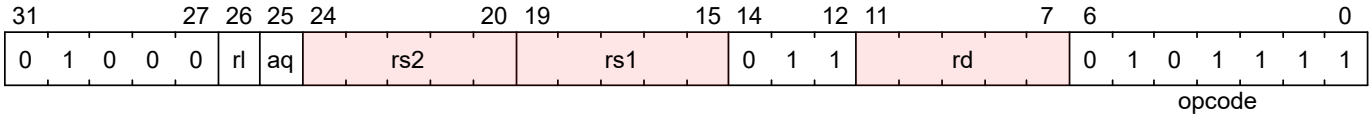
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply and the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.7. amoor.d

Encoding



Format

```
amoor.d rd,rs2,(rs1)
```

Description

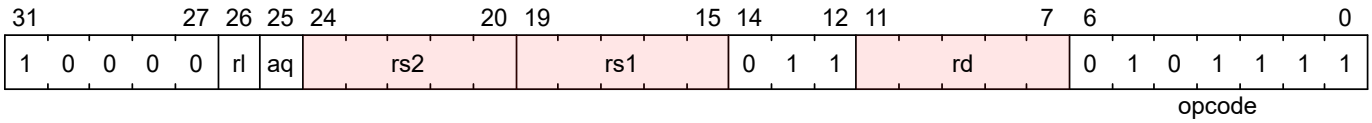
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply or the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.8. amomin.d

Encoding



Format

```
amomin.d rd,rs2,(rs1)
```

Description

atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply min the loaded value and the original 64-bit value in rs2, then store the result back to the address

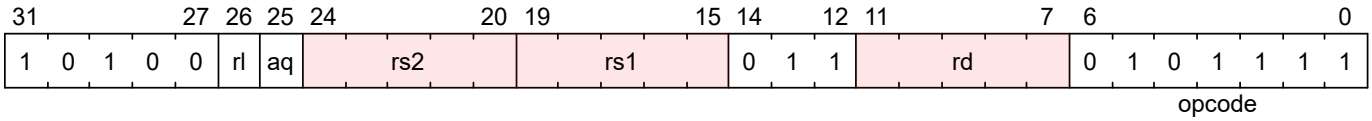
in rs1.

Implementation

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

6.9. amomax.d

Encoding



Format

```
amomax.d rd,rs2,(rs1)
```

Description

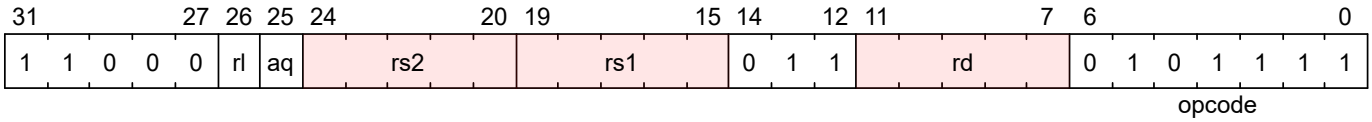
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply max the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.10. amominu.d

Encoding



Format

```
amominu.d rd,rs2,(rs1)
```

Description

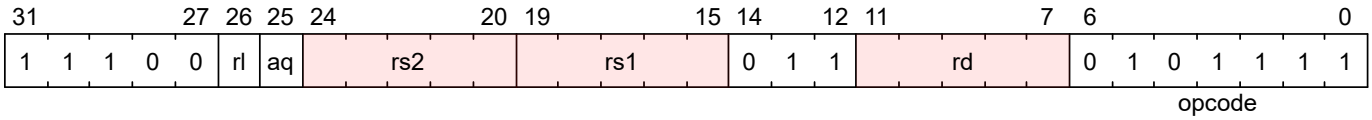
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply unsigned min the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.11. amomaxu.d

Encoding



Format

```
amomaxu.d rd,rs2,(rs1)
```

Description

atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply unsigned max the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

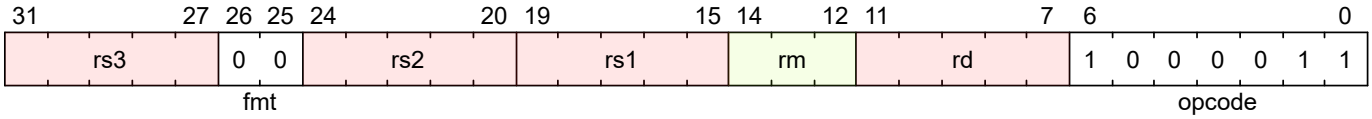
```
x[rd] = AMO64(M[x[rs1]] MAXU x[rs2])
```

---

7. RV32F, RV64D Instructions

7.1. fmadd.s

Encoding



Format

fmadd.s rd,rs1,rs2,rs3

Description

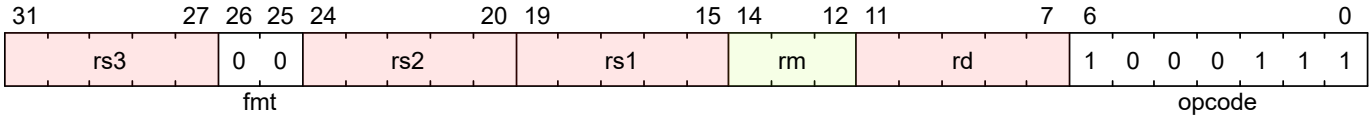
Perform single-precision fused multiply addition.

Implementation

```
f[rd] = f[rs1]×f[rs2]+f[rs3]
```

7.2. fmsub.s

Encoding



Format

fmsub.s rd,rs1,rs2,rs3

Description

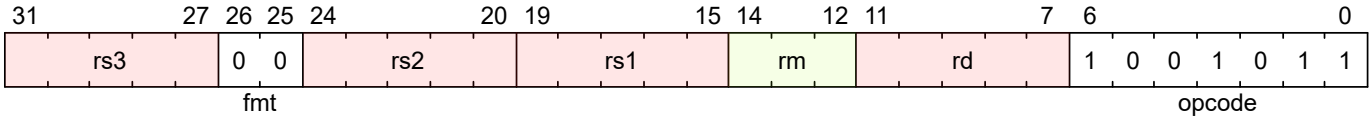
Perform single-precision fused multiply addition.

Implementation

```
f[rd] = f[rs1]×f[rs2]-f[rs3]
```

7.3. fnmsub.s

Encoding



Format

fnmsub.s rd,rs1,rs2,rs3

Description

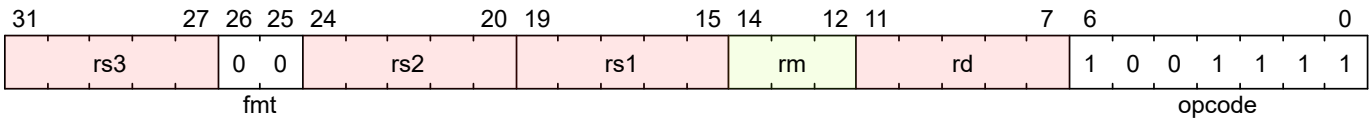
Perform negated single-precision fused multiply subtraction.

Implementation

```
f[rd] = -f[rs1]×f[rs2]+f[rs3]
```

7.4. fnmadd.s

Encoding



Format

fnmadd.s rd,rs1,rs2,rs3

Description

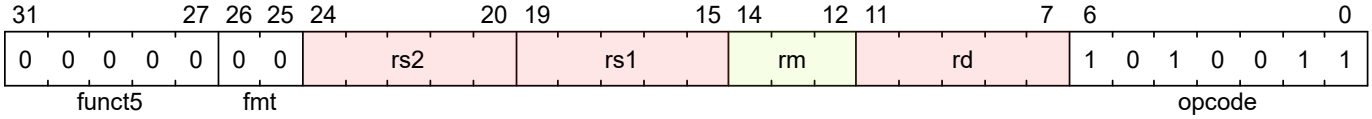
Perform negated single-precision fused multiply addition.

Implementation

```
f[rd] = -f[rs1]×f[rs2]-f[rs3]
```

7.5. fadd.s

Encoding



Format

fadd.s rd,rs1,rs2

Description

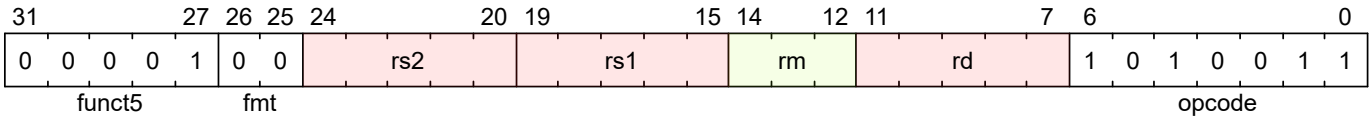
Perform single-precision floating-point addition.

Implementation

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

7.6. fsub.s

Encoding



Format

fsub.s rd,rs1,rs2

Description

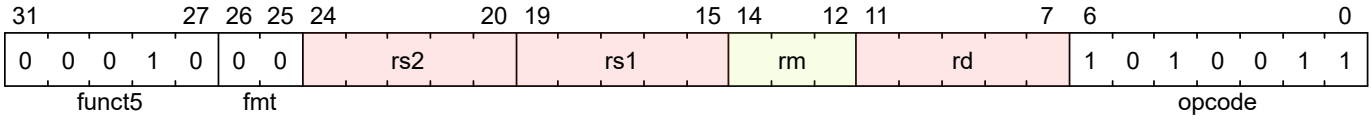
Perform single-precision floating-point substraction.

Implementation

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

7.7. fmul.s

Encoding



Format

fmul.s rd,rs1,rs2

Description

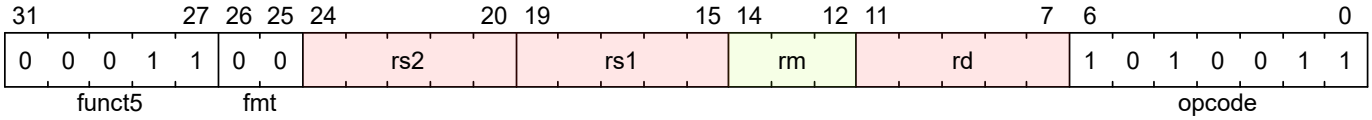
Perform single-precision floating-point multiplication.

Implementation

```
f[rd] = f[rs1] × f[rs2]
```

7.8. fdiv.s

Encoding



Format

fdiv.s rd,rs1,rs2

Description

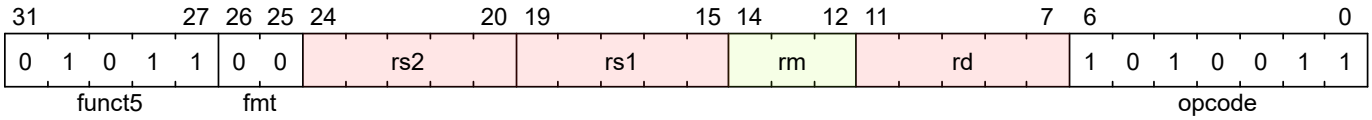
Perform single-precision floating-point division.

Implementation

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

7.9. fsqrt.s

Encoding



Format

fsqrt.s rd,rs1

Description

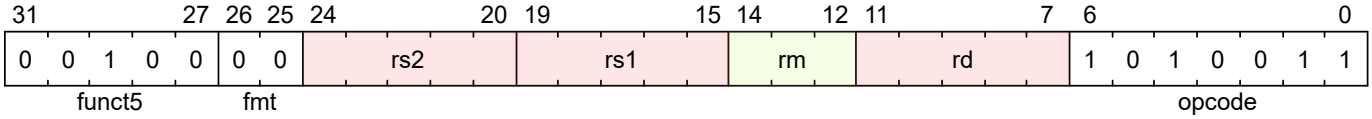
Perform single-precision square root.

Implementation

```
f[rd] = sqrt(f[rs1])
```

7.10. fsgnj.s

Encoding



Format

fsgnj.s rd,rs1,rs2

Description

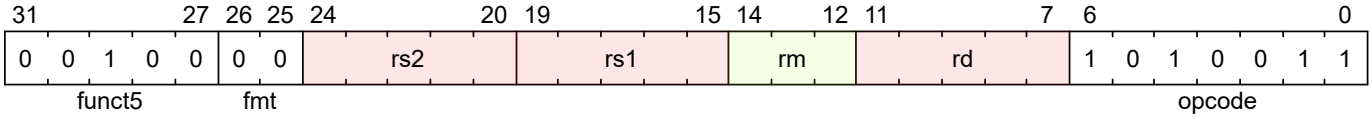
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is rs2's sign bit.

Implementation

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

7.11. fsgnjn.s

Encoding



Format

fsgnjn.s rd,rs1,rs2

Description

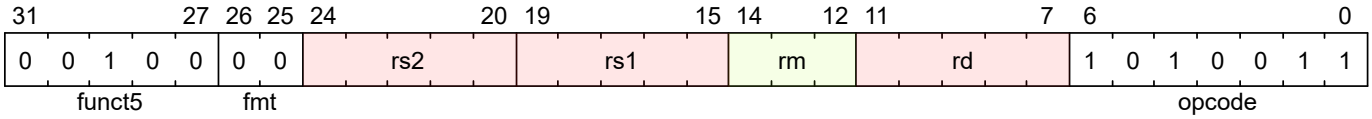
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is opposite of rs2's sign bit.

Implementation

```
f[rd] = {~f[rs2][31], f[rs1][30:0]}
```

7.12. fsgnjx.s

Encoding



Format

fsgnjx.s rd,rs1,rs2

Description

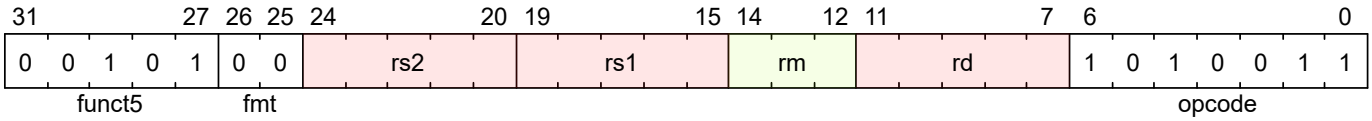
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is XOR of sign bit of rs1 and rs2.

Implementation

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

7.13. fmin.s

Encoding



Format

fmin.s rd,rs1,rs2

Description

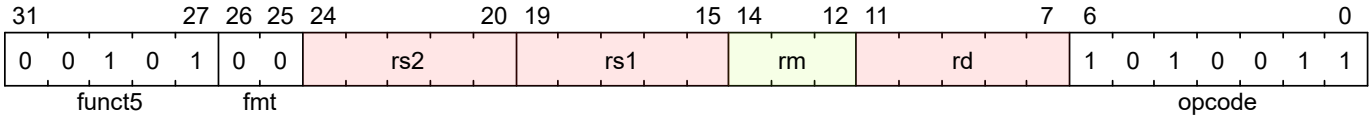
Write the smaller of single precision data in rs1 and rs2 to rd.

Implementation

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

7.14. fmax.s

Encoding



Format

fmax.s rd,rs1,rs2

Description

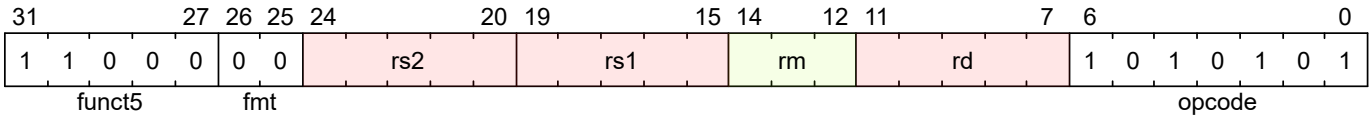
Write the larger of single precision data in rs1 and rs2 to rd.

Implementation

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

7.15. fcvt.w.s

Encoding



Format

fcvt.w.s rd,rs1

Description

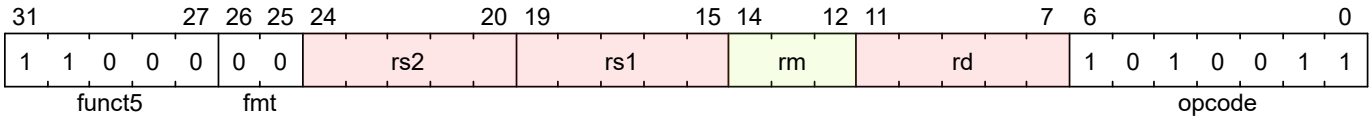
Convert a floating-point number in floating-point register rs1 to a signed 32-bit in integer register rd.

Implementation

```
x[rd] = sext(f32->s32(f[rs1]))
```

7.16. fcvt.wu.s

Encoding



Format

fcvt.wu.s rd,rs1

Description

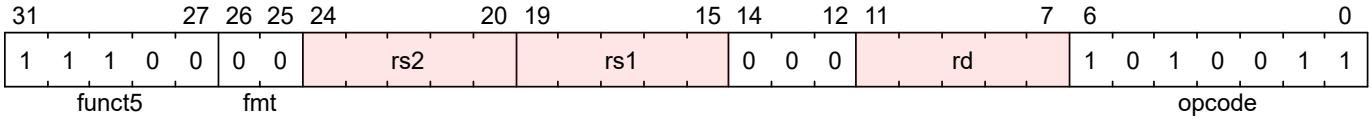
Convert a floating-point number in floating-point register rs1 to a signed 32-bit in unsigned integer register rd.

Implementation

```
x[rd] = sext(f32->u32(f[rs1]))
```

7.17. fmv.x.w

Encoding



Format

fmv.x.w rd,rs1

Description

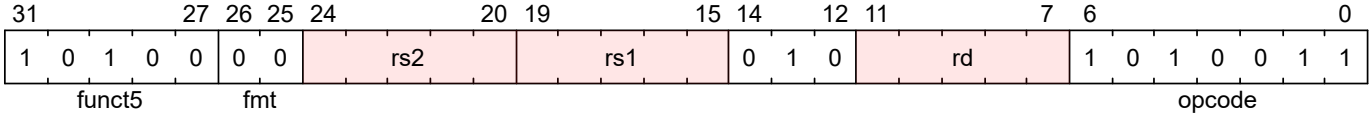
Move the single-precision value in floating-point register rs1 represented in IEEE 754-2008 encoding to the lower 32 bits of integer register rd.

Implementation

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

7.18. feq.s

Encoding



Format

feq.s rd,rs1,rs2

Description

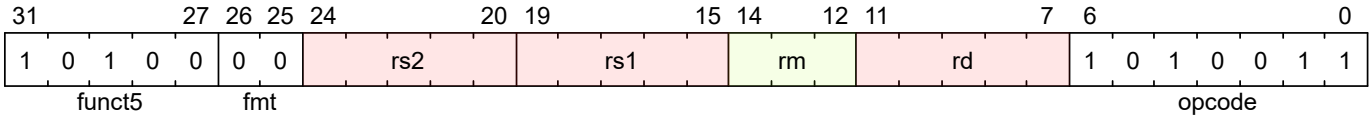
Performs a quiet equal comparison between single-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

7.19. flt.s

Encoding



Format

flt.s rd,rs1,rs2

Description

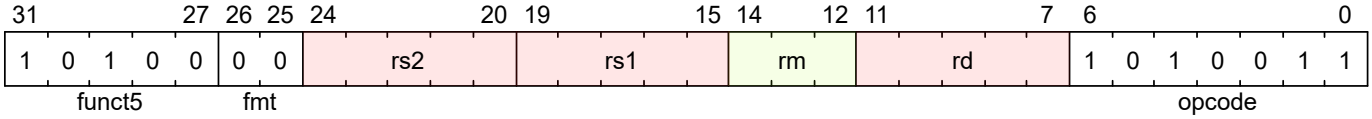
Performs a quiet less comparison between single-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

7.20. fle.s

Encoding



Format

fle.s rd,rs1,rs2

Description

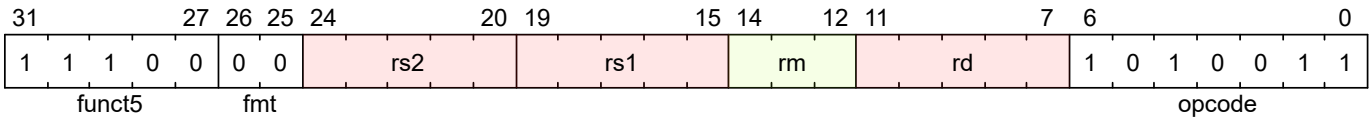
Performs a quiet less or equal comparison between single-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

7.21. fclass.s

Encoding



Format

fclass.s rd,rs1

Description

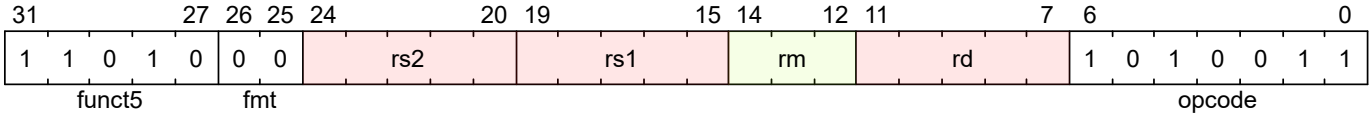
Examines the value in single-precision floating-point register rs1 and writes to integer register rd a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in [classify table]_. The corresponding bit in rd will be set if the property is true and clear otherwise. All other bits in rd are cleared. Note that exactly one bit in rd will be set.

Implementation

```
x[rd] = classify_s(f[rs1])
```

7.22. fcvt.s.w

Encoding



Format

fcvt.s.w rd,rs1

Description

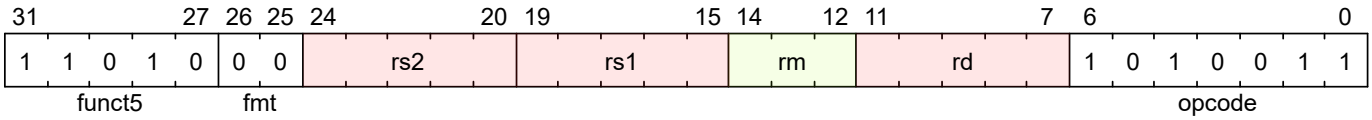
Converts a 32-bit signed integer, in integer register rs1 into a floating-point number in floating-point register rd.

Implementation

```
f[rd] = s32->f32(x[rs1])
```

7.23. fcvt.s.wu

Encoding



Format

fcvt.s.wu rd,rs1

Description

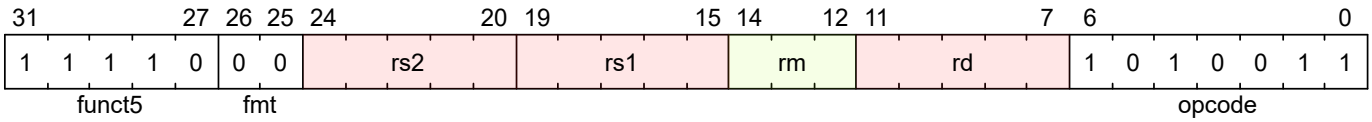
Converts a 32-bit unsigned integer, in integer register rs1 into a floating-point number in floating-point register rd.

Implementation

```
f[rd] = u32->f32(x[rs1])
```

7.24. fmv.w.x

Encoding



Format

fmv.w.x rd,rs1

Description

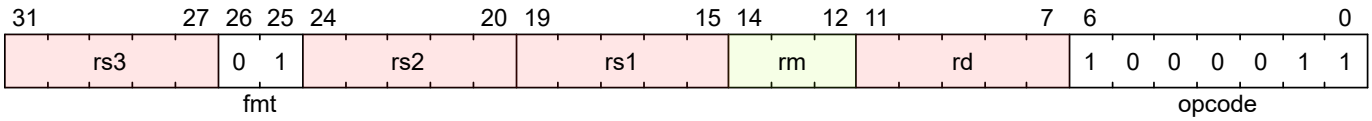
Move the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register rs1 to the floating-point register rd.

Implementation

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

7.25. fmadd.d

Encoding



Format

fmadd.d rd,rs1,rs2,rs3

Description

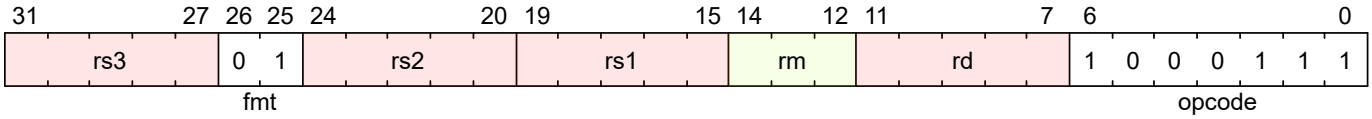
Perform double-precision fused multiply addition.

Implementation

```
f[rd] = f[rs1]×f[rs2]+f[rs3]
```

7.26. fmsub.d

Encoding



Format

fmsub.d rd,rs1,rs2,rs3

Description

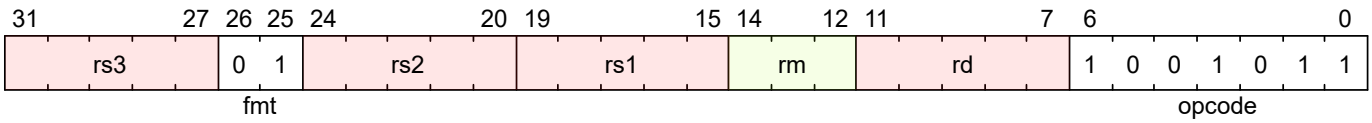
Perform double-precision fused multiply subtraction.

Implementation

```
f[rd] = f[rs1]×f[rs2]-f[rs3]
```

7.27. fnmsub.d

Encoding



Format

fnmsub.d rd,rs1,rs2,rs3

Description

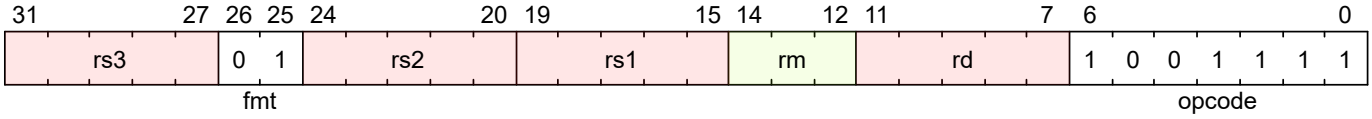
Perform negated double-precision fused multiply subtraction.

Implementation

```
f[rd] = -f[rs1]×f[rs2]+f[rs3]
```

7.28. fnmadd.d

Encoding



Format

fnmadd.d rd,rs1,rs2,rs3

Description

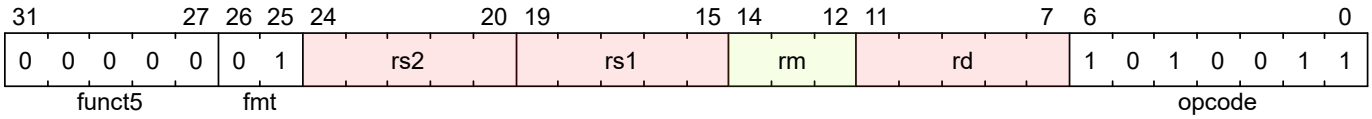
Perform negated double-precision fused multiply addition.

Implementation

```
f[rd] = -f[rs1]×f[rs2]-f[rs3]
```

7.29. fadd.d

Encoding



Format

fadd.d rd,rs1,rs2

Description

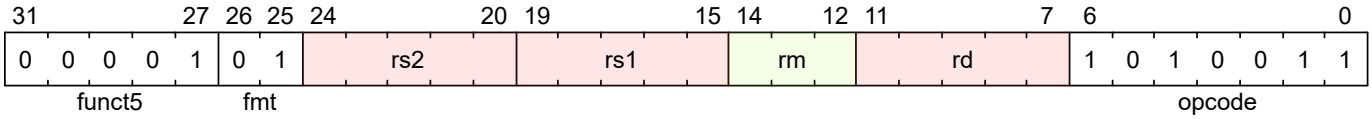
Perform double-precision floating-point addition.

Implementation

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

7.30. fsub.d

Encoding



Format

fsub.d rd,rs1,rs2

Description

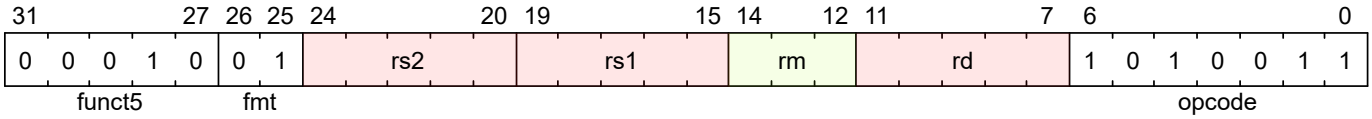
Perform double-precision floating-point addition.

Implementation

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

7.31. fmul.d

Encoding



Format

fmul.d rd,rs1,rs2

Description

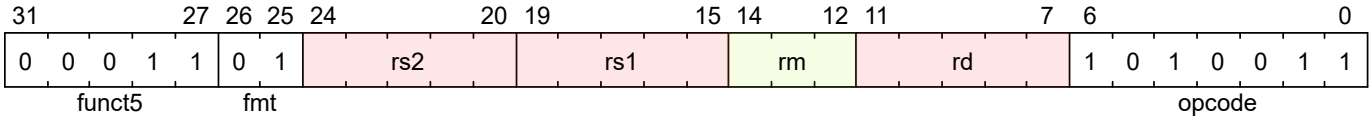
Perform double-precision floating-point addition.

Implementation

```
f[rd] = f[rs1] × f[rs2]
```

7.32. fdiv.d

Encoding



Format

fdiv.d rd,rs1,rs2

Description

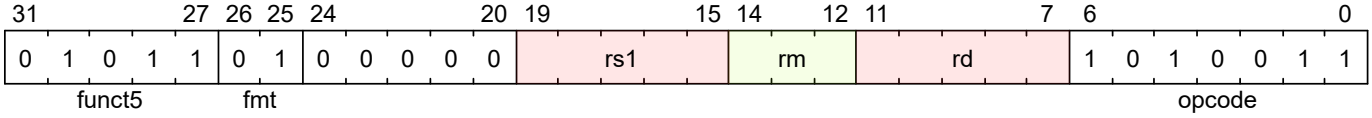
Perform double-precision floating-point addition.

Implementation

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

7.33. fsqrt.d

Encoding



Format

fsqrt.d rd,rs1

Description

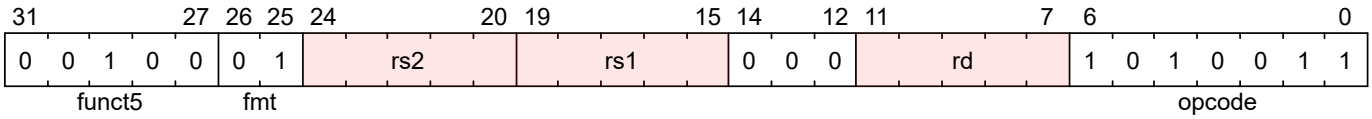
Perform double-precision square root.

Implementation

```
f[rd] = sqrt(f[rs1])
```

7.34. fsgnj.d

Encoding



Format

fsgnj.d rd,rs1,rs2

Description

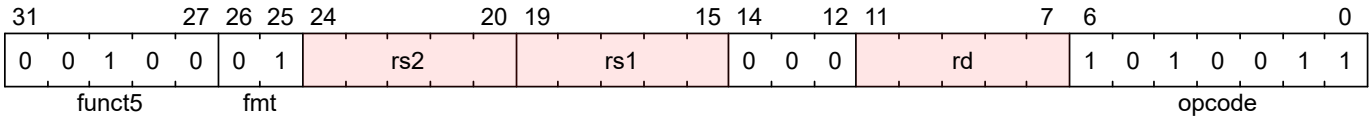
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is rs2's sign bit.

Implementation

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

7.35. fsgnjn.d

Encoding



Format

fsgnjn.d rd,rs1,rs2

Description

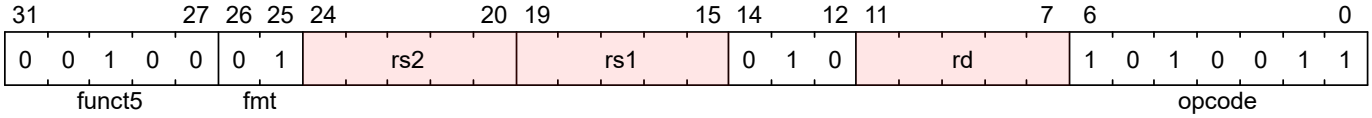
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is opposite of rs2's sign bit.

Implementation

```
f[rd] = {~f[rs2][63], f[rs1][62:0]}
```

7.36. fsgnjx.d

Encoding



Format

fsgnjx.d rd,rs1,rs2

Description

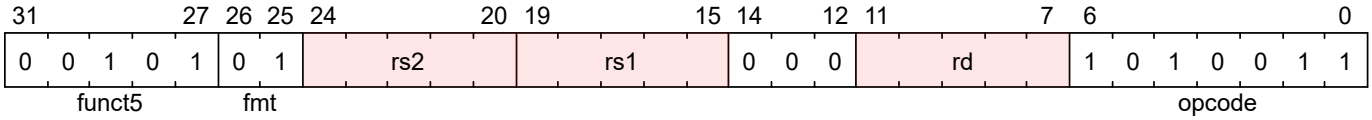
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is XOR of sign bit of rs1 and rs2.

Implementation

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

7.37. fmin.d

Encoding



Format

fmin.d rd,rs1,rs2

Description

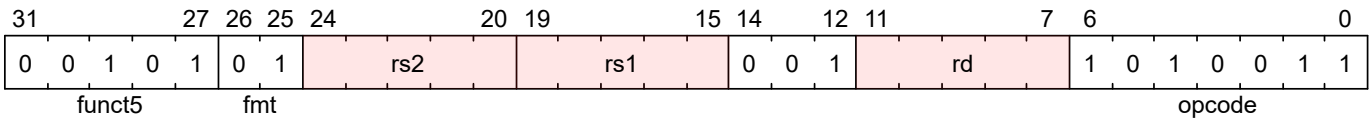
Write the smaller of double precision data in rs1 and rs2 to rd.

Implementation

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

7.38. fmax.d

Encoding



Format

fmax.d rd,rs1,rs2

Description

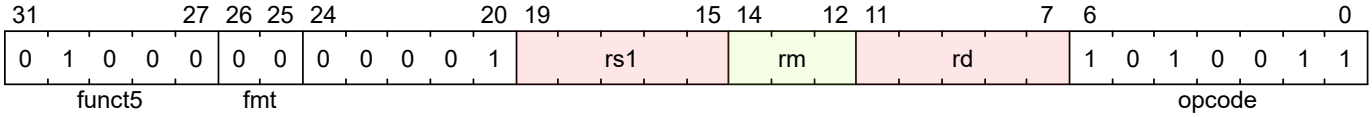
Write the larger of double precision data in rs1 and rs2 to rd.

Implementation

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

7.39. fcvt.s.d

Encoding



Format

fcvt.s.d rd,rs1

Description

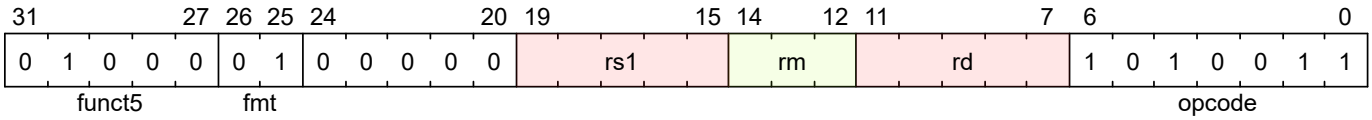
Converts double floating-point register in rs1 into a floating-point number in floating-point register rd.

Implementation

```
f[rd] = f64->f32(f[rs1])
```

7.40. fcvt.d.s

Encoding



Format

fcvt.d.s rd,rs1

Description

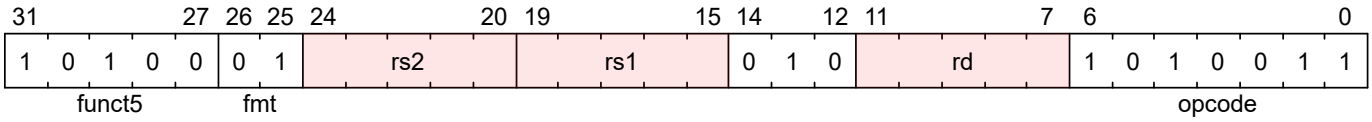
Converts single floating-point register in rs1 into a double floating-point number in floating-point register rd.

Implementation

```
f[rd] = f32->f64(f[rs1])
```

7.41. feq.d

Encoding



Format

feq.d rd,rs1,rs2

Description

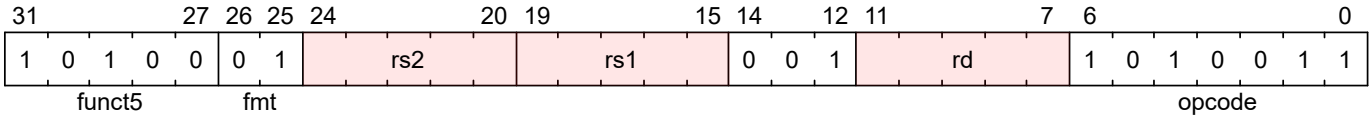
Performs a quiet equal comparison between double-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

7.42. flt.d

Encoding



Format

flt.d rd,rs1,rs2

Description

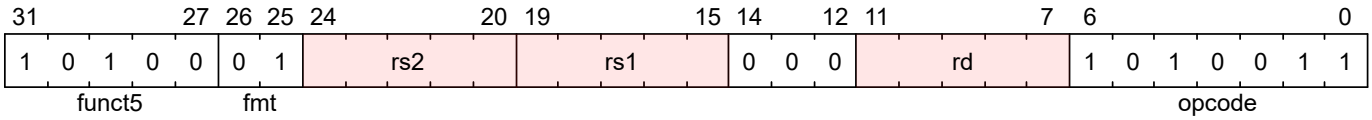
Performs a quiet less comparison between double-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

7.43. fle.d

Encoding



Format

fle.d rd,rs1,rs2

Description

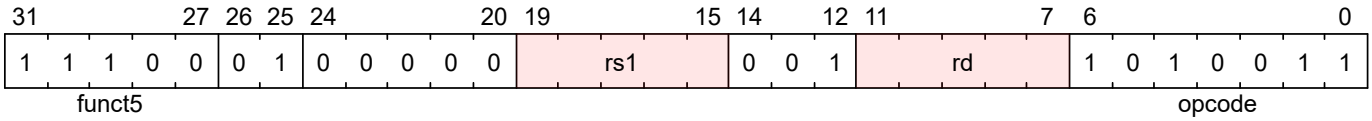
Performs a quiet less or equal comparison between double-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

7.44. fclass.d

Encoding



Format

fclass.d rd,rs1

Description

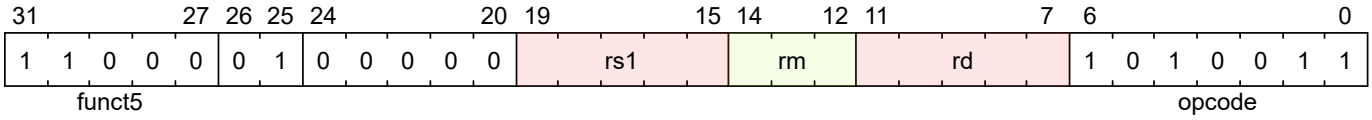
Examines the value in double-precision floating-point register rs1 and writes to integer register rd a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in table [classify table]_. The corresponding bit in rd will be set if the property is true and clear otherwise. All other bits in rd are cleared. Note that exactly one bit in rd will be set.

Implementation

```
x[rd] = classifys(f[rs1])
```

7.45. fcvt.w.d

Encoding



Format

fcvt.w.d rd,rs1

Description

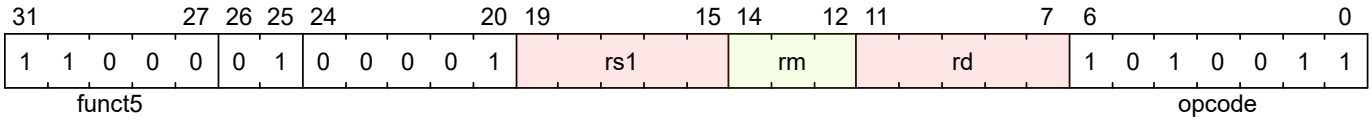
Converts a double-precision floating-point number in floating-point register rs1 to a signed 32-bit integer, in integer register rd.

Implementation

```
x[rd] = sext(f64->s32(f[rs1]))
```

7.46. fcvt.wu.d

Encoding



Format

fcvt.wu.d rd,rs1

Description

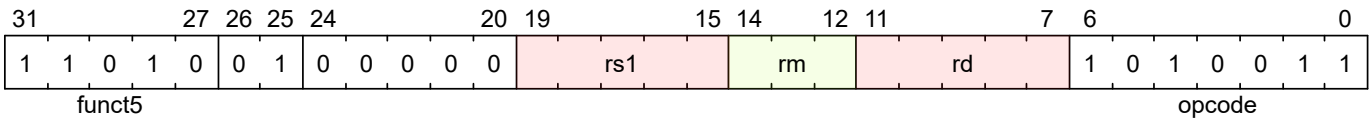
Converts a double-precision floating-point number in floating-point register rs1 to a unsigned 32-bit integer, in integer register rd.

Implementation

```
x[rd] = sext(u32f64(f[rs1]))
```

7.47. fcvt.d.w

Encoding



Format

fcvt.d.w rd,rs1

Description

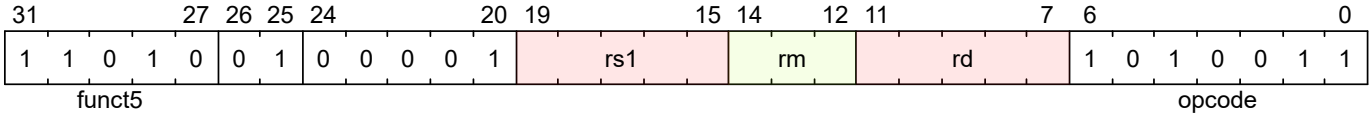
Converts a 32-bit signed integer, in integer register rs1 into a double-precision floating-point number in floating-point register rd.

Implementation

```
x[rd] = sext(f64->s32(f[rs1]))
```

7.48. fcvt.d.wu

Encoding



Format

fcvt.d.wu rd,rs1

Description

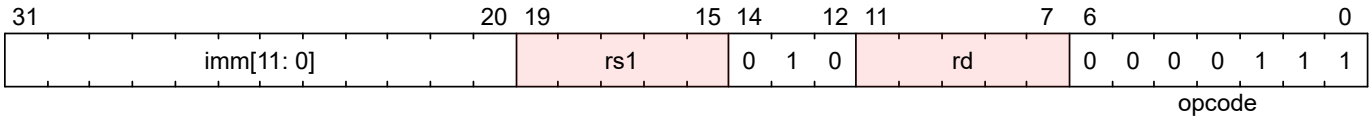
Converts a 32-bit unsigned integer, in integer register rs1 into a double-precision floating-point number in floating-point register rd.

Implementation

```
f[rd] = u32->f64(x[rs1])
```

7.49. flw

Encoding



Format

flw rd,offset(rs1)

Description

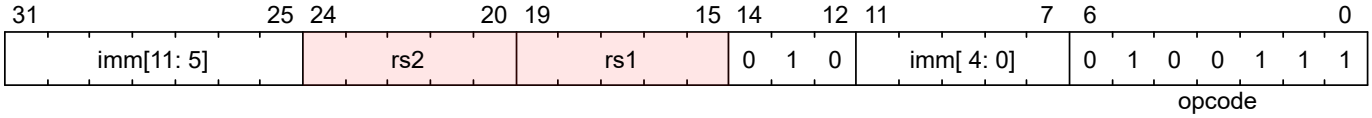
Load a single-precision floating-point value from memory into floating-point register rd.

Implementation

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

7.50. fsw

Encoding



Format

fsw rs2,offset(rs1)

Description

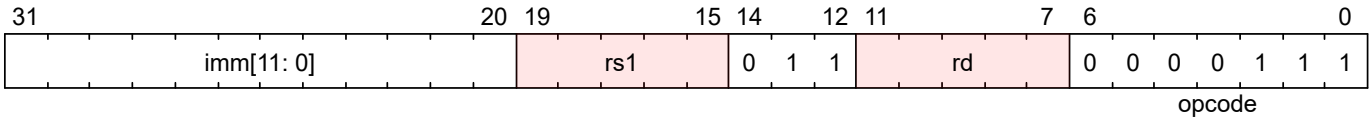
Store a single-precision value from floating-point register rs2 to memory.

Implementation

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

7.51. fld

Encoding



Format

fld rd,rs1,offset

Description

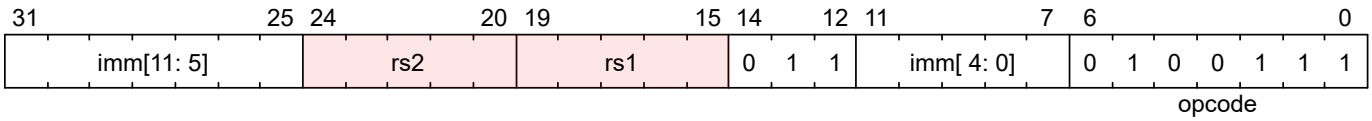
Load a double-precision floating-point value from memory into floating-point register rd.

Implementation

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

7.52. fsd

Encoding



Format

fsd rs2,offset(rs1)

Description

Store a double-precision value from the floating-point registers to memory.

Implementation

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

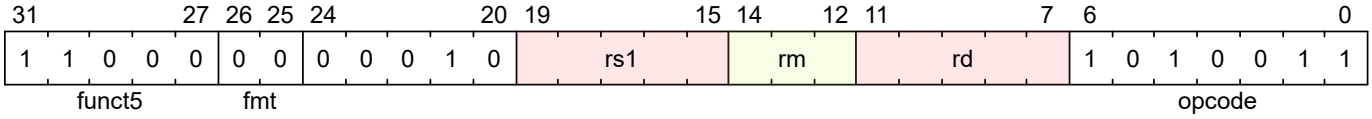
Table 1. Classify Table:

rd bit	Meaning
0	rs1 is -infinity
1	rs1 is a negative normal number.
2	rs1 is a negative subnormal number.
3	rs1 is −0.
4	rs1 is +0.
5	rs1 is a positive subnormal number.
6	rs1 is a positive normal number.
7	rs1 is +infinity
8	rs1 is a signaling NaN.
9	rs1 is a quiet NaN.

8. RV64F Instructions

8.1. fcvt.l.s

Encoding



Format

```
fcvt.l.s rd,rs1
```

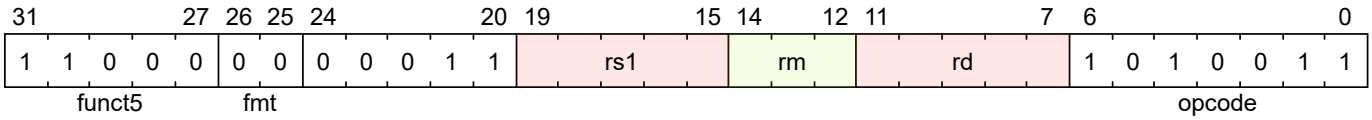
Description

Implementation

```
x[rd] = f32->s64(f[rs1])
```

8.2. fcvt.lu.s

Encoding



Format

```
fcvt.lu.s rd,rs1
```

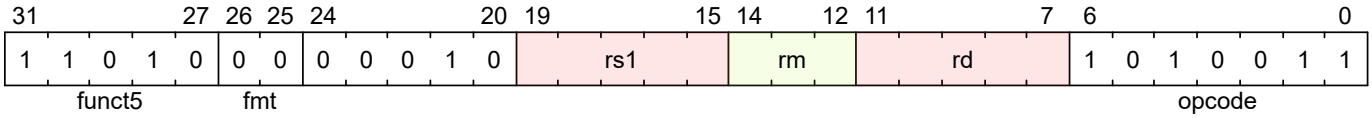
Description

Implementation

```
x[rd] = f32->u64(f[rs1])
```

8.3. fcvt.s.l

Encoding



Format

```
fcvt.s.l rd,rs1
```

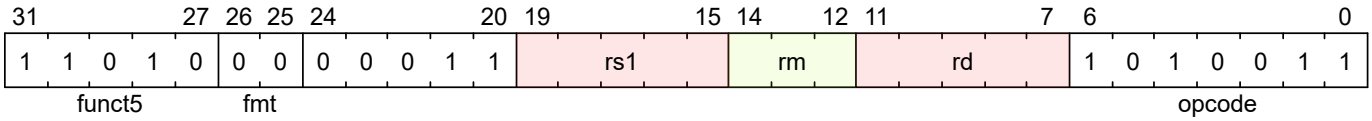
Description

Implementation

```
f[rd] = s64->f32(x[rs1])
```

8.4. fcvt.s.lu

Encoding



Format

```
fcvt.s.lu rd,rs1
```

Description

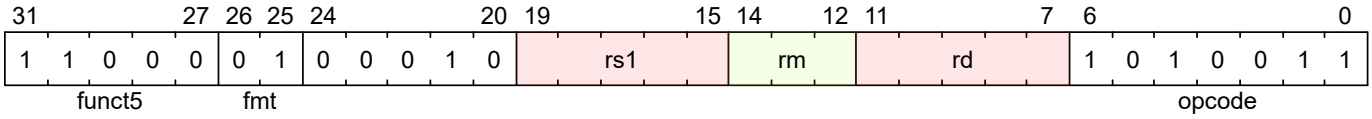
Implementation

```
f[rd] = u64->f32(x[rs1])
```

9. RV64D Instructions

9.1. fcvt.l.d

Encoding



Format

fcvt.l.d rd,rs1

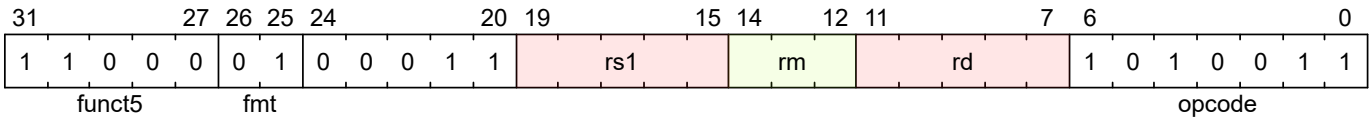
Description

Implementation

```
x[rd] = f64->s64(f[rs1])
```

9.2. fcvt.lu.d

Encoding



Format

fcvt.lu.d rd,rs1

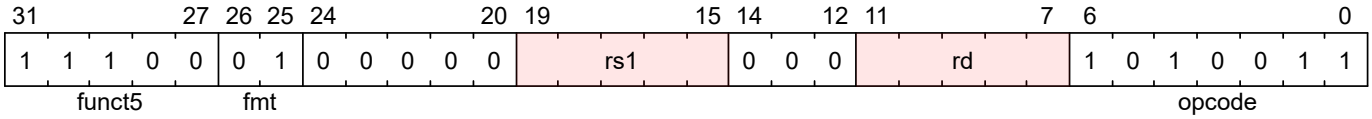
Description

Implementation

x[rd] = f64→u64(f[rs1])

9.3. fmv.x.d

Encoding



Format

fmv.x.d rd,rs1

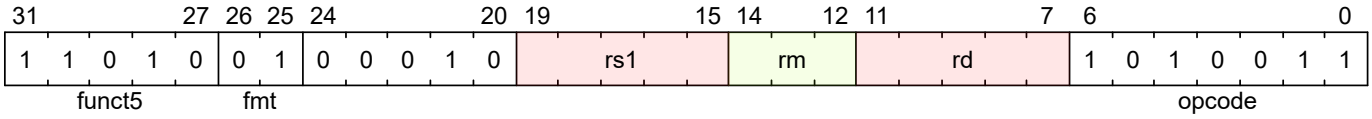
Description

Implementation

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

9.4. fcvt.d.l

Encoding



Format

fcvt.d.l rd,rs1

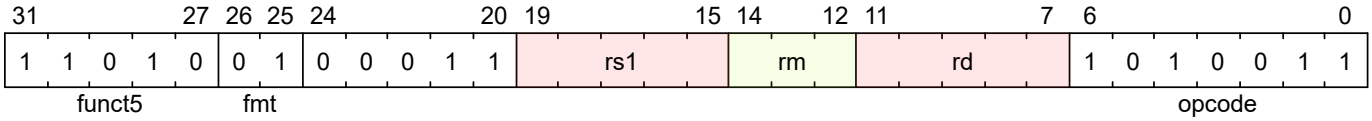
Description

Implementation

```
f[rd] = s64->f64(x[rs1])
```

9.5. fcvt.d.lu

Encoding



Format

fcvt.d.lu rd,rs1

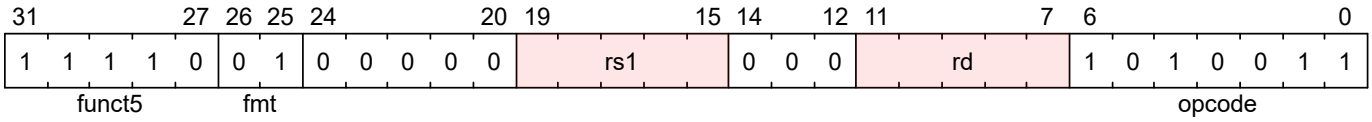
Description

Implementation

```
f[rd] = u64->f64(x[rs1])
```

9.6. fmv.d.x

Encoding



Format

fmv.d.x rd,rs1

Description

Implementation

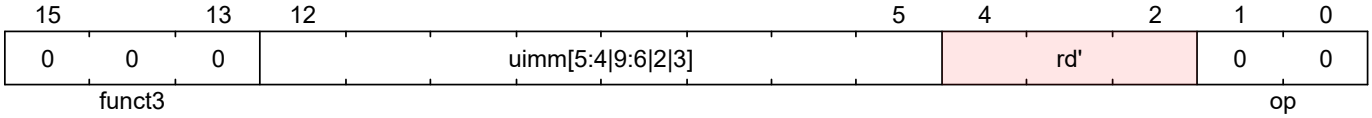
```
f[rd] = x[rs1][63:0]
```

---

10. RV32C, RV64C Instructions

10.1. c.addi4spn

Encoding



Format

```
c.addi4spn rd',uimm
```

Description

Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer, x2, and writes the result to rd'. This instruction is used to generate pointers to stack-allocated variables, and expands to addi rd', x2, nzuimm[9:2].

Implementation

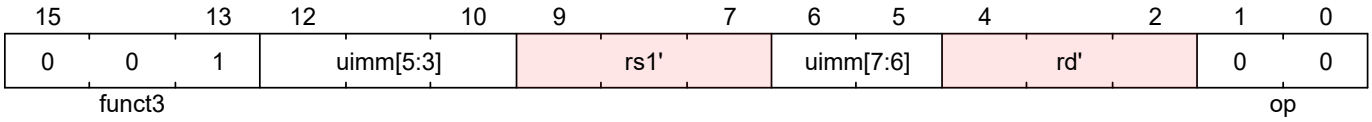
```
x[8+rd'] = x[2] + uimm
```

Expansion

```
addi x2,x2,nzimm[9:4]
```

10.2. c.fld

Encoding



Format

```
c.fld rd',uimm(rs1')
```

Description

Load a double-precision floating-point value from memory into floating-point register rd'. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1'.

Implementation

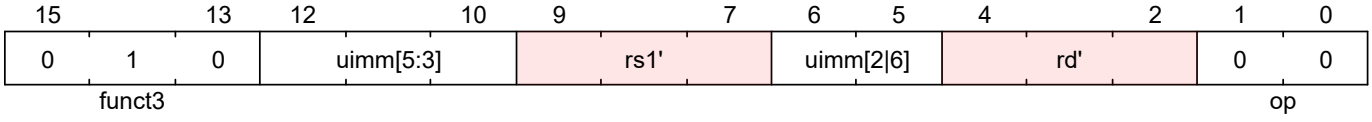
```
f[8+rd'] = M[x[8+rs1'] + uimm][63:0]
```

Expansion

```
fld rd',offset[7:3](rs1')
```

10.3. c.lw

Encoding



Format

```
c.lw rd',uimm(rs1')
```

Description

Load a 32-bit value from memory into register rd'. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.

Implementation

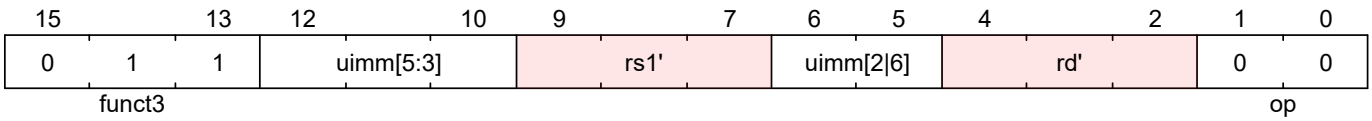
```
x[8+rd'] = sext(M[x[8+rs1'] + uimm][31:0])
```

Expansion

```
lw rd',offset[6:2](rs1')
```

10.4. c.flw

Encoding



Format

```
c.flw rd',uimm(rs1')
```

Description

Load a single-precision floating-point value from memory into floating-point register rd'. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.

Implementation

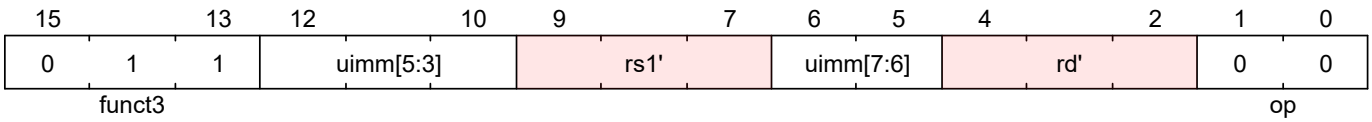
```
f[8+rd'] = M[x[8+rs1'] + uimm][31:0]
```

Expansion

```
lw rd',offset[6:2](rs1')
```

10.5. c.ld

Encoding



Format

```
c.ld rd',uimm(rs1')
```

Description

Load a 64-bit value from memory into register rd'. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1'.

Implementation

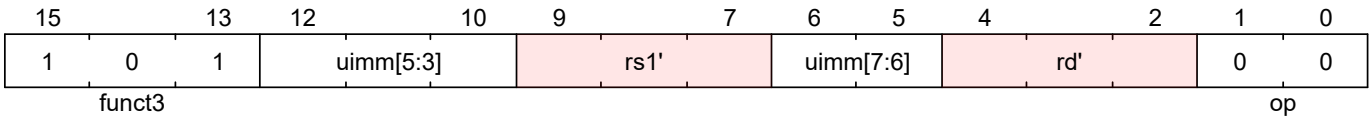
```
x[8+rd'] = M[x[8+rs1'] + uimm][63:0]
```

Expansion

```
ld rd', offset[7:3](rs1')
```

=== c.fsd

Encoding



Format

```
c.fsd rd',uimm(rs1')
```

Description

Store a double-precision floating-point value in floating-point register rs2' to memory. It computes an effective address by adding the zeroextended offset, scaled by 8, to the base address in register rs1'.

Implementation

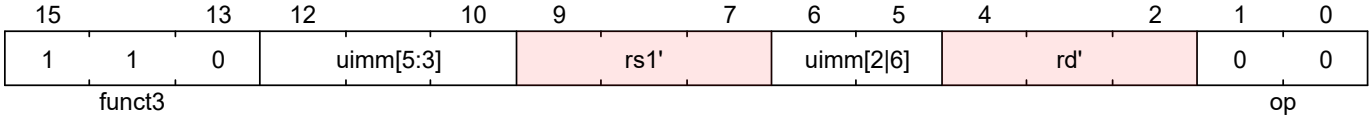
```
  M[x[8+rs1'] + uimm][63:0] = f[8+rs2']
---
```

Expansion

```
fsd rs2',offset[7:3](rs1')
```

10.6. c.sw

Encoding



Format

```
c.sw rd',uimm(rs1')
```

Description

Store a 32-bit value in register rs2' to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.

Implementation

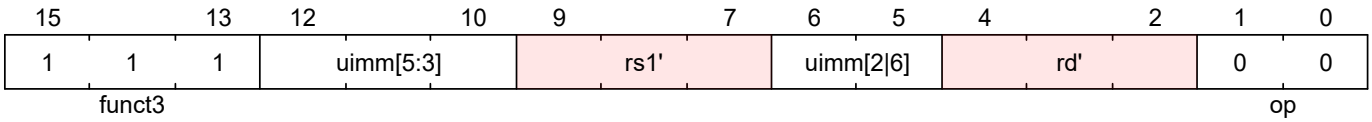
```
M[x[8+rs1'] + uimm][31:0] = x[8+rs2']
```

Expansion

```
sw rs2',offset[6:2](rs1')
```

10.7. c.fsw

Encoding



Format

```
c.fsw rd',uimm(rs1')
```

Description

Store a single-precision floating-point value in floatingpoint register rs2' to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.

Implementation

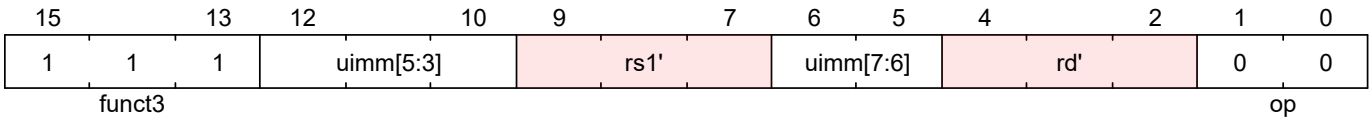
```
M[x[8+rs1'] + uimm][31:0] = f[8+rs2']
```

Expansion

```
fsw rs2', offset[6:2](rs1')
```

10.8. c.sd

Encoding



Format

```
c.sd rd',uimm(rs1')
```

Description

Store a 64-bit value in register rs2' to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1'.

Implementation

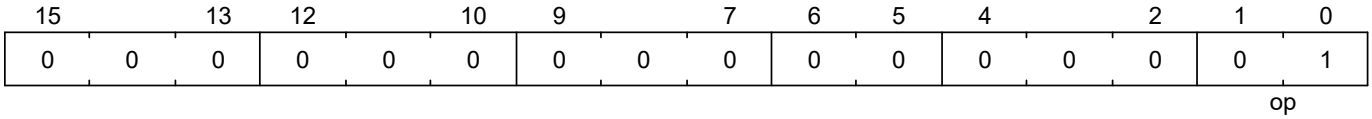
```
M[x[8+rs1'] + uimm][63:0] = x[8+rs2']
```

Expansion

```
sd rs2', offset[7:3](rs1')
```

10.9. c.nop

Encoding



Format

```
c.nop
```

Description

Does not change any user-visible state, except for advancing the pc.

Implementation

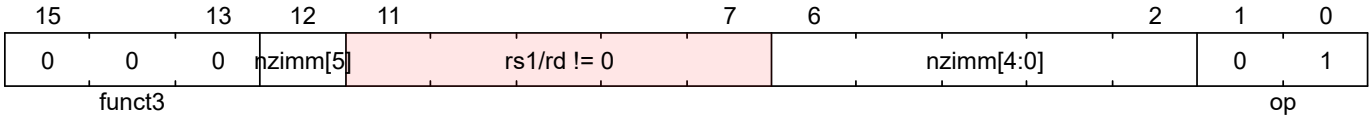
```
None
```

Expansion

```
addi x0, x0, 0
```

10.10. c.addi

Encoding



Format

```
c.addi rd,u[12:12]|u[6:2]
```

Description

Add the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd.

Implementation

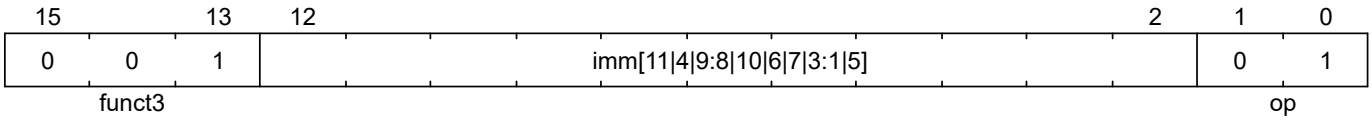
```
x[rd] = x[rd] + sext(imm)
```

Expansion

```
addi rd, rd, nzimm[5:0]
```

10.11. c.jal

Encoding



Format

```
c.jal offset
```

Description

Jump to address and place return address in rd.

Implementation

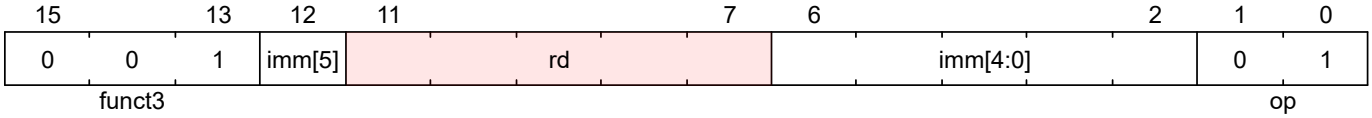
```
x[1] = pc+2; pc += sext(offset)
```

Expansion

```
jal x1, offset[11:1]
```

10.12. c.addiw

Encoding



Format

```
c.addiw rd,imm
```

Description

Add the non-zero sign-extended 6-bit immediate to the value in register rd then produce 32-bit result, then sign-extends result to 64 bits.

Implementation

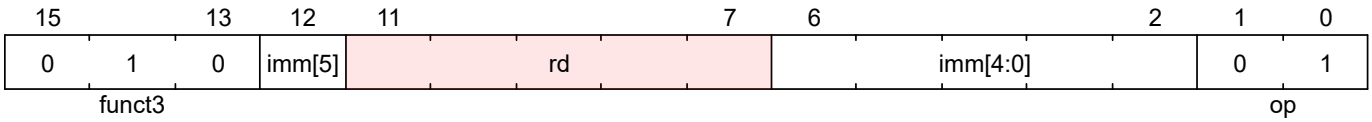
```
x[rd] = sext((x[rd] + sext(imm))[31:0])
```

Expansion

```
addiw rd,rd,imm[5:0]
```

10.13. c.li

Encoding



Format

```
c.li rd,uimm
```

Description

Load the sign-extended 6-bit immediate, imm, into register rd. C.LI is only valid when rd!=x0.

Implementation

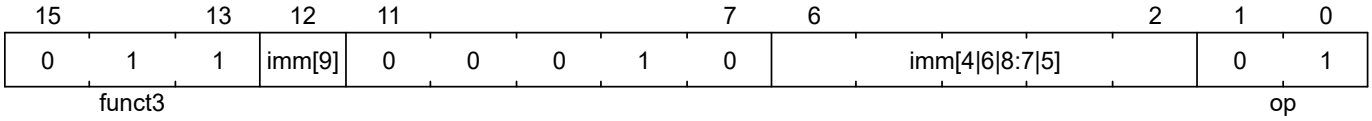
```
x[rd] = sext(imm)
```

Expansion

```
addi rd,x0,imm[5:0]
```

10.14. c.addi16sp

Encoding



Format

```
c.addi16sp imm
```

Description

Add the non-zero sign-extended 6-bit immediate to the value in the stack pointer (sp=x2), where the immediate is scaled to represent multiples of 16 in the range (-512,496).

Implementation

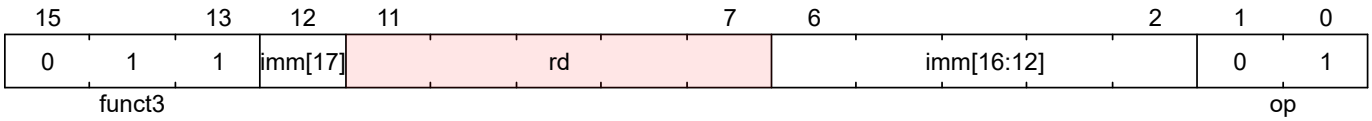
```
x[2] = x[2] + sext(imm)
```

Expansion

```
addi x2,x2, nzimm[9:4]
```

10.15. c.lui

Encoding



Format

```
c.lui rd,uimm
```

Description

Implementation

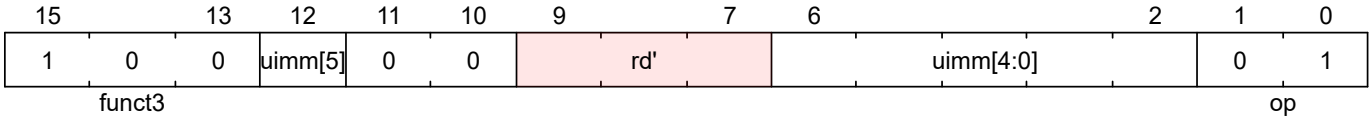
```
x[rd] = sext(imm[17:12] << 12)
```

Expansion

```
lui rd,nzuimm[17:12]
```

10.16. c.srli

Encoding



Format

```
c.srli rd',uimm
```

Description

Perform a logical right shift of the value in register rd' then writes the result to rd'. The shift amount is encoded in the shamt field, where shamt[5] must be zero for RV32C.

Implementation

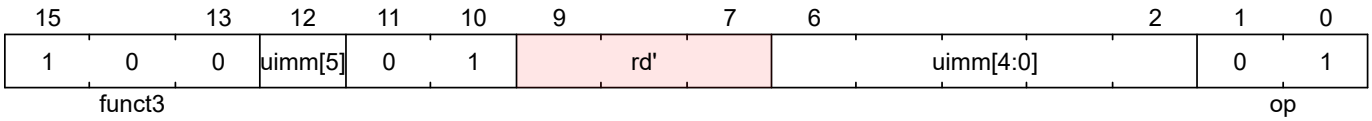
```
x[8+rd'] = x[8+rd'] >>u uimm
```

Expansion

```
srli rd',rd',64
```

10.17. c.srai

Encoding



Format

```
c.srai rd',uimm
```

Description

Perform a arithmetic right shift of the value in register rd' then writes the result to rd'. The shift amount is encoded in the shamt field, where shamt[5] must be zero for RV32C.

Implementation

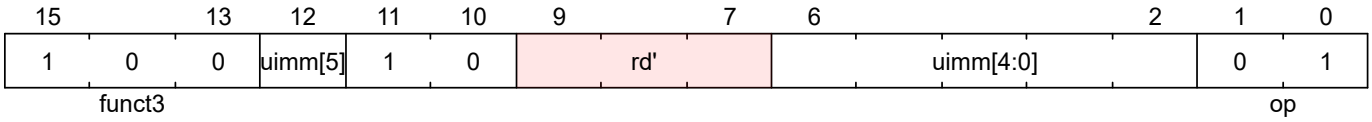
```
x[8+rd'] = x[8+rd'] >>s uimm
```

Expansion

```
srai rd',rd',shamt[5:0]
```

10.18. c.andi

Encoding



Format

```
c.andi rd',uimm
```

Description

Compute the bitwise AND of of the value in register rd' and the sign-extended 6-bit immediate, then writes the result to rd'.

Implementation

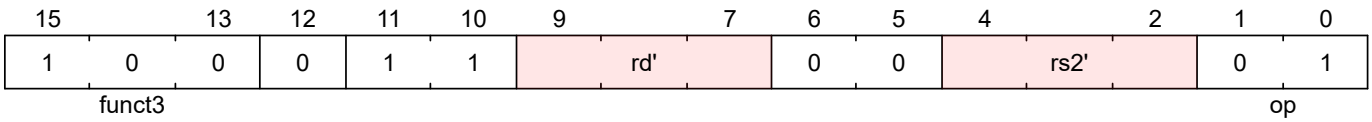
```
x[8+rd'] = x[8+rd'] & sext(imm)
```

Expansion

```
andi rd',rd',imm[5:0]
```

10.19. c.sub

Encoding



Format

```
c.sub rd',rd'
```

Description

Subtract the value in register rs2' from the value in register rd', then writes the result to register rd'.

Implementation

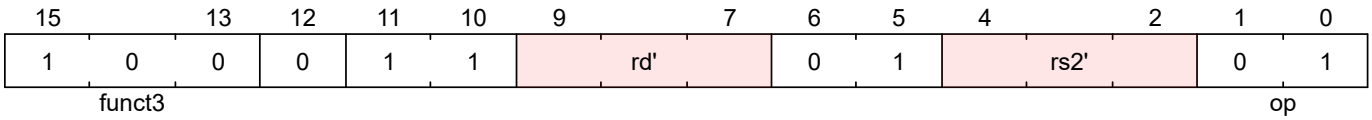
```
x[8+rd'] = x[8+rd'] - x[8+rs2']
```

Expansion

```
sub rd',rd',rs2'
```

10.20. c.xor

Encoding



Format

```
c.xor rd',rd'
```

Description

Compute the bitwise XOR of the values in registers rd' and rs2', then writes the result to register rd'.

Implementation

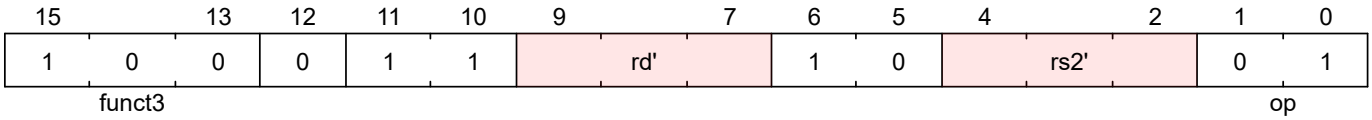
```
x[8+rd'] = x[8+rd'] ^ x[8+rs2']
```

Expansion

```
xor rd',rd',rs2'
```

10.21. c.or

Encoding



Format

```
c.or rd',rd'
```

Description

Compute the bitwise OR of the values in registers rd' and rs2', then writes the result to register rd'.

Implementation

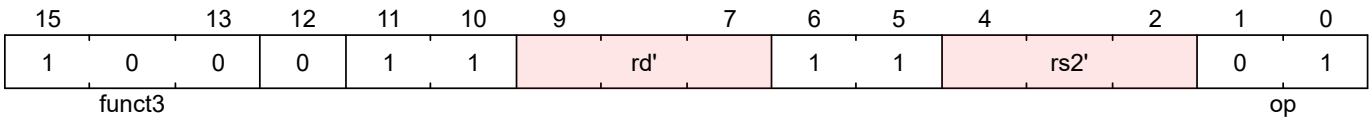
```
x[8+rd'] = x[8+rd'] | x[8+rs2']
```

Expansion

```
 or rd',rd',rs2
```

10.22. c.and

Encoding



Format

```
c.and rd',rd'
```

Description

Compute the bitwise AND of the values in registers rd' and rs2', then writes the result to register rd'.

Implementation

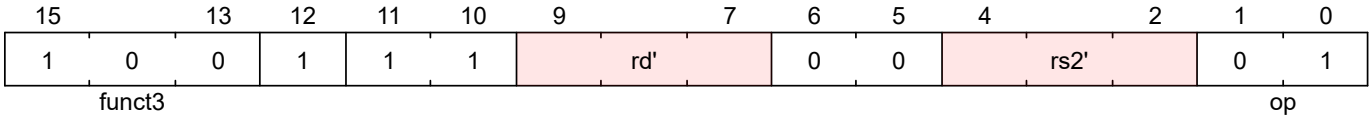
```
x[8+rd'] = x[8+rd'] & x[8+rs2']
```

Expansion

```
and rd',rd',rs2'
```

10.23. c.subw

Encoding



Format

```
c.subw rd',rs2'
```

Description

Subtract the value in register rs2' from the value in register rd', then sign-extends the lower 32 bits of the difference before writing the result to register rd'.

Implementation

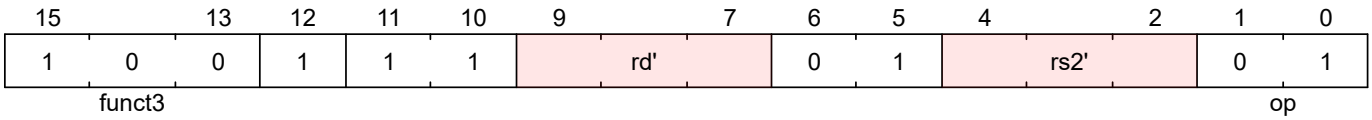
```
x[8+rd'] = sext((x[8+rd'] - x[8+rs2'])[31:0])
```

Expansion

```
subw rd',rd',rs2'
```

10.24. c.addw

Encoding



Format

```
c.addw rd',rs2'
```

Description

Add the value in register rs2' from the value in register rd', then sign-extends the lower 32 bits of the difference before writing the result to register rd'.

Implementation

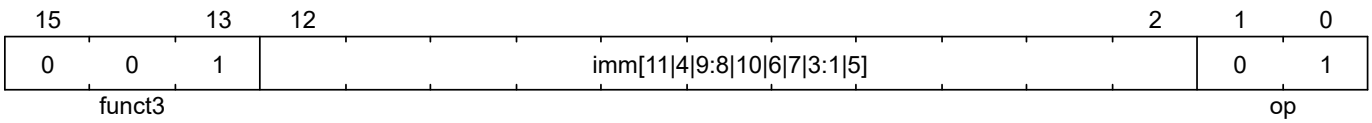
```
x[8+rd'] = sext((x[8+rd'] + x[8+rs2'])[31:0])
```

Expansion

```
addw rd',rd',rs2'
```

10.25. c.j

Encoding



Format

```
c.j offset
```

Description

Unconditional control transfer.

Implementation

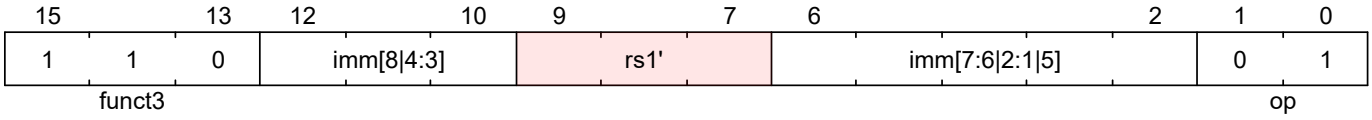
```
pc += sext(offset)
```

Expansion

```
jal x0,offset[11:1]
```

10.26. c.beqz

Encoding



Format

```
c.beqz rs1',offset
```

Description

Take the branch if the value in register rs1' is zero.

Implementation

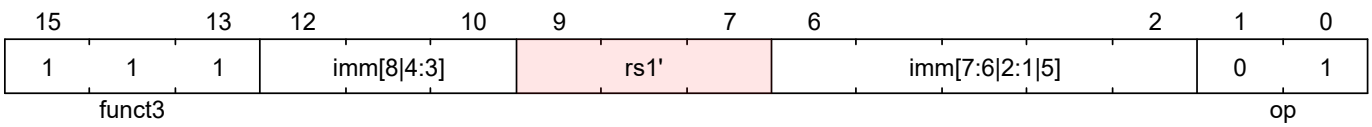
```
if (x[8+rs1'] == 0) pc += sext(offset)
```

Expansion

```
beq rs1',x0,offset[8:1]
```

10.27. c.bnez

Encoding



Format

```
c.bnez rs1',offset
```

Description

Take the branch if the value in register rs1' is not zero.

Implementation

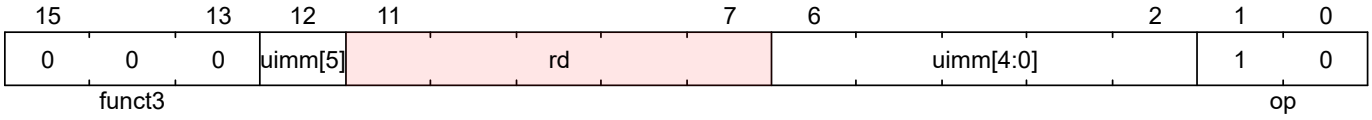
```
if (x[8+rs1'] != 0) pc += sext(offset)
```

Expansion

```
bne rs1',x0,offset[8:1]
```

10.28. c.slli

Encoding



Format

```
c.slli rd,uimm
```

Description

Perform a logical left shift of the value in register rd then writes the result to rd. The shift amount is encoded in the shamt field, where shamt[5] must be zero for RV32C.

Implementation

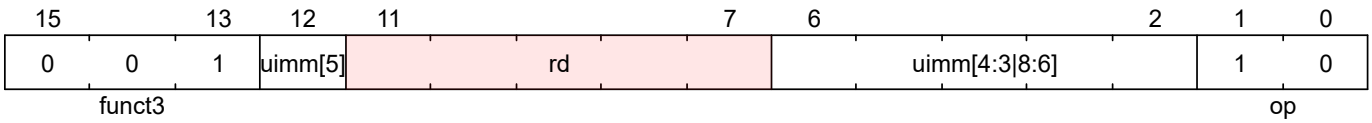
```
x[rd] = x[rd] << uimm
```

Expansion

```
slli rd,rd,shamt[5:0]
```

10.29. c.fldsp

Encoding



Format

```
c.fldsp rd,uimm(x2)
```

Description

Load a double-precision floating-point value from memory into floating-point register rd. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2.

Implementation

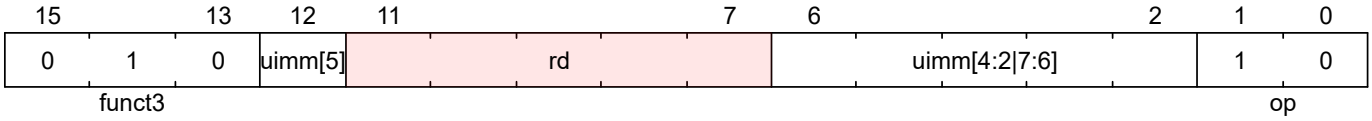
```
f[rd] = M[x[2] + uimm][63:0]
```

Expansion

```
fld rd,offset[8:3](x2)
```

10.30. c.lwsp

Encoding



Format

```
c.lwsp rd,uimm(x2)
```

Description

Load a 32-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2.

Implementation

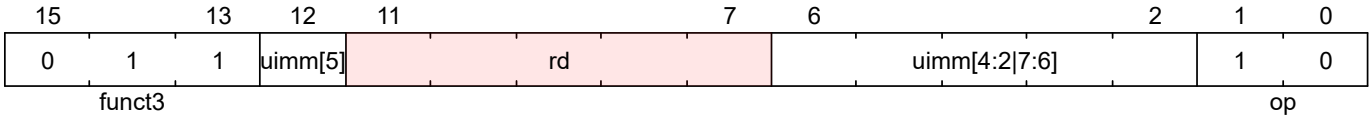
```
x[rd] = sext(M[x[2] + uimm][31:0])
```

Expansion

```
lw rd,offset[7:2](x2)
```

10.31. c.flwsp

Encoding



Format

```
c.flwsp rd,uimm(x2)
```

Description

Load a single-precision floating-point value from memory into floating-point register rd. It computes its effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2.

Implementation

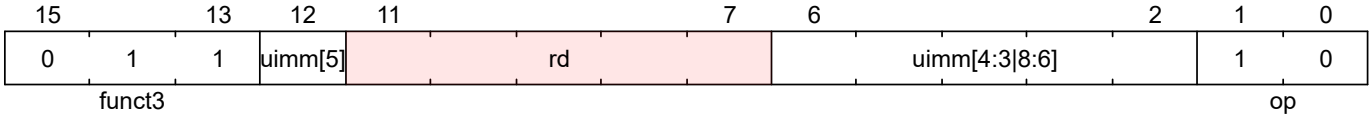
```
f[rd] = M[x[2] + uimm][31:0]
```

Expansion

```
flw rd,offset[7:2](x2)
```

10.32. c.ldsp

Encoding



Format

```
c.ldsp rd,uimm(x2)
```

Description

Load a 64-bit value from memory into register rd. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2.

Implementation

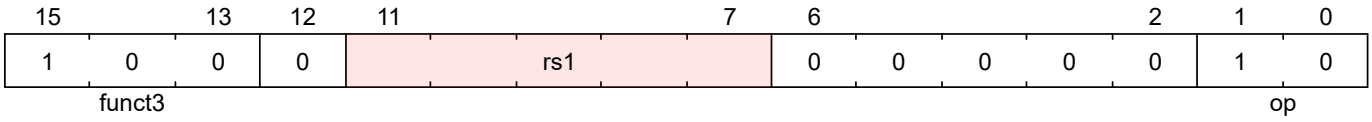
```
x[rd] = M[x[2] + uimm][63:0]
```

Expansion

```
ld rd,offset[8:3](x2)
```

10.33. c.jr

Encoding



Format

```
c.jr rs1
```

Description

Performs an unconditional control transfer to the address in register rs1.

Implementation

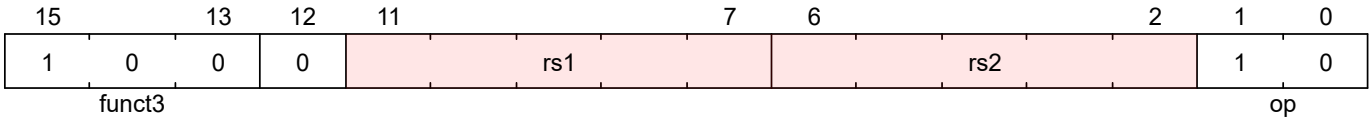
```
pc = x[rs1]
```

Expansion

```
jalr x0,rs1,0
```

10.34. c.mv

Encoding



Format

```
c.mv rd,rs2'
```

Description

Copy the value in register rs2 into register rd.

Implementation

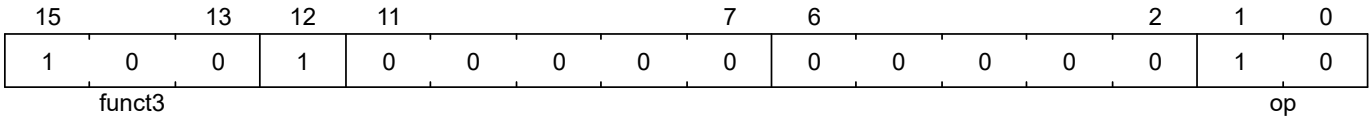
```
x[rd] = x[rs2]
```

Expansion

```
add rd, x0, rs2
```

10.35. c.ebreak

Encoding



Format

```
c.ebreak
```

Description

Cause control to be transferred back to the debugging environment.

Implementation

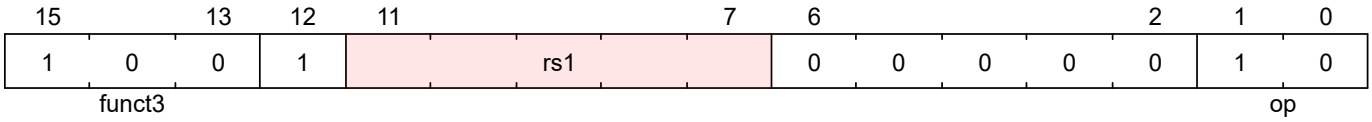
```
RaiseException(Breakpoint)
```

Expansion

```
ebreak
```

10.36. c.jalr

Encoding



Format

```
c.jalr rd
```

Description

Jump to address and place return address in rd.

Implementation

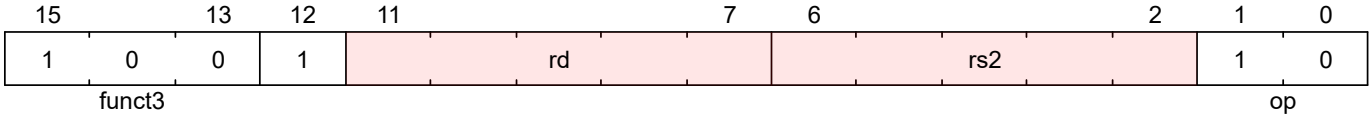
```
t = pc+2; pc = x[rs1]; x[1] = t
```

Expansion

```
jalr x1,rs1,0
```

10.37. c.add

Encoding



Format

```
c.add rd,rs2'
```

Description

Add the values in registers rd and rs2 and writes the result to register rd.

Implementation

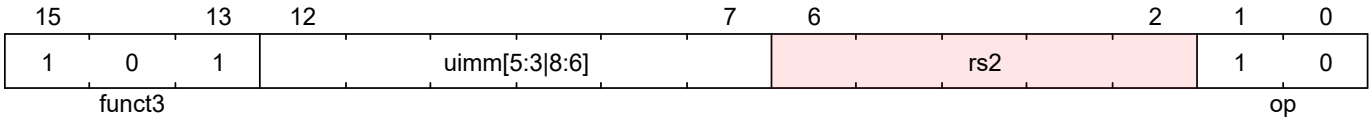
```
x[rd] = x[rd] + x[rs2]
```

Expansion

```
add rd,rd,rs2
```

10.38. c.fsdsp

Encoding



Format

```
c.fsdsp rs2,uimm(x2)
```

Description

Store a double-precision floating-point value in floating-point register rs2 to memory. It computes an effective address by adding the zeroextended offset, scaled by 8, to the stack pointer, x2.

Implementation

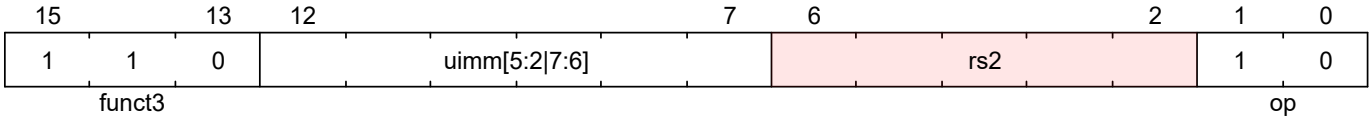
```
M[x[2] + uimm][63:0] = f[rs2]
```

Expansion

```
fsd rs2,offset[8:3](x2)
```

10.39. c.swsp

Encoding



Format

```
c.swsp rs2,uimm(x2)
```

Description

Store a 32-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2.

Implementation

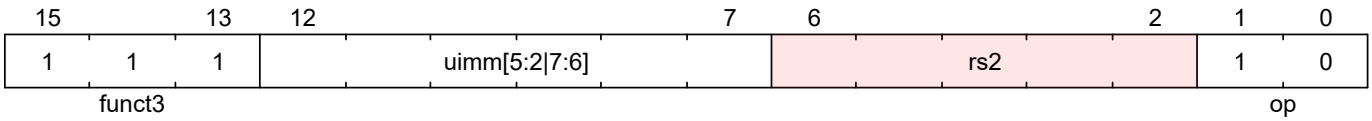
```
M[x[2] + uimm][31:0] = x[rs2]
```

Expansion

```
sw rs2,offset[7:2](x2)
```

10.40. c.fswsp

Encoding



Format

```
c.fswsp rs2,uimm(rs2)
```

Description

Store a single-precision floating-point value in floating-point register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2.

Implementation

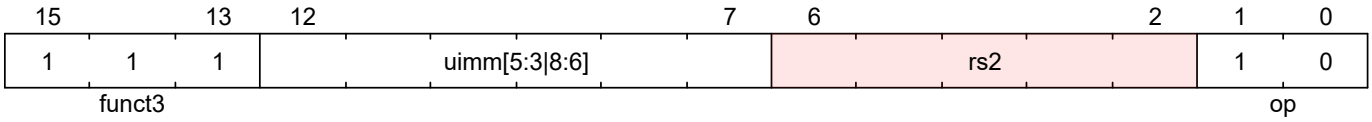
```
M[x[2] + uimm][31:0] = f[rs2]
```

Expansion

```
fsw rs2,offset[7:2](x2)
```

10.41. c.sdsp

Encoding



Format

```
c.sdsp rs2,uimm(x2)
```

Description

Store a 64-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2.

Implementation

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

Expansion

```
sd rs2,offset[8:3](x2)
```

---

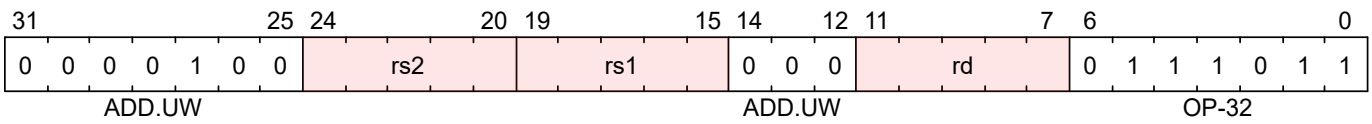
11. RVB Instructions

11.1. add.uw

Format

```
add.uw rd, rs1, rs2
```

Encoding



Description

This instruction performs an XLEN-wide addition between rs2 and the zero-extended least-significant word of rs1.

Implementation

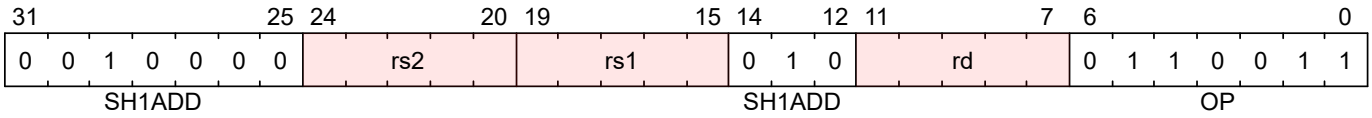
```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);
X(rd) = base + index;
```

11.2. sh1add

Format

```
sh1add rd, rs1, rs2
```

Encoding



Description

This instruction shifts rs1 to the left by 1 bit and adds it to rs2.

Implementation

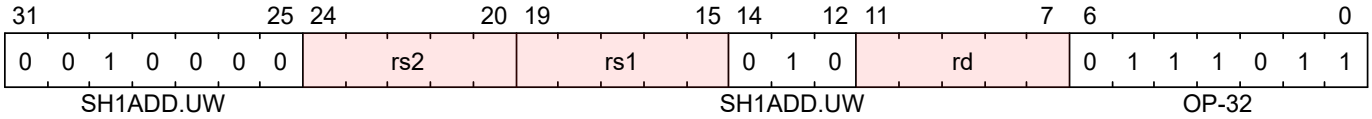
```
X(rd) = X(rs2) + (X(rs1) << 1);
```

11.3. sh1add.uw

Format

```
sh1add.uw rd, rs1, rs2
```

Encoding



Description

This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 1 place.

Implementation

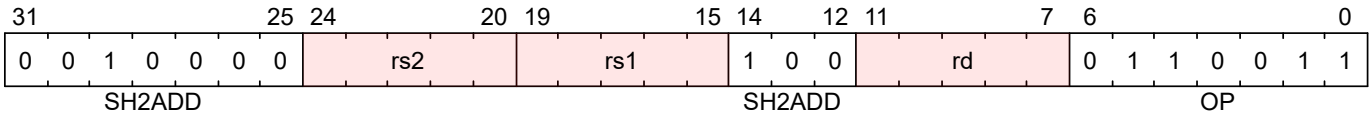
```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 1);
```

11.4. sh2add

Format

```
sh2add rd, rs1, rs2
```

Encoding



Description

This instruction shifts rs1 to the left by 2 places and adds it to rs2.

Implementation

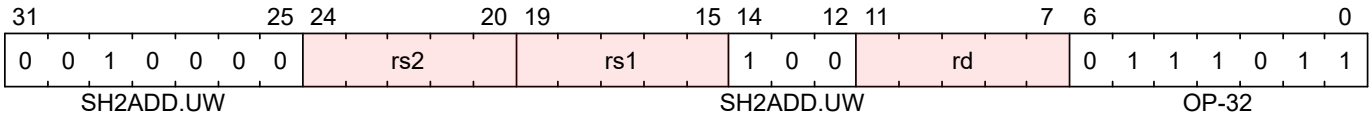
```
X(rd) = X(rs2) + (X(rs1) << 2);
```

11.5. sh2add.uw

Format

```
sh2add.uw rd, rs1, rs2
```

Encoding



Description

This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 2 places.

Implementation

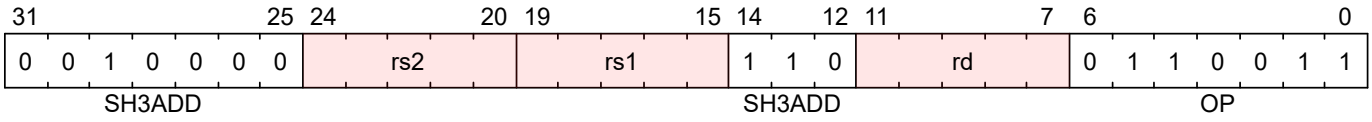
```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 2);
```

11.6. sh3add

Format

```
sh3add rd, rs1, rs2
```

Encoding



Description

This instruction shifts rs1 to the left by 3 places and adds it to rs2.

Implementation

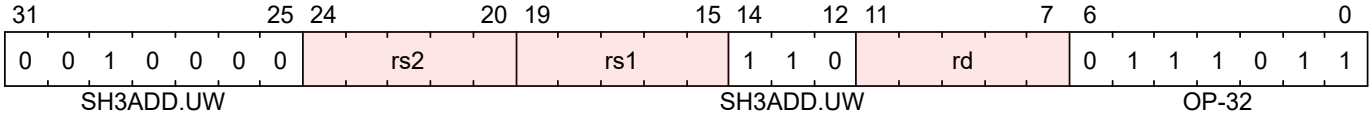
```
X(rd) = X(rs2) + (X(rs1) << 3);
```

11.7. sh3add.uw

Format

```
sh3add.uw rd, rs1, rs2
```

Encoding



Description

This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 3 places.

Implementation

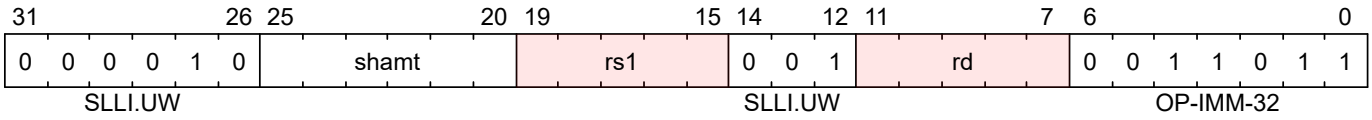
```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 3);
```

11.8. slli.uw

Format

```
slli.uw rd, rs1, shamt
```

Encoding



Description

This instruction takes the least-significant word of rs1, zero-extends it, and shifts it left by the immediate.

Implementation

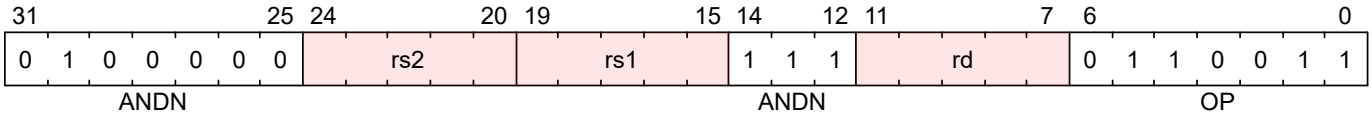
```
X(rd) = (EXTZ(X(rs)[31..0]) << shamt);
```

11.9. andn

Format

```
andn rd, rs1, rs2
```

Encoding



Description

This instruction performs the bitwise logical AND operation between rs1 and the bitwise inversion of rs2.

Implementation

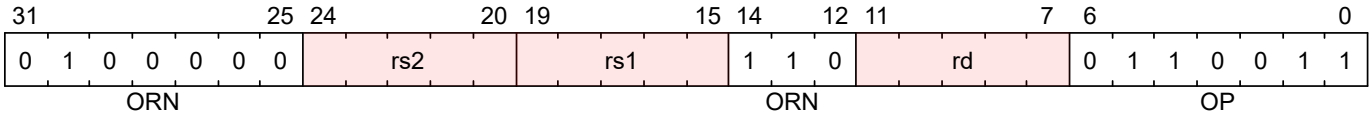
```
X(rd) = X(rs1) & ~X(rs2);
```

11.10. orn

Format

```
orn rd, rs1, rs2
```

Encoding



Description

This instruction performs the bitwise logical OR operation between rs1 and the bitwise inversion of rs2.

Implementation

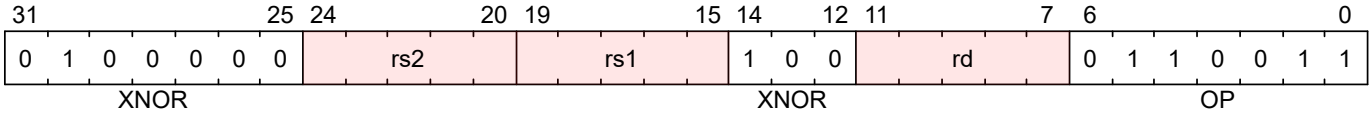
```
X(rd) = X(rs1) | ~X(rs2);
```

11.11. xnor

Format

```
xnor rd, rs1, rs2
```

Encoding



Description

This instruction performs the bit-wise exclusive-NOR operation on rs1 and rs2.

Implementation

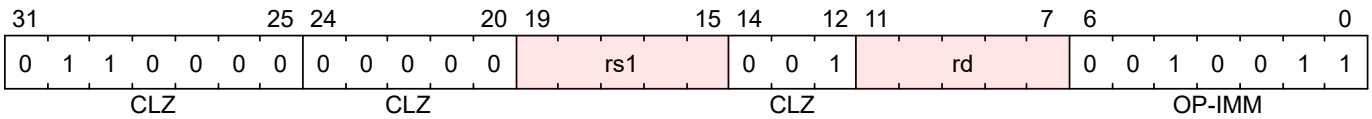
```
X(rd) = ~(X(rs1) ^ X(rs2));
```

11.12. clz

Format

```
clz rd, rs
```

Encoding



Description

This instruction counts the number of 0's before the first 1, starting at the most-significant bit (i.e., XLEN-1) and progressing to bit 0. Accordingly, if the input is 0, the output is XLEN, and if the

most-significant bit of the input is a 1, the output is 0.

Implementation

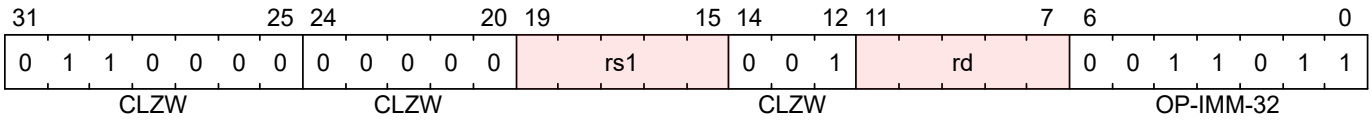
```
val HighestSetBit : forall ('N : Int), 'N >= 0. bits('N) -> int

function HighestSetBit x = {
  foreach (i from (xlen - 1) to 0 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return -1;
}

let rs = X(rs);
X[rd] = (xlen - 1) - HighestSetBit(rs);
```

11.13. clzw

Format

```
clzw rd, rs
```

Encoding



Description

This instruction counts the number of 0's before the first 1 starting at bit 31 and progressing to bit 0. Accordingly, if the least-significant word is 0, the output is 32, and if the most-significant bit of the word (i.e., bit 31) is a 1, the output is 0.

Implementation

```
val HighestSetBit32 : forall ('N : Int), 'N >= 0. bits('N) -> int

function HighestSetBit32 x = {
  foreach (i from 31 to 0 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return -1;
}

let rs = X(rs);
X[rd] = 31 - HighestSetBit(rs);
```

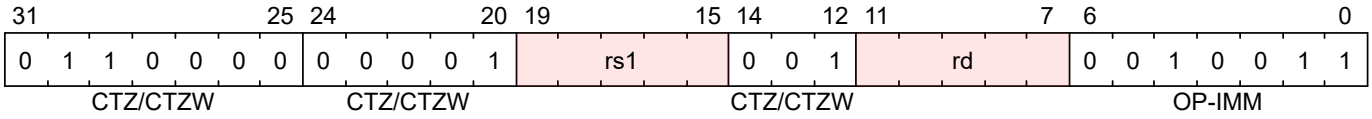
Implementation

=== ctz

Format

```
ctz rd, rs
```

Encoding



Description

This instruction counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit (i.e., XLEN-1). Accordingly, if the input is 0, the output is XLEN, and if the least-significant bit of the input is a 1, the output is 0.

Implementation

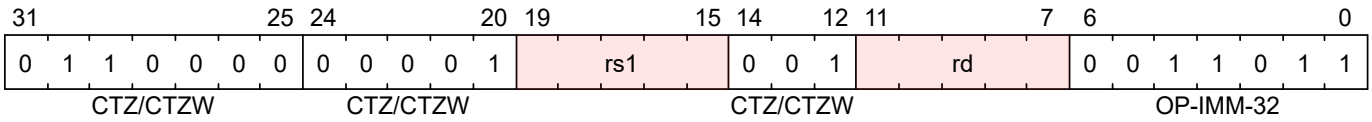
```
val LowestSetBit : forall ('N : Int), 'N >= 0. bits('N) -> int

function LowestSetBit x = {
  foreach (i from 0 to (xlen - 1) by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return xlen;
}

let rs = X(rs);
X[rd] = LowestSetBit(rs);
```

11.14. ctzw

Format

```
ctzw rd, rs
```

Encoding



Description

This instruction counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit of the least-significant word (i.e., 31). Accordingly, if the least-significant word is 0, the output is 32, and if the least-significant bit of the input is a 1, the output is 0.

Implementation

```
val LowestSetBit32 : forall ('N : Int), 'N >= 0. bits('N) -> int

function LowestSetBit32 x = {
  foreach (i from 0 to 31 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return 32;
```

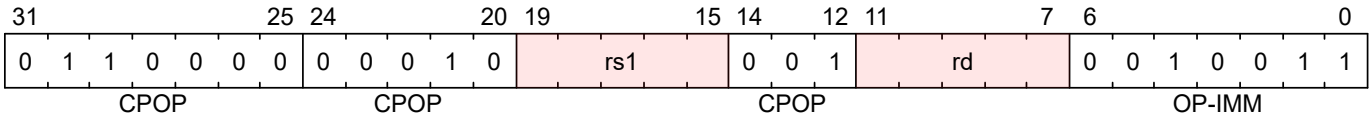
```
}

let rs = X(rs);
X[rd] = LowestSetBit32(rs);
```

11.15. cpop

Format

```
cpop rd, rs
```

Encoding



Description

This instructions counts the number of 1's (i.e., set bits) in the source register.

Implementation

```
let bitcount = 0;
let rs = X(rs);

foreach (i from 0 to (xlen - 1) in inc)
    if rs[i] == 0b1 then bitcount = bitcount + 1 else ();

X[rd] = bitcount
```

Note

Software Hint

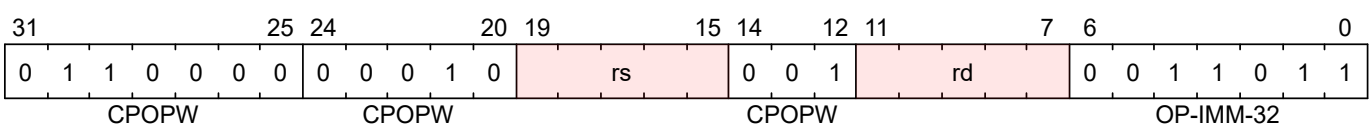
This operations is known as population count, popcount, sideways sum, bit summation, or Hamming weight.

The GCC builtin function `__builtin_popcount (unsigned int x)` is implemented by cpop on RV32 and by **cpopw** on RV64. The GCC builtin function `__builtin_popcountl (unsigned long x)` for LP64 is implemented by **cpop** on RV64.

11.16. cpopw

Format

```
cpopw rd, rs
```



Description

This instructions counts the number of 1's (i.e., set bits) in the least-significant word of the source register.

Implementation

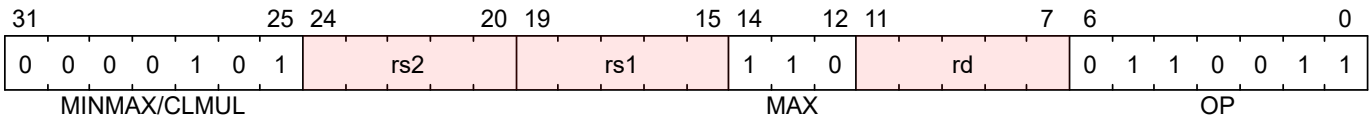
```
let bitcount = 0;
let val = X(rs);

foreach (i from 0 to 31 in inc)
    if val[i] == 0b1 then bitcount = bitcount + 1 else ();

X[rd] = bitcount
```

11.17. max

Format

```
max rd, rs1, rs2
```

Encoding



Description

This instruction returns the larger of two signed integers.

Implementation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_s rs2_val
             then rs2_val
             else rs1_val;

X(rd) = result;
```

Note

Software Hint

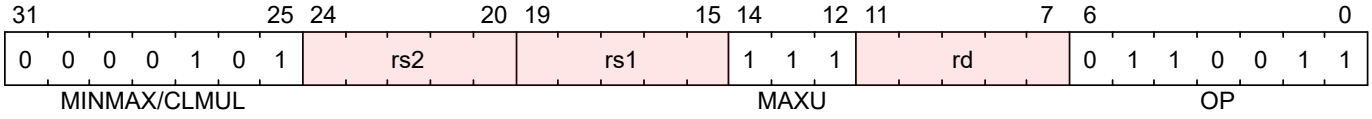
Calculating the absolute value of a signed integer can be performed using the following sequence: **neg rD,rS** followed by **max rD,rS,rD**. When using this common sequence, it is suggested that they are scheduled with no intervening instructions so that implementations that are so optimized can fuse them together.

11.18. maxu

Format

```
maxu rd, rs1, rs2
```

Encoding



Description

This instruction returns the larger of two unsigned integers.

Implementation

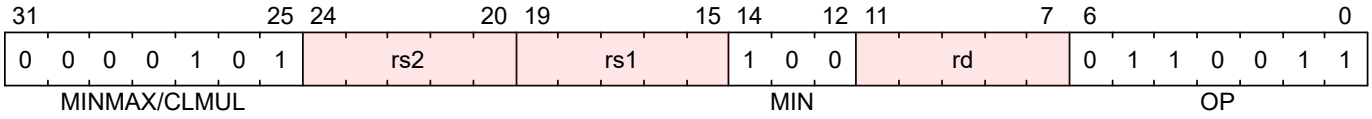
```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_u rs2_val
             then rs2_val
             else rs1_val;

X(rd) = result;
```

11.19. min

Format

```
min rd, rs1, rs2
```

Encoding



Description

This instruction returns the smaller of two signed integers.

Implementation

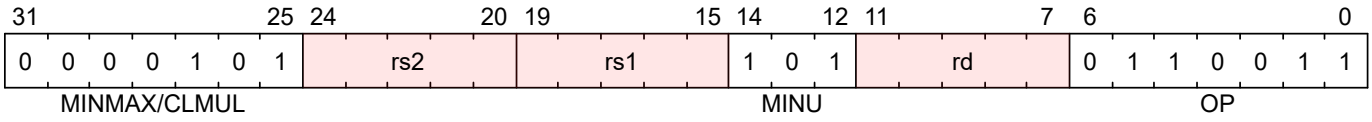
```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_s rs2_val
             then rs1_val
             else rs2_val;

X(rd) = result;
```

11.20. minu

Format

```
minu rd, rs1, rs2
```

Encoding



Description

This instruction returns the smaller of two unsigned integers.

Implementation

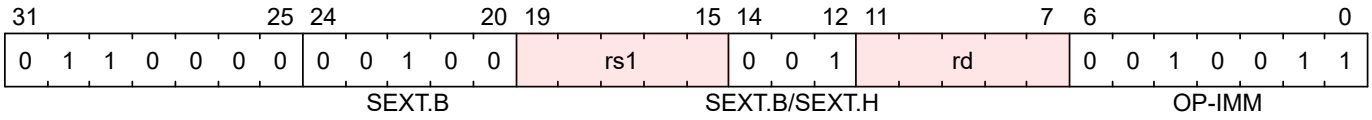
```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_u rs2_val
             then rs1_val
             else rs2_val;

X(rd) = result;
```

Implementation::=== sext.b

Format

```
sext.b rd, rs
```

Encoding



Description

This instruction sign-extends the least-significant byte in the source to XLEN by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

Implementation

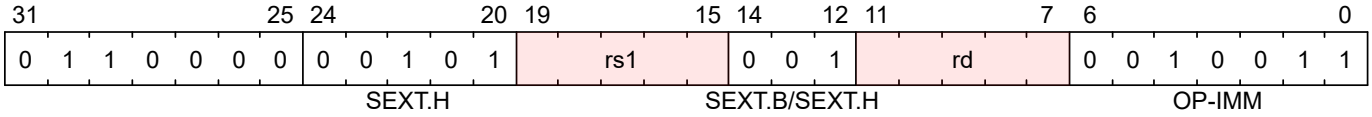
```
X(rd) = EXTS(X(rs)[7..0]);
```

11.21. sext.h

Format

```
sext.h rd, rs
```

Encoding



Description

This instruction sign-extends the least-significant halfword in rs to XLEN by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

Implementation

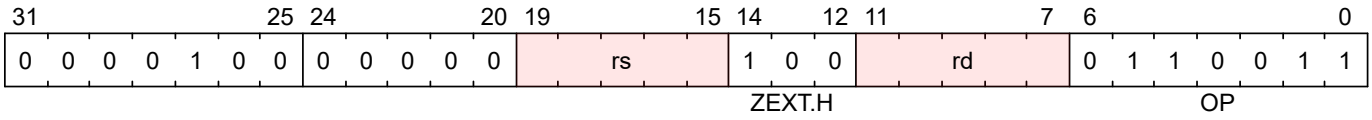
```
X(rd) = EXTS(X(rs)[15..0]);
```

11.22. zext.h

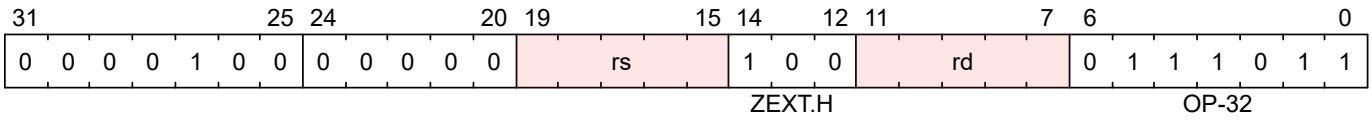
Format

```
zext.h rd, rs
```

Encoding (RV32)



Encoding (RV64)



Description

This instruction zero-extends the least-significant halfword of the source to XLEN by inserting 0's into all of the bits more significant than 15.

Implementation

```
X(rd) = EXTZ(X(rs)[15..0]);
```

Note

Note

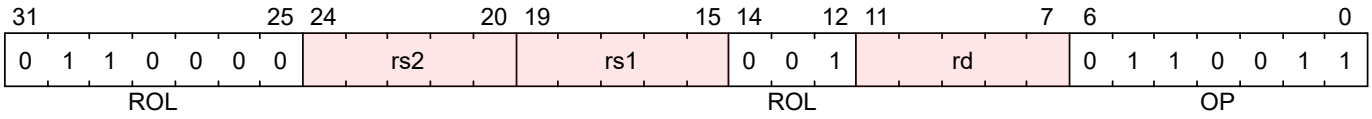
The **zext.h** mnemonic corresponds to different instruction encodings in RV32 and RV64.

11.23. rol

Format

```
rol rd, rs1, rs2
```

Encoding



Description

This instruction performs a rotate left of rs1 by the amount in least-significant log2(XLEN) bits of rs2.

Implementation

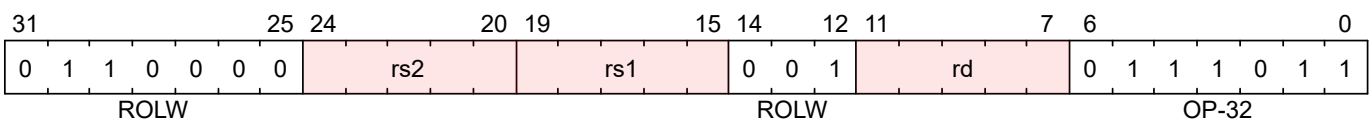
```
let shamt = if   xlen == 32
            then X(rs2)[4..0]
            else X(rs2)[5..0];
let result = (X(rs1) << shamt) | (X(rs1) >> (xlen - shamt));

X(rd) = result;
```

11.24. rolw

Format

```
rolw rd, rs1, rs2
```

Encoding



Description

This instruction performs a rotate left on the least-significant word of rs1 by the amount in least-significant 5 bits of rs2. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Implementation

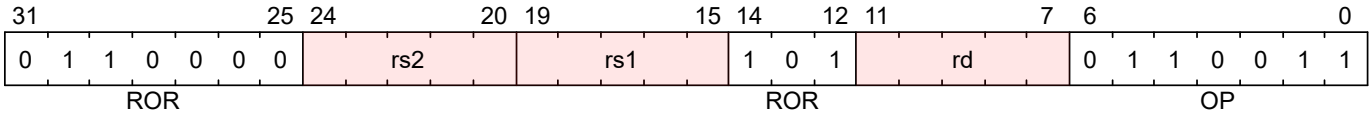
```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 << shamt) | (rs1 >> (32 - shamt));
X(rd) = EXTS(result[31..0]);
```

11.25. ror

Format

```
ror rd, rs1, rs2
```

Encoding



Description

This instruction performs a rotate right of rs1 by the amount in least-significant log2(XLEN) bits of rs2.

Implementation

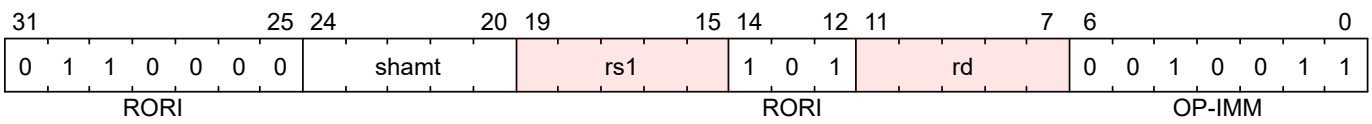
```
    let shamt = if   xlen == 32
                then X(rs2)[4..0]
                else X(rs2)[5..0];
    let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

    X(rd) = result;
```

11.26. rori

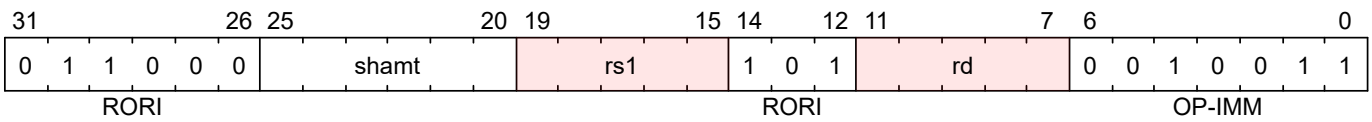
Format

```
rori rd, rs1, _shamt_
```

Encoding (RV32)



Encoding (RV64)



Description

This instruction performs a rotate right of rs1 by the amount in the least-significant log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Implementation

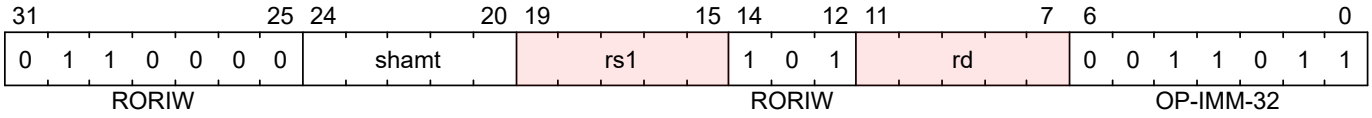
```
let shamt = if   xlen == 32
            then shamt[4..0]
            else shamt[5..0];
let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

X(rd) = result;
```

11.27. roriw

Format

```
roriw rd, rs1, _shamt_
```

Encoding



Description

This instruction performs a rotate right on the least-significant word of rs1 by the amount in the least-significant log2(XLEN) bits of *shamt*. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Implementation

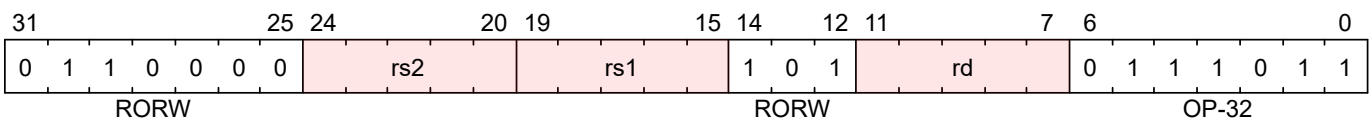
```
let rs1_data = EXTZ(X(rs1)[31..0];
let result = (rs1_data >> shamt) | (rs1_data << (32 - shamt));
X(rd) = EXTS(result[31..0]);
```

11.28. rorw

Format

```
rorw rd, rs1, rs2
```

Encoding



Description

This instruction performs a rotate right on the least-significant word of rs1 by the amount in least-significant 5 bits of rs2. The resultant word is sign-extended by copying bit 31 to all of the more-significant bits.

Implementation

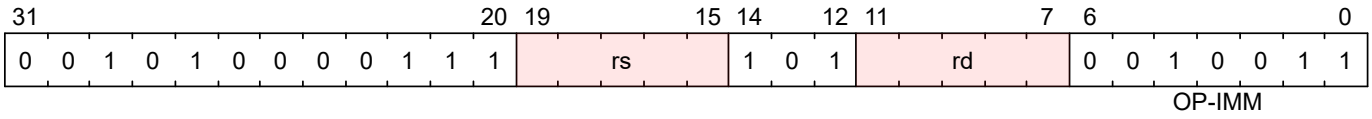
```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 >> shamt) | (rs1 << (32 - shamt));
X(rd) = EXTS(result);
```

11.29. orc.b

Format

```
orc.b rd, rs
```

Encoding



Description

Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result rd to all zeros if no bit within the respective byte of rs is set, or to all ones if any bit within the respective byte of rs is set.

Implementation

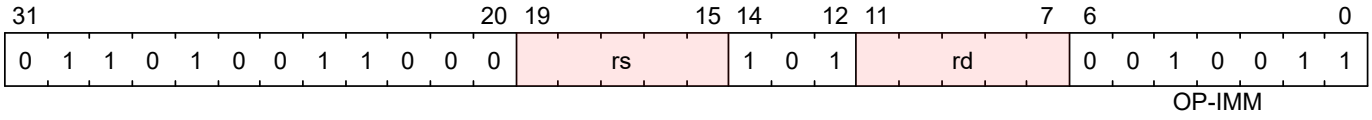
```
let input = X(rs);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 8) by 8) {
output[(i + 7)..i] = if   input[(i + 7)..i] == 0
                        then 0b00000000
                        else 0b11111111;
}

X[rd] = output;
```

11.30. rev8

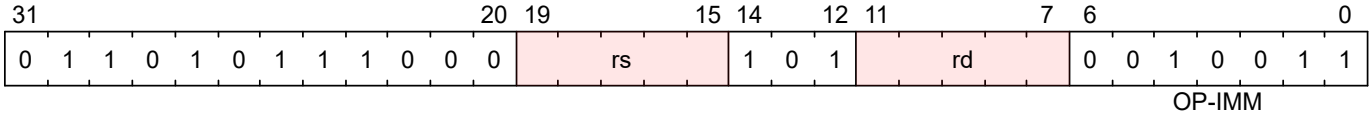
Format

```
rev8 rd, rs
```

Encoding (RV32)



Encoding (RV64)



Description

This instruction reverses the order of the bytes in rs.

Implementation

```
let input = X(rs);
let output : xlenbits = 0;
let j = xlen - 1;

foreach (i from 0 to (xlen - 8) by 8) {
   output[i..(i + 7)] = input[(j - 7)..j];
   j = j - 8;
}

X[rd] = output
```

> **Note**
>
> Note
>
> The **rev8** mnemonic corresponds to different instruction encodings in RV32 and RV64.

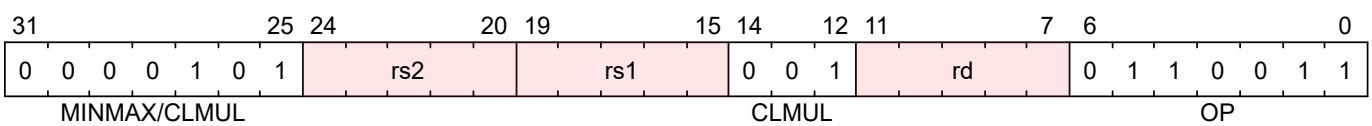
> **Note**
>
> Software Hint
>
> The byte-reverse operation is only available for the full register width. To emulate word-sized and halfword-sized byte-reversal, perform a `rev8 rd,rs` followed by a `srai rd,rd,K`, where K is XLEN-32 and XLEN-16, respectively.

11.31. clmul

Format

```
clmul rd, rs1, rs2
```

Encoding



Description

clmul produces the lower half of the 2·XLEN carry-less product.

Implementation

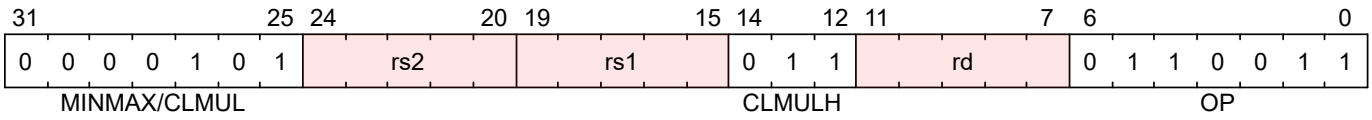
```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 1) by 1) {
   output = if   ((rs2_val >> i) & 1)
             then output ^ (rs1_val << i);
             else output;
}

X[rd] = output
```

11.32. clmulh

Format

```
clmulh rd, rs1, rs2
```

Encoding



Description

clmulh produces the upper half of the 2·XLEN carry-less product.

Implementation

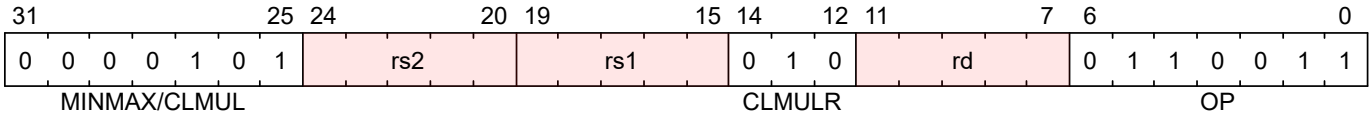
```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 1 to xlen by 1) {
   output = if   ((rs2_val >> i) & 1)
             then output ^ (rs1_val >> (xlen - i));
             else output;
}

X[rd] = output
```

11.33. clmulr

Format

```
clmulr rd, rs1, rs2
```

Encoding



Description

clmulr produces bits 2·XLEN−2:XLEN-1 of the 2·XLEN carry-less product.

Implementation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 1) by 1) {
   output = if   ((rs2_val >> i) & 1)
             then output ^ (rs1_val >> (xlen - i - 1));
             else output;
}

X[rd] = output
```

Note

Note

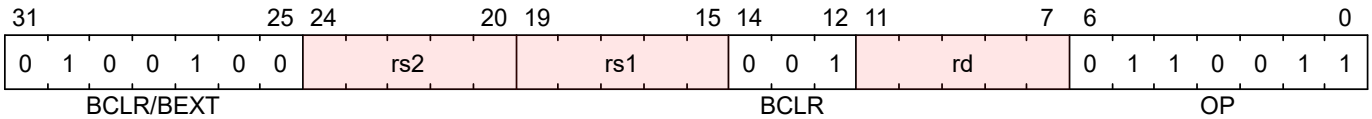
The **clmulr** instruction is used to accelerate CRC calculations. The **r** in the instruction's mnemonic stands for *reversed*, as the instruction is equivalent to bit-reversing the inputs, performing a **clmul**, then bit-reversing the output.

11.34. bclr

Format

```
bclr rd, rs1, rs2
```

Encoding



Description

This instruction returns rs1 with a single bit cleared at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

Implementation

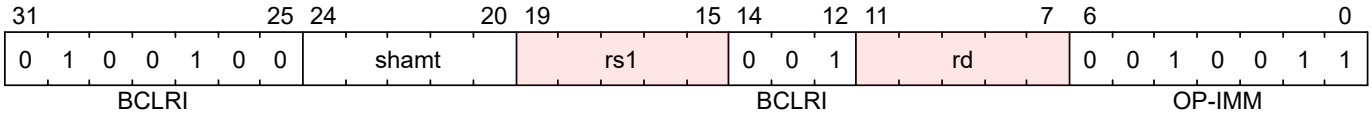
```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) & ~(1 << index)
```

11.35. bclri

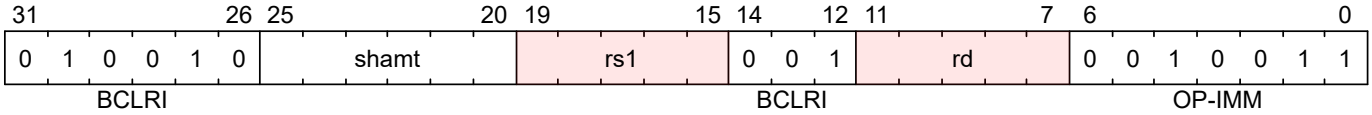
Format

```
bclri rd, rs1, shamt
```

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns rs1 with a single bit cleared at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Implementation

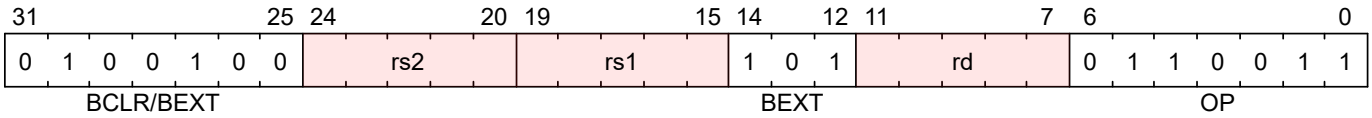
```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) & ~(1 << index)
```

11.36. bext

Format

```
bext rd, rs1, rs2
```

Encoding



Description

This instruction returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

Implementation

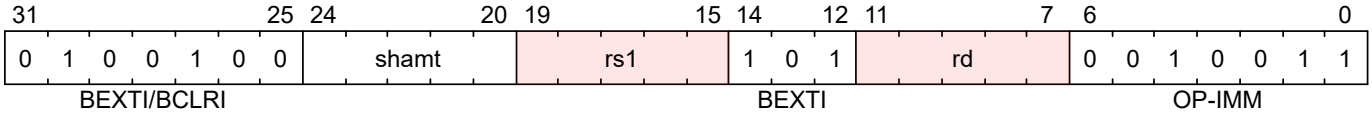
```
let index = X(rs2) & (XLEN - 1);
X(rd) = (X(rs1) >> index) & 1;
```

11.37. bexti

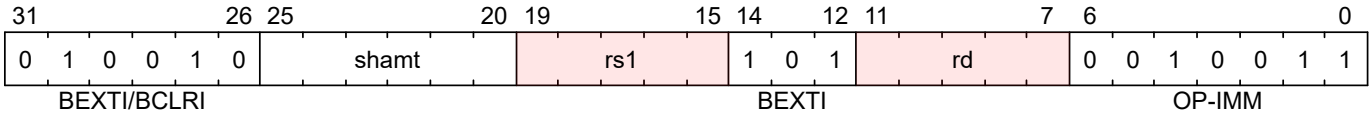
Format

```
bexti rd, rs1, shamt
```

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns a single bit extracted from rs1 at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Implementation

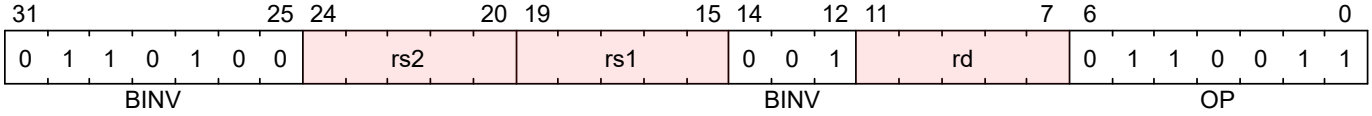
```
let index = shamt & (XLEN - 1);
X(rd) = (X(rs1) >> index) & 1;
```

11.38. binv

Format

```
binv rd, rs1, rs2
```

Encoding



Description

This instruction returns rs1 with a single bit inverted at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

Implementation

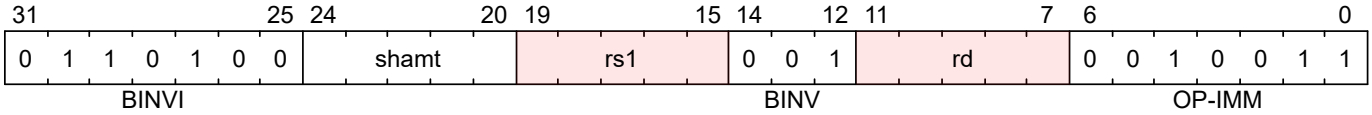
```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) ^ (1 << index)
```

11.39. binvi

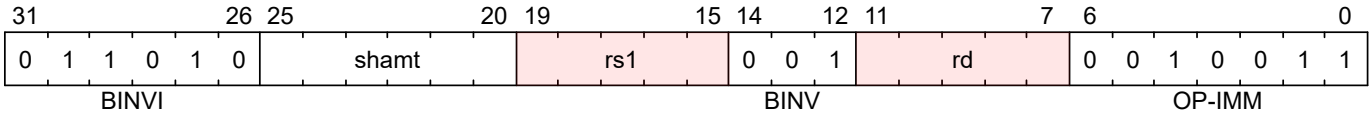
Format

```
binvi rd, rs1, shamt
```

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns rs1 with a single bit inverted at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Implementation

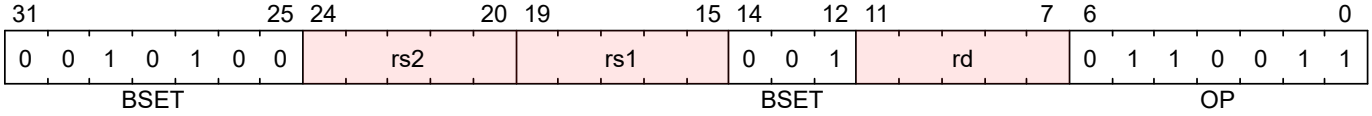
```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) ^ (1 << index)
```

11.40. bset

Format

```
bset rd, rs1,rs2
```

Encoding



Description

This instruction returns rs1 with a single bit set at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

Implementation

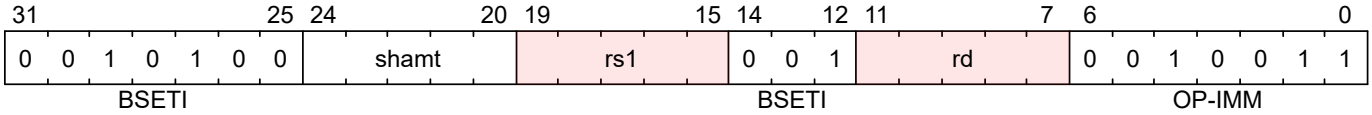
```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) | (1 << index)
```

11.41. bseti

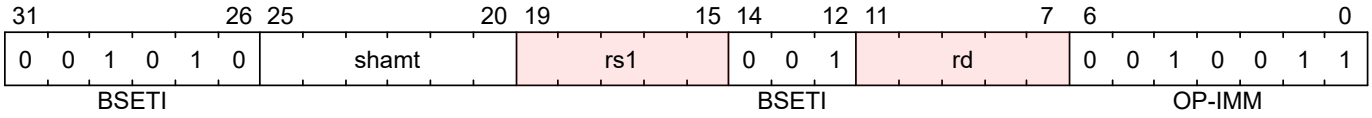
Format

```
bseti rd, rs1,shamt
```

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns rs1 with a single bit set at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Implementation

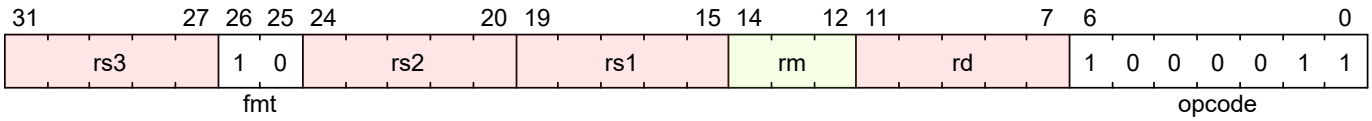
```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) | (1 << index)
```

---

12. RV32Zfh / RV64Zfh Standard Extension

12.1. fmadd.h

Encoding



Format

fmadd.h rd,rs1,rs2,rs3

Description

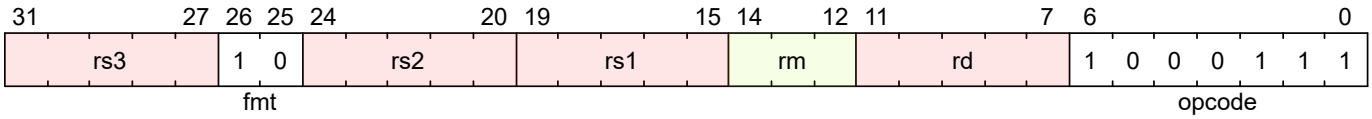
Perform half-precision fused multiply addition.

Implementation

```
f[rd] = f[rs1]×f[rs2]+f[rs3]
```

12.2. fmsub.h

Encoding



Format

fmsub.h rd,rs1,rs2,rs3

Description

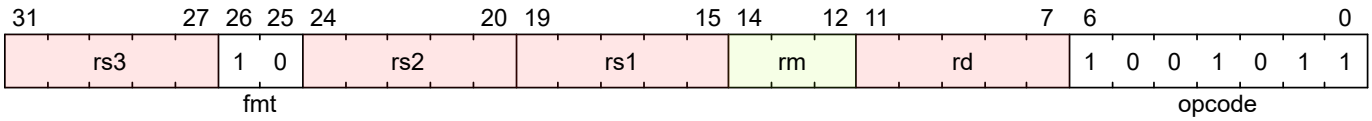
Perform half-precision fused multiply addition.

Implementation

```
f[rd] = f[rs1]×f[rs2]-f[rs3]
```

12.3. fnmsub.h

Encoding



Format

fnmsub.h rd,rs1,rs2,rs3

Description

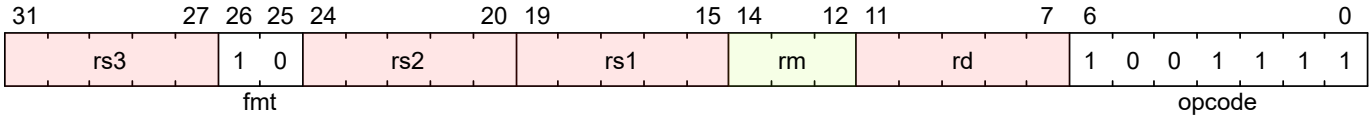
Perform negated half-precision fused multiply subtraction.

Implementation

```
f[rd] = -f[rs1]×f[rs2]+f[rs3]
```

12.4. fnmadd.h

Encoding



Format

fnmadd.h rd,rs1,rs2,rs3

Description

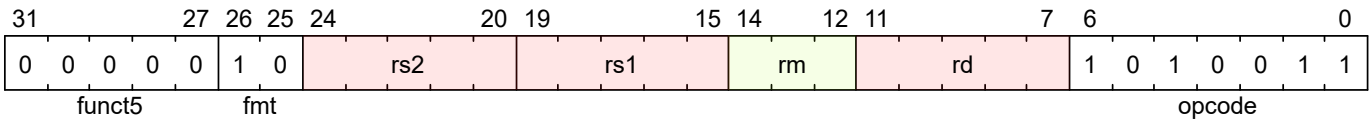
Perform negated half-precision fused multiply addition.

Implementation

```
f[rd] = -f[rs1]×f[rs2]-f[rs3]
```

12.5. fadd.h

Encoding



Format

fadd.h rd,rs1,rs2

Description

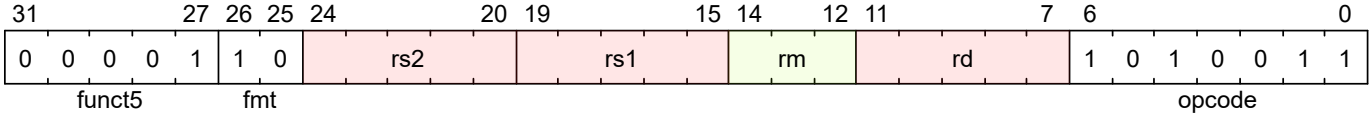
Perform half-precision floating-point addition.

Implementation

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

12.6. fsub.h

Encoding



Format

fsub.h rd,rs1,rs2

Description

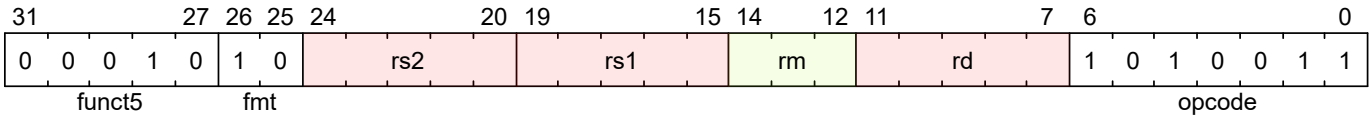
Perform half-precision floating-point substraction.

Implementation

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

12.7. fmul.h

Encoding



Format

fmul.h rd,rs1,rs2

Description

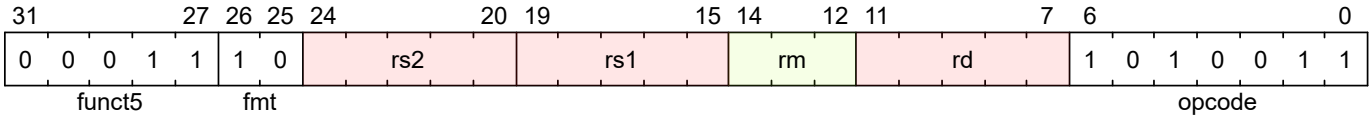
Perform half-precision floating-point multiplication.

Implementation

```
f[rd] = f[rs1] × f[rs2]
```

12.8. fdiv.h

Encoding



Format

fdiv.h rd,rs1,rs2

Description

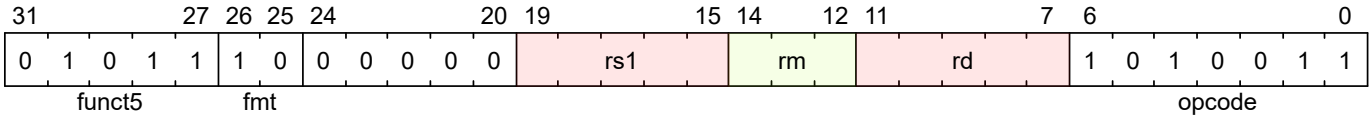
Perform half-precision floating-point division.

Implementation

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

12.9. fsqrt.h

Encoding



Format

fsqrt.h rd,rs1

Description

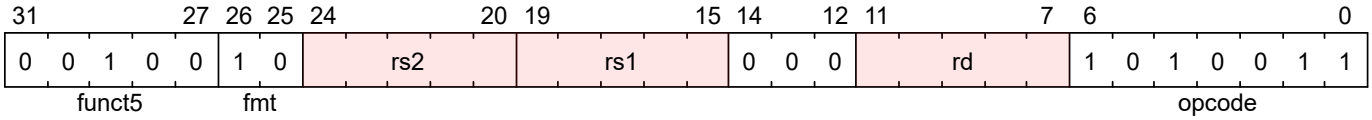
Perform half-precision square root.

Implementation

```
f[rd] = sqrt(f[rs1])
```

12.10. fsgnj.h

Encoding



Format

fsgnj.h rd,rs1,rs2

Description

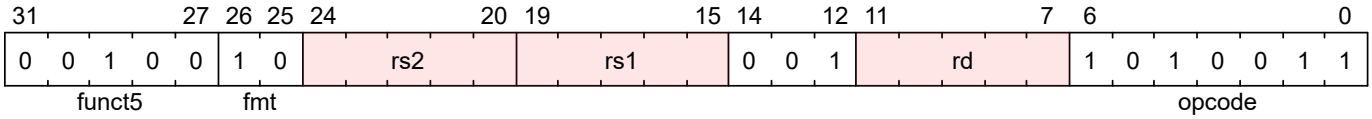
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is rs2's sign bit.

Implementation

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

12.11. fsgnjn.h

Encoding



Format

fsgnjn.h rd,rs1,rs2

Description

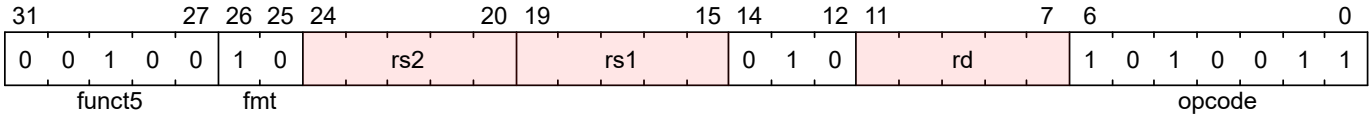
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is opposite of rs2's sign bit.

Implementation

```
f[rd] = {~f[rs2][31], f[rs1][30:0]}
```

12.12. fsgnjx.h

Encoding



Format

fsgnjx.h rd,rs1,rs2

Description

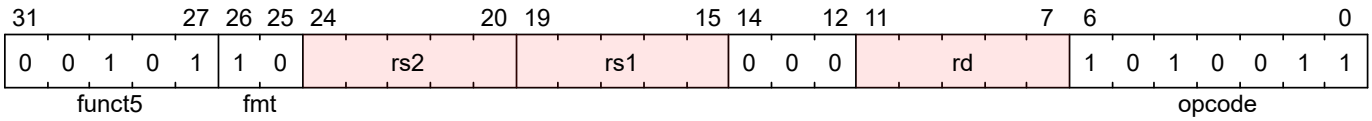
Produce a result that takes all bits except the sign bit from rs1. The result's sign bit is XOR of sign bit of rs1 and rs2.

Implementation

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

12.13. fmin.h

Encoding



Format

fmin.h rd,rs1,rs2

Description

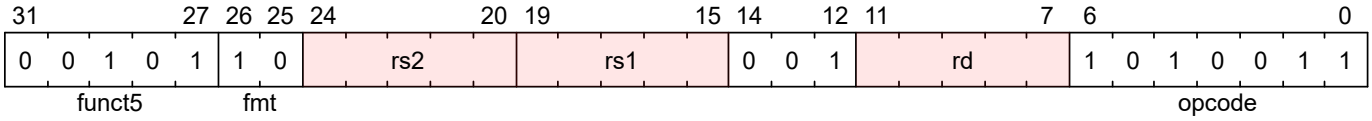
Write the smaller of single precision data in rs1 and rs2 to rd.

Implementation

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

12.14. fmax.h

Encoding



Format

fmax.h rd,rs1,rs2

Description

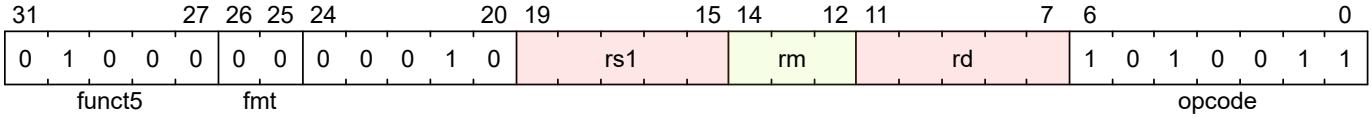
Write the larger of single precision data in rs1 and rs2 to rd.

Implementation

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

12.15. fcvt.s.h

Encoding



Format

fcvt.s.h rd,rs1

Description

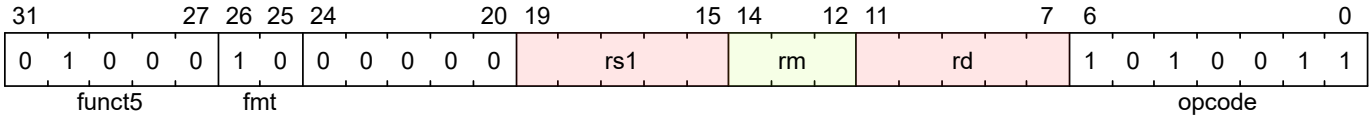
Convert a half-precision floating-point number in floating-point register rs1 to a single-precision flaoting-point number register rd.

Implementation

```
f[rd] = f16->f32(f[rs1])
```

12.16. fcvt.h.s

Encoding



Format

fcvt.h.s rd,rs1

Description

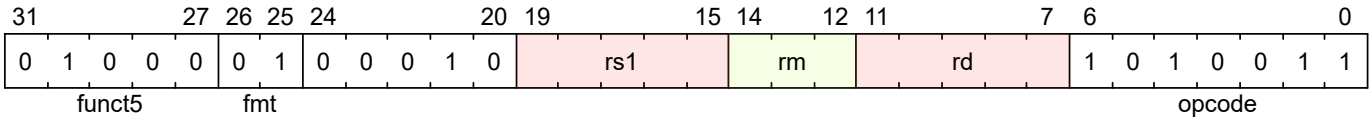
Convert a single-precision floating-point number in floating-point register rs1 to a half-precision flaoting-point number register rd.

Implementation

```
f[rd] = f32->f16(f[rs1])
```

12.17. fcvt.d.h

Encoding



Format

fcvt.d.h rd,rs1

Description

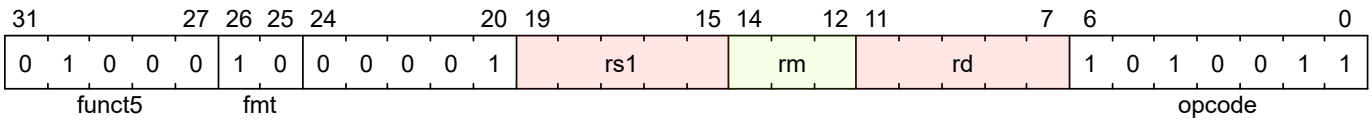
Convert a half-precision floating-point number in floating-point register rs1 to a double-precision flaoting-point number register rd.

Implementation

```
f[rd] = f16->f64(f[rs1])
```

12.18. fcvt.h.d

Encoding



Format

fcvt.h.d rd,rs1

Description

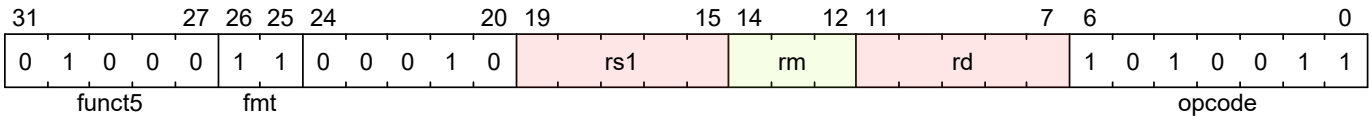
Convert a double-precision floating-point number in floating-point register rs1 to a half-precision flaoting-point number register rd.

Implementation

```
f[rd] = f64->f16(f[rs1])
```

12.19. fcvt.q.h

Encoding



Format

fcvt.q.h rd,rs1

Description

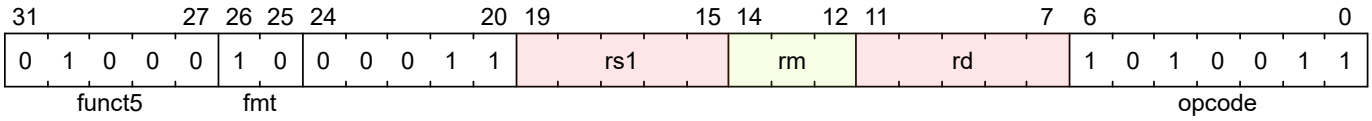
Convert a half-precision floating-point number in floating-point register rs1 to a quad-precision flaoting-point number register rd.

Implementation

```
f[rd] = f16->f128(f[rs1])
```

12.20. fcvt.h.q

Encoding



Format

fcvt.h.q rd,rs1

Description

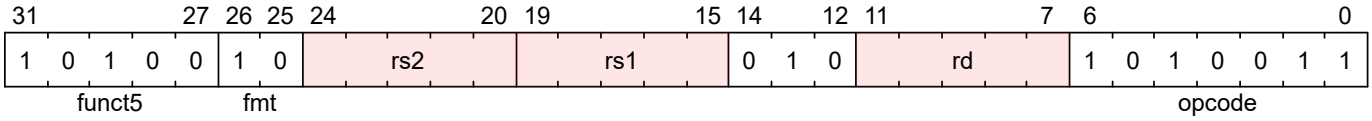
Convert a quad-precision floating-point number in floating-point register rs1 to a half-precision flaoting-point number register rd.

Implementation

```
f[rd] = f128->f16(f[rs1])
```

12.21. feq.h

Encoding



Format

feq.h rd,rs1,rs2

Description

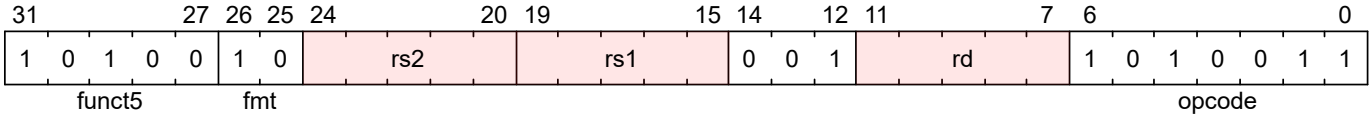
Performs a quiet equal comparison between half-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

12.22. flt.h

Encoding



Format

flt.h rd,rs1,rs2

Description

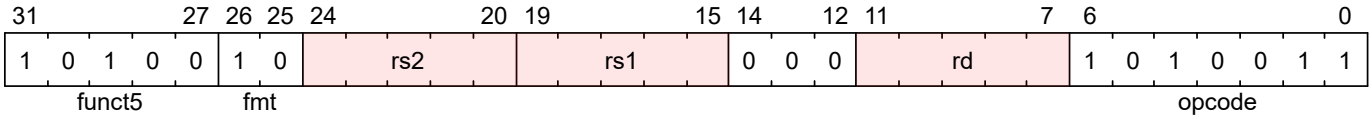
Performs a quiet less comparison between half-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

12.23. fle.h

Encoding



Format

fle.h rd,rs1,rs2

Description

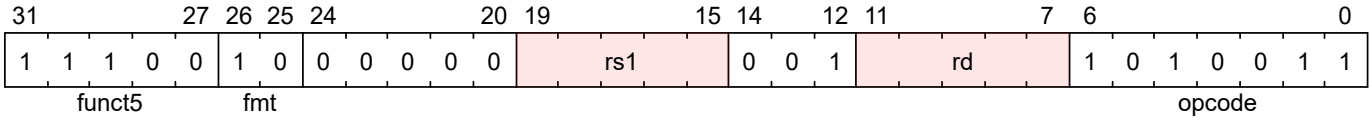
Performs a quiet less or equal comparison between half-precision floating-point registers rs1 and rs2 and record the Boolean result in integer register rd. Only signaling NaN inputs cause an Invalid Operation exception. The result is 0 if either operand is NaN.

Implementation

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

12.24. fclass.h

Encoding



Format

fclass.h rd,rs1

Description

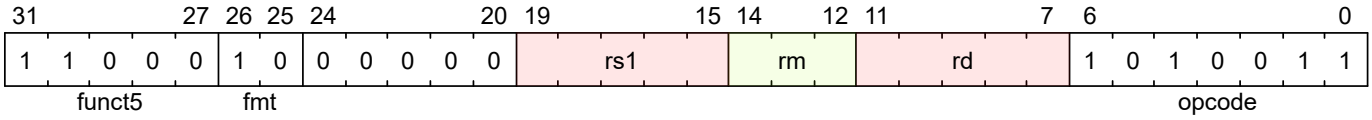
Examines the value in half-precision floating-point register rs1 and writes to integer register rd a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in [classify table]_. The corresponding bit in rd will be set if the property is true and clear otherwise. All other bits in rd are cleared. Note that exactly one bit in rd will be set.

Implementation

```
x[rd] = classify_s(f[rs1])
```

12.25. fcvt.w.h

Encoding



Format

fcvt.w.h rd,rs1

Description

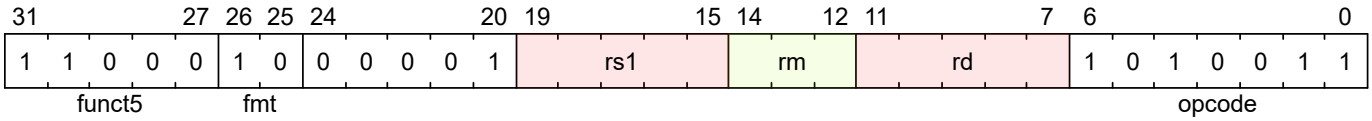
Converts half-precision floating point number in floating point register in rs1 into a sigend integer in integer register rd.

Implementation

```
f[rd] = f16->s32(x[rs1])
```

12.26. fcvt.wu.h

Encoding



Format

fcvt.wu.h rd,rs1

Description

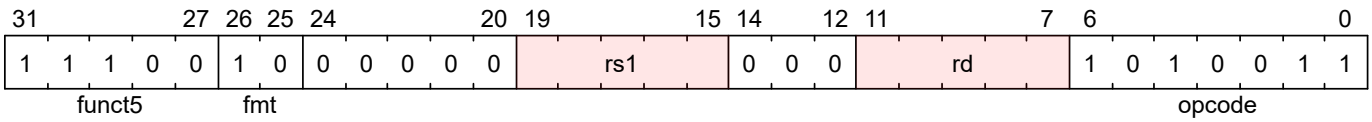
Converts half-precision floating point number in floating point register in rs1 into a unsigned integer in integer register rd.

Implementation

```
f[rd] = f16->u32(x[rs1])
```

12.27. fmv.x.h

Encoding



Format

fmv.x.w rd,rs1

Description

Move the half-precision value in floating-point register rs1 represented in IEEE 754-2008 encoding to the lower 32 bits of integer register rd.

Implementation

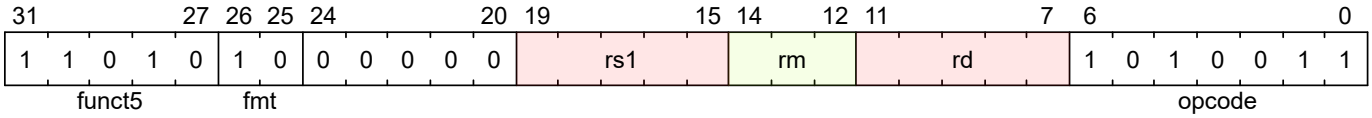
```
x[rd] = sext(f[rs1][15:0])
```

12.28. fcvt.h.w

Format

fcvt.h.w rd,rs1

Encoding



Description

Converts a 32-bit signed integer, in integer register rs1 into a half-precision floating-point number in floating-point register rd.

Implementation

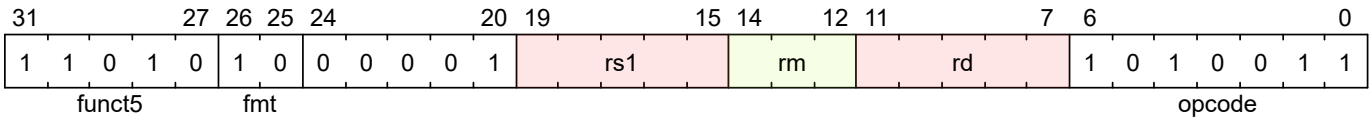
```
f[rd] = u32->f16(x[rs1])
```

12.29. fcvt.h.wu

Format

fcvt.h.wu rd,rs1

Encoding



Description

Converts a 32-bit unsigned integer, in integer register rs1 into a half-precision floating-point number in floating-point register rd.

Implementation

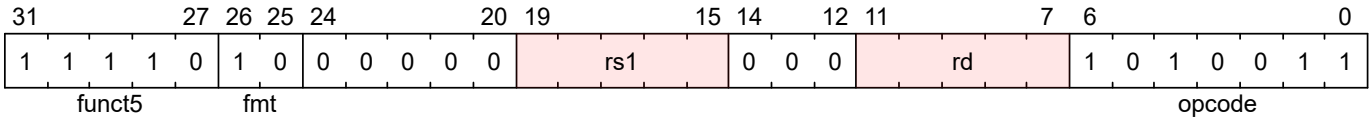
```
f[rd] = u32->f16(x[rs1])
```

12.30. fmv.h.x

Format

fmv.h.x rd,rs1

Encoding



Description

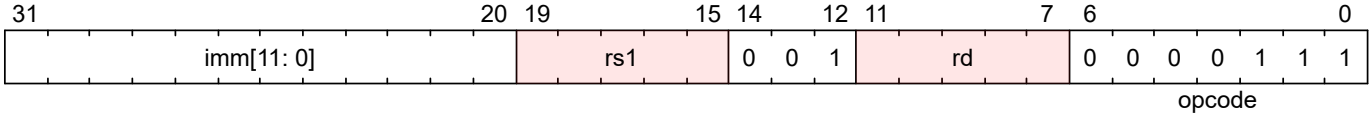
Move the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 15 bits of integer register rs1 to the floating-point register rd.

Implementation

```
f[rd] = x[rs1][15:0]
```

12.31. flh

Encoding



Format

flh rd,offset(rs1)

Description

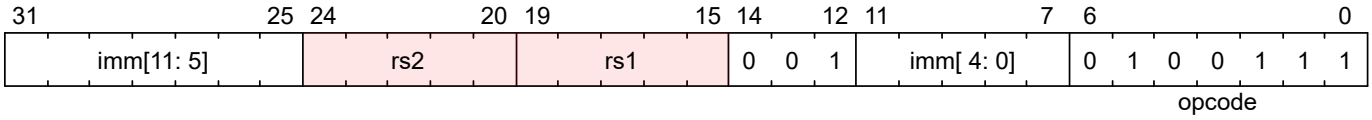
Load a half-precision floating-point value from memory into floating-point register rd.

Implementation

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

12.32. fsh

Encoding



Format

fsw rs2,offset(rs1)

Description

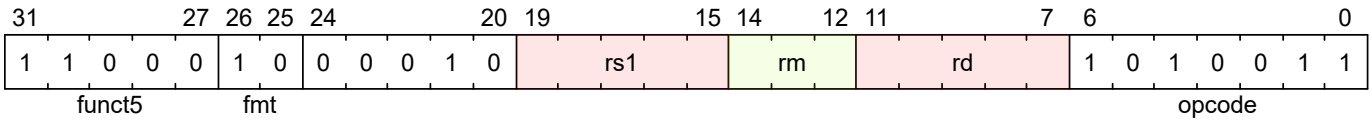
Store a half-precision value from floating-point register rs2 to memory.

Implementation

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

12.33. fcvt.l.h

Encoding



Format

fcvt.l.h rd,rs1

Description

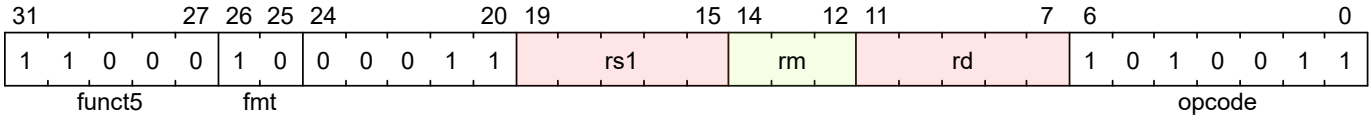
Converts half-precision floating point number in floating point register in rs1 into a 64-bit sigend integer in integer register rd.

Implementation

```
f[rd] = f16->s64(x[rs1])
```

12.34. fcvt.lu.h

Encoding



Format

fcvt.lu.h rd,rs1

Description

Converts half-precision floating point number in floating point register in rs1 into a 64-bit unsigned integer in integer register rd.

Implementation

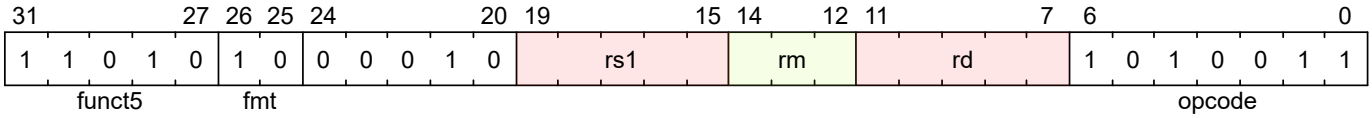
```
f[rd] = f16->s64(x[rs1])
```

12.35. fcvt.h.l

Format

fcvt.h.l rd,rs1

Encoding



Description

Converts a 64-bit signed integer, in integer register rs1 into a half-precision floating-point number in floating-point register rd.

Implementation

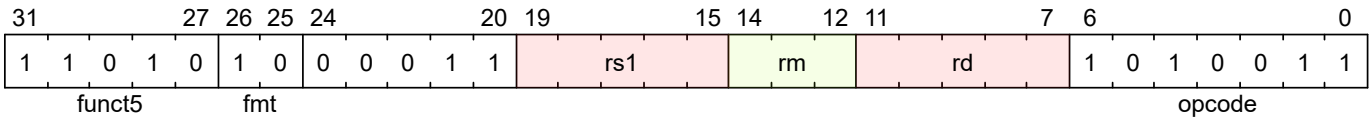
```
f[rd] = u64->f16(x[rs1])
```

12.36. fcvt.h.lu

Format

fcvt.h.lu rd,rs1

Encoding



Description

Converts a 64-bit unsigned integer, in integer register rs1 into a half-precision floating-point number in floating-point register rd.

Implementation

```
f[rd] = u64->f16(x[rs1])
```

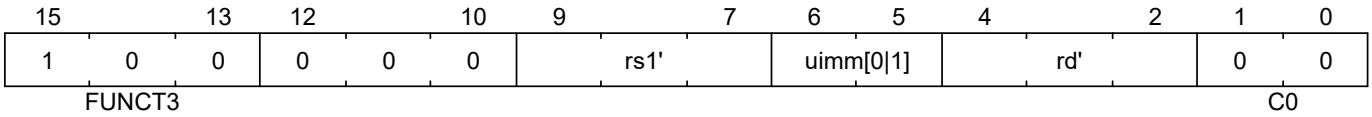
13. Zc Extension

13.1. c.lbu

Mnemonic

```
c.lbu rd', uimm(rs1')
```

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = encoding[6];
```

Description

This instruction loads a byte from the memory address formed by adding rs1' to the zero extended immediate uimm. The resulting byte is zero extended to XLEN bits and is written to rd'.

Note

rd' and rs1' are from the standard 8-register set x8-x15.

Prerequisites

None

Implementation

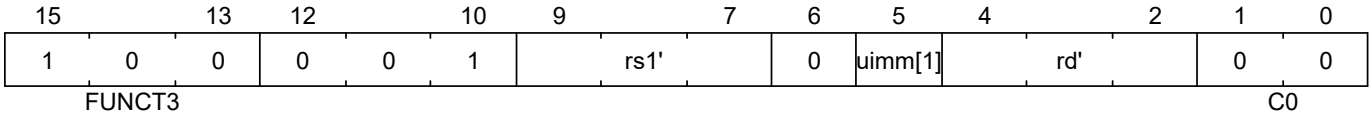
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

X(rdc) = EXTZ(mem[X(rs1c)+EXTZ(uimm)][7..0]);
```

13.2. c.lhu

Mnemonic

```
c.lhu rd', uimm(rs1')
```

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction loads a halfword from the memory address formed by adding rs1' to the zero extended immediate uimm. The resulting halfword is zero extended to XLEN bits and is written to rd'.

Note

rd' and rs1' are from the standard 8-register set x8-x15.

Prerequisites

None

Implementation

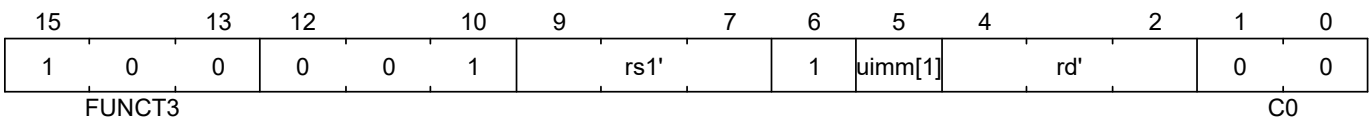
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

X(rdc) = EXTZ(load_mem[X(rs1c)+EXTZ(uimm)][15..0]);
```

13.3. c.lh

Mnemonic

```
c.lh rd', uimm(rs1')
```

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction loads a halfword from the memory address formed by adding rs1' to the zero extended immediate uimm. The resulting halfword is sign extended to XLEN bits and is written to rd'.

Note

rd' and rs1' are from the standard 8-register set x8-x15.

Prerequisites

None

Implementation

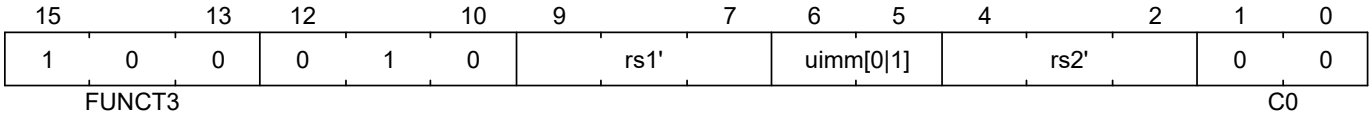
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

X(rdc) = EXTS(load_mem[X(rs1c)+EXTZ(uimm)][15..0]);
```

13.4. c.sb

Mnemonic

```
c.sb rs2', uimm(rs1')
```

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = encoding[6];
```

Description

This instruction stores the least significant byte of rs2' to the memory address formed by adding rs1' to the zero extended immediate uimm.

Note

rs1' and rs2' are from the standard 8-register set x8-x15.

Prerequisites

None

Implementation

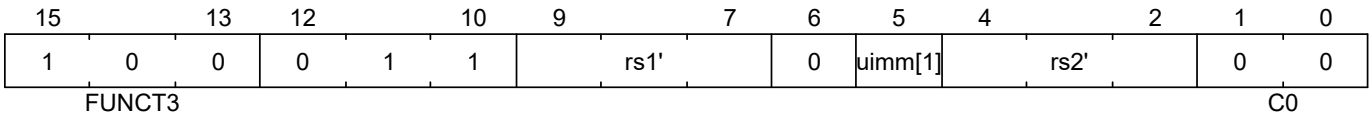
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

mem[X(rs1c)+EXTZ(uimm)][7..0] = X(rs2c)
```

13.5. c.sh

Mnemonic

```
c.sh rs2', uimm(rs1')
```

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
  uimm[31:2] = 0;
  uimm[1]    = encoding[5];
  uimm[0]    = 0;
```

Description

This instruction stores the least significant halfword of rs2' to the memory address formed by adding rs1' to the zero extended immediate uimm.

Note

rs1' and rs2' are from the standard 8-register set x8-x15.

Prerequisites

None

Implementation

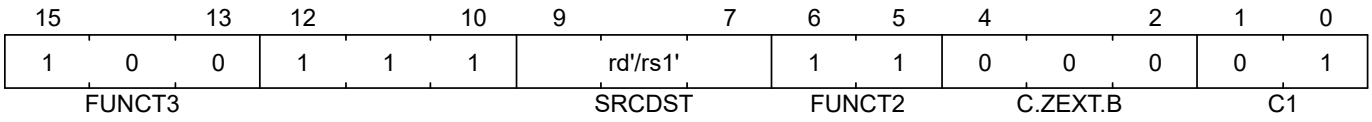
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

mem[X(rs1c)+EXTZ(uimm)][15..0] = X(rs2c)
```

13.6. c.zext.b

Mnemonic

```
c.zext.b rd'/rs1'
```

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant byte of the operand to XLEN bits by inserting zeros into all of the bits more significant than 7.

Note

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

```
andi rd'/rs1', rd'/rs1', 0xff
```

Note

The SAIL module variable for rd'/rs1' is called *rsdc*.

Implementation

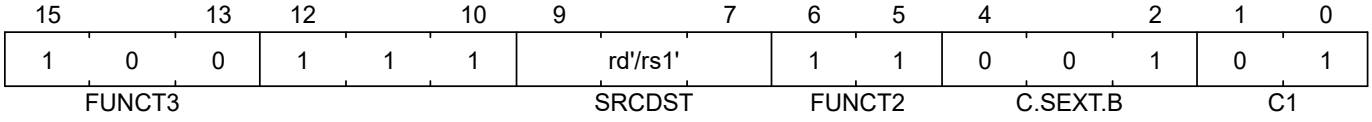
```
X(rsdc) = EXTZ(X(rsdc)[7..0]);
```

13.7. c.sext.b

Mnemonic

```
c.sext.b rd'/rs1'
```

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It sign-extends the least-significant byte in the operand to XLEN bits by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

Note

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

Note

The SAIL module variable for rd'/rs1' is called *rsdc*.

Implementation

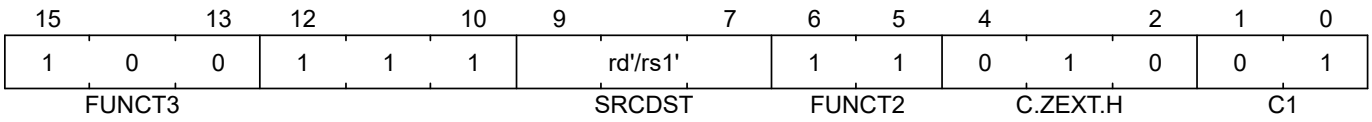
```
X(rsdc) = EXTS(X(rsdc)[7..0]);
```

13.8. c.zext.h

Mnemonic

```
c.zext.h rd'/rs1'
```

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant halfword of the operand to XLEN bits by inserting zeros into all of the bits more significant than 15.

Note

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

Note

The SAIL module variable for rd'/rs1' is called *rsdc*.

Implementation

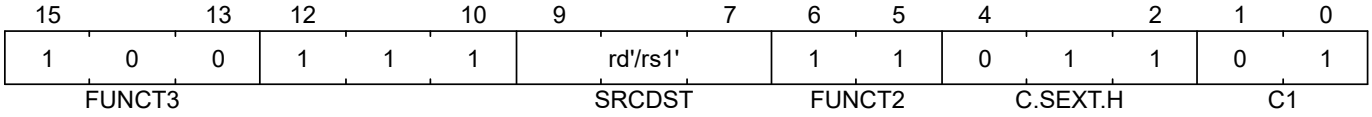
```
X(rsdc) = EXTZ(X(rsdc)[15..0]);
```

13.9. c.sext.h

Mnemonic

```
c.sext.h rd'/rs1'
```

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It sign-extends the least-significant halfword in the operand to XLEN bits by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

Note

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

Note

The SAIL module variable for rd'/rs1' is called *rsdc*.

Implementation

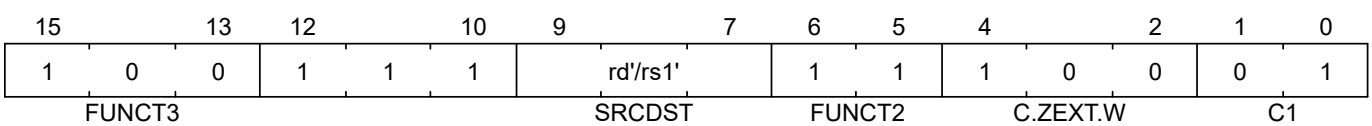
```
X(rsdc) = EXTS(X(rsdc)[15..0]);
```

13.10. c.zext.w

Mnemonic

```
c.zext.w rd'/rs1'
```

Encoding (RV64)



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant word of the operand to XLEN bits by inserting zeros into all of the bits more significant than 31.

Note

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zba is also required.

32-bit equivalent

```
add.uw rd'/rs1', rd'/rs1', zero
```

Note

The SAIL module variable for rd'/rs1' is called *rsdc*.

Implementation

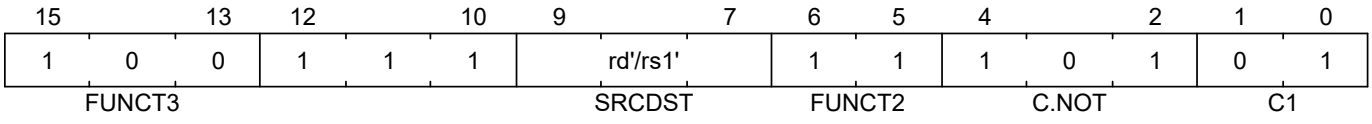
```
X(rsdc) = EXTZ(X(rsdc)[31..0]);
```

13.11. c.not

Mnemonic

```
c.not rd'/rs1'
```

Encoding (RV32, RV64)



Description

This instruction takes the one's complement of rd'/rs1' and writes the result to the same register.

Note

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

```
xori rd'/rs1', rd'/rs1', -1
```

> **Note**
>
> The SAIL module variable for rd'/rs1' is called *rsdc*.

Implementation

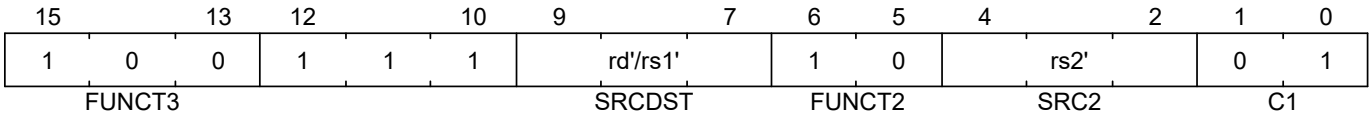
```
X(rsdc) = X(rsdc) XOR -1;
```

13.12. c.mul

Mnemonic

```
c.mul _rsd'_, rs2'
```

Encoding (RV32, RV64)



Description

This instruction multiplies XLEN bits of the source operands from *rsd'* and rs2' and writes the lowest XLEN bits of the result to *rsd'*.

> **Note**
>
> rd'/rs1' and rs2' are from the standard 8-register set x8-x15.

Prerequisites

M or Zmmul must be configured.

> **Note**
>
> The SAIL module variable for rd'/rs1' is called *rsdc*, and for rs2' is called *rs2c*.

Implementation

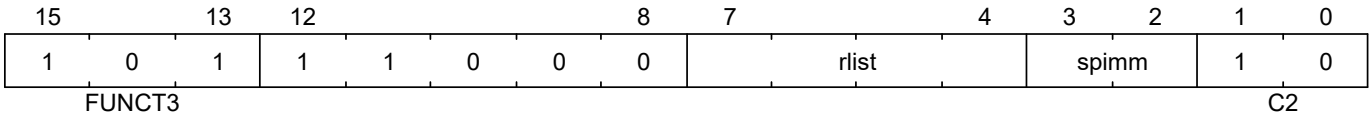
```
let result_wide = to_bits(2 * sizeof(xlen), signed(X(rsdc)) * signed(X(rs2c)));
X(rsdc) = result_wide[(sizeof(xlen) - 1) .. 0];
```

13.13. cm.push

Mnemonic

```
cm.push _{reg_list}, -stack_adj_
```

Encoding (RV32, RV64)



Note

rlist values 0 to 3 are reserved for a future EABI variant called *cm.push.e*

Assembly Syntax

```
cm.push {reg_list},  -stack_adj
cm.push {xreg_list}, -stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
```

```
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pushes (stores) the registers in *reg_list* to the memory below the stack pointer, and then creates the stack frame by decrementing the stack pointer by *stack_adj*, including any additional stack space requested by the value of *spimm*.

Note

All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see [insns-pushpop].

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Implementation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("sw x[i], 0(addr)");
      8:  asm("sd x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

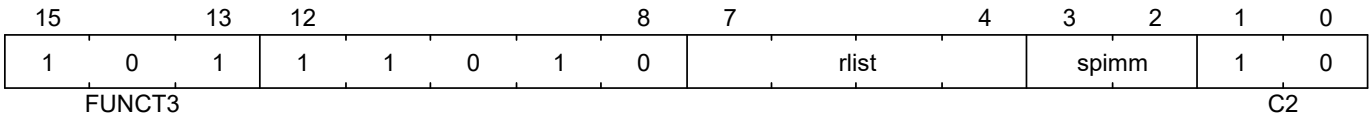
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

sp-=stack_adj;
```

13.14. cm.pop

Mnemonic

```
cm.pop  _{reg_list}, stack_adj_
```

Encoding (RV32, RV64)



Note

rlist values 0 to 3 are reserved for a future EABI variant called *cm.pop.e*

Assembly Syntax

```
cm.pop {reg_list},  stack_adj
cm.pop {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
```

```
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, and then adjusts the stack pointer by *stack_adj*.

> **Note**
>
> All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see [insns-pushpop].

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Implementation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

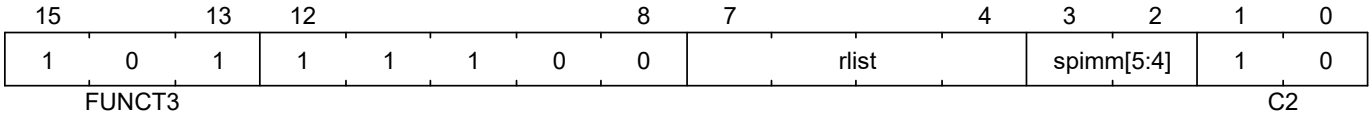
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

sp+=stack_adj;
```

13.15. cm.popretz

Mnemonic

```
cm.popretz _{reg_list}, stack_adj_
```

Encoding (RV32, RV64)



Note

rlist values 0 to 3 are reserved for a future EABI variant called *cm.popretz.e*

Assembly Syntax

```
cm.popretz {reg_list},  stack_adj
cm.popretz {xreg_list}, stack_adj
```

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
```

```
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, adjusts the stack pointer by *stack_adj*, moves zero into a0 and then returns to *ra*.

> **Note**
>
> All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see [insns-pushpop].

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Implementation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
```

```
if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

Note

The *li a0, 0* **could** be executed more than once, but is included in the atomic section for convenience.

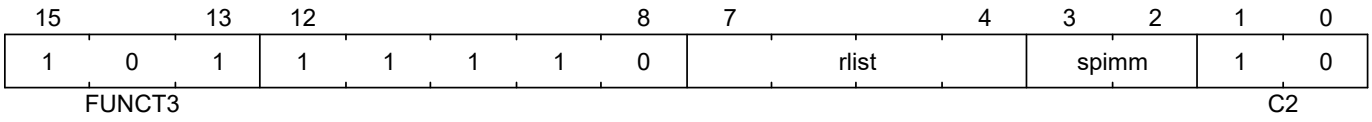
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

asm("li a0, 0");
sp+=stack_adj;
asm("ret");
```

13.16. cm.popret

Mnemonic

```
cm.popret _{reg_list}, stack_adj_
```

Encoding (RV32, RV64)



Note

rlist values 0 to 3 are reserved for a future EABI variant called *cm.popret.e*

Assembly Syntax

```
cm.popret {reg_list},  stack_adj
cm.popret {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
```

```
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, adjusts the stack pointer by *stack_adj* and then returns to *ra*.

> **Note**
>
> All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see [insns-pushpop].

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Implementation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
```

```
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

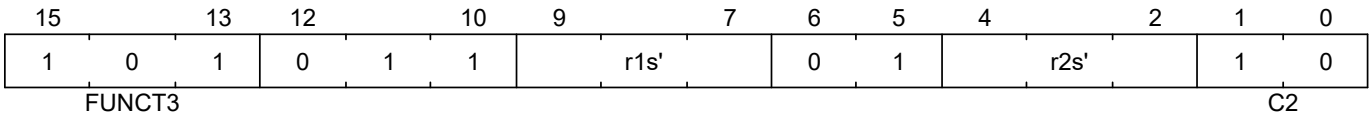
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

sp+=stack_adj;
asm("ret");
```

13.17. cm.mvsa01

Mnemonic

```
cm.mvsa01 _r1s'_, _r2s'_
```

Encoding (RV32, RV64)



> **Note**
>
> For the encoding to be legal *r1s'* != *r2s'*.

Assembly Syntax

```
cm.mvsa01 r1s', r2s'
```

Description

This instruction moves *a0* into *r1s'* and *a1* into *r2s'*. *r1s'* and *r2s'* must be different. The execution is atomic, so it is not possible to observe state where only one of *r1s'* or *r2s'* has been updated.

The encoding uses *sreg* number specifiers instead of *xreg* number specifiers to save encoding space. The mapping between them is specified in the pseudocode below.

> **Note**
>
> The *s* register mapping is taken from the UABI, and may not match the currently unratified EABI. *cm.mvsa01.e* may be included in the future.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

Implementation

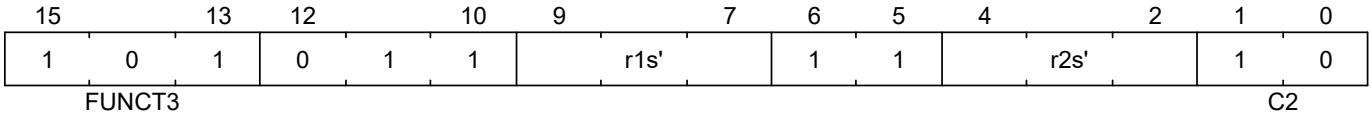
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
  reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[xreg1] = X[10];
X[xreg2] = X[11];
```

13.18. cm.mva01s

Mnemonic

```
cm.mva01s _r1s'_, _r2s'_
```

Encoding (RV32, RV64)



Assembly Syntax

```
cm.mva01s r1s', r2s'
```

Description

This instruction moves *r1s'* into *a0* and *r2s'* into *a1*. The execution is atomic, so it is not possible to observe state where only one of *a0* or *a1* have been updated.

The encoding uses *sreg* number specifiers instead of *xreg* number specifiers to save encoding space. The mapping between them is specified in the pseudocode below.

Note

The *s* register mapping is taken from the UABI, and may not match the currently unratified EABI. *cm.mva01s.e* may be included in the future.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

Implementation

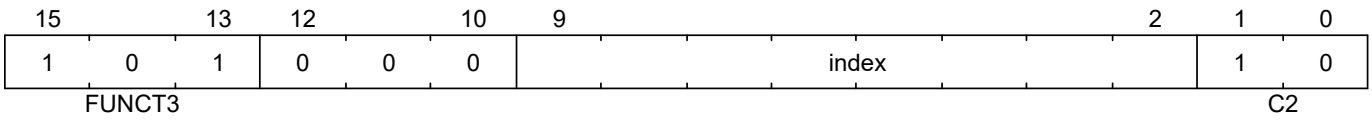
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
  reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[10] = X[xreg1];
X[11] = X[xreg2];
```

13.19. cm.jt

Mnemonic

```
cm.jt _index_
```

Encoding (RV32, RV64)



Note

For this encoding to decode as *cm.jt*, *index<32*, otherwise it decodes as *cm.jalt*, see [insns-cm_jalt].

Note

If jvt.mode = 0 (Jump Table Mode) then *cm.jt* behaves as specified here. If jvt.mode is a reserved value, then *cm.jt* is also reserved. In the future other defined values of jvt.mode may change the behaviour of *cm.jt*.

Assembly Syntax

```
cm.jt index
```

Description

cm.jt reads an entry from the jump vector table in memory and jumps to the address that was read.

For further information see [insns-tablejump].

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

Implementation

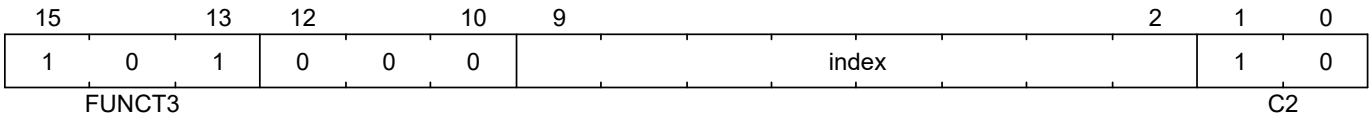
```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

# target_address is temporary internal state, it doesn't represent a real register
# InstMemory is byte indexed

switch(XLEN) {
  32:  table_address[XLEN-1:0] = jvt.base + (index<<2);
  64:  table_address[XLEN-1:0] = jvt.base + (index<<3);
}

//fetch from the jump table
target_address[XLEN-1:0] = InstMemory[table_address][XLEN-1:0];

j target_address[XLEN-1:0]&~0x1;
```

13.20. cm.jalt

Mnemonic

```
cm.jalt _index_
```

Encoding (RV32, RV64)



Note

For this encoding to decode as *cm.jalt*, *index>=32*, otherwise it decodes as *cm.jt*, see Jump via table.

Note

If jvt.mode = 0 (Jump Table Mode) then *cm.jalt* behaves as specified here. If jvt.mode is a reserved value, then *cm.jalt* is also reserved. In the future other defined values of jvt.mode may change the behaviour of *cm.jalt*.

Assembly Syntax

```
cm.jalt index
```

Description

cm.jalt reads an entry from the jump vector table in memory and jumps to the address that was read, linking to *ra*.

For further information see [insns-tablejump].

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

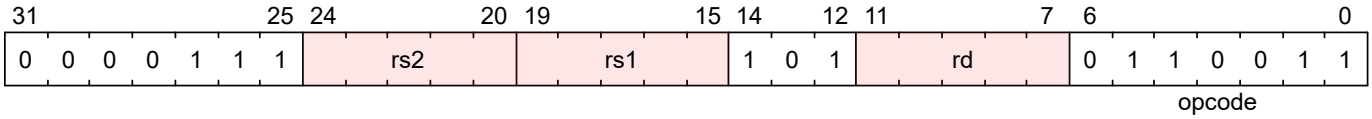
Implementation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

# target_address is temporary internal state, it doesn't represent a real register
# InstMemory is byte indexed

switch(XLEN) {
  32:  table_address[XLEN-1:0] = jvt.base + (index<<2);
  64:  table_address[XLEN-1:0] = jvt.base + (index<<3);
}

//fetch from the jump table
target_address[XLEN-1:0] = InstMemory[table_address][XLEN-1:0];

jal ra, target_address[XLEN-1:0]&~0x1;
```

14. Zicond Extension

14.1. czero.eqz

Encoding



Format

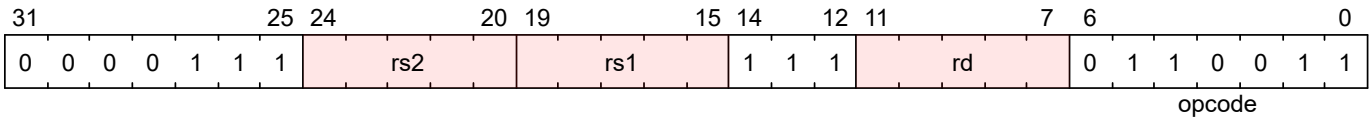
```
czero.eqz rd, rs1, rs2
```

Description

If rs2 contains the value zero, this instruction writes the value zero to rd. Otherwise, this instruction copies the contents of rs1 to rd. This instruction carries a syntactic dependency from both rs1 and rs2 to rd. Furthermore, if the Zkt extension is implemented, this instruction's timing is independent of the data values in rs1 and rs2.

14.2. czero.nez

Encoding



Format

```
czero.nez rd, rs1, rs2
```

Description

If rs2 contains a nonzero value, this instruction writes the value zero to rd. Otherwise, this instruction copies the contents of rs1 to rd. This instruction carries a syntactic dependency from both rs1 and rs2 to rd. Furthermore, if the Zkt extension is implemented, this instruction's timing is independent of the data values in rs1 and rs2.

15. Register Definitions

15.1. Integer Registers

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary/alternate link register	Caller
x6	t1	Temporaries	Caller
x7	t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10	a0	Function arguments/return values	Caller
x11	a1	Function arguments/return values	Caller
x12	a2	Function arguments	Caller
x13	a3	Function arguments	Caller
x14	a4	Function arguments	Caller
x15	a5	Function arguments	Caller
x16	a6	Function arguments	Caller
x17	a7	Function arguments	Caller
x18	s2	Saved registers	Callee
x19	s3	Saved registers	Callee
x20	s4	Saved registers	Callee

Register	ABI Name	Description	Saver
x21	s5	Saved registers	Callee
x22	s6	Saved registers	Callee
x23	s7	Saved registers	Callee
x24	s8	Saved registers	Callee
x25	s9	Saved registers	Callee
x26	s10	Saved registers	Callee
x27	s11	Saved registers	Callee
x28	t3	Temporaries	Caller
x29	t4	Temporaries	Caller
x30	t5	Temporaries	Caller
x31	t6	Temporaries	Caller

15.2. Floating Point Registers

Register	ABI Name	Description	Saver
f0	ft0	FP temporaries	Caller
f1	ft1	FP temporaries	Caller
f2	ft2	FP temporaries	Caller
f3	ft3	FP temporaries	Caller
f4	ft4	FP temporaries	Caller
f5	ft5	FP temporaries	Caller
f6	ft6	FP temporaries	Caller
f7	ft7	FP temporaries	Caller
f8	fs0	FP saved registers	Callee
f9	fs1	FP saved registers	Callee
f10	fa0	FP arguments/return values	Caller
f11	fa1	FP arguments/return values	Caller
f12	fa2	FP arguments	Caller
f13	fa3	FP arguments	Caller
f14	fa4	FP arguments	Caller
f15	fa5	FP arguments	Caller
f16	fa6	FP arguments	Caller
f17	fa7	FP arguments	Caller
f18	fs2	FP saved registers	Callee
f19	fs3	FP saved registers	Callee
f20	fs4	FP saved registers	Callee
f21	fs5	FP saved registers	Callee
f22	fs6	FP saved registers	Callee

Register	ABI Name	Description	Saver
f23	fs7	FP saved registers	Callee
f24	fs8	FP saved registers	Callee
f25	fs9	FP saved registers	Callee
f26	fs10	FP saved registers	Callee
f27	fs11	FP saved registers	Callee
f28	ft8	FP temporaries	Caller
f29	ft9	FP temporaries	Caller
f30	ft10	FP temporaries	Caller
f31	ft11	FP temporaries	Caller