RISC-V Instruction Set Specifications

Table of Contents

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.

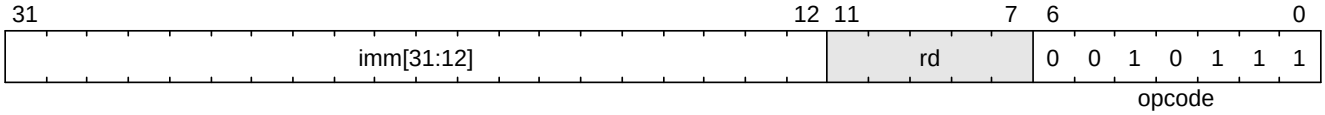
Implementation

```
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.

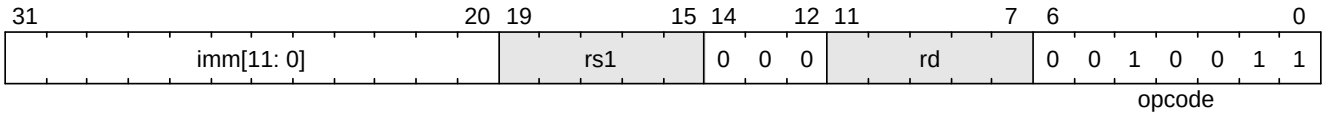
Implementation

```
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.

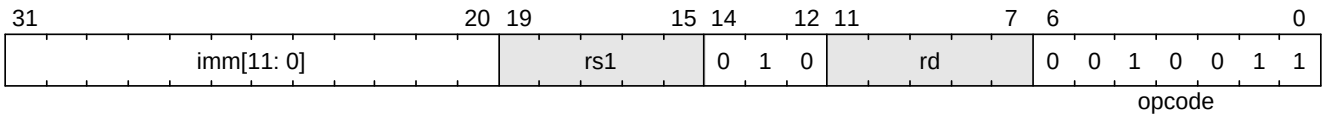
Implementation

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

1.4. slti

set less than immediate

Encoding



Format

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

Description

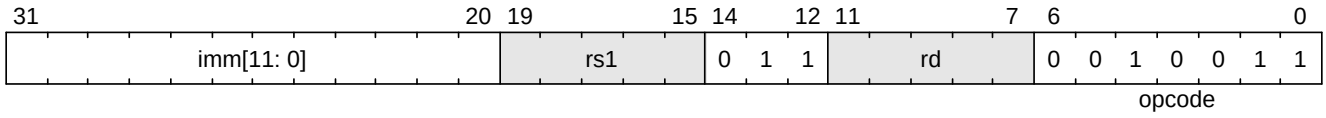
Place the value 1 in register rd if register rs1 is less than the signextended immediate when both are treated as signed numbers, else 0 is written to rd.

Implementation

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

1.5. sltiu

Encoding



Format

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

Description

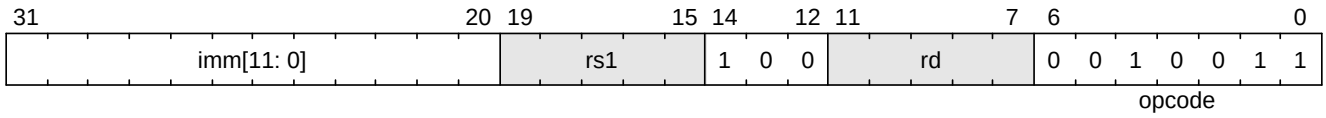
Place the value 1 in register rd if register rs1 is less than the immediate when both are treated as unsigned numbers, else 0 is written to rd.

Implementation

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

1.6. xori

Encoding



Format

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

Description

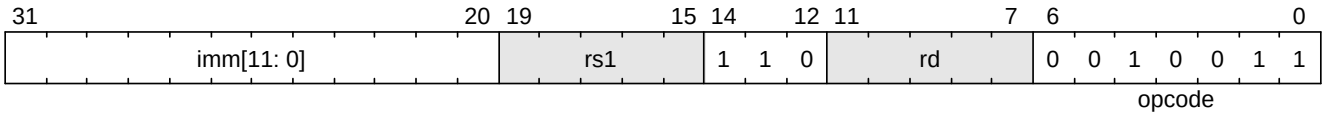
Performs bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd Note, "XORI rd, rs1, -1" performs a bitwise logical inversion of register rs1(assembler pseudo-instruction NOT rd, rs)

Implementation

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

1.7. ori

Encoding



Format

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

Description

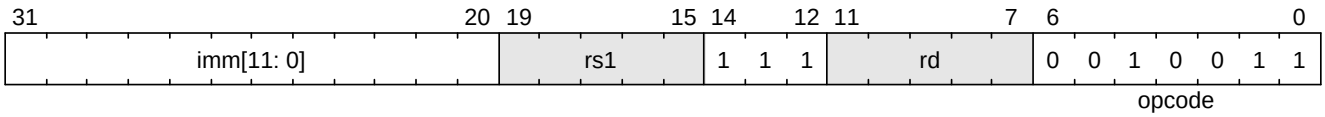
Performs bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd

Implementation

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

1.8. andi

Encoding



Format

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

Description

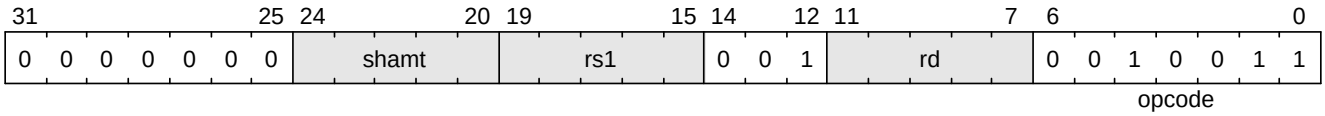
Performs bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd

Implementation

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

1.9. slli

Encoding



Format

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

Description

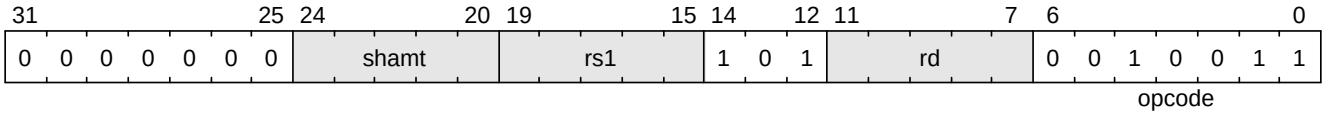
Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate In RV64, bit-25 is used to shamt[5].

Implementation

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

1.10. srli

Encoding



Format

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

Description

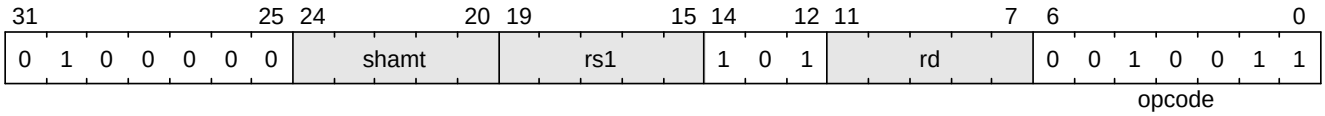
Performs logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate In RV64, bit-25 is used to shamt[5].

Implementation

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

1.11. srai

Encoding



Format

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

Description

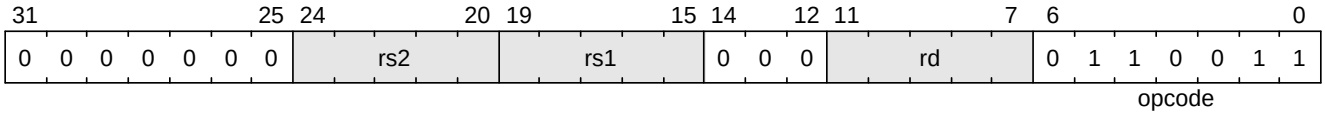
Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of the immediate In RV64, bit-25 is used to shamt[5].

Implementation

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

1.12. add

Encoding



Format

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

Description

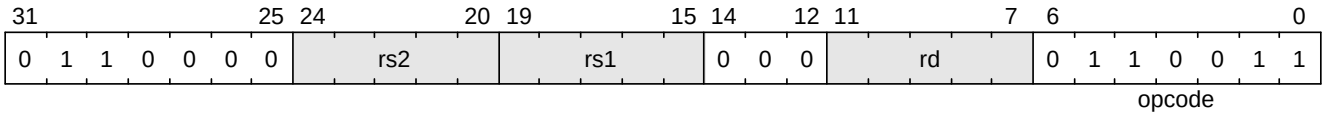
Adds the registers rs1 and rs2 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

Implementation

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

1.13. sub

Encoding



Format

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

Description

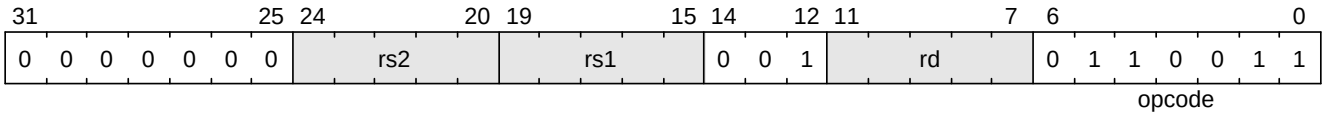
Subs the register rs2 from rs1 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

Implementation

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

1.14. sll

Encoding



Format

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

Description

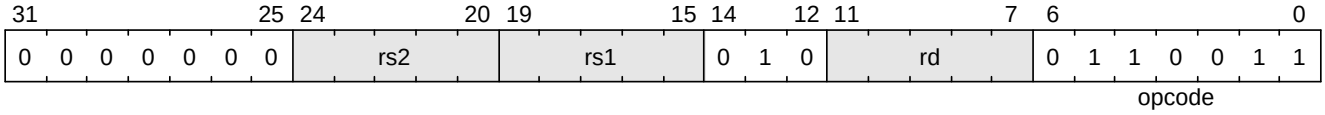
Performs logical left shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

Implementation

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

1.15. slt

Encoding



Format

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

Description

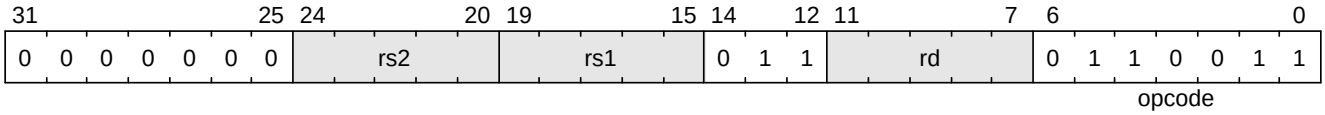
Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as signed numbers, else 0 is written to rd.

Implementation

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

1.16. sltu

Encoding



Format

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

Description

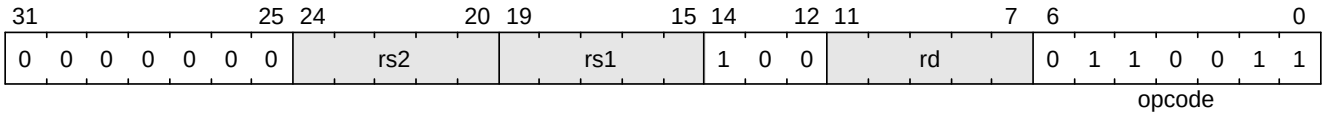
Place the value 1 in register rd if register rs1 is less than register rs2 when both are treated as unsigned numbers, else 0 is written to rd.

Implementation

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

1.17. xor

Encoding



Format

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

Description

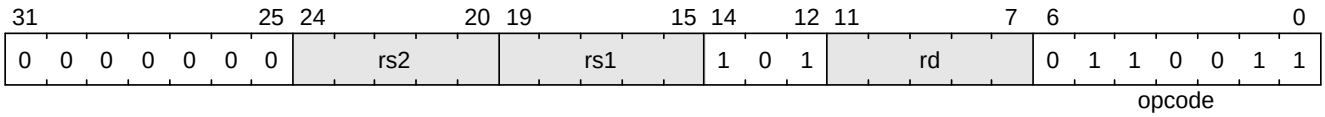
Performs bitwise XOR on registers rs1 and rs2 and place the result in rd

Implementation

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

1.18. srl

Encoding



Format

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

Description

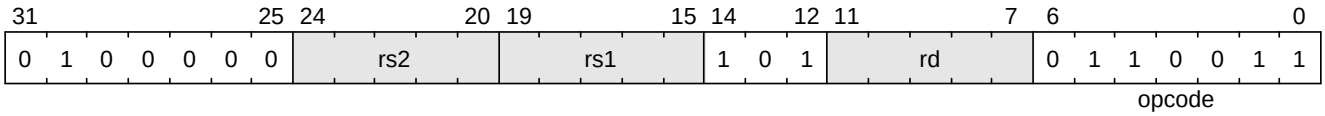
Logical right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2

Implementation

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

1.19. sra

Encoding



Format

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

Description

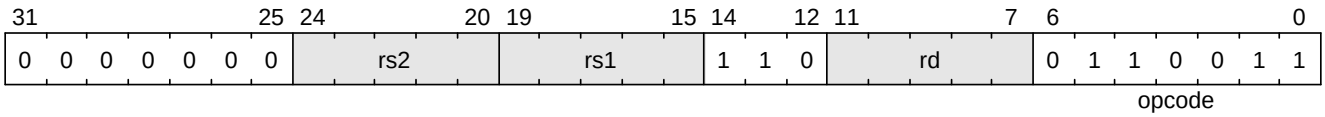
Performs arithmetic right shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2

Implementation

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

1.20. or

Encoding



Format

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

Description

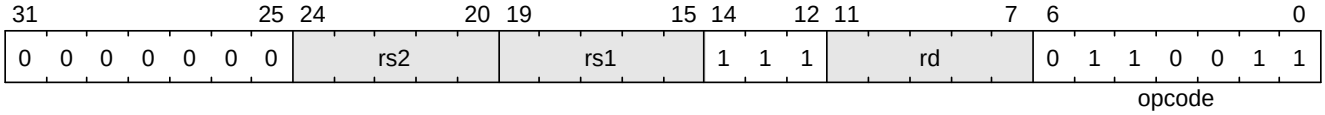
Performs bitwise OR on registers rs1 and rs2 and place the result in rd

Implementation

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

1.21. and

Encoding



Format

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

Description

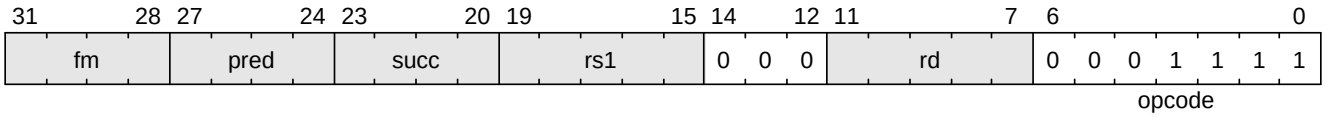
Performs bitwise AND on registers rs1 and rs2 and place the result in rd

Implementation

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

1.22. fence

Encoding



Format

```
fence pred, succ
```

Description

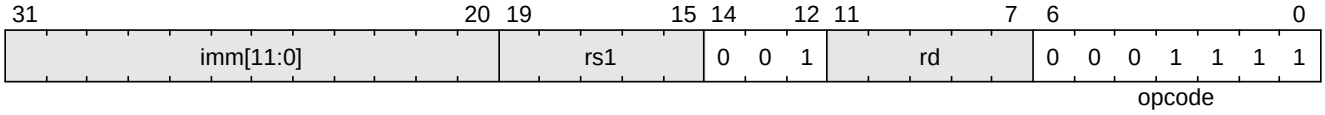
Used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads ®, and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart 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.

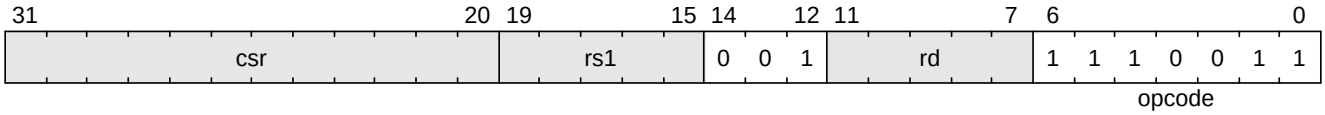
Implementation

```
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.

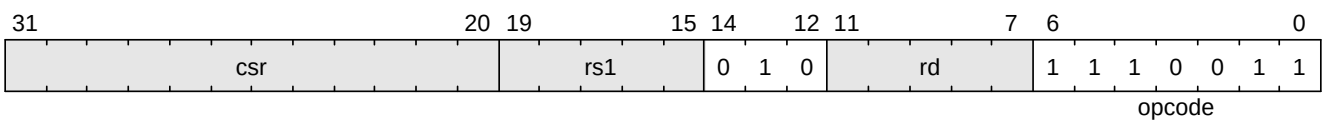
Implementation

```
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).

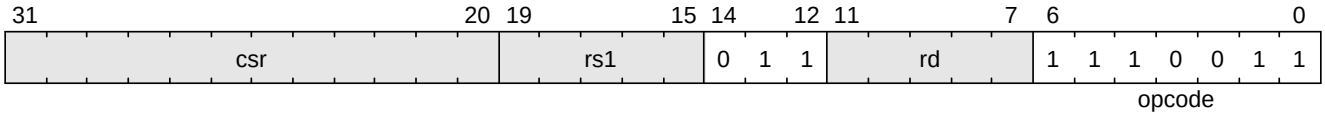
Implementation

```
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

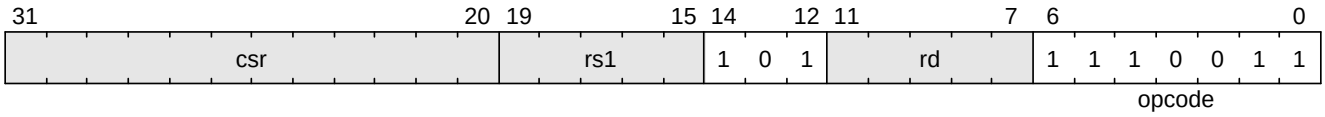
Reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register rd. The initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be cleared in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

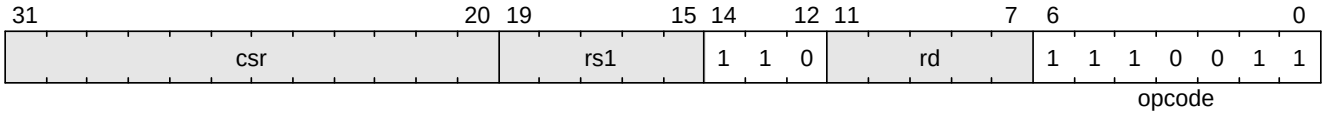
Update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.

Implementation

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

1.28. csrrsi

Encoding



Format

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

Description

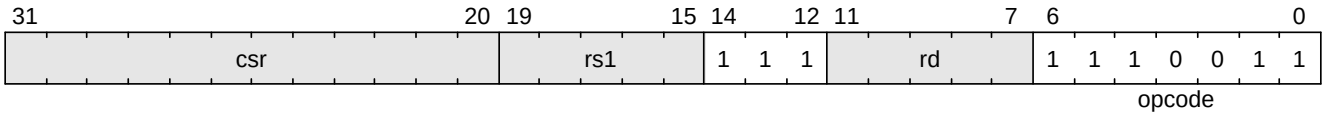
Set CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.

Implementation

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

1.29. csrrci

Encoding



Format

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

Description

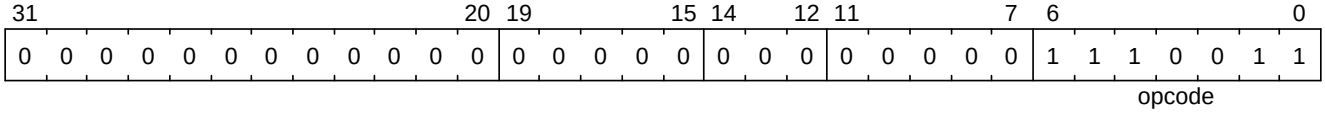
Clear CSR bit using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the rs1 field.

Implementation

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

1.30. ecall

Encoding



Format

```
ecall
```

Description

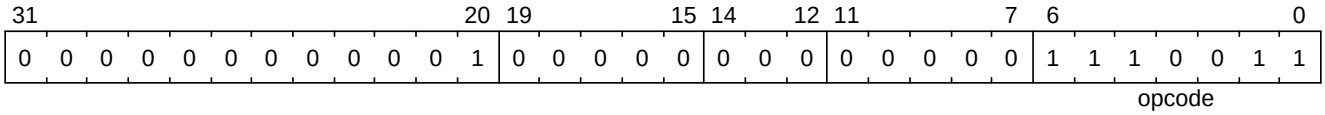
Make a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception, environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation.

Implementation

```
RaiseException(EnvironmentCall)
```

1.31. ebreak

Encoding



Format

```
ebreak
```

Description

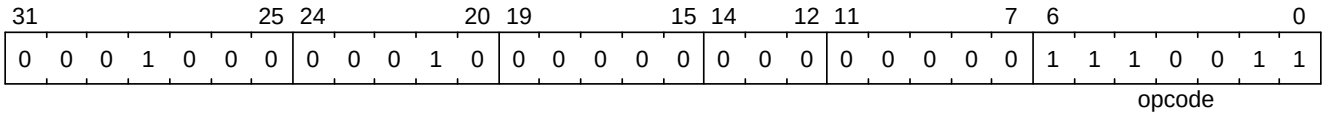
Used by debuggers to cause control to be transferred back to a debugging environment. It generates a breakpoint exception and performs no other operation.

Implementation

```
RaiseException(Breakpoint)
```

1.32. sret

Encoding



Format

```
sret
```

Description

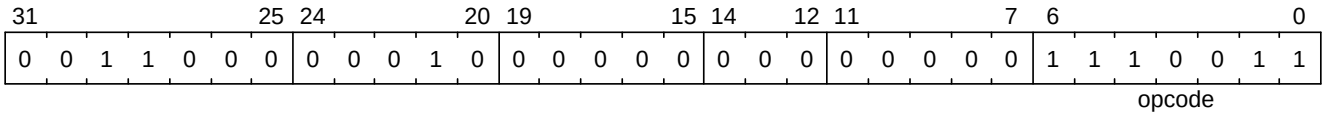
Return from traps in S-mode, and SRET copies SPIE into SIE, then sets SPIE.

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.

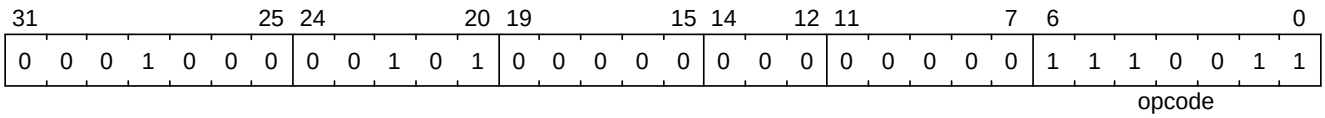
Implementation

```
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

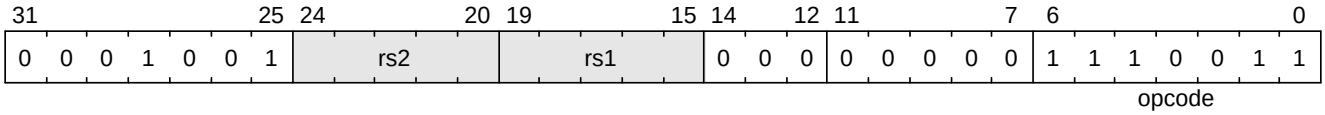
all privileged modes, and optionally available to U-mode. This instruction may raise an illegal instruction exception when TW=1 in mstatus.

Implementation

```
while (noInterruptsPending) idle
```

1.35. sfence.vma

Encoding



Format

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

Description

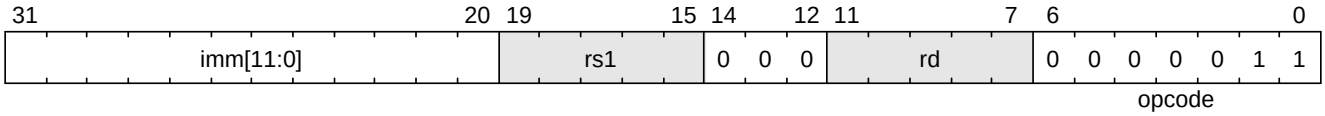
Guarantees that any previous stores already visible to the current RISC-V hart are ordered before all subsequent implicit references from that hart to the memory-management data structures. The SFENCE.VMA is used to flush any local hardware caches related to address translation. It is specified as a 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

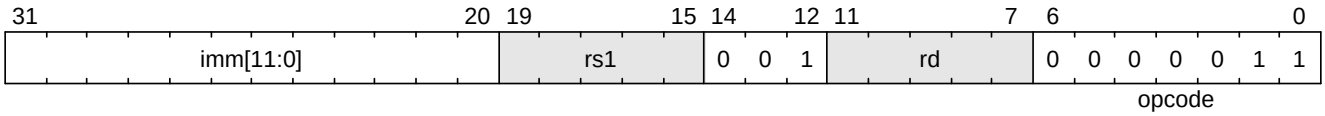
Loads a 8-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.37. lh

Encoding



Format

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

Description

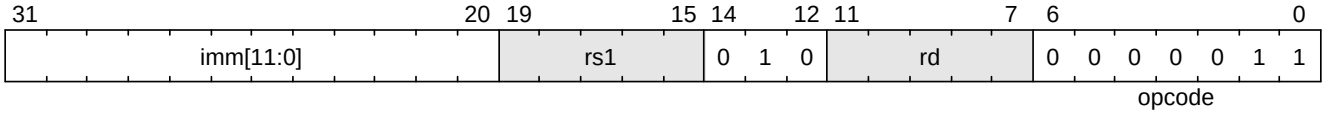
Loads a 16-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.38. lw

Encoding



Format

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

Description

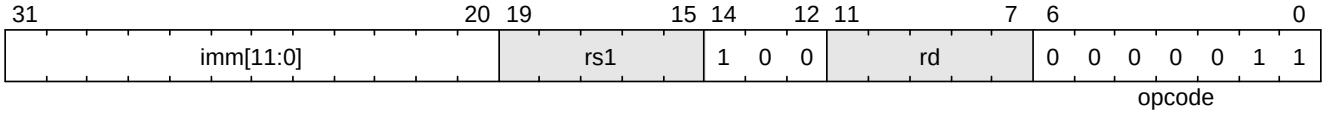
Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.39. lbu

Encoding



Format

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

Description

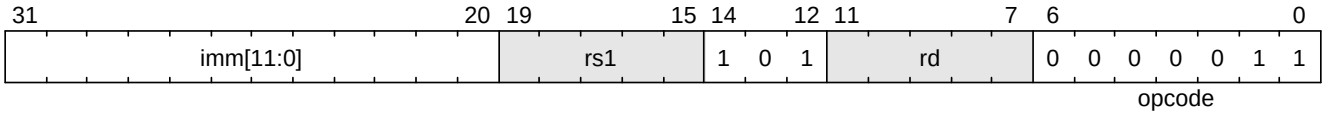
Loads a 8-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.40. lhu

Encoding



Format

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

Description

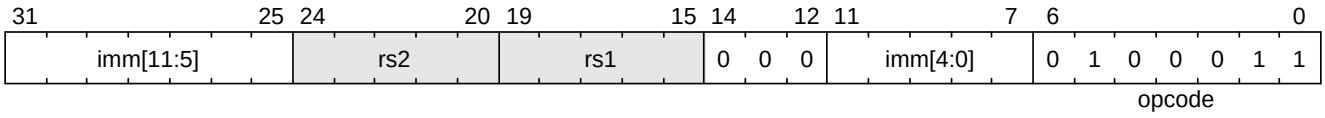
Loads a 16-bit value from memory and zero-extends this to XLEN bits before storing it in register rd.

Implementation

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

1.41. sb

Encoding



Format

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

Description

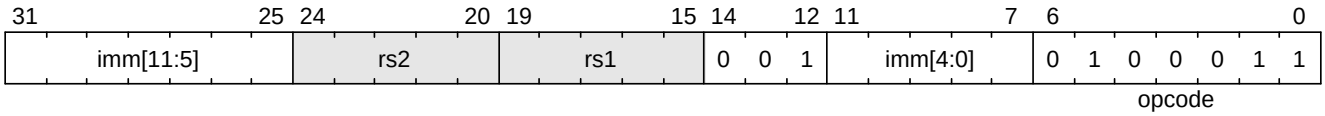
Store 8-bit, values from the low bits of register rs2 to memory.

Implementation

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

1.42. sh

Encoding



Format

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

Description

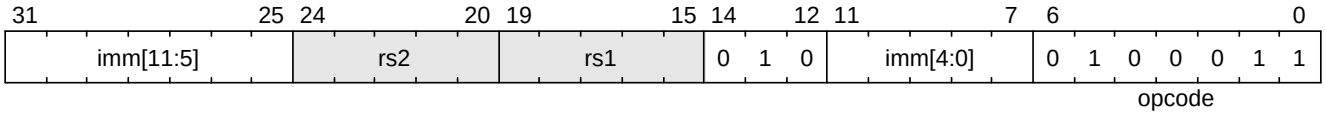
Store 16-bit, values from the low bits of register rs2 to memory.

Implementation

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

1.43. sw

Encoding



Format

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

Description

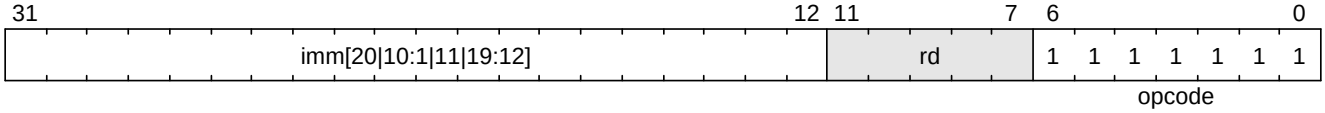
Store 32-bit, values from the low bits of register rs2 to memory.

Implementation

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

1.44. jal

Encoding



Format

```
jal rd,offset
```

Description

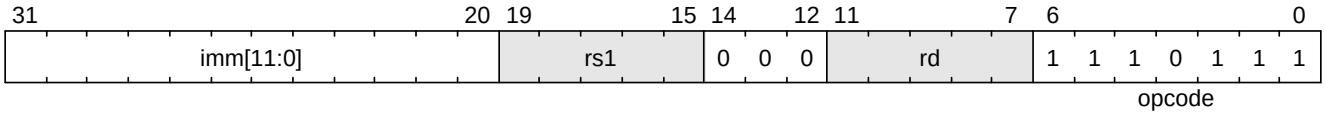
Jump to address and place return address in rd.

Implementation

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

1.45. jalr

Encoding



Format

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

Description

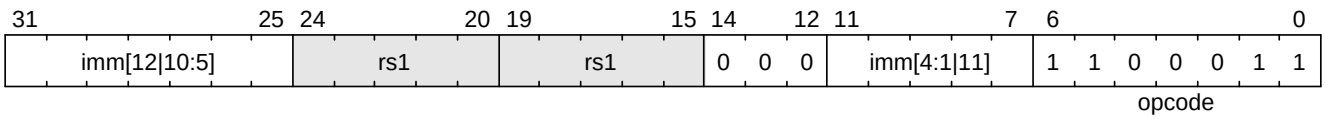
Jump to address and place return address in rd.

Implementation

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

1.46. beq

Encoding



Format

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

Description

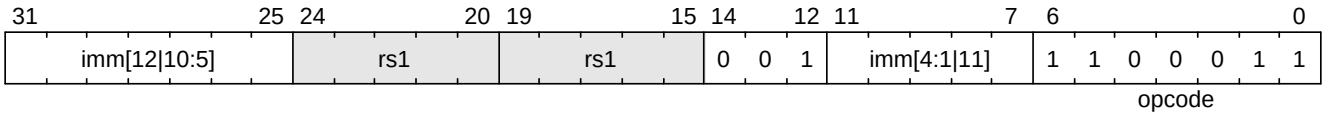
Take the branch if registers rs1 and rs2 are equal.

Implementation

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

1.47. bne

Encoding



Format

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

Description

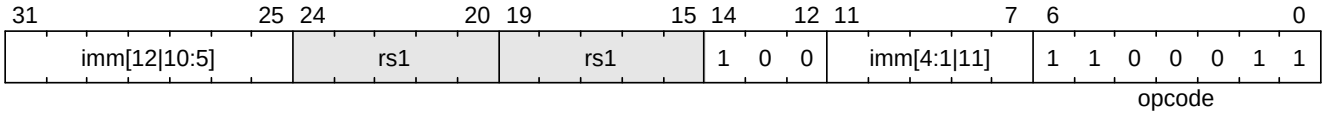
Take the branch if registers rs1 and rs2 are not equal.

Implementation

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

1.48. blt

Encoding



Format

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

Description

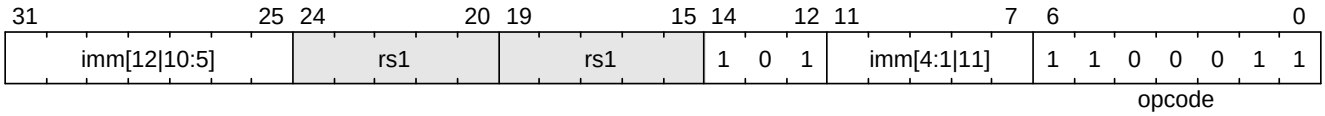
Take the branch if registers rs1 is less than rs2, using signed comparison.

Implementation

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

1.49. bge

Encoding



Format

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

Description

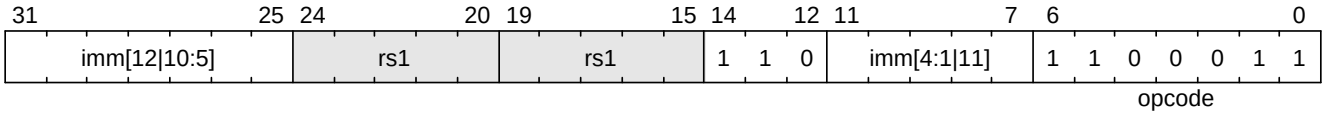
Take the branch if registers rs1 is greater than rs2, using signed comparison.

Implementation

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

1.50. bltu

Encoding



Format

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

Description

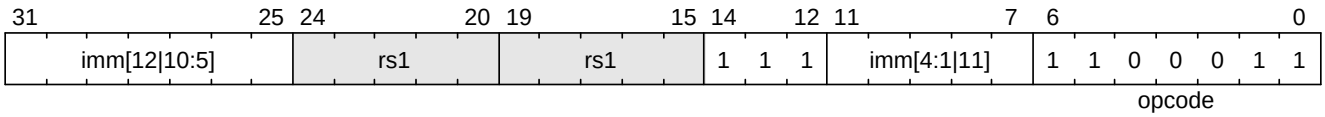
Take the branch if registers rs1 is less than rs2, using unsigned comparison.

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.

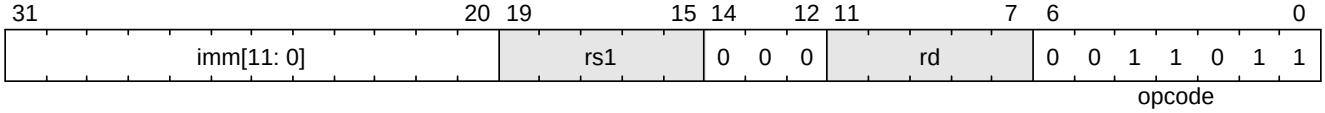
Implementation

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

2. RV64I Instructions

2.1. addiw

Encoding



Format

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

Description

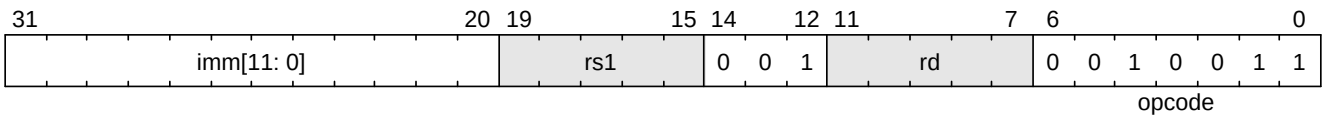
Adds the sign-extended 12-bit immediate to register rs1 and produces the proper sign-extension of a 32-bit result in rd. Overflows are ignored and the result is the low 32 bits of the result sign-extended to 64 bits. Note, ADDIW rd, rs1, 0 writes the sign-extension of the lower 32 bits of register rs1 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

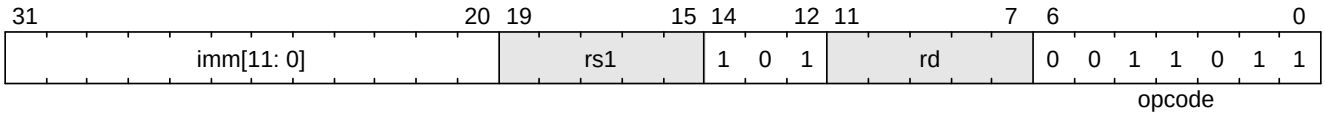
Performs logical left shift on the 32-bit of value in register rs1 by the shift amount held in the lower 5 bits of the immediate. Encodings with $imm[5] neq 0$ are reserved.

Implementation

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

2.3. srliw

Encoding



Format

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

Description

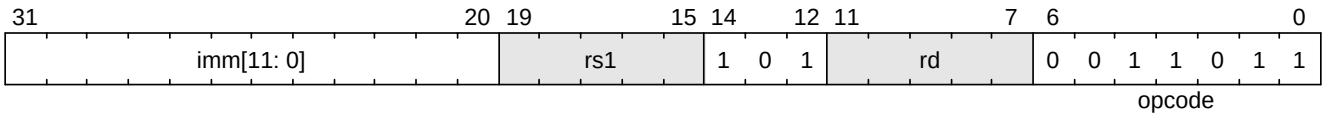
Performs logical right shift on the 32-bit of value in register rs1 by the shift amount held in the lower 5 bits of the immediate. Encodings with $imm[5] neq 0$ are reserved.

Implementation

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

2.4. sraiw

Encoding



Format

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

Description

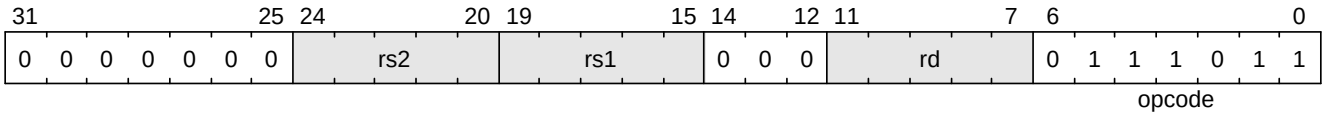
Performs arithmetic right shift on the 32-bit of value in register rs1 by the shift amount held in the lower 5 bits of the immediate. Encodings with $imm[5] neq 0$ are reserved.

Implementation

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

2.5. addw

Encoding



Format

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

Description

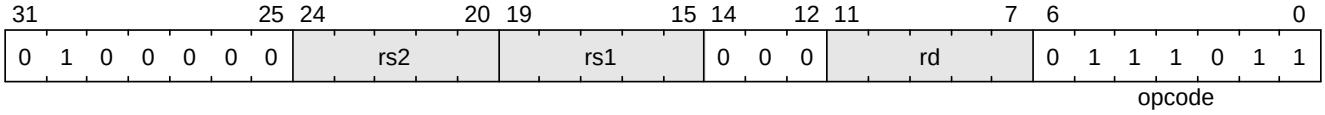
Adds the 32-bit of registers rs1 and 32-bit of register rs2 and stores the result in rd. Arithmetic overflow is ignored and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

Implementation

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

2.6. subw

Encoding



Format

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

Description

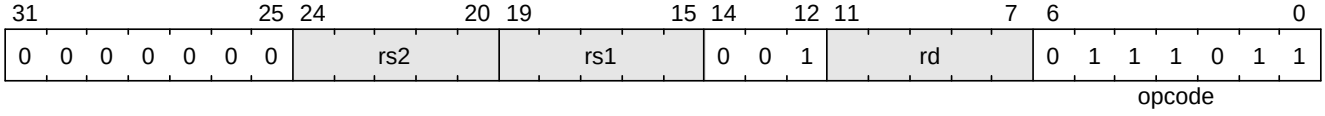
Subtract the 32-bit of registers rs1 and 32-bit of register rs2 and stores the result in rd. Arithmetic overflow is ignored and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

Implementation

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

2.7. sllw

Encoding



Format

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

Description

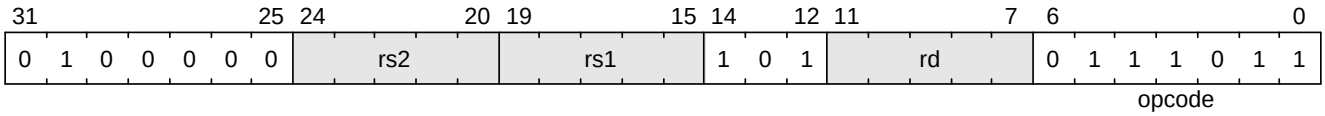
Performs logical left shift on the low 32-bits value in register rs1 by the shift amount held in the lower 5 bits of register rs2 and produce 32-bit results and written to the destination register rd.

Implementation

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

2.8. srlw

Encoding



Format

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

Description

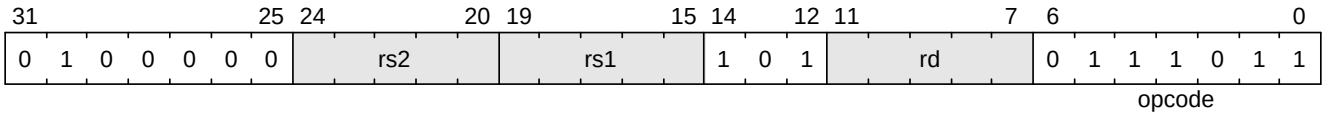
Performs logical right shift on the low 32-bits value in register rs1 by the shift amount held in the lower 5 bits of register rs2 and produce 32-bit results and written to the destination register rd.

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

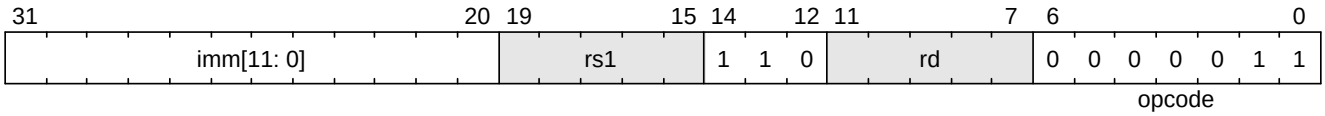
register rd.

Implementation

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

2.10. lwu

Encoding



Format

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

Description

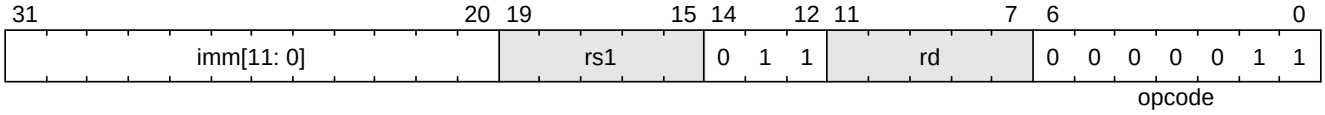
Loads a 32-bit value from memory and zero-extends this to 64 bits before storing it in register rd.

Implementation

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

2.11. ld

Encoding



Format

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

Description

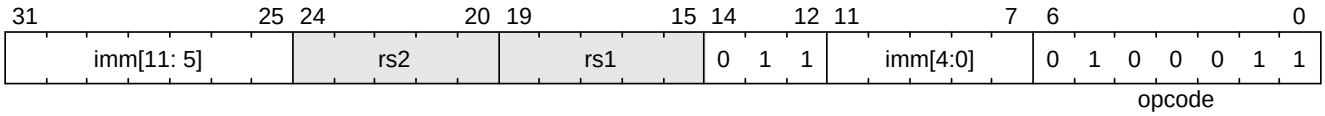
Loads a 64-bit value from memory into register rd for RV64I.

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.

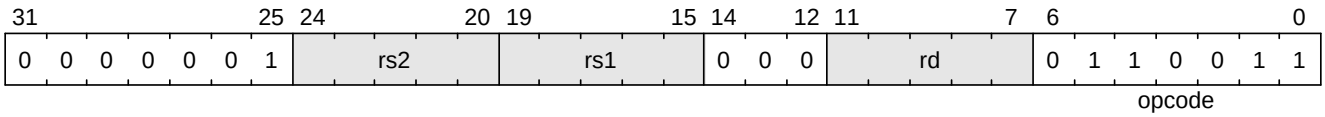
Implementation

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

3. RV32M, RV64M Instructions

3.1. mul

Encoding



Format

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

Description

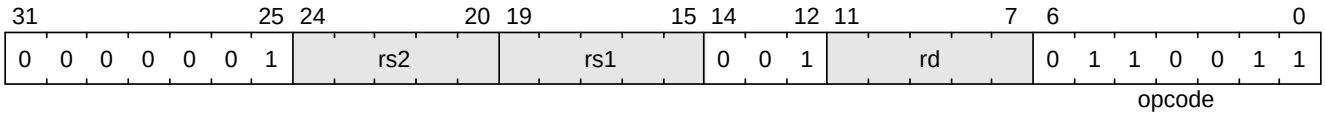
performs an XLEN-bit \times XLEN-bit multiplication of signed rs1 by signed rs2 and places the lower XLEN bits in the destination register.

Implementation

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

3.2. mulh

Encoding



Format

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

Description

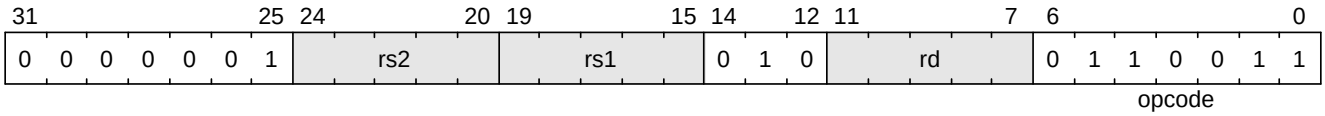
performs an XLEN-bit \times XLEN-bit multiplication of signed rs1 by signed rs2 and places the upper XLEN bits in the destination register.

Implementation

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

3.3. mulhsu

Encoding



Format

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

Description

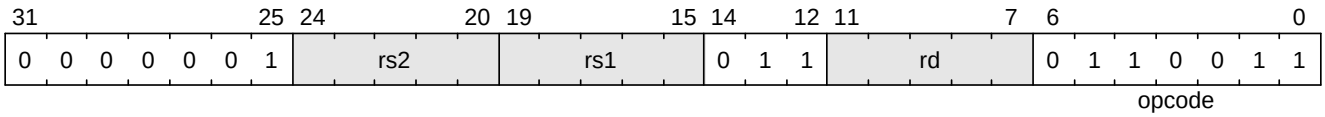
performs an XLEN-bit \times XLEN-bit multiplication of signed rs1 by unsigned rs2 and places the upper XLEN bits in the destination register.

Implementation

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

3.4. mulhu

Encoding



Format

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

Description

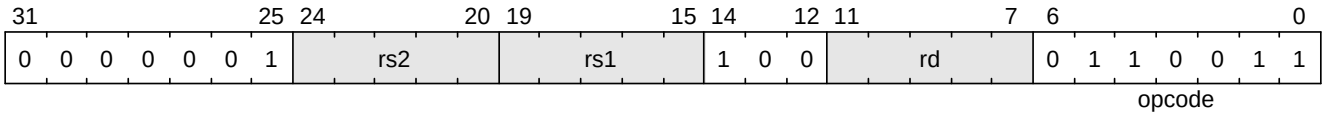
performs an XLEN-bit \times XLEN-bit multiplication of unsigned rs1 by unsigned rs2 and places the upper XLEN bits in the destination register.

Implementation

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

3.5. div

Encoding



Format

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

Description

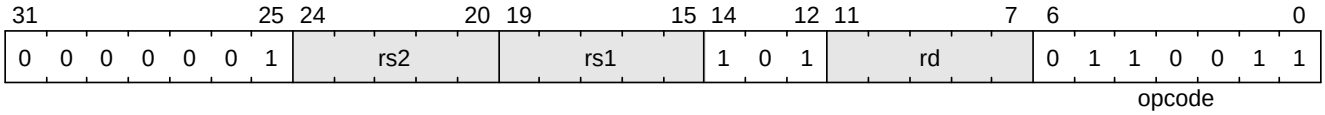
perform an XLEN bits by XLEN bits signed integer division of rs1 by rs2, rounding towards zero.

Implementation

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

3.6. divu

Encoding



Format

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

Description

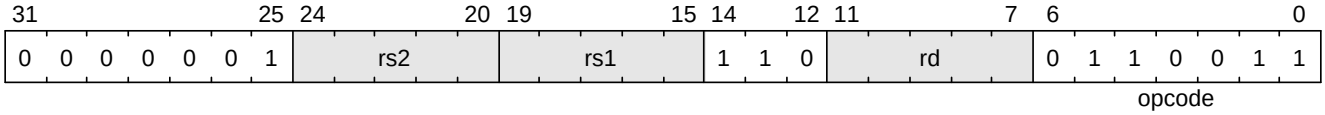
perform an XLEN bits by XLEN bits unsigned integer division of rs1 by rs2, rounding towards zero.

Implementation

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

3.7. rem

Encoding



Format

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

Description

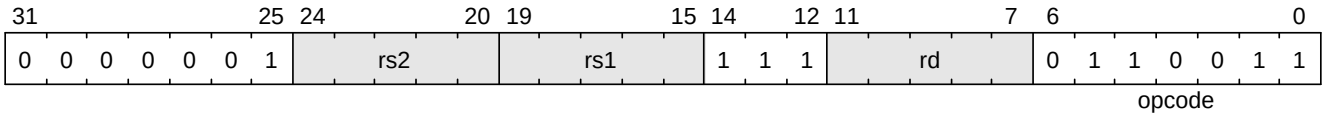
perform an XLEN bits by XLEN bits signed integer reminder of rs1 by rs2.

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.

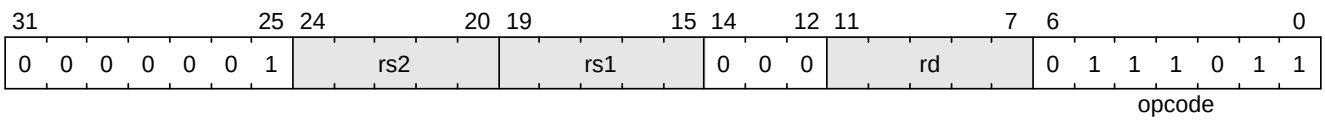
Implementation

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

4. RV64M Instructions

4.1. mulw

Encoding



Format

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

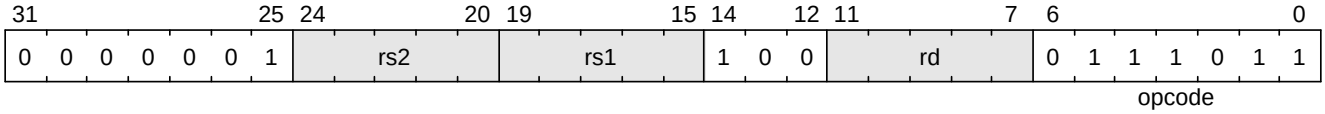
Description

Implementation

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

4.2. divw

Encoding



Format

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

Description

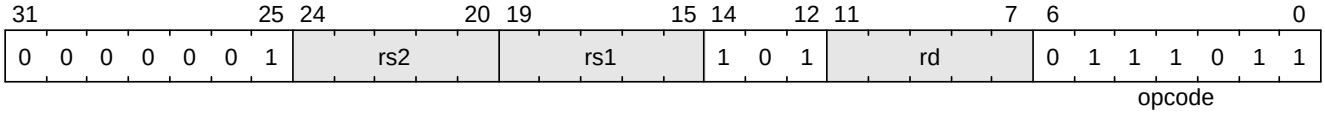
perform an 32 bits by 32 bits signed integer division of rs1 by rs2.

Implementation

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

=== divuw

Encoding



Format

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

Description

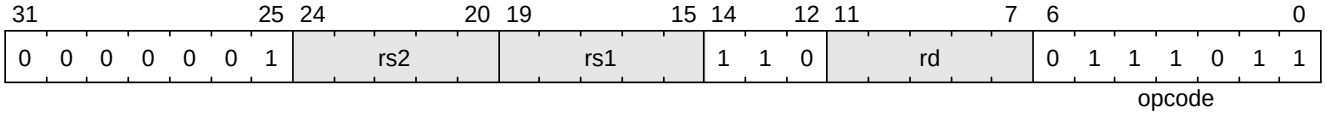
perform an 32 bits by 32 bits unsigned integer division of rs1 by rs2.

Implementation

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

4.3. remw

Encoding



Format

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

Description

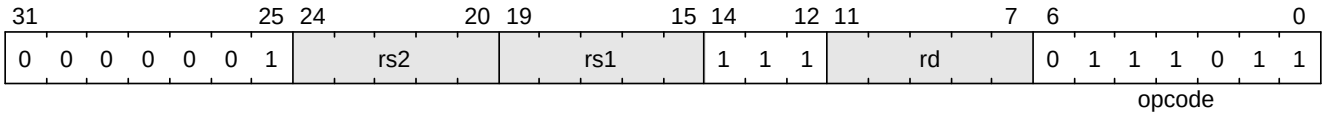
perform an 32 bits by 32 bits signed integer reminder of rs1 by rs2.

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.

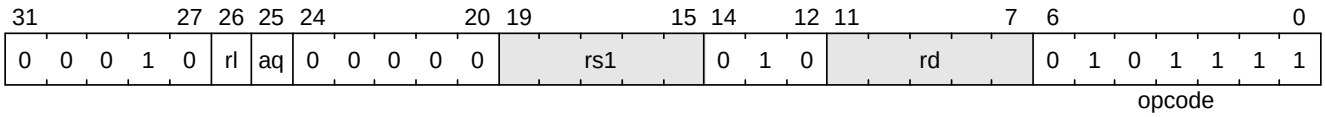
Implementation

```
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

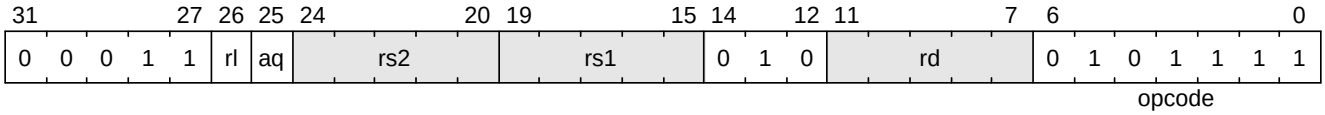
load a word from the address in rs1, places the sign-extended value in rd, and registers a reservation on the memory address.

Implementation

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

5.2. sc.w

Encoding



Format

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

Description

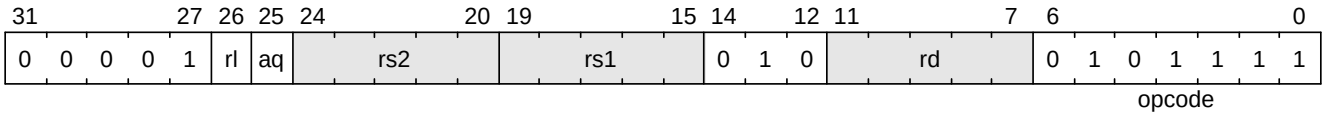
write a word in rs2 to the address in rs1, provided a valid reservation still exists on that address. SC writes zero to rd on success or a nonzero code on failure.

Implementation

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

5.3. amoswap.w

Encoding



Format

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

Description

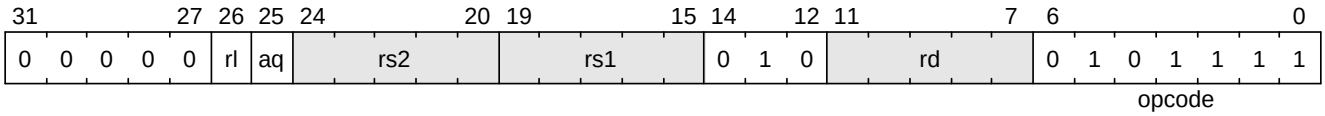
atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, swap the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.4. amoadd.w

Encoding



Format

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

Description

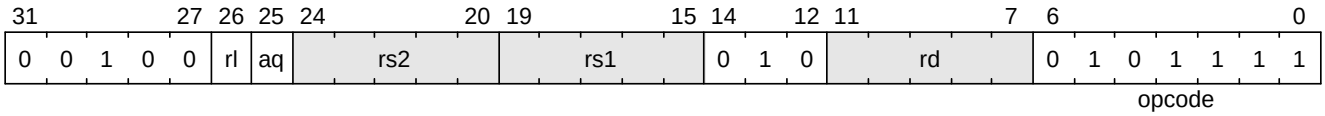
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply add the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.5. amoxor.w

Encoding



Format

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

Description

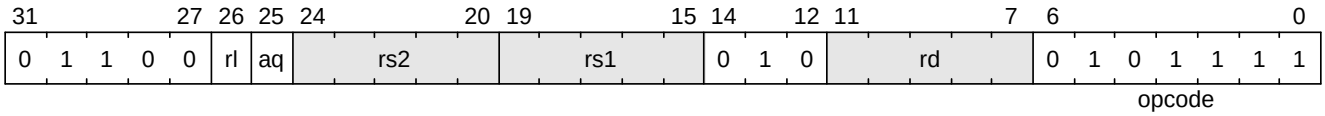
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply exclusive or the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.6. amoand.w

Encoding



Format

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

Description

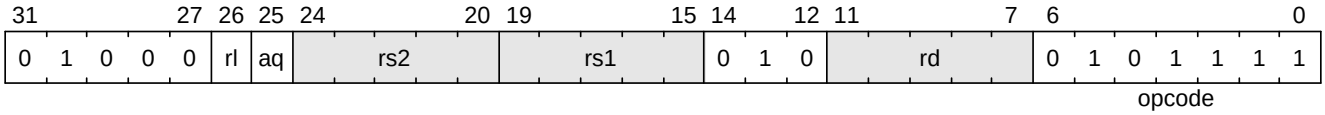
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply and the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.7. amoor.w

Encoding



Format

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

Description

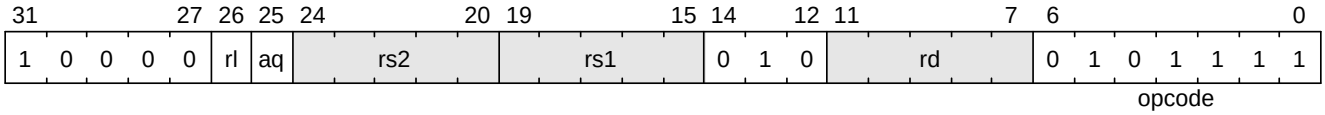
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply or the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

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

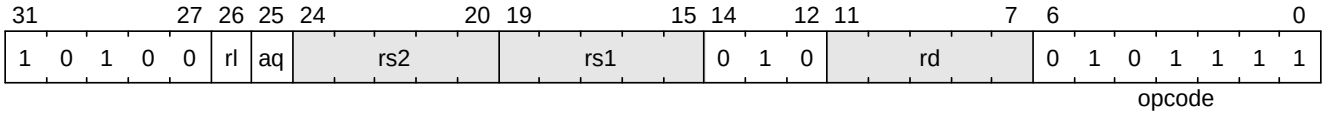
rd, apply min operator the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.9. amomax.w

Encoding



Format

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

Description

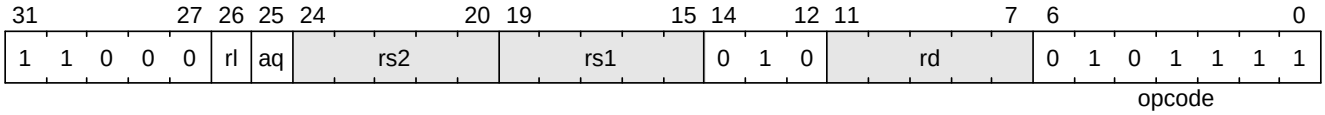
Atomically load a 32-bit signed data value from the address in rs1, place the value into register rd, apply max operator the loaded value and the original 32-bit signed value in rs2, then store the result back to the address in rs1.

Implementation

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

5.10. amominu.w

Encoding



Format

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

Description

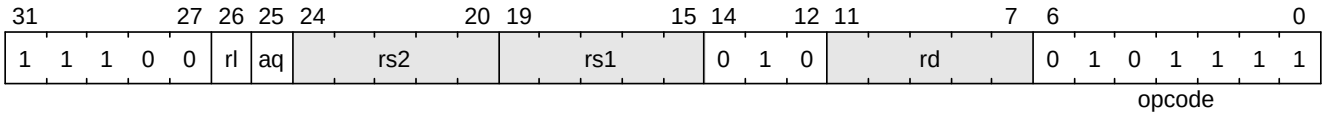
Atomically load a 32-bit unsigned data value from the address in rs1, place the value into register rd, apply unsigned min the loaded value and the original 32-bit unsigned value in rs2, then store the result back to the address in rs1.

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.

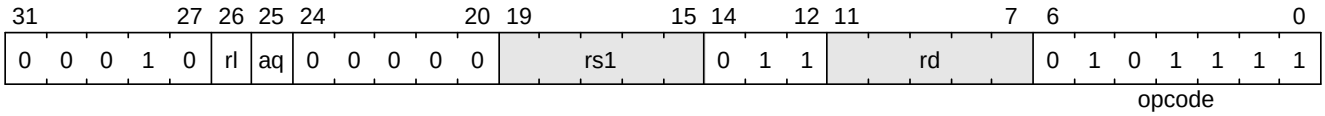
Implementation

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

6. RV64A Instructions

6.1. lr.d

Encoding



Format

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

Description

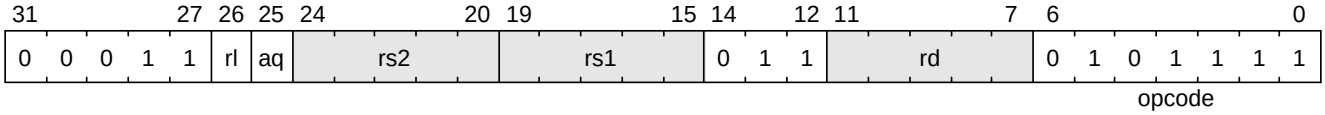
load a 64-bit data from the address in rs1, places value in rd, and registers a reservation on the memory address.

Implementation

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

6.2. sc.d

Encoding



Format

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

Description

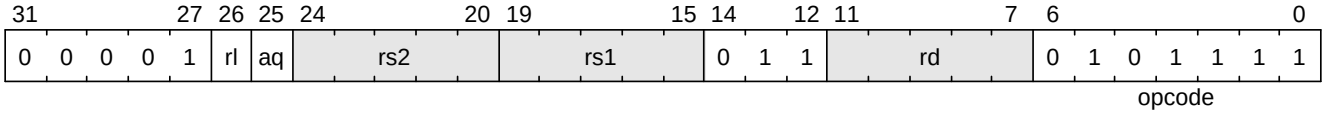
write a 64-bit data in rs2 to the address in rs1, provided a valid reservation still exists on that address. SC writes zero to rd on success or a nonzero code on failure.

Implementation

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

6.3. amoswap.d

Encoding



Format

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

Description

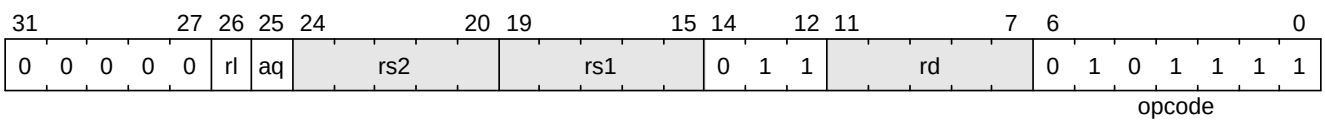
atomically load a 64-bit data value from the address in rs1, place the value into register rd, swap the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.4. amoadd.d

Encoding



Format

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

Description

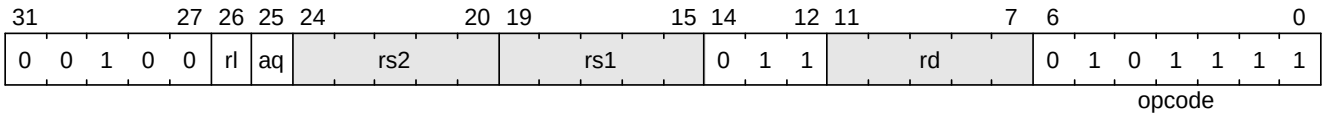
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply add the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.5. amoxor.d

Encoding



Format

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

Description

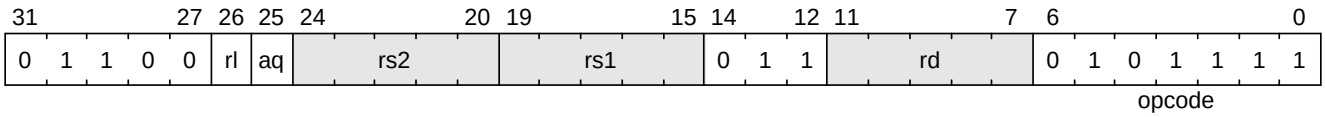
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply xor the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.6. amoand.d

Encoding



Format

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

Description

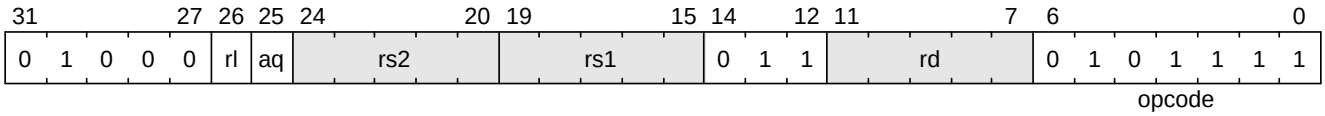
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply and the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.7. amoor.d

Encoding



Format

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

Description

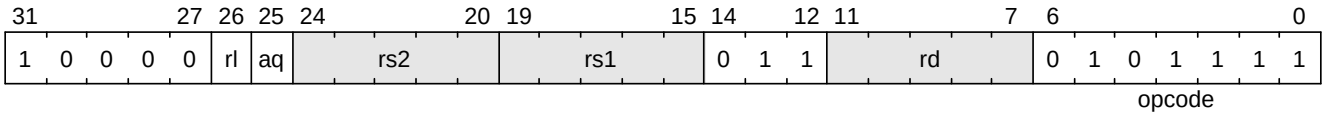
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply or the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.8. amomin.d

Encoding



Format

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

Description

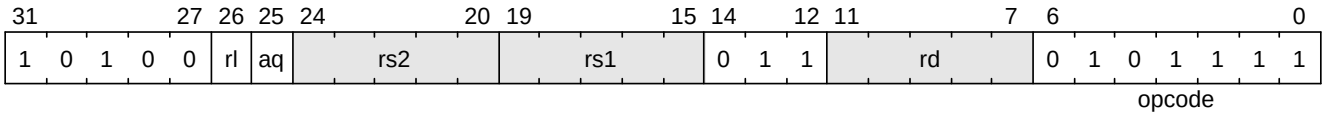
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply min the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.9. amomax.d

Encoding



Format

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

Description

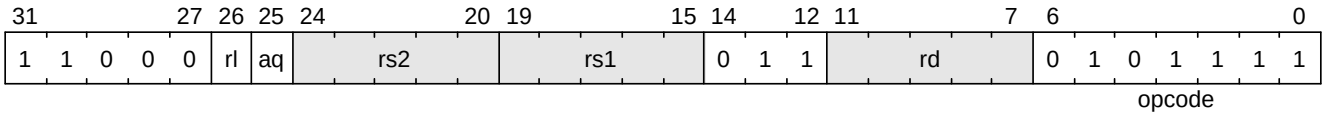
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply max the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

Implementation

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

6.10. amominu.d

Encoding



Format

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

Description

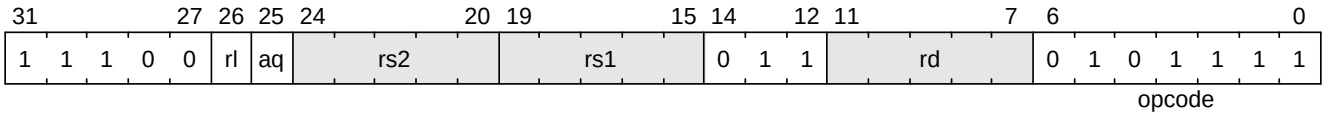
atomically load a 64-bit data value from the address in rs1, place the value into register rd, apply unsigned min the loaded value and the original 64-bit value in rs2, then store the result back to the address in rs1.

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

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	00	rs2	rs1	rm	rd	10000	11

Format

[verse] — fmadd.s rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = f[rs1]×f[rs2]+f[rs3] —

7.2. fmsub.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	00	rs2	rs1	rm	rd	10001	11

Format

[verse] — fmsub.s rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = f[rs1]×f[rs2]-f[rs3] —

7.3. fnmsub.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	00	rs2	rs1	rm	rd	10010	11

Format

[verse] — fnmsub.s rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = -f[rs1]×f[rs2+f[rs3] —

7.4. fnmadd.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	00	rs2	rs1	rm	rd	10011	11

Format

[verse] — fnmadd.s rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = -f[rs1]×f[rs2]-f[rs3] —

7.5. fadd.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	rm	rd	10100	11

Format

[verse] — fadd.s rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point addition. —

Implementation

[verse] — f[rd] = f[rs1] + f[rs2] —

7.6. fsub.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00001	00	rs2	rs1	rm	rd	10100	11

Format

[verse] — fsub.s rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point substraction. —

Implementation

[verse] — f[rd] = f[rs1] - f[rs2] —

7.7. fmul.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00010	00	rs2	rs1	rm	rd	10100	11

Format

[verse] — fmul.s rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point multiplication. —

Implementation

[verse] — f[rd] = f[rs1] × f[rs2] —

7.8. fdiv.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00011	00	rs2	rs1	rm	rd	10100	11

Format

[verse] — fdiv.s rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point division. —

Implementation

[verse] — f[rd] = f[rs1] / f[rs2] —

7.9. fsqrt.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01011	00	00000	rs1	rm	rd	10100	11

Format

[verse] — fsqrt.s rd,rs1 —

Description

[verse] — Perform single-precision square root. —

Implementation

[verse] — f[rd] = sqrt(f[rs1]) —

7.10. fsgnj.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00100	00	rs2	rs1	000	rd	10100	11

Format

[verse] — fsgnj.s rd,rs1,rs2 —

Description

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

Implementation

[verse] — f[rd] = \{f[rs2][31], f[rs1][30:0]} —

7.11. fsgnjn.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00100	00	rs2	rs1	001	rd	10100	11

Format

[verse] — fsgnjn.s rd,rs1,rs2 —

Description

[verse] — 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

[verse] — f[rd] = \{~f[rs2][31], f[rs1][30:0]} —

7.12. fsgnjx.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00100	00	rs2	rs1	010	rd	10100	11

Format

[verse] — fsgnjx.s rd,rs1,rs2 —

Description

[verse] — 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

[verse] — f[rd] = \{f[rs1][31] ^ f[rs2][31], f[rs1][30:0]} —

7.13. fmin.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00101	00	rs2	rs1	000	rd	10100	11

Format

[verse] — fmin.s rd,rs1,rs2 —

Description

[verse] — Write the smaller of single precision data in rs1 and rs2 to rd. —

Implementation

[verse] — f[rd] = min(f[rs1], f[rs2]) —

7.14. fmax.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00101	00	rs2	rs1	001	rd	10100	11

Format

[verse] — fmax.s rd,rs1,rs2 —

Description

[verse] — Write the larger of single precision data in rs1 and rs2 to rd. —

Implementation

[verse] — f[rd] = max(f[rs1], f[rs2]) —

7.15. fcvt.w.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	00	00000	rs1	rm	rd	10100	11

Format

[verse] — fcvt.w.s rd,rs1 —

Description

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

Implementation

[verse] — x[rd] = sext(s32{f32}(f[rs1])) —

7.16. fcvt.wu.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	00	00001	rs1	rm	rd	10100	11

Format

[verse] — fcvt.wu.s rd,rs1 —

Description

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

Implementation

[verse] — x[rd] = sext(u32{f32}(f[rs1])) —

7.17. fmv.x.w

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11100	00	00000	rs1	000	rd	10100	11

Format

[verse] — fmv.x.w rd,rs1 —

Description

[verse] — 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

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

7.18. feq.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
10100	00	rs2	rs1	010	rd	10100	11

Format

[verse] — feq.s rd,rs1,rs2 —

Description

[verse] — Performs a quiet equal comparison between 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

[verse] — x[rd] = f[rs1] == f[rs2] —

7.19. flt.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
10100	00	rs2	rs1	001	rd	10100	11

Format

[verse] — flt.s rd,rs1,rs2 —

Description

[verse] — Performs a quiet less comparison between 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

[verse] — x[rd] = f[rs1] < f[rs2] —

7.20. fle.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
10100	00	rs2	rs1	000	rd	10100	11

Format

[verse] — fle.s rd,rs1,rs2 —

Description

[verse] — Performs a quiet less or equal comparison between 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

[verse] — x[rd] = f[rs1] ⇐ f[rs2] —

7.21. fclass.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11100	00	00000	rs1	001	rd	10100	11

Format

[verse] — fclass.s rd,rs1 —

Description

[verse] — Examines the value in 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

[verse] — x[rd] = classifys(f[rs1]) —

7.22. fcvt.s.w

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	00	00000	rs1	rm	rd	10100	11

Format

[verse] — fcvt.s.w rd,rs1 —

Description

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

Implementation

[verse] — f[rd] = f32{s32}(x[rs1]) —

7.23. fcvt.s.wu

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	00	00001	rs1	rm	rd	10100	11

Format

[verse] — fcvt.s.wu rd,rs1 —

Description

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

Implementation

[verse] — f[rd] = f32{u32}(x[rs1]) —

7.24. fmv.w.x

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11110	00	00000	rs1	000	rd	10100	11

Format

[verse] — fmv.w.x rd,rs1 —

Description

[verse] — 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

[verse] — f[rd] = x[rs1][31:0] —

7.25. fmadd.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	01	rs2	rs1	rm	rd	10000	11

Format

[verse] — fmadd.d rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = f[rs1]×f[rs2]+f[rs3] —

7.26. fmsub.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	01	rs2	rs1	rm	rd	10001	11

Format

[verse] — fmsub.d rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = f[rs1]×f[rs2]-f[rs3] —

7.27. fnmsub.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	01	rs2	rs1	rm	rd	10010	11

Format

[verse] — fnmsub.d rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = -f[rs1]×f[rs2+f[rs3] —

7.28. fnmadd.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
rs3	01	rs2	rs1	rm	rd	10011	11

Format

[verse] — fnmadd.d rd,rs1,rs2,rs3 —

Description

[verse] — Perform single-precision fused multiply addition. —

Implementation

[verse] — f[rd] = -f[rs1]×f[rs2]-f[rs3] —

7.29. fadd.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	01	rs2	rs1	rm	rd	10100	11

Format

[verse] — fadd.d rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point addition. —

Implementation

[verse] — f[rd] = f[rs1] + f[rs2] —

7.30. fsub.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00001	01	rs2	rs1	rm	rd	10100	11

Format

[verse] — fsub.d rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point addition. —

Implementation

[verse] — f[rd] = f[rs1] - f[rs2] —

7.31. fmul.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00010	01	rs2	rs1	rm	rd	10100	11

Format

[verse] — fmul.d rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point addition. —

Implementation

[verse] — f[rd] = f[rs1] × f[rs2] —

7.32. fdiv.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00011	01	rs2	rs1	rm	rd	10100	11

Format

[verse] — fdiv.d rd,rs1,rs2 —

Description

[verse] — Perform single-precision floating-point addition. —

Implementation

[verse] — f[rd] = f[rs1] / f[rs2] —

7.33. fsqrt.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01011	01	00000	rs1	rm	rd	10100	11

Format

[verse] — fsqrt.d rd,rs1 —

Description

[verse] — Perform single-precision square root. —

Implementation

[verse] — f[rd] = sqrt(f[rs1]) —

7.34. fsgnj.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00100	01	rs2	rs1	000	rd	10100	11

Format

[verse] — fsgnj.d rd,rs1,rs2 —

Description

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

Implementation

[verse] — f[rd] = \{f[rs2][63], f[rs1][62:0]} —

7.35. fsgnjn.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00100	01	rs2	rs1	001	rd	10100	11

Format

[verse] — fsgnjn.d rd,rs1,rs2 —

Description

[verse] — 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

[verse] — f[rd] = \{~f[rs2][63], f[rs1][62:0]} —

7.36. fsgnjx.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00100	01	rs2	rs1	010	rd	10100	11

Format

[verse] — fsgnjx.d rd,rs1,rs2 —

Description

[verse] — 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

[verse] — f[rd] = \{f[rs1][63] ^ f[rs2][63], f[rs1][62:0]} —

7.37. fmin.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00101	01	rs2	rs1	000	rd	10100	11

Format

[verse] — fmin.d rd,rs1,rs2 —

Description

[verse] — Write the smaller of single precision data in rs1 and rs2 to rd. —

Implementation

[verse] — f[rd] = min(f[rs1], f[rs2]) —

7.38. fmax.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0

00101	01	rs2	rs1	001	rd	10100	11

Format

[verse] — fmax.d rd,rs1,rs2 —

Description

[verse] — Write the larger of single precision data in rs1 and rs2 to rd. —

Implementation

[verse] — f[rd] = max(f[rs1], f[rs2]) —

7.39. fcvt.s.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01000	00	00001	rs1	rm	rd	10100	11

Format

[verse] — fcvt.s.d rd,rs1 —

Description

[verse] — Converts double floating-point register in rs1 into a floating-point number in floating-point register rd. —

Implementation

[verse] — f[rd] = f32{f64}(f[rs1]) —

7.40. fcvt.d.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01000	01	00000	rs1	rm	rd	10100	11

Format

[verse] — fcvt.d.s rd,rs1 —

Description

[verse] — Converts single floating-point register in rs1 into a double floating-point number in floating-point register rd. —

Implementation

[verse] — f[rd] = f64{f32}(f[rs1]) —

7.41. feq.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
10100	01	rs2	rs1	010	rd	10100	11

Format

[verse] — feq.d rd,rs1,rs2 —

Description

[verse] — Performs a quiet equal comparison between 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

[verse] — x[rd] = f[rs1] == f[rs2] —

7.42. flt.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
10100	01	rs2	rs1	001	rd	10100	11

Format

[verse] — flt.d rd,rs1,rs2 —

Description

[verse] — Performs a quiet less comparison between 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

[verse] — x[rd] = f[rs1] < f[rs2] —

7.43. fle.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
10100	01	rs2	rs1	000	rd	10100	11

Format

[verse] — fle.d rd,rs1,rs2 —

Description

[verse] — Performs a quiet less or equal comparison between 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

[verse] — x[rd] = f[rs1] ⇐ f[rs2] —

7.44. fclass.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11100	01	00000	rs1	001	rd	10100	11

Format

[verse] — fclass.d rd,rs1 —

Description

[verse] — Examines the value in 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

[verse] — x[rd] = classifys(f[rs1]) —

7.45. fcvt.w.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	01	00000	rs1	rm	rd	10100	11

Format

[verse] — fcvt.w.d rd,rs1 —

Description

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

Implementation

[verse] — x[rd] = sext(s32{f64}(f[rs1])) —

7.46. fcvt.wu.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	01	00001	rs1	rm	rd	10100	11

Format

[verse] — fcvt.wu.d rd,rs1 —

Description

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

Implementation

[verse] — x[rd] = sext(u32f64(f[rs1])) —

7.47. fcvt.d.w

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	01	00000	rs1	rm	rd	10100	11

Format

[verse] — fcvt.d.w rd,rs1 —

Description

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

Implementation

[verse] — x[rd] = sext(s32{f64}(f[rs1])) —

7.48. fcvt.d.wu

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	01	00001	rs1	rm	rd	10100	11

Format

[verse] — fcvt.d.wu rd,rs1 —

Description

[verse] — Converts a 32-bit unsigned integer, in integer register rs1 into a double-precision

floating-point number in floating-point register rd. —

Implementation

[verse] — f[rd] = f64{u32}(x[rs1]) —

7.49. flw

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11	:0]		rs1	010	rd	00001	11

Format

[verse] — flw rd,offset(rs1) —

Description

[verse] — Load a single-precision floating-point value from memory into floating-point register rd. —

Implementation

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

7.50. fsw

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11	:5]	rs2	rs1	010	imm[4:0]	01001	11

Format

[verse] — fsw rs2,offset(rs1) —

Description

[verse] — Store a single-precision value from floating-point register rs2 to memory. —

Implementation

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

7.51. fld

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11	:0]		rs1	011	rd	00001	11

Format

[verse] — fld rd,rs1,offset —

Description

[verse] — Load a double-precision floating-point value from memory into floating-point register rd. —

Implementation

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

7.52. fsd

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11	:5]	rs2	rs1	011	imm[4:0]	01001	11

Format

[verse] — fsd rs2,offset(rs1) —

Description

[verse] — Store a double-precision value from the floating-point registers to memory. —

Implementation

[verse] — 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

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	00	00010	rs1	rm	rd	10100	11

Format

[verse] — fcvt.l.s rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — x[rd] = s64{f32}(f[rs1]) —

8.2. fcvt.lu.s

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	00	00011	rs1	rm	rd	10100	11

Format

[verse] — fcvt.lu.s rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — x[rd] = u64{f32}(f[rs1]) —

8.3. fcvt.s.l

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	00	00010	rs1	rm	rd	10100	11

Format

[verse] — fcvt.s.l rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — f[rd] = f32{s64}(x[rs1]) —

8.4. fcvt.s.lu

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	00	00011	rs1	rm	rd	10100	11

Format

[verse] — fcvt.s.lu rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — f[rd] = f32{u64}(x[rs1]) — == RV64D Instructions

8.5. fcvt.l.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	01	00010	rs1	rm	rd	10100	11

Format

[verse] — fcvt.l.d rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — x[rd] = s64{f64}(f[rs1]) —

8.6. fcvt.lu.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11000	01	00011	rs1	rm	rd	10100	11

Format

[verse] — fcvt.lu.d rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — x[rd] = u64{f64}(f[rs1]) —

8.7. fmv.x.d

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11100	01	00000	rs1	000	rd	10100	11

Format

[verse] — fmv.x.d rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — x[rd] = f[rs1][63:0] —

8.8. fcvt.d.l

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	01	00010	rs1	rm	rd	10100	11

Format

[verse] — fcvt.d.l rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — f[rd] = f64{s64}(x[rs1]) —

8.9. fcvt.d.lu

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11010	01	00011	rs1	rm	rd	10100	11

Format

[verse] — fcvt.d.lu rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — f[rd] = f64{u64}(x[rs1]) —

8.10. fmv.d.x

cccc|

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
11110	01	00000	rs1	000	rd	10100	11

Format

[verse] — fmv.d.x rd,rs1 —

Description

[verse] —

```
--
```

Implementation

[verse] — f[rd] = x[rs1][63:0] — == RV32C, RV64C Instructions

8.11. c.addi4spn

cccc|

15-13	12-5	4-2	1-0
000	imm	rd'	00

Format

[verse] — c.addi4spn rd',uimm —

Description

[verse] — 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

[verse] — x[8+rd'] = x[2] + uimm —

Expansion

[verse] — addi x2,x2,nzimm[9:4] —

8.12. c.fld

cccc|

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

Format

[verse] — c.fld rd',uimm(rs1') —

Description

[verse] — 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

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

Expansion

[verse] — fld rd',offset[7:3](rs1') —

8.13. c.lw

cccc|

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

Format

[verse] — c.lw rd',uimm(rs1') —

Description

[verse] — 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

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

Expansion

[verse] — lw rd',offset[6:2](rs1') —

8.14. c.flw

cccc|

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

Format

[verse] — c.flw rd',uimm(rs1') —

Description

[verse] — 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

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

Expansion

[verse] — lw rd',offset[6:2](rs1') —

8.15. c.ld

cccc|

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

Format

[verse] — c.ld rd',uimm(rs1') —

Description

[verse] — 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

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

Expansion

[verse] — ld rd', offset[7:3](rs1') —

8.16. c.fsd

cccc|

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

Format

[verse] — c.fsd rd',uimm(rs1') —

Description

[verse] — 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

[verse] — M[x[8+rs1'] + uimm][63:0] = f[8+rs2'] —

Expansion

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

8.17. c.sw

cccc|

15-13	12-10	9-7	6-5	4-2	1-0

110	uimm[5:3]	rs1'	uimm[2	6]	rs2'

Format

[verse] — c.sw rd',uimm(rs1') —

Description

[verse] — 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

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

Expansion

[verse] — sw rs2',offset[6:2](rs1') —

8.18. c.fsw

cccc|

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

Format

[verse] — c.fsw rd',uimm(rs1') —

Description

[verse] — 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

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

Expansion

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

8.19. c.sd

cccc|

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

Format

[verse] — c.sd rd',uimm(rs1') —

Description

[verse] — 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

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

Expansion

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

8.20. c.nop

cccc|

15-13	12-10	9-7	6-5	4-2	1-0
000	0	0	0	0	01

Format

[verse] — c.nop —

Description

[verse] — Does not change any user-visible state, except for advancing the pc. —

Implementation

[verse] — None —

Expansion

[verse] — addi x0, x0, 0 —

8.21. c.addi

cccc|

15-13	12	11-7	6-2	1-0
000	nzimm[5]	rs1/rd!=0	nzimm[4:0]	01

Format

[verse] — c.addi rd,u[12:12]|u[6:2] —

Description

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

Implementation

[verse] — x[rd] = x[rd] + sext(imm) —

Expansion

[verse] — addi rd, rd, nzimm[5:0] —

8.22. c.jal

cccc|

15-13	12-2	1-0
001	imm[119:863:1	5]

Format

[verse] — c.jal offset —

Description

[verse] — Jump to address and place return address in rd. —

Implementation

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

Expansion

[verse] — jal x1, offset[11:1] —

8.23. c.addiw

cccc|

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

Format

[verse] — c.addiw rd,imm —

Description

[verse] — 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

[verse] — x[rd] = sextx[rd] + sext(imm[31:0]) —

Expansion

[verse] — addiw rd,rd,imm[5:0] —

8.24. c.li

cccc|

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

Format

[verse] — c.li rd,uimm —

Description

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

Implementation

[verse] — x[rd] = sext(imm) —

Expansion

[verse] — addi rd,x0,imm[5:0] —

8.25. c.addi16sp

cccc|

15-13	12	11-7	6-2	1-0
011	imm[9]	00010	imm[48:7	5]

Format

[verse] — c.addi16sp imm —

Description

[verse] — 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

[verse] — x[2] = x[2] + sext(imm) —

Expansion

[verse] — addi x2,x2, nzimm[9:4] —

8.26. c.lui

cccc|

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

Format

[verse] — c.lui rd,uimm —

Description

[verse] —

```
--
```

Implementation

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

Expansion

[verse] — lui rd,nzuimm[17:12] —

8.27. c.srli

cccc|

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

Format

[verse] — c.srli rd',uimm —

Description

[verse] — 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

[verse] — x[8+rd'] = x[8+rd'] >>u uimm —

Expansion

[verse] — srli rd',rd',64 —

8.28. c.srai

cccc|

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

Format

[verse] — c.srai rd',uimm —

Description

[verse] — 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

[verse] — x[8+rd'] = x[8+rd'] >>s uimm —

Expansion

[verse] — srai rd',rd',shamt[5:0] —

8.29. c.andi

cccc|

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

Format

[verse] — c.andi rd',uimm —

Description

[verse] — 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

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

Expansion

[verse] — andi rd',rd',imm[5:0] —

8.30. c.sub

cccc|

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

Format

[verse] — c.sub rd',rd' —

Description

[verse] — Subtract the value in register rs2' from the value in register rd', then writes the result to register rd'. —

Implementation

[verse] — x[8+rd'] = x[8+rd'] - x[8+rs2'] —

Expansion

[verse] — sub rd',rd',rs2' —

8.31. c.xor

cccc|

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

Format

[verse] — c.xor rd',rd' —

Description

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

Implementation

[verse] — x[8+rd'] = x[8+rd'] ^ x[8+rs2'] —

Expansion

[verse] — xor rd',rd',rs2' —

8.32. c.or

cccc|

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

Format

[verse] — c.or rd',rd' —

Description

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

Implementation

[verse] — x[8+rd'] = x[8+rd'] | x[8+rs2'] —

Expansion

[verse] — or rd',rd',rs2 —

8.33. c.and

cccc|

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

Format

[verse] — c.and rd',rd' —

Description

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

Implementation

[verse] — x[8+rd'] = x[8+rd'] & x[8+rs2'] —

Expansion

[verse] — and rd',rd',rs2' —

8.34. c.subw

cccc|

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

Format

[verse] — c.subw rd',rs2' —

Description

[verse] — 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

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

Expansion

[verse] — subw rd',rd',rs2' —

8.35. c.addw

cccc|

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

Format

[verse] — c.addw rd',rs2' —

Description

[verse] — 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

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

Expansion

[verse] — addw rd',rd',rs2' —

8.36. c.j

cccc|

15-13	12-2	1-0
101	imm[119:863:1	5]

Format

[verse] — c.j offset —

Description

[verse] — Unconditional control transfer. —

Implementation

[verse] — pc += sext(offset) —

Expansion

[verse] — jal x0,offset[11:1] —

8.37. c.beqz

cccc|

15-13	12-10	9-7	6-2	1-0
110	offset[8	4:3]	rs1'	offset[7:65]

Format

[verse] — c.beqz rs1',offset —

Description

[verse] — Take the branch if the value in register rs1' is zero. —

Implementation

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

Expansion

[verse] — beq rs1',x0,offset[8:1] —

8.38. c.bnez

cccc|

15-13	12-10	9-7	6-2	1-0
111	offset[8	4:3]	rs1'	offset[7:65]

Format

[verse] — c.bnez rs1',offset —

Description

[verse] — Take the branch if the value in register rs1' is not zero. —

Implementation

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

Expansion

[verse] — bne rs1',x0,offset[8:1] —

8.39. c.slli

cccc|

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

Format

[verse] — c.slli rd,uimm —

Description

[verse] — 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

[verse] — x[rd] = x[rd] << uimm —

Expansion

[verse] — slli rd,rd,shamt[5:0] —

8.40. c.fldsp

cccc|

15-13	12	11-7	6-2	1-0

001	uimm[5]	rd	uimm[4:3	8:6]

Format

[verse] — c.fldsp rd,uimm(x2) —

Description

[verse] — 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

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

Expansion

[verse] — fld rd,offset[8:3](x2) —

8.41. c.lwsp

cccc|

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

Format

[verse] — c.lwsp rd,uimm(x2) —

Description

[verse] — 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

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

Expansion

[verse] — lw rd,offset[7:2](x2) —

8.42. c.flwsp

cccc|

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

Format

[verse] — c.flwsp rd,uimm(x2) —

Description

[verse] — 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

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

Expansion

[verse] — flw rd,offset[7:2](x2) —

8.43. c.ldsp

cccc|

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

Format

[verse] — c.ldsp rd,uimm(x2) —

Description

[verse] — 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

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

Expansion

[verse] — ld rd,offset[8:3](x2) —

8.44. c.jr

cccc|

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

Format

[verse] — c.jr rs1 —

Description

[verse] — Performs an unconditional control transfer to the address in register rs1. —

Implementation

[verse] — pc = x[rs1] —

Expansion

[verse] — jalr x0,rs1,0 —

8.45. c.mv

cccc|

15-13	12	11-7	6-2	1-0
100	0	rs1	rs2	10

Format

[verse] — c.mv rd,rs2' —

Description

[verse] — Copy the value in register rs2 into register rd. —

Implementation

[verse] — x[rd] = x[rs2] —

Expansion

[verse] — add rd, x0, rs2 —

8.46. c.ebreak

cccc|

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

Format

[verse] — c.ebreak —

Description

[verse] — Cause control to be transferred back to the debugging environment. —

Implementation

[verse] — RaiseException(Breakpoint) —

Expansion

[verse] — ebreak —

8.47. c.jalr

cccc|

15-13	12	11-7	6-2	1-0

100	1	rs1	00000	10

Format

[verse] — c.jalr rd —

Description

[verse] — Jump to address and place return address in rd. —

Implementation

[verse] — t = pc+2; pc = x[rs1]; x[1] = t —

Expansion

[verse] — jalr x1,rs1,0 —

8.48. c.add

cccc|

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

Format

[verse] — c.add rd,rs2' —

Description

[verse] — Add the values in registers rd and rs2 and writes the result to register rd. —

Implementation

[verse] — x[rd] = x[rd] + x[rs2] —

Expansion

[verse] — add rd,rd,rs2 —

8.49. c.fsdsp

cccc|

15-13	12-7	4-2	1-0
101	uimm[5:3	8:6]	rs2

Format

[verse] — c.fsdsp rs2,uimm(x2) —

Description

[verse] — 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

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

Expansion

[verse] — fsd rs2,offset[8:3](x2) —

8.50. c.swsp

cccc|

15-13	12-7	4-2	1-0
110	uimm[5:2	7:6]	rs2

Format

[verse] — c.swsp rs2,uimm(x2) —

Description

[verse] — 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

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

Expansion

[verse] — sw rs2,offset[7:2](x2) —

8.51. c.fswsp

cccc|

15-13	12-7	4-2	1-0
111	uimm[5:2	7:6]	rs2

Format

[verse] — c.fswsp rs2,uimm(rs2) —

Description

[verse] — 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

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

Expansion

[verse] — fsw rs2,offset[7:2](x2) —

8.52. c.sdsp

cccc|

15-13	12-7	4-2	1-0
111	uimm[5:3	8:6]	rs2

Format

[verse] — c.sdsp rs2,uimm(x2) —

Description

[verse] — 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

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

Expansion

[verse] — sd rs2,offset[8:3](x2) —

9. Register Definitions

9.1. 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

Register	ABI Name	Description	Saver
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
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

9.2. 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

Register	ABI Name	Description	Saver
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
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