The RISC-V OTTER Assembly Language Manual

Version: 1.01

Table of Contents

Table of Contents ..... - 2 -
Acknowledgements ..... - 4 -
The RISC-V Assembler ..... - 5 -
The RISC-V OTTER Instruction Set ..... - 6 -
RISC-V OTTER Assembly Instructions Formats ..... - 6 -
Instruction Type: R-type ..... - 7 -
Instruction Type: I-type ..... - 8 -
Instruction Type: S-type ..... - 9 -
Instruction Type: B-type ..... - 10 -
Instruction Type: U-type ..... - 11 -
Instruction Type: J-type ..... - 11 -
RISC-V OTTER Assembly Instructions Brief Listing ..... - 12 -
RISC-V OTTER Assembly Instruction Overview ..... - 13 -
RISC-V OTTER Immediate Value Generation ..... - 14 -
Detailed RISC-V OTTER Assembly Instruction Description ..... - 15 -
`add` ..... - 16 -
`addi` ..... - 16 -
`and` ..... - 17 -
`andi` ..... - 17 -
`auipc` ..... - 18 -
`beq` ..... - 18 -
`beqz` ..... - 19 -
`bge` ..... - 19 -
`bgeu` ..... - 20 -
`bgez` ..... - 21 -
`bgt` ..... - 22 -
`bgtu` ..... - 23 -
`bgtz` ..... - 24 -
`ble` ..... - 25 -
`bleu` ..... - 26 -
`blez` ..... - 27 -
`blt` ..... - 28 -
`bltz` ..... - 29 -
`bltu` ..... - 30 -
`bne` ..... - 30 -
`bnez` ..... - 31 -
`call` ..... - 31 -
`csrr` ..... - 32 -
`csrrc` ..... - 32 -
`csrw` ..... - 33 -
`j` ..... - 33 -

jal ..... - 34 -
jalr ..... - 35 -
jr ..... - 35 -
la ..... - 36 -
lb ..... - 36 -
lbu ..... - 37 -
lh ..... - 37 -
lhu ..... - 38 -
li ..... - 38 -
lw ..... - 39 -
lui ..... - 39 -
mret ..... - 40 -
mv ..... - 40 -
neg ..... - 41 -
nop ..... - 41 -
not ..... - 42 -
or ..... - 42 -
ori ..... - 43 -
ret ..... - 43 -
sb ..... - 44 -
seqz ..... - 44 -
sgtz ..... - 45 -
sh ..... - 45 -
sw ..... - 46 -
sll ..... - 46 -
slli ..... - 47 -
slt ..... - 47 -
slti ..... - 48 -
sltiu ..... - 48 -
sltu ..... - 49 -
sltz ..... - 49 -
snez ..... - 50 -
sra ..... - 50 -
srai ..... - 51 -
srl ..... - 51 -
srli ..... - 52 -
sub ..... - 52 -
tail ..... - 53 -
xor ..... - 54 -
xori ..... - 54 -
RISC-V OTTER Assembly Language Style File ..... - 55 -

Acknowledgements

Transitioning to the RISC-V OTTER was initially the work of Joseph Callenes-Sloan. The RISC-V OTTER replaced the RAT MCU, which effectively modernized and removed many constraints from using the RAT MCU to teach a course in computer architecture and assembly language programming. Teaching any course for the first time requires a ton of work, but designing and implementing the course for the first time, which is what Joseph did, requires even more work. Bridget Benson was the first instructor outside of Joseph to use the RISC-V OTTER; Joseph's and Bridget's work has paved the way for other instructors using the RISC-V OTTER.

The RISC-V Assembler

The RISC-V OTTER Instruction Set

The RISC-V OTTER instructions are the RV32I instructions from the open RISC-V architecture. The RISC-V OTTER instruction set comprises of two types of instructions: base instructions and pseudoinstrutions. The base pseudoinstructions are special cases of the base instructions.

RISC-V OTTER Assembly Instructions Formats

The RISC-V OTTER instruction set has seven types of instruction formats. Table 1 shows each of these formats.

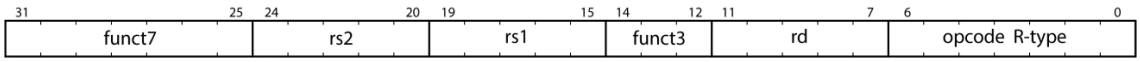
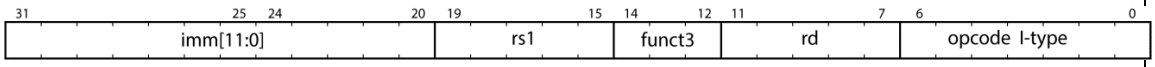
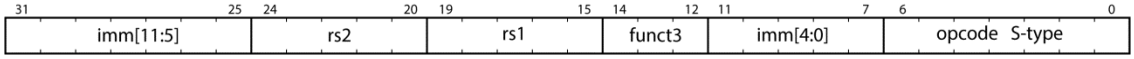
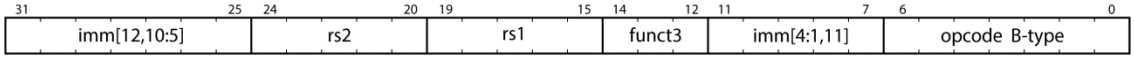
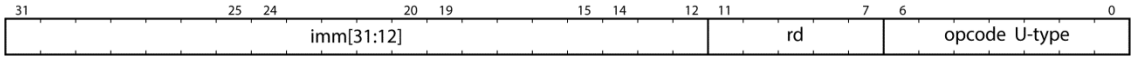
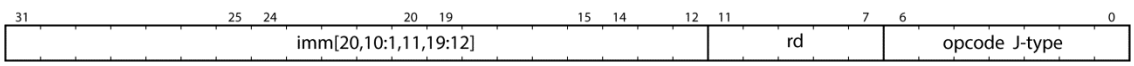
Instr Type	Instruction Format
R-type	
I-type	
S-type	
B-type	
U-type	
J-type	

Table 1: Instruction types and associated instruction formats.

Instruction Type: R-type

Figure 1 shows the R-type instruction format. Table 2 lists the instructions using the R-type format.

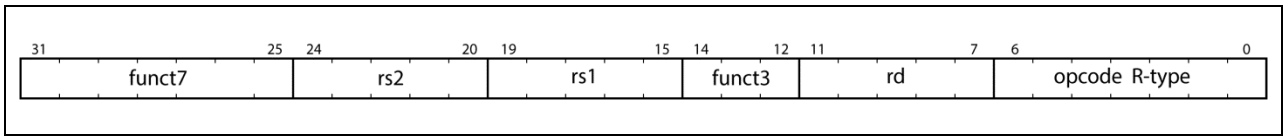


Figure 1: R-type instruction format.

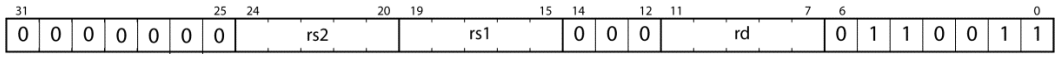
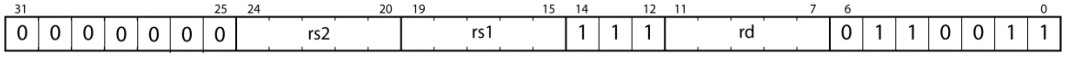
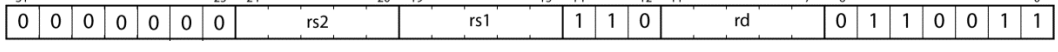
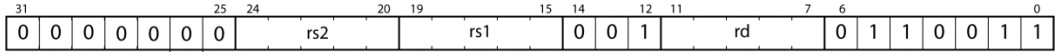
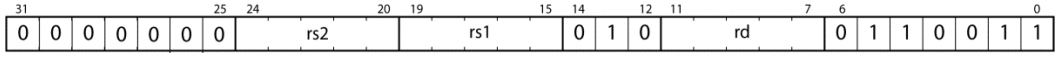
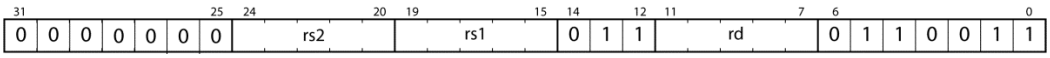
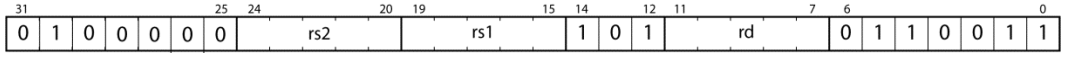
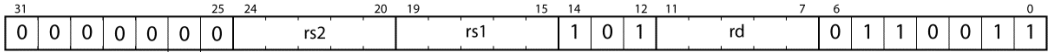
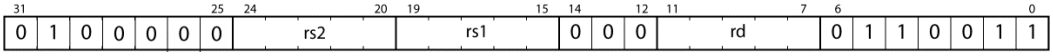
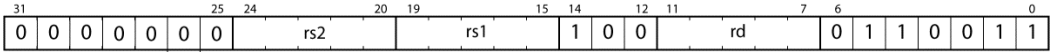
add `add rd,rs1,rs2`	
and `and rd,rs1,rs2`	
or `or rd,rs1,rs2`	
sll `sll rd,rs1,rs2`	
slt `slt rd,rs1,rs2`	
sltu `sltu rd,rs1,rs2`	
sra `sra rd,rs1,rs2`	
srl `srl rd,rs1,rs2`	
sub `sub rd,rs1,rs2`	
xor `xor rd,rs1,rs2`	

Table 2: R-type instructions with opcodes.

Instruction Type: I-type

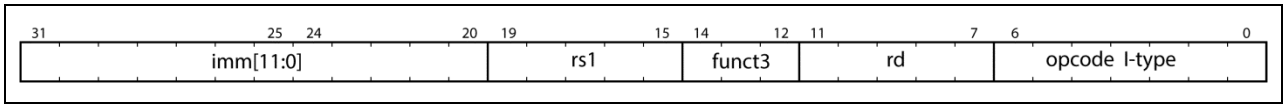


Figure 2: I-type instruction format.

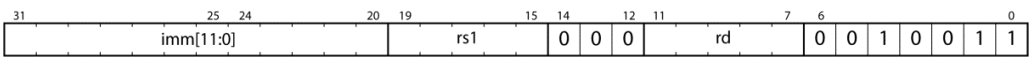
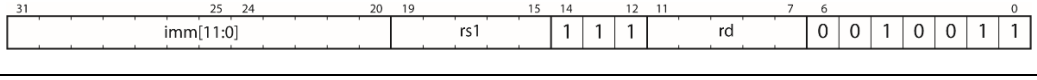
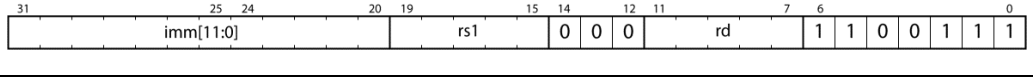
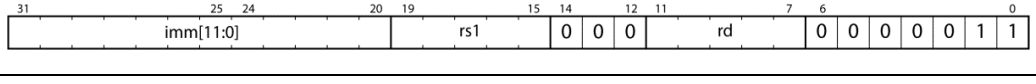
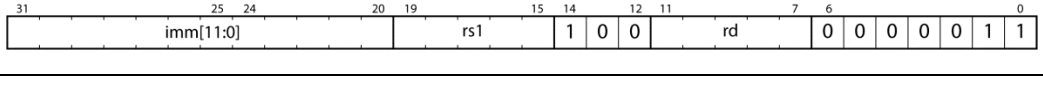
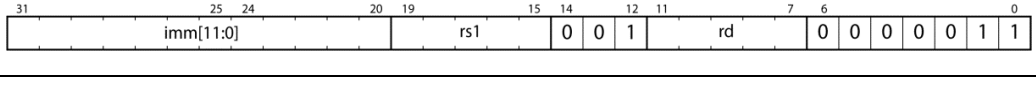
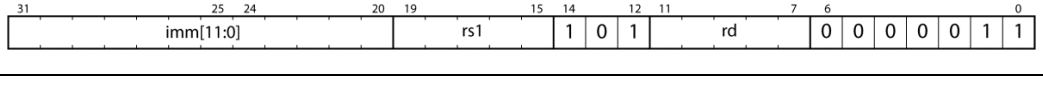
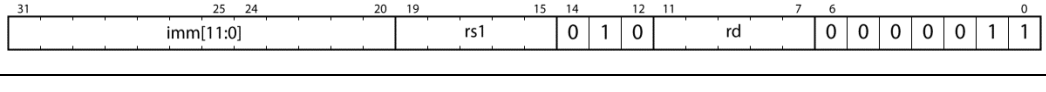
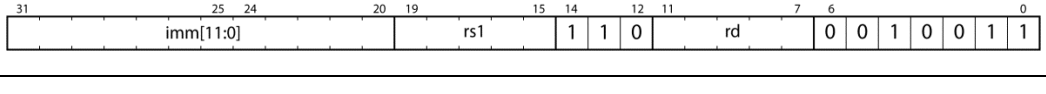
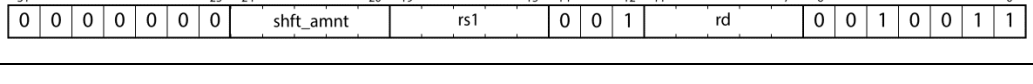
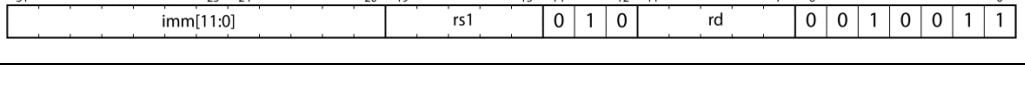
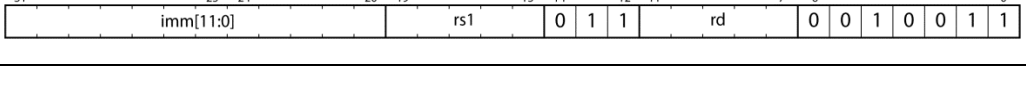
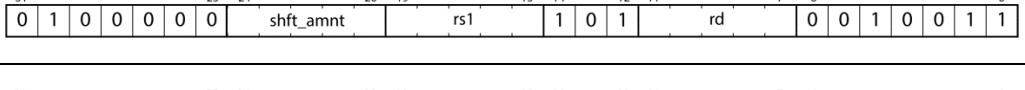
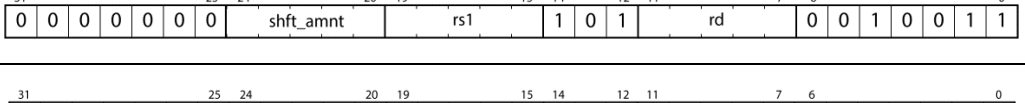
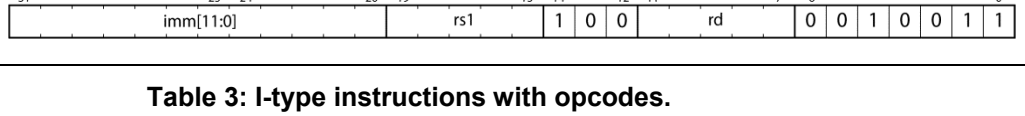
addi `addi rd,rs1,imm`	
andi `andi rd,rs1,imm`	
jalr `jalr rd,rs1,imm`	
lb `lb rd,imm(rs1)`	
lbu `lbu rd,imm(rs1)`	
lh `lh rd,imm(rs1)`	
lhu `lhu rd,imm(rs1)`	
lw `lw rd,imm(rs1)`	
ori `addi rd,rs1,imm`	
slli `addi rd,rs1,imm`	
slti `slti rd,rs1,imm`	
sltiu `sltiu rd,rs1,imm`	
srai `srai rd,rs1,imm`	
srli `srli rd,rs1,imm`	
xori `xori rd,rs1,imm`	

Table 3: I-type instructions with opcodes.

Instruction Type: S-type

Figure 3 shows the R-type instruction format. Table 4 lists the instructions using the R-type format.

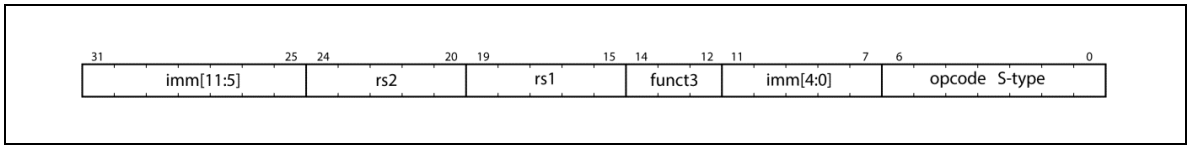


Figure 3: S-type instruction format.

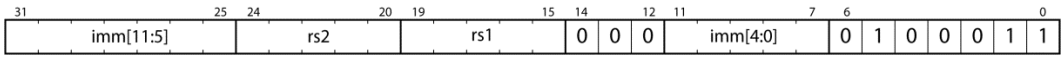
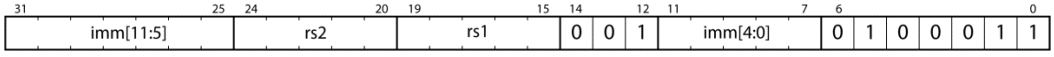
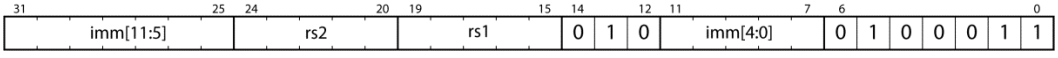
sb `sb rs2,imm(rs1)`	
sh `sh rs2,imm(rs1)`	
sw `sw rs2,imm(rs1)`	

Table 4: S-type instructions with opcodes.

Instruction Type: B-type

Figure 4 shows the R-type instruction format. Table 5 lists the instructions using the R-type format.

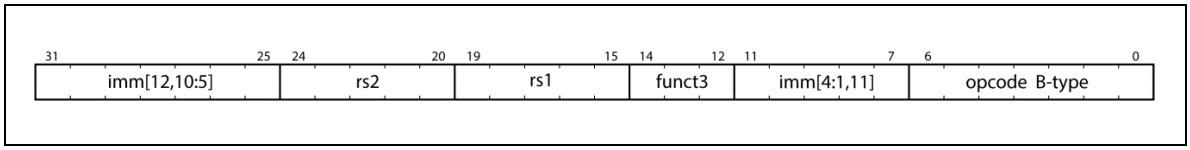


Figure 4: B-type instruction format.

beq `beq rs1,rs2,imm`	imm[12,10:5] | rs2 | rs1 | 0 0 0 | imm[4:1,11] | 1 1 0 0 0 1 1
bge `beg rs1,rs2,imm`	imm[12,10:5] | rs2 | rs1 | 1 0 1 | imm[4:1,11] | 1 1 0 0 0 1 1
bgeu `bgeu rs1,rs2,imm`	imm[12,10:5] | rs2 | rs1 | 1 1 1 | imm[4:1,11] | 1 1 0 0 0 1 1
blt `blt rs1,rs2,imm`	imm[12,10:5] | rs2 | rs1 | 1 0 | imm[4:1,11] | 1 1 0 0 0 1 1
bltu `bltu rs1,rs2,imm`	imm[12,10:5] | rs2 | rs1 | 1 1 0 | imm[4:1,11] | 1 1 0 0 0 1 1
bne `bne rs1,rs2,imm`	imm[12,10:5] | rs2 | rs1 | 0 0 1 | imm[4:1,11] | 1 1 0 0 0 1 1

Table 5: B-type instructions with opcodes.

Instruction Type: U-type

Figure 5 shows the R-type instruction format. Table 6 lists the instructions using the R-type format.

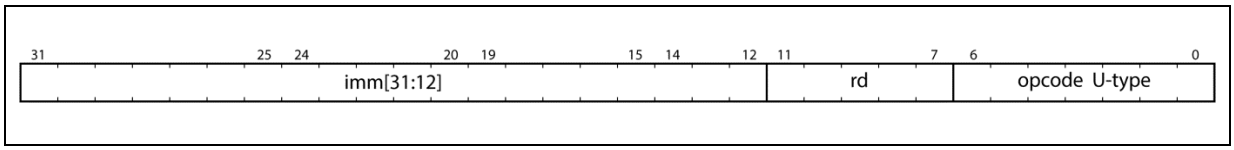


Figure 5: U-type instruction format.

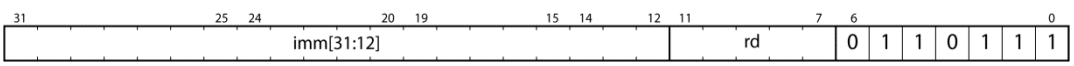
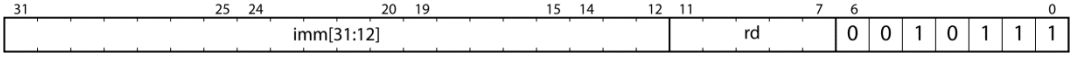
lui `lui rd,imm`	
auipc `auipc rs2,imm`	

Table 6: U-type instructions with opcodes.

Instruction Type: J-type

Figure 6 shows the R-type instruction format. Table 7 lists the instructions using the R-type format.

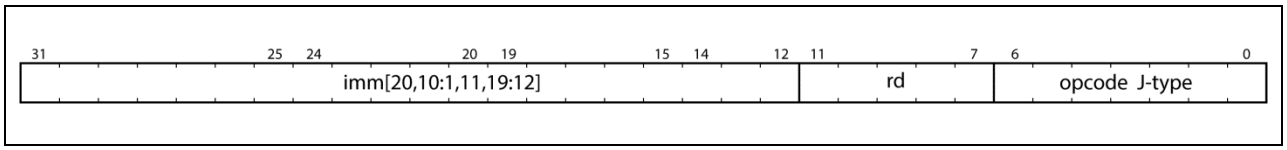


Figure 6: J-type instruction format.

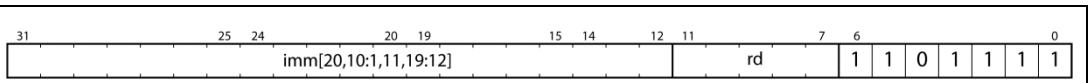
jal `jal rd,imm`	

Table 7: J-type instructions with opcodes.

RISC-V OTTER Assembly Instructions Brief Listing

Program Control					
jal	rd,imm	**j**	imm	**jal**	imm
jalr	rd,rs1,imm	**jr**	rs	**jalr**	rs
call	imm			**tail**	imm
ret		**mret**			
beq	rs1,rs2,imm	**beqz**	rs1,imm		
bne	rs1,rs2,imm	**bnez**	rs1,imm		
blt	rs1,rs2,imm	**blez**	rs1,imm	**bgt**	rs1,rs2,imm
bge	rs1,rs2,imm	**bgez**	rs1,imm	**ble**	rs1,rs2,imm
bltu	rs1,rs2,imm	**bltz**	rs1,imm	**bgtu**	rs1,rs2,imm
bgeu	rs1,rs2,imm	**bgtz**	rs1,imm	**bleu**	rs1,rs2,imm

Load/Store (& I/O)					
lb	rd,imm(rs1)			**sb**	rs2,imm(rs1)
lh	rd,imm(rs1)			**sh**	rs2,imm(rs1)
lw	rd,imm(rs1)	**lw**	rd,imm	**sw**	rs2,imm(rs1)
lbu	rd,imm(rs1)			**sw**	rd2,imm,rt
lhb	rd,imm(rs1)				

Operations					
addi	rd,rs1,imm	**add**	rd,rs1,rs2		
		sub	rd,rs1,rs2	**neg**	rd,rs1
xori	rd,rs1,imm	**xor**	rd,rs1,rs2	**not**	rd,rs1
ori	rd,rs1,imm	**or**	rd,rs1,rs2		
andi	rsd,rs1,imm	**add**	rd,rs1,rs2		
slli	rd,rs1,imm	**sll**	rd,rs1,rs2		
srli	rd,rs1,imm	**srl**	rd,rs1,rs2	**sgtz**	rd,rs1
srai	rd,rs1,imm	**sra**	rd,rs1,rs2	**sltz**	rd,rs1
slti	rd,rs1,imm	**slt**	rd,rs1,rs2	**snez**	rd,rs1
sltiu	rd,rs1,imm	**sltu**	rd,rs1,rs2	**seqz**	rd,rs1

Auxillary					
nop		**auipc**	rd,imm	**lui**	rd,imm
csrrw	rd,csr,rs1	**li**	rd,imm	**mv**	rd,rs
la	rd,imm				

Table 8: RISC-V OTTER Brief format instruction set listing.

RISC-V OTTER Assembly Instruction Overview

Table 9 lists RISC-V OTTER instructions including instruction format, description, and RTL description.

Instruction	Description	RTL	
add rd,rs1,rs2	addition	X[rd] ← X[rs1] + X[rs2]	
addi rd,rs1,imm	addition with immediate	X[rd] ← X[rs1] + sext(imm)	
and rd,rs1,rs2	bitwise AND	X[rd] ← X[rs1] · X[rs1]	
andi rd,rs1,imm	Bitwise AND immediate	X[rd] ← X[rs1] · sext(imm)	
auipc rd,imm	add upper immediate to PC	X[rd] ← PC + sext(imm)	
beq rs1,rs2,imm	branch if equal	PC ← PC + sext(imm) if (X[rs1] == X[rs2])	
beqz rs1,imm	branch if equal to zero	PC ← PC + sext(imm) if (X[rs1] == 0)	
bge rs1,rs2,imm	branch if greater than or equal	PC ← PC + sext(imm) if (X[rs1] $\geq_s$ X[rs2])	
bgeu rs1,rs2,imm	branch if greater than or equal unsigned	PC ← PC + sext(imm) if (X[rs1] $\geq_u$ X[rs2])	
bgez rs1,imm	branch if greater than or equal to zero	PC ← PC + sext(imm) if (X[rs1] $\geq_s$ 0)	
bgt rs1,rs2,imm	branch if greater than	PC ← PC + sext(imm) if (X[rs1] $>_s$ X[rs2])	
bgtu rs1,rs2,imm	branch if greater than unsigned	PC ← PC + sext(imm) if (X[rs1] $>_u$ X[rs2])	
bgtz rs1,rs2,imm	branch if greater than zero	PC ← PC + sext(imm) if (X[rs1] $>_s$ 0)	
ble rs1,rs2,imm	branch if less than or equal	PC ← PC + sext(imm) if (X[rs1] $\leq_s$ X[rs2])	
bleu rs1,rs2,imm	branch if less than or equal (unsigned)	PC ← PC + sext(imm) if (X[rs1] $\leq_u$ X[rs2])	
blez rs1,rs2,imm	branch if less than or equal zero	PC ← PC + sext(imm) if (X[rs1] $\leq_s$ 0)	
blt rs1,rs2,imm	branch if less than	PC ← PC + sext(imm) if (X[rs1] $<_s$ X[rs2])	
bltz rs1,imm	branch if less than zero	PC ← PC + sext(imm) if (X[rs1] $<_s$ 0)	
bltu rs1,rs2,imm	branch if less than (unsigned)	PC ← PC + sext(imm) if (X[rs1] $<_u$ X[rs2])	
bne rs1,rs2,imm	branch if not equal	PC ← PC + sext(imm) if (X[rs1] ≠ X[rs2])	
bnez rs1,imm	branch if not equal to zero	PC ← PC + sext(imm) if (X[rs1] ≠ 0)	
call rd,symbol **call symbol**	branch to subroutine	X[rd] ← PC + 8; PC ← &symbol	
csrr rd,csr	control & status register read	X[rd] ← CSRs[csr]	
csrrc rd,csr,rs1	control & status register read and clear	X[rd] ← CSRs[csr]; CSRs[csr] ← X[rd] & ~X[rs1]	
csrw csr,rs1	control & status register clear immedia	CSRs[csr] ← CSRs[csr] & ~immed	
j imm	unconditional branch	PC ← PC + sext(imm)	
jal rd,imm **jal imm**	unconditional branch with offset	X[rd] ← PC + 4; PC ← PC + sext(imm)	
jalr rd,imm(rs1) **jalr imm(rs1)**	unconditional branch with offset & link	X[rd] ← PC+4; PC ← (X[rs1] + sext(imm)) & ~1	
jr rs1	unconditional branch to register address	PC ← X[rs1]	
la rd,symbol	load absolute address of symbol	X[rd] ← &symbol	
lb rd,imm(rs1)	load byte	X[rd] ← sext(M[X[rs1] + sext(imm)] [7:0])	
lbu rd,imm(rs1)	load byte unsigned	X[rd] ← M[X[rs1] + sext(imm)] [7:0]	
lh rd,imm(rs1)	load halfword	X[rd] ← sext(M[X[rs1] + sext(imm)] [15:0])	
lhu rd,imm(rs1)	load halfword unsigned	X[rd] ← M[X[rs1] + sext(imm)] [15:0]	
li rd,imm	load immediate	X[rd] ← imm	
lw rd,imm(rs1)	load word into register	X[rd] ← M[X[rs1] + sext(imm)] [31:0]	
lui rd,imm	load upper immediate	X[rd] ← imm[31:12] << 12	
mret	machine mode exception return	X[rd] ← sext(imm[31:12] << 12)	
mv rd,rs1	move	X[rd] ← X[rs1]	
neg rd,rs2	negate	X[rd] ← -X[rs2]	
nop	no operation	nada (PC ← PC + 4)	
not rd,rs2	ones complement	X[rd] ← ~X[rs2]	
or rd,rs1,rs2	bitwise inclusive OR	X[rd] ← X[rs1] | X[rs2]	
or rd,rs1,imm	bitwise inclusive OR immediate	X[rd] ← X[rs1] | sext(imm)	
ret	return from subroutine	PC ← X1	
sb rs2,imm(rs1)	store byte in memory	M[X[rs1] + sext(imm)] ← X[rs2][7:0]	
seqz rd,rs1	set if equal to zero	X[rd] ← (X[rs1] == 0) ? 1 : 0	
sgtz rd,rs2	set if greater than zero	X[rd] ← (X[rs2] $>_s$ 0) ? 1 : 0	
sh rs2,imm(rs1)	store halfword in memory	M[X[rs1] + sext(imm)] ← X[rs2][15:0]	
sw rs2,imm(rs1)	store word	M[X[rs1] + sext(imm)] ← X[rs2]	
sll rd,rs1,rs2	logical shift left	X[rd] ← X[rs1] << X[rs2]	
slli rd,rs1,shft_amt	logical shift left immediate	X[rd] ← X[rs1] << shft_amt	
slt rd,rs1,rs2	set if less than	X[rd] ← (X[rs1] $<_s$ X[rs2]) ? 1 : 0	
slti rd,rs1,imm	set if less than immediate	X[rd] ← (X[rs1] $<_s$ sext(imm)) ? 1 : 0	
sltiu rd,rs1,imm	set if less than immediate unsigned	X[rd] ← (X[rs1] $<_u$ sext(imm)) ? 1 : 0	
sltu rd,rs1,rs2	set if less than unsigned	X[rd] ← (X[rs1] $<_u$ X[rs2]) ? 1 : 0	
sltz rd,rs1	set if less than zero	X[rd] ← (X[rs1] $<_s$ 0) ? 1 : 0	
snez rd,rs2	set if not equal to zero	X[rd] ← (X[rs2] ≠ 0) ? 1 : 0	
sra rd,rs1,rs2	arithmetic shift right	X[rd] ← X[rs1] $>>_s$ X[rs2]	
srai rd,rs1,shft_amt	arithmetic shift right immediate	X[rd] ← X[rs1] $>>_s$ shft_amt	
srl rd,rs1,rs2	logical shift right	X[rd] ← X[rs1] >> X[rs2]	
srli rd,rs1,imm	logical shift right immediate	X[rd] ← X[rs1] >> imm	
sub rd,rs1,rs2	subtract	X[rd] ← X[rs1] – X[rs2]	
tail symbol	tail call	PC ← &symbol; X[6] ← some_new_val	

Table 9: RISC-V OTTER Instructions with RTL description.

RISC-V OTTER Immediate Value Generation

Table 10 lists the immediate value format for the RISC-V OTTER.

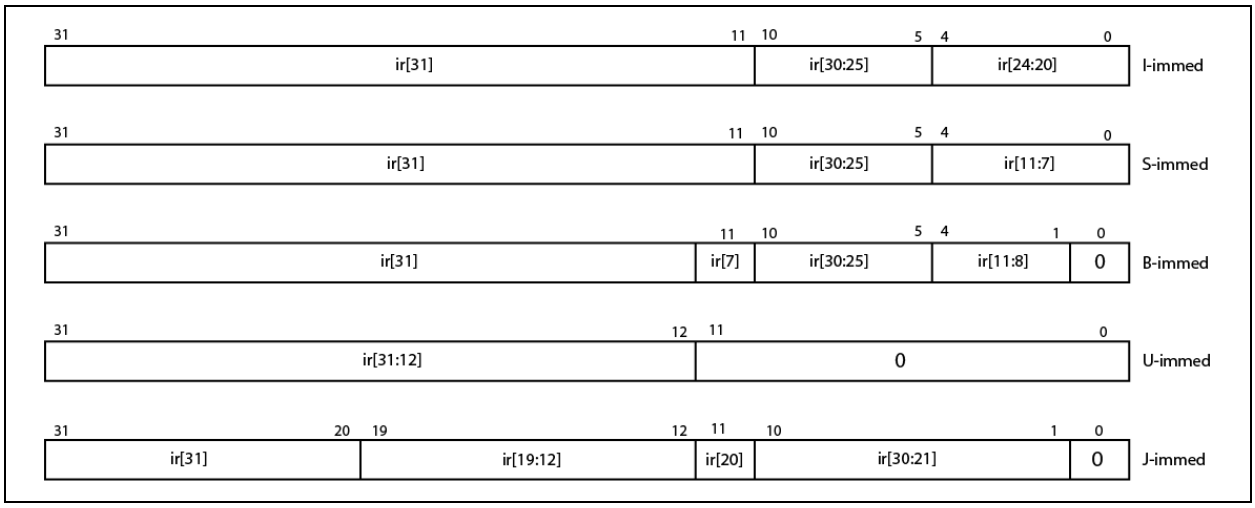


Table 10: RISC-V OTTER Immediate values based on instruction formats.

Detailed RISC-V OTTER Assembly Instruction Description

The following section lists each of the RISC-V instructions in a detailed format. The instruction details include the following:

- Instruction mnemonic for instructions and pseudoinstructions
- Short instruction description
- Associated RTL statement(s)
- Detailed instruction format (for ABI instructions only)
- Instruction usage example
- An ever-so-helpful “Also See” listing

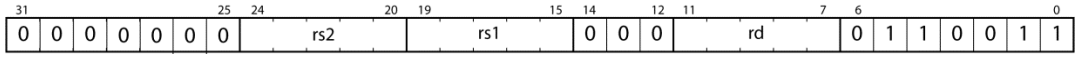
add	***addition***

RTL: *X[rd] ← X[rs1] + X[rs2]*

Forms: `add rd,rs1,rs2`

Description: The add instruction performs an addition operation on the two source operands rs1 & rs2 and stores the result in the destination operand rd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same register as the destination. Both source operands are treated as signed values in 2's complement format. The **`add`** instruction ignores any arithmetic overflow resulting from the operation.

Instruction Format (R-type)



Usage:

```
add   X10,X10,X12    # addition of values in registers X10 & X12;
                     # result stored in X10; X12 is not affected.
                     #  X10 = 0x0000_00A4  X12 = 0x0000_00C7    (before exec)
                     #  X10 = 0x0000_016B  X12 = 0x0000_00C7    (after exec)
```

See Also: **`addi, sub`**

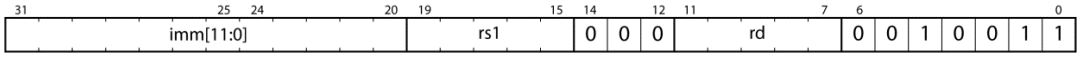
addi	***addition with immediate***

RTL: *X[rd] ← X[rs1] + sext(imm)*

Forms: `addi rd,rs1,imm`

Description: The add instruction performs an addition operation on the operand rs1 and the immediate value and stores the result in the destination operand rd. The instruction overwrites value in the destination operand; the source operand is not affected unless it specifies same register as the destination. The 12-bit immediate value is sign-extended before addition. Both source operands are treated as signed values in 2's complement format. The **`addi`** instruction ignores any arithmetic overflow resulting from the operation.

Instruction Format (I-type)



Usage:

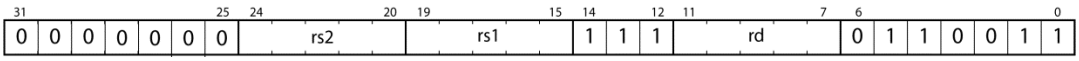
```
add   X10,X11,0x0DC  # addition of values in X10 to 0xDC
                     # result stored in X10; X12 is not affected.
                     #  X10 = 0x0000_0045  X11 = 0x0000_0024    (before exec)
                     #  X10 = 0x0000_0100  X11 = 0x0000_0024    (after exec)
```

See Also: **`add, sub`**

and	***bitwise AND***	
RTL: *X[rd] ← X[rs1] · X[rs1]*	**Forms:**	**and rd,rs1,rs2**

Description: The and instruction performs a bit-wise AND operation on the two source operands rs1 & rs2 and stores the result in the destination operand rd. The instruction overwrites value in the destination operand; source operands are not affected unless they specify same register as the destination.

Instruction Format (R-type)



Usage:

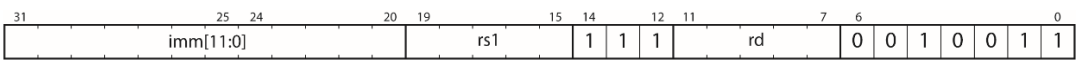
```
and    X1,X4,X5  # bitwise and of values in register X1 & X4;
                 # result is placed in X1; X4 & X5 values don't change
                 #  X1=0x0000_00A4; X4=0x0000_00C7  (before execution)
                 #  X5=0x0000_0084 (before execution)
                 #  X1=0x0000_0084  X4=0x0000_00C7 X5=0x0000_0084 (after exec)
```

See Also: andi, or, xor

andi	***bitwise AND***	
RTL: *X[rd] ← X[rs1] · sext(imm)*	**Forms:**	**andi rd,rs1,imm**

Description: The and instruction performs a bit-wise AND operation on the source operand rs1 & the immediate value and stores the result in the destination operand rd. The instruction overwrites value in the destination operand; the source operand is not affected unless it specifies same register as the destination operand. The immediate value is a 12-bit value that is sign-extended before AND operation.

Instruction Format (I-type)



Usage:

```
andi    X2,X4,0xFF0  # bitwise and of values X4 & 0x0F4;
                     # result is placed in X2; X4 value doesn't change
                     # immed value is signed extended to 0xFFFF_FFF0
                     #  X2=0x0000_00E2; X4=0x0000_00C9  (before exec)
                     #  X1=0x0000_00C0  X4=0x0000_00C9  (after exec)
```

See Also: and, ori, xori

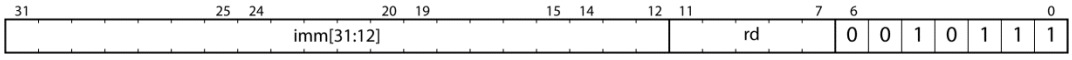
auipc	***add upper immediate to PC***

RTL: *X[rd] ← PC + sext(imm)*

Forms: `auipc rd,imm`

Description: The **auipc** instruction sums the current value of the program counter (PC) and the immediate value, shifts the result 12-bits to the left, and stores the result in the destination register rd. This instruction only modifies the destination register rd.

Instruction Format (U-type)



Usage:

```
    auipc    X10,0x1DFF2  # add the immediate value to the PC, left-shifts
                          # the result 12 bits, and stores result in X10
                          #
                          # PC=0x0000_007E X10=0x0000_FFFF  (before exec)
                          # PC=0x0000_007E X10=0x1DFF_207E  (after exec)
```

See Also: lui, jal, jalr

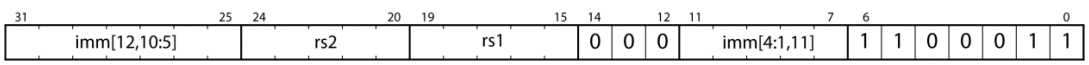
beq	***branch if equal***

RTL: *PC ← PC + sext(imm) if (X[rs1] == X[rs2])*

Forms: `beq rs1,rs2,imm`

Description: The beq instruction can cause the PC to be modified by adding a signed offset to it if the value in the two source registers are equal. If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value.

Instruction Format (B-type)



Usage:

```
       beq     X10,X11,Junk # branch to the instruction at the address
                            #  associated with the Junk label if the values in
                            #  X10 & X11 are equal
       nop                  #     X10=0x0000_FFFF X11=0x0000_FFFF  (before exec)
       nop                  #     PC=0x0F00_0000                   (before exec)
       nop                  # branch taken
Junk:  nop                  #     X10=0x0000_FFFF X11=0x0000_FFFF  (after exec)
                            #     PC=0x0F00_0010                   (after exec)
```

See Also: bne

beqz	***branch if equal to zero***	(pseudoinstruction: **beq**)
RTL: *PC ← PC + sext(imm) if (X[rs1] == 0)*	**Form:**	**beqz rs1,imm**

Description: The **beqz** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source register is zero. If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not the source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. **beqz** is a pseudoinstruction based on the **beq** instruction, and is equivalent to: "**beq rs1,X0,imm**".

Instruction Format

Usage:

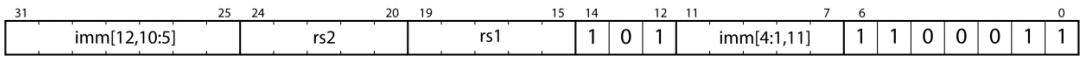
```
        beqz    X10,Oak     # branch to the instruction at the address
                            #  associated with the Oak label if the value in
        nop                    #  X10 equals 0
        nop                 #     X10=0x0000_0000  PC=0x00DF_0000   (before exec)
        nop                 #
        nop                 # branch taken
Oak:    nop                 #     X10=0x0000_0000  PC=0x00DF_00014  (after exec)
```

See Also: **beq**

bge	**branch if greater than or equal**
RTL: *PC ← PC + sext(imm) if (X[rs1] ≥$_s$ X[rs2])*	**Forms:** **bge rs1,rs2,imm**

Description: The **bge** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source registers rs1 is greater than or equal to the value in source register rs2 (both source operands are treated as signed numbers in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value.

Instruction Format (B-type)



Usage:

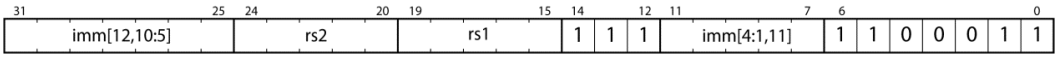
```
        bge     X10,X11,Dog  # branch to the instruction at the address
                             #  associated with the Dog label if the value in
                             #  X10 is greater than or equal to the value in X11
        nop                  #     X10=0x0000_FFFF  X11=0x8000_FFF0  (before exec)
        nop;                 #     PC=0x0F00_0C00                    (before exec)
        nop                  #
        nop                  # branch taken
Dog:    nop                  #     X10=0x0000_FFFF  X11=0x8000_FFF0  (after exec)
                             #     PC=0x0F00_0C14                    (after exec)
```

See Also: **blt**

bgeu	branch if greater than or equal unsigned		
RTL: *PC ← PC + sext(imm) if (X[rs1] ≥$_u$ X[rs2])*		**Forms:**	**bgeu rs1,rs2,imm**

Description: The **bgeu** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source registers rs1 is greater than or equal to the value in source register rs2 (both source operands are treated as unsigned numbers). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value.

Instruction Format (B-type)



Usage:

```
        bgeu    X10,X11,Dog  # branch to the instruction at the address
                             #  associated with the Dog label if the value in
                             #  X10 is greater than or equal to the value in & X11
        nop                  #     X10=0xC000_FFFF  X11=0x8000_FFF0  (before exec)
        nop;                 #     PC=0x0FE0_0500                    (before exec)
        nop                  #
        nop                  # branch taken
Dog:    nop                  #     X10=0xC000_FFFF  X11=0x8000_FFF0  (after exec)
                             #     PC=0x0FE0_0514                    (after exec)
```

See Also: **bgeu**

bgez	branch if greater than or equal to zero	(pseudoinstruction -- **bge**)
RTL: *PC ← PC + sext(imm) if (X[rs1] ≥$_s$ 0)*	**Form:**	**bgez rs1,imm**

Description: The **bgez** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source register is greater than or equal to zero (the source operand is treated as signed number in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. **bgez** is a pseudoinstruction based on the **bge** instruction and is equivalent to "**bge rs1,X0,imm**".

Instruction Format	

Usage:

```
        beqz    X10,Pine    # branch to the instruction at the address
                            #  associated with the Pine label if the values in
        nop                 #  X10 is greater than or equal to 0
        nop                 #     X10=0x0000_0010  PC=0x012F_0008   (before exec)
        nop                 #
        nop                 # branch taken
Pine:   nop                 #     X10=0x0000_0010  PC=0x012F_001C  (after exec)
```

See Also: **bge**

bgt	branch if greater than	(pseudoinstruction -- blt)
RTL: *PC ← PC + sext(imm) if (X[rs1] >$_s$ X[rs2])*	**Form:**	**bgt rs1,rs2,imm**

Description: The **bgt** instruction can cause the PC to be modified by adding a signed offset to it if the value in source register rs1 is greater than the value in source register rs2 (the source operands are treated as a signed numbers in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. The **bgt** instruction is a pseudoinstruction based on the **blt** instruction, and is equivalent to: "**blt rs2,rs1,offset**".

Instruction Format

Usage:

```
        bgt     X10,X11,Gum  # branch to the instruction at the address
                             #  associated with the Gum label if the value in X10
        nop                  #  is greater than the value in X11
        nop                  #     X10=0x2000_2003 X11=0x2000_0002  (before exec)
        nop                  #     PC=0x0E31_0004                   (before exec)
        nop                  # branch taken
Gum:    nop                  #     X10=0x2000_2003 X11=0x2000_0002  (after exec)
                             #     PC=0x0E31_0018                   (after exec)
```

See Also: **blt**

bgtu	branch if greater than (unsigned)	(pseudoinstruction -- bltu)
RTL: *PC ← PC + sext(imm) if (X[rs1] >$_u$ X[rs2])*	**Form:**	**bgtu rs1,rs2,imm**

Description: The **bgtu** instruction can cause the PC to be modified by adding a signed offset to it if the value in source register rs1 is greater than the value in source register rs2 (the source operands are treated as a unsigned numbers). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. The **bgtu** instruction is a pseudoinstruction based on the **bltu** instruction and is equivalent to the following: "**bltu rs2,rs1,imm**".

Instruction Format

Usage:

```
        bgtu    X10,X11,Red  # branch to the instruction at the address
                             #  associated with the Red label if the value in X10
        nop                  #  is greater than the value in X11
        nop                  #     X10=0xC000_0002 X11=0xB358_A332  (before exec)
        nop                  #     PC=0x0E31_0014                   (before exec)
        nop                  # branch taken
Red:    nop                  #     X10=0xC000_0002 X11=0xB358_A332  (after exec)
                             #     PC=0x0E31_0028                   (after exec)
```

See Also: **bltu**

bgtz	branch if greater than zero	(pseudoinstruction -- blt)
RTL: *PC ← PC + sext(imm) if (X[rs1] >$_s$ 0)*	**Form:**	**bgtz rs1,rs2,imm**

Description: The **bgtz** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source register is greater than zero (the source operand is treated as a signed number in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not the source register rs1. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. The **bgtz** instruction is a pseudoinstruction based on the **blt** instruction and is equivalent to "**blt X0,rs2,imm**".

Instruction Format

Usage:

```
        bgtz    X10,Hog     # branch to the instruction at the address
                            #  associated with the Hog label if the value in
        nop                 #  X10 is greater than 0
        nop                 #     X10=0x0000_0011  PC=0x0679_000C   (before exec)
        nop                 #
        nop                 # branch taken
Hog:    nop                 #     X10=0x0000_0011  PC=0x0679_0020   (after exec)
```

See Also: **blt, bgtu**

ble	branch if less than or equal	(pseudoinstruction -- bge)
RTL: *PC ← PC + sext(imm) if (X[rs1] ≤$_s$ X[rs2])*	**Form:**	**ble rs1,rs2,imm**

Description: The **ble** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source register rs1 is less than or equal to the value in source register rs2 (the source operands are treated as signed numbers in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. The **ble** instruction is a pseudoinstruction based on the **bge** instruction and is equivalent to "**bge rs2,rs1,imm**".

Instruction Format

Usage:

```
        ble     X10,X11,Hot  # branch to the instruction at the address
                             #  associated with the Hot label if the value in X10
        nop                  #  is less than or equal the value in X11
        nop                  #     X10=0xBEE1_0002 X11=0xBEE1_0002  (before exec)
        nop                  #     PC=0x0E31_001C                   (before exec)
        nop                  # branch taken
Hot:    nop                  #     X10=0xBEE1_0002 X11=0xBEE1_0002  (after exec)
                             #     PC=0x0E31_0030                   (after exec)
```

See Also:

bleu	branch if less than or equal (unsigned)	(pseudoinstruction -- bgeu)
RTL: *PC ← PC + sext(imm) if (X[rs1] ≤$_u$ X[rs2])*	**Form:**	**bleu rs1,rs2,imm**

Description: The **bleu** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source register rs1 is less than or equal to the value in source register rs2 (the source operands are treated as signed numbers in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. The **ble** instruction is a pseudoinstruction based on the **bgeu** instruction and is equivalent to "**bgeu rs2,rs1,imm**".

Instruction Format

Usage:

```
        bleu    X10,X11,Beg  # branch to the instruction at the address
                             #  associated with the Beg label if the value in X10
        nop                  #  is less than or equal the value in X11
        nop                  #     X10=0xFEE1_7439 X11=0xFEE1_743A  (before exec)
        nop                  #     PC=0x7E34_0044                   (before exec)
        nop                  # branch taken
Beg:    nop                  #     X10=0xFEE1_7439 X11=0xFEE1_743A  (after exec)
                             #     PC=0x7E34_0058                   (after exec)
```

See Also: **bgeu**

blez	branch if less than or equal zero	(pseudoinstruction -- bge)
RTL: *PC ← PC + sext(imm) if (X[rs1] ≤$_s$ 0)*	**Form:**	**blez rs1,rs2,imm**

Description: The **blez** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source register is less than or equal to zero (the source operands is treated as signed number in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not the source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. The blez pseudo instruction is equivalent to "**bge X0,rs2,imm**".

Instruction Format

Usage:

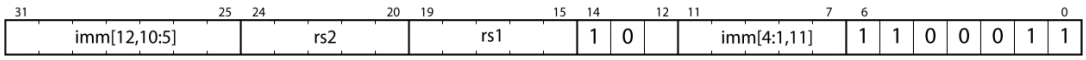
```
        blez    X10,Nom     # branch to the instruction at the address
                            #  associated with the Nom label if the value in
        nop                 #  X10 is less than or equal to 0
        nop                 #     X10=0xE000_0010  PC=0x0A34_103C   (before exec)
        nop                 #
        nop                 # branch taken
Nom:    nop                 #     X10=0xE000_0011  PC=0x0A34_0050   (after exec)
```

See Also: **bge**

blt	branch if less than		
RTL: *PC ← PC + sext(imm) if (X[rs1] <$_s$ X[rs2])*		**Form:**	**blt rs1,rs2,imm**

Description: The **blt** instruction can cause the PC to be modified by adding a signed offset to it if the value in source register rs1 is less than the value in source register rs2 (the source operands are treated as signed numbers in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value.

Instruction Format (B-type)



Usage:

```
        blt     X10,X11,Elm  # branch to the instruction at the address
                             #  associated with the Elm label if the value in X10
        nop                  #  is less than the value in X11
        nop                  #     X10=0xFFFF_EEE7 X11=0xFFFF_EEE8  (before exec)
        nop                  #     PC=0x0F21_0000                   (before exec)
        nop                  # branch taken
Elm:    nop                  #     X10=0xFFFF_EEE7 X11=0xFFFF_EEE8  (after exec)
                             #     PC=0x0F21_0014                   (after exec)
```

See Also: bgt

bltz	branch if less than zero	(pseudoinstruction -- blt)
RTL: *PC ← PC + sext(imm) if (X[rs1] <$_s$ 0)*	**Form:**	**bltz rs1,imm**

Description: The **bltz** instruction can cause the PC to be modified by adding a signed offset if the value in the source register is less than zero (the source operand is treated as a signed number in 2's complement format). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not the source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. The **bltz** instruction is a pseudoinstruction based on the **blt** instruction and is equivalent to "**blt rs1,X0,imm**".

Instruction Format

Usage:

```
        bltz    X10,Mug      # branch to the instruction at the address
                             #  associated with the Mug label if the value in X10
        nop                  #  is less than 0
        nop                  #     X10=0x8000_0001          (before exec)
        nop                  #     PC=0x0F21_000C           (before exec)
        nop                  # branch taken
Mug:    nop                  #     X10=0x8000_0001          (after exec)
                             #     PC=0x0F21_0020           (after exec)
```

See Also: **blt**

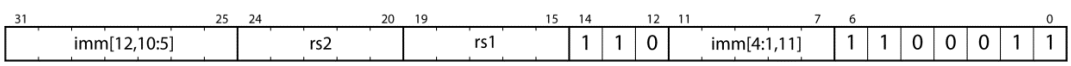
bltu	branch if less than (unsigned)

RTL: *PC ← PC + sext(imm) if (X[rs1] <$_u$ X[rs2])*

Form: `bltu rs1,rs2,imm`

Description: The **bltu** instruction can cause the PC to be modified by adding a signed offset if the value in source register rs1 is less than the value in source register rs2 (the source operands are treated as unsigned numbers). If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value.

Instruction Format (B-type)



Usage:

```
        bltu    X10,X11,Pig  # branch to the instruction at the address
                             #  associated with the Pig label if the value in
                             #  X10 is less than the value in X11
        nop                  #     X10=0x8000_FFFF  X11=0xE000_FFF0  (before exec)
        nop                  #     PC=0x0FE3_0700                    (before exec)
        nop                  #
        nop                  # branch taken
Pig:    nop                  #     X10=0xE000_FFFF  X11=0x8000_FFF0  (after exec)
                             #     PC=0x0FE3_0714                    (after exec)
```

See Also:

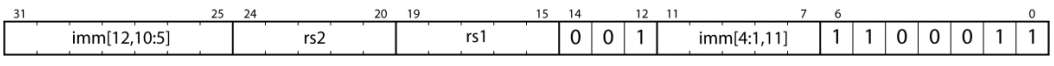
bne	branch if not equal

RTL: *PC ← PC + sext(imm) if (X[rs1] ≠ X[rs2])*

Form: `bne rs1,rs2,imm`

Description: The **bne** instruction can cause the PC to be modified by adding a signed offset to it if the value in the two source registers are not equal. If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not either source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value.

Instruction Format (B-type)



Usage:

```
        bne     X20,X21,Bob  # branch to the instruction at the address
                             #  associated with the Junk label if the value in
                             #  X20 is not equal to the value in X21
        nop                  #     X20=0x0000_FFFF  X21=0x8000_FFFF  (before exec)
        nop                  #     PC=0x0FE3_2800                    (before exec)
        nop                  #
        nop                  # branch taken
Bob:    nop                  #     X20=0x0000_FFFF  X21=0x8000_FFFF  (after exec)
                             #     PC=0x0FE3_2814                    (after exec)
```

See Also: **beq**

bnez	branch if not equal to zero	(pseudoinstruction -- **bne**)
RTL: *PC ← PC + sext(imm) if (X[rs1] ≠ 0)*	**Form:**	**bnez rs1,imm**

Description: The **bnez** instruction can cause the PC to be modified by adding a signed offset to it if the value in the source register is not equal to zero. If the condition evaluates as true, the branch is taken (the address of the next instruction to executed is loaded into the PC); otherwise, the PC advances normally. This instruction can only change the PC and not the source register. The branch address is formed by sign-extending the 12-bit immediate value to 32-bits before being added to the PC. This is a PC-relative branch; the resulting PC value can increase or decrease based on the sign of the immediate value. **bnez** is a pseudoinstruction based on the **bne** instruction and is equivalent to: "**bne rs1,X0,imm**".

Instruction Format

Usage:

```
        bnez    X20,Who     # branch to the instruction at the address
                            #  associated with the Junk label if the value in
                            #  X20 is not equal to 0
        nop                 #     X20=0x0000_FF3F        (before exec)
        nop                 #     PC=0x0AA3_3900         (before exec)
        nop                 #
        nop                 # branch taken
Who:    nop                 #     X20=0x0000_FF3F  X11=0x8000_FFFF  (after exec)
                            #     PC=0x0AA3_3914                    (after exec)
```

See Also: **bne**

call	branch to subroutine	(pseudoinstruction – auipc, jalr)
RTL: *X[rd] ← PC + 8; PC ← &symbol*	**Forms:**	**call rd,symbol** **call symbol**

Description: The **call** instruction is a pseudoinstruction used to transfer program control to another location in program memory. The **call** instruction causes the assembler the assembler to issue two ABI instructions: **auipc** & **jalr**; these two instructions formulate a 32-bit value that is loaded into the PC (thus forming an absolute address). The destination register rd is overwritten with the return value, which is the address value of the instruction two instruction slots after the **call** instruction. The **call** instruction uses X1 as the destination register if a register is not included as an operand in the **call** instruction.

Instruction Format

Usage:

```
        call    X1,Sue      # branch to the instruction at the address
                            #  associated with the Sue; store return address in X1
        nop                 # X1=0x0044_2220   PC=0x0FD3_1494    (before exec)
        nop                 #
        nop                 #
Sue:    nop                 # X1=0x0FD3_149C   PC=0x0FD3_14A4    (after exec)
```

See Also: **auipc, jalr, jal**

csrr	control & status register read	(pseudoinstruction -- csrrs)
RTL: *X[rd] ← CSRs[csr]*	**Form:**	`csrr rd,csr`
Description:		
Instruction Format		
Usage:	`;--------` `wewe`	
See Also:		

csrrc	control & status register read and clear
RTL: *X[rd] ← CSRs[csr]; CSRs[csr] ← X[rd] & ~X[rs1]*	**Form:** `csrrc rd,csr,rs1`
Description:	
Instruction Format (B-type)	
Usage:	`;--------` `wewe`
See Also:	

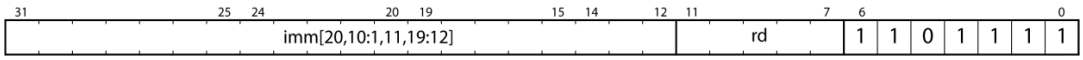
csrw	control & status register clear immed	(pseudoinstruction -- csrrw)
RTL: *CSRs[csr] ← CSRs[csr] & ~immed*	**Form:**	**csrw csr,rs1**
Description:		
Instruction Format		
Usage:	;-------- wewe	
See Also:		

j	unconditional branch	(pseudoinstruction -- jal)
RTL: *PC ← PC + sext(imm)*	**Form:**	**j imm**
Description: The j is a pseudoinstruction based on the **jal** instruction. The **j** instruction is an unconditional branch instruction that modifies the PC by adding the current PC value to a sign-extended version of the immediate value, which transfers program execution to the address of an instruction that is not the "next" instruction. This **j** instruction is equivalent to "**jal X0,imm**".		
Instruction Format		
Usage:	j -12 # unconditional branch to the instruction at the address # adjusted by the immediate value # PC=0x001F_0500 (before exec) # PC=0x001F_04FC (after exec)	
See Also: **jal, jalr, jr**		

jal	unconditional branch with offset		
RTL: *X[rd] ← PC + 4; PC ← PC + sext(imm)*		**Form:**	**jal rd,imm** **jal imm**

Description: The **jal** instruction is an unconditional branch instruction that modifies the PC by adding an immediate value to it, which transfers program execution to the address of an instruction that is not the "next" instruction. The jal instruction writes the address of the instruction after jal to the destination rd. The instruction then sign extends the 20-bit immediate value, adds it to the current PC, and then loads the result into the PC. The **jal** instruction is a PC-relative unconditional branch; the resulting PC value can increase or decrease based on the sign of the immediate value. If the destination operand rd is omitted from the **jal** instruction, the assembler will use X1 as the destination register.

Instruction Format (J-type)



Usage:

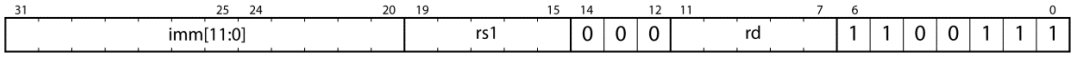
```
        jal     X8,Emu        # branch to the instruction at the address
                              #  associated with the Emu label; place address of
                              #  next instruction in PC
        nop                   #     X8=0xE000_FFFF             (before exec)
        nop                   #     PC=0x00EF_0500             (before exec)
        nop                   #
Emu:    nop                   #     X8=0x00EF_0504            (after exec)
                              #     PC=0x00EF_0510             (after exec)
```

See Also: **jalr, j, call, ret**

jalr	unconditional branch with offset & link		
RTL: *X[rd] ← PC+4; PC ← (X[rs1] + sext(imm)) & ~1*		**Form:**	**jalr rd,imm(rs1)** **jalr imm(rs1)**

Description: The jalr instruction is an unconditional branch instruction that modifies the PC by overwriting it with a summation of the source register value and an immediate value, which transfers program execution to the address of an instruction that is not the "next" instruction. The jalr instruction writes the address of the instruction after **jalr** to the destination rd. The instruction sign extends the 20-bit immediate value, multiplies it by two, and then clears the LSB before adding it to the value in the source register; the resulting value is loaded into the PC, which ensures instruction access from program memory must happen on halfword boundaries. If the destination operand rd is omitted, the jalr instruction assumes the destination operand to be X1. When **jalr** is used to transfer program control from subroutines back to calling code, the destination register is assigned but not used.

Instruction Format (I-type)



Usage:

```
jalr    X4,4(X1)    # jump to address specified in X1 register
                    #     X1=0x0045_FF00  X4=0x0034_0034  (before exec)
                    #     PC=0x00E4_0520                  (before exec)

                    #     X1=0x0045_FF00  X4=0x00E4_0524  (after exec)
                    #     PC=0x0022_FF82                  (after exec)
```

See Also: jal, j, jr

jr	unconditional branch to register address	(pseudoinstruction -- jalr)
RTL: *PC ← X[rs1]*	**Form:**	**jr rs1**

Description: The **jr** is a pseudoinstruction based on the jalr instruction. The jr instruction is an unconditional branch instruction that modifies the PC by overwriting it with the value in the source register, which transfers program execution to the address of an instruction that is not the "next" instruction in program memory. This jr instruction is equivalent to "**jalr X0,0(rs1)**".

Instruction Format

Usage:

```
jr    X1          # jump to address specified in X1 register
                  #     X1=0x001A_FB00                  (before exec)
                  #     PC=0x00E3_7500                  (before exec)

                  #     X1=0x001A_FB00                  (after exec)
                  #     PC=0x001A_FB00                  (after exec)
```

See Also: jalr, jal

la	load absolute address of symbol	(pseudoinstruction – `auipc` & `addi`)
RTL: *X[rd] ← &symbol*	**Form:**	**`la rd,symbol`**

Description: The **`la`** is a pseudoinstruction which causes the assembler to issue two ABI instructions: **`auipc`** & **`addi`**. The **`auipc`** instructions loads the upper 20 bits of the address associated with the label into the destination register rd (the 12 LSBs are zeroed); the **`addi`** instruction loads the 12 lower bits of the label by adding the immediate value of the **`addi`** instruction to the destination register rd, which contains the upper 20-bits set by the **`auipc`** instruction . The assembler takes care of the lower-level address formatting details.

Instruction Format

Usage:

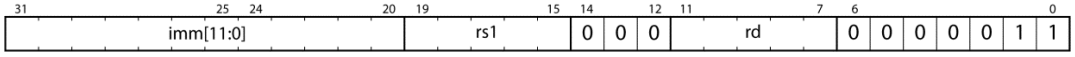
```
        la    x10,Ear:    # load the address associated with the Ear label into
                          #  the destination registerruction at the address
        nop               # X10=0x0044_2330          (before exec)
        nop               # X10=0x00D3_1494          (after exec)
                          #
Ear:    nop               # Instr Addr associated with Ear = 0x00D3_1494
```

See Also: **`lw, sw`**

lb	load byte
RTL: *X[rd] ← sext(M[X[rs1] + sext(imm)] [7:0])*	**Form:** **`lb rd,imm(rs1)`**

Description: The **`lb`** is a memory access instruction that loads a byte from memory into a specified register. The **`lb`** instruction forms the address of the data to be loaded from memory by sign-extending the 12-bit immediate value and adding it to the value specified in source register rs1. The single byte read from memory is sign-extended before being loaded into the destination register rd.

Instruction Format (I-type)



Usage:

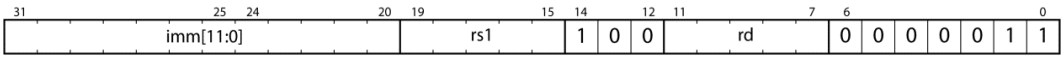
```
      lb   X10,-8(X20)   # load byte from memory at address specified by the
                         #  immediate value and source register (X20)
                         #     X20=0x0010_FB09                  (before exec)
                         #     X10=0x00E2_7500                  (before exec)
                         #     M[0x0010_FB01]=0xF3              (before exec)

                         #     X20=0x0010_FB20                  (after exec)
                         #     X10=0xFFFF_FFF3                  (after exec)
                         #     M[0x0010_FB01]=0xF3              (after exec)
```

See Also: **`lhw, lw`**

lbu	load byte unsigned		
RTL: *X[rd] ← M[X[rs1] + sext(imm)] [7:0]*		**Form:**	**lbu rd,imm(rs1)**

Description: The **lbu** is a memory access instruction that loads a byte from memory into a specified register. The **lbu** instruction forms the address of the data to be loaded from memory by sign-extending the 12-bit immediate value and adding it to the value specified in source register rs1. The single byte read from memory is zero-extended before being loaded into the destination register rd.

Instruction Format (I-type)



Usage:

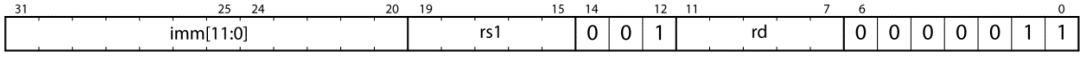
```
lbu   X10,-8(X20)  # load byte from memory at address specified by the
                   #  immediate value and source register (X20)
                   #     X20=0x0010_FB09                  (before exec)
                   #     X10=0x00E2_7500                  (before exec)
                   #     M[0x0010_FB01]=0xF3              (before exec)

                   #     X20=0x0010_FB09                  (after exec)
                   #     X10=0x0000_00F3                  (after exec)
                   #     M[0x0010_FB01]=0xF3              (after exec)
```

See Also: **lb, lh, lhu, lw**

lh	load halfword		
RTL: *X[rd] ← sext(M[X[rs1] + sext(imm)] [15:0])*		**Form:**	**lh rd,imm(rs1)**

Description: The **lh** is a memory access instruction that loads a halfword (two bytes) from memory into a specified register. The **lh** instruction forms the address of the data to be loaded from memory by sign-extending the 12-bit immediate value and adding it to the value specified in source register rs1. The halfword read from memory is sign-extended before being loaded into the destination register rd.

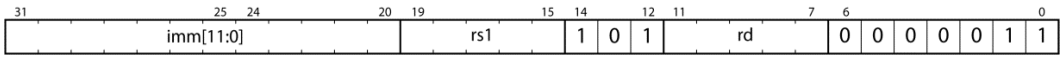
Instruction Format (I-type)



Usage:

```
lh    X10,4(X12)    # load halfword from memory at address specified by the
                    #  immediate value and source register (X20)
                    #     X12=0x021F_FB05                  (before exec)
                    #     X10=0x0DE2_75AA                  (before exec)
                    #     M[0x021F_FB09]=0xDEAD            (before exec)

                    #     X12=0x021F_FB05                  (after exec)
                    #     X10=0xFFFF_DEAD                  (after exec)
                    #     M[0x0010_FB09]=0xDEAD            (after exec)
```

See Also: **lhu, lb, lw**

lhu	load halfword unsigned		
RTL: *X[rd] ← M[X[rs1] + sext(imm)] [15:0]*		**Form:**	**lhu rd,imm(rs1)**
Description: The **lhu** is a memory access instruction that loads a halfword (two bytes) from memory into a specified register. The **lhu** instruction forms the address of the data to be loaded from memory by sign-extending the 12-bit immediate value and adding it to the value specified in source register rs1. The halfword read from memory is zero-extended before being loaded into the destination register rd.			
Instruction Format (I-type)			

Usage:	`lhu X10,4(X12) # load halfword from memory at address specified by the`
	`# immediate value and source register (X12)`
	`# X12=0x021F_FB00 (before exec)`
	`# X10=0x0DE2_75AA (before exec)`
	`# M[0x021F_FB04]=0xDEAD (before exec)`
	`# X12=0x021F_FB00 (after exec)`
	`# X10=0x0000_DEAD (after exec)`
	`# M[0x0010_FB04]=0xDEAD (after exec)`		

See Also: **lh, lb, lhu, lw**			

li	load immediate	(pseudoinstruction – addi)
RTL: *X[rd] ← imm*	**Form:**	**li rd,imm**
Description: The **li** instruction writes an immediate value to the destination register rd. This is an pseudoinstruction based on several possible ABI instruction including **addi** and is equivalent to the following instruction: "**addi rd,X0,imm**".		
Instruction Format		

Usage:	`li X9,1023 # write an immediate value into destination register X9`
	`# X9=0x021F_3B8A (before exec)`
	`# X9=0x0000_03FF (before exec)`	

See Also: **addi, lui**		

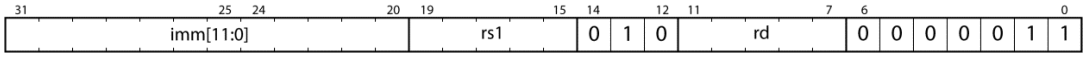
lw	load word into register

RTL: *X[rd] ← M[X[rs1] + sext(imm)] [31:0]*

Forms: `lw rd,imm(rs1)`

Description: The **lw** is a memory access instruction that loads a word (four bytes) from memory into the specified destination register rd. The **lw** instruction forms the address of the data to be loaded from memory by sign-extending the 12-bit immediate value and adding it to the value specified in source register rs1. The resulting data is copied from the specified memory location to the destination register rd.

Instruction Format (I-type)



Usage:

```
lw    X10,8(X13)   # load halfword from memory at address specified by the
                   #  immediate value and source register (X13)
                   #     X13=0x021F_FB0C                  (before exec)
                   #     X10=0x0DE2_F5AB                  (before exec)
                   #     M[0x021F_FB14]=0xDEAD_BEEF       (before exec)

                   #     X13=0x021F_FB04                  (after exec)
                   #     X10=0xDEAD_BEEF                  (after exec)
                   #     M[0x021F_FB14]=0xDEAD_BEEF       (after exec)
```

See Also: **lb, lbu, lh, lhu**

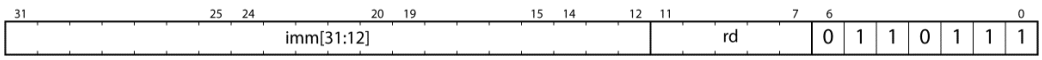
lui	load upper immediate

RTL: *X[rd] ← imm[31:12] << 12*

Form: `lui rd,imm`

Description: The **lui** instruction loads an immediate value into the destination rd. The immediate value is right-shifted 12 bit locations and the lower 12-bits are cleared before loading the destination register is loaded.

Instruction Format (U-type)



Usage:

```
lui    X10,258     # load immediate value to X10
                   #     X10=0x021F_3B0D                  (before exec)
                   #     X10=0x0010_2000                  (after exec)
```

See Also: **auipc**

mret	machine mode exception return	
RTL: *X[rd] ← sext(imm[31:12] << 12)*	**Form:**	**mret**
Description:		
Instruction Format (R-type)		
Usage:	;-------- wewe	
See Also:		

mv	move	(pseudoinstruction – addi)
RTL: *X[rd] ← X[rs1]*	**Form:**	**mv rd,rs1**
Description: The **mv** is a pseudoinstruction based on the addi instruction. The **mv** instruction copies the contents of the source register rs1 into the destination register rd. The contents of the source register does not change. The **mv** instruction is equivalent to the following instruction: "**addi rd,rs1,0**".		
Instruction Format		
Usage:	mv X10,X11 # copy the contents of source register X11 into # destination register X10 # X10=0x021F_3B0D X11=0345_668A (before exec) # X10=0x0345_668A X11=0345_668A (after exec)	
See Also: **addi**		

neg	negate	(pseudoinstruction – sub)
RTL: *X[rd] ← -X[rs2]*	**Form:**	**neg rd,rs2**

Description: The **neg** is a pseudoinstruction that performs a 2's complement on the context of the source register rs2 and places the result in the destination register rd. The **neg** instruction is equivalent to the following instruction: "**sub rd,X0,rs2**".

Instruction Format	

Usage:

```
neg    X12,X13   # perform 2's complement on the value in the source
                 #  register X13 and copy result to destination register X12
                 #
                 #  X12=0x021F_3B00  X13=0000_0001   (before exec)
                 #  X12=0xFFFF_FFFF  X13=0000_0001   (after exec)
```

See Also: **sub, not**

nop	no operation	(pseudoinstruction – addi)
RTL: *nada (PC ← PC + 4)*	**Form:**	**nop**

Description: The nop is a pseudoinstruction that effectively does nothing other than advancing the PC by four (PC = *PC + 4).* The **nop** instruction is equivalent to the following instruction: "**addi X0,X0,0**".

Instruction Format	

Usage:

```
nop          # do nothing (be like an academic administrator)
             #  PC=0x0A1E_3B00         (before exec)
             #  PC=0x0A1E_3B04         (after exec)
```

See Also: **addi**

not	ones complement	(pseudoinstruction – xori)
RTL: *X[rd] ← ~X[rs2]*	**Form:**	**not rd,rs2**

Description: The **not** is a pseudoinstruction that performs a 1's complement (toggles all bits) on the content of the source register rs2 and places the result in the destination register rd. The **not** instruction is equivalent to the following instruction: "**xori rd,rs1,-1**".

Instruction Format

Usage:

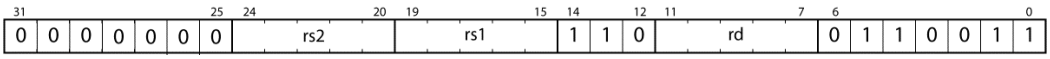
```
not    X14,X15   # perform 1's complement on the value in the source
                 #  register X15 and copies result to destination register X14
                 #
                 #  X14=0x021F_3B00  X15=0x5555_5555   (before exec)
                 #  X14=0xAAAA_AAAA  X15=0x5555_5555   (after exec)
```

See Also: **xori, neg**

or	bitwise inclusive OR	
RTL: *X[rd] ← X[rs1] | X[rs2]*	**Form:**	**or rd,rs1,rs2**

Description: The or instruction performs a bitwise inclusive OR between the values in the two source registers rs1 and rs2 and stores the result in the destination register rd. The or instruction does not change either value in the source registers unless a source register is also a destination register.

Instruction Format (R-type)



Usage:

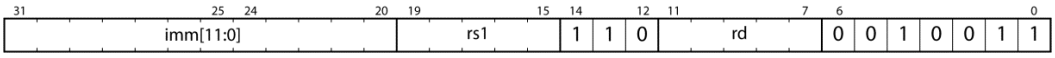
```
or   X14,X15,X16  # perform an inclusive OR in the values in source registers
                  #  X15 & X16 and stores result in destination register X14
                  #
               # X14=0x9832_AD34 X15=0x0034_3B00 X16=0xFFFF_00FF  (before exec)
               # X14=0xFFFF_3BFF X15=0x0034_3B00 X16=0xFFFF_00FF  (after exec)
```

See Also: **ori, and, xor**

ori	bitwise inclusive OR immediate	
RTL: *X[rd] ← X[rs1] | sext(imm)*	**Form:**	**or rd,rs1,imm**

Description: The **ori** instruction performs a bitwise inclusive OR between the value in the source register rs1 and the immediate value. The immediate value is signed extended from 12-bits to 32 bits before being ORed with the value in the source register. The result of the OR operation is stored in the destination register rd. The **ori** instruction does not change the value in the source register unless it is also specified as the destination register.

Instruction Format (I-type)



Usage:

```
ori   X14,X15,255  # perform an inclusive OR in the values in source registers
                   #  X15 & X16 and stores result in destination register X14
                   #
                   # X14=0x0232_AD34 X15=0x0EEE_3B11     (before exec)
                   # X14=0x0EEE_3BFF X15=0x0EEE_3B11     (after exec)
```

See Also: **or, andi, xori**

ret	return from subroutine	(pseudoinstruction -- jalr)
RTL: *PC ← X[1]*	**Form:**	**ret**

Description: The **ret** is a pseudoinstruction that is used to transfer program control form the end of a subroutine back the calling code (from the callee back to the caller). The **ret** instruction only works when the return address has been stored in register X1 (ra), which by convention is considered the return address register. The **ret** instruction is equivalent to the following instruction: "**jalr X0,0(X1)**".

Instruction Format

Usage:

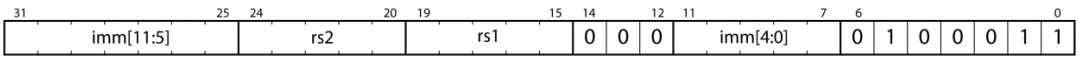
```
        ret      # return from subroutine
                 #  X1=0x0236_FE30  PC=0x0323_3434    (before exec)
                 #  X1=0x0236_FE30  PC=0x0236_FE30    (after exec)
```

See Also: **jalr**

sb	Store byte in memory		
RTL: *M[X[rs1] + sext(imm)] ← X[rs2][7:0]*		**Form:**	**sb rs2,imm(rs1)**

Description: The **sb** instruction stores a byte specified by the least significant byte in source register rs2 in memory at the address formed by sign-extending the 12-bit immediate value and adding it to the contents of the source register rs1. The byte that is stored is the eight LSBs of the source register rs2. The **sb** instruction does not change the contents of the source registers rs1 and rs2.

Instruction Format (S-type)



Usage:

```
sb    X14,8(X15)  # store the 8 LSBs of register X14 at the
                  #  address specified by X15 and the offset
                  #  X14=0x0000_4591 X15=0x0000_2345  (before exec)
                  #  M[0x0000_234D]=0x11              (before exec)
                  #
                  #  X14=0x0000_4591 X15=0x0000_2345  (after exec)
                  #  M[0x0000_234D]=0x91              (after exec)
```

See Also: **sh, sw**

seqz	Set if equal to zero	(pseudoinstruction -- sltiu)
RTL: *X[rd] ← (X[rs1] == 0) ? 1 : 0*	**Form:**	**seqz rd,rs1**

Description: The **seqz** instruction compares the value in the source register rs1 to 0; if the value in rs1 equals 0, the destination register rd is set to 1; otherwise the destination register is set to 0. The **seqz** instruction treats the source operand as a signed numbers in two's complement format. The **seqz** instruction does not change the source operand. The **seqz** instruction is a pseudoinstruction based on the **sltiu** instruction and is equivalent to: "**sltiu rd,rs1,1**"

Instruction Format

Usage:

```
seqz   X10,X12   # if the value in X12 equals 0, the 1 is written to X10;
                 # otherwise 0 is written to X10.
                 #
                 # X10=0x1812_DD74 X12=0x1000_0001   (before exec)
                 # X10=0x0000_0001 X12=0x1000_0001   (after exec)
```

See Also: **slt, sltiu**

sgtz	Set if greater than zero	(pseudoinstruction -- **slt**)
RTL: *X[rd] ← (X[rs2] >$_s$ 0) ? 1 : 0*	**Form:**	**sgtz rd,rs2**

Description: The **seqz** instruction compares the value in the source register rs1 to 0; if the value in rs1 equals 0, the destination register rd is set to 1; otherwise the destination register is set to 0. The **seqz** instruction treats the source operand as a signed numbers in two's complement format. The **seqz** instruction does not change the source operand. The **seqz** instruction is a pseudoinstruction based on the **sltiu** instruction and is equivalent to: "**slt rd,X0,rs2**"

Instruction Format

Usage:

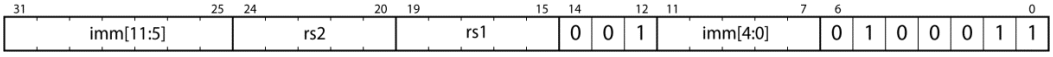
```
sgtz   X10,X12   # if the value in X12 is greater than 0, then 1
                 #  is written to X10; otherwise 0 is written to X10.
                 #
                 # X10=0x1812_DD74 X12=0x8000_0001   (before exec)
                 # X10=0x0000_0000 X12=0x8000_0001   (after exec)
```

See Also: **slt, sltiu**

sh	store halfword in memory	
RTL: *M[X[rs1] + sext(imm)] ← X[rs2][15:0]*	**Form:**	**sh rs2,imm(rs1)**

Description: The **sh** instruction stores a halfword (two bytes) specified by the least significant two bytes in source register rs2 in memory at the address formed by sign-extending the 12-bit immediate value and adding it to the contents of the source register rs1. The halfword that is stored is the 16 LSBs of the source register rs2. The **sh** instruction does not change the contents of the rs1 or rs2 source registers.

Instruction Format (S-type)



Usage:

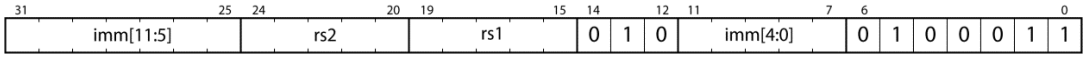
```
sh    X10,2(X11)  # store the 16 LSBs of register X10 at the
                  #  address specified by X11 and the offset
                  #  X10=0x0011_7591 X11=0x0000_F34A  (before exec)
                  #  M[0x0000_F34C]=0x1133            (before exec)
                  #
                  #  X10=0x0011_7591 X11=0x0000_F34A  (after exec)
                  #  M[0x0000_F34C]=0x7591            (after exec)
```

See Also: **sb, sw**

sw	store word		
RTL: *M[X[rs1] + sext(imm)] ← X[rs2]*		**Form:**	**sw rs2,imm(rs1)**

Description: The **sw** instruction stores a halfword (two bytes) in memory at the address formed by sign-extending the 12-bit immediate value and adding it to the contents of the source register rs1. The word that is stored is the value in the source register rs2. The **sw** instruction does not change the contents of the rs1 or rs2 source registers.

Instruction Format (S-type)



Usage:

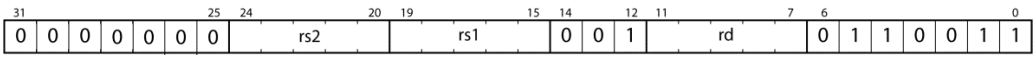
```
      sw    X10,4(X11)  # store the value in register X10 at the
                        #  address specified by X11 and the offset
                        #  X10=0x2211_7591 X11=0x0000_F348  (before exec)
                        #  M[0x0000_F34C]=0x3411_FACE       (before exec)
                        #
                        #  X10=0x2211_7591 X11=0x0000_F348  (after exec)
                        #  M[0x0000_F34C]=0x2211_7591       (after exec)
```

See Also: **sb, sh, shw**

sll	logical shift left		
RTL: *X[rd] ← X[rs1] << X[rs2]*		**Form:**	**sll rd,rs1,rs2**

Description: The **sll** instruction left shifts the data in the source register rs1 by the number of times specified in source register rs2, replaces the vacated bits by zero, then stores the result in the destination register rd. The **sll** instruction does not alter the values in either source register. The **slli** instruction only considers the five LSBs of the rs2 source register.

Instruction Format (R-type)



Usage:

```
    sll   X20,X22,X23  # left shift the value in X22 by the amount specified
                       #  by the lower five bits in X23; store result in X20
                       #
                  # X20=0x1812_AD34 X22=0x0001_3B00 X23=0xE000_10C8  (before exec)
                  # X20=0x0001_3B00 X22=0x0001_3B00 X23=0xE000_10C8  (after exec)
```

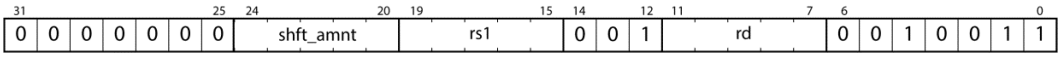
See Also:

slli	logical shift left immediate

RTL: *X[rd] ← X[rs1] << shft_amnt*	**Form:**	**slli rd,rs1,shft_amnt**

Description: The **slli** instruction left shifts the data in the source register rs1 by the number of times specified by the 5-bit shft_amnt value, replaces vacated bits by zero, then stores the result in the destination register rd. The **slli** instruction does not alter the value in the source register rs2.

Instruction Format (I-type)



Usage:

```
    slli   X15,X18,16  # left shift the value in X18 by the amount specified
                       #  by the immediate value (shft_amnt); store result in X15
                       #
                       # X15=0x0F13_AD34 X18=0x237F_3C11   (before exec)
                       # X15=0x3C11_0000 X18=0x237F_3C11   (after exec)
```

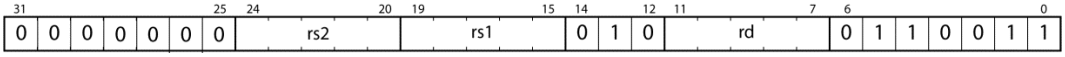
See Also: **sll, srl, srli**

slt	Set if less than

RTL: *X[rd] ← (X[rs1] <$_s$ X[rs2]) ? 1 : 0*	**Form:**	**slt rd,rs1,rs2**

Description: The **slt** instruction compares the values in the two source registers; if the value in rs1 is less than the value in rs2, the destination register rd is set to 1; otherwise the destination register is set to 0. The **slt** instruction treats both source operands as signed numbers in two's complement format. The **slt** instruction does not change either source operand.

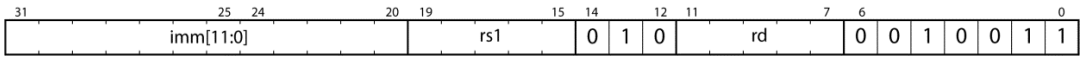
Instruction Format (R-type)



Usage:

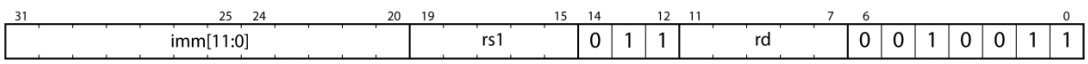
```
    slt   X10,X12,X13  # if the value in X12 is less than the value in 13,
                       #  1 is written to X10; otherwise 0 is loaded to X10
                       #
                  # X10=0x1812_AD74 X12=0xFFFF_FFFE X13=0xFFFF_FFFF  (before exec)
                  # X10=0x0000_0001 X12=0xFFFF_FFFE X13=0xFFFF_FFFF  (after exec)
```

See Also: **sltiu**

slti	Set if less than immediate		
RTL: *X[rd] ← (X[rs1] <$_s$ sext(imm)) ? 1 : 0*		**Form:**	**slti rd,rs1,imm**
Description: The **slti** instruction compares the value of the source register rs1 with the immediate value; if the value in the source register is less than the immediate value, the destination register rd is loaded to 1; otherwise the destination register is loaded with 0. The **slti** sign-extends the 12-bit immediate value before the comparison; both source operands are interpreted as signed values in two's complement format. The **slti** instruction does not change the source operand.			
Instruction Format (I-type)			

Usage:	`slti X10,X13,-12 # if the value in X13 is less than the immediate value (-12),`
	`# 1 is loaded to X10; otherwise 0 is loaded to X10`
	`#`
	`# X10=0x1845_AD74 X13=0xFFFF_FFF5 (before exec)`
	`# X10=0x0000_0000 X13=0xFFFF_FFF5 (after exec)`		

See Also:			

sltiu	Set if less than immediate unsigned		
RTL: *X[rd] ← (X[rs1] <$_u$ sext(imm)) ? 1 : 0*		**Form:**	**sltiu rd,rs1,imm**
Description: The **sltiu** instruction compares the value of the source register rs1 with the immediate value; if the value in the source register is less than the immediate value, the destination register rd is loaded to 1; otherwise the destination register is loaded with 0. The **sltiu** zero-extends the 12-bit immediate value before the comparison; the **sltiu** instruction interprets both source operands as unsigned values. The **sltiu** instruction does not change the source operand.			
Instruction Format (I-type)			

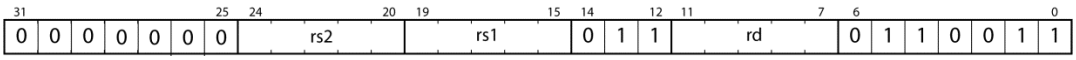
Usage:	`sltiu X10,X14,192 # if the value in X14 is less than the immediate value (192),`
	`# 1 is loaded to X10; otherwise 0 is loaded to X10`
	`#`
	`# X10=0x0F45_AD74 X14=0x0000_00BF (before exec)`
	`# X10=0x0000_0001 X14=0x0000_00BF (after exec)`		

See Also: **sltu, slt, slti**			

sltu	Set if less than unsigned		
RTL: *X[rd] ← (X[rs1] <$_u$ X[rs2]) ? 1 : 0*		**Form:**	**sltu rd,rs1,rs2**

Description: The **sltu** instruction compares the values in the two source registers; it the value in rs1 is less than the value in rs2, the destination register rd is set to 1; otherwise the destination register is set to 0. The **sltu** instruction interprets both source operands as unsigned values. The **sltu** instruction does not change either source operand.

Instruction Format (R-type)



Usage:

```
    sltu   X10,X12,X13  # if the value in X12 is less than the value in X13,
                        #  1 is loaded to X10; otherwise 0 is loaded to X10
                        #
                  # X10=0x1812_AD74 X12=0xFFFF_FFFF X13=0xFFFF_FFFE  (before exec)
                  # X10=0x0000_0000 X12=0xFFFF_FFFF X13=0xFFFF_FFFE  (after exec)
```

See Also:

sltz	Set if less than zero		(pseudoinstruction -- slt)
RTL: *X[rd] ← (X[rs1] <$_s$ 0) ? 1 : 0*		**Form:**	**sltz rd,rs1**

Description: The **sltz** pseudoinstruction writes a 1 to the destination register rd if the value in the source register is less than zero; otherwise a 0 is written to the destination register rd. The source operand is treated as a signed binary number in two's complement format. The **sltz** pseudo instruction is equivalent to "**slt rd,rs1,X0**" The **sltz** pseudoinstruction does not change the source operand.

Instruction Format

Usage:

```
     sltz   X11,X15   # if the value in X15 is less than 0,
                      #  1 is loaded to X11; otherwise 0 is loaded to X11
                      #
                      # X11=0x1845_ED74  X15=0x8000_0002   (before exec)
                      # X11=0x0000_0001  X15=0x8000_0002   (after exec)
```

See Also: **snez, slt, sltu**

snez	Set if not equal to zero	(pseudoinstruction -- sltu)
RTL: *X[rd] ← (X[rs2] ≠ 0) ? 1 : 0*	**Form:**	**snez rd,rs2**

Description: The **snez** pseudoinstruction writes a 1 to the destination register rd if the value in the source register is not equal to zero; otherwise a 0 is written to the destination register rd. The **snez** pseudoinstruction works with both signed and unsigned values. The **snez** pseudo instruction is equivalent to "**sltu rd,X0,rs2**" The **snez** pseudoinstruction does not change the source operand.

Instruction Format

Usage:

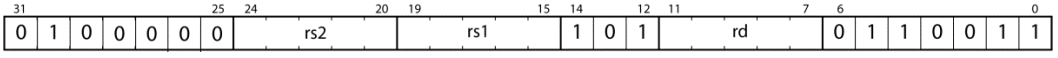
```
snez   X12,X20   # if the value in X20 is not equal to 0, then
                 #  1 is loaded to X12; otherwise 0 is loaded to X12
                 #
                 # X11=0x1845_ED74  X20=0x0000_0000   (before exec)
                 # X11=0x0000_0000  X20=0x0000_0000   (after exec)
```

See Also: **sltz, sltu**

sra	Arithmetic shift right	
RTL: *X[rd] ← X[rs1] >>$_s$ X[rs2]*	**Form:**	**sra rd,rs1,rs2**

Description: The **sra** instruction right shifts the data in source register rs1 by the number of times specified in source register rs2, replaces the vacated bits by a copy of the left-most bit in rs1 (considered to be the sign bit), then stores the result in the destination register rd. The **sra** instruction does not alter the values in either source register. The **sra** instruction only considers the five LSBs of the rs2 source register.

Instruction Format (R-type)



Usage:

```
sra   X10,X12,X13  # arithmetic right shift X12 value the amount specified
                   #  by the five LSBs in X13; store result in X10
                   #
              # X10=0x1812_AD34 X12=0xC001_1B00 X13=0xE034_10C8  (before exec)
              # X10=0xFFC0_011B X12=0xC001_1B00 X13=0xE034_10C8  (after exec)
```

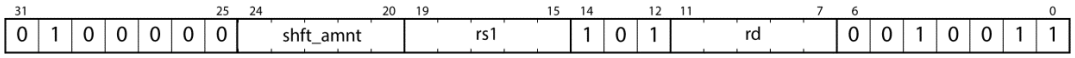
See Also:

srai	arithmetic shift right immediate

RTL: *X[rd] ← X[rs1] >>$_s$ shft_amnt*	**Form:**	**srai rd,rs1,shft_amnt**

Description: The **srai** instruction right shifts the data in source register rs1 by the number of times specified by the shft_amnt field in the srai instruction, replaces the vacated bits by a copy of the left-most bit (considered to be the sign bit) in rs1, then stores the result in the destination register rd. The **srai** instruction does not alter the values in the source register.

Instruction Format (I-type)



Usage:

```
    srai   X10,X15,16  # arithmetic right shift X15 value the amount specified
                       #  by the immediate value (shft_amnt); store result in X10
                       #
                       # X10=0x0D12_1D02  X15=0xE301_3C14   (before exec)
                       # X10=0xFFFF_E301  X15=0xE301_3C14   (after exec)
```

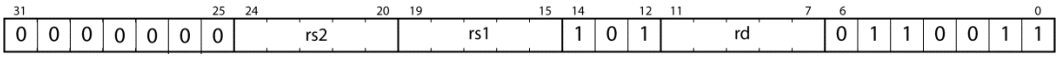
See Also:

srl	Logical shift right

RTL: *X[rd] ← X[rs1] >> X[rs2]*	**Form:**	**srl rd,rs1,rs2**

Description: The **srl** instruction right shifts the data in source register rs1 by the number of times specified in source register rs2, replaces the vacated bits by zero, then stores the result in the destination register rd. The **srl** instruction does not alter the values in either source register. The **srl** instruction only considers the five LSBs of the rs2 source register.

Instruction Format (R-type)



Usage:

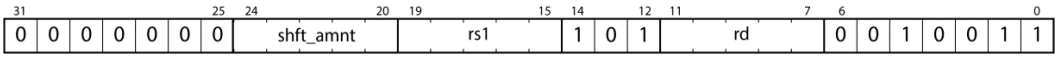
```
   srl   X10,X20,X21  # right the value in X20 by the amount specified
                      #  by the lower five bits in X21; store result in X10
                      #
                 # X10=0x1812_AD34 X20=0x0EDF_1B00 X21=0xE034_10CD  (before exec)
                 # X10=0x0000_EDF1 X20=0x0EDF_1B00 X21=0xE034_10CD  (after exec)
```

See Also:

srli	Logical shift right immediate		
RTL: *X[rd] ← X[rs1] >> imm*		**Form:**	**srli rd,rs1,imm**

Description: The **srli** instruction left shifts the data in the source register rs1 by the number of times specified by the 5-bit shft_amnt value, replaces vacated bits by zero, then stores the result in the destination register rd. The **srli** instruction does not alter the value in the source register rs2.

Instruction Format (I-type)



Usage:

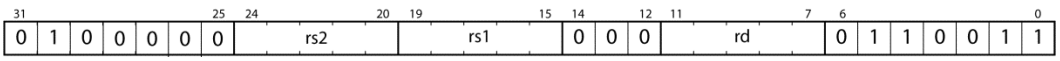
```
srli   X14,X15,20  # right shift the value in X14 by the amount specified
                   #  by the immediate value (shft_amnt); store result in X14
                   #
                   # X14=0x0F12_AD22 X15=0xC300_3C14   (before exec)
                   # X14=0x0000_0C30 X15=0xC300_3C14   (after exec)
```

See Also:

sub	subtract		
RTL: *X[rd] ← X[rs1] – X[rs2]*		**Form:**	**sub rd,rs1,rs2**

Description: The **sub** instruction subtracts the value in source register rs2 from the value in source register rs1, then stores the result in the destination register rd. Both source operands are treated as signed values in 2's complement format. The **sub** instruction does not alter the values in the either source register unless they are also the destination register. The **sub** instruction ignores any arithmetic overflow from the operation.

Instruction Format (R-type)



Usage:

```
sub   X10,X15,X16  #subtract value in X16 from value in X15;
                   # store result in X10
                   #
               # X10=0x1812_AD34 X15=0x0001_1B70 X16=0xFFFF_FFFF  (before exec)
               # X10=0x0001_1B71 X16=0x0001_1B70 X16=0xE034_10CD  (after exec)
```

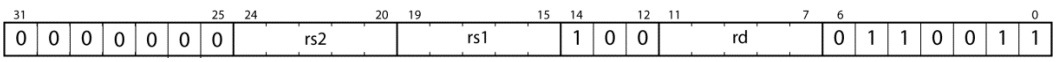
See Also:

tail	tail call		
RTL: *PC ← &symbol; X[6] ← some_new_val*		**Form:**	**tail symbol**
Description:			
Instruction Format (R-type)			
Usage:	blah blah blah		
See Also:			

xor	bitwise exclusive OR		
RTL: *X[rd] ← X[rd1] ^ X[rd2]*		**Form:**	**xor rd,rs1,rs2**

Description: The **xor** instruction performs a bitwise exclusive OR between the values in the two source registers rs1 & rs2 and then stores the result in the destination register rd. The **xor** instruction does not change either value in the source registers unless a source register is also a destination register.

Instruction Format (R-type)



Usage:

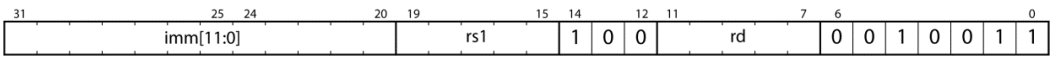
```
or   X14,X15,X16  # perform an inclusive OR in the values in source registers
                  #  X15 & X16 and stores result in destination register X14
                  #
               # X14=0x9832 AD34 X15=0x0000 3B00 X16=0xFFFF 0000  (before exec)
               # X14=0x67CD_3B00 X15=0x0000_3B00 X16=0xFFFF_0000  (after exec)
```

See Also: **or, and, xori**

xori	bitwise exclusive OR		
RTL: *X[rd] ← X[rd1] ^ sext(imm)*		**Form:**	**xor rd,rs1,imm**

Description: The **xori** instruction performs a bitwise inclusive OR between the value in the source register rs1 and the immediate value. The immediate value is signed extended from 12-bits to 32 bits before the being exclusive ORed with the value in the source register. The result of the XOR operation is stored in the destination register rd. The **xori** instruction does not change the value in the source register unless it is also specified as the destination register.

Instruction Format (I-type)



Usage:

```
xori   X14,X15,0xFFF # perform an inclusive XOR in the values in source registers
                     #  X15 & X16 and stores result in destination register X14
                     #
                     # X14=0x0232_AED4 X15=0x1111_3EEE     (before exec)
                     # X14=0xEEEE_C111 X15=0x1111_3EEE     (after exec)
```

See Also:

RISC-V OTTER Assembly Language Style File

Figure 7 shows an example assembly language program highlighting respectable RAT assembly language source code appearance.

```
;- Programmer: Pat Wankaholic
;- Date: 09-29-10
;-
;- This program does something really cool. Here's the description...
;-----------------------------------------------------------------------
             RET
```

Figure 7: Example RISC-V OTTER assembly language code showing required coding style.