The RISC-V MCU Assembly Language Manual

Table of Contents

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.

This document is a result of the focus and dedication of Cal Poly instructors; Cal Poly had no part in the creation of this document.

The RISC-V Assembler

This section is intentionally blank.

The RISC-V OTTER Registers

The RISC-V OTTER has 32, 32-bit registers (x0-31). To enable the assembly code to become more portable and reusable, a common usage is defined for each of the 32 registers. This effort is aided by giving each register an alternate name that assemblers can also understand. This not only makes the code more portable, but also more readable. Table 1 below shows each register along with its corresponding alternate name and usage designation. Table 2 shows the same table but referenced via the alternative register names. Table 3 shows the preferred register usage associated with passing data to and from subroutines.

Register Name	Alternate Name	Usage Description
x0	zero	Hardwired to zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate link register
x6-7	t1-2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10-11	a0-1	Function arguments / return values
x12-17	a2-7	Function arguments
x18-27	s2-11	Saved register
x28-31	t3-6	Temporaries

Table 1: RISC-V register names and common usage designation.

Alternate Name	Register Name	Usage Description
zero	x0	Hardwired to zero
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-6	x5, x6-7, x28-31	Temporaries
s0-11	x8-9, x18-27	Saved register
a0-7	x10-17	Function arguments

Table 2: RISC-V registers grouped by usage.

Register type	Calling Code	Subroutine (SR)		Comments
		Can SR change register values?	Should SR save register values?	
Argument (a0-a7)	SR may change regs so calling code must save regs before call if calling code relies on these reg values not changing.	Yes	Not necessary; SR can change when passing data to calling code.	These regs change based on how the calling code and SR pass and/or return data
Saved (s0-s11)	Calling code relies on SR not permanently changing these regs	No	Must save & restore if SR changes reg values	Save to stack or to unused Temporary registers
Temporary (t0-t6)	SR may change regs; calling code must save regs if calling code relies on these values not changing.	Yes	No	Working regs for subroutine

Table 3: Table of Preferred RISC-V Register Usage

The RISC-V OTTER Memory Map

The RISC-V OTTER has a 32-bit address space and can address 4GiB (2^{32} bytes) of data. However, the hardware is limited to 64kb of memory for program code, data, and stack. The RISC-V OTTER is implemented as a Von Neumann architecture, which just means that all the memory shares the same address space. This architecture simplifies the hardware design and allows the programmer to have flexibility of how to best optimize the usage of memory. To give a starting framework that should be adequate for all of programming tasks in this course, the memory in the RISC-V OTTER will be divided as shown below in Figure 1.

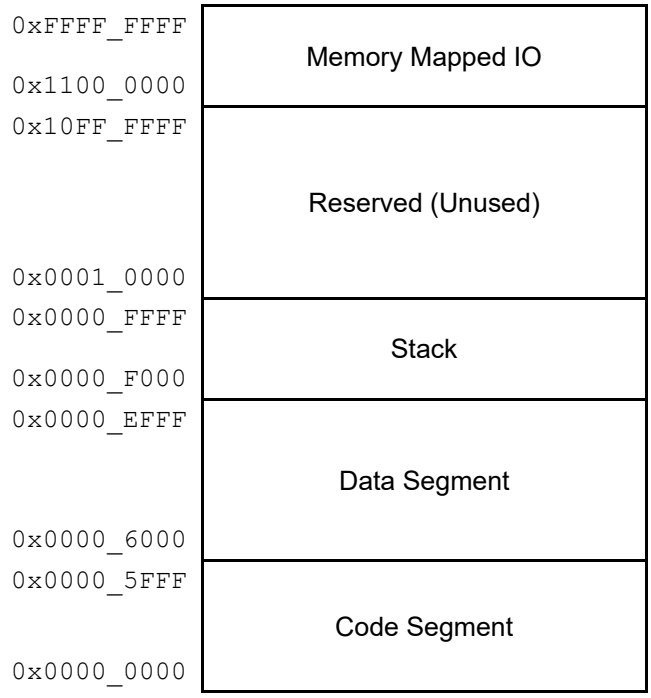


Figure 1: RISC-V OTTER Memory Map

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. Most instructions fall into the six standard RISC-V instruction types listed in Table 3, however there is another instruction type that the RISC-V OTTER uses for some interrupt related instructions. Table 3 shows the formats of the six standard RISC-V instruction types; the Detailed RISC-V OTTER Assembly Instruction Description section shows the format of the seventh type (see **`csrrw`**, for example).

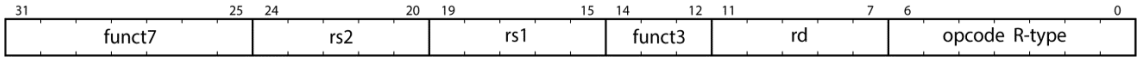
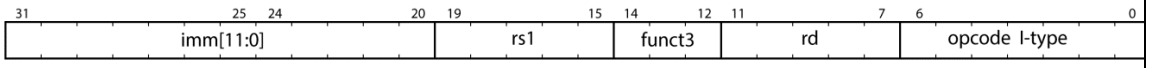
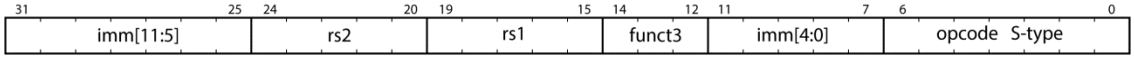
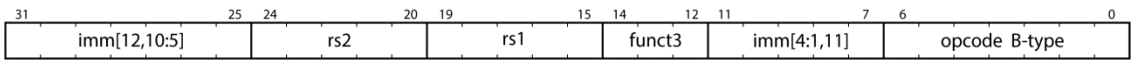
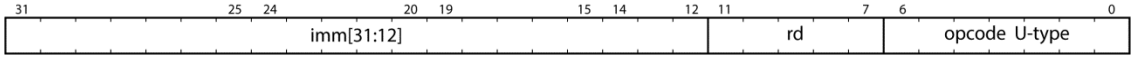
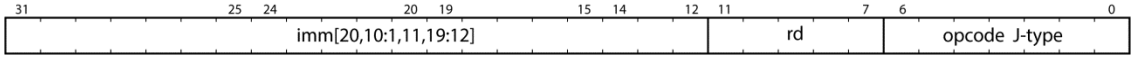
Instr Type	Instruction Format
R-type	
I-type	
S-type	
B-type	
U-type	
J-type	

Table 4: Instruction types and associated instruction formats.

Instruction Type: R-type

Figure 2 shows the R-type instruction format. Table 4 lists the instructions using the R-type format.

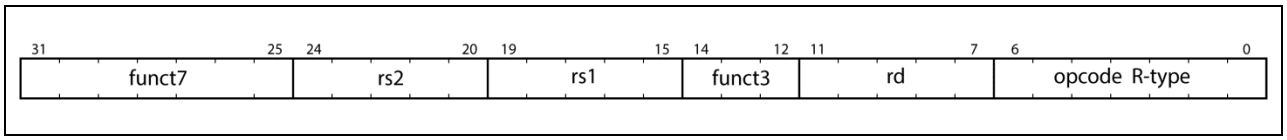


Figure 2: R-type instruction format.

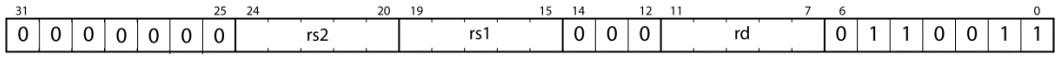
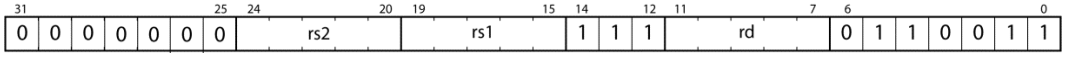
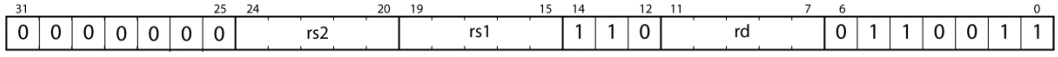
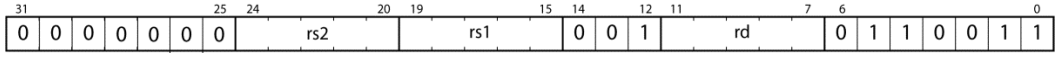
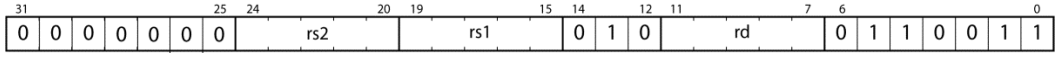
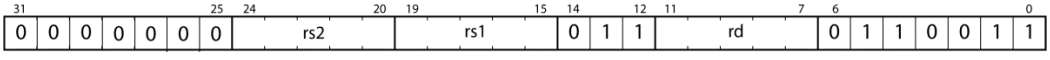
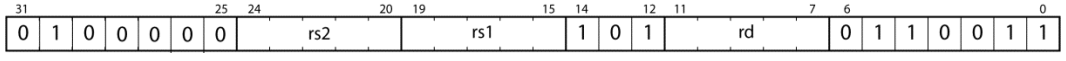
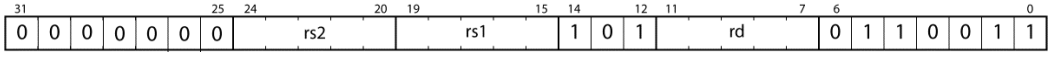
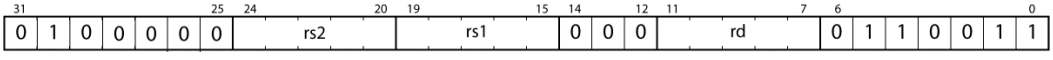
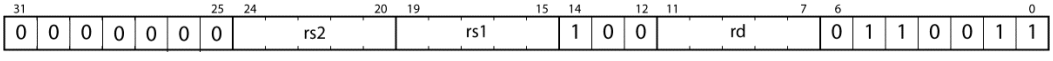
add `add rd,rs1,rs2`	
and `and rd,rs1,rs2`	
or `or rd,rs1,rs2`	
sll `sll rd,rs1,rs2`	
slt `slt rd,rs1,rs2`	
sltu `sltu rd,rs1,rs2`	
sra `sra rd,rs1,rs2`	
srl `srl rd,rs1,rs2`	
sub `sub rd,rs1,rs2`	
xor `xor rd,rs1,rs2`	

Table 5: R-type instructions with opcodes.

Instruction Type: I-type

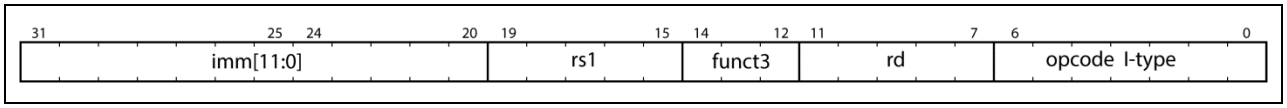


Figure 3: I-type instruction format.

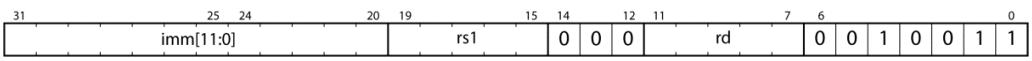
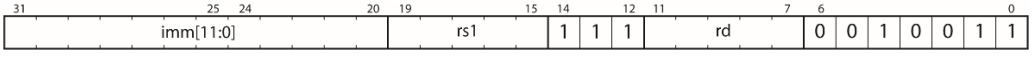
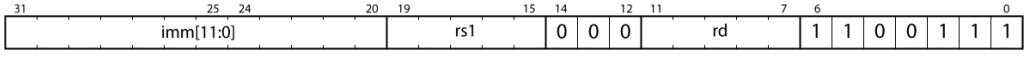
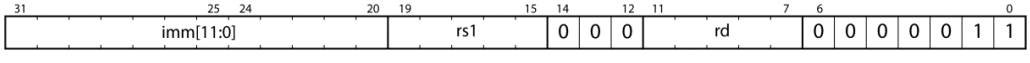
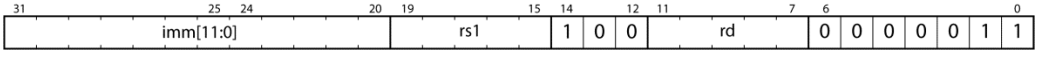
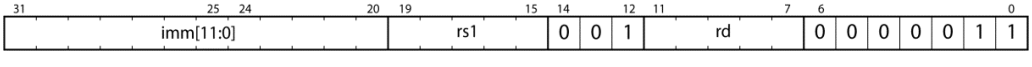
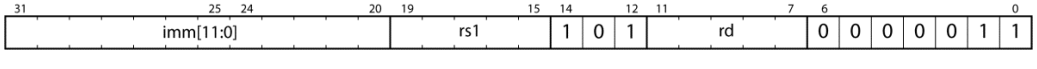
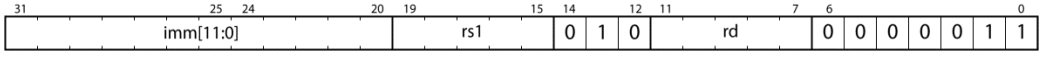
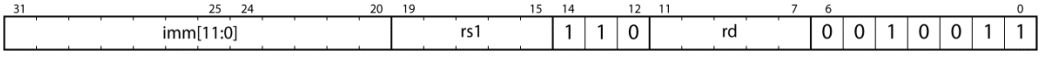
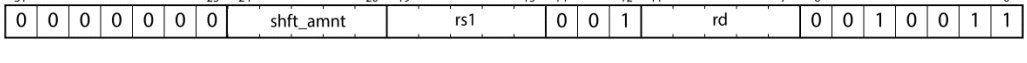
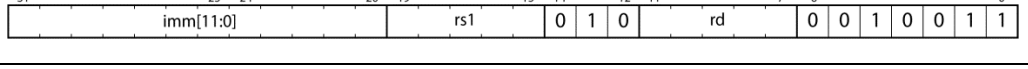
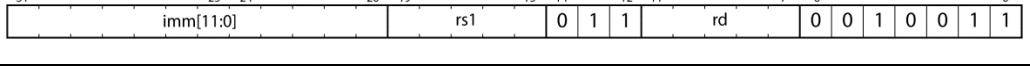
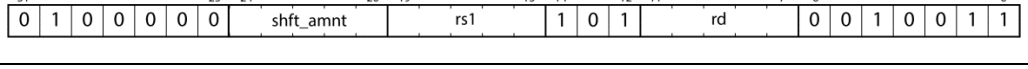
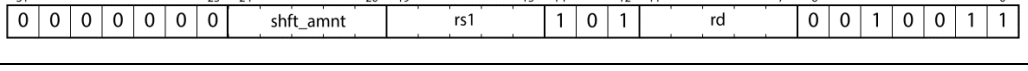
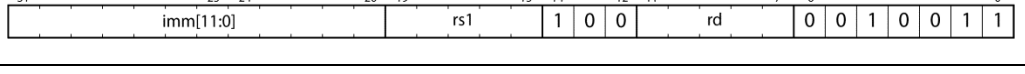
addi `addi rd,rs1,imm`	
andi `andi rd,rs1,imm`	
jalr `jalr rd,rs1,imm`	
lb `lb rd,imm(rs1)`	
lbu `lbu rd,imm(rs1)`	
lh `lh rd,imm(rs1)`	
lhu `lhu rd,imm(rs1)`	
lw `lw rd,imm(rs1)`	
ori `ori rd,rs1,imm`	
slli `slli rd,rs1,imm`	
slti `slti rd,rs1,imm`	
sltiu `sltiu rd,rs1,imm`	
srai `srai rd,rs1,imm`	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

Table 6: I-type instructions with opcodes.

Instruction Type: S-type

Figure 4 shows the S-type instruction format. Table 6 lists the instructions using the S-type format.

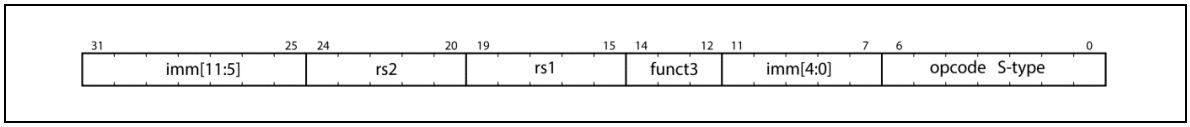


Figure 4: S-type instruction format.

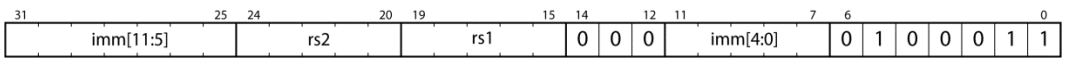
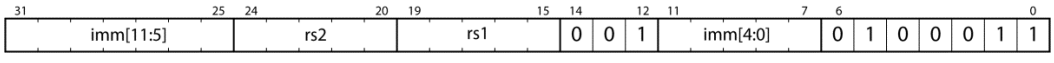
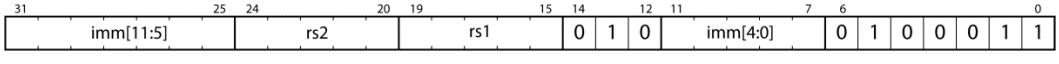
sb `sb rs2,imm(rs1)`	
sh `sh rs2,imm(rs1)`	
sw `sw rs2,imm(rs1)`	

Table 7: S-type instructions with opcodes.

Instruction Type: B-type

Figure 5 shows the B-type instruction format. Table 7 lists the instructions using the B-type format.

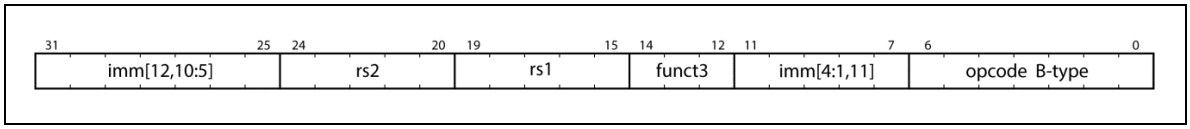


Figure 5: B-type instruction format.

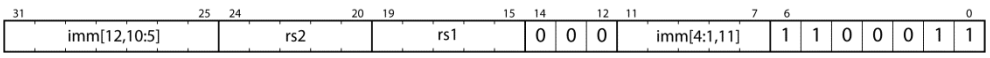
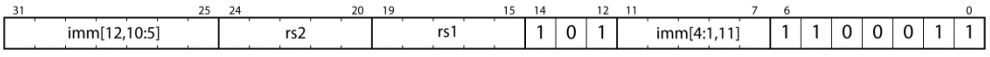
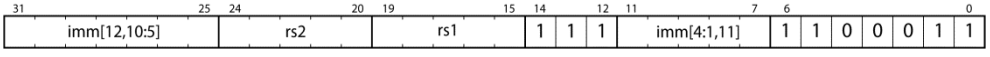
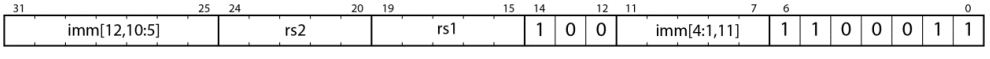
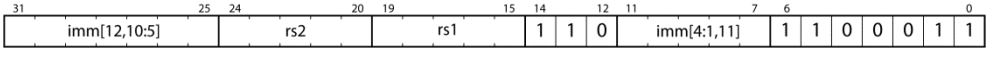
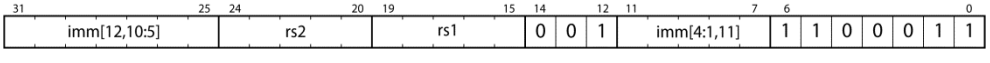
beq `beq rs1,rs2,imm`	
bge `bge rs1,rs2,imm`	
bgeu `bgeu rs1,rs2,imm`	
blt `blt rs1,rs2,imm`	
bltu `bltu rs1,rs2,imm`	
bne `bne rs1,rs2,imm`	

Table 8: B-type instructions with opcodes.

Instruction Type: U-type

Figure 6 shows the U-type instruction format. Table 8 lists the instructions using the U-type format.

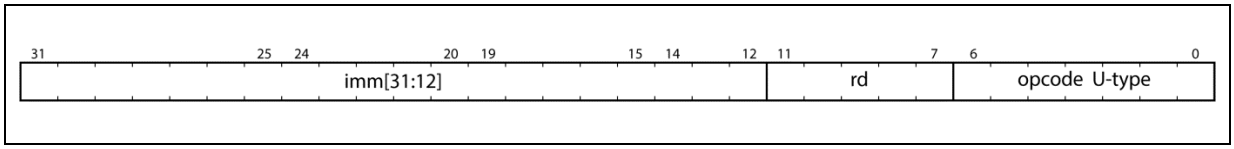


Figure 6: U-type instruction format.

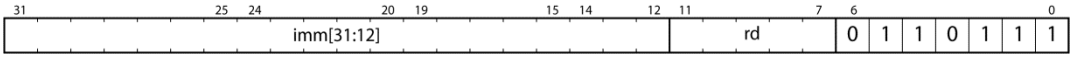
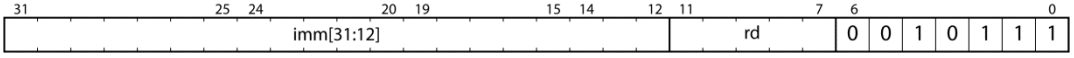
lui `lui rd,imm`	
auipc `auipc rd,imm`	

Table 9: U-type instructions with opcodes.

Instruction Type: J-type

Figure 7 shows the J-type instruction format. Table 9 lists the instructions using the J-type format.

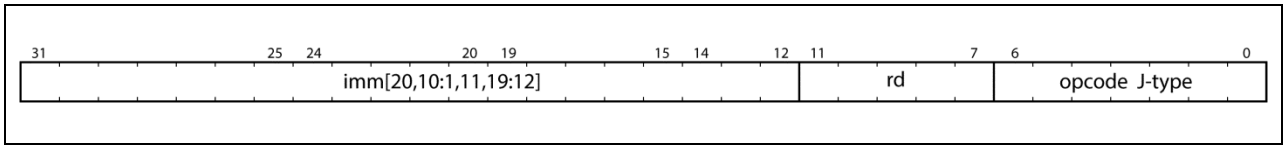


Figure 7: J-type instruction format.

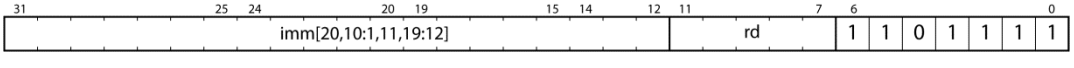
jal `jal rd,imm`	

Table 10: J-type instructions with opcodes.

The RISC-V OTTER ISA Formats and Opcodes

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12,10:5]	rs2	rs1	funct3	imm[4:1,11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20,10:1,11,19:12]				rd	opcode	J-type

RISC-V Base Instruction Set

imm[31:12]				rd	0110111	LUI	U
imm[31:12]				rd	0010111	AUIPC	U
imm[20,10:1,11,19:12]				rd	1101111	JAL	J
imm[11:0]		rs1	000	rd	1100111	JALR	I
imm[11:0]		rs1	000	rd	0000011	LB	I
imm[11:0]		rs1	001	rd	0000011	LH	I
imm[11:0]		rs1	010	rd	0000011	LW	I
imm[11:0]		rs1	100	rd	0000011	LBU	I
imm[11:0]		rs1	101	rd	0000011	LHU	I
imm[11:0]		rs1	000	rd	0010011	ADDI	I
imm[11:0]		rs1	010	rd	0010011	SLTI	I
imm[11:0]		rs1	011	rd	0010011	SLTIU	I
imm[11:0]		rs1	110	rd	0010011	ORI	I
imm[11:0]		rs1	100	rd	0010011	XORI	I
imm[11:0]		rs1	111	rd	0010011	ANDI	I
0000000	*imm[4:0]	rs1	001	rd	0010011	SLLI	I
0000000	*imm[4:0]	rs1	101	rd	0010011	SRLI	I
0100000	*imm[4:0]	rs1	101	rd	0010011	SRAI	I
imm[12,10:5]	rs2	rs1	000	imm[4:1,11]	1100011	BEQ	B
imm[12,10:5]	rs2	rs1	001	imm[4:1,11]	1100011	BNE	B
imm[12,10:5]	rs2	rs1	100	imm[4:1,11]	1100011	BLT	B
imm[12,10:5]	rs2	rs1	101	imm[4:1,11]	1100011	BGE	B
imm[12,10:5]	rs2	rs1	110	imm[4:1,11]	1100011	BLTU	B
imm[12,10:5]	rs2	rs1	111	imm[4:1,11]	1100011	BGEU	B
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB	S
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH	S
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	S
0000000	rs2	rs1	000	rd	0110011	ADD	R
0100000	rs2	rs1	000	rd	0110011	SUB	R
0000000	rs2	rs1	001	rd	0110011	SLL	R
0000000	rs2	rs1	010	rd	0110011	SLT	R
0000000	rs2	rs1	011	rd	0110011	SLTU	R
0000000	rs2	rs1	100	rd	0110011	XOR	R
0000000	rs2	rs1	101	rd	0110011	SRL	R
0100000	rs2	rs1	101	rd	0110011	SRA	R
0000000	rs2	rs1	110	rd	0110011	OR	R
0000000	rs2	rs1	111	rd	0110011	AND	R
csr		rs1	001	rd	1110011	CSRRW	sys
csr		rs1	011	rd	1110011	CSRRC	sys
csr		rs1	010	rd	1110011	CSRRS	sys
0011000	01000	0000	000	00000	1110011	MRET	sys

Table 11: Everything you want to know about RISC-V instructions but were afraid to ask.

RISC-V OTTER Assembly Instructions Brief Listing

Program Control					
jal	rd,imm	**j**	imm		
jalr	rd,rs1,imm	**jr**	rs		
call	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)			**sw**	rs2,imm(rs1)
lbu	rd,imm(rs1)				
lhu	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	rd,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					
csrrc	rd,csr,rs1	**auipc**	rd,imm	**li**	rd,imm
csrrs	rd,csr,rs1	**lui**	rd,imm	**la**	rd,imm
csrrw	rd,csr,rs1			**mv**	rd,rs
csrw	rs1,csr			**nop**	

Table 12: RISC-V OTTER Brief format instruction set listing.

(pseudoinstructions are shaded)

RISC-V OTTER Assembly Instruction Overview

Table 12 lists RISC-V OTTER instructions; shaded instructions are pseudoinstructions

Instruction	Description	RTL	Comment
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)<<12)	
beq rs1,rs2,imm	branch if equal	PC ← PC + sext(imm) if (X[rs1] == X[rs2])	imm ≠ value
beqz rs1,imm	branch if equal to zero	PC ← PC + sext(imm) if (X[rs1] == 0)	imm ≠ value
bge rs1,rs2,imm	branch if greater than or equal	PC ← PC + sext(imm) if (X[rs1] $\geq_s$ X[rs2])	imm ≠ value
bgeu rs1,rs2,imm	branch if greater than or equal unsigned	PC ← PC + sext(imm) if (X[rs1] $\geq_u$ X[rs2])	imm ≠ value
bgez rs1,imm	branch if greater than or equal to zero	PC ← PC + sext(imm) if (X[rs1] $\geq_s$ 0)	imm ≠ value
bgt rs1,rs2,imm	branch if greater than	PC ← PC + sext(imm) if (X[rs1] $>_s$ X[rs2])	imm ≠ value
bgtu rs1,rs2,imm	branch if greater than unsigned	PC ← PC + sext(imm) if (X[rs1] $>_u$ X[rs2])	imm ≠ value
bgtz rs1,rs2,imm	branch if greater than zero	PC ← PC + sext(imm) if (X[rs1] $>_s$ 0)	imm ≠ value
ble rs1,rs2,imm	branch if less than or equal	PC ← PC + sext(imm) if (X[rs1] $\leq_s$ X[rs2])	imm ≠ value
bleu rs1,rs2,imm	branch if less than or equal (unsigned)	PC ← PC + sext(imm) if (X[rs1] $\leq_u$ X[rs2])	imm ≠ value
blez rs1,rs2,imm	branch if less than or equal zero	PC ← PC + sext(imm) if (X[rs1] $\leq_s$ 0)	imm ≠ value
blt rs1,rs2,imm	branch if less than	PC ← PC + sext(imm) if (X[rs1] $<_s$ X[rs2])	imm ≠ value
bltz rs1,imm	branch if less than zero	PC ← PC + sext(imm) if (X[rs1] $<_s$ 0)	imm ≠ value
bltu rs1,rs2,imm	branch if less than (unsigned)	PC ← PC + sext(imm) if (X[rs1] $<_u$ X[rs2])	imm ≠ value
bne rs1,rs2,imm	branch if not equal	PC ← PC + sext(imm) if (X[rs1] ≠ X[rs2])	imm ≠ value
bnez rs1,imm	branch if not equal to zero	PC ← PC + sext(imm) if (X[rs1] ≠ 0)	imm ≠ value
call label	branch to subroutine WARNING: overwrites X6	X[rd] ← PC + 8; PC ← &symbol (rd=X1 if rd omitted)	imm ≠ value overwrites X6
csrrc rd,csr,rs1	control & status reg read and bit clear	X[rd] ← CSR[csr]; CSR[csr] ← CSR[csr] & ~rs1	clears part of reg
csrrs rd,csr,rs1	control & status reg read and bit set	X[rd] ← CSR[csr]; CSR[csr] ← CSR[csr] | rs1	sets part of reg
csrrw rd,csr,rs1	control & status register read & write	X[rd] ← CSR[csr]; CSR[csr] ← rs1	writes entire reg
csrw rs1,csr	control & status register write	CSR[csr] ← rs1	
j imm	unconditional branch	PC ← PC + sext(imm)	imm ≠ value
jal rd,imm **jal imm**	unconditional branch with offset	X[rd] ← PC + 4; PC ← PC + sext(imm)	imm ≠ value rd=X1 if rd omitd
jalr rd,rs1,imm **jalr rs1** **jalr rs1,imm**	unconditional branch with offset & link	X[rd] ← PC+4; PC ← (X[rs1] + sext(imm)) & ~1	imm ≠ value rd=X1 if rd omitd
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] ← zext(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] ← zext(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 << 12	
mret	machine mode exception return	PC ← CSR[mepc]; CSR[mstatus(mie) ← mstatus(mpie)	
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]	
ori 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][4:0]	
slli rd,rs1,imm	logical shift left immediate	X[rd] ← X[rs1] << imm[4:0]	
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] [4:0]	
srai rd,rs1,imm	arithmetic shift right immediate	X[rd] ← X[rs1] $>>_s$ imm[4:0]	
srl rd,rs1,rs2	logical shift right	X[rd] ← X[rs1] >> X[rs2][4:0]	
srli rd,rs1,imm	logical shift right immediate	X[rd] ← X[rs1] >> imm[4:0]	
sub rd,rs1,rs2	subtract	X[rd] ← X[rs1] – X[rs2]	
xor rd,rs1,rs2	exclusive OR	X[rd] ← X[rs1] ^ X[rs2]	
xori rd,rs1,imm	exclusive OR immediate	X[rd] ← X[rs1] ^ sext(imm)	

Table 13: RISC-V OTTER Instructions with RTL description.

RISC-V OTTER Immediate Value Generation

Table 13 lists the immediate value format for the RISC-V OTTER.

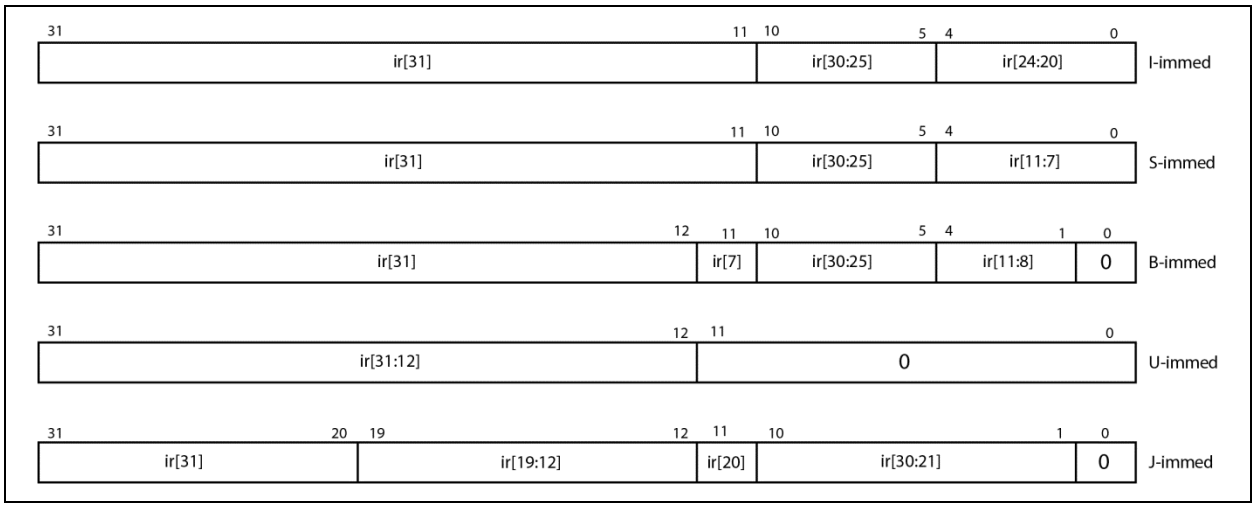


Table 14: RISC-V OTTER Immediate values based on instruction formats.

Detailed RISC-V OTTER Assembly Instruction Description

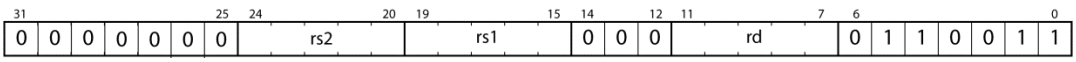
The following section lists each of the RISC-V instructions in a detailed format. The instruction details include the following:

- 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:

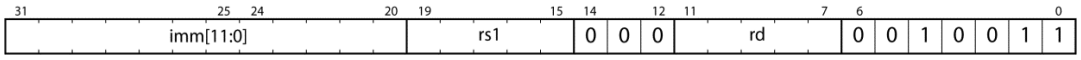
```
add   x10,x10,x12    # addition of values in registers X10 & X12;
                     # result stored in X10; X12 is not affected.
                     #  x10 = 0x0000_00A4  x12 = 0x0000_00C7    (before exec)
                     #  x10 = 0x0000_016B  x12 = 0x0000_00C7    (after exec)
```

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:

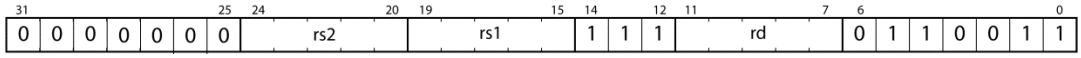
```
addi  X10,X11,0x0DC  # addition of values in X11 to 0xDC
                     # result stored in X10; X11 is not affected.
                     #  X10 = 0x0000_0045  X11 = 0x0000_0024    (before exec)
                     #  X10 = 0x0000_0100  X11 = 0x0000_0024    (after exec)
```

See Also: **add, sub**

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

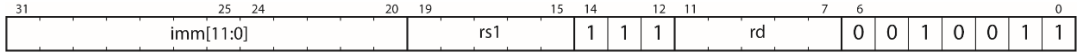
```
and    X1,X4,X5  # bitwise and of values in register X4 & X5;
                 # result is placed in X1; X4 & X5 values don't change
                 #  X1=0x0000_00A4; X4=0x0000_00C7  (before execution)
                 #  X5=0x0000_0084 (before execution)
                 #  X1=0x0000_0084  X4=0x0000_00C7 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 & 0xFF0;
                     # result is placed in X2; X4 value doesn't change
                     # immed value is sign extended to 0xFFFF_FFF0
                     #  X2=0x0000_00E2; X4=0x0000_00C9  (before exec)
                     #  X1=0x0000_00C0  X4=0x0000_00C9  (after exec)
```

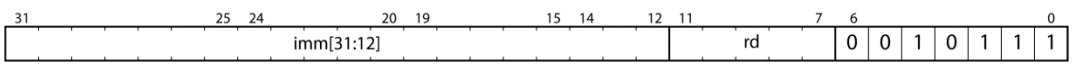
See Also: and, ori, xori

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

RTL: *X[rd] ← PC + (imm<<12)*	**Forms:**	**auipc rd,imm**

Description: The **auipc** instruction sums an immediate value and the current value of the program counter (PC), and stores the results in the destination register rd. The 20-bit immediate value is left-shifted 12 bit locations and the lower 12-bits are cleared before summing. This instruction only modifies the destination register rd.

Instruction Format (U-type)



Usage:

```
    auipc    X10,0x1DFF2  # add the U-type immediate value to the PC, the
                          # immediate is left shifted 12-bits before adding to PC
                          #
                          # PC=0x0000_0078 X10=0x0000_FFFF  (before exec)
                          # PC=0x0000_007C X10=0x1DFF_2078  (after exec)
```

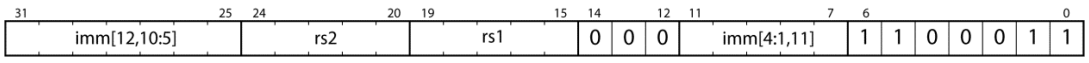
See Also: lui, jal, jalr

beq	***branch if equal***

RTL: *PC ← PC + sext(imm << 1) 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 be 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 imm operand must be a label.

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 << 1) 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 be 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**". The imm operand must be a label.

Usage:

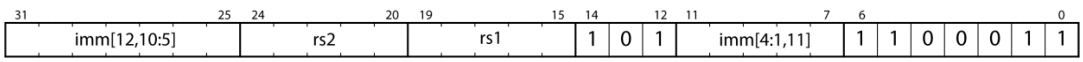
```
        beqz    X10,Oak     # branch to the instruction at the address
                            #  associated with the Oak label if the value in
        nop                    #  X10 equals 0
        nop                 #     X10=0x0000_0000  PC=0x00DF_0000   (before exec)
        nop                 #
        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 << 1) 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 be 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 imm operand must be a label.

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**

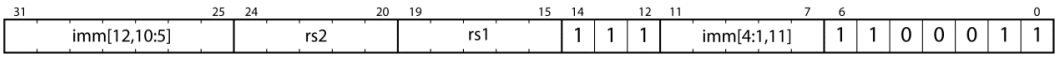
bgeu	branch if greater than or equal unsigned

RTL: *PC ← PC + sext(imm << 1) 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 be 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 imm operand must be a label.

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 << 1) 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 be 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**". The imm operand must be a label.

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 << 1) 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 be 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**". The imm operand must be a label..

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 << 1) 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 (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 be 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**". The imm operand must be a label.

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 << 1) 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 be 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**". The imm operand must be a label.

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 << 1) 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 be 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**". The imm operand must be a label.

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: **bge**

bleu	branch if less than or equal (unsigned)	(pseudoinstruction -- bgeu)
RTL: *PC ← PC + sext(imm << 1) 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 (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 be 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**". The imm operand must be a label.

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 << 1) 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 be 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**". The imm operand must be a label.

Usage:

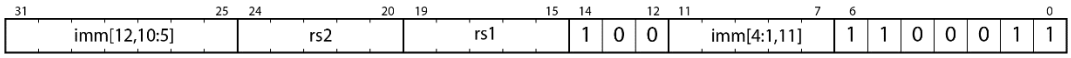
```
        blez    X10,Nom      # branch to the instruction at the address
                             #  associated with the Nom label if the value in
        nop                  #  X10 is less than or equal to 0
        nop                  #     X10=0xE000_0010  PC=0x0A34_103C   (before exec)
        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 << 1) 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 be 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 imm operand must be a label.

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 << 1) 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 be 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**". The imm operand must be a label.

Usage:

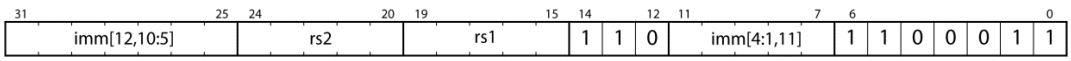
```
        bltz    X10,Mug      # branch to the instruction at the address
                             #  associated with the Mug label if the value in X10
        nop                  #  is less than 0
        nop                  #     X10=0x8000_0001          (before exec)
        nop                  #     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 << 1) 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 (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 be 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 imm operand must be a label.

Instruction Format (B-type)



Usage:

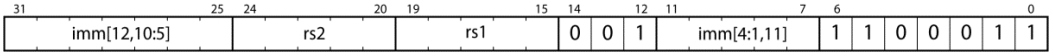
```
        bltu    X10,X11,Pig  # branch to the instruction at the address
                             #  associated with the Pig label if the value in
                             #  X10 is less than the value in X11
        nop                  #     X10=0x8000_FFFF  X11=0xE000_FFF0  (before exec)
        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: **blt**

bne	branch if not equal		
RTL: *PC ← PC + sext(imm << 1) 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 be 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 imm operand must be a label.

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 be 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**". The imm operand must be a label.

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 ← &label*	**Forms:**	**call label**

Description: The **call** instruction is a pseudoinstruction used to transfer program control to another location in program memory. The **call** instruction causes 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. The label operand can't be a number. WARNING: the **auipc** instruction associated with the **call** pseudoinstruction overwrites X6 (t1), which can cause hard-to-find errors in your programs.

Usage:

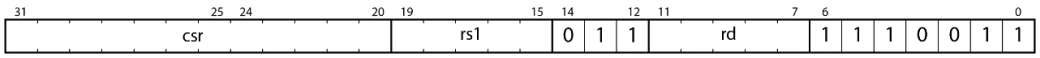
```
        call       Sue     # branch to the instruction at the address
                           #  associated with the Sue; store return address in X1
        nop                # X1=0x0044_2220   PC=0x0FD3_1494    (before exec)
        nop                #
        nop                #
Sue:    nop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

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

csrrc	control & status register read & clear bits	
RTL: *X[rd] ← CSR[csr]; CSR[csr] ← CSR[csr] & ~rs1*	**Form:**	**csrrc rd,csr,rs1**

Description: This instruction copied the designated CSR register to rd then clears bits in the CSR register as designated by the set bits in rs1. The CSR contains several special purpose registers including: **mepc** (CSR[0x341]), **mtvec** (CSR[0x305]), and **mstatus** (CSR[0x300]); this instruction allows for the clearing individual bits in the CSR register rather than writing a new 32-bit value.

Instruction Format



Usage:

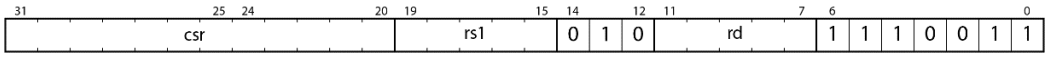
```
  csrrc     x10,0x341,x15  #
                           # x10=0x0000_0300   x15=0x0000_4000  (before exec)
                           # CSR[0x341]=0xFFFF_FFFF            (before exec)
                           #
                           # x10=0xFFFF_FFFF   x15=0x0000_4000  (after exec)
                           # CSR[0x341]=0xFFFF_BFFF            (after exec)
```

See Also: mret, csrrs, csrrw

csrrs	control & status register read & set bits	
RTL: *X[rd] ← CSR[csr]; CSR[csr] ← CSR[csr] | rs1*	**Form:**	**csrrs rd,csr,rs1**

Description: This instruction copied the designated CSR register to rd then sets bits in the CSR register as designated by the set bits in rs1. The CSR contains several special purpose registers including: **mepc** (CSR[0x341]), **mtvec** (CSR[0x305]), and **mstatus** (CSR[0x300]); this instruction allows for the setting individual bits in the CSR register rather than writing a new 32-bit value.

Instruction Format



Usage:

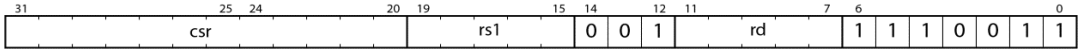
```
csrrs     x10,0x341,x15  #
                         # x10=0x0000_8FFF   x15=0x0000_8000  (before exec)
                         # CSR[0x341]=0xFFFF_0000             (before exec)

                         # x10=0xFFFF_0000   x15=0x0000_8000  (after exec)
                         # CSR[0x341]=0xFFFF_8000             (after exec)
```

See Also: mret, csrrc, csrrw

csrrw	control & status register read & write	
RTL: *X[rd] ← CSR[csr]; CSR[csr] ← rs1*	**Form:**	**csrrw rd,csr,rs1**

Description: This instruction reads from and writes to the CSR. The CSR contains several special purpose registers including: **mepc** (CSR[0x341]), **mtvec** (CSR[0x305]), and **mstatus** (CSR[0x300]).

Instruction Format



Usage:

```
csrrw     x10,mepc,x15   #
                         # x10=0x0000_0300   x15=0x4040_4000  (before exec)
                         # CSR[0x341]=0x3333_3333             (before exec)

                         # x10=0x3333_3333   x15=0x4040_4000  (after exec)
                         # CSR[0x341]=0x4040_4000             (after exec)
```

See Also: mret, csrrc, csrrs

<table>
<tr><td><code>csrw</code></td><td colspan="2">control & status register write</td><td>Pseudoinstrution (csrrw)</td></tr>
<tr><td colspan="2">RTL: X[rd] ← CSR[csr]; CSR[csr] ← rs1</td><td>Form:</td><td><code>csrw rs1, csr</code></td></tr>
<tr><td colspan="4">Description: This instruction writes the value in rs1 to the CSR address specified by csr. The value of csr is the address of one of the CSR registers. <code>csrw</code> is a pseudoinstruction based on the csrrw instruction and is equivalent to: “csrrw <code>X0,csr,rs1</code>”.</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td>Usage:</td><td colspan="3"><pre>
csrw   x15,mtvec # x15=0x00FF_EE00   (before exec)
                   # CSR[mtvec]=0x0000_AAAA (before exec)

                   # x15=0x00FF_EE00   (after exec)
                   # CSR[mtvec]=0x00FF_EE00 (after exec)
</pre></td></tr>
<tr><td colspan="4">See Also: mret, csrrw</td></tr>
</table>

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**". The immed value must be a label.

Usage:

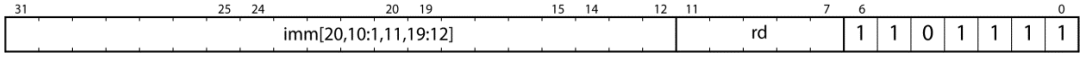
```
        j     Bug       # unconditional branch to the instruction at the address
        nop             #  adjusted by the immediate value
        nop             #     PC=0x001F_0500             (before exec)
        nop             #
Bug:    nop             #     PC=0x001F_0510             (after exec)
```

See Also: **jal, jalr, jr**

jal	unconditional branch with offset & link
RTL: *X[rd] ← PC + 4; PC ← PC + sext(imm << 1)*	**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 "links" by saving the address of the instruction after jal ("next" instruction) 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. The immediate operand must be a label.

Instruction Format (J-type)



Usage:

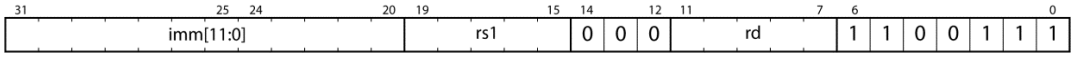
```
        jal     X8,Emu        # branch to the instruction at the address
                              #  associated with the Emu label; place address of
                              #  next instruction in PC
        nop                   #     X8=0xE000_FFFF              (before exec)
        nop                   #     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 & link		
RTL: *X[rd] ← PC+4; PC ← (X[rs1] + sext(imm))*		**Forms:**	**jalr rd,rs1,imm** **jalr rd,imm(rs1)** **jalr rs1** **jalr rs1,imm**

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 "links" by saving the address of the instruction after **jalr** to the destination rd. The instruction sign extends the 12-bit immediate value and adds it to the value in the source register; the resulting value is loaded into the PC. Care should be taken to ensure the resulting value falls on a word boundary. 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, X0 is used for the destination register. The immediate value cannot be a label.

Instruction Format (I-type)



Usage:

```
jalr    X4,X1,12     # jump to address specified in X1 register + immed value
                     #     X1=0x0045_FF00  X4=0x0034_0034  (before exec)
                     #     PC=0x00E4_0520                  (before exec)

                     #     X1=0x0045_FF00  X4=0x00E4_0524  (after exec)
                     #     PC=0x0045_FF0C                  (after exec)
```

See Also: jal, j, jr

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

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)**". The imm operand can't be a label. VENUS does not support "**jr rs1,imm**" version.

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`** instruction 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.

Usage:

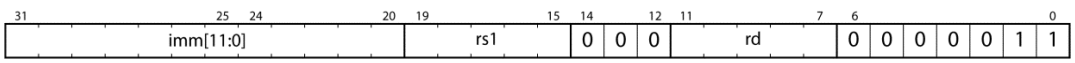
```
        la    x10,Ear:    # load the address associated with the Ear label into
                          #  the destination register at the address
        nop               # X10=0x0044_2330          (before exec)
        nop               # X10=0x00D3_1494          (after exec)
              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

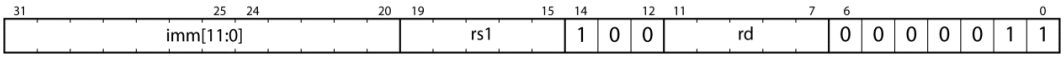
Instruction Format (I-type)



Usage:

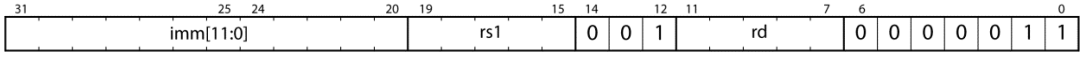
```
      lb   X10,-8(X20)   # load byte from memory at address specified by the
                         #  immediate value and source register (X20)
                         #     X20=0x0010_FB09                  (before exec)
                         #     X10=0x00E2_7500                  (before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

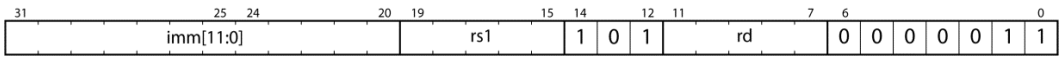
Usage:	`lbu X10,-8(X20) # load byte from memory at address specified by the`
	`# immediate value and source register (X20)`
	`# X20=0x0010_FB09 (before exec)`
	`# X10=0x00E2_7500 (before exec)`
	`# M[0x0010_FB01]=0xF3 (before exec)`
	`# X20=0x0010_FB09 (after exec)`
	`# X10=0x0000_00F3 (after exec)`
	`# 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 and is equivalent to the following instruction: "**addi rd,X0,imm**" if the immediate value can be represented with the 12-bit immediate field in the addi instruction, or a combination of two instructions (**lui** & **addi**) if the immediate can't be represented by a 12-bit immediate value.		

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**		

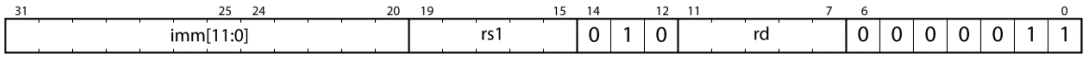
lw	load word into register

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**

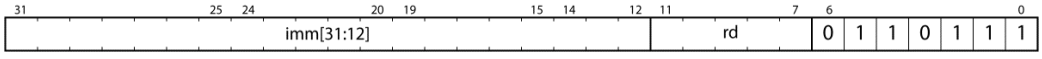
lui	load upper immediate

RTL: *X[rd] ← imm << 12*

Form: `lui rd,imm`

Description: The **lui** instruction loads an immediate value into the destination rd. The 20-bit immediate value is left-shifted 12 bit locations and the lower 12-bits are cleared before loading the result into destination register.

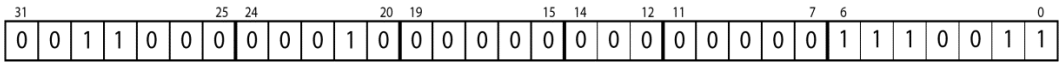
Instruction Format (U-type)



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: *PC] ← CSR[mepc]*	**Form:**	**mret**
Description: This instruction serves as a return from interrupt by loading the CSR[**mepc]** register into the PC. The mepc register is one of the CSR registers. The mepc register is loaded as part of the interrupt cycle and represent the address of the instruction that would have been executed had the MCU not entered the interrupt cycle.		
Instruction Format		

Usage:	`mret  # copy csr[mepc] into PC`
	`# CSR[mepc]=0x4678_0000`
	`# PC = 0x2020_3030 (before exec)`
	`# PC = 0x4678_0000 (after exec)`	

See Also: csrrw		

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**".		

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**".		

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**".		

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**".		

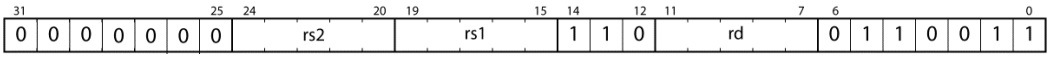
Usage:	`not X14,X15 # perform 1's complement on the value in the source`
	`# register X15 and copies result to destination register X14`
	`#`
	`# X14=0x021F_3B00 X15=0x5555_5555 (before exec)`
	`# X14=0xAAAA_AAAA X15=0x5555_5555 (after exec)`	

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:

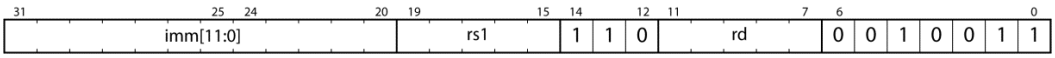
```
or   X14,X15,X16  # perform an inclusive OR in the values in source registers
                  #  X15 & X16 and stores result in destination register X14
                  #
                # X14=0x9832_AD34 X15=0x0034_3B00 X16=0xFFFF_00FF  (before exec)
                # X14=0xFFFF_3BFF X15=0x0034_3B00 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 12-bit immediate value is sign extended 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,x1,0**".

Usage:

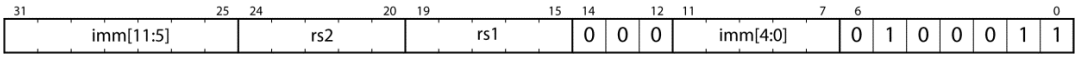
```
ret      # return from subroutine
         #  X1=0x0236_FE30  PC=0x0323_3434    (before exec)
         #  X1=0x0236_FE30  PC=0x0236_FE30    (after exec)
```

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**"

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**"

Usage:

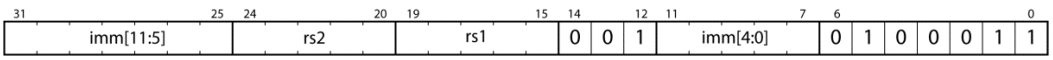
```
sgtz   X10,X12   # if the value in X12 is greater than 0, then 1
                 #  is written to X10; otherwise 0 is written to X10.
                 #
                 # X10=0x1812_DD74 X12=0x8000_0001   (before exec)
                 # X10=0x0000_0000 X12=0x8000_0001   (after exec)
```

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:

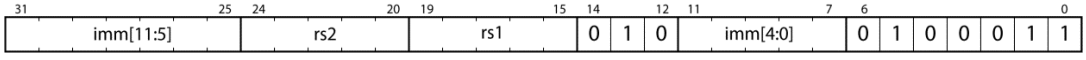
```
sh    X10,2(X11)  # store the 16 LSBs of register X10 at the
                  #  address specified by X11 and the offset
                  #  X10=0x0011_7591 X11=0x0000_F34A  (before exec)
                  #  M[0x0000_F34C]=0x1133            (before exec)
                  #
                  #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

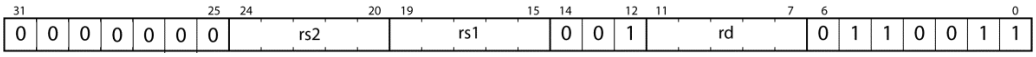
```
sw    X10,4(X11)  # store the value in register X10 at the
                  #  address specified by X11 and the offset
                  #  X10=0x2211_7591 X11=0x0000_F348  (before exec)
                  #  M[0x0000_F34C]=0x3411_FACE       (before exec)
                  #
                  #  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 **sll** instruction only considers the five LSBs of the rs2 source register.

Instruction Format (R-type)



Usage:

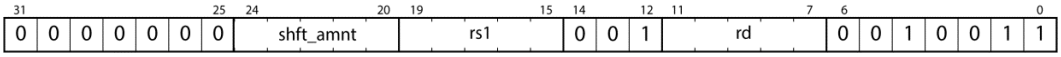
```
sll   X20,X22,X23  # left shift the value in X22 by the amount specified
                   #  by the lower five bits in X23; store result in X20
                   #
                # X20=0x1812_AD34 X22=0x0001_3B00 X23=0xE000_10C8  (before exec)
                # X20=0x013B_0000 X22=0x0001_3B00 X23=0xE000_10C8  (after exec)
```

See Also: **slli**

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)
```

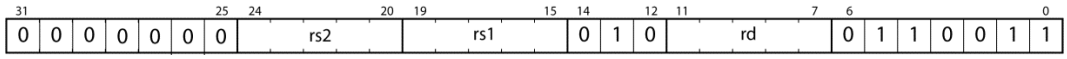
See Also: **sll, srl, srli**

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)
```

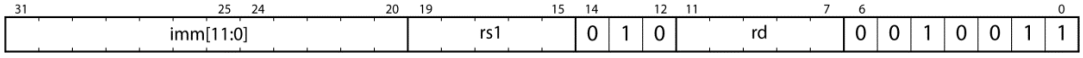
See Also: **sltiu**

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:

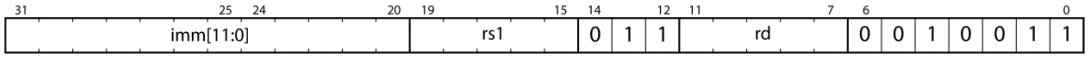
sltiu	Set if less than immediate unsigned

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**

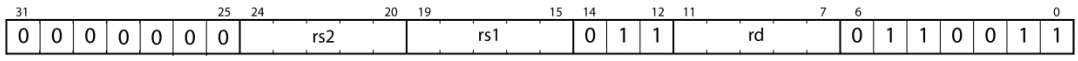
sltu	Set if less than unsigned

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.

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.

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)
```

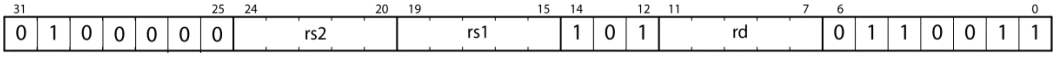
See Also: **sltz, sltu**

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)
```

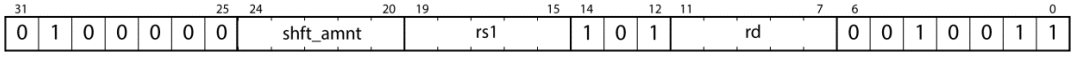
See Also:

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 5-bit 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:

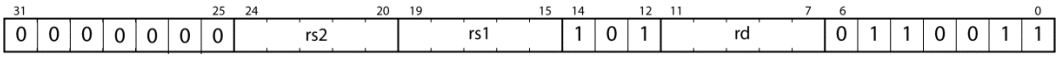
```
srai   X10,X15,16  # arithmetic right shift X15 value the amount specified
                   #  by the immediate value (shft_amnt); store result in X10
                   #
                   # X10=0x0D12_1D02  X15=0xE301_3C14   (before exec)
                   # X10=0xFFFF_E301  X15=0xE301_3C14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

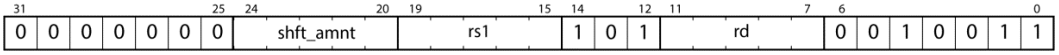
```
srl   X10,X20,X21  # right shift the value in X20 by the amount specified
                   #  by the lower five bits in X21; store result in X10
                   #
               # X10=0x1812_AD34 X20=0x8EDF_1B00 X21=0xE034_10CD  (before exec)
               # X10=0x0004_1EF8 X20=0x0EDF_1B00 X21=0xE034_10CD  (after exec)
```

See Also: sra

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

Description: The **srli** instruction right 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:

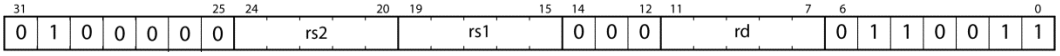
```
srli   X14,X15,20  # right shift the value in X14 by the amount specified
                   #  by the immediate value (shft_amnt); store result in X14
                   #
                   # X14=0x0F12_AD22 X15=0xC300_3C14   (before exec)
                   # X14=0x0000_0C30 X15=0xC300_3C14  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

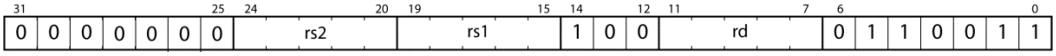
```
sub   X10,X15,X16  #subtract value in X16 from value in X15;
                   # store result in X10
                   #
               # X10=0x1812_AD34 X15=0x0001_1B70 X16=0xFFFF_FFFF  (before exec)
               # X10=0x0001_1B71 X16=0x0001_1B70 X16=0xE034_10CD  (after exec)
```

See Also:

xor	bitwise exclusive OR		
RTL: *X[rd] ← X[rs1] ^ X[rs2]*		**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:

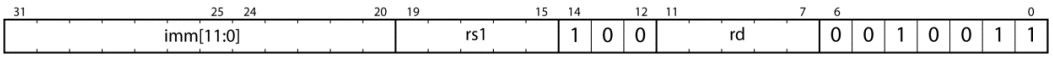
```
xor   X14,X15,X16 # perform an inclusive OR in the values in source registers
                  #  X15 & X16 and stores result in destination register X14
                  #
               # X14=0x9832_AD34 X15=0x0000_3B00 X16=0xFFFF_0000  (before exec)
               # X14=0xFFFF_3B00 X15=0x0000_3B00 X16=0xFFFF_0000  (after exec)
```

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

xori	bitwise exclusive OR		
RTL: *X[rd] ← X[rs1] ^ 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 12-bit immediate value is sign extended 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,-1     # 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 8 shows an example assembly language program highlighting respectable OTTER assembly language source code appearance.

```
# Programmer: Paul Hummel
# Date: 01-27-22
#
# This program uses a simple ISR to count the number of interrupts
# The ISR sets a flag that is checked in an infinite loop in main
# When the flag is set, the updated count is output to the 7 Seg,
# interrupts are re-enabled, and the interrupt flag is reset

.eqv SSEG, 0x11000040

INIT: li    s1, SSEG         #store port address in register
      la    t0, ISR
      csrrw x0, mtvec, t0    #setup ISR address

      li    t0, 0x08         #MIE bit in CSR[mstatus]
      li    t1, 0x80         #MPIE bit in CSR[mstatus]
      csrrs x0, mstatus, t0  #enable interrupts

      addi  t0, x0, 0        #clear interrupt flag
      addi  s0, x0, 0        #clear interrupt count
      sw    s0, 0(s1)        #clear SSEG

LOOP: beq   t0, x0, LOOP     #check for interrupt flag

      sw    s0, 0(s1)        #update SSEG with new count
      csrrs x0, mstatus, t0  #re-enable interrupts
      addi  t0, x0, 0        #clear interrupt flag
      j     LOOP

ISR:  addi  t0, x0, 1        #set interrupt flag
      addi  s0, s0, 1        #count interrupts

      csrrc x0, mstatus, t1  #clear MPIE to disable interrupts on return
      mret                   #return from ISR
```

Figure 8: Example RISC-V OTTER assembly language code showing required coding style.