

CSC 411

Computer Organization (Fall 2024)

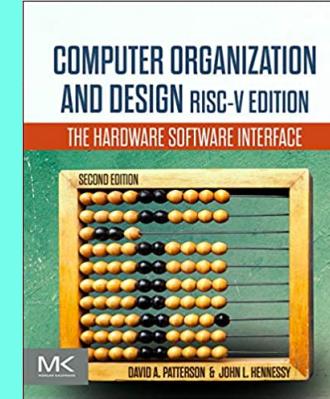
Lecture 11: RISC-V load/store, branches, conditionals, loops

Prof. Marco Alvarez, University of Rhode Island

Disclaimer

Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)
The Hardware/Software Interface



RISC-V Interpreter

Input your RISC-V code here:

```
1 addi x10, x0, -2000
2 sw x10, 48(x0)
3 lb x9, 48(x0)
4 addi x10, x0, -2000
5 lbu x7, 49(x0)
6
7
8
9
10
11
12
13
14
15
```

Reset Step Run CPU: 32 Hz

{line 1}: addi x10, x0, -2000
{line 2}: sw x10, 48(x0)
{line 3}: lb x9, 48(x0)
{line 4}: lb x8, 49(x0)

Features

- Reset to load the code, Step one instruction, or Run all instructions
- Set a breakpoint by clicking on the line number (only for Run)
- View registers on the right, memory on the bottom of this page

Supported Instructions

- Arithmetics: ADD, ADDI, SUB
- Logical: AND, ANDI, OR, ORI, XOR, XORI
- Sets: SLT, SLTU, SLTU, SLT, SLTUI
- Memory: LW, SW, LBU, SB, SLL, SLLI
- Memory: LW, SW, LB, SB
- PC: LUI, AUIPC
- Jumps: JAL, JALR
- Branches: BEQ, BNE, BLT, BGE, BLTU, BGEU

RISC-V Reference: [riscv-spec-v2.2.pdf](https://riscv.org/specifications/riscv-v2.2.pdf)

<https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/>

Venus Editor Simulator Chocopy

Run Step Prev Reset Dump Trace Re-assemble from Editor

PC	Machine Code	Basic Code	Original Code
0x0	0x83000513	addi x10 x0 -2000	addi x10, x0, -2000
0x4	0x02A02823	sw x10 48(x0)	sw x10, 48(x0)
0x8	0x03000483	lb x9 48(x0)	lb x9, 48(x0)
0xc	0x03100403	lb x8 49(x0)	lb x8, 49(x0)
0x10	0x03104383	lbu x7 49(x0)	lbu x7, 49(x0)

Registers Memory Cache VDB

Integer (R) Floating (F)

zero 0x00000000

ra (x1) 0x00000000

sp (x2) 0x7FFFFFFD

gp (x3) 0x10000000

tp (x4) 0x00000000

t0 (x5) 0x00000000

t1 (x6) 0x00000000

t2 (x7) 0x00000000

s0 (x8) 0xFFFFFFFFF8

s1 (x9) 0x00000030

a0 (x10) 0xFFFFFFF830

a1 (x11) 0x7FFFFFFD

a2 (x12) 0x00000000

a3 (x13) 0x00000000

a4 (x14) 0x00000000

a5 (x15) 0x00000000

a6 (x16) 0x00000000

Copy! Download! Clear!

console output

<https://venus.cs61c.org/>

Larger constants

- Immediate instructions
 - **12-bit immediate** values are sufficient for most cases (range -2048 to 2047)
- Load upper immediate
 - **lui** loads upper 20 bits of 32-bit imm into rd, lower bits are set to zero
 - often used together with **addi** for loading full 32-bit constants

lui rd, imm

addi x1, x0, 2046	x1 (ra)	2046	0x000007fe 0b000000000000000000000000000011
addi x2, x0, 2047	x2 (sp)	2047	0x000007ff 0b000000000000000000000000000011
addi x3, x0, 2048	x3 (gp)	-2048	0xfffff800 0b1111111111111111111111111100
addi x4, x0, 2049	x4 (tp)	-2047	0xfffff801 0b1111111111111111111111111100
lui x5, 200000	x5 (t0)	819200000	0x30d40000 0b001100001101010000000000
addi x6, x5, 2000	x6 (t1)	819202000	0x30d407d0 0b001100001101010000000011

Load and store instructions

Load and store instructions

- RISC-V architecture overview
 - load-store architecture: only **load** and **store** instructions access memory
 - 32 general-purpose registers (x0-x31)
 - memory is **byte-addressed**: each address identifies an 8-bit byte
 - **little endian** memory organization
- RISC-V and x86 do not require words to be **aligned in memory** (alignment strongly encouraged)
 - alignment restriction forces **words** to start at addresses that are multiples of the word size



Loading and storing data

- Values can be loaded from memory



- Values can be stored in memory

Load word (lw)

- The **lw** instruction loads a 32-bit value from memory into rd
 - effective address = rs1 + sign-extended imm

destination register base register
lw rd, imm(rs1)
 offset

Practice

- C code
 - assume all are int variables

```
g = h + A[8];
```
- RISC-V code
 - index 8 requires offset of 32 bytes (4 integers)
 - assume g in x1, h in x2, and base address of A in x3

```
lw x1, 32(x3)
add x1, x1, x2
```

Store word (sw)

- The **sw** instruction stores a 32-bit value from rs2 to memory
 - effective address = rs1 + sign extended imm

source register base register
sw rs2, imm(rs1)
 offset

Practice

- C code
 - assume all are int variables

```
A[7] = h + A[5];
```
- RISC-V code
 - assume h in x1 and base address of A in x2
 - can use an additional register x3

Load and store variations

Byte operations

- load byte: **lb**
 - loads a 8-bit value, **sign-extended** to 32-bits before storing in rd
- load unsigned byte: **lbu**
 - loads a 8-bit value, **zero-extended** to 32-bits before storing in rd
- store byte: **sb**
 - stores 8-bit value from the 8 **lower bits** of register rs2 to memory

Practice

What are the values in
registers x9, x8, and x7?

```
addi x10, x0, -2000
sw x10, 48(x0)
lb x9, 48(x0)

lb x8, 49(x0)
lbu x7, 49(x0)
```

Load and store variations

Halfword operations

- load byte: **lh**
 - loads a 16-bit value, **sign-extended** to 32-bits before storing in rd
- load unsigned byte: **lhu**
 - loads a 16-bit value, **zero-extended** to 32-bits before storing in rd
- store byte: **sh**
 - stores 16-bit value from the 16 **lower bits** of register rs2 to memory

Practice

- What are the resulting values in x3 and x4 respectively?

- express the values using 32-bits (in hexadecimal notation)

```
addi x1, x0, 0x80AB
sw x1, 0(x2)
lb x3, 1(x2)
lbu x4, 0(x2)
```

Practice

- Translate the following C code into RISC-V

- can use an additional register x3

```
// assume x, y are integer pointers  
// stored in x1 and x2 respectively  
*x = *y;
```

Conditionals and loops

Register optimization

- Load-store architecture

- all memory operations involve load-store operations
- accessing memory is significantly slower than accessing registers

- Register optimization

- to minimize memory accesses and improve execution performance
 - compilers keep frequently used variables in registers
 - less frequently used variables may be "spilled" to memory to free up registers for more important data
- challenging: number of available registers, code complexity, frequency of variable usage
- register optimization** is important!

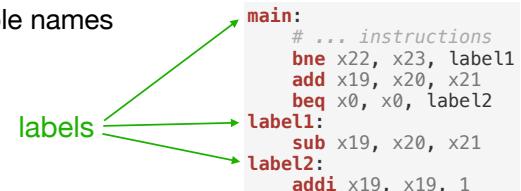
Labels

- A label is an identifier to a particular line of code

- does not count as code, it is just a point of reference

- Labels are defined with unique names

- similar to variable names



- RISC-V has instructions that can “jump” to specific labels

- conditional and unconditional branches

Branches

Conditional branches

- change control to a labeled instruction if condition is **true**
- beq, bne, blt, bge, bltu, bgeu**

opname rs1, rs2, imm

Unconditional jumps

- change control unconditionally
- jal** (jump and link), **jalr** (jump and link register)
- j** (jump) — note this is a **pseudo-instruction**, not an instruction
- example — branch to L1 unconditionally

j L1

Pseudo-instructions

- RISC-V uses pseudo-instructions alongside real instructions
- provide convenient shorthand for common operations
- pseudo-instructions are translated by the assembler into one or more actual RISC-V instructions
- some examples:

pseudo-instruction	equivalent RISC-V instruction
not rd, rs1	xori rd, rs1, -1
mv rd, rs1	addi rd, rs1, 0
nop	addi x0, x0, 0
li rd, imm	translates into one (addi) or two (lui, addi) real instructions depending on the magnitude of the immediate value

Branch/jump instructions

Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

Practice

```
// assume f, g, h, i, j are in
// x19, x20, ...
if (i == j) {
    f = g + h;
} else {
    f = g - h;
}
```

main:

```
# ... instructions
bne x22, x23, label1
add x19, x20, x21
beq x0, x0, label2
label1:
sub x19, x20, x21
label2:
# ... instructions
```

Practice

what are the values in x1 and x2?

```
addi x1, x0, 1
slli x1, x1, 5
addi x2, x0, 10
blt x1, x2, if
beq x0, x0, done
if:
done: addi x1, x1, 1
done: addi x2, x2, 1
```

Signed vs unsigned

- Signed comparison

- blt, bge

- Unsigned comparison

- bltu, bgeu

- Example

```
# assume x22 stores 0xFFFFFFFF
# assume x23 stores 0x00000001
# which instruction branches?
blt x22, x23, Label
bltu x22, x23, Label
```

Loops

- Conditional branches are key to writing loops in RISC-V
 - multiple ways of writing a loop

```
# assume x1 holds the value 4 and x2
# is zero, what is the value of x2?
loop:
    bge x0, x1, done
    addi x1, x1, -1
    addi x2, x2, 2
    beq x0, x0, loop
done:
```

Practice

```
// assume i in x22, k in x24
// base address of save in x25
while (save[i] == k) {
    i += 1;
}
```

main:

... instructions

label3:

```
slli x10, x22, 2
add x10, x10, x25
lw x9, 0(x10)
bne x9, x24, label4
addi x22, x22, 1
beq x0, x0, label3
```

label4:

... instructions

Practice

```
// assume a, b, c, d
// are in x1, x2, x3, x4
// base address of data in x5
do {
    a = a + data[c];
    c = c + d;
} while (c != b);
```

Practice

```
// assume a, b, c, v are
// in x1, x2, x3, x5
switch (v) {
    case 0:
        a = b + c;
        break;
    case 1:
        a = b - c;
        break;
    case 2:
        a = b * c;
        break;
}
```

Practice

```
# translate the following loop to C
# assume that the C-level integer `i` is
# in register x1, x4 holds the C-level
# integer `result`, and x2 holds the
# base address of the integer `MemArray`
    addi x1, x0, 0
    addi x3, x0, 100
loop:
    lw    x5, 0(x2)
    add  x4, x4, x5
    addi x2, x2, 4
    addi x1, x1, 1
    blt x1, x3, loop
```

Summary of instructions

A few standard operations do not appear in the table:

- **not** (exists as a pseudo-instruction)
- **multiplication/division/mod** (exist as an extension to the base RISC-V)
- **floating point arithmetic** (also exist as an extension)

RISC-V operands			
Name	Example	Comments	
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.	
2 ³⁰ memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.	

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	$x5 = x6 + x7$	Three register operands; add
	Subtract	sub x5, x6, x7	$x5 = x6 - x7$	Three register operands; subtract
	Add immediate	addi x5, x6, 20	$x5 = x6 + 20$	Used to add constants
	Load word	lw x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword, unsigned	lh x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword, signed	lhu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Signed halfword from memory to register
	Store halfword	sh x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	lb x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
Data transfer	Load byte, unsigned	lbu x5, 40(x6)	$x5 = \text{Memory}[x6 + 40]$	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
	Load reserved	lrd x5, (x6)	$x5 = \text{Memory}[x6]$	Load; 1st half of atomic swap
	Store conditional	sc,d x7, x5, (x6)	$\text{Memory}[x6] = x5; x7 = 0/1$	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345000	$x5 = 0x12345000$	Loads 20-bit constant shifted left 12 bits
	And	and x5, x6, x7	$x5 = x6 \& x7$	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	$x5 = x6 \mid x8$	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	$x5 = x6 \wedge x9$	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	$x5 = x6 \& 20$	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	$x5 = x6 \mid 20$	Bit-by-bit OR reg. with constant
Logical	Exclusive or immediate	xori x5, x6, 20	$x5 = x6 \wedge 20$	Bit-by-bit XOR reg. with constant
	Shift left logical	sll x5, x6, x7	$x5 = x6 \ll x7$	Shift left by register
	Shift right logical	srl x5, x6, x7	$x5 = x6 \gg x7$	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	$x5 = x6 \gg x7$	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	$x5 = x6 \ll 3$	Shift left by immediate
	Shift right logical immediate	srai x5, x6, 3	$x5 = x6 \gg 3$	Shift right by immediate
	Shift right arithmetic immediate	sral x5, x6, 3	$x5 = x6 \gg 3$	Arithmetic shift right by immediate