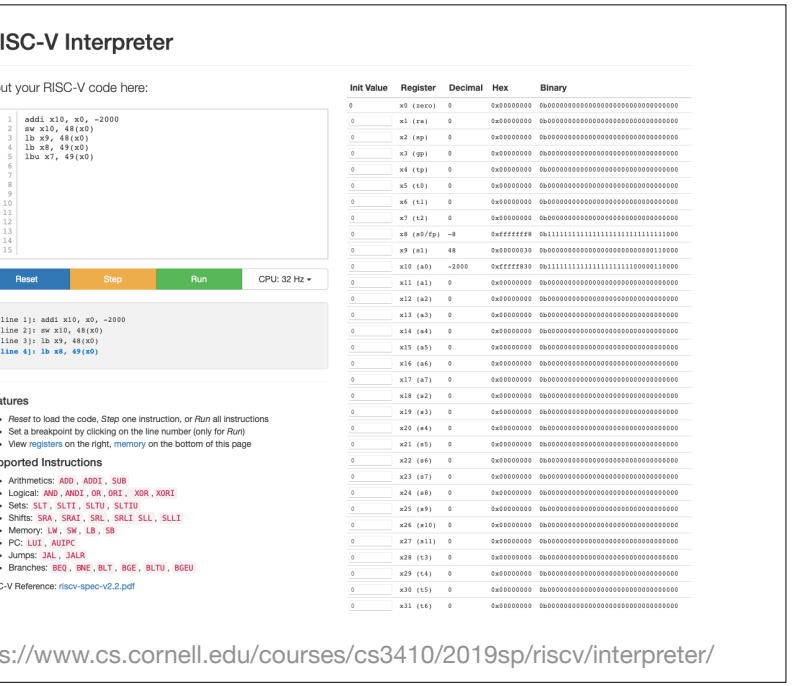
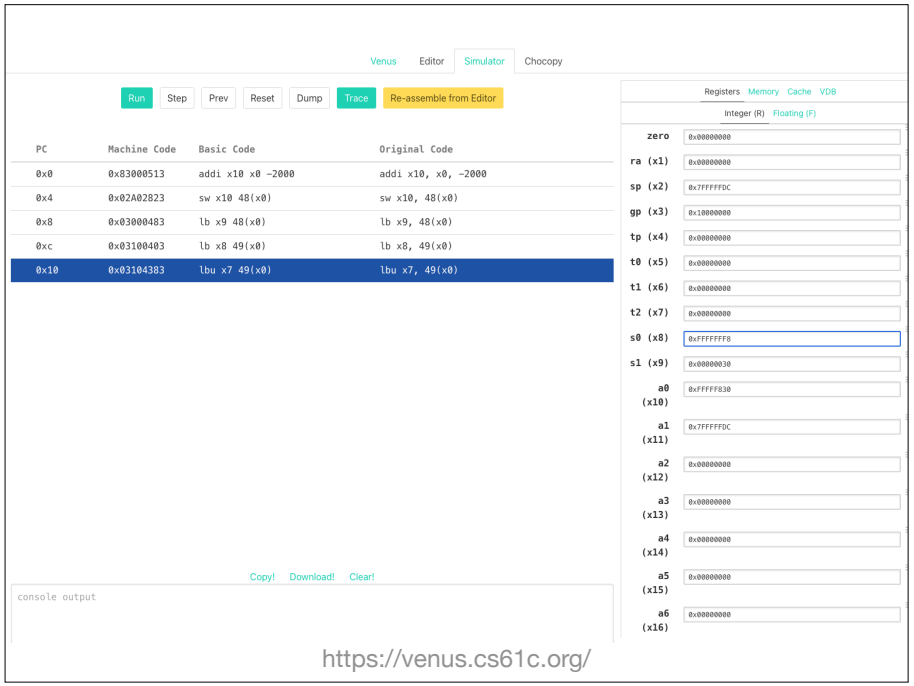
CSC 411

Computer Organization (Spring 2024)
Lecture 13: RISC-V conditionals and loops

Prof. Marco Alvarez, University of Rhode Island



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



https://venus.cs61c.org/

Practice

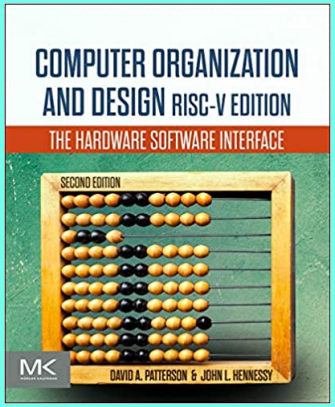
```
# What are the values in
# registers x9, x8, and x7?

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

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

Disclaimer

Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)

The Hardware/Software Interface



So far ...

- Addition / subtraction

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

- Add immediate

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

- Load / store

```
lw rd, imm(rs1)
sw rs2, imm(rs1)
```

Larger constants

- **12-bit I-immediate** values are sufficient for most cases (range goes from -2048 to 2047)
- Using larger constants
  - `lui` (load upper immediate) is used to build 32-bit constants, placing the 20-bit U-immediate value in the top 20 bits of the destination register `rd`, filling in the lowest 12 bits with zero
  - a subsequent operation can be used to fill in the lowest 12 bits with another value

```
lui rd, imm
```

```
addi x1, x0, 2046
addi x2, x0, 2047
addi x3, x0, 2048
addi x4, x0, 2049

lui x5, 200000
addi x6, x5, 2000
```

0	x1 (ra)	2046	0x000007fe	0b00000000000000000000011
0	x2 (sp)	2047	0x000007ff	0b00000000000000000000011
0	x3 (gp)	−2048	0xfffff800	0b11111111111111111111100
0	x4 (tp)	−2047	0xfffff801	0b11111111111111111111100
0	x5 (t0)	819200000	0x30d40000	0b00110000110101000000000
0	x6 (t1)	819202000	0x30d407d0	0b00110000110101000000011

Logical operations

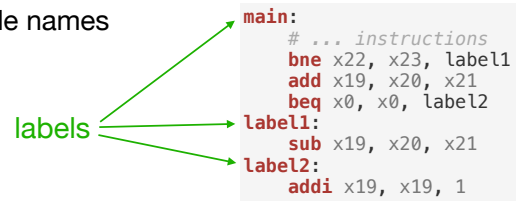
- Instructions for bitwise manipulation

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

Conditionals and loops

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 branch
  - change control to a labeled instruction if condition is `true`
  - `beq, bne, blt, bge, bltu, bgeu`
  - example — branch to L1 if rs1 equals to rs2:

```
beq rs1, rs2, L1
```

- Unconditional branch
  - change control unconditionally
  - `j` (jump) — note this is a **pseudo-instruction**, not an instruction
  - example — branch to L1 unconditionally

```
j L1
```

Branch instructions

Conditional branch	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
Unconditional branch	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

Pseudo-instructions

- RISC-V uses pseudo-instructions alongside real instructions
  - provide shorthand syntax for common operations
  - pseudo-instructions are interpreted by the assembler and translated into 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

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:
        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

```
# 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
```

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;
}
```

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
Data transfer	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ 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 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate