CS 3410 Lab 5

Spring 2026

Cornell Bowers C·IS
Computer Science

Agenda

1. RISC-V Recap
2. Lab Task

RISC-V Recap

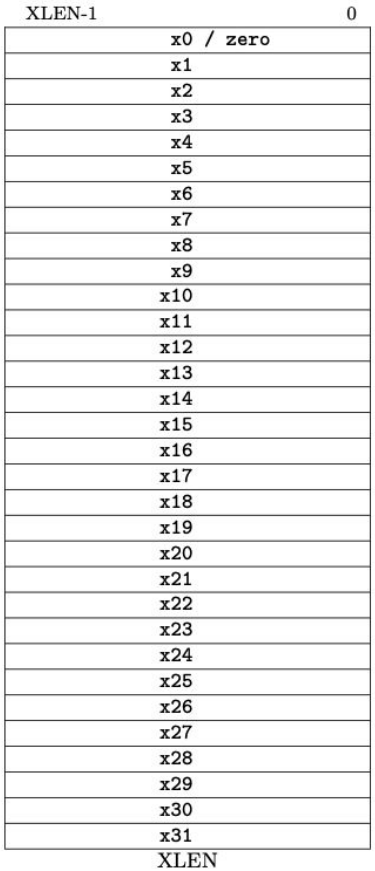
RISC-V 64

32-bit (4 byte) instructions

32 registers x0 - x31. Register names:

- x0 → zero : (always stores value of 0)
- x10 - x17 → a0 - a7
- x5, x6, x7, x28 - x31 → t0 - t6
- x8, x9, x18 - x27 → s0 - s11

64-bit memory



Arithmetic Instructions

- **add** rd, rs1, rs2 reg[rd] ← reg[rs1] + reg[rs2]

  "Add values in source registers rs1 and rs2, writeback to destination register rd"

- **addi** rd, rs1, constant reg[rd] ← reg[rs1] + constant

  "Add value in source reg. rs1 to constant, writeback to destination register rd"

- **sub** rd, rs1, rs2 reg[rd] ← reg[rs1] - reg[rs2]

Multiplication & Division!

- **mul** rd, rs1, rs2 reg[rd] ← reg[rs1] * reg[rs2]
- **div** rd, rs1, rs2 reg[rd] ← reg[rs1] / reg[rs2]

Use shift operations when multiplying/dividing by powers of 2!

- **slli** rd, rs1, shamt reg[rd] ← reg[rs1] << shamt
- **srli** rd, rs1, shamt reg[rd] ← reg[rs1] >>_u shamt
- **srai** rd, rs1, shamt reg[rd] ← reg[rs1] >>_s shamt

Use **slli** to multiply by powers of 2.

Use **srli** to divide unsigned numbers, **srai** to divide signed numbers.

Example: Multiplication

Assume the following register allocation:

- `x` → x5
- `y` → x6
- `z` → x7

C:

```
z = x * y * 2;
```

Assembly:

```
mul x7, x5, x6
slli x7, x7, 1
```

Memory Access: Load and Store!

Load word (32 bit): **lw** rd, offset(rs1)

Store word (32 bit): **sw** rs2, offset(rs1)

Second operand is the address.

- offset(rs1): get value from register rs1, add the constant offset to it → this is the address to read to / write from

lw: puts value at address offset(rs1) into register rd.
sw: stores value in register rs2 at address offset(rs1).

Example: Array access

Assume:

- A has been properly initialized in memory
- @A → x5
- `y` → x6
- x7, x8 are for temp values

C:

```
int A = {1, 2, 3, 58, 0};

A[3] = 69;

y = A[4] + 42;
```

Assembly:

```
addi x7, x0, 69

sw x7, 12(x5)

lw x8, 16(x5)

addi x6, x8, 42
```

Control Flow: Jump & Branch

Branch If Equal: **beq** rs1, rs2, some_label

Branch instructions: choose between moving on to next instruction or jumping to *label.*

- **beq:** if rs1 equals rs2, then jump to location some_label
- Other conditional branches: **bne** (branch if not equal), **blt** (branch if less than), **bge** (branch if greater than or equal to)

Jump: **j** some_label

- **j:** jump to location of some_label

Resources & Instruction Lookup:

- RISC-V reference card
- Exhaustive reference sheet
- ISA manual

Worksheet

Some exercises:

- Translating C → assembly

Tips:

- Consult the RISC-V reference card and ISA manual!
- After writing your assembly file, run it with the 3410 RISC-V Interpreter
  - Initialize register values in the register file display on right
  - Reset to load code, Step one instruction, or Run all instructions

Solutions to Worksheet

1. Load and Store

Solution:

```
lw x2, 4(x1)      # x2 = arr[1]
lw x3, 12(x1)     # x3 = arr[3]
sw x2, 12(x1)     # arr[3] = x2
sw x3, 4(x1)      # arr[1] = x3
```

2. Conditional Control Flow

Recall: x16 holds x, x17 holds y

Solution:

```
1  .if:
2  bge x16, x17, .else
3  sub x17, x16, x17
4  slli x17, x17, 1
5  beq x0, x0, .exit
6  .else:
7  addi x17, x17, -1
8  .exit:
```

Check if x < y

y = x - y

y = y * 2

(j .exit also works here)

y = y - 1

3. Loops

Solution:

- Option 1: Incorrect → should branch with BGE, this loop exits if i < y
- Option 2: Incorrect → should branch with BGE, can enter loop if i > y (suppose y is initialized to a negative value)
- Option 3: Correct
- Option 4: Incorrect → loop never re-iterates, missing branch after body
- Option 5: Correct

4. Putting Everything Together

Recall:
- x1 holds arr
- x2 holds size
- x3 holds product (initialized to 1)
- x4 holds i (uninitialized)

Solution:

```
1   add x4, x0, x0
2   .loop:
3   bge x4, x2, .exit
4   slli x5, x4, 2
5   add x5, x5, x1
6   lw x6, 0(x5)
7   mul x3, x3, x6
8   addi x4, x4, 1
9   j .loop
10  .exit:
```

```
# i = 0

# Check if i < size
# x5 = i * 4
# x5 = x5 + arr (this computes &arr[i])
# x6 = arr[i]
# product = product * arr[i]
#  i = i + 1
```

Assignment Tips

Task 1:

- Pay special attention to the assumptions (register allocation for variables, which registers for temp. values, etc.) that we list.
- You are not graded on tests! But for your own sake, jot down some test cases to run on the interpreter:
  - Different initializations of registers
  - Expected register state after execution

Task 2:

- Once you've written the function in C, try testing it yourself with `main`!