香 港 中 文 大 學
The Chinese University of Hong Kong

CENG3420

Lab 3-2: RISC-V Litter Computer (RISC-V LC)

Lancheng ZOU & Su ZHENG
(Original: Chen BAI)
Department of Computer Science & Engineering
Chinese University of Hong Kong
lczou23@cse.cuhk.edu.hk &
szheng22@cse.cuhk.edu.hk

Spring 2023

Outline

Introduction

Introduction
Overview

Use C programming language to finish lab assignments in following weeks.

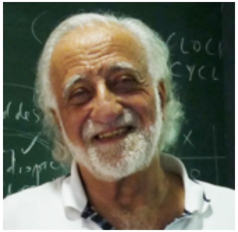
- Lab 2.1 – implement the RISCV-LC Assembler
- Lab 2.2 – implement the RISCV-LC ISA Simulator
- $\rightarrow$ Lab 3.x – implement the RISCV-LC Simulator

NOTICE

Lab2 & Lab3 are challenging!
Once you have passed Lab2 & Lab3, you will be more familiar with RV32I & a basic implementation!

Introduction
Our Lab2 & Lab3 are Inspired by LC-3b

- LC-3b: **Little Computer 3, b** version.
- Relatively simple instruction set.
- Most used in teaching for CS & CE.
- Developed by Yale Patt@UT & Sanjay J. Patel@UIUC.

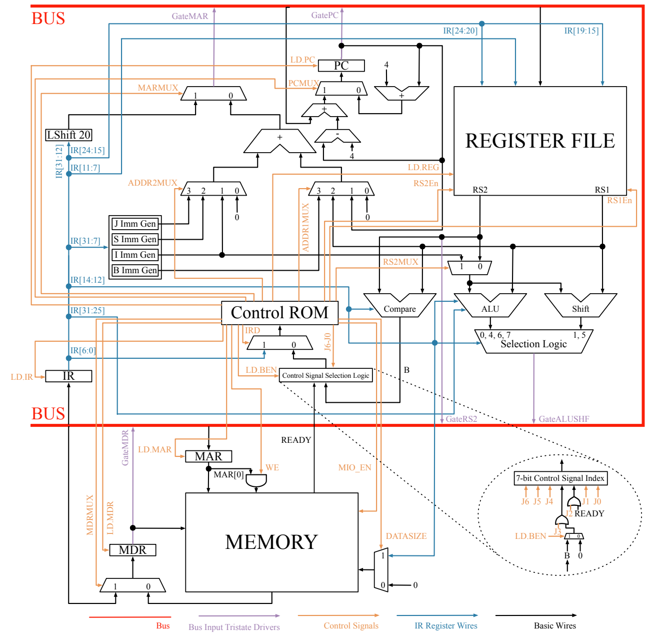


Lab3 Overview

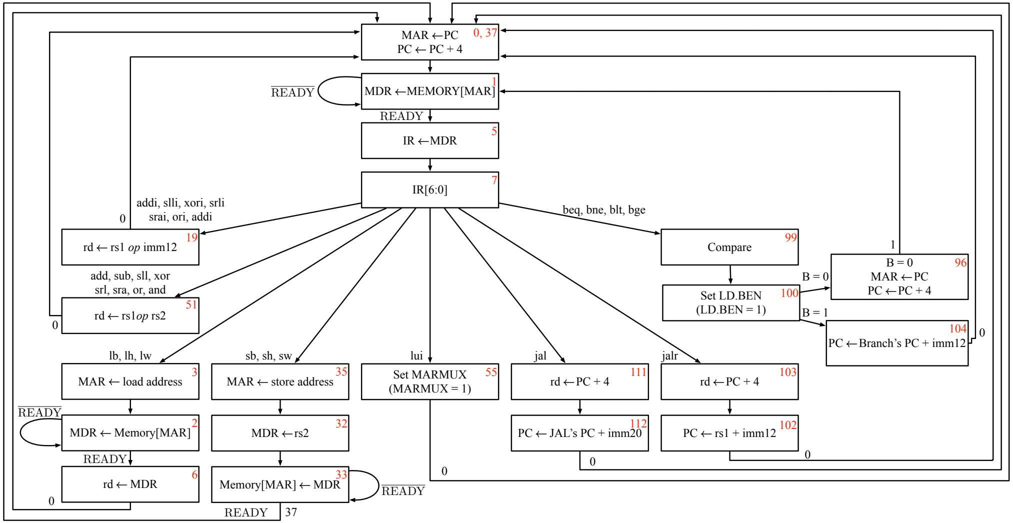
What will we do in Lab 3-2?

- Implement the memory-related operations: load & store.

Introduction
RISC-V LC



Introduction
RISC-V LC



Finite State Machine in RISCV-LC

- A state in RISCV-LC is indexed by 7 bits (7 control signals, *i.e.*, J6, J5, J4, J3, J2, J1, J0).
- A state in RISCV-LC is consist of 33 bits (33 control signals).
- A control read-only memory (ROM) stores all states.
- Control Signal Selection Logic & IR[6:0] (IRD decides the multiplexor) controls the state transition.
- There are 22 different states for RISCV-LC.

RISCV-LC Execution Model

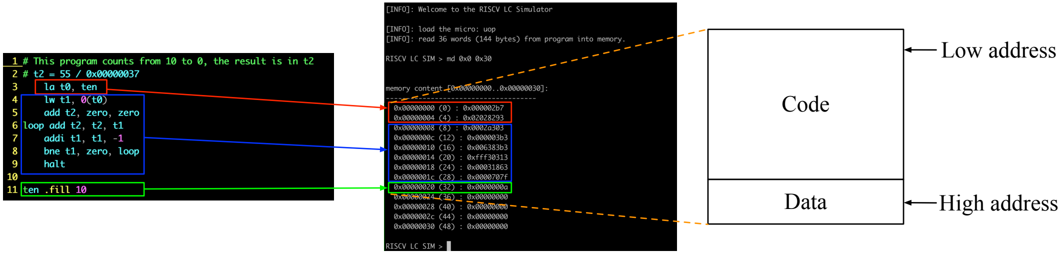
RISCV-LC Execution Model
Notice

Notice

- A design with a single bus and distributed registers.
- A single memory holds both instructions and data.
- The first instruction is placed at the address 0x0.
- The data are placed at the end of all instructions.
- PC is initialized as 0x0.
- No pipeline, and no caches.

RISCV-LC Execution Model

The Data Structure Organization in Memory



```
# This program counts from 10 to 0, the result is in t2
# t2 = 55 / 0x00000037
    la t0, ten
    lw t1, 0(t0)
    add t2, zero, zero
loop add t2, t2, t1
    addi t1, t1, -1
    bne t1, zero, loop
    halt

ten .fill 10
```

```
[INFO]: Welcome to the RISCV LC Simulator

[INFO]: load the micro: uop
[INFO]: read 36 words (144 bytes) from program into memory.

RISCV LC SIM > md 0x0 0x30


memory content [0x00000000..0x00000030]:
-------------------------------------
  0x00000000 (0) : 0x000002b7
  0x00000004 (4) : 0x02028293
  0x00000008 (8) : 0x0002a303
  0x0000000c (12) : 0x000003b3
  0x00000010 (16) : 0x006383b3
  0x00000014 (20) : 0xfff30313
  0x00000018 (24) : 0x00031863
  0x0000001c (28) : 0x0000707f
  0x00000020 (32) : 0x0000000a
  0x00000024 (36) : 0x00000000
  0x00000028 (40) : 0x00000000
  0x0000002c (44) : 0x00000000
  0x00000030 (48) : 0x00000000

RISCV LC SIM >
```

Source codes ↔ Machine codes ↔ Organization in memory

RISCV-LC Execution Model
Initial Steps

1. PC $\rightarrow$ BUS
2. BUS $\rightarrow$ MAR
3. PC +4 $\rightarrow$ PC
4. Memory[MAR] $\rightarrow$ MDR
5. MDR $\rightarrow$ BUS
6. BUS $\rightarrow$ IR
7. Generate control signals according to IR[6:0]

The detail information is shown in Lab 3-1 slides.

Implementations

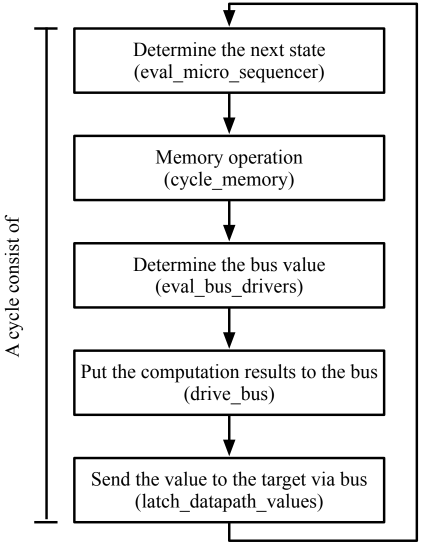
Implementations I
Operations in One Clock Cycle

In “riscv-lc.c”:

```
/*
 * execute a cycle
 */
void cycle() {
    /*
     * core steps
     */
    eval_micro_sequencer();
    cycle_memory();
    eval_bus_drivers();
    drive_bus();
    latch_datapath_values();

    CURRENT_LATCHES =
        NEXT_LATCHES;

    CYCLE_COUNT++;
}
```


A cycle consist of
Determine the next state (eval_micro_sequencer)
Memory operation (cycle_memory)
Determine the bus value (eval_bus_drivers)
Put the computation results to the bus (drive_bus)
Send the value to the target via bus (latch_datapath_values)

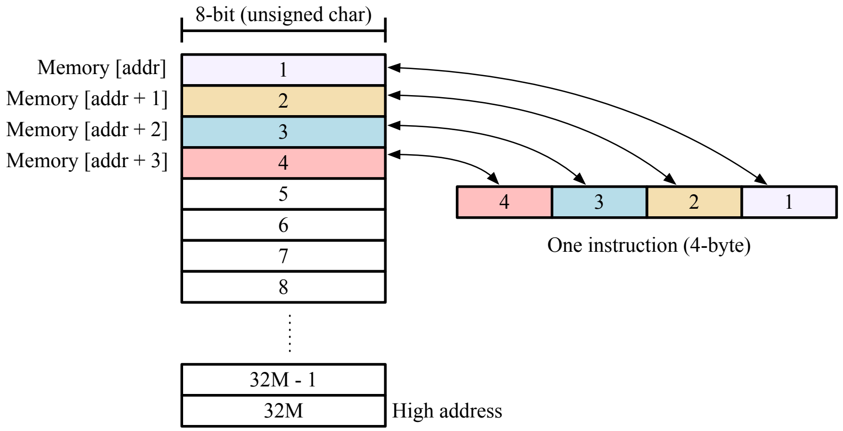
Operations in one clock cycle.

Implementations I
Memory Specifications

In “riscv-lc.h”

```
/* Main memory */
#define MEM_CYCLES 5
#define BYTES_IN_MEM 0x2000000
unsigned char MEMORY[BYTES_IN_MEM];
/* `MEM_VAL` saves the output of the main memory at each cycle
     */
int MEM_VAL;
```

Implementations II
Memory Specifications



Memory organization: little endian.

Implementations I
Data Size Selection

In “functions.c”, we select the memory operation per byte, half word, or word if “data_size” is asserted.

```
int datasize_mux(unsigned int data_size, int funct3, int zero) {
    if (data_size) {
        switch(data_size) {
            case 0:
                return zero;
            case 1:
                return ~(funct3 & 0x3);
        }
    } else
        return zero;
}
```

Implementations I
Data Size Selection

In “functions.c”, we select the memory operation per byte, half word, or word if “data_size” is asserted.

```
return ~(funct3 & 0x3);
```

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	

Instruction formats of memory operations.

For example, for "lw", we will get "∼(0x2 & 0x3)"

Implementations I
Write Memory

The potential implementation:

```
// 8-bit
MEMORY[CURRENT_LATCHES.MAR] = MASK7_0(CURRENT_LATCHES.MDR);
// 16-bit
MEMORY[CURRENT_LATCHES.MAR] = MASK7_0(CURRENT_LATCHES.MDR);
MEMORY[CURRENT_LATCHES.MAR + 1] = MASK15_8(CURRENT_LATCHES.MDR);
// ...
...
```

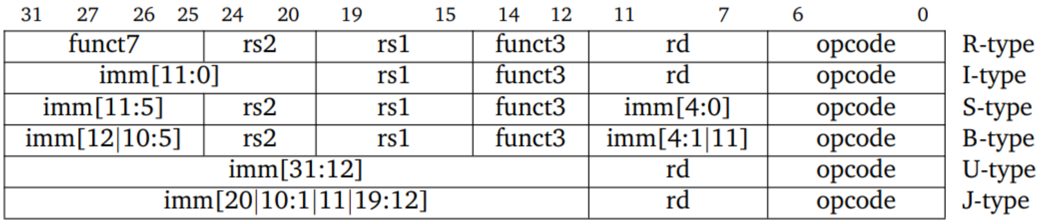
Implementations I
Read Memory

The potential implementation:

```
// 8-bit
val = sext_unit(MEMORY[CURRENT_LATCHES.MAR], 8);
// 16-bit
val = sext_unit((MEMORY[CURRENT_LATCHES.MAR + 1] << 8) + MEMORY[
    CURRENT_LATCHES.MAR], 16);
// ...
...
```

Implementations I
Load Values

```
// LD_REG
REGS[mask_val(CURRENT_LATCHES.IR, 11, 7)] = BUS;
// Why do we load a register with a value from bit 7 to bit 11?
```



31 27 26 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

Instruction formats of RISCV operations.

Lab 3-2 Assignment

Lab 3-2 Assigment
Compile & Run RISCV-LC Simulator

Please visit the website for more information:
https://github.com/MingjunLi99/ceng3420

Get the RV32I Assembler

In the terminal of your computer, type these commands:

- git clone https://github.com/MingjunLi99/ceng3420.git
- cd ceng3420
- git checkout lab3.2

Linux/MacOS environment is suggested

You may use windows subsystem for linux (WSL) or the linux servers in CSE.

Run the RISC-V LC

- $ ./riscv-lc <uop> <*.bin> # RISCV-LC can execute successfully if you have implemented it.

Lab3.2 Assignment
Assignment Content

In **riscv-lc.c**,

- Finish cycle_memory
- Finish latch_datapath_values

The unimplemented codes are commented with Lab3-2 assignment

Lab 3-2 Assignment
Verification

Benchmarks

Verify your codes with these benchmarks (inside the `benchmarks` directory)

- isa.bin
- count10.bin
- swap.bin
- add4.bin

Verification

- isa.bin → a3 = -18/0xffffffee and MEMORY[0x84 + 16] = 0xffffffee
- count10.bin → t2 = 55/0x00000037
- swap.bin → NUM1 (memory address: 0x00000034) changes from 0xabcd to 0x1234 and NUM2 (memory address: 0x00000038) changes from 0x1234 to 0xabcd
- add4.bin → BL (memory address: 0x00000038) changes from -5 (0xfffffffb) to -1 (0xffffffff)

Lab 3-2 Assignment
Submission

Submission Method:

Submit the zip file (including codes and a report) after the whole lectures of Lab3 into **Blackboard**.

Lab 3-2 Assignment
Tips

Tips

Inside `docs`, there are five valuable documents for your reference!

- riscv-lc.pdf
- fsm.pdf
- opcodes-rv32i: RV32I opcodes
- riscv-spec-20191213.pdf: RV32I specifications
- risc-v-asm-manual.pdf: RV32I assembly programming manual

z