Lab 2: Single-Cycle Processor

Computer Organization 2024

1. Goal

A single-cycle processor execute one instruction per clock cycle. It is much simpler than pipelined processors, which you will learn more about it in future lectures, but it is beneficial for understanding the basic concepts of how an instruction is processed. The behavior of single-cycle processor is quite different to pipelined one under MIPS architecture, so we do not recommend you to submit a pipelined processor for this lab.

The main goal of this lab is to implement a single-cycle processor in textbook 4.1-4.4 which can execute a subset of 32-bit MIPS instructions. This lab will help you understand:

1. Essential components of a processor.
2. Construction of the datapath and how data flow thought it.
3. How control signals are generated from operation & function code.
4. Basic concepts of clocking, instruction format, memory allocation, data alignment and endian.
5. Simple usage of SPIM, a MIPS simulator with an assembler.

2. Description

We recommend you to complete this lab by:

1. Read textbook *"4.1 Introduction"* & *"4.2 Logic Design Conventions"* before you start this lab.
2. Read textbook (p. 316-318) *"4.4 The ALU Control"*, then complete the design of ALU control unit in `alu_control.v`.
3. Read textbook (p. 318-321) *"4.4 Designing the Main Control Unit"*, then complete the design of main control unit in `control.v`.
4. Copy your design of ALU and register files in Lab 1 (e. g. `alu.v` , `reg_file.v`), and (Optional) test your design so far by yourself.
5. Read textbook (p. 321-327) *"4.4 Operation of the Datapath & Finalizing Control"* , then complete the design of single-cycle processor (without `j` jump instruction).
   Understand how each instructions flow through the datapath and what happens during each clock cycle.
   Before coding, we recommend you to read `instr_mem.v` and `data_mem.v` to under stand how memories are defined.
6. Test your design by `tb_single_cycle.sv` and `0.s` assembly program.
   We encourage you to write your own program for testing.
   See Appendix for more information about testing with assembly.
7. Read the rest of section 4.4 of textbook, then add `j` jump instruction support into your processor.
   There is no guide since this part! (Optional) test your design by yourself.
8. Implement `li` load immediate support. (Optional) test your design by yourself.
9. Complete the report and have a piece of cake. 🍰

Important Notes

- Instruction Memory (text) has 1 KB and starts at `0x00400000` , which should be your initial PC.
- Data Memory (dynamic data) has 1 KB and starts at `0x10008000` , which your processor can access.
- In Lab 1, you may write to `reg_file.registers[0]` while guarantee read output of `$0` to be zero. However, in Lab 2, please make sure that `reg_file.registers[0]` always be zero for testbench to check register file.

(40%) Arithmetic Instructions & NOP

These are the basic arithmetic instructions which your ALU from Lab 1 can already accomplish.

Instruction	Assembly	Meaning
`add`	`add rd, rs, rt`	`rd` = `rs` + `rt` (signed)
`sub`	`sub rd, rs, rt`	`rd` = `rs` - `rt` (signed)
`and`	`and rd, rs, rt`	`rd` = `rs` & `rt` (bitwise)
`or`	`or rd, rs, rt`	`rd` = `rs`
`slt`	`slt rd, rs, rt`	`rd` = (`rs` < `rt`) ? 1 : 0

There is a special "pseudo instruction" in MIPS called `nop`. It will be translated into `sll $0, $0, 0` (shift left logical) whose machine code is `0x00000000` full zeros. It is quite obvious that this instruction does nothing. In fact, this is exactly why it's called "no operation" and is useful in spite of doing nothing. You will find it helpful when you start to write assembly with branch or jump instructions in the next lab.

Instruction	Assembly	Meaning
`nop`	`nop`	do nothing (translated into `sll $0, $0, 0`)

You don't need to implement `sll` , but you have to make sure that your processor do nothing when executing machine code `0x00000000`.

> We provided the MIPS "Green Card", which contains useful information about MIPS instruction set architecture.

(20%) Load & Store

Load & store instructions are a little bit hard to implement since you need to interact with memory and it's related to clock.

Instruction	Assembly	Meaning
`lw`	`lw rt, immd(rs)`	`rt` = mem[`rs` + `immd`] *(sign-extended)*
`sw`	`sw rt, immd(rs)`	mem[`rs` + `immd`] = `rt` *(sign-extended)*

Make sure you understand when does read/write happen during the clock.

(10%) Branch

Branch instruction check the content of registers and "skip next `immd` instructions".

Instruction	Assembly	Meaning
`beq`	`beq rs, rt, immd`	if (`rs` == `rt`) PC = PC + 4 + (`immd` * 4) *(sign-extended)*

Important Question: Does single cycle implementation of MIPS has "Branch Delay Slot" ? The answer is NO. The processor in this lab should NOT have branch delay slot. Be careful when you test with assembly programs.

(10%) Jump

Jump instruction is simple, jump to the instruction "within" the block.

Instruction	Assembly	Meaning
`j`	`j address`	PC = { (PC+4)[31:28], `address` , 2'b0 }

Same as branch, there should be no delay slot.

(10%) Load Immediate

`li rdest immd` loads a 32-bit immediate `immd` into the desired register `rdest` . Obviously, there's no way our processor can complete this operation by a single instruction. The assembler will translate this pseudo instruction into two real instructions `lui` and `ori` , which writes upper and lower half word each. You need to implement these instructions:

Instruction	Assembly	Meaning
`lui`	`lui rt, immd`	`rt` = { `immd` , 16'b0 }
`ori`	`ori rt, rs, immd`	`rt` = `rs`

Hint: Observe the machine code of `lui`. Can `ori` and `lui` share the same ALU operation? Is there an unused ALUOp code? What is the value of `rs` field in `lui`? How can you take advantage of it?
Hint: Sign-extend is not enough. Add more types of immediate and control signals.

(10%) Report

1. **Architecture Diagrams**

   Show your ALU control (if you have one), main control and single-cycle processor design by "Schematic" tool in Vivado, or draw them by yourself.
   And briefly explain them.

2. **Experimental Result**
   1. Show the waveform screen shot of the test we provided.
   2. What other cases you've tested? Why you choose them?

3. **Answer the following Questions**
   1. When does write to register/memory happen during the clock cycle? How about read?

2. Translate the "branch" pseudo instructions (`blt` , `bgt` , `ble` , `bge`) in the Green Card into real instructions. Only `at` register can be modified, and other common registers should not be modified.
3. Give a single `beq` assembly instruction that causes infinite loop. (consider that there's no delay slot)
4. The `j` instruction can only jump to instructions within the "block" defined by "(PC+4)[31:28]". Design a method to allow `j` to jump to the next block (block number + 1) using another `j` .
5. Why a Single-Cycle Implementation Is Not Used Today?

4. (optional) **Problems Encountered & Solution**

   List some important problem you've met during this lab and there solution.

5. (optional) **Feedback**

   Any thing you want to say to TA team about this lab. How can we improve the lab?

3. Submission

Your submission must be a zip file named `Lab2_ID.zip` where `ID` is your student ID, and structured as below:

```
Lab2_123456789.zip      # There should be NO sub-directory e.g. Lab2_123456789/
├── Lab2_123456789.pdf  # Report (Must be PDF)
└── src                 # contains your source code
    ├── alu_control.v   # (if you have one)
    ├── alu.v           # your ALU (and associated files) from Lab 1
    ├── control.v
    ├── data_mem.v      # it will be replaced when judging
    ├── instr_mem.v     # it will be replaced when judging
    ├── reg_file.v      # your register file from Lab 1
    └── single_cycle.v  # Must included
```

There should be only one module per `.v` file, and the module name should be the same as file name.
The report should be named `Lab2_ID.pdf` and we ONLY accept PDF, any other format will not be scored.
If you want to use System Verilog, the filename must ends with `.sv` .
Do NOT include any testbench, test case and other irrelevant files in your submission.

Before you submit, make sure to pass the testbenches we provided.

!! Any Plagiarism is NOT allowed !!
Any late submission gets only 80% of original score.
Any submission after E3 window is closed will not be accepted!

Appendix: Test with Assembly Program

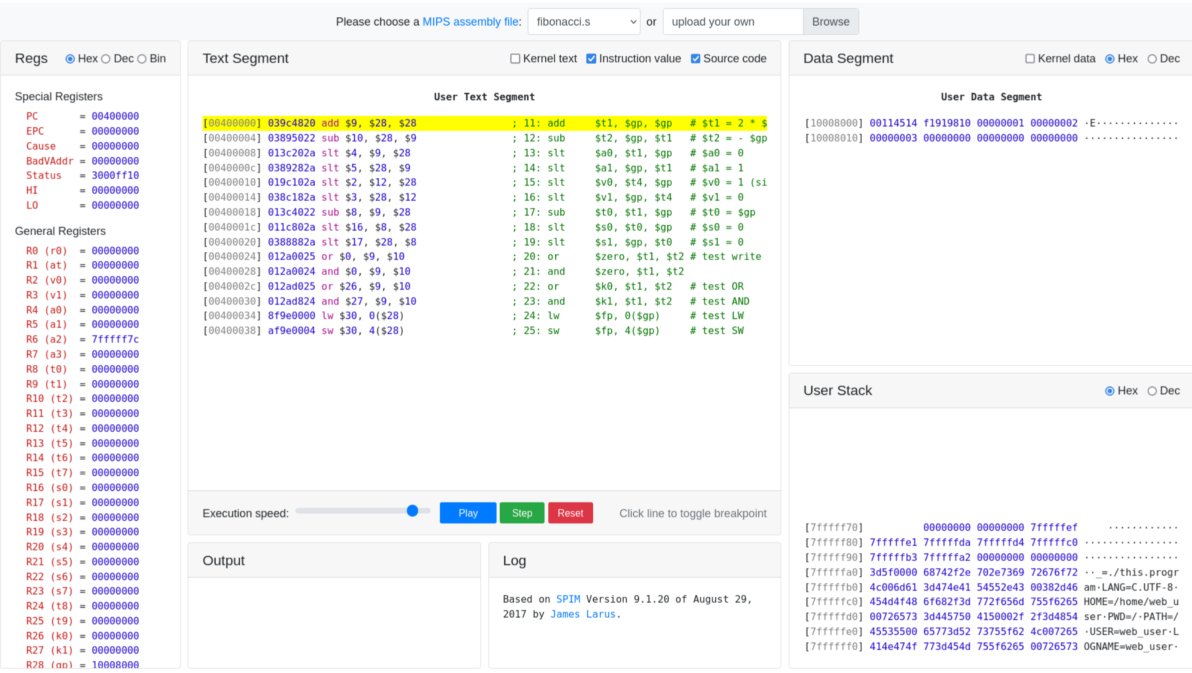
To make testing your processor easier, using assembler and simulator is necessary.
`testbench` directory contains a simple test for arithmetic and load/store instructions, a script `foramt.py`, and testbench `tb_single_cycle.sv`.

```
testbench/
```

```
├── 0.s                   # MIPS assembly program
├── 0.reg.txt             # raw text copied from JsSPIM
├── 0.text.txt
├── 0.data.txt
├── 0.ans_reg.txt         # expext result after program executed
├── 0.ans_data.txt
├── 0.reg.mem             # formated memory files read by testbench
├── 0.text.mem
├── 0.data.mem
├── 0.ans_reg.mem
├── 0.ans_data.mem
├── format.py             # python script to format .txt to .mem
└── tb_single_cycle.sv    # testbench
```

JsSpim

These `.txt` file were copied from outputs on JsSpim an online MIPS32 simulator based on SPIM. You can upload your `.s` assembly program and observe its execution on simulator. Try the provided `0.s` !



- `reg.txt` is copied from "General Registers" in the left.
  `ans_reg.txt` is copied from the same place after execution.
- `text.txt` is copied from "User Text Segment" in the middle. (Make sure you check the "Instruction value" box)
- `data.txt` is copied from "User Data Segment" in the right.
  `ans_data.txt` is copied from the same place after execution.

Important notes:

- In your assembly code, `.data` should be followed by `0x10008000` and `.text` should be followed by `0x00400000`
  Because JsSpim allocates memory segments different to our lab.
- The `.text` should contains at least 9 instructions to cover out default codes in JsSpim.

- **[Very important]** Make sure you use the Lab 2 version:
  https://kevinshlo.github.io/JsSpim/Lab2.html
- Ignore the assemble files in top-left box (e. g. `fibonacci.s`). They won't work.
- The provided `0.s` contains `j` and `li` instructions. Be aware while testing.

> **[Disclaimer]** JsSpim is not developed by TA team, we forked and modified it to fit our lab.
> It is open sourced on github by ShawnZhong.
> Please do not spam issues. If you like it, give it a star.

Format Script

Python script `format.py` can help you format the copied `.txt` file into `.mem` file which testbench can read.
Check usage in the script.

Example for formating the provided test case:

```
cd testbench            # enter directory where *.txt files at
python3 format.py 0 0   # format 0.reg.txt ... into 0.reg.mem ...
```

> requires Python 3.6 or newer

Testbench

`tb_single_cycle.sv` is similar to testbench in Lab 1.
It reads `.mem` files as test case and automatically test the processor.
Make sure `.mem` files are named as `<number>.reg.mem` etc., and added into your Vivado project.