Computer Organization Lab 3

Single-cycle CPU With Memory Access And Jump

Lab Deadline: 6/1 (Mon.) 23:55

Demo Date: TBA

I. Introduction

In this lab, you need to add the ability to access memory and J-type instructions based on the CPU created in the previous lab.

II. Requirements

(a) Environments & Prerequisites

- You and your teammate cooperatively complete this lab. But both of you need to know every detail, including the part handled by your partner.
- Source code should be able to be compiled by Icarus Verilog and generate waveform using GTKWave.
- Data_Memory.v, Reg_File.v, Test_Bench.v provided are fully functional modules. Feel free to modify them to test your program.
- All your designs should be implemented as combinational circuits.
- Your CPU needs to support all the instructions specified in the previous lab.
- Prepend your student ID and name at the top of each source code file. The format should follow the guideline provided below.

```
// Author: 0123456 連花鴿, 0654321 郜基殿

module CPU(
// The rest of the file is omitted
```

(b) R-type Instruction Set

R-type instructions are identified by an opcode of 0. They can be differentiated by their funct values.

31:26	25:21	20:16	15:11	10:6	5:0
opcode (0)	rs	rt	rd	shamt	funct

The following instructions are required in this lab.

Instruction Name	Instruction	Pseudo Code	funct
Shift left logical	sll rd, rt, shamt	rd = rt << shamt	000 000
Signed multiplication	mul rd, rs, rt	rd = (rs * rt)(LSB 32 bit)	011 000
Jump register	jr rs	pc = rs	001 000

There is a difference between shift instructions and other arithmetic operations in R-type instructions. For shift related assembly it is "op $rd, $rt, $rs", and "op $rd, $rs, $rt" for other arithmetic operations. Where the sequence of $rs and $rt is swapped. Make sure you do not fall into this pitfall.

nop (no operation) means CPU does nothing when taking this instruction. In MIPS, this can be done by "**sll r0, r0, 0"**, which is an instruction that has 32 zeroes and has the meaning of "shift left 0 bit on register 0 and save to register 0". Thus, you should implement **sll** first as the CPU will continue to execute **nop** until the simulation ends.

Because MIPS is a 32-bit architecture, multiplying two 32-bit number s would result in a 64-bit number. In order to solve this issue, MIPS introduces two registers, HI and LO to store the upper and lower 32 bits of multiplication result. For simplicity, we hijack the opcode for **mult** and call it **mul** in this lab. You only need to store the lower 32 bits of multiplication result in $rd register. Also, during the evaluation, there will not be cases that the multiplication result exceeds 32 bits.

(c) I-type Instruction Set

I-type instructions can be identified by opcode with any number greater than 3. All of these instructions feature a 16-bit immediate, which is sign-extended to a 32-bit value in every instruction except for the and, or, and xor instructions which are zero-extended.

31:26	25:21	20:16	15:0
opcode	rs	rt	imm

The following instructions are required in this lab.

Instruction Name	Instruction	Pseudo Code	opcode
Load word	lw rt, imm(rs)	rt = *(4 byte)(rs + imm(sign ext.))	100 011
Save word	sw rt, imm(rs)	*(4 byte)(rs + imm(sign ext.)) = rt	101 011
Branch if less or equal to 0	blez rs, imm	if(rs <= 0) PC += 4 + (imm << 2)(sign ext.)	000 110
Branch if greater than 0	bgtz rs, imm	if(rs > 0) PC += 4 + (imm << 2)(sign ext.)	000 111

(c) J-type Instruction Set

These instructions are identified and differentiated by their opcode numbers (2 and 3).

Instruction Name	Instruction	Pseudo Code	opcode
Jump	j addr	PC = {(PC + 4)[31:28], addr, 2b'00}	000 010
Jump and link	jal addr	r31($ra) = PC + 8 PC = {(PC + 4)[31:28], addr, 2b'00}	000 011

(d) Report

Each person needs to submit a report in Chinese or English with the following contents:

- Architecture diagram: Draw your own copy of the diagram, do not copy any existing diagram (including the ones you find on the internet or from your teammate). You should revise the one you made for lab2.
- Implementation details: What modifications are made compare to lab2?
- Questions: Please answer the following questions in your report
  - Please list the machine code of each branch/jump (**j**, **bne**, **beq** and **bgtz**) instructions in "_CO_Lab3_test_data_bubble_sort.txt" on the report.
  - What is the main purpose of "**jal**" and "**jr**" instructions? Please write down a simple program which explains the scenario.
  - Assume there is an instruction "**beqi $rt, imm, offset**" (branch if equal to immediate). Although it is not supported by our CPU, it can be achieved by the combination of two instructions "**ori $ra, $zero, imm**" and "**beq $rt, $ra, offset**". Can the instruction "**bge $rs, $rt, offset**" (branch if greater than or equal, branch if $rs >= $rt) be replaced with instructions implemented in lab2/3? (Hint: slt)
  - Following the previous question, can we reduce some of the instructions in lab2/3 without reducing the capability of CPU? If possible, what is the minimum instruction set, and what are the advantages and disadvantages of this reduction?
- Contribution
  - If you finish the lab on your own. State "I finish all requirements on my own" in this section.
  - Otherwise, state the individual contribution in this lab. E.g. ALU module, test case generation, bug fixing, etc.
  - You can make a complaint if your teammate barely does anything.
- Discussions, problem encountered and miscellaneous
  - Anything you want to share, suggestions, etc.

III. Testing

- Refer to lab2 specification for compile and testing commands.
- You can change the test data you want to load with the same method in lab2.
- You should check if all the instructions required in lab2 yield the same results. As we will use some of those instructions to evaluate your program. And you might accidentally produce bugs in the process of modifying your program.
- The test data provided perform basic unit test, which checks one type of instruction only. You should create your own test data to test your design thoroughly. There will be hidden test data when TAs evaluate your design.
- Because the simulation will not stop until specified cycles (defined in Test_Bench.v END_COUNT). But now you have implemented **nop** (**sll** to be exact). You don't need to adjust END_COUNT any more.
- In this lab, other than the registers that will be inspected in lab2 (r0 ~ r11) during the simulation. There will be two more registers, r29($sp) and r31($ra) to be examined. $sp has been initialized to 128 and $ra to 0.
- To test **lw** and **sw** instruction. Memory has been initialized with different values. Please refer to Data_Memory.v for more details.
- Make sure your module can be tested with Test_Bench.v provided by us, or we cannot test your code.

IV. Grading

- Plagiarism is strictly prohibited. Including the source code you find on the Internet or inherited from others. Same applies to your report. You will get 0 points in this lab if you choose to do so. Despite that, we still encourage you to discuss with your classmates.
- Testing will be done on Ubuntu 18.04 with Icarus Verilog 10.2
- You will get points after you complete the following required items
  - CPU instructions (85 %)

- R-type (16 %)
  - sll (4 %)
  - mul (4 %)
  - jr (8 %)
- I-type (42 %)
  - lw (13 %)
  - sw (13 %)
  - blez (8 %)
  - bgtz (8 %)
- J-type (16 %)
  - j (8 %)
  - jal (8 %)
- Bubble sort (5 %)
  - The test case is "_CO_Lab3_test_data_bubble_sort.txt".You have to convert the assembly code to machine code and test it on your own, but you don't have to submit the machine code.
- Hidden test cases (6 %)
- Report (35 %)
  - Architecture diagram (5 %)
  - Implementation details (5 %)
  - Questions (20 %)
  - Contribution (0 %, but we will evaluate your overall performance based on this)
  - Discussions (5 %)

V. Submission

- You need to compress the following files into a zip file and name it "<YourID>_<TeammateID>.zip", "<YourID>.zip" if you finish the lab on your own. Other archive formats are not acceptable.
  - e.g. "0123456_0654321.zip", for you and "0654321_0123456.zip" for your teammate. "0123456.zip" if you go solo.

  - All source code (*.v) excluding ProgramCounter.v, Instr_Memory.v, Reg_File.v, Test_Bench.v, Data_Memory.v.
    - Both you and your teammate need to submit a copy of your source code in the zip file.
  - Do not include .vcd, .vvp files.
  - Do not put files inside a folder then archive it. Compress all the files directly.
  - A report named "<YourID>.pdf" e.g. "0123456.pdf".
    - Reminder: You need to submit a report written by yourself. Even you finish this lab as a team.
- If you do not follow the policy above, you will get up to 10 points penalty.
- Upload the zip file to New E3 before 6/1 (Mon.) 23:55.
- You will get 15 points penalty per day for late submission. Any submission will be rejected after 6/4 (Thu.) 23:55.

VI. Contacts & Discussions

Please contact 許賀傑 (hchsu0426@cs.nctu.edu.tw) and 周煥然 (kulugu2@gmail.com) via New E3 if you have any problems.

Head to KCW's COVID (Computer Organization Via Interactive Distance-learning) Microsoft Teams, channel #lab3_discussions for open discussions.

VII. Hints & Suggestions

- Similar to other compilers, iverilog does not produce all the warning messages by default. You can add "-Wall" to force the compiler to print all the warning messages. Refer to https://iverilog.fandom.com/wiki/Iverilog_Flags for more info.
- There are memory access instructions and jump instructions in this lab, you may need to add wire to your opcode decoder. Refer to the diagram below for more info.

- Each ISA has its calling convention. For MIPS you can refer to https://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventionsSummary.pdf for a detailed tour. This is not covered in this course but we still suggest you give a read. However, we will still mimic function call in our test case. You can take a look at "_CO_Lab3_test_data_function_call.txt" for a simple example with explanation.
- Fun fact. In x86, you can use "**mov**" instruction (which can move values between registers or memory) to replace all other instructions, including arithmetic operation, comparison, jump. You can take a look at the code and example here https://github.com/xoreaxeaxeax/movfuscator
- The figure below is an architecture diagram for your reference.