CSC3050 Project 3 Report

Tianci Hou 121090184

★ I have solved all the hazards and passed all the testcases.

1 Project Description

This is my implementation for the course project 3 of CSC3050, 2023 Spring, CUHK(SZ). In this project, we are required to write a simple ALU and a 5-stage Pipelined CPU.

2 Set Up Instruction

2.1 Prerequisites

- iverilog
- Make >= 4.2.1

2.2 Compile the project

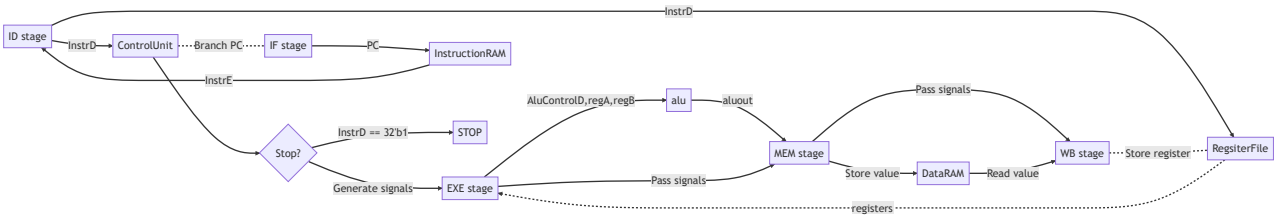
```
$ cd /path/to/the/project/src/alu
$ make compile
$ cd /path/to/the/project/src/cpu
$ make compile
```

3 Project Structure and Process

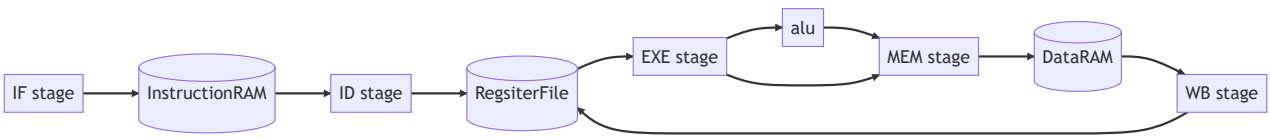
NOTE: In order to make the report look more concise, the complete schematic diagram is placed on the last page, and the flowcharts below only contain some key information.

3.1 Data flow and overall structure

The overall flow chart of this project:



The data flow about the 32 registers and RAMs:



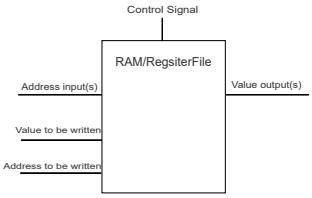
Source files:

- **cpu.v:** Top design for linking all individual modules.
- **global_const.v:** This file defined all the global constants especially for control signals in MUX.
- **mux4.v mux5bit.v:** The MUX is modularized because the design has many select statements.
- **alu.v:** An alu with control signals.

- **ControlUnit.v:** The control unit. Decode the instruction and generate control signals.
- **HazardUnit.v:** Deal with data hazards by using forwarding and stalling.
- **InstructionRAM.v:** The memory stored all the instructions. It is a read-only memory.
- **RegsiterFile.v:** The memory stored the 32 registers.
- **DataRAM.v:** The main memory for **sw** instruction to store the value.
- **PipelineID.v PipelineIF.v PipelineEXE.v PipelineMEM.v PipelineWD.v:** 5-staged registers. Just for one cycle of delay and can be reset and enabled by the control signals.

3.2 RAM and register file

I did not use the provided RAM.v and implemented them myself. All the RAMs and Register File modules follow the schematic diagram below:

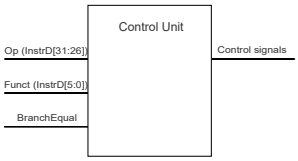


To ensure that the instruction can be completed in 5 cycles. All output operations use combinational logic, in other words, asynchronous. The MUX is used directly to find the value of the corresponding address from memory and output it.

For write operations, we perform them at the **failing edge** of the CLK to avoid glitch.

3.3 Control unit

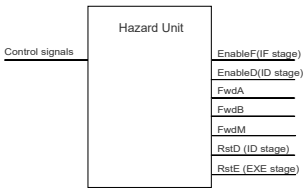
The control unit is like the brain of the CPU, mainly responsible for decoding instruction and transmitting the corresponding control signals to other modules. The schematic diagram of the control unit I implemented is as follows:



The control signal is a combination of logic. Unlike the diagram on the textbook, I have an extra input in the control unit, BranchEqual, to solve the beq and bne instructions. BranchEqual will be HIGH when RD1 == RD2.

3.4 Hazard unit

I have integrated Hazard unit. The schematic diagram is as follows:



I use both data forwarding and stalling. FwdA and FwdB are the control signals to decide which registers are passed into regA and regB of alu. Rst and Enable are the control signals for stage registers.

3.5 alu

A simple alu in the question 1 can be seen as a combination of control unit and alu in the cpu. No additional explanation about the behavior of simple alu will be given here.

Alu has three inputs regA, regB and alu control signal.

One trick is that in the cpu, we can combine many control signals. For example, **add, addu, addi, addiu, sw, jal** all need to perform additive operations. This is decided when the control unit passes the signals, which has the advantage of reducing the redundant originals.

Also because we do not need to deal with errors such as overflow in the second question, the signals of flags can be simply ignored.

The complete schematic diagram of alu is given on the last page.

3.6 Sign extend

In fact sign extend is not be implemented as a independent module. But there is something wrong in the diagram on the text book so I would like mention here. Sign extension also requires signals to determine whether to extend to a signed number or an unsigned number.

According to the instructions in the assignments, all simple bit operations (**andi, ori, xori**) do not need to be extended to signed numbers, in other words, {16'b0,imm} be the new number. However, those unisgned operations require extending the number to a signed number before performing the operation, such as **addiu**.

3.7 Stage registers

As stated earlier in the report, the stage register modules I implemented only have the effect of delaying one cycle, they do not support any arithmetic. They will do the passing operation at the **rising edge** of the CLK.

4 Checker and Results

I wrote a simple shell script to automatically check the results.

More specifically, a new directory called **custom_test** needs to be created in the root directory and put the shell script in it, in order to reduce the possibility of damage to the source program and source test cases.

```
#!/bin/bash
cd ../src/cpu
make compile
cd ../../custom_test
cp ../src/cpu/CPU ./CPU
for i in $(seq 1 8)
do
echo testcase:$i
cp machine_code$i.txt CPU_instruction.bin
vvp CPU -l CPU.log
diff --strip-trailing-cr data.bin DATA_RAM$i.txt
done
```

To illustrate the efficiency of my design. I added the integer CLK_cnt in **DataRAM.v** to count the cycles. (They were commented out in the submitted code.)

```
integer CLK_cnt;
initial begin
    CLK_cnt = 0;
end
always @(posedge CLK) begin
    CLK_cnt = CLK_cnt + 1;
end
```

The results (removing iverilog warnings) are:

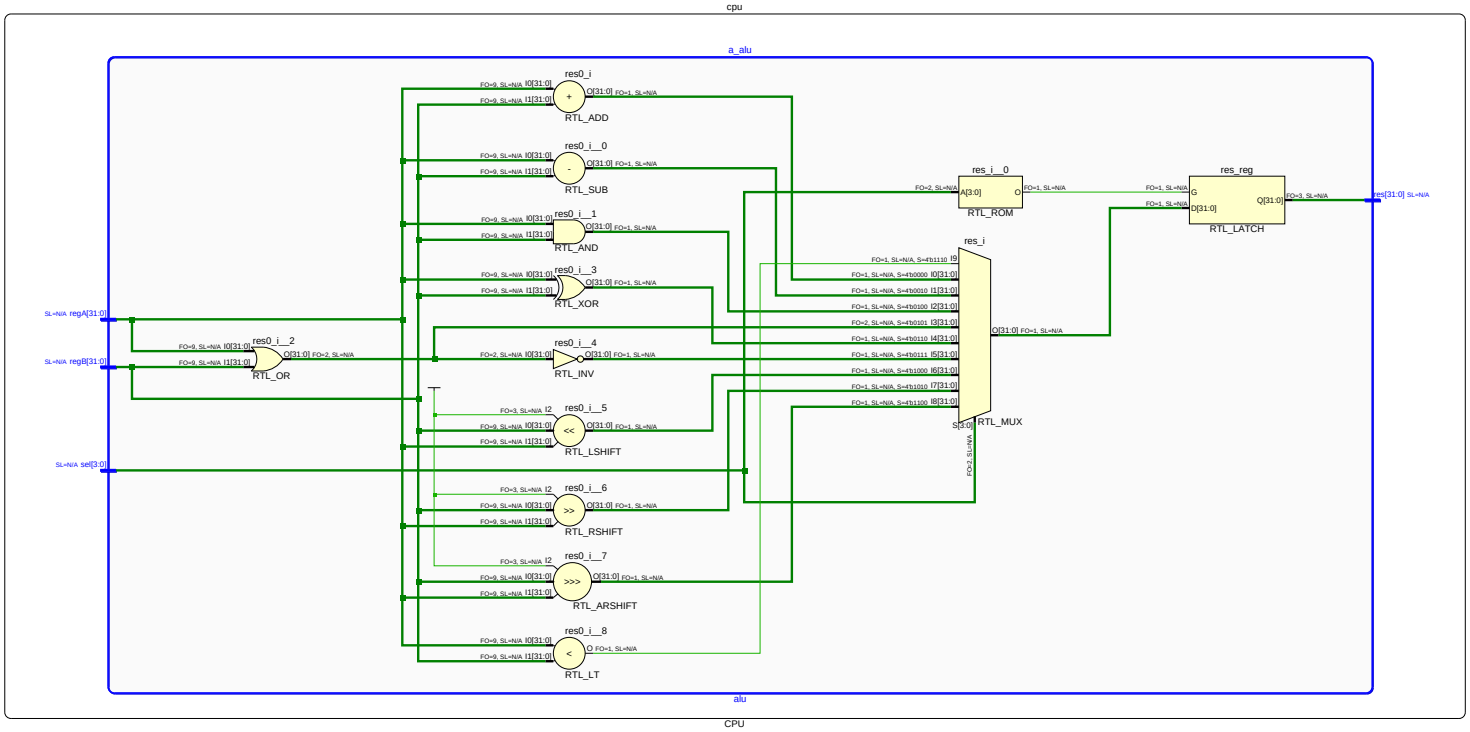
```
 bash checker.sh
iverilog -o CPU cpu.v test_cpu.v;
testcase:1
CLK_cnt =          55
testcase:2
CLK_cnt =          14
testcase:3
CLK_cnt =          17
testcase:4
CLK_cnt =          16
testcase:5
CLK_cnt =         178
testcase:6
CLK_cnt =          53
testcase:7
CLK_cnt =          44
testcase:8
CLK_cnt =          27
```

The results show that my program have passed all testcases.

5 Complete Schematic Diagrams

NOTE: These images are vector images that can be zoomed in to see details.

5.1 ALU



5.2 CPU