The Chinese University of Hong Kong, Shenzhen

CSC3050

Computer Architecture

Assignment 4 Report

Author: Yang Yuzhe
Student ID: 121090684

December 14, 2023

1 Overview

In project 4, we are required to implement a 5-stage pipelined CPU using Verilog language which can execute the MIPS instructions. We are provided a template and we are expected to implement `Hazard.v IF.v ID.v EX.v MEN.v WB.v`. For testing, we have totally 8 test files, which have different types of hazard existing in each test file.
This project, with the provided template, is like having the framework of the entire CPU circuit already set up. All I need to do is implement specific features like hazards.

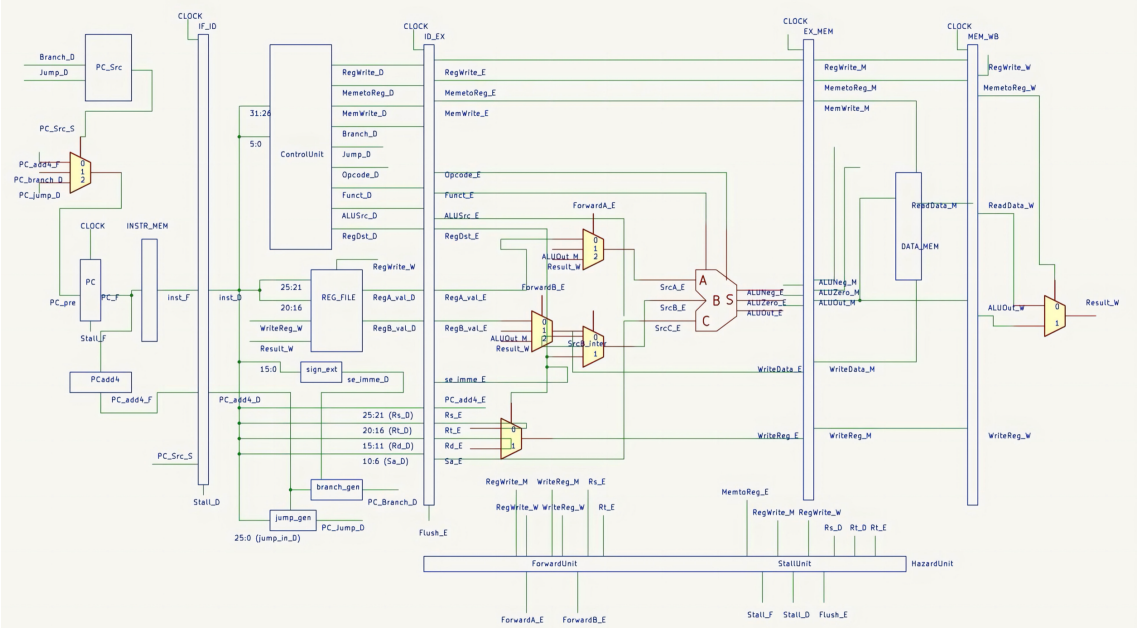

Figure 1: the Structure of the 5-stage CPU

2 Implementation

2.1 EX MEM WB

In this section, since the template has already established the overall framework, you only need to set the input and output signals. It's worth noting in Verilog the distinction between = and <=; the former is used for combinational logic assignments, while the latter is typically used in sequential logic for flip-flop assignments.
In the implementation of the ALU in the EX stage, it is generally similar to the implementation of the ALU in Project 3. However, SrcA and SrcB are assumed to be 32-bit by default, and when dealing with a 16-bit immediate, they should be zero-extended to accommodate it.

2.2 IF

In the IF stage, the implementation involves incorporating multiple MUX to manage various data paths. These multiplexers are crucial for selecting the appropriate input

for each stage of the pipeline. The specific configuration and inputs of these MUXs depend on the design requirements and the architecture of the processor. They are typically utilized to control the flow of instructions and manage the selection of addresses or data sources during instruction fetch.

2.3 ID

Firstly, since I haven't addressed the jal instruction in the hazard section, I need to set the correct address for $ra in the REG_FILE.

```
always@(Jump_D, Opcode_D) begin
    if (Jump_D == 1'b1 && Opcode_D == 6'b000011) // jal
    begin
        simu_register[31] <= PC_pre - 4;
    end
end
```

Control Unit

In the `CONTROL_UNIT` module, we need to configure various lines based on the input `opcode_in` and `funct_in`. These lines will determine the control signals in the output.

`IF_ID`

When the Stall signal is set to 1, the pipeline pauses, waiting for the current stage's instruction to complete. The Flush signal is used to control whether a pipeline flush is necessary. When the Flush signal is set to 1, the pipeline is cleared, and the current stage's instruction is discarded.
When the Stall signal is set to 0 (indicating no need to pause the pipeline) and the Flush signal is set to 0 (indicating no need to flush the pipeline), the input signals inst_in and PC_add4_in are assigned to their corresponding output signals inst_out and PC_add4_out. This signifies that under normal circumstances, the values of the instruction and PC are passed on to the next stage.
When the Flush signal is set to 1 (indicating a need to flush the pipeline), the output signals inst_out and PC_add4_out are reset to 0. This indicates that when a pipeline flush is required, the values of the instruction and PC are discarded, and the pipeline is reset to a known state.

2.4 Hazard

Forward D

The Forward_D function is employed to check whether an instruction in the execute stage (E), memory stage (M), or write-back stage (W) is currently writing to the register R_E. If such an instruction exists and the register number being written is non-zero, forwarding is required, and Forward_D is set to 1. Otherwise, Forward_D is set to 0.

```
if ((RegWrite_E && WriteReg_E == R_E && (WriteReg_E != 0)) ||
    (RegWrite_M && WriteReg_M == R_E && (WriteReg_M != 0)) ||
    (RegWrite_W && WriteReg_W == R_E && (WriteReg_W != 0))) begin
```

```
    Forward_D = 1'b1;
end else begin
    Forward_D = 1'b0;
end
```

Forward E

The Forward_E function is utilized to examine whether an instruction in the memory stage (M) or write-back stage (W) is writing to the register R_E. If an instruction is writing in the memory stage and the register number being written is non-zero, Forward_E is set to 01. If an instruction is writing in the write-back stage and the register number being written is non-zero, Forward_E is set to 10. Otherwise, Forward_E is set to 00.

```
if(RegWrite_M && (WriteReg_M != 0) && (WriteReg_M == R_E)) begin
    Forward_E = 2'b01;
end else if(RegWrite_W && (WriteReg_W != 0) && (WriteReg_W == R_E))
   begin
    Forward_E = 2'b10;
end else begin
    Forward_E = 2'b00;
end
```

Stall

The Stall unit in a processor is designed to manage signals dictating when to temporarily halt the pipeline execution, addressing issues like data hazards and control hazards.
In the MIPS architecture, the jr and branch instructions alter the control flow of the program during execution. This means that after executing these instructions, the next instruction to be executed may not be the immediate successor in the program. Therefore, when the CPU encounters these instructions, they need to be executed promptly rather than waiting for other instructions.
For jr and branch instructions, as they alter the program's control flow, they need to be executed immediately and cannot wait for other instructions. Hence, when these instructions are encountered, the Stall signal is set to 0 to ensure that the pipeline does not stall and can promptly execute these instructions.
For the other instruction, stall signal should be detected as following conditions:

```
if ((MemtoReg_E && (WriteReg_E == Rs_D || WriteReg_E == Rt_D)) ||
    (MemtoReg_M && (WriteReg_M == Rs_D || WriteReg_M == Rt_D)) ||
    (MemtoReg_W && (WriteReg_W == Rs_D || WriteReg_W == Rt_D))) begin
```

3 Testing & Debugging

For debugging the code, I use the VCD dump file to show the waveform of the CPU. The VCD dump file is generated by the following code.

```
initial begin
    $dumpfile("cpu_test.vcd");
    $dumpvars(0, cpu_test);
end
```

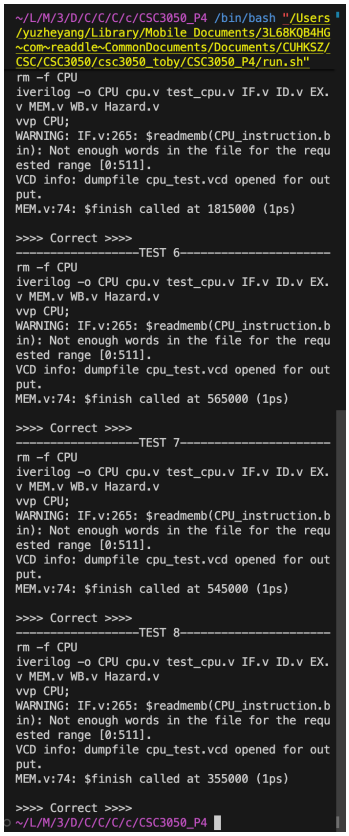

Figure 2: Output result

For testing the code, I firstly write `convert.py` in order to convert the binary file to text file and vice versa. Then I write `cmp.py` to compare the output of the CPU with the expected output. If the output is wrong, the program will show the wrong `DATA_MEM`.

Then I write a shellscript to test the CPU. The shellscript is shown below.

```
#!/bin/bash

# cd CSC3050_P4 || exit
for i in {1..8}; do
    echo "----------TEST ${i}----------"
    python convert.py cpu_test/machine_code${i}.txt CPU_instruction.bin
    t2b

    make clean
    make test

    echo ""
    python convert.py data.bin data.out b2t
    python cmp.py data.out cpu_test/DATA_RAM${i}.txt
done
```

To test the code, you need to run the following command.

```
bash run.sh
```

The part of the output result is shown in figure2.