Lab Report

Experiment: **Pipeline CPU Lab**

Department： **Electrical Engineering**

Name: **Xinyue Ma 20170765**

Pengfei Gao 20206017

I INTRODUCTION

The central processing unit (CPU), as the computing and control core of a computer system, is the final execution unit for information processing and program operation. In this lab, we design a Pipeline CPU based on Verilog.

RISC-V is an open instruction set architecture (ISA) based on Reduced Instruction Set Computing (RISC). We aim to implement about 30 instructions of RISC-V in this lab. Compared with single-cycle CPU and multi-cycle CPU, each instruction utilizes difference part of CPU in pipeline CPU. If the pipeline stages are not stalled, the five-stage pipeline CPU completes one instruction per every clock cycle, hence the throughput is higher.

II DESIGN

In this lab, memory model, register file and clock signal is given, thus we just should implement other parts of CPU. To implement the five-stage pipeline CPU, we use a bunch of additional pipeline registers to forward the control signals and data that are needed in each pipeline stage.

2.1 Data Path

2.1.1 ALU

ALU part should implement the different calculation function of two inputs according to the given instruction. There are total 16 kinds of instructions ALU should deal with.

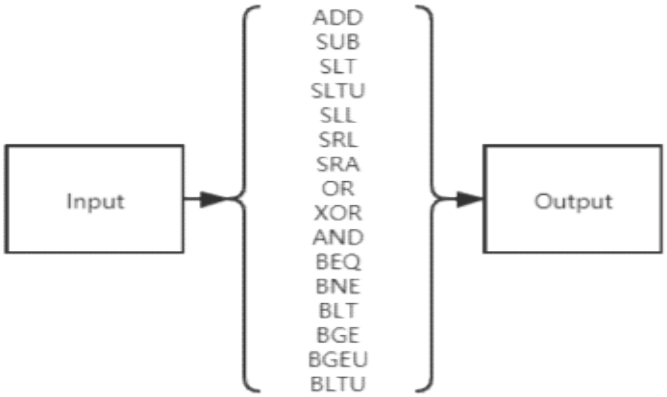

Figure 2.1 flow chart of ALU

2.1.2 IMM processing

Because different instruction format has different immediate subfields. We use IMM processing to combine different parts of immediate according to their types for ALU operation.

2.1.3 Data Hazard Detection

We use forwarding unit to bypass the results to the following instruction if it is necessary. We set several registers to detect the occurrence of data hazard. If data hazard happens, the ALU will choose the ALU result of the previous instruction, if the instruction is Load, ALU will choose the result of the previous data memory.

2.2 Control Path

Every instruction will go through 5 stages in this CPU. In IF stage, according to the instruction address in the program counter (PC), we take the instruction out from memory, and PC will automatically increase to gain the next instruction address. Due to the fact that we can not get the PC value in time, we use the “always-not-taken” branch predictor, assume the next PC value is always PC+4, if the instruction is branch or jump, we will stall and move to return address/jump address. In ID stage, we analyze and decode the instruction obtained in the fetch instruction stage, determine the operation that this instruction needs to complete. In EX stage, the instruction action will be executed concretely according to the operation control signal. In MEM stage, only LW, SW instructions can reach this stage, we write the data to the storage location specified by the data address in memory, or retrieves the data from the storage location. Finally, in WB stage, the result of an instruction execution or data from the access memory will be written back to the appropriate destination register.

In addition, we use pipeline flow part to control registers when the stages transform. Normally we set all output registers equal to their corresponding input registers. However, when IF stage transforms to ID stage or ID stage transforms to EX stage, if the instruction is branch, we should stall the pipeline and reset all corresponding registers.

III IMPLEMENTATION

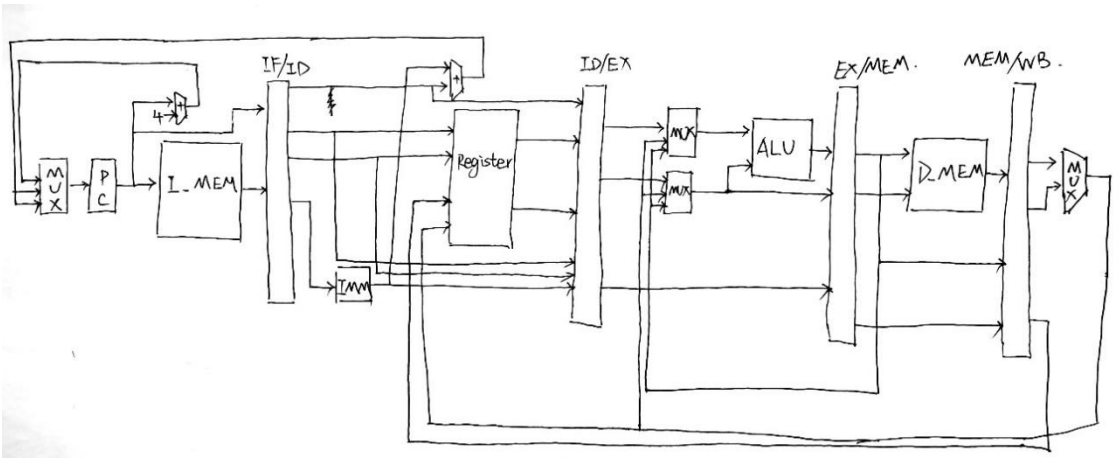

Figure 3.1 Overall design figure (not include control path)

IV EVALUATION

After finishing the code, we use the given Testbench files to evaluate the effectiveness. We used ModelSim to simulate our project, and according to the wave figure, all tests are pass.

```
# Finish:      10822 cycle
# Success.
# ** Note: $finish    : D:/Academics/kaist/ 2020 Spring/EE312/Assignments/Assignr
```

Figure 5.1 Results of the Test (Sort)

```
# Finish:         86 cycle
# Success.
# ** Note: $finish    : D:/Academics/kaist/ 2020 Spring/EE312/Assignments/Assignmer
```

Figure 5.2 Results of the Test (For loop)

```
# Test #    1 has been passed
# Test #    2 has been passed
# Test #    3 has been passed
# Test #    4 has been passed
# Test #    5 has been passed
# Test #    6 has been passed
# Test #    7 has been passed
# Test #    8 has been passed
# Test #    9 has been passed
# Test #   10 has been passed
# Test #   11 has been passed
# Test #   12 has been passed
# Test #   13 has been passed
# Test #   14 has been passed
# Test #   15 has been passed
# Test #   16 has been passed
# Test #   17 has been passed
```

Figure 5.3 Results of the Test (Inst)

V DISCUSSION

In this lab, we use “always-not-taken” predictor to implement the branch prediction. In the future we can use always-taken” branch prediction with the BTB or “2-bit saturation counter” with the BTB to get a better performance.

VI CONCLUSION

In this lab, we make a VISC-V Pipeline CPU based on Verilog. After simulation, every given test was passed. Thus, the lab achieves the expected result successfully.