Design Document

Overview

In this project, a 5-stage pipelined CPU is implemented using Verilog language The five stages are respectively IF, ID, EX, MEM, WB. And four pipeline register modules are used to carry data and control signals in the previous stage. At one time, all stages will have their execution instruction. In stage IF, the CPU read one instruction in the instruction memory with the new given pc address. In the stage ID, the processor will divide the instruction into different parts and decode the MIPS instruction with the registers and control unit. The operation code and function code in MIPS instruction is sent to the control unit, which recognize the type of instruction and initialize the control signals. In the stage EX, ALU will handle serval instructions, which include arithmetic, logical, shifting, branch, and so on. In the stage MEM, data will be fetched from or stored to the data memory by data transfer instructions. In the stage WB, data will be written back to registers if needed. Thus, the CPU can execute several instructions at the same time, which will greatly increase the execution speed.

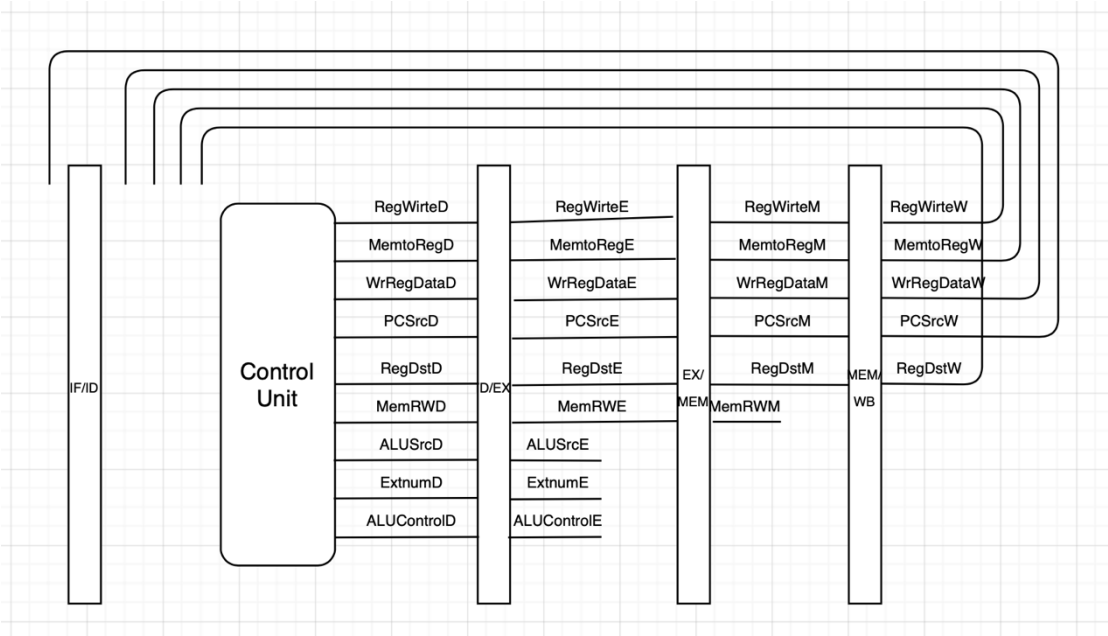
In the design of my CPU, PipeReg_IFID, PipeReg_IDEX, PipeReg_EXMEM, PipeReg_MEMWB modules are corresponding pipeline register modules. PC module is to count the pc address. PCAddr is to pre-execute the jal and j jump address before sending it to PC module. RegFile module stores all data stored in the registers, where CPU can read or write data in registers. ALU module is designed to handle arithmetic Logical. ControlUnit module is to initialize the control signals by recognizing the instructions. NumExten module is to operate a sign number extension on some data that will be put into ALU. For solving hazards, there're some addition modules. ForwardingUnit module can correctly handle handle MEM_to_EX and WB_to_EX data hazards. HazardDetection module can handle the hazard about lw stall. The design details are illustrated below.

Module Spec

1. Control Unit

1.1 The basic control signals flow chart

Table 1.1.1 Control Signals Flow Chart



1.2 The Functions of Control Signals

<1> RegWrite: it controls the process of writing data into registers. “0” for sw instruction. “1” for other instructions.

<2> MemtoReg: it selects the data between data got from memory and ALU results. And this data will be written back to registers. "1" for lw. “0” for others.

<3> WrRegData: it selects the data between jal target address and data after Memtoreg selecting. “0” for jal. “1” for others.

<4> PCSrc: it handles the situations that change the PC. “00” for normal situations. “10” for jr. “11” for j and jal. “01” for branch instructions.

<5> RegDst: it selects the output port for some instructions. “00” for jal, which will store data in $ra. “01” for addi, addiu, andi, ori, xori, lw, which will store data into rt. “10” for add, addu, and, nor, or, sll, sllv, sra, srav, srl, srlv, sub, subu, xor, slt, where it will store data into rd.

<6> MemRW: it determines whether it’s a store or read data operation. “1” for sw: store data. “0” for lw: read data.

<7> ALUSrc: it chooses the rd or extension number as one of inputs of ALU. “1” for "1": addi, addiu, andi, ori, xori, lw, sw. “0” for others.

<8> Extnum: it performs sign number extension on several instructions. Notice that xori instruction doesn’t need sign number extension. “0” for xori. “1” for others.

<9> ALUControl: it’s used to distinguish different ALU operations. I divide all instructions into 14 situations. So, I use four bits signals to distinguish them.

2. Register File

This module stores all data stored in the registers, where CPU can read or write data in registers. And it obeys the rule “first write then read”. The registers are represented

by

```
reg [31:0] register [0:31]
```

The multiplexer in the WB stage is combined into this part for convenience. So it takes overall four control signals. They're MemtoReg, WrRegData, RegDst and RegWrite. The first three control signals are to select the data that will be written in the register file. And the fourth control signals determine whether this data needs to be written back.

3. PC, PCAddr

These two modules deal with the program counter. PCAddr module is in the IF stage. It recognizes j and jal instructions, pre-excutes the target address, and sends it to the PC module. PC module firstly gets the reset signal and then handle four situations:

<1> Normal situation: PC = PC + 1;

<2> Branch Instructions: The offset of PC is determined by the offset of branch instructions;

<3> jr jump: PC branches to an instruction address in a register;

<4> jal, j jum: PC directly jumps to the target address.

Data Flow Chart

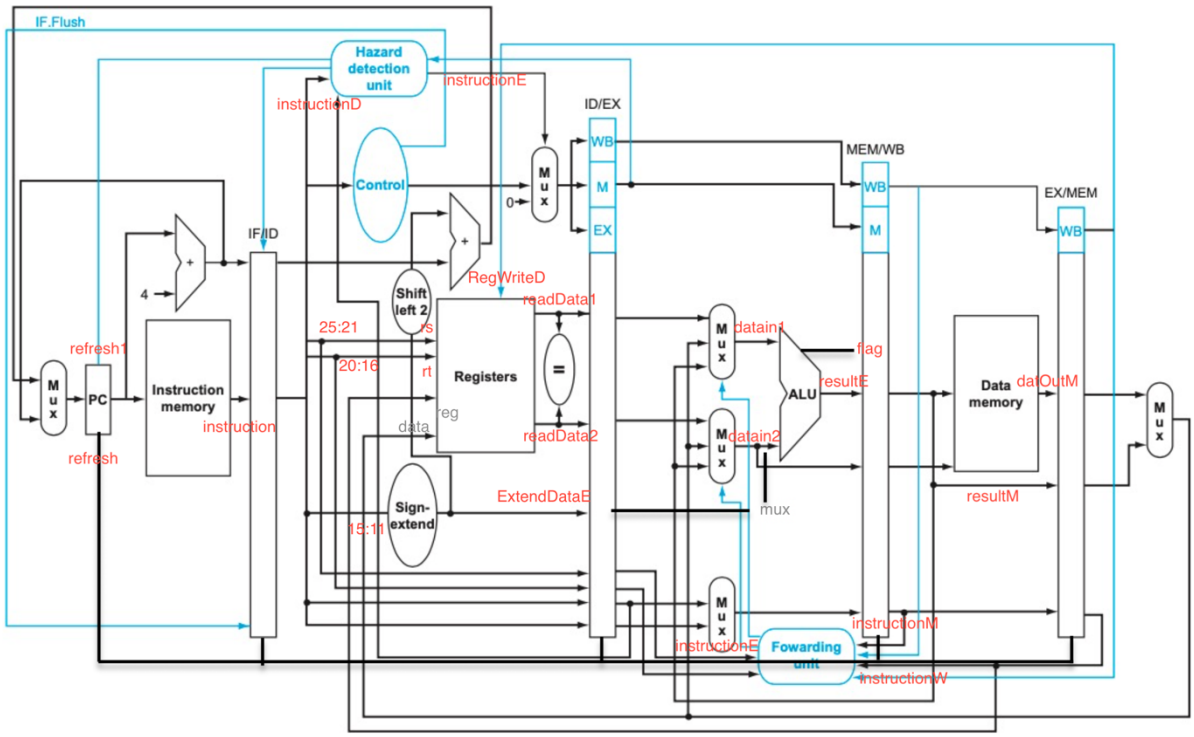


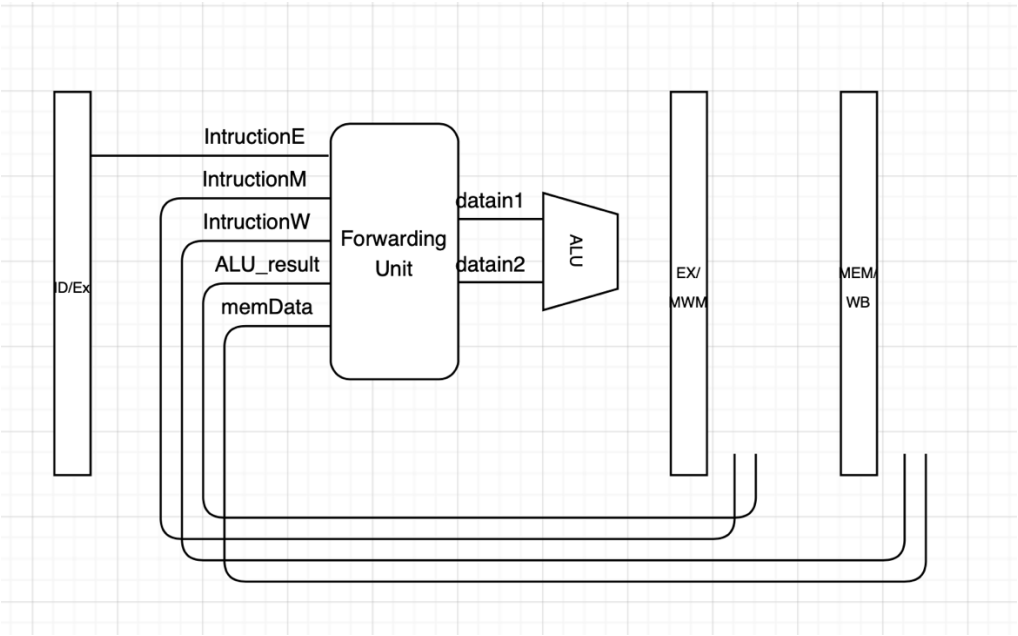
Table x. The Final Design of My CPU Notice that the hazard detection unit is moved to IF stage (after catching the instruction).

Hazards Treatment

1. ForwardingUnit

This module can correctly handle data hazards. And I combine the forwarding unit and multiplexer for convenience. The basic forwarding unit logic is shown below.

Table 1.1 Forwarding Unit Logic Chart



2. HazardDetrction

This Unit handle the data hazard of lw stall. It detects the lw instruction and judges whether there exists a data hazard. If yes, it will set the “refresh1” signal to 1. The IF/ID pipeline register will clean the data stored. And PC will stall for one cycle (to read the instruction again). Notice that I place this module in IF stage (after catching the instruction). So, the input instruction values should be “instruction” and “instructionD”.

3. Structure Hazard

To solve structural hazard, I extend each instruction execution process to five stages. And in the RegFile module, I first write data back to registers and then read data.

4. Branch Implementation

This hazard is solved by stalling for two cycles and resetting the value of PC. PC will be set to the branch target address. And signals in IF/ID and ID/EX pipeline

registers will be cleaned.

5. Jump Implementation

To reduce the CPI, J and Jal are handled in ID stage. Then it will send a flush signal to IF/ID pipeline registers to flush the instruction next to the j and jal instruction.

CPI Results

#Test	Cycles	Instructions	CPI
1	56	52	1.08
2	15	11	1.36
3	18	13	1.38
4	17	12	1.42
5	219	175	1.25
6	61	50	1.22
7	49	40	1.225
8	29	23	1.26

Additions

To run my code, you can first input “make”, then input “./cpu”. The terminal will display the complete data memory.

If you want to change the test machine code file, you can copy the test codes into intructions.bin file.

If you modify the module after the “make” operation, it will show off a notice.

```
make: `cpu' is up to date.
```

Then you can tab “make clean” and then “make” again.

And I have passed all eight tests on my computer.