Report of Five-Stage Pipelined MIPS CPU

In this project, a basic 5-stage pipelined CPU was implemented following the basic design described in the textbook.

Compilation and Test Overview

As described in the README file, basically to compile and test all the sample given, type

```
> make
> make auto -i
```

This will generate a file called `DIFFERENCE` containing the results of the `diff` command.

For more details you can refer to the README file.

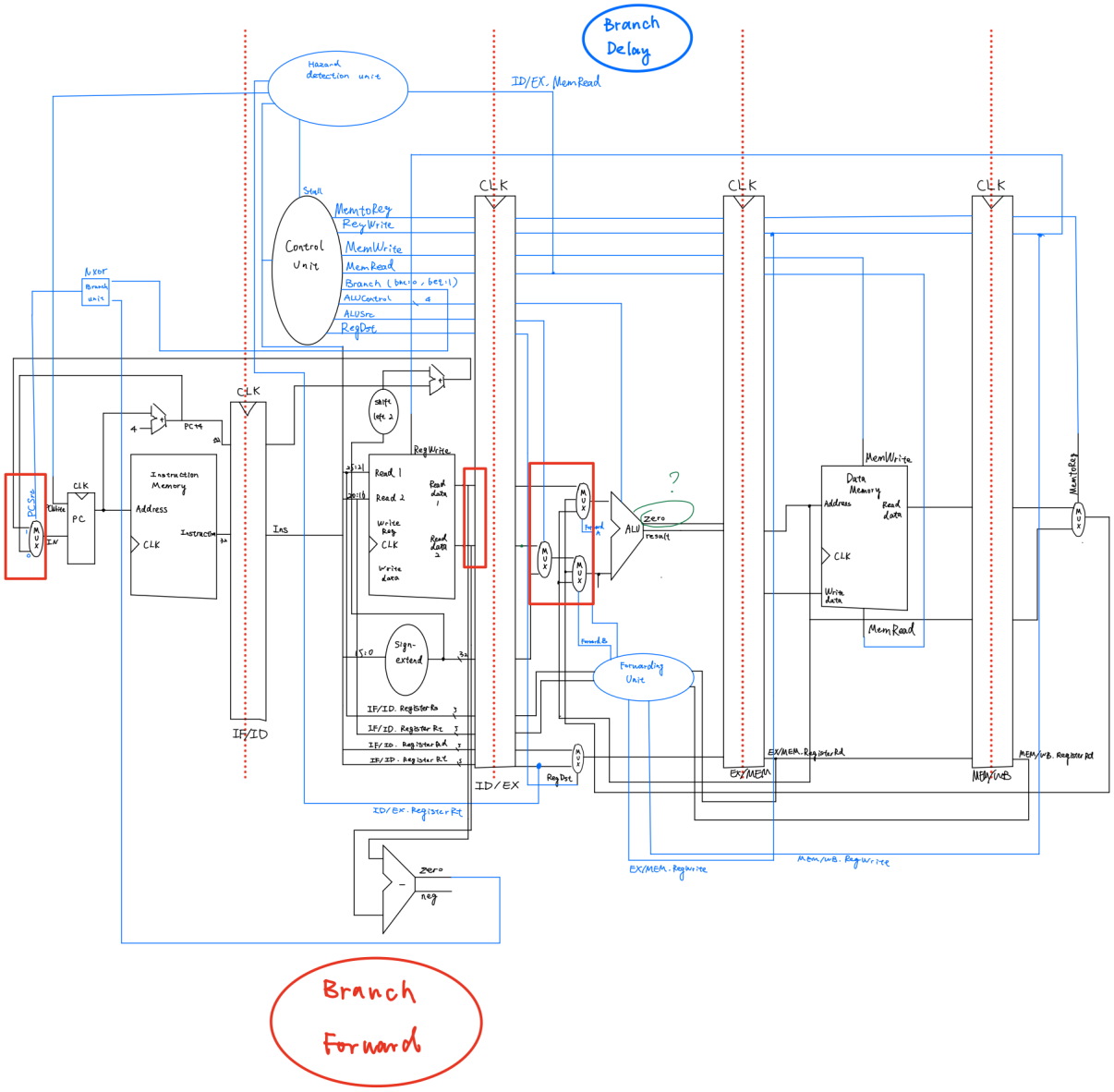
Big Thoughts

I try to use an OOP style to construct the project. Since, there are many blocks, it is natural to capsulate these bolcks in modules and instantiate them in a topper module, i.e.,

```
top_module // in the test bench
----the instance of CPU
--------the instances of function modules in CPU
```

Below is the (not that detailed) design diagram,

Detailed MIPS 5-stage CPU with forwarding unit and hazard detection unit



As the above diagram shows, in the project, I adopt the method of moving the branch part from MEM stage to ID stage to speed up the performance, which naturally generates two units, Branch Forwarding Unit and Branch Delay Unit (though j-type instructions also go through these units). Besides, there are also the basic forwarding and hazard detection units. As blocked in the red rectangles, the multiplexors in these parts are not detailed, the more detailed ones can be refered in the next section.

Implementation Details

Write and read in the register file at the same time

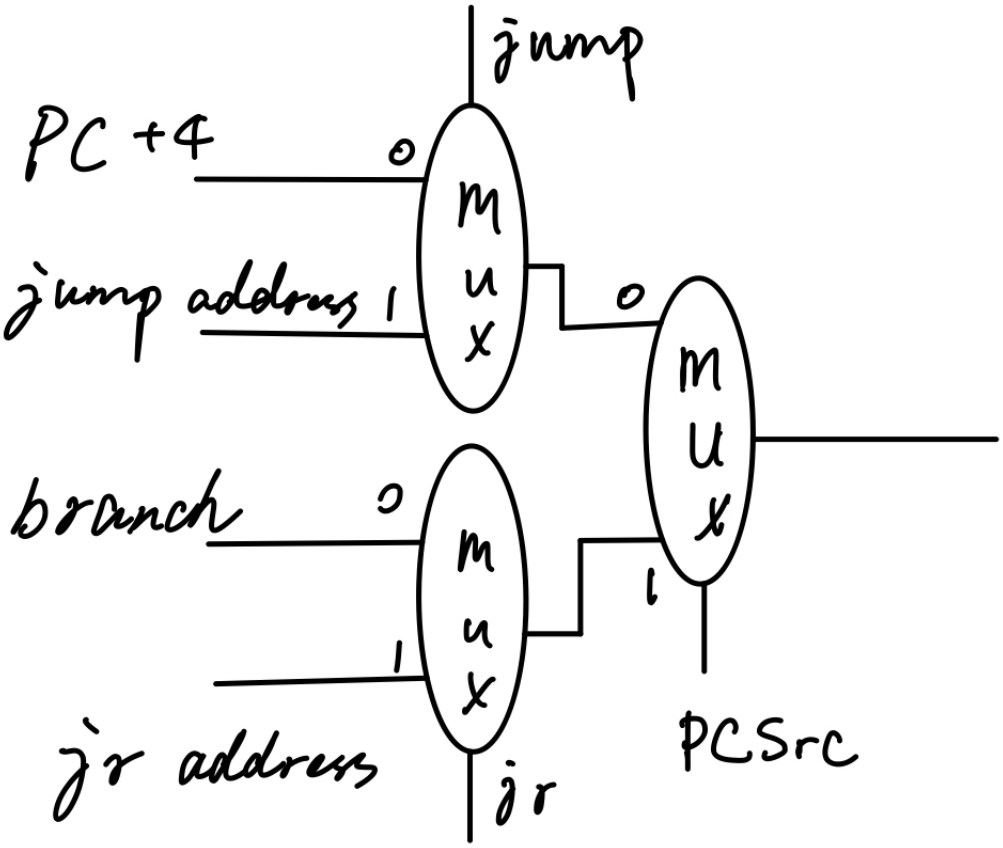
This hazard is described as the structural hazard. It can be solved by a forwarding inside the register file. Below is part of the implementation of the verilog file:

```
always @(posedge CLK) begin
        if (RegWrite) begin
            registers[writeReg] <= writeData;
```

```
        end
    end
    always @(*) begin
        if ((readReg1 == writeReg) && RegWrite) // here
            readData1 = writeData;
        else
            readData1 = registers[readReg1];
    end
    always @(*) begin
        if ((readReg2 == writeReg) && RegWrite) // and here
            readData2 = writeData;
        else
            readData2 = registers[readReg2];
    end
```

Muxes before PC

Below is a diagram illustrating the details of the multiplexors before PC:



Hazard detection

This unit detects whether a stall will happen. It can solve the following types of hazard:

```
add $1, $2, $3
# at least 1-cycle stall
beq $1, $2, address
```

```
----------------------------
lw $1, 20($2)
# at least 2-cycle stall
beq $1, $2, address
----------------------------
addi $1, $zero, 20
# at least 1-cycle stall
jr $1
----------------------------
lw $1, 20($2)
# at least 2-cycle stall
jr $1
```

The detailed logic can be referred from the source code.

ALU src details

This part is the most messy part in the CPU, since there are many different sources for ALU and the control unit has to decide to use which source. There are basically three types of souces: arithematic(from registers or immediates), load/store address calculation, and shift. The details can be referred from the `CPU.v` file.

Forwarding details

This part basically implementates the forwarding scheme for ALU. As described in the textbook, it solves the regular arithmetic types:

- 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
- 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
- 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
- 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Besides, it also supports the forwarding for store word, e.g.,

```
add $1, $2, $3
# forwarding
sw $1, 20($4)
```

Branch forward and related details

This unit is similar to the forwarding unit in the EX stage, but it has some to perform after the stall of the branch or jump instructions.

Performance Analysis

	Number of Instructions	Number of Clock Cycles	CPI
Test 1	53	56	1.057
Test 2	12	15	1.250
Test 3	14	18	1.286
Test 4	14	17	1.214
Test 5	25	179	7.160
Test 6	52	54	1.038
Test 7	45	47	1.044
Test 8	25	31	1.240

Pitfalls

The given file can be misleading

The misleading part, I think during my debugging, is that the reading of one instruction or some data in the memory needs the CLK to trigger which is conunter-intuitive. Thus, to improve the performance, I change the reading of the RAM to asychronously triggered.

The branch/jump delay slot needs handling

By default, when taking a branch or jump instruction (whether the branch is taken or not) the next instruction is automatically loaded into the IF stage after the branch goes to ID stage because at that time the branch is not taken and `PC+4` just takes place. This is actually a good thing for the compilers to effciently use these slots, but in this project, it needs handling properly when the branch or jump is indeed taken.

The solution is simple. For jump and the taken branch instructions, the program will detect: 1. whether the branch/jump will be taken; 2. if taken, then delay unit will flush the ID/EX register.

The forwarding scheme of JAL and JR

Basically, `jal` is similar to plain `j` except for the link action, and it can be reduced to

```
jal 20
----------------------------
j 20
addi $at, $zero, PC+4
```

Thus, only use some multiplexors to control the data flow of the registers and control the control unit to generate control code can achieve this instruction.

The `jr` is a R-type instruction which is different from the `j` or `jal`. To achieve this instruction, I add it to the branch part(in my design, `beq` has a branch code of `2'b01`, `bne` has a branch code of `2'b10`, not branch for `2'b0`, and `jr` for `2'b11`).