Computer Architecture 2020 Fall Project 1

TA: 蔡承佑

1. Problem Description

In this homework, you are going to extend your Homework 4 to a pipelined CPU. This CPU has 32 registers and 1KB instruction memory. It should take 32-bit binary codes as input and should do the corresponding RISC-V instructions, saving the result of arithmetic operations into the corresponding registers. Besides the instruction specified in Homework 4, you have to support `lw`, `sw`, `beq` in this Project additionally. We will examine the correctness of your implementation by dumping the value of each register and data memory after each cycle.

1.1. Load / Store Operations

In this project, you are provided a "Data_Memory" module and are required to implement `lw` and `sw` instructions. Figure 1 shows the data path after adding data memory to your Homework 4. The dashed lines are placeholders for modules in the following sections. You can just see them as real lines.

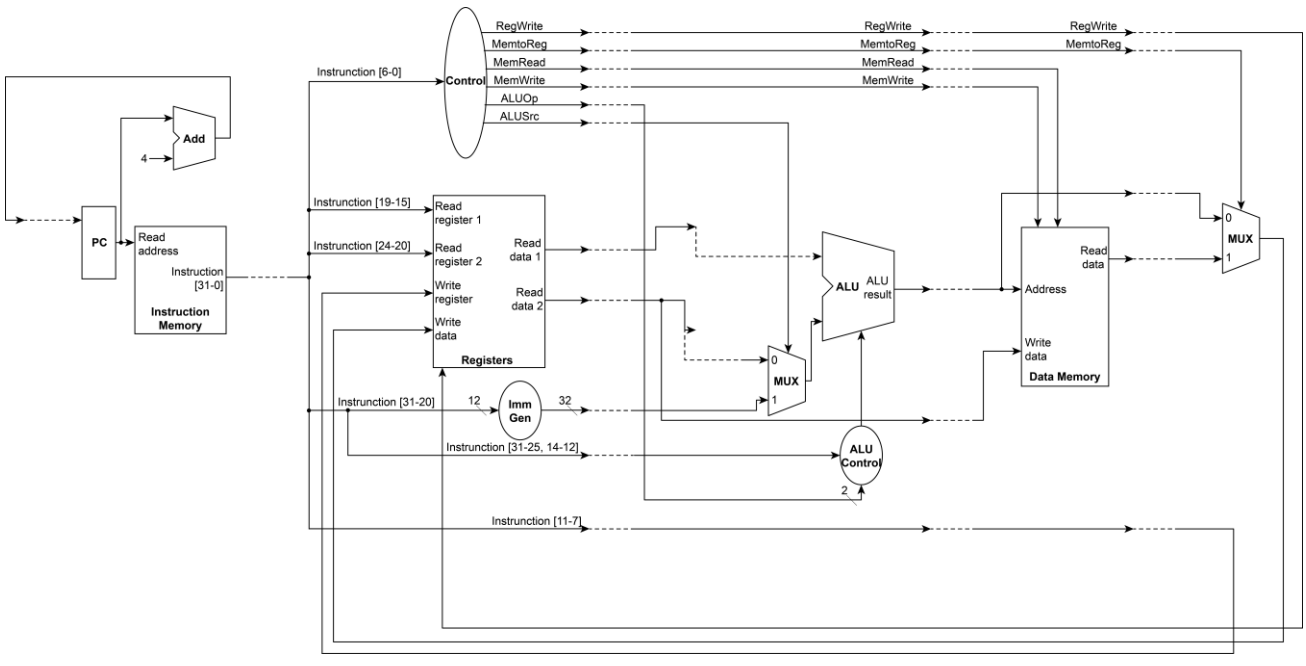

Figure 1 Single-cycle CPU with Data Memory

1.2. Pipeline Registers

To support pipeline execution, the first step is adding pipeline registers to the CPU. Pipeline registers store control signals and data from the last step and isolate each

step. Note that you have to use the non-blocking assignment in your pipeline register to get the correct result. And you have to properly initialize reg values in your pipeline registers.

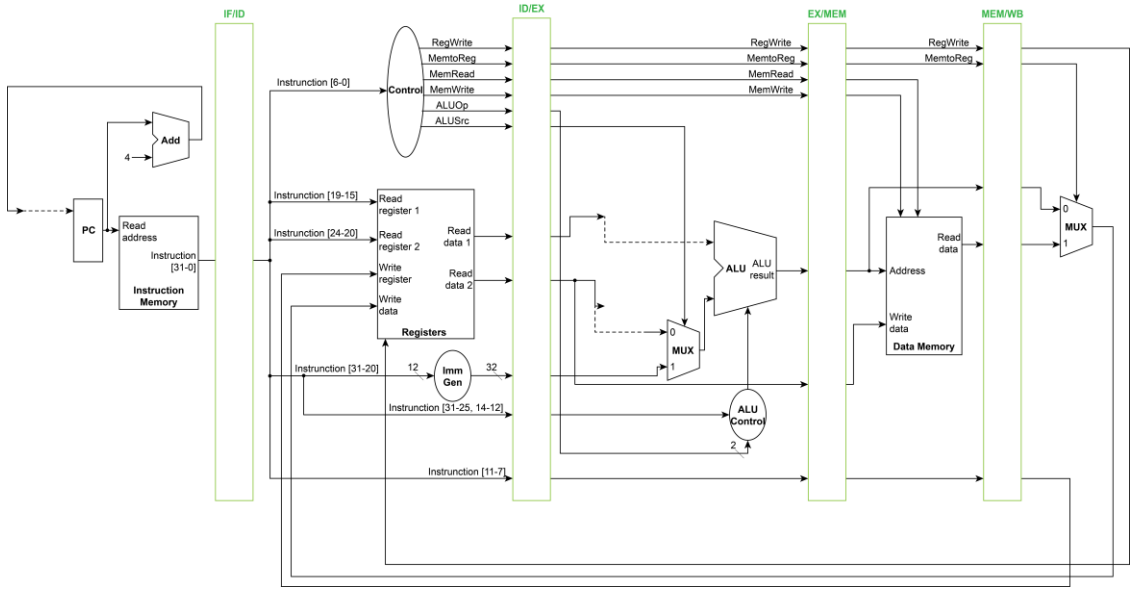

Figure 2 Datapath after adding pipeline registers (colored in green)

1.3. Data Hazard and Forwarding Control

The first issue after adding pipeline execution is to handle data hazards. For example, if we execution "`add t3, t1, t2`" and "`add t4, t1, t3`" sequentially, t3 will not be written back until the 5th cycle. But the second instruction is brought into the execution stage at the 4th cycle, causing a different result from single-cycle implementation. To properly handle such data hazards, we can forward the ALU result of the first instruction to the execution stage of the second instruction.

As a result, we shall have a "forwarding unit" exploring whether we should forward data from MEM stage or WB stage to EX stage. If the forwarding unit finds that the data in the later stage will be written back to the register that is taken in the EX stage, the data should be forwarded from the later stage to ensure correct arithmetic operation. To be more specific, Table 1, Listing 1, and Listing 2 are the recommended method to resolve data hazards from the textbook.

Note that the forwarding unit is placed in EX stage, implying that we only forward to EX stage. You don't have to handle the cases that forwarding to ID stage is necessary to keep the correctness. For example,

```
add x5, x6, x7
beq x5, x4, BRANCH
```

instruction sequences like this will not be involved in our evaluation.

MUX	value	Source	Explanation
ForwardA	00	ID/EX	The first ALU operand comes from the register files
	10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
	01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB	00	ID/EX	The second ALU operand comes from the register files
	10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
	01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Table 1 Forwarding Control

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs1)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

Listing 1 EX hazard

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
     and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
     and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Listing 2 MEM hazard

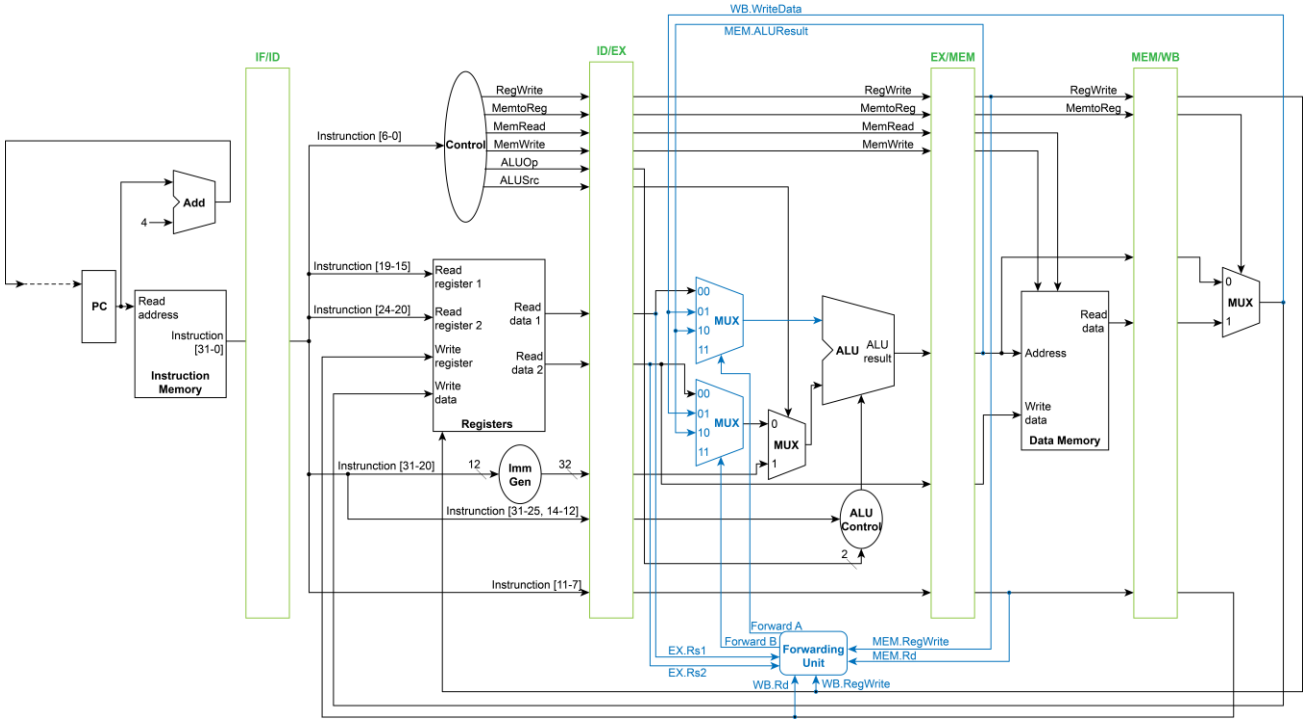

Figure 3 Data path after adding Forwarding Control (colored in blue)

1.4. Hazard Detection, Stall, and Flush

Besides data hazards caused by arithmetic operations, which can be resolved by forwarding, data hazards caused by "load" cannot be resolved simply by forwarding and requires stall. Another major difference between Homework 4 and this project is that we have to support branch instruction, which causes control hazards for the wrong prediction.

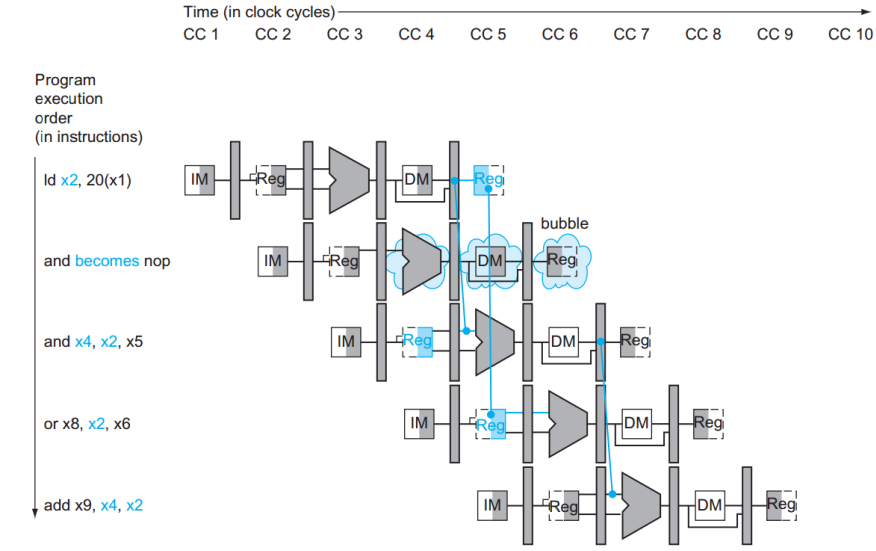

FIGURE 4.57 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise, the `or` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

Figure 4 An example to stall from textbook

As a result, we have to implement a "hazard detection unit" to detect whether to stall the pipeline or to flush when a control hazard happens. The hazard detection unit detects whether the `rd` in EX stage is the same as `rs1` or `rs2` in ID stage. If so, adding a nop (no operation) to the pipeline to resolve data hazard.

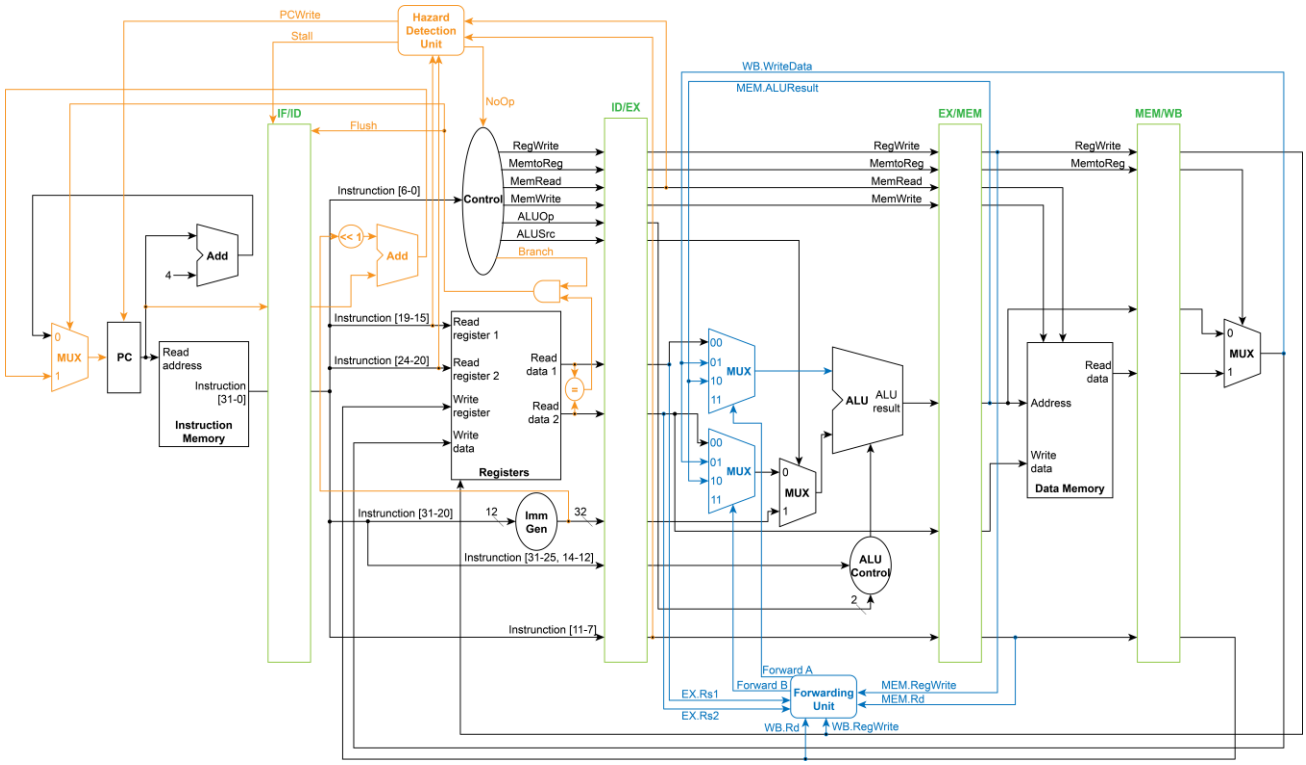

Figure 5 Final Datapath after Adding Hazard Detection Unit (colored in orange)

Note that to mitigate the impact of the branch instruction, we implement the branch decision at ID stage without using ALU, which is a recommended implementation by the textbook. You don't have to handle the forwarding to ID stage. We will avoid the test cases that require forwarding to ID stage (example at the end of 1.3).

1.5. Instructions

Besides instructions specified in Homework 4, you have to support additional 3 instructions, `lw`, `sw`, `beq`. Their machine code is as follows.

funct7	rs2	rs1	funct3	rd	opcode	function
0000000	rs2	rs1	111	rd	0110011	and
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	000	rd	0110011	add

0100000	rs2	rs1	000	rd	0110011	sub
0000001	rs2	rs1	000	rd	0110011	mul
imm[11:0]		rs1	000	rd	0010011	addi
0100000	imm[4:0]	rs1	101	rd	0010011	srai
imm[11:0]		**rs1**	**010**	**rd**	**0000011**	**lw**
imm[11:5]	**rs2**	**rs1**	**010**	**imm[4:0]**	**0100011**	**sw**
imm[12,10:5]	**rs2**	**rs1**	**000**	**imm[4:1,11]**	**1100011**	**beq**

1.6. Input / Output Format

Besides the modules listed above, you are also provided "testbench.v" and "instruction.txt". After you finish your modules and CPU, you should compile all of them including "testbench.v". A recommended compilation command would be

```
$ iverilog *.v -o CPU.out
```

Then by default, your CPU loads "instruction.txt", which should be placed in the same directory as CPU.out, into the instruction memory. This part is written in "testbench.v". You don't have to change it. "instruction.txt" is a plain text file that consists of 32 bits (ASCII 0 or 1) per line, representing one instruction per line. For example, the first 3 lines in "instruction.txt" are

```
0000000_00000_00000_000_01000_0110011 //add  $t0,$0,$0
00000001010_00000_000_01001_0010011  //addi $t1,$0,10
00000001101_00000_000_01010_0010011  //addi $t2,$0,13
```

Note that underlines and texts after "//" (i.e. comments) are neglected. They are inserted simply for human readability. Therefore, the CPU should take "00000000000000000000010000110011" and execute it in the first cycle, then "00000000101000000000010010010011" in the second cycle, and "00000000110100000000010100010011" in the third, and so on.

Also, if you include unchanged "testbench.v" into the compilation, the program will generate a plain text file named "output.txt", which dumps values of all registers and data memory at each cycle after execution. The file is self-explainable.

1.7. Modules You Need to Add or Modify

1.7.1. Control

Because your CPU has to support load/store instructions in this project, which are not involved in Homework 4, you have to add some additional control signals in the Control module.

1.7.2. Pipeline Registers

As is introduced in section 1.2, you have to implement 4 pipeline registers to isolate 5 pipeline stages, and passing essential information to the next stage.

1.7.3. Forwarding Unit

As is introduced in section 1.3, you need a forwarding unit and two multiplexers to control forwarding from MEM stage or WB stage.

1.7.4. Hazard Detection Unit

As is introduced in section 1.4, you need a hazard detection unit to handle the necessary stall and nop (no operation).

1.7.5. testbench

You have to initialize reg in your pipeline registers before any instruction is executed. It is recommended that you initialize them in the "initial" block of the testbench.v. Except for registers initialization, please do not change the output format (`$fdisplay` part) of this file.

1.7.6. Others

You can add more modules than listed above if you want. Figure 5 is simply a recommended data path for you to refer to. You are free to change some details as long as your CPU can perform correctly.

1.7.7. CPU

Use structure modeling to connect the input and output of modules following the data path in Figure 5.

1.8. Reminder

Project 2 will be strongly related to this homework. Please make sure you can fully understand how to write this homework; otherwise, you may encounter difficulties in your next project. Plagiarism is strongly prohibited.

2. Report

2.1. Modules Explanation

You should briefly explain how the modules you implement work in the report. You have to explain them in human-readable sentences. Either English or Chinese is welcome, but no Verilog. Explaining Verilog modules in Verilog is nonsense. Simply pasting your codes into the report with no or little explanation will get zero points for the report. You have to write more detail than Section 1.7.

Take "PC.v" as an example, an acceptable report would be:

> PC module reads clock signals, reset bit, start bit, and next cycle PC as input, and outputs the PC of the current cycle. This module changes its internal register "pc_o" at the positive edge of the clock signal. When the reset signal is set, PC is reset to 0. And PC will only be updated by next PC when the start bit is on.

And following report will get zero points.

> The inputs of PC are clk_i, rst_i, start_i, pc_i, and ouput pc_o.
> It works as follows:

```
always@(posedge clk_i or negedge rst_i) begin
    if(rst_i) begin
        pc_o <= 32'b0;
    end
    else begin
        if(start_i)
            pc_o <= pc_i;
        else
            pc_o <= pc_o;
    end
end
```

2.2. Members & Teamwork

Specify your team members and your work division. For example, who writes the pipeline registers, who is in charge of connecting wires in the CPU, etc.

2.3. Difficulties Encountered and Solutions in This Project

Write down the difficulties if any you encountered in doing this project, and the final solution to them.

2.4. Development Environment

Please specify the OS (e.g. MacOS, Windows, Ubuntu 18.04) and compiler (e.g. iverilog) or IDE (e.g. ModelSim) you use in the report, in case that we cannot reproduce the same result as the one in your computer.

3. Submission Rules

Put all your Verilog codes into a directory named "codes", then put "codes" and your report (should be named in format "teamXX_project1_report.pdf") into a directory named "teamXX_project1". **Note that you have to REMOVE Instruction_Memory.v, Data_Memory.v, Registers.v, PC.v, instruction.txt, output.txt, which are provided by TA, in your submission.** You can change "teamXX" into whatever you want. But please use ASCII-printable characters only. Finally, zip this directory, and upload this zip file onto NTU COOL before **12/15/2020 (Tue.) 23:59.**

In short, we should see a single directory like the following structure after we type

```
$ unzip teamXX_project1.zip
```

in Linux terminal:

- `teamXX_project1/`
  - `teamXX_project1/codes`
    - `teamXX_project1/codes/CPU.v`
    - `teamXX_project1/codes/ALU.v`
    - …
  - `teamXX_project1/teamXX_project1_report.pdf`

Make sure you remove the following files before submission: Instruction_Memory.v, Data_Memory.v, PC.v, Registers.v, instruction.txt, output.txt. And make sure your testbench.v reads instructions from "instruction.txt" and output to "output.txt".

4. Evaluation Criteria

We will compile your program in following command:

```
teamXX_project1/codes/ $ iverilog *.v ../../*.v -o CPU.out
```

, where `../../*.v` includes `Instruction_Memory.v, Data_Memory.v, PC.v, Registers.v`, and the working directory is in your "codes/" directory. That is, some modules are provided outside the directory you submit.

4.1. Programming Part (80%)

We will test your program by following set of instructions files.

- basic pipeline implementation without hazard, forwarding, and branch (30%)
- with data forwarding to resolve necessary data hazard but no stall (20%)
- data hazard including forwarding and necessary stall (20%)
- instructions including branch to examine the correctness of flush (10%)
- We will have a demonstration session on both projects. The time slots will be announced around mid-December. All of the team members must attend. You have to come up to briefly explain how your program works to get the credits of the programming part.

4.2. Report (20%)

4.3. Other

- Minor mistakes (examples below) causing compilation error: -10 pts
  - wrong usage of "`include"
  - submitting unnecessary files (those provided by TA except CPU.v and testbench.v)
  - other mistakes that can be fixed within 5 lines
- Major mistakes causing compilation error: 0 pts on programming part
- No show up at demonstration: 0 pts on programming part
- Wrong directory format: -10 pts
- Wrong I/O paths: -10 pts
- Late submission: -10 pts per day
- Plagiarism: 0 pts.

email to eclabta@gmail.com if you have any questions.