Lab4-1 Report

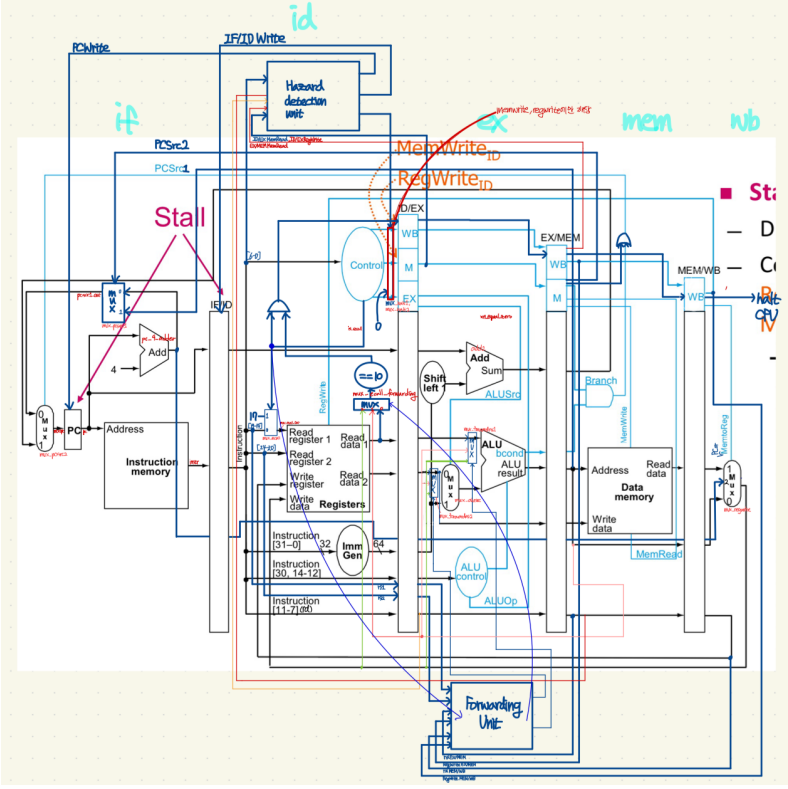
20230642 이채영, 20230683 박한비

1. Introduction

Lab4-1에서는 RISC-V instruction을 기반으로 하는 5-stage Pipelined CPU를 구현하였다. Pipelined CPU는 Multi-cycle CPU와 같이 하나의 instruction을 여러 사이클에 걸쳐 수행한다. 하지만 각 stage가 각 사이클마다 다른 instruction을 수행할 수 있고, 사이클 길이가 Single-cycle CPU보다 훨씬 짧으므로 Single-cycle CPU와 Multi-cycle CPU보다 훨씬 나은 성능을 보인다.

2. Design

이번 Lab4-1에서는 controlflow를 고려하지 않은 pipelined CPU를 구현하였다. 기본적으로 Lec8 Pipelined cpu – Data Hazard 교안의 14쪽과 28쪽, Lab4-1 introduction 교안 13쪽에 제시된 구조를 바탕으로 아래와 같이 구현하였다. 기존에 구현했던 Single-cycle CPU에 Data hazard를 감지하는 Hazard Detection unit, Data forwarding을 구현하는 Forwarding unit을 추가했으며, Pipelined CPU를 구현하기 위해 각 stage 사이에 Pipeline 레지스터들을 추가하였다. 더불어 halt 신호를 인식하고 수행하기 위해 IF/ID 단계에 mux, and 모듈을 배치하였고, MEM/WB 단계까지 신호가 전달되도록 설계하였다.



Pipelined CPU의 작동 과정

Pipelined CPU는 한 cycle동안 모든 stage가 각각의 instruction을 처리할 수 있게 하여, 일정 사이클 동안 더 많은 instruction을 처리할 수 있도록 설계된 CPU이다. 이를 위해 각 stage에서는 이전 stage에서의 정보와 시그널을 저장하는 레지스터가 있고(IF/ID, ID/EX, EX/MEM, MEM/WB 단계에 존재), 레지스터를 통해 전달받은 값을 이용해 instruction을 이전 stage에 이어서 처리한다.

그러나 instruction 사이에 true dependency가 있는 경우 data hazard가 발생해 pipelined CPU를 stall해야 한다. 각 instruction의 register use 및 produce 여부 및 위치(stage)는 아래와 같다. (교안 Lec8-pipelined-CPU-datahaz의 내용에 ECALL 사례 추가)

True dependency가 있는지는 Hazard Detection unit에서 판단하고, stall 할지를 결정한다. Stall 여부는 아래 식을 따르며 해당 식은 교안 Lec8-pipelined-CPU-datahaz의 내용에 ECALL의 경우를 추가한 것이다.

```
Stall =(opcode== ECALL) && [{(rd_EX== 17) && RegWrite_EX}|| {(rd_MEM== 17) && MemRead_MEM}] || (opcode != ECALL) && MemRead_EX && [{(rs1_ID == rd_EX) && use_rs1(IR_ID)} || {(rs2_ID == rd_EX) && use_rs2(IR_ID)}]
```

그리고 일부 instruction에서의 data hazard는 data forwarding으로 해결된다. Data forwarding Unit에서 이를 구현하여 ALU의 피연산자로 들어갈 값을 결정한다. Data forwarding 식은 아래와 같으며 해당 식은 교안 Lec8-pipelined-CPU-datahaz의 내용에 ECALL의 경우를 추가한 것이다.

```
If (rs1_EX != x0) && (rs1_EX == rd_MEM) && RegWrite_MEM then
        Forwarding_rs1 = 1 (from MEM stage)
Else if (rs1_EX != x0) && (rs1_EX == rd_WB) && RegWrite_WB then
        Forwarding_rs1 = 2 (from WB stage)
Else
        Forwarding_rs1 = 0 (No forwarding)

If (rs2_EX != x0) && (rs2_EX == rd_MEM) && RegWrite_MEM then
        Forwarding_rs2 = 1 (from MEM stage)
Else if (rs1_EX != x0) && (rs2_EX == rd_WB) && RegWrite_WB then
        Forwarding_rs1 = 2 (from WB stage)
Else
        Forwarding_rs2 = 0 (No forwarding)

If IsECALL && (rd_MEM == 17) && RegWrite_MEM then
        Forwarding_ecall = 1 (from MEM stage)
Else if IsECALL && (rd_WB == 17) && RegWrite_WB then
        Forwarding_ecall = 2 (from WB stage)
Else
        Forwarding_ecall = 0 (No forwarding)
```

3. Implementation

1) cpu.v

Pipleline CPU를 구현한 메인 모듈로, 내부에 여러 모듈들을 include하고 wire로 적절히 연결하며 cpu의 회로를 구현하였다. 또한 Pipleline CPU의 stage 사이에 각 stage의 값을 저장해놓기 위한 latch(pipeline register)들을 cpu.v 상에 추가해 놓았다. 해당 register들은 clock signal의 positive edge에 반응하여 stage사이에서 필요한 값들을 이동시켜준다(IF/ID latch의 경우 IF/IDWrite도 signal로 받는다). 위의 design에서 제시한 설계도와 동일한 구조이다.

```
// Update IF/ID pipeline registers here
always @(posedge clk_and_ishalted) begin
  if (reset) begin
    IF_ID_inst <= 0;
    IF_ID_pc <= 0;
    IF_ID_pc_add_4 <= 0;
  end
  else if (IF_ID_write) begin
    IF_ID_inst <= instr;
    IF_ID_pc <= current_pc;
    IF_ID_pc_add_4 <= current_pc_add_4;
  end
end
```

```
// Update ID/EX pipeline registers here
always @(posedge clk_and_ishalted) begin
  if (reset) begin
    ID_EX_alu_src <= 0;
    ID_EX_is_jal <= 0;
    ID_EX_is_jalr <= 0;
    ID_EX_is_branch <= 0;
    ID_EX_mem_write <= 0;
    ID_EX_mem_read <= 0;
    ID_EX_pc_or_mem_to_reg <= 0;
    ID_EX_reg_write <= 0;

    ID_EX_rs1_data <= 0;
    ID_EX_rs2_data <= 0;
    ID_EX_imm <= 0;
    ID_EX_opcode <= 0;
    ID_EX_funct3 <= 0;
    ID_EX_funct7_5 <= 0;
    ID_EX_rs1 <= 0;
    ID_EX_rs2 <= 0;
    ID_EX_rd <= 0;
    ID_EX_pc <= 0;
    ID_EX_pc_add_4 <= 0;
    ID_EX_is_halted <= 0;
  end

  else begin
    ID_EX_alu_src <= alu_src;
    ID_EX_is_jal <= is_jal;
    ID_EX_is_jalr <= is_jalr;
    ID_EX_is_branch <= is_branch;
    ID_EX_mem_write <= mem_write_final;
    ID_EX_mem_read <= mem_read;
    ID_EX_pc_or_mem_to_reg <= pc_or_mem_to_reg;
    ID_EX_reg_write <= reg_write_final;

    ID_EX_rs1_data <= read_data_1;
    ID_EX_rs2_data <= read_data_2;
    ID_EX_imm <= imm_gen_out;
    ID_EX_opcode <= IF_ID_inst[6:0];
    ID_EX_funct3 <= IF_ID_inst[14:12];
    ID_EX_funct7_5 <= IF_ID_inst[30];

    ID_EX_rs1 <= {27'b0, IF_ID_inst[19:15]};
    ID_EX_rs2 <= {27'b0, IF_ID_inst[24:20]};
    ID_EX_rd <= {27'b0, IF_ID_inst[11:7]};
    ID_EX_pc <= IF_ID_pc;
    ID_EX_pc_add_4 <= IF_ID_pc_add_4;
    ID_EX_is_halted <= IF_ID_is_halted;
  end
end
```

```
// Update EX/MEM pipeline registers here
always @(posedge clk_and_ishalted) begin
  if (reset) begin
    EX_MEM_is_jal <= 0;
    EX_MEM_is_jalr <= 0;
    EX_MEM_is_branch <= 0;
    EX_MEM_mem_write <= 0;
    EX_MEM_mem_read <= 0;
    EX_MEM_is_branch <= 0;
    EX_MEM_pc_or_mem_to_reg <= 0;
    EX_MEM_reg_write <= 0;

    EX_MEM_alu_result <= 0;
    EX_MEM_alu_bcond <= 0;
    EX_MEM_dmem_data <= 0;
    EX_MEM_rd <= 0;
    EX_MEM_pc_add_imm <= 0;
    EX_MEM_pc_add_4 <= 0;
    EX_MEM_is_halted <= 0;
  end
  else begin
    EX_MEM_is_jal <= ID_EX_is_jal;
    EX_MEM_is_jalr <= ID_EX_is_jalr;
    EX_MEM_is_branch <= ID_EX_is_branch;
    EX_MEM_mem_write <= ID_EX_mem_write;
    EX_MEM_mem_read <= ID_EX_mem_read;
    EX_MEM_pc_or_mem_to_reg <= ID_EX_pc_or_mem_to_reg;
    EX_MEM_reg_write <= ID_EX_reg_write;

    EX_MEM_alu_result <= alu_result;
    EX_MEM_alu_bcond <= alu_bcond;
    EX_MEM_dmem_data <= mux_forwardrs2_result;
    EX_MEM_rd <= ID_EX_rd;
    EX_MEM_pc_add_imm <= pc_imm_adder_result;
    EX_MEM_pc_add_4 <= ID_EX_pc_add_4;
    EX_MEM_is_halted <= ID_EX_is_halted;
  end
end
```

```
// Update MEM/WB pipeline registers here
always @(posedge clk_and_ishalted) begin
  if (reset) begin
    MEM_WB_pc_or_mem_to_reg <= 0;
    MEM_WB_reg_write <= 0;

    MEM_WB_mem_to_reg_src_0 <= 0;
    MEM_WB_mem_to_reg_src_1 <= 0;
    MEM_WB_mem_to_reg_src_2 <= 0;
    MEM_WB_rd <= 0;
    MEM_WB_is_halted <= 0;
  end
  else begin
    MEM_WB_pc_or_mem_to_reg <= EX_MEM_pc_or_mem_to_reg;
    MEM_WB_reg_write <= EX_MEM_reg_write;

    MEM_WB_mem_to_reg_src_0 <= EX_MEM_alu_result;
    MEM_WB_mem_to_reg_src_1 <= mem_dout;
    MEM_WB_mem_to_reg_src_2 <= EX_MEM_pc_add_4;
    MEM_WB_rd <= EX_MEM_rd;
    MEM_WB_is_halted <= EX_MEM_is_halted;
  end
end

mux_4x1 mux_regwrite(
  .in_1(MEM_WB_mem_to_reg_src_0),
  .in_2(MEM_WB_mem_to_reg_src_1),
  .in_3(MEM_WB_mem_to_reg_src_2),
  .in_4(0),
  .ctrl(MEM_WB_pc_or_mem_to_reg),
  .out(mux_regwrite_result)
);
```

위는 각 stage 사이의 pipeline register들의 업데이트를 구현한 코드이다.

2) pc.v

pc(program counter)의 값을 업데이트하며 CPU가 현재(그리고 다음) 수행할 instruction line의 메모리를 가리키도록 하여 프로세스의 흐름을 통제한다. 해당 모듈은 Lab2의 Single-Cycle CPU에서 구현한 모듈을 그대로 이용하였다.

3) DataMemory.v

data를 저장한 magic memory를 지니고 있으며 input으로 들어오는 인덱스에 따라 따라 메모리에서 인덱스 주소에 해당하는 값을 출력하거나 입력(이 경우 write 값도 필요)한다. 메모리에서 인덱스 주소에 해당하는 값을 출력하여 레지스터에 저장하는 load instruction을 수행하거나 레지스터 값을 반대로 메모리에 입력, 저장하는 store

instruction을 수행한다. 해당 모듈은 교안에 따라 제공된 모듈을 수정하지 않고 그대로 이용하였다.

4) InstMemory.v

instruction line을 저장한 magic memory를 지니고 있으며 input으로 들어오는 pc에 따라메모리에서 pc 주소에 해당하는 instruction을 출력하여 cpu가 해당 instruction을 해석하고 실행할 수 있게 한다. 해당 모듈은 교안에 따라 제공된 모듈을 수정하지 않고 그대로 이용하였다.

5) RegisterFile.v

32개의 register를 지니고 있으며 input으로 들어오는 인덱스에 따라 register의 값을 출력하거나 입력(이 경우 write 값도 필요)한다. 메모리에서 pc 주소에 해당하는 instruction을 출력하여 cpu가 해당 instruction을 해석하고 실행할 수 있게 한다. 해당 모듈은 교안에 따라 제공된 모듈을 수정하지 않고 그대로 이용하였다.

6) immediate_generator.v

instruction에서 이용되는 immediate 값이 instruction format 상에선 나누어져 저장되어 있다. 이를 instruction 타입을 참고해 완전한 immediate 값으로 합쳐서 output 으로 출력해준다. 해당 모듈은 Lab2의 Single-Cycle CPU에서 구현한 모듈을 그대로 이용하였다.

7) alu_control_unit.v

Instruction의 타입에 따라 ALU.v에서 적절한 arithmetic 연산을 수행하도록 alu의 operation code를 설정해준다. 해당 모듈은 Lab2의 Single-Cycle CPU에서 구현한 모듈을 그대로 이용하였다.

8) alu.v

CPU 상에서 arithmetic operation instruction과 그 외 instruction을 위한 arithmetic 연산을 수행한다. 해당 모듈은 Lab2의 Single-Cycle CPU에서 구현한 모듈을 그대로 이용하였다.

9) control_unit.v

instruction의 타입에 따라 각각의 모듈이 특정 일(write 등)을 수행할 지 여부를 결정하고 mux 등을 통해 cpu.v 회로 상의 흐름을 조절하며 또 instruction에 맞는 input(alu에 input 2로 immediate 값이 들어갈 지 register 값이 들어갈 지 등)이 각 모듈에 들어가도록 도와주는 control 신호들을 설정해준다. Lab2의 Single-Cycle CPU에서 구현한 모듈과 거의 동일하나 기존에 두가지로 나뉘어 나가던 register에 들어갈 데이터 타입을 결정해주는 시그널들(mem_to_reg, pc_to_reg)을 하나의 시그널(pc_or_mem_to_reg)로 합쳐 두개의 2x1 mux에 걸쳐 처리되던 register의 input data 결정 과정을 하나의 4x1 mux로 처리하게 수정하였다.

10) and_gate, or_gate, mux_2x1, mux_4x1

각각 CPU 상의 signal들을 and/or 연산으로 합쳐주거나, 회로 상 흐름을 조절하는 역할을 한다.

11) HazardDetectionUnit.v

```
`include "opcodes.v"

module HazardDetectionUnit (
    input [6:0] opcode ,  // input
    input [31:0] rs1,
    input [31:0] rs2,
    input [31:0] ID_EX_rd,
    input [31:0] EX_MEM_rd,
    input ID_EX_mem_read,
    input EX_MEM_mem_read,
    input ID_EX_reg_write,
    output reg pc_write,        // output
    output reg IF_ID_write,
    output reg is_stall);     // output

    always@(*) begin
        is_stall = 0;
        if(opcode == `ECALL) begin
            if (17 == ID_EX_rd && ID_EX_reg_write) begin
                is_stall = 1;
            end
            if (17 == EX_MEM_rd && EX_MEM_mem_read) begin
                is_stall = 1;
            end
        end
        else if(ID_EX_mem_read) begin
            if (rs1 == ID_EX_rd && (opcode == `ARITHMETIC || opcode == `ARITHMETIC_IMM || opcode == `LOAD || opcode == `STORE || opcode == `JALR || opcode == `BRANCH)) begin
                is_stall = 1;
            end
            if (rs2 == ID_EX_rd && (opcode == `ARITHMETIC || opcode == `STORE || opcode == `BRANCH)) begin
                is_stall = 1;
            end
        end

        pc_write = !is_stall;
        IF_ID_write = !is_stall;
    end


endmodule
```

Design에서 제시한 식을 그대로 코드로 작성하여 hazard의 발생 여부를 계산하였다. Hazard가 발생했을 경우 is_stall signal을 켜주고 pc_write, IF_ID_write를 꺼서 stall이 일어날 수 있게 해주었다. (if, id stage는 해당 stage를 위해 저장해놓은 pipeline register 값들을 전부 0으로 초기화해 일어나지 않도록 막았다. / id stage에서 ex stage로 진전하는

instr은 Memwrite, RegWrite를 0으로 설정해 해당 instr의 결과가 register 혹은 memory에 반영되지 않도록 하였다.)

12) DataForwardingUnit.v

```
`include "opcodes.v"

module DataForwardingUnit (
    input is_ecall,
    input [31:0] ID_EX_rs1,
    input [31:0] ID_EX_rs2,
    input [31:0] EX_MEM_rd,
    input [31:0] MEM_WB_rd,
    input EX_MEM_reg_write,
    input MEM_WB_reg_write,
    output reg [1:0] Forwarding_rs1,
    output reg [1:0] Forwarding_rs2,
    output reg [1:0] Forwarding_ecall);     // output

    always@(*) begin
        Forwarding_rs1 = 0;
        Forwarding_rs2 = 0;
        Forwarding_ecall = 0;
        if(ID_EX_rs1 != 0) begin
            if ((ID_EX_rs1 == EX_MEM_rd) && EX_MEM_reg_write) begin
                Forwarding_rs1 = 1;
            end
            else if ((ID_EX_rs1 == MEM_WB_rd) && MEM_WB_reg_write) begin
                Forwarding_rs1 = 2;
            end
        end
        if(ID_EX_rs2 != 0) begin
            if ((ID_EX_rs2 == EX_MEM_rd) && EX_MEM_reg_write) begin
                Forwarding_rs2 = 1;
            end
            else if ((ID_EX_rs2 == MEM_WB_rd) && MEM_WB_reg_write) begin
                Forwarding_rs2 = 2;
            end
        end
        if(is_ecall) begin
            if ((17 == EX_MEM_rd) && EX_MEM_reg_write) begin
                Forwarding_ecall = 1;
            end
            else if ((17 == MEM_WB_rd) && MEM_WB_reg_write) begin
                Forwarding_ecall = 2;
            end
        end
    end

endmodule
```

Design에서 제시한 식을 그대로 코드로 작성하여 rs1, rs2, x17번 register(ecall의 경우)에서의 forwarding 발생 여부를 계산하였다. Forwarding이 발생했을 경우 각각 Forwarding_rs1, Forwarding_rs2, Forwarding_ecall signal을 켜주어 mux로 register file의 output이 다른 경로의 값을 가져와 forwarding이 일어날 수 있게 해주었다.

4. Discussion

1) ECALL instruction의 hazard 및 forwarding 문제

ECALL instruction에서 발생하는 hazard를 해결하기 위해 is_halted를 계산하는 회로를 기존에 있던 id stage에서 ex stage로 옮겨 다른 instruction과 동일하게 forwarding, stall을 진행하였는데 해당 방식으로 하다 보니 cycle 수가 기존보다 줄어드는 문제가 발생하였다. 따라서 다시 id stage로 옮긴 후 다른 instruction들과 stall의 조건을 다르게 설정하고 나가는 forwarding signal이 다르도록(이전에서는 rs1의 forwarding으로 함께 해결) 하

였다. is_halted를 계산하는 회로는 다른 instruction들과 다르게 ex stage가 아닌 id stage에서 rs2 register의 값을 이용하기 때문에 forwarding을 진행하여도 rs1의 값을 write하는 instruction이 load가 아니고 ex stage에 있다면 1 stall, 또 load이고 ex stage에 있다면 2 stall, mem stage에 있다면 1 stall이 발생한다. 따라서 기존의 ex stage 상의 mem_read 시그널(ex stage에 load 있는지 판별) 외에 ex stage 상의 reg_write 시그널(ex stage에 reg write하는 instruction 있는지 판별)과 mem stage 상의 mem_read 시그널(ex stage에 load 있는지 판별)을 이용하여 hazard를 detection-stall을 설정할 수 있도록 조건식을 작성해 해결했다.

5. Conclusion

Lab4에서 작성한 코드로 주어진 테스트벤치(Non-control flow input file)를 실행한 결과는 아래 왼쪽과 같으며, 테스트벤치 실행 후 레지스터의 값들이 적절하게 할당된 것을 확인할 수 있다.

또한 아래 오른쪽의 Single-Cycle CPU를 이용해 동일한 테스트벤치를 실행한 결과와 비교해보면 테스트벤치 실행에 걸리는 사이클 수가 Pipeline CPU에서 더 큰 것을 볼 수 있다. 그러나 한 instruction을 완전히 실행하는 데 걸리는 시간을 cycle로 설정하는 Single-Cycle CPU과 달리 Pipeline CPU에서는 한 instruction을 5 stage로 분할하여 각 stage를 실행하는 데 걸리는 시간을 cycle로 설정(보통 가장 오래 걸리는 stage-load를 기준으로)하기 때문에 cycle의 주기가 훨씬 줄어들어 실제 총 실행시간은 Single-Cycle CPU보다 Pipeline CPU에서 훨씬 줄어들 것이다.

```
### SIMULATING ###
TEST END
SIM TIME : 94
TOTAL CYCLE : 46 (Answer : 46)
FINAL REGISTER OUTPUT
 0 00000000 (Answer : 00000000)
 1 00000000 (Answer : 00000000)
 2 00002ffc (Answer : 00002ffc)
 3 00000000 (Answer : 00000000)
 4 00000000 (Answer : 00000000)
 5 00000000 (Answer : 00000000)
 6 00000000 (Answer : 00000000)
 7 00000000 (Answer : 00000000)
 8 00000000 (Answer : 00000000)
 9 00000000 (Answer : 00000000)
10 0000000a (Answer : 0000000a)
11 0000003f (Answer : 0000003f)
12 fffffff1 (Answer : fffffff1)
13 0000002f (Answer : 0000002f)
14 0000000e (Answer : 0000000e)
15 00000021 (Answer : 00000021)
16 0000000a (Answer : 0000000a)
17 0000000a (Answer : 0000000a)
18 00000000 (Answer : 00000000)
19 00000000 (Answer : 00000000)
20 00000000 (Answer : 00000000)
21 00000000 (Answer : 00000000)
22 00000000 (Answer : 00000000)
23 00000000 (Answer : 00000000)
24 00000000 (Answer : 00000000)
25 00000000 (Answer : 00000000)
26 00000000 (Answer : 00000000)
27 00000000 (Answer : 00000000)
28 00000000 (Answer : 00000000)
29 00000000 (Answer : 00000000)
30 00000000 (Answer : 00000000)
31 00000000 (Answer : 00000000)
Correct output : 32/32
```

```
### SIMULATING ###
TEST END
SIM TIME : 80
TOTAL CYCLE : 39
FINAL REGISTER OUTPU
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 0000000a
11 0000003f
12 fffffff1
13 0000002f
14 0000000e
15 00000021
16 0000000a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
```

Discussion에서 말했던 대로 cycle 수를 맞추기 위해 ECALL instruction 시 is_halted 계산을 하는 stage를 id로 설정하였는데 is_halted 시그널은 결국 wb stage까지 전달되어 wb stage에서 output으로 내보내는 시그널이기 때문에 굳이 id stage에서 처리하지 않고 뒤로 보내어 뒤 stage에서 계산한다면 ECALL instruction에서 hazard 시 발생하는 stall을 줄여 소요되는 cycle 수를 더 줄일 수 있을 것이다.

Lab4를 통해 Pipeline CPU의 구조와 data hazard의 처리 원리 및 과정을 습득할 수 있었다. Stage가 눈으로 명확히 보일만큼 분명하게 나누어져 있는 Pipeline CPU의 설계를 그리면서 각 stage에서 일어나는 일에 대해 머릿속으로 쉽게 그릴 수 있게 되었다.