컴퓨터 구조 Lab4 Pipelined CPU (non-control)

20210207 이지현
20210794 정유진

1. Introduction

1.1. Lab 4는 Vivado를 이용하여 5-stage pipelined RISC-V CPU를 구현하는 것을 목표로 한다.

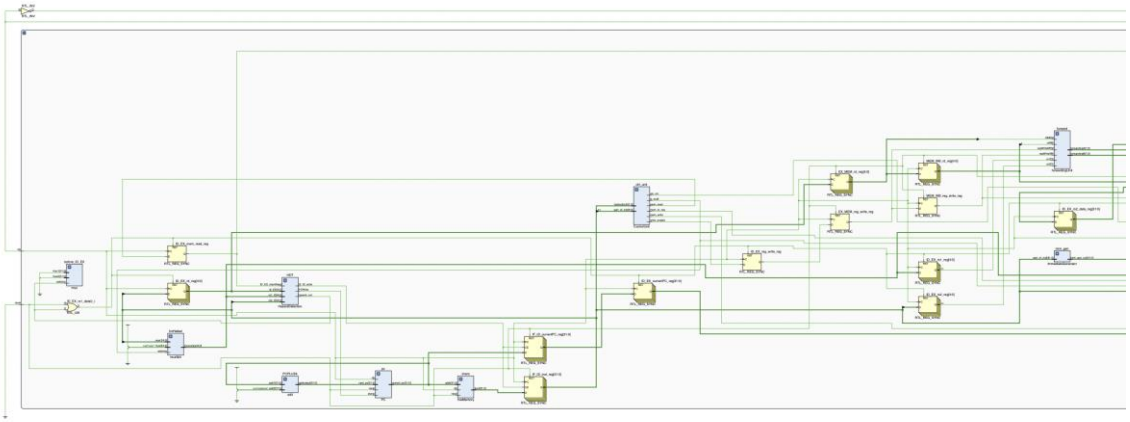
1.2. Data hazard의 처리는 stall 파트와 data forwarding 파트로 나누어 구현하였고, control hazard는 해당 랩에서는 처리하지 않았다.

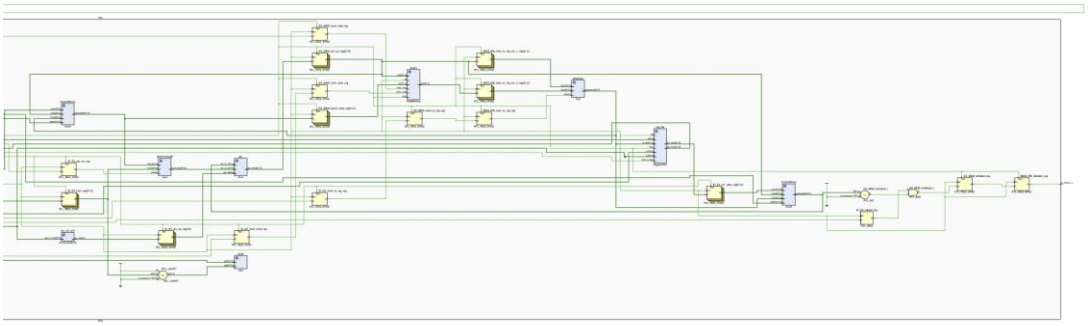
1.3. 주어진 스켈레톤 코드는 top.v, Memory.v, RegisterFile.v, cpu.v, opcodes.v가 있고 추가로 만든 V 파일은 ALU.v, ALUControlUnit.v, ImmediateGenerator.v, ControlUnit.v, PC.v, mux5bit.v, mux.v, mux4.v, HazardDetection, forwardingUnit가 있다.

1.4. 테스트 코드는 non_control_flow가 있으며, Ripes를 통해 얻은 레지스터 값과 베릴로그 상에서 얻은 레지스터 값과 비교하며 정상적인 동작을 확인할 수 있다.

1.5. Pipelined CPU의 design과 implementation을 설명할 때 각각의 모듈이 synchronous인지 asynchronous인지 확인하며 각각의 스테이지를 설명하고자 한다.

2. Design





2.1. Pipelined CPU 동작이 이루어지는 과정에 대한 순차적인 설명은 다음과 같다.

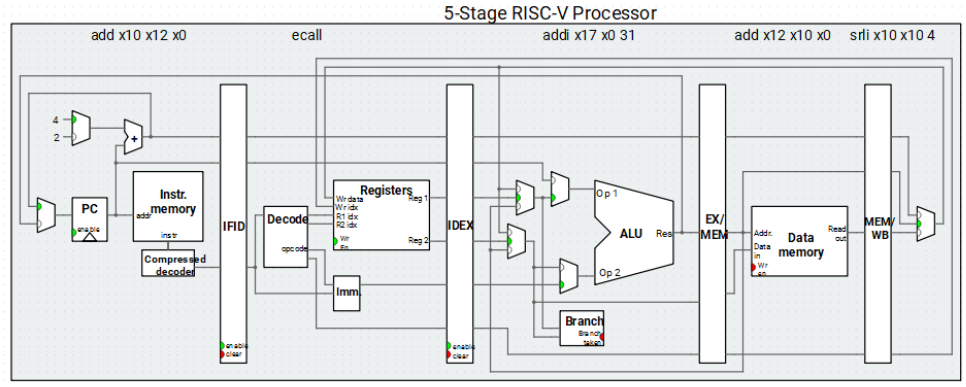
- Instruction processing은 보통 5가지 절차로 이루어진다.
- 파이프라인 CPU는 HW 활용률(utilization) 향상을 통한 throughput 향상을 위해 한 stage씩 instruction이 쪼개서 작업이 진행되고, 여기서는 5개의 stage의 파이프라인을 가진다. => 5 stage pipelined cpu
  - IF : instruction fetch로, 인스트럭션 실행을 위해선 메모리로부터 읽어와야한다.
    - Instruction memory, PC 가 주로 수행한다.
  - ID : instruction decode로, 레지스터 파일에서 레지스터 값을 읽어오며 (operand fetch) instruction에 따른 control signal, immedgate value를 발생시킨다.
    - ControlUnit, immediateGenerator, RegisterFile 가 주로 수행한다.
  - EX : execution으로, ALU 등을 통한 연산이 이루어진다.
    - ALU, ALUControlUnit 가 주로 수행한다.
  - MEM : data memory access로, instruction (load/store)에서 접근하고자 하는 메모리를 읽어온다.
    - Data memory 가 주로 수행한다.
  - WB : write back으로, 레지스터 파일을 업데이트 하고 메모리에서 읽은 값을 rd에 저장하는 등의 과정이 이루어진다.
    - RegisterFile 가 주로 수행한다.
- 단, 이 과정에서 발생하는 data hazard는 stall과 data forwarding을 통해서 해결할 수 있는데, 이를 위해 hazard detection unit과 forwarding unit을 추가해서 조건에 따라 stall되거나 앞서 생성된 와이어값을 레지스터에 적용되지 않은 상태에서도 forwarding해서 가져와 쓸 수 있도록 하였다.
- Data forwarding을 구현하였기에 load 다음 instruction이 true dependency를 가진다면 stall 하나를 가진다.
- isHalted = 1일 때는 isHalted 신호가 MEM/WB 레지스터에 갈 때까지 1개의 stall이 생긴다.
- 위와 같은 메커니즘으로 파이프라인 CPU는 작동된다.

2.2. Single cycle CPU와 Pipelined CPU의 total cycle을 non-control flow input file을 넣어서 비교하면 다음과 같다.

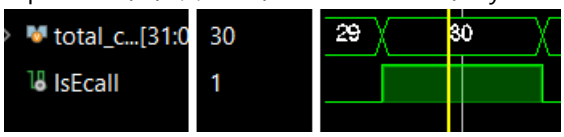
- Single-cycle CPU (in Lab2, non-control flow) cycles = 51 cycle
- Pipelined CPU (in Lab4, non-control flow) cycles = 56 cycle
  - Ripes 상 = 62 cycles
  - 두 값이 다른 이유 : 우선, Ripes 상에서는 0 Cycle일 때 가장 첫번째 instruction이 IF stage에 들어와 있으므로, 1 Cycle일 때 첫번째 instruction이

들어오는 우리의 구현과는 다르다.

두 값이 다른 두번째 이유는 Ecall instruction에서의 처리이다.

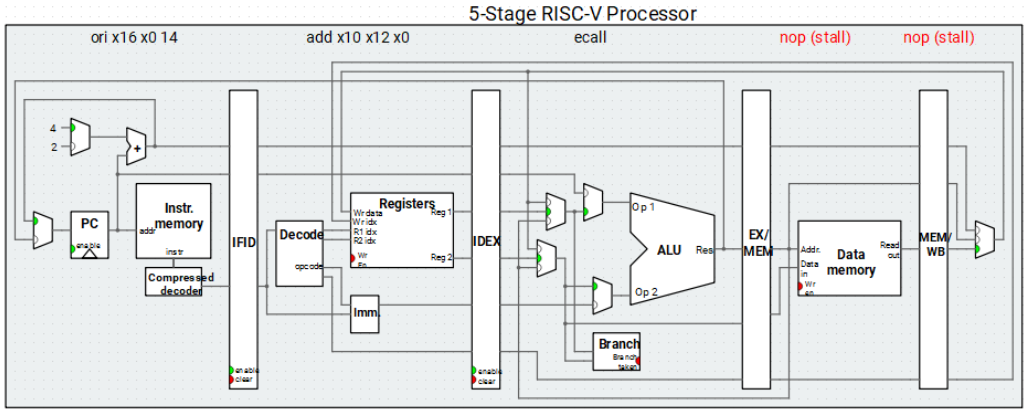


Ripes 상에서 첫 번째 Ecall은 29번째 Cycle일 때로,



Ecall 전까지는 우리의 구현과 사이클 수가 일치한다.

Ripes 상에서는 Ecall instruction시 EXE stage까지 가서 stall을 만들어냈었다.



그 이유는 rf17의 값을 올바르게 확인하기 위함인 것으로 추측된다.

하지만, 우리의 구현에서는 ecall 명령어일때, 17번째 레지스터에 대한 포워딩을 수행하면서 stall이 없어지거나 한 개가 되게 된다.

테스트 케이스를 분석한 결과, stall이 한 개가 되는 lw 연산은 없었다. 따라서 ecall 명령어 3회로 Ripes상보다 2*3 = 6 사이클이 적어져야 한다.

하지만, 우리의 마지막 종료 구현에서, is_halted는 MEM/WB 레지스터에서 가져오게끔 되어 있으므로 1 사이클이 추가된다.

사이클을 계산하면, 56 – 1(표기의 차이) + 2 * 3(stall 차이) + 1(마지막 ecall) = 62로 Ripes 상의 사이클 차이가 설명된다.

- Single cycle CPU와 pipelined CPU의 non-control flow cycle수는 51, 56으로 pipelined CPU가 더 많다.
- Stall이 없다고 가정했을 때 Pipelined CPU에서의 5 clk에서 완료되는 instruction의 수는 single cycle CPU에서의 1 clk에서 완료되는 instruction의 수와 같다.

- 그렇기에 single cycle cpu에서보다 4개의 cycle이 더 들며, stall이 있을 때마다 1개의 Cycle씩이 더 필요하다. Hazard detection unit에서 발견된 hazard out은 1개이므로 stall도 하나가 더 들어갔으므로, 같은 Instruction에 대해 총 5개의 cycle이 single cycle cpu에 비해 더 필요하다고 할 수 있다.
- 따라서 Single cycle CPU가 51 사이클, Pipelined CPU가 +5 해서 56 사이클을 갖게 된다.

2.3. Hazard detection을 어떻게 구현했는지, 언제 detect되는지를 설명하면 다음과 같다.

- HazardDetection.v 에 구현하였다.

```
module HazardDetection(
    input [4:0] rs1_ID,
    input [4:0] rs2_ID,
    input [4:0] rd_EX,
    input ID_EX_memRead,
    output reg PCWrite,
    output reg IF_ID_write,
    output reg hazard_out
    );

    always @(*) begin
        hazard_out = 0;
        PCWrite = 1;
        IF_ID_write = 1;
        if ((((rs1_ID == rd_EX) &&  rs1_ID!=0) || ((rs2_ID == rd_EX) && rs2_ID!=0)) && ID_EX_memRead) begin
            PCWrite = 0;
            IF_ID_write = 0;
            hazard_out = 1;
        end
    end

endmodule
```

-
- Input : rs1_ID, rs2_ID, rd_EX, ID_EX_memRead
- Output : PCWrite, IF_ID_write, hazard_out
- Combinational logic으로 구현하였으며, stall이 필요한 경우 hazard detection의 hazard out을 1로 내보내 알려주는 기능을 갖고 있다.
- **Data forwarding을 구현하였기에 load일 경우에 true dependencies(RAW)를 갖고 있다면 1 stall을 가지면 된다.** 따라서 ID_EX_memRead signal이 1이어야만 작동한다. (Load instruction을 나타내는 signal이므로)
- 이 때, true dependency를 가지는 경우는 ID stage의 rs1과 EX stage의 rd가 같고 rs1_ID가 0이 아닐 때, ID stage의 rs2와 EX stage의 rd가 같으면 rs2_ID가 0이 아닌 경우가 있을 수 있다.
- 이런 경우, hazard_out 은 1이 되어 hazard detection이 되었음을 알리고, PCwrite와 IF_ID_write를 0으로 만들어 다음 instruction이 실행되는 것을 막을 수 있다.

2.4. Data forwarding을 어떻게 구현했는지, 언제 forward되는지 설명하면 다음과 같다.

```
module forwardingUnit(
    input [4:0] rs1EX,
    input [4:0] rs2EX,
    input [4:0] rdMEM,
    input regWriteMEM,
    input [4:0] rdWB,
    input regWriteWB,
    output reg [1:0]forwardingA,
    output reg [1:0]forwardingB
    );

    always @(*) begin
        if(rs1EX != 0 && rs1EX == rdMEM && regWriteMEM)
            forwardingA = 2'b01;
        else if(rs1EX != 0 && rs1EX == rdWB && regWriteWB)
            forwardingA = 2'b10;
        else
            forwardingA = 2'b00;
        if(rs2EX != 0 && rs2EX == rdMEM && regWriteMEM)
            forwardingB = 2'b01;
        else if(rs2EX != 0 && rs2EX == rdWB && regWriteWB)
            forwardingB = 2'b10;
        else
            forwardingB = 2'b00;
    end

endmodule
```

- forwardingA는 rs1의 값을 어디에서 가져오는지, forwardingB는 rs2의 값을 어디에서 가져오는지를 나타낸 변수이다.
- (일반성을 잃지 않고, rs1에 대해서 설명하겠다)
  EXE stage에서 사용되는 rs1이 0(x0)이 아니고, 이 rs1이 바로 뒤 MEM stage의 rd이면서, MEM stage에서 register Write가 되었다면, rs1이 바로 뒤 MEM stage를 통해 새로 써진다는 의미이다. 따라서, EXE/MEM register에서 rs1을 가져와 주어야 한다.
  EXE stage에서 사용되는 rs1이 0이 아니고, 이 rs1이 뒤의 뒤 WB stage의 rd이면서, WB stage에서 register Write가 되었다면, rs1이 WB stage를 통해 새로 써진다는 의미이다. 따라서 MEM/WB register에서 rs1을 가져와 주어야 한다.
  우선순위는 MEM stage가 위이므로 else if을 사용하였다.

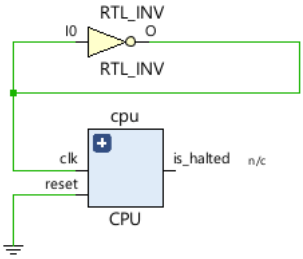
2.5. 다음은 각각의 모듈에 대한 **synchronous/asynchronous 디자인의** 포괄적인 설명이다.

Cpu	Top module로서 sub module들의 인스턴스 선언과 wire 변수들의 연결이 이루어지는 곳이다.
ALU	ALU 연산을 한다. ALU operation의 종류와, ALU 연산을 할 input 2개를 asynchronous하게 읽어와서, 연산을 할 result와, branch

	operation일 경우에는 branch condition에 해당하는 지를 리턴한다.
ALUControlUnit	Instruction를 해독하여 ALU operation을 내보내주는 역할을 한다. Instruction을 asynchronous하게 읽어온다.
ControlUnit	Asynchronous하게 Instruction을 읽어와, 이에 해당하는 CPU의 동작을 확인하여, 각 module에 내보내준다.
Memory	Instruction memory와 data memory로 나뉘며, Instruction memory에서는 asynchronous하게 instruction을 읽어오며 CPU Reset시 synchronous하게 초기화가 이루어진다. Data memory에서는 asynchronous하게 memory에서 data를 읽어오며 synchronous하게 data를 write한다. 마찬가지로 CPU Reset시 synchronous하게 초기화가 이루어진다.
Mux	selector가 0이면 output으로 mux1을 assign하고, 아니면 output으로 mux2를 assign한다. Dataflow modeling이다.
Mux4	Selector가 00이면 output으로 mux1을 assign하고, 01이면 mux2를 assign하고, 10이면 mux3를 assign한다. Dataflow modeling이다.
Mux5bit	Selector가 0이면 output으로 5비트 mux1을 assign하고, 아니면 mux2를 assign한다. Dataflow modeling이다.
Opcodes	연산들의 funct3과 opcode를 선언해 놓았다.
PC	Synchronous하게 매 clk마다 current_pc를 next_pc(pc+4)의 값으로 업데이트 해준다.
RegisterFile	register file의 rs1 rs2 값을 asynchronous하게 읽어오고 write_enable = 1이고 rd가 0이 아니라면 synchronous하게 rd에 write한다. CPU Reset시 synchronous하게 초기화가 이루어진다.
hazardDetection	hazard detection 조건을 만족한다면 Asynchronous하게 PCwrite, IF_ID_write, hazard_out signal을 업데이트해준다.
forwardingUnit	input인 rs1EX, rs2EX, rdMEM, regWriteMEM, rdWB, regWriteWB가 바뀜에 따라 asynchronous하게 forwarding이 어떻게 되는지를 내보낸다.
Top	CPU instance를 선언하고, 처음에 reset을 한 뒤, clk를 특정 시간 간격마다 -clk로 바꾸어주어 synchronous한 동작에 도움을 준다. 프로그램이 끝나면, total_cycle과 register의 상황을 출력해 준다.

3. Implementation

3.1. <cpu.v> = top module



Input : clk, reset

Output : is_halted

사용한 wire/Reg 변수들의 설명은 다음과 같다.

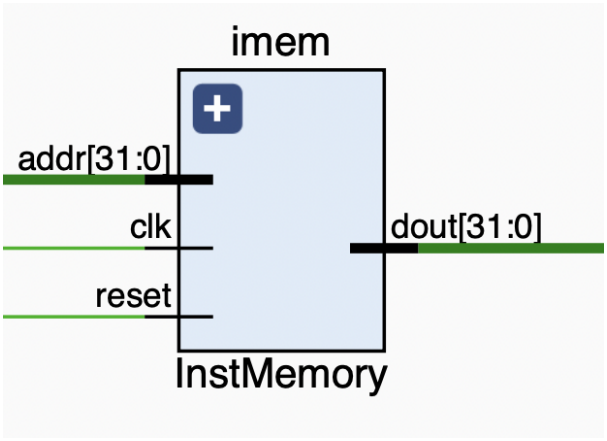
PC	nextPC : 다음 clock의 PC currentPC : 현재 PC real_currentPC : mux를 거치고 난 현재 PC currentPCplus4 : 현재 PC + 4값 beforeInst : Instruction memory를 나온 dout
ALU	ID_EX_alu_op : 결정된 ALU opreation real_ALU_inA : ALU에 들어가는 최종 첫 번째 operand real_ALU_inB : ALU에 들어가는 최종 두 번째 operand alu_bcond : 여기선 항상 0 = alu_zero ALU_out : ALU output
register	writeRegister : RD RegReadData1 : RS1 RegReadData2 : RS2 MemData : memory data rf17 : Regster file [17] <- 레지스터파일 사용X
memory	real_DataOutput : mux를 거치고 난 data output DataReadData : mux를 거치기 전 data output
Immediate generator & mux step	Imm_gen_out : immediate generator output
Control unit	isJal : Jal instruction인지 여부 output isJalr : Jalr instruction인지 여부 output isBranch Branch instruction인지 여부 output RegWrite : 레지스터에 쓰는지 여부 output MemRead : 메모리를 읽는지 여부 output MemToReg : 메모리의 값을 레지스터로 가져오는지 여부 output MemWrite : 메모리에 쓰는지 여부 output ALUSrc : ALU 연산에 immediate가 활용되는지 여부

	output PCtoReg : PC가 register에 저장되는지 여부 output(여기서는 항상 0) Is_ecall : ecall instruction인지 여부 output
Forwarding unit	forwardingA : A를(rs1) 어디서 가져오는지를 나타냄 forwardingB : B를(rs2) 어디서 가져오는지를 나타냄
Hazard detection	IF_ID_write : IF_ID 파이프라인 레지스터에 write 가능 signal PCWrite : 다음 PC 가져오기 가능 signal Hazard_out : hazard detection 여부 signal afterhaltingMuxrs1 : (ecall 연산일 경우 rs1을 17로, 아닐 경우 기존의 instruction에서 rs1을 가져오게 된다.) CPU에서 다룰 최종적인 rs1의 값이라 할 수 있다.
Pipeline (파이프라인 레지스터 / 설명은 생략한다)	IF_ID_inst : IF_ID_currentPC: ID_EX_alu_op : ID_EX_alu_src : ID_EX_mem_write ID_EX_mem_read ID_EX_mem_to_reg ID_EX_reg_write ID_EX_rs1_data ID_EX_rs2_data ID_EX_rs1 ID_EX_rs2 ID_EX_imm ID_EX_ALU_ctrl_unit_input ID_EX_rd ID_EX_currentPC ID_EX_isHalted ID_EX_isEcall EX_MEM_mem_write EX_MEM_mem_read EX_MEM_is_branch EX_MEM_mem_to_reg EX_MEM_reg_write EX_MEM_alu_out EX_MEM_dmem_data EX_MEM_rd

	EX_MEM_bcond EX_MEM_SUM_out EX_MEM_isHalted MEM_WB_mem_to_reg MEM_WB_reg_write MEM_WB_mem_to_reg_src_1 MEM_WB_mem_to_reg_src_2 MEM_WB_isHalted

3.2. <Memory.v>

- Instruction Memory

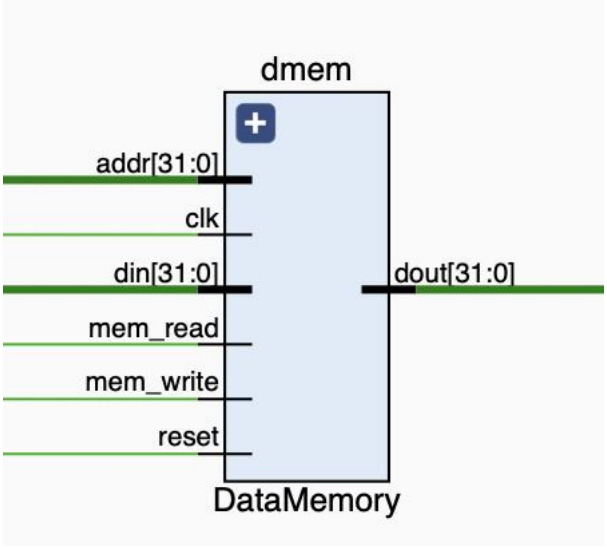


Input : reset, clk, addr

Output : dout

  - Asynchronous logic : imm_addr를 이용해 memory에 access해 dout에 담는 과정을 addr가 변할 때마다 asynchronous하게 진행하였다.
  - (Positive clk) Synchronous logic : reset버튼이 눌러졌을 때 instruction memory를 0으로 초기화하는 과정은 clk이 positive일 때마다 synchronous하게 진행하였으며, pc값이 돌아올때마다 각각의 테스트 코드에 있는 memory 경로의 값을 읽어오는 과정도 clk에 따라 진행되었다

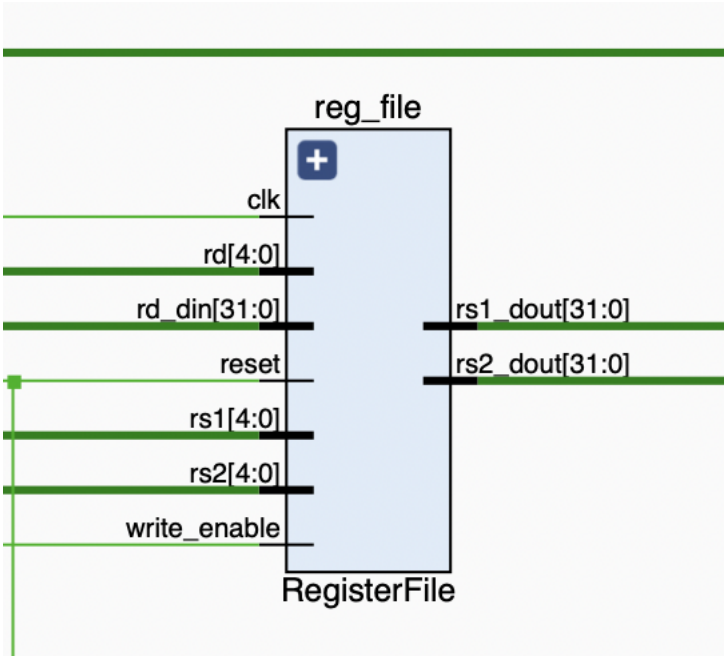
- Data Memory



Input : reset, clk, addr, din, mem_read, mem_write
Output : dout

- Asynchronous logic : mem_read signal이 1이면 dout에 mem[dmem_addr]를 대입하는 과정은 assign의 오른쪽 값이 바뀌면 대입되는 방식을 이용했으므로 asynchronous한하게 진행된다고 볼 수 있다.
- (Positive clk) Synchronous logic : positive clk일 때 mem_write signal이 1인지 확인하고 din값을 memory에 write하는 과정과 reset이 1일때 data memory를 초기화하는 과정은 synchronous하게 진행된다.

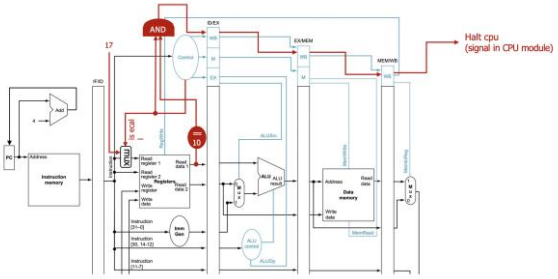
3.3. <RegisterFile.v >



Input : reset, clk, rs1, rs2, rd, rd_din, write_enable
Output : rs1_dout, rs2_dout

- Asynchronous logic : rf[rs1], rf[rs2]값이 변할 때마다 rs1_dout, rs2_dout에 assign문을 통해 대입해주는 과정은 asynchronous하다.
- (Positive clk) Synchronous logic : positive clk일 때마다 rf[0]의 값은 변하면 안되므로 rd가 0이 아니고 write enable이 1인지를 확인해 rd_din을 레지스터 파일에 대입해주는 과정은 synchronous하다. 마찬가지로 reset이 1일 때 레지스터 파일을 0으로 초기화해주는 과정또한 synchrounous하다.

● 단, 이때 registerfile.v를 변경할 수 없으므로, isHalted는 다른 방법으로 해결해야한다.

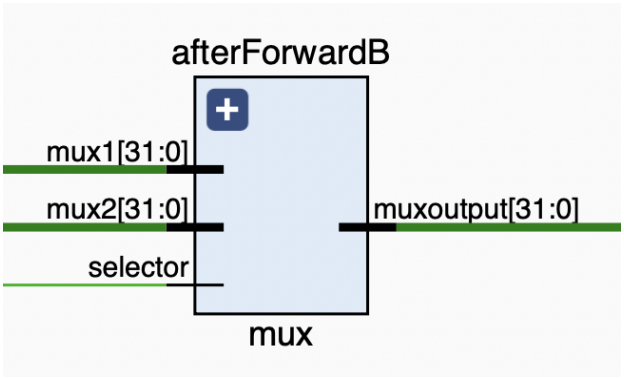


- Rs1앞에서 mux를 달고 17과 경쟁하는데, 이는 isEcall signal에 의해 결정된다. 이 Rs1 레지스터 값이 10과 같으면 파이프라인 레지스터 is_halted에 전달되어 최종적으로 ishalted가 도출된다.

3.4. <mux.v>

mux1과 mux2와 selector를 input으로 받아서 selector가 0이면 mux1을, 1이면 mux2를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분들은 다음과 같다.

● AfterForwardB

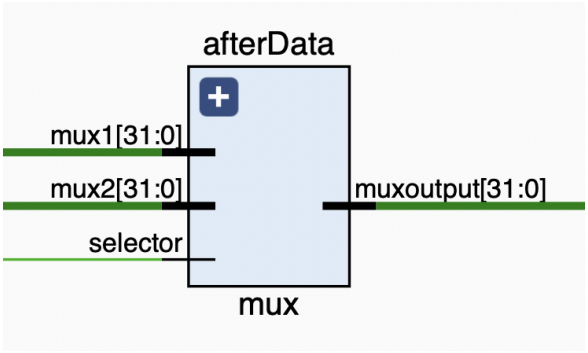


Input : mux1 (real_ID_EX_rs2_before), mux2 (ID_EX_imm), selector(ID_EX_alu_src)
Output : muxoutput(real_ALU_inB)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다.
  Real_ID_EX_rs2_before와 ID_EX_imm 사이에서 real_ALU_inB signal을 통해 한

값만 내보내주는 조건문 역할을 해준다.

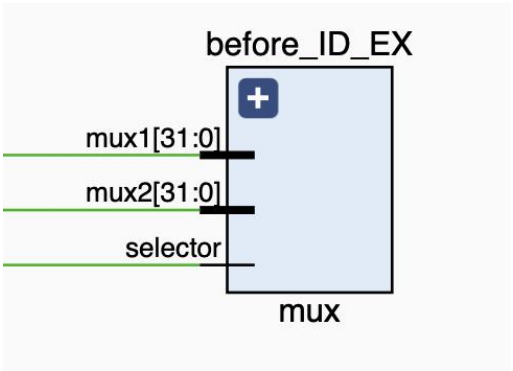
- afterData



Input : mux1 (MEM_WB_mem_to_reg_src1), mux2 (MEM_WB_mem_to_reg_src_2), selector(MEM_WB_mem_to_reg)
Output : muxoutput(real_DataReadData)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. MEM_WB_mem_to_reg_src1과 MEM_WB_mem_to_reg_src_2 사이에서 MEM_WB_mem_to_reg signal을 통해 한 값만 내보내주는 조건문 역할을 해준다.

- Before_ID_EX



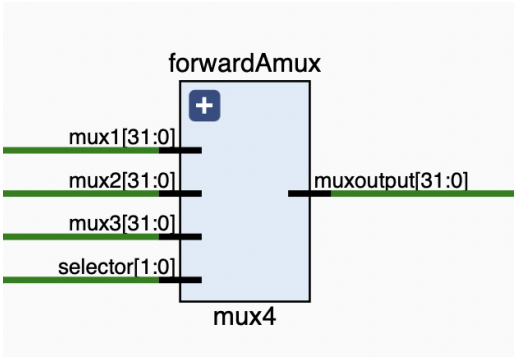
Input : mux1 (control_out), mux2 (0), selector(hazard_out)
Output : muxoutput(real_pipeline_signal)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. Control_out과 0 사이에서 hazard_out selector를 통해 한 값만 내보내주는 조건문 역할을 한다.

3.5. <mux4.v>

mux1과 mux2와 mux3과 selector를 input으로 받아서 selector가 00이면 mux1을, 01이면 mux2를 10이면 mux3를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분들은 다음과 같다.

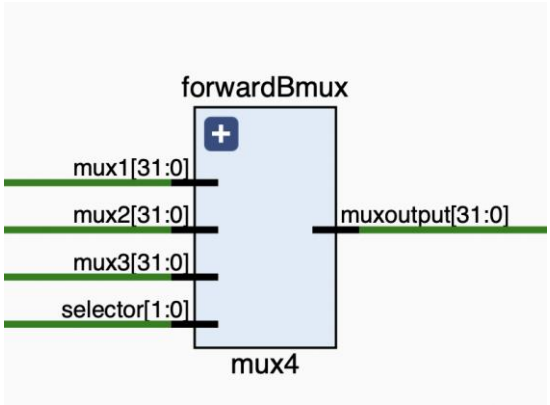
- forwardAmux



Input : mux1 (ID_EX_rs1_data), mux2 (EX_MEM_alu_out), mux3(real_DataReadData), selector(forwardingA)
Output : muxoutput(real_ALU_inA)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. ID_EX_rs1_data와 4와 EX_MEM_alu_out와 real_DataReadData중에서 forwardingA signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

- forwardBmux



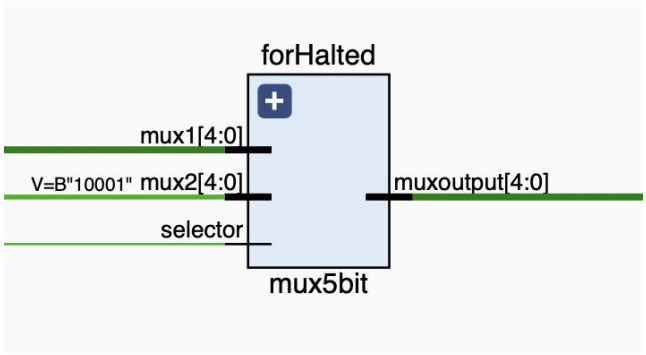
Input : mux1 (ID_EX_rs2_data), mux2 (EX_MEM_alu_out), mux3(real_DataReadData), selector(forwardingB)
Output : muxoutput(real_ID_EX_rs2_before)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. ID_EX_rs2_data와 4와 EX_MEM_alu_out와 real_DataReadData중에서 forwardingB signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

3.6. <mux5bit.v>

5 bit mux1과 5 bit mux2와 selector를 input으로 받아서 selector가 0이면 mux1을, 1이면 mux2를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분은 다음과 같다.

- forHalted

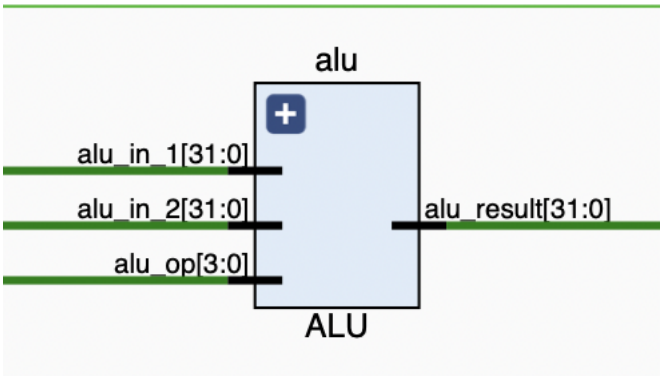


Input : mux1 (IF_ID_inst[19:15]), mux2 (5'b10001), selector(IsEcall)
Output : muxoutput(aftergaltingMuxrs1)

  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. IF_ID_inst[19:15]와 17 중에서 isEcall signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

3.7. <ALU.v>

- ALU



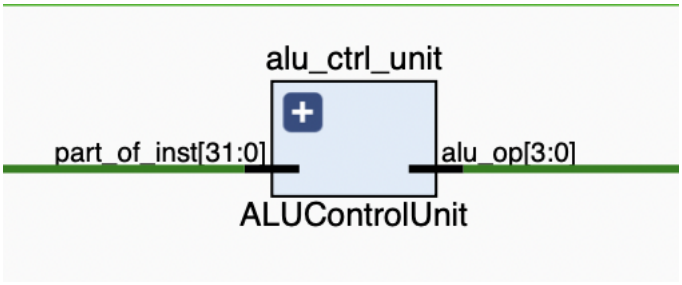
input : alu_in_1, alu_in_2, alu_op
output : alu_bcond, alu_result

- Asynchronous Logic : switch-case문을 이용해 alu_op 4비트에 따라 alu_result의

연산이 달라지도록 구현하였다. 특히 branch instruction의 경우 op 4비트 MSB가 1인 특징을 지니며, BEQ BNE BLT, BGE 의 조건에 따라 alu_bcond가 설정된다. 이들은 asynchronous하게 연산된다.

3.8. <ALUControlUnit.v>

- alu_ctrl_unit

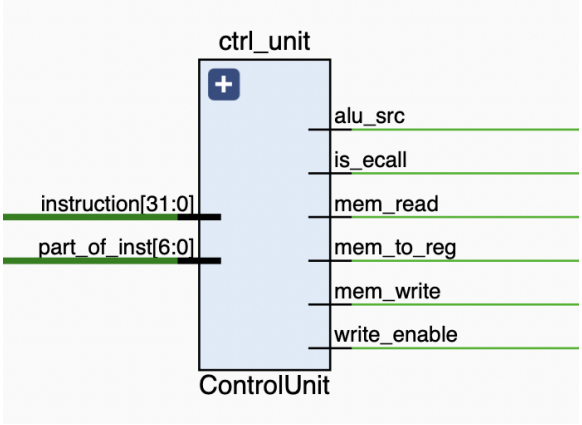


input : part_of_inst

output : alu_op

- Asynchronous Logic : input인 part_of_list가 바뀔 때마다 asynchronous하게 동작하여. 각 part_of_inst에 맞는 alu_op를 대입한다.
- Implementation : part_of_inst를 가공해서 input으로 넣을 수도 있었겠지만, 가공하지 않고 32bit의 whole instruction을 input으로 넣었다.
  우선, opcode에 해당하는 part_of_inst[6:0]의 값에 따라 연산의 종류에 대한 case를 구분했다.
  ARITHMETIC 연산일 때는, ALU 연산에 해당하는 부분인 part_of_inst[14:12]와, part_of_inst[30]을 추가로 검토해 주었다. add와 sub는 funct3이 일치하고, sub 연산만 part_of_inst[30]이 1이기 때문에 따로 구분했다.
  ARITHMETIC_IMM 연산일 때도, ALU 연산 부분은 part_of_inst[14:12]를 사용하여 alu operation을 구분해 주었다.
  LOAD, JALR, STORE 연산일 때의 ALU 연산은 add이다. 왜냐하면 각각 ALU unit에서 immediate value와의 연산 부분이 있기 때문이다. (JAL은 별도의 Add unit을 활용하여 ALU 연산이 존재하지 않는다.)
  BRANCH 연산일 때는 part_of_inst[14:12]를 사용하여 별도로 alu_operation을 구분해 주어야 했다. 그 이유는, ALU unit에서 rs1과 rs2에 따른 branch condition도 계산해야 하기 때문이다.

3.9. <ControlUnit.v>



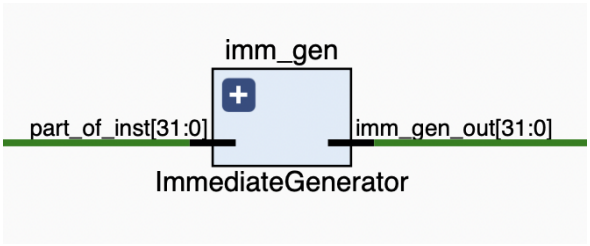
input : instruction, part_of_inst

output : alu_src, is_ecall, mem_read, mem_to_reg, mem_write, write_enable

- Asynchronous logic : input인 instruction이 바뀔 때마다 asynchronous하게 동작하며, 각 instruction에 맞는 control signal을 blocking assignment로 대입한다. 자세한 구현은 lab2의 single-cycle CPU에서의 구현과 일치한다.

3.10. <ImmediateGenerator.v>

- imm_gen



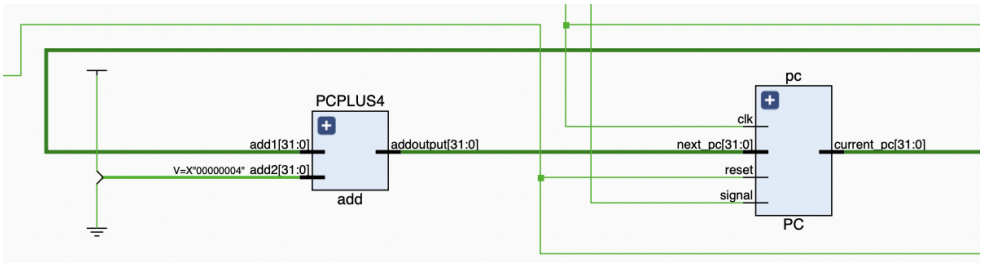
input : part_of_inst

output : imm_gen_out

- Asynchronous Logic : input인 part_of_inst이 바뀔 때마다 asynchronous하게 동작하며, 각 instruction을 해독하여 내부에 있는 immediate value를 찾아낸다.
- Implementation : 우선, opcode인 part_of_inst[6:0]을 통해 명령어의 종류를 구분했다. 각 명령어들에 대하여 RISC-V 규정에 따라 immediate value를 찾을 수 있다. 자세한 값은 Introduction에 있는 1.5의 표를 참조하였다. STORE와 BRANCH, JAL 명령의 경우에는 sign-extension을 해 주었다.

3.11. <PC.v>

- pc



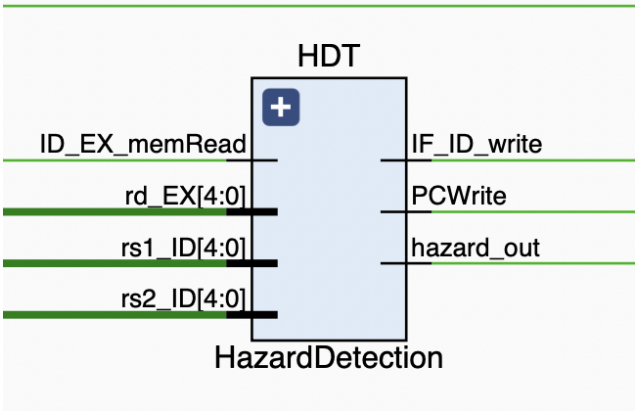
input : clk. next_pc, reset, signal

output : current_pc

- (Positive clk) Synchronous Logic : positive clk일 때마다 우선 reset이 1일 경우 current_pc를 0으로 초기화해준다. 아닐 경우 next_pc(pc+4)를 current_pc에 넣어 준다. 단, 이 과정은 signal (PCWrite)이 1일 때만 시행한다. (여기서 첫번째 clk에도 작동하도록 current_pc == 0 일 때도 pc+4값을 받도록 임의로 설정했다. 이는 Non-controlflow lab이기에 가능한 것이며, 다음 랩에서 수정할 예정이다.)

3.12. <HazardDetection.v>

- HDT



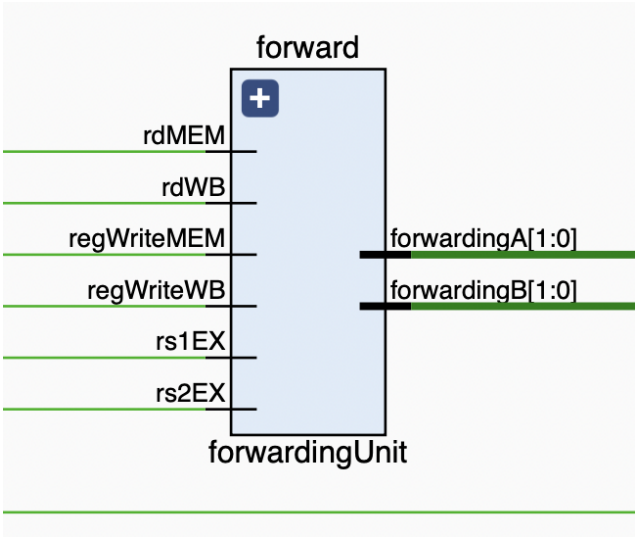
Input : ID_EX_memRead, rd_EX, rs2_ID, rs2_ID

Output : IF_ID_write, PCWrite, hazard_out

  - Asynchronous logic : hazard detection 조건을 만족하는 경우, PCWrite 를 0으로, IF_ID_write를 0으로, hazard out을 1로 설정해 업데이트해준다.

3.13. <ForwardingUnit.v>

- Forward

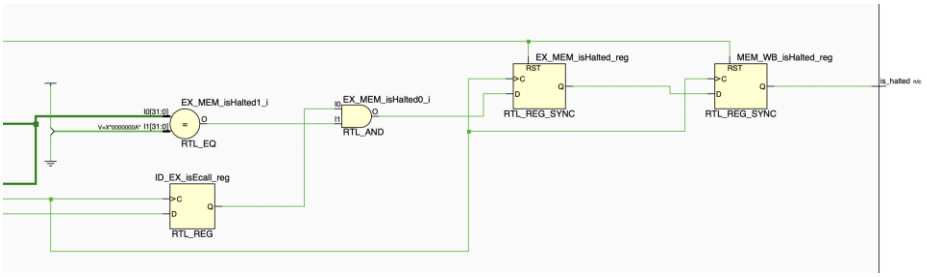


input : rdMEM, rdWB, regWriteMEM, regWriteWB, rs1EX, rs2EX
output : forwardingA, forwardingB

- Asynchronous logic : input 값들의 변화에 따라 asynchronous하게 forwardingA, forwardingB에 값을 넣어준다. 기본적으로 forwardingA,B의 값은 0이다. 자세한 구현은 본 보고서의 2.4에 있다.

4. Discussion

4.1. Data forwarding과 stall



데이터 포워딩을 하지 않으면, Fetching을 ID stage에서 정지시켜야 하지만, 데이터 포워딩을 하면 Fetching을 EXE stage에서 정지시켜야 한다는 차이점이 있다. 왜냐하면, lw 연산은 무조건 MEM stage 이후에 값이 나오는데, 똑같이 데이터 포워딩을 해도 ID stage에서 정지시킨다면 2 stall을 기다려야 하기 때문이다. EXE stage에서 정지시키는 것이 stall을 하나 더 줄일 수 있다. 대신 4.3에서 후술할 복잡한 mux를 사용해야 한다.

4.2. Ecall instruction

Ecall instruction시, rf17의 값을 확인하여 정상적으로 종료되었는지 여부를 파악해야 한다. 이때, Ripes에서는 Ecall일 시 rf17을 포워딩하지 않고 ecall 뒤의 명령어들이 끝날 때까지 기다려주었지만, 우리는 기존의 포워딩 로직을 활용하기 위하여 Ecall시에도 17번째 레시스터에 포워딩 기법을 사용하였다.

4.3. ALU 앞의 mux들간의 경쟁관계 with data forwarding

ALU의 input은 real_ALU_inA, real_ALU_inB이다. real_ALU_inA는 rs1, EX/MEM, MEM/WB

에서의 값을 가져오는 포워딩 기법을 통해 만들어졌다. 하지만, real_ALU_inB 같은 경우에는 포워딩을 한 값과, immediate 중 하나가 될 수 있기 때문에 포워딩 mux 이후 이를 선택해 주는 mux를 하나 더 만들어 주어야 한다.

5. Conclusion

5.1. 수업시간에 배운 5-stage pipelined CPU를 직접 구현하면서 CPU의 동작을 이해하고, 컴퓨터처럼 사고하는 능력을 배울 수 있었다.

5.2. RISC-V를 해독하고 신호를 보내는 부분과, 메모리, 레지스터를 다루는 부분으로 역할을 분담하였다. Hazard Detection (stall)부분과 Forwarding 부분으로 분담하였다.

5.3. 파이프라인 레지스터들이 들어가 회로가 훨씬 더 복잡해졌지만, 명칭의 조건을 정하고 필요한 변수들을 토의를 통해 설정하며 올바르게 구현할 수 있었다.

5.4. Pipelined cpu가 Single cycle cpu 보다 어떤 점이 좋고, 어떻게 성능의 향상이 이루어지는지 직접 구현해보면서 이해할 수 있었다. 이를 위해 어떻게 stall을 하고, 어떻게 forwarding을 하는지 모호했는데, 직접 signal과 파이프라인 레지스터를 만져보며 이해할 수 있었다.