컴퓨터 구조 Lab4-2 Pipelined CPU (with control hazard)

20210207 이지현
20210794 정유진

1. Introduction

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

1.2. Data hazard의 처리는 stall 파트와 data forwarding 파트로 나누어 구현하였고, 이는 랩 4-1에서 구현한 내용이다.

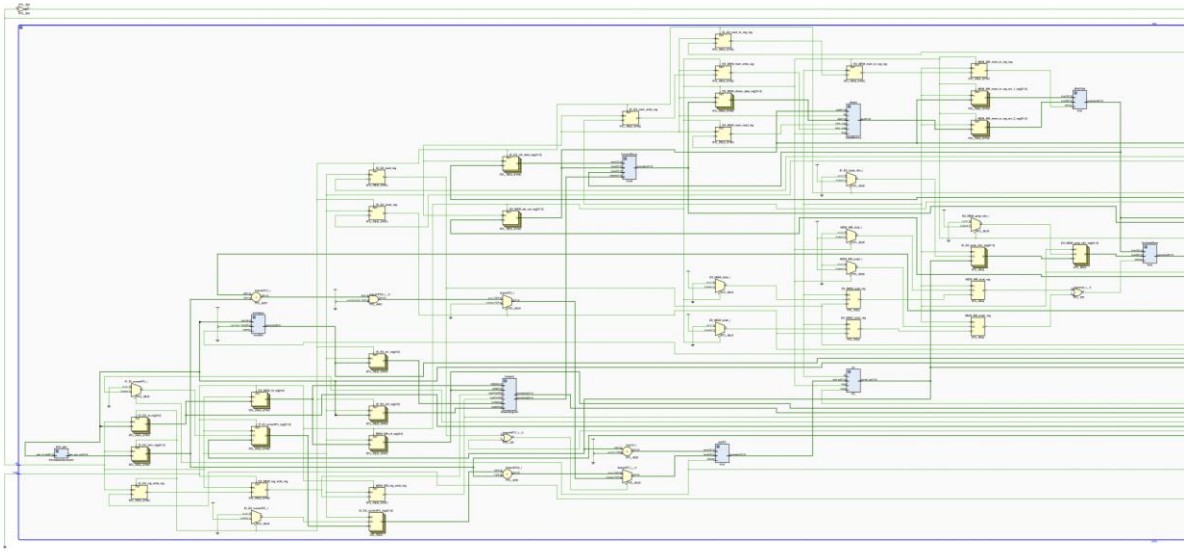
1.3. Control hazard의 처리가 이번 4-2 랩의 주 내용이고, 우리는 not-taken을 베이스로 구현하였다.

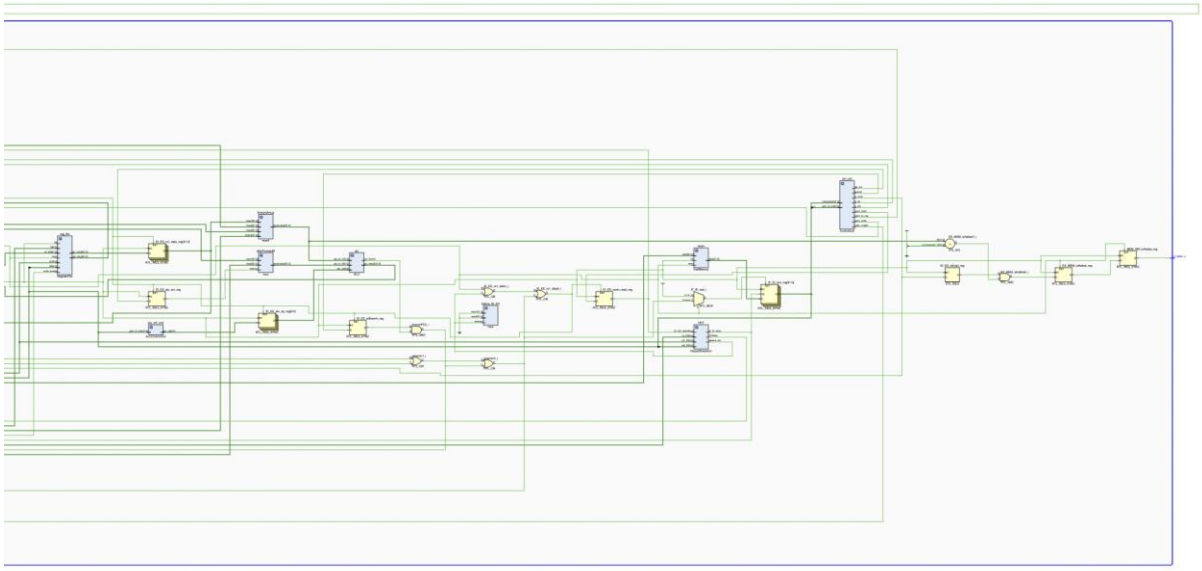
1.4. 주어진 스켈레톤 코드는 top.v, Memory.v, RegisterFile.v, cpu.v, opcodes.v가 있고 추가로 만든 V 파일은 ALU.v, ALUControlUnit.v, ImmediateGenerator.v, ControlUnit.v, PC.v, mux5bit.v, mux.v, mux4.v, HazardDetection, forwardingUnit가 있다.

1.5. 테스트 코드는 basic, ifelse, loop, non-control flow, recursvie_mem 가 있으며, Ripes를 통해 얻은 레지스터 값과 베릴로그 상에서 얻은 레지스터 값과 비교하며 정상적인 동작을 확인할 수 있다.

1.6. 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.
- Control hazard는 EXE stage에서 resolve된다. 즉, 잘못된 prediction에 대한 패널티는 bubble 2개 (flush 2개)가 되도록 설정했다.
- 위와 같은 메커니즘으로 파이프라인 CPU는 작동된다.

2.2. How to handle branch prediction?

- Always not-taken으로 branch predictor를 구현하였다.
- PC 상에서의 bubble

```
always @(*) begin
  if (ID_EX_isJal || ID_EX_isJalr || (ID_EX_isBranch && alu_bcond)) begin
      isTaken = 1;
  end
  else begin
      isTaken = 0;
  end
end
```

따로 isTaken이라는 reg를 만들어서 ID_EX의 inst가 jal, jalr, 혹은 branch instruction이면서 alu_bcond가 1인 경우를 보아 isTaken되었다고 알려주도록 하였다. 이 파라미터를 이용해서 아래의 파이프라인 레지스터와 Mux를 control하였다.

- IF/ID pipeline register 상에서의 bubble

```
// Update IF/ID pipeline registers here
always @(posedge clk) begin
  if (reset) begin
      IF_ID_inst <= 0;
      IF_ID_currentPC <= 0;
  end
  else if(isTaken) begin
      IF_ID_inst <= 0;
  end
  else begin
          if(IF_ID_write == 0) begin
          end
          else begin
              IF_ID_inst <= beforeInst;
              IF_ID_currentPC <= currentPC;
          end
  end
end
```

- isTaken selector가 1일 때 IF_ID_inst를 0으로 만들어서 bubble을 만든다.

- ID/EXE pipeline register 상에서의 bubble

```
// Update ID/EX pipeline registers here
always @(posedge clk) begin
  if (reset || hazard_out || isTaken) begin
      ID_EX_rs1_data <= 0;
      ID_EX_rs2_data <= 0;
      ID_EX_rs1 <= 0;
      ID_EX_rs2 <= 0;
      ID_EX_rd <= 0;
      ID_EX_imm <= 0;
      ID_EX_ALU_ctrl_unit_input <= 0;

      //signal
      ID_EX_alu_op <= 0;
      ID_EX_alu_src <= 0;
      ID_EX_mem_write <= 0;
      ID_EX_mem_read <= 0;
      ID_EX_mem_to_reg <= 0;
      ID_EX_reg_write <= 0;
      ID_EX_isHalted <= 0;
      ID_EX_isEcall <= IsEcall;
      ID_EX_isJal <= 0;
      ID_EX_isJalr <= 0;
      ID_EX_isBranch <= 0;
  end
```

- hazard가 있거나 isTaken이라면 버블을 만들어 총 2개의 bubble이 생성되도록 한다.

- Taken일 때 branchPC의 예상

```
always @(*) begin
 if(ID_EX_isJal || (ID_EX_isBranch && alu_bcond )) branchPC = ID_EX_currentPC + ID_EX_imm;
 else if(ID_EX_isJalr) begin
     branchPC = ID_EX_rs1_data + ID_EX_imm;
     branchPC = branchPC & 32'hFFFFFFFE;
 end
 else branchPC = 0;
```

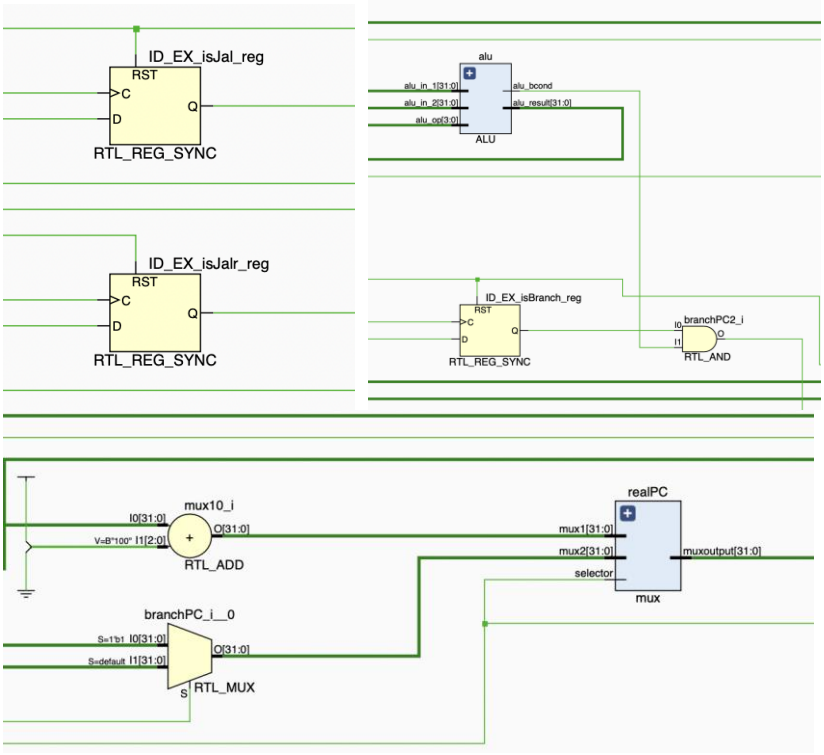
- branchPC를 계산한다. 이때, jal, branch 연산일 경우에는 currentPC에다가 immediate를 더해주고, jalr 연산일 경우에는 레지스터의 값(rs1의 값)에다가 immediate를 더해준 값을 branchPC로 계산한다.

- branchPC와 currentPC+4 사이의 결정 mux

```
mux realPC(
      .mux1(currentPC + 4),
      .mux2(branchPC),
      .muxoutput(nextPC),
      .selector(isTaken)
);
```

- NextPC로 들어가기 전에 currentPC+4 (not-taken)일 때와 branchPC (taken)일 때를 잘 결정해서 isTaken selector로 결정한다.

2.3. Describe our design of branch predictor.



- ID_EX_isJal 과 ID_EX_isJalr, 그리고 ID_EX_isBranch와 alu_bcond를 가져와서 isTaken signal을 만들고 해당 시그널로 realPC signal을 조절하는 모습을 볼 수 있다.
- 해당 파라미터와 프로세스를 통해 branch predictor를 디자인하였다.

2.4. Compare total cycles of always-taken : Ripes / Velioge

- basic ripese : 36 / Verilog : 34 cycles
- ifelse ripese : 44 / Verilog : 42 cycles
- loop ripese : 323 / Verilog : 321 cycles
- non-controlflow ripese : 323 / Verilog : 321 cycles
- recursive ripes : 1188 / Verilog : 1186 cycles

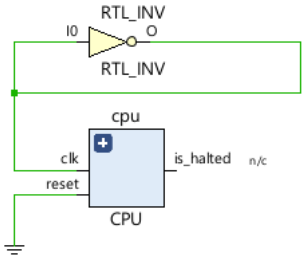
2.5. 다음은 각각의 모듈에 대한 **synchronous/asynchronous 디자인의** 포괄적인 설명이다.

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.

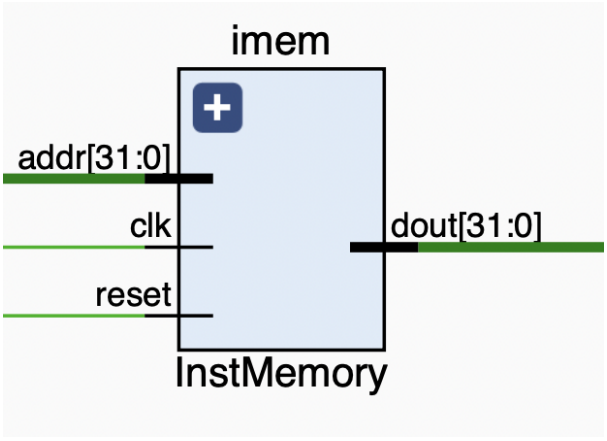
PC	nextPC : 다음 clock의 PC currentPC : 현재 PC real_currentPC : mux를 거치고 난 현재 PC beforeInst : Instruction memory를 나온 dout
ALU	alu_op : 결정된 ALU opreation real_ALU_inA : ALU에 들어가는 최종 첫 번째 operand real_ALU_inB : ALU에 들어가는 최종 두 번째 operand alu_bcond : 여기선 항상 0 = alu_zero ALU_out : ALU output
register	WriteRegister : 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 real_real_DataReadData : 2번째 mux를 거지고 난 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.
Branch predictor	branchPC : taken했을 때 뛰어야하는 PC값을 저장한다. isTaken : taken 여부를 설정하는 핵심 파라미터이다.
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 ID_EX_isJal ID_EX_isJalr ID_EX_isBranch 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 EX_MEM_jump_rdin 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 MEM_WB_isJal MEM_WB_isJalr

3.2. <Memory.v>

- Instruction Memory

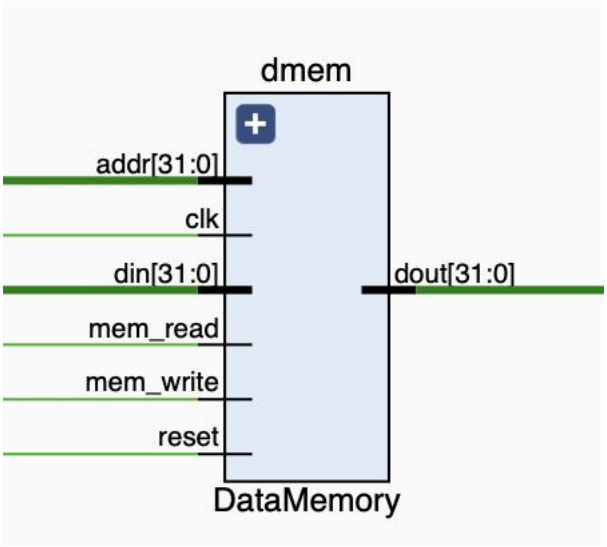


Input : reset, clk, addr

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

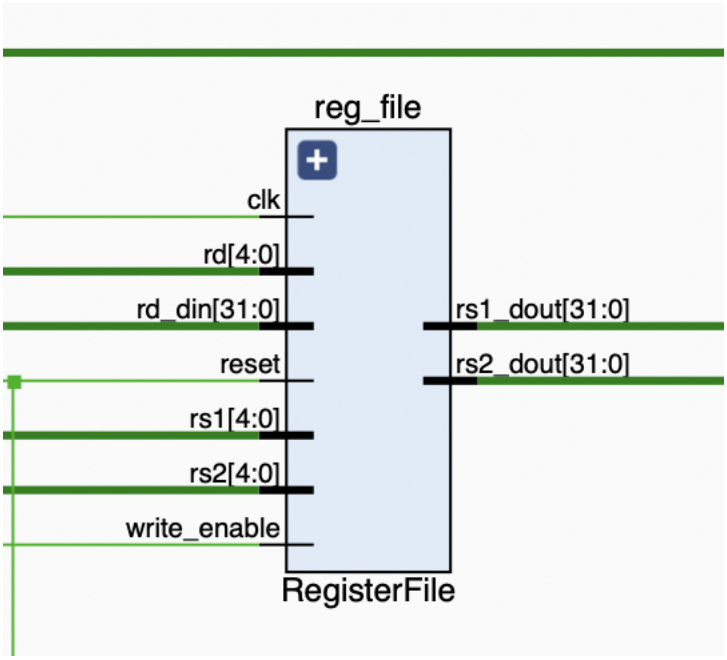
- Data Memory



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.

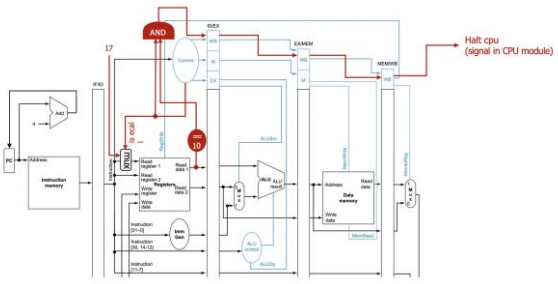
3.3. <RegisterFile.v >



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.

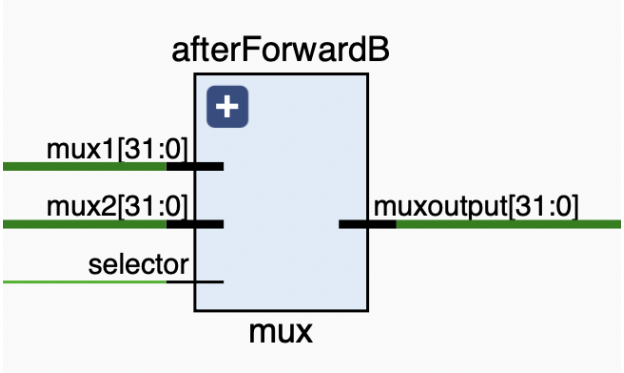


- Rs1앞에서 mux를 달고 17과 경쟁하는데, 이는 isEcall signal에 의해 결정된다. 이 Rs1 레지스터 값이 10과 같으면 파이프라인 레지스터 is_halted에 전달되어 최종적으로 ishalted가 도출된다.

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

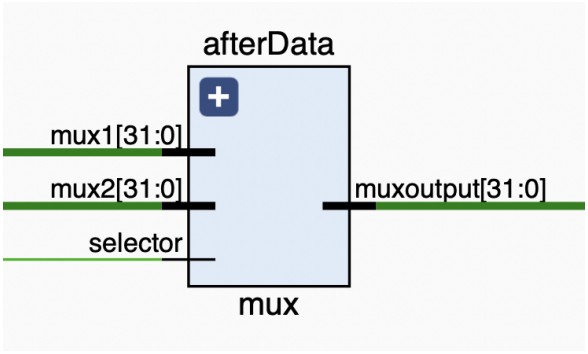


Input : mux1 (real_ID_EX_rs2_before), mux2 (ID_EX_imm), selector(ID_EX_alu_src)
Output : muxoutput(real_ALU_inB)

- 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.

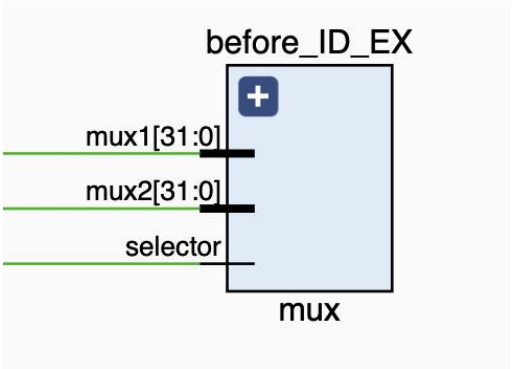
- afterData



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.

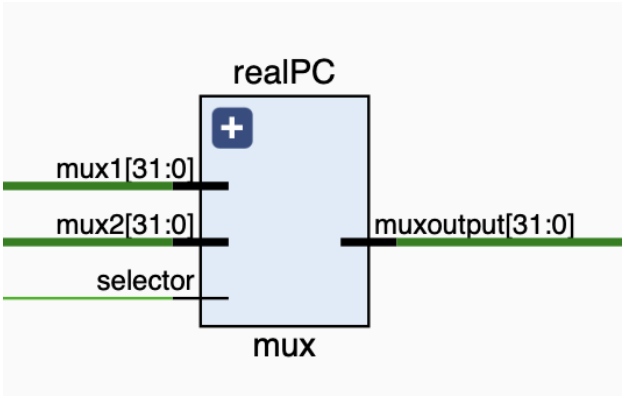
- Before_ID_EX



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.

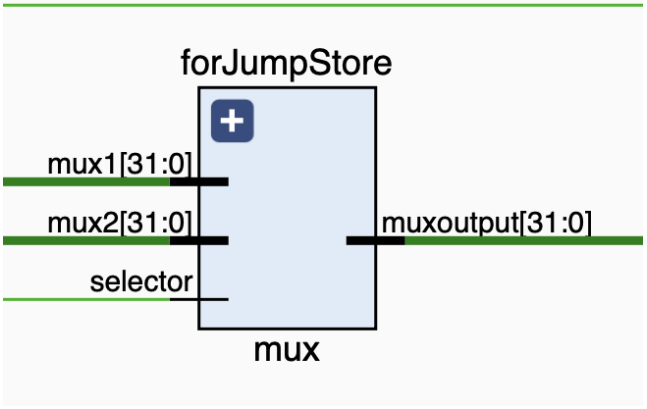
- realPC



Input : mux1 (currentPC+4), mux2(branchPC), selector(isTaken)
Output : muxoutput (nextPC)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. current+4과 branchPC사이에서 isTaken selector를 통해 한 값만 내보내주는 조건문 역할을 한다.

- forJumpStore



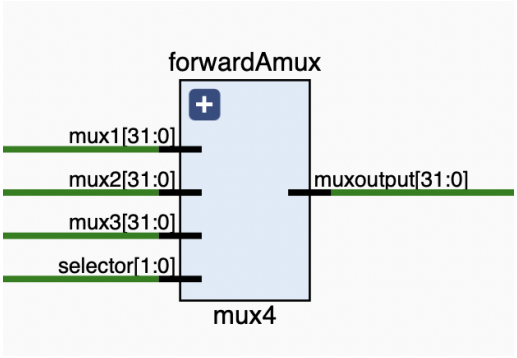
Input : mux1 (real_DataReadData), mux2(EX_MEM_jump_rdin),
selector(MEM_WB_isJal || MEM_WB_isJalr)
Output : muxoutput (real_real_DataReadData)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. Real_DataReadData과 EX_MEM_jump_rdin사이에서 MEM_WB_isJal || MEM_WB_isJalr 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.

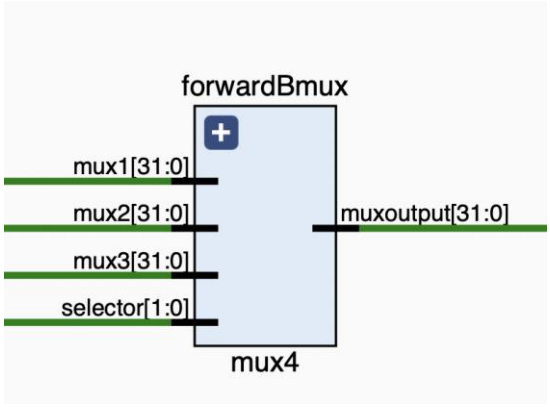
- forwardAmux



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)

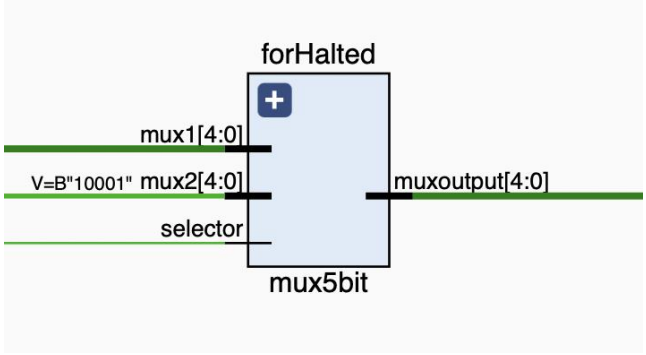
  - Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. ID_EX_rs2_data와 4와 EX_MEM_alu_out와 real_DataReadData중에서 forwardingB signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

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

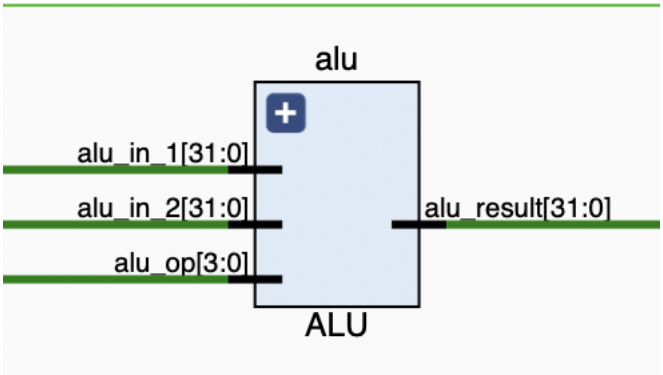


Input : mux1 (IF_ID_inst[19:15]), mux2 (5'b10001), selector(IsEcall)
Output : muxoutput(aftergaltingMuxrs1)

  - 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

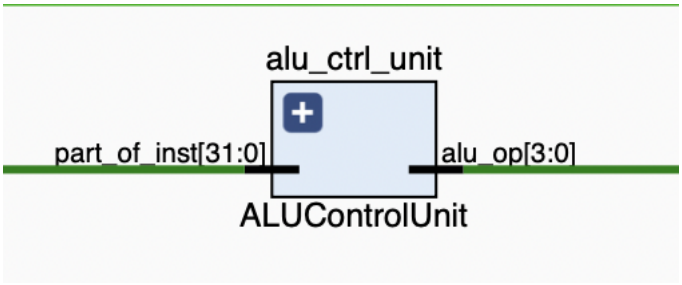


input : alu_in_1, alu_in_2, alu_op
output : alu_bcond, alu_result

- 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

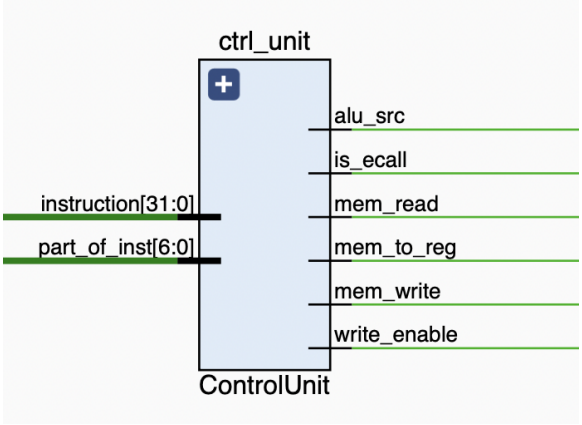


input : part_of_inst

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>



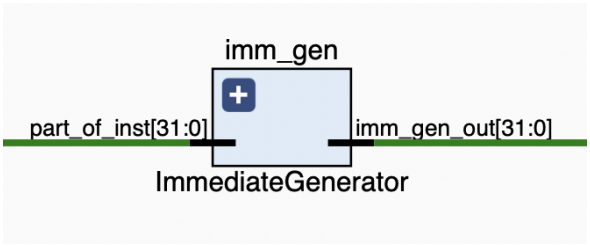
input : instruction, part_of_inst

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



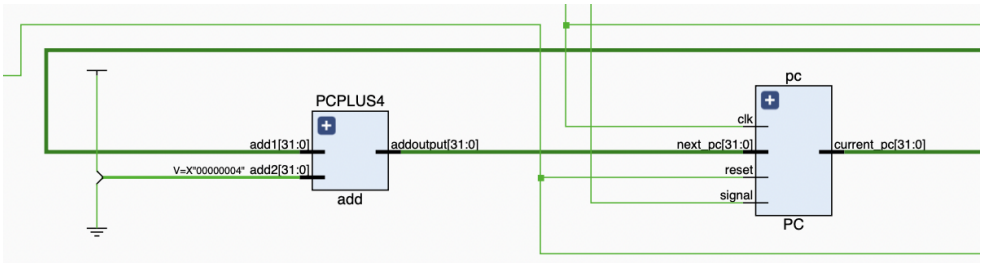
input : part_of_inst

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



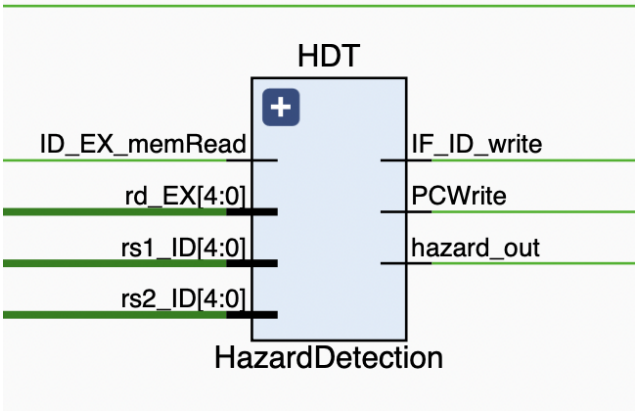
input : clk. next_pc, reset, signal

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 넣어준다. 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



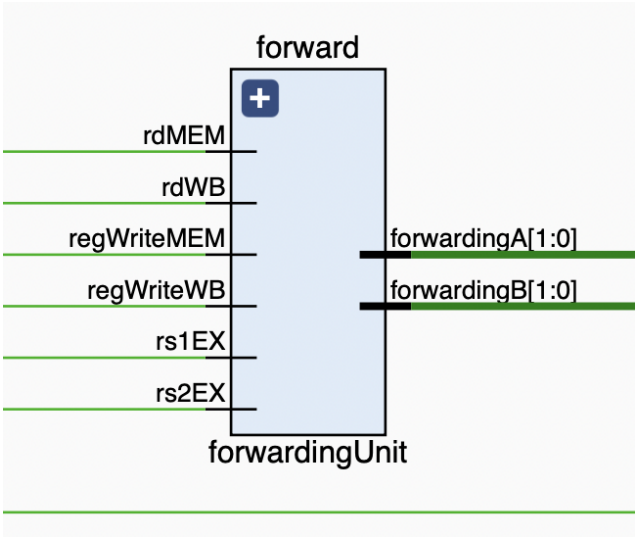
Input : ID_EX_memRead, rd_EX, rs2_ID, rs2_ID

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. Jal, Jalr instruction의 다음 pc가 레지스터에 저장되지 않는 상황

- 파이프라인으로 넘어오면서 control flow를 처음 다루다 보니 점프 후에 다음 instruction이 레지스터에 저장되지 않으면서 해당 로직을 사용하는 recursive_mem에서 오작동을 함을 확인할 수 있었다.
- 이를 해결하기 위해 isJal, isJalr에 대한 파이프라인 레지스터를 구현하고, register in data 이전에 mux를 하나 더 만들어 두 경우 다음 pc를 저장하도록 구현하였다.
-
```
EX_MEM_isJal <= ID_EX_isJal;
EX_MEM_isJalr <= ID_EX_isJalr;
```
-
```
MEM_WB_isJal <= EX_MEM_isJal;
MEM_WB_isJalr <= EX_MEM_isJalr;
```
-
```
mux forJumpStore(
    .mux1(real_DataReadData),
    .mux2(EX_MEM_jump_rdin),
    .muxoutput(real_real_DataReadData),
    .selector(MEM_WB_isJal || MEM_WB_isJalr)
);
```

4.2. isTaken일 때의 IF_ID_inst 불러오기와 관련

isTaken일 때는 pipeline을 flush하는 것이므로, control 신호를 0으로 해 주어야 한다. 또한, IF stage에서도 instruction을 0으로 해 주어야 하는데, 이는 bubble일 때 잘못된

instruction이 불려서 결과에 영향을 끼치지 못하게 하기 위함이다.

5. Conclusion

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

5.2. 파이프라인 레지스터들을 업데이트하는 부분에 대한 조건이 달라졌고, EX stage에서 resolve되도록 구현을 해야한다는 조건이 지정되어있었기에 구현을 하는데 있어서 제한적이라고 느껴졌다.

5.3. BTB를 이용한 always-Taken을 구현하려 했지만, 프로그램이 끝나는 지점에서 PC가 제한된 부분보다 크게 읽혀오는 오류와, 그 이후 PC가 음수가 되는 오류가 발생하였다. 이를 고치는 데 실패하였다.