컴퓨터 구조 Lab5

20190782 컴퓨터공학과 최서영
20190439 컴퓨터공학과 오승훈

Introduction

이번 Lab은 Pipelined cpu를 구현하는 것이다. Pipelined cpu는 multicycle cpu와 다르게 여러 개의 instruction이 동시에 진행된다. 기본적으로 모든 instruction이 5 stage(IF, ID, EX, MEM, WB) 모두 사용한다고 가정하고 설계되며, 매 cycle마다 하나의 instruction이 읽히게 된다. 이때, instruction에 따라 stall 혹은 flush가 발생하게 된다. 또한 이렇게 발생하는 flush 및 stall을 줄이기 위해 forwarding 기술도 사용하게 된다. 이번 랩에서는 stall, flush, forwarding 기술을 접목한 pipelined cpu를 설계하고, control hazard 부분에서 always not taken, always taken, 2-bit global prediction을 이용한 cpu 3개를 비교해보고자 한다.

Design

이번 lab에서는 adder, alu, alu_control_unit, control_unit, cpu, datapath, register_file, util, sign_extend을 제외하고 stall을 해야 하는지를 결정하는 hazard, forwarding을 해야 하는지를 판단하고 어떤 data를 써야하는 지를 결정하는 forwarding_unit, always taken과 2-bit global prediction에서 사용되는 branch_predictor 모듈이 추가되어 회로를 구성하고 있다.

control_unit의 경우 clock의 posedge에서 input으로 들어온 opcode와 func_code(instruction의 정보)를 이용하여 해당 instruction에 알맞은 signal들을 결정하고, 해당 signal을 output으로 내보내 준다. 이때, 만약 flush나 혹은 stall등의 이유로 input으로 들어오는 instruction이 z값으로 들어온다면 read_m1(instruction을 읽어오는 signal)을 제외하곤 모든 signal을 0으로 초기화 시켜준다.

MUX2_to_1와 MUX4_to_1은 기존의 mux의 기능이 같다. 이러한 mux들은 memory를 읽을 때 어떤 address를 사용할지, WB state에서 register에 update할 data는 무엇일지, ALU에 들어갈 input을 결정할 때, pc값을 update할 때 어떤 값으로 update할지 등을 결정할 때 사용된다. 이러한 mux에 서 값을 고르는 signal은 control_unit, forwarding unit 등 다른 모듈에서 오게 된다.

Alu_control_unit은 alu에 넣어주는 funcCode, opcode, branchType을 결정해주는 module이다. 이때, input으로 받는 opcode와 funct(각 instruction의 정보)에 맞는 funcCode를 set하고, opcode를 이용하여 branch인 instruction을 걸러내어 branchTyp과 funcCode를 set해준다.

Register_file은 register 값들을 불러오고 수정할 수 있도록 구현한 모듈이다. 값을 불러오는 경우 rs, rt에 해당하는 port에 들어온 번호에 맞게 register 내부의 값을 output으로 보내주고 값을

수정하는 경우 rd의 값에 대응하는 register에 값을 대입한다. 이때, register 값을 수정하는 것은 오직 negative edge clock일 때, reg_write signal이 1일때만 수정이 가능하다.

Sign_extend는 immediate 값의 bit수를 맞춰주기 위한 모듈로 imm 값을 input으로 받아 Imm_extend를 output으로 내놓고 이 값을 ALU의 input을 결정해주는 MUX로 들어간다.

Adder는 말 그대로 input으로 들어간 두개의 값을 더해주는 역할을 한다.

ALU는 기존의 ALU 연산과 더불어 bcond를 판단하는 기능을 수행한다. ALU는 combinational logic이고, non-blocking으로 계산된다. input으로 들어온 data 2와 func_code, branch_type을 이용하여 input에 알맞은 output을 결정한다.

Hazard는 해당 instruction이 stall이 발생하는지를 확인하여 output으로 stall이 발생하면 1, 아니면 0을 내보 내준다. ID_EX에서 memory를 접근하는 load 혹은 store 일 때, IF_ID 단계에서 사용하는 register와 겹치는지를 확인하여 겹친다면 stall이 필요하다.

Forwarding_unit에서는 ID_EX에서 사용하려고 하는 register가 뒤의 단계(EX_MEM or MEM_WB)에서 WB을 하려고 하는 register와 같은지를 판별하여 forwarding이 필요한지를 판단하고, 필요하다면 어떤 단계의 data를 사용해야 하는지를 판단하는 모듈이다.

Datapath은 control_unit으로부터 signal을 받아 위에서 소개한 모듈들을 연결하고, 각 stage의 data를 저장하여 각 stage에 필요한 데이터를 전달해주는 역할을 한다. 이때, 데이터는 signal과 instruction, register 값 등이 있다. 또한 stall이나 flush가 발생해야 하는지를 확인하고, 해야 한다면 그에 맞는 data와 signal을 setting하여 다음 stage로 보내는 역할을 한다.

Branch_predictor는 다음 instruction을 받아올 때, 어떤 주소의 instruction을 받아야 하는지를 결정하는 모듈이다. Always_not_taken에서는 이 모듈을 사용하지 않고 항상 PC+1의 instruction만 받아오며, 만약 잘못 받아오게 되면 flush를 발생시키고 알맞은 PC값으로 업데이트하여 다시 instruction을 받아오도록 하였다. Always_taken과 2-bit global prediction에서는 해당 모듈을 사용하게 된다. Always_taken에서는 현재 PC가 BTB_table에서 hit이면서, jump 혹은 branch라면, table에 저장되어 있는 PC값을 다음 PC값으로 반환하고, 아닌 경우엔 PC+1 값을 다음 PC 값으로 반환한다. 2-bit global prediction에서는 현재 PC가 BTB_table에서 hit일 때, jump면 table에 저장된 값을, branch라면 hypothesis counter를 이용하여 taken인지 not taken인지를 확인해서 taken인 경우만 table에 저장된 값을 반환한다.

Pipelined CPU는 multicycle CPU와 다른 점이 많다. 먼저, Pipelined CPU는 여러 instruction이 동시 진행하여 하나의 instruction만 수행하는 multicycle CPU보다 더 빠르게 동작할 수 있다. 하지만, 계산을 모두 진행하고 다음 instruction을 수행하는 것이 아니기 때문에 RAW나 잘못된 instruction을 수행하는 경우가 존재한다. 때문에 이를 해결하기 위해 forwarding 및 stall과 flush가 존재한다. 또한 multicycle CPU는 각 instruction이 불필요한 stage를 수행하지는 않았지만, Pipelined CPU는 기본적으로 모든 instruction이 5 stage를 모두 진행하게 된다.

Implementation

이번 구현에 있어서는 always not taken, always taken, 2-bit prediction 이렇게 총 3가지를 구현을 해야했기 때문에 각각의 특징에 따라서 cpu top module과 branch_prediction module이 차이가 존재한다. 그리고 또한 현재 cpu는 data forwarding이 이미 완료된 상태로 구현이 되어있다.

A. Always not taken

a. CPU top module

Always not taken에서는 기본적으로

```
initial begin
        pc_value <= 16'h0000;
end

control_unit CONTROL(reset_n, bcond, control_unit_data[15:12], control_unit_data[5:0], clk, pc_write_cond, pc_write, i_or_d, mem_to_reg, ir_write,
pc_to_reg, pc_src, halt, wwd, reg_write, read_m1, con_read_m2, con_write_m2, alu_src, if_id_mux_con, jalr, id_ex_jalr);

datapath DATA(clk, reset_n, read_m1, address1, data1, read_m2, write_m2, address2, data2, num_inst, output_port, is_halted, pc_value_next, pc_write_cond, i_or_d, mem_to_reg, ir_write,
pc_to_reg, pc_src, halt, wwd, reg_write, con_read_m2, con_write_m2, control_unit_instruction, alu_src, bcond, if_id_mux_con, if_id_pc_next, is_stall, jalr, id_ex_jalr, pc_value_alu_out);

always @(posedge clk) begin
        if(!reset_n) begin
                pc_value <= pc_value;
        end
        else if(id_ex_jalr) begin
                pc_value <= pc_value_alu_out;
        end
        else begin
                if(is_stall) begin
                        pc_value <= pc_value;
                end
                else begin
                        if(pc_src || (pc_write_cond && bcond)) begin
                                pc_value <= if_id_pc_next;
                        end
                        else begin
                                pc_value <= pc_value_next;
                        end
                end
        end
end
ays not taken pc_value update
assign control_unit_data = (is_stall) ? control_unit_instruction : data1;
assign address1 = pc_value;
```

Control unit과 datapath만 존재한다는 것이 기존 multicycle이나 single cycle cpu와 동일한 구조이며, pc값은 항상 pc + 1이 되지만, stall, flush가 발생하는 상황 또는 jump인 경우 pc 값이 다르게 들어가도록 구현을 해뒀다.

b. Branch prediction

Always not taken은 따로 branch prediction 모듈이 필요가 없기 때문에 구현이 되어 있지 않다.

B. Always taken

a. CPU top module

```
branch_predictor PREDICTOR(data1, reset_n, pc_value, is_flush, is_BJ_type, actual_next_PC, actual_PC - 1 , next_PC, update_need);

assign is_BJ_type = ((data1[15:12] < 4) || (data1[15:12] == 9 || data1[15:12] == 10)
        || (data1[15:12] == 15 && (data1[5:0] == 25 || data1[5:0] == 26)));
assign update_need = (id_ex_jalr && !(jalr_check)) || (!(b_j_check) && is_BJ_type_update);
assign is_flush = (id_ex_jalr && !(jalr_check)) || (!(b_j_check) && is_BJ_type_update) || is_stall;

assign actual_next_PC = id_ex_jalr ? pc_value_alu_out : if_id_pc_next;
assign actual_PC = id_ex_jalr ? id_ex_pc_reg : if_id_pc_reg;

always @(posedge clk) begin
        if(!reset_n) begin
                pc_value <= pc_value;
        end
        else begin
                if((id_ex_jalr && !(jalr_check))) begin
                        pc_value <= pc_value_alu_out;
                end
                else if((!(b_j_check) && is_BJ_type_update)) begin
                        pc_value <= if_id_pc_next;
                end
                else begin
                        pc_value <= next_PC;
                end
        end
end
```

위와 같이 always not taken보다 datapath와 controlpath에서 몇 개의 추가적인 signal들이 조금 더 나오게 되고, pc 값 update를 하는 방법이 다르며, branch prediction이 구현이 되어 있다. 앞서 말한 signal은 control path에서 언급을 할 예정이다.

b. Branch prediction

```
module branch_predictor(data1, reset_n, PC, is_flush, is_BJ_type, actual_next_PC, actual_PC, next_PC, update_need);

        input [15:0]data1;
        input reset_n;
        input [`WORD_SIZE-1:0] PC;
        input is_BJ_type;
        input [`WORD_SIZE-1:0] actual_next_PC; //computed actual next PC from branch resolve stage
        input [`WORD_SIZE-1:0] actual_PC; // PC from branch resolve stage
        input update_need;
        input is_flush;
```

다음과 같이 모듈이 정의가 되어있고

```
always @(*) begin
        if(is_flush) begin
                next_PC = PC;
        end
        else if(is_BJ_type == 1) begin
                if(BTB_tag[PC[3:0]] === PC[15:4]) begin
                        next_PC = BTB_next_pc[PC[3:0]];
                end
                else begin
                        next_PC = PC + 1;
                        BTB_tag[PC[3:0]] = PC[15:4];
                end
        end
        else begin
                next_PC = PC + 1;
        end
end
```

여기서 보는 바와 같이 16bit 중 아래 4개의 bit을 index bit으로 사용하고, 나머지 상위 12개 bit를 tag bit으로 사용하여 만든 BTB_tag라는 것을 만들어두고, 현재 fetch 되고 있는 instruction이 branch or jump type인 경우에, index에 있는 tag가 hit이면 BTB_next_pc에 저장되어 있는 값을 반환을 하고, 아닌 경우에는 PC + 1 값을 넣도록 구현을 해뒀다. 그리고 branch or jump type이 아닌 경우에는 pc + 1 값을 다음 pc 값으로 가지도록 구현을 해뒀다.

```
if(update_need) begin
        BTB_next_pc[actual_PC[3:0]] = actual_next_PC;
end
else begin
        BTB_next_pc[actual_PC[3:0]] = BTB_next_pc[actual_PC[3:0]];
end
```

BTB_next_pc는 위와 같이 update가 필요한 상황 즉 jal, jalr, branch인 경우 실제 다음 pc 값을 넣도록 만들어서 구현을 해뒀다.

C. 2-bit prediction

a. CPU top module

CPU top module의 경우 always taken과 동일하게 구현이 되어있다. 하지만 assign이 되는 값들이 조금 다르다는 것을 확인이 가능하다. 대표적으로 check_predict가 추가적으로 구현이 되어 있다. Check_predict의 경우 현재 branch가 taken인지 not taken인지를 가져와서 branch_predictor의 2-bit의 state를 변경해주기 위해서 사용이 된다.

```
control_unit CONTROL(reset_n, bcond, control_unit_data[15:12], control_unit_data[5:0], clk, pc_write_cond, pc_write, i_or_d, mem_to_reg, ir_write,
pc_to_reg, pc_src, halt, wwd, reg_write, read_m1, con_read_m2, con_write_m2, alu_src, if_id_mux_con, jalr, id_ex_jalr, is_BJ_type_update);

datapath DATA(clk, reset_n, read_m1, address1, data1, read_m2, write_m2, address2, data2, num_inst, output_port, is_halted, pc_value_next,
pc_write_cond, i_or_d, mem_to_reg, ir_write, pc_to_reg, pc_src, halt, wwd, reg_write, con_read_m2, con_write_m2, control_unit_instruction,
alu_src, bcond, if_id_mux_con, if_id_pc_next, is_stall, jalr, id_ex_jalr, pc_value_alu_out, is_BJ_type_update,
id_ex_is_BJ_type_update, b_j_check, jalr_check, if_id_pc_reg, id_ex_pc_reg);

branch_predictor PREDICTOR(data1, reset_n, pc_value, is_flush, is_BJ_type, actual_next_PC, actual_PC - 1 , next_PC, update_need, check_predict);

assign is_BJ_type = ((data1[15:12] < 4) || (data1[15:12] == 9 || data1[15:12] == 10)
        || (data1[15:12] == 15 && (data1[5:0] == 25 || data1[5:0] == 26)));
assign update_need = pc_src && (!jalr) || bcond || id_ex_jalr;
assign is_flush = (id_ex_jalr && !(jalr_check)) || (!(b_j_check) && is_BJ_type_update) || is_stall;

assign check_predict = pc_write_cond;
assign actual_next_PC = id_ex_jalr ? pc_value_alu_out : if_id_pc_next;
assign actual_PC = id_ex_jalr ? id_ex_pc_reg : if_id_pc_reg;

always @(posedge clk) begin
        if(!reset_n) begin
                pc_value <= pc_value;
        end
        else begin
                if((id_ex_jalr && !(jalr_check))) begin
                        pc_value <= pc_value_alu_out;
                end
                else if((!(b_j_check) && is_BJ_type_update)) begin
                        pc_value <= if_id_pc_next;
                end
                else begin
                        pc_value <= next_PC;
                end
        end
end
```

b. Branch prediction

먼저 2-bit prediction에서 사용된 predictor는 hypothesis counter를 이용하여 구현을 하였고, 그래서 상위 bit가 1인 경우 taken이고, 상위 bit가 0인 경우 not-taken을 하도록 하는 구현을 해뒀다.

```
always @(*) begin
        if(is_flush) begin
                next_PC = PC;
        end
        else if(is_BJ_type == 1) begin
                if(BTB_tag[PC[3:0]] === PC[15:4] && data1[15:12] < 4) begin // hit and branch
                        if(BTB_2_bit[1]) begin
                                next_PC = BTB_next_pc[PC[3:0]];
                        end
                        else begin
                                next_PC = PC + 1;
                                BTB_next_pc[PC[3:0]] = PC + 1;
                        end
                end
                else if(BTB_tag[PC[3:0]] === PC[15:4]) begin // hit and jump
                        next_PC = BTB_next_pc[PC[3:0]];
                end
                else begin // miss
                        next_PC = PC + 1;
                        BTB_tag[PC[3:0]] = PC[15:4];
                        BTB_next_pc[PC[3:0]] = PC + 1;
                end
        end
        else begin
                next_PC = PC + 1;
        end
end
```

위의 always taken과의 가장 큰 차이점은 branch인 경우와 jump인 경우를 구분해서 pc 값을 다르게 내줘야 한다는 점이다. Jump인 경우 hit이 되는 경우에 한해서는 BTB_next_pc에 있는 값을 다음 pc 값으로 보내주면 되고, branch의 경우 hit이면서, saturation counter인 BTB_2_bit의 상위 비트가 1인 경우에는 BTB_next_pc에 있는 값을 내어주고, 아닌 경우에는 PC + 1 값을 다음 pc 값으로 넣어준다는 특징을 가진다.

```
always @(*) begin
        if(check_predict) begin // branch
                if(update_need) begin
                        if(BTB_2_bit == 3) begin
                                BTB_2_bit = 3;
                        end
                        else if(BTB_2_bit == 2) begin
                                BTB_2_bit = 3;
                        end
                        else if(BTB_2_bit == 1) begin
                                BTB_2_bit = 3;
                        end
                        else begin
                                BTB_2_bit = 1;
                        end
                        BTB_next_pc[actual_PC[3:0]] = actual_next_PC;
                end
                else begin
                        if(BTB_2_bit == 3) begin
                                BTB_2_bit = 2;
                        end
                        else if(BTB_2_bit == 2) begin
                                BTB_2_bit = 0;
                        end
                        else if(BTB_2_bit == 1) begin
                                BTB_2_bit = 0;
                        end
                        else begin
                                BTB_2_bit = 0;
                        end
                end
        end
        else if(update_need) begin
                BTB_next_pc[actual_PC[3:0]] = actual_next_PC;
        end
        else begin // branch else
                BTB_2_bit = BTB_2_bit;
        end
end
```

그리고 또한 check_predict 는 branch 인지 아닌지를 판별해주는 bit로 branch 인 경우에 update_need bit이 1인 경우에는 현재 branch가 taken인 경우이므로, hypothesis counter의 값을 값에 맞게끔 변경(값에 맞게 증가) 시켜주고, 0인 경우에는 현재 branch가 not taken인 경우이므로, hypothesis counter의 값을 값에 맞게끔 변경(값에 맞게 감소) 시켜준다.

D. Other module

다른 모듈들은 이제 모든 case에 있어서 동일하다. Register file과 sign_extend, util, adder, alu_control_unit은 모두 multi cycle cpu와 동일한 코드를 사용하였다.

a. Datapath

Datapath의 멀티사이클과 가장 큰 차이점은 IF, ID, EX, MEM, WB stage 각각의 instruction과 pc 값들을 레지스터에 저장을 해두고, 각 스테이지에서 필요한 동작을 진행해 줘야한다는 점이다.

1) IF stage

IF stage는 instruction을 읽어오기만 하면된다. 이 instruction은 Memory에서 현재 해당하는 pc 값을 읽어오면 되고, cpu top module에서 data1으로 존재한다. 그리고 이 data1은 negedge에 들어온다는 것을 특징으로 한다. 그리고 stall이나 flush가 일어난 경우에는 pc 값을 증가를 시키지 않는 것으로 stall과 flush를 구현하였다. 이는 branch prediction module에서 구현이 되어 있다.

2) ID stage

ID stage에 진행해야하는 연산이 좀 많은데

현재 ID stage에 들어가는 instruction에 대한 control unit을 먼저 가져와야한다.

```
assign control_unit_data =  (is_stall) ? control_unit_instruction : (((id_ex_jalr && !(jalr_check)) || (!(b_j_check) && is_BJ_type_update)) ? 16'bz : data1);
assign address1 = pc_value;
```

하지만 control_unit에 들어가는 data 값은 data1이 될 때도 있고, stall이 되는 경우에는 현재 ID stage에 사용되기 위해서 IF/ID register에 저장되어 있는 instruction일 때도 있기 때문에 상황에 맞춰서 control unit에 들어가는 instruction을 결정해주면 된다. 그리고 또한 flush가 되는 경우에는 control signal이 필요가 없어지기 때문에 16'bz를 넣어주면 된다.

```
always @(posedge clk) begin
        if (opcode == 4'd15) begin
                pc_write_cond <= 1'b0; //branch then 1
                pc_write<= 1'b1; //if pc_wirte == 1 then pc update, fetch and jump then 1

                read_m1 <= 1'b1;
                read_m2 <= 1'b0;
                write_m2 <= 1'b0;
                alu_src <= 1'b0;

                reg_write <= 1'b0;

                if(func_code == 6'd25) begin //JPR
                        halt <= 1'b0;
                        wwd <= 1'b0;
                        mem_to_reg <= 2'b00;
                        reg_write <= 1'b0;
                        if_id_mux_con <= 2'b10;

                        pc_src <= 1'b1;
                        pc_to_reg <= 1'b0;
                        jalr <= 1'b1;
                        is_BJ_type_update <= 1'b0;
                end
```

Control unit은 다음과 같이 각각의 instruction의 opcode를 통해서 필요한 signal을 모두 posedge에 synchronous 하도록 되어 있다. 그리고 이때 생성된 signal은 ID stage를 위한 것이고, EX stage로 넘어갈 때 모든 signal을 register에 저장을 해둔다.

```
always @(posedge clk) begin
        if(is_stall) begin
                if_id_pc_reg <= if_id_pc_reg;
                if_id_instruction_reg <= if_id_instruction_reg;

        end
        else if(id_ex_jalr && !(jalr_check)) begin
                if_id_pc_reg <= if_id_pc_reg;
                if_id_instruction_reg <= 16'bz;

        end
        else if(is_BJ_type_update && !(b_j_check)) begin
                if_id_pc_reg <= if_id_pc_reg;
                if_id_instruction_reg <= 16'bz;

        end

        else begin
                if_id_pc_reg <= pc_value;       //reg pc? ?? ?? ?
                if_id_instruction_reg<= data1;      //instruction ????? ?

        end
end
```

일단 기본적으로 위에 있는 것은 instruction이 어떻게 유지가 되는지이다. Stall이 일어난 경우에는 ID stage의 pc값과 instruction을 update를 해줄 필요가 없으므로 현재 값과 동일한 값을 넣게 되고, flush가 일어난 경우 현재 fetch된 instruction이 필요 없어지는 것이므로 16'bz를 넣어준다. 그 외의 경우에 있어서는 data1과 pc_value 즉 IF stage의 pc 값과 instruction을 받아온다.

그 후 register file에서 값을 받아오고, branch가 맞았는지 틀렸는지를 판별해 줘야한다. 근데 여기서도 조심해야할 점이 branch는 register에 있는 값을 이용하기 때문에 branch에 이용되는 register 역시 forwarding이 필요로 하다는 점이다. 그래서 다음과 같이 forwarding unit을 사용하여 현재 instruction으로 이용되고 있는 register들의 실제 값들을 받아오도록 되어 있다.

```
register_file REGISTER (read_out1, read_out2, register_read_1_index, register_read_2_index, mem_wb_rd_index, write_back_data, pc_to_reg, mem_wb_reg_write, clk, reset_n);

forwarding_unit branch_forwarding(16'bz, register_read_1_index, if_id_instruction_reg, 16'bz, register_read_2_index,
id_ex_rd_index, id_ex_reg_write, ex_mem_rd_index, ex_mem_reg_write, branch_rs1_selection, branch_rs2_selection);

mux4_1 branch_rs1_mux (branch_rs1_selection, id_ex_alu_output, ex_mem_alu_value, read_out1, 16'bz, branch_rs1_out);

mux4_1 branch_rs2_mux (branch_rs2_selection, id_ex_alu_output, ex_mem_alu_value, read_out2, 16'bz, branch_rs2_out);
```

그리고 이제 branch가 맞았는지 틀렸는지를 결정하기 위해서

```
wire signed [15:0] signed_read_out1;
assign signed_read_out1 = branch_rs1_out;
always @(*) begin
        if(if_id_instruction_reg[15:12] == 0) begin
                if(branch_rs1_out != branch_rs2_out) begin
                        bcond = 1'b1;
                end
                else begin
                        bcond = 1'b0;
                end
        end
        else if(if_id_instruction_reg[15:12] == 1) begin
                if(branch_rs1_out == branch_rs2_out) begin
                        bcond = 1'b1;
                end
                else begin
                        bcond = 1'b0;
                end
        end
        else if(if_id_instruction_reg[15:12] == 2) begin
                if(signed_read_out1 > 0) begin
                        bcond = 1'b1;
                end
                else begin
                        bcond = 1'b0;
                end
        end
        else if(if_id_instruction_reg[15:12] == 3) begin
                if(signed_read_out1 < 0) begin
                        bcond = 1'b1;
                end
                else begin
                        bcond = 1'b0;
                end
        end
        else begin
                bcond = 1'b0;
        end
end
```

Bcond 값을 맞았는지 틀렸는지 판별해서 넣어주면 된다.

```
assign b_j_check = pc_write_cond ? (bcond ? if_id_pc_mux_out == pc_value - 1 : if_id_pc_reg === if_id_pc_mux_out):((pc_value -1) === if_id_pc_mux_out);

assign if_id_pc_next = (pc_write_cond && !(bcond) && (if_id_pc_reg != if_id_pc_mux_out)) ? if_id_pc_reg : if_id_pc_mux_out;
```

그리고 b_j_check라고 하는 현재 fetch가 되고 있는 값이 맞는지 틀렸는지를 결정해주는 bit를 내주면 되고, 그때 실제로 fetch되어 사용되야하는 pc 값을 결정해서 내보내준다.

3) EX stage

EX stage부터는 instruction을 결정하는 과정이 매우 복잡한데 stall이 되었는가, flush가 되었는가 등에 따라서

```
if(is_stall) begin
        id_ex_write_m2 <= 1'b0;
        id_ex_read_m2 <= 1'b0;
        id_ex_reg_write <= 1'b0;
        id_ex_halt <= 1'b0;
        id_ex_wwd <= 1'b0;
        id_ex_alu_src <= 1'b0;
        id_ex_instruction <= 16'bz;
end
else if(id_ex_jalr && !(jalr_check)) begin
        id_ex_write_m2 <= 1'b0;
        id_ex_read_m2 <= 1'b0;
        id_ex_reg_write <= 1'b0;
        id_ex_halt <= 1'b0;
        id_ex_wwd <= 1'b0;
        id_ex_alu_src <= 1'b0;
        id_ex_instruction <= 16'bz;
        id_ex_F <= 1'b1;
end
else if (if_id_instruction_reg === 16'bz) begin
        id_ex_write_m2 <= 1'b0;
        id_ex_read_m2 <= 1'b0;
        id_ex_reg_write <= 1'b0;
        id_ex_halt <= 1'b0;
        id_ex_wwd <= 1'b0;
        id_ex_alu_src <= 1'b0;
        id_ex_instruction <= 16'bz;
        id_ex_F <= 1'b0;
end
else begin
        id_ex_write_m2 <= con_write_m2;
        id_ex_read_m2 <= con_read_m2;
        id_ex_reg_write <= reg_write;
        id_ex_halt <= halt;
        id_ex_wwd <= wwd;
        id_ex_alu_src <= alu_src;
        id_ex_instruction <= if_id_instruction_reg;
        id_ex_F <= (is_BJ_type_update && !(b_j_check)) || (id_ex_jalr && !(jalr_check));
end
```

위와 같이 control signal과 instruction을 다르게 register에 저장하게 되고, 그 외에 필요한 것들에 대해서는

```
id_ex_is_BJ_type_update <= is_BJ_type_update;
id_ex_jalr <= jalr;
id_ex_pc_to_reg <= pc_to_reg;
id_ex_pc_reg <= if_id_pc_reg;
id_ex_S <= is_stall;
id_ex_reg1 <= read_out1;
id_ex_reg2 <= read_out2;
id_ex_sign_out <= sign_out;
id_ex_reg_index1 <= register_read_1_index;
id_ex_reg_index2 <= register_read_2_index;
id_ex_rd_index <= register_write_reg_index;
```

위와 같이 동작을 진행한다. 여기서 S, F로 되어있는 signal을 볼 수 있는데 S는 stall이 된 경우, F는 flush가 된 경우에 1로 설정되는 bit이다. 이는 나중에 WB stage가 끝난 후 num_instruction의 값을 조절해주기 위해서 사용되는 bit이다.

EX stage에서는 alu 동작이 수행이 되는데, 여기서도 역시 forwarding이 필요하기 때문에 forwarding을 통해서 어떤 값이 들어갈지 결정을 해주고, 그리고 sign_extend된 값이 필요한 경우를 고려해서 mux를 총 3개를 만들어주고, alu에 필요한 값을 넣어주면 된다.

```
hazard_detect HAZARD (if_id_instruction_reg, id_ex_rd_index, id_ex_read_m2, is_stall);

wire [15:0]id_ex_alu_mux1_out, id_ex_alu_mux2_out;
wire [1:0] rs1_selection, rs2_selection;
forwarding_unit FORWARDING(id_ex_reg1, id_ex_reg_index1, id_ex_instruction , id_ex_reg2, id_ex_reg_index2, ex_mem_rd_index, ex_mem_reg_write,
mem_wb_rd_index, mem_wb_reg_write, rs1_selection, rs2_selection);

wire[15:0] id_ex_mux_out;

mux4_1 id_ex_alu_input1_mux (rs1_selection, ex_mem_alu_value, mem_wb_mux_out, id_ex_reg1, 16'h0000, id_ex_alu_mux1_out);
mux4_1 id_ex_alu_input2_mux (rs2_selection, ex_mem_alu_value, mem_wb_mux_out, id_ex_reg2, id_ex_pc_reg, id_ex_alu_mux2_out);

mux2_1 if_id_mux (id_ex_alu_src, id_ex_alu_mux2_out, id_ex_sign_out, id_ex_mux_out);

wire [3:0] id_ex_funcCode;
wire [1:0] id_ex_branchType;

alu_control_unit ALU_CONTROL(id_ex_instruction[5:0], id_ex_instruction[15:12], clk, id_ex_funcCode, id_ex_branchType);

wire id_ex_overflow, id_ex_bcond;
alu id_ex_main_alu (id_ex_alu_mux1_out, id_ex_mux_out, id_ex_funcCode, id_ex_branchType, id_ex_alu_output, id_ex_overflow, id_ex_bcond);
```

위와 같이 연결이 되는 것을 확인할 수 있다. 그리고 jalr의 경우 EX stage에서 IF, ID 값들이 제대로 된 pc값인지 아닌지를 판별이 가능하기 때문에

```
assign jalr_check = ((if_id_pc_reg - 1) === id_ex_alu_output);
```

위와 같은 signal을 추가해서 제대로 된 값이 아닌 경우에는 ID, IF에 있는 instruction을 모두 flush를 해주면 된다.

4) MEM stage

```
always @(posedge clk) begin
        if(id_ex_jalr && !(jalr_check)) begin
                ex_mem_F <= 1'b1;
        end
        else begin
                ex_mem_F <= id_ex_F;
        end
        ex_mem_alu_value <= id_ex_alu_output;
        ex_mem_reg2 <= id_ex_alu_mux2_out;
        ex_mem_instruction <= id_ex_instruction;
        ex_mem_rd_index <= id_ex_rd_index;
        ex_mem_write_m2 <= id_ex_write_m2;
        ex_mem_read_m2 <= id_ex_read_m2;
        ex_mem_reg_write <= id_ex_reg_write;
        ex_mem_halt <= id_ex_halt;
        ex_mem_wwd <= id_ex_wwd;
        ex_mem_S <= id_ex_S;
        ex_mem_pc_reg <= id_ex_pc_reg;
        ex_mem_pc_to_reg <= id_ex_pc_to_reg;
        ex_mem_jalr <= id_ex_jalr;
end
```

Mem stage에서는 instruction을 EX stage에 있던 것들과 signal 역시 모두 다 받아오면 되고, 하나 더 고려해줘야 할 점이 있다면 jalr인 경우 flush가 두 번 된다는 점을 고려해서 flush bit을 한 번 더 설정을 해줘야 한다는 점이다.

```
assign address2 = ex_mem_alu_value;
assign data2 = ex_mem_write_m2 ? ex_mem_reg2 : 16'bz;

reg [15:0] mem_wb_memory_value, mem_wb_alu_value;
```

그리고 위와 같이 data를 Memory에서부터 load를 수행하거나 store를 진행해주면 된다.

5) WB stage

여기서는 load가 되는 경우에 필요하기 때문에 data2를 mem_wb_memory_value에 저장을 시켜주고 연산을 진행하면 된다. 여기서는 register file에 값을 적어주는 과정과 num instruction 값만 변경을 해주면 된다. 하지만 여기서 alu 값이 저장되는 경우와 data2 즉 memory에서 값을 읽어오는 두 경우를 고려를 해서 WB stage를 진행해주면 된다.

```
always @(posedge clk) begin
        mem_wb_memory_value <= data2;
        mem_wb_alu_value <= ex_mem_alu_value;
        mem_wb_instruction <= ex_mem_instruction;
        mem_wb_rd_index <= ex_mem_rd_index;
        mem_wb_write_m2 <= ex_mem_write_m2;
        mem_wb_read_m2 <= ex_mem_read_m2;
        mem_wb_reg_write <= ex_mem_reg_write;
        mem_wb_halt <= ex_mem_halt;
        mem_wb_wwd <= ex_mem_wwd;
        mem_wb_S <= ex_mem_S;
        mem_wb_F <= ex_mem_F;
        mem_wb_pc_reg <= ex_mem_pc_reg;
        mem_wb_pc_to_reg <= ex_mem_pc_to_reg;
end
```

위는 WB stage에 필요한 instruction과 signal을 조절하기 위한 register 값 변경에 관한 부분이고,

```
always @(posedge clk) begin
        if(ex_mem_S) begin
                num_inst <= num_inst;
        end
        else if(mem_wb_F) begin
                num_inst <= num_inst;
        end
        else begin
                num_inst <= num_inst + 1;
        end
end
```

위의 부분은 num_instruction의 값을 조절하기 위한 부분이다.

```
mux2_1 REGISTER_IN (mem_wb_pc_to_reg, mem_wb_mux_out, mem_wb_pc_reg, write_back_data);
```

그리고 이것은 WB에 필요한 data를 결정하기 위한 mux부분이다.

b. Control unit

Control unit의 경우 하나 고려해야할 점이 있다면 instruction을 어떤 것을 사용하냐에 대한 문제이다. Data1을 이용하여 진행을 하면 flush나 stall이 되는 경우에 대해서 고려를 못 해주기 때문에 flush나 stall이 되는 경우도 고려를 해서 control unit에 들어가는 instruction을 조절해주면 된다.

```
assign control_unit_data =  (is_stall) ? control_unit_instruction : (((id_ex_jalr && !(jalr_check)) || (!(b_j_check) && is_BJ_type_update)) ? 16'bz : data1);
```

그 부분을 조절해주는 코드이며, stall이 된 경우 ID stage에 있는 instruction을 읽어서 사용하면 되고, flush가 되는 경우 control signal이 필요로 하지 않으므로 16'bz를 넣어 주고, 그 외의 경우에 한해서는 data1을 이용하여 signal을 posedge 마다 생성해주면 된다.

```
always @(posedge clk) begin
        if (opcode == 4'd15) begin
                pc_write_cond <= 1'b0; //branch then 1
                pc_write<= 1'b1; //if pc_wirte == 1 then pc update, fetch and jump then 1

                read_m1 <= 1'b1;
                read_m2 <= 1'b0;
                write_m2 <= 1'b0;
                alu_src <= 1'b0;

                reg_write <= 1'b0;

                if(func_code == 6'd25) begin //JPR
                        halt <= 1'b0;
                        wwd <= 1'b0;
                        mem_to_reg <= 2'b00;
                        reg_write <= 1'b0;
                        if_id_mux_con <= 2'b10;

                        pc_src <= 1'b1;
                        pc_to_reg <= 1'b0;
                        jalr <= 1'b1;
                        is_BJ_type_update <= 1'b0;
                end
```

그리고 instruction을 받은 후 opcode에 맞게 각 signal에 맞는 bit를 설정해주면 된다. 그리고 또 다른 특징이 하나 있다면 현재 생성된 signal이 ID stage 이후 EX, MEM, WB stage에서도 필요로 하기 때문에 이 값들을 모두 register로 저장을 해두고, 사용하게 된다. 이 저장은 Datapath 에서 진행이 된다.

c. Forwarding unit

Forwarding unit에서는 id_ex 에서 사용하려고 하는 register 값이 ex_mem와 index가 같은지, mem_wb과 index가 같은지를 판별해서 다음과 같이 if, else if, else로 나타내는 것이 중요하다는 점이다. 그리고 또한 실제로 WB이 되는지 안되는지도 중요하므로 reg_write이 1인 경우에 한해서만 forwarding을 진행해주면 된다.

```
always @(*) begin
        if((id_ex_rs1_index == ex_mem_rd_index) && ex_mem_reg_write) begin
                rs1_selection = 2'b00;
        end
        else if((id_ex_rs1_index == mem_wb_rd_index) && mem_wb_reg_write) begin
                rs1_selection = 2'b01;
        end
        else begin
                rs1_selection = 2'b10;
        end
```

위의 내용은 설명에 해당하는 부분이고, rs2 역시도 동일하게 진행해주면 된다.

d. Hazard unit

Hazard가 발생하는 경우는 ID EX에 있는 값이 memory를 read 즉 load를 해오는 경우이고, 실질적으로 사용하는 register 값이 한 개라도 겹치는 경우이다. 그래서 실질적으로 어떤 register를 사용하는지를 결정하는 use_rs1, rs2 값을 결정을 해주고, 현재 ID stage에 있는 register 값이 무엇이냐를 판별하여, EX stage에 memory read가 있는지를 봐주고, 있는 경우 stall을 해주면 된다.

```
always @(*) begin
        if(IDEX_rd === 2'bz) begin
        check_bit = 1;
        end
        else begin
                if(IFID_IR[15:12] == `JMP_OP || IFID_IR[15:12] == `JAL_OP ||(IFID_IR[15:12] == 15 && (IFID_IR[5:0] == 27 || IFID_IR[5:0] == 29))) begin
                        use_rs1 = 0;
                        use_rs2 = 0;
                end
                else if (IFID_IR[15:12] == `LHI_OP) begin
                        use_rs1 = 0;
                        use_rs2 = 1;
                end
                else if (
                (IFID_IR[15:12] == 15 && (IFID_IR[5:0] < 8 && 3 < IFID_IR[5:0] || IFID_IR[5:0] == 28 || IFID_IR[5:0] == 25|| IFID_IR[5:0] == 26))
                                || IFID_IR[15:12] == 2 || IFID_IR[15:12] == 3) begin
                        use_rs1 = 1;
                        use_rs2 = 0;
                end
                else begin
                        use_rs1 = 1;
                        use_rs2 = 1;
                end
        check_bit = 0;
        end
end

assign is_stall = check_bit ? 0 : (((IFID_IR[11:10] == IDEX_rd) && use_rs1) || ((IFID_IR[9:8] == IDEX_rd) && use_rs2)) && IDEX_M_mem_read;
```

여기서 check_bit가 있는 이유는 16'bz로 flush를 진행하였는데 이 경우 is_stall 값이 이상하게 판별하는 경우가 있었기 때문에 16'bz인 경우에 is_stall의 값을 제대로 결정해주기 위해서 사용이 되었다.

e. Alu control_unit

Alu control unit의 경우 multicycle과 동일하지만 다른 점이 있다면 EX stage에 필요한 연산만을 진행해주면 된다는 점이고, 그리고 또한 branch를 모두 ID stage에서 소진을 시켜줬기 때문에 밑과 같이 funcCode가 무엇인지만 결정을 해주면 된다는 점이다. 하지만 branch 부분을 코드에서 빼지는 않았다.

```
always@(*) begin
        if(opcode == 4'b0000 || opcode == 4'b0001 || opcode == 4'b0010 || opcode == 4'b0011) begin
                branchType = opcode[1:0];
                funcCode = 4'd9; //branch funcCode == 9
        end
        else if(opcode == 4'd15) begin
                if(funct == 6'd0) begin
                        funcCode = 4'd0; // ADD
                end
                else if(funct == 6'd1) begin
                        funcCode = 4'd1; // SUB
                end
```

위의 설명에 해당하는 부분은 다음과 같다.

f. Register file

저번 multicycle과 동일하지만 register WB을 negedge clock에 진행한다는 점이 다른 점이다.

```
initial begin
        register[0] = 0;
        register[1] = 0;
        register[2] = 0;
        register[3] = 0;
end

assign read_out1 = register[read1];
assign read_out2 = register[read2];

always@(negedge clk) begin
        if(reg_write == 1) begin
        register[dest] <= write_data;
        end

        else if(pc_to_reg) begin
        register[dest] <= write_data;
        end

        else begin
        register[dest] <= register[dest];
        end
end
//TODO: implement register file
```

그리고 이번에는 pc_to_reg 즉 pc 값을 register에 넣는 경우에 한한 instruction이 없는데 그 이유는 pc 값을 넣는 경우여도, alu를 거쳐서 계산되어 나오기 때문에 결국에는 reg_write 값이 1이고, alu에서 나온 wire를 이용하면 되기 때문이다.

g. Sign extend

Sign extend는 바뀐 부분이 없다.

```
always@(*) begin
        if(instruction[15:12] == 4'b1001 || instruction[15:12] == 4'b1010) begin
                out_sign = instruction[11] ? {8'b1111, instruction[11:0]}:{8'b0000, instruction[11:0]};
        end
        else begin
                out_sign = instruction[7] ? {8'b11111111, instruction[7:0]}:{8'b00000000, instruction[7:0]};
        end
end
```

h. ALU

Alu는 전혀 바뀐 부분이 없기 때문에 다시 사용했다.

```
always@(*) begin
        if(func_code == 4'd0) begin
                C = A + B;
                overflow_flag = ((A[15] & B[15] & (~C[15])) | ((~A[15]) & (~B[15]) & C[15]));
        end
        else if(func_code == 4'd1) begin
                C = A - B;
                overflow_flag = ((A[15] & (~B[15]) & (~C[15])) | ((~A[15]) & B[15] & C[15]));
        end
        else if(func_code == 4'd2) begin
                C = A & B;
        end
```

i. Util

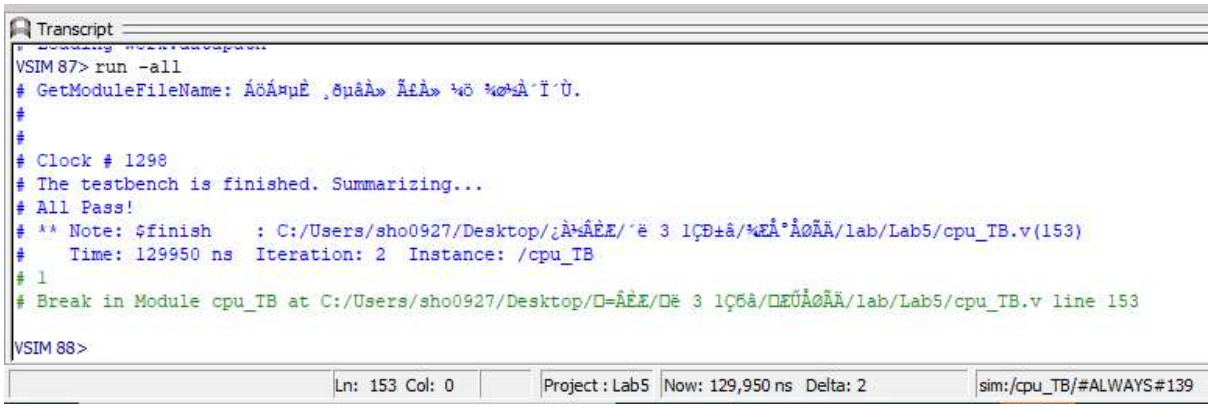
Util 역시 재사용하였다.

```
module mux2_1 (sel, i1, i2, o);
   input sel;
   input [15:0] i1, i2;
   output reg [15:0] o;

   always @ (*) begin
      case (sel)
         0: o = i1;
         1: o = i2;
      endcase
   end

endmodule
```

Discussion

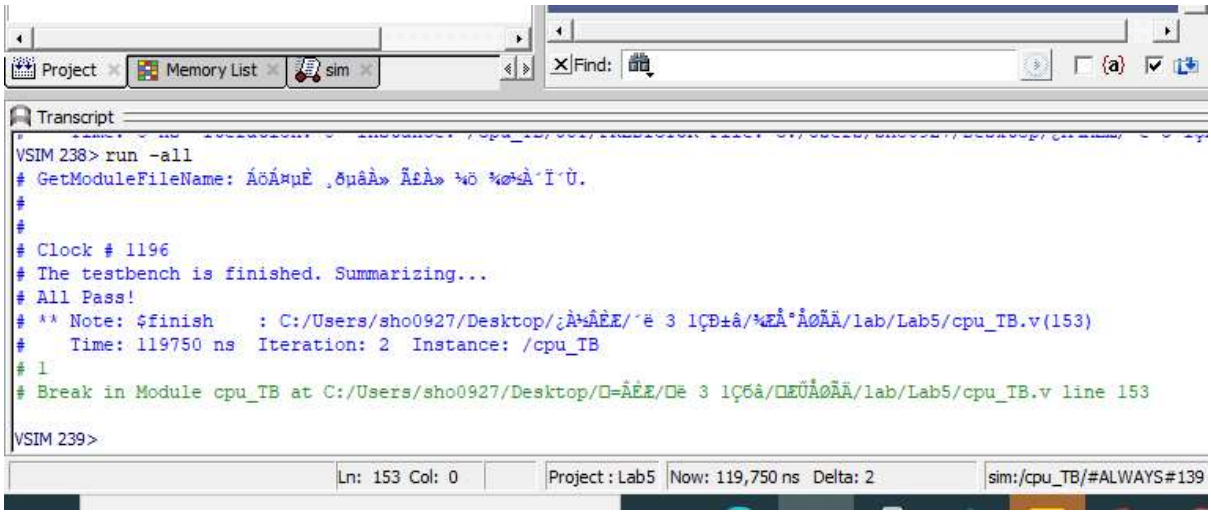
이번 lab에서는 다른 것보다 always not taken, always taken, branch prediction 이렇게 총 3가지를 만들어야했기 때문에 더더욱 힘든 과제가 되었다고 생각된다. Always not taken을 만들고, 그 후 always taken, branch prediction은 쉬울 것 같다고 생각했지만 CPU top module과 branch predictor 모듈이 조금의 변경이 필요로 했고, datapath 역시 조금의 추가가 필요로 했다. 결론적으로 비교를 해보면

Always not taken의 경우



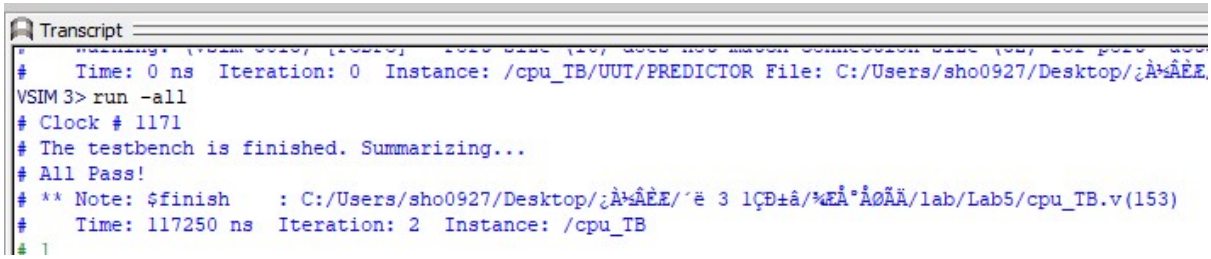
다음과 같이 1298의 clock을 가지고, all pass를 하는 것을 확인할 수 있고

Always taken의 경우



다음과 같이 1196의 clock을 가지고, all pass를 하는 것을 확인할 수 있으며,

2-bit prediction의 경우



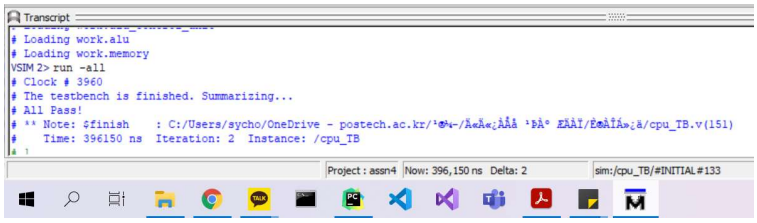
위의 경우처럼 1171의 clock을 가지고, all pass를 하는 것을 확인이 가능하다.

이 경우 always not taken과의 성능을 비교를 해보면, always taken을 사용하는 경우 8.5%의 성능 향상이 있었으며, 2-bit prediction을 사용하면 10.8%의 성능 향상이 있었음을 확인이 가능하다.

그리고 2-bit saturation counter 역시 구현을 해봤는데 이 경우에는 1186이 나왔고, all pass가 나오는 것을 확인 하였다. 이 경우에는 9.4%의 성능 향상이 있는 것을 확인하였다.

하지만 2-bit prediction model에서 하나 의문이 들었는데 이 경우에 처음 counter의 initial 값에 따라서 clock의 차이가 있을 것이라고 생각하였지만, clock의 차이가 존재하지 않았다. 이 이유는 확실하게 분석은 하지 못하겠지만 현재 Test bench에서 Branch가 taken이 되고, not taken이 되고 하는 부분이 반복되는 것이 있어서 이러한 결과가 나오지 않았나라는 생각이 들었다.

그리고 multicycle과도 성능을 비교를 해보자면



위의 cycle 수 3960이 우리의 multicycle의 clock 수였기 때문에

Always not taken의 경우 : 3.05배의 성능 향상이 이뤄졌고,

Always taken의 경우 : 3.30배의 성능 향상이 이뤄졌고,

2-bit saturation model의 경우 : 3.338배의 성능 향상이 이뤄졌으며,

2-bit hypothesis model의 경우 : 3.38배의 성능 향상이 이뤄졌음을 확인할 수 있다.

Conclusion

결과적으로 이번 pipeline의 모든 구현을 완료하였으며, forwarding을 한 경우에 한해서 always not taken, always taken, 2-bit saturation model, 2-bit hypothesis model 모두의 경우에 한해서 성

능을 비교를 해볼 수 있었다. 또한 이번 lab을 하면서 어려워했던 timing을 정확히 이제 이해를 하고 있다는 것을 알게 되었다. 그리고 branch를 ID로 당겨서 사용을 하기 위해서는 그 부분에도 forwarding이 필요를 한다는 것을 깨닫게 된 lab이라고 생각한다. 마지막으로 === 이라는 것을 사용하여 16'bz인 경우에 값이 같은지를 판별할 수 있는 operator도 있다는 것을 깨닫게 되었다.