Lab 5 - Pipelined CPU

20160595 수학과 최규민
20160169 물리학과 최수민

1. Introduction

이번 과제에서는 Pipelined CPU를 구현한다. Pipelined CPU는 한 instruction을 여러개의 stage로 나누어, 모든 stage가 동시에 병렬로 동작하여 같은 시간에 더 많은 instruction을 수행할 수 있게 한다. 이는 개별 instruction의 실행시간을 감소하지는 못해도, 전체적인 throughput의 향상을 가져온다. 다음 instruction를 이전 instruction이 끝나기 전에 함께 실행하는 파이프라인의 특성상 structural, data, control hazard가 발생할 수 있는데, 이를 파이프라인 설계 및 stall, forwarding을 통해 해결하고, branch prediction unit을 구현하여 더 좋은 성능을 낼 수 있도록 한다.

이번 과제를 통해 배워야 하는 것은 다음의 두 가지로 요약할 수 있다.

1. Pipelined CPU구현을 통해 어떻게 파이프라이닝 기법이 throughput의 향상을 가져올 수 있는지 이해한다.
2. 각종 data, control hazard가 일어나는 원인을 이해하고, 이를 해결하는 방법을 이해한다.

2. Design

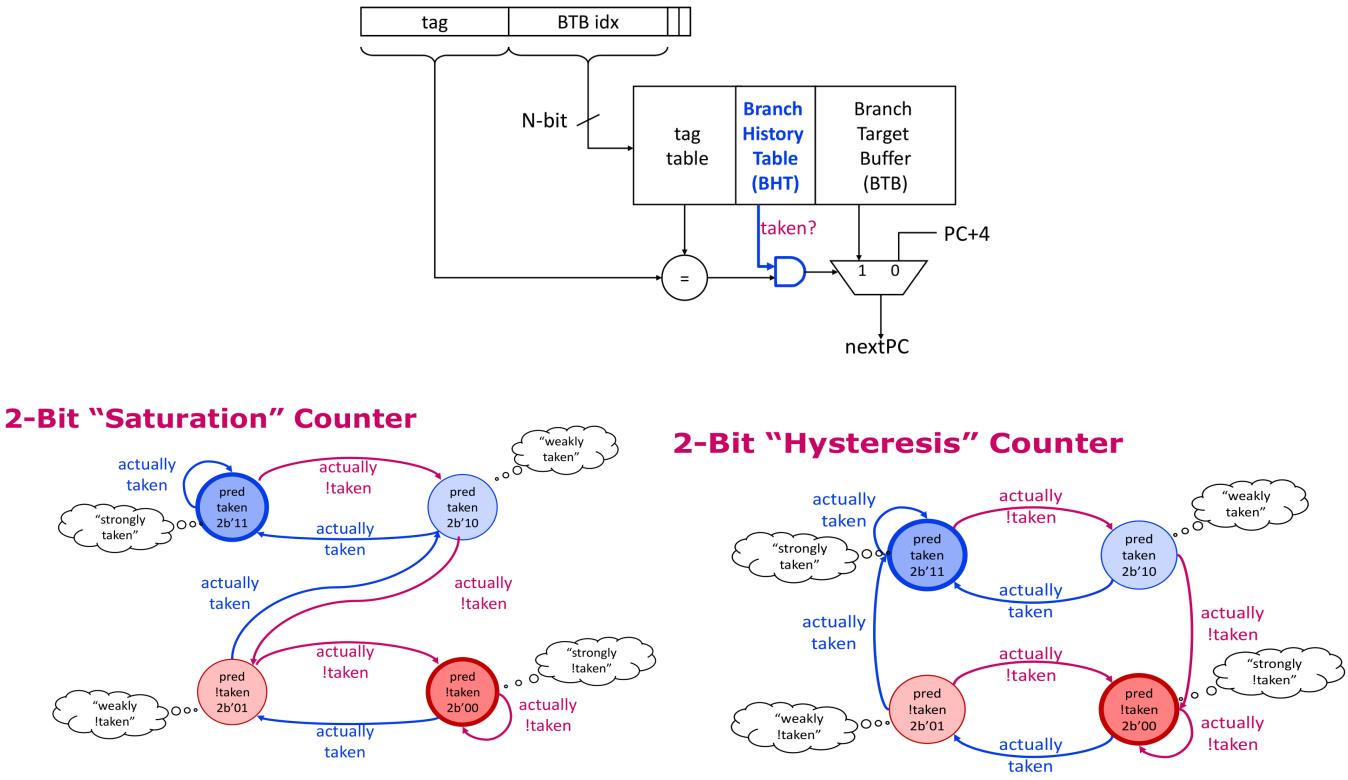
Multi Cycle CPU에서는 instruction의 실행을 여러 단계로 나눴을 때 각 단계별로 사용하는 리소스가 다르다면 특정 단계를 실행하는 동안 사용되지 않는 리소스가 있게 된다. Pipelined CPU에서는 이렇게 사용되지 않는 리소스를 다른 instruction이 쓸 수 있도록 해서 여러 instruction을 중첩해 실행하여 전체 throughput을 높일 수 있다. 이 때 여러 instruction을 중첩해 실행하기 위해선 나중 단계에서 필요한 데이터를 보관하기 위한 pipeline register를 필요로 한다. 따라서 명령어들을 IF, ID, EX, MEM, WB의 5단계로 나누고 각 단계 사이에 pipeline register를 추가했으며, instruction의 진행에 따라 필요한 데이터와 control signal들을 다음 단계의 pipeline register으로 전달하도록 설계했다.

여러 instruction을 중첩해 실행하다보니 발생하는 hazard들이 있는데, 이 hazard들을 해결하기 위해 추가 모듈들을 필요로 한다. 먼저 data hazard는 레지스터에 값을 쓰는 단계가 레지스터의 값을 읽는 단계보다 뒤쪽에 있기 때문에 발생하는 것으로, 이를 해결하기 위해서 forwarding_unit을 사용하여 데이터가 ALU에서 계산되어 나오자마자 이를 필요로 하는 유닛으로 전달해 지연없이 코드를 실행할 수 있게 하였다. 단, LWD 명령어 뒤에 이 결과값을 읽는 명령어가 뒤따라 나오면 forwarding만으로 해결할 수 없기 때문에 hazard_detect 유닛에서 stall 조건을 확인하여 파이프라인을 지연시킨다.

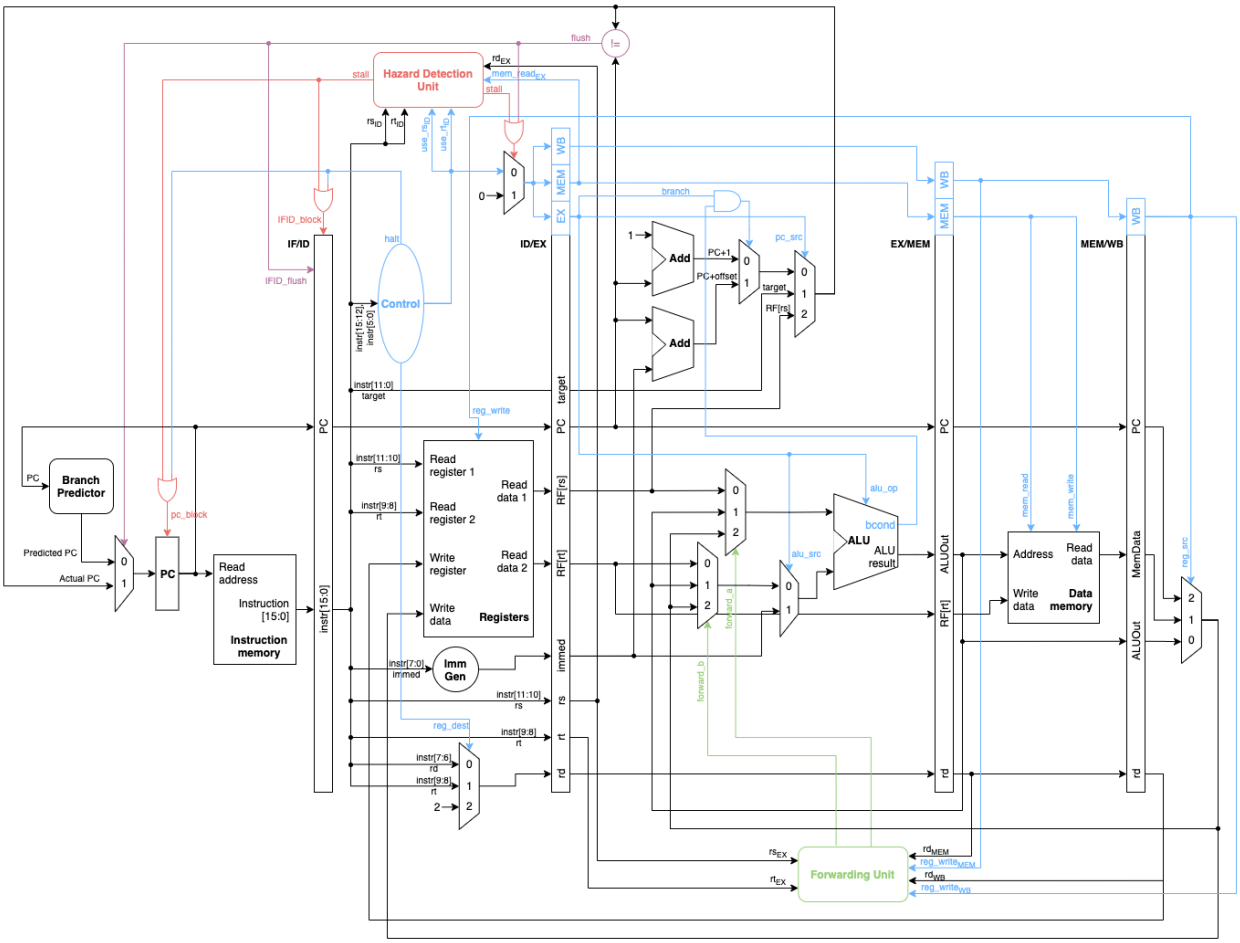
Control hazard는 모든 instruction이 PC를 읽고 쓰기 때문에 발생한다. 다음 instruction을 결정하기 위해선 이전 instruction을 ID 단계에서 다음 PC를 결정해야 하는데, 이 경우 매번 ID 단계까지 기다려야

하기 때문에 성능이 절반으로 감소한다. 따라서 instruction이 ID 단계까지 완료되도록 기다리지 않고 다음 PC를 먼저 예측하여 fetch해온 뒤, 실제 PC값과 다르게 예측했을 경우 가져온 instruction을 flush하도록 설계했다. 이에 따라 flush하는 로직과 현재 PC를 바탕으로 다음 PC를 예측하는 branch predictor 모듈을 추가했다. 이 때 예측이 틀릴 경우의 패널티를 줄이기 위해 branch 및 jump의 실제 PC값이 MEM 단계가 아닌 EX 단계에서 결정되도록 설계하였다.

Branch predictor의 경우 여러 방식이 있지만, TSC CPU 기준으로 다음 PC를 항상 PC+1로 예측하는 always not taken 방식과, branch와 jump에 해당하는 instruction의 경우 이전 target 값으로 예측하고 그 외의 instruction은 모두 PC+1로 예측하는 always taken 방식, 2-bit global predictor를 바탕으로 다음 PC를 예측하는 방식, 그리고 2-bit per-address predictor를 바탕으로 다음 PC를 예측하는 방식의 총 4가지 구현 방식으로 설계하였다. 자세한 설계는 모듈 별 설명에 적었다.



[그림 1] Branch Target Buffer(BTB)를 사용한 Branch Predictor의 구조.
2-bit global predictor의 경우 saturation 방식과 hysteresis 방식을 모두 나타내었다.



[그림 2] Pipelined CPU의 datapath. 각 pipeline register별로 저장하는 값과 전달되는 값이 표시되어 있다.

Stage/Unit	Signal	값	효과
ID	use_rs	0	아무 효과 없음
		1	instruction에서 RF[rs]를 사용함을 알림(hazard detection)
	use_rt	0	아무 효과 없음
		1	instruction에서 RF[rt]를 사용함을 알림(hazard detection)
EX	alu_op		instruction별 연산 실행
	alu_src	0	RF[rt]가 ALU의 두번째 operand
		1	Immediate값이 ALU의 두번째 operand
	branch	0	PC+1
		1	PC+immediate

	pc_src [1:0]	00	PC +1 또는 PC+immediate
		01	target(instr[11:0])
		10	RF[rs]
MEM	**mem_read**	0	아무 효과 없음
		1	메모리에서 값 읽기 가능
	mem_write	0	아무 효과 없음
		1	메모리 값 변경 가능
WB	**reg_write**	0	RF값 변경 불가능
		1	RF값 변경 가능
	reg_dest [1:0]	00	RF[rd] 값 변경
		01	RF[rt] 값 변경
		10	RF[2] 값 변경
	reg_src [1:0]	00	ALUOut값을 레지스터에 입력
		01	MDR 값을 레지스터에 입력
		10	PC값을 레지스터에 입력
	wwd	0	아무 효과 없음
		1	RF[rs] 값을 outputport로 내보냄
	halt	0	아무 효과 없음
		1	is_halted를 알리고 이후 추가적인 instruction fetch가 없음
	new_inst	0	아무 효과 없음
		1	num_inst + 1
Forwarding Unit	**Forward_a [1:0]**	00	기존 값 사용
		01	Forwarding from EX/MEM
		10	Forwarding from WB/MEM
	Forward_b [1:0]	00	기존 값 사용
		01	Forwarding from EX/MEM
		10	Forwarding from WB/MEM

[표 1] Control unit에서 사용되는 각 control signal의 기능. Stage/Unit 컬럼에서는 해당 시그널이 사용되는 유닛이나 단계를 나타냈으며, 2비트 control 중 사용되지 않는 시그널은 표시하지 않았다.

Instruction	halt	reg_ dest	use_ rs	use_ rt	alu_ src	branch	pc_ src	mem_ read	mem _write	reg_ write	reg_ src
ADD, SUB, AND, ORR	0	00	1	1	0	0	00	0	0	1	00
NOT, TCP, SHL, SHR	0	00	1	0	0	0	00	0	0	1	00
ADI, ORI	0	01	1	0	1	0	00	0	0	1	00
LHI	0	01	0	0	1	0	00	0	0	1	00
LWD	0	01	1	0	1	0	00	1	0	1	01
SWD	0	00	1	0	1	0	00	0	1	0	00
BNE, BEQ	0	00	1	1	0	1	00	0	0	0	00
BGZ, BLZ	0	00	1	0	0	1	00	0	0	0	00
JMP	0	00	0	0	0	0	01	0	0	0	00
JAL	0	10	0	0	0	0	01	0	0	1	10
JPR	0	00	1	0	0	0	10	0	0	0	00
JRL	0	10	1	0	0	0	10	0	0	1	10
WWD	0	00	1	0	0	0	00	0	0	0	00
HLT	1	00	0	0	0	0	00	0	0	0	00

[표 2] 각 instruction별 control signal의 값.

메인 모듈은 CPU 모듈이며, 서브 모듈에는 전반적인 모듈의 연결과 pipeline register가 선언되어있는 datapath 모듈과 control을 구성하는 control_unit 모듈, 그리고 그 외의 hazard, forwarding_unit, branch_predictor, register_file, alu, memory, immediate_generator, mux4_1, mux2_1이 있다.

2.1. CPU 모듈

CPU 모듈은 input으로 reset signal(reset_n), clock signal(clk), data1을 받고, inout port로 사용하는 data2가 있다. output으로는 read_m1과 read_m2, write_m2, address1, address2, 그리고 num_inst, is_halted, output_port가 있다. read_m1, address1, data1은 메모리에서 instruction을 읽어오는데 사용하고, read_m2, write_m2, address2, data2는 메모리에서 데이터를 읽어오거나 쓸 때 사용한다. 또한 하나의 instruction이 끝날 때마다 num_inst를 증가시켜 출력시켜주고, WWD instruction의 경우 레지스터값을 output_port를 통해 출력해준다. HLT instruction의 경우 is_halted를 출력하여 프로그램이 halt되었음을 알린다. 이 모든 input, output, inout들은 datapath 모듈과 그대로 연결된다.

2.2. datapath 모듈

CPU 모듈과 동일한 input, output, inout을 가진다. 실질적으로 다른 서브 모듈들에 적절한 input과 output을 연결하여 데이터를 전달하는 것을 담당한다. 메모리에 control signal와 데이터를 전달하고, PC값과 메모리에서 읽어온 값인 instruction과 memory data를 register 변수에 저장하며, register pipeline에 해당하는 reg 변수들의 업데이트를 담당한다. 또한 wb 단계의 halt, wwd, new_inst 값에 따라 cpu에 적절한 output을 전달한다.

2.3. control_unit 모듈

control_unit 모듈은 읽어온 instruction을 바탕으로 control signal들을 생성한다. 각 control signal의 의미와 instruction별 control signal의 값은 표 1, 2에 있다. 모듈의 input으로는 instruction의 opcode와 function을 받아오며 output으로 표 1, 2에 정의된 control signal들과 ALU에 가하는 시그널인 funcCode와 branchType을 가진다.

2.4. alu 모듈

alu 모듈은 input으로 연산을 할 두 수(A, B)와 alu_control_unit 모듈의 output인 func_code, branch_type을 받아 output으로 적절한 연산의 결과(C)와 오버플로 여부를 판단하는 overflow_flag, 그리고 branch 조건이 성립함을 나타내는 bcond를 내보내는 combinational logic으로 구성된다.

2.5. hazard 모듈

Forwarding unit으로 해결 불가능한 LWD 명령어에 따른 data hazard의 경우에는 파이프라인 지연이 필요한데, hazard 모듈은 ID 단계에서 지연(stall)을 추가할 수 있게 한다. ID 단계의 source register rs, rt와 EX 단계의 destination register가 같고, LWD 명령어이면 1 사이클을 지연시킨다. 체크하는 조건은 다음과 같다.

$$stall = [\ (rs_{ID}==rd_{EX}\ \&\&\ use\_rs_{ID})\ ||\ (rt_{ID}\ ==\ rd_{EX}\ \&\&\ use\_rt_{ID})\ ]\ \&\&\ mem\_read_{EX}$$

2.6. forwarding_unit 모듈

어떤 레지스터에 값을 쓰는 instruction이 그 레지스터의 값을 읽는 instruction보다 먼저 실행될 때, RAW(Read After Write) dependency로 인해 Data hazard가 발생한다. 이 때 Stall 없이 레지스터에 값을 쓰기 전 ALU에서 값이 계산되면, 이후의 instruction에서 레지스터에서 읽어온 값이 아니라 앞의 instruction에서 계산된 값을 사용하도록 forwarding한다. Forwarding을 하는 조건은 다음과 같다.

if (rs_{EX} == rd_{MEM}) && reg_write_{MEM} then
    forward_a = 01
else if (rs_{EX} == rd_{WB}) && reg_write_{WB} then
    forward_a = 10

else
	forward_a = 00

if (rt_{EX} == rd_{MEM}) && reg_write_{MEM} then
	forward_b = 01
else if (rt_{EX} == rd_{WB}) && reg_write_{WB} then
	forward_b = 10
else
	forward_b = 00

2.7. branch_predictor 모듈

총 3가지 모듈로 나누어 위에서 설명한 4가지 방식을 구현한다.

1) branch_predictor_always_not_taken 모듈
항상 다음 PC를 PC+1로 예측하고, flush인 경우에는 input의 실제 PC로 다음 PC를 설정한다.

2) branch_predictor_always_taken 모듈
EX 단계에서 branch가 taken 됐을 때의 PC를 계산하면, 이를 branch predictor 모듈로 전달하여 Branch Target Buffer(BTB)에 저장한다. 입력으로 들어온 PC에 대해 BTB를 조회하여 있으면 BTB의 PC값을, 없으면 현재 PC+1을 예측값으로 사용한다. flush인 경우에는 input의 실제 PC로 다음 PC를 설정한다.

3) branch_predictor_global_predictor 모듈
always_taken 모듈과 마찬가지로 BTB에 taken PC값을 저장한다. 2-bit global predictor의 경우, branch, jump instruction이 taken되었는지에 대한 값을 바탕으로 2비트 global state를 업데이트하고 이로부터 예측값을 결정한다. 2비트 state machine은 saturation counter와 hysteresis counter 방식을 사용한다. 2-bit per-address predictor의 경우 branch, jump instruction이 taken되었는지에 대한 정보를 바탕으로 BTB의 각 entry별 2비트 state를 업데이트하고, 이 entry별 state로부터 예측값을 결정한다.
flush인 경우에는 input의 실제 PC로 다음 PC를 설정한다.

2.8. 기타 모듈

Register_file, immediate_generator, mux2_1, mux4_1, memory 모듈은 single cycle CPU, multi cycle CPU에서의 Design과 동일하다. Register_file은 clock synchronous하게 업데이트시킨다. 다만 pipeline register가 업데이트된 후 레지스터를 업데이트할 수 있도록 clock의 negative edge를 기준으로 write를 한다. Memory 모듈은 CPU에서 주는 read_m과 write_m에 따라 clock synchronous하게 memory를 읽고 쓸수 있게 한다.

2.8. 그 외 로직

*** Flush Logic**

flush = (predictedPC(pc_id) != actualPC(actual_pc)) && pc_id != 0 (파이프라인 시작시) && pc_ex != ~0 (flush 상태일때)

*** PC와 IF/ID pipeline 업데이트**

PC: stall || halt 이면 PC에 쓸 수 없도록 하여 PC값을 유지한다.

IF/ID: stall || halt 이면 IF/ID에 쓸 수 없도록 하여 IF/ID의 pipeline register에 저장된 값을 유지한다.
flush이면 instruction을 nop으로 바꾼다.

ID/EX: stall || flush이면 control을 0으로 바꿔 architectural state가 변경되지 않도록 한다.

*** new_inst 증가와 WWD, HLT 명령어**

모든 valid한 명령어는 new_inst = 1로 설정되고, WB 단계가 끝날 때 new_inst == 1이면 num_inst를 1 증가한다.

WWD의 경우 WB 단계에서 output_port로 지정된 레지스터의 값을 내보낸다.

HLT의 경우 WB 단계가 완료되면 is_halted를 assert한다.

3. Implementation

3.1. CPU 모듈 (Datapath 모듈)

```
else begin
        // update pc
        if(stall) begin
                pc <= pc;
        end else begin
                pc <= pc_nxt;
        end

        // update IF/ID pipeline register (instr from data)
        if(!flush & !stall) begin
                pc_id <= pc;
                new_inst_id <= new_inst_if;
        end else if(stall) begin // stall
                pc_id <= pc_id;
                new_inst_id <= new_inst_id;
        end     else begin // flush
                pc_id <= ~0;
                new_inst_id <= 0;
        end
```

[그림 3] CPU(Datapath) 모듈의 코드 중 일부. Posedge마다 시그널에 맞게 pc, data를 업데이트 해주고 stall, flush 조건을 확인하면서 파이프라인 레지스터값을 posedge마다 흘려준다.

```
// get memory data
assign read_m1 = 1;
assign address1 = pc;
assign write_m2 = mem_write_mem;
assign read_m2 = mem_read_mem;
assign address2 = alu_out_mem;
assign data2 = read_m2? `WORD_SIZE'bz: rf_rt_mem;

// instruction memory
always @(posedge clk) begin
        if (!reset_n) begin
                instr <= 0;
        end else if(stall) begin
                instr <= instr;
        end
        else begin
                instr <= data1;
        end
end

// data memory
always @(posedge clk ) begin
        if (!reset_n) begin
                mem_read_data <= 0;
        end else begin
                if(read_m2) begin
                        mem_read_data <= data2;
                end
        end
end
```

[그림 4] CPU(Datapath) 모듈의 코드 중 일부. 메모리의 값을 읽어오거나 쓰기 위해 control signal과 적절한 데이터를 연결하고, Posedge clk마다 instruction과 memory data를 읽어와 저장한다. 이 때 stall의 경우 ID 단계의 pipeline register값의 변경을 막아야 하기 때문에 instruction을 새로 가져오지 않는다.

```
// set flush
always @(*) begin
        fcond1 = (actual_pc != pc_id);
        fcond2 = (pc_id != 0);
        fcond3 = (pc_ex != `WORD_SIZE'hffff);

        flush = (fcond1 & fcond2 & fcond3) ? 1: 0;
end

// set halt
assign is_halted = halt;

// get branch always taken pc
assign actual_taken_pc = branch_ex ? (pc_ex + `WORD_SIZE'b1 + immed_ex) : (pc_src_ex == 2'b01 ? {pc_ex[15:12], target} : rf_rs_forwarded);
```

```
always @(posedge clk) begin
        if (wwd_wb) begin
                output_port <= rf_rs_wb;
        end

        if (new_inst_wb) begin
                num_inst <= num_inst + 1;
        end
end
```

[그림 5] CPU(Datapath) 모듈의 코드 중 일부. flush를 결정하는 로직과 halt 시그널을 전달하고 wwd일 때 output_port로 적절한 값을 보내며 num_inst를 증가시키는 로직이다. Actual_taken_pc의 경우 always taken branch predictor를 구현하기 위해 계산을 따로 해주었다.

Datapath 모듈은 CPU 모듈과 동일한 input, output, inout을 가지고, 세부 구현은 모두 datapath 모듈에 구현하였다. CPU 모듈(datapath 모듈)에는 각 서브 모듈의 input과 output을 전달할 수 있는 각종 wire 변수와, 메모리에서 읽어온 instruction 또는 데이터를 저장하는 data1, 2와 파이프라인으로 흘려줄 레지스터 변수 및 control signal들을 가지고 있으며 각 서브 모듈의 인스턴스들이 선언되어 있다. CPU 모듈(datapath 모듈)에서는 테스트와 출력을 위해 num_inst를 업데이트시켜주고, output_port에 레지스터 값을 출력해주며, 메모리에서 읽어온 데이터를 저장하고, 여러 서브 모듈간의 data를 연결하는 datapath 전반을 구현하였다.

3.2. control_unit 모듈

```
// generate control signals
always @(*) begin
        halt = 0;
        wwd = 0;
        reg_dest = 0;
        use_rs = 0;
        use_rt = 0;
        alu_src = 0;
        branch = 0;
        mem_read = 0;
        mem_write = 0;
        reg_write = 0;
        pc_src = 0;
        reg_src = 0;
        new_inst = 0;

        case(opcode)
                4'd15: begin
                        case(func_code)
                                `INST_FUNC_ADD, `INST_FUNC_SUB, `INST_FUNC_AND, `INST_FUNC_ORR: begin
                                        use_rs = 1;
                                        use_rt = 1;
                                        reg_write = 1;
                                        new_inst = 1;
                                end
                                `INST_FUNC_NOT, `INST_FUNC_TCP, `INST_FUNC_SHL, `INST_FUNC_SHR: begin
                                        use_rs = 1;
                                        reg_write = 1;
                                        new_inst = 1;
                                end
                                `INST_FUNC_JPR: begin
                                        use_rs = 1;
                                        pc_src = 2'b10;
                                        new_inst = 1;
                                end
```

[그림 6] control_unit 모듈의 코드 중 일부. 현재 instruction에 맞게 출력할 signal들을 결정한다.

```
// generate alu control signals
always @(*) begin
        alu_func_code = 4'd15;
        alu_branch_type = 4'd0;

        case (opcode)
                `ALU_OP: begin
                        case(func_code)
                                `INST_FUNC_JPR: alu_func_code = `FUNC_ID1; // pc <- rs
                                `INST_FUNC_JRL: alu_func_code = `FUNC_ID1; // pc <- rs
                                `INST_FUNC_WWD: alu_func_code = `FUNC_ID1; // outputport <- rs
                                default: alu_func_code = func_code[3:0];
                        endcase
                end
                `ADI_OP: alu_func_code = `FUNC_ADD;
                `ORI_OP: alu_func_code = `FUNC_ORR;
                `LHI_OP: alu_func_code = `FUNC_ID2; // immediate : alu_input_2
                `LWD_OP, `SWD_OP: alu_func_code = `FUNC_ADD;
                `BNE_OP, `BEQ_OP ,`BGZ_OP, `BLZ_OP: begin
                        alu_func_code = `FUNC_Bxx;
                        alu_branch_type = opcode[1:0]; //branch type for bne = 0, beq = 1, bgz = 2, blz = 3
                end
        endcase
end
```

[그림 7] control_unit 모듈에서 alu에서 사용되는 alu_func_code와 alu_branch_type을 결정하는 코드. Instruction의 opcode와 funct에 따라 적절한 alu_func_code, alu_branch_type을 생성해 출력한다.

그림 4, 5는 control_unit 모듈에서 instruction의 opcode와 func_code를 바탕으로 control signal들을 생성하는 combinational logic을 나타낸다. 그림 5에서는 일부 케이스만 나타냈지만, 실제 코드에서는 각 instruction의 opcode와 func_code에 따라 각각 다른 control signal을 생성한다. 각 instruction에 따른 control signal의 값은 표 2을 따른다.

3.3. alu 모듈

```
// localparam for branch types
   localparam BNE = 2'd0;
   localparam BEQ = 2'd1;
   localparam BGZ = 2'd2;
   localparam BLZ = 2'd3;

   always @(*) begin

      // reset output
      alu_out = 0;
      overflow_flag = 0;

      case(func_code)
         `FUNC_ADD: begin
            alu_out = A + B;
            // NOTE : From Soomin's labl overflow detection
            if((A[`NumBits - 1] === B[`NumBits - 1])
            && (A[`NumBits - 1] !== alu_out[`NumBits -1]))
               overflow_flag = 1;
            else
               overflow_flag = 0;
         end
         `FUNC_SUB: begin
            alu_out = A - B;
            if((A[`NumBits - 1] !== B[`NumBits - 1])
            && (A[`NumBits - 1] !== alu_out[`NumBits -1]))
               overflow_flag = 1;
            else
               overflow_flag = 0;
         end
         `FUNC_AND: alu_out = A & B;
         `FUNC_ORR: alu_out = A | B;
         `FUNC_NOT: alu_out = ~A;
         `FUNC_TCP: alu_out = ~A + 1;
         `FUNC_SHL: alu_out = {A[14:0], 1'b0};
         `FUNC_SHR: alu_out = {A[15], A[15:1]};
         `FUNC_ID1: alu_out = A;
         `FUNC_ID2: alu_out = B;

         `FUNC_Bxx: begin
            case(branch_type)
               BNE: bcond = (A != B)? 1 : 0; // 1 if not equal
               BEQ: bcond = (A == B)? 1 : 0; // 1 if equal
               BGZ: bcond = (A[15] == 0 && A != 0)? 1 : 0; //  1 if A > 0
               BLZ: bcond = A[15]; //  1 if A < 0
            endcase
            alu_out = bcond;
         end

         default: begin // not happen
            alu_out = 0;
            overflow_flag = 0;
            bcond = 0;
         end
      endcase
   end
```

[그림 8] alu 모듈의 코드. func_code와 branch_type에 따라 두 input에 대해 적절한 연산을 수행하여 그 결과를 반환한다. Overflow를 체크하고, branch condition이 맞는 경우에는 bcond를 인가시킨다.

alu 모듈은 control_unit에서 생성한 alu_func_code와 alu_branch_type에 따라 적절한 연산을 수행하여 output해주도록 구현하였다. ADD, SUB연산시에는 overflow 여부를 체크하여 overflow_flag를 인가시키고, branch type에 따라 조건이 맞는 경우 bcond를 인가시킨다.

3.4. hazard 모듈

```
module hazard_detect(IFID_IR, IDEX_rd, use_rs, use_rt, IDEX_M_mem_read, is_stall);

        // Stall with forwarding
        input [`WORD_SIZE-1:0] IFID_IR;
        input [1:0]  IDEX_rd;
        input use_rs, use_rt;
        input IDEX_M_mem_read;

        output reg is_stall;

        wire [1:0] IFID_rs, IFID_rt;

        assign IFID_rs = IFID_IR[11:10];
        assign IFID_rt = IFID_IR[9:8];

        always @(*) begin
                is_stall = (((IFID_rs == IDEX_rd) && use_rs)
                 || ((IFID_rt == IDEX_rd) && use_rt)) && IDEX_M_mem_read ? 1:0;
        end

endmodule
```

[그림 9] hazard 모듈의 코드. ID 단계의 instruction이 rs, rt를 사용하고
그 레지스터가 EX단계의 rd와 동일할 경우 stall한다.

Hazard 모듈은 LWD 명령어 뒤에 결과값을 읽는 명령어가 나올 때 지연(stall)을 추가할 수 있게 한다. IF/ID 파이프라인 레지스터 rs, rt와 ID/EX 파이프라인 레지스터 rd, mem_read control을 입력받아 mem_read이면서, ID 단계의 instruction이 rs 혹은 rt를 사용하고 그 레지스터가 rd와 동일하다면 is_stall을 assert시켜 datapath 모듈에서 stall 여부를 판단할 수 있게 한다.

3.5. forwarding_unit 모듈

```
module forwarding_unit(rs_EX, rt_EX, rd_MEM, reg_write_MEM, rd_WB, reg_write_WB,
                       forward_A, forward_B);

    input [1:0] rs_EX, rt_EX, rd_MEM, rd_WB;
    input reg_write_MEM, reg_write_WB;

    output reg [1:0] forward_A, forward_B;

    always @(*) begin
        if((rs_EX == rd_MEM) && reg_write_MEM) begin
            assign forward_A = 2'b01;
        end else if((rs_EX == rd_WB) && reg_write_WB) begin
            assign forward_A = 2'b10;
        end else begin
            assign forward_A = 2'b00;
        end

        if((rt_EX == rd_MEM) && reg_write_MEM) begin
            assign forward_B = 2'b01;
        end else if((rt_EX == rd_WB) && reg_write_WB) begin
            assign forward_B = 2'b10;
        end else begin
            assign forward_B = 2'b00;
        end
    end
endmodule
```

[그림 10] forwarding_unit 모듈의 코드. EX 단계에서 ALU input에 해당하는 rs, rt가 MEM, WB 단계의 rd와 같으면 forwarding을 하도록 forwarding signal을 설정한다.

Forwarding unit은 EX 단계의 rs, rt와 MEM, WB 단계의 rd가 값고 MEM, WB 단계의 instruction이 레지스터에 값을 쓸 경우 forwarding하도록 적절한 signal을 출력으로 전달한다. 해당 조건은 설계 단계에서 결정한 Forwarding logic을 그대로 따른다.

3.6. branch_predictor 모듈

```
branch_predictor_global_predictor BranchPredictor(
        .clk(clk),
        .reset_n(reset_n),
        .PC(pc),
        .is_flush(flush),
        .is_BJ_type(branch_ex || (pc_src_ex != 2'b0)),
        .actual_taken_PC(actual_taken_pc),
        .actual_next_PC(actual_pc),
        .actual_PC(pc_ex),
        .next_PC(pc_nxt)
);
```

[그림 11] datapath 모듈 내에서의 branch_predictor 모듈의 선언부.

각 branch_predictor 모듈은 IF 단계의 PC를 input으로 받아 다음에 fetch해올 next_PC를 output으로 출력한다. 이 때 input들 중 actual_PC는 EX 단계의 PC값, actual_next_PC는 EX 단계에서의 실제 다음 PC값에 해당하며, actual_taken_PC는 EX 단계에 있는 branch, jump instruction가 taken됐을 때의 target PC에 해당한다. is_BJ_type은 BTB의 업데이트 여부를 결정하기 위해 사용되고, 업데이트 시 BTB의 index와 새로운 tag를 actual_PC로부터 가져온다. is_flush는 예측 실패 여부를 알고, next_PC를 acutal_next_PC로 설정하기 위해 사용된다.

1) branch_predictor_always_not_taken 모듈

```
assign next_PC = (is_flush | !(PC < 16'hc6))? actual_next_PC: PC + 1;
```

[그림 12] branch_predictor_always_not_taken 모듈의 코드 중 일부. 모든 instruction에 대해 다음 PC값을 PC+1로 예측한다.

항상 다음 PC를 PC+1로 예측하고, flush인 경우에는 input의 실제 PC로 다음 PC를 설정한다.

2) branch_predictor_always_taken 모듈

```
integer i;
reg [`TAG_SIZE:0] tagtable [0:(2**`IDX_SIZE)-1];
reg [`WORD_SIZE-1:0] btb [0:(2**`IDX_SIZE)-1];

wire [`TAG_SIZE-1:0] tag; wire [`IDX_SIZE-1:0] idx;
assign tag = PC[`WORD_SIZE-1:`IDX_SIZE];
assign idx = PC[`IDX_SIZE-1:0];

always @(posedge clk) begin
        if (!reset_n) begin
                tagtable[0] <= ~0;      tagtable[1] <= ~0;      tagtable[2] <= ~0;      tagtable[3] <= ~0;
                tagtable[4] <= ~0;      tagtable[5] <= ~0;      tagtable[6] <= ~0;      tagtable[7] <= ~0;
                tagtable[8] <= ~0;      tagtable[9] <= ~0;      tagtable[10] <= ~0;     tagtable[11] <= ~0;
                tagtable[12] <= ~0;     tagtable[13] <= ~0;     tagtable[14] <= ~0;     tagtable[15] <= ~0;
                tagtable[16] <= ~0;     tagtable[17] <= ~0;     tagtable[18] <= ~0;     tagtable[19] <= ~0;
                tagtable[20] <= ~0;     tagtable[21] <= ~0;     tagtable[22] <= ~0;     tagtable[23] <= ~0;
                tagtable[24] <= ~0;     tagtable[25] <= ~0;     tagtable[26] <= ~0;     tagtable[27] <= ~0;
                tagtable[28] <= ~0;     tagtable[29] <= ~0;     tagtable[30] <= ~0;     tagtable[31] <= ~0;

                btb[0] <= ~0;   btb[1] <= ~0;   btb[2] <= ~0;   btb[3] <= ~0;
                btb[4] <= ~0;   btb[5] <= ~0;   btb[6] <= ~0;   btb[7] <= ~0;
                btb[8] <= ~0;   btb[9] <= ~0;   btb[10] <= ~0;  btb[11] <= ~0;
                btb[12] <= ~0;  btb[13] <= ~0;  btb[14] <= ~0;  btb[15] <= ~0;
                btb[16] <= ~0;  btb[17] <= ~0;  btb[18] <= ~0;  btb[19] <= ~0;
                btb[20] <= ~0;  btb[21] <= ~0;  btb[22] <= ~0;  btb[23] <= ~0;
                btb[24] <= ~0;  btb[25] <= ~0;  btb[26] <= ~0;  btb[27] <= ~0;
                btb[28] <= ~0;  btb[29] <= ~0;  btb[30] <= ~0;  btb[31] <= ~0;

        end else begin
                if (is_BJ_type) begin
                        tagtable[actual_PC[`IDX_SIZE-1:0]] <= actual_PC[`WORD_SIZE-1:`IDX_SIZE];
                        btb[actual_PC[`IDX_SIZE-1:0]] <= actual_taken_PC;
                end
        end
end

always @(*) begin
        if (is_flush) begin
                next_PC = actual_next_PC;
        end else if (tagtable[idx] == tag) begin
                next_PC = btb[idx];
        end else begin
                next_PC = !(PC < 16'hc6)? PC: PC + 1;
        end
end
```

[그림 13] branch_predictor_always_taken 모듈의 코드 중 일부. Clock synchronous하게 btb와 tagtable을 업데이트하며, 다음 PC값을 예측하는 것은 combinational logic으로 구성되어 있다. 이 때 특정 entry와 매칭되는 경우 next_PC값을 btb에 저장된 taken PC값을 사용하고, 그 외의 경우 다음 PC값을 PC+1로 예측한다.

EX 단계에서 branch가 taken 됐을 때의 PC를 계산하면, 이를 branch predictor 모듈로 전달하여 btb와 tagtable을 posedge clk에 업데이트한다. 입력으로 들어온 PC에 대해 tag 부분과 idx 부분을 나누고, idx로 btb와 tagtable을 조회하여 PC의 tag와 tabtable의 tag을 비교한다. 이 두 값이 일치하면 BTB의 PC값을, 없으면 현재 PC+1을 예측값으로 사용한다. flush인 경우에는 input의 실제 PC로 다음 PC를 설정한다.

3) branch_predictor_global_predictor 모듈

```
if (is_BJ_type) begin
        tagtable[actual_PC[`IDX_SIZE-1:0]] <= actual_PC[`WORD_SIZE-1:`IDX_SIZE];
        btb[actual_PC[`IDX_SIZE-1:0]] <= actual_taken_PC;

        if(actual_next_PC != actual_taken_PC) begin // actually not taken

                // global saturation counter
                if(sat_cnt == 2'b00)            sat_cnt <= 2'b00;
                else if(sat_cnt == 2'b01)       sat_cnt <= 2'b00;
                else if(sat_cnt == 2'b10)       sat_cnt <= 2'b01;
                else                                    sat_cnt <= 2'b10;

                // global hysteresis counter
                if(hys_cnt == 2'b00)            hys_cnt <= 2'b00;
                else if(hys_cnt == 2'b01)       hys_cnt <= 2'b00;
                else if(hys_cnt == 2'b10)       hys_cnt <= 2'b00;
                else                                    hys_cnt <= 2'b10;

                // indexed saturation counter
                if(bht_sat[actual_PC[`IDX_SIZE-1:0]] == 2'b00)          bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b00;
                else if(bht_sat[actual_PC[`IDX_SIZE-1:0]] == 2'b01) bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b00;
                else if(bht_sat[actual_PC[`IDX_SIZE-1:0]] == 2'b10) bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b01;
                else                                                                                    bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b10;

                // indexed hysteresis counter
                if(bht_hys[actual_PC[`IDX_SIZE-1:0]] == 2'b00)          bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b00;
                else if(bht_hys[actual_PC[`IDX_SIZE-1:0]] == 2'b01) bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b00;
                else if(bht_hys[actual_PC[`IDX_SIZE-1:0]] == 2'b10) bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b00;
                else                                                                                    bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b10;

        end else begin // actually taken

                // global saturation counter
                if(sat_cnt == 2'b00)            sat_cnt <= 2'b01;
                else if(sat_cnt == 2'b01)       sat_cnt <= 2'b10;
                else if(sat_cnt == 2'b10)       sat_cnt <= 2'b11;
                else                                    sat_cnt <= 2'b11;

                // global hysteresis counter
                if(hys_cnt == 2'b00)            hys_cnt <= 2'b01;
                else if(hys_cnt == 2'b01)       hys_cnt <= 2'b11;
                else if(hys_cnt == 2'b10)       hys_cnt <= 2'b11;
                else                                    hys_cnt <= 2'b11;

                // indexed saturation counter
                if(bht_sat[actual_PC[`IDX_SIZE-1:0]] == 2'b00)          bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b01;
                else if(bht_sat[actual_PC[`IDX_SIZE-1:0]] == 2'b01) bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b10;
                else if(bht_sat[actual_PC[`IDX_SIZE-1:0]] == 2'b10) bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b11;
                else                                                                                    bht_sat[actual_PC[`IDX_SIZE-1:0]] <= 2'b11;

                // indexed hysteresis counter
                if(bht_hys[actual_PC[`IDX_SIZE-1:0]] == 2'b00)          bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b01;
                else if(bht_hys[actual_PC[`IDX_SIZE-1:0]] == 2'b01) bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b11;
                else if(bht_hys[actual_PC[`IDX_SIZE-1:0]] == 2'b10) bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b11;
                else                                                                                    bht_hys[actual_PC[`IDX_SIZE-1:0]] <= 2'b11;
        end
end
```

[그림 14] branch_predictor_global_predictor 모듈의 코드 중 일부. Clock synchronous하게 btb와 tagtable, predictor state를 업데이트한다. 각 구현 방식에 따라 global counter를 업데이트하거나 per-address counter를 업데이트한다.

```
always @(*) begin
        if (is_flush) begin
                next_PC = actual_next_PC;
        end else if (tagtable[idx] == tag) begin
                next_PC = (sat_cnt >= 2'b10)? btb[idx] : (!(PC < 16'hc6)? PC: PC + 1);
                // next_PC = (hys_cnt >= 2'b10)? btb[idx] : (!(PC < 16'hc6)? PC: PC + 1);
                // next_PC = (bht_sat[idx] >= 2'b10)? btb[idx] : (!(PC < 16'hc6)? PC: PC + 1);
                // next_PC = (bht_hys[idx] >= 2'b10)? btb[idx] : (!(PC < 16'hc6)? PC: PC + 1);
        end else begin
                next_PC = !(PC < 16'hc6)? PC: PC + 1;
        end
end
```

[그림 15] branch_predictor_global_predictor 모듈의 코드 중 일부. 다음 PC값을 예측하는 것은 combinational logic으로 구성되어 있다. 이 때 특정 entry와 매칭되는 경우 next_PC값을 btb에 저장된 taken PC값을 사용하고, 그 외의 경우 다음 PC값을 PC+1로 예측한다.

always_taken 모듈과 마찬가지로 BTB에 taken PC값을 저장한다. 2-bit global predictor의 경우, branch, jump instruction이 taken되었는지에 대한 값을 바탕으로 2비트 global state를 업데이트하고 이로부터 예측값을 결정한다. 2비트 state machine은 saturation counter와 hysteresis counter 방식을 사용한다. 2-bit per-address predictor의 경우 branch, jump instruction이 taken되었는지에 대한 정보를 바탕으로 BTB의 각 entry별 2비트 state를 업데이트하고, 이 entry별 state로부터 예측값을 결정한다. flush인 경우에는 input의 실제 PC로 다음 PC를 설정한다.

3.8. register_file 모듈 기타 모듈

```
reg [`WORD_SIZE-1:0] RF [`NUM_REGS-1:0]; // 4 registers each 16 bits long

initial begin
    RF[0] = `WORD_SIZE'b0;
    RF[1] = `WORD_SIZE'b0;
    RF[2] = `WORD_SIZE'b0;
    RF[3] = `WORD_SIZE'b0;
end

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

always @(negedge clk) begin
            if (!reset_n) begin
                    RF[0] = `WORD_SIZE'b0;
                    RF[1] = `WORD_SIZE'b0;
                    RF[2] = `WORD_SIZE'b0;
                    RF[3] = `WORD_SIZE'b0;
            end
            else begin
                    // write back if reg_write is high
                    if (reg_write) RF[dest] <= write_data;
                    else RF[dest] <= RF[dest];
            end
end
```

[그림 16] register_file 모듈의 일부. 16비트짜리 reg 변수 4개가 있는 array를 선언하고 있으며, input에 따라 적절한 register의 값을 출력하고 변경한다.

register_file 모듈은 16비트짜리 register 4개를 사용하여 데이터를 저장하고 있으며, input의 read1, read2에 해당하는 register값을 읽어 output해주는 동시에, reg_write가 assert된 경우 negedge clk에 synchronous하게 해당 레지스터의 값을 변경한다. Negedge clk을 사용하는 이유는 pipeline register가 업데이트된 이후 레지스터를 업데이트해야 하기 때문에, posedge clk에 pipeline register를 업데이트하고 난 뒤 레지스터가 업데이트되도록 하였다. 또한 dist = 3인 data hazard가 발생할 경우, negedge clk에 write가 일어난 후 read하는 값이 올바른 값으로 전달되므로 register file 내부적으로 internal forwarding이 적용된다.

3.9. 기타 모듈

immediate_generator, mux2_1, mux4_1 모듈을 구현하여 그림 2와 같이 연결해주었다. 이 모듈들은 control signal에 의존하여 적절한 동작을 수행한다.

4. Discussion

- **Multicycle CPU와 결과 비교**

  Multicycle CPU

```
# Clock # 3816
# The testbench is finished. Summarizing...
# All Pass!
```

  Pipelined CPU (always not taken)

```
# Clock # 1435
# The testbench is finished. Summarizing...
# All Pass!
```

  Multicycle CPU에서 3816 Clock이 걸리는 반면, Pipelined CPU에선 1435 Clock이 걸려 약 2.66에 해당하는 speedup이 되었다. 이로부터 여러 instruction을 중첩해 실행하는 Pipeline CPU가 Multicycle CPU보다 훨씬 빠른 것을 확인할 수 있었다.

- **Branch predictor 결과 비교**

  1) Always not taken

```
# Clock # 1435
# The testbench is finished. Summarizing...
# All Pass!
```

  2) Always taken

```
# Clock # 1227
# The testbench is finished. Summarizing...
# All Pass!
```

  Always not taken의 경우 1435 Clock이 걸리는 반면, always taken의 경우 1227 Clock이 걸려 약 1.17에 해당하는 speedup이 되었다. 이로부터 branch, jump instruction을 항상 taken으로 예측하는 경우가 그렇지 않은 경우보다 성능이 향상되는 것을 확인할 수 있었다. 이는 misprediction rate를 줄인 경우에 해당한다.

  3) 2-bit global saturation counter

```
# Clock # 1227
# The testbench is finished. Summarizing...
# All Pass!
```

  4) 2-bit global hysteresis counter

```
# Clock # 1227
# The testbench is finished. Summarizing...
# All Pass!
```

  5) 2-bit per-address saturation counter

```
# Clock # 1263
# The testbench is finished. Summarizing...
# All Pass!
```

  6) 2-bit per-address hysteresis counter

```
# Clock # 1255
# The testbench is finished. Summarizing...
# All Pass!
```

2-bit predictor를 사용한 경우 종류에 무관하게 Always not taken보다 약 1.14~1.17에 해당하는 speedup을 가진다. 하지만 always taken과 비교했을 때 2-bit global predictor의 경우 성능에 큰 향상이 없었다. 이는 주어진 instruction의 특성상 global에서 연속적으로 not taken이 나오는 경우가 없어 2-bit global predictor와 always taken의 결과가 동일해진 것으로 보인다. 또한 2-bit per-address predictor의 경우 오히려 always taken보다 성능이 더 하락했다. 이는 주어진 instruction의 마지막 fibonacci 계산하는 부분에서 서로 다른 JAL에 대한 JPR 반복적으로 사용되어 일정한 target을 갖지 않는 경우가 많아 오히려 flush가 늘어난 것으로 보인다.

- Always not taken predictor이거나 다른 predictor를 사용했을 때에도 BTB에 해당 PC가 등록되지 않았다면 nextPC = PC + 1 로 예측하게 된다. 이는 거의 모든 경우에서 문제를 일으키지 않지만, 마지막 PC에서 PC + 1을 하게 되면 메모리 범위를 초과하게 되어 오류가 발생한다. 이번 testbench의 경우 16'hc6이 마지막 PC였기 때문에 이 값을 초과하지 않도록 branch predictor에 (PC < 16'hc6) 조건을 추가하여 오류를 해결하였다.

- Nop을 구분하기 위해 nop = 16'FFFF로 설정하여 구현하였다.

- Flush는 branch의 예측값이 잘못되어 실제로 실행해야 할 pc와 다를 경우 발생한다. 이는 (pc_id != actual_pc) 조건문으로 판단 가능한데, 이 조건문만으로는 파이프라인이 새로 시작하여 pc_id == 0일 때, Flush를 실행하고 있어 pc_ex == 16'FFFF 일 때의 경우도 Flush 조건으로 판단할 수 있다. 따라서 추가적으로 이들을 제외하는 조건을 작성하여 최종적으로 구현한 flush의 조건은 flush = (pc_id != actual_pc) && (pc_id != 0) && (pc_ex != 16'hFFFF)로 구현하였다.

- Data Hazard에 있어서 dist = 1, 2인 경우는 forwarding unit의 구현을 통해 해결할 수 있었고, dist = 3인 경우는 Register File의 read를 combinational로 주어 internal forwarding이 되도록 하여 해결할 수 있었다.

5. Conclusion

Pipeline CPU를 구현하여 이전에 구현했던 multicycle CPU와 실행 clock 비교를 파이프라이닝 기법이 throughput의 향상을 크게 가져올 수 있는 것을 확인하였고, 파이프라인으로 인해 발생하는 각종 data, control hazard를 stall, flush, forwarding으로 해결하였다. Branch predictor 방식도 다양하게 구현해 봄으로써 정확한 predictor를 만드는 것 또한 throughput 향상에 기여할 수 있다는 것을 확인하였다.