컴퓨터 구조 (CSED311)

<Lab4-2 - Pipelined CPU with Control Hazard>

Team ID : 33
Student 1 : 곽민성 (Gwak Minseong, 20230840)
Student 2 : 김재환 (Kim Jaehwan, 20230499)

1. Introduction

이번 랩의 목표는 Pipelined CPU를 Verilog를 통해 구현하는 것이다. 지난 랩에서는 Data Hazard만을 고려하여 Pipelined CPU를 구현하였지만, 이번 랩에서는 Control Hazard 상황에서까지 올바르게 동작하는 CPU를 만들어야 한다. 이 랩을 성공적으로 진행함으로써 Pipelined CPU의 세부 구조와 동작을 정확하게 이해할 수 있었다.

2. Design

우리는 아래 강의 노트의 CPU 구조를 사용하여 Pipelined CPU를 구현하였다.

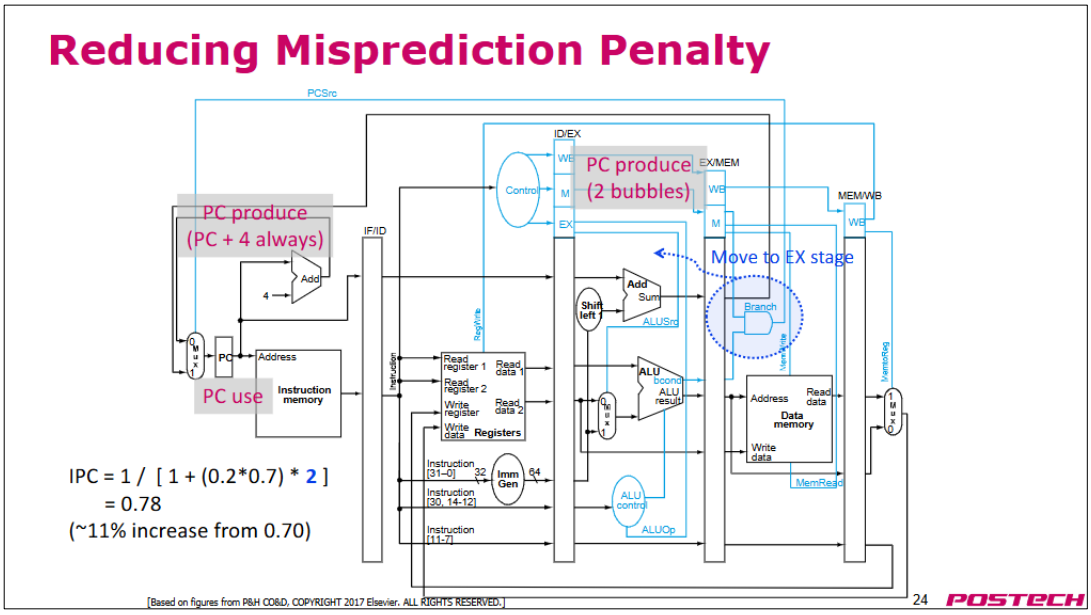

그림 1 Pipelined CPU with producing PC at EX stage.

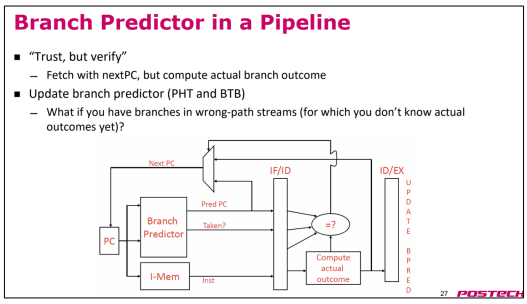

그림 2 Pipelined CPU with Branch Predictor

	R/I-Type	LD	SD	Bxx	JAL	JALR
IF	use (produce)	use (produce)	use (produce)	use	use	use
ID						
EX				produce	produce	produce
MEM						
WB						

그림 3 Our PC Producing and usage diagram. (Slightly modified, since our implementation of Pipelined CPU produces next PC of Bxx, JAL instruction at EX)

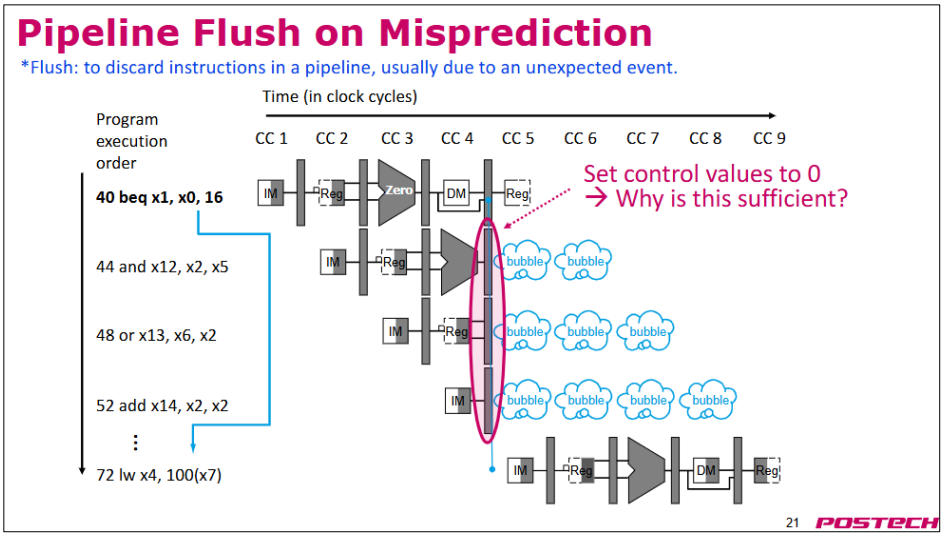

그림 4 We need 2 bubble to handle misprediction.

Pipelined CPU에서 모든 명령어는 IF, ID, EX, MEM, WB를 통과하는데 각 한 사이클씩 거치게 되며, 해당 단계를 수행하는데 필요한 전 단계의 명령어의 수행 결과와 Control Unit의 신호는 중간 단계의 레지스터에 저장되게 된다 (IF/ID, ID/EX, EX/MEM, MEM/WB). Pipelined CPU의 신호는 다른 사이클에 그 신호가 사용된다는 점을 빼면, 기본적으로 Single-cycle CPU와 동일하기 때문에 Lab2에서 구현했던 Single-cycle CPU의 Control Unit을 사용해서 구현하였다. 또, 우리는 Lab 4-1에서 사용한 구현에서 다음 모듈을 추가해서 이번 Lab 4-2를 성공적으로 수행하였다.

Lab4-1에서 이번에 추가된 모듈은 다음과 같다. 기존 모듈은 이전 랩 보고서를 참고하라.

모듈 이름	설명	Synchronous / Asynchronous
`ControlflowDetectionUnit`	Instruction을 입력으로 받아, 받은 instruction이 Branch, JAL, JALR중 하나인지 검사하여 반환한다.	Asynchronous
`BranchPredictUnit`	Current PC를 입력으로 받아, 다음 PC가 무엇일지 Gshare 방법을 통해 예측한 Predict PC를 반환한다. 또, 특정 PC에서 taken / not taken으로 예측했는지를 입력으로 받아 BTB를 clock cycle에 맞추어 Gshare를 업데이트한다.	Asynchronous (Predict) / Synchronous (Update)
`BubbleGen`	IF/ID/EX 단계에서 다시 돌아오는 잘못 예측했다는 신호를 받아, 어떤 모듈에 버블을 만들어야 할지 반환한다.	Asynchronous

3. Implementation

우리는 위 모듈들을 다음과 같이 구현하였다.

ControlflowDetectionUnit

```
`include "opcodes.v"

module ControlflowDetectUnit(
    input [6:0] Instr,
    output is_ctrlflow
);

assign is_ctrlflow = (Instr == `BRANCH) || (Instr == `JAL) || (Instr == `JALR);

endmodule

```

BranchPredictUnit

```
module BranchPredictUnit(
    input [31:0] current_pc,
    output[31:0] predict_pc,
    input        clk,
    input        reset,

    input        update,
    input [31:0] faux_pas_pc,
    input        actual_behavior
);

wire [24:0] tag = current_pc[31:7];
wire [4:0] index = current_pc[6:2];

reg [24:0] tag_table[0:31];
reg [31:0] BTB[0:31];
reg [4:0] BHSR;
reg [1:0] PHT[0:31];

wire [24:0] upd_tag = faux_pas_pc[31:7];
wire [4:0] upd_index = faux_pas_pc[6:2];


always @(posedge clk) begin
    if (reset) begin
        BHSR <= 5'b0;
        for (integer i = 0; i < 32; i = i + 1) begin
            tag_table[i] <= 25'b0;
            BTB[i] <= 32'b0;
            PHT[i] <= 2'b00;
        end
    end
    else if(update) begin
        BHSR <= {BHSR[3:0], actual_behavior};
        tag_table[upd_index] <= upd_tag;
        BTB[upd_index] <= faux_pas_pc;
        if (actual_behavior) begin
            if (PHT[upd_index ^ BHSR] < 2'b11) begin
                PHT[upd_index ^ BHSR] <= PHT[upd_index ^ BHSR] + 1;
            end
        end
        else begin
            if (PHT[upd_index ^ BHSR] > 2'b00) begin
                PHT[upd_index ^ BHSR] <= PHT[upd_index ^ BHSR] - 1;
            end
        end
    end
end

assign predict_pc = (tag == tag_table[index]) && (PHT[index ^ BHSR] > 2'b01) ? BTB[index] : current_pc + 4;
// assign predict_pc = current_pc + 4; // always not taken

endmodule

```

BubbleGen

```
module BubbleGen(
    // input taken,
    // input hit,
    input IF_wrong,
    input ID_wrong,
    input EX_wrong,
    output IF_is_bubble,
    output ID_is_bubble
);

assign ID_is_bubble = EX_wrong;
assign IF_is_bubble = EX_wrong || ID_wrong;
// assign isbubble = taken ^ hit;

endmodule

```

4. Discussion

A. Control Hazard가 발생하는 이유

만약 어떠한 조치도 취하지 않고 Pipelined CPU를 구현한 경우 control 문(Bxx, JAL, JALR)이 아닌경우에도 다음 PC는 항상 ID단계 (또는 그 이상)에서 나오기 때문에, 한 명령어마다 항상 한 번 이상 stall 되어야 하는 문제가 있다. 이를 해결하기 위해서 Branch prediction 방법을 사용하였다. 저번 랩에서는 다음 PC를 항상 현재 PC에 4를 더한 값으로 예측하였음과 동시에 control 문이 없는 테스트만 진행하였으나, 이번 랩에서는 control문을 적절히 처리하도록 구현해야 한다.

B. 어떻게 Branch prediction을 다루었는가?

Branch Prediction을 어떻게 하는지와 상관없이, 다음 로직을 구현해야한다. 모든 예측이 틀리더라도 적절하게 수행하여야 한다. (Gshare를 통해 Branch Prediction을 구현하였다)

1. Branch Prediction이 맞는 경우: 그대로 진행한다.
2. Branch Prediction이 틀린 경우: 이전에 잘못 예측함으로 인해서 실행되지 말았어야할 명령어를 취소한다. 즉, nop으로 만들기 위해 write 신호를 포함한 컨트롤 신호를 끈다.

구현상으로는 다음과 같다.

1. 잘못 예측한 경우를 감지하고 `단계_pred_wrong` 와이어 변수를 통해 신호를 보낸다.
2. 이를 `BubbleGen`모듈에서 받고, `단계_is_bubble` 와이어를 통해 해당 단계를 nop로 만들라는 신호를 준다.
3. 각 단계의 레지스터에서 `단계_is_bubble` 와이어 변수를 받으면 해당 단계에서 전달되는 대부분 컨트롤 신호를 0으로 처리한다.

C. Gshare와 always taken, always not taken 사이클 수 비교

Test name	Gshare	Always not taken	Always taken
basic	35	35	35
ifelse	43	43	51
loop	326	322	326
non-controlflow	46	46	46
recursive	1203	1187	1229

* 모든 레지스터의 값은 모두 동일하였다. 즉, Branch Prediction이 어떻든, 심지어 항상 PC를 0으로 예측하더라도 올바르게 동작한다.
* control 명령어가 없는 basic, non-controlflow 테스트는 사이클 수가 동일하였다.
* ifelse 테스트는 Always taken만 사이클 수가 높게 나왔다. 이는 Always taken 방법이 ifelse보다 반복문에 더 친숙하며, ifelse에서는 잘못된 예측이 많은 것으로 생각된다.
* loop와 recursive 테스트에서 Gshare 구현이 Always not taken보다 오히려 더 사이클 수가 증가했는데, 이는 Gshare가 처음에 초기화 된 상태로 시작하고, 분기 예측에 익숙해지기까지 시간이 부족했기 때문이라고 생각된다.

5. Conclusion

위에서 설명한 구조대로 CPU를 구현하여 제공된 모든 테스트 벤치에 대해서 정답과 같은 시뮬레이션 레지스터 결과를 얻었다.

	Test Bench
Register Result	FINAL REGISTER OUTPUT 0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002ffc (Answer : 00002ffc) 3 00000000 (Answer : 00000000) 4 00000000 (Answer : 00000000) 5 00000000 (Answer : 00000000) 6 00000000 (Answer : 00000000) 7 00000000 (Answer : 00000000) 8 00000000 (Answer : 00000000) 9 00000000 (Answer : 00000000) 10 0000000a (Answer : 0000000a) 11 0000003f (Answer : 0000003f) 12 fffffff1 (Answer : fffffff1) 13 0000002f (Answer : 0000002f) 14 0000000e (Answer : 0000000e) 15 00000021 (Answer : 00000021) 16 0000000a (Answer : 0000000a) 17 0000000a (Answer : 0000000a) 18 00000000 (Answer : 00000000) 19 00000000 (Answer : 00000000) 20 00000000 (Answer : 00000000) 21 00000000 (Answer : 00000000) 22 00000000 (Answer : 00000000) 23 00000000 (Answer : 00000000) 24 00000000 (Answer : 00000000) 25 00000000 (Answer : 00000000) 26 00000000 (Answer : 00000000) 27 00000000 (Answer : 00000000) 28 00000000 (Answer : 00000000) 29 00000000 (Answer : 00000000) 30 00000000 (Answer : 00000000) 31 00000000 (Answer : 00000000) Correct output : 32/32

위와 같이 정상 동작하는 모습을 확인할 수 있다. 사이클 수는 다음과 같다.

```
ubuntu@subvnic:~/CSED311/Lab04-2/Lab4-2$ ./convert
Usage: ./convert [TestName or TestNumber]
TestNumber:
  0) basic
  1) ifelse
  2) loop
  3) non-controlflow
  4) recursive

ubuntu@subvnic:~/CSED311/Lab04-2/Lab4-2$ ./check
Test 0
TOTAL CYCLE : 35 (Answer : 36)
Correct output : 32/32
Test 1
TOTAL CYCLE : 43 (Answer : 44)
Correct output : 32/32
Test 2
TOTAL CYCLE : 326 (Answer : 323)
Correct output : 32/32
Test 3
TOTAL CYCLE : 46 (Answer : 46)
Correct output : 32/32
Test 4
TOTAL CYCLE : 1203 (Answer : 1188)
Correct output : 32/32
```

./convert는 정규표현식을 이용해 tb_top.cpp, InstMomory.v의 테스트벤치 파일 명을 수정하는 프로그램이며, ./check는 ./convert를 이용해 5개의 테스트 벤치를 빠르게 검사할 수 있는 자동화 도구이다.

6. References

[1] Lecture notes.