Lab4-2 Report

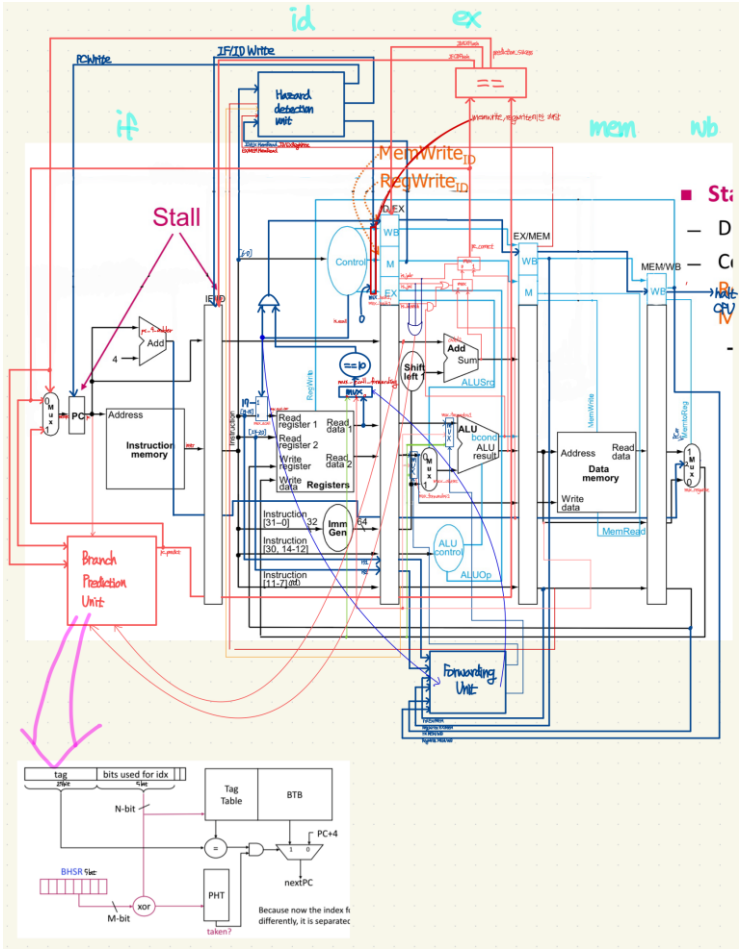
20230642 이채영, 20230683 박한비

1. Introduction

Lab4-2에서는 RISC-V instruction을 기반으로 하는 5-stage Pipelined CPU를 구현하였다. Pipelined CPU는 Multi-cycle CPU와 같이 하나의 instruction을 여러 사이클에 걸쳐 수행한다. 하지만 각 stage가 각 사이클마다 다른 instruction을 수행할 수 있고, 사이클 길이가 Single-cycle CPU보다 훨씬 짧으므로 Single-cycle CPU와 Multi-cycle CPU보다 훨씬 나은 성능을 보인다.

2. Design

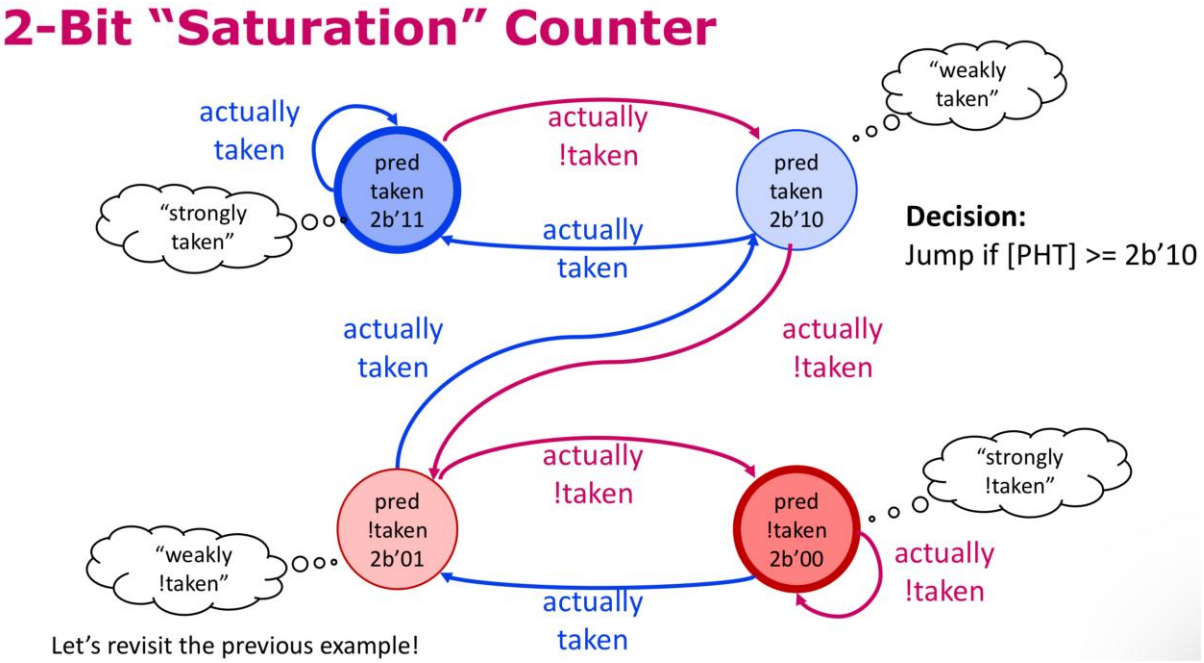
이번 Lab4-2에서는 이전 Lab4-1의 controlflow를 고려하지 않은 pipelined CPU에 이어, controlflow를 고려한 pipelined CPU를 구현하였다. 기본적으로 Lab4-1 report에서 제시한 설계도 바탕에 controlflow 관련 (control hazard 및 branch prediction) 회로를 추가하여 아래와 같이 설계하였다.



다음 pc를 예측하는 Branch Prediction Unit을 추가하여 예측하는 pc의 instruction을 수행하도록 하였고 해당 정확한 pc와 예측 pc를 비교해 예측 pc가 틀렸을 경우 flush를 수행 및 Branch Prediction 성공 여부를 전달하는 신호들을 추가하였다. 또한 이러한 flush로 인해 생기는 bubble 수를 두개로 줄이기 위해 정확한 pc 생성과 예측 pc와의 비교를 ex stage로 옮겼다.

Branch Prediction

Branch Prediction Unit은 기본적으로 Lec10-pipelined-CPU–branchpredict 교안의 21쪽. Gshare 구조를 바탕으로 구현하였고 이는 위의 전체 설계도에 포함되었다.
branch/jal/jalr일 경우 Branch Prediction Unit의 PHT (Pattern History Table)에서 jump를 taken할 지의 여부를 결정하게 되어있는데 이 여부는 2-bit 형태로 예측하며, Lec10-pipelined-CPU–branchpredict 교안의 15쪽 2-Bit Saturation Counter의 논리를 그대로 차용하였다.



3. Implementation

Lab4-2에서는 Lab4-1의 Pipelined CPU에서 BranchPredictionUnit 모듈을 추가하고, cpu 부분을 수정하였다. ALU, Register file 등 다른 모듈들은 Lab4-1과 동일하다.

1) cpu.v

Pipelined CPU를 구현한 메인 모듈로, 2.Design에서 제시한 설계도를 바탕으로 내부에 여러 모듈들은 include하고 wire로 적절히 연결하여 CPU를 구현하였다. Lab4-1과 비교하여 달라진 부분은, Branch가 taken되었는지 판단하는 부분을 EX stage로 옮기고 IF 부분에 BranchPrediction Unit 모듈을 추가하였다. 또한 Lab4-1에서는 교안에서 제시되었던 대로 immediate_generator에서 생성된 immediate 값을 왼쪽으로 1만큼 shift 한 후 pc+imm 계산에 활용했는데, 현재 구현한 CPU에서는 shift할 경우 잘못된 pc 값이 도출되어 shift 연산을 없앴다.

2) BranchPredictionUnit.v

Gshare 구조를 바탕으로 Branch Prediction을 구현하였다.

```
reg [24:0] tag_table [31:0];
reg [31:0] btb [31:0];
reg [1:0] pht [31:0];
reg [4:0] bhsr;

wire [31:0] current_pc_add_4_inBranchPrediction;
wire [4:0] btb_idx;
wire [24:0] tag;
wire [4:0] idx_xor_bhsr;

add pc_4_adder(
    .A(current_pc),
    .B(32'b100),
    .C(current_pc_add_4_inBranchPrediction)
);
```

먼저 tag_table, btb, pht, bhsr을 위와 같이 정의하였다. Tag는 PC의 상위 25비트, index는 tag 이후의 5비트를 사용하였다.

```
assign btb_idx = current_pc[6:2];
assign tag = current_pc[31:7];
assign idx_xor_bhsr = bhsr ^ btb_idx;

always@(*) begin
    if((tag_table[btb_idx] == tag) && (btb[btb_idx] != 0) && (pht[idx_xor_bhsr] == 2'b10 || pht[idx_xor_bhsr] == 2'b11)) begin
        pc_predict = btb[btb_idx];
    end
    else begin
        pc_predict = current_pc_add_4_inBranchPrediction;
    end
end
```

Index와 bhsr을 xor연산한 후 pht에 일치하는 결과가 있는지 확인하고, tag table에서 pc의 tag와 일치하는 것이 있는지를 확인한다. 두 가지가 모두 확인되면 btb에 저장되어 있던 pc(branch taken)를 다음 pc로 예측하고, 아니면 pc+4(branch not taken)을 다음 pc로 예측한다.

```
// btb, tag update
if ((prediction_success == 0) && (pc_correct != ID_EX_pc + 4)) begin
    btb[ID_EX_pc[6:2]] <= pc_correct;
    tag_table[ID_EX_pc[6:2]] <= ID_EX_pc[31:7];
end

// pht, bhsr update
if (prediction_taken) begin
    integer pht_entry = {27'b0, bhsr} ^ {27'b0, ID_EX_pc[6:2]};
    if ((pc_correct == ID_EX_pc + 4)) begin // not taken
        case (pht[pht_entry])
            2'b00 : pht[pht_entry] <= 2'b00;
            2'b01 : pht[pht_entry] <= 2'b00;
            2'b10 : pht[pht_entry] <= 2'b01;
            2'b11 : pht[pht_entry] <= 2'b10;
            default: pht[pht_entry] <= 2'b00;
        endcase

        bhsr <= {bhsr[3:0], 1'b0};
    end
    else begin // taken
        case (pht[pht_entry])
            2'b00 : pht[pht_entry] <= 2'b01;
            2'b01 : pht[pht_entry] <= 2'b10;
            2'b10 : pht[pht_entry] <= 2'b11;
            2'b11 : pht[pht_entry] <= 2'b11;
            default: pht[pht_entry] <= 2'b00;
        endcase

        bhsr <= {bhsr[3:0], 1'b1};
    end
```

위의 코드는 prediction 결과가 틀렸을 때 btb, tag, pht, bhsr을 업데이트하는 부분이다. Not taken으로 예측했으나 틀린 경우 btb의 해당 index에는 branch taken으로 이동한 pc 값이 저장된다. pht는 매 prediction마다 2-bit saturation counter에 따라 업데이트한다. bhsr은 왼쪽으로 한 칸씩 shift하고, LSB에 새로운 taken 값을 넣는다.

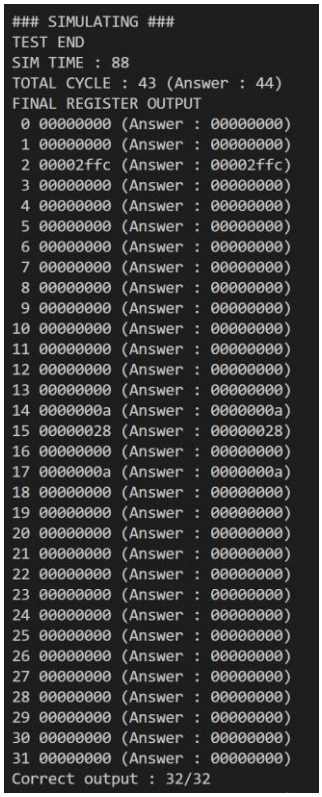
4. Discussion

1) Branch Prediction Unit의 btb 업데이트 문제

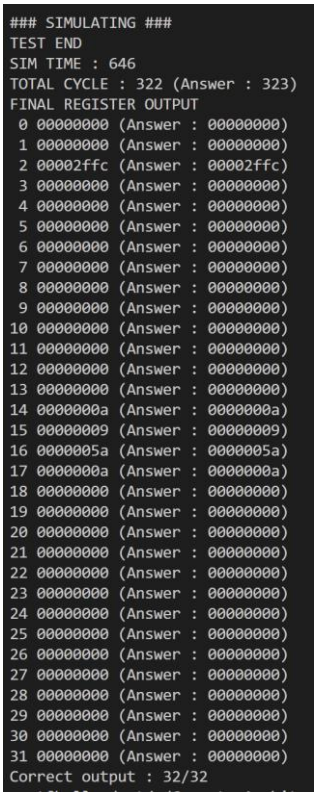
처음에 EX stage에 branch predicition을 always not taken으로 구현한 후, always taken으로 수정하기 위해 btb를 추가하였다. 그러나 실제 pc값에 따라 btb를 업데이트하는 과정에서 예측 pc/실제 pc 비교가 제대로 이루어지지 않는 어려움이 있었다. 따라서 BranchPredictionUnit을 IF stage로 옮기고, Prediction을 하는 부분(IF stage)과 Taken/Not taken에 따른 table 업데이트를 하는 부분(EX stage)을 명확하게 구분하였다. 즉, Branch prediction은 모두 IF stage에서 이뤄지도록 하였다. 그리고 EX stage에서 실제 pc 값을 가져와 prediction이 맞았는지 확인하고 BTB, PHT, Tag Table, BHSR이 업데이트되도록 하였다.

2) Always-not-taken / Always-taken predictor와 2-bit Gshare predictor cycle 비교

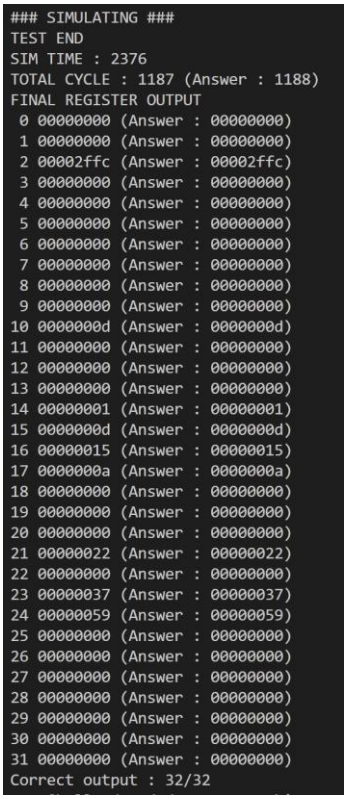
Always-not-taken predictor로 구현했을 때 controlflow instruction이 포함된 테스트 벤치의 cycle은 아래와 같다.



ifelse

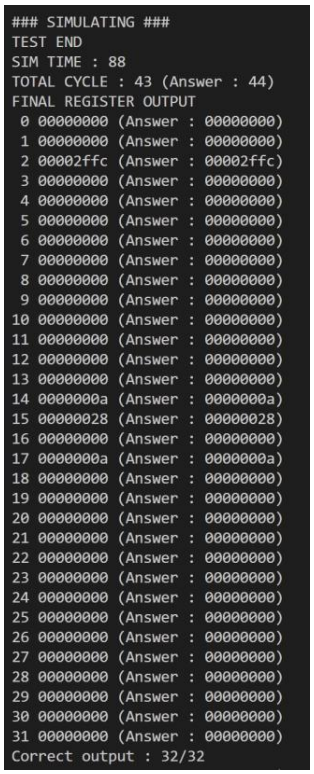


loop

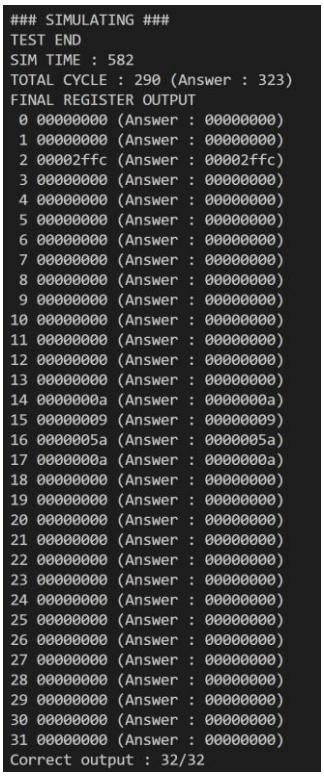


recursive

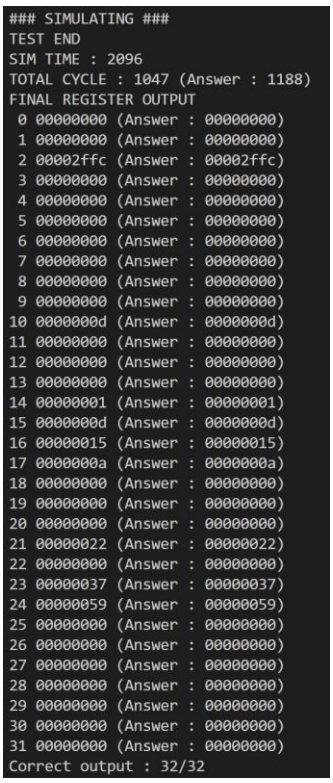
또한 always-taken predictor로 구현했을 때 결과는 아래와 같다.



ifelse



loop



recursive

5. Conclusion에서 사진을 첨부하겠지만, 2-bit Gshare Predictor로 구현한 결과는 ifelse(43/44), loop(302/323), recursive(1077/1188)이다. Always-not-taken Predictor보다는 사이클이 확연히 줄어들었지만 Always-taken Predictor가 가장 작은 사이클을 가지는 것을 볼 수 있다. 이는 loop, recursive testbench의 controlflow instruction이 대부분의 경우 taken 되기 때문에 2-bit Gshare보다 Always-taken 구조가 더 작은 사이클을 가진다. 프로그램이 커지고 taken, not-taken이 번갈아 나오는 테스트벤치에 대해서는 2-bit Gshare Predictor가 가장 정확한 Prediction을 수행할 것이다.

5. Conclusion

Lab4-2에서 작성한 코드로 주어진 testbench를 실행한 결과는 아래와 같다. 모든 testbench에 대하여 레지스터 값들이 적절하게 할당된 것을 볼 수 있으며, loop, recursive 테스트벤치의 경우 Gshare를 통한 branch prediction의 결과로 정답 cycle보다 더 적은 cycle이 소모된 것을 볼 수 있다.

```
### SIMULATING ###
TEST END
SIM TIME : 72
TOTAL CYCLE : 35 (Answer : 36)
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 00000013 (Answer : 00000013)
11 00000003 (Answer : 00000003)
12 fffffffd7 (Answer : fffffffd7)
13 00000037 (Answer : 00000037)
14 00000013 (Answer : 00000013)
15 00000026 (Answer : 00000026)
16 0000001e (Answer : 0000001e)
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
```

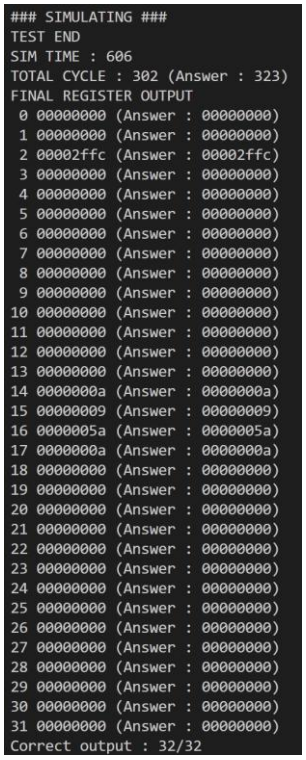
basic_mem

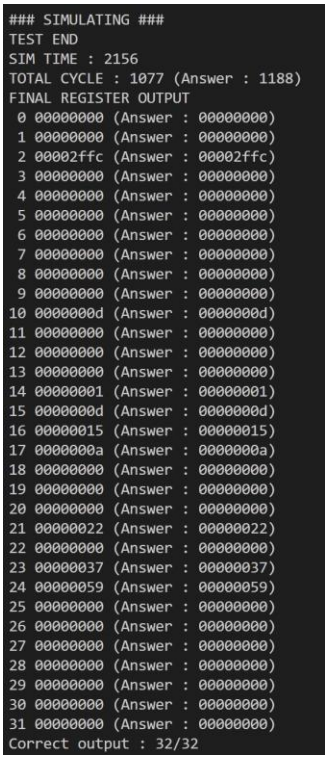
```
### SIMULATING ###
TEST END
SIM TIME : 88
TOTAL CYCLE : 43 (Answer : 44)
FINAL REGISTER OUTPUT
 0 00000000 (Answer : 00000000)
 1 00000000 (Answer : 00000000)
 2 00002ffc (Answer : 00002ffc)
 3 00000000 (Answer : 00000000)
 4 00000000 (Answer : 00000000)
 5 00000000 (Answer : 00000000)
 6 00000000 (Answer : 00000000)
 7 00000000 (Answer : 00000000)
 8 00000000 (Answer : 00000000)
 9 00000000 (Answer : 00000000)
10 00000000 (Answer : 00000000)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 00000000 (Answer : 00000000)
14 0000000a (Answer : 0000000a)
15 00000028 (Answer : 00000028)
16 00000000 (Answer : 00000000)
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
```

ifelse_mem

```
### SIMULATING ###
TEST END
SIM TIME : 94
TOTAL CYCLE : 46 (Answer : 46)
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
```

non-controlflow_mem





loop_mem recursive_mem

Lab4-2를 통해 Branch Prediction의 원리를 이해하고 Pipelined CPU에 적용해보며, 각 instruction별로 5개의 stage에서 어떤 일이 일어나는지 습득할 수 있었다.