1. Introduction

이번 실험의 목표는 지난 lab 4-1에서 구현한 non-control-flow pipelined CPU를 control-flow Instruction도 실행할 수 있도록 개선하는 것이다.

A. Why should we predict next PC?

	R/I-Type	LW	SW	Bxx	JAL	JALR
IF	use	use	use	use	use	use
ID	produce	produce	produce			
EX						
MEM				produce	produce	produce
WB						

control-flow Instruction을 고려하기 시작하면 더 이상 다음 PC가 기존 PC+4라고 보장할 수 없기에, ID stage까지 가서 다음 PC값이 결정되어야 그제서야 다음 Instruction을 가져올 수 있다. 심지어 전 Instruction 이 만약 Jump나 Branch Instruction이라면 MEM단계까지 가서야 다음 PC값이 결정되기에, 1~3사이클씩 stall이 매번 걸리게 되고 CPU성능이 심각하게 저하가 된다.

이를 해결하기 위해 모종의 방법으로 다음 PC값이 '실제로 결정' 되기 전에 미리 '예측' 할 필요가 있다.

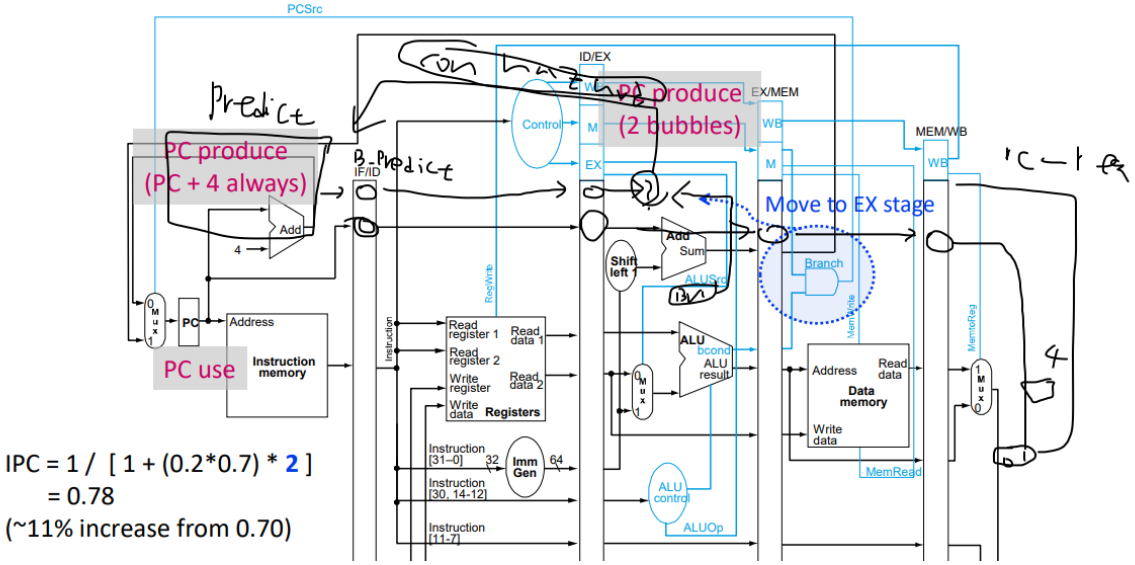
가장 기초적인 PC예측 방법으로는, control-flow Instruction보다 일반 none-control-flow Instruction이 훨씬 더 많다는 것을 이용하여, 언제나 다음 PC값이 PC+4라고 예측했다가, 나중 단계에서 control-flow Instruction이었음이 밝혀지면, 진행중이던 stage를 전부 flush시키고 맞는 PC값으로 틀린 부분부터 다시 시작하는 always-not-taken 방식이 있으며,

더 정확하고(동시에 복잡한) PC예측 방법으로 control-flow Instruction이 오면 그 타겟 주소를 Branch Target Buffer에 저장하고, Branch 방향을 적극적으로 예측하여 다음 PC를 예측하는 방법 등이 있으며 여러 메커니즘이 존재한다.

우리 조는 Global BHSR와 2bit predictor가 존재하는 BTB를 결합한 gshare predictor를 선택했으며, 비교용으로 Global BHSR는 없는 단순 2bit predictor with BTB 구조의 predictor도 설계하였다.

2. Design

A. How to handle branch prediction?



IPC = 1 / [1 + (0.2*0.7) * **2**]
= 0.78
(~11% increase from 0.70)

lab4-1에서 단순 PC+4adder로 해결한 next PC Logic을 Branch Predictor로 교체하여 다음 PC값을 예측하도록 해야 하며, 이 예측 값이 실제 PC값과 같은 지 확인하고,

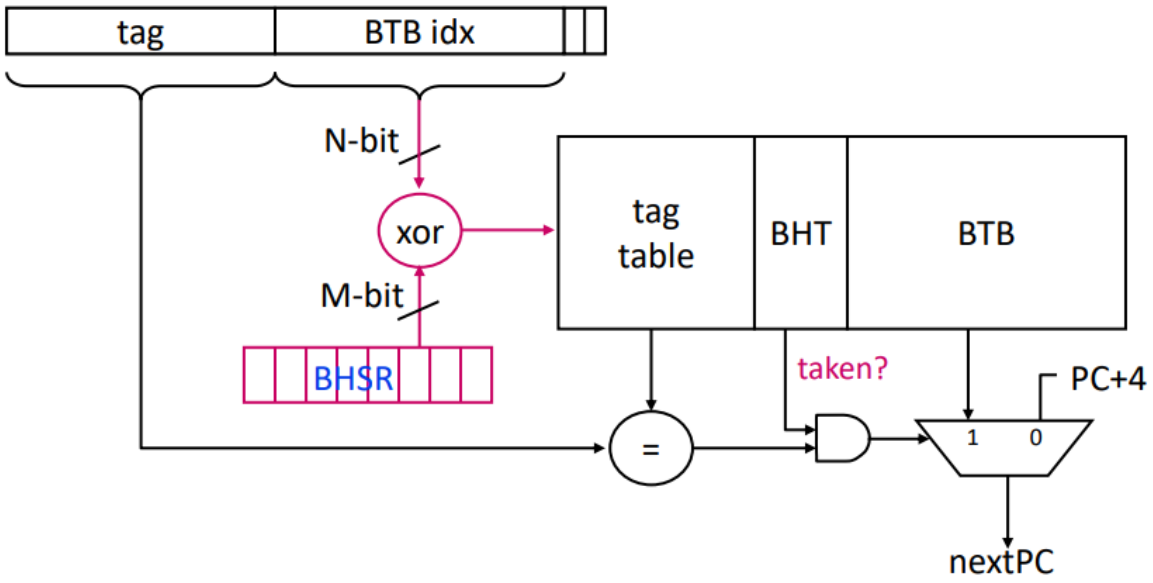
(Branch decide logic이 위의 그림처럼 lab guideline을 따라 MEM에서 EX stage로 옮겨갔으므로, 이 확인은 EX stage에서 가능하다)

만약 앞서 예측한 PC가 틀렸다면, 틀린 PC를 기반으로 실행 중이던 ID, EX stage를 kill하고 IF stage로 맞는 PC값을 보내서 틀린 지점에서 자시 시작해야 한다.

이를 위해 IF stage에 branch predictor가, EX stage에 틀린 PC여부를 판별하는 control hazard detection unit이 추가되어야 하며, control hazard detection unit이 사용할 수 있도록 Branch Predictor가 Predict한 branch 방향과 PC값 등을 파이프라인 레지스터를 따라 EX stage까지 보내야 한다.

추가적으로 branch predictor내부의 BTB table을 수정할 정보를 EX stage로부터 IF stage로 끌어와야 하며, branch predictor가 이전 Instruction이 control-flow Instruction이었는지 확인하기 위해, ID stage로부터 바로 이전 Instruction의 opcode도 가져와서 확인한다.

B. Describe your design of branch predictor.



기본적으로 강의자료의 "Gshare" Branch Prediction 구조를 따라서 만들었다.

```
reg [59:0] Branch_table [31:0]; // valid 1bit tag 25bit BHT 2bit Branch target 32bit.
reg [4:0] Branch_histroy_shift_Reg;
```

구조:

32개의 entry로 구성된 BTB테이블이 Branch Target을 저장하고 있다. 각 entry는 Valid bit 1비트, tag 25비트, BHT 2비트, BTB32비트로 구성되어 총 60비트이다.

또한, 5비트짜리 BHSR(Branch History Shift Register)가 최근 5번 Branch의 결과를 기록한다. 새로운 Branch Instruction이 실행될 때 마다 그 결과를 Shift해 가며 기록한다. 기존 값을 전부 왼쪽으로 1비트 Shift하고, 하위 비트에 최근 Branch Instruction이 taken이면 1, not taken이면 0을 채워 넣는다.

또한, Indexing 할 때는, PC의 하위 7비트 중 하위 2 비트를 제외하고 (RISC-V PC는 언제나 4의 배수이므로) 상위 5비트를 BHSR와 xor연산해서 Index로 사용하게 된다. (이때 그냥 이 5비트를 그대로 Index로 쓴다면 2bit global BTB predictor가 된다.)

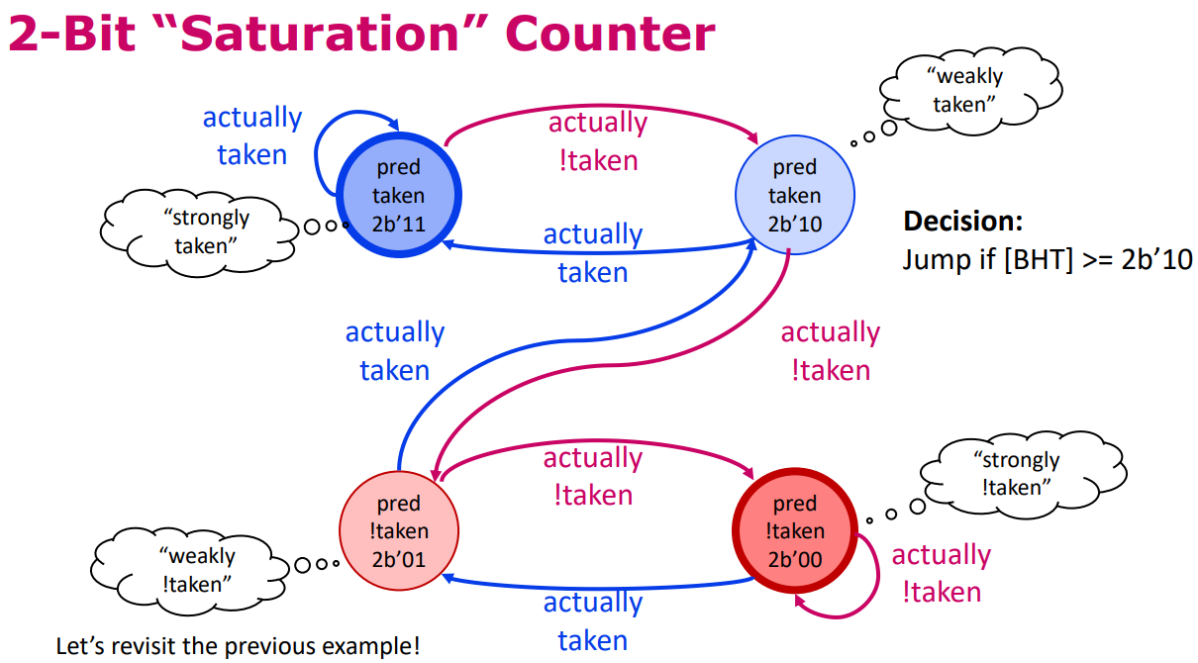
작동 방식:

1.Branch Table Update

branch Instruction이 EX stage까지 도달하여 결과가 계산되었다면, (jal instruction은 always taken Branch로, branch의 일종처럼 취급한다)

Branch Predictor는 EX stage로부터 해당 Branch Instruction의 PC값, target address, taken 여부 등을 받아온다. 이 받아온 PC값을 바탕으로 앞서 설명한 것처럼 BHSR와 xor연산을 하여 Indexing

하고, 해당 table entry에 hit 한다면(tag가 일치하며 valid bit 이 1) BHT 2비트를 2-Bit "Saturation" Counter logic에 따라 업데이트를 하며, 해당 table entry가 miss라면, 받아온 정보의 Branch Instruction 정보로 덮어쓰며, 기존 정보는 evict된다.



위에서 말한 2-Bit "Saturation" Counter logic을 그림으로 나타낸 것이다.

```
if(Update_Table_hit) begin
    if(ID_EX_actual_branch_taken) begin
        case(Branch_table[Update_index][33:32])
        2'b11: Branch_table[Update_index][33:32] <= 2'b11;
        2'b10: Branch_table[Update_index][33:32] <= 2'b11;
        2'b01: Branch_table[Update_index][33:32] <= 2'b10;
        2'b00: Branch_table[Update_index][33:32] <= 2'b01;
        endcase
    end
    else begin
        case(Branch_table[Update_index][33:32])
        2'b11: Branch_table[Update_index][33:32] <= 2'b10;
        2'b10: Branch_table[Update_index][33:32] <= 2'b01;
        2'b01: Branch_table[Update_index][33:32] <= 2'b00;
        2'b00: Branch_table[Update_index][33:32] <= 2'b00;
        endcase
    end
end
```

위 그림의 FSM logic을 코드로 나타내면 좌측과 같이 된다.

이 모든 "Branch Table Update"는 다른 모든 레지스터들과 마찬가지로 Clock synchronous 하게 Update 된다.

2. Branch Predicton

분기 예측은 1단계에서 Branch Table에 지금까지 Update 한 정보들을 바탕으로 이루어진다.

이전 단계 Instruction의 opcode가 Branch/jal instruction이었으며, 현재 PC값을 바탕으로 Indexing 하였을 때 해당 table entry에 hit하고, 그 table entry의 2bit BHT counter logic이 "branch taken"을 가리키고 있다는 3가지 조건을 모두 동시에 만족할 때 에만 BTB로부터 Branch/jump target 주소를 가져와 다음 PC예측값으로 내보낸다. 위 조건이 하나라도 만족되지 않으면 다음 PC값은 PC+4로 예측하게 된다.

3. Prediction Error Correction

그러나 Branch Predictor가 틀린 결과를 예측했다면, 이를 바로잡지 않으면 잘못된 코드가 실행되어 버린다.

따라서 Branch Predictor가 예측한 분기 방향 및 Next PC를 파이프라인 레지스터를 따라 EX stage까지 전달하고, Branch Predictor의 추측이 정말로 맞았는지 확인해서, 틀렸다면 잘못된 예측을 기반으로 실행 중이던 ID, EX stage를 kill하고, IF stage의 PC값에 올바른 Next PC값을 전달하여 틀린 부분부터 다시 시작하는 Logic이 필요하다.

Branch Predictor가 틀린 경우는 크게 두 가지이다.

1. Jump Direction Predict Error – jal/taken branch인데, table miss 혹은 BHT등의 이유로 not taken으로 예측하여 PC+4로 예측한 경우, 또는 not taken branch인데 BHT등의 이유로 Branch target으로 PC를 예측한 경우.
2. Hash Collision Error – 실제로 taken branch였고, Prediction도 Branch taken으로 맞게 예측하였으나, Hash Collision으로 인해, 다른 PC값인데도 table hit가 발생해서 잘못된 Branch target으로 예측하였을 경우.

2 번은 "Gshare" Branch Prediction 구조 에서만 발생하며, 더 단순한 2bit global BTB predictor에서는 오히려 발생하지 않는다. 그 이유는, 2bit global BTB predictor는 PC하위 비트들을 그대로 index로 사용하기에, PC값이 다르다면 절대로 hit가 일어날 수 없으나, "Gshare" Branch Predictior는 BHSR와 xor하는 Hash function이 다른 PC값도 당시 BHSR값에 따라 같은 Index를 도출하는 경우가 있고, 이 경우 만약 tag까지 같다면 발생해서는 안 되는 Branch table hit가 일어나서 잘못된 값이 예측되는 것이다.

따라서 EX stage의 Control Hazard Detection Unit는 1,2번의 조건을 검사하고, kill신호와 hash collision신호를 내보낸다.

kill 신호는 IF/ID 파이프라인 레지스터와 ID/EX 파이프라인 레지스터를 전부 0으로 만들어 ID와 EX stage에 들어있는 잘못된 instruction들을 flush한다.

또한 Next PC Logic은 kill신호와 hash collision신호가 없다면 Branch Predictor의 예측 PC를 Next PC로 사용하지만, kill 신호가 들어온다면 PC+4 혹은 PC+Imm같은 올바른 PC값으로 PC값을 갈아끼우게 된다. hash collision 신호가 들어온다면 Branch taken 예측까지는 맞으나 Branch target이 잘못된 것 이므로, PC+Imm하여 올바른 branch target값으로 교체한다.

3.Implementation

기본적인 Pipelined CPU 구조는 Lab 4-1을 그대로 따르므로, 이번 lab에서 추가된 부분만 설명하겠다.

1. Branch Predictor

```
module BranchPredictorGshare(
    input reset,
    input clk,
    input [31:0] current_pc,
    input [31:0] ID_EX_pc,
    input [31:0] ID_EX_branch_target,
    input ID_EX_is_jal,
    input ID_EX_is_branch,
    input ID_EX_actual_branch_taken,
    input [6:0] opcode,
    output [31:0] btb_next_pc,
    output predict_branch_taken);
```

Branch Predictor의 Input과 output이다.

```
reg [59:0] Branch_table [31:0]; // valid 1bit tag 25bit BHT 2bit Branch target 32bit.
reg [4:0] Branch_histroy_shift_Reg;
```

앞서 말한대로 60bitx32bit Branch table과 5bit BHSR를 사용한다.

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

cpu reset신호시 모든 값을 0으로 초기화한다.

```
assign index = current_pc[6:2] ^ Branch_histroy_shift_Reg;
assign tag = current_pc[31:7];
assign Table_entry = Branch_table[index];

assign Update_index = ID_EX_pc[6:2] ^ Branch_histroy_shift_Reg;
assign Update_tag = ID_EX_pc[31:7];
```

indexing은 PC하위 비트들과 BSHR를 xor연산 하여 이루어진다.

```
if(ID_EX_is_branch || ID_EX_is_jal) begin
    if(Update_Table_hit) begin
        if(ID_EX_actual_branch_taken) begin
            case(Branch_table[Update_index][33:32])
            2'b11: Branch_table[Update_index][33:32] <= 2'b11;
            2'b10: Branch_table[Update_index][33:32] <= 2'b11;
            2'b01: Branch_table[Update_index][33:32] <= 2'b10;
            2'b00: Branch_table[Update_index][33:32] <= 2'b01;
            endcase
        end
        else begin
            case(Branch_table[Update_index][33:32])
            2'b11: Branch_table[Update_index][33:32] <= 2'b10;
            2'b10: Branch_table[Update_index][33:32] <= 2'b01;
            2'b01: Branch_table[Update_index][33:32] <= 2'b00;
            2'b00: Branch_table[Update_index][33:32] <= 2'b00;
            endcase
        end
    end
    else begin
            Branch_table[Update_index][59] <= 1'b1; //valid 1
            Branch_table[Update_index][58:34] <= ID_EX_pc[31:7]; //set tag
            if(ID_EX_actual_branch_taken) begin
                Branch_table[Update_index][33:32] <= 2'b11;
            end // because of jal 11.
            else begin
                Branch_table[Update_index][33:32] <= 2'b00;
            end
            Branch_table[Update_index][31:0] <= ID_EX_branch_target; //BTB
    end
```

Branch Table업데이트 Logic. 만약 방금 결과가 결정된 Branch Instruction과 일치하는 엔트리가 테이블에 존재하면 2bit saturation counter를 알맞게 변화시키고, 그런 엔트리가 존재하지 않는다면, 해당하는 index의 기존 정보를 evict 하고 정보를 덮어쓴다.

```
if(ID_EX_is_jal) begin
    // Branch_histroy_shift_Reg <= Branch_histroy_shift_Reg<<1 + 1;
    Branch_histroy_shift_Reg <= {Branch_histroy_shift_Reg[3:0], 1'b1};
end
else begin
    if(ID_EX_actual_branch_taken) begin
    // Branch_histroy_shift_Reg <= Branch_histroy_shift_Reg<<1 + 1;
    Branch_histroy_shift_Reg <= {Branch_histroy_shift_Reg[3:0], 1'b1};
    end
    else begin
    // Branch_histroy_shift_Reg <= Branch_histroy_shift_Reg<<1;
    Branch_histroy_shift_Reg <= {Branch_histroy_shift_Reg[3:0], 1'b0};
    end
end
```

```
assign Table_hit = (Table_entry[59] && (Table_entry[58:34] == tag))? 1 : 0;
assign predict_branch_taken = (Table_entry[33] && Table_hit) && (opcode == `BRANCH || opcode == `JAL);
assign btb_next_pc = (predict_branch_taken) ? (Table_entry[31:0]) : (current_pc + 4);
```

BHSR업데이트 로직, 1비트씩 shift하며 최신 Branch 결과 패턴을 기록한다.

PC예측 Logic. BTB table hit && 해당 엔트리의 BHT 2bit saturation counter 가 'taken'을 예측 중이면 BTB에 기록된 Branch target을 가져와서 다음 PC 예측으로 사용한다. 아닌 경우에는 기본적으로 PC+4로 예측한다.

2. Control Hazard Detection Unit

```
module ControlHazardDetectionUnit(
    input ID_EX_is_jal,
    input ID_EX_is_jalr,
    input ID_EX_is_branch,
    input ID_EX_predict_branch_taken,// 1:0 taken/not taken
    input ID_EX_actual_branch_taken,// 1:0 taken/not taken
    input [31:0] ID_EX_branch_target,
    input [31:0] ID_EX_branch_predict_target,
    output kill, //stop IF,ID
    output hash_colision
);
    wire jump_error;

    assign jump_error = ((ID_EX_is_jal && !ID_EX_predict_branch_taken) || ID_EX_is_jalr) ? 1
    assign hash_colision = ID_EX_actual_branch_taken&&ID_EX_predict_branch_taken&&(ID_EX_bran
    assign kill = jump_error||hash_colision;



: ((ID_EX_is_branch && (ID_EX_predict_branch_taken != ID_EX_actual_branch_taken)) ? 1 : 0);
ch_target != ID_EX_branch_predict_target);
```

EX stage에 존재하며, 현재 EX에서 실행중인 instruction이 jalr이거나, jal인데 not taken으로 예측을 했거나, branch instruction인데 분기 예측이 틀렸다 거나 하는 상황을 감지하여, 틀린 예측을 기반으로 실행 중이던 stage들을 kill하는 신호를 내보내는 Unit이다.

이런 판단을 내릴 수 있도록, Branch Predictor가 예측한 분기 방향과 점프 주소 등을 EX stage에서 실제 결과와 비교할 수 있도록 Pipeline 레지스터를 따라 EX stage 까지 보낼 필요가 있다.

```
always @(posedge clk) begin
  if (reset || kill) begin
      IF_ID_IR <= 0;
      IF_ID_pc <= 0;
      IF_ID_predict_branch_taken <= 0;
      IF_ID_branch_predict_target <= 0;
  end
  else if (!IF_ID_stall) begin
      IF_ID_IR <= imem_to_IR;
      IF_ID_pc <= curr_pc;
      IF_ID_predict_branch_taken <= predict_branch_taken;
      IF_ID_branch_predict_target <= btb_next_pc;
  end

end
```

위의 kill신호를 받으면, 왼쪽과 같이 잘못된 stage들의 pipeline레지스터가 전부 0으로 변하며 잘못된 instruction이 사라진다.

3. Next PC Logic

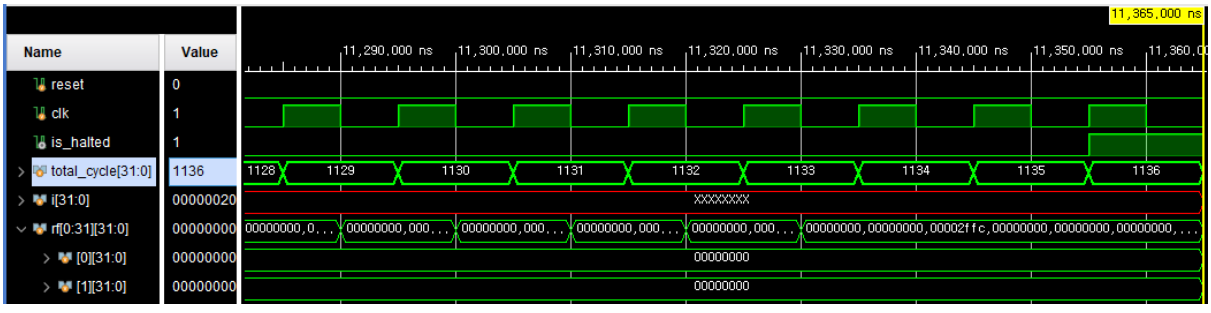
```
always @(*) begin
if(hash_colision) next_pc = ID_EX_pc + ID_EX_imm;
else begin
        if (kill) begin
            if (ID_EX_is_jalr) begin
                next_pc = jalr_next_pc;
            end
            else begin
                    if (ID_EX_predict_branch_taken) begin
                        next_pc = ID_EX_pc + 4;
                    end
                    else begin
                        next_pc = ID_EX_pc + ID_EX_imm;
                    end
                end

    end
    else begin
        next_pc = btb_next_pc;
    end
end
```

Next PC Logic은 기본적으로 특별한 일이 없으면 Branch Predictor가 도출한 예측 값을 Next PC로 사용하나, kill신호나 hash collision같은 신호가 들어오면, 알맞은 PC로 Next PC를 교체하여, 틀린 부분부터 다시 시작하게 된다.

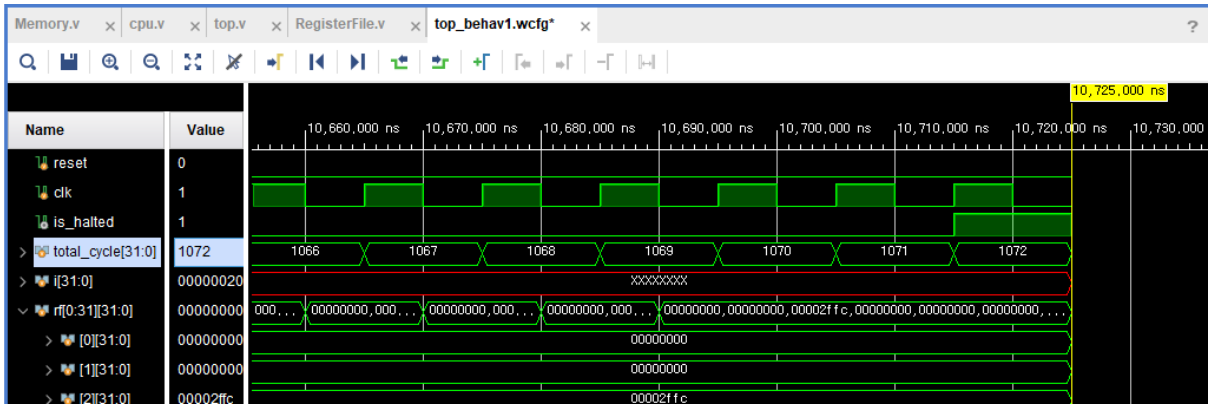
예를 들어, kill신호가 들어왔는데, EX단계의 Instruction이 not taken으로 예측되었다면, 실제로는 taken branch인데 not taken으로 잘못 예측 했었다는 뜻 이므로, PC+IMM = Branch target으로 Next PC를 수정하여 틀린 부분부터 다시 시작한다.

4. Discussion

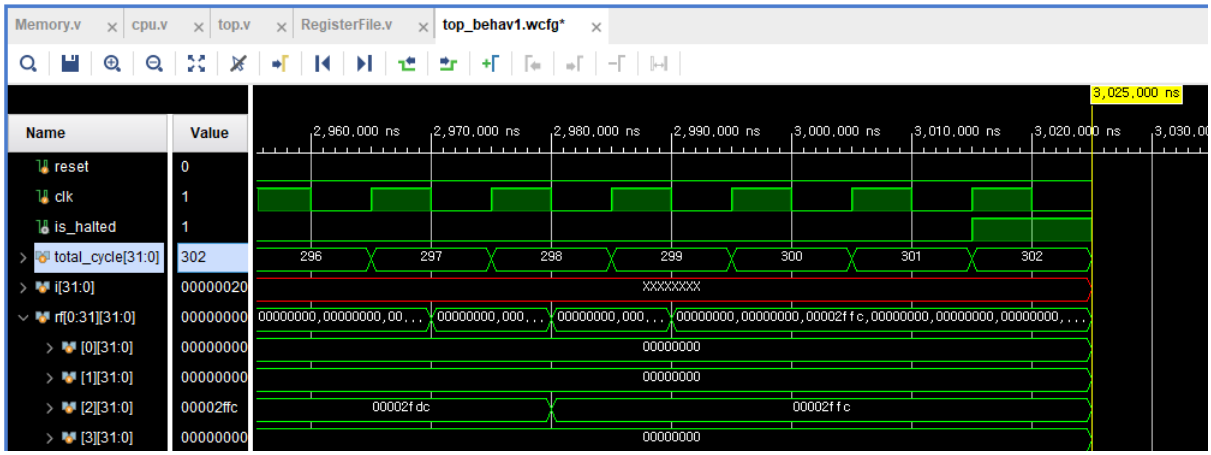
Compare total cycles: (Gshare를 구현하면 2bit global prediction와의 비교 하라고 공지)



recursive_mem gshare prediction: 1136사이클

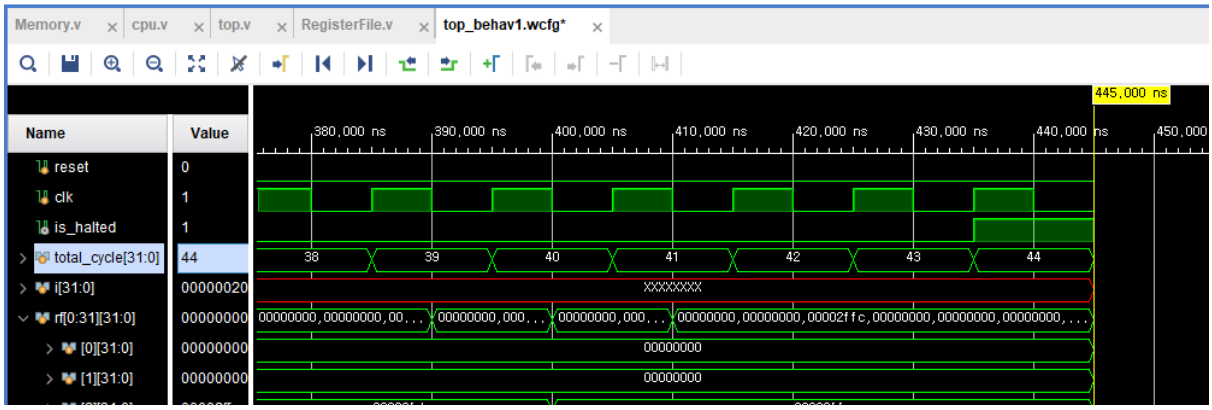


recursive_mem 2bit global prediction: 1072사이클

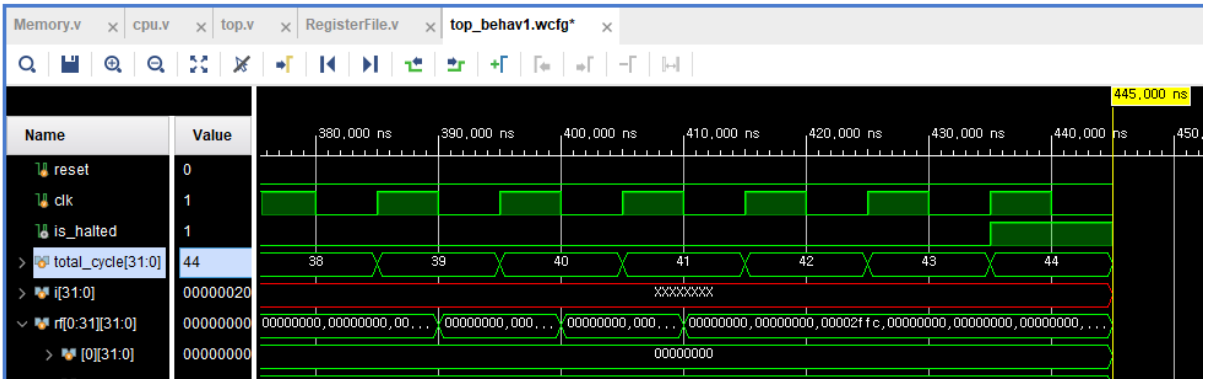


loop_mem gshare prediction: 302사이클

loop_mem 2bit global prediction: 290사이클



ifelse_mem gshare prediction: 44사이클



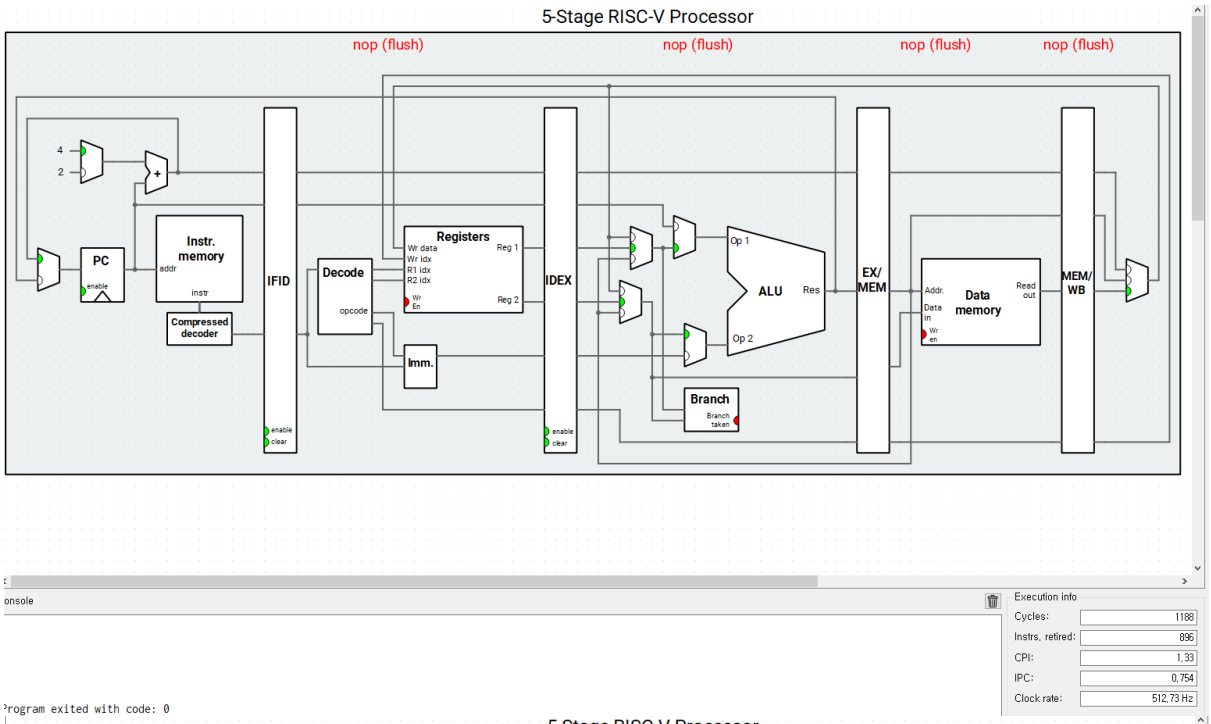
ifelse_mem 2bit global prediction: 44사이클

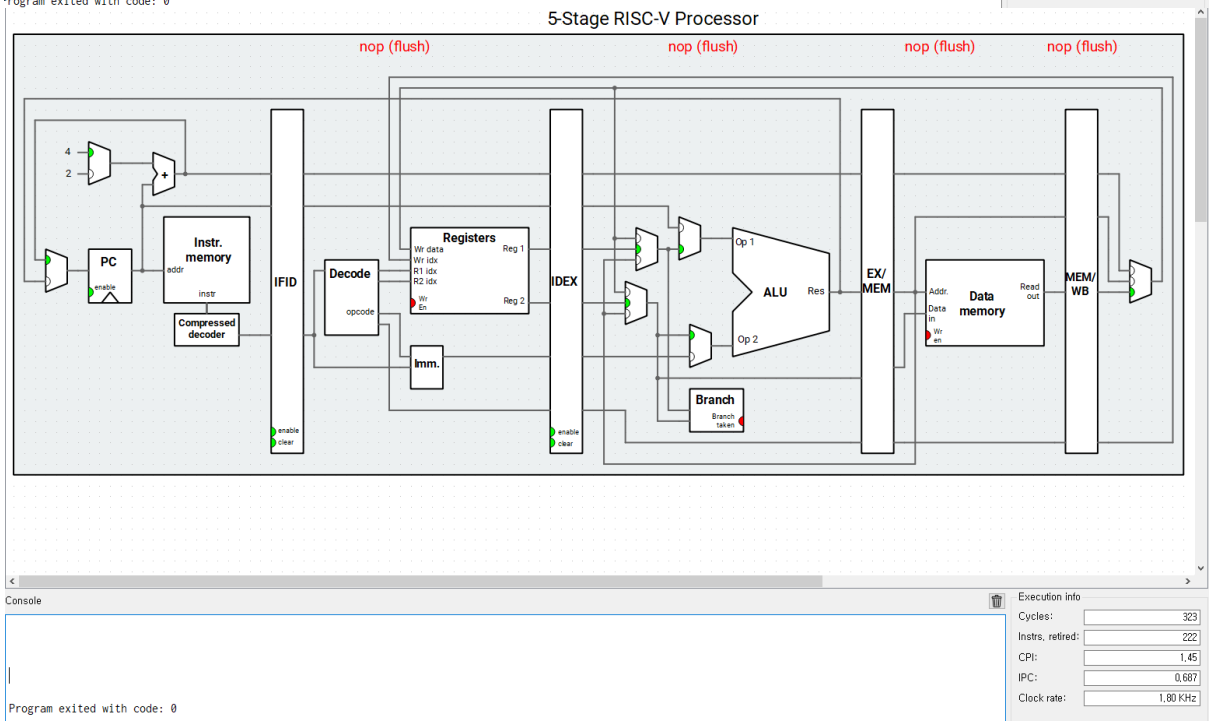
두 Prediction방식 모두 같은 Branch instruction이 계속 반복되는 loop과 recursive코드에서만 성능 차이가 나고 기타 코드는 성능이 동일하다. 이는 당연한데, Branch Instruction이 반복되지 않으면 모두 최초로 맞닥뜨린다는 소리이고, 이 경우 당연히 Branch Table에 해당 Branch에 대한 정보가 없을 것이기에 전부 다 동일하게 PC+4로 예측할 것이기 때문이다.

다만, 원래는 더 '고급' 방식인 gshare predictor가 더 기초적인 2bit global prediction방식에 비해 성능이 살짝 떨어진다. 그 이유는 이렇게 코드가 짧고 PC값도 작은 상태에서는 괜히 xor연산을 하면 hash collision이 절대 발생하지 않는 2bit global prediction의 단순 Indexing 방식에 비해, conflict가 더 자주 발생하여 자꾸 정보가 Evict되기 때문일 것으로 추정했다.

애초에 이런 짧은 코드로는 BTB학습도 채 되기 전에 끝나기에 성능 비교에 적절하지 않으며, 만약 프로그램이 더 크고 Branch Instruction이 더 자주, 많이 나온다면 Gshare가 더 성능이 높게 나올 수도 있을 것이다.

5. Conclusion





확신은 할 수 없으나, 회로 구조로부터 always-not-taken prediction을 사용한다고 추정되는(PC에 단순 Adder만 연결되어 있음) RIPES 시뮬레이터로 같은 코드를 돌려보면 recursive는 1188사이클, loop는 323사이클이 걸림을 알 수 있다.

같은 상황에서 우리 조가 설계한 Gshare Predictor는 1188->1136, 323->302사이클로 단축시키므로 성능 향상을 확인할 수 있다.