

2025 Spring

CSED311

컴퓨터구조

Lab 4-2 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

I. 서론

II. 디자인

- How to handle branch prediction?
- Design of branch predictor

III. 구현

IV. 논의 사항

V. 결론

- Compare total cycles of 2-bit global prediction with that of always-taken and always-not-taken
- Compare total cycles of always-taken with that of always-not-taken

I. 서론

이 과제의 목표는 Lab 4-1에서 구현했던 pipelined CPU를 data hazard 뿐만 아니라 control hazard도 해결할 수 있도록 변형하는 것이다. 더 좋은 퍼포먼스를 위해 Branch prediction 모듈도 Gshare 방식으로 구현하였다.

Control hazard는 기본적으로 pc에서 일어나는 data hazard이다. 하지만 이를 Lab4-1에서 구현한 것처럼 stall만을 이용하여 해결하면 한 instruction을 수행하는 데에 stall을 한 번씩은 해주어야 해서 거의 두 배로 느려진다. 따라서 대부분의 cpu들은 branch prediction이라는 방법을 사용해 control hazard를 data hazard와는 다른 방식으로 해소한다.

II. 디자인과 III. 구현에는 다음과 같은 내용을 중점적으로 서술하겠다. A.의 내용에서는 cpu.v의 내용물이, B.의 내용부터는 BranchPredictor.v 모듈 내부의 내용물이 수정된다.

- A. Control instructions과 control hazard을 위한 cpu.v의 수정
(BranchPredictor 모듈을 cpu.v에 배치하는 것까지 포함)
- B. BTB
- C. PHT
- D. BHSR 및 Gshare 로직

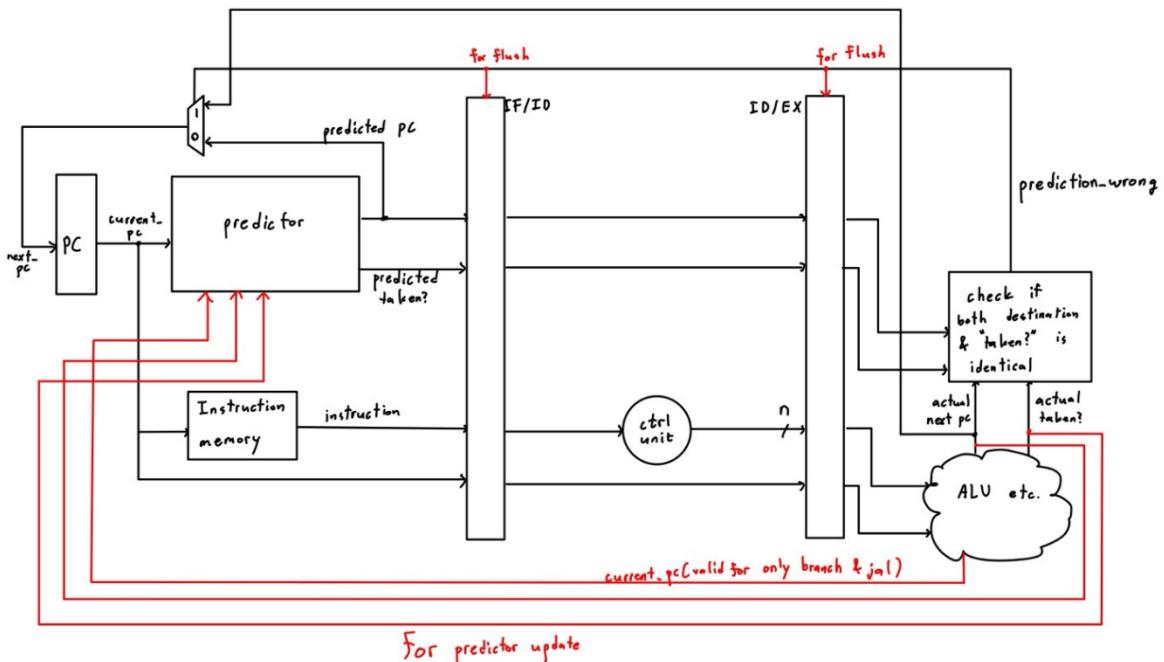
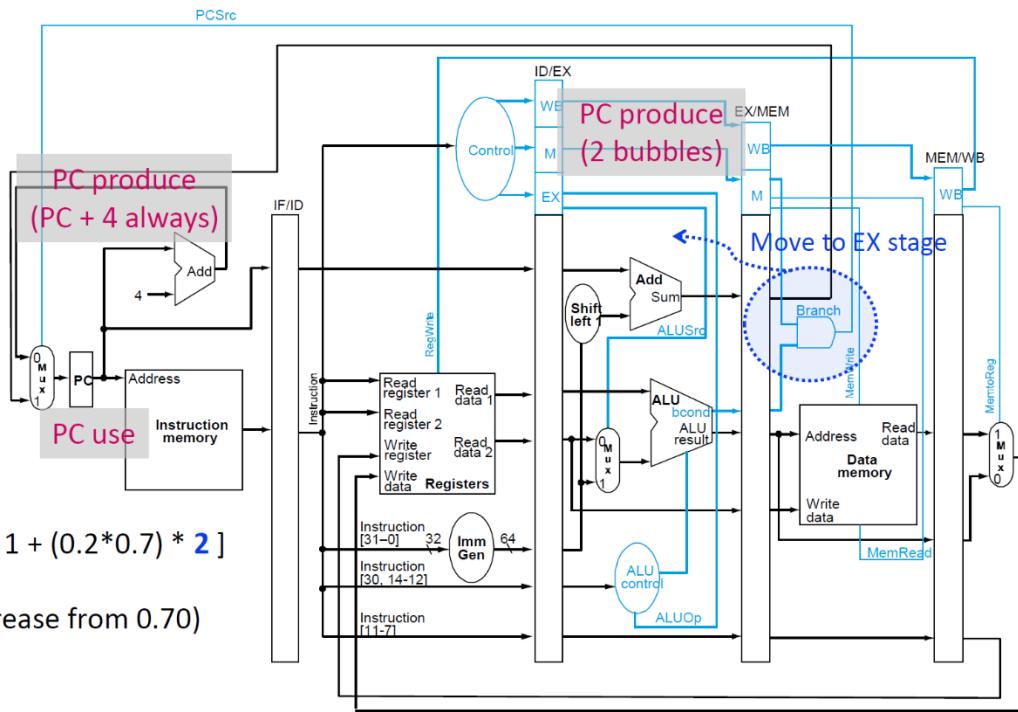
II. 디자인

A. cpu.v의 수정

기본적인 구조는 다음 장의 첫 번째 그림을 참고했다. 다만 첫 번째 그림의 구조도는 "PC + 4"로 always not taken을 예측하는 형태이기 때문에, 일반적인 branch predictor을 이식하기 위해서는 조금의 추가 작업이 필요하다. Branch predictor과 관련되어서 추가되어야 할 로직들을 구조도로 그려 보면 다음 장의 두 번째 그림과 같아진다.

다음 장의 두 번째 그림에서 predictor이 always not taken인 경우 언급해볼 만한 사항은 다음과 같다. Always not taken의 구조도에서 branch predictor을 이식했을 때의 달라지는 점을 강조하기 위해 이 사항들을 정리하였다.

- Branch predictor은 단순히 predicted pc를 current pc + 4로 output하는 모듈이다.
 - 어차피 pc+4로 예측하기 때문에 branch predictor에 업데이트해 주어야 하는 값이 존재하지 않는다. (아래의 빨간 wire 3개)
 - 또한 어차피 predicted taken은 0이기 때문에 이 또한 output할 필요가 없다.
- Prediction_wrong은 branch_taken과 동치로 생각할 수 있다. 이 정보는 current_pc만 사용하여 알아낼 수 있기 때문에, predicted pc나 predicted taken같은 시그널들을 파이프라인으로 전달할 필요가 없어진다.



Branch predictor의 예측이 잘못된 것이었을 경우 이를 해결하기 위해 IF와 ID단계의 명령어들을 nop로 만들어 주는 flush를 진행해야 한다. 이는 IF/ID의 current_pc를 0으로, ID/EX의 컨트롤 비트들을 모두 0으로 만들어 잘못된 명령어의 결과가 programmer visible state에 반영되지 못하게 하였다.

추가로, control flow 관련 명령어들을 수행하기 위한 작업들도 cpu에 해 주어야 한다. 이를 구현하기 위해 다음과 같은 control 비트들을 정의해 주었다.

- **is_jal** : 명령어가 jal인 경우 그 정보를 EX 단계까지 갖고 가기 위한 비트
- **is_jalr** : 명령어가 jalr인 경우 그 정보를 EX 단계까지 갖고 가기 위한 비트
- **branch** : opcode가 BRANCH인 경우 1로 설정되는 비트
- **pc_to_reg**: 레지스터에 pc의 값과 관련된 값을 저장해야 할 때 켜지는 비트. jal이나 jalr 명령어는 레지스터에 pc+4를 저장해야 하기 때문에 이런 비트를 정의했다.

위의 정보들을 적절한 모듈에 연결해 주면 branch, jal, jalr 명령어를 ISA에 맞게 구현할 수 있다.

여담으로, 처음엔 BranchUnit 모듈을 만들어 분기 여부와 분기 시의 주소를 해당 모듈에서 결정하도록 했다. 그런데 이 작업들은 ALU에서 한번에 할 수 있는 작업이라, 모듈을 없애고 ALU의 정보와 mux 몇 개로 대체했다.

B. BTB

BTB는 목적지 address를 저장하기 위한 일종의 hash table이다.

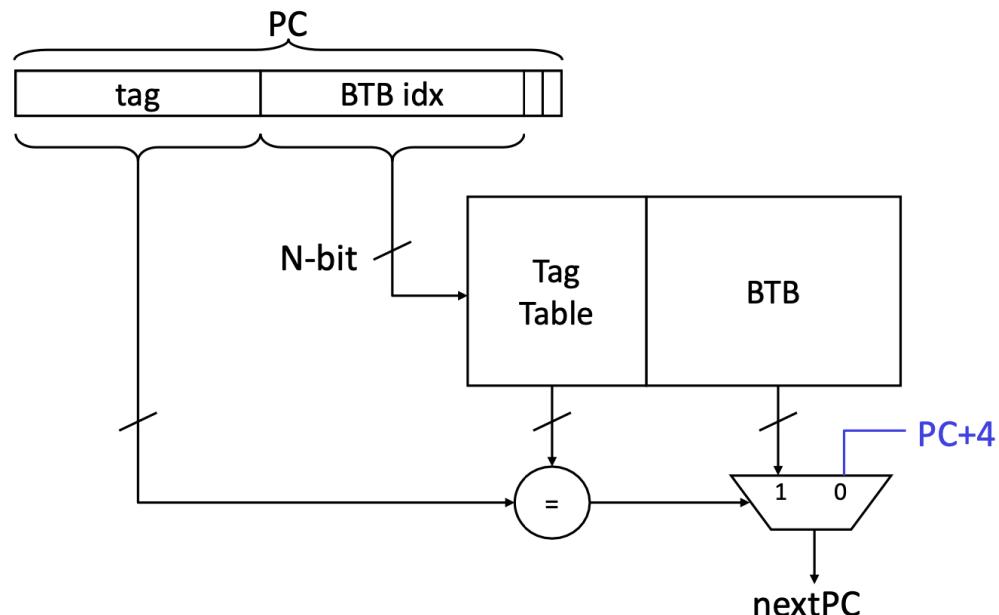
index와 tag를 이용해 BTB에는 branch, jal 명령에 따른 pc 이동 결과만 저장할 수 있도록 했다. (jump가 일어나지 않는 명령어들은 어차피 pc+4를 predict할 것이므로 낭비이고, jalr의 경우 목적지 address가 가변적이기 때문에 저장하지 않기로 했다.) 그림에서 알 수 있듯 pc의 tag 부분이 tag table의 BTB index에 저장된 값과 같다면 nextPC를 BTB에 저장된 값으로 예측한다. 만약 같지 않다면 pc+4를 다음 pc로 이용한다.

BTB는 모든 entry에 대해 0으로 초기화 되고, clock에 맞춰 업데이트 된다. BTB와 tag table은 업데이트는 pc가 branch나 jal 명령어이면서 flush상태가 아닌 경우에만 일어난다. BTB에는 다음 pc에 대한 정보, tag table에는 pc의 하위 5비트를 저장한다.

우리의 구현에서는 각 entry에 대해 valid 비트를 추가하였다. Cache로 비유하면 cold miss를 방지하기 위한 수단이다. 우선 우리는 tag table 엔트리를 모두 0으로 초기화하였다. 그래서 pc의 tag가 0인 경우 우리가 BTB를 업데이트하기 전에도 tag table과 pc의 tag가 일치할 수 있는데, BTB를 0으로 초기화했으므로 nextPC를 0으로 predict해버리는 상황이 발생한다. 따라서 이런 일을 방지하기 위해 valid 비트를 도입하였다.

여기까지를 바탕으로 Always-taken branch predictor을 만들 수 있다. 구조도는 아래와 같다.

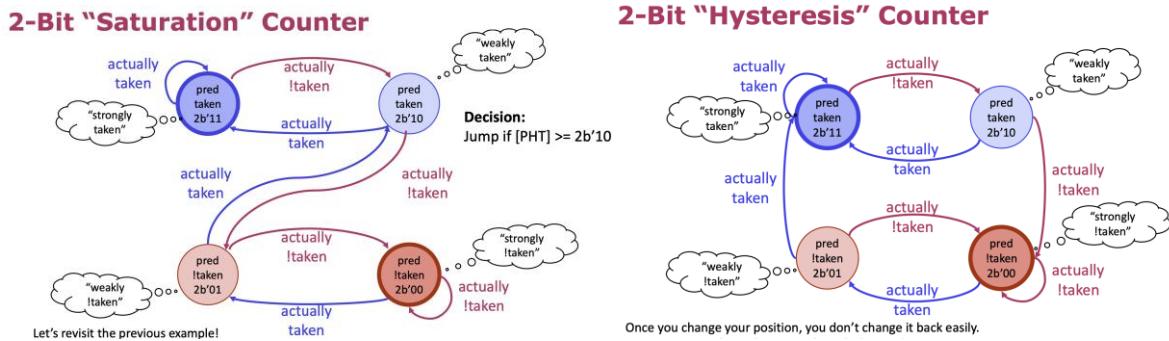
Tagged BTB



C. PHT

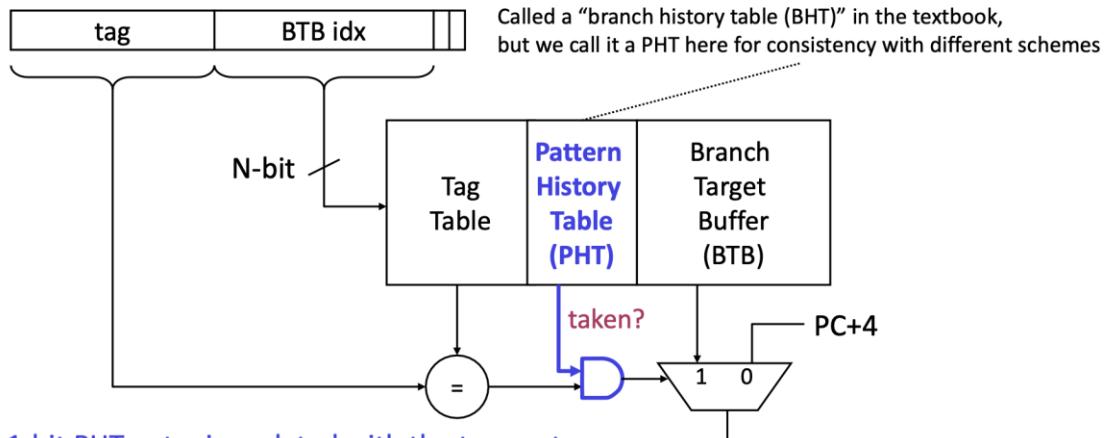
PHT는 이전의 branch 여부에 대한 정보를 담고 있는 모듈이다. 더 좋은 prediction을 하기 위해 PHT의 정보를 이용한다. PHT의 구현은 2-bit state machine으로 했으며 교안에 소개된 두 가지 방식으로 모두 구현해보았다.

PHT의 구현을 위해서 HysteresisCounter와 SaturationCounter 모듈을 만들었다. PHT의 값을 이런 counter들로 업데이트해 주기 위함이다. State machine의 디자인은 아래의 교안 내용을 참고했다. 아래와 같은 state machine들을 branch taken 여부의 정보를 이용해 업데이트하는 방식이다.



여기까지를 바탕으로 2-bit global predictor 을 만들 수 있다. 구조도는 아래와 같다.

Pattern History Table and Target Buffer

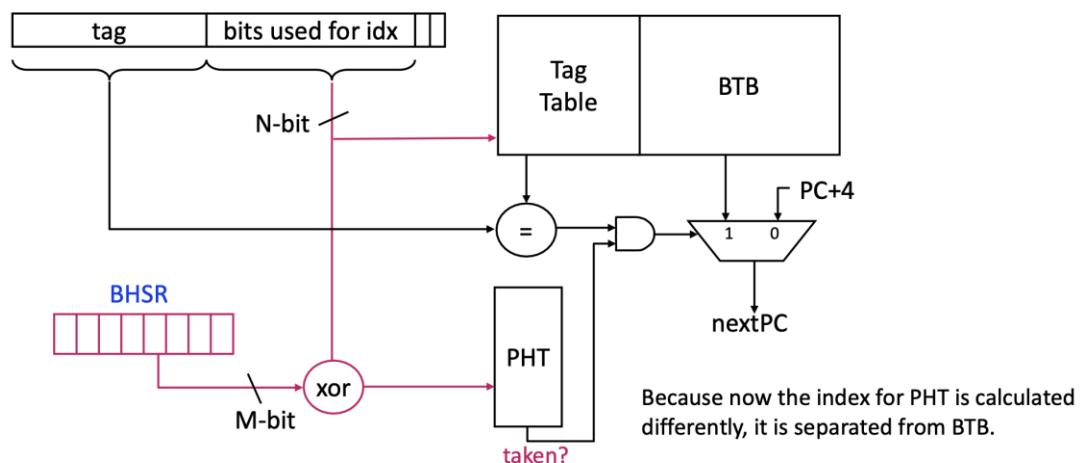


- The 1-bit PHT entry is updated with the true outcome after each execution of a branch
- Only taken branches and jumps are held in BTB
- Large or sophisticated PHT can be designed separately from the BTB

D. BHSR 및 Gshare 로직

Gshare 방식에서는 PHT와 더불어 BHSR이 추가되어야 한다. Global BHSR에는 프로그램 내의 모든 branch 명령어에 대해서 최근 m개의 명령이 taken이었는지 not taken이었는지에 대한 정보가 기록된다. m과 n이 꼭 같을 필요는 없지만, 본 구현에서는 PHT와 BTB 엔트리를 둘 다 32개로 맞추기 위해 둘 다 5비트로 길이를 맞췄다. 구조는 수업 ppt 자료에 나와 있는 다이어그램을 참고 했다.

“Gshare” Branch Prediction [McFarling]



- **Global BHSR (Branch History Shift Register)** tracks the outcomes of the last M branch instructions (e.g., NT → T ..., where T is 1 and NT is 0)
- Would a global BHSR be sufficient?

III. 구현

A. cpu.v의 수정

먼저 **jal/jalr/branch** 명령어들을 처리해야 함에 따라 일부 수정된 모듈들에 대해 설명하도록 하겠다.

- StallDetection.v

```
// for use_rs1 and use_rs2
assign rs1_and_rs2_conditions = (ID_opcode == `ARITHMETIC) || (ID_opcode == `STORE) || (ID_opcode == `BRANCH);
assign rs1_conditions = (ID_opcode == `ARITHMETIC_IMM) || (ID_opcode == `LOAD) || (ID_opcode == `JALR);
```

Data hazard를 막기 위해 stall이 필요한지를 판단하는 로직에서, rs1과 rs2를 사용하는 명령어의 종류에 **BRANCH**와 **JALR**을 추가했다.

- ControlUnit.v

- **is_jal**: 명령어가 jal인 경우 1로 설정
- **is_jalr**: 명령어가 jalr인 경우 1로 설정
- **branch**: 명령어의 opcode가 BRANCH인 경우 1로 설정
- **pc_to_reg**: 명령어가 jal/jalr인 경우 1로 설정

위의 4개의 출력 레지스터가 추가되었다. 또한 **case**문에서 이 레지스터들의 값을 결정하는 블록들이 추가되었다.

```
`BRANCH : begin
    branch = 1'b1;
    alu_src = 1'b0;
    alu_op = 2'b10;
end
`JALR : begin
    is_jalr = 1'b1;
    alu_src = 1'b1;
    reg_write = 1'b1;
    alu_op = 2'b11;
    pc_to_reg = 1'b1;
end
`JAL : begin
    is_jal = 1'b1;
    reg_write = 1'b1;
    alu_op = 2'b11;
    pc_to_reg = 1'b1;
end
```

Flush의 경우 reset 시그널과의 or 연산을 통해 nop로 처리하였다. 아래는 IF/ID 부분에서의 파이프라인 레지스터 코드인데, ID/EX에서도 비슷하게 reset과의 or을 통해 flush를 하였다.

```
// Update IF/ID pipeline registers here
always @(posedge clk) begin
    if (reset || prediction_wrong) begin
        IF_ID_inst <= 32'b0;
        IF_ID_pc <= ~32'b0;
        IF_ID_predicted_next_pc <= ~32'b0;
        IF_ID_predicted_branch_taken <= 1'b0;
    end
    else if (!is_stall) begin
        IF_ID_inst <= instruction;
        IF_ID_pc <= current_pc;
        IF_ID_predicted_next_pc <= predicted_next_pc;
        IF_ID_predicted_branch_taken <= predicted_branch_taken;
    end
end
```

cpu.v 자체는 II. 디자인에서 보여 준 구조도를 거의 그대로 코드로 옮긴 것이기 때문에 크게 설명할 것은 없어보인다. 한 가지 언급할만한 점은 pc가 초기화될 때는 ~32'b0으로 초기화된다는 것이다. 32'b0은 실제로 pc가 가질 수 있는 legal한 값이기 때문에 굳이 32'b0 대신 ~32'b0을 사용하였다. 나중에 보겠지만 BranchPredictor는 **update_pc**로 ~32'b0을 받은 경우 업데이트를 하지 않는다.

아래는 실제 주소(**actual_addr**)와 실제 분기 여부(**actual_branch_taken**), 그를 토대로 prediction이 맞았는지의 여부를 계산하는 코드이다. 주목할 점은 다음과 같다.

- **prediction_wrong**의 경우 **predicted_next_pc**가 ~32'b0이면 0으로 설정된다. 이는 pipeline으로 받은 명령어가 flush나 stall로 인해 nop로 바뀌었다는 의미이기 때문이다.
- **update_pc**의 경우 jal이나 branch 명령어가 아닌 경우 ~32'b0으로 설정된다.
 - **update_pc**는 BranchPredictor이 자신의 상태(BTB, PHT 등)를 업데이트하기 위해 받는 pc의 값인데, ~32'b0이면 무시한다. (앞서 논의했듯이 jalr이나 alu연산 등은 branch predictor로 다루지 않기로 했기 때문이다.)
 - 한 가지 더 봐야 할 점은 파이프라인으로 받은 명령어가 nop였을 경우 **ID_EX_pc**도 ~32'b0으로 초기화되어 branch predictor의 업데이트가 disable된다는 점이다. 이로써 branch predictor 내부에 추가적인 stall이나 flush에 대한 처리를 하지 않아도 된다는 것을 알 수 있다.

```
// jalr will use alu_result
// jal or branch will have pc + imm
// all other will have pc+4 (enforced by actual_branch_taken being false)
assign branch_addr = ID_EX_is_jalr ? alu_result : ID_EX_pc + ID_EX_imm;
assign actual_branch_taken = ID_EX_is_jal || ID_EX_is_jalr || (alu_bcond && ID_EX_branch);
assign actual_addr = actual_branch_taken ? branch_addr : ID_EX_pc + 4;

assign prediction_wrong =
    (ID_EX_predicted_next_pc != ~32'b0) &&
    (
        ID_EX_predicted_branch_taken != actual_branch_taken ||
        ID_EX_predicted_next_pc      != actual_addr
    );

// Assumption: BTB doesn't store JALR values
assign update_pc = (ID_EX_is_jal || ID_EX_branch) ? ID_EX_pc : ~32'b0;
```

B. BranchPredictor.v

앞서 언급했듯 우리는 Gshare predictor을 구현했다. 먼저 prediction logic을 알아본 뒤, update logic을 알아보자. Predictor의 내부에는 BTB, PHT, BHSR 등의 register이 있음도 기억하자.

B-1. Prediction Logic

아래 코드를 참고하자. 일단은 현재의 pc가 BTB의 entry와 태그가 일치하면, PHT의 entry에 따라 prediction을 진행하면 된다. (valid비트도 당연히 켜진 상태여야 할 것이다.)

```
// Prediction based on PHT
// if valid and the tags match, use BTB
assign tag_match = (tag[predict_BTB_index] == current_pc[31:5]);

assign predicted_next_pc = (valid[predict_BTB_index]) && (tag_match) && (PHT[predict_PHT_index][1])
    ? BTB[predict_BTB_index]
    : current_pc + 4;
assign predicted_branch_taken = (valid[predict_BTB_index]) && (tag_match) && (PHT[predict_PHT_index][1])
    ? 1'b1
    : 1'b0;
```

그럼 이제 중요한 부분은 BTB와 PHT의 몇 번째 entry를 사용할 것이냐는 문제이다. Gshare로 구현했으므로, BTB는 pc의 마지막 5비트, PHT는 pc의 마지막 5비트와 BHSR을 xor한 값을 활용하면 될 것이다.

```
assign predict_BTB_index = current_pc[4:0];
assign predict_PHT_index = current_pc[4:0] ^ BHSR;
```

B-2. Update Logic

아래의 코드는 BTB, PHT, BHSR에 대한 정의, 초기화 및 업데이트에 관련된 중추적인 로직이다. 이를 바탕으로 BTB, PHT, BHSR의 업데이트 로직을 하나하나 순서대로 설명하겠다.

```
// BTB, PHT, BSHR definition, initialization and update
reg [31:0]      BTB[0:31]; // 32-bit entries; 32 entries.
reg [26:0]      tag[0:31]; // (32-5)-bit entries; 32 entries.
reg             valid[0:31];
reg [1:0]       PHT[0:31]; // 2-bit counter; 32 entries.
reg [4:0]       BHSR;

always @(posedge clk) begin
    if (reset) begin
        for (i = 0; i < 32; i = i + 1) begin
            BTB[i] <= 32'b0;
            tag[i] <= 27'b0;
            valid[i] <= 1'b0;
            PHT[i] <= 2'b10;           // Default to guessing weakly taken
        end
        BHSR <= 5'b0;
    end
    else if (update_pc != ~32'b0) begin
        tag[update_BTB_index] <= update_pc[31:5];
        BTB[update_BTB_index] <= update_BTB;
        valid[update_BTB_index] <= 1'b1;
        PHT[update_PHT_index] <= next_PHT; // Calculated by a 2-bit counter module
        BHSR <= {BHSR[3:0], update_taken};
    end
end
```

BHSR

- 코드에 쓰인 대로, 전 BHSR 을 왼쪽으로 쉬프트한 뒤 새로운 branch_taken 정보를 받아 와 최하위비트에 배치하면 된다.

BTB

- Index 찾기: update_pc의 하위 5비트 이용
→ `assign update_BTB_index = update_pc[4:0];`
- 업데이트할 값: tag 에는 update_pc 의 상위 27 비트, BTB 에는 branch address. update_BTB 로 받은 값이 branch address 이고, 이를 BTB 에 저장하면 된다.

PHT

- Index 찾기: predict_PHT_index 를 두 사이클만큼 기다렸다가 사용
 - Predict 에 쓰인 PHT 의 엔트리가 업데이트되어야 하는 것이다. Prediction 에 대한 결과, 즉 update 해야 할 정보는 두 사이클만큼 이따가 predictor 에 도착하므로, prediction 에 활용된 PHT index 를 두 사이클만큼 기다리는 것으로 index 를 얻어낼 수 있다.

```
always @(posedge clk) begin
    if (reset) begin
        IF_ID_PHT_index <= 5'b0;
        ID_EX_PHT_index <= 5'b0;
    end
    else begin
        IF_ID_PHT_index <= predict_PHT_index;
        ID_EX_PHT_index <= IF_ID_PHT_index;
    end
end
assign update_PHT_index = ID_EX_PHT_index;
```

- 업데이트할 값: 2-bit saturation counter 이용. next_PHT 는 아래와 같은 모듈의 output 으로 산출된다. Counter 은 saturation counter, hysteresis counter 등을 사용할 수 있다. 둘 다 구현했는데 우리의 Gshare 에서는 saturation counter 를 사용했다. counter 들의 소스코드도 다음 장에 같이 첨부하겠다.

```
SaturationCounter saturation_counter(
    .branch_taken(update_taken),           // input
    .current_state(PHT[update_PHT_index]), // input
    .next_state(next_PHT)                 // output
);
```

```

module SaturationCounter(
    input branch_taken,
    input [1:0] current_state,
    output reg [1:0] next_state
);

    always @(*) begin
        case(current_state)
            2'b00: begin
                if(branch_taken) next_state = 2'b01;
                else next_state = 2'b00;
            end
            2'b01: begin
                if(branch_taken) next_state = 2'b10;
                else next_state = 2'b00;
            end
            2'b10: begin
                if(branch_taken) next_state = 2'b11;
                else next_state = 2'b01;
            end
            2'b11: begin
                if(branch_taken) next_state = 2'b11;
                else next_state = 2'b10;
            end
        endcase
    end
endmodule

```

```

module HysteresisCounter(
    input branch_taken,
    input [1:0] current_state,
    output reg [1:0] next_state
);

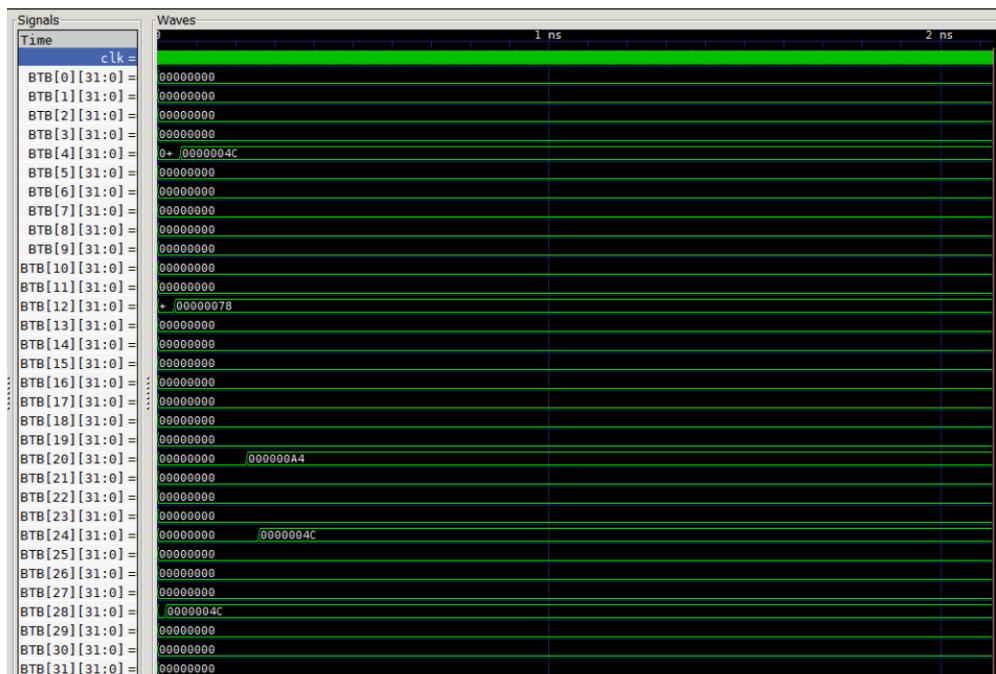
    always @(*) begin
        case(current_state)
            2'b00: begin
                if(branch_taken) next_state = 2'b01;
                else next_state = 2'b00;
            end
            2'b01: begin
                if(branch_taken) next_state = 2'b11;
                else next_state = 2'b00;
            end
            2'b10: begin
                if(branch_taken) next_state = 2'b11;
                else next_state = 2'b00;
            end
            2'b11: begin
                if(branch_taken) next_state = 2'b11;
                else next_state = 2'b10;
            end
        endcase
    end
endmodule

```

IV. 논의 사항

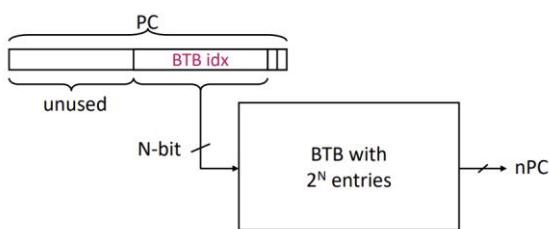
구현을 하다가, pc는 어차피 4의 배수로 align 되어 있어서 뒤의 두 비트는 항상 0으로 설정되어 있을 것이다. 그래서 BTB를 접근할 때 4의 배수인 entry만 접근하는 것은 아닌지 의심이 들었다.

실제로 gtkwave로 recursive 프로그램을 돌린 결과의 BTB를 확인했을 때 4의 배수인 entry만 업데이트가 된 것을 확인할 수 있다.



그래서 index로 `pc[4:0]`를 사용하던 것을 `pc[6:2]`로 수정할까 고민했으나, 교안에서도 그냥 `pc[4:0]`을 사용하는 것 같아서 그렇게 놔두었다. 이 판단의 근거가 되는 교안의 슬라이드를 아래에 첨부하였다.

When a Collision Happens?



- Assume BTB index is 10 bit \rightarrow 1024-entry BTB

PC: 00000000000000000000000000000000 0011001100 : ADD \rightarrow PC+4?
VS
PC: 11111.....1111111111111111 0011001100 : JAL \rightarrow A target?
VS
PC: 11100.....0011.....000....110 0011001100 : SUB \rightarrow PC+4?

V. 결론

A. 결과 비교

우선 AlwaysNotTaken, AlwaysTaken, 2BitSaturation, Gshare에 대한 각각의 테스트 결과를 비교하겠다. tb의 결과에 대한 스크린샷은 이후부터 첨부되어 있다.

	NOT TAKEN	TAKEN	2-BIT	GSHARE	(PROVIDED)
BASIC	35	35	35	35	36
IFELSE	43	43	43	43	44
LOOP	322	290	294	290	323
NON-CONTROLFLOW	46	46	46	46	46
RECURSIVE	1187	1075	1079	1067	1188

모든 버전이 제공된 사이클 수와 일치하거나 작다. 각 버전들에서 차이가 있는 부분은 loop test 와 recursive test였다. Always not taken CPU에서 가장 많은 사이클 수를 필요로 했고, 이는 예상한 결과였다. Branch의 경우 평균적으로 take하는 경우가 그렇지 않은 경우보다 많기 때문이다. Gshare에서 가장 적은 사이클을 이용했으며 이 또한 예상한 결과였다. Gshare는 지난 branch들에 대한 정보를 학습해 예측 값을 내놓기 때문이다.

눈에 띄는 부분은 2-bit state machine으로 구현한 버전이 always taken CPU보다 더 많은 사이클 수를 필요로 했다는 점이다. 이러한 결과는 테스트 코드가 어떻게 구성되어 있느냐에 따라 달라질 수 있다. 만약 테스트 코드에 거의 항상 not taken 되는 branch가 많았다면 2-bit state machine이 더 좋은 성능을 보였을 것이다. 하지만 loop는 loop를 빠져나올 때를 제외하면 항상 taken 되는 branch를 갖고 있기 때문에 조금이나마 always taken CPU에 유리한 부분이 있었던 것으로 분석했다.

C. Always-taken CPU

## SIMULATING ## TEST END SIM TIME : 72 TOTAL CYCLE : 35 (Answer : 36) FINAL REGISTER OUTPUT	## SIMULATING ## TEST END SIM TIME : 88 TOTAL CYCLE : 43 (Answer : 44) FINAL REGISTER OUTPUT	## SIMULATING ## TEST END SIM TIME : 582 TOTAL CYCLE : 290 (Answer : 323) FINAL REGISTER OUTPUT
0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002fffc (Answer : 00002fffc) 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 ffffffd7 (Answer : ffffffd7) 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)	0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002fffc (Answer : 00002fffc) 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 Correct output : 32/32 Correct output : 32/32

## SIMULATING ## TEST END SIM TIME : 94 TOTAL CYCLE : 46 (Answer : 46) FINAL REGISTER OUTPUT	## SIMULATING ## TEST END SIM TIME : 2152 TOTAL CYCLE : 1075 (Answer : 1188) FINAL REGISTER OUTPUT	
0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002fffc (Answer : 00002fffc) 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 ffffff1 (Answer : ffffff1) 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)	0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002fffc (Answer : 00002fffc) 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 0000000d (Answer : 0000000d) 11 00000000 (Answer : 00000000) 12 00000000 (Answer : 00000000) 13 00000000 (Answer : 00000000) 14 00000001 (Answer : 00000001) 15 0000000d (Answer : 0000000d) 16 00000015 (Answer : 00000015) 17 0000000a (Answer : 0000000a) 18 00000000 (Answer : 00000000) 19 00000000 (Answer : 00000000) 20 00000000 (Answer : 00000000) 21 00000022 (Answer : 00000022) 22 00000000 (Answer : 00000000) 23 00000037 (Answer : 00000037) 24 00000059 (Answer : 00000059) 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 Correct output : 32/32

