Computer Architecture (CSED 311), Spring 2025
Lab 4-2: Pipelined CPU w/ control flow instructions

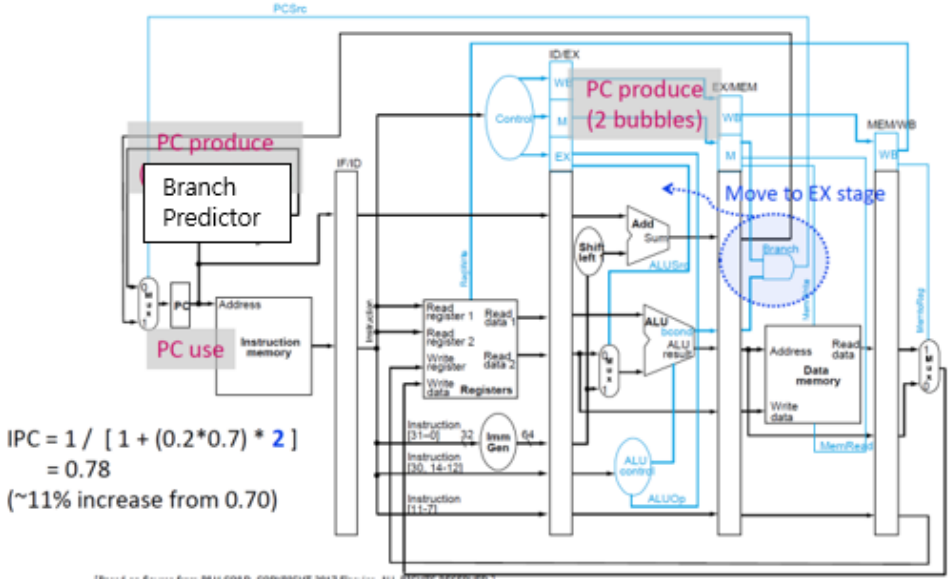
구현웅 (20210940), 김민서 (20220826)
5/13, 2025

1. Introduction

본 랩에서는 non-control flow pipelined CPU를 바탕으로 control flow와 branch prediction 포함한 pipelined CPU를 제작한다. Control flow의 처리 중 발생하는 control hazard를 분석하고 flush를 통해 해결하며, control hazard에 대한 비용을 줄이기 위해 branch prediction을 도입한다. 이를 통해 본 랩은 pipelined CPU에서 control hazard를 최소화하고 검출 및 해결 방안에 대해 이해하는 것을 목표로 한다.

2. Design

본 랩의 기존 pipelined CPU의 구조를 기반으로 한다. Data hazard와 마찬가지로 branch hazard는 BranchHazardUnit 검출 및 관련된 신호를 생성한다. 생성된 pc의 값과 그 이후에 fetch된 pc의 값을 비교해 서로다르다면 branch hazard가 발생한 것으로 취급한다. 또한, 랩에서 제시한 것과 같이 branch, jar, jarl에 나타날 수 있는 pc의 값인 pc+imm, pc+register 값은 모두 EX 단계에서 생성되어, 기존의 forwarding을 그대로 사용할 수 있도록 구성하였다. pc의 값은 non-control flow instruction의 경우에는 항상 pc+4를 사용하도록 하며, 그외에는 branch predictor에서 제공하는 값을 다음 pc로 사용한다. Branch predictor와 branch hazard는 서로 독립적으로 구현 및 동작한다.



2.1. Handling branch hazard

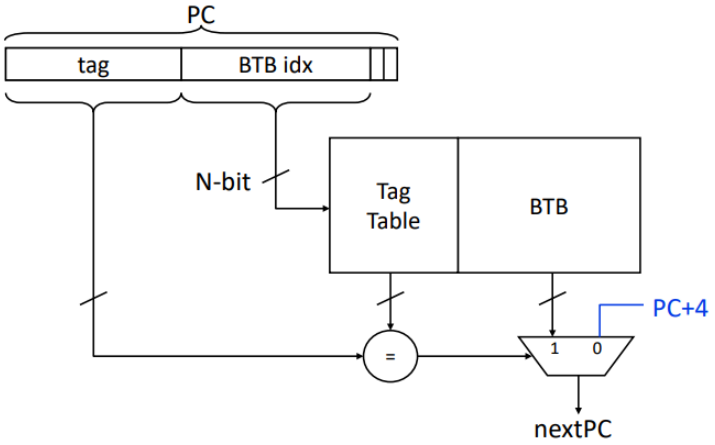
Branch prediction을 통해 control flow instruction에 대해서도 다음 pc가 결정이 되기 전에 계속 instruction fetch를 수행한다. 이에 따라 예측된 pc의 값이 EX 단계에서 생성된 실제 pc의 값과 다르다면 fetch된 잘못된 instruction을 모두 제거된다. 적절한 forwarding 통해 EX

stage에서 pc+imm, pc+register의 값이 생성되며, 이 값과 ID stage에 들어간 instruction의 pc를 비교함으로 branch hazard를 검출할 수 있다. Branch hazard를 검출했다면 이전의 ID의 insturciton과 IF의 instruction은 잘못된 pc에서 유래했으므로 flush한다. IF의 instruction은 IF/EX register에 non-op instruction으로 덮어 써서 구현할 수 있으며, ID의 instruction도 유사하게 ID/EX의 write signal을 모두 비활성화해 EX/MEM register 넣어주어 구현한다.

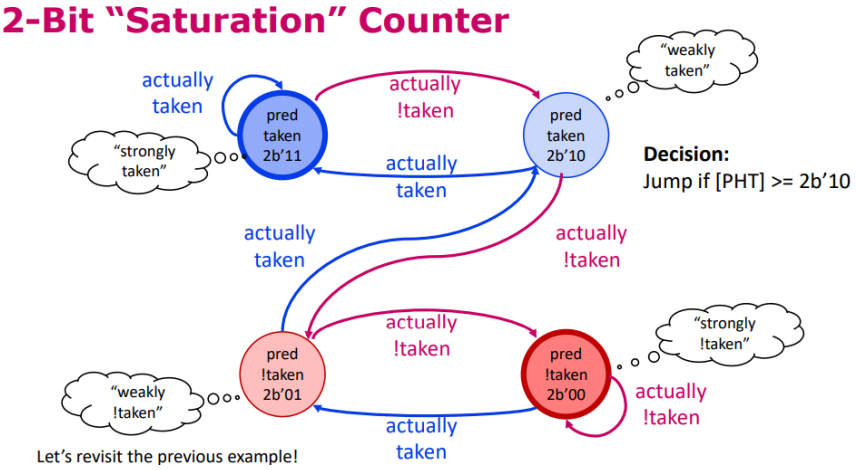
2.2. Branch predictor

본 구현의 Branch predictor는 BTB(branch target buffer)와 PHT(pattern history table) 크게 두 가지 영역으로 나뉜다. BTB는 pc을 index로 활용해 예측 pc의 값을 저장하는 테이블이며, PHT는 BTB의 pc를 사용할지 말지에 대해 branch의 patter을 저장해둔 테이블이다.

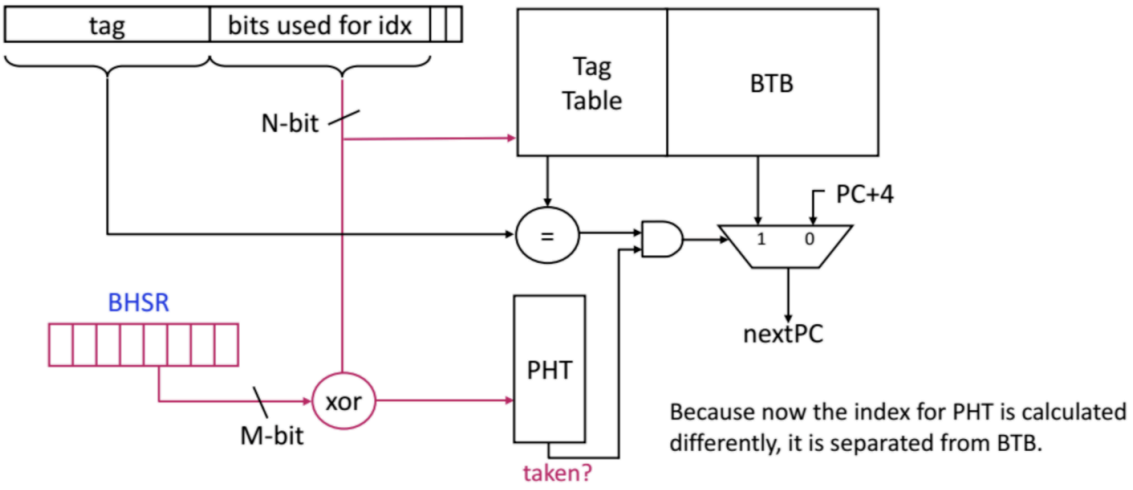
BTB는 pc의 index와 tag bit로 나누어 index를 기준으로 테이블에서 요소를 불러오며, 각각의 요소는 pc와 tag를 함께 가지고 있어 만약 pc의 tag와 BTB의 tag가 서로 같다면 BTB의 pc데이터를 채택한다.



PHT는 branch에 대해 예측 결과를 수용했는지에 대해 그 결과를 bit shifter regter에 저장해두며, 이와 pc의 xor 연산 결과를 PHT의 index로 이용한다. PHT의 각각의 원소는 2-bit saturation counter이며, 각각의 상태에 따라 BTB에서 예측한 pc를 사용할지 결정한다. 그리고 예측한 pc의 값이 올바랐는지에 따라 각각의 state는 갱신된다.



Gshare은 여기에 BHSR (Branch History Shift Register) 레지스터를 추가하여 이전 분기 이력으로부터 현재의 분기를 예측한다. BHSR은 이전 분기 명령어들의 branch taken 여부를 관리하는 레지스터로, PHT index를 구할 때 BTB index와 xor 연산되어 사용된다.



2.3. Comparing total cycles

Pipelined CPU의 forwarding에 따르면 data hazard가 발생하면 1 cycle 만큼의 stall이 발생한다. 이는 control flow instruction에 대해서도 마찬가지이며, memory write를 수행하고 바로 다음에 branch, jalr에 의해 같은 위치에 read를 시행한다면 data hazard가 발생해 1 cycle stall이 발생한다.

Branch hazard는 predictor의 성능에 따라 요구되는 cycles가 크게 바뀐다. branch, jar, jalr에 대해 predictor 올바른 pc의 값을 생성했다고 하면 flush는 일어나지 않고 손실되는 cycle은 없다. 하지만 잘못된 pc의 값을 생성했다고 하면 ID stage와 IF stage의 instruction은 flush되어 총 2 cycles 만큼의 손해를 보게 된다. Control flow instruction은 pc에 값에 따라 1 cycle 혹은 3 cycles 소요된다고 생각할 수 있다. 수치적인 예시를 들어보자면, 총 100개의 instruction으로 이루어진 코드에 data hazard는 없다고 하자. 그 중에서 20%는 control flow instruction이고 predictor가 예측에 성공할 확률을 70%로 가정한다면 80 cycles + (1 cycle * 14 + 3 cycle * 6) = 112 cycles 이 도출되며, 따라서 cycles이 12% 추가로 소요된다.

3. Implementation

본 랩에서는 이전 lab 4-1에서 구현한 pipelined CPU에 branch hazard 감지, branch prediction을 실행하는 모듈을 추가로 구현하여 control-flow instruction에 대해서도 사용할 수 있도록 확장하였다.

3.1. Branch Hazard

Branch hazard 감지는 ID stage에서 이루어진다. JAL, JALR, BRANCH 명령어에서 이전에 예측한 pc와 실제 pc의 값을 비교하여 차이가 있을 경우, 바로 이전 stage에 있는 두 명령어는 잘못 실행되었으므로 IF, ID stage를 flush한다. flush하지 않는 경우, non-control flow는 PC+4를, control flow는 branch predictor의 예측을 가져다가 next_pc로 쓴다. flush해야 하는 경우는 잘못 분기 예측된 명령어의 실제 next pc를 사용하도록 한다. 구현은 다음과 같다.

```
module BranchHazardUnit (
    input      [ 6:0] IF_opcode,
    input      [ 6:0] ID_EX_opcode,
    input      [31:0] IF_pc,         //pc on IF stage
    input      [31:0] IF_ID_pc,      //pc on ID stage
    input      [31:0] ID_EX_pc,      //pc on EX stage
    input      [31:0] pc_BTB,        //BLT output
    input      [31:0] pc_4,          //pc+4
    input      [31:0] pc_imm,        //pc+imm
    input      [31:0] pc_A,          //alu_result
    input             bcond,         //rs1_dout == rs2_dout
    output reg        flush_IF_ID,
    output reg        flush_ID_EX,
    output reg [31:0] next_pc
);

  wire [31:0] branch_real_pc;
  wire pc_4_signal;

  assign branch_real_pc = bcond ? pc_imm : ID_EX_pc + 4;
  assign pc_4_signal = (IF_opcode == `ARITHMETIC) || (IF_opcode ==
`ARITHMETIC_IMM) || (IF_opcode == `LOAD) || (IF_opcode == `STORE) ||
(IF_opcode == `ECALL);

  always @(*) begin
    // JALR, detected in EX stage
    if (ID_EX_opcode == `JALR && pc_A != IF_ID_pc) begin
      next_pc = pc_A;
      flush_IF_ID = 1;
      flush_ID_EX = 1;
    end  // JAL, detected in EX stage
    else if (ID_EX_opcode == `JAL && pc_imm != IF_ID_pc) begin
      next_pc = pc_imm;
      flush_IF_ID = 1;
      flush_ID_EX = 1;
    end  // Branch, detected in EX stage
    else if (ID_EX_opcode == `BRANCH && branch_real_pc != IF_ID_pc)
begin
      next_pc = branch_real_pc;
      flush_IF_ID = 1;
      flush_ID_EX = 1;
    end  // On non-control inst, next pc is static ( pc+4 )
    else if (pc_4_signal == 1) begin
      next_pc = pc_4;
      flush_IF_ID = 0;
      flush_ID_EX = 0;
```

```
    end else begin
      // On control flow, use BLT pc
      next_pc = pc_BTB;
      flush_IF_ID = 0;
      flush_ID_EX = 0;
    end
  end

endmodule
```

3.2. Branch Prediction

본 랩에서 구현한 Gshare branch predictor는 BTB, PHT, 그리고 BHSR로 구성된다. BHSR은 Branch 명령어가 실행되었을 때만 업데이트된다. PHT의 업데이트 여부는 분기 예측 후 두 사이클 이후에 EX stage에 도달했을 때 구할 수 있다. PHT의 구현에는 2-bit saturation counter를 사용하였다. BTB와 PHT의 업데이트 조건은 branch가 taken될 때만 해당된다. branch가 taken된 여부는 EX stage에서 명령어가 JAL, JALR이거나 또는 BRANCH인 동시에 alu에서 branch condition이 1인 경우이다.

```
module BranchPrediction #(
    parameter integer BTB_DEPTH = 32,
    parameter integer TAG_WIDTH = 30 - $clog2(BTB_DEPTH),
    parameter integer BTB_IDX_WIDTH = $clog2(BTB_DEPTH)
) (
    input  [              6:0] opcode,
    input  [              6:0] ID_EX_opcode,
    input  [             31:0] ID_EX_pc,
    input  [BTB_IDX_WIDTH-1:0] ID_EX_pht_idx,
    input  [             31:0] pc,
    input  [             31:0] pc_4,          // pc+4
    input  [             31:0] predicted_branch_target,
    input  [             31:0] actual_branch_target,
    input                      actual_taken,
    input                      reset,         // positive reset signal
    input                      clk,
    output [             31:0] pc_predicted,
    output [BTB_IDX_WIDTH-1:0] pht_idx
);
  integer i;

  wire use_btb = opcode == `JAL || opcode == `JALR || opcode == `BRANCH;
  wire ID_EX_use_btb = ID_EX_opcode == `JAL || ID_EX_opcode == `JALR ||
ID_EX_opcode == `BRANCH;

  reg [31:0] btb[BTB_DEPTH];
```

```
  reg [TAG_WIDTH-1:0] tag_table[BTB_DEPTH];
  reg [BTB_IDX_WIDTH-1:0] bhsr;
  reg [1:0] pht[BTB_DEPTH];

  wire [BTB_IDX_WIDTH+1:2] btb_idx;
  wire [BTB_IDX_WIDTH+1:2] ID_EX_btb_idx;
  wire [TAG_WIDTH-1:0] tag;
  wire [TAG_WIDTH-1:0] ID_EX_tag;
  wire [TAG_WIDTH-1:0] tag_table_entry;
  wire pht_taken;
  wire is_predict;
  wire is_predict_correct;

  assign {tag, btb_idx} = pc[31:2];
  assign {ID_EX_tag, ID_EX_btb_idx} = ID_EX_pc[31:2];
  assign pht_idx = bhsr ^ btb_idx;
  assign pht_taken = pht[pht_idx][1];
  assign is_predict = use_btb && pht_taken && tag == tag_table[btb_idx];
  assign pc_predicted = is_predict ? btb[btb_idx] : pc_4;
  assign is_predict_correct = actual_branch_target ==
predicted_branch_target;

  always @(posedge clk) begin
    if (reset) begin
      for (i = 0; i < BTB_DEPTH; i = i + 1) begin
        btb[i] <= 0;
        tag_table[i] <= 0;
        pht[i] <= 2'b11;
      end
      bhsr <= 0;
    end else begin
      if (ID_EX_use_btb) begin
        if (ID_EX_opcode == `BRANCH) begin
          bhsr <= {actual_taken, bhsr[BTB_IDX_WIDTH-1:1]};
        end
        if (actual_taken) begin
          if (!is_predict_correct) begin
            btb[ID_EX_btb_idx] <= actual_branch_target;
            tag_table[ID_EX_btb_idx] <= ID_EX_tag;
          end
          unique case (pht[ID_EX_pht_idx])
            2'b11: pht[ID_EX_pht_idx] <= 2'b11;
            2'b10: pht[ID_EX_pht_idx] <= 2'b11;
            2'b01: pht[ID_EX_pht_idx] <= 2'b10;
            2'b00: pht[ID_EX_pht_idx] <= 2'b01;
          endcase
```

```
        end else begin
          unique case (pht[ID_EX_pht_idx])
            2'b00: pht[ID_EX_pht_idx] <= 2'b00;
            2'b01: pht[ID_EX_pht_idx] <= 2'b00;
            2'b10: pht[ID_EX_pht_idx] <= 2'b01;
            2'b11: pht[ID_EX_pht_idx] <= 2'b10;
          endcase
        end
      end
    end
  end


endmodule
```

4. Discussion

4.1. Compare Cycle

2-bit global prediction의 효과를 보기 위해 recursive.asm에 대해 cycles 분석을 비교해본다. Always not taken은 pc+4를 그대로 사용하는 경우이며, always taken은 non-control flow에서는 pc+4를, control-flow에서는 BTB의 예측을 사용한다. Gshare은 PHT와 BHSR로부터 taken 여부를 결정한다. 다음이 실험의 결과이다.

always not taken	always taken	Gshare
1187 cycles	1047 cycles	1045 cycles

Always not taken의 경우 항상 pc+4를 fetch함으로 매 brach마다 flush가 일어나 가장 많은 cycles를 소모하고 있으며, always taken의 경우 BTB의 예측을 사용함으로 훨씬 cycle의 수가 줄어든 것을 확인할 수 있다. Gshare의 경우 PHT의 각 entry는 state가 11, 즉 strongly taken 상태로 초기화되어 BTB의 값을 채택하는 것으로 시작하기에 always taken과 큰 cycles 차이를 보이지는 않는다. 하지만 코드의 수가 길어지고 분기가 복잡해질수록 gshare과 always taken의 차이는 더욱 벌어질 것으로 기대할 수 있다.

5. Conclusion

본 랩에서는 branch prediction을 수행하는 pipelined CPU를 설계하였다. Branch prediction에 Gshare를 이용해 global branch를 반영하도록 하였으며, PHT는 2bit의 상태를 지녀 유동적으로 BTB의 값을 사용할지 결정하도록 구현했다. 또한, 예측한 pc가 실제와 다를 경우에는 CPU에 fetch된 instruction을 non-op로 바꾸거나 control signal에서 write를 비활성화 시켜 flush를 구현하였다. 이 과정을 통해 pipelined CPU에서 control flow를 어떻게 처리할 것이며 control hazard에 대응하고 최소화하는 방법을 익혔다. 게다가, Control flow라는 새로운 시스템은 pipelined CPU에 도입하는 과정은 CPU 구조의 이해와 확장에도 큰 견식을 주었다.