CSED311 Lab4-2a

-Pipelined CPU, Branch Predictor-

Team ID: 2

20180740 염재후

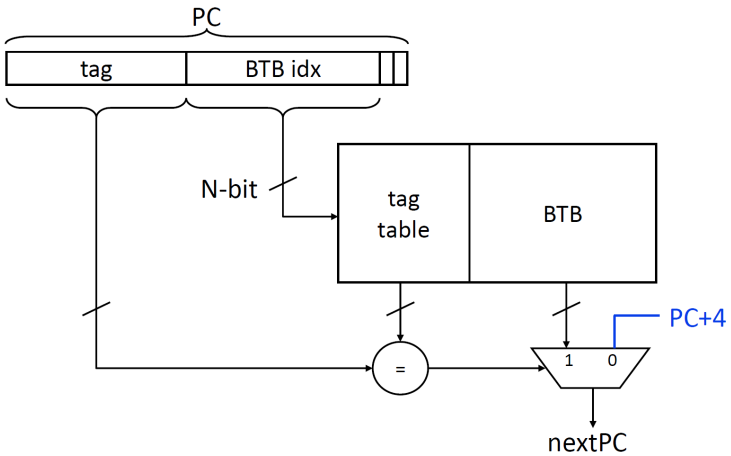
1. Introduction

이번 랩에서는 branch predictor를 통해 Control hazard를 다뤄보고자 한다. Control hazard란, 구현에 따라 다르지만, 이번 랩에서는 다음 PC value를 계산하는 Stage가 EX Stage이기 때문에 나타나는 문제이다. 다음 PC value가 EX Stage에서 계산된다면, 그 동안 다음 Instruction에 접근하지 못하게 된다. 따라서 Branch Predictor를 통해서 다음 PC value를 예측하는 것이다. 이렇게 예측된 PC value로 Instruction에 접근하여 계속 실행하되, 예측한 PC value와 실제 PC value를 비교하여, 같지 않을 경우, Correctness를 보장할 수 없기 때문에, 그 동안 실행되었던 instruction들을 flush함과 동시에 계산된 PC value부터 다시 시작하게 된다.

이번 Lab에서는 always not taken, always taken, 2bit global prediction, gshare prediction의 선택지가 있었다. 그 중 선택한 predictor는 always taken으로, always not taken과도 비교해보고자 한다.

2. Design

다음 사진은 수업시간에 Gshare Branch Prediction에 대해서 다룬 수업 슬라이드이다.



Always taken predictor의 디자인은 이 슬라이드에 맞춰서 할 수 있었다. 먼저, BTB(Branch Target Buffer) entry가 32개이기 때문에 BTB index bit는 5bit가 된다. 따라서, tag bit에 25bit, Index bit에 5bit, 마지막으로, PC의 LSB와 바로 그 옆 비트는 항상 0이기 때문에 Branch Predictor에서 사용하지 않는다. 이 때 이 슬라이드에서 나오지는 않았지만, tag table과 BTB를 초기화하는 과정에서, 동일한 0x0이더라도, 초기화 된 값과 BTB에 저장된 값을 다르게 인식하기 위해서 각 index마다 valid bit를 두어서, tag와 BTB entry가 초기화된 값인지 아닌지 판단하게 두었다.

Branch Predictor 외에도, 지난 lab과 비교해서 Data path의 변화도 필요하다. 가장 먼저, 실제로 이동했어야 할 PC를 계산해주는 모듈과, 이를 예측했던 PC value와 비교해주는 module이 필요하다. 그 외에도, jalr, jalr, branch instruction을 처리하기 위한 control signal과 control signal을 저장하기 위한 register가 필요하다.

원래 가야 했었을 actual PC를 계산해주는 module은 branch와 alu_bocond 혹은 jal을 고려하여 PC+Imm value를 할당하거나, jalr을 통해 GPR[rs1]+Imm 혹은 PC+4를 계산해줘야 한다. 또한 이렇게 계산된 actual PC와 예측했던 값인 pred PC을 비교하여 만약 실제로 가야 했었던 control flow와 다르다면, 잘못된 pc value로 업데이트 되었던 instruction을 flush해주고, 동시에 PCSrc를 조정하여, actual PC가 업데이트 될 수 있도록 해준다.

3. Implementation

a. Compute_Actual_PC

```
module compute_actual_pc(
    input [`WordSize-1: 0] current_pc,
    input [`WordSize-1: 0] rs1,
    input [`WordSize-1: 0] imm_gen_out,
    input isbranch,
    input isjal,
    input isjalr,
    input alu_bcond,
    output reg [`WordSize-1: 0]target
);
    always@(*) begin
        if(isjal || (isbranch && alu_bcond)) begin
            target = current_pc + imm_gen_out;
        end

        else if(isjalr) begin
            target = (rs1 + imm_gen_out) & 32'hfffffffe;
        end

        else
            target = current_pc + 4;
    end
endmodule
```

ID/EX register까지 가지고 온 PC value를 이용하여 실제로 가야했던 PC value를 계산해주는 모듈이다. Jal과 branch, 그리고 jalr이 서로 계산하는 방법이 다르기 때문에, 각 경우를 나눠서 target을 계산해준다.

b. Always Taken Predictor

```
    wire [4:0] BTB_index;
    wire [24:0] BTB_tag;

    //for write BTBs
    wire [4:0] BTB_past_index;
    wire [24:0] BTB_past_tag;


// registers
    reg valid_bit[0:31];
    reg [24: 0] tag_bit [0:31];
    reg [`WordSize -1:0] BTB [0:31];

// assign
    //for read BTB
    assign BTB_index = current_pc[6:2];
    assign BTB_tag = current_pc[31:7];

    // for write BTB
    assign BTB_past_index = pc_of_target[6:2];
    assign BTB_past_tag = pc_of_target[31:7];
```

```
always@(*) begin
    if(tag_bit[BTB_index] == BTB_tag && valid_bit[BTB_index]) begin
        pred_pc = BTB[BTB_index];
        istaken = 1;
    end
    else begin
        pred_pc = current_pc + 4;
        istaken = 0;
    end
end

// Write BTB synchronoushly
always@(posedge clk) begin
    if(reset) begin
        for (i = 0; i < 32; i = i + 1) begin
            BTB[i] <= 32'b0;
            valid_bit[i] <= 0;
            tag_bit[i] <= 0;
        end
    end

    else begin
        if(is_conditional) begin
            valid_bit[BTB_past_index] <= 1;
            tag_bit[BTB_past_index] <= BTB_past_tag;
            BTB[BTB_past_index] <= actual_target;
        end
    end
end
```

각각 Always Taken Preidictor의 declaration과 logic part이다. synchronous하게

write하고, asynchronous하게 read한다. 먼저 read의 경우, 미리 assign해둔 index 값을 통해서 값을 tag와 valid bit를 통해서 Taken인 경우에, branch를 taken으로 예측한다. 즉, 저장된 tag라면 무조건 taken을 한다.

Synchronous write의 경우, branch, jal, jalr인 경우에만 tag bit를 업데이트 하도록 했다. Actual taken일 경우, 동일한 pc를 업데이트하고, Actual not taken일 경우, 업데이트해주기 때문에 Taken, Not taken에 따라 나눌 필요가 없다.

c. Mispredicion detector

```
module misprediction_detector_always_taken(
    input [`WordSize -1: 0] pred_pc,
    input [`WordSize -1: 0] target,
    input is_conditional,
    output reg PCSrc,
    output reg isflush
);
//if PCSrc = 0, next pc is pred_pc
//if PCSrc = 1, next pc is target
    always@(*) begin
        //isconditional is asserted if branch or jal or jalr
        if(pred_pc != target && is_conditional) begin
            PCSrc = 1;
            isflush = 1;
        end

        else begin
            PCSrc = 0;
            isflush = 0;
        end
    end
endmodule
```

예측했던 PC value와 다를 경우에 앞서 언급했던 module에서 계산한 actual PC를 PC로 업데이트하기 위한 PCSrc signal을 asserted해주며, 동시에 flush signal을 asserted해줘서 잘못 갱신된 instruction들을 flush해준다. 그렇지 않을 경우, 정확히 예측한 것이기 때문에 계속해서 진행한다.

d. Control Unit

```
`JAL: begin
    mem_read = 0;
    mem_to_reg =0;
    mem_write =0;
    reg_write = 1;
    alu_src = 1;
    pc_to_reg = 1;
    isbranch =0;
    isjal=1;
    isjalr=0;
    ALUOp = `ALUOP_Else;
end

`BRANCH : begin
    mem_read = 0;
    mem_to_reg = 0;
    mem_write = 0;
    reg_write = 0;
    alu_src = 0;
    pc_to_reg = 0;
    isbranch =1;
    isjal=0;
    isjalr=0;
    ALUOp = `ALUOP_SBtype;
end

`JALR: begin
    mem_read = 0;
    mem_to_reg =0;
    mem_write =0;
    reg_write = 1;
    alu_src = 0;
    pc_to_reg = 1;
    isbranch =0;
    isjal=0;
    isjalr=1;
    ALUOp = `ALUOP_Else;
end
```

지난 Lab에서는 사용하지 않았던 jalr, branch, jal을 표시하는 signal과 pc to reg signal이 새로 추가되었다.

e. ALU input

```
assign alu_in_1_pc = (ID_EX_pc_to_reg) ? ID_EX_PC : alu_in_1;
assign alu_in_2_imm_pc = (ID_EX_pc_to_reg) ? 4 : alu_in_2_imm;
ALU alu (
  .alu_op(alu_op),      // input
  .alu_in_1(alu_in_1_pc),    // input
  .alu_in_2(alu_in_2_imm_pc),    // input
  .alu_result(alu_out),  // output
  .alu_bcond(alu_bcond)     // output
);
```

Rd register에 PC+4를 계산하는 작업이 추가되면서, pc_to_reg signal이 asserted 될 경우, alu에 들어가는 input을 mux를 통해 선택되도록 했다.

4. Discussion

Always not taken, always taken의 방법을 사용해 branch를 예측해봤다. 각각 걸린 cycle은 다음과 같다.

	Not taken	Taken
Non-control flow	59	59
If else	43	43

Recursive	1187	1035

이에 대해서 non control flow가 동일한 이유는 Branch prediction이 관여할 여지가 없었으며, if else가 다른 이유는 너무 짧은 cycle이기 때문에 valid bit가 1로 된 후, always taken으로 예측하기 전에 끝났을 것이라고 예측했다. 실제로 ripes에서 돌려본 결과로도, 동일한 branch가 2번 이상 나온 경우가 없었다.

Recursive의 경우, cycle이 다르게 나오게 되었는데, Taken이 Not taken에 비해 더 좋은 효율을 보여주는 것을 알 수 있다.

5. Conclusion

이 랩을 통해서 always not taken, always taken같은 다양한 Branch Predictor를 구현할 수 있었고, 이를 통해서 Control Hazard를 더 효율적으로 해소하는 방법에 대해서 더 자세히 알 수 있었다.