1. Introduction

   본 과제의 목적은 Verilog를 이용하여 간단한 Pipelined CPU를 설계하고 구현함으로써, Pipelined CPU의 동작 원리를 이해하는 데 있다. Pipelined CPU는 여러 명령어를 병렬로 처리하여 성능을 향상시키는 구조를 가지며, 본 과제에서는 컨트롤 플로우를 포함하여 Branch Prediction을 적용한 Pipelined CPU를 구현한다. 이를 통해 파이프라인 구조의 기본 개념과 동작을 학습하고, 하드웨어 설계에서의 병렬 처리 기법을 익히는 데 중점을 둔다.

2. Design

   CPU는 Moore Machine 기반의 FSM(Finite State Machine)으로 설계되었다. 본 과제에서 구현한 Pipelined CPU는 이전 Lab2에서 구현한 Single Cycle CPU를 기반으로 하며, 이를 파이프라인 구조로 확장하였다. 이를 위해 일부 모듈의 인터페이스를 수정하고, 추가적으로 Hazard Detection Unit, Forwarding Unit, 그리고 Gshare를 구현하여 데이터 및 제어 흐름의 안정성을 확보하였다.

   Branch Prediction이란 IF(Instruction Fetch) 단계에서 PC(Program Counter)를 기반으로 다음 명령어의 PC를 예측하는 것을 의미한다. 만약 예측이 실패할 경우, 잘못된 파이프라인을 flush하고 올바른 PC를 새로 fetch해야 한다. 우리 팀은 교재의 설계를 따라 CPU를 수정하였으며, branch 명령어에 대한 결정은 ID(Instruction Decode) 단계에서 이루어진다. 이로 인해 branch 및 JAL(Jump and Link) 명령어의 misprediction 시 한 사이클의 페널티가 발생한다. 반면, JALR(Jump and Link Register) 명령어는 레지스터 값을 기반으로 점프하기 때문에 ALU(Arithmetic Logic Unit)에서 연산이 필요하며, 이는 EX(Execute) 단계에서 결정된다. 따라서 JALR 명령어의 misprediction 시 두 사이클의 페널티가 발생한다. 또한, JALR 명령어는 점프 위치가 레지스터 값을 기반으로 가변적이기 때문에 Branch Predictor와는 무관하다.

   우리 팀은 교재에 제시된 Gshare 예측 방식을 구현하였으며, BTB(Branch Target Buffer)와 PHT(Pattern History Table)는 각각 32개의 entry를 가지도록 설계하였다. PHT는 2비트 히스테리시스(hysteresis)를 사용하여 구현하였다.

3. Implementation

   먼저, branch 및 JAL 명령어에 대한 결정(예: 실제 next PC)을 ID(Instruction Decode) 단계에서 수행하도록 CPU를 수정하였다. 이는 교재의 설계도를 참고하여 다음과 같이 설계되었다. 데이터 포워딩 및 flush 조건은 아래 코드와 같이 설정하였으며, branch 및 JAL 명령어인 경우 실제 next PC를 결정한다.

```
always @(*) begin
  if (ID_branch) begin
    case (IF_ID_inst[14:12])
      3'b000: ID_bcond = (ID_rs1_data_forwarded == ID_rs2_data_forwarded);
      3'b001: ID_bcond = (ID_rs1_data_forwarded != ID_rs2_data_forwarded);
      3'b100: ID_bcond = ($signed(ID_rs1_data_forwarded) < $signed(ID_rs2_data_forwarded));
      3'b101: ID_bcond = ($signed(ID_rs1_data_forwarded) >= $signed(ID_rs2_data_forwarded));
      default: ID_bcond = 0;
    endcase
  end
  else begin
    ID_bcond = 0;
  end
end
wire [31:0] ID_next_pc = (ID_is_jal || ID_bcond) ? (IF_ID_pc + ID_imm_gen_out) : IF_ID_pc + 4;
wire ID_jal_branch_flush = (IF_ID_inst != 0) && (ID_next_pc != IF_pc) && !is_stall;
```

EX(Execute) 단계에서 JALR(Jump and Link Register) 명령어가 수행되는 경우, 다음 코드와 같이 next PC를 결정하고 flush 여부를 설정한다. 유의할 점은, branch 또는 JAL 직후에는 단순히 EX_next_pc != IF_ID_pc만으로는 조건이 충분하지 않으므로, ID_EX_is_jal || ID_EX_branch라는 추가적인 조건이 필요하다.

```
  wire [31:0] EX_next_pc = ID_EX_is_jalr ? EX_alu_result : ID_EX_pc + 4;
  wire EX_jalr_flush = (ID_EX_inst != 0) && (EX_next_pc != IF_ID_pc)
&& !(ID_EX_is_jal || ID_EX_branch);
```

이후, 실제로 next PC는 다음 코드와 같이 결정된다. 먼저 파이프라인에 가장 먼저 진입한 EX 단계를 확인한 뒤, ID 단계를 검토하고, 이후 stall 여부를 확인한 후 predictor가 예측한 PC를 최종적으로 이용한다. Flush 신호는 해당 단계에서 결정된 PC가 사용된다는 것을 의미하며, 이를 통해 파이프라인의 동작을 제어한다.

```
  always @(*) begin
    if (EX_jalr_flush) begin
      next_pc = EX_next_pc;
    end
    else if (ID_jal_branch_flush) begin
      next_pc = ID_next_pc;
    end
    else if (is_stall) begin
      next_pc = IF_pc;
    end
    else begin
      next_pc = predicted_pc;
    end
  end
```

Flush 신호는 다음과 같이 각 파이프라인 업데이트 시 실제 flush에도 사용된다. 이때, branch 및 JAL에 의한 flush는 ID 단계만 처리하면 되지만, JALR에 의한 flush는 EX 단계도 처리해야 한다.

```
  // Update IF/ID pipeline registers here
  always @(posedge clk) begin
    if (reset || ID_jal_branch_flush || EX_jalr_flush) begin
      IF_ID_inst <= 0;
      IF_ID_pc <= 0;
    end
    else if (!is_stall) begin
      IF_ID_inst <= IF_inst;   // will be used in ID stage
      IF_ID_pc <= IF_pc;       // will be used in ID stage
    end
  end
  // Update ID/EX pipeline registers here
  always @(posedge clk) begin
    if (reset || is_stall || EX_jalr_flush) begin
      // From the control unit
      ID_EX_alu_op <= 0;       // will be used in EX stage
      ID_EX_alu_src <= 0;      // will be used in EX stage
       (중략)
```

위와 같은 수정들을 통해 CPU는 branch 명령어를 효율적으로 처리할 수 있으며, branch prediction이 잘못된 경우에도 flush를 통해 명령어들이 올바르게 실행될 수 있도록 보장한다. 이러한 구현 방식에서 branch predictor가 next PC를 항상 0으로 설정하거나, 항상 PC + 100 등으로 예측하더라도 CPU는 문제없이 동작한다.

이후, branch predictor로 Gshare를 구현하였다. Design 섹션에서 언급한 바와 같이, 32개의 entry를 갖는 BTB(Branch Target Buffer)와 PHT(Pattern History Table)를 구현하였다. PHT는 2비트 히스테리시스를 사용하며, 5비트 BHSR(Branch History Shift Register)도 구현하였다. 인터페이스 및 내부 레지스터, 와이어, 변수는 다음 코드와 같이 설계되었다. 또한, 교재의 구현에 valid bit을 추가하여 해당 BTB가 실제로 유효한 값인지 확인하도록 하였다. 이는 초기 BTB 값을 0으로 설정했을 때, 그것이 실제 주소 0인지 아니면 초기 값인지를 구분하기 위함이다.

```
module Gshare(
    input clk,
    input reset,
    input is_stall,
    input [31:0] IF_pc,
    input ID_branch,
    input ID_bcond,
    input [31:0] IF_ID_pc,
    input [31:0] ID_next_pc,
    output reg [31:0] predicted_pc
);
    reg [24:0] TagTable[31:0];
    reg valid[31:0];
    reg [31:0] BTB[31:0];
    reg [1:0] PHT[31:0];
    reg [4:0] BHSR;
    integer i;
    wire [31:7] Tag = IF_pc[31:7];
    wire [31:7] update_Tag = IF_ID_pc[31:7];
    wire [4:0] BTB_idx = IF_pc[6:2];
    wire [4:0] update_BTB_idx = IF_ID_pc[6:2];
    wire [4:0] PHT_idx = BTB_idx ^ BHSR;
    wire [4:0] update_PHT_idx = update_BTB_idx ^ BHSR;
```

다음 PC를 예측하는 과정은 비동기적으로 이루어진다. Tag와 valid bit을 먼저 확인한 뒤, PHT의 판단에 따라 taken으로 예측된 경우 BTB의 PC를, 그렇지 않을 경우 PC + 4를 next PC로 예측한다.

```
    always @(*) begin
        if ((TagTable[BTB_idx] == Tag) && valid[BTB_idx] && (PHT[PHT_idx] >=
2)) begin
            predicted_pc = BTB[BTB_idx];
        end
        else begin
            predicted_pc = IF_pc + 4;
        end
    end
```

Gshare의 각 요소는 다음과 같이 초기화된다. PHT는 2(약한 taken)로 초기화하였다.

```
        if (reset) begin
            for (i=0; i<32; i=i+1) begin
                TagTable[i] <= 0;
                valid[i] <= 0;
                BTB[i] <= 0;
                PHT[i] <= 2;
            end
            BHSR <= 0;
        end
```

각 posedge clk마다 branch 명령어가 수행되고, stall 상태가 아닌 유효한 상태라면 BHSR, BTB, PHT 등이 업데이트된다. 이때, ID_branch와 ID_bcond는 각각 ID_jal과 OR 연산으로 전달된다. 즉, JAL 명령어는 성공한 branch로 간주된다. PHT는 2비트 히스테리시스를 사용하므로, 0은 강한 not taken, 1은 약한 not taken, 2는 약한 taken, 3은 강한 taken을 의미한다. 기본적으로 branch 명령어는 taken일 가능성이 높기 때문에, PHT 초기화를 2로 설정하였다.

```
        else begin
            if (ID_branch && !is_stall) begin
                BHSR <= {ID_bcond, BHSR[4:1]};
                if (ID_bcond) begin
                    TagTable[update_BTB_idx] <= update_Tag;
                    valid[update_BTB_idx] <= 1;
                    BTB[update_BTB_idx] <= ID_next_pc;
                    case (PHT[update_PHT_idx])
                        0: PHT[update_PHT_idx] <= 1;
                        1: PHT[update_PHT_idx] <= 3;
                        2: PHT[update_PHT_idx] <= 3;
                        3: PHT[update_PHT_idx] <= 3;
                        default: PHT[update_PHT_idx] <= 2;
                    endcase
                end
                else begin
                    case (PHT[update_PHT_idx])
                        0: PHT[update_PHT_idx] <= 0;
                        1: PHT[update_PHT_idx] <= 0;
                        2: PHT[update_PHT_idx] <= 0;
                        3: PHT[update_PHT_idx] <= 2;
                        default: PHT[update_PHT_idx] <= 2;
                    endcase
                end
            end
        end
```

4. Discussion

위와 같이 파이프라인 CPU를 구현한 뒤, recursive_mem.txt 테스트를 실행한 결과 1089 사이클이 소요되었다. 아래의 next PC를 예측하는 코드에서 PHT와 관련된 조건을 단순히 1 또는 0으로 변경하면, Always Taken 또는 Always Not Taken 방식으로 동작하도록 설정할 수 있다. 이를 통해 사이클 수를 비교한 결과, Always Taken의 경우 1090 사이클, Always Not Taken의 경우 1146 사이클이 소요되었다. 이 결과는 branch 명령어가 taken되는 빈도가 더 높다는 것을 의미한다. 또한, 2비트 상태 머신(state machine)을 도입하여 PHT(+BHSR)를 사용했을 때에는 1 사이클이 줄어들었다. 주어진 테스트 케이스가 제한적이기 때문에 이 차이를 통계적으로 유의미하다고 보기는 어려울 수 있다. 그러나 사이클 수가 줄어든 점에서 PHT를 활용한 branch prediction의 긍정적인 효과를 확인할 수 있었다.

5. Conclusion

본 과제를 통해 Pipelined CPU의 설계와 구현 과정을 학습하며, 파이프라인 구조의 동작 원리와 성능 향상 메커니즘을 이해할 수 있었다. 데이터 해저드 해결을 위해 Hazard Detection Unit과 Data Forwarding을 구현하고, branch와 JAL 명령어를 ID 단계에서 처리하여 misprediction 비용을 줄였다. 또한, Gshare를 활용한 branch prediction을 통해 Always Taken 및 Always Not Taken 방식과 비교하며 branch 명령어의 taken 비율을 확인할 수 있었다. 이를 통해 Pipelined CPU 설계의 핵심 개념과 성능 최적화 기법을 익힐 수 있었다.