1. Introduction

본 과제의 목적은 Verilog를 이용해 간단한 Pipelined CPU를 설계하고 구현하며, 이를 통해 Pipelined CPU의 동작 원리를 이해하는 데 있다. Pipelined CPU는 여러 명령어를 병렬로 처리하여 성능을 향상시키는 구조로, 본 과제에서는 컨트롤 플로우가 없는 단순한 상황에 대한 Pipelined CPU를 구현한다. 이를 통해 파이프라인 구조의 기본적인 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 Unit과 Forwarding Unit을 구현하여 데이터 및 제어 흐름의 안정성을 확보하였다. Pipelined CPU의 동작 원리는 다음과 같다.

1) Instruction Fetch (IF) Stage

프로그램 카운터(PC)를 이용해 명령어 메모리에서 명령어를 읽어오는 단계이다. 이 단계에서는 다음 명령어를 가져오기 위해 PC를 업데이트하며, IF/ID 레지스터를 통해 다음 단계로 데이터를 전달한다.

2) Instruction Decode (ID) Stage

IF 단계에서 전달된 명령어를 해석하며, 레지스터 파일로부터 필요한 데이터를 읽어온다. 또한, 제어 신호를 생성하여 이후 단계에서 수행할 작업을 지정한다. 이 과정에서 데이터 종속성을 확인하기 위해 Hazard Detection Unit이 동작하며, 필요시 파이프라인을 스톨하거나 거쳐야 할 데이터를 처리한다. 결과는 ID/EX 레지스터를 통해 전달된다.

3) Execution (EX) Stage

ALU(산술 논리 연산 장치)를 이용해 명령어의 연산을 수행하는 단계이다. Forwarding Unit을 통해 이전 단계의 종속 데이터를 해결하며, EX/MEM 레지스터를 통해 연산 결과 및 제어 신호를 다음 단계로 전달한다.

4) Memory Access (MEM) Stage

메모리 접근이 필요한 명령어에 대해 데이터를 읽거나 쓰는 작업을 수행한다. 이 단계에서는 EX 단계에서 전달된 연산 결과를 기반으로 메모리 접근이 이루어지며, MEM/WB 레지스터에 결과를 저장한다.

5) Write Back (WB) Stage

MEM 단계에서 전달된 데이터를 레지스터 파일에 기록한다. 이 단계에서는 연산 결과나 메모리에서 읽어온 데이터를 최종적으로 프로세서의 상태에 반영한다.

Pipelined 구조의 핵심은 각 명령어가 서로 다른 단계에서 동시에 실행될 수 있도록 하는 것이다. 이를 위해 각 단계 사이에 **Pipeline Register(IF/ID, ID/EX, EX/MEM, MEM/WB)**를 도입하여 명령어의 상태를

저장하고, 병렬 처리를 가능하게 한다. 또한, 데이터 및 제어 흐름 문제를 해결하기 위해 Hazard Detection Unit과 Forwarding Unit을 설계하여 데이터 종속성을 관리하고, 파이프라인의 효율성을 높였다.

3. Implementation

파이프라인 CPU에서 분기(branch) 상황을 고려하지 않는 경우, 데이터 해저드(Data Hazard) 중 Read After Write (RAW) 해저드만을 처리하면 된다. RAW 해저드는 이전 명령어의 결과가 이후 명령어에서 필요할 때 발생하며, 이 경우 파이프라인을 스톨(stall) 시켜 데이터가 준비될 수 있도록 해야 한다. 한편, Data Forwarding을 구현하면 스톨 사이클을 줄일 수 있으며, 이번 과제에서는 Data Forwarding을 포함한 구조를 구현하였다. Data Forwarding을 구현했을 경우와 그렇지 않을 경우, 해저드를 감지하는 방식이 달라진다. 본 과제에서는 Data Forwarding을 적용한 설계를 기준으로 하여 해저드를 감지하였다. 다음은 주요 동작 방식이다.

1) 스톨 발생 조건

스톨은 load 명령어가 EX 단계에 위치하고, 메모리에서 데이터를 읽어오는 중(mem_read가 활성화된 상태)일 때 발생한다. 이 경우, rs1 및 rs2 두 레지스터가 이후 명령어에서 사용될 예정이라면 파이프라인을 스톨시켜야 한다.

2) use_rs1 및 use_rs2 조건

교재에서는 use_rs1 및 use_rs2 조건을 확인하여 명령어가 실제로 해당 레지스터를 사용하는지 검사하도록 제안하였다. 그러나 과제의 테스트 케이스를 통과하기 위해서는 이 조건을 구현하지 않아도 무방하였다.

아래는 위 설명을 바탕으로 구현된 Hazard Detection Unit의 코드이다:

```
module HazardDetectionUnit(
    input [4:0] rs1_id,
    input [4:0] rs2_id,
    input [4:0] rd_ex,
    input reg_write_ex,
    input mem_read_ex,
    input is_ecall,
    output reg is_stall);

    // TODO: Maybe we neet to check if instruction is really use rs1 or rs2
    wire use_rs1 = (rs1_id != 0);
    wire use_rs2 = (rs2_id != 0);

    wire is_forwarding_stall = ((rs1_id == rd_ex) && use_rs1 || (rs2_id == 
rd_ex) && use_rs2) && mem_read_ex;
    wire is_ecall_stall = is_ecall && rd_ex == 17 && reg_write_ex;
    assign is_stall = is_forwarding_stall || is_ecall_stall;

endmodule
```

Data Forwarding은 파이프라인 CPU에서 데이터 해저드를 줄이기 위해 이전 명령어의 결과를 바로 다음 명령어로 전달하는 기법이다. 데이터를 가져오는 소스는 크게 세 가지로 나뉘며, MEM 스테이지, WB 스테이지, 그리고 레지스터 파일이다. 이 세 가지 소스는 최신 데이터를 기준으로 우선순위가 부여되며,

순서가 중요하다. 다음은 데이터 소스를 판단하는 방법과 각 조건에 대한 설명이다.

1) MEM 스테이지에서 데이터 가져오기

EX 스테이지의 레지스터가 0번이 아니고, EX 스테이지의 레지스터가 MEM 스테이지의 rd와 동일하며, MEM 스테이지의 reg_write 신호가 활성화된 경우, MEM 스테이지가 최신 데이터를 가지고 있으므로, MEM 스테이지에서 데이터를 가져온다.

2) WB 스테이지에서 데이터 가져오기

MEM 스테이지의 조건을 만족하지 않으면서, EX 스테이지의 레지스터가 0번이 아니고, EX 스테이지의 레지스터가 WB 스테이지의 rd와 동일하며, WB 스테이지의 reg_write 신호가 활성화된 경우, WB 스테이지에서 데이터를 가져온다.

3) 레지스터 파일에서 데이터 가져오기

위 두 조건에 해당하지 않는 경우, 레지스터 파일에서 데이터를 가져온다.

위 과정을 rs1과 rs2 두 레지스터에 대해 각각 독립적으로 처리하면 된다. 이를 구현한 코드는 아래와 같다.

```
module ForwardingUnit (
    input [4:0] rs1_ex,
    input [4:0] rs2_ex,
    input [4:0] rd_mem,
    input [4:0] rd_wb,
    input reg_write_mem,
    input reg_write_wb,
    output reg [1:0] ForwardA,
    output reg [1:0] ForwardB);
    always @(*) begin
        if (rs1_ex != 0 && rs1_ex == rd_mem && reg_write_mem) begin
            ForwardA = 1;
        end else if (rs1_ex != 0 && rs1_ex == rd_wb && reg_write_wb) begin
            ForwardA = 2;
        end else begin
            ForwardA = 0;
        end
        if (rs2_ex != 0 && rs2_ex == rd_mem && reg_write_mem) begin
            ForwardB = 1;
        end else if (rs2_ex != 0 && rs2_ex == rd_wb && reg_write_wb) begin
            ForwardB = 2;
        end else begin
            ForwardB = 0;
        end
    end
endmodule
```

4. Discussion

위와 같은 방식으로 Hazard Detection Unit을 설계함으로써 파이프라인의 RAW 해저드를 감지하고, 데이터 충돌을 방지할 수 있다. 또한 Data Forwarding을 활용하여 스톨 사이클을 줄임으로써 파이프라인의 효율성을 극대화하였다.

Data Forwarding은 파이프라인에서 데이터를 효율적으로 전달하여 RAW 해저드를 줄이고 성능을 향상시키는 데 중요한 역할을 한다. MEM 스테이지와 WB 스테이지의 데이터를 우선적으로 참조하며, 필요한 경우 레지스터 파일에서 데이터를 가져온다. 위 구현 방식은 이러한 동작 원리를 충실히 반영하며, 각 조건을 명확히 정의하여 파이프라인의 안정성과 효율성을 보장한다.

직접 구현한 Single Cycle CPU와 Pipelined CPU로 Non-control flow input file을 실행한 결과는 아래 사진과 같다. Single Cycle CPU는 39사이클이 소요되었고, Pipelined CPU는 46사이클이 소요되었다. 하지만 이러한 결과가 Single Cycle CPU가 더 빠르다는 것을 의미하지는 않는다. Single Cycle CPU의 경우, 모든 명령어가 가장 오래 걸리는 명령어의 실행 시간에 맞추어 실행되기 때문에 사이클 수가 적다고 하더라도 실제 수행 시간은 길어질 수 있다. 반면, Pipelined CPU는 각 명령어를 서로 다른 파이프라인 단계에서 동시에 처리하며, 한 사이클의 시간이 훨씬 짧다. 즉, 실제 동작 시간을 비교하면 Pipelined CPU가 더 적은 시간을 소요하며, 효율적으로 작동한다. 이는 파이프라인 구조가 병렬 처리를 통해 명령어 처리 속도를 높이는 데 최적화되어 있기 때문이다.

```
### SIMULATING ###
TEST END
SIM TIME : 80
TOTAL CYCLE : 39
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 0000000a
11 0000003f
12 fffffff1
13 0000002f
14 0000000e
15 00000021
16 0000000a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
```

```
### SIMULATING ###
TEST END
SIM TIME : 94
TOTAL CYCLE : 46 (Answer : 46)
FINAL REGISTER OUTPUT
 0 00000000 (Answer : 00000000)
 1 00000000 (Answer : 00000000)
 2 00002ffc (Answer : 00002ffc)
 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 fffffff1 (Answer : fffffff1)
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)
```

## 5. Conclusion

본 과제를 통해 Pipelined CPU의 설계와 구현 과정을 학습하였으며, 파이프라인 구조의 동작 원리와 성능 향상 메커니즘을 이해할 수 있었다. 특히, 데이터 해저드와 같은 복잡한 문제를 해결하기 위해 Hazard Detection Unit과 Data Forwarding을 설계 및 활용하였으며, 이를 통해 병렬 처리의 효율성을 극대화하고 CPU 성능을 효과적으로 향상시킬 수 있음을 확인하였다.