# 컴퓨터구조및모바일프로세서 과제3 보고서


과목: 컴퓨터구조및모바일프로세서 2분반
학과: 모바일시스템공학과
학번: 32224020
이름: 전은지
**Rest Freedays: 0

# Index

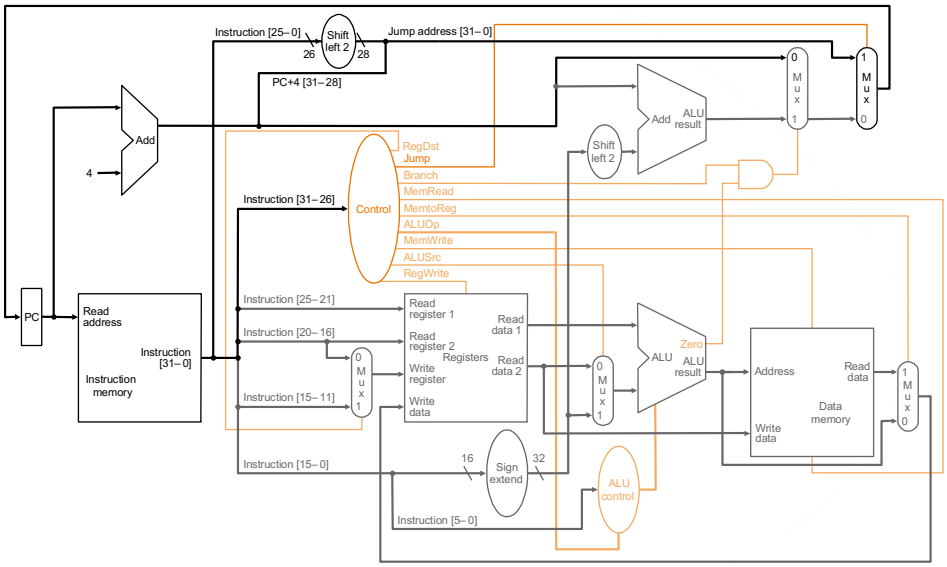
# I. Introduction

이번 프로젝트의 목표는 파이프라인 단계로 향상된 MIPS 시뮬레이터를 구현하고 single cycle로 구현된 시뮬레이터와 효율을 비교하는 것이다. 이론적으로, 5단계 파이프라인 실행을 통해 CPU 클럭 사이클 시간을 1/5로 줄일 수 있다. 파이프라인 구현 시 생길 수 있는 위험과 종속성 같은 문제들을 해결하여 5단계 파이프라인을 갖춘 MIPS 프로세서를 만드는 것을 목적으로 한다.

# II. Body

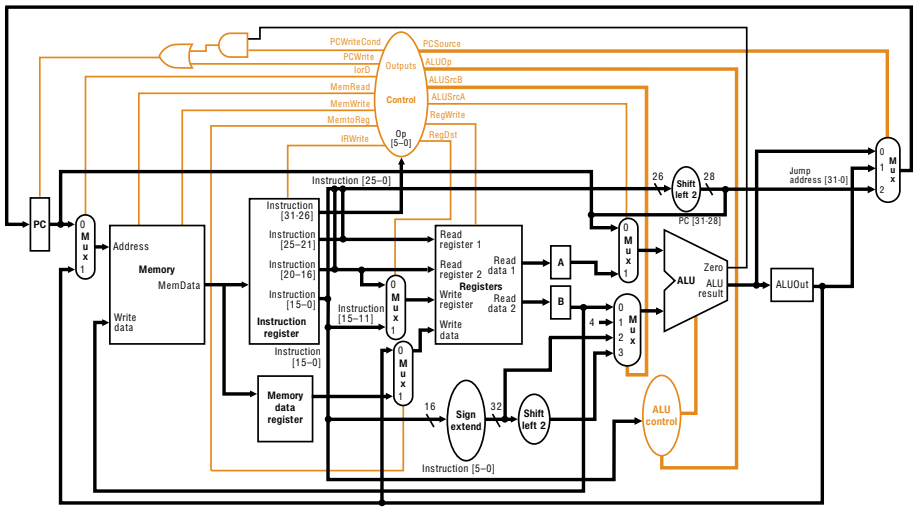
## 1. Background Information

### 1.1 Single Cycle




싱글 사이클 프로세서는 모든 명령어가 하나의 클럭 사이클 동안에 실행되는 프로세서를 의미한다. 명령어의 Fetch, Decode, Execute, Memory access, Write-back 등의 모든 단계가 하나의 사이클 내에서 완료됨을 의미한다. 싱글 사이클 프로세서는 단순한 설계를 통해 예측 가능한 성능을 제공하지만, 모든 명령어가 동일한 사이클내에서 진행되기 때문에 실행 시간이 오래 걸리고 복잡한 명령어의 경우 클럭의 주기를 길게 만들기 때문에 프로세서의 성능을 저하시키므로 비효율적인 시스템이다.

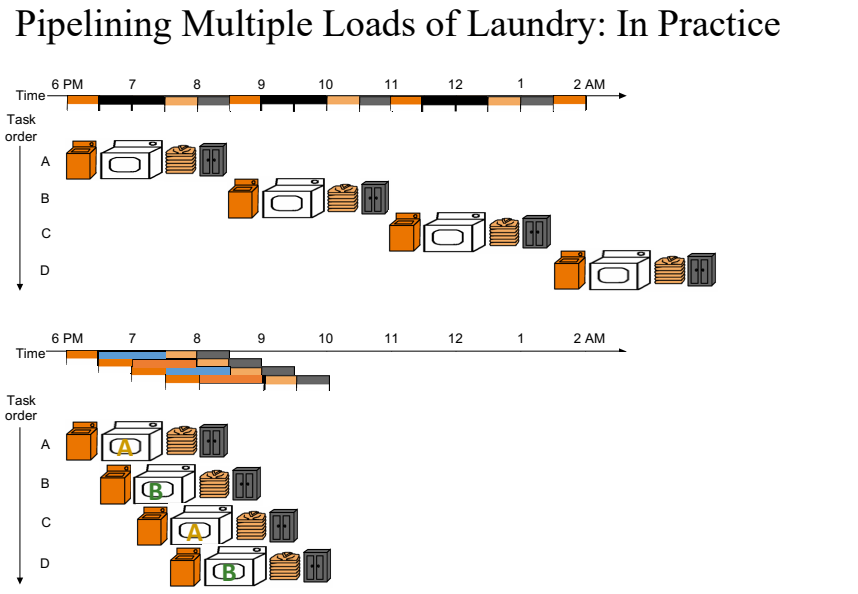
### 1.2 Multi Cycle




멀티 사이클 프로세서는 명령어를 여러 개의 클럭 사이클에 걸쳐 실행하는 프로세서이다. 이 프

로세서는 각 명령어를 실행하는 데 필요한 단계를 여러 클럭 사이클에 걸쳐 분할하여 순차적으로 처리한다. 싱글 사이클과 마찬가지로 명령어의 Fetch, Decode, Execute, Memory access, Write-back 등의 모든 단계를 진행하지만, 멀티 사이클은 각 단계가 별도의 클럭 사이클을 차지한다는 점에서 싱글 사이클과 차이가 난다. 각 명령어가 필요한 만큼만 클럭 사이클을 사용하기 때문에 명령어마다 소요 시간이 최적화될 수 있고, 명령어 사이클을 나눠서 처리하기 때문에 싱글 사이클에 비해 복잡도가 감소한다는 장점이 있다. 그러나 각 사이클마다 제어 신호를 다르게 생성해야 하므로 제어 회로가 복잡해질 수 있다는 단점이 있다.

## 1.3 Pipeline




파이프라인이란 CPU의 성능을 향상시키기 위해 명령어의 실행 단계를 여러 개의 연속적인 처리 단계로 분할하여 각 단계를 병렬로 처리하는 기술이다. 각 단계에서 명령어를 처리함으로써 CPU가 여러 명령어들을 동시에 실행할 수 있게 하기 때문에 처리 속도가 높아진다.

위 사진을 파이프라인 처리 과정의 예시로 들 수 있다. 1번 과정의 경우 하나의 과정이 모두 끝난 후에 다음 과정을 시작하기 때문에 처리 시간이 오래 걸린다. 그러나 2번 과정의 경우 하나의 과정이 끝나기 전에 다음 과정을 시작하고 한 번에 여러 개의 스테이지를 병렬로 처리하기 때문에 1번 과정보다 훨씬 적은 시간이 소요되는 것을 확인할 수 있다.

| Instruction | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9 |
|---|---|---|---|---|---|---|---|---|---|
| Inst 1 | IF | ID | EX | MEM | WB | | | | |
| Inst 2 | | IF | ID | EX | MEM | WB | | | |
| Inst 3 | | | IF | ID | EX | MEM | WB | | |
| Inst 4 | | | | IF | ID | EX | MEM | WB | |
| Inst 5 | | | | | IF | ID | EX | MEM | WB |

파이프라인의 명령어 처리 순서를 시각화 하면 다음과 같다. 여러 개의 스테이지를 병렬로 처리하기 위하여 파이프라인은 래치를 필수로 한다. 래치란 각 파이프라인 단계 사이에 위치한 레지

스터로 각 단계의 데이터를 저장하여 다음 단계로 전달하고, 파이프라인의 각 단계가 독립적으로 작동할 수 있게 하는 역할을 하여 파이프라인을 제어한다. 래치는 IF/ID 래치, ID/EX 래치, EX/MEM 래치, MEM/WB 래치가 있다. 각각의 래치는 이전 스테이지의 데이터를 저장한 후 이후 스테이지로 전달한다.

## 1.4 Data Dependency

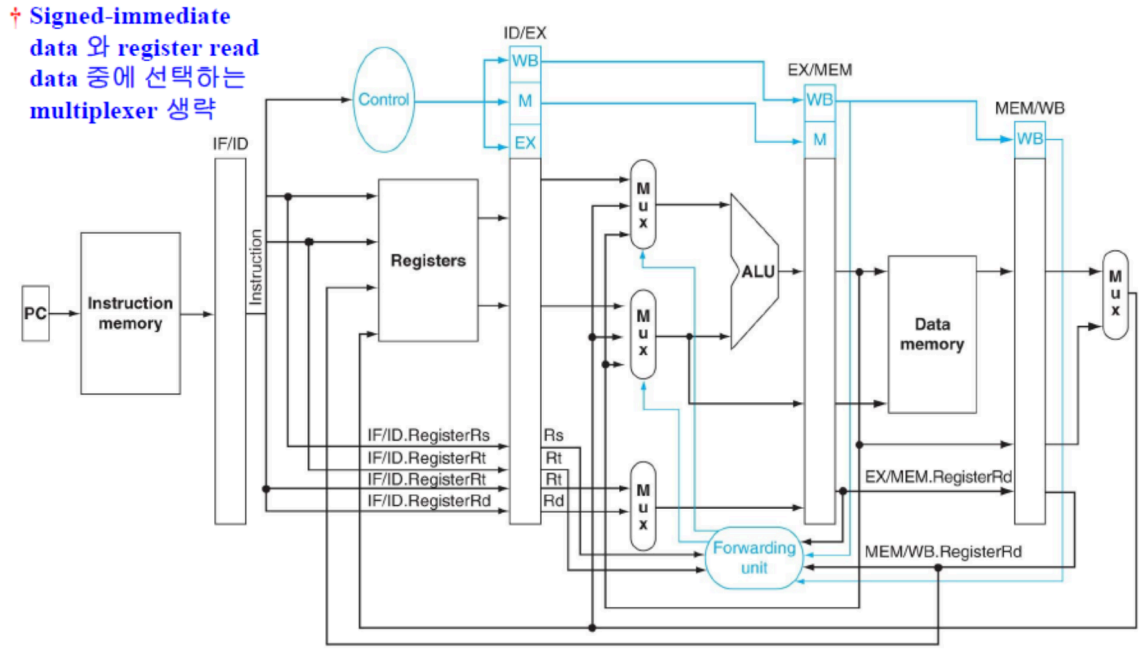
데이터 종속성은 컴퓨터 프로그램에서 명령어들이 동일한 데이터 자원을 공유할 때 발생하는 관계를 의미한다. 프로그램의 명령어 실행 순서와 결과에 중요한 영향을 미치며, 특히 파이프라인 프로세서에서 성능을 최적화하는 데 문제가 된다. 데이터 종속성은 크게 3가지 유형으로 분류된다. 먼저, RAW(Read After Wrtie) 유형이다. 이 유형은 어떤 명령어가 데이터를 읽기 전에 이전 명령어가 데이터를 써야 하는 경우 발생한다. 가장 일반적인 형태의 데이터 종속성으로 포워딩(Forwarding)이나 스톨(Stall)을 통해 해결한다. 두번째 유형은 WAR(Write After Read)이다. WAR 유형은 어떤 명령어가 데이터를 쓰기 전에 이전 명령어가 데이터를 읽어야 하는 경우 발생한다. 이는 주로 쓰기 연산이 읽기 연산보다 늦게 이루어질 때 발생한다. 이 유형은 레지스터 리네이밍을 통해 해결할 수 있다. 마지막 유형은 WAW(Write After Write)이다. WAW 유형은 어떤 명령어가 데이터를 쓰기 전에 이전 명령어가 데이터를 써야 하는 경우 발생한다. 이는 주로 두 쓰기 연산이 동시에 이루어질 때 발생한다. 이 유형 역시 레지스터 리네이밍을 통해 해결할 수 있다. 레지스터 리네이밍의 경우 동일한 물리적 레지스터를 다른 이름으로 사용하는 방법이므로 많은 설명이 필요하지 않다. 다음으로 알아볼 것은 포워딩과 스톨이다.

### 1.4.1 Forwarding

포워딩은 데이터 하자드(Data Hazard)를 해결하기 위해 사용되는 기술로, 이전 명령어가 생성한 데이터를 현재 명령어가 필요로 할 때, 이를 중간 레지스터를 통해 즉시 전달하는 방법이다. 이를 통해 데이터가 필요한 시점에 전달되어 파이프라인의 진행을 방해하지 않고 연속성을 유지할 수 있다. 데이터 포워딩을 하기 위해서는 데이터 하자드의 존재를 판단해야 하고 레지스터 넘버를 비교해야 한다. 비교 조건은 다음과 같다.

① EX/MEM.RE == ID/EX.RS
② EX/MEM.RE == ID/EX.RT
③ MEM/WB.RD == ID/EX.RS
④ MEM/WB.RD == ID/EX.RT

레지스터 $0은 포워딩 하면 안 되기 때문에 EX/MEM.RD ≠ 0 또는 MEM/WB.RD ≠ 0 이어야 한다. 또한, 앞선 명령어가 레지스터에 값을 작성하는 명령어이어야 하므로 EX/MEM 또는 MEM/WB 단계에서 RegWrite 신호가 활성화 상태여야 한다.




위 그림은 데이터 포워딩 유닛이 어떻게 데이터 하자드를 해결하는지 설명한다. 그러나 포워딩만으로 모든 데이터 하자드를 해결할 수 있는 것은 아니다. Load 명령어 바로 다음에 오는 명령어가 Load 명령어의 결과를 요구하는 경우 Load-Use 데이터 하자드가 발생하게 되는데 이때 사용되는 것이 스톨이다.

### 1.4.2 Stall

스톨(Stall)은 파이프라인 CPU에서 데이터 하자드(Data Hazard) 또는 제어 하자드(Control Hazard)를 해결하기 위해 사용되는 중요한 기술이다. 스톨은 특정 명령어의 실행을 일시적으로 멈추어, 필요한 데이터나 제어 신호가 준비될 때까지 기다리는 방법을 의미한다. 스톨은 파이프라인의 각 단계가 독립적으로 작동할 수 있도록 하여, 데이터 불일치와 같은 문제를 방지한다. 스톨은 특정 단계에서 명령어의 실행을 멈추게 하고, 그 단계의 처리가 완료될 때까지 기다리도록 한다. 스톨이 발생하면, 파이프라인의 나머지 단계는 영향을 받지 않고 계속해서 실행된다. 스톨은 주로 제어 유닛에 의해 제어되며, 필요한 경우 파이프라인 레지스터에 0을 기록하여 명령어의 실행을 일시 중지시킨다. Load-Use 데이터 하자드는 Load 명령어가 메모리에서 데이터를 읽어오기 전에 그 데이터를 사용하는 명령어가 파이프라인에 존재할 때 발생하는 데이터 하자드이다. 제어 유닛이 ID 단계에서 데이터 의존성을 감지하고 스톨 신호를 생성하여 파이프라인 레지스터와 PC에 전송한다. 스톨 신호를 받은 단계는 명령어의 실행을 중지하고, 해당 단계의 파이프라인 레지스터에 0을 기록한다. 필요한 데이터가 준비되면, 스톨 신호를 해제하고 파이프라인의 실행을 재개하여 데이터 해자드를 해결할 수 있다.

대부분의 현대적인 파이프라인 CPU는 포워딩과 스톨을 함께 사용하여 최대한 많은 데이터 하자드를 해결한다. 포워딩은 가능한 모든 경우에 사용하여 데이터 하자드를 최소화하고, 포워딩으로 해결할 수 없는 하자드는 스톨을 사용하여 해결한다. 이렇게 하면 파이프라인의 효율성을 극대화할 수 있다.

## 1.5 Control Dependency

컨트롤 종속성(은 프로그램 실행 흐름에서 명령어 간의 제어 흐름 관계를 나타내는 개념으로, 주로 분기 명령어와 관련이 있다. 분기 명령어의 결과에 따라 이후 명령어의 실행 여부가 결정되는 상황에서 발생한다. 이러한 종속성은 파이프라인 프로세서에서 중요한 문제를 야기하며, 명령어의 순차적 실행을 방해하고 성능 저하를 초래할 수 있다. 분기 명령어가 파이프라인에 들어오면, 그 결과가 나올 때까지 다음 명령어의 실행 여부를 결정할 수 없기 때문에 문제가 발생한다. 이는 파이프라인의 효율을 저하시킬 수 있으며, 다음과 같은 주요 문제를 일으킨다. 먼저, 파이프라인 버블 문제가 있다. 이 문제는 분기 명령어의 결과를 기다리는 동안 파이프라인에 비어 있는 단계가 생기게 되고 결국 파이프라인의 효율을 떨어뜨린다. 다음은 파이프라인 플러시이다. 잘못된 분기 예측으로 인해 잘못된 명령어가 파이프라인에 들어가면, 이를 제거하고 올바른 명령어로 대체해야 한다. 이 과정에서 파이프라인이 일시적으로 정지하게 되어 성능 저하를 초래한다. 마지막으로 명령어 지연)이 있다. 분기 명령어의 결과가 확정될 때까지 후속 명령어의 실행이 지연된다. 이는 프로그램의 전체 실행 시간을 늘리는 문제를 초래한다. 이러한 컨트롤 종속성을 해결하기 위해 다양한 방법들이 사용된다. 첫 번째로, 분기 예측(Branch Prediction)이 있다. 이는 하드웨어 또는 소프트웨어가 분기 명령어의 결과를 예측하여 파이프라인의 다음 명령어를 결정하는 방법으로, 예측이 맞으면 성능이 향상되지만 틀리면 파이프라인 플러시가 발생하여 성능 저하가 일어난다. 분기 예측은 정적 분기 예측과 동적 분기 예측으로 나뉜다. 정적 분기 예측은 컴파일러 또는 프로그램이 분기 예측을 미리 설정하는 방법으로, 항상 분기가 발생한다고 가정하거나 발생하지 않는다고 가정하는 방식이다. 동적 분기 예측은 하드웨어가 실행 중에 분기 패턴을 학습하고 예측을 조정하는 방법으로, 일반적으로 더 높은 정확도를 제공한다. History-based Prediction은 과거의 분기 결과를 기반으로 예측하는 방법이며, Two-Level Adaptive Branch Prediction은 글로벌 분기 히스토리와 패턴 테이블을 사용하여 예측을 수행하는 방식이다. 두 번째로, 투명 실행(Speculative Execution)이 있다. 이는 분기 예측에 따라 명령어를 실행하고, 예측이 맞으면 결과를 커밋하고 틀리면 파이프라인을 플러시하는 방법으로, 성능을 크게 향상시킬 수 있지만 예측 실패 시의 오버헤드가 존재한다. 투명 실행 제어는 CPU가 예측된 경로를 따라 명령어를 실행하며, 예측이 틀렸을 때 결과를 폐기하는 방법이다. 세 번째로, 딜레이 슬롯(Delay Slot)이 있다. 이는 분기 명령어 뒤에 있는 하나 또는 여러 개의 명령어를 항상 실행되도록 하는 방법으로, 분기 명령어의 결과와 상관없이 실행되므로 컨트롤 종속성을 줄이는 데 도움이 된다. 컴파일러는 분기 명령어 뒤에 실행해도 무방한 명령어를 배치하여 파이프라인의 비효율성을 줄이는 최적화를 수행한다. 네 번째로, Predication이 있다. 이는 분기 명령어를 제거하고 조건에 따라 명령어를 실행하거나 무시하는 방법으로, 조건부 이동 명령어 등을 사용하여 분기 없이 제어 흐름을 구현할 수 있다. ARM 아키텍처의 조건부 실행은 대부분의 명령어가 조건부로 실행될 수 있어 분기 명령어의 필요성을 줄인다. 이와 같이, 분기 예측, 투명 실행, 딜레이 슬롯, Predication 등의 기법들은 파이프라인의 효율성을 높이고 전체 시스템 성능을 향상시키기 위해 컨트롤 종속성을 효과적으로 관리하는 방법들이다.

## 2. Program Description

```c
typedef struct {
    uint32_t instruction;
    uint32_t pc;
} IF_ID;

typedef struct {
    uint32_t instruction;
    uint32_t pc;
    uint32_t rs;
    uint32_t rt;
    uint32_t rd;
    int16_t immediate;
    uint32_t address;
    uint32_t reg_rs_value;
    uint32_t reg_rt_value;
} ID_EX;
```

```c
typedef struct {
    uint32_t instruction;
    uint32_t pc;
    uint32_t alu_result;
    uint32_t rt;
    uint32_t rd;
    uint32_t reg_rt_value;
} EX_MEM;

typedef struct {
    uint32_t instruction;
    uint32_t pc;
    uint32_t mem_data;
    uint32_t alu_result;
    uint32_t rd;
} MEM_WB;
```

먼저, IF_ID, ID_EX, EX_MEM, MEM_WB의 네 가지 파이프라인 레지스터 구조체를 정의한다. IF_ID 구조체는 fetch 단계와 decode 단계 사이의 데이터를 저장하며, 명령어와 프로그램 카운터를 포함한다. ID_EX 구조체는 decode 단계와 execute 단계 사이의 데이터를 저장하며, 명령어, 프로그램 카운터, 레지스터 번호(rs, rt, rd), 즉시 값(immediate), 주소(address), 레지스터 값(rs와 rt의 값)을 포함한다. EX_MEM 구조체는 execute 단계와 memory access 단계 사이의 데이터를 저장하며, 명령어, 프로그램 카운터, ALU 결과, rt, rd, rt의 값을 포함한다. MEM_WB 구조체는 memory access 단계와 write-back 단계 사이의 데이터를 저장하며, 명령어, 프로그램 카운터, 메모리 데이터, ALU 결과, rd를 포함한다. 모든 레지스터를 0으로 초기화하였다.

이제 각각의 단계를 살펴보도록 하겠다. Fetch 함수는 프로그램 카운터가 메모리 크기를 초과하지 않으면 명령어 메모리에서 명령어를 읽어와 if_id 레지스터에 저장하고, 프로그램 카운터를 증가시키며, 명령어 수를 증가시킨다. 명령어와 프로그램 카운터 값을 출력한다. Decode 함수는 if_id 레지스터의 명령어를 읽어와 id_ex 레지스터에 저장하고, 명령어의 필드(rs, rt, rd, immediate, address)를 추출하여 저장하며, 해당 레지스터 값도 저장한다. Eexecute 함수는 id_ex 레지스터의 명령어를 읽어와 명령어의 타입과 opcode를 확인하고, 각 명령어에 맞는 연산을 수행한다. R-type 명령어의 경우, funct에 따라 연산을 수행하고, ALU 결과를 ex_mem 레지스터에 저장한다. I-type 및 J-type 명령어에 대해서도 적절한 연산을 수행하고, pc와 관련된 변수들을 업데이트한다. Mem_access 함수는 ex_mem 레지스터의 명령어를 읽어와 LW와 SW에 대해 메모리에서 데이터를 읽거나 메모리에 데이터를 쓴다. 읽은 데이터는 mem_wb 레지스터에 저장하고, 주소의 유효성을 검사하여 오류를 방지한다. Write_back 함수는 mem_wb 레지스터의 명령어를 읽어와 연산 결과 또는 메모리 데이터를 rd에 write-back 한다.

```
void forward() {
    // Forwarding from MEM stage to EX stage
    if (ex_mem.rd != 0) {
        if (ex_mem.rd == id_ex.rs) {
            id_ex.reg_rs_value = ex_mem.alu_result;
        }
        if (ex_mem.rd == id_ex.rt) {
            id_ex.reg_rt_value = ex_mem.alu_result;
        }
    }

    // Forwarding from WB stage to EX stage
    if (mem_wb.rd != 0) {
        if (mem_wb.rd == id_ex.rs) {
            id_ex.reg_rs_value = mem_wb.alu_result;
        }
        if (mem_wb.rd == id_ex.rt) {
            id_ex.reg_rt_value = mem_wb.alu_result;
        }
    }
}
```

주어진 forward 함수는 파이프라인의 데이터 의존성을 해결하기 위해 데이터를 전달하는 역할을 한다. 이 함수는 MEM(Memory Access) 단계와 WB(Write Back) 단계에서 EX(Execute) 단계로 데이터를 포워딩하여, 파이프라인의 효율성을 높이고 데이터 의존성으로 인한 지연을 최소화한다. 먼저, MEM 단계에서 EX 단계로의 포워딩을 살펴보겠다. 함수의 첫 부분에서는 ex_mem.rd가 0이 아닌지 확인한다. 이는 MEM 단계에서 목적 레지스터(rd)가 0이 아닌 경우에만 포워딩을 수행하기 위해서이다. MIPS 아키텍처에서 레지스터 0은 항상 0이므로, 목적 레지스터가 0이 아닌 경우에만 포워딩을 수행해야 한다. 그 다음, MEM 단계의 목적 레지스터(ex_mem.rd)가 현재 디코딩 중인 명령어의 소스 레지스터(id_ex.rs)와 같은지 확인한다. 같다면, 현재 디코딩 중인 명령어의 소스 레지스터 값(id_ex.reg_rs_value)에 MEM 단계의 ALU 결과(ex_mem.alu_result)를 포워딩한다. 또한, MEM 단계의 목적 레지스터(ex_mem.rd)가 현재 디코딩 중인 명령어의 타겟 레지스터(id_ex.rt)와 같은지 확인한다. 같다면, 현재 디코딩 중인 명령어의 타겟 레지스터 값(id_ex.reg_rt_value)에 MEM 단계의 ALU 결과(ex_mem.alu_result)를 포워딩한다. 다음으로, WB 단계에서 EX 단계로의 포워딩을 살펴보겠다. 함수의 두 번째 부분에서는 mem_wb.rd가 0이 아닌지 확인한다. 이는 WB 단계에서 목적 레지스터(rd)가 0이 아닌 경우에만 포워딩을 수행하기 위해서이다. 그 다음, WB 단계의 목적 레지스터(mem_wb.rd)가 현재 디코딩 중인 명령어의 소스 레지스터(id_ex.rs)와 같은지 확인한다. 같다면, 현재 디코딩 중인 명령어의 소스 레지스터 값(id_ex.reg_rs_value)에 WB 단계의 ALU 결과(mem_wb.alu_result)를 포워딩한다. 또한, WB 단계의 목적 레지스터(mem_wb.rd)가 현재 디코딩 중인 명령어의 타겟 레지스터(id_ex.rt)와 같은지 확인한다. 같다면, 현재 디코딩 중인 명령어의 타겟 레지스터 값(id_ex.reg_rt_value)에 WB 단계의 ALU 결과(mem_wb.alu_result)를 포워딩한다. 데이터 포워딩을 통해 레지스터 값에 올바른 값이 할당되고 명령어들이 올바른 값을 사용할 수 있다.

3. Result

```
**************************************************
Cycle: 10
R[2]: 0
Number of instructions: 9
Number of memory access instructions: 2
Number of Register ops: 5
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct: 0, mis predict: 0, total predict: 0
**************************************************
```

simple.bin

```
**************************************************
Cycle: 12
R[2]: 100
Number of instructions: 11
Number of memory access instructions: 4
Number of Register ops: 6
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct: 0, mis predict: 0, total predict: 0
**************************************************
```

simple2.bin

```
**************************************************
Cycle: 20
R[2]: 1
Number of instructions: 19
Number of memory access instructions: 7
Number of Register ops: 6
Number of branch instruction: 1
Number of jump instruction: 2
Predict correct: 0, mis predict: 1, total predict: 1
**************************************************
```

simple3.bin

```
Cycle 9578: PC = 0x0000001C
Write Back: Instruction = 0x27BD0020, Register[29] = 0x0000002A
Executing instruction: 0x00000000
rs: R[0] = 0x00000000, rt: R[0] = 0x00000000
opcode: 0
Decode: PC = 0x0000001C, Instruction = 0x03C0E821
Fetch: PC = 0x00000020, Instruction = 0x8FBF001C

Cycle 9579: PC = 0x00000020
Write Back: Instruction = 0x03E00008, Register[0] = 0x00000000
Executing instruction: 0x03C0E821
rs: R[30] = 0x0000000A, rt: R[0] = 0x00000000
opcode: 0
Decode: PC = 0x00000020, Instruction = 0x^C
```

simle4.bin

결과에서 보이다시피 simple.bin과 simple2.bin은 구현에 성공하였으나 simple3.bin은 사이클이 20번만 돌아가고 최종 결과가 도출되었고, simple4.bin은 무한루프에 빠지는 결과가 도출되었다.

코드를 만들며 다양한 문제가 발생하였는데 처음 simple.bin을 실행시킬 당시 명령어 개수가 잘못 도출되는 문제가 발생하였다. Instruction_count를 셀 때 fetch 단계에서 하나씩 증가를 시켜야 하는데 당시 코드에서는 메인 함수에서 증가를 시키고 있었고 이를 발견하고 수정하여 명령어 개수는 제대로 도출되게 하였다.

이후 마주한 문제는 레지스터 연산 개수가 정확히 나오지 않는 것이었다. 레지스터 연산의 경우 R-type과 I-type 연산을 실행할 때 카운트가 되어야 하는데 명령어 개수와 똑같이 나오는 문제가 발생하여 문제점을 알아보았다. 코드가 nop 명령어를 만났을 경우 레지스터 연산으로 간주하고 있어서 연산 개수가 늘어나고 있었고 R-type 연산 중 nop이 아닌 경우만 레지스터 연산으로 카운트하도록 코드를 수정하였다.

Reg[2]에 값이 누적이 되지 않는 문제는 저번 과제와 동일한 문제여서 금방 해결할 수 있었다. 저번과 마찬가지로 LW 연산 시 rd에 rs 값이 아닌 rt 값이 할당되고 있었고 rs 값이 할당되도록 수정하여 제대로 값이 누적되게 하였다.

그러나 simple3.bin의 사이클이 20번만 돌아가고 완료되는 문제는 해결하지 못하였다. 디버깅을 해보니 bne 명령어를 실행할 때 rs와 rt 값이 0으로 동일하여 bne가 실행되지 않고 있음을 발견하였다. 따라서 이전 명령어인 slti 실행 시 문제가 있다고 판단하여 확인해 본 결과 slti에서 실행한 결과값들이 다음 명령어인 bne를 실행할 때 제대로 전달되지 않는다는 사실을 알게 되었다. 이 문제는 forwarding이 제대로 되지 않아 생긴 문제라고 생각되어 오랜 시간 고민해보았으나 결국 해결하지 못하였다. simple4.bin을 실행하였을 때 무한루프에 빠지는 것도 Data dependency와 Control dependency 문제를 제대로 해결하지 못했기 때문이라고 생각된다.

이번 과제를 통해 내용을 이해한 후 큰 틀을 잡고 하나씩 처리하는 것이 매우 중요하다는 사실을 깨닫게 되었다. 또한, 디버깅의 중요성 역시 아주 잘 알게 되었다. Data dependency와 Control dependency를 해결하기 위한 Forwarding과 Stall을 제대로 구현하지 못했다는 점이 매우 아쉽고 후에 다시 한 번 코드를 리뷰해보며 잘못된 부분을 인지한 후 고쳐보도록 할 것이다.

IV. References

https://cse.sc.edu/~jbakos/611/cpu/multicycle_cpu.shtml

https://cseweb.ucsd.edu/~j2lau/cs141/week6.html

https://blog.naver.com/mayooyos/220878015447