

Implementation of Pipelined MIPS

성영준 Yeong-Jun Seong

32202231(남은 휴일 1일)

단국대학교 모바일시스템공학과

목차

1. 서론
2. Pipelined
 - 2.1 Single Cycle
 - 2.2 Multi Cycle
 - 2.3 Pipeline
 - 2.4 Latch
 - 2.5 Hazard
 - 2.5.1 Data Hazard
 - 2.5.2 Control Hazard
 - 2.5.3 Structurel Hazard
 - 2.6 Data path
3. Program
 - 3.1 main
 - 3.2 Latch
 - 3.3 stage
4. Program의 핵심 요소
 - 4.1 Detect and Wait
 - 4.2 ANT
 - 4.3 ALT
 - 4.4 LTP
 - 4.5 BTB
 - 4.6 Fowarding
5. 실행결과
6. 결론

1.1 서론

본 프로그램은 파이프라인 기반 프로세서 아키텍처를 기반으로 한 프로그램의 설계 및 구현되었다. 파이프라인 프로세서는 명령어를 여러 단계로 나누어 각 단계가 동시에 수행되도록 하여 전체 처리 속도를 향상시키는 특성을 가진 컴퓨터 아키텍처이다. 각 명령어의 처리 시간이 단일 사이클 동안 완전히 이루어지는 것이 아니라, 여러 사이클에 걸쳐 동시에 처리됨으로써 시스템의 효율성과 성능을 높인다. 이러한 특성은 고성능 컴퓨팅 환경이나 복잡한 제어 시스템에서 특히 유용하게 사용된다.

파이프라인 아키텍처의 핵심은 명령어의 처리 단계를 나누고 각 단계를 독립적으로 실행하는 것이다. 이를 위해, 해당 프로그램에서 프로세서는 명령어를 패치, 디코드, 실행, 메모리 접근, 그리고 결과 쓰기의 일련의 단계로 나누어 처리한다. 각 단계는 별도의 하드웨어 구성 요소에 의해 처리되며, 이들 구성 요소는 동시에 다른 명령어의 다른 단계를 수행할 수 있다. 예를 들어, 한 명령어가 실행 단계에 있을 때, 다른 명령어는 디코드 단계에 있을 수 있다.

프로그램의 설계 방식은 이러한 파이프라인 아키텍처의 원리를 충실히 따르며, 명령어의 각 단계를 정확하게 구현하기 위해 다양한 하드웨어 시뮬레이션 기법을 사용한다. ALU(산술 논리 단위)는 산술 및 논리 연산을 담당하며, 분기 및 점프와 같은 제어 명령어는 프로그램 카운터(PC)의 업데이트 메커니즘을 통해 처리된다. 메모리 접근은 데이터와 명령어를 저장하는 메모리 시스템을 통해 이루어지며, 최종 결과는 목적지 레지스터에 쓰여진다.

또한, 제어 유닛은 명령어의 유형에 따라 필요한 제어 신호를 생성하여 각 하드웨어 구성 요소가 적절한 작업을 수행하도록 지시한다. 이 신호들은 멀티플렉서를 통해 구체적인 데이터 경로를 선택하며, 이를 통해 데이터가 올바른 목적지로 전달되거나 적절한 연산이 수행되도록 한다. 파이프라인의 각 단계에서는 하드웨어 자원 간의 충돌을 방지하고 데이터 의존성을 해결하기 위한 다양한 메커니즘이 구현되어 있다.

이 프로그램의 구현 방식은 특히 시뮬레이션을 통해 실제 하드웨어에 근접한 경험을 제공하기 위해 설계되었다. 이는 학습자가 실제 하드웨어와 유사한 환경에서 프로그램의 흐름을 이해하고, 각 구성 요소의 상호 작용을 실시간으로 관찰할 수 있게 한다. 이를 통해 파이프라인 아키텍처의 복잡한 동작 원리를 보다 직관적으로 이해할 수 있다.

종합적으로, 이 프로그램은 파이프라인 프로세서의 설계 및 운용 원리를 따라 개발된 프로그램으로, 이를 개발하면서 파이프라인 아키텍처에 대한 이해를 심화할 수 있었다. 각 단계에서 발생하는 데이터와 제어 흐름을 시뮬레이션하여 파이프라인의 효율성과 성능을 높이는 방법을 학습할 수 있었다.

2.0 Pipeline에 대해서

본 프로그램은 Pipelined의 핵심 원칙을 반영하여 설계 되었기에 제작한 프로그램에 대한 설명에 앞서 Single Cycle, Multi Cycle에 대한 설명을 거쳐 Pipelined에 대한 이해를 바탕으로 프로그램에 대한 설명을 이어나갈 예정이다.

2.1 Single Cycle

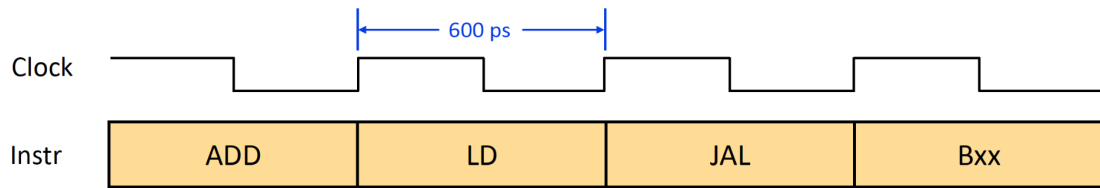
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

[1] Single Cycle의 문제점

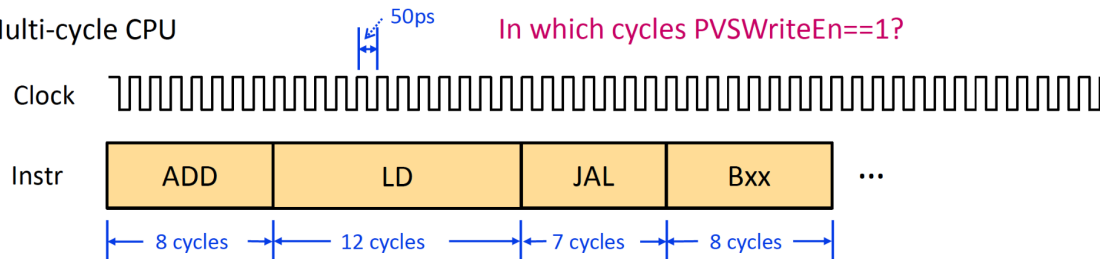
싱글 사이클 아키텍처는 하나의 명령어가 실행되기 위해서는 한 번의 사이클을 모두 완료해야 다음 사이클을 실행할 수 있다. 즉, 한 번의 사이클 동안 하나의 명령어만 실행할 수 있다는 특성을 가지고 있다. 이러한 구조는 명령어가 단순하고 예측 가능하며, 구현이 용이하다는 장점이 있다. 하지만 이 방식은 성능 면에서 몇 가지 문제를 야기한다. 싱글 사이클 아키텍처에서 각 명령어는 고정된 사이클 시간을 필요로 한다. 나머지 명령어들이 상대적으로 짧은 시간 내에 완료되더라도 해당 사이클이 끝날 때까지 대기해야 한다. 이는 명령어 실행 시 불필요한 처리 시간의 낭비를 초래한다. 또한, 싱글 사이클 아키텍처는 하드웨어의 활용도가 낮다는 문제를 가지고 있다. 각 명령어가 실행될 때, 하드웨어 자원들은 명령어의 특정 단계에서만 사용되고 나머지 시간 동안에는 비활성화 상태로 대기하게 된다. 이는 하드웨어 자원의 비효율적인 사용으로 이어지며, 전체 시스템의 성능을 저하시킨다. 이러한 문제를 보완하기 위해 멀티 사이클 아키텍처가 등장하게 되었다.

2.2 Multi Cycle

■ Single-cycle CPU



■ Multi-cycle CPU

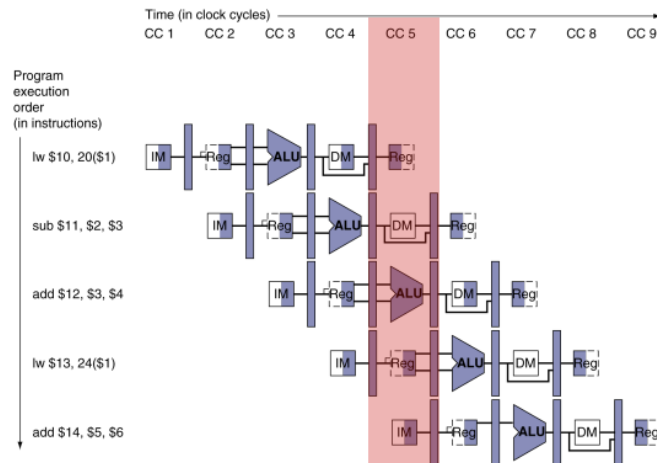


[2]Single Cycle 과 Multi Cycle 비교

멀티 사이클 아키텍처는 싱글 사이클 아키텍처의 한계를 보완하기 위해 등장했다. 싱글 사이클 아키텍처에서는 하나의 명령어가 충분히 실행될 수 있도록 1개의 사이클을 지정하지만, 이는 각 명령어의 가장 긴 실행 시간을 기준으로 사이클 시간을 설정해야 한다는 문제를 안고 있다. 따라서 각 명령어의 실제 처리 시간과 관계없이, 모든 명령어는 동일한 길이의 사이클을 사용하게 되어 처리 시간의 낭비가 발생한다.

멀티 사이클 아키텍처는 이러한 문제를 해결하기 위해 명령어의 실행을 여러 단계로 나누고, 각 단계마다 별도의 사이클 시간을 할당하는 방식을 채택한다. 명령어 실행의 5단계를 각각의 사이클로 분할하고, 각 사이클의 시간을 명령어 처리에 필요한 최소 시간으로 설정함으로써 보다 효율적인 처리 시간을 제공한다. 이때, 각 단계의 사이클 시간은 해당 단계에서 가장 시간이 많이 필요한 명령어(Critical Path)를 기준으로 설정된다. 하지만 멀티 사이클 아키텍처도 한계가 있다. 명령어 처리 속도가 긴 명령어가 많거나, 명령어의 조합에 따라 멀티 사이클 아키텍처의 효율성이 떨어질 수 있다. 특히, 각 단계가 순차적으로 실행되기 때문에, 한 사이클에서 모든 단계를 다 완료해야 다음 명령어로 넘어갈 수 있다는 제약이 존재한다. 이러한 비효율성을 극복하고자 파이프라인 아키텍처가 도입되었다.

2.3 Pipeline



[3] Pipelined

파이프라인 아키텍처는 단일 명령어 처리인 싱글 사이클과 멀티 사이클 방식의 한계를 극복하기 위해 등장한 개념이다. 파이프라인은 여러 명령어를 병렬로 처리하여 처리 속도를 높이는 방식으로, 단일 명령어 처리보다 훨씬 높은 성능을 제공한다. 파이프라인 아키텍처는 이러한 문제를 해결하고자 명령어의 각 단계를 병렬로 처리하는 방식을 도입했다. 파이프라인은 여러 명령어를 중첩하여 동시에 처리할 수 있도록 설계되었다. 이는 마치 공장에서 여러 작업이 동시에 진행되는 것과 비슷한 개념이다.

파이프라인 아키텍처는 다음과 같은 단계로 구성된다:

1. 명령어 인출 (Instruction Fetch)
2. 명령어 디코딩 (Instruction Decode)
3. 실행 (Execution)
4. 메모리 접근 (Memory Access)
5. 쓰기 (Write Back)

각 단계는 독립적으로 작동하며, 한 단계가 완료되면 즉시 다음 단계로 넘어간다. 이를 통해, 각 사이클에서 여러 명령어의 다른 단계가 동시에 실행될 수 있다. 즉, 파이프라인 아키텍처는 각 명령어를 처리하는 총 시간을 줄이는 것이 아니라, 병렬로 명령어를 처리함으로써 전체 프로그램의 처리 성능을 높인다. 이를 ILP(Instruction-Level Parallelism)라고 하며, 여러 명령어를 중첩하여 실행하는 방식으로, 전체 시스템의 효율성을 극대화한다. 결론적으로 파이프라인 아키텍처는 단일 명령어 처리 방식의 비효율성을 극복하고, 명령어 처리의 효율성을 극대화하기 위해 도입된 혁신적인 기술이다. 각 명령어를 여러 단계로 나누고 이를 병렬로 처리함으로써, 단일 명령어 처리

방식보다 훨씬 높은 성능을 제공한다. 파이프라인 아키텍처는 현대 컴퓨터 시스템에서 중요한 역할을 담당하며, 명령어 처리의 효율성을 극대화하여 전체 프로그램의 처리 속도를 높이는 데 기여하고 있다.

2.4 Latch

파이프라인 아키텍처는 단일 명령어 처리 방식의 한계를 극복하고 명령어 처리 속도를 향상시키기 위해 도입된 혁신적인 기술이다. 파이프라인을 구현하면서 중요한 요소 중 하나가 바로 "Latch"이다. Latch는 각 단계에서 명령어의 값을 저장하고 처리하기 위해 필요한 하드웨어 구성 요소로, 파이프라인의 효율적인 운영을 가능하게 한다. 단계별 처리 과정에서 중요한 것은 각 명령어의 상태를 정확하게 유지하고 관리하는 것이다. 만약 명령어마다 구분 없이 값을 처리한다면, 명령어의 흐름이 뒤섞여 파이프라인의 효율적인 동작이 불가능해진다. 따라서 각 명령어의 상태를 저장하고 다음 단계로 넘겨주는 역할을 하는 Latch가 필요하다. Latch는 각 파이프라인 단계 사이에 위치하여, 해당 단계에서 처리된 정보를 저장한다. 예를 들어, 첫 번째 명령어가 IF 단계를 완료하면, 그 결과를 Latch에 저장하고, 다음 사이클에서는 이 정보를 ID 단계로 넘긴다. 이렇게 함으로써 각 명령어의 상태를 정확하게 유지하고, 파이프라인의 흐름을 원활하게 이어갈 수 있다.

Latch는 다음과 같은 역할을 수행한다.

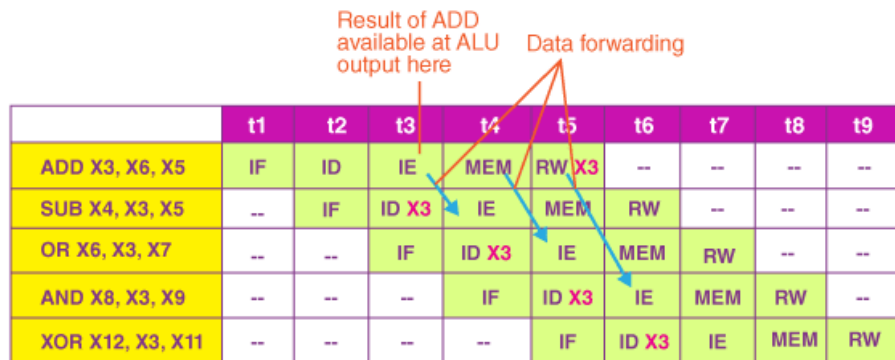
1. 정보 저장: 각 단계에서 처리된 명령어의 값을 저장하여 다음 단계로 넘긴다.
2. 명령어 구분: 명령어마다 상태를 유지하고, 다음 단계로 넘어갈 때 필요한 정보를 제공한다.
3. 동기화: 파이프라인의 각 단계를 동기화하여, 명령어들이 혼합되지 않고 순차적으로 처리되도록 한다.

Latch의 중요성은 파이프라인의 효율적인 동작을 보장하는 데 있다. 각 명령어의 상태를 정확하게 관리하지 않으면, 파이프라인의 단계별 처리 과정에서 명령어의 흐름이 뒤섞여 오류가 발생할 수 있다. Latch는 이러한 문제를 방지하고, 명령어의 상태를 정확하게 유지하여 파이프라인의 성능을 극대화한다.

2.5.0 Hazard

병렬실행시 문제점이 여럿 생기는데 이것을 Hazard라고 한다. Hazard에는 Data Hazard, Control Hazard, Structure Hazard가 있다.

2.5.1.1 데이터 위험 (Data Hazard)



[4] Data Hazard

데이터 위험(Data Hazard)은 파이프라인 아키텍처에서 명령어 간의 데이터 의존성으로 인해 발생하는 문제를 말한다. 이는 특정 명령어가 필요한 데이터가 아직 이전 명령어에서 계산되지 않았을 때 발생하며, 파이프라인의 효율적인 실행을 방해한다. 데이터 위험은 주로 파이프라인의 성능 저하와 처리 지연을 초래할 수 있다. 이러한 데이터 위험을 해결하기 위한 다양한 기법이 존재한다.

데이터 위험의 종류

데이터 위험(Data Hazard)은 파이프라인 아키텍처에서 명령어 간의 데이터 의존성으로 인해 발생하는 문제를 의미한다. 이러한 데이터 위험은 크게 세 가지로 분류된다: 읽기 후 쓰기(Read After Write, RAW) 위험, 쓰기 후 읽기(Write After Read, WAR) 위험, 그리고 쓰기 후 쓰기(Write After Write, WAW) 위험이다. 각 데이터 위험의 정의와 예시는 다음과 같다.

1. 읽기 후 쓰기(Read After Write, RAW) 위험

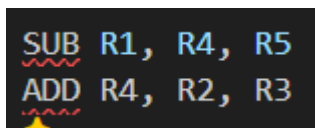
정의: 읽기 후 쓰기(Read After Write, RAW) 위험은 한 명령어가 사용하는 레지스터의 값이 이전 명령어에 의해 변경될 때 발생한다. 이는 특정 레지스터의 값이 업데이트되기 전에 후속 명령어가 그 값을 읽으려 할 때 나타난다.

```
ADD R1, R2, R3
SUB R4, R1, R5
```

첫 번째 명령어인 ADD R1, R2, R3는 레지스터 R1에 새로운 값을 계산하여 저장한다. 이 명령어가 실행된 후, 두 번째 명령어 SUB R4, R1, R5는 레지스터 R1의 값을 사용하여 연산을 수행해야 한다. 그러나 만약 첫 번째 명령어가 레지스터 R1에 값을 쓰기 전에 두 번째 명령어가 R1의 값을 읽으려 한다면, 아직 갱신되지 않은 잘못된 값을 읽게 되어 연산 결과에 오류가 발생할 수 있다. 이러한 상황이 바로 RAW 위험이다.

2. 쓰기 후 읽기(Write After Read, WAR) 위험

정의: 쓰기 후 읽기(Write After Read, WAR) 위험은 이전 명령어가 읽기 작업을 수행하기 전에 후속 명령어가 쓰기 작업을 완료하는 경우 발생한다. 이는 후속 명령어가 레지스터에 값을 쓰기 전에 이전 명령어가 해당 레지스터의 값을 읽으려 할 때 나타난다.

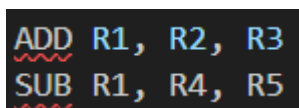


```
SUB R1, R4, R5
ADD R4, R2, R3
```

첫 번째 명령어인 SUB R1, R4, R5는 레지스터 R4의 값을 읽어야 한다. 그러나 두 번째 명령어 ADD R4, R2, R3가 실행되면서 레지스터 R4에 새로운 값을 저장하려 한다. 만약 두 번째 명령어가 첫 번째 명령어가 레지스터 R4의 값을 읽기 전에 실행된다면, WAR 위험이 발생할 수 있다. 이는 첫 번째 명령어가 의도한 값을 읽지 못하고, 후속 명령어에 의해 변경된 값을 읽게 되는 문제를 초래할 수 있다.

3. 쓰기 후 쓰기(Write After Write, WAW) 위험

정의: 쓰기 후 쓰기(Write After Write, WAW) 위험은 두 명령어가 같은 레지스터에 값을 쓰려고 할 때 발생하며, 후속 명령어가 먼저 쓰기 작업을 수행하면 문제가 발생한다. 이는 두 명령어가 동시에 같은 레지스터에 값을 쓰려고 할 때 나타난다.



```
ADD R1, R2, R3
SUB R1, R4, R5
```

첫 번째 명령어인 ADD R1, R2, R3는 레지스터 R1에 새로운 값을 저장하려 한다. 두 번째 명령어 SUB R1, R4, R5도 레지스터 R1에 값을 저장하려 한다. 만약 두

번째 명령어가 첫 번째 명령어보다 먼저 실행되어 레지스터 R1에 값을 쓰게 되면, 첫 번째 명령어의 결과가 덮어쓰여지게 된다. 이로 인해 원하는 연산 결과를 얻지 못하게 되며, 이러한 상황이 바로 WAW 위험이다.

이러한 데이터 위험들은 파이프라인의 순차적 실행을 방해하고, 성능 저하를 초래할 수 있다. 이를 해결하기 위해 다양한 기법들이 사용되며, 이러한 기법들은 데이터 의존성을 관리하고 해결하는 데 중요한 역할을 한다. 데이터 위험을 효과적으로 처리함으로써, 파이프라인 아키텍처는 고성능의 명령어 처리 능력을 유지할 수 있다.

2.5.1.2 Data Hazard의 해결 기법

데이터 위험(Data Hazard)은 파이프라인 구조의 프로세서에서 명령어 간의 데이터 의존성으로 인해 발생하는 문제로, 이는 파이프라인의 성능 저하와 처리 지연을 초래할 수 있다. 이를 해결하기 위해 여러 기법이 사용되며, 대표적인 방법으로는 포워딩(Forwarding), 스톨링(Stalling), 레지스터 리네이밍(Register Renaming), 명령어 재배치(Instruction Reordering) 등이 있다. 이러한 기법들은 데이터 의존성을 관리하고 해결함으로써 파이프라인의 효율성을 높이는 데 중요한 역할을 한다.

포워딩(Forwarding)

정의: 포워딩은 데이터가 필요할 때 이전 명령어의 결과를 레지스터를 거치지 않고 직접 전달하는 방법이다. 이를 통해 후속 명령어가 필요한 데이터를 즉시 사용할 수 있게 하여 데이터 위험을 줄인다.

작동 원리

- EX/MEM 파이프라인 레지스터에서 바로 ALU로 데이터를 전달한다.
- MEM/WB 파이프라인 레지스터에서 ALU로 데이터를 전달한다.

장점: 포워딩은 데이터 위험을 줄여 파이프라인의 성능을 향상시킬 수 있다. 데이터가 즉시 전달됨으로써 파이프라인의 지연을 최소화하고 명령어 처리 속도를 높일 수 있다.

스톨링(Stalling)

정의: 스톨링은 데이터 위험이 해결될 때까지 파이프라인의 진행을 멈추는 방법이다.
파이프라인의 일부 단계를 일시적으로 멈추어 후속 명령어가 데이터 위험 없이 실행될 수 있도록 한다.

작동 원리

- 현재 명령어를 멈추고, 이후 명령어들의 실행을 지연시킨다.
- 데이터가 준비될 때까지 파이프라인을 일시적으로 멈춘다.

단점: 스톨링은 파이프라인의 효율을 떨어뜨리고, 전체 실행 시간이 늘어날 수 있다. 이는 파이프라인이 멈추는 동안 다른 명령어가 대기해야 하기 때문이다.

레지스터 리네이밍(Register Renaming)

정의: 레지스터 리네이밍은 명령어들이 동일한 레지스터를 사용하지 않도록 가상 레지스터를 사용하는 방법이다. 이를 통해 WAR 및 WAW 위험을 줄여 파이프라인 효율성을 높인다.

작동 원리

- 컴파일러나 하드웨어가 동일한 레지스터 사용을 피하기 위해 가상 레지스터를 할당한다.

장점: 레지스터 리네이밍은 WAW 및 WAR 위험을 줄여 파이프라인 효율성을 높인다.
이는 동일한 레지스터를 사용하는 명령어 간의 충돌을 방지하여 데이터 위험을 최소화할 수 있다.

명령어 재배치(Instruction Reordering)

정의: 명령어 재배치는 컴파일러가 데이터 의존성을 줄이기 위해 명령어의 순서를 변경하는 방법이다. 이를 통해 파이프라인의 효율성을 높이고 데이터 위험을 줄일 수 있다.

작동 원리

- 컴파일러가 명령어를 분석하여 데이터 의존성이 없는 명령어를 먼저 실행하도록 순서를 변경한다.

장점: 명령어 재배치는 데이터 위험을 줄이고 파이프라인의 효율을 높인다. 이는 데이터 의존성이 없는 명령어를 먼저 실행함으로써 파이프라인의 지연을 최소화할 수 있다.

데이터 위험을 해결하기 위해 다양한 기법들이 사용된다. 포워딩은 데이터가 필요할 때 즉시 전달하여 데이터 위험을 줄이고, 스톨링은 파이프라인을 일시적으로 멈추어 데이터가 준비될 때까지 기다린다. 레지스터 리네이밍은 가상 레지스터를 사용하여 동일한 레지스터 사용을 피하고, 명령어 재배치는 데이터 의존성이 없는 명령어를 먼저 실행하여 파이프라인의 효율을 높인다. 이러한 기법들은 파이프라인 아키텍처의 효율성을 극대화하고 데이터 위험을 최소화하기 위해 사용된다.

Detect-and-Wait

Detect-and-Wait 기법은 데이터 해저드를 처리하기 위해 사용하는 기법으로, 명령어 간의 의존성을 감지하고 파이프라인을 정지시켜 필요한 데이터가 준비될 때까지 기다리는 방법이다. 이 기법은 데이터가 필요할 때 해당 데이터를 올바르게 사용할 수 있도록 보장하여 데이터 무결성을 유지하고, 잘못된 결과를 피할 수 있게 한다.

작동 원리

1. 해저드 감지: 파이프라인에서 실행 중인 명령어들이 데이터 해저드를 발생시키는지 검사한다. 이는 주로 명령어 디코드 단계에서 이루어진다.
2. 스톨 삽입: 데이터 해저드가 감지되면 파이프라인의 실행을 일시적으로 중지시키고, 스톨을 삽입한다. 스톨은 파이프라인의 다른 단계가 진행되지 않도록 하고, 필요한 데이터가 준비될 때까지 기다리게 한다.
3. 데이터 포워딩: 데이터가 준비되면, 필요한 명령어에 데이터를 전달한다. 이를 통해 데이터가 올바르게 사용될 수 있도록 한다.
4. 파이프라인 재개: 데이터가 전달된 후, 파이프라인의 실행을 재개한다. 스톨이 제거되고 명령어 실행이 계속된다.

장점

- 단순성: 이 기법은 매우 단순하고 구현하기 쉽다. 복잡한 하드웨어나 추가적인 제어 로직이 필요하지 않다.
- 데이터 무결성 보장: 데이터 해저드를 감지하고, 필요한 데이터가 준비될 때까지 기다림으로써 데이터 무결성을 보장할 수 있다.
- 유연성: 모든 종류의 데이터 해저드에 대해 적용할 수 있어 범용적인 해결 방법으로 사용 가능하다.

단점

- 파이프라인 효율성 저하: 스톨이 발생하면 파이프라인의 실행이 멈추므로, 전체 시스템의 성능이 저하될 수 있다. 스톨 사이클이 증가하면 파이프라인의 효율성이 크게 떨어진다.
- 성능 저하: 데이터가 준비될 때까지 기다려야 하므로, 파이프라인의 처리 속도가 느려질 수 있다. 이는 전체 명령어 처리 시간을 증가시켜 성능을 저하시킨다.

Detect-and-Wait 기법은 데이터 해저드를 처리하는 가장 단순하고 직관적인 방법 중 하나이다. 데이터 무결성을 보장하고 잘못된 결과를 피할 수 있지만, 파이프라인의 효율성과 성능 저하라는 단점을 가지고 있다. 이러한 단점을 극복하기 위해 다른 고급 기법들도 함께 고려할 필요가 있다. 그러나 Detect-and-Wait 기법은 여전히 데이터 해저드를 처리하는 기본적인 방법으로 중요한 역할을 한다.

2.5.2.1 제어 위험(Control Hazard)

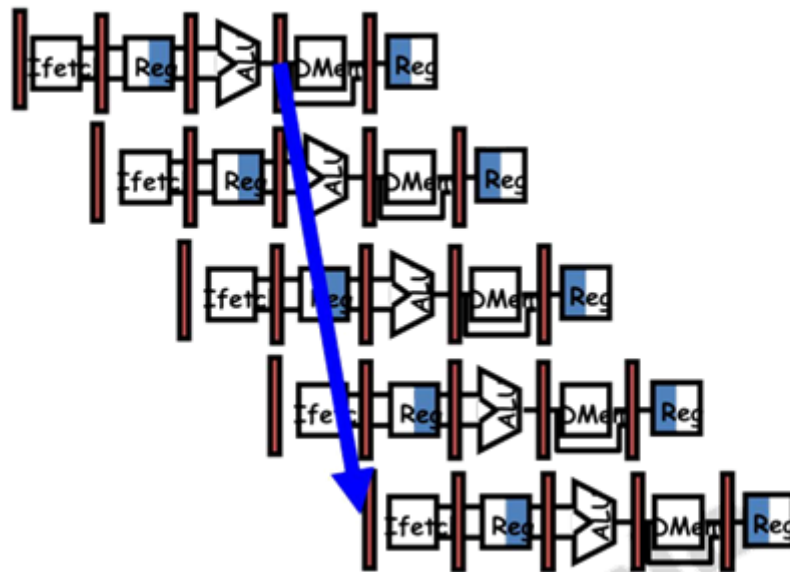


Figure 13.1

[5] Control Hazard

제어 위험(Control Hazard)은 분기 명령어로 인해 발생하는 문제로, 파이프라인 프로세서에서 분기 여부가 결정되기 전에 다음 명령어를 가져오면서 발생한다. 이러한 제어 위험은 파이프라인의 성능을 저하시키며, 명령어 처리의 지연을 초래할 수 있다.

제어 위험(Control Hazard)의 정의

제어 위험은 분기 명령어(예: 조건부 점프, 무조건 점프 등)로 인해 프로그램 카운터(Program Counter, PC)의 값이 변경될 때 발생한다. 파이프라인 구조에서는 명령어가 순차적으로 실행되지만, 분기 명령어가 등장하면 다음에 실행할 명령어의 주소를 정확히 예측해야 한다. 분기 명령어의 결과가 나오기 전에 잘못된 명령어를 가져오게 되면 파이프라인이 잘못된 명령어를 처리하게 되어 성능 저하가 발생한다.

제어 위험의 발생 원인

1. 분기 명령어의 존재: 분기 명령어는 다음에 실행할 명령어의 주소를 변경한다. 이로 인해 파이프라인은 다음 명령어를 잘못 예측할 수 있다.
2. 분기 결과의 지연: 분기 명령어의 조건이 평가되기까지 시간이 걸리며, 그 동안 파이프라인은 명령어를 계속 가져오게 된다.
3. 분기 예측 실패: 파이프라인이 분기 명령어의 결과를 잘못 예측하면 잘못된 명령어를 가져와 실행하게 된다.

2.5.2.2 분기 예측 기법

분기 예측(Branch Prediction)은 분기 명령어로 인해 발생하는 제어 해저드(Control Hazard)를 완화하기 위한 기술이다. 이 기술의 목적은 분기가 해결되기 전에 분기의 결과를 예측하여 파이프라인이 명령어를 추측적으로 계속 실행할 수 있도록 하는 것이다. 분기 예측은 크게 정적 분기 예측(Static Branch Prediction)과 동적 분기 예측(Dynamic Branch Prediction)으로 나뉜다.

정적 분기 예측 (Static Branch Prediction)

정적 분기 예측은 분기의 특성이나 과거의 행동에 기반하여 분기의 결과를 미리 정해두는 간단한 접근 방식이다. 정적 분기 예측은 주로 두 가지 방법으로 나뉜다: 항상 분기를 취한다고 예측하는 방법(Always Taken, ALT)과 분기를 취하지 않는다고 예측하는 방법(Always Not Taken, ANT)이다.

1. ANT (Ahead Not Taken)

정의: ANT 기법은 분기 명령어가 발생했을 때, 분기가 발생하지 않을 것으로 예측하고 다음 명령어를 순차적으로 가져오는 방법이다.

작동 원리:

- 분기 명령어가 발생하면 분기가 발생하지 않을 것으로 가정한다.
- 분기 예측이 맞을 경우, 파이프라인의 성능 저하가 없다.
- 분기 예측이 틀릴 경우, 잘못 가져온 명령어를 버리고 올바른 명령어를 가져온다.

장점: ANT 기법은 구현이 간단하고, 분기 예측이 맞을 경우 성능 저하가 없다.

단점: 분기 예측이 틀릴 경우, 파이프라인이 재시작되어 성능 저하가 발생할 수 있다.

2. ALT (Ahead Likely Taken)

정의: ALT 기법은 분기 명령어가 발생했을 때, 분기가 발생할 것으로 예측하고 분기된 주소의 명령어를 가져오는 방법이다.

작동 원리:

- 분기 명령어가 발생하면 분기가 발생할 것으로 가정한다.
- 분기 예측이 맞을 경우, 파이프라인의 성능 저하가 없다.
- 분기 예측이 틀릴 경우, 잘못 가져온 명령어를 버리고 올바른 명령어를 가져온다.

장점: 분기 예측이 맞을 경우, 성능 저하 없이 파이프라인을 효율적으로 유지할 수 있다.

단점: 분기 예측이 틀릴 경우, ANT 기법보다 더 큰 성능 저하가 발생할 수 있다.

동적 분기 예측 (Dynamic Branch Prediction)

동적 분기 예측은 실행 시간 정보를 사용하여 보다 정확한 예측을 수행하는 방법이다. 이 접근 방식은 분기 대상 버퍼(branch target buffers) 또는 분기 히스토리 테이블(branch history tables)을 사용하여 프로그램 실행 중 분기의 행동을 추적한다. 이러한 버퍼는 분기의 과거 결과에 대한 정보를 저장하여 미래의 행동을 예측하는 데 도움을 준다.

3. LTP (Loop Termination Prediction)

정의: LTP 기법은 반복 루프의 종료를 예측하여 분기 명령어의 결과를 예측하는 방법이다.

작동 원리:

- 반복 루프의 패턴을 분석하여 루프가 종료될 시점을 예측한다.
- 분기 명령어가 반복 루프 내에 있을 경우, 루프의 반복 횟수를 예측하여 종료 시점을 예측한다.
- 예측이 맞을 경우, 파이프라인의 성능 저하가 없다.
- 예측이 틀릴 경우, 잘못 가져온 명령어를 버리고 올바른 명령어를 가져온다.

장점: 반복 루프의 종료 시점을 정확히 예측하면 파이프라인의 성능을 크게 향상시킬 수 있다.

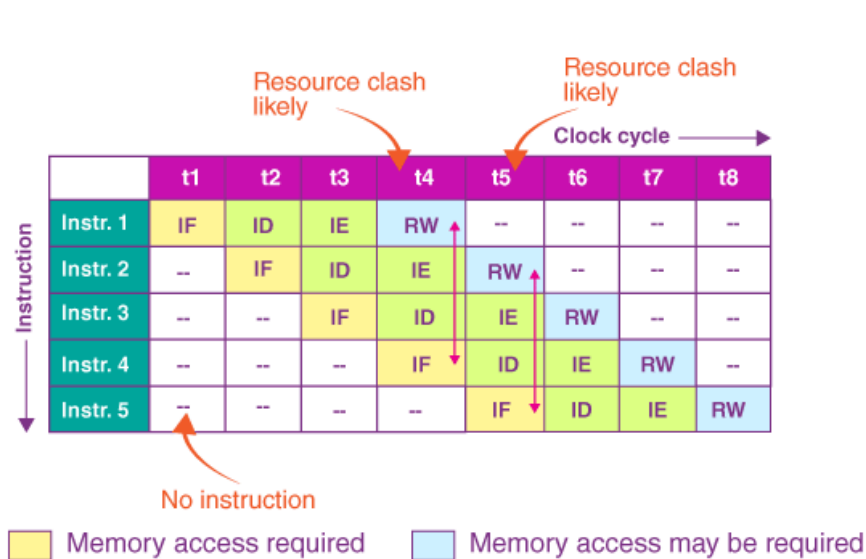
단점: 루프의 패턴을 정확히 분석하지 못하면 예측 실패로 인한 성능 저하가 발생할 수 있다.

분기 예측 기법의 비교

기 법	정의	작동 원리	장점	단점
ANT	분기 발생하지 않음 예측	분기 발생하지 않을 것으 로 가정	구현 간단, 성능 저하 없음	예측 실패 시 성능 저 하
ALT	분기 발생 예측	분기 발생할 것으로 가정	성능 저하 없음	예측 실패 시 큰 성능 저하
LTP	반복 루프 종료 예 측	루프 패턴 분석, 종료 시점 예측	성능 크게 향상	예측 실패 시 성능 저 하

분기 예측은 파이프라인의 효율성을 높이고 제어 해저드로 인한 성능 저하를 줄이기 위한 중요한 기술이다. 정적 분기 예측은 단순하고 구현이 용이하지만, 정확도가 떨어질 수 있다. 반면, 동적 분기 예측은 실행 시간 정보를 활용하여 보다 정확한 예측을 수행할 수 있지만, 추가적인 하드웨어와 복잡한 로직이 필요하다. 다양한 분기 예측 기법을 통해 파이프라인의 성능을 극대화하고, 제어 해저드로 인한 영향을 최소화할 수 있다.

2.5.3.1 구조적 해저드(Structural Hazard)



[6]Structural Hazard

구조적 해저드(Structural Hazard)는 파이프라인 프로세서에서 동일한 자원을 동시에 여러 명령어가 사용하려고 할 때 발생하는 문제를 말한다. 이는 파이프라인의 성능을 저하시킬 수 있으며, 자원 충돌을 피하기 위해 다양한 해결 방법이 필요하다.

구조적 해저드의 정의

구조적 해저드는 파이프라인에서 동시에 실행되는 여러 명령어가 동일한 하드웨어 자원(예: 메모리, 레지스터 파일, 버스 등)을 동시에 사용하려고 할 때 발생한다. 이는 특정 자원을 두 개 이상의 명령어가 동시에 사용하려고 할 때 자원 충돌(Resource Conflict)로 인해 발생하며, 파이프라인의 효율성을 저하시킨다.

구조적 해저드의 발생 원인

1. 자원 공유: 파이프라인의 여러 단계에서 동일한 자원을 공유할 때 발생한다. 예를 들어, 명령어 인출 단계(IF)와 메모리 접근 단계(MEM)가 동일한 메모리를 사용하려 할 때 발생한다.
2. 자원 부족: 파이프라인이 여러 자원을 동시에 사용할 수 있는 충분한 하드웨어를 갖추지 못한 경우 발생한다.
3. 동시 접근: 두 개 이상의 명령어가 동일한 사이클 내에 동일한 자원에 접근하려고 할 때 발생한다.

2.5.3.2 Structural Hazard의 해결 방법

구조적 해저드를 해결하기 위해 몇 가지 기법이 사용된다. 대표적인 방법으로는 모든 명령어의 단계 개수를 일치시켜 같은 자원의 소모를 예방하는 방법과 메모리의 분리(IF, MEM)가 있다.

1. 모든 명령어의 단계 개수 일치

정의: 모든 명령어의 단계 개수를 일치시키는 것은 파이프라인 내의 모든 명령어가 동일한 수의 단계를 거쳐 실행되도록 하는 방법이다. 이를 통해 각 단계에서 동일한 자원을 사용할 수 있도록 조정한다.

작동 원리

- 파이프라인의 각 단계에서 필요한 자원을 명령어별로 동일하게 분배한다.
- 모든 명령어가 동일한 단계를 거치게 함으로써 자원 충돌을 피한다.

- 예를 들어, 5단계 파이프라인(명령어 인출, 명령어 디코드, 실행, 메모리 접근, 쓰기 백)을 구성하여 모든 명령어가 동일한 단계를 거치도록 한다.

장점: 자원 충돌을 최소화하여 파이프라인의 효율성을 높일 수 있다.

단점: 특정 명령어에 대해 불필요한 단계가 추가될 수 있어, 전체 파이프라인의 효율성이 저하될 수 있다.

2. 메모리의 분리 (IF, MEM)

정의: 메모리의 분리는 명령어 인출 단계(IF)와 메모리 접근 단계(MEM)에서 사용하는 메모리를 분리하여 자원 충돌을 피하는 방법이다.

작동 원리

- 명령어 인출 단계와 메모리 접근 단계에서 사용하는 메모리를 물리적으로 분리한다.
- 예를 들어, 명령어 인출을 위한 전용 메모리와 데이터 접근을 위한 전용 메모리를 각각 사용한다.
- 명령어 인출 단계에서는 전용 명령어 캐시를 사용하고, 메모리 접근 단계에서는 데이터 캐시를 사용하여 자원 충돌을 피한다.

장점: 메모리 접근 충돌을 방지하여 파이프라인의 성능을 최적화할 수 있다.

단점: 추가적인 메모리 자원이 필요하여 하드웨어 비용이 증가할 수 있다.

구조적 해저드는 파이프라인 프로세서에서 동일한 자원을 동시에 사용하려는 명령어들로 인해 발생하는 문제로, 파이프라인의 성능을 저하시킬 수 있다. 이를 해결하기 위해 모든 명령어의 단계 개수를 일치시키거나, 메모리를 분리하여 자원 충돌을 피하는 방법이 사용된다. 이러한 해결 방법을 통해 파이프라인의 효율성을 극대화하고, 명령어 처리의 지연을 최소화할 수 있다. 구조적 해저드를 효과적으로 관리함으로써 파이프라인 프로세서의 성능을 최적화할 수 있을 것이다.

2.6 Data path

파이프라인의 데이터패스는 명령어가 파이프라인 스테이지를 통과하는 동안 데이터의 흐름을 관리하는 구성 요소로, 레지스터, ALU(Arithmetic Logic Unit), 연산 장치 등 다양한 하드웨어 컴포넌트로 구성된다. 이러한 데이터패스 구성 요소들은 파이프라인의 각 스테이지에서 필요한 데이터를 전달하고 연산을 수행하는 역할을 한다. 명령어가 파이프라인을 통과하면서 필요한 데이터는 레지스터를 통해 전달되고, ALU나 메모리 등의 하드웨어 컴포넌트에서 연산과 데이터 액세스가 이루어진다. 각 파이프라인 스테이지는 특정 작업을 수행하며, 데이터는 이러한 스테이지를 거치면서 처리된다. 데이터패스는 파이프라인의 성능을 향상시키고, 명령어 처리의 효율성을 극대화하는 데 기여한다. 포워딩(Forwarding)과 스톨링(Stalling)과 같은 기법은 데이터 위험을 줄여 파이프라인의 원활한 작동을 보장한다. 또한, 제어 유닛은 각 스테이지에서 필요한 제어 신호를 생성하여 명령어의 흐름을 효과적으로 관리한다. 포워딩은 데이터가 필요할 때 이전 명령어의 결과를 레지스터를 거치지 않고 직접 전달하는 방법으로, 데이터 위험을 줄여 파이프라인의 성능을 향상시킨다. 스톨링은 위험이 해결될 때까지 파이프라인의 진행을 멈추는 방법으로, 파이프라인의 효율은 떨어지지만 데이터 일관성을 유지하는 데 중요하다. 데이터패스는 명령어가 파이프라인 스테이지를 통과하는 동안 데이터의 흐름을 관리하는 중요한 구성 요소이다. 레지스터, ALU, 메모리, 데이터 선택기, 제어 유닛 등의 하드웨어 컴포넌트로 구성된 데이터패스는 파이프라인의 성능을 극대화하고 명령어 처리의 효율성을 높이는 데 기여한다. 데이터 위험을 해결하기 위한 다양한 기법들은 파이프라인의 원활한 작동을 보장하며, 이를 통해 효율적인 명령어 처리를 가능하게 한다.

3. Program

지금부터 파이프라인의 핵심 원칙들을 반영하여 개발한 프로그램에 대한 설명을 이어가도록 하겠다.

3.1 main 함수

main 함수는 CPU 시뮬레이션의 핵심적인 실행 루틴을 담당하며, CPU와 메모리를 초기화하고 프로그램을 로드한 후 각 사이클 동안 CPU의 다양한 단계를 수행한다. 이 과정에서 각 단계의 결과를 출력하고, 최종적으로 시뮬레이션 결과를 출력한다. 이 함수는 CPU와 메모리 구조체를 초기화하고, 프로그램을 로드하며, 각 사이클 동안 CPU의 주요 단계들을 실행한다.

```

int main() {
    CPU_Struct cpu_struct;
    Memory_Structure memory;
    char* input_file = "C:\\Users\\user\\Desktop\\Single cycle\\simple3.bin";

    cpu_struct.data_hazard_detected = 1; // Default: forwarding
    cpu_struct.control_hazard_detected = 1; // Default: ANT

    cpu_init(&cpu_struct);
    init_memory(&memory, MEM_SIZE, 0x0, 0x00100000, 0x00100001, 0x01001000);

    if(load_program(&memory, input_file)) return 0;

    while (cpu_struct.reg.program_counter != 0xFFFFFFFF) {
        mem_access(&cpu_struct, &memory);
        write_back(&cpu_struct, &memory);
        print_cycle(&cpu_struct);
        int tmp_pc = cpu_struct.reg.program_counter;
        cpu_fetch(&cpu_struct, &memory);
        print_fetch(&cpu_struct, tmp_pc);
        cpu_decode(&cpu_struct);
        print_decode(&cpu_struct);
        cpu_execute(&cpu_struct);
        print_execute(&cpu_struct);
        print_memory(&cpu_struct);
        print_write_back(&cpu_struct);
        detect_hazards(&cpu_struct);
        print_fwd(&cpu_struct);
        latch_update(&cpu_struct);
    }
    print_results(&cpu_struct);

    return 0;
}

```

[7] main 함수

CPU 구조체인 `cpu_struct`와 메모리 구조체인 `memory`를 선언하고, 실행할 프로그램의 바이너리 파일 경로를 지정한다. 데이터 및 제어 해저드 감지 플래그를 초기값으로 설정한다. `cpu_init` 함수를 호출하여 CPU 구조체를 초기화하고, `init_memory` 함수를 호출하여 메모리 구조체를 초기화한다. `MEM_SIZE`, `0x0`, `0x00100000`, `0x00100001`, `0x01001000`은 각각 메모리 크기, 텍스트 시작 주소, 텍스트 끝 주소, 데이터 시작 주소 및 데이터 끝 주소를 의미한다. 그후 지정된 바이너리 파일에서 프로그램을 메모리에 로드한다. 파일을 로드하는 데 실패하면 프로그램을 종료한다. 프로그램 카운터가 `0xFFFFFFFF`가 될 때까지 CPU 사이클을 반복한다. 각 사이클에서 다음의 단계들이 수행된다.

1. 메모리 접근 단계: `mem_access` 함수 호출
2. 쓰기 뒤 단계: `write_back` 함수 호출

3. 사이클 출력: 현재 사이클 정보를 출력하는 `print_cycle` 함수 호출
4. 명령어 인출: `cpu_fetch` 함수 호출
5. 명령어 인출 출력: `print_fetch` 함수 호출
6. 명령어 디코딩: `cpu_decode` 함수 호출
7. 명령어 디코딩 출력: `print_decode` 함수 호출
8. 명령어 실행: `cpu_execute` 함수 호출
9. 명령어 실행 출력: `print_execute` 함수 호출
10. 메모리 단계 출력: `print_memory` 함수 호출
11. 쓰기 뒤 단계 출력: `print_write_back` 함수 호출
12. 해저드 감지: `detect_hazards` 함수 호출
13. 포워딩 출력: `print_fwd` 함수 호출
14. 래치 업데이트: `latch_update` 함수 호출

시뮬레이션이 종료된 후 최종 결과를 출력하는 `print_results` 함수를 호출한다.

3.2 Latch

본 프로그램에서 래치(Latch)는 파이프라인의 핵심적인 역할을 담당한다. 파이프라인은 명령어를 여러 단계로 나누어 각 단계에서 병렬로 처리함으로써 CPU의 성능을 향상시키는 중요한 기법이다. 앞서 서론에서 말했듯이 래치는 이러한 파이프라인 단계 사이에서 중간 결과와 제어 신호를 저장하는 저장 요소로서, 데이터가 파이프라인을 통해 흐를 때 임시 저장소의 역할을 한다.

3.2.1 IFID Latch

IFID 래치는 명령어 인출(IF: Instruction Fetch) 단계와 명령어 디코딩(ID: Instruction Decode) 단계 사이에서 데이터와 제어 신호를 저장하는 저장 요소이다. 이 래치는 파이프라인의 중간 결과와 제어 신호를 저장하며, 데이터가 파이프라인을 통해 흐르는 동안 임시 저장소로서의 역할을 한다. 본 구현에서는 각 래치가 2개의 요소로 이루어진 배열로 구성되어 있으며, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다.

IFID 래치는 명령어 인출 단계에서 디코딩 단계로 데이터를 전달하기 위한 임시 저장소이다. 명령어 인출 단계에서 읽어들이는 명령어와 프로그램 카운터(PC) 값을 저장하며, 이러한 값들은 디코딩 단계에서 사용된다. IFID 래치는 명령어와 관련된 제어 신호를 포함하여 명령어가 올바르게 디코딩되고 실행될 수 있도록 한다.

코드에서 IFID 래치는 IFID_Latch 구조체로 정의되어 있으며, 이는 다음과 같은 요소들을 포함하고 있다.

```
typedef struct _IFID_Latch {  
    unsigned int valid;  
    unsigned int data;  
    unsigned int next_pc;  
    unsigned int pre_taken;  
    unsigned int branch_history_index;  
    unsigned int branch_history_found;  
} IFID_Latch;
```

[8] IFID Latch

- valid: 래치가 유효한 데이터를 포함하고 있는지 여부를 나타낸다.
- data: 인출된 명령어를 저장한다.
- next_pc: 다음 프로그램 카운터 값을 저장한다.
- pre_taken: 이전 단계에서 예측된 분기 여부를 저장한다.
- branch_history_index: 분기 예측을 위한 분기 히스토리 인덱스를 저장한다.
- branch_history_found: 분기 히스토리 예측이 발견되었는지 여부를 저장한다.

IFID 래치의 초기화 및 사용

cpu_init 함수에서 IFID 래치는 초기화된다. 각 래치는 2개의 요소로 이루어진 배열로, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다. IFID 래치는 매 사이클마다 업데이트된다. **cpu_fetch** 함수에서 명령어 인출 단계가 수행되며, 이 단계에서 인출된 명령어와 프로그램 카운터 값이 IFID 래치에 저장된다.

IFID 래치의 업데이트

매 사이클이 끝날 때마다 래치는 업데이트되며, 이전 단계의 출력을 현재 단계의 입력으로 설정한다. **latch_update** 함수는 각 래치를 업데이트하여 파이프라인의 데이터 흐름을 유지한다.

IFID 래치는 명령어 인출 단계와 명령어 디코딩 단계 사이에서 데이터와 제어 신호를 저장하는 중요한 역할을 한다. 이는 명령어가 파이프라인을 통해 흐를 때 임시 저장소로 사용되며, 파이프라인의 원활한 작동을 보장한다. 본 구현에서는 IFID 래치가 2개의 요소로 이루어진 배열로 구성되어 있으며, 매 사이클마다 업데이트되어 다음 단계에 필요한 데이터를 제공한다.

3.2.2 IDEX Latch

IDEX 래치는 명령어 디코딩(ID: Instruction Decode) 단계와 명령어 실행(EX: Execution) 단계 사이에서 데이터와 제어 신호를 저장하는 저장 요소이다. 이 래치는 파이프라인의 중간 결과와 제어 신호를 저장하며, 데이터가 파이프라인을 통해 흐르는 동안 임시 저장소로서의 역할을 한다. 본 구현에서는 각 래치가 2개의 요소로 이루어진 배열로 구성되어 있으며, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다. IDEX 래치는 명령어 디코딩 단계에서 명령어 실행 단계로 데이터를 전달하기 위한 임시 저장소이다. 디코딩 단계에서 해독된 명령어와 관련된 제어 신호, 레지스터 값, 즉시값 등의 데이터를 저장하며, 이러한 값들은 실행 단계에서 사용된다. IDEX 래치는 명령어가 올바르게 실행될 수 있도록 필요한 모든 정보를 포함한다.

코드에서 IDEX 래치는 `IDEX_Latch` 구조체로 정의되어 있으며, 이는 다음과 같은 요소들을 포함하고 있다.

```
typedef struct _IDEX_Latch {
    unsigned int valid;
    unsigned int read_data1;
    unsigned int read_data2;
    unsigned int opcode;
    unsigned int source_register1;
    unsigned int source_register2;
    unsigned int immediate_value;
    unsigned int destination_register;
    unsigned int shift_amount;
    unsigned int sign_extended_data;
    unsigned int zero_extended_data;
    unsigned int lui_shifted_data;
    unsigned int branch_history_index;
    unsigned int branch_history_found;
    unsigned int function;
    unsigned int next_pc;
    unsigned int address;
    unsigned int pre_taken;
    Control_Unit control;
} IDEX_Latch;
```

[9] IDEX Latch

- valid: 래치가 유효한 데이터를 포함하고 있는지 여부를 나타낸다.
- read_data1: 첫 번째 소스 레지스터에서 읽어들이는 데이터를 저장한다.
- read_data2: 두 번째 소스 레지스터에서 읽어들이는 데이터를 저장한다.
- opcode: 명령어의 opcode를 저장한다.
- source_register1: 첫 번째 소스 레지스터의 번호를 저장한다.
- source_register2: 두 번째 소스 레지스터의 번호를 저장한다.
- immediate_value: 즉시값을 저장한다.
- destination_register: 목적 레지스터의 번호를 저장한다.
- shift_amount: 시프트 연산에 사용되는 시프트 양을 저장한다.
- sign_extended_data: 부호 확장된 즉시값을 저장한다.
- zero_extended_data: 제로 확장된 즉시값을 저장한다.
- lui_shifted_data: 상위 16비트가 설정된 즉시값을 저장한다.
- branch_history_index: 분기 예측을 위한 분기 히스토리 인덱스를 저장한다.
- branch_history_found: 분기 히스토리 예측이 발견되었는지 여부를 저장한다.
- function: R형 명령어의 함수 필드를 저장한다.

- next_pc: 다음 프로그램 카운터 값을 저장한다.
- address: J형 명령어의 주소 필드를 저장한다.
- pre_taken: 이전 단계에서 예측된 분기 여부를 저장한다.
- control: 제어 신호를 저장하는 구조체를 포함한다.

IDEX 래치의 초기화 및 사용

cpu_init 함수에서 IDEX 래치는 초기화된다. 각 래치는 2개의 요소로 이루어진 배열로, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다. IDEX 래치는 매 사이클마다 업데이트된다. **cpu_decode** 함수에서 명령어 디코딩 단계가 수행되며, 이 단계에서 해독된 명령어와 관련된 데이터와 제어 신호가 IDEX 래치에 저장된다.

IDEX 래치는 명령어 디코딩 단계와 명령어 실행 단계 사이에서 중요한 역할을 하며, 명령어의 실행을 위해 필요한 모든 데이터를 저장한다. 이 래치는 파이프라인이 원활하게 작동하도록 보장하며, CPU의 성능을 향상시키는 데 기여한다. 본 프로그램에서 IDEX 래치는 코드 내에서 효과적으로 초기화되고, 매 사이클마다 업데이트되며, 이를 통해 파이프라인의 각 단계가 올바르게 실행될 수 있도록 지원한다.

3.2.3 EX MEM Latch

```
typedef struct _EXMEM_Latch {
    unsigned int valid;
    unsigned int read_data1;
    unsigned int read_data2;
    unsigned int branch_target;
    unsigned int destination_register;
    unsigned int alu_result;
    unsigned int next_pc;
    struct {
        unsigned int zero;
        unsigned int negative;
        unsigned int greater_than_zero;
        unsigned int less_equal_zero;
    } flags;
    Control_Unit control;
} EXMEM_Latch;
```

EXMEM 래치는 명령어 실행(EX: Execution) 단계와 메모리 접근(MEM: Memory Access) 단계 사이에서 데이터와 제어 신호를 저장하는 저장 요소이다. 이 래치는 파이프라인의 중간 결과와 제어 신호를 저장하며, 데이터가 파이프라인을 통해 흐르는 동안 임시 저장소로서의 역할을 한다. 본 구현에서는 각 래치가 2개의 요소로 이루어진 배열로 구성되어 있으며, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다. EXMEM 래치는 명령어 실행 단계에서 메모리 접근 단계로 데이터를 전달하기 위한 임시 저장소이다. 실행 단계에서 수행된 연산 결과와 관련된 제어 신호, 레지스터 값 등을 저장하며, 이러한 값들은 메모리 접근 단계에서 사용된다. EXMEM 래치는 명령어가 올바르게 메모리 접근 및 저장을 수행할 수 있도록 필요한 모든 정보를 포함한다.

코드에서 EXMEM 래치는 EXMEM_Latch 구조체로 정의되어 있으며, 이는 다음과 같은 요소들을 포함하고 있다.

- valid: 래치가 유효한 데이터를 포함하고 있는지 여부를 나타낸다.
- read_data1: 첫 번째 소스 레지스터에서 읽어들이는 데이터를 저장한다.
- read_data2: 두 번째 소스 레지스터에서 읽어들이는 데이터를 저장한다.
- branch_target: 분기 명령어의 목표 주소를 저장한다.
- destination_register: 목적 레지스터의 번호를 저장한다.
- alu_result: ALU 연산 결과를 저장한다.
- next_pc: 다음 프로그램 카운터 값을 저장한다.
- flags: ALU 연산의 결과 플래그를 저장한다.
 - zero: 연산 결과가 0인지 여부를 나타낸다.
 - negative: 연산 결과가 음수인지 여부를 나타낸다.
 - greater_than_zero: 연산 결과가 0보다 큰지 여부를 나타낸다.
 - less_equal_zero: 연산 결과가 0보다 작거나 같은지 여부를 나타낸다.
- control: 제어 신호를 저장하는 구조체를 포함한다.

EXMEM 래치의 초기화 및 사용

cpu_init 함수에서 EXMEM 래치는 초기화된다. 각 래치는 2개의 요소로 이루어진 배열로, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다. EXMEM 래치는 매 사이클마다 업데이트된다. **cpu_execute** 함수에서 명령어

실행 단계가 수행되며, 이 단계에서 수행된 연산 결과와 관련된 데이터와 제어 신호가 EXMEM 래치에 저장된다.

EXMEM 래치의 업데이트

매 사이클이 끝날 때마다 래치는 업데이트되며, 이전 단계의 출력을 현재 단계의 입력으로 설정한다. **latch_update** 함수는 각 래치를 업데이트하여 파이프라인의 데이터 흐름을 유지한다.

EXMEM 래치는 명령어 실행 단계와 메모리 접근 단계 사이에서 중요한 역할을 하며, 명령어의 올바른 메모리 접근 및 연산 결과 저장을 위해 필요한 모든 데이터를 저장한다. 이 래치는 파이프라인이 원활하게 작동하도록 보장하며, CPU의 성능을 향상시키는 데 기여한다. 본 프로그램에서 EXMEM 래치는 코드 내에서 효과적으로 초기화되고, 매 사이클마다 업데이트되며, 이를 통해 파이프라인의 각 단계가 올바르게 실행될 수 있도록 지원한다.

3.2.4 MEM WB Latch

MEMWB 래치는 메모리 접근(MEM: Memory Access) 단계와 쓰기 뒤(WB: Write Back) 단계 사이에서 데이터와 제어 신호를 저장하는 저장 요소이다. 이 래치는 파이프라인의 중간 결과와 제어 신호를 저장하며, 데이터가 파이프라인을 통해 흐르는 동안 임시 저장소로서의 역할을 한다. 본 구현에서는 각 래치가 2개의 요소로 이루어진 배열로 구성되어 있으며, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다. MEMWB 래치는 메모리 접근 단계에서 쓰기 뒤 단계로 데이터를 전달하기 위한 임시 저장소이다. 메모리 접근 단계에서 메모리에 접근하여 읽거나 쓴 데이터, ALU 연산 결과, 목적 레지스터 번호, 제어 신호 등을 저장하며, 이러한 값들은 쓰기 뒤 단계에서 사용된다. MEMWB 래치는 명령어가 최종적으로 레지스터 파일에 데이터를 올바르게 쓰기 위해 필요한 모든 정보를 포함한다.

코드에서 MEMWB 래치는 MEMWB_Latch 구조체로 정의되어 있으며, 이는 다음과 같은 요소들을 포함하고 있다.

```
typedef struct _MEMWB_Latch {
    unsigned int valid;
    unsigned int destination_register;
    unsigned int alu_result;
    unsigned int read_data;
    unsigned int next_pc;
    Control_Unit control;
} MEMWB_Latch;
```

[11]MEM WB Latch

- valid: 래치가 유효한 데이터를 포함하고 있는지 여부를 나타낸다.
- destination_register: 목적 레지스터의 번호를 저장한다.
- alu_result: ALU 연산 결과를 저장한다.
- read_data: 메모리에서 읽어들이는 데이터를 저장한다.
- next_pc: 다음 프로그램 카운터 값을 저장한다.
- control: 제어 신호를 저장하는 구조체를 포함한다.

MEMWB 래치의 초기화 및 사용

cpu_init 함수에서 MEMWB 래치는 초기화된다. 각 래치는 2개의 요소로 이루어진 배열로, 첫 번째 요소는 이전 단계의 출력을 포함하고, 두 번째 요소는 다음 단계에 필요한 값을 제공한다. MEMWB 래치는 매 사이클마다 업데이트된다. **mem_access** 함수에서 메모리 접근 단계가 수행되며, 이 단계에서 수행된 메모리 접근 결과와 관련된 데이터와 제어 신호가 MEMWB 래치에 저장된다.

MEMWB 래치의 업데이트

매 사이클이 끝날 때마다 래치는 업데이트되며, 이전 단계의 출력을 현재 단계의 입력으로 설정한다. **latch_update** 함수는 각 래치를 업데이트하여 파이프라인의 데이터 흐름을 유지한다.

MEMWB 래치의 쓰기 뒤 단계

write_back 함수는 쓰기 뒤 단계에서 MEMWB 래치에 저장된 데이터를 사용하여 레지스터 파일에 값을 쓰는 역할을 한다.

MEMWB 래치는 메모리 접근 단계와 쓰기 뒤 단계 사이에서 중요한 역할을 하며, 명령어의 최종 결과를 레지스터 파일에 올바르게 쓰기 위해 필요한 모든 데이터를 저장한다. 이 래치는 파이프라인이 원활하게 작동하도록 보장하며, CPU의 성능을 향상시키는 데 기여한다. 본 프로그램에서 MEMWB 래치는 코드 내에서 효과적으로 초기화되고, 매 사이클마다 업데이트되며, 이를 통해 파이프라인의 각 단계가 올바르게 실행될 수 있도록 지원한다.

한 Cycle이 끝나면 latch_update 함수를 통해 latch를 업데이트 한다.

3.3.0 stage

파이프라인에는 5가지의 stage가 있다 이 코드에서도 이러한 방향을 통해 파이프라인이 구현이 되어있다.

3.3.1 IF

IF(Instruction Fetch) 단계는 CPU 파이프라인에서 명령어를 메모리로부터 가져오는 첫 번째 단계이다. 이 단계는 주로 프로그램 카운터(PC)를 사용하여 현재 실행할 명령어의 주소를 지정하고, 해당 주소로부터 명령어를 읽어오는 역할을 한다. IF 단계에서 읽어온 명령어는 이후 파이프라인 단계로 전달되어 디코딩되고 실행된다. 본 구현에서 IF 단계는 cpu_fetch 함수 내에서 수행된다.

IF 단계의 구성 요소

IF 단계는 다음과 같은 주요 구성 요소들로 이루어져 있다.

- 프로그램 카운터(PC): 현재 실행할 명령어의 주소를 저장하는 레지스터이다.
- 명령어 메모리: 명령어들이 저장된 메모리 공간이다.
- IFID 래치: 명령어 인출 단계에서 디코딩 단계로 데이터를 전달하기 위한 임시 저장소이다.

IF 단계는 cpu_fetch 함수 내에서 구현되어 있다. 이 함수는 다음과 같은 작업을 수행한다.

```

void cpu_fetch(CPU_Struct* cpu_struct, Memory_Structure* mem) {
    if (cpu_struct->reg.program_counter != 0xFFFFFFFF) {
        cpu_struct->ifid_latch[0].valid = 1;
        cpu_struct->ifid_latch[0].data = fetch_instruction_memory(mem, cpu_struct->reg.program_counter);
        cpu_struct->alu_control_unit.alu_control = 0;

        switch (cpu_struct->control_hazard_detected) {
            case 0: {
                cpu_struct->reg.program_counter += 4;
                cpu_struct->ifid_latch[0].next_pc = cpu_struct->reg.program_counter;
            } break; // detect and wait
            case 1:
                cpu_struct->reg.program_counter = cpu_struct->reg.program_counter + 4;
                cpu_struct->ifid_latch[0].next_pc = cpu_struct->reg.program_counter;
                break; // ant
            case 2: {
                int branch_history_index = (0x100 - 1);
                branch_history_index <= 2;
                branch_history_index &= cpu_struct->reg.program_counter;
                branch_history_index >= 2;
                cpu_struct->ifid_latch[0].branch_history_index = branch_history_index;
            }
        }
    }
}

```

[12] IF

프로그램 카운터 검사 및 명령어 인출

프로그램 카운터가 유효한지 검사하고, 메모리로부터 명령어를 인출하여 IFID 래치에 저장한다. 이와 함께 ALU 제어 신호를 초기화한다.

Control Hazard 처리

Control Hazard가 감지되면, 다양한 해저드 처리 방법에 따라 프로그램 카운터를 업데이트하고, IFID 래치에 다음 프로그램 카운터 값을 저장한다. 제어 해저드 처리 방식은 다음과 같다.

- case 0: 해저드를 감지하고 대기한다.
- case 1: 프로그램 카운터를 단순히 증가시킨다.
- case 2: 분기 히스토리를 참조하여 다음 프로그램 카운터를 결정한다.
- case 3: 동적 분기 예측을 사용하여 프로그램 카운터를 결정한다.

IF 단계는 CPU 파이프라인에서 중요한 역할을 한다. 이 단계는 프로그램 카운터를 기반으로 메모리에서 명령어를 가져와 IFID 래치에 저장하고, 프로그램 카운터를 업데이트한다. 제어 해저드가 감지되면 다양한 방법으로 프로그램 카운터를 처리하여

올바른 명령어 인출을 보장한다. 본 구현에서 IF 단계는 `cpu_fetch` 함수 내에서 수행되며, CPU의 성능을 최적화하고 파이프라인의 원활한 작동을 지원한다.

3.3.2 ID

ID(Instruction Decode) 단계는 CPU 파이프라인에서 명령어를 해독하고, 필요한 데이터를 추출하는 두 번째 단계이다. 이 단계는 IF(Instruction Fetch) 단계에서 가져온 명령어를 해독하여 명령어의 종류, 소스 레지스터, 목적 레지스터, 즉시 값 등의 정보를 추출한다. 또한, 이 단계에서는 제어 신호를 생성하여 파이프라인의 다음 단계로 전달된다. 본 구현에서 ID 단계는 `cpu_decode` 함수 내에서 수행된다.

ID 단계의 구성 요소

ID 단계는 다음과 같은 주요 구성 요소들로 이루어져 있다.

- IFID 래치: IF 단계에서 가져온 명령어와 관련된 데이터를 저장한다.
- 레지스터 파일: 명령어 실행에 필요한 레지스터 값을 저장하는 구조이다.
- IDEX 래치: ID 단계에서 추출한 데이터를 저장하여 다음 단계로 전달한다.
- 제어 유닛: 명령어의 종류에 따라 제어 신호를 생성한다.

```
void cpu_decode(CPU_Struct* cpu_struct) {
    if (cpu_struct->ifid_latch[1].valid) {
        unsigned int ins = cpu_struct->ifid_latch[1].data;
        R_type_instruction r_instruction;
        I_type_instruction i_instruction;
        J_type_instruction j_instruction;

        r_instruction.opcode = (ins >> 26) & 0x3F;
        r_instruction.source_register1 = (ins >> 21) & 0x1F;
        r_instruction.source_register2 = (ins >> 16) & 0x1F;
        r_instruction.destination_register = (ins >> 11) & 0x1F;
        r_instruction.shift_amount = (ins >> 6) & 0x1F;
        r_instruction.function = ins & 0x3F;
    }
}
```

[13] ID

명령어 해독 및 데이터 추출, 제어 신호 생성 및 레지스터 파일 접근, IDEX 래치 업데이트
ID 단계는 CPU 파이프라인에서 중요한 역할을 한다. 이 단계는 IF 단계에서 인출된 명령어를 해독하고, 필요한 데이터를 추출하여 IDEX 래치에 저장한다. 또한, 제어 신호를

생성하여 명령어가 올바르게 실행될 수 있도록 한다. 본 구현에서 ID 단계는 `cpu_decode` 함수 내에서 수행되며, CPU의 성능을 최적화하고 파이프라인의 원활한 작동을 지원한다.

3.3.3 EX

EX(Execution) 단계는 CPU 파이프라인에서 명령어를 실제로 실행하는 세 번째 단계이다. 이 단계에서는 ALU(Arithmetic Logic Unit)를 사용하여 연산을 수행하거나, 주소 계산을 실행하는 등의 작업이 이루어진다. EX 단계는 이전 단계인 ID(Instruction Decode) 단계에서 추출된 데이터를 바탕으로 연산을 수행하며, 그 결과를 다음 단계로 전달한다. 본 구현에서 EX 단계는 `cpu_execute` 함수 내에서 수행된다.

EX 단계의 구성 요소

EX 단계는 다음과 같은 주요 구성 요소들로 이루어져 있다.

- IDEX 래치: ID 단계에서 전달된 데이터를 저장한다.
- ALU: 산술 및 논리 연산을 수행하는 장치이다.
- EXMEM 래치: EX 단계에서 수행된 연산 결과를 저장하여 MEM 단계로 전달한다.
- 제어 유닛: 명령어의 종류에 따라 제어 신호를 생성한다.

```
void cpu_execute(CPU_Struct* cpu_struct) {
    if (cpu_struct->idex_latch[1].valid) {
        cpu_struct->exmem_latch[0].control = cpu_struct->idex_latch[1].control;

        alu_ctrl_ops(&(cpu_struct->alu_control_unit));

        int fwd_data = *(cpu_struct->forwarding_mux[cpu_struct->memwb_latch[1].control.mem_to_reg]);
        unsigned int read_mux2[3] = { cpu_struct->idex_latch[1].read_data2, fwd_data, cpu_struct->exmem_latch[1].alu_result };
        unsigned int read_mux1[3] = { cpu_struct->idex_latch[1].read_data1, fwd_data, cpu_struct->exmem_latch[1].alu_result };

        cpu_struct->exmem_latch[0].read_data1 = read_mux1[cpu_struct->forwarding_unit[0]];
        cpu_struct->exmem_latch[0].read_data2 = read_mux2[cpu_struct->forwarding_unit[1]];
    }
}
```

[14] EX

제어 신호 및 ALU 연산, 포워딩 및 ALU 입력 설정, ALU 연산 수행 및 결과 저장, 분기 결정 및 프로그램 카운터 업데이트

EX 단계는 CPU 파이프라인에서 중요한 역할을 한다. 이 단계는 명령어의 연산을 실제로 수행하고, 그 결과를 EXMEM 래치에 저장하여 다음 단계로 전달한다. 또한, 분기 명령어의 조건을 평가하여 프로그램 카운터를 업데이트한다. 본 구현에서 EX 단계는

cpu_execute 함수 내에서 수행되며, CPU의 성능을 최적화하고 파이프라인의 원활한 작동을 지원한다.

3.3.4 MEM

MEM(Memory Access) 단계는 CPU 파이프라인에서 명령어의 메모리 접근을 처리하는 네 번째 단계이다. 이 단계에서는 명령어의 종류에 따라 메모리에 데이터를 읽거나 쓰는 작업이 수행된다. MEM 단계는 EX(Execution) 단계에서 계산된 메모리 주소를 바탕으로 메모리 접근을 수행하며, 그 결과를 다음 단계인 WB(Write Back) 단계로 전달한다. 본 구현에서 MEM 단계는 mem_access 함수 내에서 수행된다.

MEM 단계의 구성 요소

MEM 단계는 다음과 같은 주요 구성 요소들로 이루어져 있다.

- EXMEM 래치: EX 단계에서 전달된 데이터를 저장한다.
- 메모리: 명령어와 데이터를 저장하는 메모리 공간이다.
- MEMWB 래치: MEM 단계에서 수행된 메모리 접근 결과를 저장하여 WB 단계로 전달한다.
- 제어 유닛: 명령어의 종류에 따라 제어 신호를 생성한다.

```
void mem_access(CPU_Struct* cpu_struct, Memory_Structure* mem) {
    if (cpu_struct->exmem_latch[1].valid) {
        cpu_struct->memwb_latch[0].control = cpu_struct->exmem_latch[1].control;

        if (cpu_struct->exmem_latch[1].control.mem_write) {
            store_data_memory(mem, cpu_struct->exmem_latch[1].alu_result, cpu_struct->exmem_latch[1].read_data2, cpu_struct->exmem_latch[1].control)
        }

        if (cpu_struct->exmem_latch[1].control.mem_to_reg) {
            unsigned int lw = fetch_data_memory(mem, cpu_struct->exmem_latch[1].alu_result);
            unsigned int lb = (((cpu_struct->exmem_latch[1].control.extra >> 2) & 1) ? ((int)lw >> 24) : (lw >> 24));
            unsigned int lh = (((cpu_struct->exmem_latch[1].control.extra >> 2) & 1) ? ((int)lw >> 16) : (lw >> 16));

            cpu_struct->mux_load[3] = lw;
            cpu_struct->mux_load[0] = lb;
            cpu_struct->mux_load[1] = lh;
            cpu_struct->mux_load[2] = 0;

            cpu_struct->memwb_latch[0].read_data = cpu_struct->mux_load[(cpu_struct->exmem_latch[1].control.extra & 0x3)];
        }
    }
}
```

[14] MEM

메모리 쓰기, 메모리 읽기, 결과 저장

MEM 단계는 CPU 파이프라인에서 중요한 역할을 한다. 이 단계는 명령어의 메모리 접근을 처리하며, 메모리 읽기와 쓰기 작업을 수행한다. MEM 단계는 EX 단계에서 계산된

메모리 주소를 바탕으로 메모리 접근을 수행하며, 그 결과를 MEMWB 래치에 저장하여 다음 단계로 전달한다. 본 구현에서 MEM 단계는 mem_access 함수 내에서 수행되며, CPU의 성능을 최적화하고 파이프라인의 원활한 작동을 지원한다.

3.3.5 WB

WB(Write Back) 단계는 CPU 파이프라인에서 연산 결과를 레지스터 파일에 쓰는 최종 단계이다. 이 단계에서는 메모리 접근(MEM) 단계에서 읽어온 데이터나 ALU(Arithmetic Logic Unit) 연산 결과를 레지스터 파일에 기록한다. WB 단계는 CPU의 연산 결과가 실제로 저장되는 단계로서, 프로그램의 상태를 갱신하는 역할을 한다. 본 구현에서 WB 단계는 write_back 함수 내에서 수행된다.

WB 단계의 구성 요소

WB 단계는 다음과 같은 주요 구성 요소들로 이루어져 있다.

- MEMWB 래치: MEM 단계에서 전달된 데이터를 저장한다.
- 레지스터 파일: 명령어 실행에 필요한 레지스터 값을 저장하는 구조이다.
- 제어 유닛: 명령어의 종류에 따라 제어 신호를 생성한다.

```
void write_back(CPU_Struct* cpu_struct, Memory_Structure* mem) {
    if (cpu_struct->memwb_latch[1].valid) {
        cpu_struct->mux_alu_result[0] = cpu_struct->memwb_latch[1].alu_result;
        cpu_struct->mux_alu_result[1] = cpu_struct->memwb_latch[1].read_data;

        cpu_struct->mux_member_PC[0] = cpu_struct->mux_alu_result[cpu_struct->memwb_latch[1].control.mem_to_reg];
        cpu_struct->mux_member_PC[1] = cpu_struct->memwb_latch[1].next_pc + 4;

        reg_file_write(&(cpu_struct->reg), cpu_struct->memwb_latch[1].destination_register, cpu_struct->mux_member_PC[cpu_struct->memwb_latch[1].
    }
}
```

[15] WB

유효성 검사 및 데이터 설정, 데이터 선택 및 레지스터 파일 쓰기

WB 단계는 CPU 파이프라인에서 최종 결과를 레지스터 파일에 쓰는 중요한 단계이다. 이 단계는 연산 결과를 레지스터 파일에 반영하여 프로그램의 상태를 갱신한다. 본 구현에서 WB 단계는 write_back 함수 내에서 수행되며, CPU의 연산 결과를 올바르게 저장하고, 파이프라인의 원활한 작동을 지원한다.

4.1 Detect and Wait

Detect and Wait 기법은 `cpu_struct.control_hazard_detected` 변수를 통해 제어된다. 이 변수는 분기 명령어를 처리할 때 사용되는 기법을 결정한다.

```
cpu_struct.control_hazard_detected = 1;
```

`cpu_fetch` 함수는 명령어를 메모리에서 가져와 파이프라인에 삽입하는 역할을 한다.

Detect and Wait 기법은 `control_hazard_detected` 변수가 0으로 설정되었을 때 동작한다. 이 경우, 분기 명령어가 감지되면 다음과 같은 절차를 따른다

1. 명령어 가져오기: 현재 프로그램 카운터(PC)에서 명령어를 가져와 `ifid_latch[0]`에 저장한다.
2. 파이프라인 멈춤: `control_hazard_detected` 변수가 0인 경우, `program_counter`를 단순히 4만큼 증가시켜 다음 명령어를 가져오도록 설정한다. 실제 분기 결과를 확인하기 전까지는 이 값이 유효하다.
3. 분기 결과 확인: 분기 명령어가 실제로 실행되어 분기 여부가 결정되기 전까지 파이프라인의 다른 명령어들은 진행을 멈춘다. 이는 `cpu_decode`, `cpu_execute` 등의 함수에서 추가적인 논리를 통해 제어될 수 있다.

4.2 ANT

코드에서 ANT 기법은 `cpu_struct.control_hazard_detected` 변수를 통해 제어된다. 이 변수는 분기 명령어를 처리할 때 사용되는 기법을 결정한다.

`cpu_fetch` 함수는 명령어를 메모리에서 가져와 파이프라인에 삽입하는 역할을 한다.

ANT 기법은 `control_hazard_detected` 변수가 1으로 설정되었을 때 동작한다. 이 경우, 분기 명령어가 감지되면 다음과 같은 절차를 따른다:

1. 명령어 가져오기: 현재 프로그램 카운터(PC)에서 명령어를 가져와 `ifid_latch[0]`에 저장한다.
2. 순차적 실행: `control_hazard_detected` 변수가 1인 경우, `program_counter`를 단순히 4만큼 증가시켜 다음 명령어를 가져오도록 설정한다. 이는 분기가 일어나지 않는다고 가정하고 다음 명령어를 실행하기 위함이다.

3. 분기 결과 확인: 실제로 분기 명령어가 실행될 때, 분기 여부가 확정되면 잘못된 예측에 의해 실행된 명령어들은 파이프라인에서 플러시(flush) 된다.

이는 `cpu_decode`, `cpu_execute` 등의 함수에서 추가적인 논리를 통해 제어될 수 있다.

ANT 기법은 매우 단순한 Branch Prediction 기법으로, 모든 분기 명령어가 분기를 하지 않는다고 가정하여 처리한다. 코드에서 `cpu_struct.control_hazard_detected` 변수를 통해 쉽게 제어할 수 있으며, 파이프라인의 분기 예측 문제를 해결하는 기본적인 접근 방식으로 사용될 수 있다. 그러나 분기 명령어가 자주 발생하는 환경에서는 더 복잡한 Branch Prediction 기법을 사용하는 것이 성능 향상에 도움이 된다.

4.3 ALT

코드에서 ALT 기법은 `cpu_struct.control_hazard_detected` 변수를 통해 제어된다. 이 변수는 분기 명령어를 처리할 때 사용되는 기법을 결정한다. `cpu_fetch` 함수는 명령어를 메모리에서 가져와 파이프라인에 삽입하는 역할을 한다. ALT 기법은 `control_hazard_detected` 변수가 2로 설정되었을 때 동작한다. 이 경우, 분기 명령어가 감지되면 다음과 같은 절차를 따른다:

1. 명령어 가져오기: 현재 프로그램 카운터(PC)에서 명령어를 가져와 `ifid_latch[0]`에 저장한다.
2. 분기 예측 인덱스 계산: 분기 예측 인덱스를 계산하여 현재 명령어 주소에 매핑된 분기 기록을 확인한다.
3. 분기 예측: 분기 기록이 존재하고, 현재 PC가 기록된 분기 명령어 주소와 일치하면, 분기 목표 주소로 PC를 업데이트한다.
4. 분기 실패 처리: 분기 기록이 없거나 현재 PC가 기록된 분기 명령어 주소와 일치하지 않으면, 순차적으로 다음 명령어를 실행한다.
5. 분기 결과 확인: 실제로 분기 명령어가 실행될 때, 분기 여부가 확정되면 잘못된 예측에 의해 실행된 명령어들은 파이프라인에서 플러시(flush) 된다. 이는 `cpu_decode`, `cpu_execute` 등의 함수에서 추가적인 논리를 통해 제어될 수 있다.

ALT 기법은 매우 단순한 Branch Prediction 기법으로, 모든 분기 명령어가 분기한다고 가정하여 처리한다. 코드에서 `cpu_struct.control_hazard_detected` 변수를 통해 쉽게 제어할 수 있으며, 파이프라인의 분기 예측 문제를 해결하는 기본적인 접근 방식으로 사용될 수 있다. 그러나 분기 명령어가 자주 발생하는 환경에서는 더 복잡한 Branch Prediction 기법을 사용하는 것이 성능 향상에 도움이 된다.

4.4 LTP

코드에서 LTP 기법은 `cpu_struct.control_hazard_detected` 변수를 통해 제어된다. 이 변수는 분기 명령어를 처리할 때 사용되는 기법을 결정한다. `cpu_fetch` 함수는 명령어를 메모리에서 가져와 파이프라인에 삽입하는 역할을 한다. LTP 기법은 `control_hazard_detected` 변수가 3으로 설정되었을 때 동작한다. 이 경우, 분기 명령어가 감지되면 다음과 같은 절차를 따른다:

1. 명령어 가져오기: 현재 프로그램 카운터(PC)에서 명령어를 가져와 `ifid_latch[0]`에 저장한다.
2. 분기 예측 인덱스 계산: 분기 예측 인덱스를 계산하여 현재 명령어 주소에 매핑된 분기 기록을 확인한다.
3. 분기 예측 확인: 분기 기록이 존재하고, 현재 PC가 기록된 분기 명령어 주소와 일치하면, 분기 예측 비트에 따라 다음 PC를 결정한다.
 - 예측 비트가 0: 분기하지 않는다고 예측하여 다음 명령어를 순차적으로 실행한다.
 - 예측 비트가 1: 분기한다고 예측하여 분기 목표 주소로 PC를 업데이트한다.
4. 분기 실패 처리: 분기 기록이 없거나 현재 PC가 기록된 분기 명령어 주소와 일치하지 않으면, 순차적으로 다음 명령어를 실행한다.
5. 분기 결과 확인: 실제로 분기 명령어가 실행될 때, 분기 여부가 확정되면 잘못된 예측에 의해 실행된 명령어들은 파이프라인에서 플러시(flush) 된다. 이는 `cpu_decode`, `cpu_execute` 등의 함수에서 추가적인 논리를 통해 제어될 수 있다.

LTP 기법은 이전의 분기 결과를 기반으로 분기 예측을 수행하여, 분기 예측 정확도를 높이는 방식이다. 코드에서 `cpu_struct.control_hazard_detected` 변수를 통해 쉽게 제어할 수 있으며, 파이프라인의 분기 예측 문제를 해결하는 효율적인 접근 방식 중

하나이다. 그러나 분기 패턴이 복잡하거나 자주 변경되는 환경에서는 더 복잡한 Branch Prediction 기법을 사용하는 것이 성능 향상에 도움이 된다.

4.5 BTB

BTB는 분기 명령어의 목표 주소를 저장하여, 분기 예측 시 빠르게 분기 목표 주소를 참조할 수 있게 하는 캐시이다. 이를 통해 분기 예측의 정확성을 높이고, 분기 명령어로 인한 파이프라인의 중단을 최소화할 수 있다.

```
typedef struct _Branch_History_Unit {  
    unsigned int branch_instruction_address;  
    unsigned int branch_target_address;  
    unsigned int prediction_bit;  
} Branch_History_Unit;
```

코드에서 BTB는 Branch_History_Unit 구조체와 이를 관리하는 배열 branch_history_unit을 통해 구현된다. 이 구조체는 분기 명령어의 주소, 목표 주소, 예측 비트를 포함하고 있다.

CPU 구조체는 다음과 같이 BTB를 포함하고 있다.

```
typedef struct _CPU_Struct {  
    Register_Struct reg;  
    IFID_Latch ifid_latch[2];  
    IDEX_Latch idex_latch[2];  
    EXMEM_Latch exmem_latch[2];  
    MEMWB_Latch memwb_latch[2];  
    Branch_History_Unit branch_history_unit[256];  
    ALU_Control_Unit alu_control_unit;  
    ALU_Struct alu_struct;  
    unsigned int data_hazard_detected;  
    unsigned int control_hazard_detected;  
    unsigned int forwarding_unit[2];  
    unsigned int* forwarding_mux[2];  
    unsigned int* mux_write_reg[3];  
    unsigned int* mux_alu_src[4];  
    unsigned int mux_branchtaken[2];  
    unsigned int mux_load[4];  
    unsigned int mux_alu_result[2];  
    unsigned int mux_branch[4];  
    unsigned int mux_jump[2];  
    unsigned int mux_jump_register[2];  
    unsigned int mux_member_PC[2];  
} CPU_Struct;
```

BTB는 CPU 초기화 과정에서 초기화된다. 모든 BTB 엔트리는 기본값으로 설정된다. BTB는 `cpu_fetch` 함수에서 사용된다. 분기 예측을 위해 BTB에 저장된 정보를 참조하여 다음 프로그램 카운터(PC)를 결정한다.

이 코드에서 BTB 예측은 다음과 같은 단계로 이루어진다.

BTB 인덱스 계산: 현재 프로그램 카운터(PC)를 기반으로 BTB 인덱스를 계산한다.

1. BTB 조회: 계산된 인덱스를 사용하여 BTB를 조회하고, 해당 분기 명령어가 있는지 확인한다.
2. 분기 예측: BTB에 분기 명령어가 저장되어 있는 경우, 해당 목표 주소로 PC를 업데이트합니다. 그렇지 않은 경우, 다음 명령어로 진행한다.

분기 명령어가 실행될 때, BTB는 다음과 같이 갱신된다. 이는 실제 분기 명령어의 실행 결과를 기반으로 한다. BTB는 분기 명령어의 목표 주소를 캐싱하여, 분기 예측의 성능을 향상시키는 중요한 기법이다. 코드에서는 `Branch_History_Unit` 구조체와 이를 관리하는 배열을 통해 BTB를 구현하고 있으며, 분기 명령어의 예측과 갱신 과정을 효율적으로 처리하고 있다. 이를 통해 분기 명령어로 인한 파이프라인 중단을 최소화하고, 전체적인 성능을 향상시킬 수 있다. 이와 같이 포워딩은 파이프라인에서 데이터 의존성을 해결하고, CPU의 효율을 높이는 중요한 역할을 한다. Forwarding을 통해 불필요한 스톨을 줄이고, 연산 결과를 신속하게 다음 명령어에 전달할 수 있다. 이러한 과정은 전체 시스템의 성능을 최적화하는 데 기여한다.

4.6 Forwarding

Forwarding은 주로 `cpu_execute` 함수와 `detect_hazards` 함수에서 구현되어 있다. 이 함수들은 데이터 해저드를 감지하고, 필요한 경우 포워딩을 수행하는 역할을 한다. CPU 구조체 내에서 포워딩 관련 변수를 초기화한다. 포워딩 유닛은 두 개의 포워딩 경로를 관리하며, 초기에는 모두 0으로 설정한다. 포워딩 유닛은 `detect_hazards` 함수에서 데이터 해저드를 감지하고, 필요할 때 포워딩을 수행한다.

5.1 실행결과

```
char* input_file = "C:\\Users\\user\\Desktop\\Single cycle\\simple4.bin";
```

다음에 입력받을 바이너리 파일을 받은 후 실행하면

(Data Hazard 및 Control Hazard 감지된 경우 기준)

```
cpu_struct.data_hazard_detected = 1; // Default: forwarding  
cpu_struct.control_hazard_detected = 1; // Default: ANT
```

```
=====PROGRAM RESULT=====  
Return value (R[2]) :          0  
Total Cycle :                  10  
Executed 'R' instruction :      3  
Executed 'I' instruction :      4  
Executed 'J' instruction :      0  
Number of Branch Taken :        0  
Number of Memory Access Instruction : 2  
Number of STALL :              10  
=====
```

simple 결과

```
=====PROGRAM RESULT=====  
Return value (R[2]) :          100  
Total Cycle :                  12  
Executed 'R' instruction :      3  
Executed 'I' instruction :      7  
Executed 'J' instruction :      0  
Number of Branch Taken :        0  
Number of Memory Access Instruction : 4  
Number of STALL :              10  
=====
```

simple2 결과

```
=====PROGRAM RESULT=====  
Return value (R[2]) :          5050  
Total Cycle :                  1535  
Executed 'R' instruction :      104  
Executed 'I' instruction :      920  
Executed 'J' instruction :       1  
Number of Branch Taken :        101  
Number of Memory Access Instruction : 613  
Number of STALL :              717  
=====
```

simple3 결과


```

=====PROGRAM RESULT=====
Return value (R[2]) :          55
Total Cycle :                  294
Executed 'R' instruction :      60
Executed 'I' instruction :      153
Executed 'J' instruction :       11
Number of Branch Taken :         9
Number of Memory Access Instruction : 100
Number of STALL :               73
=====

```

simple4 결과

```

=====PROGRAM RESULT=====
Return value (R[2]) :          1
Total Cycle :                  1292
Executed 'R' instruction :      222
Executed 'I' instruction :      637
Executed 'J' instruction :       65
Number of Branch Taken :        45
Number of Memory Access Instruction : 486
Number of STALL :               325
=====

```

gcd 결과

```

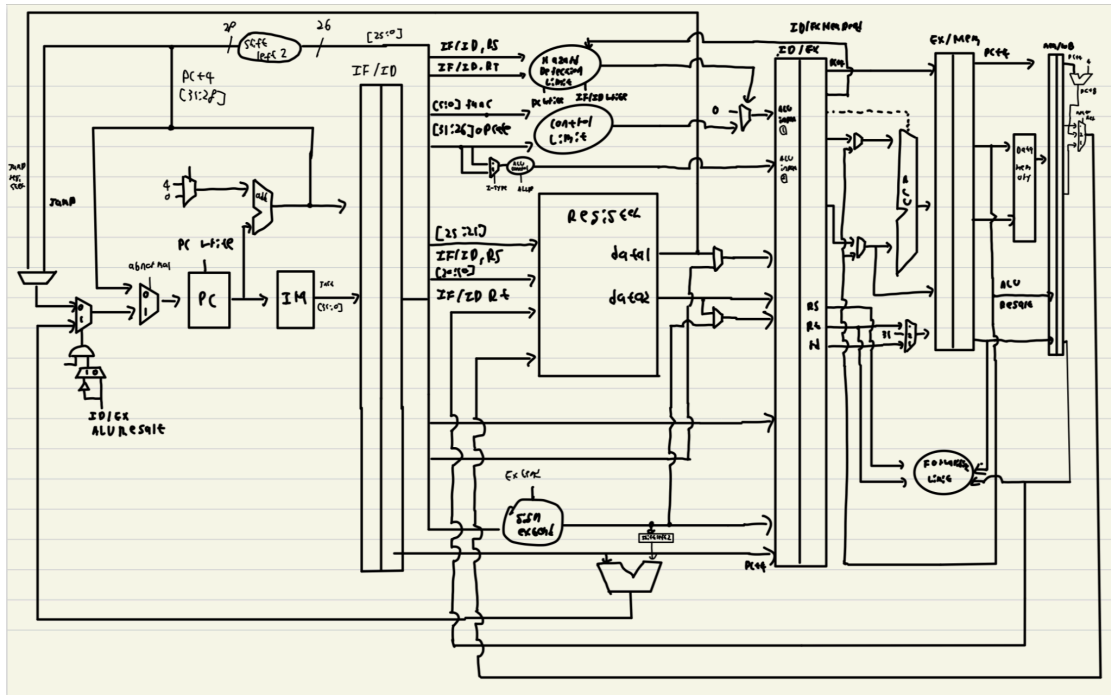
=====PROGRAM RESULT=====
Return value (R[2]) :          55
Total Cycle :                  3171
Executed 'R' instruction :      546
Executed 'I' instruction :      1697
Executed 'J' instruction :       164
Number of Branch Taken :        54
Number of Memory Access Instruction : 1095
Number of STALL :               388
=====

```

fib 결과

다음과 같이 실행이 되는 것을 확인할 수 있다.

6.1 결론



직접 그린 파이프라인 데이터패스

직접 파이프라인의 데이터패스를 여러번 그려보며 이해 대한 이해를 바탕으로 프로젝트를 진행하며 겪은 도전 중 하나는 디버깅과 오류 수정이었다. 초기 설계에서 예상치 못한 문제들이 발생할 때마다, 문제의 원인을 찾고 해결책을 모색하는 과정이 필수적이었다. 예를 들어, 데이터 해저드와 제어 해저드를 처리하는 과정에서 발생하는 예상치 못한 상황들은 상당한 난관이었다. 이러한 문제들을 해결하기 위해 많은 시간을 투자해야 했고, 이를 통해 시스템적 사고와 문제 해결 능력이 크게 향상되었다고 느낀다. 파이프라인 단계 간의 데이터 흐름을 관리하고 각 단계별로 발생할 수 있는 다양한 상황을 처리하는 것은 매우 까다로웠다. 이를 해결하기 위해 두 가지 주요 메커니즘을 구현하였다. 포워딩 유닛과 스톱 메커니즘이었다. 특히, 제어 해저드를 예측하고 처리하는 부분은 매우 복잡한 작업이었다. 제어 해저드는 분기 명령어가 실행될 때 발생하며, 분기 명령어의 조건이 참인지 거짓인지에 따라 다음에 실행할 명령어 주소가 달라지게 된다. 이를 해결하기 위해 몇 가지 전략을 사용하였다. 종합적으로, 이번 프로젝트는 단순히 기술적 지식을 넓히는 것을 넘어서, 실제 하드웨어 시스템의 작동 원리를 체험하고, 효과적인 문제 해결 전략을 경험하는 기회였다. 이 경험은 나의 전문성을 한층 더 발전시키는 데 큰 도움이 되었으며, 향후 다양한 기술적인 도전을 하는 것에 자신감을 심어주었다. 파이프라인 CPU 아키텍처가 단일 사이클 CPU와 어떻게 다른지, 그리고 왜 더 효율적인지에 대한 이해를 깊게 할 수 있었고, 이러한 기술들은 실제 하드웨어 설계에서 매우 중요한 요소라는 것을 실감하게 되었다.

[참고 문헌]

[1]Single Cycle의 문제점, Gofo-Coding. "4. Single Cycle, Multi Cycle, Pipeline." Gofo Coding Blog. <https://gofo-coding.tistory.com/entry/4-Single-Cycle-Multi-Cycle-Pipeline>.

[2]Single Cycle 과 Multi Cycle 비교, Park, KyungPhil. "Pipeline CPU(1) Single_Cycle_CPU VS Multi_Cycle_CPU." KyungPhil Dev Blog. <https://kyungphildev.github.io/computer-architecture/pipeline-1/>.

[3]Pipelined, T.G. "Pipelined." Velog. <https://velog.io/@tg-96/%ED%8C%8C%EC%9D%B4%ED%94%84%EB%9D%BC%EC%9D%B4%EB%8B%9D>.

[4] Data Hazard, Byjus. "Data Hazards in Computer Architecture." Byjus. Byjus. "Data Hazards in Computer Architecture." Byjus. <https://byjus.com/gate/data-hazards-in-computer-architecture-notes/>.

[5] Control Hazard, Mount, David. "Handling Control Hazards." University of Maryland. <https://www.cs.umd.edu/~meesh/411/CA-online/chapter/handling-control-hazards/index.html>.

[6]Structurel Hazard, Byjus. "Structural Hazards in Computer Architecture." Byjus. <https://byjus.com/gate/structural-hazards-in-computer-architecture-notes/>.