Computer Architecture

2023-06-18

엄현호

32192587@dankook.ac.kr

Contents

5. Build Environment for Testing Simulator

6.Lesson

1.Introduction

파이프라인은 명령 처리 주기를 별개의 처리 단계로 나누는 기술 입니다. 이 기술은 각단계에서 다른명령을 처리 함으로써 프로그램 순서대로 연속적으로 명령어가 처리되어 명령어처리 속도가 빠르다는 특징이 있습니다. 그래서 이 보고서에서는 파이프라인 마이크로아키텍처 운영,구현 및 프로젝트를 설명합니다. 이보고서에 제시된 작업은 mips를 사용하는 마이크로 아키텍처를 구현하고 싱글사이클에서 파이프 라인 개념을 적용시켜 보여줄 것입니다. 이 프로젝트의 첫번쨰 단계는 파이프 라인 에대한 요구사항을 background로 보여줍니다 두번쨰로는 파이프라인의 데이터 종속성으로 인한 hazard오류들을 해결하는 코드를 구현하고 stall을 처리해 보여줍니다.

셋째로 모든 bin파일을 실행하는 결과하면을 보여주고 느낀점,배운점을 서술합니다.

2.Background

2-1. Single-Cycle Microarchitecture

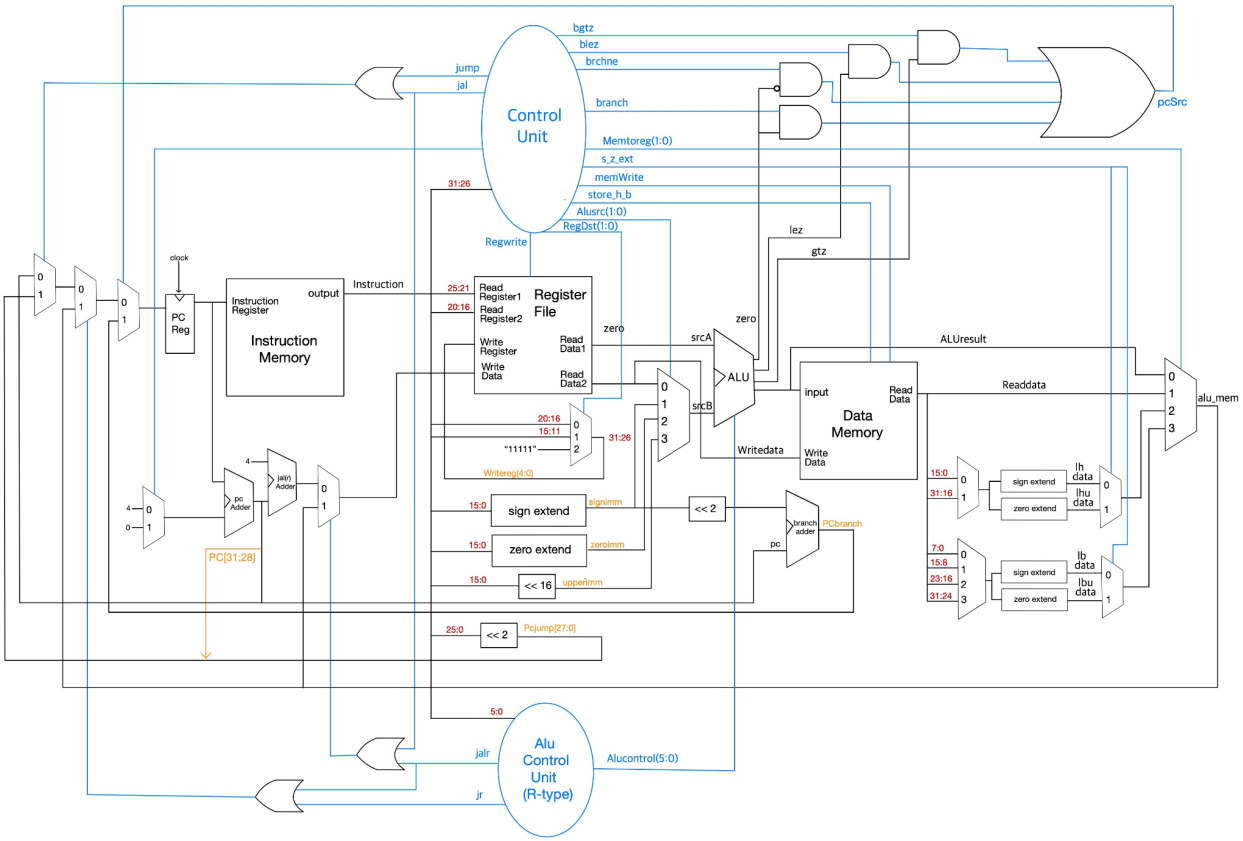


그림**1**

단일 주기 기계에서 각 명령어는 단일 클럭 주기를 가지며 명령어 실행이 끝나면 모든 상태가 업데이트됩니다. 주기 실행 시간은 많은 명령어가 실행에 그렇게 긴 실행 시간이 필요하지 않음에도 불구하고 가장 느린 명령어 실행 시간에 의해 결정됩니다. 따라서 **MIPS ISA**에서는 **Load Word(LW)** 명령어가 가장 오랜 시간이 소요되기 때문에 **Load Word(LW)** 명령어 실행의 흐름에 따라 한 사이클이 수행됩니다. 단일 주기마이크로아키텍처에는 조합 읽기, 동기식 쓰기, 동기식 메모리의 세 가지 주요 개념이 있습니다. 조합 판독에서 읽기 데이터 포트의 출력은 레지스터 파일 내용과 해당 읽기 선택 포트의 조합 기능입니다. 동기식 쓰기에서는 쓰기 활성화가 할당될 때 선택한 레지스터가 양의 에지 클럭 전환으로 업데이트되며, 이는 클럭 에지 사이의 읽기 출력에 영향을 미치지 않습니다. 동기식 메모리에서는 데이터 준비 시간을 알려주는 메모리와 비교합니다. 예를 들어, 준비 비트는 읽기 또는 쓰기가 완료되었음을 나타냅니다. 그림 **1**는 **MIPS ISA**의 형태로 명령을 실행할 수 있는 단일 주기 마이크로아키텍처의 총 데이터 경로를 보여줍니다. 다음 마이크로아키텍처는 부동 소수점을 다루는 명령을 제외한 대부분의 **MIPS** 명령을 실행할 수 있습니다.

그림 **1**의 단일 주기 **MIPS** 시뮬레이터에 따르면 이전 시뮬레이터는 아래의 명령 유형을 실행할 수 있습니다**:**

1. R-Type 및 **I-Type** 산술 명령**: add, and, or, slt, sub, lui, sll, srl**

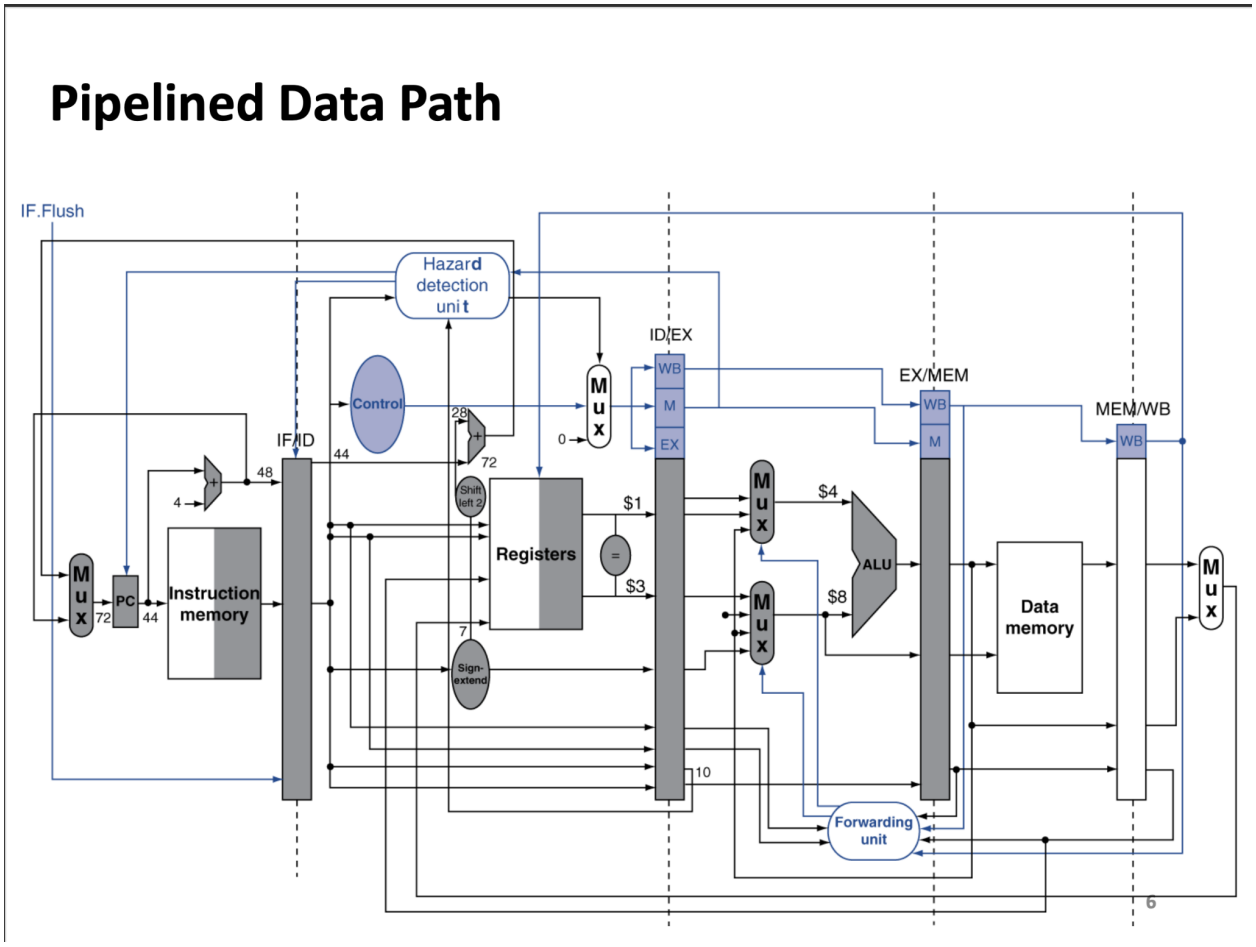
2. Load Word 명령어**: lw**

3. Store Word 사용설명서 **: sw**

4. Branch Taken 및 **Branch Not Taken** 지침**: beq, bne**

5. 점프 지침**: j, jal, jr, jalr**

2-2. Pipelined Microarchitecture



-그림2-

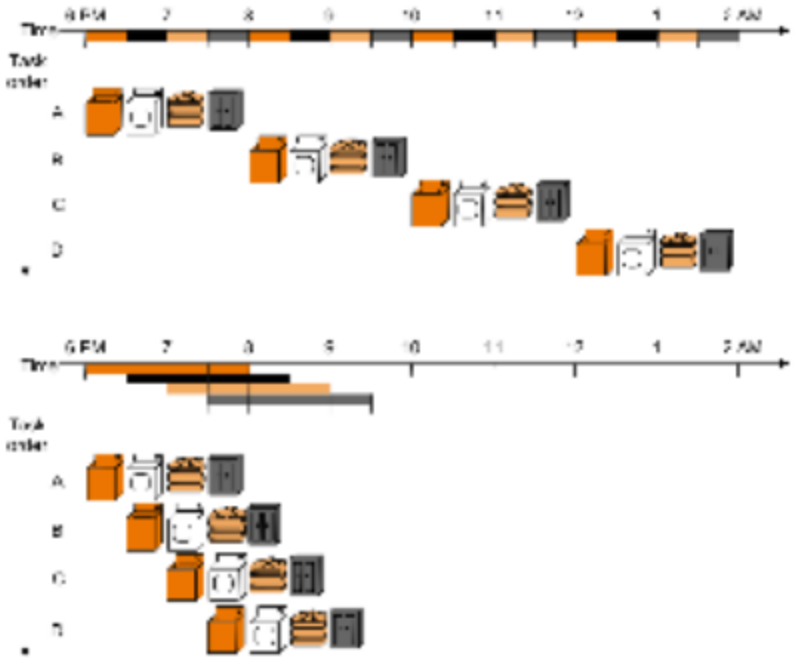
단일 주기 마이크로아키텍처는 **LW(Load Word)** 명령어의 사이클 처리 시간을 가지고 있는 반면, 다른 명령어들은 실행 시간이 더 짧습니다. 이는 건축적 비효율성을 초래합니다. 연구자들은 다음과 같은 마이크로아키텍처의 성능을 높이기 위해 더 적은 사이클을 사용하여 명령을 수행하고 클럭 주파수를 증가시키려고 노력하였으며, 이에 따라 다중 사이클의 개념이 대두되었습니다.

다중 주기 데이터 경로는 명령을 별도의 단계로 나눕니다. 각 단계는 단일 클럭 사이클을 사용하며, 각 기능 장치는 서로 다른 클럭 사이클에서 사용되는 경우 지침에 따라 두 번 이상 사용할 수 있습니다. 이렇게 구현하면 필요한 하드웨어의 양을 줄일 수 있습니다. 즉, 평균 명령 시간을 단축합니다.

다중 주기 마이크로아키텍처는 클럭 속도가 더 빠르고, 간단한 명령을 위한 실행 시간이 더 빠르며, 고가의 하드웨어에서 재사용이 가능하지만, 시퀀싱 오버헤드의 지불은 빈번하게 발생할 수 있습니다**(**다중 주기 **MIPS** 프로세서 **- ETHZ).** 그림 **2**은 다중 사이클 마이크로아키텍처의 데이터 경로를 보여줍니다. 다음 마이크로아키텍처의 제어 장치는 상태 기계로 작동합니다. 그림 **2**는 멀티 사이클 마이크로아키텍처의 현재 상태로 인한 메인 컨트롤러 작동을 보여줍니다. 다중 주기 마이크로아키텍처에서 명령이 실행되는 동안 컨트롤러 상태가 변경됩니다. 메인 컨트롤러는 프로세서의 모든 상태에 대해 그림 **2**에 표시된 상태 그래프로 인해 작동을 결정합니다.

다중 주기 마이크로아키텍처를 사용하면 작업에 다른 주기가 소요되고 명령당 평균 클럭**(CPI)** 시간이 단축됩니다. 따라서 명령 처리 성능을 높일 수 있습니다. 그러나 다중 주기 마이크로아키텍처의 보다 복잡한 제어와 다음 아키텍처의 오버헤드는 해결해야 할 남은 문제입니다.

2-3. *Performance Analysis*



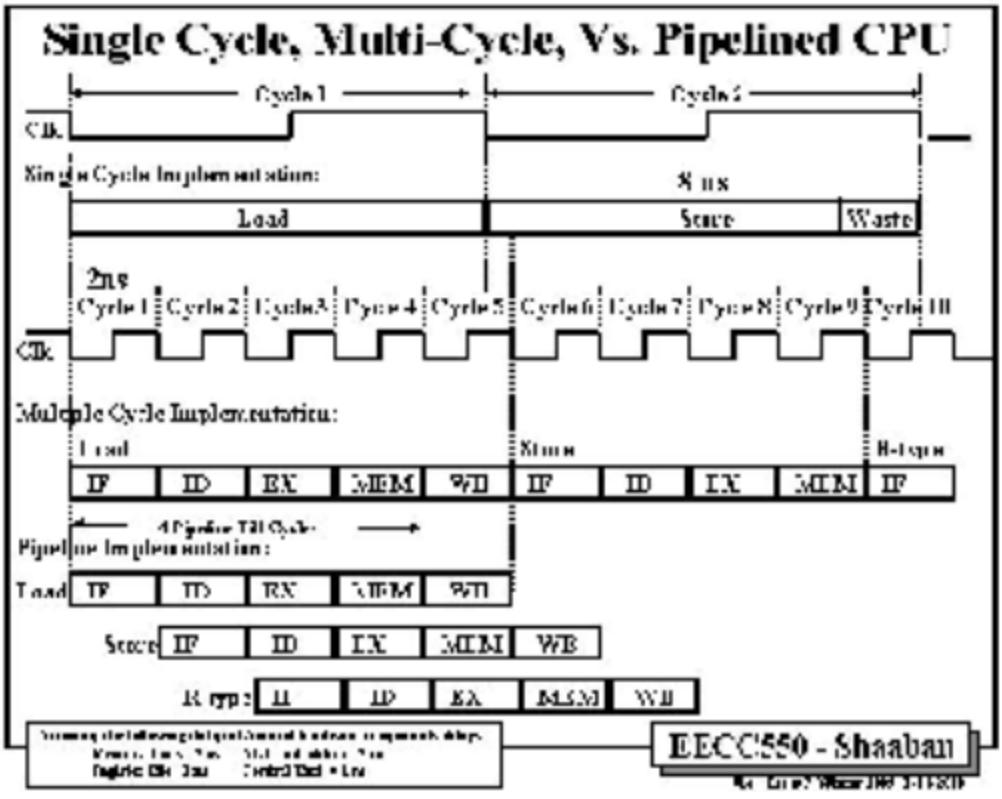


그림3

그림**3** 에는 단일 주기 **CPU,** 다중 주기 **CPU** 및 파이프라인 **CPU**의 클럭 주기가 나와 있습니다. 아래 그림에 따르면 멀티사이클 방식을 적용할 때 **CPI**는 증가하지만 클럭 사이클 시간은 감소하는 것을 확인할 수 있습니다. 따라서 총 성능이 향상됩니다. 파이프라인의 경우 모든 사이클에서 명령을 실행하도록 설정하여 **CPI**를 줄일 수 있습니다. 그러면 우리는 프로세서의 성능을 향상시킬 수 있습니다. 결과적으로 파이프라인 마이크로아키텍처가 가장 효율적인 클럭 시간 실행을 보여주는 것을 확인할 수 있습니다.

2-4. Hazard

컴퓨터 구조에서의 **Hazard**는 파이프라인 처리, 명령어 실행, 데이터 의존성 등과 관련된 잠재적인 위험 상황을 나타내며, 이를 효과적으로 관리하는 것은 매우 중요합니다. 따라서 컴퓨터 구조에서의 **Hazard**를 세 가지 측면에서 분석하고 관리하는 방법을 제시합니다. 구체적으로는 **Data Hazard, Control Hazard, Structural Hazard**에 초점을 맞추어 다룹니다.

a) data hazard

Data Hazard는 명령어들 간에 데이터 의존성으로 인해 발생하는 **Hazard**를 의미합니다. 데이터 의존성은 한 명령어가 다른 명령어의 실행 결과에 의존하는 경우를 가리킵니다. 이는 파이프라인 처리에서 데이터의 불일치와 충돌을 초래할 수 있으며, 잘못된 결과를 얻을 수 있습니다.

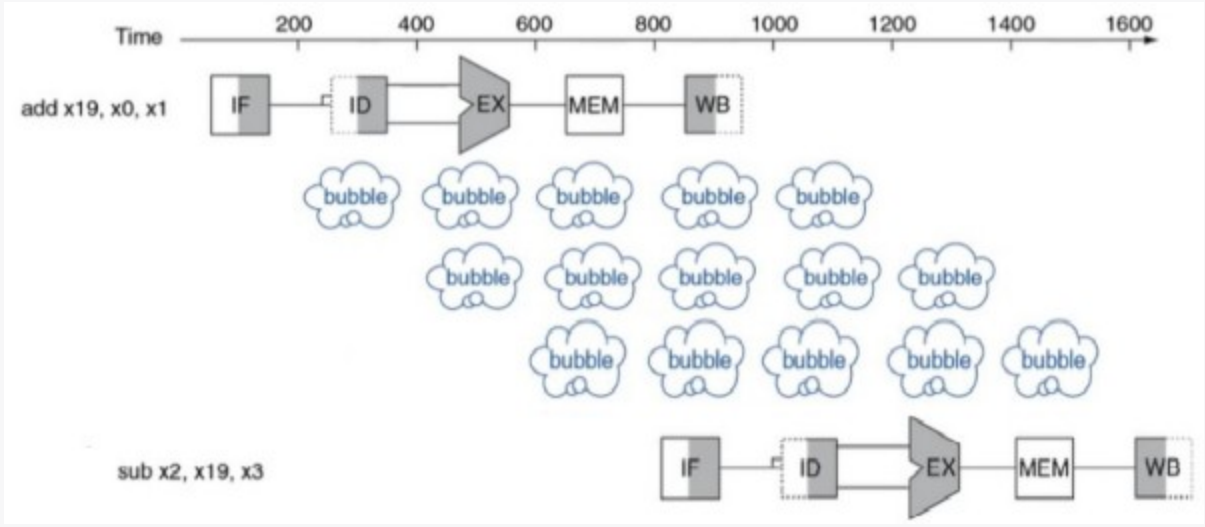


그림**4**

예시로 위 그림**4**를 보면 **x19**는 두 명령어 모두에게 영향을 끼칩니다. 전형적인 **5-stage**의 순서대로 각각의 명령어들이 실행되고 있는데 현재 중간에 **bubble** 에 해당하는 부분이 있습니다. **x19**는 **add**명령어의 **register**인대 **sub**에선 **x19**가 **operands**입니다. 만약 원래대로 정상적인 동작을 한 다면 **sub** 명령어의 **fetch(if)**가 **add**명령어의 **ID**에 이루어져야 하는데 그렇게 되면 **x19**는 계산한 값이 나오는게 아니라 이전에 저장되어있는 다른 값이 **sub**명령어의 오퍼랜드 값으로 들어갑니다. 이는 실제 저장동작이 **WB**에서 이루어지기 때문입니다. 때문에 **sub**명령어는 실제적으로 **add**명령어가 **data**를 **wrire**하기 전까지 **wait**한 후에 디코딩을 해야되고 **register**을 읽어야합니다. 이를 **data hazard**라고 합니다.

b) struct hzard

Structural Hazard는 하드웨어 자원의 한정성으로 인해 발생하는 **Hazard**를 의미합니다. 여러 명령어들이 동시에 동일한 자원에 접근하려고 할 때 충돌이 발생하며, 파이프라인 처리에 방해를 줄 수 있습니다.

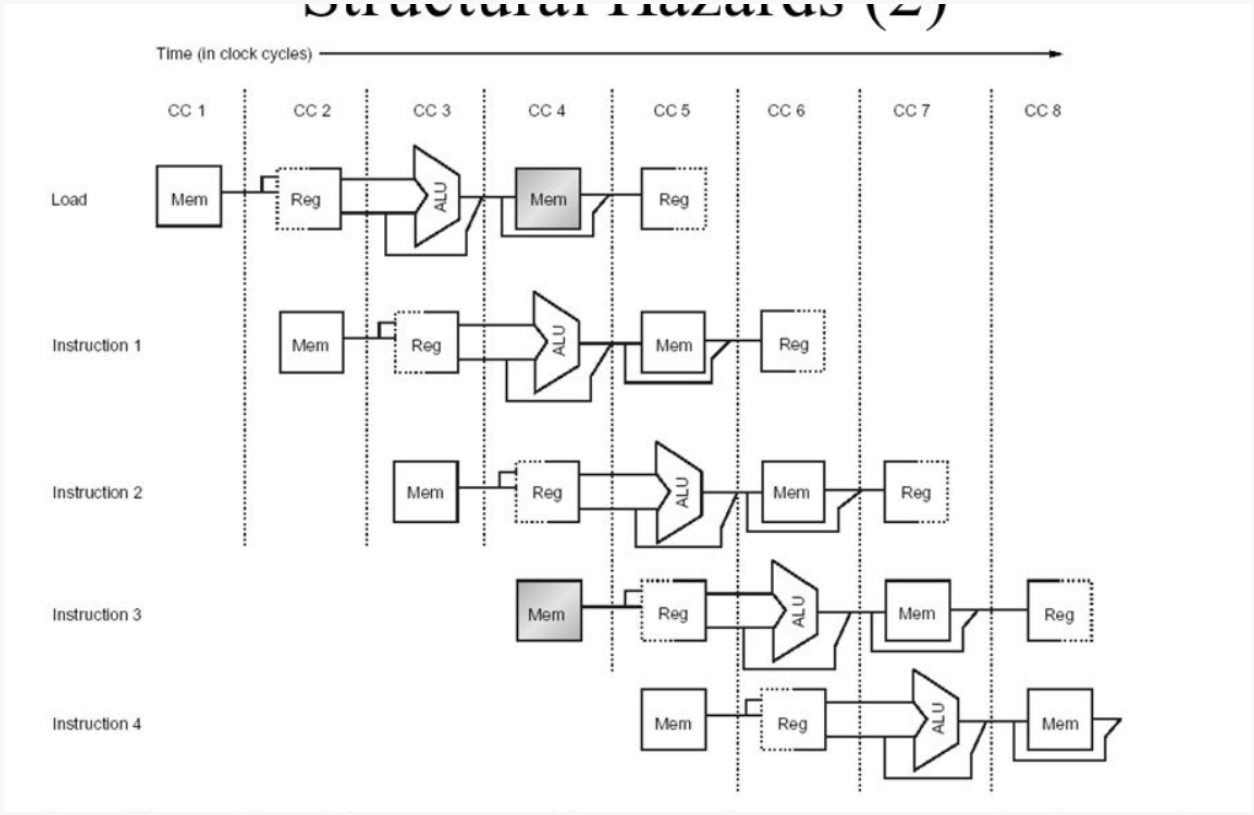


그림 **5**

예시로 그림**5**을 보면 **nstruction1** 에서는 **data fetch**를 위해, **instruction4**에서는 **instruction fetch**를 위해 **CC5**일 때, 한번에 **mem**에 접근하려고 하고 있다. 이 때, 구조적 해저드가 발생한다.

c) control hazard

Control Hazard는 분기 명령어와 같이 제어 흐름에 관련된 명령어들 간의 **Hazard**를 의미합니다**.** 분기 예측 실패나 지연된 분기로 인해 잘못된 명령어가 실행될 수 있으며**,** 이는 프로그램의 정확성과 성능을 저하시킬 수 있습니다**.** 예시로 beq 혹은 J 명령어 등 분기 명령어를 통해 명령어를 건너뛰어버리면 PC 값이 변해버리고, 중간에 있는 명령어를 실행하지 않는 현상이 있다.

2-5. Latch

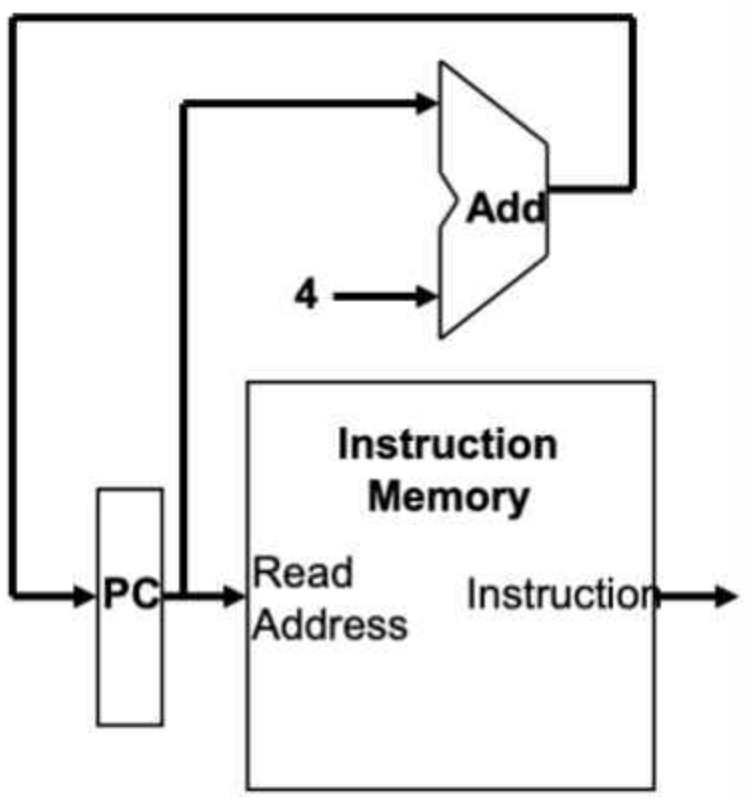


그림**6**

a) IF (Instruction Fetch) : 이□ 단계에서는□ 명령어□ 메모리에서□ 명령어를□ 가져옵니다**.** 프로그램□ 카운터□ **(PC)**를□ 증가시키고**,** 해당□ 주소에서□ 명령어를□
인출하여□명령어□레지스터**(IR)**에□저장합니다**. IF** 단계는□다음에□수행될□명령어의□주소를□결정하는□역할을□합니다**.**명령어□메모리**(Instruction Memory)**에서□**PC**값에□해당하는□주소에□저장되어있는□**4**바이트□크기의□**Instruction**을□

가져오고, **PC**의□값을□**4** 증가시켜주는□과정이□**IF**단계입니다**.** 물론,이후에□**J** 타입의□명령어나□**I** 타입의□**Branch** 명령에의해□**PC+4** 이외의□ 다른□ 연산도□ 추가될□ 수□ 있으나, 가장□ 기본적인□ **IF**의□ 과정입니다**.**

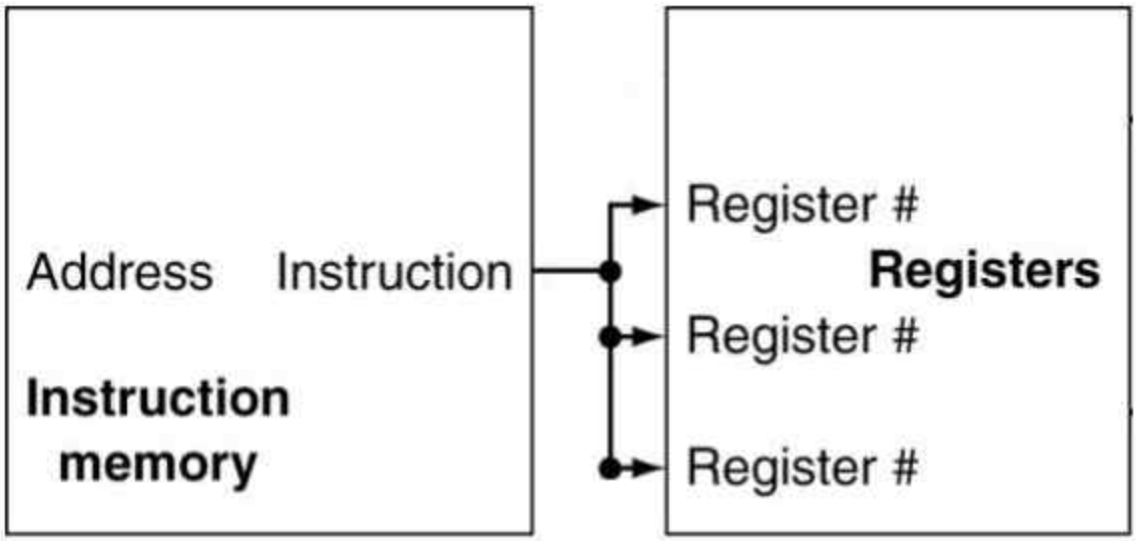


그림**7**

b) ID/RF (Instruction Decode/Register Fetch) : ID/RF 단계에서는□ 명령어를□ 해독하고□ 필요한□레지스터□ 값을□ 읽어옵니다**.** 명령어에서□ 필요한□ 제어□ 신호와□ 레지스터□ 번호□ 등을□ 추출하여□ 제어□ 유닛과□ 레지스터□ 파일에□ 전달합니다**. ID/RF** 단계에서는□ **Instruction Memory**에서□ 읽어드린□ 명령어를□ **RS, RT, RD, SHAMT, IMMED, FUNCT** 등으로□ 분석하는□ 작업과**, RS, RT, RD**에□ 해당하는□ 레지스터□ 값을□ 읽습니다**.**

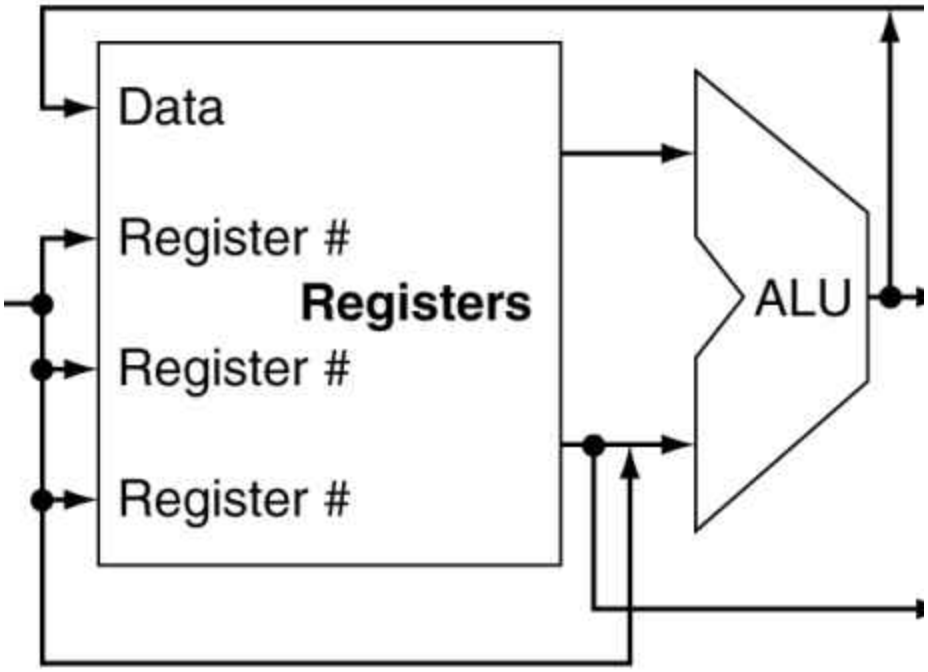


그림**8**

c) EX/AG (Execute/Address Generate) : EX/AG 단계에서는□ 산술□ 논리□ 연산과□주소□계산□등의□연산을□수행합니다**. ALU (**산술□논리□장치**)**에서□연산을□리하고**,** 필요한□ 경우□ 주소□ 계산을□ 수행하여□ 데이터□ 메모리□ 주소를□ 생성합니다**.EX/AG** 단계에서는□ 레지스터에서□ 읽어들인□ 값□ 및□ **IMMED**값□ 등과□ 산술□ 논리□ 연산을□진행합니다**.** 이때□ 결과값은□ 단순□ 데이터□ 값일□ 수도□ 있고□ 특정□ 주소**(Address)**값이□ 될□ 수도□ 있습니다

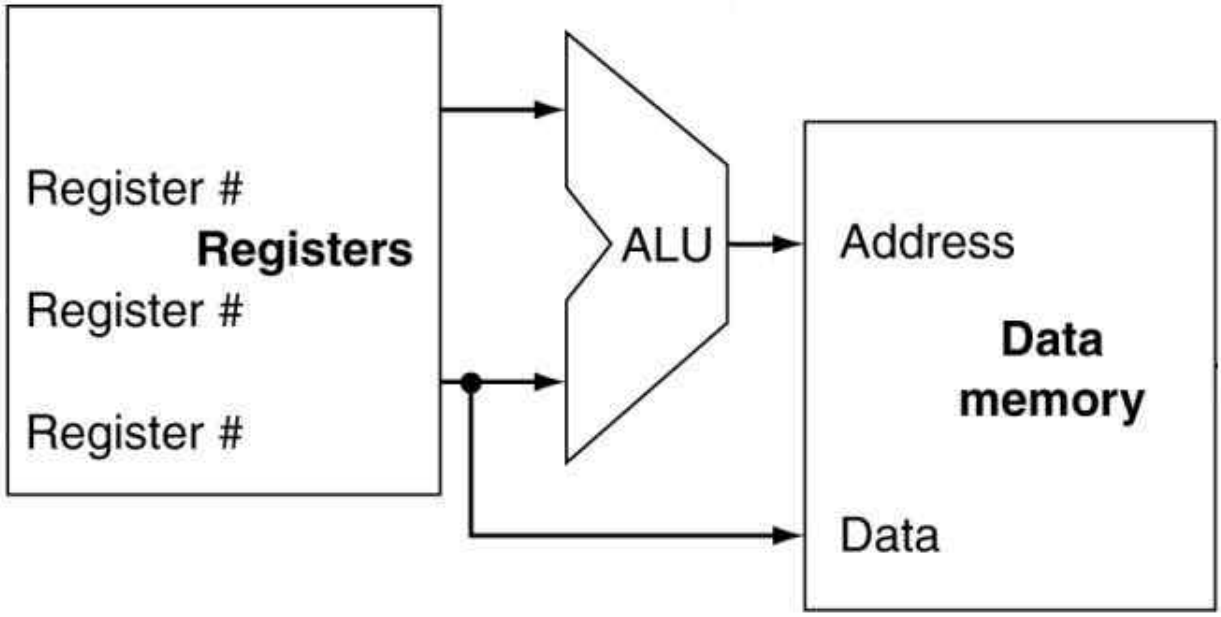


그림**9**

d) MEM (Memory Access) : MEM 단계에서는□ 데이터□ 메모리에□접근하여□ 데이터를□읽거나□쓰기□작업을□수행합니다**.** 주소□계산□결과로부터□데이터□메모리에□ 접근하고**,** 필요한□ 경우□ 데이터를□ 읽어올□ 수□ 있습니다**.**
MEM 단계에서는□ **ALU**에서□계산된□값을□**Address**로□하는□메모리에□값을□쓰거나□읽어올수□ 있습니다**.**이는□메모리□접근과□관련된□명령어인□**SW(StoreWord)**혹은□**LW(Load Word)**에□ 따라□결정됩니다**.**

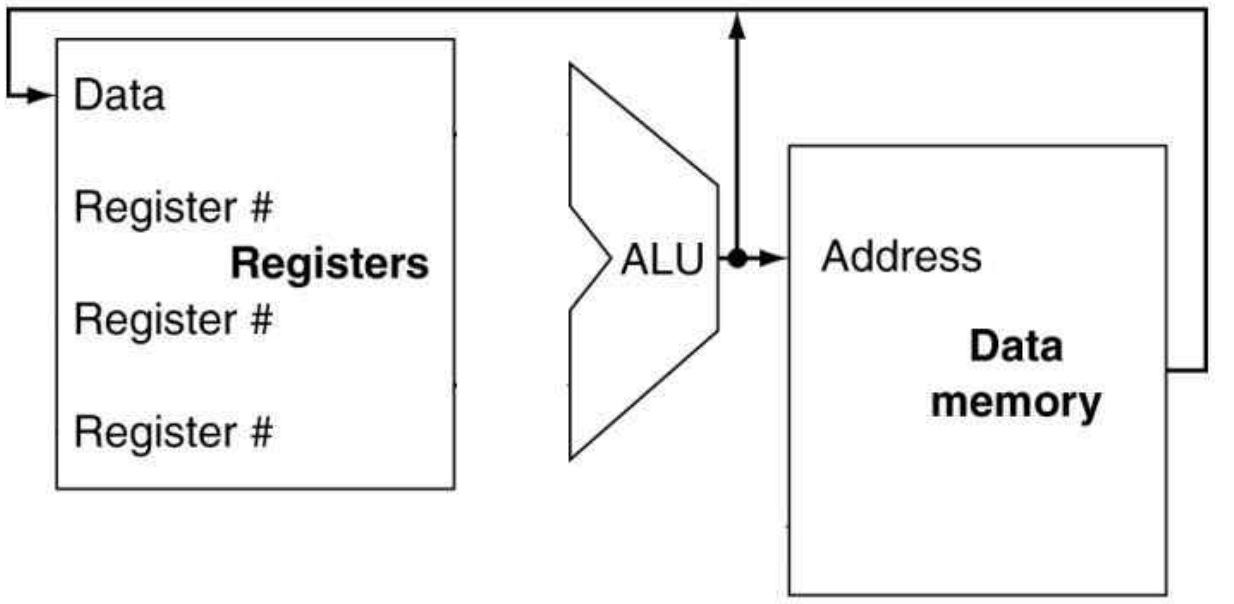


그림**10**

e) WB (Write Back) : WB 단계에서는□ 연산의□ 결과를□ 레지스터□ 파일에□ 기록합니다**. EX/AG** 또는□**MEM** 단계에서□계산된□결과나□데이터를□레지스터에□저장합니다**.ALU**에서□ 계산된□ 값□ 혹은□ **Data Memory**에서□ 읽어들인□ 값을□ **ID/RF**단계에서의□ **Rt**혹은□**Rd**에□ 저장합니다**.**

2-6. Stalling

스톨링**(Stalling)**은 컴퓨터 구조에서 **Hazard**를 관리하기 위해 사용되는 기법 중하나로 주로 데이터 의존성**(Data Dependency)**으로 인한 **Hazard**를 처리하는 데에 활용된다.데이터 의존성은 한 명령어가 다른 명령어의 실행 결과에 의존하는 경우를 의미합니다. 예를 들어, 이전 명령어가 어떤 데이터를 계산하고, 그 값을 다음 명령어가 사용해야 하는 경우 데이터 의존성이 발생합니다. 파이프라인 구조에서는 명령어들이 동시에 실행되기 때문에, 이전 명령어의 결과가 아직 다음 명령어에 이용되기 전에 다음 명령어가 실행되면 잘못된 결과를 얻을 수 있습니다. 이러한 상황에서 스톨링은 파이프라인을 지연시키는 방법으로 데이터 의존성을 해결합니다. 즉, 데이터 의존성이 발생한 경우, 다음 명령어가 실행되지 않고 파이프라인의 진행을 멈추는 것입니다. 이전 명령어의 실행 결과가 사용 가능해질 때까지 기다렸다가, 그 후에야 다음 명령어를 실행합니다. 이렇게 함으로써 데이터 의존성에 의한 **Hazard**를 방지하고, 올바른 결과를 얻을 수 있습니다. 하지만 스톨링은 파이프라인의 효율성을 저하시키고 최소한의 필요한 경우에만 사용되어야 퍼포먼스를 높일 수 있습니다.

2-7. Forwarding

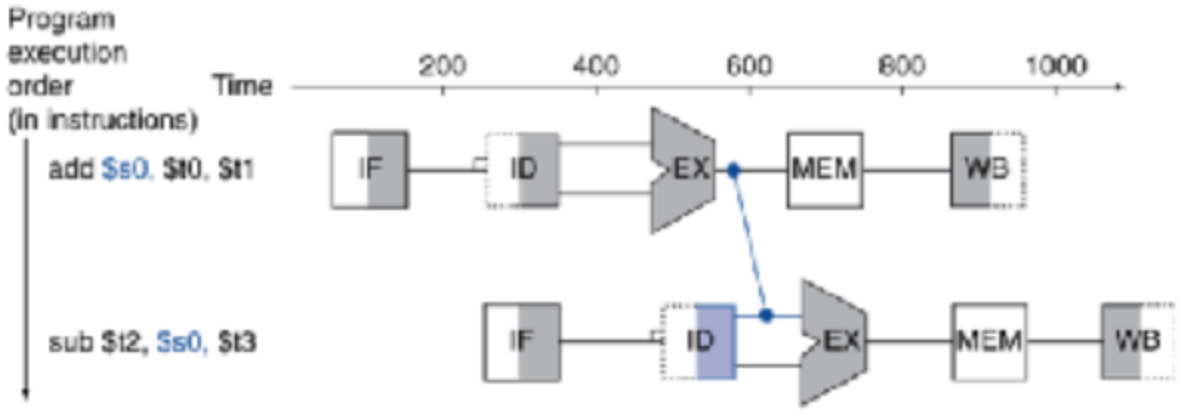


그림**11**

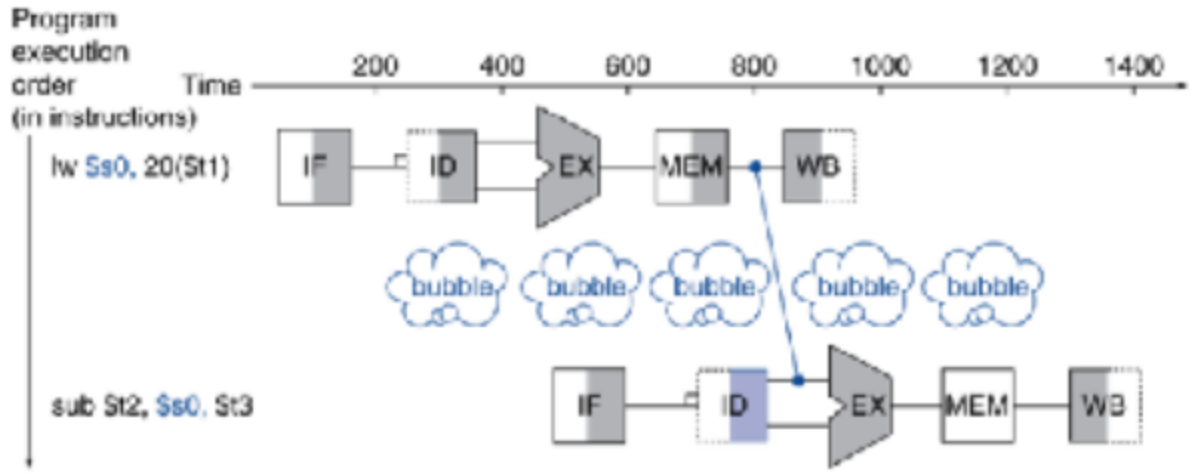


그림**12**

2-6에서 봤듯이 **stall**은 프로세서의 성능이 저하됩니다. 따라서 설치 횟수를 줄이면서 데이터 의존성을 해결할 수 있다면 프로세서의 성능이 향상 됩니다. 이런 **stall**없이 데이터 종속성을 해결하는 방법 중하나가 **Forwarding**입니다.

이 방법에서는 안티 및 출력 종속성을 지연시킬 필요가 없습니다. 그러나 논리를 따르는 것은 스코어보드보다 더 복잡하며, 파이프라인을 더 깊고 넓힐수록 논리는 더 복잡해집니다.

결합된 종속성 검사 로직 중 하나는 데이터 전달/바이패스입니다. 데이터 종속성에서 시작된 문제는 종속 명령어가 생산자 명령어가 레지스터 파일에 값을 기록할 때까지 디코딩 단계에서 기다려야 한다는 것입니다. 이 방법의 목적은 파이프라인을 불필요하게 지연시키지 않는 것입니다. 데이터 전달/바이패스 방법은 종속 명령어에 필요한 데이터 값을 관찰하고, 필요한 값은 값을 사용할 수 있는 즉시 파이프라인의 후반 단계에서 직접 공급됩니다. 이 방법을 적용하면 값이 제공될 수 있는 지점까지 종속 명령어가 파이프라인으로 이동할 수 있으므로 지연 횟수가 줄어듭니다.

데이터 전달/바이패스 방법에서는 세 가지 유형의 종속 사례를 고려해야 합니다. 그림 **11**은 데이터 종속성의 첫 번째 사례를 보여줍니다. 그림에서 확인할 수 있듯이, 두 번째 명령어는 첫 번째 명령어의 실행 단계에서 계산된 디코드 단계의 값을 사용하는 것입니다. 이 경우 단계 사이의 거리는 **1**입니다. 그림 **12**는 데이터 의존성의 두 번째 사례를 보여줍니다. 그림은 디코딩 단계의 값을 사용하여 두 번째 명령의 상황을 보여줍니다. 이 값은 첫 번째 명령의 데이터 메모리에서 실제 값입니다. 이 경우 단계 사이의 거리는 **2**입니다. 마지막 경우는 스테이지 사이의 거리가 **3**보다 큰 상황입니다. 이 경우, 우리는 라이트백 단계 이후에 디코드 단계를 명령함으로써 이 경우를 무시할 수 있습니다. 그림 **13**은 위에 제시된 사례를 포함한 조건을 보여줍니다. 그림 **13**과 같은 상태에서 감지하여 작동하는 하드웨어 구성요소를 구현함으로써 데이터 전달/바이패스 방식을 적용하고 프로세서를 최적화할 수 있습니다.

3. Piplined Microarchitecture Design

3-1. Datapath Implement

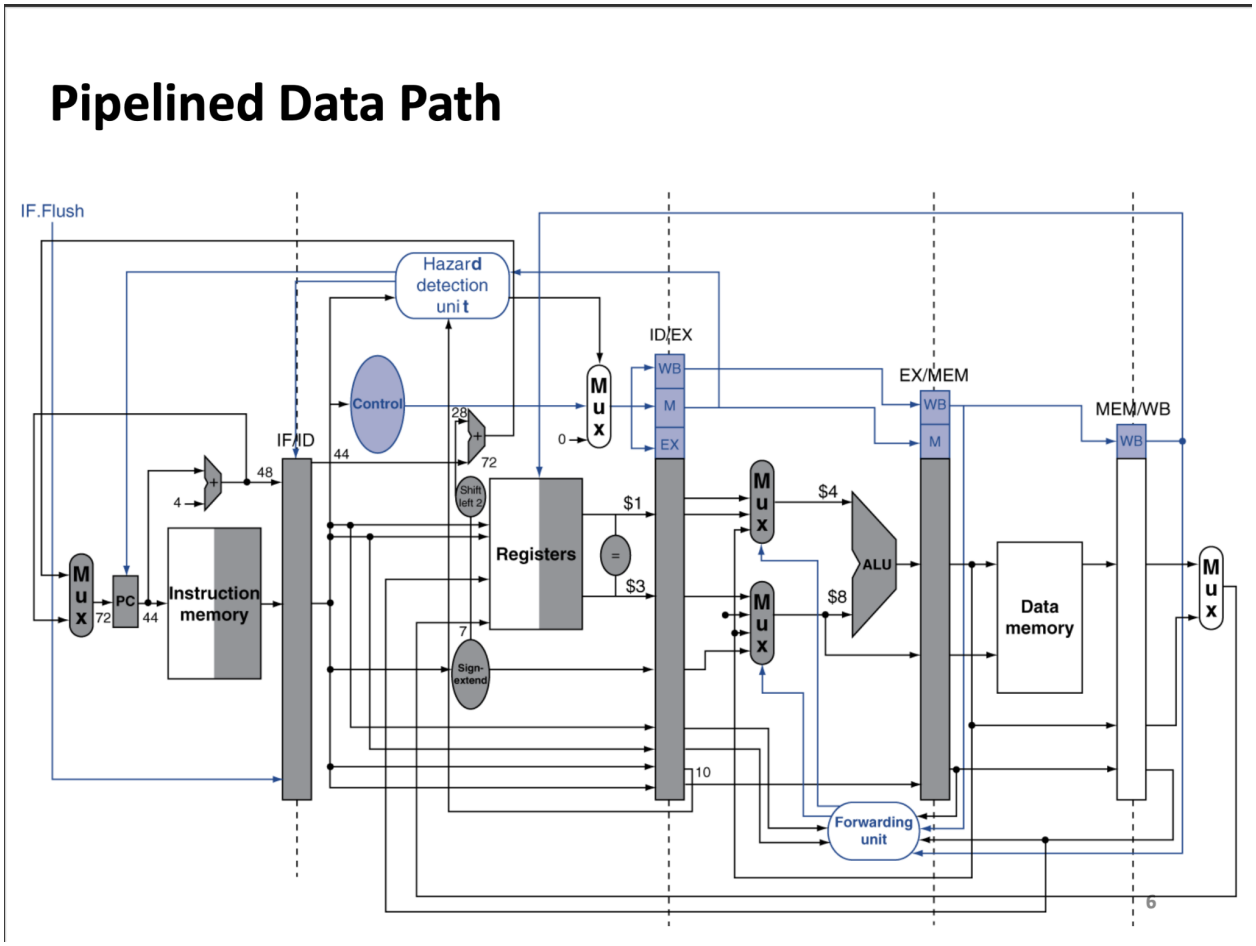


그림**13**

위사진은 제가 **c**코드로 데이타패스 를 설계할때 쓰인 **Piplined Microarchitecture DataPath**입니다**.**

3-2. Design of this Processor

시뮬레이터를 구현하기 위한 아이디어와 방법에 대해 설명할 것입니다. 파이프라인 마이크로아키텍처에서는 여러 명령어가 동일한 주기로 실행됩니다. 클럭 주기마다 명령어 메모리에서 새 명령어를 가져오고, 각 명령어 실행 단계는 새롭고 다른 명령어를 얻습니다. 파이프라인 MIPS 시뮬레이터의 구현은 이러한 개념을 포함하며 파이프라인 data path, Latch, Forwarding,hazard detection units 따라 구현됩니다

3-2-1 Latch

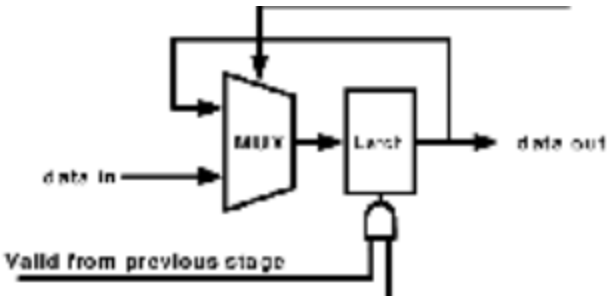


그림14

파이프라인이 항상 채워진 것은 아닙니다. 파이프라인의 흐름이 중단되면 중단 전의 모든 단계가 중지되며, 이를 스톨이라고 합니다. 나머지 파이프라인은 계속 작동할 수 있습니다. 이 작업을 제어하기 위해 래치에 유효한 비트가 포함되어 있습니다. 이전 단계의 유효한 비트는 클럭 신호를 게이트하는 데 사용됩니다. 다음 단계의 홀드 신호는 멀티플렉서를 사용하여 정지 상태(파이프라인 래치)로부터 데이터를 재순환시킵니다. 이전 단계의 유효한 비트는 게이트 ...에 사용됩니다. 래치의 구조는 그림 14에 나와 있습니다.

아키텍처에 래치를 적용하여 파이프라인 마이크로아키텍처 프로세서를 구현할 수 있습니다. 한 가지 고려 사항은 래치의 접근 시간이 시계 주기 시간보다 짧아야 한다는 것입니다. 클럭 주기 시간보다 길어지면 아키텍처의 전체 주기가 점점 더 많은 단계로 분할될 때 마이크로 아키텍처의 비효율성이 발생합니다. 오늘날 래치에 대한 액세스 시간은 프로세서의 클럭 사이클 시간과 거의 같습니다.

3-2-2 Forward Unit

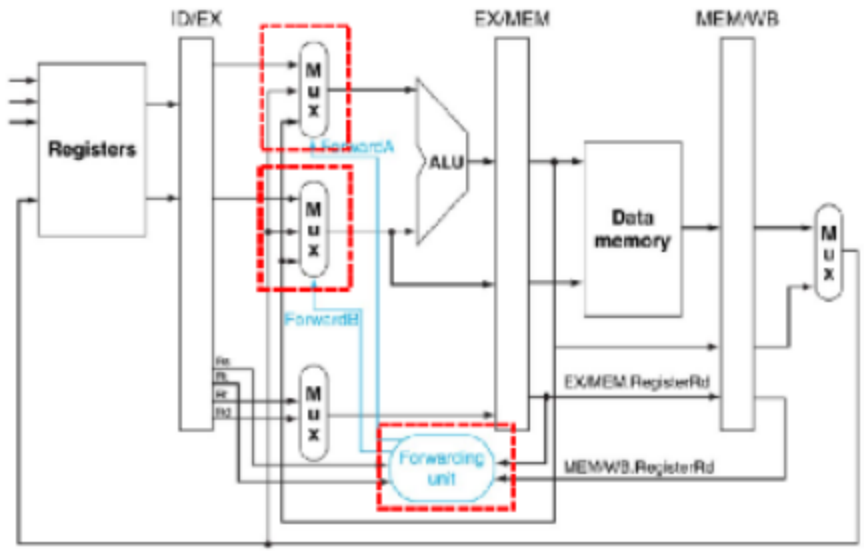


그림15

MUX Control Signal	Single Value	MUX Selection
ForwardA	00	First ALU operand = ID/EX.A
	10	First ALU operand = EX/MEM.AR
	01	First ALU operand MEM/WB.A or MEM/WB.LR
ForwardB	00	Second ALU operand = ID/EX.A
	10	Second ALU operand = EX/MEM.AR
	01	Second ALU operand = MEM/WB.A or MEM/WB.LR

그림16

데이터 포워딩은 스톨링 및 점수 탑승 방식에 비해 스톨 수가 감소하여 데이터 의존성을 해결할 수 있습니다. 그림 20은 파이프라인 마이크로아키텍처에서 전진 장치의 구현을 보여줍니다. 아키텍처에 데이터 전달 방법을 구현하려면 데이터 종속성이 발생했는지 여부를 관찰하고 결정해야 합니다. 데이터 종속성 감지는 다음 조건을 따릅니다:

레지스터 번호가 0이면 전달되지 않아야 합니다.

이전 명령은 레지스터 파일에 값을 기록하는 명령이어야 합니다.

데이터를 레지스터 번호와 비교합니다:

EX/MEM.RD == ID/EX.RS

EX/MEM.RD == ID/EX.RT

MEM/WB.RD == ID/EX.RS

MEM/WB.RD == ID/EX.RT

이러한 조건은 그림 16에 표로 제시되어 있습니다. 이러한 조건에서 작동하도록 포워드 유닛을 구현함으로써 데이터 포워딩 방법을 적용하고 파이프라인 마이크로아키텍처의 데이터 의존성을 해결할 수 있습니다.

3-2-3 Hazard Detection Unit

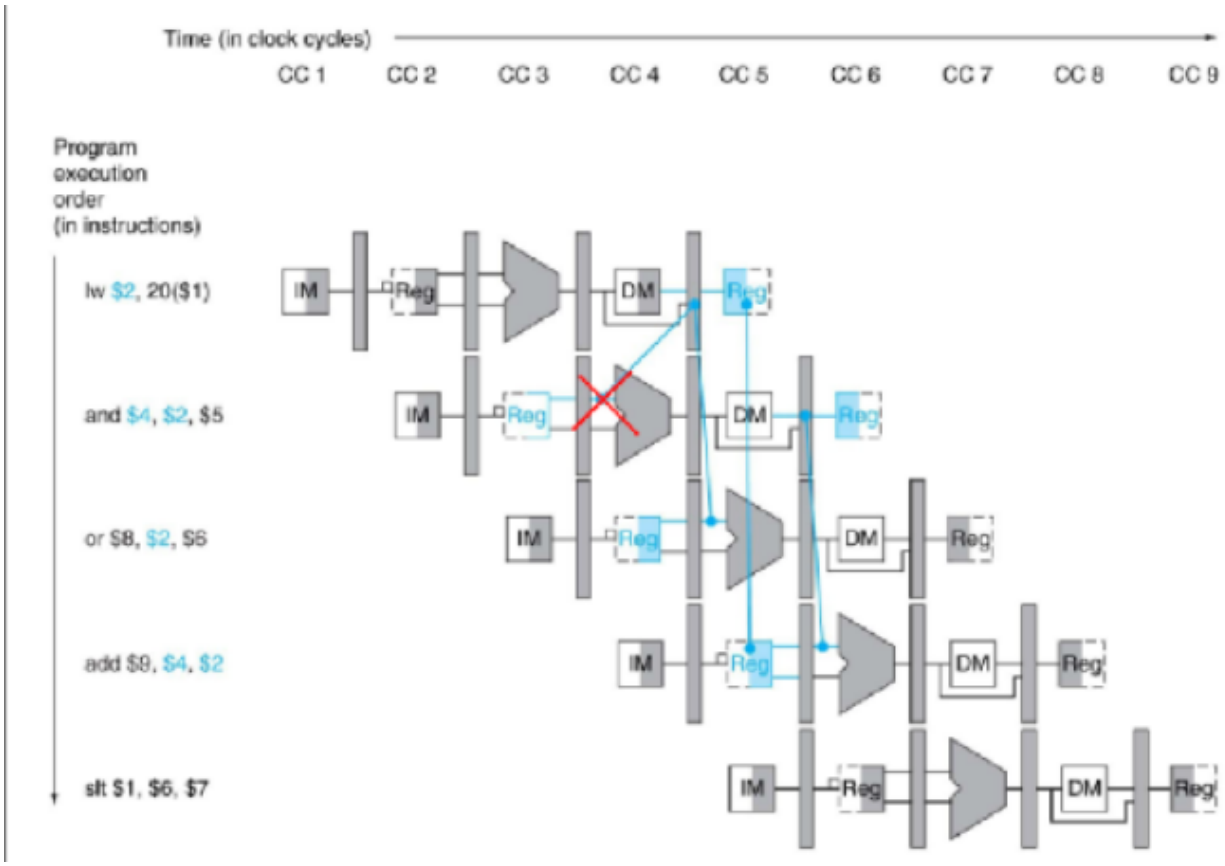

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
IM
Reg
DM

그림17

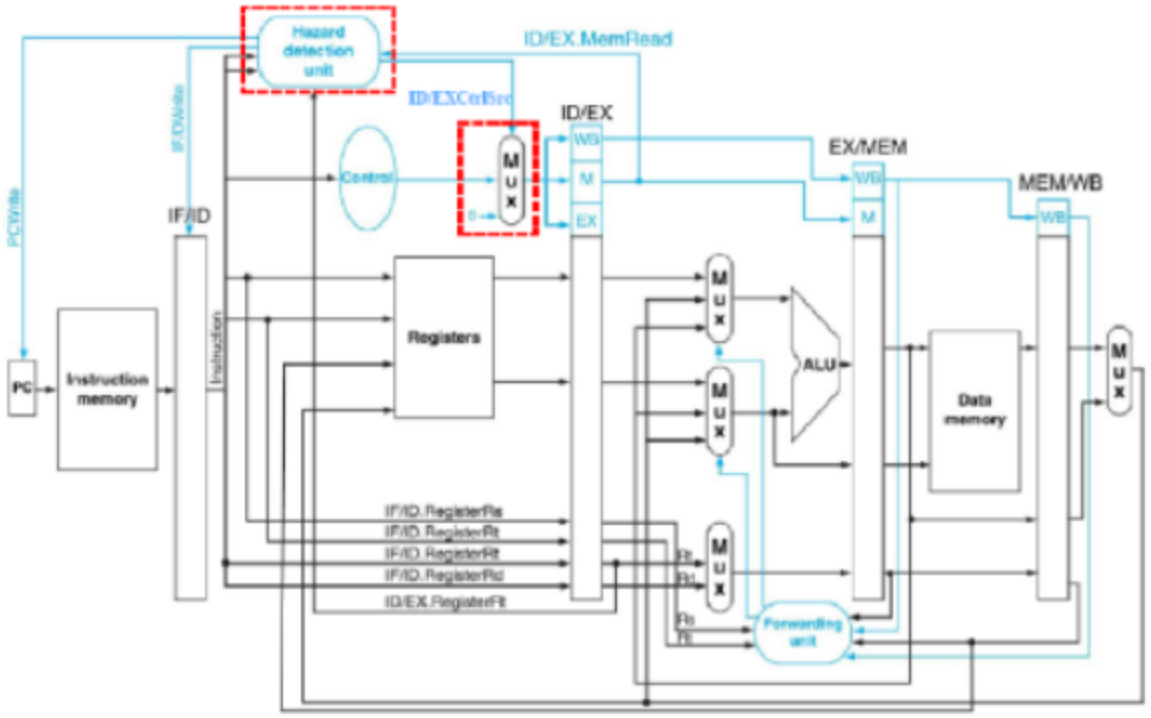

Hazard detection unit
ID/EX.MemRead
ID/EX.CtrlSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
M u x
IF/ID
PCWrite
IF/DWrite
PC
Instruction memory
Instruction
Registers
ALU
Data memory
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRd
ID/EX.RegisterRt
Forwarding unit

그림18

데이터 포워딩을 적용하여 대부분의 데이터 종속성을 해결하더라도 여전히 해결되지 않은 데이터 종속성이 있습니다. 해결되지 않은 데이터 종속성은 로드 사용 사례입니다. 그림 18에서 로드 명령 직후에 나타나는 명령은 로드 명령의 결과를 요구합니다. 다음 데이터 종속성에 대한 솔루션은 파이프라인을 중지하는 것이며, 이를 파이프라인 버블 생성이라고도 합니다. 파이프라인 스톨은 다음 방법으로 이루어집니다:

1. 명령 디코드(ID) 단계에서 데이터 종속 명령을 1사이클 동안 중지합니다.
2. IF/ID 레지스터와 PC의 업데이트를 차단합니다.
3. ID/EX 레지스터에 NOP 지침을 삽입합니다.

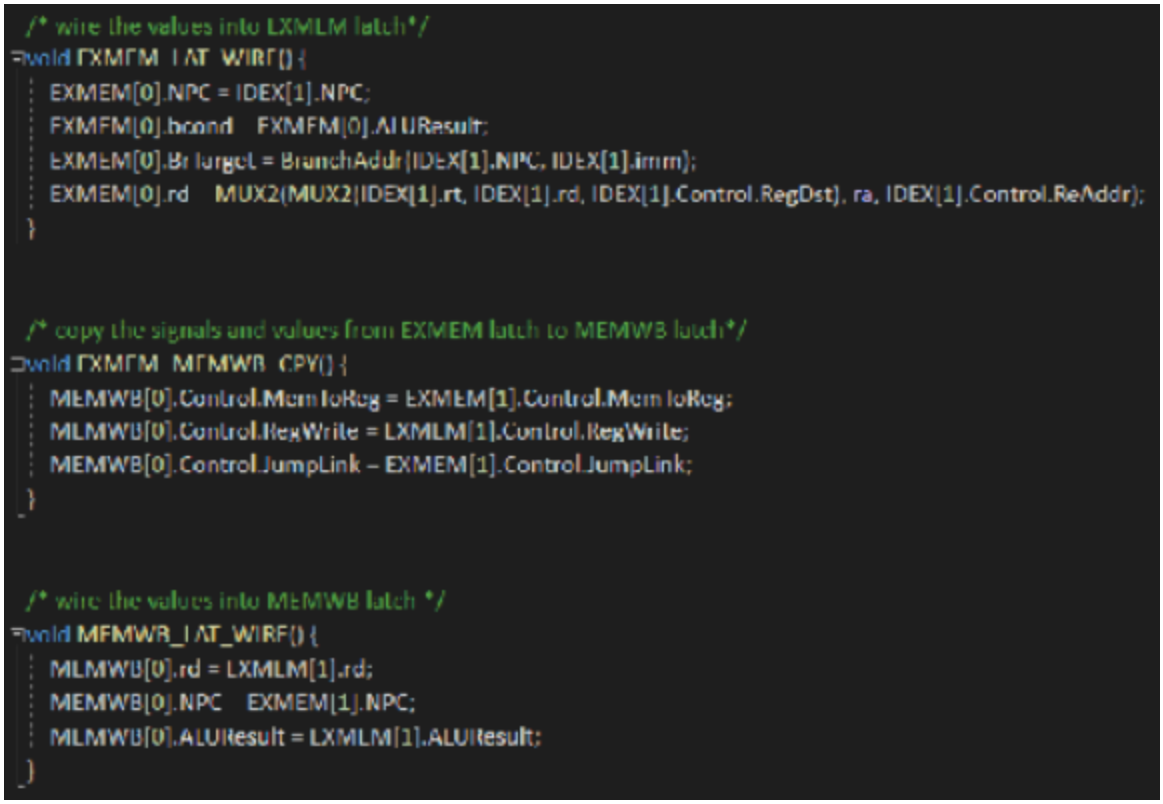
그림 23에 제시된 위험 감지 장치는 다음과 같은 제어 신호를 생성해야 합니다. PCWrite, IF/IDWrite, ID/EXTRISrc. PCWrite 신호는 PC 값을 기록할지 여부를 결정합니다. IF/IDWrite 신호는 IF/ID 레지스터 쓰기를 결정하며, ID/EXtrlSrc는 ID/EX 레지스터에 대한 제어 입력을 선택합니다. 마이크로아키텍처에 위험 감지 장치를 적용하여 부하 사용 데이터 종속성을 해결할 수 있습니다.

4.Implementation

4-1 Latch

```
/* wire the values into IDEX latch*/
void IDLX_LAT_WIRE() {
    IDLX[0].rs = inst->rs;
    IDEX[0].rt = inst->rt;
    IDEX[0].rd = inst->rd;
    IDEX[0].NPC = IFID[1].NPC;
    IDEX[0].funct = inst->funct;
    IDLX[0].shamt = inst->shamt;
    IDLX[0].opcode = inst->opcode;
    IDEX[0].IRIIdx = IFID[1].IRIIdx;
    IDFX[0].PHTIdx = IFID[1].PHTIdx;
    IDFX[0].LHTIdx = IFID[1].LHTIdx;
    IDFX[0].Control.PreTaken = IFID[1].PreTaken;
    IDEX[0].Control.BHTFound = IFID[1].BHTFound;
    IDEX[0].imm = MUX2(/*excel xlend(inst->immediate), Signl xlend(inst->immediate), Signl x);
}


/* copy the signals and values from IDEX latch to EXMEM latch */
void IDEX_EXMEM_CPY() {
    EXMEM[0].Control.ALUOp = IDEX[1].Control.ALUOp;
    EXMEM[0].Control.ALUSrc = IDEX[1].Control.ALUSrc;
    EXMEM[0].Control.Branch = IDEX[1].Control.Branch;
    EXMEM[0].Control.Equal = IDLX[1].Control.Equal;
    EXMEM[0].Control.JumpLink = IDEX[1].Control.JumpLink;
    EXMEM[0].Control.JumpReg = IDEX[1].Control.JumpReg;
    EXMEM[0].Control.MemRead = IDEX[1].Control.MemRead;
    EXMEM[0].Control.MemToReg = IDEX[1].Control.MemToReg;
    EXMEM[0].Control.MemWrite = IDEX[1].Control.MemWrite;
    EXMEM[0].Control.RegWrite = IDLX[1].Control.RegWrite;
    EXMEM[0].Control.RType = IDEX[1].Control.RType;
    EXMEM[0].Control.Shift = IDFX[1].Control.Shift;
}
```

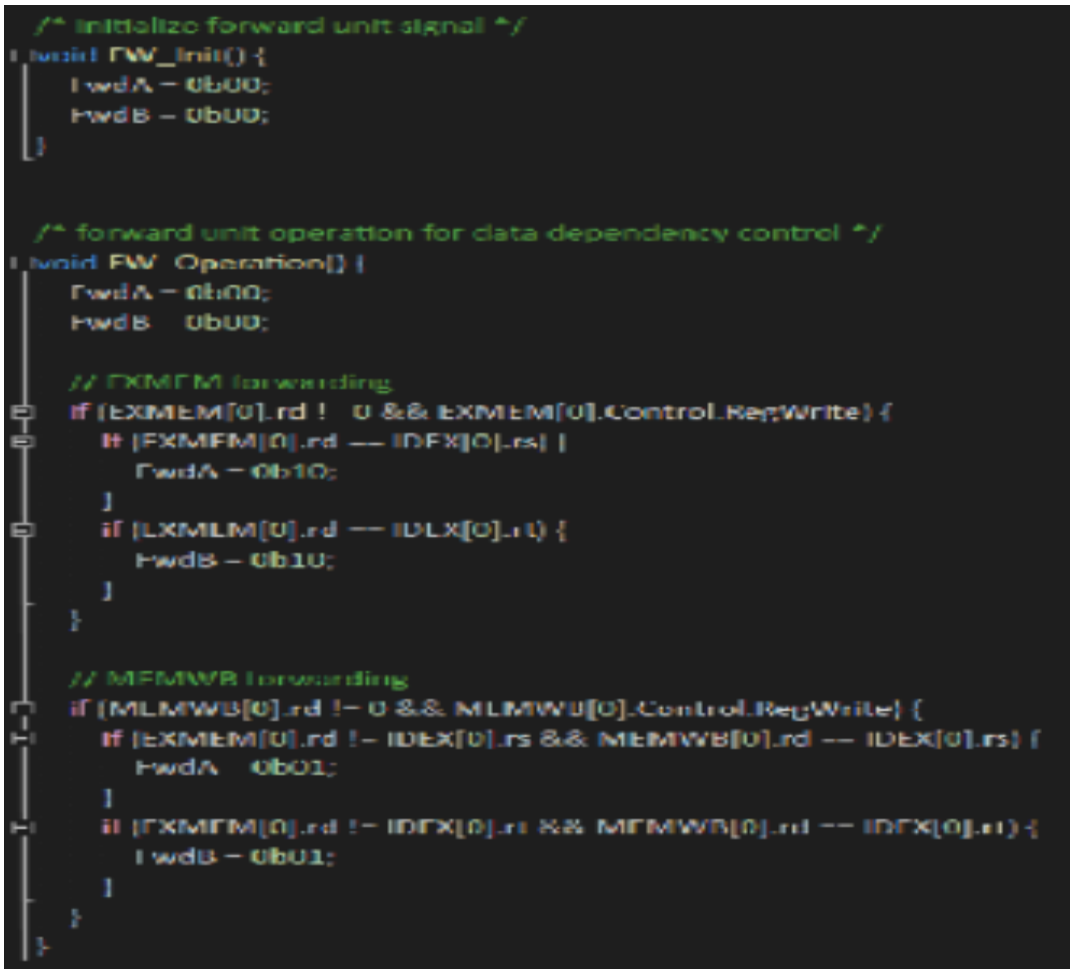
```
/* initialize latches of each stage*/
void LAT_Init() {
    IFID[0].valid = false;
    IDEX[0].valid = false;
    EXMEM[0].valid = false;
    MEMWB[0].valid = false;

    memset(&IFID[0], 0, sizeof(IFID_LAT));
    memset(&IFID[1], 0, sizeof(IFID_LAT));
    memset(&IDFX[0], 0, sizeof(IDFX_LAT));
    memset(&IDEX[1], 0, sizeof(IDEX_LAT));
    memset(&EXMEM[0], 0, sizeof(EXMEM_LAT));
    memset(&EXMEM[1], 0, sizeof(EXMEM_LAT));
    memset(&MEMWB[0], 0, sizeof(MEMWB_LAT));
    memset(&MEMWB[1], 0, sizeof(MEMWB_LAT));
}


/* update latches */
void LAT_Update() {
    memcpy(&IFID[1], &IFID[0], sizeof(IFID_LAT));
    memcpy(&IDFX[1], &IDFX[0], sizeof(IDFX_LAT));
    memcpy(&EXMEM[1], &EXMEM[0], sizeof(EXMEM_LAT));
    memcpy(&MEMWB[1], &MEMWB[0], sizeof(MEMWB_LAT));

    memset(&IFID[0], 0, sizeof(IFID_LAT));
    memset(&IDLX[0], 0, sizeof(IDLX_LAT));
    memset(&EXMEM[0], 0, sizeof(EXMEM_LAT));
    memset(&MEMWB[0], 0, sizeof(MEMWB_LAT));
}
```



```
/* wire the values into LXMLM latch*/
void EXMEM_LAT_WIRE() {
    EXMEM[0].NPC = IDEX[1].NPC;
    EXMEM[0].bcond   EXMEM[0].ALUResult;
    EXMEM[0].Br_target = BranchAddr(IDEX[1].NPC, IDEX[1].imm);
    EXMEM[0].rd    MUX2(MUX2(IDEX[1].rt, IDEX[1].rd, IDEX[1].Control.RegDst), ra, IDEX[1].Control.ReAddr);
}


/* copy the signals and values from EXMEM latch to MEMWB latch*/
void EXMEM_MEMWB_CPY() {
    MEMWB[0].Control.MemtoReg = EXMEM[1].Control.MemtoReg;
    MEMWB[0].Control.RegWrite = LXMLM[1].Control.RegWrite;
    MEMWB[0].Control.JumpLink = EXMEM[1].Control.JumpLink;
}


/* wire the values into MEMWB latch */
void MEMWB_LAT_WIRE() {
    MEMWB[0].rd = LXMLM[1].rd;
    MEMWB[0].NPC    EXMEM[1].NPC;
    MEMWB[0].ALUResult = LXMLM[1].ALUResult;
}
```

위의 그림은 래치 작동에 사용되는 기능을 보여줍니다. LAT_Init() 함수는 각 래치의 모든 유효 비트를 false로 초기화하고 래치의 값을 0으로 설정합니다. LAT_Update() 함수는 각 래치의 모든 값을 왼쪽에서 오른쪽으로 이동시키고 왼쪽 값을 0으로 설정합니다. "WIRE"로 끝나는 함수는 이전 또는 현재 단계의 명령에 의해 생성된 값을 배선합니다. "CPY"로 끝나는 기능은 이전 단계의 제어 신호를 복사합니다. 위 그림에 나와 있는 래치 기능을 구현하여 각 단계에서 래치를 제어할 수 있습니다.

4-2. Forward Unit

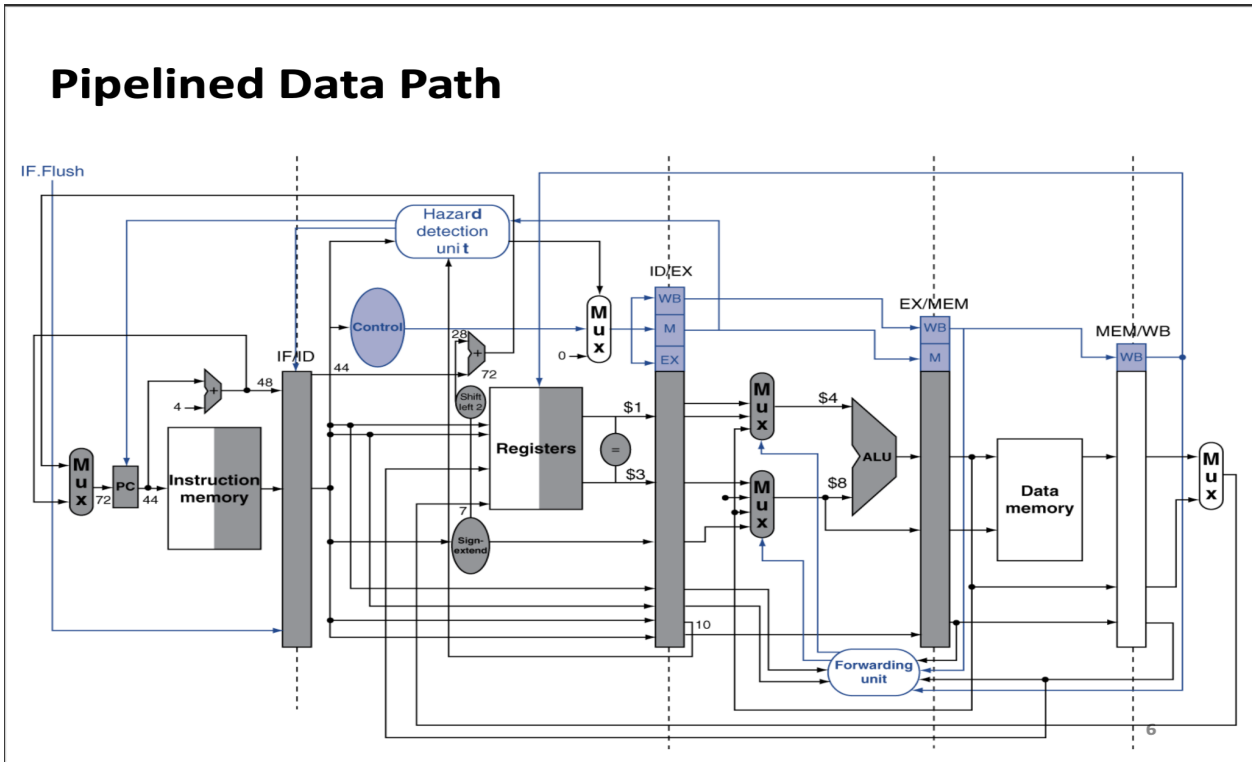


위 그림은 다음 조건에 따른 코드 구현입니다. 고려 사항 중 하나는 기능 작동의 초기 단계에서 빨간색 사각형으로 표시된 대로 초기 전달 제어 신호를 0으로 설정해야 한다는 것입니다. 그 이유는 데이터 의존성이 발생하지 않기 때문에 원래 RS 또는 RT 값을 출력해야 하기 때문입니다. 다음 코드를 구현하고 사이클 후에 실행함으로써 값을 전달하고 일반적인 경우에 대한 데이터 종속성을 해결할 수 있습니다.

4-3. Hazard Detection Unit

```
/* initialize hazard detection unit signals */
void HDU_Init() {
    PCWrite = true;
    IFIDWrite = true;
    IDEXCtrlSrc = false;
}


/* hazard detection unit operation */
void HDU_Operation() {
    if (IDLX[0].Control.MemRead && (IDLX[0].rt == Inst->rs || IDLX[0].rt == Inst->rt)) {
        PCWrite = false;
        IFIDWrite = false;
        IDLXCtrlSrc = true;
    }
    else {
        PCWrite = true;
        IFIDWrite = true;
        IDEXCtrlSrc = false;
    }
}
```

위 사진은 데이터 종속성을 해결하려면 Detection Unit이 필요한대



위그림 대로 흘러가게 조건에 따른 코드 구현입니다. 이 코드에서 중요한 부분은 제어 신호가 조건에 맞지 않을 경우 항상 제어 신호를 원래 값으로 설정해야 한다는 것으로 다음 기능을 코드로 구현 함으로써 Detection Unit으로 부하 사용 데이터 종속성을 해결 합니다.

4-4. Main

```
/* main */
void main(int argc, char* argv[]) {
    PROGRAM = argv[1];
    /* initialization */
    init();

    /* cycle start */
    do {
        // LINE;
        cycle++;

        /* execute the stages */
        MEM();
        WB(); // write back first so that write operation happens before read operation
        IF();
        ID();
        EX();

        /* fix the data hazards and the delay slot due to memory read operaiton*/
        HU_Operation();
        FW_Operation();

        /* print out the logs */
         LOG_Print();

        /* update the latches */
        LAT_Update();
        // LINE;
        if (PC == 0x18ecc) {
            printf("Cycle: %d\n", Register[v1]);

        }
    } while (PC < 0xffffffff);

    /* termination */
    terminate();
}
```

파이프라인 MIPS 시뮬레이터의 주요 기능에 대한 소스 코드를 보여 줍니다. 먼저 명령 페치(IF), 명령 디코드(ID), 실행(EX), 메모리 액세스(MEM), 쓰기 되돌리기(WB)의 5단계를 실행합니다. 둘째, 전방 장치와 위험 감지 장치를 사용하여 데이터 종속성을 해결합니다. 데이터를 전달한 후 로그를 출력하고 래치를 업데이트하겠습니다.

이 소스 코드에는 두 가지 고려 사항이 있습니다. 첫째, 단계 실행 순서는 일반적인 단일 주기 실행과 약간 다릅니다. 메모리 액세스 단계 및 라이트 백 단계가 초기에 실행됩니다. 이러한 방식으로 사이클을 구현한 이유는 데이터 종속성을 해결하려면 레지스터 쓰기 작업이 레지스터 읽기 작업보다 먼저 수행되어야 하기 때문입니다. 두 번째는 실행 단계 직후의 데이터 전달 및 데이터 위험 감지입니다. 이렇게 한 이유는 데이터 종속 레지스터의 값이 레지스터 파일에 기록되기 전에 결정되어야 하기 때문입니다. 다음과 같은 고려 사항을 포함하는 주요 기능을 구현함으로써, 우리는 적절한 파이프라인 MIPS 시뮬레이터를 구축할 수 있습니다.

4-5. IF

```
/* IF (instruction fetch) stage*/
void IF() {
    if (PC != 0xffffffff) {
        total_instruction++;
        IFID[0].valid = true; // change the valid bit into 1
        IFID[0].IR = IM_ReadMemory(PC); // fetch the instruction

        PC = PCAddr(PC); IFID[0].NPC = PC;
        }
    }
```

IF(명령어 가져오기) 함수의 소스 코드를 보여 줍니다. 먼저 유효한 비트를 True로 변경하여 IF/ID 래치가 작동을 시작할 수 있도록 합니다. 그런 다음 명령 메모리에서 명령을 읽고 PLECORD 변수에 의해 선택된 다음 예측 변수에 따라 예측을 수행합니다.

4-6. ID

```
void ID() {
    if (IFID[1].valid) {
        /* instruction decode */
        inst->opcode = (IFID[1].IR >> 26) & 0x3f;
        inst->rs = (IFID[1].IR >> 21) & 0x1f;
        inst->rt = (IFID[1].IR >> 16) & 0x1f;
        inst->rd = (IFID[1].IR >> 11) & 0x1f;
        inst->shamt = (IFID[1].IR >> 6) & 0x1f;
        inst->funct = (IFID[1].IR >> 0) & 0x3f;
        inst->immediate = (IFID[1].IR >> 0) & 0xffff;
        inst->address = (IFID[1].IR >> 0) & 0x3ffffff;

        /* execute stage */
        RF_Read(inst->rs, inst->rt);
        CU_Operation(inst->opcode, inst->funct);

        IDEX[0].valid = true; // change the valid bit into 1
        if (!IDEXCtrlSrc) IDEX_EXMEM_CPY();
        else memset(&IDEX[0].Control, 0, sizeof(IDEX_SIG)); // if data hazard is detected, flush the control signals
        IDEX_LAT_WIRE();

        if (PCWrite && Jump) jump_instruction++;
        if (PCWrite) PC = MUX2(PC, JumpAddr(IFID[1].NPC, inst->address), Jump);
    }
}
```

소스 명령어 디코딩(ID) 기능을 보여줍니다. 이전 단계의 유효한 비트가 참일 때 작동합니다. 첫째, MIPS ISA에 따라 주어진 명령어를 디코딩합니다. 두 번째로 레지스터에서 값을 읽고 제어 신호를 생성합니다. 셋째, 유효한 비트를 True로 변경하여 ID/EX 래치가 작동을 시작할 수 있도록 합니다. 데이터 종속성이 감지되면 제어 신호가 0으로 플러시됩니다. 그렇지 않으면 제어 신호를 복사하고 값을 다음 래치에 배선합니다. 마지막에 점프 작업이 실행됩니다.

4-7. EX

```
void EX() {
    if (IDEX[1].valid) {
        /* execute stage */
        uint32_t ALUControl = ALU_Control(IDEX[1].Control.ALUOp, MUX2(IDEX[1].opcode, IDEX[1].funct, IDEX[1].Control.Rtype));
        EXMEM[0].ReadData1 = MUX3(IDEX[1].ReadData1, MUX2(MEMWB[1].ALUResult, MEMWB[1].ReadData, MEMWB[1].Control.MemToReg), EXMEM[1].ALUResult, FwdA);
        EXMEM[0].ReadData2 = MUX3(IDEX[1].ReadData2, MUX2(MEMWB[1].ALUResult, MEMWB[1].ReadData, MEMWB[1].Control.MemToReg), EXMEM[1].ALUResult, FwdB);
        EXMEM[0].ALUResult = ALU_Operation(ALUControl, MUX2(EXMEM[0].ReadData1, EXMEM[0].ReadData2, IDEX[1].Control.Shift), MUX2(MUX2(EXMEM[0].ReadData2, IDEX[1].imm, IDEX[1].Control.ALUSrc),
        ANT_Execution();
        EXMEM[0].valid = true; // change the valid bit into 1
        EXMEM_MEMWB_CPY();
        EXMEM_LAT_WIRE();

        if (PCWrite && IDEX[1].Control.JumpReg) jump_instruction++;
        if (PCWrite) PC = MUX2(PC, EXMEM[0].ReadData1, IDEX[1].Control.JumpReg);
    }
}
```

소스 실행 코드(EX) 함수를 보여 줍니다. 먼저 유효한 비트가 참인지 확인하고, 참이면 작동합니다. 그런 다음 전달된 데이터를 고려하여 레지스터 값을 읽습니다. 이러한 값은 순방향 제어 신호에 의해 제어되는 3개의 입력 데이터를 가져오는 멀티플렉서에 의해 결정됩니다. 셋째, EX/MEM 래치가 작동을 시작할 수 있도록 유효한 비트를 true로 변경합니다. 그런 다음 제어 신호와 와이어 값을 이전 단계에서 다음 래치로 복사합니다. 마지막에 점프 레지스터 작업을 실행합니다.

4-8. MEM

```
void MEM() {
    if (EXMEM[1].valid) {
        DM_MemoryAccess(EXMEM[1].ALUResult, 32, EXMEM[1].ReadData2, EXMEM[1].Control.MemRead, EXMEM[1].Control.MemWrite);
        MEMWB[0].valid = true; // change the valid bit into 1
        MEMWB_LAT_WIRE();
    }
}
```

에는 MEM(메모리 액세스) 기능의 소스 코드가 나와 있습니다. 이전 단계의 유효한 비트가 참일 때 작동합니다.

메모리 액세스 제어 신호가 참이면 데이터 메모리에 액세스하여 로드 또는 저장 작업을 수행합니다. 그런 다음 MEM/WB 래치가 작동하도록 유효한 비트를 변경하고 값을 다음 래치로 배선합니다.

4-9. WriteBack and Init

```
void WB() {
    if (MEMWB[1].valid)
        RF_Write(MEMWB[1].rd, MUX2(MUX2(MEMWB[1].ALUResult, MEMWB[1].ReadData, MEMWB[1].Control.MemToReg), MEMWB[1].NPC, MEMWB[1].Control.JumpLink), MEMWB[1].Control.RegWrite);
}


/* initialize all of the hardware components */
void init() {
    RF_Init();
    IM_Init();
    CU_Init();
    DM_Init();
    FW_Init();
    HU_Init();
    LAT_Init();
    LOG_Init();

    PC = 0x00000000;
    inst = (instruction*)malloc(sizeof(instruction));
}
```

WB(write back) 함수의 소스 코드입니다. 이전 단계의 유효한 비트가 참일 때 작동합니다. 레지스터 쓰기 제어 신호가 참이면 멀티플렉서에서 쓰기 되돌림 값을 선택하여 레지스터 파일에 저장합니다. 그리고 init을 통해 다시 값을 넣어줍니다.

5. Build Environment for Testing Simulator

5-1. Build Environments and make command

실행방법은 먼저 환경이 리눅스 환경이어야합니다. vi editor,gcc compiler가 필요합니다 환경이아니라면 groomide에서 리눅스환경을 만들고 파일을 업로드한후 실행해야 됩니다. 먼저 IM.c폴더에서 `fp_in = fopen("input1/input.bin", "rb");`

이부분의 경로를 바꿔주시고

터밀널에 명령어를 아래처럼 해주세요.

make clean -> clean all object

make main -> build program 해주시고

./main 으로 실행 해주시면 됩니다.

5-2. run screen

input1.bin

```
Cycle: 1
[IF] :: PC: 00000000, IR: 27bdfff0
[ID] :: Stall
[EX] :: Stall
[MA] :: Stall
[WB] :: Stall
Cycle: 2
[IF] :: PC: 00000004, IR: afbe000c
[ID] :: opcode: (09) rs: 29 rt: 29 immediate: fff0
[EX] :: Stall
[MA] :: Stall
[WB] :: Stall
Cycle: 3
[IF] :: PC: 00000008, IR: 03a0f025
[ID] :: opcode: (2b) rs: 29 rt: 30 immediate: 000c
[EX] :: R[29]: 00fffff0 = R[29] + fffffff0
[MA] :: Stall
[WB] :: Stall
Cycle: 4
[IF] :: PC: 0000000c, IR: afc00000
[ID] :: opcode: (00) rs: 29 rt: 0 rd: 30 shamt: 00 funct: (25)
[EX] :: Store Memory Address: 00fffffc = R[29] + 0000000c
[MA] :: No Memory Access
[WB] :: Stall
Cycle: 5
[IF] :: PC: 00000010, IR: afc00004
[ID] :: opcode: (2b) rs: 30 rt: 0 immediate: 0000
[EX] :: R[30]: 00fffff0 = R[29] | R[0]
[MA] :: No Memory Access
[WB] :: R[29]: 00fffff0
Cycle: 6
[IF] :: PC: 00000014, IR: 1000000a
[ID] :: opcode: (2b) rs: 30 rt: 0 immediate: 0004
[EX] :: Store Memory Address: 00fffff0 = R[30] + 00000000
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 7
[IF] :: PC: 00000018, IR: 00000000
[ID] :: opcode: (04) rs: 0 rt: 0 immediate: 000a
[EX] :: Store Memory Address: 00fffff4 = R[30] + 00000004
[MA] :: No Memory Access
[WB] :: R[30]: 00fffff0
Cycle: 8
[IF] :: PC: 0000001c, IR: 8fc30000
[ID] :: Stall
[EX] :: Jump to  00000014 = R[0] == R[0]
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 9
[IF] :: PC: 00000040, IR: 8fc20004
[ID] :: Stall
[EX] :: Stall
[MA] :: No Memory Access
```

```
Cycle: 9
[IF] :: PC: 00000040, IR: 8fc20004
[ID] :: Stall
[EX] :: Stall
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 10
[IF] :: PC: 00000044, IR: 00000000
[ID] :: opcode: (23) rs: 30 rt: 2 immediate: 0004
[EX] :: Stall
[MA] :: Stall
[WB] :: No Register Write Back
Cycle: 11
[IF] :: PC: 00000048, IR: 2842000a
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: Load Memory Address: 00fffff4 = R[30] + 00000004
[MA] :: Stall
[WB] :: Stall
Cycle: 12
[IF] :: PC: 0000004c, IR: 1440fff3
[ID] :: opcode: (0a) rs: 2 rt: 2 immediate: 000a
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: ReadData: 00000000 = M[R[00fffff4]
[WB] :: Stall
Cycle: 13
[IF] :: PC: 00000050, IR: 00000000
[ID] :: opcode: (05) rs: 2 rt: 0 immediate: fff3
[EX] :: R[2]: 1 = (R[2] < 0000000a)? 1 : 0
[MA] :: No Memory Access
[WB] :: R[2]: 00000000
Cycle: 14
[IF] :: PC: 00000054, IR: 8fc20000
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: Branch not equal X = R[2] == R[0]
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
Cycle: 15
[IF] :: PC: 00000058, IR: 03c0e825
[ID] :: opcode: (23) rs: 30 rt: 2 immediate: 0000
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: R[2]: 00000001
```

```
Cycle: 14
[IF] :: PC: 00000054, IR: 8fc20000
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: Branch not equal X = R[2] == R[0]
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
Cycle: 15
[IF] :: PC: 00000058, IR: 03c0e825
[ID] :: opcode: (23) rs: 30 rt: 2 immediate: 0000
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: R[2]: 00000001
Cycle: 16
[IF] :: PC: 0000005c, IR: 8fbe000c
[ID] :: opcode: (00) rs: 30 rt: 0 rd: 29 shamt: 00 funct: (25)
[EX] :: Load Memory Address: 00fffff0 = R[30] + 00000000
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 17
[IF] :: PC: 00000060, IR: 27bd0010
[ID] :: opcode: (23) rs: 29 rt: 30 immediate: 000c
[EX] :: R[29]: 00fffff0 = R[30] | R[0]
[MA] :: ReadData: 00000000 = M[R[00fffff0]
[WB] :: R[0]: 00000000
Cycle: 18
[IF] :: PC: 00000064, IR: 03e00008
[ID] :: opcode: (09) rs: 29 rt: 29 immediate: 0010
[EX] :: Load Memory Address: 00fffffc = R[29] + 0000000c
[MA] :: No Memory Access
[WB] :: R[2]: 00000000
Cycle: 19
[IF] :: PC: 00000068, IR: 00000000
[ID] :: opcode: (00) rs: 31 rt: 0 rd: 0 shamt: 00 funct: (08)
[EX] :: R[29]: 01000000 = R[29] + 00000010
[MA] :: ReadData: 00000000 = M[R[00fffffc]
[WB] :: R[29]: 00fffff0
Cycle: 20
[IF] :: Program Terminated
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: PC: 00000064 = R[31]
[MA] :: No Memory Access
[WB] :: R[30]: 00000000

******************result********************
Final return value: 0
Number of executed instruction: 20
Number of executed memory access instruction: 6
Number of executed register operation instruction: 7
Number of executed taken branch instruction: 2
Number of jump instruction: 1
********************************************
```

input2.bin

```
Cycle: 1
[IF] :: PC: 00000000, IR: 27bdffd8
[ID] :: Stall
[EX] :: Stall
[MA] :: Stall
[WB] :: Stall
Cycle: 2
[IF] :: PC: 00000004, IR: afbf0024
[ID] :: opcode: (09) rs: 29 rt: 29 immediate: ffd8
[EX] :: Stall
[MA] :: Stall
[WB] :: Stall
Cycle: 3
[IF] :: PC: 00000008, IR: afbe0020
[ID] :: opcode: (2b) rs: 29 rt: 31 immediate: 0024
[EX] :: R[29]: 00ffffd8 = R[29] + ffffffd8
[MA] :: Stall
[WB] :: Stall
Cycle: 4
[IF] :: PC: 0000000c, IR: 03a0f025
[ID] :: opcode: (2b) rs: 29 rt: 30 immediate: 0020
[EX] :: Store Memory Address: 00fffffc = R[29] + 00000024
[MA] :: No Memory Access
[WB] :: Stall
Cycle: 5
[IF] :: PC: 00000010, IR: 24020004
[ID] :: opcode: (00) rs: 29 rt: 0 rd: 30 shamt: 00 funct: (25)
[EX] :: Store Memory Address: 00fffff8 = R[29] + 00000020
[MA] :: No Memory Access
[WB] :: R[29]: 00ffffd8
Cycle: 6
[IF] :: PC: 00000014, IR: afc2001c
[ID] :: opcode: (09) rs: 0 rt: 2 immediate: 0004
[EX] :: R[30]: 00ffffd8 = R[29] | R[0]
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 7
[IF] :: PC: 00000018, IR: 8fc4001c
[ID] :: opcode: (2b) rs: 30 rt: 2 immediate: 001c
[EX] :: R[2]: 00000004 = R[0] + 00000004
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 8
[IF] :: PC: 0000001c, IR: 0c00000f
[ID] :: opcode: (23) rs: 30 rt: 4 immediate: 001c
[EX] :: Store Memory Address: 00fffff4 = R[30] + 0000001c
[MA] :: No Memory Access
[WB] :: R[30]: 00ffffd8
```

```
Cycle: 9
[IF] :: PC: 00000020, IR: 00000000
[ID] :: opcode: (03) address: 000000f
[EX] :: Load Memory Address: 00fffff4 = R[30] + 0000001c
[MA] :: No Memory Access
[WB] :: R[2]: 00000004
Cycle: 10
[IF] :: PC: 0000003c, IR: 27bdffe0
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: R[31]: 00000020; PC: 0000001c
[MA] :: ReadData: 00000004 = M[R[00fffff4]
[WB] :: No Register Write Back
Cycle: 11
[IF] :: PC: 00000040, IR: afbf001c
[ID] :: opcode: (09) rs: 29 rt: 29 immediate: ffe0
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: R[4]: 00000004
Cycle: 12
[IF] :: PC: 00000044, IR: afbe0018
[ID] :: opcode: (2b) rs: 29 rt: 31 immediate: 001c
[EX] :: R[29]: 00ffffb8 = R[29] + ffffffe0
[MA] :: No Memory Access
[WB] :: R[31]: 00000020
Cycle: 13
[IF] :: PC: 00000048, IR: 03a0f025
[ID] :: opcode: (2b) rs: 29 rt: 30 immediate: 0018
[EX] :: Store Memory Address: 00ffffd4 = R[29] + 0000001c
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
Cycle: 14
[IF] :: PC: 0000004c, IR: afc40020
[ID] :: opcode: (00) rs: 29 rt: 0 rd: 30 shamt: 00 funct: (25)
[EX] :: Store Memory Address: 00ffffd0 = R[29] + 00000018
[MA] :: No Memory Access
[WB] :: R[29]: 00ffffb8
Cycle: 15
[IF] :: PC: 00000050, IR: 8fc30020
[ID] :: opcode: (2b) rs: 30 rt: 4 immediate: 0020
[EX] :: R[30]: 00ffffb8 = R[29] | R[0]
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 16
[IF] :: PC: 00000054, IR: 24020001
[ID] :: opcode: (23) rs: 30 rt: 3 immediate: 0020
[EX] :: Store Memory Address: 00ffffd8 = R[30] + 00000020
[MA] :: No Memory Access
[WB] :: No Register Write Back
```

```
Cycle: 17
[IF] :: PC: 00000058, IR: 14620004
[ID] :: opcode: (09) rs: 0 rt: 2 immediate: 0001
[EX] :: Load Memory Address: 00ffffd8 = R[30] + 00000020
[MA] :: No Memory Access
[WB] :: R[30]: 00fffb8
Cycle: 18
[IF] :: PC: 0000005c, IR: 00000000
[ID] :: opcode: (05) rs: 3 rt: 2 immediate: 0004
[EX] :: R[2]: 00000001 = R[0] + 00000001
[MA] :: ReadData: 00000004 = M[R[00ffffd8]
[WB] :: No Register Write Back
Cycle: 19
[IF] :: PC: 00000060, IR: 24020001
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: Branch not equal X = R[3] == R[2]
[MA] :: No Memory Access
[WB] :: R[3]: 00000004
Cycle: 20
[IF] :: PC: 00000064, IR: 1000000b
[ID] :: opcode: (09) rs: 0 rt: 2 immediate: 0001
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: R[2]: 00000001
Cycle: 21
[IF] :: PC: 00000068, IR: 00000000
[ID] :: opcode: (04) rs: 0 rt: 0 immediate: 000b
[EX] :: R[2]: 00000001 = R[0] + 00000001
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 22
[IF] :: PC: 0000006c, IR: 8fc20020
[ID] :: Stall
[EX] :: Jump to  00000064 = R[0] == R[0]
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
Cycle: 23
[IF] :: PC: 00000094, IR: 03c0e825
[ID] :: Stall
[EX] :: Stall
[MA] :: No Memory Access
[WB] :: R[2]: 00000001
Cycle: 24
[IF] :: PC: 00000098, IR: 8fbf001c
[ID] :: opcode: (00) rs: 30 rt: 0 rd: 29 shamt: 00 funct: (25)
[EX] :: Stall
[MA] :: Stall
[WB] :: No Register Write Back
```

```
Cycle: 25
[IF] :: PC: 0000009c, IR: 8fbe0018
[ID] :: opcode: (23) rs: 29 rt: 31 immediate: 001c
[EX] :: R[29]: 00ffffb8 = R[30] | R[0]
[MA] :: Stall
[WB] :: Stall
Cycle: 26
[IF] :: PC: 000000a0, IR: 27bd0020
[ID] :: opcode: (23) rs: 29 rt: 30 immediate: 0018
[EX] :: Load Memory Address: 00ffffd4 = R[29] + 0000001c
[MA] :: No Memory Access
[WB] :: Stall
Cycle: 27
[IF] :: PC: 000000a4, IR: 03e00008
[ID] :: opcode: (09) rs: 29 rt: 29 immediate: 0020
[EX] :: Load Memory Address: 00ffffd0 = R[29] + 00000018
[MA] :: ReadData: 00000020 = M[R[00ffffd4]
[WB] :: R[29]: 00ffffb8
Cycle: 28
[IF] :: PC: 000000a8, IR: 00000000
[ID] :: opcode: (00) rs: 31 rt: 0 rd: 0 shamt: 00 funct: (08)
[EX] :: R[29]: 00ffffd8 = R[29] + 00000020
[MA] :: ReadData: 00ffffd8 = M[R[00ffffd0]
[WB] :: R[31]: 00000020
Cycle: 29
[IF] :: PC: 000000ac, IR: 00000000
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: PC: 000000a4 = R[31]
[MA] :: No Memory Access
[WB] :: R[30]: 00ffffd8
Cycle: 30
[IF] :: PC: 00000020, IR: 00000000
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: R[29]: 00ffffd8
Cycle: 31
[IF] :: PC: 00000024, IR: 03c0e825
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 32
[IF] :: PC: 00000028, IR: 8fbf0024
[ID] :: opcode: (00) rs: 30 rt: 0 rd: 29 shamt: 00 funct: (25)
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
```

```
Cycle: 31
[IF] :: PC: 00000024, IR: 03c0e825
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: No Register Write Back
Cycle: 32
[IF] :: PC: 00000028, IR: 8fbf0024
[ID] :: opcode: (00) rs: 30 rt: 0 rd: 29 shamt: 00 funct: (25)
[EX] :: R[0]: 00000000 = R[0] << R[0]
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
Cycle: 33
[IF] :: PC: 0000002c, IR: 8fbe0020
[ID] :: opcode: (23) rs: 29 rt: 31 immediate: 0024
[EX] :: R[29]: 00fffd8 = R[30] | R[0]
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
Cycle: 34
[IF] :: PC: 00000030, IR: 27bd0028
[ID] :: opcode: (23) rs: 29 rt: 30 immediate: 0020
[EX] :: Load Memory Address: 00ffffc = R[29] + 00000024
[MA] :: No Memory Access
[WB] :: R[0]: 00000000
Cycle: 35
[IF] :: PC: 00000034, IR: 03e00008
[ID] :: opcode: (09) rs: 29 rt: 29 immediate: 0028
[EX] :: Load Memory Address: 00ffff8 = R[29] + 00000020
[MA] :: ReadData: ffffffff = M[R[00ffffc]
[WB] :: R[29]: 00fffd8
Cycle: 36
[IF] :: PC: 00000038, IR: 00000000
[ID] :: opcode: (00) rs: 31 rt: 0 rd: 0 shamt: 00 funct: (08)
[EX] :: R[29]: 01000000 = R[29] + 00000028
[MA] :: ReadData: 00000000 = M[R[00ffff8]
[WB] :: R[31]: ffffffff
Cycle: 37
[IF] :: Program Terminated
[ID] :: opcode: (00) rs: 0 rt: 0 rd: 0 shamt: 00 funct: (00)
[EX] :: PC: 00000034 = R[31]
[MA] :: No Memory Access
[WB] :: R[30]: 00000000

******************result*********************
Final return value: 1
Number of executed instruction: 37
Number of executed memory access instruction: 12
Number of executed register operation instruction: 17
Number of executed taken branch instruction: 2
Number of jump instruction: 3
*********************************************
```

6. Lesson

파이프라인을 구현하고 마무리하면서 많은 소감과 배운 점을 얻었습니다. 첫째로, 파이프라인을 구현함으로써 명령어의 처리 속도와 성능을 획기적으로 향상시킬 수 있다는 것을 몸소 느낄 수 있었습니다. 이전에는 단일 사이클 처리 방식으로 작업을 수행했는데, 파이프라인을 도입하면서 동시에 여러 명령어를 병렬로 실행할 수 있게 되었습니다. 이는 작업을 효율적으로 분할하고 처리 시간을 단축시킬 수 있는 큰 장점이었습니다.

둘째로, 데이터 의존성과 관련하여 발생하는 **Hazard**를 관리하는 방법들에 대해 배울 수 있었습니다. 데이터 의존성이 발생할 때 스톨링이나 데이터 포워딩과 같은 기법을 적용하여 이를 해결할 수 있다는 것을 알게 되었습니다. 데이터 의존성을 효과적으로 관리하면 파이프라인의 성능을 향상시킬 수 있으며, 잘못된 결과를 예방할 수 있습니다. 이러한 **Hazard** 관리 기법은 컴퓨터 구조 설계에 중요한 역할을 하는 것을 깨달았습니다.

셋째로, 파이프라인의 설계와 동작 원리에 대한 이해를 높일 수 있었습니다. 파이프라인은 여러 단계로 구성되어 있고, 각 단계는 순차적으로 처리되는데, 이를 효율적으로 조율하는 것이 중요합니다. 파이프라인의 구성 요소와 각 단계의 처리 시간, 파이프라인 버퍼의 역할 등을 깊이 이해하게 되었습니다. 이를 통해 파이프라인을 최적화하고 성능을 극대화할 수 있는 방법들을 탐구할 수 있었습니다.