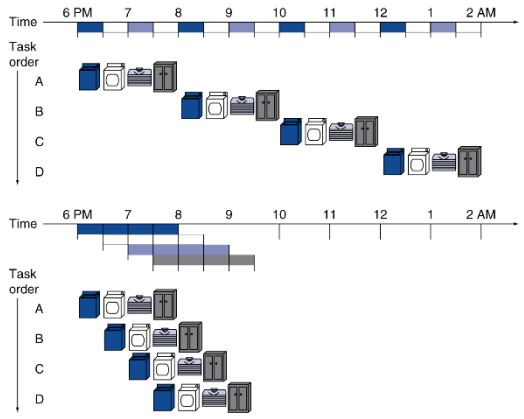
CA 4장 프로세서

4.5 파이프라이닝

파이프라이닝 : 여러 명령어가 중첩되어 실행되는 구현 기술

파이프라이닝은 단일 일의 끝내는데 걸리는 시간을 단축하지 못하지만, 해야 할 일이 많을 경우에는 처리량이 증가하므로 일을 끝내는데 걸리는 전체시간을 단축시킴



Four loads :

speedup=8/3.5=2.3

Non-stop "

speedup=2n/0.5n+1.5 ≈ 4

=number of stages

(4에 가까울라면 4보다 훨씬 커야함)

MIPS Pipeline

1. IF: Instruction fetch from memory

2. ID: Instruction decode & register read

3. EX: Execute operation or calculate address

4. MEM: Access memory operand

5. WB: Write result back to register

Review: Single Cycle vs. Multiple Cycle Timing

stage 레지스터 오버 헤드로 인해 단일 사이클 클록의 1/5보다 멀티 사이클 클럭이 느리다.

How Can We Make It Even Faster?

여러 명령어 사이클을 작고 작은 단계로 분할하십시오. 작업을 수행함에 따라 상태 레지스터를 로드하는 데 많은 시간이 소비되는 returns감소 지점이 있습니다 현재 명령어가 완료되기 전에 다음 명령어를 가져 와서 실행하십시오.

Pipelining – (all?) modern processors are pipelined for performance

A Pipelined MIPS Processor

현재 명령어가 완료되기 전에 다음 명령어를 가져 와서 실행하십시오.(파이프라인)

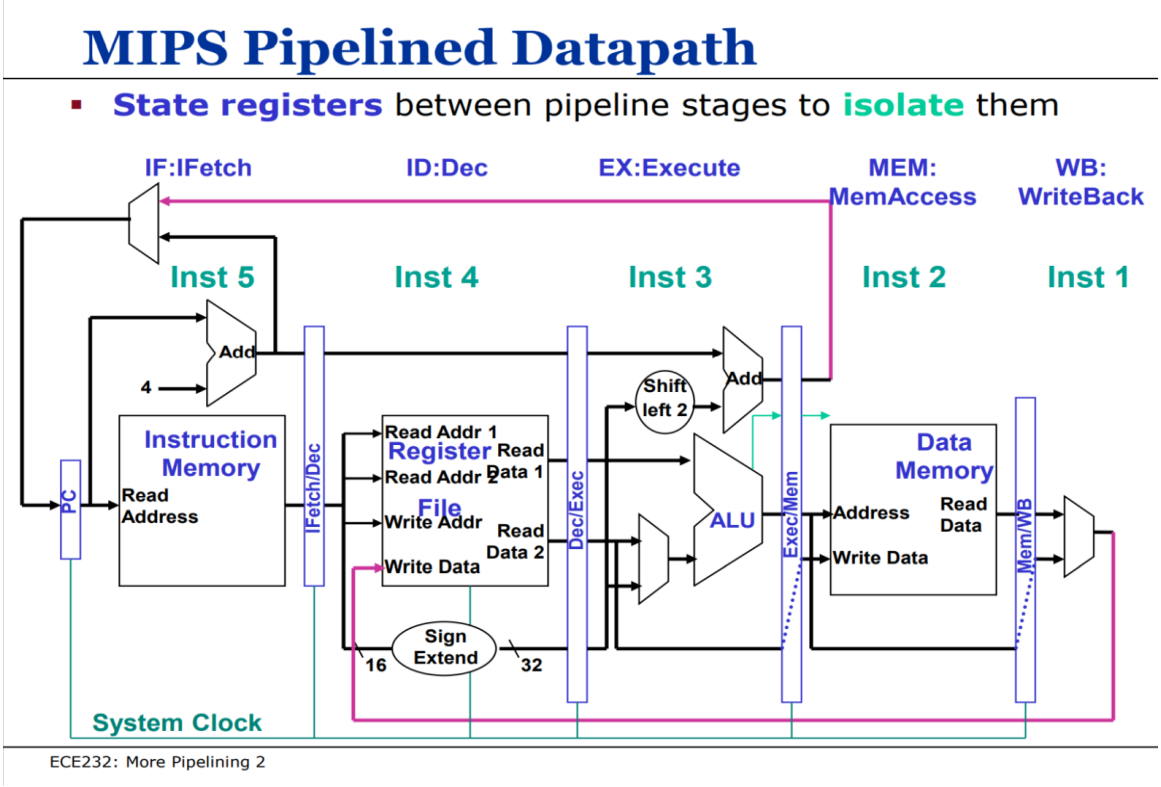
- throughput 증가 - 명령어 latency(실행시간) 은 줄지 않는다.

파이프 라인의 stage time은 가장 느린 자원에 의해 제한 받는데 ALU 연산이나 메모리 접근이 된다. 그래서 몇 개의 단계는 낭비되는 사이클이다.

Why Pipeline? For Performance!

Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

MIPS Pipeline Datapath Modifications



Pipeline Performance => 예제 책 275

Pipeline Speedup

$$\text{명령어 사이의 시간(pipelined)} = \frac{\textit{명령어 사이의 시간}(nonpipelined)}{\textit{파이프 단계}(stages)\textit{수}}$$

위에는 이상적인 경우다. 단계가 완벽하게 균형잡혀 있지 않거나 파이프라이닝이 어느정도 오베헤드를 포함하기 때문에 속도 향상은 단계 수보다 작아지게 된다.

파이프라이닝은 개별 명령어의 실행시간을 줄이지는 못하지만 명령어 처리량을 증가해 성능 향상

Pipelining and ISA Design

MIPS 명령어 집합은 원래 파이프라인 실행을 위해 설계된 것이다.

첫째, 모든 MIPS 명령어는 32 bit 같은 길이를 가진다.

한 사이클에서 fetch와 decode을 쉽게한다.

x86은 1바이트에서부터 15바이트까지 변하기 때문에 파이프라인닝이 힘듬

둘째, MIPS는 몇가지 안되는 명령어 형식을 가지고 있다.

모든 명령어에서 근원지 레지스터 필드는 같은 위치에 있다. 이 같은 대칭성은 decode와 read registers을 한단계에서 할 수 있게한다.

셋째, MIPS에서는 메모리 피연산자가 적재와 저장 명령어에서만 나타난다.

3번째 단계에서 주소를 계산하고 4번째단계에서 메모리 접근이 가능

넷째, 피연산자는 메모리에 정렬(align) 되어 있어야 한다.

메모리 접근은 only 한 사이클만 걸린다.

What makes it hard

- structural hazards, control hazards, data hazards

Pipeline Hazards : 다음 명려어가 다음 클럭 사이클에 실행될 수 없는 상황

Structural Hazard

같은 클럭 사이클에 실행하기를 원하는 명령어의 조합을 하드웨어가 지원할 수 없어서 계획된 명령어가 적절한 클럭 사이클에 실행될 수 없는 사건.

MIPS 에서 메모리가 하나라고 생각해보자. Load/Store은 data 접근이 필요하다. 만약 하나가 데이터에 접근하고 다른 하나가 메모리에서 명령어를 가져오면 메모리가 한 개이기 때문에 동시에 발생하지 못해서 구조적 해저드를 피할 수 없다.

따라서 파이프 라인 된 데이터 path에는 별도의 명령어 / 데이터 메모리가 필요합니다.

Data Hazard

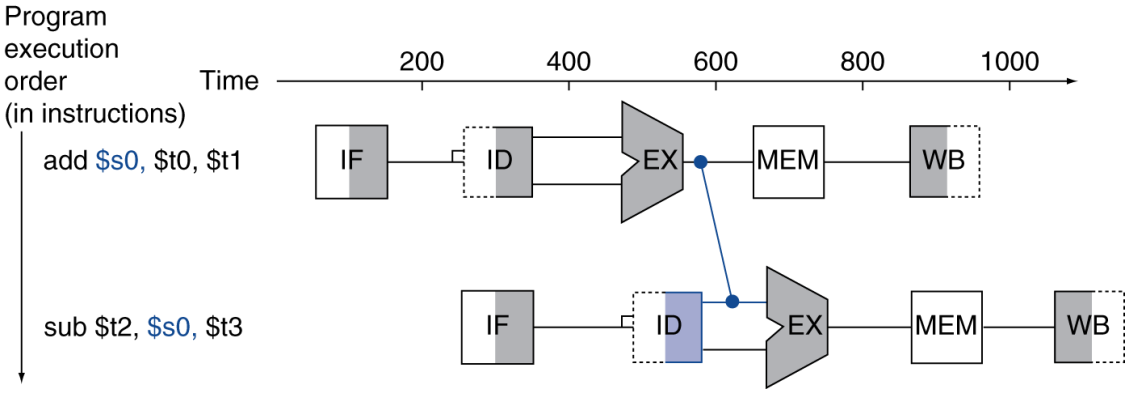
명령어를 실행하는데 필요한 데이터가 아직 준비되지 않아서 계획된 명령어가 적절한 클럭 사이클에 실행될 수 없는 사건.

파이프라인에서는 어떤 명령어가 아직 파이프라인에 있는 앞선 명령어에 종속성을 가질 때 데이터 해저드가 일어난다. 예를 들어 add 명령어 바로 다음에 add의 합($s0)을 사용하는 뺄셈 명령어가 뒤따라오면 add 명령어의 5번째 단계까지는 결과 값을 쓰지 않아서 파이프라인이 세개의 클럭 사이클을 낭비함.

add $s0, $t0, $t1
sub $t2, $s0, $t3

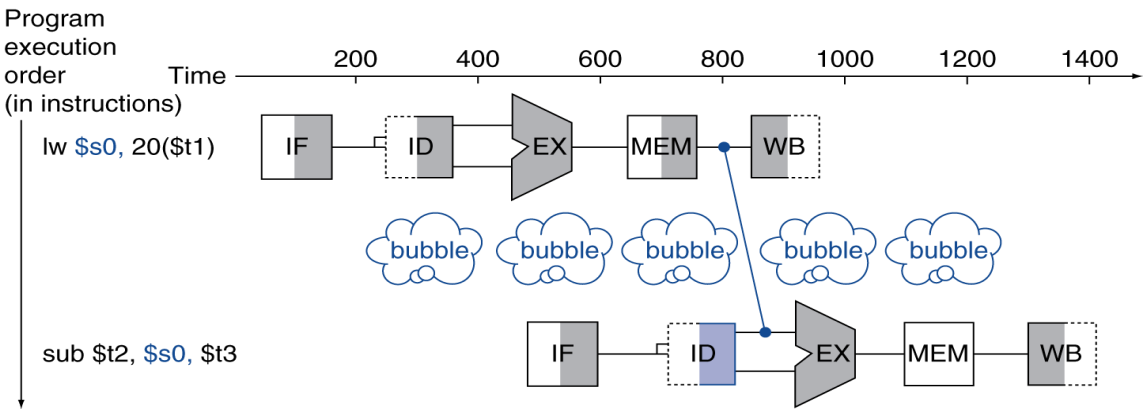
forwarding or bypassing

프로그래머가 볼 수 있는 레지스터나 메모리에 아직 나타나지 않은 데이터를 기다리기보다는 내부 버퍼로부터 가져옴으로써 데이터 해저드를 해결하는 방법.



forward 통로가 시간상 뒤로 돌아가는 것은 불가능.

load-use data hazard의 경우에는 forwarding을 해도 한 단계가가 지연되어야 한다. 파이프라인 지연 bublle은 해저드를 해결하기 위해 생기는 지연이다.



오른쪽 반이 어두운 것이 읽기 왼쪽 반이 어두운 것이 쓰기이다.

Code Scheduling to Avoid Stalls => 책 예제 p281

Control Hazards(branch hazard)

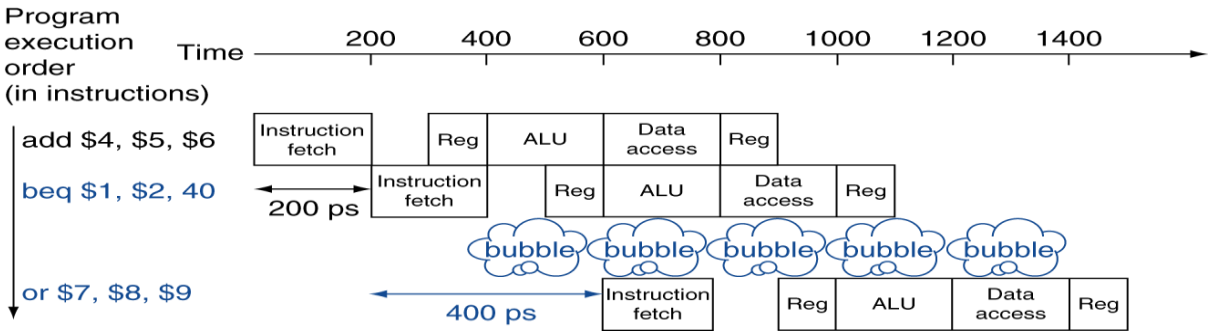
인출한 명령어가 필요한 명령어가 아니기 때문에 적절한 명령어가 적절한 클럭 사이클에 실행될 수 없는 사건. 명령어 주소의 흐름이 파이프라인이 기대한것과 다르기 때문에 발생.

다른 명령어들이 실행중에 한 명령어의 결과 값에 기반을 둔 결정을 할 필요가 있을 때 일어남.

두가지 해결책 => 지연, 예측

- **지연**

컴퓨터에서는 분기명령어가 이러한 것을 결정 작업에 해당. 메모리에서 방금 분기 명령어를 받았을뿐, 파이프라인은 다음 명령어가 어느 것인지 알 수 없다. 해결방법이 파이프라인이 분기의 결과를 판단하고 어느 주소에서 다음 명령어를 가져올지 알게 될 때까지 기다리는 것이다.

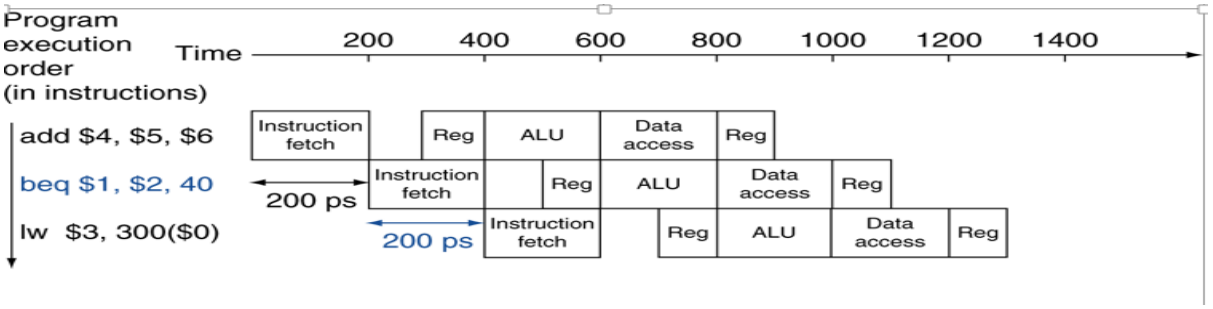


- **예측**

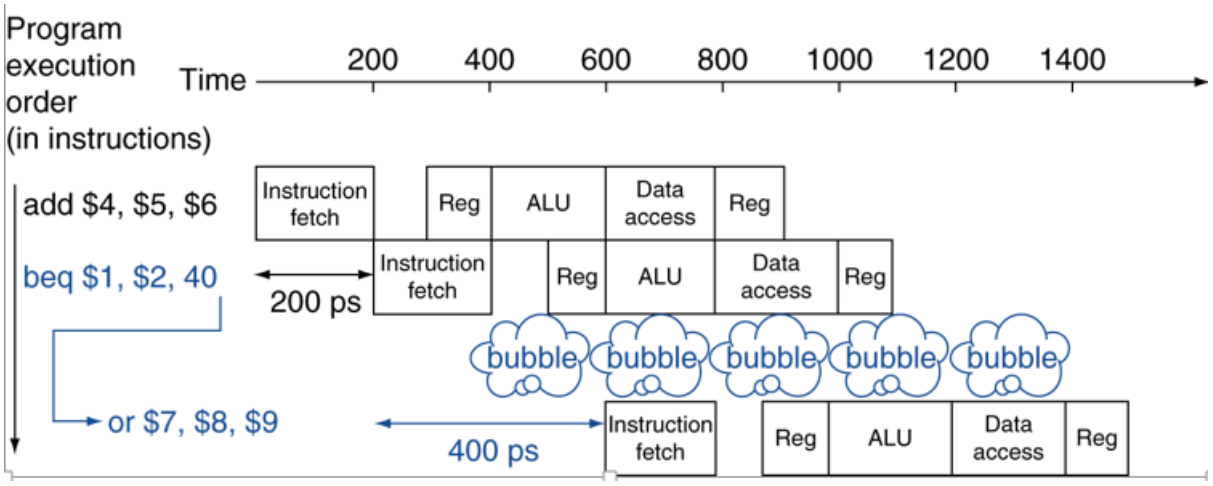
대부분의 컴퓨터가 분기 명령어를 다루기 위해서 예측을 사용한다. 간단한 방법은 분기가 항상 실패한다고 예측하는 것이다. 예측이 옳으면 파이프라인은 최고 속도, 아니면 실제로 분기가 일어날때만 파이프라인 지연된다.

Branch Prediction : 실제 분기 겨로가가 확인될 때까지 기다리는 대신, 분기 결과를 가정하고 그 가정하에 파이프라인을 진행해 나가는 분기 해저드 해결 방법 , 어떤 경우는 분기한다(taken)한다고 예측하고 어떤 경우는 분기하지 않는다고(untaken) 한다고 예측한다.

분기가 일어나지 않을 때의 파이프라인—



분기가 일어날 때의 파이프라인



More-Realistic Branch Prediction

- Static branch prediction : 보편적 행동에 의존하며 특정 분기 명령어의 개별성을 고려x

ex) 순환문의 끝에는 순환문의 꼭대기로 점프하라는 분기 명령어가 있다. 이 명령어들은 분기가 일어날 가능성이 높고 분기 방향이 후방이므로, 이에 착안하여 현재 위치보다 작은 주소로 점프하는 분기 명령어는 항상 분기로 예측

- Dynamic branch prediction : 개별 분기 명령어의 행동에 의존하는 예측을 하며 프로그램이 진행되는 도중에 예측을 바꿀 수 있다, 예측이 어긋날 경우 잘 못 예측한 분기 명령어 뒤에 나오는 명령어들을 무효화하고 올바른 분기 주소로부터 파이프라인을 다시 시작해야함.

두개의 해결책 모두다 긴 파이프라인에서 문제를 악화시킴.

요점정리

Pipelining은 동시에 실행되는 명령어의 수를 증가시키며 명령어들이 시작하고 끝나는 속도를 증가시킨다. 파이프라이닝은 각각의 명령어의 실행은 끝내는데 걸리는 시간을 단축시키지 않는데 이 시간을 지연시간이라고 부른다. 예를 들어 5 단계 파이프라인은 한 명령어가 끝나는데 5 클럭 사이클이 걸린다. 파이프라이닝은 각각의 명령어 지연시간보다는 처리율을 향상시킨다. 명령어 집합은 파이프라인 설계를 쉽게도 하고 어렵게도 한다. 파이프라인 설계자는 이미 구조적 해저드, 제어 해저드, 데이터 해저드 등과 맞닥뜨려 이를 해결해 왔다. 분기 예측, 전방전달은 올바른 결과를 얻으면서도 컴퓨터를 빠르게 하는데 도움이 된다.

4.6 파이프라인 데이터패스 및 제어

MIPS Pipeline

1. IF: Instruction fetch from memory

2. ID: Instruction decode & register read

3. EX: Execute operation or calculate address

4. MEM: Access memory operand

5. WB: Write result back to register

이렇게 명령어들은 왼쪽에서 오른쪽으로 움직인다. 하지만 두가지 예외가 있다.

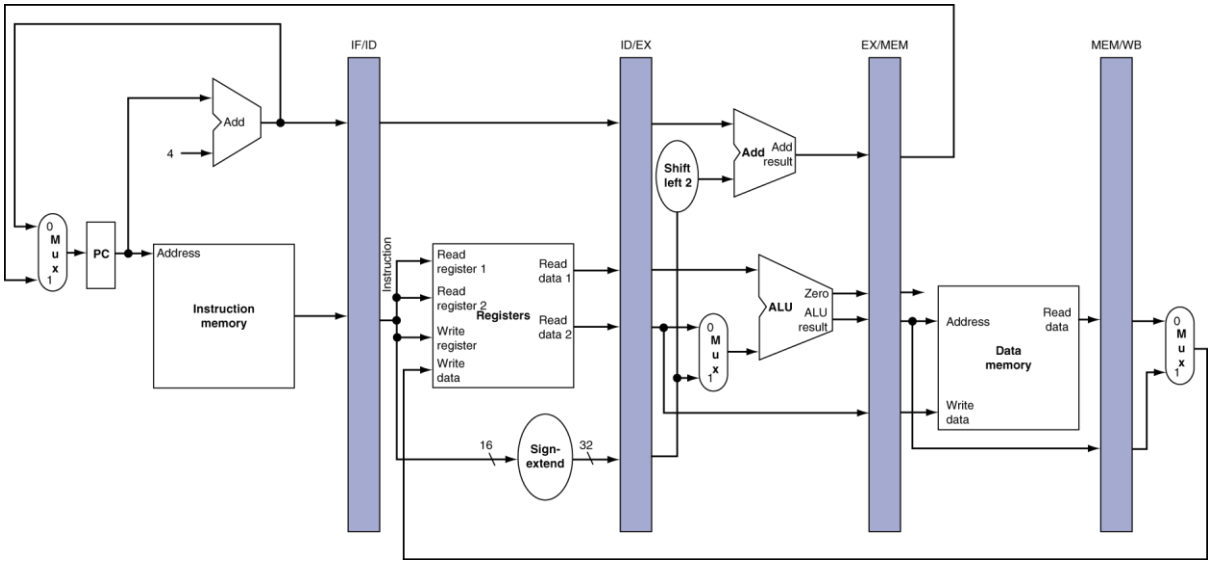
● 쓰기 단계 : 이 결과를 데이터패스의 중앙에 있는 레지스터 파일에다 쓴다. –데이터헤저드

● PC의 다음 값 선정 : 증가된 PC값과 MEME 단계의 분기 주소 중에서 고른다. –제어 헤저드

오른쪽에서 왼쪽으로 흐르는 데이터는 현재 명령어에 영향을 주지 않는다. 파이프라인의 뒤쪽에 있는 명령어들만이 이 같은 역방향 데이터 흐름에 영향을 받는다.

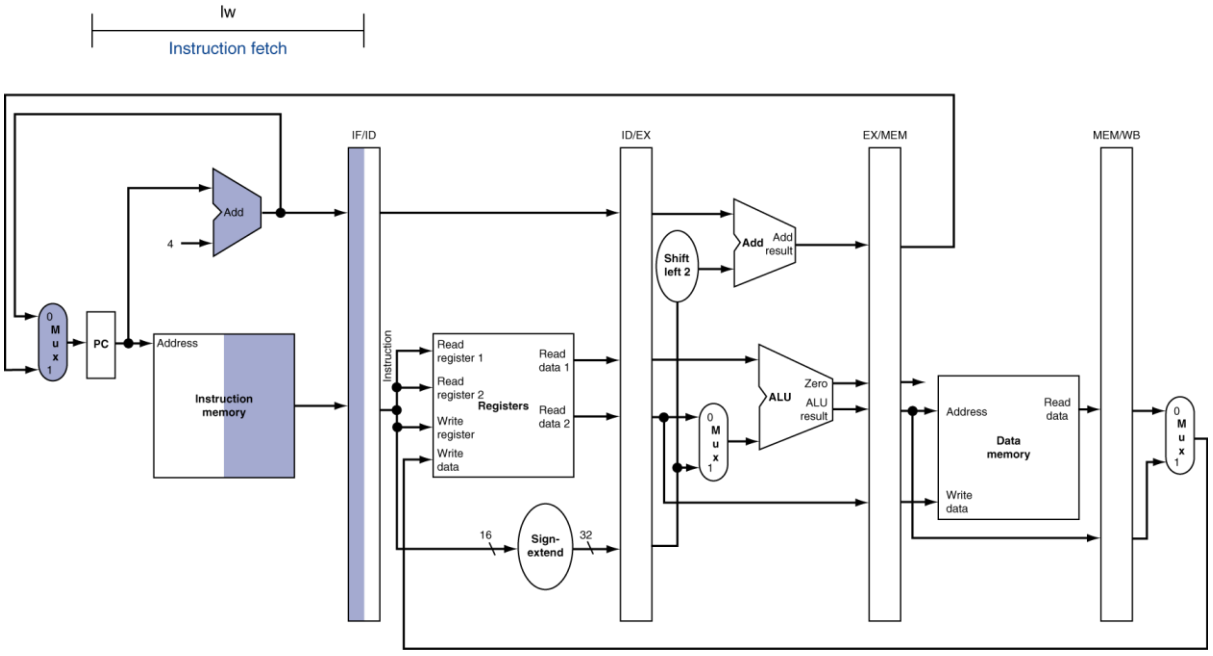
Pipeline registers

이전 사이클에서 생성된 정보를 보관하는 것.



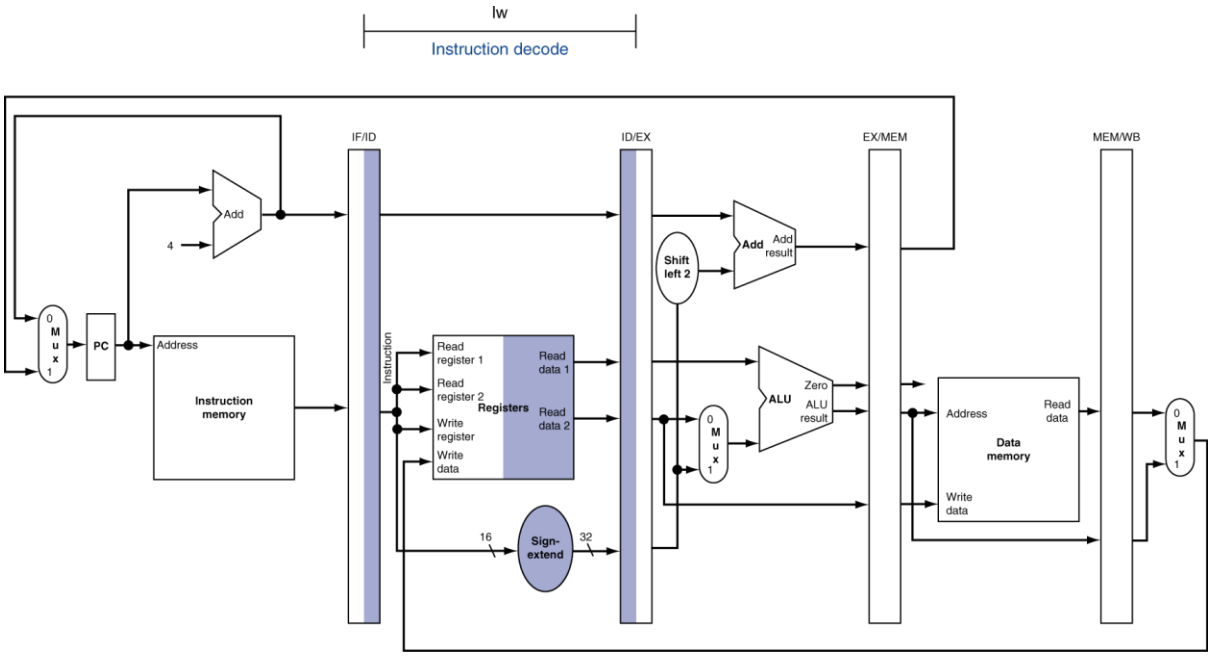
Load & Store => p293 ~p297

1. IF for lw

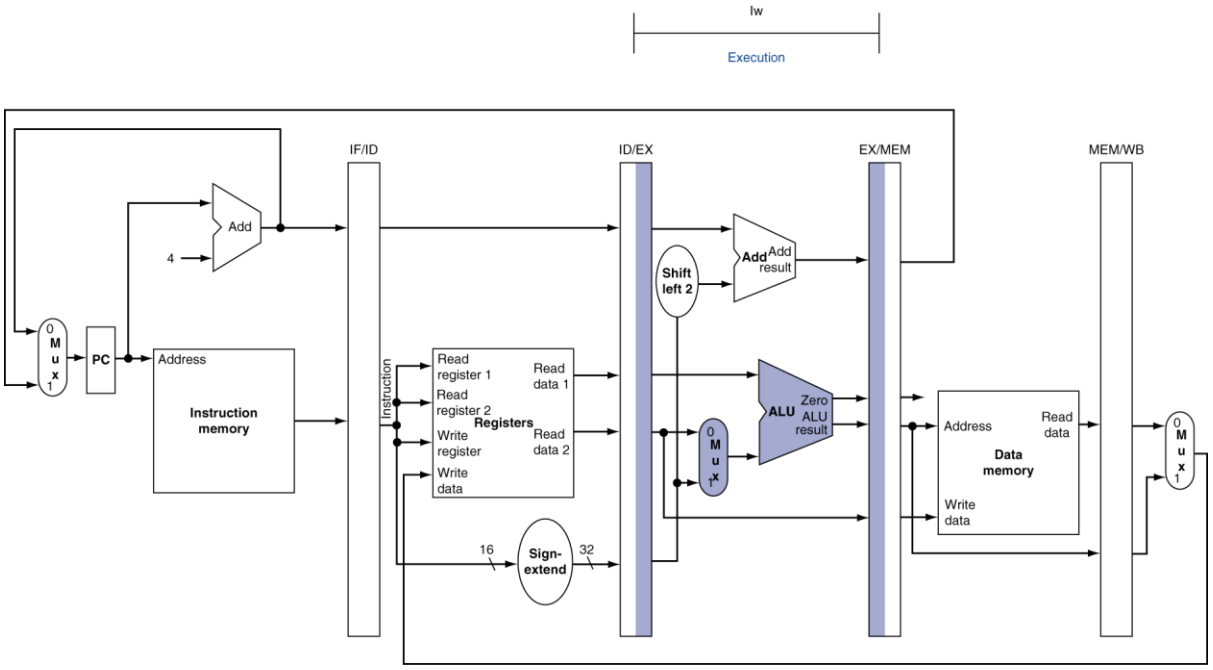


PC값에 4을 더한것도 IF/ID 레지스터에 저장한다. 그 이유는 BEQ와 같은 명령어처럼 뒤에 필요한 경우를 위해서이다.

2. ID for lw

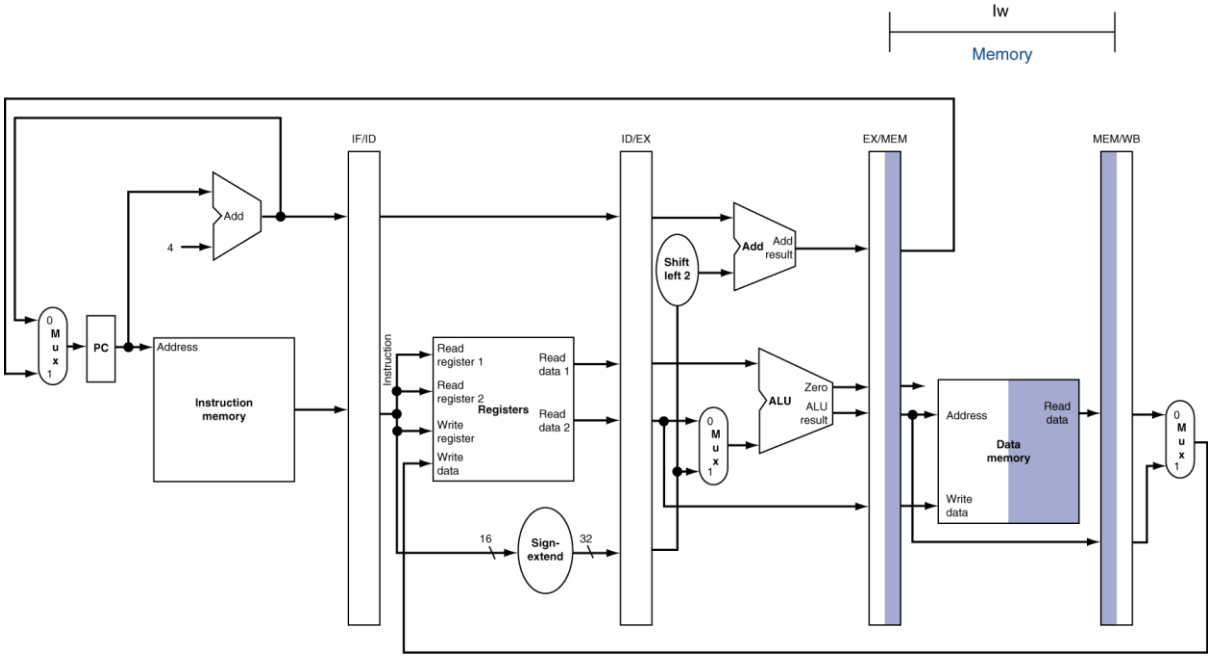


3. EX for lw

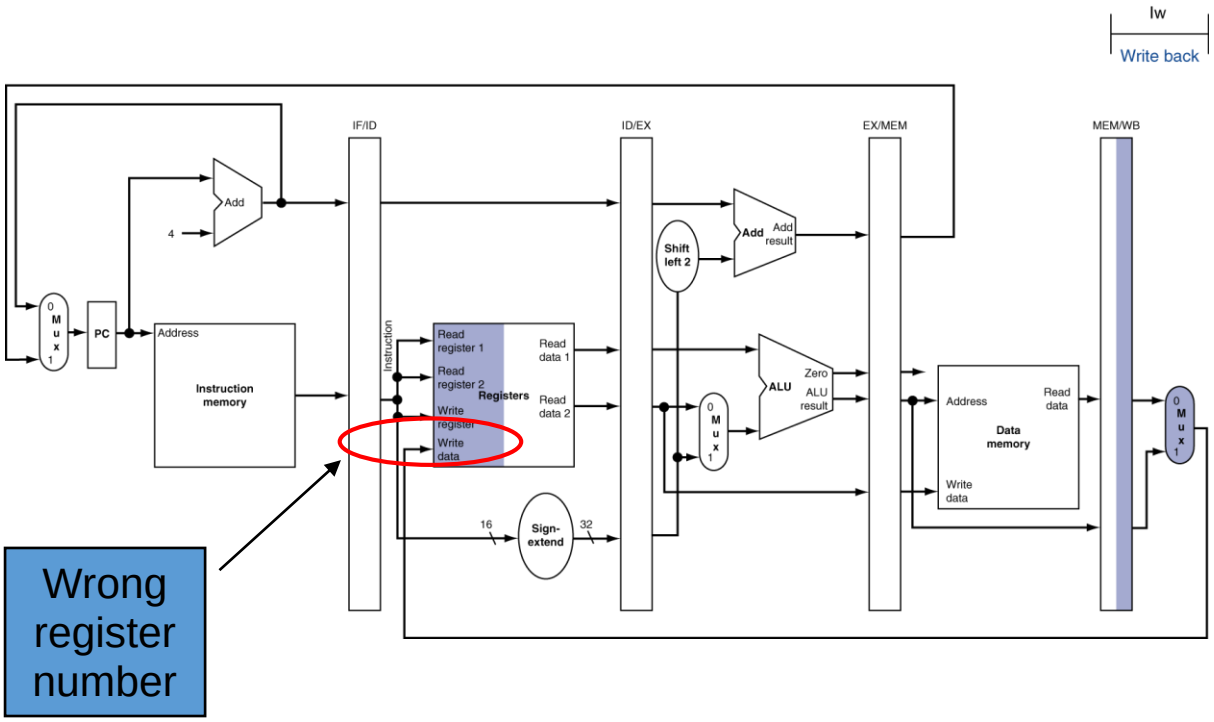


lw는 레지스터를 부호확장된 수치에 더해서 그 합을 EX/MEM 파이프라인 레지스터에 저장한다.

4. MEM for lw



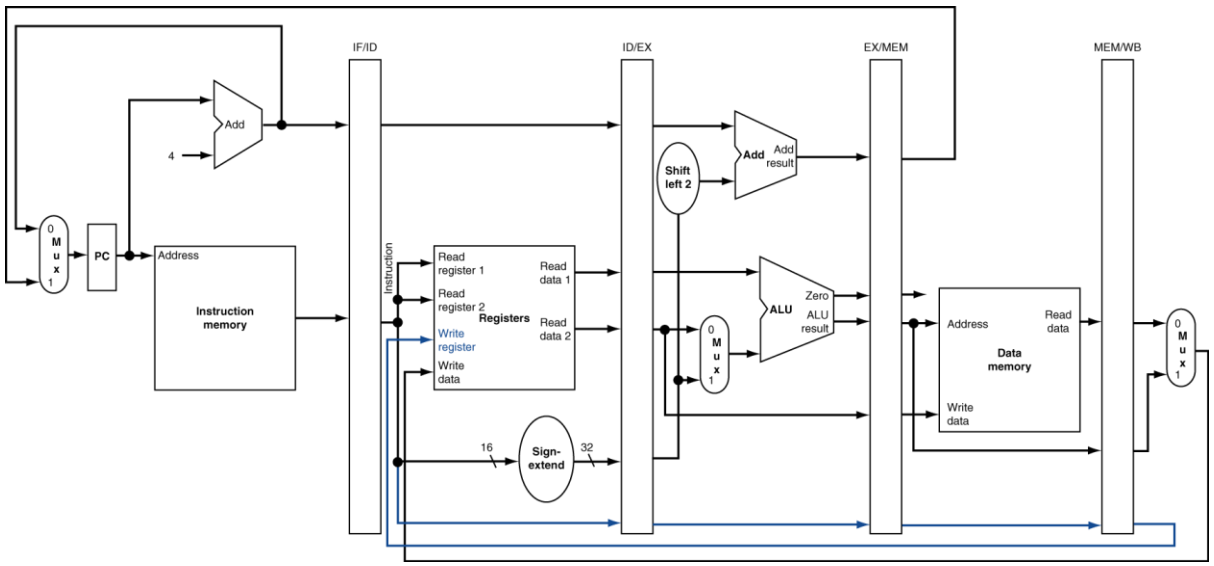
5. WB for lw



Corrected Datapath for Load

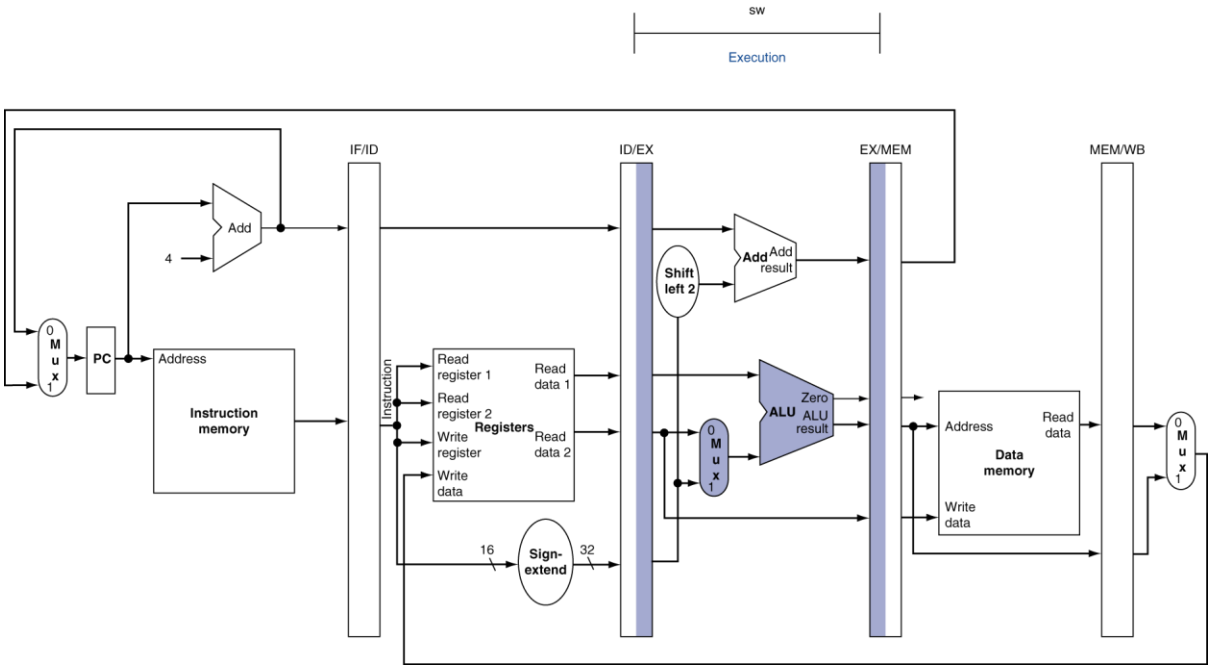
lw 명령어의 최종 단꼐에서 어느 레지스터가 변하는가? 즉 어느 명령어가 쓰기 레지스터 번호 제공하는가?

IF/ID 파이프라인 레지스터에 있는 명령어가 쓰기 레지스터 번호를 제공하는데, 사실 이 명령어는 적재 명령어보다 상당히 뒤에 실행되는 명령어이다. 따라서 쓰기 레지스터 번호는 데이터와 함께 MEM/WB 파이프라인 레지스터에 가져온다. 레지스터 번호는 ID 파이프 단계에서부터 MEM/WB 파이프라인 레지스터까지 전달되므로 이 마지막 세개의 파이프라인 레지스터에 5비트가 추가된다.



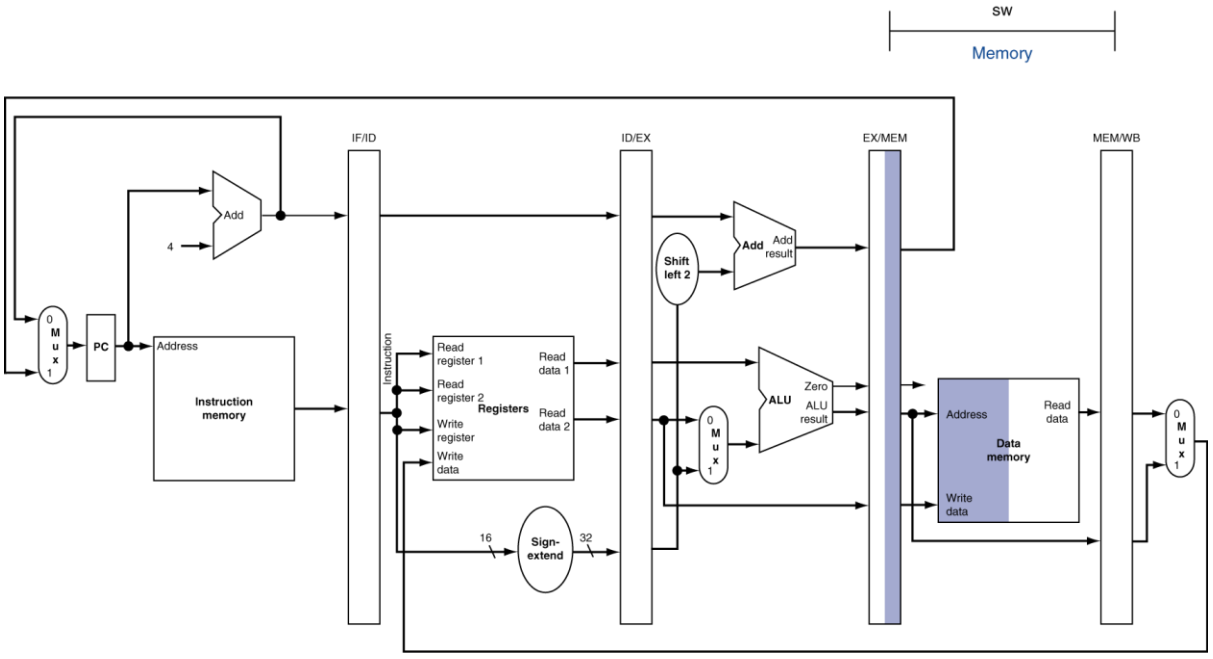
sw는 EX전까지는 똑같다.

Ex for sw

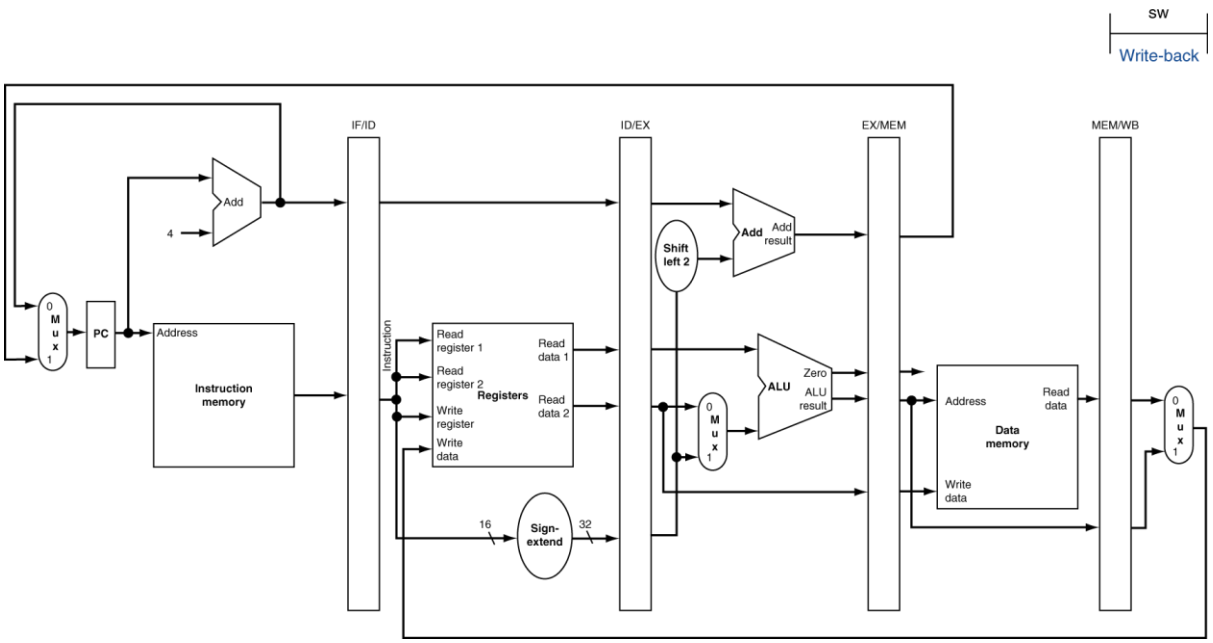


sw 는 lw 와 다르게 두번째 레지스터 값이 EX/MEM 파이프라인 레지스터에 적재되는데 이 값은 다음 단계에서 사용된다.

MEM for store



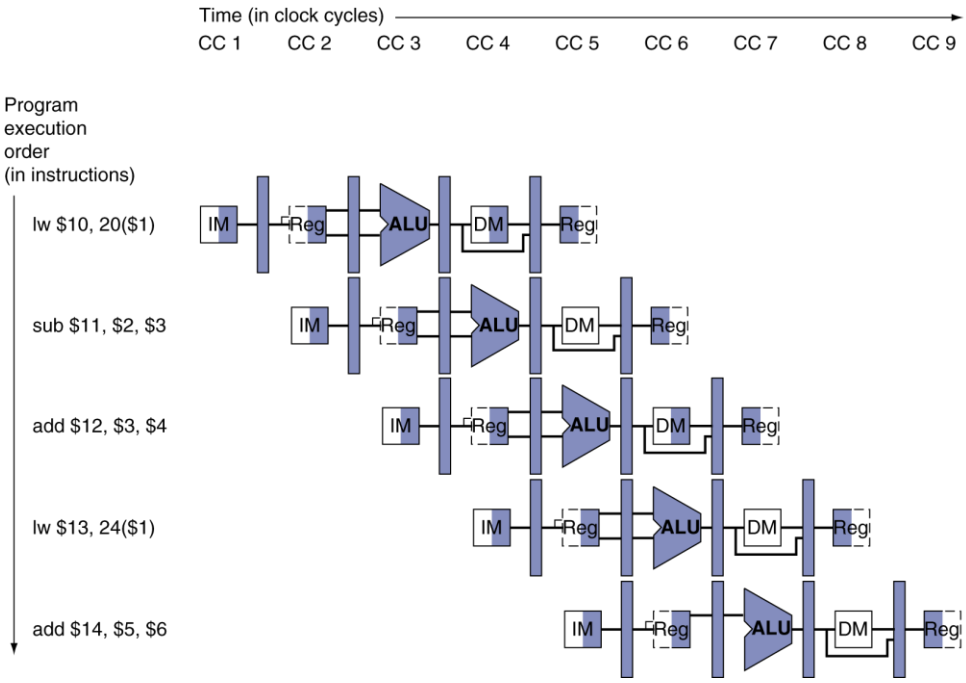
WB for Store



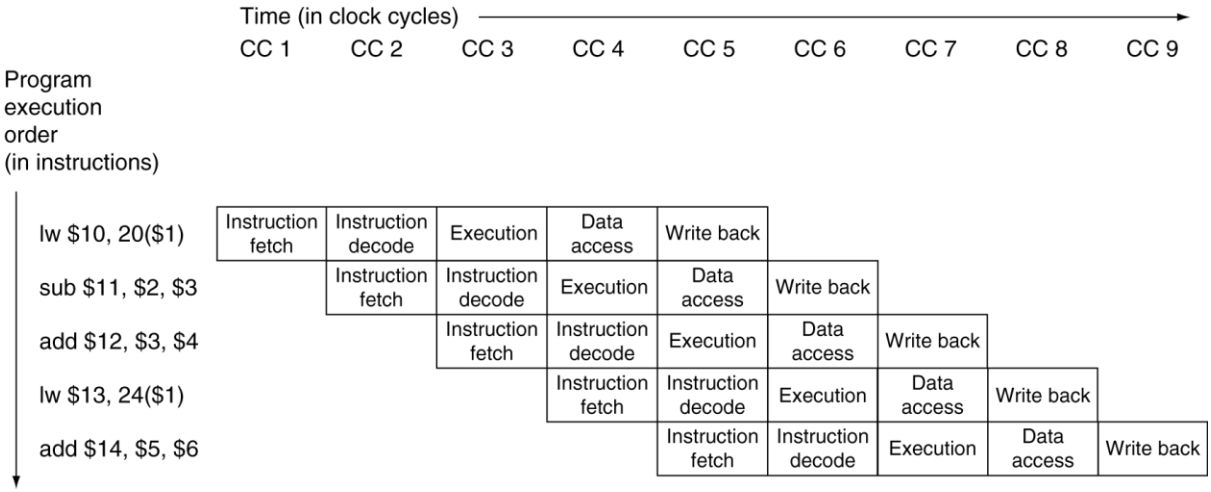
아무일도 없다.

데이터패스의 각 구성요소, 즉 명령어 메모리, 레지스터 읽기 포트, ALU, 데이터 메모리, 레지스터 쓰기 포트 등은 한 파이프라인 단계에서만 사용 가능. 그렇지 않으면 구조적 헤저드가 발생한다.

Multi-Cycle Pipeline Diagram



Traditional form



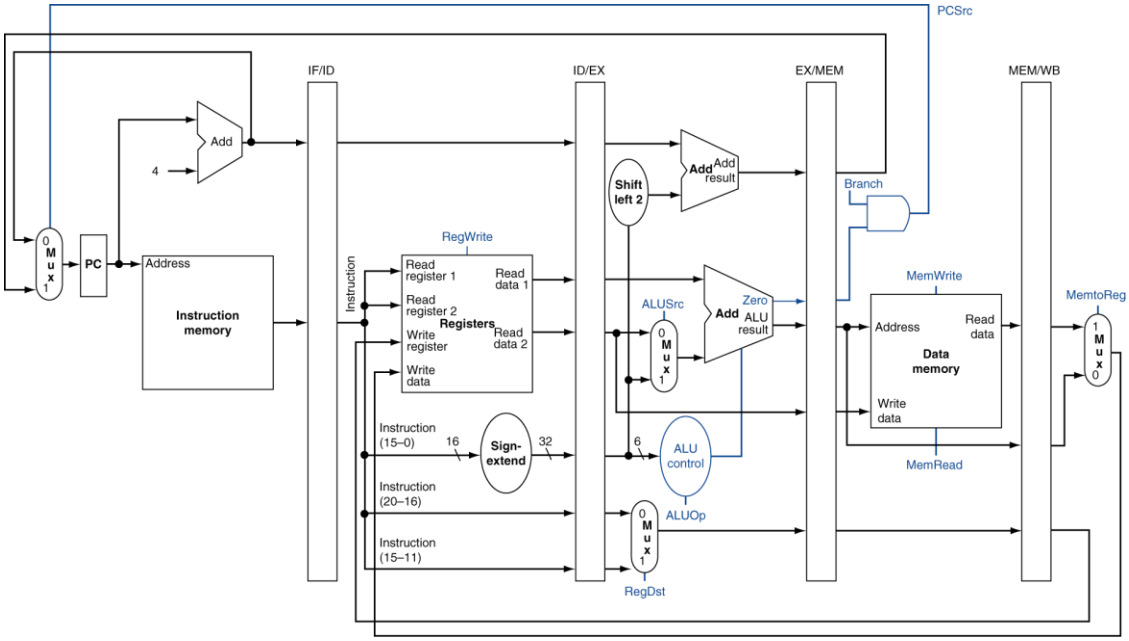
위와 같이 그려도 상관없다.

스스로 점검하기 답.

- 몇 개의 명령어가 더 적은 단계를 거치도록 하는 것은 도움이 되지 않는다. 왜냐하면 처리량은 클럭 사이클에 의해 결정되고, 명령어당 파이프 단계의 수는 지연시간에 영향을 미치지 처리량에는 영향을 미치지 않기 때문이다.

- 명령어들이 더 적은 사이클이 걸리도록 노력한느 대신에 파이프라인을 길게 만들려고 해야한다. 이렇게 하면 명령어들은 더 많은 사이클이 걸리지만 사이클은 더 짧아진다. 이것이 성능을 높일 수 있다.

파이프라인 제어



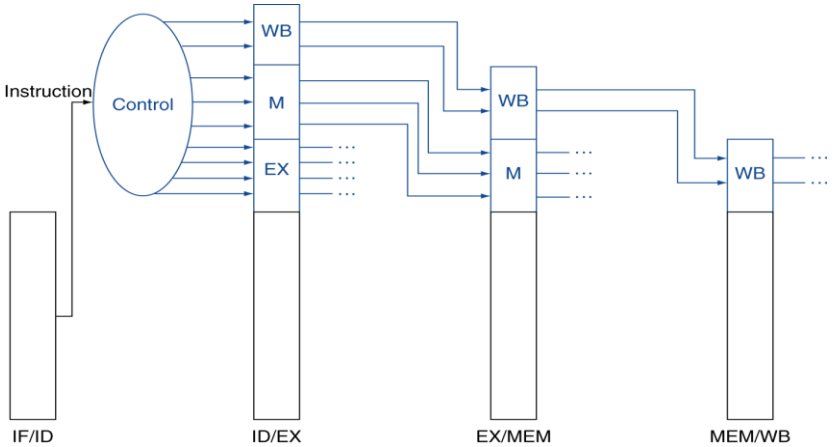
1. **IF**: 명령어 메모리를 읽고 PC 값을 쓰기 위한 제어신호들은 항상 인가되므로 이 파이프라인 단계에서는 제어할 것이 없다.

2. **ID**: 이전 단계에서와 마찬가지로 매 클럭 사이클마다 같은일이 일어나기 때문에 제어선X

3. **EX**: 설정할 신호들은 RegDst, ALUOp, ALUsrc이다. 이 신호들은 목적지 레지스터와 ALU 연산을 선택하고 Read data2와 부호확장된 수치 중 하나를 ALU의 입력으로 선택

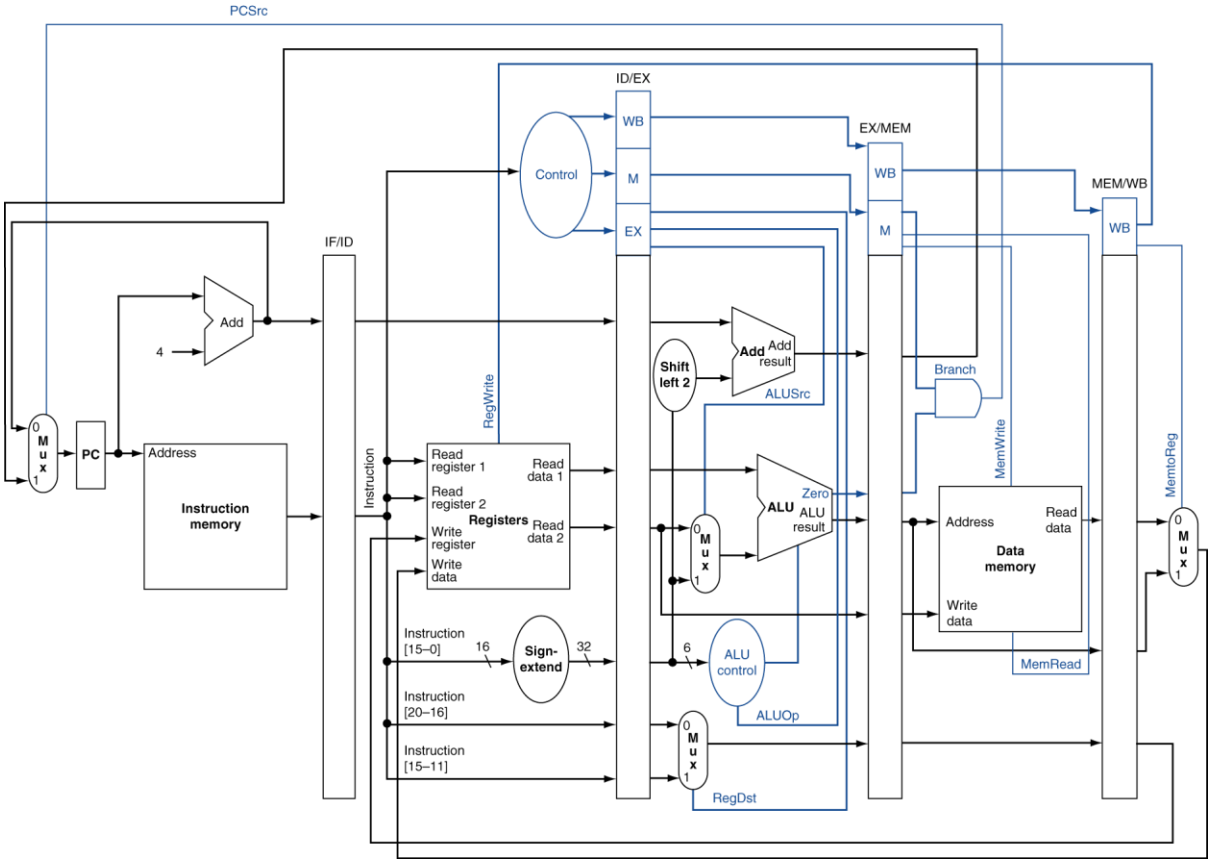
4. **MEM**: 제어선은 Branch, MemRead(lw), MemWrite(sw)이다.

5. **WB**: MemtoReg, RegWrite인데 MemtoReg는 레지스터 파일에 ALU결과를 보낼 것인가 메모리 값을 보낼 것인가 결정하고 RegWrite는 선택된 값을 레지스터에 쓰게 하는 것이다.



9개중에 4개가 EX 단계에서 사용(RegDst, ALUOp1, ALUOp0, ALUSrc), 3개가 MEM 단계에서 사용(Branch, MemRead(lw), MemWrite(sw)), 2개가 WB단계에서 사용(MemtoReg, RegWrite)

Pipelined Control



4.7 데이터 해저드: 전방전달 대 지연

Data Hazards in ALU Instructions

sub $2, $1,$3

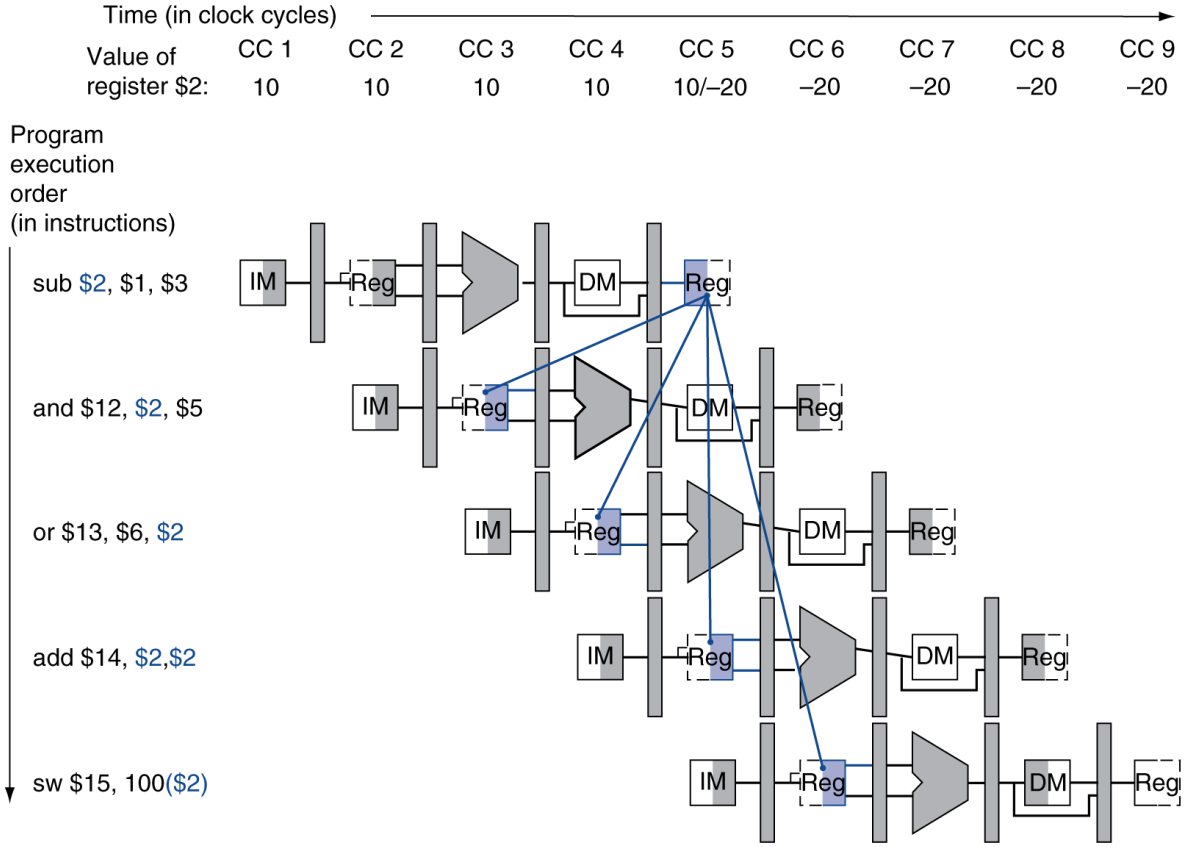
and $12,$2,$5

or $13,$6,$2

add $14,$2,$2

sw $15,100($2)

Dependencies & Forwarding



파란색이 종속성을 나타낸것이다.

같은 클럭사이클에서 읽기와 쓰기가 동시에 발생한다면 쓰여진값을 새로 읽을것이다. 저기서는 add와 sw만이 sub된 $2 값을 가지게된다. AND와 OR는 sub되기 이전 값인 $2을 가지게된다.

Detecting the Need to Forward

ID/EX.RegisterRs = 파이프라인 레지스터 ID/EX에 있는 한 레지스터의 번호, 즉 레지스터 파일의 첫번째 읽기 포트에 실린 레지스터 번호를 나타낸다.

ALU operand register numbers in EX stage are given by **ID/EX.RegisterRs, ID/EX.RegisterRt**

Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

and와 sub의 해저드는 and 명령어가 EX 단계에 있고 앞선 명령어(sub)가 MEM 단계에 있을 때 검출된다. 즉 1a의 조건이다.

1. 어떤 명령어들은 레지스터에 쓰기를 하지 않기 때문에 이 같은 방침은 정확하지 않다. 필요 없을 때에도 전방전달을 하는 경우가 있기 때문이다. 이럴때는 EX단계와 MEM 단계동안에 파이프라인 레지스터의 WB 제어 필드를 조사하면 RegWrite 신호가 인가되었는지 알 수 있다.

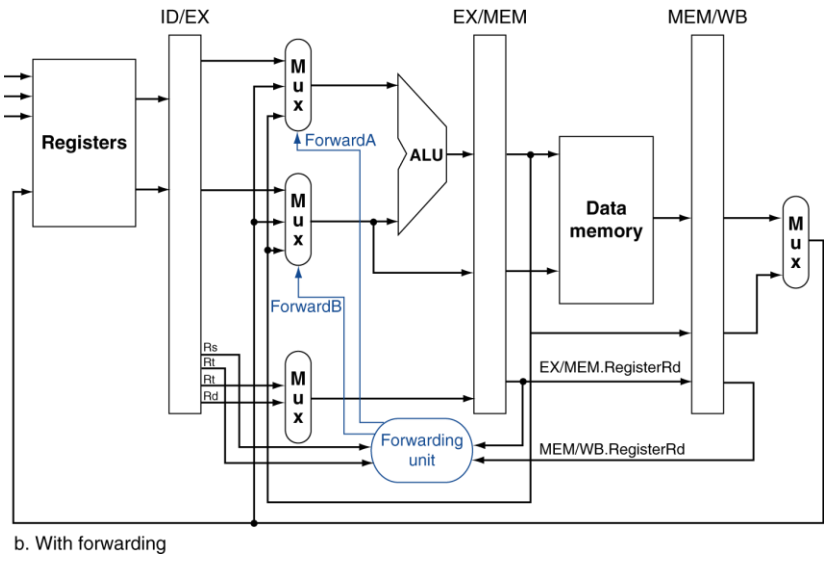
EX/MEM.RegWrite, MEM/WB.RegWrite

2. 파이프라인에 있는 명령어의 목적지가 $0이라ㄴ면 결과 값을 굳이 전방전달 할 필요가 없다.

EX/MEM.RegisterRd ≠ 0,
MEM/WB.RegisterRd ≠ 0

위의 조건들을 다 충족하면 해저드 검출이 제대로 작동할 것이다.

Forwarding Paths



b. With forwarding

EX hazard

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 10

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 10

MEM hazard

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
  ForwardA = 01

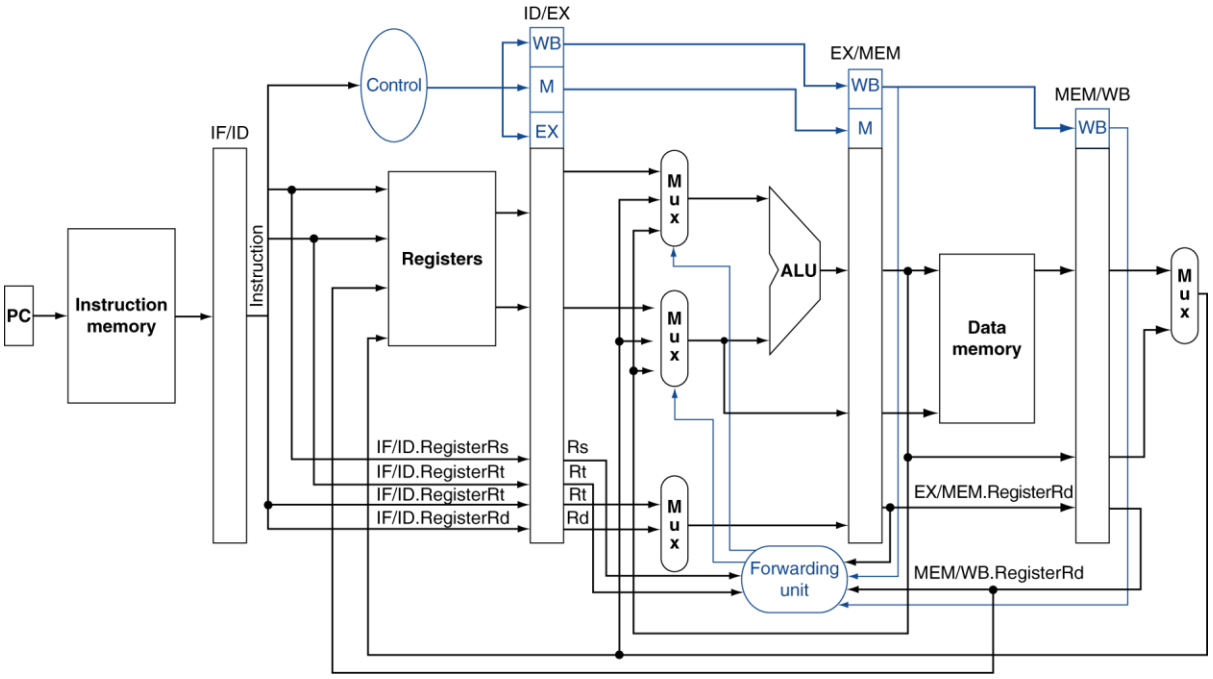
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
                and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
  ForwardB = 01

Double Data Hazard

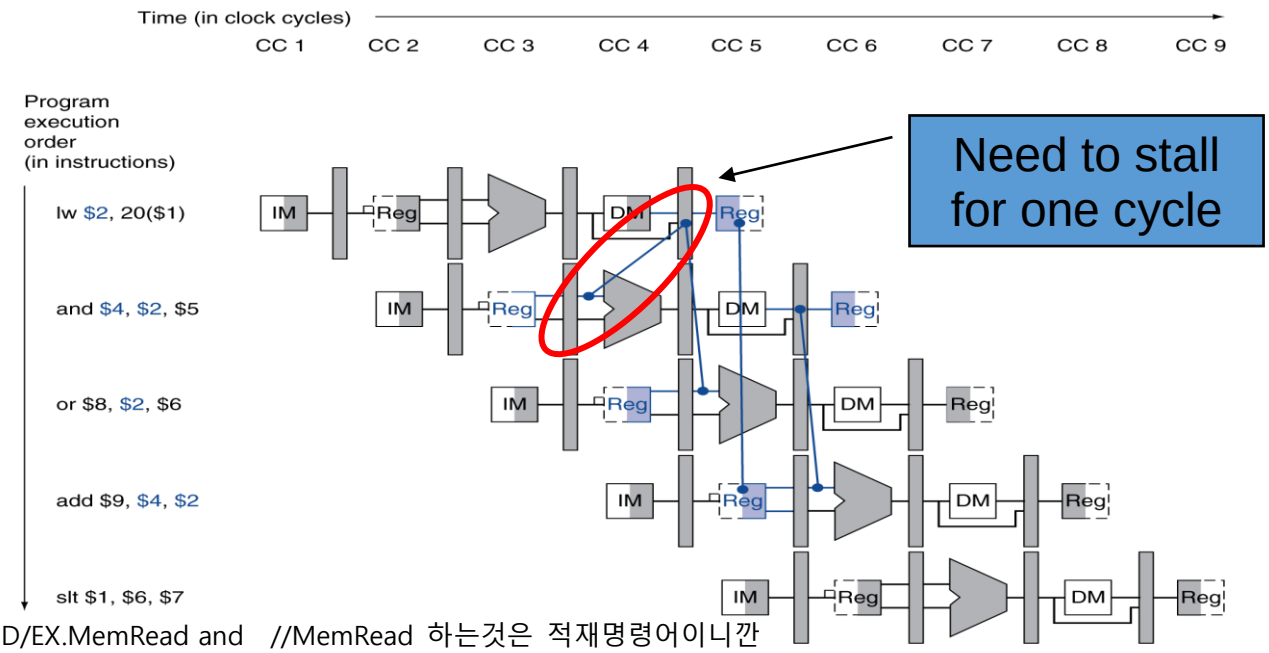
add $1,$1,$2
add **$**1,$1,$3
add $1,$1,$4

이 경우에는 헤저드가 두번 발생한다. 저럴경우에 대비하여 위에 MEM hazard에 파란색부분ㅇ르 추가하였다.

Datapath with Forwarding



Load-Use Data Hazard



ID/EX.MemRead and //MemRead 하는것은 적재명령어이니깐

((ID/EX.RegisterRt = IF/ID.RegisterRs) or

(ID/EX.RegisterRt = IF/ID.RegisterRt))

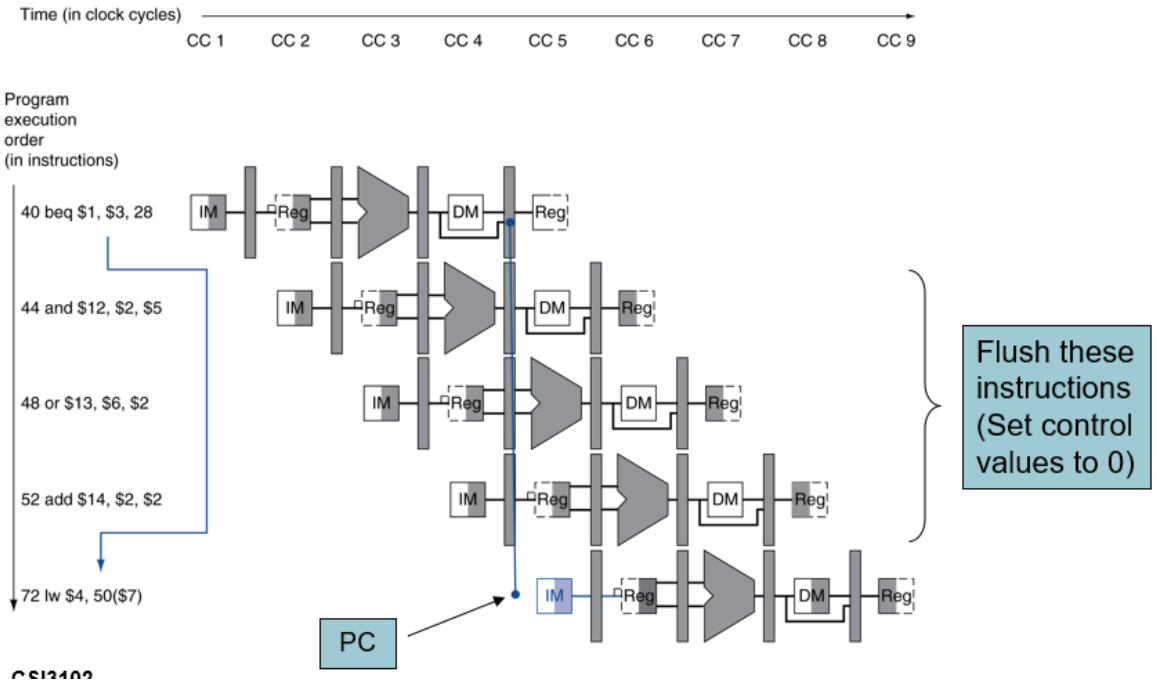
이것이 검출되면 한단계 더 지연한다. => 해저드 검출 유닛

How to Stall the Pipeline

Force control values in ID/EX register to 0 => EX, MEM, WB 단계의 9개의 제어신호를 모두 0으로 만드면 nop 명령어를 만들수 있다.

Branch Hazards

분기가 일어날지 검사는 MEM 단계에서 함. 그래서 아무조치가 없으면 세개의 명령어가 실행된다.



제어 해저드 해결하는 두가지 방법

1. 분기가 일어나지 않는다고 가정

=> 분기가 일어나지 않는다고 예측하고 명령어들을 순서대로 계속 실행한다. 만약 분기가 일어난다면 인출되고 해독되었던 명령어들은 버리고 분기 목적지에서 실행을 계속한다. 명령어를 버린 것은 적재-사용 데이터 해저드의 경우는 원래의 제어 값을 0으로 바꾼다. 적재-사용 지연의 경우 ID 단계의 제어 값만을 0으로 바구어 파이프라인을 통해 천천히 지나가도록함.

flush가 명령어를 버린다는 뜻

2. 분기에 따른 지연 줄이기

분기 성능을 향상시키는 한가지 방법은 분기가 일너라때의 비용을 줄이는 것이다.

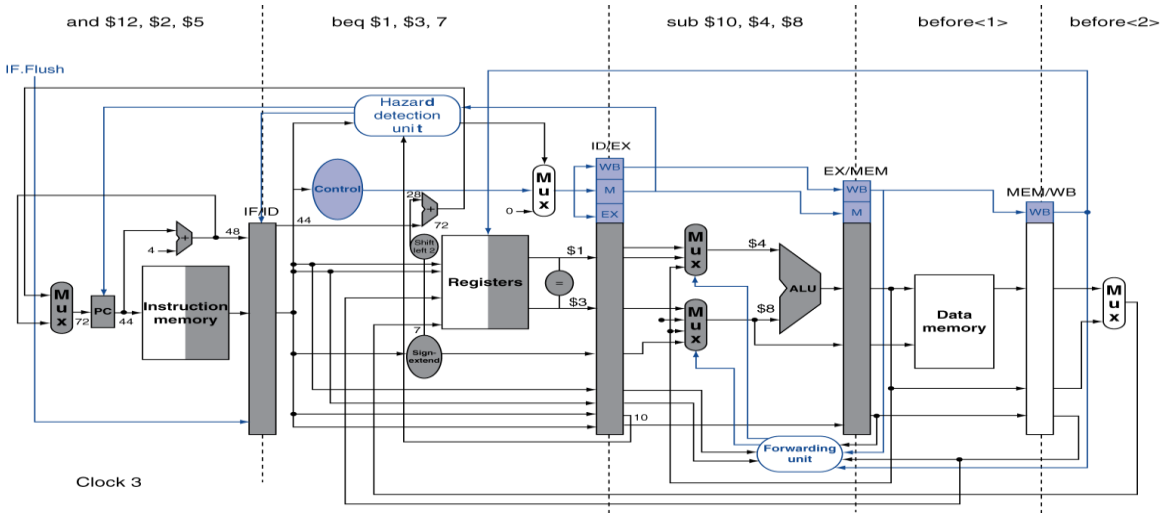
분기 명령어의 경우 다음 PC 값은 MEM 단꼐에서 선정된다고 가정하였다. 만약 이 분기 결정을 좀 더 앞당겨 할 수 있으면 더 적은 수의 명령어를 없애 버려도 된다.

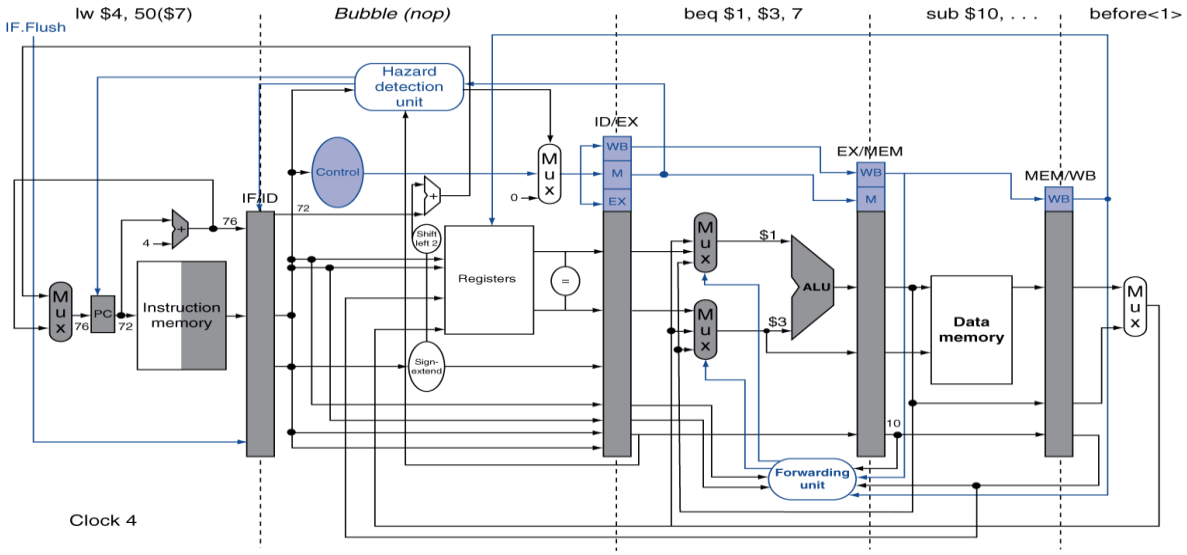
IF 단계의 명령어를 버리기 위해서 IF.Flush라는 제어선을 추가하는데 이 제어선은 IF/ID 파이프라인 레지스터의 명령어 필드를 0으로 만든다. 레지스터 필드를 0으로 만든 것은 nop으로 만든다는 것이다.

Example: branch taken

이 명령어 코드에서 분기가 일어날 때 무슨일 생기는지를 보여라. 분기가 일어나지 않는 것으로 파이프라인이 최적화되어 있고 분기 실행을 ID 단계로 옮겼다고 가정하라.

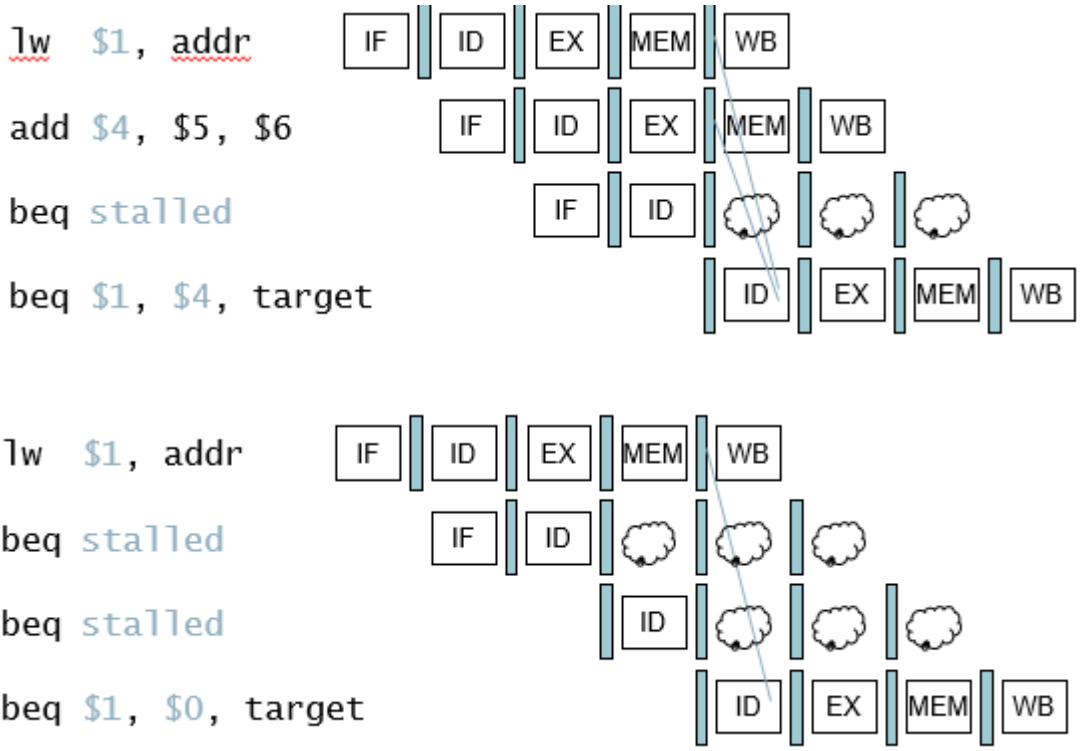
36: sub $10, $4, $8
40: beq $1, $3, 7 // PC branch 주소는 40+4+7*4=72
44: and $12, $2, $5
48: or $13, $2, $6
52: add $14, $4, $2
56: slt $15, $6, $7
...
72: lw $4, 50($7)





클럭사이클 3의 ID 단계는 분기가 일어나야한다고 판단한다.따라서 다음 PC 주소로 72가 선택디고 이미 인출된 명령어는 0으로 만든다.(nop(버블) 삽입)

Data Hazards for Branches

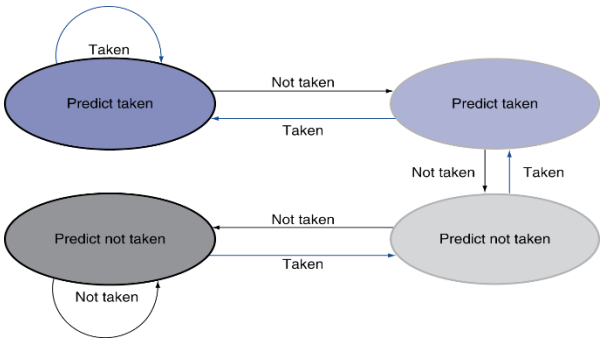


Dynamic Branch Prediction

프로그램 실행 중에 분기를 예측한다. 한가지 방법은 이 분기 명령어가 지난번에 실행되었을 때 분기가 일어났는지를 알아보기 위해 명령어 주소를 살펴보는 것이다. 만약 분기가 일어났다면 지난번과 같은 주소에서 새로운 명령어를 가져오도록 한다.

이 같은 기법을 구현하는 한가지 방법은 분기 예측 버퍼(Branch prediction buffer) 또는 분기 이력표(branch history table)을 이용한다. 분기 예측 버퍼는 분기 명령어의 하위 주소에 의해 인덱스되는 작은 메모리인데, 분기 명령어의 최근 분기 여부를 나타내는 하나 이상의 비트를 가지고 있다.

2-Bit Predictor



예측이 두번 잘못되었을 때 예측 값이 바뀐다.

분기가 일어나는 경우, 안 일어나는 경우 중 한쪽이 훨씬 많은 분기 명령어는 1비트 대신 2비트를 사용하면 예측이 틀리는 경우를 한번으로 줄 일 수 있다.

분기 예측 버퍼는 IF 단계에서 명려령어 주소로 접근한다. 분기가 일어난다고 예측되면 PC 값이 알려지자마자 목적지 주소로부터 명령어를 가져온다. 분기가 일어나지 않는다고 예측되면 순차주소에서 명령어를 가져오고 실행이 계속된다.

4.9 예외

Exceptions and Interrupts

"Unexpected" events requiring change in flow of control

Exception(예외) : 원인이 내부적인지 외부적인 구분하지 않고 제어흐름에서의 예기치 못한 변화를 지칭

인터럽트 : 사건이 외부적인 요인으로 일어날 경우.

Handling Exceptions

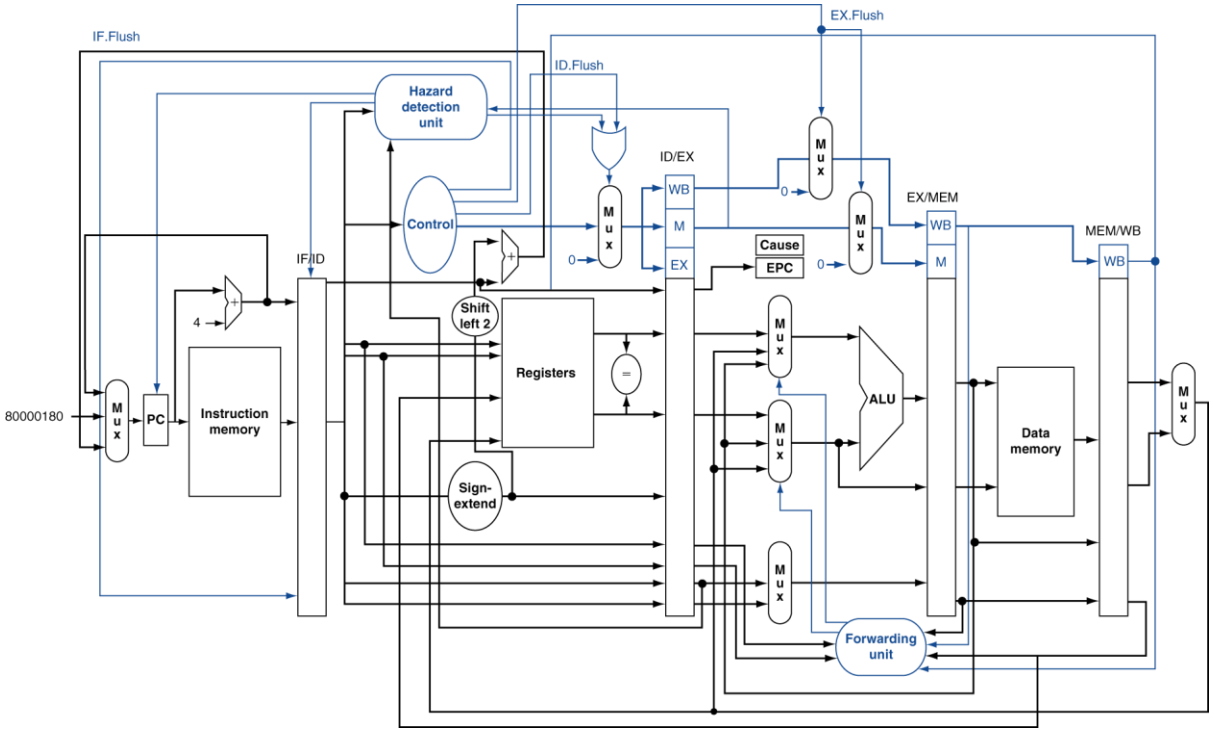
MIPS에서는 exception을 System Control Coprocessor (CP0)에서 관리함.

예외가 일어나면 문제가 일으킨 명령어의 주소를 EPC에 저장하고 어던 특정 주소에 있는 운영체제로 제어를 옮긴다. exception의 원인을 알아내는 방법은 두가지가 있다.

1. Cause register : 이 상태 레지스터는 예외의 원인을 나타내는 필드를 가지고 있다.

2. vector interrup : 제어가 옮겨져야 되는 주소가 예외의 원인에 의해 결정된다.

Pipeline with Exceptions



멀티플렉서의 새 입력 8000 0180(hex)는 예외가 발생했을 때 명령어를 인출하기 시작할 초기 주소이다.

예제 및 스스로 점검하기

p275, p279~282, p283, p287, p302, p309, p323, p325, p328, p332