

Chapter_04.The Processor

Introduction

- What you will learn in this chapter
 - 프로세서를 구현하는 것에 대한 원칙과 사용되는 기술을 배운다. 다음 두 가지를 통해 배우게 된다.
 - Simplified design
 - Pipelined design
 - 핵심 인스트럭션들의 집합의 구현
 - 메모리 참조 인스트럭션 : `ld`, `sd`
 - 논리 연산 인스트럭션 : `add`, `sub`, `and`, `or`
 - 분기 인스트럭션 : `beq`
 - 명령어 집합 구조가 구현의 여러 측면을 결정하는 방법
 - 컴퓨터의 clock rate 와 CPI 에 영향을 끼치는 다양한 구현 전략에서 어떻게 어떤 것이 효율적인지 알아내고 선택할 수 있을까?

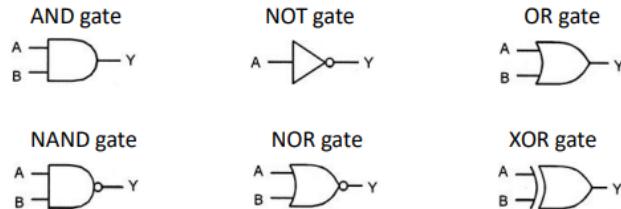
Overview of Implementation

- Instruction execution
 - 먼저 다음 두 가지 단계를 따른다.

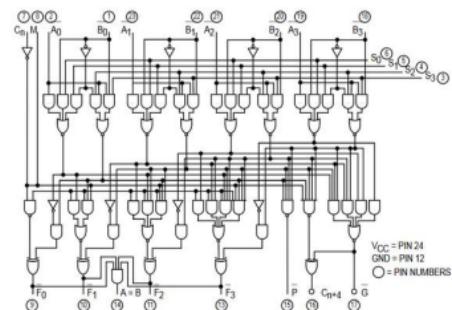
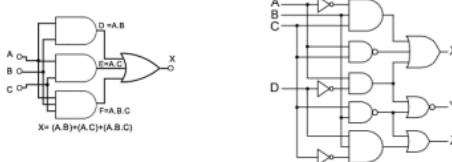
- PC에 있는 address를 이용해 수행할 instruction을 fetch한다.
- instruction에 지정된 register를 읽는다.
 - **ld** 는 register 1개, 대부분의 reg는 2개를 읽어들인다.
- 다음 단계는 instruction class(memory-reference, arithmetic-logical or branches)에 따라 다르다.
 - RISC-V의 단순성과 규칙성은 인스트럭션들 간의 유사성을 높이도록 만든다.
 - 예) Use of ALU
 - For arithmetic results
 - Memory address
 - Branch target address

Logic Design Basics

- Logic gates
 - Basic building blocks of electronic circuit



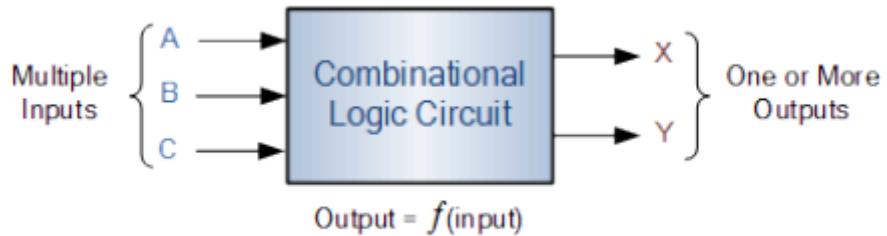
- Logic circuits
 - Circuits made of one or more logic gates that execute logical operations



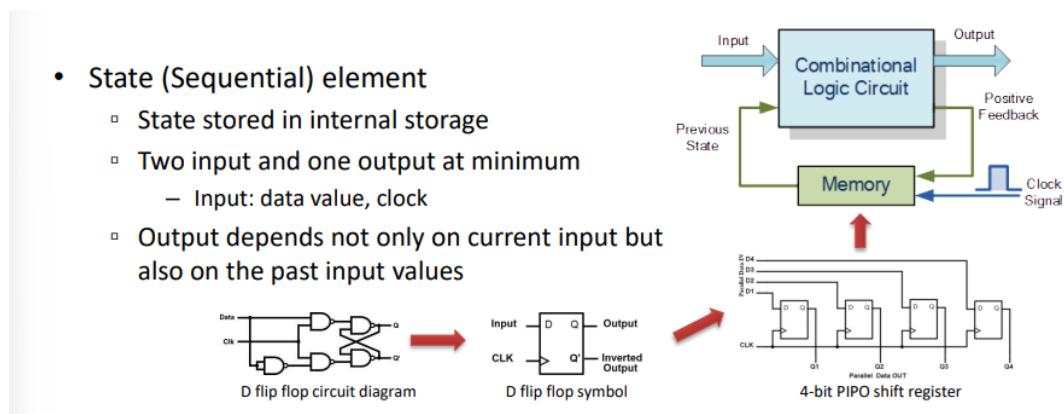
Logic Design Basics

- Digital Logic Element types 디지털 논리 요소의 타입
 - Combinational element
 - 데이터 Operate에 기반

- 내부 저장소가 없다
- 인풋의 함수가 곧 아웃풋이다. 만약, 동일한 인풋이 주어지면, produce는 같은 아웃풋을 낸다.



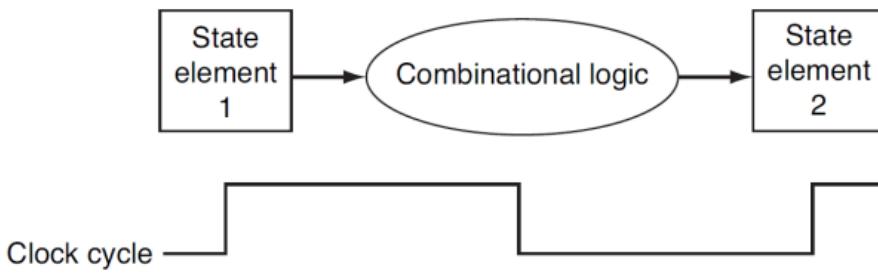
- State(Sequential) element
 - State는 내부 저장소에 보관한다.
 - 두 개의 인풋과 한 개의 아웃풋이 최소로 존재한다.
 - Input : data value, clock
 - 아웃풋은 현재 인풋에만 의지하지 않고, 과거의 인풋 값에 대해서도 영향을 받는다.



Logic Design Basis

- Clocks
 - 읽고 쓰는 것이 동시에 일어나는 경우, 예측할 수 없는 아웃풋을 야기할 수 있기 때문에, 읽기/쓰기의 타이밍을 제어하기 위한 신호 → 여러 개의 메모리를 동기화 시켜준다.
- Edge-triggered clocking
 - low → high OR high → low로 transition 되는 순간

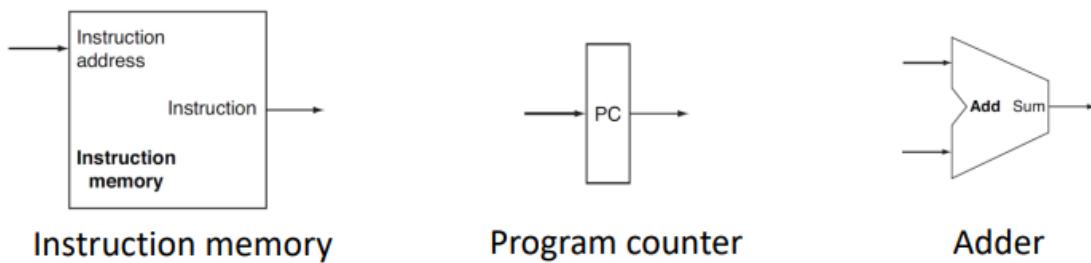
- sequential logic element 안의 data는 오직 clock edge에서 update된다.
- combinational logic은 state element(memory or register)에서 인풋을 가져와 state element로 data를 output



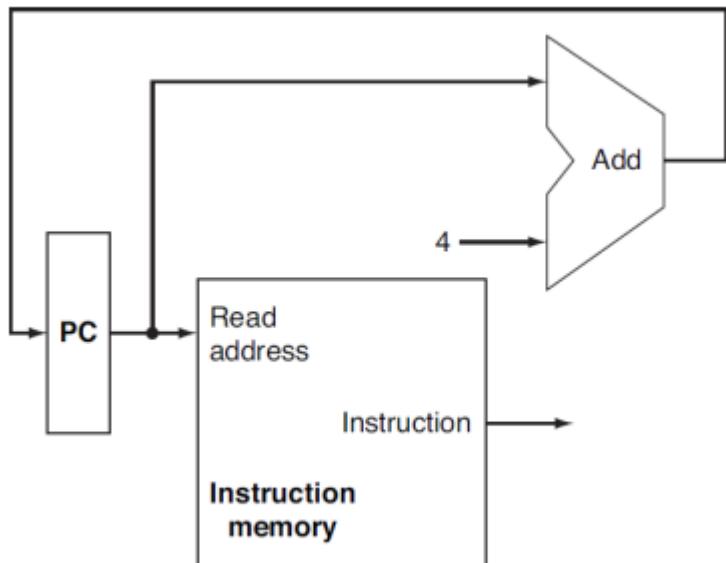
- Signal must propagate from SE1 to SE2 in 1 clock
- Optimal clock cycle length is determined by the combinational logic

Building Datapath

- Datapath Elements



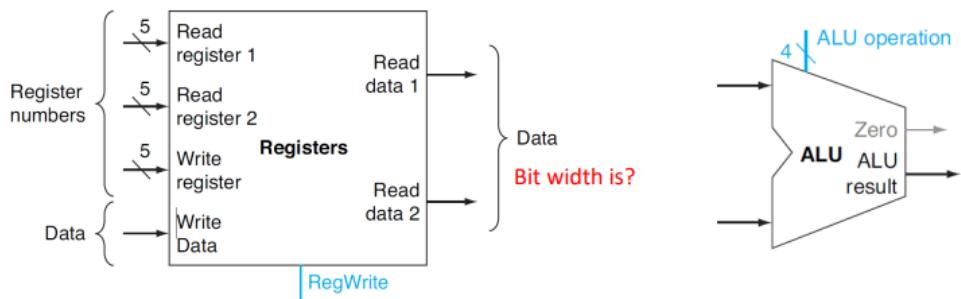
- Instruction memory
 - Input : instruction address
 - output: instruction
 - 실제 cpu에 존재하는 memory가 아닌, cpu 밖에 있는 memory의 interface를 의미한다.
 - memory에 실제로 접근하는 것은 매우 느린 operation. 100~200cycle 소모, → 너무 느리기 때문에, 1 cycle 만에 같은 일을 하는 기술을 사용한다.(cache memory)
- Fetching the instruction



- instruction 은 32bit, 즉, 4byte
- 그러므로 매 cycle마다 PC의 값이 4씩 증가 → 다음 instruction의 address
- 하지만, 해당 diagram은 branch 불가능

Decoding and Executing instructions

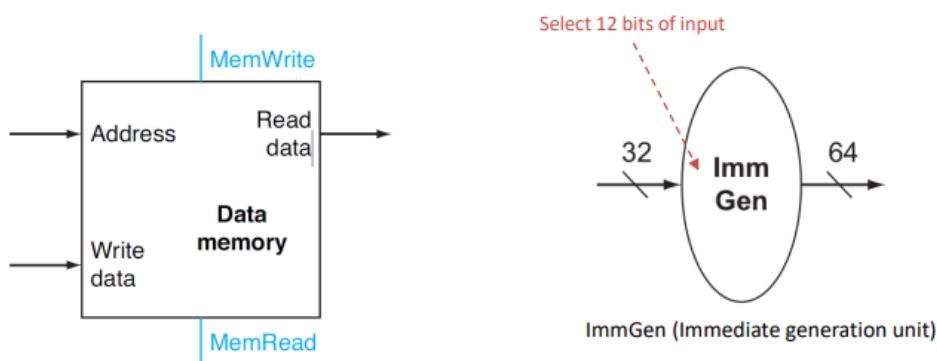
- R-type instruction
 - 두 개의 레지스터를 읽는다.
 - 산술/논리 연산을 수행한다.
 - 레지스터에 값을 쓴다.
- Register file
 - 레지스터 집합은 특정 레지스터 넘버에 접근 가능하다.
 - cpu 내부에 있는 register와의 interface
 - 6개의 port(4 input, 2 output)
 - Reading: 두 개의 레지스터 넘버와 두 개의 data output이 있다.
 - RegWrite : clock signal. 이것이 1인 경우 write 수행, 0인 경우 무시
 - Writing : 한 개의 레지스터 넘버 그리고 data가 써진다.
 - Write control 신호에 의해 제어된다 → **RegWrite** 가 무조건 asserted된다.



→ Bit width = 64bit, 아웃풋의 bit 크기는 64다.

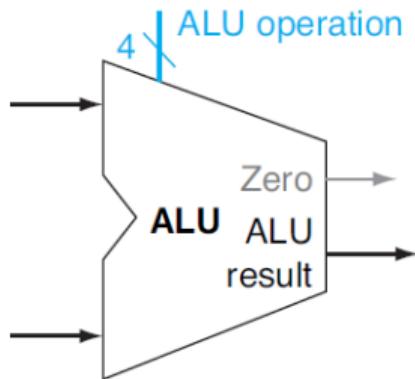
Decoding and Executing Instructions

- Load/Store instructions
 - Load/Store instructions **compute(Calculate) a memory address** by adding (Load/Store instruction은 처음에 memory address를 계산해야한다.)
 - `ld x1, offset(x2)`
 - `x2` 가 base address를 저장하고 있고, 거기에 offset을 더한 값이 access 할 memory address
 - `x2` 가 저장하고 있는 값은 64bit이고, offset은 12bit이기 때문에, sign extension 이 필요하다.
 - Id : 메모리로부터 읽어와서 레지스터 파일에 쓴다
 - sd: 레지스터 파일로부터 읽어와서, 메모리에 쓴다.
 - A unit to sign-extend 12-bit offset to 64-bit : 12비트 오프셋을 64비트로 부호를 확장하는 단위
 - sign-extend: 상위 부호 비트를 대상 데이터의 상위 비트에 복제하여 데이터의 사이즈를 키우는 것이다.

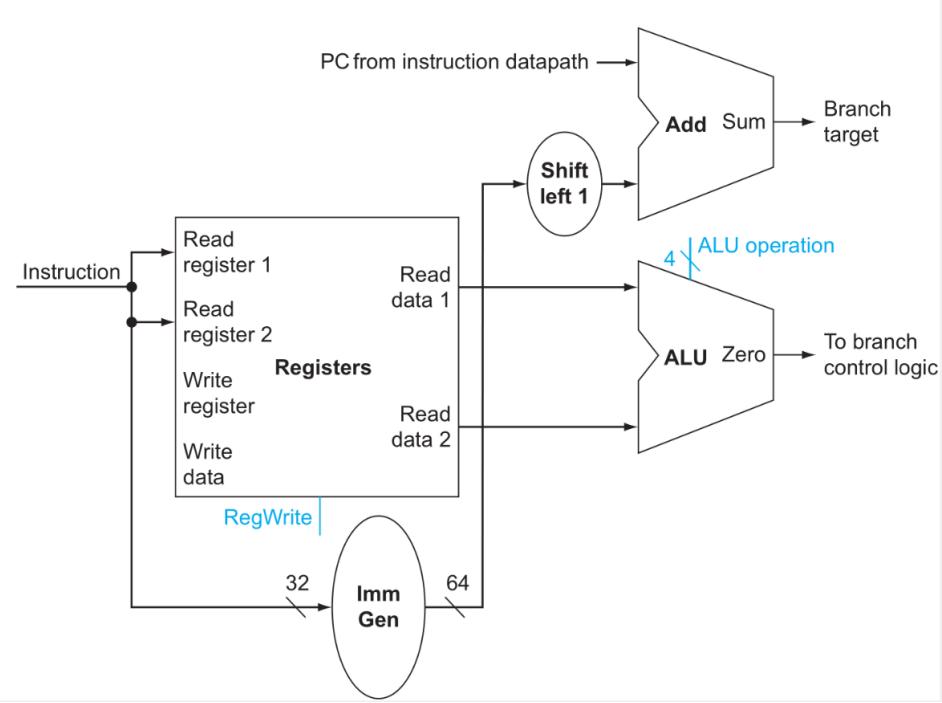


Branch Instruction Support

- Branch instructions
 - 두 개의 레지스터가 같은지에 대해 비교하는 것이다.
 - 12비트의 오프셋은 분기 타겟 주소의 계산을 위해 있다(상대적 분기 인스트럭션)
 - `beq x1, x2, offset` 을 예로 들어보자.
 - 먼저, 두 register의 값이 같은지 비교해야한다.
 - 같은지 비교하기 위해, $x1 - x2 == 0$ 인지 아닌지 판단한다.
 - $x1 - x2 == 0$ 인 경우, `Zero == 1`



- 분기 타겟 주소 계산을 위해
 - branch instruction의 offset 또한 12bit → ImmGen component 가 필요하다.
 - ImmGen의 output 을 shift left 1을 하여, PC에 저장된 address에 더한 것이 branch target address
 - **shift left 1을 하는 이유는, branch instruction의 immediate field에는 LSB의 0bit가 encode되어 있지 않기 때문**
- When branch is taken:
 - offset+PC 를 PC로 교체한다.
- When branch is not taken:
 - PC+4

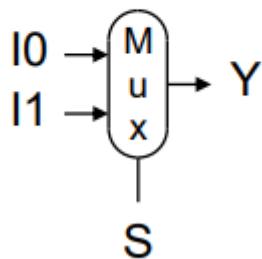


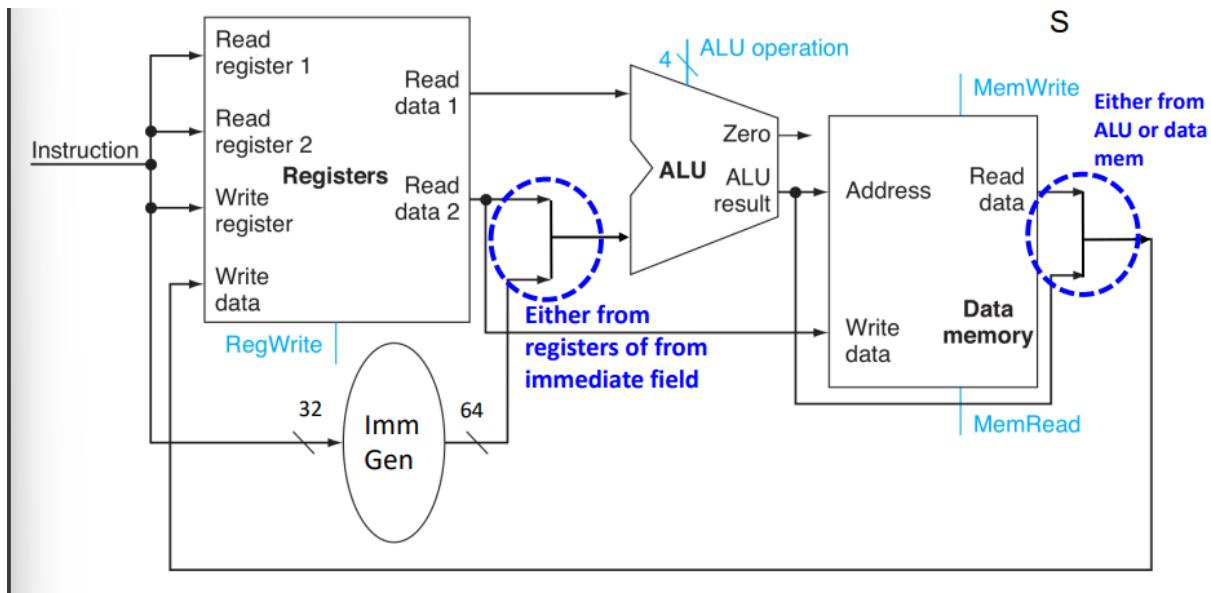
Creating a Single Datapath

- 데이터 라인은 함께 배선할 수 없다. → multiplexers 가 필요하다.
- Multiplexer : |0 와 |10| 들어오면, control signal에 따라 input 2개 중 하나만 선택

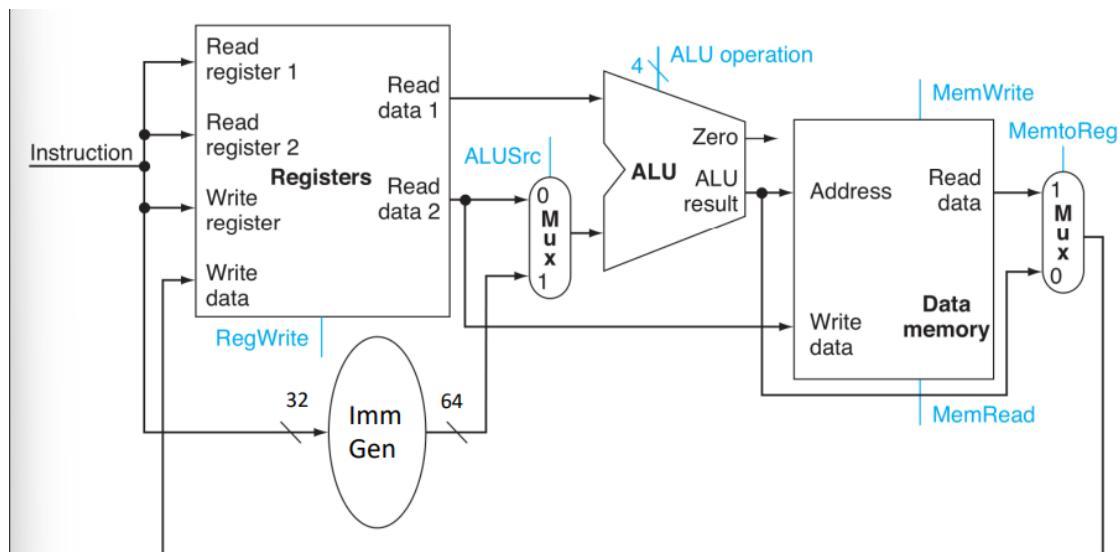
○ Multiplexer

$$\bullet Y = S ? I_1 : I_0$$



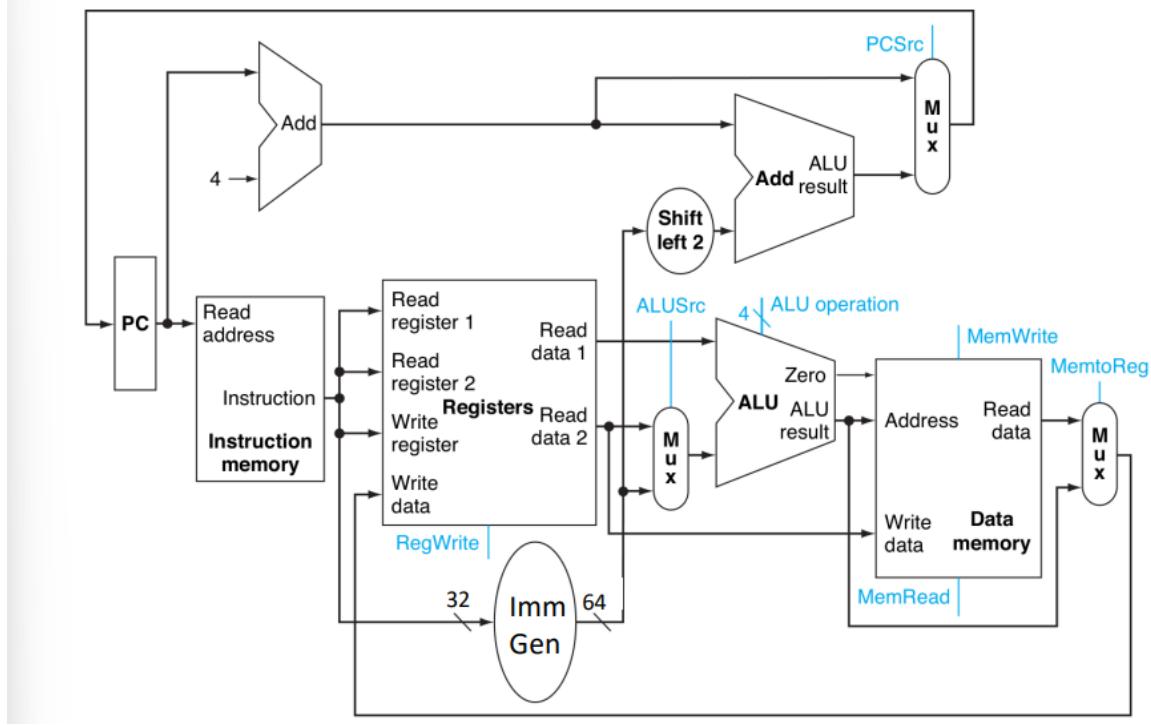


- The simplest datapath is to execute all instructions in one clock cycle : 가장 간단한 데이터 경로는 하나의 클럭 주기에서 모든 명령을 실행하는 것입니다.
 - No datapath resource can be used more than once per instruction : 한 명령당 (그러니깐, 명령 단위 당) 데이터 경로 리소스를 두 번 이상 사용할 수 없습니다.
 - Therefore, we need separate instruction and data memories : 그러므로, 인스트럭션을 분리하고, 데이터를 기억해야 한다.



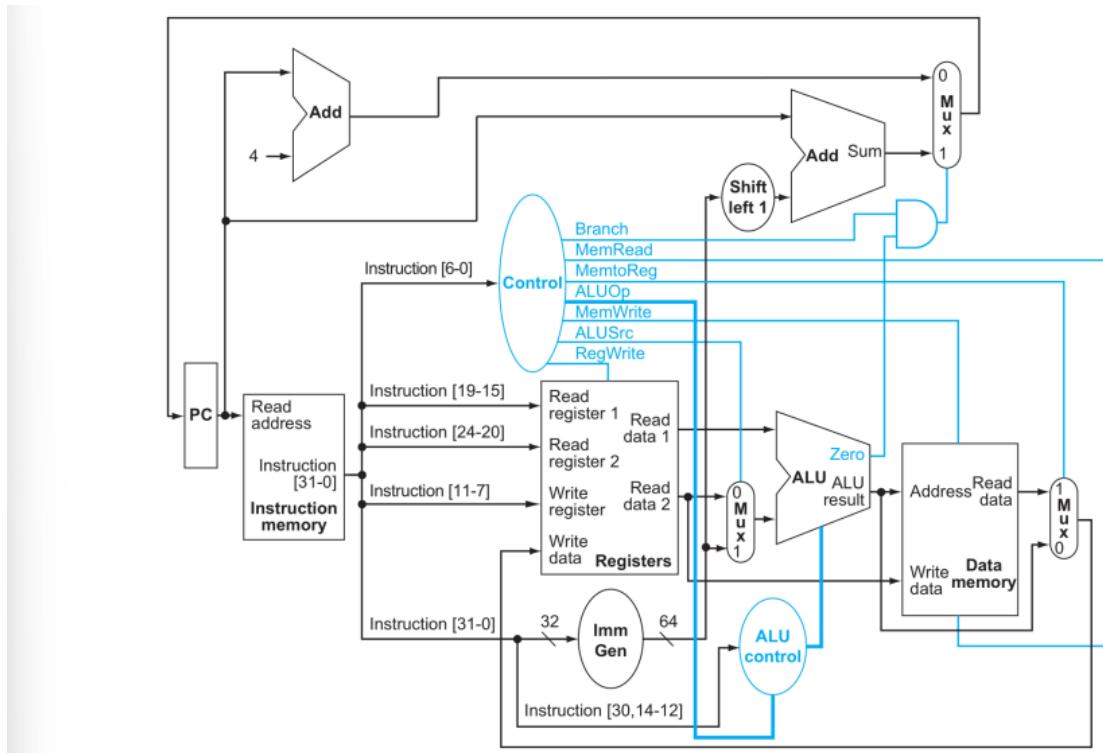
Complete DataPath (without control)

- Single clock cycle datapath for R-type, load/store, and branch instructions : R-type, load/store 및 분기 명령을 위한 단일 클럭 주기 데이터 경로



Complete DataPath(with control)

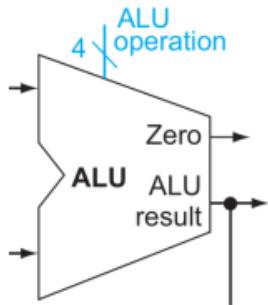
- R-type, load/store 및 분기 명령을 위한 단일 클럭 주기 데이터 경로



- control 과 ALU control 로 control unit 을 나눈 이유 → 더 simple한 design을 위해서
- 해당 datapath는 **ld**, **sd**, **add**, **sub**, **and**, **or**, **beq** 만 execute가 가능하다.

ALU Control

- ALU operations



ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

- ld**, **sd** : addition 으로 메모리 주소를 계산한다.
- R-type : AND, OR, add, sub 는 func7 과 func3의 값에 의해 정해진다.
- Branch : 두 개의 연산자를 빼서 결과값이 0 과 동일한지 본다.

ALU Control Unit

- ALU Control : 작은 control 단위는 ALU한테 control signal을 보낸다.
 - Inputs : funct3, funct7, 2bit control field(ALUOp)
 - Multiple levels of decoding : main control unit의 크기를 줄임으로써 latency의 성능을 향상시킨다.
 - Output: 4 bit ALU control code

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Truth Table for ALU Control

- 진리표는 funct 영역과 ALUOp 신호에 유용하다.
 - 진리표는 매우 방대하다. → 모든 인풋 combinations : $2^{12} = 4096$

ALUOp		Funct7 field										Funct3 field		Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]			
0	0	X	X	X	X	X	X	X	X	X	X	0010		
X	1	X	X	X	X	X	X	X	X	X	X	0110		
1	X	0	0	0	0	0	0	0	0	0	0	0010		
1	X	0	1	0	0	0	0	0	0	0	0	0110		
1	X	0	0	0	0	0	0	0	1	1	1	0000		
1	X	0	0	0	0	0	0	0	1	1	0	0001		

- 테이블을 간결하게 유지하기 위해 don't care 를 둑어주는 방식을 취할 수 있다.
 - don't care term : input column은 X로 나타낸다. 아웃풋은 인풋의 column에 대응하는 것에 대해 전혀 의존받지 않는다.
 - e.g.) ALUOp == 00 → ALU 집합은 funct 영역에 상관없이 0010으로 간다.
- 진리표가 구축되면, gates로 가는데 적합하게 되어진다.

Designing the Main Control Unit

- 우선, 첫 번째로 instruction의 형태를 관찰해 보아라.
- 단순한 design을 위해, field의 location을 최대한 fix

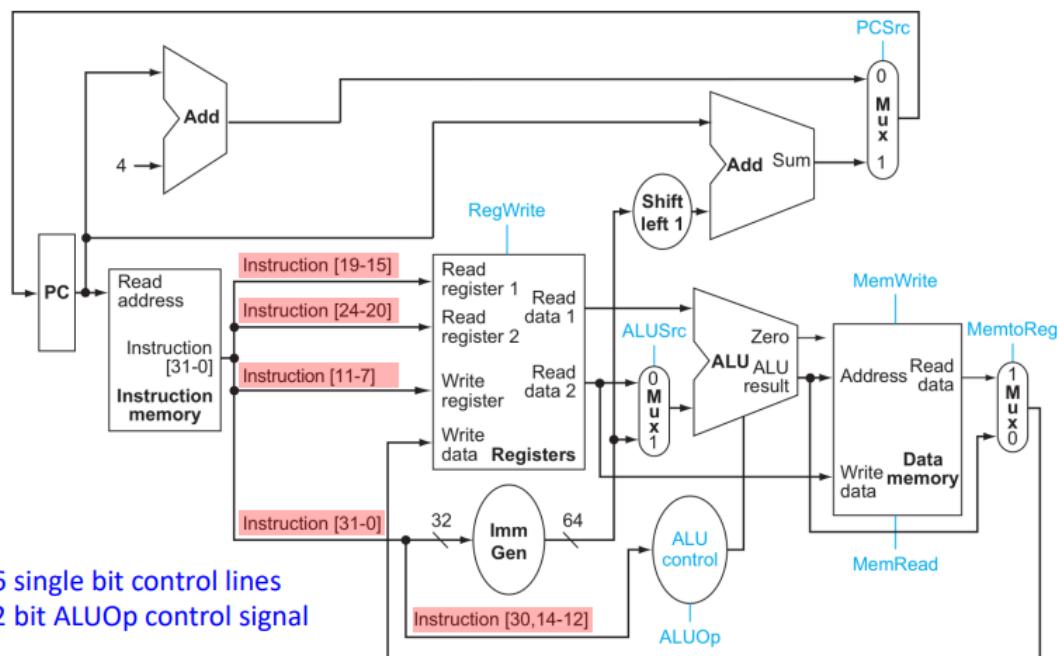
Name (Bit position)	Fields						51
	31:25	24:20	19:15	14:12	11:7	6:0	
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode	51
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode	3
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	35
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	99

- Opcode field (6:0)** → opcode에 달려있다.
- funct3 field (14:12) && funct7(31:25)** → 확장된 opcode 필드로 사용한다.
- rs1(19:15)** → R-type and branch
- rs2(24:20)** → R-type and branch

- rd(11:7) → R-type and load
- 또 다른 연산자는 12-bit offset가 있는데 이는 branch 혹은 load-store을 위해 존재 한다.

Datapath with Control Lines

- Datapath : 명령 레이블과 필요한 모든 멀티플렉서 및 제어 신호를 추가하여 데이터 경로를 업데이트한다.

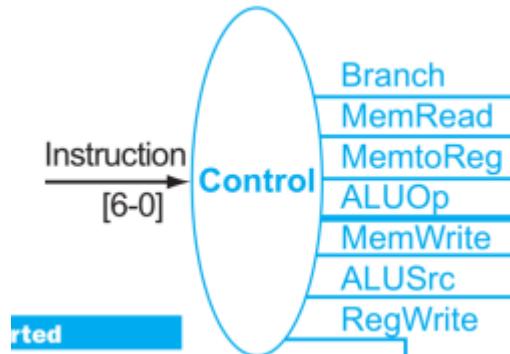


- 형관펜으로 친 부분 숫자 암기
- ALUOp 는 2bit, 나머지는 1bit → 총 8bit의 output
- 빨간색으로 색칠한 부분은 instruction을 decoding 하는 부분

Control Signals

- Control unit 은 instruction의 opcode와 funct fields 의 set based이다.
 - The PCSrc control line은 예외이다.
 - 명령어가 beq이고, ALU의 출력이 0이면 PCSrc가 asserted한다.
 - input : instruction 의 opcode → instruction 의 type에 대한 정보만 알 수 있다.

- 모든 signal은 instruction type에 따라 결정되는데, PCSrc는 예외
 - PCSrc는 instruction[6] beq 이고, ALU의 output이 zero인 경우에만 asserted

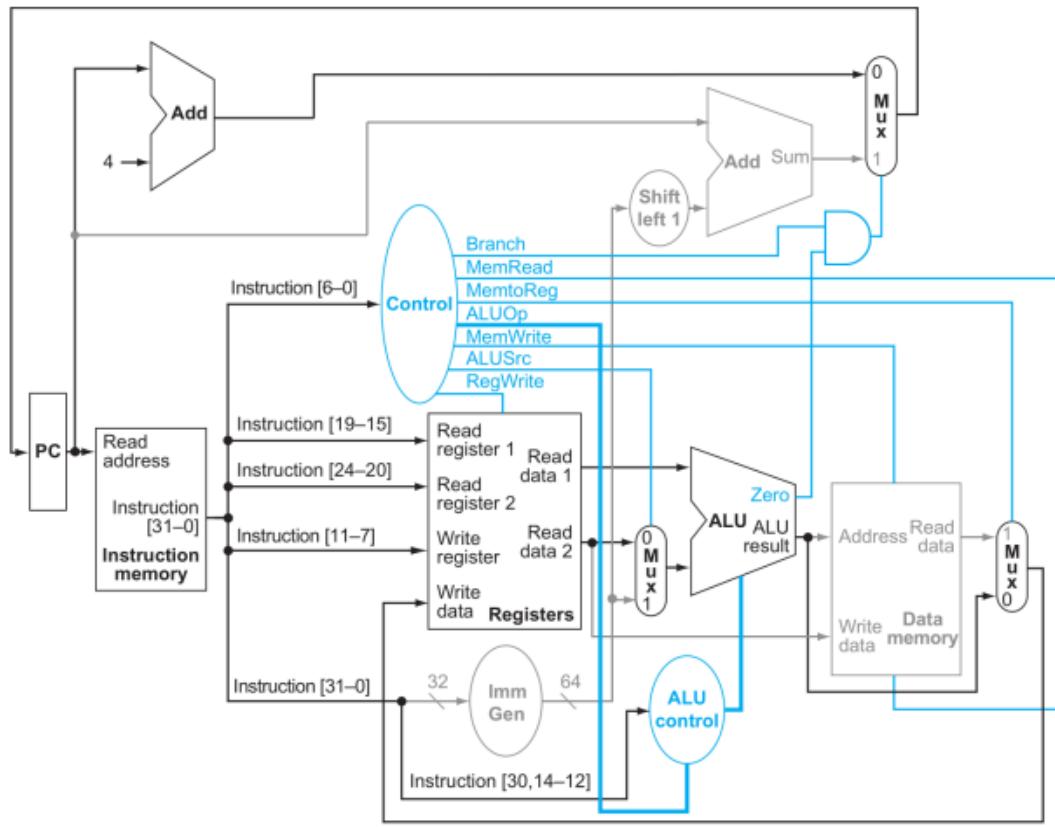


Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

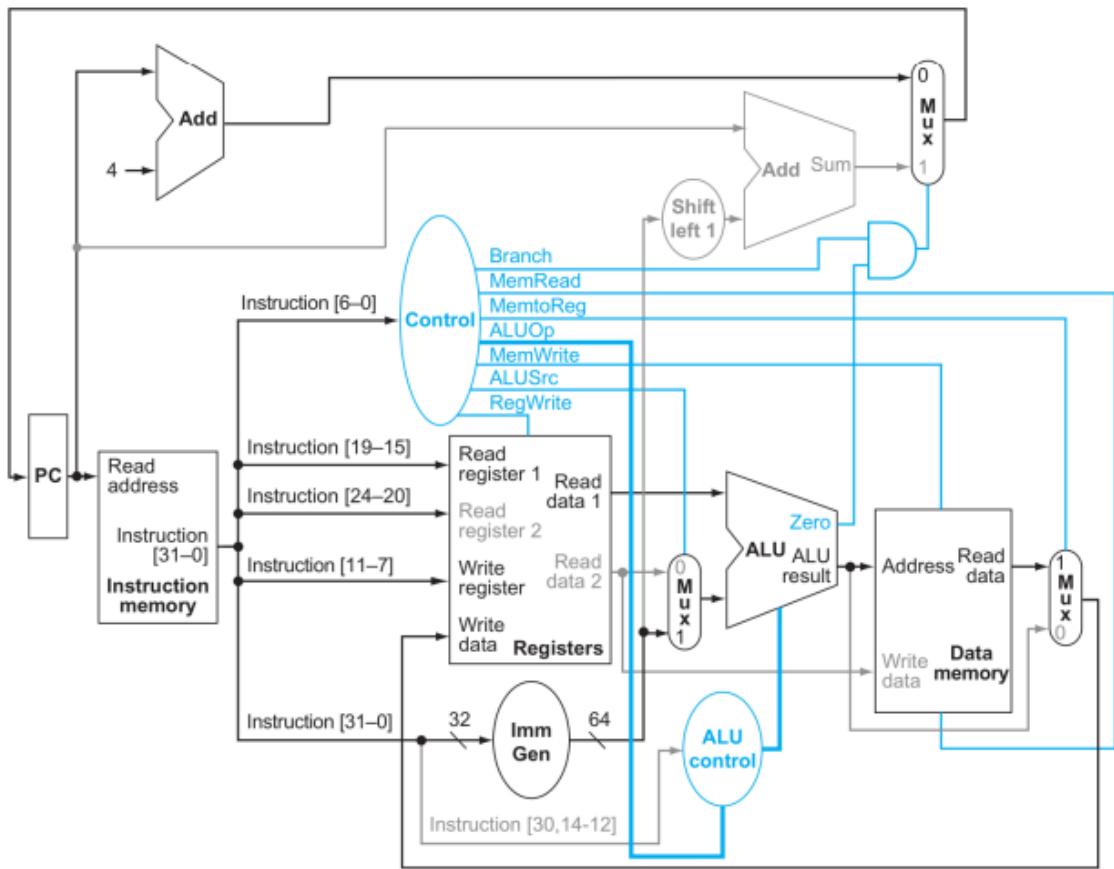
Datapath in operation for an R-type

- add x1, x2, x3



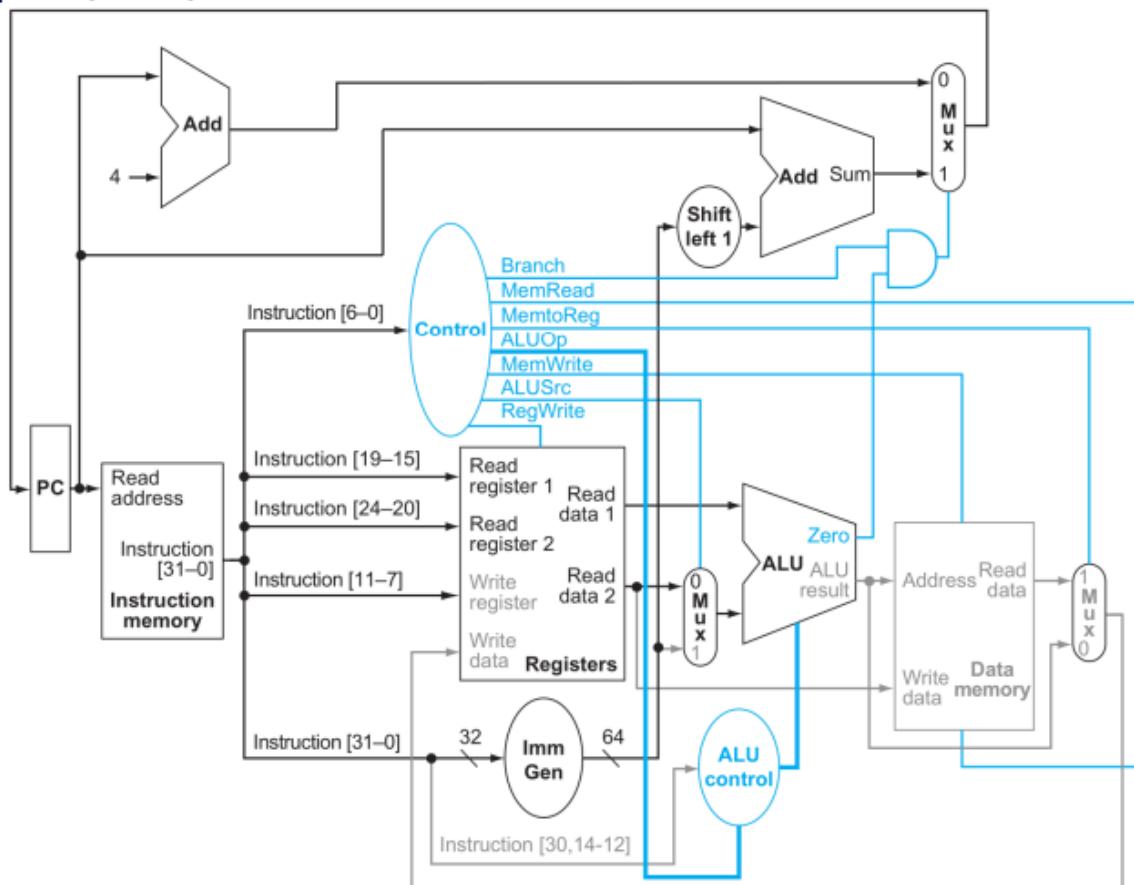
Datapath in operation for an Load

Id x1, offset(x2)



Datapath in operation for an Branch

beq x1, x2, offset



Why Single-cycle Implementation is not Used Today?

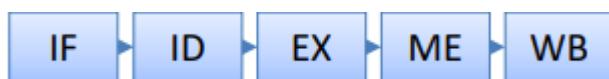
- **single-cycle implementation** : 모든 instruction이 1cycle만에 execute
- 현대적인 디자인에 사용하기에는 너무 비효율적이다.
 - 프로세서 중 가장 긴 path는 clock cycle을 결정한다.
 - instruction을 load해 오는 것은 다음 일련의 다섯가지의 units 을 따른다.
 - Instruction memory → register file → ALU → data memory → register file
 - add나 sub 등은 ld에 비해 더 빨리 수행될 수 있지만, clock cycle을 ld에 맞춰야 하기 때문에 비효율적이다.
- 그러면, instruction 마다 clock period를 다르게 하는 것은? 불가능. 설계 원칙 making the common case fast를 위배하기 때문이다.
- 그래서 나온 것이 **pipeline**

RISC-V Pipeline

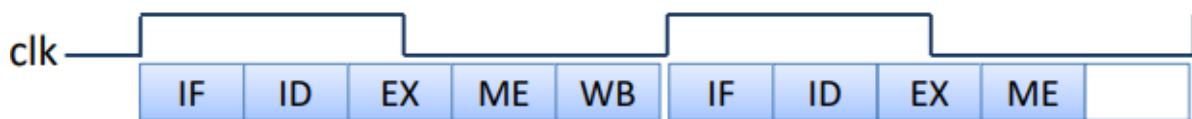
- RISC-V instructions classically take five steps
1. IF → 메모리로부터 값을 instruction으로 가져온다.
 2. ID → register을 읽고, instruction을 decode한다.
 3. EX → 연산을 계산하거나, address를 계산한다.
 4. MEM → data memory에 있는 operand에 access(필요한 경우)
 5. WB → register에 값을 적는다.

Pipelining

- Instruction Execution Steps

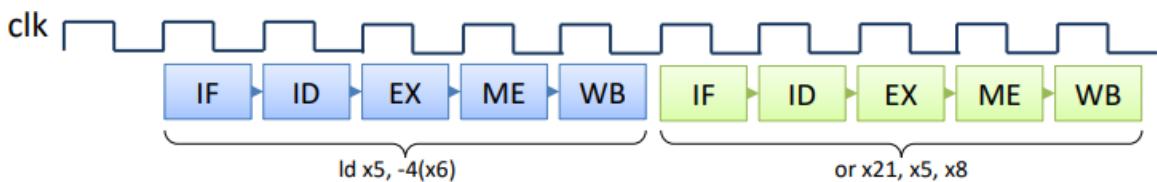


- Single cycle execution

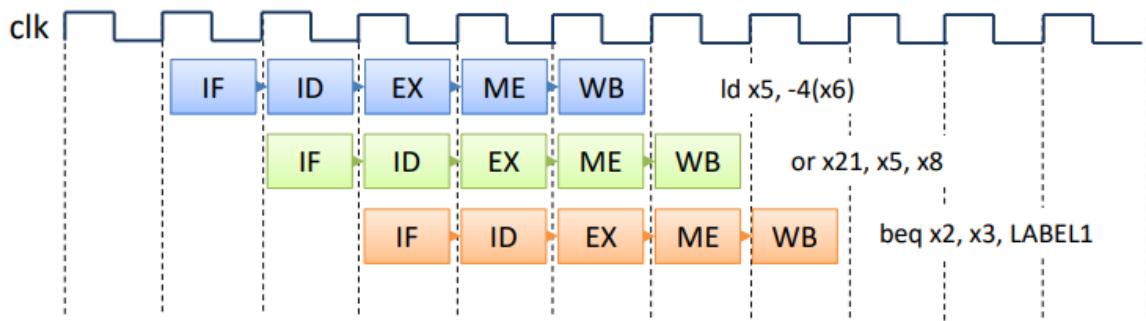


- Multi cycle execution

- Each step uses only part of the CPU H/W (components_



- Pipelined execution



- Pipeline Performance

- throughput : 단위 시간 당 처리 할 수 있는 일의 개수 = #of pipeline stages(pipeline stages의 개수)
- Time-between-instructions_pipelined = Time-between-instructions_nonpipelined/#of pipeline stages

- Performance comparison

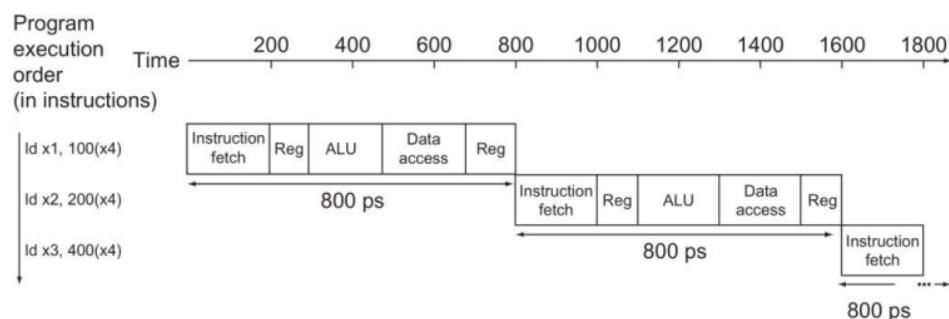
- Multiplexors, control unit, PC access and sign extension의 overhead가 없다고 가정한다.

Instruction Class	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
Id	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sd	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
Branch	200 ps	100 ps	200 ps			500 ps

- 이론적으로, 3개의 Id instruction의 상위 영역의 수행은

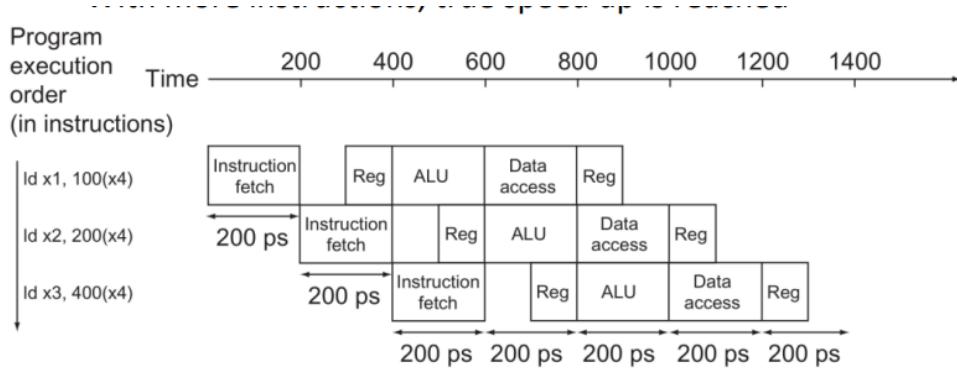
- Single-cycle, nonpipelined execution

- $800\text{ps} \times 3 = 2400\text{ps} = 2.4\text{ns}$



- Pipelined execution

- 총 pipelined time = $(800\text{ps}+200\text{ps})+200\text{ps}+200\text{ps} = 1400\text{ps}$
- 더 많은 instruction이지만, 도달 시간은 더 짧아짐



Pipelining increases the instruction throughput(단위 시간당 처리 할 수 있는 일의 갯수 즉, 인스트럭션은 파이프라인에서 증가한다.)

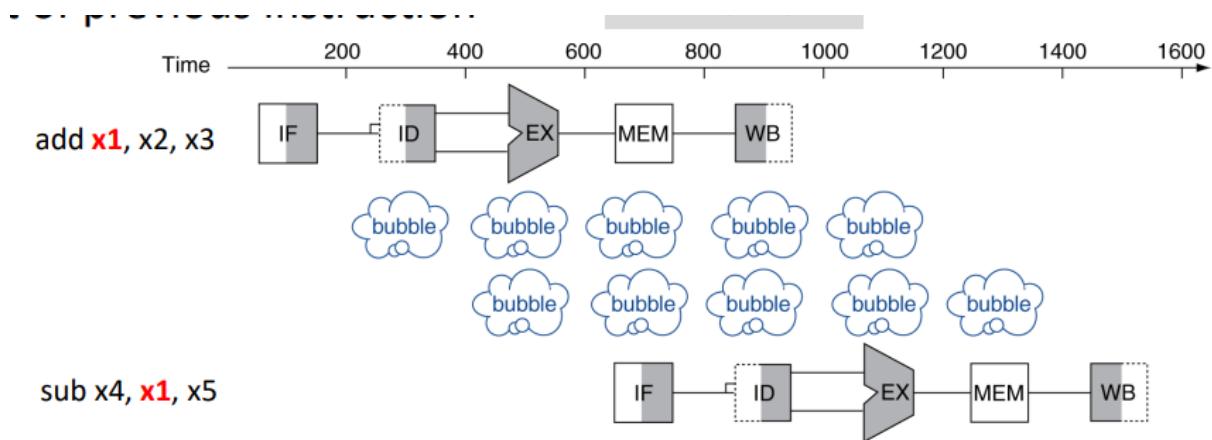
Pipeline Hazards

- Hazards
 - 파이프라인에서, 다음과 같은 상황들이 발생하는데, 바로, 다음 instruction이 다음 cycle에서 실행 되지 못 하는 문제이다.
- Type of hazard
 - Structural Hazards
 - H/W 은 특정 명령 조합을 지원 할 수 없다.
 - resource conflict
 - ex) 오직 한 개의 memory unit이면 어떨까?
 - Data Hazards
 - 이 Hazards는 다음으로 수행 할 instruction이 필요한데, 그 전 cycle에서 아직 준비가 안 된 상태이다.
 - Stall(=wait) inst이 준비 될 때 까지 기다리는 것이다.
 - Control Hazards = branch hazard
 - 앞의 instruction의 결과로 결정을 내려야 할 때 발생하는 Hazard

Data hazards

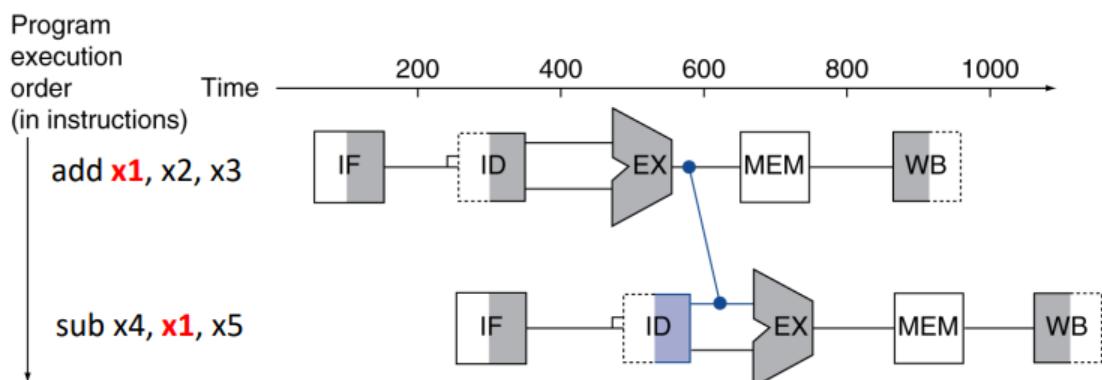
- Instruction이 앞에 inst의 결과를 필요로 하는 경우이다.

```
add x10, x12, x13  
sub x15, x10, x11  
//이와 같은 관계를 data dependency라고 한다.
```



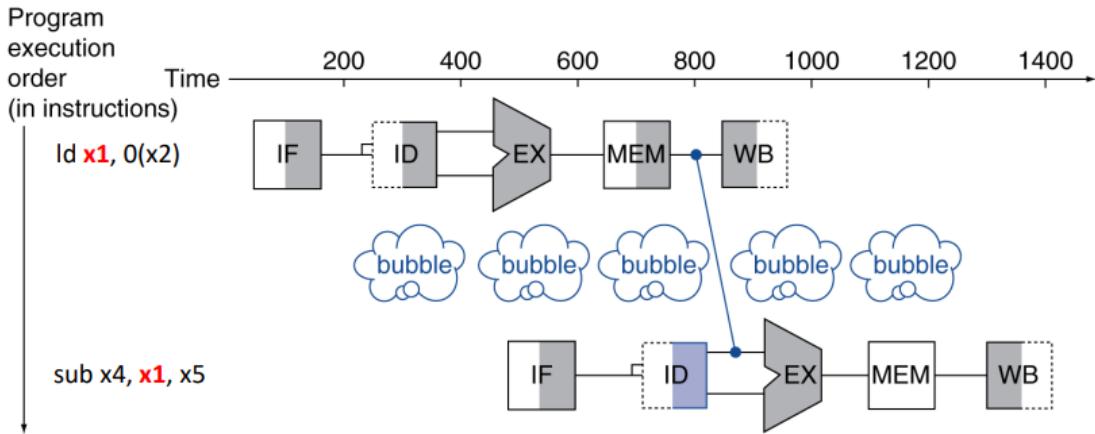
- Forwarding

- 그럴 땐 이 forwarding이라는 것을 사용하는데, 이건, 한 단계마다 인스트럭션이 순차적으로 실행 될 때, 필요한 인스트럭션이 있다면, ex단계에서 먼저 끝내고 바로 다음 인스트럭션을 보내주는 것이다.



Data Hazards(2/3)

- load-use data hazard
 - 포워딩을 한다고 해도, stall(bubble)를 피할 수 없는 경우가 있다.



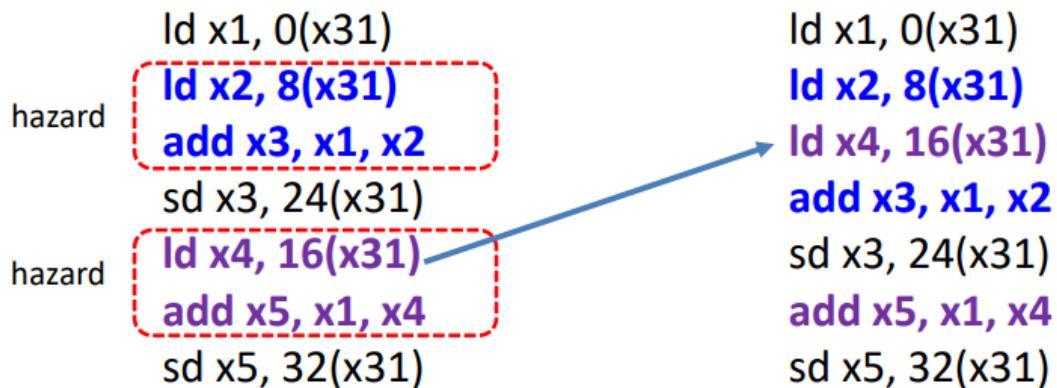
그럼 우린 어떤 것을 할 수 있을까? code reordering

Data Hazards(3/3)

- Code Reordering

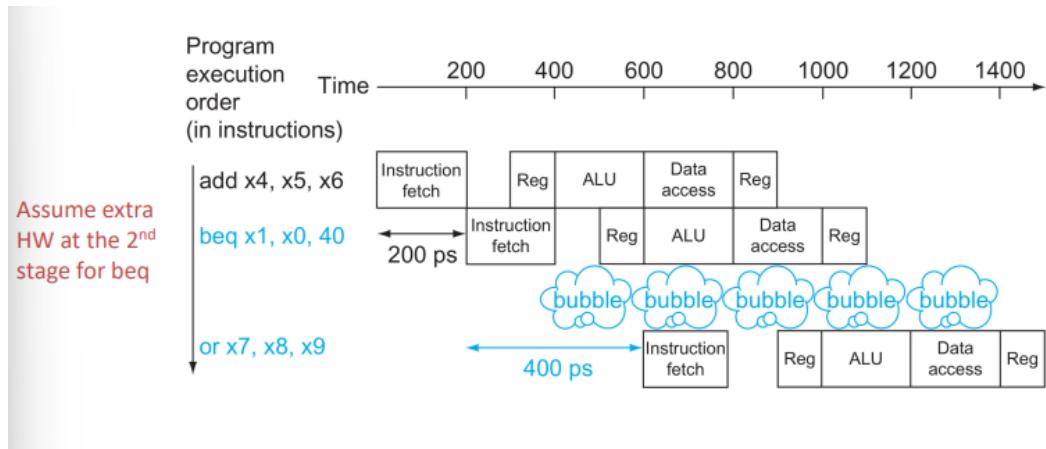
```
a = b+e;
c = b+f;
```

이 경우는 b가 겹쳐져있다. 이럴 때, bubble가 생기게 되는데, 어차피 피할 수 없는 버블이라면, 차라리 그 자리에 지장이 가지 않는 instruction을 넣어서 미리 계산이 되도록 한 다음, 다음에 원래 필요했던 instruction에 대한 결과값을 보내줘서 낭비되는 cycle 없이 실행이 잘 되도록 하는 것이다.



Control Hazards

- 이 Hazard는 현재 명령이 도달 할 때의 EX stage가 오기 전까진 instruction의 값을 알 수 있는 것이다.
 - branch instruction
 - Unable to fetch the correct instruction(올바른 instruction의 값을 얻을 수 없다.)

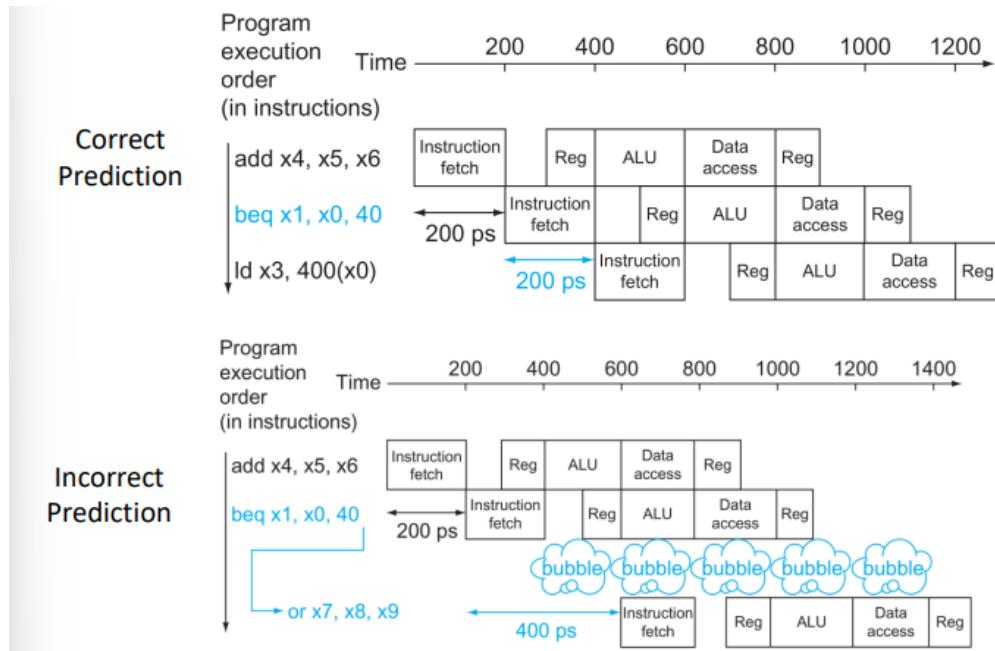


Solutions

1. stall
2. extra H/W
3. prediction

Control Hazards

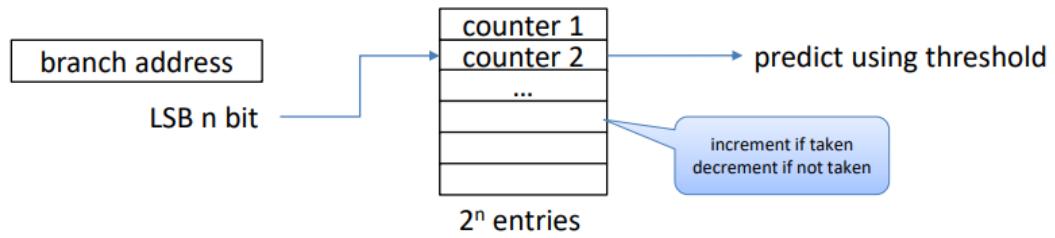
- Branch Prediction(branch 예측)
 - stall 을 사용하는 경우 : prediction이 잘못된 경우
 - Simple Prediction ↔ Dynamic Prediction
 - Predict : branch가 점프를 할 것인가 안 할 것인가



prediction이 오라로 된 경우, bubble없이 연산이 연속으로 잘 시행되지만, prediction에 error가 난 경우, bubble가 들어가고, 그 후에, 타겟 주소 계산이 시작된다.

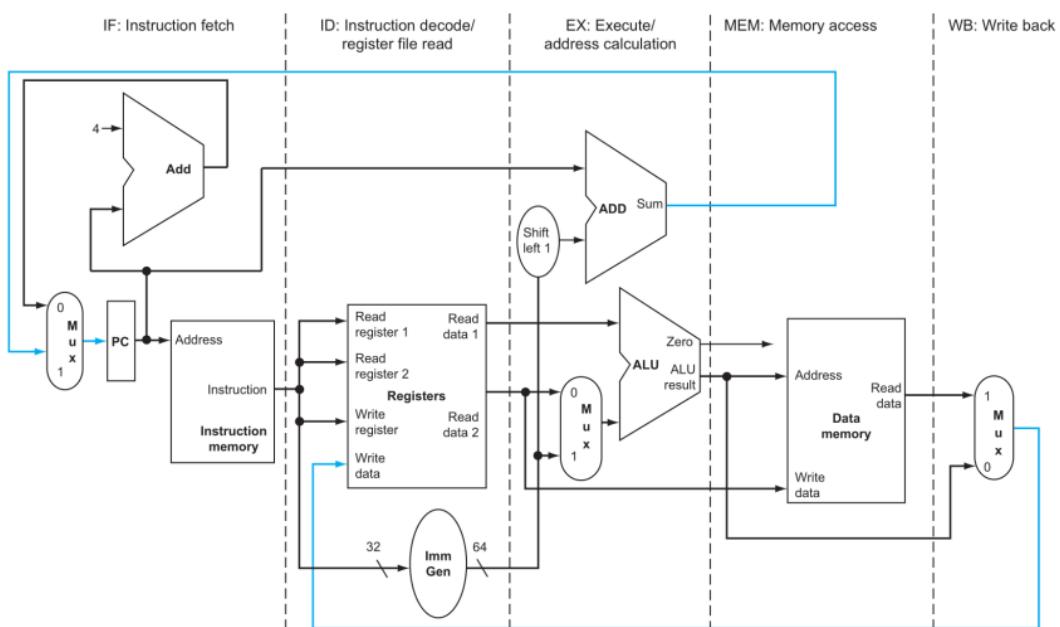
Control Hazards

- Branch instruction : 평균적으로 17%정도 등장한다.
- Static branch prediction
 - 이전에 실행했던 결과를 수집한 정보를 사용
 - ex) loop의 맨 아래 : 분기가 취해질 가능성이 더 높다.
 - It relies only on the typical behavior (전형적인 행동에만 의존함.)
 - 분기 명령을 구분하지 않는다.
- Dynamic branch prediction
 - 실행중에 정보를 모아두고 이 정보를 사용
 - 최근 branch 결정을 기록에 저장해 놓음

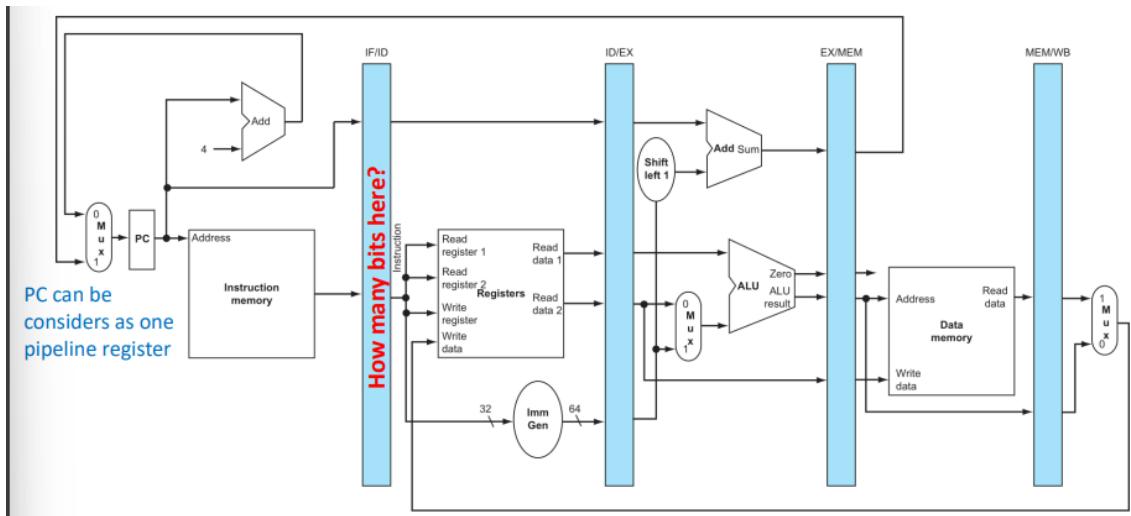


Pipelined Datapath

- RISC-V는 pipeline에서 다섯 단계를 가진다.(IF, ID, EX, MEM, WB)
 - 단일 clock cycle 동안 최대 5개의 명령이 실행

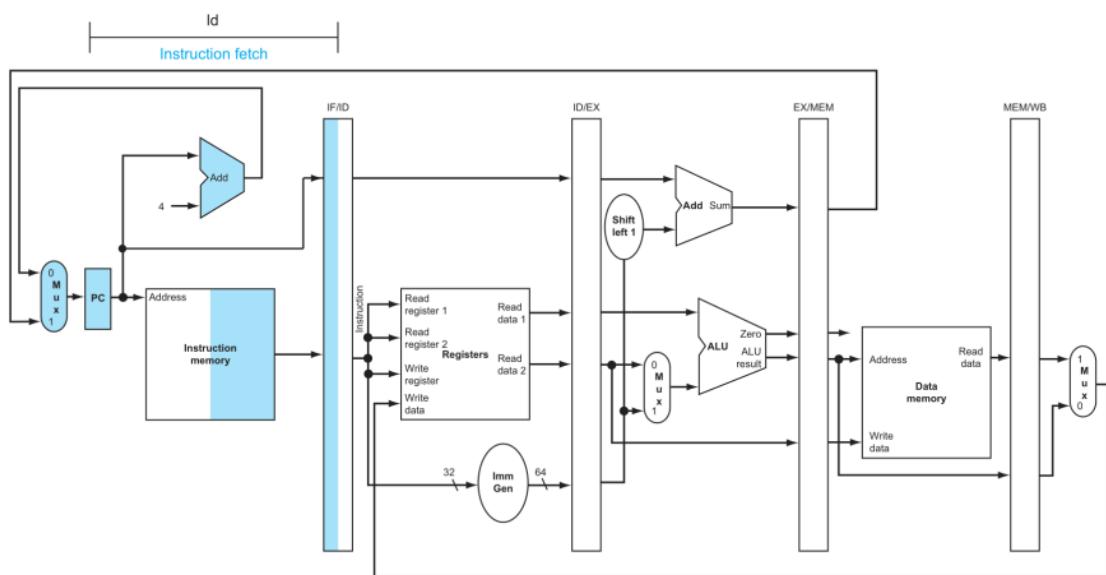


- Pipeline registers
 - 파이프라인은 이전 단계의 cycle에서 얻은 정보를 stages들 사이에 있는 reg에 저장할 필요가 있음.
 - Pipeline reg은 해당 단계의 모든 데이터를 저장 할 수 있을 만큼 충분히 넓어야 함.
 - IF/ID, ID/EX, EX/MEM, MEM/WB



IF Stage for Load Instruction

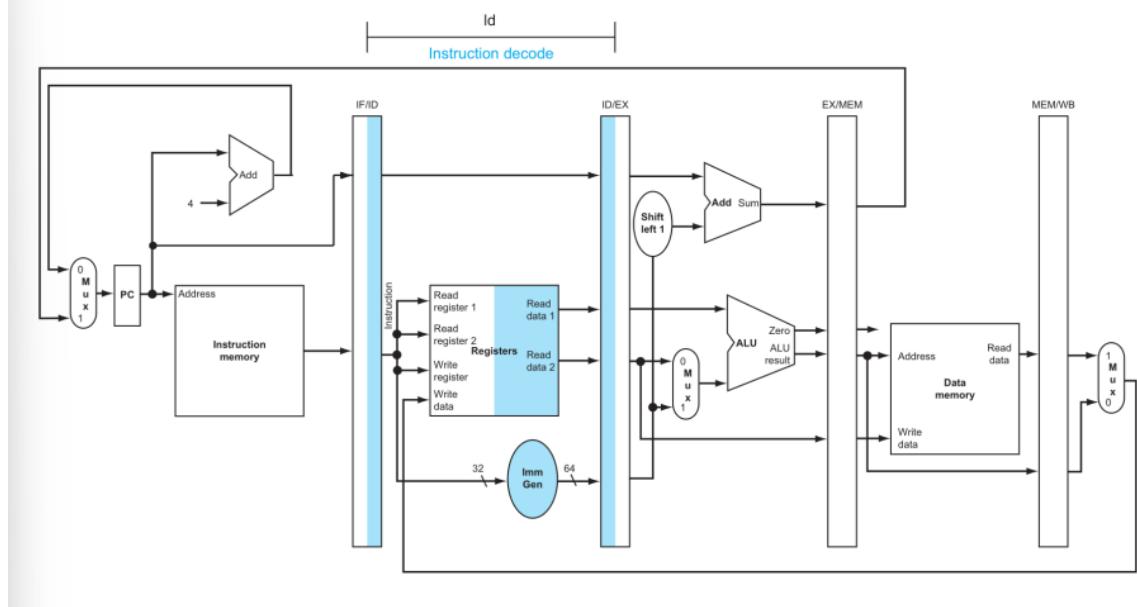
- IF inst : memory로부터 사용할 PC에 있는 주소를 읽어오고, IF/ID Pipeline reg에 두는 것
 - PC주소는 4씩 증가한 다음 다시 PC에 기록된다.



ID Stage for Load Instruction

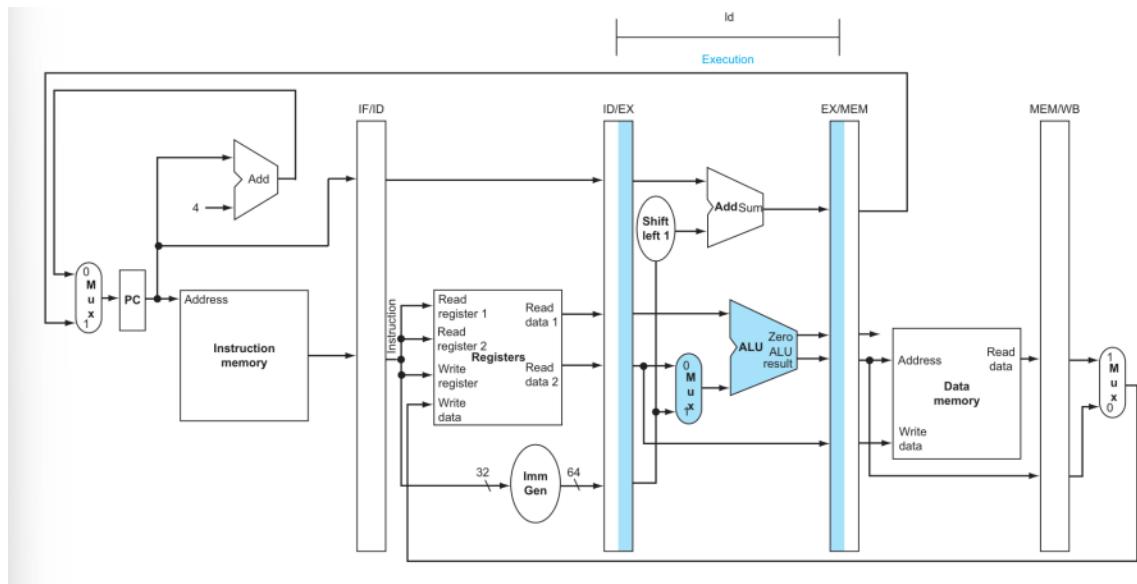
- IF/ID reg의 instruction은 immediate field 와 reg numbers를 지원해준다.

- immediate field 는 sign-extended 한다.(→ 64bit), 그리고, reg numbers는 두 개의 reg에 접근하는데 주로 쓰인다. → 모든 세개의 값들은 ID/EX reg 에 저장된다.



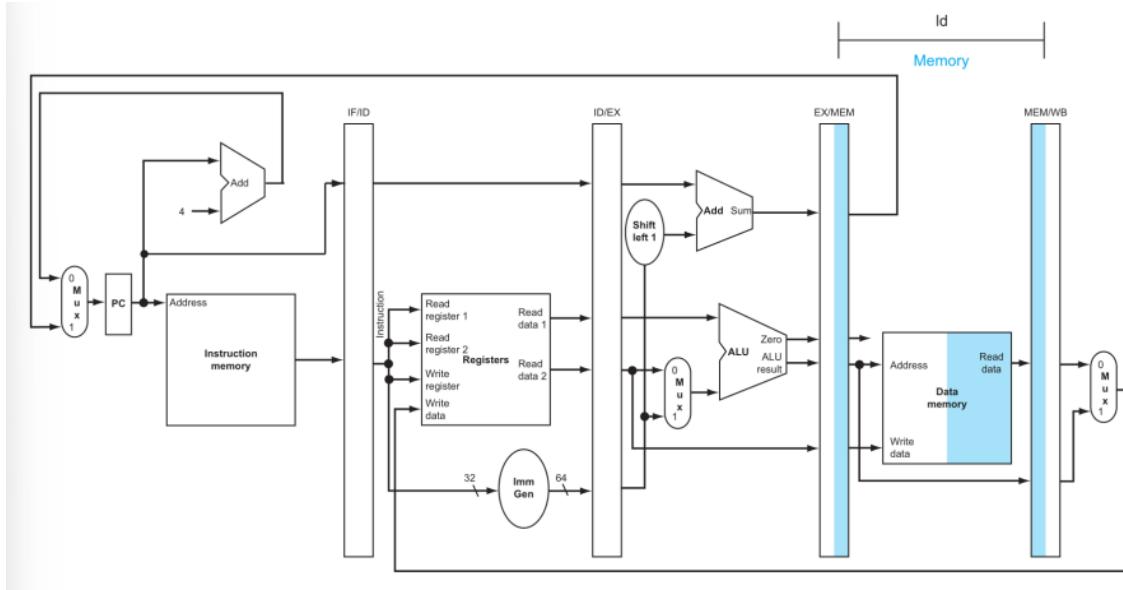
EX Stage for Load Instruction

- ALU는 reg값을 더하고, ID/EX reg 로부터 온 immediate value를 sign-extendededgksek.
 - 결과 값은 EX/MEM reg에 저장한다.



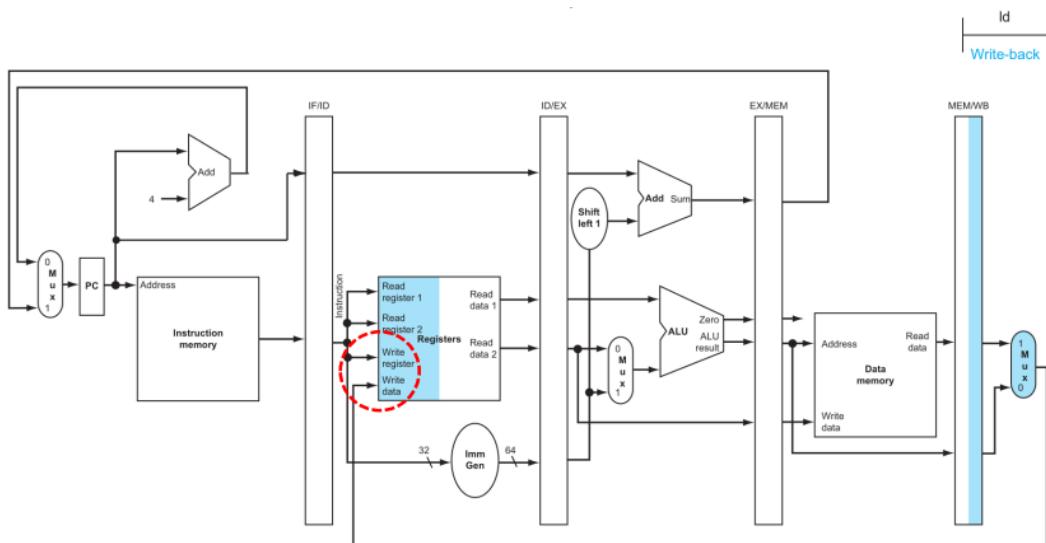
MEM Stage for Load Instruction

- Id inst은 EX/MEM reg로부터 memory에 저장된 주소를 읽어온다.
 - 해당 data 값은 MEM/WB reg에 저장된다.



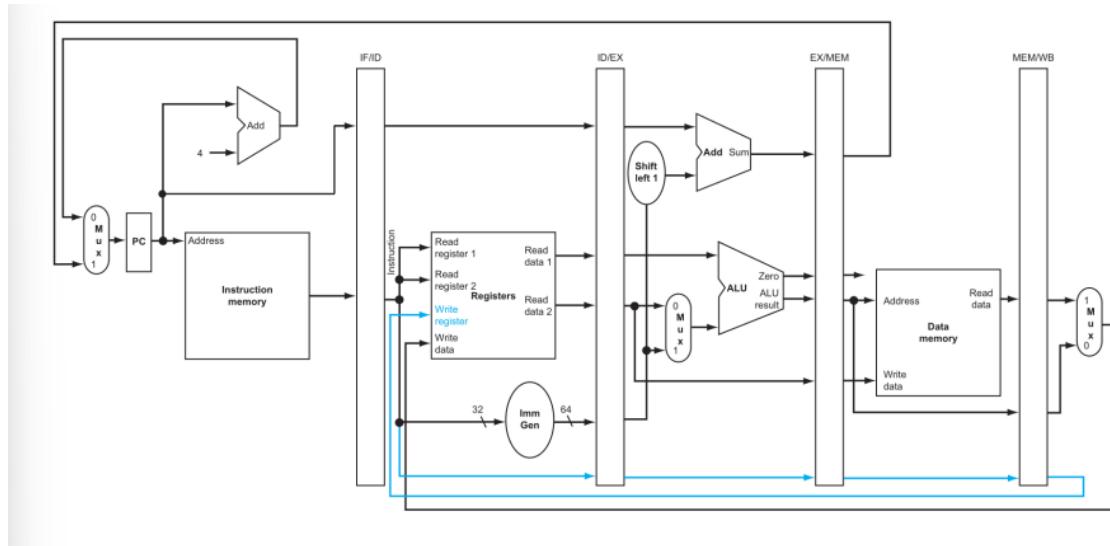
WB Stage for Load Instruction(with Bug)

- data는 MEM/WB reg에서 값을 읽고 reg file에 값을 쓴다.
 - The IF/ID reg는 reg number에 적는 것을 제공해준다.
 - 이 명령어는 로드 명령어 이후에 상당히 발생함.



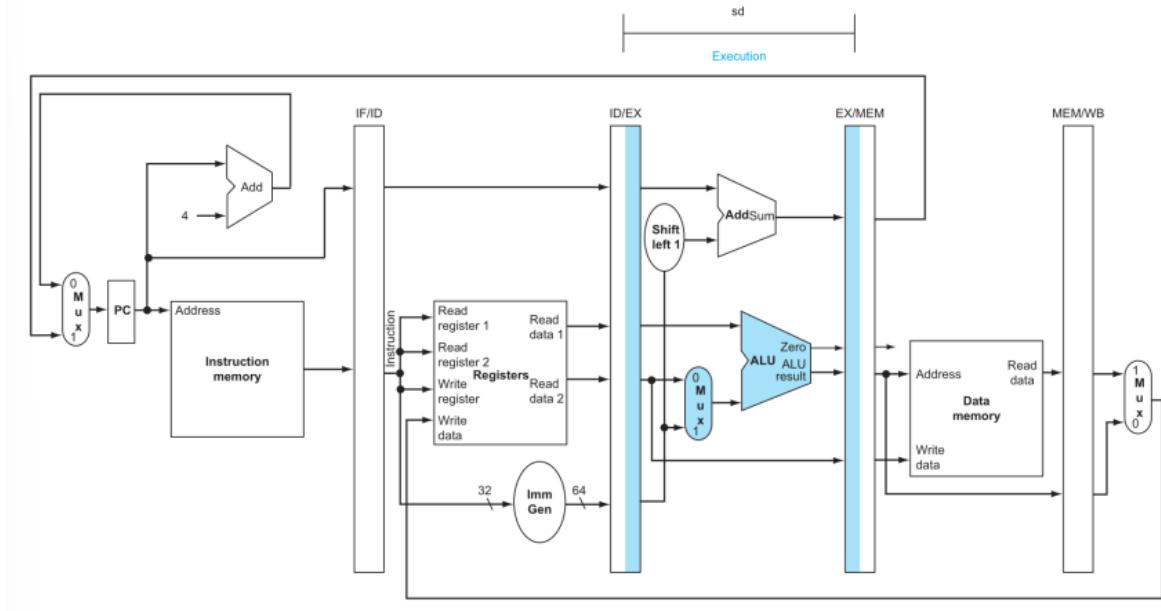
Corrected Datapath for Load Instruction

- rd number은 load instruction에 있는데, 이는 preserved 할 이유가 있다.
 - Passing the write reg number to the 1) ID/EX reg → 2) EX/MEM reg → 3) MEM/WB reg



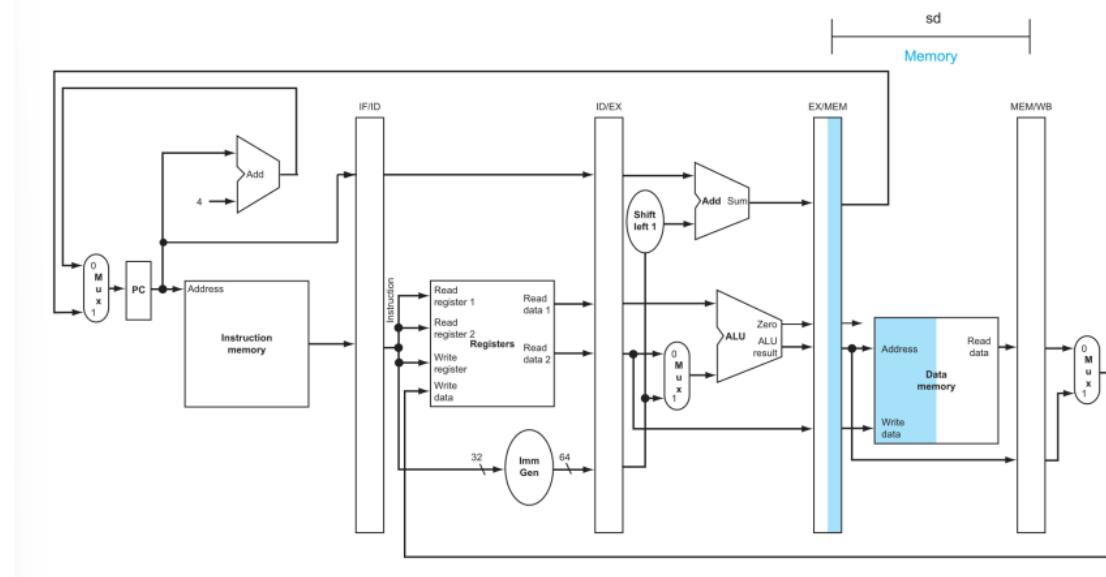
EX Stage for Store Instruction

- 저장 명령에 대한 IF 및 ID 단계는 로드에 대한 단계와 거의 동일함.
- EX stage에서, rs2의 값은 EX/MEM reg에서 다음 stag로 load한다.



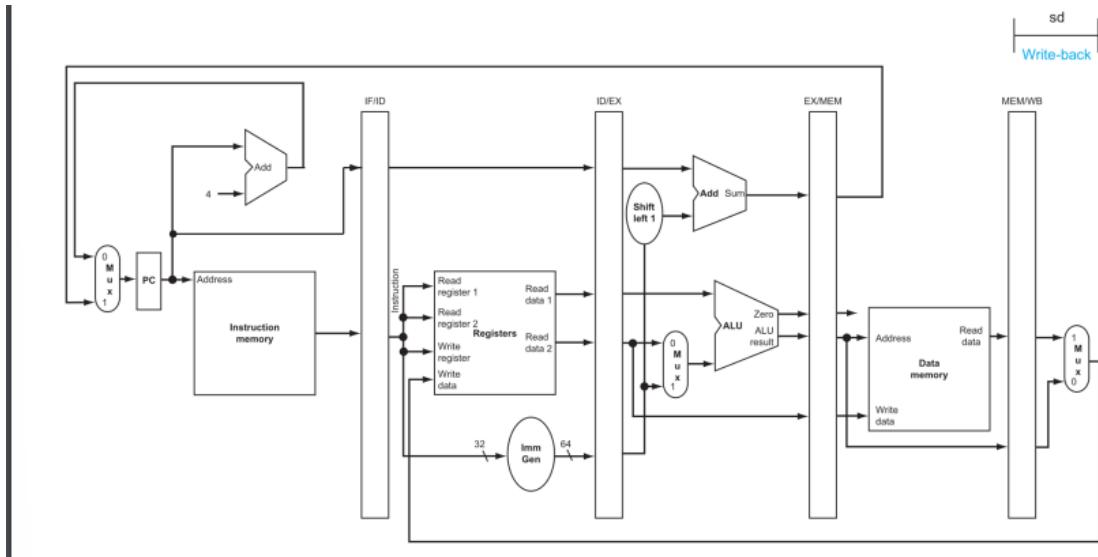
MEM Stage for Store Instruction

- data 가 data memory에 쓰여지기 시작하는 것.
 - 저장할 데이터가 포함된 레지스터는 이전 단계에서 읽고 ID/EX레지스터에 저장한다.
 - data는 또한 EX/MEM reg에 둔다.



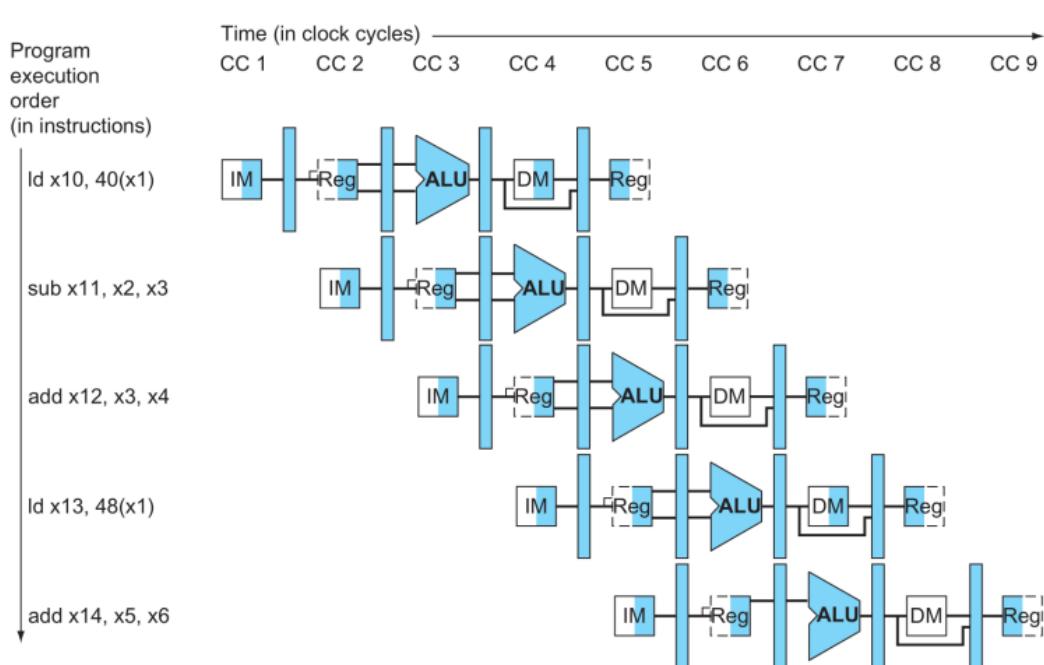
WB Stage for Store Instruction

- instruction에 저장하기 위함, WB 단계에서는 어떠한 일도 일어나지 않는다.
 - 모든 instruction은 이미 이전 프로세스에 대한 값이 저장되어 있다.
 - 이것은 어떠한 값이 없어도 다음 스테이지로 넘기는 작업을 한다.



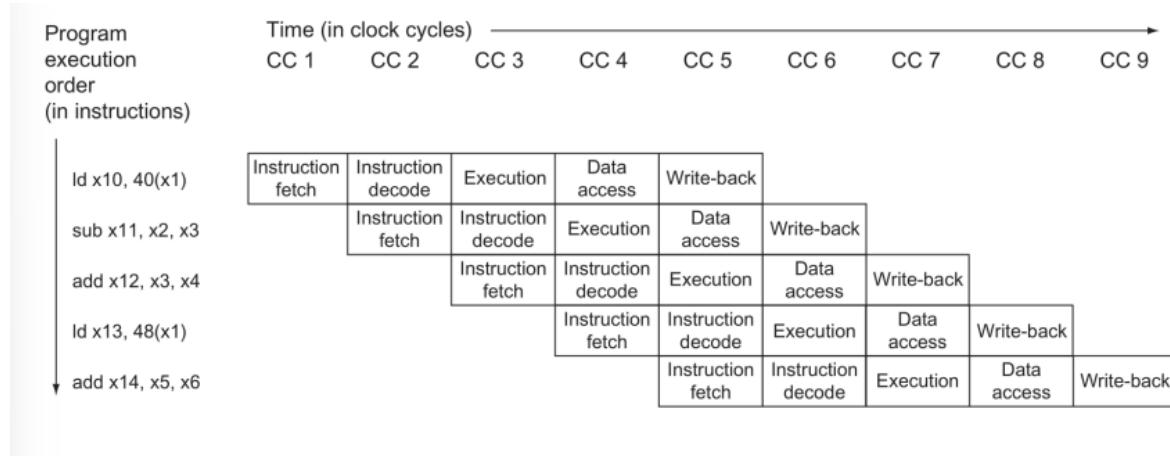
Multiple Clock Cycle Pipeline Diagram

- 아래 그림은, 각 단계에서 resources가 어떻게 사용되는지 보여준다.



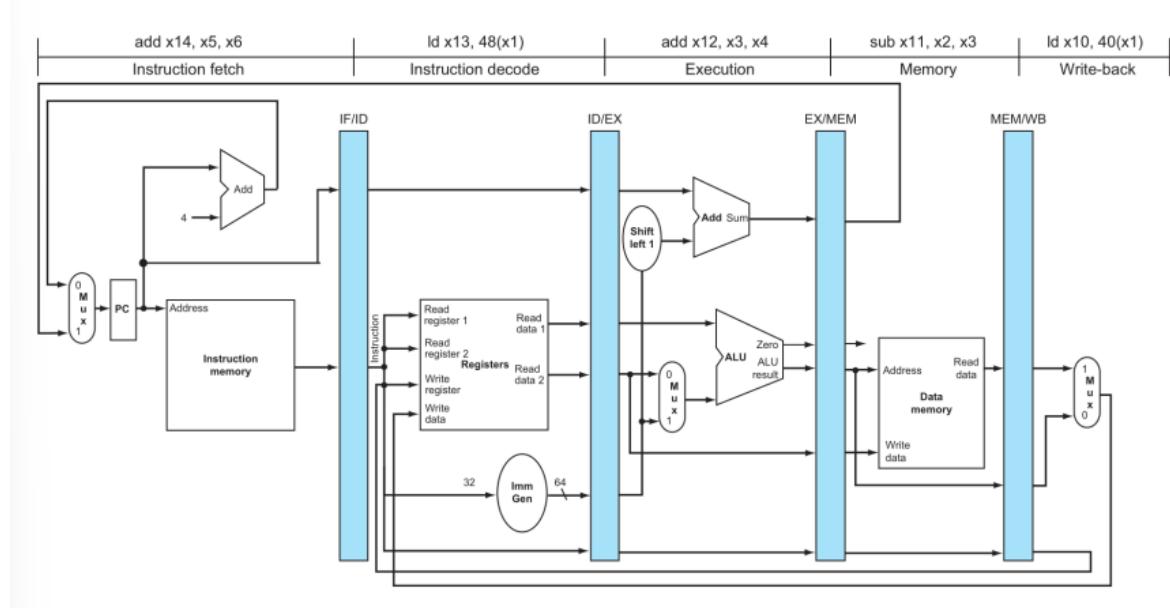
Multiple Clock Cycle Pipeline Diagram

- 전통적인 multiple cycle pipeline diagram은 수행 요소대신 각 단계의 이름을 사용한다.



Single-clock-cycle Diagram

- The single-clock-cycle figure shows the flow of data and control signals through the pipeline stages.

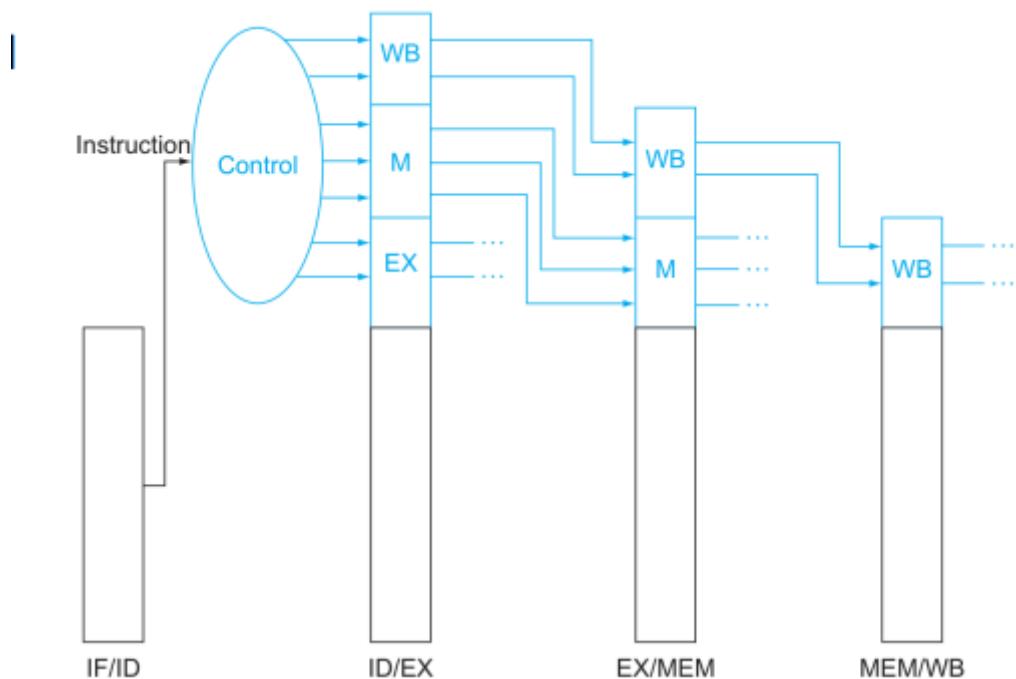


Pipeline Control

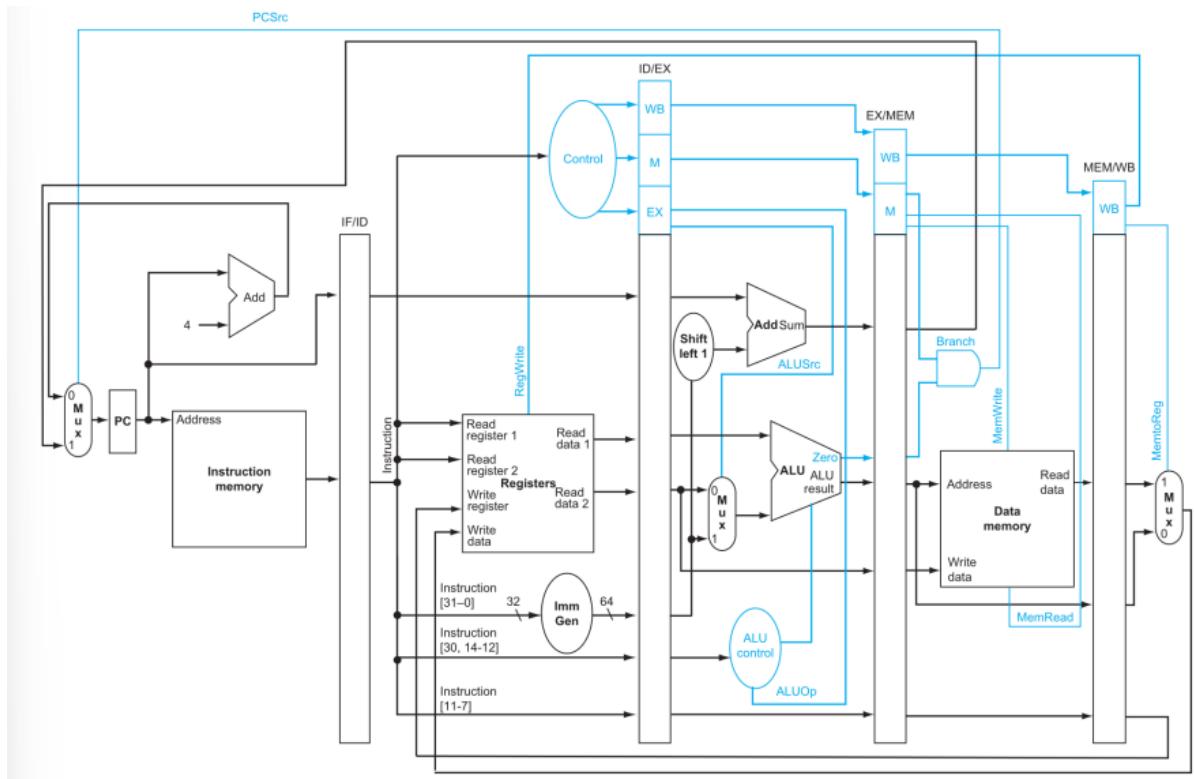
- Control lines grouped by stages

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

- Pipeline reg는 control 신호를 포함해서 확장한다.
 - Control lines 는 EX 단계를 포함한다.
 - ID/EX : 7개의 control lines
 - EX/MEM: 5 control lines
 - MEM/WB : 2 control lines

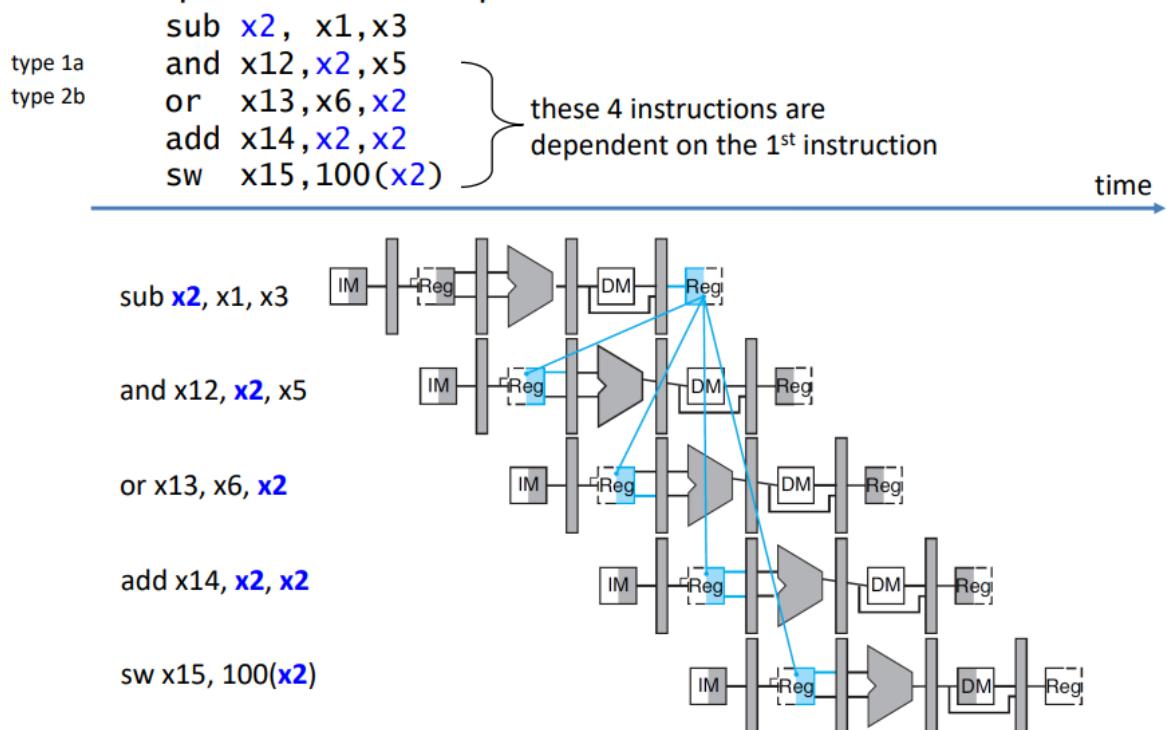


Pipelined Datapath with Control



Data Hazards

- 아래 inst의 예가 있다.



위의 그림에서 볼 수 있다 시피, reg가 필요로 하는 곳은 총 4개가 있는데, 문제는 시간의 문제가 있다. 왜냐면 and, or 연산에서는 reg가 필요한데, 저기서는 reg를 바로 얻어올수가 없기 때문이다.

Dependence Detection

■ Notations

- Pipeline Registers
 - IF/ID, ID/EX, EX/MEM, MEM/WB
- Fields in the pipeline registers
 - ID/EX.RegisterRs

■ Detection

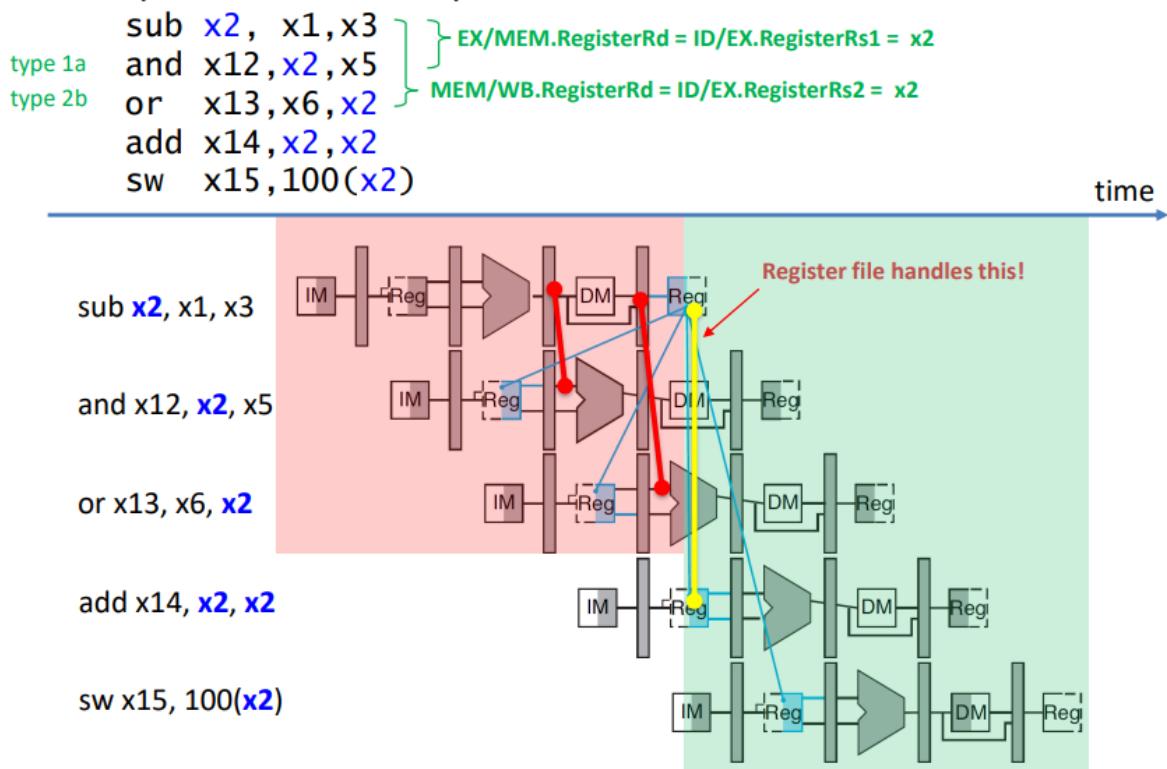
- Pass register numbers across the stages

■ Data hazard is detected when:

- type 1a • EX/MEM.RegisterRd = ID/EX.RegisterRs1
- type 1b • EX/MEM.RegisterRd = ID/EX.RegisterRs2
- type 2a • MEM/WB.RegisterRd = ID/EX.RegisterRs1
- type 2b • MEM/WB.RegisterRd = ID/EX.RegisterRs2

Data Hazards

- Example instruction sequence



- Problem with current detection logic
 - It always forwards automatically regardless of instructions
→ Check if RegWrite signal is active
 - EX/MEM.RegWrite=1, MEM/WB.RegWrite=1
 - Attempt to write to x0
→ Forwarding logic will deliver non-zero to **and** instruction
 - EX/MEM.RegisterRd ≠ 0

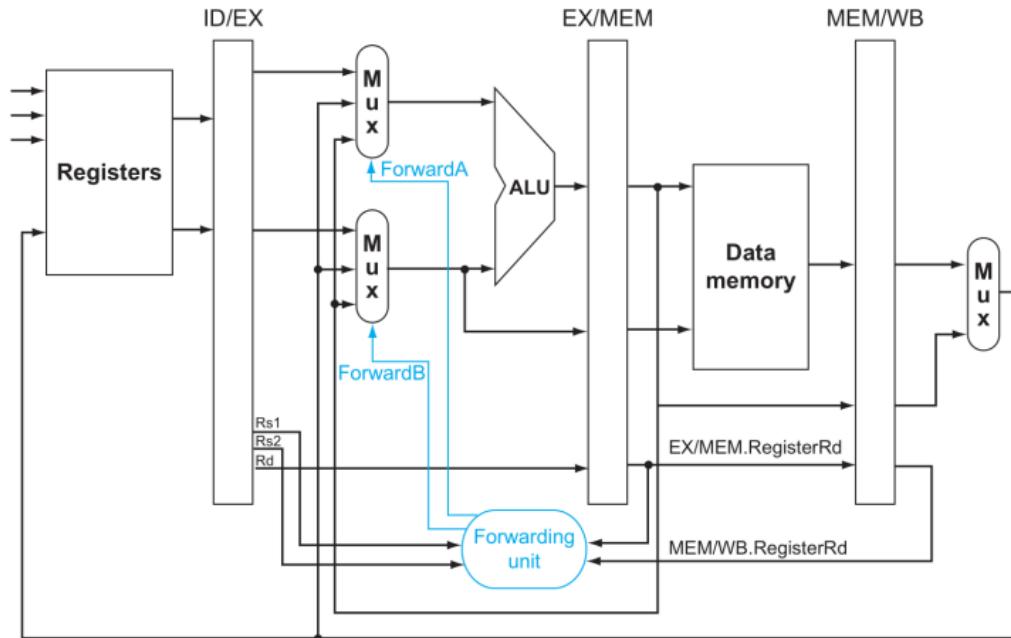
위의 그림에서는 스택에서 rs, rd의 위치만 맞으면 인스트럭션의 종류와는 상관없이 위치에 따라서만 값을 가져왔다. 그렇게 되면, Id나 r타입의 경우 구분이 안 되기에, RegWrite와 같은 신호를 준다. 그리고 x0의 경우는 forwarding이 되지 않도록 주의한다.

```

    sub x0, x1, x3
    and x12, x0, x5
  
```

Forwarding Unit

- The multiplexors are expanded to add the forwarding paths
 - The multiplexors select either the register file values or one of the forwarded values



- Control values for the forwarding multiplexors

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

- Full logic for EX hazard (type 1)

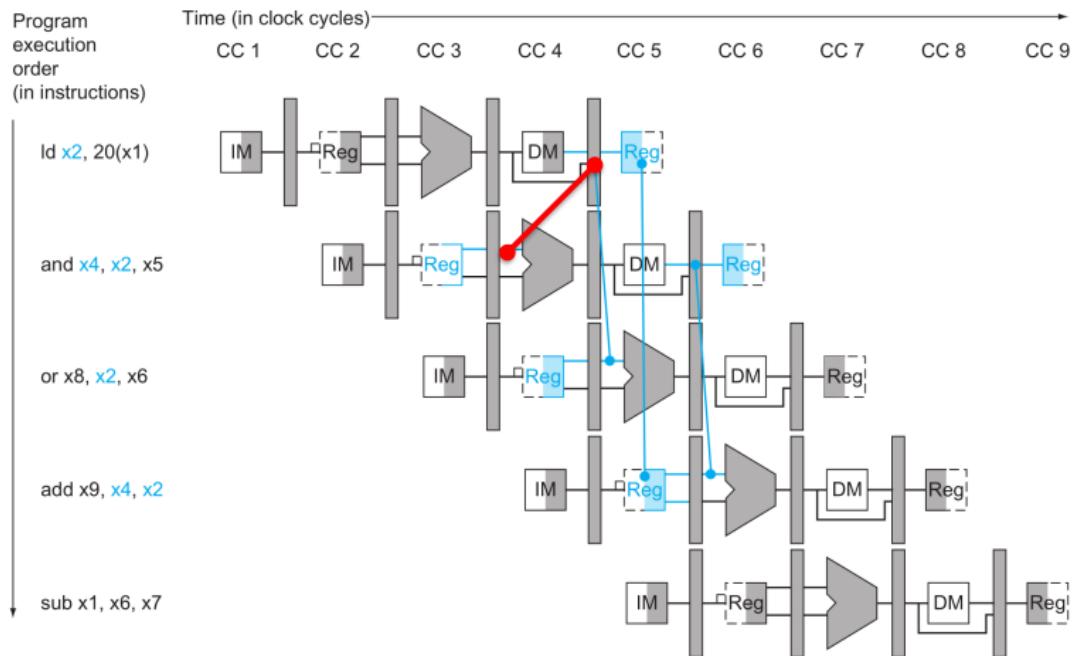
```

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
  
```

- Write the full logic for MEM hazard (type 2)

Load-use Harzard

dl hazard는 forwarding을 해결 할 수 없다. 왜냐면, load와 and 사이에 시간적 차가 발생하기 때문에, 따라서 이 결과에 의해, stall가 들어가야 한다.



Hazard Detection Unit

Hazard detection은 ID stage에서 stall이 일어났을 때, 발생 할 수 있다.

- Logic

```

if(ID/EX.MemRead and
((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
(ID/EX.RegisterRd = IF/ID.RegisterRs2)))
    stall the pipeline
  
```

test if it is a load instruction

Stalling a pipeline

PC reg 및 IF/ID 없데이트 방지(+4)

다음 사이클에서 같은 inst이 페치할수있기에

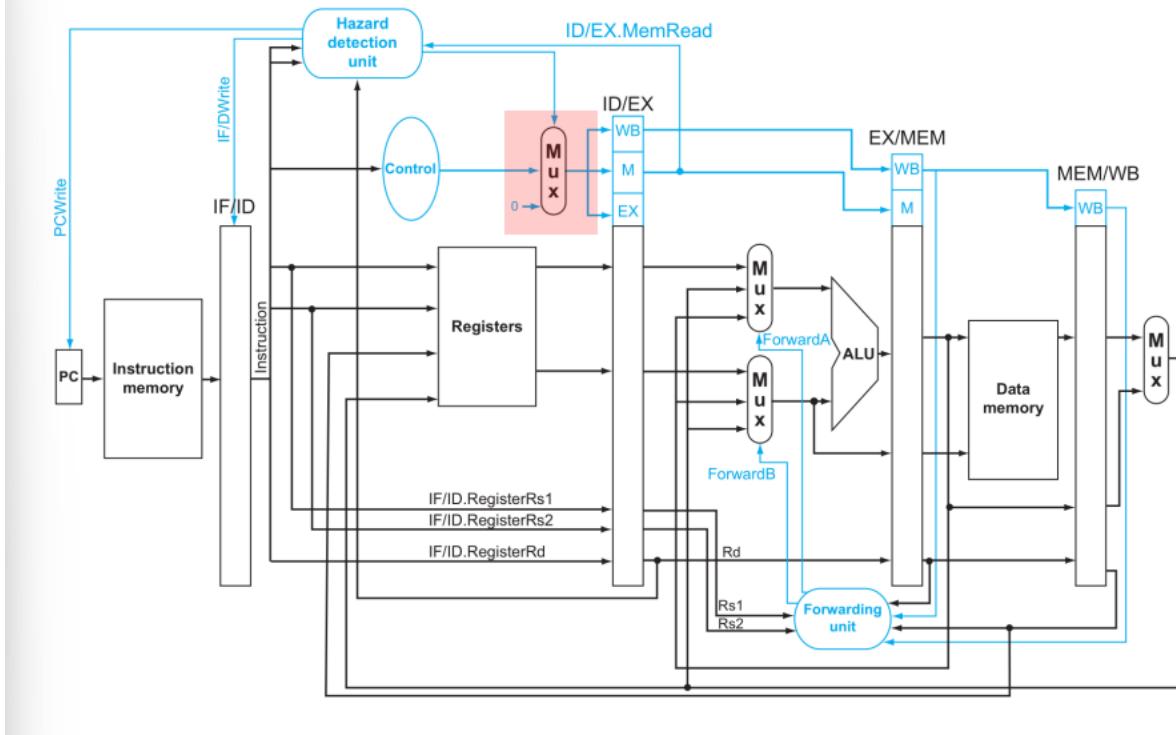
Insert nops(no-operation)

ID/EX 단계에 있는 모든 신호를 0으로 만들어 버리기

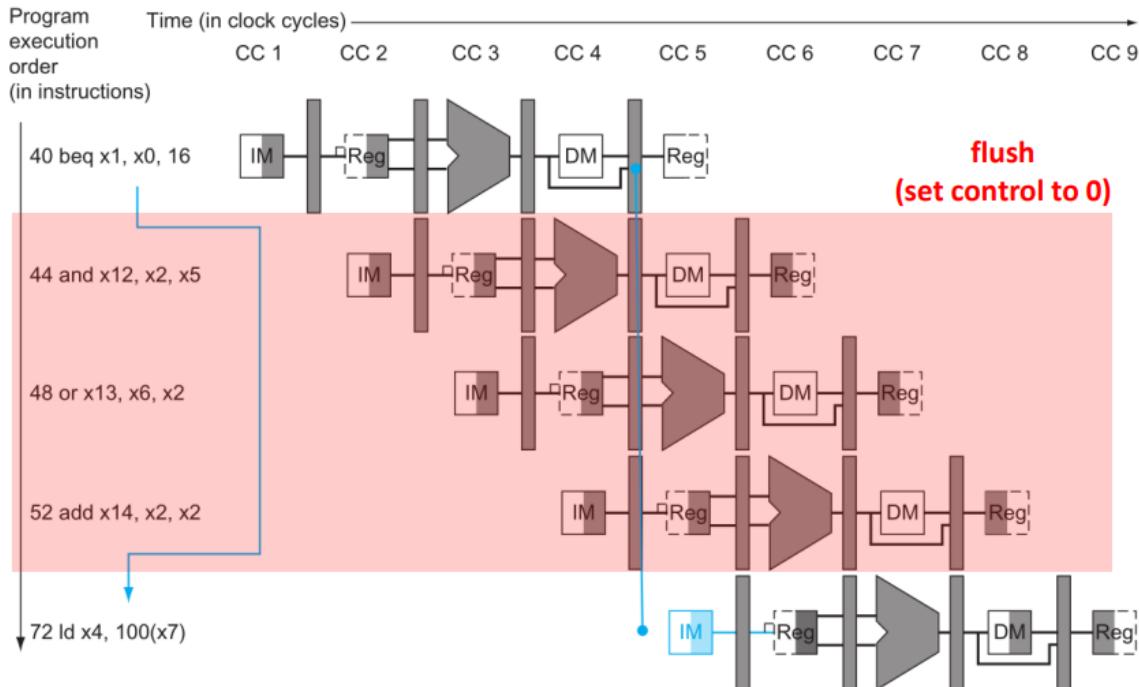
어떠한 값도 쓸 수가 없다.

stalling 후, forwarding logic kicks in.

Datapath with Forwarding and Hazard Detection



Control Hazard



Reducing the Delay of Branches

- 조건 분기문의 수행을 파이프라인의 앞쪽에 미리 보낸다. → 두 개의 액션이 반드시 필요로 한다.
 - Branch target address를 계산한다.
 - EX → ID로 옮긴다.
 - branch decision을 평가한다.
 - HW for branch decision : ID로 옮기기에 가벼운지 체크한다.
 - Moving the HW affects the forwarding logic

More case to consider

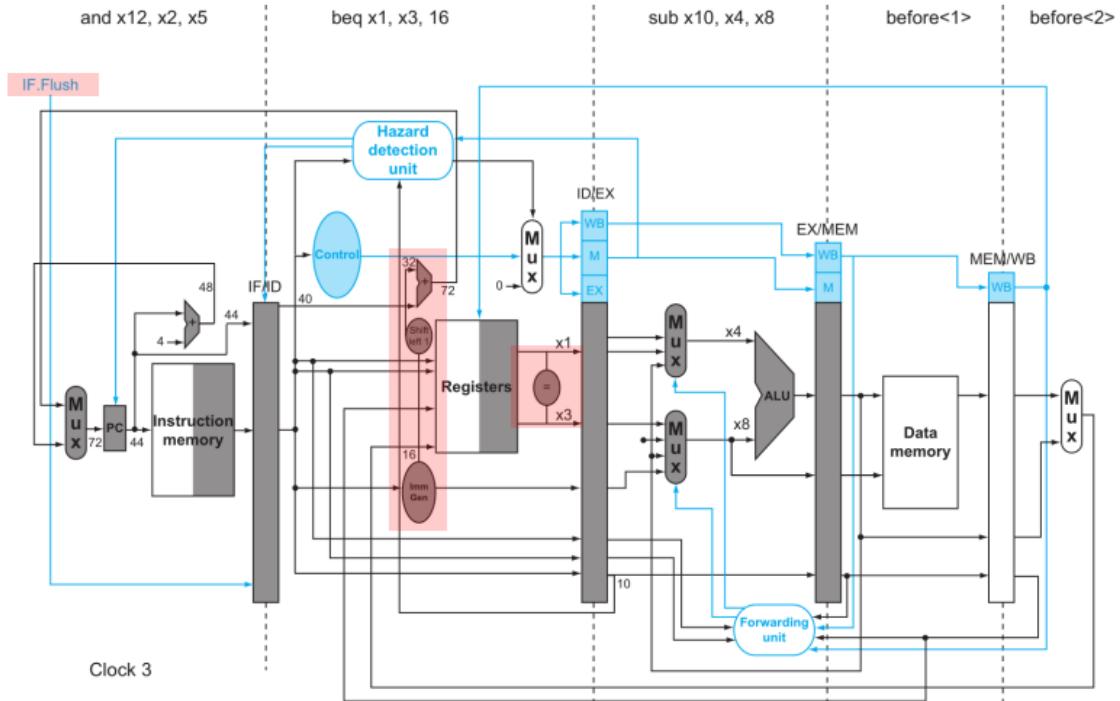
ALU inst은 reg dependency와 함께 branch inst를 따른다.

Data hazard(HW을 ID로 옮겨갔기 때문)

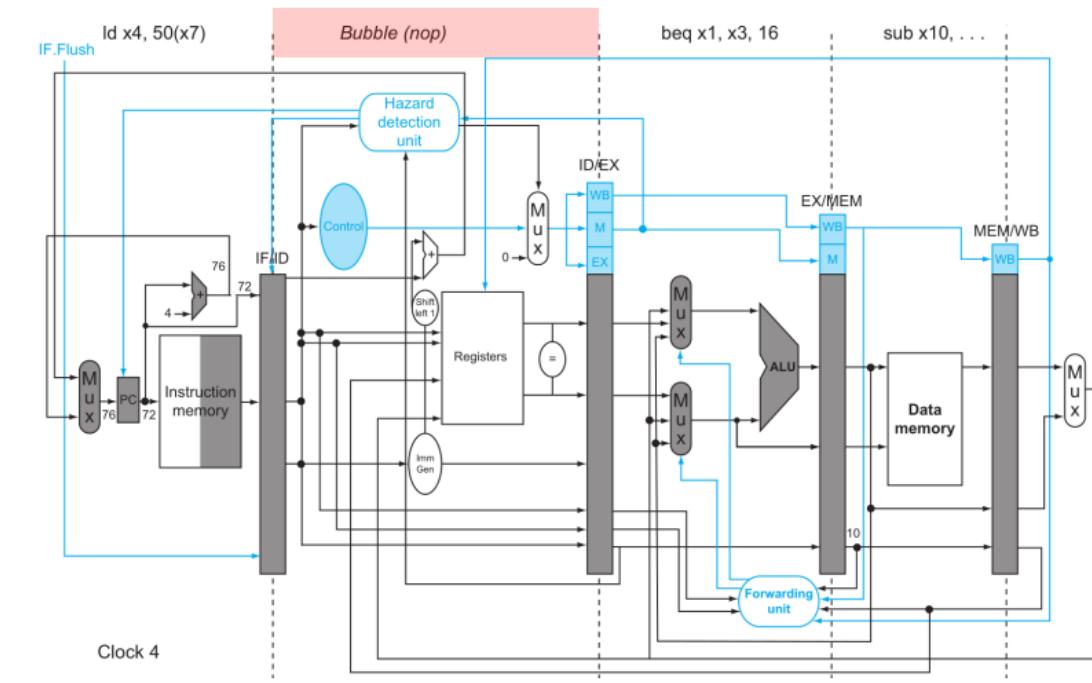
Load inst은 branch inst를 따른다. : the same

EX : Branch Taken

- The ID stage of cycle 3 determines a branch must be taken
 - Select 72 as the next PC address and zeros the instruction fetched for next



- The instruction at location 72 is being fetched
 - Single bubble or nop instruction in the pipeline due to the taken branch

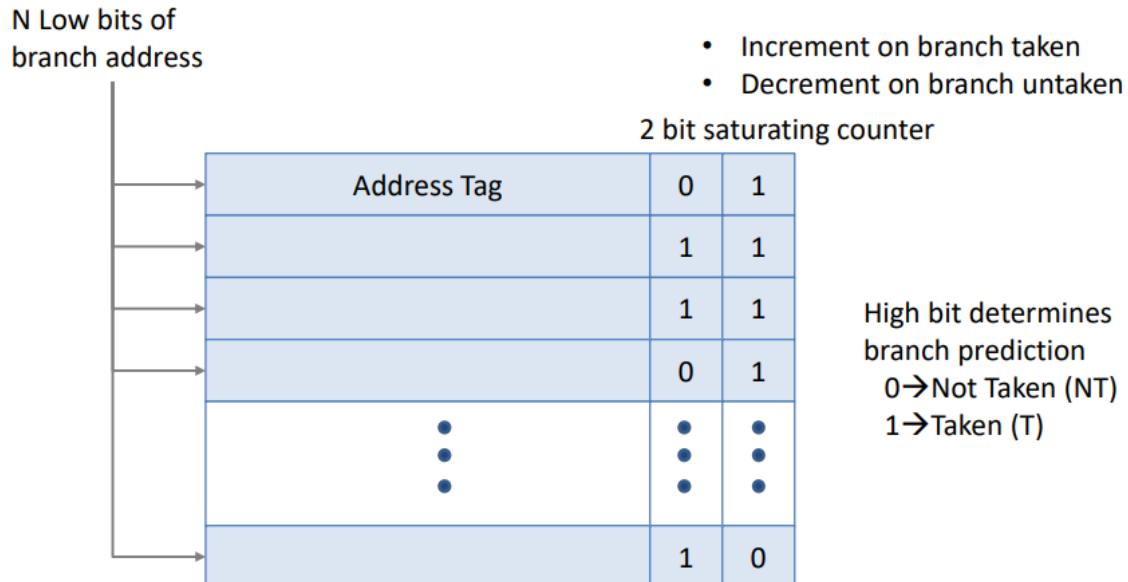


Dynamic Branch Prediction

- Static prediction : 깊은 파이프라인을 기대하기는 어렵다(그리고, multiple 이슈도 있다) → branch penalty 가 높다.
- Runtime information을 사용해서 prediction 의 정확성을 높일 필요가 있다. → 'Dynamic
 - Branch prediction buffer(Branch history table)
 - beq address의 작은 메모리의 LSB의 인덱스
 - 분기가 최근에 취해졌는지 여부를 나타내는 예측 비트

Branch Prediction Buffer

Branch History Table (BHT), Branch History Buffer (BHB)



Branch Prediction Accuracy

- Using SPEC 89 benchmark

