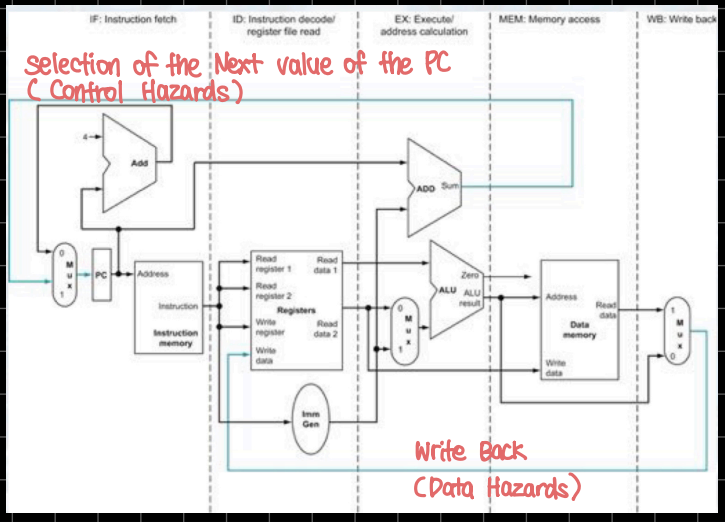
A Pipelined Datapath

- 5 stage
  - ① IF : Instruction Fetch
  - ② ID : Instruction Decode + register file read
  - ③ EX : Execution or address calculation
  - ④ MEM : Data Memory access
  - ⑤ WB : Write Back



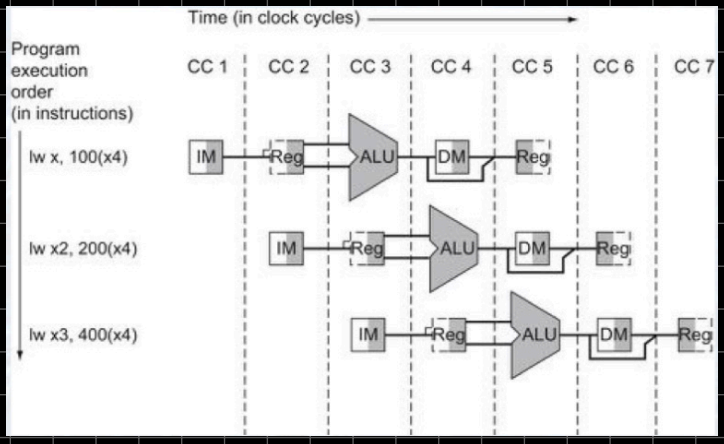
- Instructions / data move generally from left → right
  - ↳ there are two exceptions :
    - ① the write back stage
      placing the result back into register file in the middle of the data path
    - ② selection of the next value of the PC
      choosing between the incremented PC and the branch address from the MEM stage

그림의 파란색선 → 특이하게 우에서 좌로 이동

⇒ pipelined datapath에서 Hazard가 발생할 수 있다

※ Right-to-Left Flow Leads to Hazards

Stylized Version 간단한 버전

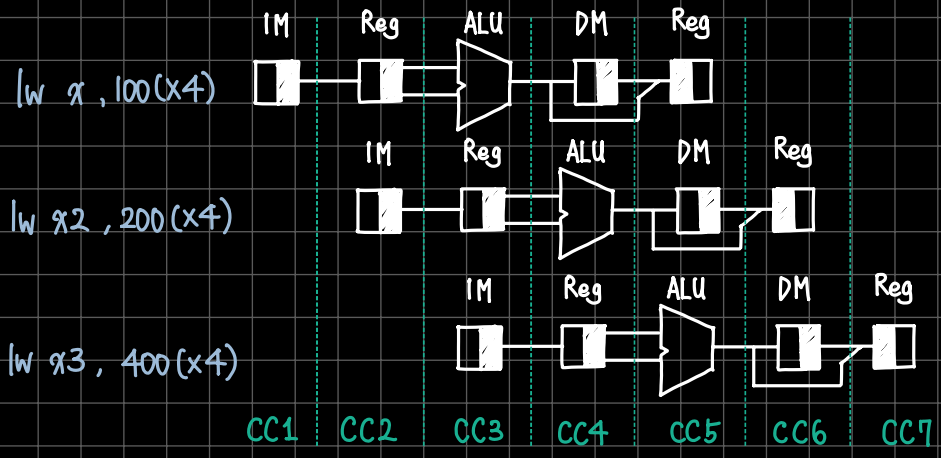


▯ ⇝ writing data at register

▯ ⇝ reading data from register

- pretending that each instruction has its own datapath
- placing these datapaths on a time line to show their relationship

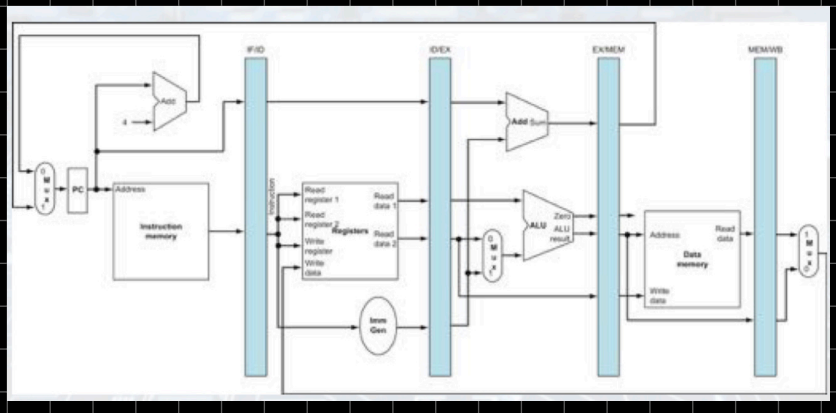
Pipeline Registers



- Figure seems to suggest that three instructions need three datapaths
  - the instruction memory is used during only one of the five stages
  - It can be shared by other instructions during other four stages
- To retain the value of an instruction
  the value (instruction) must be saved in a register

⇒ we must place registers wherever there are dividing lines between stages

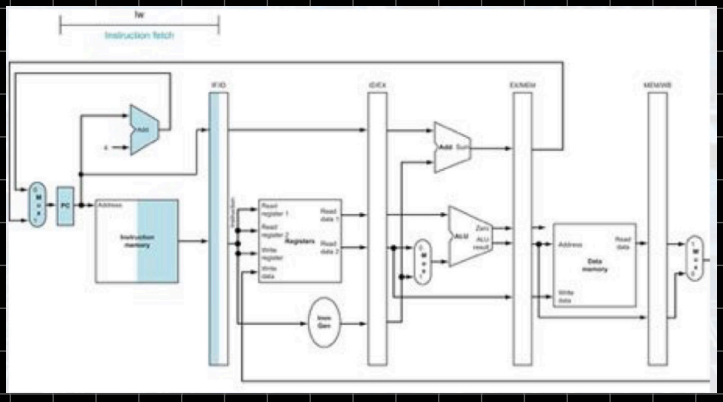
Datapath with Pipeline Registers 각 단계의 연산 결과를 저장하는 파이프라인 레지스터



- IF/ID : Instruction Fetch
  Instruction Decode
- ID/Ex : Instruction Decode
  Execution
- Ex/Mem : Execution
  Memory Access
- Mem/WB : Memory Access
  Write Back

load word의 예시 → 5단계를 모두 이용하는 명령어라서

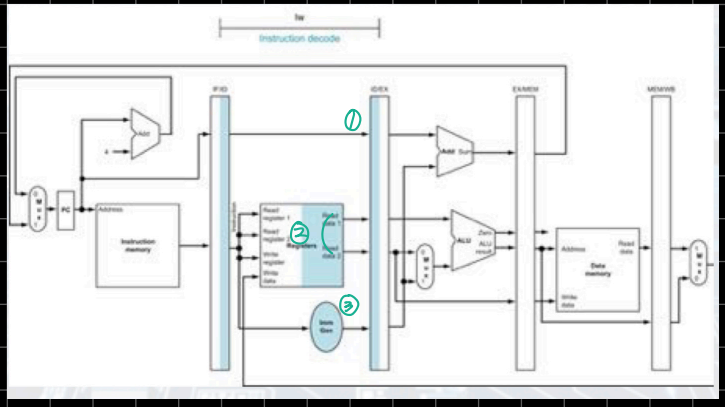
1. Instruction Fetch 단계 요약 : beq에 사용될것을 대비하여 (PC+4)의 new PC value 가 저장된다



- The instruction is read from instruction memory using the address in the PC and then placed in the IF/ID pipeline register
  PC 값을 이용해 instruction memory의 instruction을 읽어와서 IF/ID 파이프라인 레지스터에 저장한다
  ↳ Instruction Decode 전까지는 어떤 명령인지 알수 없기 때문에
  - The IF/ID pipeline register is similar to the instruction register

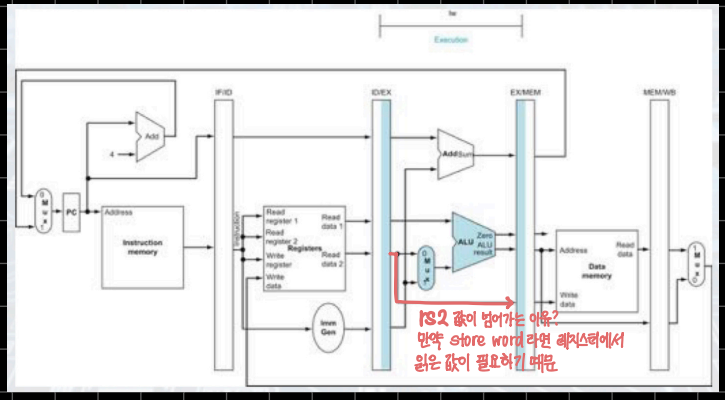
- The PC address incremented by 4 to be ready for the next clock cycle
  This PC address is also saved in the IF/ID pipeline register
  The computer cannot know what type of instruction is being fetched
  만약 명령어가 branch일 경우 나중에 PC값 연산이 있으므로 PC값을 저장해둬야 한다
  Decode 전까진 어떤 명령어인지 모르니까 일단 저장

2. Instruction Decode 단계 : rs1, rs2, imm, (PC+4)



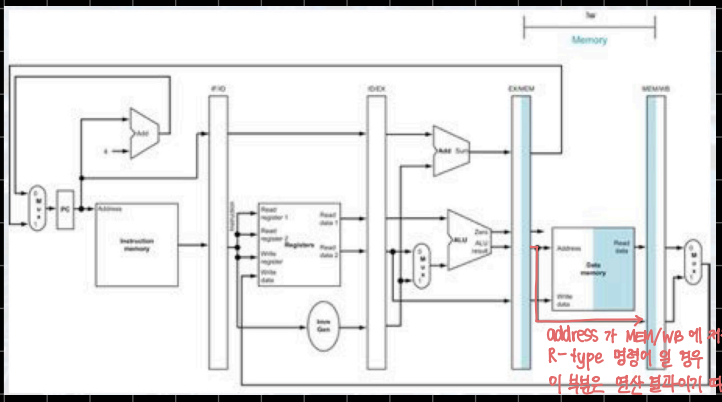
- Instruction portion of the IF/ID pipeline register supplies the immediate field and the register numbers to read
- All three values are stored in the ID/EX pipeline register along with the incremented PC address
  ↳ register 1, register 2, (PC+4) address, immediate

3. Instruction Execution 단계 sum = rs1 + imm (base address), sum 저장, (sw 면 rs2 저장)



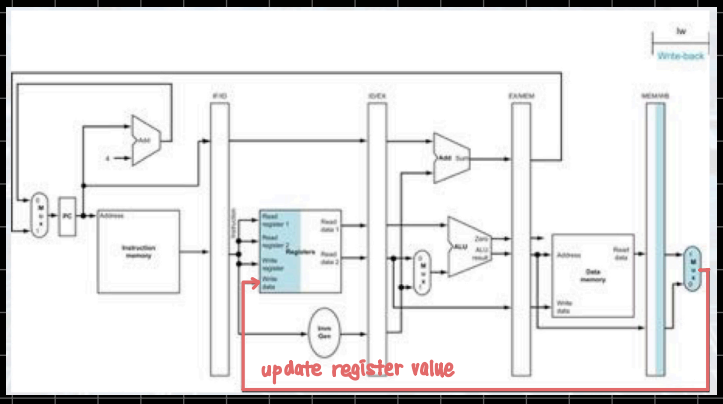
- The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using ALU
- The sum is placed in the EX/MEM pipeline register

4. Memory Access 단계



- The load instruction reads the data memory using the address from the EX/MEM pipeline register
- The instruction loads the data into the MEM/WB pipeline register

5. Write Back 단계



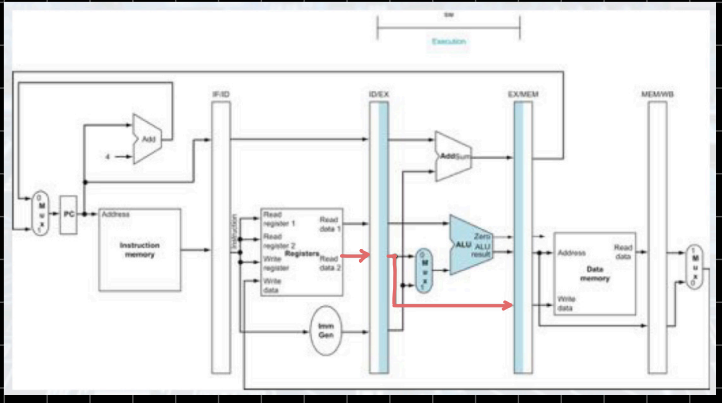
- The instruction reads the data from the MEM/WB pipeline register and writes it into the register file
  - the register file is in the middle of the figure

store word 의 예시

1. Instruction Fetch , 2. Instruction Decode
   - The first two stages show identical operations
     - Instruction Fetch
     - Instruction Decode and register file

     ~> 모든 명령어에 대해 IF, ID 단계는 동일하다
   - It is too early to know the type of the instruction

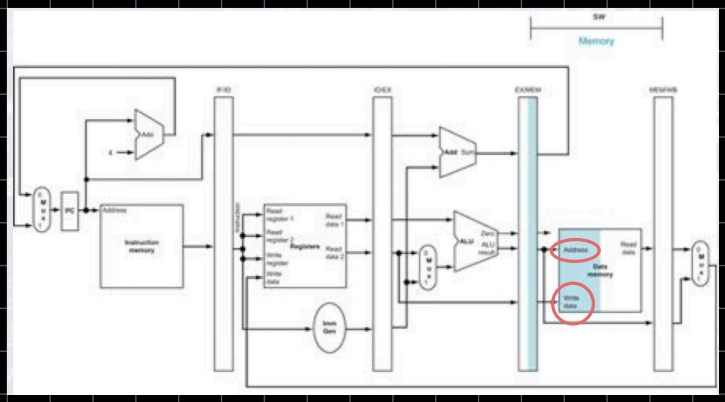
3. Instruction Execution (EX)



- One of the registers read in the ID stage need to be passed to the MEM stage
  → the rs2 value is loaded in the EX/MEM pipeline register so that it can be stored in memory

  SW의 경우 저장할 값이 mem 단계까지 전달되어 memory에 저장되어야 한다
- The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register

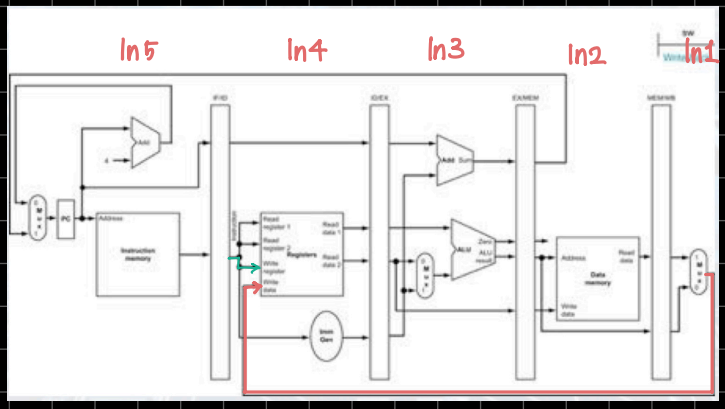
4. Memory Access



- ALU → EX/MEM pipeline register를 거쳐 전달된 address 와
  read data2 → ID/EX pipeline register → EX/MEM pipeline register를 거쳐 전달된 write data 를 이용해 memory에 data를 store

5. Write Back

→ 할게 아무것도 없다. 그냥 다음 명령을 기다리는 단계



Instruction 4의 레지스터 (write register)

Instruction 1의 레지스터 (write data)

Instruction 1의 레지스터 값을 업데이트 해야하는데
Instruction 4의 레지스터값이 업데이트 된다

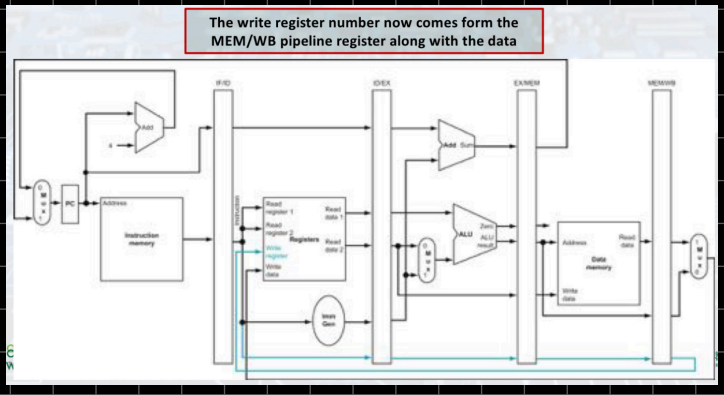
즉, In1의 write data가
In4에서 보낸 레지스터에 저장된다
다른 레지스터 넘버에 저장되는 버그 발생

★ Bug가 발생한다

Uncover a Bug

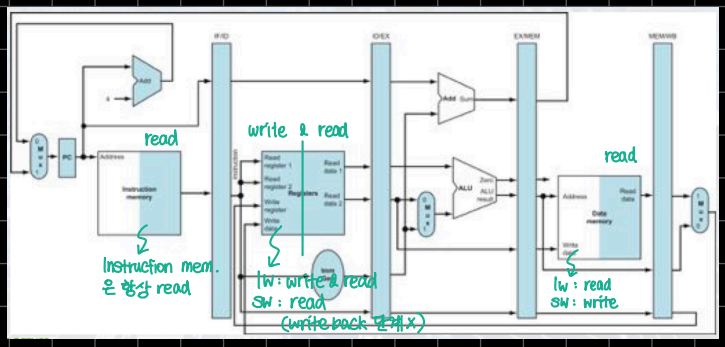
- which instruction supplies the write register number in the final stage of the load?
- The IF/ID register pipeline register supplies the write register
- We need to preserve the destination register number in the end
- A load must pass the register number from the ID/EX through EX/MEM to the MEM/WB pipeline register for the use in the WB stage

Corrected Pipelined Datapath



instruction 1의 write register number를 따로 pass 해서 올바른 위치에 저장되도록 해준다

All Five Stages of a Load Instruction

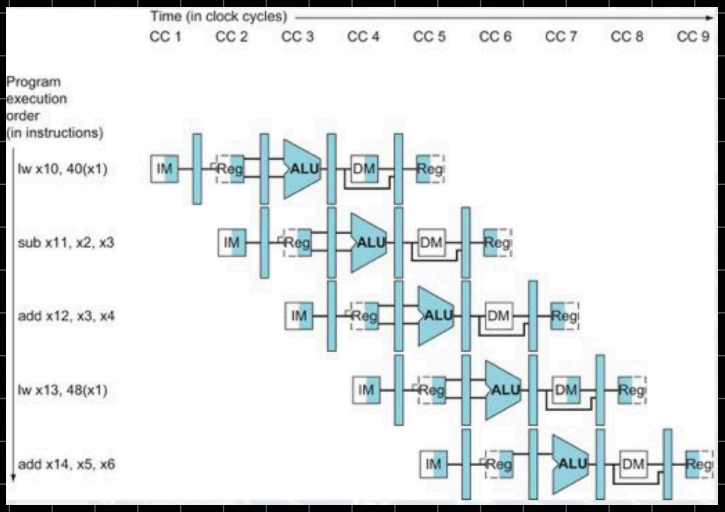


R, sw/lw, branch 中 모든 단계를 사용하는 것은 load 뿐
즉 pipeline을 시행하면 사용하지 않는 단계에선 delay가 발생하는 것이다
→ 각 단계 실행시간은 늘어나지만 throughput이 증가해서 총 실행시간이 감소한다는 것을 확인가능

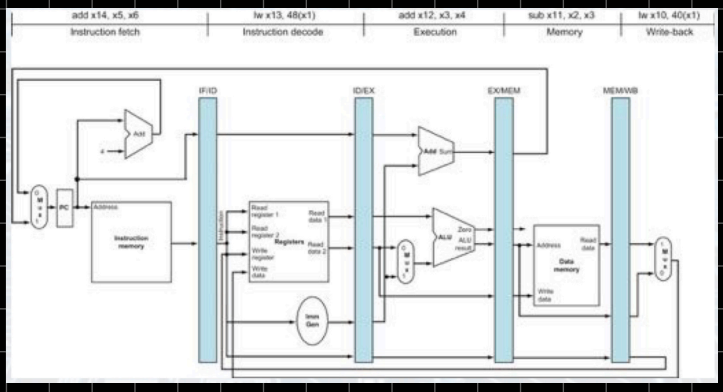
Graphically Representing Pipelines

- Two basic styles of pipeline figures :
  - Multiple-clock-cycle diagrams
  - Single-clock-cycle diagrams

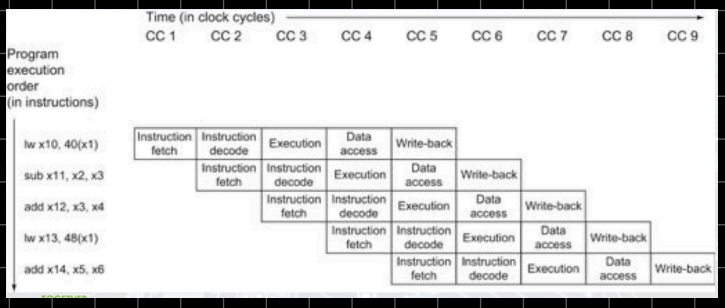
Multi-Clock-Cycle Pipeline



Single-Clock-Cycle Diagram (Clock Cycle 5)



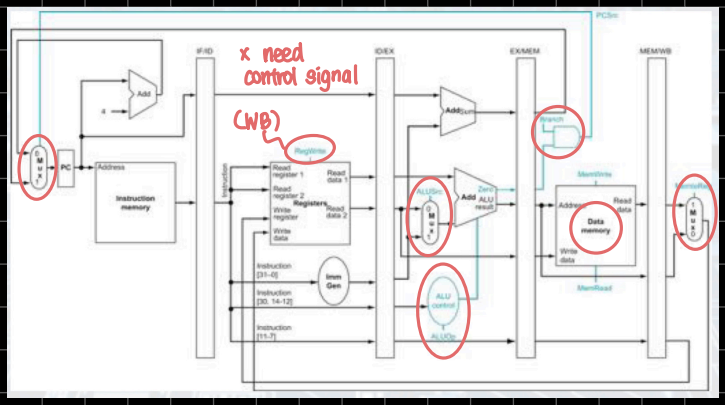
Traditional Multiple-Clock-Cycle Pipeline Diagram



Pipelined Control

- The PC is written on each clock cycle
  → no separate write signal for the PC
- There are no separate write signals for the pipeline registers
  → IF/ID, ID/EX, EX/MEM, MEM/WB
  → the pipeline registers are also written during each clock cycle
- We need only set the control values during each pipeline stage
  → Each control line is associated with a component active in only a single pipeline stage

Pipeline Datapath with Control



Five Stages for Pipelined Control

- We can divide the control lines into five groups according to the pipeline stage

1. IF : The control signals to read instruction and to write the PC are always asserted (nothing special)
   * If the previous instruction is branch, need to decide new PC values
2. ID : The same thing happens at every cycle no optional control lines to set
3. Execution / address calculation
   : RegDst, ALUOp, ALUSrc should be set
4. MEM : Branch, MemRead, MemWrite
5. WB : MemtoReg, RegWrite

↘ 0이면 rt 필드 I[20:16]가 Write reg 전달
1이면 rd 필드 I[15:11]가 Write reg 전달

Pipelined Control : ALU Control

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	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

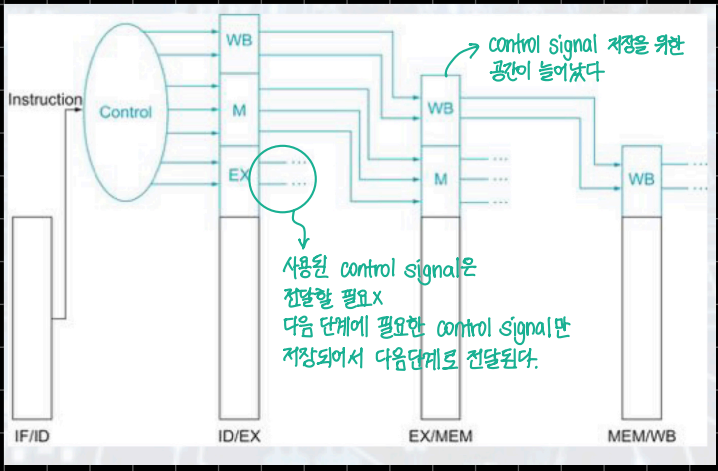
Pipelined Control : Control Signals

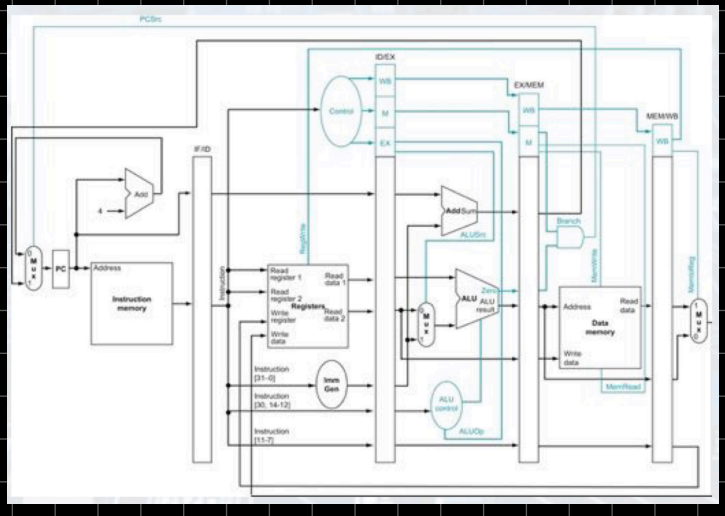
Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits 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.

Pipelined Control : Values of the Control Lines

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
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Implementation of Control Signals





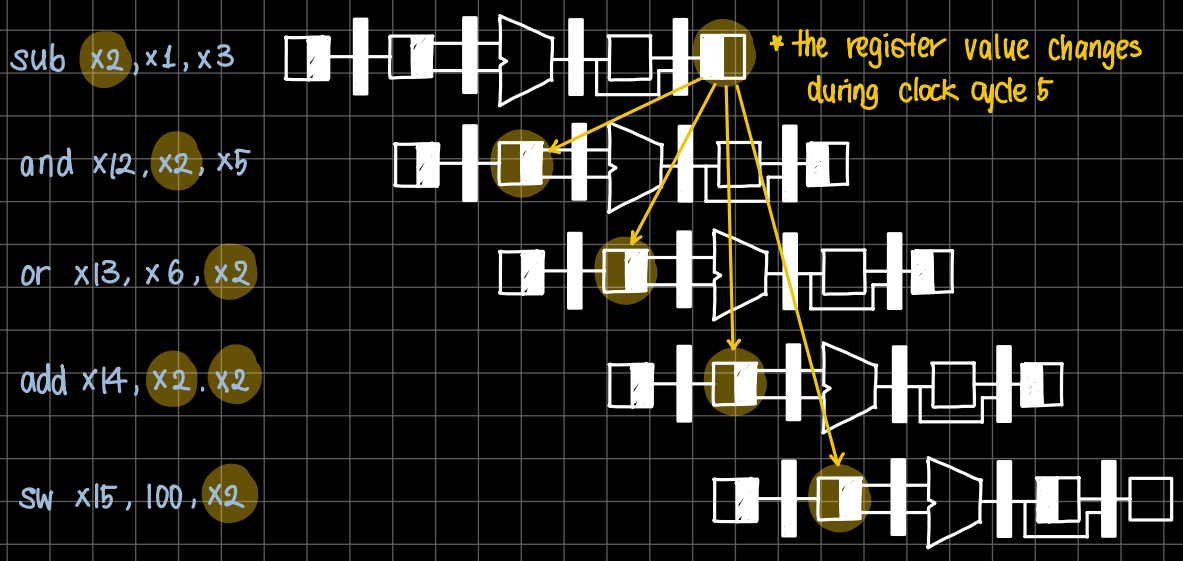
Implementing control means setting the control lines to the values in each stage for each instruction
- the simplest way is to extend the pipeline registers to include control info.
- we can create the control information during instruction decode

제어 신호 → 파이프라인 레지스터를 확장하여 제어 정보를 포함하도록한다

〈 Hazards 〉

Data Hazards

- The instructions of the previous example are independent
  - None of them used the results calculated by others
- let's look at the following code sequence :
  the last 4 instructions are all dependent on the result on the result in register x2 of the in1
  sub 이전 x2의 값이 10, sub 이후의 값이 -20이라 하자

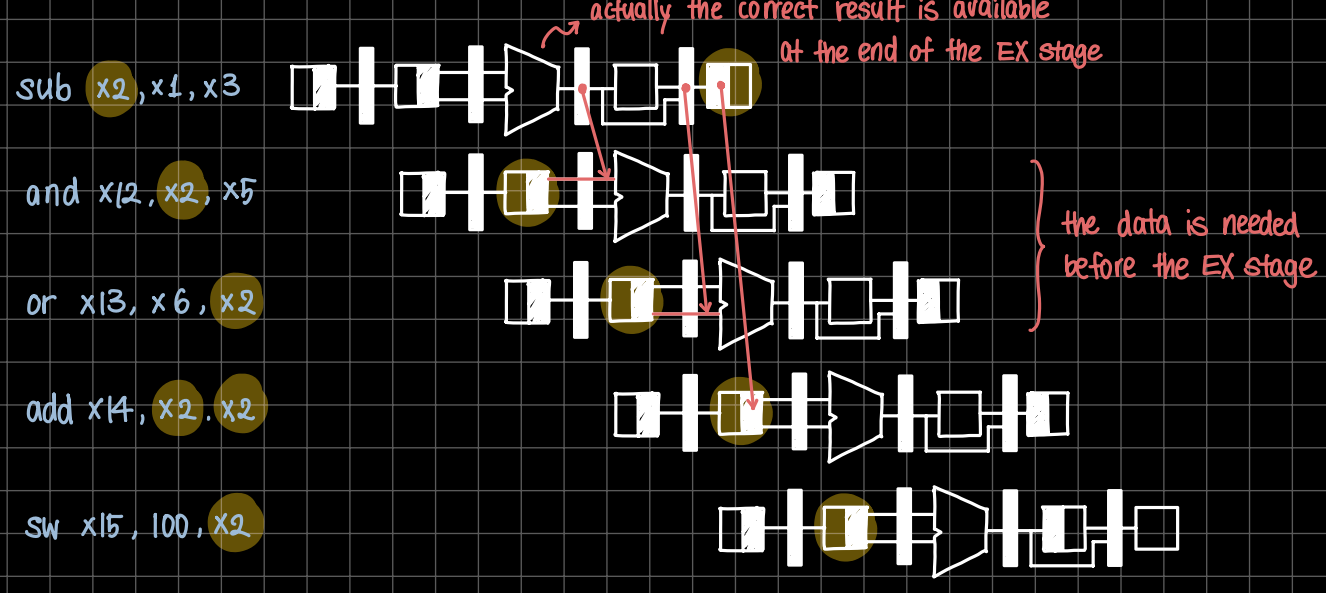


Solutions

- in 5 (sw) always reads the correct value of -20
- in 4 (add) can get a correct value
  → write 가 clock cycle 의 앞부분 절반
  read 가 clock cycle 의 뒷부분 절반이라서
  read 된 add 는 sub 이후의 값을 받을 수 있다
- in 2 / in 3 : forwarding is used

종속성 O
BUT 정상적으로 데이터 읽기 가능
fowarding x needed

- Questions : When is the data from the sub instruction actually produced?
  When is the data actually needed by the and/or instructions?



Data Hazard Detection

- Two pairs of data conditions are forwarding 필요한 경우!
  1a. EX/MEM.RegisterRd == ID/EX.RegisterRs1
  1b. EX/MEM.RegisterRd == ID/EX.RegisterRs2
  2a. MEM/WB.RegisterRd == ID/EX.RegisterRs1
  2b. MEM/WB.RegisterRd == ID/EX.RegisterRs2
  (write part) (read part)

* 레지스터의 데이터가 오경에 바뀌기 때문에 hazard 발생
  ⇒ Reg-write 신호가 활성화된 명령어
  write 없으면 읽어오는 값에도 변화가 없다
  ⇒ RISC-V에서 x0은 constant zero
  명령어 목적지가 x0이면 (ex. addi x0, x1, 2) forwarding 필요없다

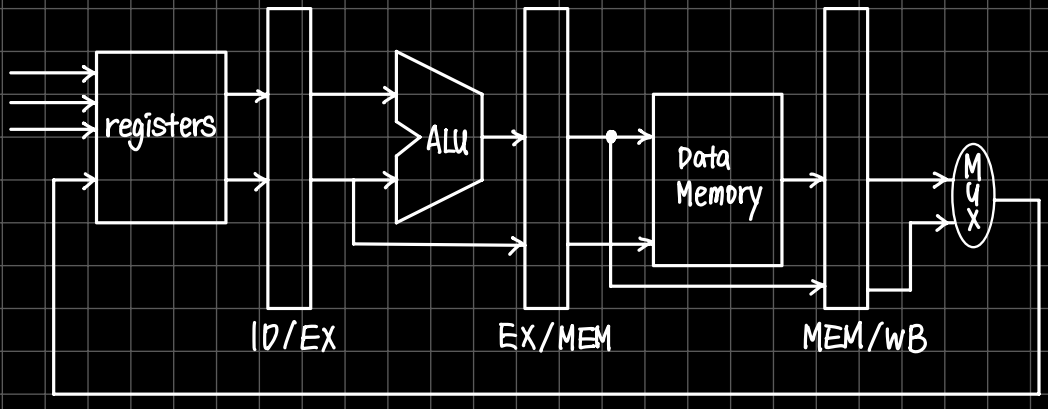
Further Conditions

- The previous conditions are not accurate
  - some instructions do not write registers
  - solution) Check to see if the RegWrite signal will be active
    (examine the WB control field of the pipeline register during the EX and MEM stages)
- The conditions work properly as long as we add…
  EX/MEM.RegisterRd≠0 to the first hazard condition, and
  MEM/WB.RegisterRd≠0 to the second
- What if destination register is x0 (= constant zero)?
  ⇒ not passing any data to next instruction

Forwarding

- If we can take the inputs to ALU from any pipeline register rather than just ID/EX,
  than we can forward the proper data
  by adding multiplexers to the input of the ALU and with the proper controls
- The forwarding control will be in the EX stage
  - The ALU forwarding multiplexers are found in that stage
  - We must pass the operand register numbers from ID stage via the ID/EX pipeline register

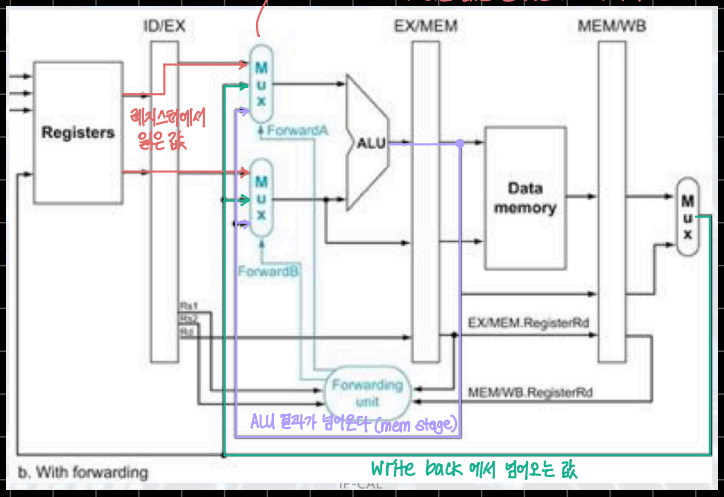
Without Forwarding



With Forwarding

★ Mux 의 역할
forwarding 된 값을 쓸 것인지
레지스터에서 읽은 값을 쓸 것인지 control

↝ 그냥 레지스터에서 읽기 (= forwarding 조건 미충족시)
mux 의 signal 이 모두 zero



b. With forwarding

EX/MEM.RegisterRd == ID/EX.RegisterRs1/2
↳ ALU 에서 넘어오는 값을 forwarding
MEM/WB.RegisterRd == ID/EX.RegisterRs1/2
↳ Write Back에서 넘어오는 값을 forwarding
Register Rd 가 x0 (constant zero) 면
multiplexer 의 signal 신경쓰지 x
forwarding 이 필요없다 (no data dependancy)

Control Values for Forwarding

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.

Forward □ = □□
- A: ALU의 1st 피연산자
- B: ALU의 2nd 피연산자
- 첫 □: MEM/WB 에서 가져온지 (1)
- 둘째 □: EX/MEM 에서 가져온지 (1)
- 0 이면 레지스터 (ID/EX) 에서 그대로 읽기

EX Hazard (Forwarded from EX/MEM)

if (EX/MEM.RegWrite && (EX/MEM.RegisterRd ≠ 0) && (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
⇒ forward A = 10

(if Rd is x0, no value stored)

RegWrite 의 control signal 을 확인해야한다
RegWrite = 0 이면 forwarding 이 필요없기 때문

if (EX/MEM.RegWrite && (EX/MEM.RegisterRd ≠ 0) && (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
⇒ forward B = 10

If the previous instruction is going to write to the register file
and the write register number matches the read register number of ALU inputs A or B,
the multiplexor picks the value from EX/MEM

MEM Hazard (Forwarded from MEM/WB)

if (MEM/WB.RegWrite && (MEM/WB.RegisterRd ≠ 0) && (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
⇒ forward A = 01

if (MEM/WB.RegWrite && (MEM/WB.RegisterRd ≠ 0) && (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
⇒ forward B = 01

One Complication: Potential data hazards between the result of the instruction in the WB stage
the instruction in the MEM stage, and the source operand of the instruction in the ALU stage

Double Data Hazard

- Consider the sequence :

```
add x1, x1, x2  (In 1)  (WB)
add x1, x1, x3  (In 2)  (MEM)
add x1, x1, x4  (In 3)  (EX)
```

In2의 계산결과가 필요한데, In1의 결과도 업데이트되는 hazard 발생 ⇒ 둘의 signal이 충돌!

⇒ revise MEM Hazard Condition : only forwarding if EX condition isn't true
쉽게 말해 In2의 destination 과 In3의 source를 비교하여 동일하지 않을 때만 In1 ⇝ In3 으로 forwarding

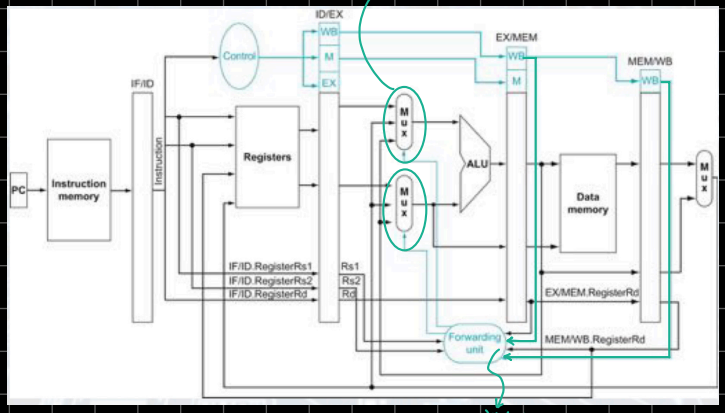
MEM Hazard (with Corrections)

If (MEM/WB.RegWrite && (MEM/WB.RegisterRd ≠ 0) &&
not (EX/MEM.RegWrite && (EX/MEM.RegisterRd ≠ 0) && (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
&& (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
⇒ forwardA = 01

If (MEM/WB.RegWrite && (MEM/WB.RegisterRd ≠ 0) &&
not (EX/MEM.RegWrite && (EX/MEM.RegisterRd ≠ 0) && (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
&& (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
⇒ forwardB = 01

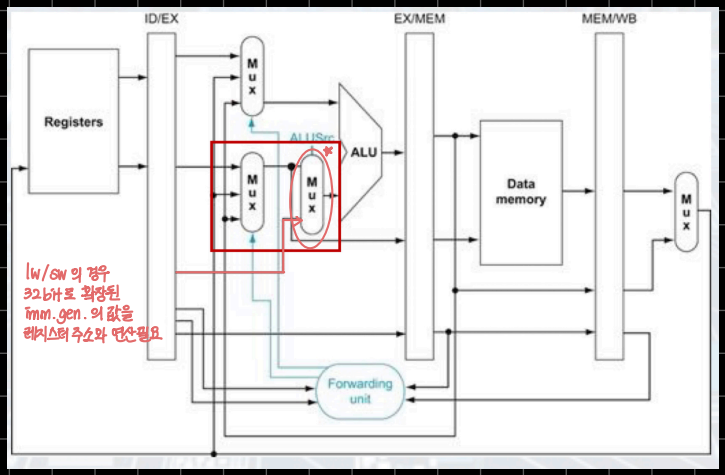
Datapath with Forwarding

Mux에서 forwarding input 3가지 ① 레지스터 ② MEM stage (ALU) ③ WB stage
세가지 중 어떤 것을 사용할지는 forwarding unit이 결정



해당 instruction이 실제로 write back을 하는지 확인하기 위해 가져온 것

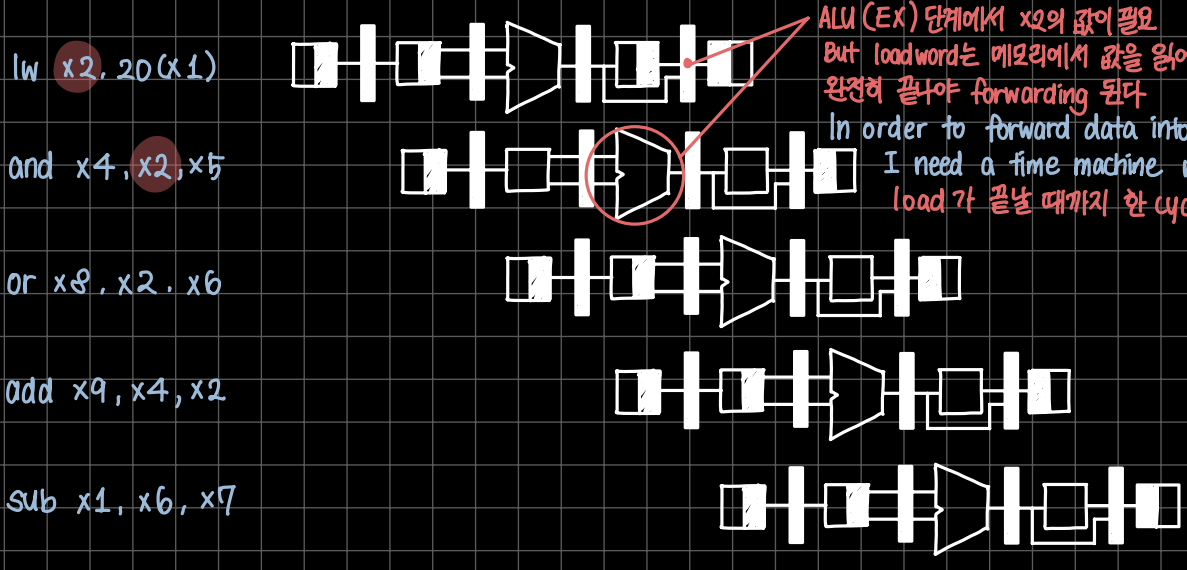
A Close-up of the Datapath



Data Hazards and Stalls

- We cannot avoid a pipeline stall when an instruction tries to read a register following a load instruction
  - The load instruction writes the same register
- Something must stall the pipeline for the combination of load followed by an instruction that reads its result

Load-Use Data Hazard



Load-Use Hazard Detection

- It Operates during the ID stage
  - it can insert the stall between the load and its use
- Condition : ↗ memory instruction 인지 확인

  If (ID/EX.MemRead && ((ID/EX.RegisterRd = IF/ID.RegisterRs1)
  || (ID/EX.RegisterRd = IF/ID.RegisterRs2)))

  ⇒ stall the pipeline
  - ↳ NOP 를 넣어준다 (= no operation)

Operations

- After this 1 cycle stall, the forwarding logic can handle dependency, and execution proceeds
- If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled
  - ↳ PC and IF/ID pipeline registers are preserved
- The back half of the pipeline starting with the EX executes nops

NOPs : NO operation

- At the hazard in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM, WB control fields of the ID/EX pipeline register to 0
  (No registers or memories are written if the control values are all 0)

Load-Use Data Hazard

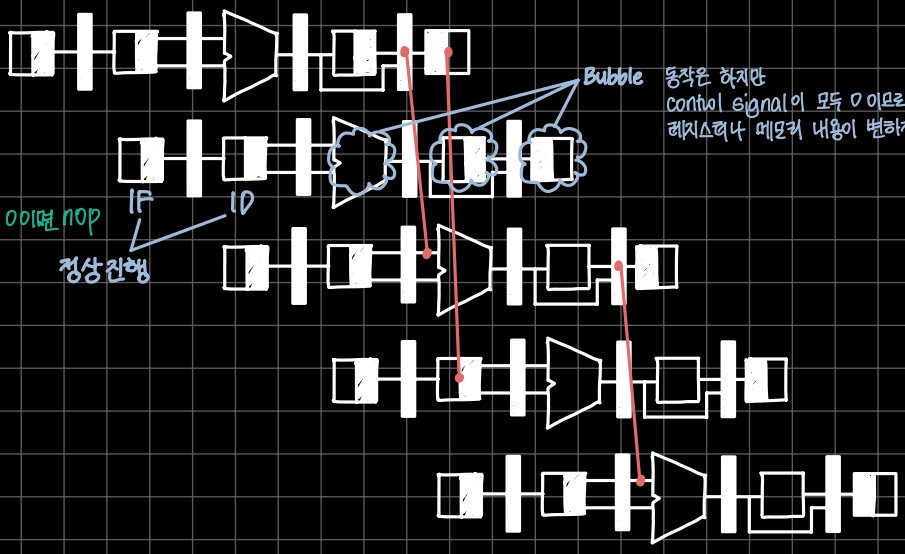
lw x2, 20(x1)

and becomes nop
bubble = 쓸데없는 instruction
7개의 control signal이 모두 0이면 nop

and x4, x2, x5

or x8, x2, x6

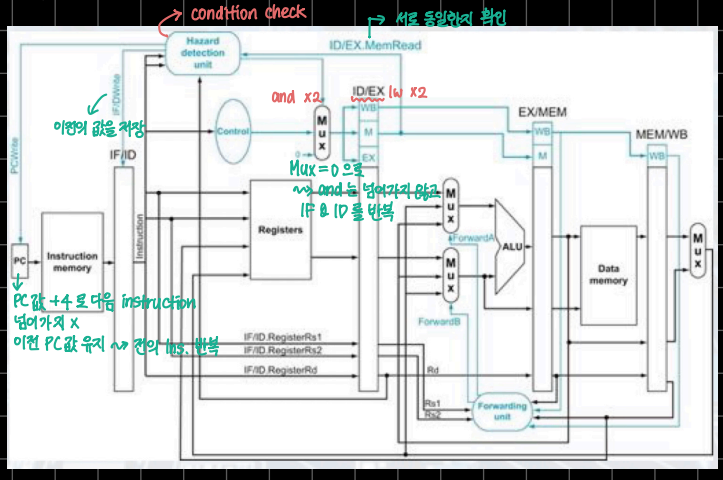
add x9, x4, x2



Stall : Observations → 결국 stall이란 같은 일을 한차례 더 반복하는 것

- The hazard forces the and/or instructions to repeat in CC4 what they did in CC3
  - and reads registers and decodes again (ID)
  - or is re-fetched from instruction memory (IF)
- Such repeated work is what a stall looks like:
  - Its effect is to stretch the time of the and & or instructions and delay the fetch of the add instruction
  - a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each cycle until it exists at the end

Datapath with Hazard Detection



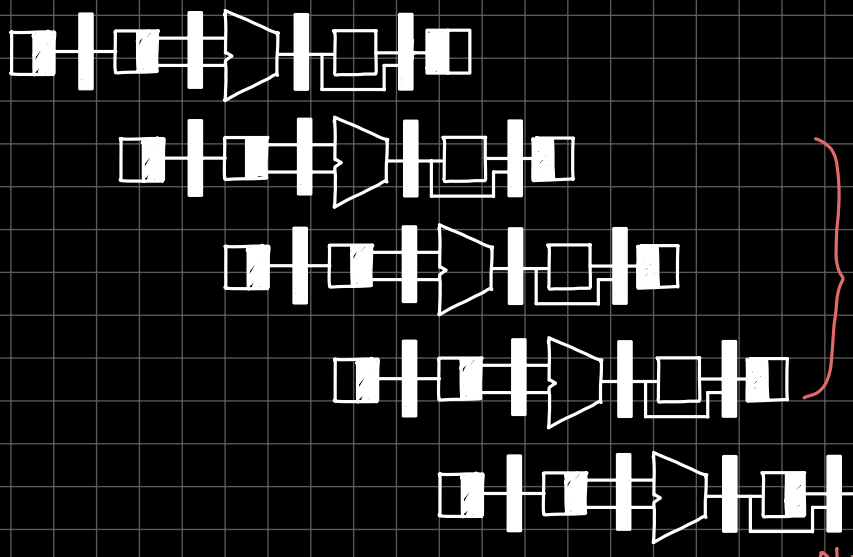
Stalls and Performance

- Stalls reduce performance (but are required to get correct results)
  성능 하락은 필연적
- Compiler can arrange code to avoid hazards and stalls
  컴파일러는 stall를 없애기 위해 코드의 순서를 편집

Control Hazard

- If branch outcome determined in MEM

```
40 beq x1,x0,x16
44 and x12,x2,x5
48 or  x13,x6,x2
52 add x14,x2,x2
72 lw  x4,100(x7)
```



이미 실행 중인데 branch 여부가 MEM 단계에서 결정 branch taken 이면 3개는 flush

이 시점에서 throughput은 5가 아니라 5-3 = 2

Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - target address adder
  - Register comparator
- Example : branch taken

branch not taken 이면 PC = 40+4

```
36 : sub x10, x4, x8
40 : beq x1, x3, 16   // PC relative branch (to 40+16*2 = 72)
44 : and x12, x2, x5
48 : or  x13, x2, x6
       :
72 : lw  x4, 50(x7)
```

원래는 이름이 쓰여 있지만 어셈블러가 relative address로 바꿔준다

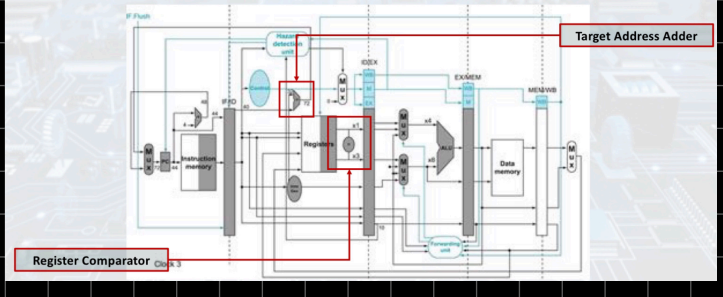
Target Address

shift left

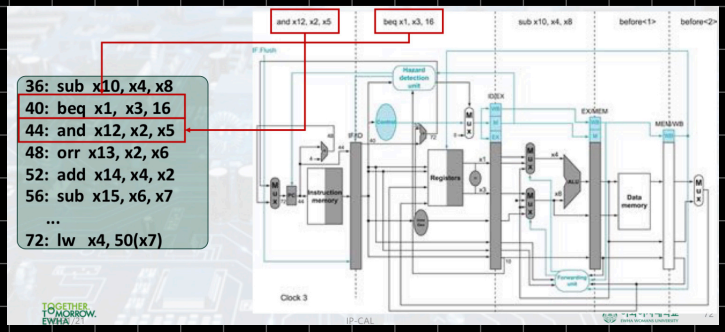
왜 x4가 아니라 x2지??
⇒ 분기명령어는 half-word 단위다
헷갈리게 하고 자료 개수이 세키들이

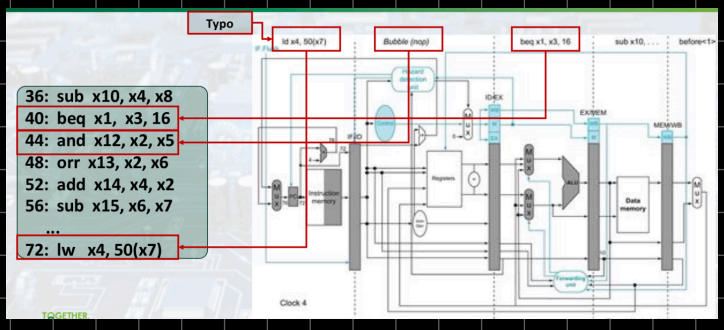
Datapath For Control Hazard

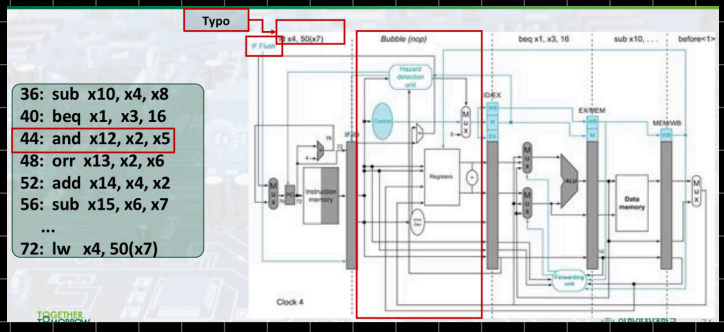
idea : branch를 mem 까지 기다리지 않고 decode stage로 가져오겠다



Example : Branch taken (예측이 틀렸을 경우)



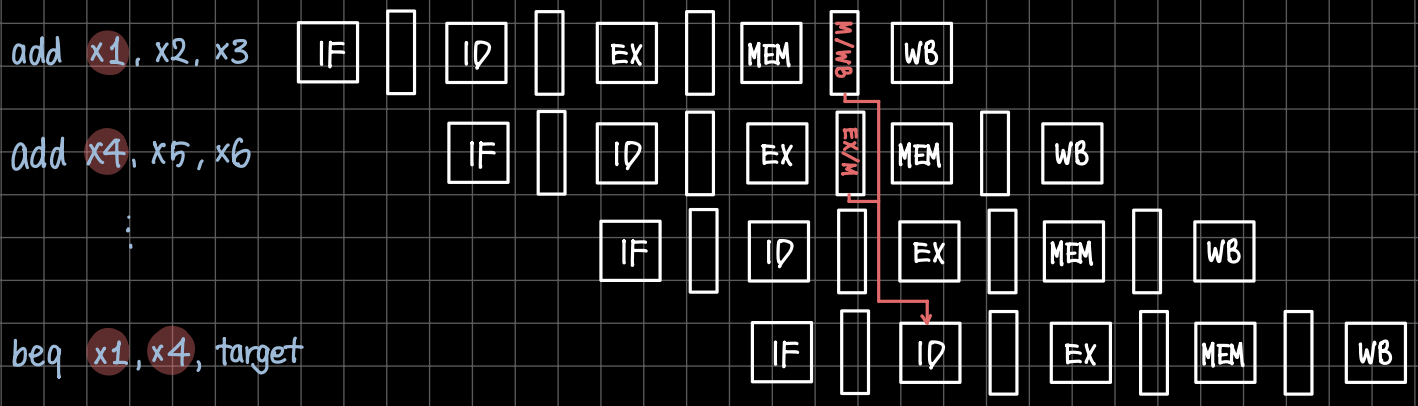




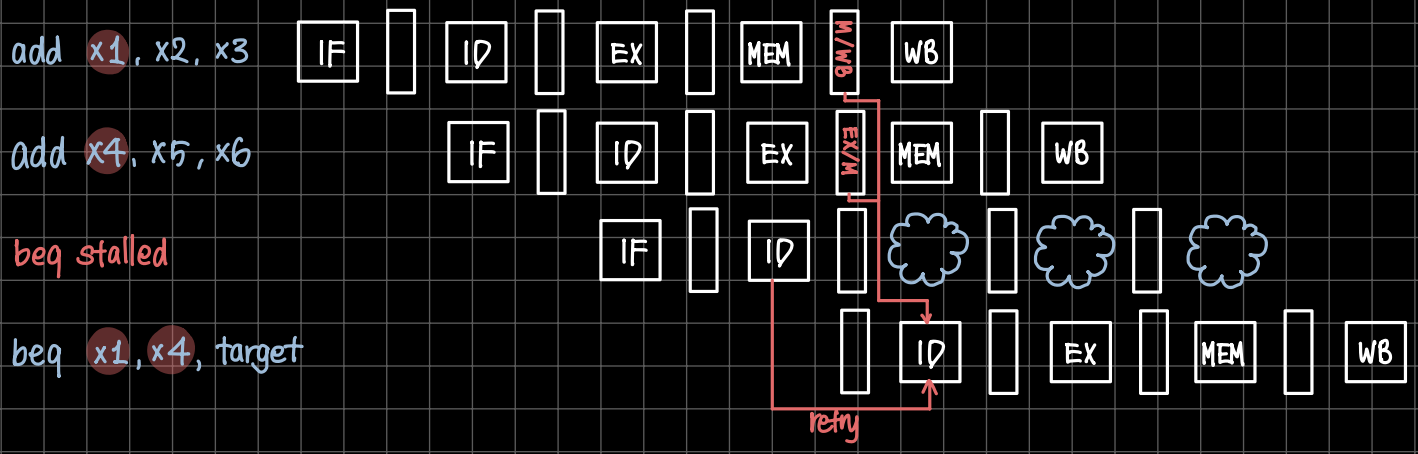
✱ 예측이 틀리면 버블이 한 번은 반드시 발생하게 된다
예측이 맞으면 버블없이 그대로 진행

Data Hazards for Branches

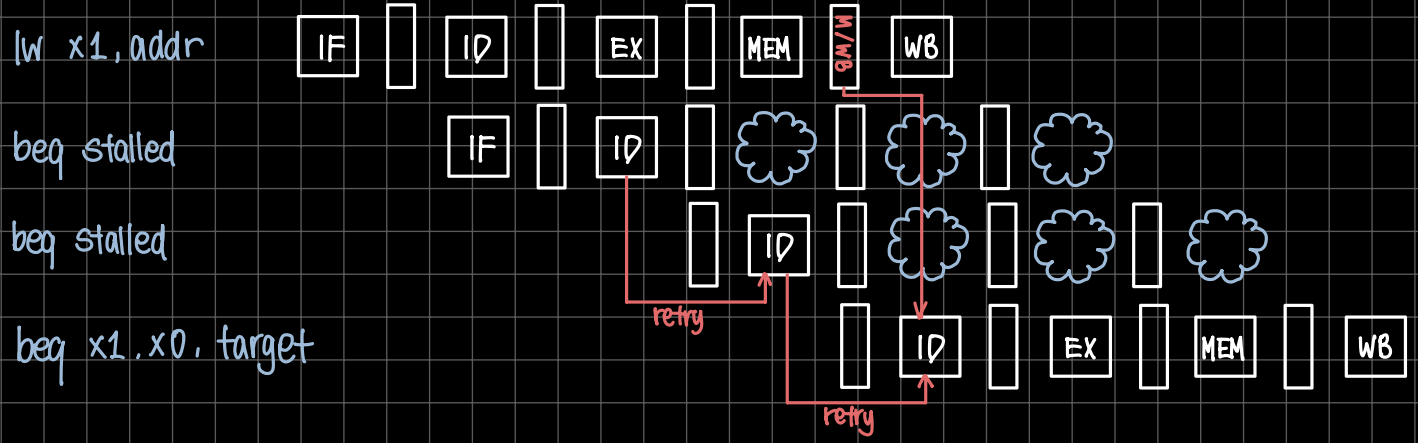
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  ⇒ need 1 stall cycle
  바로 뒤에 branch 올 경우 1cycle bubble 발생 R type + beq → 1 stall (최소한 EX가 끝나야!!!)

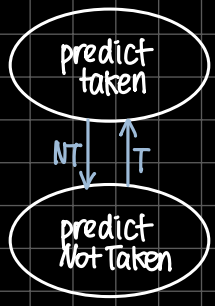


- If a comparison register is a destination of immediately preceding load instruction
  ⇒ need 2 stall cycles lw + beq → 2 stalls (메모리에서 원하는 값 찾는 과정 필수 — MEM 끝날 때까지)



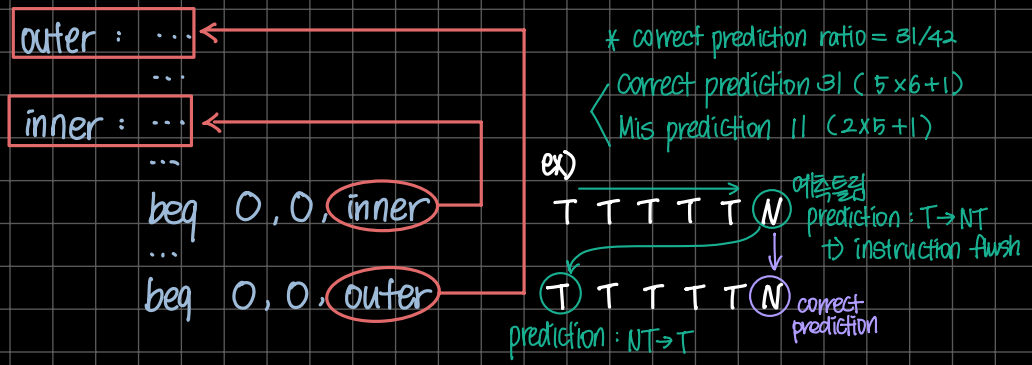
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken / not taken)
  - to execute branch
    ① check table, expect the same outcome
    ② start fetching from fall-through or target
    ③ If wrong, flush pipeline and flip prediction



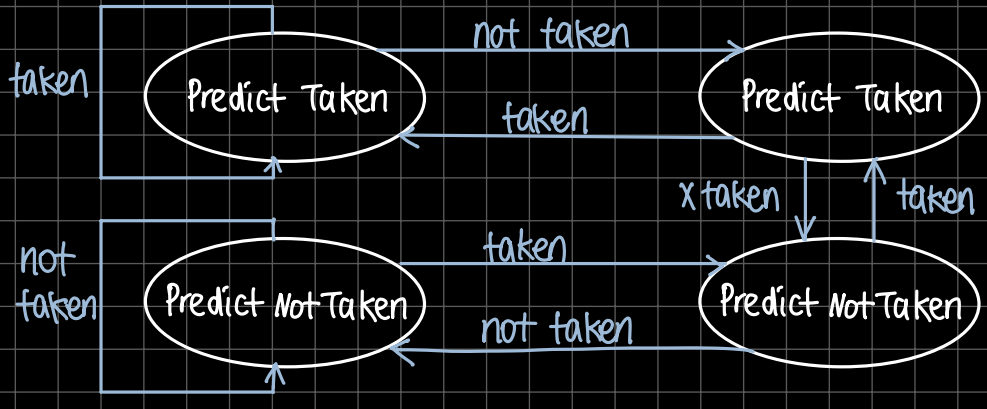
1-Bit Predictor : Shortcoming 한 번만 예측과 어긋나도 flip prediction

- Inner loop branches mispredicted twice!
  ① mispredict as taken on last iteration of inner loop
  ② mispredict as not taken on first iteration of inner loop next time around



2-bit Predictor 예측과 두번 어긋날 경우 flip prediction

- Only change prediction on two successive mispredictions



Calculating the Branch target

- Even with predictor, still need to calculate the target address
  → 1 cycle penalty for a taken branch
- Branch target buffer
  → cache of target address
  → Indexed PC when instruction fetched
  If hit and instruction is branch predicted taken,
  can fetch target immediately