

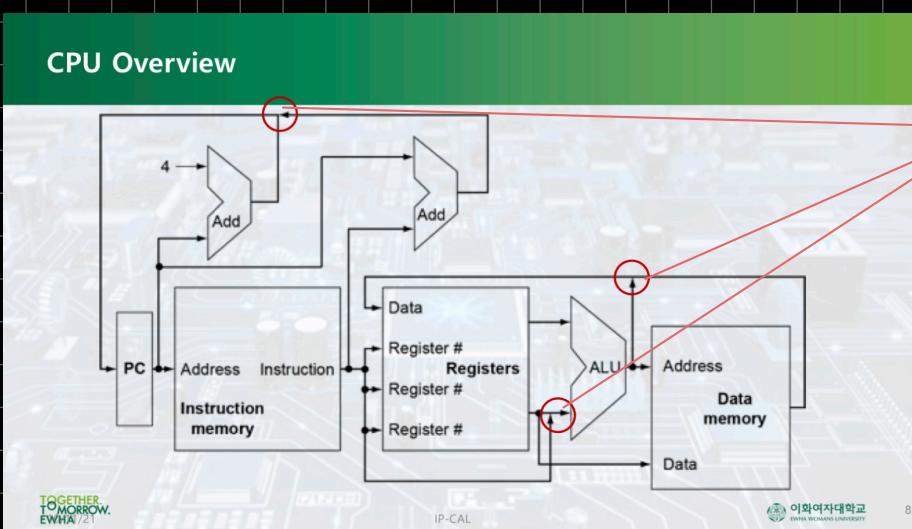
Introduction

- CPU performance factors
 - Instruction Count → determined by ISA, Compiler
 - CPI, Cycle Time → determined by CPU Hardware
- We will examine two RISC-V implementations
 - ① a simplified ver.
 - ② a more realistic pipelined ver.
- Simple subset, shows most aspects
 - memory reference : lw, sw
 - Arithmetic / logical : add, sub, and, or
 - Control transfer : beq

이 단원에선 이 7가지 핵심 명령어만 다룬 예정

Instruction Execution

- 모든 명령어의 첫 두단계는 동일하다
 - ① Program Counter 를 프로그램이 저장되어 있는 메모리에 보내어 메모리로부터 instruction 을 가져온다
 - ② 읽을 레지스터를 선택하는 instruction field 를 사용하여 하나 or 두개의 레지스터 읽는다
 - load word (lw) → 하나의 레지스터 읽기
 - 대부분은 두 개의 레지스터를 필요로 한다
- After two Steps
 - 두 단계 이후에 명령어 실행을 끝내기 위한 행동들은 명령어 종류에 따라 달라진다 각 종류 (① memory reference ② arithmetic / logical ③ control transfer)
내에서는 명령어가 무엇인지와 관계없이 필요한 행동들이 대부분 같다.
even across different instruction classes there are some similarities
 - 모든 명령어 집합이 레지스터를 읽은 뒤에는 ALU를 사용
 - ALU 사용 후 ① memory reference : 메모리를 읽거나 (lw) 저장하기 위해 (sw) 메모리에 접근한다
② arithmetic / logic : ALU의 데이터를 레지스터에 써야 한다
③ control transfer (branch) : 비교 결과에 따라 다음 명령어의 주소를 바꾸거나
PC 값을 4만큼 증가시켜 바로 다음 명령어의 주소를 갖게 한다



여러개의 정보 중 하나를 선택하여 그것을 목적지로 보내는 역할 필요

↓
multiplexor
= data selector

Two Important Aspects Omitted

① Multiplexor (Data Selector)

서로 다른 지점에서 나온 데이터를 처리하여 하나의 목적지로 보내준다

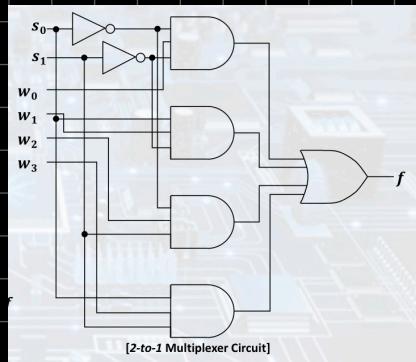
② 유닛들은 명령이 종류에 따라 다르게 제어된다

Mux : Multiplexer (4-to-1)

- Data input : 4
- Selection input : 2
- Output : 1

S_0	S_1	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

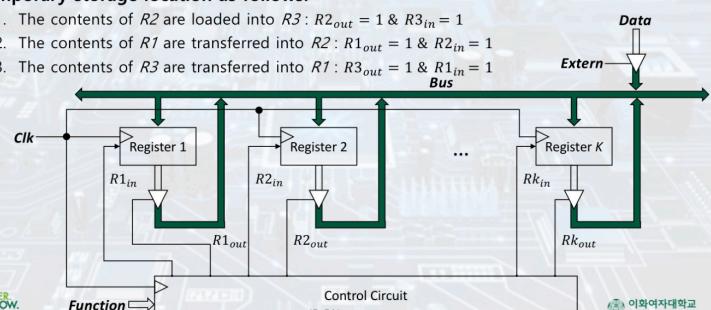
} Signal bit (S_0, S_1) 조합에 따라 출력값이 $w_0 \sim w_3$ 중에서 결정된다



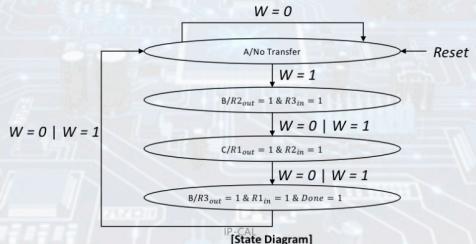
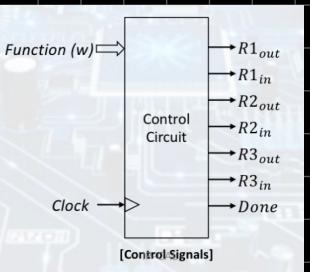
Design Control Example (Digital Logic Class)

- The contents of registers R1 and R2 can be swapped using register R3 as a temporary storage location as follows:

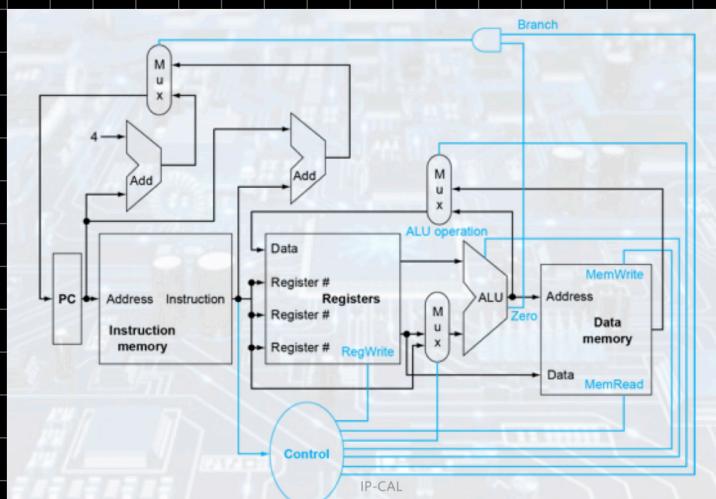
1. The contents of R2 are loaded into R3: $R2_{out} = 1 \& R3_{in} = 1$
2. The contents of R1 are transferred into R2: $R1_{out} = 1 \& R2_{in} = 1$
3. The contents of R3 are transferred into R1: $R3_{out} = 1 \& R1_{in} = 1$



레지스터 = 여러개의 flip flop (= 1 bit) \Rightarrow RISC-V : 32bit R. \times 32H
32 flip flops



→ with Multiplexor and Control Lines



Logic Design Basics

- Information encoded in Binary
 - low voltage = 0, high voltage = 1 only two values
 - one wire per bit
 - multi bit data encoded on multi wire buses
- Combinational Element
 - Operate on Data
 - Output is a function of input
- State (sequential) elements – Store information

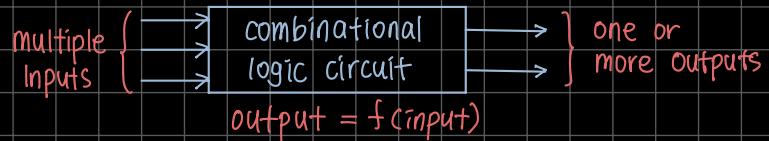
논리소자 ①

Combinational Elements 조합소자 : 데이터 값에만 동작하는 소자

- Multiplexer
- De-Multiplexer
- Arithmetic / Logic Unit (ALU)

입력 \rightarrow 논리회로 내부의 연산자들을 통해 출력,

내부기의 소자 \times 출력이 현재의 입력에만 의존



논리소자 ②

Sequential Elements 상태소자 : 상태를 포함하는 소자

- The values of the outputs depend not only on the present values of the inputs but also on the past behavior of the circuit
- 현재의 입력값 + 이전 값의 상태에 따라 출력값이 결정
- 기억장치 (memory) 가 있어 이전 출력값 보관 가능

(cf) memory 의 구분

* 래치 (latch) : level sensitive device

메모리를 활성화시켜주는 신호가 High OR Low로 고정

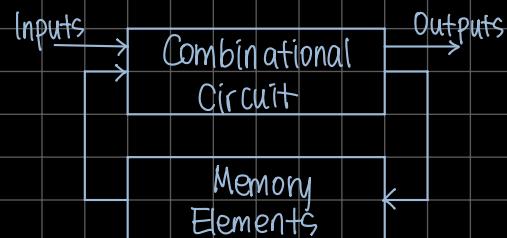
메모리 (활성화 : 입력신호가 그대로 출력신호로 나온다)

비활성화 : 마지막 상태를 유지

* 플립플롭 (flip-flop) : edge sensitive device

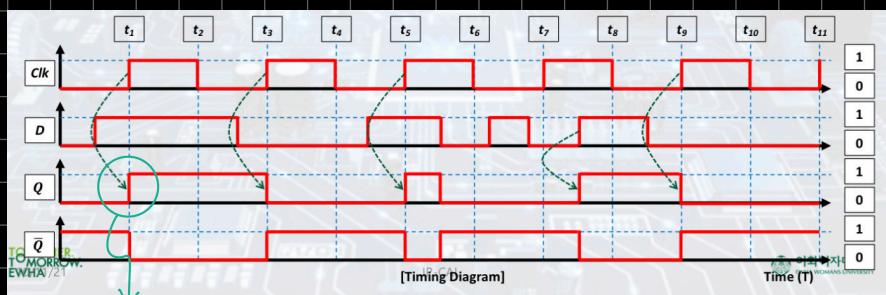
외부 Clk 신호가 rising edge OR falling edge 일 때 신호가 변한다

그 외의 구간에서는 이전 값을 유지한다



순차회로 (sequential circuit)의 출력은
입력값 뿐만 아니라 내부상태 (state)에도 의존

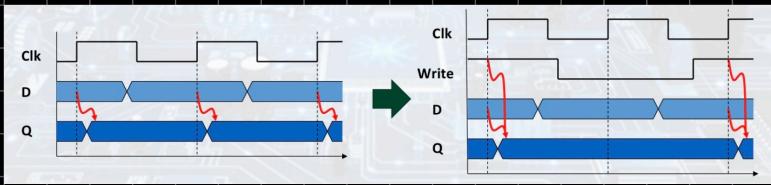
- The content of the storage elements are said to represent the state of the circuit
- Register : stores data in a circuit
 - uses a clock signal to determine when to update the stored value
 - Edge-triggered : update when Clk changes from 0 to 1



Clk 신호가 0에서 1로 변할 때만 업데이트되고
그 외의 경우에는 값을 그대로 유지한다

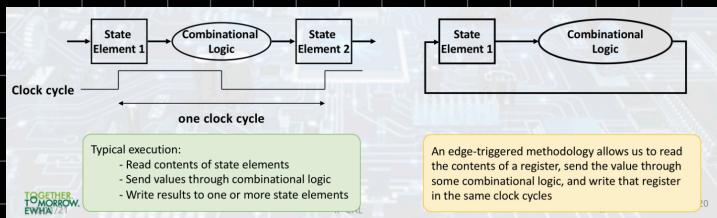
Register with write control

- Only updates on clock edge when write control input = 1
- Used when stored value is required later



클럭킹 방법론 Clocking Methodology

신호를 언제 읽고 언제 쓸 수 있는지를 정의한다
예측 가능성을 보장하기 위해 설계되었다



Edge - Triggered Clocking 예지 구동 클럭킹 방법론

→ 모든 상태 변화가 클럭에지에서만 일어난다

→ 입력 = 이전 클럭 사이클에서 쓴 값

출력 = 다음 클럭 사이클에서 사용할 수 있는 값

- Combinational logic transforms data during clock cycles
 - btw clock edges → 입/출력 모두 상태소자
 - Input from state elements, output to state elements
 - longest delay determines clock period

Building a Datapath

- Datapath : Elements that process data and addresses in the CPU

ex) Registers, ALUs, MUX's, memories ...

프로세서 안에서 데이터를 가지고 옮간하거나 데이터를 저장하는 기본 유닛

Fetching Instruction - 명령어 읽어오기

- Two state elements 상태소자 are needed to store and access instructions

① Instruction memory :

프로그램의 명령어 저장, 주소가 주어지면 해당 명령어를 보내주는 메모리 유닛

② Program Counter (PC) :

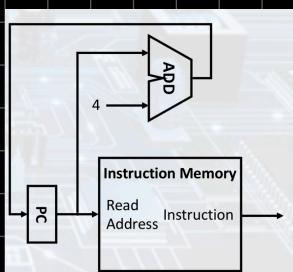
- 현재 명령어의 주소를 갖고 있는 special register

the register containing the address of the instruction in the program being executed

- 다음 명령어의 실행을 위해 다음 명령어를 가리키도록 주소 $\text{PC} + 4$ 필요하다 \rightsquigarrow Adder의 역할

Incremented by 4 to prepare for executing the next instruction

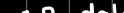
③ Adder (항상 업샘플 하는 ALU라고 생각하면 된다)



R-Format Instructions – 산술 / 논리 명령어

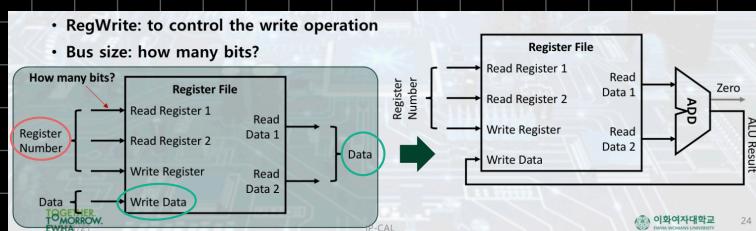
- add, sub, and, or
 - Register File : 접근할 레지스터 번호를 지정함으로써 읽고 쓸 수 있는 레지스터들의 집합으로 구성된 상태소자
파일 내의 레지스터 번호를 지정하면 어느 레지스터라도 읽고 쓸 수 있다
 - A state element consisting of a set of registers
 - The registers can be read and written by supplying a register number

- R-Format instructions

- three register operands 
 - 2 data words read from the register file
 - 1 data word written into the register file
 - How many I/O needed? \Rightarrow 4 Input, 2 Output

R-Format : ALU Operations

- 4 Inputs 2 Outputs
 - 레이스터에서 데이타 워드를 읽기 위해서는 레이스터의 입력과 출력이 하나씩 필요
읽을 레이스터 번호를 지정하는 입력 + 레이스터에서 읽은 값을 내보내는 출력
 - 데이타 워드를 쓰기 위해서는 입력이 2개 필요하다
레이스터 번호 지정 입력 + 레이스터에 쓸 데이타값을 제공하는 입력

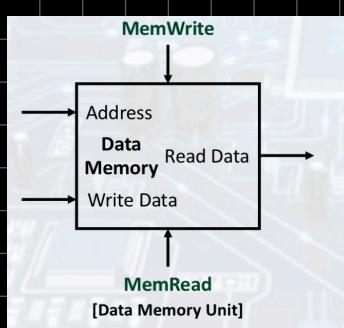


리지스터 넘버 : 리지스터 32개 \rightarrow 5bit로 주소값을 모두 표현할 수 있다

데이터 입력/출력 : 32 bit 플을 가진다

Load / Store Instructions

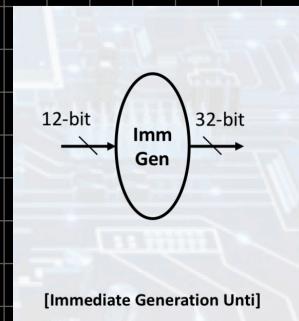
- General form
 - lw \$x1, offset_value(\$x2) 베이스 레지스터
 - sw \$x1, offset_value(\$x2)
 - Load and Store operations involve
 - (store value - read from register file during decode , written to data memory
 - load value - read from data memory , written to the register file



Mem Read : Data Memory의 값을 Read Data 출력으로 내보낸다 \rightarrow load word
Mem Write : Data Memory의 값을 Write Data 입력값으로 바꾼다 \rightarrow store word

Immediate Generation Unit

- We will need a unit to sign-extend the 12-bit offset field in the instruction to a 32-bit signed value (including shift left 1)
 - Imm Gen has 32 bit instruction as input that selects 12 bit field for load, store, branch if equal that is sign extended into a 32 bit result appearing on the output
- [2비트 → 32비트로 부호확장]

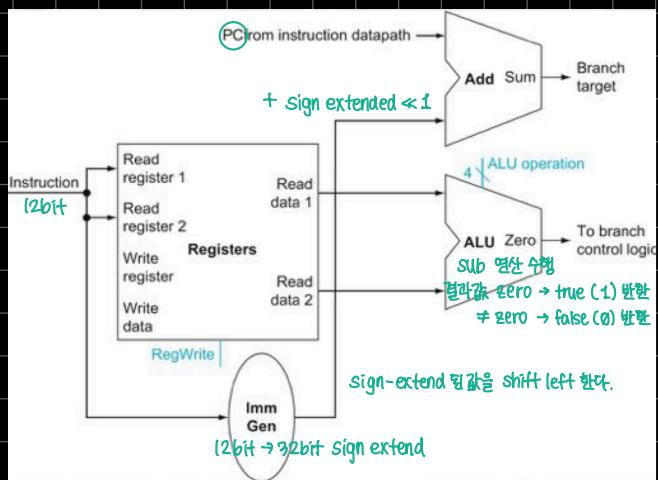


Bit shift left 1??
shift left 1 경우면 x2와 같다
Imm은 Half word 단위!! 1 word = 4byte, half word = 2byte

Branch Implementation

- read register operands
- compare operands - use ALU. subtract and check zero output
- calculate Target Address
 - It becomes the new program Counter (PC) if the branch is taken
 - the sum of the offset field of the instruction & the address of the instruction -(sign-extended offset field $\ll 1$) + PC
 - Shift left : x2 와 같다
- The base for the branch is the address of the instruction
- The offset field is shifted left 1 bit so that it is a half-word offset
 - \rightarrow it increases range PC-relative addressing에서 상수는 half-word (2byte = 4bit) 단위
 - 상수에 shiftLeft 1로 x2를 해주어 Byte 단위로 변환한다
- When the condition is
 - True : branch is taken - the branch target address becomes the new PC
 - False : branch is not taken - the incremented PC should replace the current PC
 - \hookrightarrow 1단계 +4로 바로 다음줄에 있는 명령 수행

Data Path for Branch Instructions

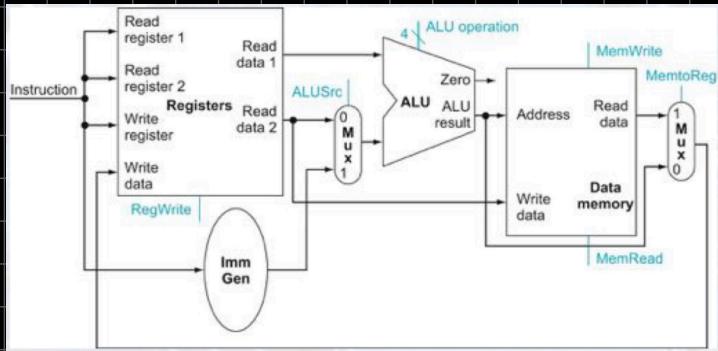


- Thus, the branch data path must do two operations
 - compute the branch target address
 - compare the register contents

Creating a Single Datapath 단일 데이터 패스 만들기

- 가장 간단한 데이타 패스는 모든 명령어를 한 클럭 사이클에 실행하는 것
 - = 데이타 패스 자원을 명령어 당 한번만 사용할 수 있다
 - 따라서 데이타 메모리와는 별도로 명령어 메모리가 필요하다
 - Each datapath element can only do one function at a time
Hence, we need separate instruction & data memories
 - 즉, 두 번 사용하는 요소는 여러개 두어야 한다
그러면서 데이타 메모리 / 명령어 메모리 둘다 필요

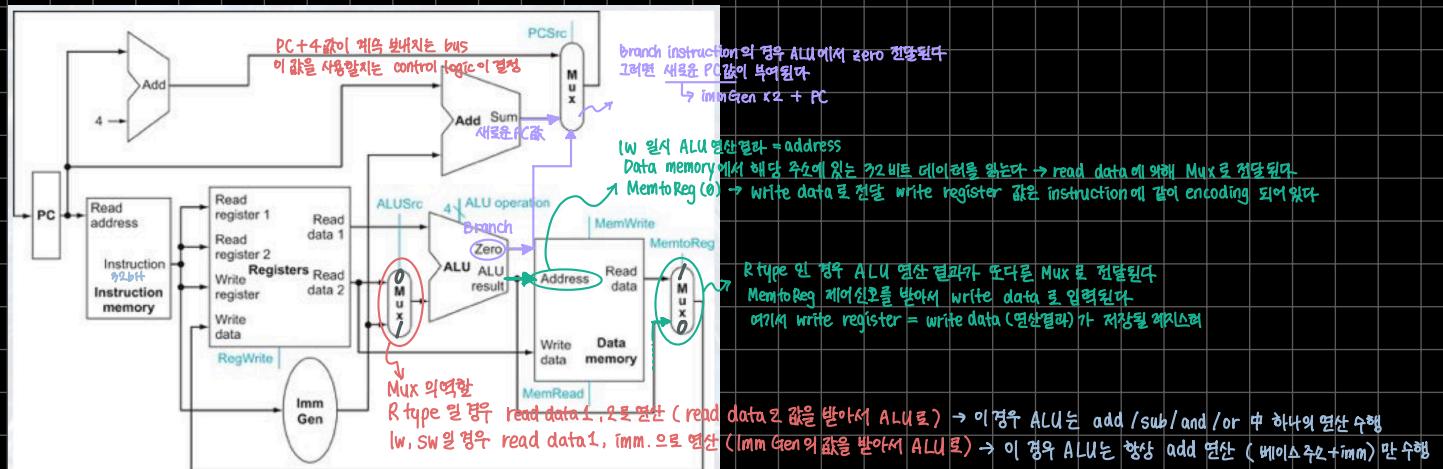
R type / Load / Store Datapath



R-type의 경우 레지스터 2개에서 값을 입력 받아 ALU에서 연산
 메모리 (lw/sw) 명령에서는 레지스터 1개 + 12bit 범위 필드를 주소 확장한 값을 받아
 ALU에서 주소를 계산한다.

Rtype은 레지스터에 저장할 값을 ALU에서
|W작래명령어는 레지스터에 저장할 값을 메모리에서 가져온다

Full Datapath



PC에 저장된 address를 Instruction 메모리가 읽고
해석되는 Instruction(32bit)를 load

- R type 의 경우 : read register 1, 2에 각각 5비트씩 레지스터의 주소를 전달

How to use ALU?

- The ALU Control : our ALU has four control inputs
제어입력 4개를 사용하는 다음 4개 조합을 정의하고 있다

ALU Control Lines (ALU 제어선)	Function 기능
0000	AND
0001	OR
0010	ADD
0110	SUB

ALU control - 4bit

[for ld and sd : ADD]

[for beq : SUB]

[for R-type : depends on the opcode]

→ funct7 + funct3

- function field + 2bit control field (= ALUOp) 를 입력으로 하는 small control unit을 통해 4bit ALU 제어신호를 발생시킨다

- ALUOp [00: ADD for lw/sw]

- [01: SUB for beq]

[10: Determined by the operation encoded in funct7 and funct3]

- The 4 bit output directly controls the ALU (ALU Control)

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

} lw, sw, beq의 경우
opcode 단위로도 ALU Control Input이 확정된다

→ R-type 의 경우 funct7, funct3 field 까지 봐줘야 한다

Multiple Levels of Decoding

- How to generate ALUOp?

⇒ 단계별 Multiple Levels of Decoding

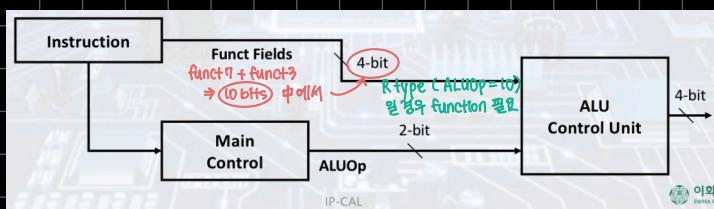
- main control unit의 크기 ↓

- 여러 개의 작은 제어유닛 사용하면 제어유닛의 속도 ↑

The main control units 주제어유닛이 ALUOp bits 를 생성

→ ALUOp are used as input to the small control unit = ALU Control Unit

→ The small control unit generates the signal ALU Control



Truth Table for ALU Control Bits

- Control signals derived from instruction

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

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

→ funct7, funct3 총 10 bit
그중 4bit 만 이용해도 구분 가능

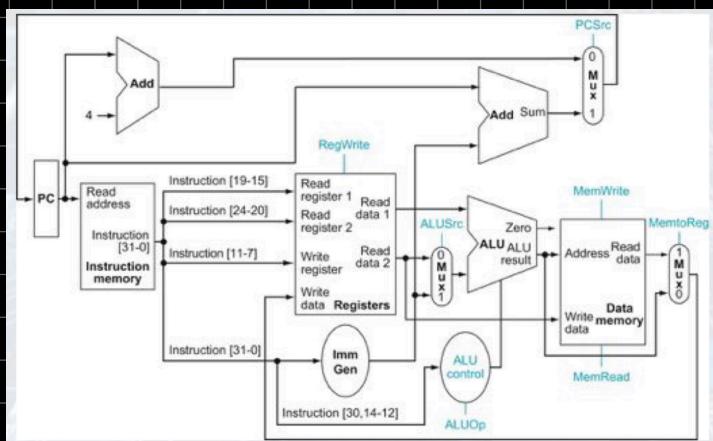
ALU Op는 opcode를 인풋으로 하는 main control에서 생성된다

ALU Control은 ALUOp + funct-field를 인풋으로 하여 생성된다

* 핵심 • opcode 통해 type을 판단한다

• funct7, funct3 통해 어떤 명령어인지 판단할 수 있다

Designing the Control Unit



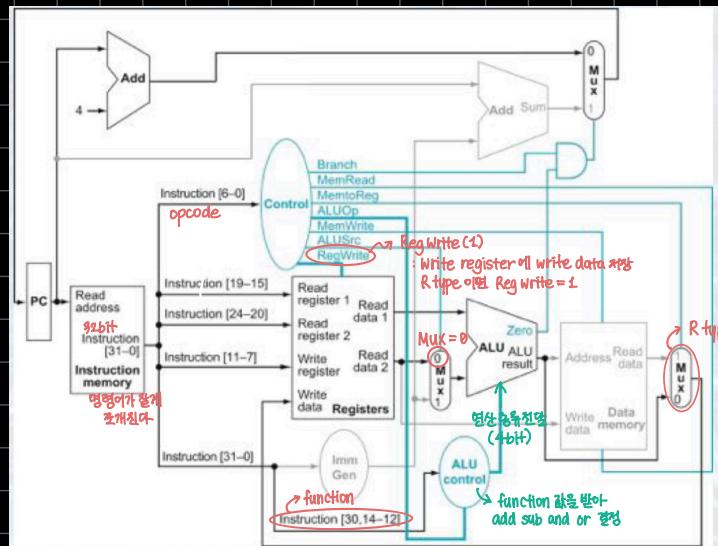
* 제어 시스템의 기능 → opcode에 의해 완전히 결정된다

deasserted (안가져온다)

asserted (가져온다)

RegWrite	x	Write Data → Write Register
ALUSrc	Read Data2가 ALU의 피연산자	복합 확장된 imm.이 ALU의 피연산자
PCSrc	new PC value : PC+4	new PC value : 불변 및 증가 주소
MemRead	x	Address에서 지정한 메모리 → Read Data
MemWrite	x	Address에서 지정한 메모리 → Write Data
MemtoReg	ALU의 출력이 Write data의 입력	[Data Memory] → Write data

R-type Instruction : example [add x1 x2 x3]



1. 명령어를 Instruction Memory에서 가져오고 PC값 증가(+4)

2. 두 레지스터(x2, x3)를 레지스터 파일로 부터 읽어온다

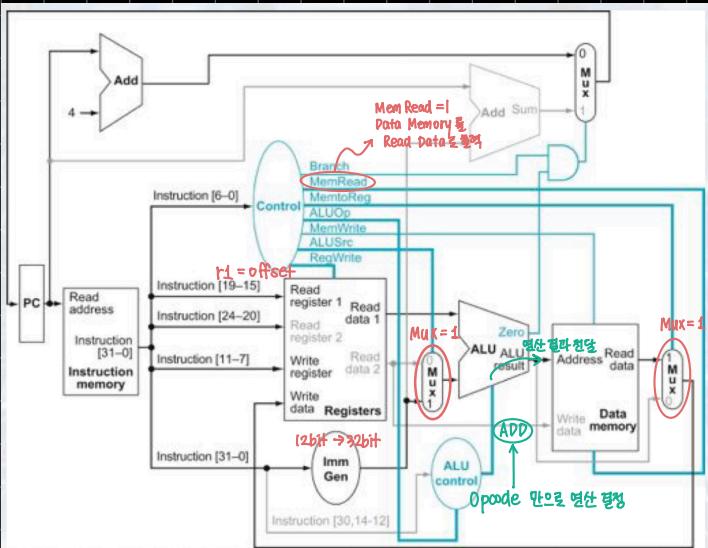
main control unit은 제어 시스템의 값들을 계산한다

3. 레지스터 파일에서 읽은 값들에 대한 ALU의 연산 function 부분을 이용해 ALU 제어 신호 생성

4. 결과값이 레지스터 파일의 목적지(x1)에 쓰여진다

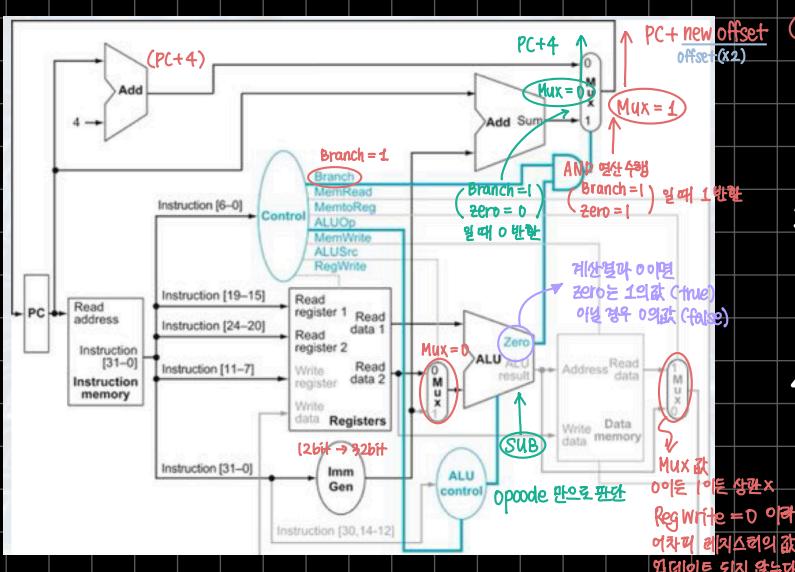
R-type 일 경우 항상 MemtoReg(0) 유지
ALU의 출력이 레지스터의 Write data 입력이 된다

Load Instruction : example [lw x1 offset(x2)]



1. 명령어를 Instruction Memory에서 가져오고 PC 값을 증가시킨다 (+4)
2. 레지스터 파일에서 레지스터 (x_2)의 값을 읽는다
3. ALU의 연산) 레지스터 파일에서 읽어들인 값과 명령어의 12비트 (offset)를 부호확장한 값의 합을 구한다
4. 이 합을 Data Memory 접근을 위한 주소로 이용
5. Memory unit에서 가져온 데이터를 레지스터 파일 (x_1)에 기록

Branch-on-Equal Instruction : example [beq x1 x2 offset]



1. 명령어를 Instruction Memory에서 가져오고 PC 값을 증가시킨다 (+4)
2. 두 개의 레지스터 (x_1, x_2)를 레지스터 파일로부터 읽는다
3. ALU는 두 값을 이용해 SUB 연산수행 명령어의 12비트 (offset) 부호확장 후 Shift left 한뒤 PC 값에 더해준다
→ 결과값이 분기 목적지의 주소
4. 어떤 Adder의 결과를 PC에 저장할지 ALU의 zero 출력을 이용하여 판단
→ Step 3의 계산 결과
= zero : sign extended, leftshifted offset + PC
≠ zero : PC+4 이용

Finalizing Control

Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	1
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	r2	imm	0
	MemtoReg	0	ALU	datamem	X
	RegWrite	1	1	0	0
Memory	MemRead	0	1	0	0
접근하는 Iw, sw만	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Reg Write의 경우
MemtoReg이 Don't Care

ALUSrc = 0 : Read data 2가 ALU의 피연산자
= 1 : 부호확장된 ImmGen 값이 ALU의 피연산자
MemtoReg = 0 : ALU 출력 → Write data의 입력 (R-type)
= 1 : Data Memory 출력 → Write data 입력 (lw)

Performance Issues

- Longest delay determines clock period
 - load instruction이 결정 (가장 오래걸린다)
 - clock은 가장 긴 instruction을 기준으로 결정되므로
- Instruction memory → register file → ALU → data memory → register file

R type 0 0 0 X 0

Beq 0 0 0 X X

SW/LW 0 0 0 0 0 ~ load Instruction 만 모두 단계 필요

- Not feasible to vary period for different instructions
 - Violates design principle "Make common case fast"
- ⇒ pipelining을 통해 개선

실제 instruction 수행시간이 줄어드는 것은X
throughput이 증가한다

Pipelining Analogy

- pipelined laundry - overlapping execution : parallelism improves performance

빨래는 4단계, 각단계에 30분씩 소요된다고 가정 → 총 2시간

\langle sequential laundry → 4 loads에 8시간 필요
pipelined laundry → 4 loads에 3.5시간 필요) $\frac{8}{3.5} \div 2.3$ 배 Speed up !!

- Non stop (load = ∞)

→ 성능향상 $2n / 0.5n + 1.5 \approx 4$ → 이상적인 조건하에서 많은 명령어가 있을 경우
파이프라이닝에 의한 속도 향상 = 파이프 단계수
하지만 어쩔 수 없이 sequential하게 작동하는 부분

Steps in Executing RISC-V

- 5 stages, one step per stage

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

Pipeline Performance

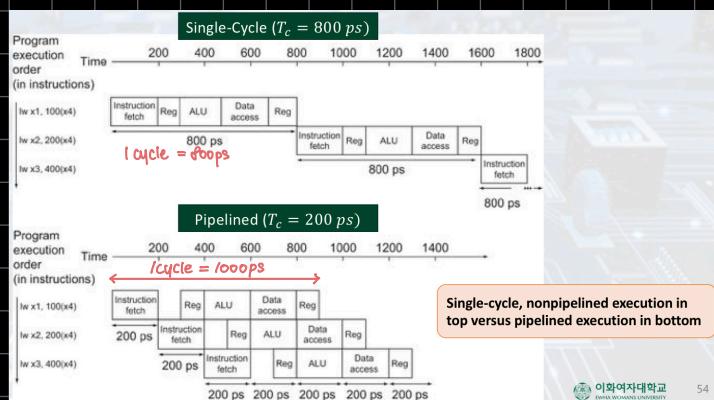
- Assumptions

200ps - memory access

200ps - ALU Operation

100ps - register file read or write

Instruction Class	Instruction Fetch	Register Read	ALU Operation	Data Access	Register Write	Total Time
ld	200 ps	100 ps	200 ps	200 ps	100ps	800 ps
sd	200 ps	100 ps	200 ps	200 ps		700 ps
R-Format	200 ps	100 ps	200 ps		100ps	600 ps
beq	200 ps	100 ps	200 ps			500 ps



1 cycle 만 고려했을 땐 시간이 오히려 증가

BUT 병렬식 수행으로 전체시간은 감소.

파이프라이닝은 개별 명령어의 실행시간을 줄이진 못하지만
명령어 처리량을 증대시켜 성능을 향상시킨다

* 여기서부터 중간이학 *

Ideal speed up

: pipelining은 throughput을 향상 ~> 성능향상
 BUT 각 단계의 execution time은 증가할수도

$$\text{Time between Instructions} = \frac{\text{Time between Instructions (nonpipelined)}}{\text{Number of stages}}$$

↳ If all stages are perfectly balanced
 모든 단계의 실행시간이 서로 동일할 경우

Pipelining and ISA Design : RISC vs. CISC

- RISC-V ISA designed for pipelining

- 모든 명령어가 32비트

easier to fetch and decode in one cycle

C²) x86 : 1~17 byte instructions (X fixed size)

자주 사용하는 명령어일수록 명령어 길이가 짧다

↳ 명령어 길이가 고정된 RISC쪽이
 명령어를 읽어오기 더 쉽다

- few and regular instruction formats

RISC과 CISC는 항상 같은 유형에
 can decode and read registers in one step

- load / store addressing

can calculate address in 3rd stage

access memory in 4th stage

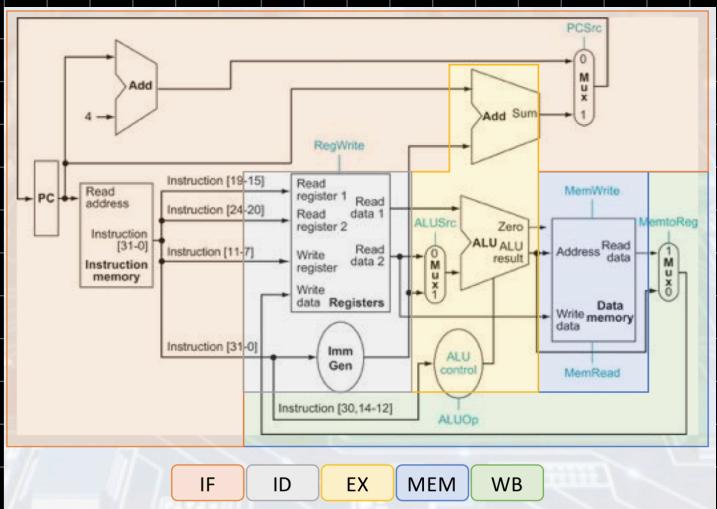
↳ x86의 경우 load / store 없음

메모리에 있는 값으로 바로 연산 가능

Reduced Instruction Set Computer (RISC) ~> RISC-V, ARM, MIPS ... 고정 (읽는 속도↑↑)

VS Complex Instruction Set Computer (CISC) ~> x86 고정 X (여러번의 사이클을 통해 읽는다)

RISC	CISC
Single Cycle Execution	many multicycle operations
Hard wired Control	micro coded multicycle operations
Load / Store architecture	register-mem and mem-mem *
Few memory addressing modes	many modes
Fixed length instruction format	many formats and lengths
Reliance on compiler optimizations	hand assemble to get good performance
Many Registers	few registers
↳ compilers are better at using them	<small>↳ 이진블레이어로 명령어 압축시 성능 향상 ↑↑ 요즘에는 기술의 발전으로 RISC, CISC 모두 compiler optimization으로 성능을 향상시키는 것이 보다 효율적</small>



Pipeline Hazard : Matching Socks in Later Load

제작의 예시 → 세탁 시 양말 한쪽씩 세탁기에 넣으면 두번 세탁기가 돌아가야 다음 단계로 진행 가능

단계 사이에 dependency 발생

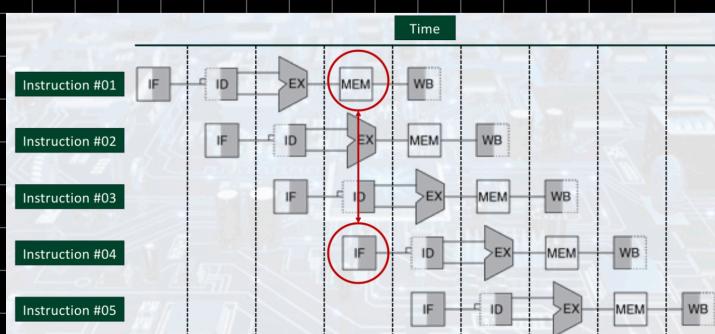
따라서 pipeline depth를 무작정 깊게 가져갈 순 없다

Hazards

- Limits to pipelining – Hazards prevent next instruction from executing during its designated clock cycle
- Three types of hazards
 - Structural hazard : A required resource is busy
 - Data Hazard : Need to wait for previous instruction to complete its data read/write
 - Control Hazard : Deciding on control action depends on previous instruction

Structural Hazards

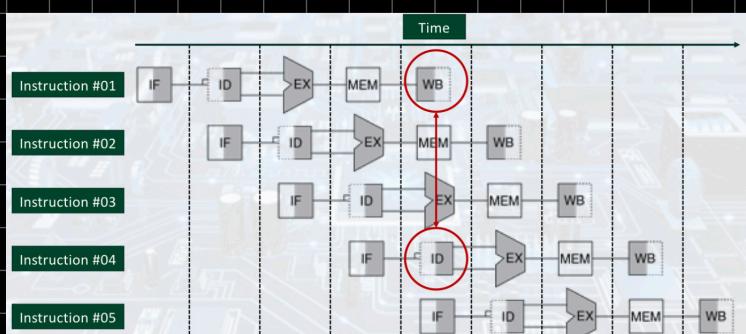
- conflict for use of a resource
 - In RISC-V pipeline with a single memory
 - load / store requires data access
 - Instruction fetch would have to stall for that cycle \rightsquigarrow would cause a pipeline "bubble"
- \rightsquigarrow Hence, pipelined datapaths require separate instruction / data memories or separate instruction / data caches



\hookrightarrow Structural Hazards : Single Memory

IF (Instruction Fetch)

MEM (Access Memory operand)



\hookrightarrow Structural Hazards : Registers

ID (Instruction Decode & Register read)

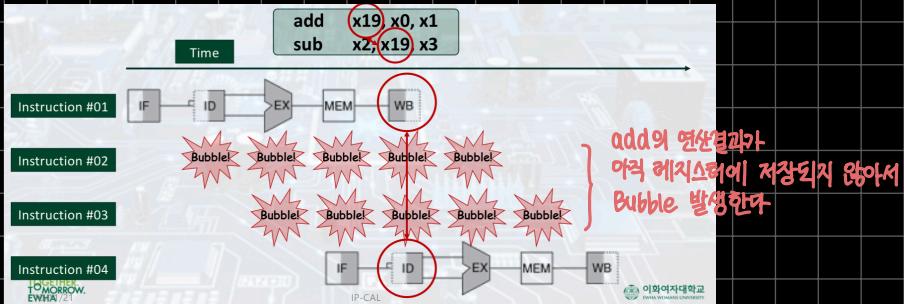
WB (Write result Back to the register)

Solution : Structural Hazards

- for Memory Hazard : two memory structures for instruction and data
- for register hazards :
 - register access is VERY fast - ALU stage의 절반도 소요되지 않는다
 - solution : introduce convention
 - (always write to registers during first half of each clock cycle
 - always read from registers during second half of each clock cycle
 - ⇒ can perform read and write during same clock cycle

Data Hazards

: an Instruction depends on completion of data access by a previous instruction

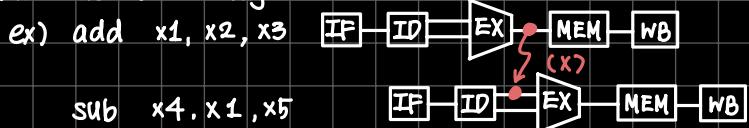


Solution : Data Hazards

- as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract
- ⇒ Forwarding / Bypassing 통해 데이터를 미리 보낸다 전방전달 / 우회전달
- Adding Extra hardware to retrieve the missing item early from the internal resources

Forwarding with Two Instructions

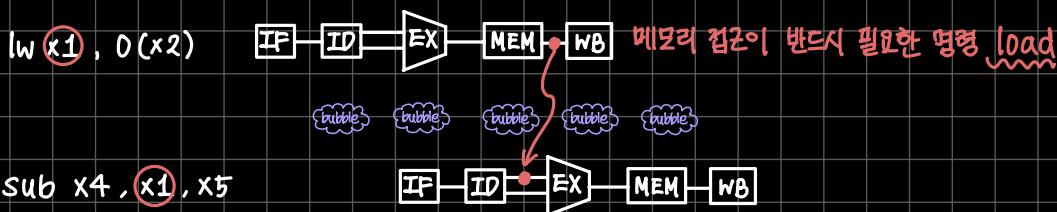
: forwarding paths are valid only if the destination stage is later in time than the source stage



이전의 단계로 전방전달 (forwarding)하는 것은
불가능하다 → 시간을 되돌리는 것 같아(?)

Load - Use Data Hazard – forwarding / Bypassing 으로도 해결 불가능

- can't always avoid stalls by forwarding
 - ↳ If value not computed when needed ... can't forward back in time



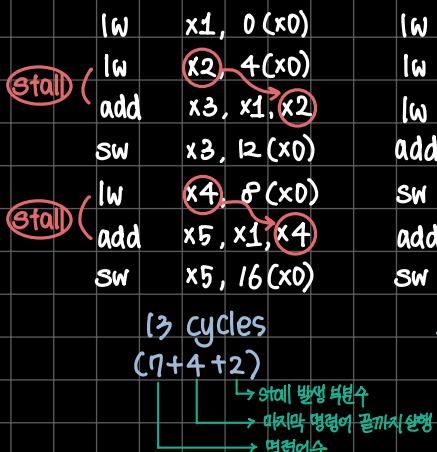
작제(lw) 뒤의 R-type – 종속되는 결과값있으면 반드시 data hazard 발생
forwarding 이용해도 반드시 버블이 한 단계 발생한다
Instruction Decode 전까지 필요한지 알 수 없기 때문

Code Scheduling to Avoid Stalls

- reorder code to avoid use of load result in the next instruction

ex) C code for

```
a = b + e  
c = b + f  
(lw → Rtype 예약 문제)
```



증속성이 있는 명령이끼리는
떨어지게 배치하여 Stall을 방지

bubble이 한단계니까
명령이 하나만 끼워넣어도 OK

① forwarding 없으면
증속된 명령어 사이의 data hazard로
두 단계의 버블 발생

② forwarding → 여전히 한단계 버블 O
(MEM/WB → D/EX)

③ forwarding + 증속된 명령어들 순서 조정
⇒ 지연을 해소

Control Hazards

- let's assume that we put in enough hardware so that we can test registers, calculate the branch target address, and update the PC during the second stage
- wait until the branch outcome determined before fetching next instruction

브랜치 관련 명령어 → 연산이 끝나야 다음 명령어가 결정되므로 stall이 발생한다

연산 완료 후 IF 진행 가능 ~ equals 판단하는 sub 연산



bubble bubble bubble bubble bubble



Possible Solution : Branch Prediction

- Always predict that branches will be untaken
(untaken → pipeline proceeds at full speed)
(taken → stall the pipeline)
- When the guess is wrong,
the pipeline control must ensure that the instructions following the wrongly guessed branch
have no effect and must restart the pipeline from the proper branch address

분기지를 택하지 않고 순서대로 진행
PC value = PC + 4

이전 명령에서 MEM, WB 전 ALU 단계 끝나자마자
다음 명령의 IF 실행 시작

→ forwarding / bypassing 으로 연산결과를
가져온 것. 즉 forwarding 이용해도
register read 가 아니라 instruction
fetch에서부터 연산결과가 필요하므로
stall이 발생한다



bubble bubble bubble bubble bubble



* 질문할 것 . . .

. Pipeline 예약의 포워딩 위치

. 포워딩이 없을 경우

. VS 포워딩 有 + 예측 성공

. VS 포워딩 無 + 예측 실패

지연 발생 횟수 비교

Dynamic Branch Prediction

PC 값을 사용해서 taken / untaken 중 선택 가능성이 더 높은 경우를 예측한다

- the static methods are rigid and rely on stereotypical behavior
 - no individuality of a specific branch
- Dynamic hardware predictors make their guesses depending on the behavior of each branch
 - keeping a history for each branch as taken or untaken, and then using the past to predict the future
 - about 90% accuracy

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
 - executes multiple instructions in parallel
 - each instruction has the same latency
- Subject to hazards — ① structural ② data ③ control
- Instruction set design affects complexity of pipeline implementation