

Chapter4. The Processor

통계 수정 삭제

wnwicks123 · 2일 전

♥ 0

CS

CS



▼ 목록 보기

3/4



=== 4장 1,2강 ===

Introduction

컴퓨터구조

- CPU performance factors
 - Instruction count
 - Determined by ISA and compiler
 - CPI and Cycle time
 - Determined by CPU hardware
- We will examine two MIPS implementations
 - A simplified version
 - A more realistic pipelined version
- Simple subset, shows most aspects
 - Memory reference: lw, sw
 - Arithmetic/logical: add, sub, and, or, slt
 - Control transfer: beq, j



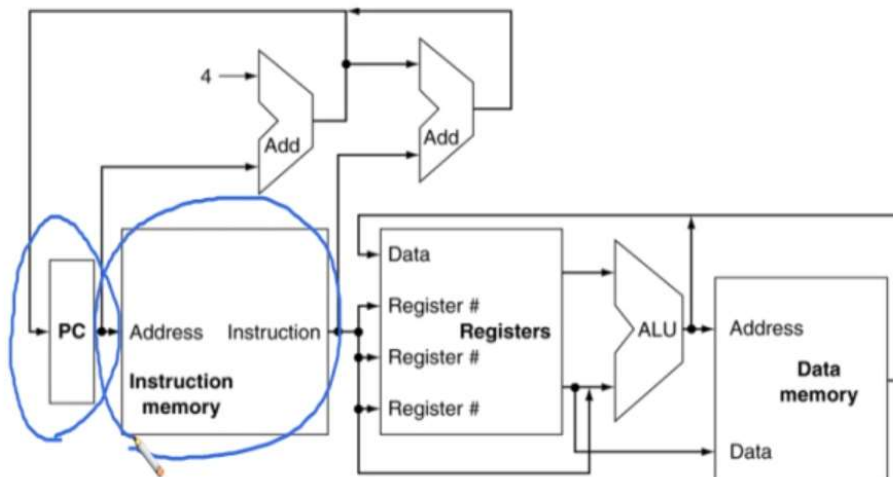
Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
 - Use ALU to calculate
 - Arithmetic result
 - Memory address for load/store
 - Branch target address
 - Access data memory for load/store
 - PC ← target address or PC + 4



CPU 구조

CPU Overview



Combinational Elements

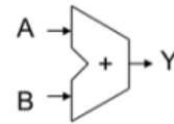
- AND-gate

- $Y = A \& B$



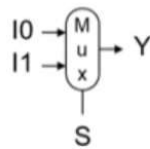
- Adder

- $Y = A + B$



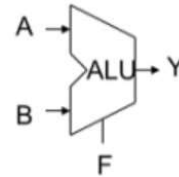
- Multiplexer

- $Y = S ? I1 : I0$



- Arithmetic/Logic Unit

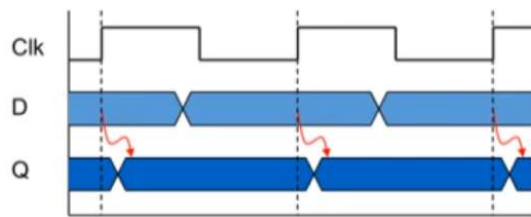
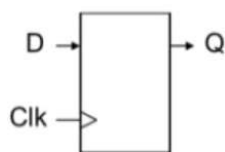
- $Y = F(A, B)$



Sequential Elements

- Register: stores data in a circuit

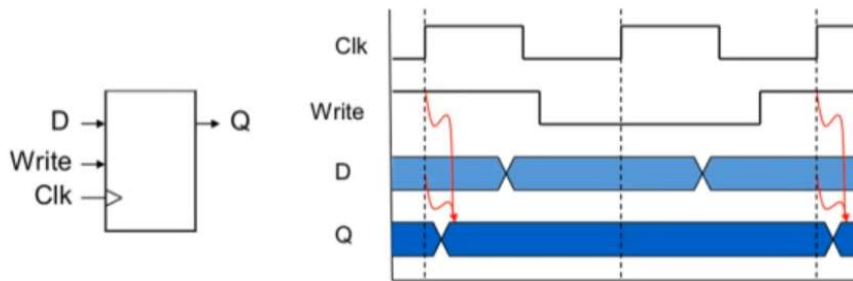
- Uses a clock signal to determine when to update the stored value
 - Edge-triggered: update when Clk changes from 0 to 1



- Edge-trigger 라고 표현하기도 한다.
- rising edge 일경우 값이 업데이트가 된다.

Sequential Elements

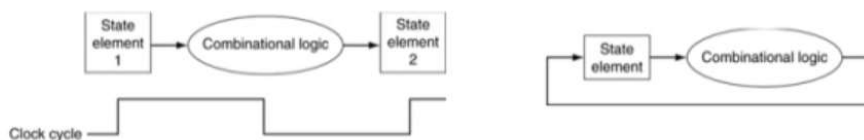
- Register with write control
 - Only updates on clock edge when write control input is 1
 - Used when stored value is required later



- 클락이 rising edge이면서 Write이 1일때 값이 업데이트 된다.

Clocking Methodology

- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Input from state elements, output to state element
 - Longest delay determines clock period



- Combinational logic : 하나의 클락사이클 동안 어떤 연산이 다 끝나야 하는것.

4.3 Building a Datapath



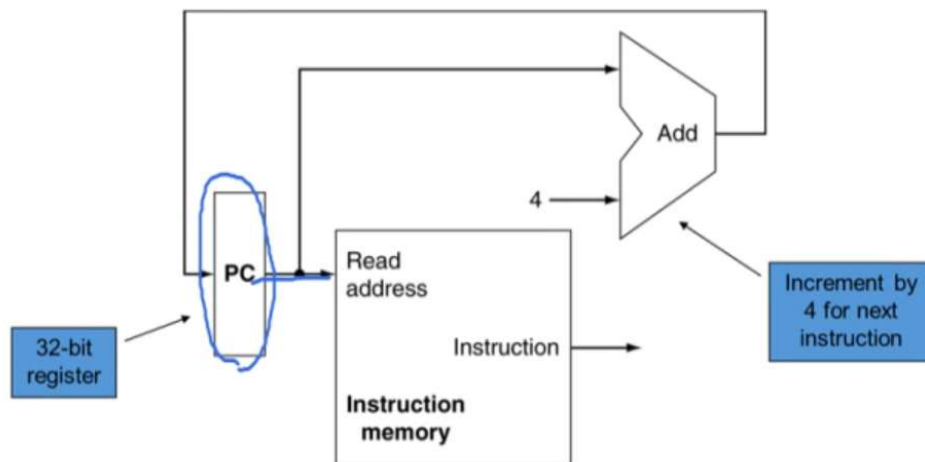
Building a Datapath

- Datapath
 - Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
 - Refining the overview design



- Datapath : 데이터를 처리하거나 데이터와 address를 cpu에서 처리해주는 elements

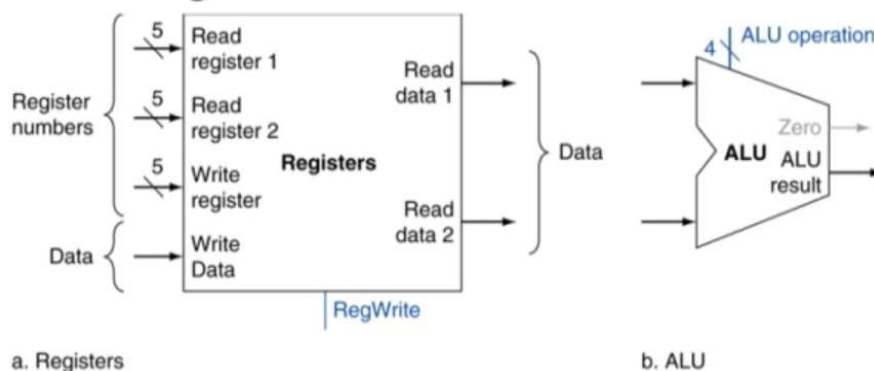
Instruction Fetch



- Instruction Fetch : 명령어를 실행할 때 첫번째로 실행하는 일.

R-Format Instructions

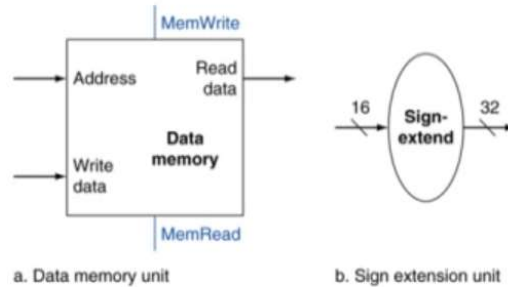
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



- 레지스터간의 연산
1. 두개의 레지스터에서 값을 읽어서 연산을 해서 그 결과를 결과 레지스터에 적는다.

Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

b. Sign extension unit

1. 레지스터값을 읽는다.
- 2.

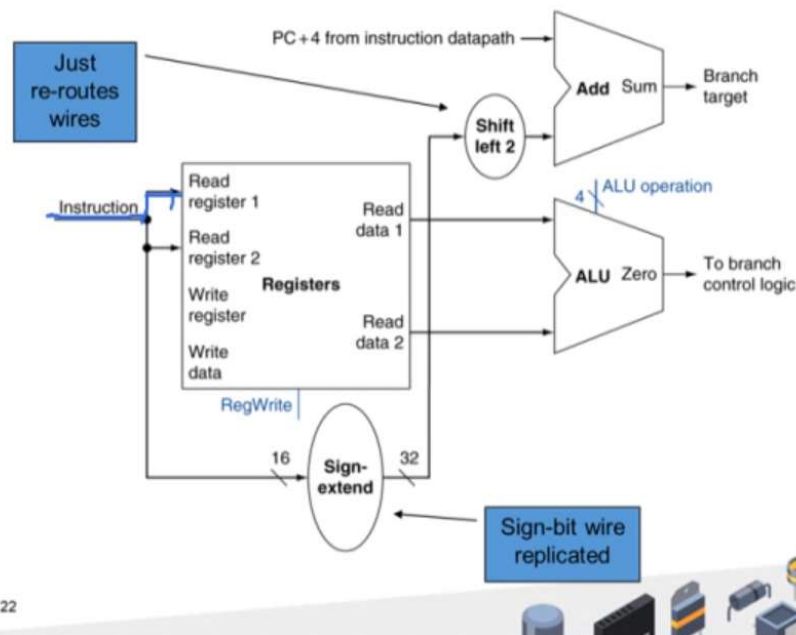
Branch Instructions

- Read register operands
- Compare operands
 - Use ALU, subtract and check Zero output
- Calculate target address
 - Sign-extend displacement
 - Shift left 2 places (word displacement)
 - Add to PC + 4
 - Already calculated by instruction fetch

1. 레지스터의 값을 읽어와야한다.
2. operands값을 계산한다. : ALU, subtract 후 Zero output을 만들어냄

3. 동시에 target address를 계산
4. 16비트를 32비트로 바꿈.
5. 왼쪽으로 2비트 움직임 (x4 왜냐하면 32비트이기 때문)
6. $PC + 4$

Branch Instructions

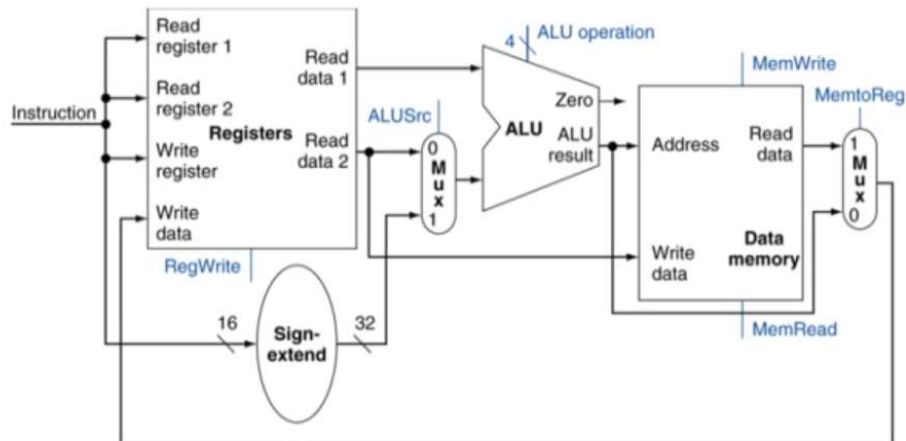


Composing the Elements

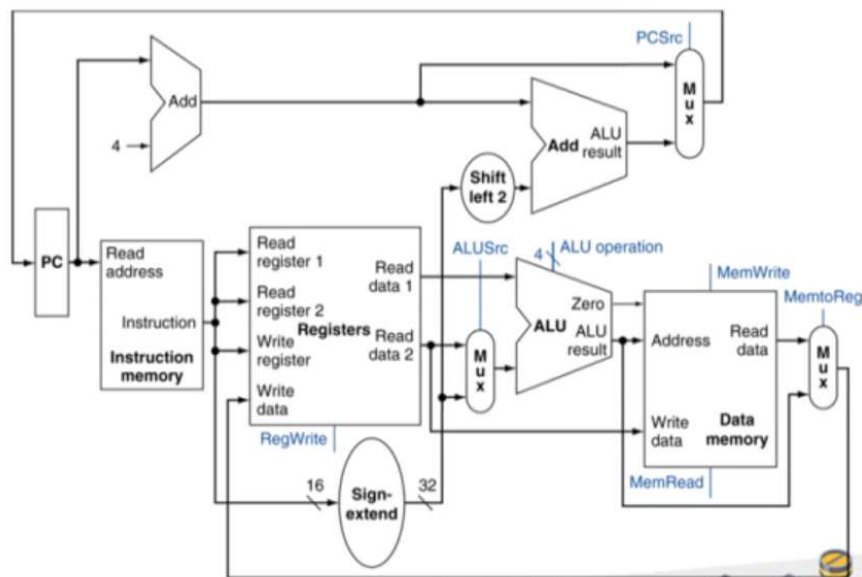
- First-cut data path does an instruction in one clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

- First-cut data path : 명령어를 하나 실행하는데 한사이클 걸리는 것.
 -- 각 시간의 데이터 패스한 컴포넌트들이 한 오퍼레이션만 한다.
 -- 데이터 메모리와 instruction 메모리를 나눠야 한다.
- 적당한 곳에 MUX를 사용한다.

R-Type/Load/Store Datapath



Full Datapath



퀴즈

“Combinational element”의 정의는?

- 아웃풋이 인풋의 function으로 결정이 되는 것.

ALU Control

- ALU used for
 - Load/Store: F = add
 - Branch: F = subtract
 - R-type: F depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

- Load/Store : 더하기
- Branch : 빼기
- R-type : 그외

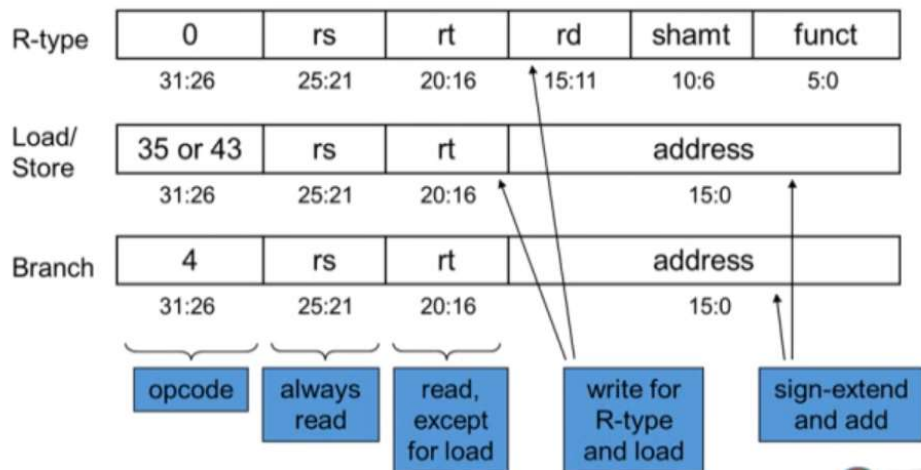
ALU Control

- Assume 2-bit ALUOp derived from opcode
- Combinational logic derives ALU control

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

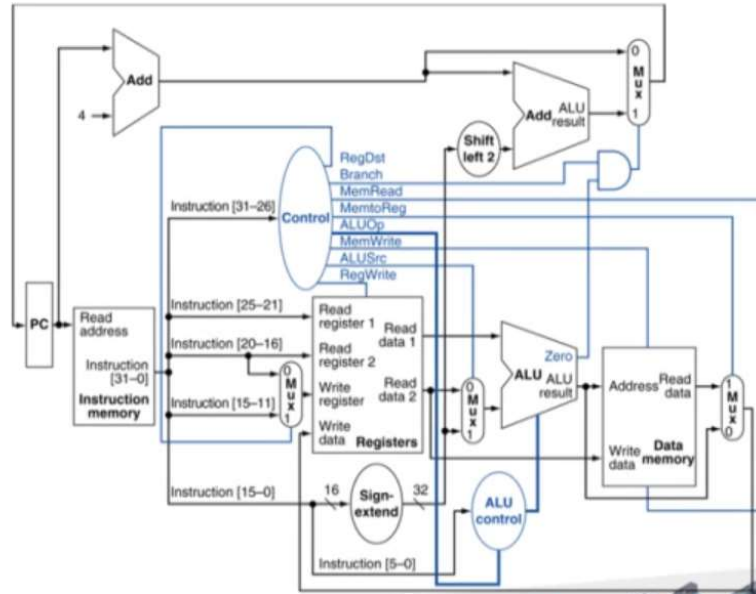
The Main Control Unit

- Control signals derived from instruction

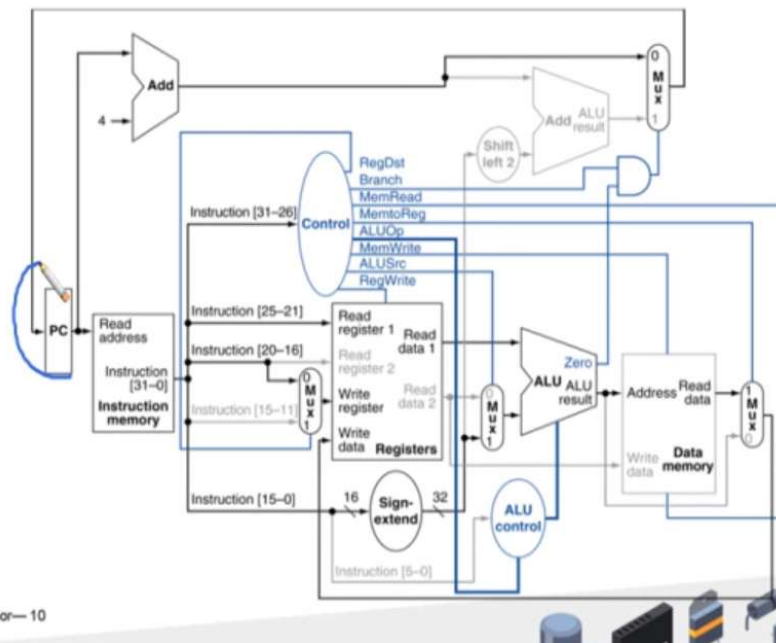


- 모든 Control signal은 해당 명령어로 부터 만들어진다.

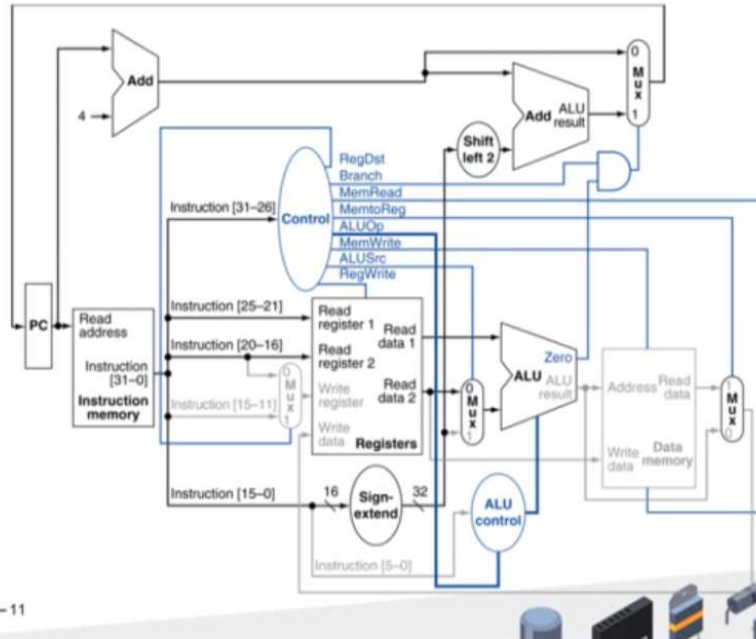
Datapath With Control



Load Instruction



Branch-on-Equal Instruction



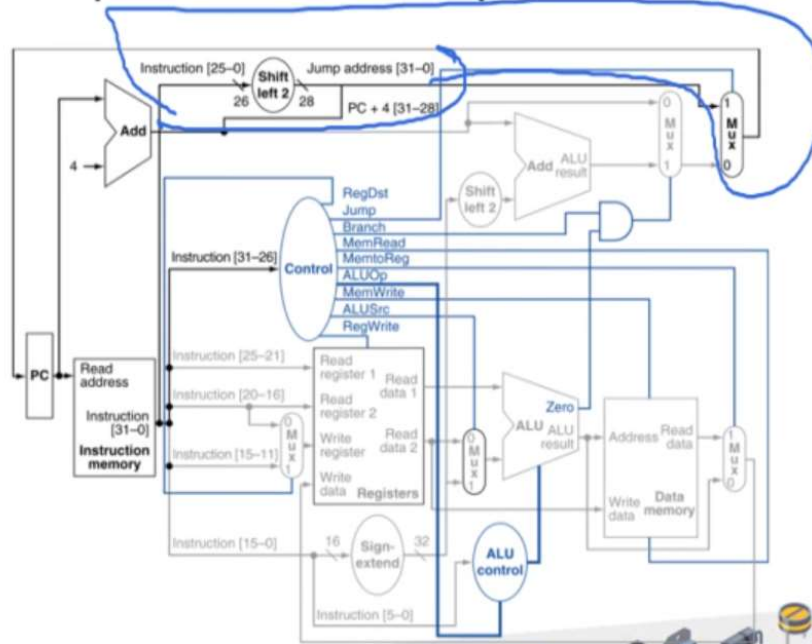
Implementing Jumps



- Jump uses word address
- Update PC with concatenation of
 - Top 4 bits of old PC
 - 26-bit jump address
 - 00
- Need an extra control signal decoded from opcode

- Jumps 명령어의 이식

Datapath With Jumps Added



퍼포먼스 이슈

Performance Issues

- Longest delay determines clock period
 - Critical path: load instruction
 - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
 - Making the common case fast
- We will improve performance by pipelining

- 가장 긴 딜레이가 clock period를 결정한다.
- Load 명령어가 가장 오래 걸린다.
- 각각의 명령어에 따라서 다른 clock period를 사용하는것은 현재의 기술로 힘들다.

- Common case 를 빠르게 하기 어렵다. 따라서 파이프라인 이라는것을 적용할 것이다.

4.5 An Overview of Pipelining



- Pipelining Analogy : 스텝의 병렬화

MIPS Pipeline

- Five 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



- MIPS의 Pipeline은 5스텝으로 이루어져 있다.

1. IF : Instruction을 메모리로부터 읽어오는것.
2. ID : Instruction을 decode하고 register를 읽는것.
3. EX : ALU 연산을 하는것.
4. MEM : 메모리에 액세스 하는것.
5. WB : 결과를 실제 레지스터에 쓰는것.

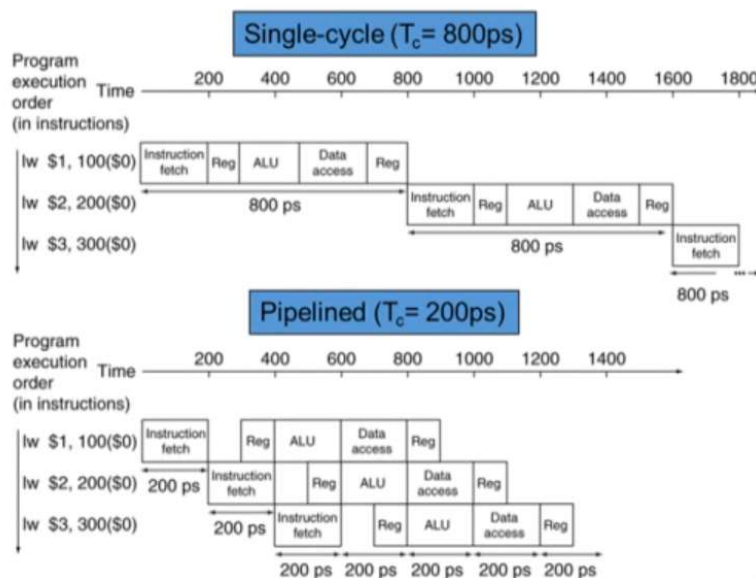
Pipeline Performance

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps



Pipeline Performance



Pipeline Speedup

- If all stages are balanced
 - i.e., all take the same time
 - $$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$
- If not balanced, speedup is less
- Speedup due to increased throughput
 - Latency (time for each instruction) does not decrease



- Pipeline Speedup : 파이프라인을 사용했을때 얼마만큼의 성능 향상을 얻는가
- 모든 스테이지가 다 균등하다는 가정하에 :
 - 파이프라인의 한 스테이지 값은 전체의 파이프라인 시간에 비례하고 스테이지의 수에 반비례 한다.
- 균등하지 않다는 가정하에 : 성능향상이 줄어든다.
- Throughput을 향상 시켜서 성능 향상을 얻는다.
 - 하나의 명령어를 실행하는데 시간은 줄어들지 않는다(오히려 늘어날 수도 있다.)

Pipelining and ISA Design

- MIPS ISA designed for pipelining
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17-byte instructions
 - Few and regular instruction formats
 - Can decode and read registers in one step
 - Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - Memory access takes only one cycle



- MIPS의 ISA는 pipelining에 꽤 적합하다.
 - 모든 명령어가 32비트이다.
 - 명령어의 포맷도 작다.
 - Load/store addressing이다.
 - memory operands이다.

퀴즈

다음 중 MIPS의 pipeline stage가 아닌 것은?

- ① Instruction Fetch
- ② Instruction decode & register read
- ③ Execute operation or calculate address
- ④ ☒ Disk Access



=== 4장 3,4강 ===

- 파이프 라인에 대해

컴퓨터구조

1차시

4.5 An Overview of Pipelining



컴퓨터구조

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
 - 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




- Hazards : 다음 사이클의 다음 명령어를 실행하지 못하는 상황 (어떠한 이유에 대해서던간에)
1. Structure hazards
어떠한 명령어를 실행할때 다른 명령어가 실행하고 있어서 리소스를 사용하지 못할때.
 2. Data hazard
Data dependency 앞의 명령어의 결과가 나오지 않아 뒤의 명령어가 실행되지 못할때.
 3. Control hazard
앞의 명령어에 의해 후의 명령어의 결과값이 달라질때.


Structure Hazards

컴퓨터구조

Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
 - Load/store requires data access
 - Instruction fetch would have to *stall* for that cycle
 - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
 - Or separate instruction/data caches


The Processor—6

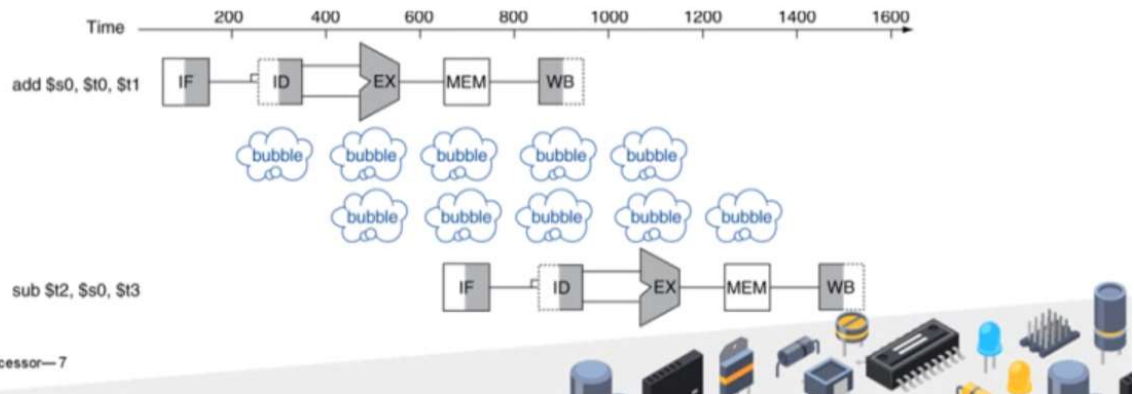


- 리소스의 사용의 충돌이 일어날때.
- 앞의 로드 명령어가 로드 명령어를 액세스 할때 까지 기다려야 한다. Hazard 발생
- 파이프라인 데이터패스에선, 명령어 메모리와 데이터 메모리를 구분한다.
- 해결법 : 리소스를 추가한다.(메모리 추가)

Data Hazards

Data Hazards

- An instruction depends on completion of data access by a previous instruction
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3

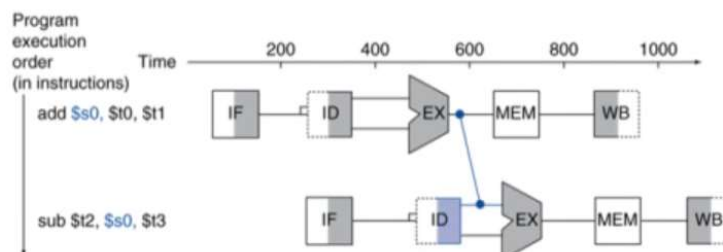


- 전의 데이터값을 후의 데이터가 사용하기 위해 전의 데이터연산이 끝날 때 까지 기다리는것.

Data Hazards를 해결하는 다른 방법

Forwarding (aka Bypassing)

- Use result when it is computed
 - Don't wait for it to be stored in a register
 - Requires extra connections in the datapath



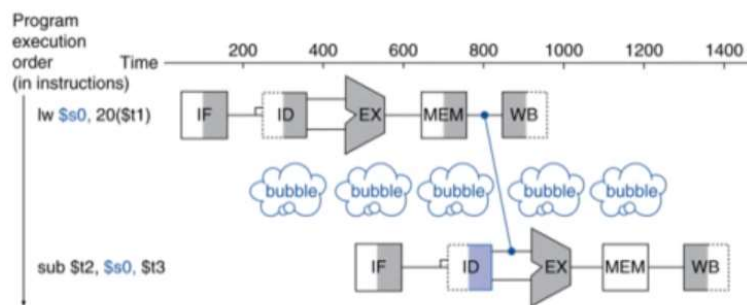
- 포워딩(바이패싱) : 어떤 결과값을 계산했을때 그 결과값을 바로 사용하는 것.
- 데이터 패스에 추가적인 커넥션이 필요하다.

Load-Use Data Hazard

컴퓨터구조

Load-Use Data Hazard

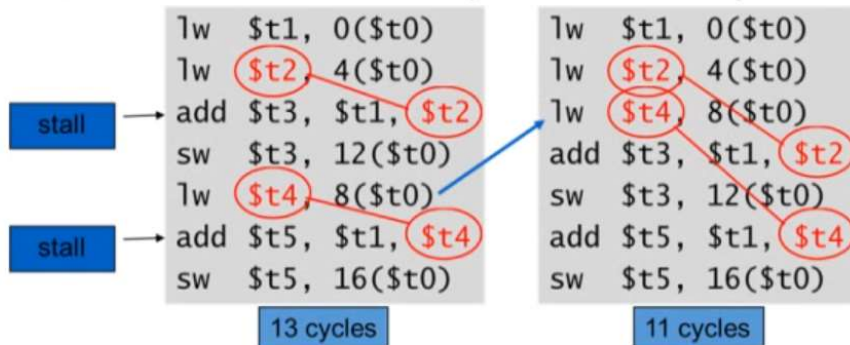
- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!



- Load-Use Data Hazard : 로드 명령어에 의해서 발생하는 Hazard

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E$; $C = B + F$;



- Hazard를 막기 위해서 코드를 rescheduling 하는것이 필요하다.

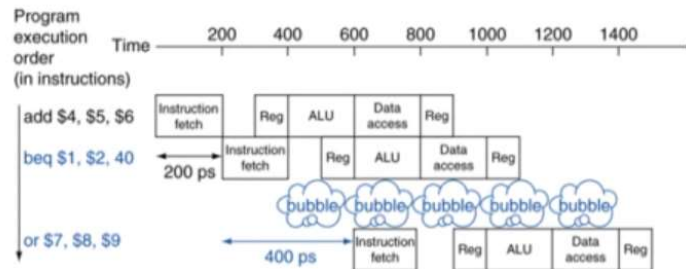
Control Hazards

- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- In MIPS pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to do it in ID stage

- Branch의 결과의 값에 따라서 다음 명령어가 결정이 된다.

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



- 다음명령어를 fetching 하기 전까지 기다려야 한다.

Branch Control Hazards를 해결하는 방법

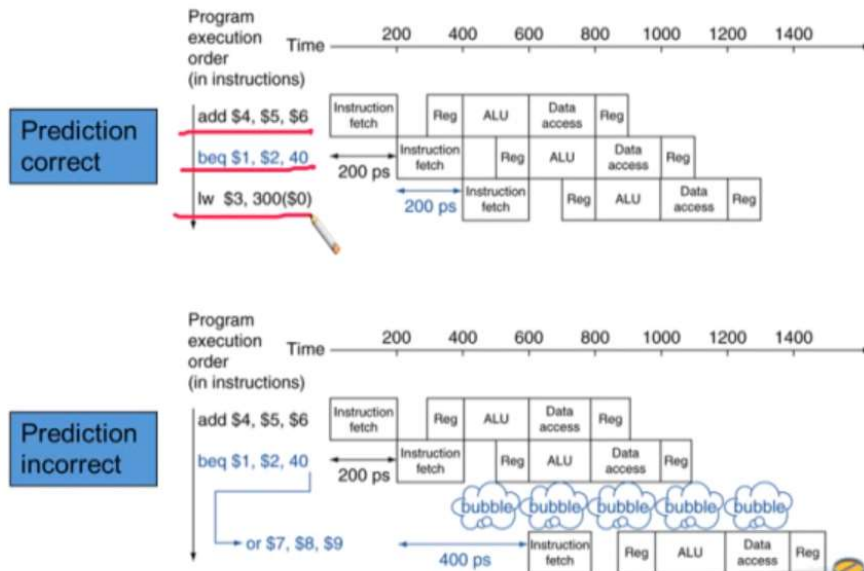
Branch Prediction

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay

- Branch Prediction : 브랜치의 결과를 예측한다.
: 브랜치를 예측해서 맞으면 진행 아니면 진행하지 않는다.
- MIPS에서는 Branch가 발생하지 않는다고 생각하고 진행한다.

MIPS with Predict Not Taken

컴퓨터구조



- 맞으면 진행 아니면 X

More-Realistic Branch Prediction

컴퓨터구조

- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches
 - Predict backward branches taken
 - Predict forward branches not taken
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history


- Branch Prediction은 크게 두 종류로 나뉜다
 1. Static branch prediction
branch prediction의 결과가 고정되어있다.(항상 값이 고정되어있다고 예측한다.)
 2. Dynamic branch prediction
branch prediction의 결과가 동적으로 바뀐다.
최근 branch 결과값을 사용하기때문에 메모리 공간이 필요하다.


컴퓨터구조

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation


The Processor— 16



- Pipeline은 throughput을 증가시켜 성능을 향상시킨다.
여러개의 명령어를 동시에 수행해서 성능을 증가 시킨다.
- 어떤 Instruction set을 가지고 있냐가 파이프라인 구현에 영향을 미친다.

퀴즈

“Hazard”의 정의는?

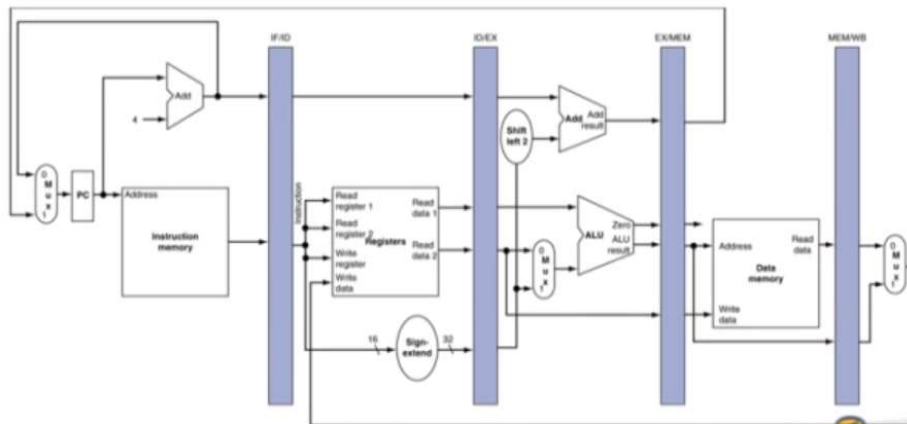
- 다음 사이클의 다음 명령어를 수행하지 못하는것을 Hazard라고 한다.

2차시

4.6 Pipelined Datapath and Control

Pipeline registers

- Need registers between stages
 - To hold information produced in previous cycle



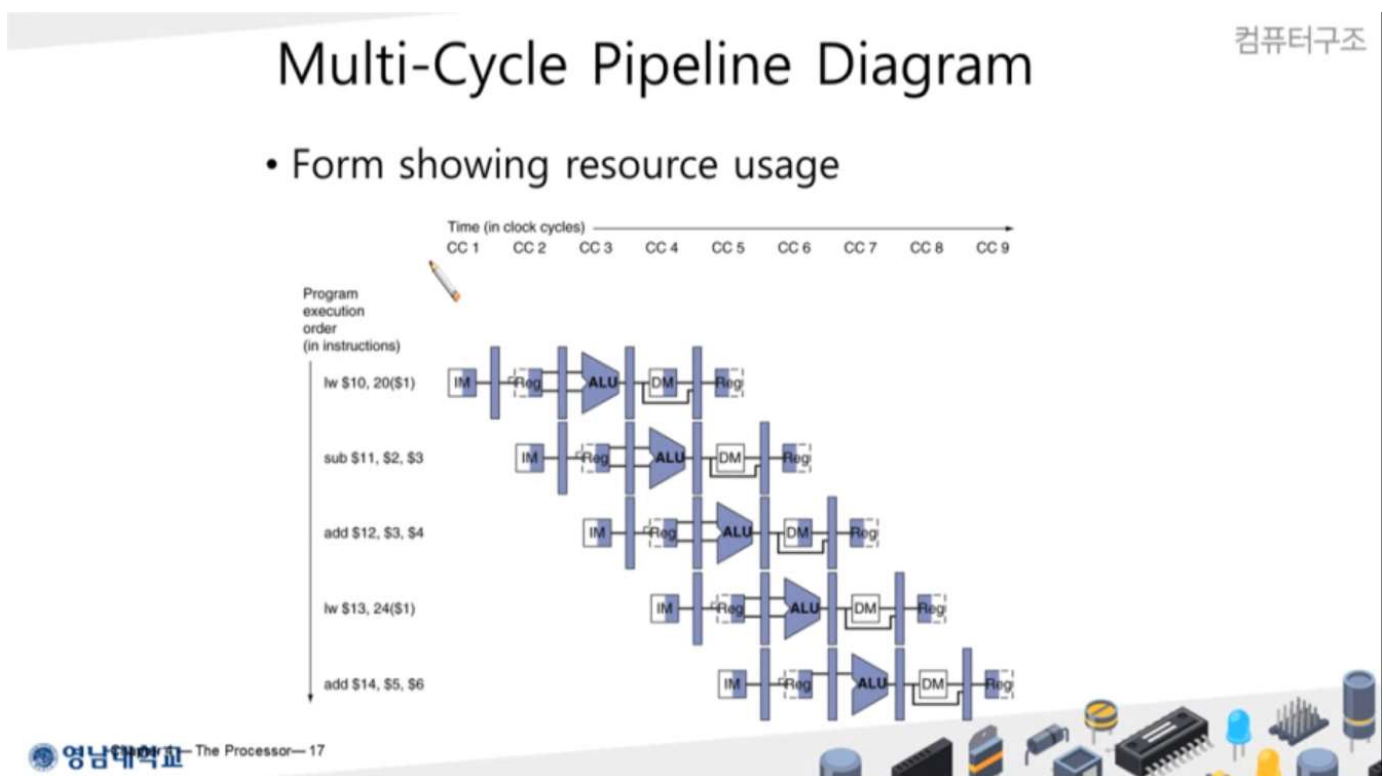
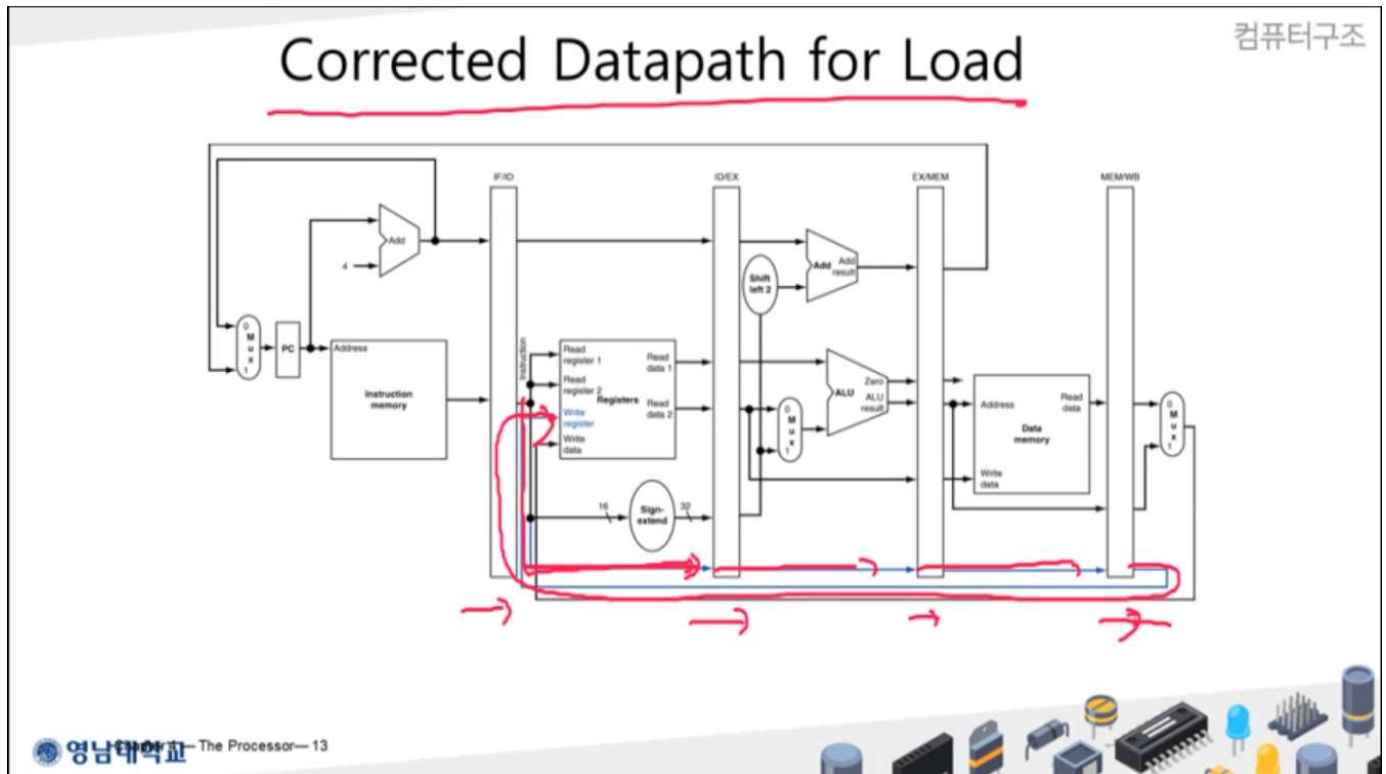
- Pipeline registers : 각 스테이지마다 그 스테이지에서 나온 값을 저장하는곳.

Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
 - "Single-clock-cycle" pipeline diagram
 - Shows pipeline usage in a single cycle
 - Highlight resources used
 - c.f. "multi-clock-cycle" diagram
 - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

- Pipeline Operation 은 두가지 다이어그램으로 표현한다.
 1. Single-clock-cycle pipeline diagram : 특정한 한 사이클의 파이프라인의 상태를 보여준다
-- 어떤 리소스를 쓰는지 다 보여준다.

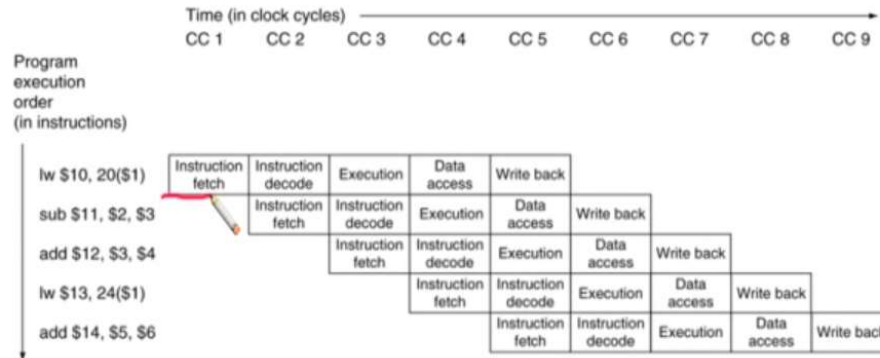
2. Multi-clock-cycle diagram : 여러 사이클에 걸쳐서 어떤식으로 실행이 되는지 보여주는것.



- Multi-Cycle Pipeline Diagram : 사이클의 시간에 지남에 따라 어떤 명령어들이 어떤 스테이지에서 각각 실행되는지 쉽게 이해 할 수 있다.

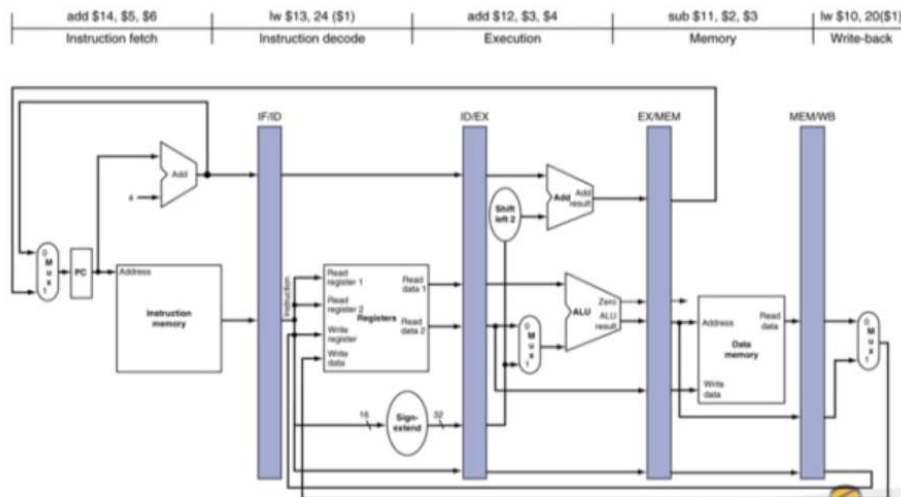
Multi-Cycle Pipeline Diagram

• Traditional form

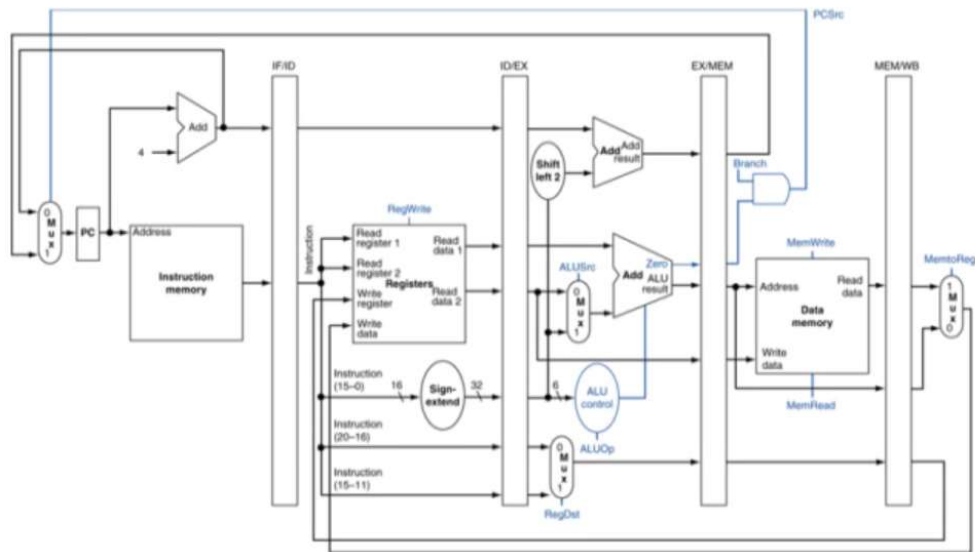


Single-Cycle Pipeline Diagram

• State of pipeline in a given cycle

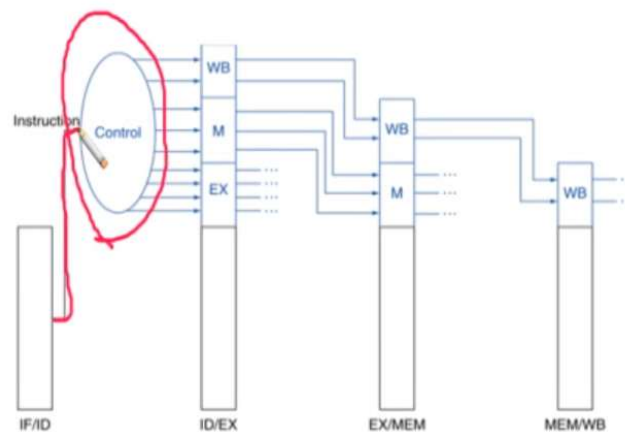


Pipelined Control (Simplified)



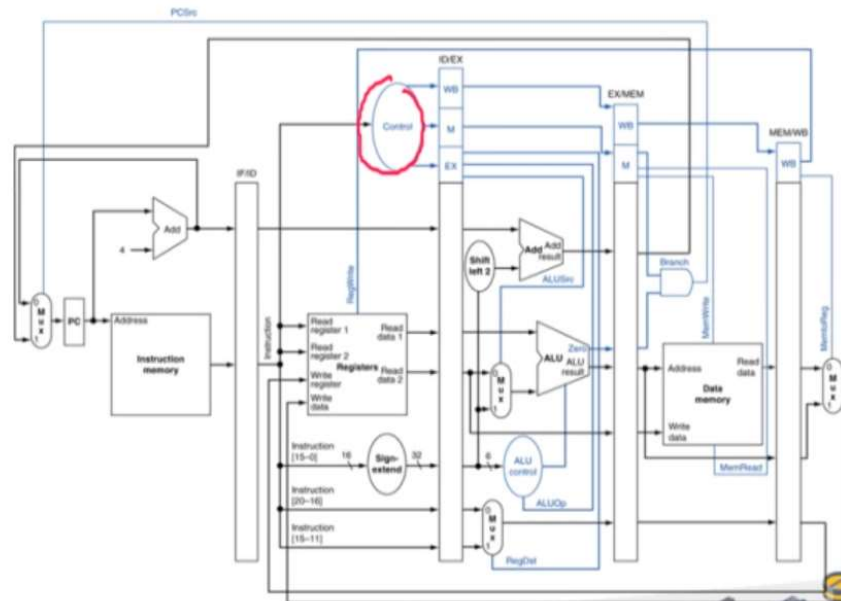
Pipelined Control

- Control signals derived from instruction
 - As in single-cycle implementation



Pipelined Control

컴퓨터구조



영남대학교 The Processor—22

컴퓨터구조

퀴즈

Pipelined processor에서 Forwarding의 정의는?

Use result when it is computed

영남대학교

- 결과가 계산이 됐을때, 바로 그 값을 사용하는 것.



도로로

다음 포스트

Chapter2. Instruction:Language of the Computer



이전 포스트

Chapter3. Arithmetic for computers

0개의 댓글

댓글을 작성하세요

댓글 작성



Powered by
Stellate