Ch.4-1 The Processor - single

▼ Introduction

- Cpu performance factors
  - Instruction count **by ISA and compiler**
  - CPI, Cycle time **by CPU hw**

  ⇒ processor의 구현 방법에 따라서 달라짐

  → MIPS에서는어떻게 구현되어 있는지 알아보자

- MIPS implementation
  - A simplified version (Simple - cycle Computer)
  - A more realistic pipelined version → pipelined 어떻게?
- 따라서 가장 기본적인 명령어 집합만 알아볼 것임
  - Memory-reference : ***lw, sw***
  - Arithmetic-logical : ***add, sub, and, or, slt***
  - Control flow : ***beq, j***

▼ Instruction Execution

1. PC : instruction memory에서 instruction 가져옴 → fetch (next → 이 친구의 memory address)
2. 읽을 register를 선택하는 instruction field를 사용하여 register read (↳ instruction의 register number)
   a. Register file : fetch한 instruction을 처리하기 위해 필요한 src file
3. 이후 명령어 종류에 따라 실행하는 내용이 달라짐

   ① 모든 명령어 → 이후에 각자의 목적에 맞게 ALU 사용(단, jump 제외!)

      i. Arithmetic result

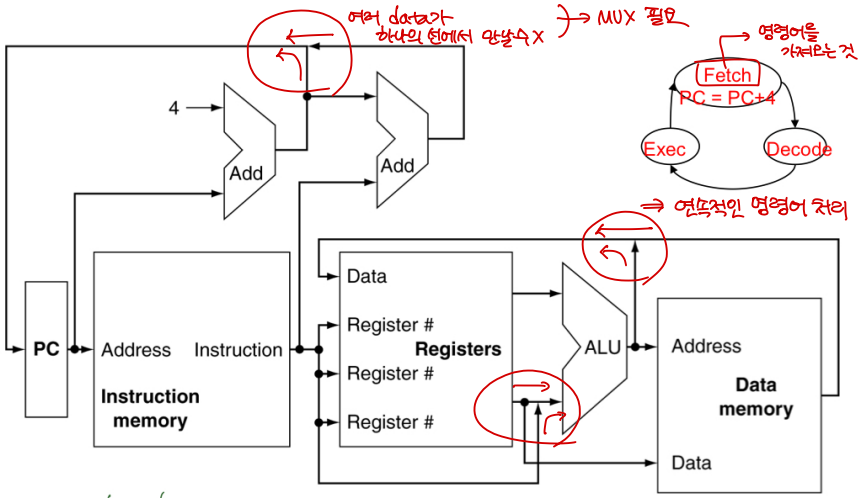
      ii. Memory address for lw/sw

      iii. Branch target address 등등...

   ② 명령어 종류에 따라 다음 단계가 다름

i. Lw/sw → memory 접근

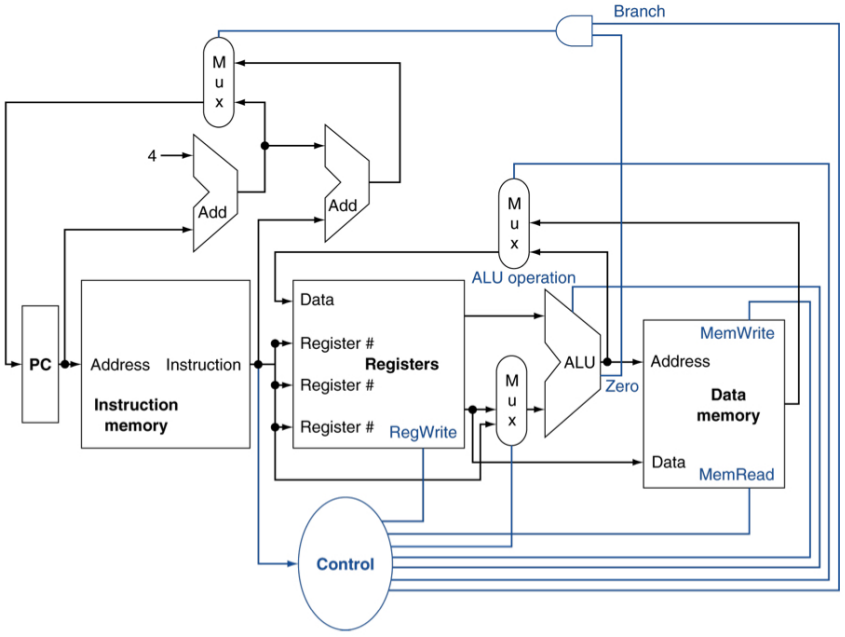
ii. Branch → target address or PC+4



→ MUX 사용 ⇒ 여러 데이터 중 하나를 선택해야 할 때 사용

→ 이들을 control하기 위해 control unit 추가

MUX에서 무엇을 선택?



- Control unit : 명령어를 입력으로 받아서 기능 unit들과 mux의 제어선 값을 결정

▼ Clocking Methodologies

신호를 읽고 쓰는 시점을 정의 → read/write 타이밍을 명확하게 지정하는 것이 중요함

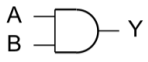
▼ 컴퓨터 회로 복습

- Low = 0 / High = 1
- Multi-wire bus → 내부적으로 여러 data 동시에 보낼 때 사용

- Combinational element
- State element

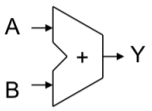
❑ AND-gate

▶ Y = A & B



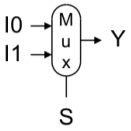
❑ Adder

▶ Y = A+B



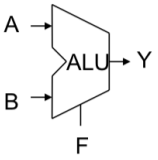
❑ Multiplexer

▶ Y = S ? I1 : I0



❑ Arithmetic/Logic Unit

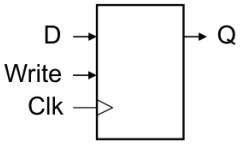
▶ Y = F(A,B)

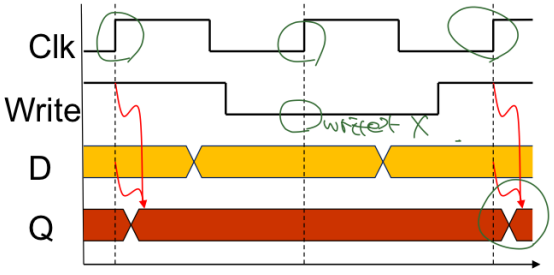


▼ Edge-triggered clocking

- Clock이 0에서 1로 바뀔 때 update(rising edge일 때)
- Register → write control + clock edge에서만!

⇒ write control = 1 ⊕ clock edge

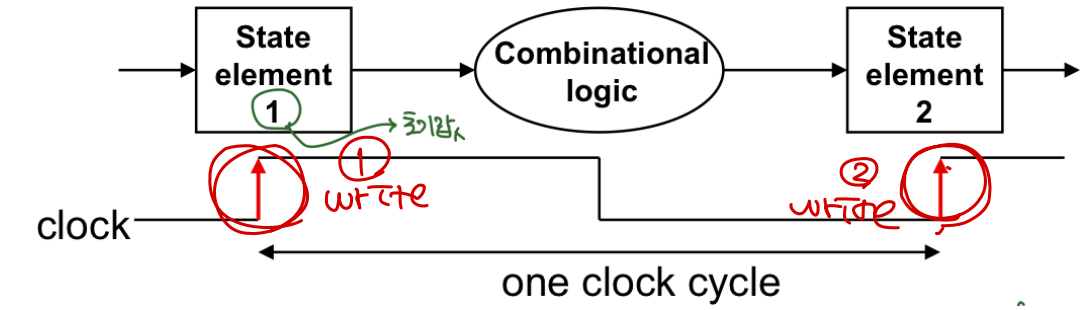




⇒ 중간에 들어오는 input은 무시

▼ Clocking methodologies

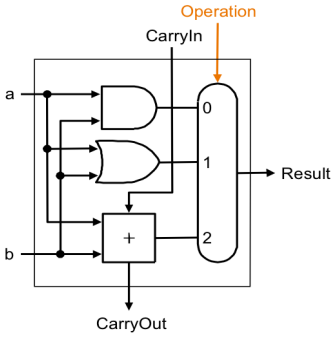
- 조합 회로 : clock cycle 동안 data 변경 가능
- Longest delay → clock period
  - 어떤 statement 시작부터 다음번 stater까지 가장 길게 걸리는 시간 ⇒ clock cycle 길이가 될 것이다!
- Typical execution
  1. State로부터 content read
  2. 조합회로를 통해 value 전송
  3. 하나 이상의 state element에 결과 write

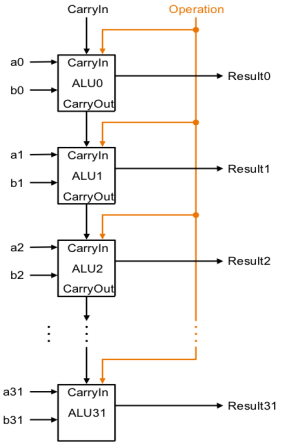


▼ ALU Extension

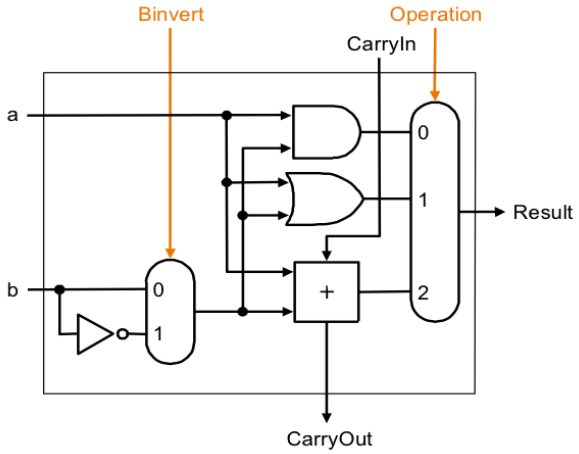
▼ ALU(arithmetic logic unit)

- 32bit ALU





- Subtraction in ALU
  - 2의 보수 적용 → b′ + 1



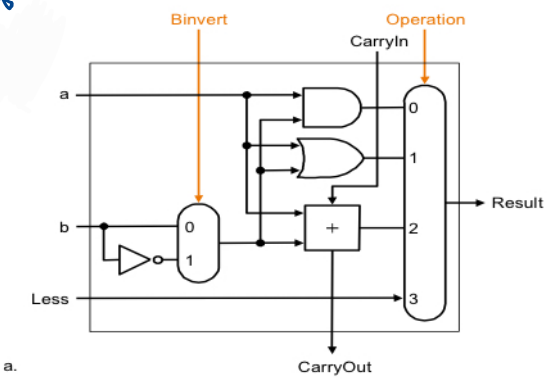
- slt(set on less than)
  - Subtraction 이용하여 구현

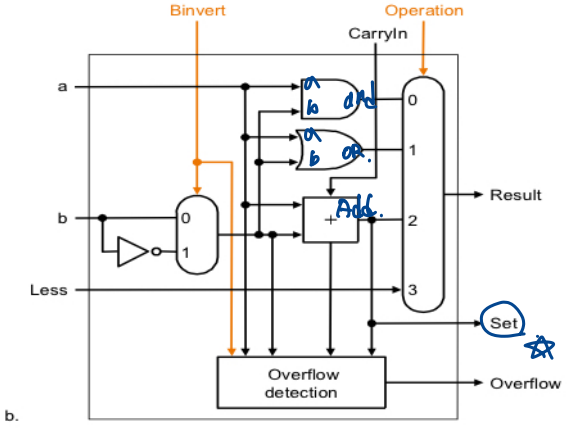
slt $t0, $s0, $s1 ⇒ R format

$s0 < $s1 ⇒ $t0 = 1

$s0 ≥ $s1 ⇒ $t0 = 0

- rs < rt ⇒ 1 / rs ≥ rt ⇒ 0
- rs - rt < 0 ⇒ rs < rt 이용





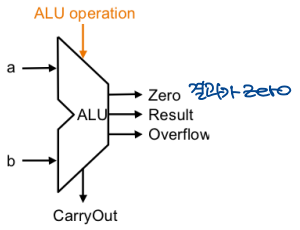
- Beq(branch equal)
  - Subtraction 이용하여 구현
    - A-b = 0 ⇒ a=b

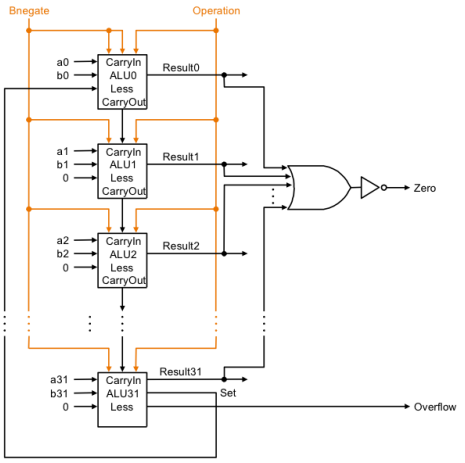
▼ ALU Extension

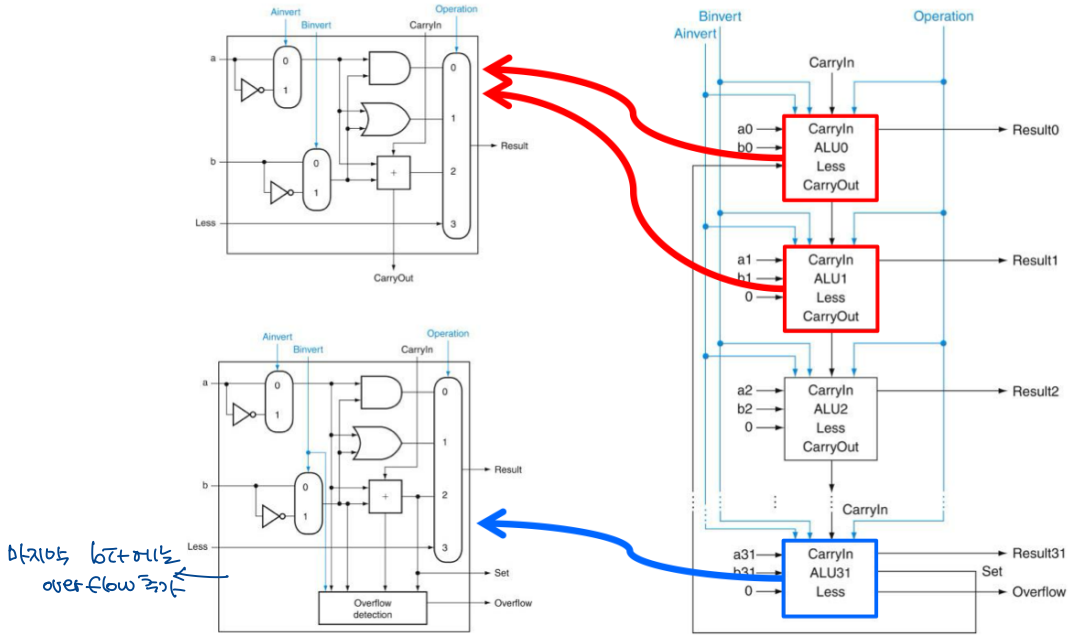
- Notice control lines:

```
OP
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

•Note: zero is a 1 when the result is zero!





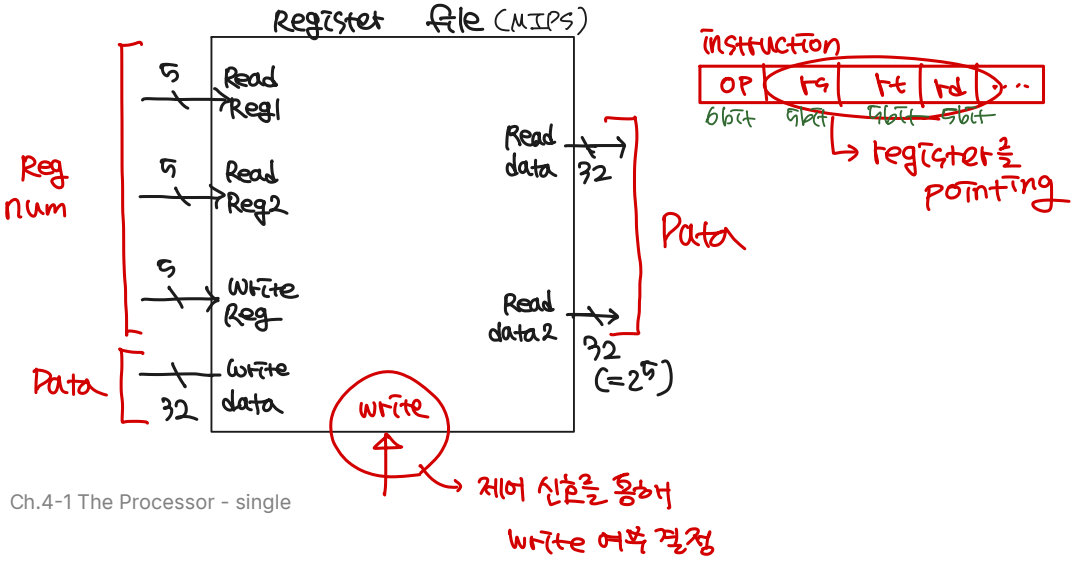


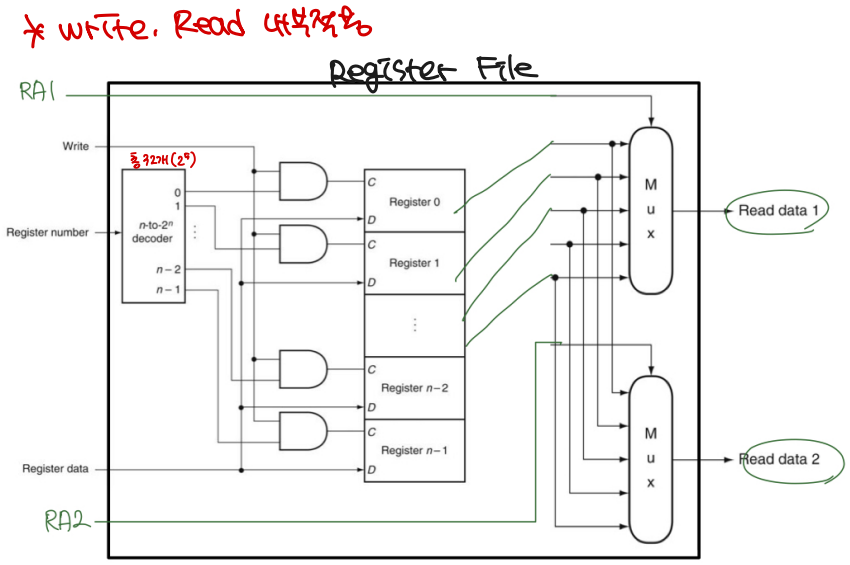
▼ Register File

Process의 범용 register 32개는 register file 속에 들어 있다!

⇒ register을 모아 놓은 것

→ file 내의 register 번호를 지정하면 어느 register라도 읽고 쓸 수 있다

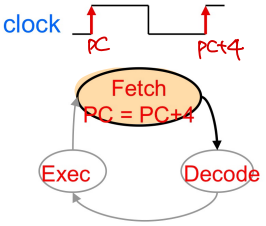




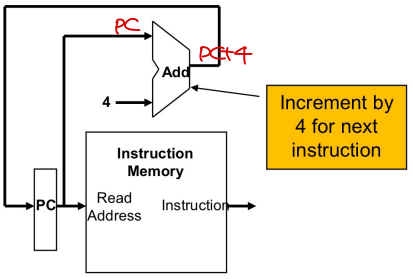
▼ Datapath

앞서 나온 ALU로 CPU의 data path 어떻게 build?

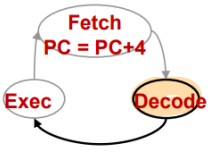
- Datapath → registers, ALUs, mux's, memories...
- MIPS datapath를 알아보자

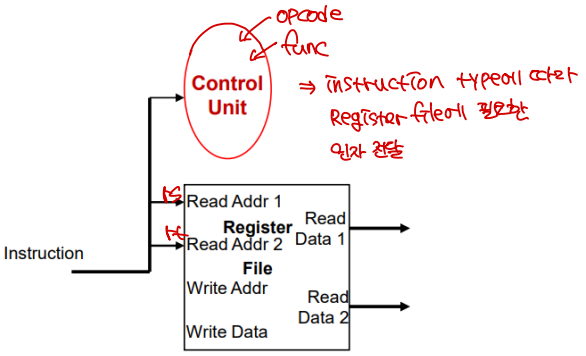


1. Fetching instructions
   i. Instruction memory로부터 instruction 읽어옴
   ii. 다음 instruction의 주소를 PC에 update PC ← PC+4
   iii. PC → 매 clock cycle마다 update
      - Control signal 필요하지 않고 clock signal로만 운영



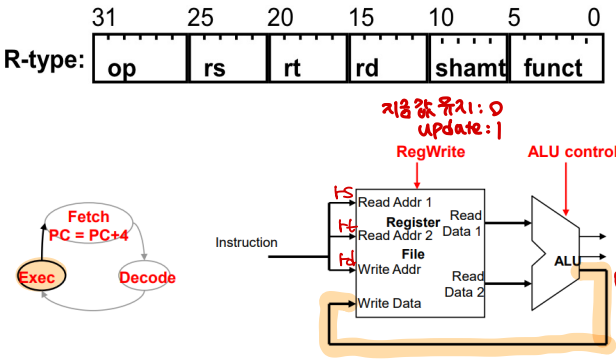
2. Decoding instructions





   i. Fetched된 instruction의 opcode와 function field bit를 control unit에 전달
   ii. Register file로부터 두 개의 value(rs, rt) read

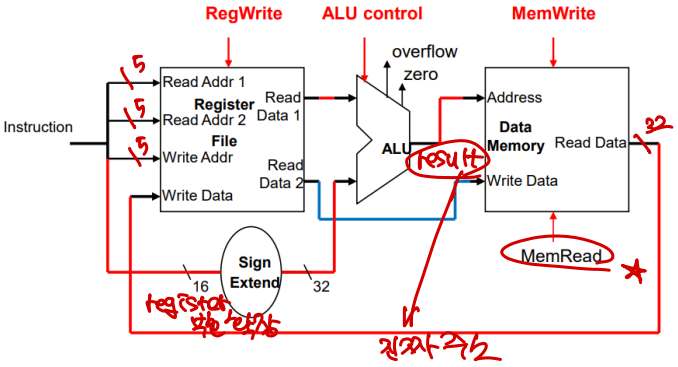
3. Executing Operations
   i. **R-format** operation(add, sub, slt, and, or)



- op, funct로 알게될 operation ⇒ rs, rt에서 op 실행
- Register file에 결과 저장(rd)
- 매 cycle마다 register file에 write하지 않아도 됨
  - Register file에 write control signal 필요

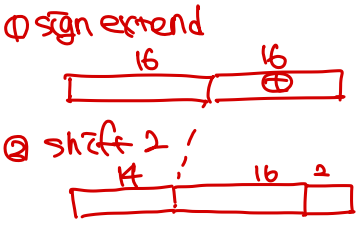
(store word는 write 필요 X)
→ 필요할 때만 허용됨

ii. **Lw/sw** operation

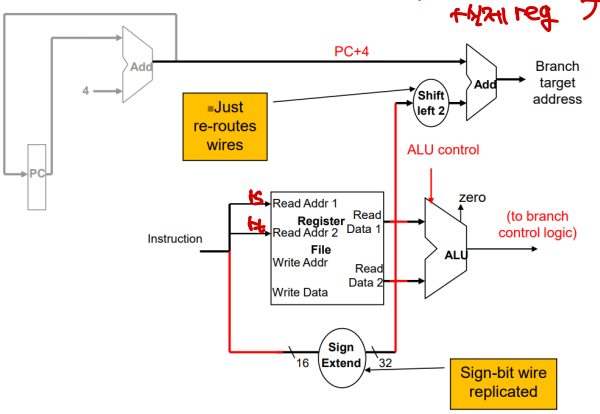


- Instruction에 16bit 부호 확장된 offset field에 base register 추가함으로써 ⇒ memory address 계산
- **Load** : data memory로부터 data read + register file에 update
- **Store** : data memory에 register value write

iii. Branch operation



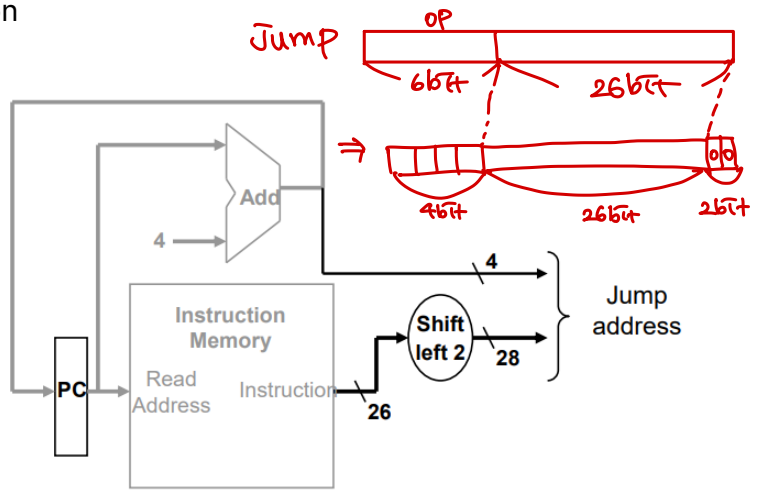
update된 PC값 ↑ 저장 reg → branch target addr



- decode 하는 도중에 register file로부터 operand 읽음
- 비교연산자 제공 ⇒ ALU 사용해서 subtract ⇒ zero를 check
  - 0 : bne
  - 1 : beq
    - ↳ zero가 true
- branch target address 계산 (in MIPS)
  - sign-extended
  - shift left 2
  - PC+4 update
    - ↳ instruction fetch에 의해 이미 계산됨

이유?
⇒ 워드 단위로 2씩 증가시킴

iv. jump operation



- PC : opcode(6bit) + 26bit

⇒ PC : opcode(4bit) + 26bit + shift 2bit(00)로 변경

full version 완주해보자!!

▼ Single Datapath

→ fetch, decode, excute가 한 cycle에 하나씩 일어난다고 가정해보자

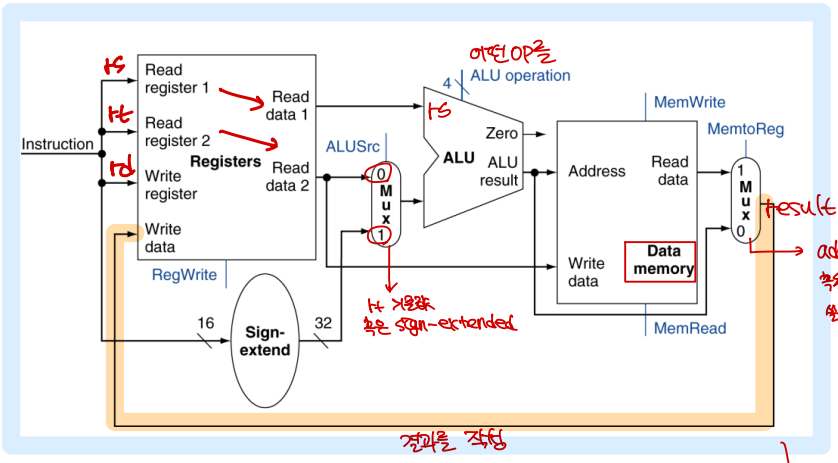
- 어떤 datapth 자원도 instruction 당 두 번 이상 사용될 수 없음
  - 두 번 이상 사용해야 한다면 여러 개 두어야 함 ⇒ duplicated

  ex) instruction mem과 data mem이 분리, 여러 개의 adder
- mux : 여러 개의 input src 중 선택하는 control을 해야 할 때 사용
- register file과 data mem의 writing을 조절하는 signal도 써야 함 → 필요할 때만 data update

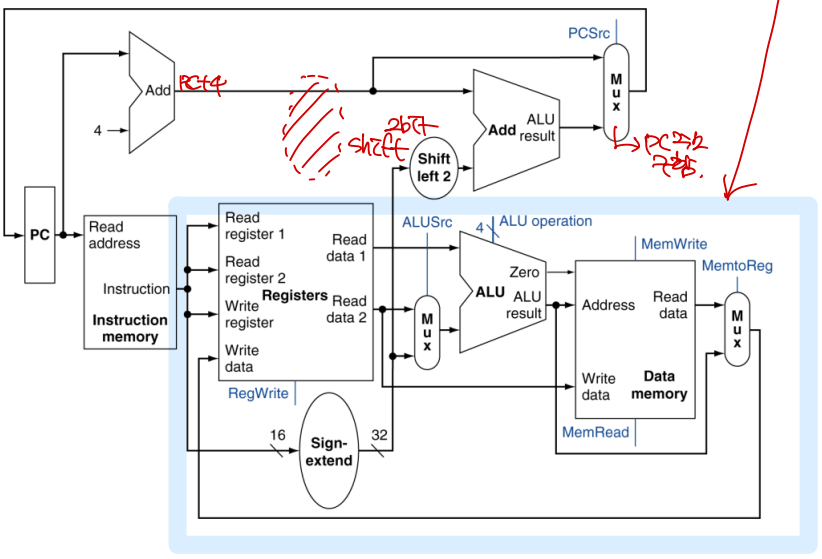
⇒ cycle time이 가장 길게 끝나는 instruction 시간에 맞추게 됨

↳ length of the longest path

- R-type + LW/SW datapth

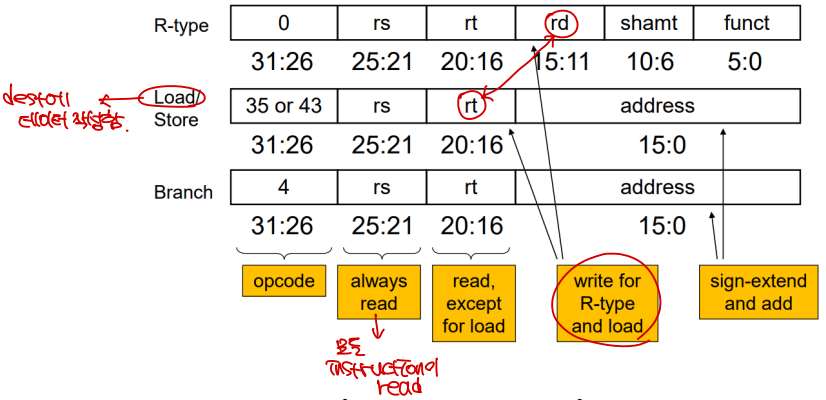


- full(모두 지원 가능)



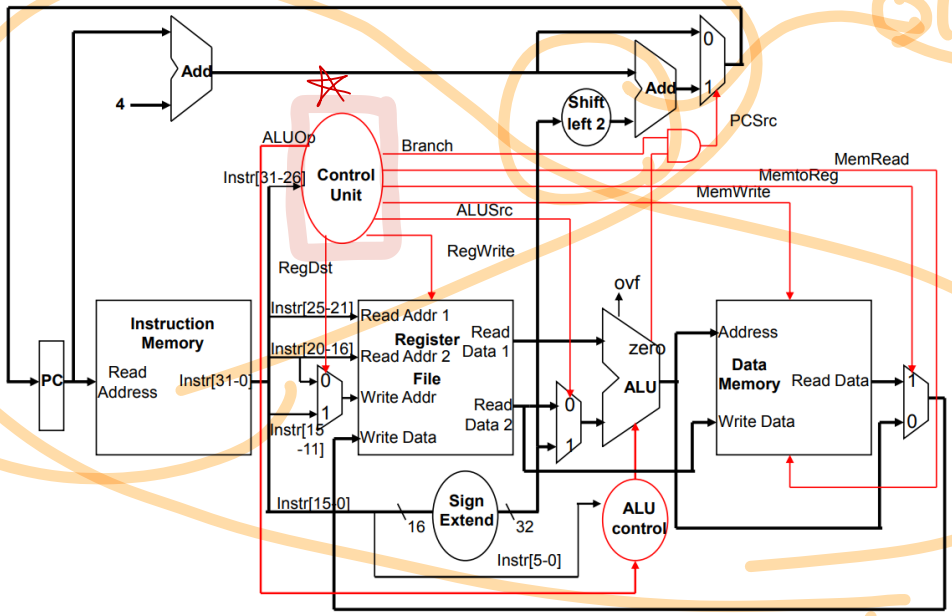
▼ Main Control Unit

instruction에 의해 control signal 결정됨

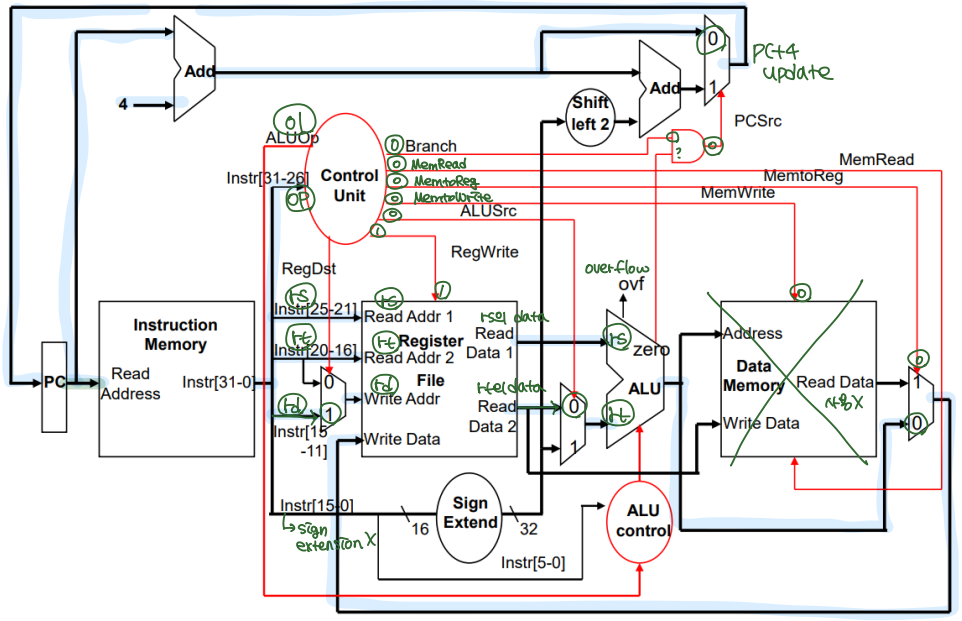


▼ Single Cycle Datapth with Control unit

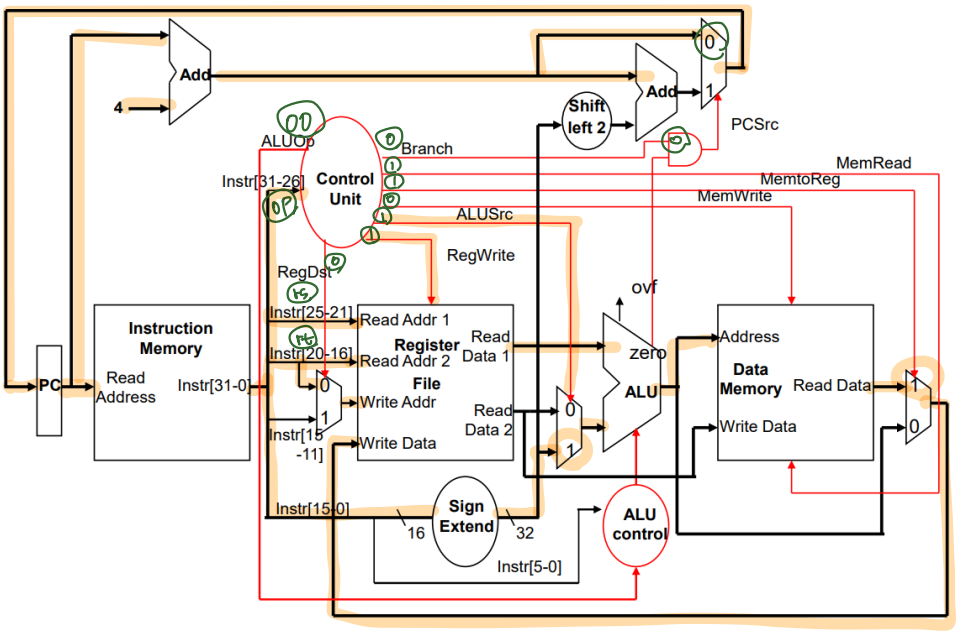
- 기본 상태


Add
4
Add
Shift left 2
0
1
PCSrc
ALUOp
Control Unit
Instr[31-26]
Branch
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
RegDst
ovf
Instruction Memory
Read Address
Instr[31-0]
PC
Instr[25-21]
Instr[20-16]
Instr[15-11]
Instr[15-0]
Register File
Read Addr 1
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
zero
ALU
Address
Data Memory
Read Data
Write Data
Sign Extend
16
32
ALU control
Instr[5-0]

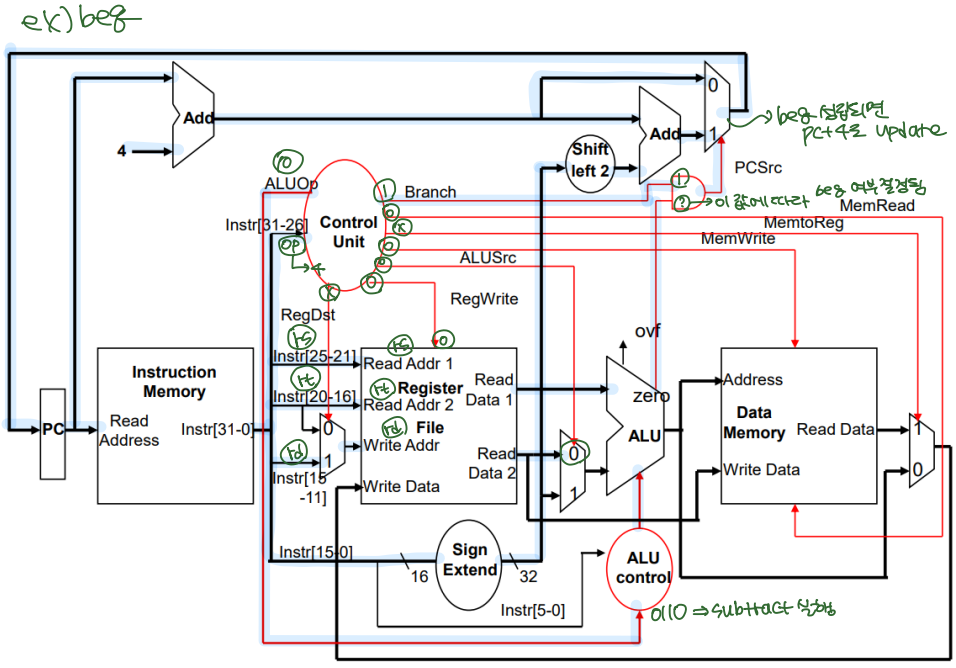
- R-type


Add
4
Add
Shift left 2
PCSrc
ALUOp
Control Unit
Instr[31-26]
Branch
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
RegDst
ovf
Instruction Memory
Read Address
Instr[31-0]
PC
Instr[25-21]
Instr[20-16]
Instr[15-11]
Instr[15-0]
Register File
Read Addr 1
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
zero
ALU
Address
Data Memory
Read Data
Write Data
Sign Extend
16
32
ALU control
Instr[5-0]

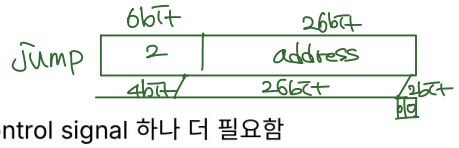
- lw


Add
4
Add
Shift left 2
PCSrc
ALUOp
Branch
Instr[31-26]
Control Unit
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
RegDst
ovf
Instruction Memory
Read Address
Instr[31-0]
PC
Instr[25-21]
Instr[20-16]
Instr[15-11]
Instr[15-0]
Read Addr 1
Register
Read Addr 2
File
Write Addr
Write Data
Read Data 1
Read Data 2
zero
ALU
Address
Data Memory
Read Data
Write Data
Sign Extend
16
32
ALU control
Instr[5-0]

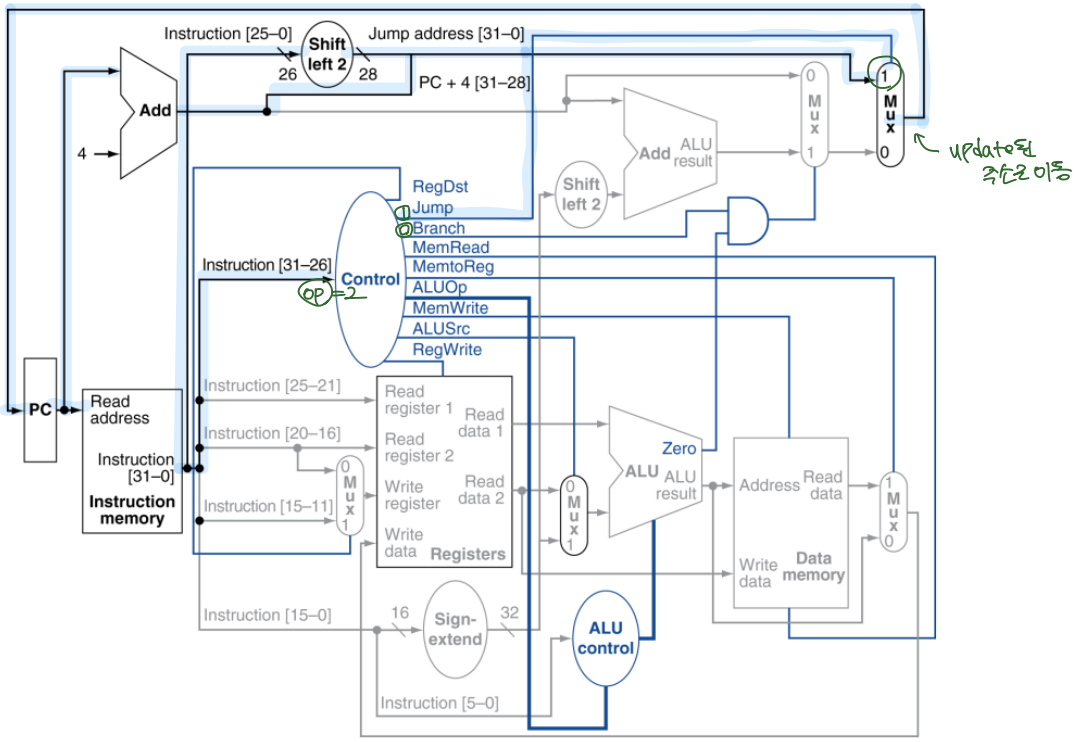
- Branch


Add
4
Add
Shift left 2
PCSrc
ALUOp
Branch
Instr[31-26]
Control Unit
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
RegDst
ovf
Instruction Memory
Read Address
Instr[31-0]
PC
Instr[25-21]
Instr[20-16]
Instr[15-11]
Instr[15-0]
Read Addr 1
Register
Read Addr 2
File
Write Addr
Write Data
Read Data 1
Read Data 2
zero
ALU
Address
Data Memory
Read Data
Write Data
Sign Extend
16
32
ALU control
Instr[5-0]

- Jump → MUX가 하나 추가됨



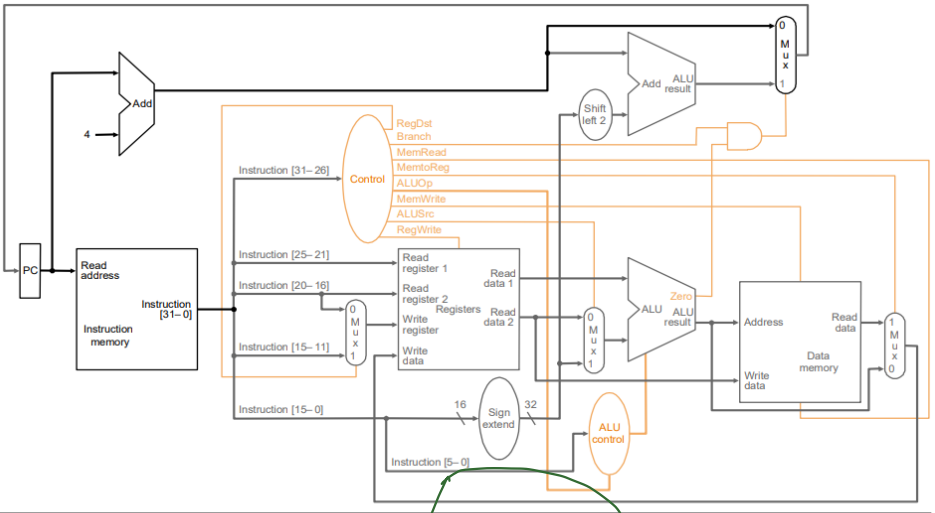
  - opcode를 decode할 때 control signal 하나 더 필요함



▼ Control

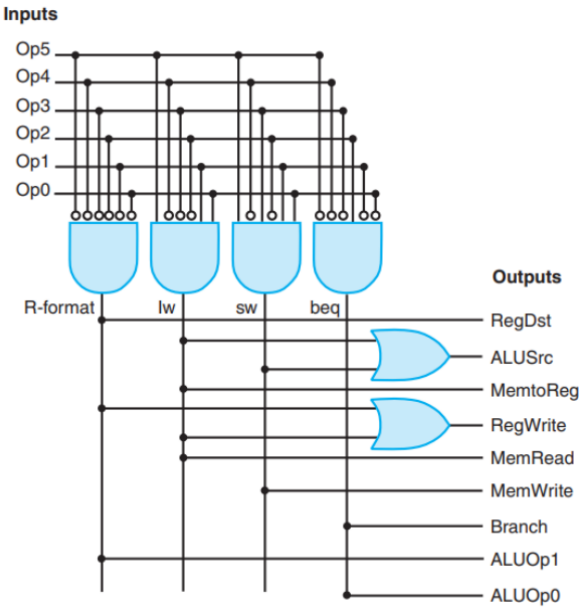
- effect of control signals (표 한글로 해석하자잉

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20–16).	The register destination number for the Write register comes from the rd field (bits 15–11).
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, lower 16 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	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

- Control Function



Name	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-format	0_{ten}	0	0	0	0	0	0
lw	35_{ten}	1	0	0	0	1	1
sw	43_{ten}	1	0	1	0	1	1
beq	4_{ten}	0	0	0	1	0	0

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

▼ ALU control

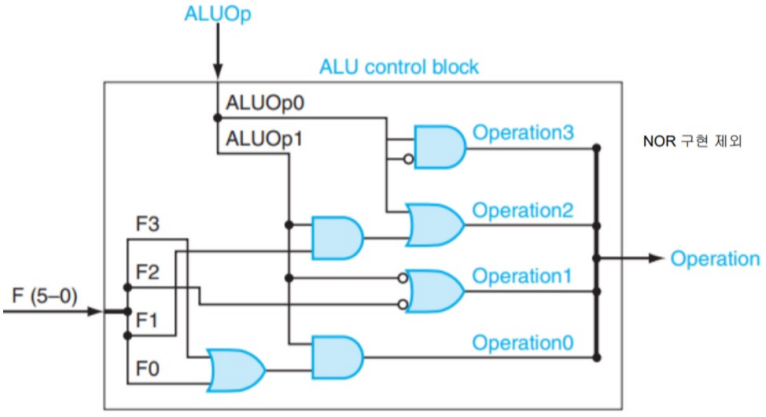
ALU는 어디에 사용?

- Lw/Sw : Func = add
- Branch : Func = subtract
- R-type : Func = funct field에 따라 달라짐

- 3-bit ALU control input 가진다고 해보자

OP code	ALU op	Inst. operation	Function field	ALU operation	ALU control
LW	00	load word	xxxxxx	add	0010
SW	00	store word	xxxxxx	add	0010
Branch equal	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

- ALU control Operation



ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111