

# MiniMIPS 기반

## XOR / Shift / Rotate

### 연산 속도 개선을 위한 CPU

# 목차

---

01	주제 선정 및 목표	05	ALU	09	구현
02	설계 철학	06	Control signal	10	비교
03	instruction	07	Instruction Datapath	11	결론
04	Datapath	08	Control State Machine	12	

# 01. 주제 선정 및 목표

# 01. 주제 선정 및 목표

---

## XOR, Shift, Rotate

암호화, 난수 생성 등에서 반복적으로 사용되는 연산

## XOR, Shift, Rotate의 비중이 높은 부분

보안 분야에서 사용되는 알고리즘, 난수 생성 알고리즘에서 해당 연산 비중이 높음

ChaCha20 : 암호 알고리즘. XOR, Rotate 연산 비중 높음.

SHA-256 : 해시 알고리즘. XOR, Shift, Rotate 연산 비중 높음.

Xorshift : 난수 생성 알고리즘. XOR, Shift 연산 비중 높음.

xoshiro : 난수 생성 알고리즘. XOR, Shift, Rotate 연산 비중 높음.

→ 새로운 명령어와 CPU 구조를 통해 성능 개선

# 01. 주제 선정 및 목표

---

## 목표

XOR, Shift, Rotate 활용 알고리즘들의 성능 개선

## Target Application

**VPN** : ChaCha20을 핵심 암호화 알고리즘으로 사용

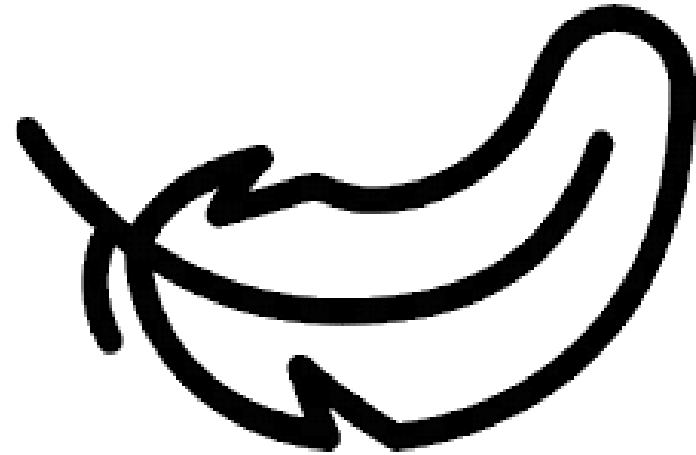
**보안/인증 시스템** : SHA-256 해시를 이용한 데이터 무결성 검증

**난수 생성 이용 프로그램** : XorShift, Xoshiro를 이용한 난수 생성

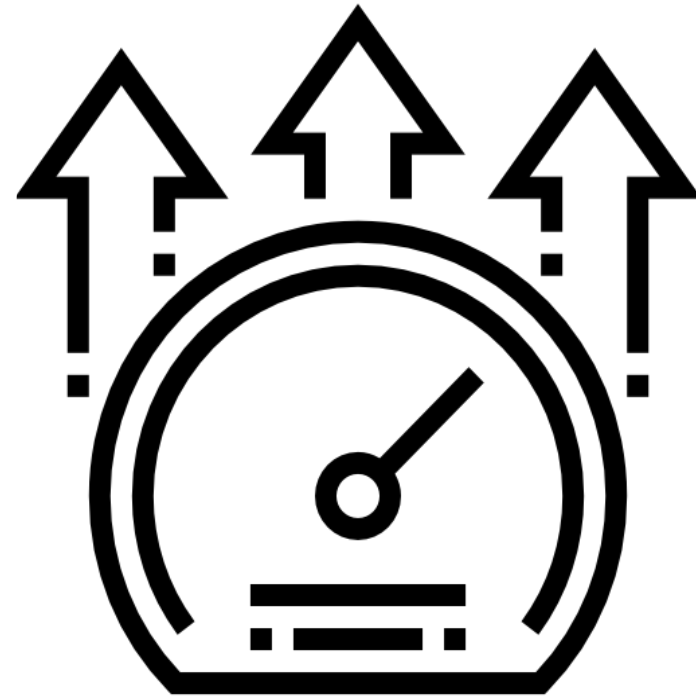
## 02. 설계 철학

## 02. 설계 철학

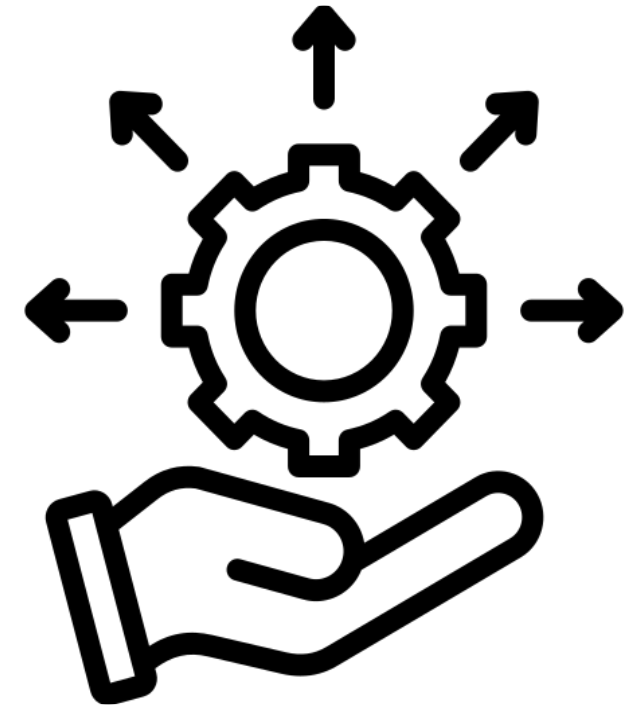
---



가벼움



빠른 속도



범용성

# 03. Instruction



# 03. Instruction

Type	Instruction	Usage
R	ADD	add rd, rs, rt
	SUB	sub rd, rs, rt
	AND	and rd, rs, rt
	OR	or rd, rs, rt
	XOR	xor rd, rs, rt
	NOR	nor rd, rs, rt
	ROT	rot rd, rs, rt
	SLL	sll rd, rs, rt
	SRL	srl rd, rs, rt
	SRA	sra rd, rs, rt
	JR	jr rs
	SYSCALL	Syscall
	SLXO	slxo rd, rs, rt, sh
	SRXO	srxo rd, rs, rt, sh
	SLT	slt rd, rs, rt

Type	Instruction	Usage
D	DXOR	dxor rs, rt, ra, rb
I	ADDI	addi rt, rs, imm
	SUBI	subi rt, rs, imm
	ANDI	andi rt, rs, imm
	ORI	ori rt, rs, imm
	XORI	xori rt, rs imm
	LW	lw rt, imm(rs)
	SW	sw rt, imm(rs)
	BEQ	beq rs, rt, loop
	BNE	bne rs, rt, loop
	SLTI	slti rt, rs, imm
J	J	j jta
	JAL	jal jta

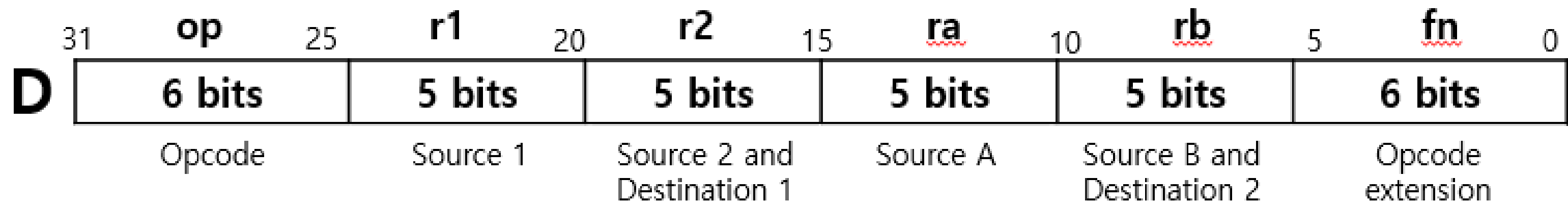
## 03. Instruction

---

### D-type 추가

기존 type 들로는 레지스터 4개 다룰 수 없음.

연산을 반복적으로 시행해야 될 때 필요에 따른 명령어 확장 가능  
ex) DADD, DSUB 등

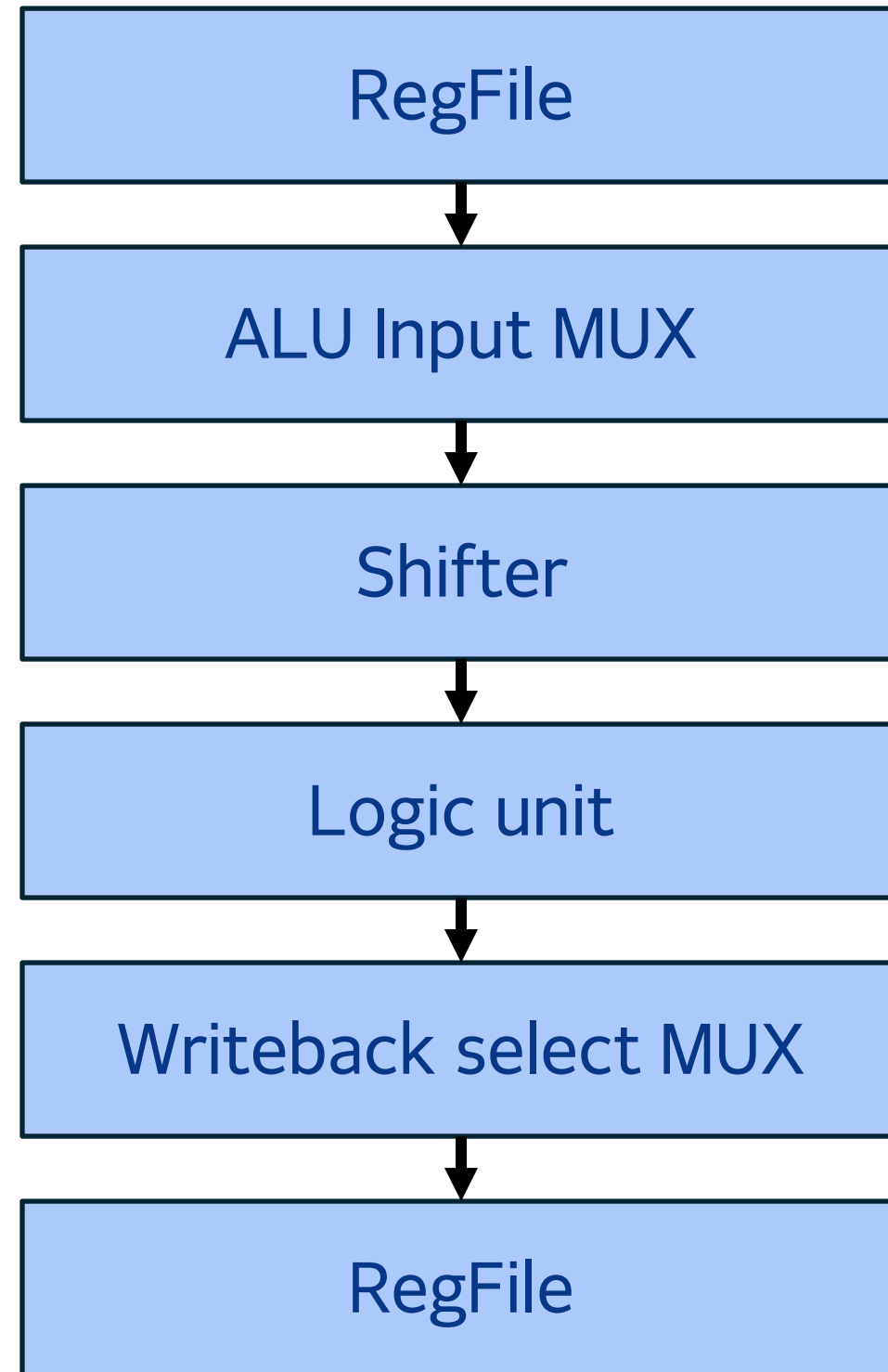


## 03. Instruction

Instruction	Usage	Opcode / fn	Type	Description
Rotate	rot rd, rs, rt	000000	R-type	rd = rs를 rt에 들어 있는 값만큼 왼쪽 rotate
		000000		
Shift left XOR	slxo rd, rs, rt, sh	000000	R-type	rd = (rs를 sh만큼 왼쪽 shift) ^ rt (MSB bit 버려지고 LSB 0으로 채워짐)
		101000		
Shift right XOR	srxo rd, rs, rt, sh	000000	R-type	rd = (rs를 sh만큼 오른쪽 shift) ^ rt (LSB bit 버려지고 MSB 0으로 채워짐)
		101001		
Double XOR	dxor r1, r2, ra, rb	000000	D-type	r2 = r1 ^ r2 rb = ra ^ rb
		110010		

# 04. Datapath

## 04. Datapath



Single cycle에서 SLXO/SRXO의 연산 경로  
(ALU 구조 변경 시)

### Single Cycle 선택 시

① SLXO / SRXO 복합 연산 명령어  
→ critical path ↑ 😞

② DXOR  
→ XOR 반복보다 속도 개선 미미 😞

## 04. Datapath

---

```
sll  $t1, $t0, 13    # x<<13
xor  $t0, $t0, $t1    # x ^= (x<<13)
```

```
srl  $t1, $t0, 17    # x>>17
xor  $t0, $t0, $t1    # x ^= (x<<17)
```

```
sll  $t1, $t0, 5      # x<<5
xor  $t0, $t0, $t1    # x ^= (x<<5)
```

---

```
# result = rotl(s0 + s3, 7) + s0
addu $t4, $t0, $t3    # t4 = s0 + s3
sll  $t5, $t4, 7      # (s0+s3) << 7
srl  $t6, $t4, 25     # (s0+s3) >> 25
or   $t5, $t5, $t6    # rotl
addu $t7, $t5, $t0    # result
```

### Pipeline 선택 시

① Data hazard 다수 발생

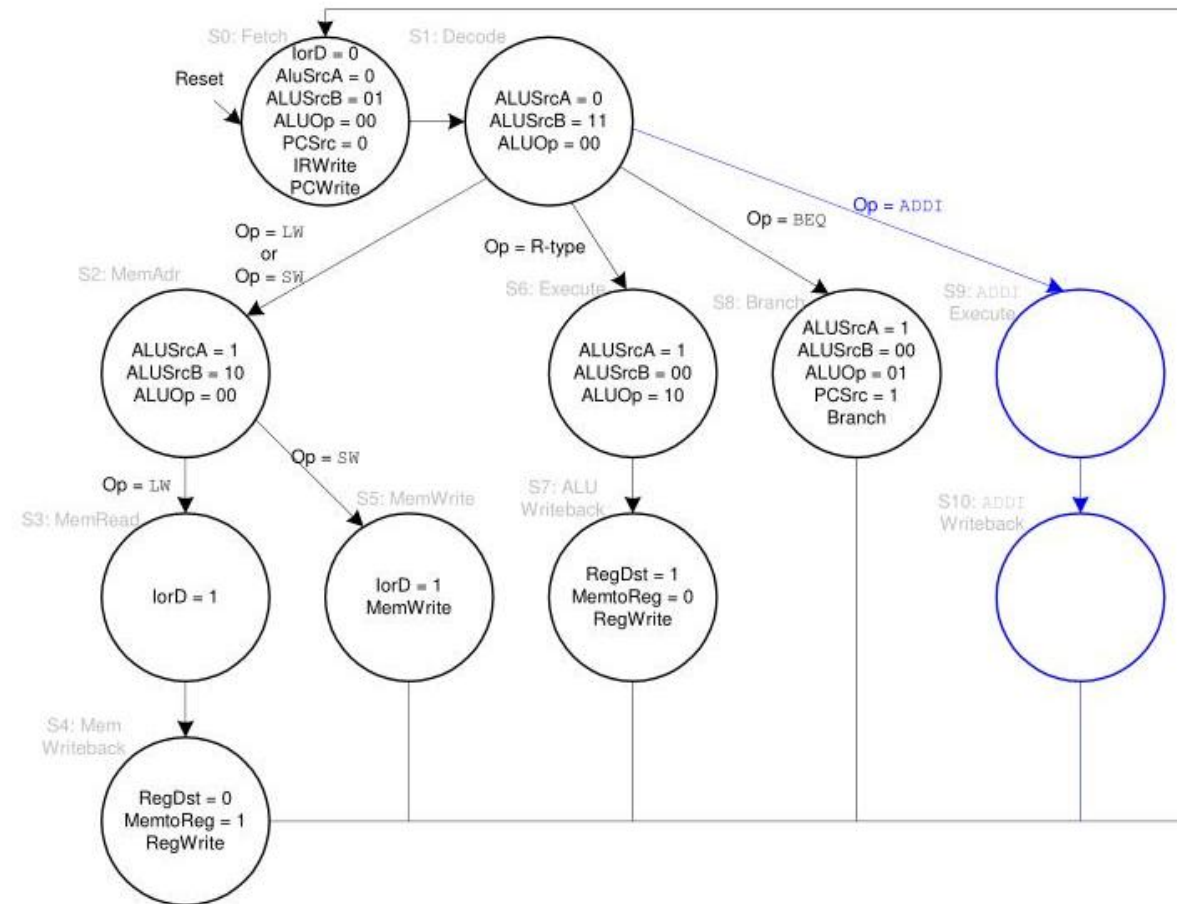
→ 매 사이클 Bubble or Forwarding 필요 😞

② 복합 연산 명령어

→ Execute 단계: 다른 단계와 분량 일치 X

→ Execute 단계에 맞춘 clock 필요 😞

# 04. Datapath



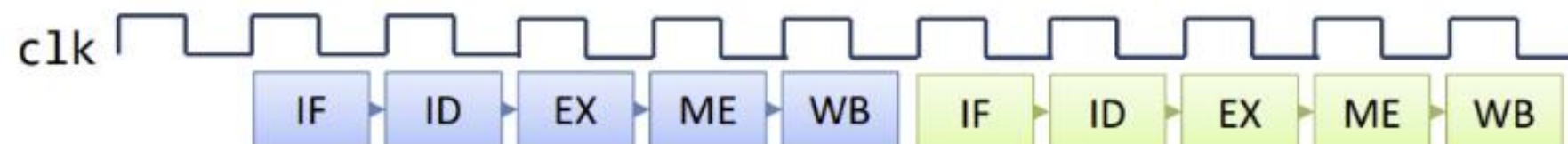
## Multi Cycle 선택 시

- ① 복합 연산 명령어  
→ critical path 늘리지 않음 😊
- ② 하드웨어 가볍게 가능 😊

### Single cycle execution

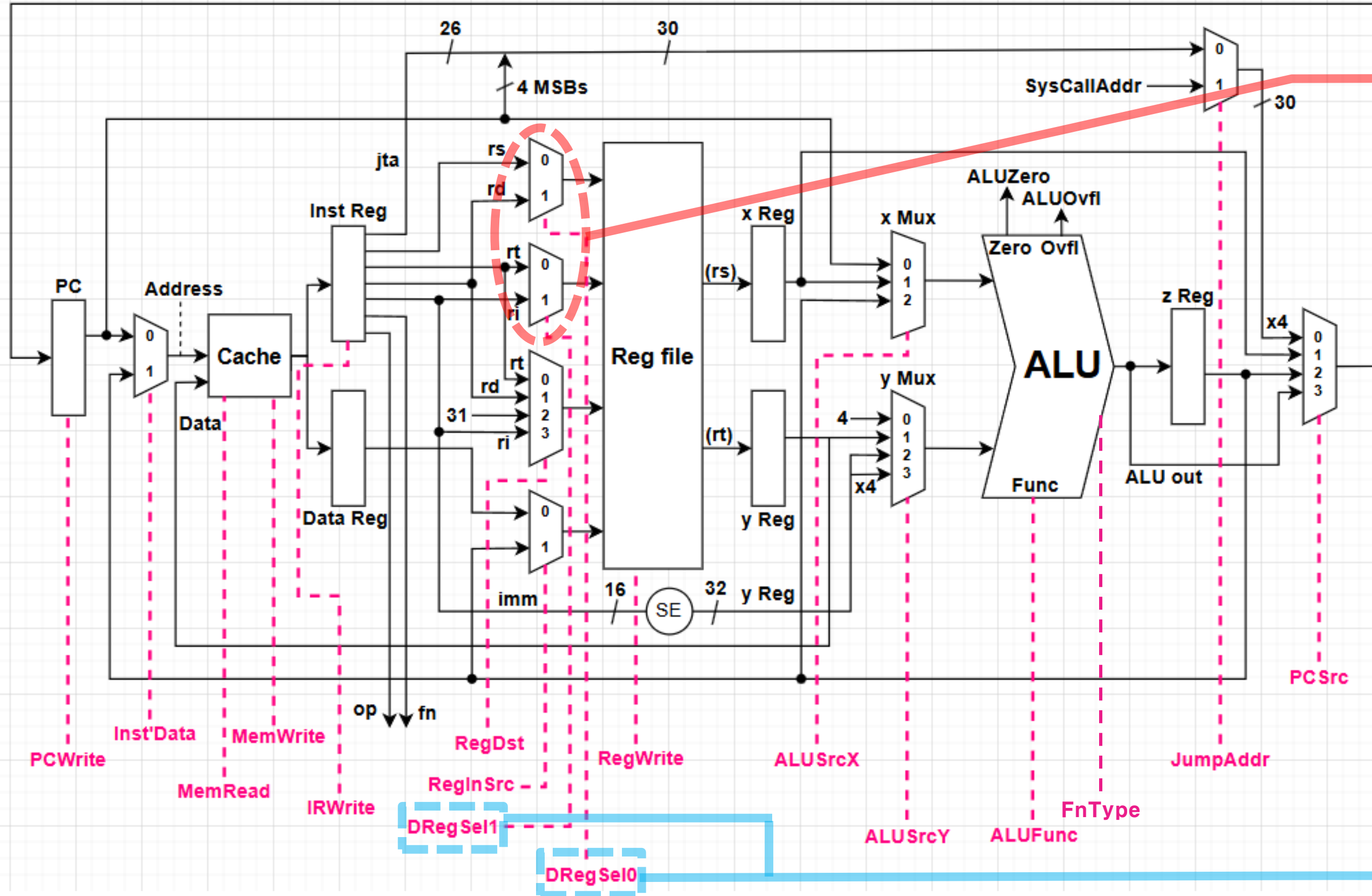


### Multi cycle execution



∴ Multi-cycle

## 04. Datapath

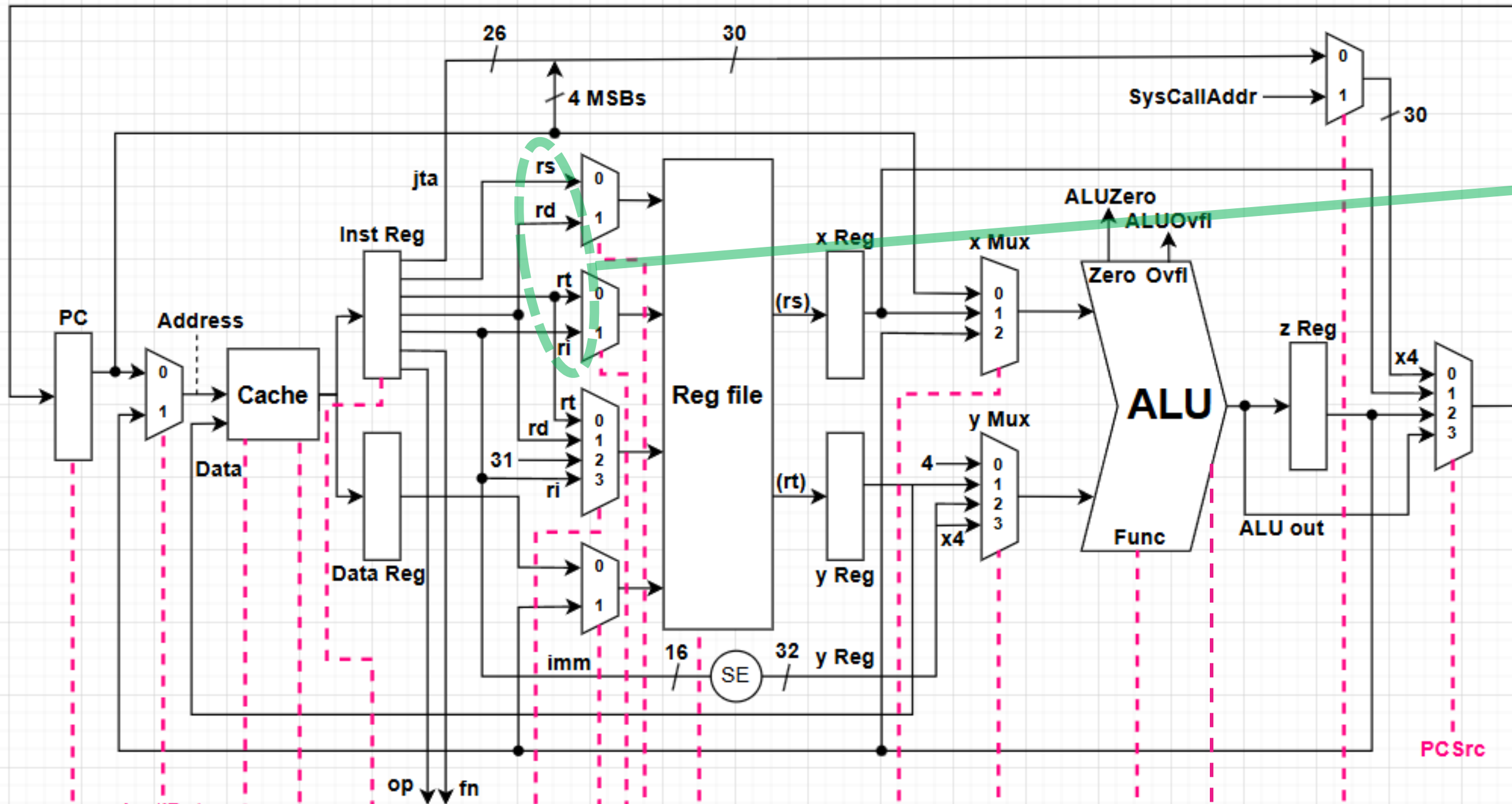


기존 miniMIPS의  
multicycle data path에서  
새롭게 만든 instruction을 위해  
Reg file 앞에 mux 2개 추가

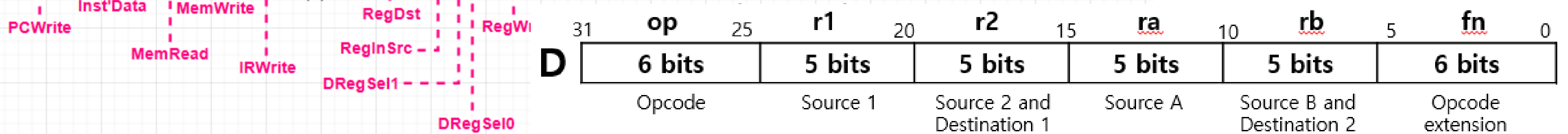
## 추가한 mux를 동작 시키기 위한 control signal 추가



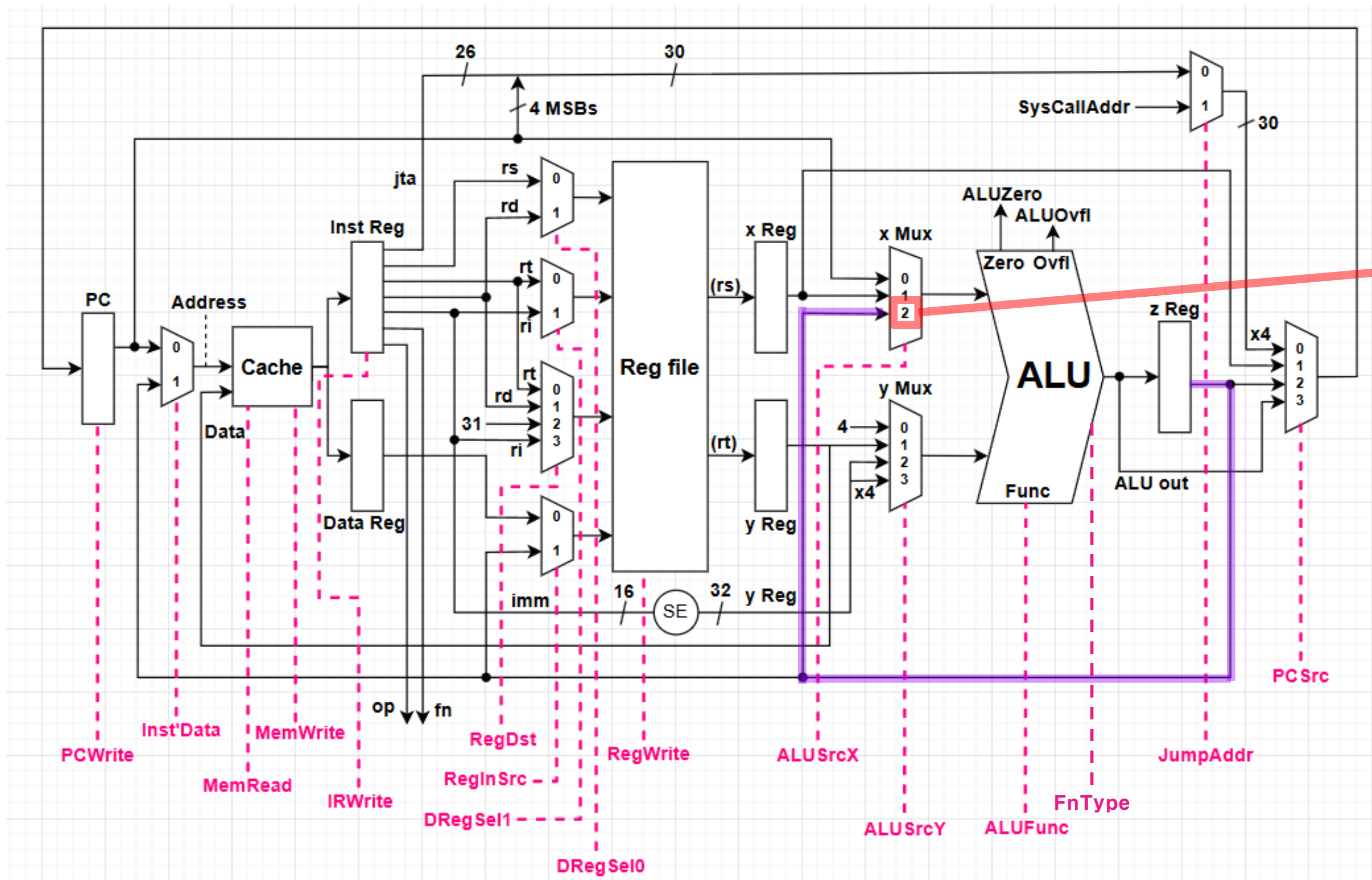
## 04. Datapath



D-type 동작 시,  
rs = r1  
rd = ra  
rt = r2  
ri = rb

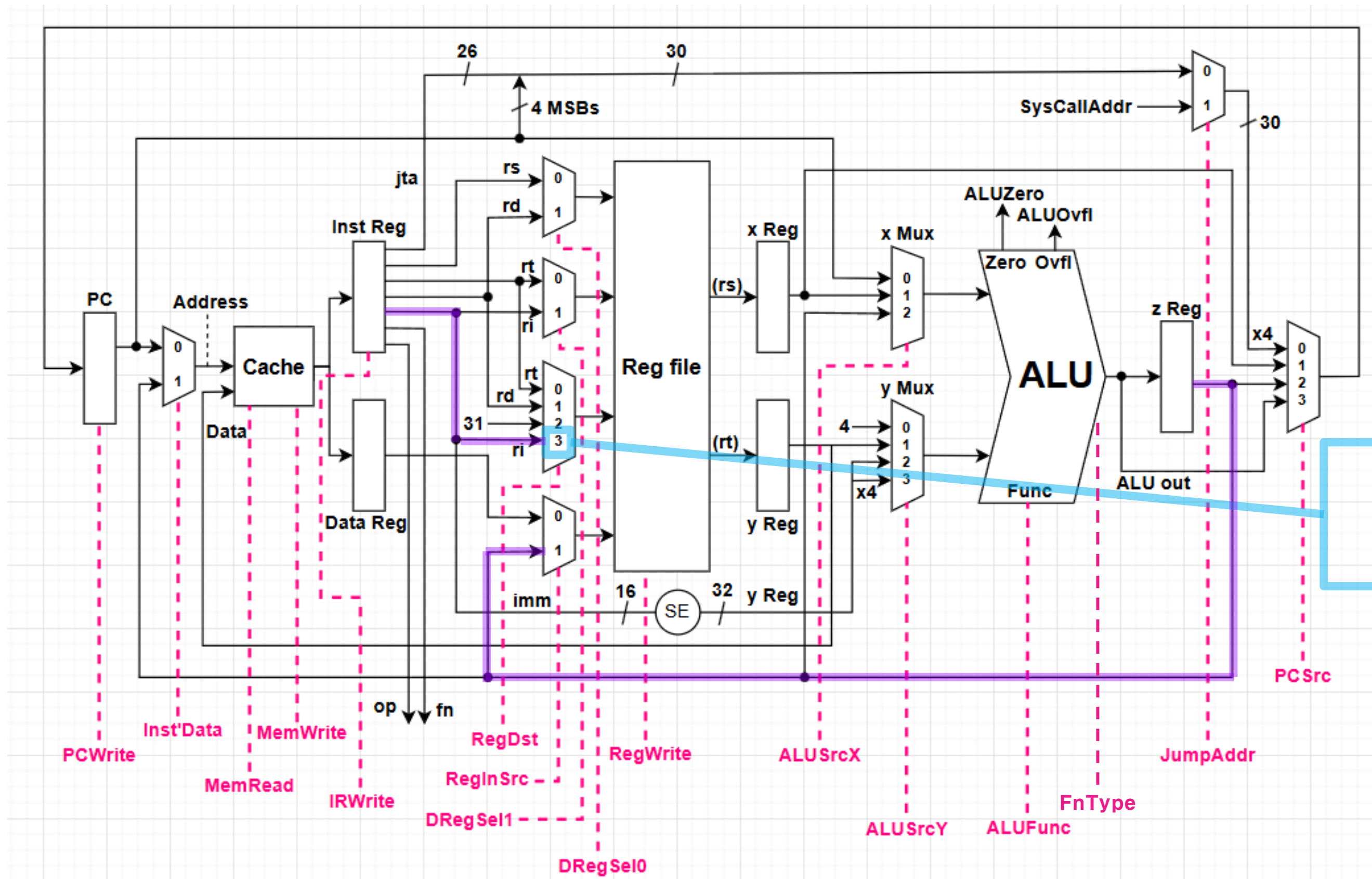


## 04. Datapath



## SLXO / SRXO 동작 시 Shift된 결과 XOR 시키기 위한 MUX 입력 추가

# 04. Datapath

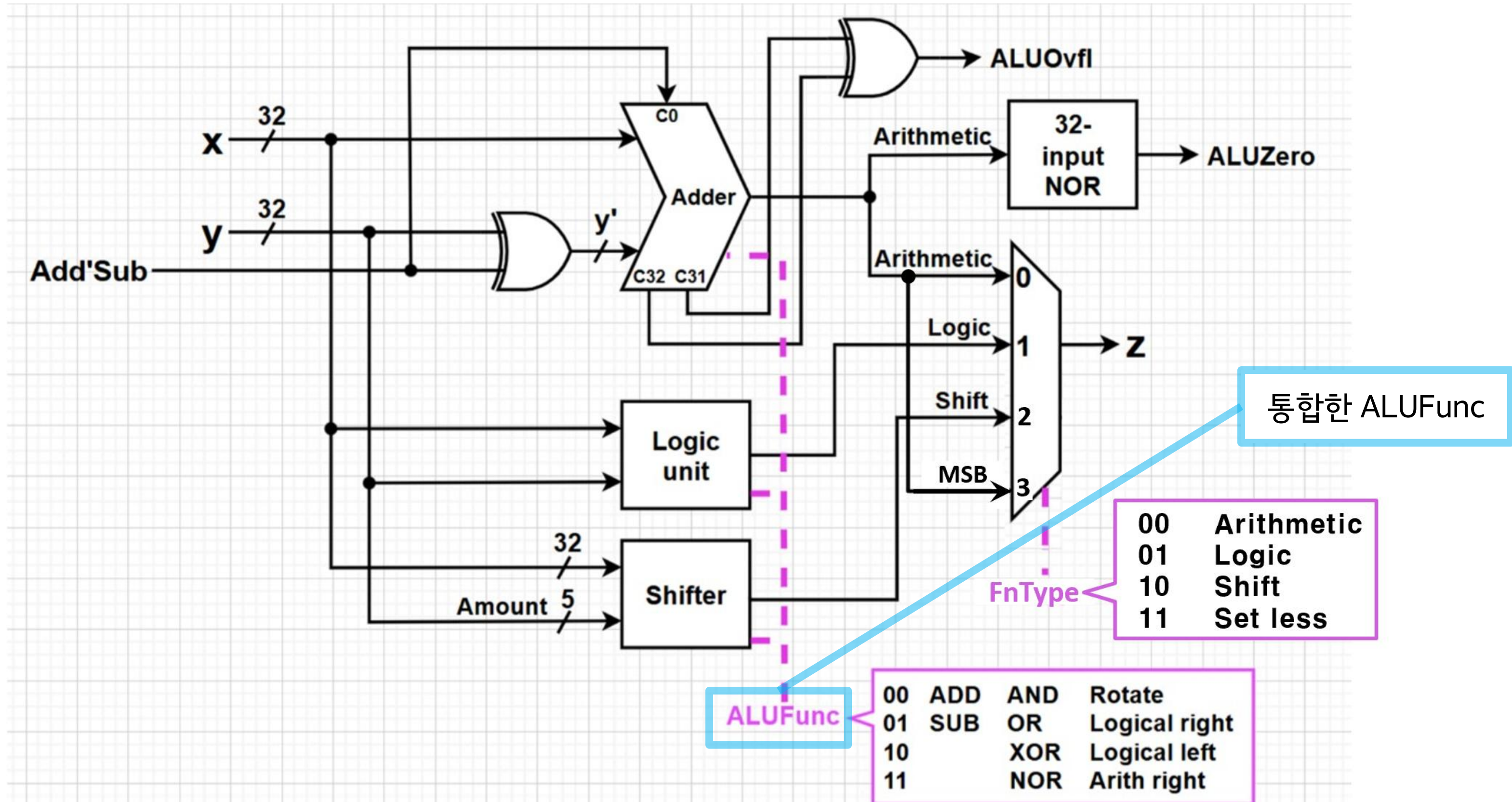


DXOR 동작 시 ri에 저장하기 위한  
MUX 입력 추가

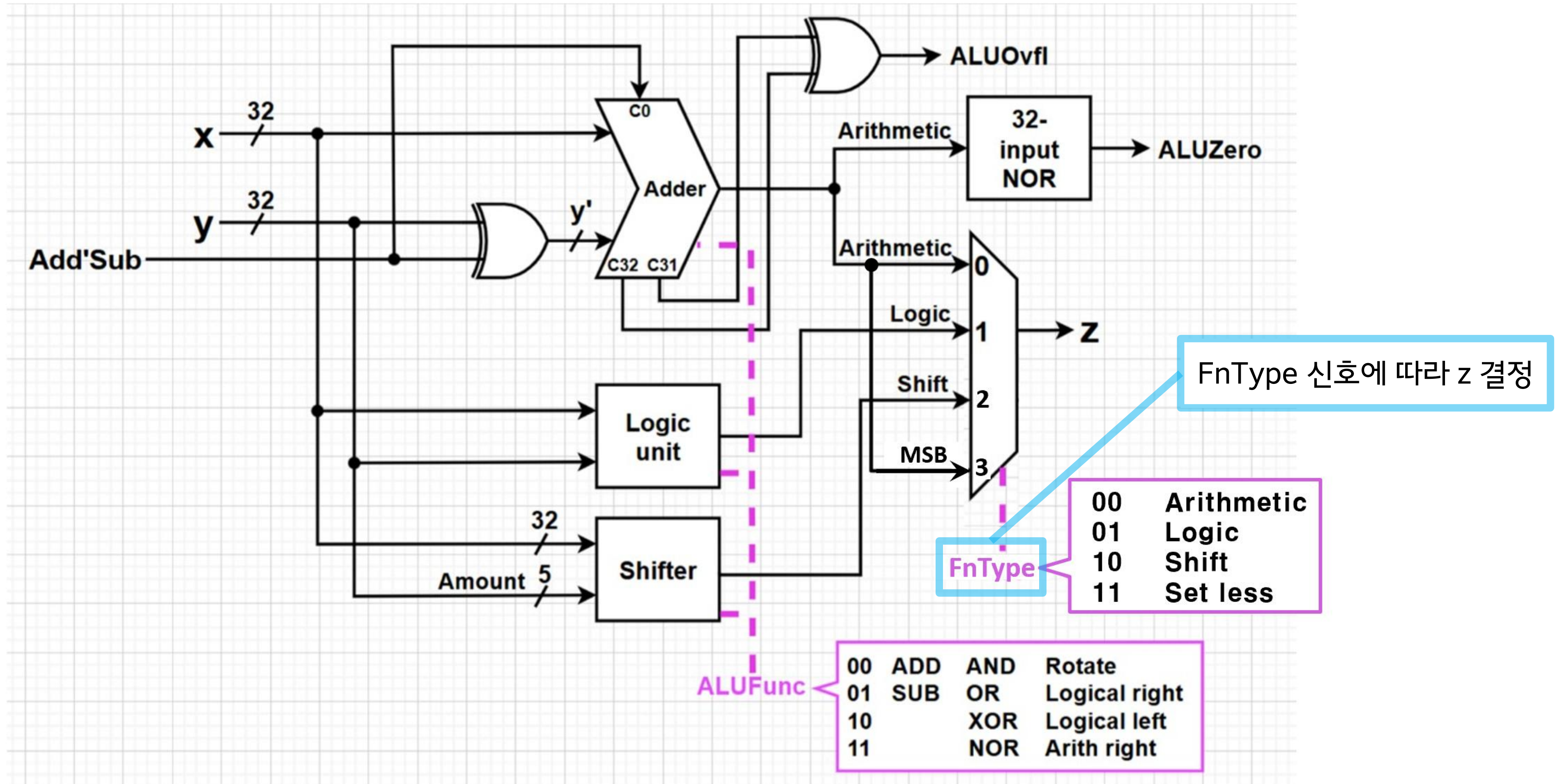
# 05. ALU



# 05. ALU

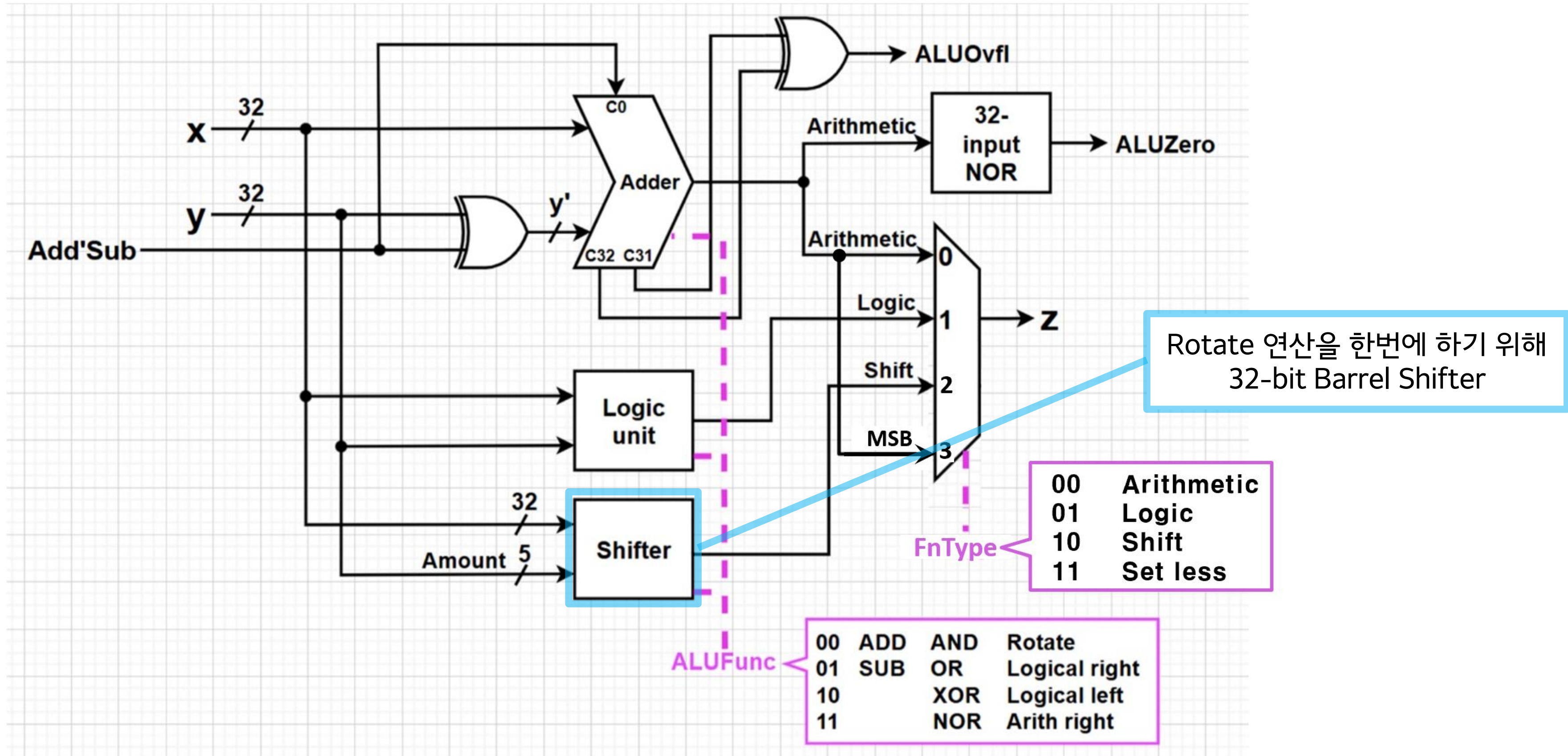


## 05. ALU

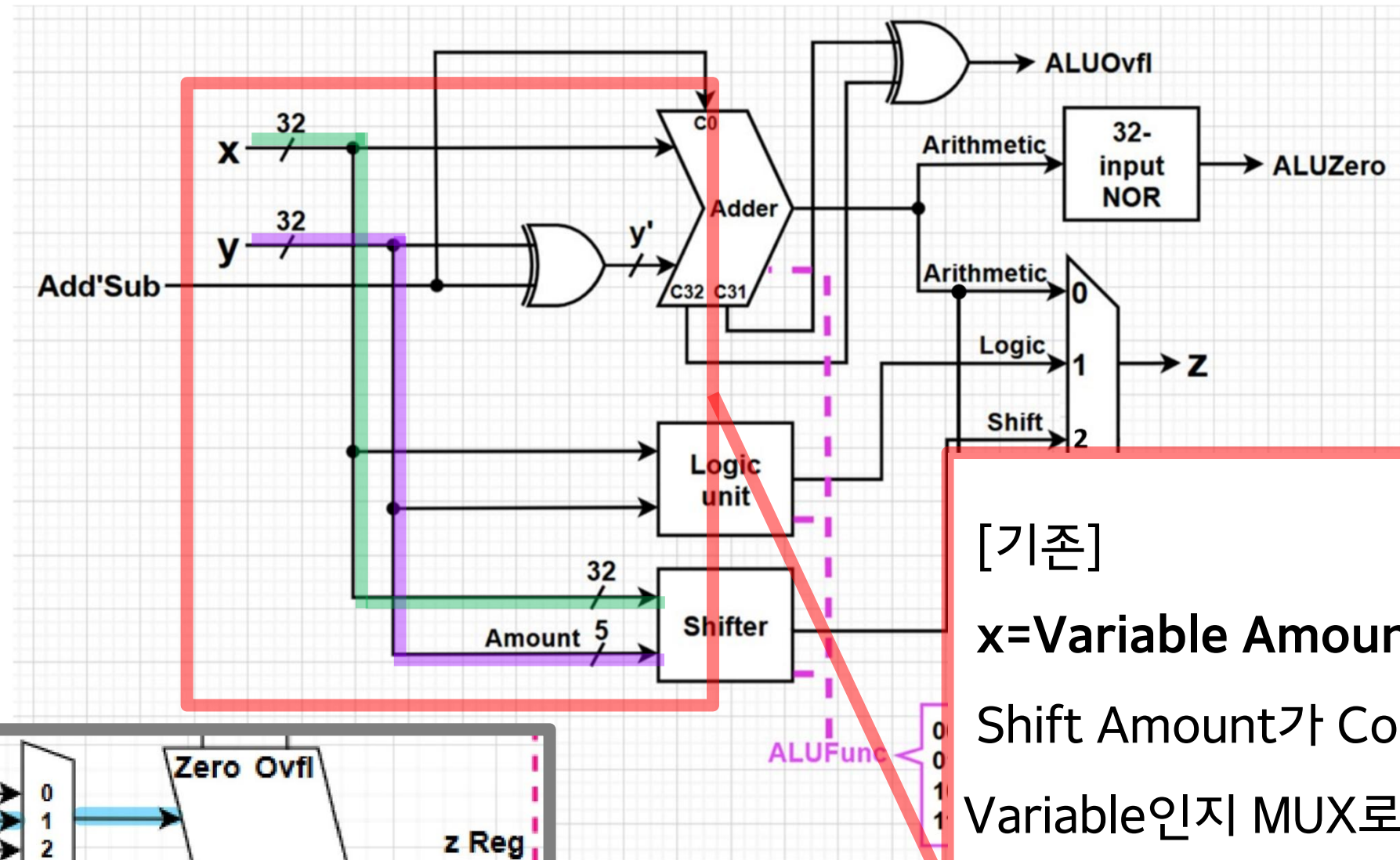
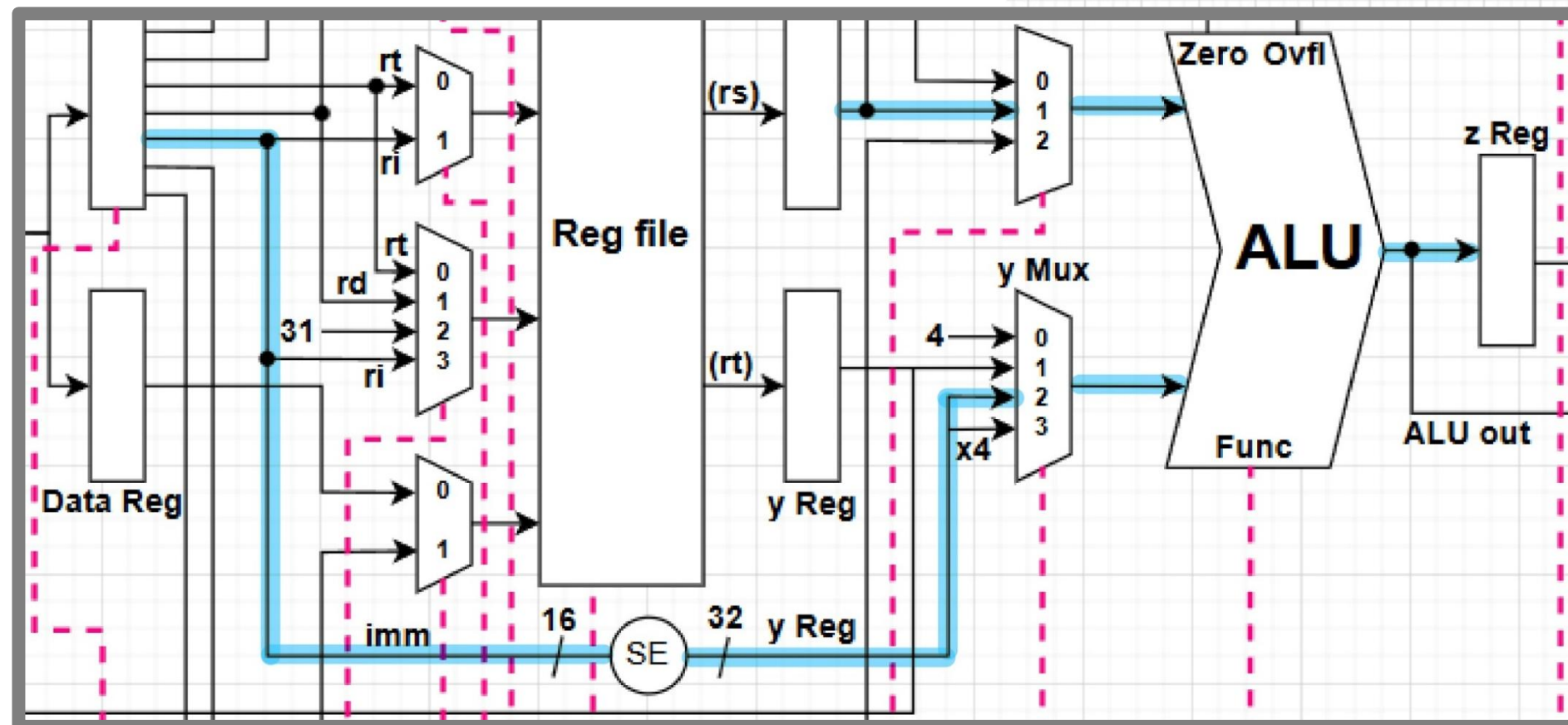
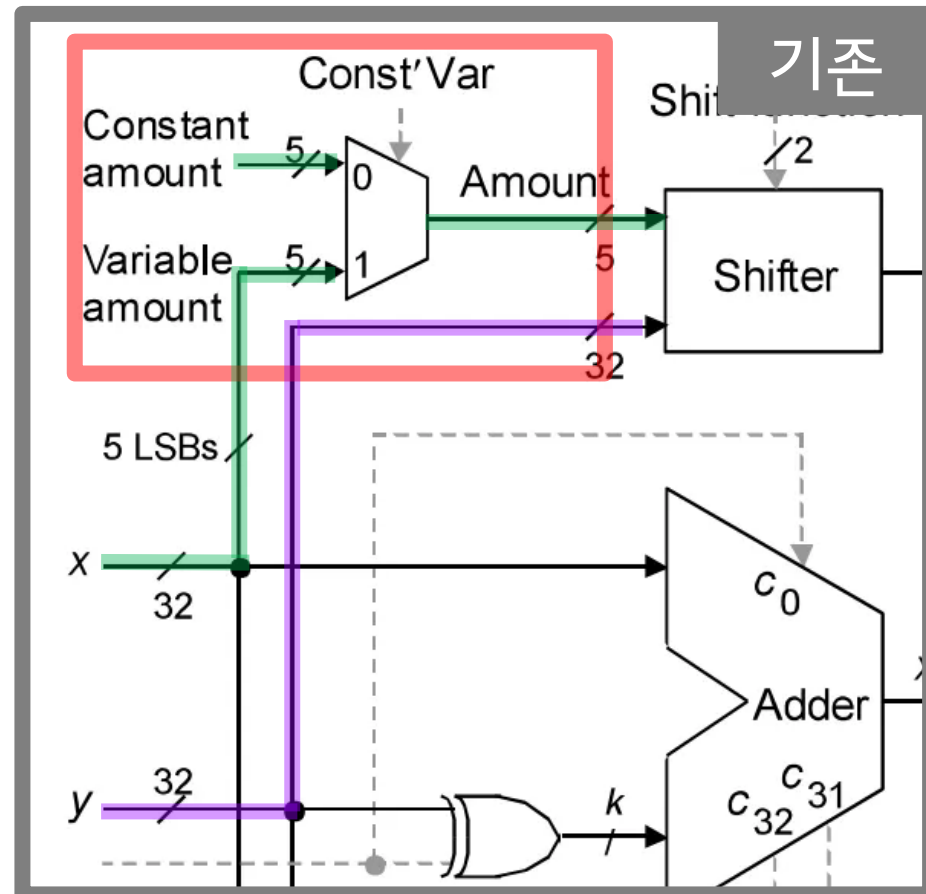




# 05. ALU



## 05. ALU



## [기존]

**x=Variable Amount, y=Shift할 대상**

## Shift Amount가 Constant인지

## Variable인지 MUX로 구별

## [변경 후]

**x=Shift할 대상, y=Shift Amount**

Shift Amount: imm라인 타고 y MUX로

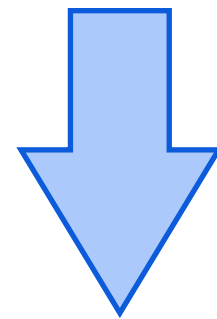
→ 넘어온 32 bit 중 5 bit만 잘라서 사용



## 05. ALU

기존

Instruction	Usage
Shift left logical	<u>sll</u> <u>rd</u> , <u>rt</u> , <u>sh</u>
Shift right logical	<u>srl</u> <u>rd</u> , <u>rt</u> , <u>sh</u>
Shift right arithmetic	<u>sra</u> <u>rd</u> , <u>rt</u> , <u>sh</u>



변경 후

Instruction	Usage
Shift left logical	<u>sll</u> <u>rd</u> , <u>rs</u> , <u>sh</u>
Shift right logical	<u>srl</u> <u>rd</u> , <u>rs</u> , <u>sh</u>
Shift right arithmetic	<u>sra</u> <u>rd</u> , <u>rs</u> , <u>sh</u>

[기존]

$y = R[rt]$ , Const amount = sh

$R[rd] = (R[rt] \text{를 } sh \text{만큼 shift})$

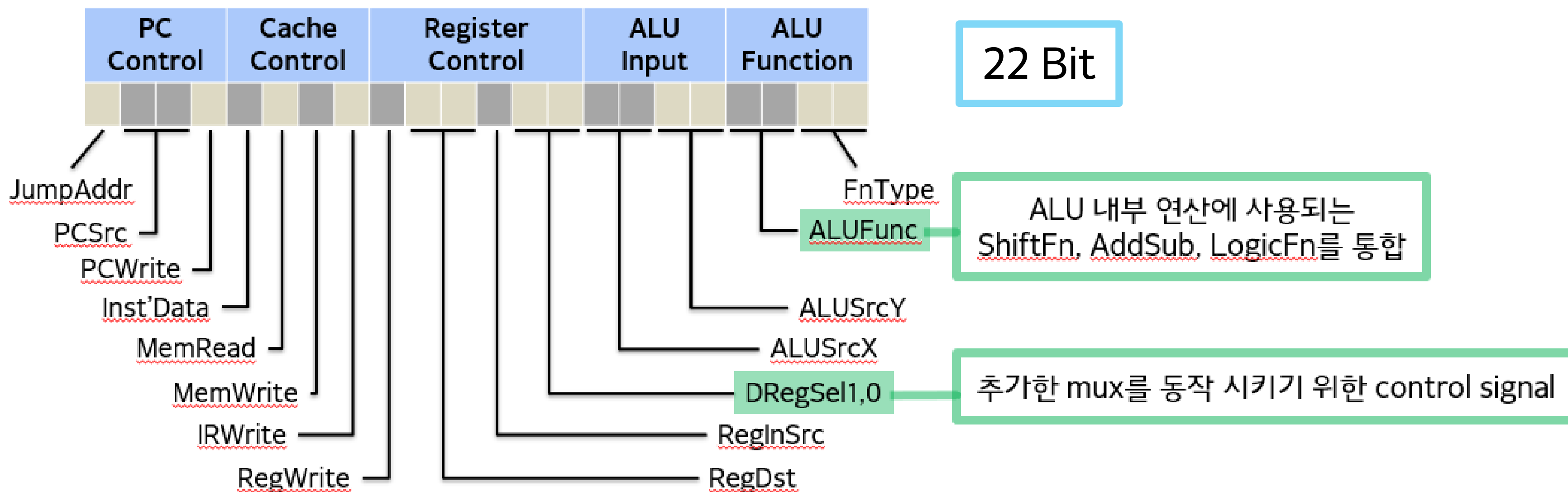
[변경 후]

$x = R[rs]$ ,  $y = sh$

$R[rd] = (R[rs] \text{를 } sh \text{만큼 shift})$

## 06. Control signal

## 06. Control signal



# 06. Control signal

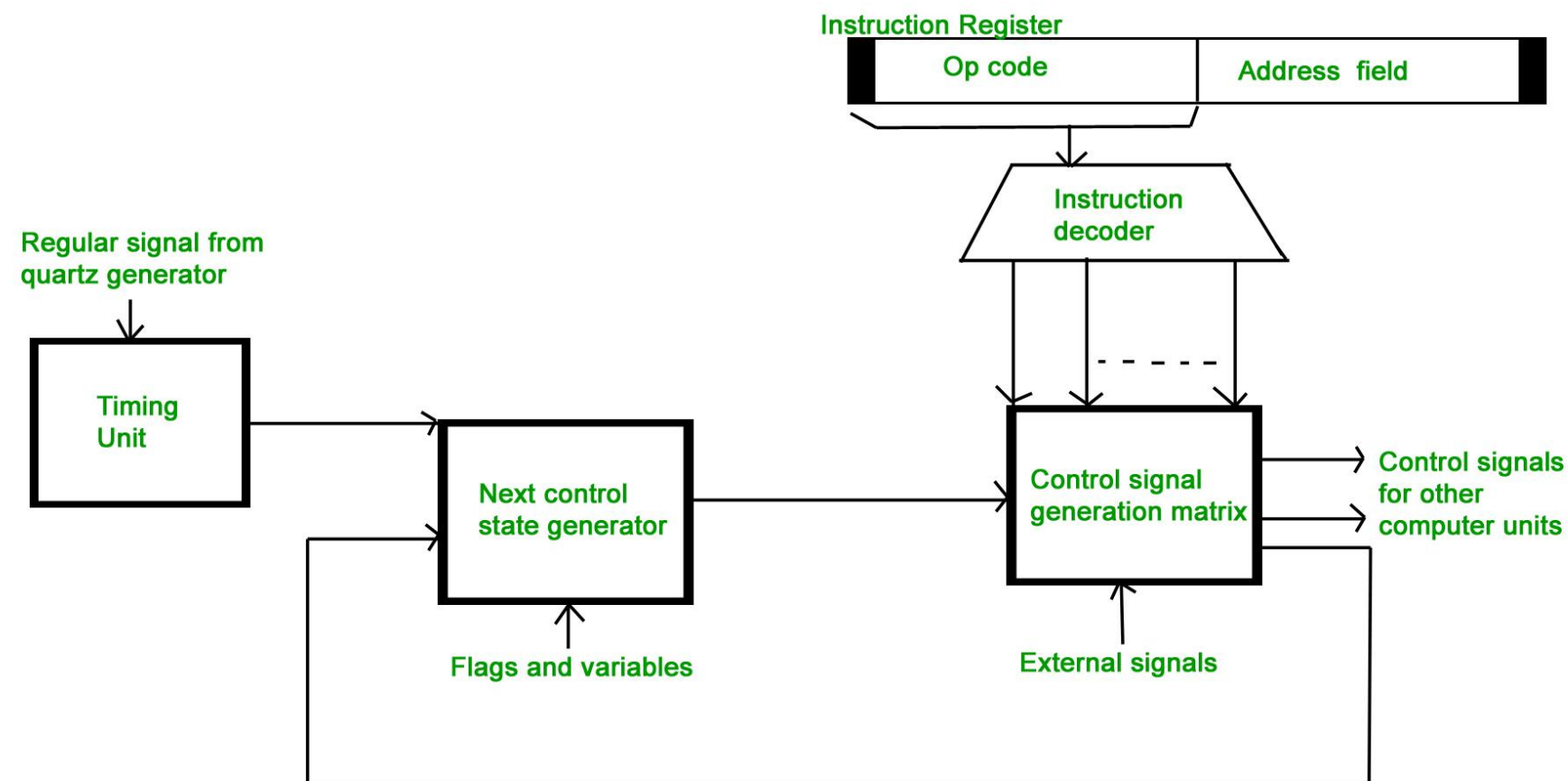
category	Control signal	0	1	2	3
Program counter	JumpAddr	jta	SysCallAddr		
	PCSrc	(jta x4) or (SysCallAddr x4)	x	z	ALU out
	PCWrite	Don't Write	Write		
Cache	Inst'Data	PC	z		
	MemRead	Don't Read	Read		
	MemWrite	Don't Write	Write		
	IRWrite	Don't Write	Write		
Register file	RegWrite	Don't Write	Write		
	RegDst	rt	rd	31	ri
	RegInSrc	data	z		
	DRegSel0	rs	rd		
	DRegSel1	rt	ri		
ALU	ALUSrcX	PC	x	z	
	ALUSrcY	4	y	imm	imm x4
	ALUFunc	ADD AND Rotate	SUB OR Logical left	XOR Logical right	Arith right
	FnType	Arithmetic	Logic	Shift	

## 06. Control signal

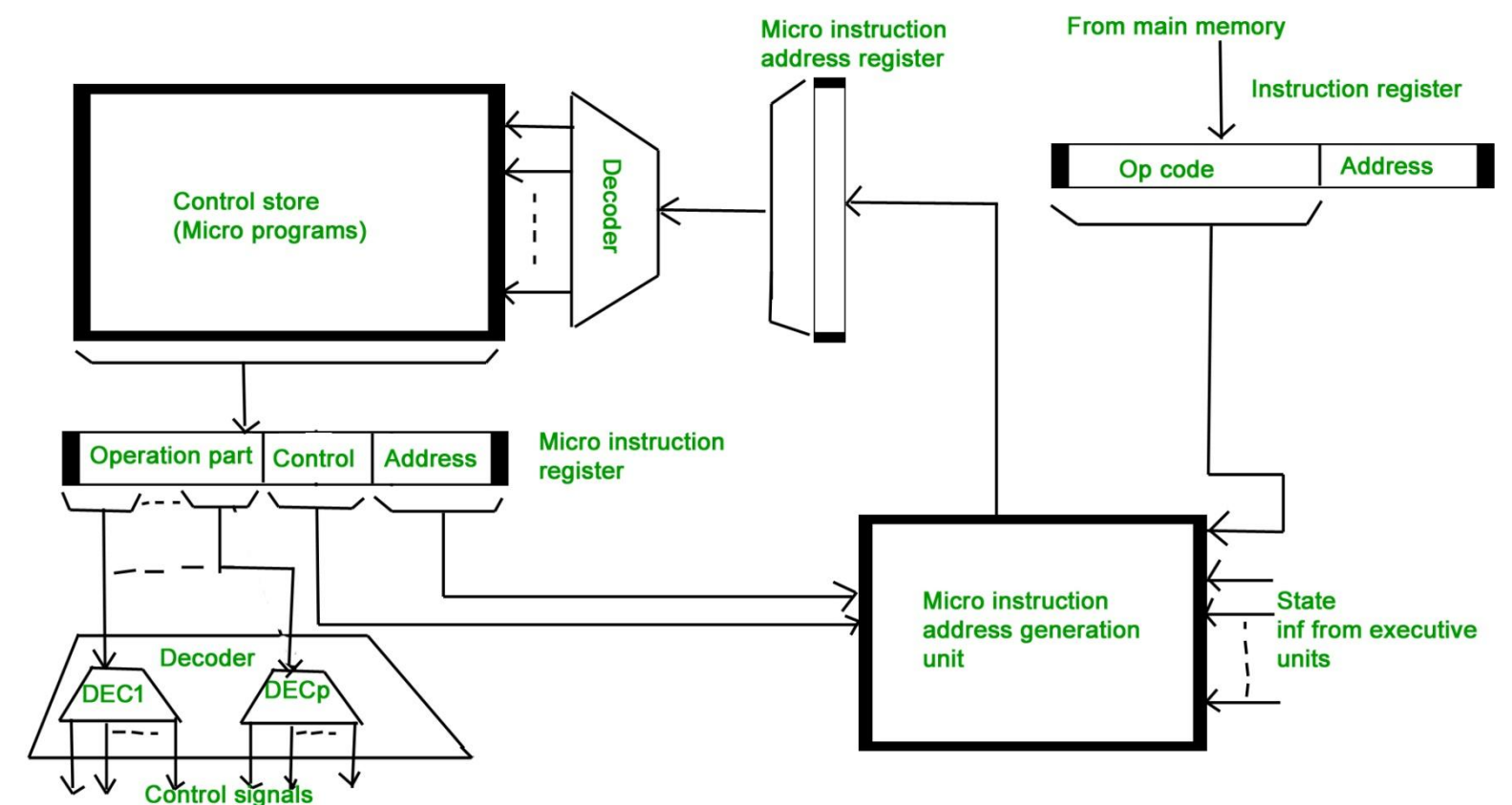
### Hardwired Control Unit으로 설계

목표: 연산 속도 개선

→ 속도 측면에서 유리한 Hardwired 방식 채택



< Hardwired Control Unit >

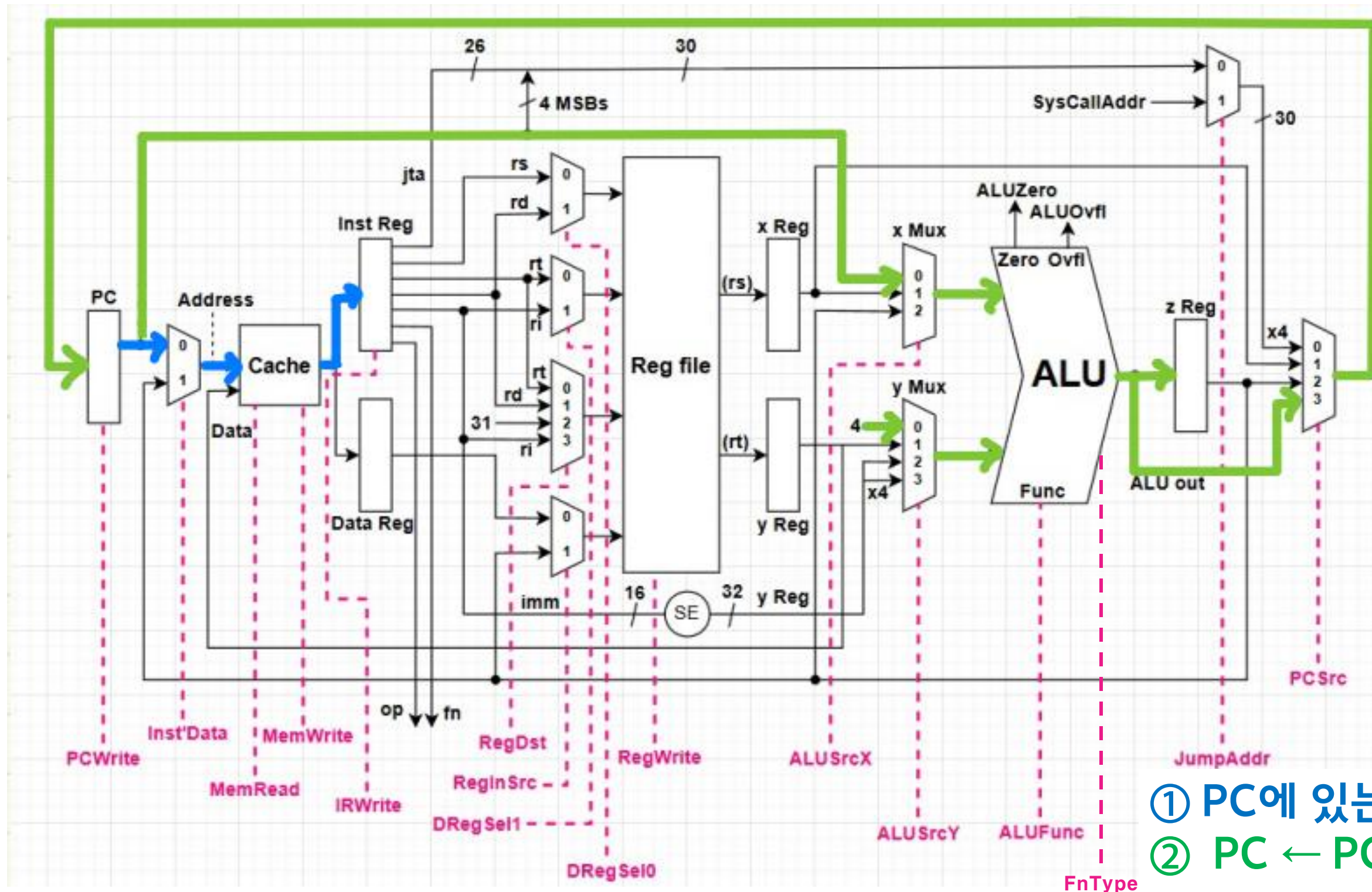


< Microprogrammed Control Unit >

# 07. Instruction Datapath

# 07. Instruction Datapath

## ROT ① Fetch



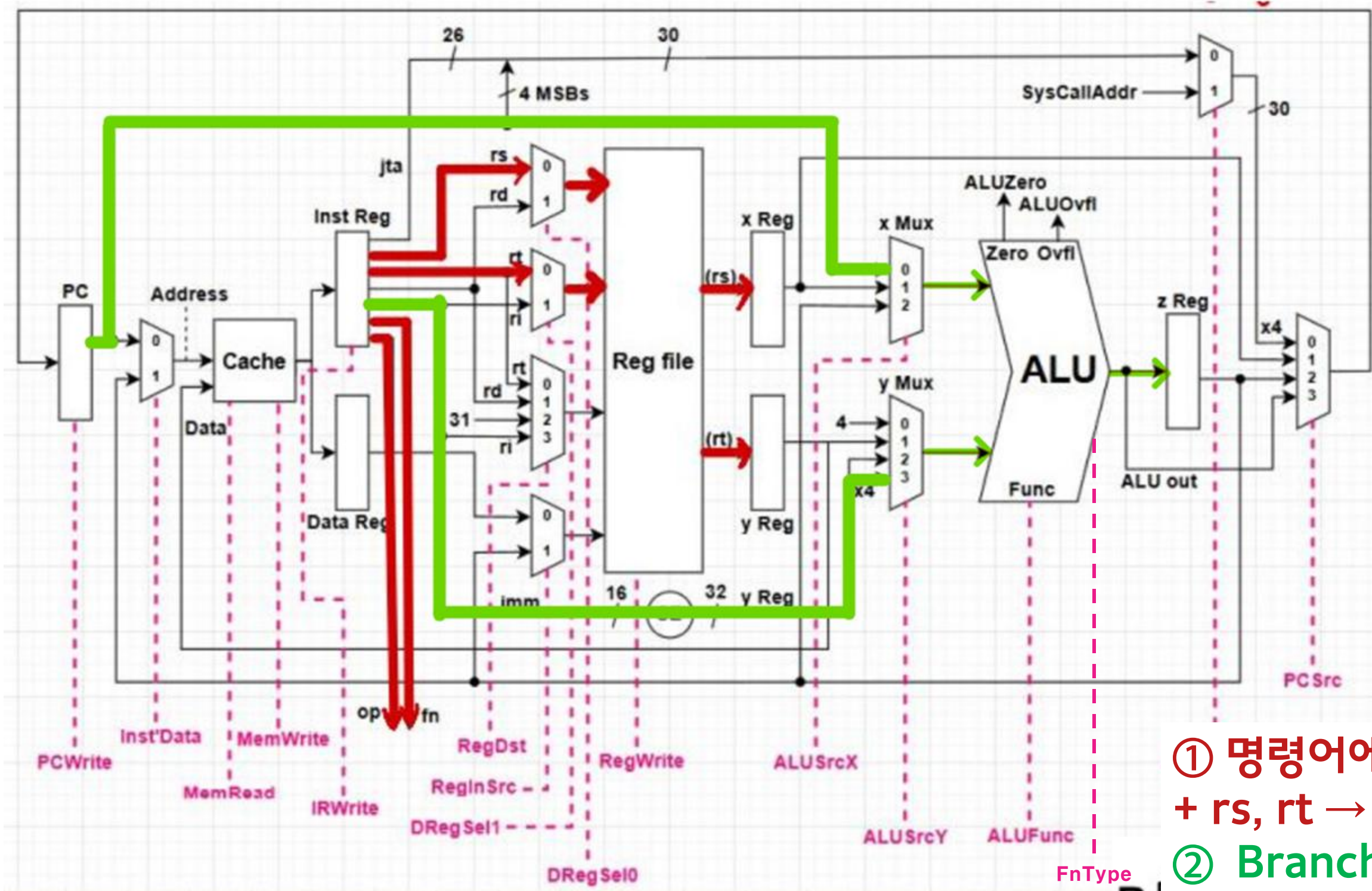
Signal	Value	Bit
PCSrc	z	10
PCWrite	Write	1
Inst'Data	PC	0
MemRead	Read	1
MemWrite	Don't Write	0
IRWrite	Write	1
RegWrite	Don't Write	00
ALUSrcX	PC	00
ALUSrcY	4	00
ALUFunc	ADD	00
FnType	Arithmetic	00

① PC에 있는 주소 → 메모리 접근 → IR 저장

②  $PC \leftarrow PC+4$



# ROT ② Decode



Signal	Value	Bit
PCWrite	Don't Write	0
MemRead	Don't Read	0
IRWrite	Don't Write	0
DRegSel0	rs	0
DRegSel1	rt	0
ALUSrcX	PC	00
ALUSrcY	imm x4	11
ALUFunc	ADD	00
FnType	Arithmetic	00

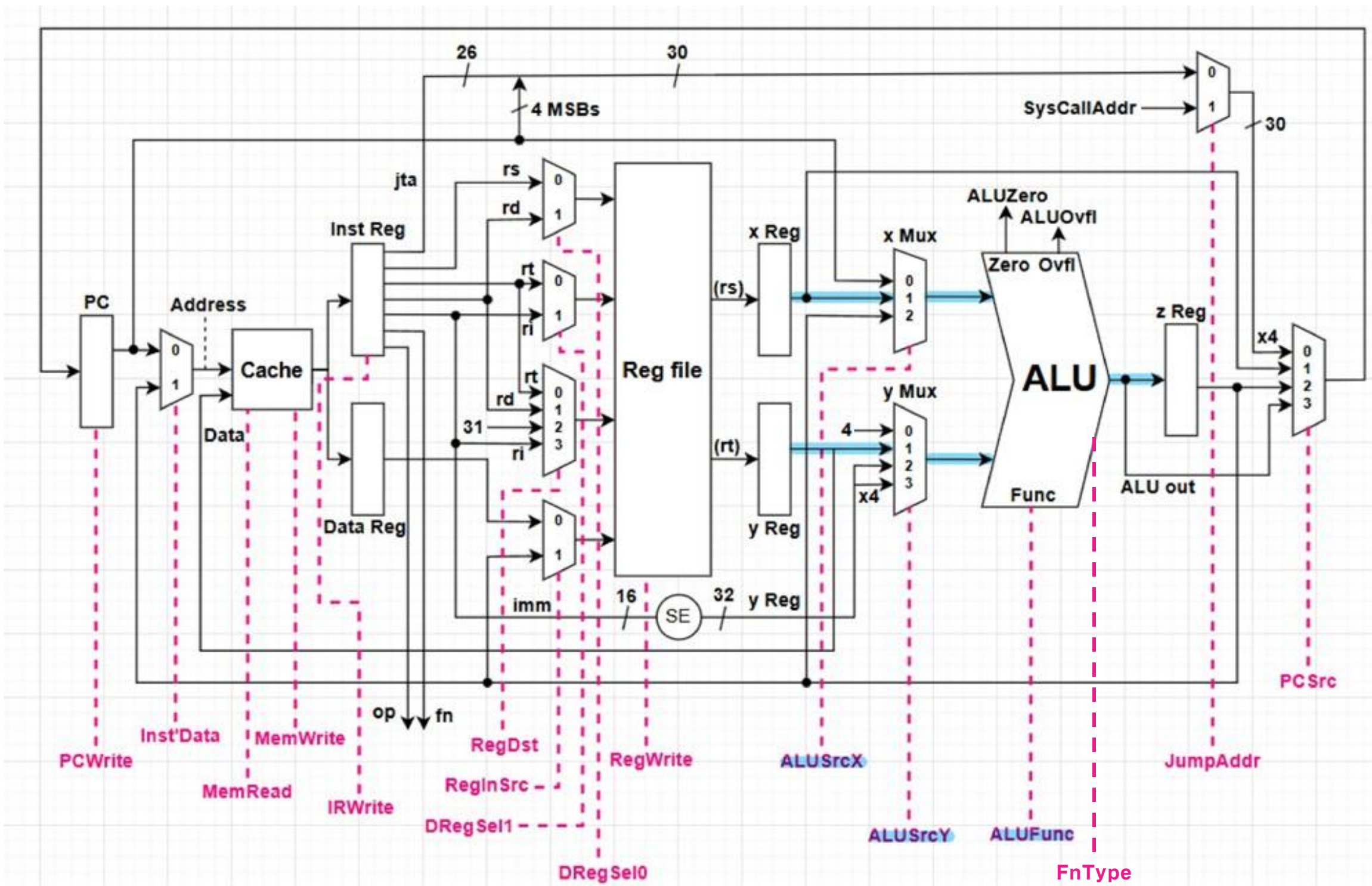
① 명령어에서 읽은 op, fn을 통해 명령어 판별  
+ rs, rt → Reg file

## ② Branch 명령어를 대비한 주소 미리 계산



# 07. Instruction Datapath

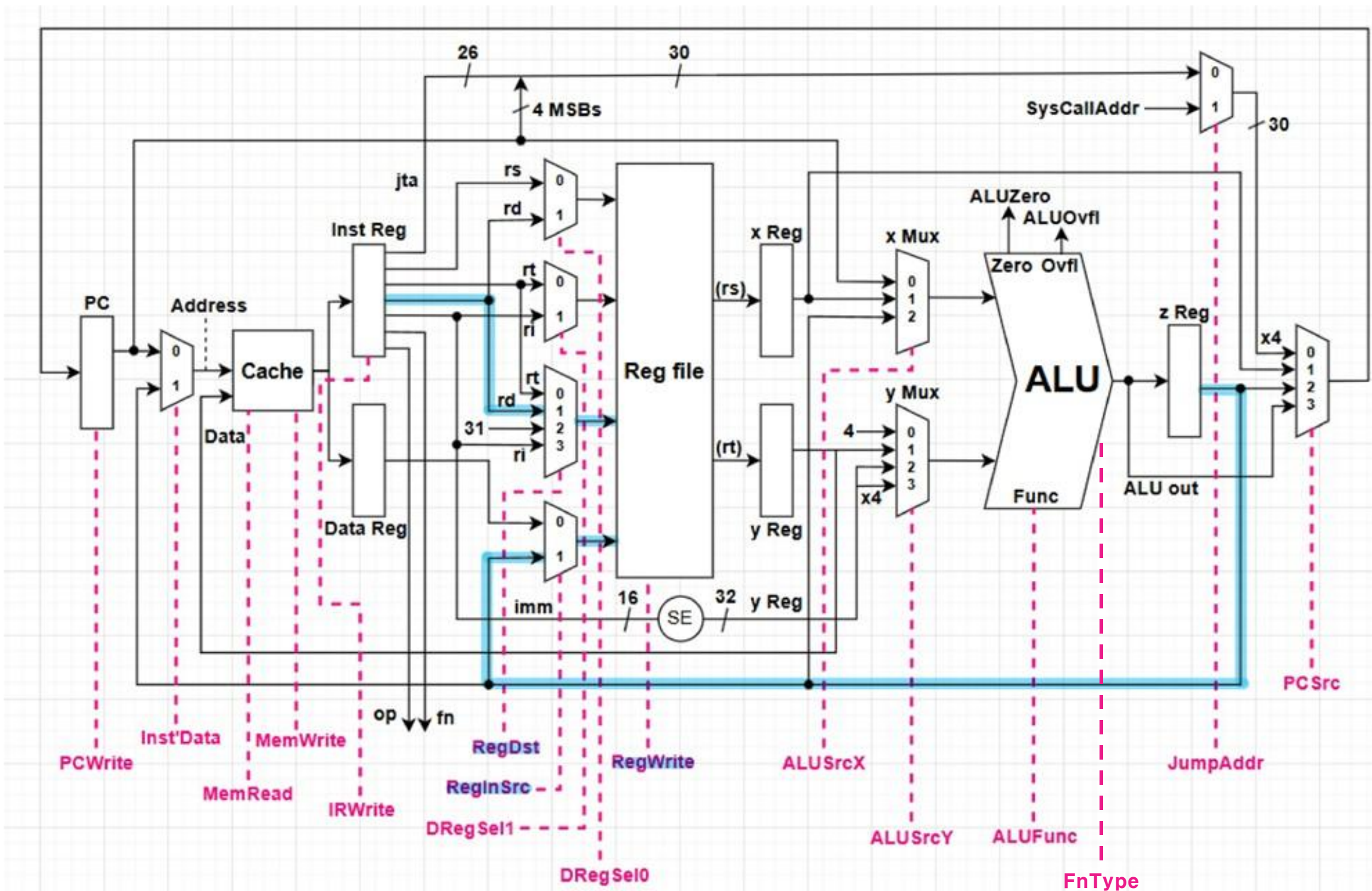
## ROT ③ Execute



Signal	Value	Bit
ALUSrcX	rs	01
ALUSrcY	rt	01
ALUFunc	Rotate	00
FnType	Shift	10

① R[rs]를 R[rt]만큼 rotate

## ROT ④ Write back

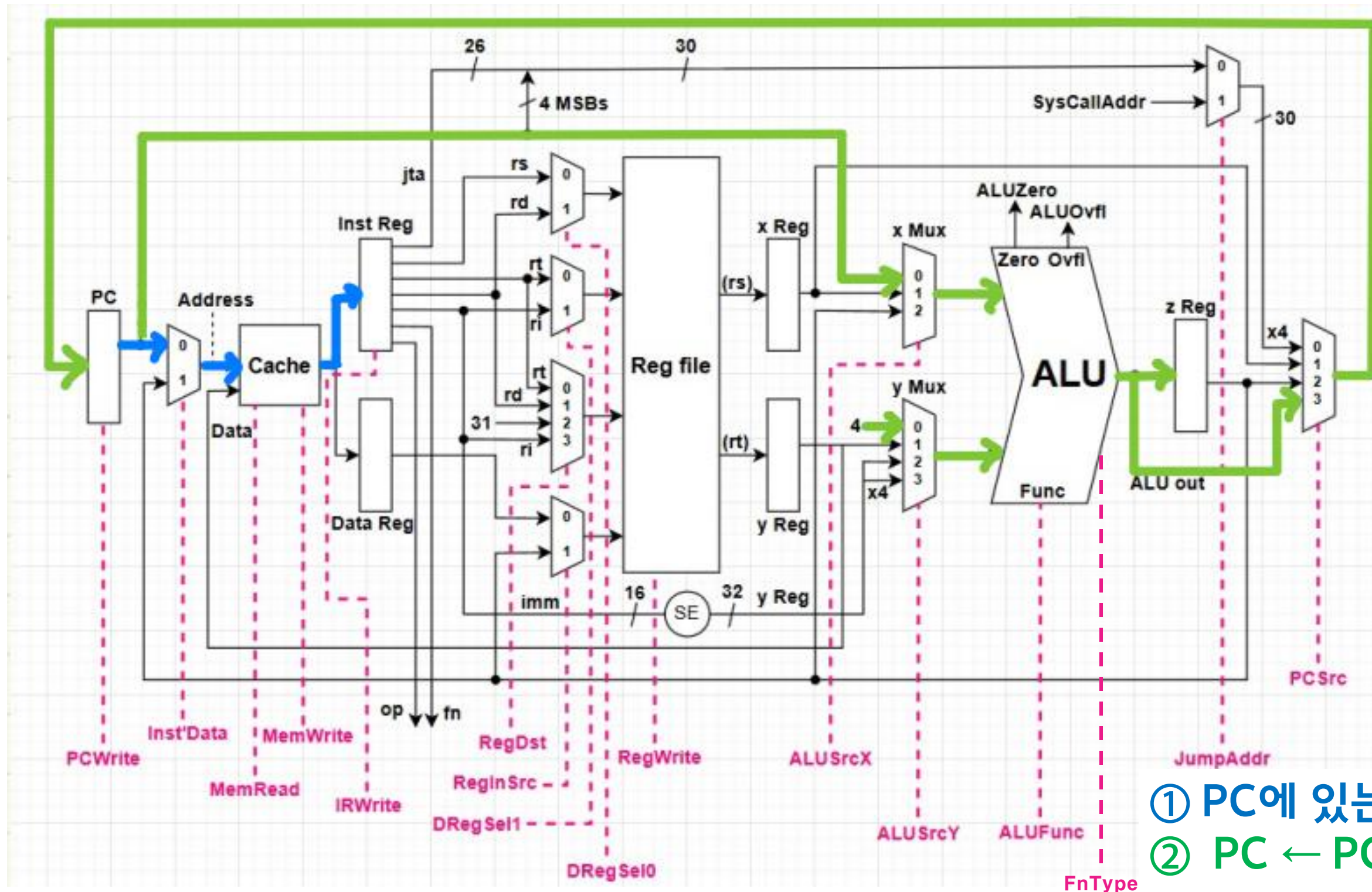


Signal	Value	Bit
RegDst	rt	00
RegInSrc	z	1
RegWrite	Write	1

### ① 연산 결과를 목적지인 rd에 저장



# SLX0, SRX0 ① Fetch



Signal	Value	Bit
PCSrc	z	10
PCWrite	Write	1
Inst'Data	PC	0
MemRead	Read	1
MemWrite	Don't Write	0
IRWrite	Write	1
RegWrite	Don't Write	00
ALUSrcX	PC	00
ALUSrcY	4	00
ALUFunc	ADD	00
FnType	Arithmetic	00

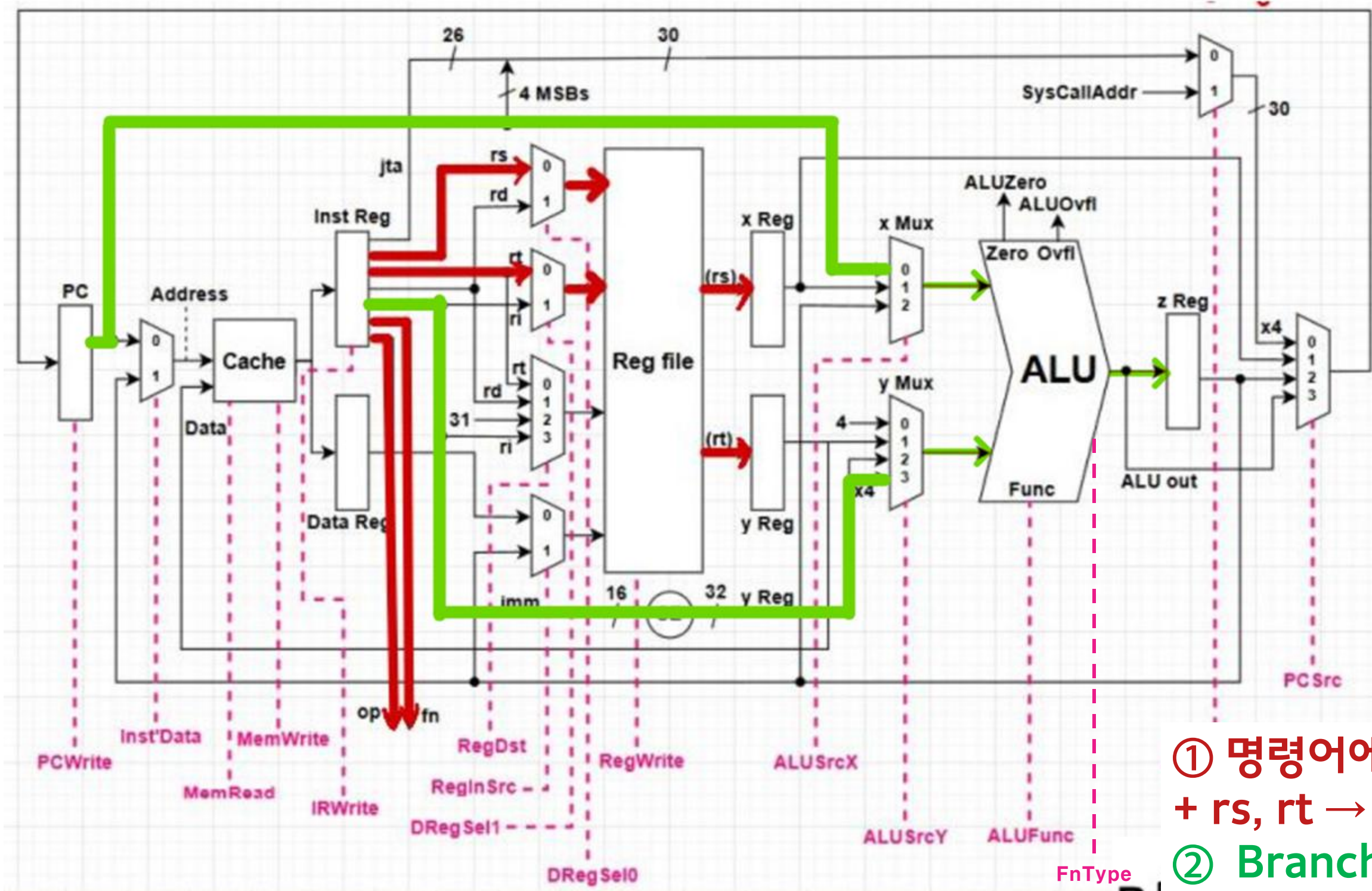
## ① PC에 있는 주소 → 메모리 접근 → IR 저장

②  $PC \leftarrow PC+4$



# 07. Instruction Datapath

## SLXO, SRXO ② Decode



Signal	Value	Bit
PCWrite	Don't Write	0
MemRead	Don't Read	0
IRWrite	Don't Write	0
DRegSel0	rs	0
DRegSel1	rt	0
ALUSrcX	PC	00
ALUSrcY	imm x4	11
ALUFunc	ADD	00
FnType	Arithmetic	00

① 명령어에서 읽은 op, fn을 통해 명령어 판별  
+ rs, rt → Reg file

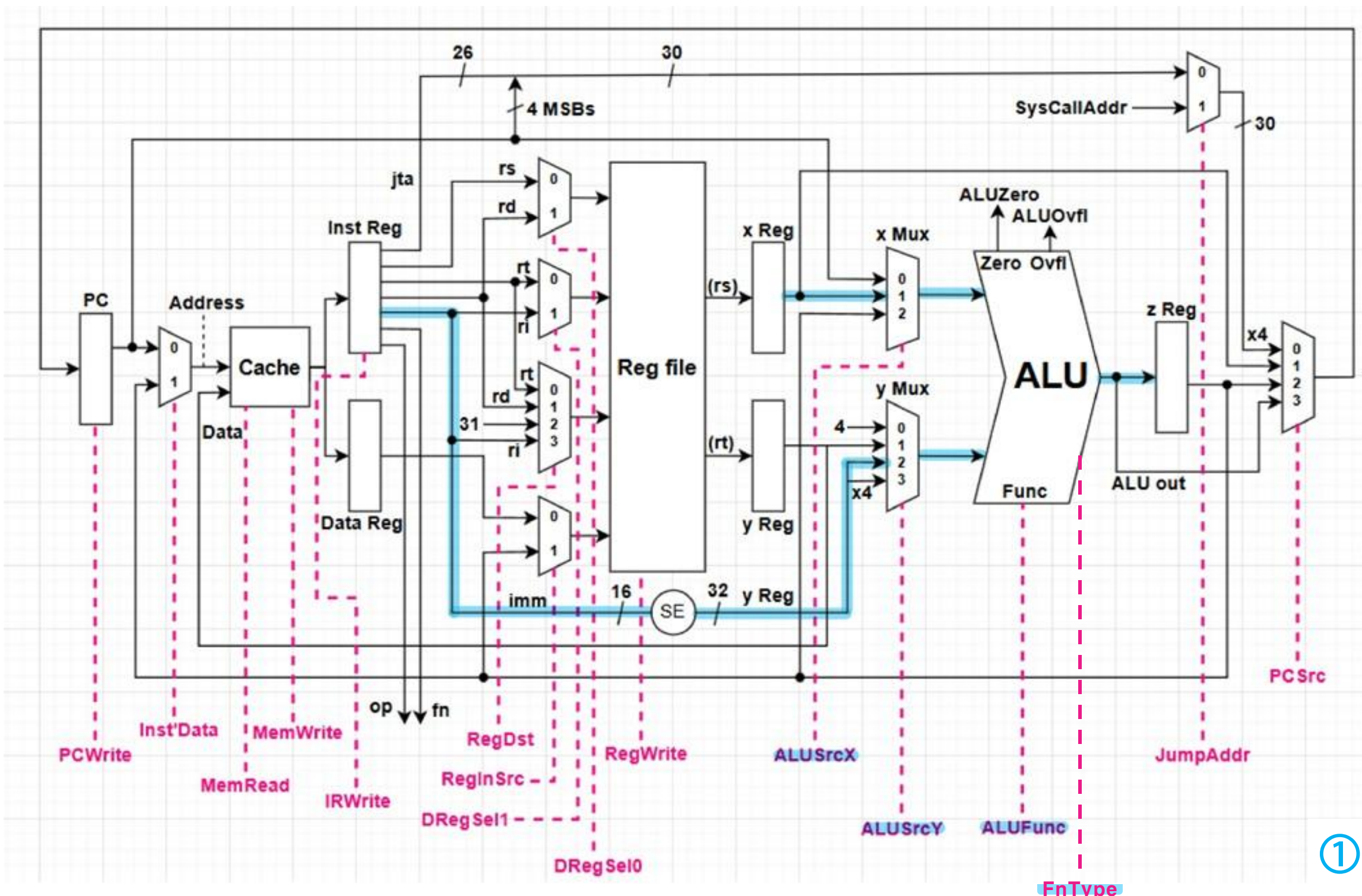
② Branch 명령어를 대비한 주소 미리 계산

# 07. Instruction Datapath

## SLX0, SRX0 ③ Execute - (1)

Signal	Value	Bit
ALUSrcX	rs	01
ALUSrcY	imm	10
ALUFunc	Shift L / R	01 / 10
FnType	Shift	10

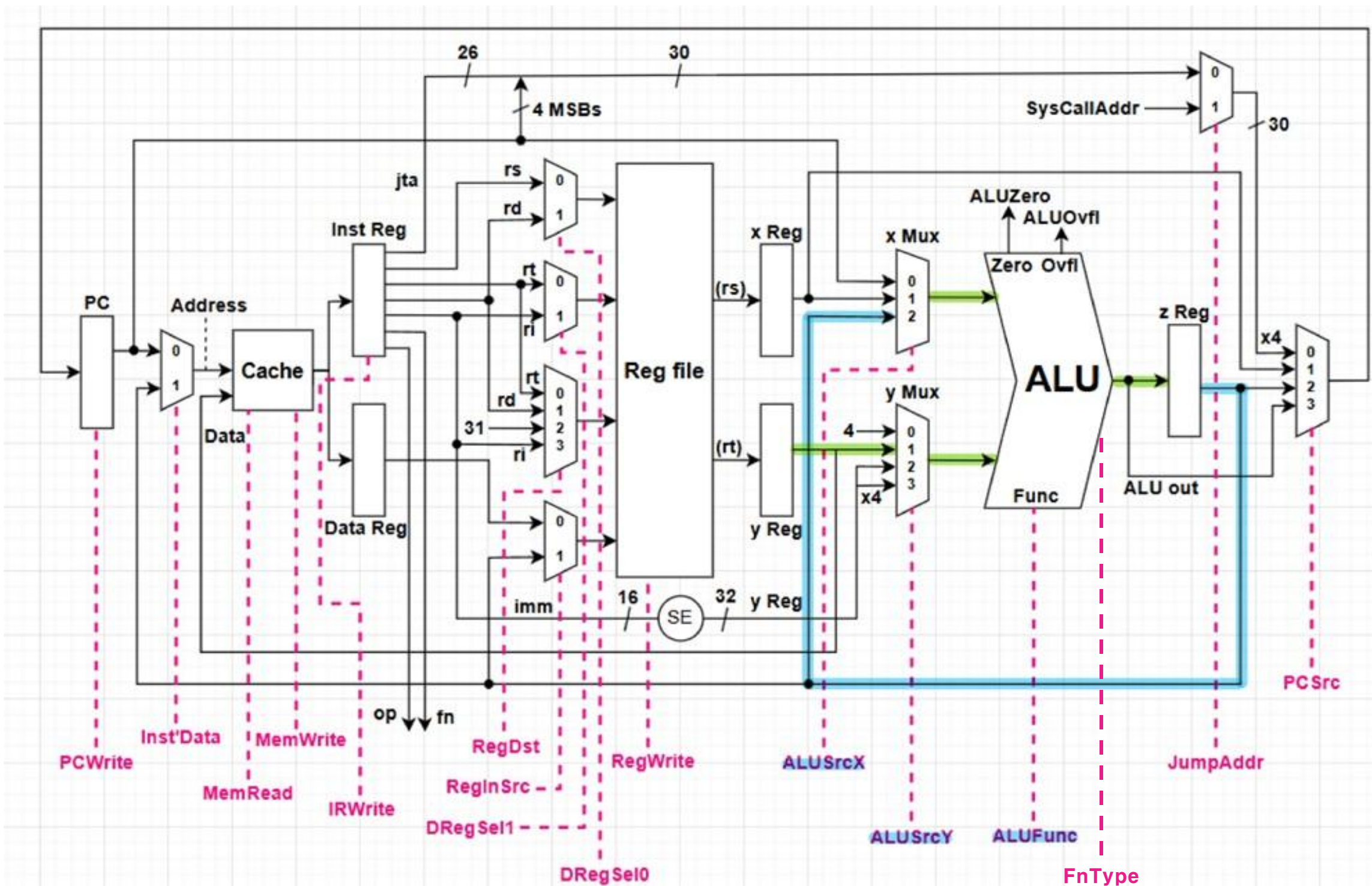
① SE imm의 하위 5 bit 만큼 R[rs] shift





# 07. Instruction Datapath

## SLXO, SRXO ④ Execute - (2)

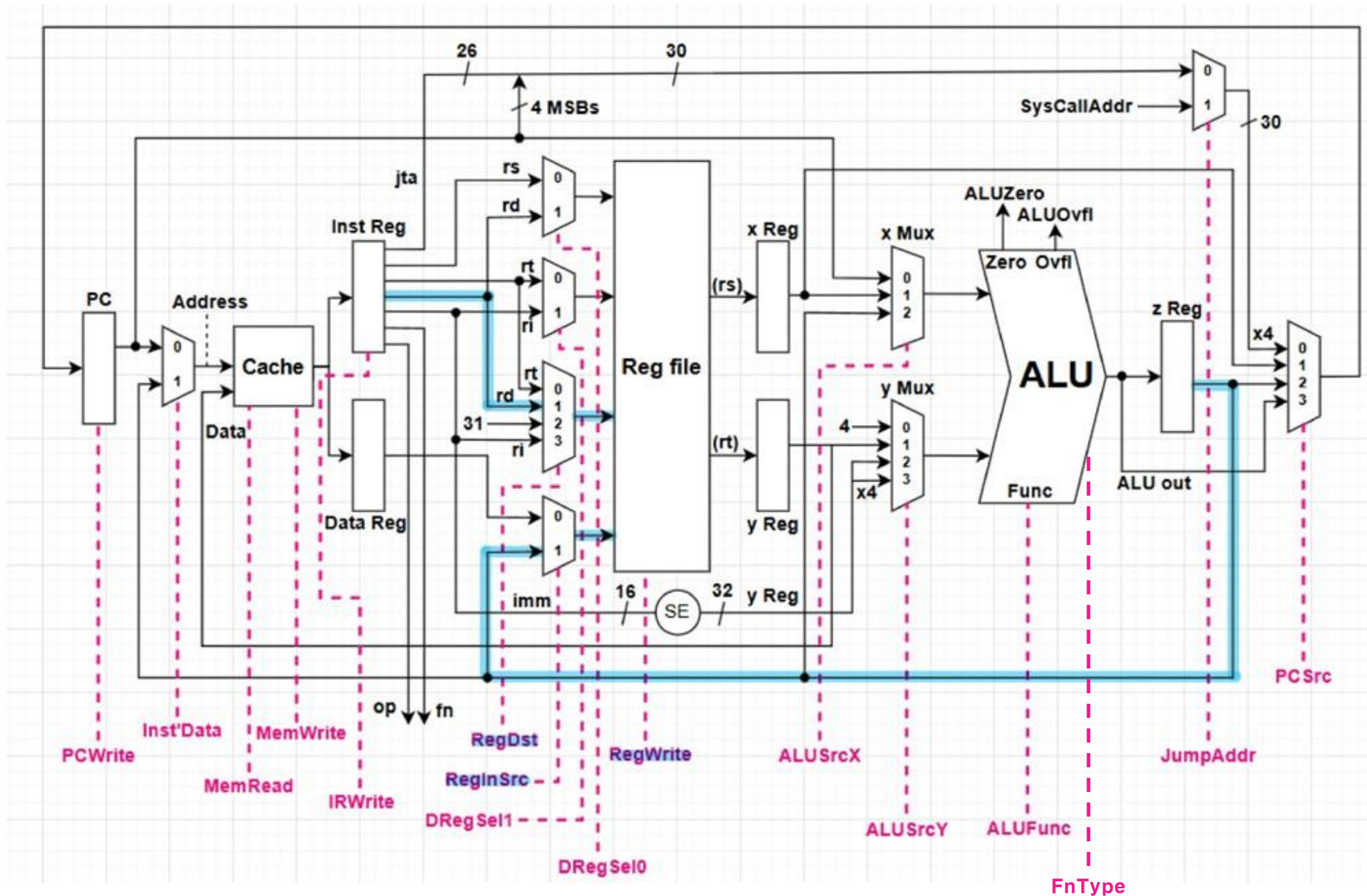


Signal	Value	Bit
ALUSrcX	z	10
ALUSrcY	rt	01
ALUFunc	XOR	10
FnType	Logic	01

- ① shift한 data -> x Mux
- ② shift한 data  $\oplus$  R[rt]

# 07. Instruction Datapath

## SLX0, SRX0 ⑤ Write back

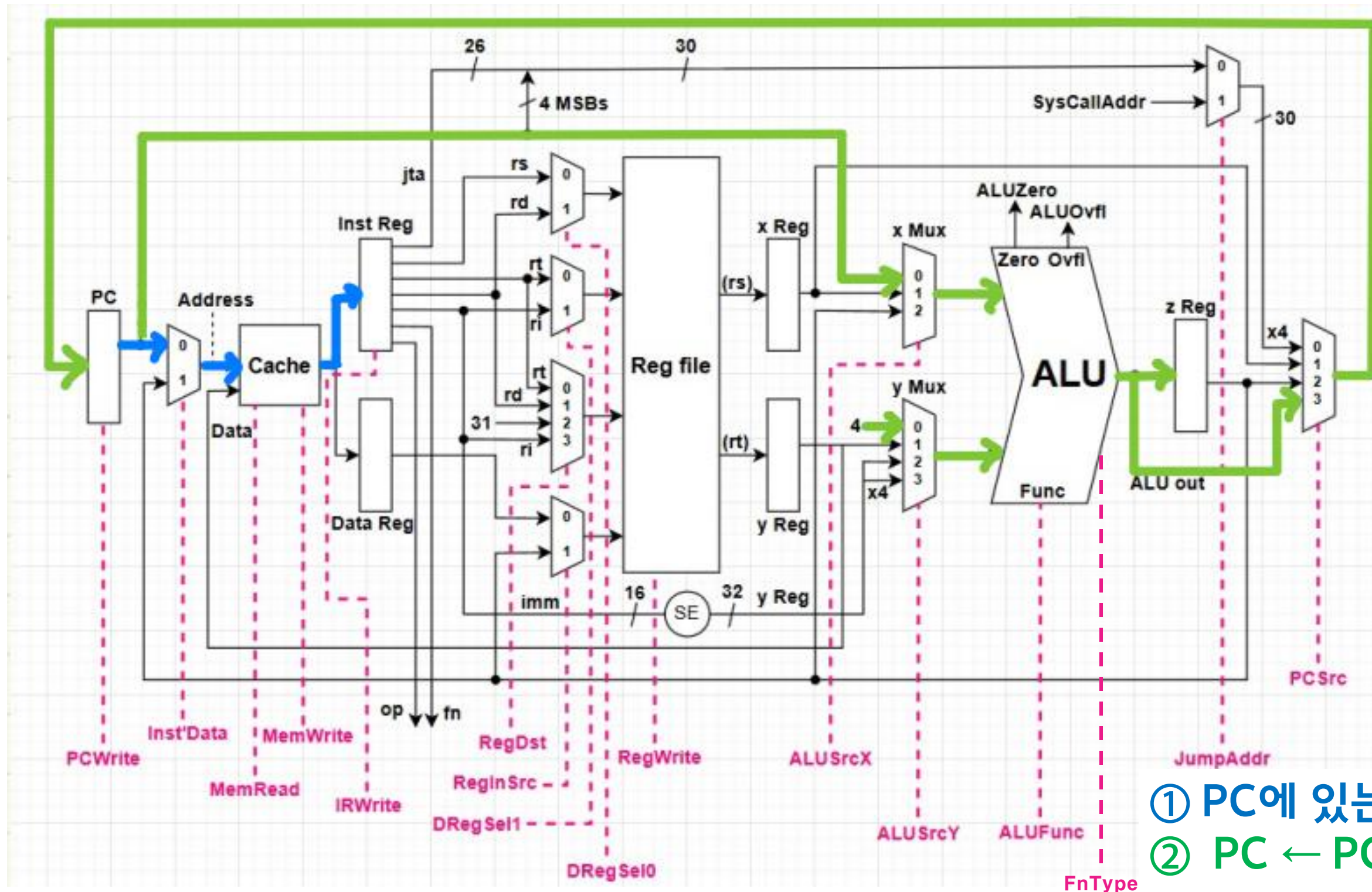


Signal	Value	Bit
RegWrite	Write	1
RegDst	rd	01
RegInSrc	z	01

① 연산 결과를 목적지인 rd에 저장



# DXOR ① Fetch



Signal	Value	Bit
PCSrc	z	10
PCWrite	Write	1
Inst'Data	PC	0
MemRead	Read	1
MemWrite	Don't Write	0
IRWrite	Write	1
RegWrite	Don't Write	00
ALUSrcX	PC	00
ALUSrcY	4	00
ALUFunc	ADD	00
FnType	Arithmetic	00

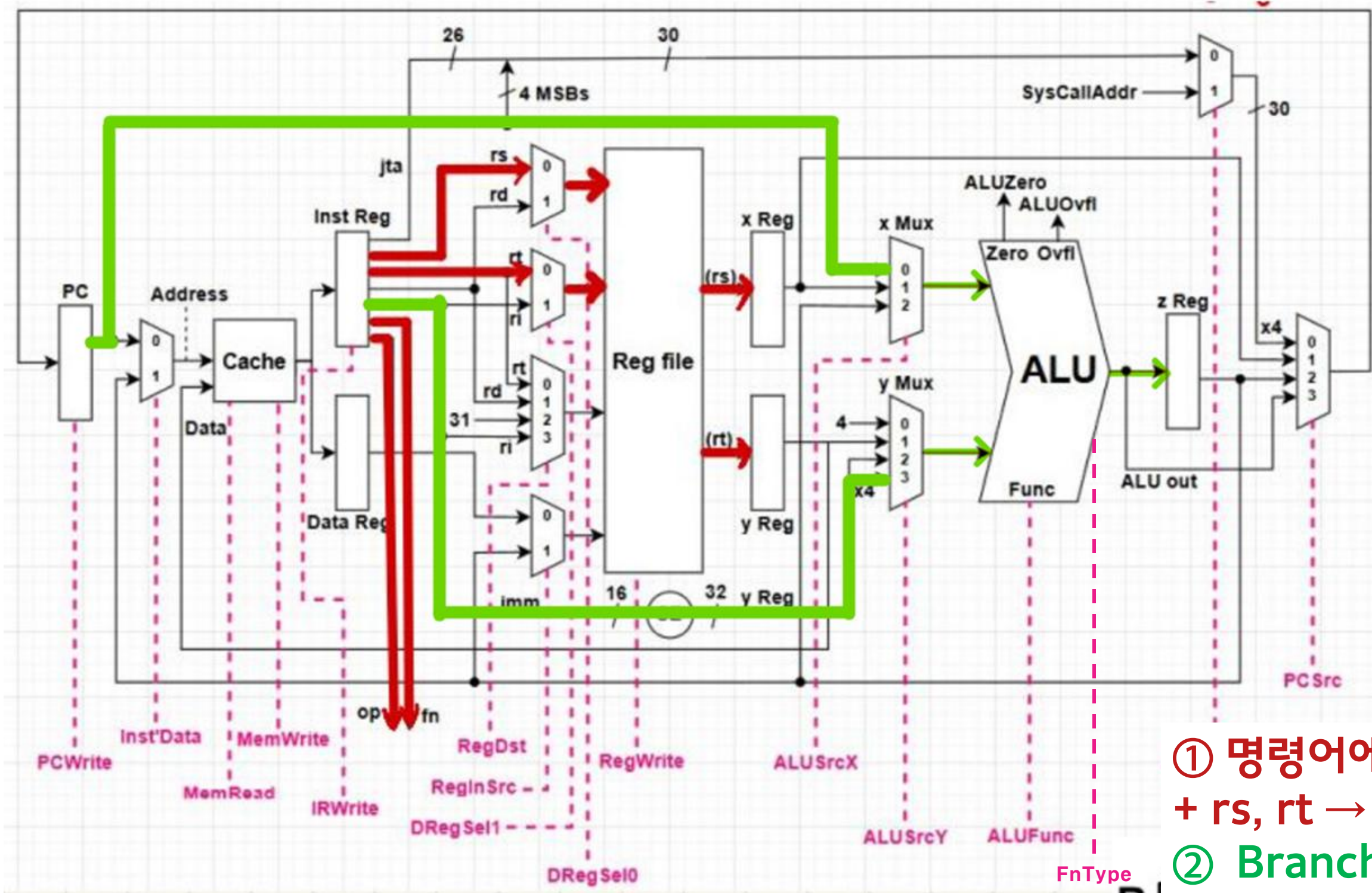
### ① PC에 있는 주소 → 메모리 접근 → IR 저장

②  $PC \leftarrow PC+4$



# 07. Instruction Datapath

## DXOR ② Decode



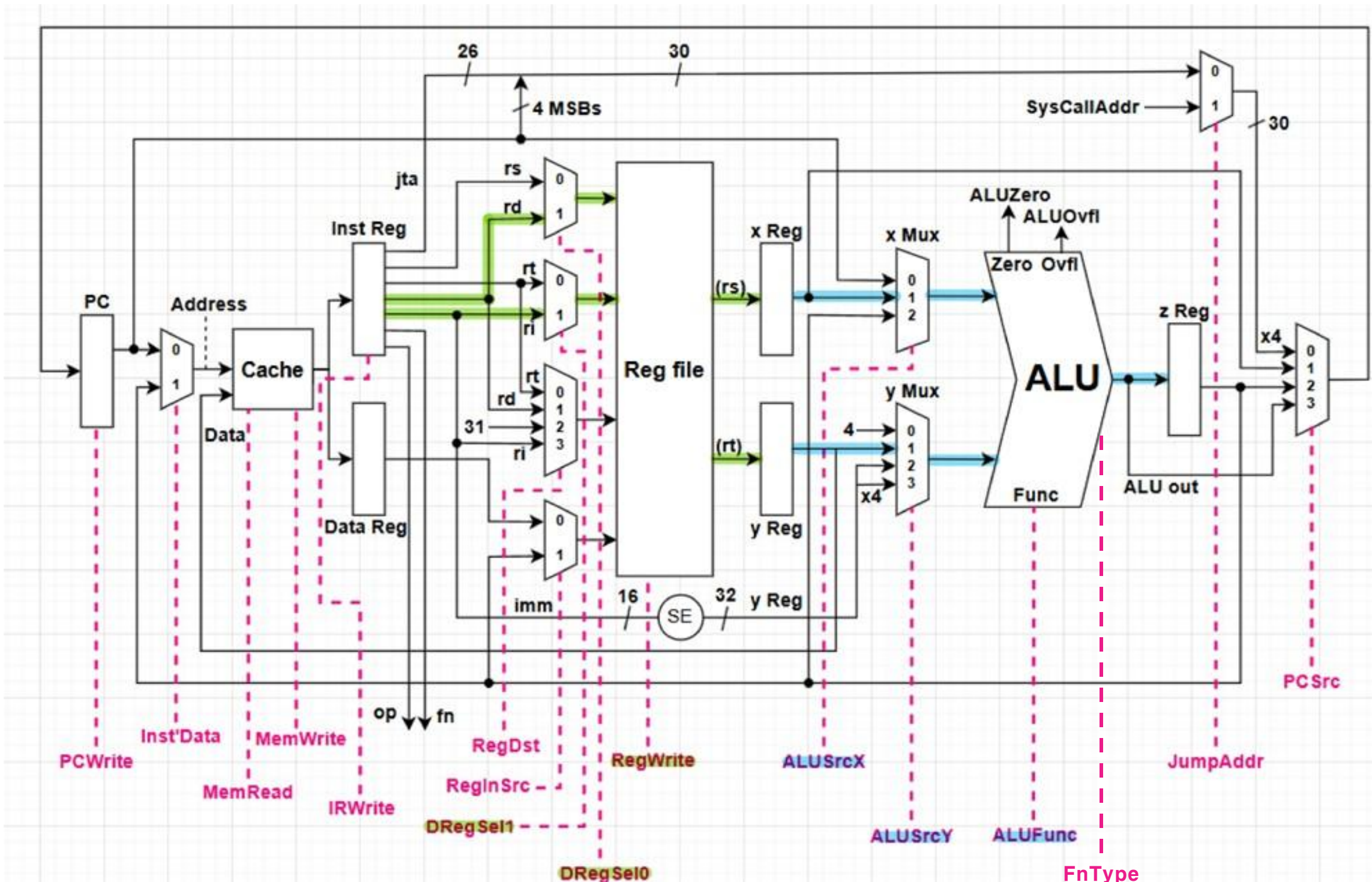
Signal	Value	Bit
PCWrite	Don't Write	0
MemRead	Don't Read	0
IRWrite	Don't Write	0
DRegSel0	rs	0
DRegSel1	rt	0
ALUSrcX	PC	00
ALUSrcY	imm x4	11
ALUFunc	ADD	00
FnType	Arithmetic	00

① 명령어에서 읽은 op, fn을 통해 명령어 판별  
+ rs, rt → Reg file

② Branch 명령어를 대비한 주소 미리 계산

# 07. Instruction Datapath

## DXOR ③ Execute – (1)



Signal	Value	Bit
RegWrite	Don't Write	0
DRegSel0	rd	1
DRegSel1	ri	1
ALUSrcX	x	01
ALUSrcY	y	01
ALUFunc	XOR	10
FnType	Logic	01

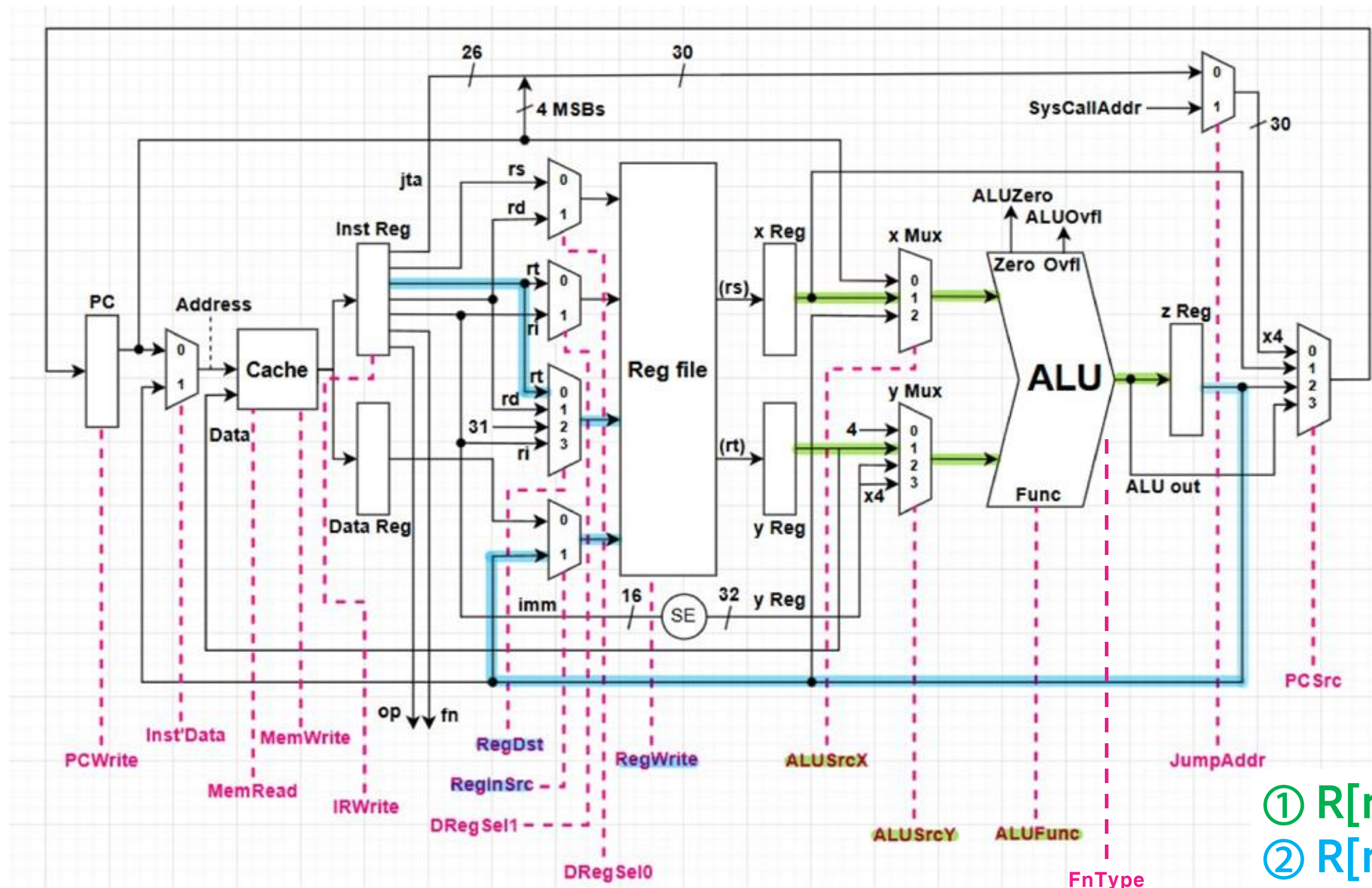
①  $R[rs] \oplus R[rt]$

②  $R[rd], R[ri] \rightarrow x \text{ Mux}, y \text{ Mux}$



# 07. Instruction Datapath

## DXOR ④ Execute – (2) + Write back – (1)

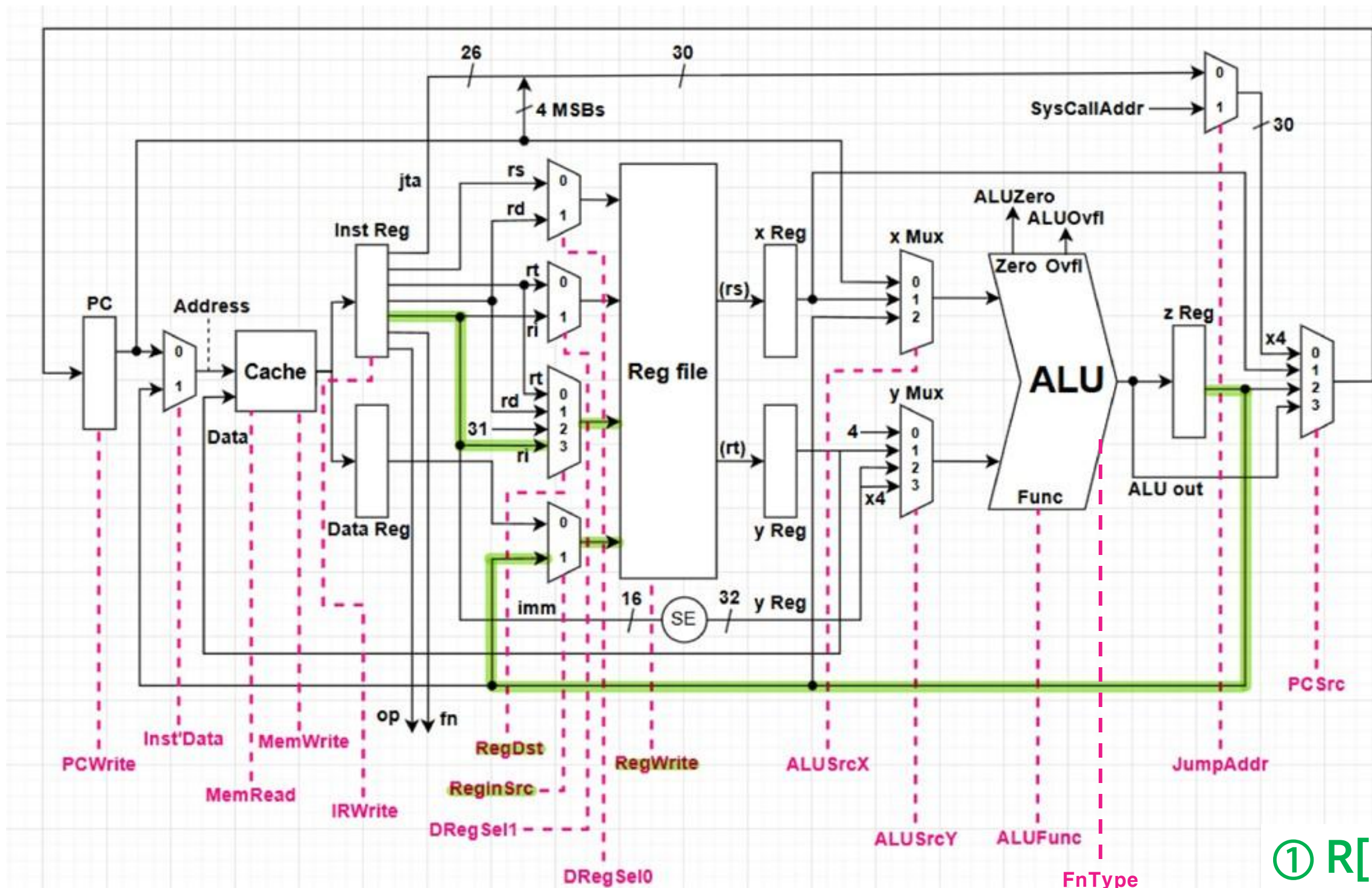


Signal	Value	Bit
RegWrite	Write	1
RegDst	rt	00
RegInSrc	z	1
ALUSrcX	rd	01
ALUSrcY	ri	01
ALUFunc	XOR	10
FnType	Logic	01

①  $R[rd] \oplus R[ri]$

②  $R[rs], R[rt]$  연산 결과를 목적지인  $rt$ 에 저장

## DXOR ⑤ Write back – (2)



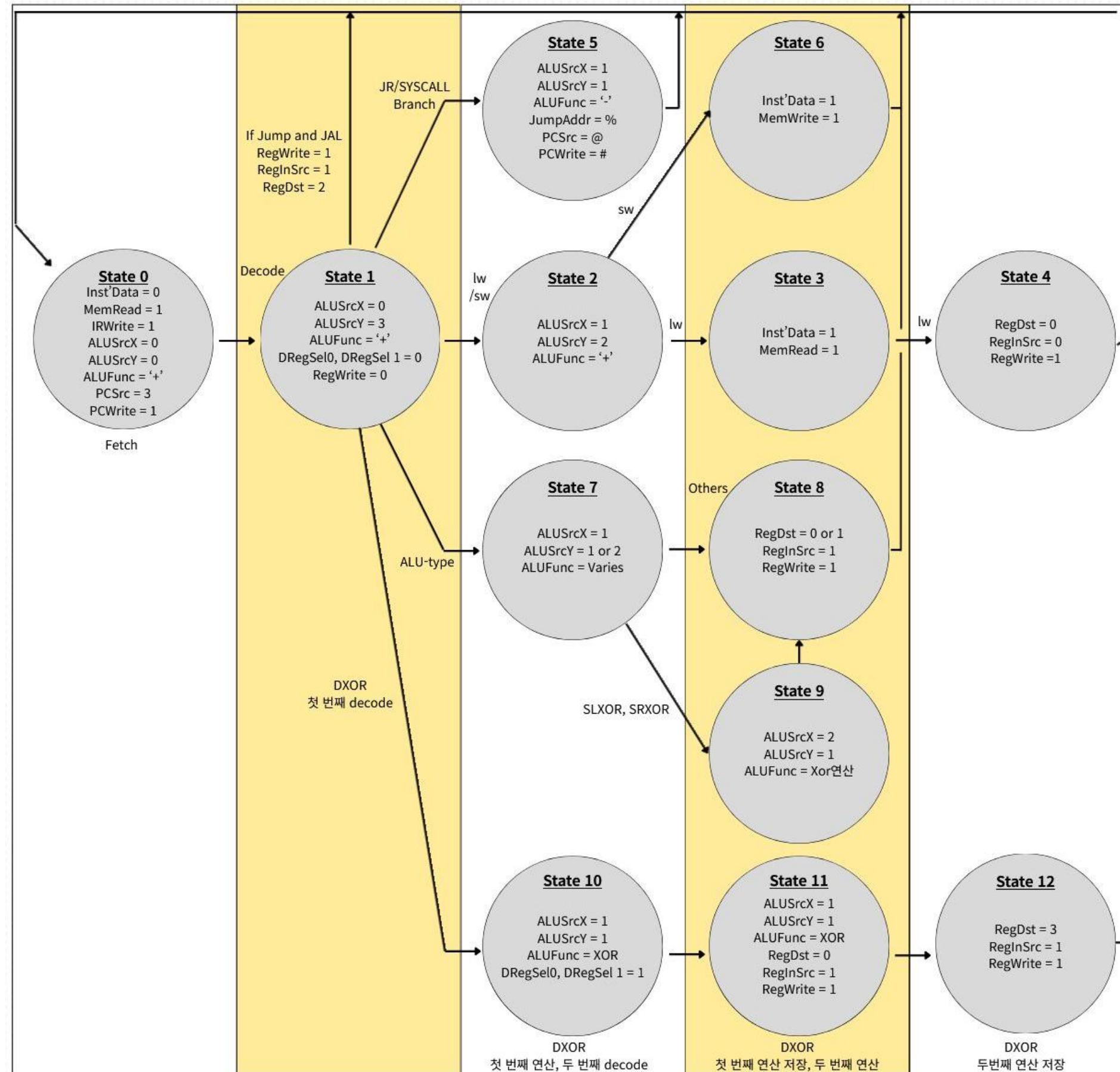
Signal	Value	Bit
RegWrite	Write	1
RegDst	ri	11
RegInSrc	z	1

① R[rd], R[ri] 연산 결과를 목적지인 ri에 저장

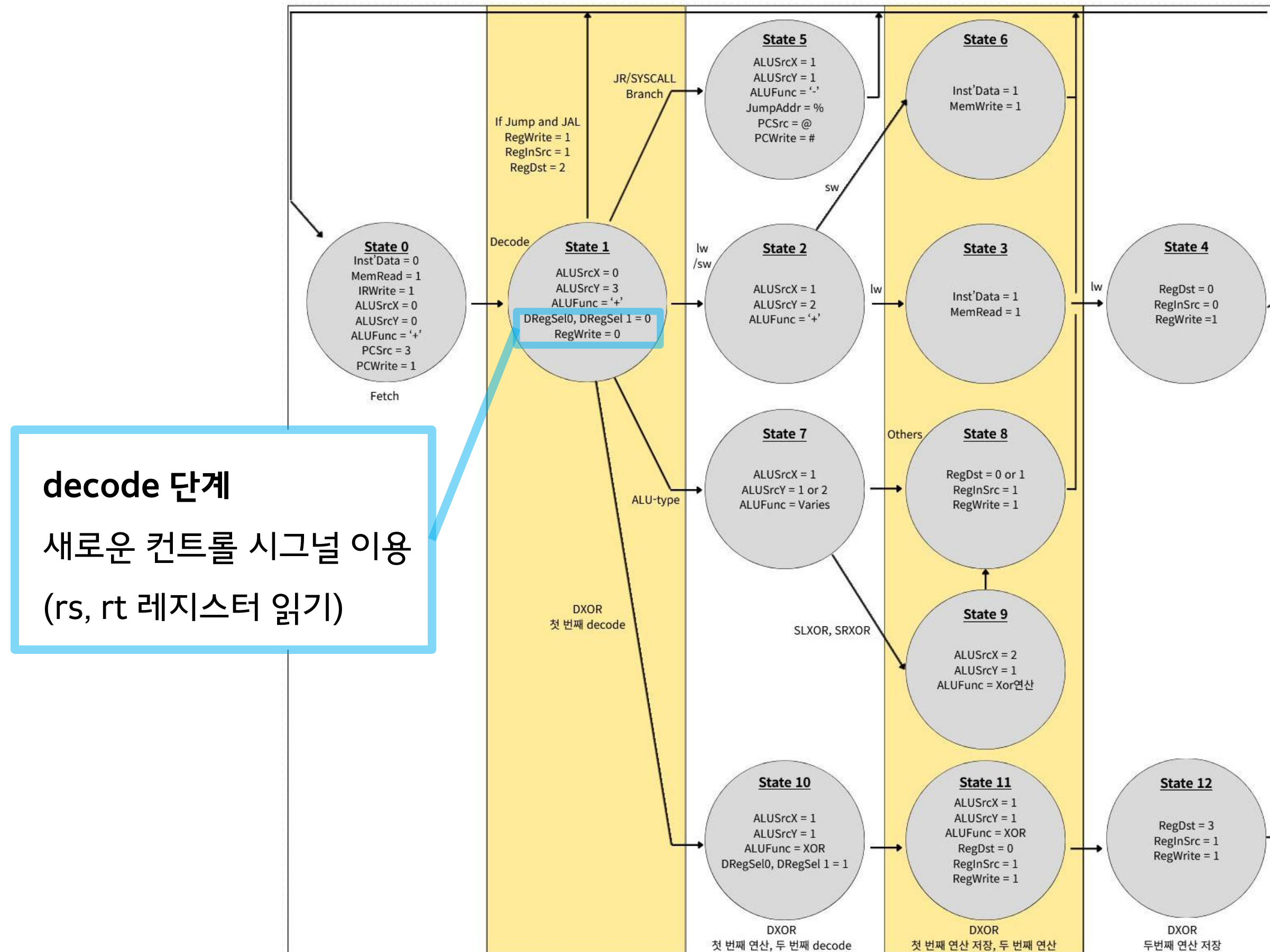
# 08. Control State Machine



# 08. Control State Machine



# 08. Control State Machine



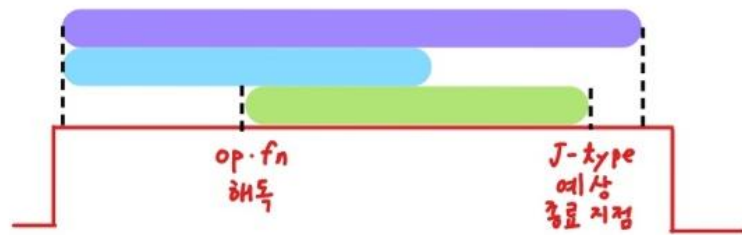


# 08. Control State Machine

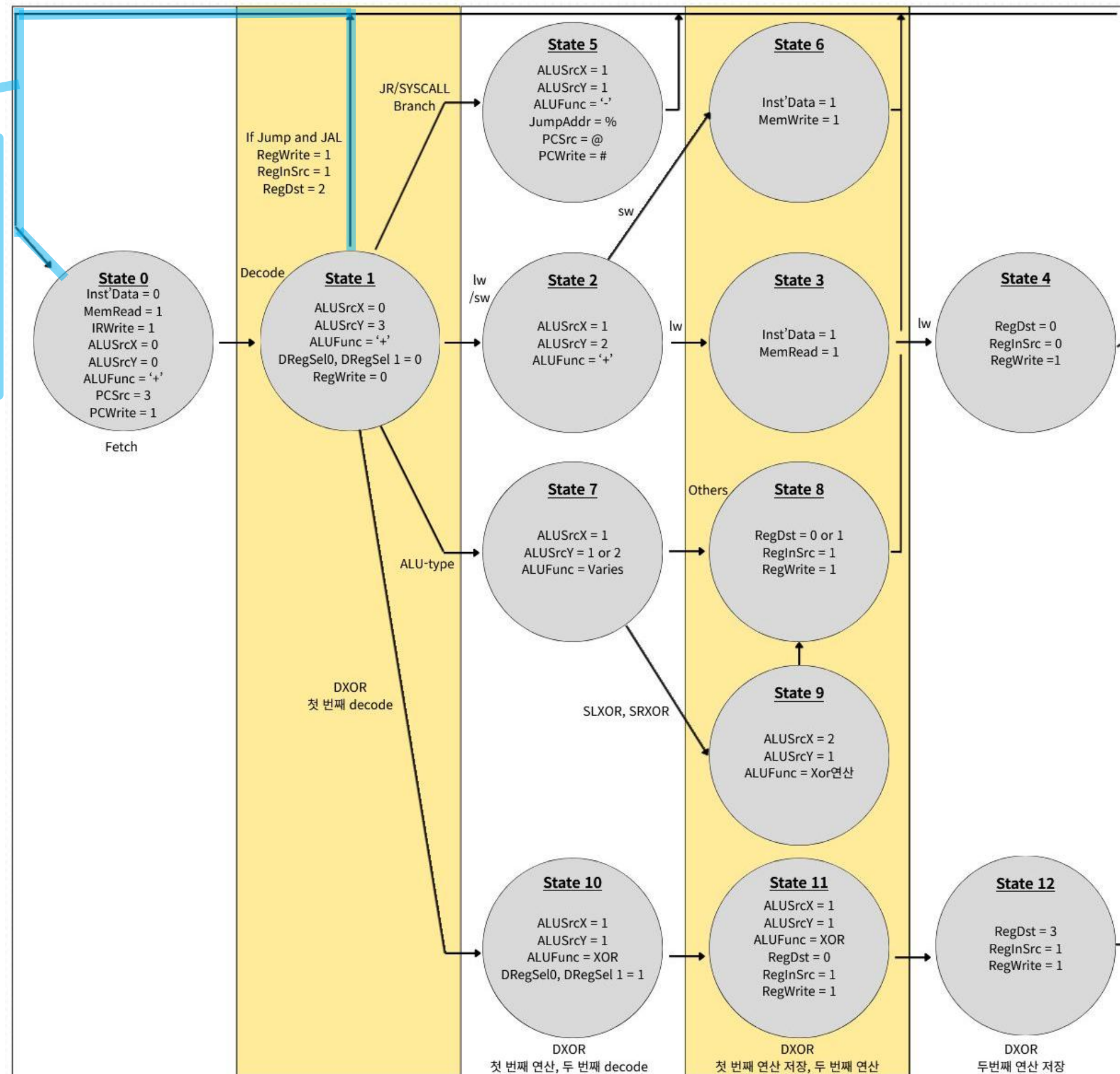
## J 타입

[기존] state 0 → 1 → 5 → 0

[변경 후] state 0 → 1 → 0

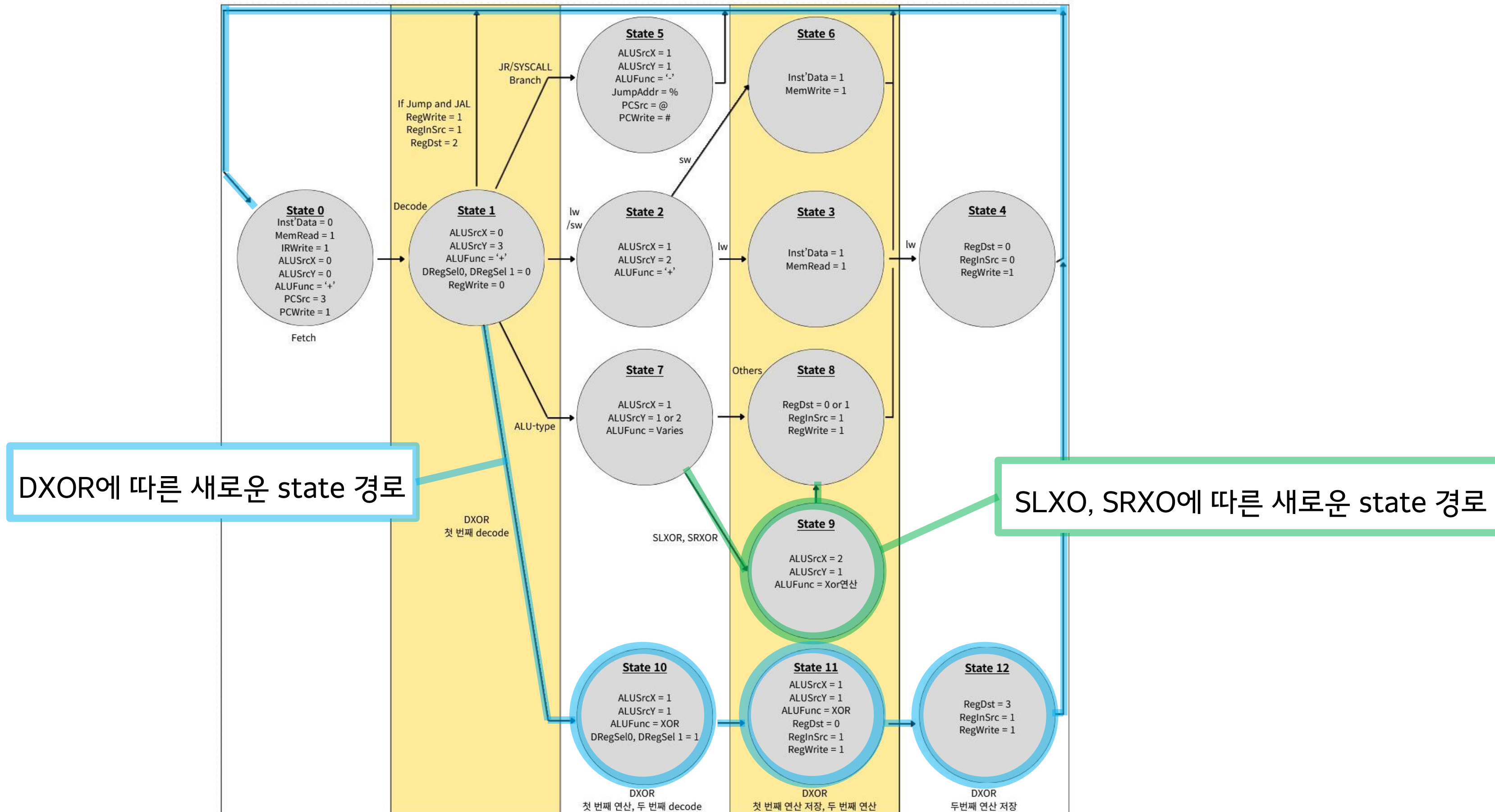


- ① Branch 명령어 대비 주소 계산
- ② Decode
- ③ j타입 명령어라면, 해당 주소로 jump





# 08. Control State Machine



## 09. 구현

## 09. 구현

---

Vivado를 이용한 하드웨어 설계



Python으로 만든 어셈블러로 testbench에 필요한 HEXCODE 생성



기존 MiniMIPS로 구현 vs 개선된 CPU로 구현

## 09. 구현

---

기존

```
xor $t4 $t4 $t5  
xor $t6 $t6 $t7
```

```
sll $t4 $t5 6  
xor $t3 $t4 $t6
```

```
sll $t4 $t5 6  
srl $t3 $t5 26  
or $t6 $t3 $t4
```



개선

```
dxor $t4 $t5 $t7 $t6
```

```
slixo $t3 $t5 $t6 6
```

```
addi $t2 $t0 6  
rot $t6 $t5 $t2
```



## 09. 구현

---

### xshift32

- ① 초기 state x 지정 (seed)
- ②  $x = x \text{ XOR } (x \ll 13)$
- ③  $x = x \text{ XOR } (x \gg 17)$
- ④  $x = x \text{ XOR } (x \ll 5)$

```
<xorshift32>  
x ^= x << 13  
x ^= x >> 17  
x ^= x << 5  
return x
```

## 09. 구현

### SHA-256

#### - $\Sigma 0$ / $\Sigma 1$

- ① 입력 word a 준비
- ② a를 오른쪽으로 2 / 6비트 rotate
- ③ a를 오른쪽으로 13 / 11비트 rotate
- ④ a를 오른쪽으로 22 / 25비트 rotate
- ⑤ 위 3개의 결과를 XOR

//  $\Sigma 0$

$\Sigma 0 = \text{ROTR}(a, 2) \wedge \text{ROTR}(a, 13) \wedge \text{ROTR}(a, 22)$

//  $\Sigma 1$

$\Sigma 1 = \text{ROTR}(e, 6) \wedge \text{ROTR}(e, 11) \wedge \text{ROTR}(e, 25)$

## 09. 구현

### xoshiro128+

- ① 초기 state (s0, s1, s2, s3) (seed)
- ②  $\text{result} = \text{rotr}(s0 + s3, 7) + s0$
- ③ XOR + SHIFT + ROTATE로 state 갱신
- ④ state 저장 후 반복 → 계속 난수 생성

```
<xoshiro128+>
```

```
s0 = state[0]
```

```
s1 = state[1]
```

```
s2 = state[2]
```

```
s3 = state[3]
```

```
# --- output 생성 ---
```

```
result = rotr(s0 + s3, 7) + s0
```

```
# --- 상태 업데이트 ---
```

```
t = s1 << 9
```

```
s2 = s2 xor s0
```

```
s3 = s3 xor s1
```

```
s1 = s1 xor s2
```

```
s0 = s0 xor s3
```

```
s2 = s2 xor t
```

```
s3 = rotr(s3, 11)
```

```
# --- 상태 저장 ---
```

```
state[0] = s0
```

```
state[1] = s1
```

```
state[2] = s2
```

```
state[3] = s3
```

## 09. 구현

### ChaCha20 암호화

- ① 초기 state를 4x4 행렬(16개의 32bit word)으로 구성  
(Key, Nounce, Counter, Constant 배치)
- ② state를 복사한 working\_state에  
**Quarter Round** (Column 4회 + Diagonal 4회) x 20회 반복
- ③ 20라운드 종료 후 working\_state와 원본 state를 word 단위로 더하기  
→ keystream
- ④ 원본 데이터와 keystream XOR  
→ 암호문 생성

#### Quarter Round

Add → XOR → rotate 과정

```
<Quarter Round>
// (a,b,c,d)는 32bit word
a = a + b
d = d ^ a
d = rotl(d,16)

c = c + d
b = b ^ c
b = rotl(b,12)

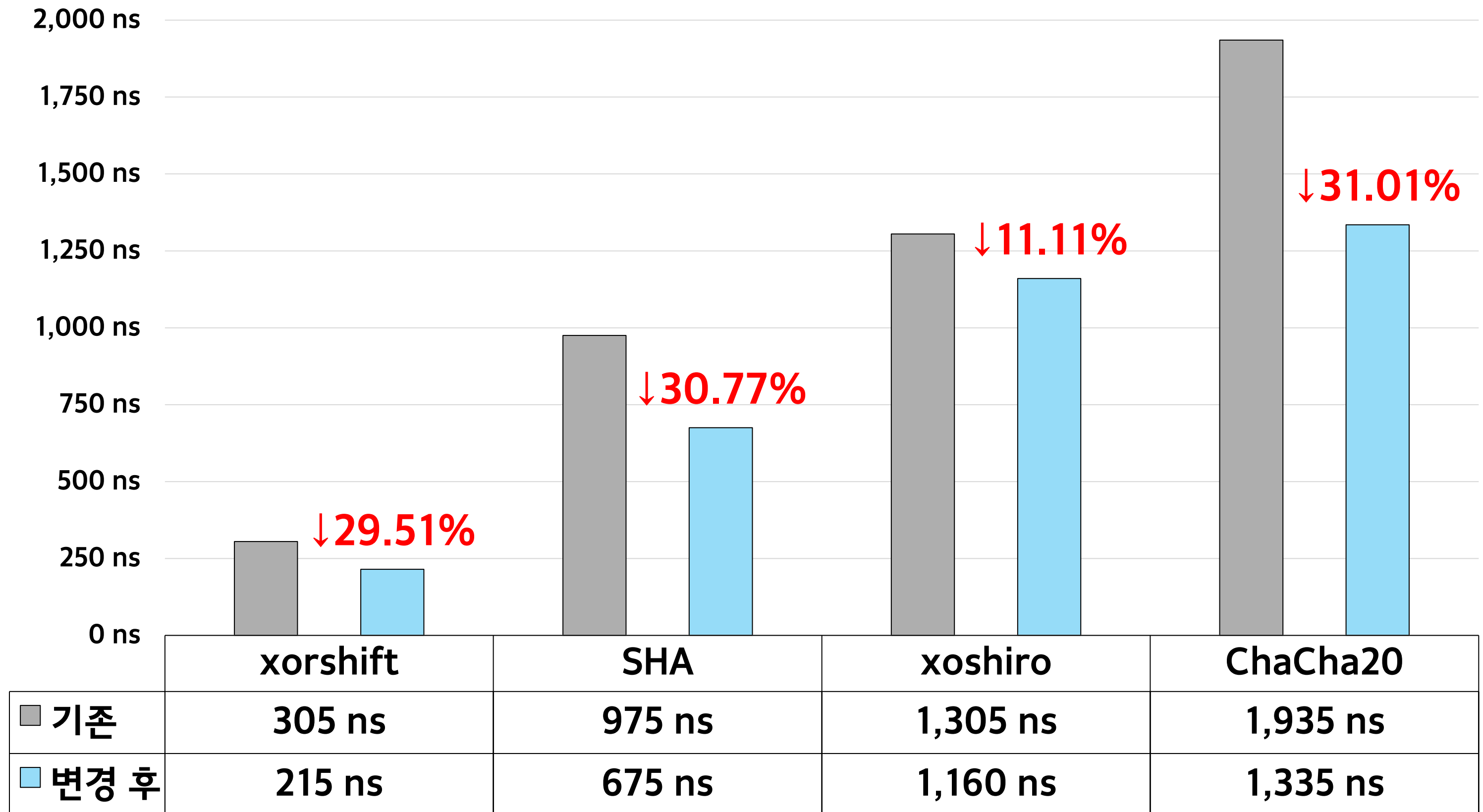
a = a + b
d = d ^ a
d = rotl(d,8)

c = c + d
b = b ^ c
b = rotl(b,7)
```



## 10. 비교

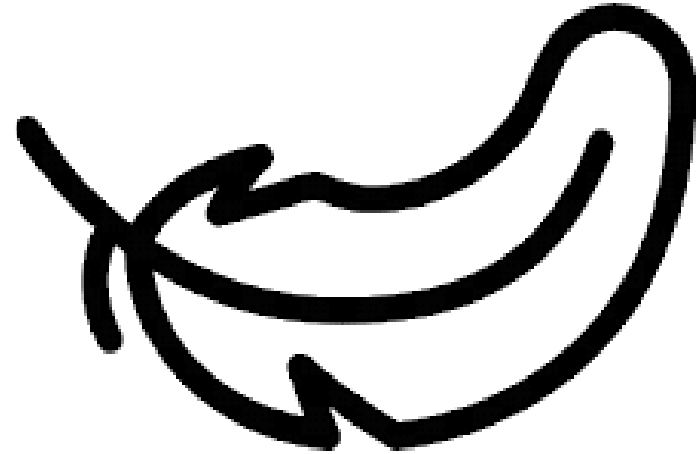
## 10. 비교



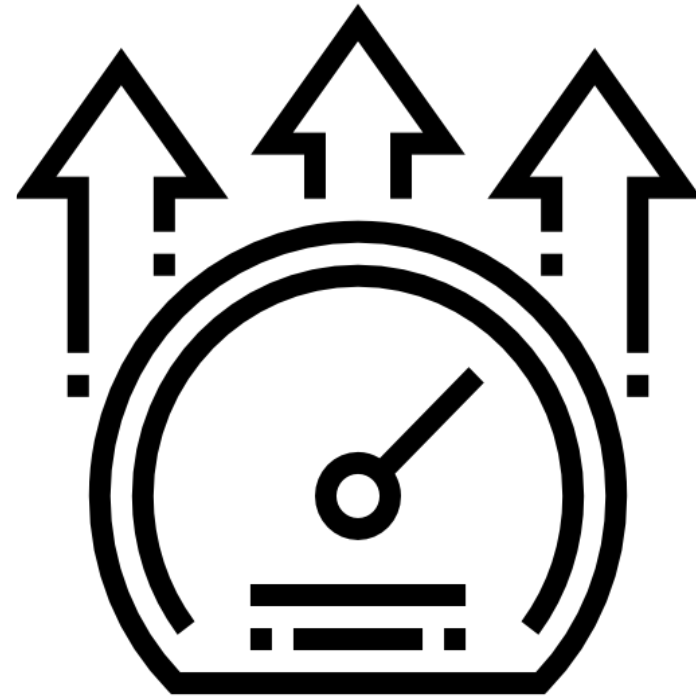
## 11. 결론

## 11. 결론

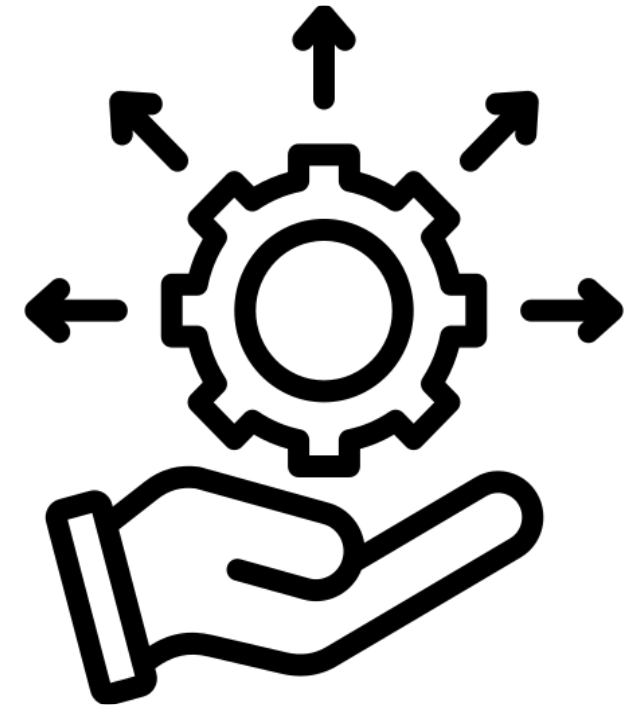
---



가벼움



빠른 속도



범용성