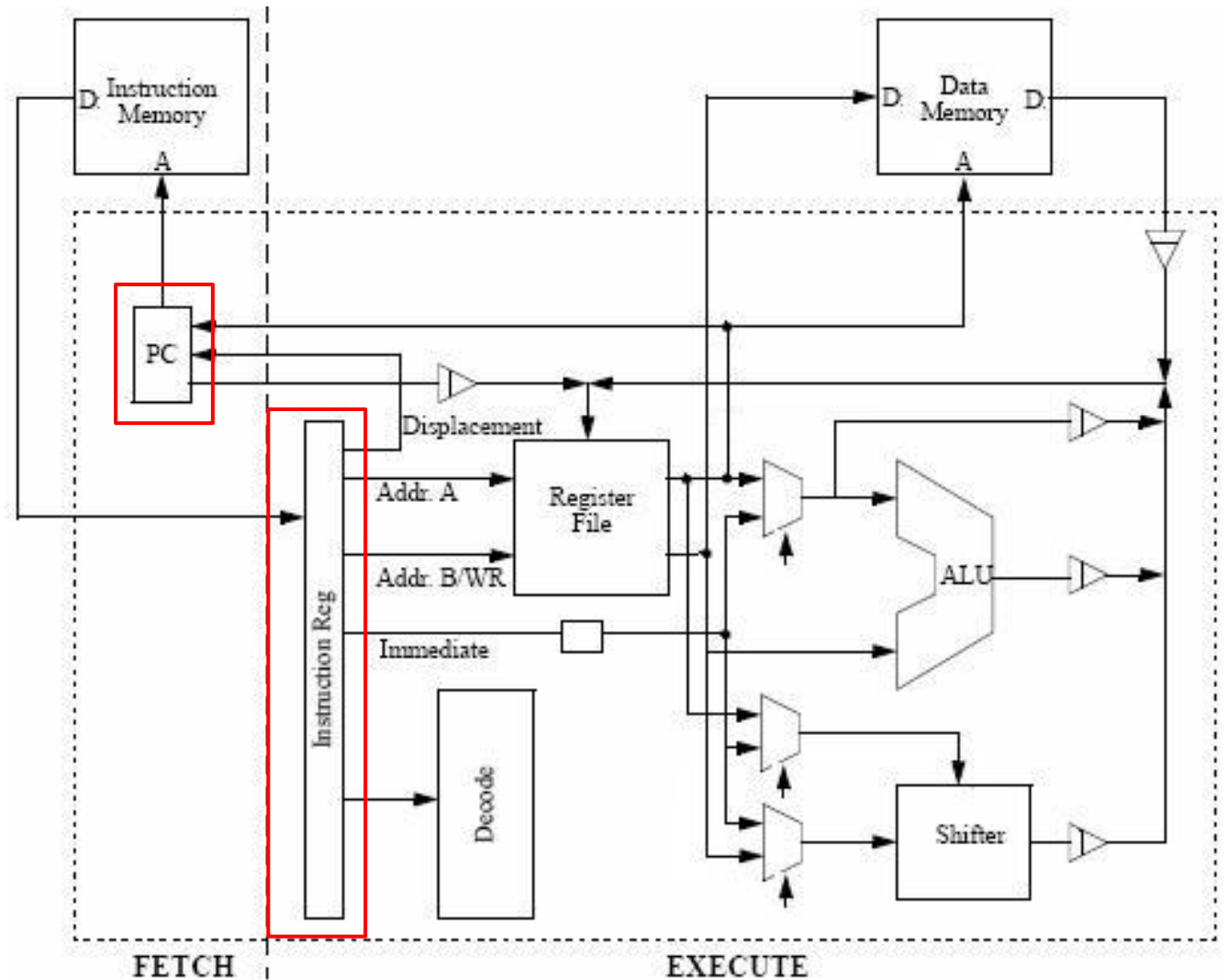


# Instruction Fetch & Decode

~~11/14 — 11/28 (11:59:59 pm)~~

# Instruction -Fetch Logic

- Consist of
  - Program counter (PC): address of next instruction
  - Instruction Register (IR): Register that stores currently executed register
- In this lab, we are going to implement these two components

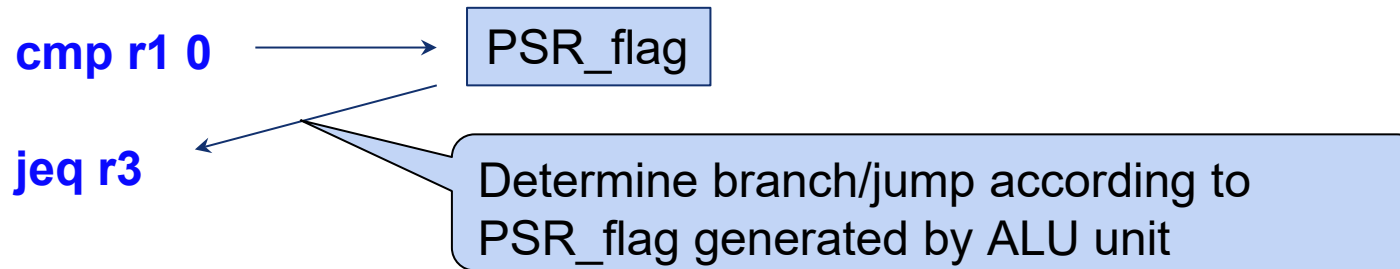


# Term Clarity: Classification of Instructions

- Normal instruction
  - Non-JAL/-Jcond/-Bcond instructions
  - PC value is incremented by 1 normally
- Bcond (conditional branch) instruction
  - Check condition, branch if condition is met
  - Update the PC by adding displacement to base address
- Jcond (conditional jump) instruction
  - Check condition, jump if condition is met
  - Read target instruction address from register file
- JAL (jump-and-link) instruction
  - Used for procedural call
  - Store the instruction address that is next to the JAL instruction into register file
  - Read target instruction address from register file and assign to PC

# Example Use of Jcond/ Bcond

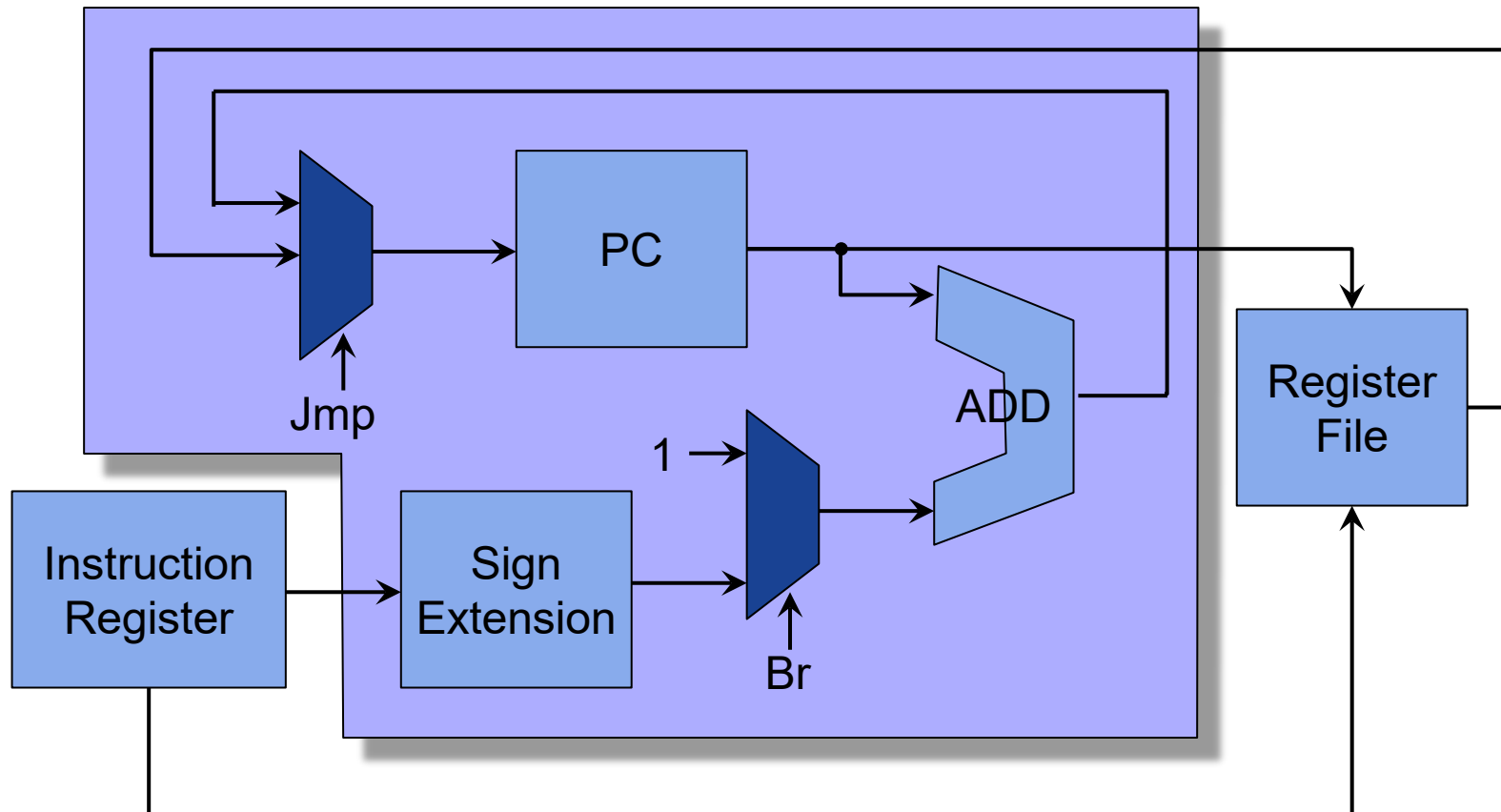
Jcond, Bcond instruction is always preceded by cmp/add/sub instruction



- Decoder checks condition along with PSR of cmp (i.e., updated this cycle)
- If condition is met, just jump!

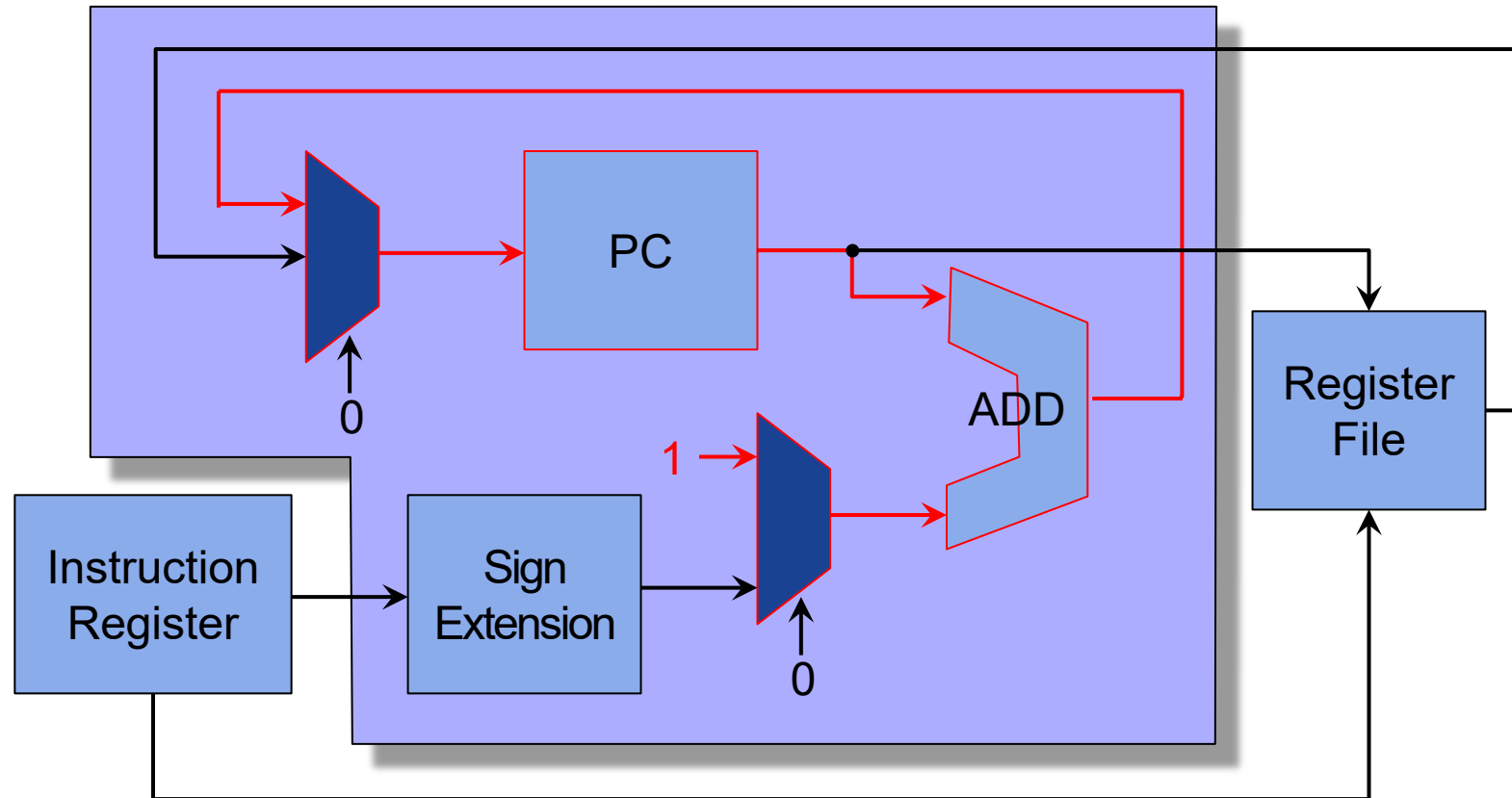
# Recap: 2-Stage Structure

- PC stores next instruction to be executed
- IR store currently executed instruction



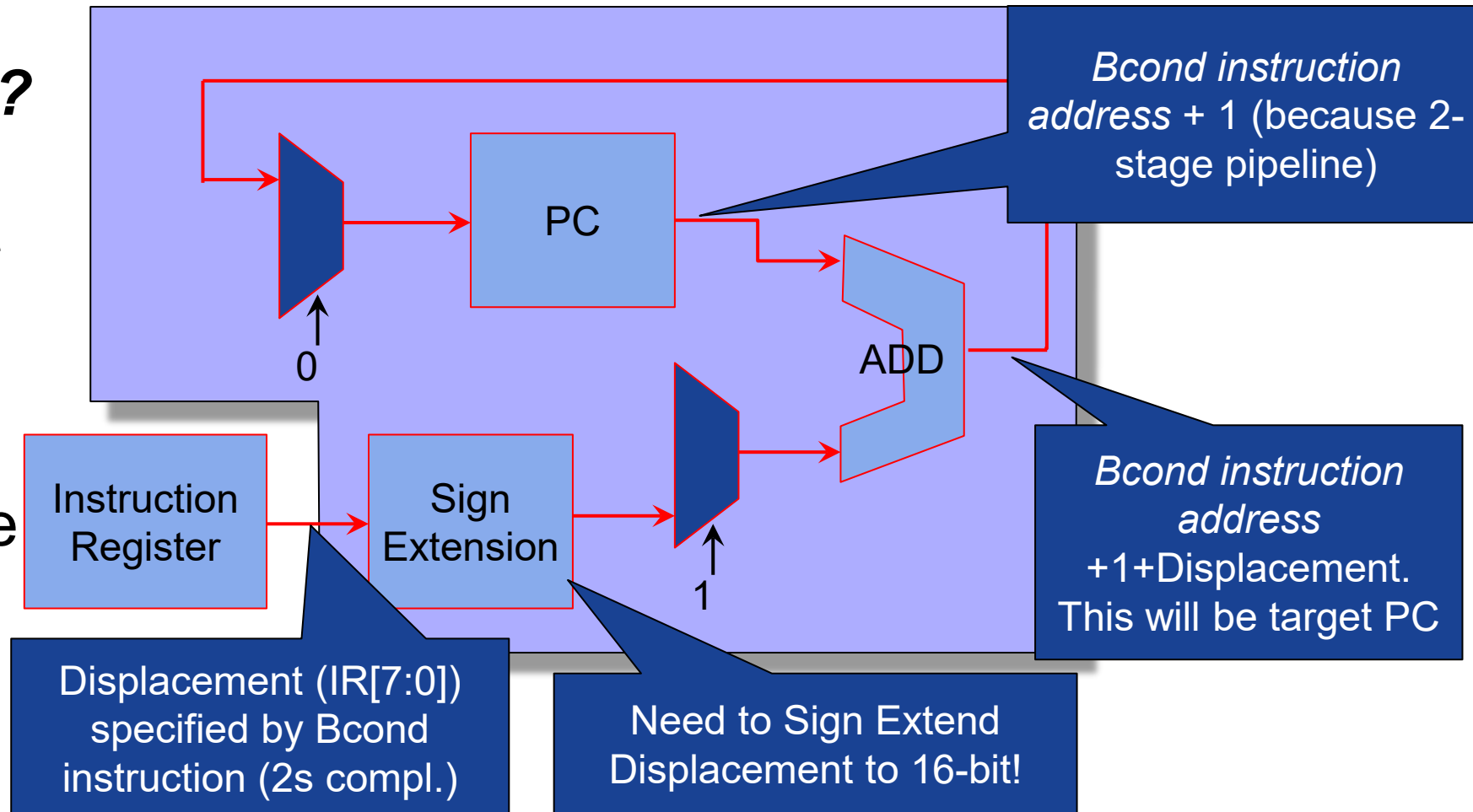
# Program Counter Operation: Normal

- REMEMBER: PC is incremented by 1 in this lab



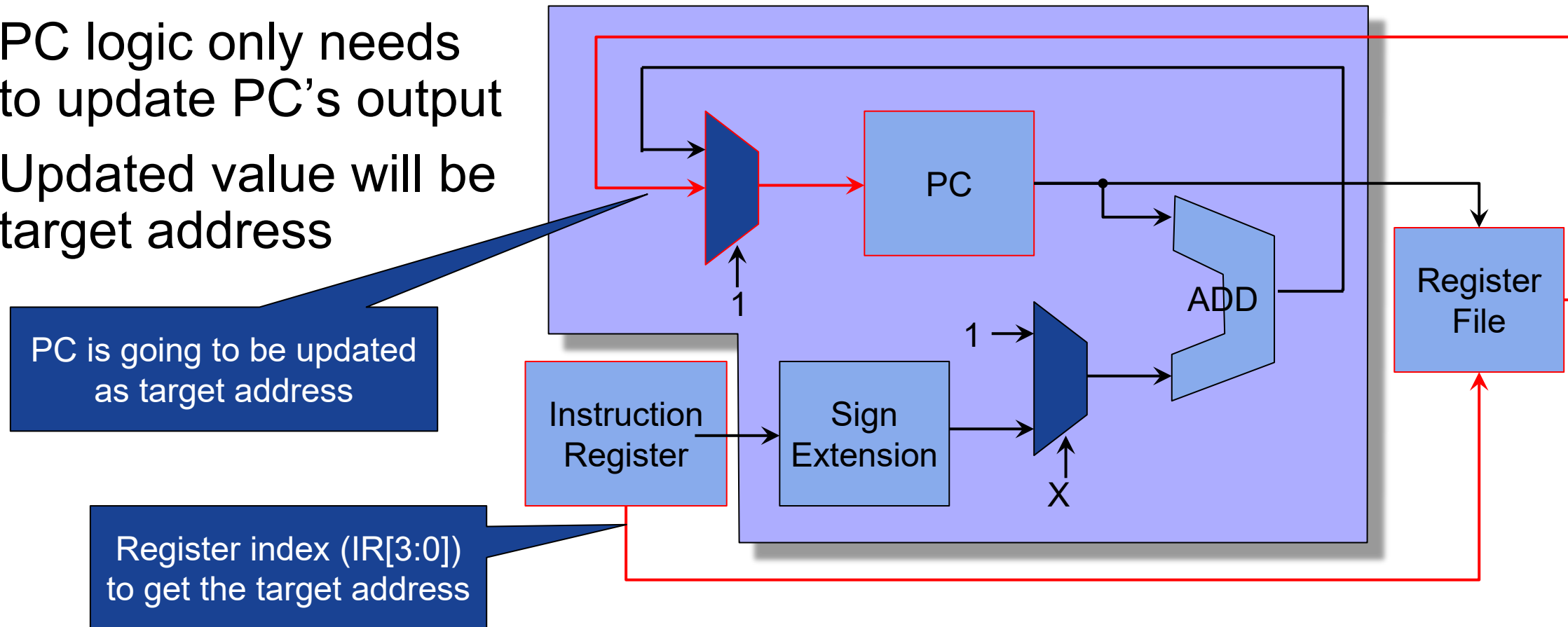
# Program Counter Operation: Bcond

- **Displacement?**  
→ difference between target instruction and the current PC
- Need to sign extend it before calculating target address



# Program Counter Operation: Jcond & JAL

- PC logic only needs to update PC's output
- Updated value will be target address





# TODO: Pseudo Code for PC

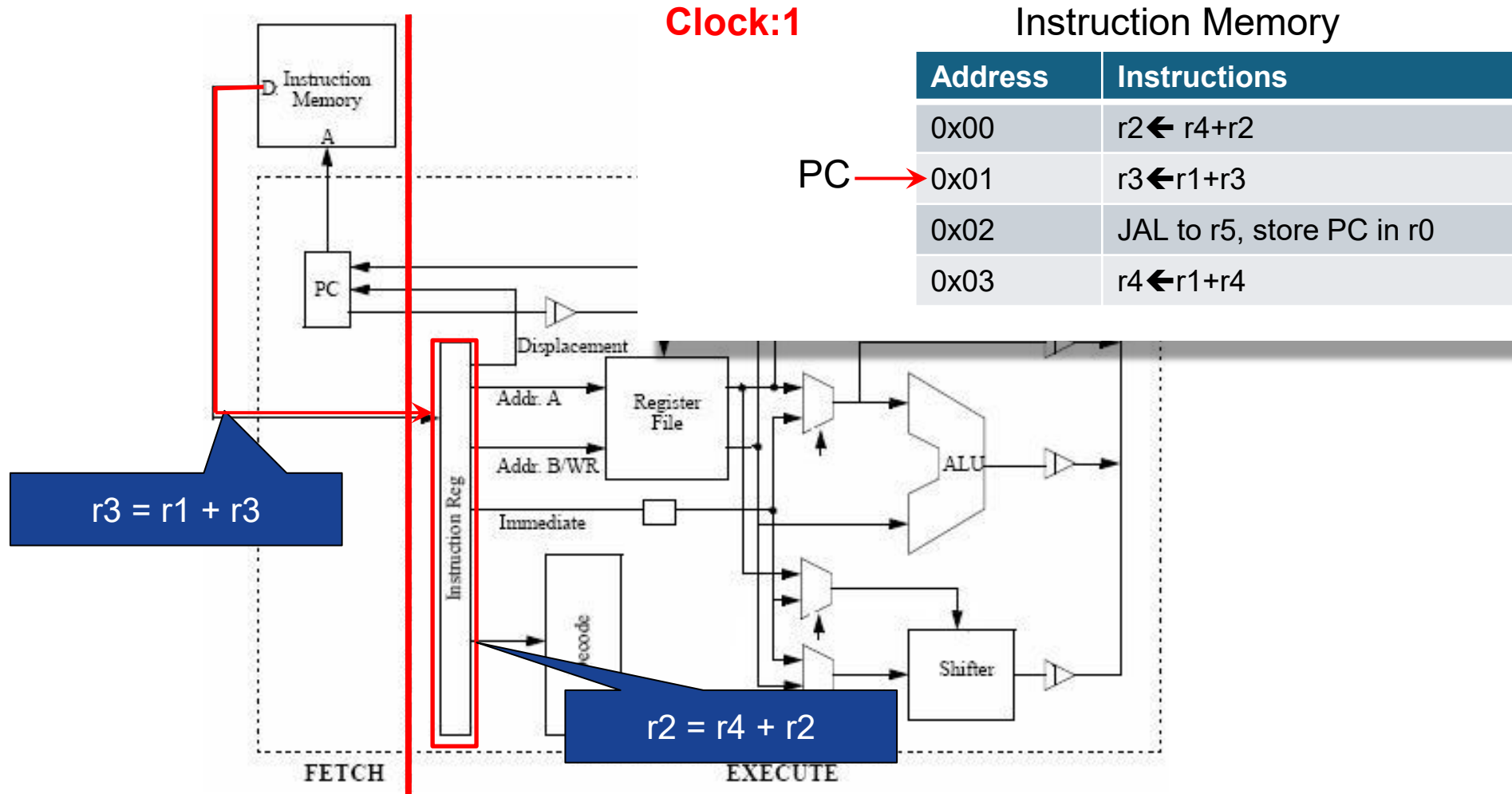
- You are going to implement this in pc.v

```
always@(posedge clk or negedge reset_b)
if (reset_b==0) output<=0;
else
    jmp=1 : addr_imem <= src;
    br =1 : addr_imem <= addr_imem+extend(displace)
    else  : addr_imem++;
```

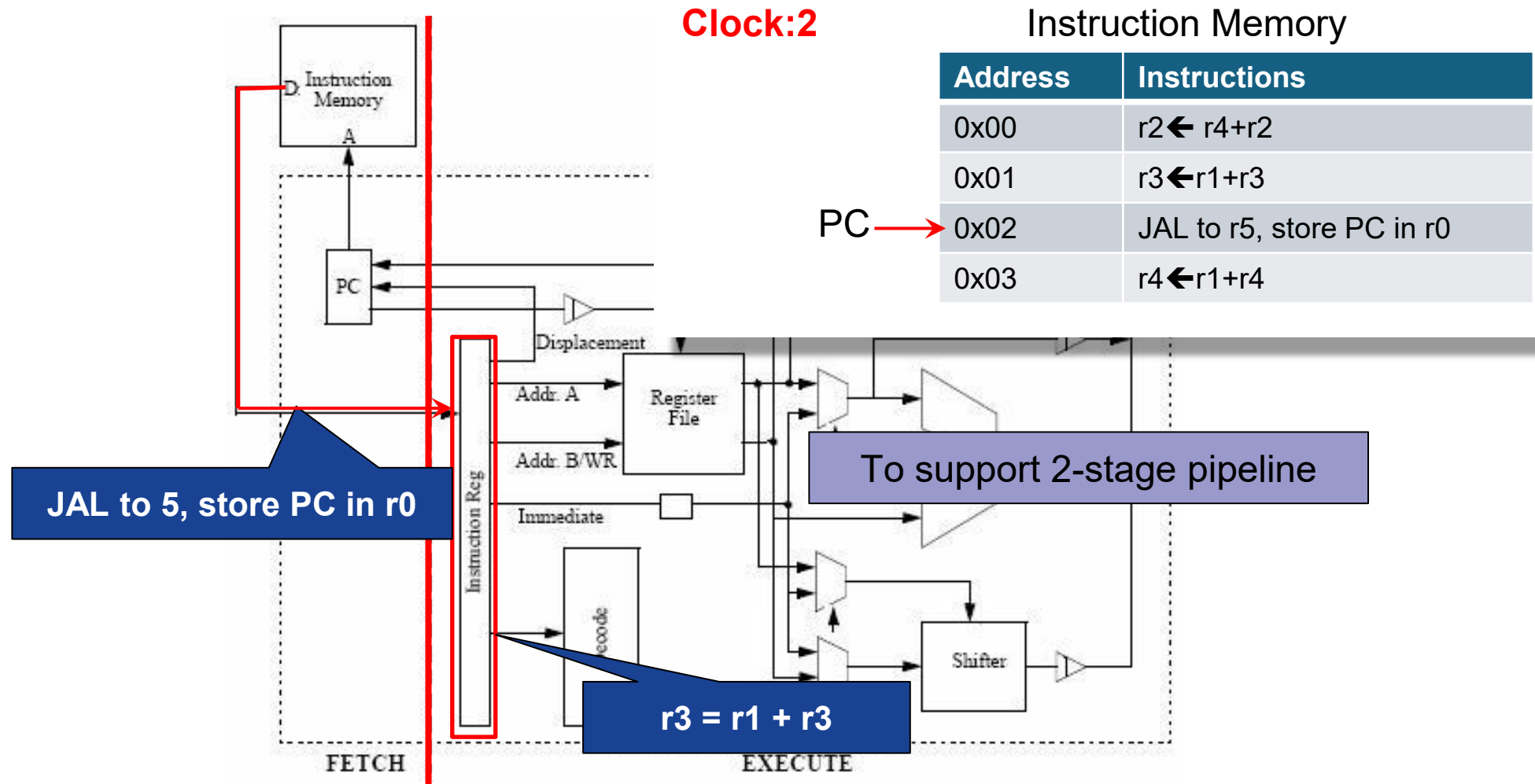
# Instruction Register

- Latches instruction from instruction memory
- Width of 16-bit in this lab → 16-bit instruction
- Corner case to consider
  - Ensure not simply executing instruction right after branch instructions

# Design Consideration

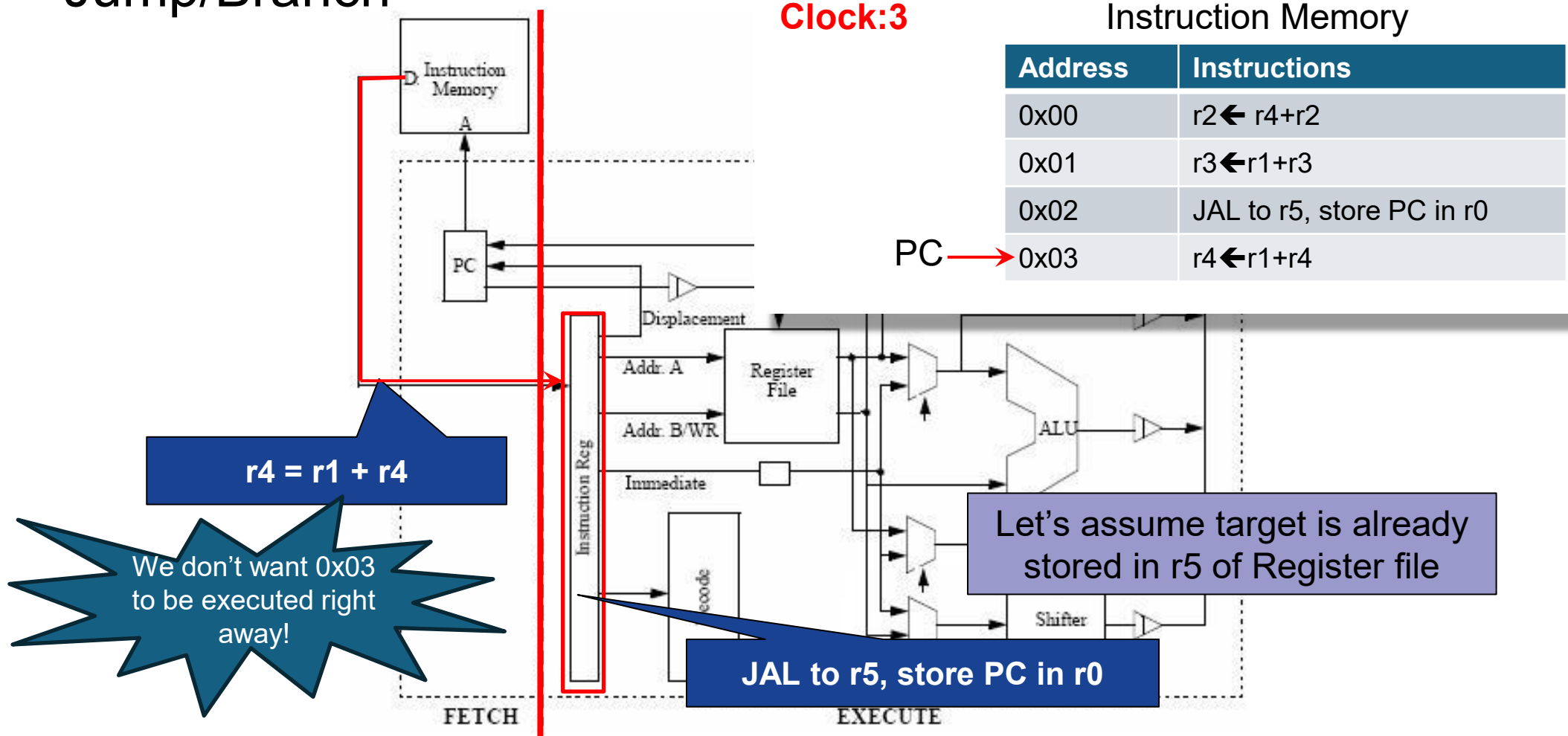


# Design Consideration



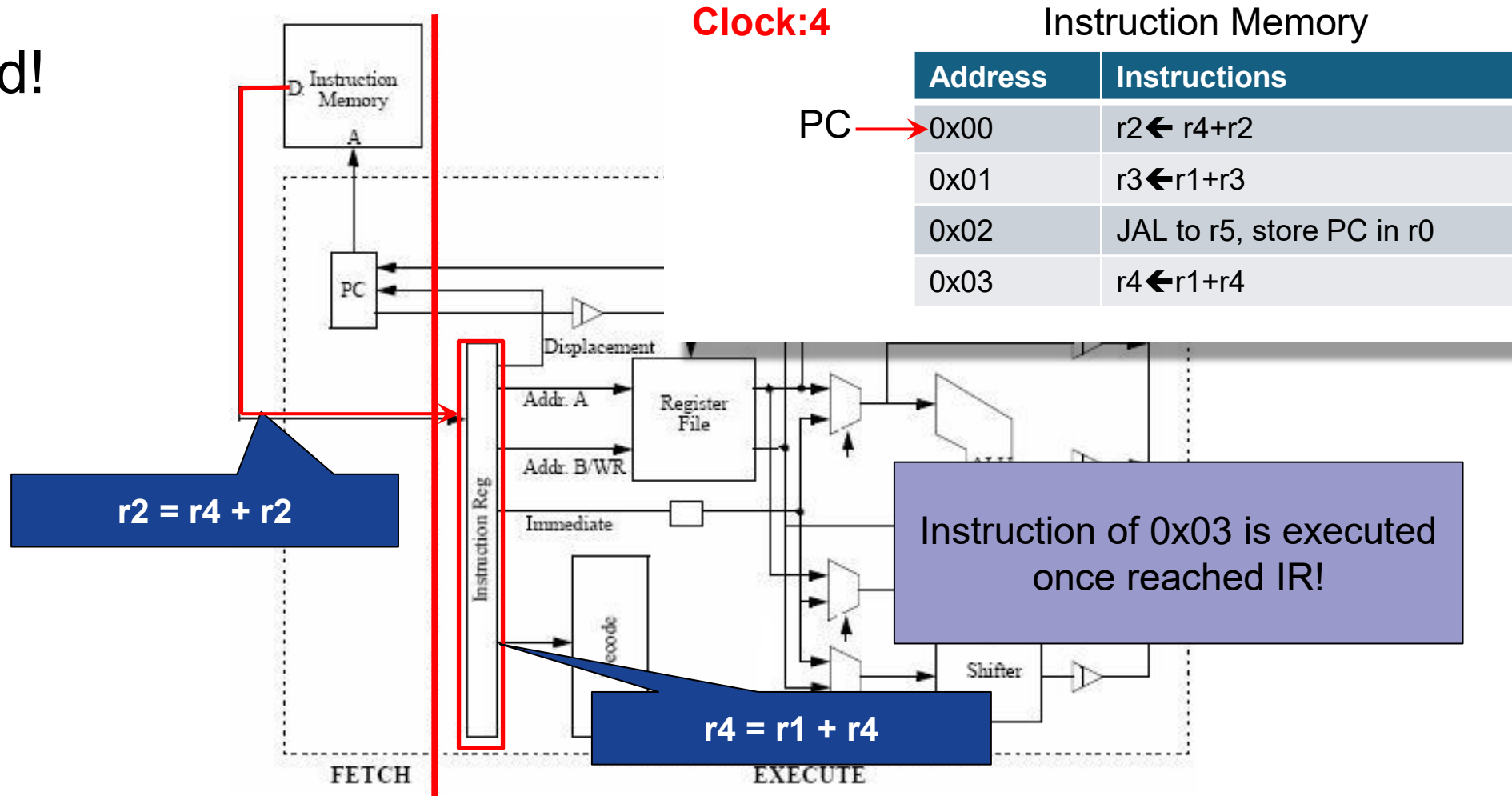
# Design Consideration

- Jump/Branch



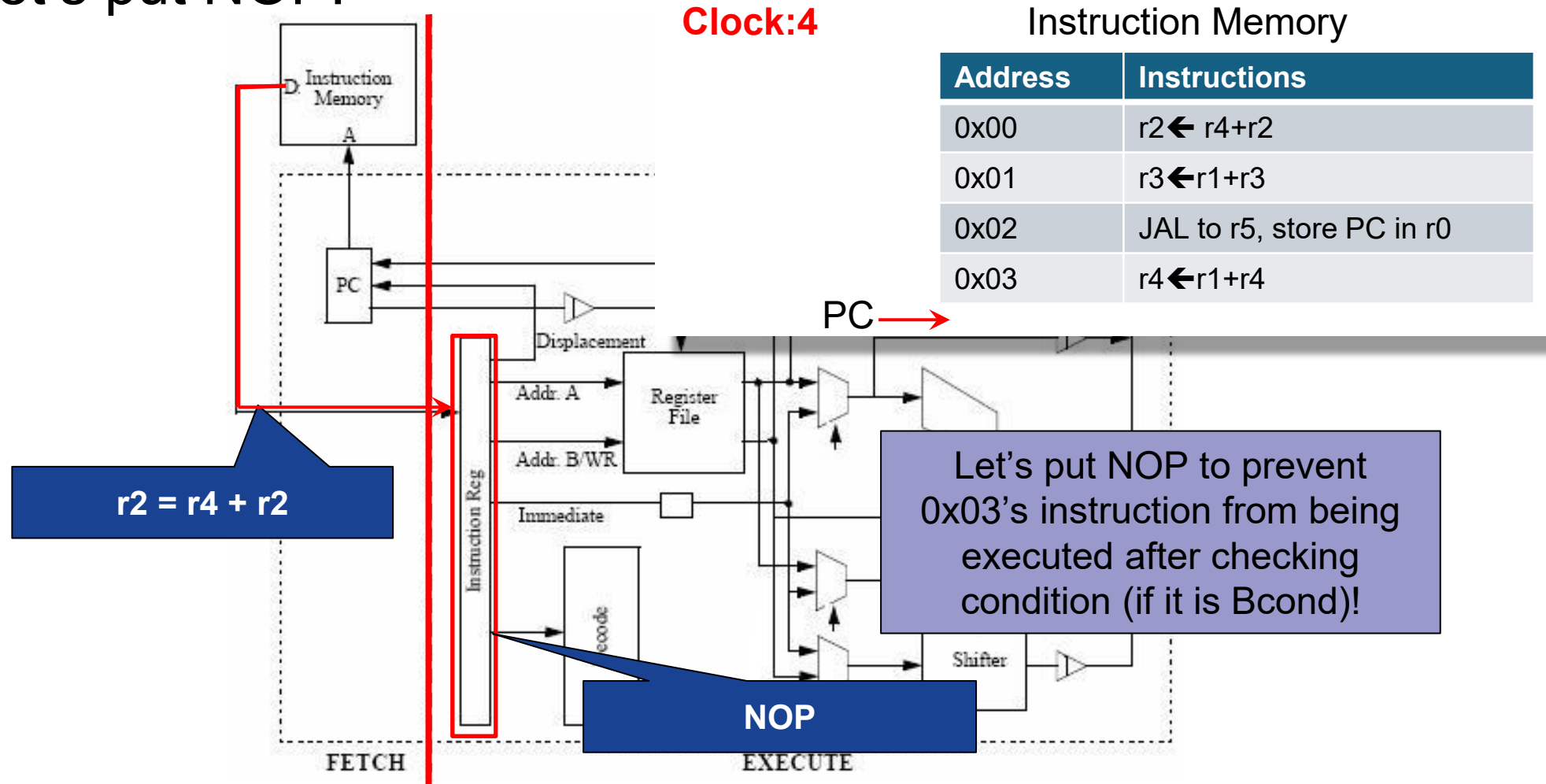
# Design Consideration

- Bad!



# Design Consideration

- Let's put NOP!



# TODO: Pseudo Code for IR

- You are going to implement this in instruction\_reg.v
- addr, immediate, disp should be determined by new instructions

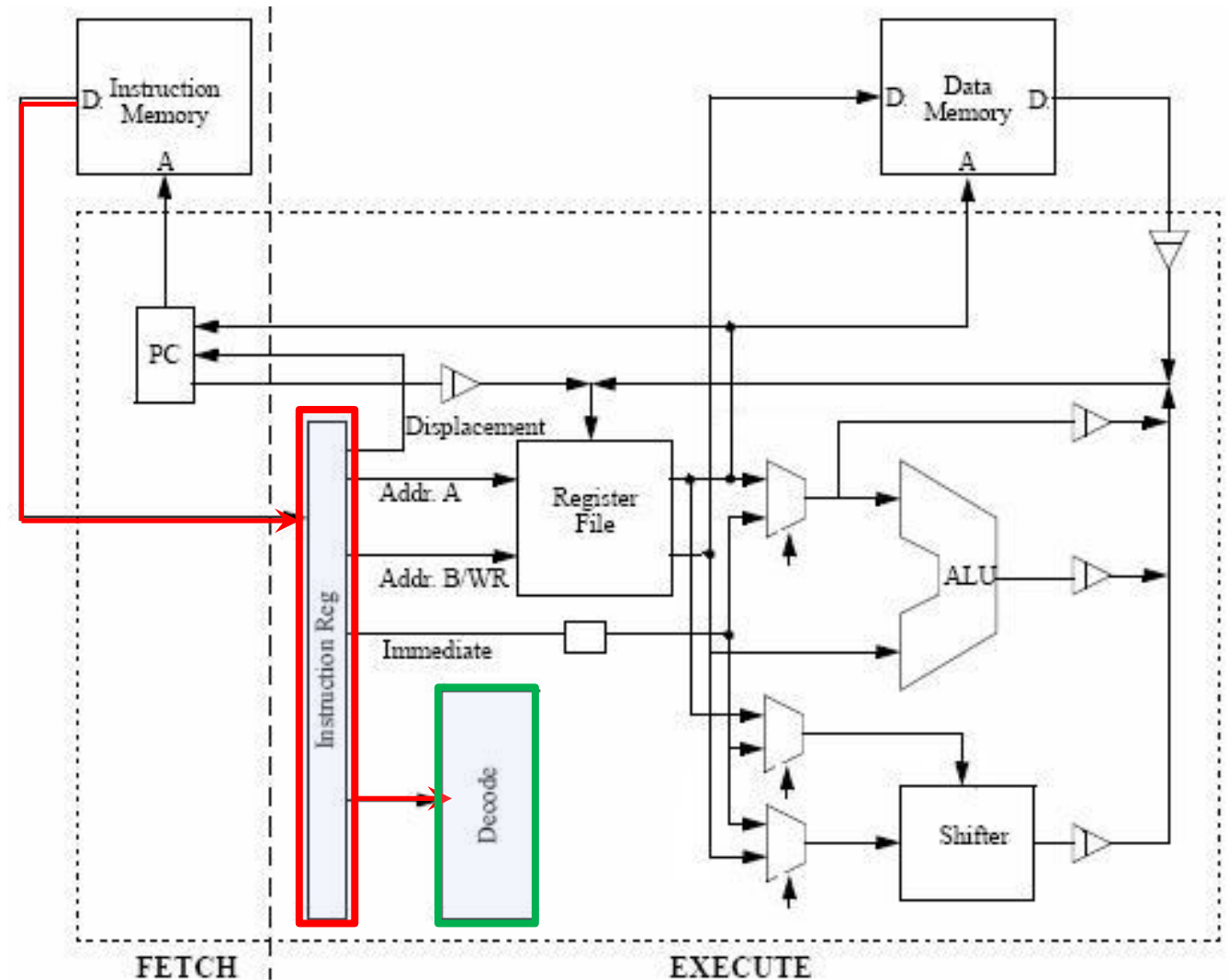
```
always@(posedge clk or negedge reset_b)
  if (reset_b==0)
    reset;
  else
    if (jmp==1||br==1)
      instruction_out = NOP;
    else
      instruction_out = inst_in;

  addr_a = instruction_out[3:0];
  addr_b = instruction_out[11:8];
  immediate = instruction_out[7:0];
  disp      = instruction_out[7:0];
```



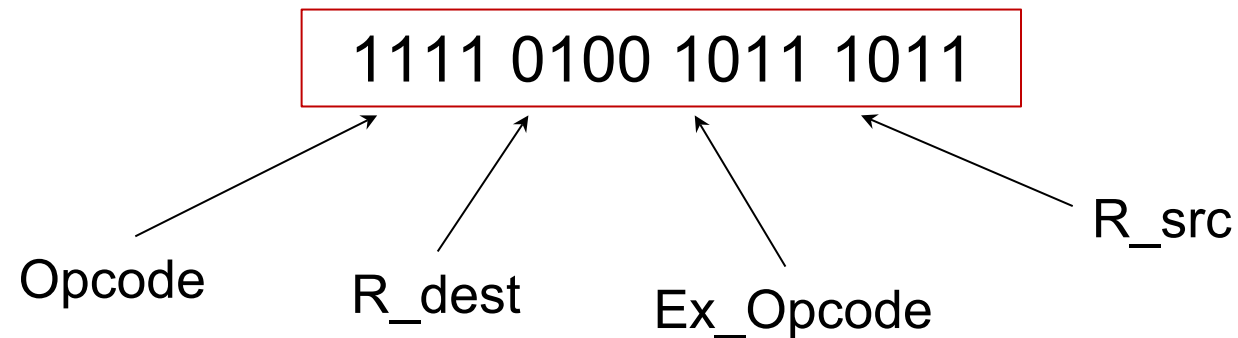
# Necessity of Instruction Decoder

- Previously we knew how to fetch instruction and update PC according to opcodes
- But, how to control logics (e.g., ALU) after fetching instruction @ IR stage?
- The decoder generates these control signals!



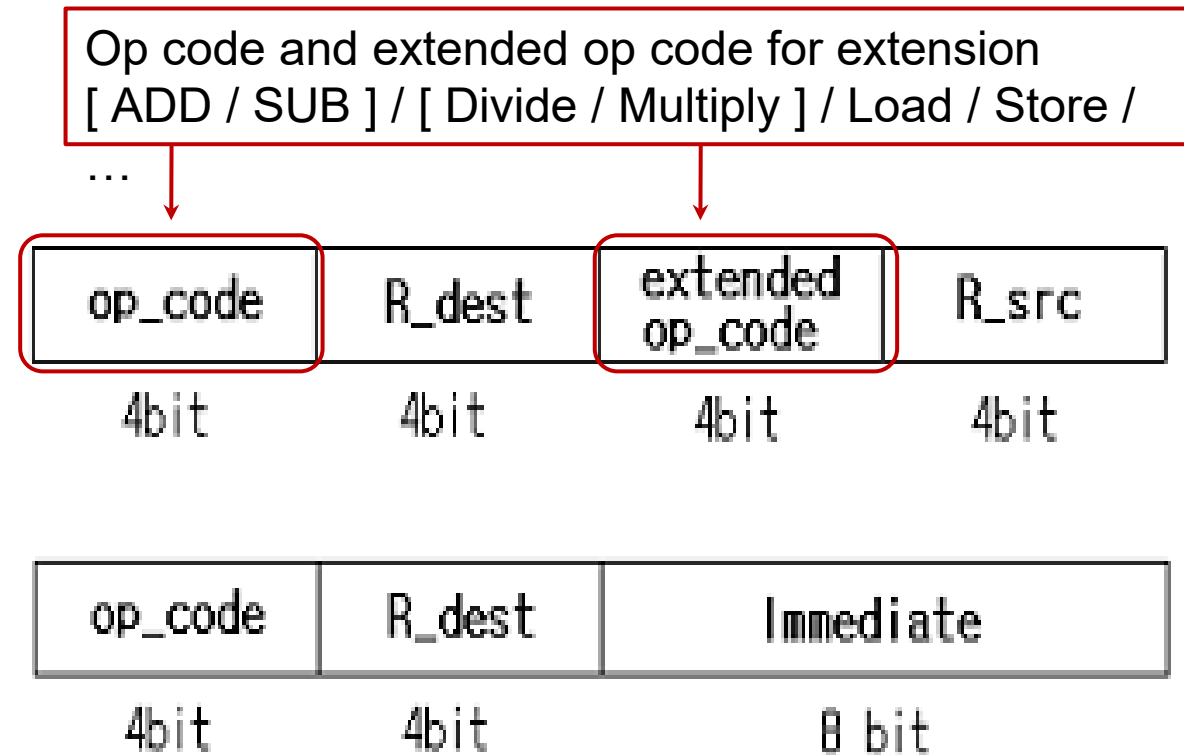
# ISA and Decoder

- Decoder knows the structure of ISA
- Knows which part is opcode
- Knows which part is register file's address
- Knows how many registers we are going to use...

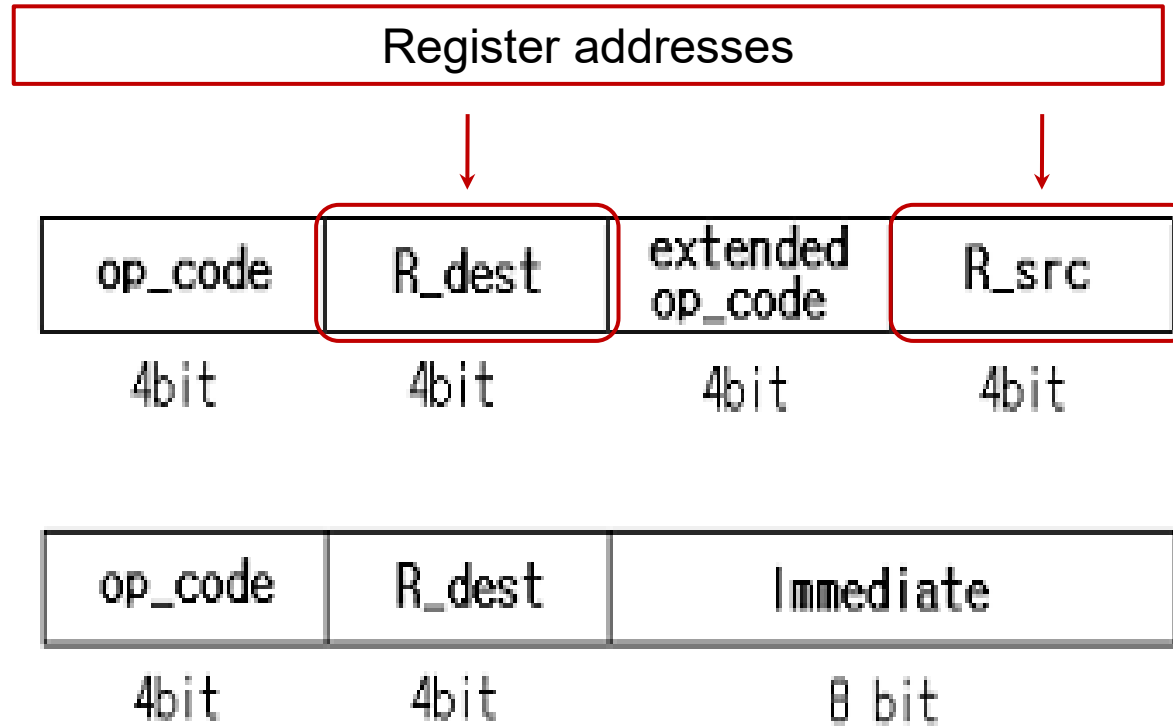


# ISA in Our Lab (1)

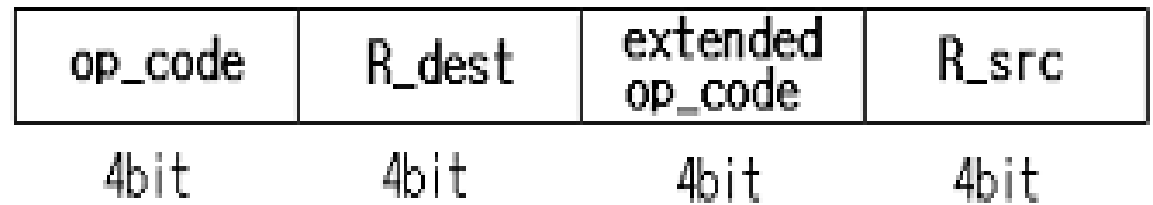
- We are not going to implement divide and multiply



# ISA in Our Lab (2)



# ISA in Our Lab (2)



Immediate field just like i-type instruction

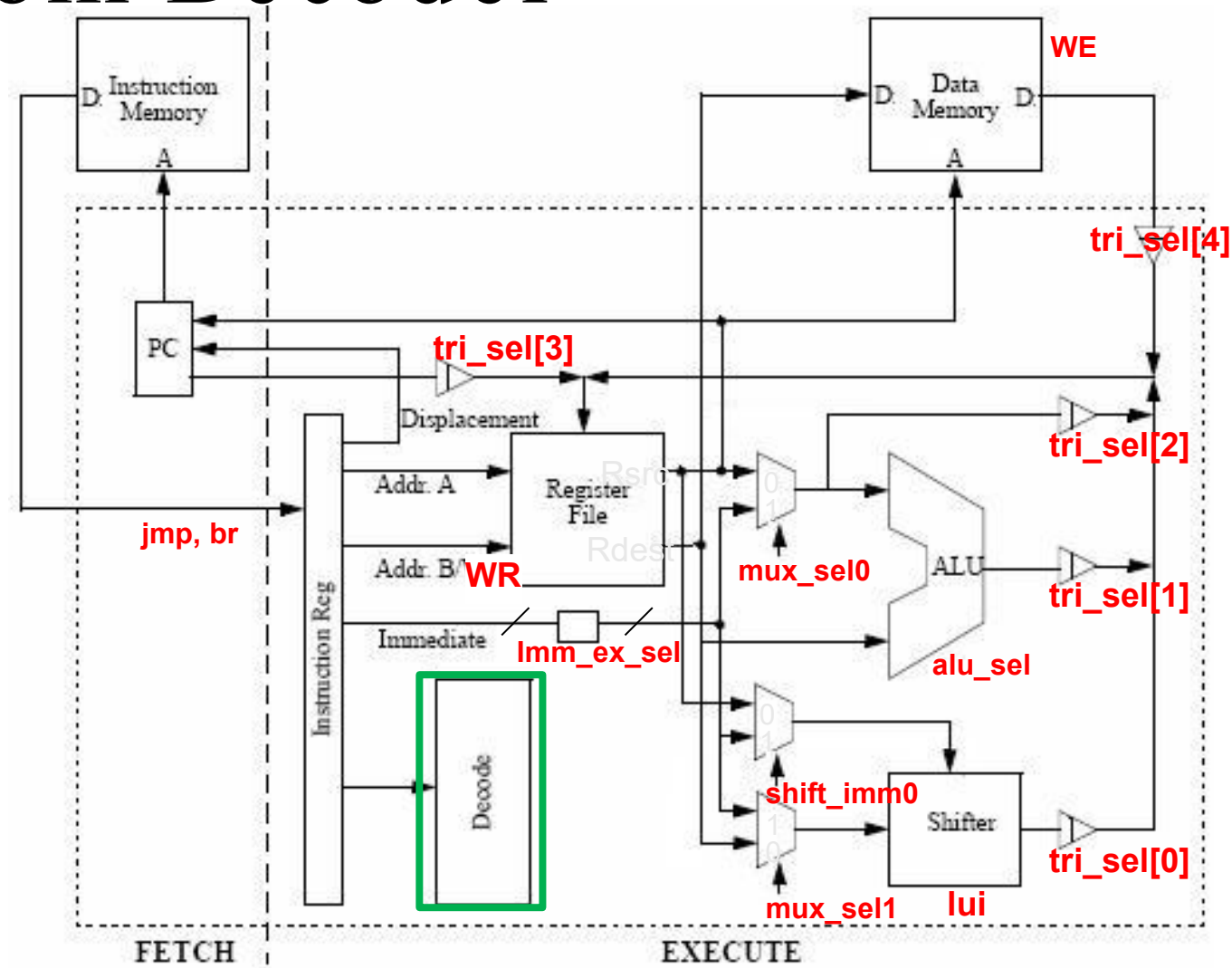
# Decoder's View of Instructions

- OP code
  - ADDI, SUBI, CMPI, ANDI, ORI, XORI
  - MOVI
  - LUI
- OP code & OP code Ext
  - ADD, SUB, CMP, AND, OR, XOR
  - MOV
  - LOAD, STOR
  - LSH, LSHI
  - JAL
- OP code & condition
  - Jcond, Bcond

# Instruction Opcodes (in decoder.v)

Opcode	Opcode-ext	Instruction
0000	0101	ADD
	1001	SUB
	1011	CMP
	0001	AND
	0010	OR
	0011	XOR
	1101	MOV
0100	0000	LOAD
	0100	STORE
	1100	Jcond
	1000	JAL
0101	-	ADDI
1000	Opcode-ext[3:1] != 3'b000	LSH
	Opcode-ext[3:1] == 3'b000	LSHI
1001	-	SUBI
1011	-	CMPI
0001	-	ANDI
0010	-	ORI
0011	-	XORI
1101	-	MOVI
1111	-	LUI
1100	-	Bcond

# Overview of Control Signals (Red) from Decoder





# Decoder's In/Out Specification

	Port Name	설명
input	Instruction[15:0]	Output from IR
	PSR_flag[4:0]	FLCNZ flag from PSR. Order of [4:0] = {F, L, C, N, Z}
output	tri_sel[4:0]	Tri State Buffer의 control (One-hot form)
	alu_sel[5:0]	ALU's mode control
	WR	Register Write Enable control
	Jmp	PC의 jmp mux control
	Br	PC의 br mux control
	mux_sel0	ALU의 data input control
	mux_sel1	Shifter의 data input control
	shift_imm	Shifter의 Rlamount input control
	Lui	Shifter의 lui(8-bit left shift) control
	WE	Data Memory Write Enable control
	imm_ex_sel	Extension mode control(signed or unsigned)

# Decoder Table (1): CHECK the Code!

	tri_sel	alu_sel	WR	jmp	br	mux_sel0	mux_sel1	shift_imm	lui	WE	imm_ex_sel
ALUi	2	?	?	0	0	1	x	x	x	0	?
ALU	2	?	?	0	0	0	x	x	x	0	x
movi	4	x	1	0	0	1	x	x	x	0	0
mov	4	x	1	0	0	0	x	x	x	0	x
lshi	1	x	1	0	0	x	0	1	0	0	x
lsh	1	x	1	0	0	x	0	0	0	0	x
lui	1	x	1	0	0	x	1	x	1	0	x
load	16	x	1	0	0	x	x	x	x	0	x
stor	0	x	0	0	0	x	x	x	x	1	x
jal	8	x	1	1	0	x	x	x	x	0	x
jcond	0	x	0	1	0	x	x	x	x	0	x
bcond	0	x	0	0	1	x	x	x	x	0	x

# Decoder Table (2)

Instruction	alu_sel	WR	imm_ex_sel
ADDI	6'b10_0000	1	1
ADD	6'b10_0000	1	x
SUBI	6'b01_0000	1	1
SUB	6'b01_0000	1	x
CMPI	6'b00_1000	0	1
CMP	6'b00_1000	0	x
ANDI	6'b00_0100	1	0
AND	6'b00_0100	1	x
ORI	6'b00_0010	1	0
OR	6'b00_0010	1	x
XORI	6'b00_0001	1	0
XOR	6'b00_0001	1	x

'Cond' bit	PSR_Values	jmp or br
0000	Z=1	1
0001	Z=0	1
1101	N=1 or Z=1	1
0010	C=1	1
0011	C=0	1
0100	L=1	1
0101	L=0	1
1010	L=0 and Z=0	1
1011	L=1 or Z=1	1
0110	N=1	1
0111	N=0	1
1000	F=1	1
1001	F=0	1
1100	N=0 and Z=0	1
1110	X	1
1111	X	0

# Cond Bits Interpretation in Rdest/cond Field

N: not; S: signed; O: overflow; LT: less than; LE: less than or equal

- 0000: EQ      0001: NEQ      → CMP
- 0010: O      0011: NO      → ADD
- 0100: LT      0101: NLT      → CMP
- 0110: SLT      0111: SNLT      → CMP (N)
- 1000: SO      1001: SNO      → SUB
- 1010: NLE      1011: LE      → CMP
- 1100: SNLE      1101: SLE      → CMP (NZ)
- 1110: Just Jump

# TODO: Implement Jcond/ Bcond Table

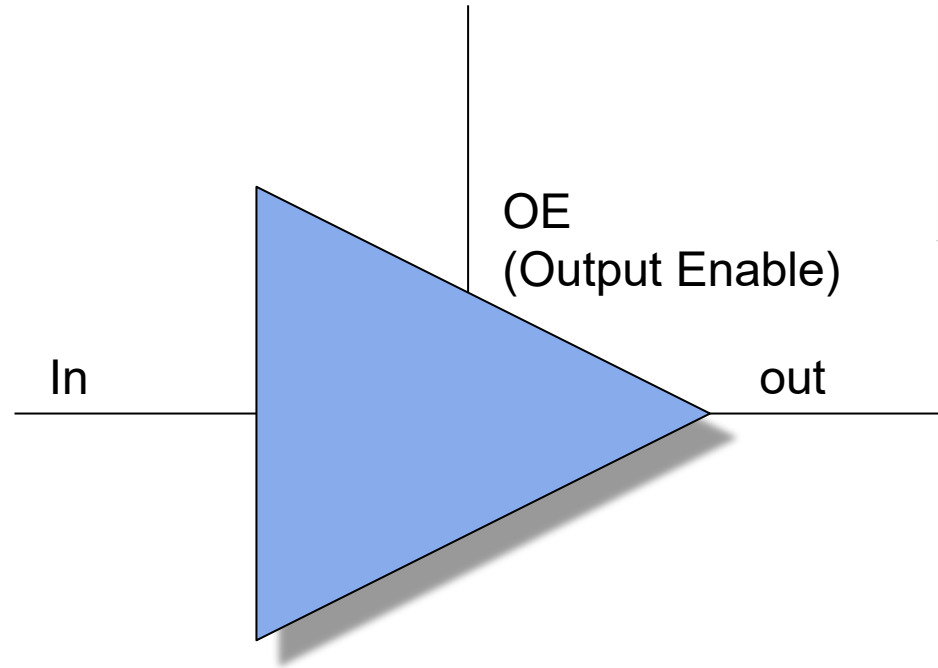
Current  
Bcond  
Jcond  
Instruction's  
Cond bits  
(or Rdest field)

'Cond' bit	PSR Values	jmp or br
0000	Z=1	1
0001	Z=0	1
1101	N=1 or Z=1	1
0010	C=1	1
0011	C=0	1
0100	L=1	1
0101	L=0	1
1010	L=0 and Z=0	1
1011	L=1 or Z=1	1
0110	N=1	1
0111	N=0	1
1000	F=1	1
1001	F=0	1
1100	N=0 and Z=0	1
1110	X	1
1111	X	0

PSR-flag from previous  
Cmp or ALU results

# Tri-State Buffers?

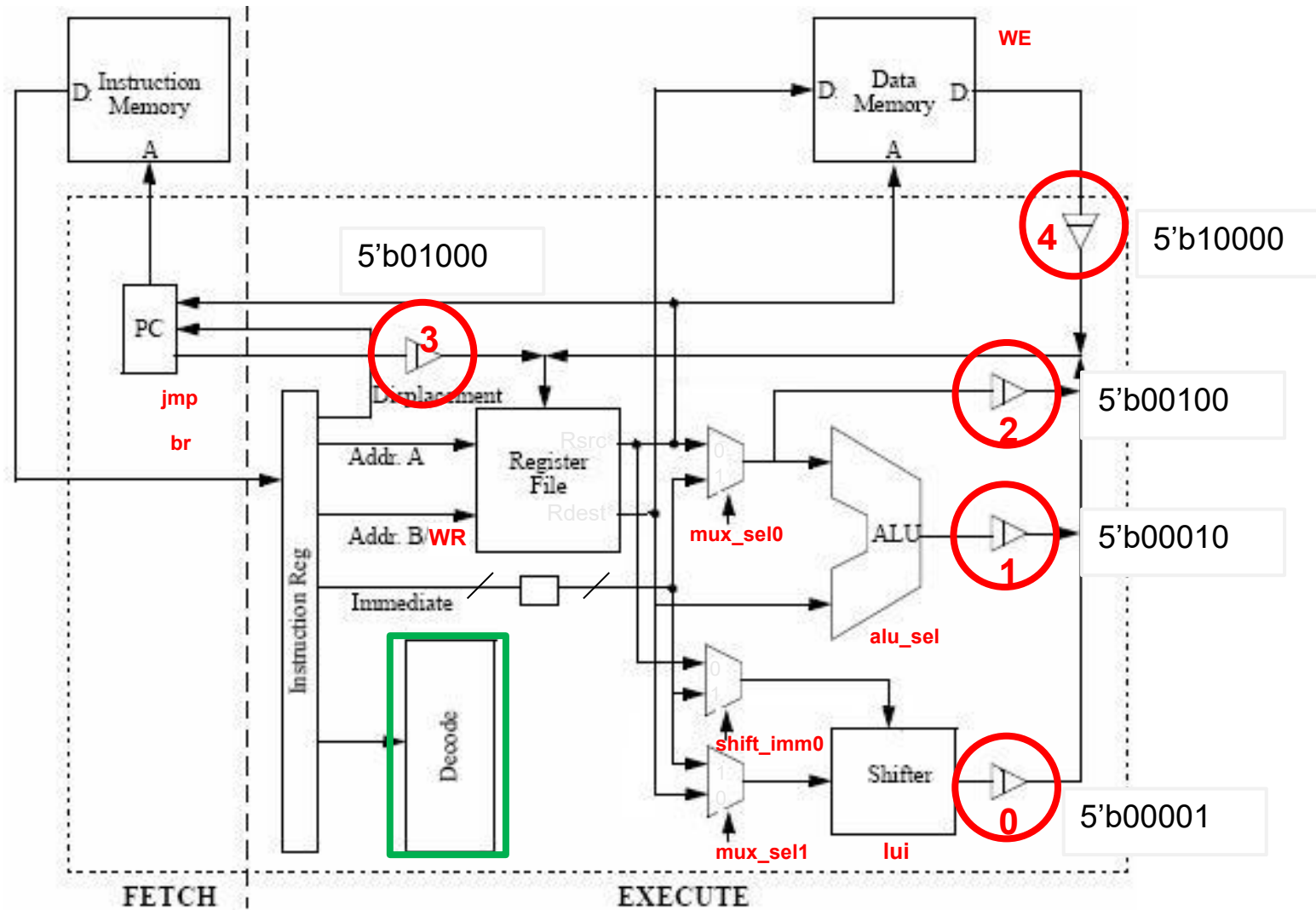
- Z indicates high-impedance → input is blocked  
- 16'bzzzz\_zzzz\_zzzz\_zzzz



IN	OE	OUT
X	0	Hi-Z
0	1	0
1	1	1

# Tri-State Buffers in This Processor

- Some signals share the line
- Need to prevent conflict of signals
- Need to use tri-state buffer to block undesired signal propagation

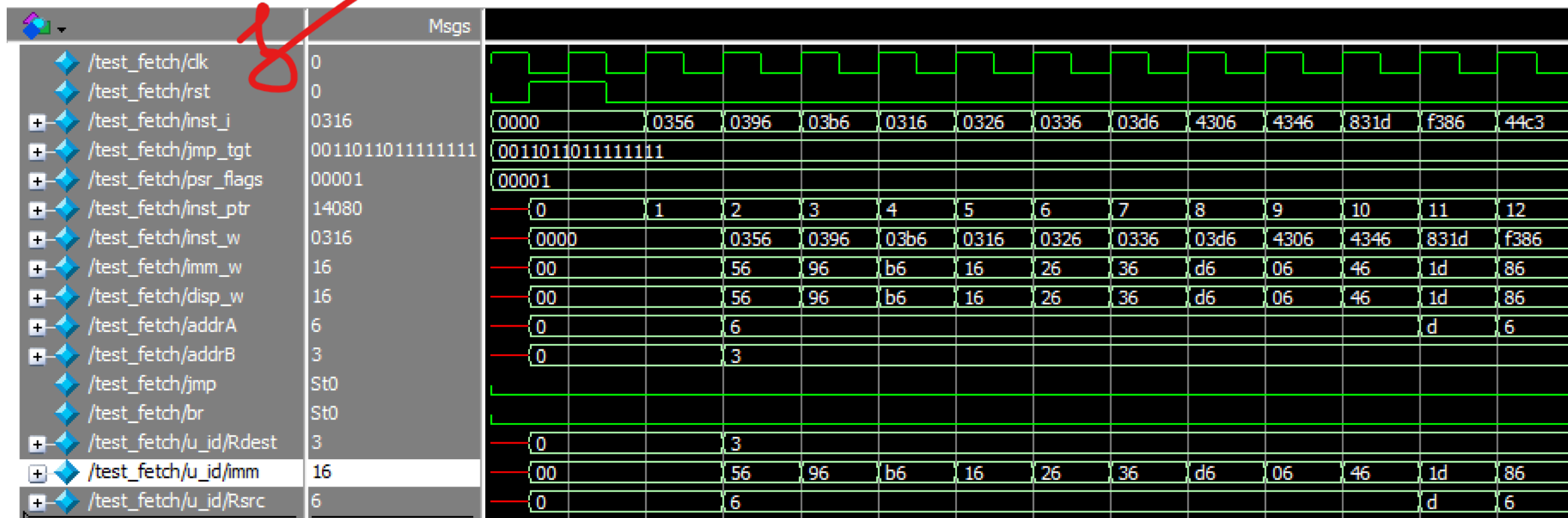


# Assignment

















- In your report...
  - Design and explain PC logic (pc.v) – 20%
  - Design and explain IR logic (instruction\_reg.v) – 20%
  - Design and explain JUMP logic in decoder (decoder.v) – 20%
  - Run testbench and validate waveforms in next slides (test\_fetch.v) – 40%
    - Please attach your capture of waveforms
- Let's define NOP as “0x0020” @ IR stage
- You need to use codes from lab2
- TIPS: Please refer to the backup slides – it might be very, very useful for understanding!



# Desired Waveforms: Usual



# Desired Waveforms: Jcond/ Bcond

	Msgs	
 /test_fetch/dk	0	
 /test_fetch/rst	0	
 /test_fetch/inst_i	0316	f386 44c3 40c3 0316 c4fd c0fd 0316
 /test_fetch/jmp_tgt	0011011011111111	0011011011111111
 /test_fetch/psr_flags	00001	00001
 /test_fetch/inst_ptr	14084	11 12 13 14 14079 14080 14081 14078 14079
 /test_fetch/inst_w	0316	831d f386 44c3 40c3 0020 c4fd c0fd 0020 0316
 /test_fetch/imm_w	16	1d 86 c3 20 fd 20 16
 /test_fetch/disp_w	16	1d 86 c3 20 fd 20 16
 /test_fetch/addrA	6	d 6 3 0 d 0 6
 /test_fetch/addrB	3	3 4 0 4 0 3
 /test_fetch/jmp	0	
 /test_fetch/br	0	
 /test_fetch/u_id/Rdest	3	3 4 0 4 0 3
 /test_fetch/u_id/imm	16	1d 86 c3 20 fd 20 16
 /test_fetch/u_id/Rsrc	6	d 6 3 0 d 0 6

# Desired Waveforms: Unconditional JMP/JAL

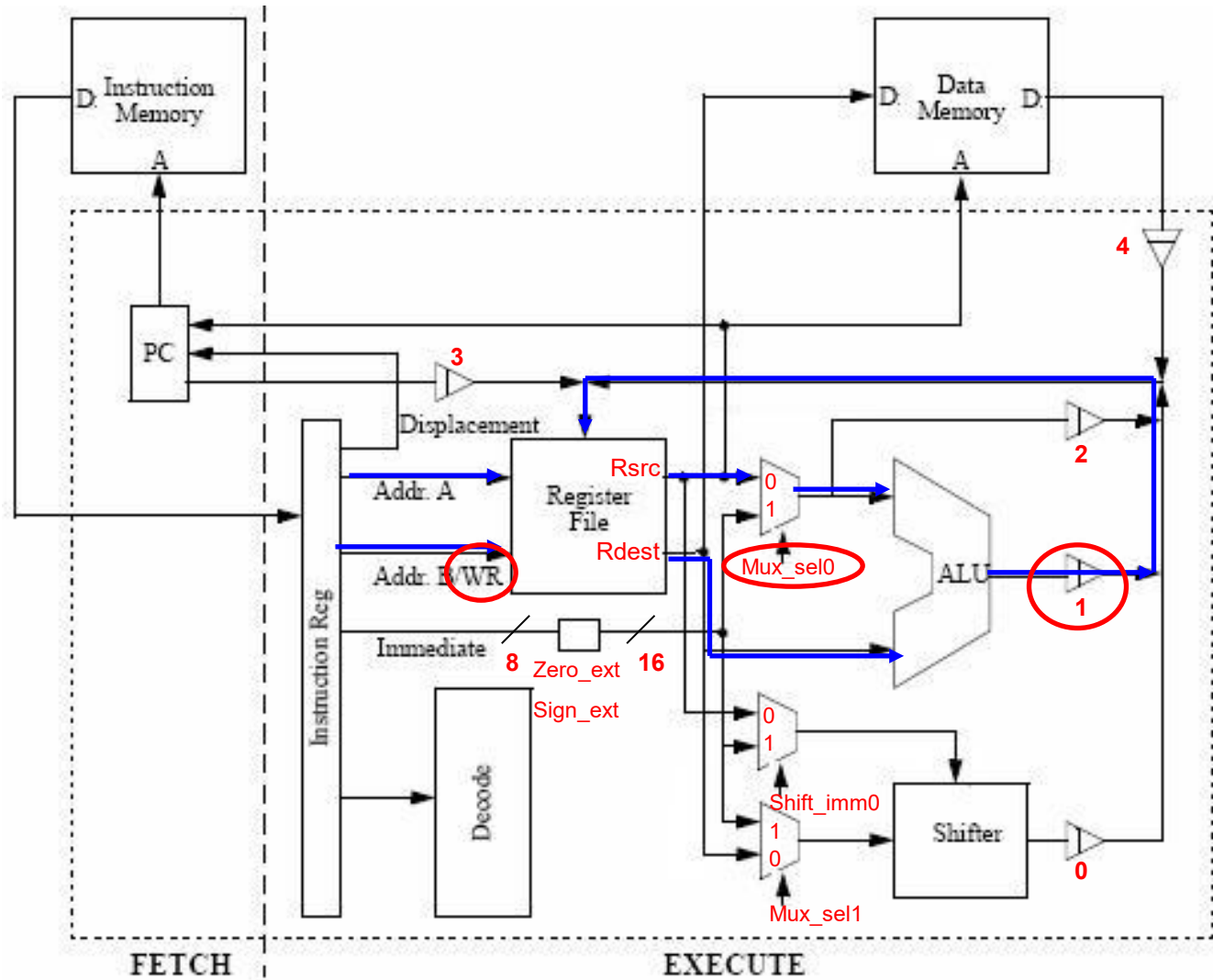
		Msgs																			
	/test_fetch/clock	0																			
	/test_fetch/rst	0																			
+	/test_fetch/inst_i	0316	0316		4ec3		0316		4083		0316										
+	/test_fetch/jmp_tgt	0011011011111111	0011011011111111																		
+	/test_fetch/psr_flags	00001	00001																		
+	/test_fetch/inst_ptr	14084	14079		14080		14081		14079		14080		14079		14080		14081		14082		14083
+	/test_fetch/inst_w	0316	0316				4ec3		0020		4083		0020		0316						
+	/test_fetch/imm_w	16	16				c3		20		83		20		16						
+	/test_fetch/disp_w	16	16				c3		20		83		20		16						
+	/test_fetch/addrA	6	6				3		0		3		0		6						
+	/test_fetch/addrB	3	3				e		0						3						
	/test_fetch/jmp	0																			
	/test_fetch/br	0																			
+	/test_fetch/u_id/Rdest	3	3				e		0						3						
+	/test_fetch/u_id/imm	16	16				c3		20		83		20		16						
+	/test_fetch/u_id/Rsrc	6	6				3		0		3		0		6						

Backup slides for Dataflow & ISA Format

# Dataflow of ALU

OP Code	Rdest	OP Code Ext	Rsrc
---------	-------	-------------	------

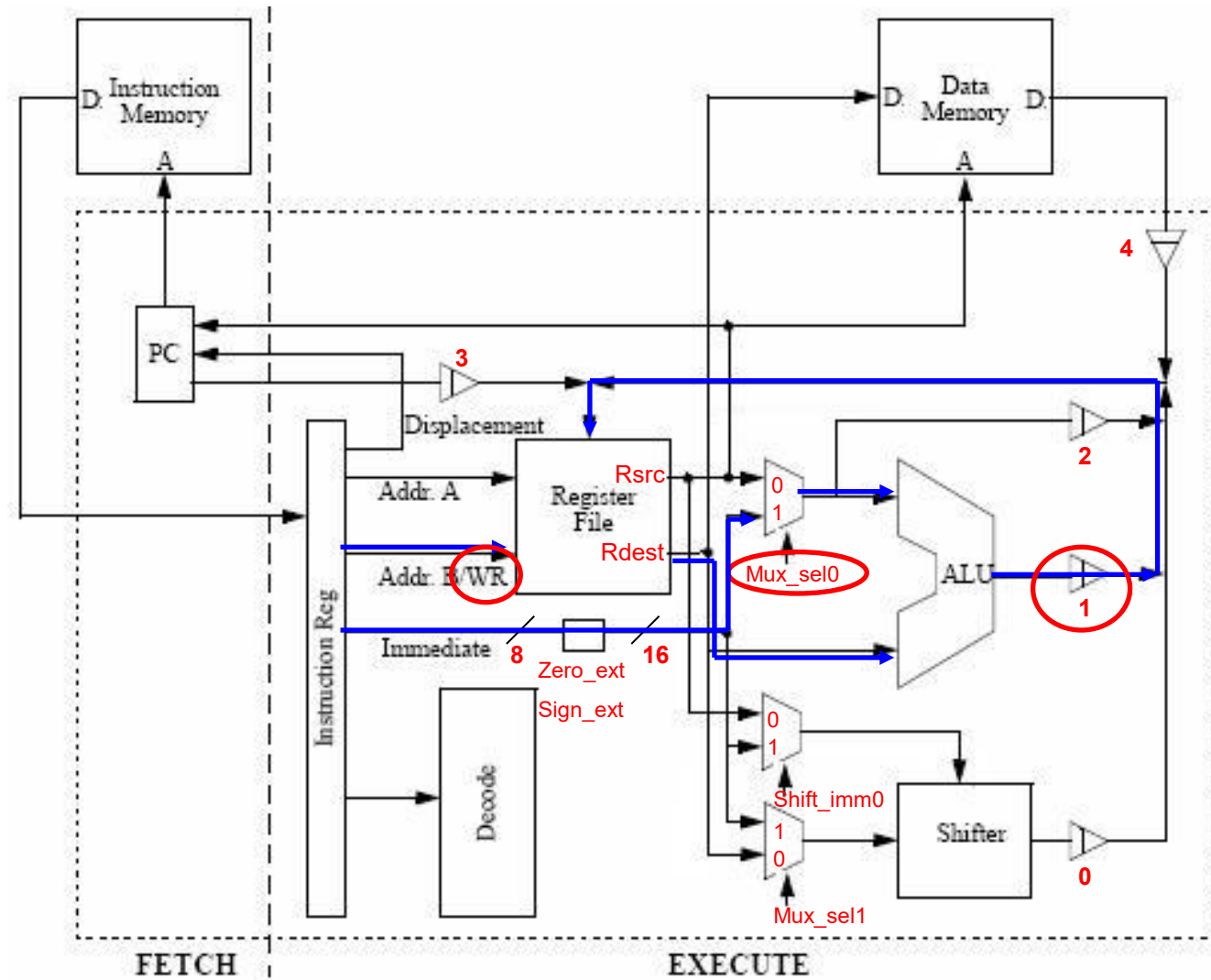
**add r1 r2  $\rightarrow$  r1 = r2 + r1**  
**sub r3 r4  $\rightarrow$  r3 = r4 - r3**



# Dataflow of $ALU_i$ (ALU-immediate)

OP Code	Rdest	Immediate
---------	-------	-----------

```
cmpi imm r1
ori  imm r3
```

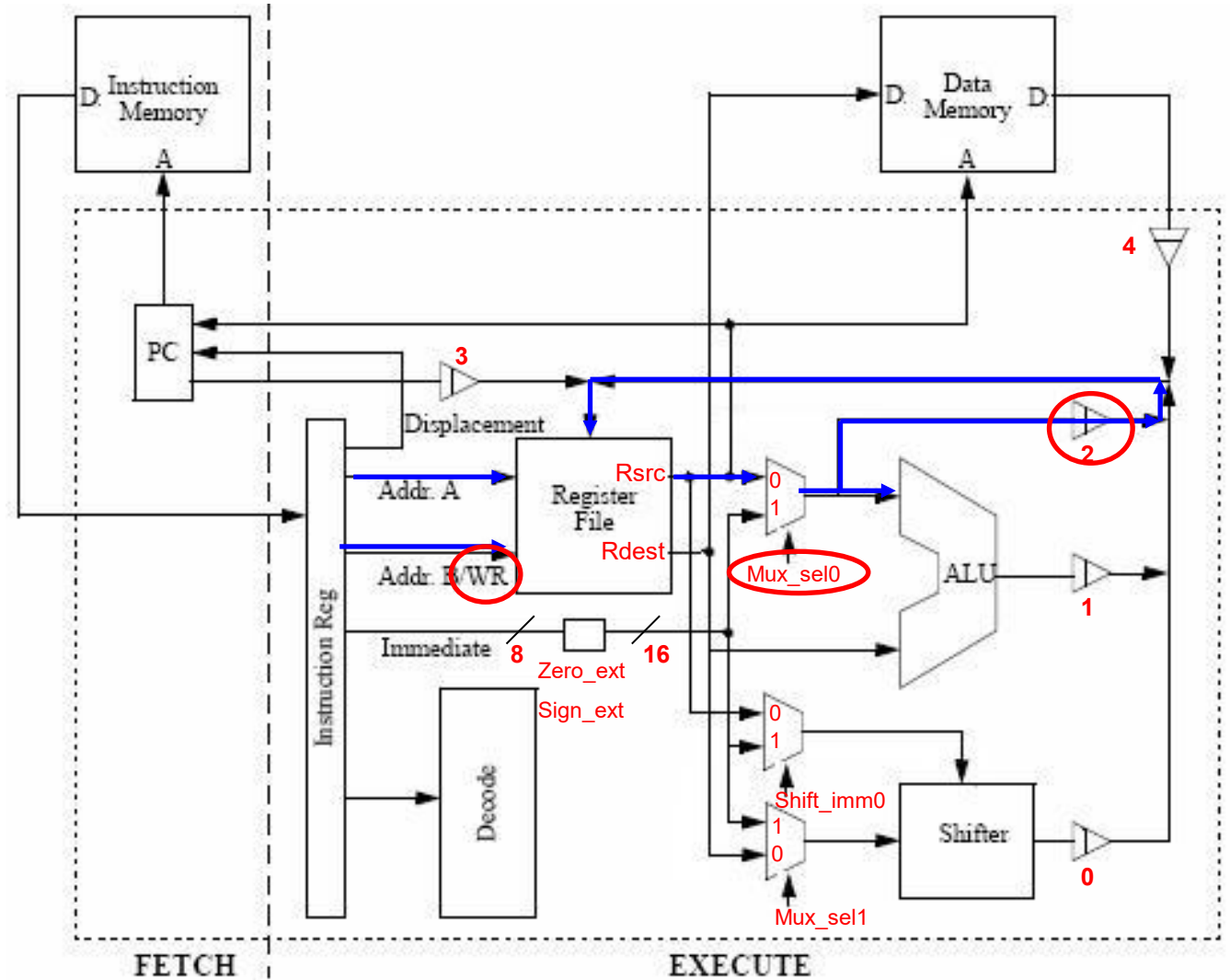


# MOV (mov) Instruction

- Used to move a register value from another

OP Code	Rdest	OP Code Ext	Rsrc
---------	-------	-------------	------

**mov r1 r2  $\rightarrow$  r1 = r2**

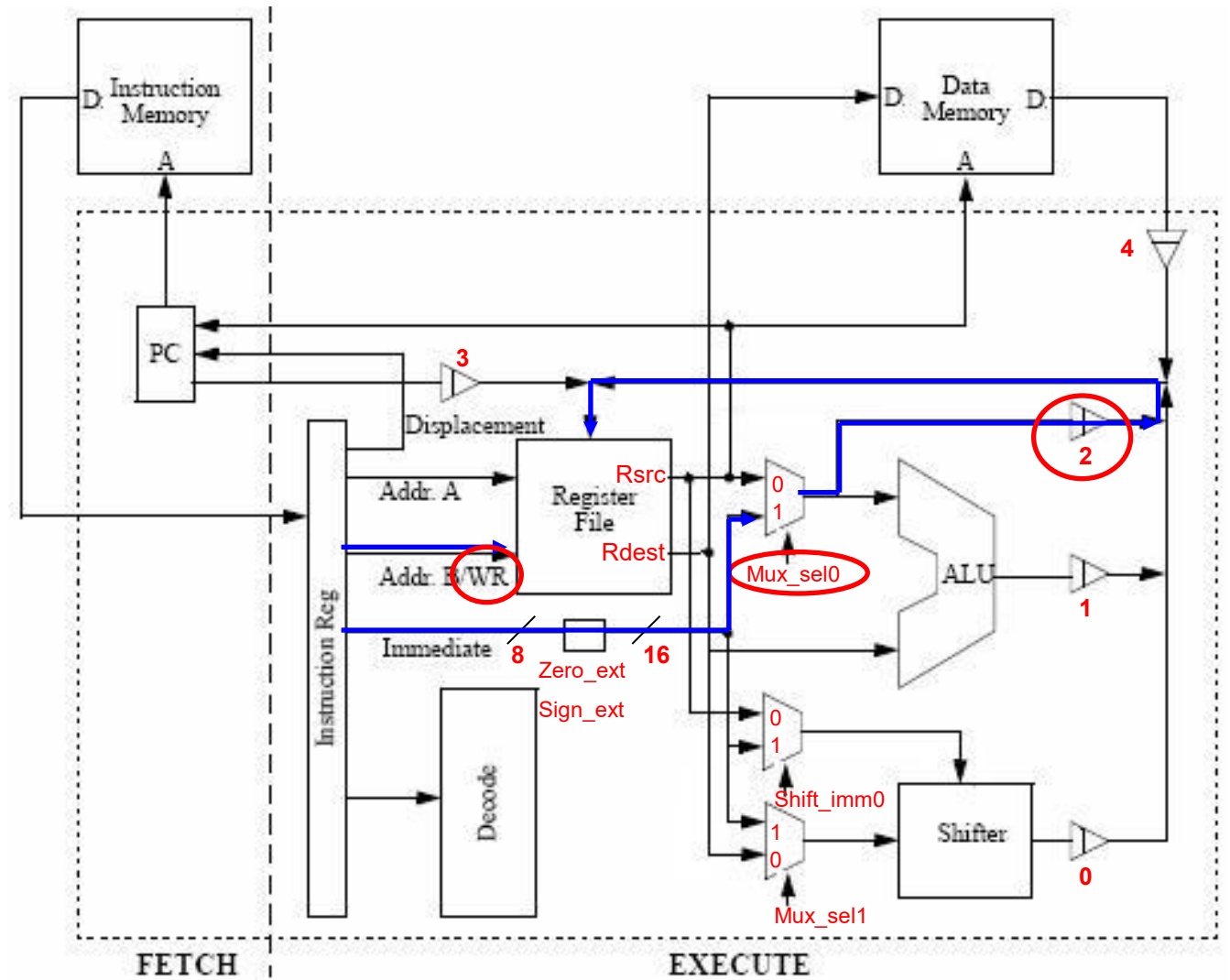


# MOVI (move-immediate)

- Move an immediate value to the destination register

OP Code	Rdest	Immediate
---------	-------	-----------

**movi imm r1**



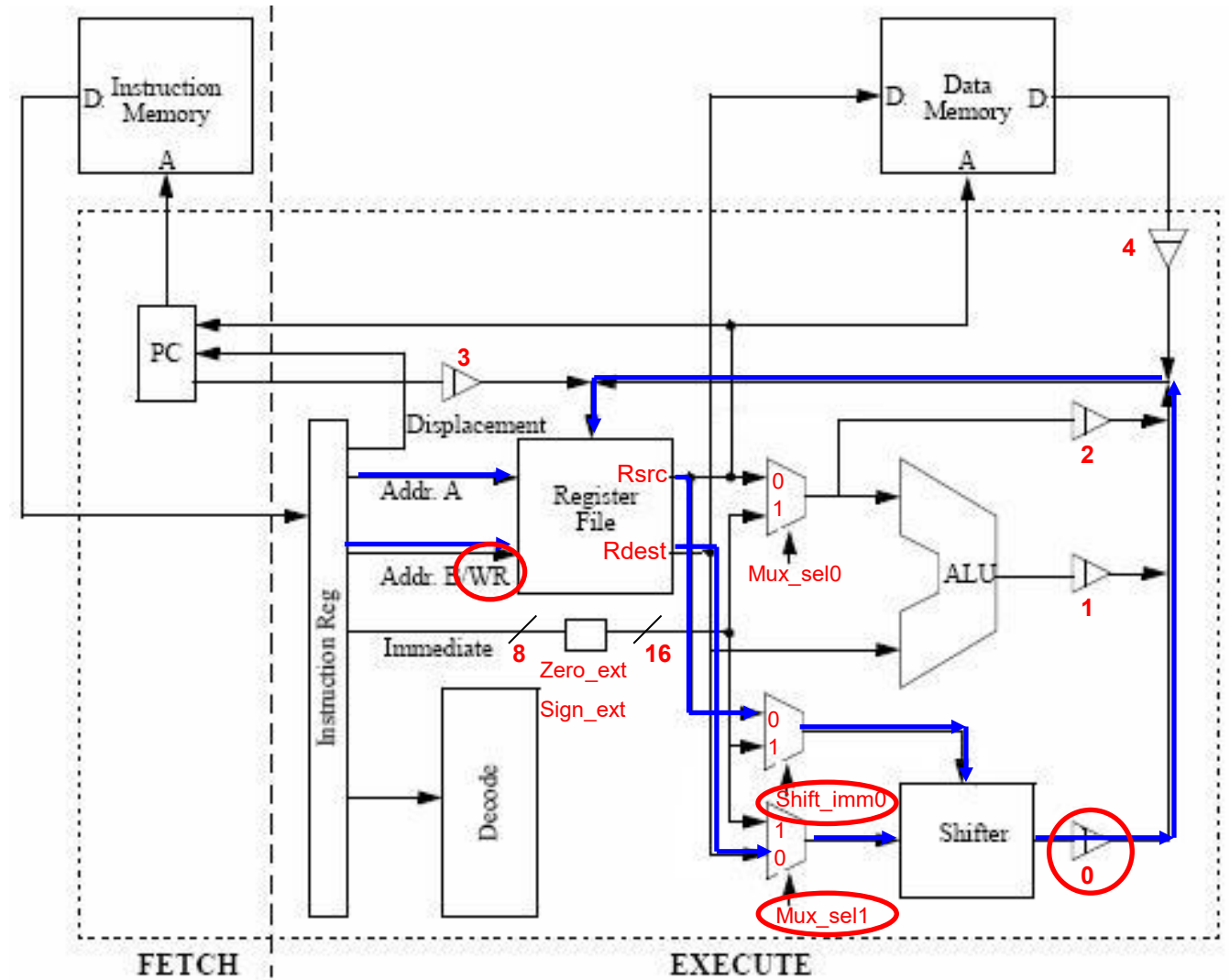


# Dataflow of LSH

- Logical shift
- Rlamount takes Rsrc as its value
- Result is stored in Rdest

OP Code	Rdest	OP Code Ext	Rsrc
---------	-------	-------------	------

**lsh r1 r2**

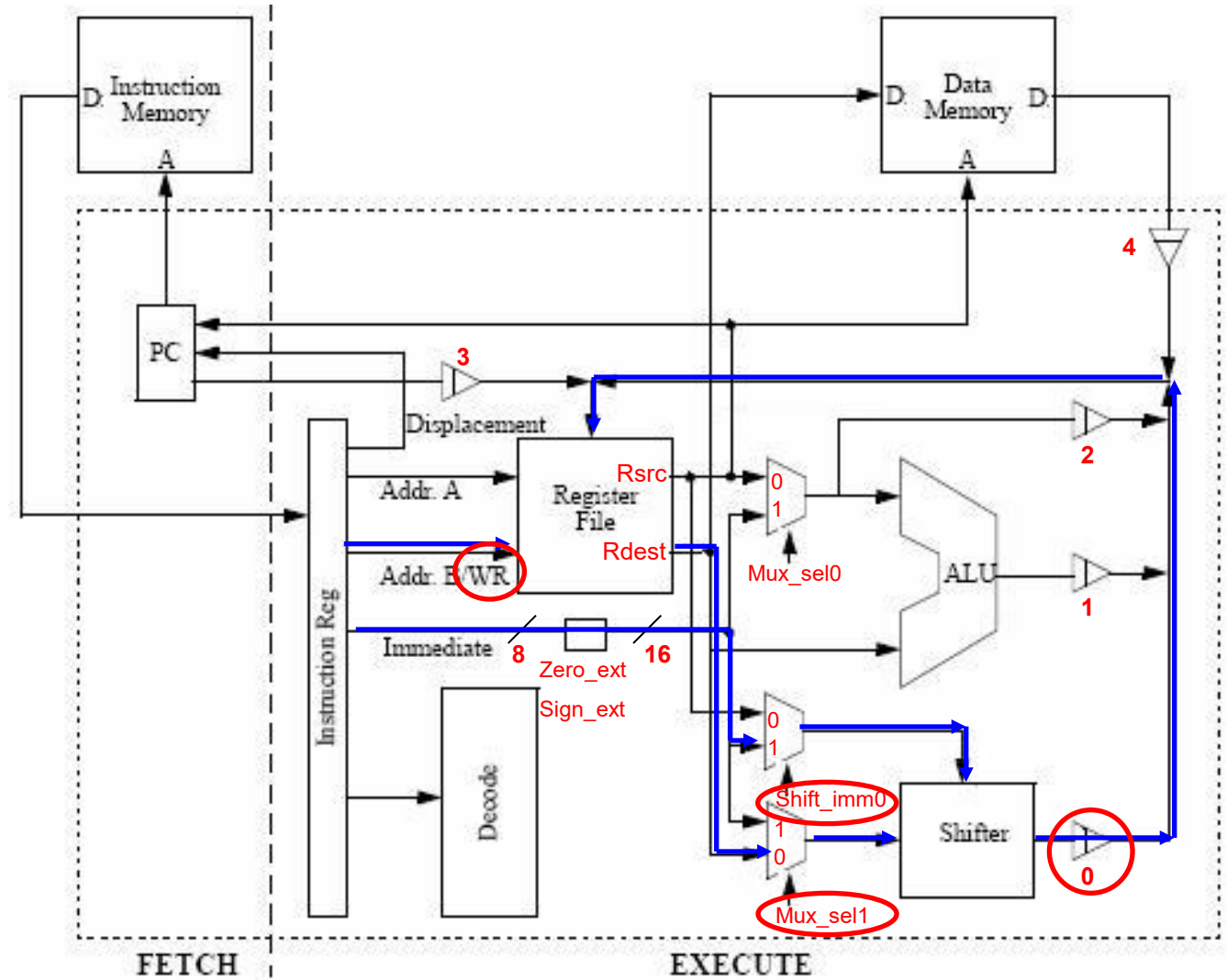


# Dataflow of LSHi

- Shifter takes Rdest as input
- Rlamount takes MSB of OpcodeEx as sign bit
- Logical shift with the amount specified by immediate field

OP Code	Rdest	000	RLamount
---------	-------	-----	----------

## Ishi RLamount r1

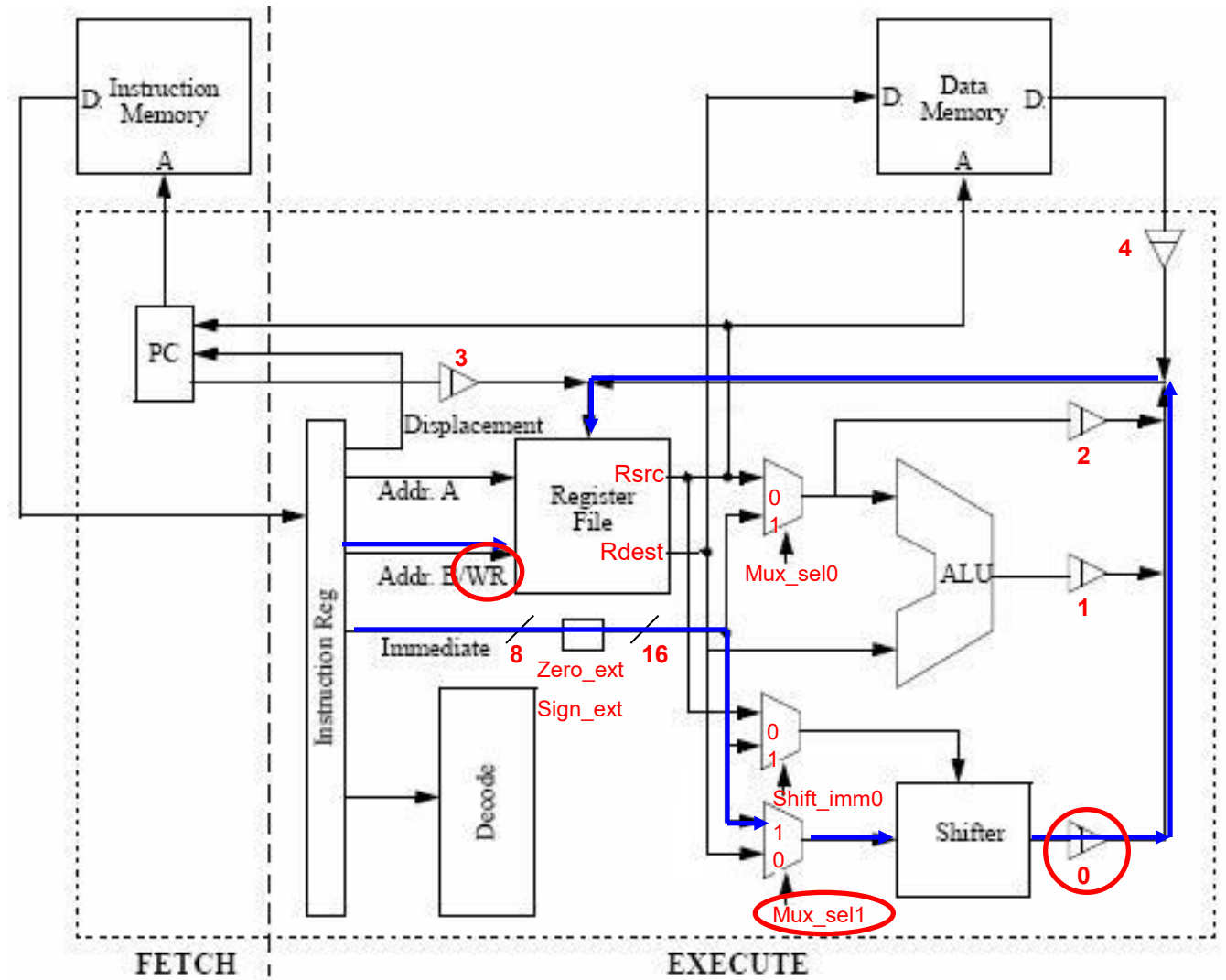


# Dataflow of LUI

- Shifter takes Immediate value as input
- Always shift 8 bit towards left
- Store result in Rdest

OP Code	Rdest	Immediate
---------	-------	-----------

**lui imm r1**

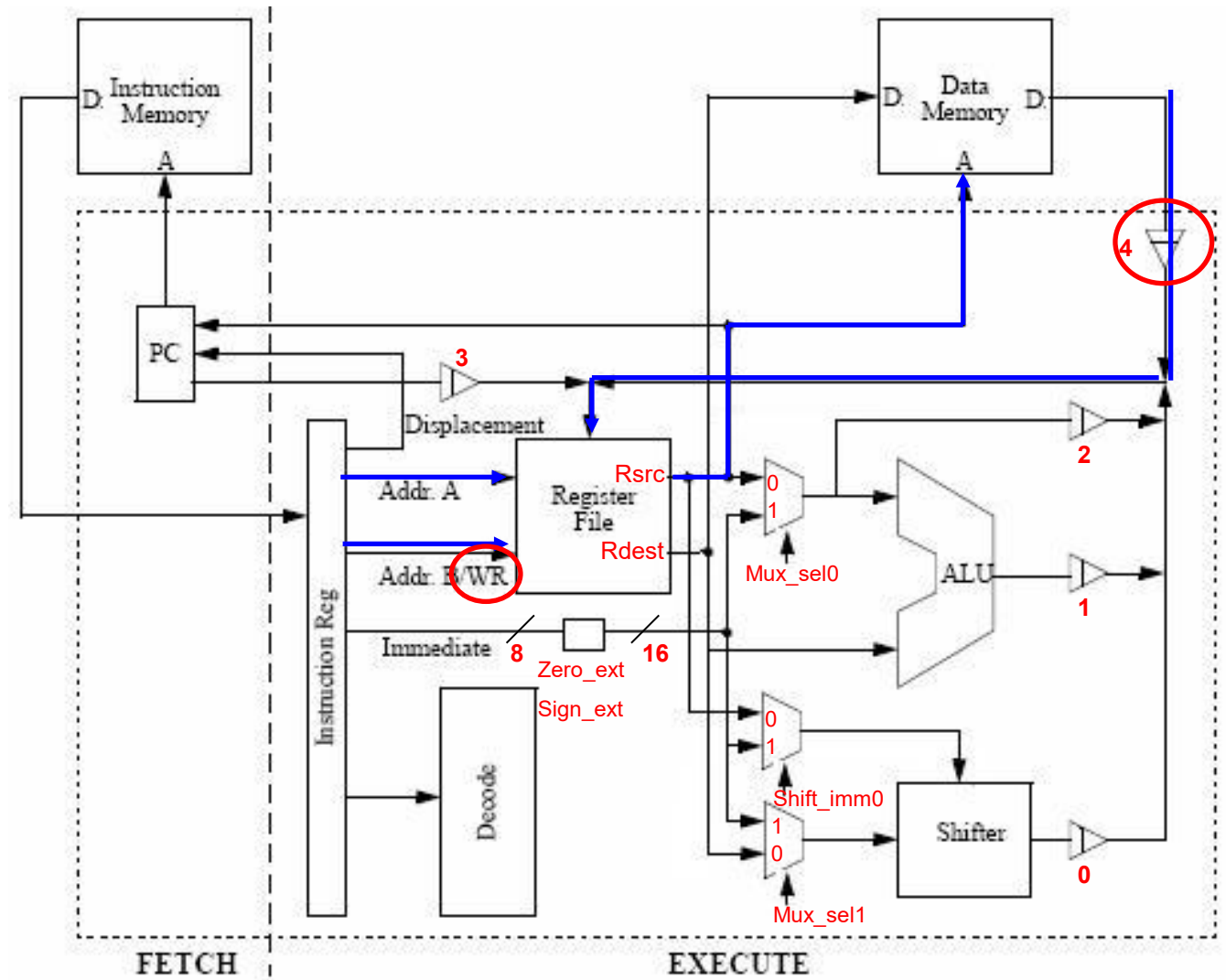


# Dataflow of LOAD

- Load the value from data memory
- Memory address is specified by Raddr
- Result is stored in Rdest

OP Code	Rdest	OP Code Ext	Raddr
---------	-------	-------------	-------

**load r1 r2**

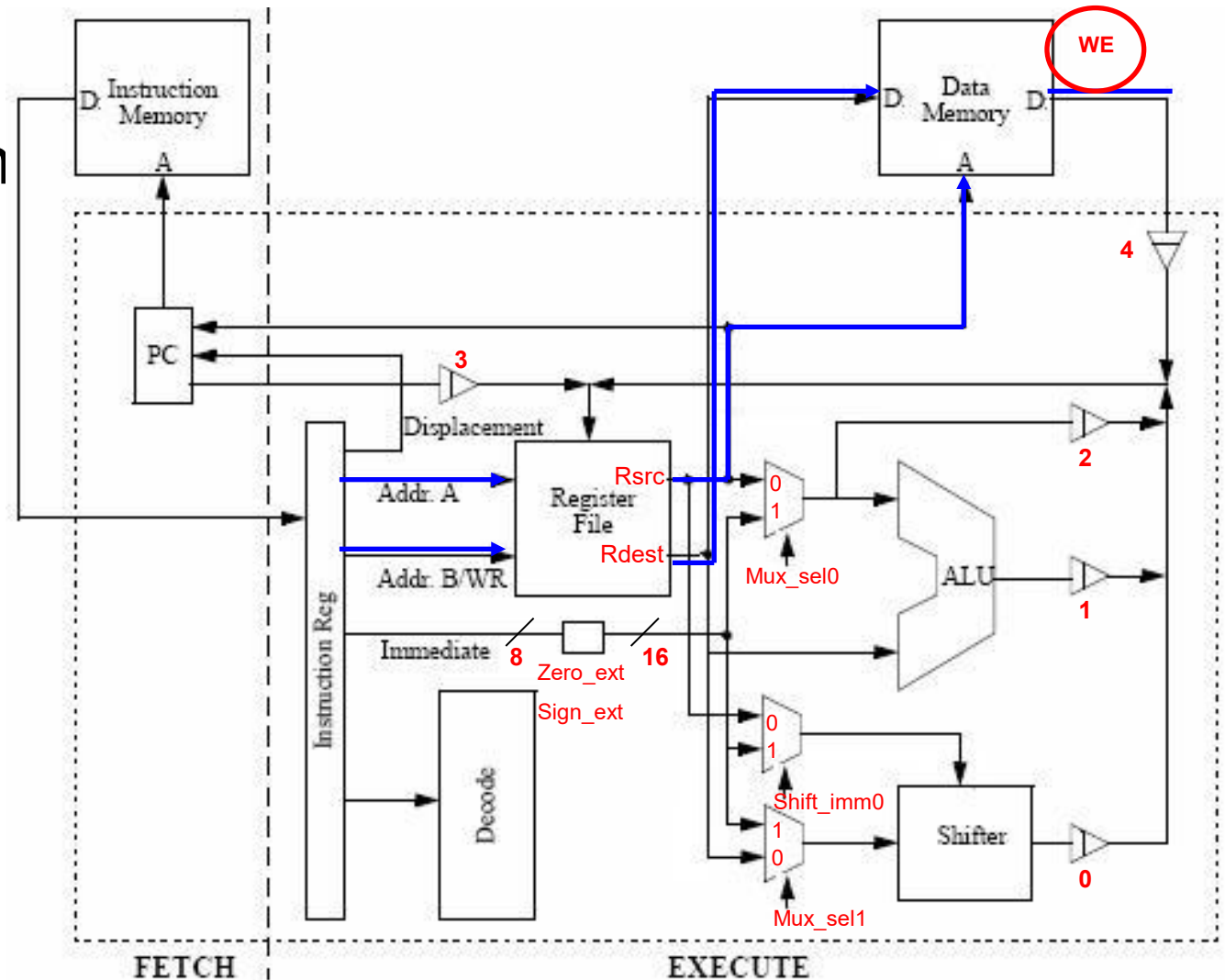


# Dataflow of STORE

- Store the value residing in Rdest into data memory
- Memory address is specified by Raddr

OP Code	Rdest	OP Code Ext	Raddr
---------	-------	-------------	-------

## store r1 r2

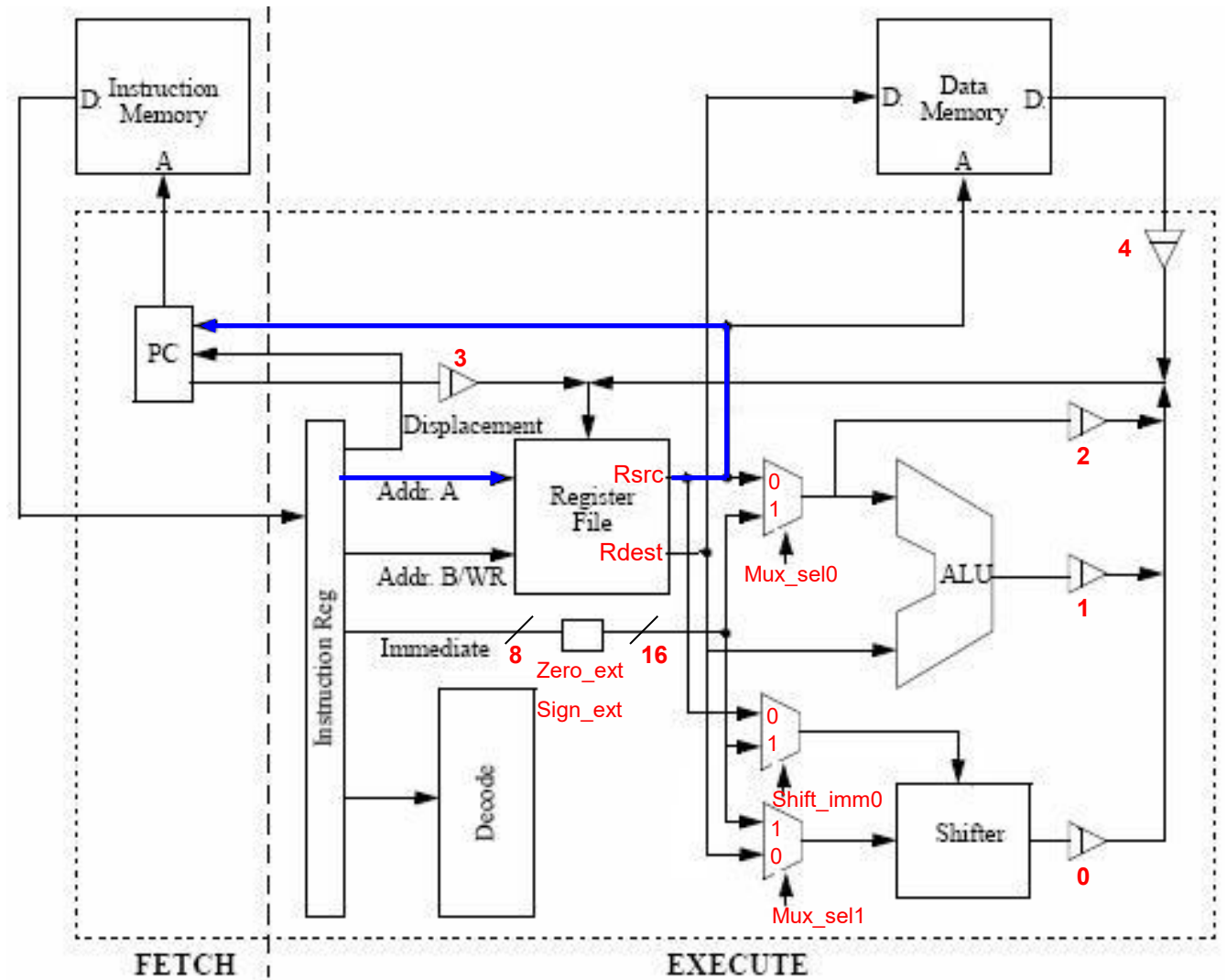


# Dataflow of JCond

- Check the condition
- Jump to Rtarget if condition is met

Or Rdest			
OP Code	cond	OP Code Ext	Rtarget

```
jeq r1  
jne r1
```

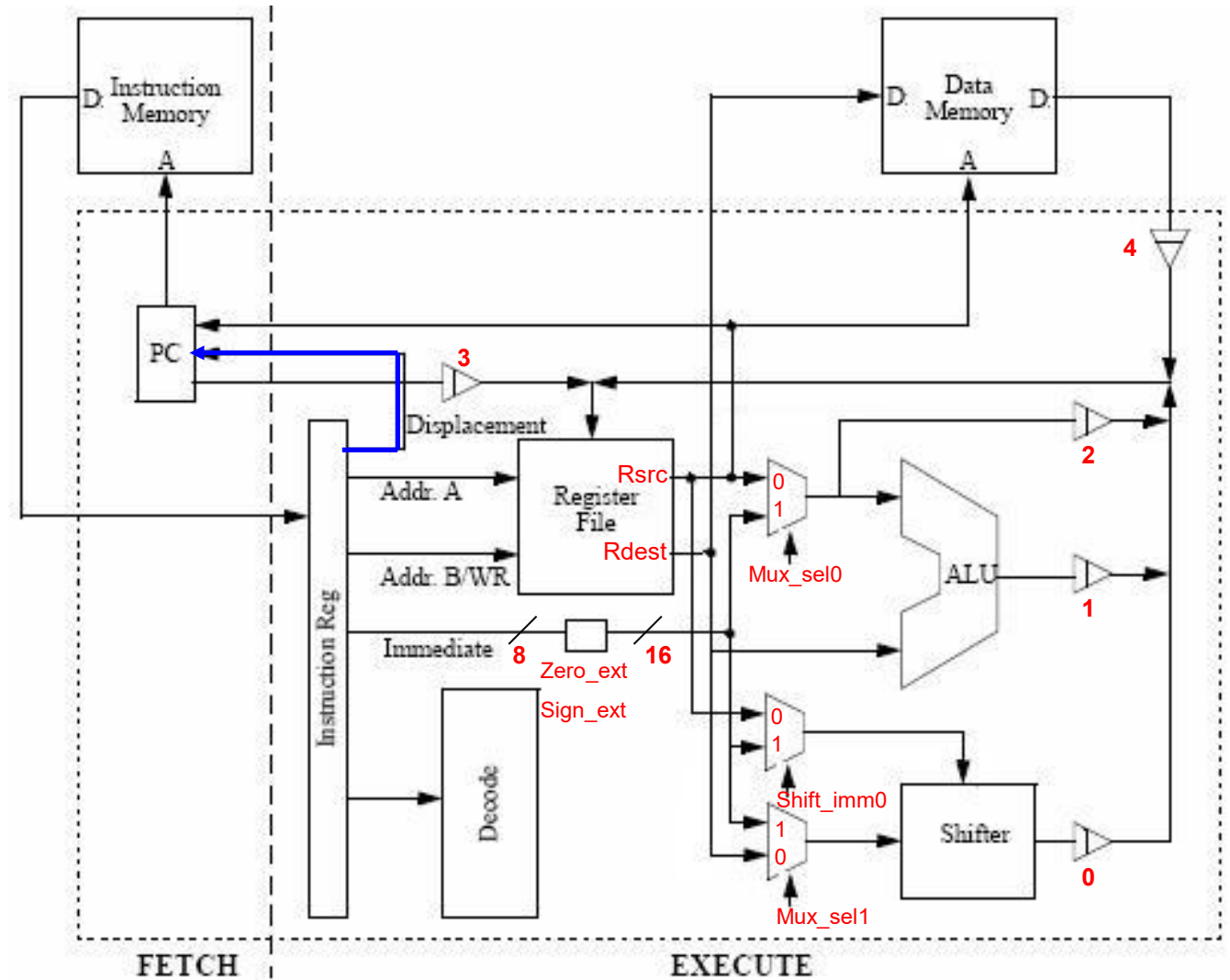


# Dataflow of BCond

- Check the condition
- Branch to target if condition is met
- Unlike Jcond, target is calculated using displacement

OP Code	cond	Displacement
---------	------	--------------

**bne imm**  
**ble imm**





# Dataflow JAL

- Store PC+1 to Reg file specified by Rlink
- Jump to Rtarget

OP Code	Rlink	OP Code Ext	Rtarget
---------	-------	-------------	---------

**jal r1 r2**

