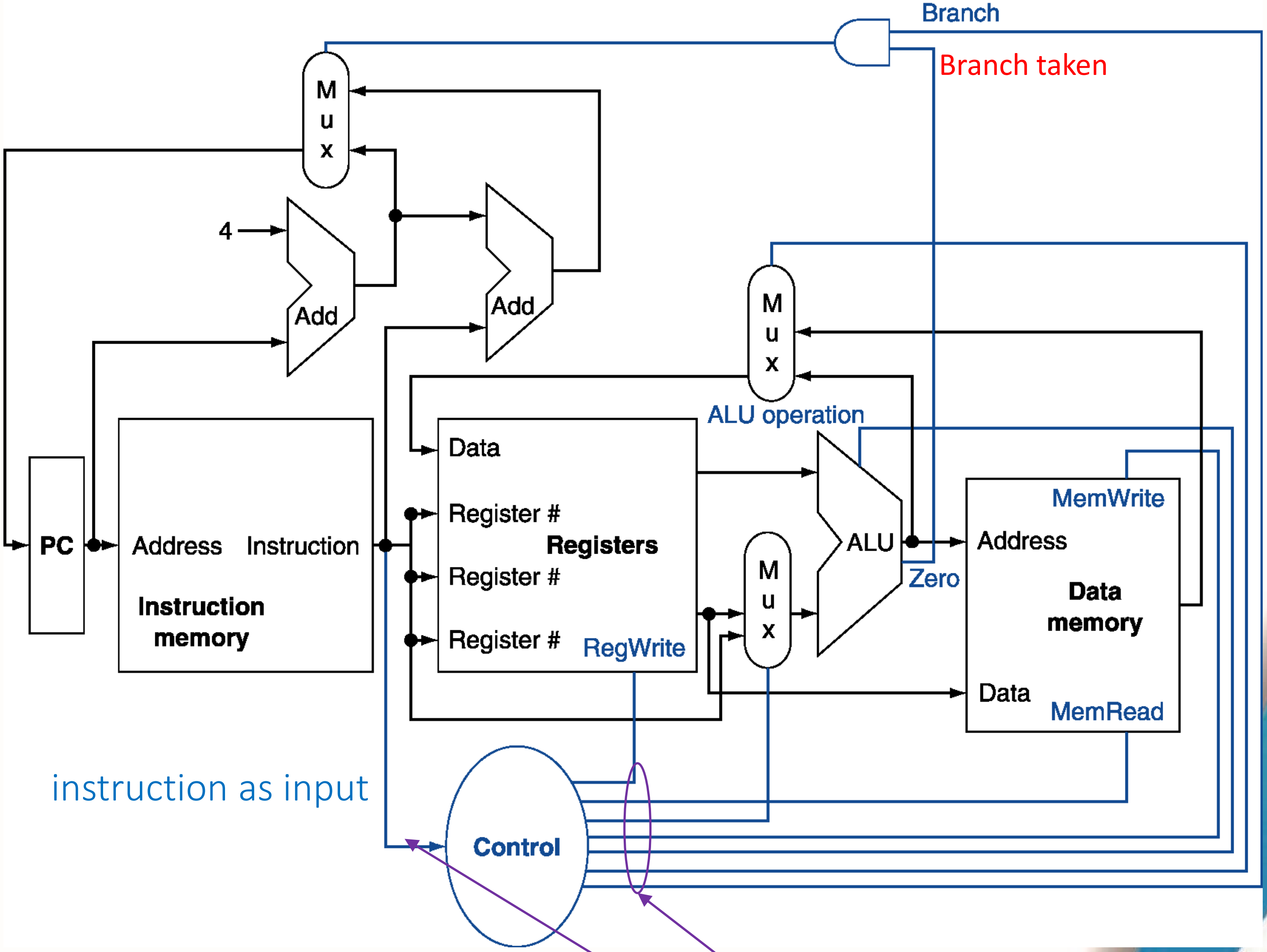
Computer Architecture and Organization

INSTRUCTOR: YAN-TSUNG PENG

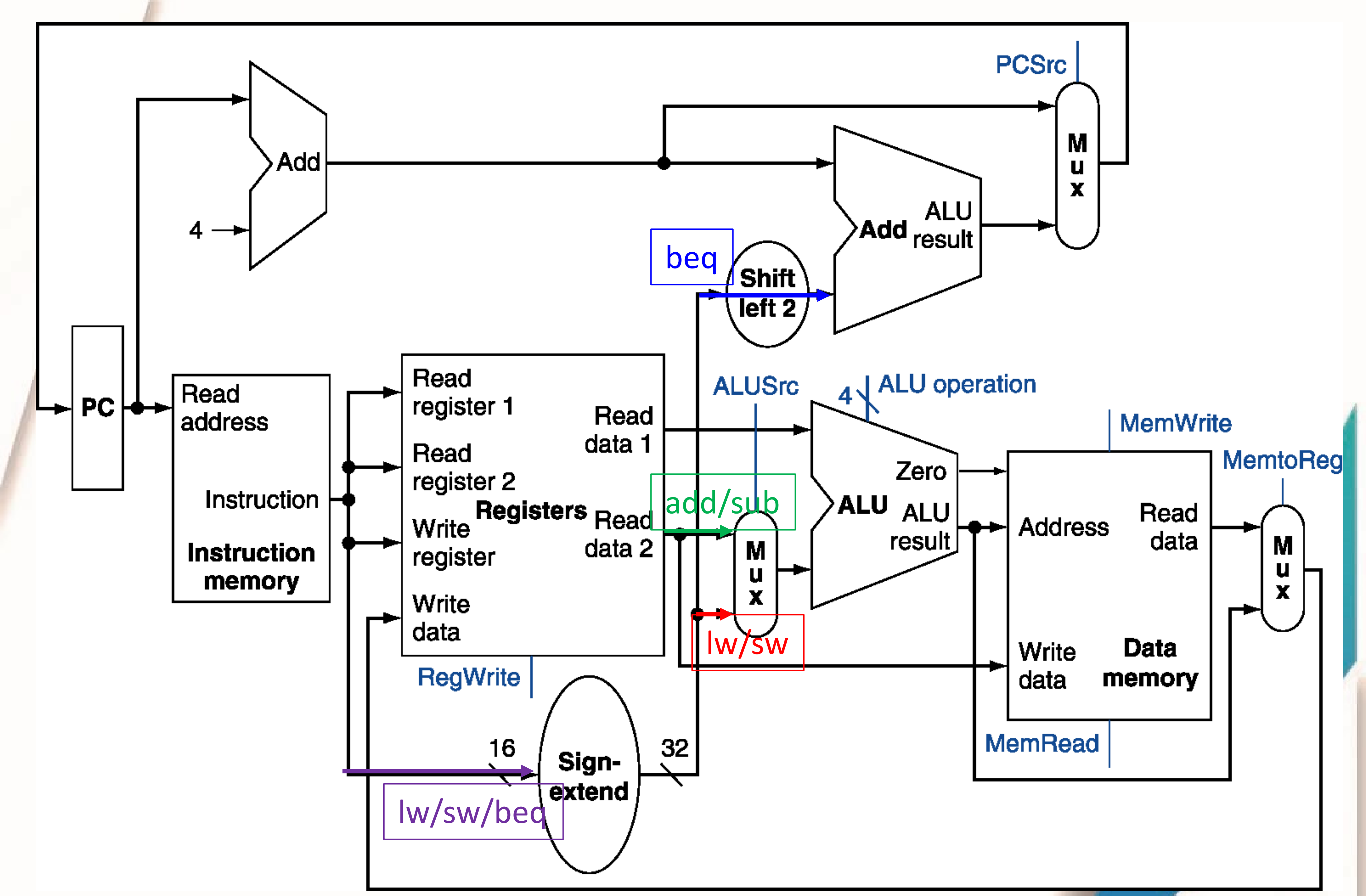
DEPT. OF COMPUTER SCIENCE, NCCU

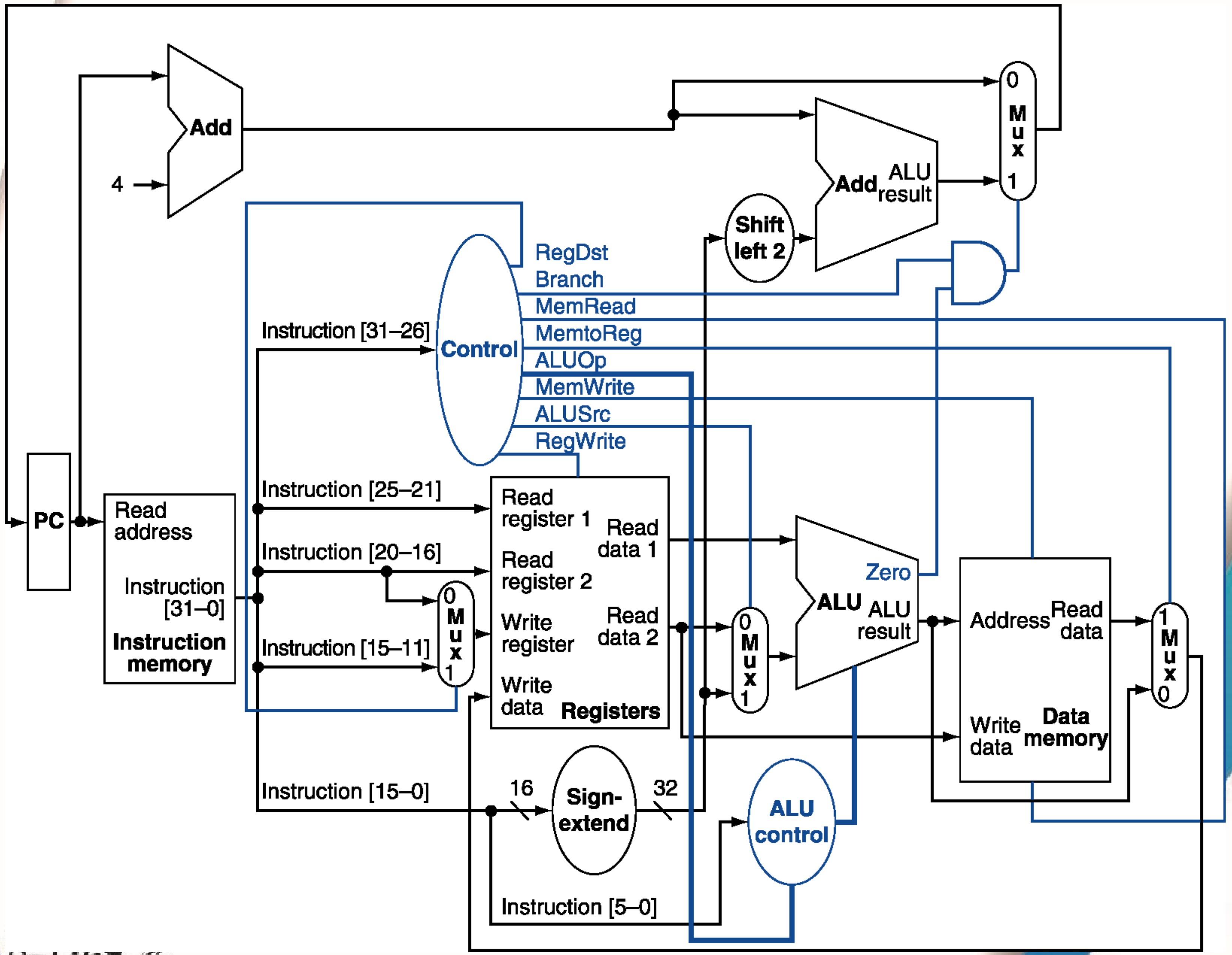
Control



instruction as input

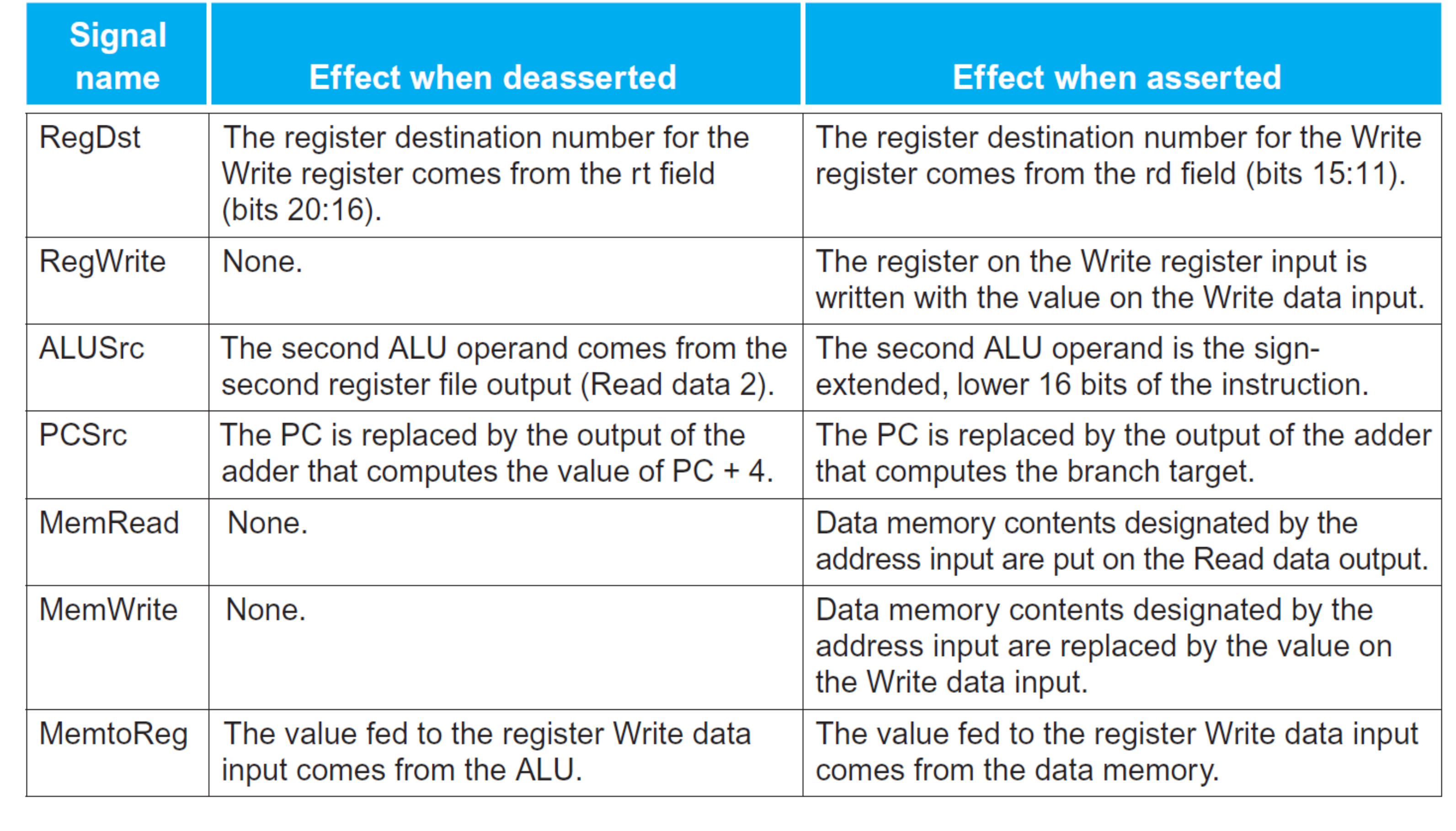
Control unit outputs control signals for all the MUXs depending upon the input instruction


PCSrc
Add
4
M
u
x
Add
ALU
result
beq
Shift
left 2
PC
Read
address
Instruction
Instruction
memory
Read
register 1
Read
register 2
Write
register
Write
data
Registers
Read
data 1
Read
data 2
ALUSrc
4
ALU operation
add/sub
M
u
x
lw/sw
Zero
ALU
ALU
result
MemWrite
MemtoReg
Address
Read
data
Write
data
Data
memory
M
u
x
MemRead
RegWrite
16
Sign-
extend
32
lw/sw/beq


Add
4
Add
ALU result
Shift left 2
0 M u x 1
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
0 M u x 1
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
0 M u x 1
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
1 M u x 0
Instruction [15–0]
16
Sign-extend
32
ALU control
Instruction [5–0]

Effect of Control Signals

7 control signals



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.

Control Signals derived from Opcode

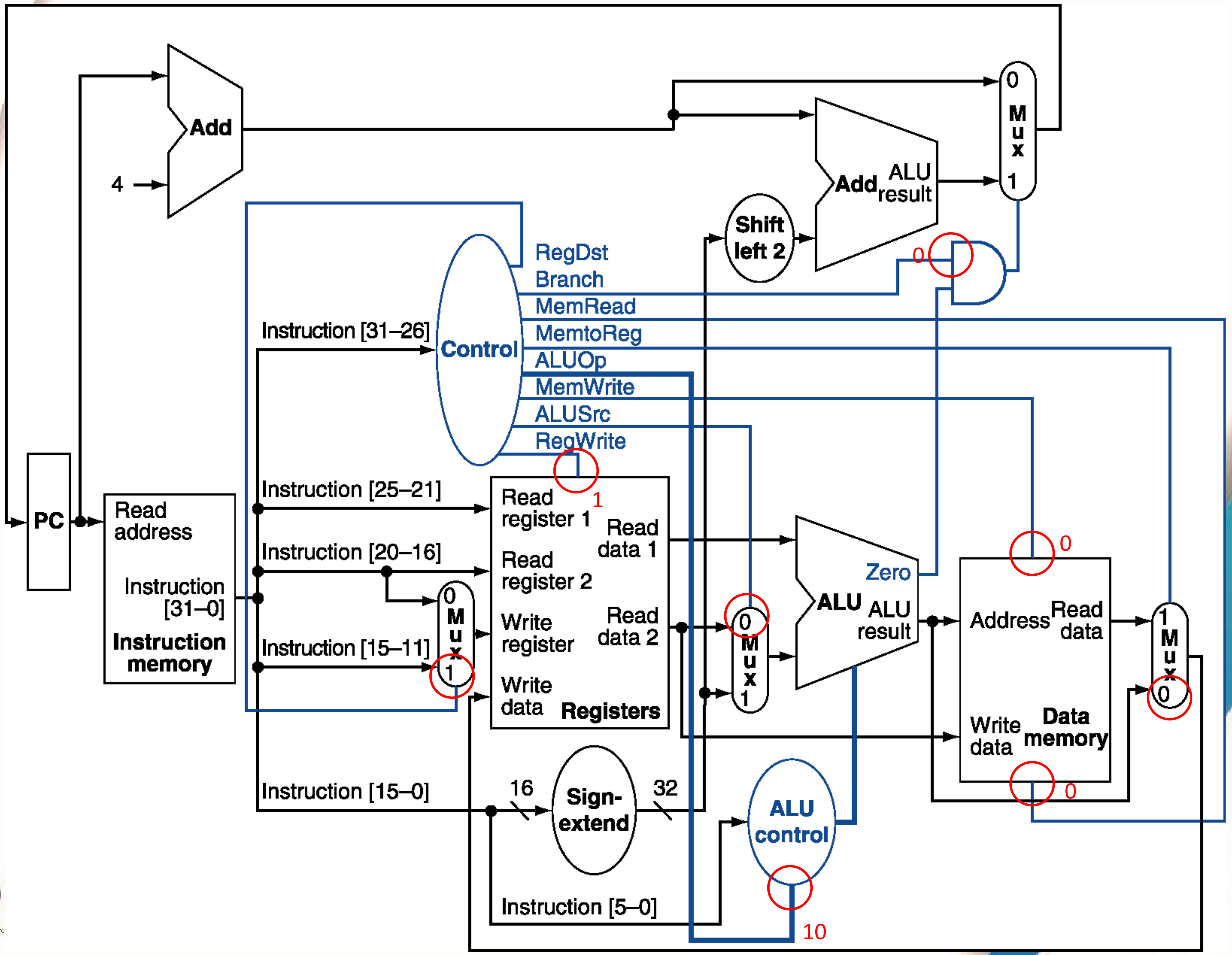
IR←Mem[PC] opcode = IR[31:26]

rs←IR[25:21] rt←IR[20:16] rd←IR[15:11] Imm16←IR[15:0] Addr26←IR[25:0]

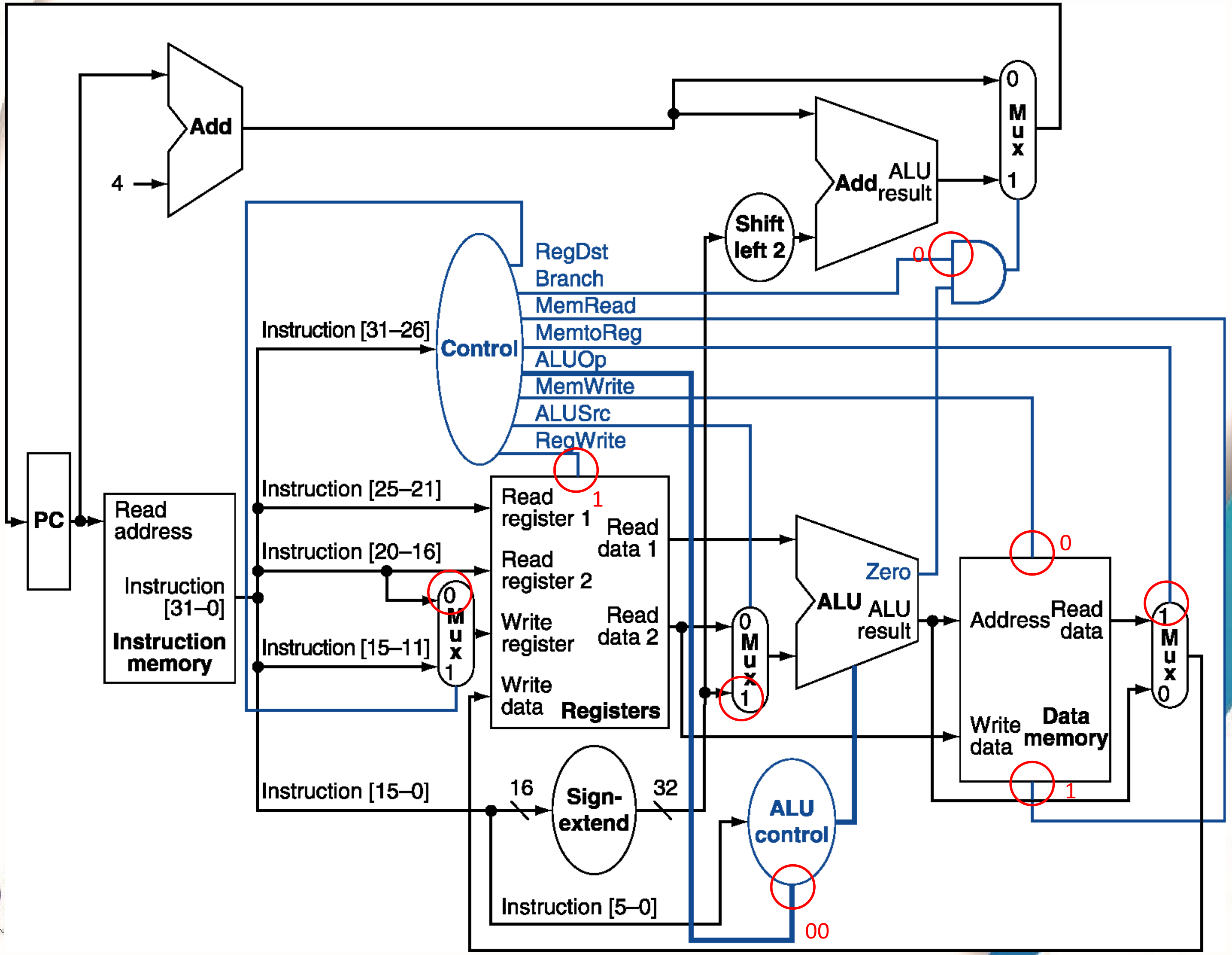
In	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

opcode	ALUOp
lw	00
sw	00
beq	01
R-type	10

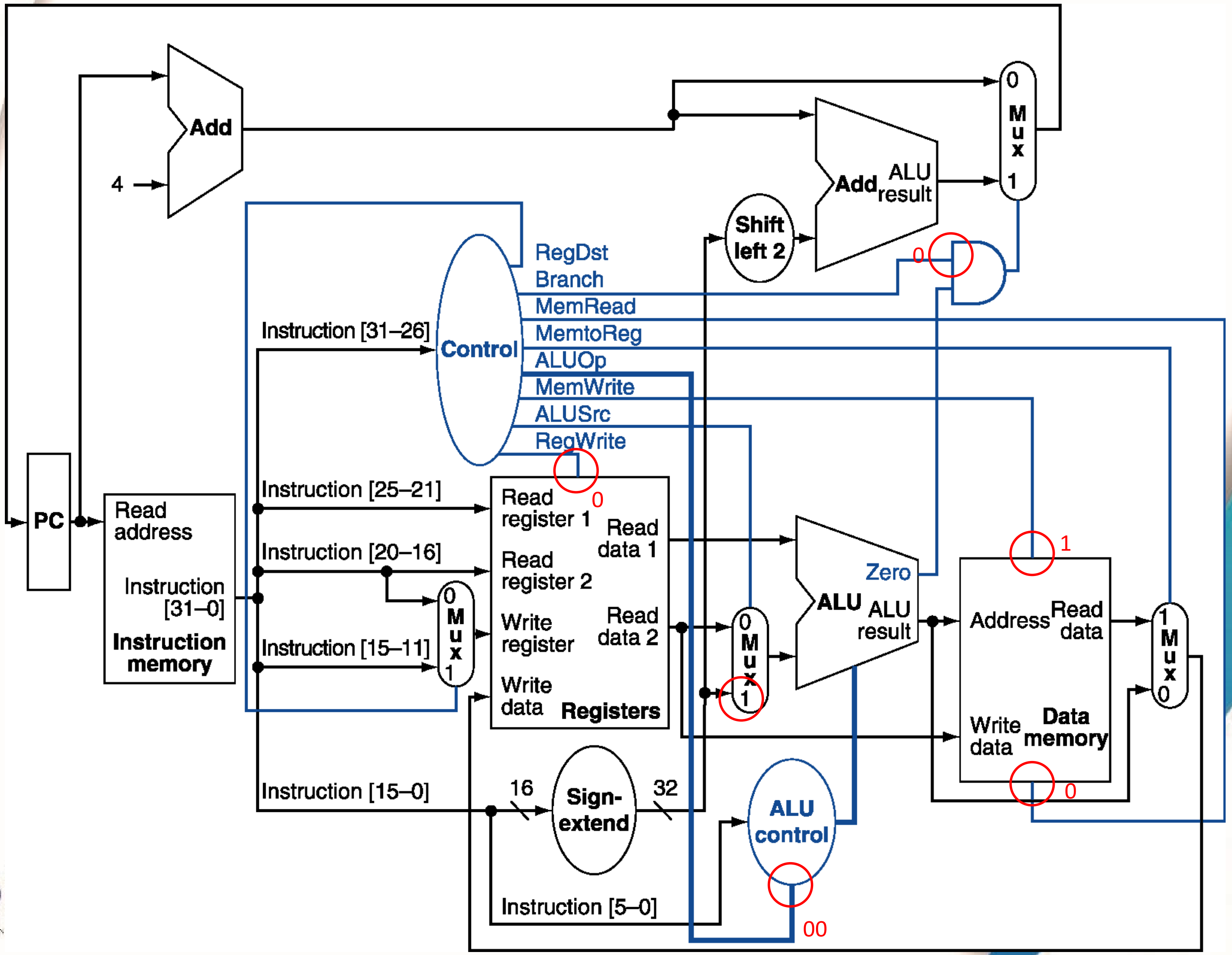
In	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0



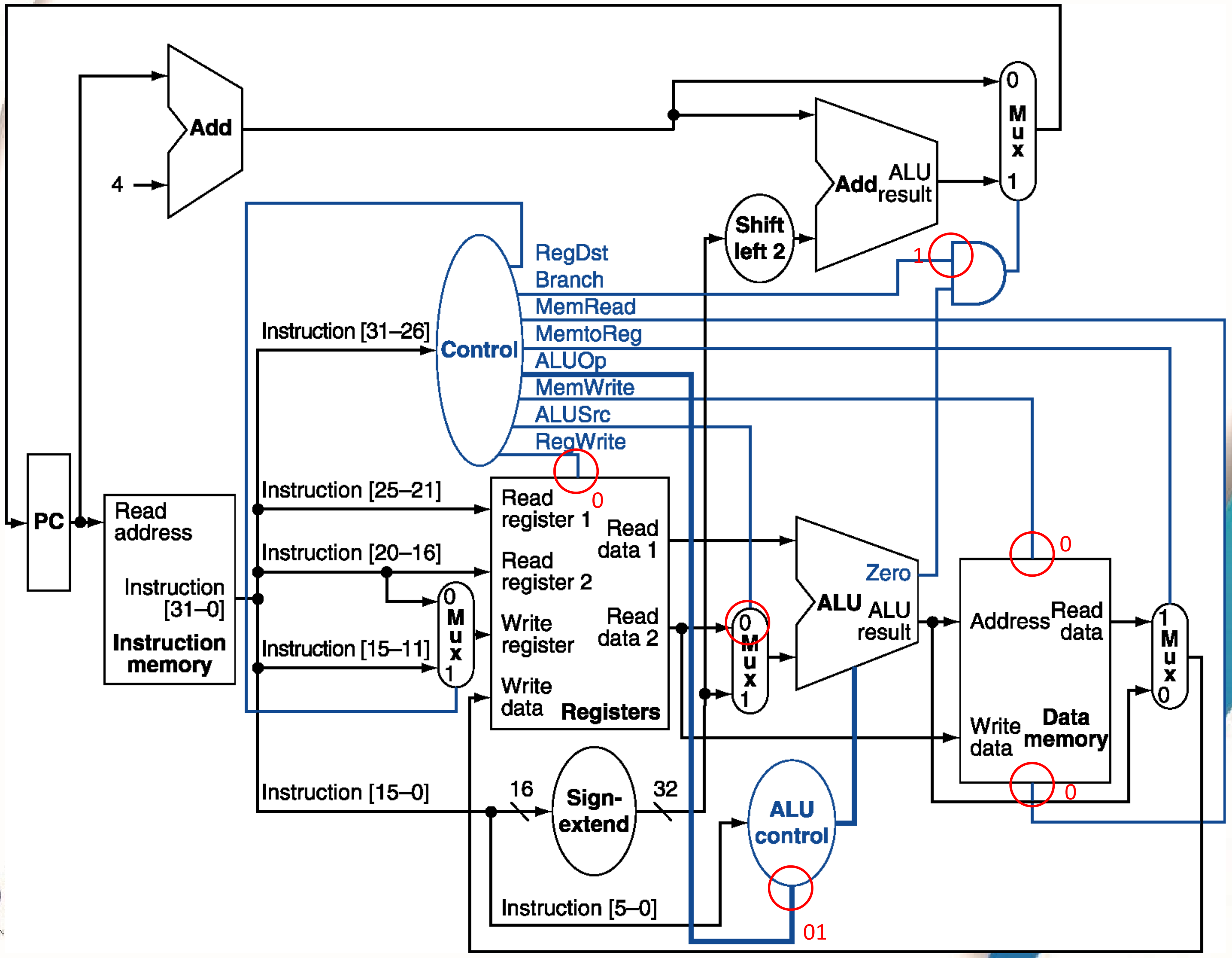
In	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
lw	0	1	1	1	1	0	0	0	0



In	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
sw	X	1	X	0	0	1	0	0	0



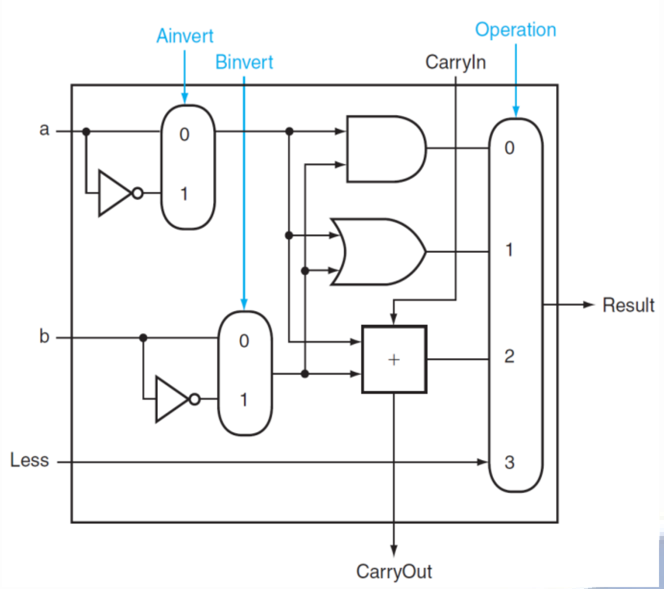
In	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
beq	X	0	X	0	0	0	1	0	1



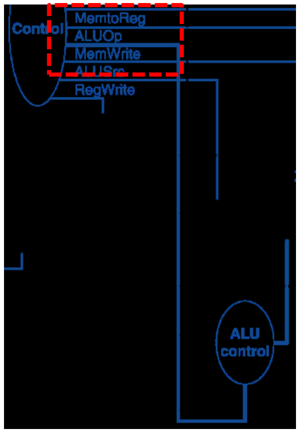
Simple Implementation Scheme for ALU

- It covers lw, sw, beq, add, sub, and, or, slt
- ALU Control

	ALU control	Function
	0000	AND
	0001	OR
add,lw, sw	0010	add
sub,beq	0110	subtract
	0111	set-on-less-than
	1100	NOR



ALU Control



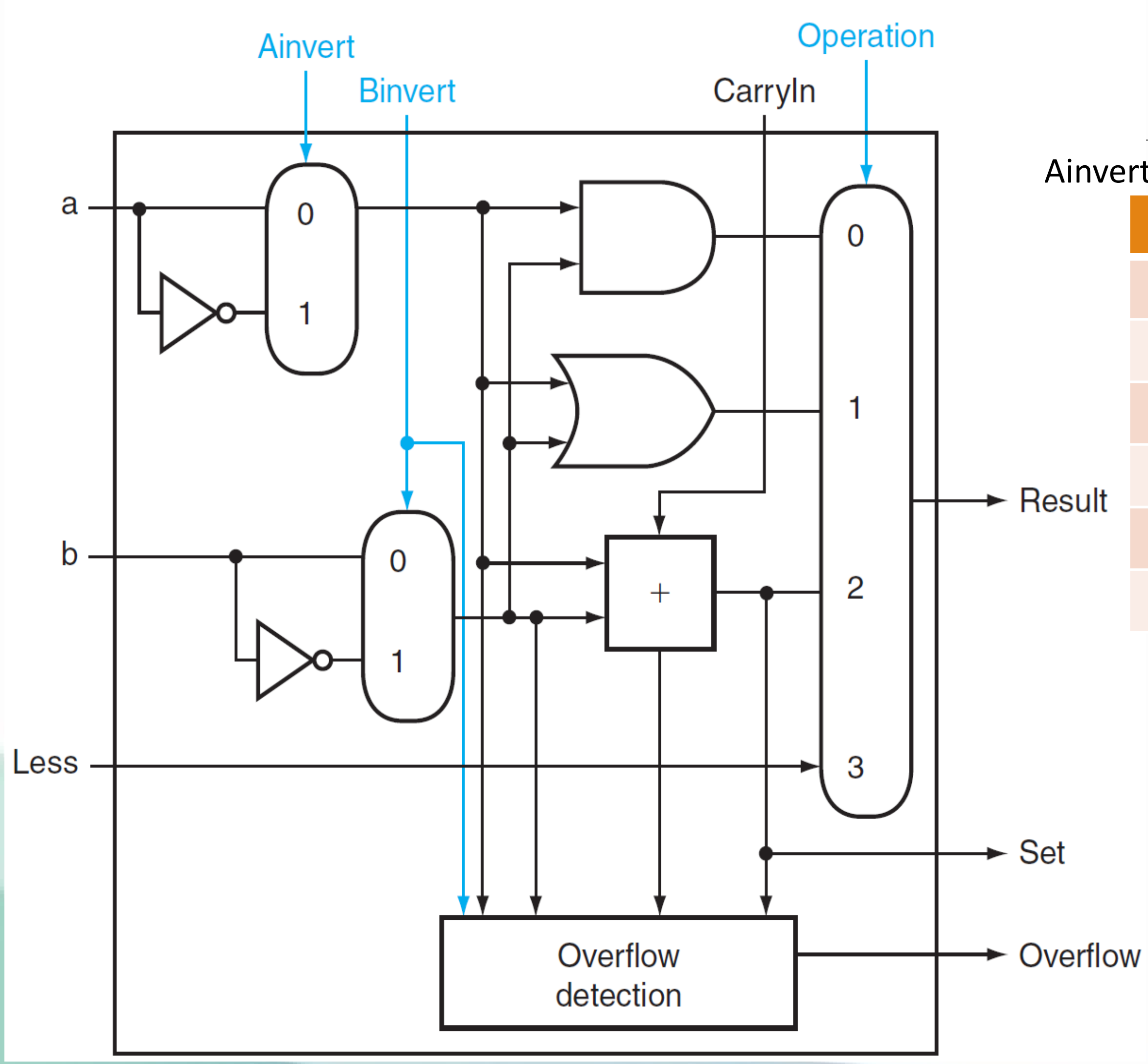
- Assume 2-bit ALUOp derived from opcode
  - I-type
    - (00) add for lw/sw
    - (01) sub for beq
  - R-type
    - (10) also considering `func` field

- Combinational decoder derives ALU control
- Using multiple levels of control can reduce the size of the main control unit and makes the control faster

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

ALU

1-bit ALU for the most significant bit



Ctrl Lines

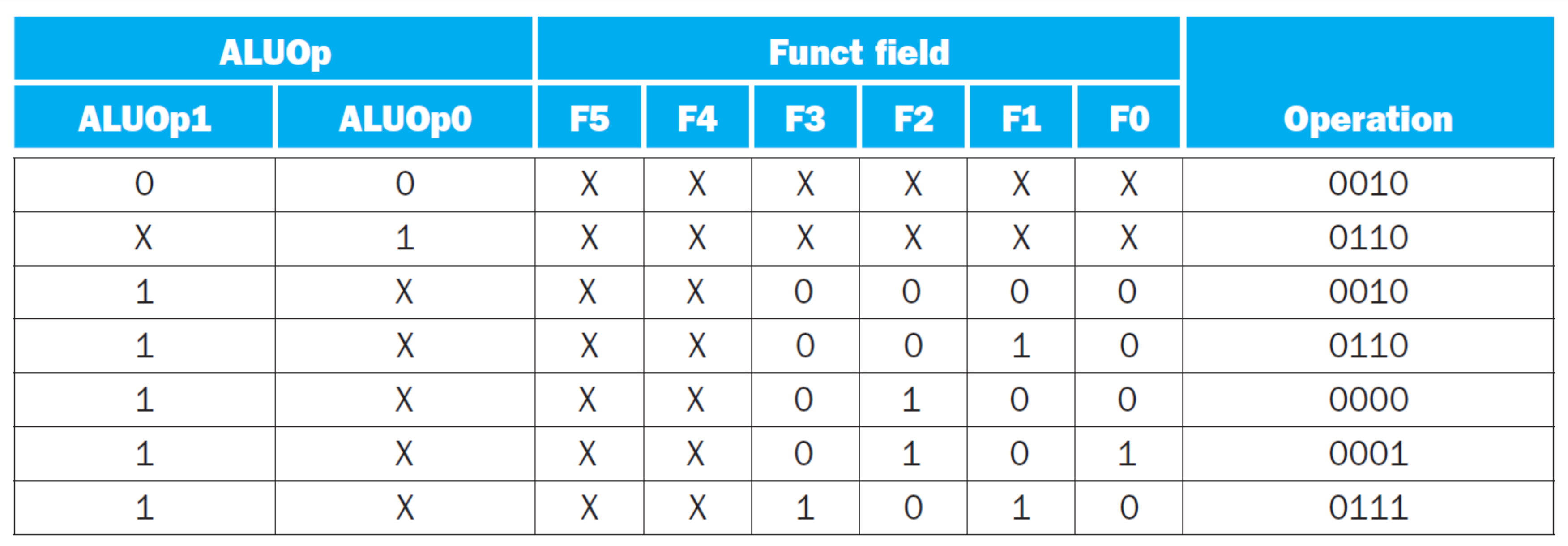
Ainvert[0]:Binvert[0]:Operation[1:0]

Ctrl Lines	Func
0000	AND
0001	OR
0010	Add
0110	Sub
0111	Slt
1100	NOR

FIGURE B.5.10 from Computer Org. and Design, 5th edition

Main Control Unit

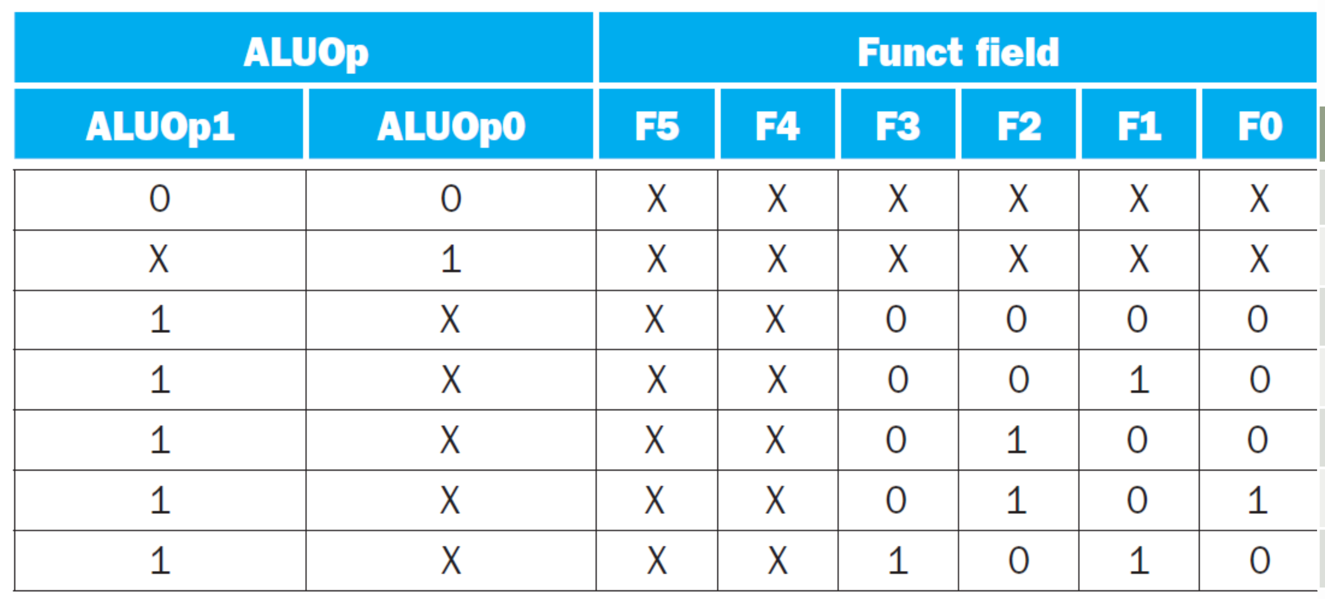
- Truth table for the 4 ALU control bits => Simplification of the 8-variable K-map => Boolean expression => circuit



ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	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

[redacted]

Truth Table for ALU Control



ALUOp		Funct field						ALU Ctrl			
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	b3	b2	b1	b0
0	0	X	X	X	X	X	X	0	0	1	0
X	1	X	X	X	X	X	X	0	1	1	0
1	X	X	X	0	0	0	0	0	0	1	0
1	X	X	X	0	0	1	0	0	1	1	0
1	X	X	X	0	1	0	0	0	0	0	0
1	X	X	X	0	1	0	1	0	0	0	1
1	X	X	X	1	0	1	0	0	1	1	1

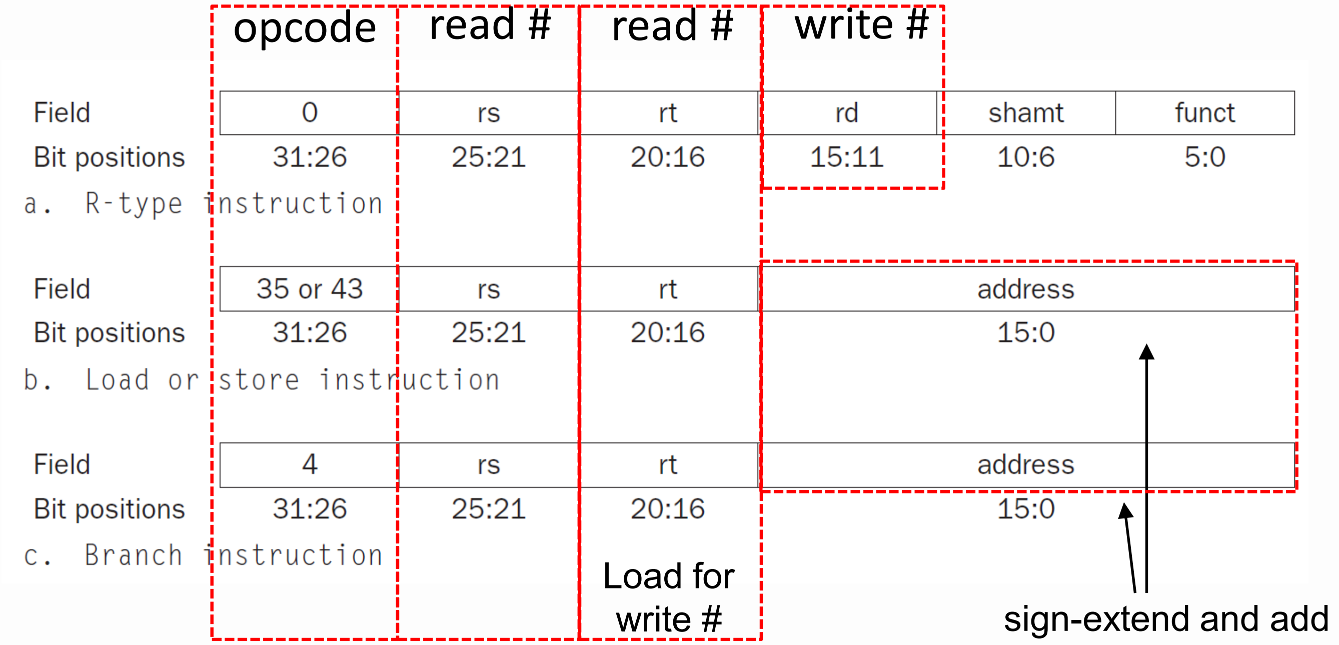
ALUCtrl3 = 0

ALUCtrl2 = ALUOp0 + ALUOp1·F2’·F1·F0’

ALUCtrl1 = ALUOp1’ + ALUOp1·F2’·F0’

ALUCtrl0 = ALUOp1·F3’·F2·F1’·F0 + ALUOp1·F3·F2’·F1·F0’

Control Signals from Instruction



opcode

read #

read #

write #

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

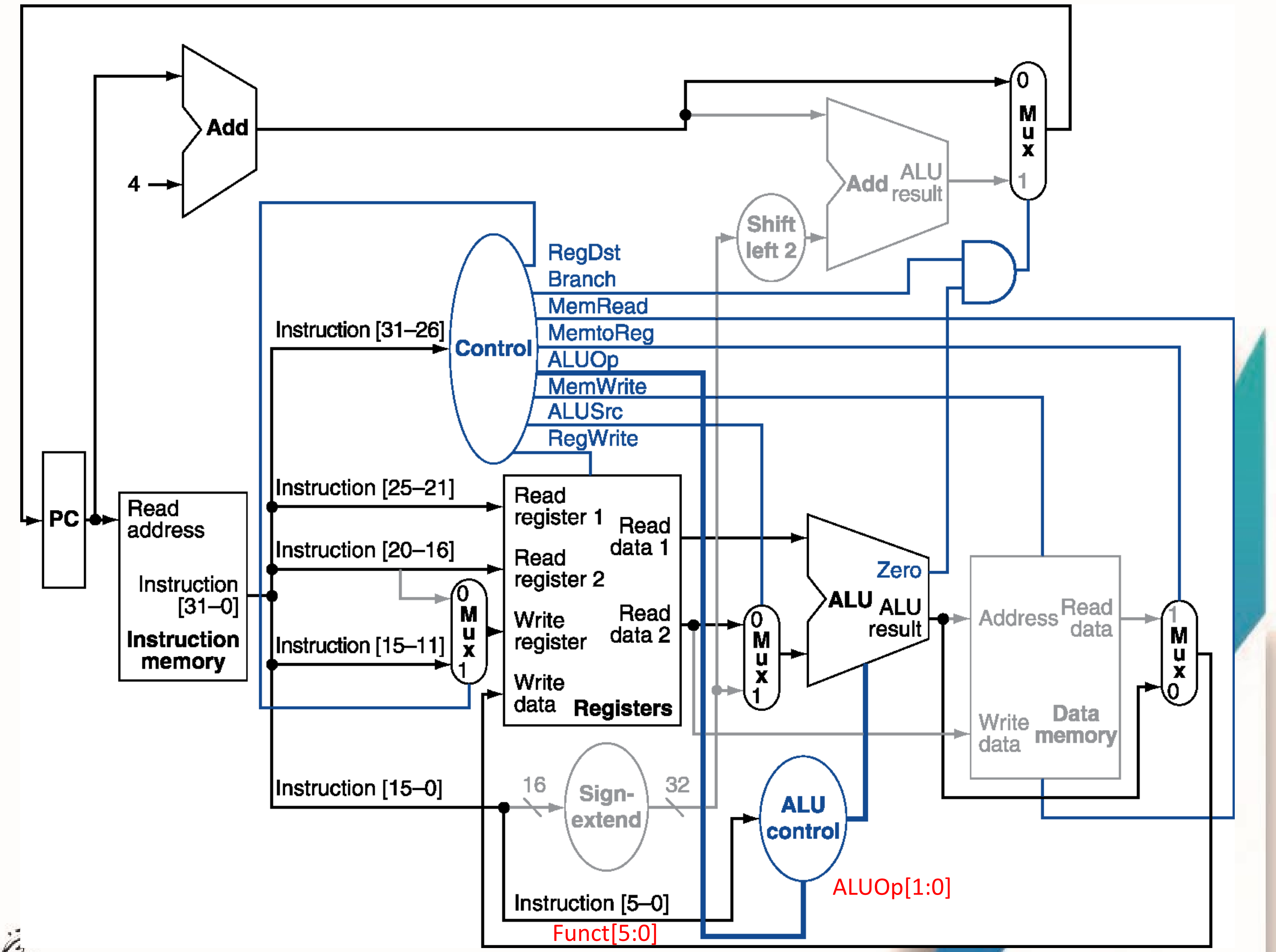
c. Branch instruction

Load for write #

sign-extend and add

R-Type Instruction

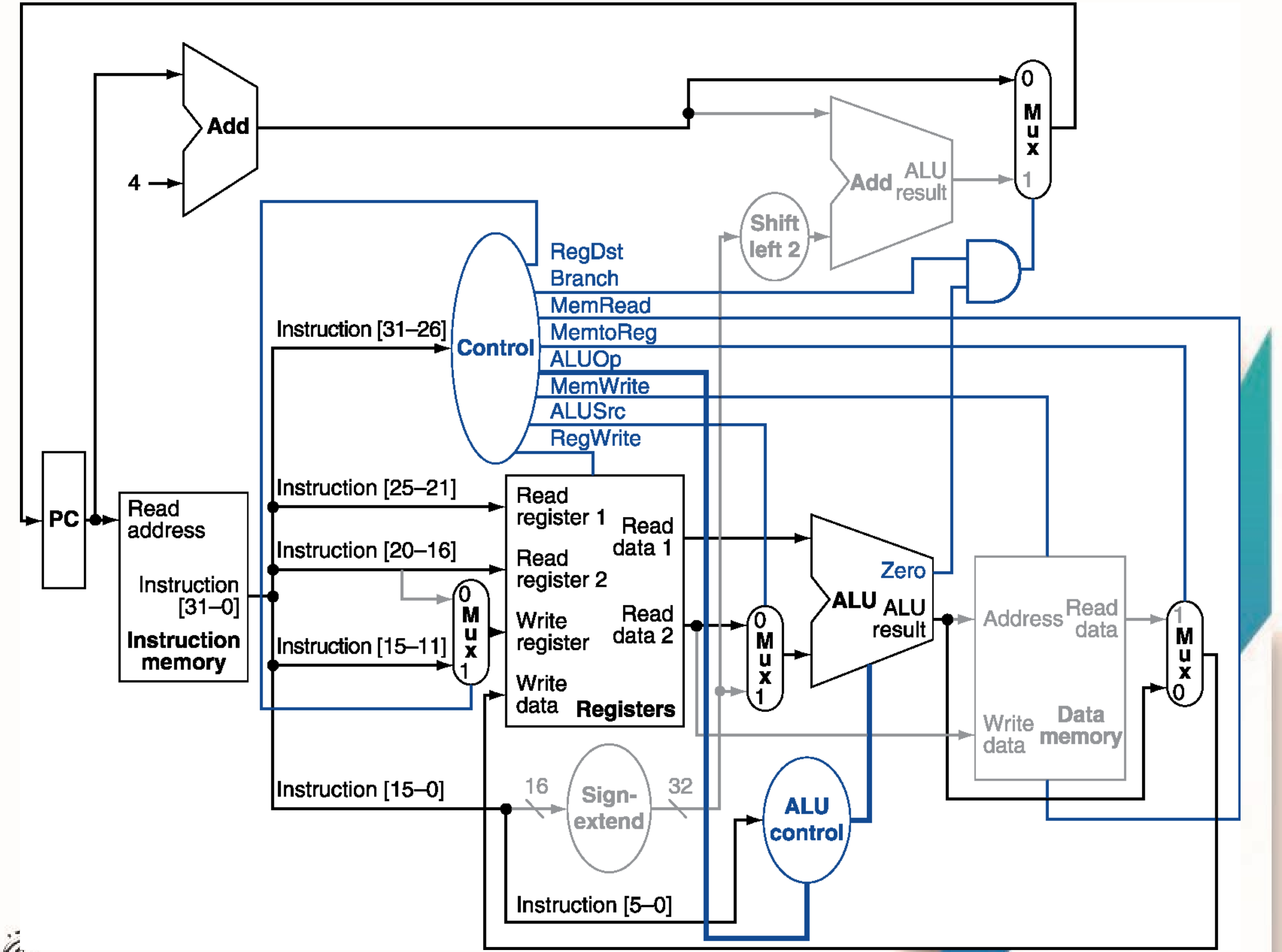
Control Signals = Opcode [31:26] + Funct[5:0]



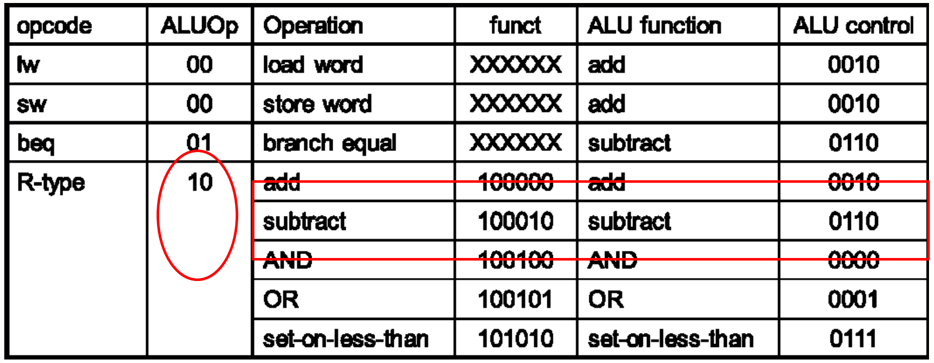
ALUOp[1:0]

Funct[5:0]

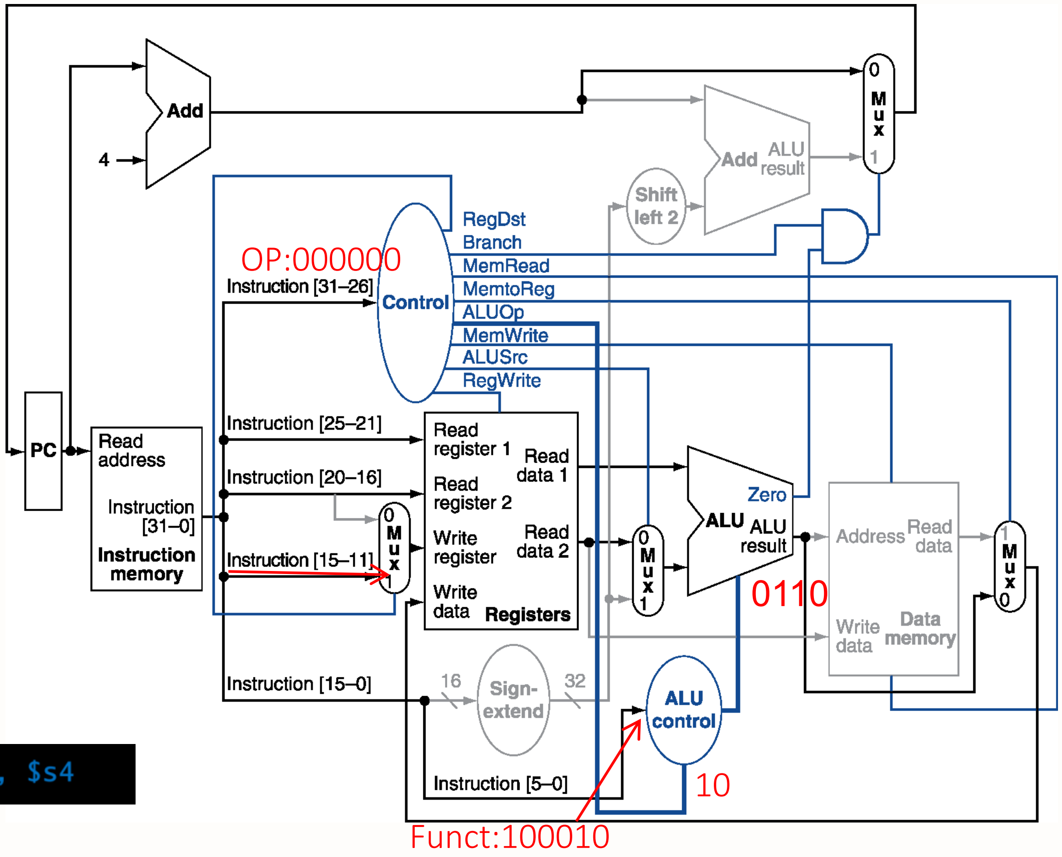
R-Type Instruction



R-Type Instruction

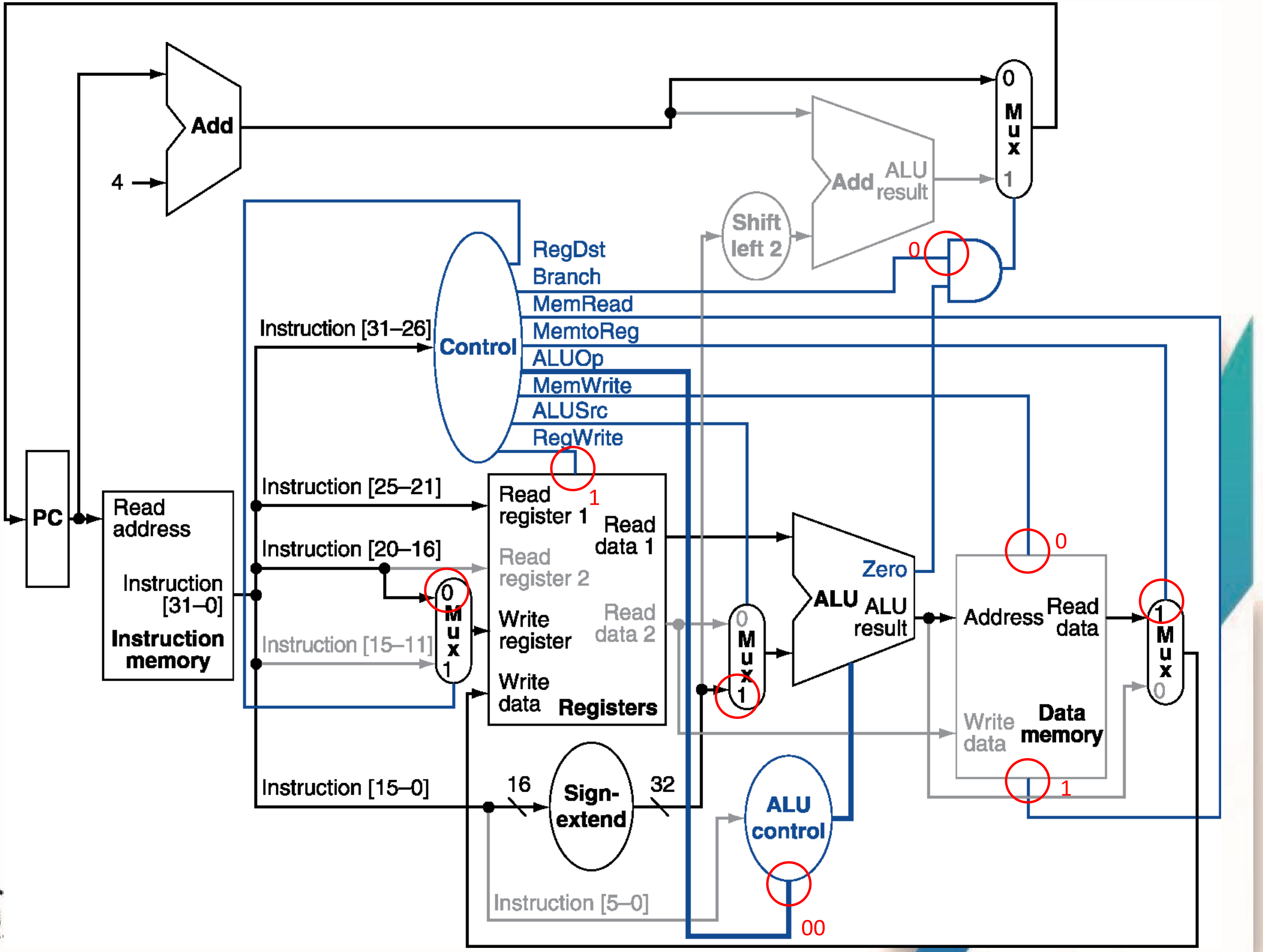


opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than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

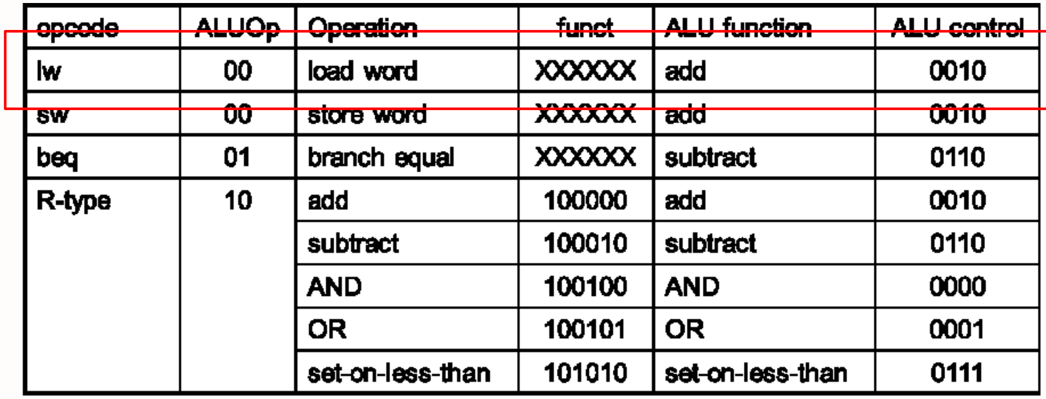


```
sub $t1, $s3, $s4
```

Load Instruction

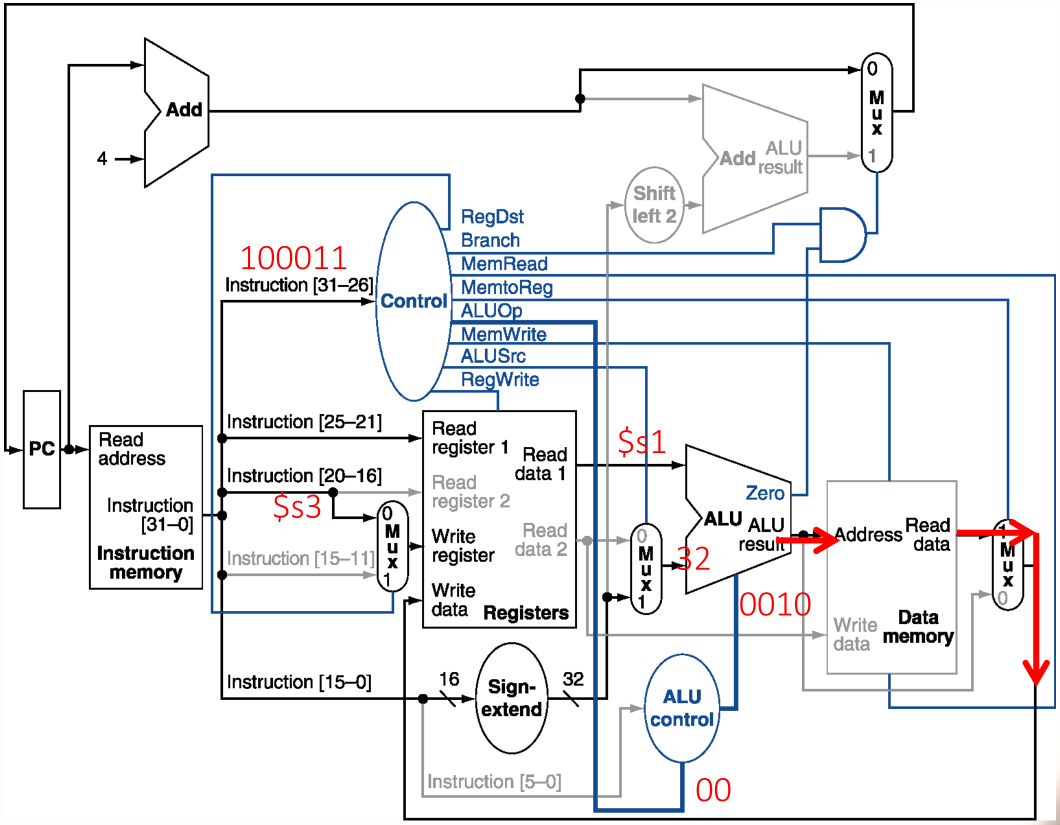


Load Instruction



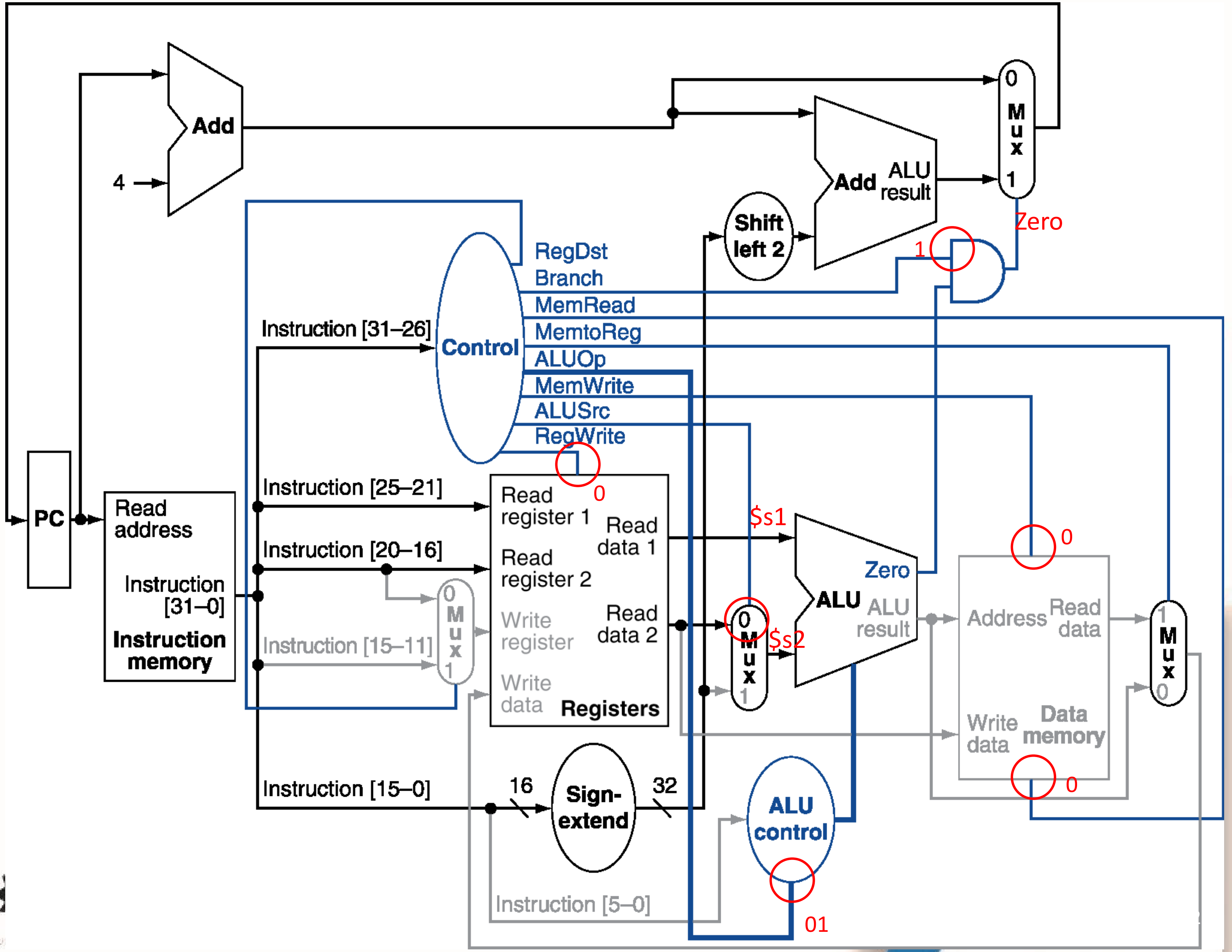
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than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

lw $s3, 32($s1)



Branch Instruction

beq $s1, $s2, Exit



Implementing Jumps

- Jump (`j` and `jal`) targets could be anywhere in text segment (starting at 0040 0000)
  - Encode full address in instruction

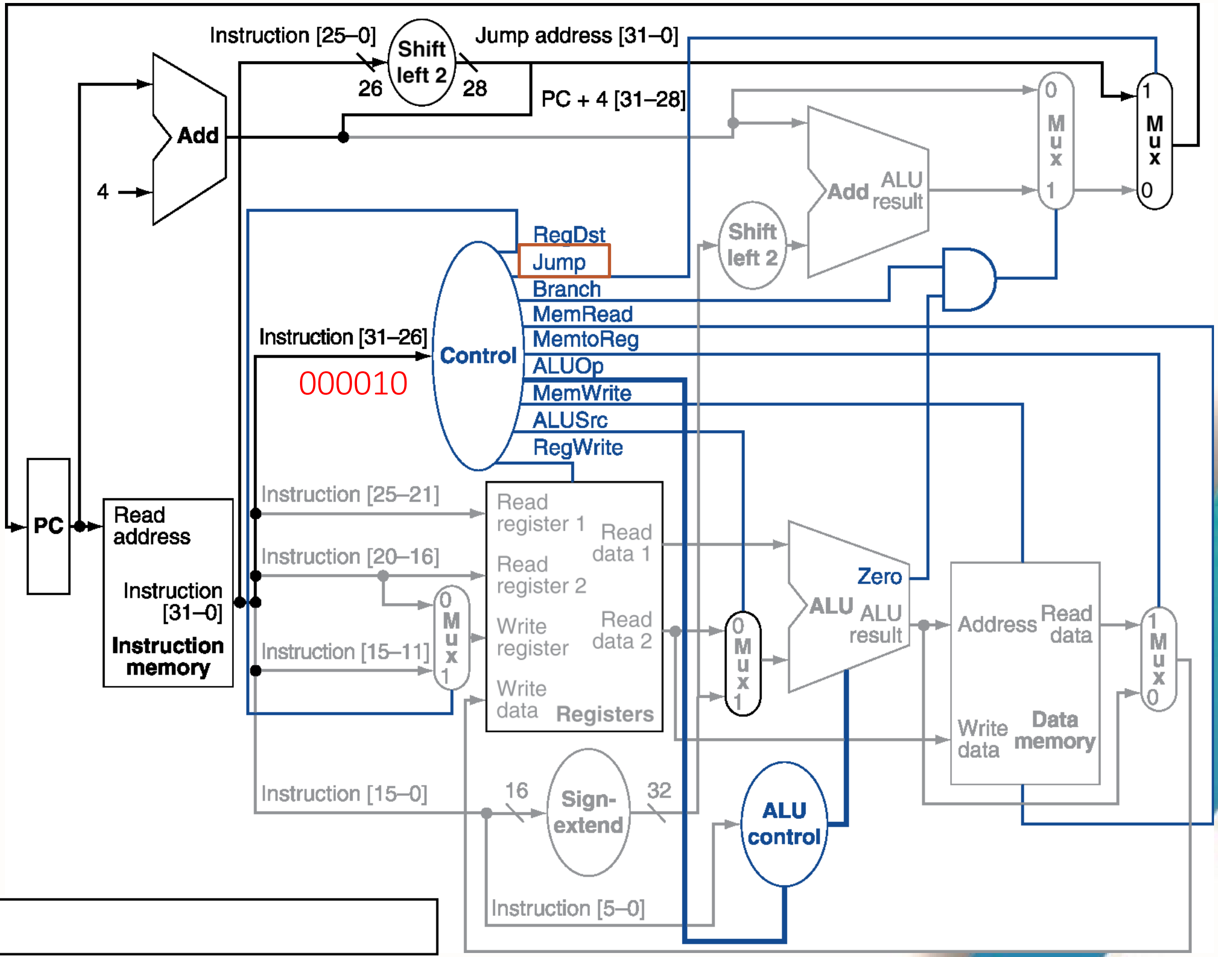
Jump	2	address
	31:26	25:0

- (Pseudo) Direct jump addressing
  - Target address = $\underline{\mathbf{PC}_{31:28}}$: address : 00

Top 4 bits of old PC

Need an extra control signal decoded from opcode

Jump Instruction

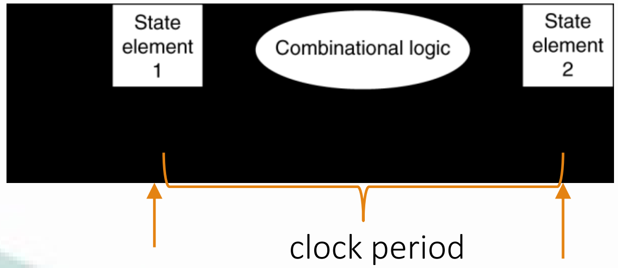


Jump	000010	address
	31:26	25:0

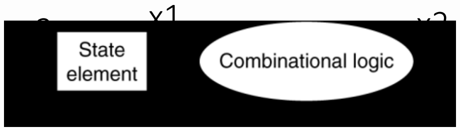
Clocking Methodology

- Combinational logic transforms data during clock cycles
- States changed and values updated when edge-triggered:
  - clock edge triggered after input signals become stable
  - A value is updated coming out a register. Next, the value is propagated through combinational logics. A new value is stored in a register. All are in same cycle.

A signal has to be transformed from S1 to S2 through C



rising edge-triggered

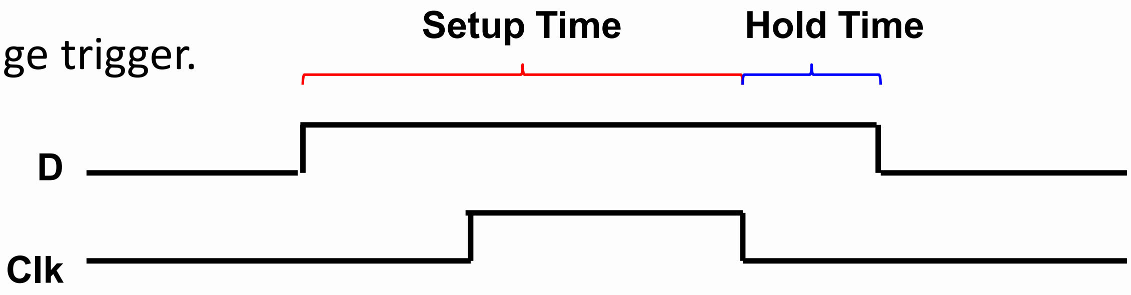


State will be x2 upon the next rising edge. Before that, it will still be x1.

Clock cycle: longest time needed for signals to propagate from one state element (through combinational logics) to reach the next storage element

Setup and Hold time Requirements

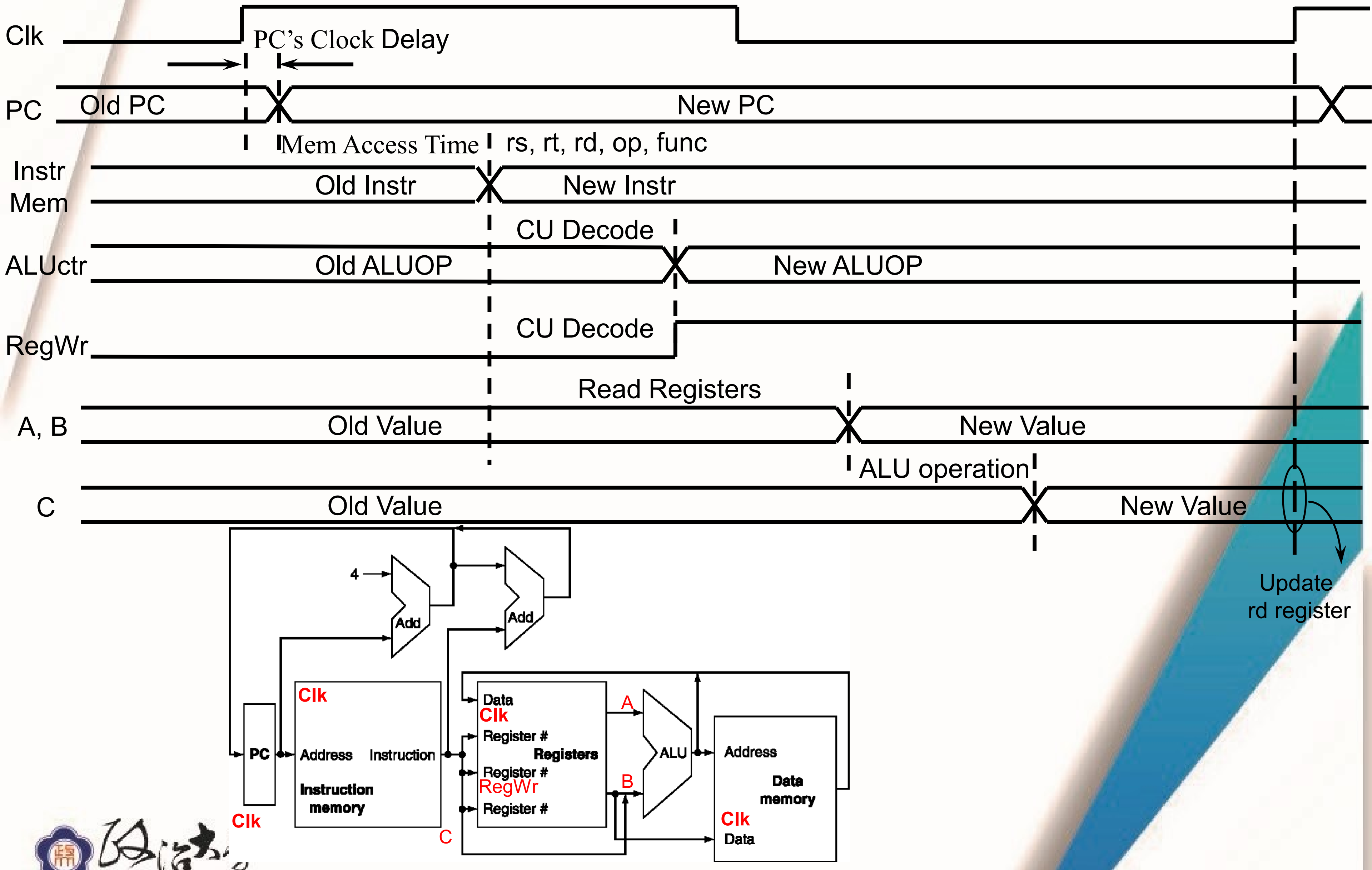
a falling-edge trigger.



Setup time: the input must remain stable for a period of time before the clock edge
Hold time: the input must remain stable for a period of time after the clock edge.

Failing to meet the time requirement could result in unpredictable output

Clock Timing Diagram (ex:R-type)



Ref: Prof. T.-T. Huang, CS4100 course slides, Designing a Single-Cycle Processor

Single-cycle Datapath

- It must have separate instruction and data memories

Since

- a. the formats of data and instructions are different in MIPS, and hence different memories are needed.
- b. having separate memories is less expensive.
- c. the processor operates in one cycle and cannot use a single-ported memory for two different accesses within that cycle

Drawback of Single-Cycle Design

- Single-cycle implementation requires every instruction have the clock cycle with the same length
- Cycle time depended on the instruction with the longest path - Load
  - Cycle time must be long enough for the load instruction:
    - PC’s Clock (signal transmission time)+ Instruction Memory Access Time + Register File Access Time + ALU Delay (address calculation) + Data Memory Access Time + **Register File Setup Time**
- Cycle time for load is much longer than the other instructions

Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction, which uses 5 functional units in series:
  - 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

    (making the worst case fast)
- Improve performance by pipelining

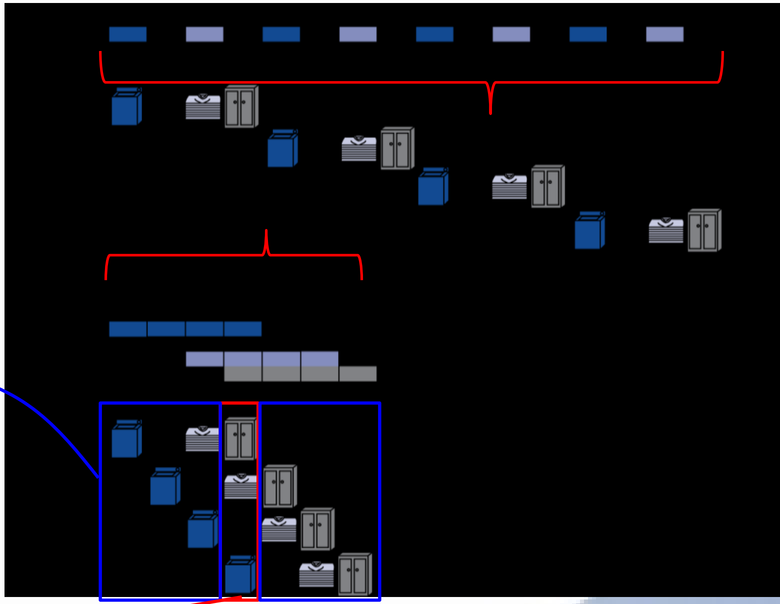
Summary

- MIPS makes control easier
  - Instructions have the same size
  - Source registers are always in same place
  - Immediates have same size, same location
- Single cycle datapath
  - CPI=1
  - Long clock cycle time

CPU Design: Pipelining

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance (throughput)
  - Not shorten the time needed for completing one load
- Assume each stage takes the same amount of time
- Let each stage take 0.5hr, and 4 stages in total for a task
- If we do four loads separately
  - 4 loads *4 stages * 0.5hr = 8hr
- Using pipelining ($k \geq 3$ loads)
  - (4 loads-3)*0.5hr+(3 stages + 3 stages)*0.5=3.5hr
- Using pipelining for infinite loads:
  - (n loads-3)*0.5hr+(3 stages + 3 stages)*0.5=0.5n+1.5hr



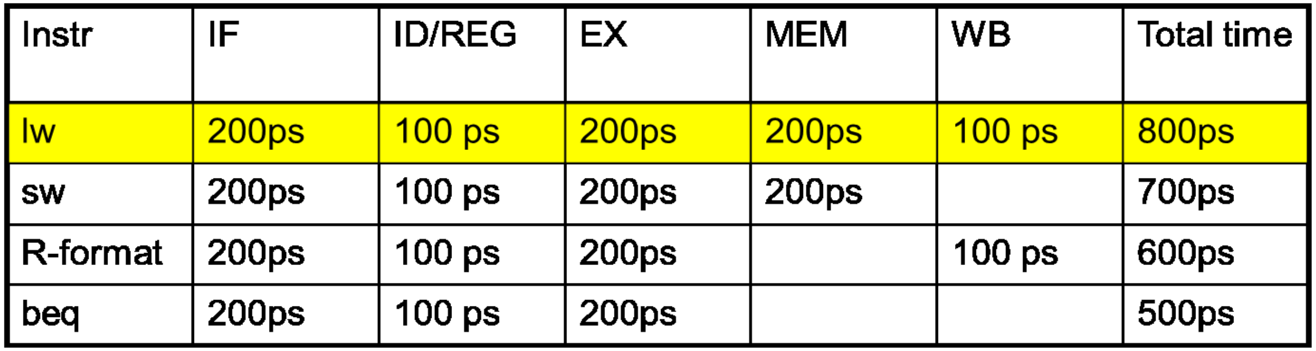
Max speed-up: $\lim_{n\to\infty} \frac{2n}{0.5n+1.5} = 4$

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
- Pipelining improves throughput instead of latency for a single instruction
- Instructions running in parallel using different hardware resources
- Clock rate is dependent on the slowest stage
- Max speed-up rate equals to the number of pipeline stages
- May need stalls for instruction dependences.

Pipeline Performance

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



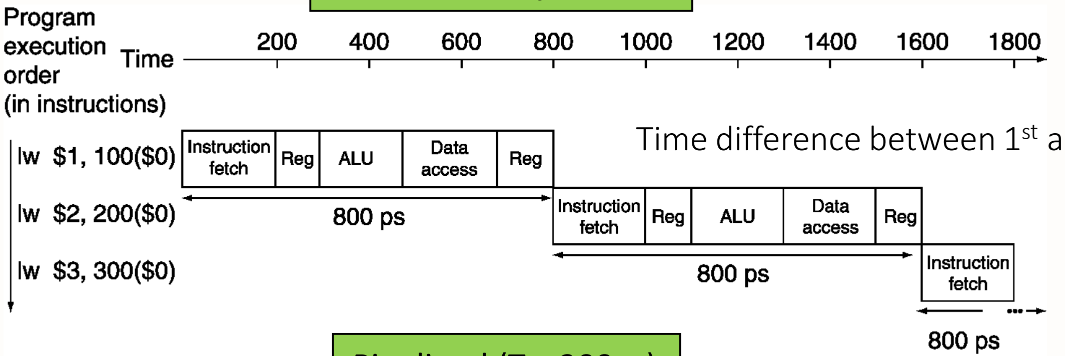
Instr	IF	ID/REG	EX	MEM	WB	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

$1 \text{ ps} = 10^{-12} \text{ s}$

Pipeline Performance

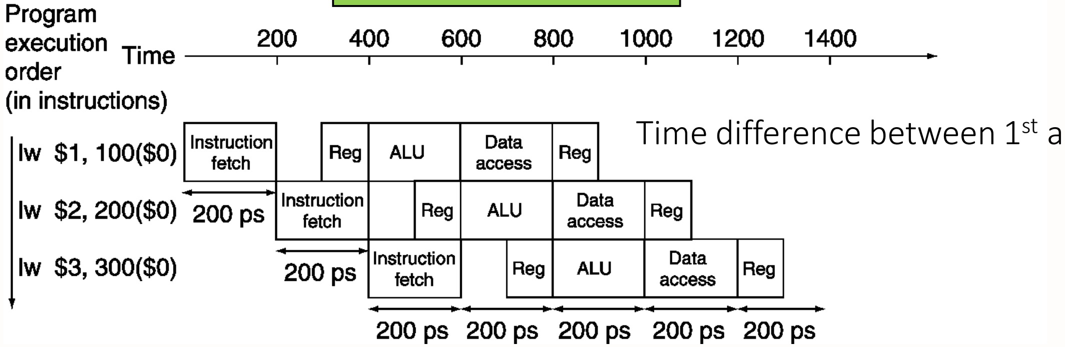
Single-cycle (T_c= 800ps)

T_c: avg. time between instructions



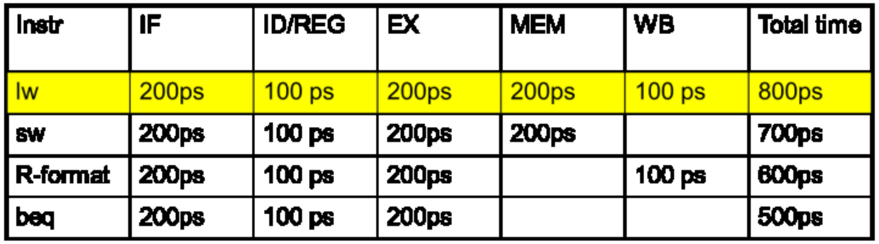
Time difference between 1st and 3rd is 1600ps

Pipelined (T_c= 200ps)



Time difference between 1st and 3rd is only 400ps

Pipeline Speedup



Instr	IF	ID/REG	EX	MEM	WB	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

When pipelining, each stage takes 200ps, so in total it's 1000ps.

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{\text{pipelined}}$
    $$= \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

$$\frac{1000\,ps}{5 stages} = 200\,ps$$

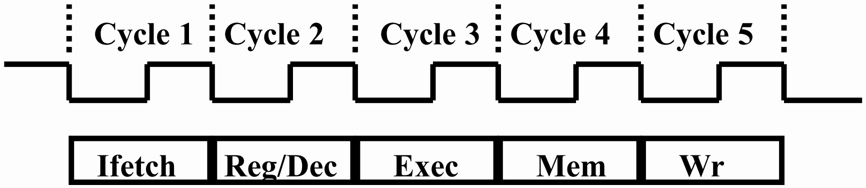
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

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 (in the ID stage)
  - Load and Store addressing (in two stages)
    - Calculate address in 3rd stage (EX)
    - Access memory in 4th stage (MEM)
  - Alignment of memory operands
    - Memory access takes only one cycle

The fact that the instruction formats of MIPS do not vary much and the length of the instructions is the same makes pipelining easy.

Pipeline Stages



IF: Instruction Fetch

Fetch the instruction from the Instruction Memory

ID: Instruction Decode

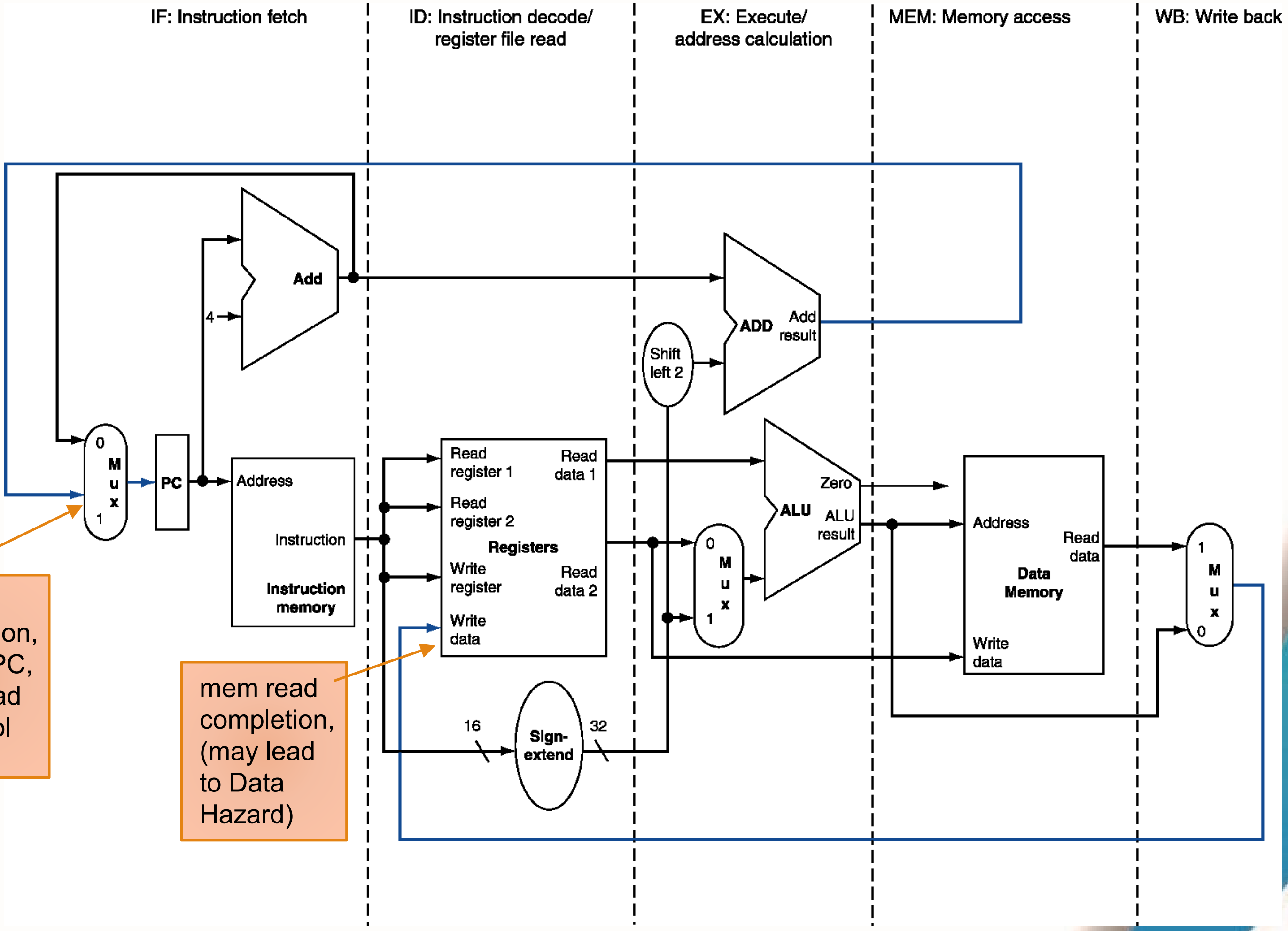
Registers fetch and instruction decode

EX: Calculate the memory address

MEM: Read the data from the Data Memory

WB: Write the data back to the register file

MIPS Pipelined Datapath - What do we need to split an instruction into stages?



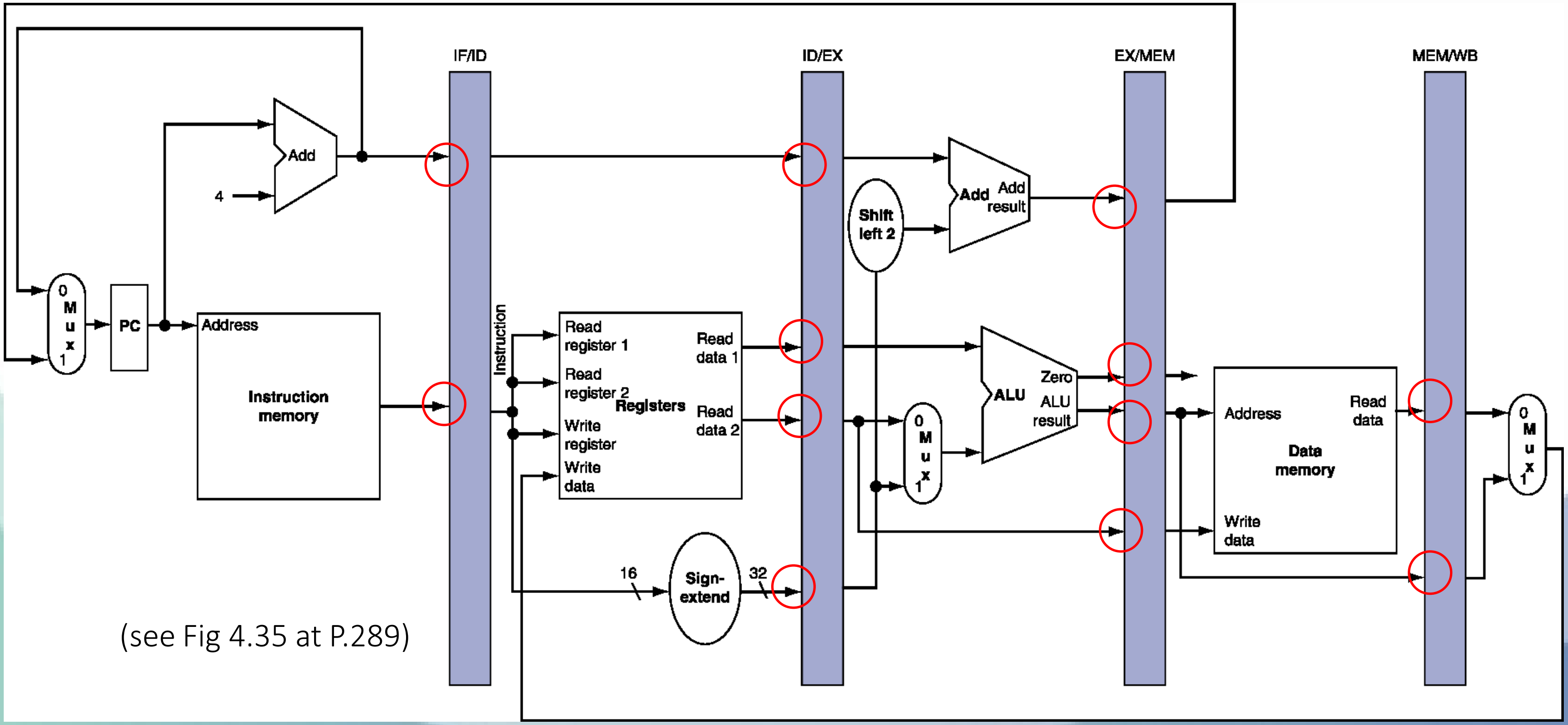
Branch Completion, write to PC, (may) lead to Control Hazard

mem read completion, (may lead to Data Hazard)

Pipelined Version of Datapath

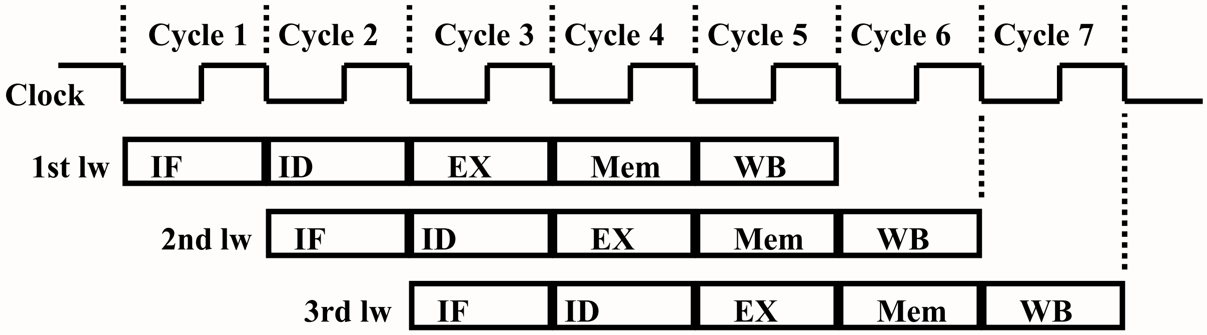
- Need registers between stages
  - To hold information produced in previous cycle (including data and control)

Pipelined registers:
IF/ID, ID/EX, EX/MEM, MEM/WB



(see Fig 4.35 at P.289)

64 bits 128 bits 97 bits 64 bits



There are 5 functional units in the pipeline datapath are:

- Instruction Memory for the IF stage
- Register File's Read ports for the ID stage
- ALU for the EXE stage
- Data Memory for the MEM stage
- Register File's Write port for the WB stage

Example - lw

- lw $t0, 1200($t1)

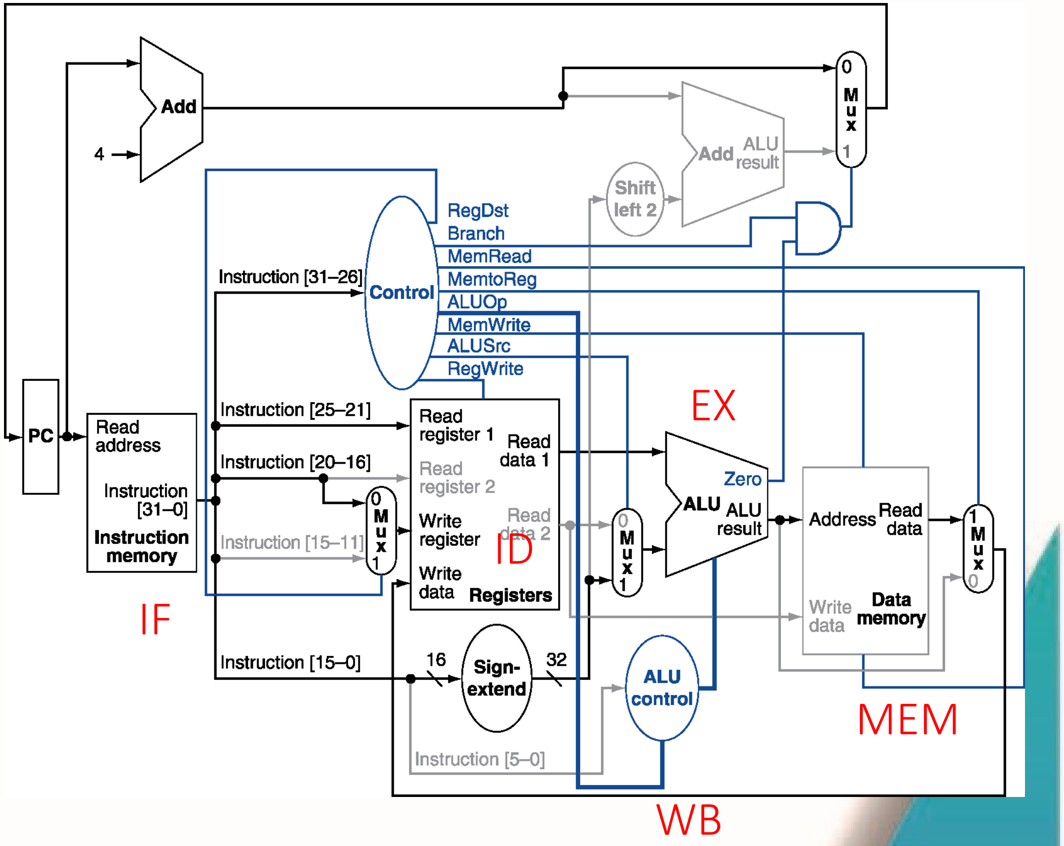
R8=01000
rt

R9=01001
rs

$$1200_{ten} = 0000010010110000_{two}$$

100011	01001	01000	0000 0100 0110 0000
6 bits	5 bits	5 bits	16 bits
op	rs	rt	address

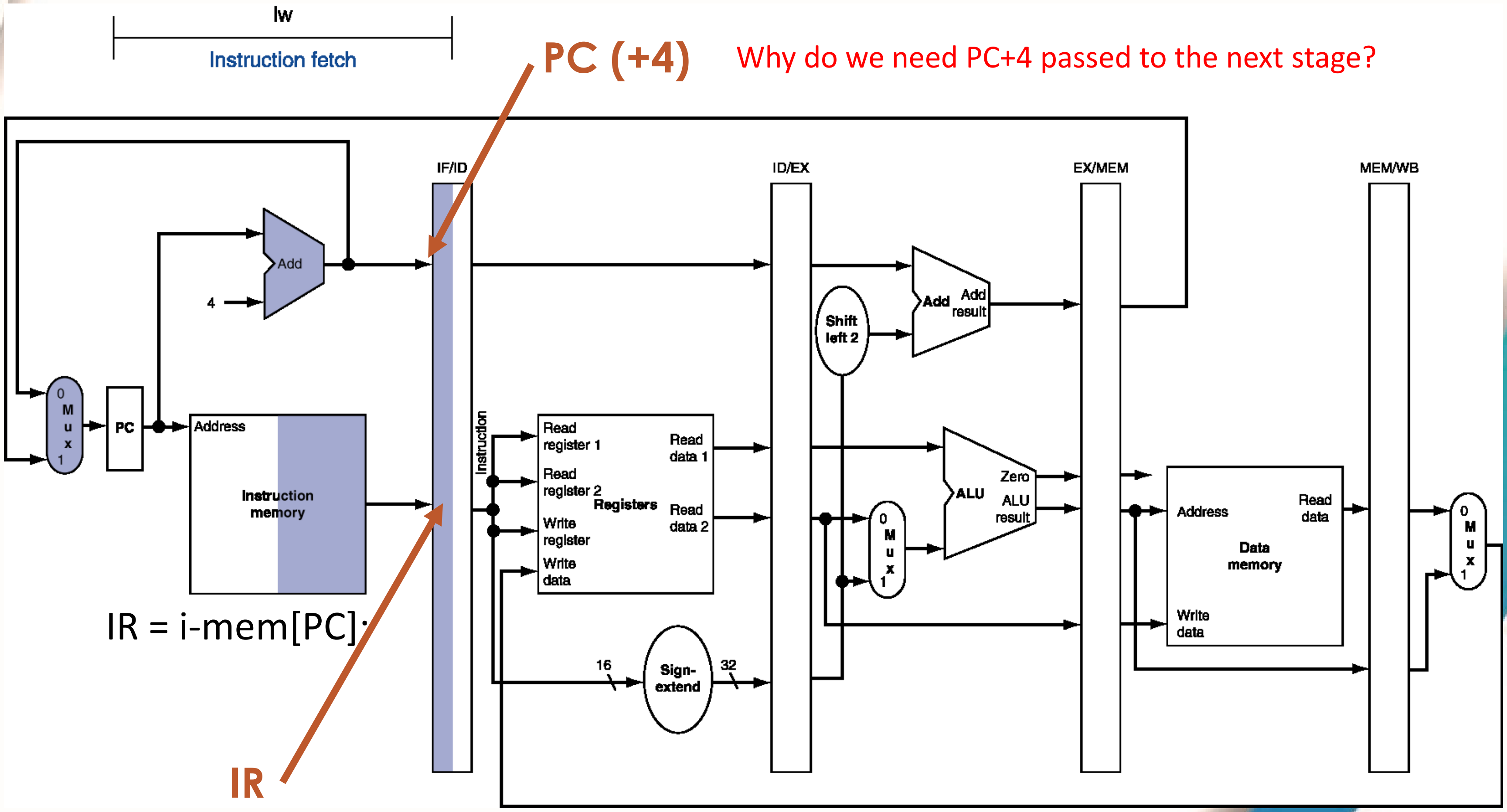
lw $t0, 1200($t1)



- IF: Fetch the instruction from the Instruction Memory
- ID: Registers fetch and instruction decode
- EX: Calculate the memory address
- MEM: Read the data from the Data Memory
- WB: Write the data back to the register file

IF Stage - lw

Fetch the instruction from the Instruction Memory



PC (+4)

Why do we need PC+4 passed to the next stage?

IR = i-mem[PC];

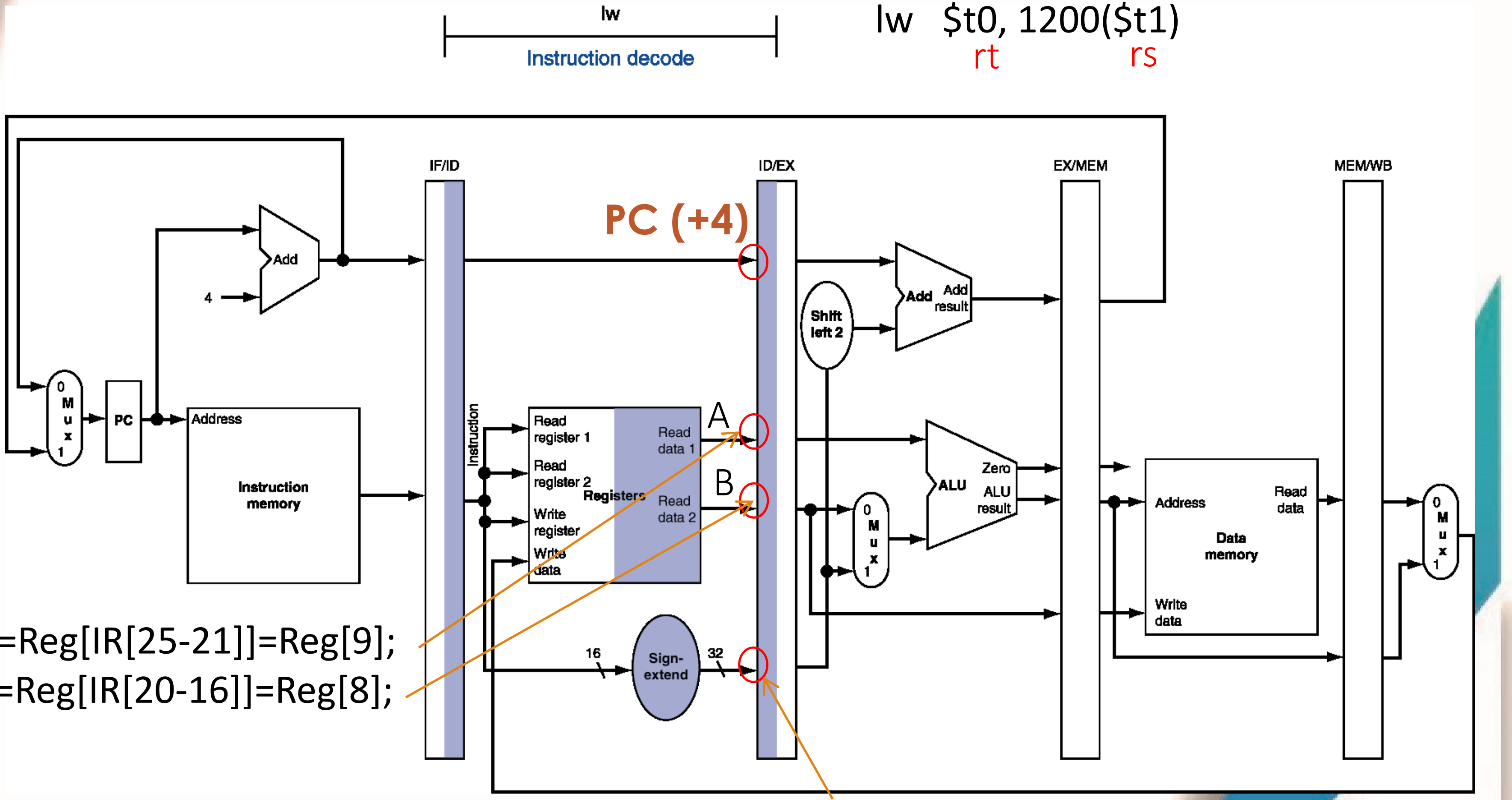
IR

lw $t0, 1200($t1) [redacted]

ID Stage - lw

Fetch registers and decode the instruction

lw $t0, 1200($t1)
rt   rs



PC (+4)

A

B

A = rs=Reg[IR[25-21]]=Reg[9];

B = rt=Reg[IR[20-16]]=Reg[8];

0000 0000 0000 0000 0000 0100 0110 0000

sign-extension

100011	01001	01000	0000 0100 0110 0000
6 bits	5 bits	5 bits	16 bits
op	rs	rt	address

EX Stage - lw

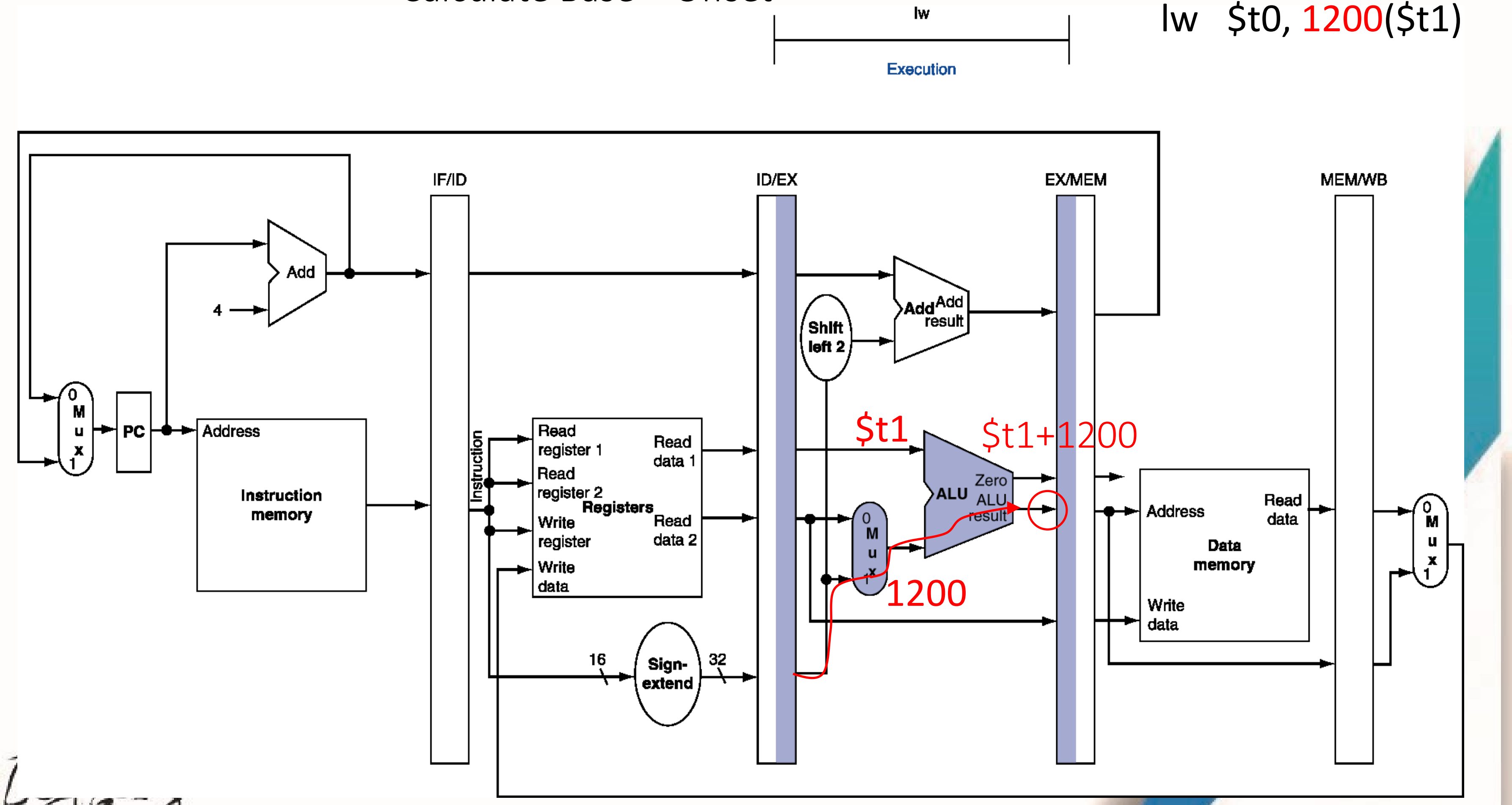
Compute the memory address ($t1+1200)

$t1 1200

ALUout = A + sign-ext(IR[15-0])

Calculate Base + Offset

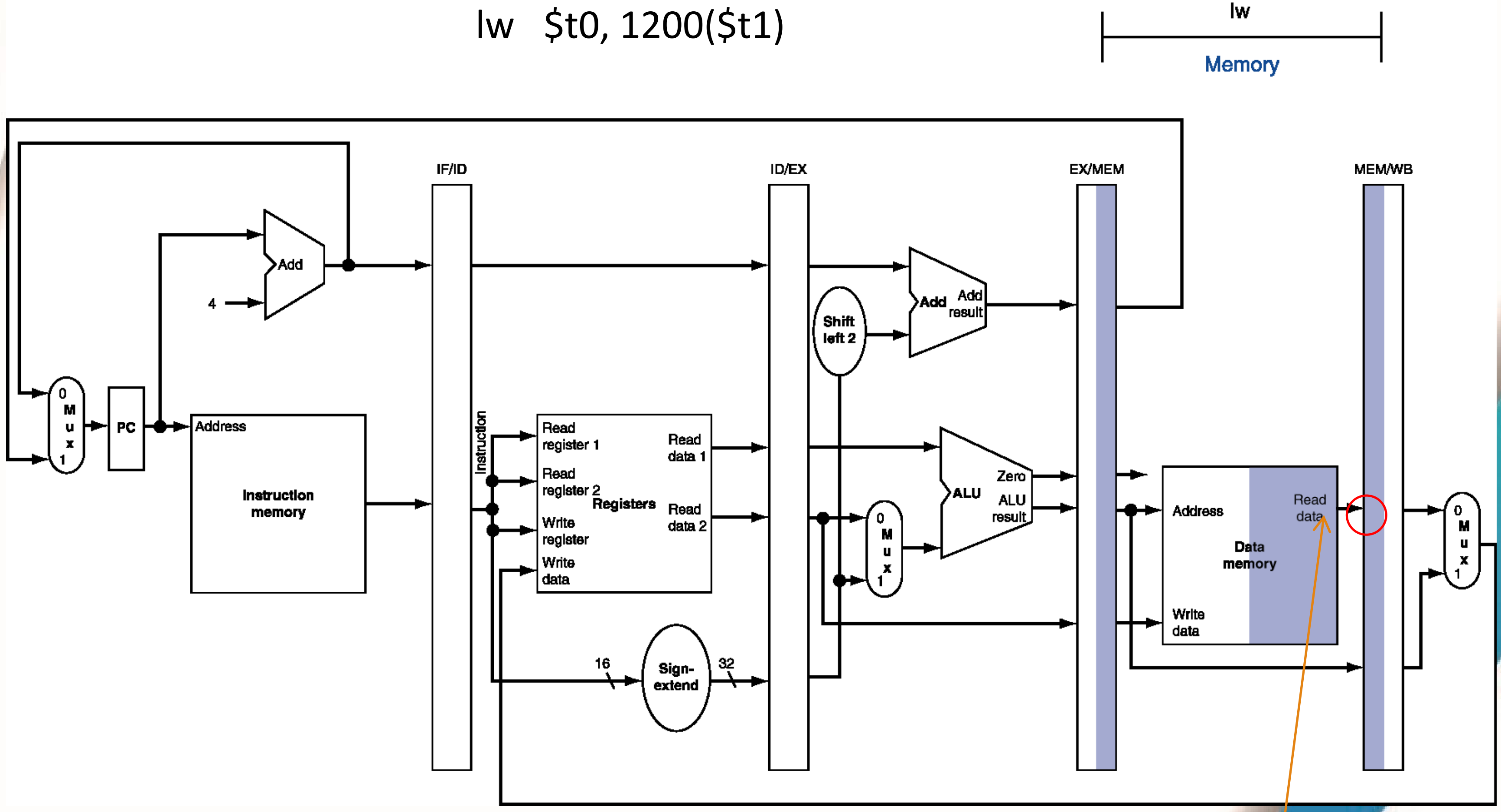
lw $t0, 1200($t1)



MEM Stage - lw

Read data from the Data Memory

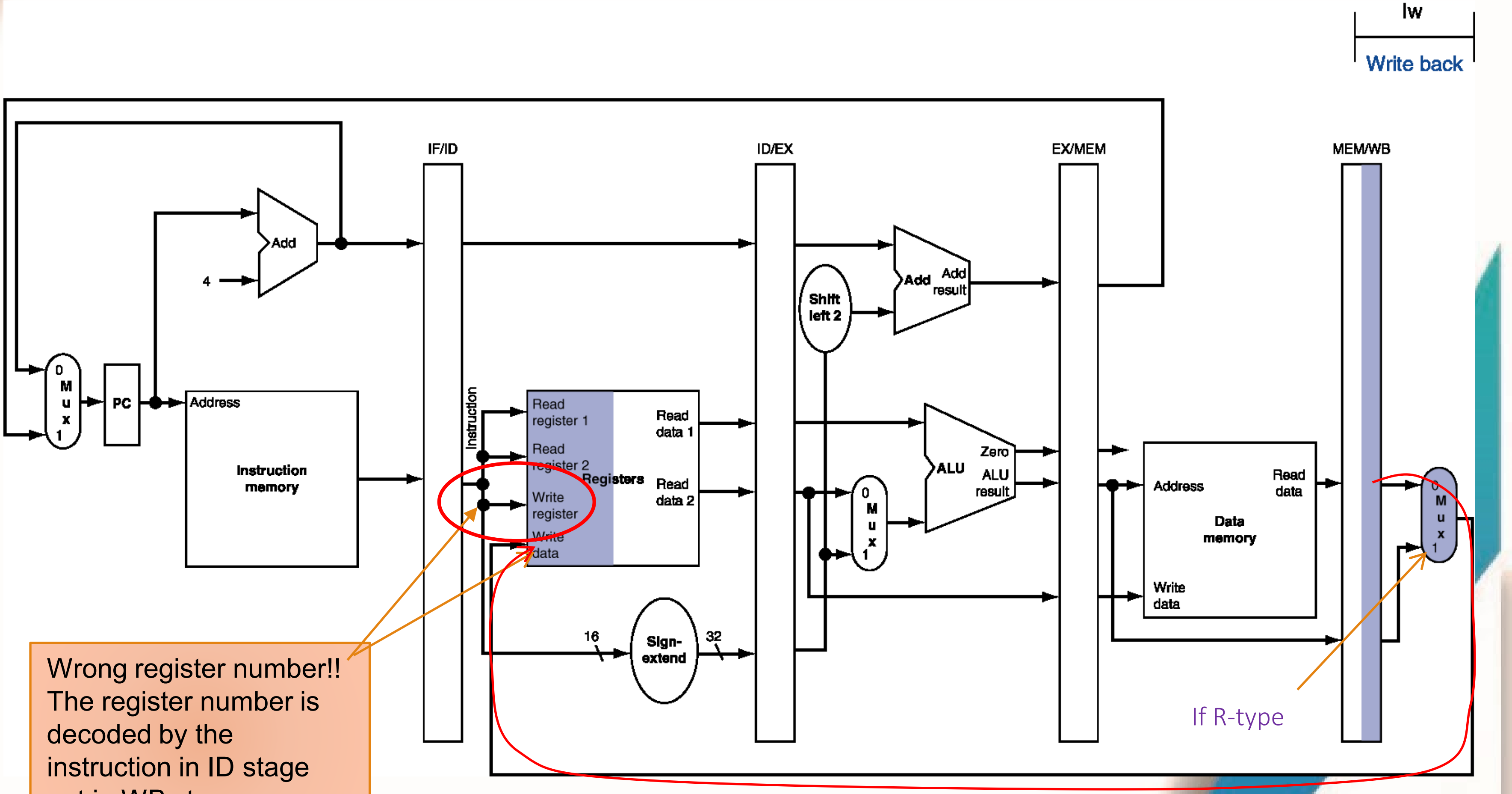
lw $t0, 1200($t1)



Read MEM[$t1+1200]

WB Stage - lw

Write the data back to the register

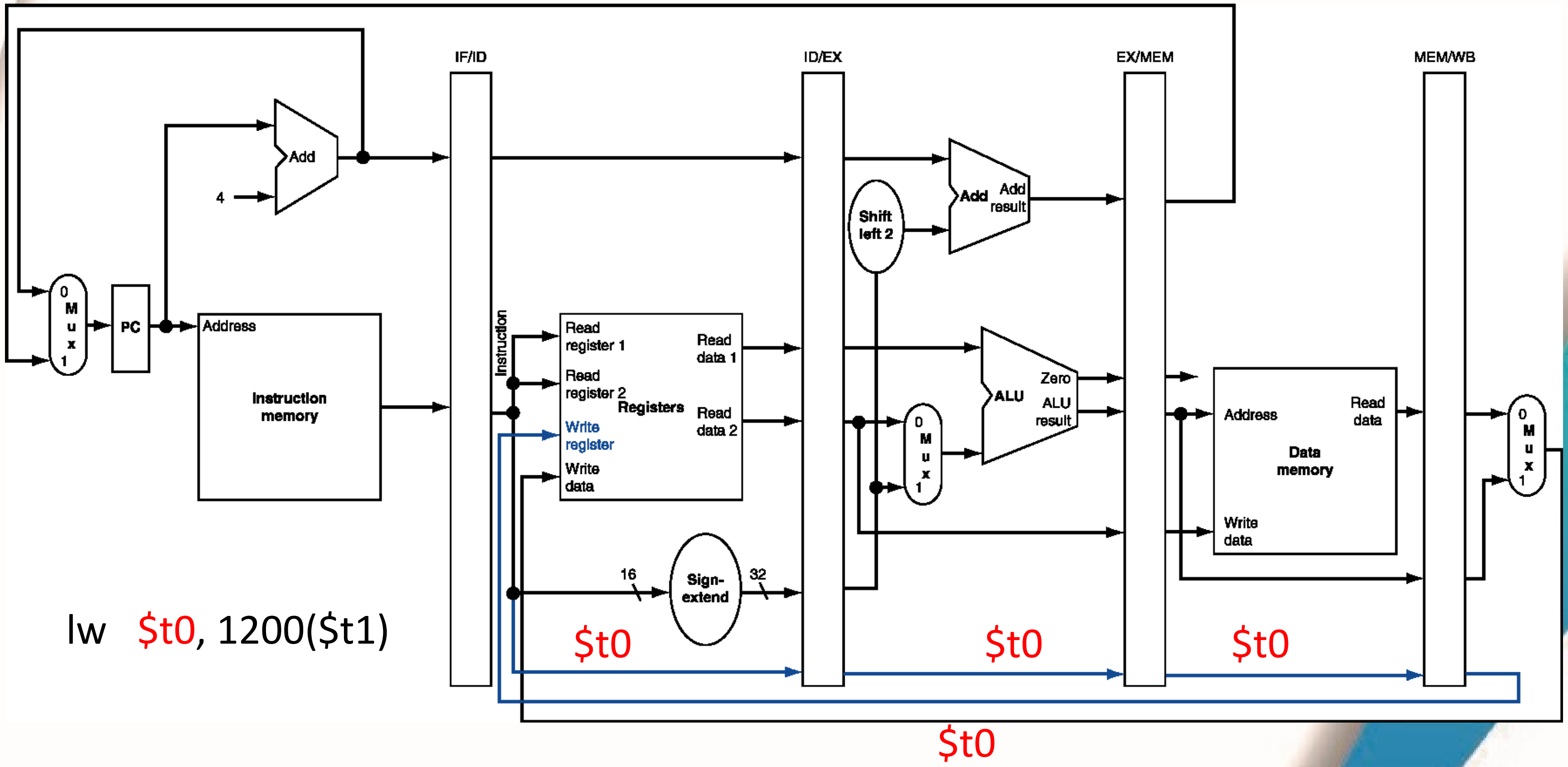


Wrong register number!! The register number is decoded by the instruction in ID stage not in WB stage

If R-type

WB Stage - lw

Corrected Datapath for Load



lw $t0, 1200($t1)

Write register number shall be decided by the instruction in WB stage.