EEL 4768
Computer Architecture

Single-Cycle Datapath

Outline

- Review of Instruction Formats
- Building a Single-Cycle Datapath
  - Control Signals and Multiplexers
  - Main Control vs. ALU Control
- Paths for Each Instruction Type
- Clock Signal

Single Cycle Datapath

- The single-cycle datapath:
  - Simple implementation
  - Each instruction takes one clock cycle to execute
  - Clock-Per-Instruction measure is: **CPI = 1**

- CPU performance:

Execution Time = Instruction Count * CPI * Clock Cycle Time

Execution Time = Instruction Count * Clock Cycle Time

- Pro: Simplicity of implementation
- Con: Not very fast, no parallelism

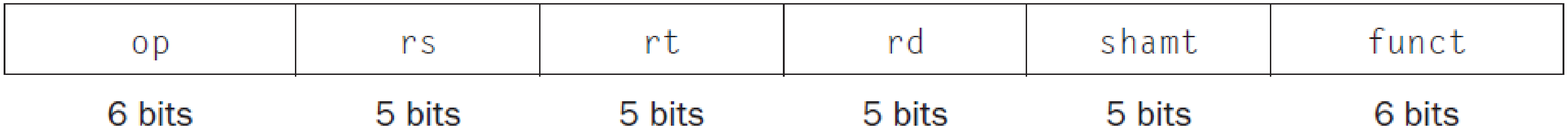
A Single-Cycle Implementation

- We will work with a subset of the MIPS instructions:

Type	Instruction	Syntax
Arithmetic	add	add $t0, $t1, $t2
	sub	sub $t0, $t1, $t2
	slt (set-on-less-than)	slt $t0, $t1, $t2 # t0=1 if t1<t2; else t0=0
Logic	and	and $t0, $t1, $t2
	or	or $t0, $t1, $t2
Data Transfer	lw (load word)	lw $t0, 12($t1)
	sw (store word)	sw $t0, 40($t1)
Decision	beq (branch-on-equal)	beq $t0, $t1, Label
	j (jump)	j Label

Review of Instruction Format: R-type

- This format is used by these instructions: add, sub, and, or, slt



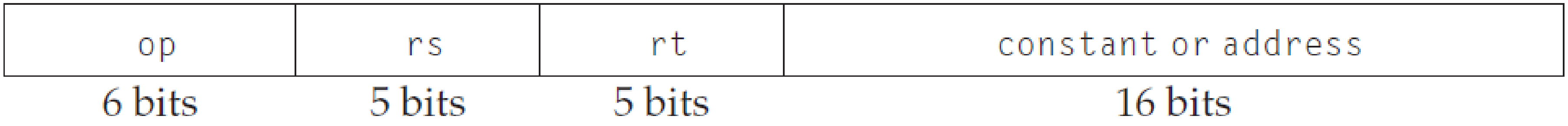
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- What will the CPU do?
  - Reads ‘rs’ and ‘rt’
  - Inspect ‘opcode’ and ‘funct’ to determine the operation (in table below)
  - Perform the operation and save the result in ‘rd’

	add	sub	and	or	slt
op field	0	0	0	0	0
funct field	32	34	36	37	42

Review of Instruction Format: I-type

- This format is used for Load Word (lw) and Store Word (sw)
  - ‘lw’ opcode=35 and ‘sw’ opcode=43



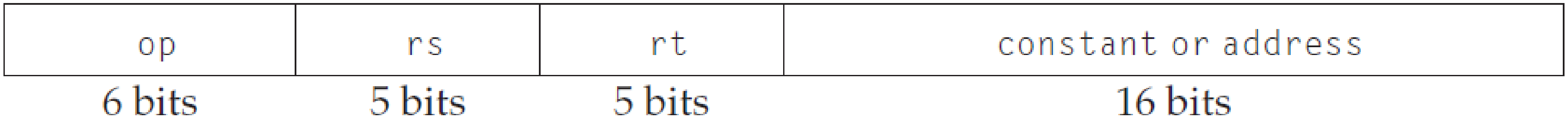
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- What will the CPU do with this instruction?
  - Computes the address as:

    ```
    Address =  rs+ (constant field sign-extended)
    ```
  - For ‘lw’, reads from the address and saves into ‘rt’
  - For ‘sw’, saves the data at ‘rt’ at the computed address

Review of Instruction Format: I-type

- Branch-on-equal (beq) uses the I-Type format below



op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- What does the CPU do for ‘beq’?
  - Read and compare the registers ‘rs’ and ‘rt’
  - Compute the branch address as:

```
Address = PC + 4 + shiftedLeft2[sign-extended(constant)]
```

  - If the two registers are equal, the PC becomes the branch address

```
PC = Address
```

Review of Instruction Format: J-type

- This is the format of the jump (‘j’) instruction

op (6 bits)	**address** (26 bits)

j instruction

- What does the CPU do for ‘j’?
  - The jump is taken unconditionally
  - PC is set to the jump address:

```
PC = [leftmost 4 bits of PC+4] [26 bits field in 'j' instruction] [00]
```

Instruction Fields Index

- Each instruction field is designated by its bit positions

  E.g.: i[31-26] designates the leftmost 6 bits of the instruction (the opcode)

i[31-26]	i[25-21]	i[20-16]	i[15-11]	i[10-6]	i[5-0]
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

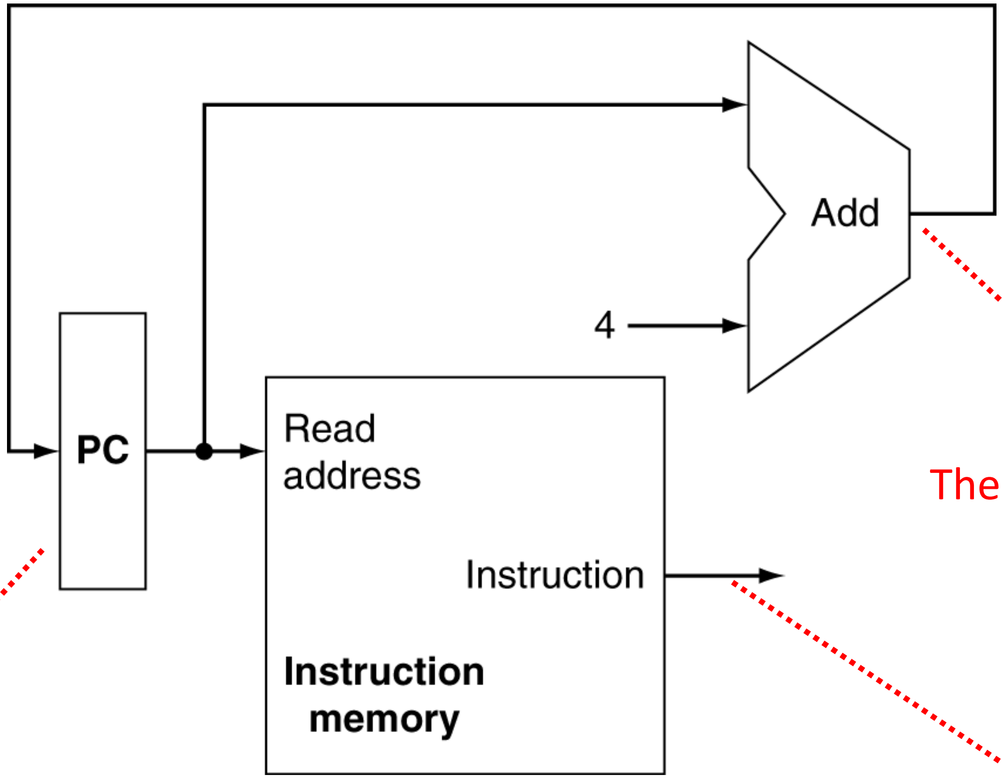
i[31-26]	i[25-21]	i[20-16]	i[15-0]
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

i[31-26]	i[25-0]
op (6 bits)	**address** (26 bits)

Instruction Fetch

- Now, we will start to build the datapath

Address of next instruction is written back to the PC



Here is: PC+4
The address of the next instruction

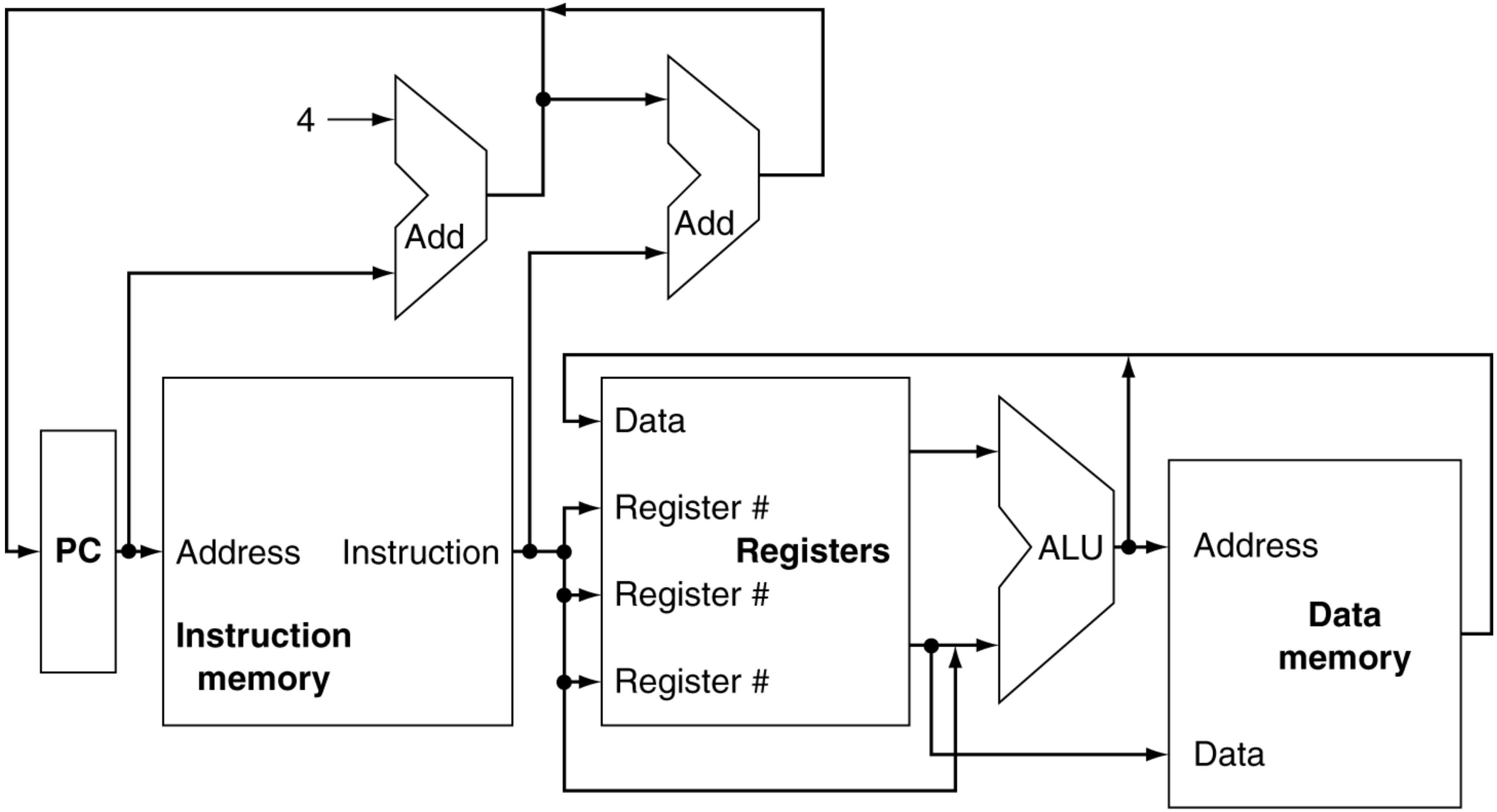
Address of current instruction

The current instruction is here

Instruction Fetch Stage

Building the Datapath

- We will keep adding more components



R-Type Instruction

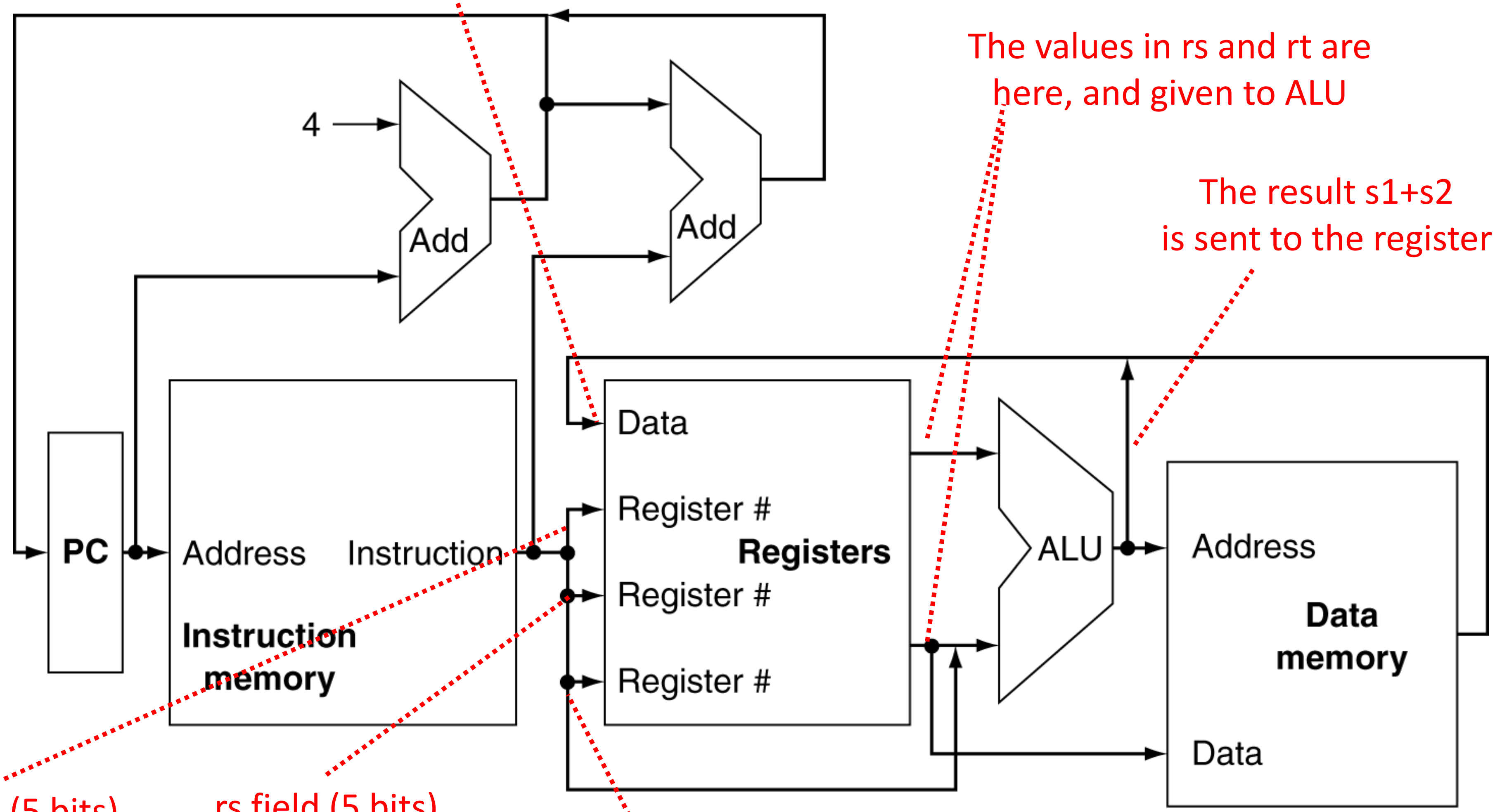
add t0, s1, s2

	s1	s2	t0		
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

The result from the ALU is written to register t0

The values in rs and rt are here, and given to ALU

The result s1+s2 is sent to the register



rd field (5 bits)
Register t0

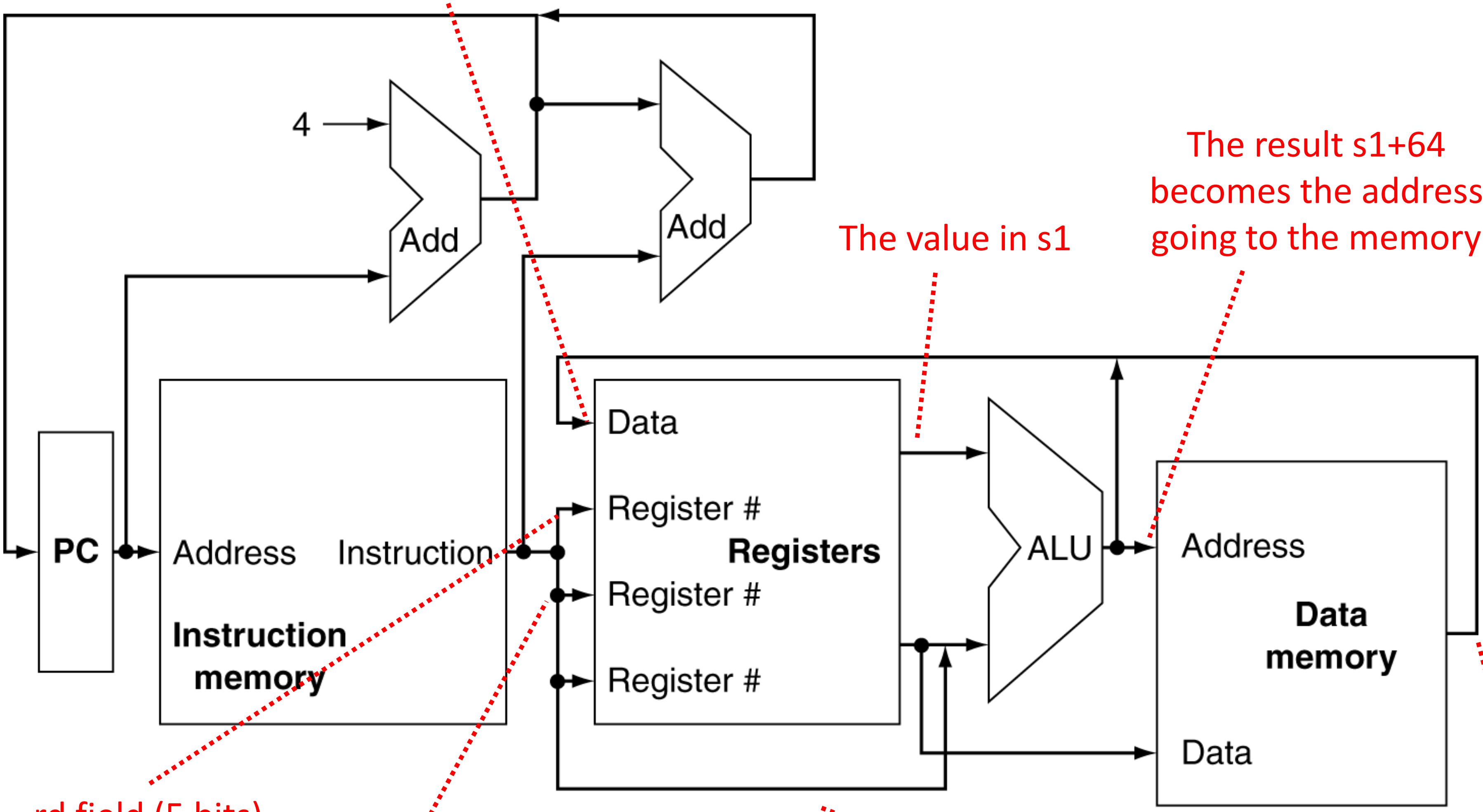
rs field (5 bits)
Register s1

rt field (5 bits)
Register s2

Load Word Instruction

lw t0, (64)s1	s1	t0	0100 0000 (64)
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

The word from memory is written to register t0



The value in s1

The result s1+64 becomes the address going to the memory

rd field (5 bits) Register t0

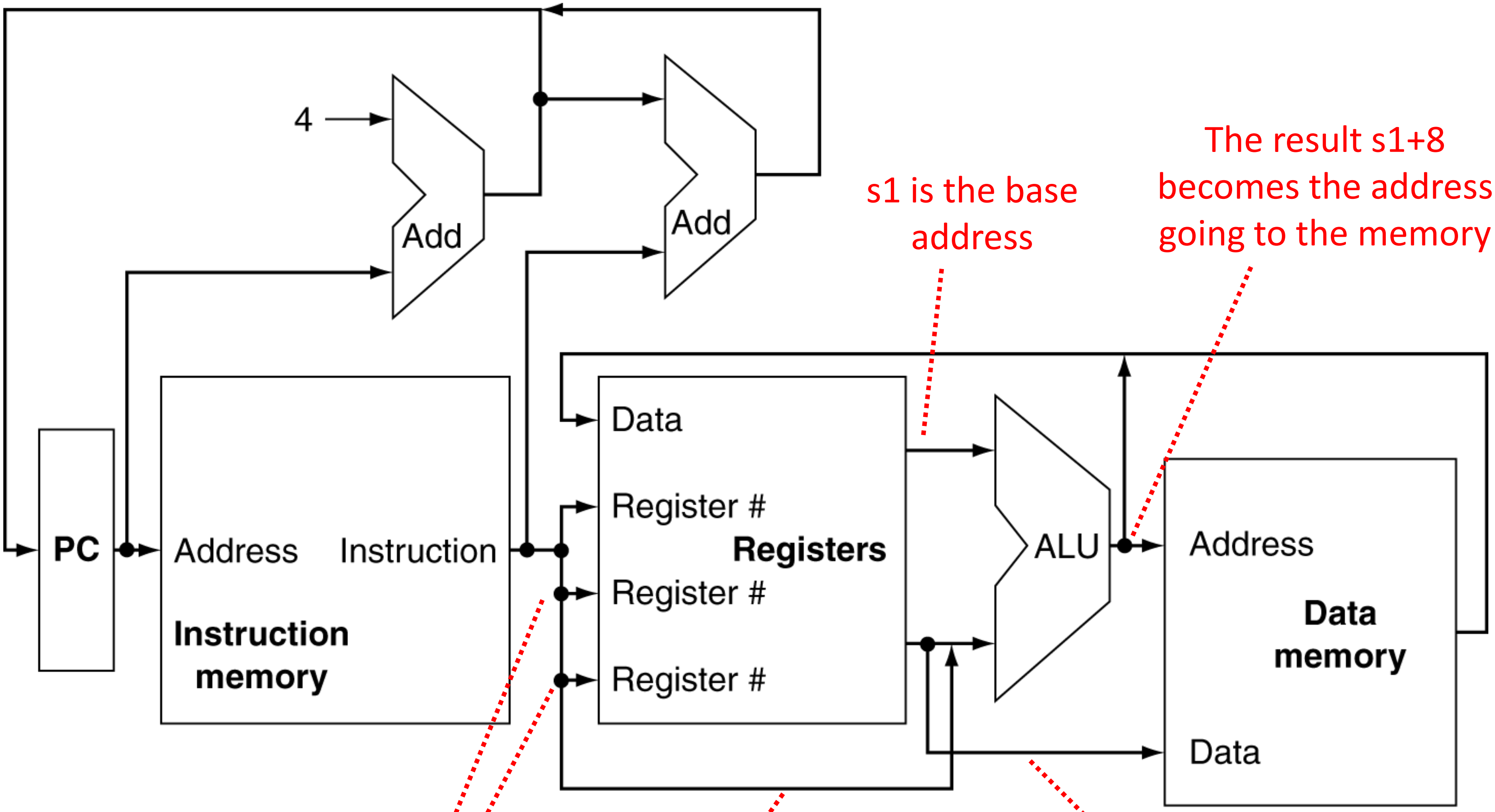
rs field (5 bits) Register s1

16-bit number in instruction; equal to 64

The data is read from memory and sent to the registers

Store Word Instruction

sw t0, (8)s1	s1	t0	0000 1000 (8)
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits



s1 is the base address

The result s1+8 becomes the address going to the memory

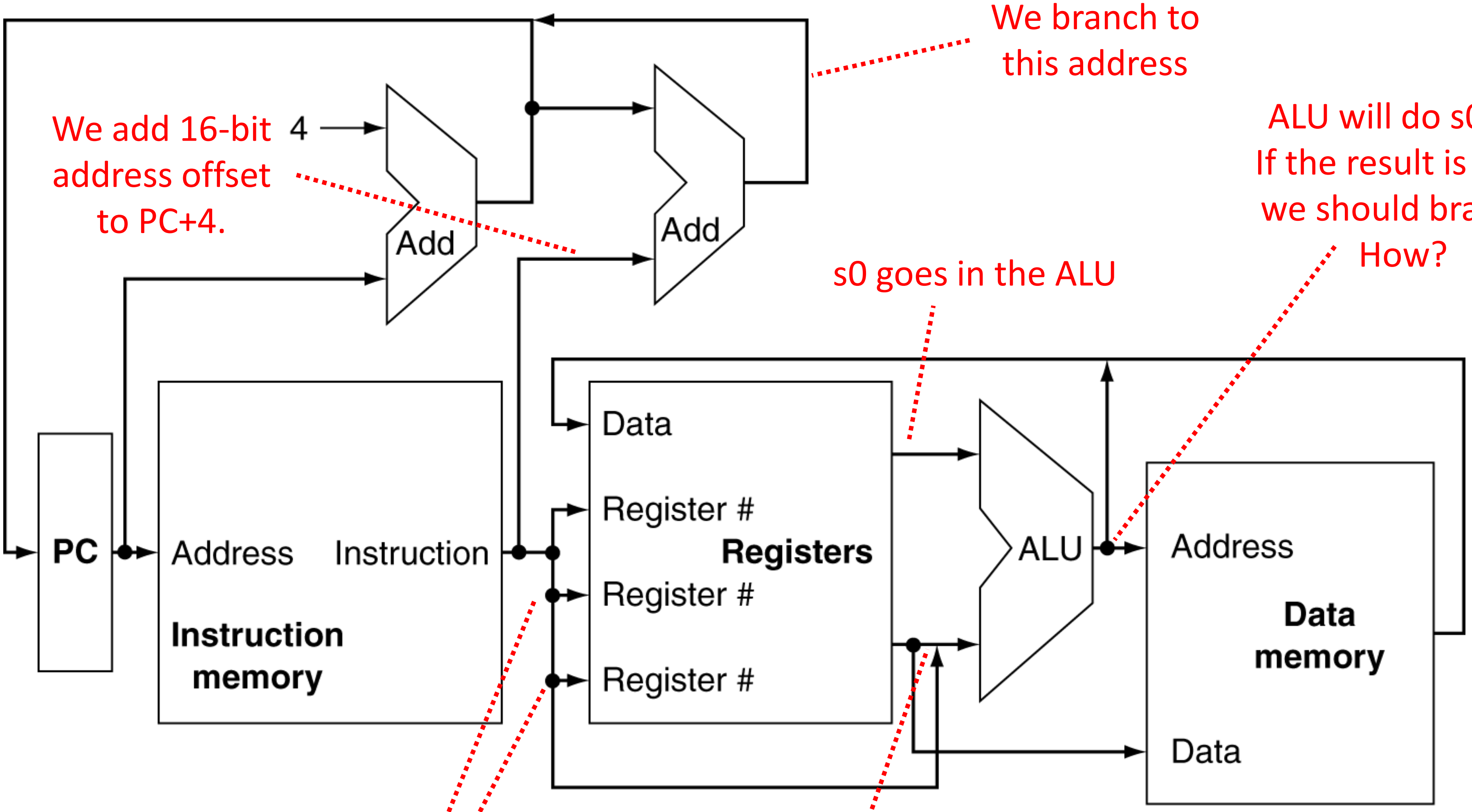
rs field (5 bits) is register s1
rt field (5 bits) is register t0

16-bit number in instruction; equal to 8

This is the word in t0.
We write it to the memory.

Branch Instruction

beq s0, s1, Loop	s0	s1	address offset
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits



We branch to this address

We add 16-bit address offset to PC+4.

ALU will do s0–s1 If the result is zero, we should branch. How?

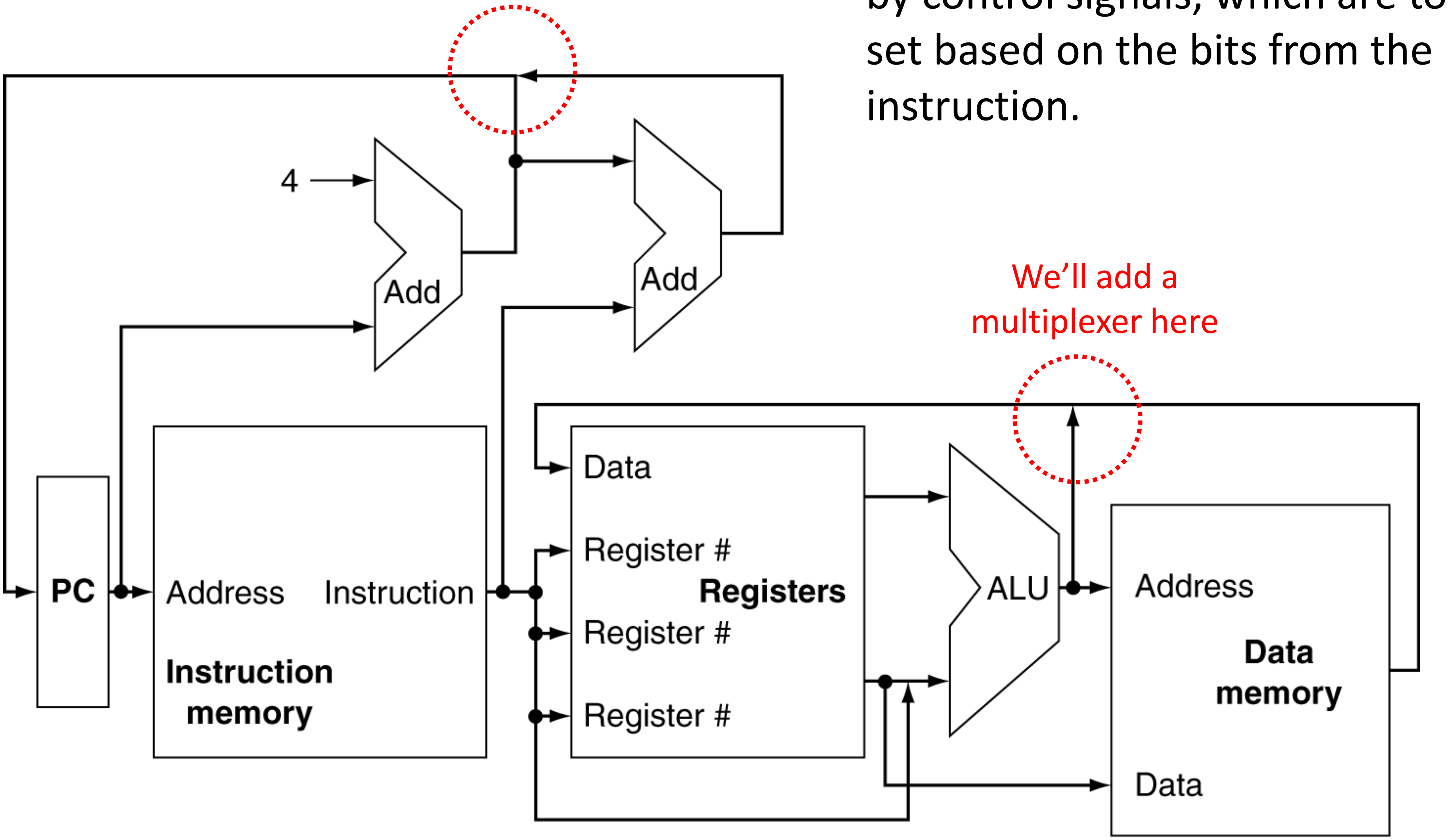
s0 goes in the ALU

rs field (5 bits) is register s0
rt field (5 bits) is register s1

s1 goes in the ALU

Need for Control Signals & Multiplexers

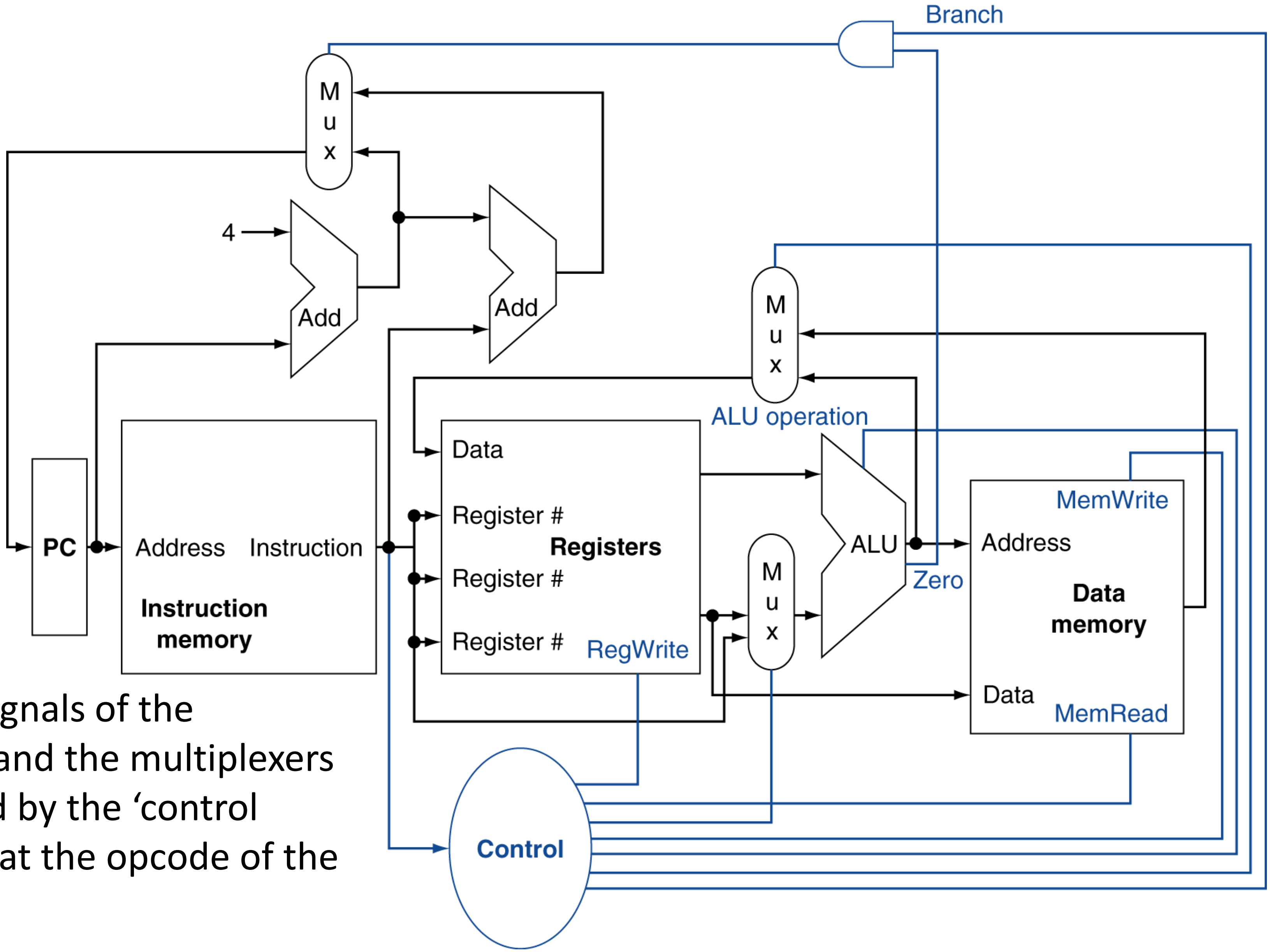
The multiplexers will be switched by control signals, which are to be set based on the bits from the instruction.



We'll add a multiplexer here

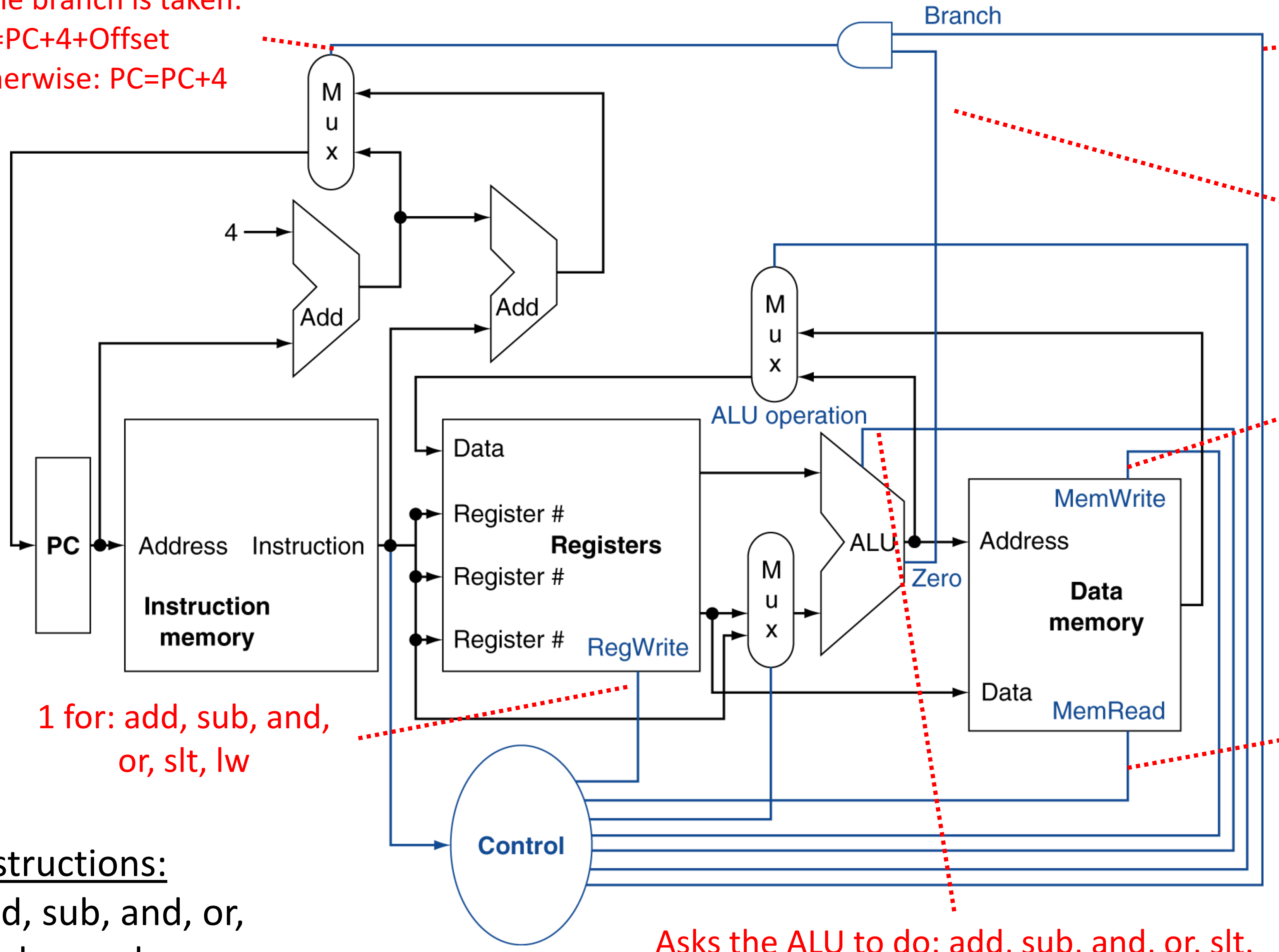
We'll add a multiplexer here

Adding Control & Multiplexers (MUXes)



The control signals of the components and the multiplexers are generated by the 'control unit'. It looks at the opcode of the instruction.

Datapath with Control & MUXes



If the branch is taken:
PC=PC+4+Offset
Otherwise: PC=PC+4

1 for beq

Zero signal is 1
ALU result is
(e.g., s0-s1) is 0

1 for sw only

1 for: add, sub, and,
or, slt, lw

1 for lw only

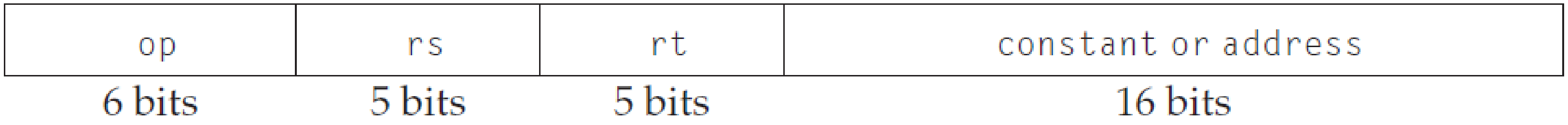
Asks the ALU to do: add, sub, and, or, slt.
Do add for sw, lw. Do sub for beq.

Instructions:

add, sub, and, or,
slt, lw, sw, beq

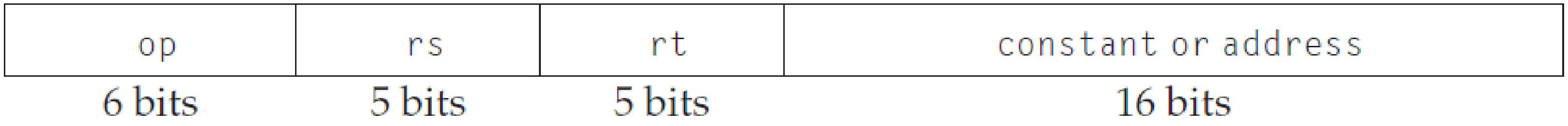
Sign Extender and Shifter

- For ‘lw’ and ‘sw’ the memory address is:
- Address = rs + (16-bit field sign extended to 32 bits)



op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

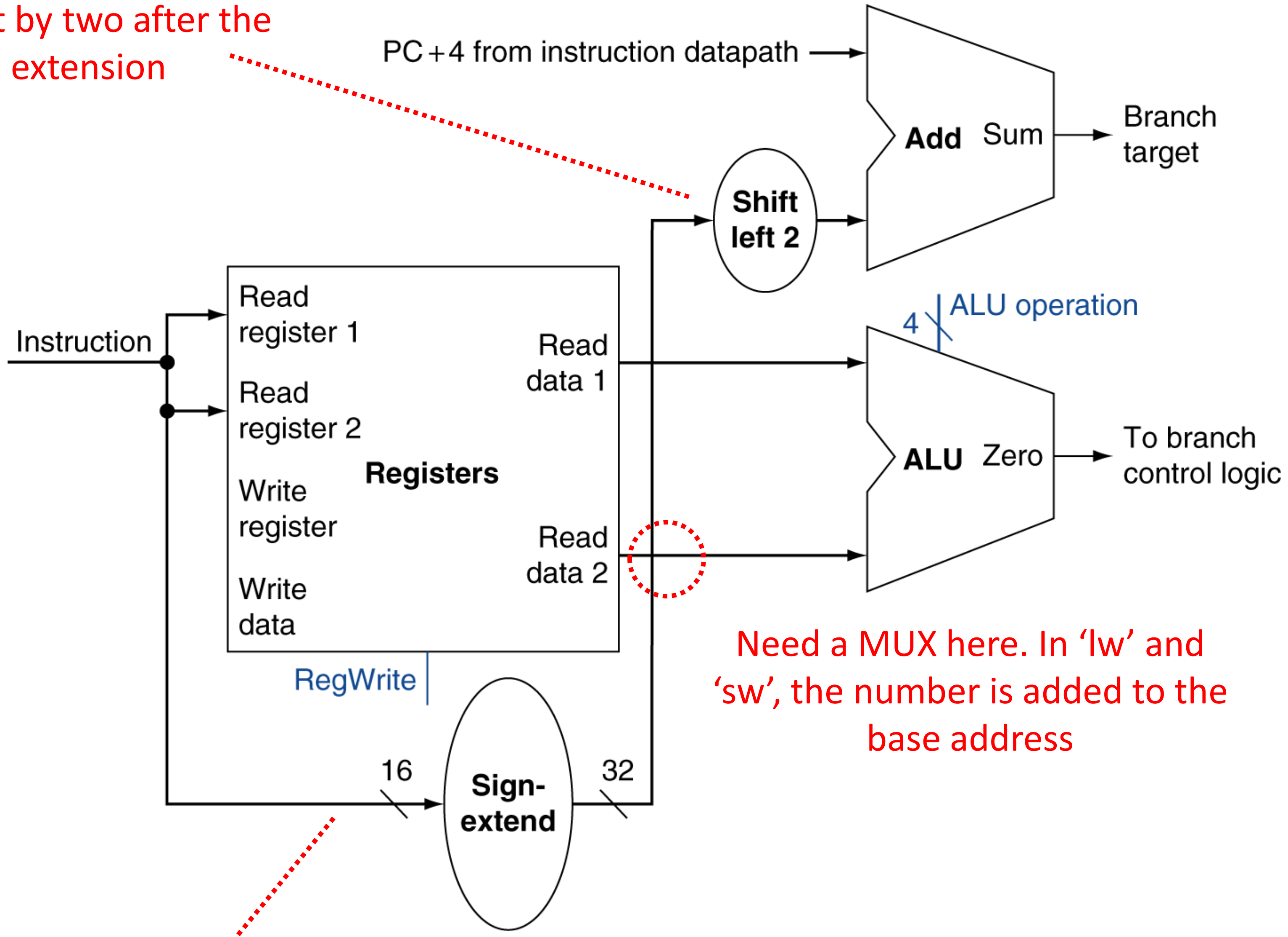
- For ‘beq’, the branch address is:
- Address = PC + 4 + (16-bit field sign extended to 32 bits then shifted left by 2)



op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Adding Sign Extender and Shifter

In beq, the address offset is shifted left by two after the sign extension

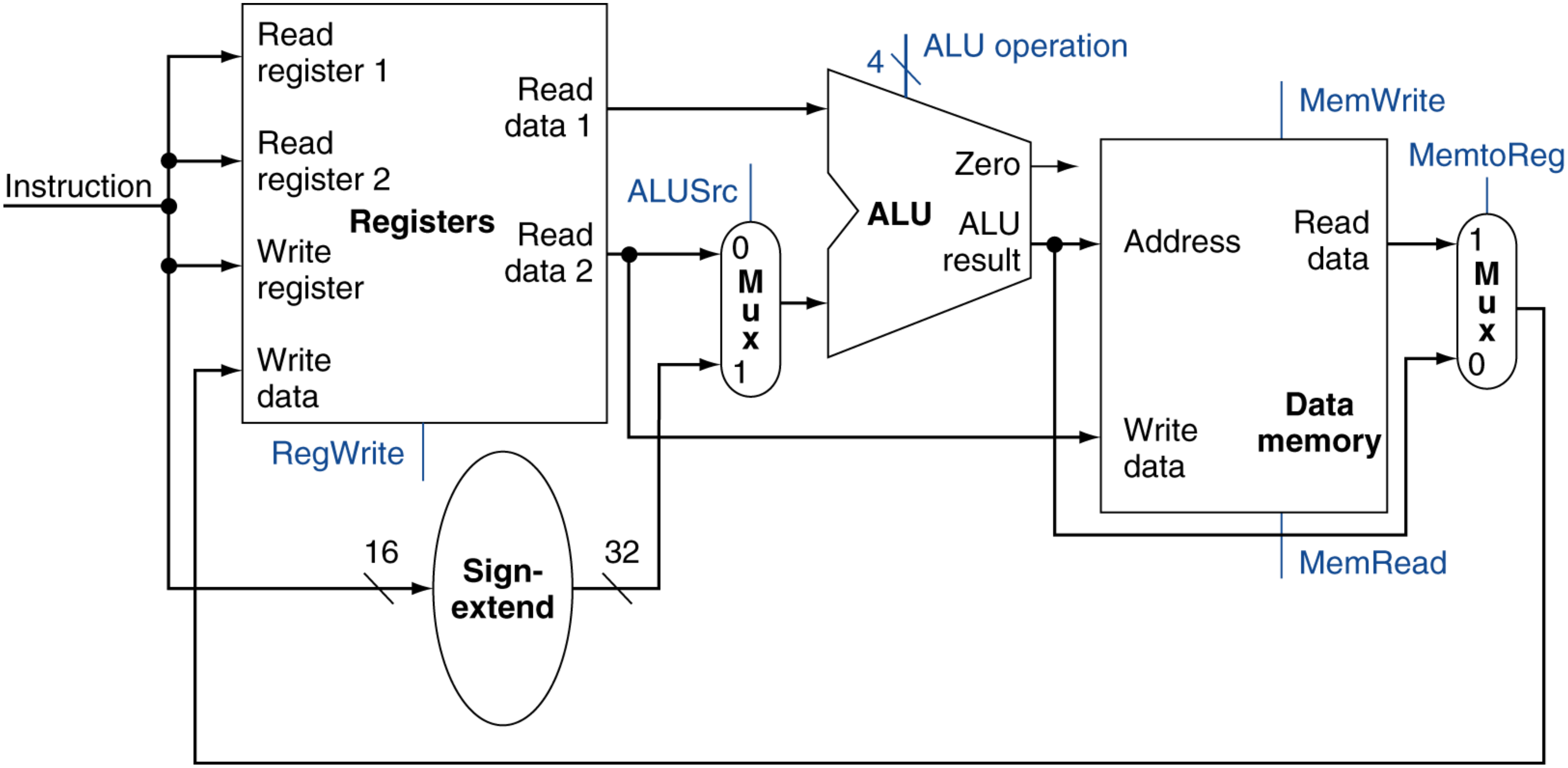


Need a MUX here. In ‘lw’ and ‘sw’, the number is added to the base address

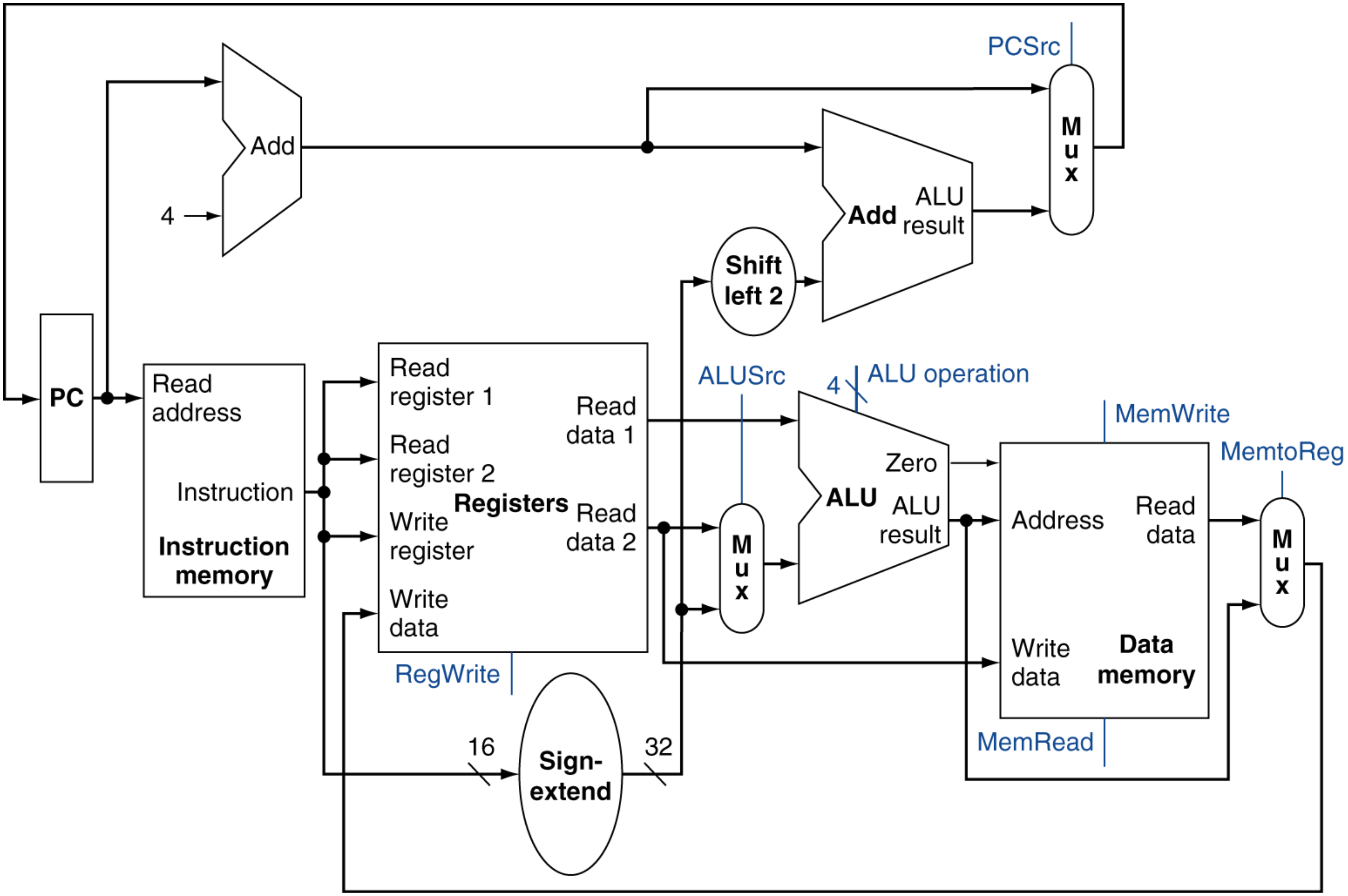
16-bit number from the instruction

Control Signals

RegWrite:	Register Write
ALUSrc:	ALU Source
ALU operation	
MemWrite:	Memory Write
MemRead:	Memory Read
MemtoReg:	Memory to Register



Datapath with Control Signals



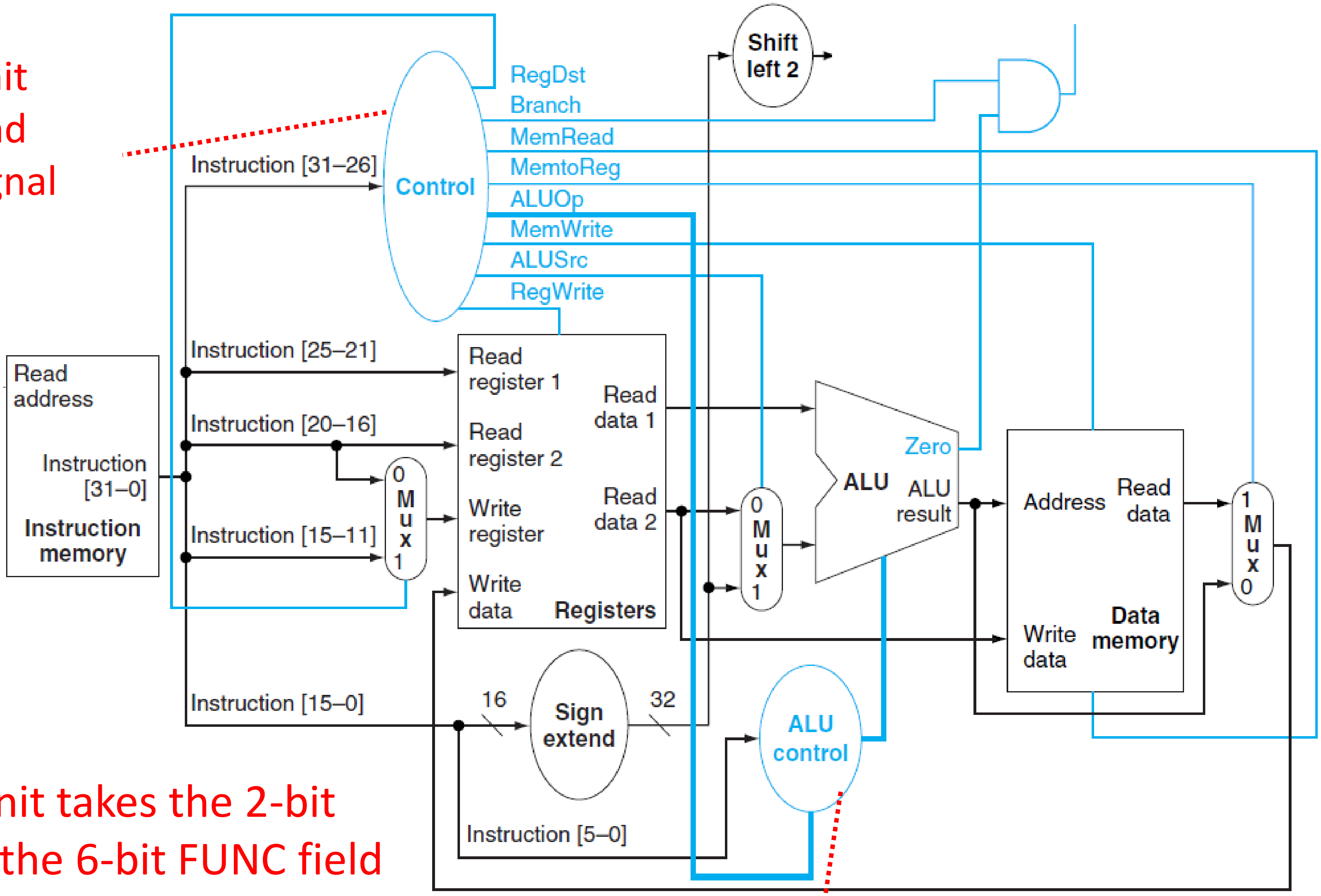
ALU Operations

- The ALU takes a 4-bit ‘ALU Operation’ field
- This 4-bit signal can designate 16 unique operations in the ALU

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

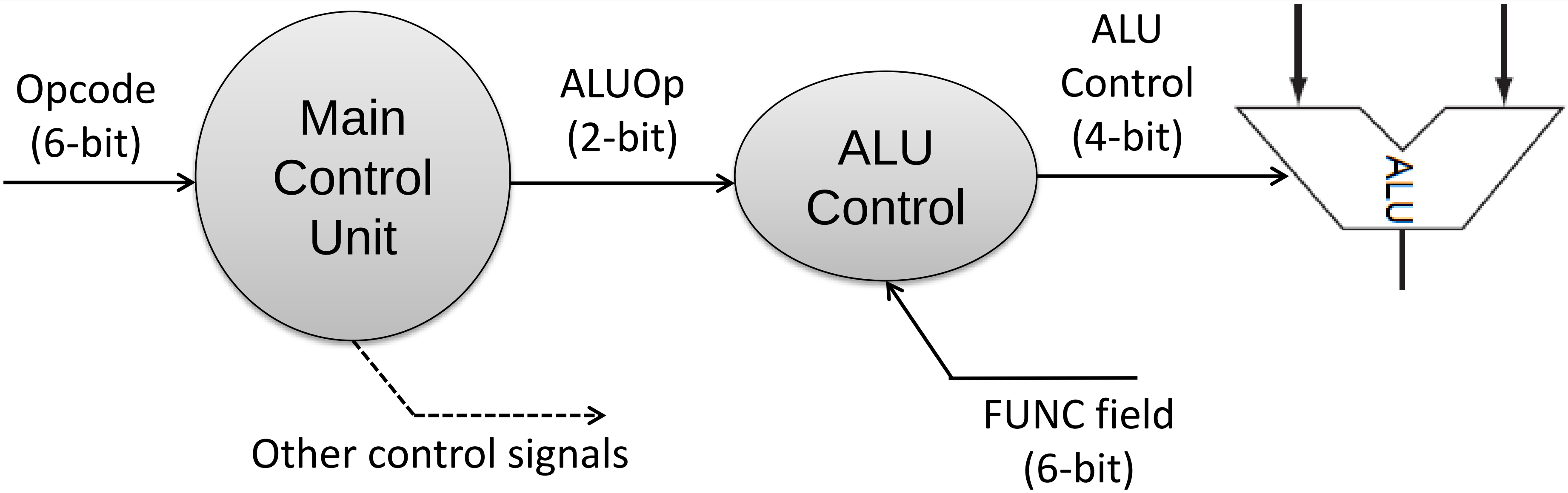
Two Levels of Control

The main control unit takes the opcode and generates a 2-bit signal called ‘ALUOp’



The ALU control unit takes the 2-bit ALUOp signal and the 6-bit FUNC field and generates the 4-bit signal to the ALU

ALU Control



Main Control Unit

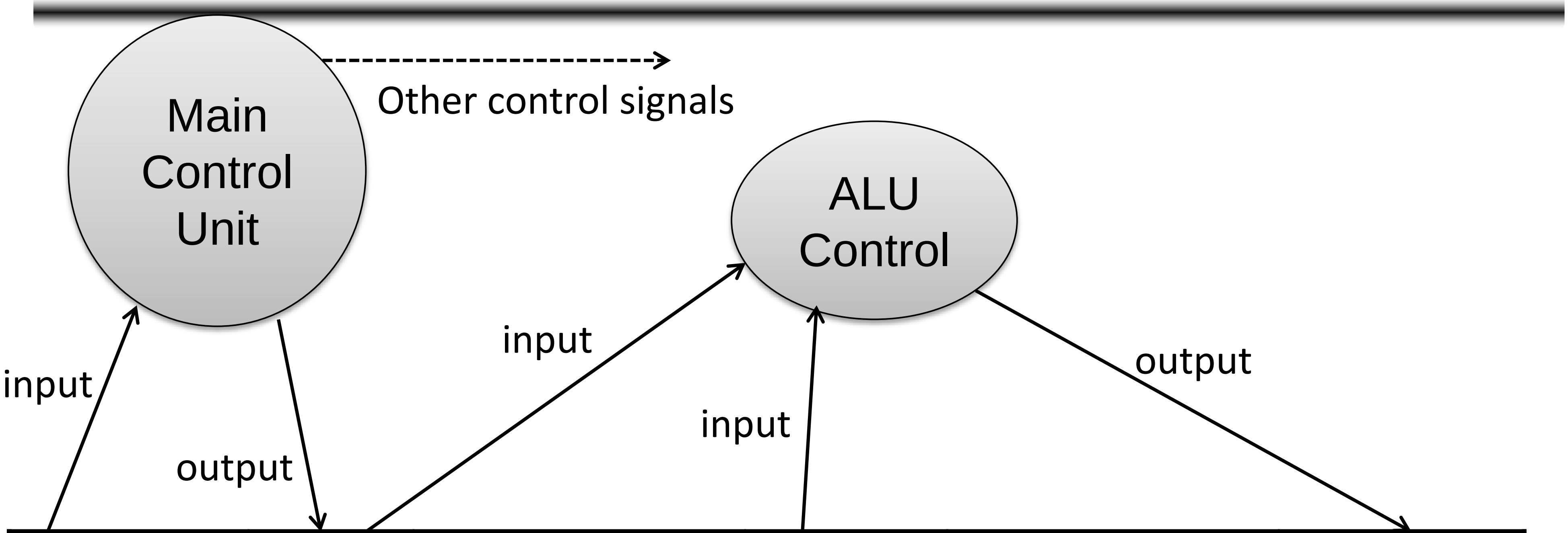
- ➢Takes the opcode
- ➢ALUOp=00 is for Addition (used for “lw”, “sw”)
- ➢ALUOp=01 is for Subtraction (used for “beq”)
- ➢ALUOp=10 is for R-type (we need the FUNC field to decide the operation)

ALU Control Unit

- ➢ Takes ALUOp and FUNC field
- ➢ ALUOp=00, do addition (output code: 0010)
- ➢ ALUOp=01, do subtraction (output code: 0110)
- ➢ ALUOp=10, look at FUNC field to decide

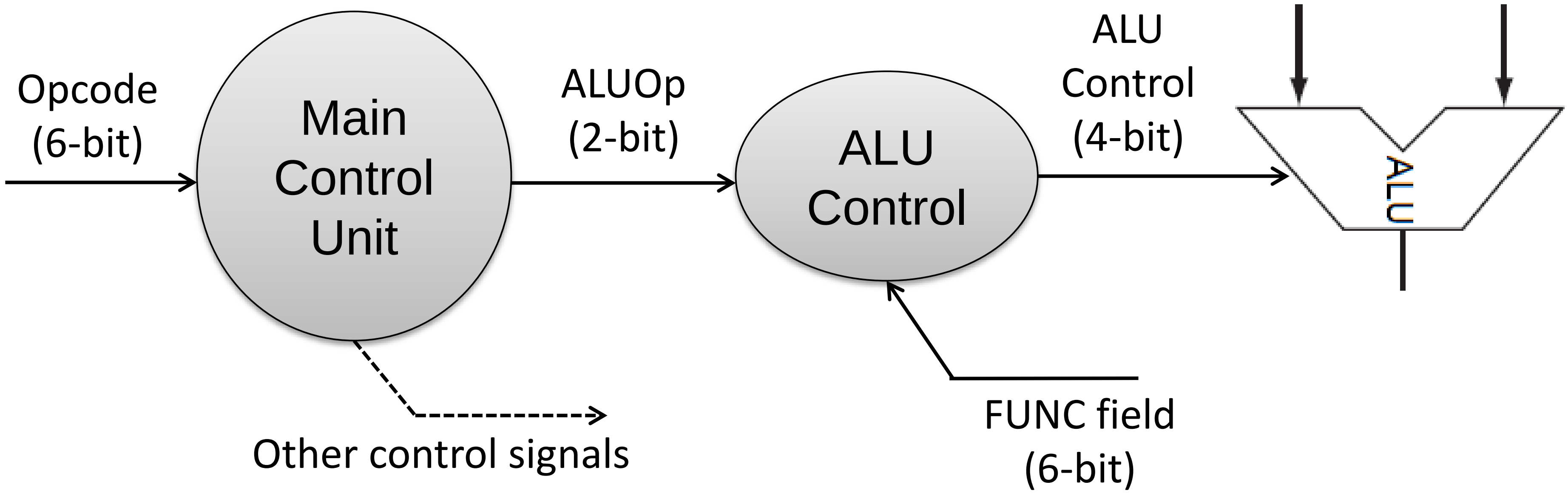
Our instructions: add, sub, and, or, slt, lw, sw, beq, j

ALU Control



opcode	ALUOp	Instruction	funct	ALU function	ALU control
100011 (35)	00	load word	XXXXXX	add	0010
101011 (43)	00	store word	XXXXXX	add	0010
000100 (4)	01	branch equal	XXXXXX	subtract	0110
000000 (0)	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

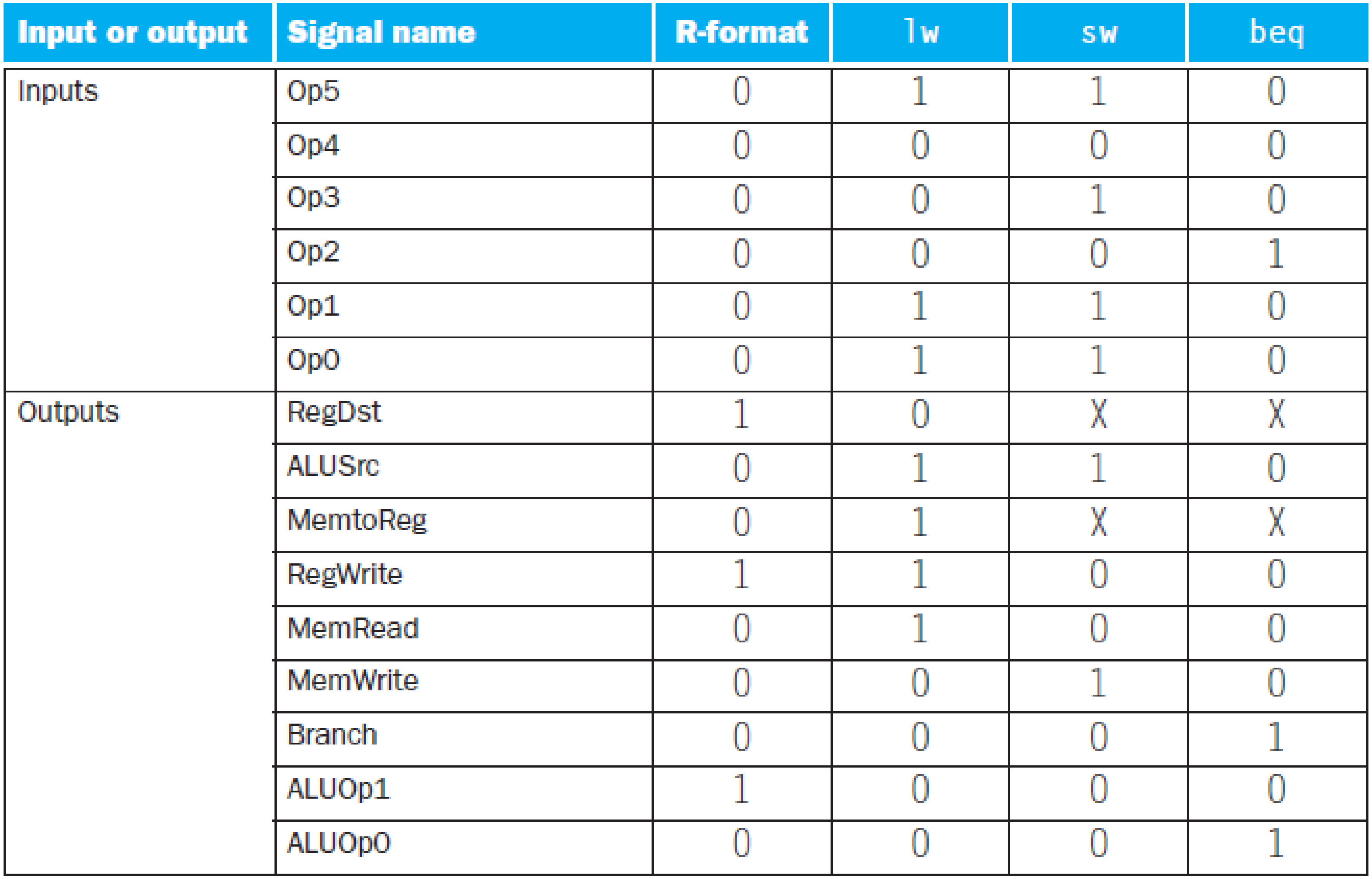
Benefit of Two Levels of Control



- If we add a new R-type instruction to the datapath
  - No change is needed in the main control
  - The main control unit will give out ’10’ (this corresponds for R-type)
  - In this case the ALU control looks at the FUNC field
  - The new R-type instruction is assigned a unique FUNC value

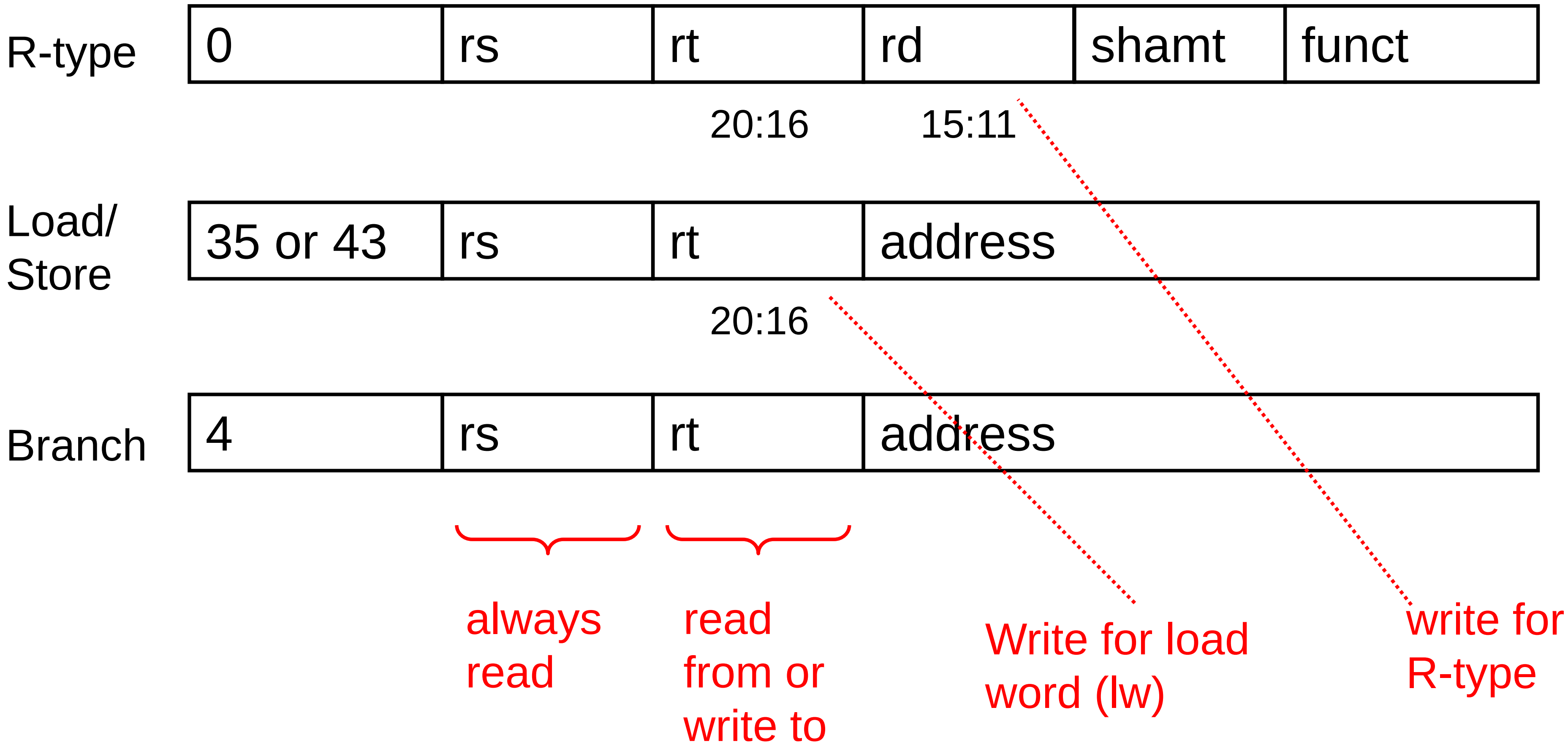
Main Control Unit

Truth table



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

Where to Write?: Destination Register



R-type	0	rs	rt	rd	shamt	funct
			20:16	15:11		
Load/ Store	35 or 43	rs	rt	address		
			20:16			
Branch	4	rs	rt	address		

always read

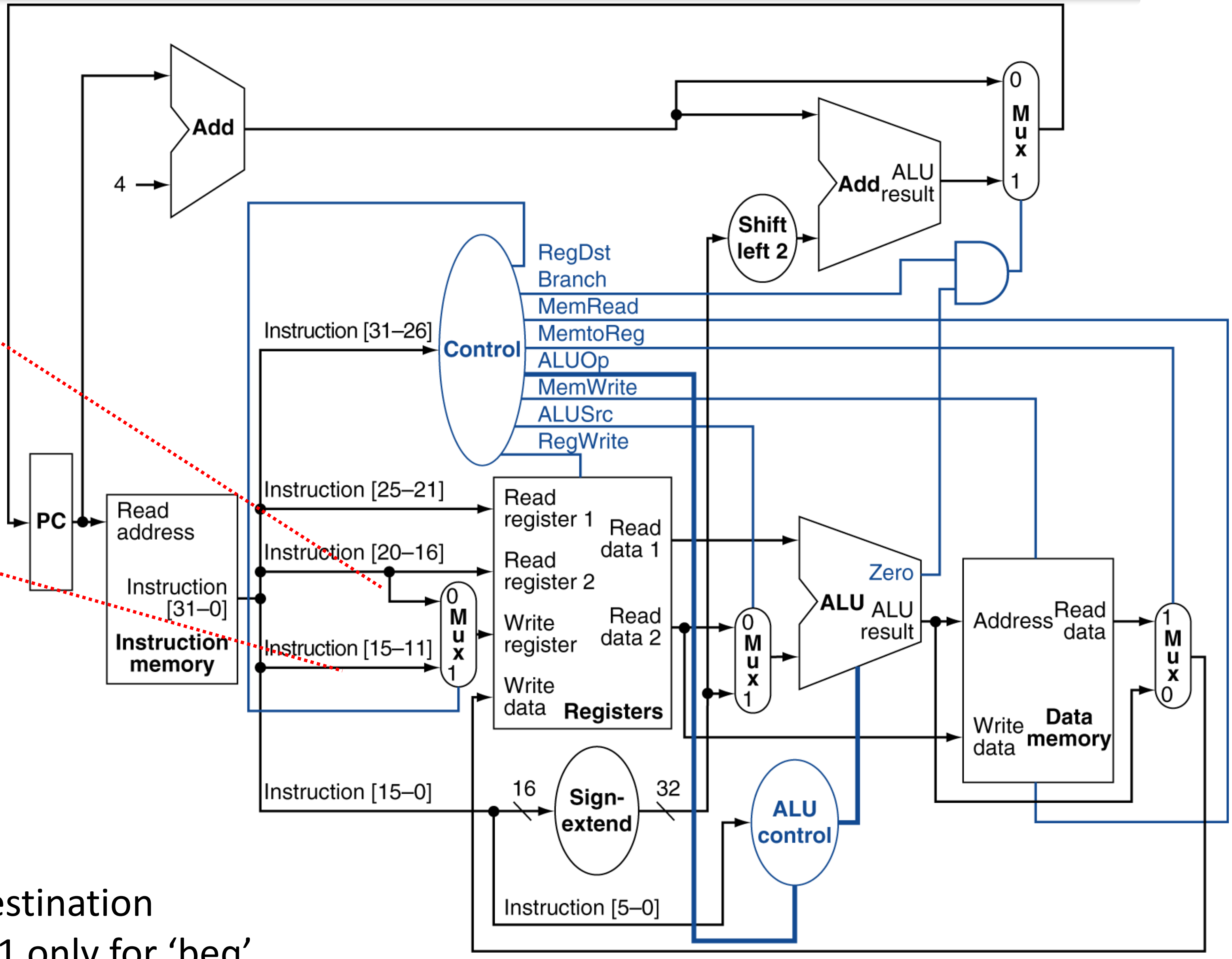
read from or write to

Write for load word (lw)

write for R-type

- ➢In R-type, we read ‘rs’ and ‘rt’ and write to ‘rd’
- ➢In load word (lw), we read ‘rs’ and write to ‘rt’
- ➢In store word (sw), we read ‘rs’ and ‘rt’

Datapath with Destination Control



For load word (lw)

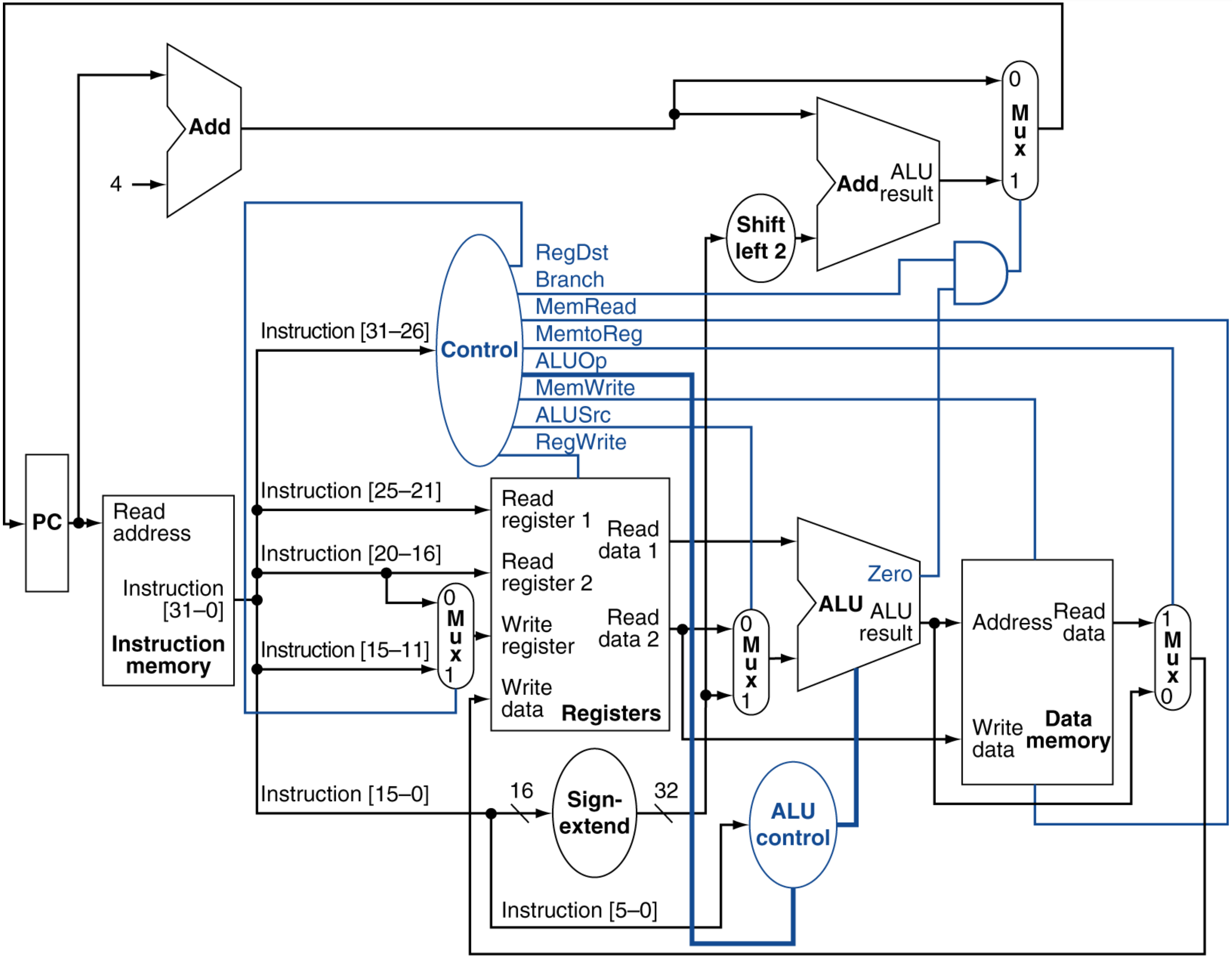
For R-type

New control signals

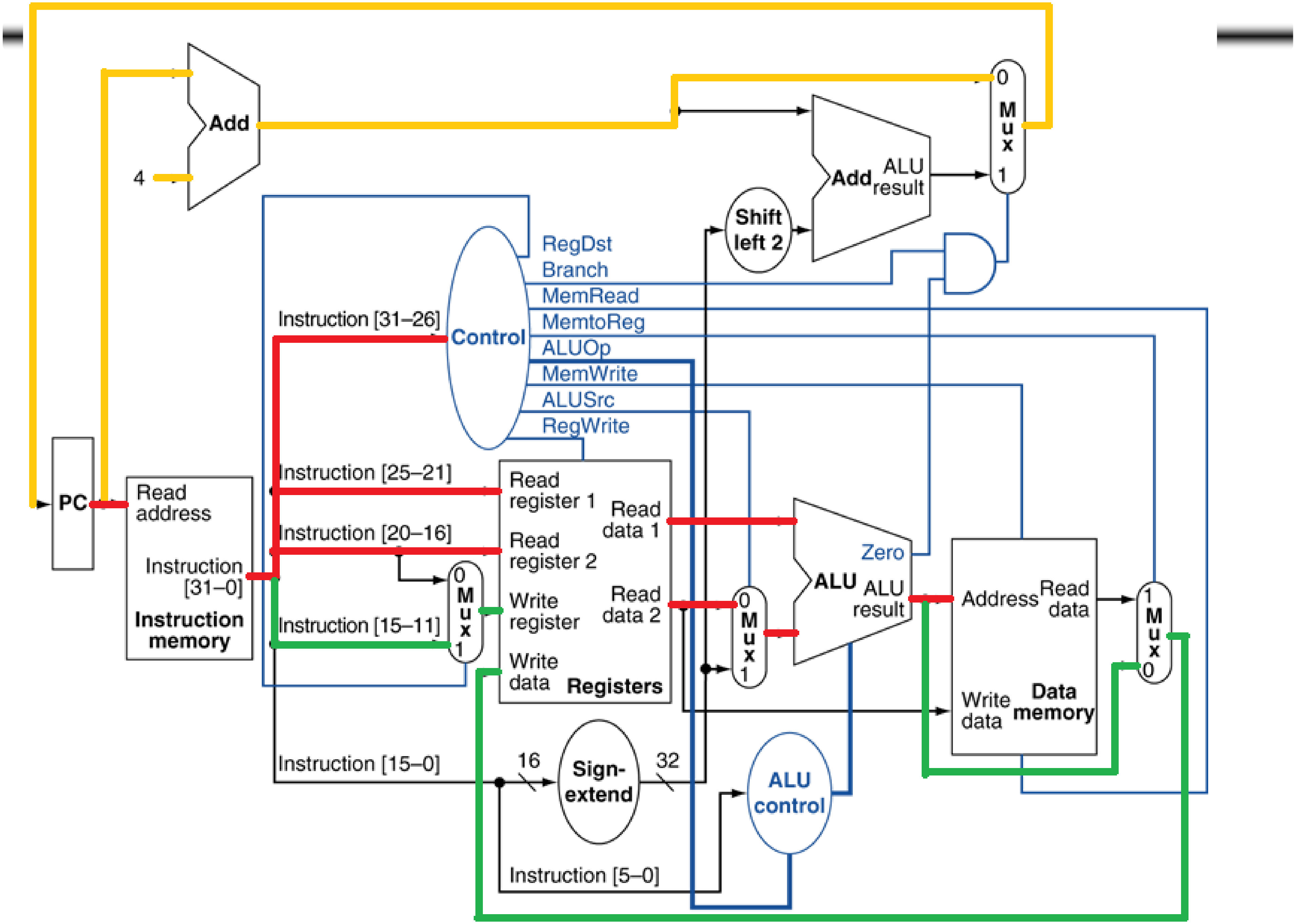
RegDst: Register Destination

Branch: is equal to 1 only for ‘beq’

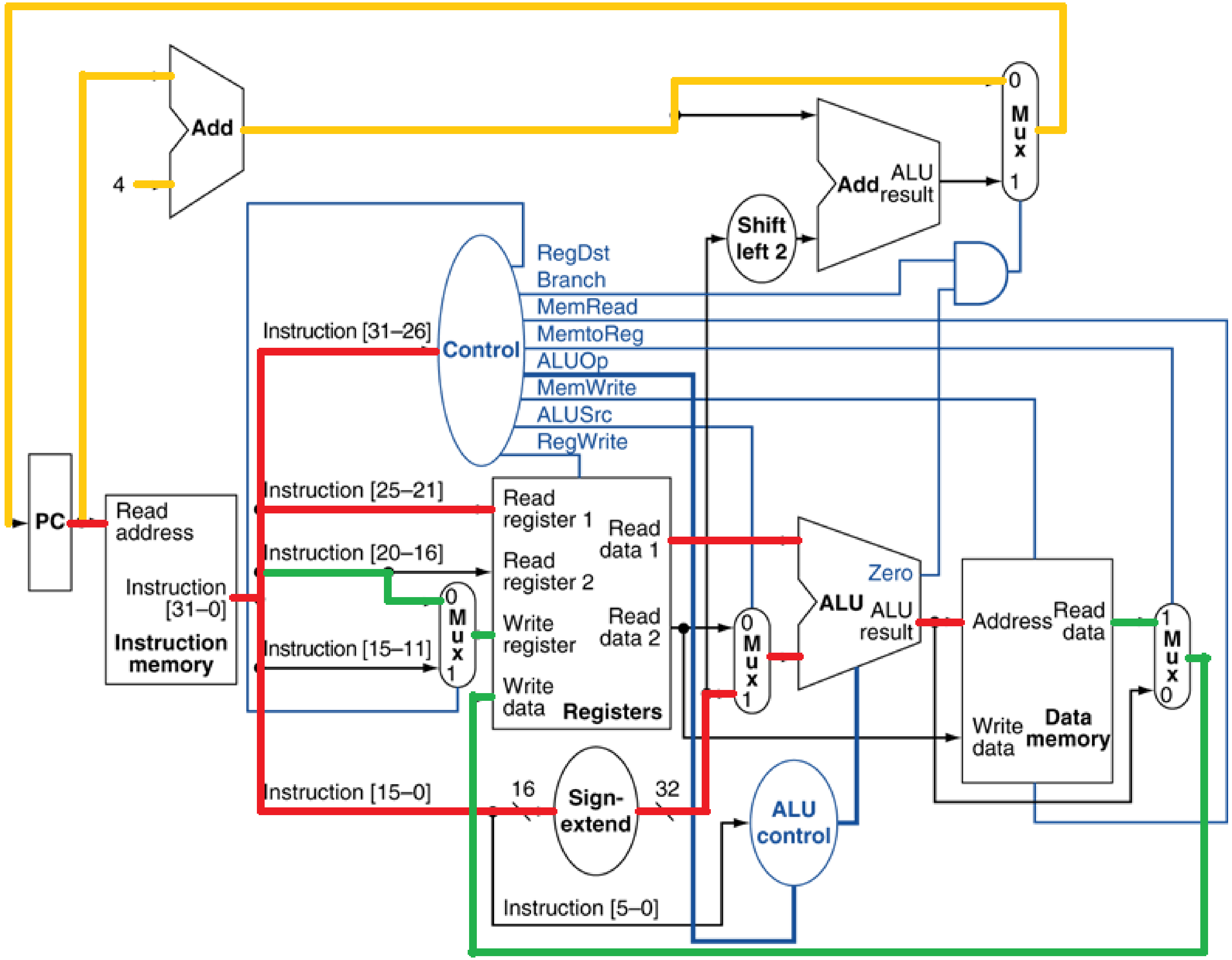
Datapath with Control



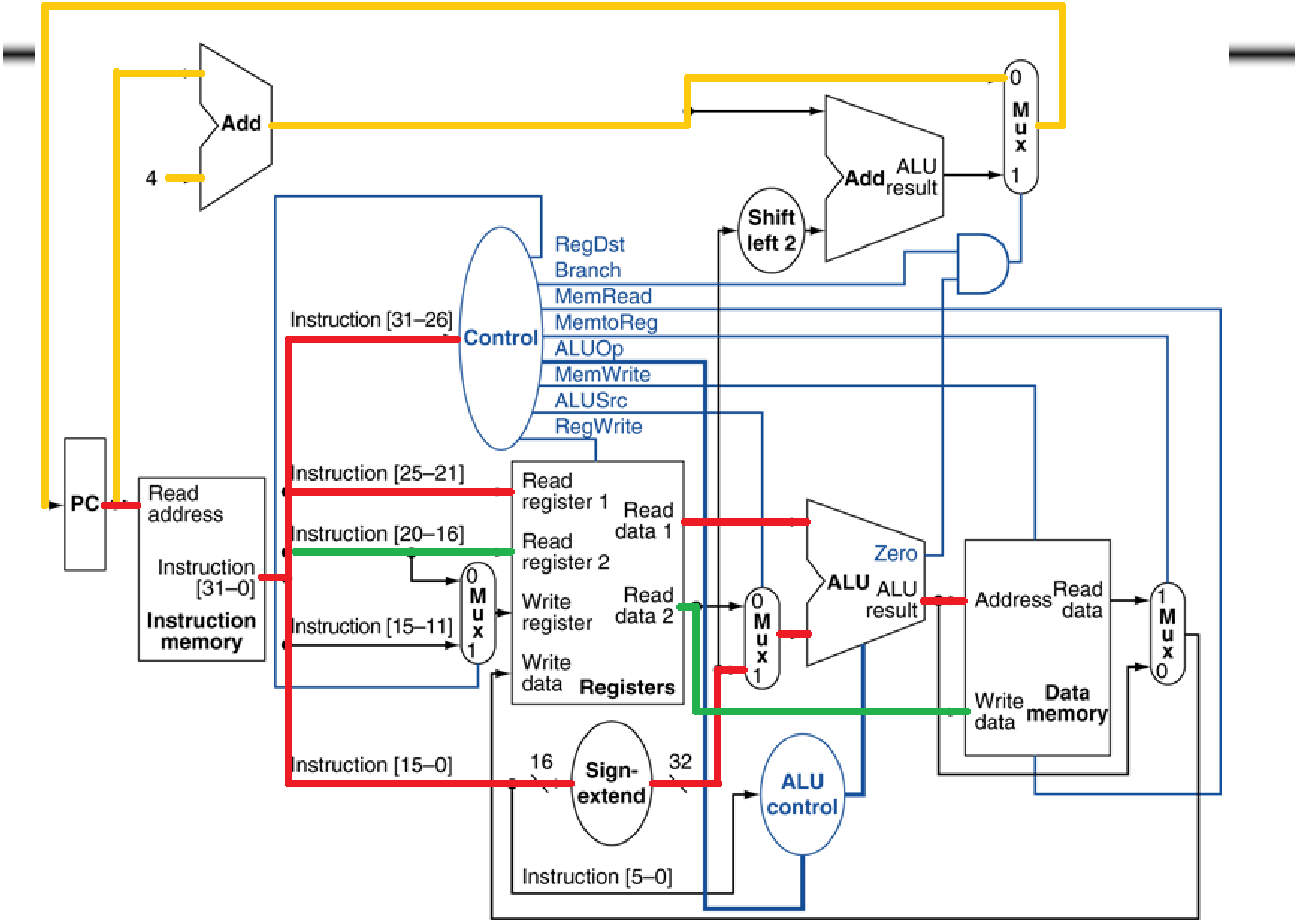
Path for R-Type



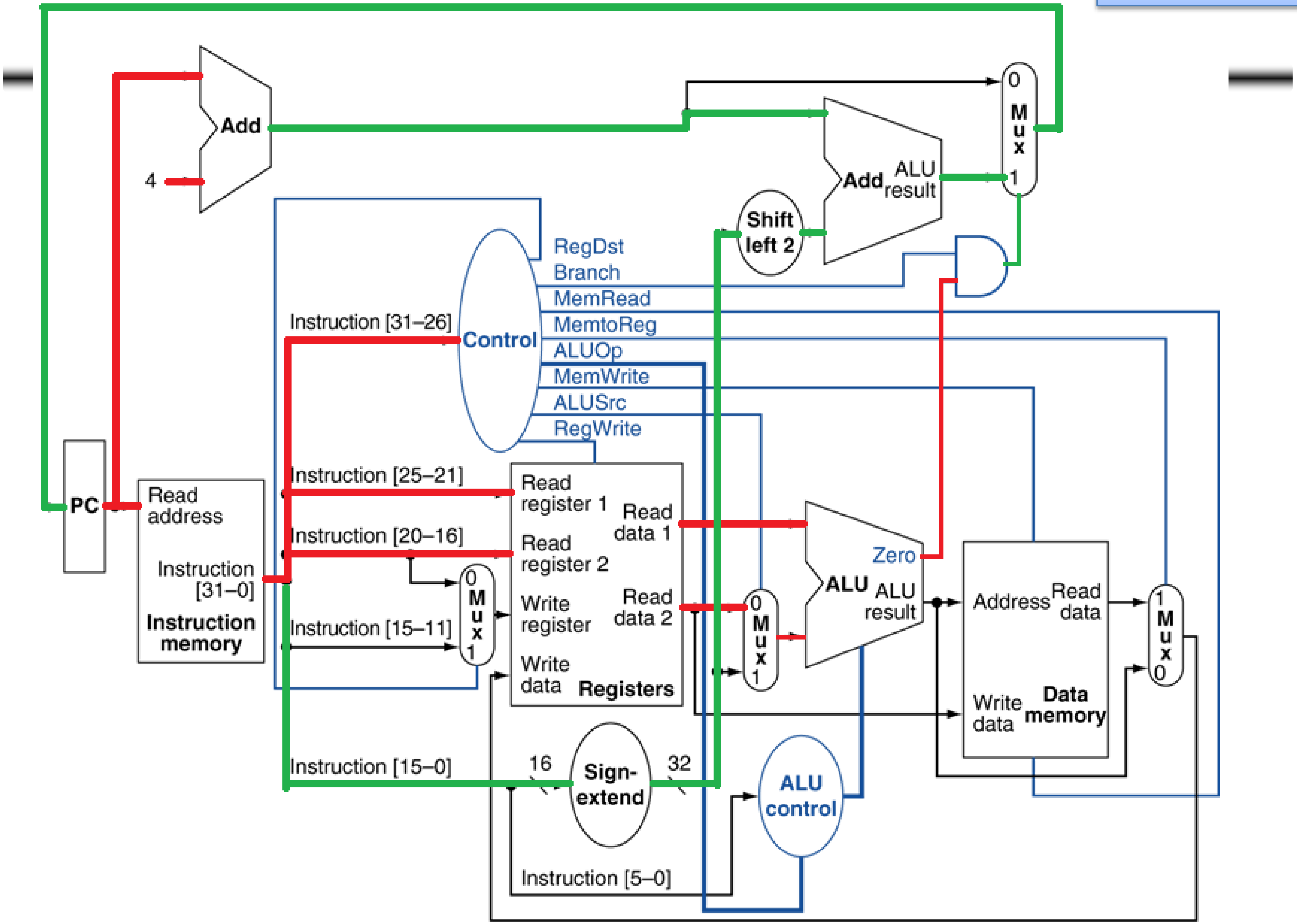
Path for Load Word



Path for Store Word

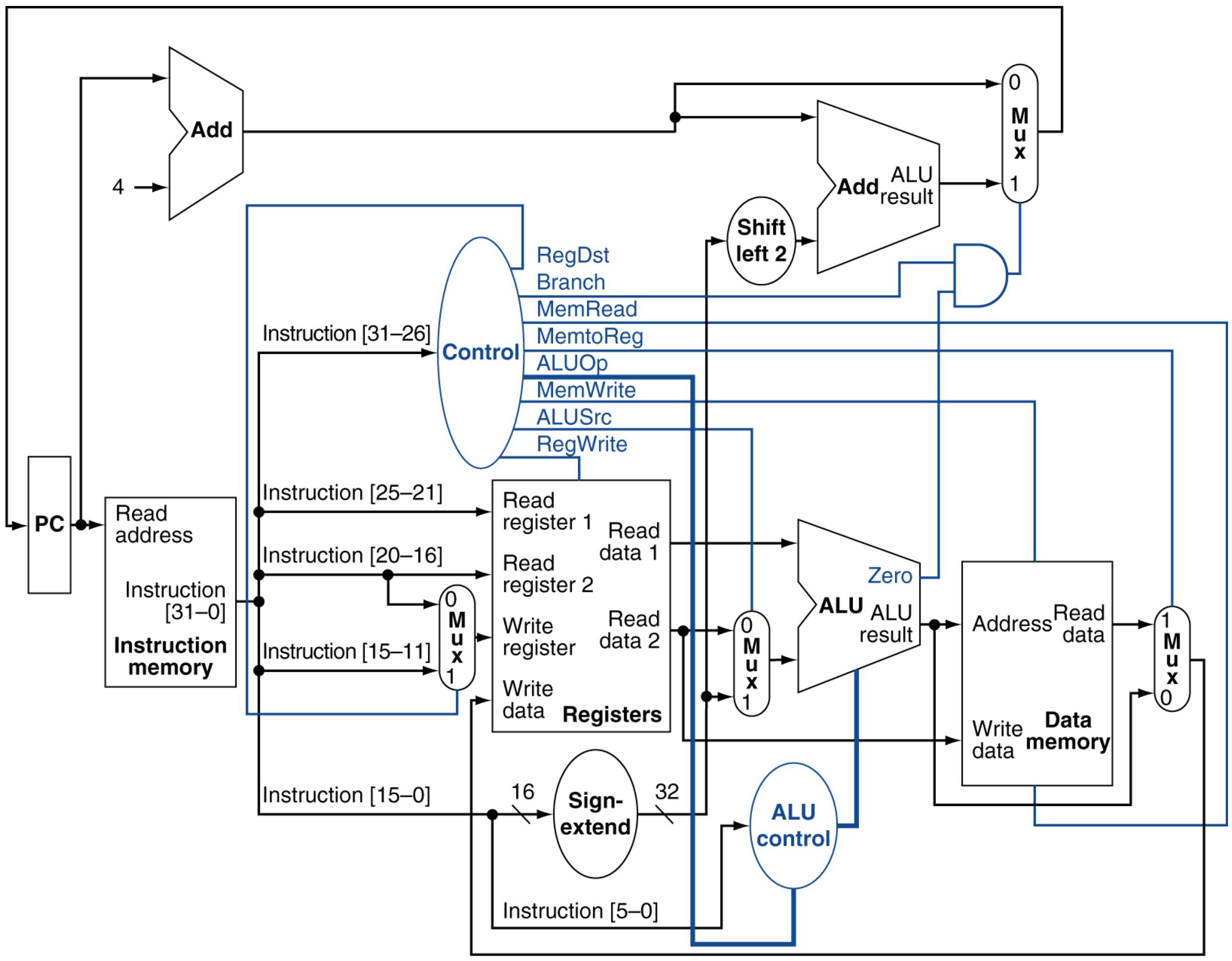


Branch (beq)


Add
4
Add
ALU
result
Shift
left 2
0
M
u
x
1
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
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]

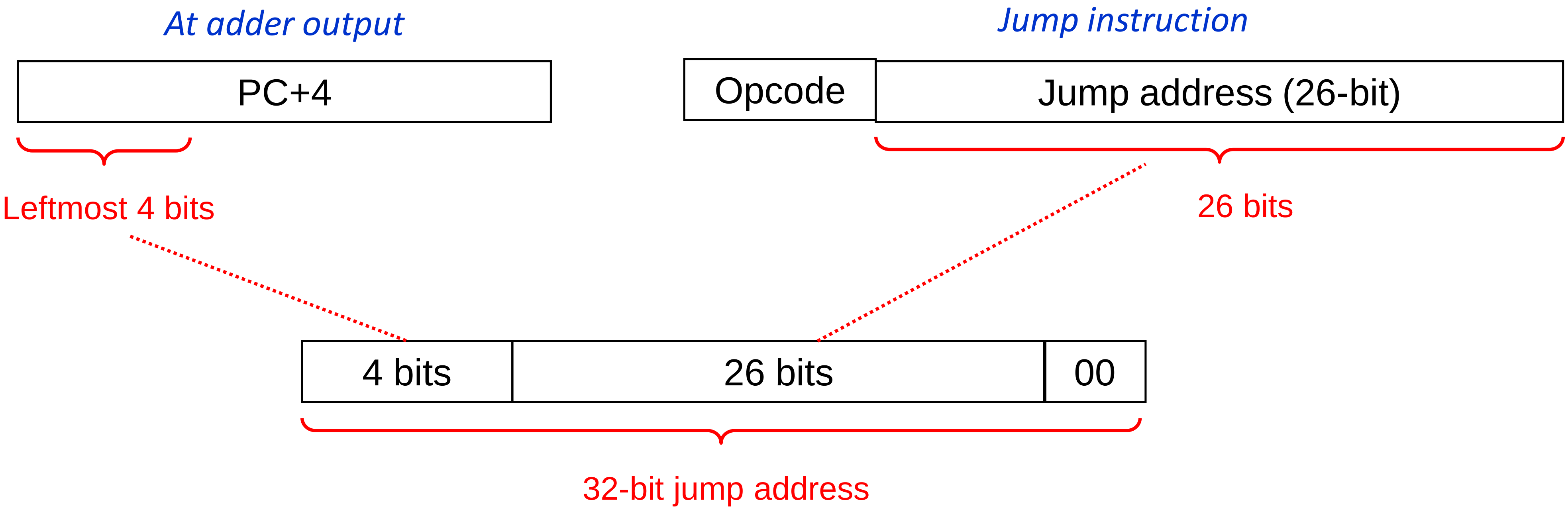
Control Signal Values

	R-type	lw	sw	beq
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



Supporting the Jump Instruction

- The datapath we have doesn’t support the ‘j’ instruction yet
- The jump address is computed as shown below
- We’ll add this mechanism in the datapath

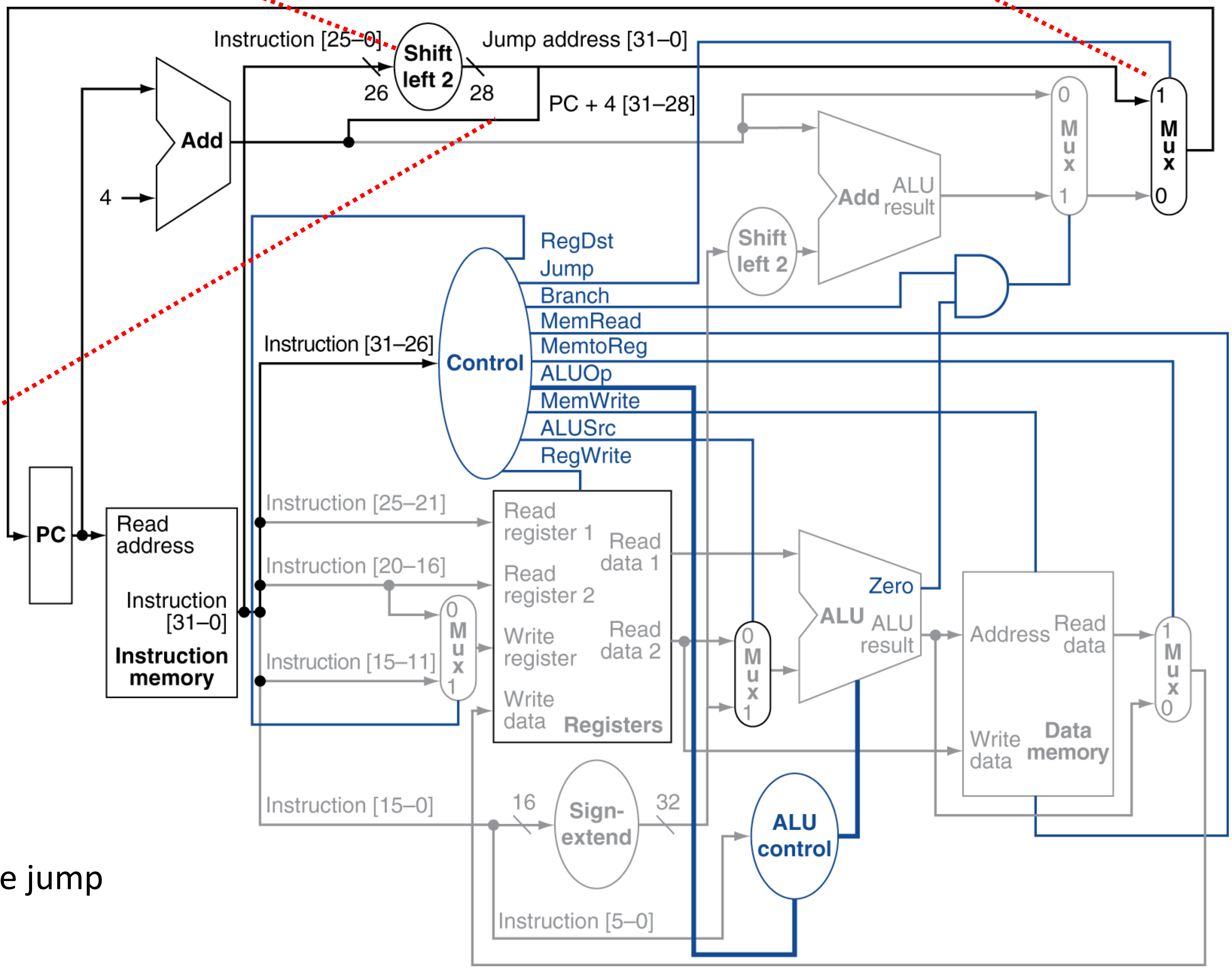


Supporting the Jump Instruction

The 26-bit field from the jump instruction is shifted left by 2 bits

We add a new MUX. We could have made the other MUX a 4-to-1, but we would have to change the other control signals.

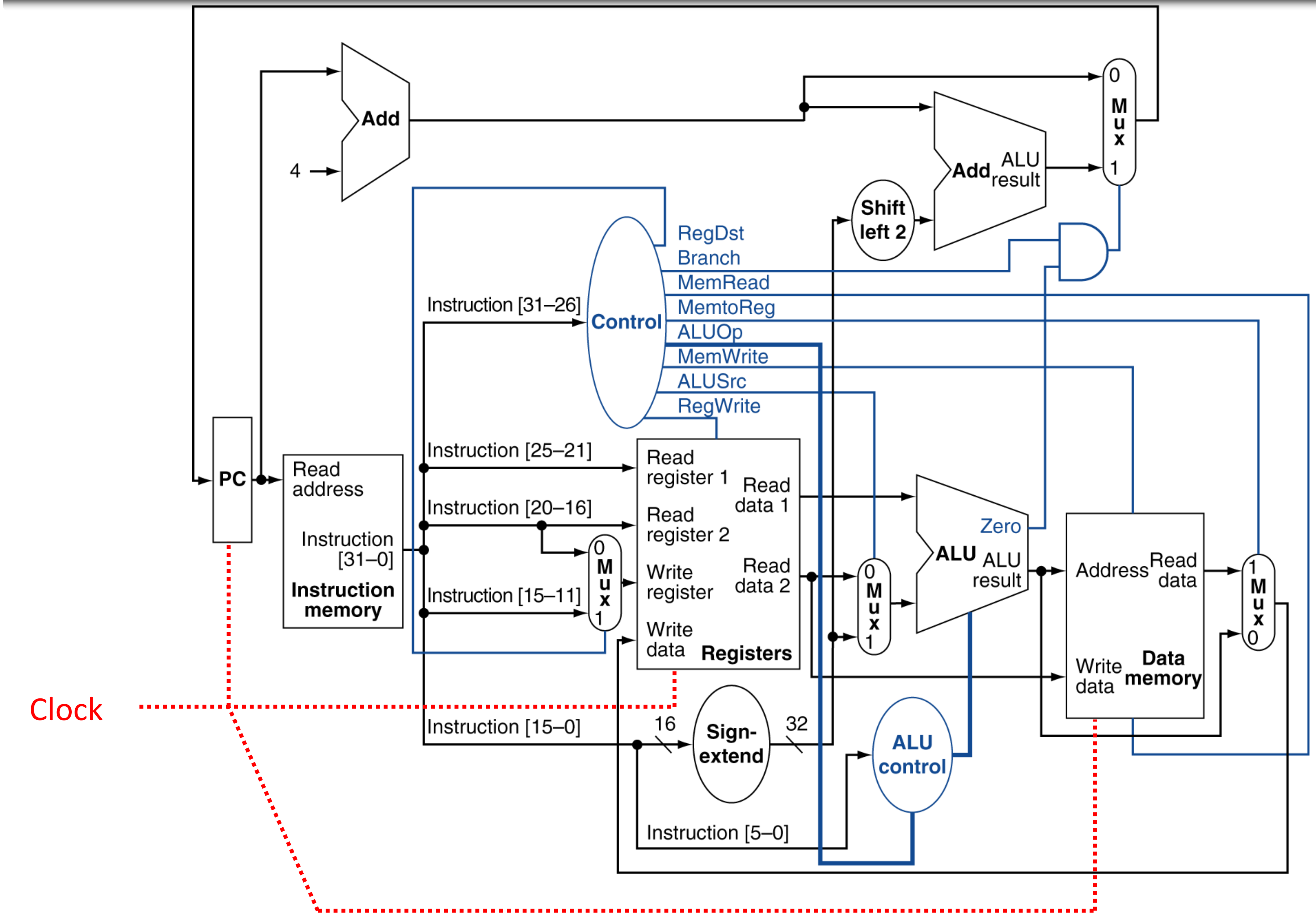
Take leftmost 4 bits from PC+4 at adder output



Jump: 1 only for the jump instruction

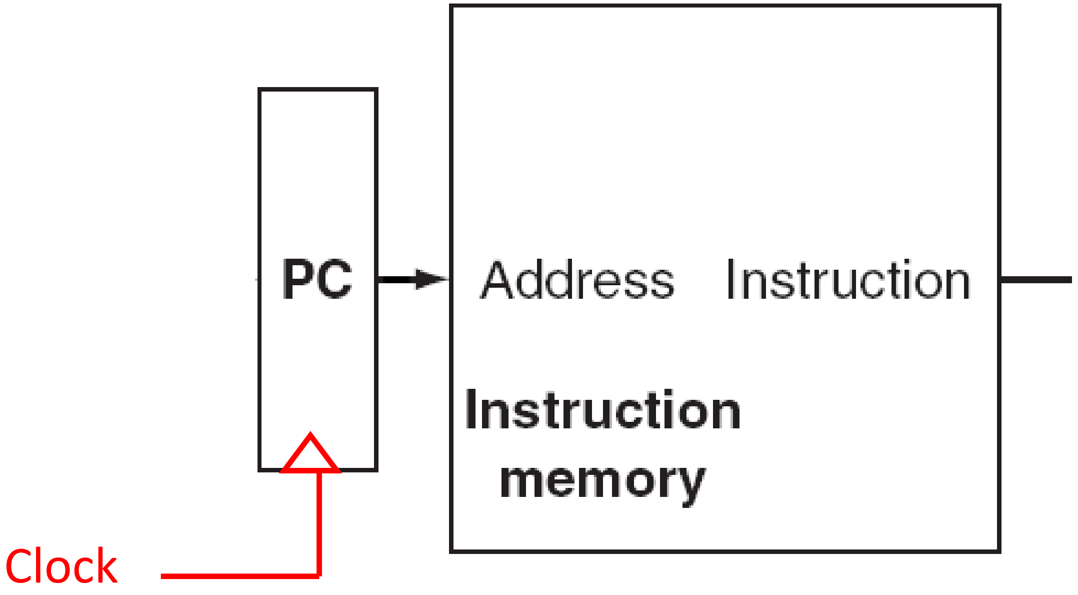
Clock in the Datapath

The clock signal is connected to the PC register, the register file and the data memory.



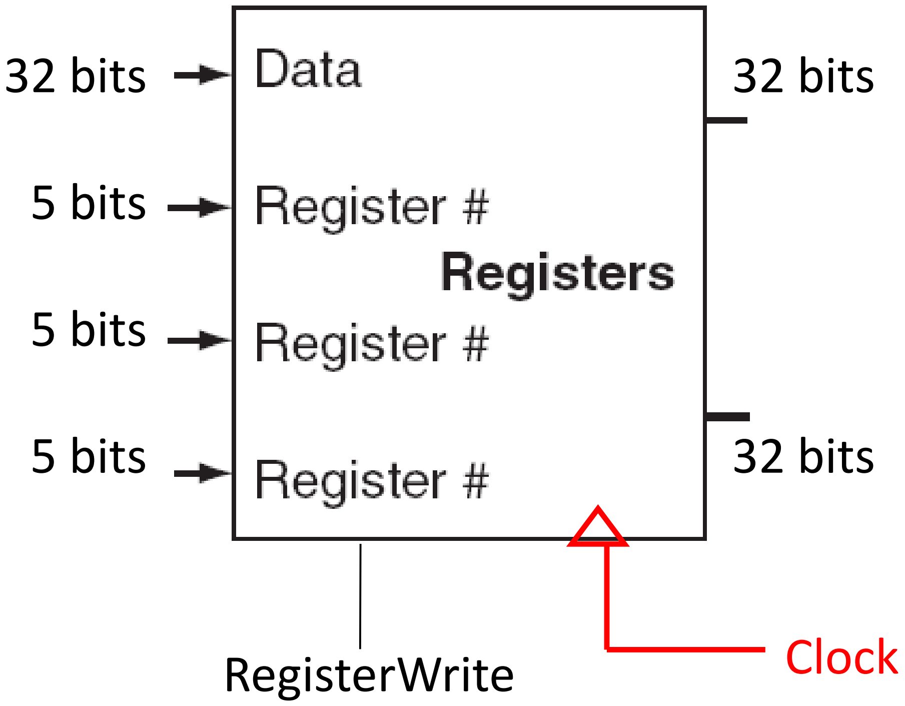
PC Register

- We'll use the positive edge of the clock
- The PC register is updated at the positive edge of the clock
- When the PC is updated a new instruction is read from the memory



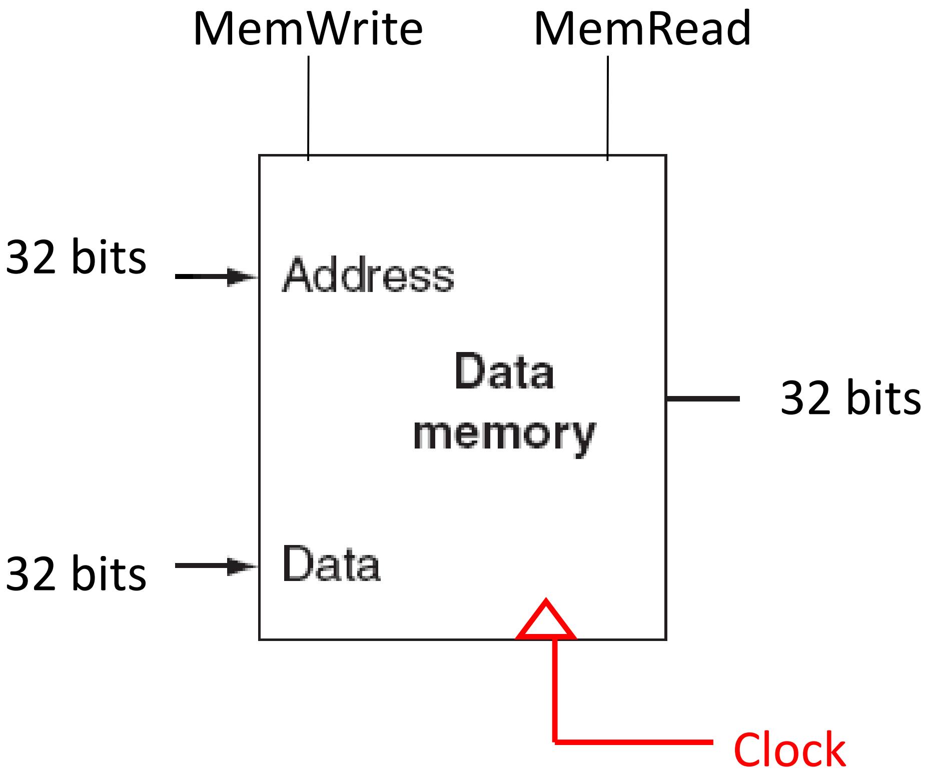
Register File

- The ‘read’ operation is not synchronized with the clock
  - When we set the two 5-bit read addresses, the two register values appear at the output immediately (not sync’d with the clock)
- The ‘write’ operation is synchronized with the clock
  - We set the 5-bit write address and the 32-bit data
  - The data is written in the register at the positive edge of the clock (on condition that RegWrite=1)



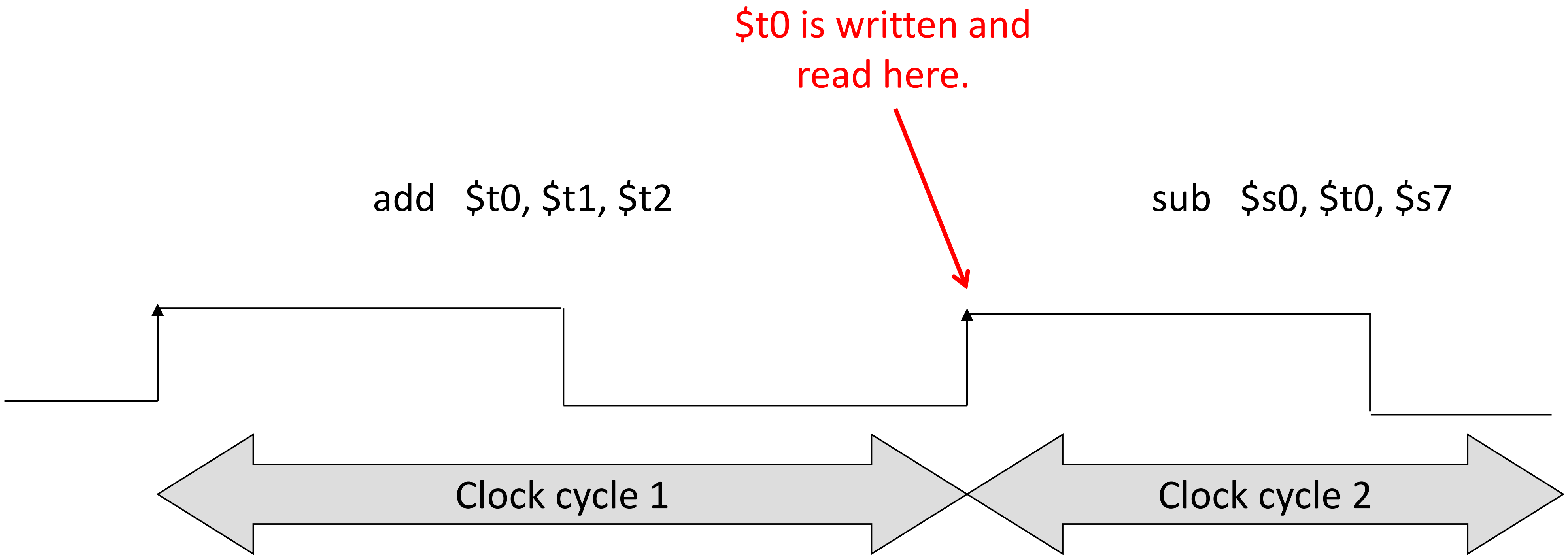
Data Memory

- The ‘read’ operation is not synchronized with the clock
  - Once we set the address, the data shows at the output, after the memory’s delay (provided MemRead=1)
- The ‘write’ operation is synchronized with the clock
  - We set the 32-bit address and the data, the data is written in the memory at the positive edge of the clock (provided MemWrite=1)



Writing and Reading the Same Register

- The ‘add’ instruction writes to $t0 at the positive edge between cycle 1 and cycle 2
- The ‘sub’ instruction starts reading $t0 at the same positive edge
- Does this lead to a racing condition?
- Usually in logic circuit, we shouldn’t read and write a data at the same time



$t0 is written and read here.

add $t0, $t1, $t2

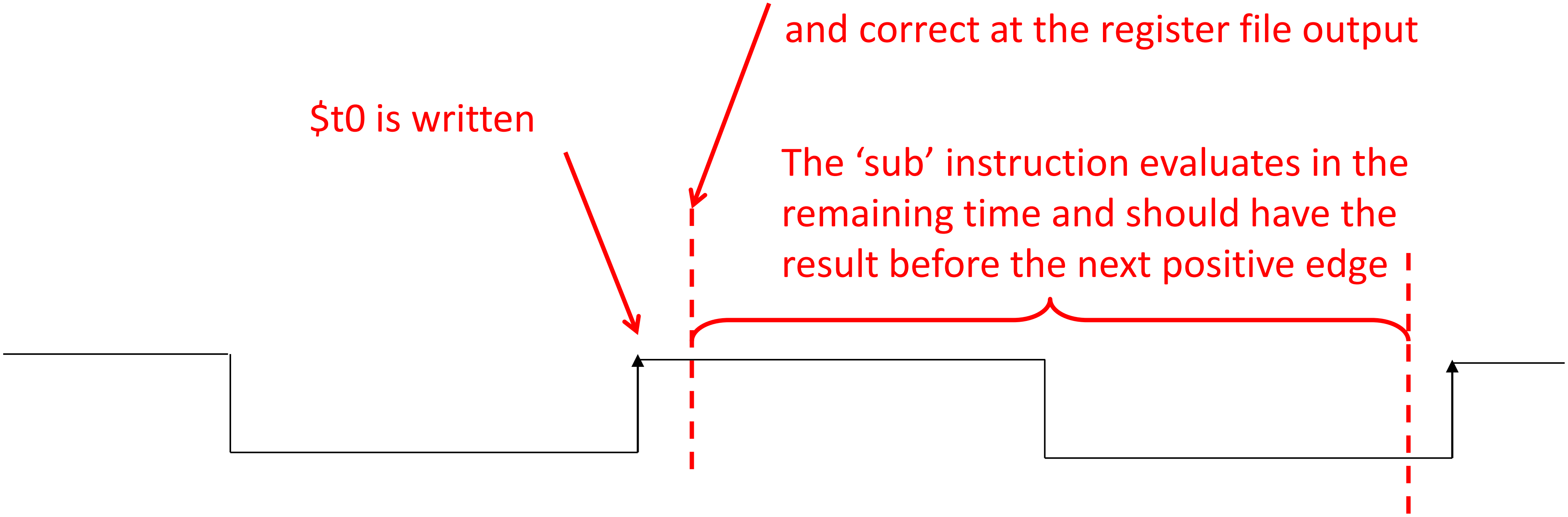
sub $s0, $t0, $s7

Clock cycle 1

Clock cycle 2

Writing and Reading the Same Register

- Register $t0 is written at the end of clock cycle 1
- When clock cycle 2 starts, the value of $t0 takes some time to become stable and correct (but after some delay, it will be correct)
- From that point, there should be enough time for the 'sub' instruction to evaluate successfully before the next positive edge arrives



After some delay, $t0 becomes stable and correct at the register file output

$t0 is written

The 'sub' instruction evaluates in the remaining time and should have the result before the next positive edge

add $t0, $t1, $t2

sub $s0, $t0, $s7

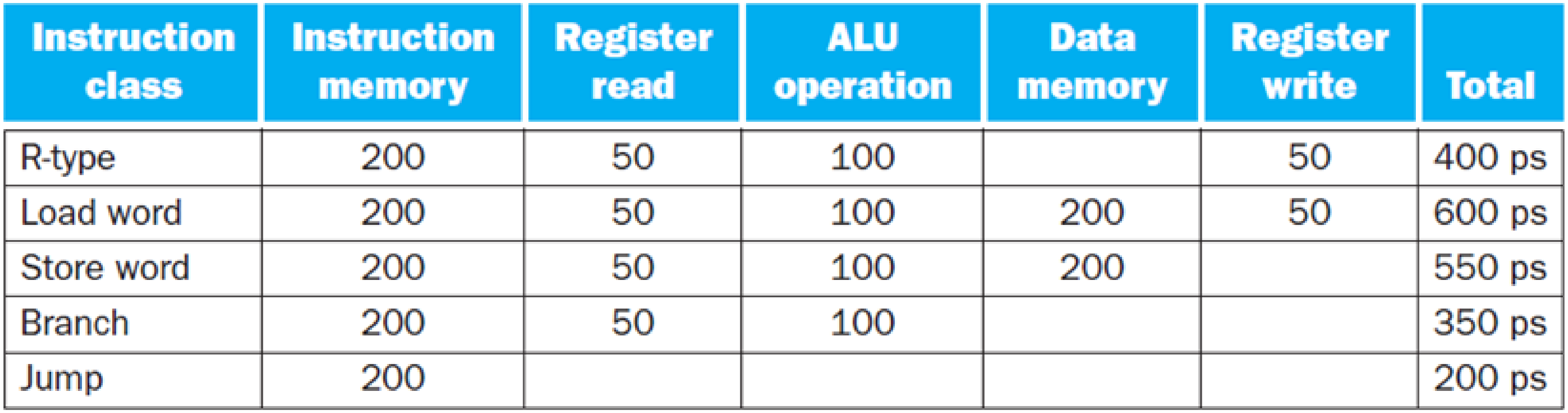
Clock Cycle Time

- At any point in time, there is one instruction in the CPU
- The instruction will finish execution in 1 clock cycle
- The instructions we’re supporting are:
  - add, sub, and, or, slt, lw, sw, beq, j

- The clock cycle length must be at least the duration of the longest instruction
- The “load word” (lw) takes the longest time:
  - Read instruction
  - Read register
  - Add 16-bit address offset to register
  - Read from memory
  - Write back to registers

Clock Cycle Time

- What is the time required by every instruction type?



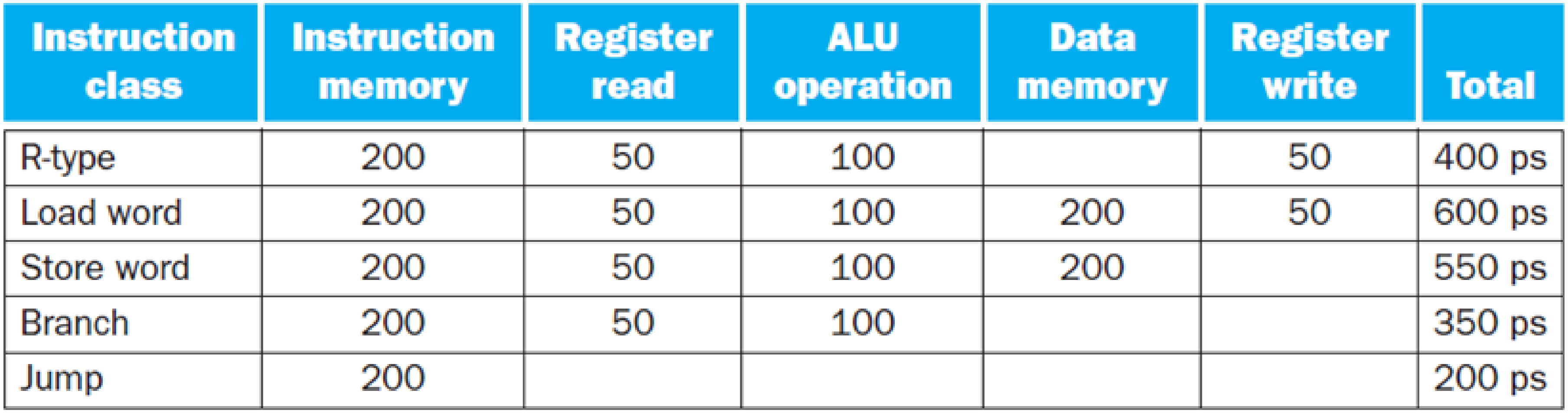
Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100		50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100			350 ps
Jump	200					200 ps

- The clock cycle time is equal to the maximum time among all the instructions
- In this example, it’s 600 ps ➔ F = 1.66 GHz

1 ps (picosecond) = 10^{-12} second

Clock Cycle Time

- How much time are we actually wasting?



Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100		50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100			350 ps
Jump	200					200 ps

- Depends on the instruction mix
- If we’re doing ‘load’ instructions only, we’re not wasting any time since it needs the 600 ps
- However, if we’re doing and R-type, we’re wasting 200 ps since the instruction needs 400ps but we’re giving 600 ps

Instruction Mix

Load:	25%
Store:	10%
R-type:	45%
Branch:	15%
Jump:	5%

- Let’s consider this instruction mix

- Hypothetically, if we’re able to give each instruction only the duration of time it needs, the average time of an instruction is:

$$\text{Time}_{\text{needed}} = (0.25*600)+(0.1*550)+(0.45*400)+(0.15*350)+(0.05*200)$$
$$= 447.5 \text{ ps}$$

- However, with the single-cycle implementation that we have, each instruction takes: 600 ps

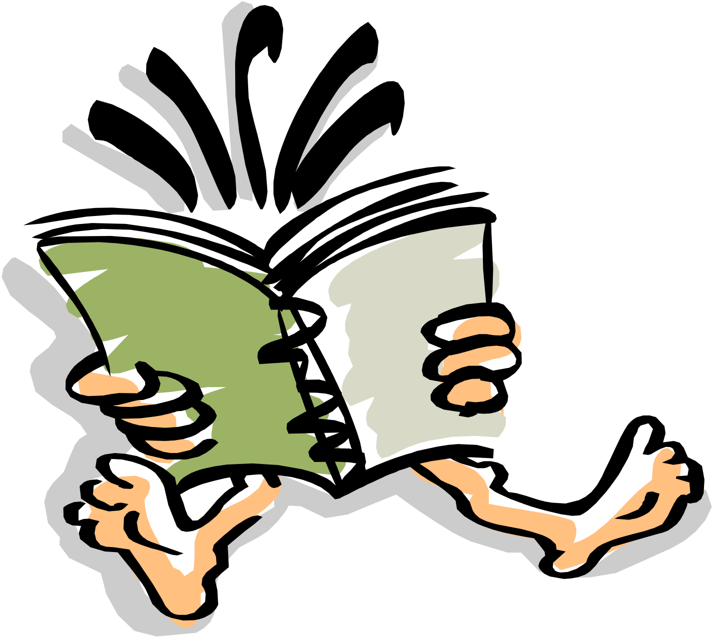
- Therefore, we can potentially speed the CPU by a factor of:

600 / 447.5 = 1.34 times

Clock Cycle Time

- A typical practice in computer design is to have a constant clock duration
- Pro:
  - Simple hardware design
  - Would have to decode the instruction (determine its clock duration) before allowing it into the datapath
- Con:
  - Waste of CPU time
- What to do?

Readings



- H&P COD
  - Chapter 4