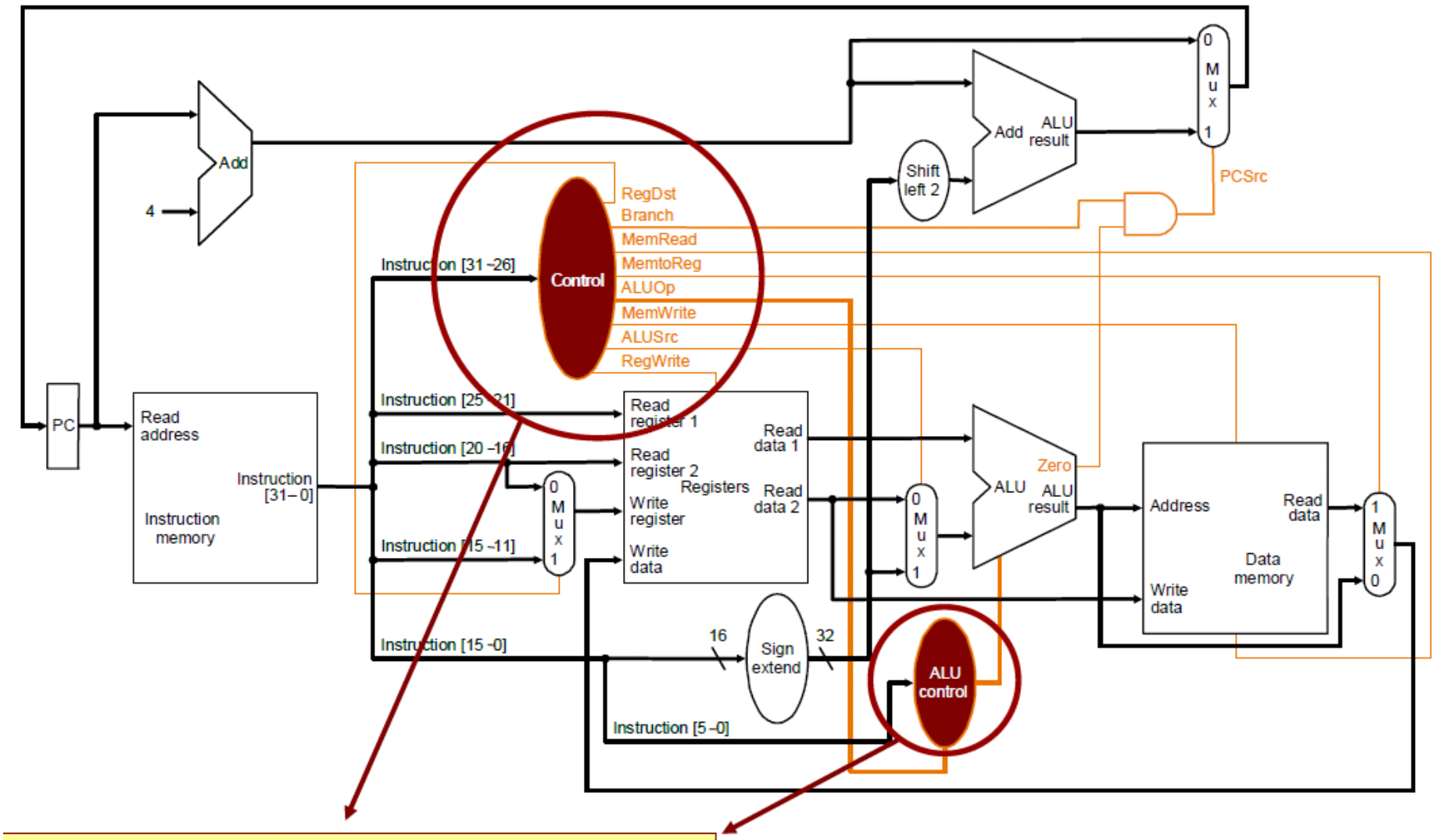
Control

Datapath + Control Unit

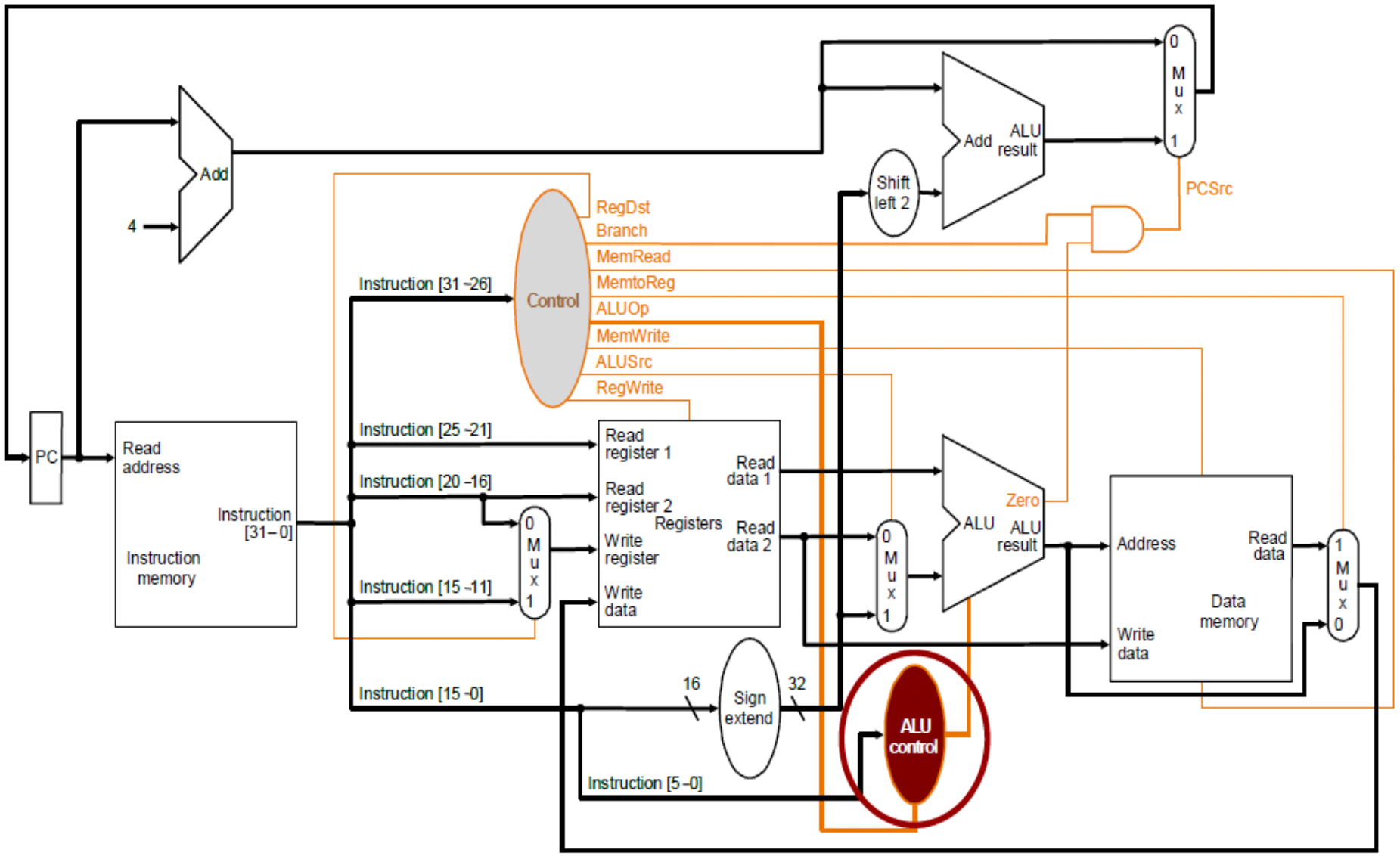


Topic we are going to discuss next

Control Unit

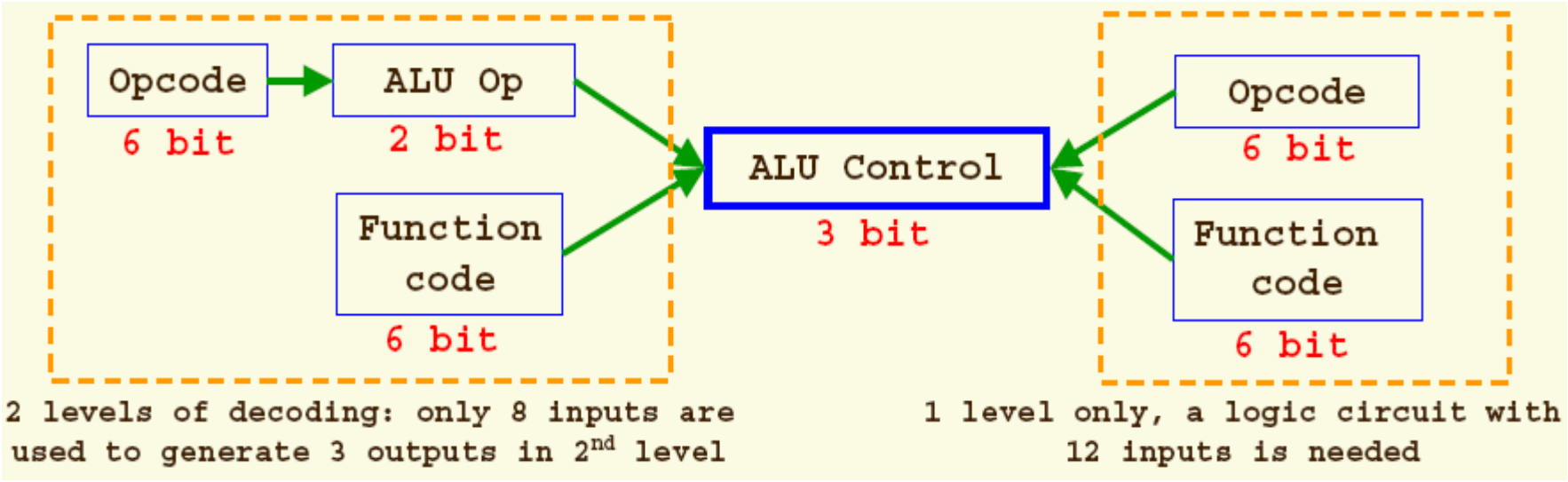
- Datapath control unit controls the whole operation of the datapath
- How? Through control signals, e.g.
  - read/write signals for state elements: RegWrite, MemWrite, MemRead
  - selector inputs for multiplexors: ALUSrc, MemtoReg, PCSrc
  - ALU control inputs (updated to 4 bits) for proper operations
- The ALU control is part of the datapath control unit

ALU Control



Generation of ALU Control Input Bits

- Two common implementation techniques:
  - 1-level decoding:
    - - more input bits.
  - 2-level decoding:
    - + less input bits, less complicated → potentially faster



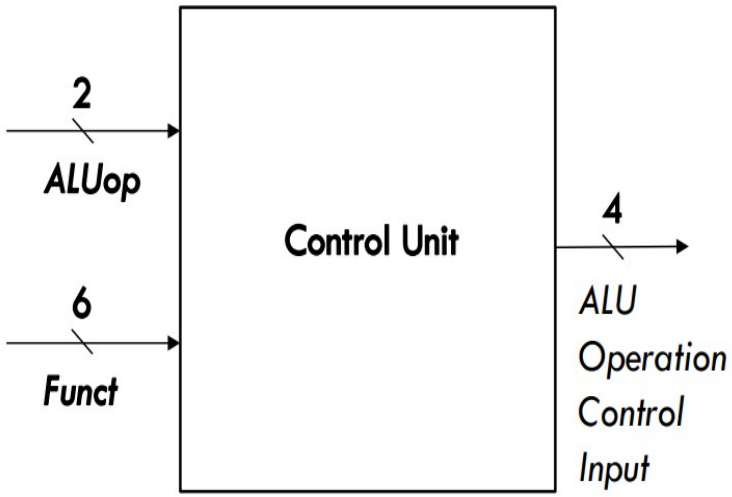
ALU Control Input Bits

- Inputs used by control unit to generate ALU control input bits:
  - **ALUOp** (2 bits)
  - **Function code** of instruction (6 bits)

Instruction operation	Desired ALU action	Instruction opcode	ALUOp	Function code	ALU control input
lw	add	load word	**00**	**XXXXXX**	**0010**
sw	add	store word	**00**	**XXXXXX**	**0010**
beq	subtract	branch equal	**01**	**XXXXXX**	**0110**
add	add	R-type	**10**	**100000**	**0010**
sub	subtract	R-type	**10**	**100010**	**0110**
and	and	R-type	**10**	**100100**	**0000**
or	or	R-type	**10**	**100101**	**0001**
slt	set on less than	R-type	**10**	**101010**	**0111**

ALU CONTROL LINES

Opcode	ALU op	Operation	Funct	ALU action	ALU Control Input
lw	00	Load word	N/A	add	0010
sw	00	Store word	N/A	add	0010
beq	01	Branch equal	N/A	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	slt	0111

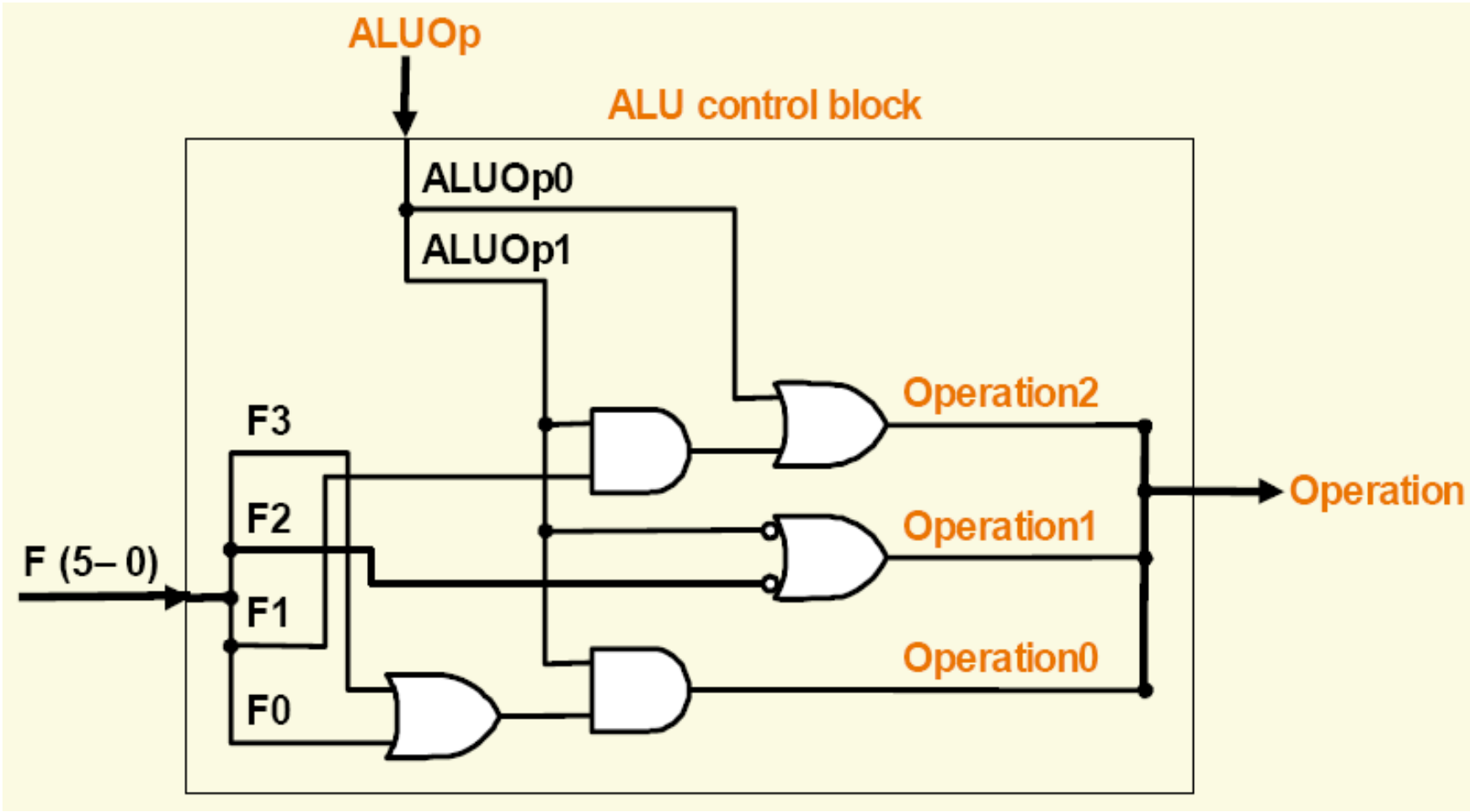


Implementing ALU Control Block

- Start from truth table
- Smart design converts many entries in the table to don't-care terms, leading to a simplified hardware implementation

ALUOp		Function code						Operation	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	0010	lw, sw
X	1	X	X	X	X	X	X	0110	beq
1	X	X	X	0	0	0	0	0010	R-type Instr.
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	

Hardware Implementation of ALU Control Block

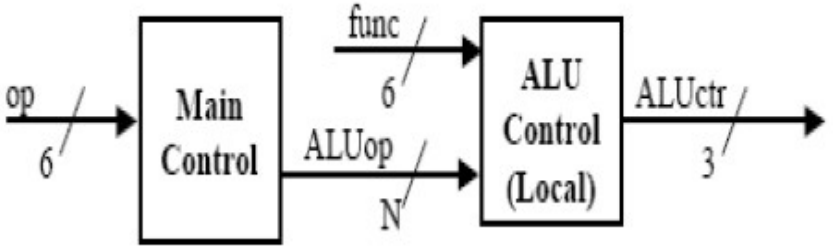


Effects of Control Signals

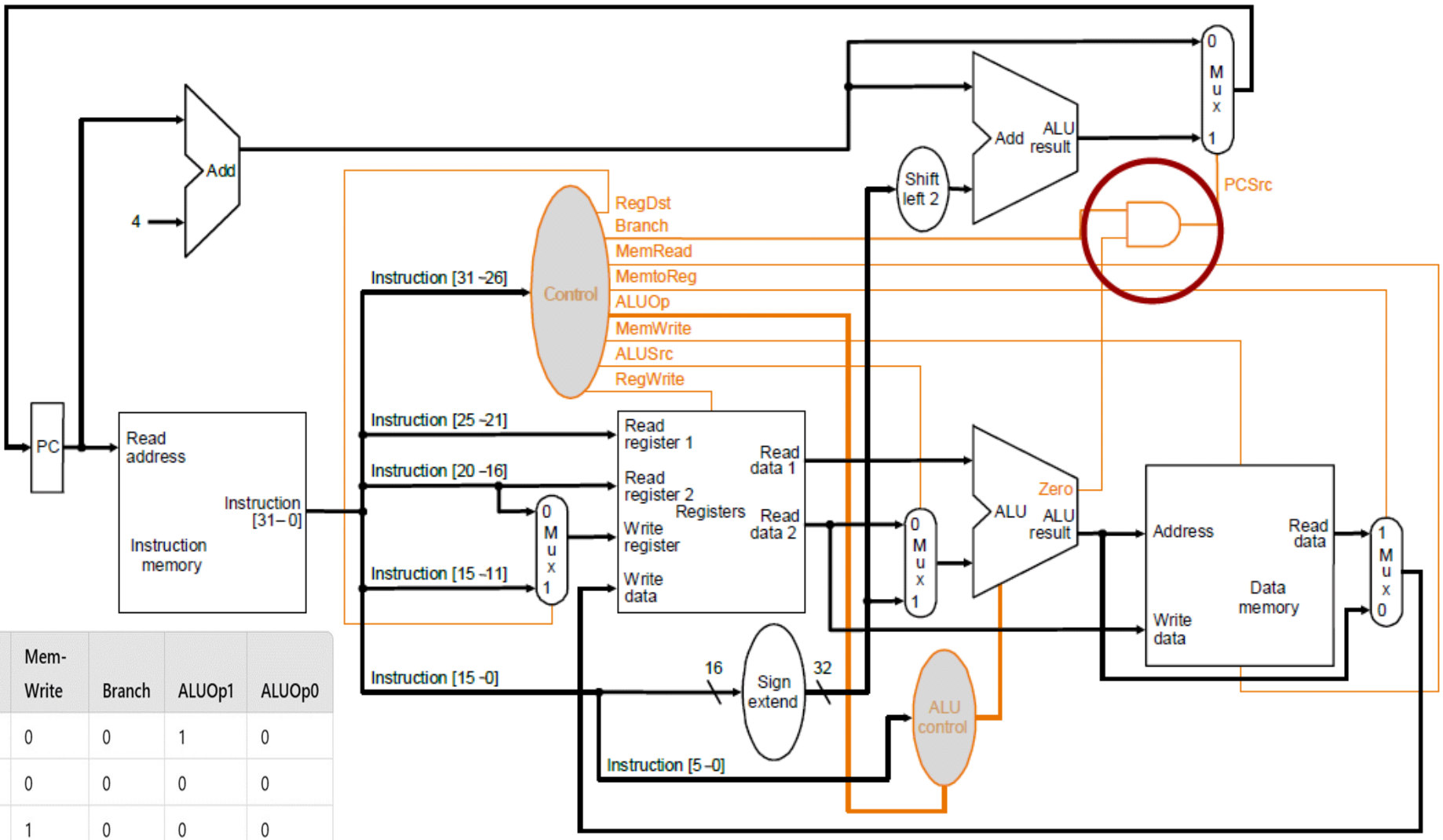
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from **rt** field (bits 20-16)	The register destination number for the Write register comes from **rd** field (bits 15-11)
RegWrite	None	Enable data write to the register specified by the register destination number
ALUSrc	The second ALU operand comes from the second register file output (Read data port 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The next PC picks up the output of the adder that computes PC+4	The next PC picks up the output of the adder that computes the branch target
MemRead	None	Enable read from memory. Memory contents designated by the address are put on the Read data output
MemWrite	None	Enable write to memory. Overwrite the memory contents designated by the address with the value on the Write data input
MemtoReg	Feed the Write data input of the register file with output from ALU	Feed the Write data input of the register file with output from memory

Setting of Control Signals

- The 9 control signals (7 from the previous table + 2 from ALUOp) can be set based entirely on the 6-bit opcode, with the exception of PCSrc.
- PCSrc control line is set if both conditions hold simultaneously:
  - Instruction is a branch, e.g. **beq**
  - Zero output of ALU is true (i.e., two source operands are equal)



Checking the Conditions for PCSrc



em- ad	Mem-Write	Branch	ALUOp1	ALUOp0
	0	0	1	0
	0	0	0	0
	1	0	0	0
	0	1	0	1
	0	0	X	X

Setting of Control Signals ~

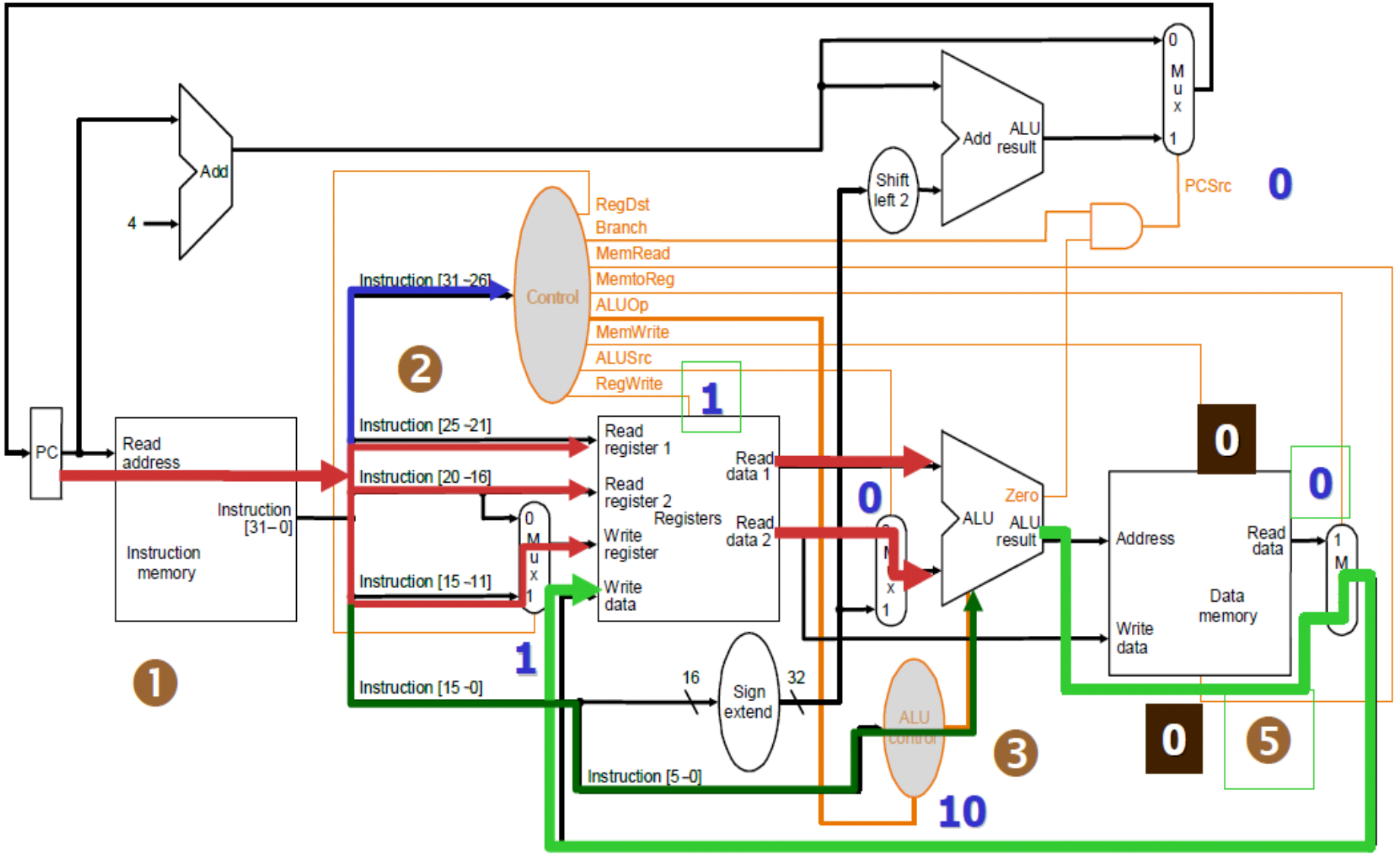
- Setting of control lines (output of control unit):

  sw & beq will not modify any register, it is ensured by making RegWrite to 0, So, we don't care what write register & write data are

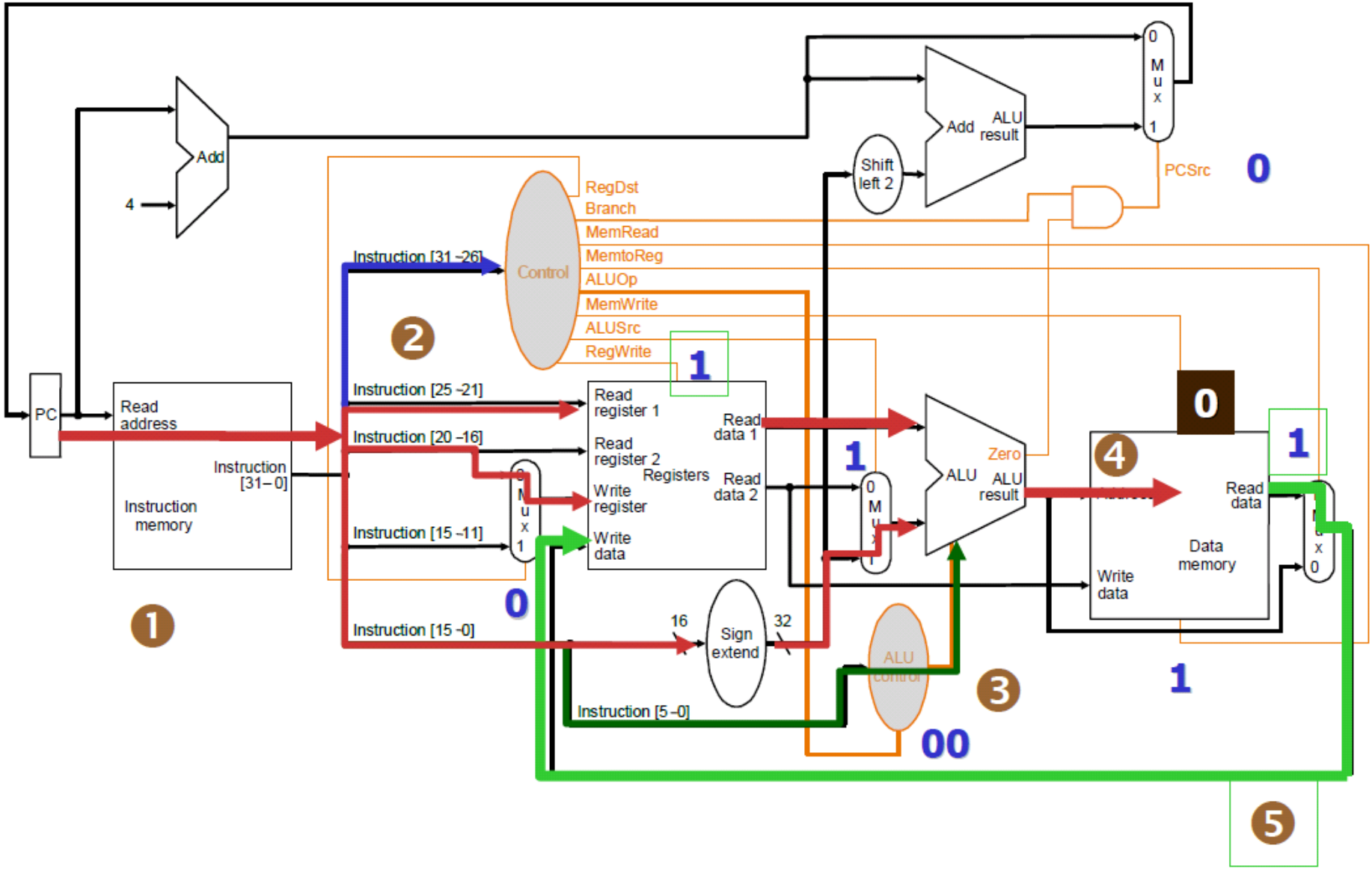
Instruction	Reg-Dst	ALU-Src	Mem-toReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Instruction	Reg-Dst	ALU-Src	Mem-toReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
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
j	X	X	X	0	0	0	0	X	X

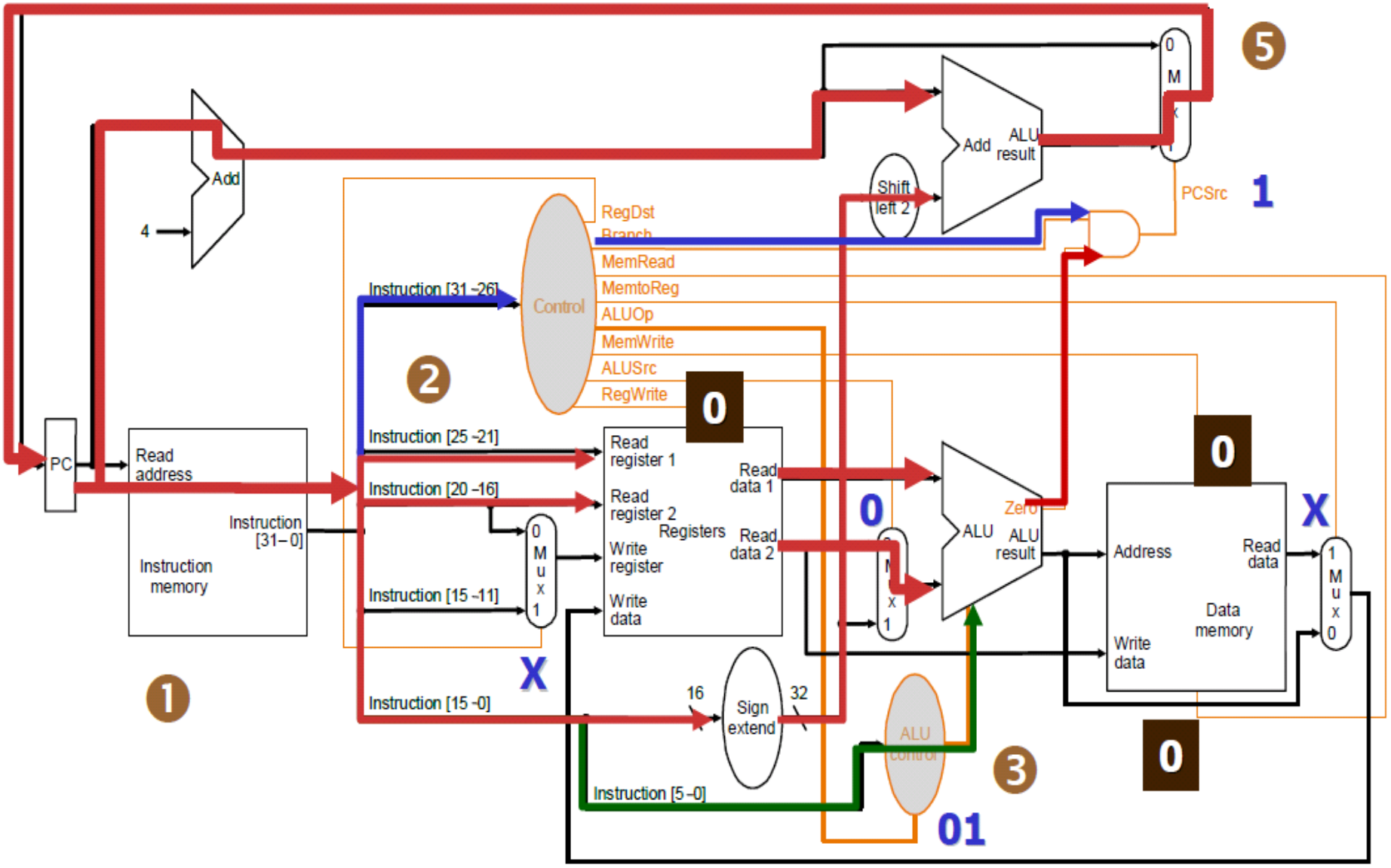
Review Datapath for R-Type Instr.



Review Datapath for Load Instr



Review Datapath for Branch Instr.

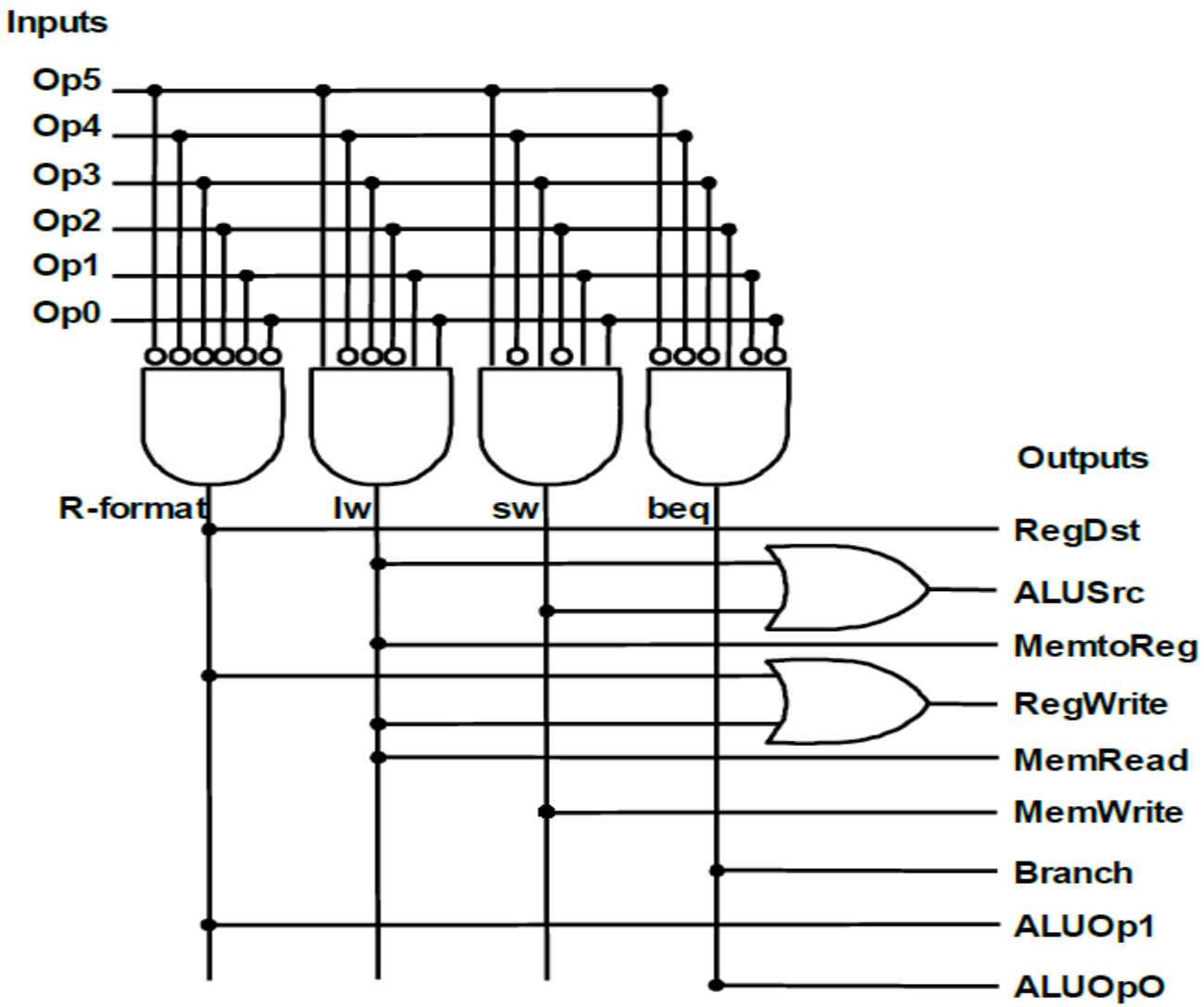


Implementing Datapath Control Unit

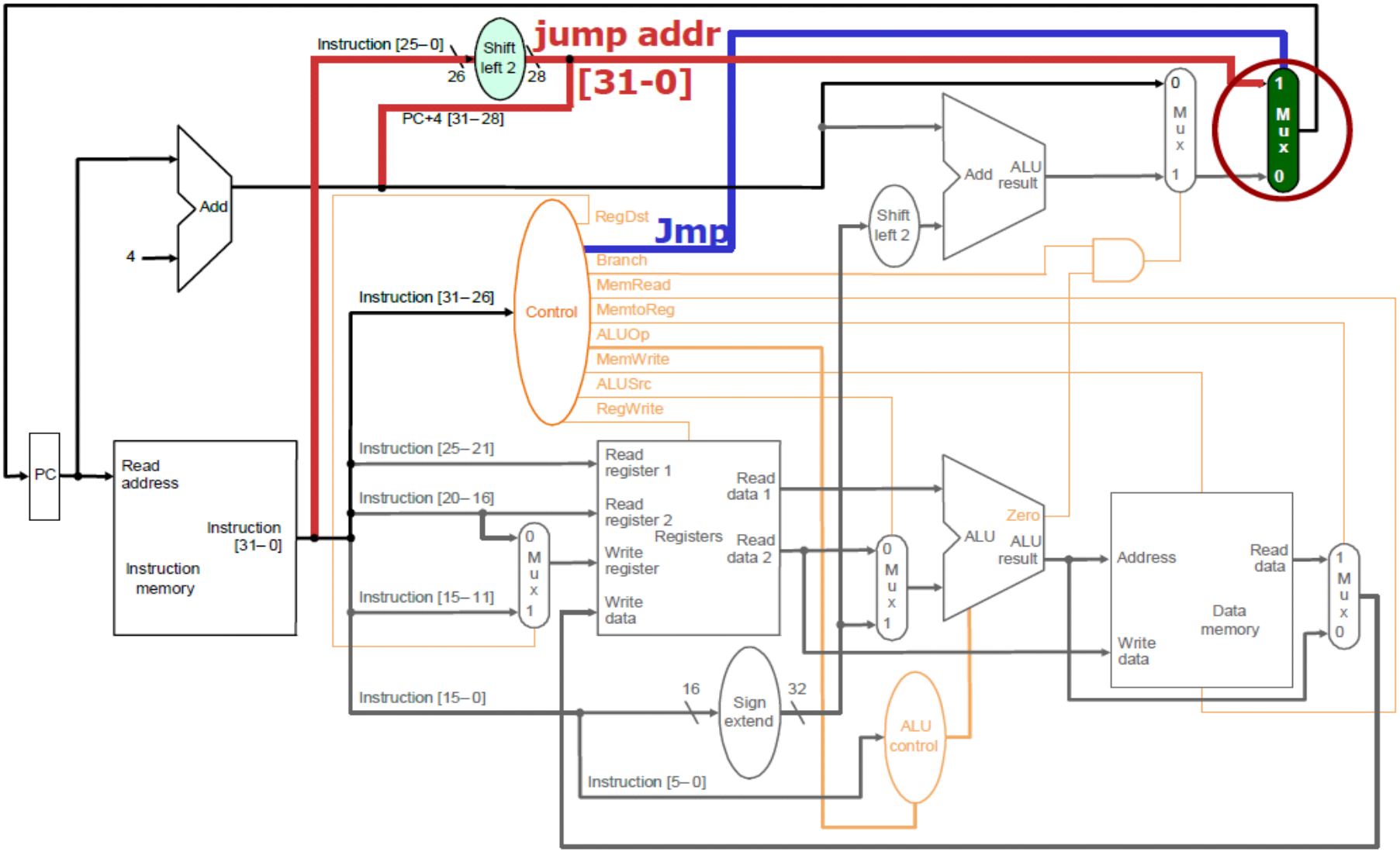
- Start with truth table

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

HW Implementation of Datapath Control Unit



Extend Datapath & Control to Handle Jump Instr



Problems with Single-Cycle Datapath Implementation

Single-cycle implementation cant run very fast, Why?

- Every instruction takes one clock cycle (CPI = 1), and
- Clock cycle is determined by the longest path in the machine
  →i.e. clock cycle is expected to be large, resulting in poor performance

What we have seen so far, the longest path is for a load instruction

- Load involves five functional units in series
- i.e. instruction mem., register file, ALU, data mem., register file

It is more severe when considering other computational instructions

- e.g. multiplication, division, and floating-point operations, etc.

Other issues

- Sharing of hardware functional units is NOT possible within a cycle

Can We Adopt Variable Clock Implementation?

Motivation:

- Many instructions do not involve all five functions units

Question:

- Why don't vary the clock for instructions with shorter execution time?

Answer:

- A variable-speed clock is very difficult to implement in hardware

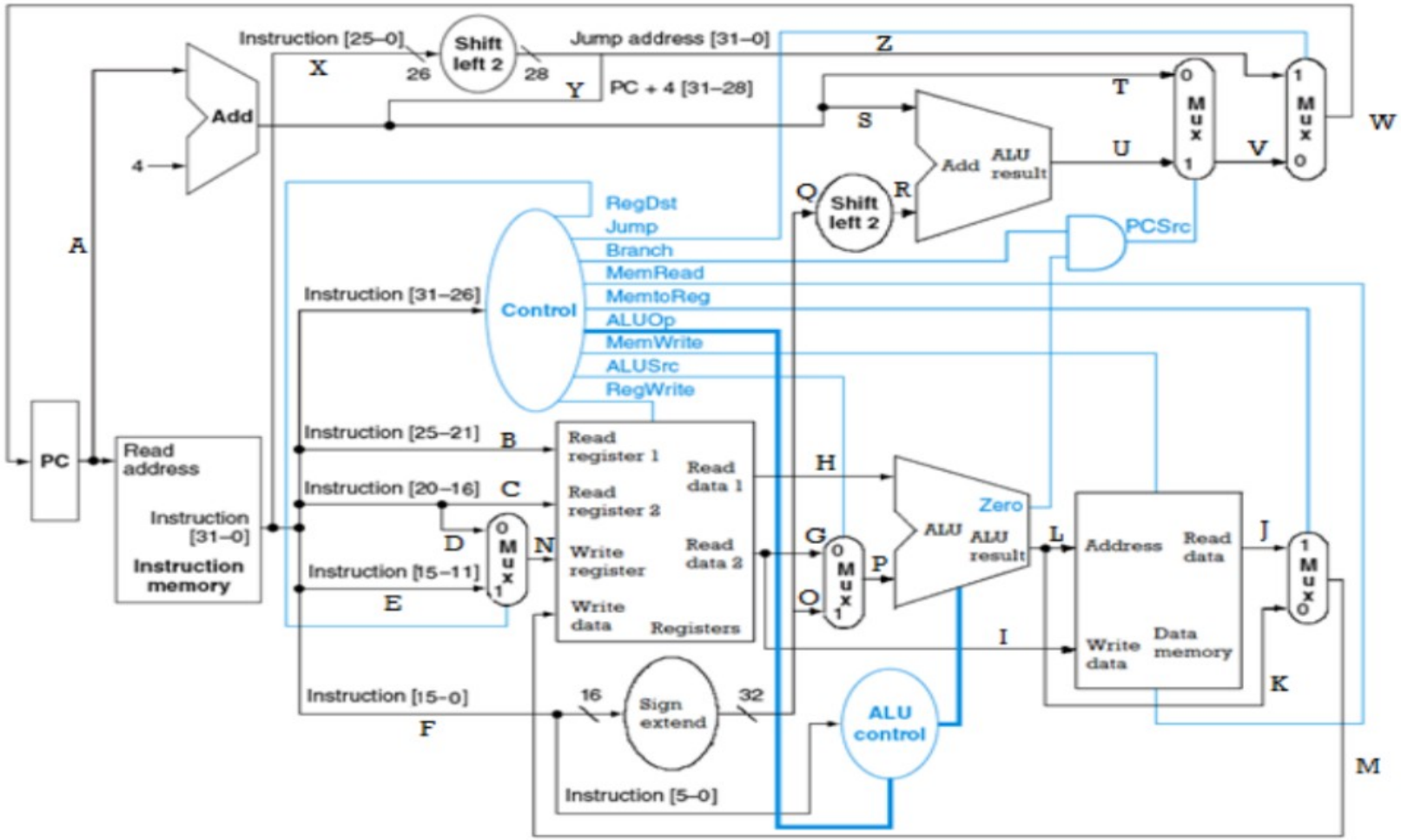
The better solution:

- Consider a shorter clock cycle shorter cycle and allow different types of instructions to take different numbers of clock cycles

Exercise: **Consider the following MIPS single cycle processor, and the snapshot of the instruction memory, data memory, and the register file.**

Task:

Execute the instructions and fill the following table , assuming that the **PC register contains 0x00400004**



Instruction Memory

Address	Contents	Instruction
0x00400000	0x0253A825	OR $S5, $S2, $S3
0x00400004	0x02309020	ADD $S2, $S1, $S0
0x00400008	0x12530001	BEQ $S2, $S3, 1
0x0040000C	0x8E93FFFE	LW $S3, -2($S4)
0x00400010	0x08100001	J 0x100001

Register File

Register	Data
$S0 (16)	0x0000000A
$S1 (17)	0x00000001
$S2 (18)	0x02EA0160
$S3 (19)	0x00000011
$S4 (20)	0x00000006
$S5 (21)	0x00123004

Data Memory

Address	Data
0x00020002	0x13
0x00020003	0x2E
0x00020004	0xA4
0x00020005	0xFF
0x00020006	0x08
0x00020007	0x2C

Instr. Buses					
A					
B					
C					
D					
E					
F					
G					
H					
I					
J					
K					
L					
M					
N					
O					
P					
Q					
R					
S					
T					
U					
V					
W					
X					
Y					
Z					
Control Sig.					
RegDst					
RegWrite					
ALUSrc					
MemRead					
MemWrite					
MemtoReg					
Jump					
Branch					
ALUOpr1					
ALUOpr0					
PCSrc					
Registers					
16 ($s0)					
17 ($s1)					
18 ($s2)					
19 ($s3)					
20 ($s4)					
21 ($s5)					

Instr. / Buses	ADD $S2, $S1, $S0
A	0x00400004
B	17 ($s1)
C	16 ($s0)
D	16 ($s0)
E	18 ($s2)
F	----
G	0x0000000A
H	0x00000001
I	-----
J	------
K	0x0000000B
L	0x0000000B
M	0x0000000B
N	18 ($s2)
O	-----
P	0x0000000A
Q	------
R	----
S	------
T	0x00400008
U	-----
V	
W	0x00400008
X	----
Y	----
Z	-----

Control Sig.	
RegDst	1
RegWrite	1
ALUSrc	0
MemRead	0
MemWrite	0
MemtoReg	0
Jump	0
Branch	0
ALUOpr1	1
ALUOpr0	0
PCSrc	0

Registers	
16 ($s0)	NO CHANGE
17 ($s1)	NO CHANGE
18 ($s2)	0x0000000B
19 ($s3)	NO CHANGE
20 ($s4)	NO CHANGE
21 ($s5)	NO CHANGE

Restı

Multicycle Datapath and control

Multicycle Implementation

- What does it mean by Multicycle?
- An instruction takes multiple cycles to execute
  → Execution of each instruction is broken into a series of steps
  1. Fetch the instruction
  2. Decode instruction & read the registers
  3. Perform ALU operation
  4. Memory access (if necessary)
  5. Write back the result
- Each step takes equal amount of time (one cycle) to complete

Why Multicycle?

Key advantage:

- Different types of instructions can have different number of cycles
- Short instructions can complete early
  - e.g. R-type instruction can execute in 4 cycles instead of 5

  → Better performance as compared to the single-cycle implementation

Other advantage:

- Better sharing of function units → less hardware → cheaper

→ A single functional unit can be used more than once per instruction (as long as it is used at different clock cycles)

  - e.g. a single memory unit is used for both instructions and data
  - e.g. need only a single ALU, rather than one ALU and two adders

Trade-off

As compared to single-cycle implementation, multicycle approach needs

A) more temporary storage to hold values between clocks.
- i.e. one or more registers added after every major functional unit
- Instruction registers (IR)
- Memory data registers (MDR)
- Registers A & B
- ALUOut

B) More complex control
- Multiplexors
- Control signals

- Question: Worth adding all these?!

Example: Single-Cycle CPU vs. Multicycle CPU

The instruction mix of a program:

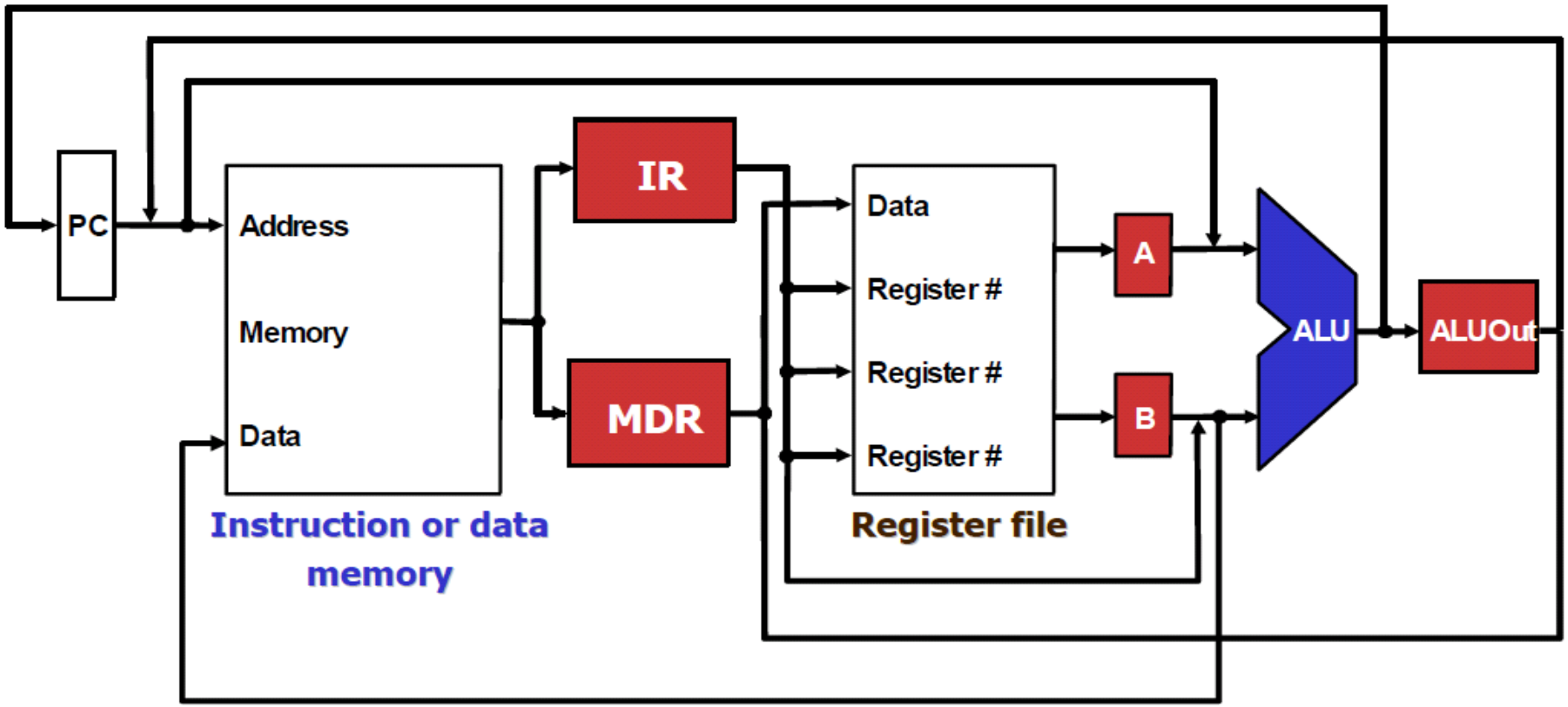
	Load	Store	Branch	Jump	ALU
# of instructions	25%	10%	11%	2%	52%
# of clocks/instr.	5	4	3	3	4

- Clock cycle of single-cycle CPU is **5x** of clock cycle of multicycle CPU
- For multicycle CPU (assuming the clock cycle is **R**)
  - $CPI_{multi} = 0.25*5 + 0.10*4 + 0.11*3 + 0.02*3 + 0.52*4 = 4.12$
  - $Time_{multi} = CPI_{multi} * \text{clock cycle}_{multi} = 4.12 * \mathbf{R} = \mathbf{4.12R}$
- For single-cycle CPU
  - $CPI_{single} = 1.0$
  - $Time_{single} = CPI_{single} * \text{clock cycle}_{single} = 1.0 * \mathbf{5R} = \mathbf{5R}$

5R / 4.12R = 1.21 ⇒ multicycle CPU speeds up by 21%

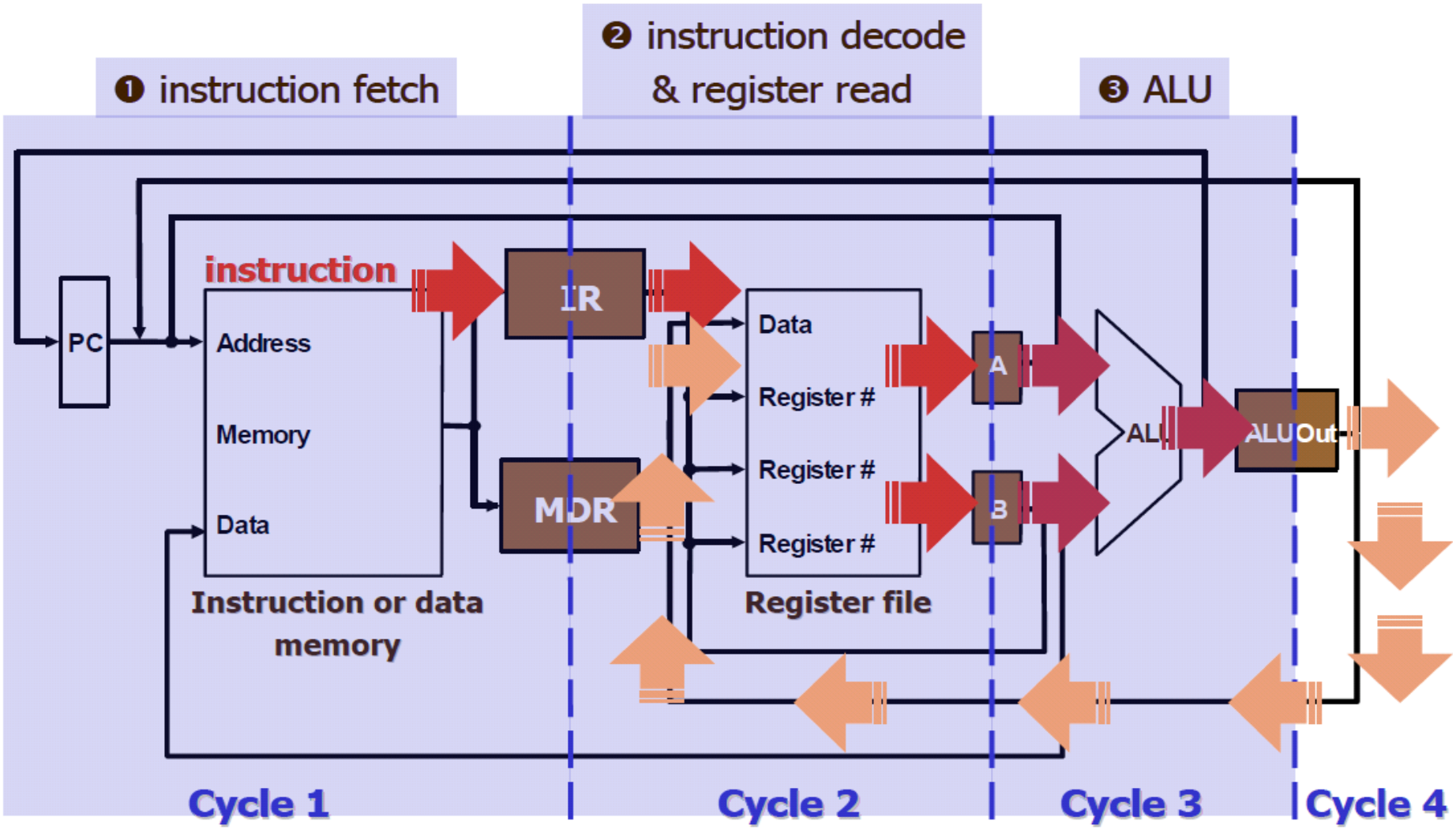
Overview of the Multicycle Datapath

- IR: Instruction Register; to hold instruction from memory
- MDR: Memory Data Register; to hold data from memory
- A, B: registers to hold operand values from register file
- ALUOut: register to hold the output of the ALU



Overview: How Do These Buffer Work?

- Let's use R-type instruction as an example



More About Temporary Storage

Memory is connected to both IR and MDR

- Output of the memory goes to both of them at the same time
- It seems like
  - Instruction read will destroy the value stored in MDR, or
  - Data read will destroy the instruction stored in IR

ALUOut

- Holds the ALU's output (which will serve as an input for next cycle)
- The ALU's output can be either
  - Data (to be written to register or memory), or
  - Branch or jump target address

Need for Additional Control Signals

- All except IR hold data only between a pair of adjacent clock cycles

  →Thus, MDR, A, B, ALUOut do not need a write control signal
- IR needs to hold the instruction until the end of instruction execution

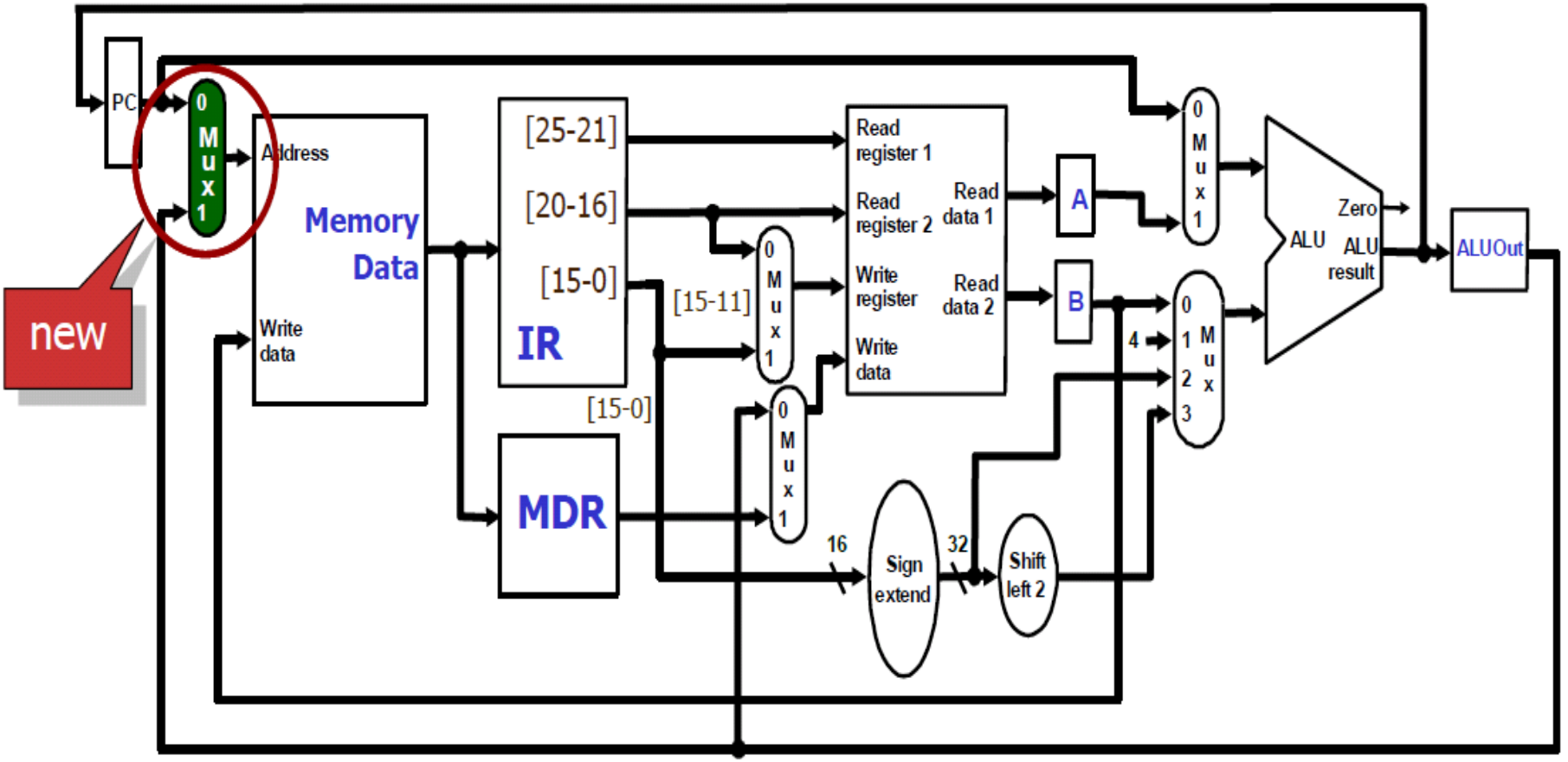
  →Thus, it requires a write control signal
- Since several functional units are shared for different purposes,

  →Multiplexors have to be added or expanded

  →Thus, additional control signals are needed
- See next couple slides

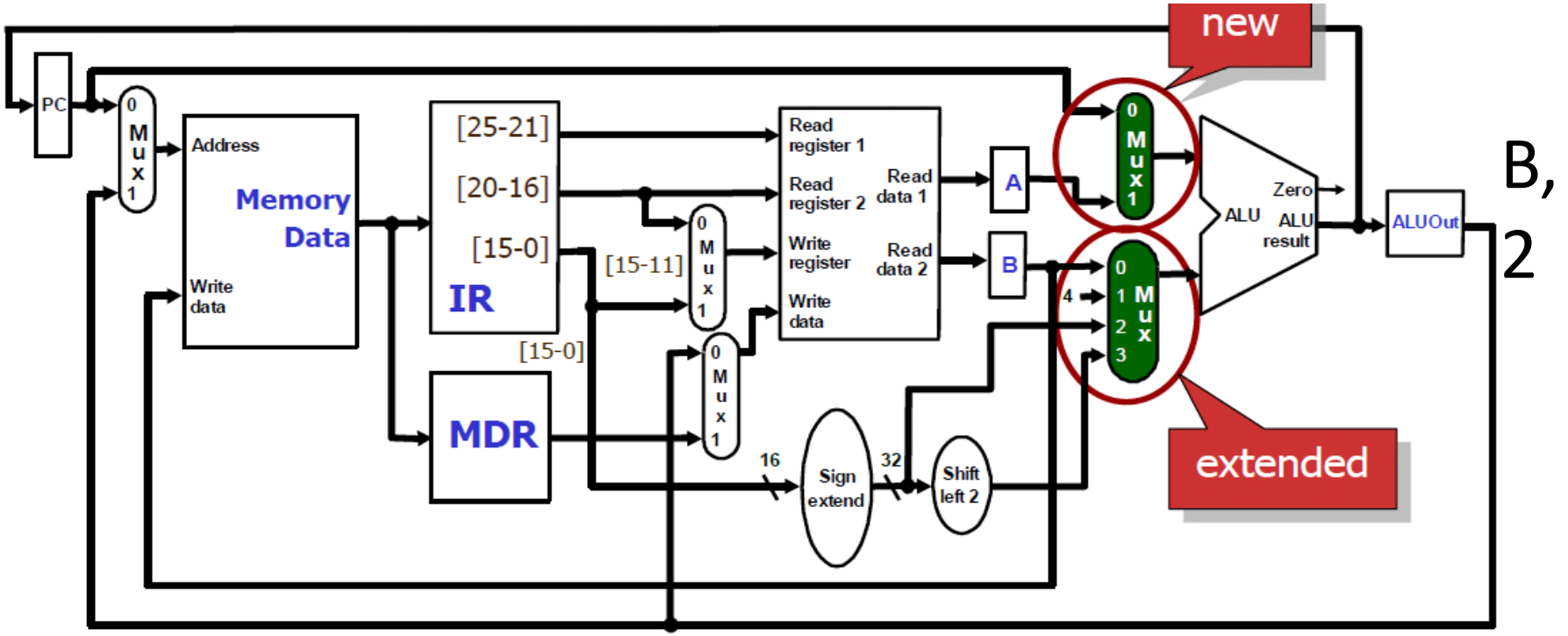
Multiplexing Addresses to Read the Memory

- Memory is shared by both instructions and data
  - The address to the memory module is no



Computation of Next PC Using the ALU

- The adders to compute next PC in single-cycle approach are removed, So, generation of next PC and execution of operation share the ALU
- →The first source to the ALU can be either A or B, 2



Constraints for Multicycle datapath

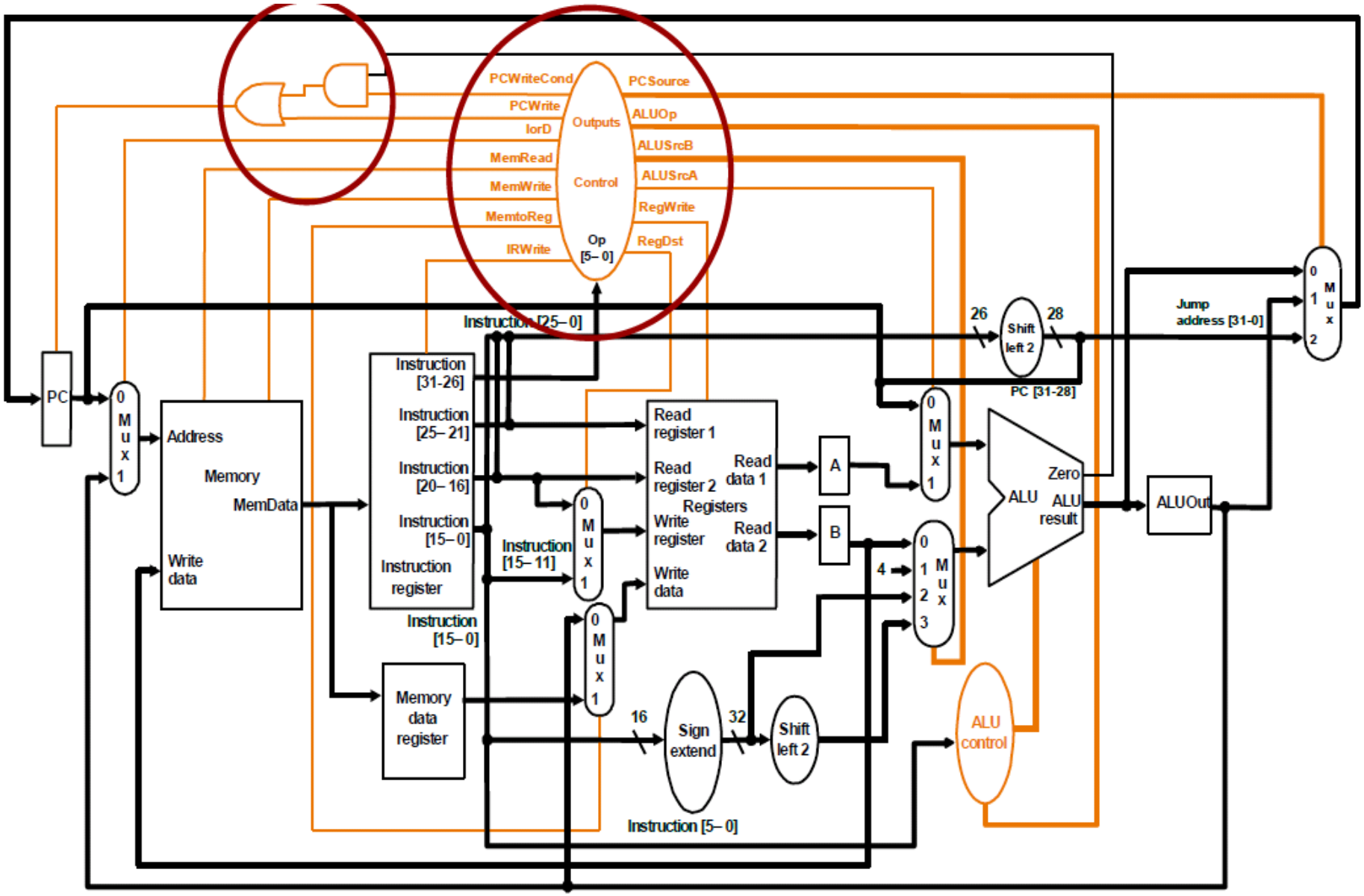
Only ONE ALU operation can be performed at each cycle

- Will calculation of next PC conflict with other ALU operation?
  - If yes, what should we do then?
  - If no, why?

Only ONE register file access can be performed at each cycle

- Will read from registers conflict with write to register?
  - If yes, what should we do then?
  - If no, why?

Datapath & Control for Multicycle Implementation



Multiple Execution Steps Per Instruction

- **Typical execution steps**:
  - **Instruction fetch**
  - **Instruction decode** and **register fetch**
  - **Execution**, **memory address computation**, or **branch completion**
  - **Memory access** or **R-type instruction completion**
  - **Memory read completion**
- Each instruction takes a few (3-5) steps.

Conclusions

- **MIPS ISA**: Three instruction formats (R,I,J)
- One cycle per stage, Different stages per format
- **One-Cycle Steps**

	R-fmt	lw	sw	beq	j
1. Instruction Fetch	■	■	■	■	■
2. Instruction Decode / Data Fetch	■	■	■	■	■
3. ALU ops / R-format Execution	■	■	■	■	■
4. R-format Completion	■	■	■		
5. Memory Access Completion		■			

Challenge: More involved control design

Multicycle DP: R-format

Step 1: Fetch instr. // Store in IR // Compute PC + 4

Step 2: Decode instruction: opcode, rd, rs, rt, funct fields

Data fetch: Apply rs, rt to Register File

Data Read into A,B buffer registers (ALUin)

Step 3: ALU operation (ALUsrcA, ALUsrcB, ALUop)

ALU output goes into ALUout register

Step 4: ALUout register contents written to Register File write input

Register number in rd written (*Assert*: RegWrite,RegDst)

CPI for R-format = 4 cycles

Multicycle DP: Store Word (SW)

Step 1: Fetch instr. // Store in IR // Compute PC + 4

Step 2: Decode instruction: opcode, rs, rt, offset fields

Data fetch: Apply rt to Register File => Base address

Data Read into A buffer register (Base)

SignExt,Shift offset field into B buffer register

Step 3: ALU operation (ALUsrcB, ALUop) => Base + Offset

ALU output goes into ALUout register

Step 4: ALUout register contents applied as Memory Address

Assert: MemWrite [ALUout => RegFile]

CPI for Store = 4 cycles

Multicycle DP: Load Word (lw)

Step 1: Fetch instr. // Store in IR // Compute PC + 4

Step 2: Decode instruction: opcode, rd, rt, offset fields

Data fetch: Apply rt to Register File => Base address

Data Read into A buffer register (Base)

SignExt,Shift offset field into B buffer register

Step 3: ALU operation (ALUsrcB, ALUop) => Base + Offset

ALU output goes into ALUout register

Step 4: ALUout register contents applied as Memory Address

Assert: MemRead

Step 5: Memory Data Out routed to Register File write input

Register number from rd written to (*Assert*:

CPI for Load = 5 cycles

Multicycle DP: Cond. Branch

Step 1: Fetch instr. // Store in IR // Compute PC + 4

Step 2: Decode instruction: opcode, rs, rt, offset fields

Data fetch: Apply rs, rt to Register File

BTA calc: SignExt,Shift offset field into B buffer register

ALU compose PC, offset => BTA

Step 3: **ALU** operation (ALUsrcA, ALUsrcB, ALUop) = compare

ALU output present at Zero register causes Control to select BTA or PC+4

CPI for Conditional Branch = 3 cycles

Multicycle DP: Jump

Step 1: Fetch instr. // Store in IR // Compute PC + 4

Step 2: Decode instruction: opcode, address fields

JTA calc: SignExt,Shift offset field [Bits 27-0]

Concatenate with PC [Bits 31-28] => JTA

Step 3: PC replaced by the Jump Target Address (JTA)

PCsource = 10, PCWrite asserted

CPI for Jump = 3 cycles

Multi-cycle datapath: summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25–21]] B = Reg [IR[20–16]] ALUOut = PC + (sign-extend (IR[15–0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15–0])	if (A == B) then PC = ALUOut	PC = PC [31–28] || (IR[25–0]<<2)
Memory access or R-type completion	Reg [IR[15–11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20–16]] = MDR		

MULTI-CYCLE DATAPATH AND CONTROL

R-FORMAT

R-format instructions require 4 cycles to complete. Let's imagine that we're executing an add instruction.

```
add $s0, $s1, $s2
```

which has the following fields

R-format instructions require 4 cycles to complete. Let's imagine that we're executing an add instruction.

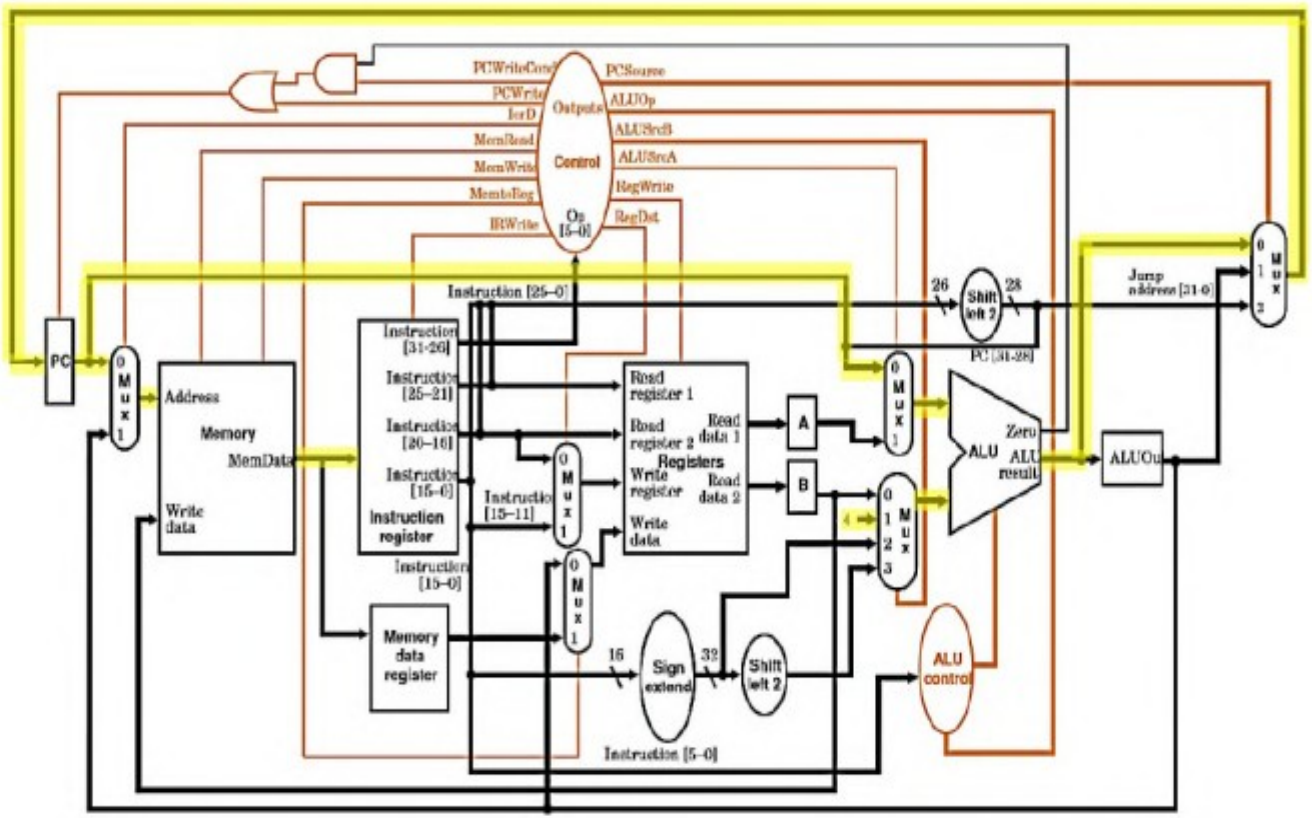
```
add $s0, $s1, $s2
```

which has the following fields

opcode	rs	rt	rd	shamt	funct
000000	10001	10010	10000	00000	100000

R-FORMAT: CYCLE 1

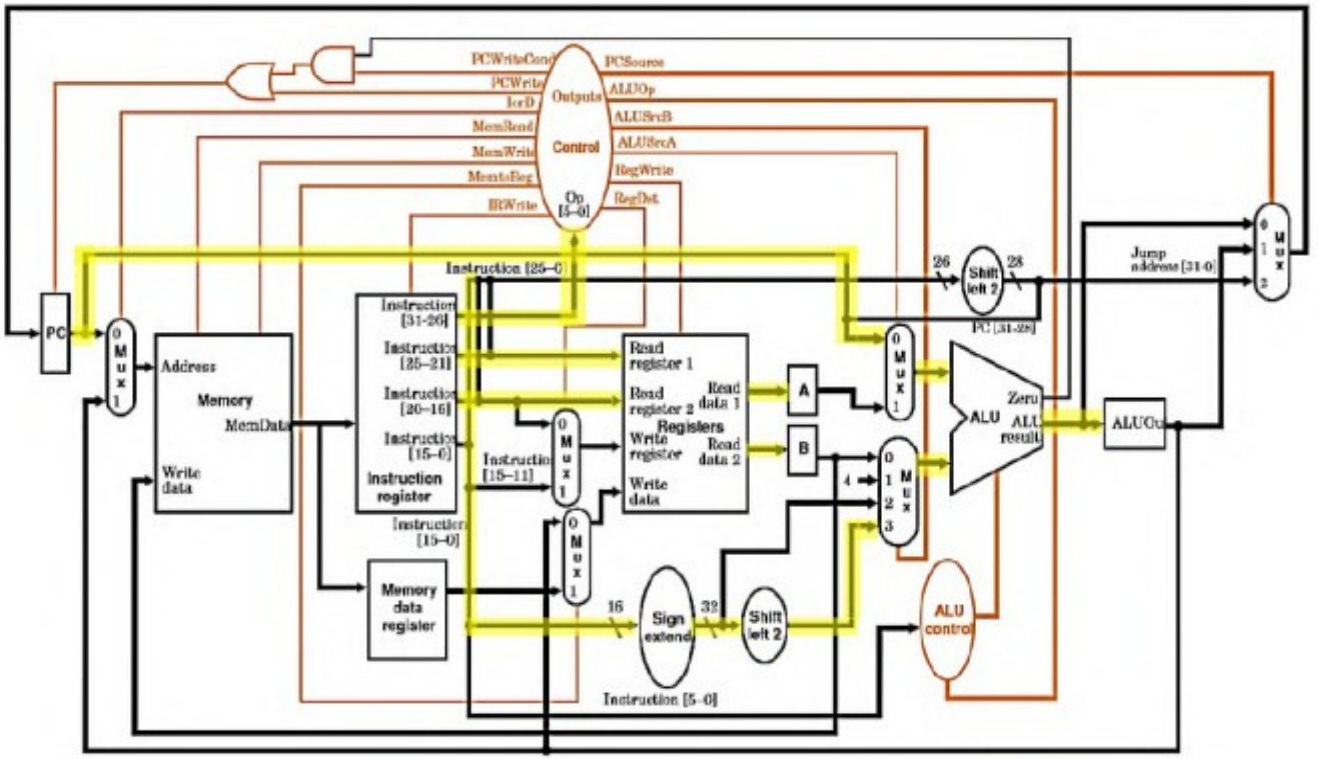
Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	00
ALUSrcB	01
ALUSrcA	0
RegWrite	0



Cycle	PCWrite	IRWrite	MemRead	MemWrite	RegWrite	ALUSrcA	ALUSrcB (2-bit)	ALUOp	RegDst	MemtoReg	IorD	PCSource
1 (Fetch)	1	1	1	0	0	0	01	00	X	X	0	00
2 (Decode)	0	0	0	0	0	0	11	00	X	X	X	X
3 (ALU)	0	0	0	0	0	1	00	10	X	X	X	X
4 (Write)	0	0	0	0	1	X	X	X	1	0	X	X

R-FORMAT: CYCLE 2

Signal	Value
ALUOp	00
ALUSrcB	11
ALUSrcA	0

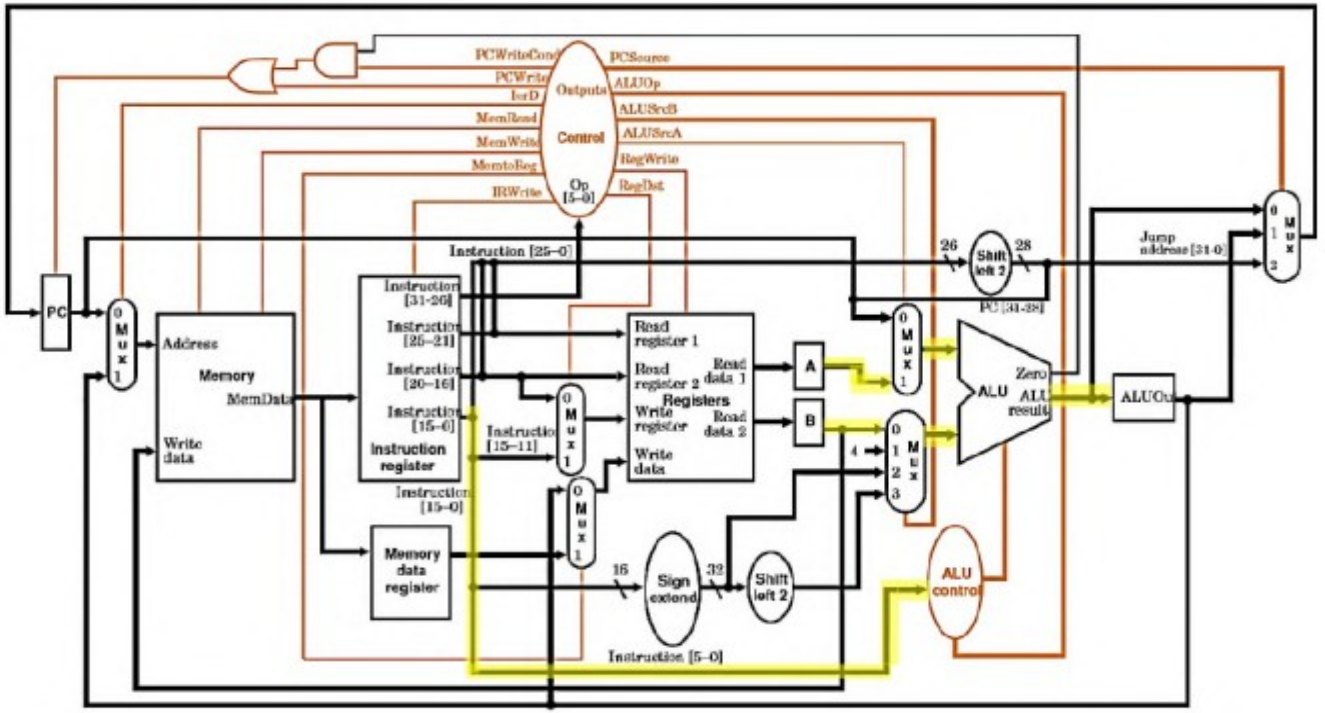


Note that we compute the speculative branching target in this step even though we will not need it. We have nothing better to do while we decode the instruction so we might as well.

Cycle	PCWrite	IRWrite	MemRead	MemWrite	RegWrite	ALUSrcA	ALUSrcB (2-bit)	ALUOp	RegDst	MemtoReg	IorD	PCSource
1 (Fetch)	1	1	1	0	0	0	01	00	X	X	0	00
2 (Decode)	0	0	0	0	0	0	11	00	X	X	X	X
3 (ALU)	0	0	0	0	0	1	00	10	X	X	X	X
4 (Write)	0	0	0	0	1	X	X	X	1	0	X	X

R-FORMAT: CYCLE 3

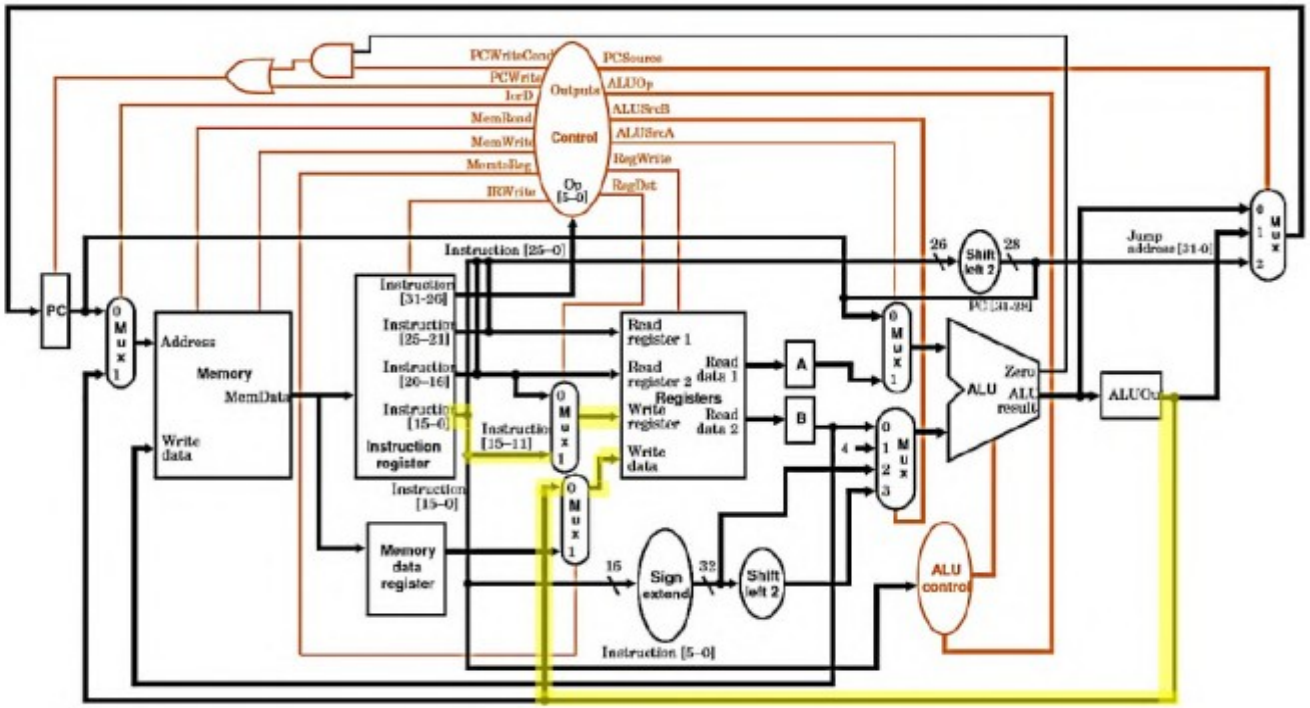
Signal	Value
ALUOp	10
ALUSrcB	00
ALUSrcA	1



Cycle	PCWrite	IRWrite	MemRead	MemWrite	RegWrite	ALUSrcA	ALUSrcB (2-bit)	ALUOp	RegDst	MemtoReg	IorD	PCSource
1 (Fetch)	1	1	1	0	0	0	01	00	X	X	0	00
2 (Decode)	0	0	0	0	0	0	11	00	X	X	X	X
3 (ALU)	0	0	0	0	0	1	00	10	X	X	X	X
4 (Write)	0	0	0	0	1	X	X	X	1	0	X	X

R-FORMAT: CYCLE 4

Signal	Value
MemtoReg	0
RegWrite	1
RegDst	1



Cycle	PCWrite	IRWrite	MemRead	MemWrite	RegWrite	ALUSrcA	ALUSrcB (2-bit)	ALUOp	RegDst	MemtoReg	IorD	PCSource
1 (Fetch)	1	1	1	0	0	0	01	00	X	X	0	00
2 (Decode)	0	0	0	0	0	0	11	00	X	X	X	X
3 (ALU)	0	0	0	0	0	1	00	10	X	X	X	X
4 (Write)	0	0	0	0	1	X	X	X	1	0	X	X

Example Control line for add instruction for each step

Cycle	PCWrite	IRWrite	MemRead	MemWrite	RegWrite	ALUSrcA	ALUSrcB (2-bit)	ALUOp	RegDst	MemtoReg	IorD	PCSource
1 (Fetch)	1	1	1	0	0	0	01	00	X	X	0	00
2 (Decode)	0	0	0	0	0	0	11	00	X	X	X	X
3 (ALU)	0	0	0	0	0	1	00	10	X	X	X	X
4 (Write)	0	0	0	0	1	X	X	X	1	0	X	X

BRANCH

Branch instructions require 3 cycles to complete. Let's imagine that we're executing a beq instruction.

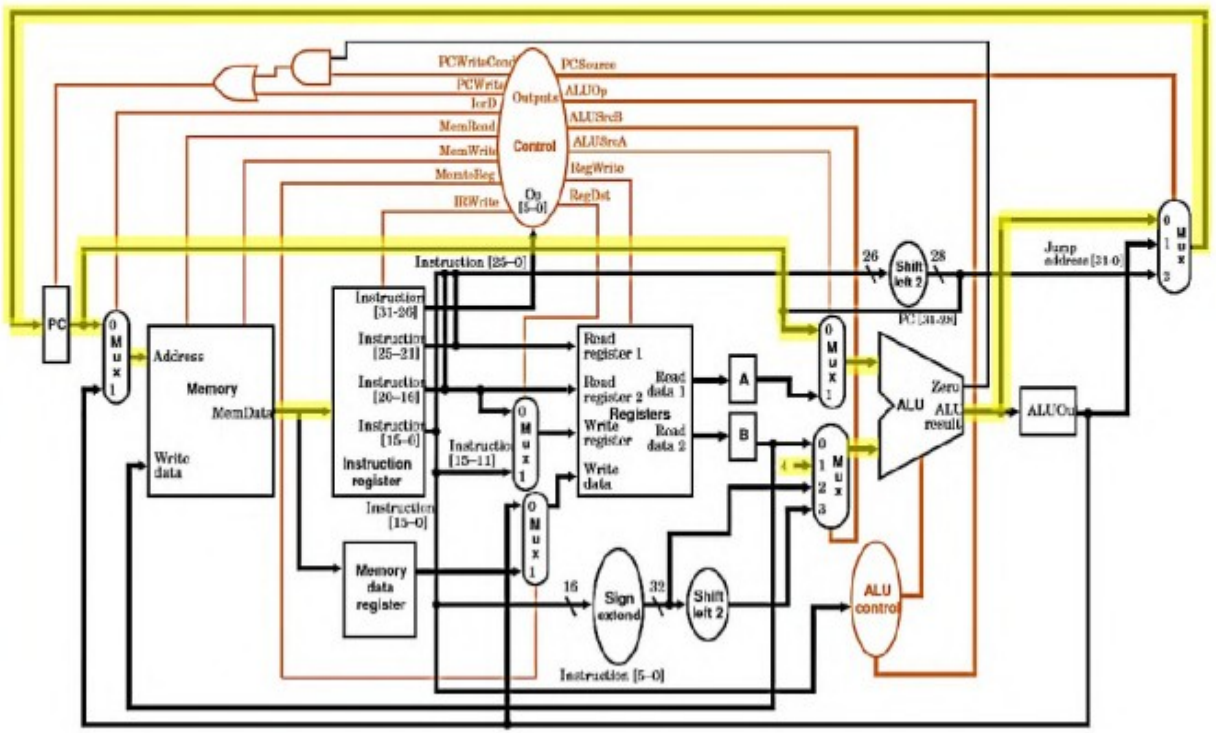
```
beq $s0, $s1, L1
```

which has the following fields:

opcode	rs	rt	immed
000100	10001	10010	XXXXXXXXXXXXXXXX

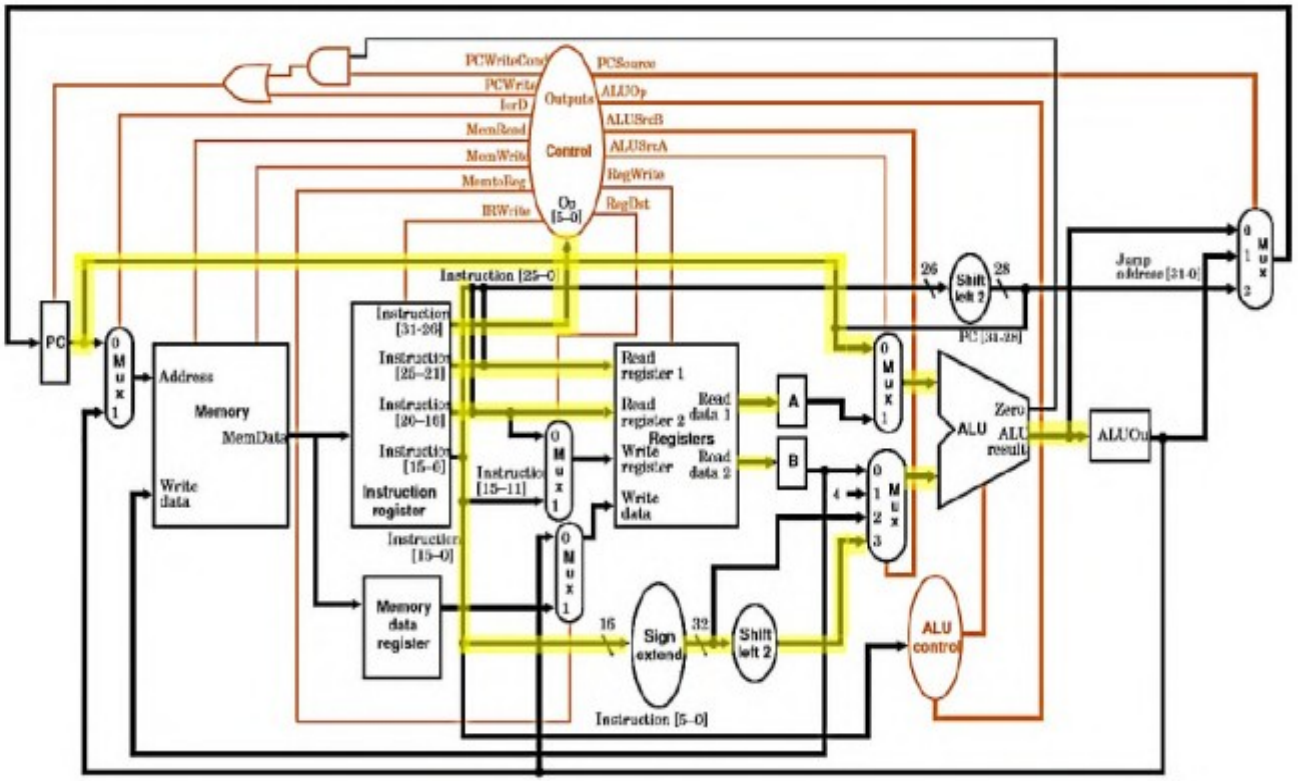
BRANCH: CYCLE 1

Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	00
ALUSrcB	01
ALUSrcA	0
RegWrite	0



BRANCH: CYCLE 2

Signal	Value
ALUOp	00
ALUSrcB	11
ALUSrcA	0



BRANCH: CYCLE 3

Signal	Value
ALUOp	01
ALUSrcB	00
ALUSrcA	1
PCSource	01
PCWriteCond	1