Computer Architecture:
MIPS Multi-Cycle Datapath

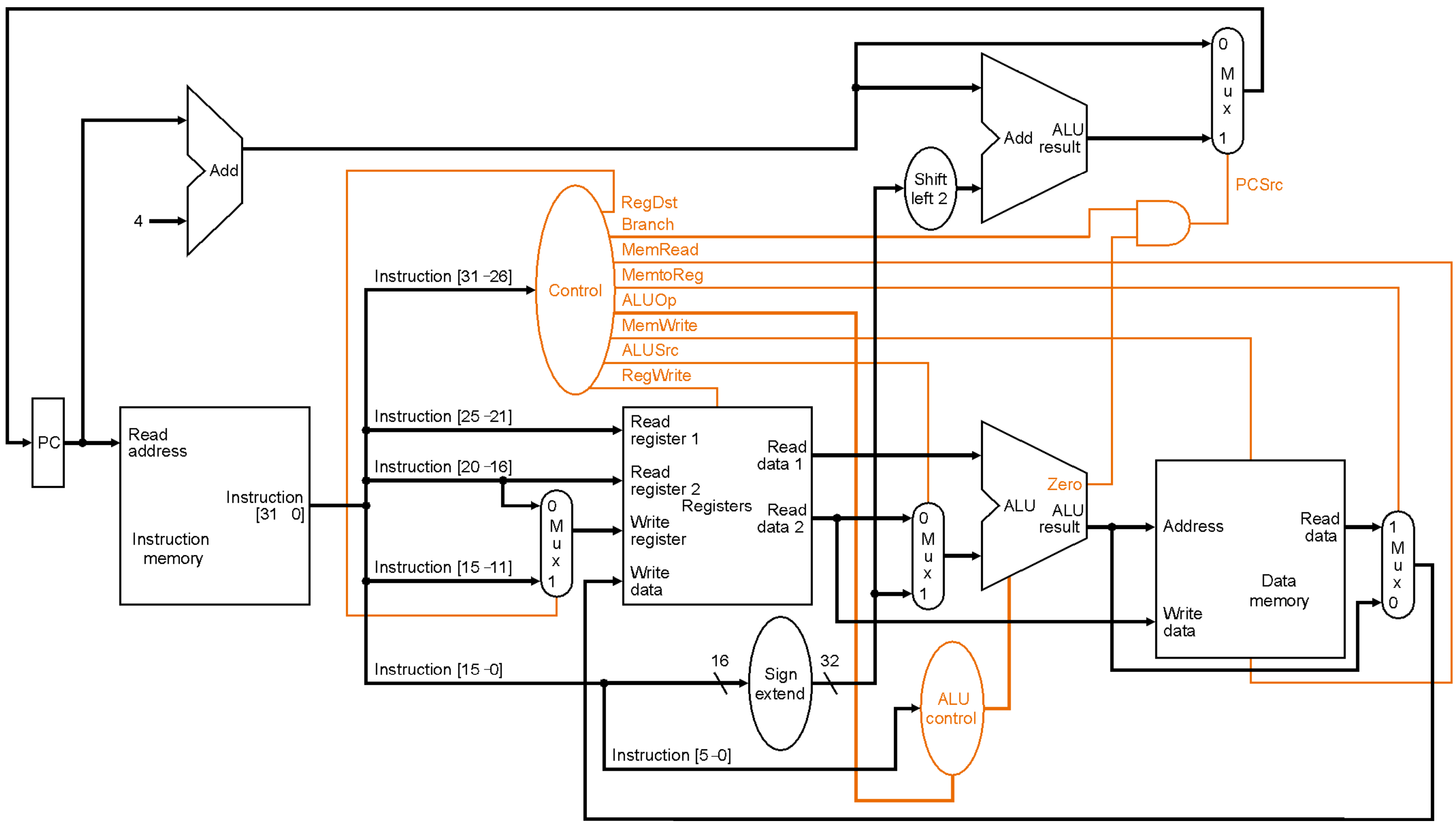
Hossein Asadi (asadi@sharif.edu)
Department of Computer Engineering
Sharif University of Technology
Spring 2024

Copyright Notice

- Some Parts (text & figures) of this Lecture adopted from following:
  - D.A. Patterson and J.L. Hennessy, "Computer Organization and Design: the Hardware/Software Interface" (MIPS), 6th Edition, 2020.
  - J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach", 6th Edition, Nov. 2017.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, CMU, Spring 2009.
  - "Computer Architecture & Engineering" handouts, by Prof. Kubiatowicz, UC Berkeley, Spring 2004.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, UWisc, Spring 2021.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC,

Quick Reminder from Previous Lecture

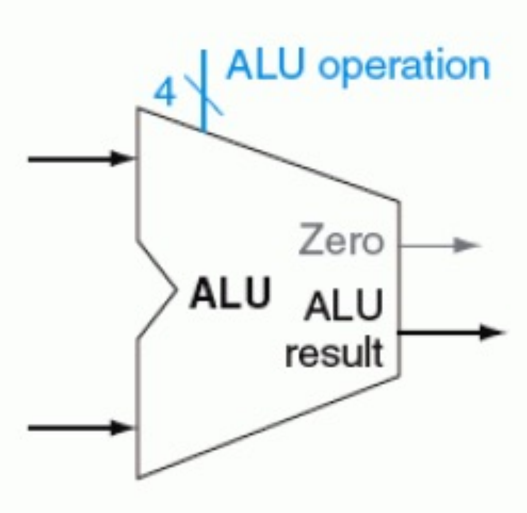
Adding Control Signals



ALU Control

- ALU Control Lines
  - Four control lines

ALU Control Lines	Function
0000	AND
0001	OR
0010	add
0110	sub
0111	set on less than
1100	NOR



ALU Control (cont.)

- ALUop
  – Used to distinguish R-type, lw/sw, beq

ALUop	Instruction	ALU Operation
00	Load/Store	Add
01	Beq	Sub
10	R-type	Determined by funct. Code (F5~F0)

ALU Control (cont.)

- ALU Control Inputs in terms of:
  - ALUop, funct field

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	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	set on less than	0111

ALU Control (cont.)

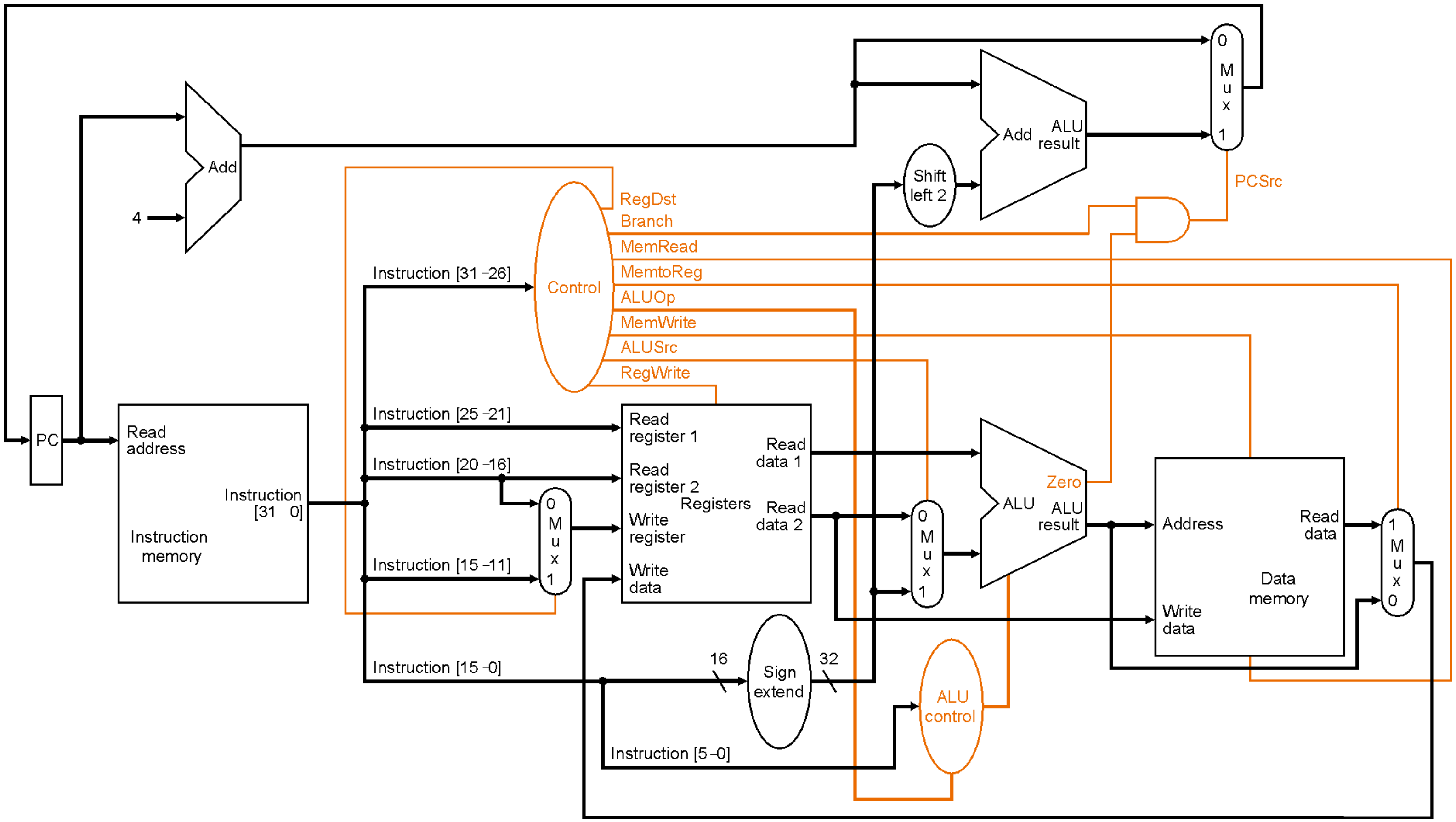
- Truth Table of ALU Control Inputs
  - 8 inputs
  - 4 outputs

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

Designing Main Control Unit

- Steps
  - Identify fields of instructions
  - Identify control lines needed for datapath
  - Figure out how to generate control lines from fields of instructions

Beq Control

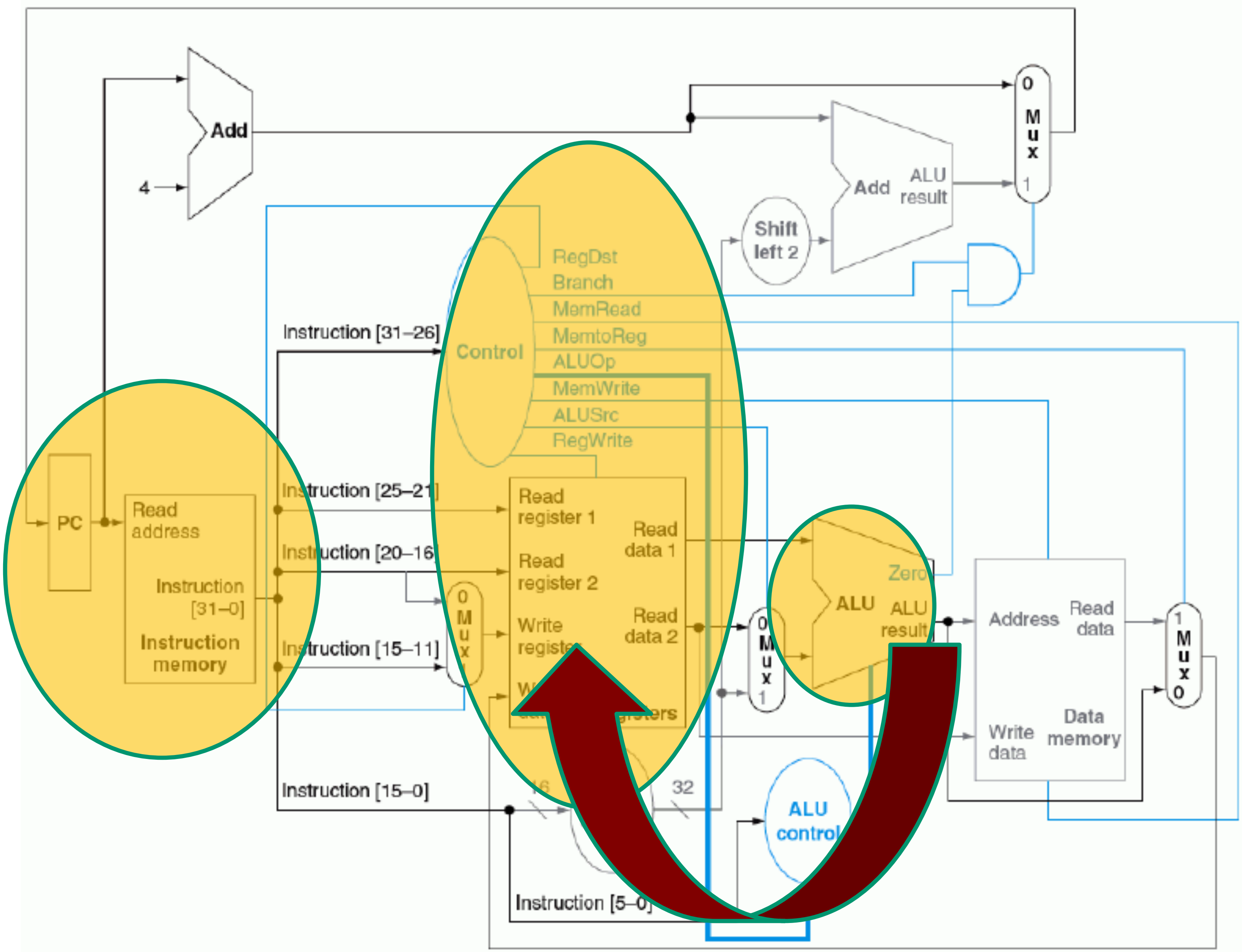


Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Operation of Datapath: R-Type

- Step 1:
  - Instruction fetched
  - PC incremented
- Step 2:
  - Two regs read from GPR
  - Main CU computes setting of control lines
- Step 3:
  - ALU control determined by funct. Code
  - Then, ALU operates on data read from GPR
- Step 4:
  - Results from ALU written into RF using bits 15:11

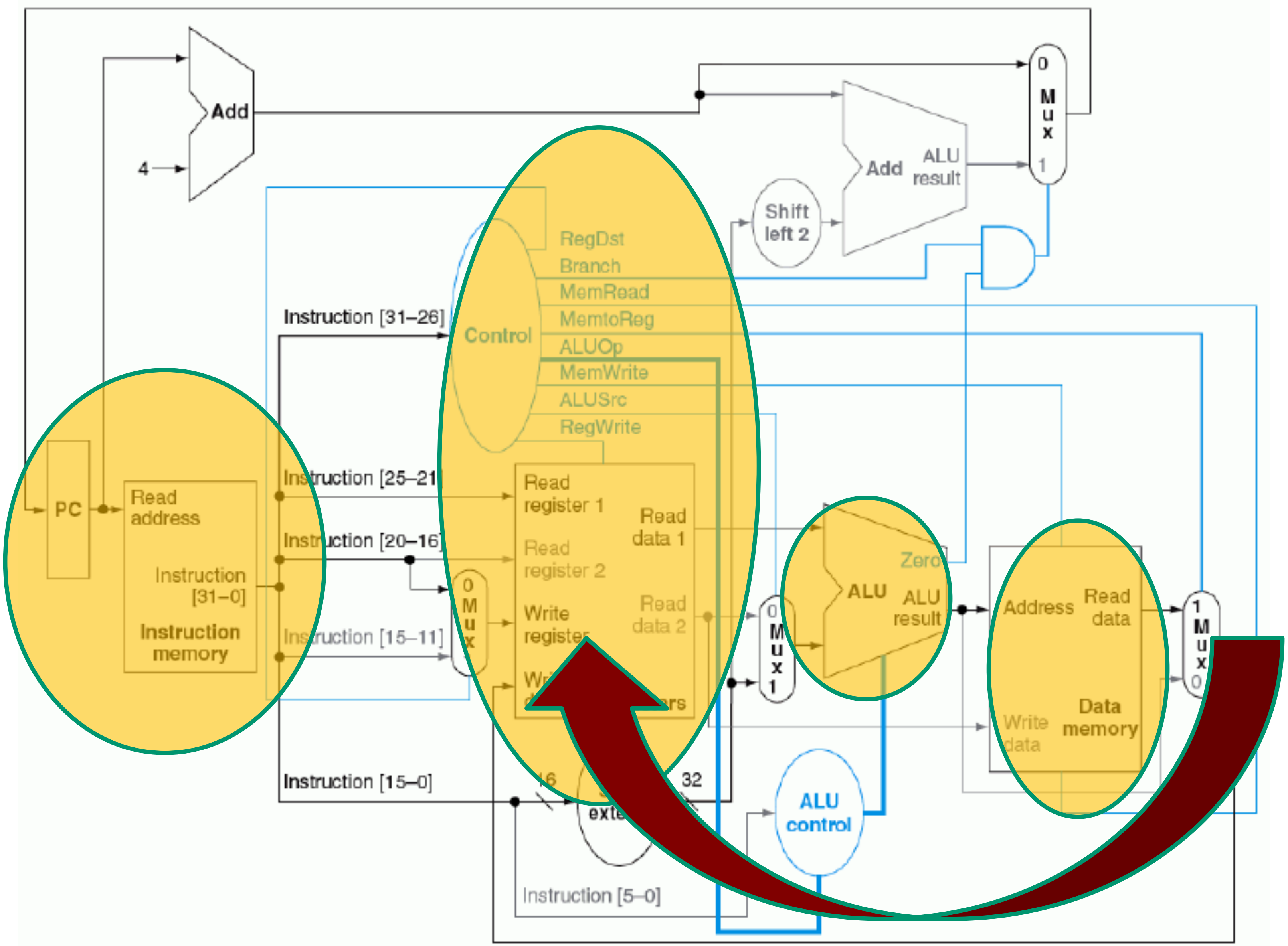
Operation of Datapath: R-Type (cont.)



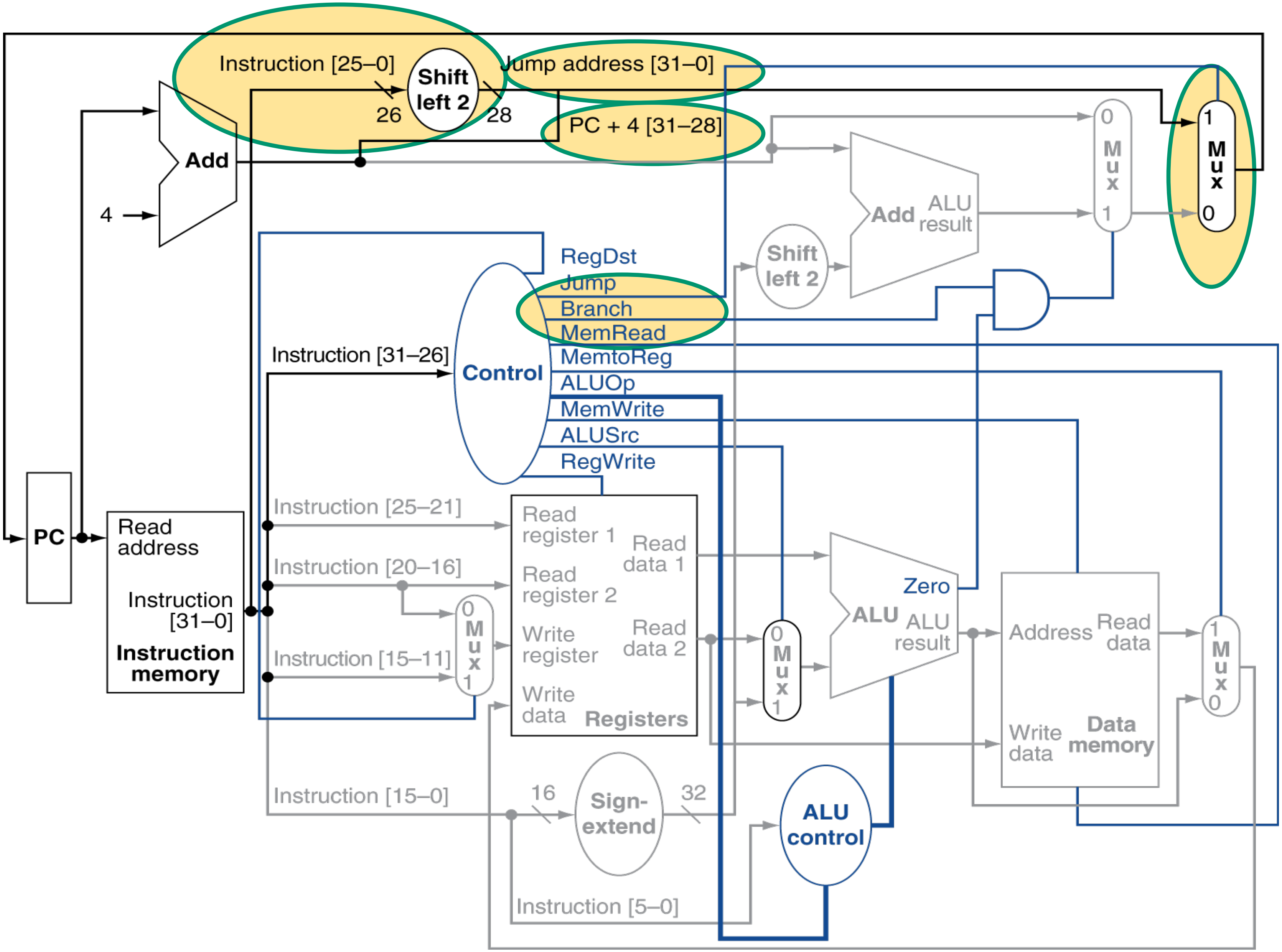
Operation of Datapath: Load

- Step 1:
  - Instruction fetched
  - PC incremented
- Step 2:
  - A reg read from GPR (e.g. $t1)
  - CU computes setting of control lines
- Step 3:
  - ALU computes target memory address
    - Based on $t1 and sign-extended value in bits 15:0
- Step 4:
  - 32-bit data read from Memory based on calculated addr.
- Step 5:
  - Data written into GPR (destination reg: bits 20:16)

Operation of Datapath: Load (cont.)



Jump Datapath



Our Lectur Today

Topics Covered Today

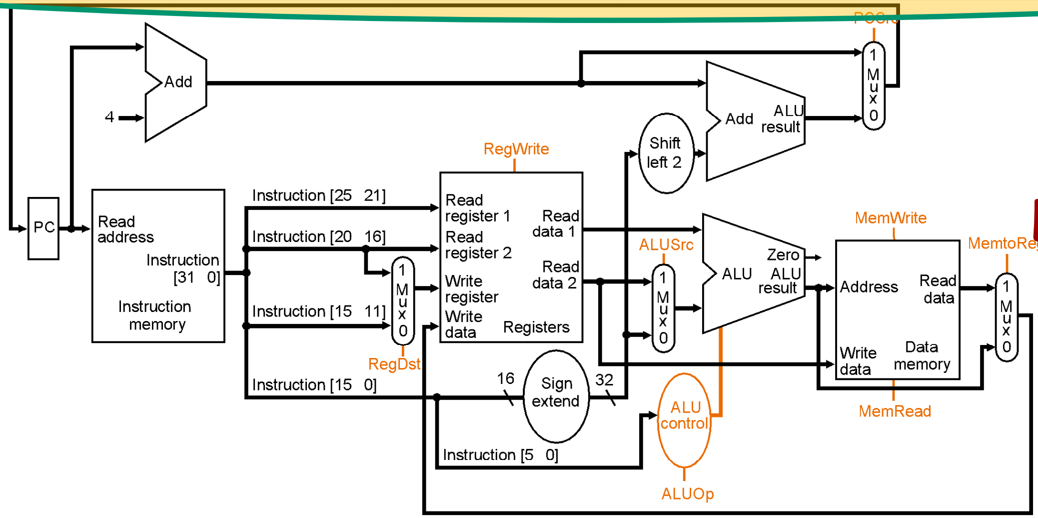
- **MIPS Multicycle Design**
  - **Shortcomings of single-cycle datapath**
  - **Basics of multicycle datapath**
  - **Major modules in multicycle**
  - **Control unit design for multicycle datapath**

Performance of Single-Cycle uArch

- Simple but Inefficient Performance
- Why?
  - Clock cycle determined by longest possible path
  - Clock cycles of all instructions same length
    - CPI = 1
- Longest Possible Path?
  - Load datapath

Single-Cycle CPU Clock Cycle Time

	I-cache	Decode, R-Read	ALU	PC update	D-cache	R-Write	Total
R-type	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0



Clock cycle time

= 4.7 + setup + hold

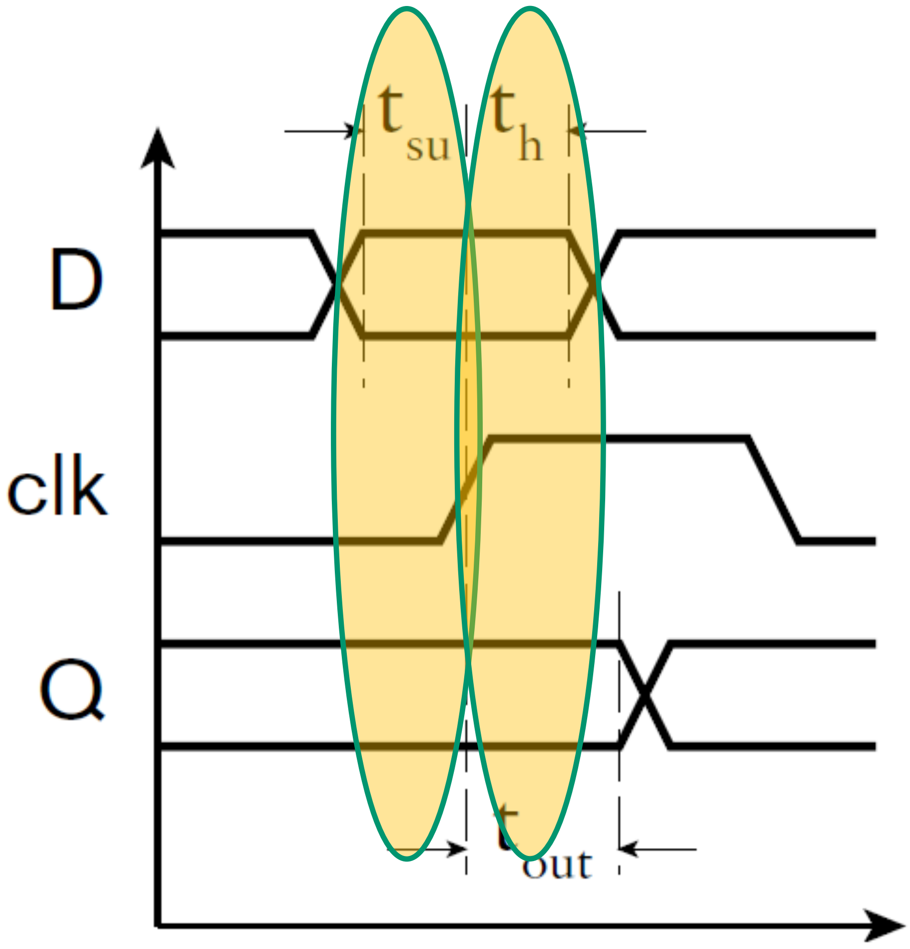
Load on critical path

Setup time?

Hold time?

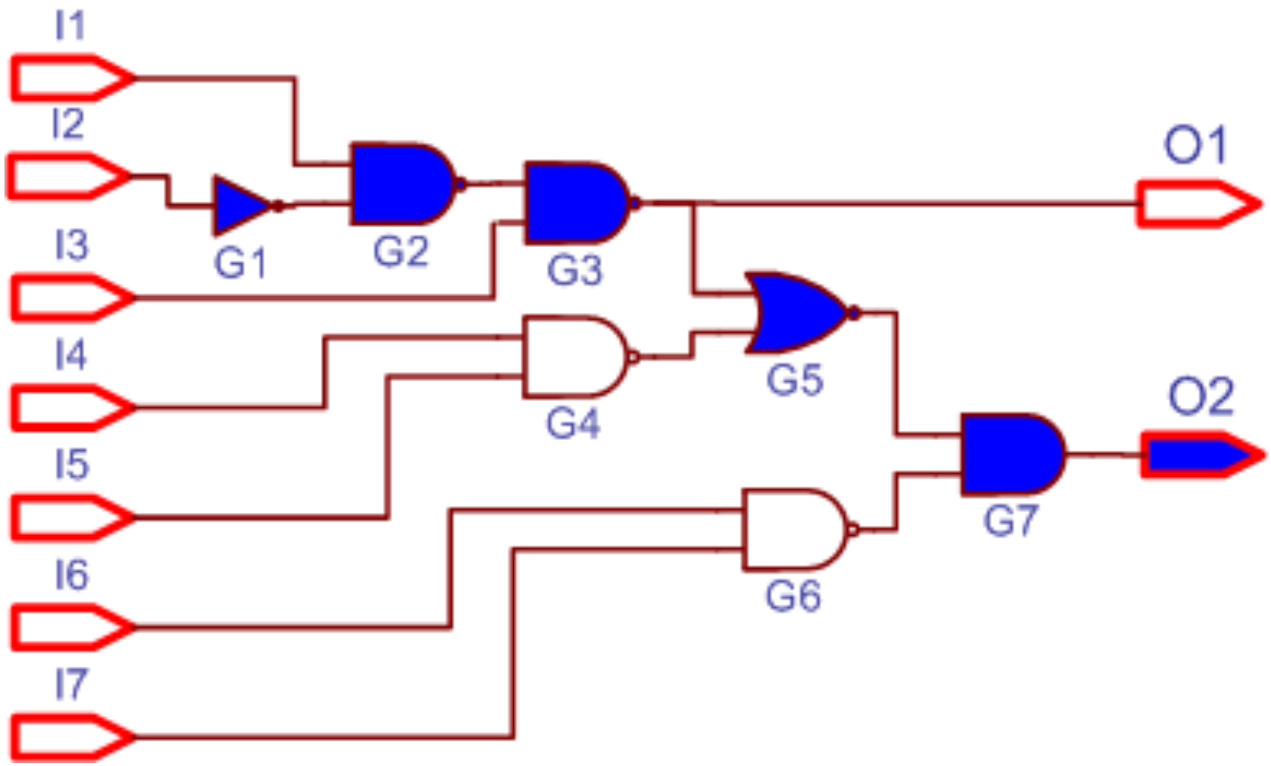
Critical path?

Setup Time & Hold Time



Critical Path Delay

- Definition:
  – A path through combinational circuit that takes as long or longer than any other



Multicycle Implementation

Goal: Balance amount of work done each cycle

	I cache	Decode, R-Read	ALU	PC update	D cache	R-Write	Total
R-type	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0

- Load needs 5 cycles
- Store and R-type need 4
- beq needs 3

Will Multi-Cycle Design be Faster?

	I cache	Decode, R-read	ALU	PC update	D cache	R-write	Total
R-type	1	1	.9	-	-	.8	3.7
Load	1	1	.9	-	1	.8	4.7
Store	1	1	.9	-	1	-	3.9
beq	1	1	.9	.1	-	-	3.0

Let's assume setup + hold time = 100ps = 0.1 ns

Single Cycle Design:

Clock cycle time = 4.7 + 0.1 = 4.8 ns

time/inst = 1 cycle/inst * 4.8 ns/cycle = 4.8 ns/inst

Multicycle Design:

Clock cycle time = 1.0 + 0.1 = 1.1

time/inst = CPI * 1.1 ns/cycle

Will Multi-Cycle Design be Faster? (cont.)

	Cycles needed	Instruction frequency
R-type	4	60%
Load	5	20%
Store	4	10%
beq	3	10%

What is **CPI** assuming this instruction mix?

CPI = 4* 0.6 + 5*0.2 + 4*0.1 + 3*0.1 = 4.1

Let's assume setup + hold time = 0.1 ns

Single Cycle Design:

Clock cycle time = 4.7 + 0.1 = 4.8 ns

time/inst = 1 cycle/inst * 4.8 ns/cycle = 4.8 ns/inst

Multicycle Design:

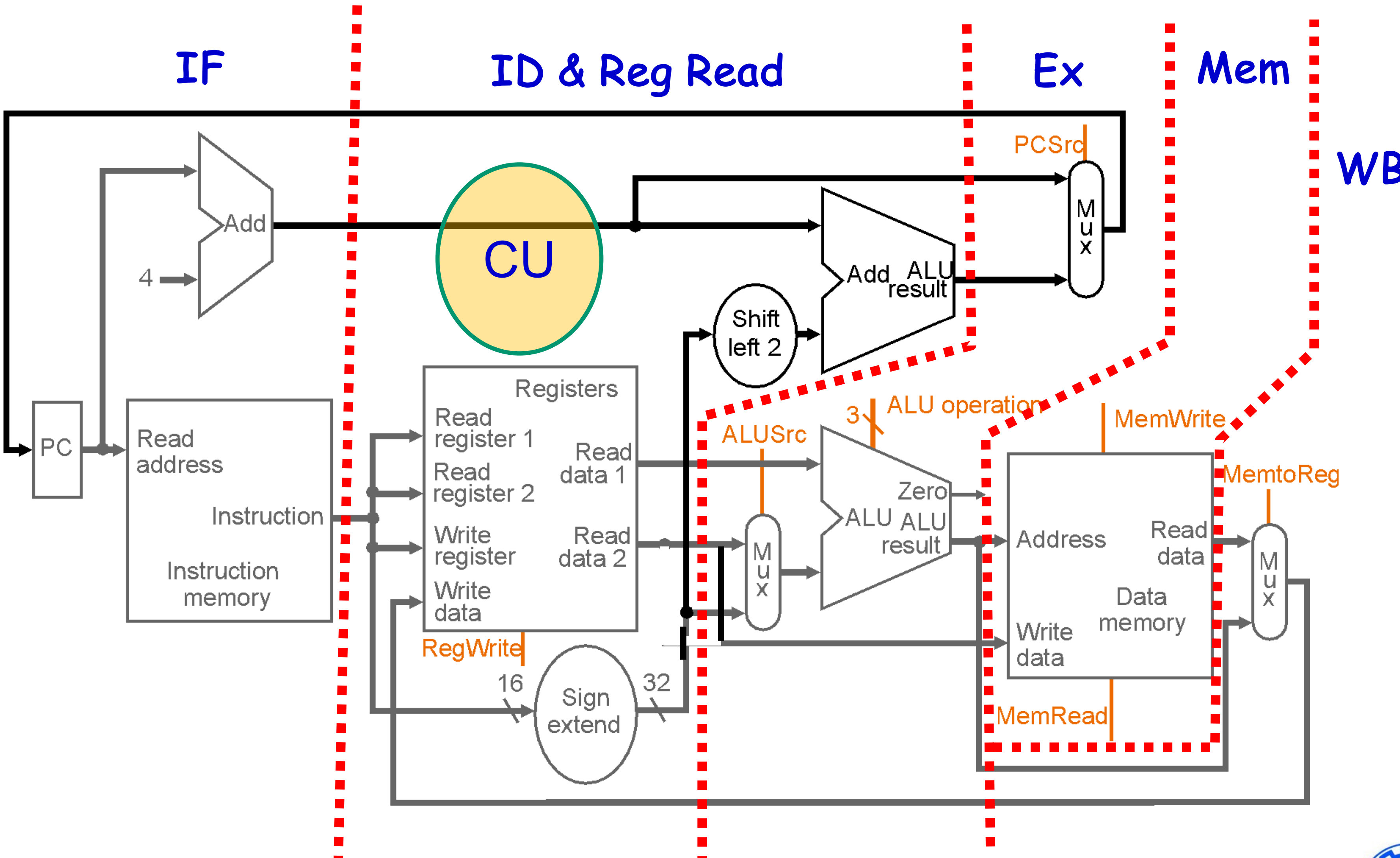
Clock cycle time = 1.0 + 0.1 = 1.1

time/inst = CPI * 1.1 ns/cycle = 4.1 * 1.1 = 4.5

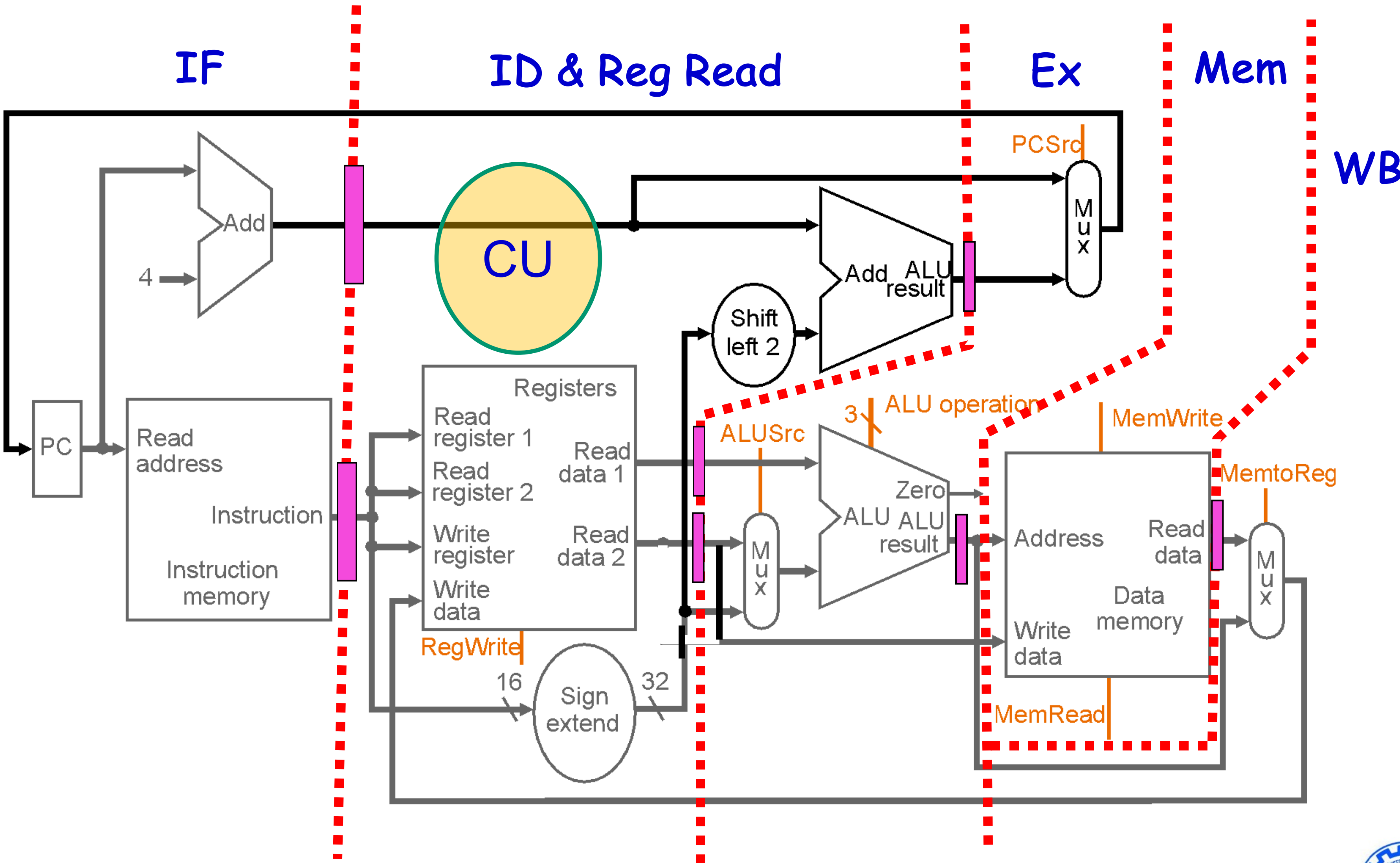
Will Multi-Cycle Design be Faster? (cont.)

- Much Smaller Clock Cycle Time
  - Compared to single-cycle datapath
- Possibly Faster Runtime
  - Compared to single-cycle datapath
- Depends on:
  - How partitioning is performed
  - Frequency of instructions in benchmark programs

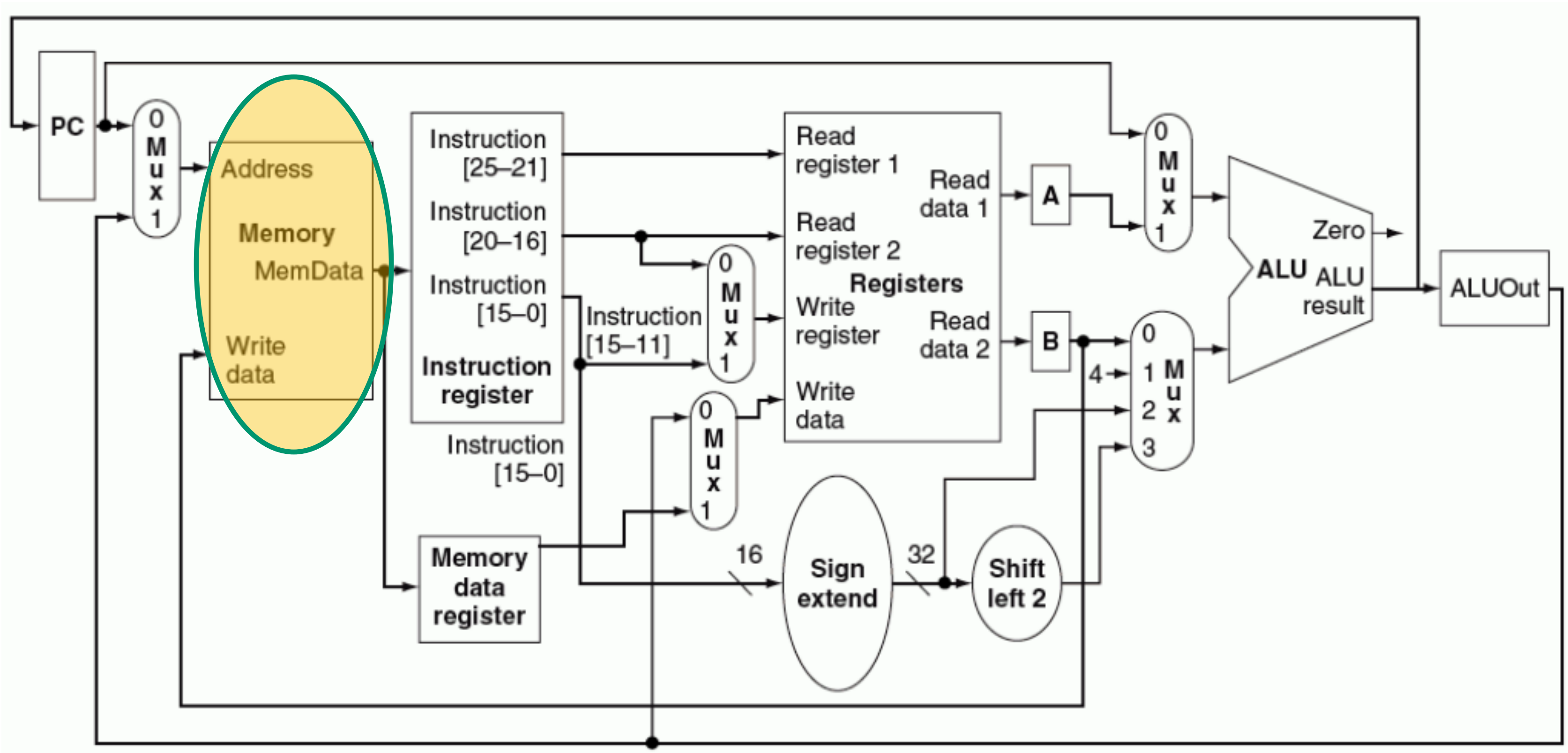
Partitioning Single-Cycle Design



Where to Add Registers?

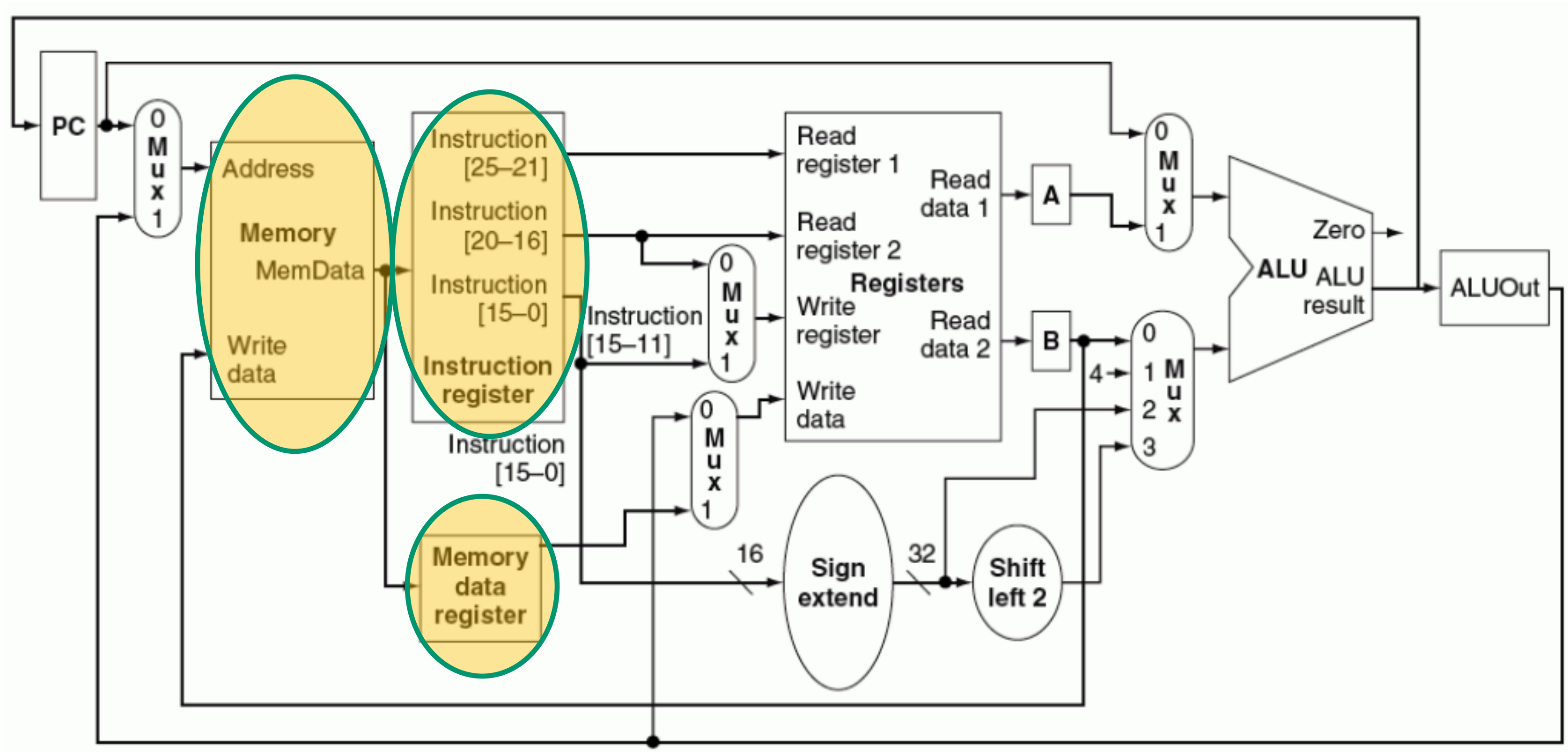


Multicycle Datapath



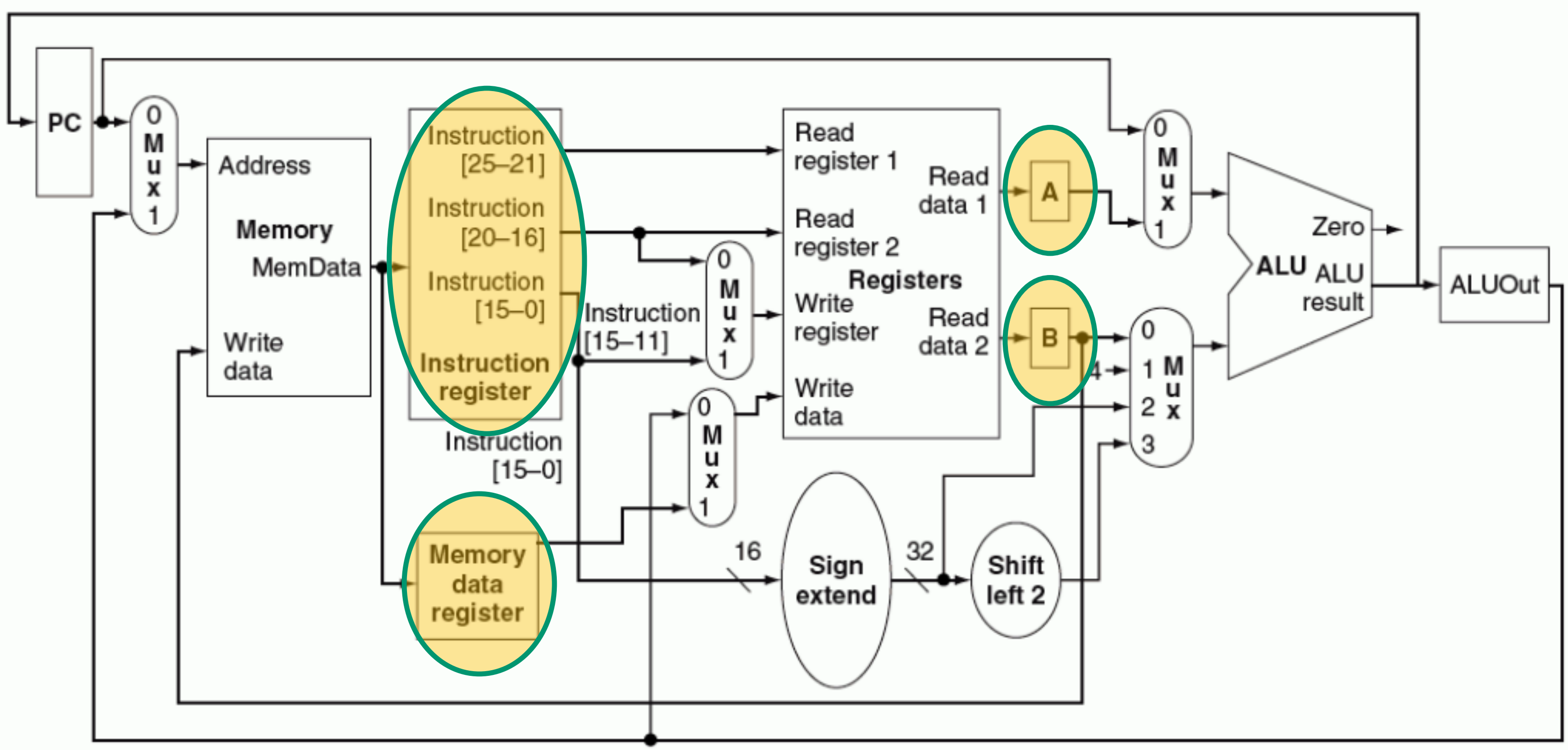
- Unified Memory
  - Used as both I-cache & D-cache
- Combines address busses of single-cycle datapath
  - Uses a MUX to select PC or ALU output

Multicycle Datapath (cont.)



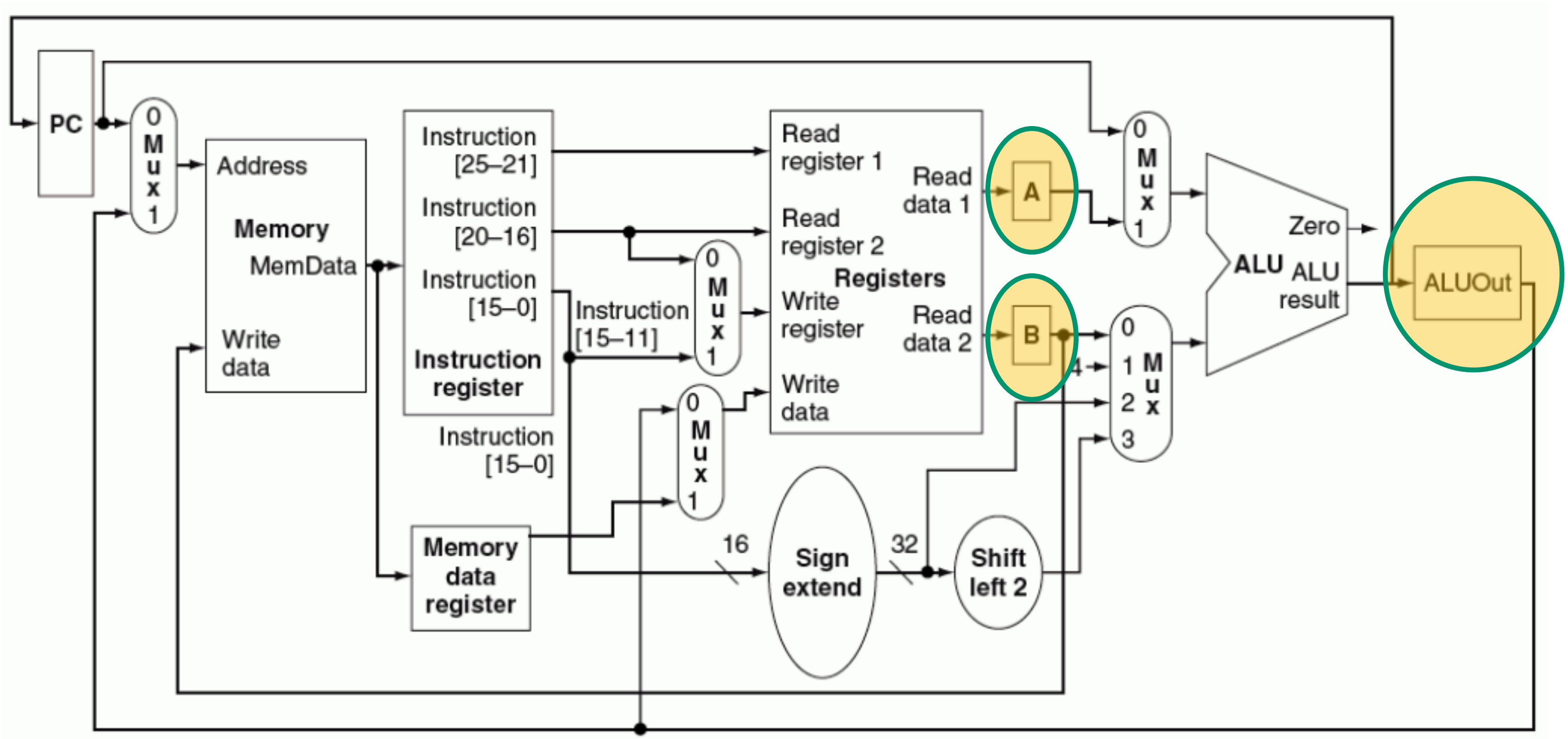
- Instruction Register
  - IR ← Mem[PC]
- Memory Data Register
  - MDR ← Mem[ALUOut]

Multicycle Datapath (cont.)



- **Reg A & B**
  - A 🡐 GPR[IR[25:21]]
  - B 🡐 GPR[IR[20:16]]

Multicycle Datapath (cont.)

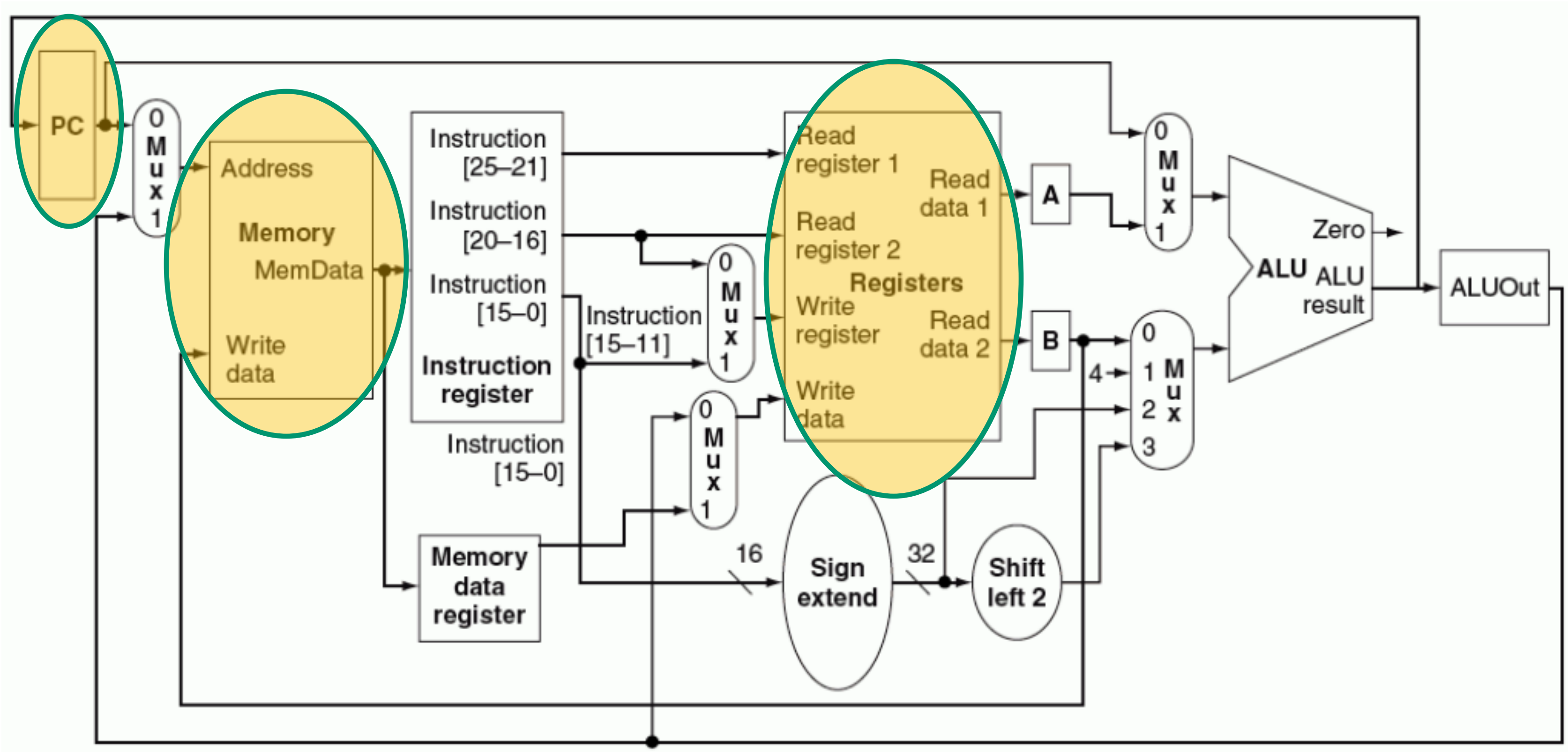


- •ALUOut ← ALU result
- •ALUOut is then either
  - Written to GPR
  - Or used as an address for Memory

Multi-Cycle vs. Single-Cycle Datapath

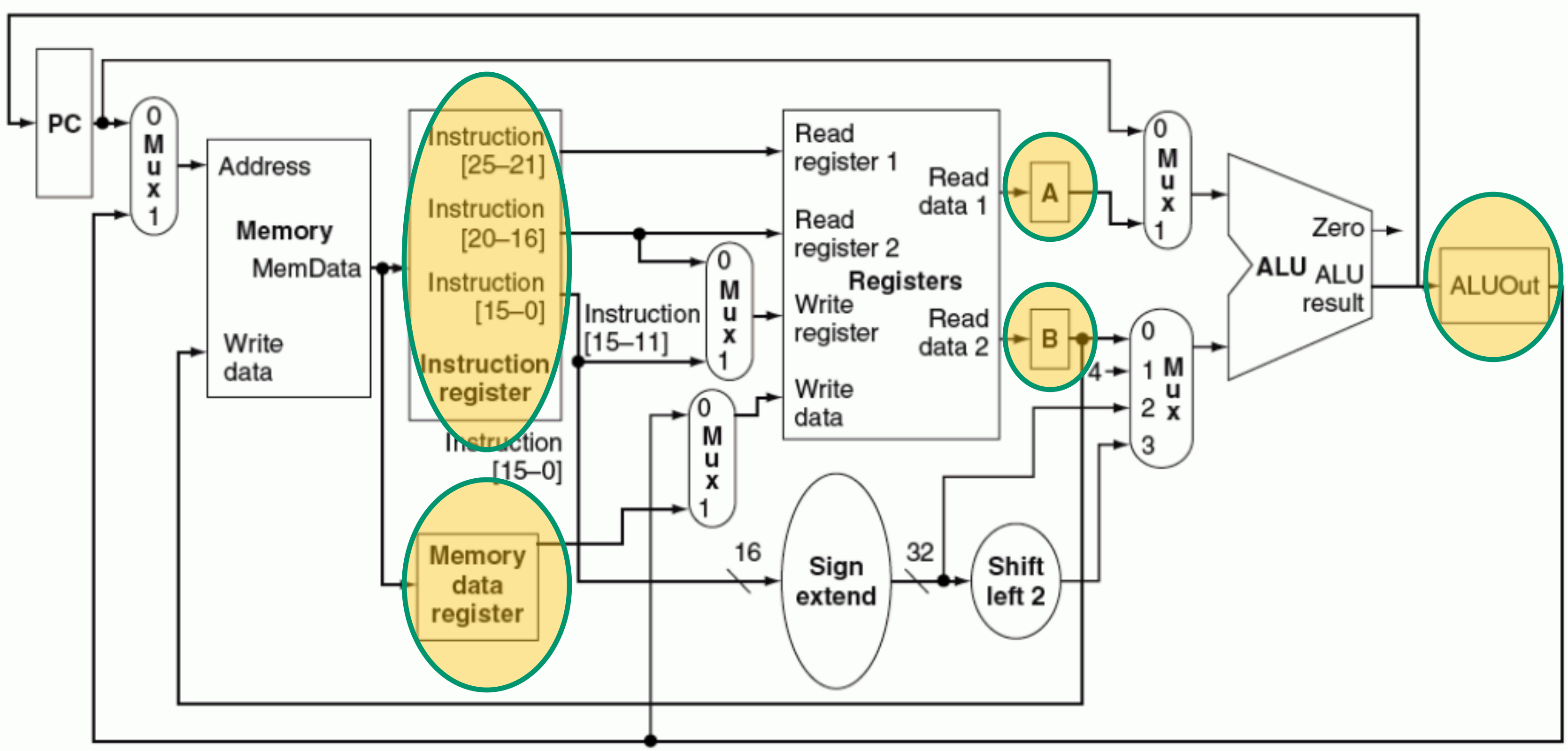
- Hardware Elements
  - Single memory unit
    - Used for both in instruction & data
    - Instruction & data must be accessed in different clock cycles
  - Single ALU unit
    - Rather than Using ALU and two adders
  - One or more registers added after every major functional unit

Multicycle Datapath (cont.)



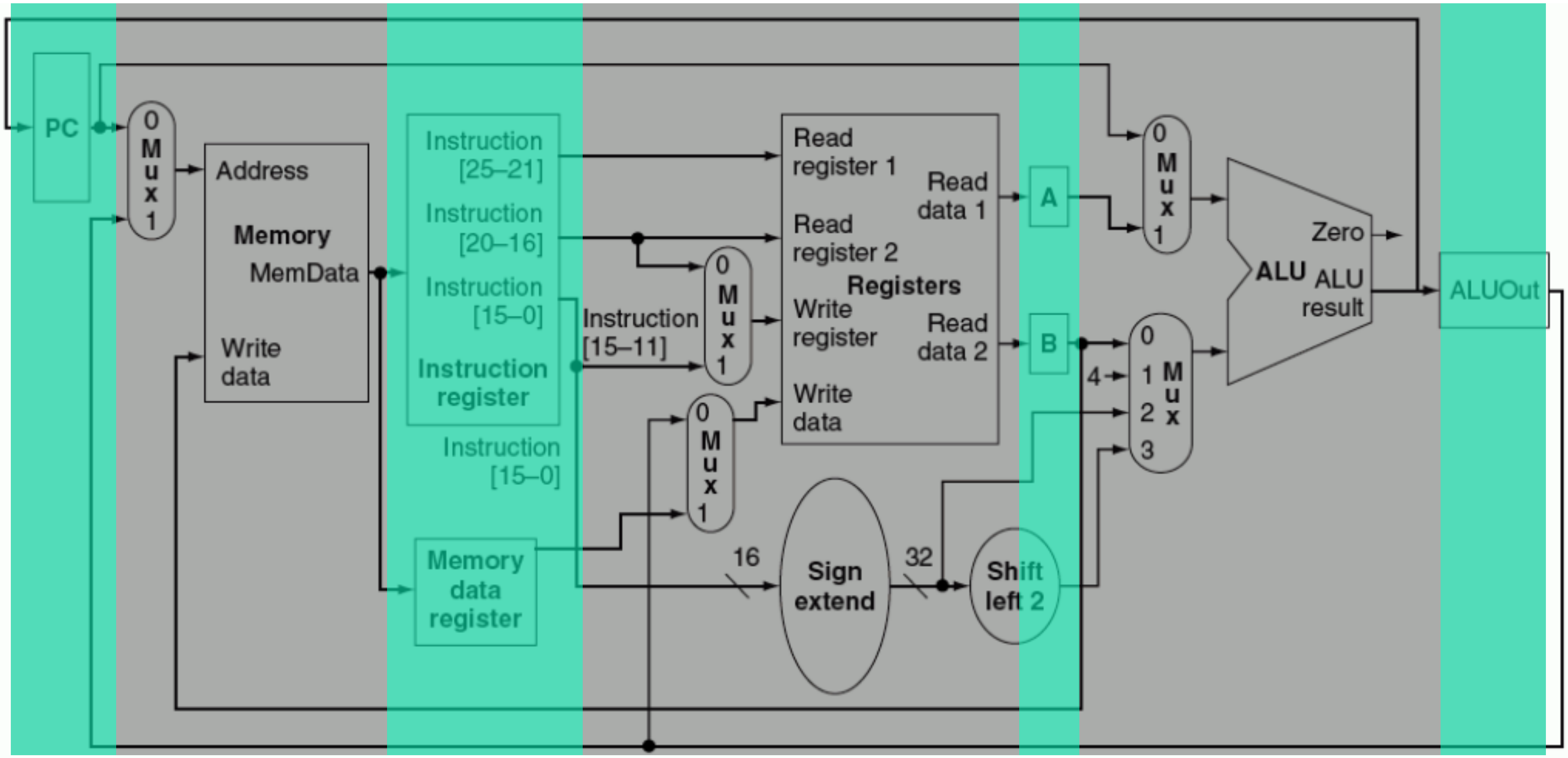
- User-Visible State Elements
  - PC
  - Memory
  - Register file

Multicycle Datapath (cont.)



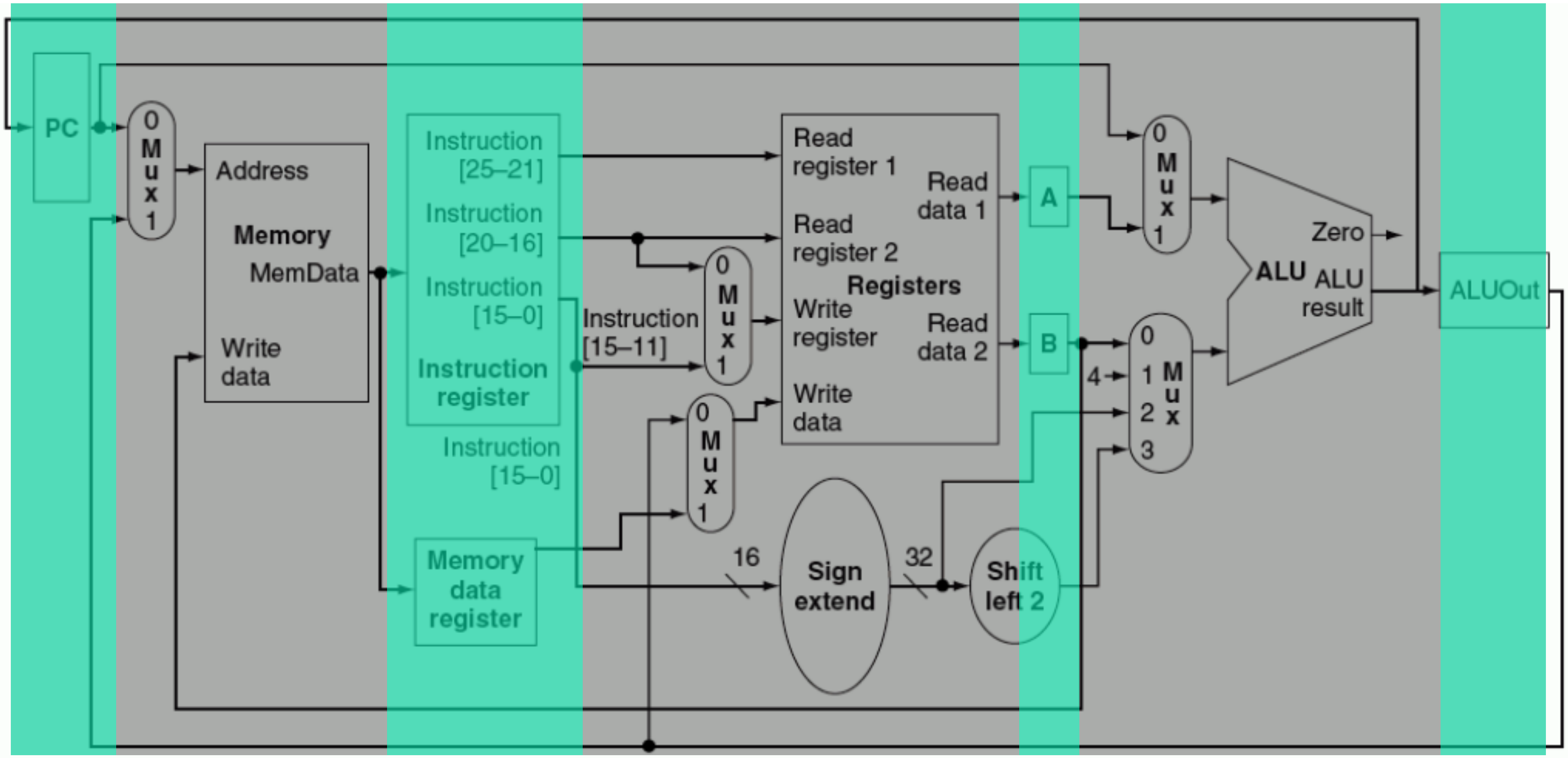
- Non-User-Visible State Elements
  - Instruction Register (IR)
  - Memory Data Register (MDR)
  - Reg A & Reg B
  - ALUout

Multicycle Datapath (cont.)



- Note: registers hold data only between a pair of adjacent clock cycles
- Question: Do we need to have write or read control signals for registers, memory, & GPR?

Multicycle Datapath (cont.)

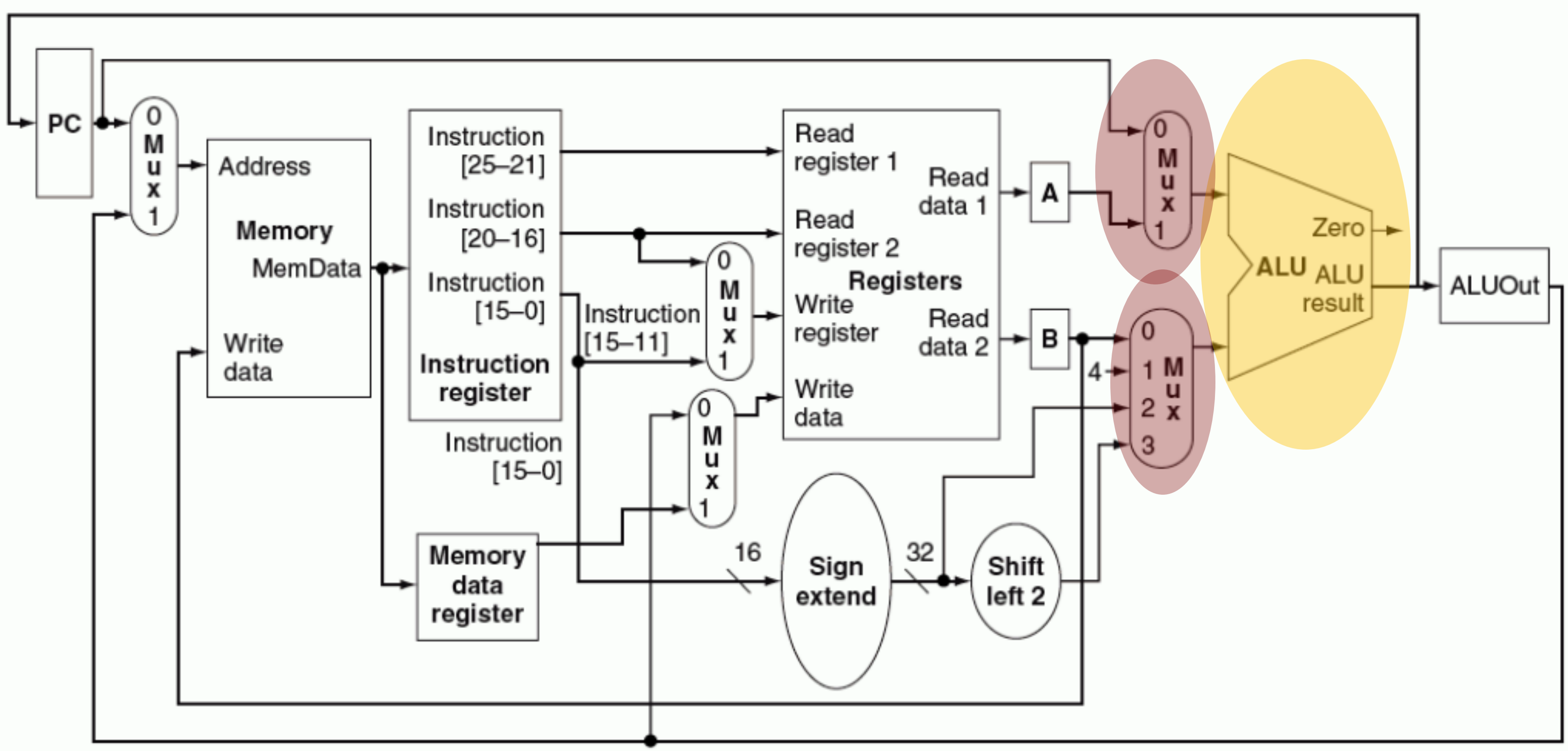


- No WR/RD control signal for non-visible regs (MDR,A,B, …)
  - WR=1, RD=1
- But IR needs to hold instr. until end of exec. of that instr.
- How about PC, memory, and GPR?
- How about read signal?

Read & Write Control Signals

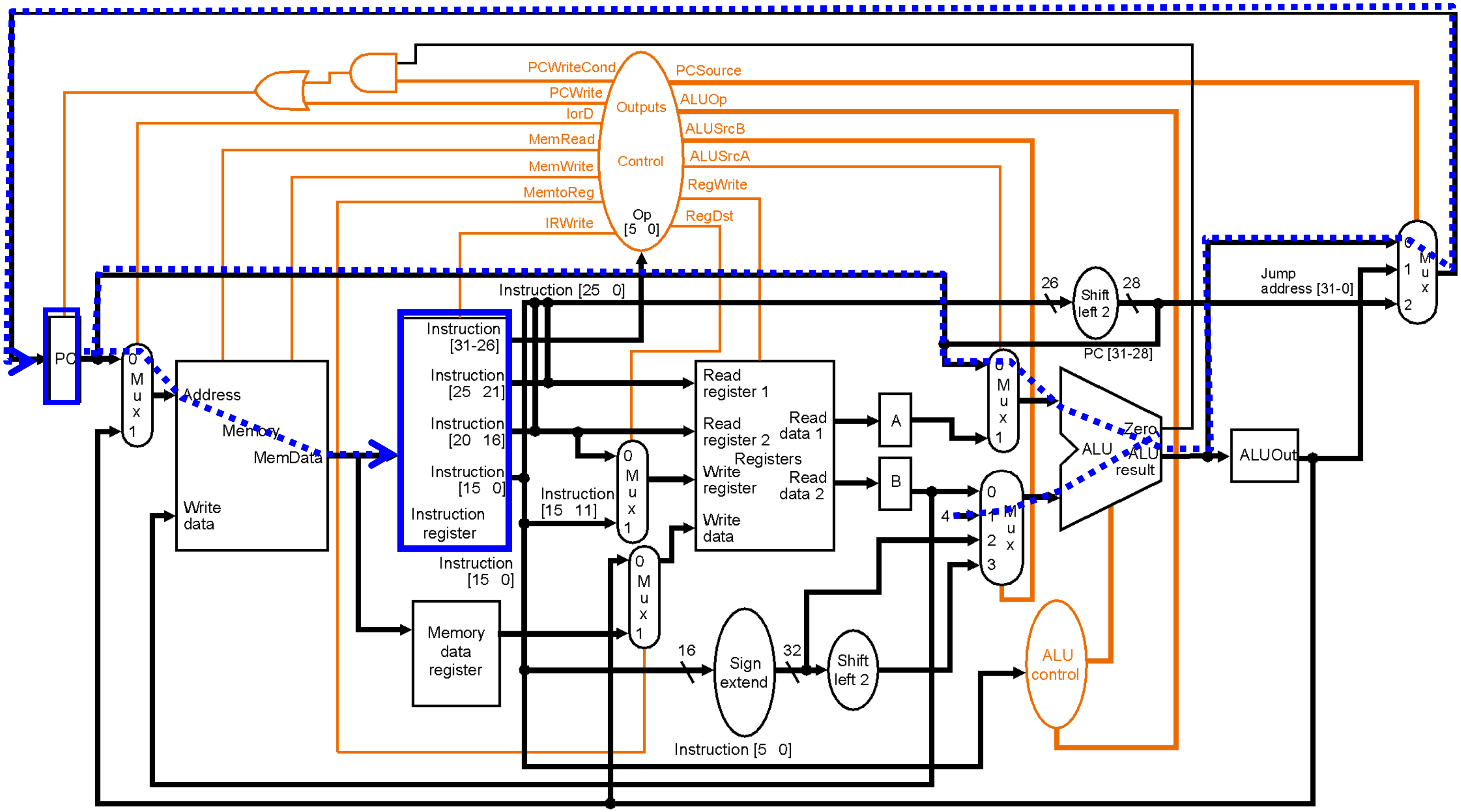
- Memory
  - Write signal required
  - Read signal required
    - If simultaneous read and write not possible
    - Twice decode circuitry for simultaneous RD/WR
- PC
  - If write signal = 1 ➔
    - PC incremented by 4 in IF & ID cycles
    - PC may capture wrong address in other cycles

Multicycle Datapath (cont.)



- Three ALUs replaced by a single ALU
- ALU must accommodate all inputs
  - Which go to three ALUs in single-cycle datapath
  - A op B / PC+4 / PC+addr. / A+immediate

Cycle 1: Instruction Fetch

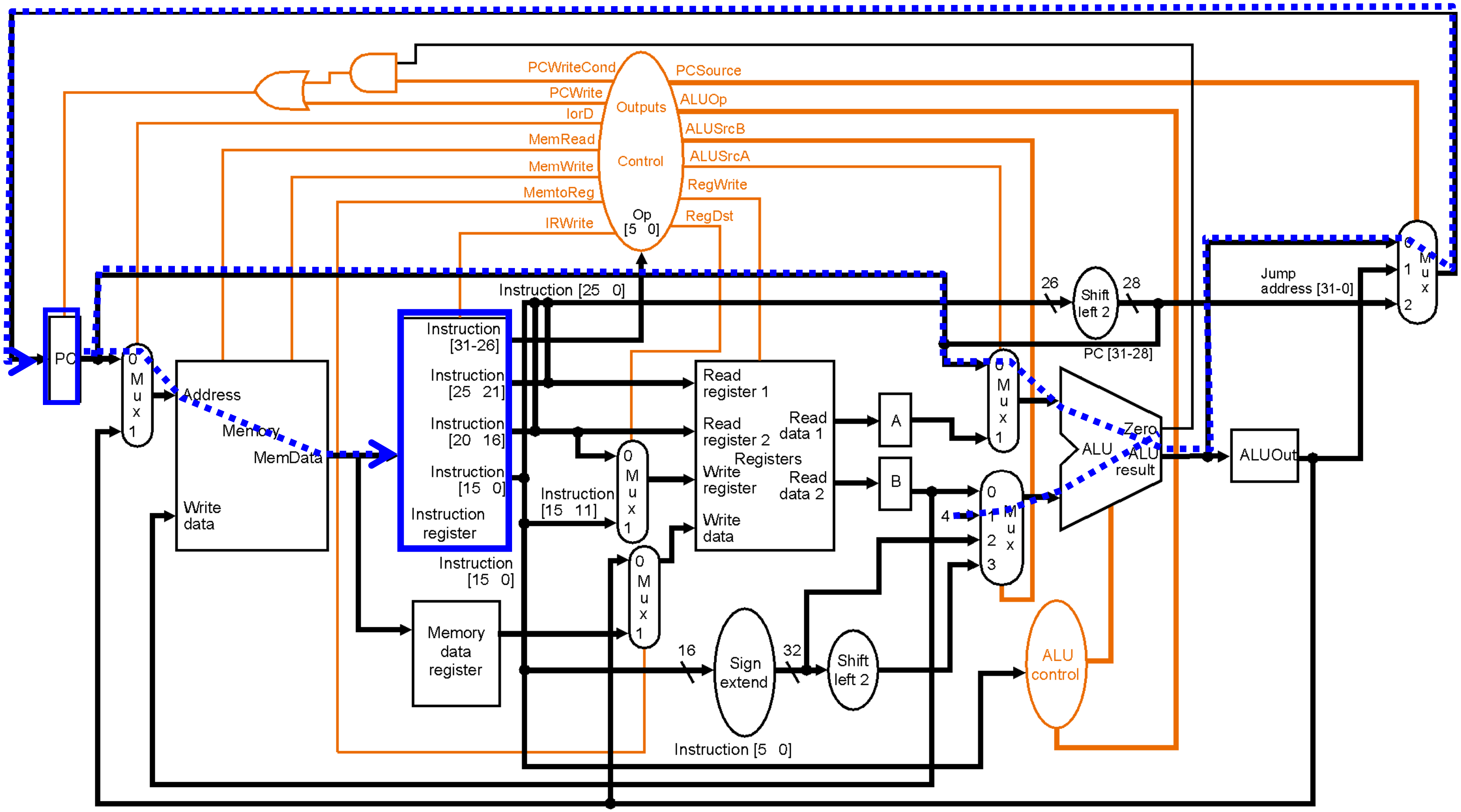


Datapath:

IR <= Mem[PC]

PC <= PC + 4

Cycle 1: Instruction Fetch



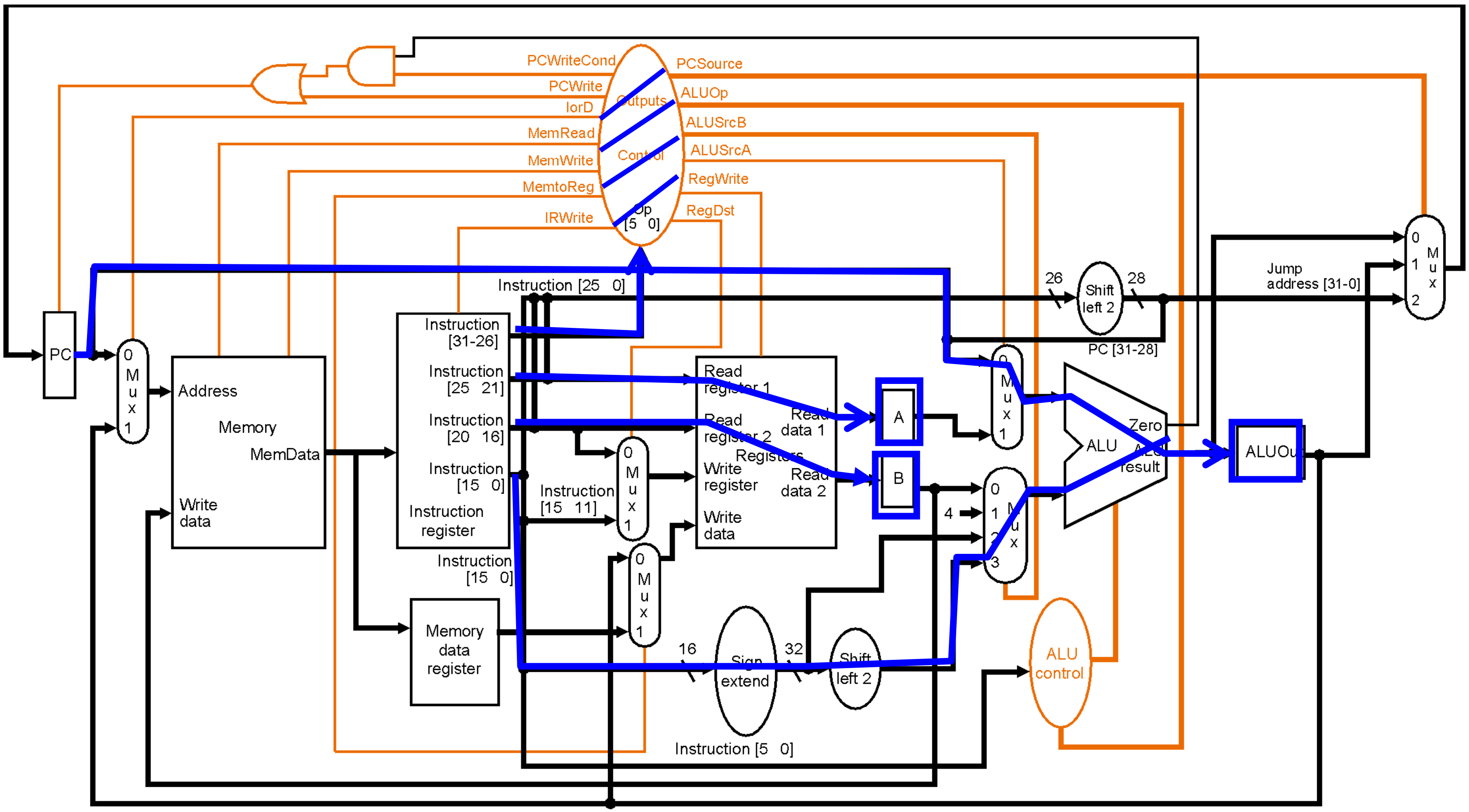
Control:

IorD=0, MemRead=1, MemWrite=0, IRwrite=1, ALUsrcA=0
ALUsrcB=01, PCWrite=1, ALUop=00, PCsource=00

Control for IF Cycle

MemRead
ALUsrcA = 0
IorD = 0
IRwrite
ALUsrcB = 01
ALUop = 00
Pcwrite
PCsource = 00

Cycle 2: ID & RF Cycle



A <= GPR[IR[25-21]]

B <= GPR[IR[20-16]]

ALUout <= PC + (sign-extend (IR[15-0]) << 2)

Cycle 2: ID & RF Cycle

A <= GPR[IR[25-21]]
B <= GPR[IR[20-16]]
ALUout <= PC + (SignEx(IR[15-0]) << 2)

- Question 1:
  - We fetch A & B from GPR even though we don’t know if they will be used.
  - Why?

Cycle 2: ID & RF Cycle

A <= GPR[IR[25-21]]

B <= GPR[IR[20-16]]

ALUout <= PC + (SignEx(IR[15-0]) << 2)

- Question 2:
  - We compute target address even though we don't know if it will be used.
    - Operation may not be branch
    - Even if it is, branch may not be taken
  - Why?

Cycle 2: ID & RF Cycle

A <= GPR[IR[25-21]]
B <= GPR[IR[20-16]]
ALUout <= PC + (SignEx(IR[15-0]) << 2)

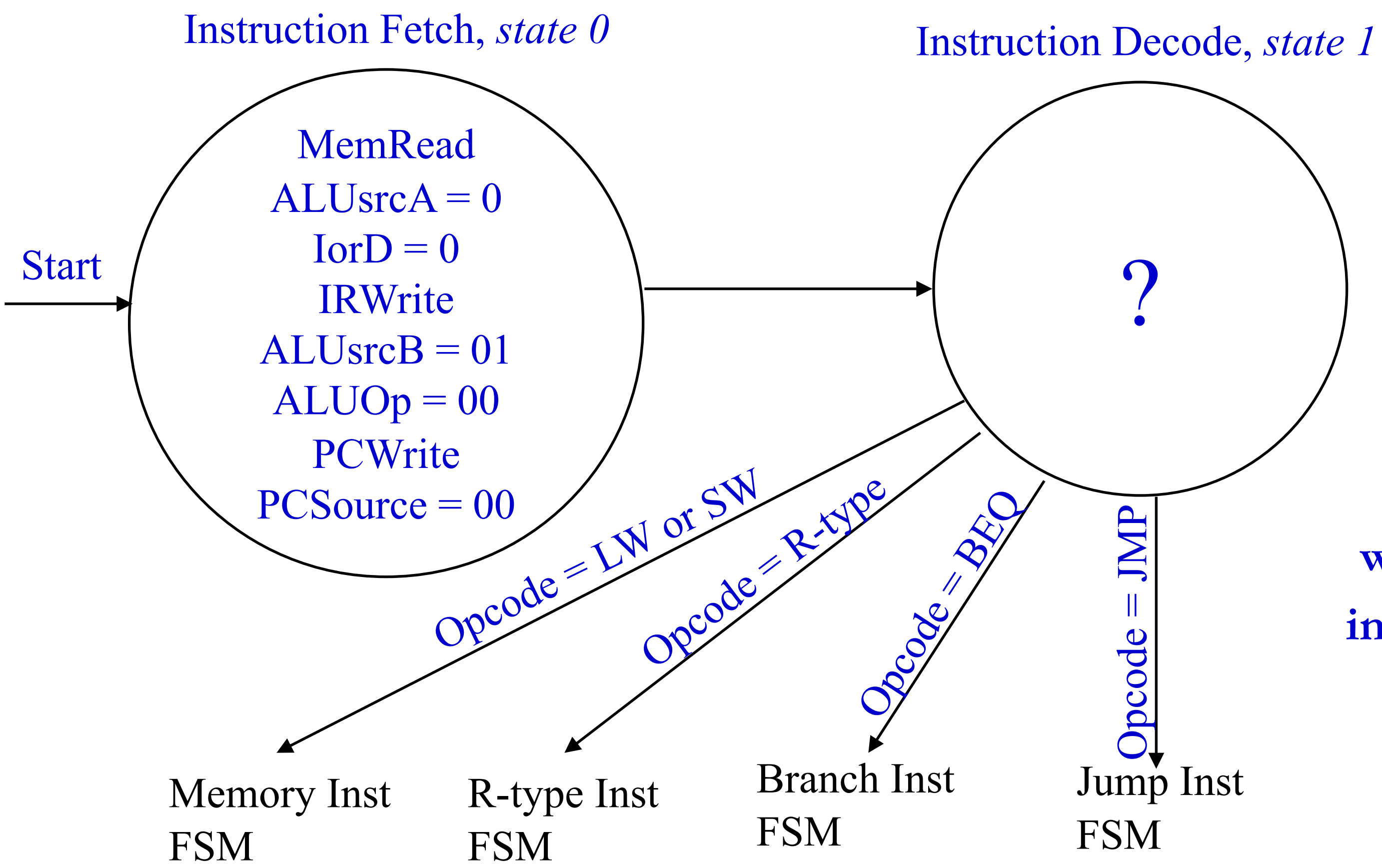
- Question 3:
  - Control signals computed in Cycle 2. However, IorD signal used in Cycle 1. How this is possible?

Cycle 2: ID & RF Cycle

A <= GPR[IR[25-21]]
B <= GPR[IR[20-16]]
ALUout <= PC + (SignEx(IR[15-0]) << 2)

- Answer:
  - Everything up to this point must be instruction-independent
    - Because we haven't decoded instruction
  - GPR and ALU are available in cycle 2 so we can use them up to fetch A & B and to calculate target branch address

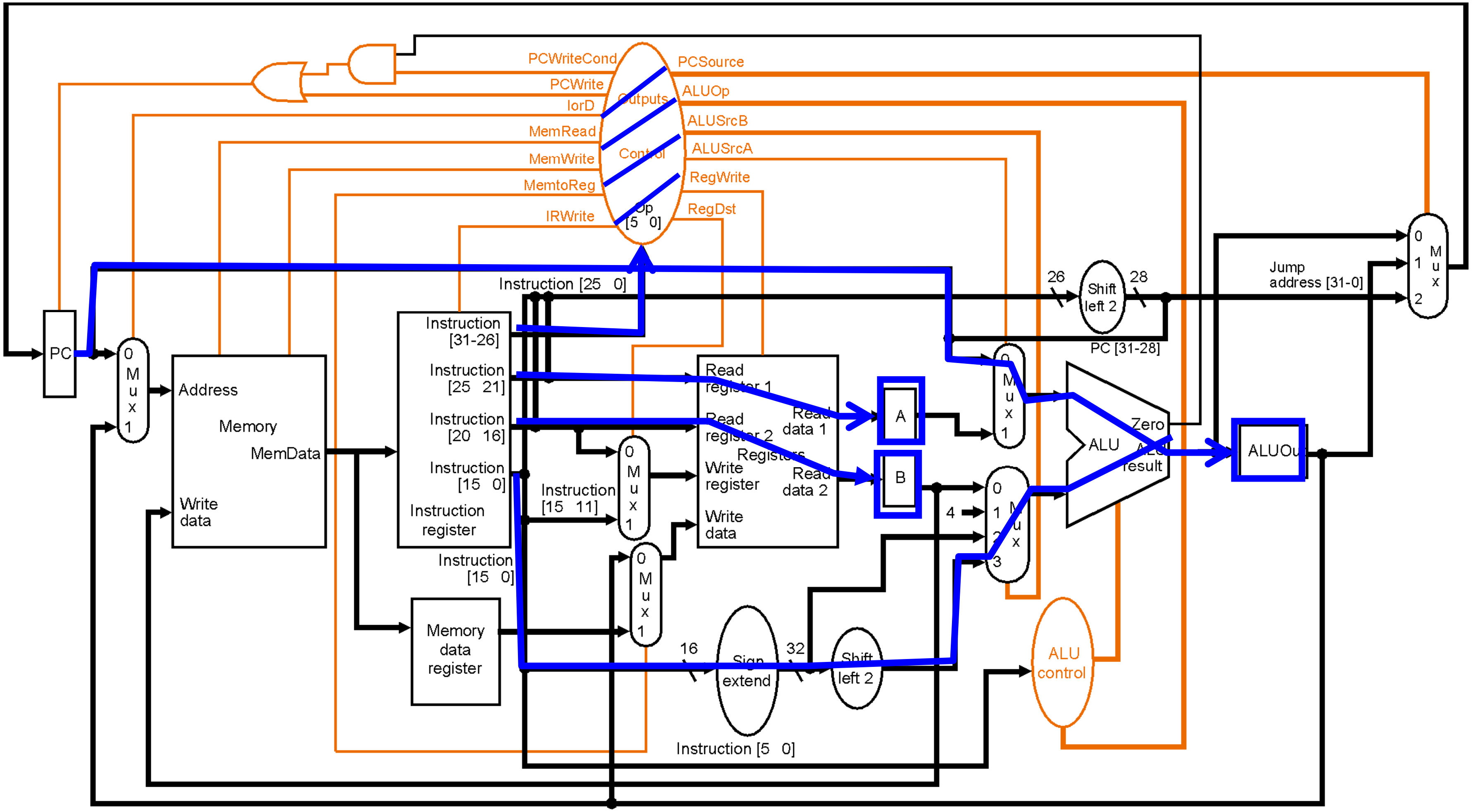
Control for First Two Cycles



After cycle two, we can treat different instructions separately

- Specification of Control
  - Using a Finite State Machine (FSM)

Cycle 2: ID & RF Cycle

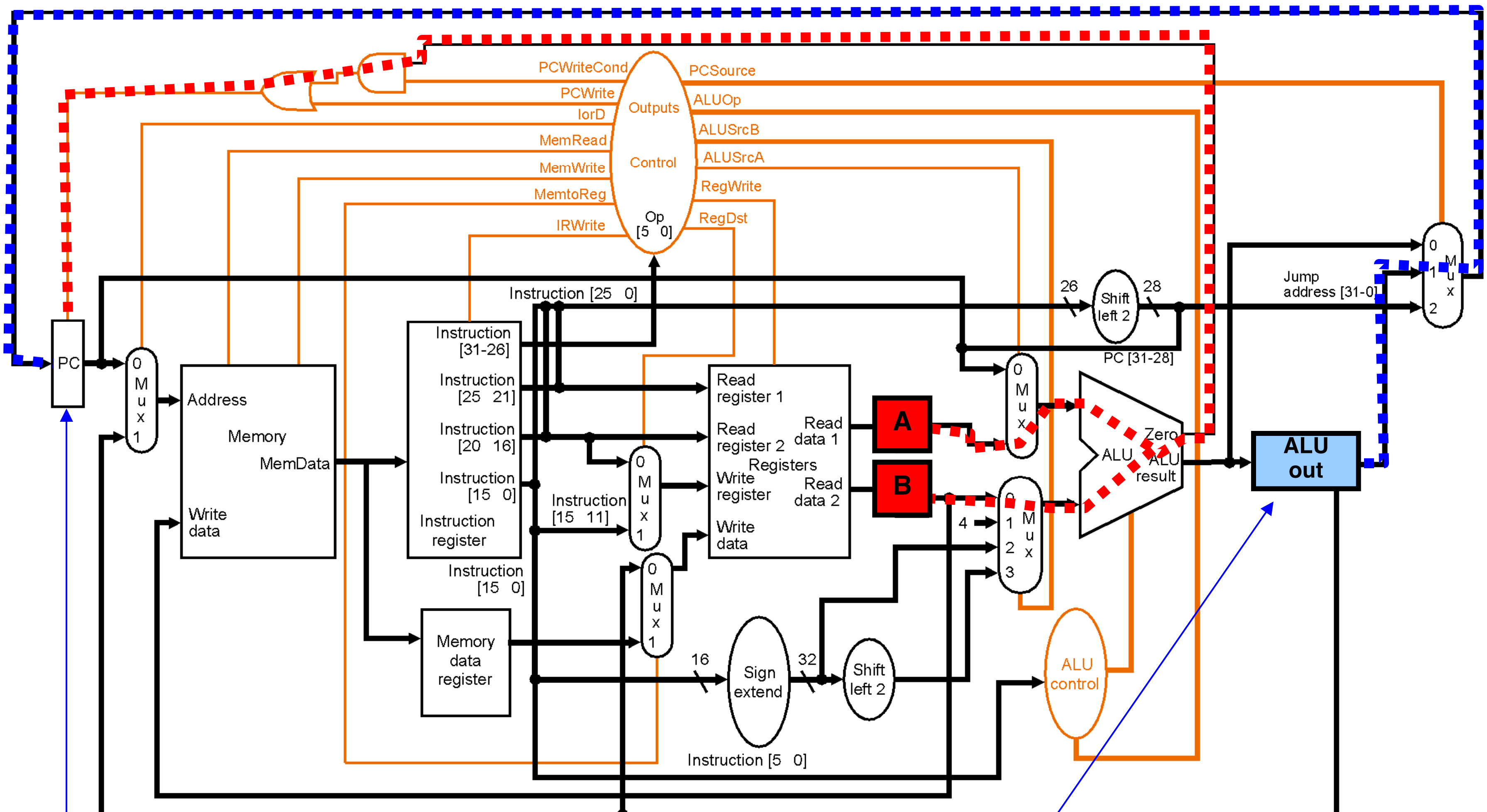


Control:

ALUSrcA=0, ALUSrcB=11, ALUOp=00

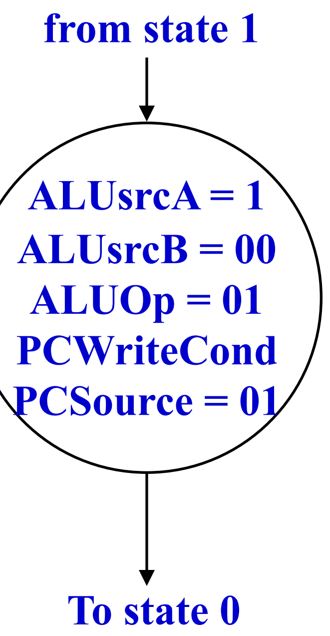
How about other signals? RegWrite, MemWrite, RegDest?

Cycle 3 for beq: Execute



- In cycle 1, PC was incremented by 4
- In cycle 2, ALUout was set to branch target
- This cycle, we conditionally update PC: if (A==B) PC=ALUout

FSM State for Cycle 3 of beq



R-type Instructions

- Cycle 3 (EXecute)

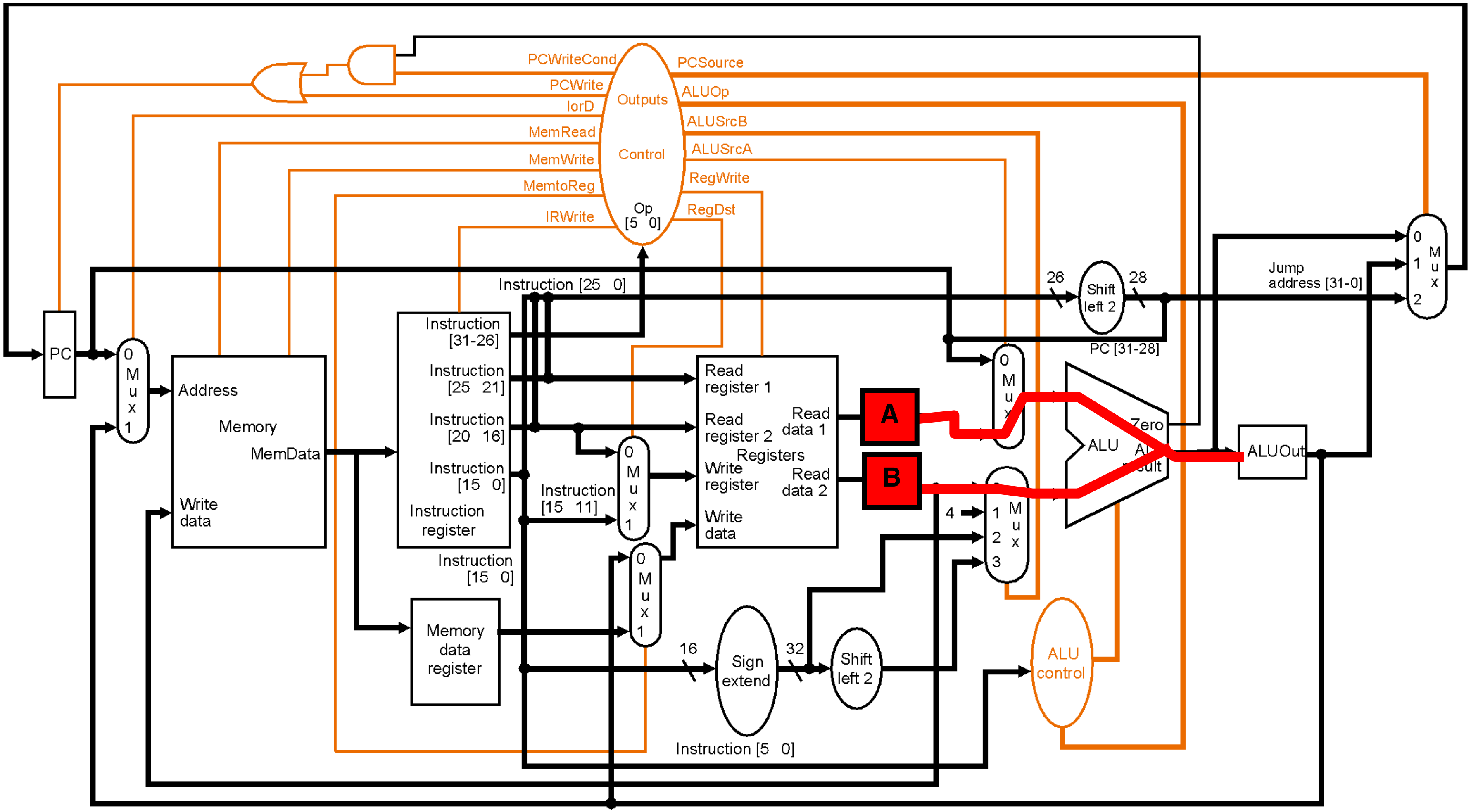
  `ALUout = A op B`

- Cycle 4 (WriteBack)

  `GPR[IR[15-11]] = ALUout`

R-type instruction is finished

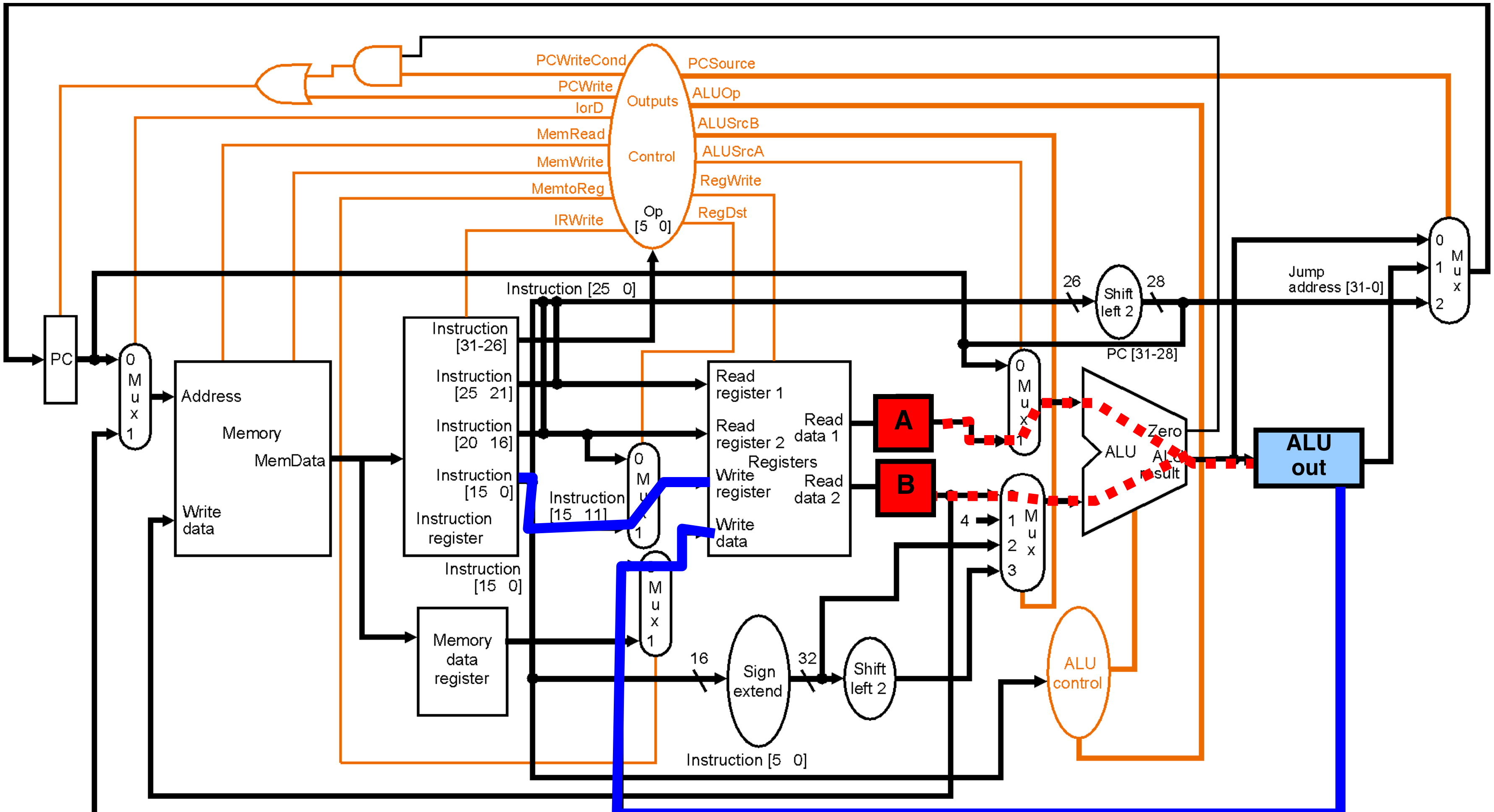
R-Type Execution



Cycle 3: **ALUout = A op B**

Cycle 4: GPR[IR[15-11]] = ALUout

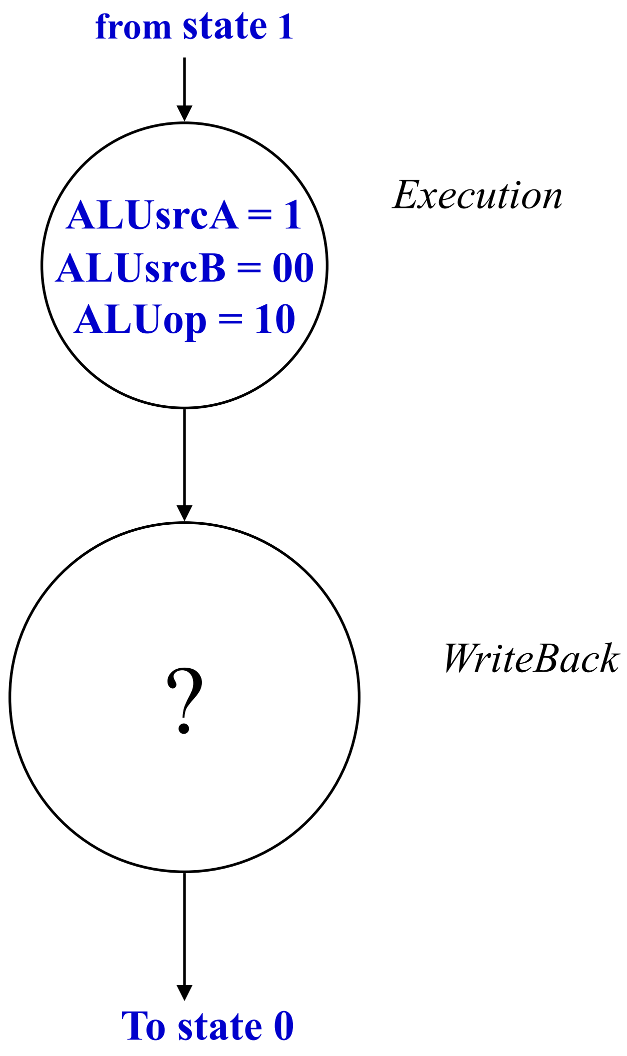
R-Type Execution & WB



Cycle 3: `ALUout = A op B`

Cycle 4: **`GPR[IR[15-11]] = ALUout`**

FSM States for R-type Instructions



Load and Store

- EXecute (cycle 3):
  - Compute memory address

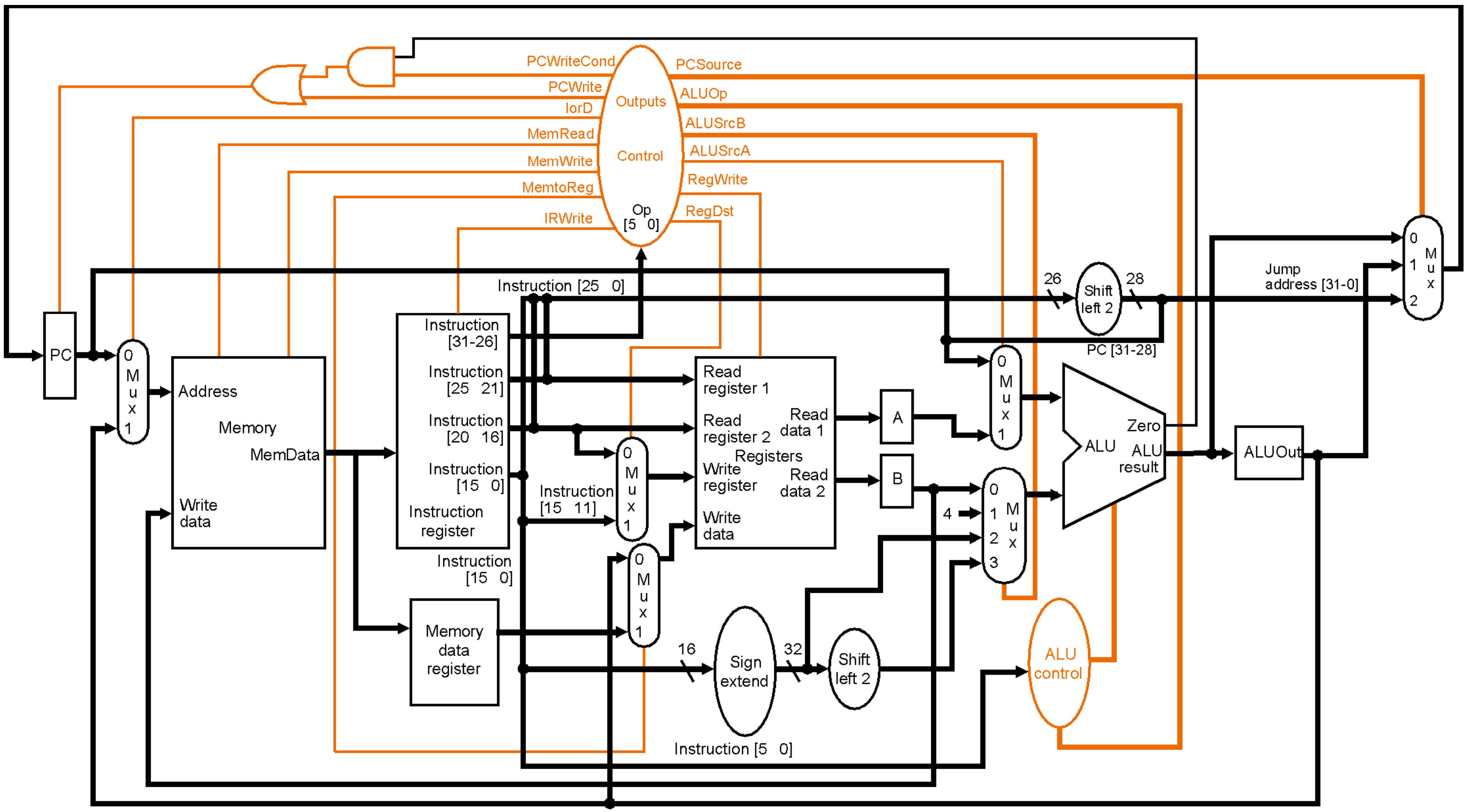
  `ALUout = A + sign-extend(IR[15-0])`
- Mem (cycle 4):
  - Access memory (read or write)

  Store: **`Mem[ALUout] = B`** (store finished)

  Load: **`MDR = Mem[ALUout]`**
- WB (cycle 5):
  - Write register (only for load))

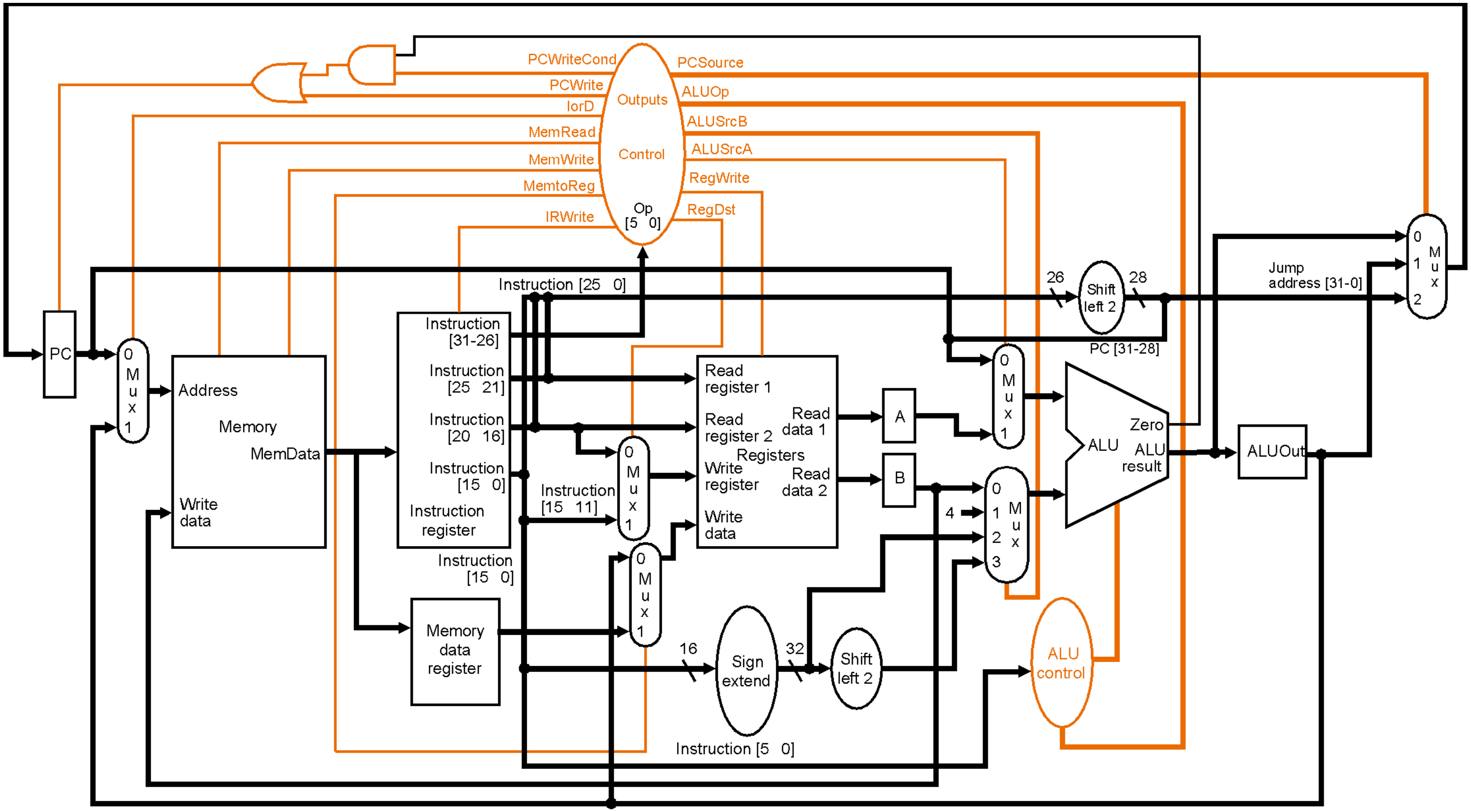
  `GPR[IR[20-16]] = MDR`

Cycle 3 for lw/sw: Address Computation



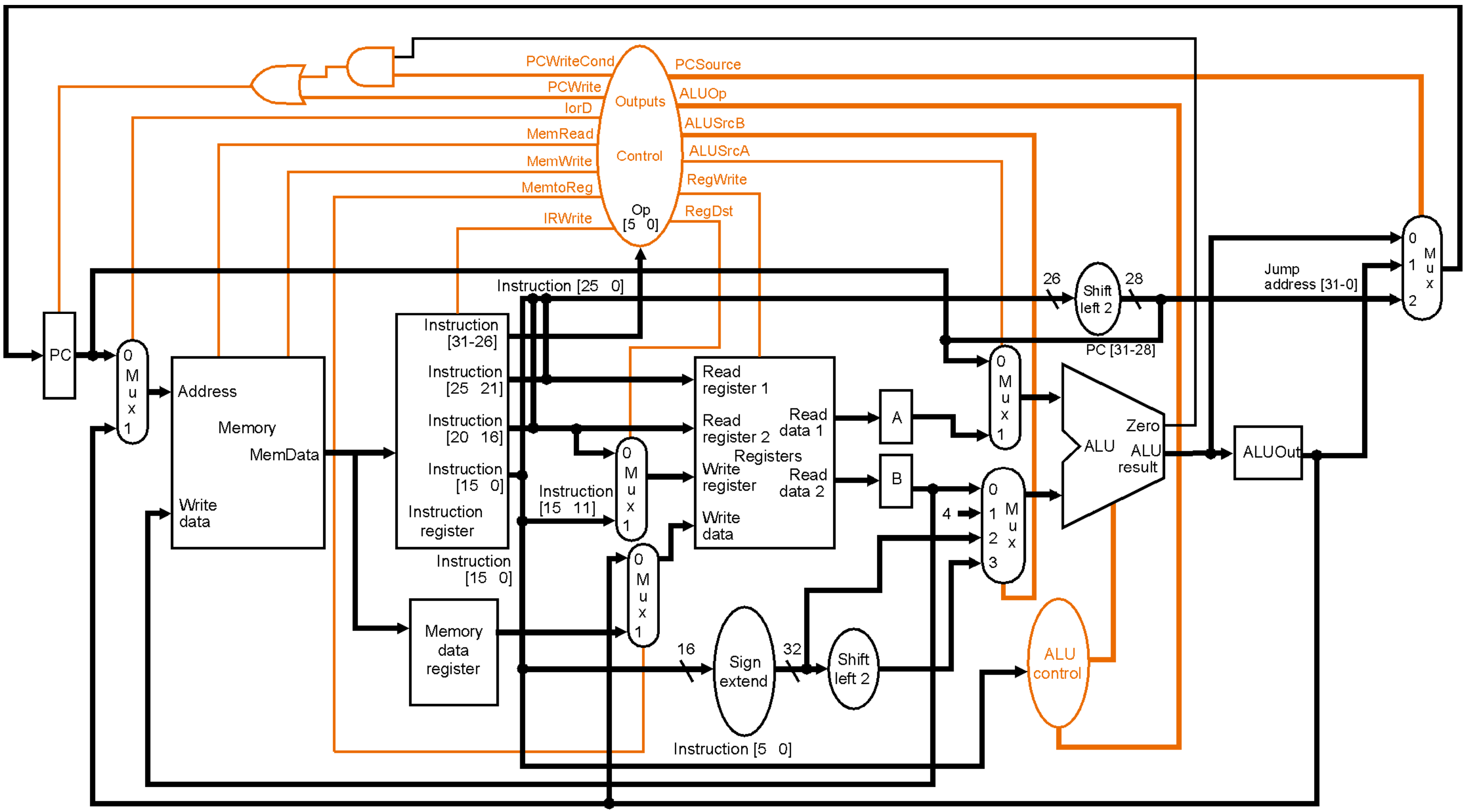
$$\texttt{ALUout = A + sign-extend(IR[15-0])}$$

Cycle 4 for Store: Memory Access



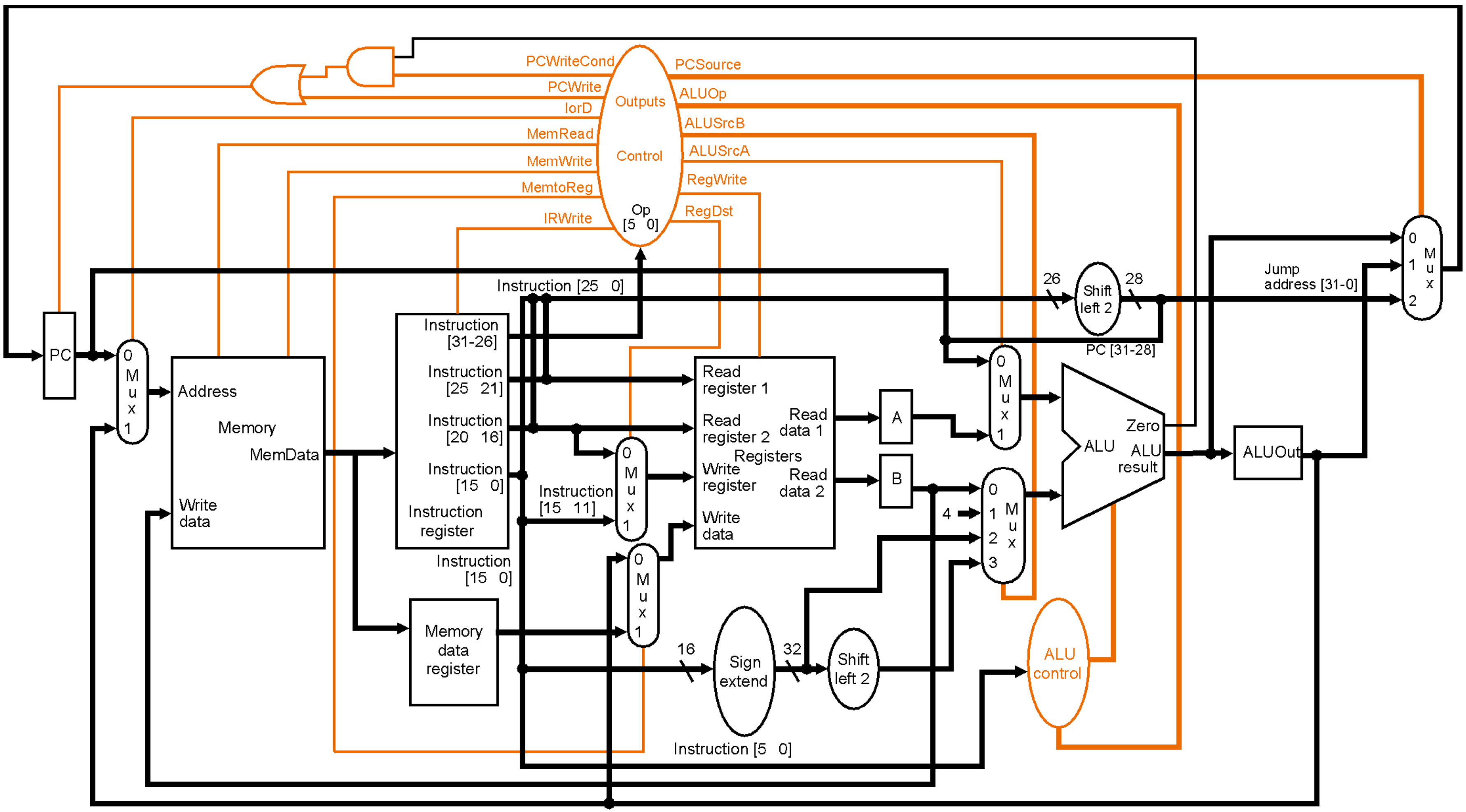
Memory[ALUout] = B

Cycle 4 for Load: Memory Access



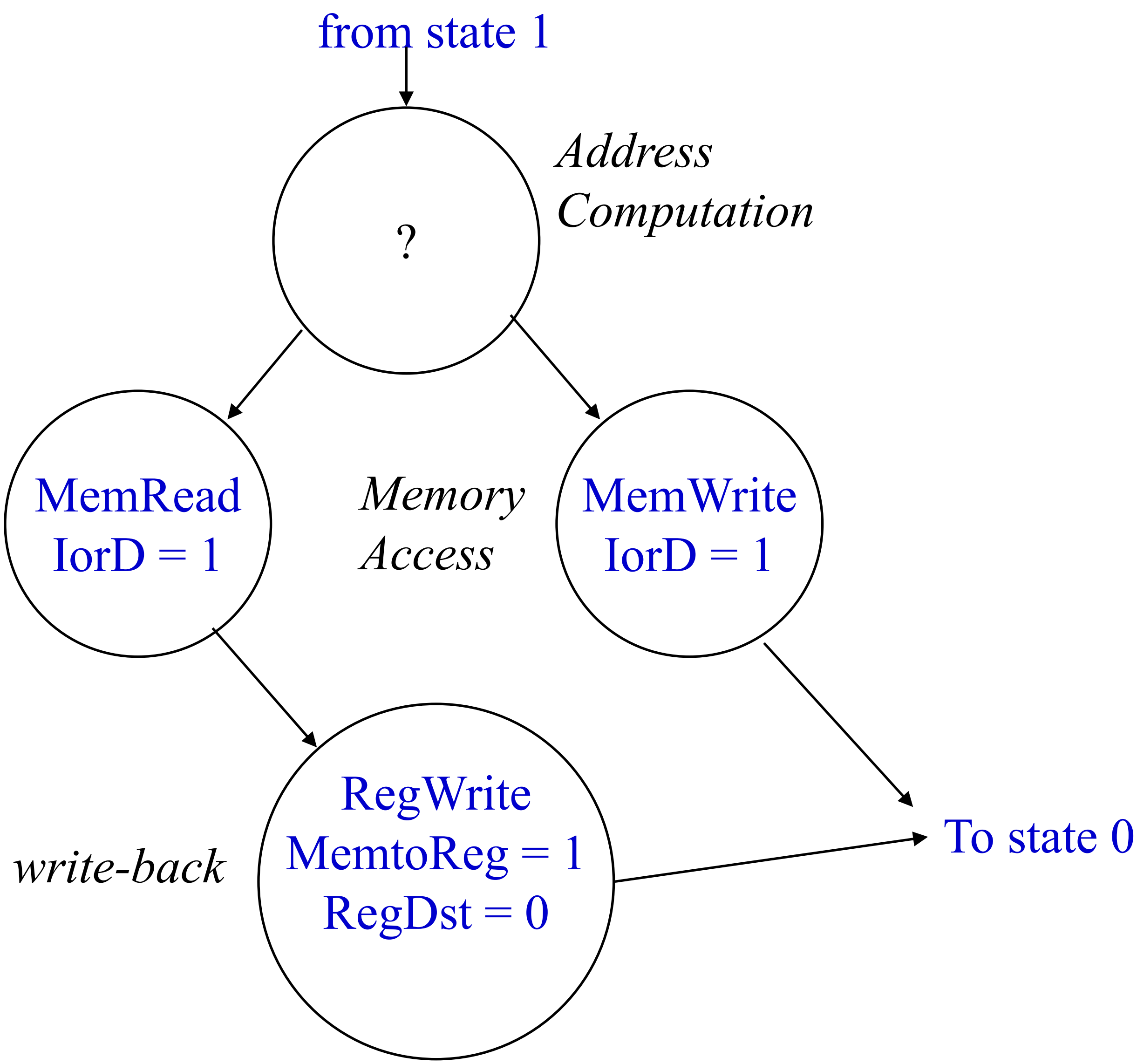
MDR = Memory[ALUout]

Cycle 5 for load: WriteBack

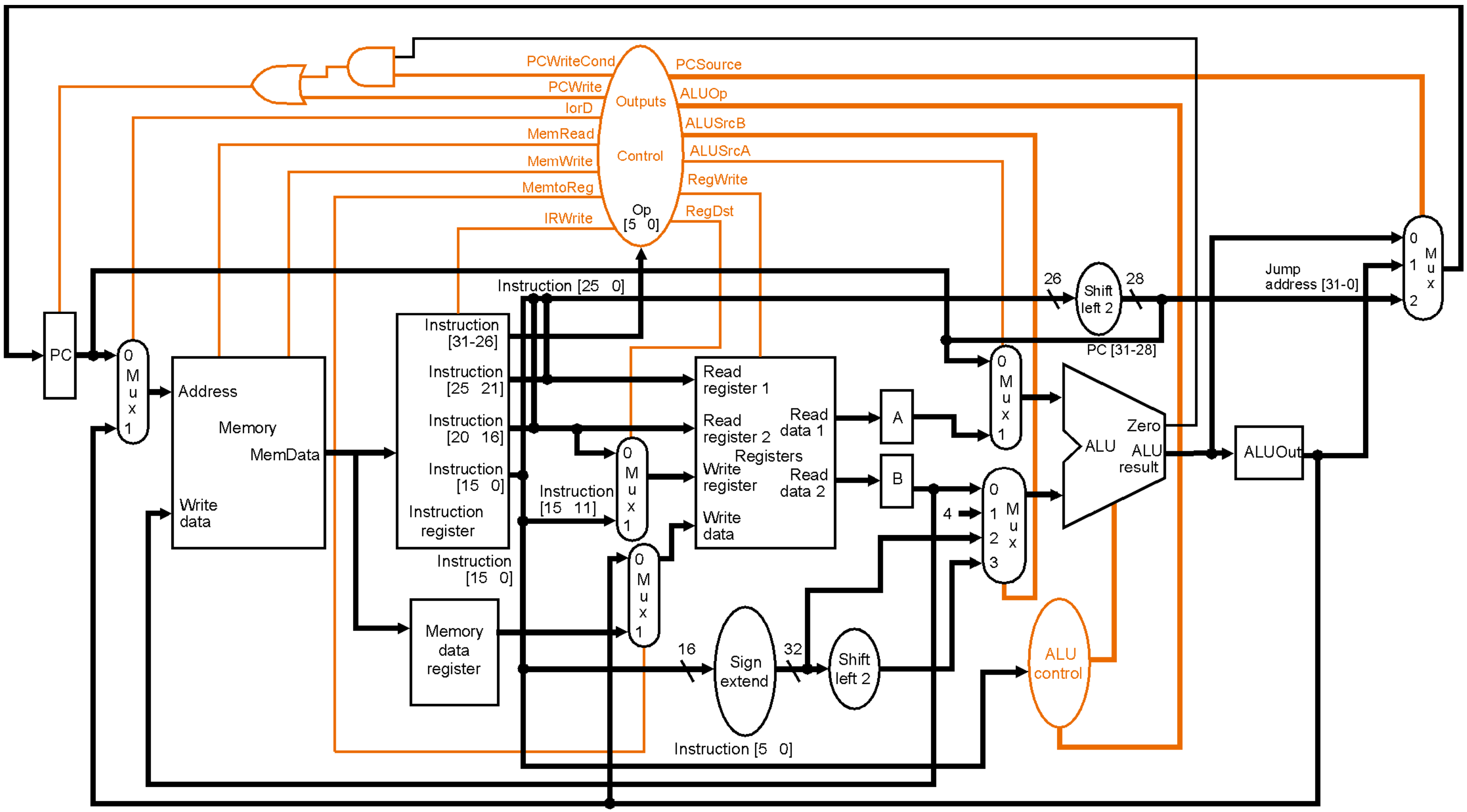


GPR[IR[20-16]] = MDR

Memory Instruction States

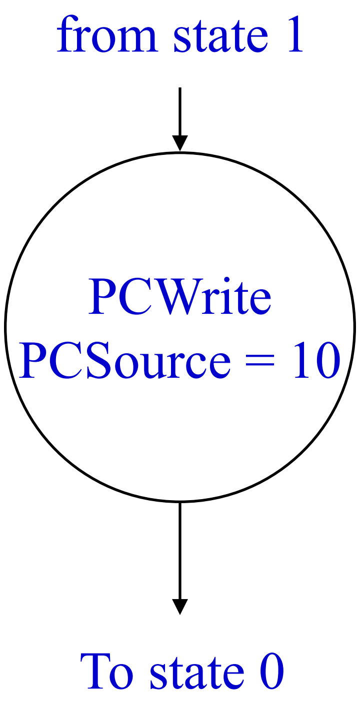


Cycle 3 for Jump

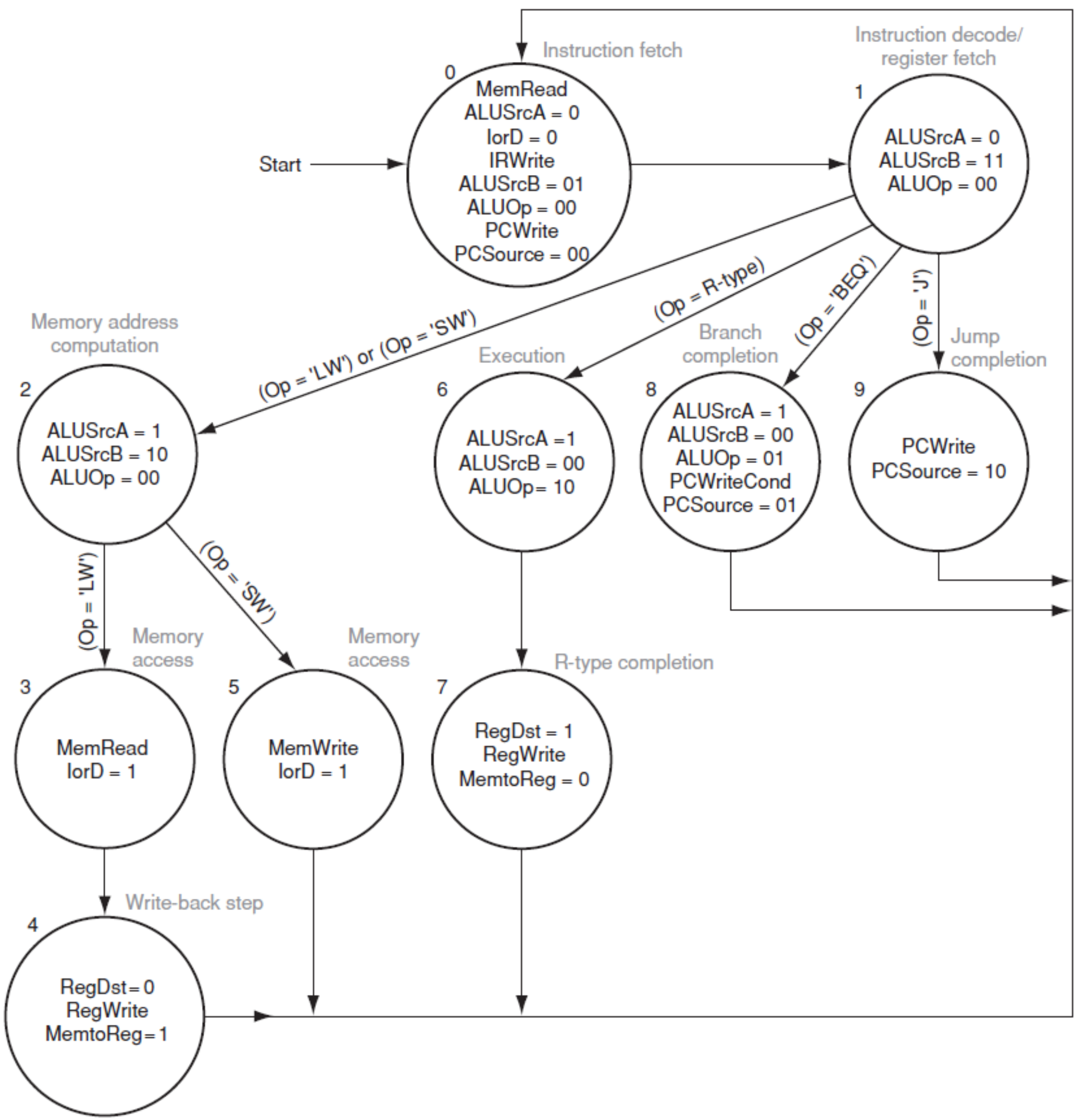


$$PC = PC[31-28] \mid (IR[25-0] << 2)$$

Cycle 3 Jump FSM state

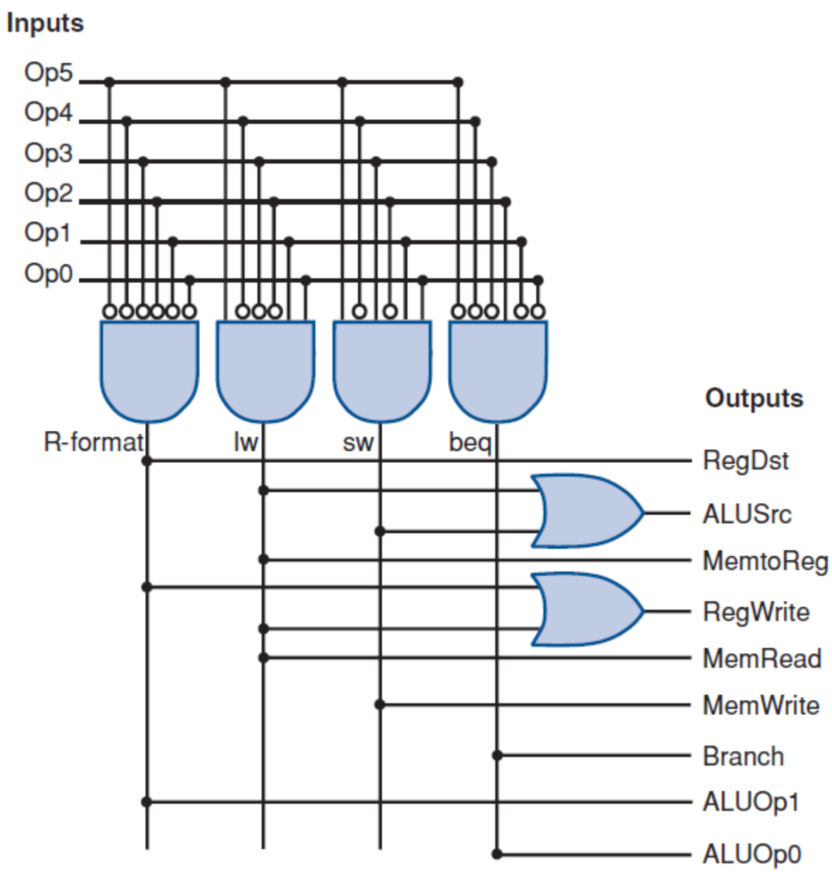


Complete FSM



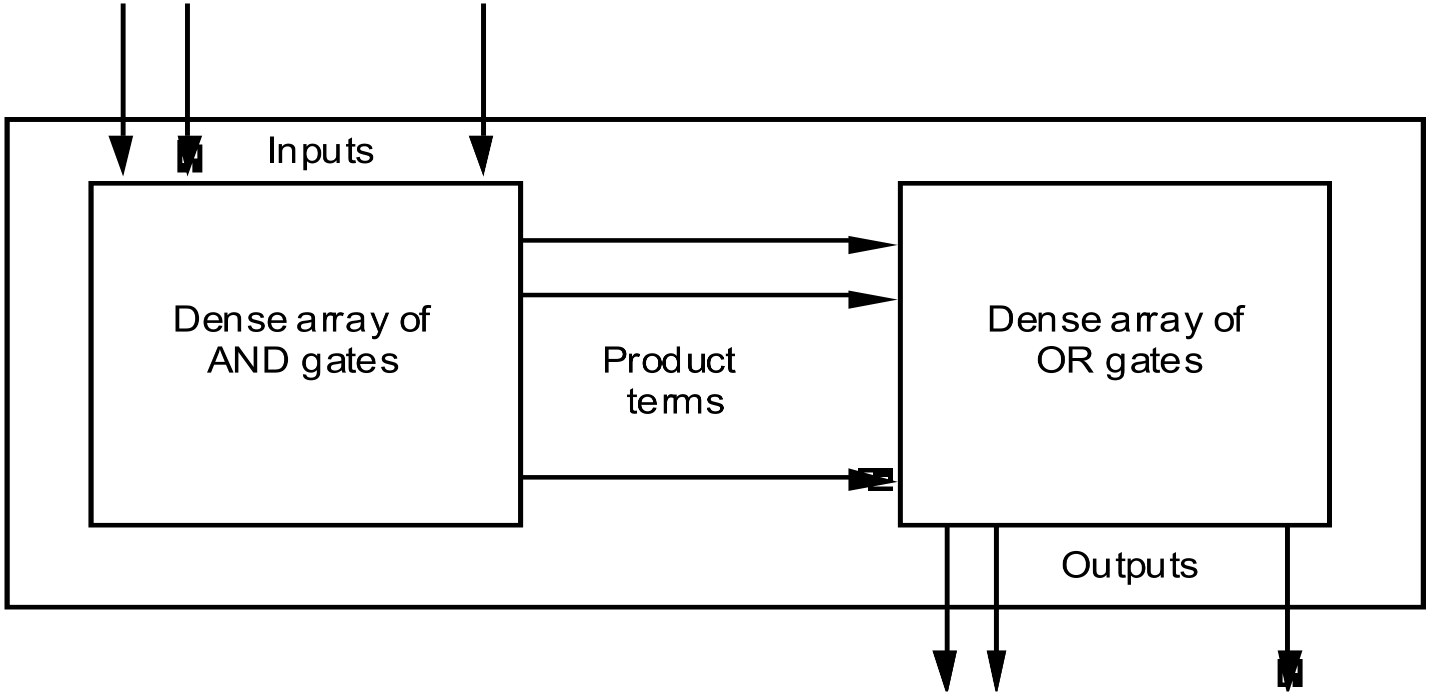
Single-Cycle Control Unit Implementation

- Unstructured LogicDesign
  - By Karnaugh Map
- PLA/PAL

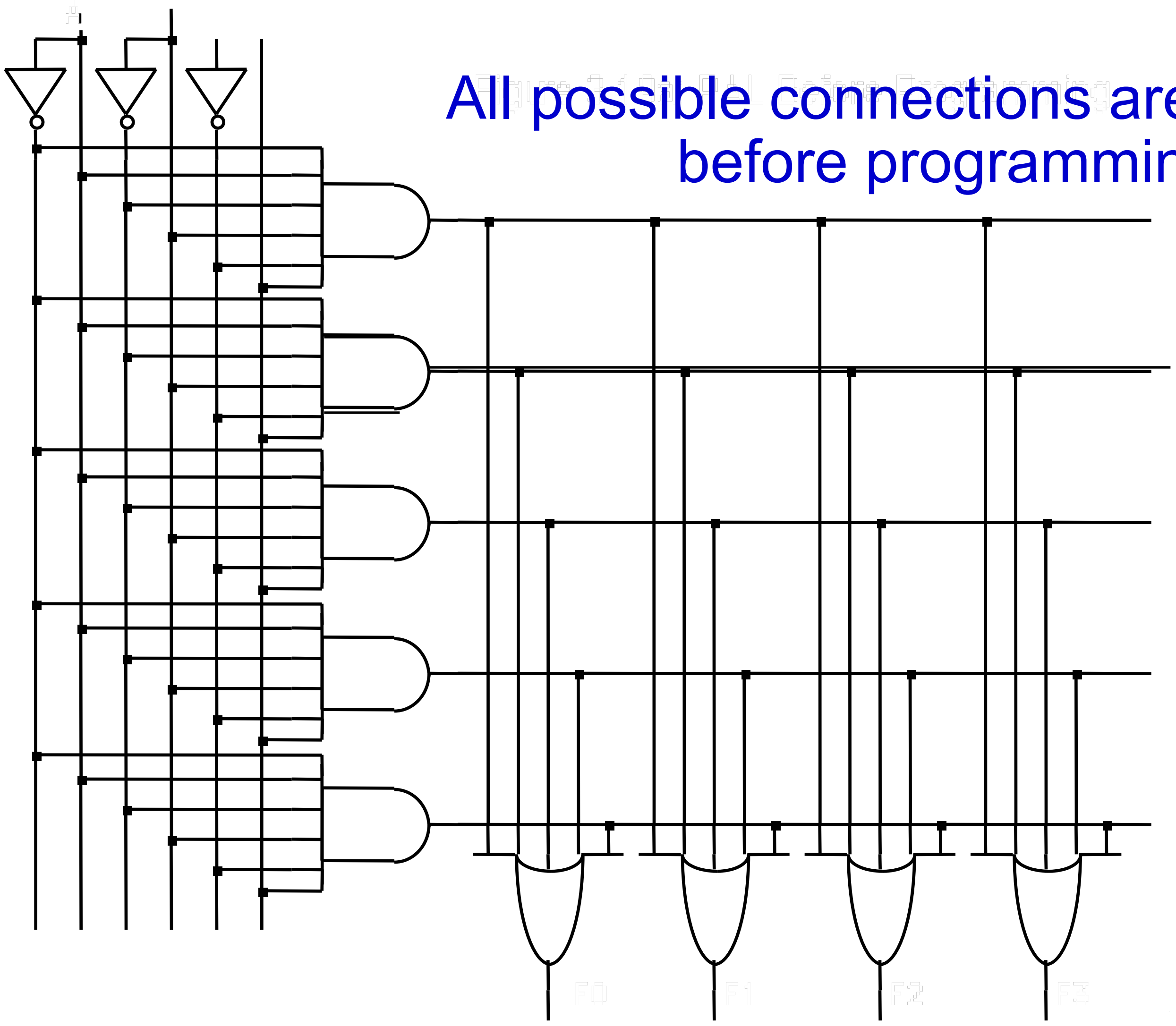


PAL/PLA

- What is PAL/PLA?
  - Pre-fabricated building block of many AND/OR gates (or NOR/NAND)
  - Personalized by making or breaking connections among gates



PLA



All possible connections are available before programming

PLA Example

$F1 = A\,B\,C$

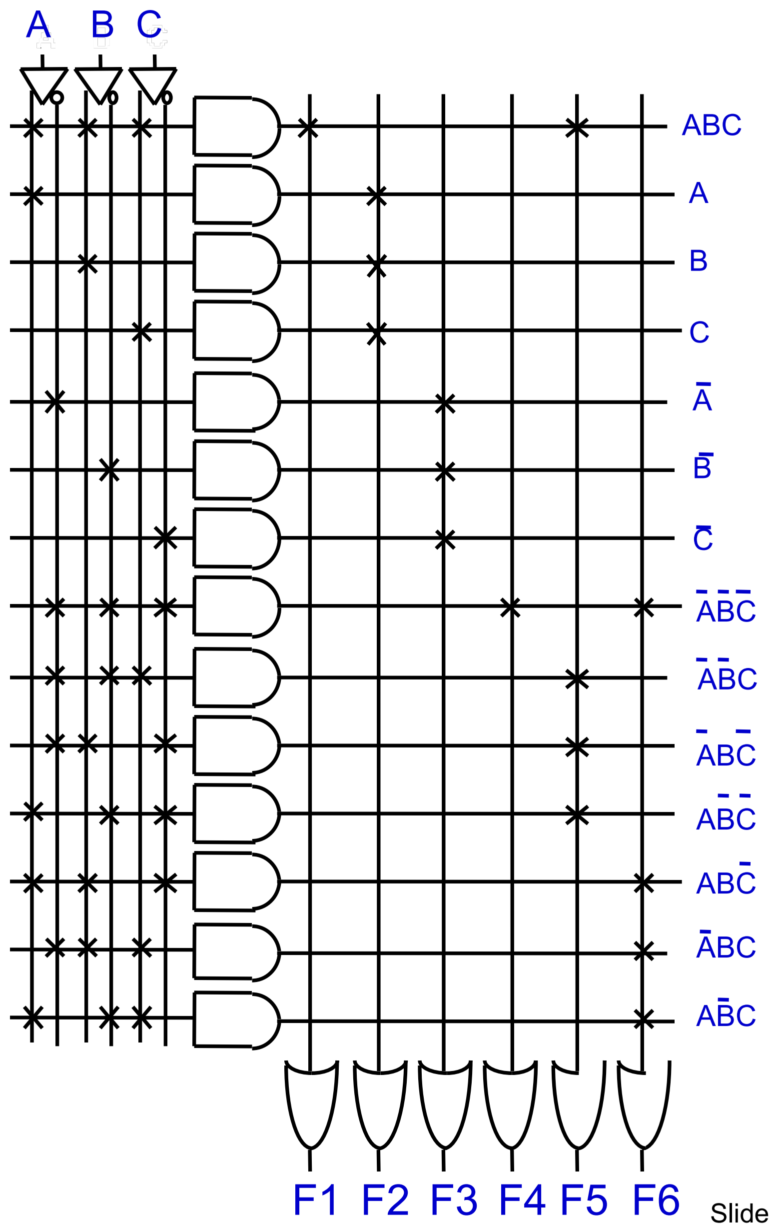
$F2 = A + B + C$

$F3 = \overline{A\,B\,C}$

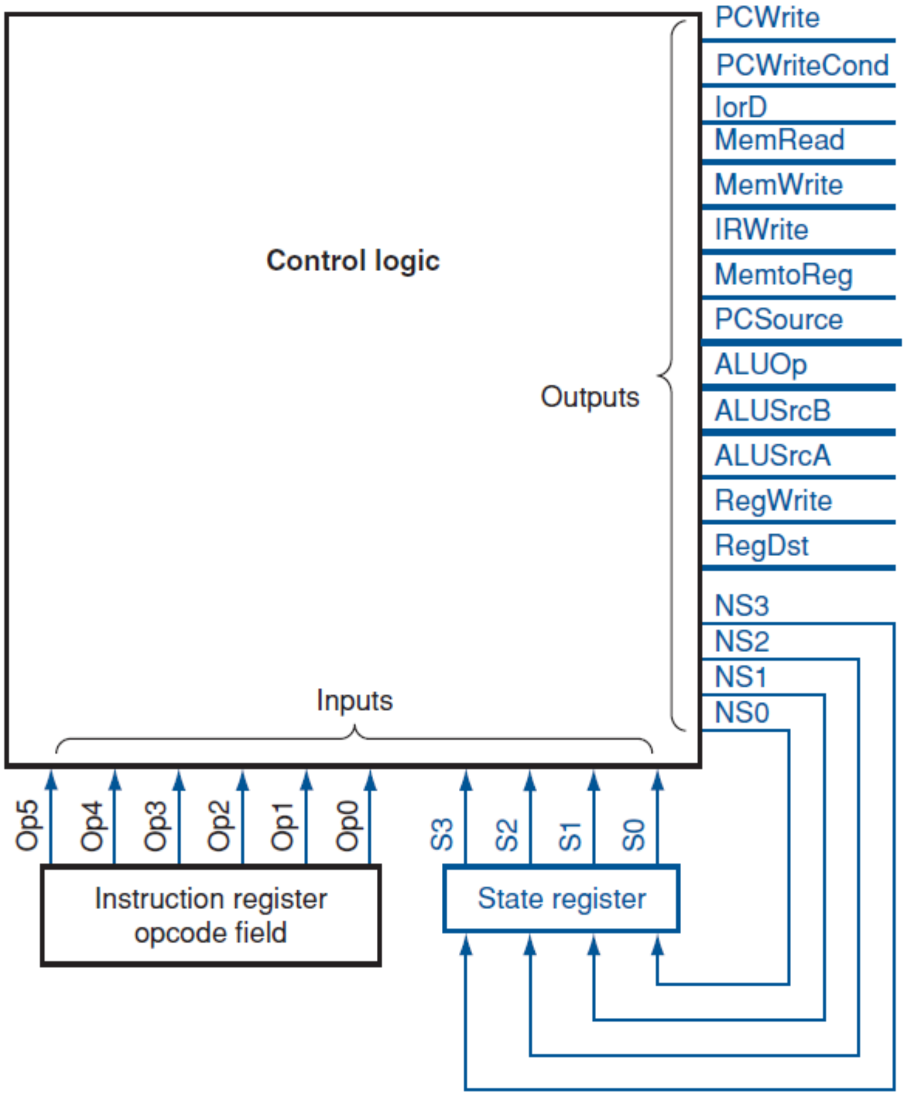
$F4 = \overline{A + B + C}$

$F5 = A \text{ xor } B \text{ xor } C$

$F6 = A \text{ xnor } B \text{ xnor } C$



Multi-Cycle Control Unit Implementation

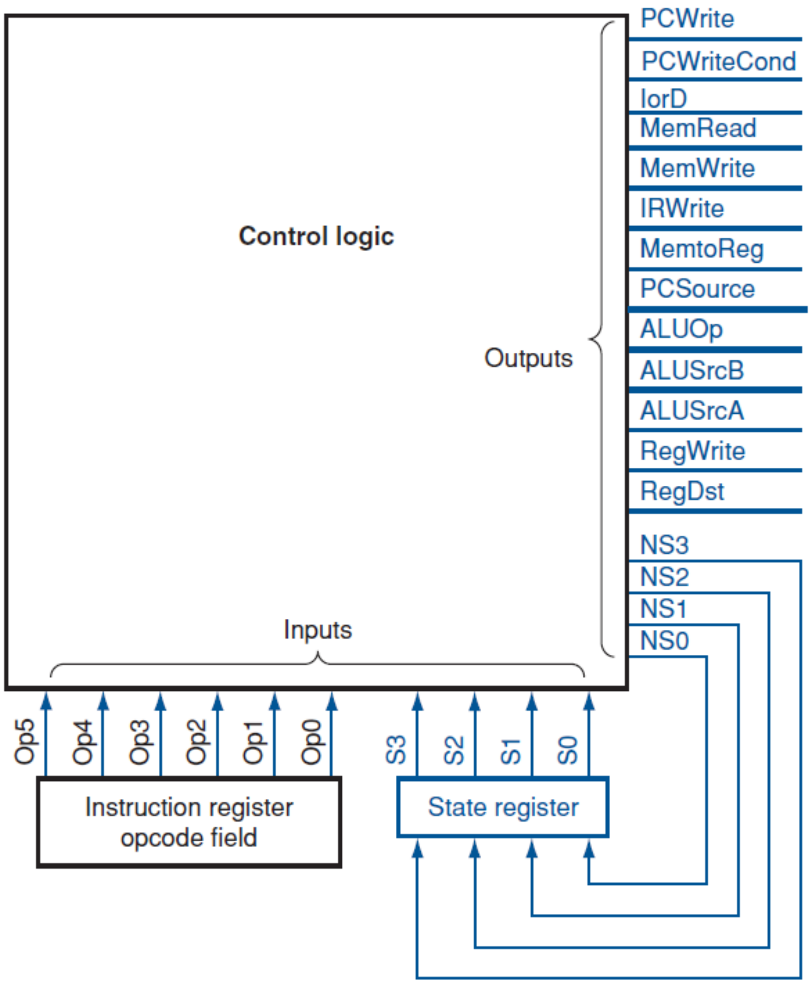


Multi-Cycle Control Unit Implementation (cont.)

- State Register (S3~S0)
- Control Logic
  - Combinational logic
  - Inputs ?
  - Outputs ?

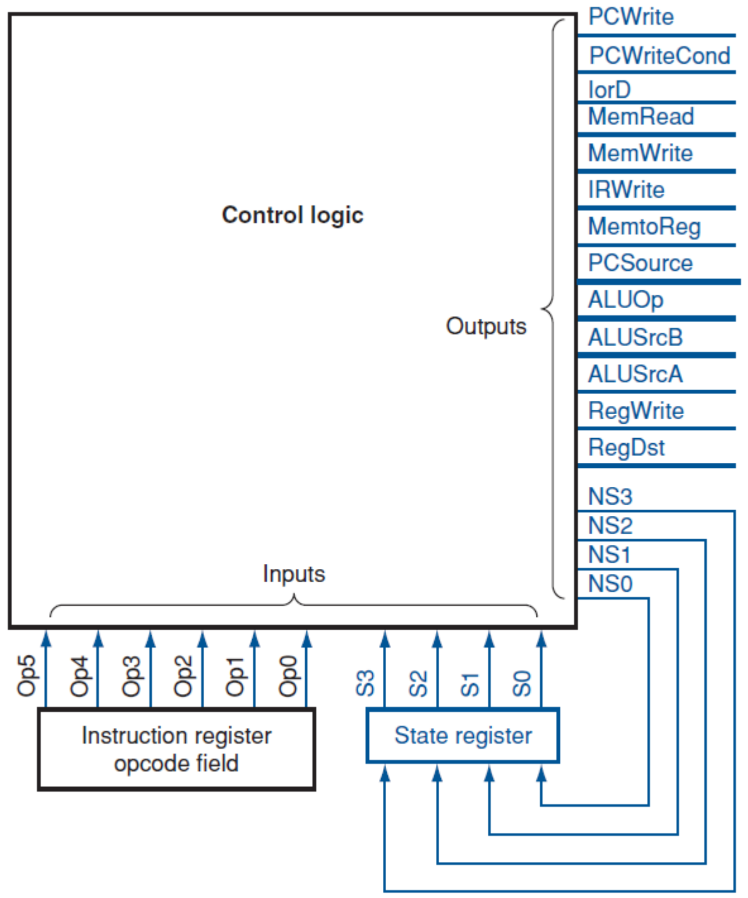
Multi-Cycle Control Unit Implementation (cont.)

- Control Logic Inputs
  - Opcode bits: Op5~Op0
  - S3~S0

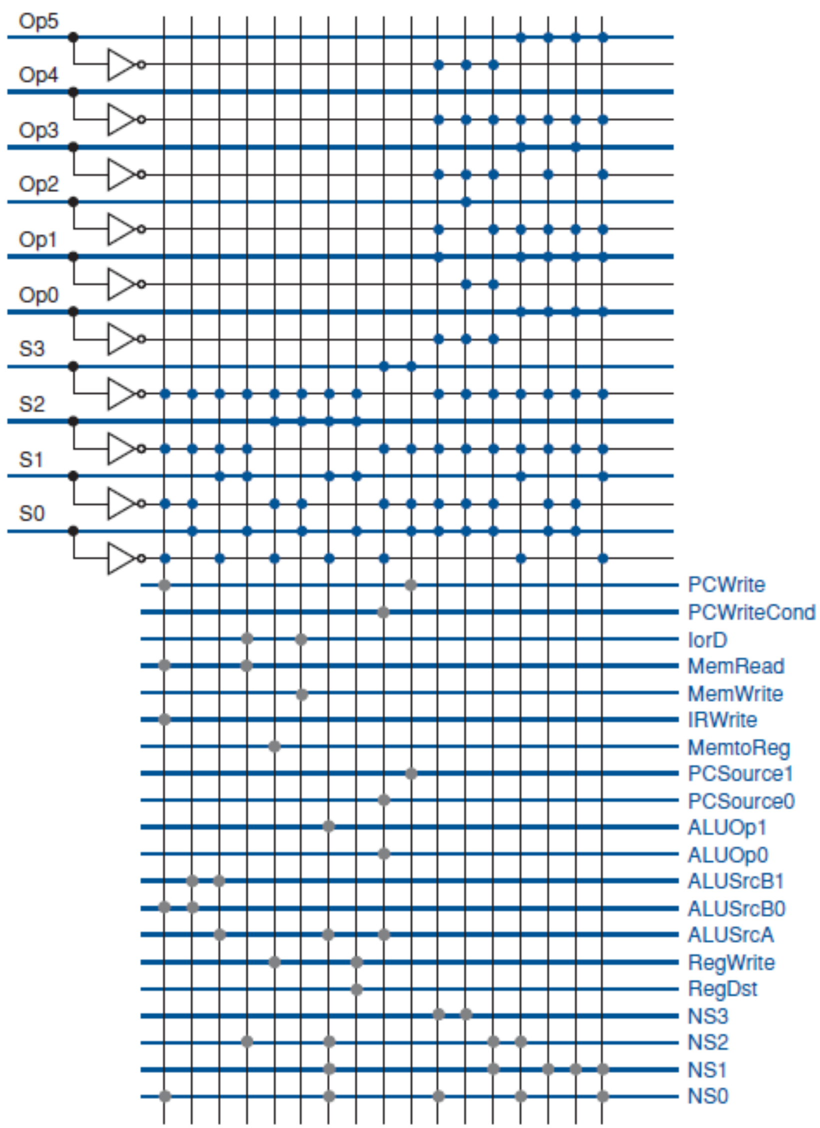


Multi-Cycle Control Unit Implementation (cont.)

- Control Logic Outputs
  - Control signals: PCWrite, IorD, …
    - Depends only on current state
  - NS3~NS0
    - Depends on both current state & opcode bits

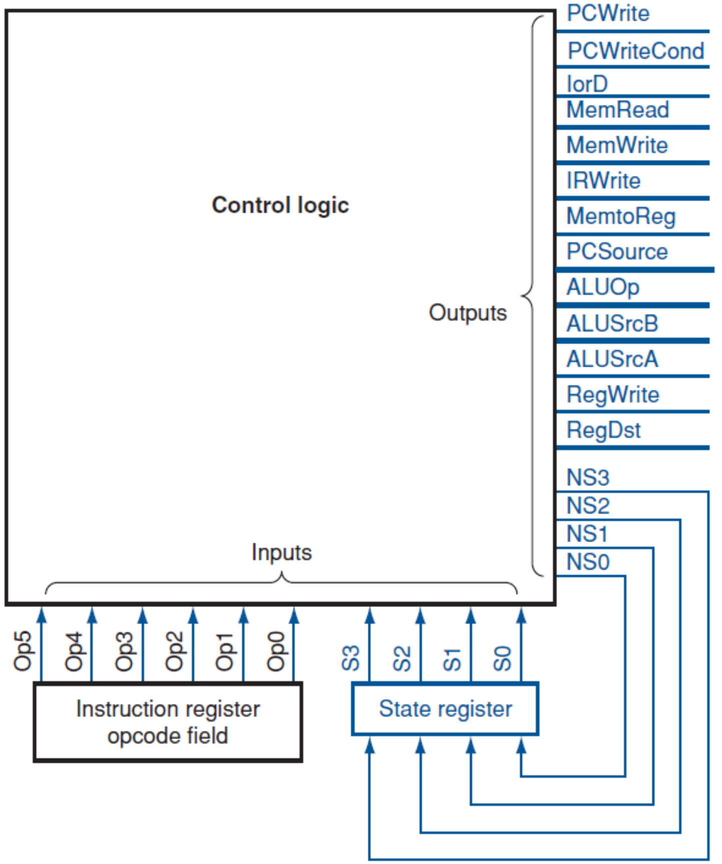


Multi-Cycle Control Unit Implementation in PLA



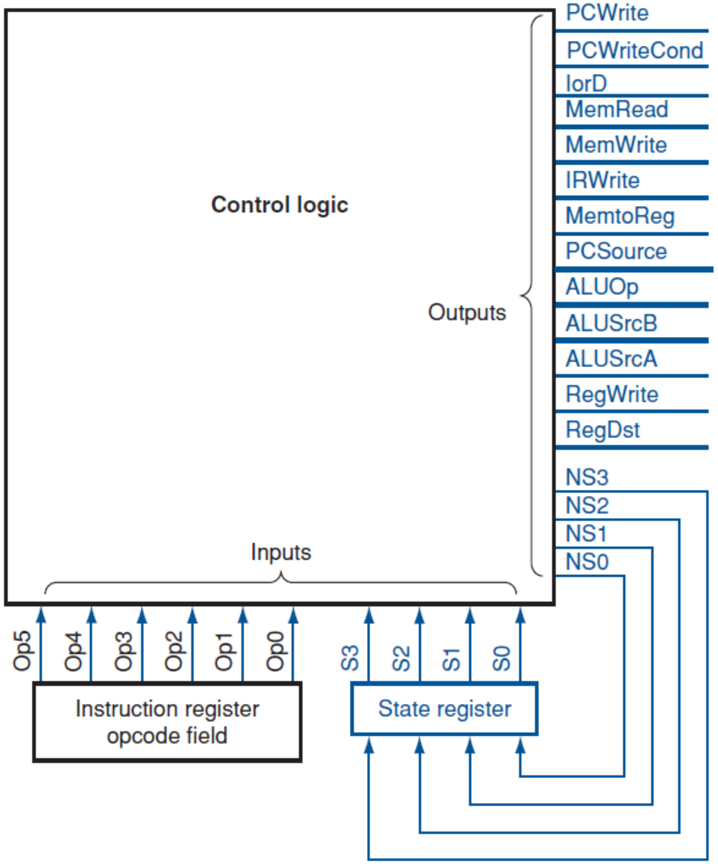
Multi-Cycle Control Unit Implementation in ROM

- ROM
  - Can be used to implement control unit
  - # of inputs: 10
  - # of outputs: 20
  - Use a ROM with:
    - Address width: 10
    - Data width: 20
    - ROM size: $20*2^{10}$ = 20Kb
    - 1024 entries



Multi-Cycle Control Unit Implementation in ROM (cont.)

- Question:
  - Can we use smaller ROM(s) to implement control unit?
- Answer: 2 Separate ROMs
  - First ROM: $16*2^4 = 256b$
    - # of inputs: 4
    - # of outputs: 16
  - Second ROM: $4*2^{10} = 4Kb$
    - # of inputs: 10
    - # of outputs: 4
  - Total ROM size: 4.3Kb

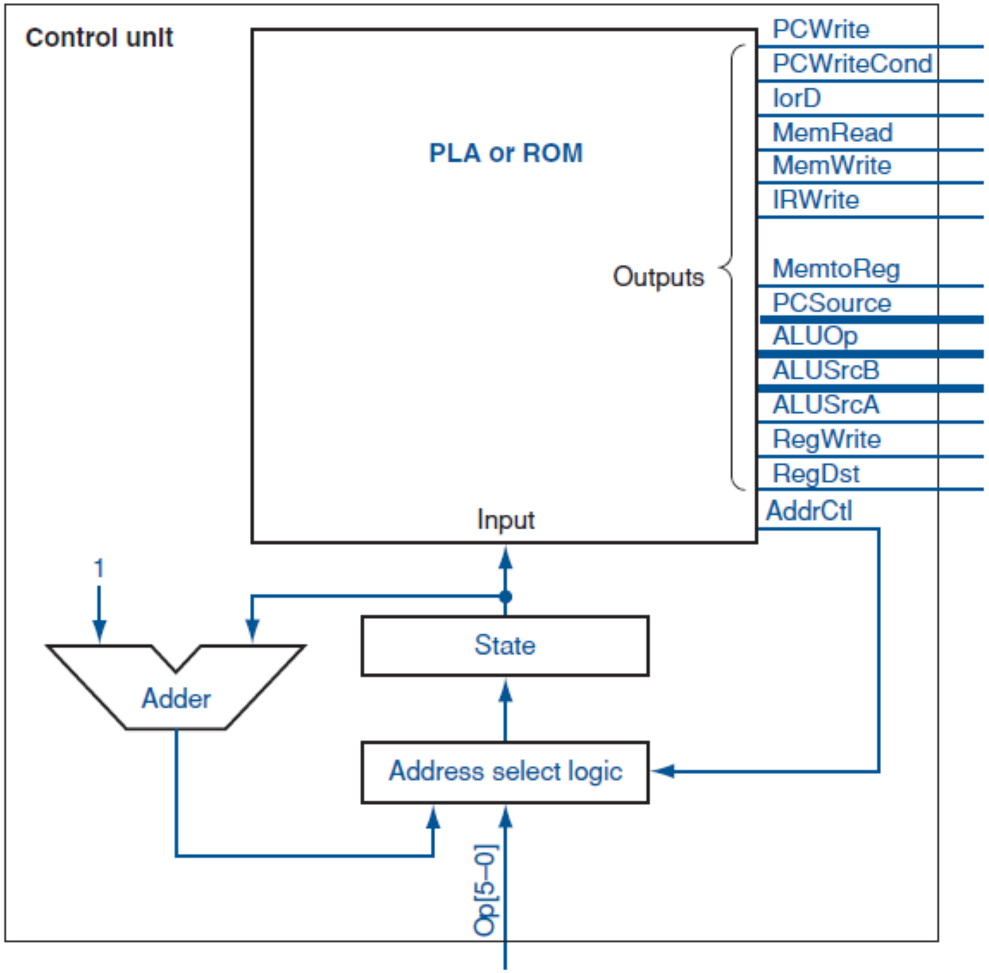


Implementing Multi-Cycle Control Using Micro-Program

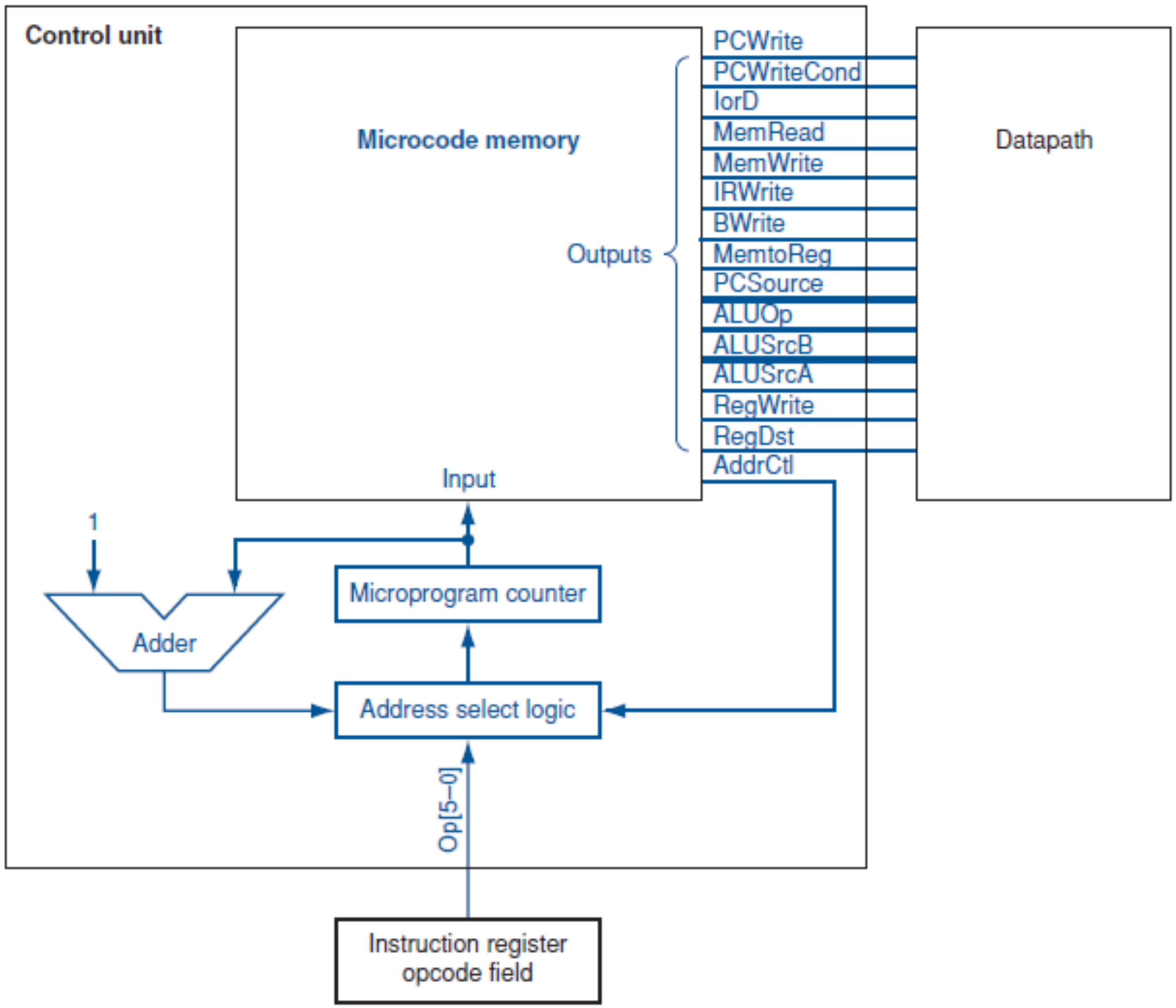
- Cons of ROM Implementation
  - 95% of ROM used to indicate next state
    - 4Kbits
  - What if we have more complex ISA?
    - FP instructions which may take several cycles
- Example:
  - Consider an FSM which requires 10 FFs
    - What would be size of ROM?
- What’s Solution?

Implementing Multi-Cycle Control Using Micro-Program (cont.)

- ROM Control Words
  - Micro-instructions
- State Register
  - Micro-program counter
  - Also called:
    - Microcode sequencer



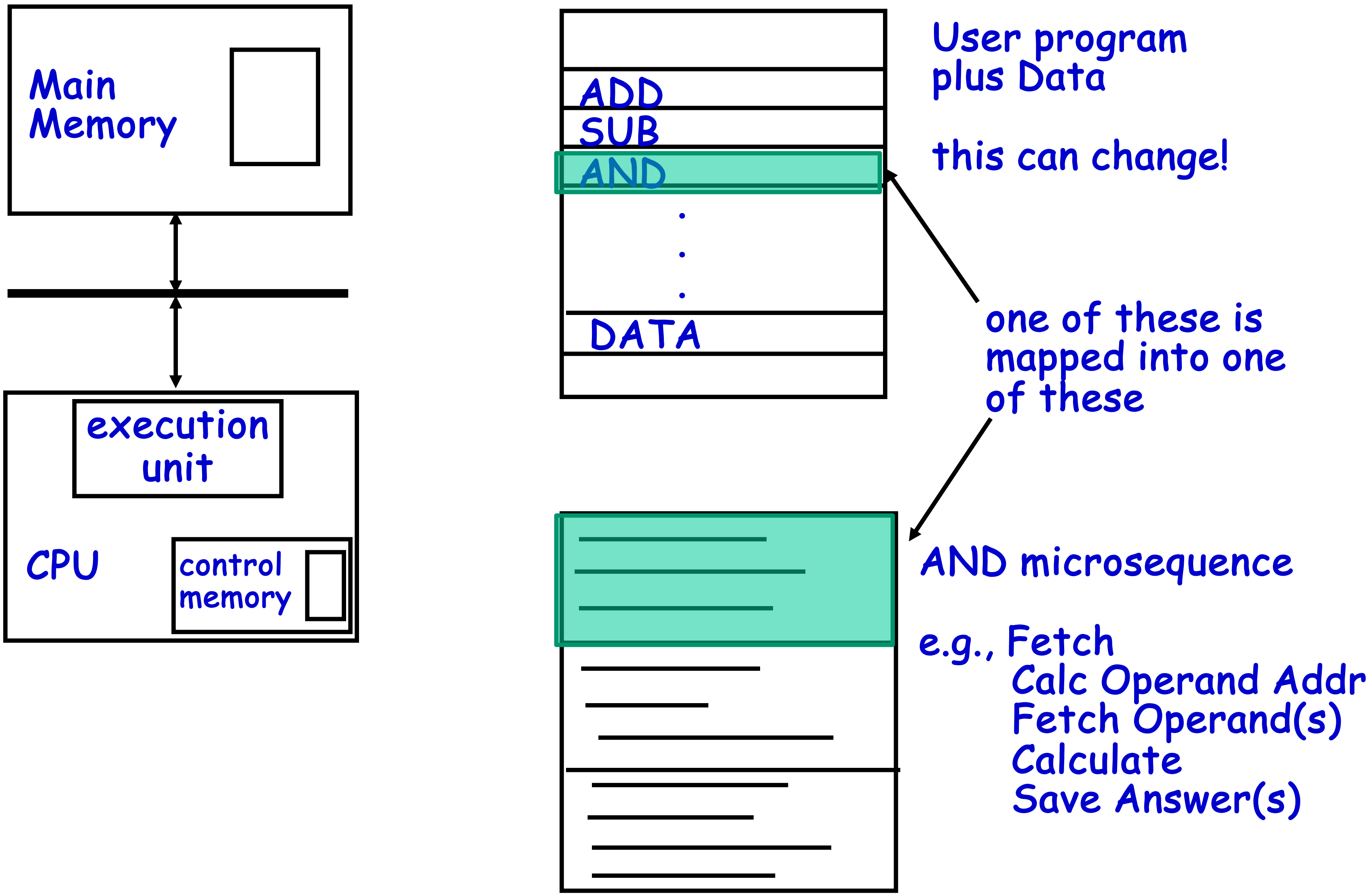
Implementing Multi-Cycle Control Using Micro-Program (cont.)



Microprogramming (cont.)

- A Convenient Method to Implement *structured* control state diagrams
  - Random logic replaced by μ-PC sequencer and ROM
  - Each line of ROM called a μ-instruction
  - Limited state transitions:
    - Branch to zero, next sequential, branch to μinstruction address from displatch ROM

Macro-Instruction Interpretation



Microprogramming (cont.)

- 80x86 Instructions
  - –Instructions translate to 1 to 4 micro-operations
- Complex 80x86 Instructions
  - –Executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations

Hardwired vs. Micro-Programmed

- Micro-Programmed
  - Can change micro-operations without changing circuit (just by reprogramming ROM)
  - Easier design approach
  - More disciplined control logic
    - Easier to debug
  - Enables more complex ISA
  - Enables family of machines with same ISA
- Hard-Wired
  - Area efficient
  - Probably less delay


Thanks for Your Attention!