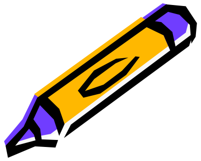
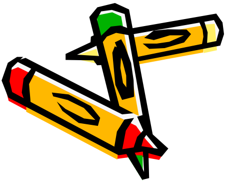
Computer Architecture

Hossein Asadi
Department of Computer Engineering
Sharif University of Technology
asadi@sharif.edu



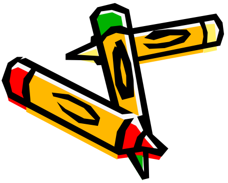
What Learned So Far?

- Lecture 1
  - Review
    - ISA, computer organization, & addressing modes
  - uArchictecure
  - uArch basic blocks



What Learned So Far? (cont.)

- Lecture 2
  - Performance metrics
    - Latency, throughput, MIPS, & MFLOPS
  - CPU time
    - CPI, IC, & clock cycle time
  - Performance evaluation
  - Standard benchmarks
  - Speedup
  - Amdahl law



What Learned So Far? (cont.)

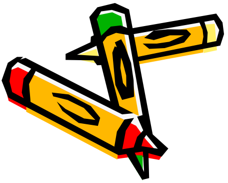
- Lecture 3
  - Register level transfer language (RTL)
  - Micro-operations
  - Bus transfer & bus implementation

What Learned So Far? (cont.)

- Lecture 4
  - Arithmetic Implementations
  - Ripple Carry (RC) Adder
  - Carry Select Adder (CSA)
  - Carry Look-Ahead (CLA) Adder
  - Combinational Multiplier
  - Sequential Multiplier
  - Booth Multiplier

What Learned So Far? (cont.)

- Lecture 5
  - Single-cycle datapath design
  - Datapath elements
    - PC, I-Mem, D-Mem, ALU, GPR, address bus, & data bus
  - Combining datapaths
  - Datapath for reg-reg operations
  - Datapath for load/store operations
  - Datapath for branch operations



What Learned So Far? (cont.)

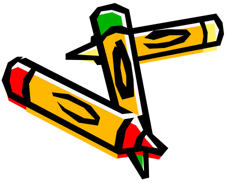
- Lecture 6
  - Control logic design for single-cycle CPU
  - ALU control design
  - Processor main control unit design
  - Implementing unconditional branch

Today's Topics

- Efficiency of Single-Cycle Datapath
- Multi-Cycle Datapath Design

Copyright Notice

- Parts (text & figures) of this lecture adopted from:
  - Computer Organization & Design, The Hardware/Software Interface, 3rd Edition, by D. Patterson and J. Hennessey, Morgan Kaufmann publishing, 2005.
  - "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, Fall 2009.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC, Spring 2009.



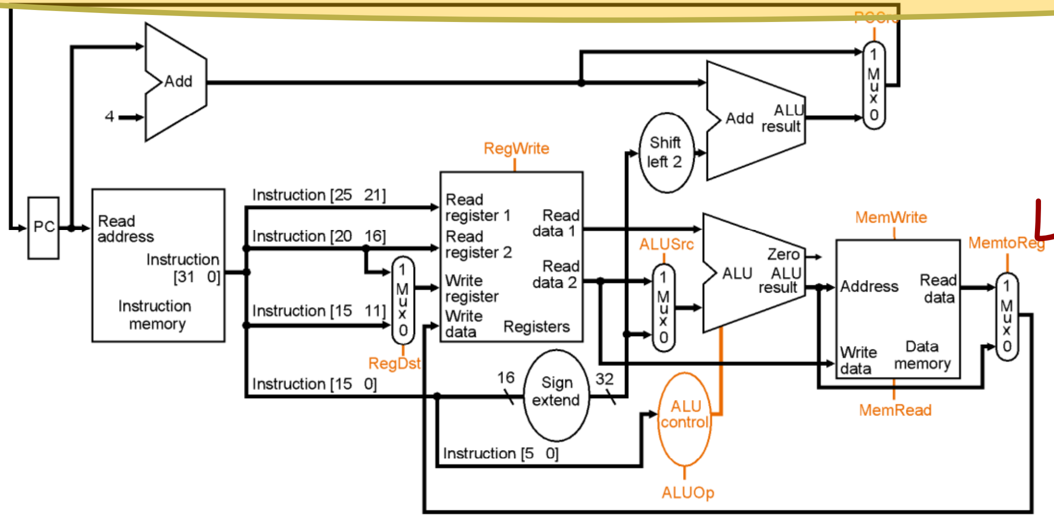
Performance of Single-Cycle uArch

- 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

با توجه به اسلاید قبل الان طول هر clock cycle برابر ۴.۷ می شود

	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

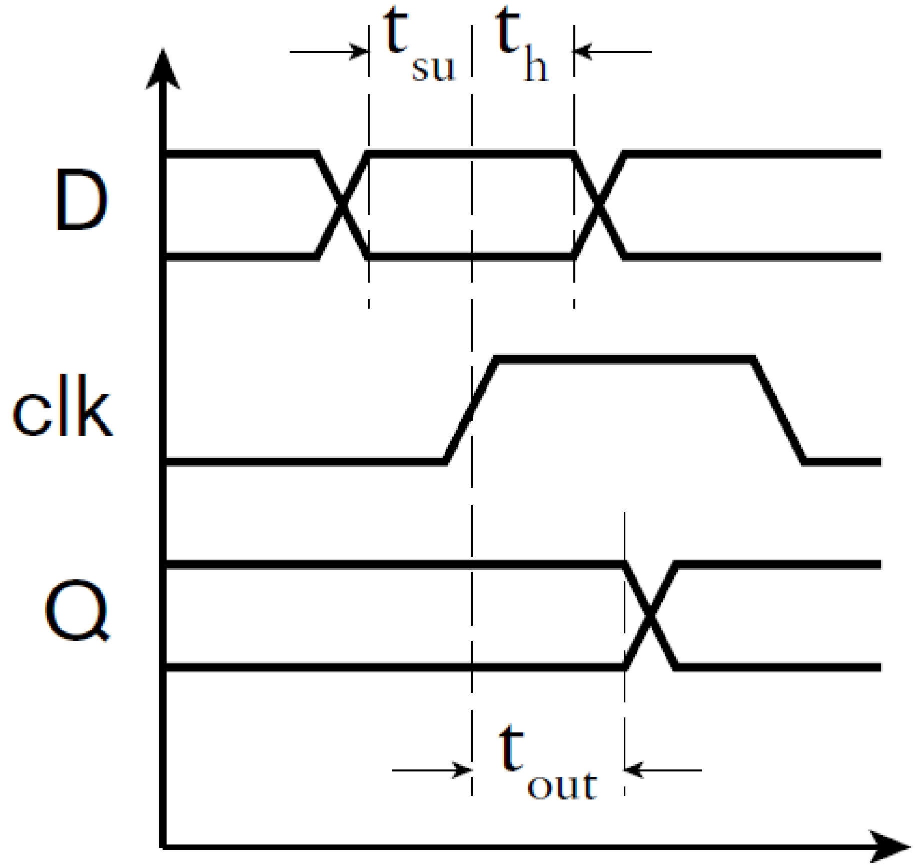
Setup time?

Hold time?

Critical path?

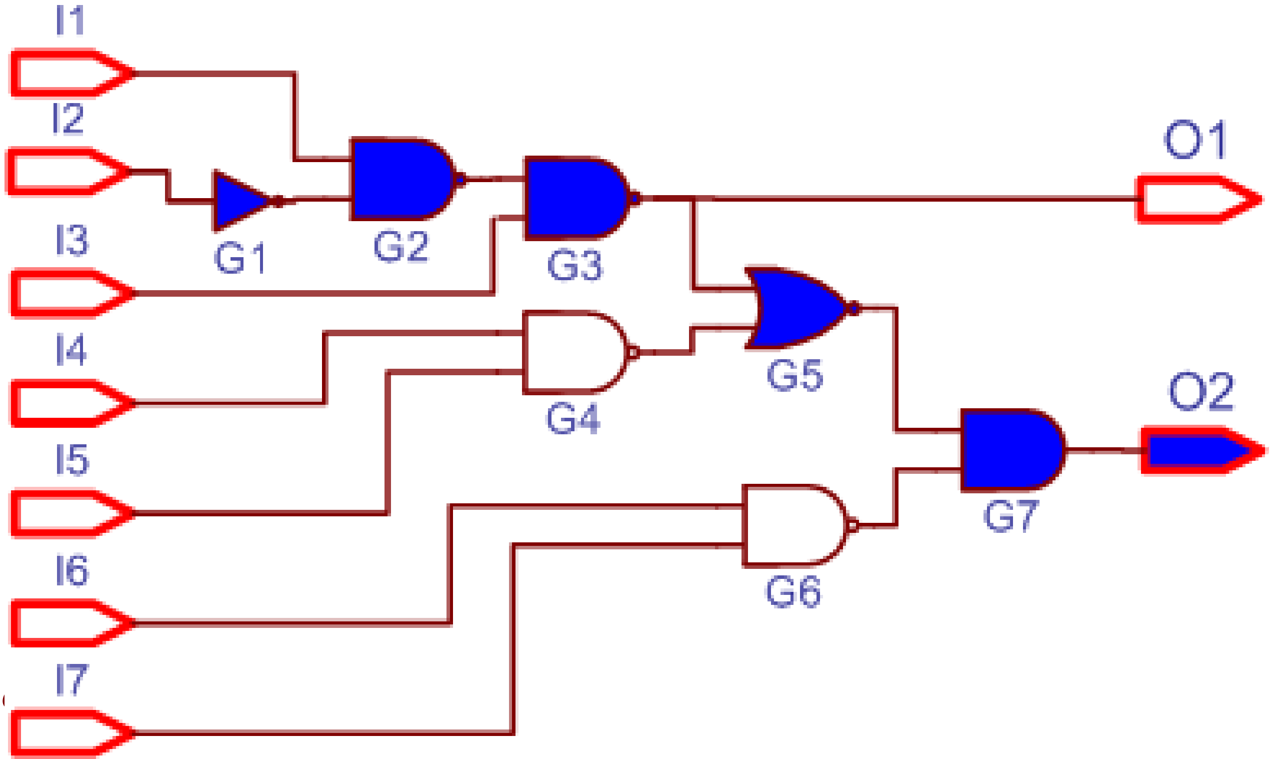
Definition

- Setup & Hold Time



Definition

- Critical Path
  - A path through combinational circuit that takes as long or longer than any other



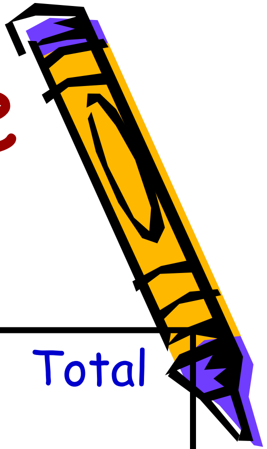
Multicycle Implementation

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:

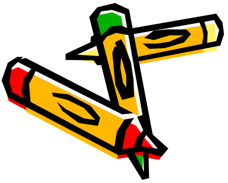
Clock cycle time = 4.7 + 0.1 = 4.8 ns

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

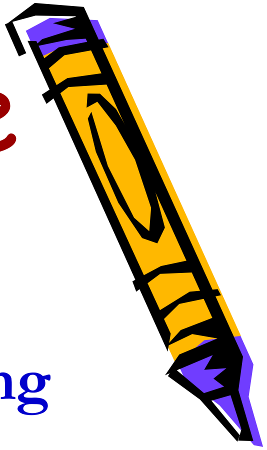
Multicycle Design:

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:

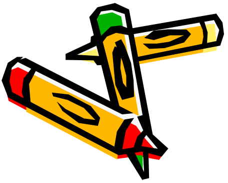
Clock cycle time = 4.7 + 0.1 = 4.8 ns

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

Multicycle Design:

Clock cycle time = 1.0 + 0.1 = 1.1

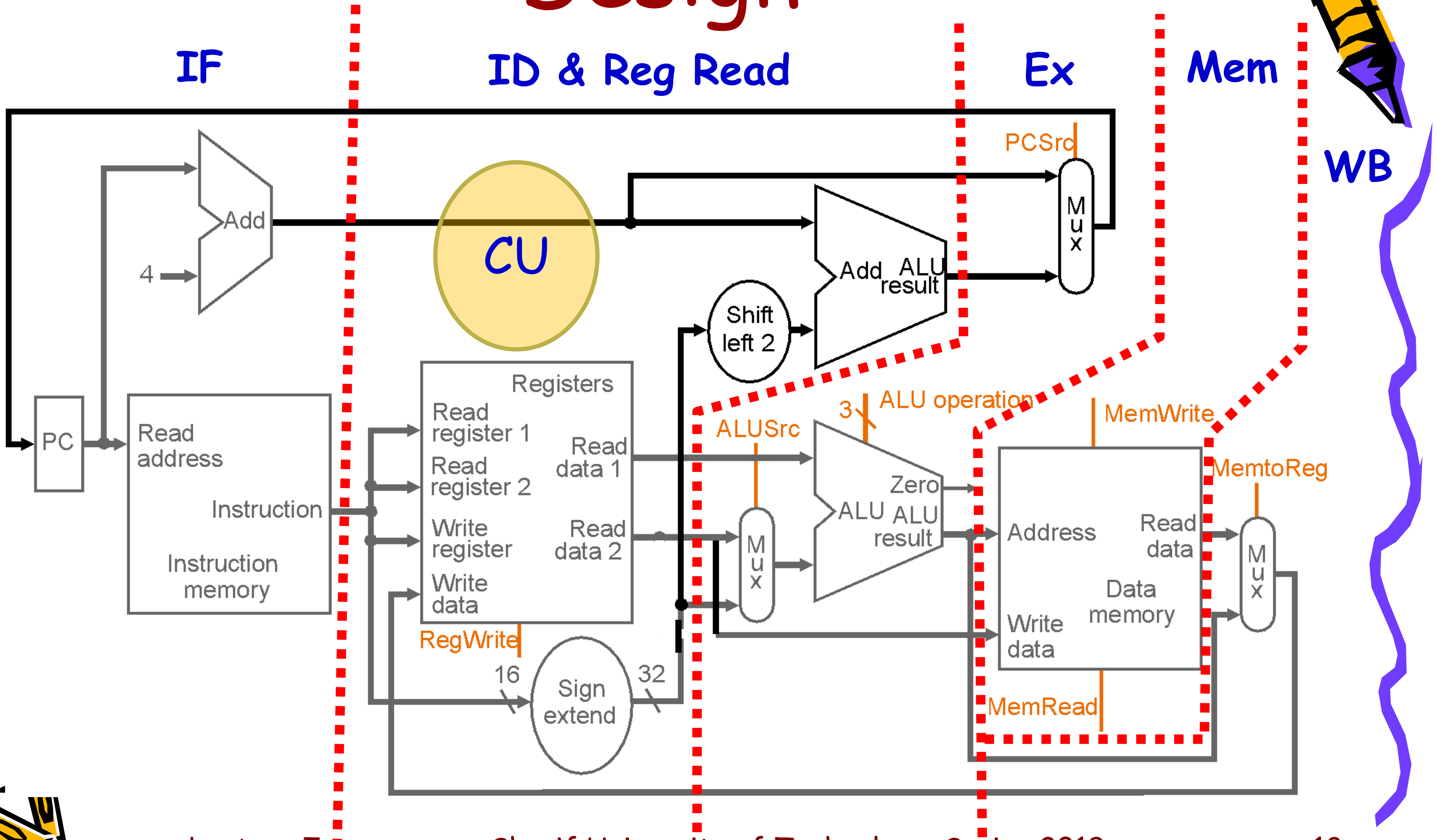
time/inst = CPI * 1.1 ns/cycle = 4.1 * 1.1 = 4.5



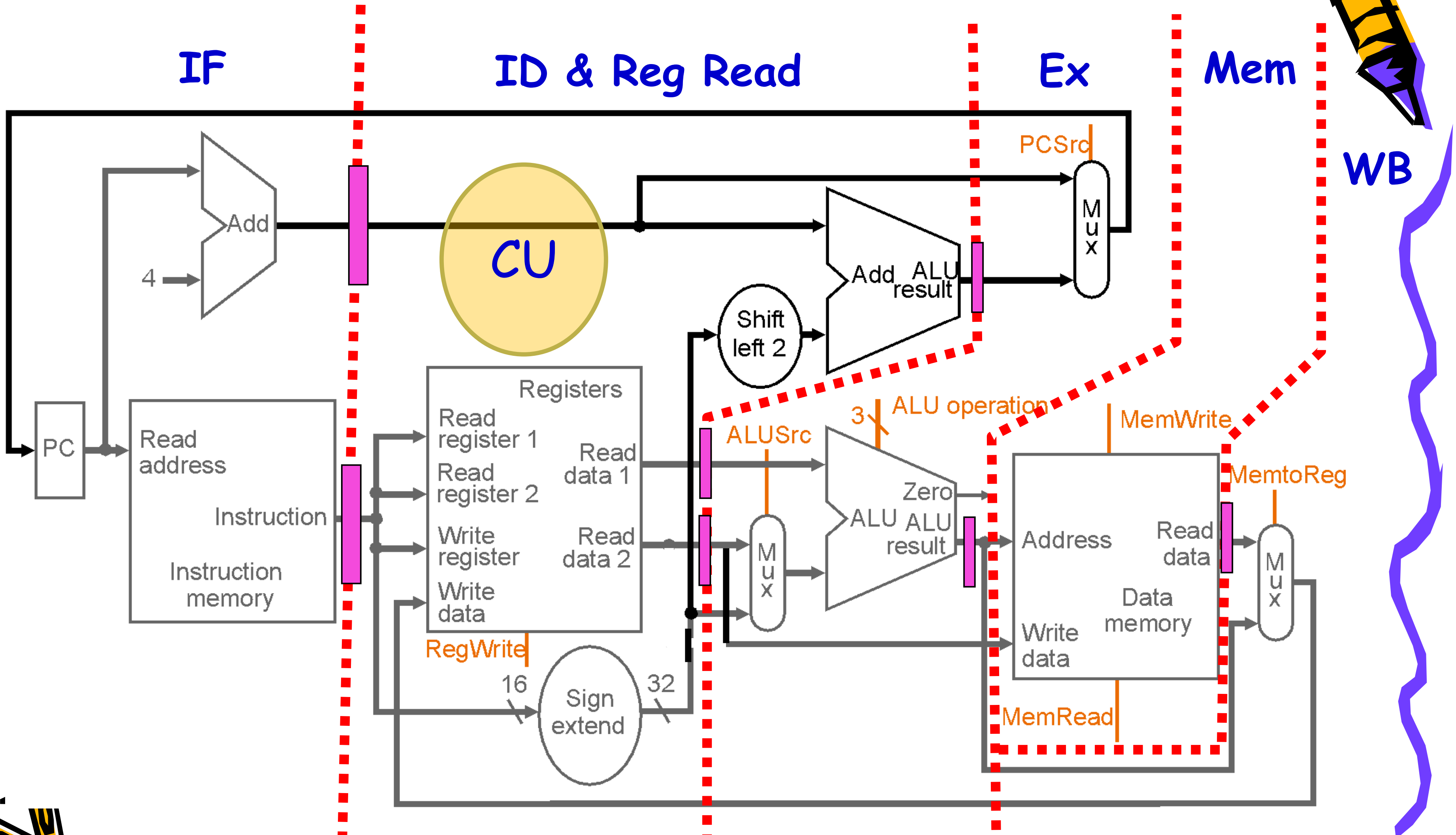
Will Multi-Cycle Design be Faster? (cont.)

- 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

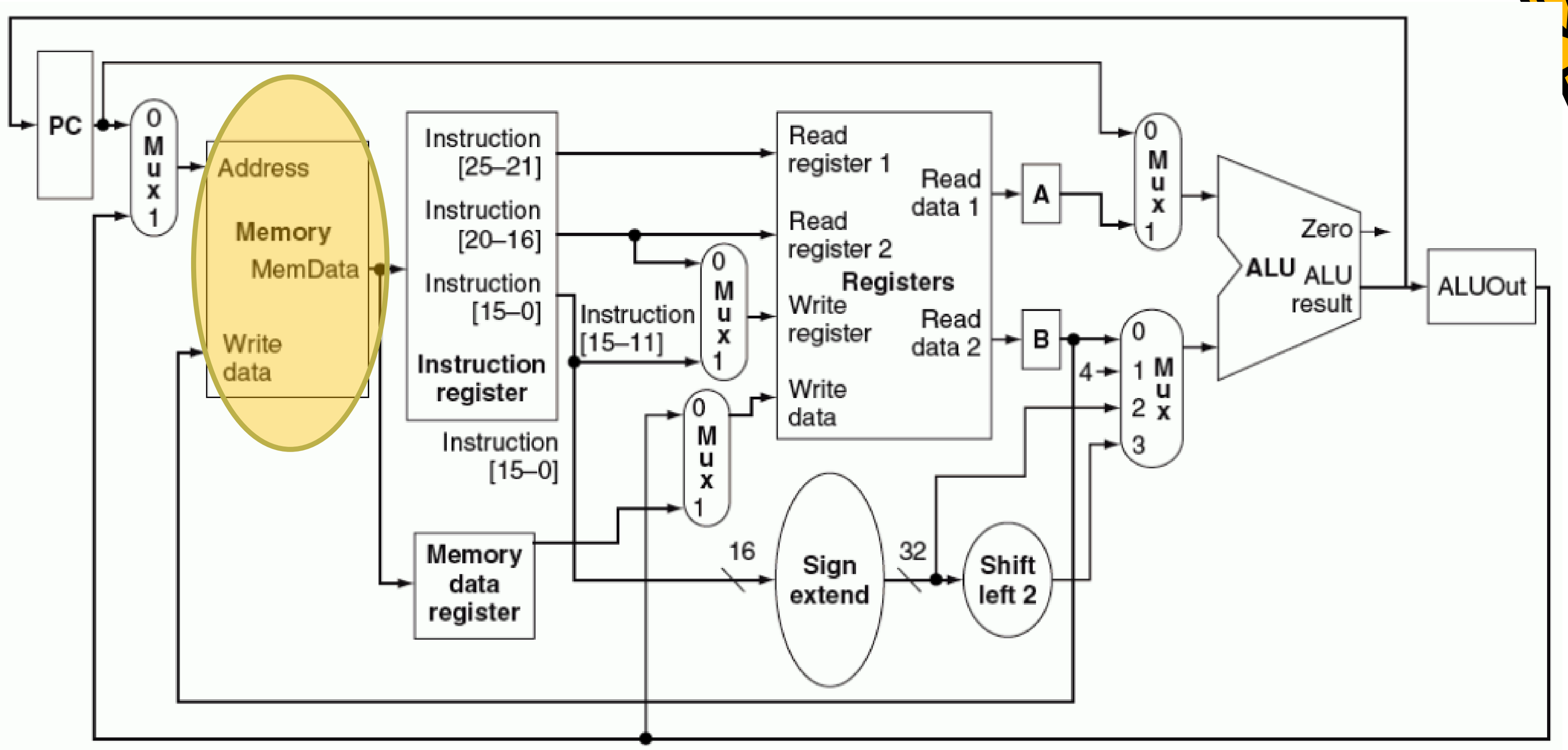
Partitioning Single-Cycle Design



Where to Add Registers?

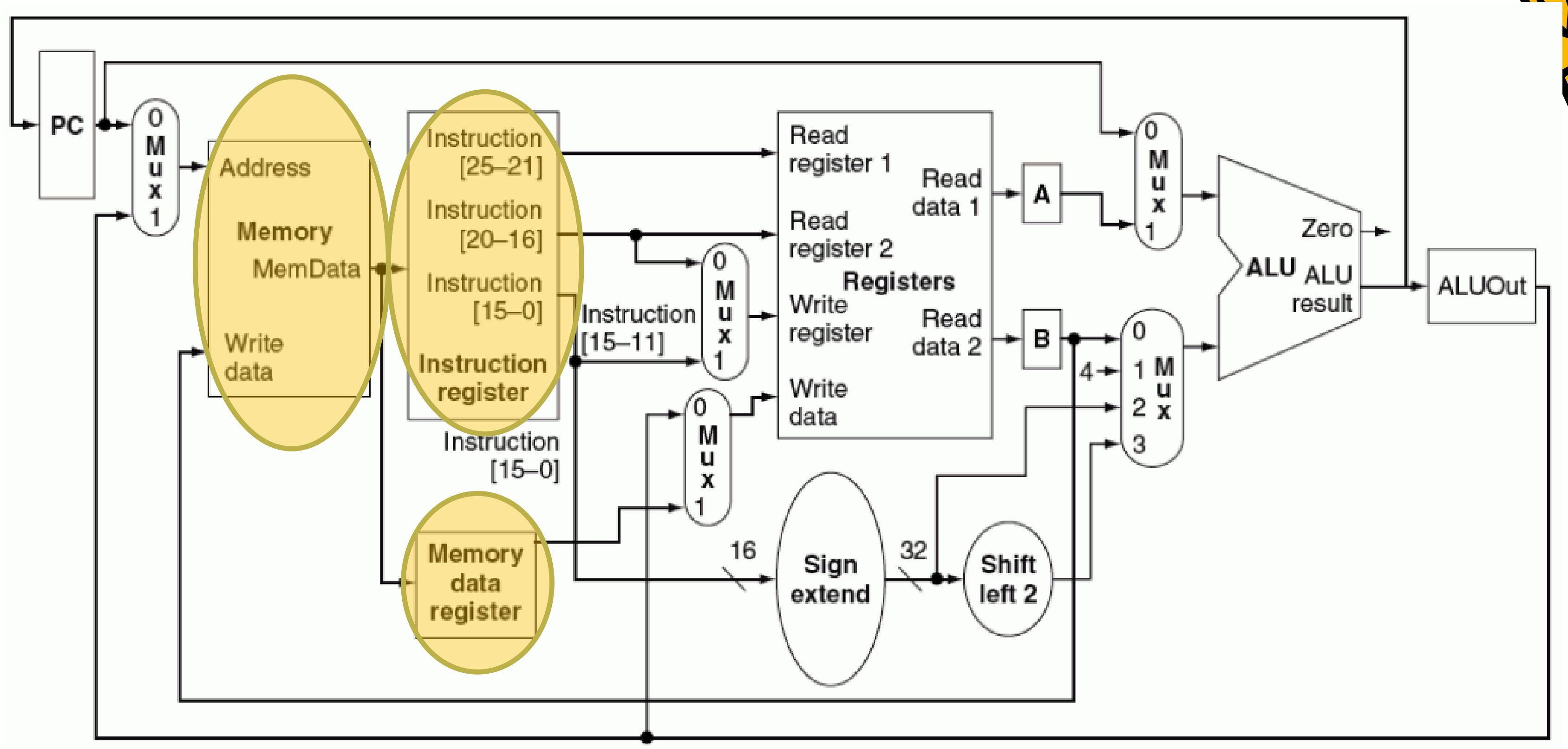


Multicycle Datapath



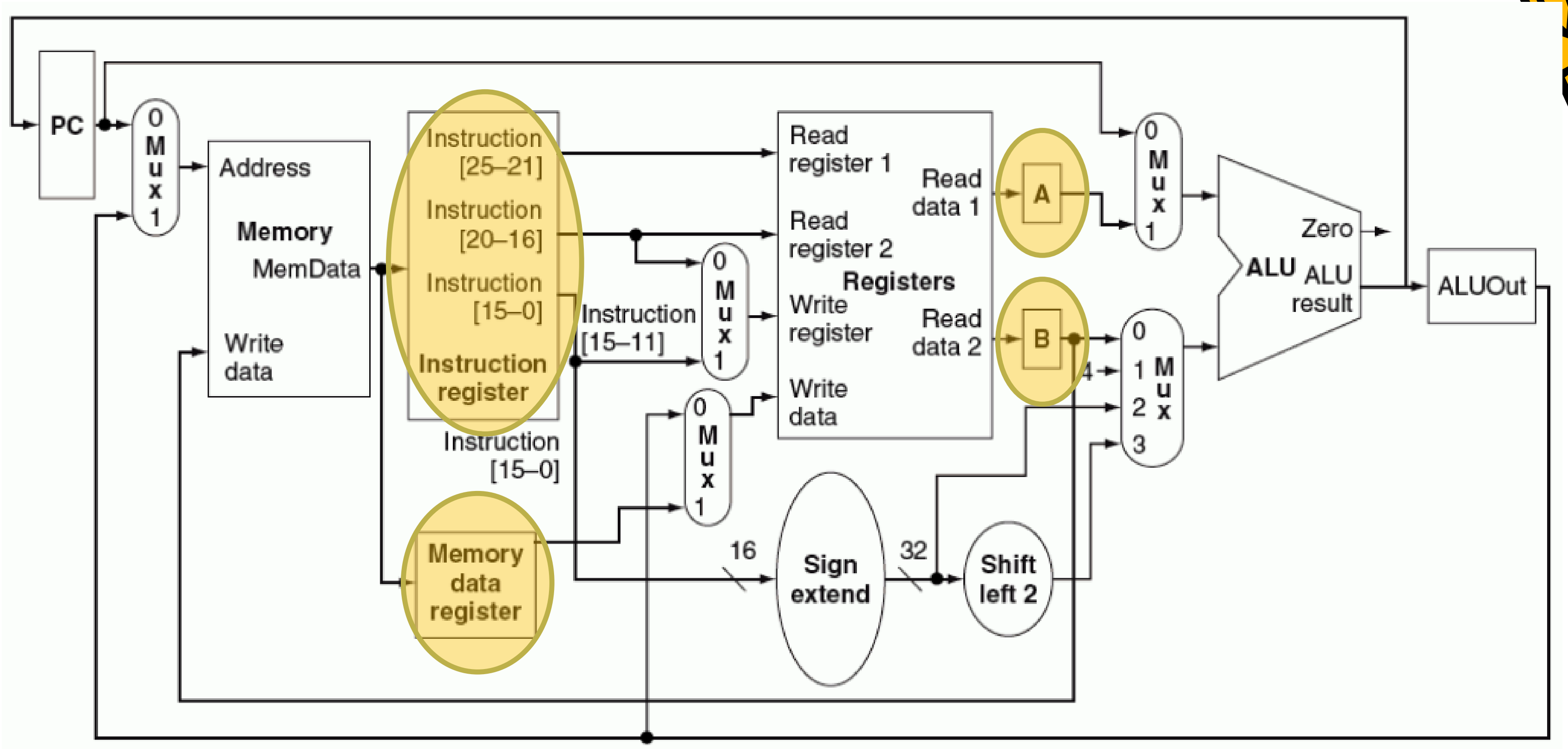
- Unified Memory
  - Used as both I-cache & D-cache
- Combines address busses of single-cycle datapath
  - Uses a MUX to select PC or ALU output

Multicycle Datapath (cont.)



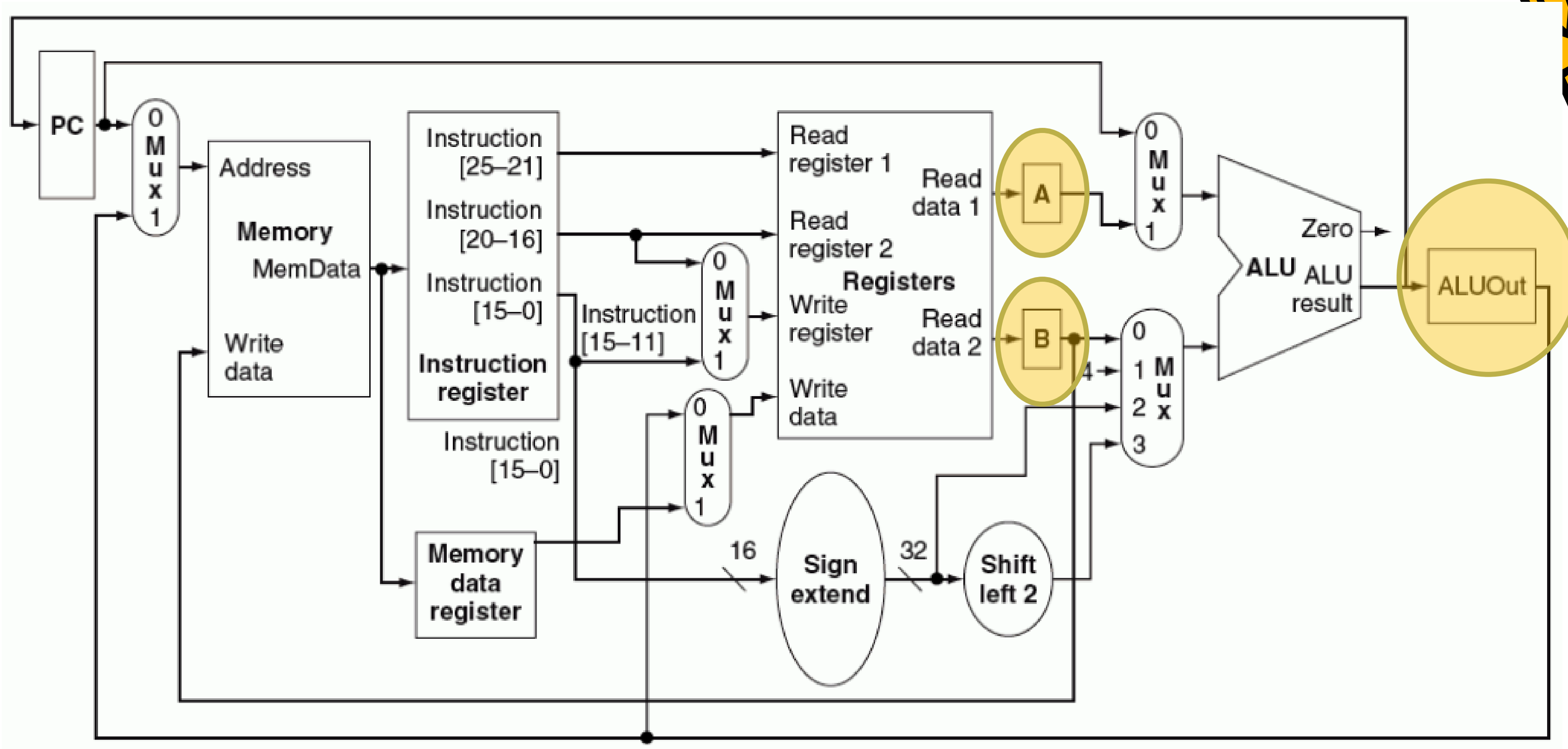
- **Instruction Register**
    - **IR ← Mem[PC]**
- **Memory Data Register**
    - **MDR ← Mem[ALUOut]**

Multicycle Datapath (cont.)



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

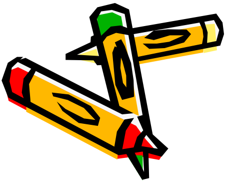
Multicycle Datapath (cont.)



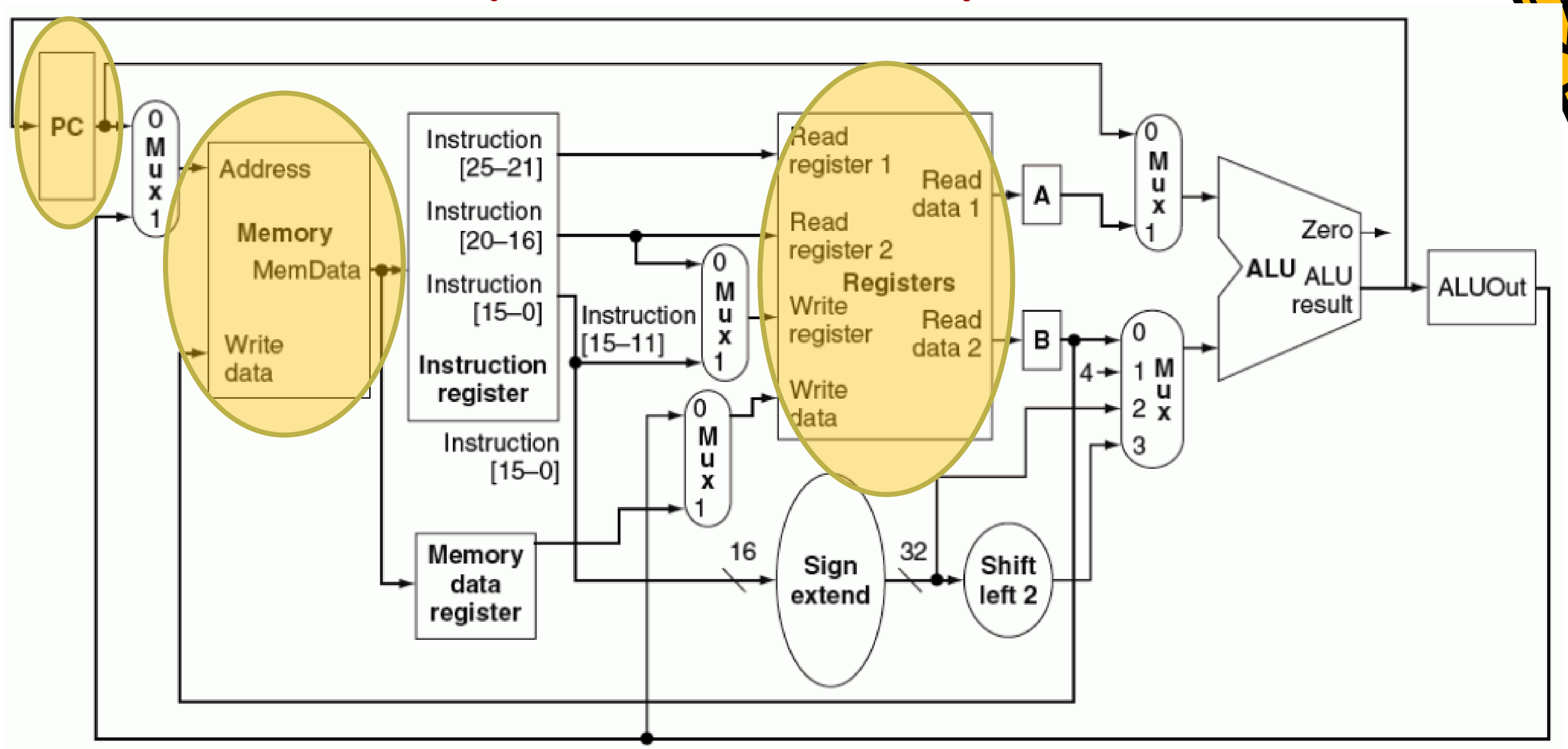
- ALUOut ← ALU result
- ALUOut is then either
  - Written to GPR
  - Or used as an address for Memory

Multi-Cycle Datapath 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

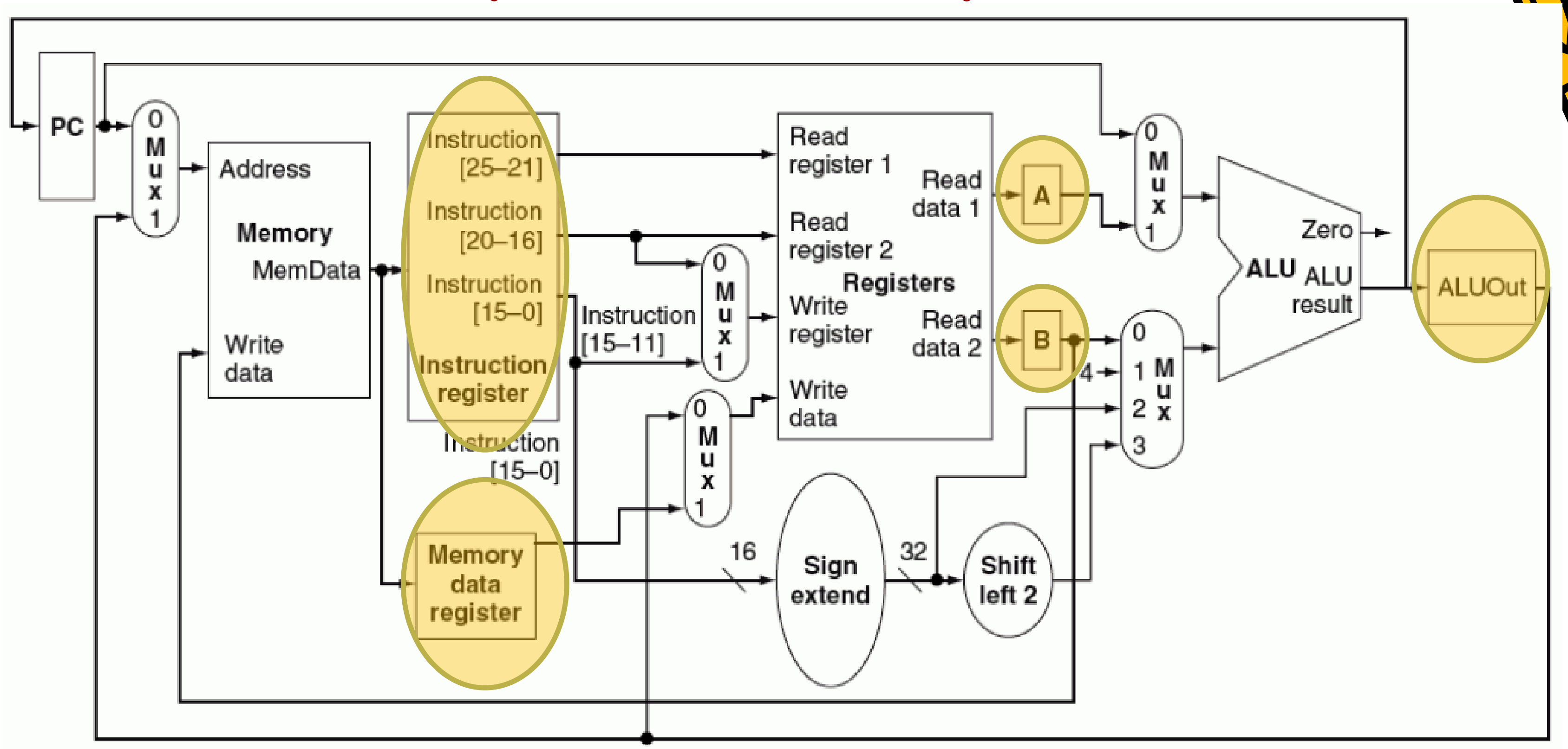


Multicycle Datapath (cont.)



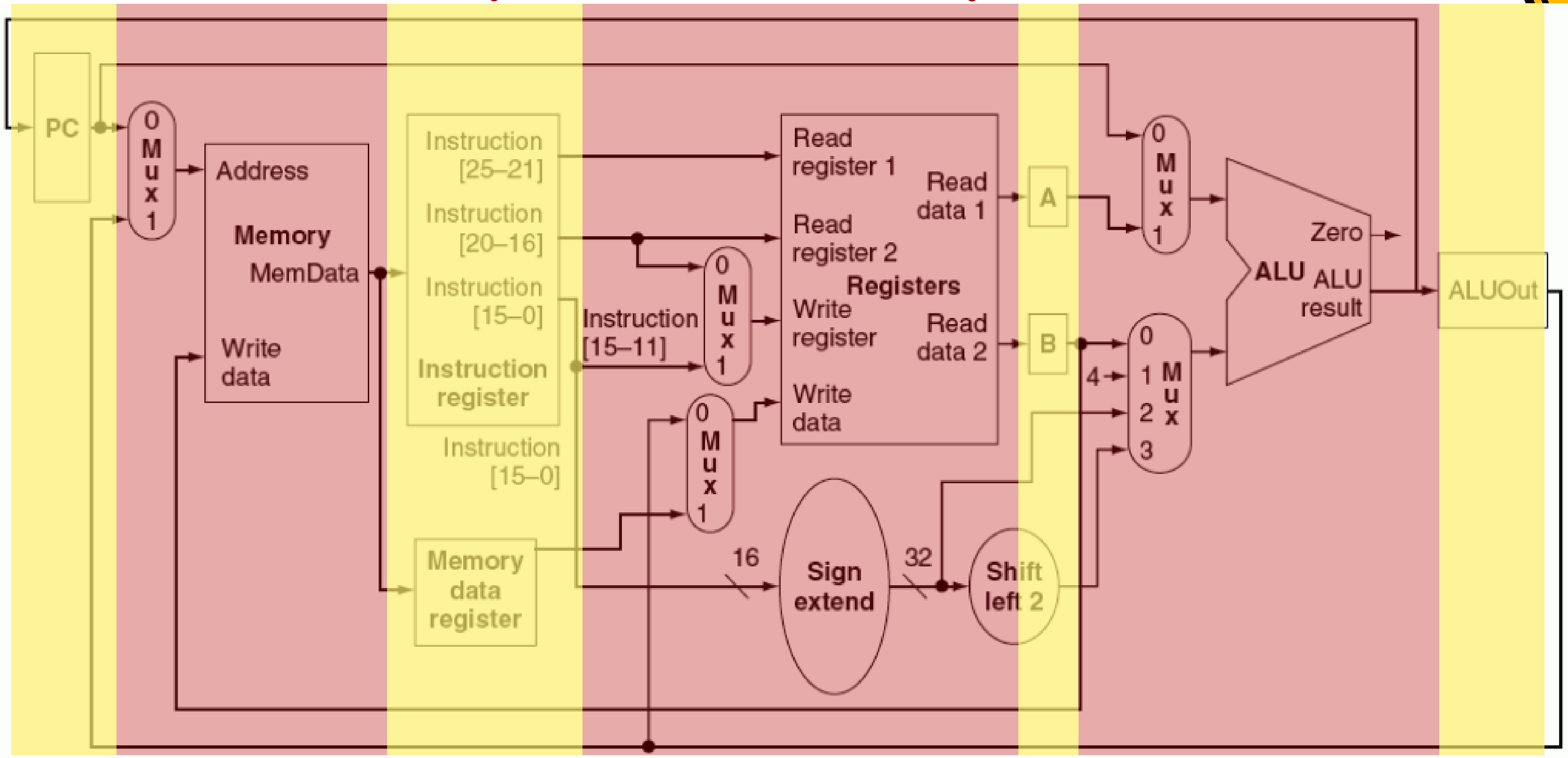
- User-Visible State Elements
  - PC
  - Memory
  - Register file

Multicycle Datapath (cont.)



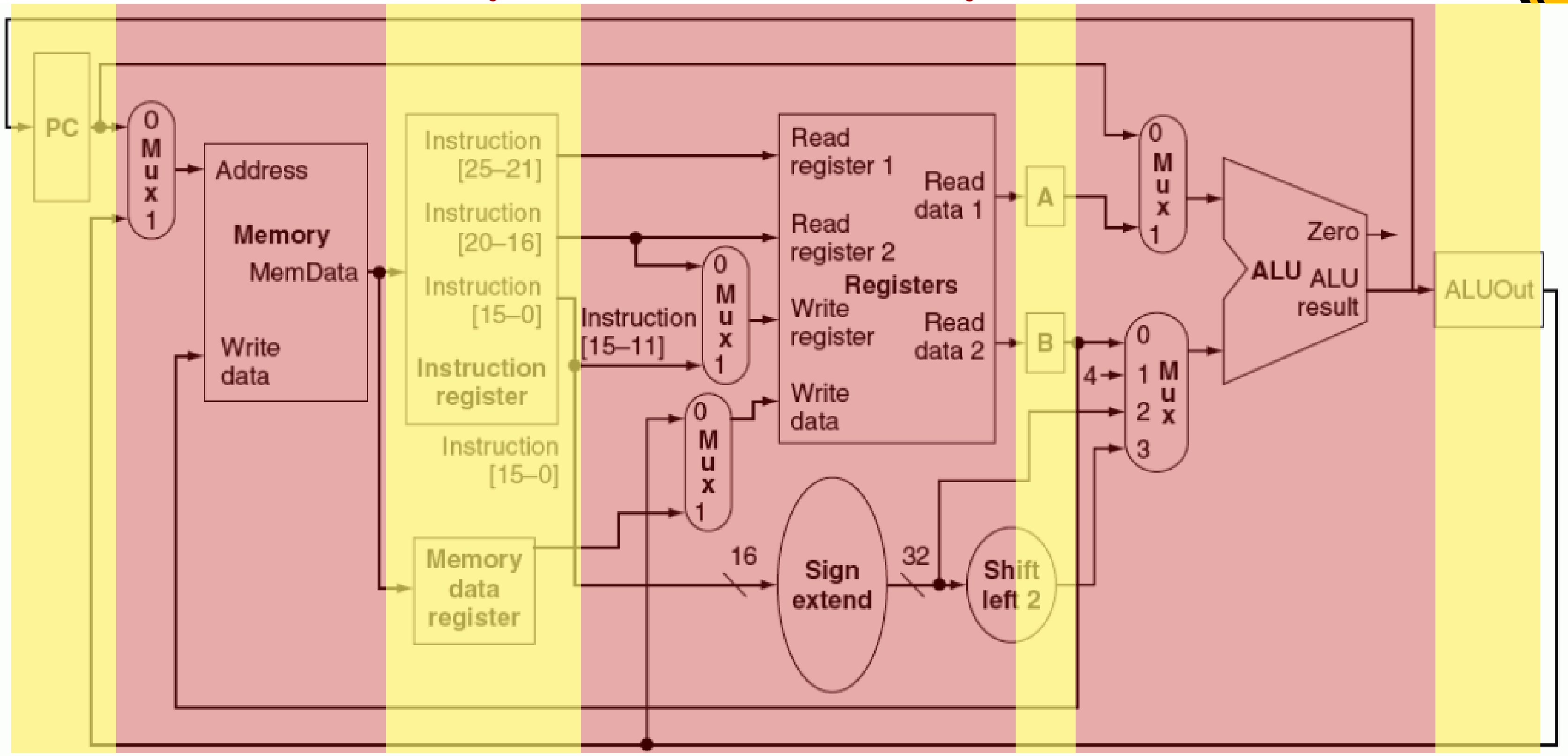
- Non-User-Visible State Elements
  - Instruction Register (IR)
  - Memory Data Register (MDR)
  - Reg A & Reg B
  - ALUout

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.)

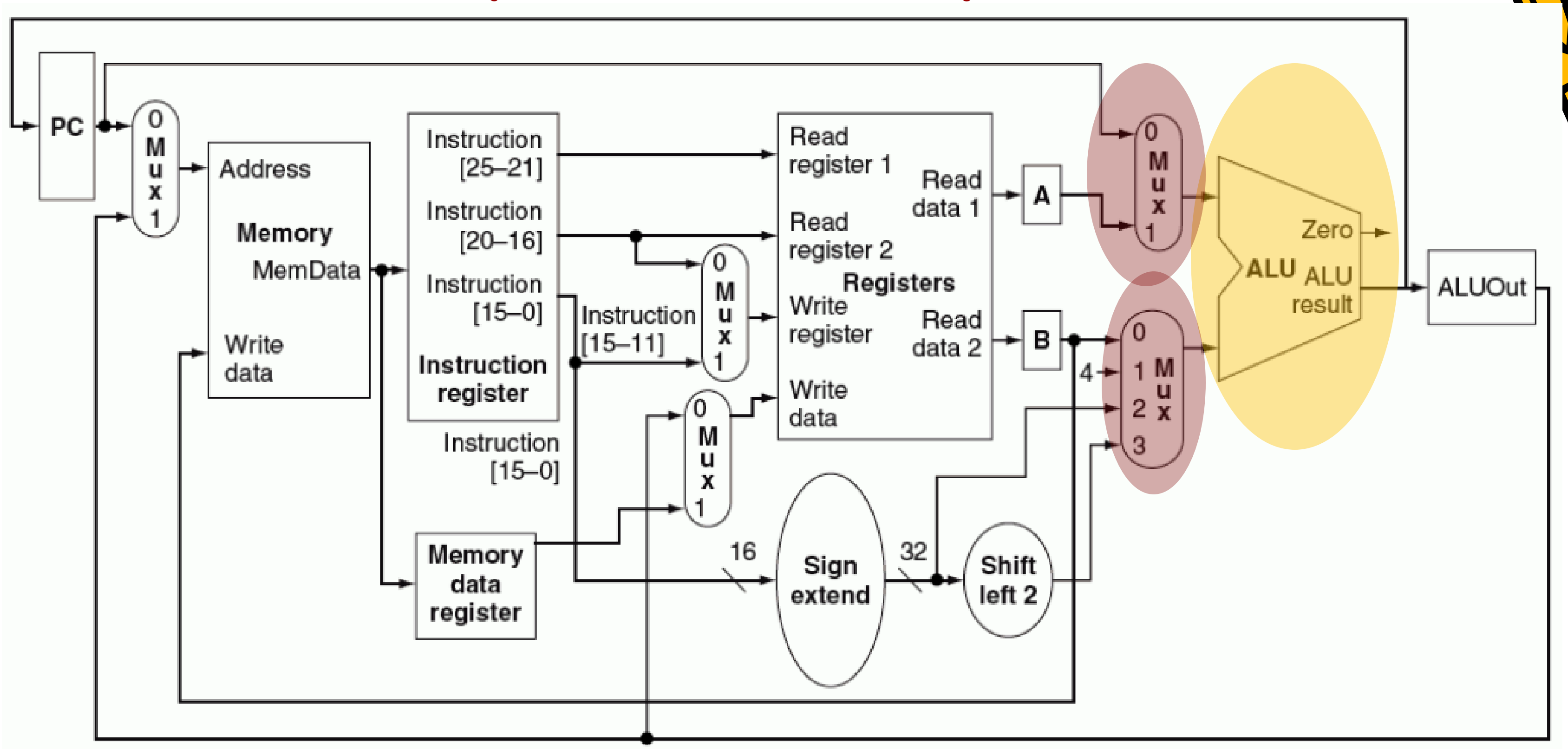


- No WR/RD control signal for non-visible regs (MDR,A,B, …)
  - WR=1, RD=1
- But IR needs to hold instr. until end of exec. of that instr.
- How about PC, memory, and GPR?
- How about read signal?

Read & Write Control Signals

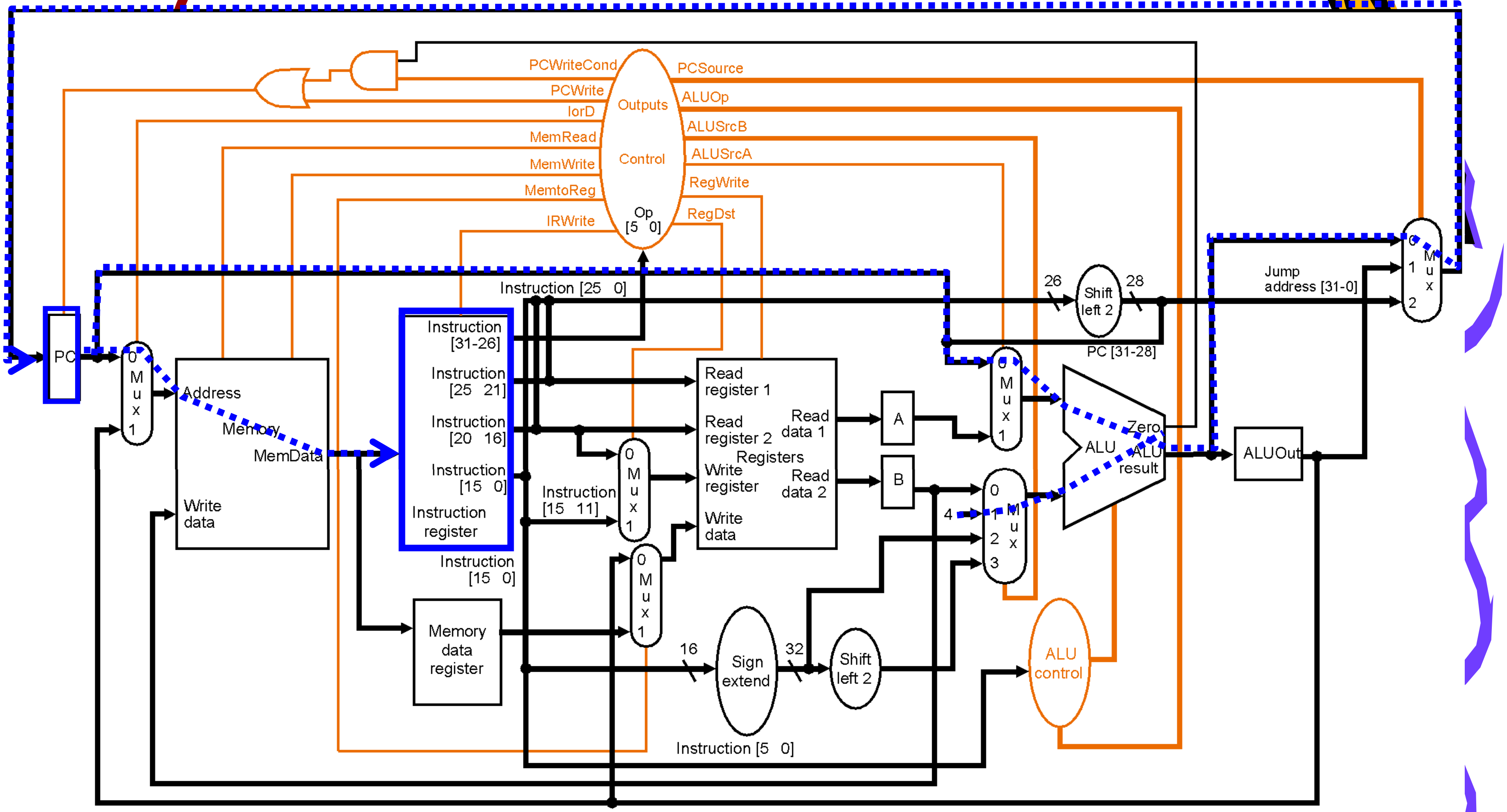
- Memory
  - Write signal required
  - Read signal required
    - If simultaneous read and write not possible
    - Twice decode circuitry for simultaneous RD/WR
- PC
  - If write signal = 1 ➔
    - PC incremented by 4 in IF & ID cycles
    - PC may capture wrong address in other cycles

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

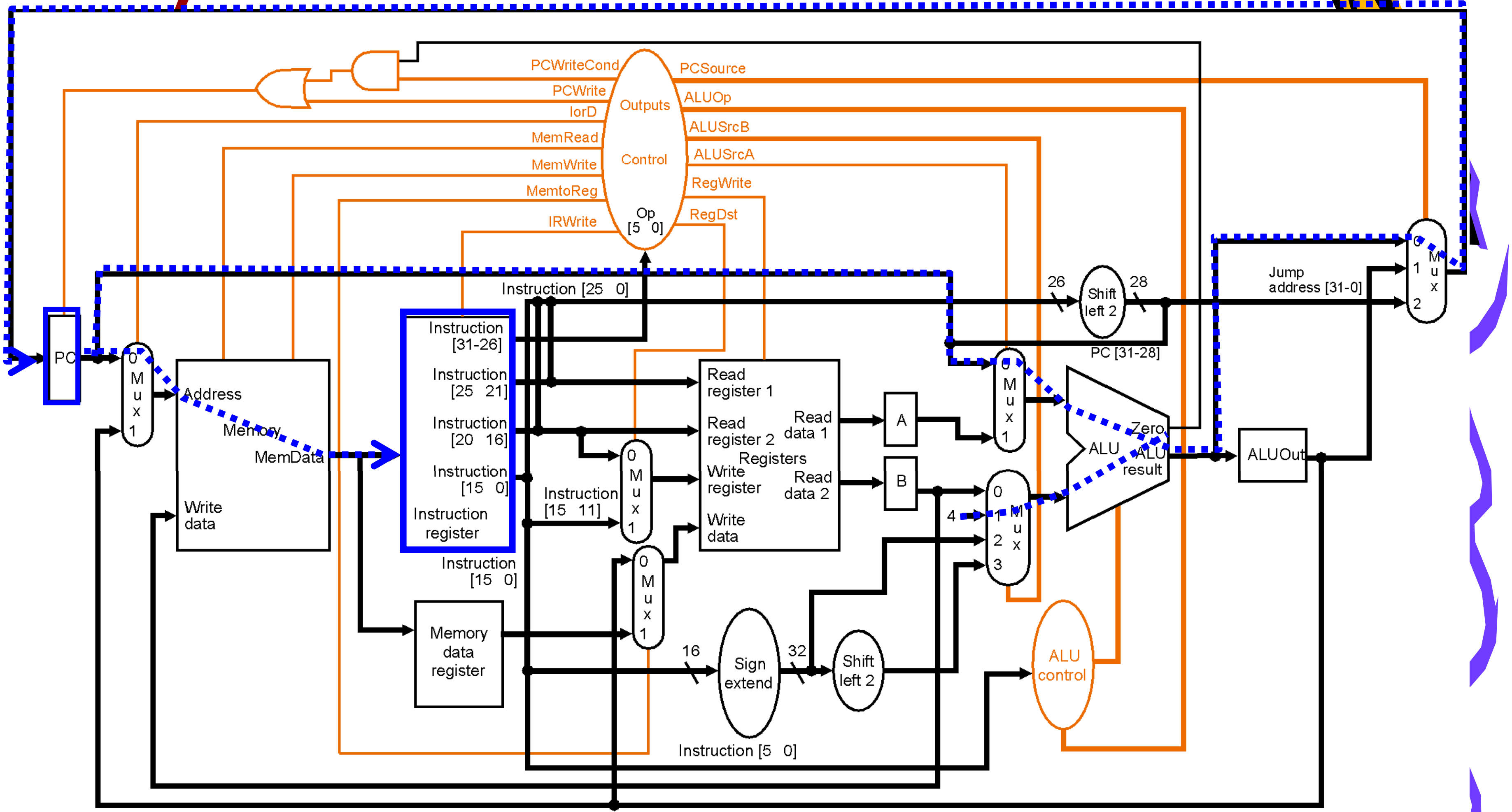


Datapath:

IR <= Mem[PC]

PC <= PC + 4

Cycle 1: Instruction Fetch



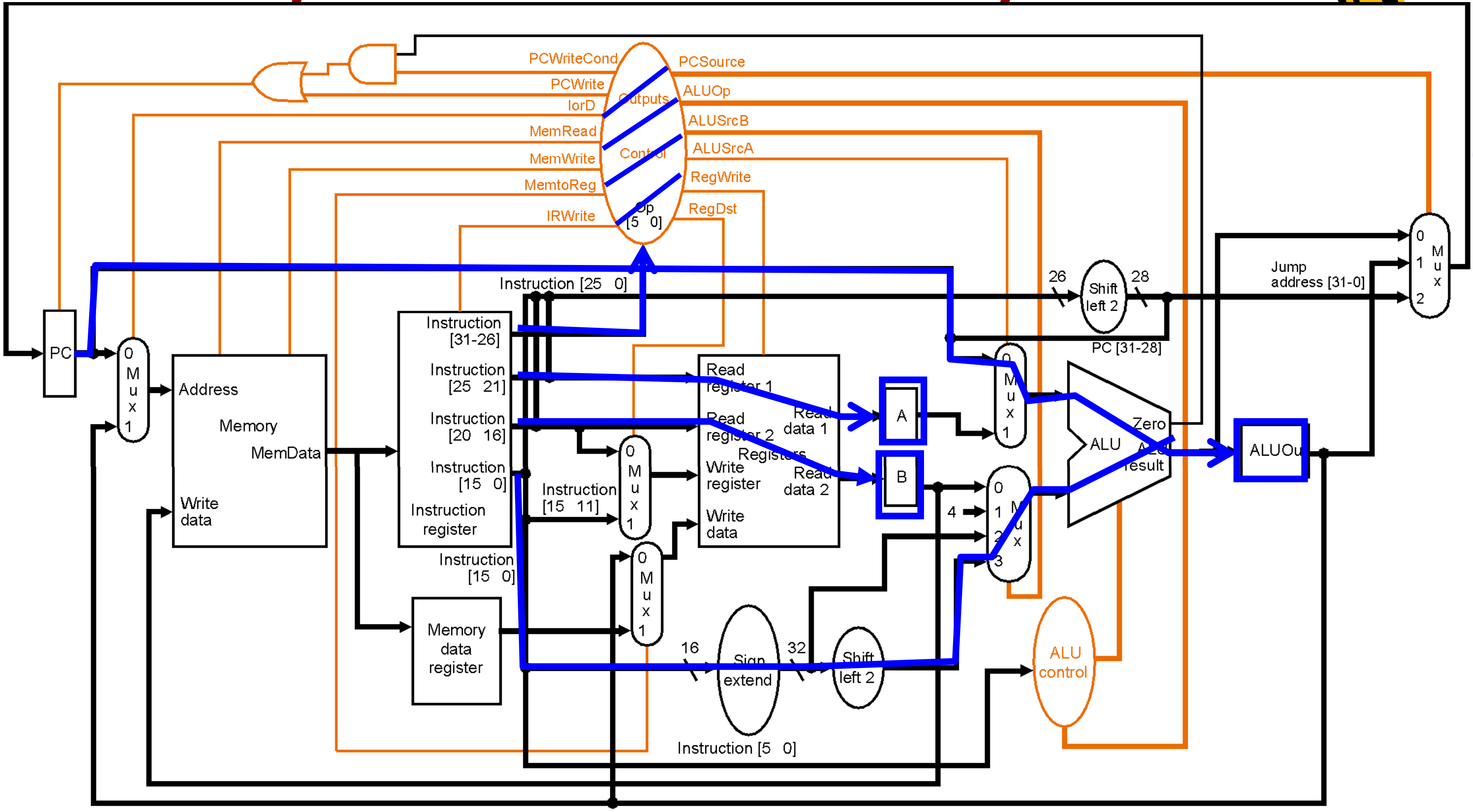
Control:

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)

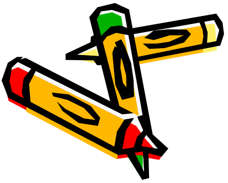
Cycle 2: ID & RF Cycle

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?



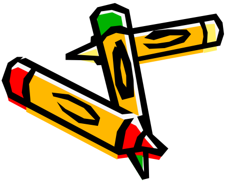
Cycle 2: ID & RF Cycle

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

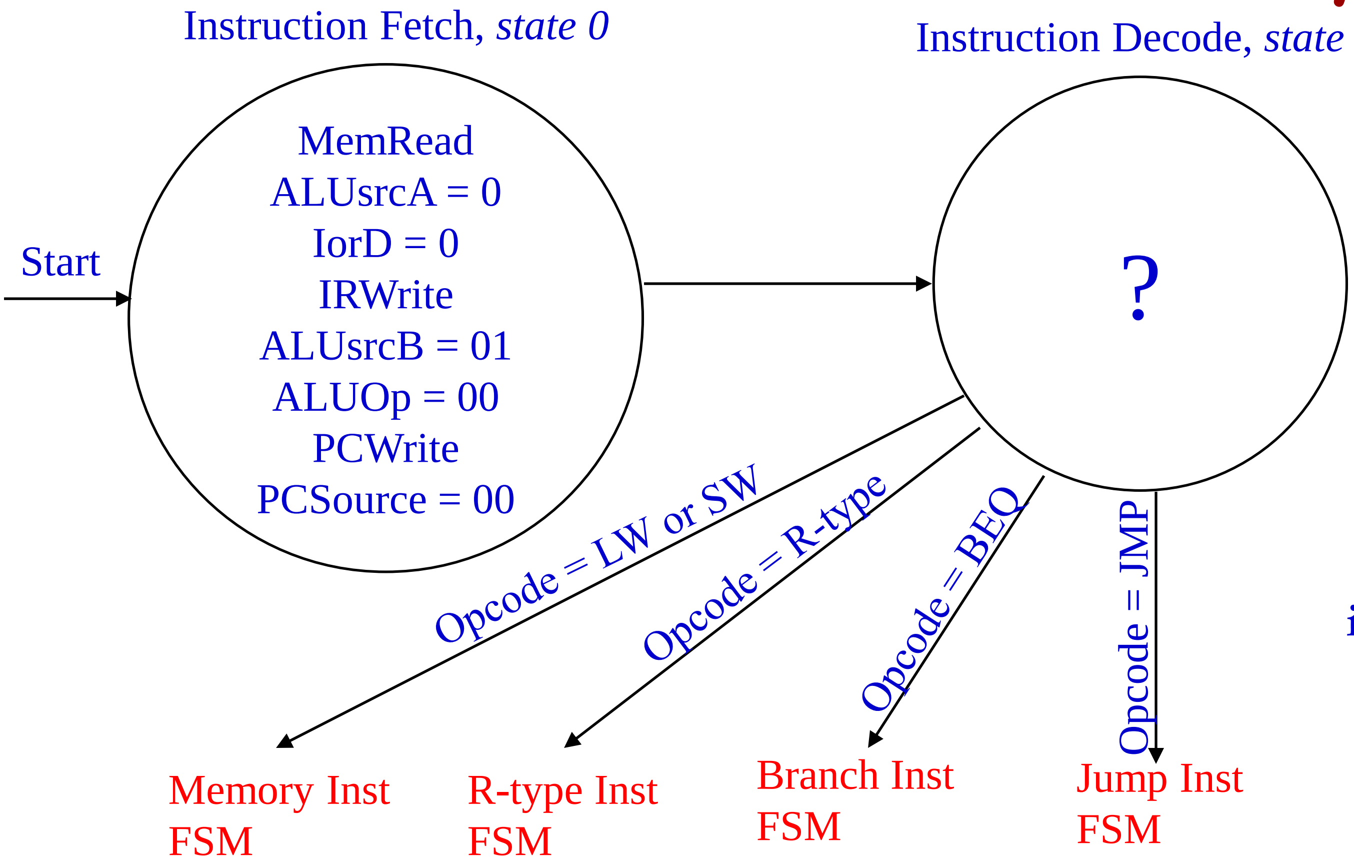
A <= GPR[IR[25-21]]

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



Instruction Fetch, *state 0*

Instruction Decode, *state 1*

Start

MemRead
ALUsrcA = 0
IorD = 0
IRWrite
ALUsrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

?

Opcode = LW or SW

Opcode = R-type

Opcode = BEQ

Opcode = JMP

Memory Inst
FSM

R-type Inst
FSM

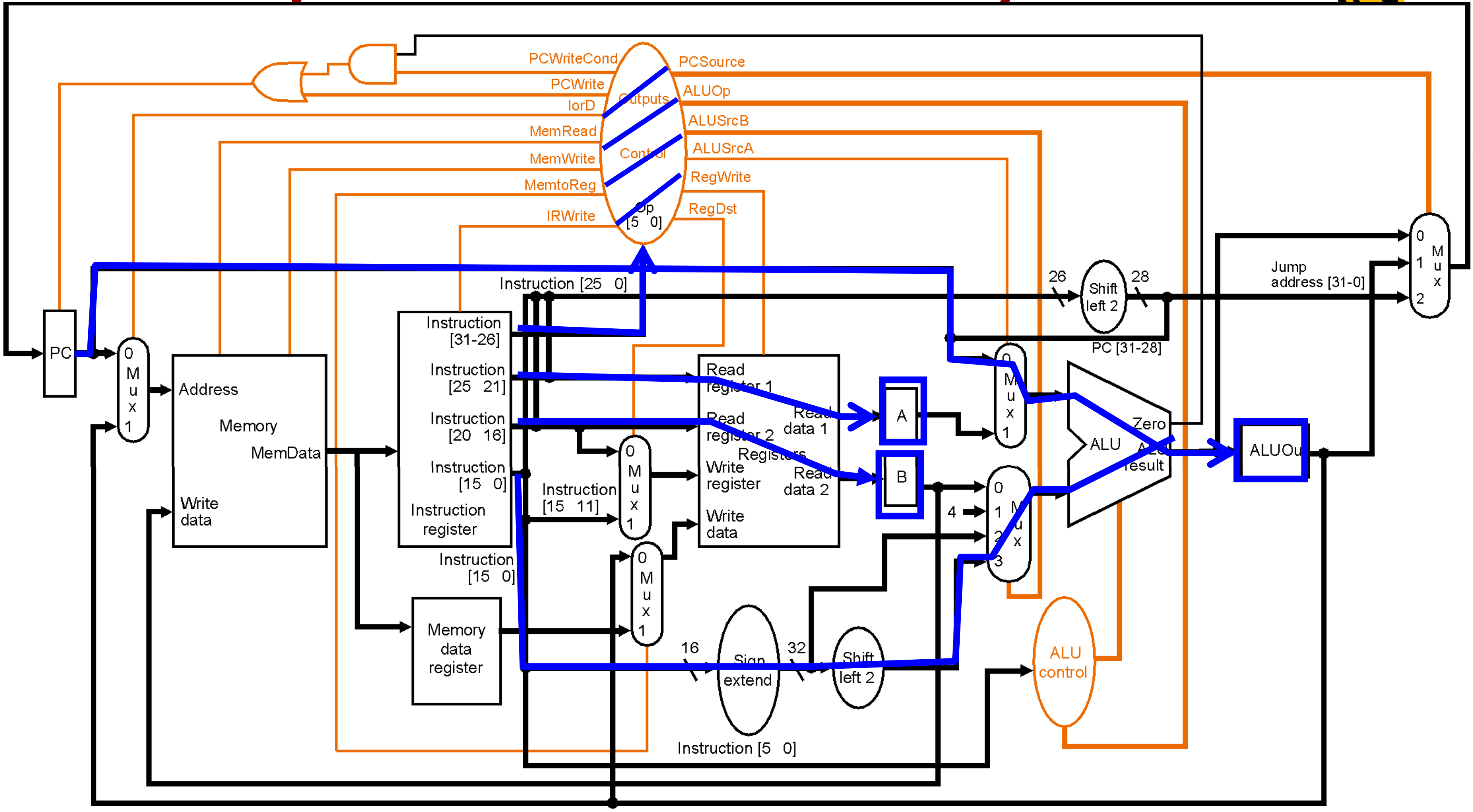
Branch Inst
FSM

Jump Inst
FSM

After cycle two,
we can treat different
instructions separately

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

Cycle 2: ID & RF Cycle

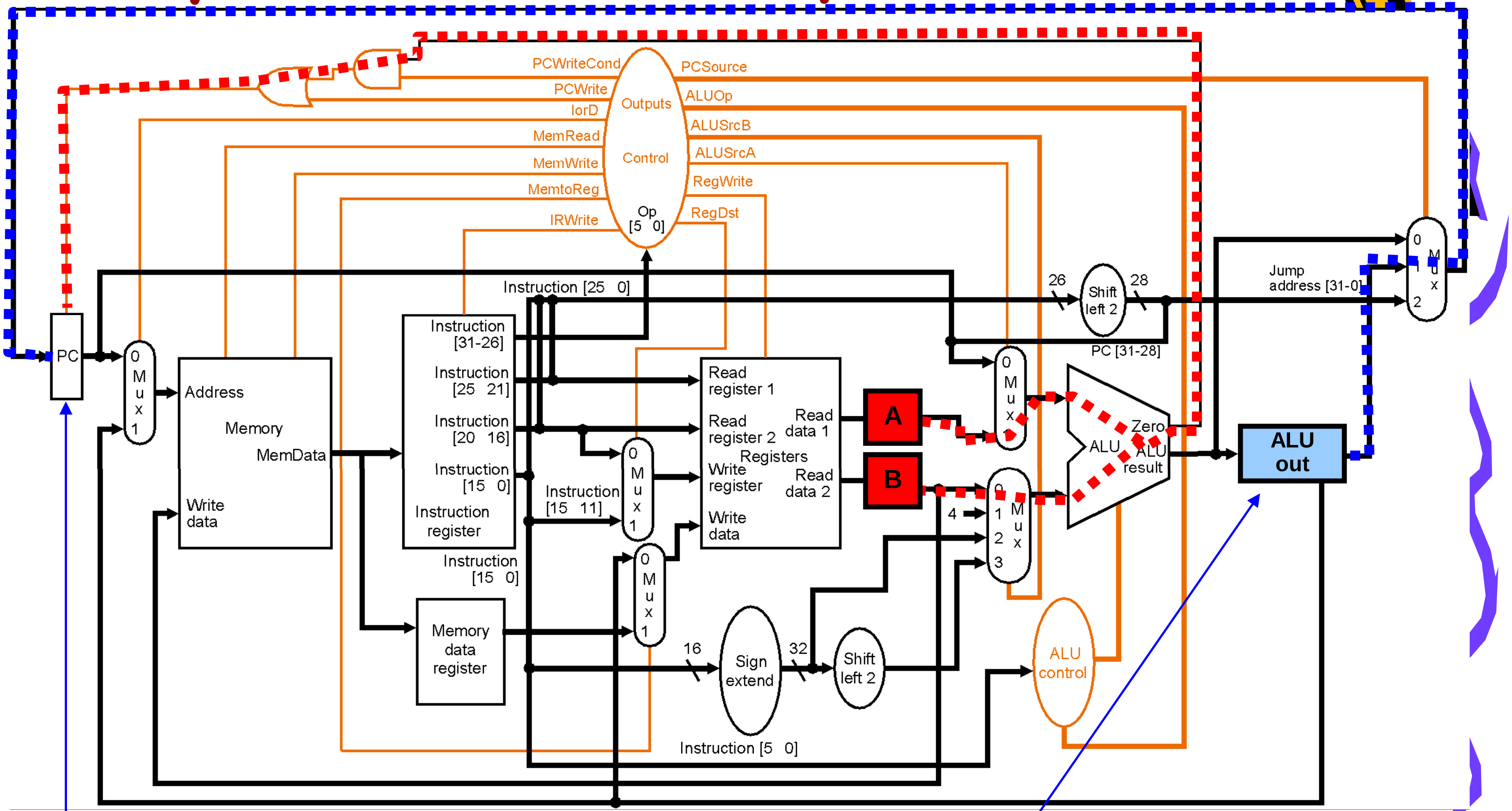


Control:

ALUSrcA=0, ALUSrcB=11, ALUOp=00

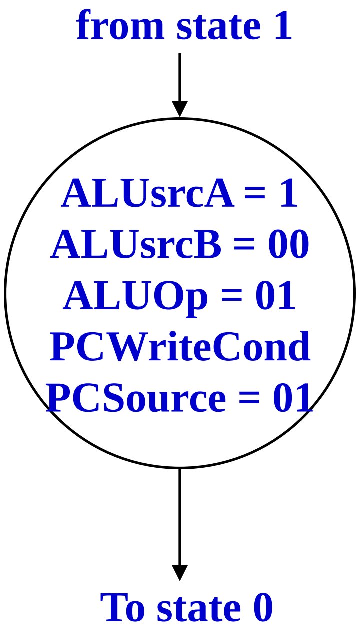
How about other signals? RegWrite, MemWrite, RegDest?

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



from state 1

ALUsrcA = 1
ALUsrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

To state 0

R-type Instructions

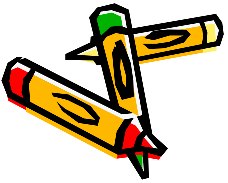
- Cycle 3 (EXecute)

```
ALUout = A op B
```

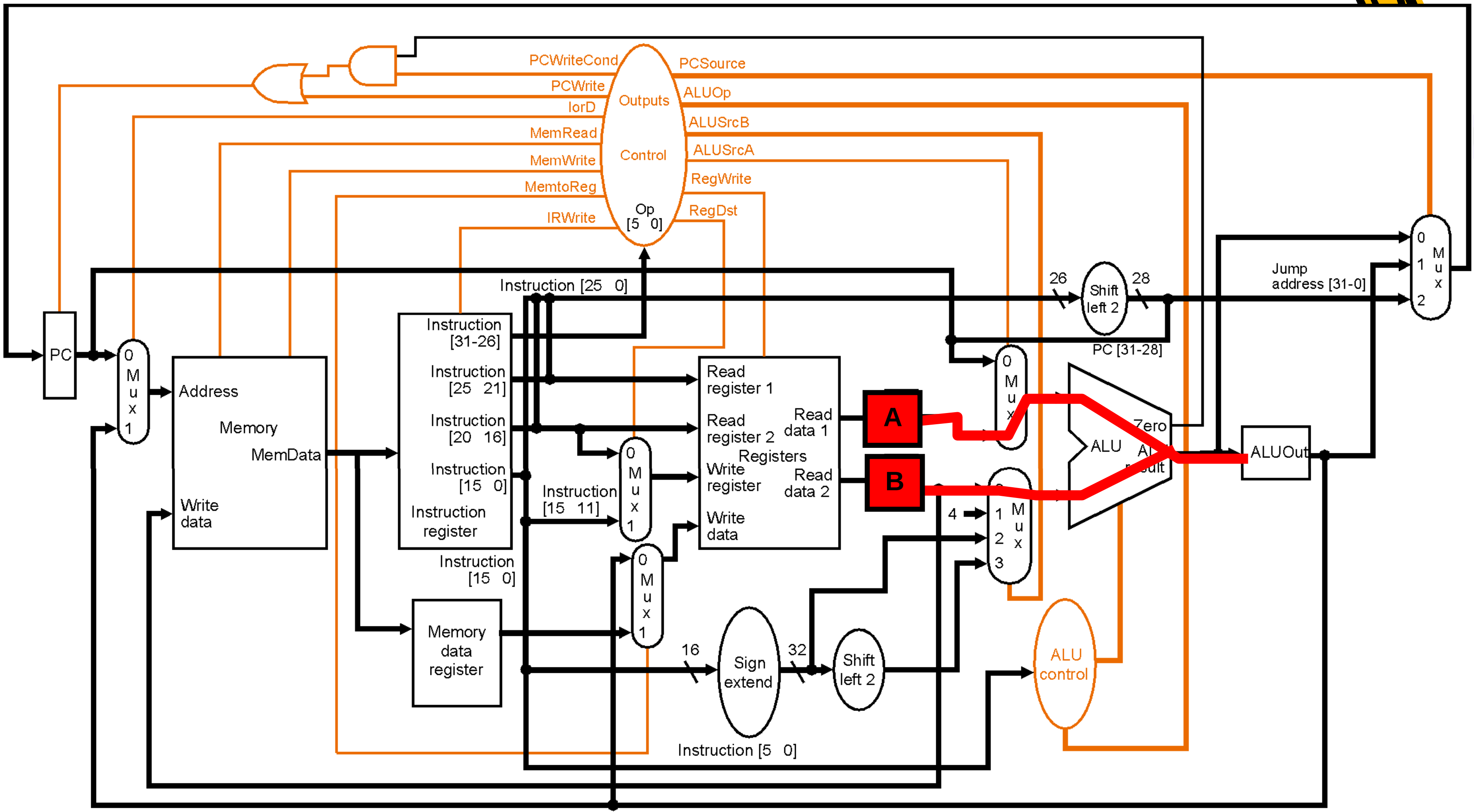
- Cycle 4 (WriteBack)

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

R-type instruction is finished



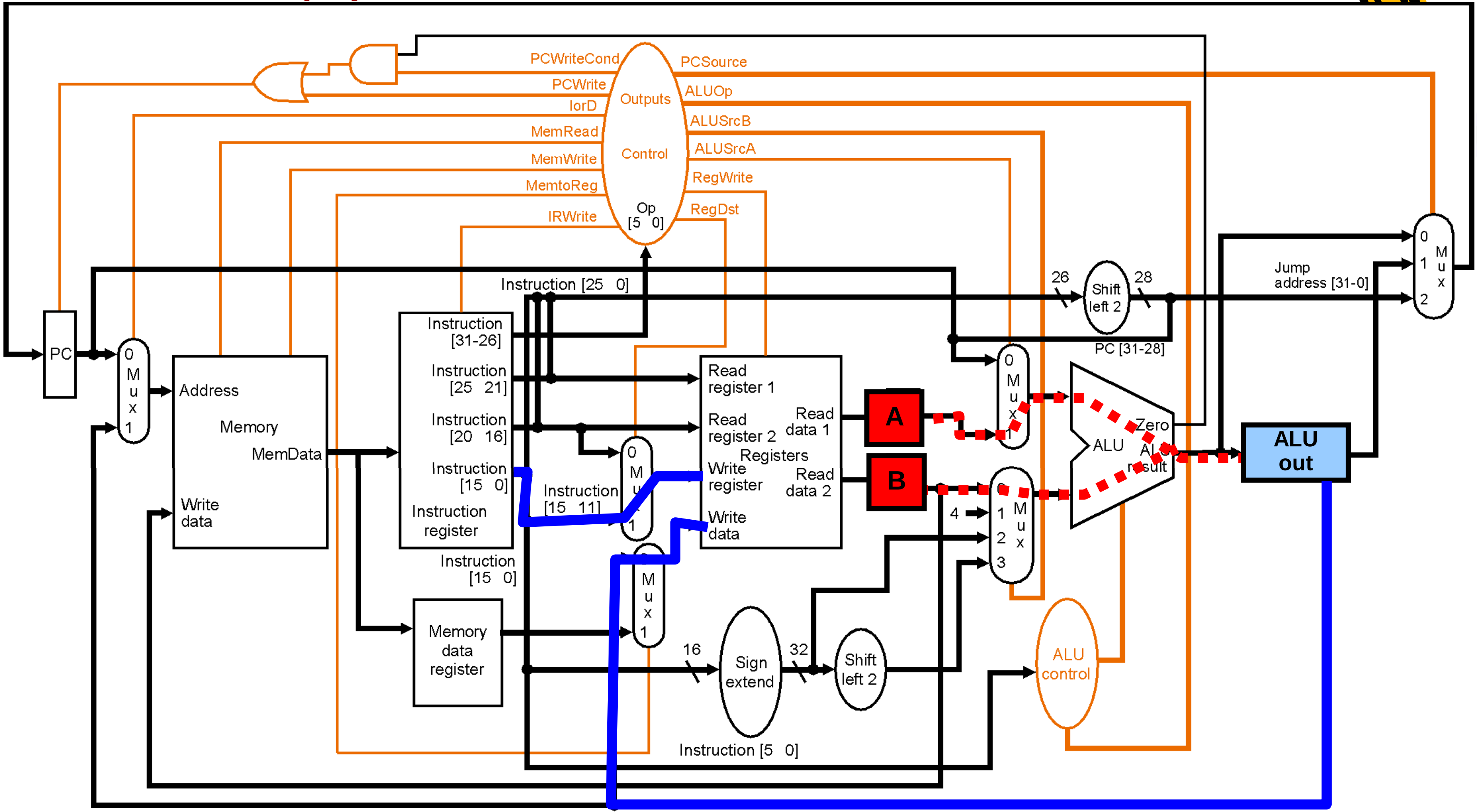
R-Type Execution



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

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

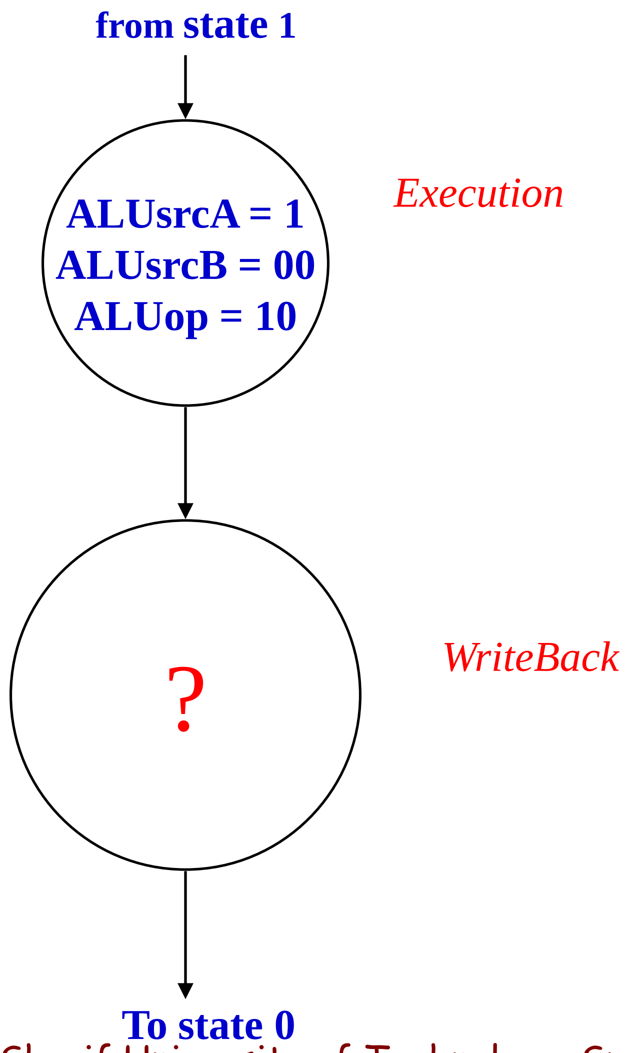
R-Type Execution & WB



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

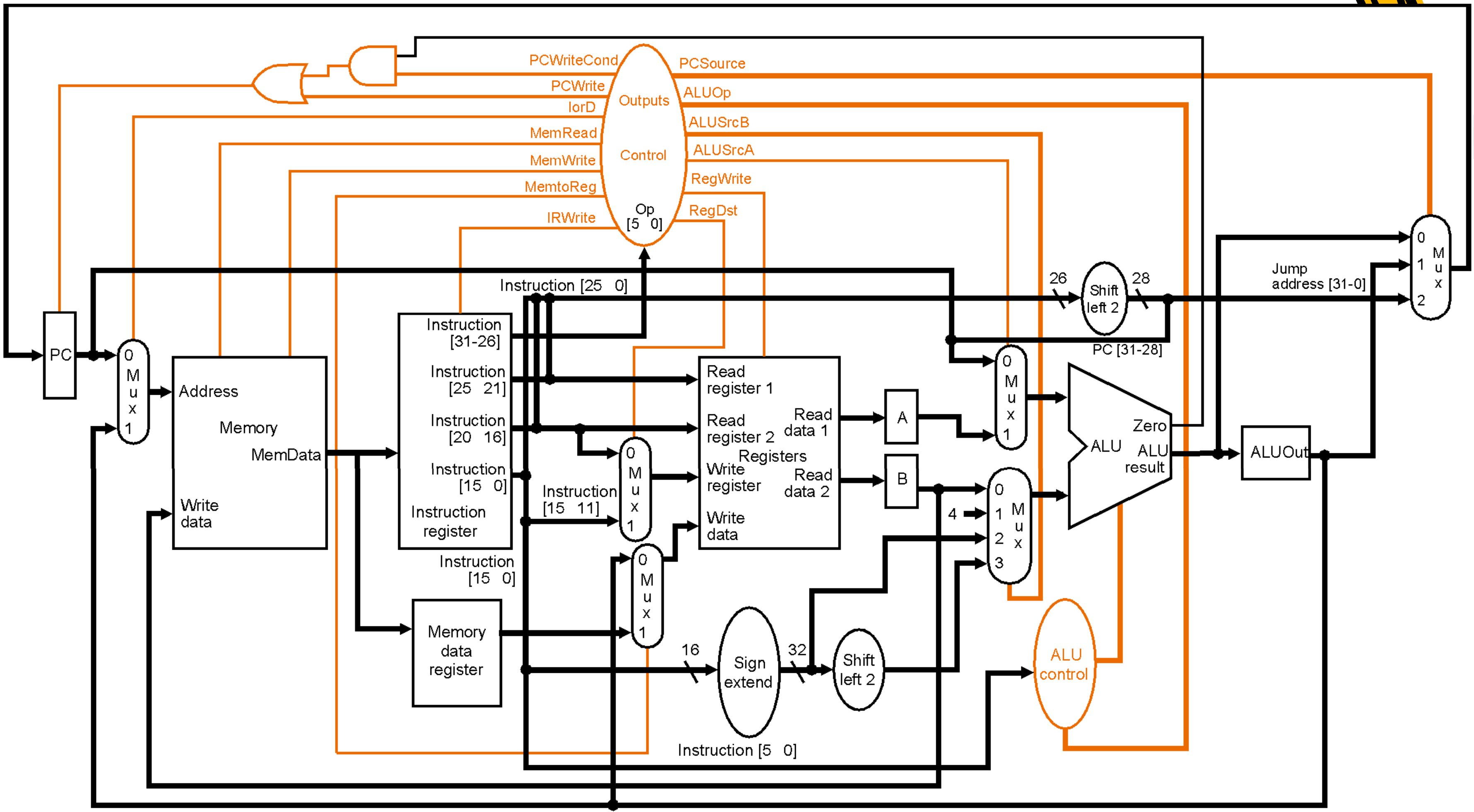
  ALUout = A + sign-extend(IR[15-0])
- Mem (cycle 4):
  - Access memory (read or write)

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

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

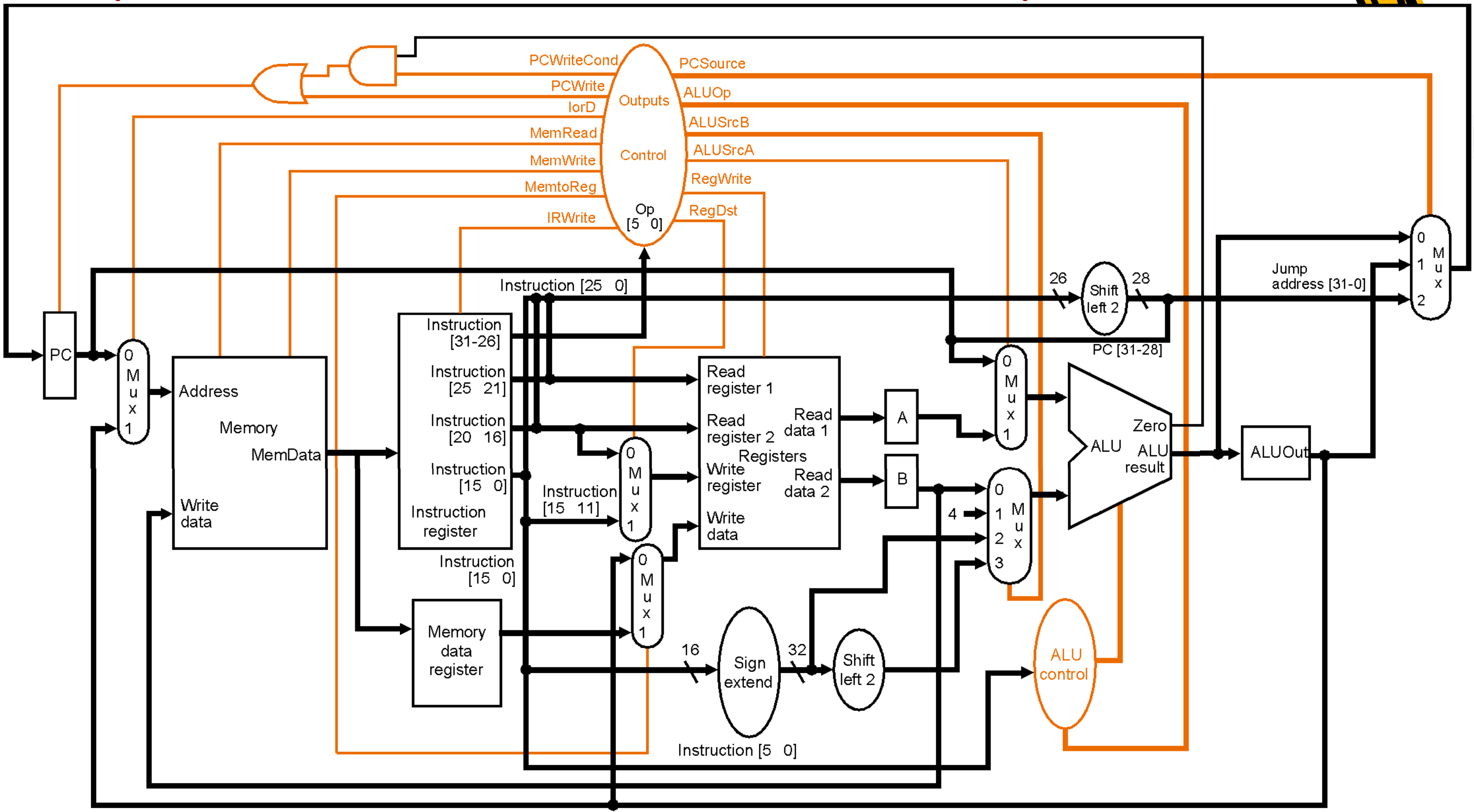
  GPR[IR[20-16]] = MDR

Cycle 3 for lw/sw: Address Computation



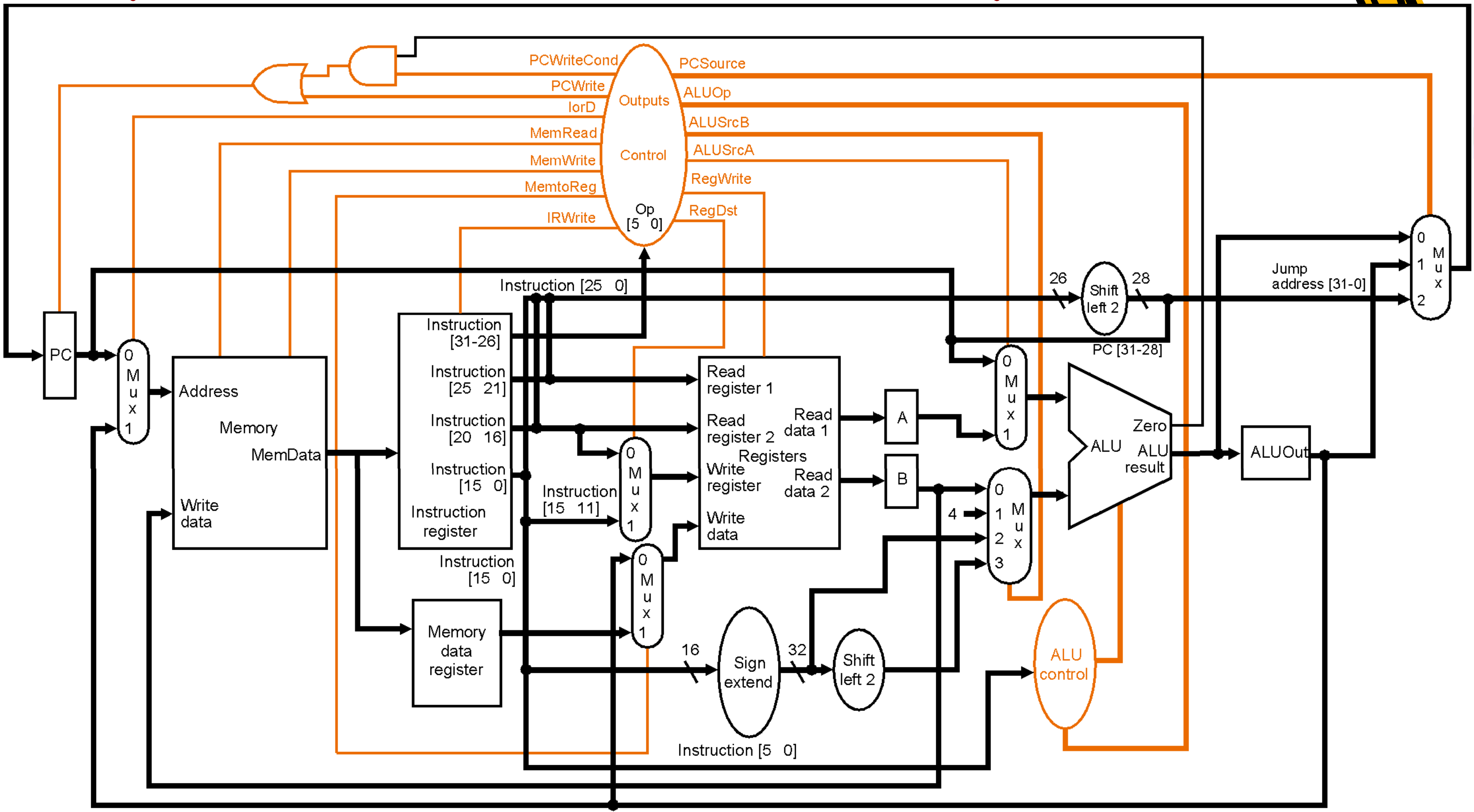
ALUout = A + sign-extend(IR[15-0])

Cycle 4 for Store: Memory Access



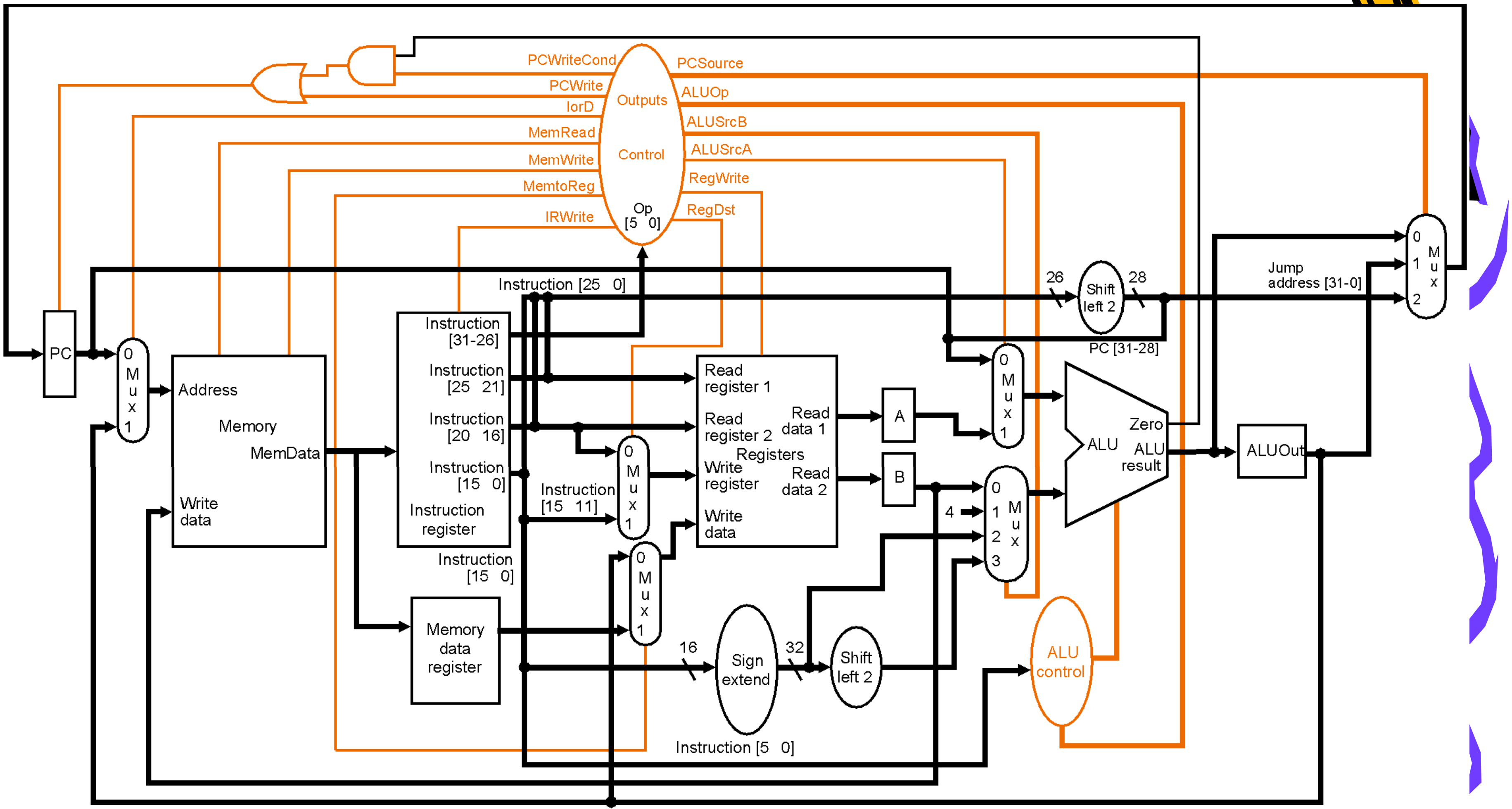
Memory[ALUout] = B

Cycle 4 for Load: Memory Access



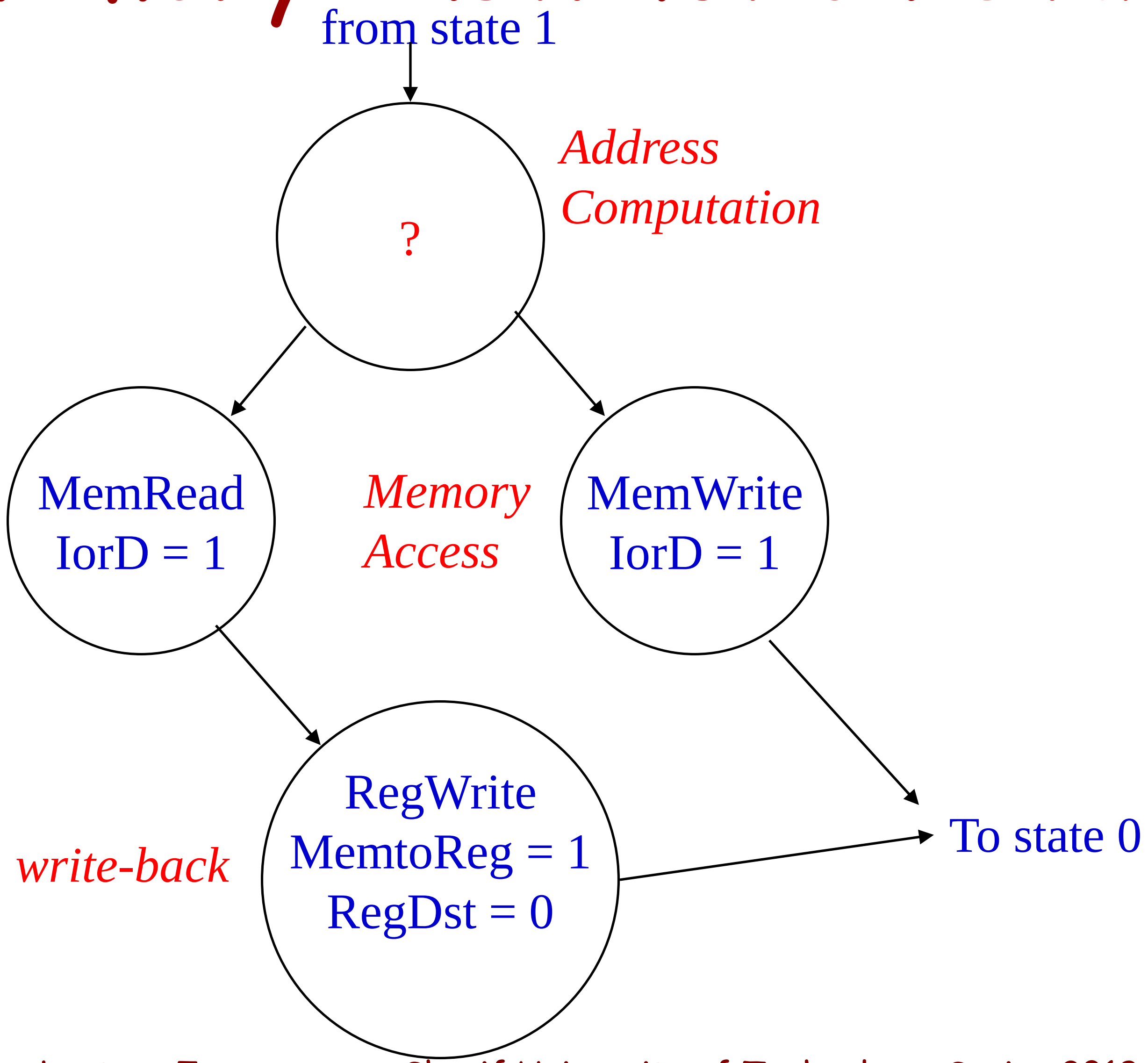
MDR = Memory[ALUout]

Cycle 5 for load: WriteBack



GPR[IR[20-16]] = MDR

Memory Instruction States



from state 1

?

Address Computation

MemRead
IorD = 1

Memory Access

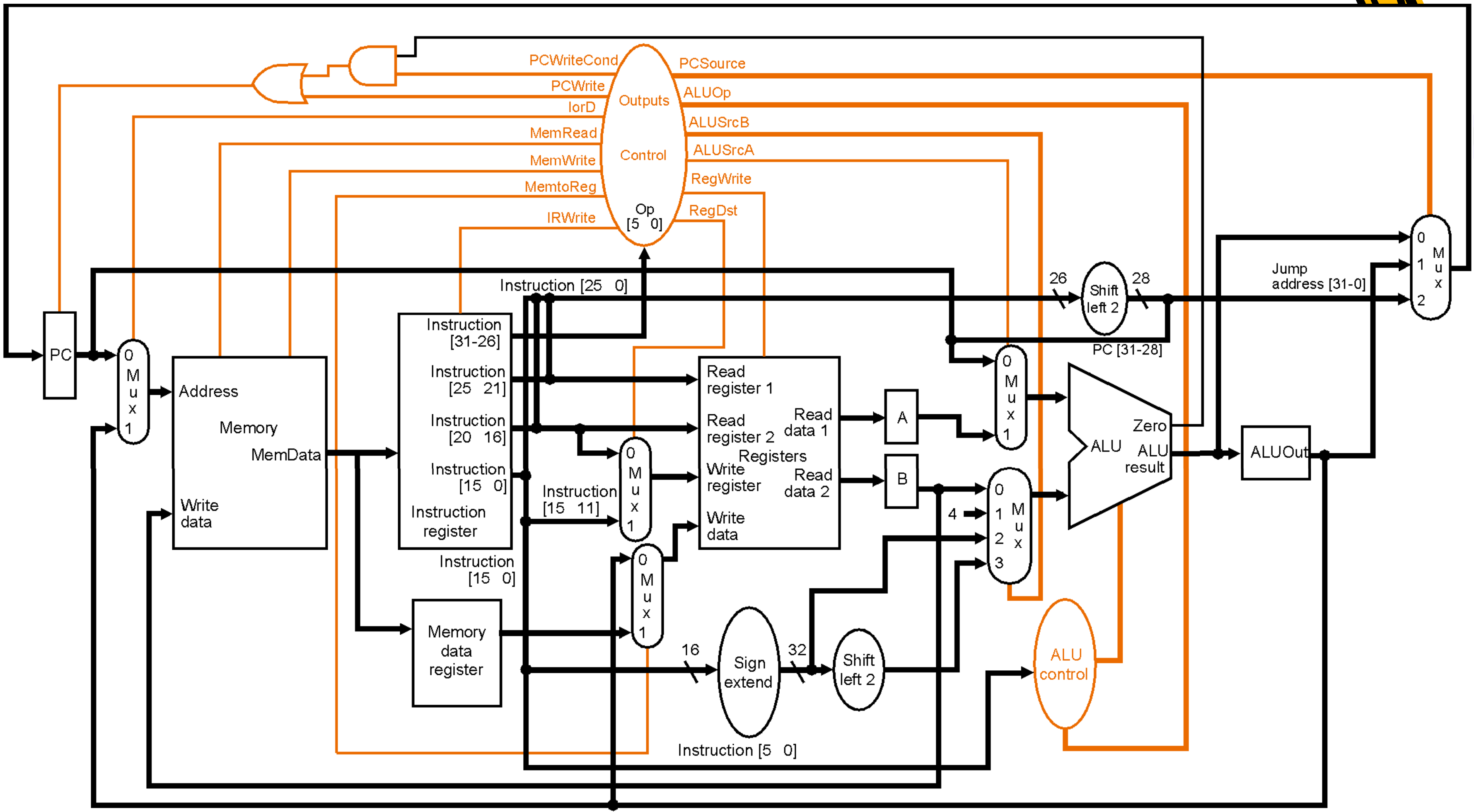
MemWrite
IorD = 1

write-back

RegWrite
MemtoReg = 1
RegDst = 0

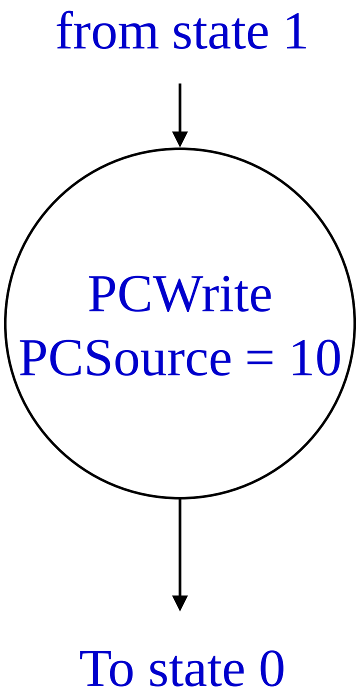
To state 0

Cycle 3 for Jump

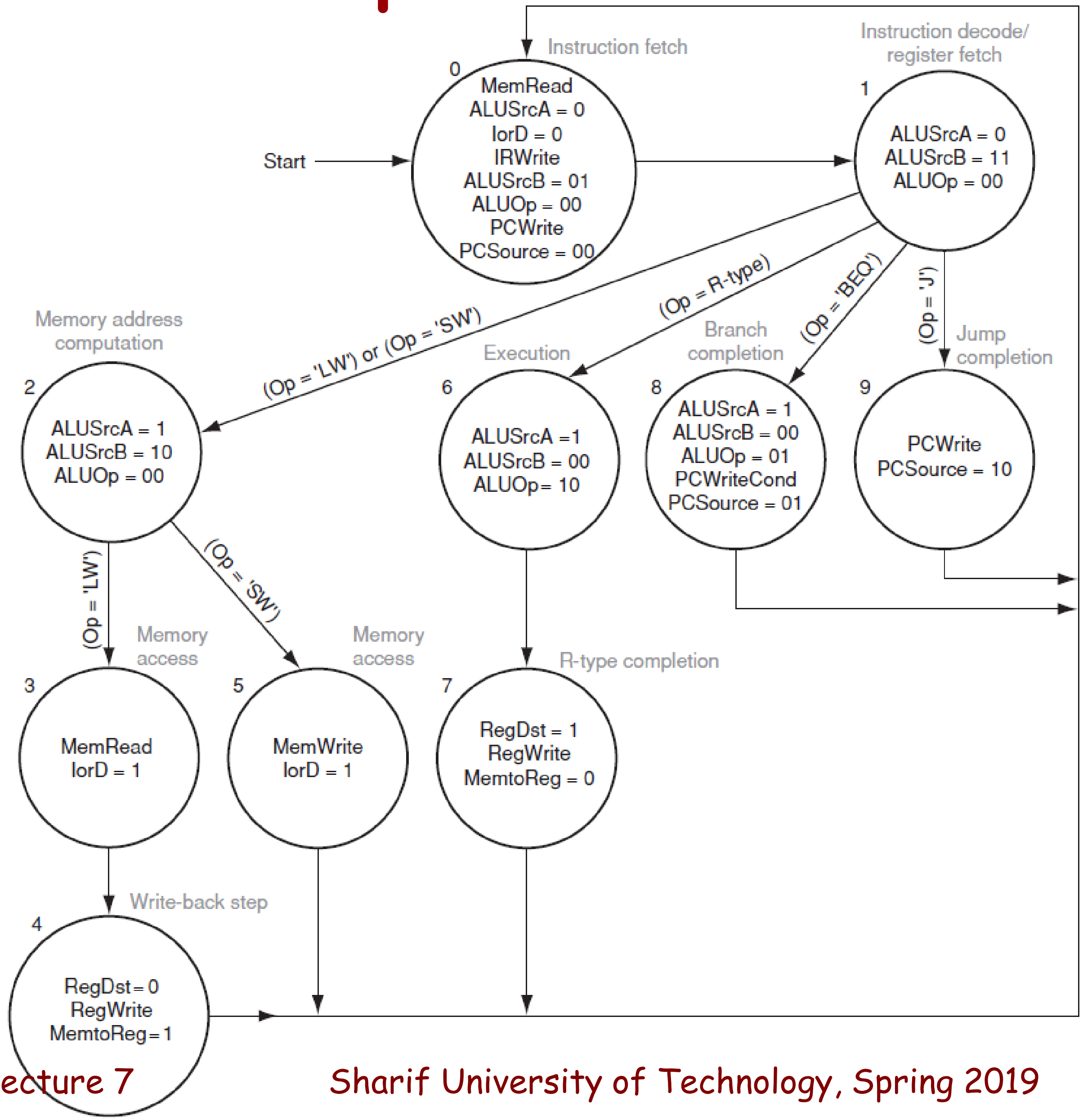


PC = PC[31-28] | (IR[25-0] <<2)

Cycle 3 Jump FSM state

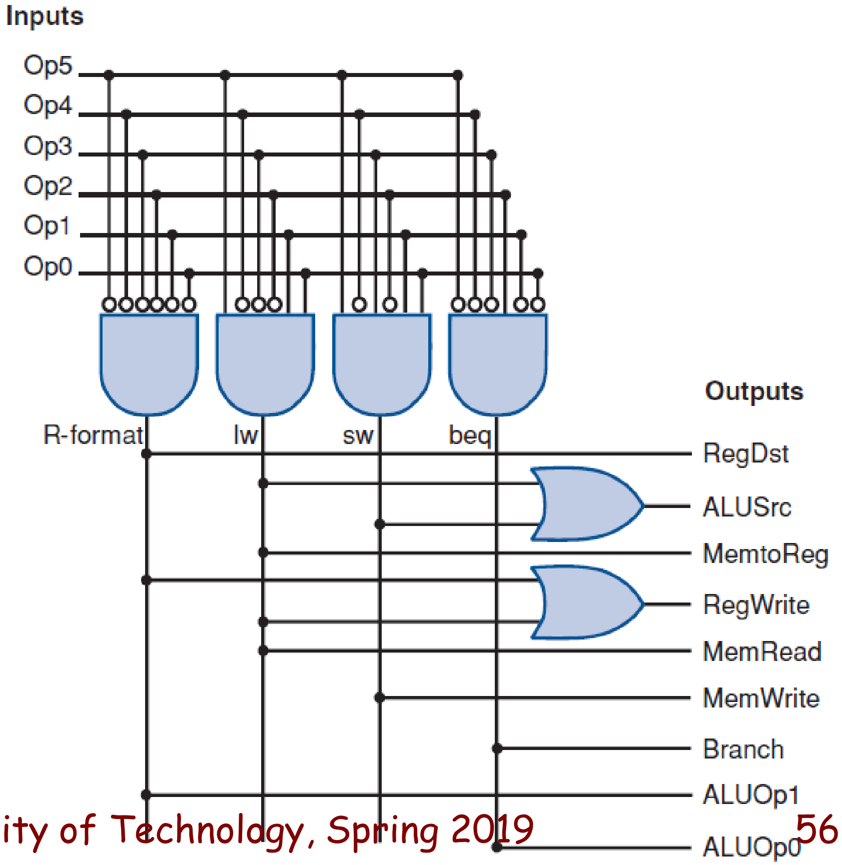


Complete FSM



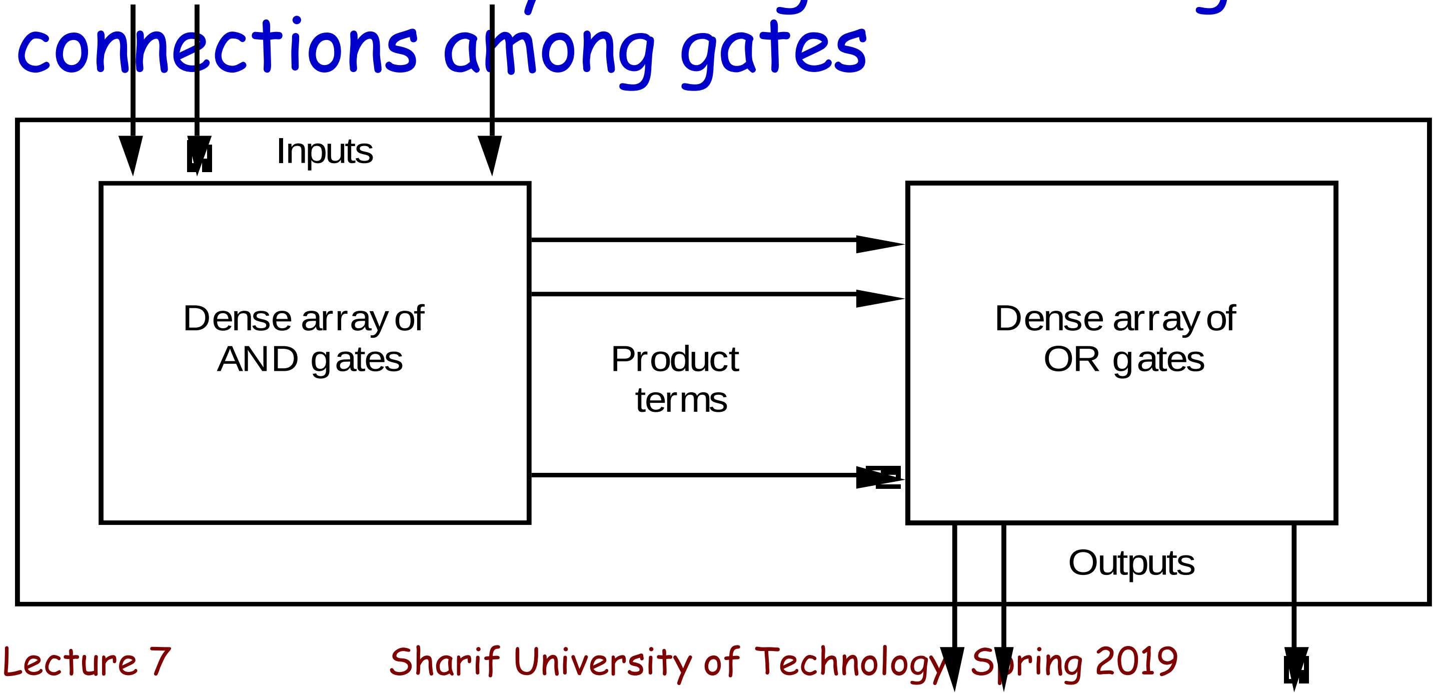
Single-Cycle Control Unit Implementation

- Unstructured LogicDesign
  - By Karnaugh Map
- PLA/PAL



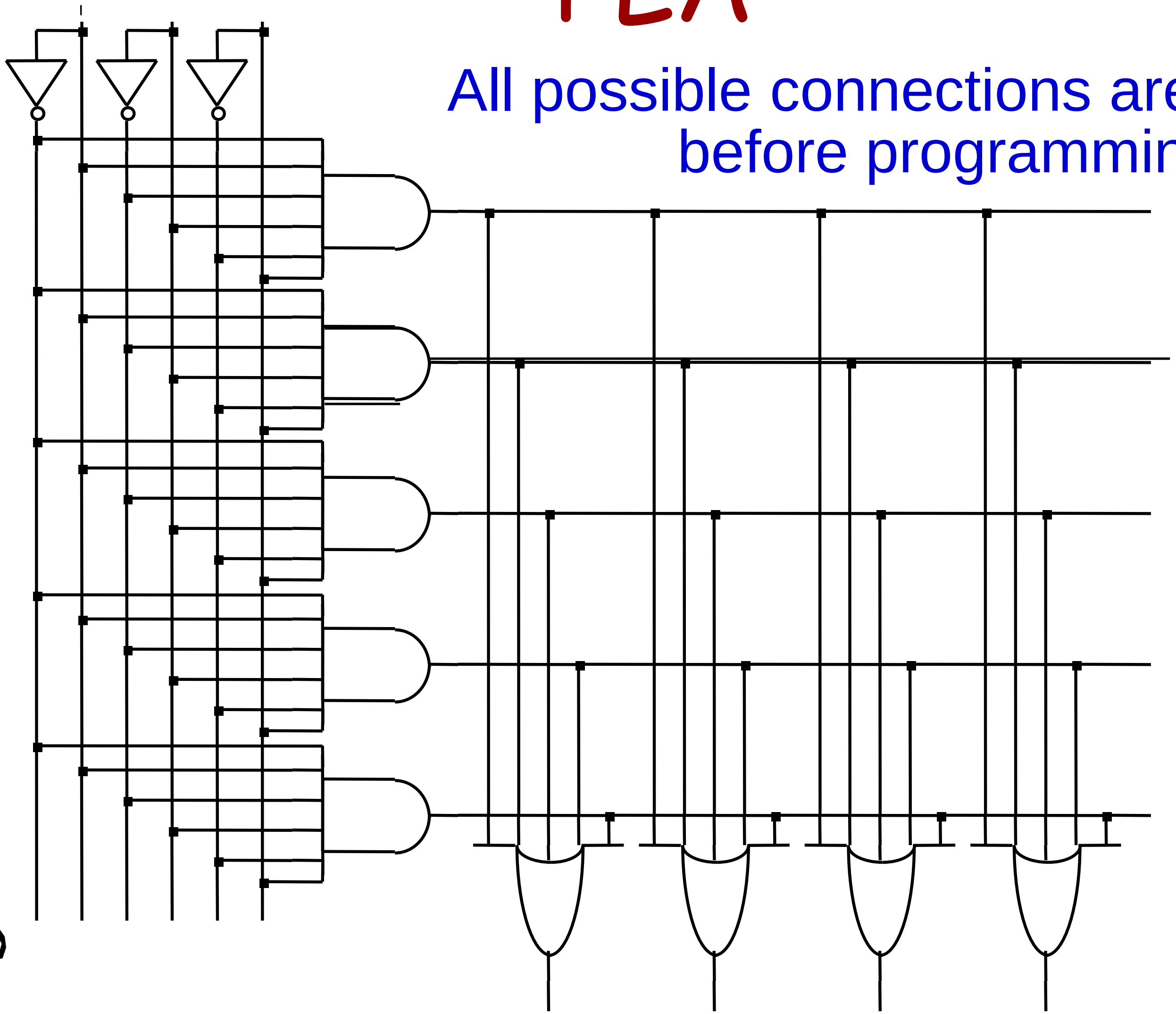
PAL/PLA

- 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

$F1 = A\,B\,C$

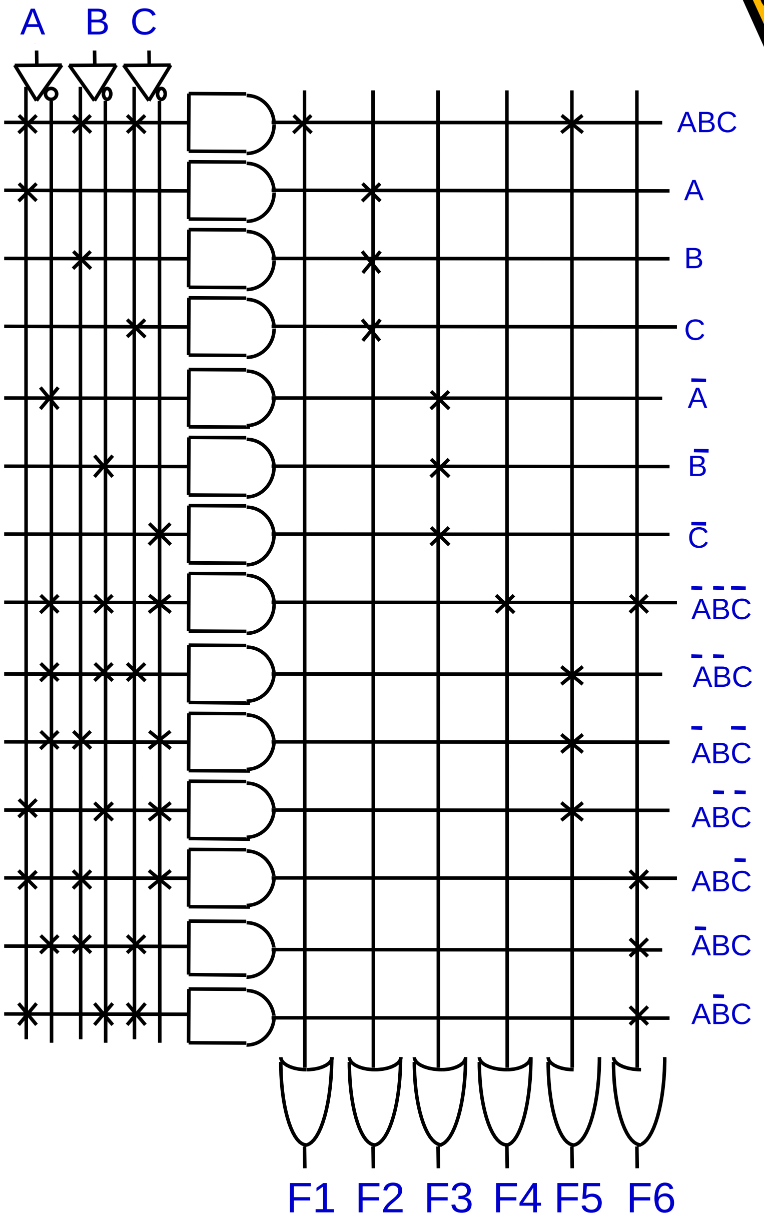
$F2 = A + B + C$

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

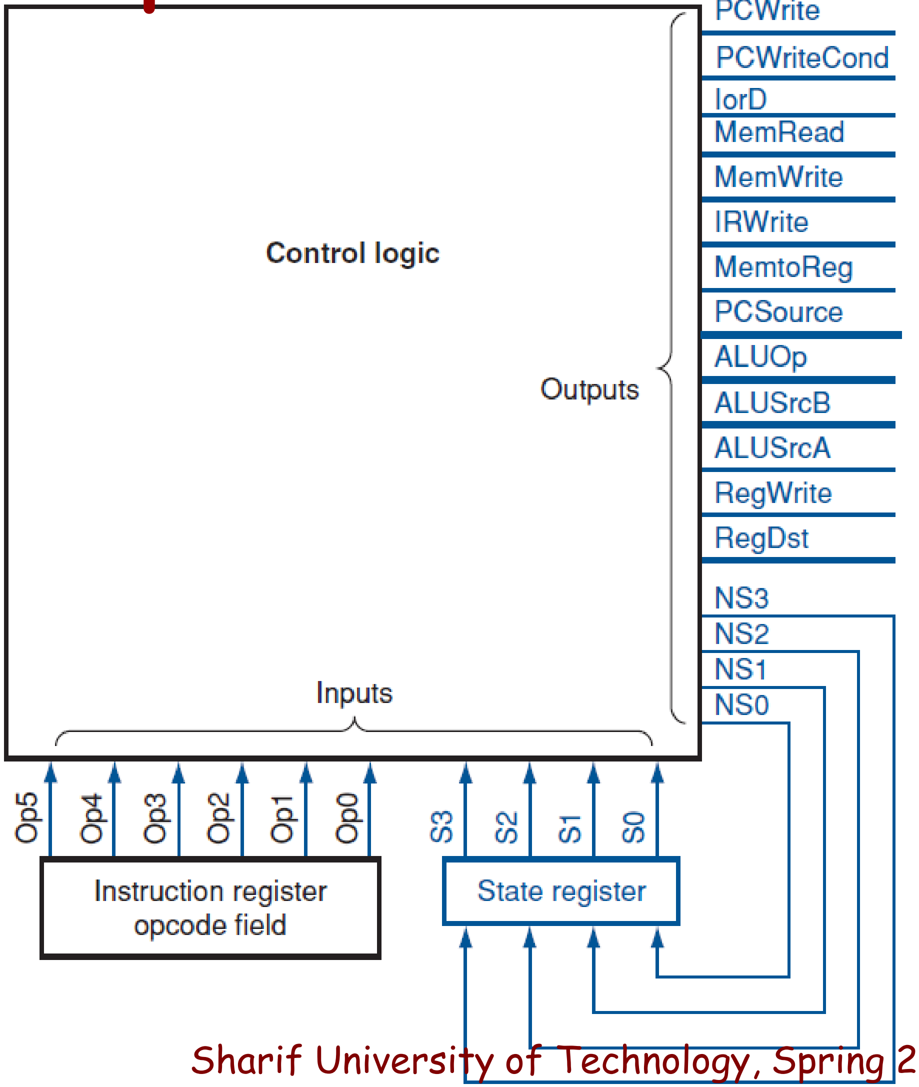
$F4 = \overline{A + B + C}$

F5 = A xor B xor C

F6 = A xnor B xnor C



Multi-Cycle Control Unit Implementation

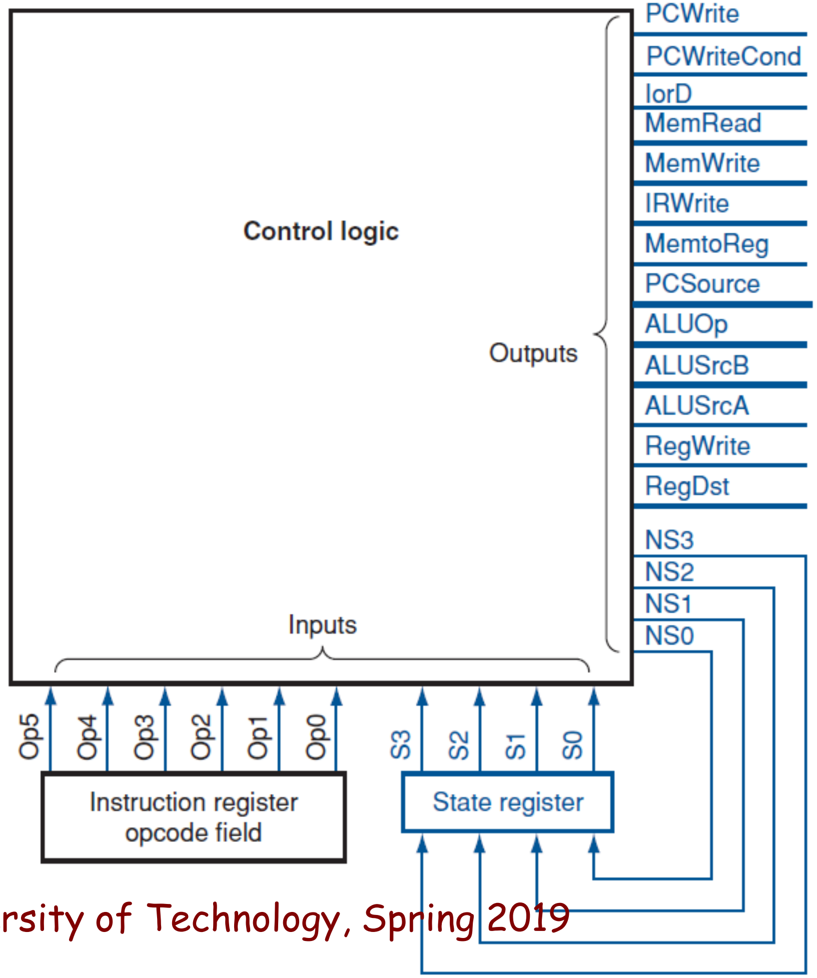


Multi-Cycle Control Unit Implementation (cont.)

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

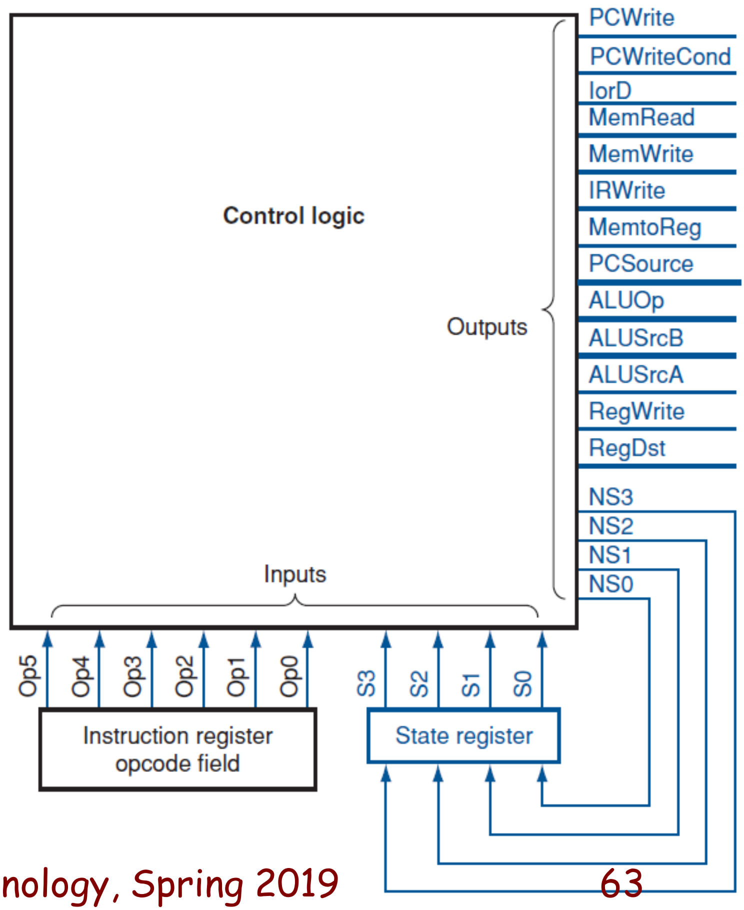
Multi-Cycle Control Unit Implementation (cont.)

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

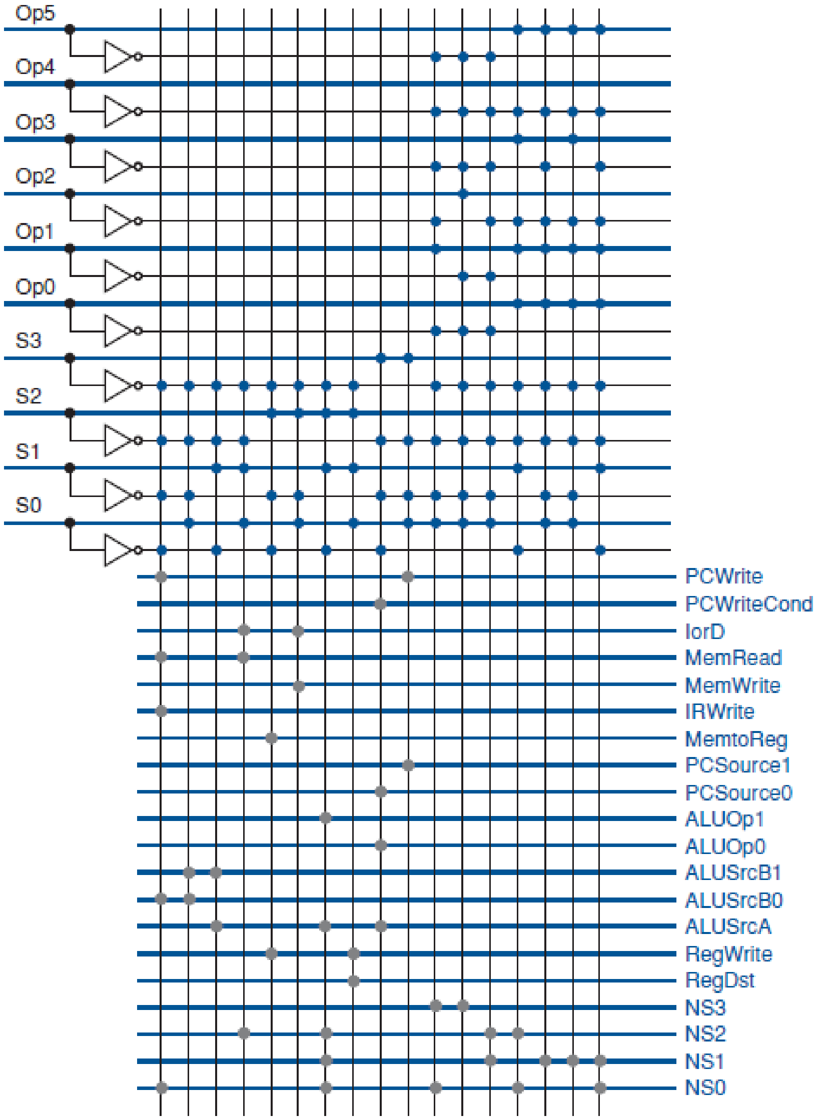


Multi-Cycle Control Unit Implementation (cont.)

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

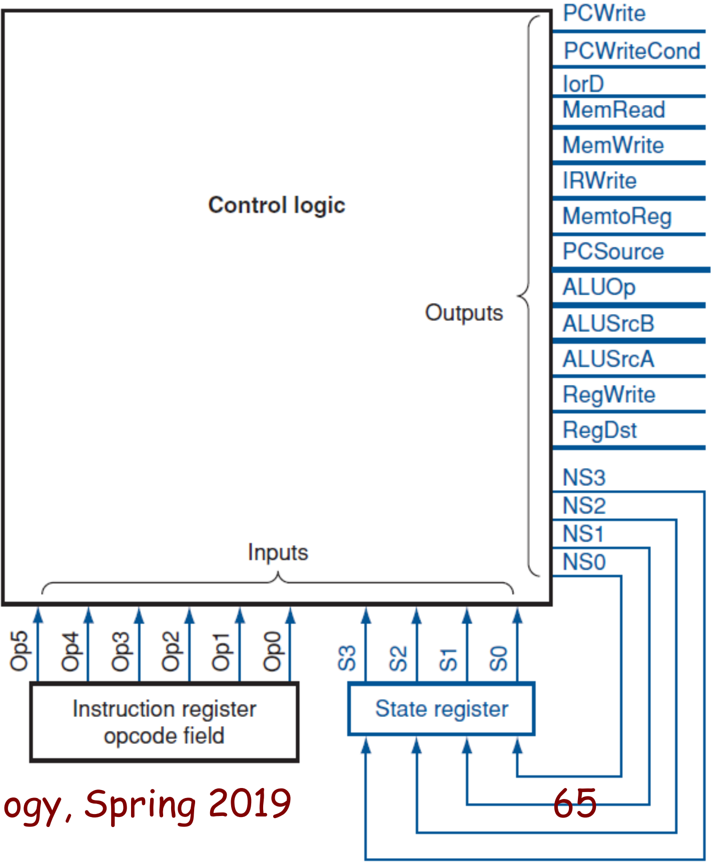


Multi-Cycle Control Unit Implementation in PLA



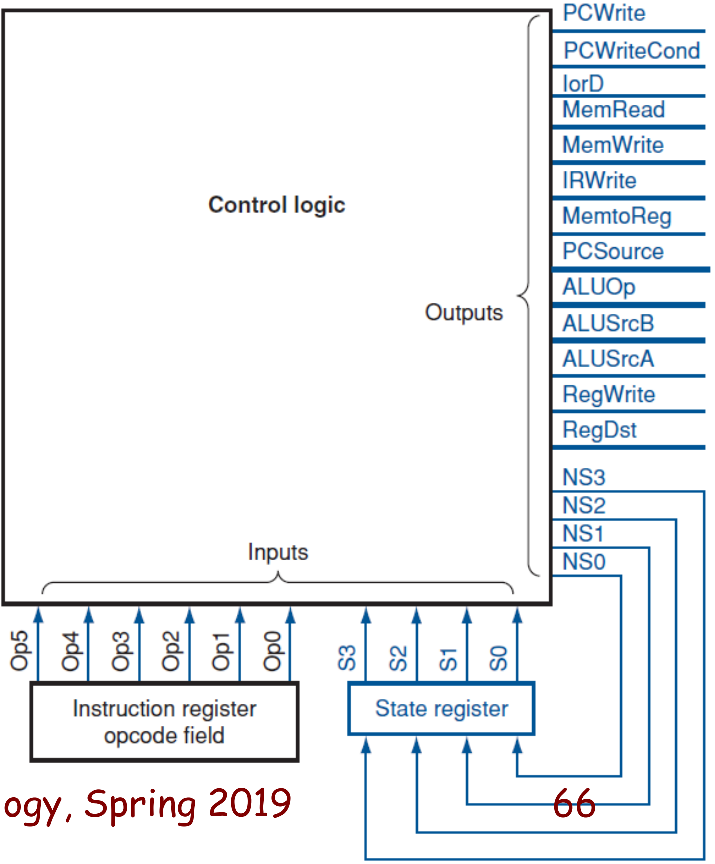
Multi-Cycle Control Unit Implementation in ROM

- 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

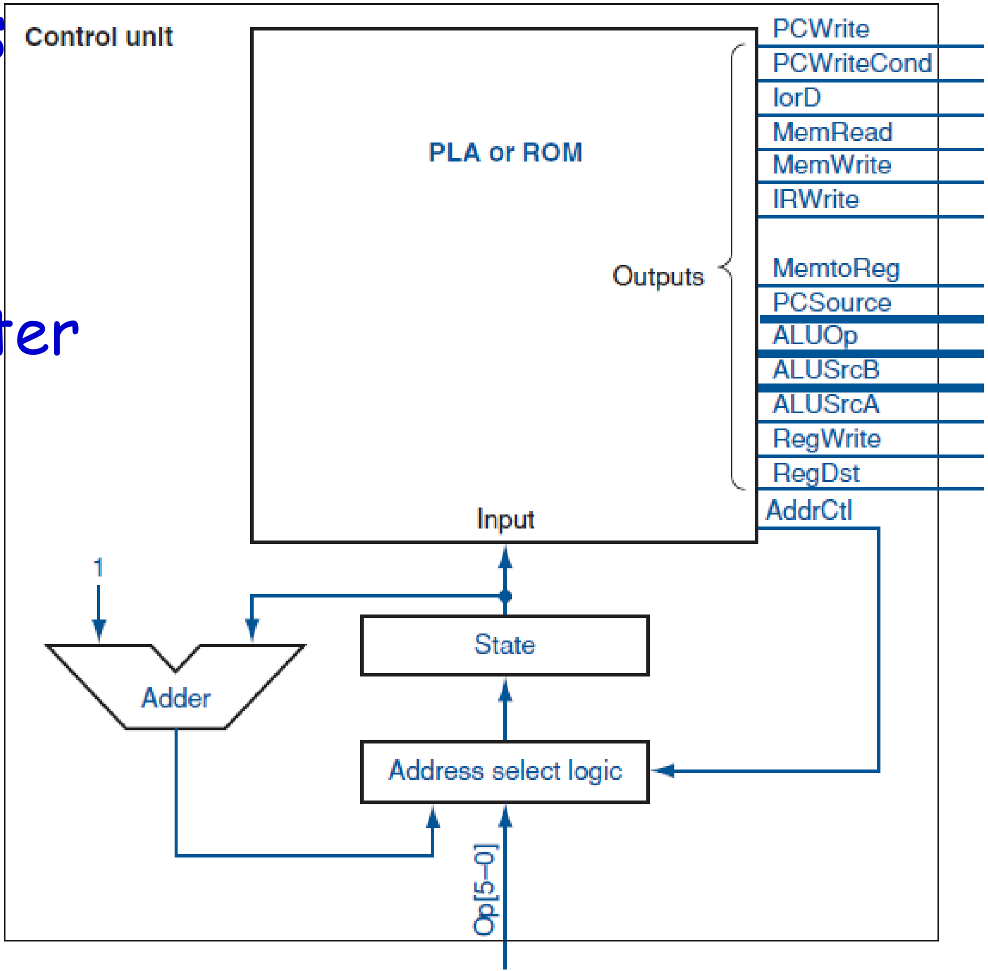


Implementing Multi-Cycle Control Using Micro-Program

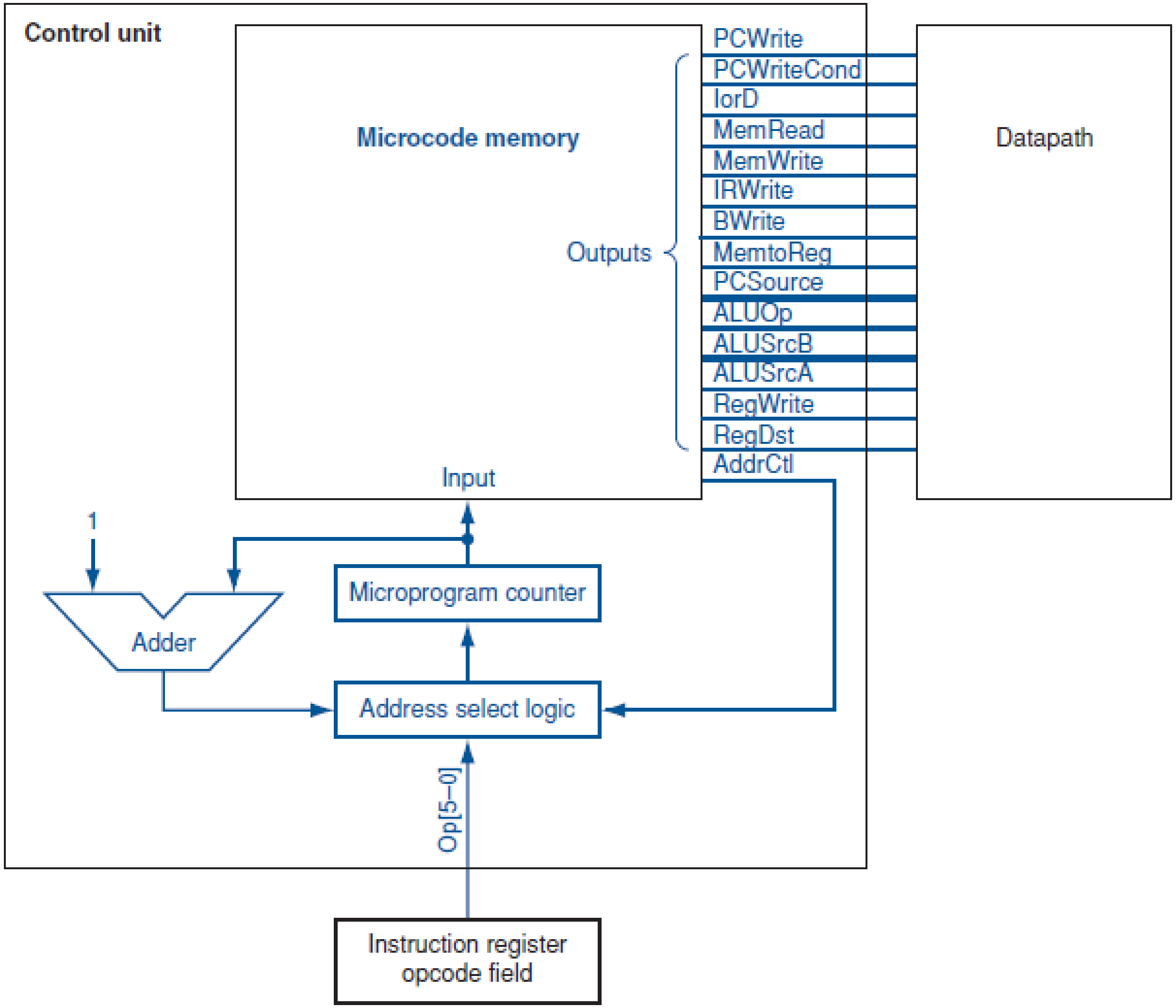
- Cons of ROM Implementation
  - 95% of ROM used to indicate next state
    - 4Kbits
  - What if we have more complex ISA?
    - FP instructions which may take several cycles
- Example:
  - Consider an FSM which requires 10 FFs
    - What would be size of ROM?
- What's Solution?

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

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

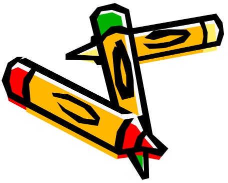


Implementing Multi-Cycle Control Using Micro-Program (cont.)

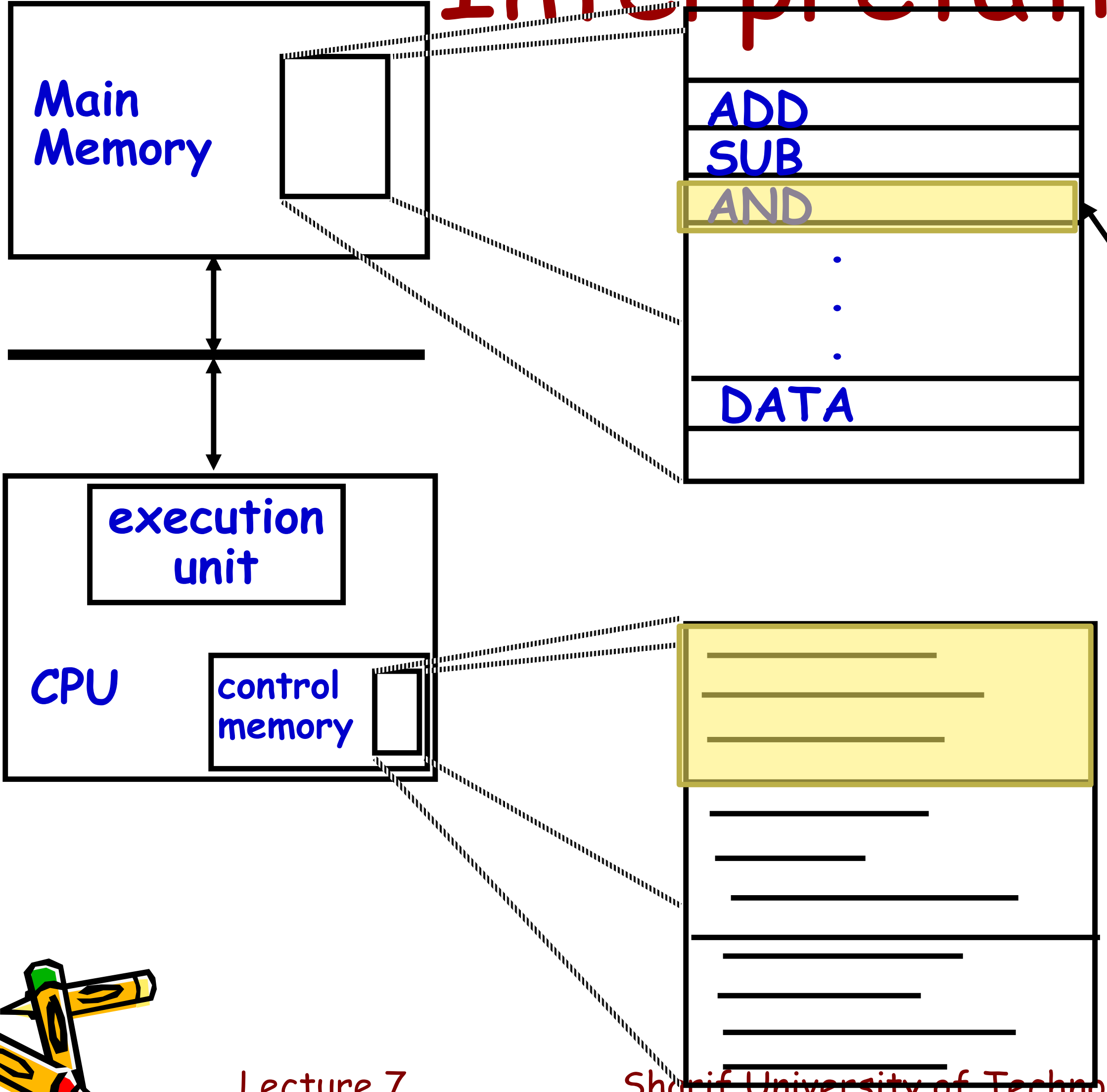


Microprogramming (cont.)

- 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



User program plus Data

this can change!

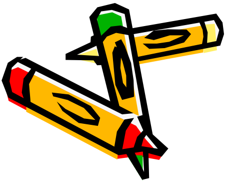
one of these is mapped into one of these

AND microsequence

e.g., Fetch
Calc Operand Addr
Fetch Operand(s)
Calculate
Save Answer(s)

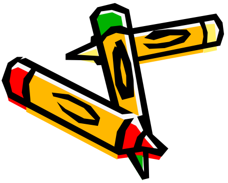
Microprogramming (cont.)

- 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



Backup