

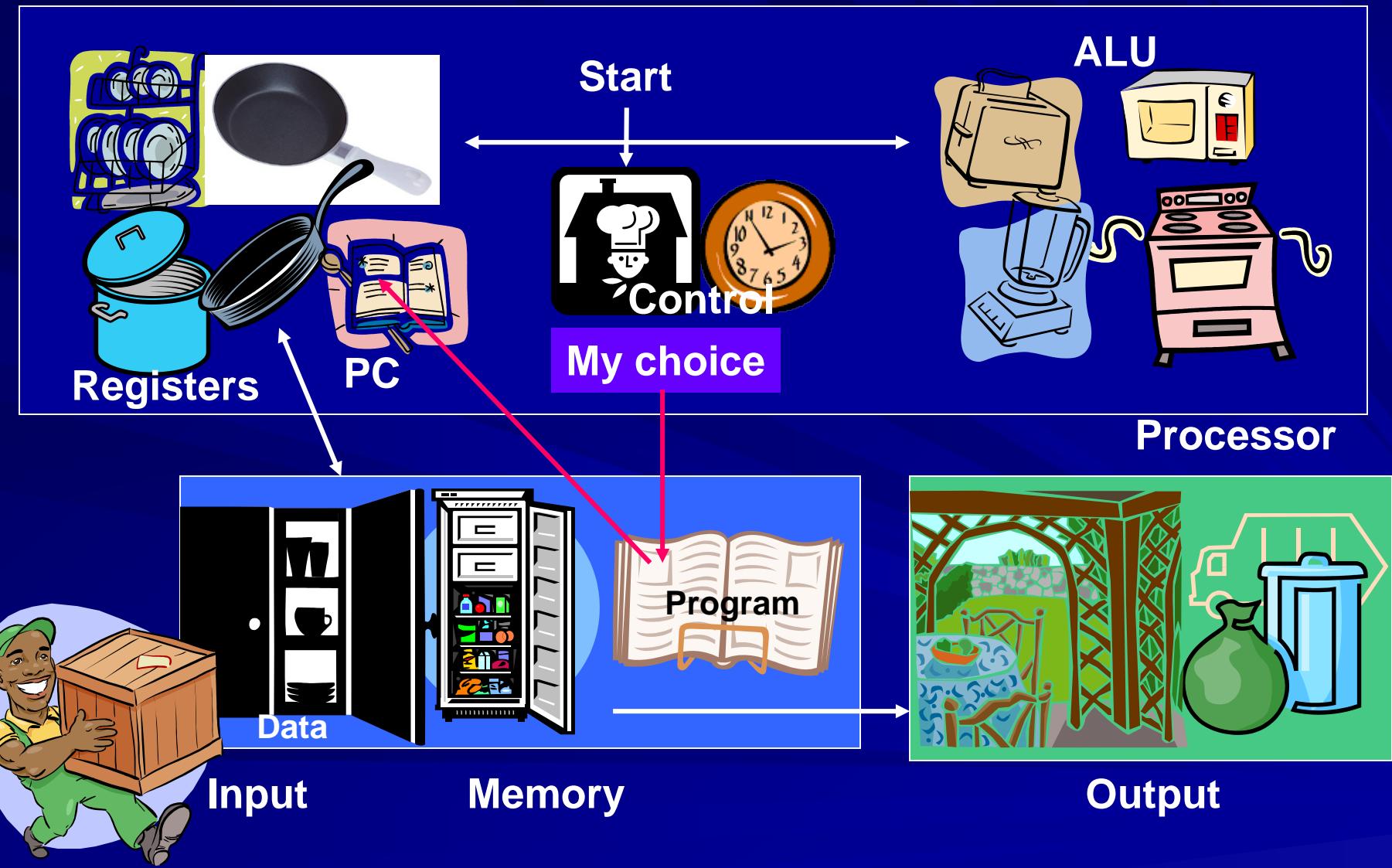
ELEC 5200-001/6200-001
Computer Architecture and Design
Spring 2019

Datapath and Control (Chapter 4)

Christopher B. Harris
Assistant Professor
Department of Electrical and Computer
Engineering
Auburn University, Auburn, AL 36849

(Adapted from slides by Vishwani D. Agrawal)

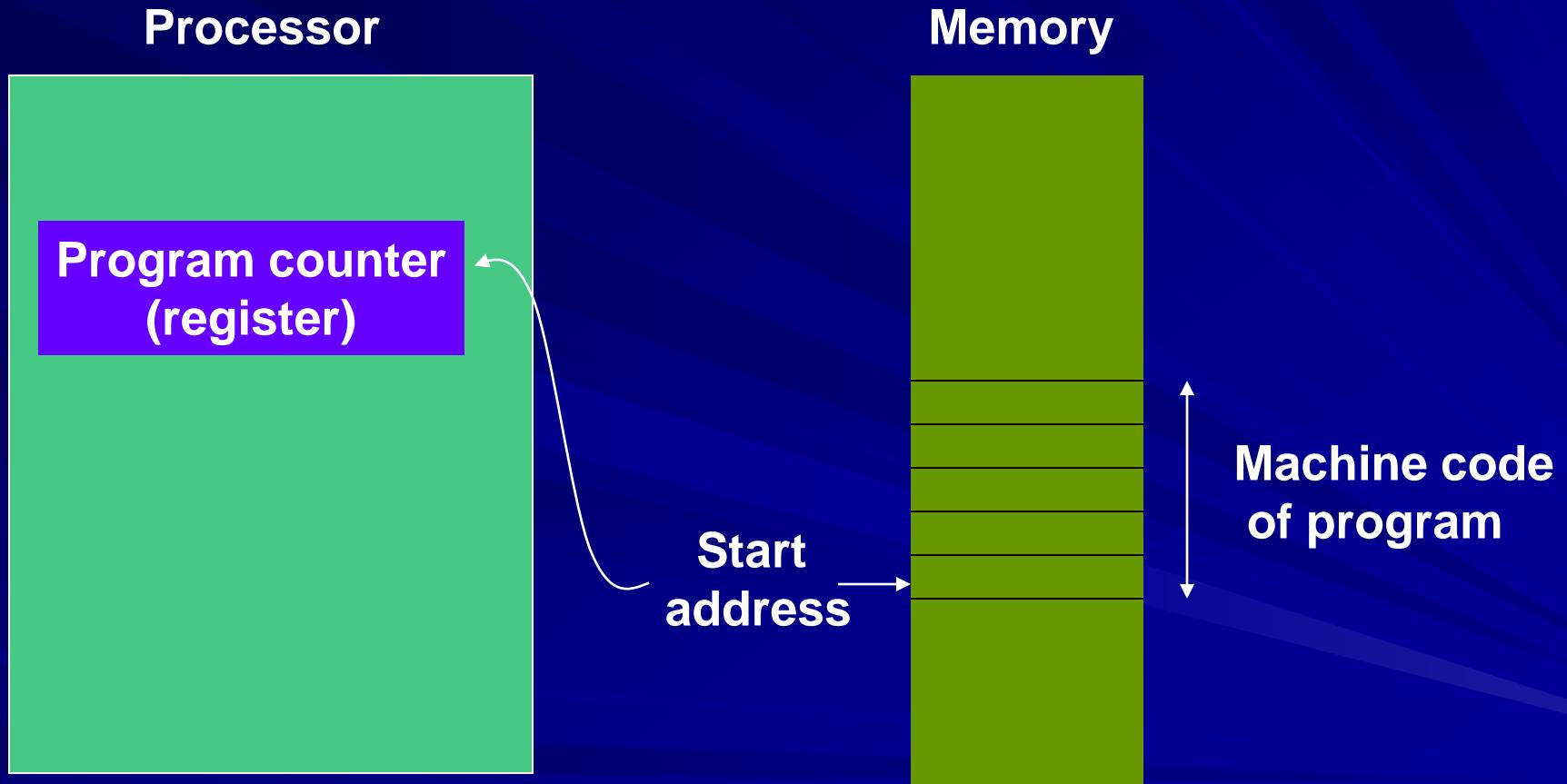
Von Neumann Kitchen



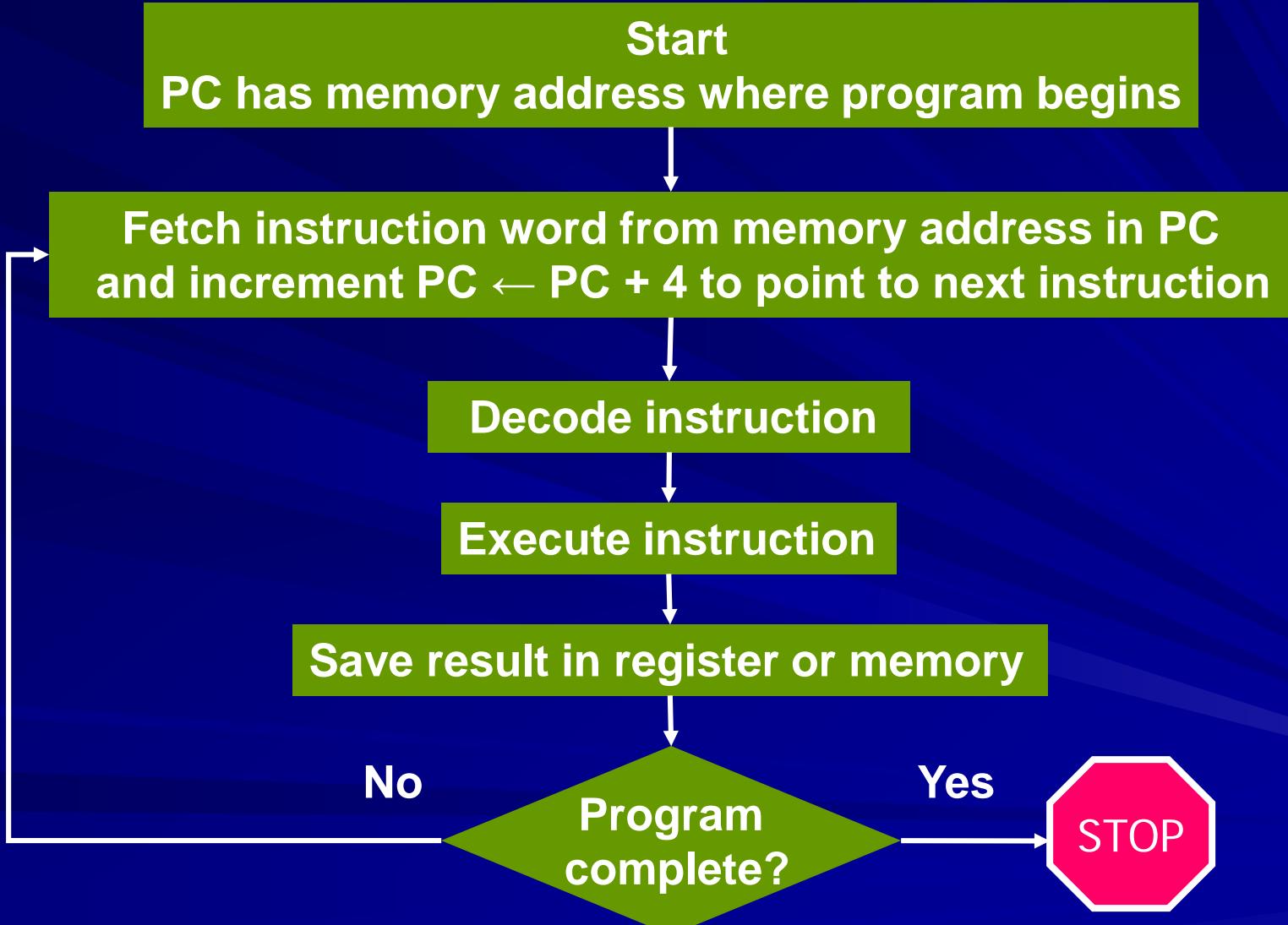
Where Does It All Begin?

- In a register called *program counter (PC)*.
- PC contains the memory address of the next instruction to be executed.
- In the beginning, PC contains the address of the memory location where the program begins.

Where is the Program?



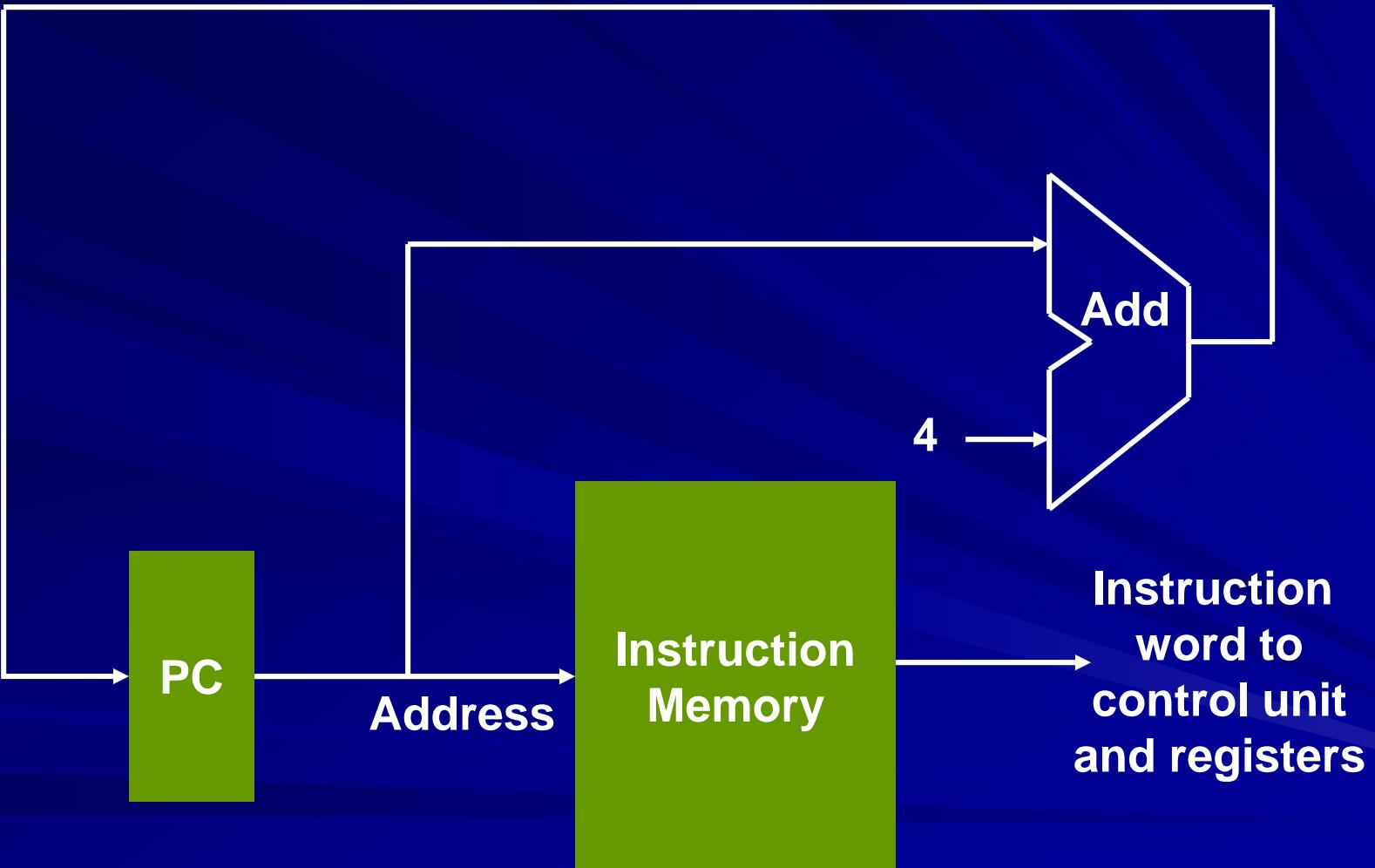
How Does It Run?



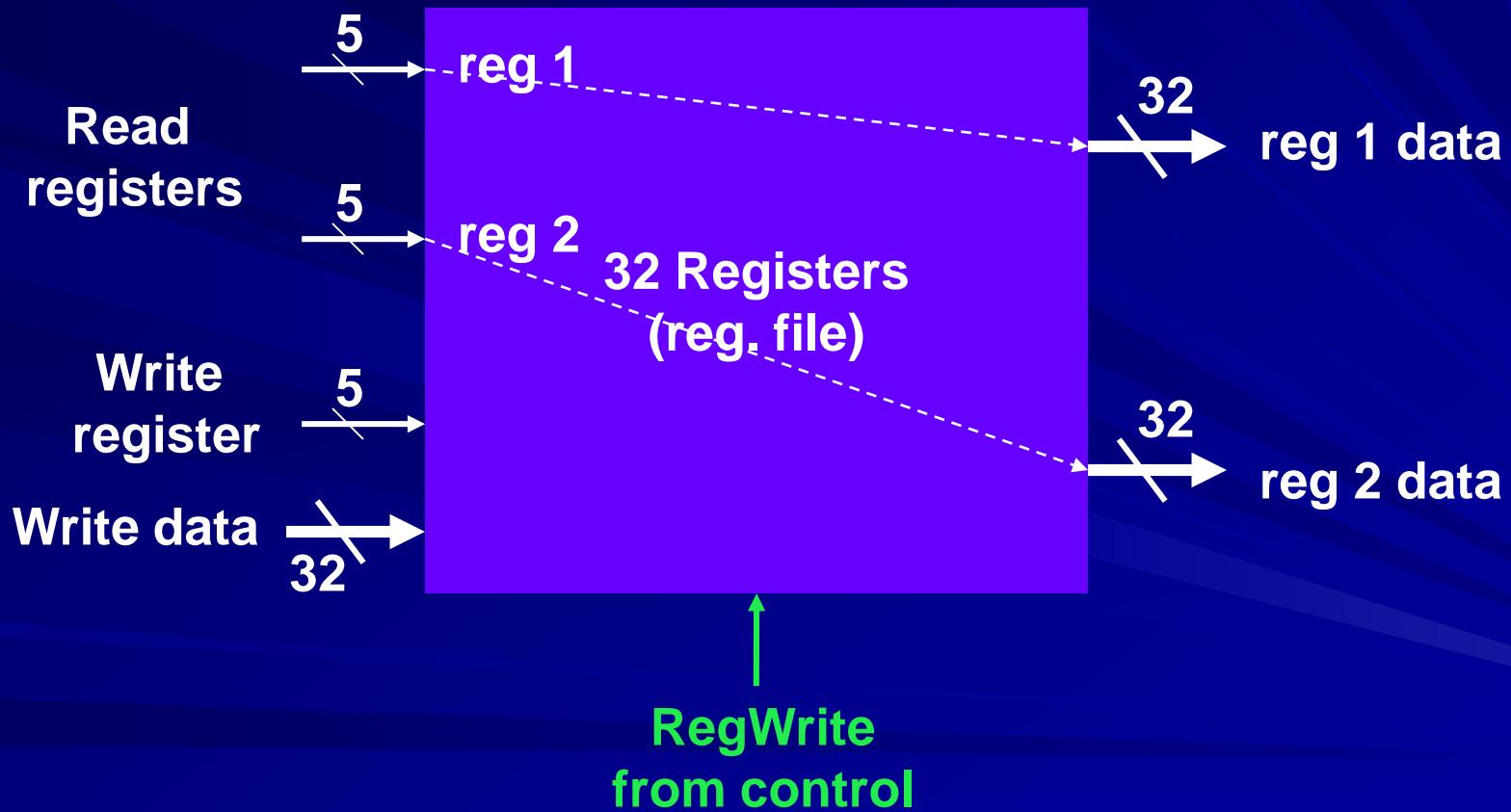
Datapath and Control

- Datapath: Memory, registers, adders, ALU, and communication buses. Each step (fetch, decode, execute, save result) requires communication (data transfer) paths between memory, registers and ALU.
- Control: Datapath for each step is set up by control signals that set up dataflow directions on communication buses and select ALU and memory functions. Control signals are generated by a control unit consisting of one or more finite-state machines.

Datapath for Instruction Fetch

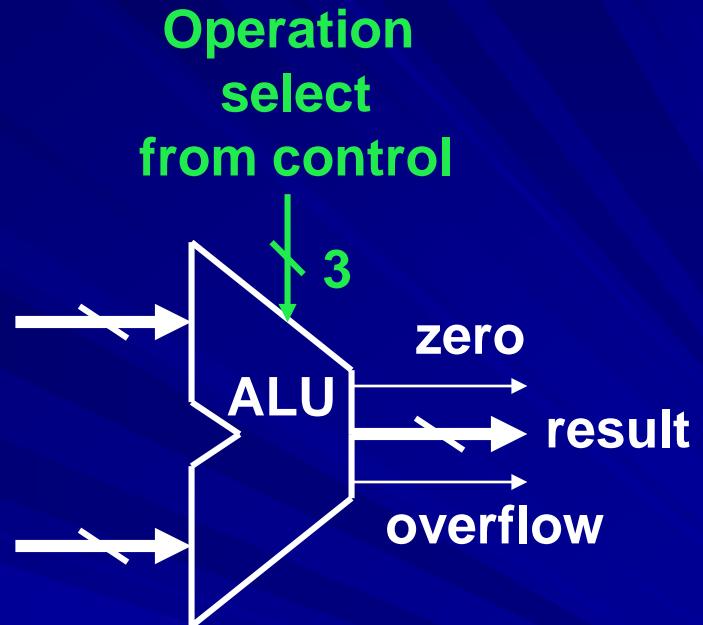


Register File: A Datapath Component



Multi-Operation ALU

Operation select	ALU function
000	AND
001	OR
010	Add
110	Subtract
111	Set on less than



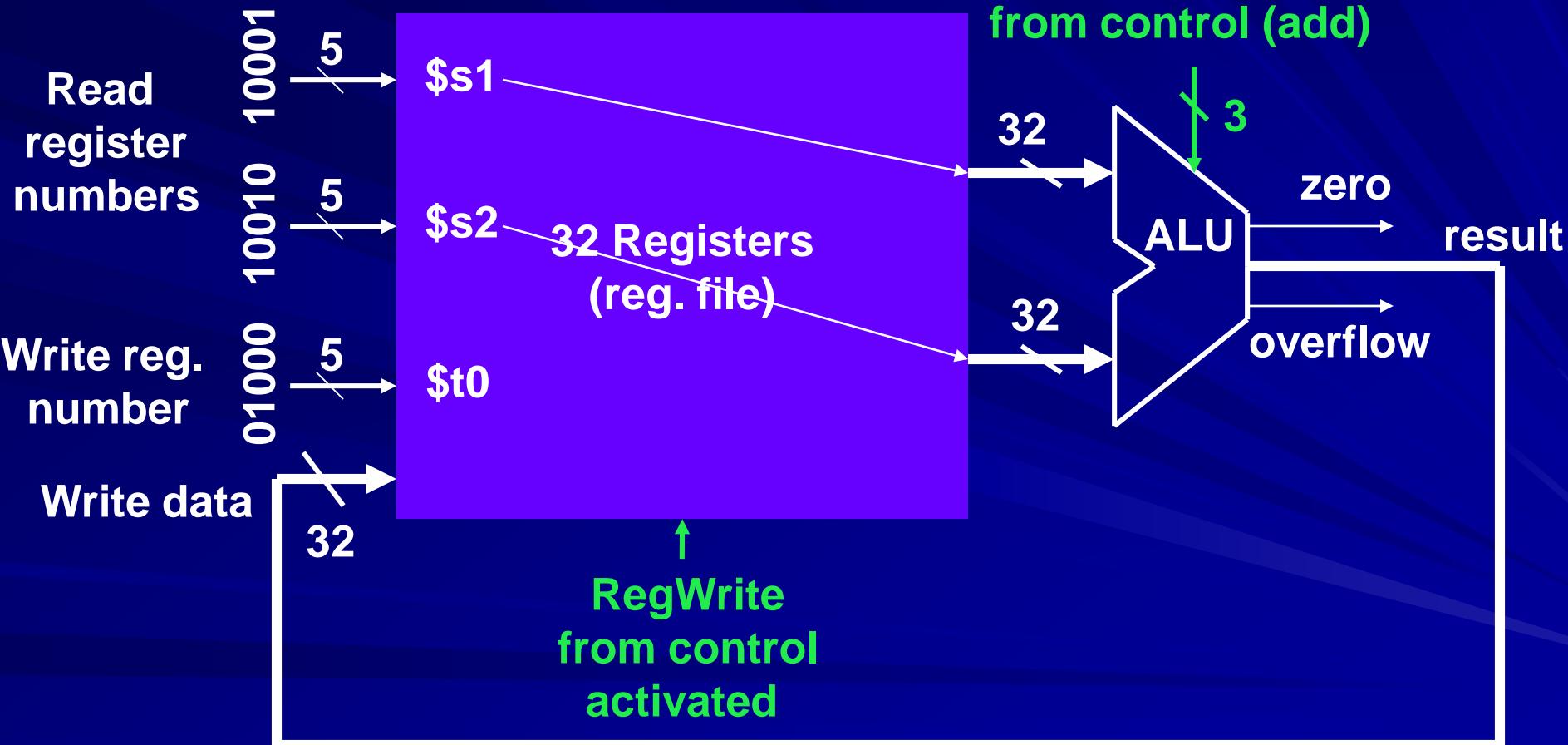
zero = 1, when all bits of result are 0

R-Type Instructions

- Also known as arithmetic-logical instructions
- add, sub, slt
- Example: add \$t0, \$s1, \$s2
 - Machine instruction word
000000 10001 10010 01000 00000 100000
opcode \$s1 \$s2 \$t0 function
 - Read two registers
 - Write one register
 - Opcode and function code go to control unit that generates RegWrite and ALU operation code.

Datapath for R-Type Instruction

000000 10001 10010 01000 00000 100000
opcode \$s1 \$s2 \$t0 function (add)



Load and Store Instructions

■ I-type instructions

■ lw \$t0, 1200 (\$t1) # incr. in bytes

100011 01001 01000 0000 0100 1011 0000

opcode \$t1 \$t0 1200

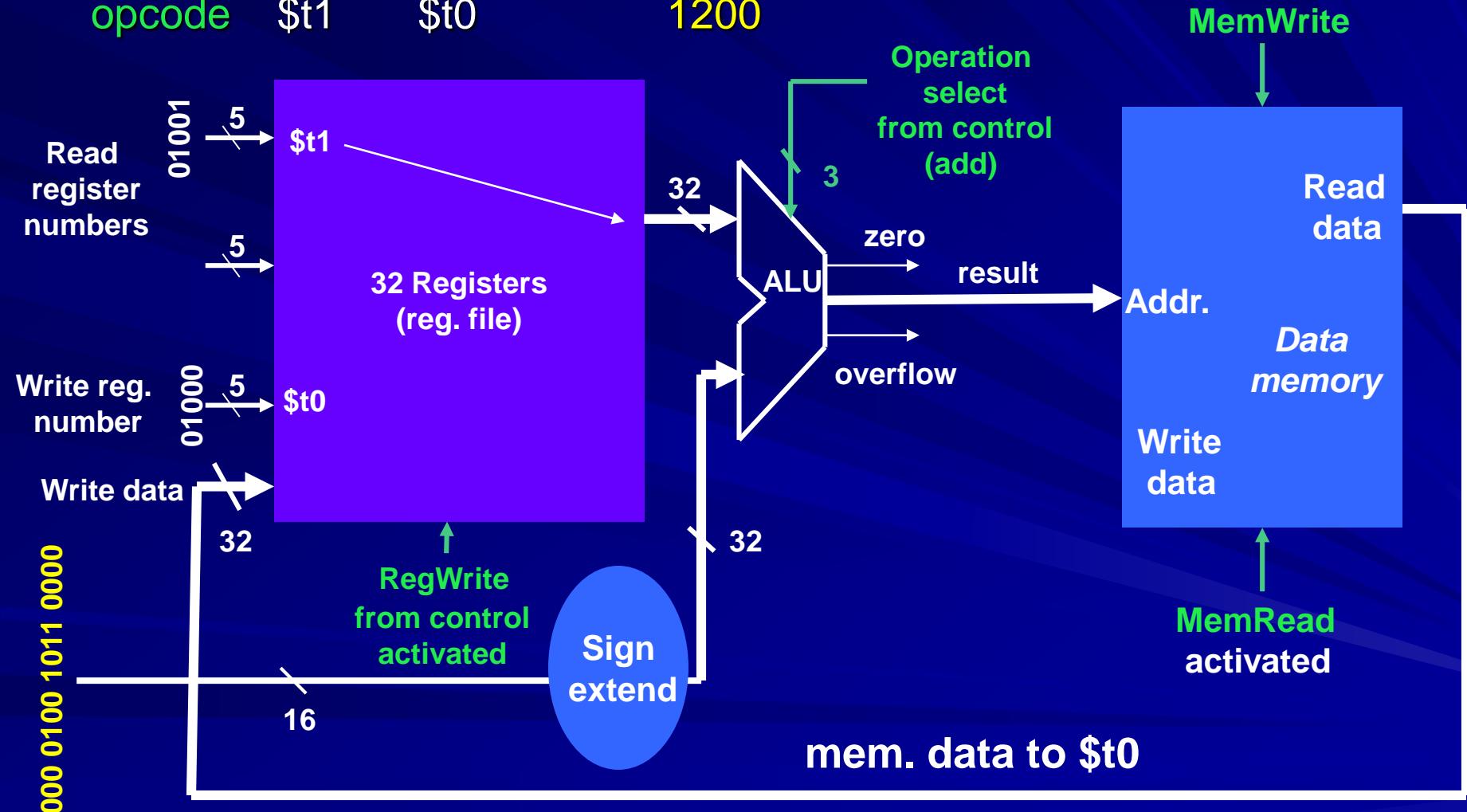
■ sw \$t0, 1200 (\$t1) # incr. in bytes

101011 01001 01000 0000 0100 1011 0000

opcode \$t1 \$t0 1200

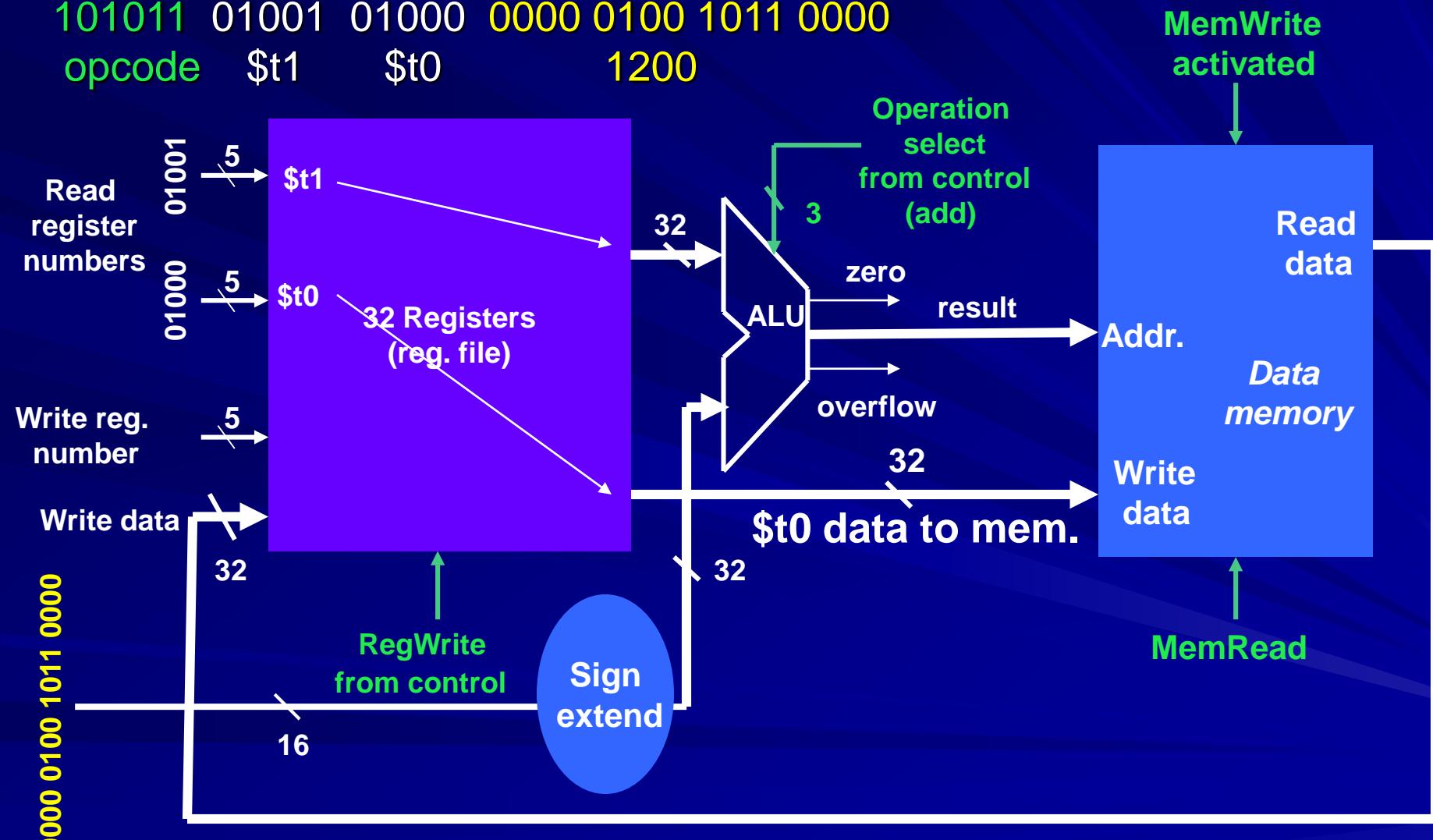
Datapath for lw Instruction

100011 01001 01000 0000 0100 1011 0000
 opcode \$t1 \$t0 1200



Datapath for sw Instruction

101011 01001 01000 0000 0100 1011 0000
 opcode \$t1 \$t0 1200



Branch Instruction (I-Type)

- beq \$s1, \$s2, 25 # if \$s1 = \$s2,
advance PC through
25 instructions

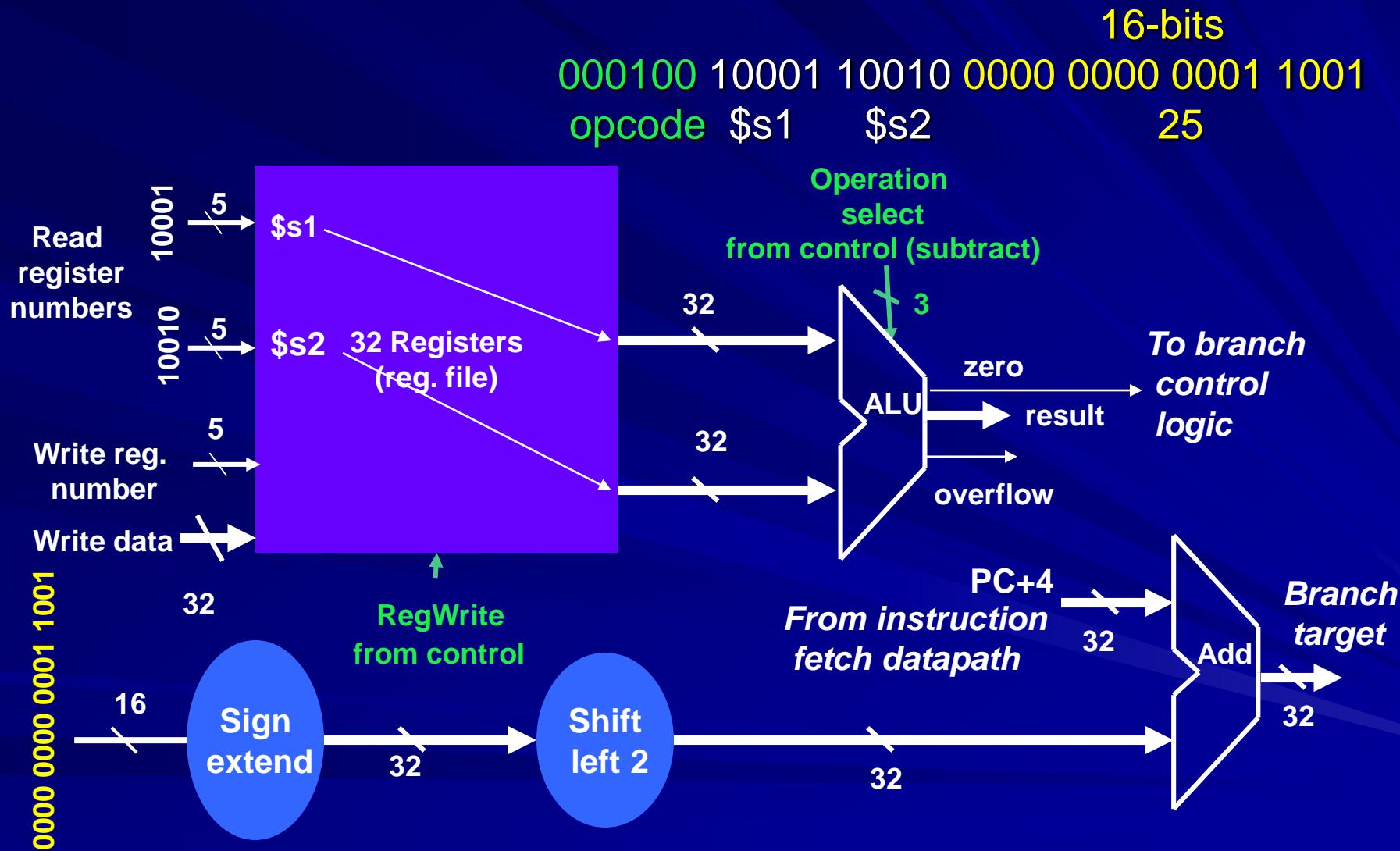
000100 10001 10010 0000 0000 0001 1001
opcode \$s1 \$s2 25



A horizontal double-headed arrow spanning the width of the 16-bit branch offset field, which is the 5-bit field starting from the 11th bit of the instruction word.

Note: Can branch within $\pm 2^{15}$ words from the current instruction address in PC.

Datapath for beq Instruction

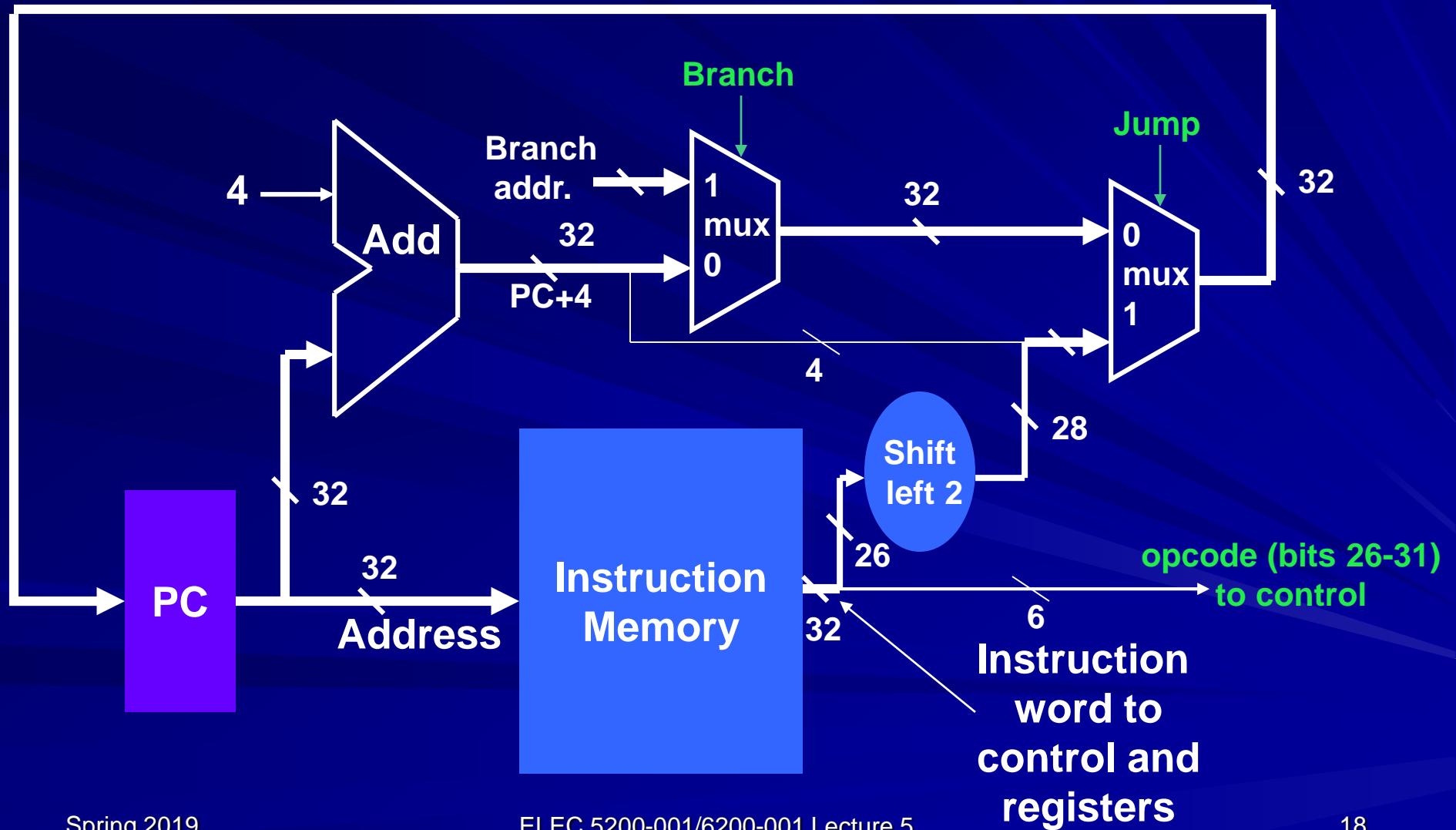


J-Type Instruction

■ j 2500 # jump to instruction 2,500

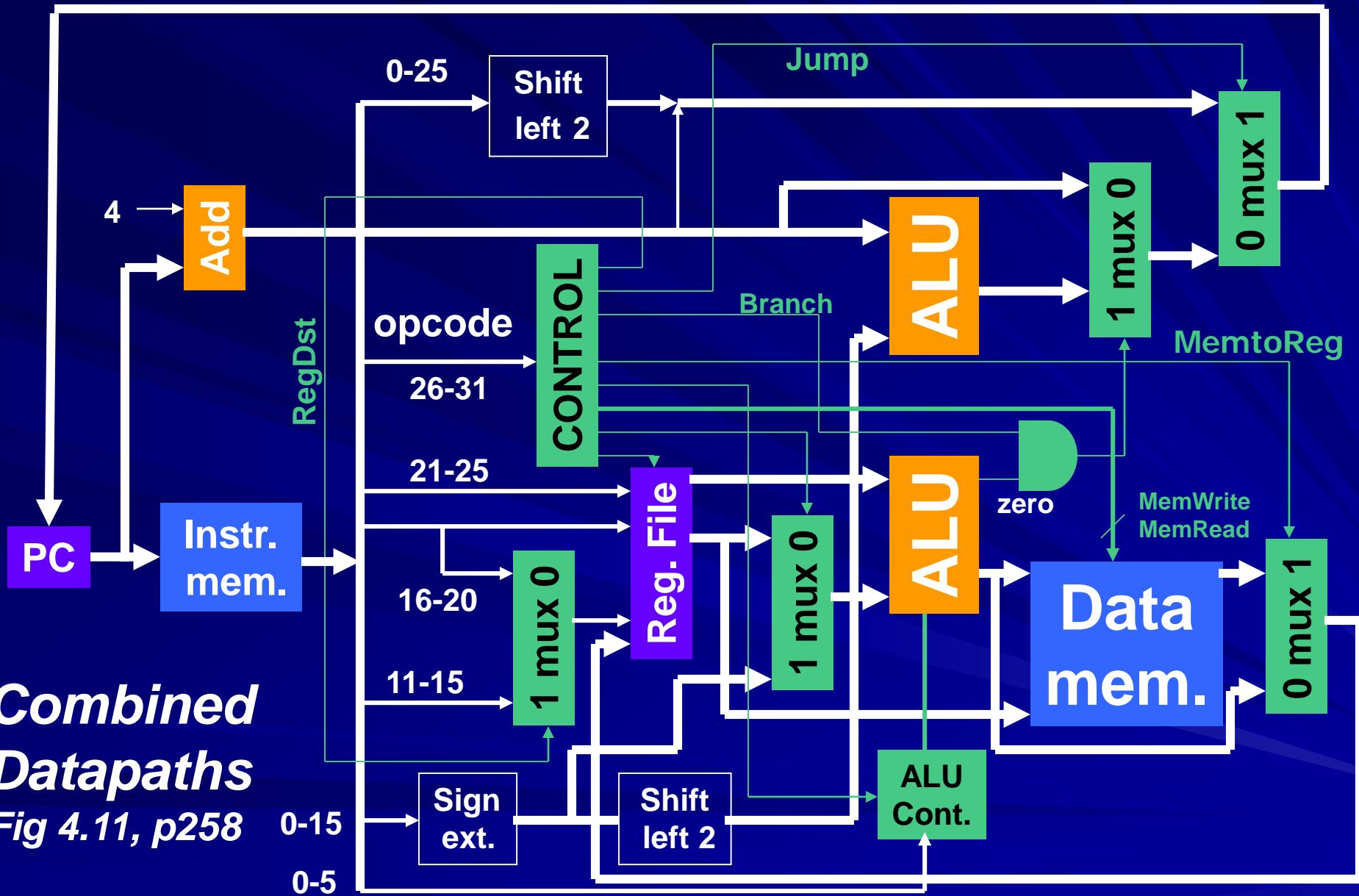


Datapath for Jump Instruction

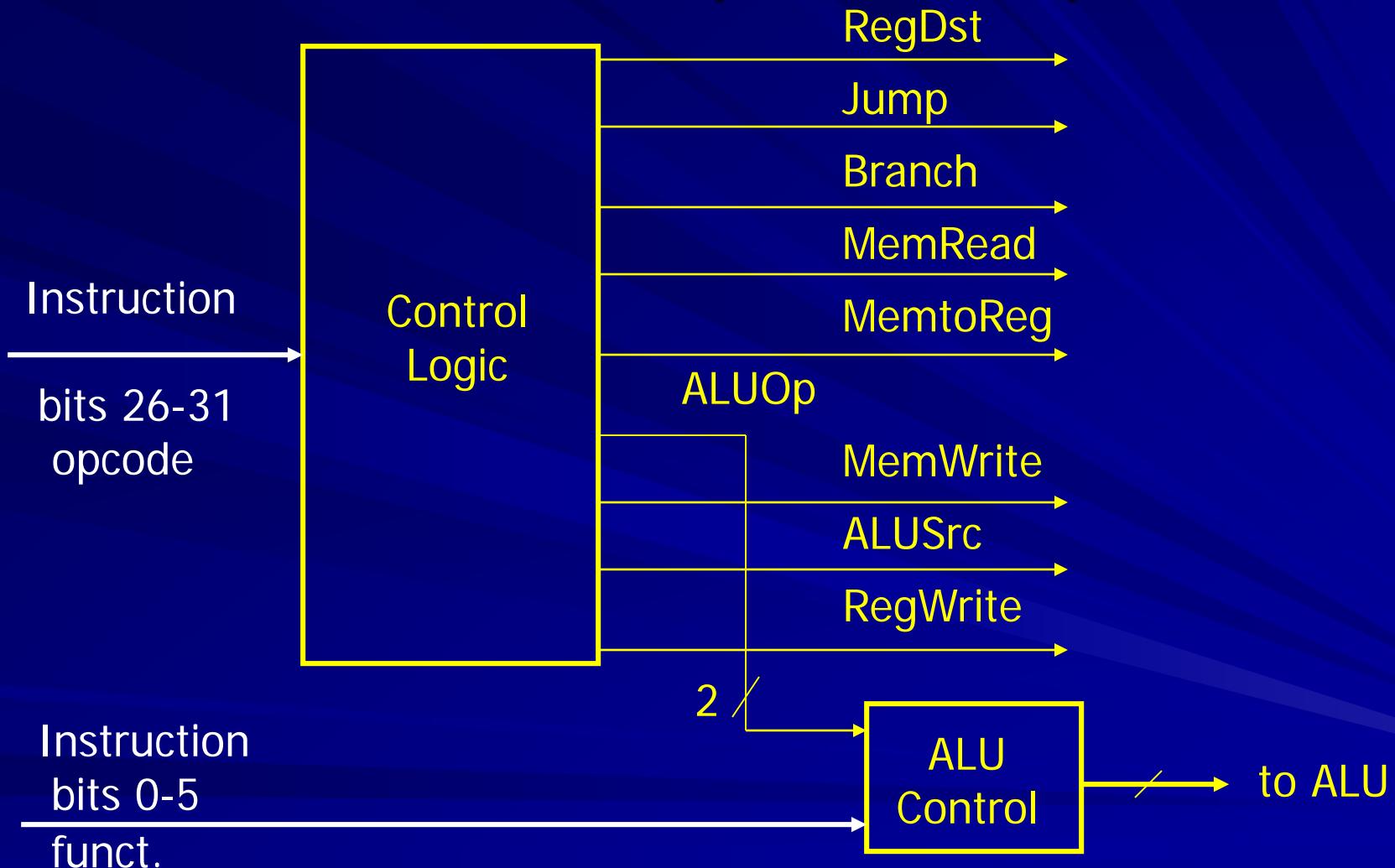


Combined Datapaths

Fig 4.11, p258



Control (Sec 4.4)



Control Logic: Truth Table

Instr type	Inputs: instr. opcode bits						Outputs: control signals								
	31	30	29	28	27	26	RegDst	Jump	ALUSrc	MemtoReg	RegWrite	MemRead	Branch	ALUOp2	ALUOp1
R	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0
Iw	1	0	0	0	1	1	0	0	1	1	1	1	0	0	0
sw	1	0	1	0	1	1	X	0	1	X	0	0	1	0	0
beq	0	0	0	1	0	0	X	0	0	X	0	0	1	0	1
j	0	0	0	0	1	0	X	1	X	X	0	X	0	X	X

How Long Does It Take?

- Assume control logic is fast and does not affect the critical timing. Major time delay components are ALU, memory read/write, and register read/write.
- Arithmetic-type (R-type)
 - Fetch (memory read) 2ns
 - Register read 1ns
 - ALU operation 2ns
 - Register write 1ns
 - Total 6ns

Time for lw and sw (I-Types)

■ ALU (R-type)	6ns
■ Load word (I-type)	
– Fetch (memory read)	2ns
– Register read	1ns
– ALU operation	2ns
– Get data (mem. Read)	2ns
– Register write	1ns
– Total	8ns
■ Store word (no register write)	7ns

Time for beq (I-Type)

■ ALU (R-type)	6ns
■ Load word (I-type)	8ns
■ Store word (I-type)	7ns
■ Branch on equal (I-type)	
– Fetch (memory read)	2ns
– Register read	1ns
– ALU operation	2ns
– Total	5ns

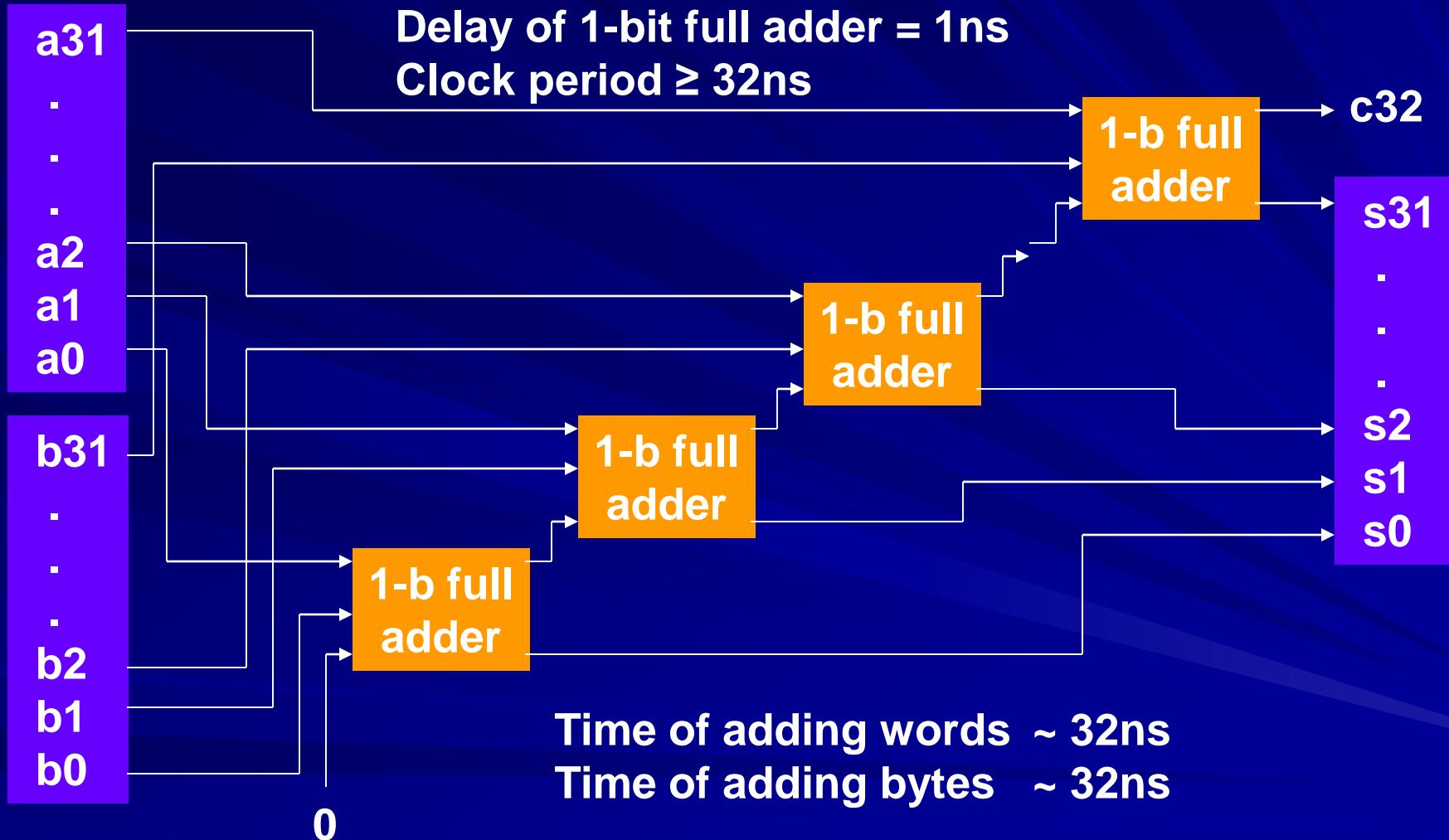
Time for Jump (J-Type)

■ ALU (R-type)	6ns
■ Load word (I-type)	8ns
■ Store word (I-type)	7ns
■ Branch on equal (I-type)	5ns
■ Jump (J-type)	
– Fetch (memory read)	2ns
– Total	2ns

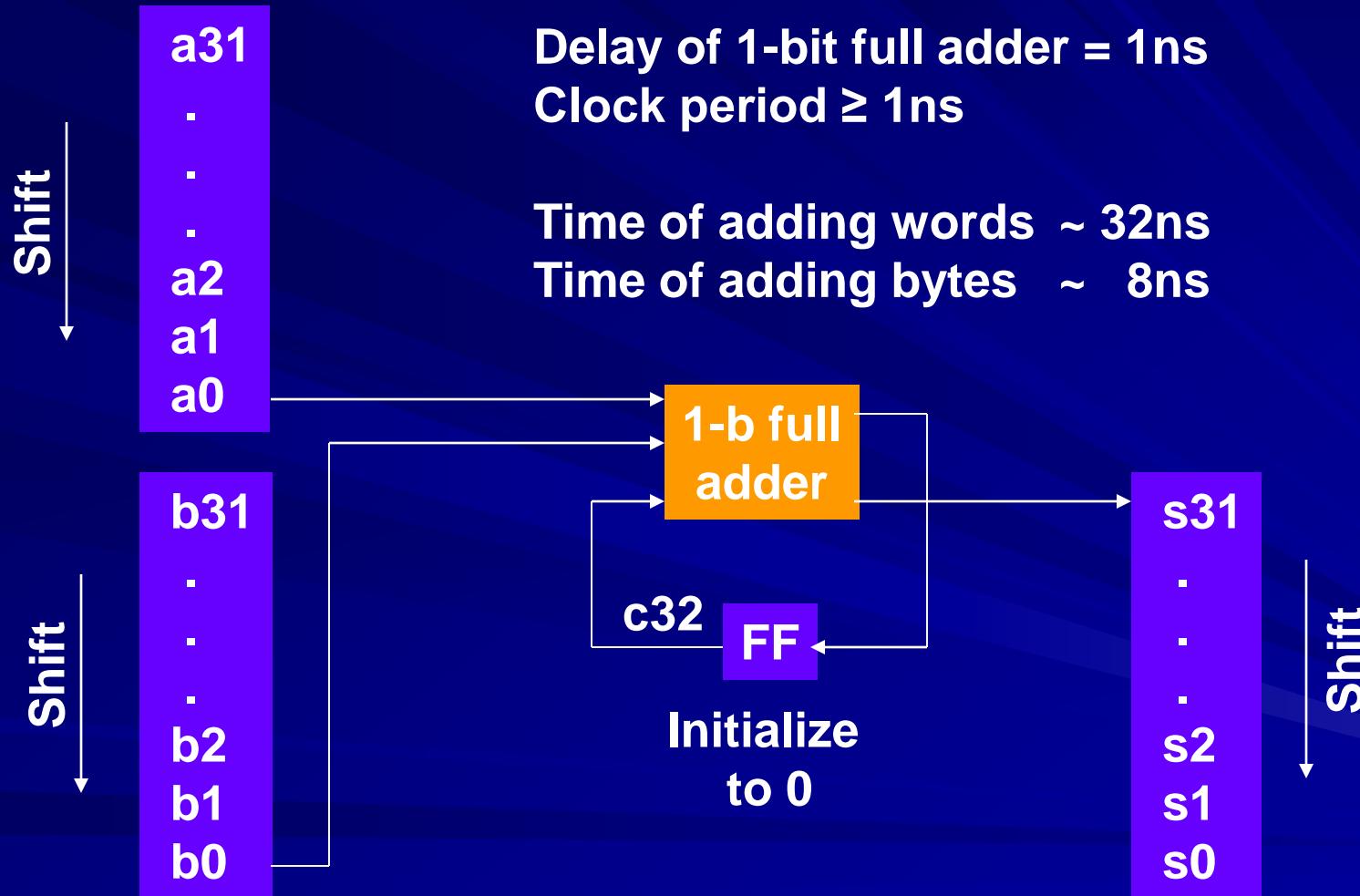
How Fast Can Clock Run?

- If every instruction is executed in one clock cycle, then:
 - Clock period must be at least 8ns to perform the longest instruction, i.e., lw .
 - This is a single cycle machine.
 - It is slower because many instructions take less than 8ns but are still allowed that much time.
- Method of speeding up: Use multicycle datapath.

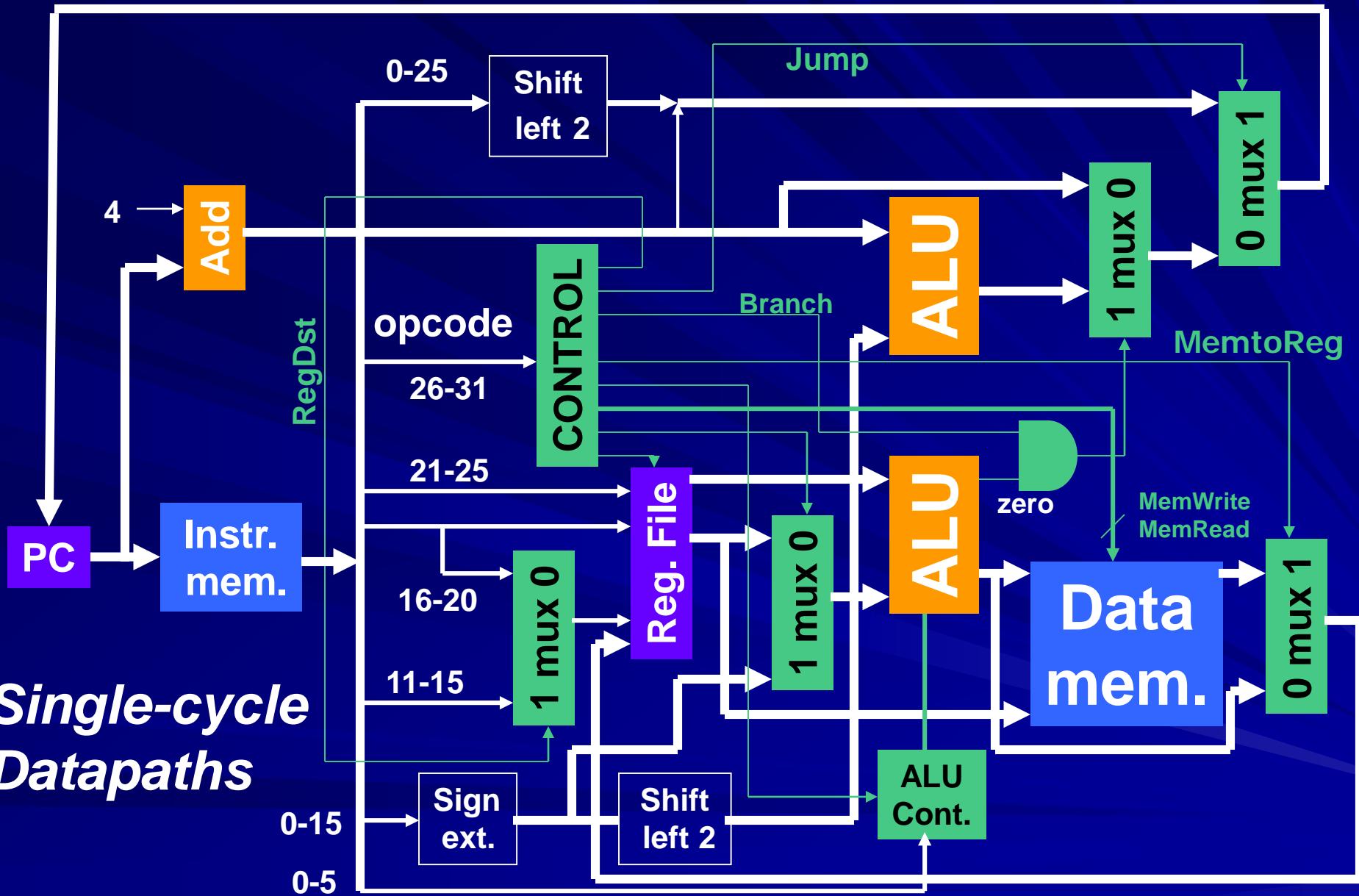
A Single Cycle Example



A Multicycle Implementation



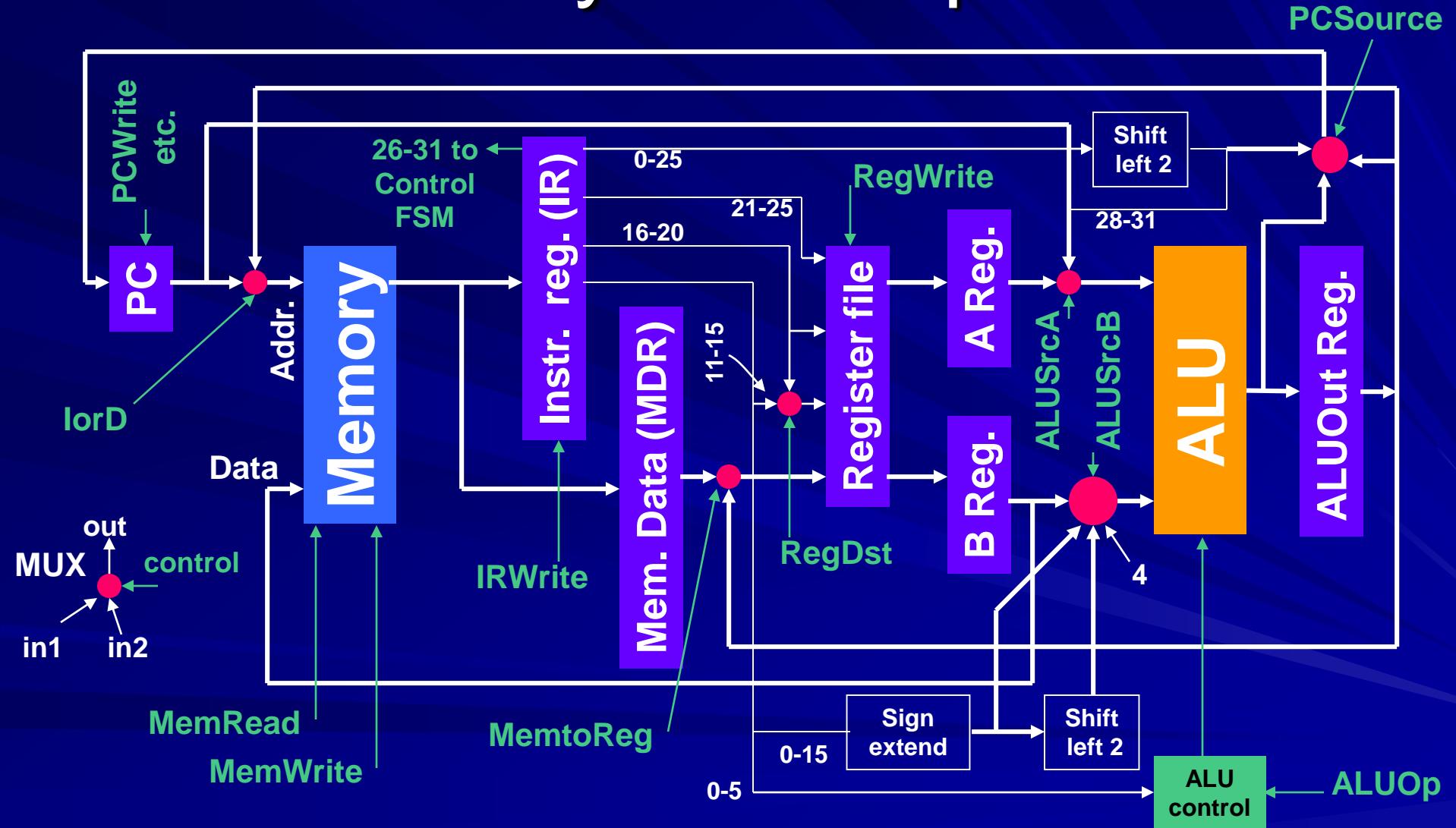
Single-cycle Datapaths



Multicycle Datapath Requirements

- Only one ALU, since it can be reused.
- Single memory for instructions and data.
- Five registers added:
 - Instruction register (IR)
 - Memory data register (MDR)
 - Three ALU registers, A and B for inputs and ALUOut for output

Multicycle Datapath



3 to 5 Cycles for an Instruction

Step	R-type (4 cycles)	Mem. Ref. (4 or 5 cycles)	Branch type (3 cycles)	J-type (3 cycles)
Instruction fetch	$IR \leftarrow \text{Memory}[PC]; PC \leftarrow PC + 4$			
Instr. decode/ Reg. fetch	$A \leftarrow \text{Reg}(IR[21-25]); B \leftarrow \text{Reg}(IR[16-20])$ $ALUOut \leftarrow PC + \{(\text{sign extend } IR[0-15]) \ll 2\}$			
Execution, addr. Comp., branch & jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + \text{sign extend } (IR[0-15])$	If $(A = B)$ then $PC \leftarrow ALUOut$	$PC \leftarrow PC[28-31]$ $(IR[0-25] \ll 2)$
Mem. Access or R-type completion	$\text{Reg}(IR[11-15]) \leftarrow ALUOut$	$MDR \leftarrow M[ALUout]$ or $M[ALUOut] \leftarrow B$		
Memory read completion		$\text{Reg}(IR[16-20]) \leftarrow MDR$		

Cycle 1 of 5: Instruction Fetch (IF)

■ Read instruction into IR, $M[PC] \rightarrow IR$

■ Control signals used:

■ lorD	=	0	select PC
■ MemRead	=	1	read memory
■ IRWrite	=	1	write IR

■ Increment PC, $PC + 4 \rightarrow PC$

■ Control signals used:

■ ALUSrcA	=	0	select PC into ALU
■ ALUSrcB	=	01	select constant 4
■ ALUOp	=	00	ALU adds
■ PCSource	=	00	select ALU output
■ PCWrite	=	1	write PC

Cycle 2 of 5: Instruction Decode (ID)

	31-26	25-21	20-16	15-11	10-6	5-0
R	opcode	reg 1	reg 2	reg 3	shamt	fncode
I	opcode	reg 1	reg 2	word address increment		
J	opcode			word address jump		

- Control unit decodes instruction
- Datapath prepares for execution
 - R and I types, reg 1 → A reg, reg 2 → B reg
 - No control signals needed
 - Branch type, compute branch address in ALUOut
 - ALUSrcA = 0 select PC into ALU
 - ALUSrcB = 11 Instr. Bits 0-15 shift 2 into ALU
 - ALUOp = 00 ALU adds

Cycle 3 of 5: Execute (EX)

- R type: execute function on reg A and reg B, result in ALUOut

- Control signals used:

■ ALUSrcA	=	1	A reg into ALU
■ ALUSrcB	=	00	B reg into ALU
■ ALUOp	=	10	instr. Bits 0-5 control ALU

- I type, lw or sw: compute memory address in ALUOut \leftarrow A reg + sign extend IR[0-15]

- Control signals used:

■ ALUSrcA	=	1	A reg into ALU
■ ALUSrcB	=	10	Instr. Bits 0-15 into ALU
■ ALUOp	=	00	ALU adds

Cycle 3 of 5: Execute (EX)

- I type, beq: subtract reg A and reg B, write ALUOut to PC

- Control signals used:

■ ALUSrcA = 1	A reg into ALU
■ ALUsrcB = 00	B reg into ALU
■ ALUOp = 01	ALU subtracts
■ If zero = 1, PCSource = 01	ALUOut to PC
■ If zero = 1, PCwriteCond =1	write PC
■ Instruction complete, go to IF	

- J type: write jump address to $PC \leftarrow IR[0-25] \text{ shift } 2$ and four leading bits of PC

- Control signals used:

■ PCSOURCE = 10	
■ PCWrite = 1	write PC
■ Instruction complete, go to IF	

Cycle 4 of 5: Reg Write/Memory

■ R type, write destination register from ALUOut

■ Control signals used:

■ RegDst	=	1	Instr. Bits 11-15 specify reg.
■ MemtoReg	=	0	ALUOut into reg.
■ RegWrite	=	1	write register
■ Instruction complete, go to IF			

■ I type, lw: read M[ALUOut] into MDR

■ Control signals used:

■ lrd	=	1	select ALUOut as mem adr.
■ MemRead	=	1	read memory to MDR

■ I type, sw: write M[ALUOut] from B reg

■ Control signals used:

■ lrd	=	1	select ALUOut as mem adr.
■ MemWrite	=	1	write memory

■ **Instruction complete, go to IF**

Cycle 5 of 5: Reg Write

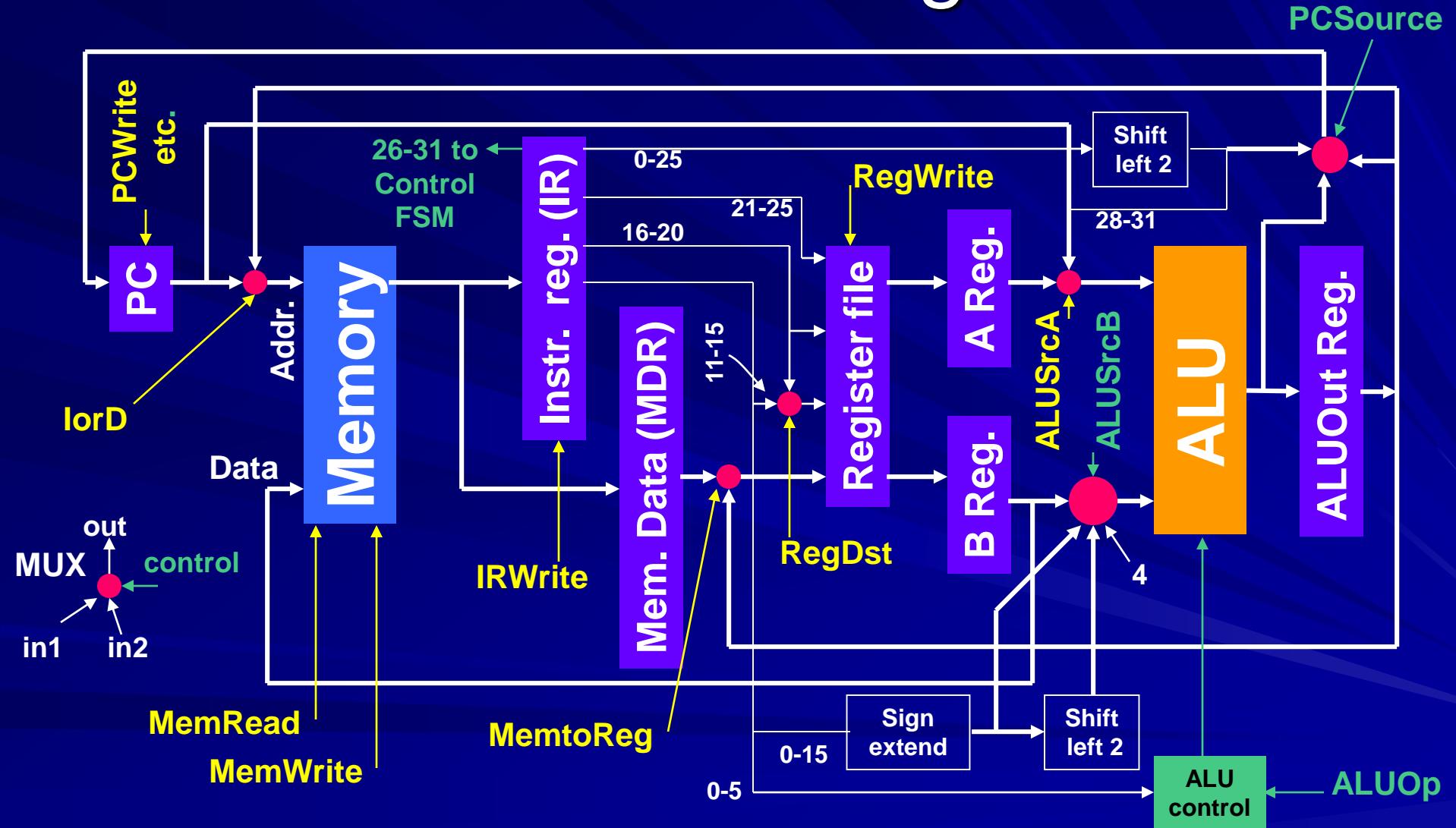
■ I type, lw: write MDR to reg[IR(16-20)]

■ Control signals used:

- RegDst = 0 instr. Bits 16-20 are write reg
- MemtoReg = 1 MDR to reg file write input
- RegWrite = 1 write reg. file
- **Instruction complete, go to IF**

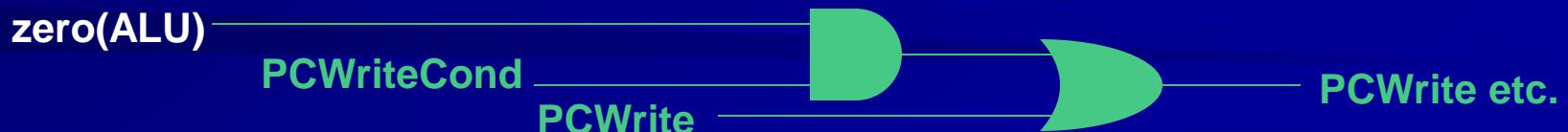
For an alternative method of designing datapath, see
N. Tredennick, *Microprocessor Logic Design, the Flowchart Method*,
Digital Press, 1987.

1-bit Control Signals

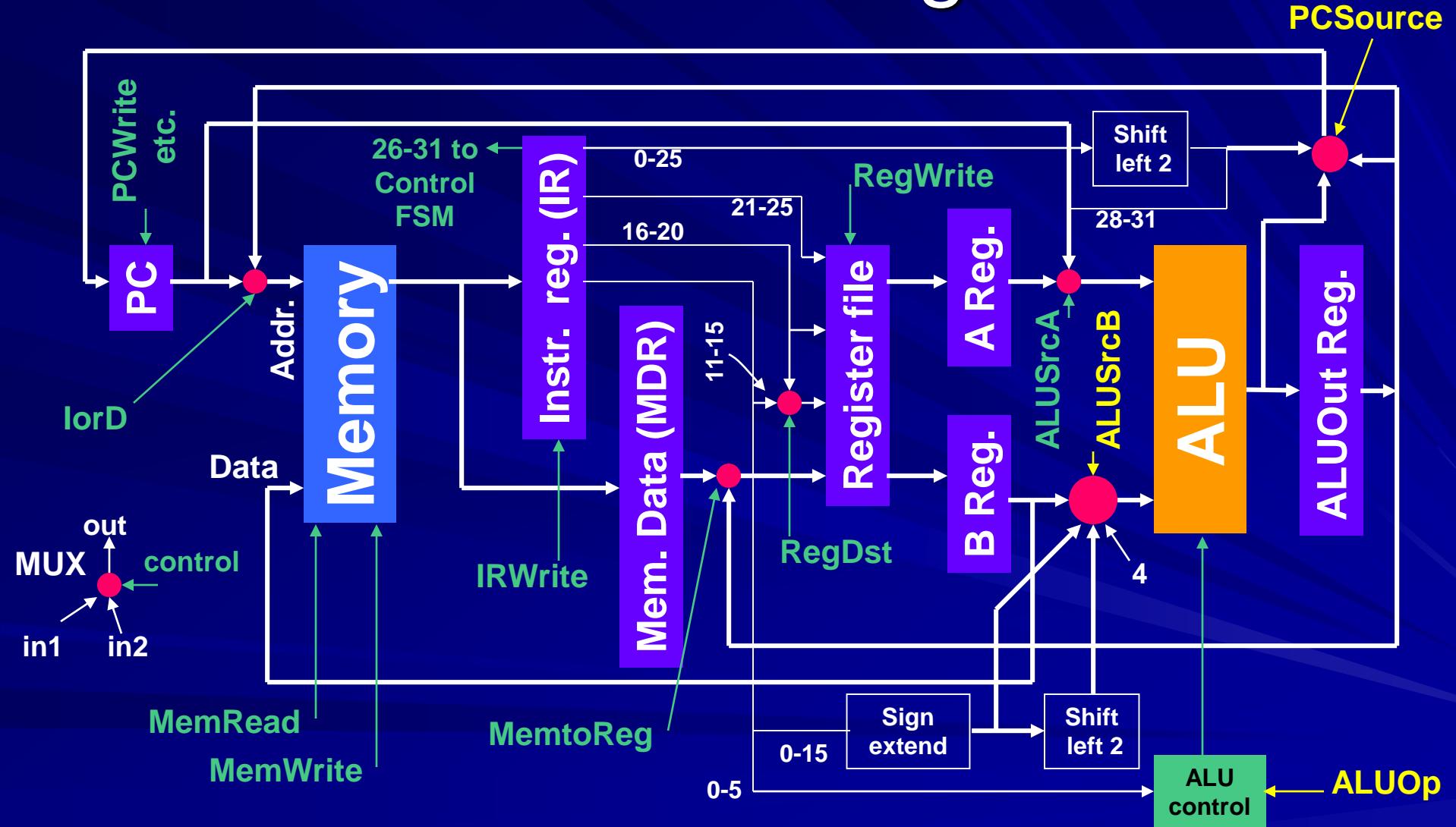


1-bit Control Signals

Signal name	Value = 0	Value =1
RegDst	Write reg. # = bit 16-20	Write reg. # = bit 11-15
RegWrite	No action	Write reg. \leftarrow Write data
ALUSrcA	First ALU Operand \leftarrow PC	First ALU Operand \leftarrow Reg. A
MemRead	No action	Mem.Data Output \leftarrow M[Addr.]
MemWrite	No action	M[Addr.] \leftarrow Mem. Data Input
MemtoReg	Reg.File Write In \leftarrow ALUOut	Reg.File Write In \leftarrow MDR
IorD	Mem. Addr. \leftarrow PC	Mem. Addr. \leftarrow ALUOut
IRWrite	No action	IR \leftarrow Mem.Data Output
PCWrite	No action	PC is written
PCWriteCond	No action	PC is written if zero(ALU)=1



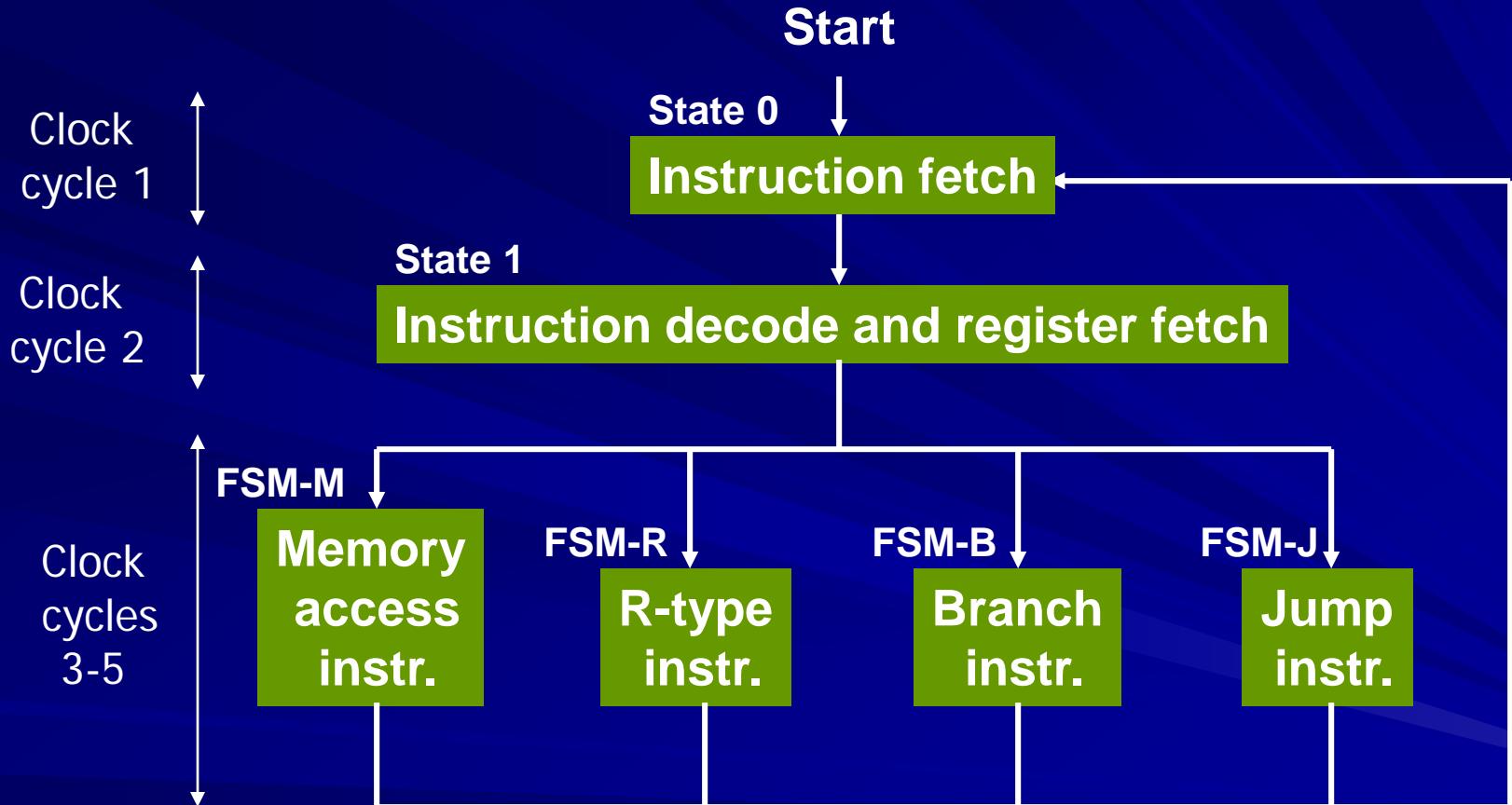
2-bit Control Signals



2-bit Control Signals

Signal name	Value	Action
ALUOp	00	ALU performs add
	01	ALU performs subtract
	10	Funct. field (0-5 bits of IR) determines ALU operation
ALUSrcB	00	Second input of ALU \leftarrow B reg.
	01	Second input of ALU \leftarrow 4 (constant)
	10	Second input of ALU \leftarrow 0-15 bits of IR sign ext. to 32b
	11	Second input of ALU \leftarrow 0-15 bits of IR sign ext. and left shift 2 bits
PCSource	00	ALU output (PC +4) sent to PC
	01	ALUOut (branch target addr.) sent to PC
	10	Jump address IR[0-25] shifted left 2 bits, concatenated with PC+4[28-31], sent to PC

Control: Finite State Machine

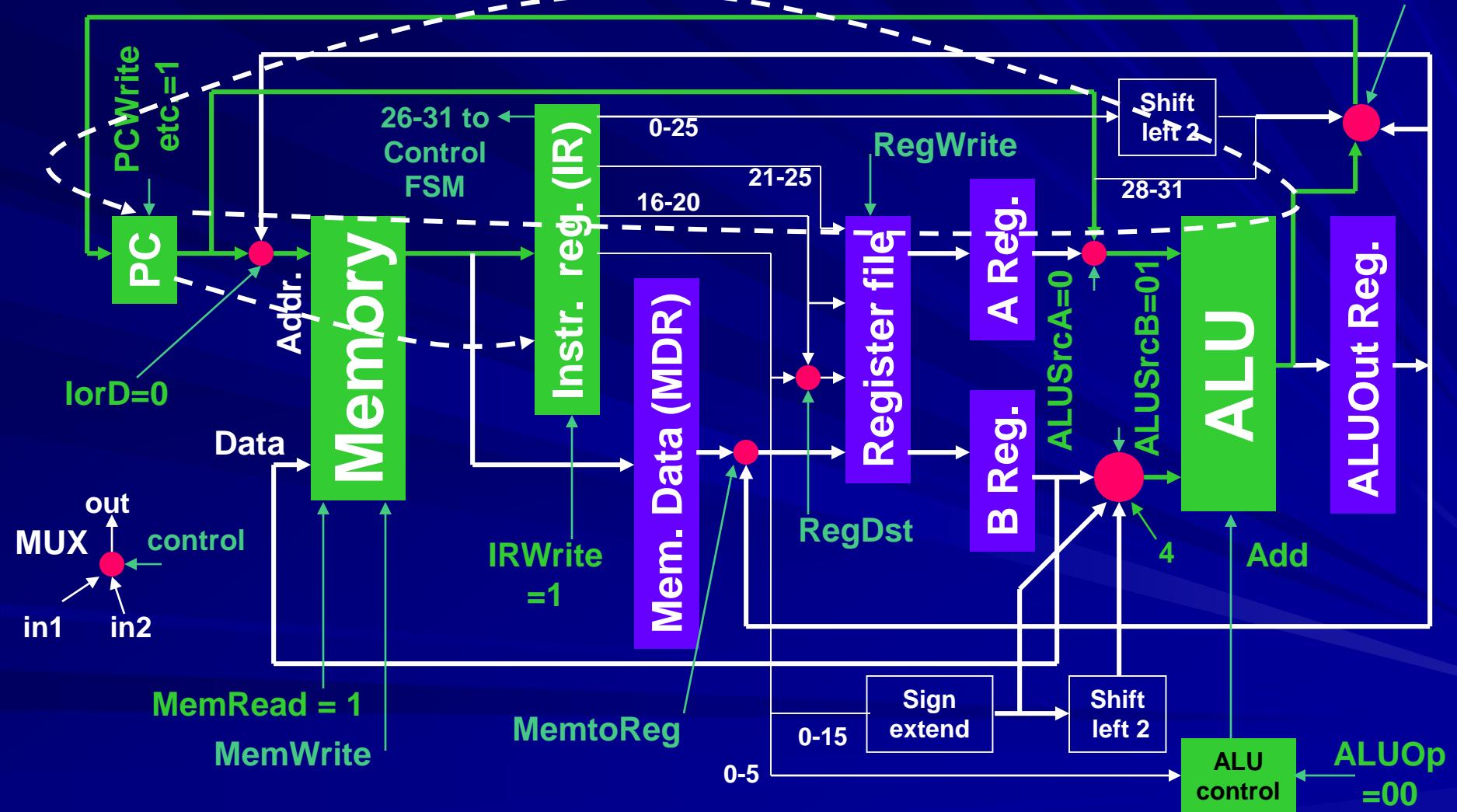


Instruction Fetch Operations

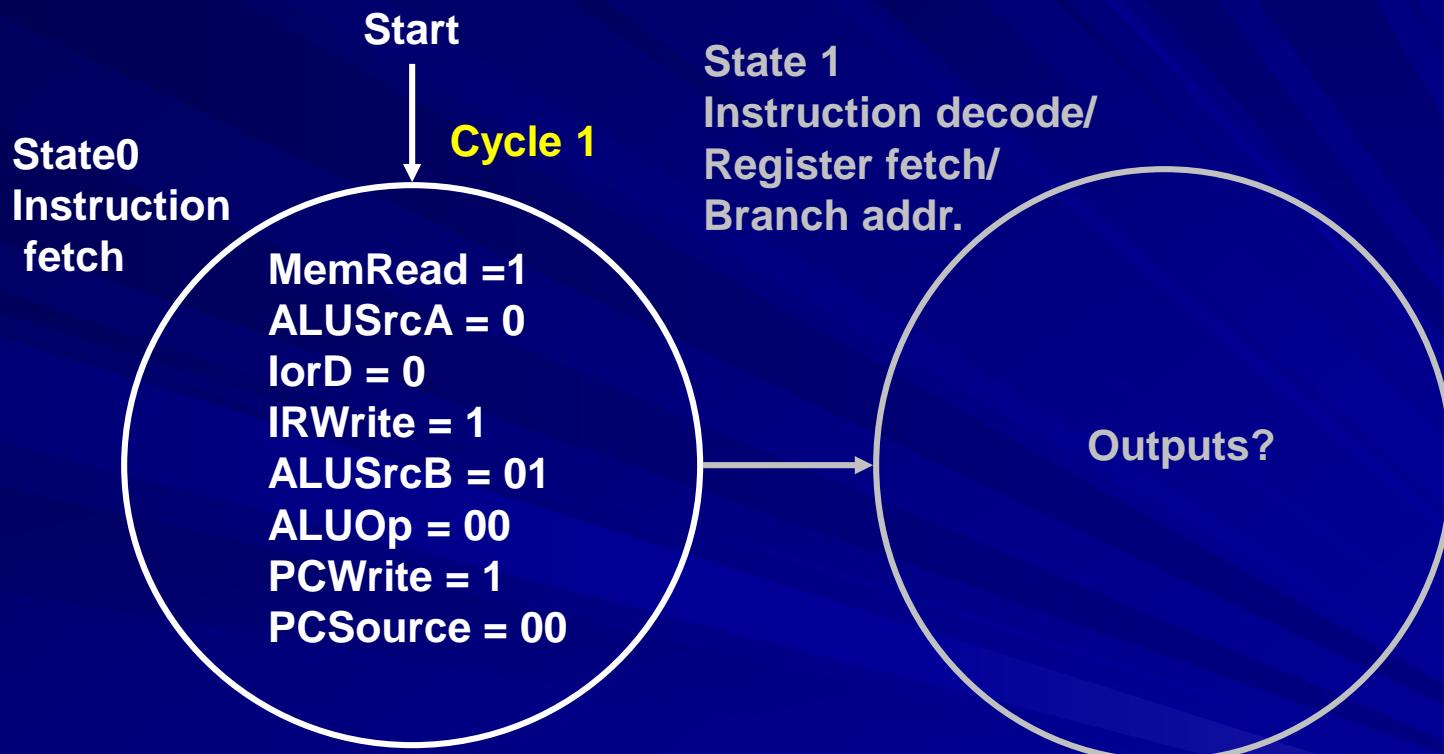
Step	R-type (4 cycles)	Mem. Ref. (4 or 5 cycles)	Branch type (3 cycles)	J-type (3 cycles)
Instruction fetch	$IR \leftarrow \text{Memory}[PC]; PC \leftarrow PC + 4$			
Instr. decode/ Reg. fetch	$A \leftarrow \text{Reg}(IR[21-25]); B \leftarrow \text{Reg}(IR[16-20])$ $ALUOut \leftarrow PC + \{(sign\ extend\ IR[0-15]) \ll 2\}$			
Execution, addr. Comp., branch & jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + sign\ extend\ (IR[0-15])$	If $(A = B)$ then $PC \leftarrow ALUOut$	$PC \leftarrow PC[28-31]$ $(IR[0-25] \ll 2)$
Mem. Access or R-type completion	$\text{Reg}(IR[11-15]) \leftarrow ALUOut$	$MDR \leftarrow M[ALUout]$ or $M[ALUOut] \leftarrow B$		
Memory read completion		$\text{Reg}(IR[16-20]) \leftarrow MDR$		

State 0: Instruction Fetch (CC1)

PCSource=00



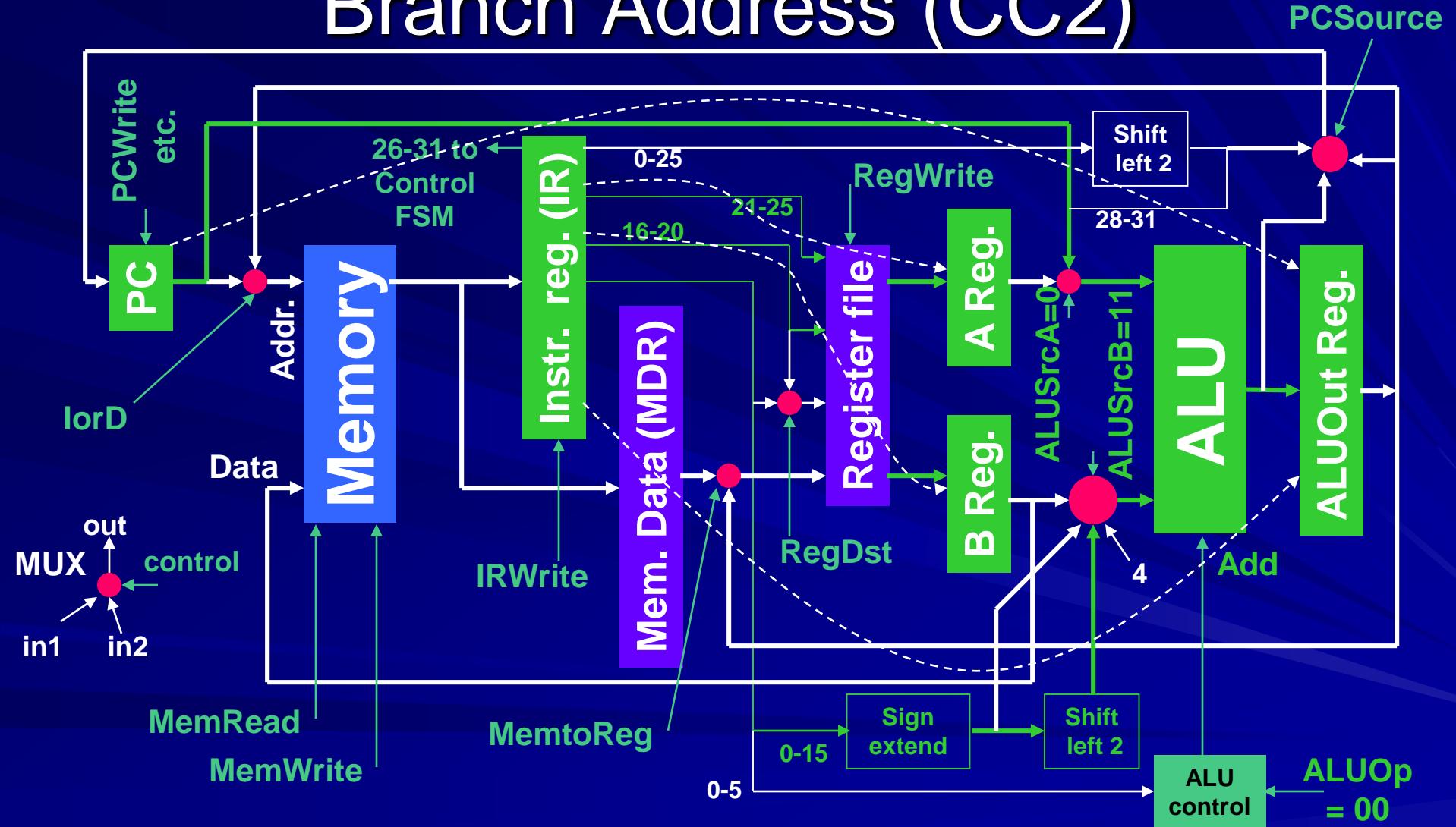
State 0 Control FSM Outputs



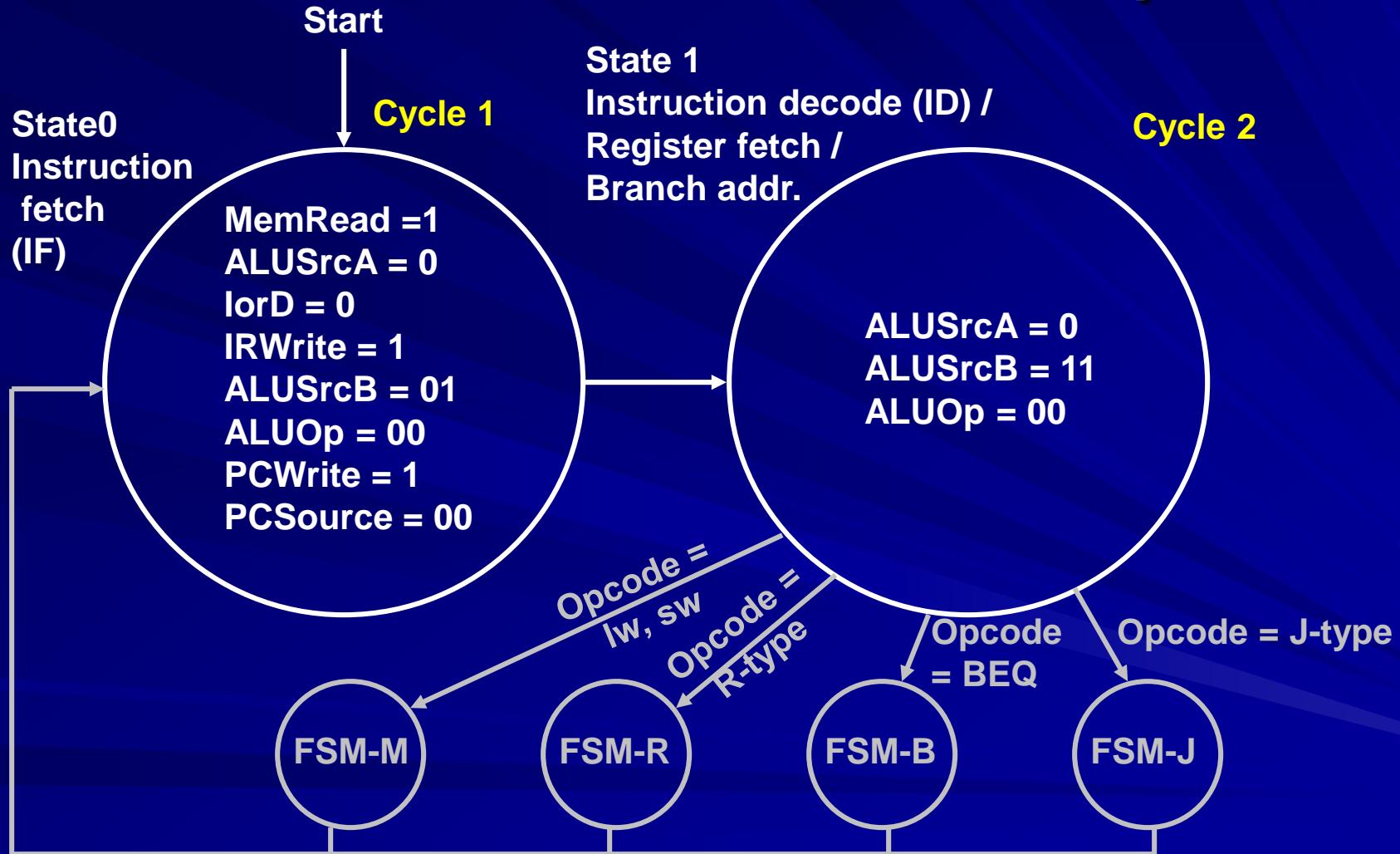
Instruction Decode Operations

Step	R-type (4 cycles)	Mem. Ref. (4 or 5 cycles)	Branch type (3 cycles)	J-type (3 cycles)
Instruction fetch	$IR \leftarrow \text{Memory}[PC]; PC \leftarrow PC+4$			
Instr. decode/ Reg. fetch	$A \leftarrow \text{Reg}(IR[21-25]); B \leftarrow \text{Reg}(IR[16-20])$ $ALUOut \leftarrow PC + \{(sign\ extend\ IR[0-15]) \ll 2\}$			
Execution, addr. Comp., branch & jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + sign\ extend\ (IR[0-15])$	If $(A = B)$ then $PC \leftarrow ALUOut$	$PC \leftarrow PC[28-31]$ $(IR[0-25] \ll 2)$
Mem. Access or R-type completion	$\text{Reg}(IR[11-15]) \leftarrow ALUOut$	$MDR \leftarrow M[ALUout]$ or $M[ALUOut] \leftarrow B$		
Memory read completion		$\text{Reg}(IR[16-20]) \leftarrow MDR$		

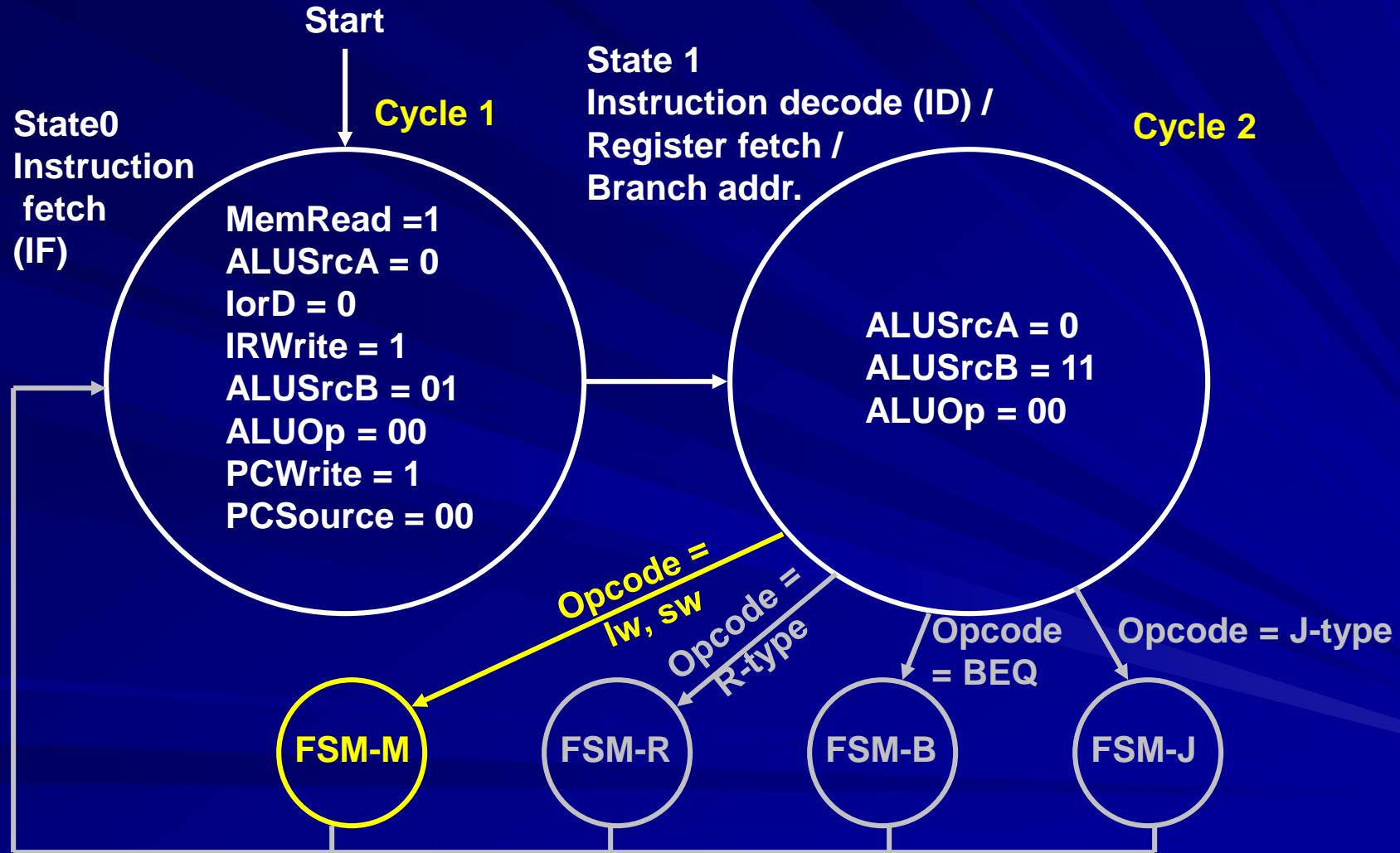
State 1: Instr. Decode/Reg. Fetch/ Branch Address (CC2)



State 1 Control FSM Outputs



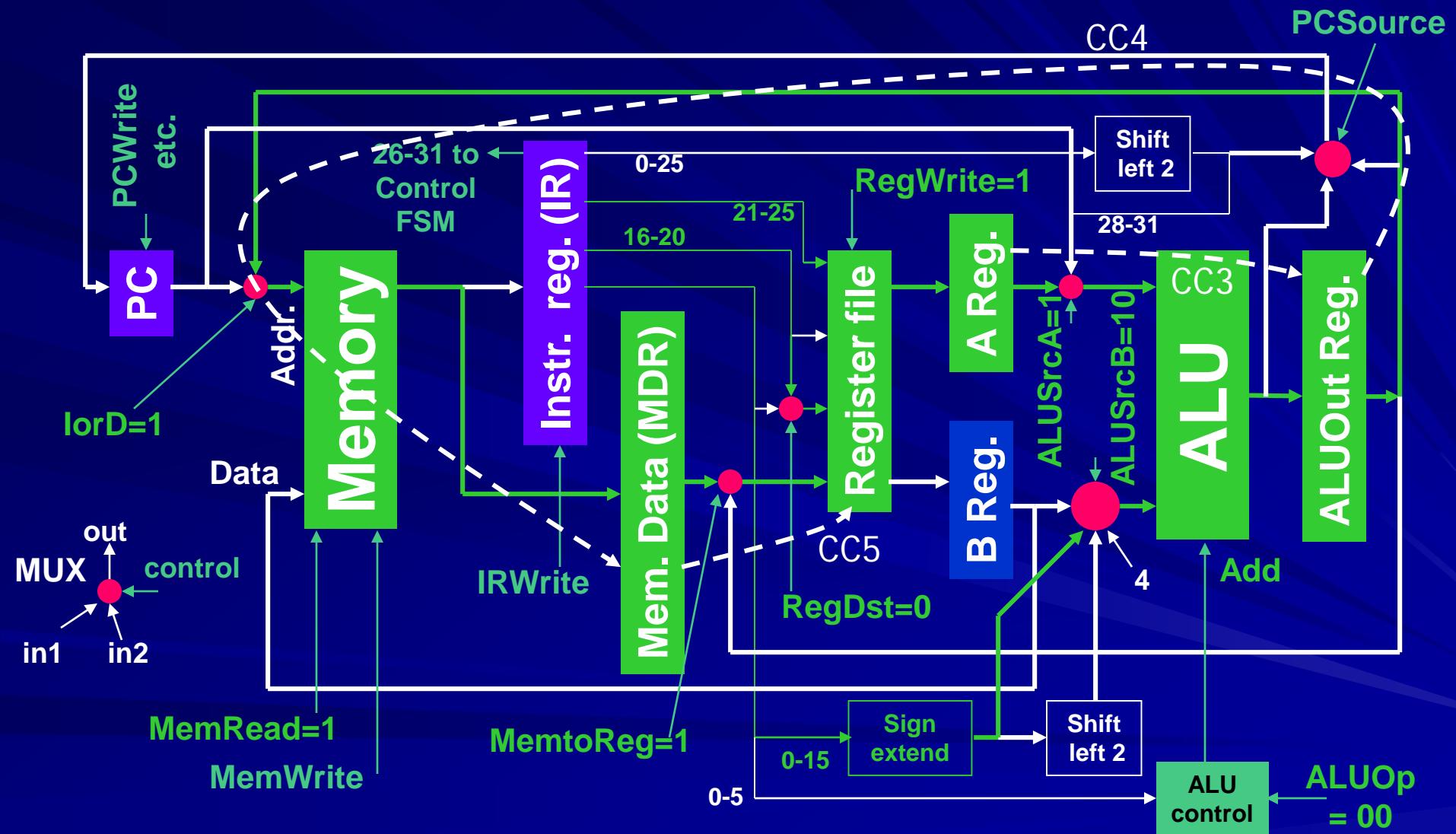
Mem Reference Sub-FSM



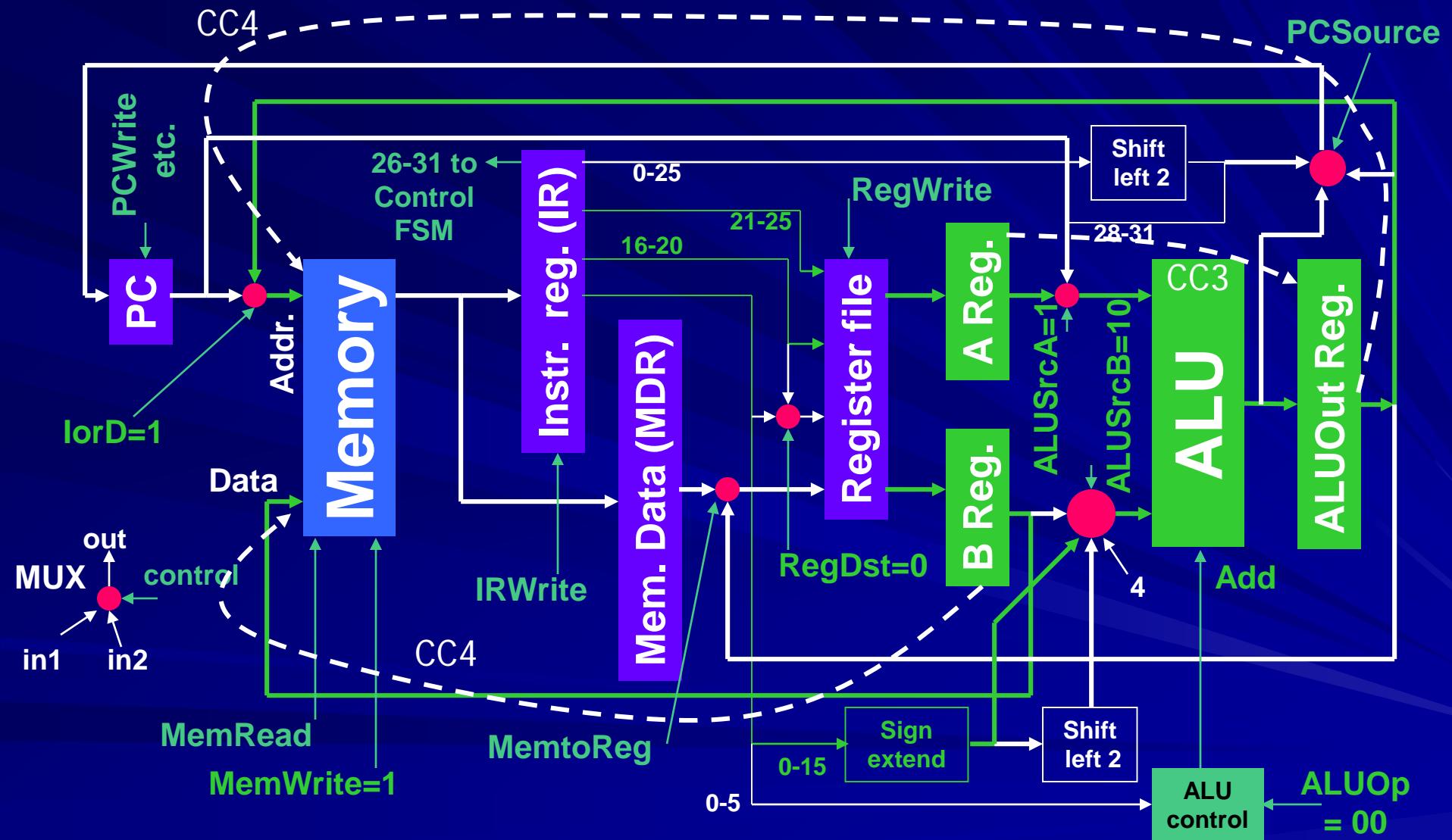
Memory Reference Operations

Step	R-type (4 cycles)	Mem. Ref. (4 or 5 cycles)	Branch type (3 cycles)	J-type (3 cycles)
Instruction fetch	$IR \leftarrow \text{Memory}[PC]; PC \leftarrow PC + 4$			
Instr. decode/ Reg. fetch	$A \leftarrow \text{Reg}(IR[21-25]); B \leftarrow \text{Reg}(IR[16-20])$ $ALUOut \leftarrow PC + \{\text{sign extend } IR[0-15]\ll 2\}$			
Execution, addr. Comp., branch & jump completion	$ALUOut \leftarrow A \text{ op } B$	ALUOut \leftarrow A+sign extend (IR[0-15])	If (A= =B) then $PC \leftarrow ALUOut$	$PC \leftarrow PC[28-31]$ $(IR[0-25]\ll 2)$
Mem. Access or R-type completion	$\text{Reg}(IR[11-15]) \leftarrow ALUOut$	MDR $\leftarrow M[ALUout]$ or $M[ALUOut] \leftarrow B$		
Memory read completion		Reg(IR[16-20]) \leftarrow MDR		

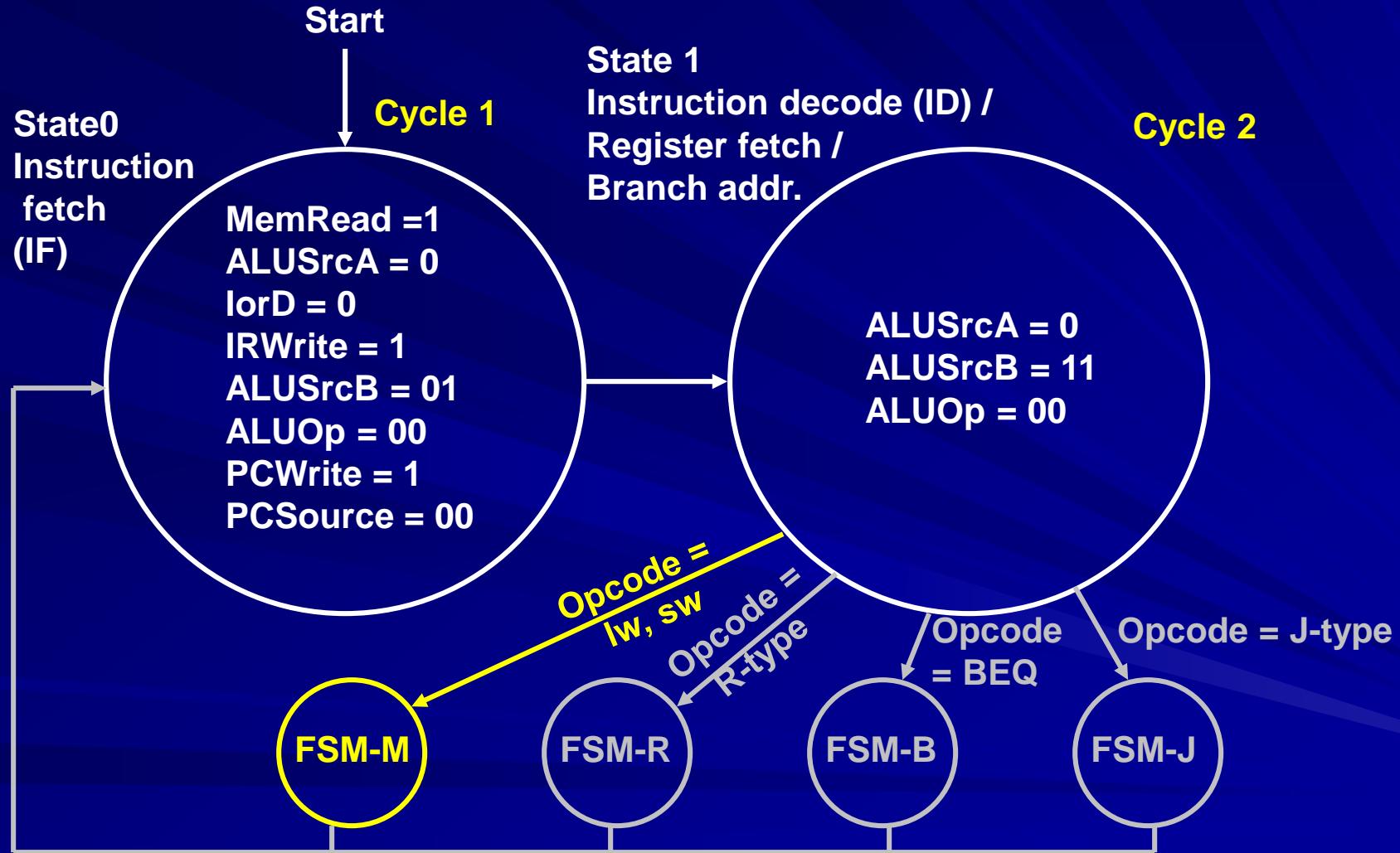
State 1 (Opcode = lw) → FSM-M (CC3-5)



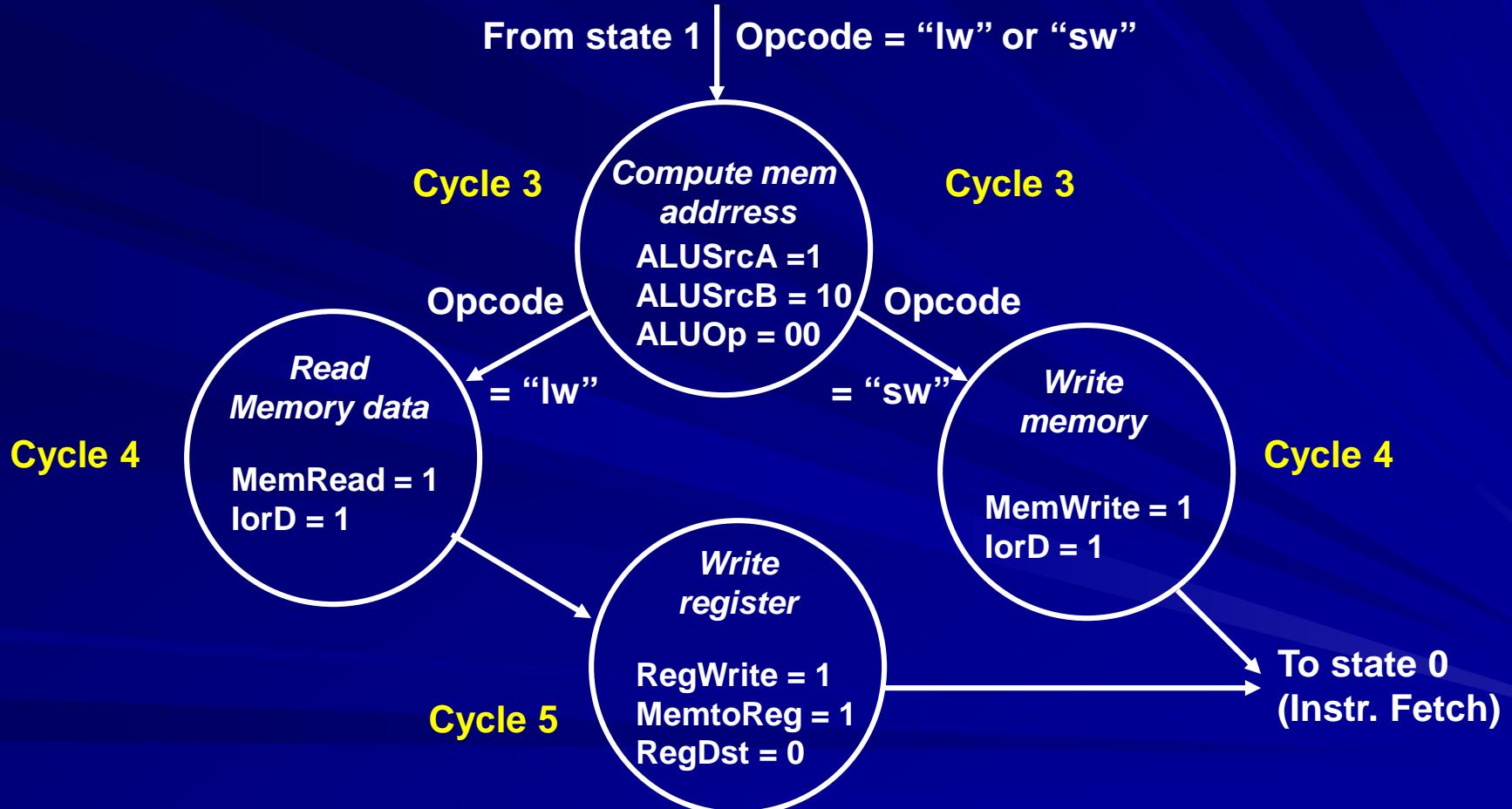
State 1 (Opcode= sw) → FSM-M (CC3-4)



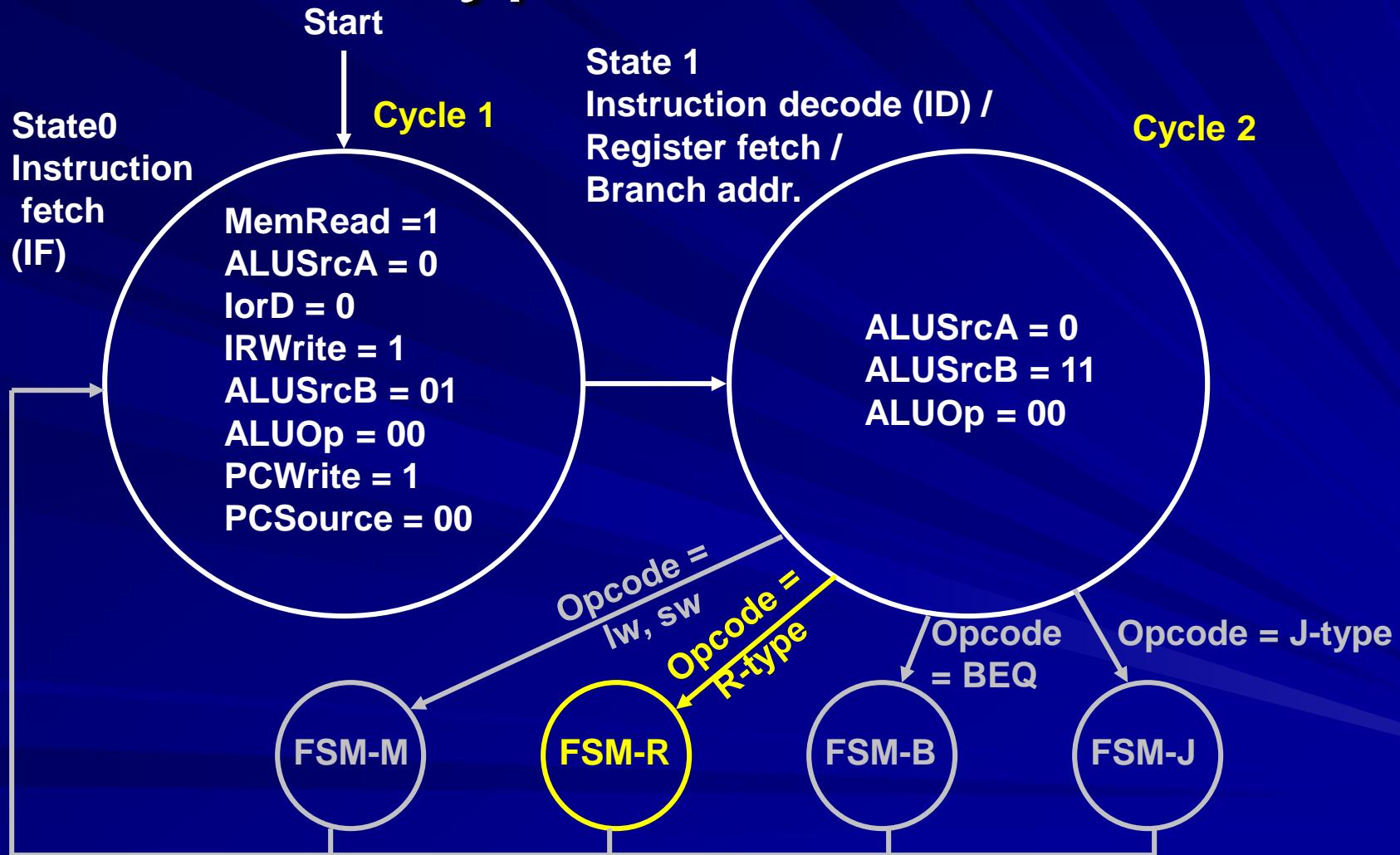
Mem Reference Sub-FSM



FSM-M (Memory Access)



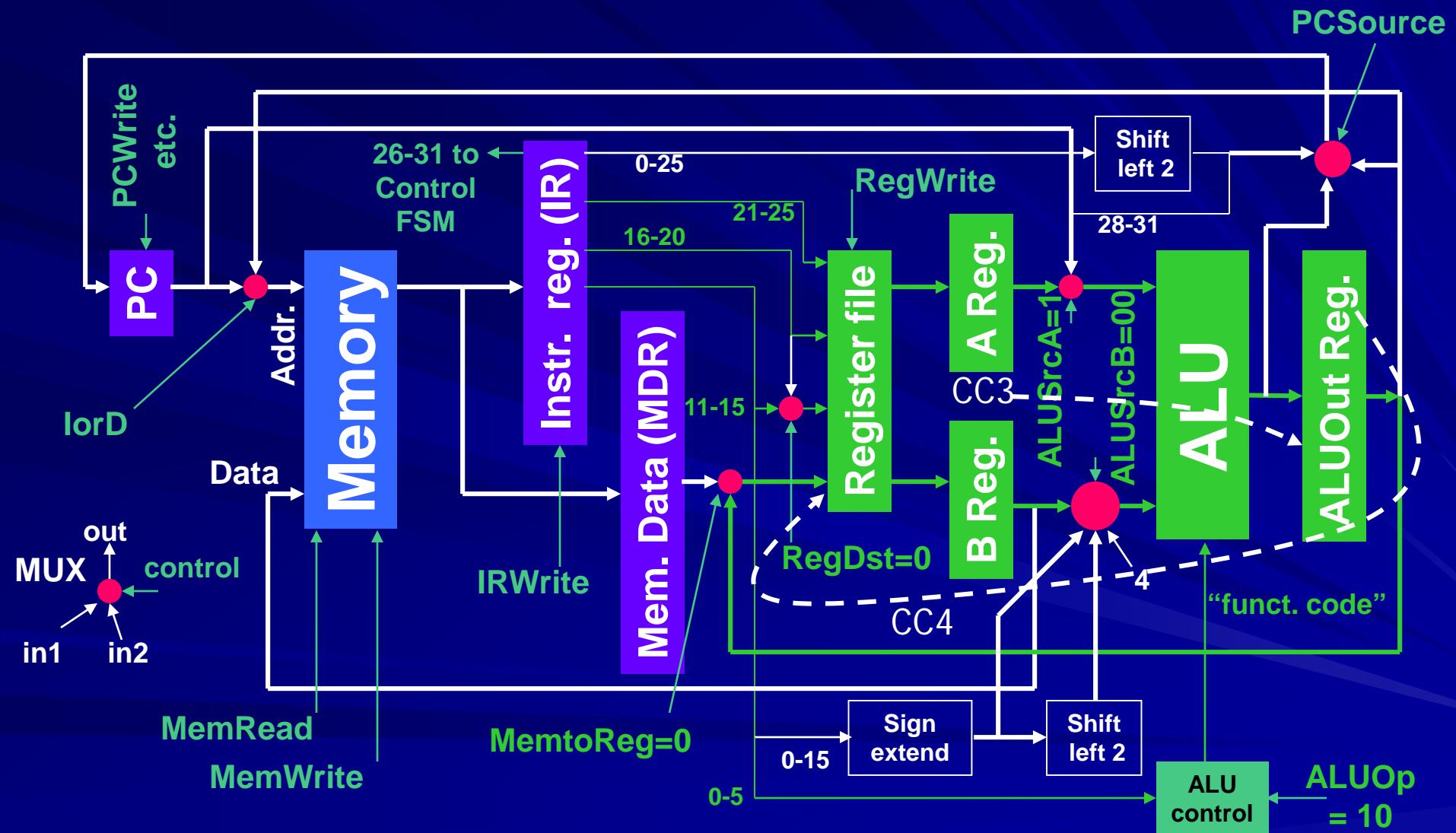
R-type Sub-FSM



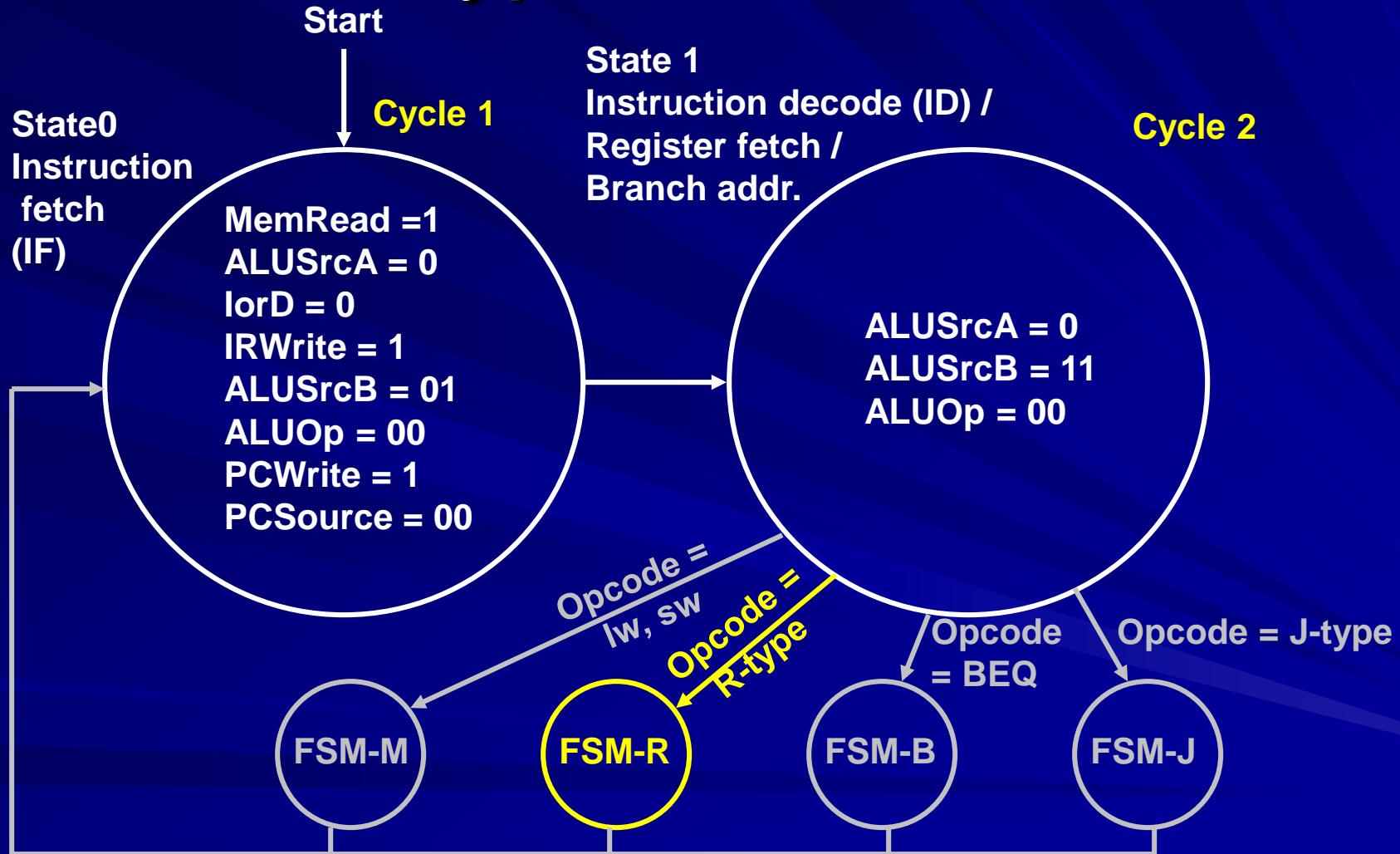
R-type Instruction Operations

Step	R-type (4 cycles)	Mem. Ref. (4 or 5 cycles)	Branch type (3 cycles)	J-type (3 cycles)
Instruction fetch	$IR \leftarrow \text{Memory}[PC]; PC \leftarrow PC + 4$			
Instr. decode/ Reg. fetch	$A \leftarrow \text{Reg}(IR[21-25]); B \leftarrow \text{Reg}(IR[16-20])$ $ALUOut \leftarrow PC + \{(\text{sign extend } IR[0-15]) \ll 2\}$			
Execution, addr. Comp., branch & jump completion	ALUOut \leftarrow A op B	$ALUOut \leftarrow$ $A + \text{sign extend } (IR[0-15])$	If $(A = B)$ then $PC \leftarrow ALUOut$	$PC \leftarrow PC[28-31]$ $(IR[0-25] \ll 2)$
Mem. Access or R-type completion	Reg(IR[11-15]) \leftarrow ALUOut	$MDR \leftarrow M[ALUout]$ or $M[ALUOut] \leftarrow B$		
Memory read completion		$Reg(IR[16-20]) \leftarrow MDR$		

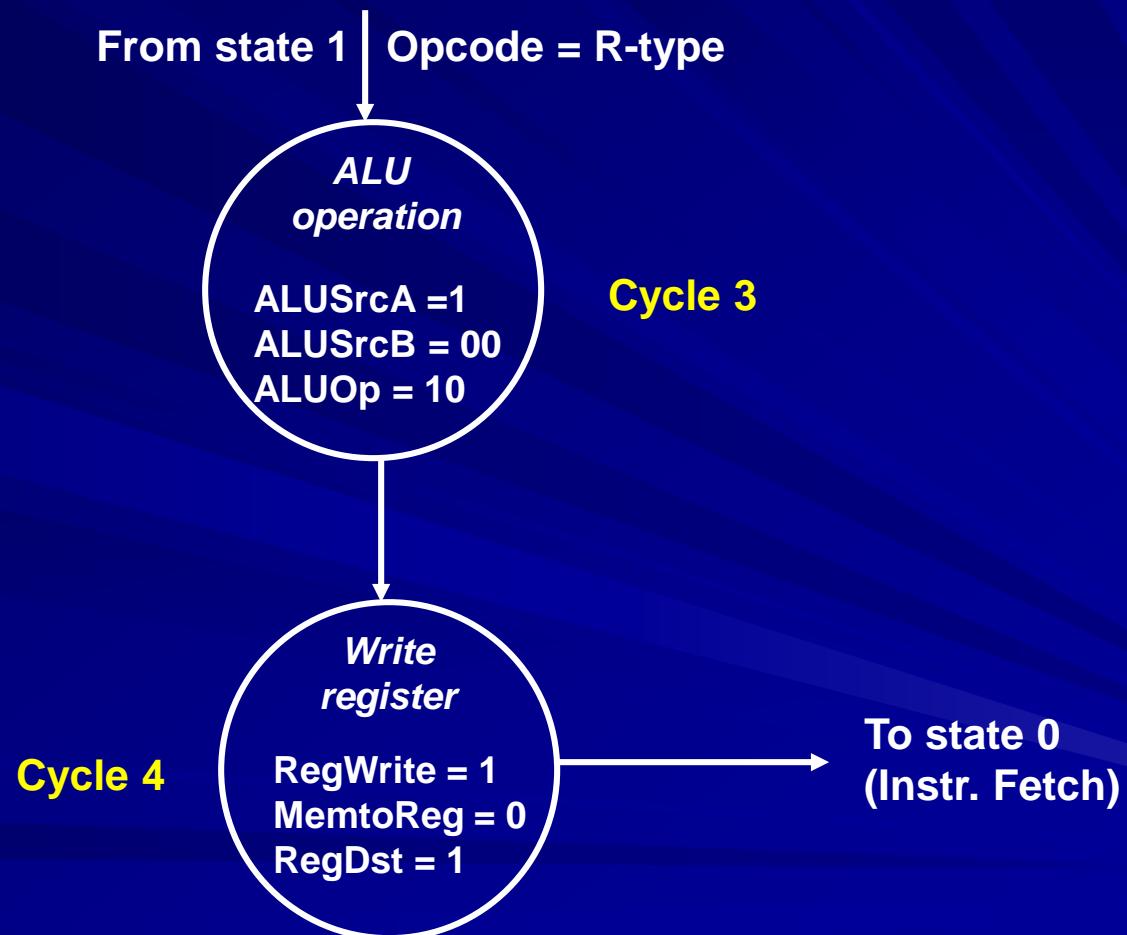
State 1(Opcode=R-type) → FSM-R (CC3-4)



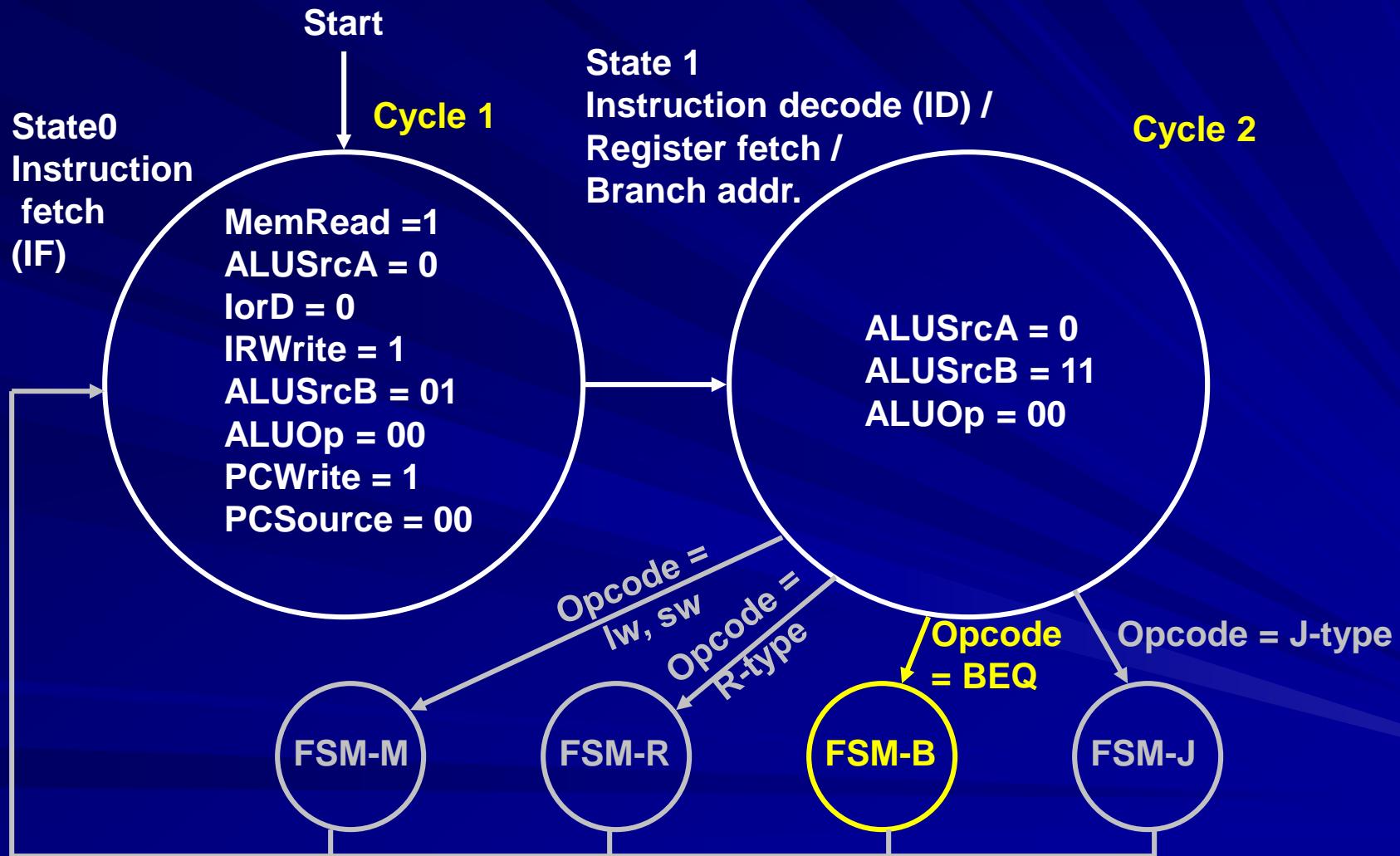
R-type Sub-FSM



FSM-R (R-type Instruction)



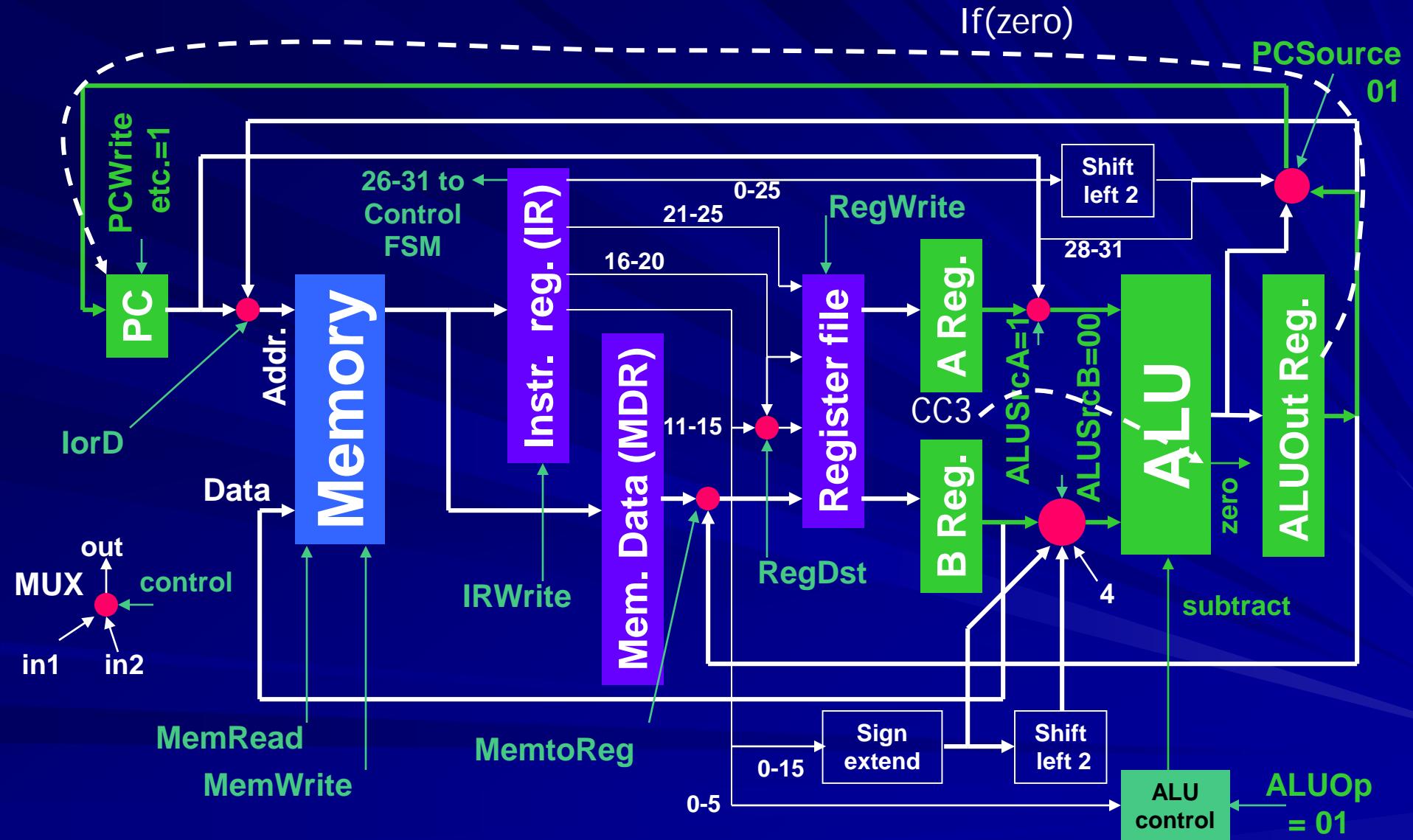
Branch Sub-FSM



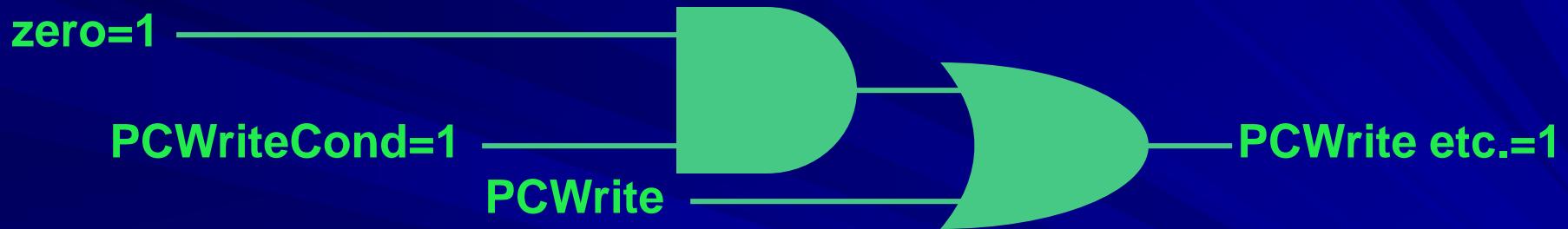
Branch Instruction Operations

Step	R-type (4 cycles)	Mem. Ref. (4 or 5 cycles)	Branch type (3 cycles)	J-type (3 cycles)
Instruction fetch	$IR \leftarrow \text{Memory}[PC]; PC \leftarrow PC + 4$			
Instr. decode/ Reg. fetch	$A \leftarrow \text{Reg}(IR[21-25]); B \leftarrow \text{Reg}(IR[16-20])$ $ALUOut \leftarrow PC + \{(\text{sign extend } IR[0-15]) \ll 2\}$			
Execution, addr. Comp., branch & jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + \text{sign extend } (IR[0-15])$	If (A = B) then $PC \leftarrow ALUOut$	$PC \leftarrow PC[28-31]$ $(IR[0-25] \ll 2)$
Mem. Access or R-type completion	$\text{Reg}(IR[11-15]) \leftarrow ALUOut$	$MDR \leftarrow M[ALUout]$ or $M[ALUOut] \leftarrow B$		
Memory read completion		$\text{Reg}(IR[16-20]) \leftarrow MDR$		

State 1 (Opcode = beq) → FSM-B (CC3)

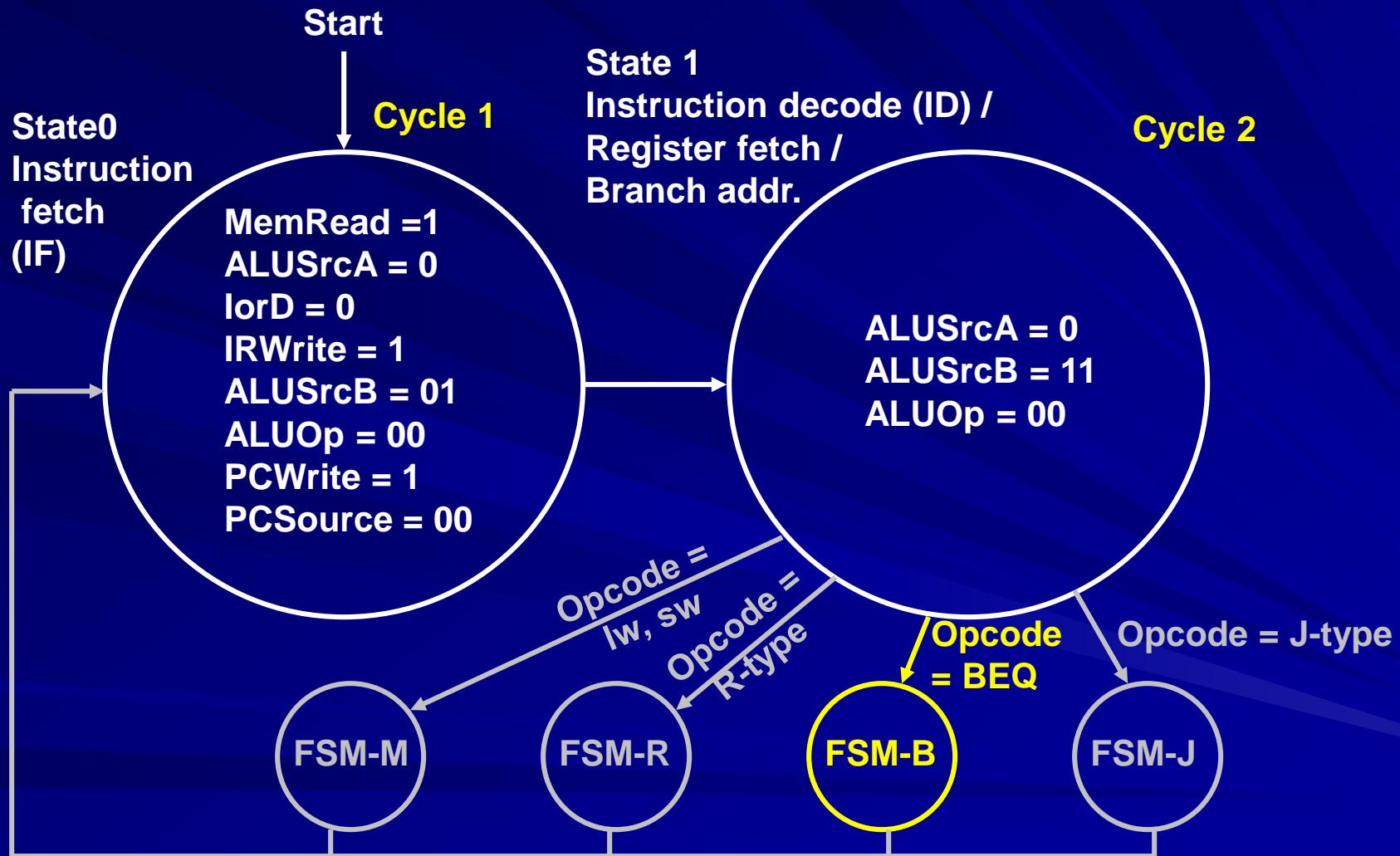


Write PC on “zero”

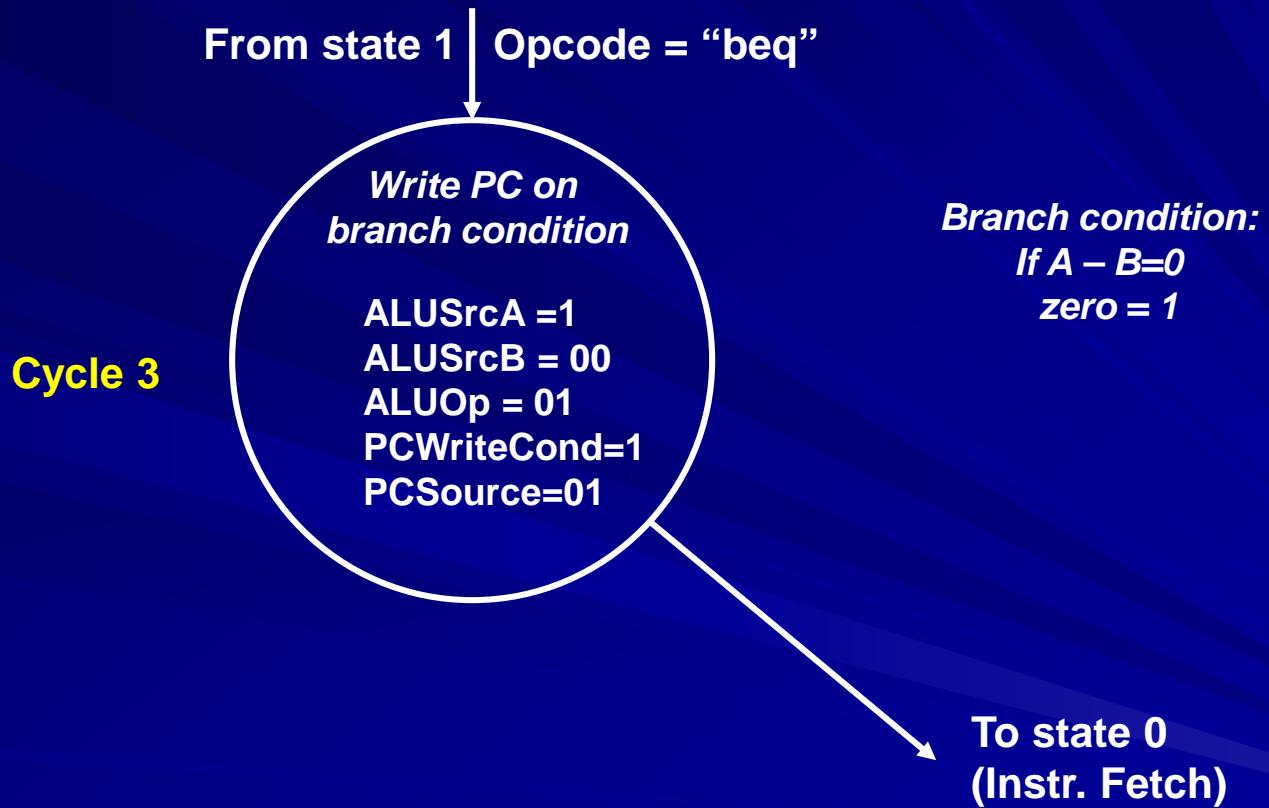


PCWrite = 1, unconditionally write PC
Cycle 1, fetch
Cycle 3, jump

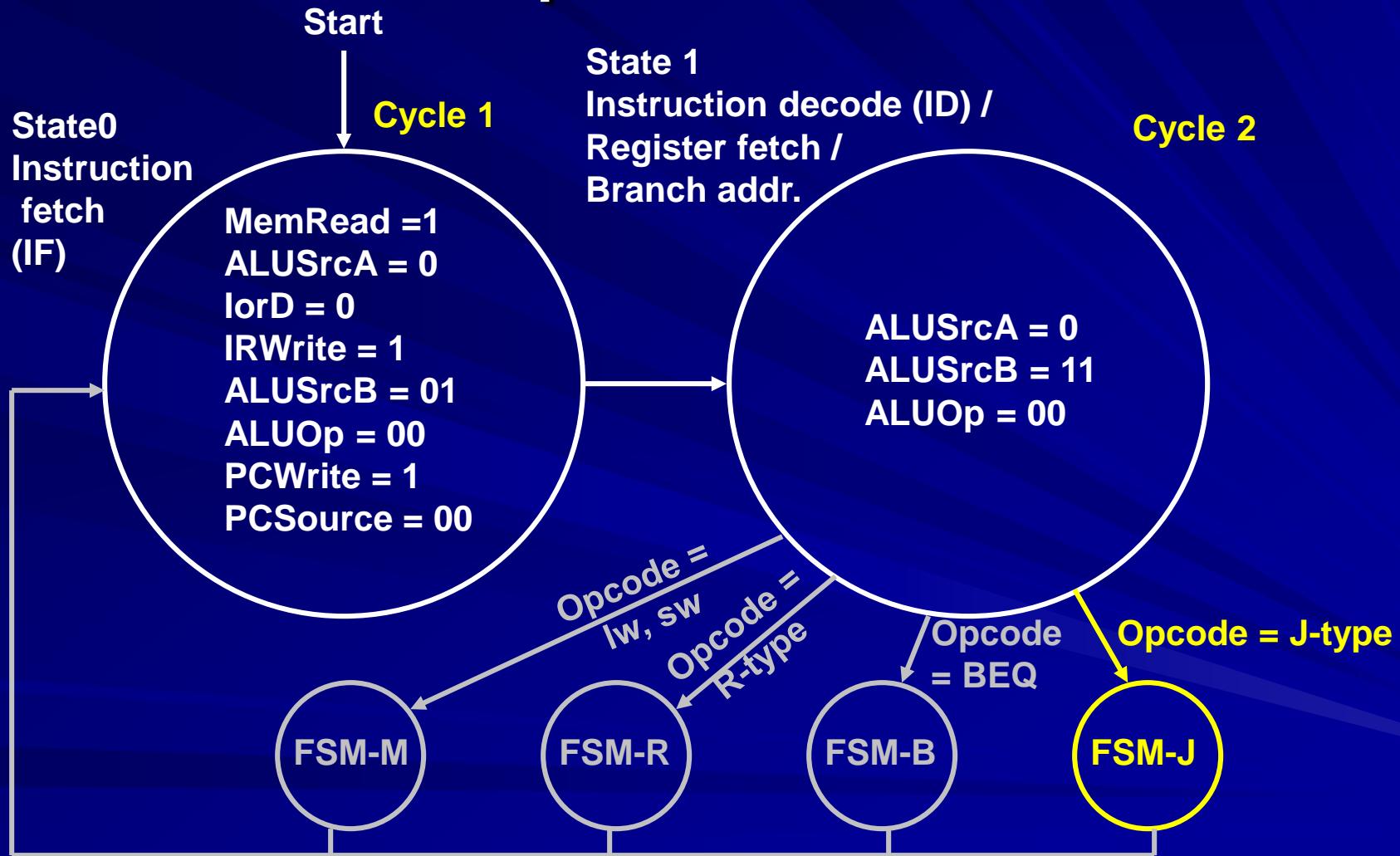
Branch Sub-FSM



FSM-B (Branch)



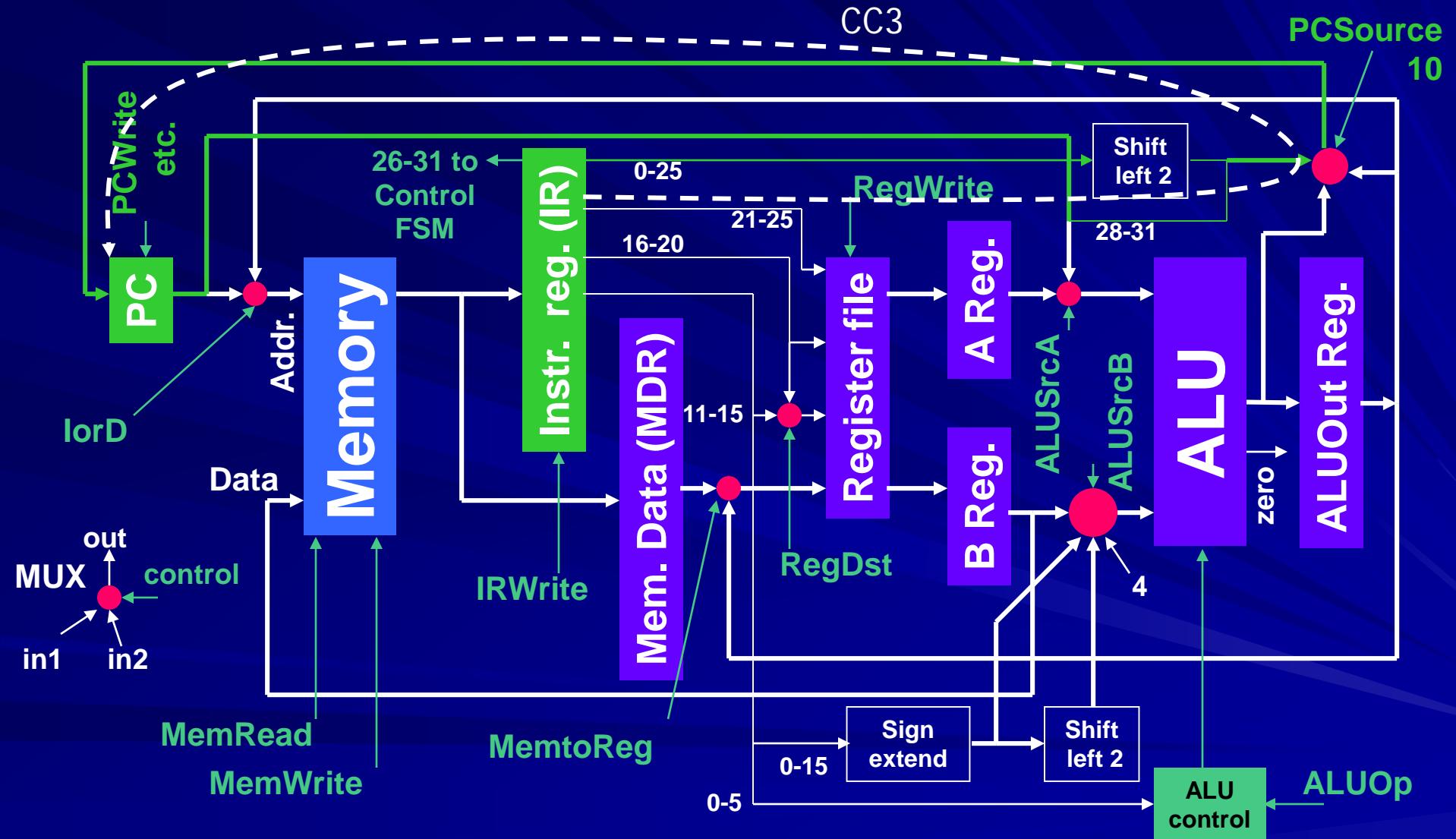
Jump Sub-FSM



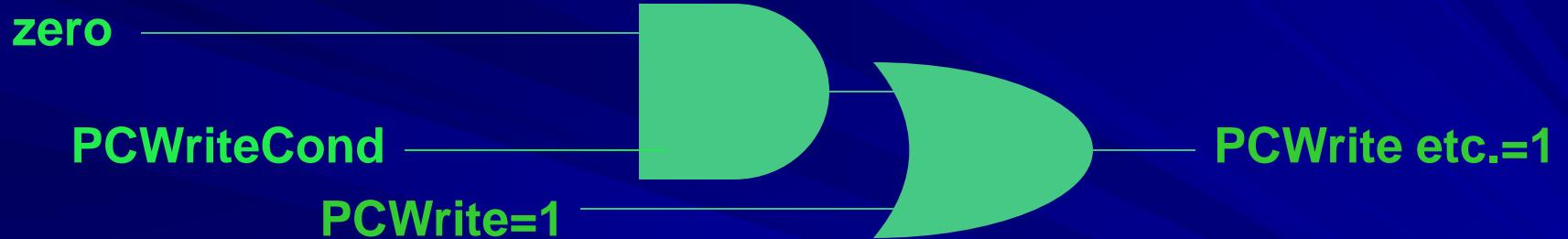
Jump Instruction Operations

Step	R-type (4 cycles)	Mem. Ref. (4 or 5 cycles)	Branch type (3 cycles)	J-type (3 cycles)
Instruction fetch	$IR \leftarrow \text{Memory}[PC]; PC \leftarrow PC + 4$			
Instr. decode/ Reg. fetch	$A \leftarrow \text{Reg}(IR[21-25]); B \leftarrow \text{Reg}(IR[16-20])$ $ALUOut \leftarrow PC + \{(\text{sign extend } IR[0-15]) \ll 2\}$			
Execution, addr. Comp., branch & jump completion	$ALUOut \leftarrow A \text{ op } B$	$ALUOut \leftarrow A + \text{sign extend } (IR[0-15])$	If $(A = B)$ then $PC \leftarrow ALUOut$	$PC \leftarrow PC[28-31]$ $(IR[0-25] \ll 2)$
Mem. Access or R-type completion	$\text{Reg}(IR[11-15]) \leftarrow ALUOut$	$MDR \leftarrow M[ALUout]$ or $M[ALUOut] \leftarrow B$		
Memory read completion		$\text{Reg}(IR[16-20]) \leftarrow MDR$		

State 1 (Opcode = j) → FSM-J (CC3)



Write PC

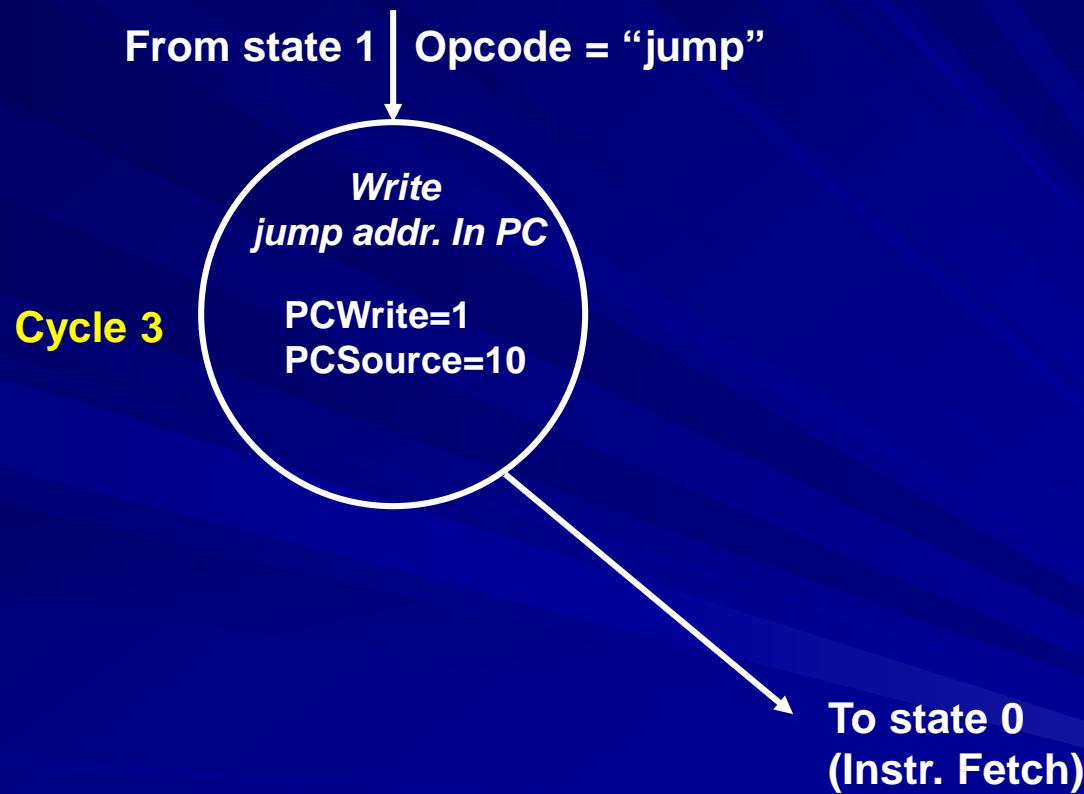


$\text{PCWrite} = 1$, unconditionally write PC

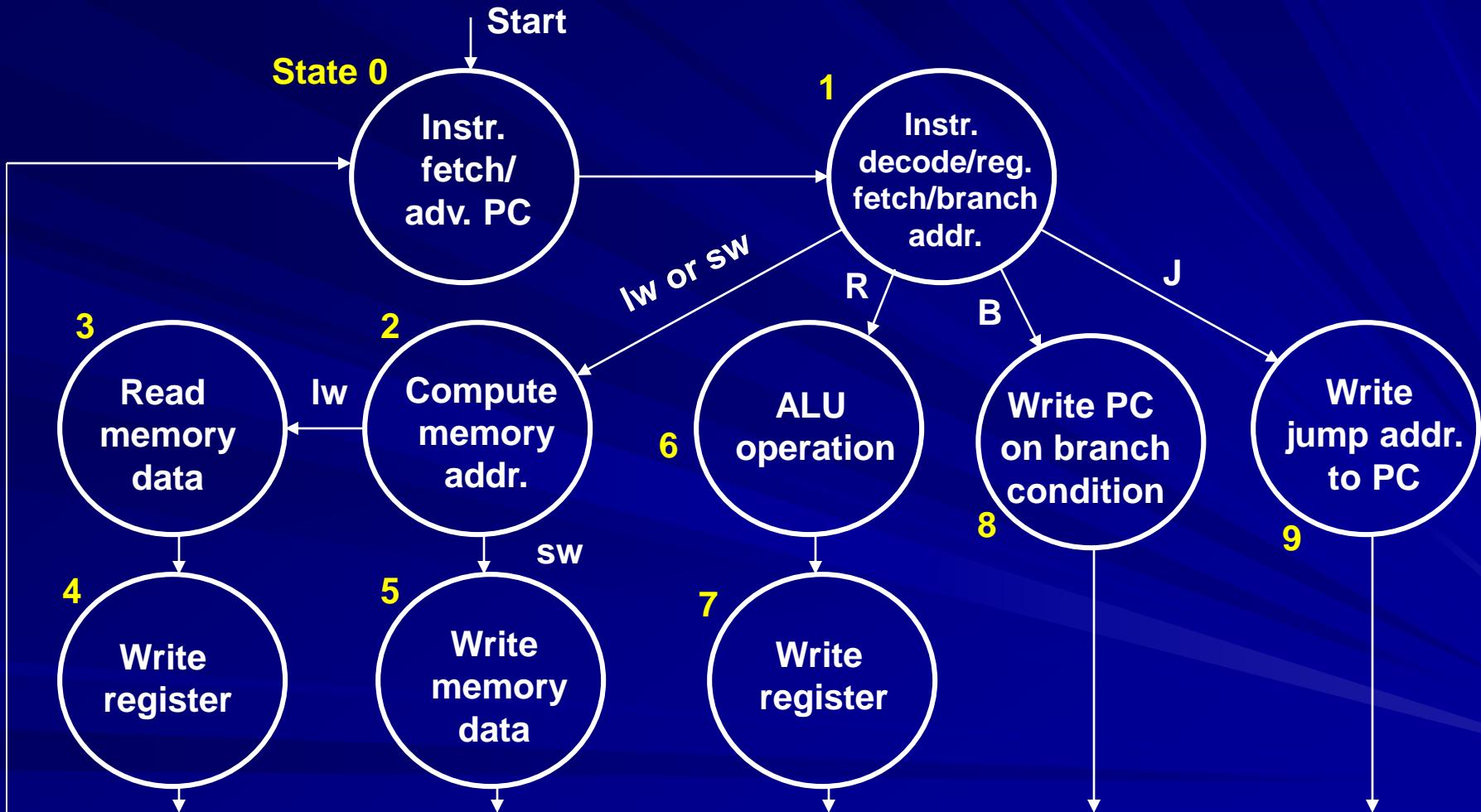
Cycle 1, fetch

Cycle 3, jump

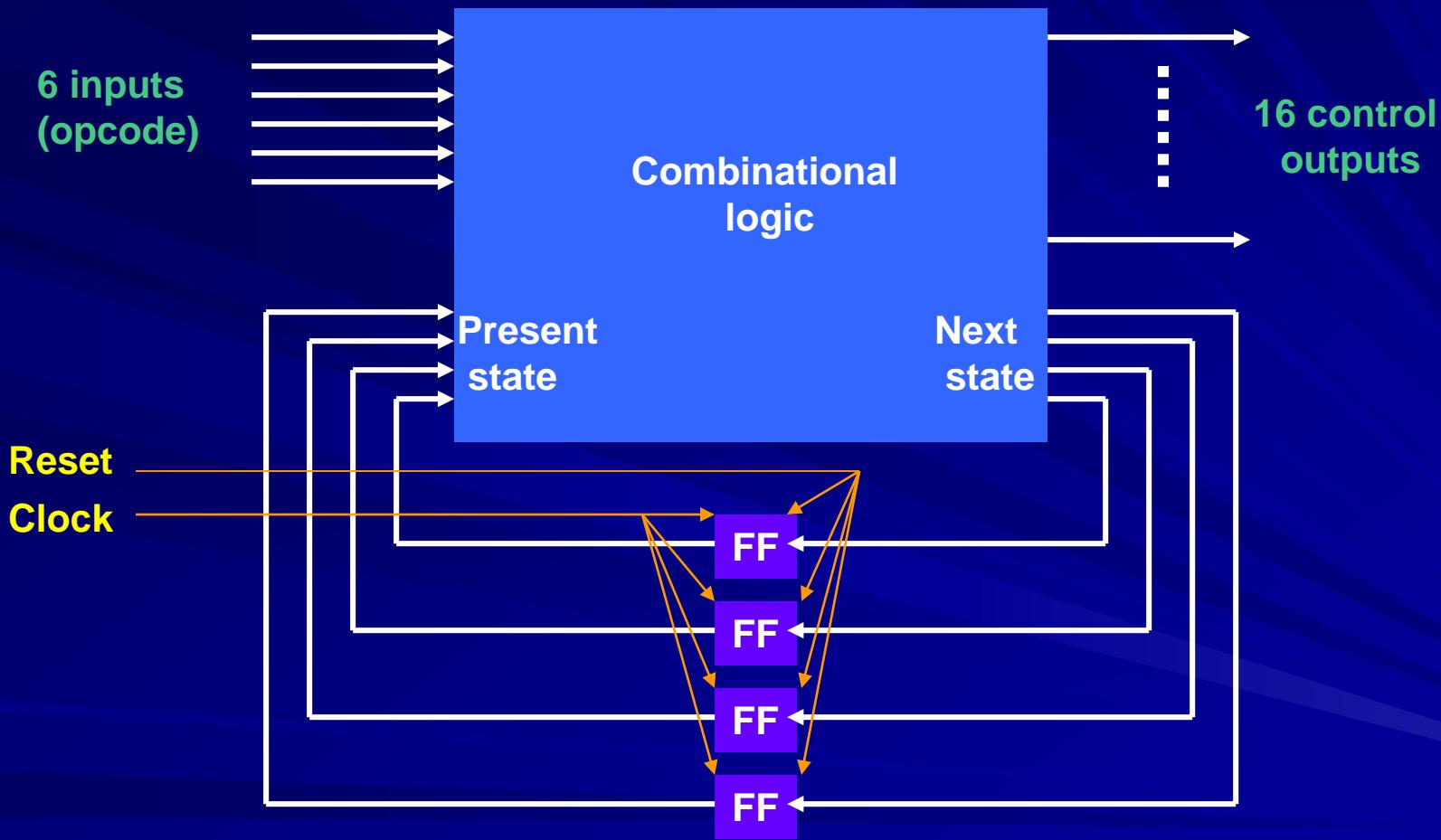
FSM-J (Jump)



Full Control FSM



Control FSM (Controller)



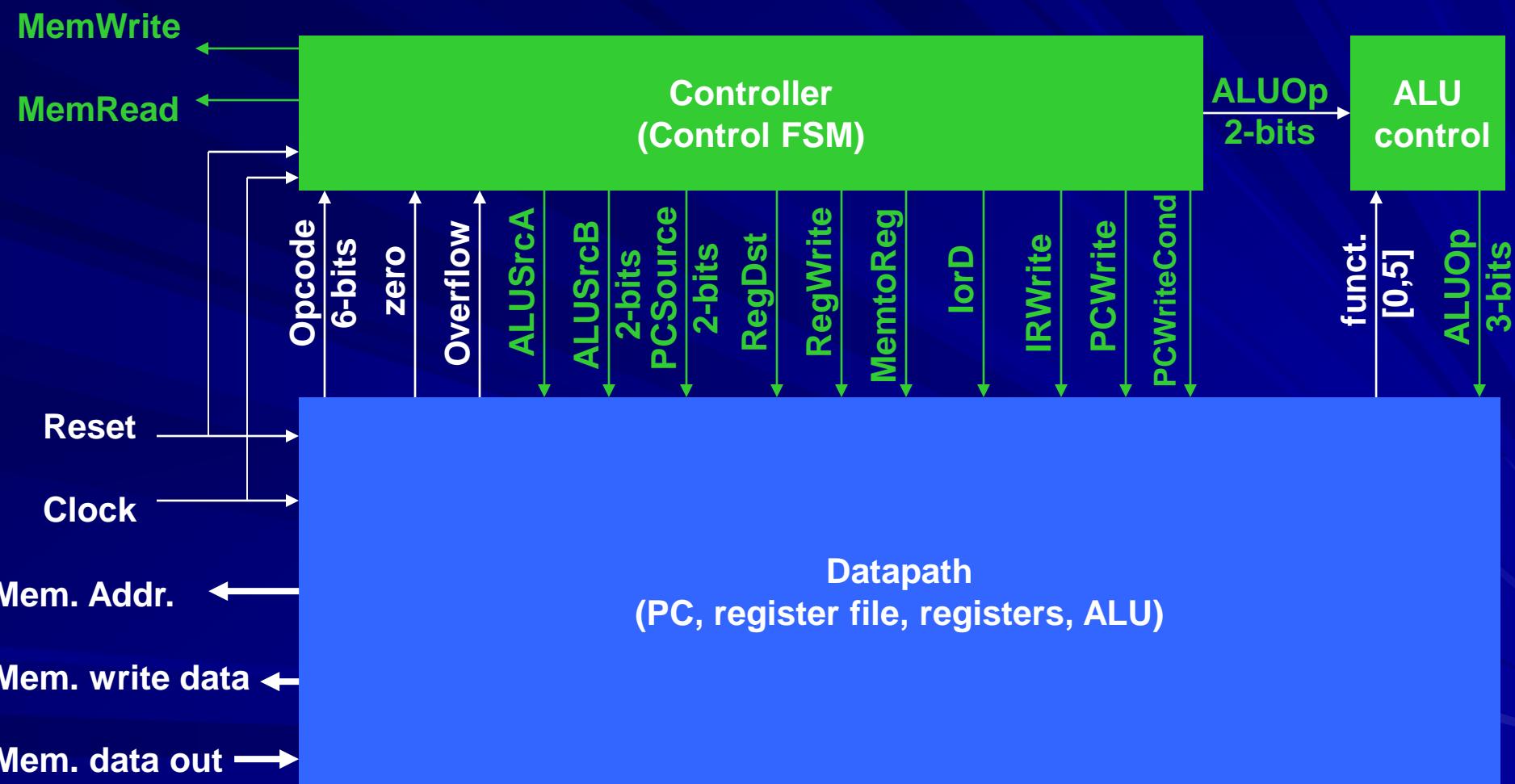
Designing the Control FSM

- Encode states; need 4 bits for 10 states, e.g.,
 - State 0 is 0000, state 1 is 0001, and so on.
- Write a truth table for combinational logic:

Opcode	Present state	Control signals	Next state
000000	0000	0001000110000100	0001
.....

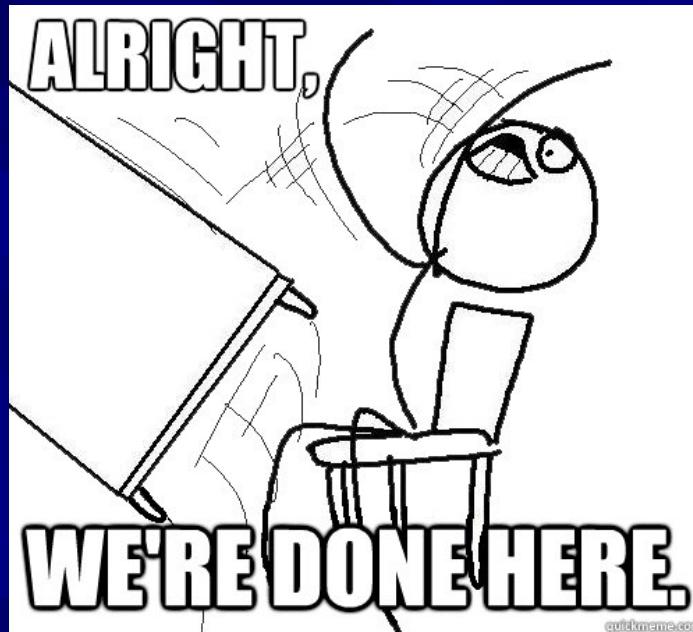
- Synthesize a logic circuit from the truth table.
- Connect four flip-flops between the next state outputs and present state inputs.

Block Diagram of a Processor



So... We're Done, Right?

- We have a datapath and control. We've built our CPU!



Not so fast! What if something goes wrong?

Exceptions and Interrupts

■ Exception

- Any **unexpected** change in the control flow (can be internal or external)
- Synchronous (happens at the same place every time)

■ Interrupt

- Externally caused change in the control flow
- Asynchronous (ex. I/O device)

Event Type	Source	Classification
I/O dev request	External	Interrupt
Arith overflow	Internal	Exception
Bad opcode	Internal	Exception
H/W malfunc	Either	Either

(See sec. 4.9 in your text)3

Exceptions in MIPS

■ When an exception occurs we must:

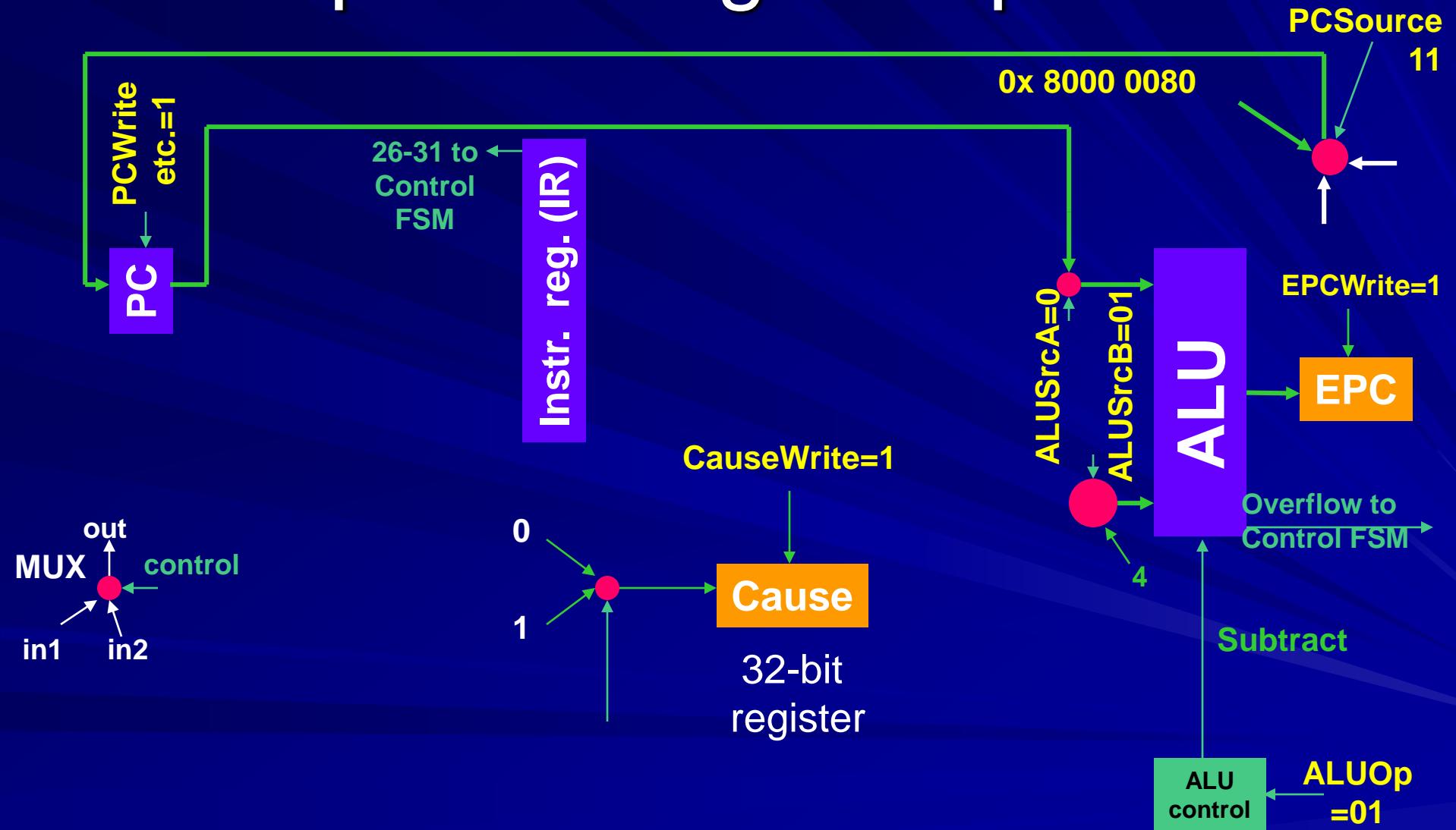
- Figure out what went wrong
- Pass control to an exception handler subroutine
- Fix the problem
- Rerun the instruction that caused the exception

■ Special registers help implement exceptions:

- Exception Program Counter (EPC): Holds the affected instruction address (similar to \$ra register).
- Cause Register: 32-bit register holds an encoded exception type. For example,

0000	0001	0010	0011
Arith Overflow	Bad Opcode	System Call	Bus Error

Implementing Exceptions



How Long Does It Take? Again

- Assume control logic is fast and does not affect the critical timing. Major time components are ALU, memory read/write, and register read/write.
- Time for hardware operations, suppose
 - Memory read or write 2ns
 - Register read 1ns
 - ALU operation 2ns
 - Register write 1ns

Single-Cycle Datapath

- R-type 6ns
- Load word (I-type) 8ns
- Store word (I-type) 7ns
- Branch on equal (I-type) 5ns
- Jump (J-type) 2ns
- Clock cycle time = 8ns
- Each instruction takes *one* cycle

Performance Parameters

- Average cycles per instruction (CPI)
- Cycle time (clock period, T)
- Execution time of a program
- For single-cycle datapath:
 - CPI = 1
 - T = hardware time for lw instruction
 - Execution time of a program
= $T \times CPI \times (\text{Total instructions executed})$

Multicycle Datapath

- Clock cycle time is determined by the longest operation, ALU or memory:
 - Clock cycle time = 2ns
- Cycles per instruction:

■ lw	5	(10ns)
■ sw	4	(8ns)
■ R-type	4	(8ns)
■ beq	3	(6ns)
■ j	3	(6ns)

CPI of a Multicycle Computer

$$\text{CPI} = \frac{\sum_k (\text{Instructions of type } k) \times \text{CPI}_k}{\sum_k (\text{instructions of type } k)}$$

where

CPI_k = Cycles for instruction of type k

Note: CPI is dependent on the instruction mix of the program being run. Standard benchmark programs are used for specifying the performance of CPUs.

Example

- Consider a program containing:

■ loads	25%
■ stores	10%
■ branches	11%
■ jumps	2%
■ Arithmetic	52%

- CPI $= 0.25 \times 5 + 0.10 \times 4 + 0.11 \times 3 + 0.02 \times 3 + 0.52 \times 4$
 $= 4.12$ for multicycle datapath
- CPI $= 1.00$ for single-cycle datapath

Multicycle vs. Single-Cycle

$$\begin{aligned}\text{Performance ratio} &= \frac{\text{Single cycle time} / \text{Multicycle time}}{\frac{(\text{CPI} \times \text{cycle time}) \text{ for single-cycle}}{(\text{CPI} \times \text{cycle time}) \text{ for multicycle}}} \\ &= \frac{1.00 \times 8\text{ns}}{4.12 \times 2\text{ns}} = 0.97\end{aligned}$$

Single cycle is faster in this case, but is it always so?

Consider Another Example

- For this program:

■ loads	5%
■ stores	5%
■ branches	30%
■ jumps	10%
■ Arithmetic	50%

- CPI $= 0.05 \times 5 + 0.05 \times 4 + 0.30 \times 3 +$
 $0.10 \times 3 + 0.50 \times 4$
 $= 3.65$ for multicycle datapath
- CPI $= 1.00$ for single-cycle datapath

Multicycle vs. Single-Cycle

$$\begin{aligned}\text{Performance ratio} &= \frac{\text{Single cycle time} / \text{Multicycle time}}{\frac{(\text{CPI} \times \text{cycle time}) \text{ for single-cycle}}{(\text{CPI} \times \text{cycle time}) \text{ for multicycle}}} \\ &= \frac{1.00 \times 8\text{ns}}{3.65 \times 2\text{ns}} = 1.096\end{aligned}$$

Multicycle is faster... in this case. The performance ratio depends on the instruction mix. Can we do better?

Next: Pipelining



<https://www.youtube.com/watch?v=IjarLbD9r30>

<https://www.youtube.com/watch?v=ANXGJe6i3G8>

<https://www.youtube.com/watch?v=5Ip4EbfPAtI>