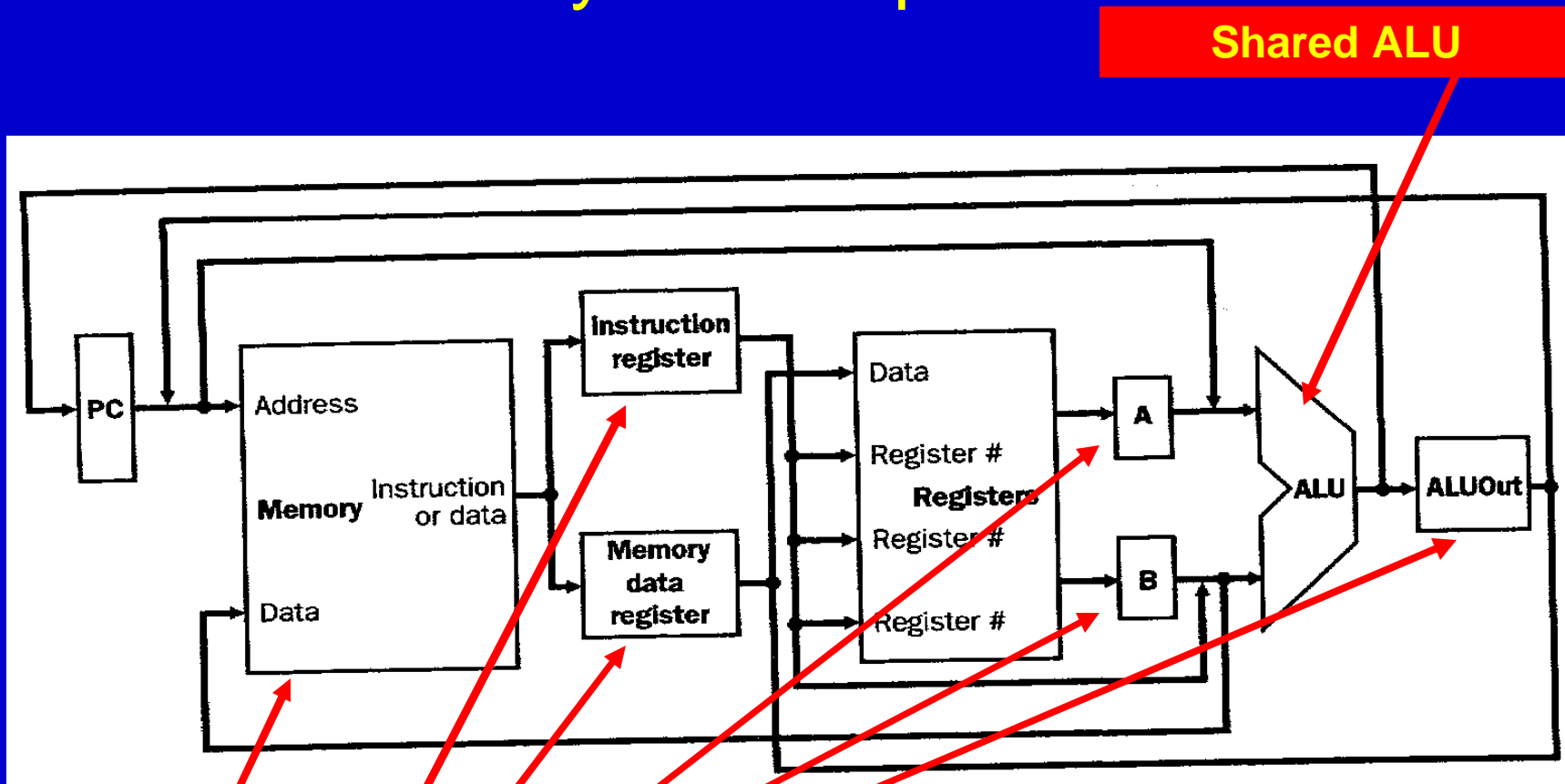


Multicycle Implementation

- Each step in instruction execution takes 1 clock cycle
- Advantages:
 - Functional units like ALU can be used more than once per instruction minimizing hardware
 - Instructions may take varying numbers of clock cycles to complete (efficiency)
- Major differences from single-cycle design:
 - Single memory for instructions and data
 - Single ALU
 - Additional registers (state elements) for output between steps

Multicycle Datapath



Shared ALU

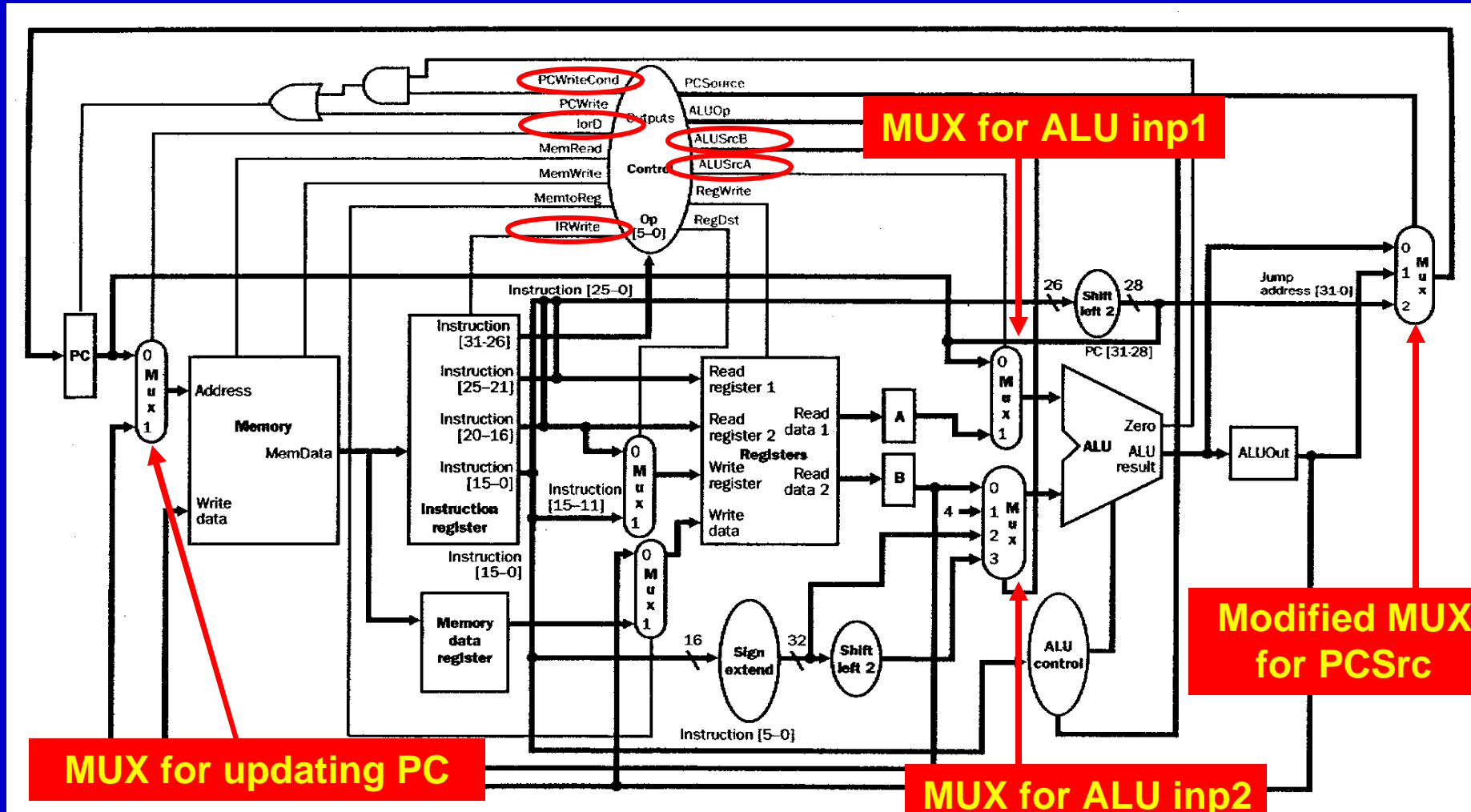
Additional registers

Combined memory unit

Multicycle Implementation

- Subsequent clock cycles
 - Data needed in a following clock cycle can be stored in the additional temporary registers (state elements)
- Subsequent instructions
 - Data needed in a following instruction is stored in a programmer visible state element (register file, memory, etc.)
- Control
 - Additional mux's and modifications to control will be necessary to create correct multicycle implementation
- Cycle length
 - Can accommodate at most one of: memory access, register file access, ALU operation

Complete Multicycle Implementation (First look)



Steps of an instruction

- Assumptions
 - No step can contain more than one ALU operation
 - No step can contain more than one register file access
 - No step can contain more than one memory access
 - Temp registers allow very fast access within cycle
- Conclusions
 - A cycle must be of sufficient length to perform longest of ALU op, reg/mem access
 - Required that any data needed in subsequent cycle stored in temp register or register/memory prior to end of step
 - Cycle must be long enough for step to fully complete
 - Must allow multiple steps to complete independent of one another

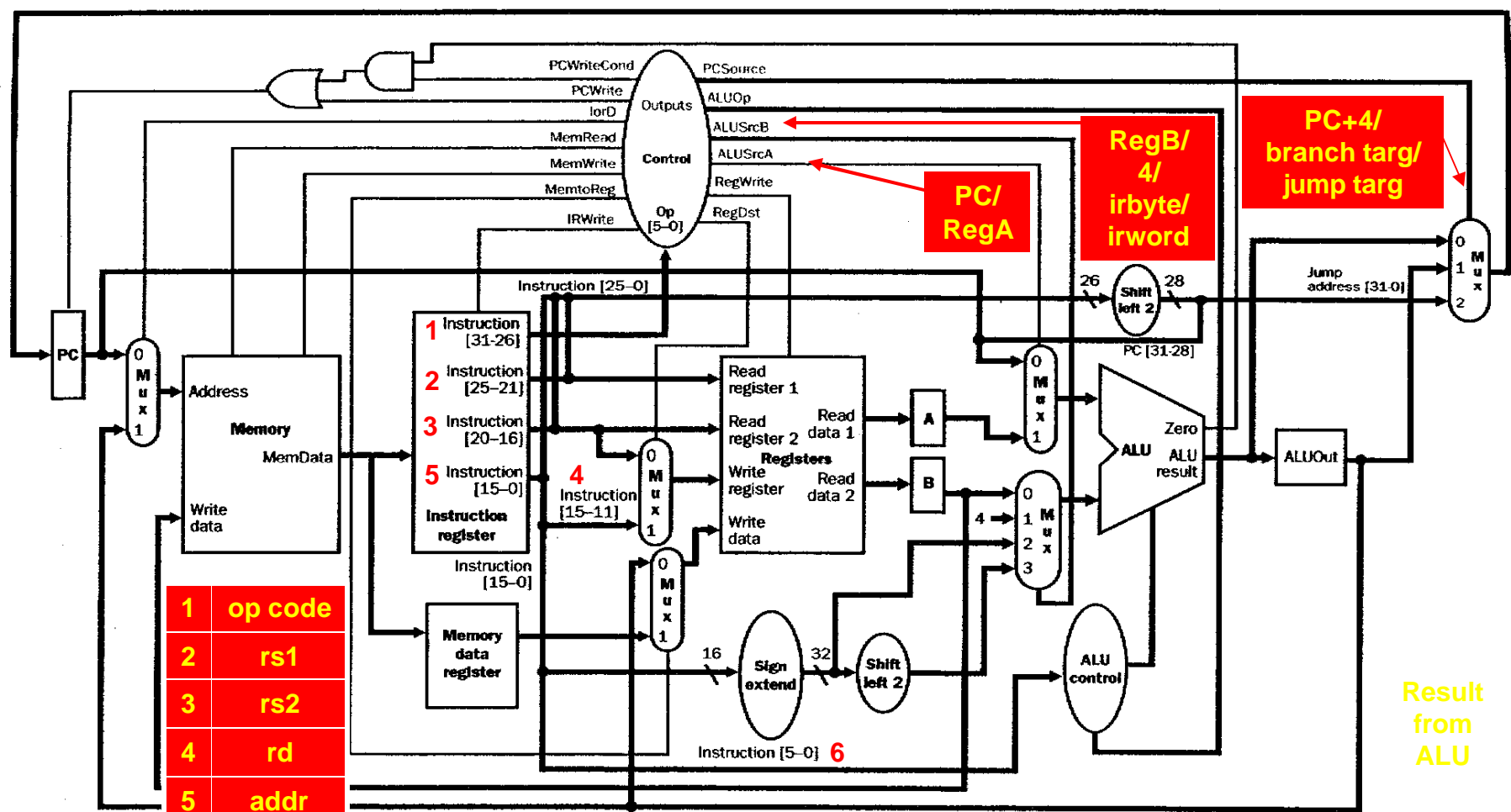
Steps of an instruction

Single-cycle Instruction execution

- R-type
 - 1 – Fetch instruction and increment PC
 - 2 – Read two source registers from register file (set control)
 - 3 – Perform ALU operation on register operands
 - 4 Write result to reg
- Store/Load
 - 1 – Fetch instruction and increment PC
 - 2 – Read one register value from register file (set control)
 - 3 – ALU computes sum of immediate and register value
 - 4 – ALU result used as address to memory
 - 5 – Data from memory written back to register (Load only)
- Branch
 - 1 – Fetch instruction and increment PC
 - 2 – Read two source registers from register file (set control)
 - 3 – ALU performs subtract on register operands (target addr computed)
 - Zero result from ALU determines PC

	R-type	Store/Load	Branch/Jump
1	Instruction fetch		
2	Instruction decode and register fetch		
3	Execution	Compute addr	Instr completion
4	Instr completion	Mem access	
5		Load completion	

Complete Multicycle implementation



1	op code
2	rs1
3	rs2
4	rd
5	addr
6	funct

Result from ALU

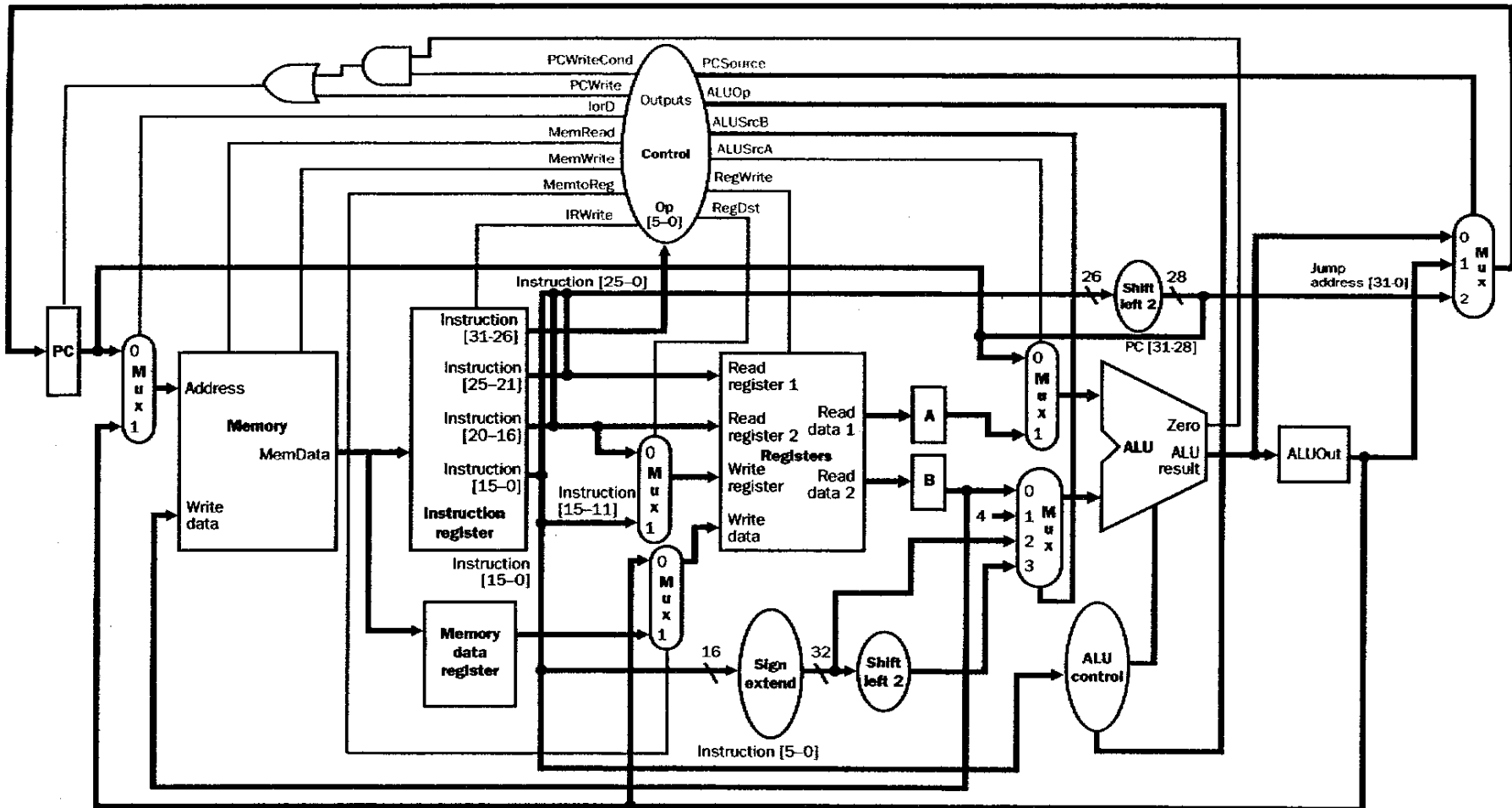
1-bit Control Signals

Signal	Deasserted (0)	Asserted (1)
RegDst(0/1mux)	Rs2 is WriteReg	Rd is WriteReg
RegWrite	No effect	Enable write to reg file
ALUSrcA(0/1mux)	Operand1 of ALU ← PC	Operand1 of ALU ← RegA
MemRead	No effect	Enable read from mem
MemWrite	No effect	Enable write to mem
MemtoReg(0/1mux)	Local Reg ← ALU result	Local Reg ← MDR
lrd(0/1mux)	Addr to mem ← PC	Addr to mem ← ALU result
IRWrite	No effect	Enable IR ← MemData
PCWrite	No effect	Enable PC ← new PC
PCWriteCond	No effect	Enable PC ← new PC on zero

2-bit Control Signals

Signal	Value	Effect
ALUOp	00	ALU add
	01	ALU sub
	10	ALU op determined by func
ALUSrcB	00	ALU Operand2 from RegB
	01	ALU Operand2 is 4
	10	ALU Operand2 is sign extended lower 16 bits of IR
	11	ALU Operand2 is sign extended lower 16 bits of IR * 4
PCSource	00	$PC \leftarrow PC + 4$ on PCWrite
	01	$PC \leftarrow$ ALU result on PCWrite
	10	$PC \leftarrow$ jump target address on PCWrite

Complete Multicycle implementation



Detailed Step 1

- Instruction Fetch Step
 - Fetch instructions from memory $IR = \text{memory}[PC]$
 - Compute address of next instruction ($PC+4$)

Signal	Value	Action
ALUSrcA(0/1mux)	0	Operand1 of ALU $\leftarrow PC$
MemRead	1	Enable read from mem
lorD(0/1mux)	0	Addr to mem $\leftarrow PC$
IRWrite	1	Enable IR $\leftarrow \text{MemData}$
PCWrite	1	Enable PC $\leftarrow \text{new PC}$
ALUOp	00	ALU add
ALUSrcB	01	ALU Operand2 is 4
PCSource	00	PC $\leftarrow PC+4$ on PCWrite

If signal not listed, assume disabled

Detailed Step 2

- Instruction Decode and register fetch
 - Instruction not known
 - Read registers rs1 and rs2 from register file (every cycle)
 - Store in RegA and RegB (A=Reg[IR[25-21] B=Reg[IR[20-16]])
 - Compute branch target address in case it is used later
 - $ALUOut = PC + (\text{sign extended}(IR[15-0]) \ll 2)$

Signal	Value	Action
ALUSrcA(0/1mux)	0	Operand1 of ALU ← PC
ALUOp	00	ALU add
ALUSrcB	11	ALU Operand2 is sign extended lower 16 bits of IR * 4

Detailed Step 3

- Memory Reference Instruction
 - $ALUOut = A + \text{sign extended}(IR[15-0])$

Signal	Value	Action
ALUSrcA(0/1mux)	1	Operand1 of ALU ← RegA
ALUOp	00	ALU add
ALUSrcB	10	ALU Operand2 is sign extended lower 16 bits of IR

- Arithmetic/Logical Instruction
 - $ALUOut = A \text{ op } B$

Signal	Value	Action
ALUSrcA(0/1mux)	1	Operand1 of ALU ← RegA
ALUOp	10	ALU op determined by func
ALUSrcB	00	ALU Operand2 from RegB

Detailed Step 3

- Branch Instruction
 - If (A == B) PC = ALUOut

Signal	Value	Action
ALUSrcA(0/1mux)	1	Operand1 of ALU ← RegA
ALUOp	01	ALU sub
ALUSrcB	00	ALU Operand2 from RegB
PCWriteCond	1	Enable PC ← new PC on zero
PCSource	01	PC ← ALU result on PCWrite

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

Signal	Value	Action
PCSource	01	PC ← ALU result on PCWrite
PCWrite	1	Enable PC ← new PC

Detailed Step 4

- Memory Reference (load)
 - $MDR = \text{Memory}[ALUOut]$

Signal	Value	Action
MemRead	1	Enable read from mem
lorD(0/1 mux)	1	Addr to mem \leftarrow ALU result

- Memory Reference (store)
 - $\text{Memory}[ALUOut] = B$

Signal	Value	Action
MemWrite	1	Enable write to mem
lorD(0/1 mux)	1	Addr to mem \leftarrow ALU result

Detailed Step 4

- Arithmetic/logical instruction (R-type)
 - $\text{Reg}[\text{IR}[15-11]] = \text{ALUOut}$

Signal	Value	Action
RegDst(0/1mux)	1	Rd is WriteReg
RegWrite	1	Enable write to reg file
MemtoReg(0/1mux)	0	Local Reg ← ALU result

Detailed Step 5

- Memory Read completion step (loads)
 - Reg[IR[20-16]]=MDR

Signal	Value	Action
RegDst(0/1mux)	0	Rs2 is WriteReg
RegWrite	1	Enable write to reg file
MemtoReg(0/1mux)	1	Local Reg ← MDR

Defining Control

- Individual control lines like single cycle will not work!
- Control is more complicated since what we do in one cycle depends on instr + last cycle
- Two methods:
 - FSM
 - Microprogramming

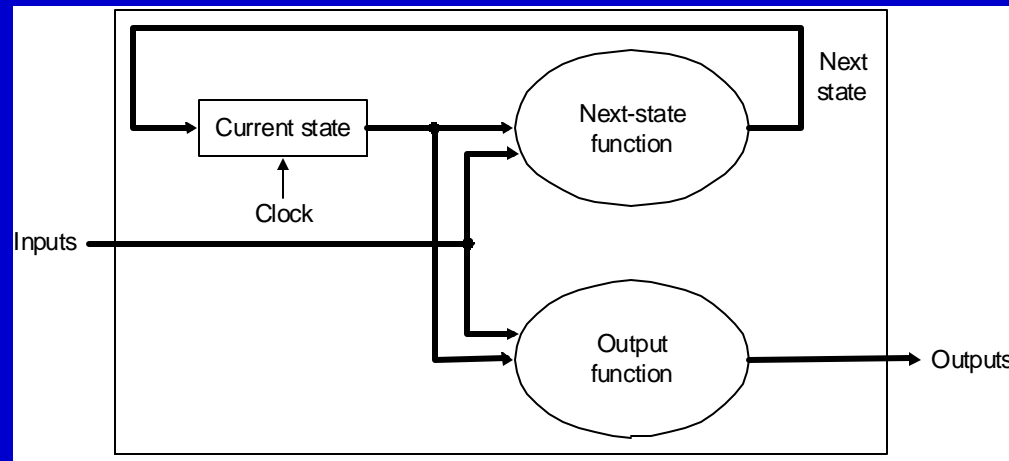
	R-type	Store/Load	Branch/Jump
1	Instruction fetch		
2	Instruction decode and register fetch		
3	Execution	Compute addr	Instr completion
4	Instr completion	Mem access	
5		Load completion	

Implementing the Control

- Sequential logic
 - Stores data in memory elements (current “state” of machine)
 - Behavior determined by input + current state
 - Truth table does not provide enough info to describe a sequential system concisely

Finite State Machines

- Describe using FSM
 - Set of states + two functions (nextstate and output)
- Set of states → all possible values of internal storage
- Nextstate → combination logic given inputs, determines next state
- Output → produces a set of outputs from current states and the inputs



- Synchronous state machines: state change with clock edge, new state computed every clock cycle

Classic Example

- Design FSM that operates a traffic light
- For control:
 - Output depends on current state only (Moore machine)
 - Nextstate combination logic using inputs at current state
- Traffic light:
 - NSEW intersection
 - red and green only (no yellow)
 - .033Hz clock (no faster than 30sec)

Asserted

Deasserted

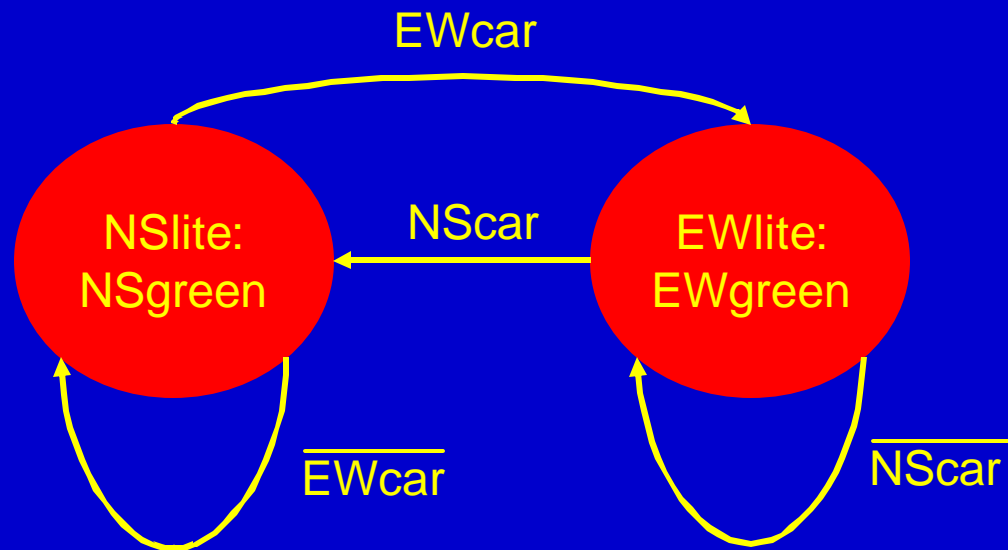
Current State	Outputs	
	NSlite	EWlite
NSGreen	1	0
EWGreen	0	1

Output function

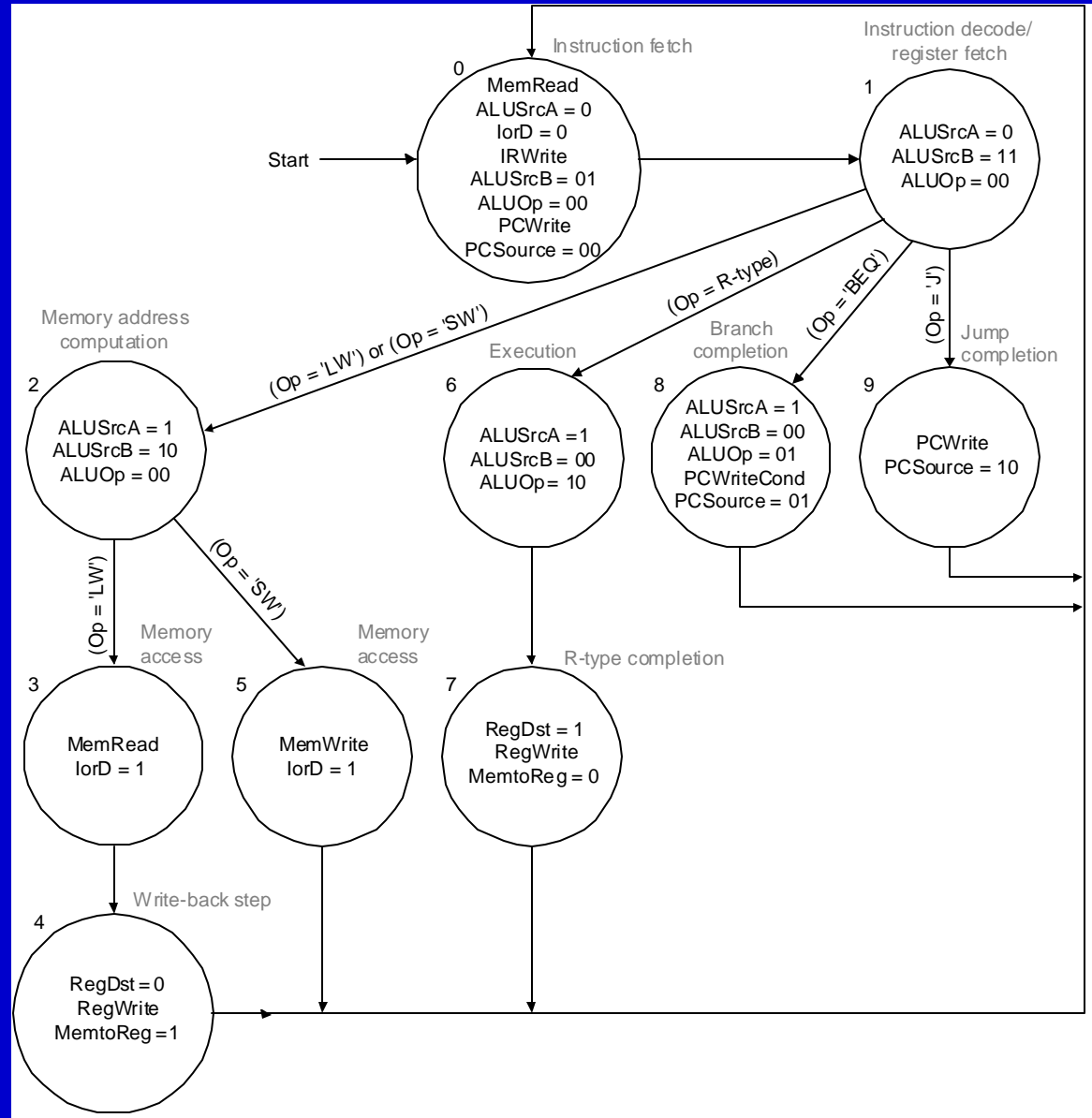
Nextstate function

Current State	Inputs(car waiting)		Next State
	NScar	EWcar	
NSGreen	0	0	NSgreen
	0	1	EWgreen
	1	0	NSgreen
	1	1	EWgreen
EWGreen	0	0	EWgreen
	0	1	EWgreen
	1	0	NSgreen
	1	1	NSgreen

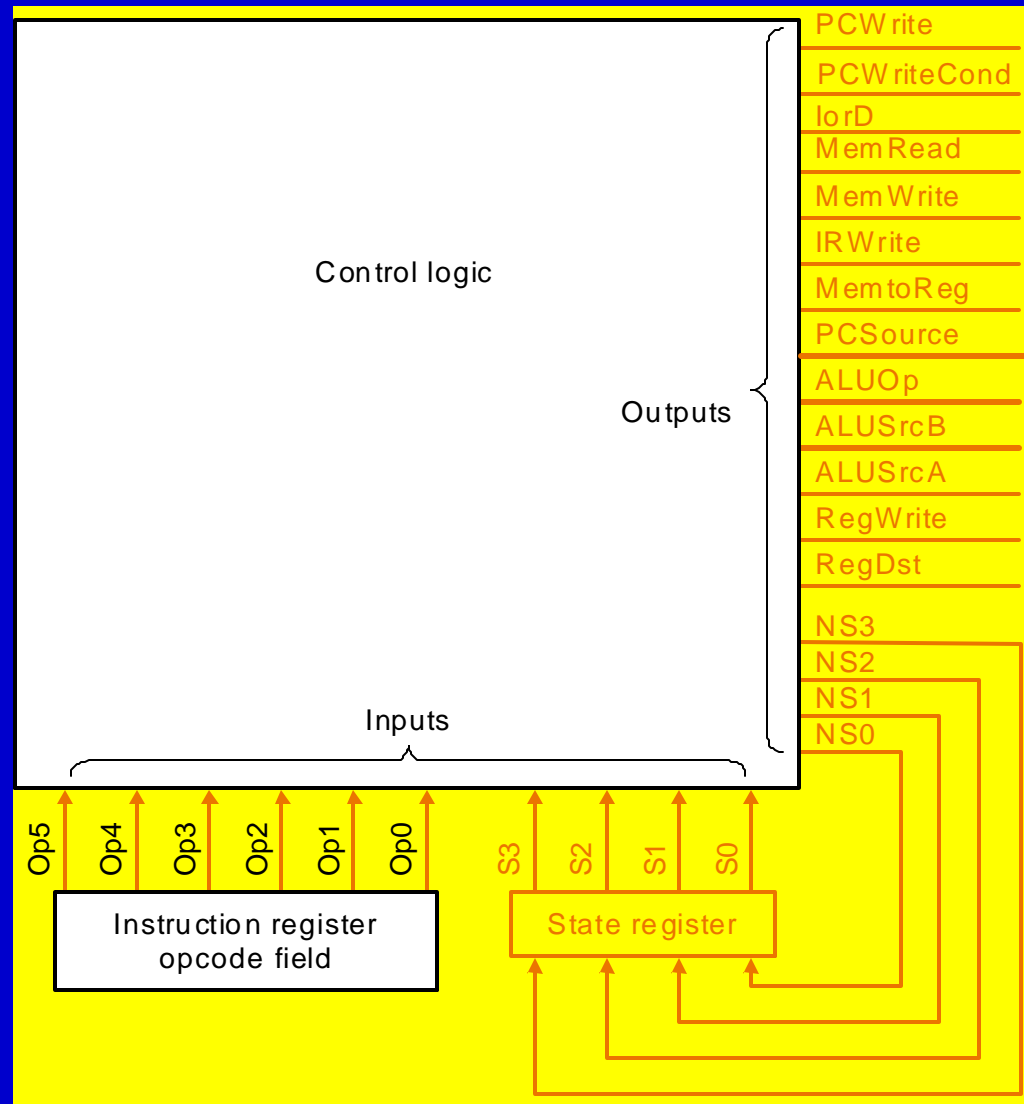
Two state traffic light FSM



FSM for multicycle control



Implementation



PLA Implementation

