Computer Architecture
Ch. 5-4: Designing a Multiple Cycle Controller

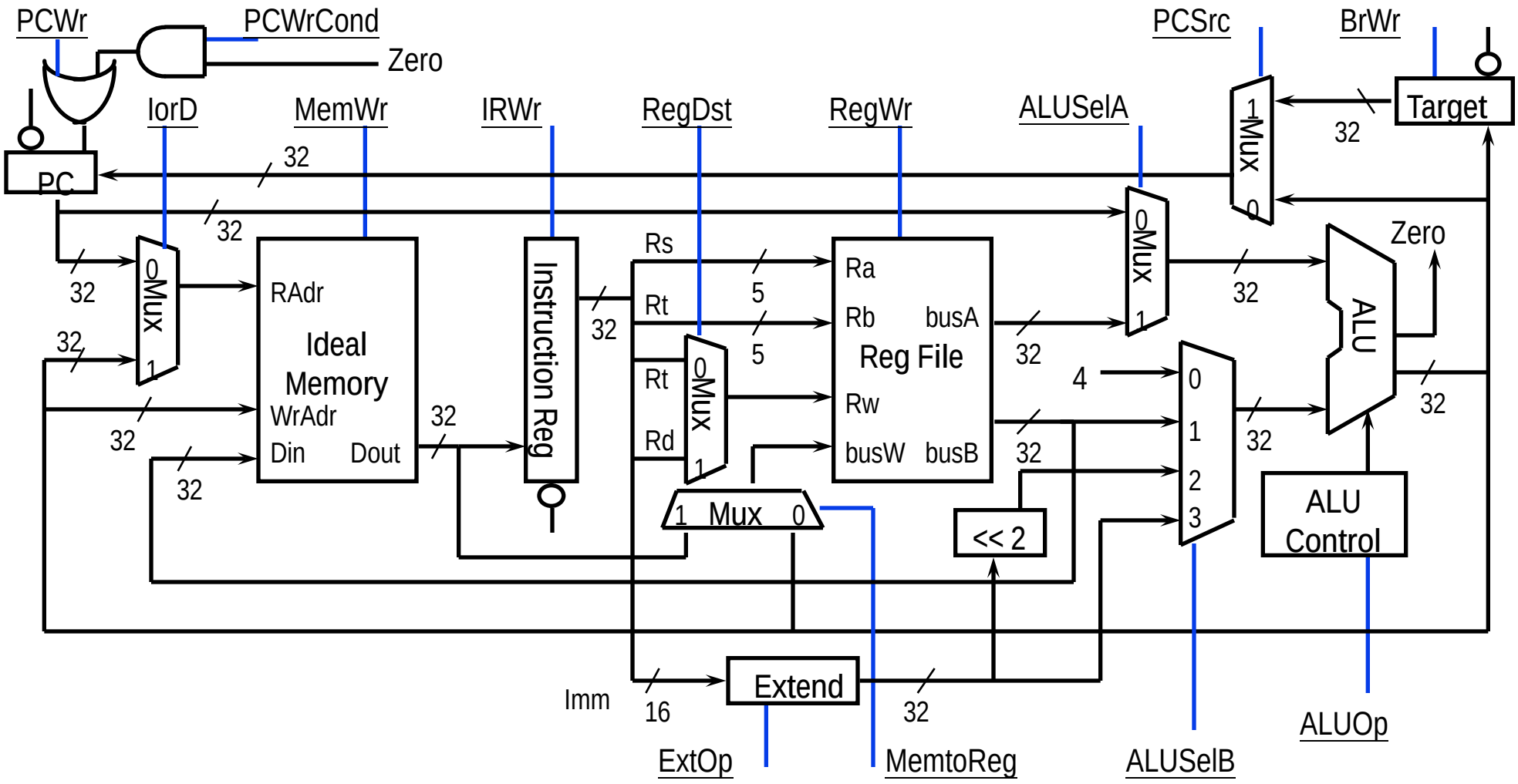
Spring, 2005

Sao-Jie Chen (csj@cc.ee.ntu.edu.tw)

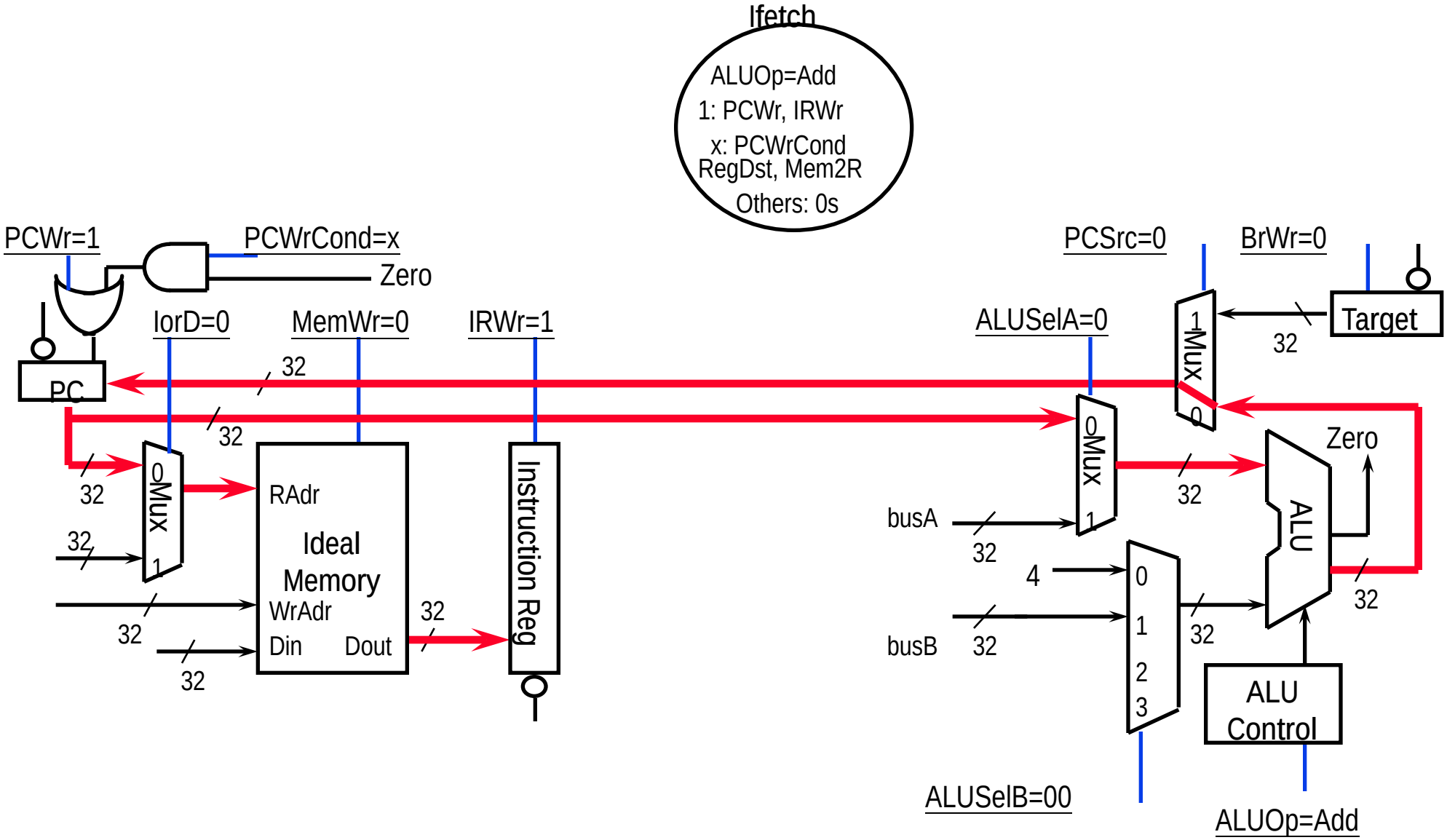
Review of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - ⇒ Cycle time: time it takes to execute the longest step
    - ⇒ Keep all the steps to have similar length
  - This is the essence of the multiple cycle processor
- The advantages of the multiple cycle processor:
  - Cycle time is much shorter
  - Different instructions take different number of cycles to complete
    - ⇒ Load takes five cycles
    - ⇒ Jump only takes three cycles
  - Allows a functional unit to be used more than once per instruction

Putting it all together: Multiple Cycle Datapath



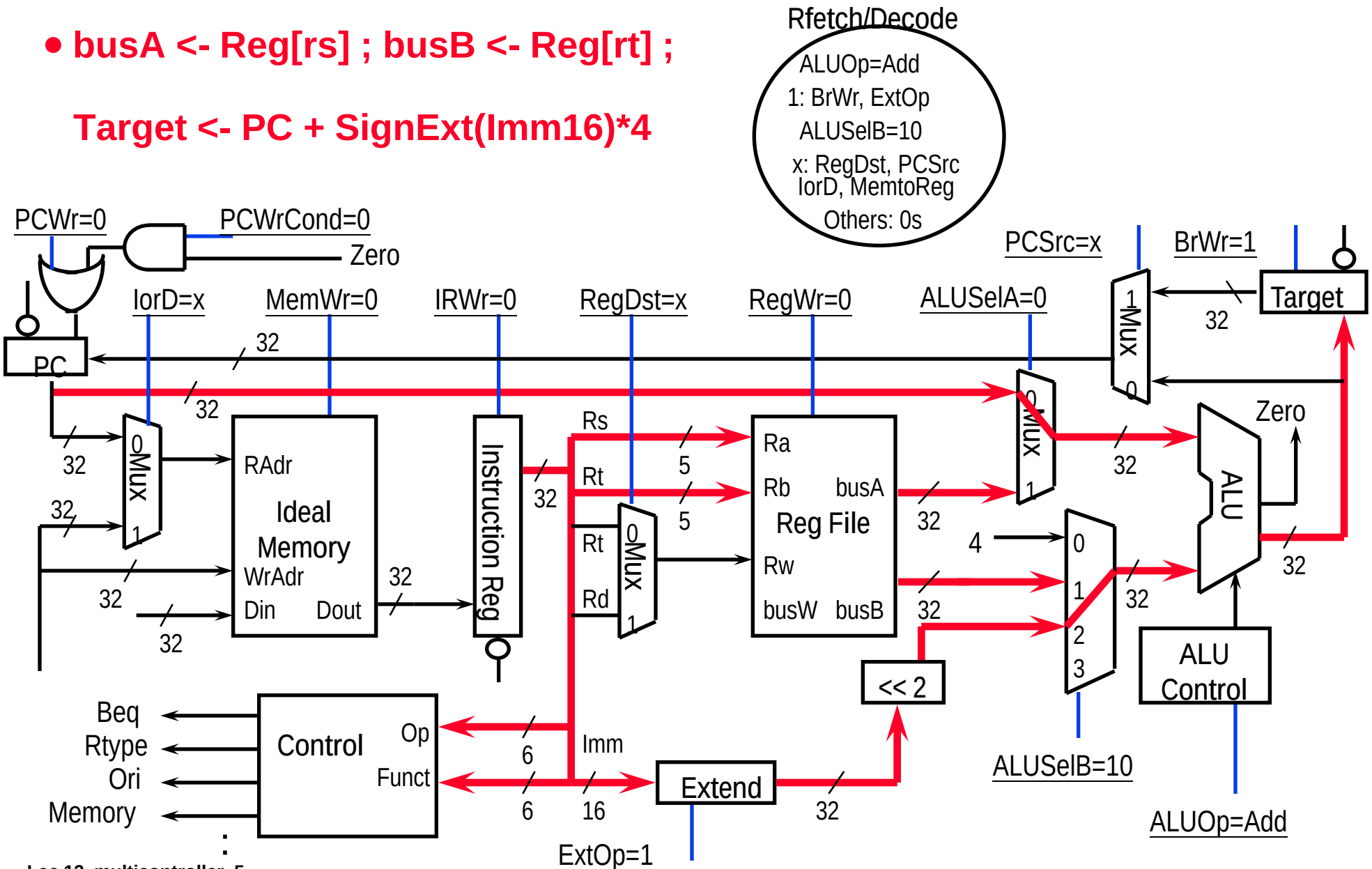
Instruction Fetch Cycle: Overall Picture



Register Fetch / Instruction Decode (Continued)

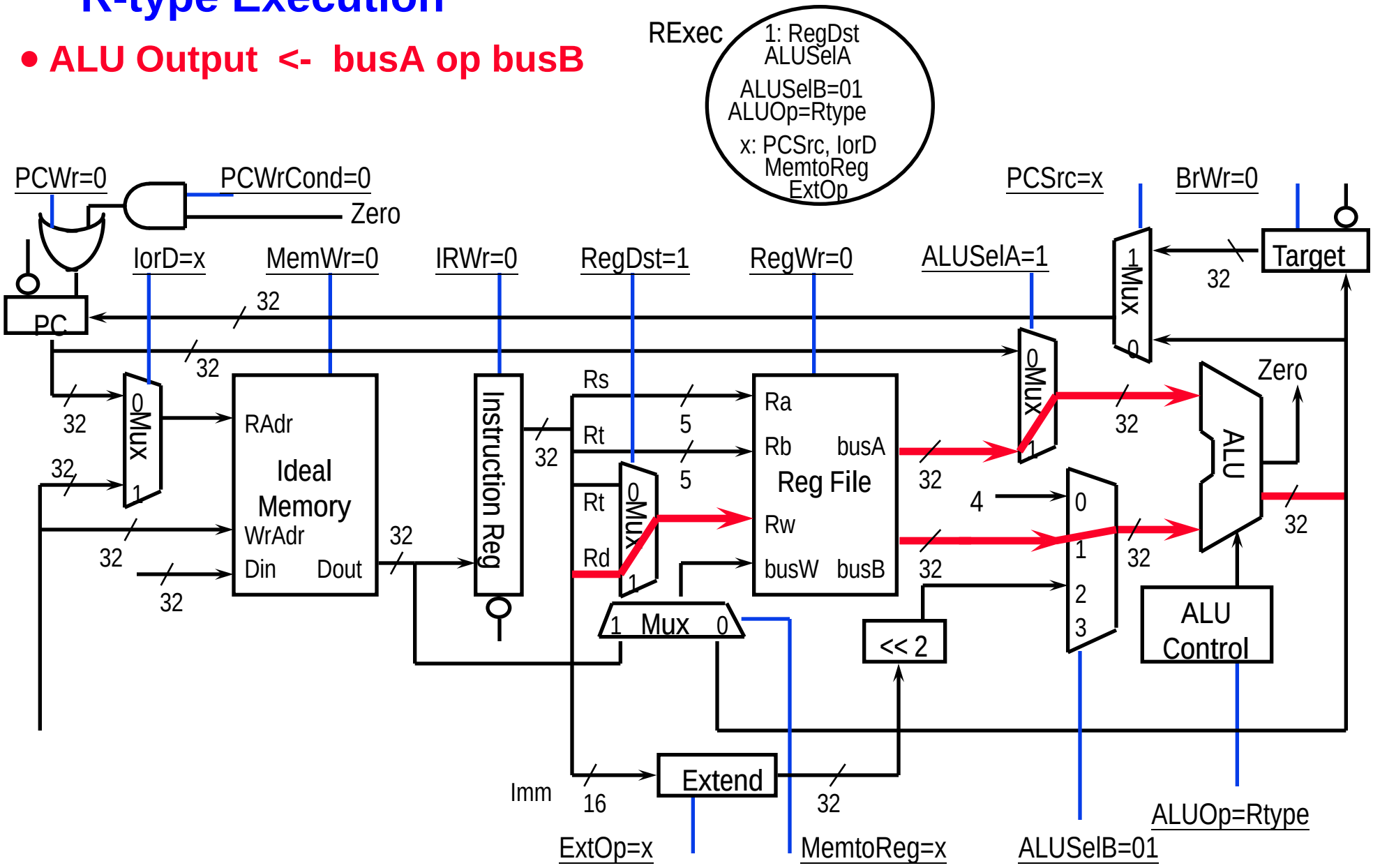
- busA <- Reg[rs] ; busB <- Reg[rt] ;

Target <- PC + SignExt(Imm16)*4



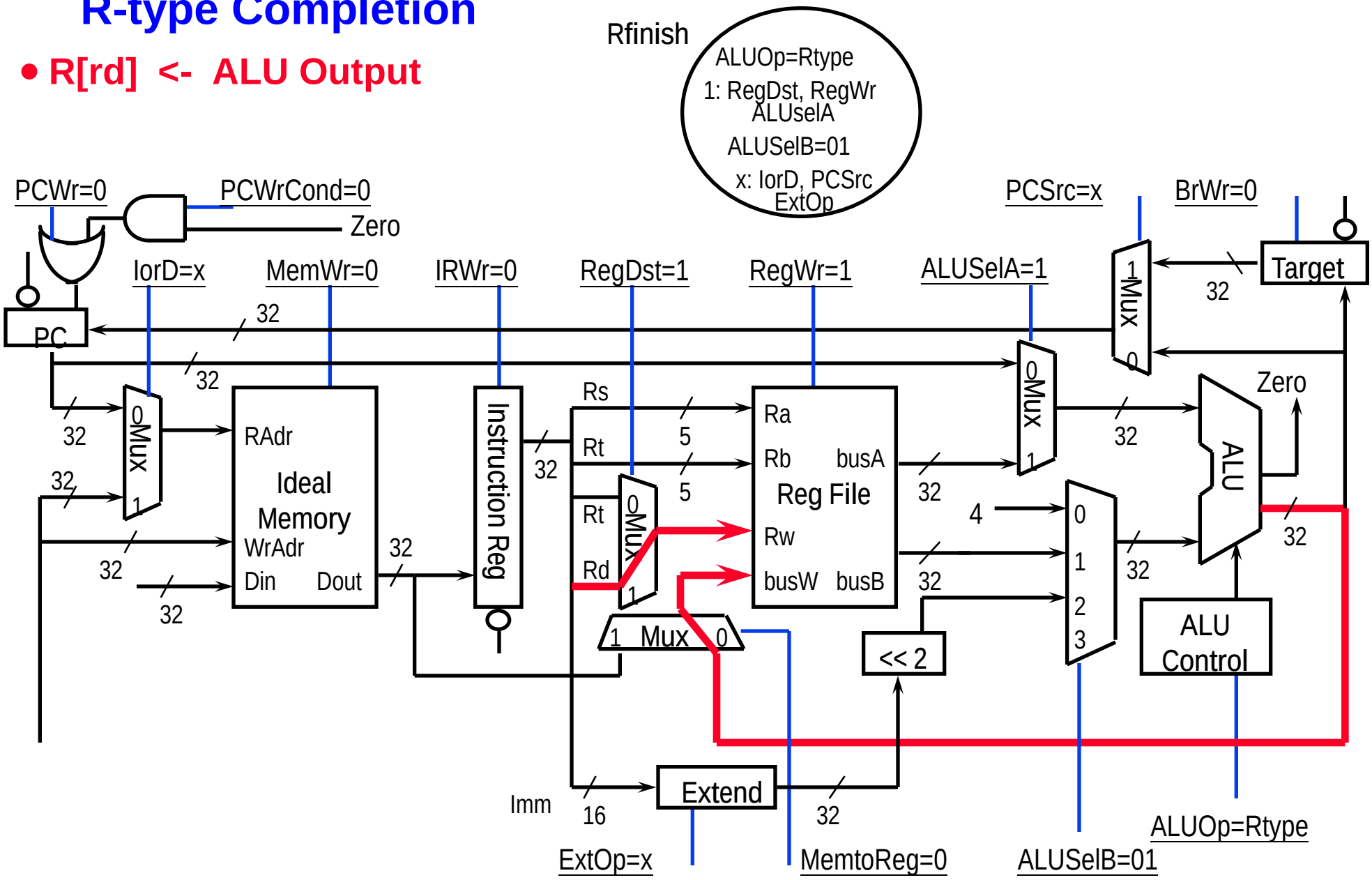
R-type Execution

- **ALU Output <- busA op busB**



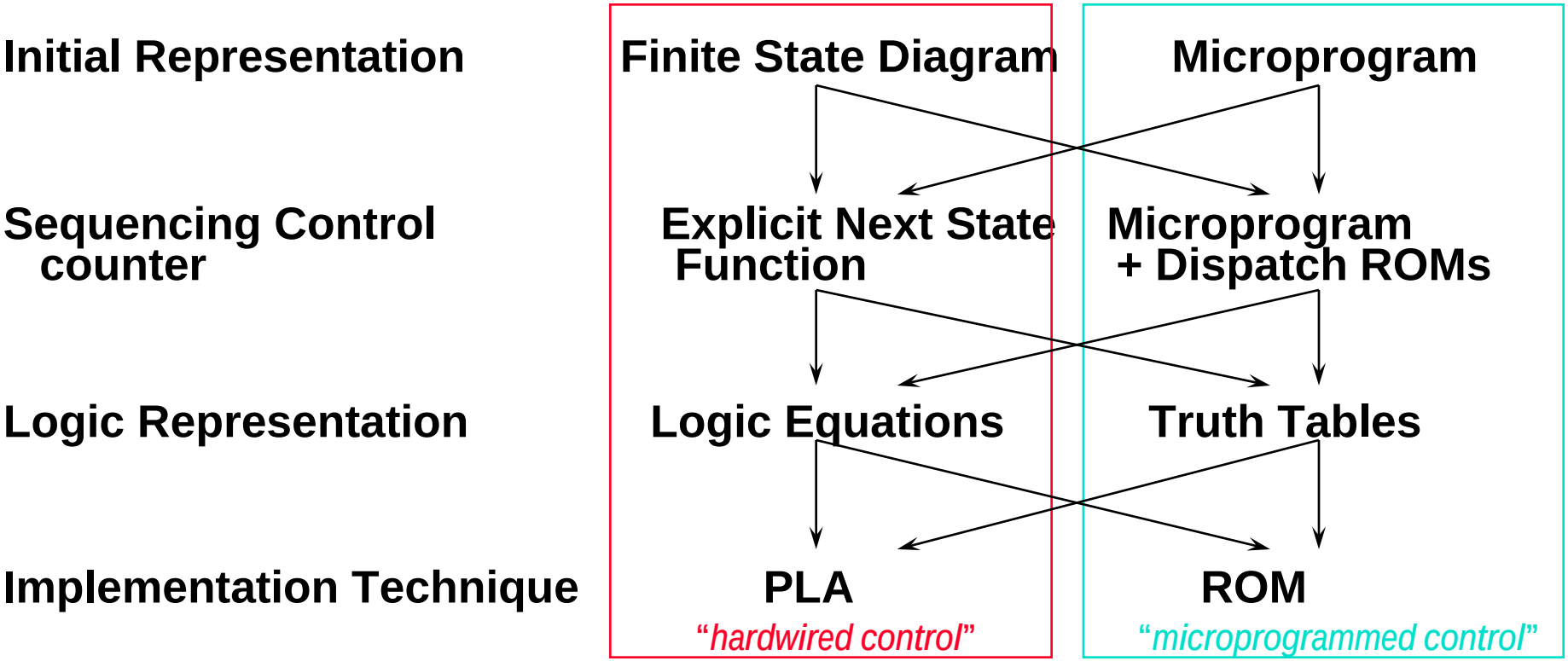
R-type Completion

- R[rd] <- ALU Output

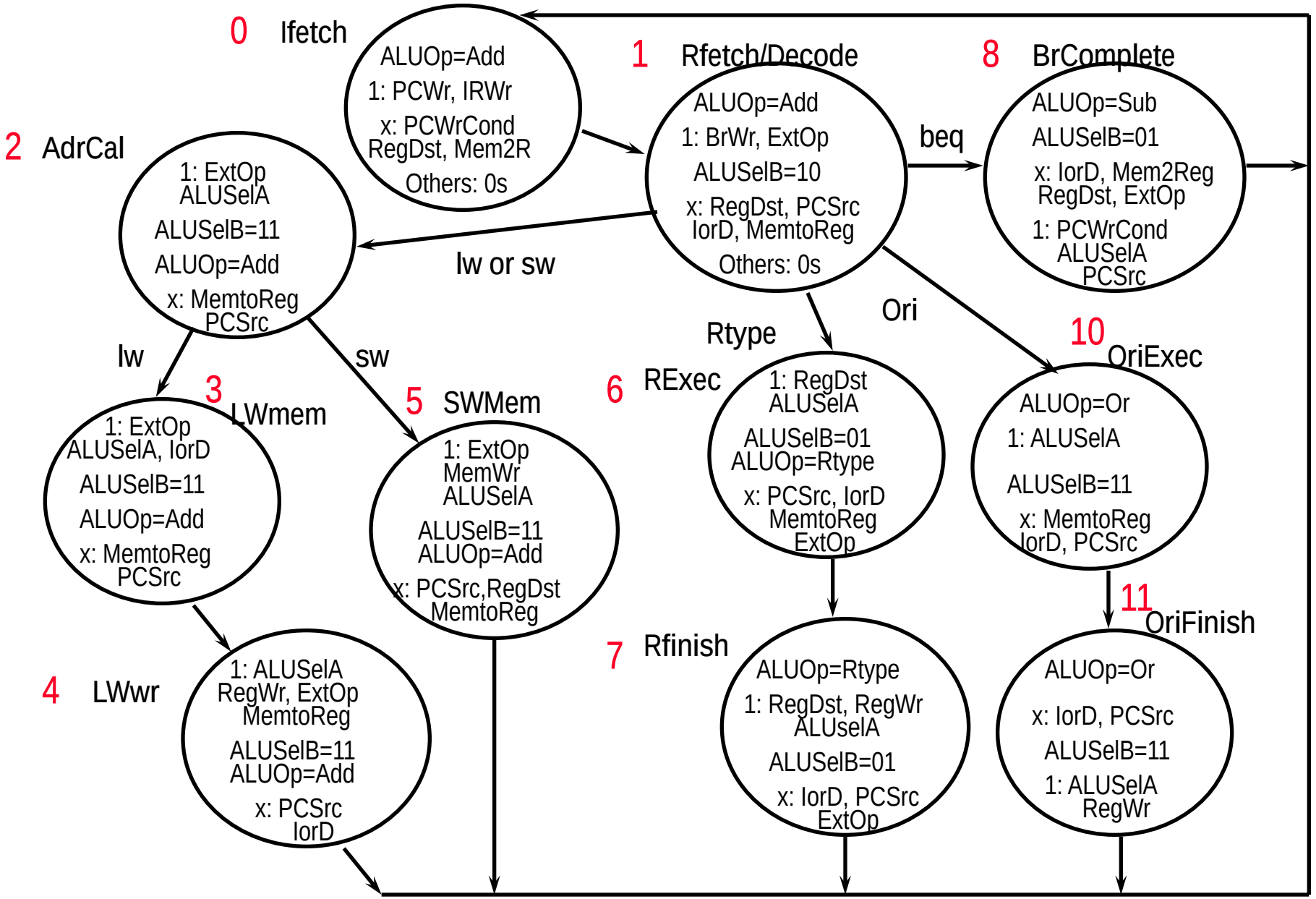


Overview of Next Two Lectures

- Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.

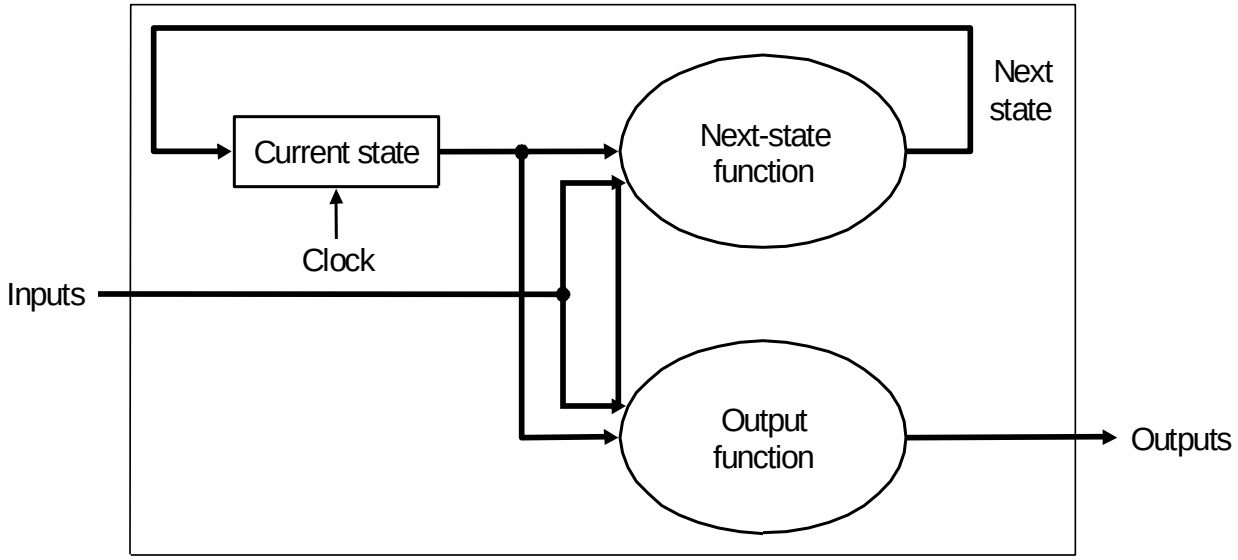


Initial Representation: Finite State Diagram



Review: Finite State Machines

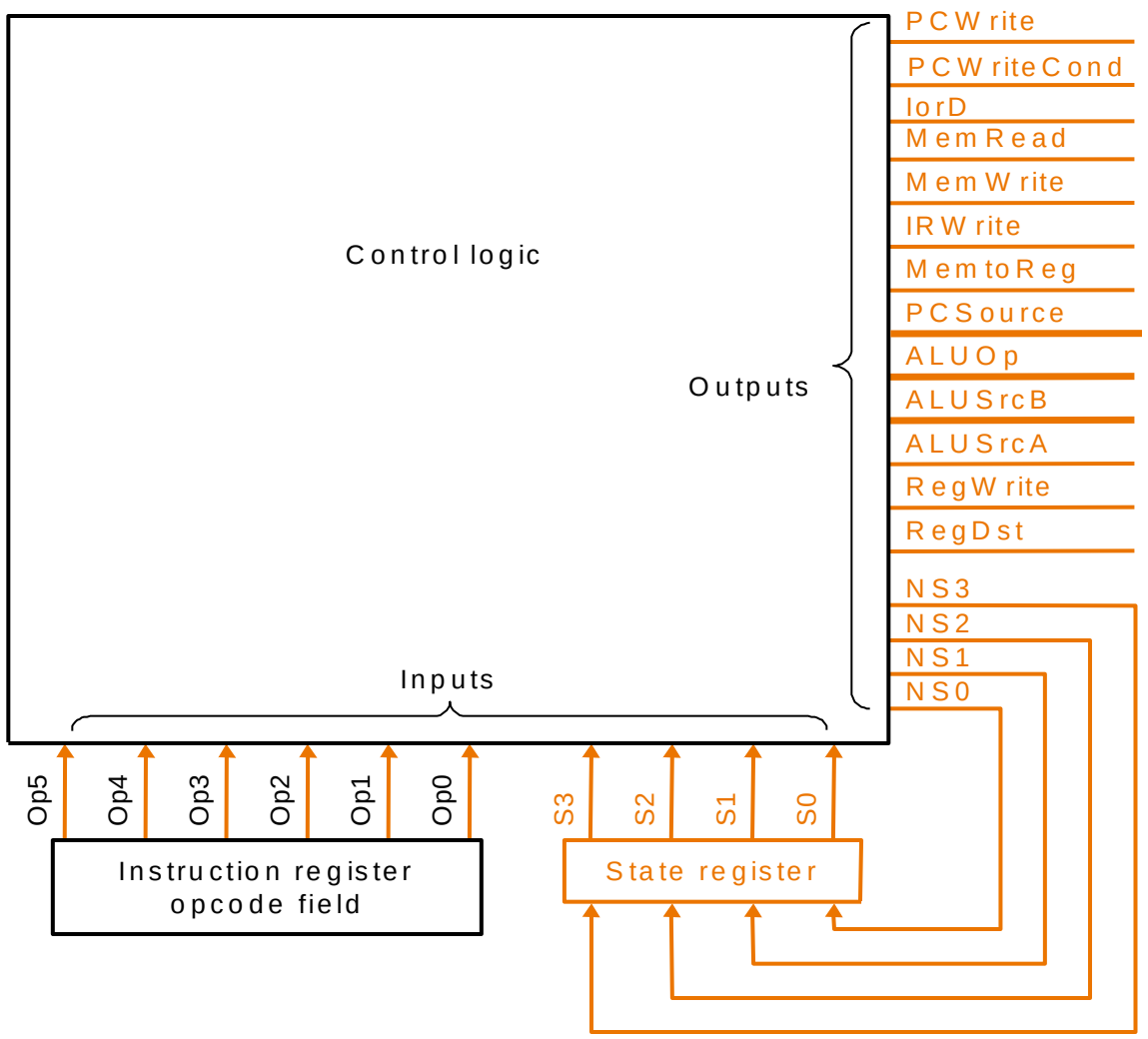
- **Finite state machines:**
  - **a set of states and internal storage**
  - **next state function (determined by current state and the input)**
  - **output function (determined by current state and possibly input)**



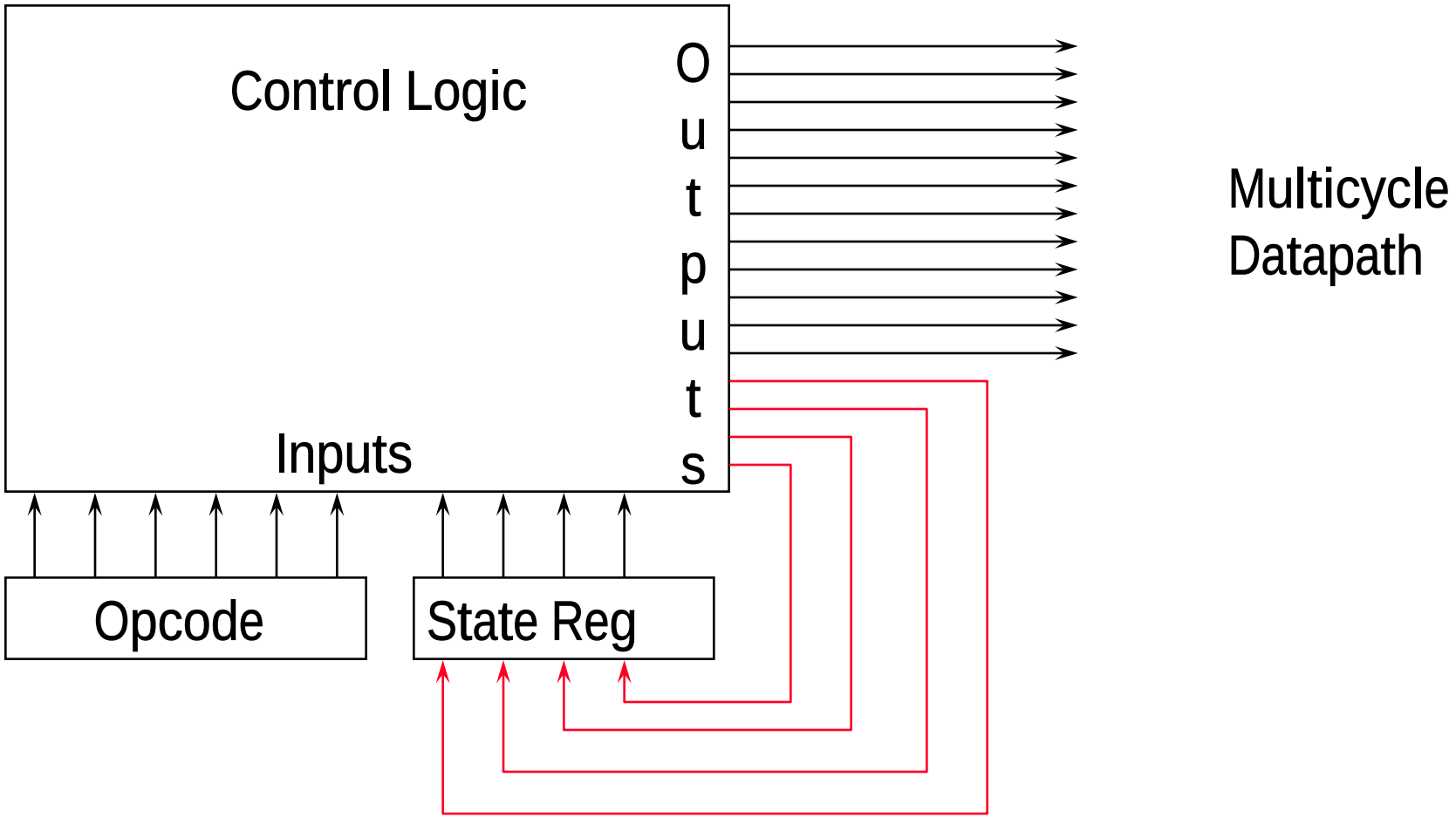
- **We'll use a Moore machine (output based only on current state)**

Finite State Machine for Control

- **Implementation:**



Sequencing Control: Explicit Next State Function



- **Next state number is encoded just like datapath controls**

Logic Representative: Logic Equations

- Next state from current state
  - State 0 -> State1
  - State 1 -> S2,S6,S8,S9,S10
  - State 2 ->_____
  - State 3 ->_____
  - State 4 ->State 0
  - State 5 -> State 0
  - State 6 -> State 7
  - State 7 -> State 0
  - State 8 -> State 0
  - State 9 -> State 0
  - State 10 -> State 11
  - State 11 -> State 0

Alternatively,
prior state & condition

S4, S5, S7, S8, S9, S11 -> State0

_____ -> State 1

_____ -> State 2

_____ -> State 3

_____ -> State 4

State 2 & op = sw -> State 5

_____ -> State 6

State 6 -> State 7

_____ -> State 8

State 1 & op = jmp -> State 9

_____ -> State 10

State 10 -> State 11

Logic Representative: Logic Equations

- Next state from current state
  - State 0 -> State1
  - State 1 -> S2,S6,S8,S9,S10
  - State 2 -> State 3 & State 5
  - State 3 -> State 4
  - State 4 ->State 0
  - State 5 -> State 0
  - State 6 -> State 7
  - State 7 -> State 0
  - State 8 -> State 0
  - State 9 -> State 0
  - State 10 -> State 11
  - State 11 -> State 0

Alternatively,
prior state & condition

S4, S5, S7, S8, S9, S11 -> State0

State 0 -> State 1

State 1 & op = lw | sw -> State 2

State 2 & op = lw -> State 3

State 3 -> State 4

State 2 & op = sw -> State 5

State 1 & op = R-type -> State 6

State 6 -> State 7

State 1 & op = beq -> State 8

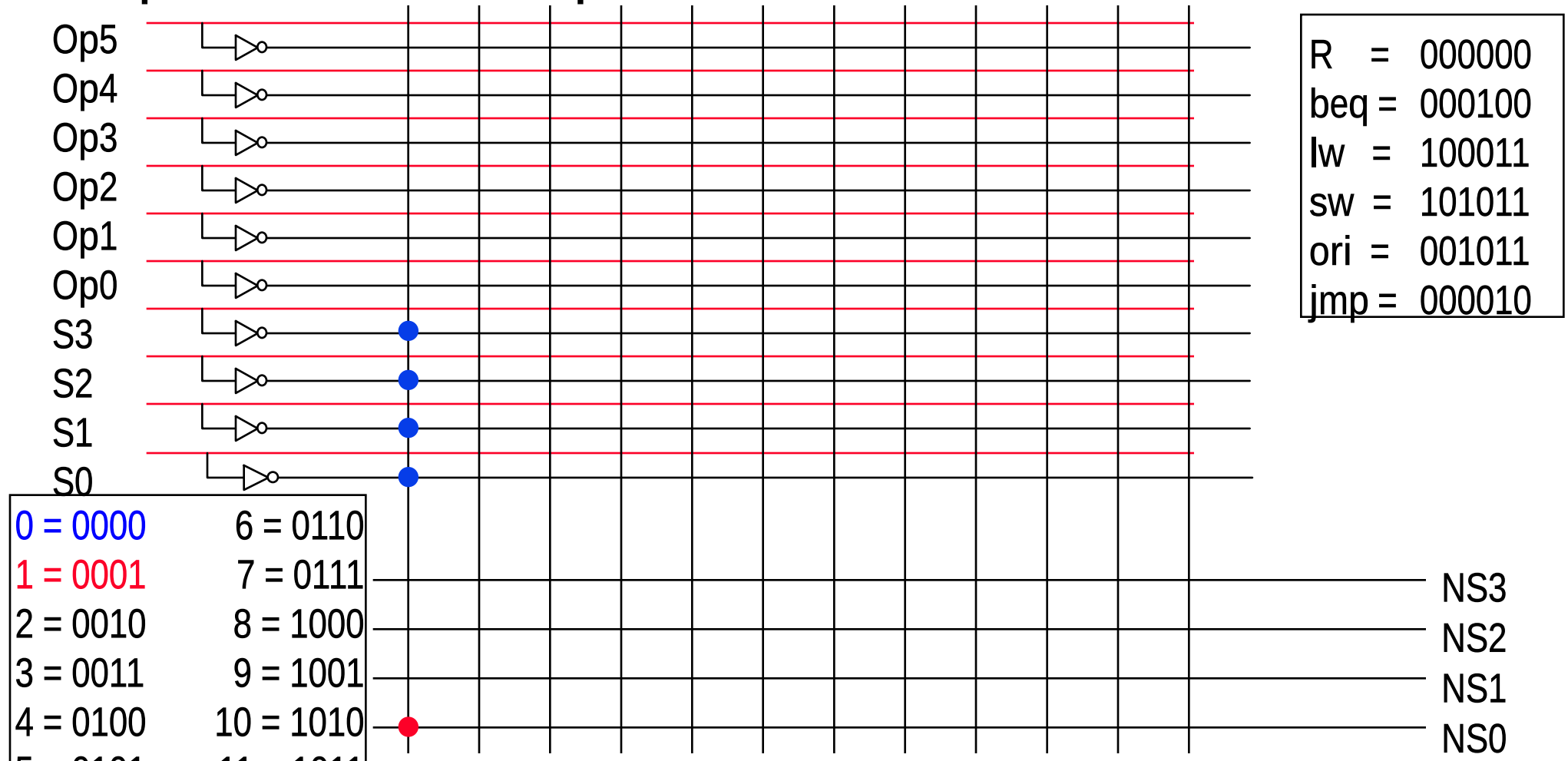
State 1 & op = jmp -> State 9

State 1 & op = ori -> State 10

State 10 -> State 11

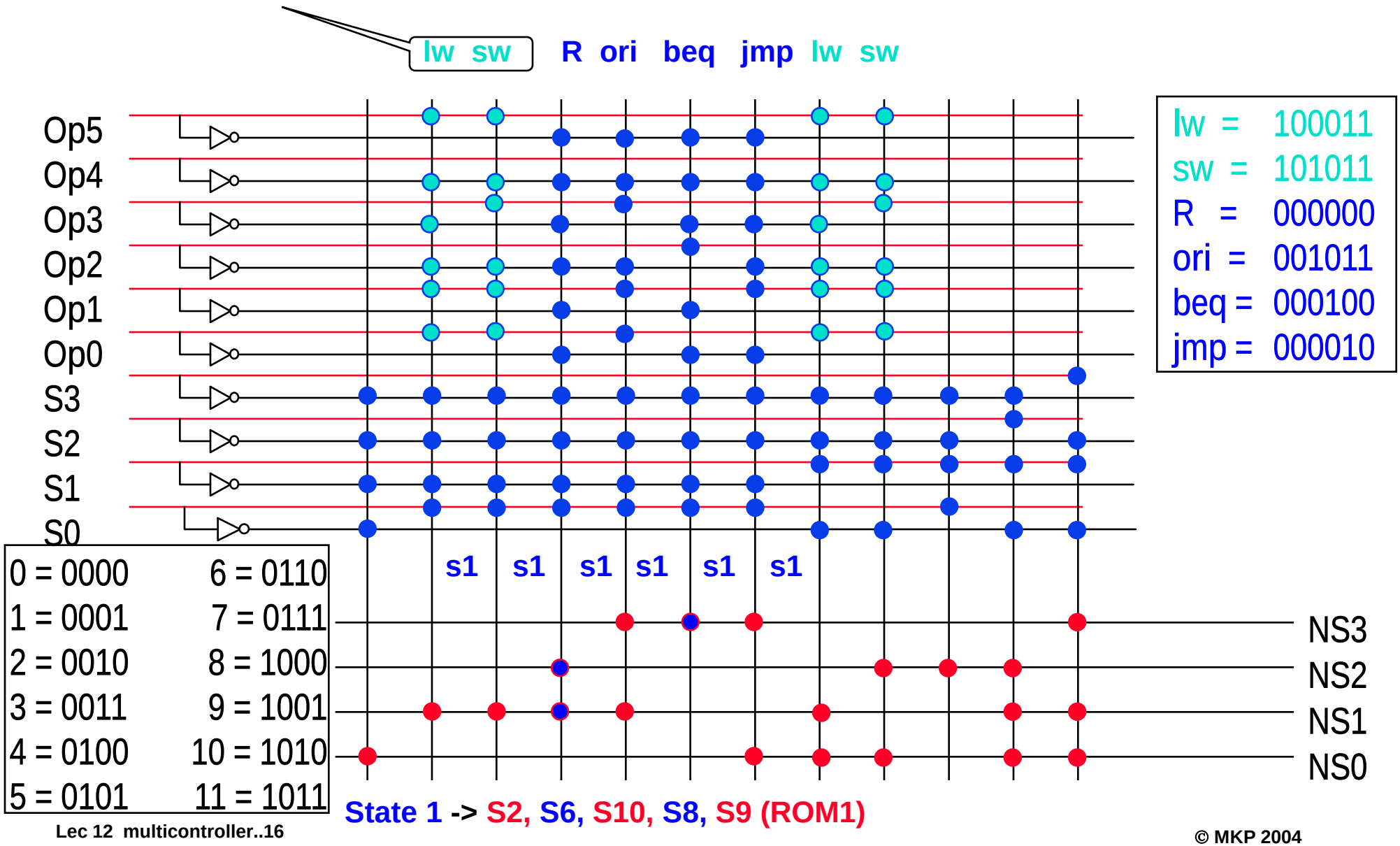
Implementation Technique: Programmed Logic Arrays

- **Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane**



Implementation Technique: Programmed Logic Arrays

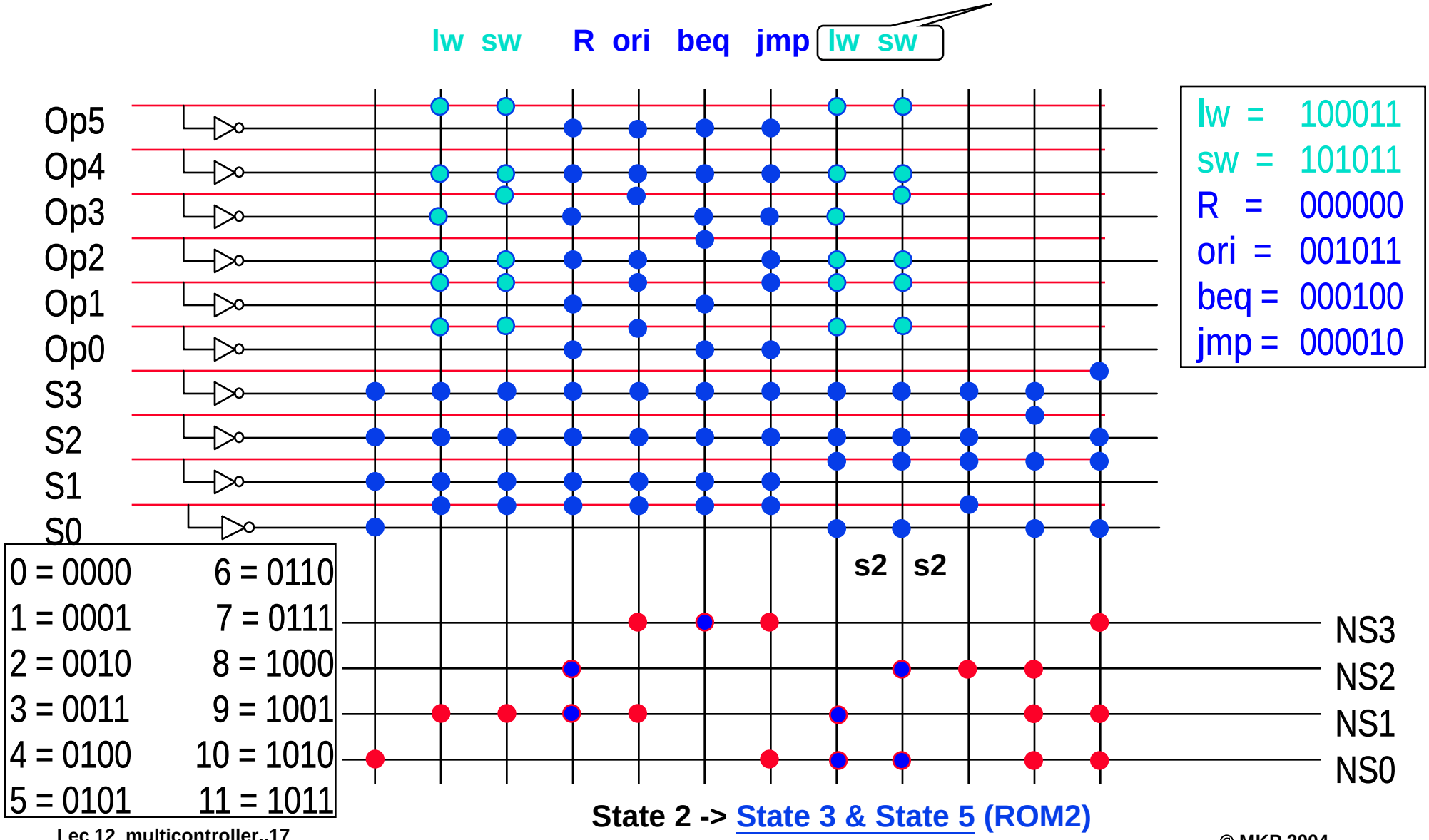
State 1 & op = lw | sw -> State 2



State 1 -> S2, S6, S10, S8, S9 (ROM1)

Implementation Technique: Programmed Logic Arrays

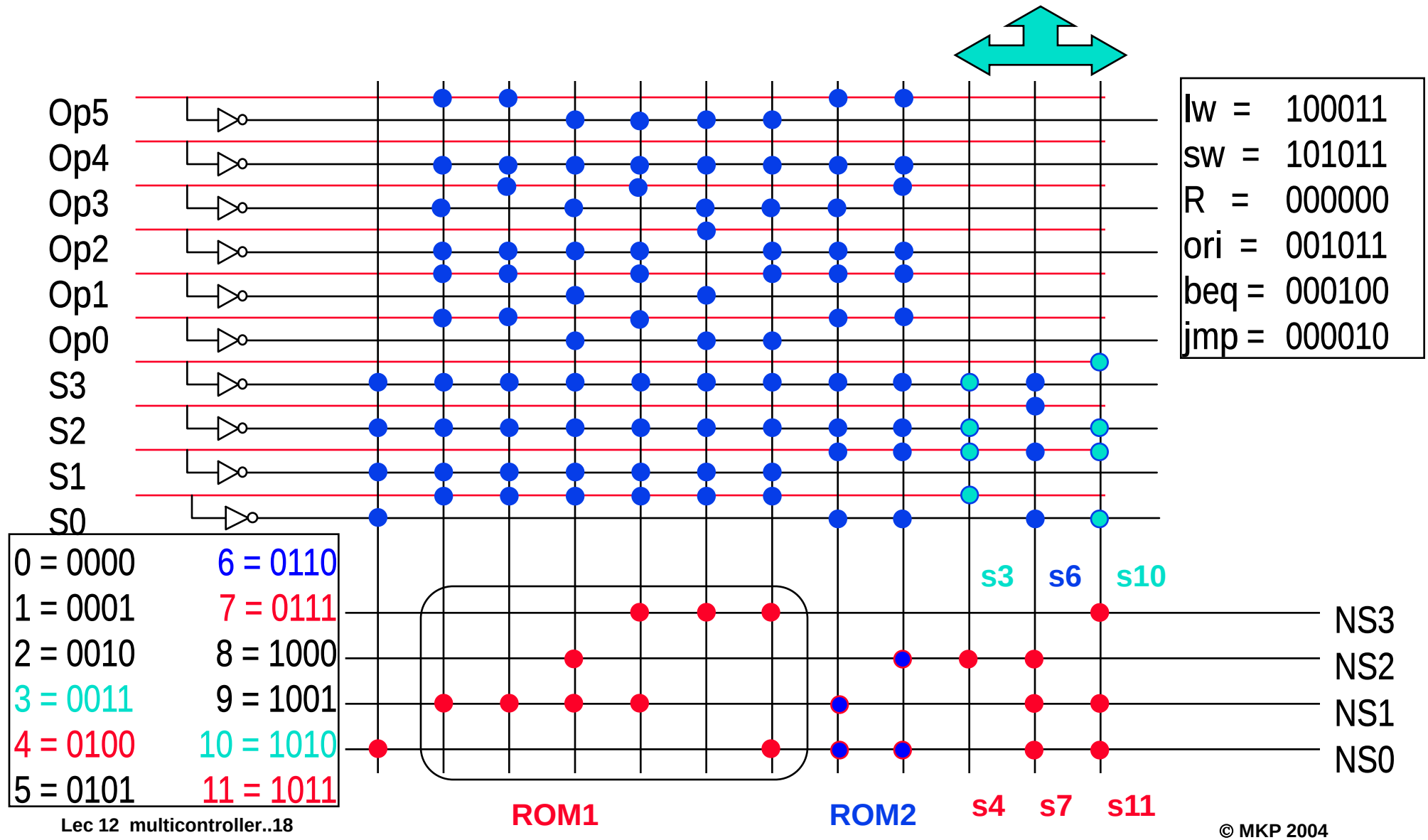
State 2 & op = lw | sw -> State 3 | Sate 5



State 2 -> State 3 & State 5 (ROM2)

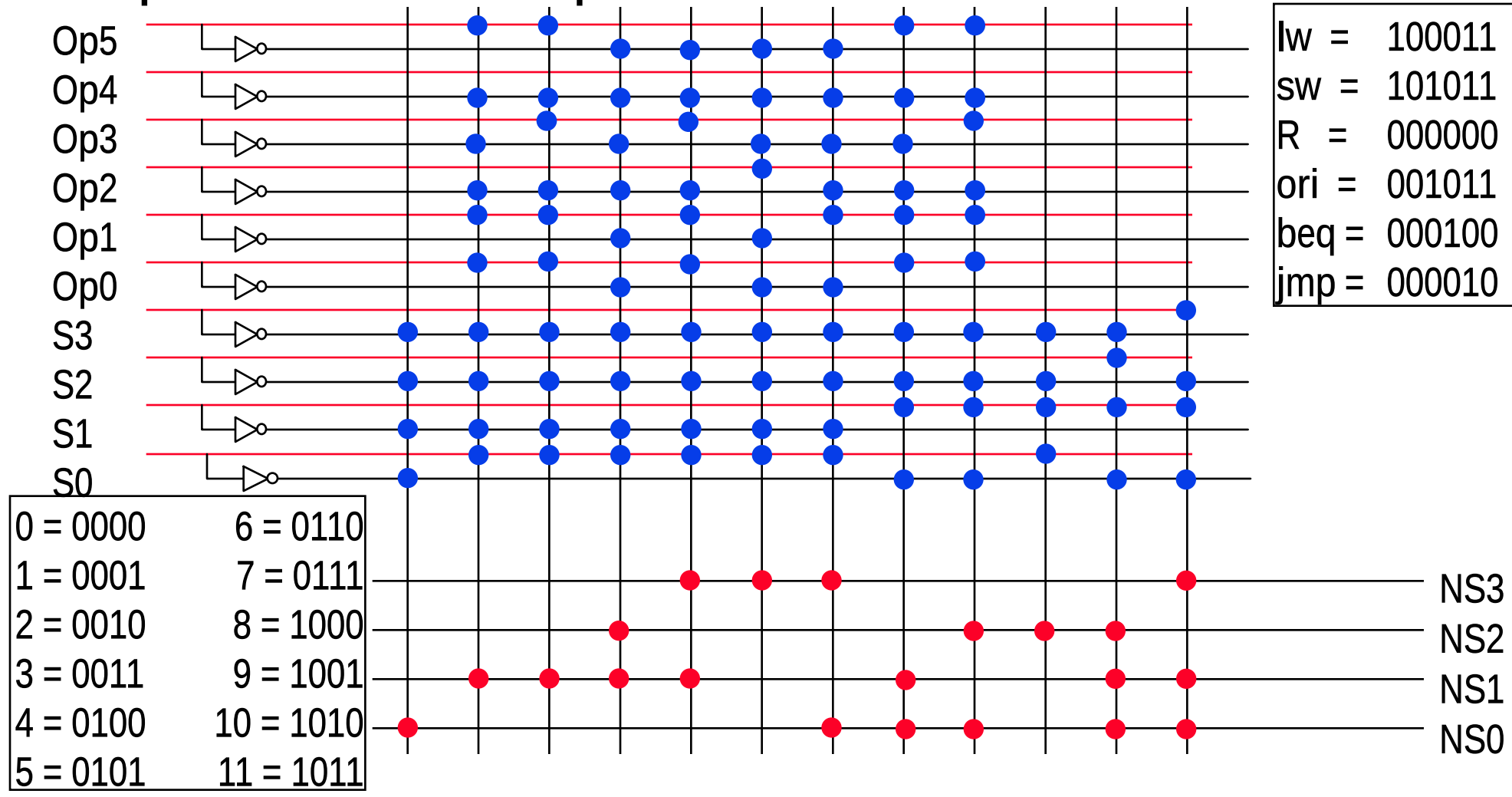
Implementation Technique: Programmed Logic Arrays

State 3 -> State 4, S6 -> S 7, S10 -> S11,



Implementation Technique: Programmed Logic Arrays

- **Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane**

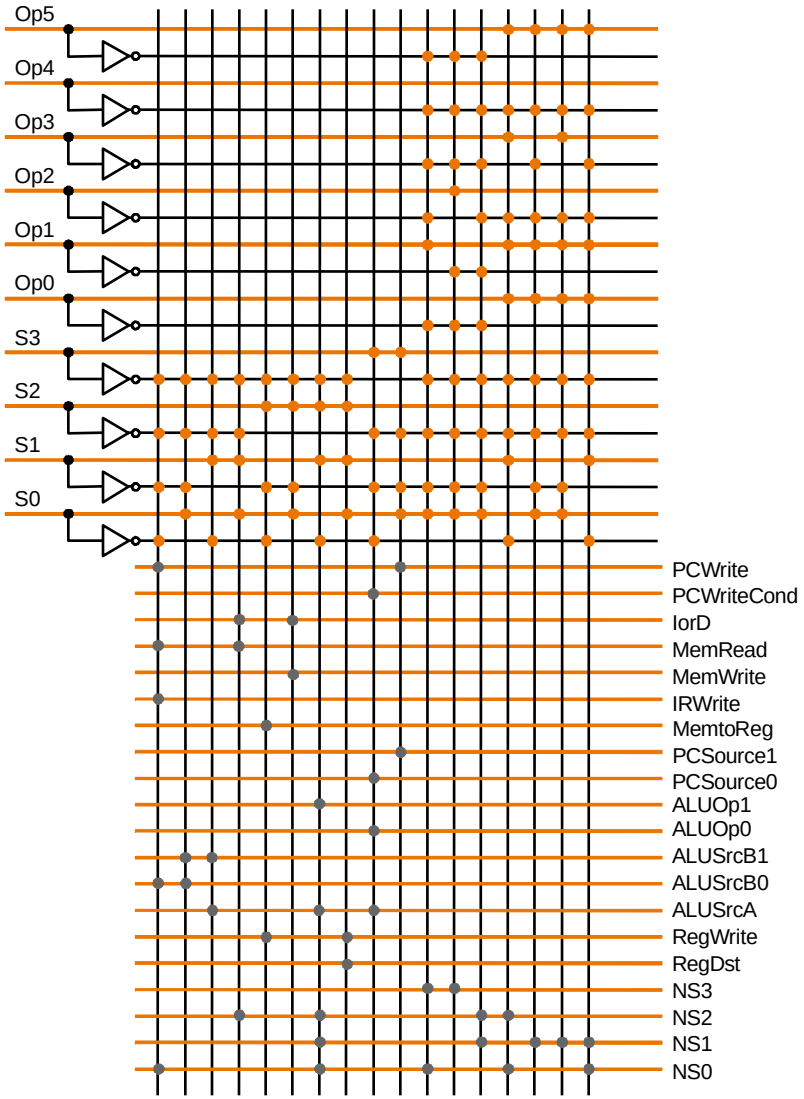


Multicycle Control

- Given numbers of FSM, can determine next state as function of inputs, including current state
- Turn these into Boolean equations for each bit of the next state lines
- Can implement easily using PLA
- What if many more states, many more conditions?
- What if need to add a state?

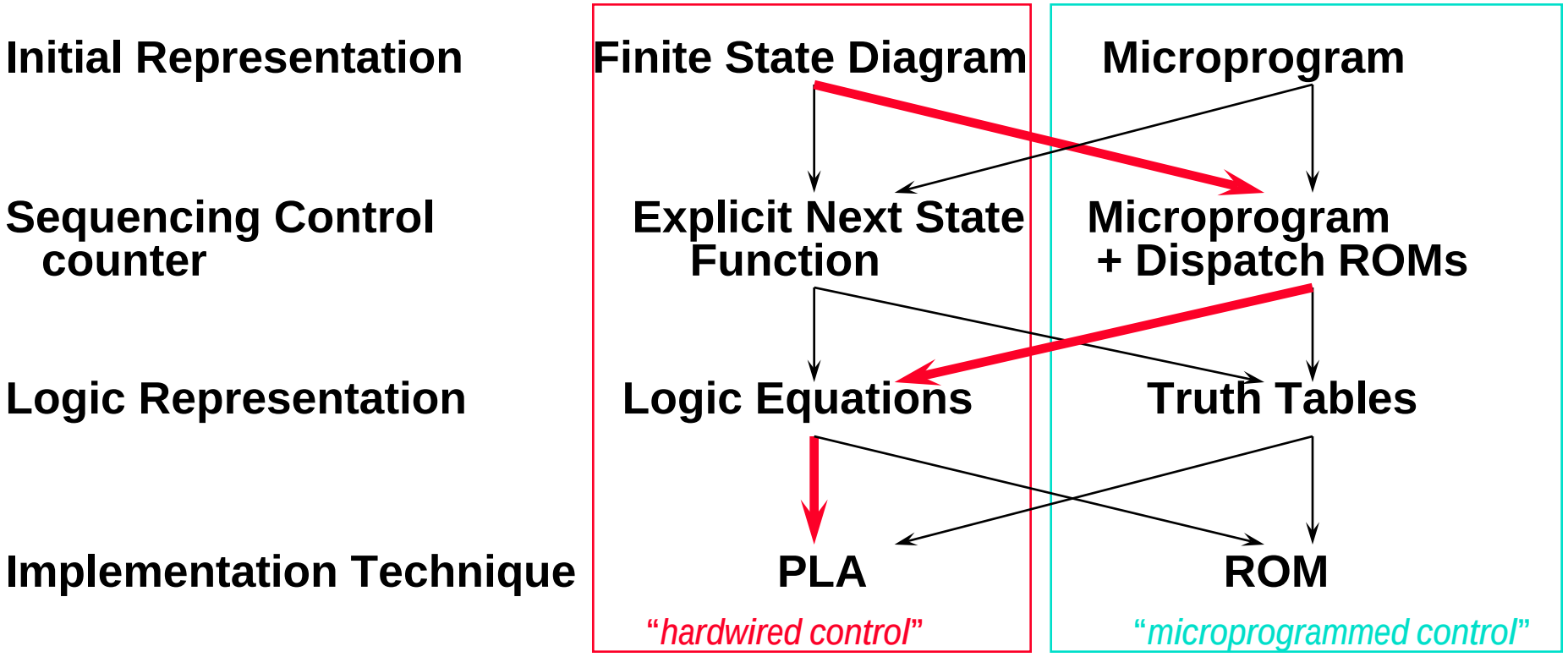
PLA Implementation

- If I picked a horizontal or vertical line could you explain it?

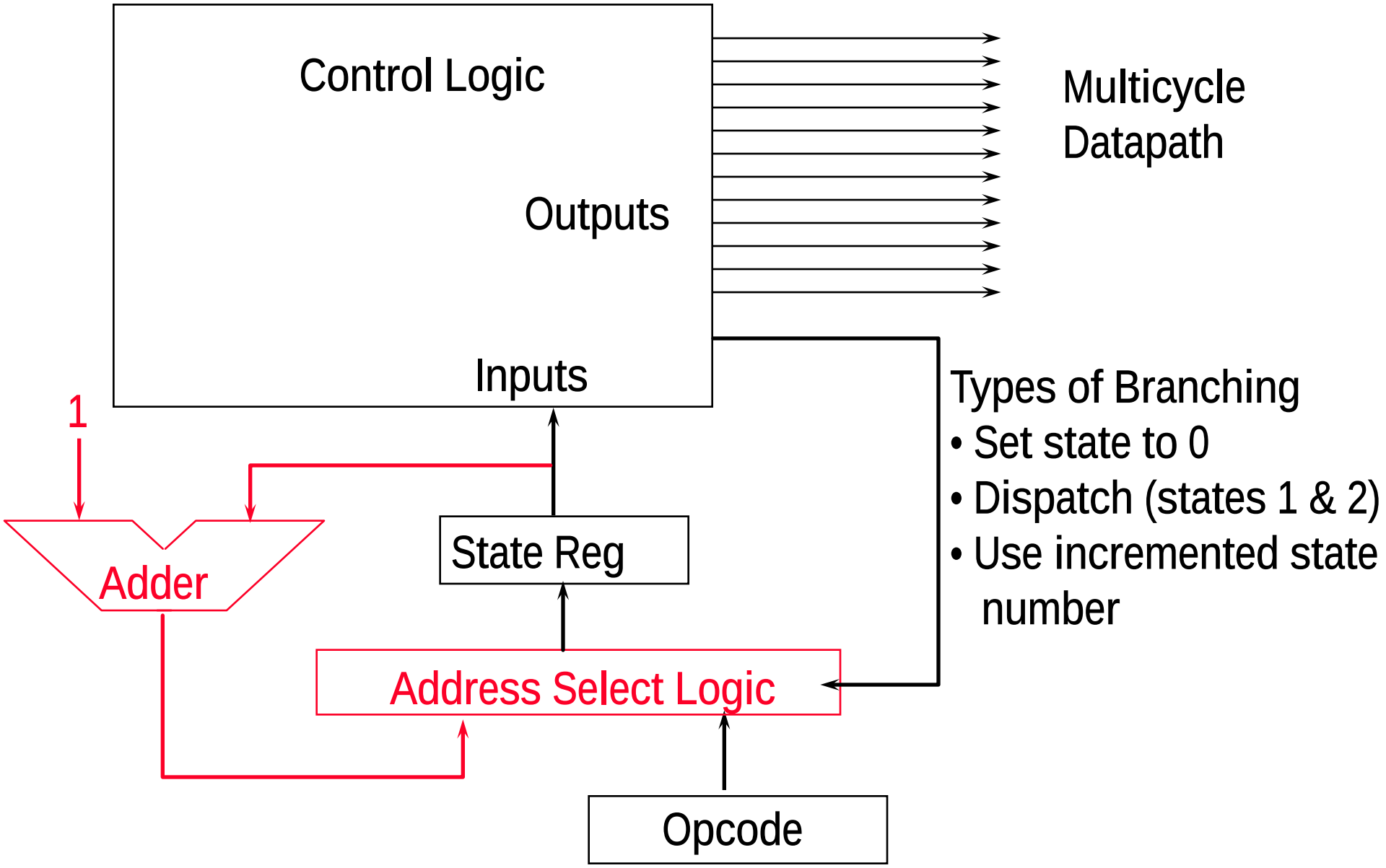


Next Iteration: Using Sequencer for Next State

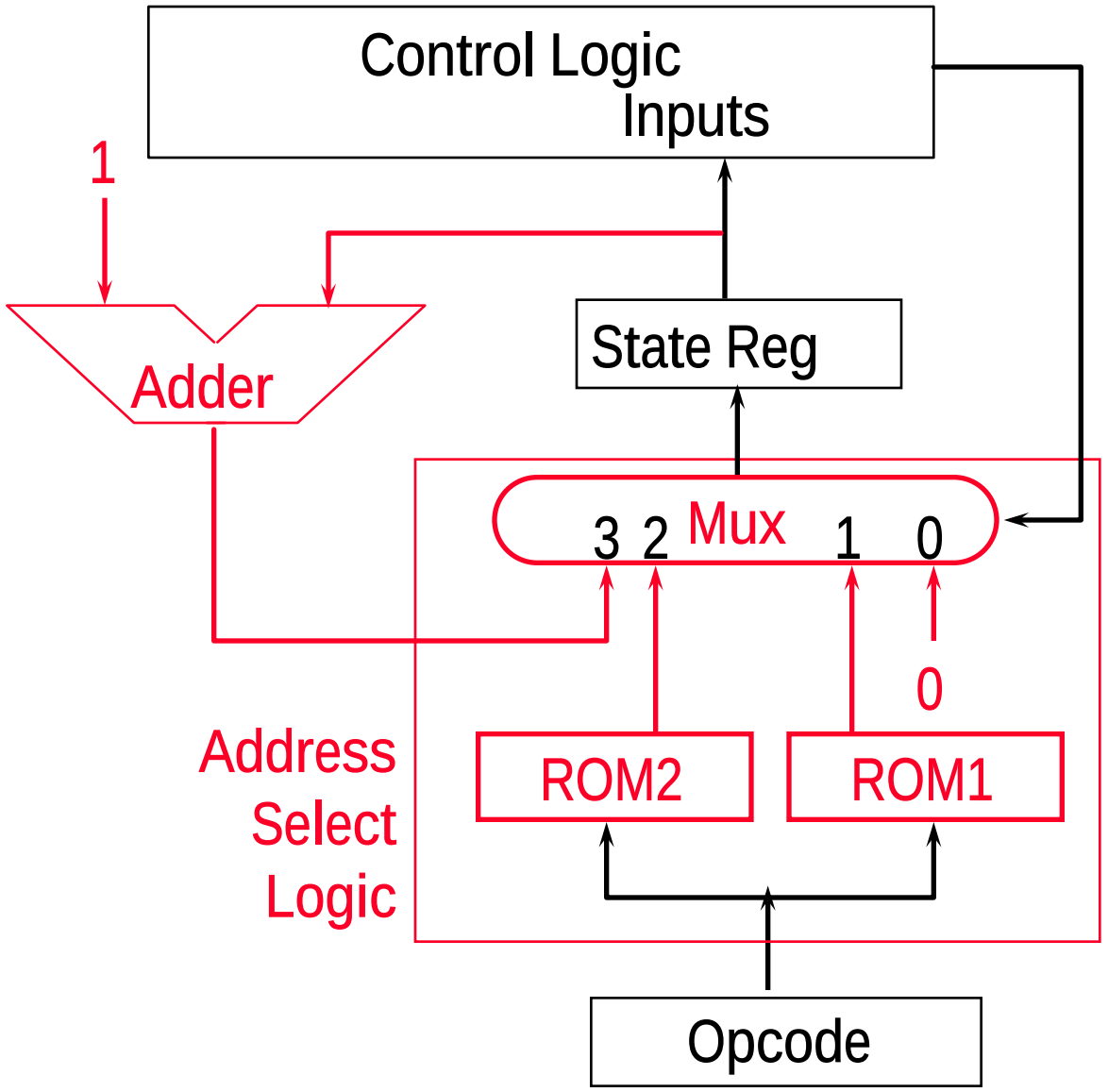
- Before Explicit Next State: Next try variation 1 step from right hand side
- Few sequential states in small FSM: suppose added floating point?
- Still need to go to non-sequential states: e.g., state 1 => 2,6,8,9,10



Sequencer-based control unit



Sequencer-based control unit details



Dispatch ROM 1

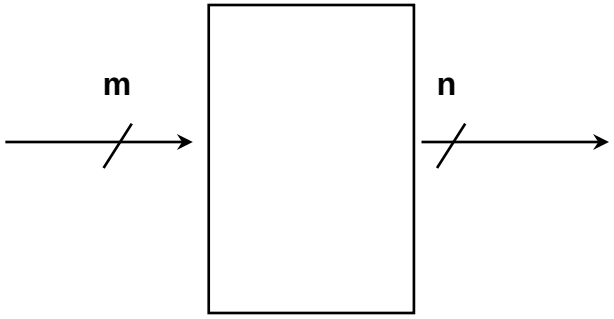
Op	*Name*	*State*
000000	Rtype	0110
000010	jmp	1001
000100	beq	1000
001011	ori	1010
100011	lw	0010
101011	sw	0010

Dispatch ROM 2

Op	*Name*	*State*
100011	lw	0011
101011	sw	0101

ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM.
  - our outputs are the bits of data that the address points to.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

m is the "height", and n is the "width"

Truth Table for the 16 Datapath Control Outputs (Fig. D.3.6)

Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

Implementing Control with a ROM (Fig. D.4.5)

- Instead of a PLA, use a ROM with one word per state ("control word")?

State number	*Control Word Bits 18-2*	*Control Word Bits 1-0*
0	10010100000001000	11
1	00000000010011000	01
2	00000000000010100	10
3	00110000000010100	11
4	00110010000010110	00
5	00101000000010100	00
6	00000000001000100	11
7	00000000001000111	00
8	01000000100100100	00
9	10000001000000000	00
10		11
11		00

ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., 2^{10} = 1024 different addresses)

- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs

- ROM is 2^{10} x 20 = 20K bits (and a rather unusual size)

- Rather wasteful, since for lots of the entries, the outputs are the same
  $\Rightarrow$ i.e., opcode is often ignored

ROM vs PLA

- Break up the table into two parts
  - ⇒ 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  - ⇒ 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  - ⇒ Total: 4.3K bits of ROM

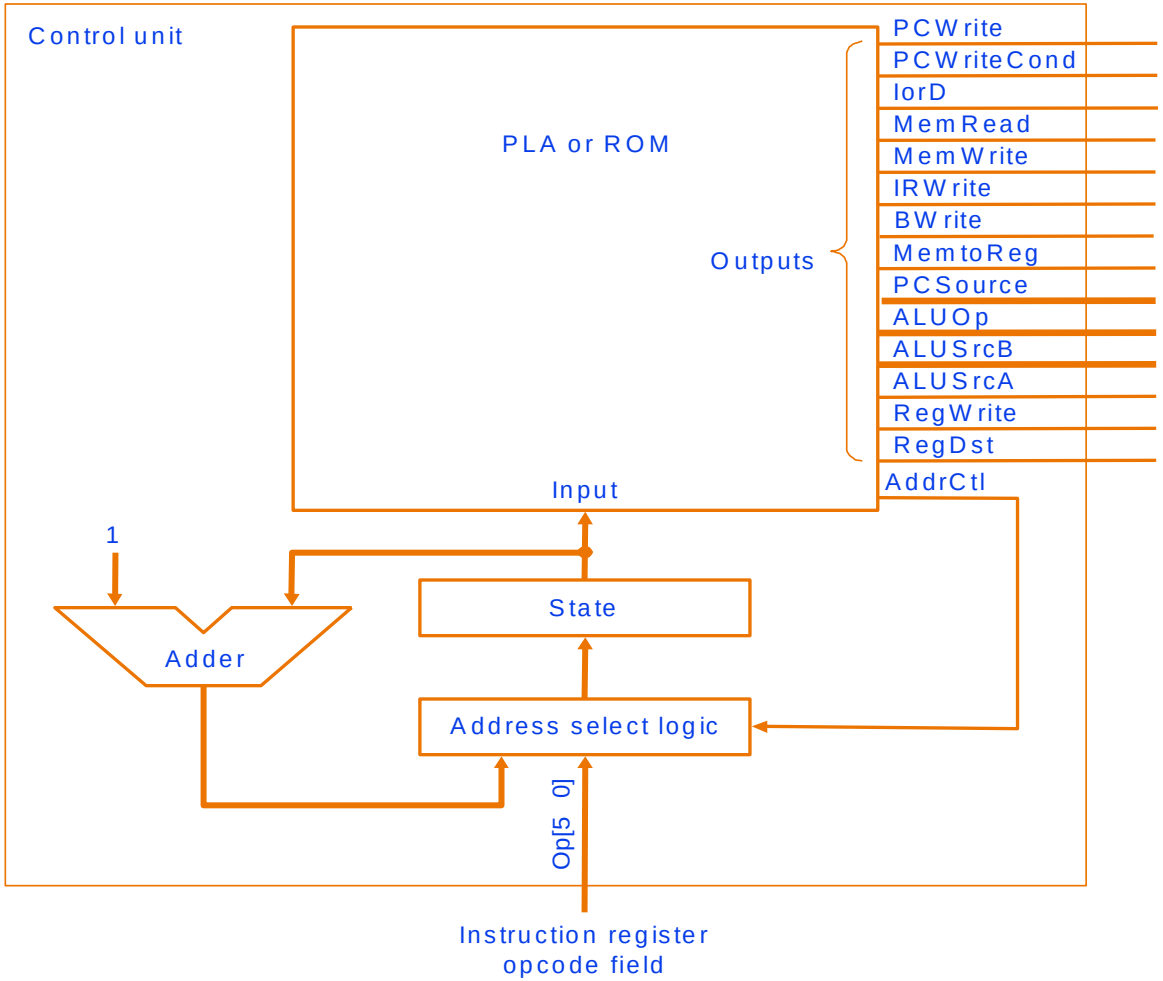
- PLA is much smaller
  - ⇒ can share product terms
  - ⇒ only need entries that produce an active output
  - ⇒ can take into account don't cares

- Size is (#inputs x #product-terms) + (#outputs x #product-terms)
  For this example = (10x 17)+ (20 x 17) = 510 PLA cells

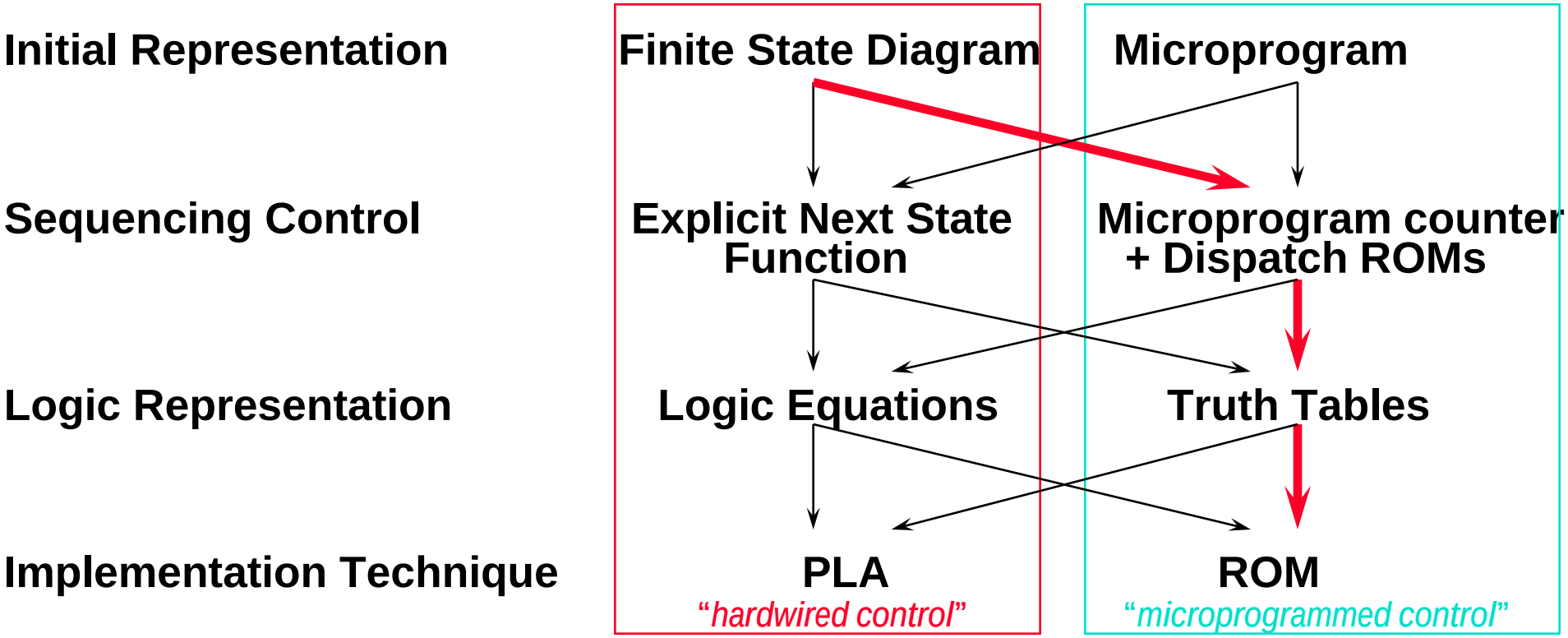
- PLA cells usually about the size of a ROM cell (slightly bigger)

Another Implementation Style

- **Complex instructions: the "next state" is often current state + 1**



Next Iteration: Using Microprogram for Representation

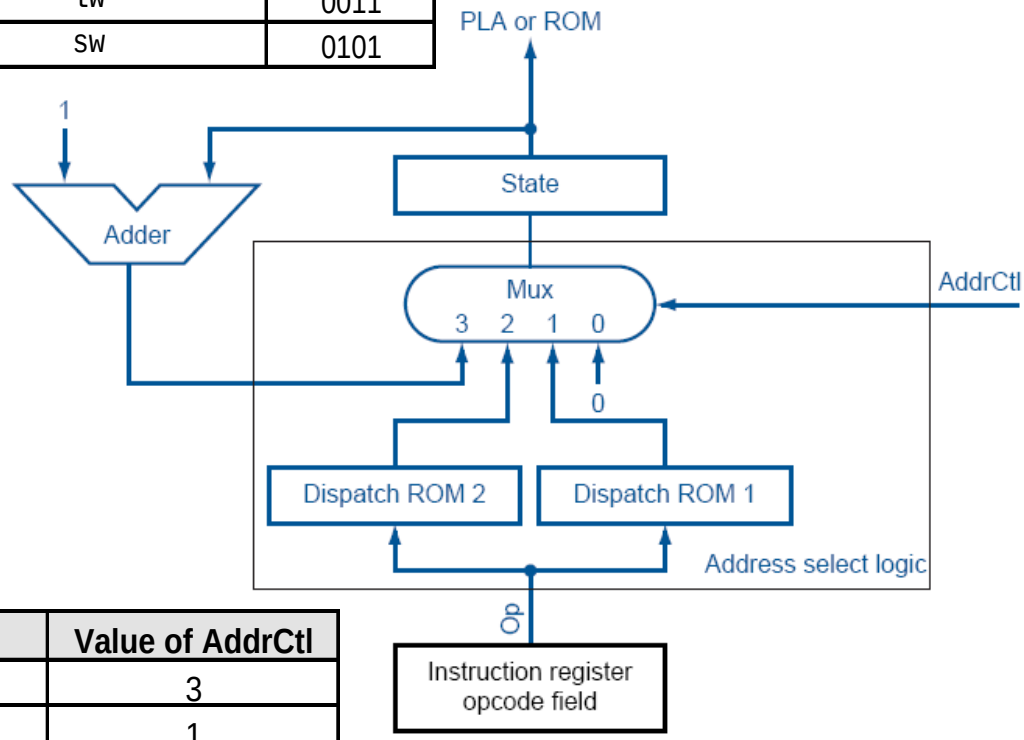


- ROM can be thought of as a sequence of control words
- Control word can be thought of as instruction: "Microinstruction"
- Rather than program in binary, use assembly language

Details (Figs. D.4.2, D.4.3, D.4.4)

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Microprogramming

- **Control is the hard part of processor design**
  - **Datapath is fairly regular and well-organized**
  - **Memory is highly regular**
  - **Control is irregular and global**

Microprogramming:

⇒ A Particular Strategy for Implementing the Control Unit of a processor by "programming" at the level of register transfer operations

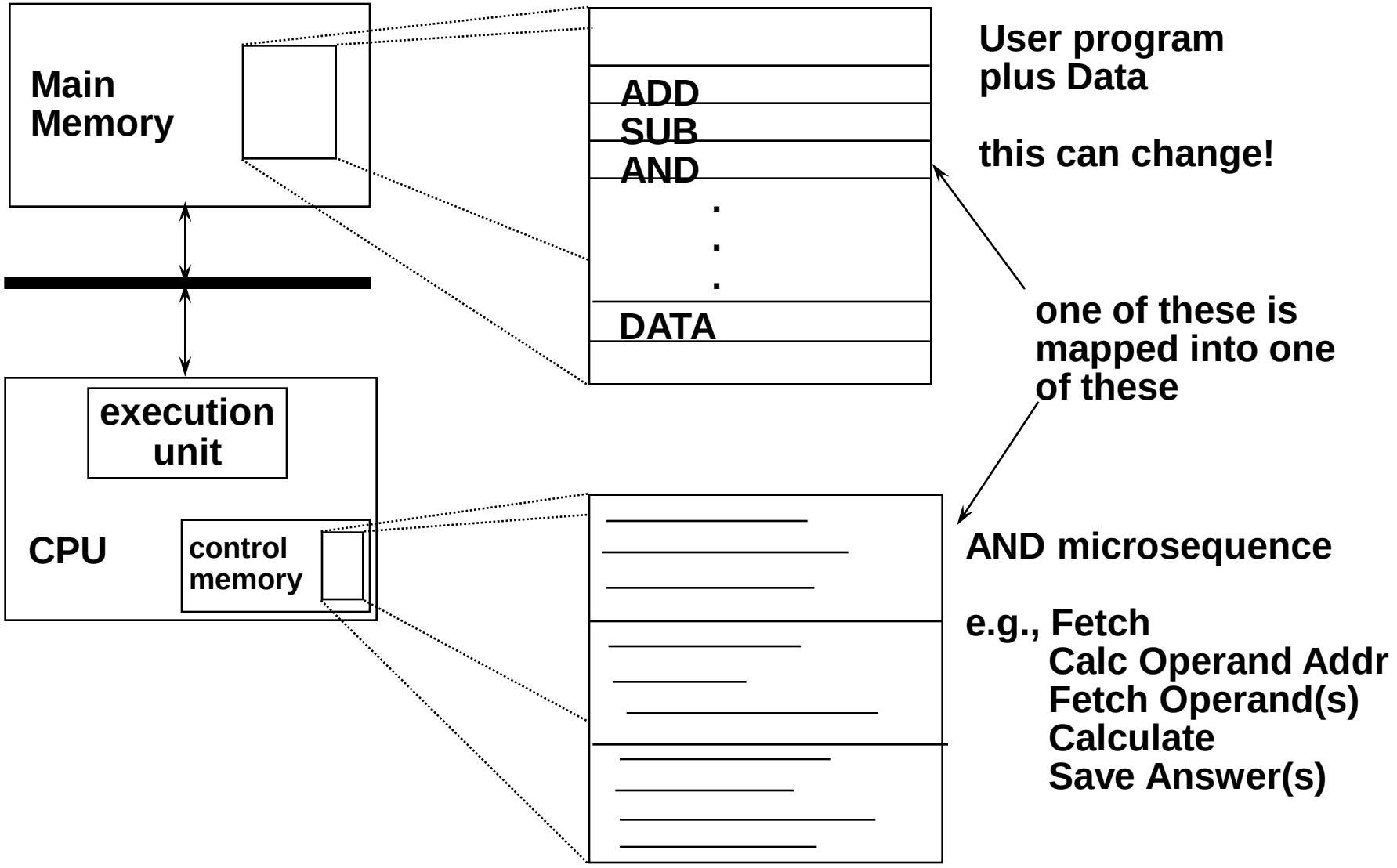
Microarchitecture:

⇒ Logical structure and functional capabilities of the hardware as seen by the microprogrammer

Historical Note:

IBM 360 Series first to distinguish between architecture & organization
Same instruction set across wide range of implementations, each with different cost/performance

Macroinstruction Interpretation



Variations on Microprogramming

- **Horizontal Microcode**

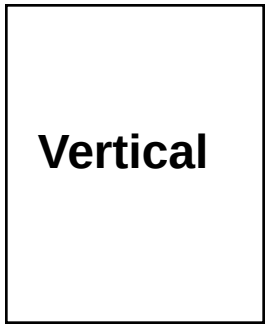
  control field for each control point in the machine

μseq	μaddr	A-mux	B-mux	bus enables	register enables	

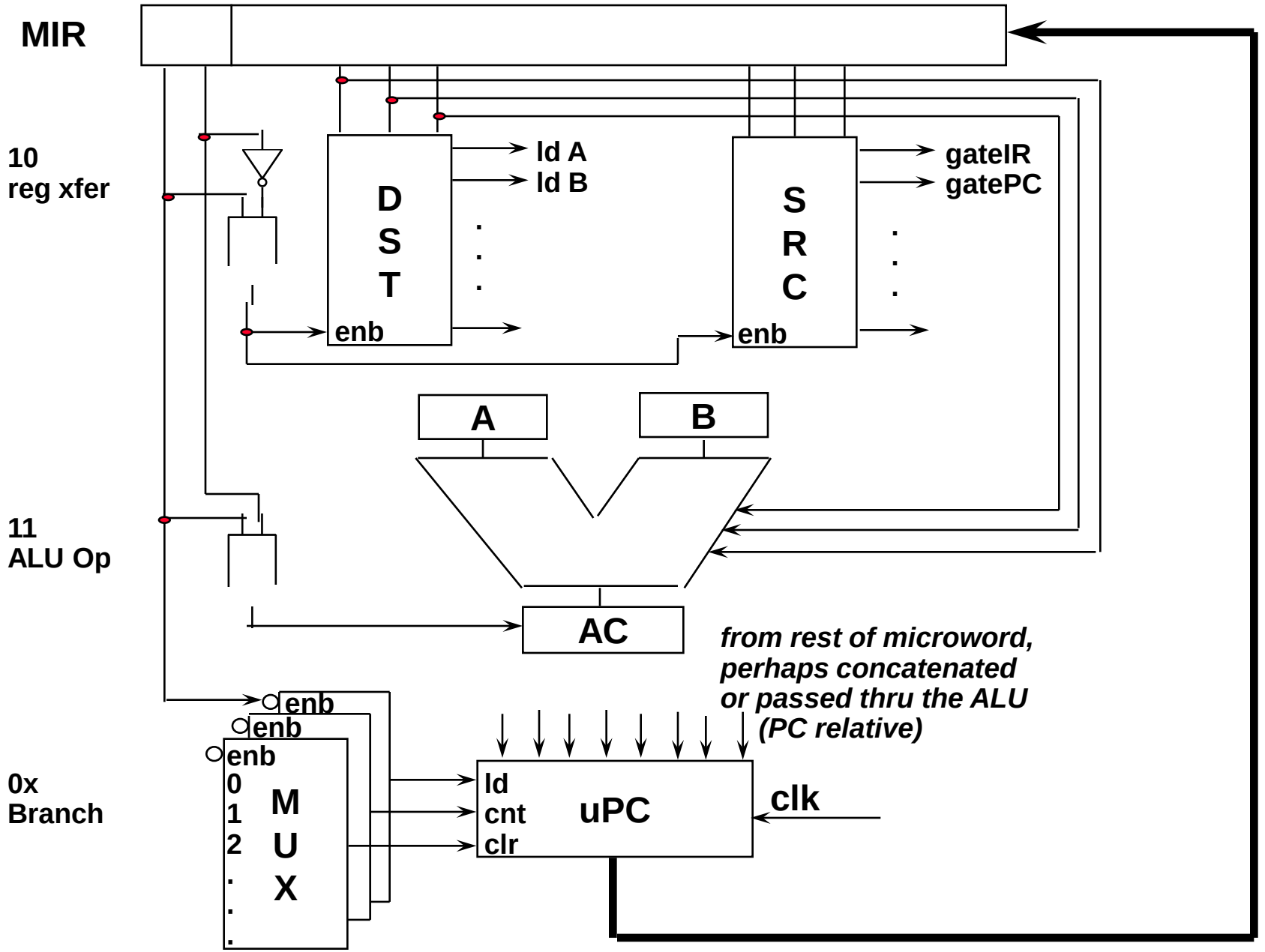
- **Vertical Microcode**

  compact microinstruction format for each class of microoperation

branch:	**μseq-op**	**μadd**
execute:	**ALU-op**	**A,B,R**
memory:	**mem-op**	**S, D**



Controller Implementation



Microprogramming Pros and Cons

- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine
  - Can tailor instruction set to application
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

Microprogramming one inspiration for RISC

- If simple instruction could execute at very high clock rate
- If you could even write compilers to produce microinstructions
- If most programs use simple instructions and addressing modes
- If microcode is kept in RAM instead of ROM so as to fix bugs
- If same memory used for control memory could be used instead as cache for “macroinstructions” ...?
- Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine?

Summary: Multicycle Control

- Microprogramming and hardwired control have many similarities, perhaps biggest difference is initial representation and ease of change of implementation, with ROM generally being easier than PLA