Lecture 13
Control Unit Implementation

School of Computer Science and Engineering
Soongsil University

4. The Processor

Finalizing the Control

Input		Output									
mnemonic	opcode	RegDst	ALUSrc	MemetoReg	RegWrite	memRead	MemWrite	Branch	ALUOp1	ALUOp2	Jump
R-type	000000	1	0	0	1	0	0	0	1	0	0
`lw`	100011	0	1	1	1	1	0	0	0	0	0
`sw`	101011	X	1	X	0	0	1	0	0	0	0
`beq`	000100	X	0	X	0	0	0	1	0	1	0
`j`	000010	X	X	X	0	0	0	X	X	X	1
`addi`	001000										

Modified Figure 4.22

Implementation of the Control

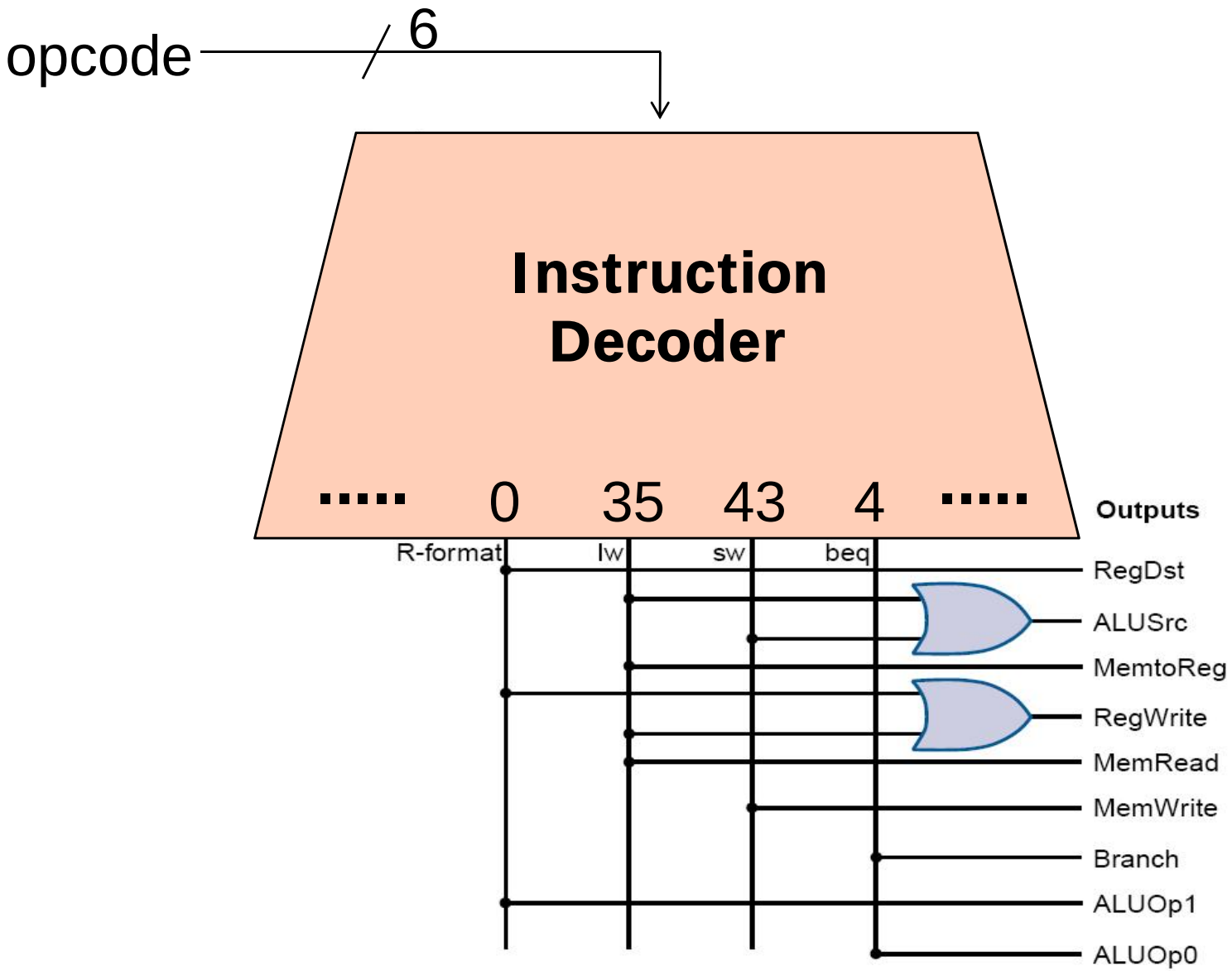


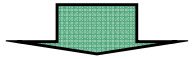
Figure C.2.5

Why a Single-Cycle Implementation Is Not Used Today

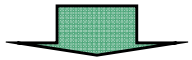
- CPI = 1
- Clock cycle is determined by the longest possible path
- Critical path
  - Load instruction
  - instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- Performance can be improved by pipelining

Problems with a Single-Cycle Implementation

- Too long a clock cycle with floating-point unit or complex instruction set
- Implementation techniques that reduce common case delay but do not reduce worst-case cycle time cannot be used.
- Some functional units must be duplicated.



Inefficient both in performance and in hardware cost.



- **Multicycle datapath**
  - Shorter clock cycle
  - Multiple clock cycles for each instruction

Multicycle Datapath

- Each step in the execution will take 1 clock cycle
- Different number of clock cycles for the different instructions
- Sharing functional units within the execution of a single instruction

Eliminated from the 4th edition!

Example from 3rd edition

- **Operation times for the major functional units**
  - Memory units : 200 ps
  - ALU and adders : 100 ps
  - Register file : 50 ps
  - Multiplexors, control unit, PC, sign-extension unit, wires: no delay
- **Instruction mix**
  - 25% `lw`, 10% `sw`, 45% R-type, 15% `beq`, 5% `j`
- **Which would be faster and by how much ?**

  (1) One clock cycle of a fixed length

  (2) One clock cycle for every instruction but variable-length clock

[Answer-1]

- CPU execution time
  = instruction count x CPI x clock cycle time
  = instruction count x clock cycle time ($\because$ CPI = 1)
- Compare the clock cycle time for both implementation.
- We can find critical path and instruction execution time for each instruction at the next page.

[Answer-2]

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

[Answer-3]

- For the fixed-length clock implementation,

  Clock cycle = 600 ps.

- For the variable-length clock implementation,

  CPU clock cycle

  = 600×0.25+550×0.1+400×0.45+350×0.15+200×0.05 = 447.5 ps

$$\frac{CPU\ Performance_{variable}}{CPU\ Performance_{fixed}} = \frac{CPU\ execution\ time_{fixed}}{CPU\ execution\ time_{variable}}$$

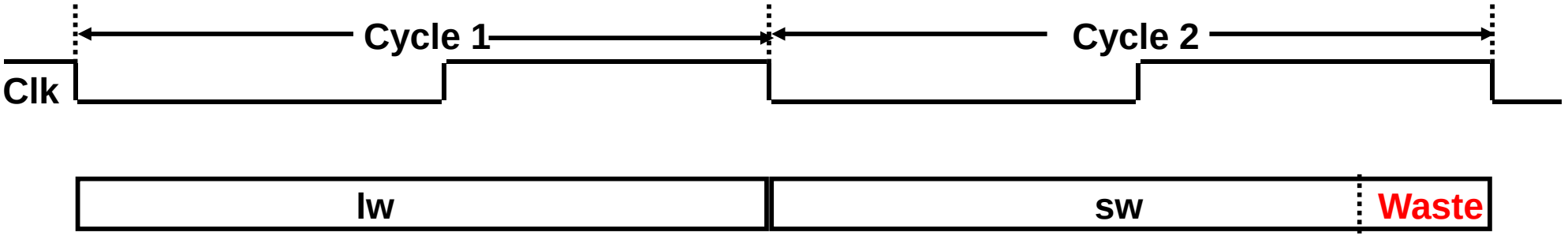
$$= \frac{IC \times CPU\ clock\ cycle_{fixed}}{IC \times CPU\ clock\ cycle_{variable}}$$

$$= \frac{CPU\ clock\ cycle_{fixed}}{CPU\ clock\ cycle_{variable}} = \frac{600}{447.5} = 1.34$$

- The variable clock implementation would be 1.34 times faster.
- But, it is hard to implement and its overhead is large.

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply

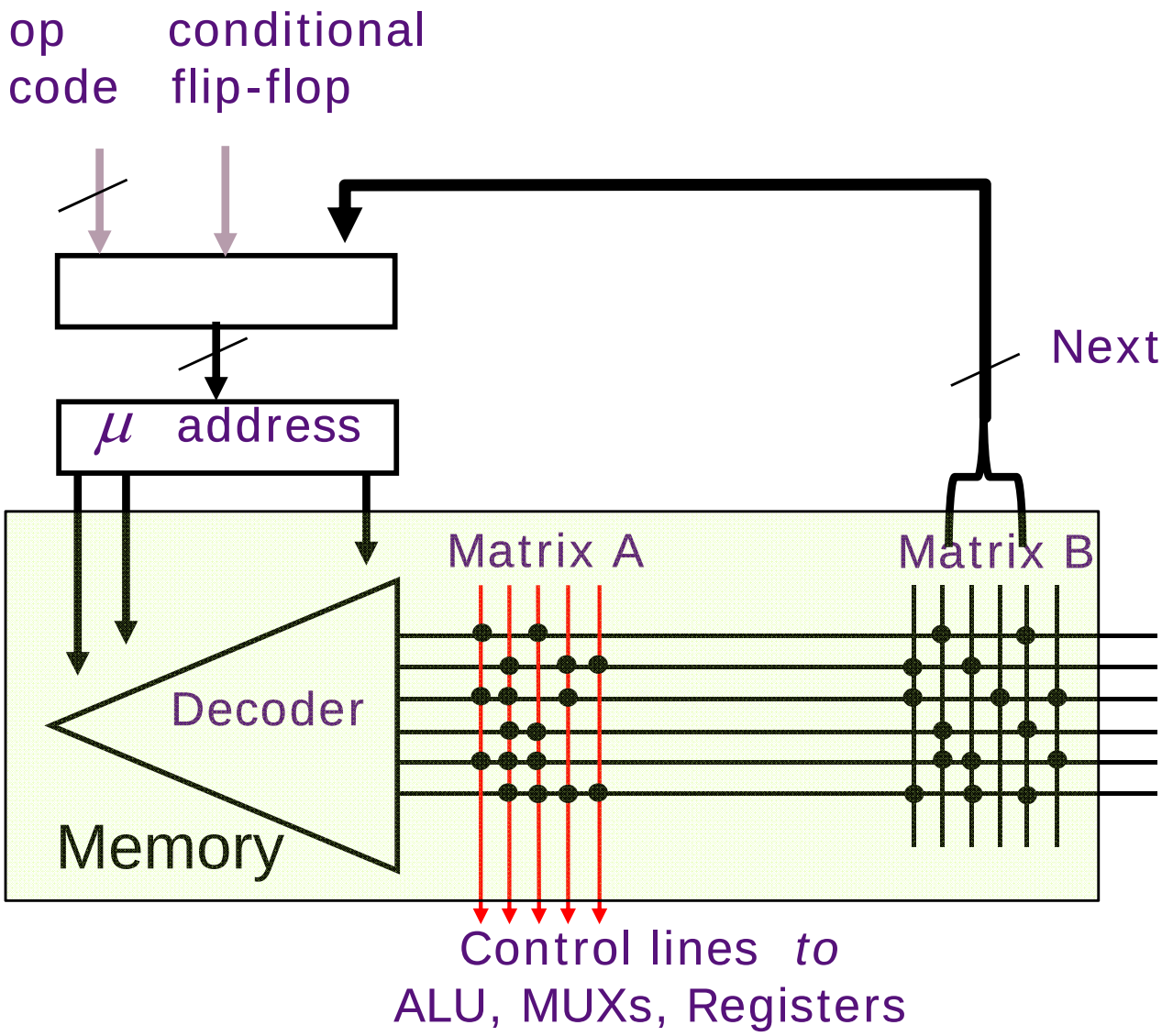


- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle
- but, is simple and easy to understand

Multicycle Implementation

Step	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
1	IR ← Memory[PC] PC ← PC + 4			
2	A ← Reg[IR[25:21]] B ← Reg[IR[20:16]] ALUOut ← PC + (sign-extend(IR[15:0]) << 2)			
3	ALUOut ← A op B	ALUOut ← A + sign-extend(IR[15:0])	If(A == B) PC ← ALUOut	PC ← PC[31:28] || (IR[25:0]<<2)
4	Reg[IR[15:11]] ← ALUOut	Load: MDR←Memory[ALUOut] or Store: Memory[ALUOut] ← B		
5		Load : Reg[IR[20:16]] ← MDR		

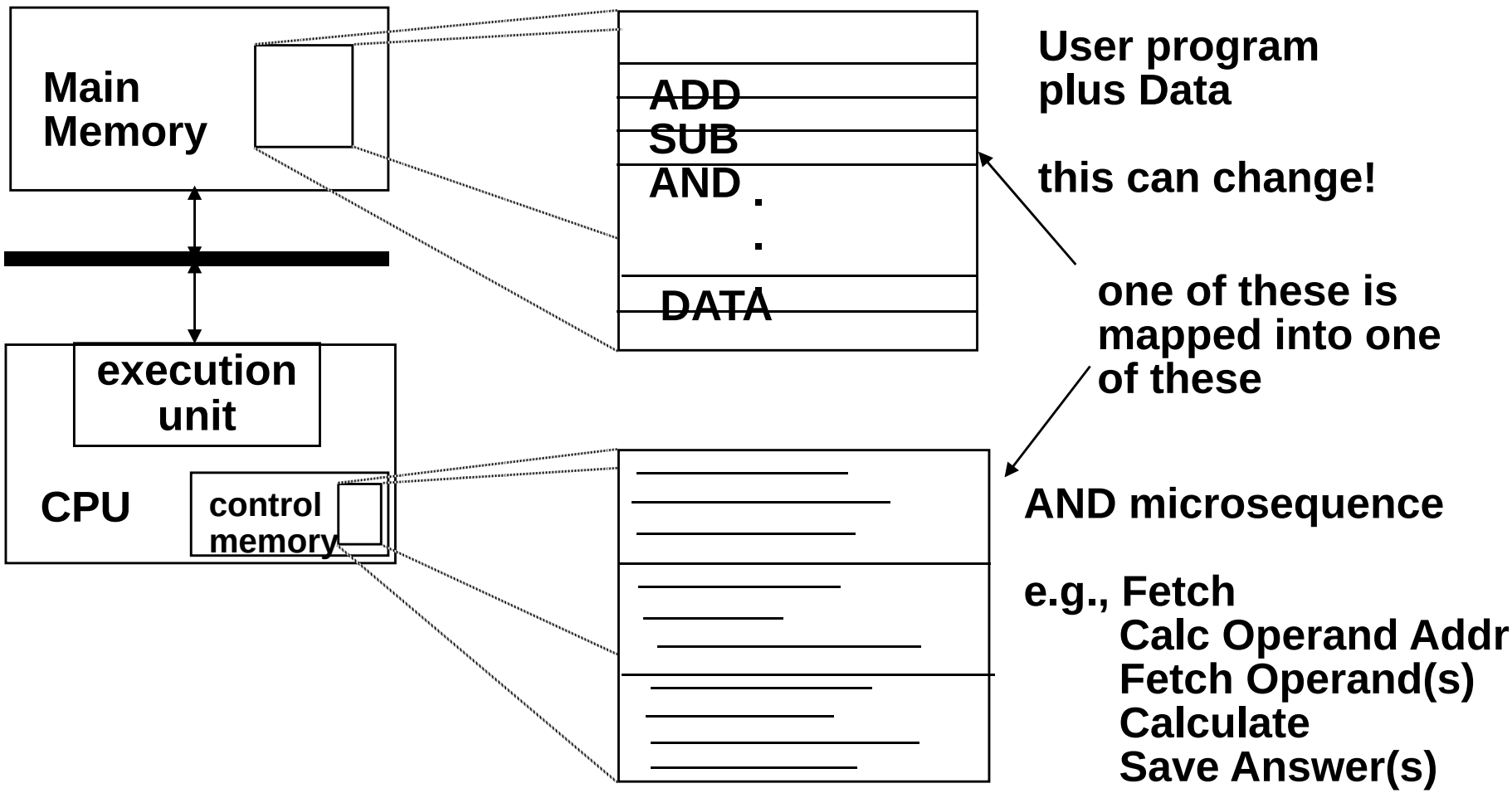
Microprogrammed Control Unit



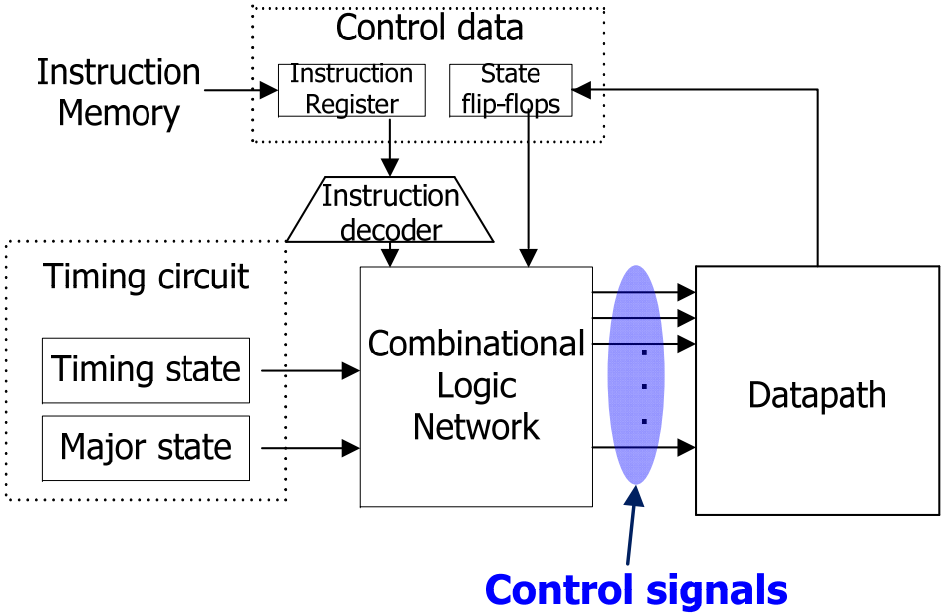
- Proposed by Maurice Wilkes in 1954
- First used in EDSAC-2, completed 1958

Embed the control logic state table in a memory array

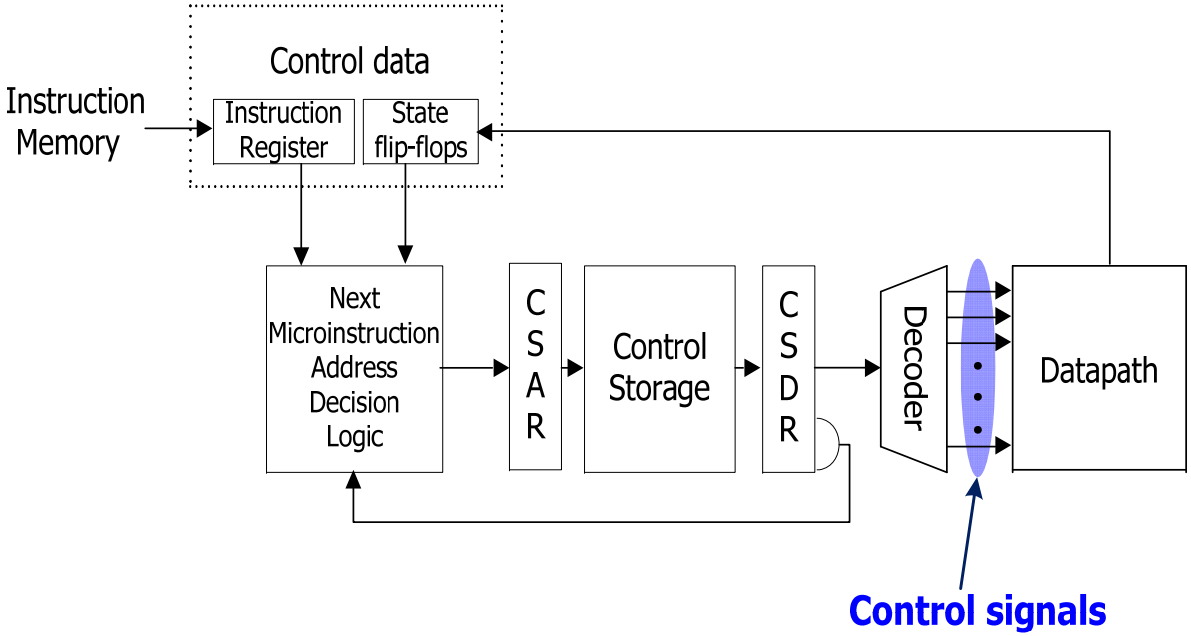
Microprogramming



Hardwired vs. Microprogrammed



Hardwired control



Microprogrammed control

Supplement

Multicycle Datapath and Control

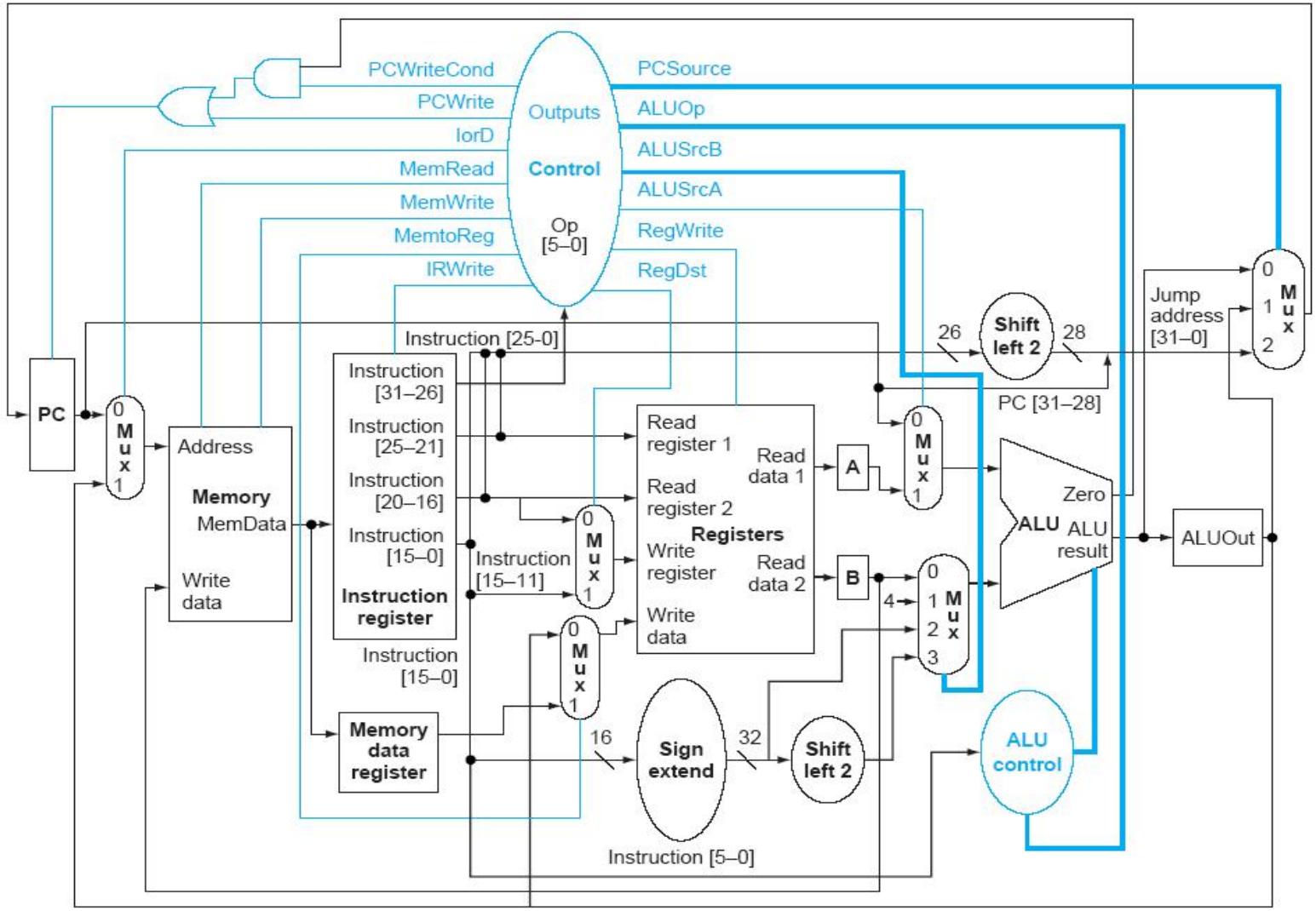


Figure 5.28
of 3rd edition

Control Signals for Multicycle Datapath

		ALU Op	ALU SrcA	ALU SrcB	Mem To Reg	Reg Dst	PC Source	IorD	Mem Read	Mem Write	Reg Write	PC Write	PC Write Cond	IR Write	State
fetch	IR=mem[PC] PC=PC+4	00	0	01			00	0	1			1		1	0
decode	A=Reg[IR[25-21]] B=Reg[IR[20-16]] ALUOut=PC+(sign-extend(IR[15-0]<<2)	00	0	11											1
Memory refer.	ALUout=A+sign-extend(IR[15-0])	00	1	10											2
	MDR=mem[ALUOut]							1	1						3
	Reg[IR[20-16]]=MDR				1	0					1				4
	ALUout=A+sign-extend(IR[15-0])	00	1	10											2
	mem[ALUOut]=B							1		1					5
R-type	ALUOut=A op B	10	1	00											6
	Reg[IR[15-11]]=ALUOut				0	1					1				7
beq	if (A==B) then PC=ALUOut	01	1	00			01						1		8
jump	PC=PC[31-28] || (IR[25-0]<<2)						10					1			9

Finite State Machine

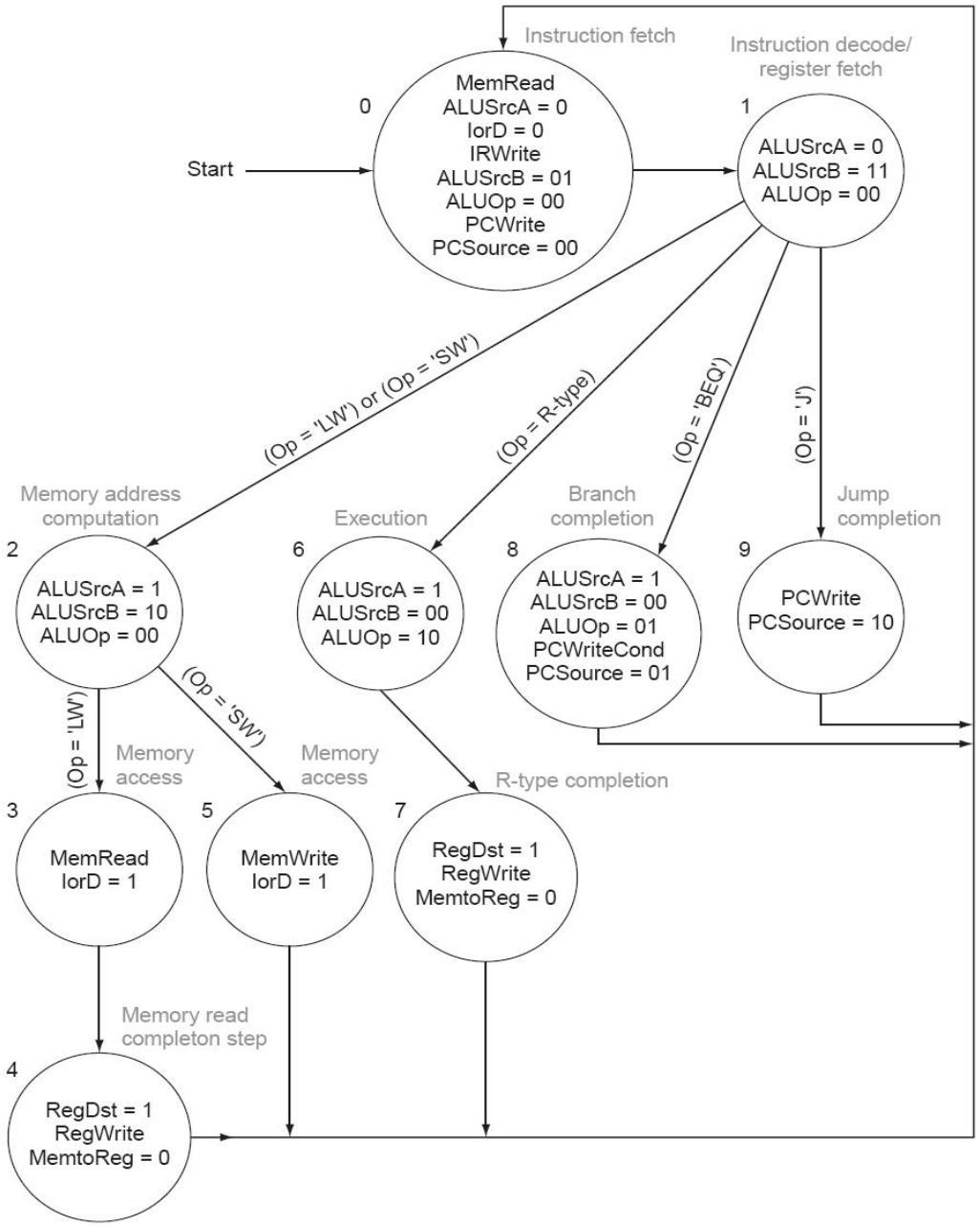


Figure 5.38
of 3rd edition

Block diagram of Hardwired Control

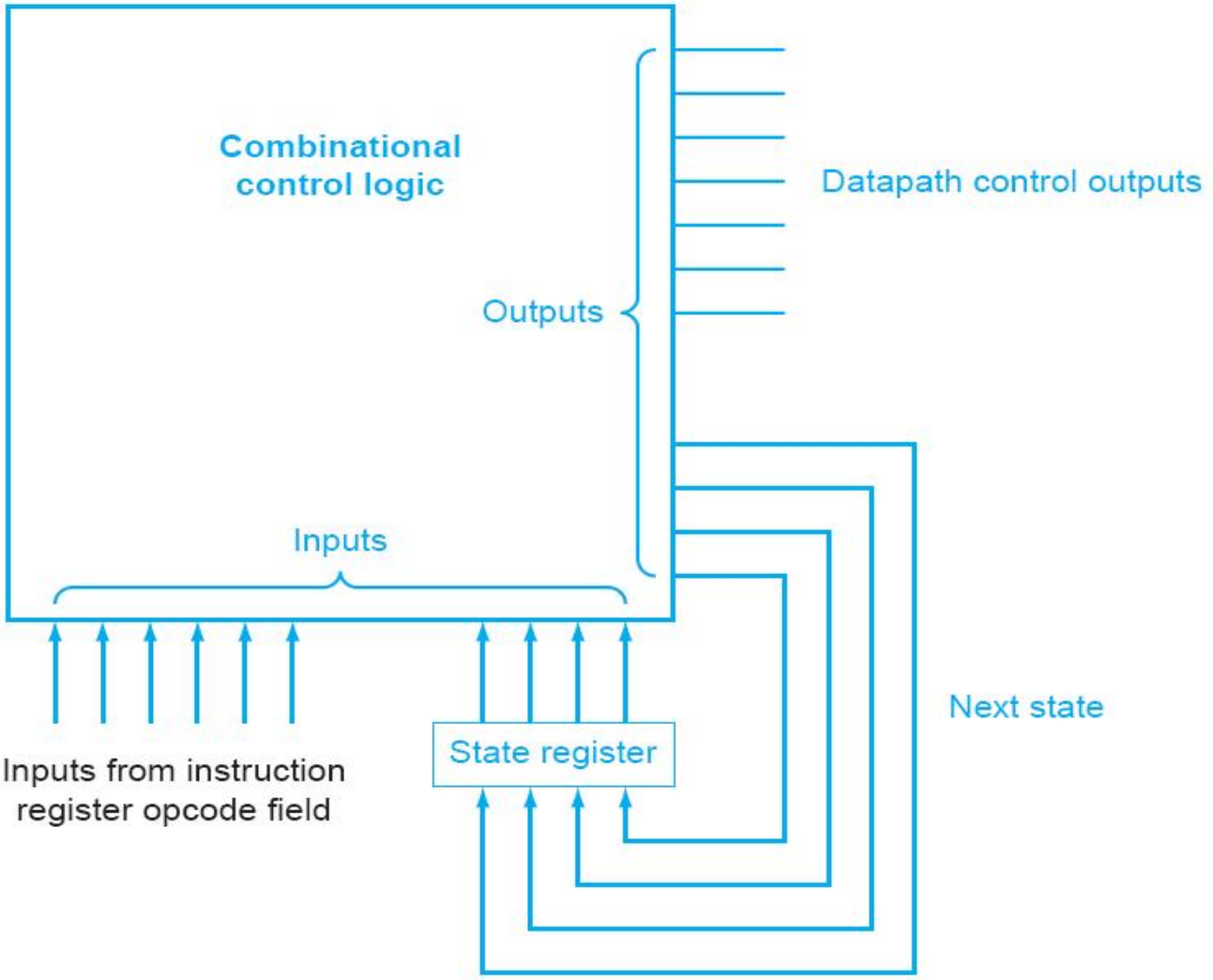


Figure 5.37
of 3rd edition

PLA Implementation

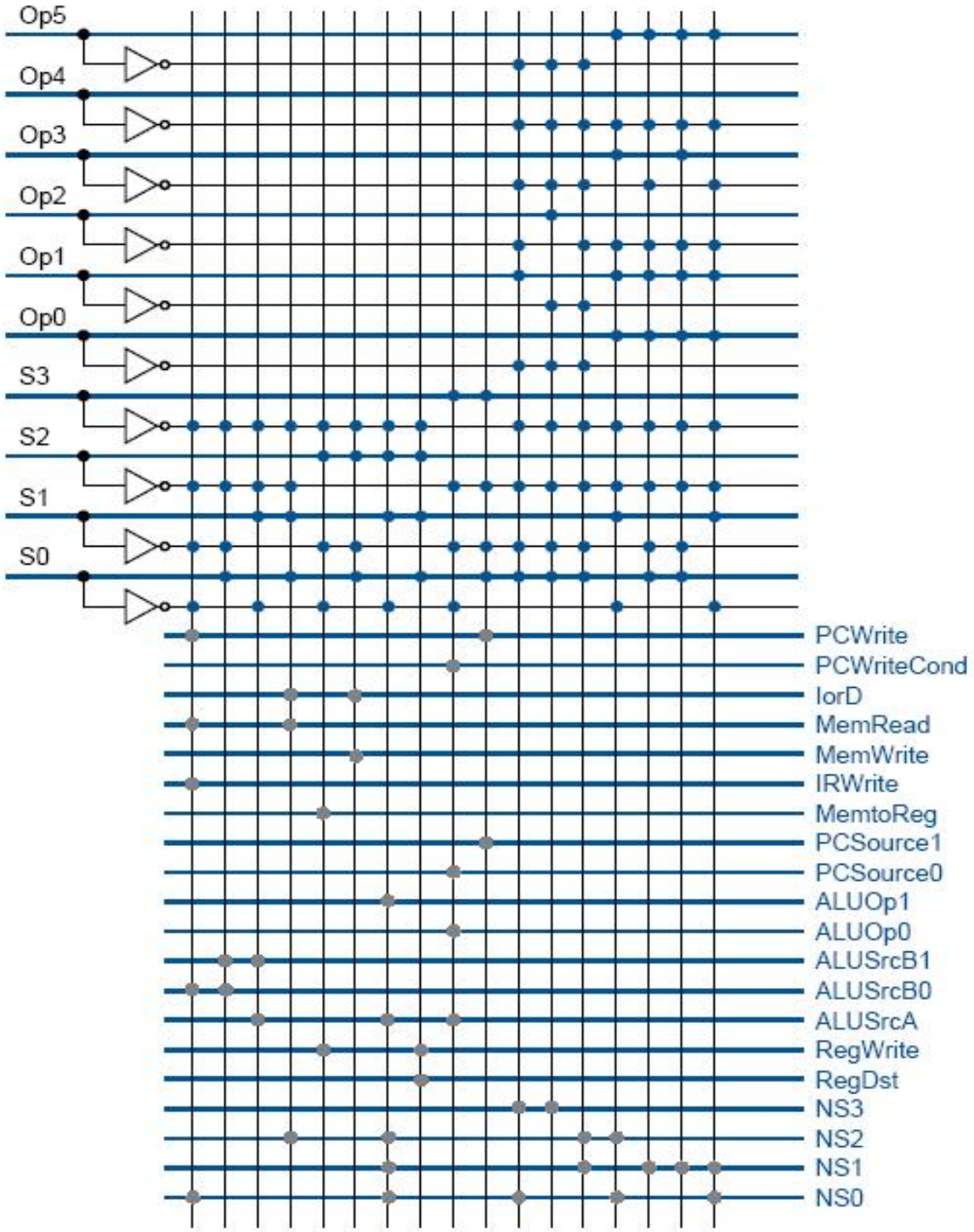


Figure C.3.9
of 3rd edition