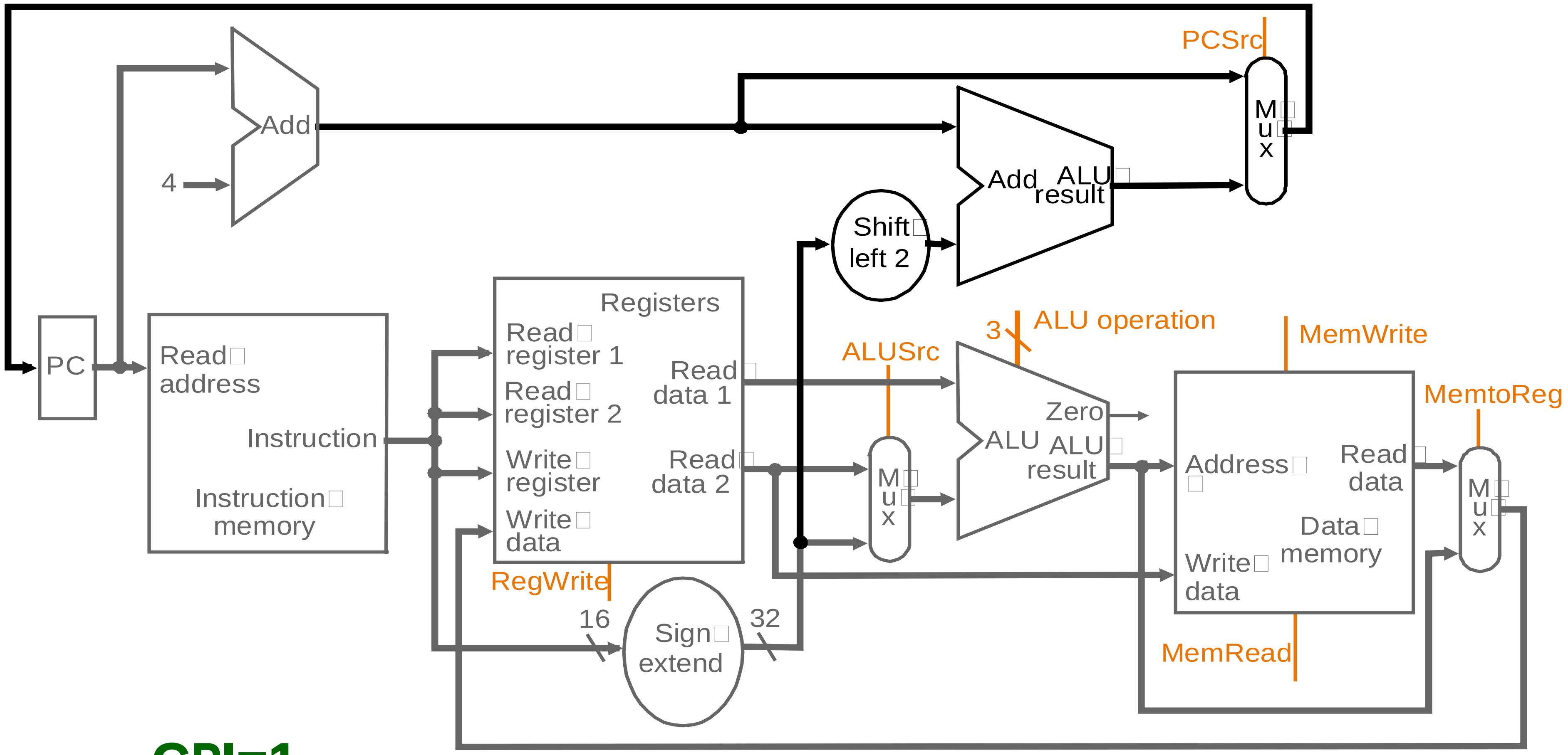
CS2006: 計算機組織

Designing a Multicycle Processor

Outline

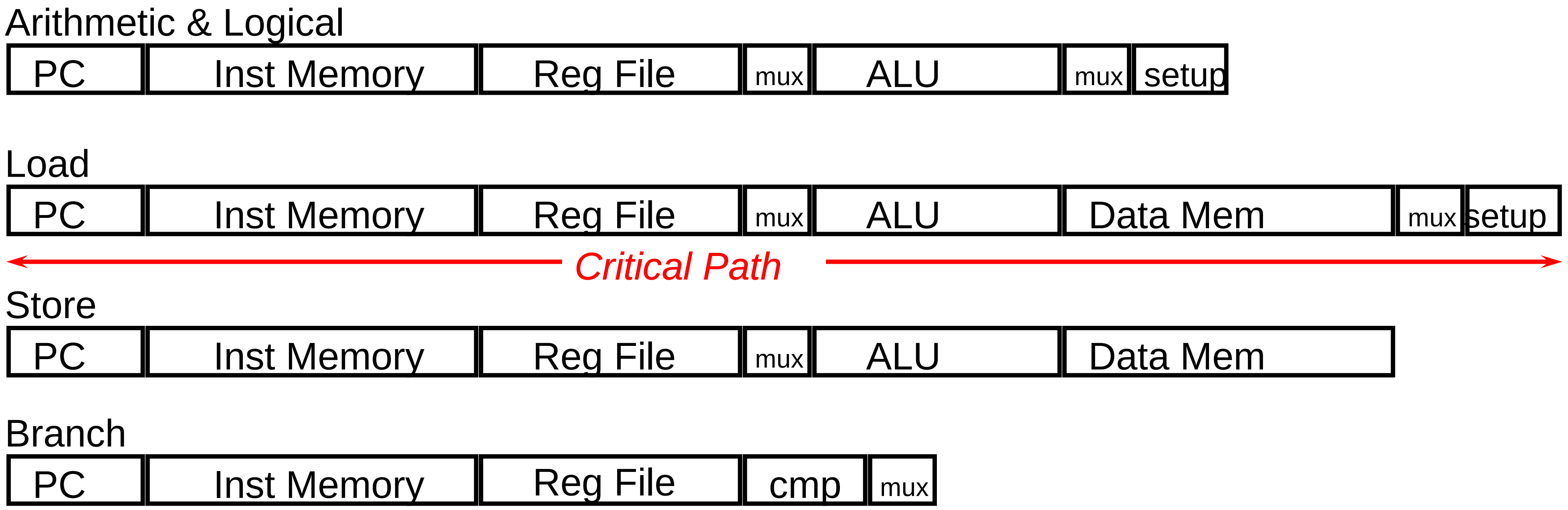
- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Recap: A Single-Cycle Processor



CPI=1

What's Wrong with Single-cycle?



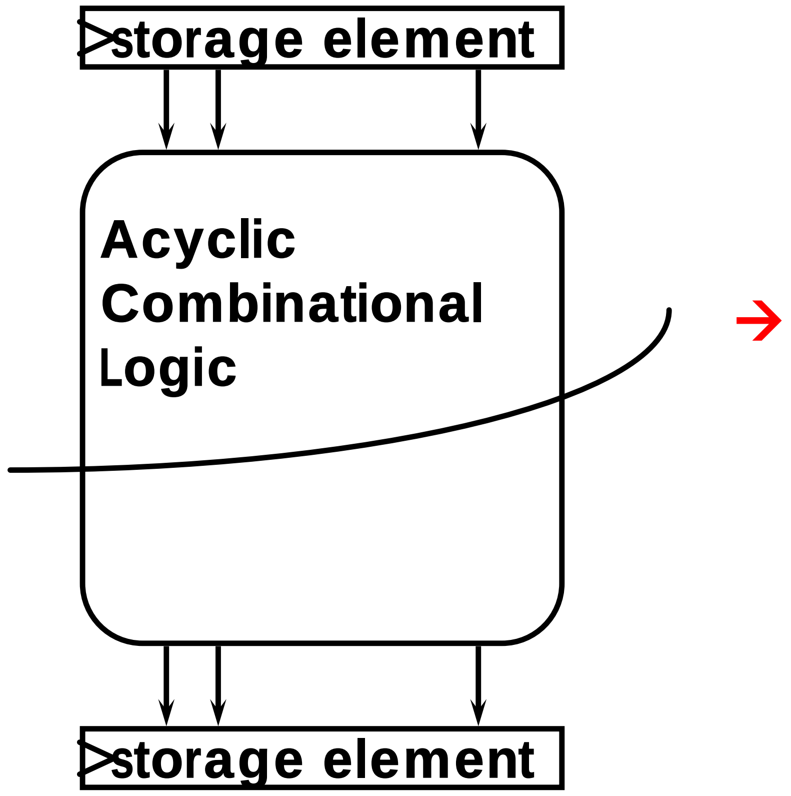
- **Long cycle time**
- **All instructions take same time as the slowest**
- **Real memory is not so ideal**
  - **cannot always get job done in one (short) cycle**
- **A FU can only be used once => higher cost**

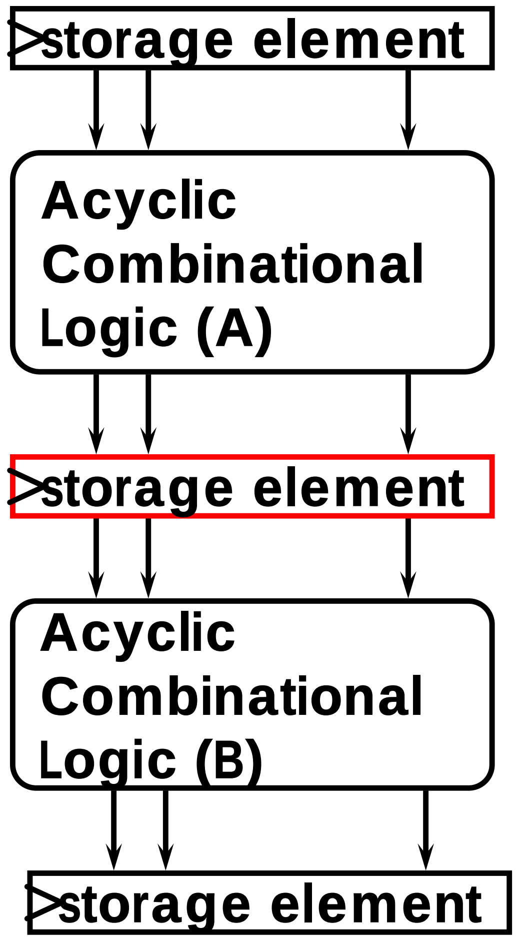
Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control (Appendix C.3)
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Multicycle Implementation

- Reduce cycle time
- Diff. Inst. take diff. cycles
- Share functional units



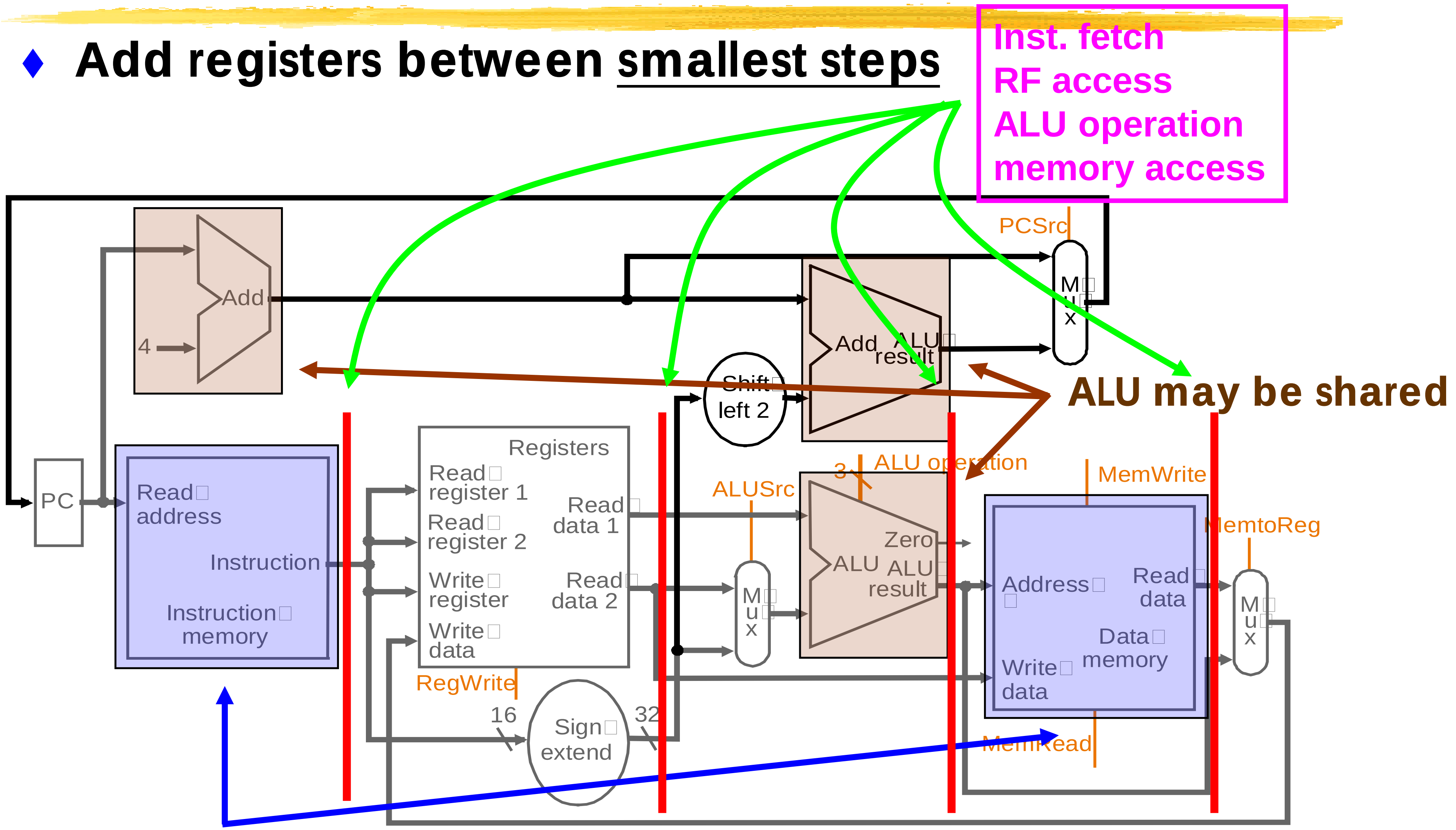

storage element
Acyclic Combinational Logic (A)
storage element
Acyclic Combinational Logic (B)
storage element

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional internal registers

Partition Single-Cycle Datapath

- Add registers between <u>smallest steps</u>



Inst./Data Memories are identical

Multicycle Datapath

- 1 memory (inst. & data), 1 ALU (addr, PC+4, add,...), registers (IR, MDR, A, B, ALUOut)
  - Storage for subsequent inst. (arch.-visible) vs. storage for same inst. but in a subsequent cycle

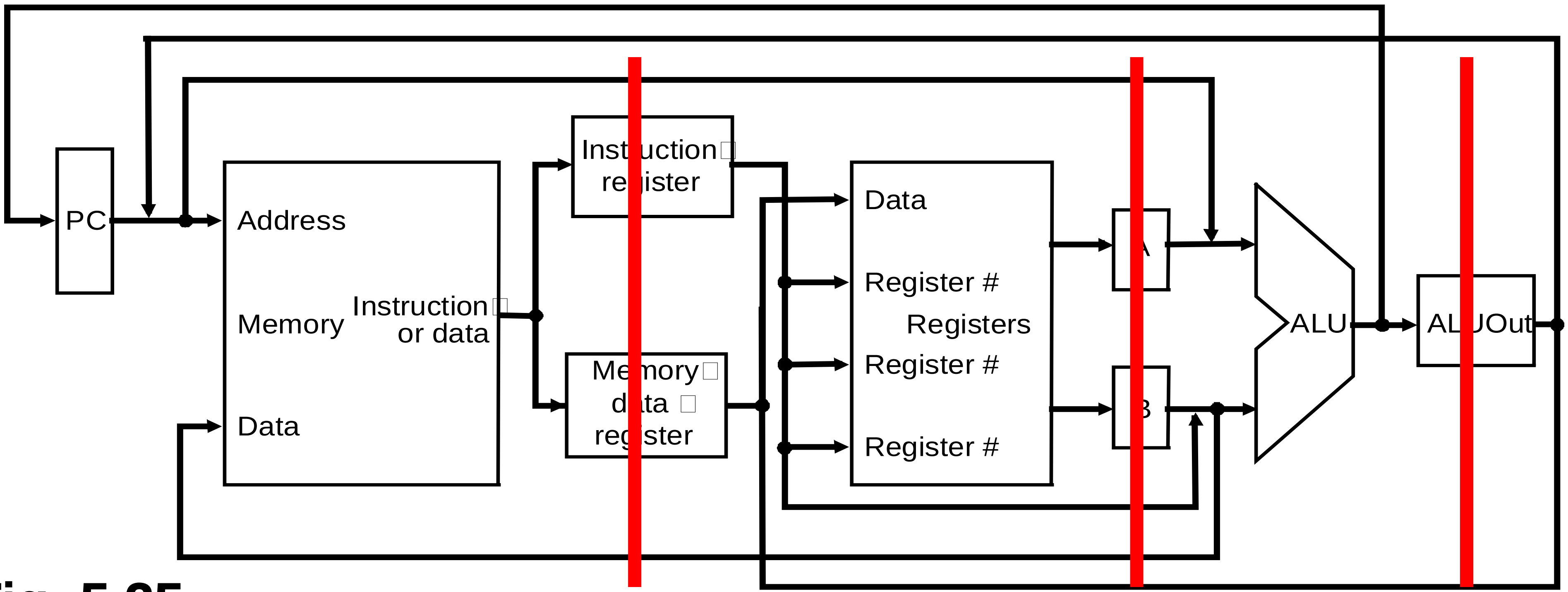


Fig. 5.25

Multicycle Datapath for Basic Inst.

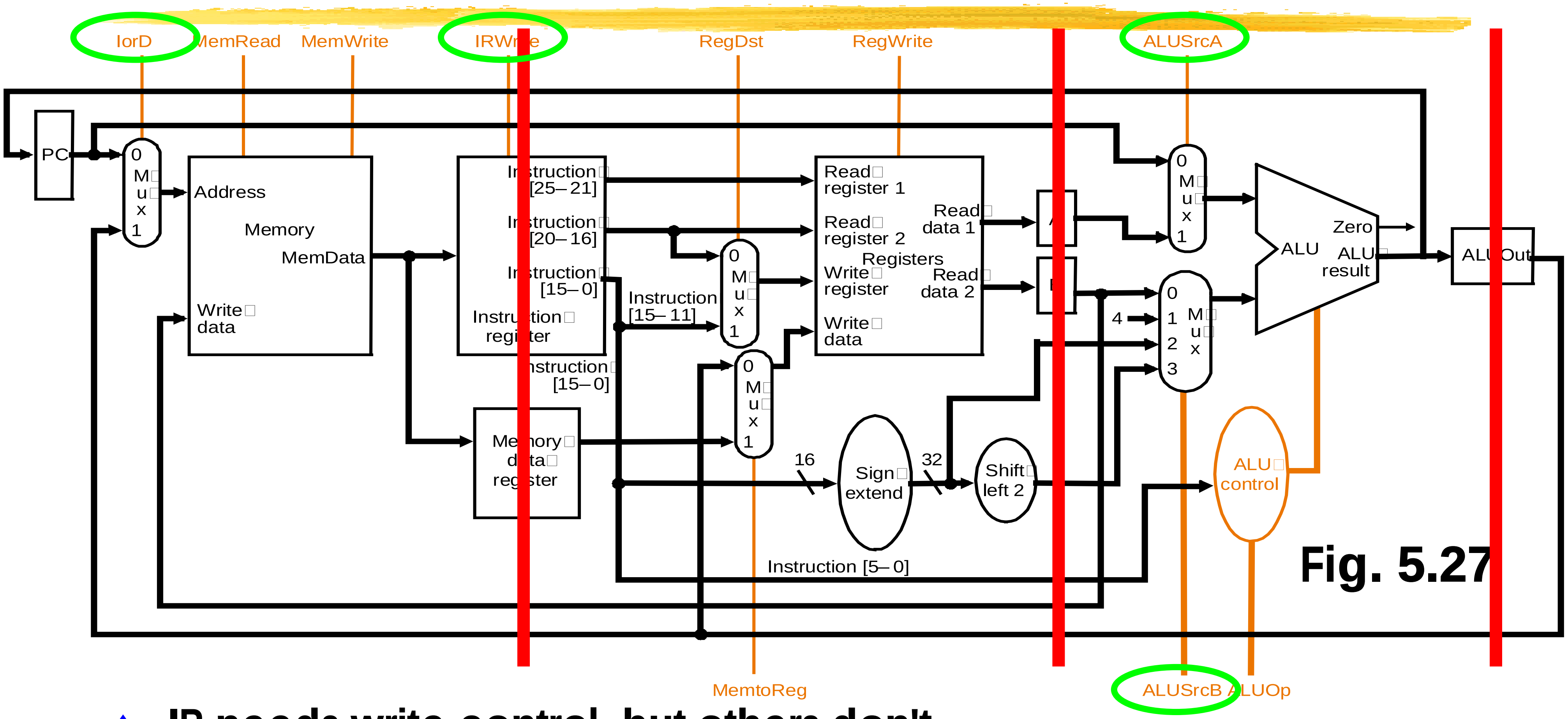
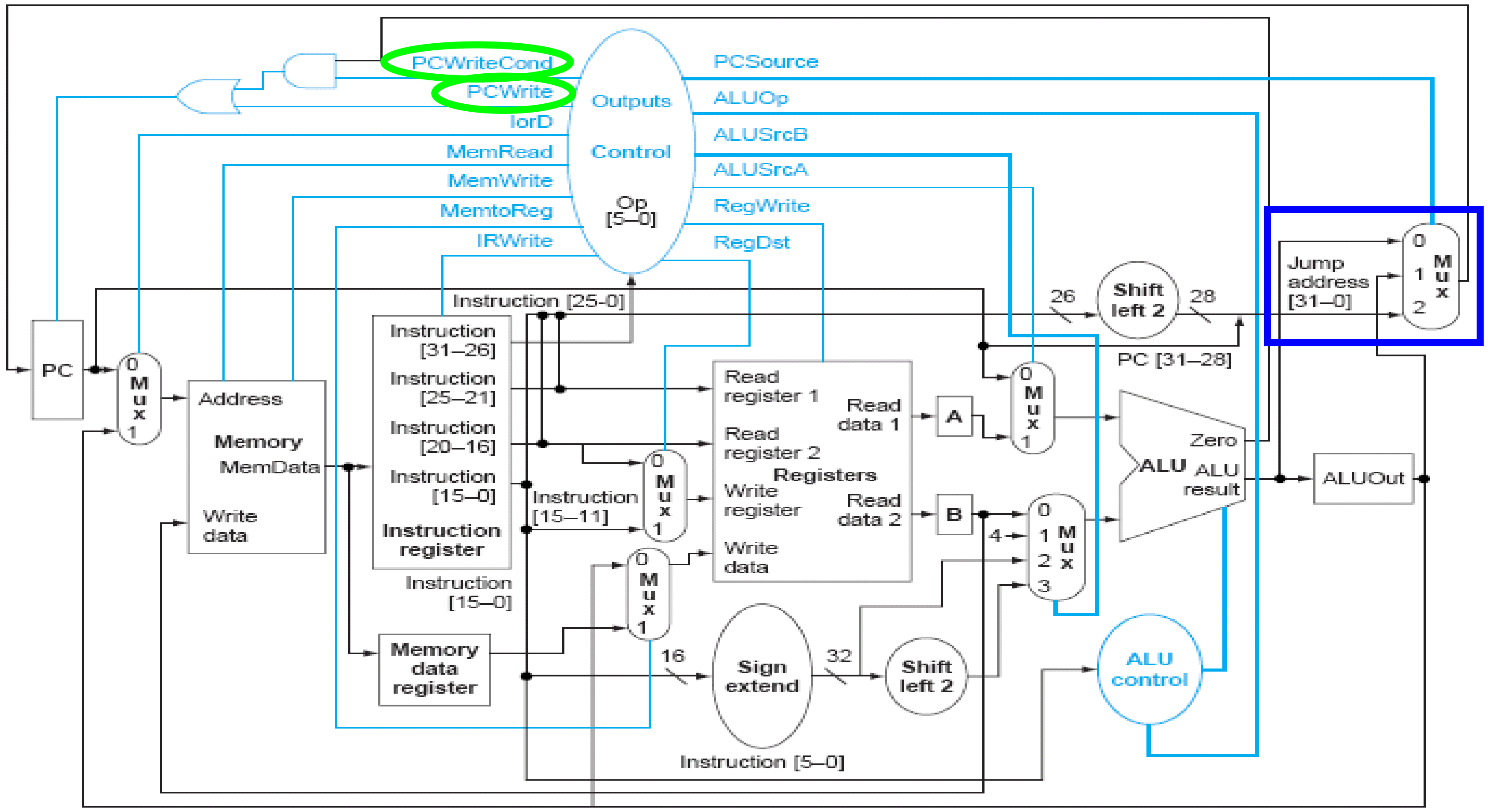


Fig. 5.27

- IR needs write control, but others don't
- MUX to select 2 sources to memory; memory needs read signal
- PC and A to one ALU input; four sources to another input

Adding Branch/Jump



- Three sources to PC
- Two PC write signals

Fig. 5.28

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control (Appendix C.3)
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type Instruction Completion
- Memory Read Completion (Write-back)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register (IR)
- Increment the PC by 4 and put the result back in the PC
- Can be described succinctly using RTL (*Register-Transfer Language*)

```
    IR = Memory[PC];
    PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt (not harmful if not needed)
- Compute the branch address (not harmful if not needed)
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut=PC+(sign-ext(IR[15-0])<<2);
```

We aren't setting any control lines based on the instruction type yet
(we are busy "decoding" it in control logic)

Step 3: Execution

ALU is performing one of three functions, based on instruction type:

- **Memory Reference:**
  `ALUOut = A + sign-extend(IR[15-0]);`

- **R-type:**
  `ALUOut = A op B;`

- **Branch:**
  `if (A==B) PC = ALUOut;`

Step 4: R-type or Memory-access

- **Loads and stores access memory**

```
MDR = Memory[ALUOut];
     or
Memory[ALUOut] = B;
```

- **R-type instructions finish**

```
Reg[IR[15-11]] = ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Step 5: Write-back

- Loads write to register

```
Reg[IR[20-16]]= MDR;
```

What about all the other instructions?

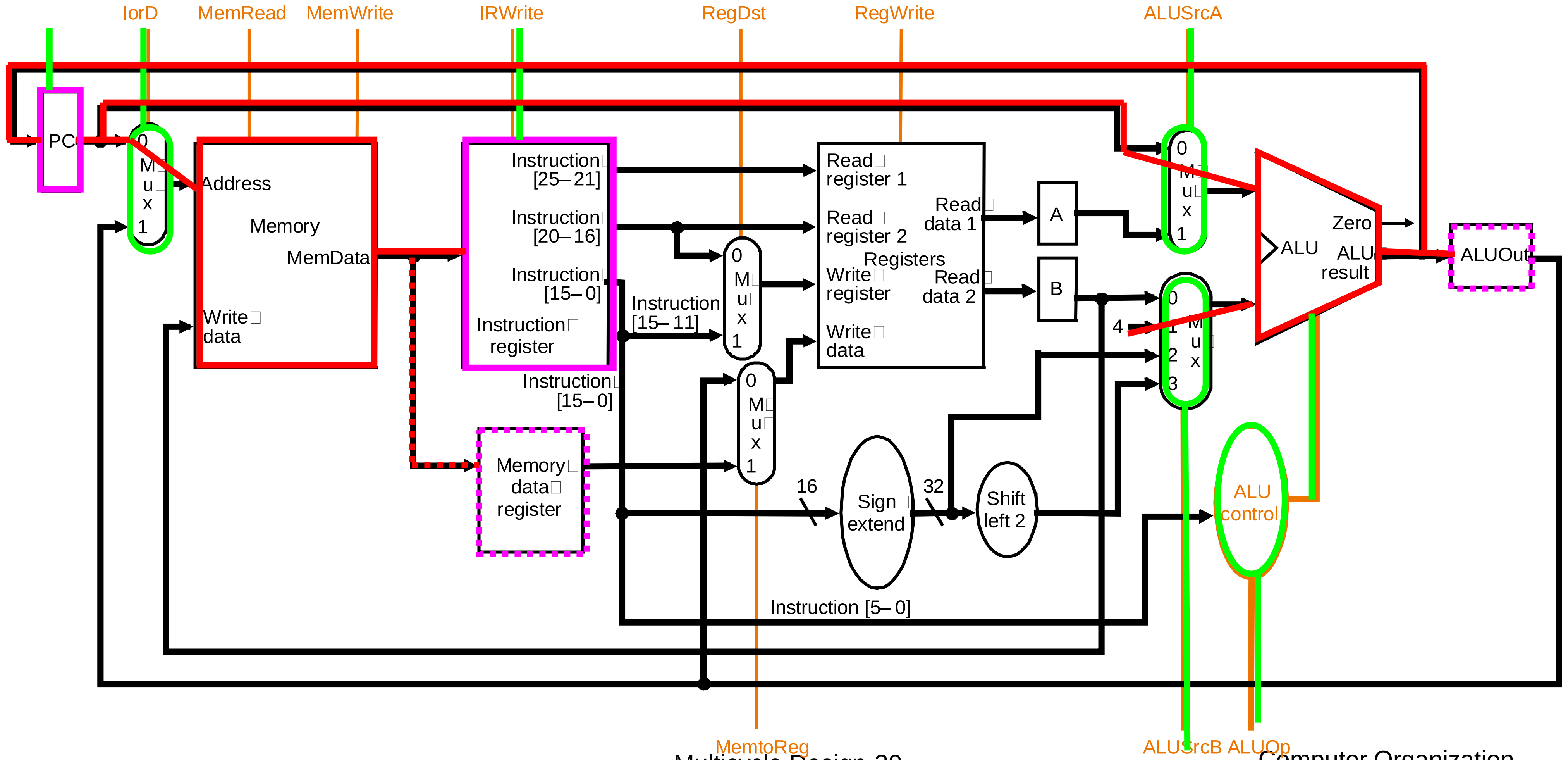
Summary of the Steps

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Fig. 5.30

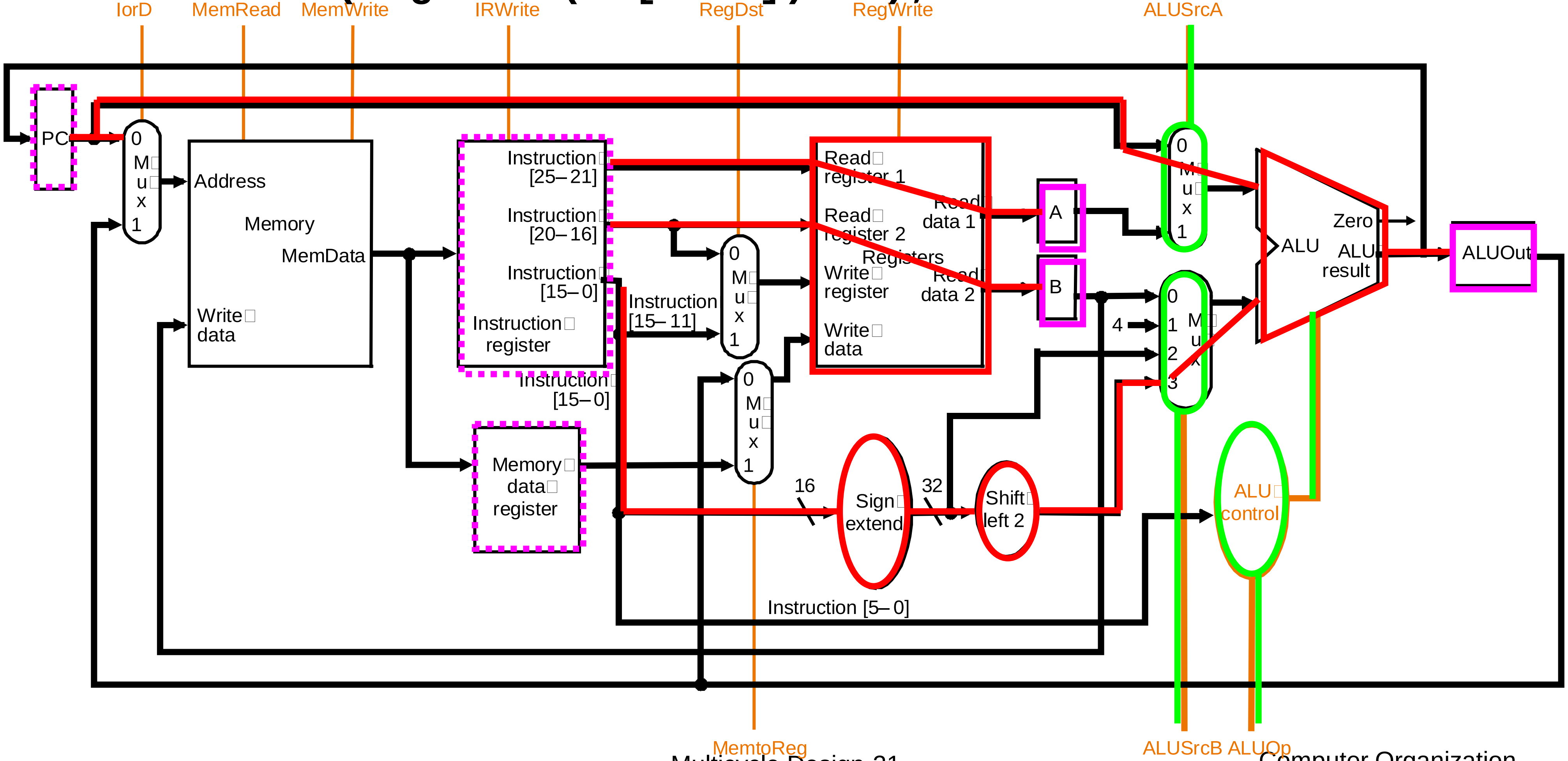
Cycle 1 of add

IR = Memory[PC];   PC = PC + 4;



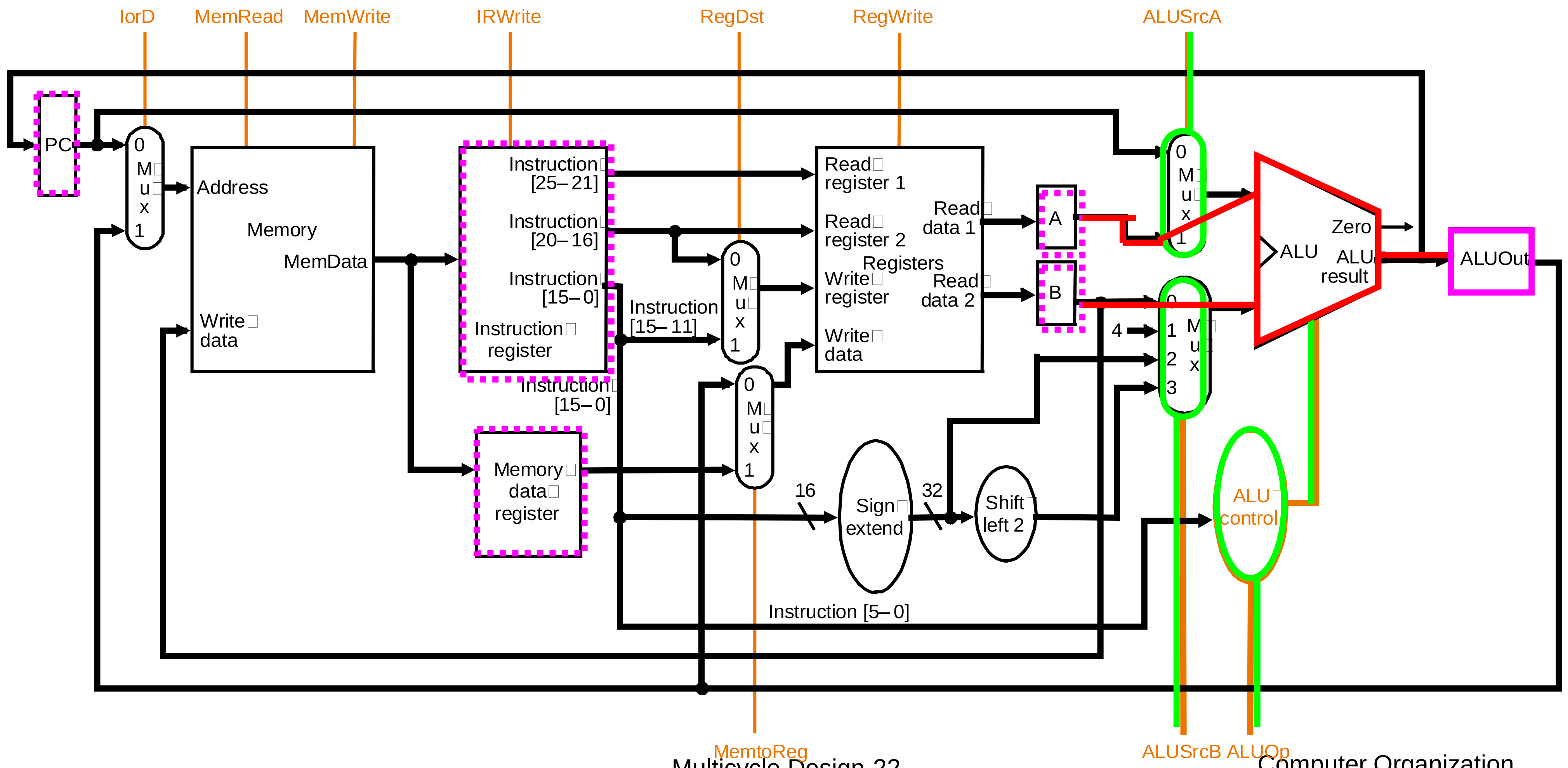
Cycle 2 of add

```
A=Reg[IR[25-21]]; B=Reg[IR[20-16]];
ALUOut=PC+(sign-ext(IR[15-0])<<2);
```



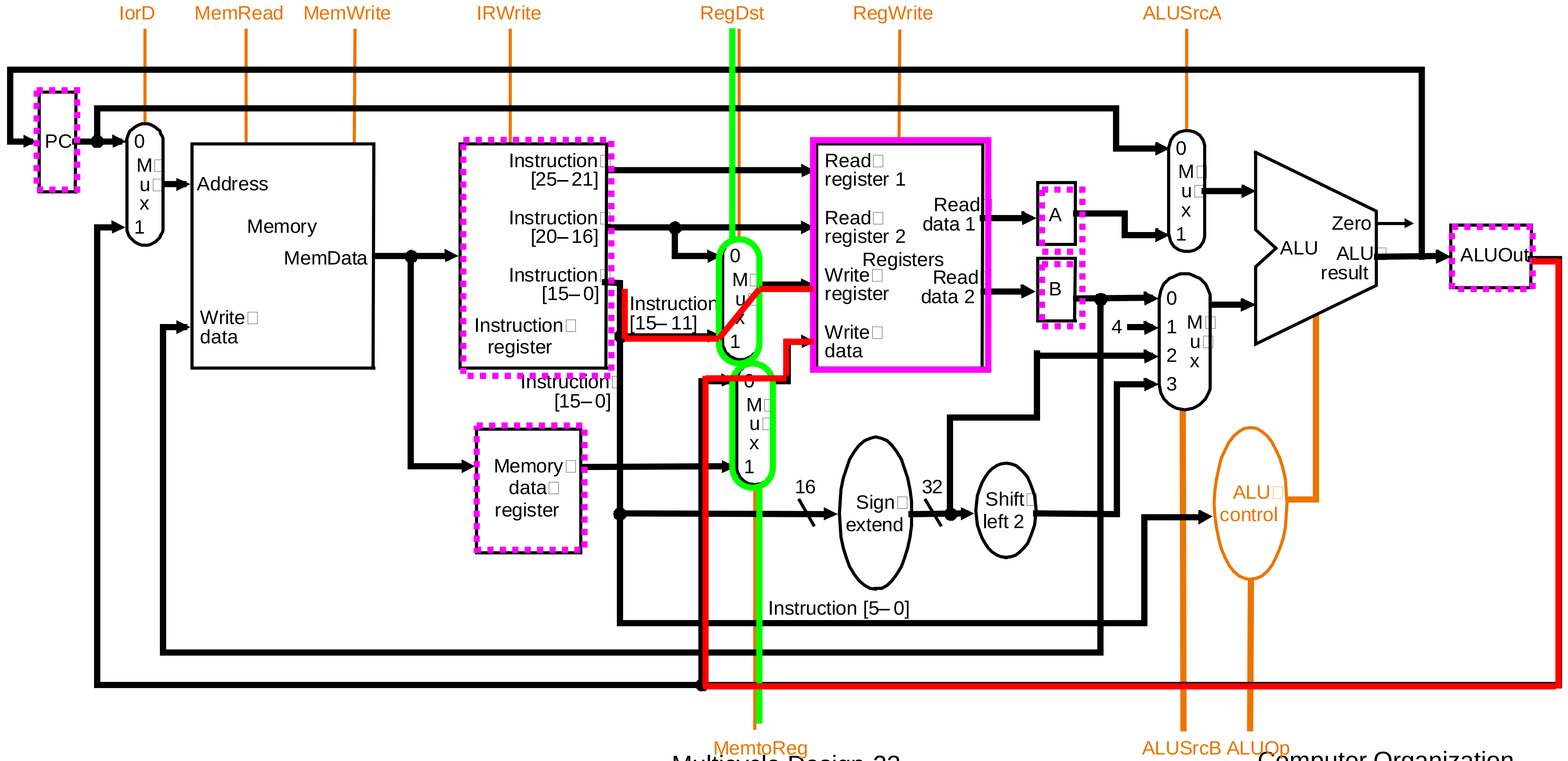
Cycle 3 of add

```
ALUOut = A op B;
```



Cycle 4 of add

```
Reg[IR[15-11]] = ALUOut;
```



Simple Questions

- How many cycles will it take to execute this code?

```
          lw $t2, 0($t3)
          lw $t3, 4($t3)
          beq $t2, $t3, Label
          add $t5, $t2, $t3
          sw $t5, 8($t3)
Label:    ...
```

assume not taken

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?

Outline

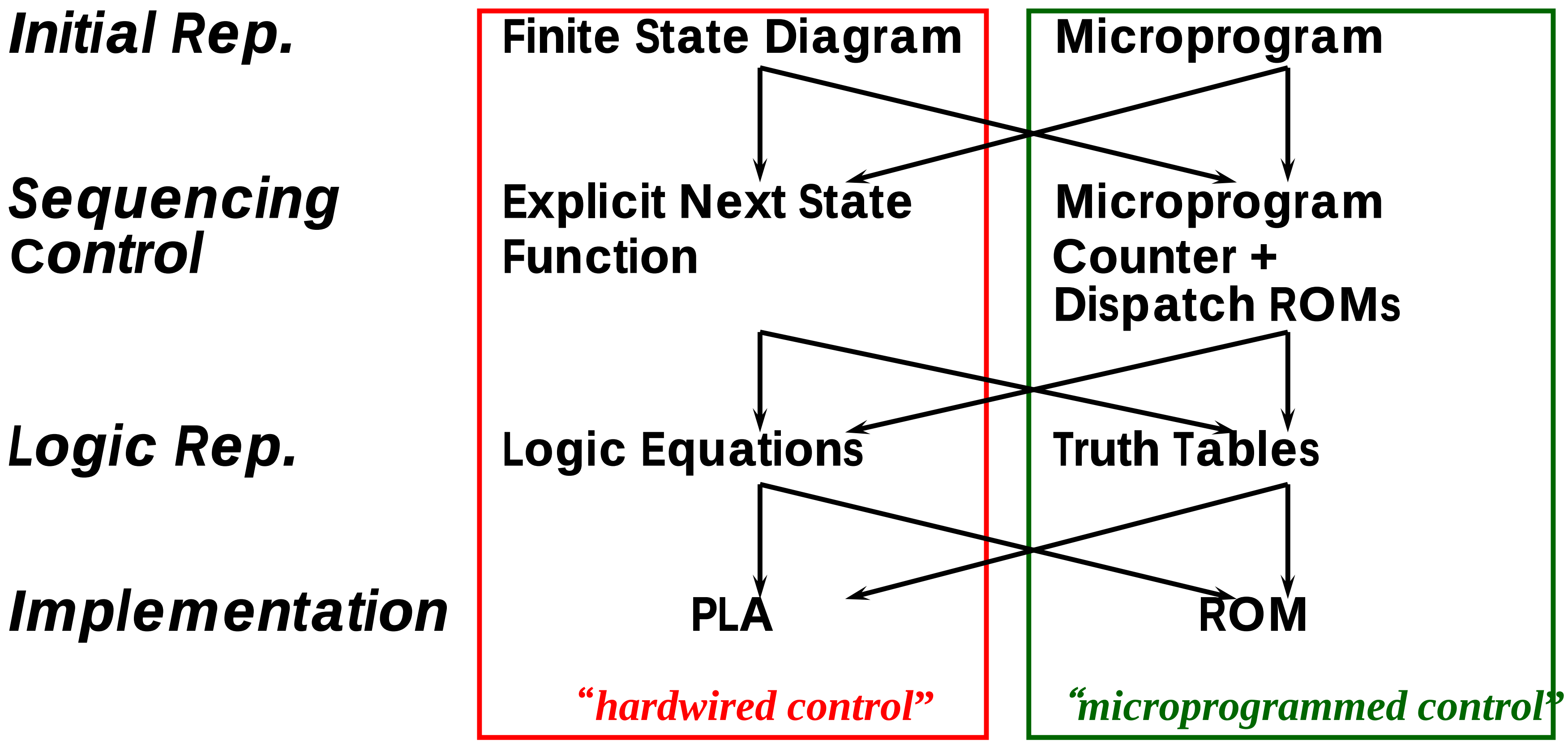
- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control (Appendix C.3)
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
    - Control must specify both the signals to be set in any step and the next step in the sequence
- Control specification
  - Use a finite state machine (graphically)
  - Use microprogramming
- Implementation can be derived from the specification and use gates, ROM, or PLA

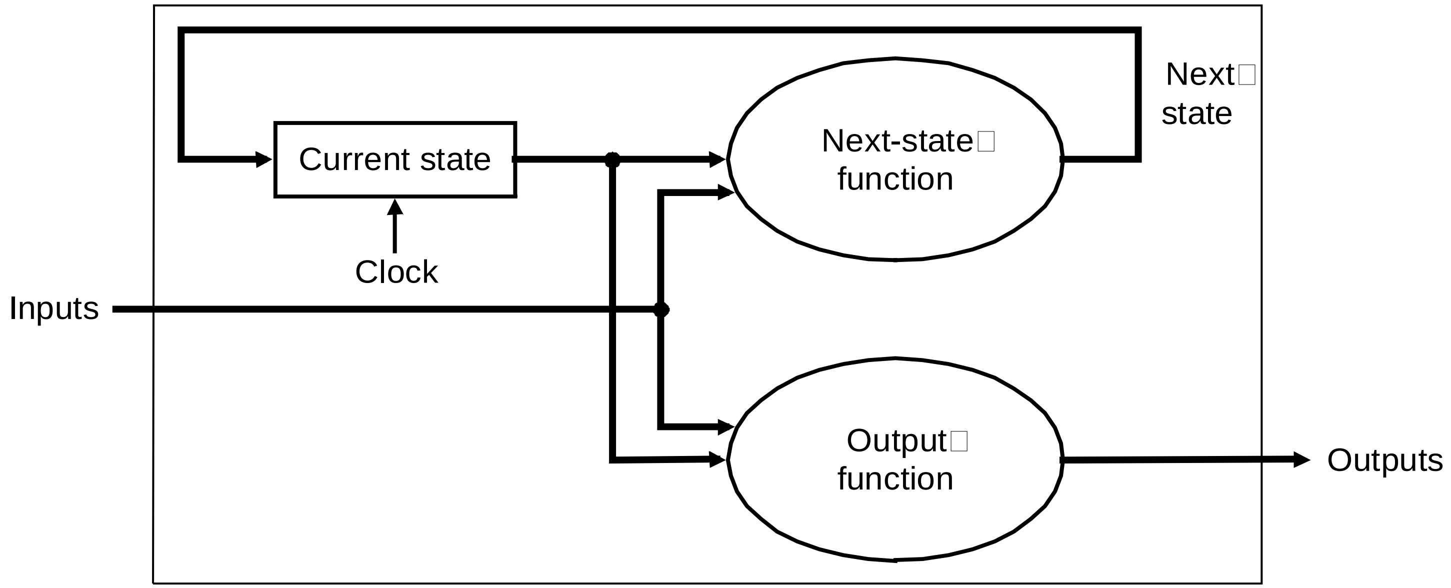
Controller Design: An Overview

- Several possible initial representations, sequence control and logic representation, and control implementation => all may be determined indep.



Review: Finite State Machines

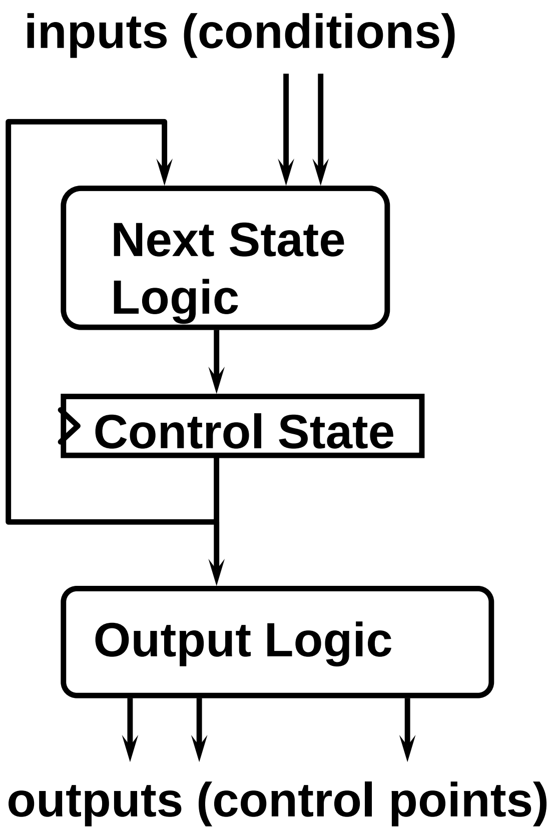
- **Finite state machines:**
  - **a set of states and**
  - **next state (set by current state and input)**
  - **output (set by current state and possibly input)**

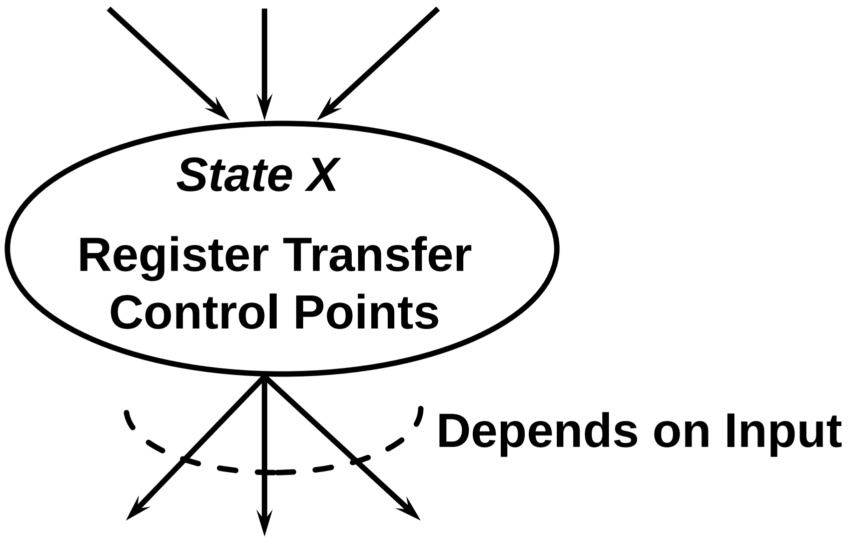


  - **We will use a *Moore Machine* (output based only on the current state)**

Our Control Model

- State specifies control points for RT
- Transfer at exiting state (same falling edge)
- One state takes one cycle





Summary of the Steps

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

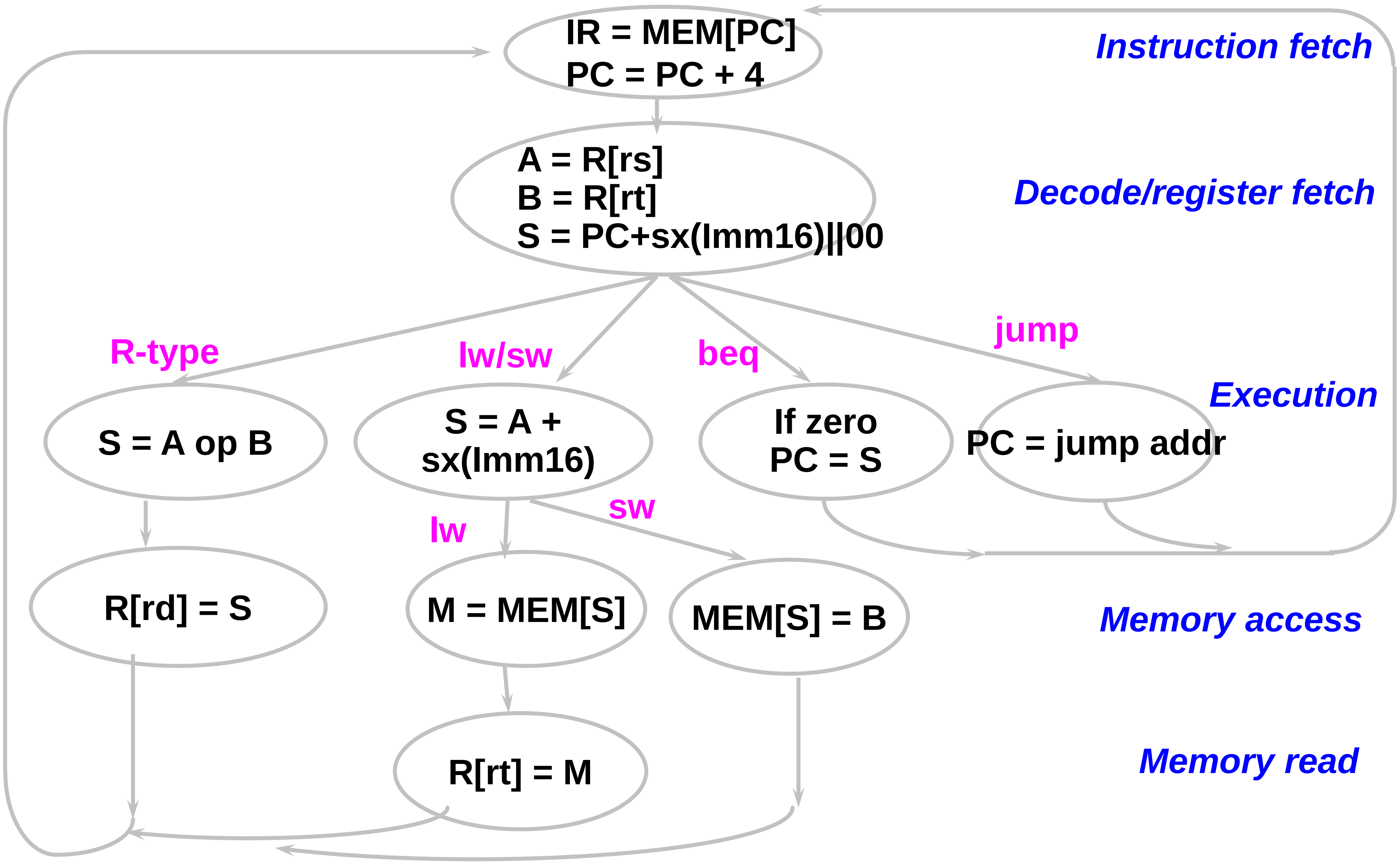
Instruction fetch	IR = Memory[PC]
	PC = PC + 4			

Instruction decode/register fetch	A = Reg [IR[25-21]]
	B = Reg [IR[20-16]]
	ALUOut = PC + (sign-extend (IR[15-0]) << 2)			

Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Fig. 5.30

Control Specification for Multicycle



Organization of Multicycle Processor

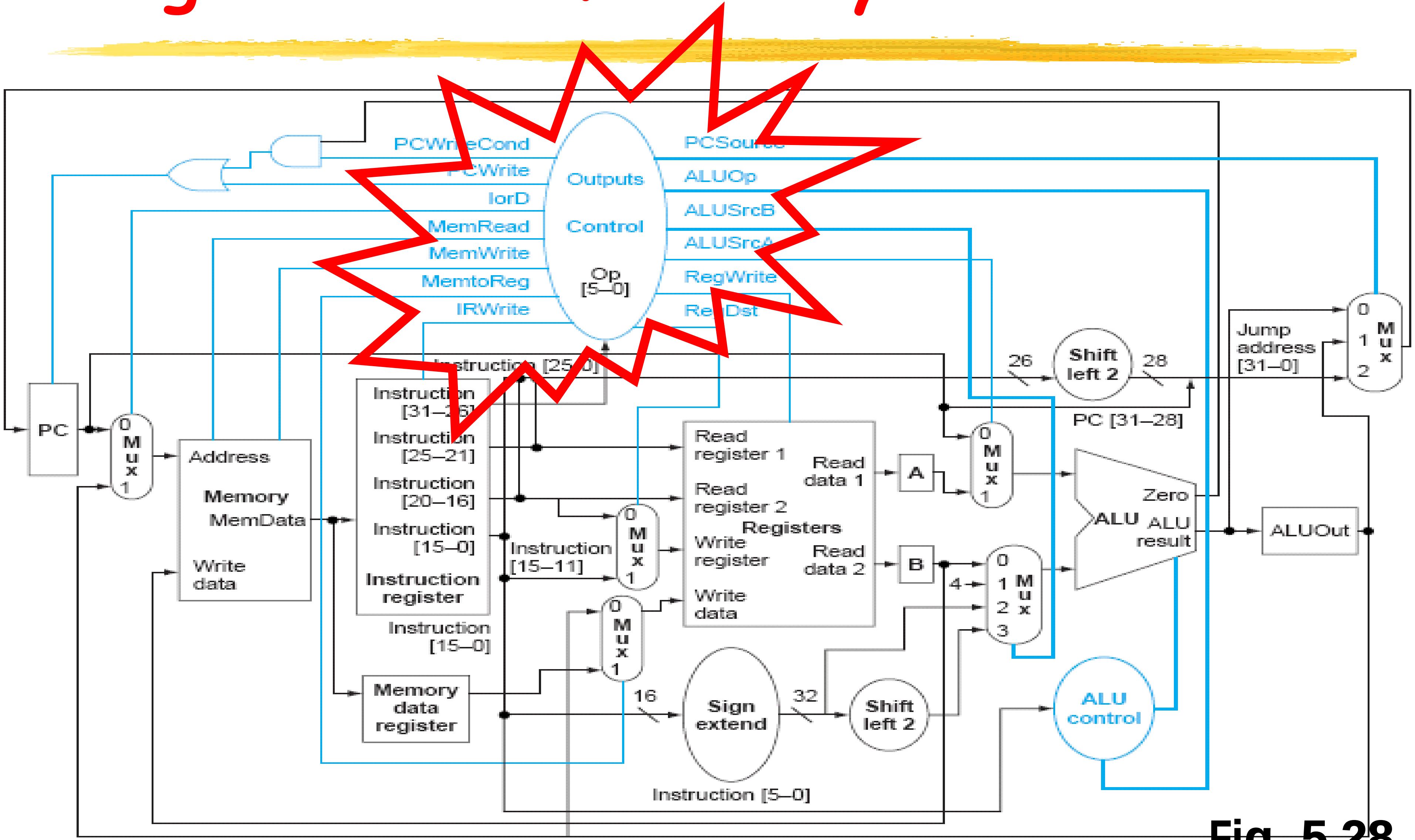


Fig. 5.28

Control Signals

Single Bit Control

Signal name	*Effect when deasserted*	*Effect when asserted*
ALUSrcA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg file is written
MemtoReg	Reg. data input = ALU	Reg. write data input = MDR
RegDst	Reg. write dest. no. = rt	Reg. write dest. no. = rd
MemRead	None	Memory at address is read
MemWrite	None	Memory at address is written
IorD	Memory address = PC	Memory address = ALUout
IRWrite	None	IR = Memory
PCWrite	None	PC = PCSource
PCWriteCond	None	If zero then PC = PCSource

Multiple Bit Control

Signal name	*Value*	*Effect*
ALUOp	00	ALU adds
	01	ALU subtracts
	10	ALU operates according to func code
ALUSrcB	00	2nd ALU input = B
	01	2nd ALU input = 4
	10	2nd ALU input = sign extended IR[15-0]
	11	2nd ALU input = sign ext., shift left 2 IR[15-0]
PCSource	00	PC = ALU (PC + 4)
	01	PC = ALUout (branch target address)
	10	PC = PC+4[31-28] : IR[25-0] << 2

Fig. 5.29

Mapping RT to Control Signals

- High-level view of the finite state machine control

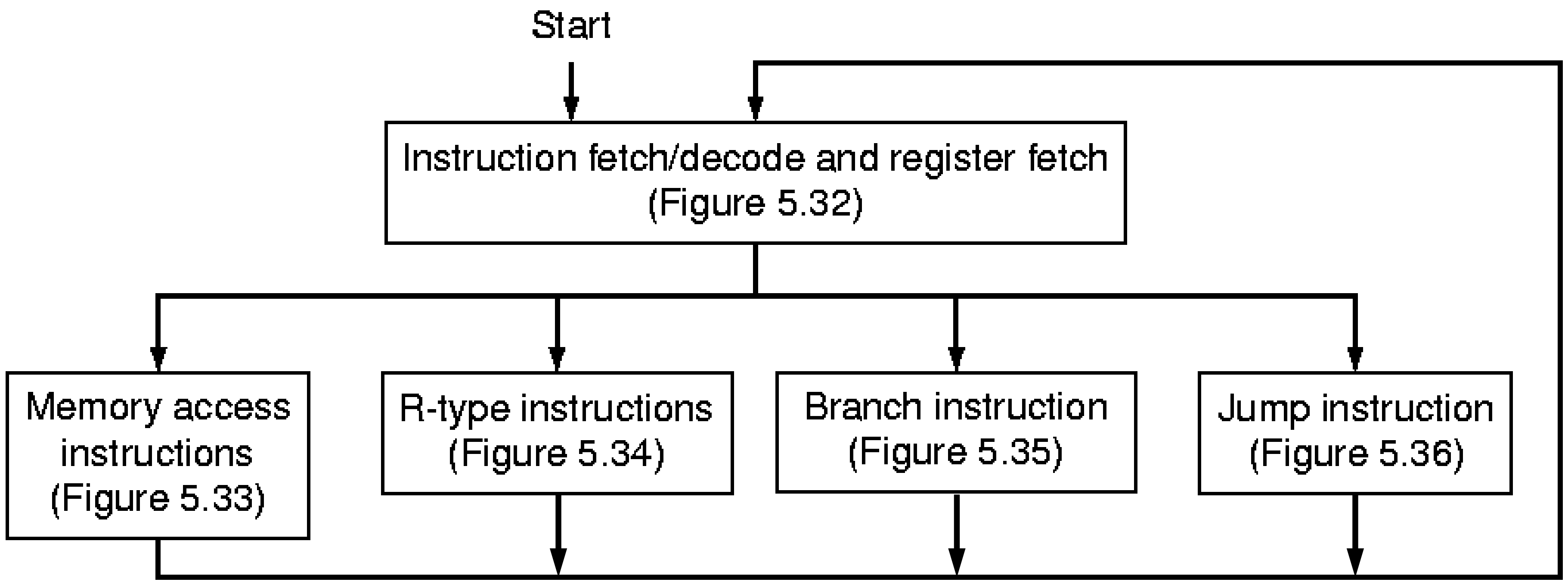

Start
Instruction fetch/decode and register fetch (Figure 5.32)
Memory access instructions (Figure 5.33)
R-type instructions (Figure 5.34)
Branch instruction (Figure 5.35)
Jump instruction (Figure 5.36)

Fig. 5.31

Mapping RT to Control Signals

- Instruction fetch and decode portion of every instruction is identical:

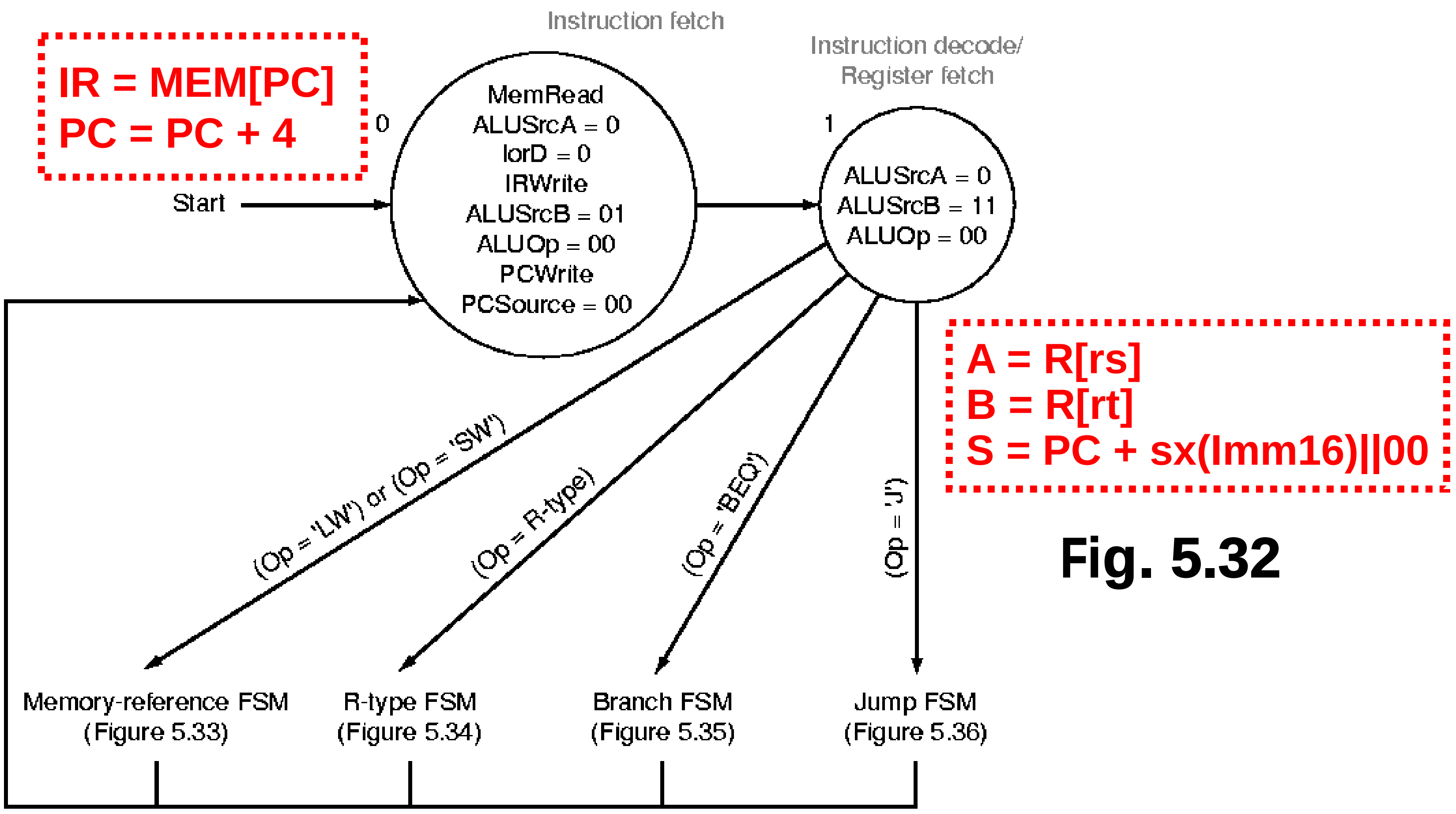


Fig. 5.32

Mapping RT to Control Signals

- FSM for controlling memory reference instructions:

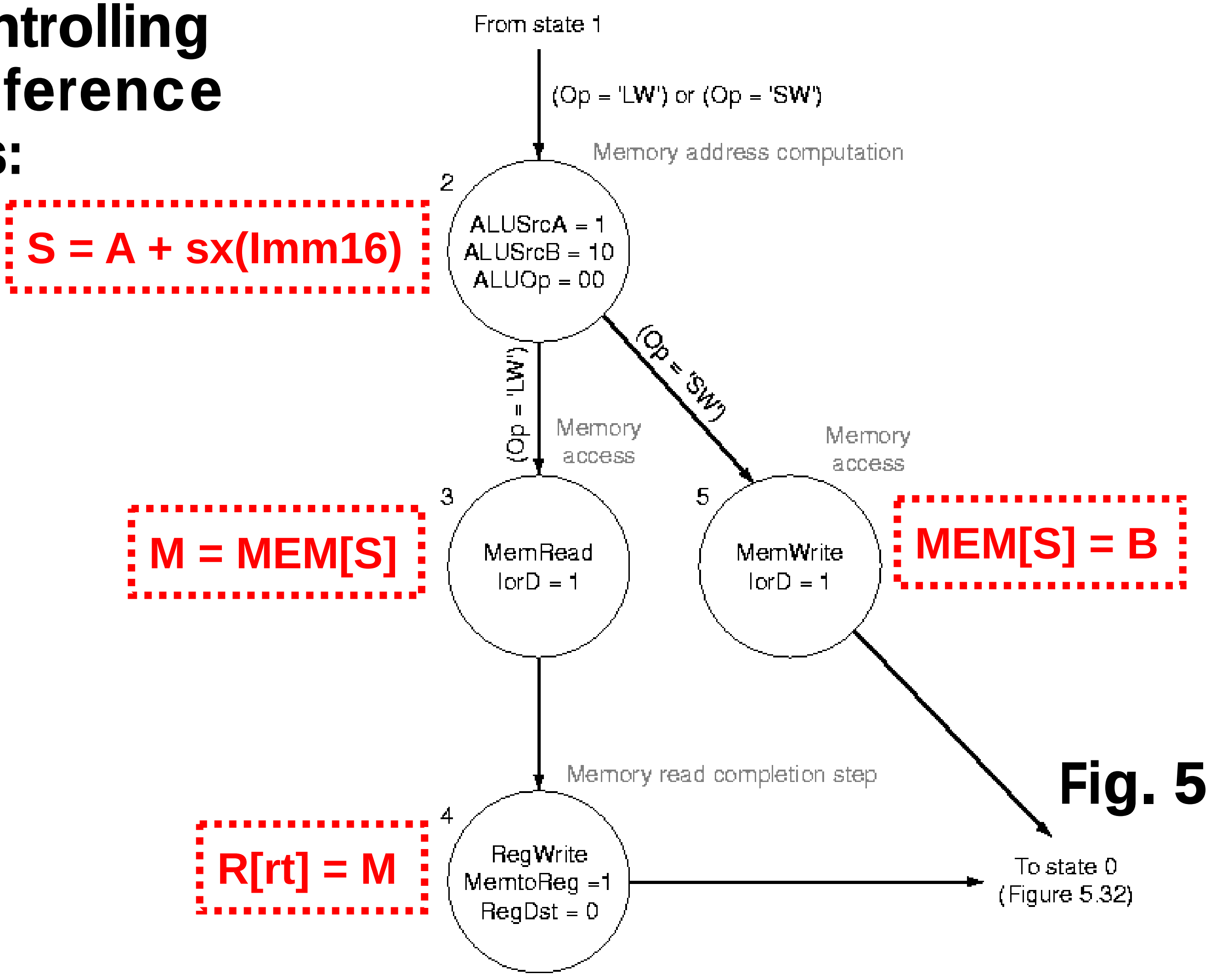


Fig. 5.33

Mapping RT to Control Signals

- FSM for controlling R-type instructions:

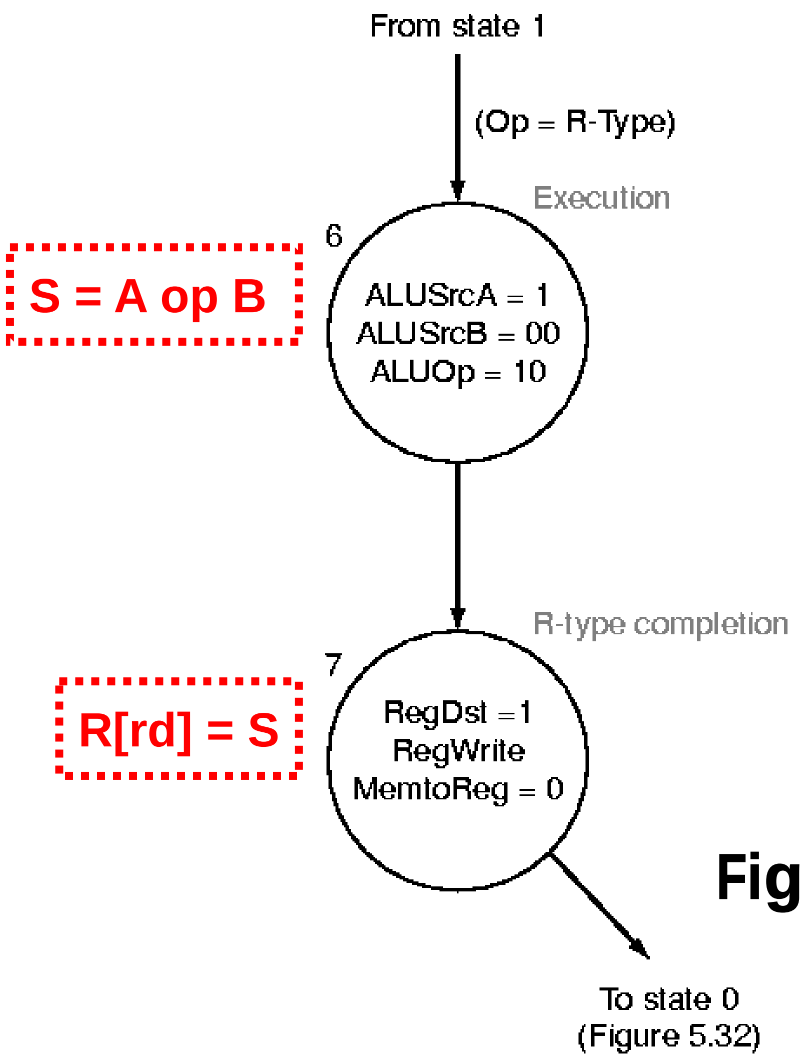


Fig. 5.34

Mapping RT to Control Signals

- FSM for controlling branch instruction:

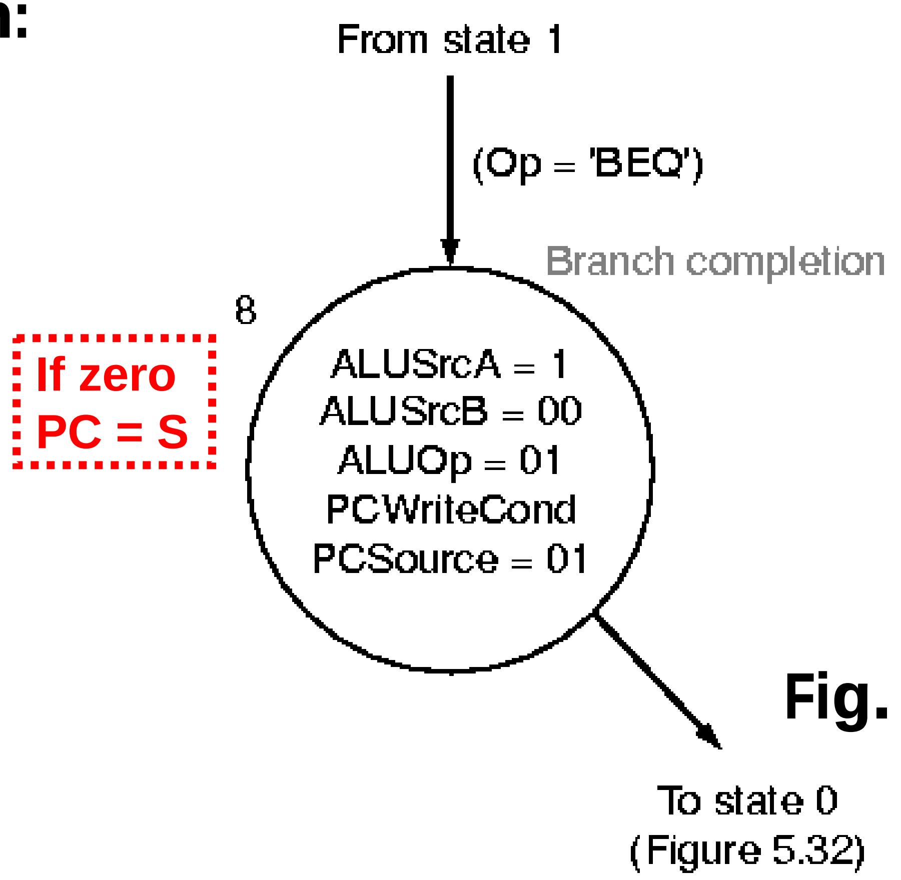


Fig. 5.35

Mapping RT to Control Signals

- FSM for controlling jump instruction:

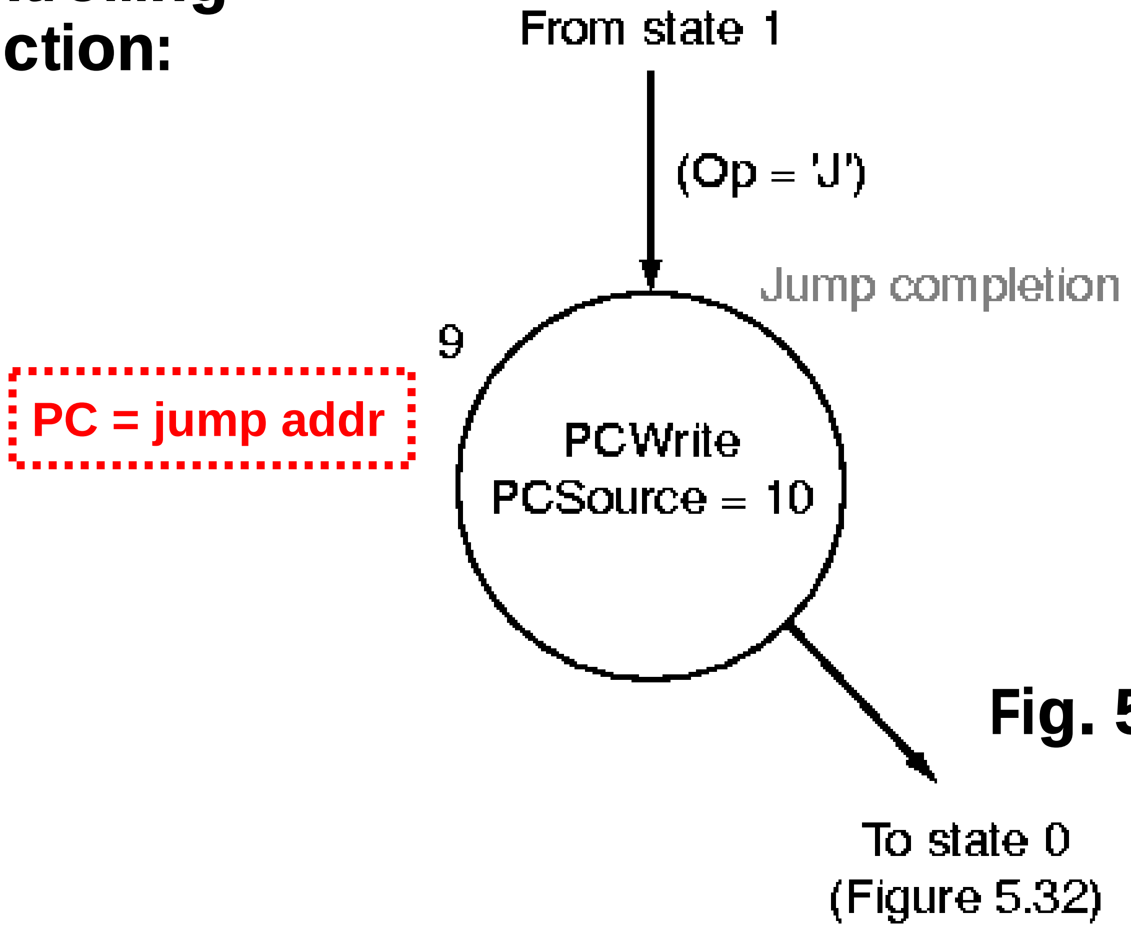


Fig. 5.36

Complete FSM

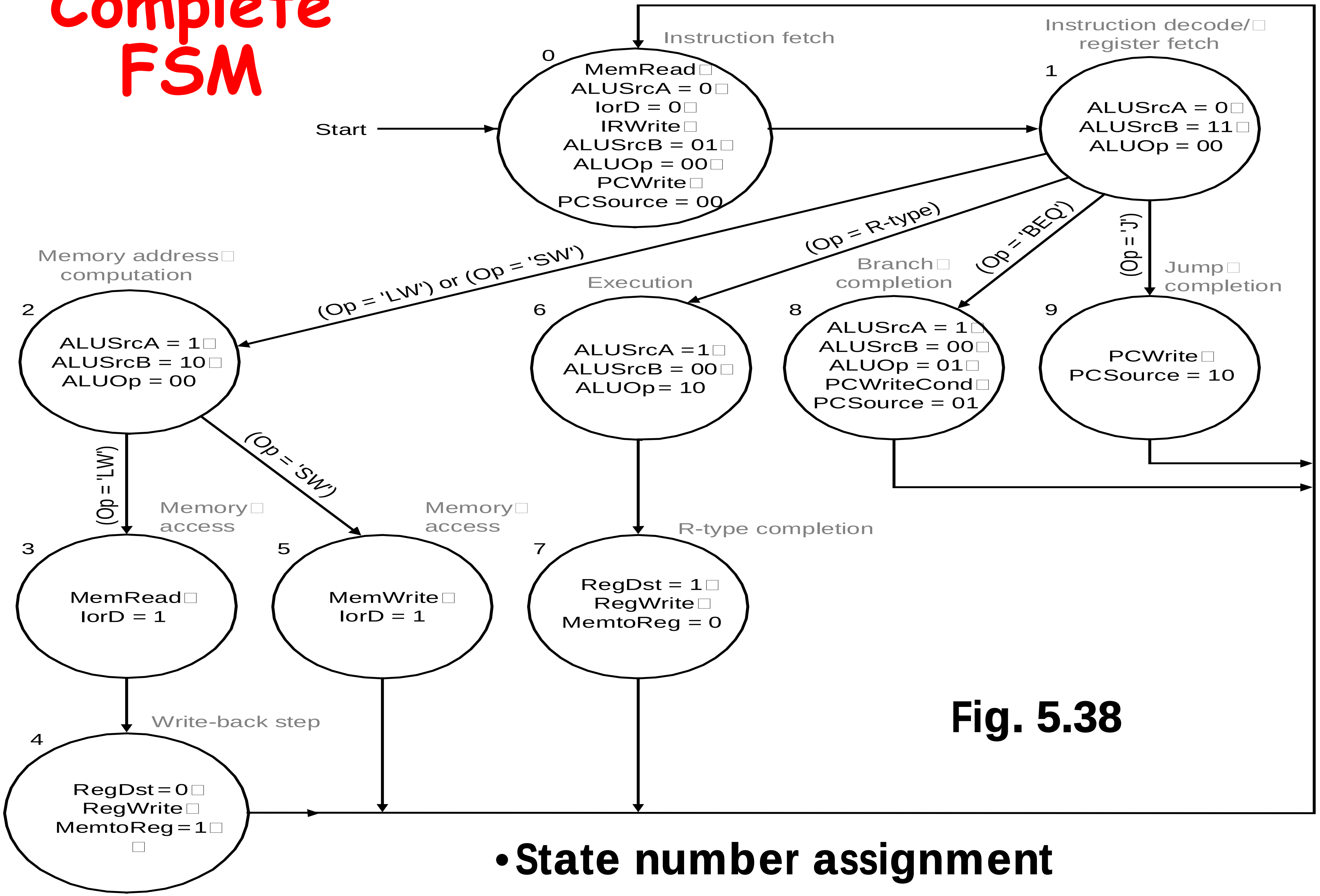


Fig. 5.38

- State number assignment

Truth Table

	input		output	
	op	S	Datapath control	NS
R-type	000000	0000	1001010000001000	0001
	000000	0001	0000000000011000	0110
	000000	0010	0000000000010100	xxxx
	⋮			
	000000	1010		
	⋮			
	000000	1111		
	000001	0000		
	⋮			
Jump	000010	0000	1001010000001000	0001
	000010	0001	0000000000011000	1001
	000010	0010	0000000000010100	xxxx
	⋮			

From FSM to Truth Table

- Please reference the logic equations in Fig. C.3.3 and the truth table in Fig. C.3.6

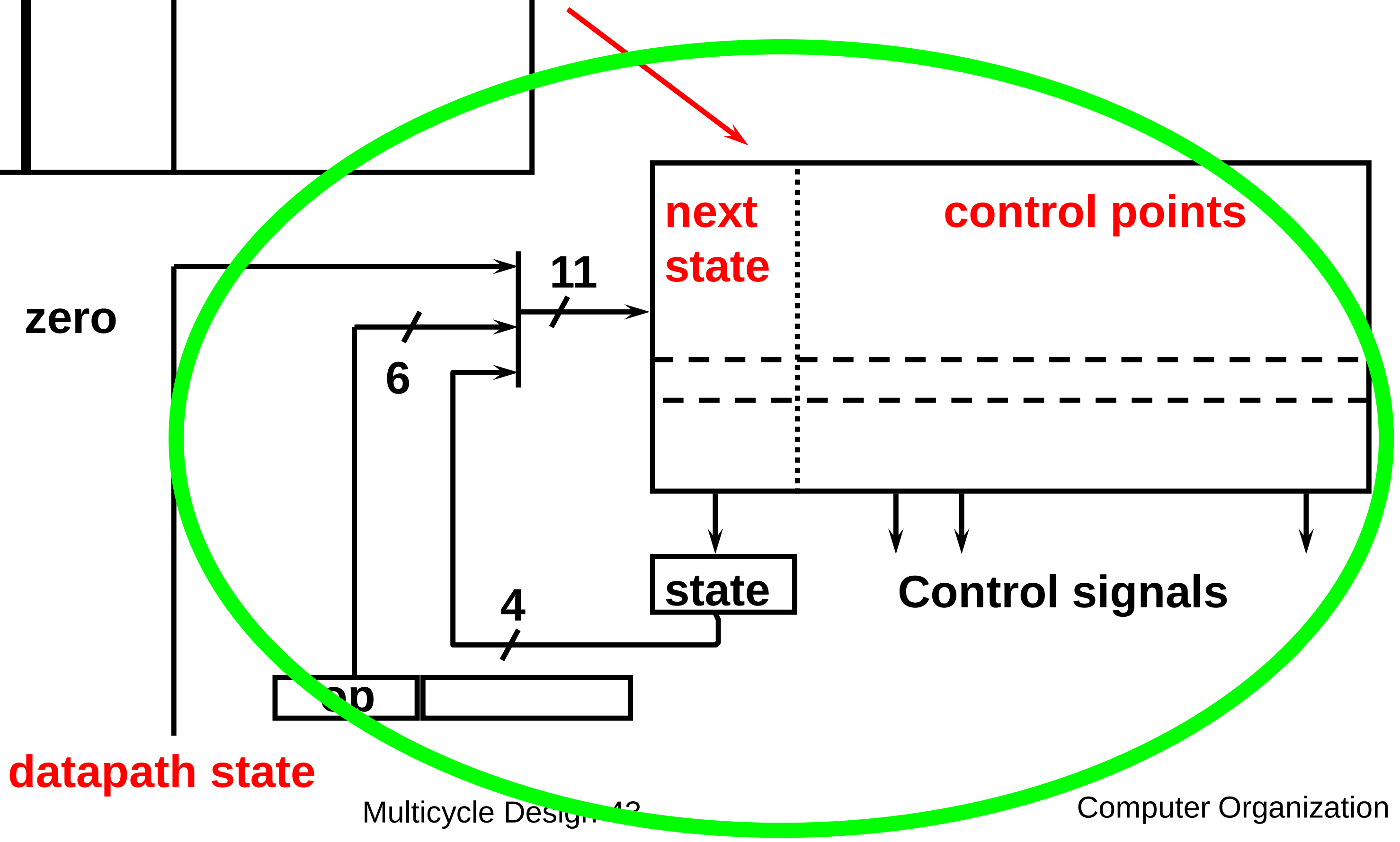
Output	Equation
PCWrite	state0 + state9
PCWriteCond	state8
IorD	state3 + state5
...	...
NextState0	
NextState1	
NextState2	
NextState3	
...	

Output	Current states									
	0	1	2	3	4	5	6	7	8	9
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
...	...									

Designing FSM Controller

state	op	cond	next state	control points

Truth Table



The Control Unit

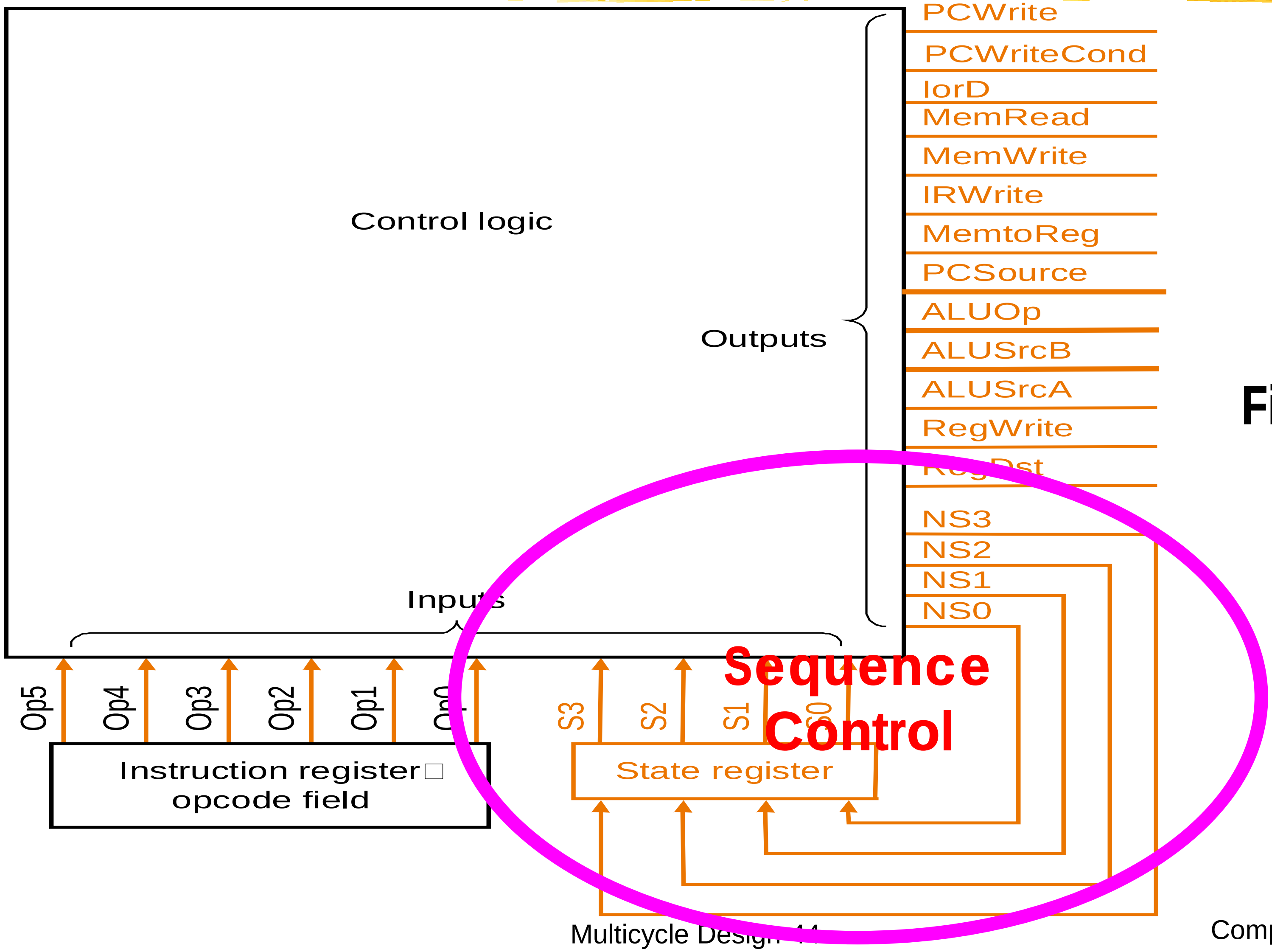
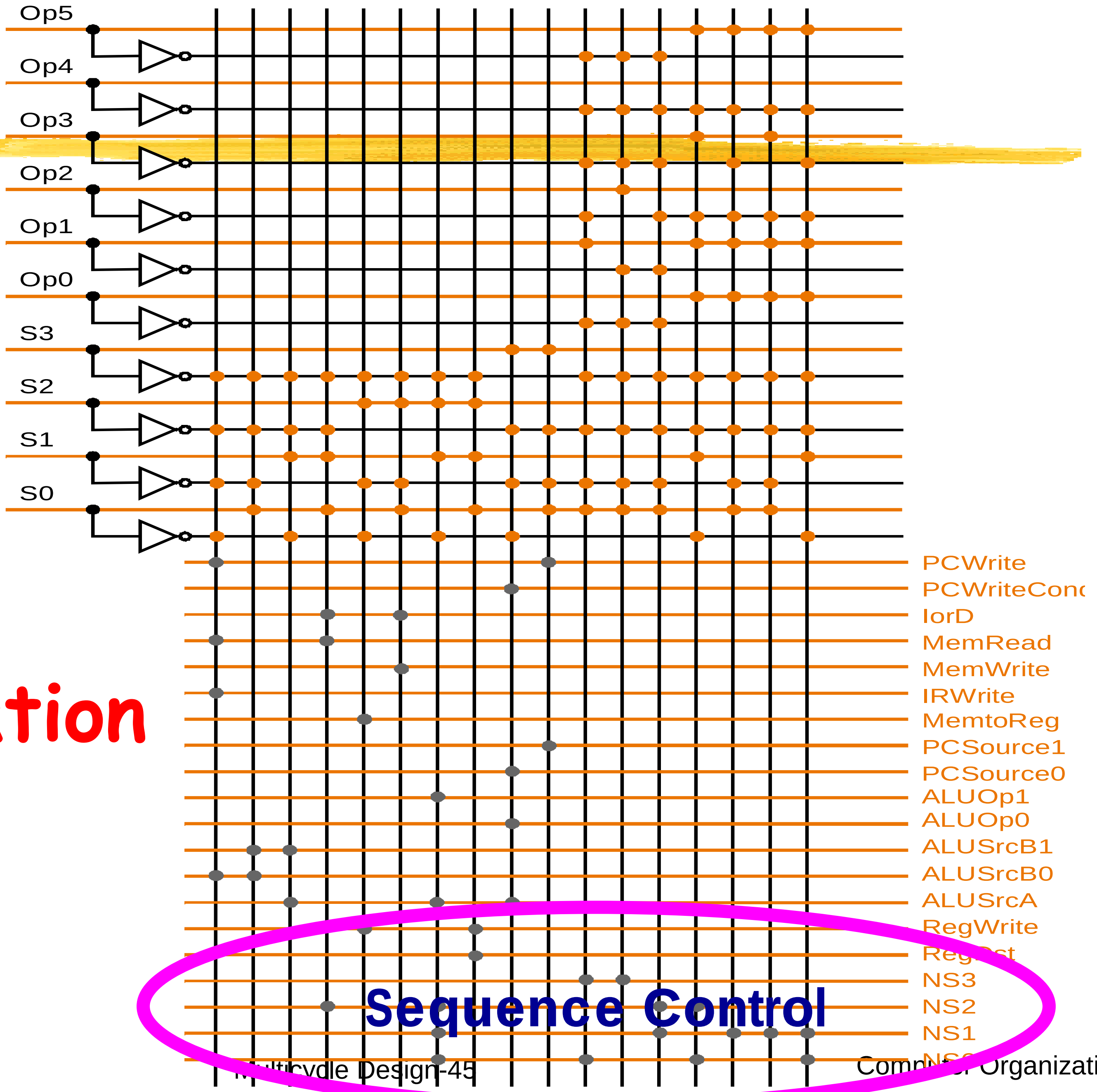


Fig. C.3.2



PLA Implementation

Fig. C.3.9

Sequence Control

ROM Implementation(Truth Table)

Address		ROM content	
op	S	Datapath control	NS
000000	0000	1001010000001000	0001
000000	0001	0000000000011000	0110
000000	0010	0000000000010100	xxxx
⋮			
000000	1010		
⋮			
000000	1111		
000001	0000		
⋮			
000010	0000	1001010000001000	0001
000010	0001	0000000000011000	1001
000000	0010	0000000000010100	xxxx
⋮			

- Rather wasteful, since for lots of entries, outputs are same or are don't-care
- Could break up into two smaller ROMs (Fig. C.3.7, C.3.8)

ROM vs. PLA

- ROM: use two smaller ROMs (Fig. C.3.7, C.3.8)
  - 4 state bits give the 16 outputs, 2^4x16 bits of ROM
  - 10 bits (op + state) give 4 next state bits, 2^{10}x 4 bits of ROM
  - Total = 4.3K bits of ROM (compared to 2^{10}x 20 bits of single ROM implementation)
- 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 × 2-input-phase × #product-terms) + (#outputs × #product-terms)

    For this example = (10 × 2 × 17) + (20 × 17) = 680 PLA cells
- PLA cell usually about the size of a ROM cell (slightly bigger)

Complete FSM

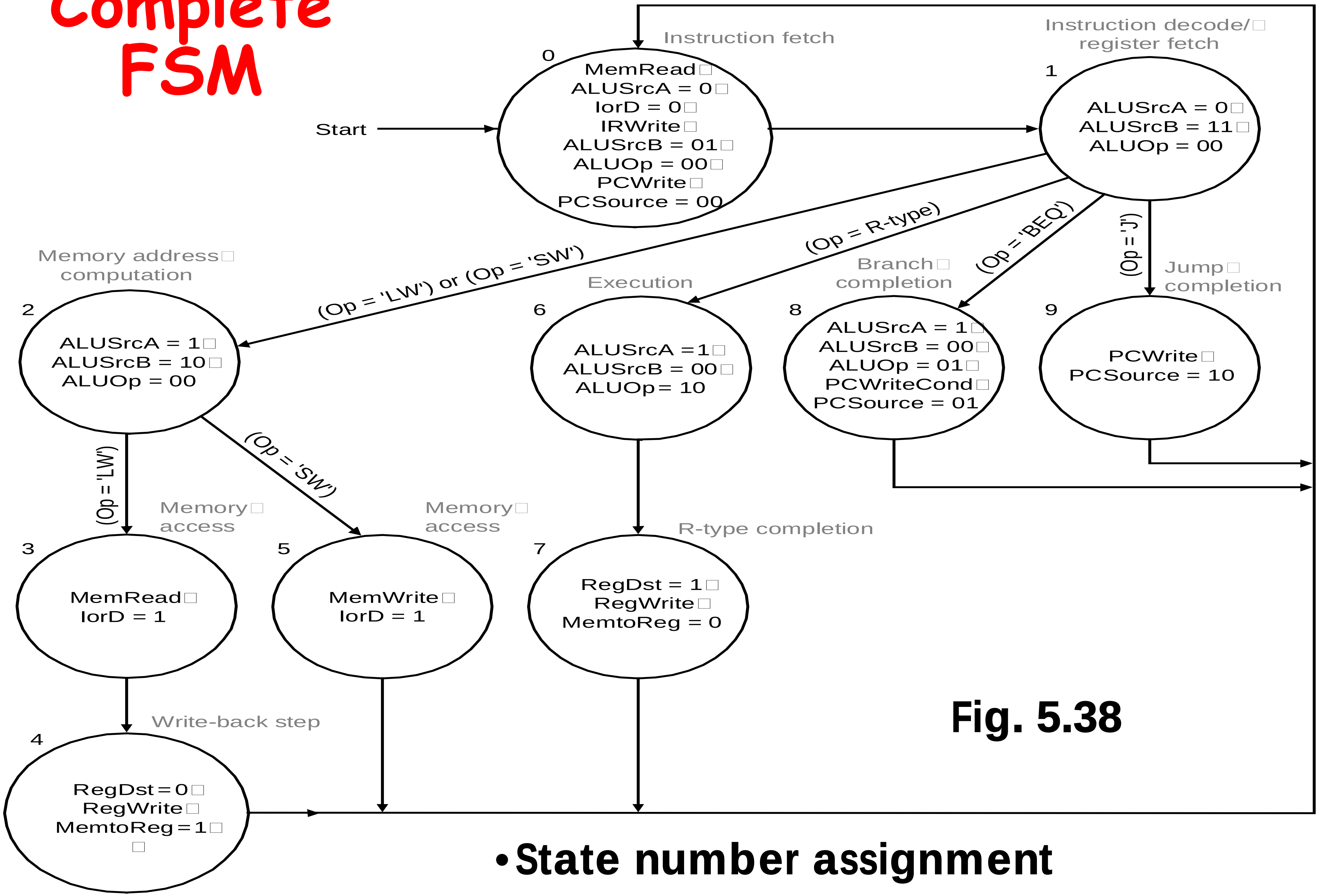
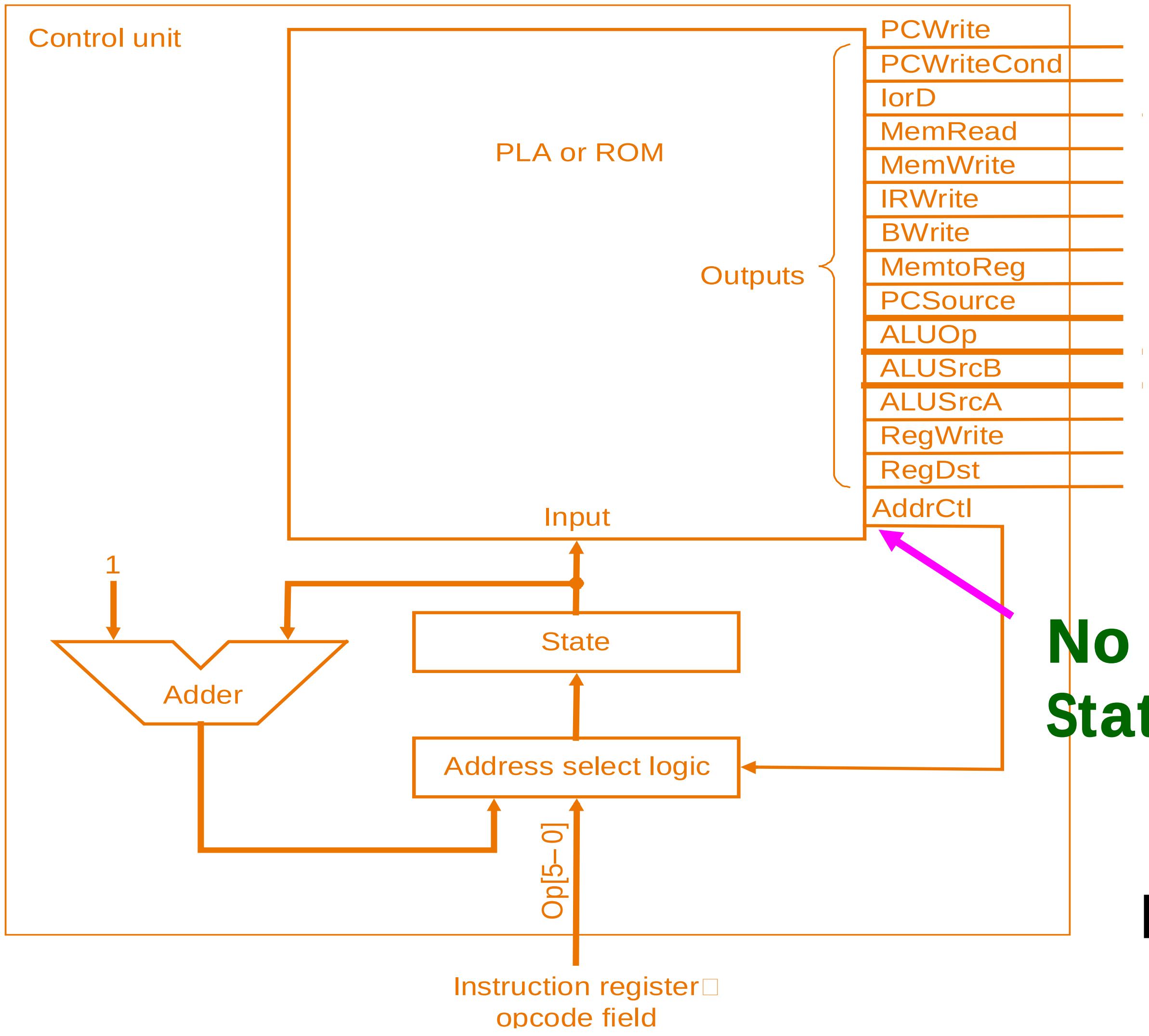


Fig. 5.38

- State number assignment

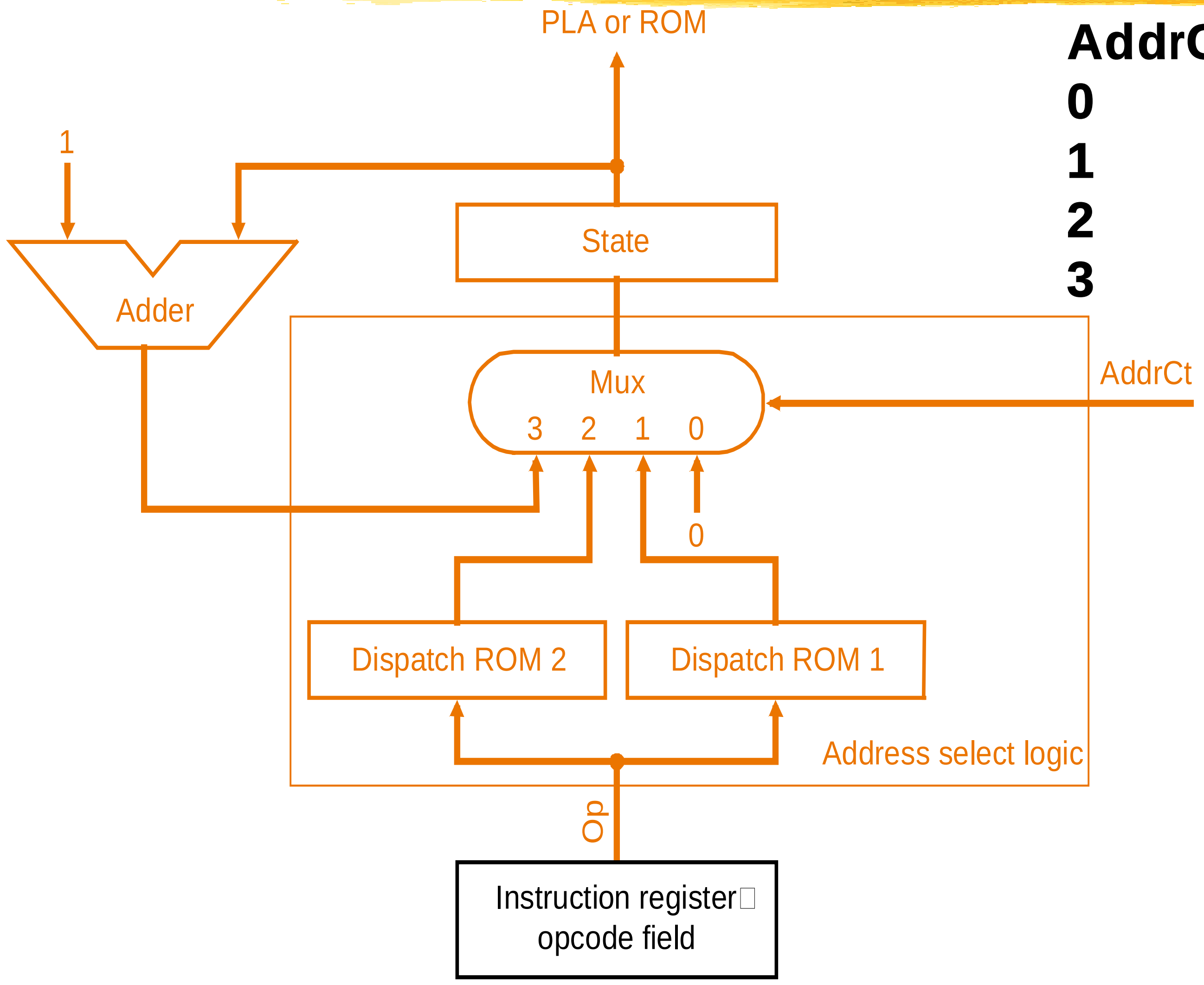
Use Counter for Sequence Control



No Next-State Lines

Fig. C.4.1

Address Select Unit



AddrCtl Value:

0	**Set state to 0**
1	**Dispatch ROM 1**
2	**Dispatch ROM 2**
3	**Use incremented state**

Fig. C.4.2

Control Contents

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

Fig. C.4.3, C.4.4

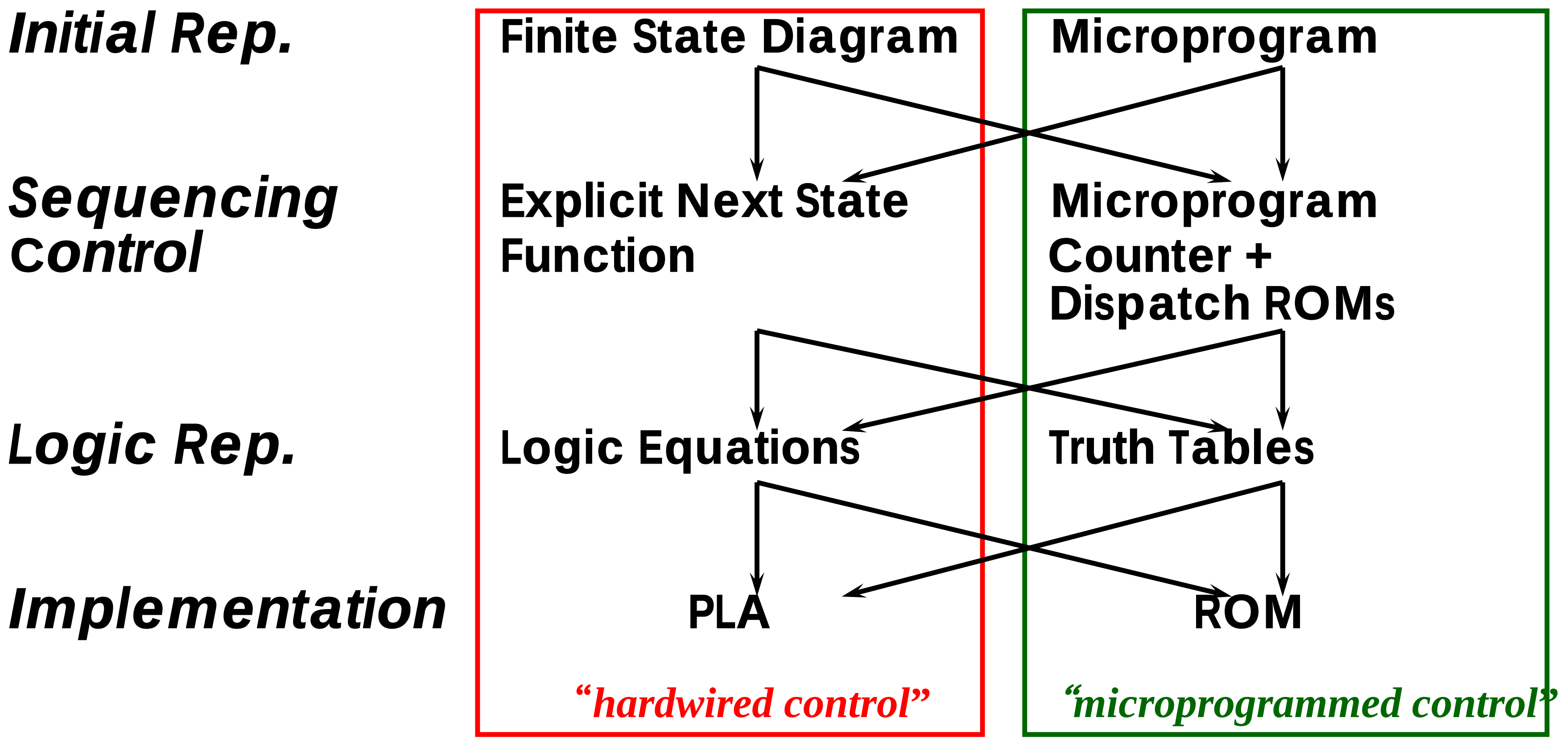
State number	Address-control action	Value of AddrCtl
0 (0000)	Use incremented state	3
1 (0001)	Use dispatch ROM 1	1
2 (0010)	Use dispatch ROM 2	2
3 (0011)	Use incremented state	3
4 (0100)	Replace state number by 0	0
5 (0101)	Replace state number by 0	0
6 (0110)	Use incremented state	3
7 (0111)	Replace state number by 0	0
8 (1000)	Replace state number by 0	0
9 (1001)	Replace state number by 0	0

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control
- Microprogramming: simplifying control (Appendix C.4)
- Exceptions

Controller Design: An Overview

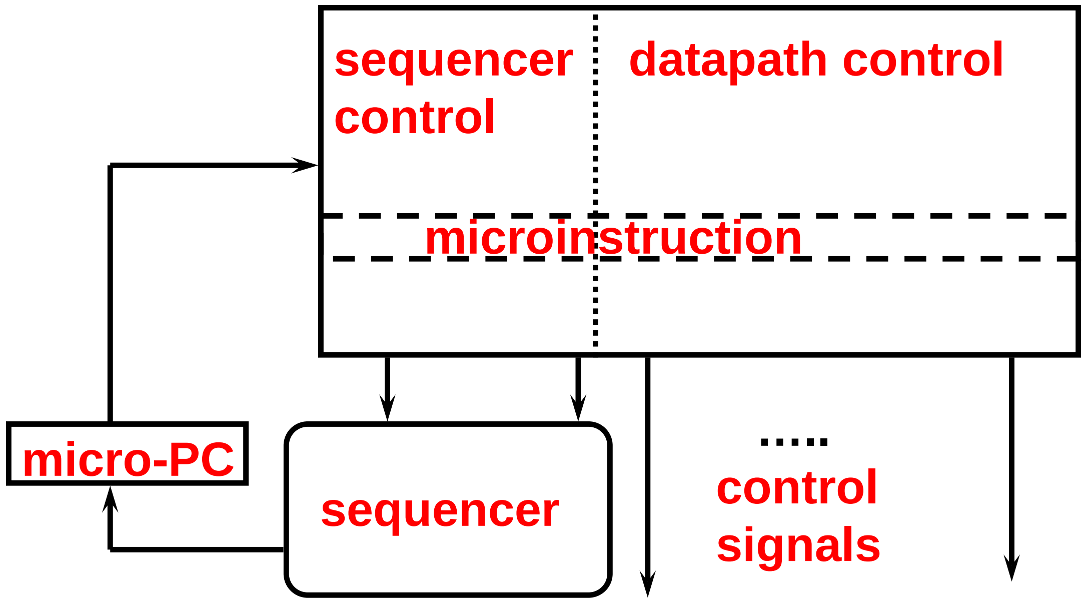
- Several possible initial representations, sequence control and logic representation, and control implementation => all may be determined indep.



Microprogram

- Control is the hard part of processor design
  - Datapath is fairly regular and well-organized
  - Memory is highly regular
  - Control is irregular and global
- But, the state diagrams that define the controller for an instruction set processor are highly structured
  - Use this structure to construct a simple "microsequencer"
  - Control reduces to programming this simple device

  => *microprogramming*



Microinstruction

- Control signals :
  - Think of the set of control signals that must be asserted in a state as an instruction
  - Executing a microinstruction has the effect of asserting the control signal specified by the microinstruction
- Sequencing
  - What microinstruction should be executed next ?
    - Execute sequentially (next state unconditionally)
    - Branch (next state also depends on inputs)

- A microprogram is a sequence of microinstructions executing a program flow chart (finite state machine)

Designing a Microinstruction Set

1) Start with a list of control signals
2) Group signals together that make sense (vs. random): called *fields*
3) Places fields in some logical order (ex: ALU operation & ALU operands first and microinstruction sequencing last)
4) Create a symbolic legend for the microinstruction format, showing name of field values and how they set control signals
   - Use computers to design computers
5) To minimize the width, encode operations that will never be used at the same time

1-3) Control Signals and Fields

Single Bit Control

Signal name	*Effect when deasserted*	*Effect when asserted*
ALUSrcA	1st ALU operand = PC	1st ALU operand = Reg[rs]
RegWrite	None	Reg file is written
MemtoReg	Reg. write data input = ALU	Reg. write data input = MDR RegDst
	Reg. write dest. no. = rt	Reg. write dest. no. = rd
MemRead	None	Memory at address is read
MemWrite	None	Memory at address is written
IorD	Memory address = PC	Memory address = ALUout
IRWrite	None	IR = Memory
PCWrite	None	PC = PCSource
PCWriteCond	None	If zero then PC = PCSource

Multiple Bit Control

Signal name	*Value*	*Effect*
ALUOp	00	ALU adds
	01	ALU subtracts
	10	ALU operates according to func code
ALUSrcB	00	2nd ALU input = B
	01	2nd ALU input = 4
	10	2nd ALU input = sign extended IR[15-0]
	11	2nd ALU input = sign extended, shift left 2 IR[15-0]
PCSource	00	PC = ALU (PC + 4)
	01	PC = ALUout (branch target address)
	10	PC = PC+4[31-28] : IR[25-0] << 2

4) Fields and Legend

Field Name	*Values for Field*	*Function of Field with Specific Value*
ALU control	Add	ALU adds
	Subt.	ALU subtracts
	Func code	ALU does function code
SRC1	PC	1st ALU input = PC
	A	1st ALU input = A (Reg[rs])
SRC2	B	2nd ALU input = B (Reg[rt])
	4	2nd ALU input = 4
	Extend	2nd ALU input = sign ext. IR[15-0]
	Extshft	2nd ALU input = sign ext., sl-2 IR[15-0]
Register control	Read	A = Reg[rs]; B = Reg[rt];
	Write ALU	Reg[rd] = ALUout
	Write MDR	Reg[rt] = MDR
Memory	Read PC	IR (MDR) = mem[PC]
	Read ALU	MDR = mem[ALUout]
	Write ALU	mem[ALUout] = B
PC write	ALU	PC = ALU output
	ALUout-cond.	IF ALU zero then PC = ALUout
	jump addr.	PC = PCSource
Sequencing	Seq	Go to sequential microinstruction
	Fetch	Go to the first microinstruction
	Dispatch 1	Dispatch using ROM1
	Dispatch 2	Dispatch using ROM2

Control Signals

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

The Microprogram

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Fig. 5.7.3

The Controller

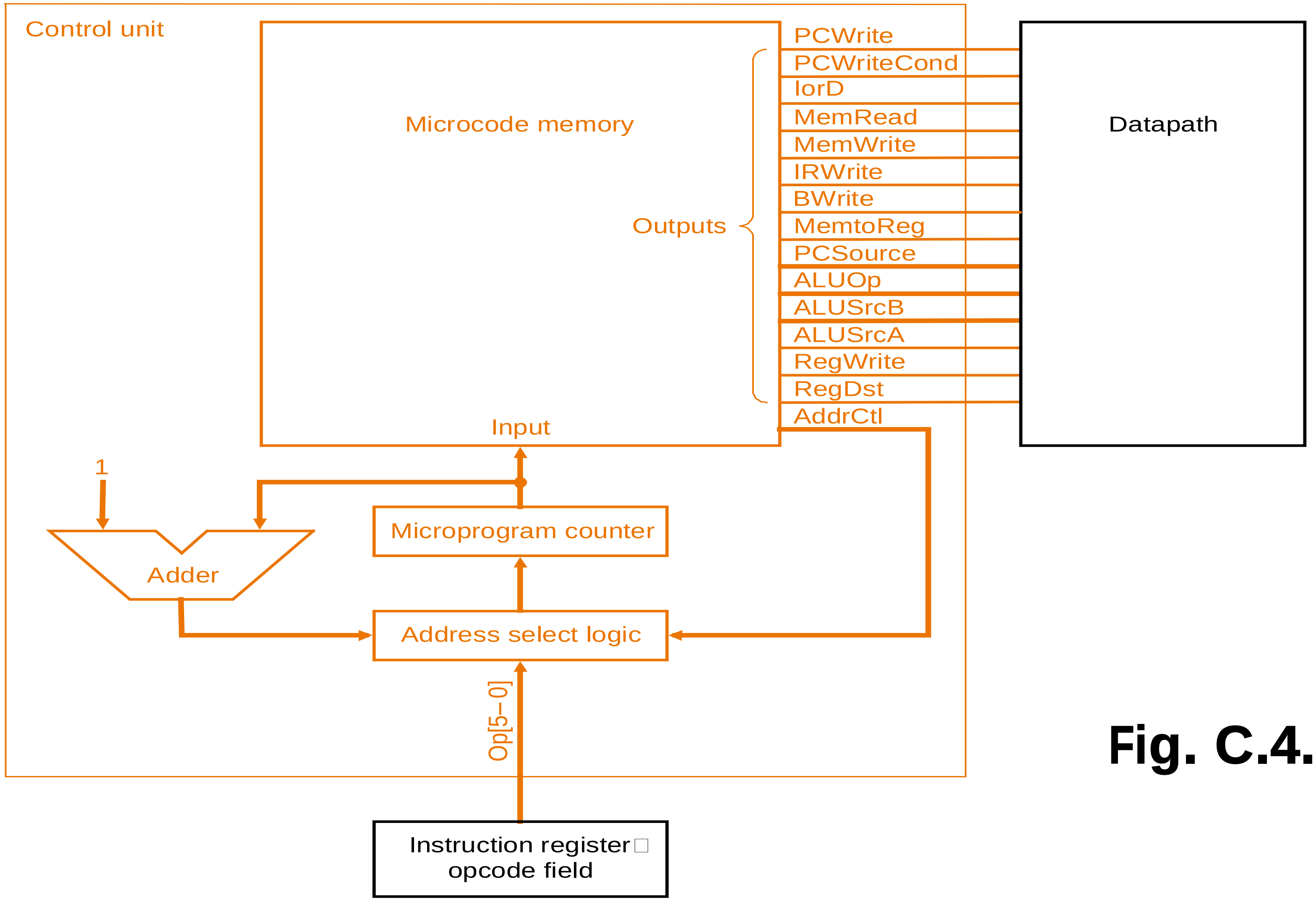


Fig. C.4.6

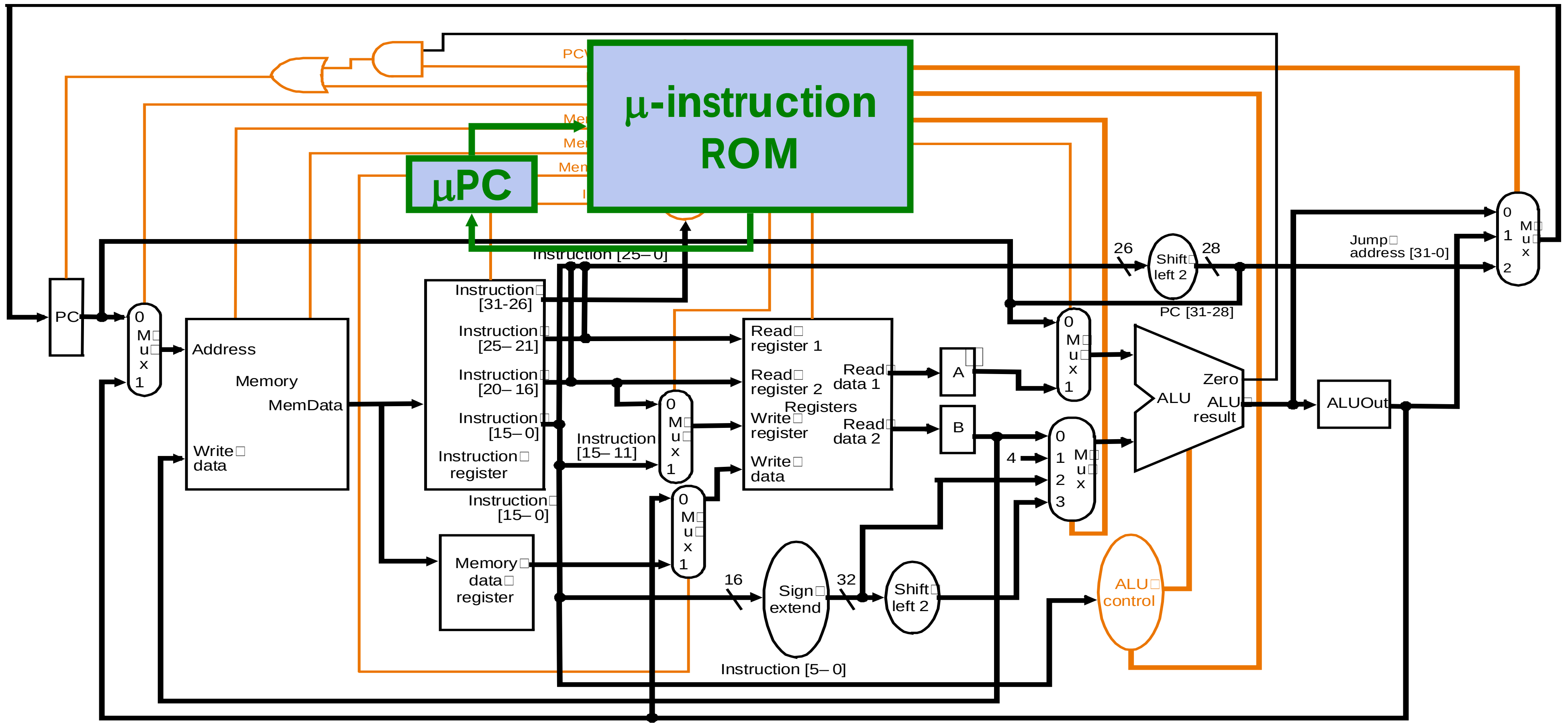
The Dispatch ROMs

Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

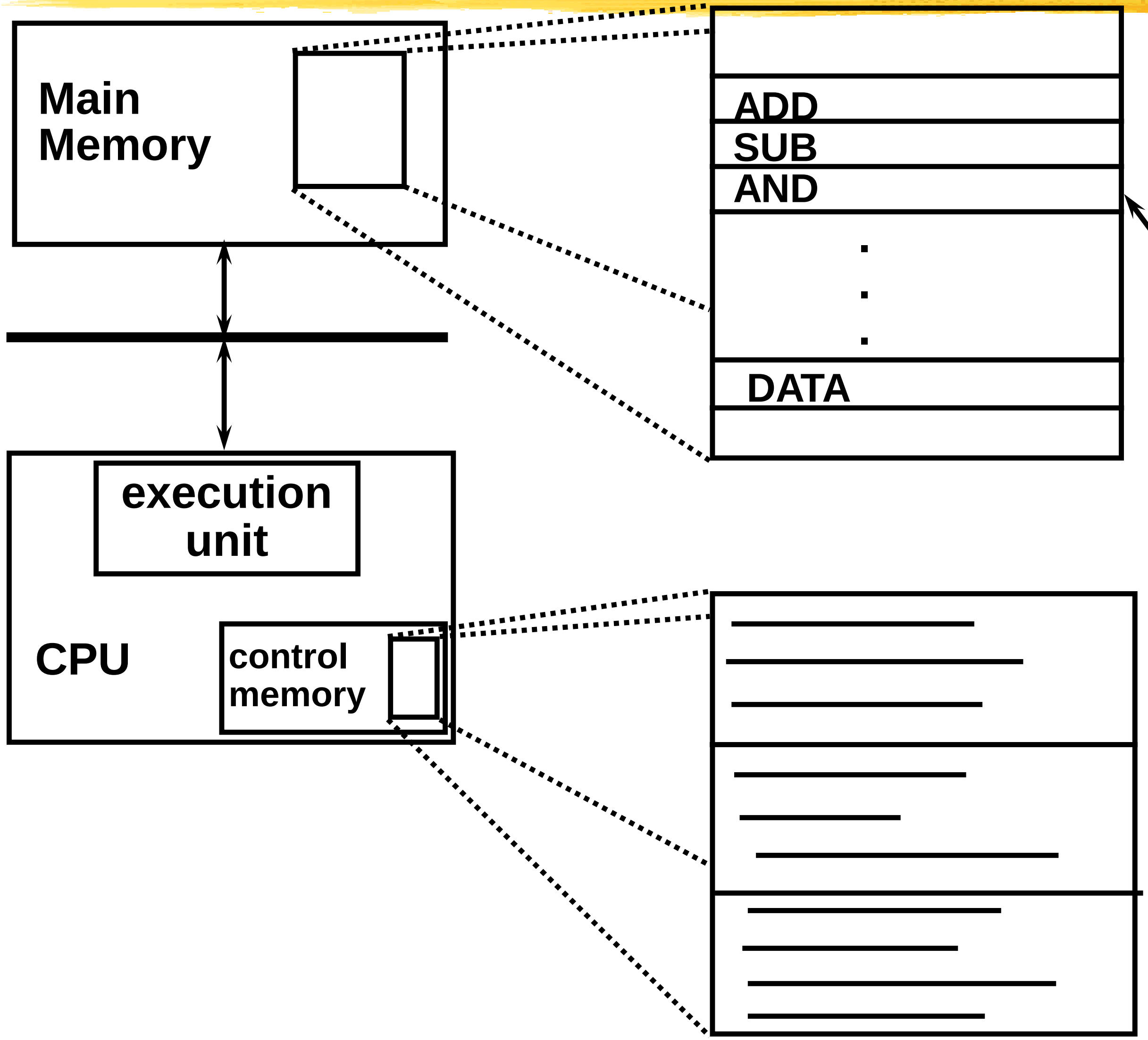
Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	LW2
101011	sw	SW2

Fig. C.5.2

Our Plan: Using ROM



Microinstruction Interpretation



User program
plus Data

this can be changed!

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 Using ROM : 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
- Generality
  - Implement multiple inst. sets on same machine
  - Can tailor instruction set to application
  - Can implement very powerful instruction sets (just more control memory)
- Compatibility
  - Many organizations, same instruction set
- Costly to implement and slow

5) Microinstruction Encoding

State number	Control bits 17- 2	Control bits 1- 0
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

Fig. C.4.5

- **Bits 7-13 can be encoded to 3 bits because there are only 7 patterns of the control word**

Minimal vs. Maximal Encoding

- ***Minimal (Horizontal):***

+ more control over the potential parallelism of operations in the datapath

- uses up lots of control store

- ***Maximal (Vertical):***

+ uses less number of control store

- extra level of decoding may slow the machine down

Recap: Designing a Microinstruction Set

1) Start with a list of control signals
2) Group signals together that make sense (vs. random): called *fields*
3) Places fields in some logical order (ex: ALU operation & ALU operands first and microinstruction sequencing last)
4) Create a symbolic legend for the microinstruction format, showing name of field values and how they set control signals
   - Use computers to design computers
5) To minimize the width, encode operations that will never be used at the same time

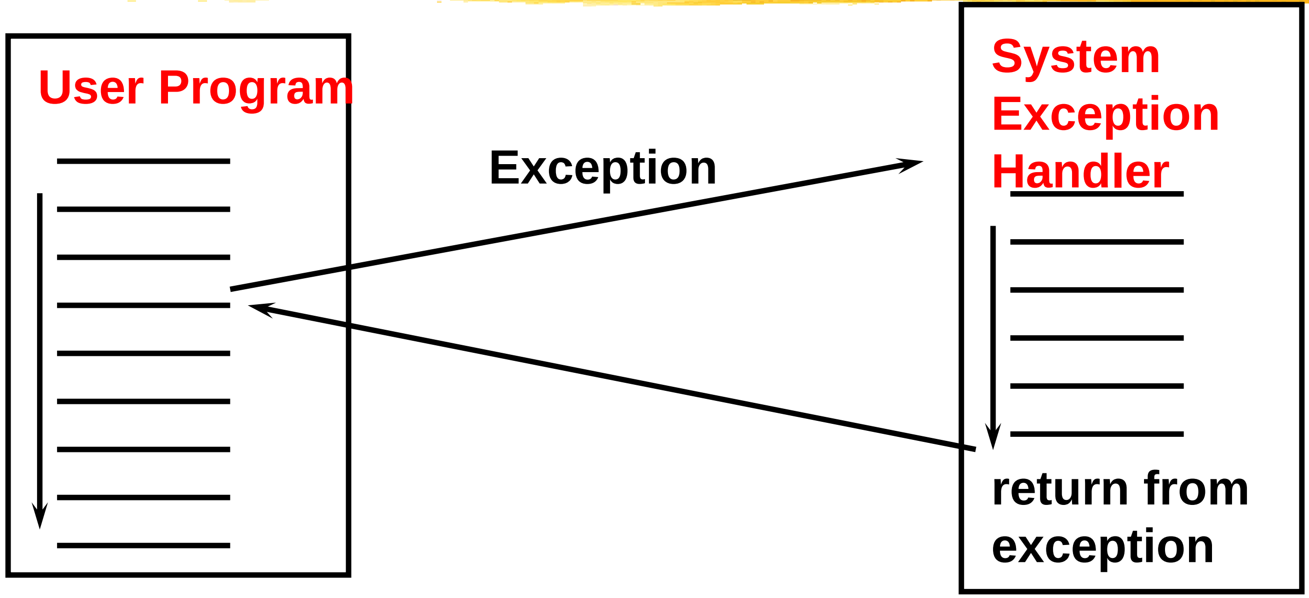
Summary of Control

- Control is specified by a finite state diagram
- Specialized state-diagrams easily captured by microsequencer
  - simple increment and “branch” fields
  - datapath control fields
- Control can also be specified by microprogramming
- Control is more complicated with:
  - complex instruction sets
  - restricted datapaths
- Simple instruction set and powerful datapath => simple control
  - could reduce hardware
  - Or go for speed => many instructions at once!

Outline

- Designing a processor
- Building the datapath
- A single-cycle implementation
- A multicycle implementation
  - Multicycle datapath
  - Multicycle execution steps
  - Multicycle control
- Microprogramming: simplifying control
- Exceptions

Exceptions



- **Normal control flow:** sequential, jumps, branches, calls, returns
- **Exception = unprogrammed control transfer**
  - system takes action to handle the exception
    - must record address of the offending instruction
    - should know cause and transfer to proper handler
    - if returns to user, must save & restore user state

User/System Modes

- By providing two modes of execution (user/system), computer may manage itself
  - OS is a special program that runs in the privileged system mode and has access to all of the resources of the computer
  - Presents "virtual resources" to each user that are more convenient than the physical resources
    - files vs. disk sectors
    - virtual memory vs. physical memory
  - protects each user program from others
- Exceptions allow the system to take action in response to events that occur while user program is executing
  - OS begins at the handler

Two Types of Exceptions

- **Interrupts:**
  - **caused by external events and asynchronous to execution**
    → may be handled between instructions
  - **simply suspend and resume user program**
- **Exceptions:**
  - **caused by internal events and synchronous to execution, ex: exceptional conditions (overflow), errors (parity), faults**
  - **instruction may be retried or simulated and program continued or program may be aborted**

MIPS Convention of Exceptions

- **MIPS convention:**
  - ***exception* means any unexpected change in control flow, without distinguishing internal or external**
  - **use *interrupt* only when the event is externally caused**

Type of event	***From where?***	***MIPS terminology***
I/O device request	**External**	**Interrupt**
Invoke OS from user program	**Internal**	**Exception**
Hardware malfunctions	**Either**	**Exception or Interrupt**
Arithmetic overflow	**Internal**	**Exception**
Using an undefined inst.	**Internal**	**Exception**

Precise Interrupts

- *Precise*: machine state is preserved as if program executed upto the offending inst.
  - Same system code will work on different implementations of the architecture
  - Position clearly established by IBM, and taken by MIPS
  - Difficult in the presence of pipelining, out-ot-order execution, ...
- *Imprecise:* system software has to figure out what is, where is, and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc., usually wish they had not done this

Handling Exceptions in Our Design

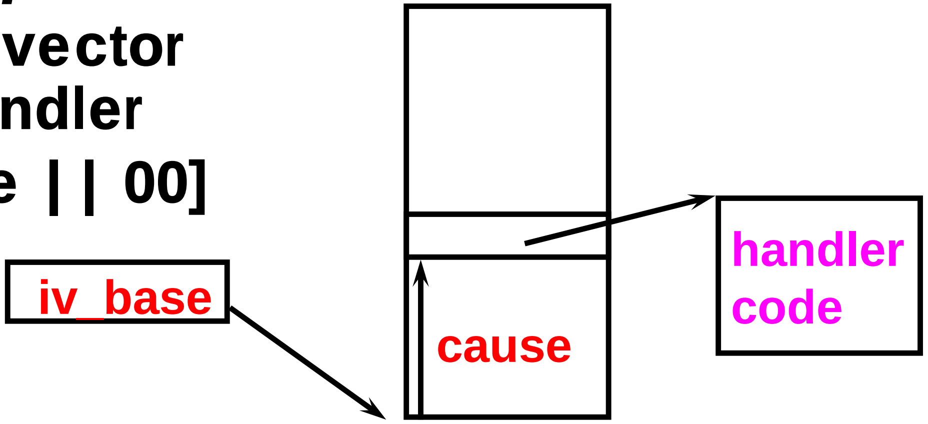
- **Consider two types of exceptions: undefined instruction & arithmetic overflow**
- **Basic actions on exception:**
  - **Save state: save the address of the offending instruction in the *exception program counter* (EPC)**
  - **Transfer control to OS at some specified address**
    → need to know the cause for the exception
    → then know the address of exception handler
  - **After service, OS can terminate the program or continue its execution, using EPC to return**

Saving State: General Approaches

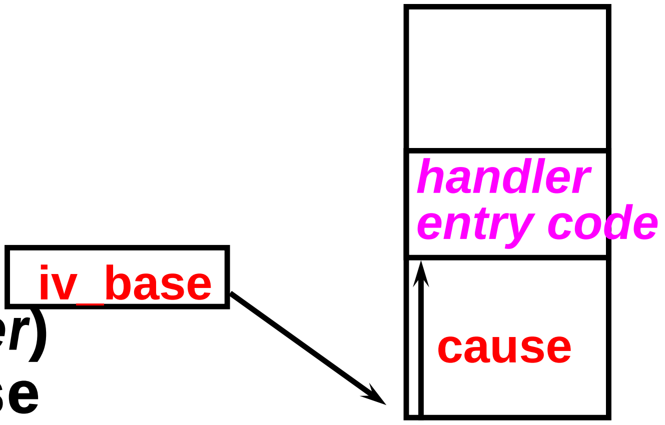
- Push it onto the stack
  - Vax, 68k, 80x86
- Save it in special registers
  - MIPS EPC, BadVaddr, Status, Cause
- Shadow Registers
  - M88k
  - Save state in a shadow of the internal pipeline registers

Addressing the Exception Handler

- Traditional approach: *interrupt vector*
  - The cause of exception is a vector giving the address of the handler
  - PC ← MEM[IV_base + cause || 00]
  - 68000, Vax, 80x86, . . .



- RISC Handler Table
  - PC ← IV_base + cause || 0000
  - Saves state and jumps
  - Sparc, PA, M88K, . . .
- MIPS approach: fixed entry
  - use a status register (*cause register*) to hold a field to indicate the cause
  - PC ← EXC_addr



Datapath with Exception Handling

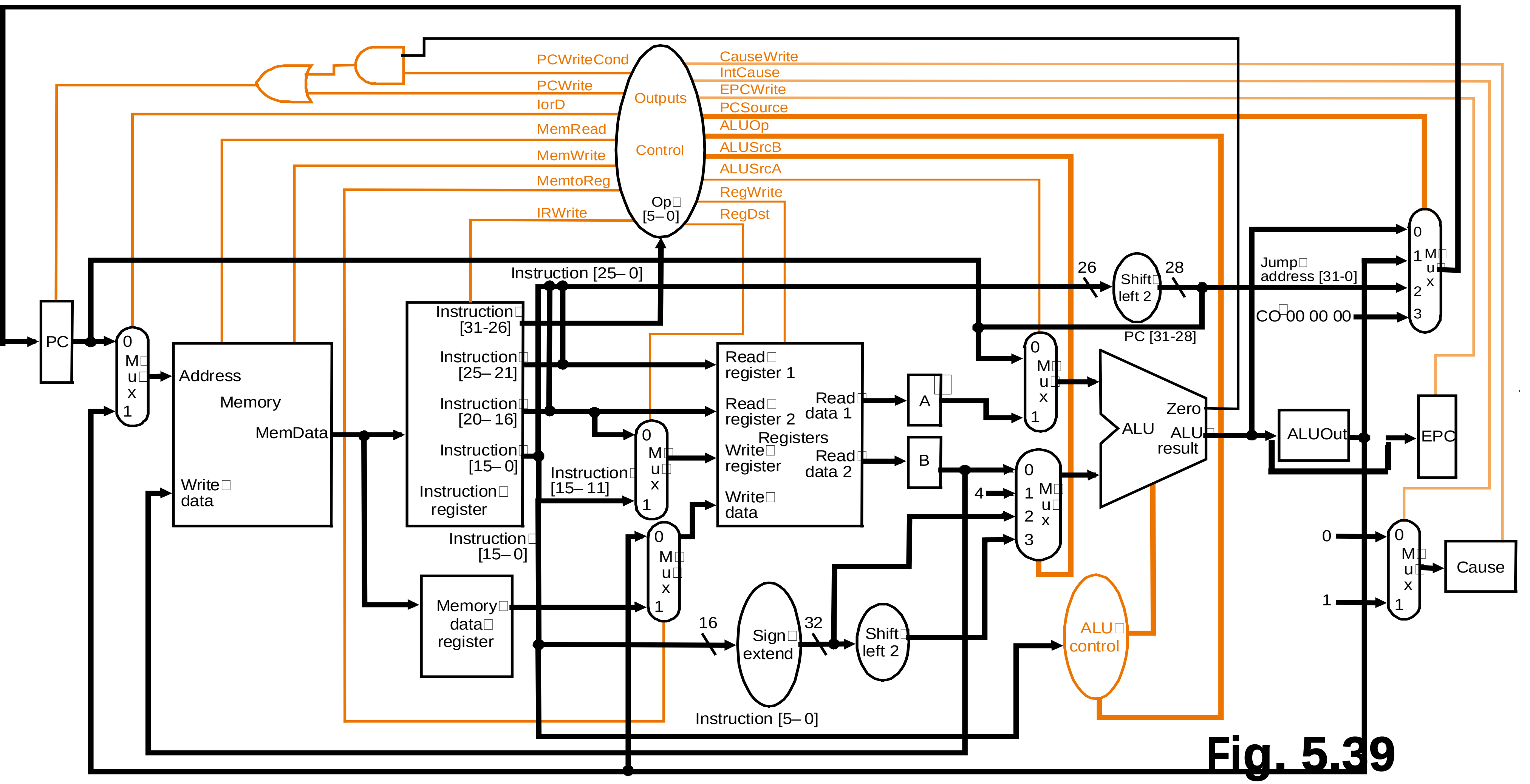


Fig. 5.39

Additions for Our Design

- EPC: reg. to hold address of affected inst.
- Cause: reg. to record cause of exception
  - Assume LSB encodes the two possible exception sources: undefined instruction=0 and arithmetic overflow=1
- Two control signals to write EPC (EPCWrite) and Cause (CauseWrite), and one control signal (IntCause) to set LSB of Cause register
- Be able to write *exception address* into PC, assuming at C000 0000$_{hex}$
  => needs a 4-way MUX to PC
- May undo PC = PC + 4 (PC = PC - 4), since want EPC to point to offending inst. (not its successor)

Exception Detection

- ***Undefined instruction***: detected when no next state is defined from state 1 for the op value
  - Handle this by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, and beq as a new state, “other”
- ***Arithmetic overflow***: detected with the *Overflow* signal out of the ALU
  - This signal is used in the modified FSM to specify an additional possible next state

Note: challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast

- Complex interactions makes the control unit the most challenging aspect of hardware design

FSM with Exception Handling

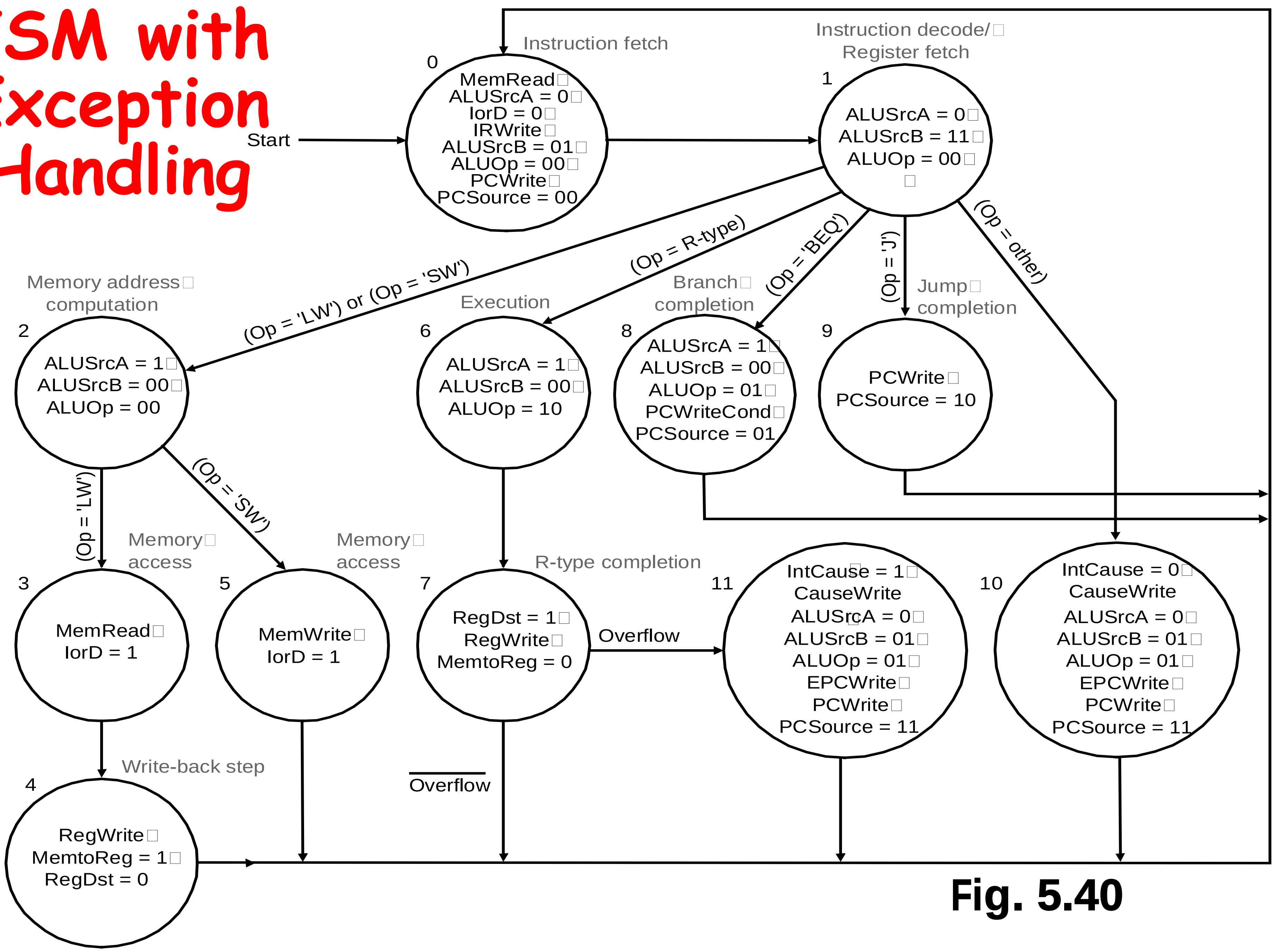


Fig. 5.40

Summary

- Specialize state diagrams easily captured by microsequencer
  - simple increment and branch fields
  - datapath control fields
- Control design reduces to microprogramming
- Exceptions are the hard part of control
  - Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes OS
  - Harder with pipelined CPUs that support page faults on memory accesses, i.e., the instruction cannot complete AND you must restart program at exactly the instruction with the exception

Datapath with Exception Handling

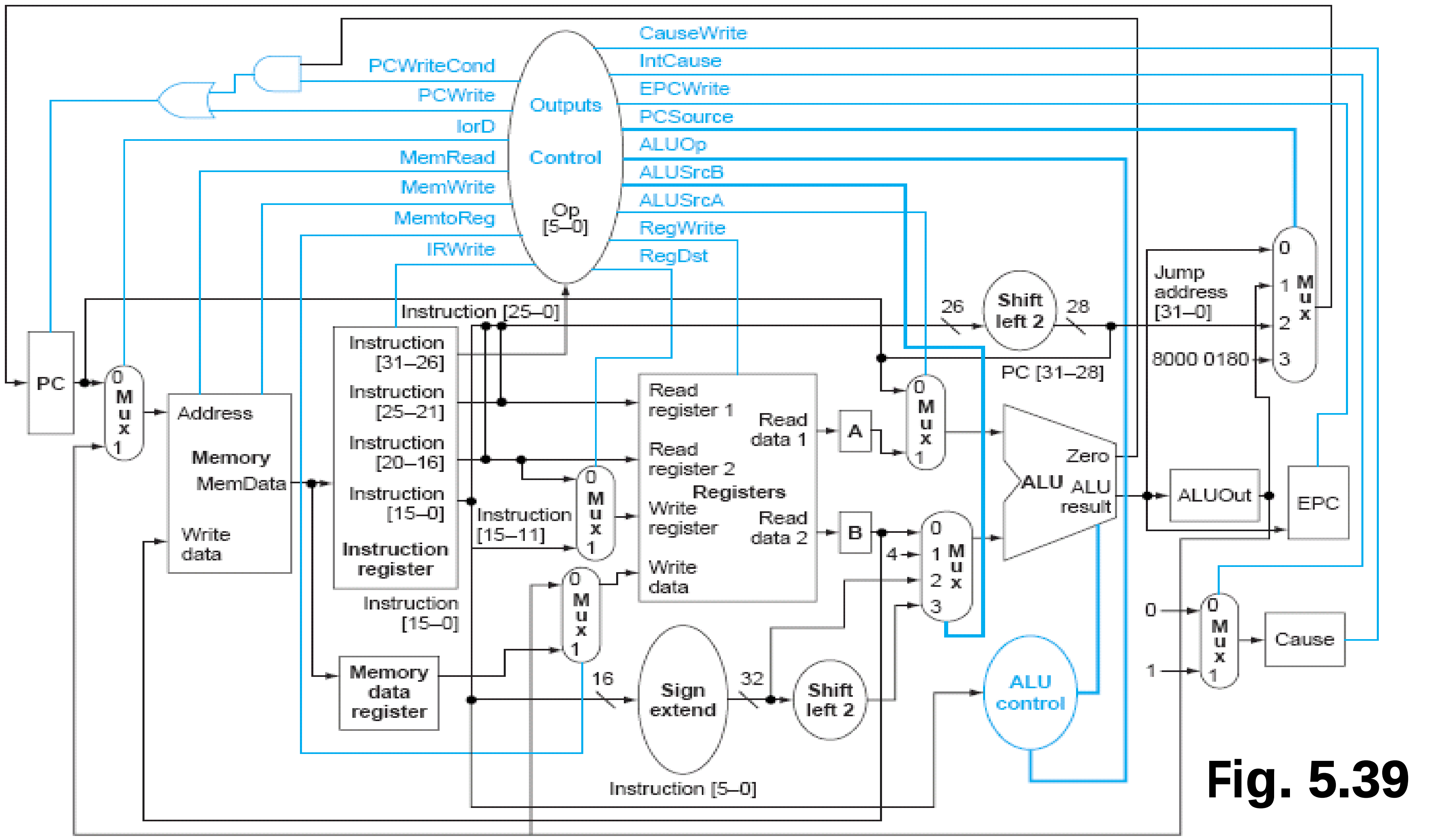


Fig. 5.39