Lecture 5: The Processor

CSCE 2610 Computer Organization

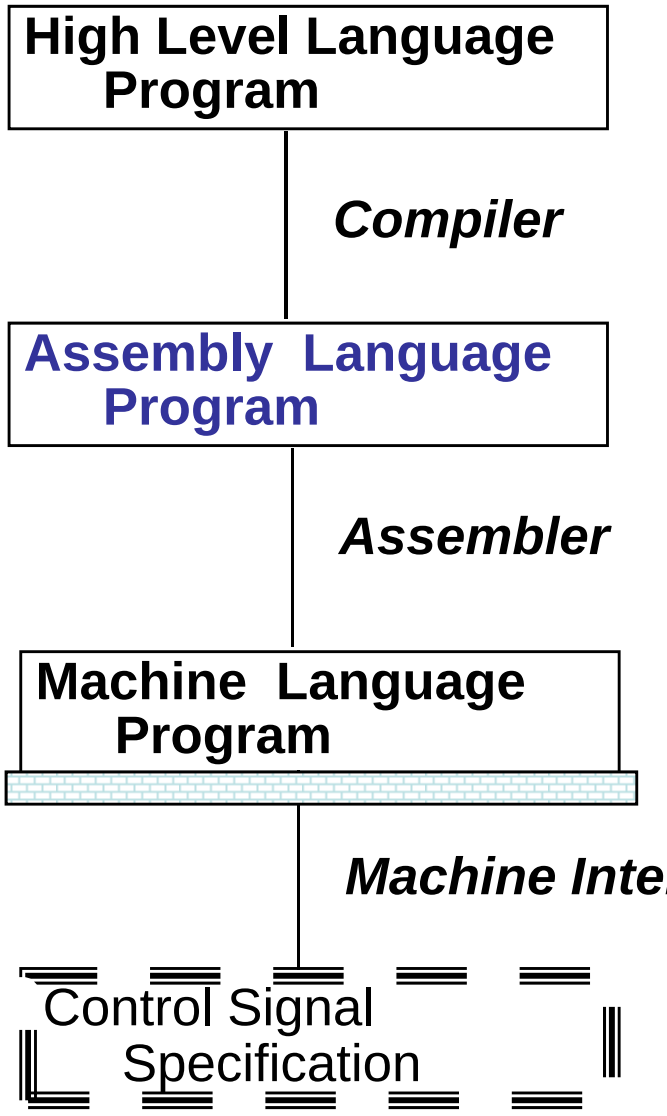
Instructor: Saraju P. Mohanty, Ph. D.

NOTE: The figures, text etc included in slides are borrowed from various books, websites, authors pages, and other sources for academic purpose only. The instructor does not claim any originality.

Lecture Outline

- Construction of a Simple MIPS Processor
- Single Cycle Processor
- Multicycle Processor

Levels of Representation

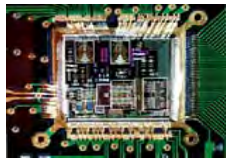



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

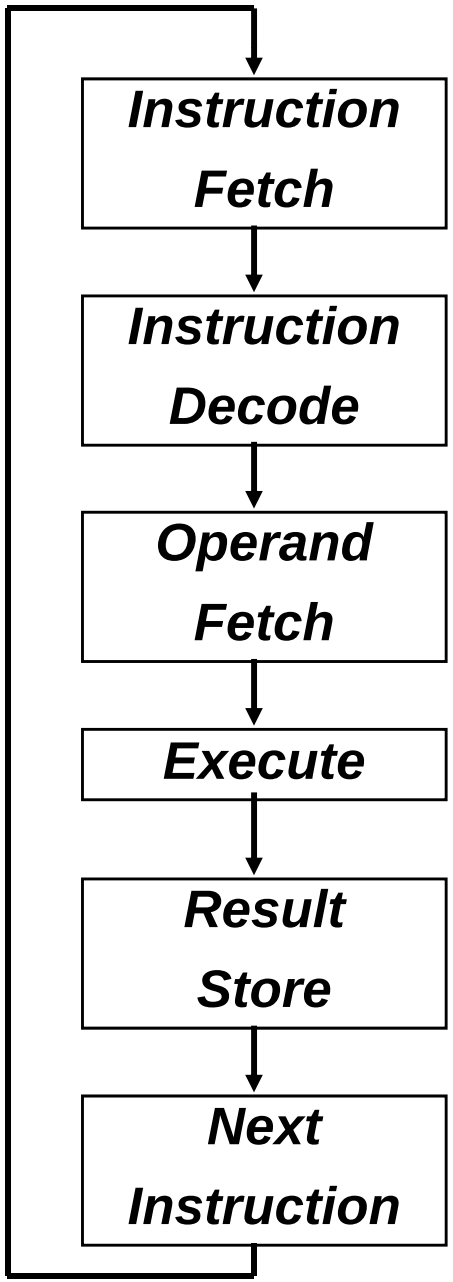
```
lw$15, 0($2)
lw$16, 4($2)
sw     $16, 0($2)
sw     $15, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

ALUOP[0:3] <= InstReg[9:11] & MASK



Execution Cycle



Obtain instruction from program storage

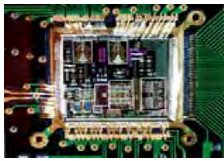
Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

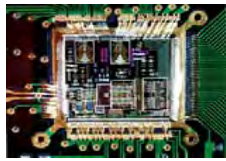
Deposit results in storage for later use

Determine successor instruction



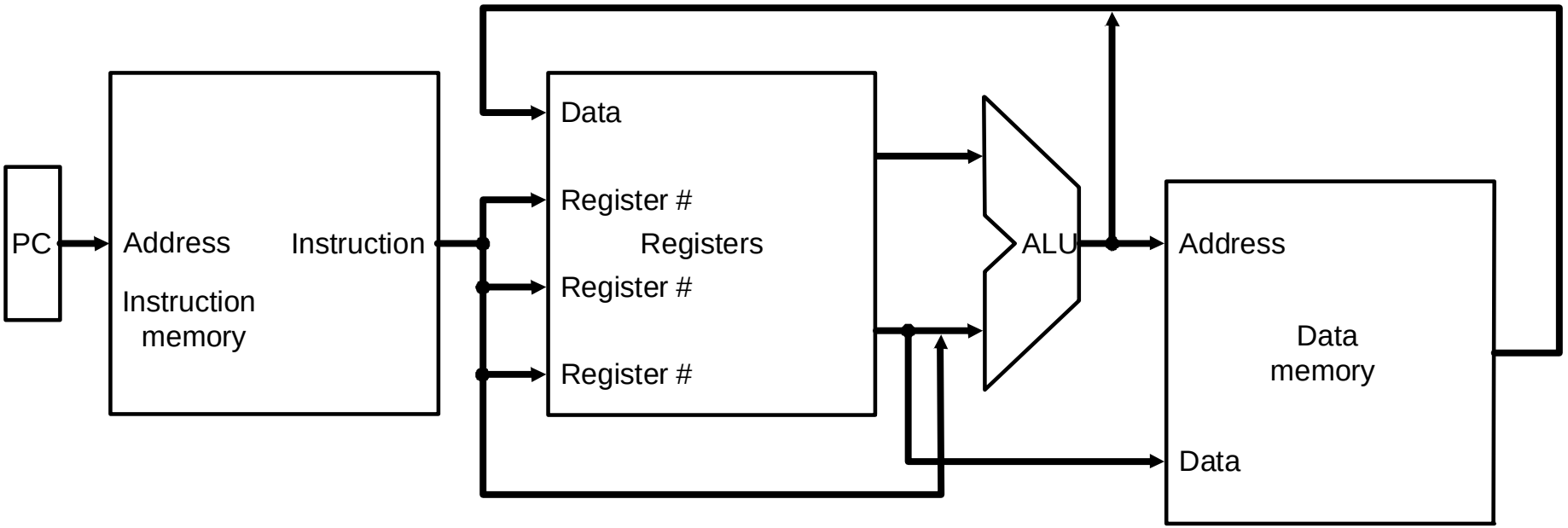
The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions:`add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do



More Implementation Details

- Abstract / Simplified View
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

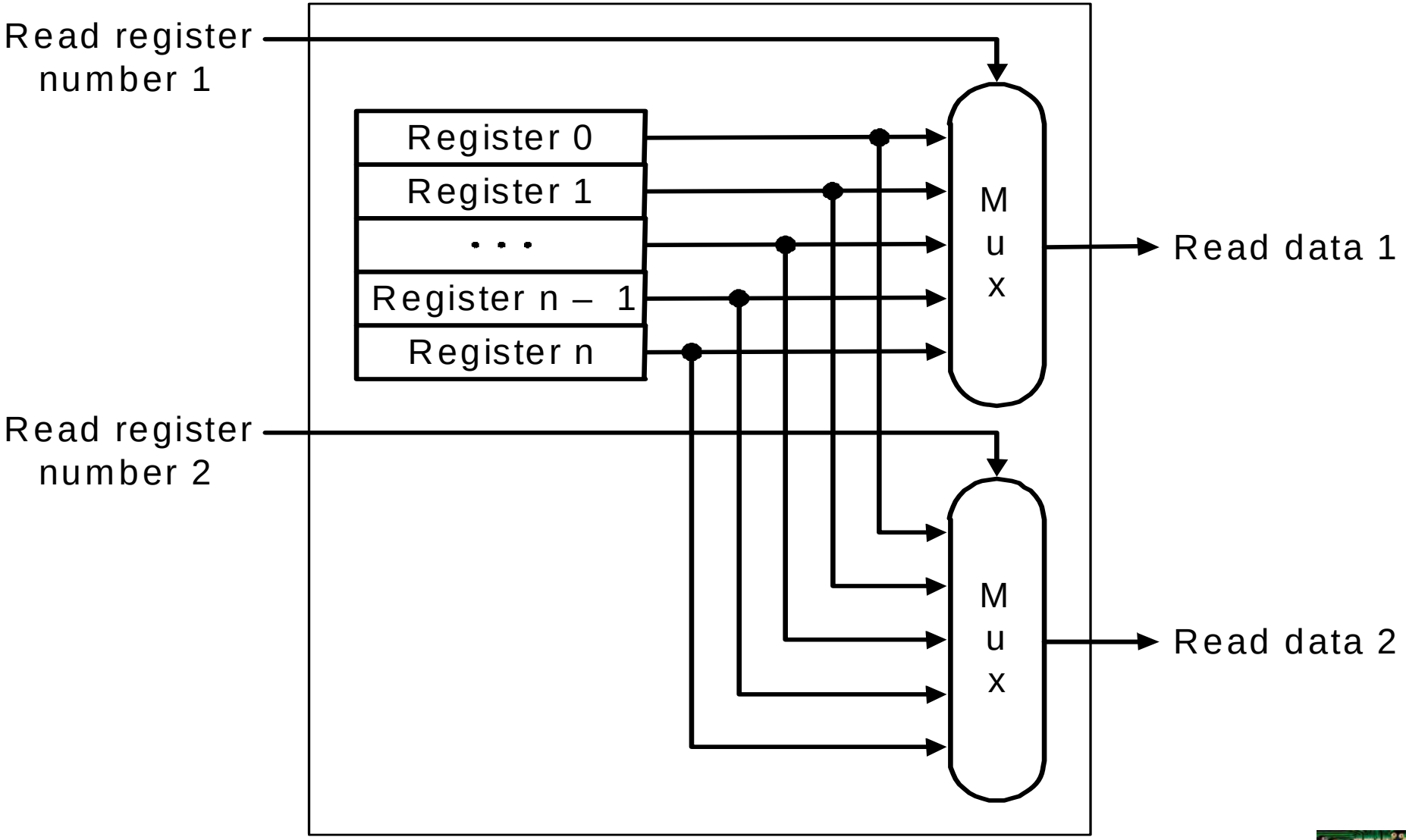


Basic Building Blocks

- Latches
- Flip-flops
- Combinational Elements
- Sequential Elements
- Clocking strategies

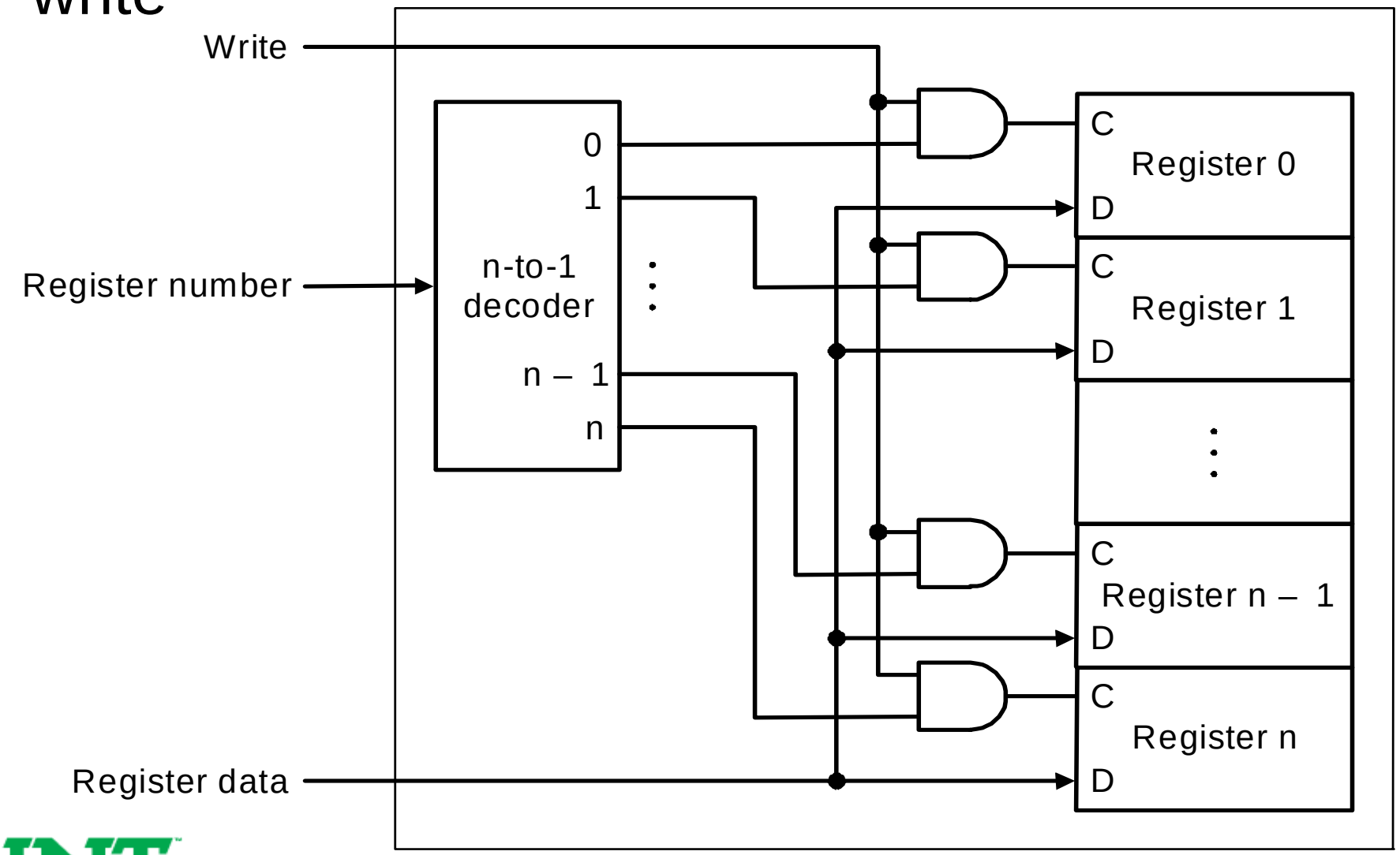
Register File: Read Operation

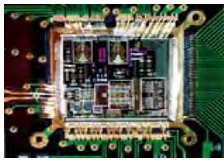
- Built using D flip-flops (Combinational in nature)



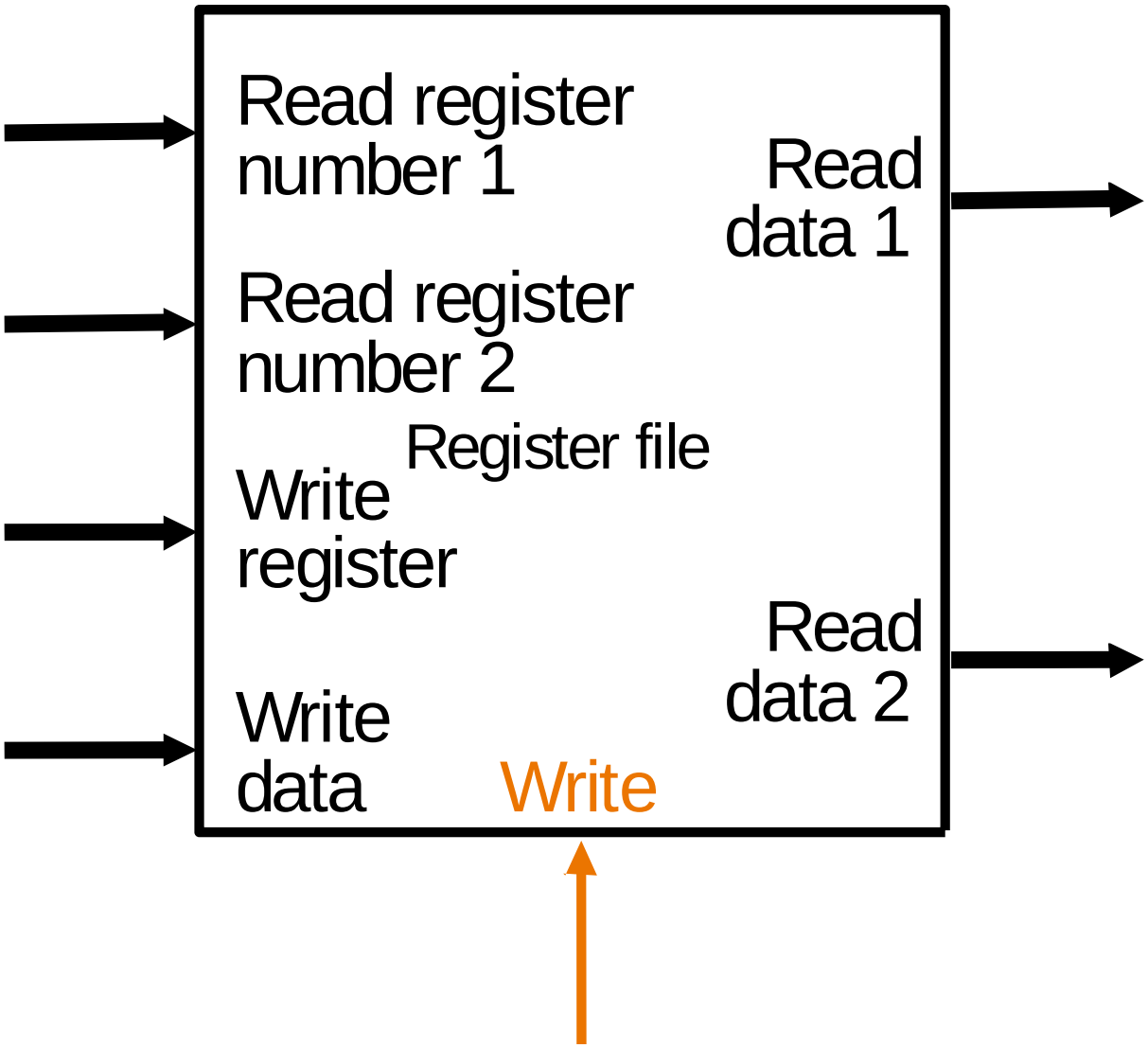
Register File: Write Operation

- We still use the real clock to determine when to write





Register File: Block Diagram



- Three Address ports
- One Data Input Port
- Two Data Output Ports
- One Write Control Signal

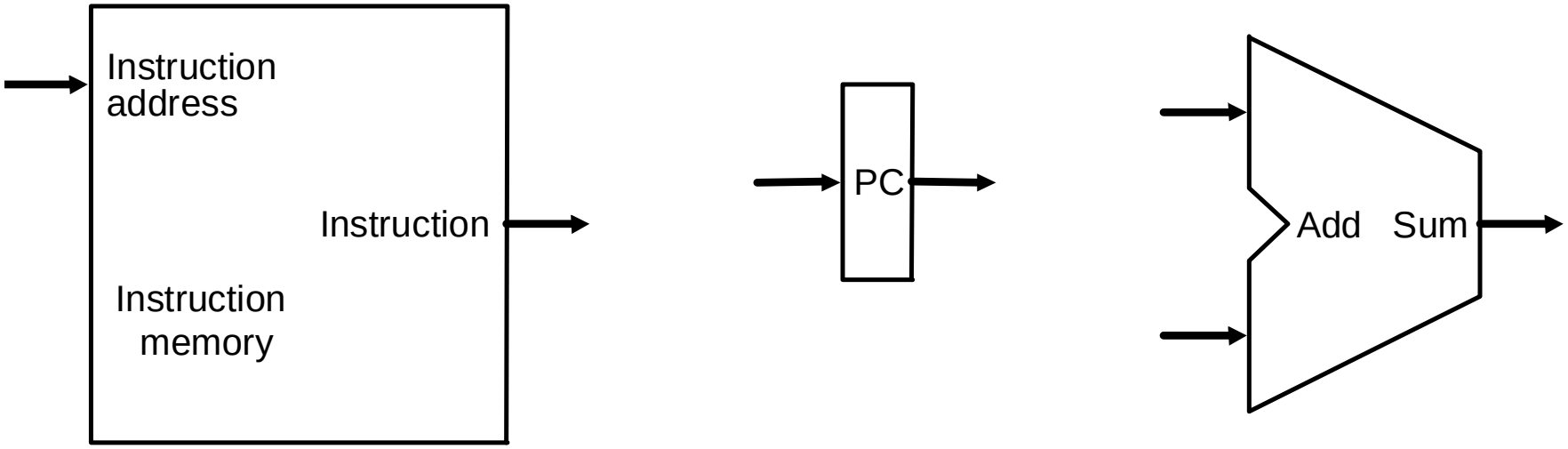
Functional Units - I

a: *Instruction Memory*

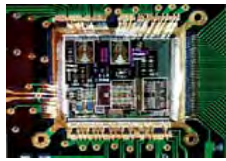
After an instruction address is put, the instruction residing at the address appears at the output port

b: *Program Counter* -- A simple up counter

c: *Adder* -- A 2's complement adder



a. Instruction memory b. Program counter c. Adder



Functional Units - II

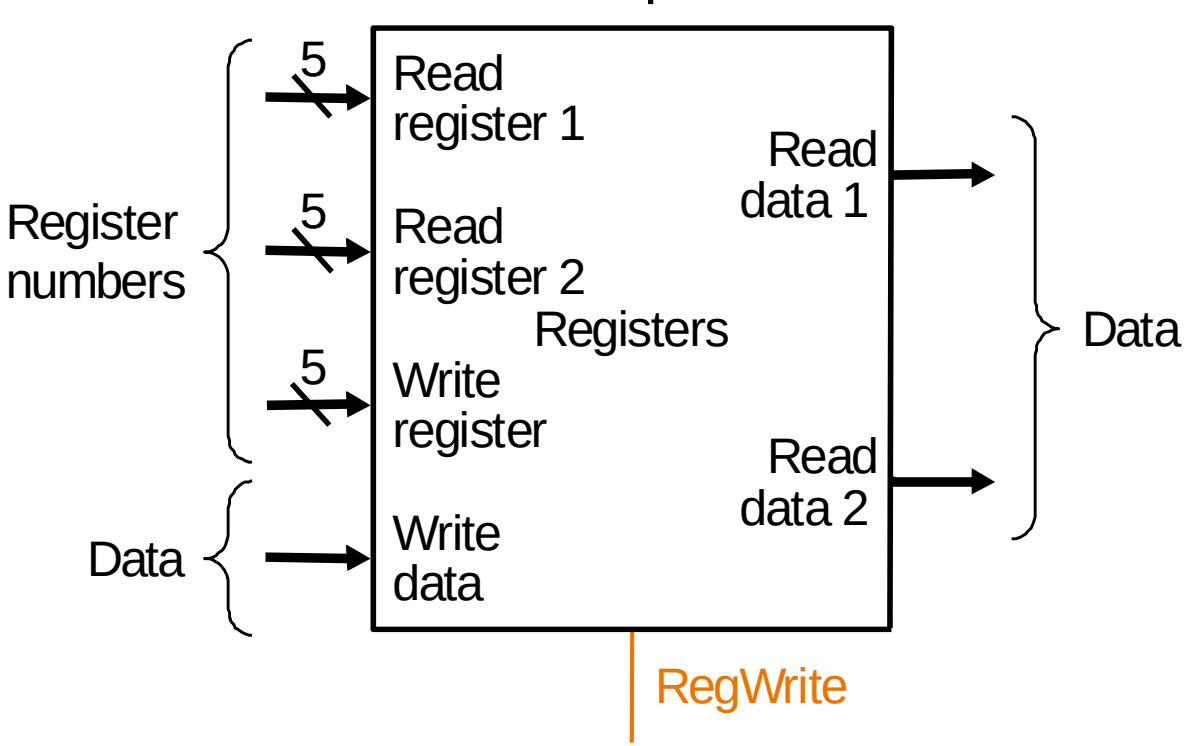
a: *Register File*

It's construction, read, and write operations as discussed previously

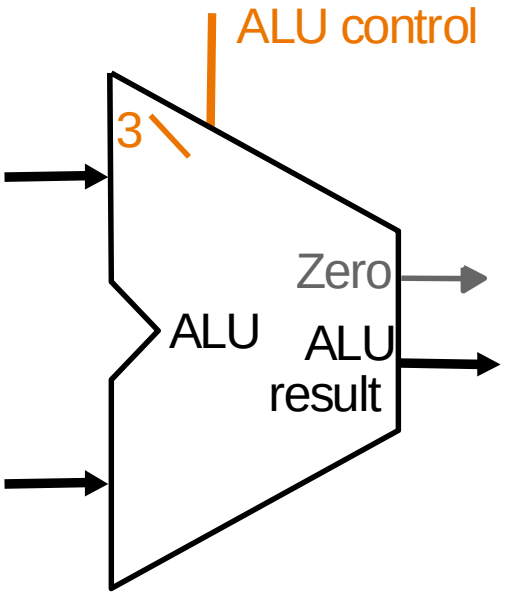
b: *ALU (Arithmetic & Logic Unit)*

Recall the ALU design we have discussed in last two classes

Note the "Zero" output



a. Register File



b. ALU

Functional Units -III

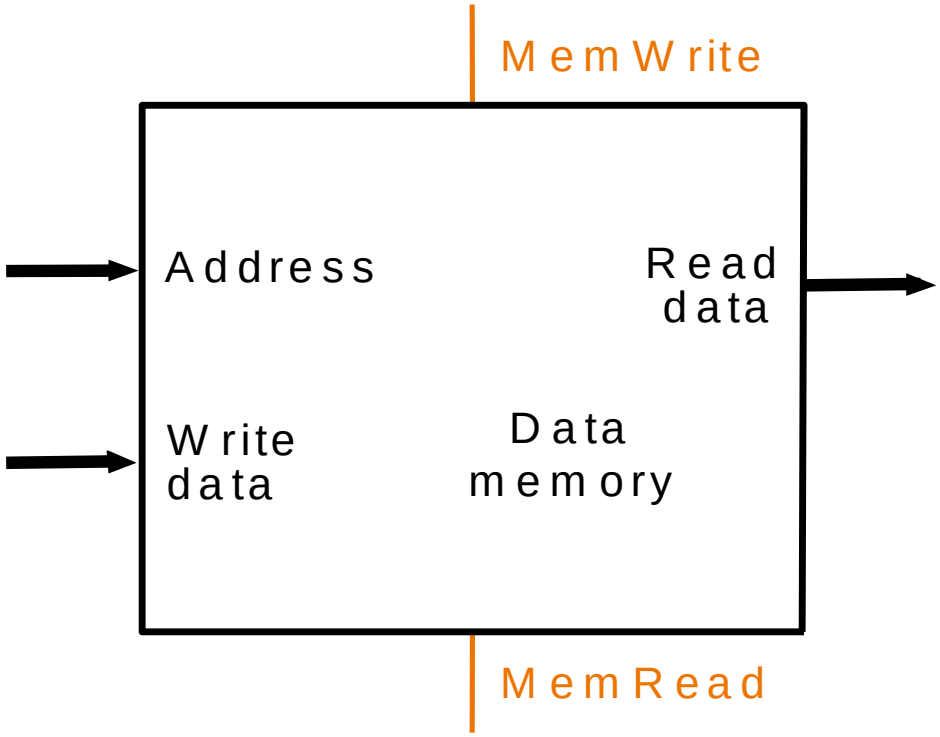
a: *Data Memory Unit*

Similar to Instruction Memory Unit, only that it can written into as well

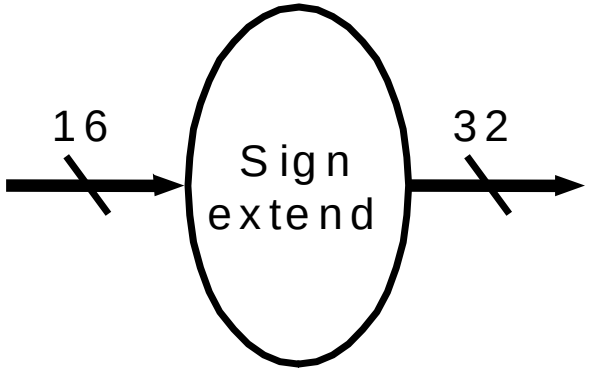
Two input ports for address and data, one output port (for data read out)

Two control signals: for Read and Write operations

b: *Sign Extension Unit* -- Extends 16-bit input operand to 32 bits



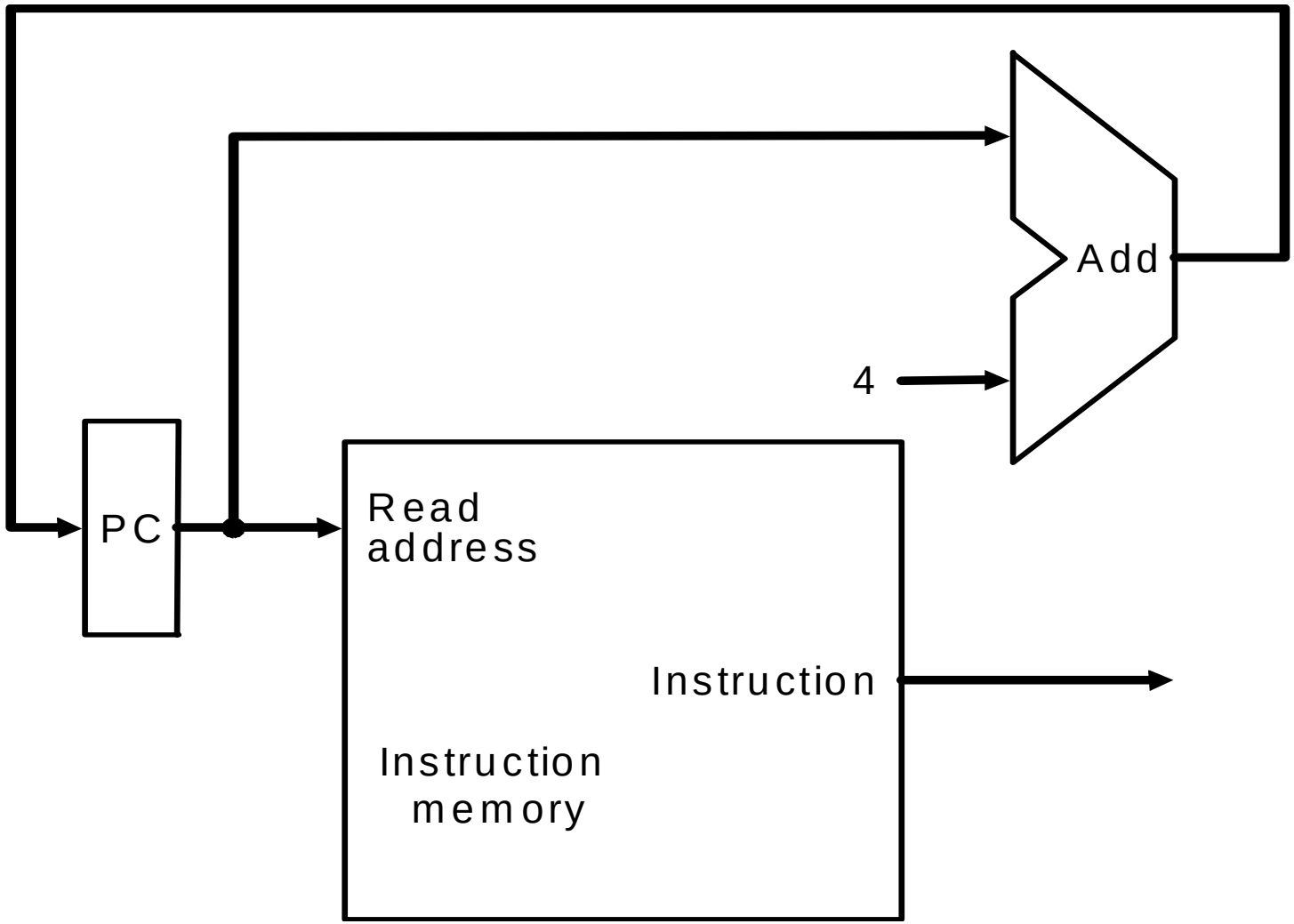
a. Data memory unit



b. Sign-extension unit

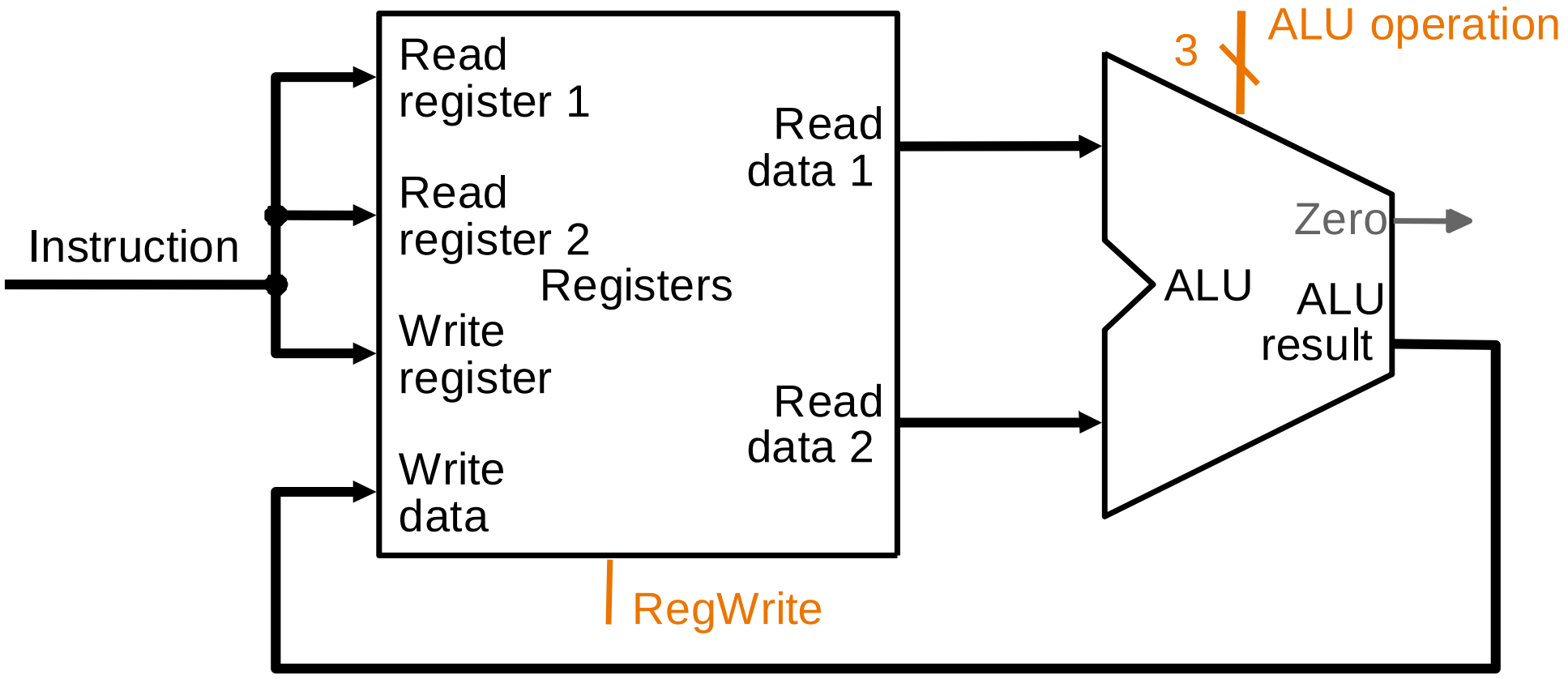
Datapath for Instruction Fetch (Piece I)

- Fetching Instructions and Incrementing the program counter by 4



Datapath for R-type Instructions (Piece II)

- Datapath for R-type Instructions

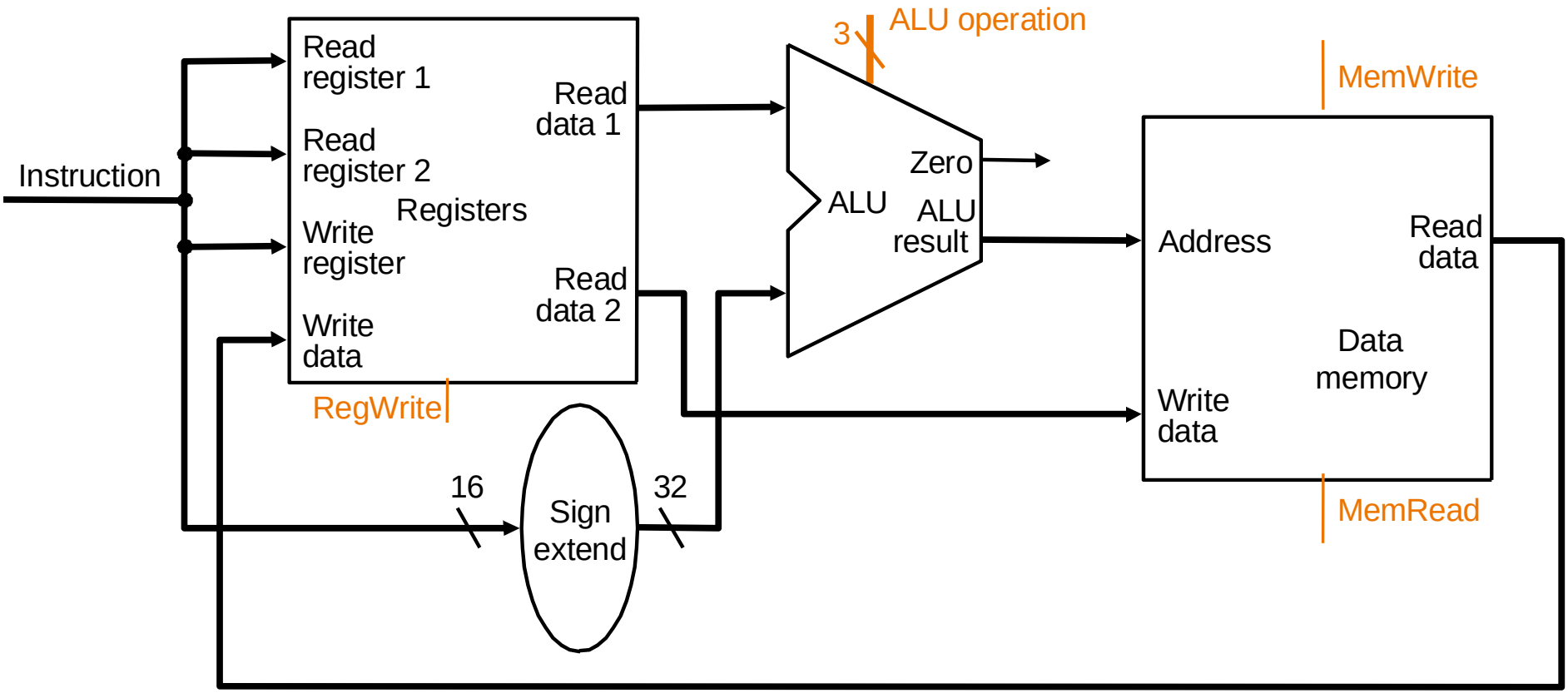


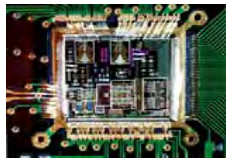
Datapath for Load/Store (Piece III)

- Datapath for load or store

  (1) Register Access; (2) Memory Address calculation; (3) Read/Write

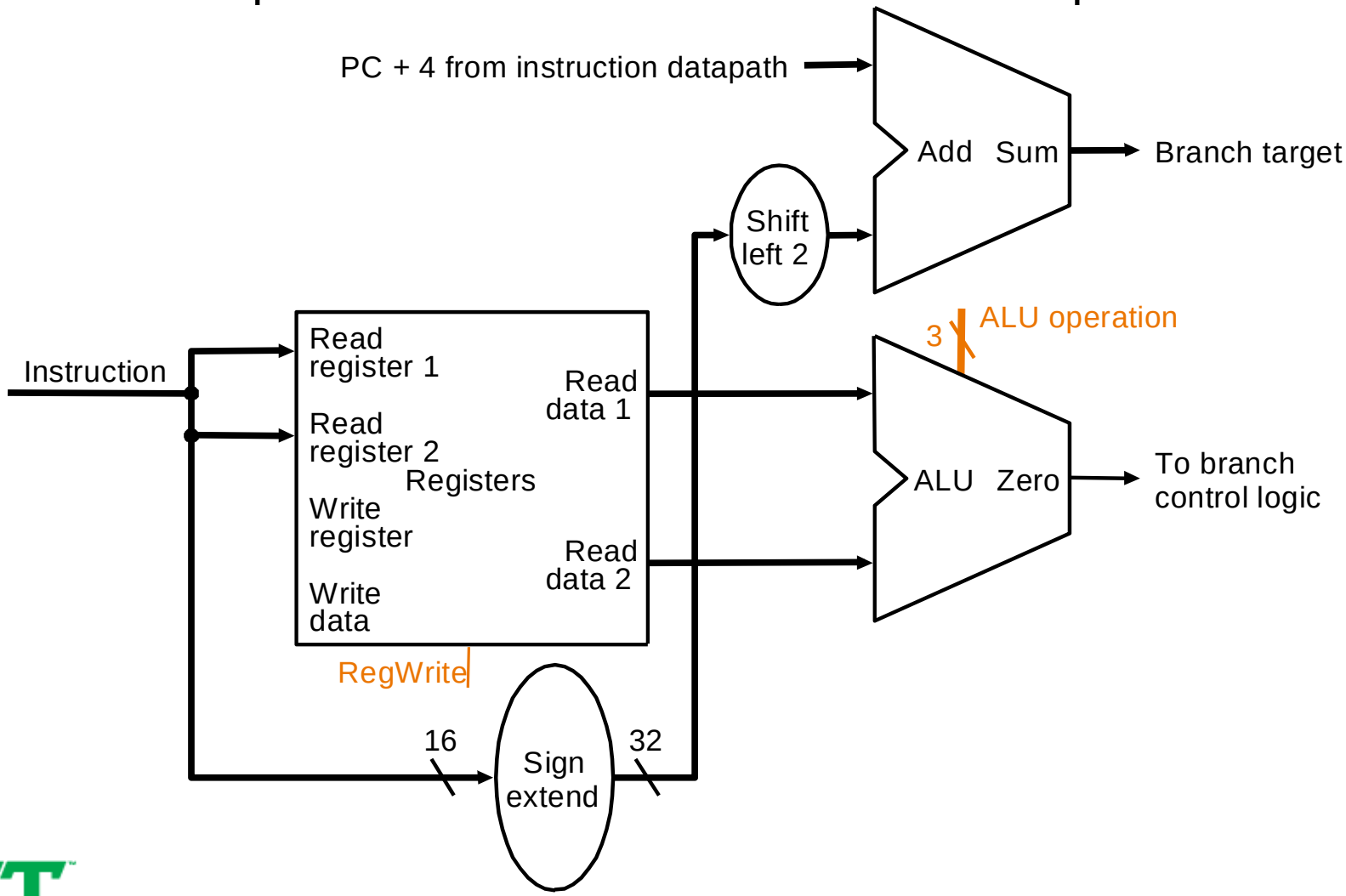
  (4) Write into Register file (if the instruction is a *load*)

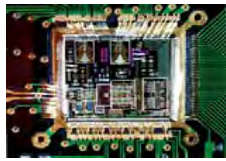




Datapath for Branch (Piece IV)

- Unit “Shift left 2” adds “00” at the low-order end of the sign-extended offset
- Control logic is used to decide whether the incremented PC or branch target should replace the PC based on the “Zero” output of the ALU

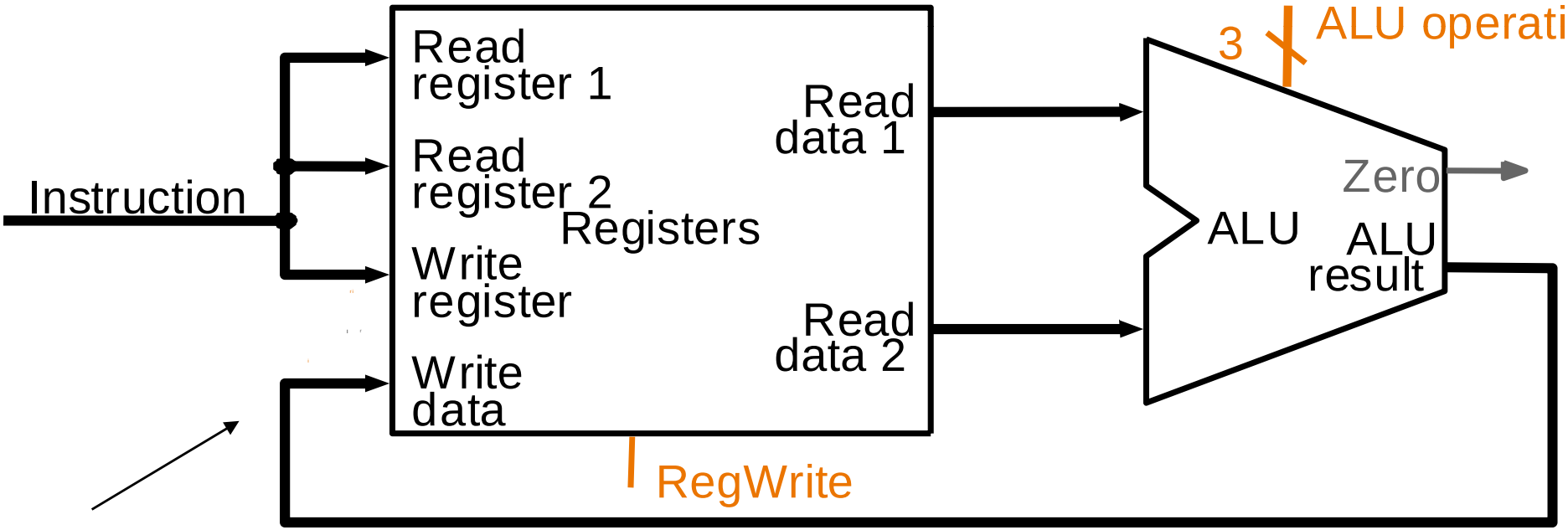




Datapath Construction Strategy

- Now, we have “pieces” of datapath that are capable of performing distinct functions
- We want to “stitch” them together to yield a final datapath that can execute all the instructions (lw, sw, add, sub, and, or, slt, beq, j)
- We will use multiplexors (or muxes for short) for stitching the datapath

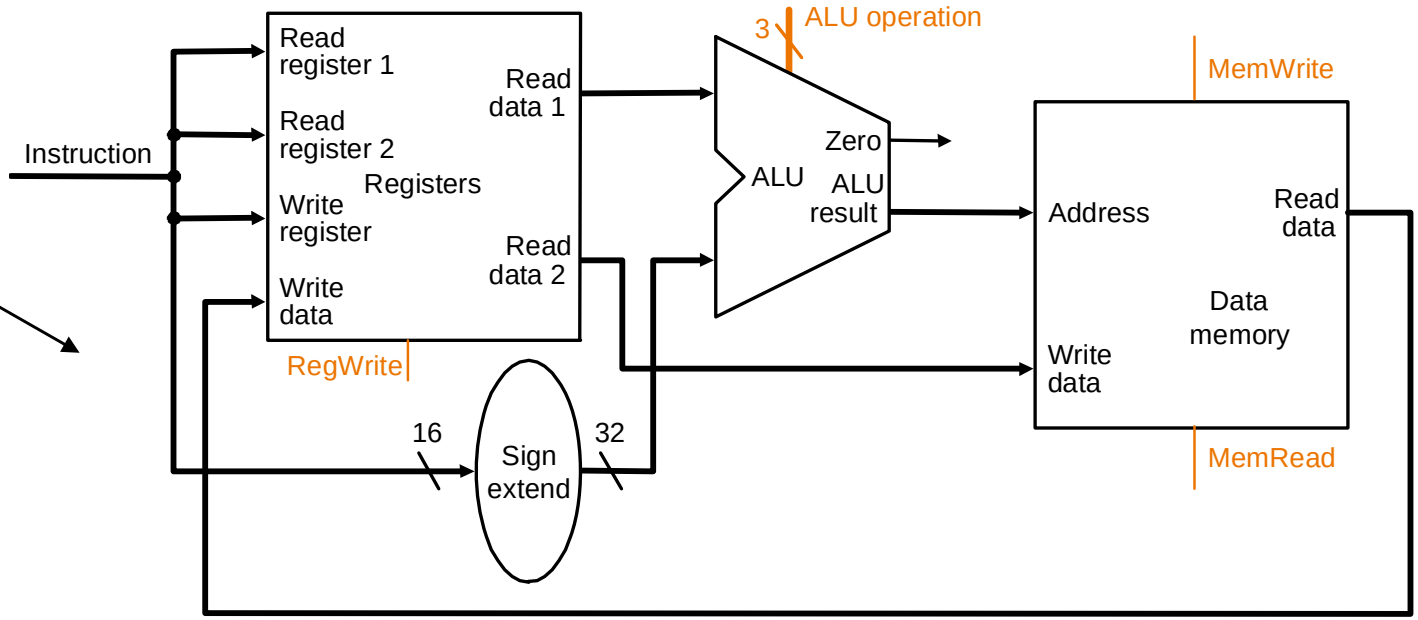
Datapath Construction (Merge Pieces II&III)



Piece II

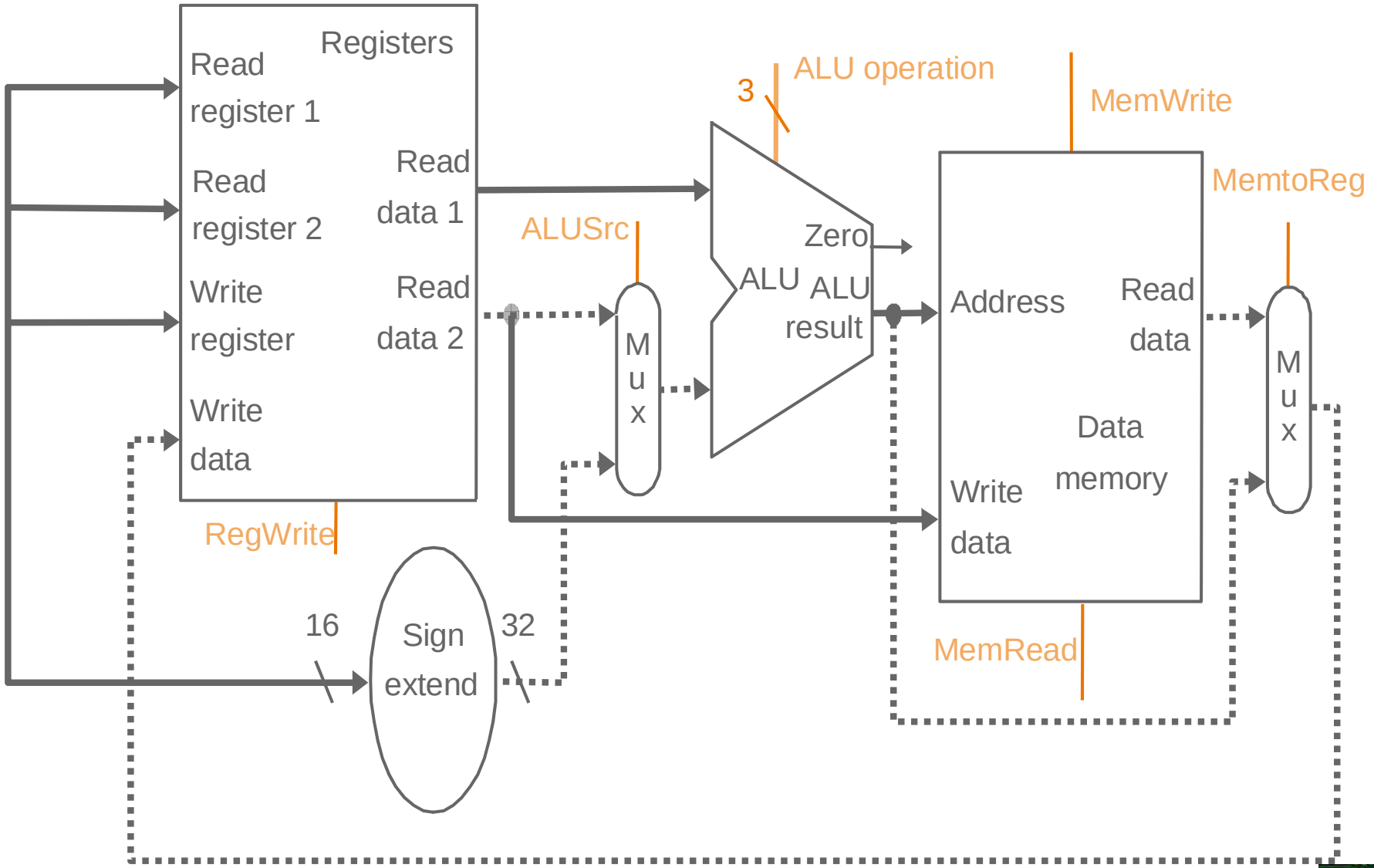
+

Piece III



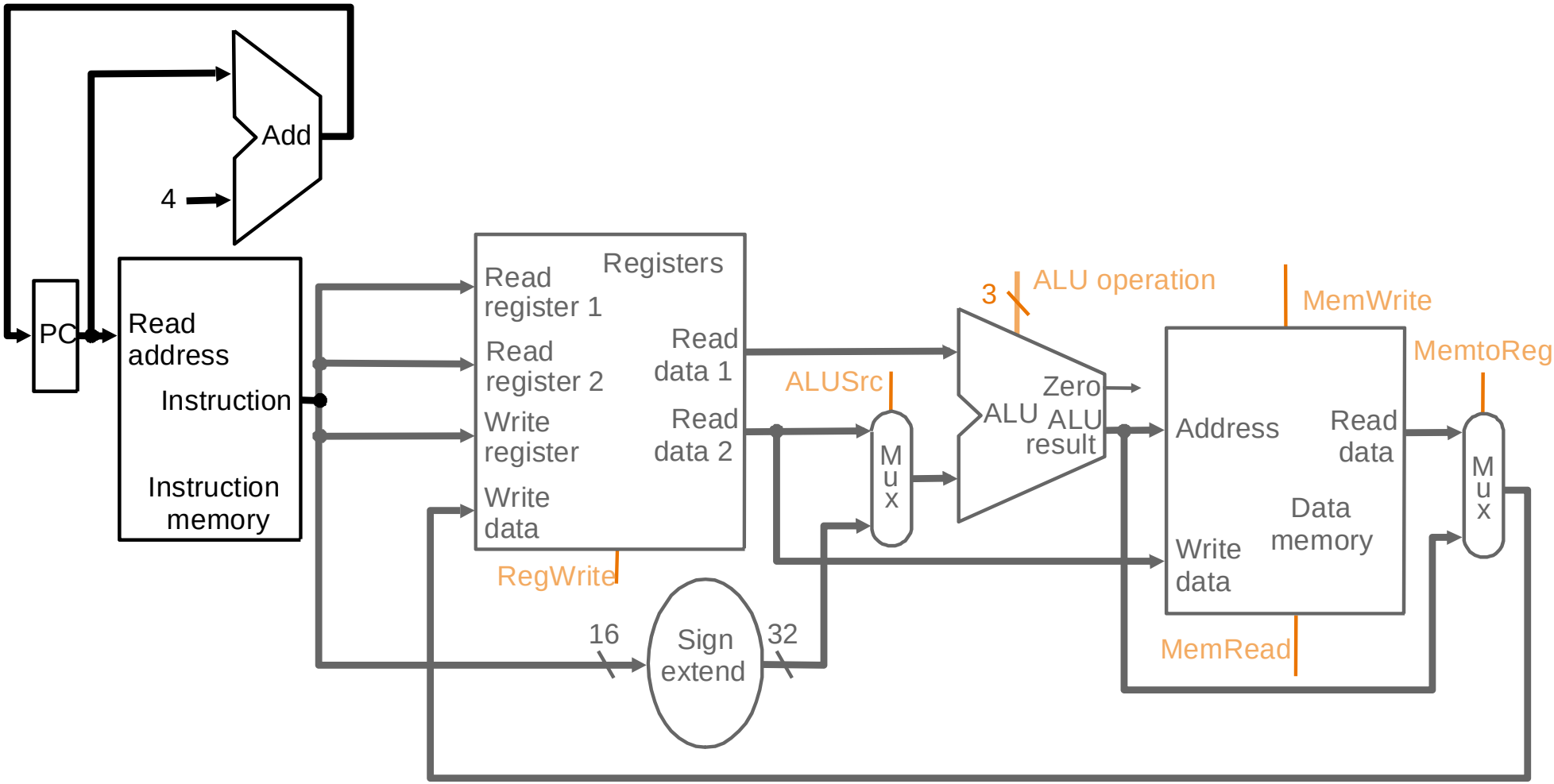
And you will get..

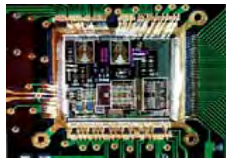
- Rule: Whenever we have more than one input feeding a functional unit, introduce a multiplexor (this gives rise to a control signal, more later..)



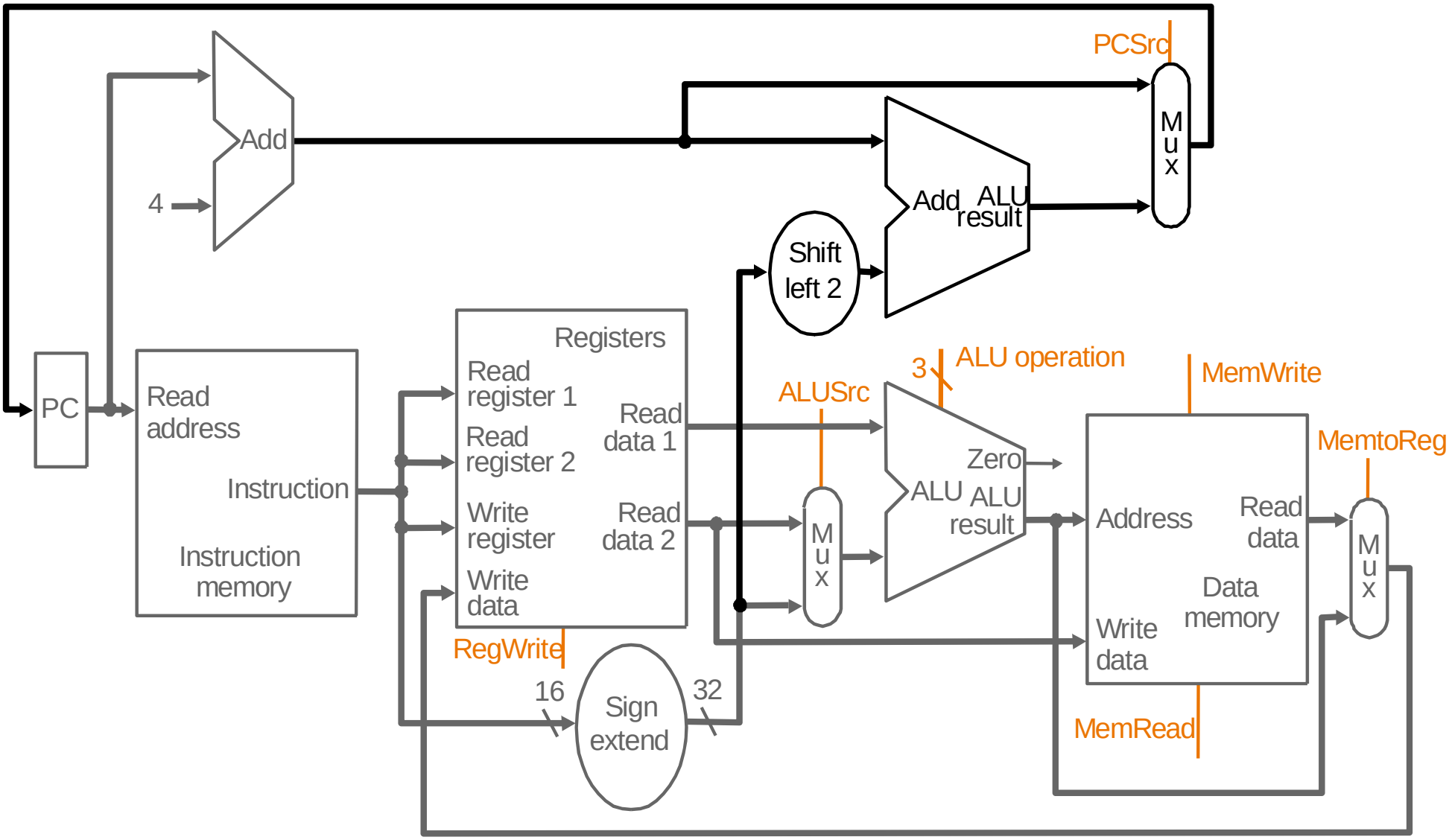
Datapath Construction … (merge Piece I)

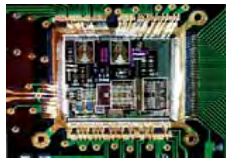
- Just tack the Instruction Fetch and PC increment logic at the front!





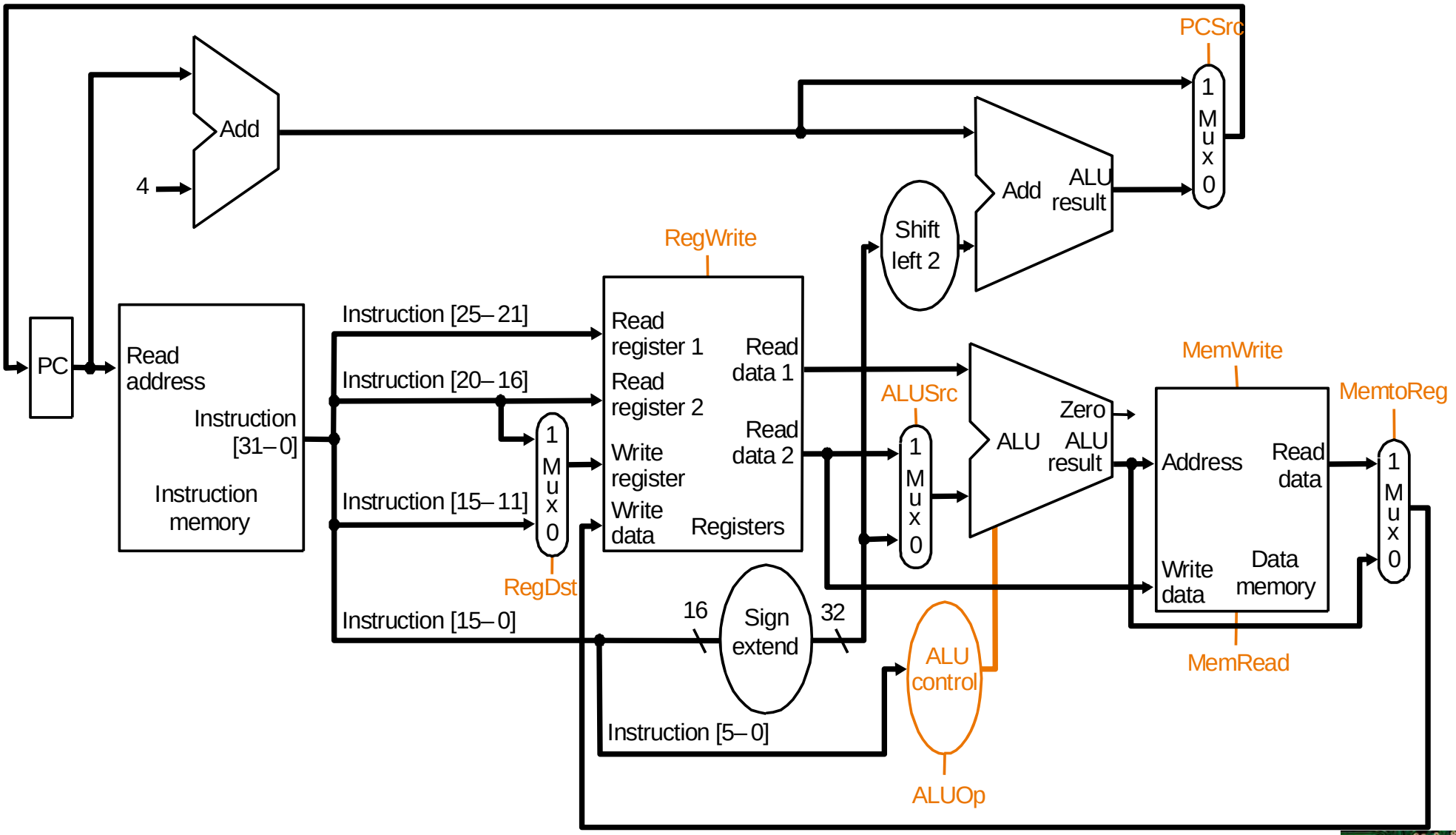
Datapath Construction (merge Piece IV)

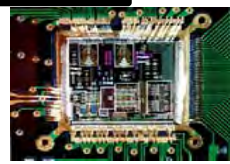




Final Datapath

- Data flows through various “paths” under the influence of control signals
- There are seven control signals (of type Read, Write, or Mux Select)





Defining the Control..

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

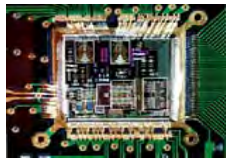
add $8, $17, $18 Instruction Format:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ALU's operation based on instruction type and function code
- We will design two control units:

(1) ALU Control to generate appropriate function select signals for the ALU

(2) Main Control to generate signals for functional units other than the ALU



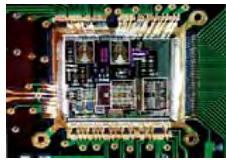
Defining the ALU Control ... Contd.

- e.g., what should the ALU do with this instruction
- Example: lw $1, 100($2)

35	2	1	100
op	**rs**	**rt**	**16 bit offset**

- ALU control input

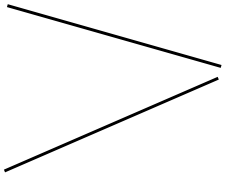
  000 AND
  001 OR
  010 add
  110 subtract
  111 set-on-less-than



ALU Control Design

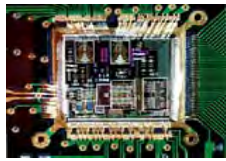
- Must describe hardware to compute 3-bit ALU control input
  - given instruction type

    00 = lw, sw
    01 = beq,
    10 = arithmetic

    ALUOp
    computed from instruction type
  - function code for arithmetic
- ALU Control inputs – How are they determined?

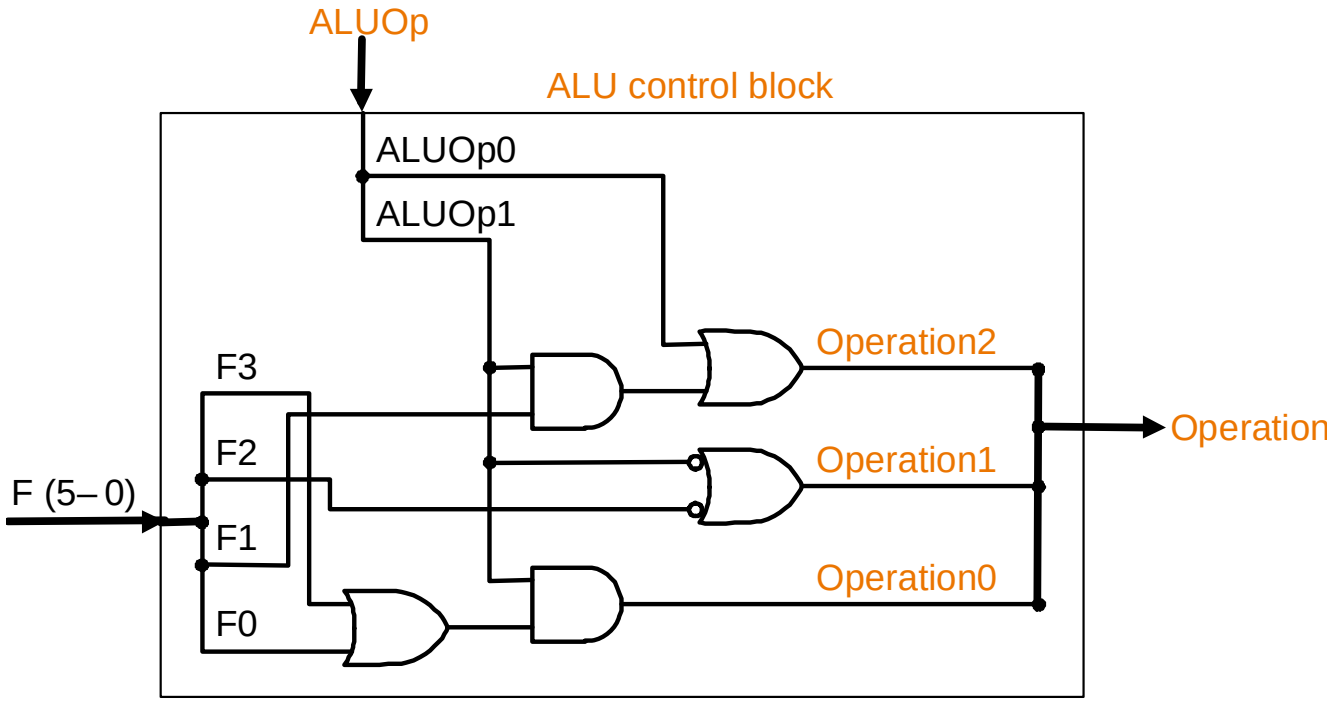
Instruction Opcode	ALUOp	Instruction Operation	Funct Field	Desired ALU Action	ALU Control Operation
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111



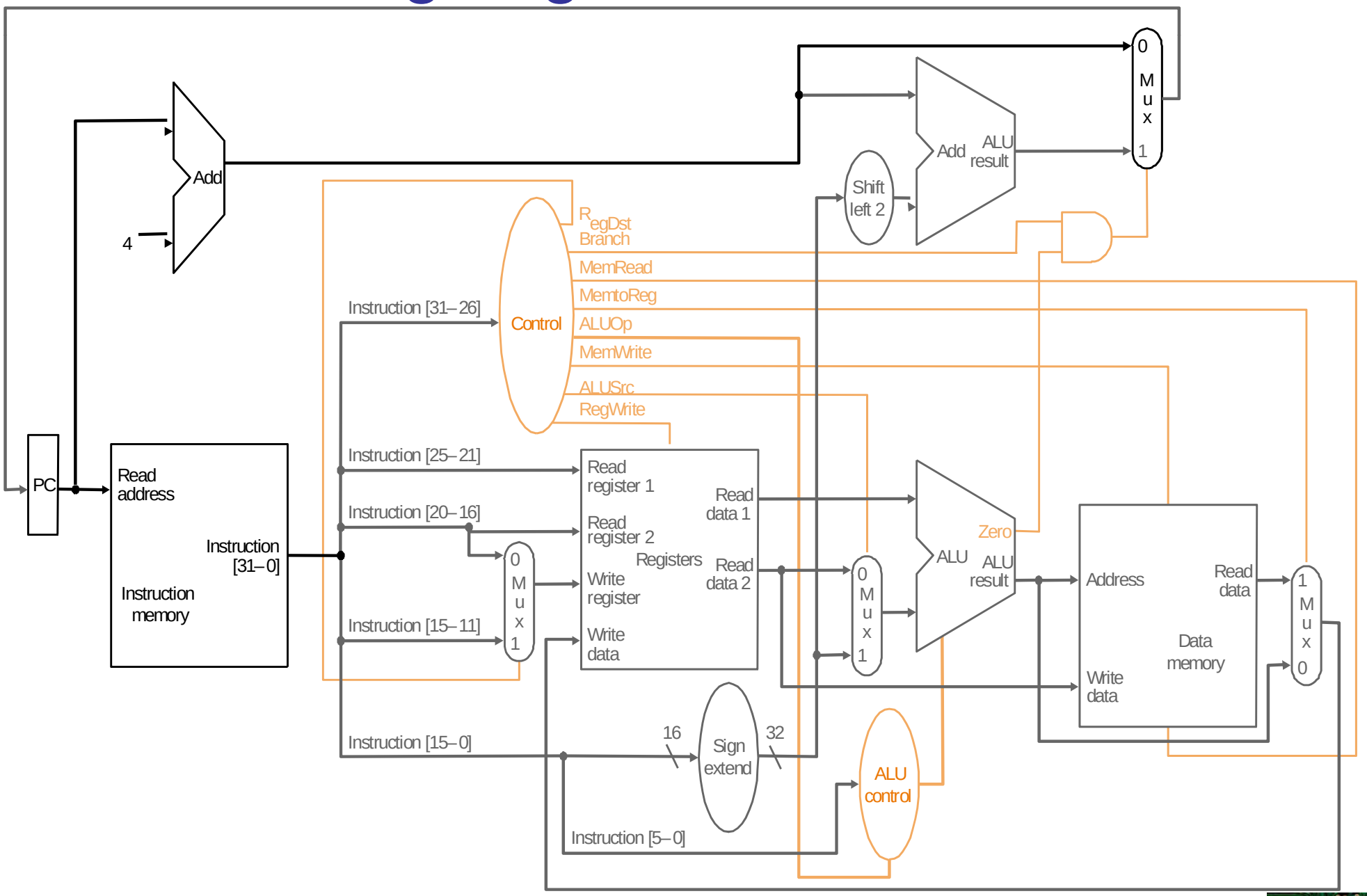
ALU Control - Truth Table & Implementation

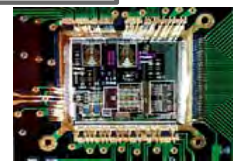
Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111



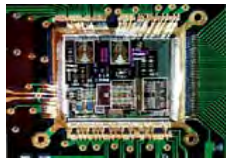
Designing the Main Control





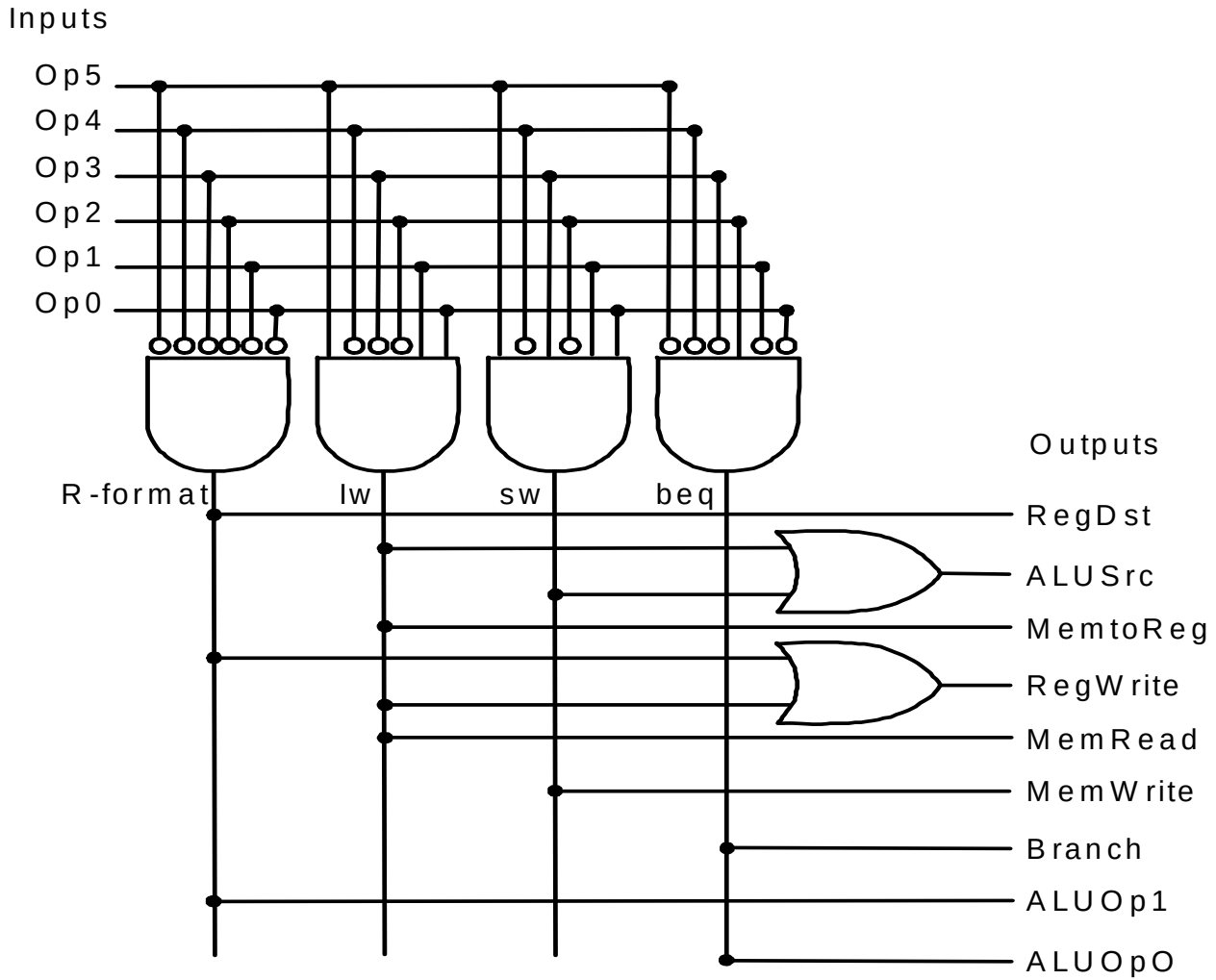
Control Signals and their Effects

Signal Name	Effect When deasserted	Effect when asserted
RegDst	The register destination number for the Write	The register destination number for the Write
	register comes from the rt field (bits 20-16)	register comes from the rd field (bits 15-11)
RegWrite	NONE	The register on the Write register input is
		written with the value on the Write data input
ALUSrc	The second ALU operand comes from the	The second ALU operand is the sign-extended
	second register file output (Read data 2)	lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that	The PC is replaced by the output of the adder
	computes the value of PC + 4.	that computes the branch target
MemRead	None	Data Memory contents designated by the address
		input are put on the Read data output
MemWrite	None	Data memory contents designated by the address
		input are replaced by the value on the Write data input
MemtoReg	The value fed to the register Write data input	The value fed to the register Write data input comes
	comes from the ALU	from the data memory.



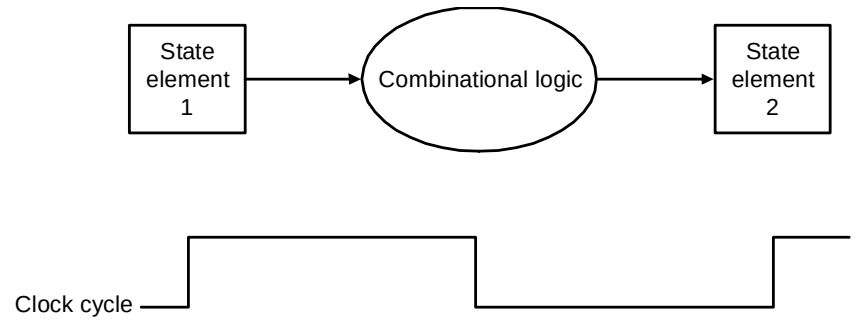
Main Control: Truth Table & Implementation

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

How does the single cycle datapath work?

- Let us understand this by highlighting the portions of the datapath when an R-type instruction is executed
- For an R-type instruction we go through the following phases:

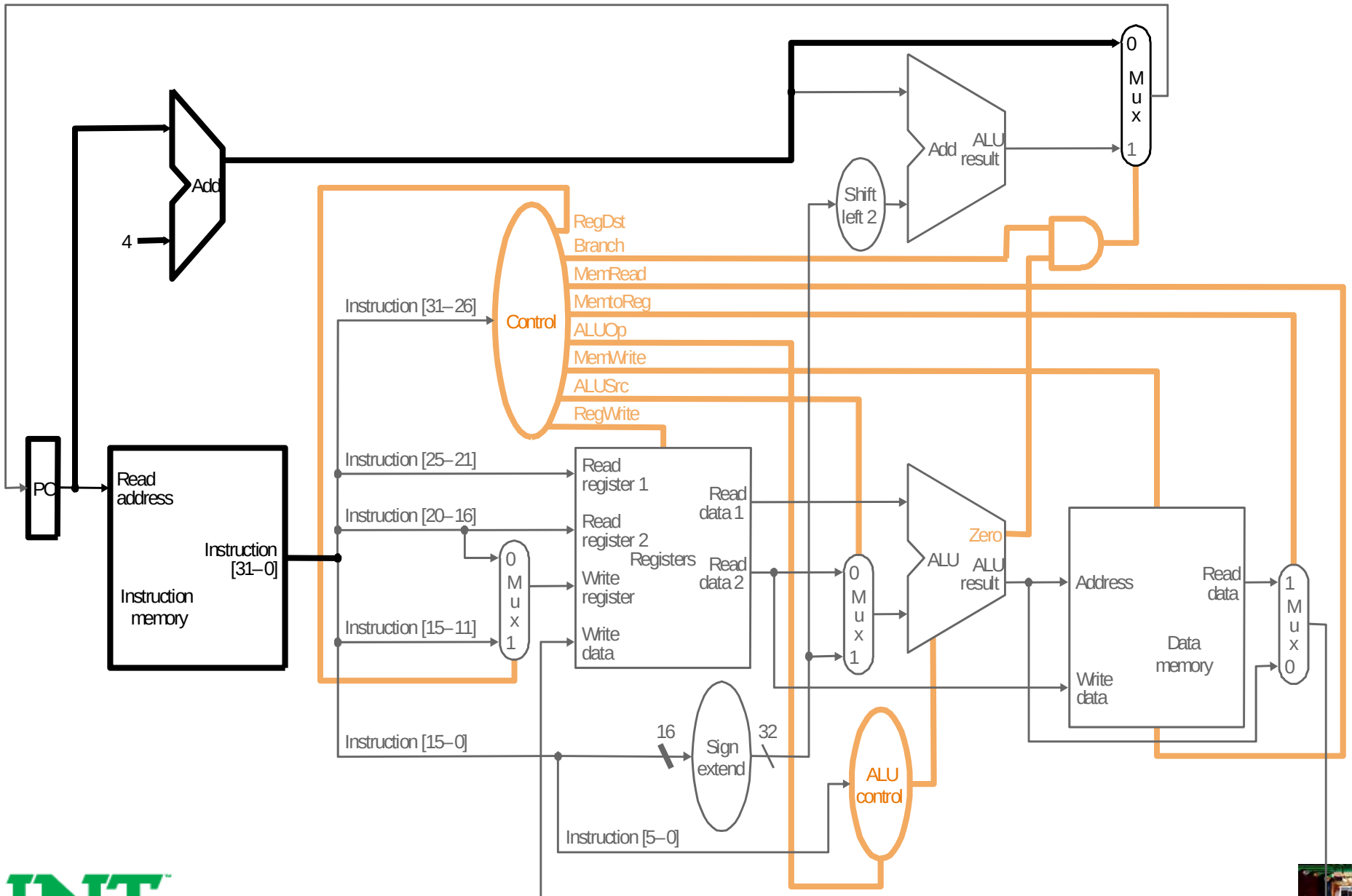
  Phase 1: Instruction Fetch

  Phase 2: Register File Read

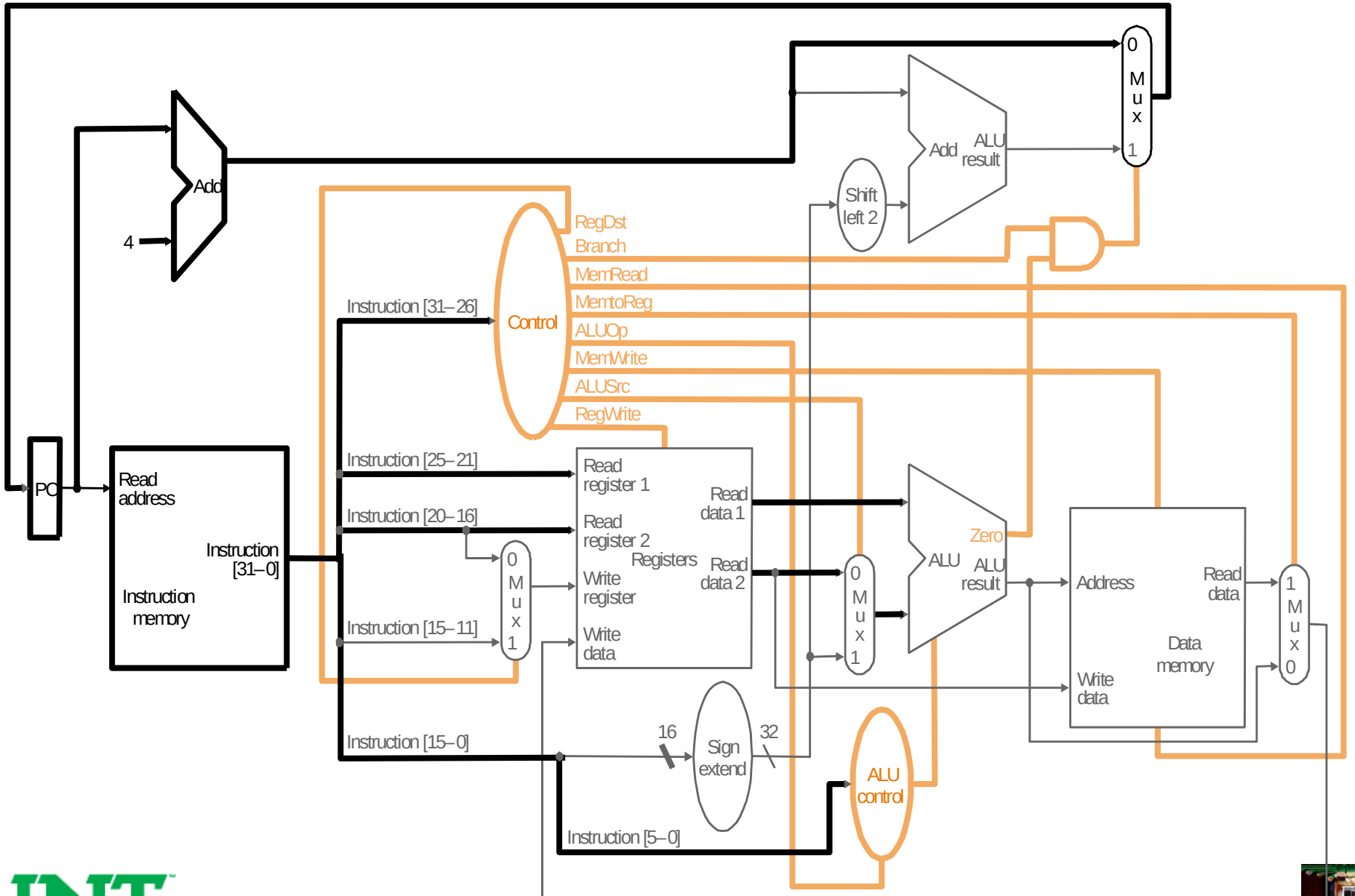
  Phase 3: ALU execution

  Phase 4: Write the Result into the Register File
- NOTE: All the four phases are completed in only ONE clock cycle and hence it is a “single cycle implementation”

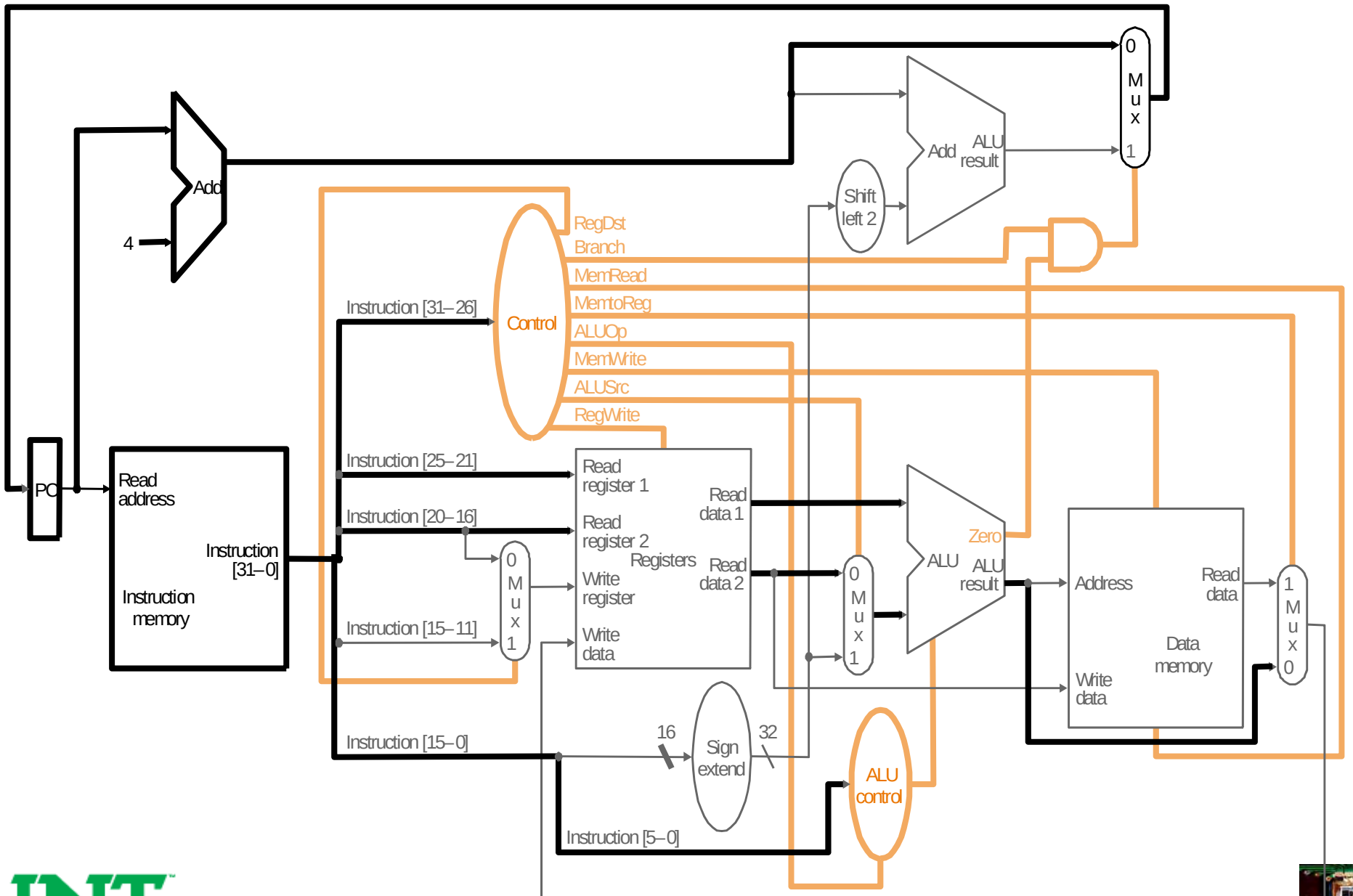
R-type Instruction – Phase 1
(Instruction Fetch)



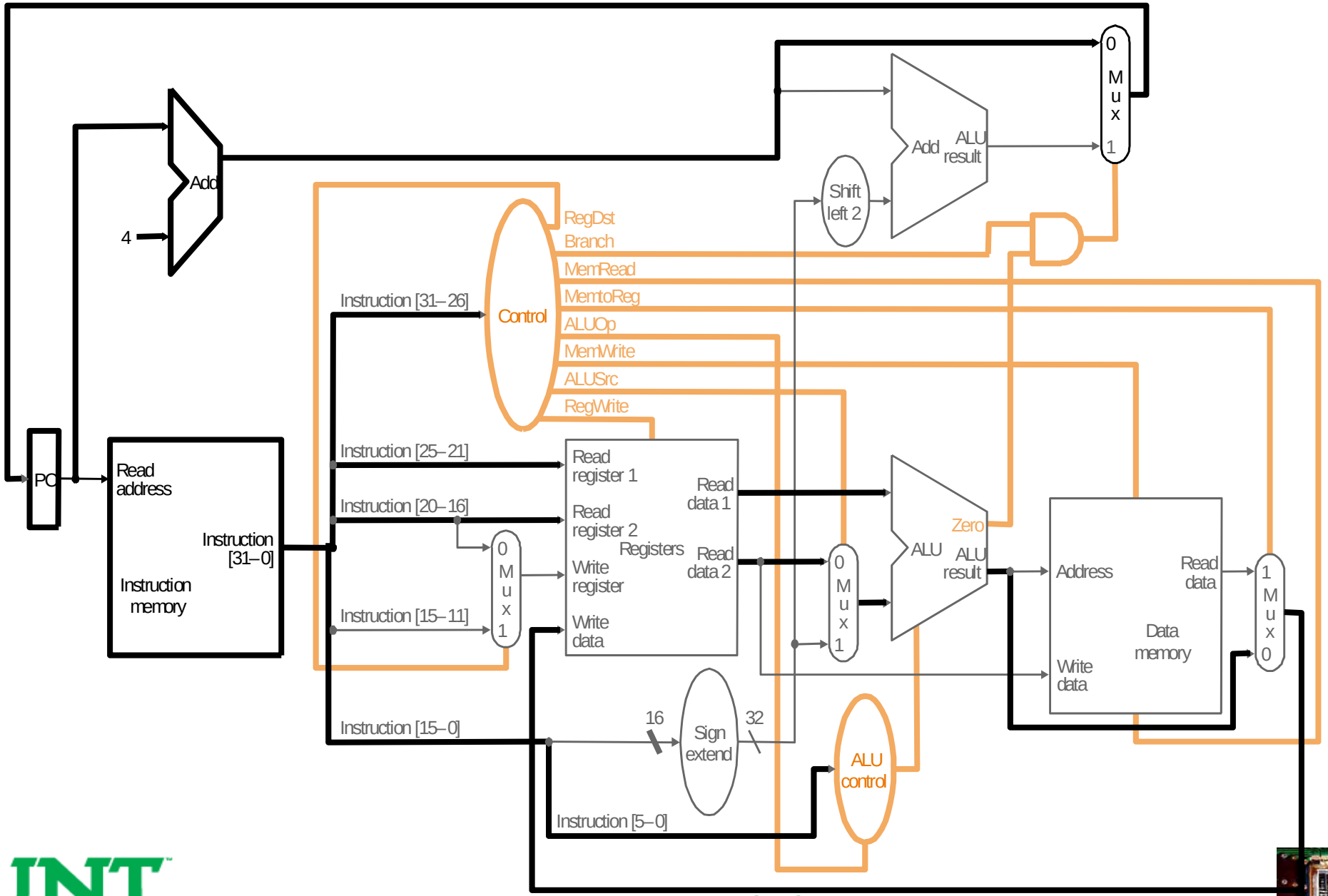
R-type Instruction – Phase 2
(Register Read)



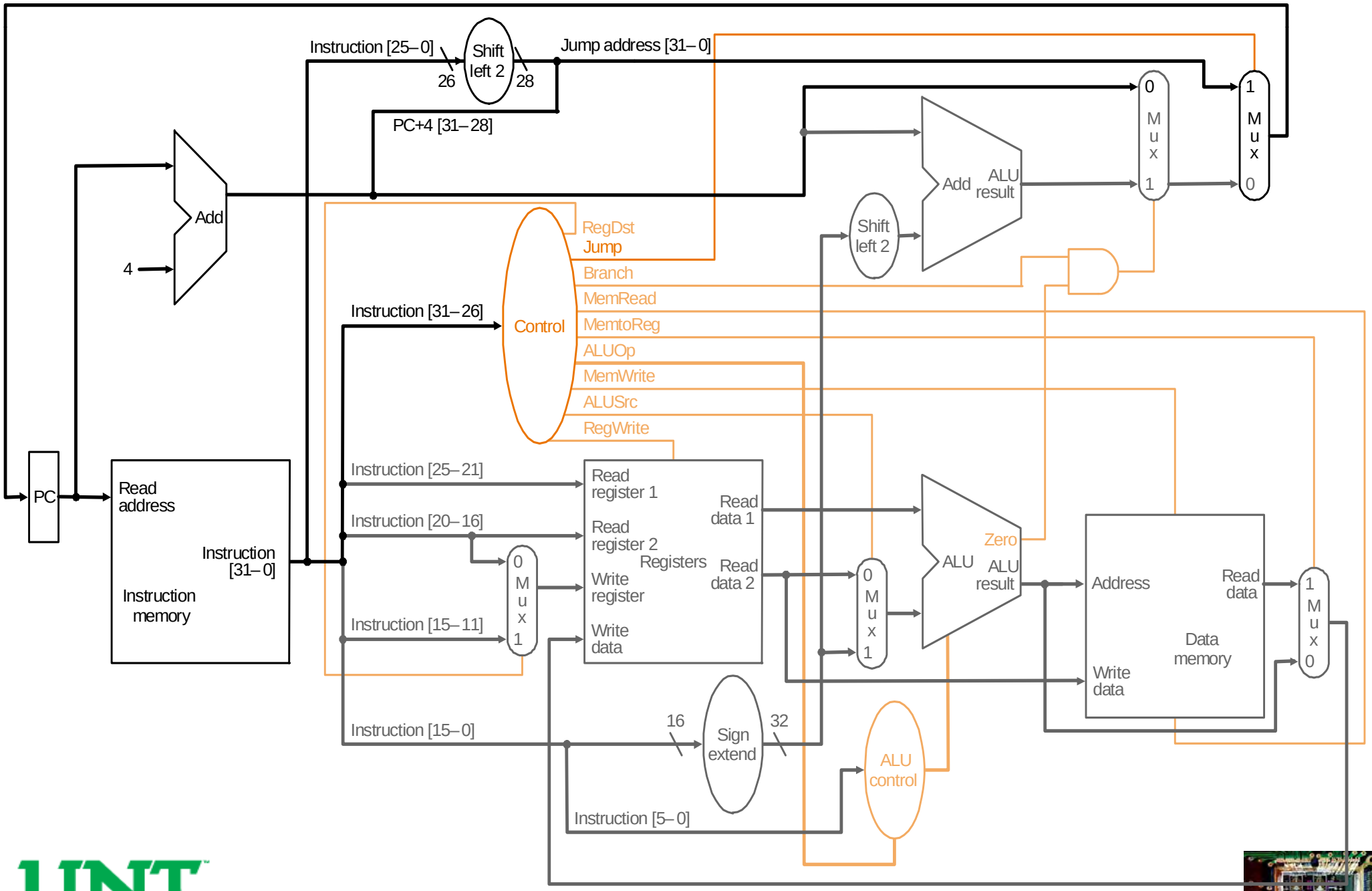
R-type Instruction – Phase 3
(ALU execution)



R-type Instruction – Phase 4
(Write the Result)



How do we handle jump?

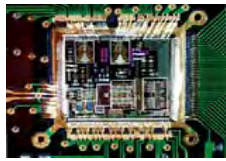


Single Cycle Implementation: Summary

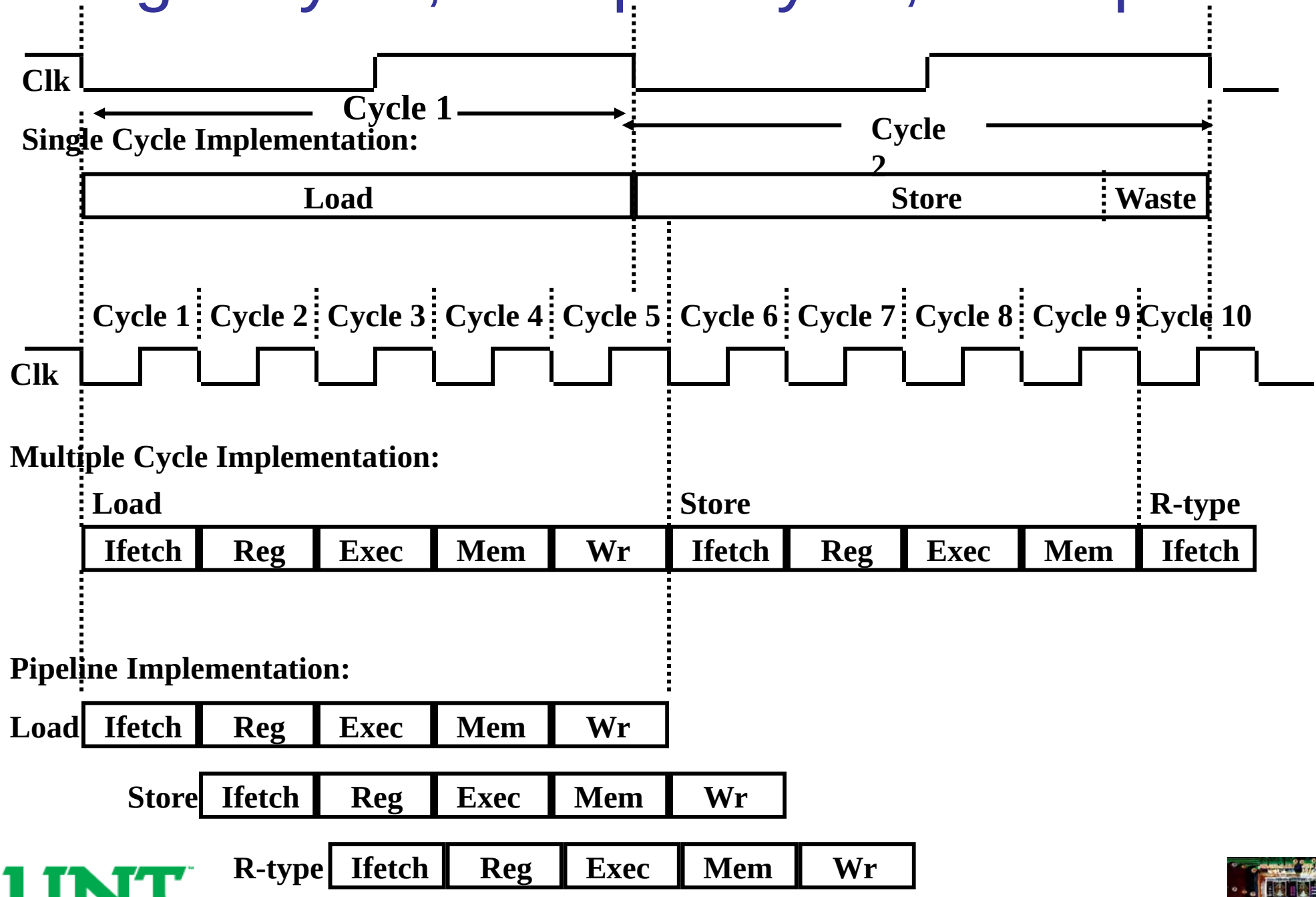
- All instructions are executed in only clock cycle
- We built a single cycle datapath from scratch
- We designed appropriate controller to generate correct correct signals
- All instructions are not born equal; that some require more work, some less => disadvantage of single cycle implementation is that the slowest instruction determines the clock cycle width
- In reality, no body implements single cycle approach.
- Given the single cycle datapath, you should be able to "highlight" active portions of the datapath for any given instruction.

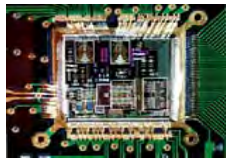
Single Cycle Implementation - Issues

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
  - Cycle width determined by the slowest instruction
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:



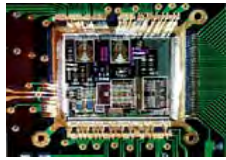
Single Cycle, Multiple Cycle, vs. Pipeline



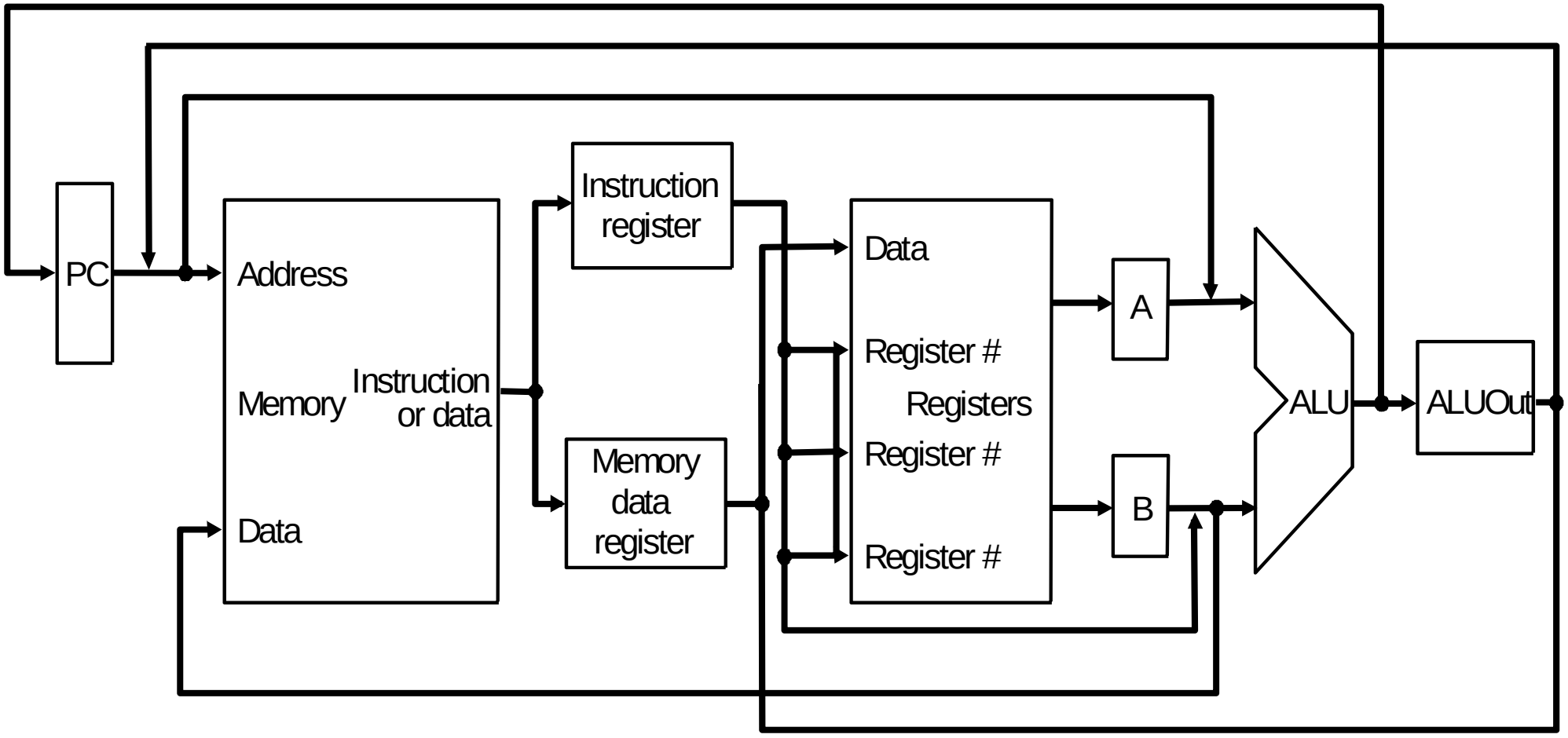


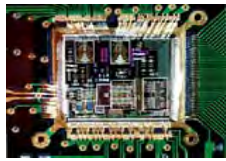
Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control
- 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



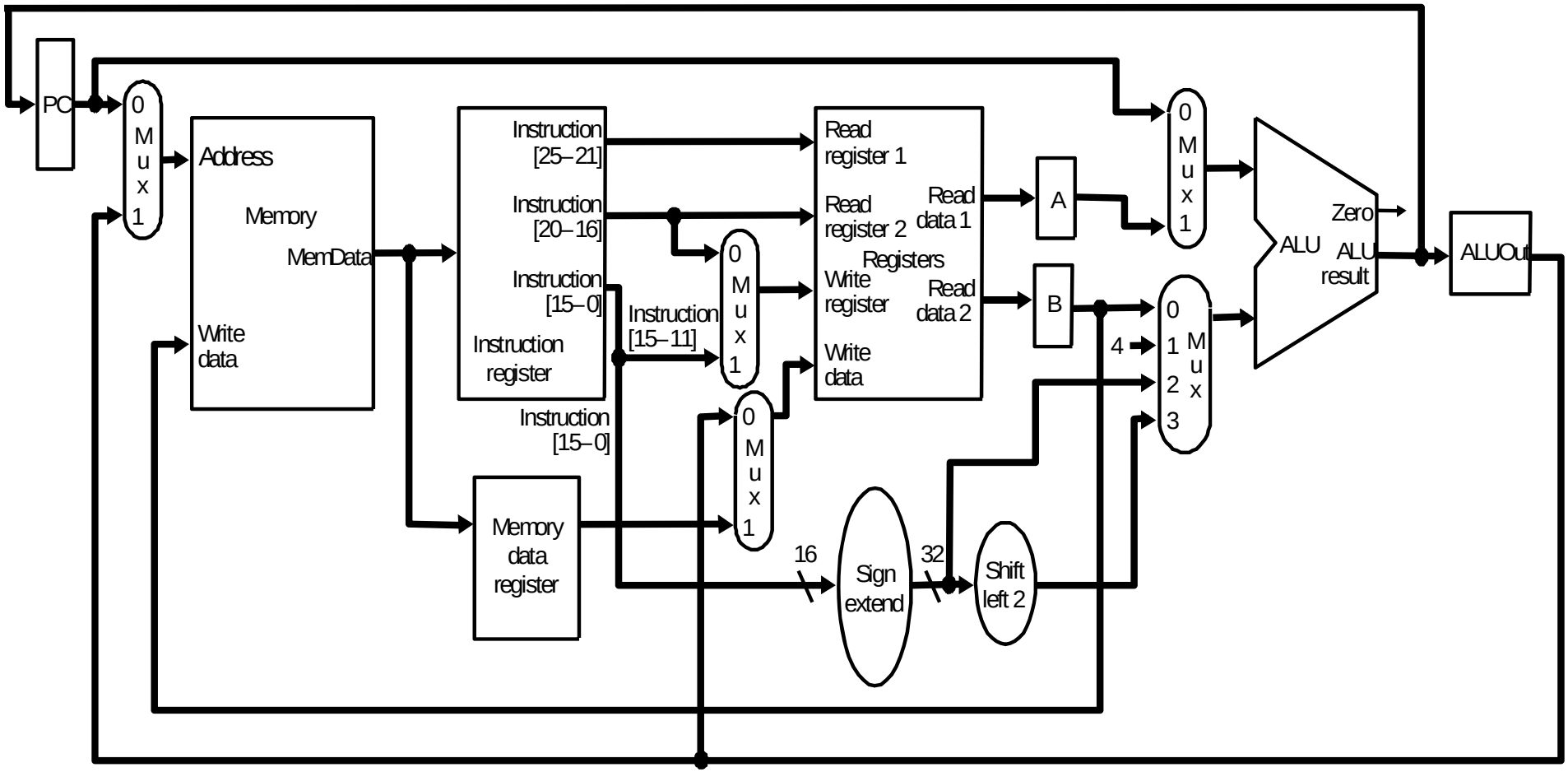
Multicycle Approach – High Level View

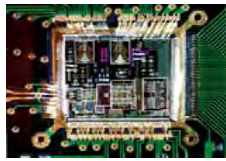




Multicycle Approach

- handles the basic instructions





Five Execution Steps

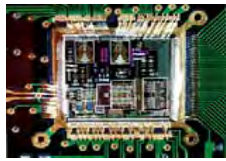
- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

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

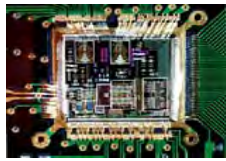
```
IR <= Memory[PC];
PC <= PC + 4;
```

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

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

Step 3: (instruction dependent)

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

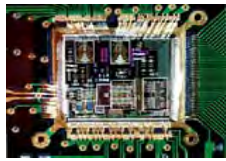
```
ALUOut <= A + sign-extend(IR[15-0]);
```

- R-type:

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

- Branch:

```
if (A==B) PC <= ALUOut;
```

Step 4: (R-type or memory-access)

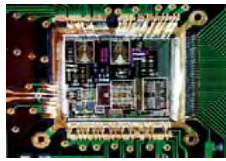
- Loads and stores access memory

```
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.



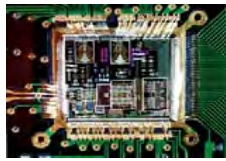
Step 5: Write-back step

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

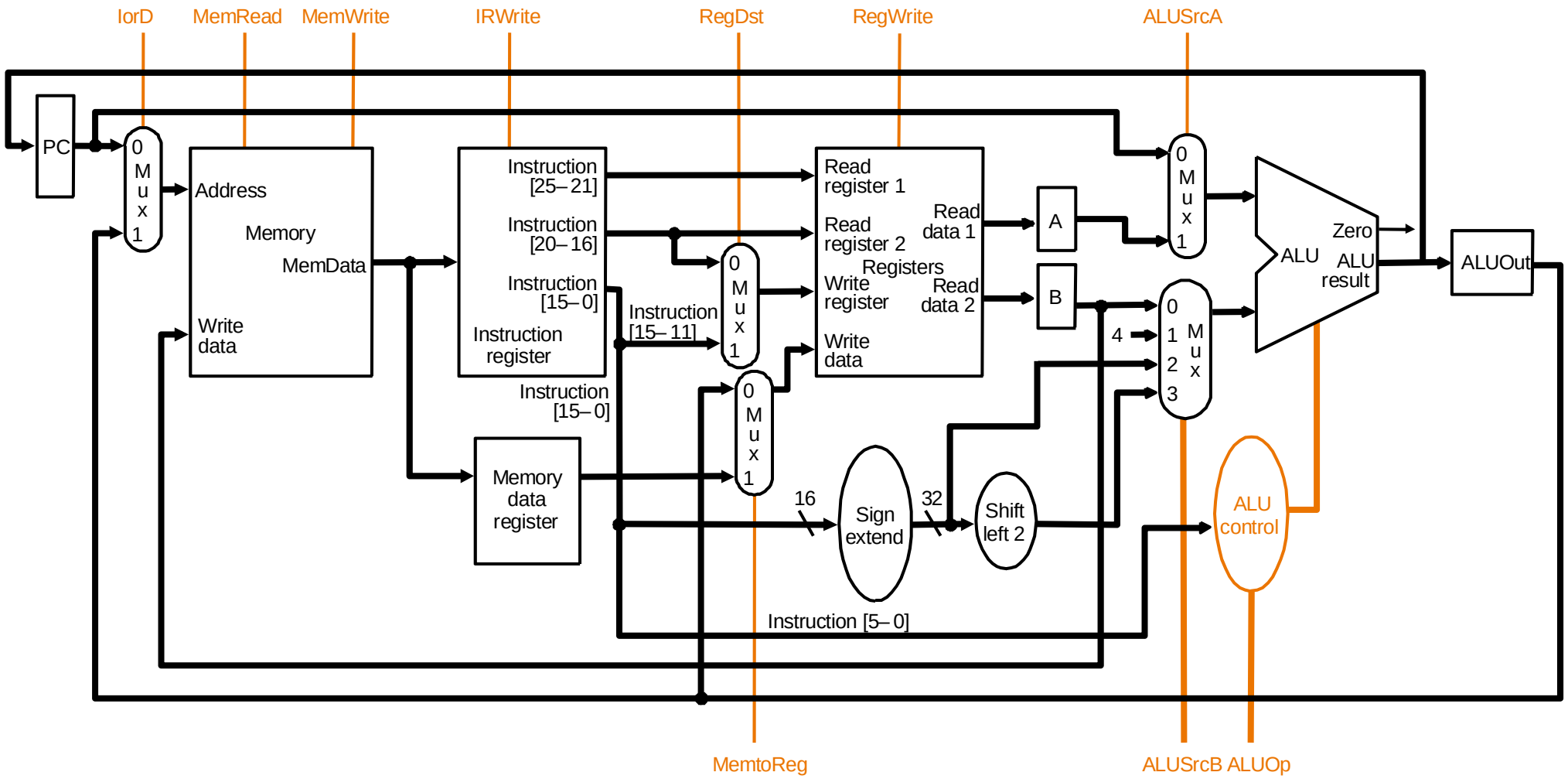
Summary of Steps taken to execute any instruction class.

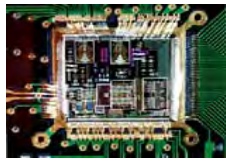
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC]			
	PC <= PC + 4			
Instruction	A <= Reg [IR[25-21]]			
decode/register fetch	B <= Reg [IR[20-16]]			
	ALUOut <= PC + (sign-extend (IR[15-0]) << 2)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

- This sequence suggests what controller must do on each clock-cycle.

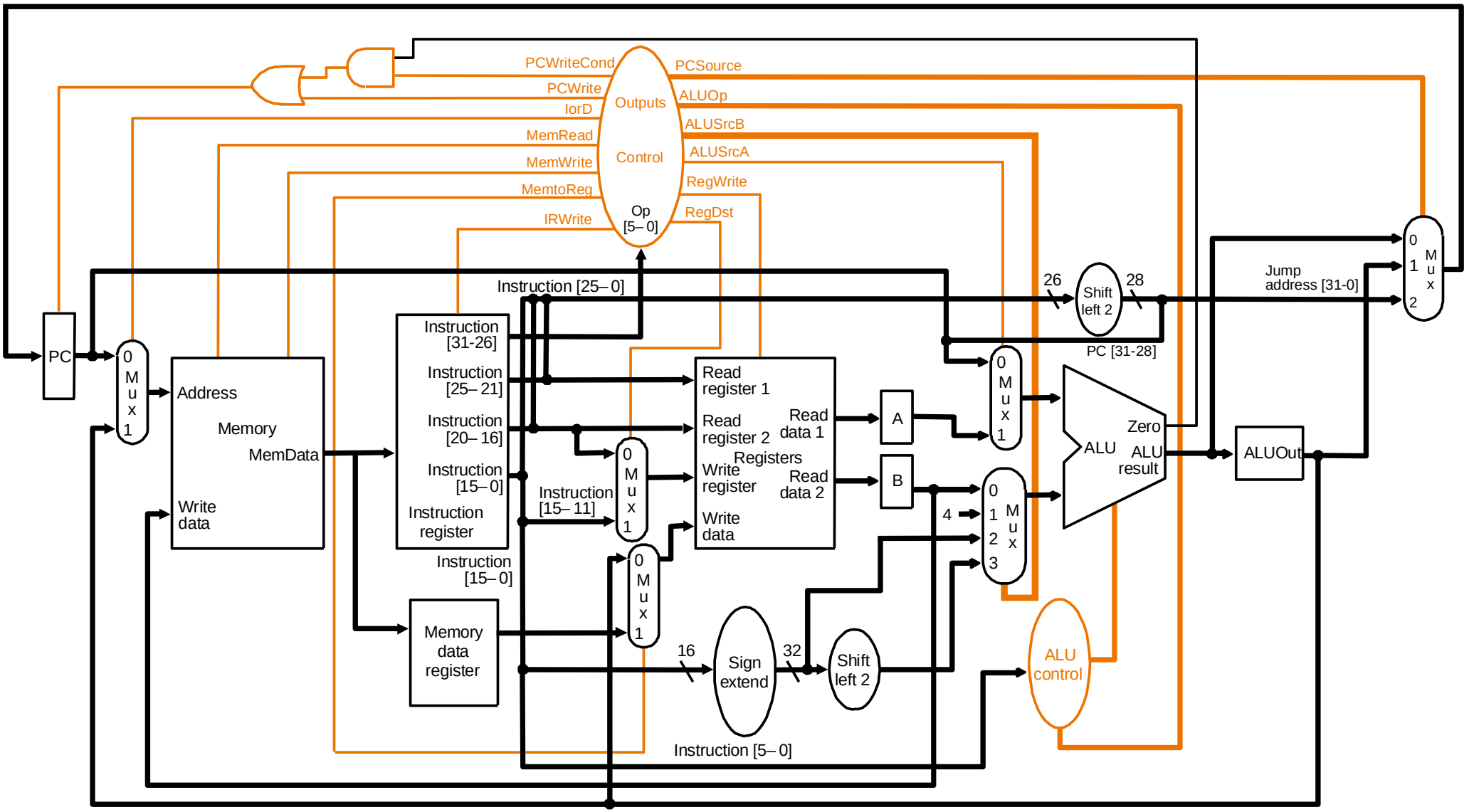


Multicycle Datapath with Control Lines



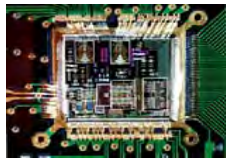


Multicycle Datapath with Controller



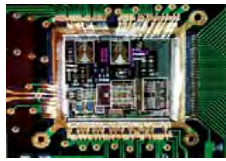
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - Use microprogramming
- Implementation can be derived from specification



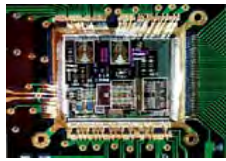
Actions of the 1-bit control signals

Signal Name	Effect When deasserted	Effect when asserted
RegDst	The register destination number for the Write	The register destination number for the Write
	register comes from the rt field	register comes from the rd field
RegWrite	NONE	The general purpose register selected by the Write
		register number is written with the value of the Write
		data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of Memory at the location specified by the
		Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the
		Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data inpu	The value fed to the register file Write datainput comes
	comes from ALUOut	from the MDR.
IorD	The PC is used to supply the address to the	ALUOut is used to supply the address to the memory
	memory unit.	unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource
PCWriteCond	None	The PC is written if the Zero output from the ALU is activ



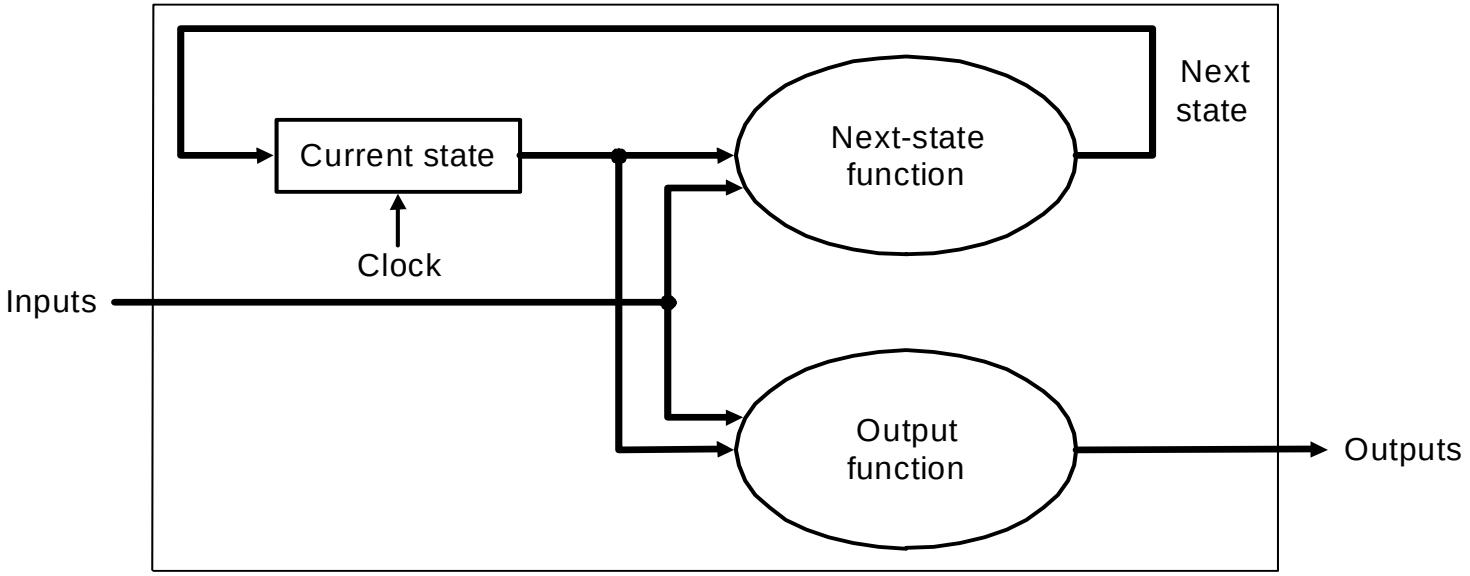
Actions of the 2-bit control signals

Signal Name	Value	Effect when asserted
	"00"	The ALU performs an add operation.
ALUOp	"01"	The ALU performs a subtract operation.
	"10"	The funct field of the instruction determines the ALU operation
	"00"	The second input to the ALU comes from the B register.
ALUSrcB	"01"	The second input to the ALU is the constant 4.
	"10"	The second input to the ALU is the sign-extended,
		lower 16 bits of the IR.
	"11"	The second input to the ALU is the sign-extended,
		lower 16 bits of the IR shifted left 2 bits
	"00"	Output of the ALU (PC + 4) is sent to the PC for writing.
PCSource	"01"	The contents of the ALUOut (the branch target address)
		are sent to the PC for writing.
	"10"	The jump target address (IR[25-0] shifted left 2 bits and
		concatenated with PC + 4[31-28]) is sent to the PC
		for writing

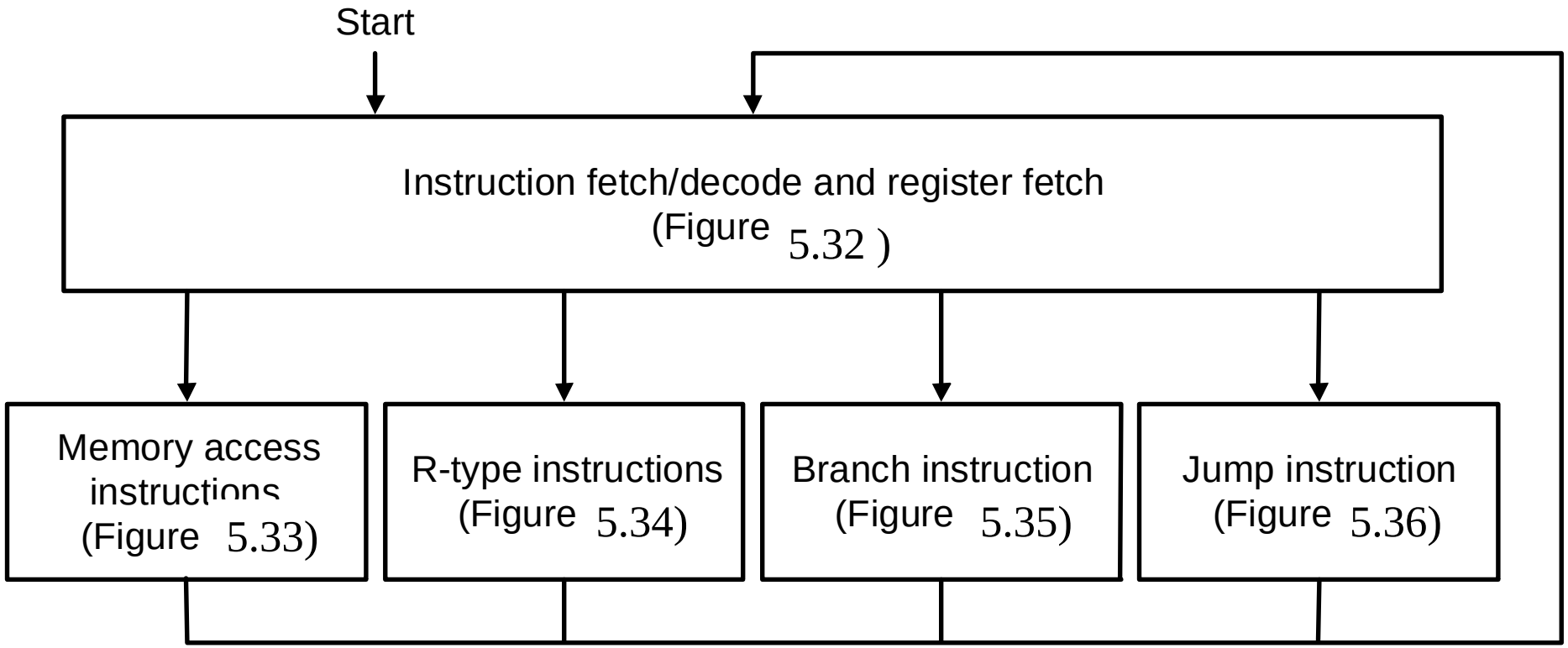


Finite state machines

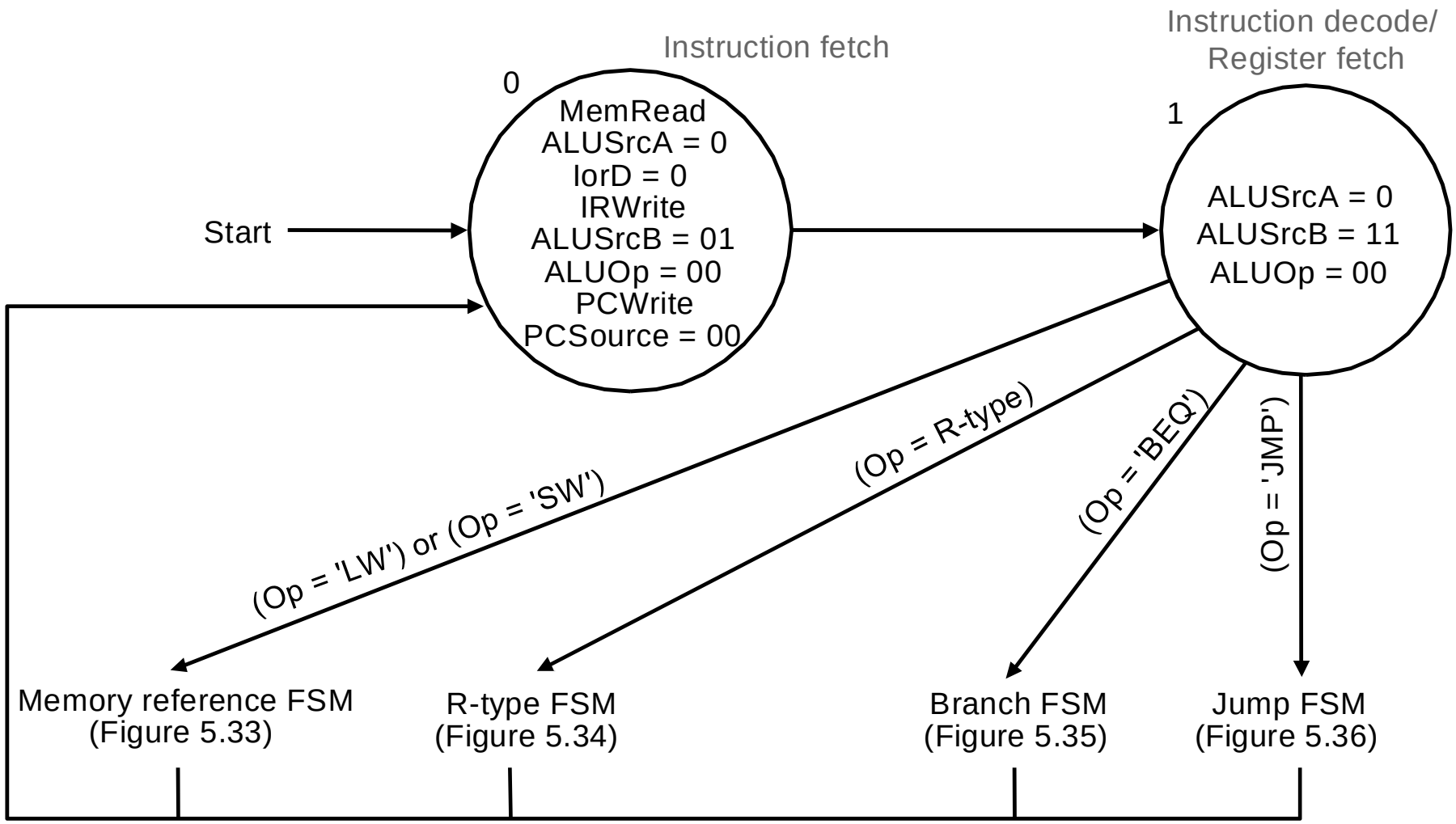
- Finite state machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)
  - We'll use a Moore machine (output based only on current state)



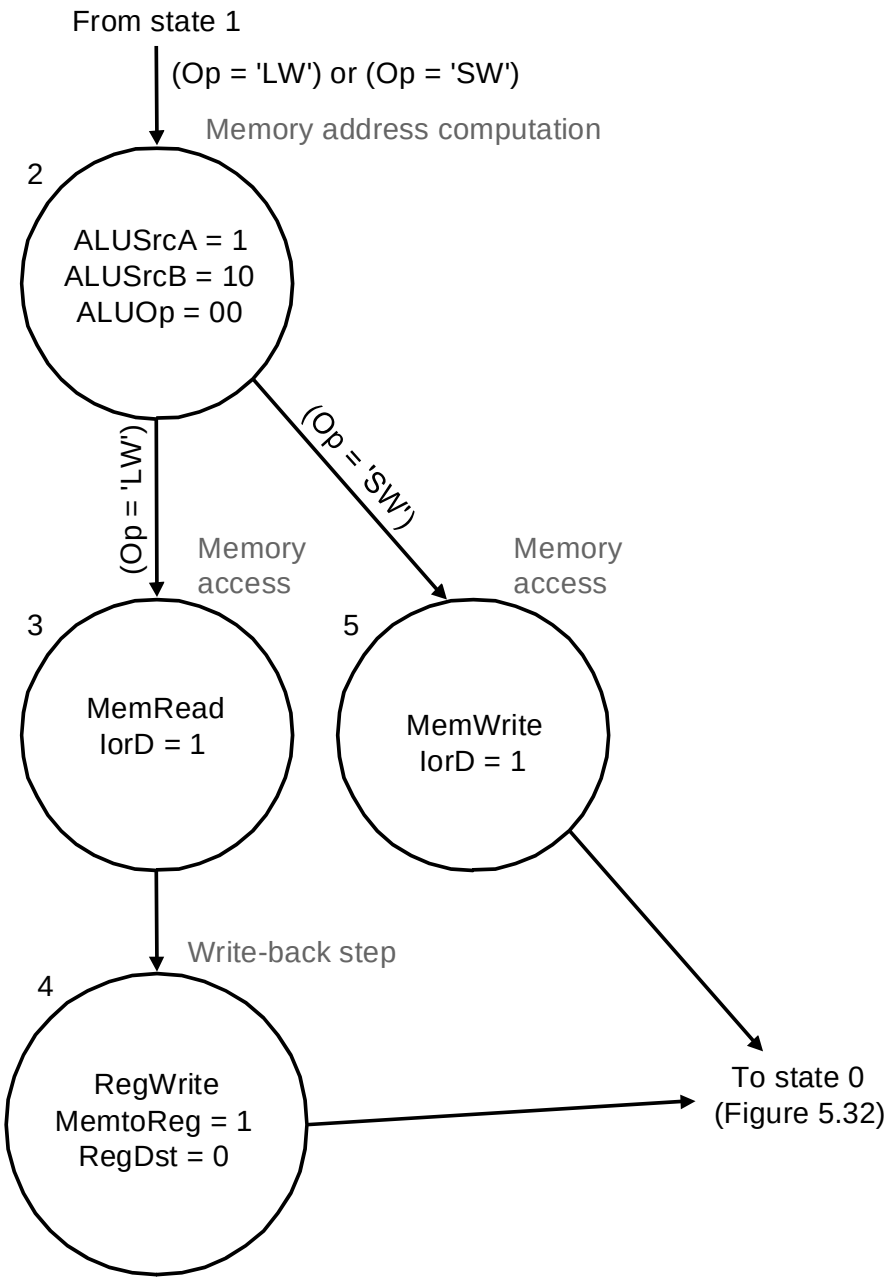
Finite State Machine Control for Multicycle Implementation



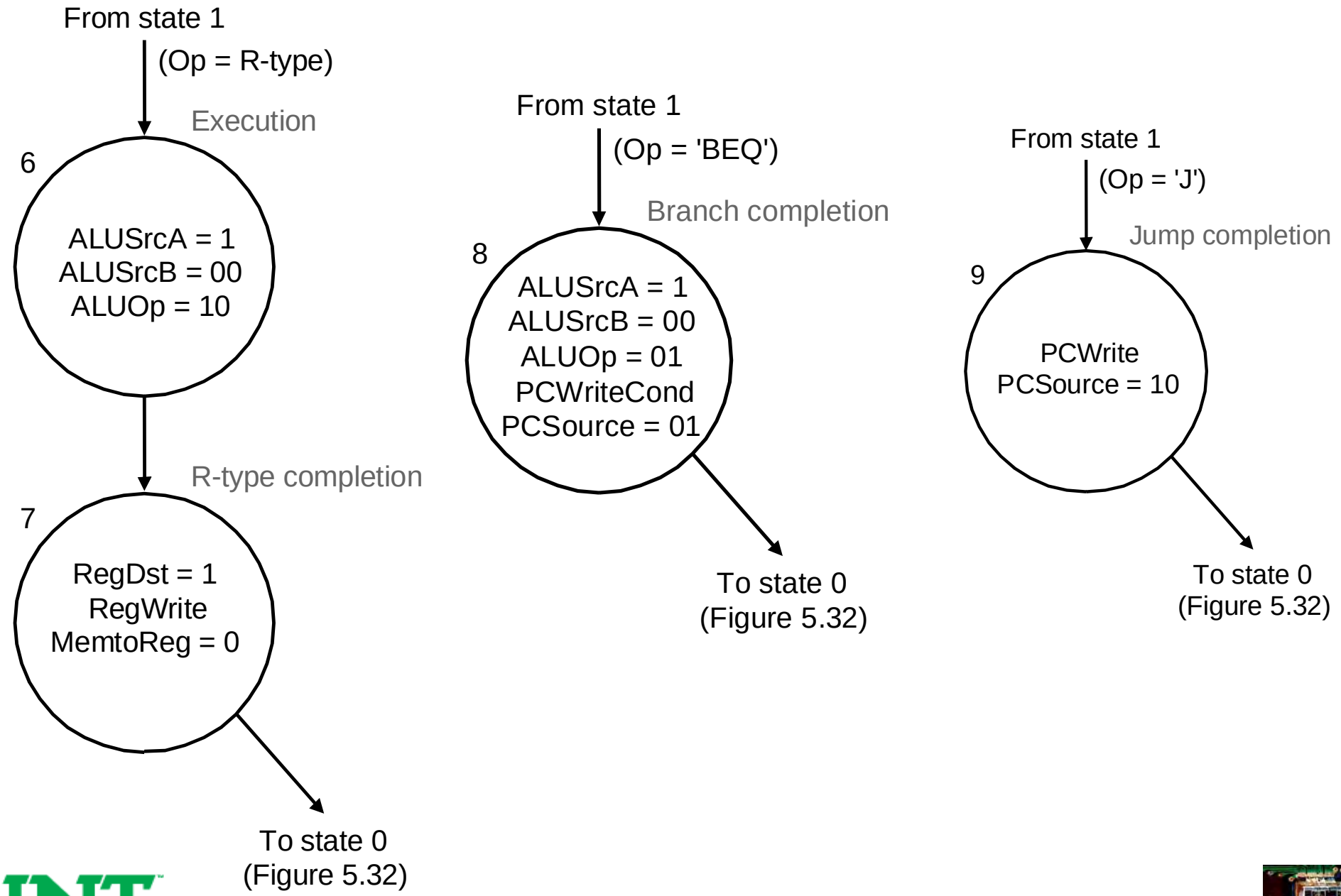
Instruction Fetch & Decode (Fig 5.32)

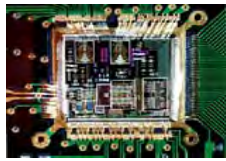


Memory Reference Instructions (Fig. 5.33)

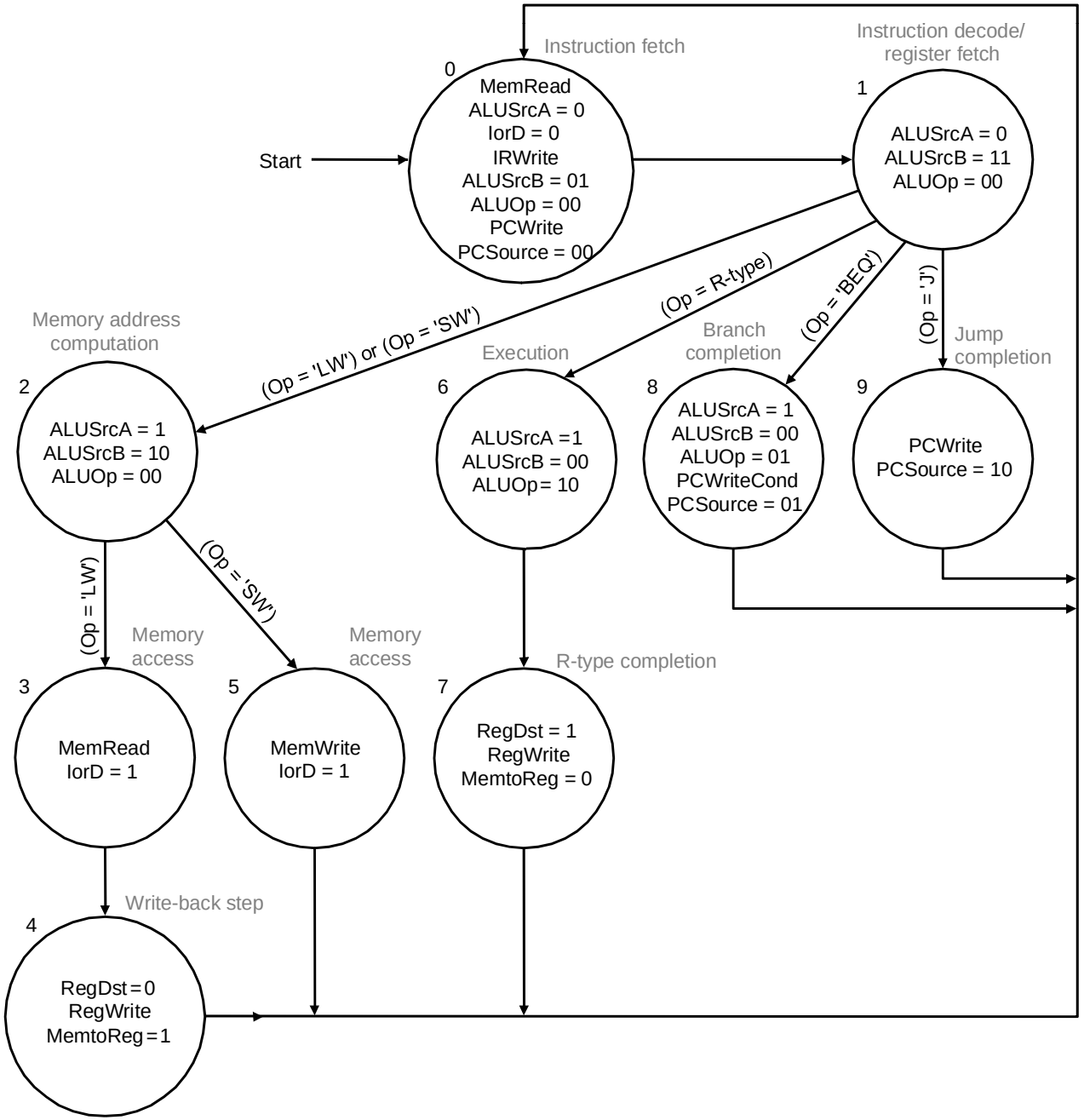


R-type, Branch, Jump..

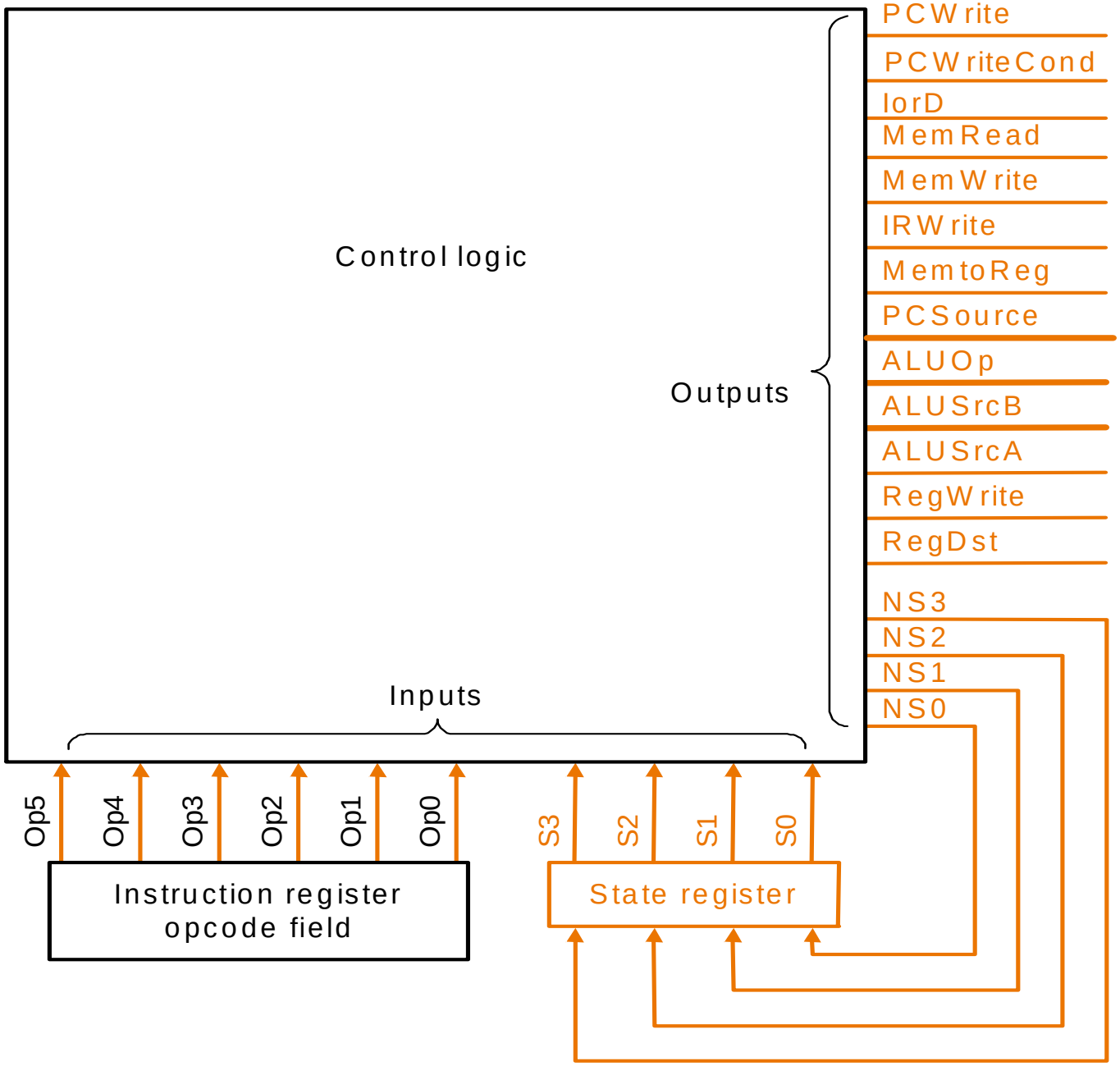




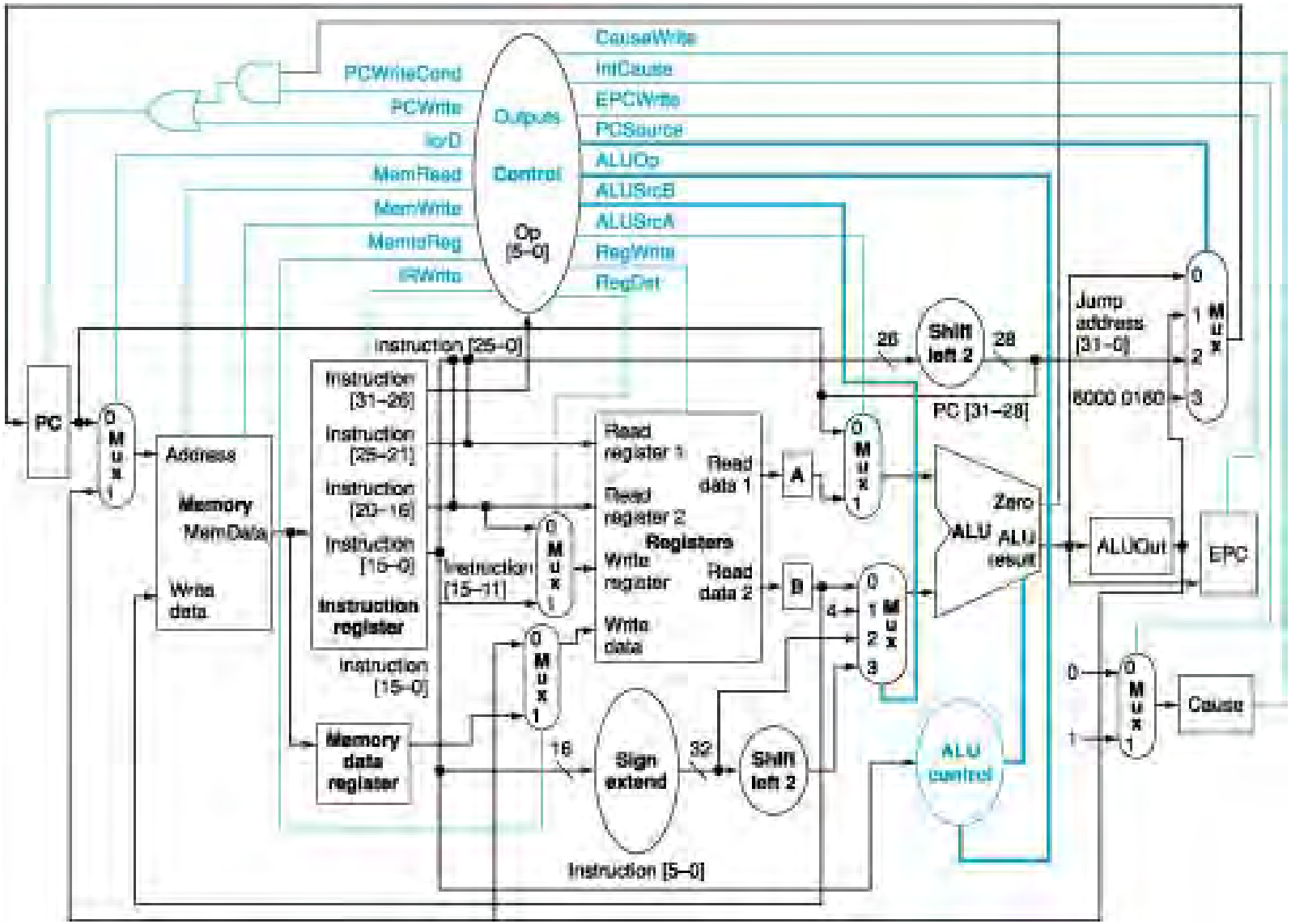
Graphical Specification of FSM



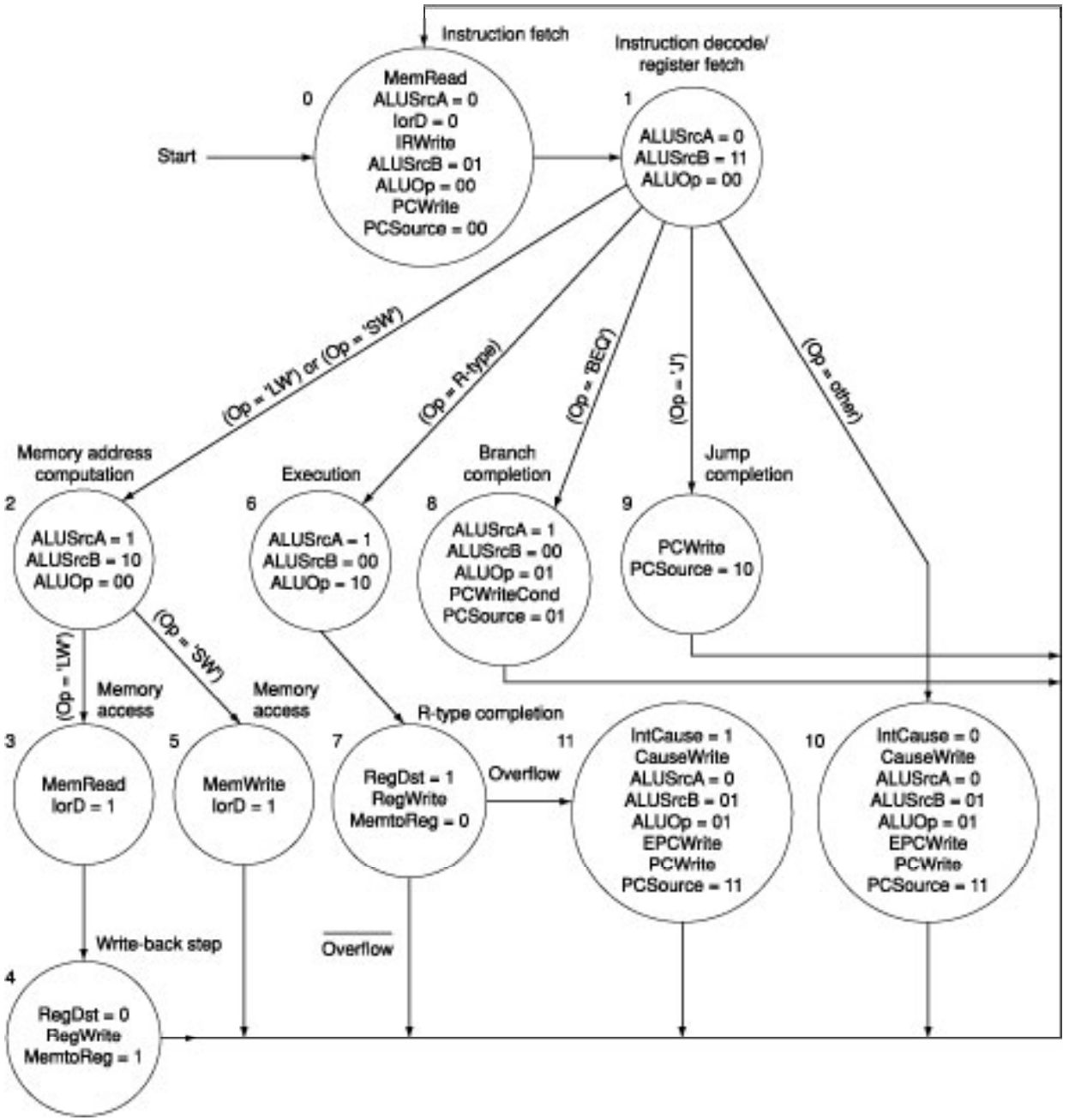
Finite State Machine for Control



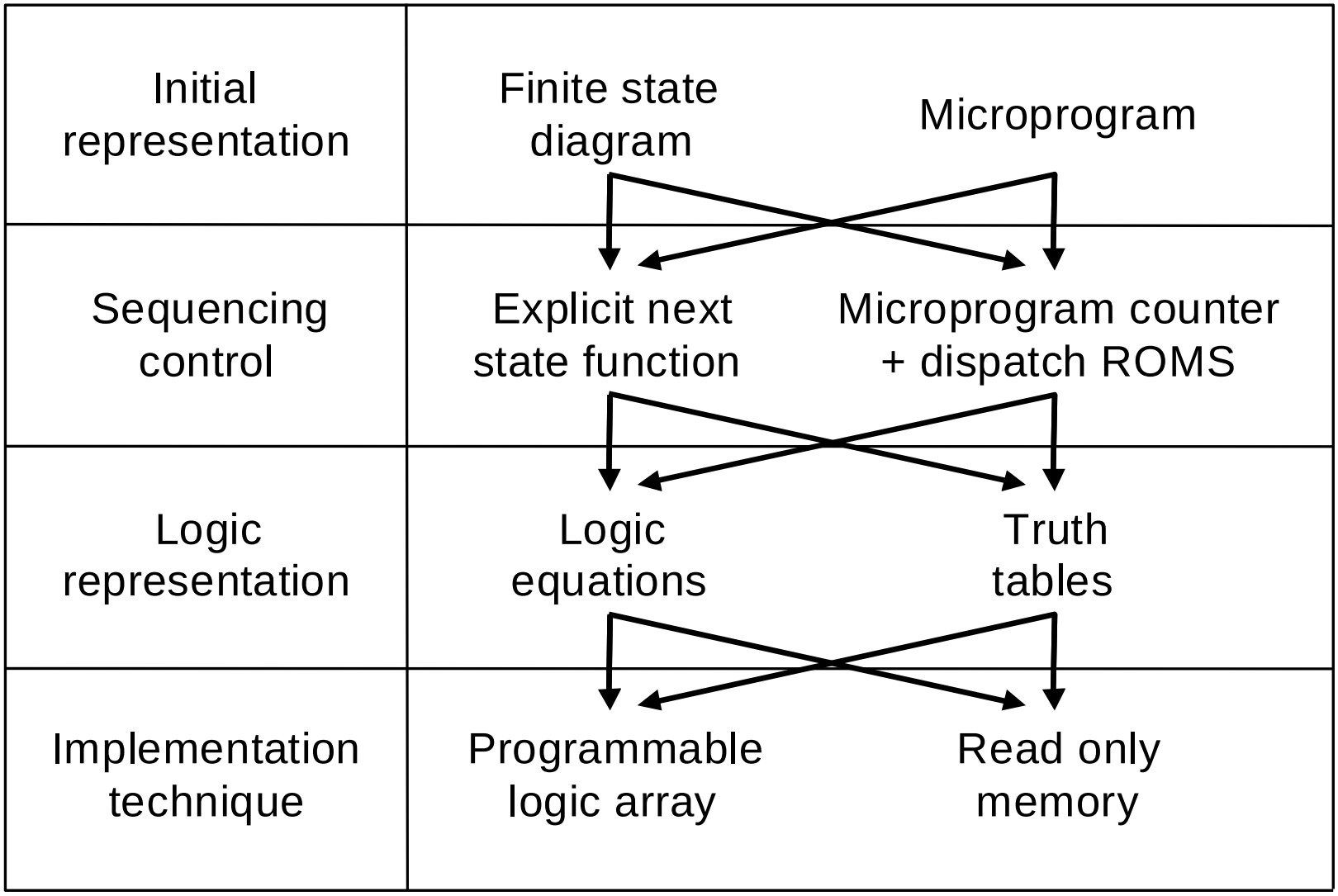
Multicycle Datapath to Handle Exceptions

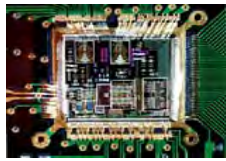


Multicycle Control to Handle Exceptions



Controller Implementation: Big Picture





Summary

- Single-cycle implementation
- Multi-cycle implementation
  - Is an effective implementation: slower instructions take more clock cycles, faster, less!
  - Higher resource sharing (area is less)
- State diagrams can be used to specify the control
- From FSM spec, we can automatically synthesize the controller implementation.
- Controller Implementation
  - Three choices: ROM, PLA, and Microprogramming
  - PLAs is more efficient in terms of area compared to ROM
  - Microprogramming is a *flexible* style (popularized by CISC)