

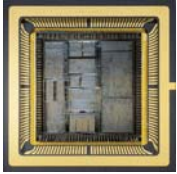
Lecture 2: General Purpose Processor Design

CSCE 6651

Advanced VLSI Systems

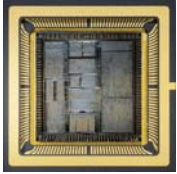
Instructor: Saraju P. Mohanty, Ph. D.

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

- General Purpose Processor
- Program Execution
- Construction of a Simple MIPS Processor
- Single Cycle Processor
- Multicycle Processor
- Pipelined Processor



Levels of Representation

High Level Language
Program

Compiler

Assembly Language
Program

Assembler

Machine Language
Program

Machine Interpretation



temp = v[k];

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

v[k+1] = temp;

lw\$15, 0(\$2)

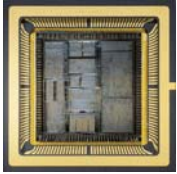
lw\$16, 4(\$2)

sw \$16, 0(\$2)

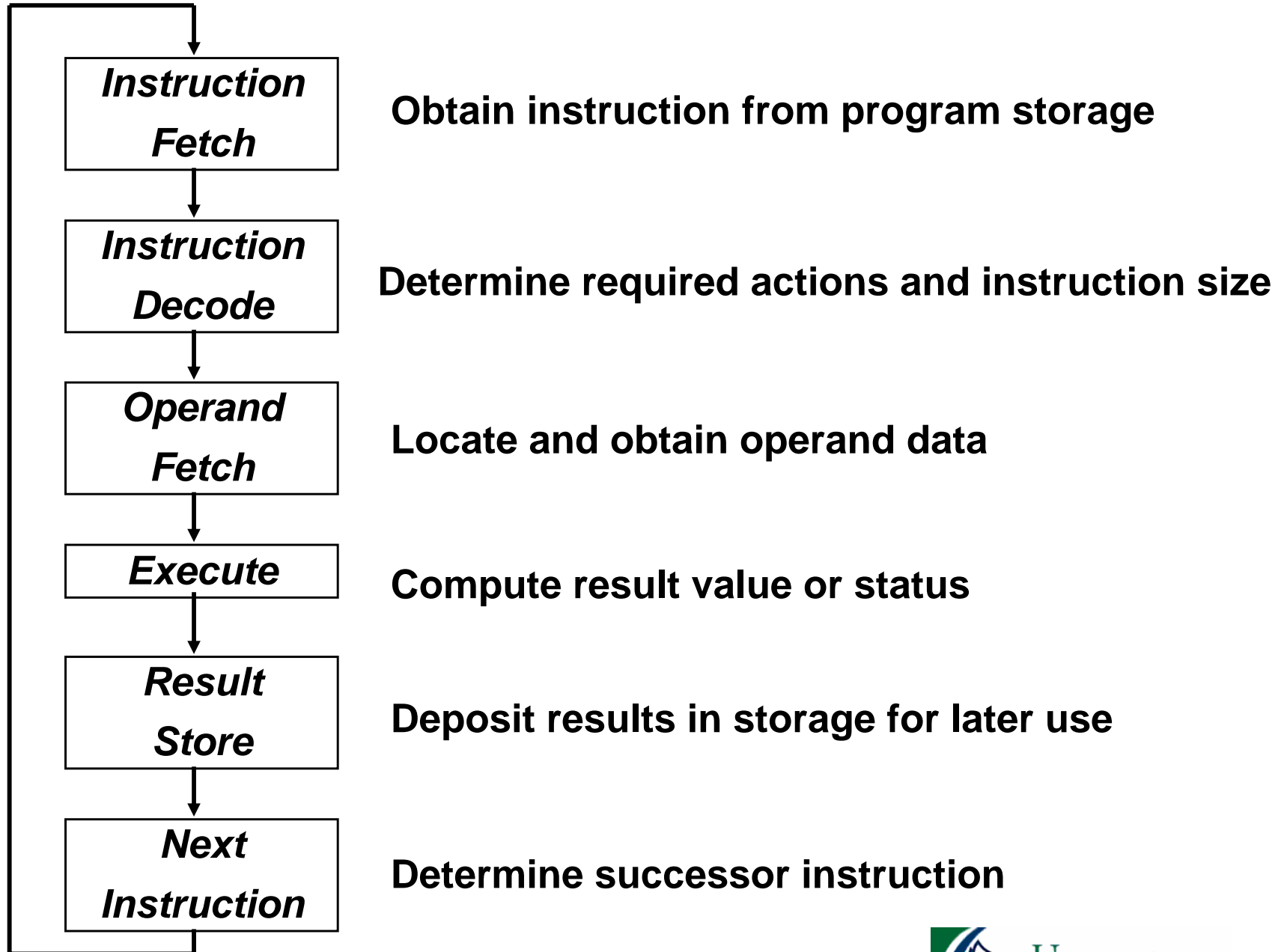
sw \$15, 4(\$2)

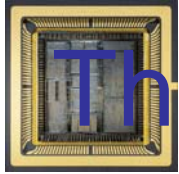
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

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



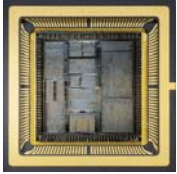
Execution Cycle





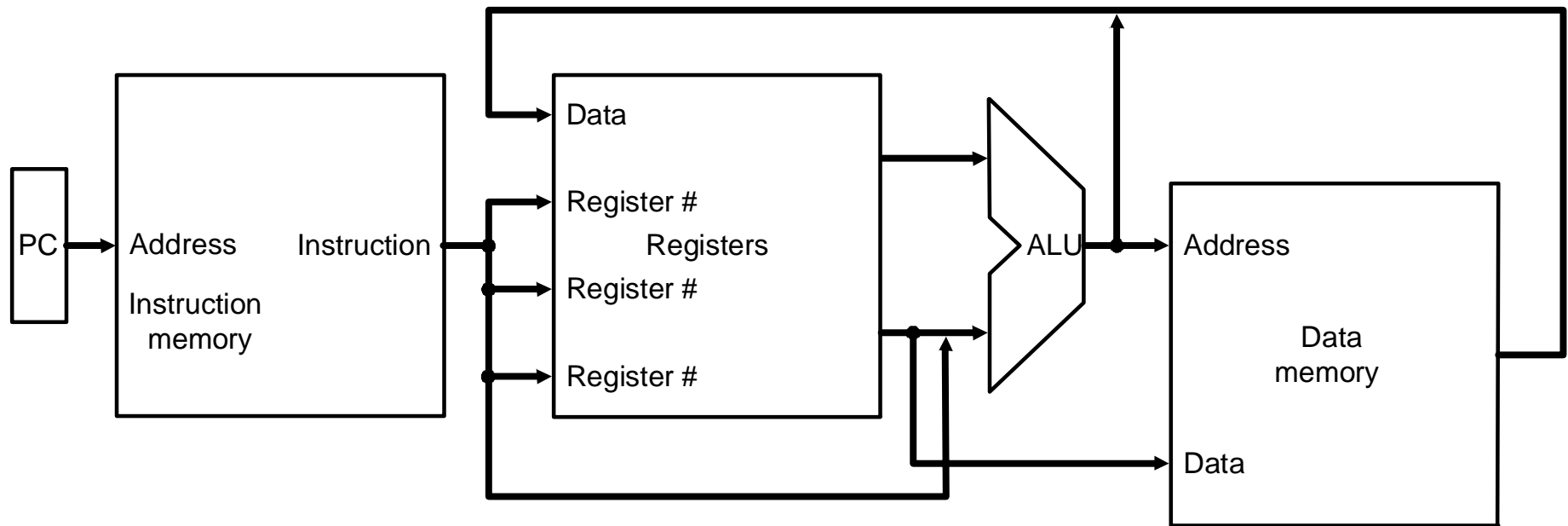
The Processor : Datapath and Control

- Consider implementation of the MIPS architecture
- 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



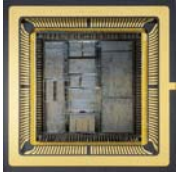
More Implementation Details

Abstract / Simplified View:



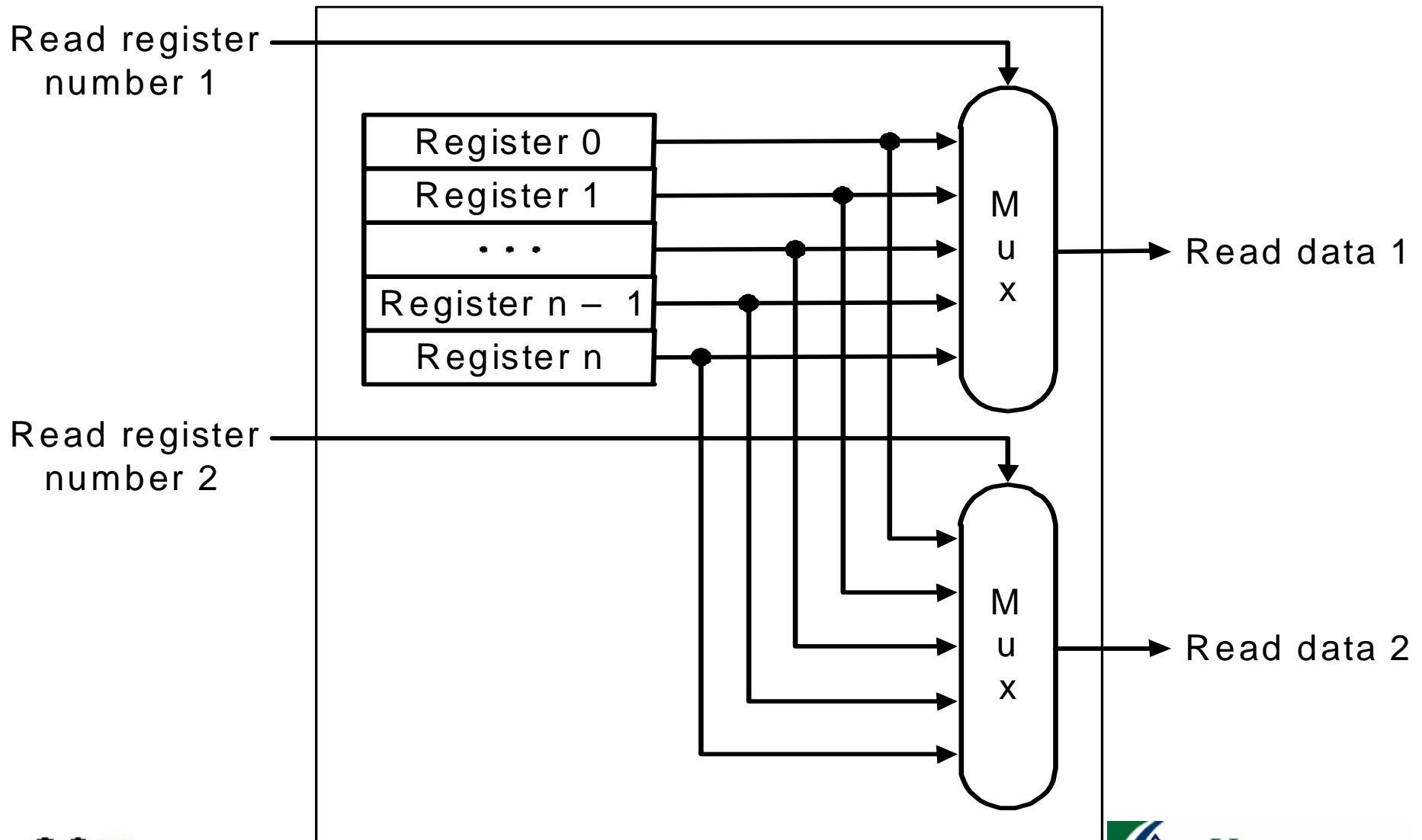
Two types of functional units:

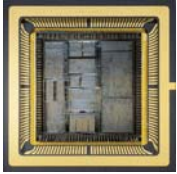
- elements that operate on data values (combinational)
- elements that contain state (sequential)



Register File: Read Operation

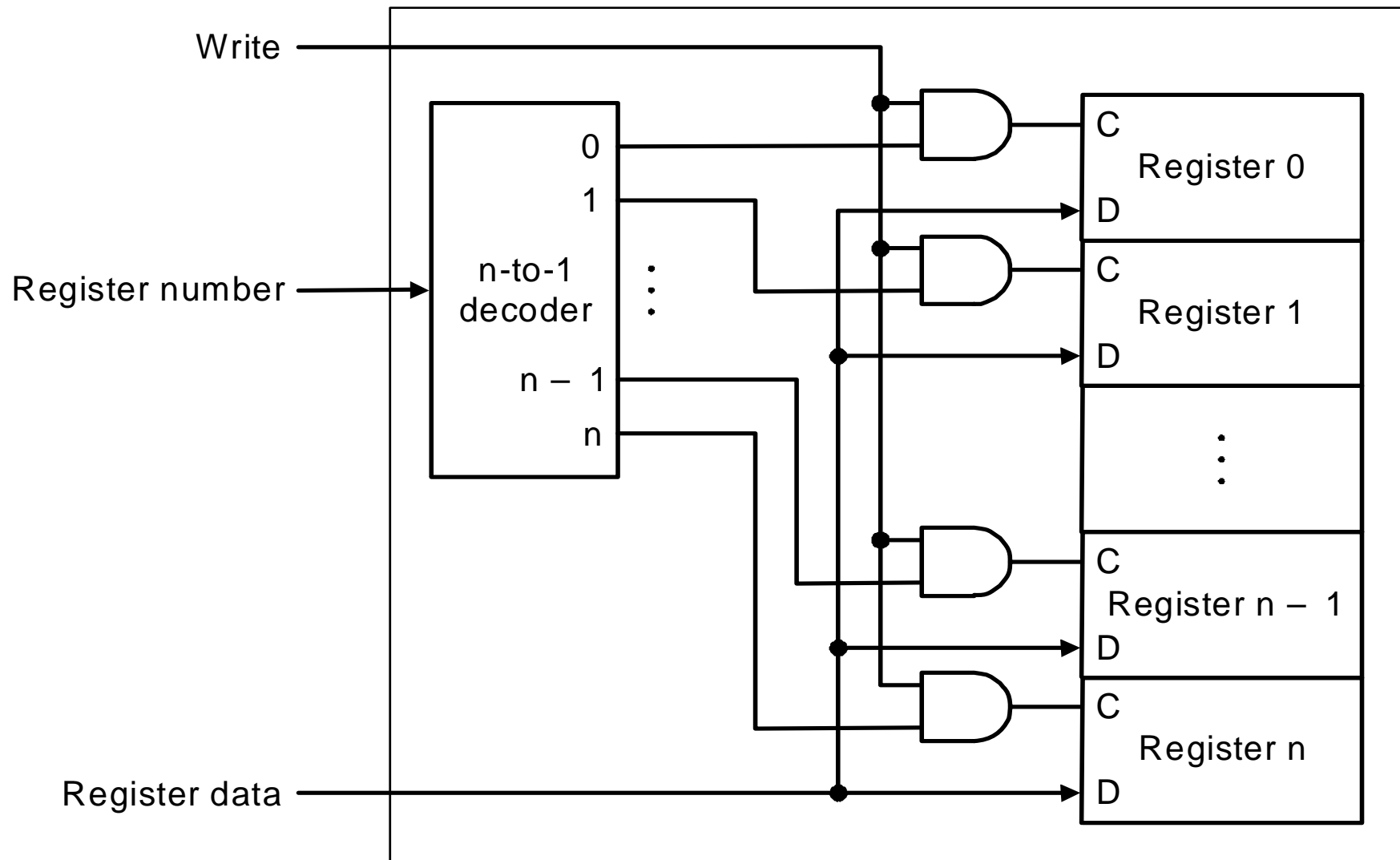
Built using D flip-flops (Combinational in nature)

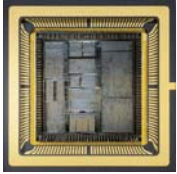




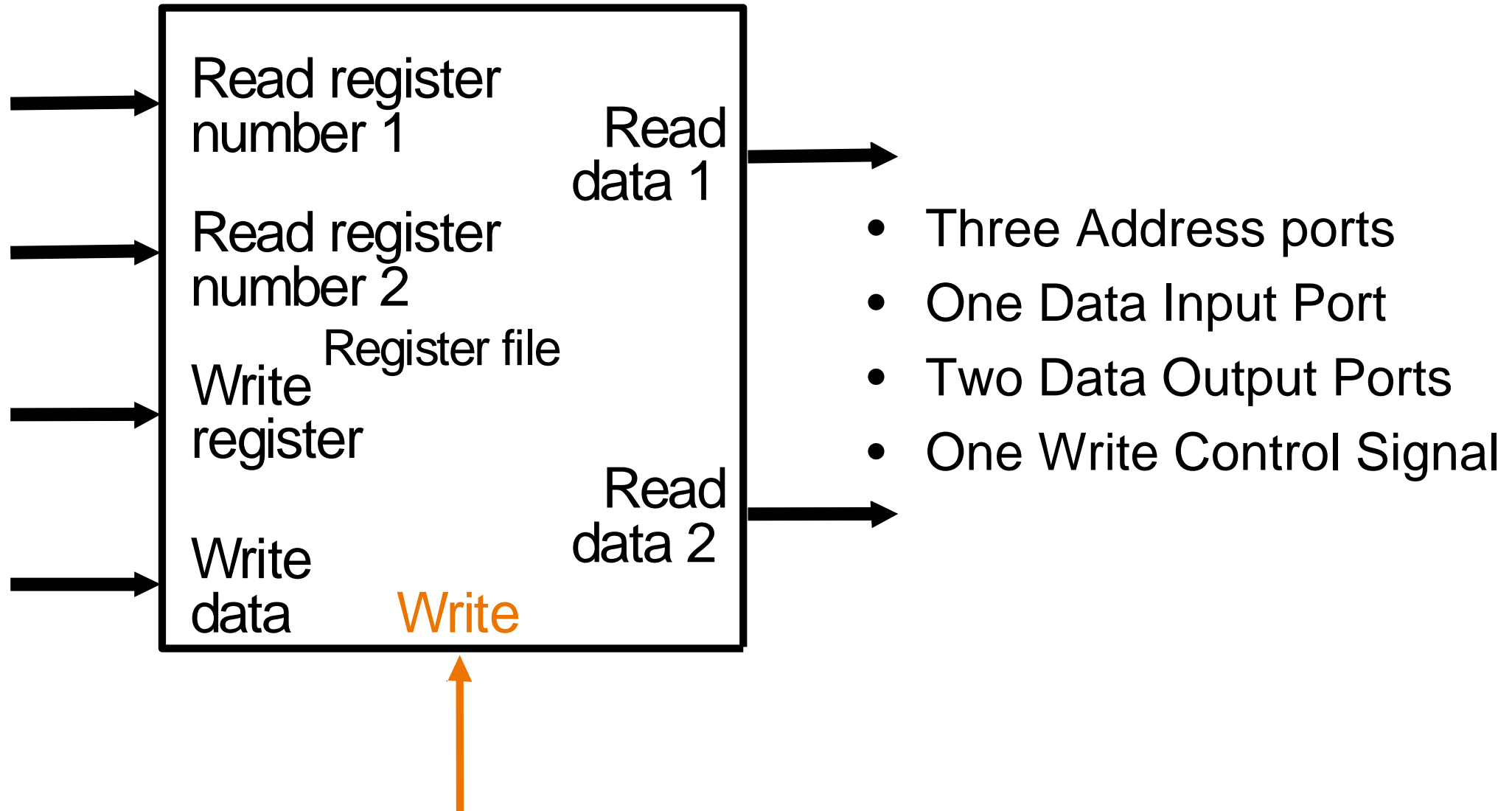
Register File: Write Operation

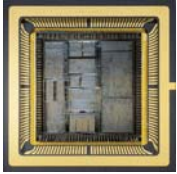
We still use the real clock to determine when to write





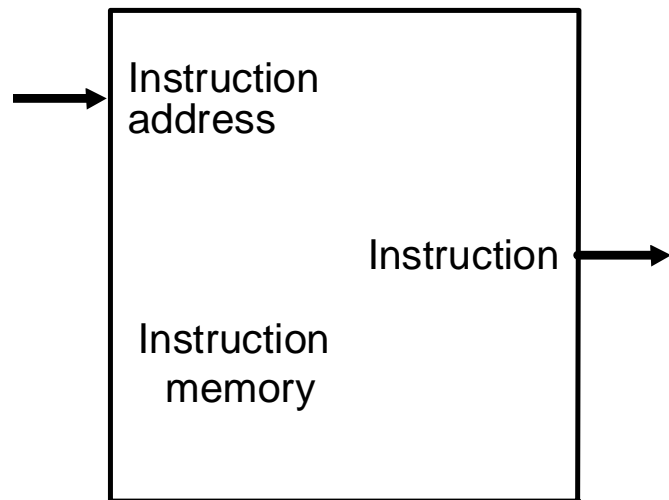
Register File: Block Diagram



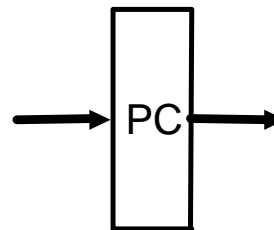


Functional Units - I

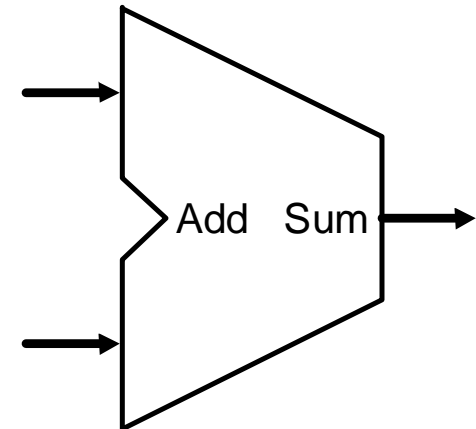
- a: *Instruction Memory* -- After an instruction address is put, the instruction residing at the address appears at the output port
- b: *Program Counter* -- A simple up counter
- c: *Adder* -- A 2's complement adder



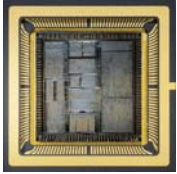
a. Instruction memory



b. Program counter

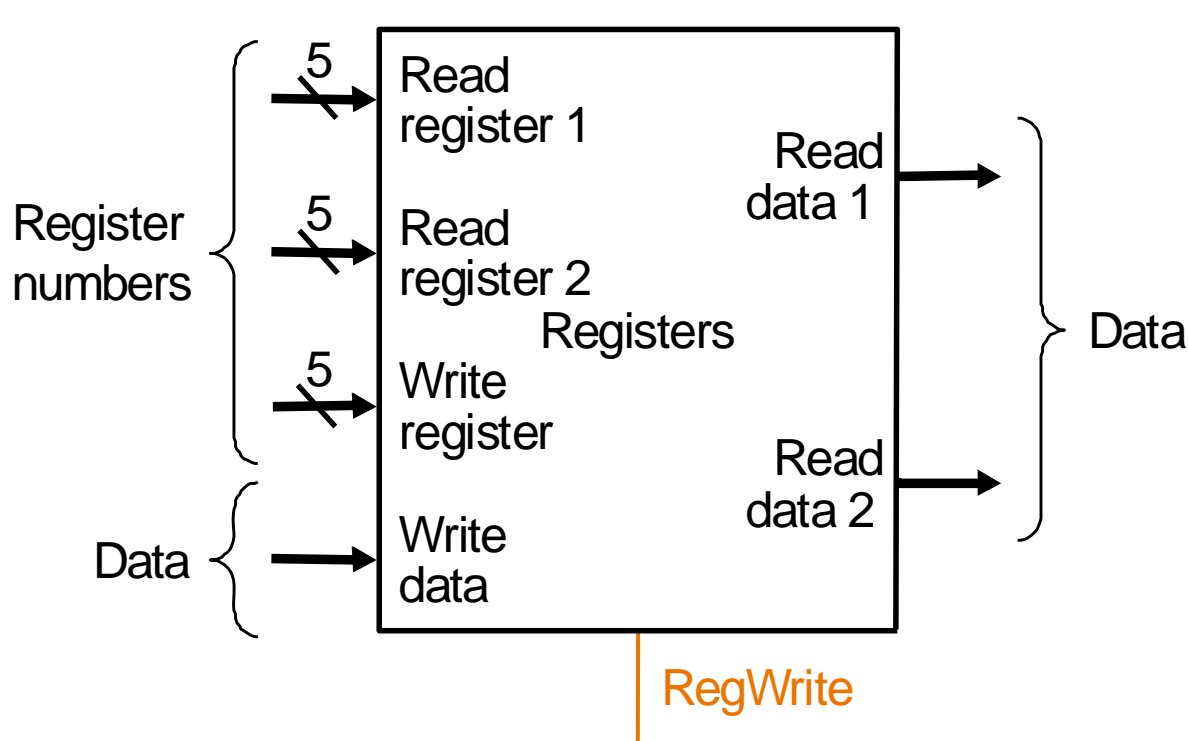


c. Adder

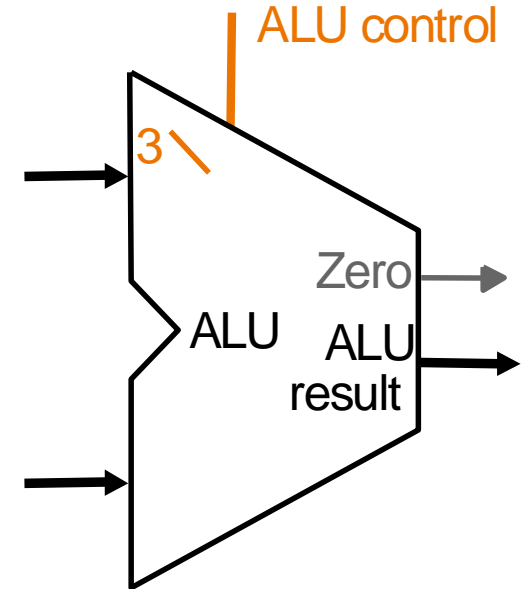


Functional Units - II

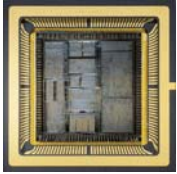
- a: *Register File* -- It's construction, read, and write operations as discussed previously
- b: *ALU (Arithmetic & Logic Unit)* -- Recall the ALU design we have discussed in last two classes and Note the “Zero” output



a. Register File



b. ALU

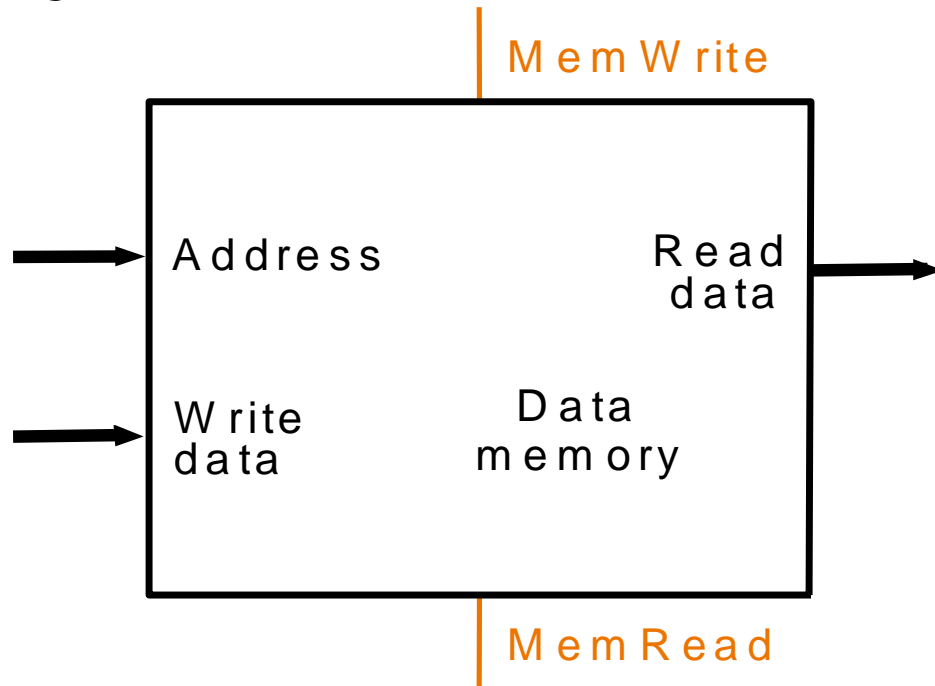


Functional Units -III

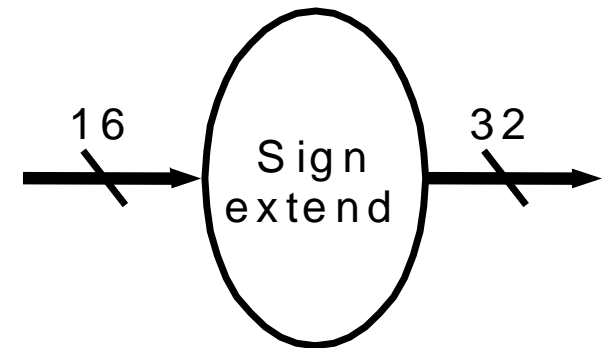
a: *Data Memory Unit*

Similar to Instruction Memory Unit, only that it can be 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

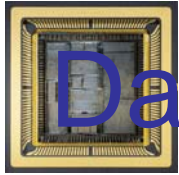
b: *Sign Extension Unit* -- Extends 16-bit input operand to 32 bits



a. Data memory unit

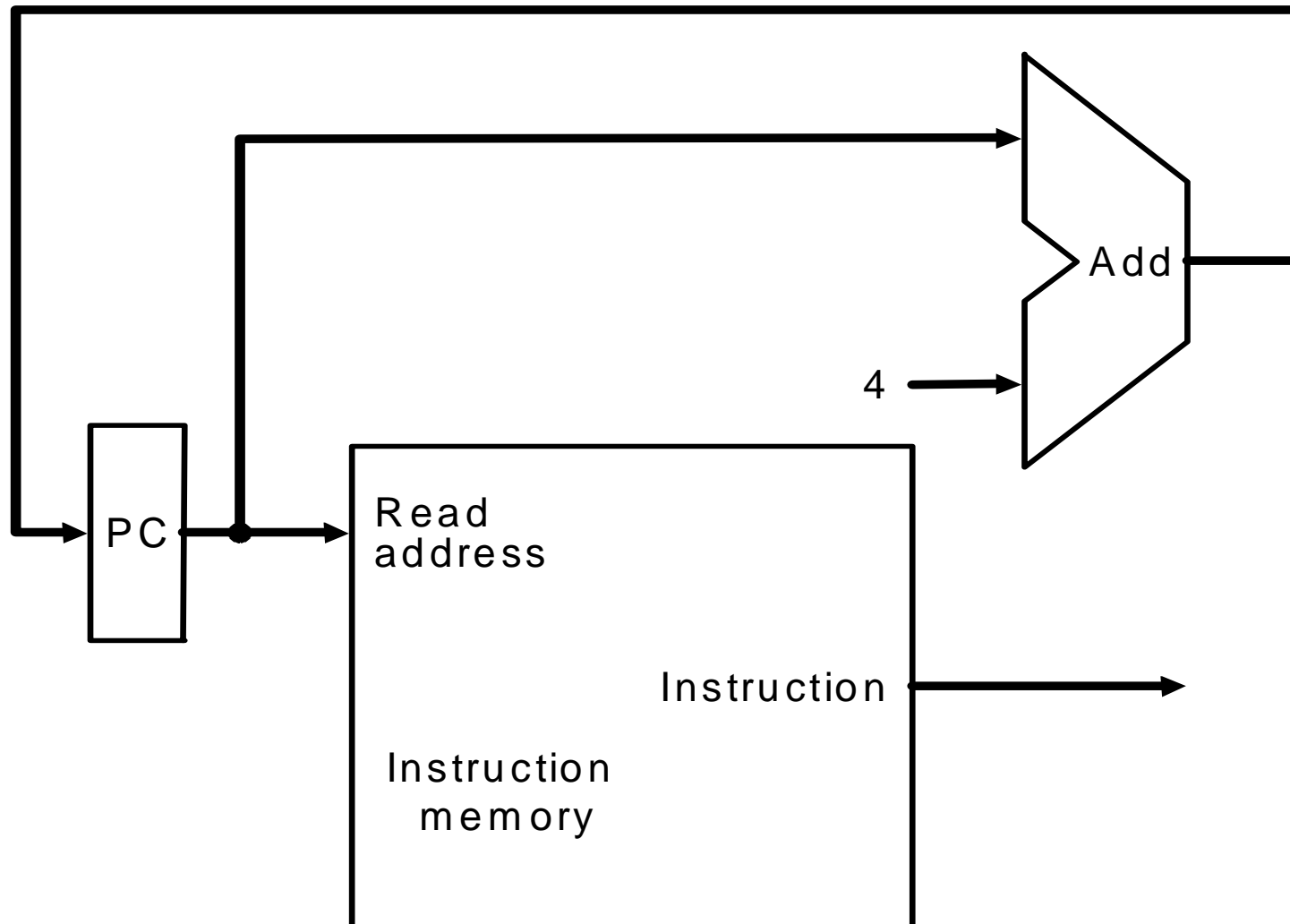


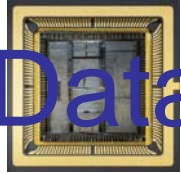
b. Sign-extension unit



Datapath for Instruction Fetch (Piece I)

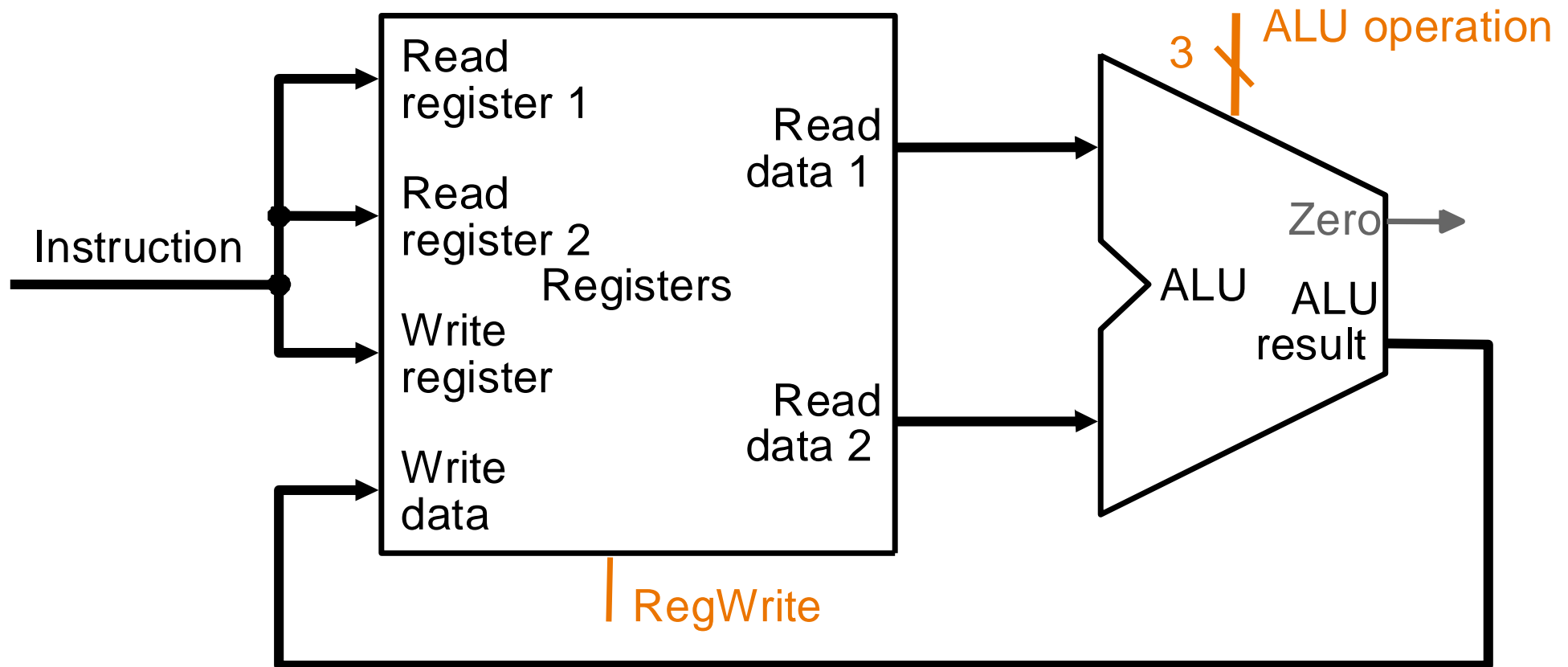
Fetching Instructions and Incrementing the program counter by 4

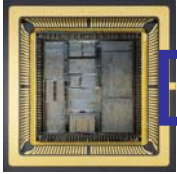




Datapath for R-type Instructions (Piece II)

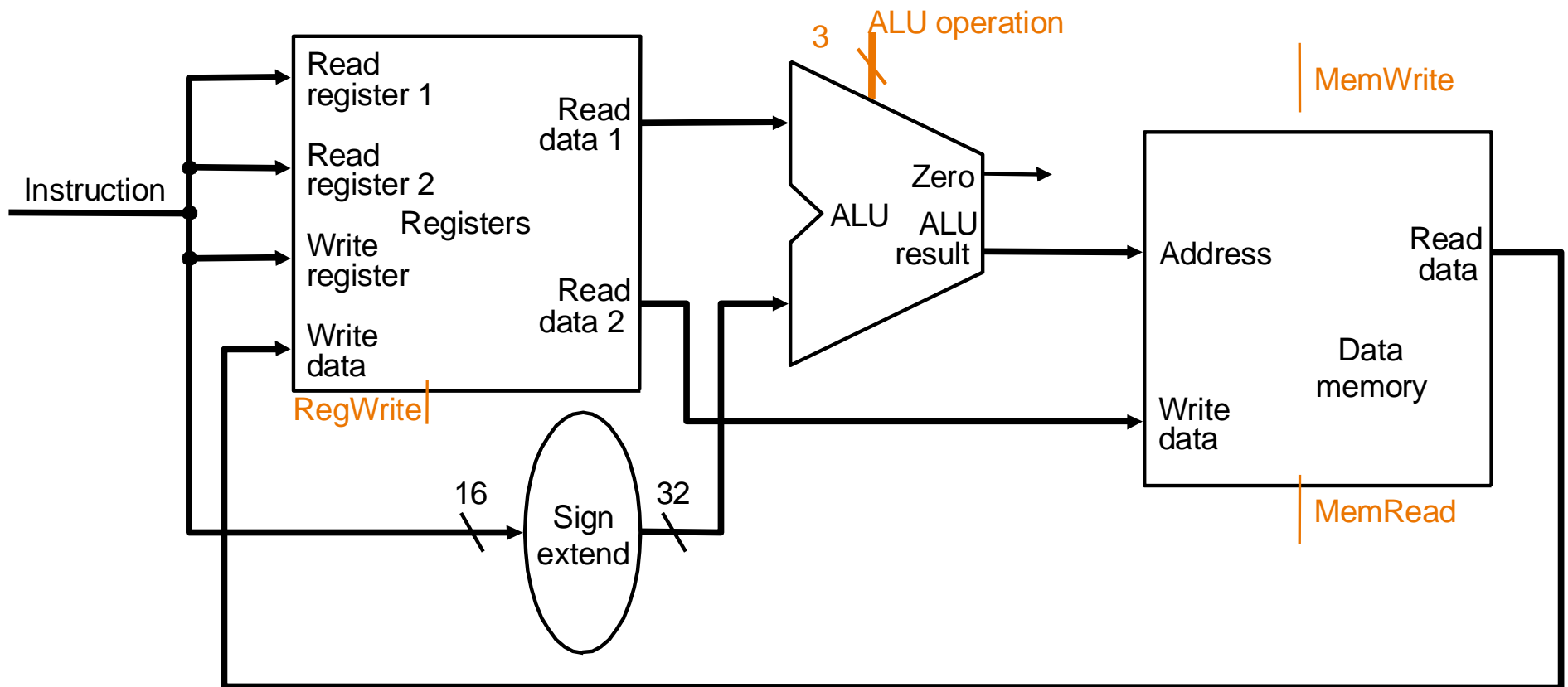
Datapath for R-type Instructions

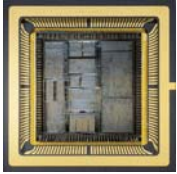




Datapath for Load/Store (Piece III)

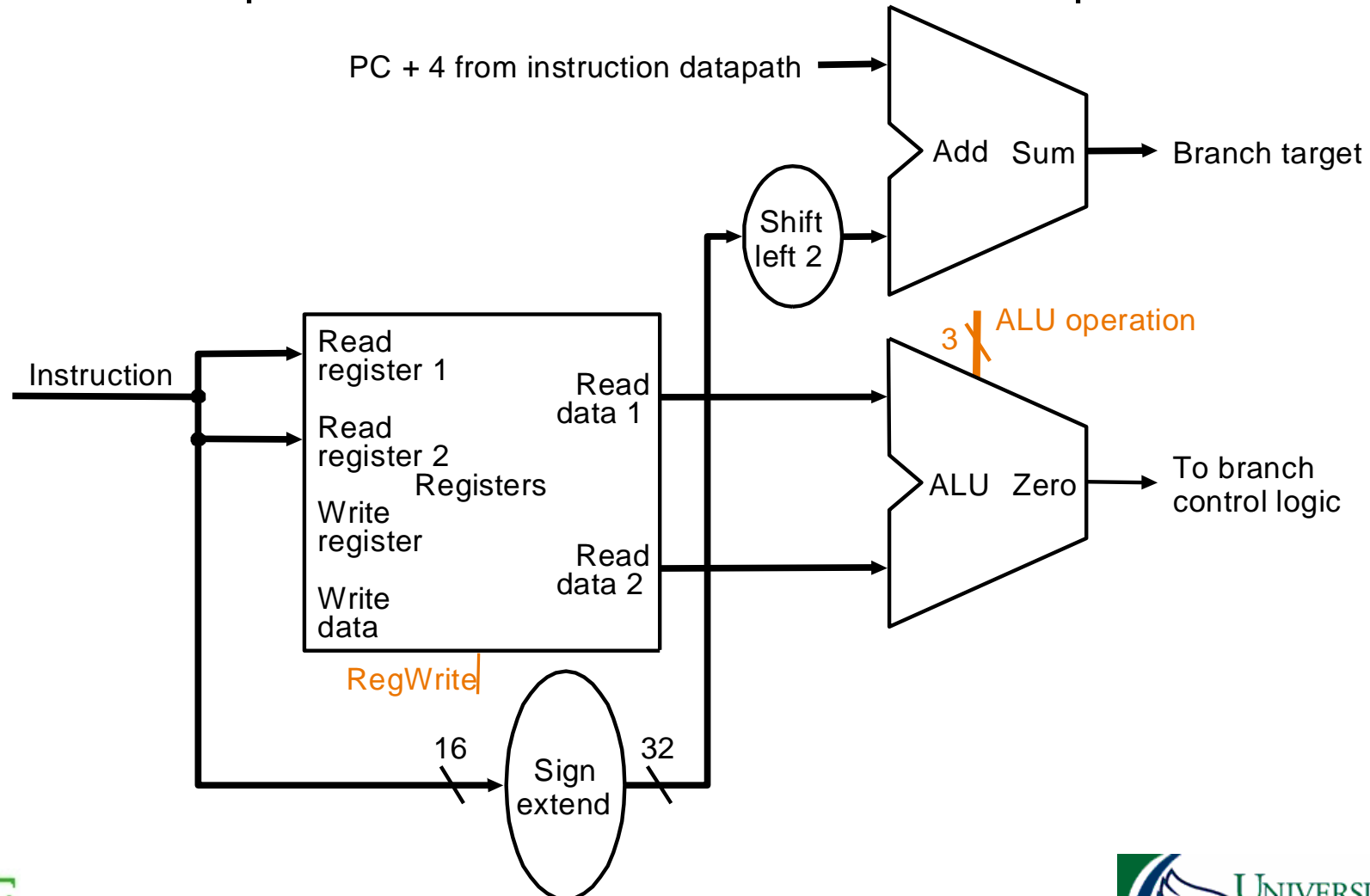
Datapath for load or store -- (1) Register Access; (2) Memory Address calculation; (3) Read/Write (4) Write into Register file (if the instruction is a *load*)

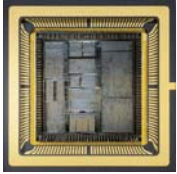




Datapath for Branch (Piece IV)

- 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

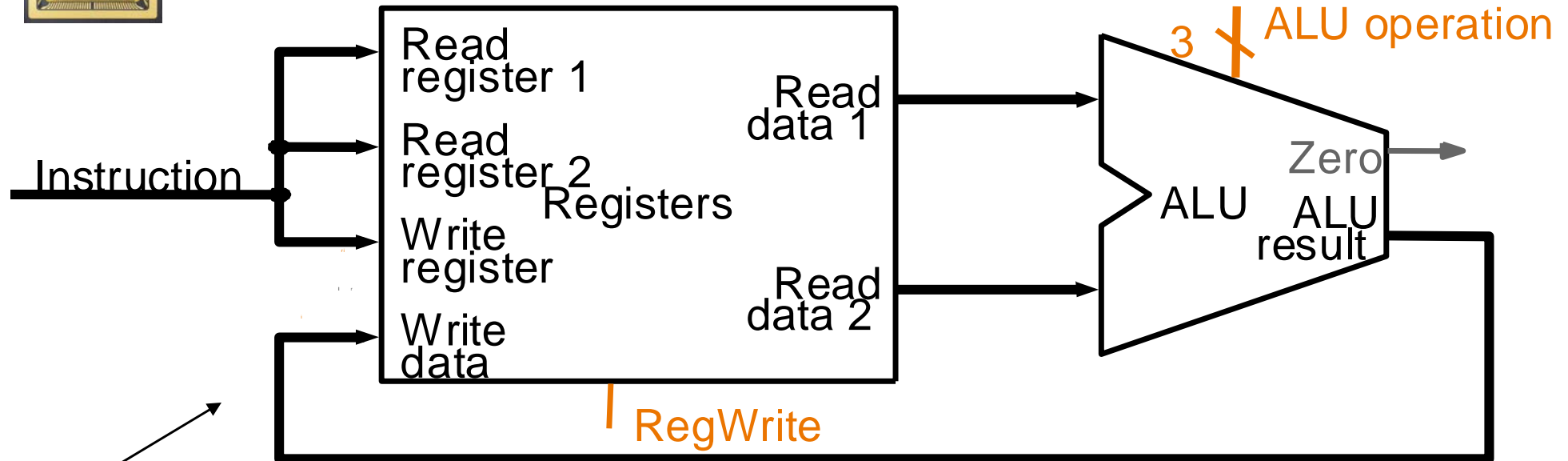
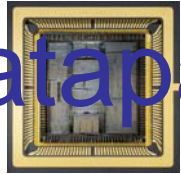




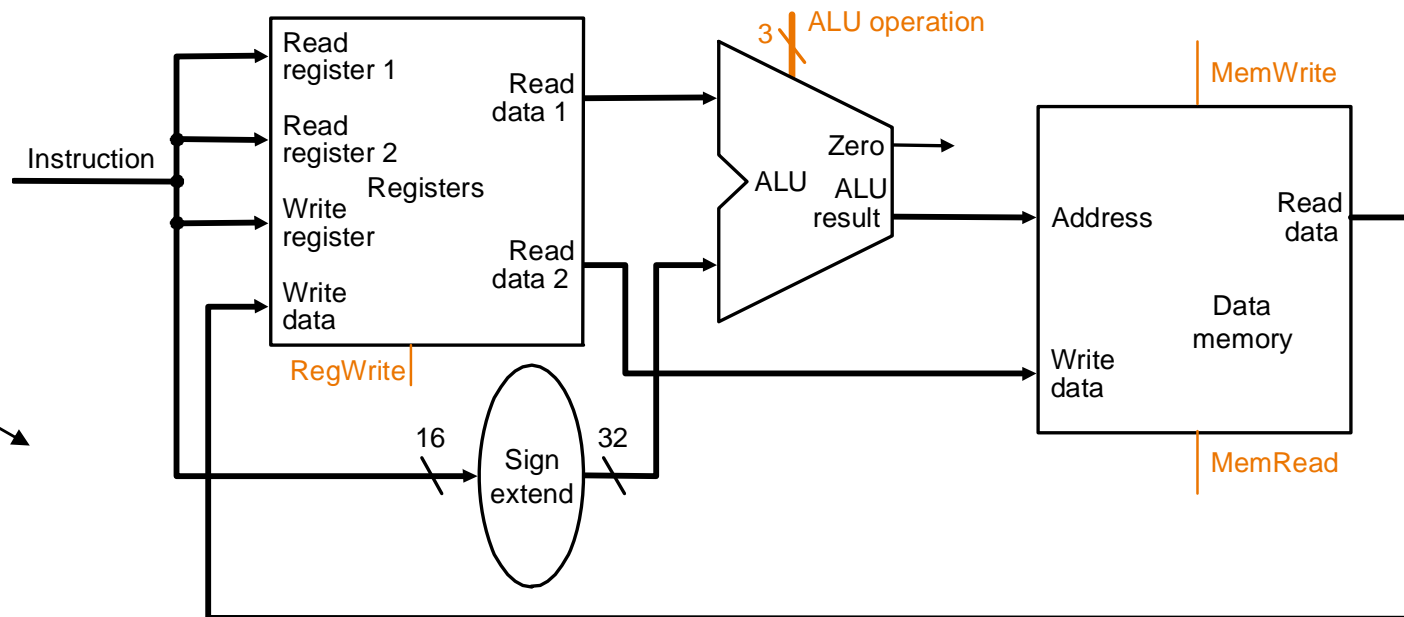
Datapath Construction Strategy

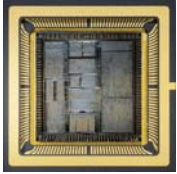
- Now, we have “pieces” of datapath that are capable of performing distinct functions
- We want to “stitch” them together to yield a final datapath that can execute all the instructions (lw, sw, add, sub, and, or, slt, beq, j)
- We will use multiplexors (or muxes for short) for stitching the datapath

Datapath Construction (Merge Pieces II & III)



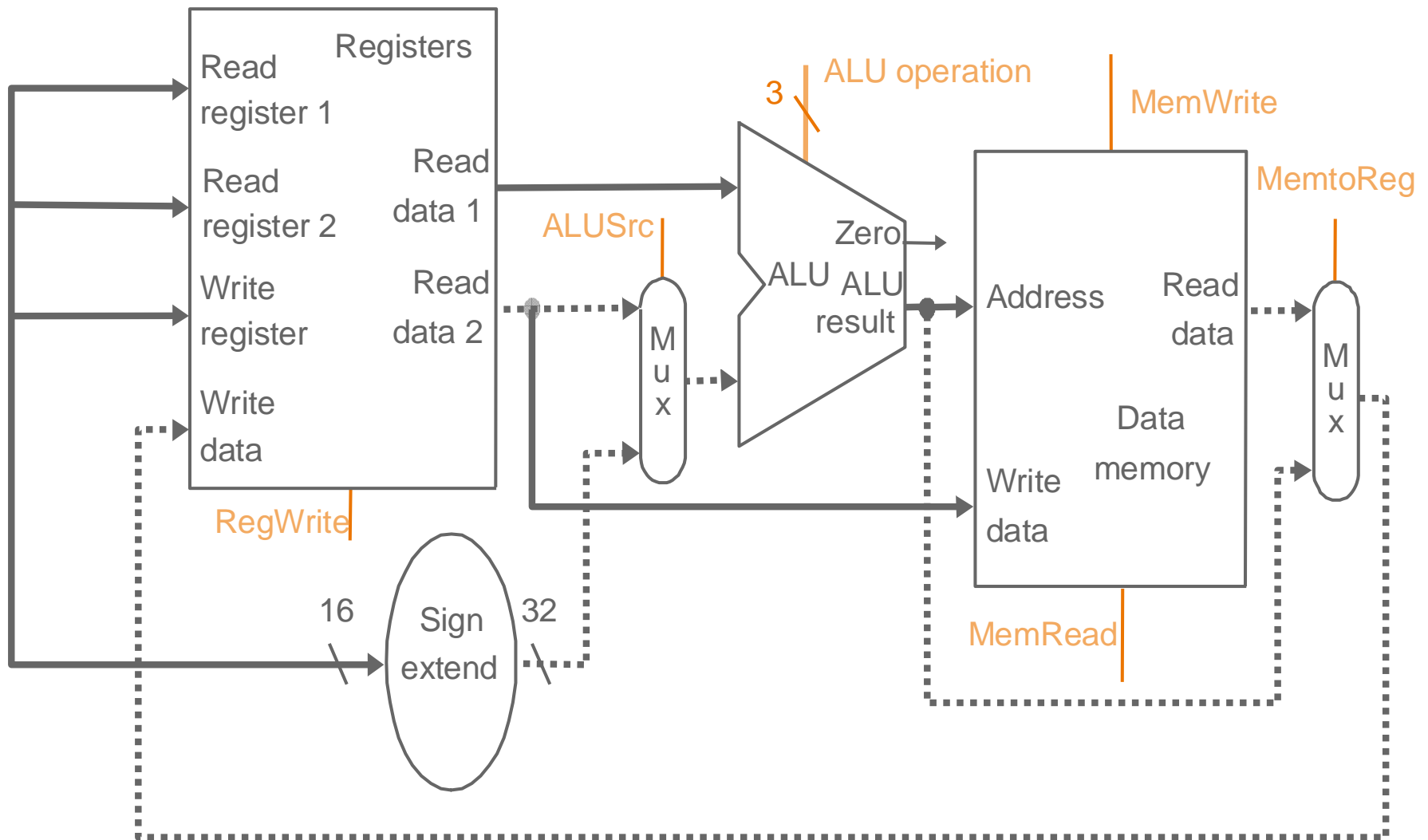
Piece II
+
Piece III

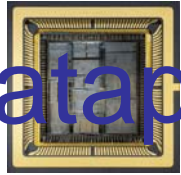




And you will get..

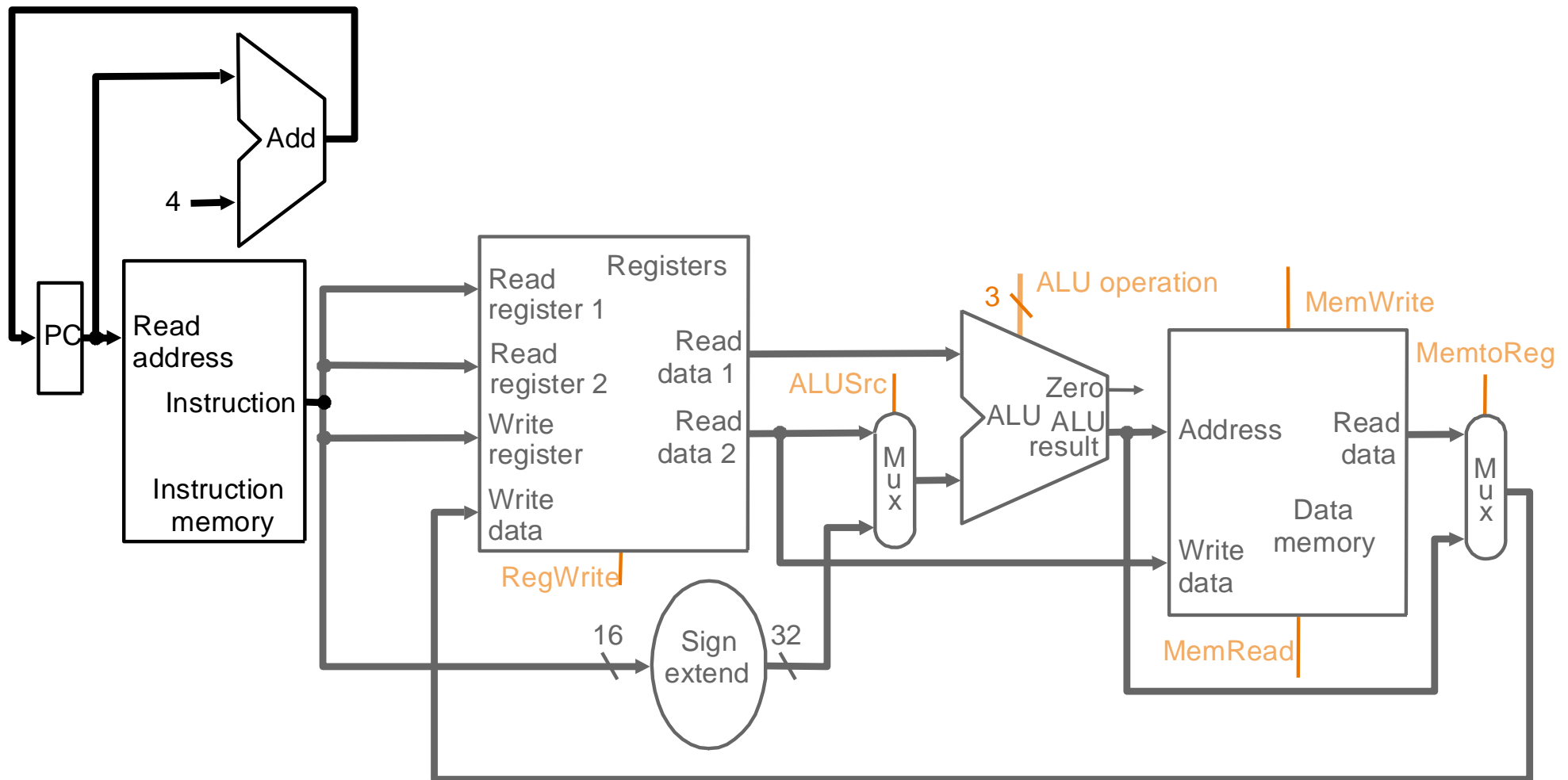
- Rule: Whenever we have more than one input feeding a functional unit, introduce a multiplexor (this gives rise to a control signal, more later..)



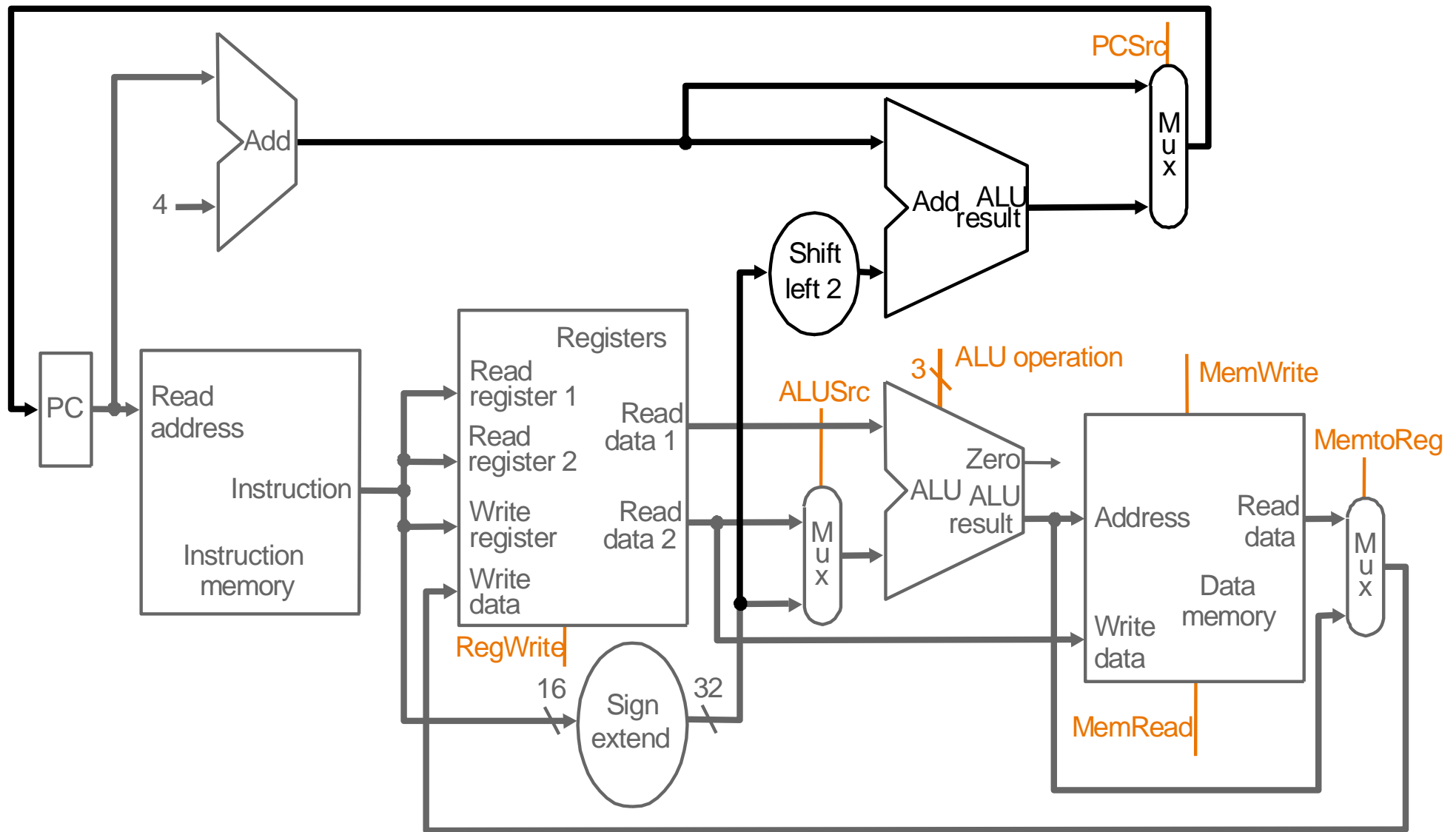


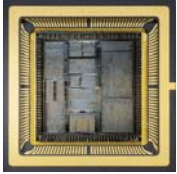
Datapath Construction ... (Now merge Piece I)

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



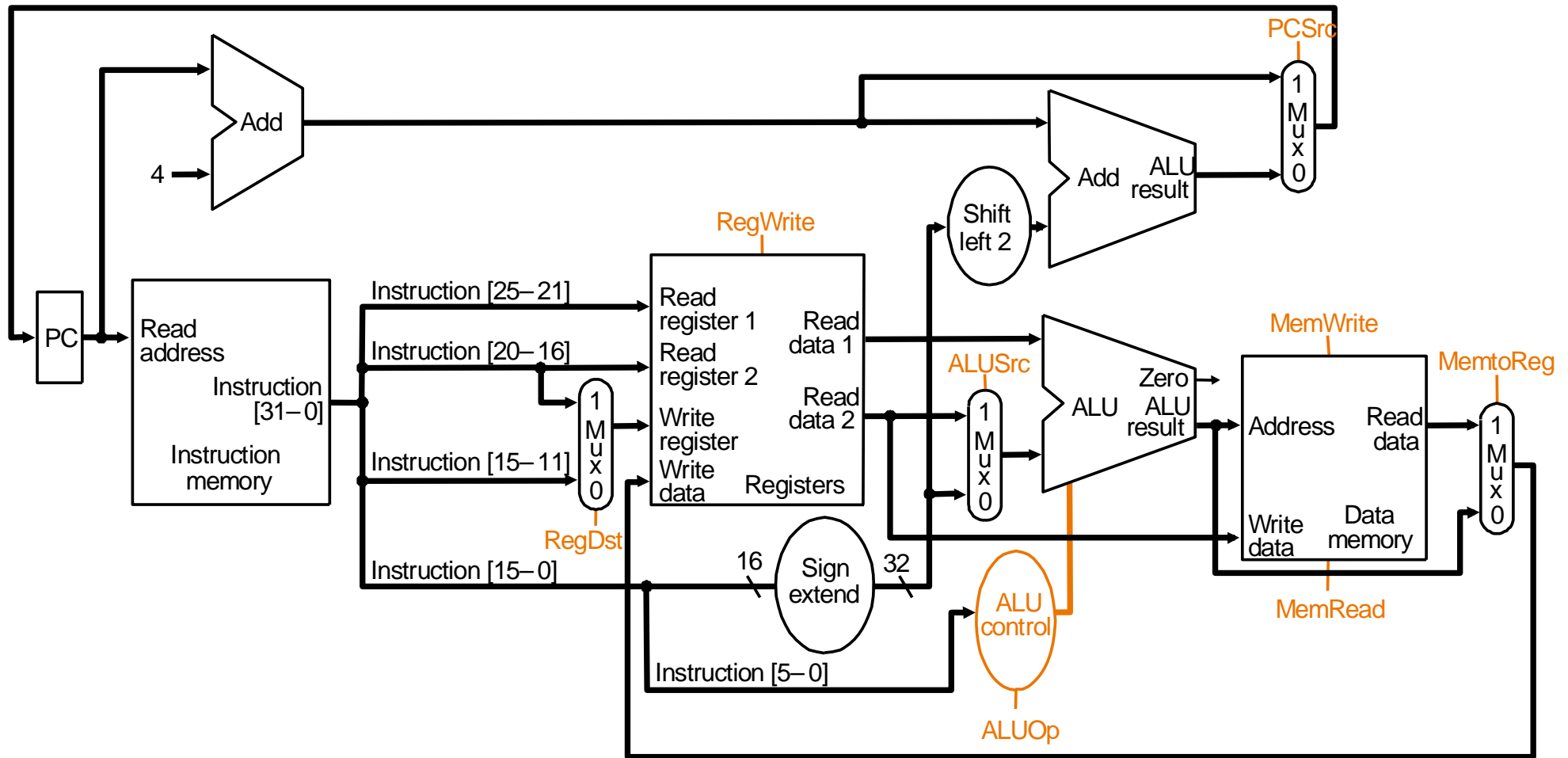
Datapath Construction (Finally, merge Piece IV)

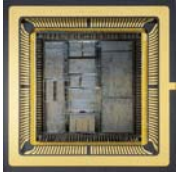




Final Datapath

- 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

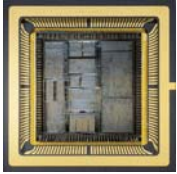
Instruction Format:

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

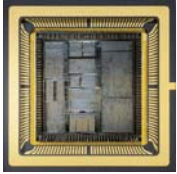
001 OR

010 add

110 subtract

111 set-on-less-than

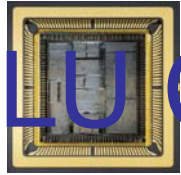
- Why is the code for subtract 110 and not 011?



ALU Control Design

- Must describe hardware to compute 3-bit ALU control input
 - Given instruction type
 - 00 = lw, sw
 - 01 = beq,
 - 10 = arithmetic
 - function code for arithmetic
- > **ALUOp**
computed from instruction type
- ALU Control inputs – How are they determined?

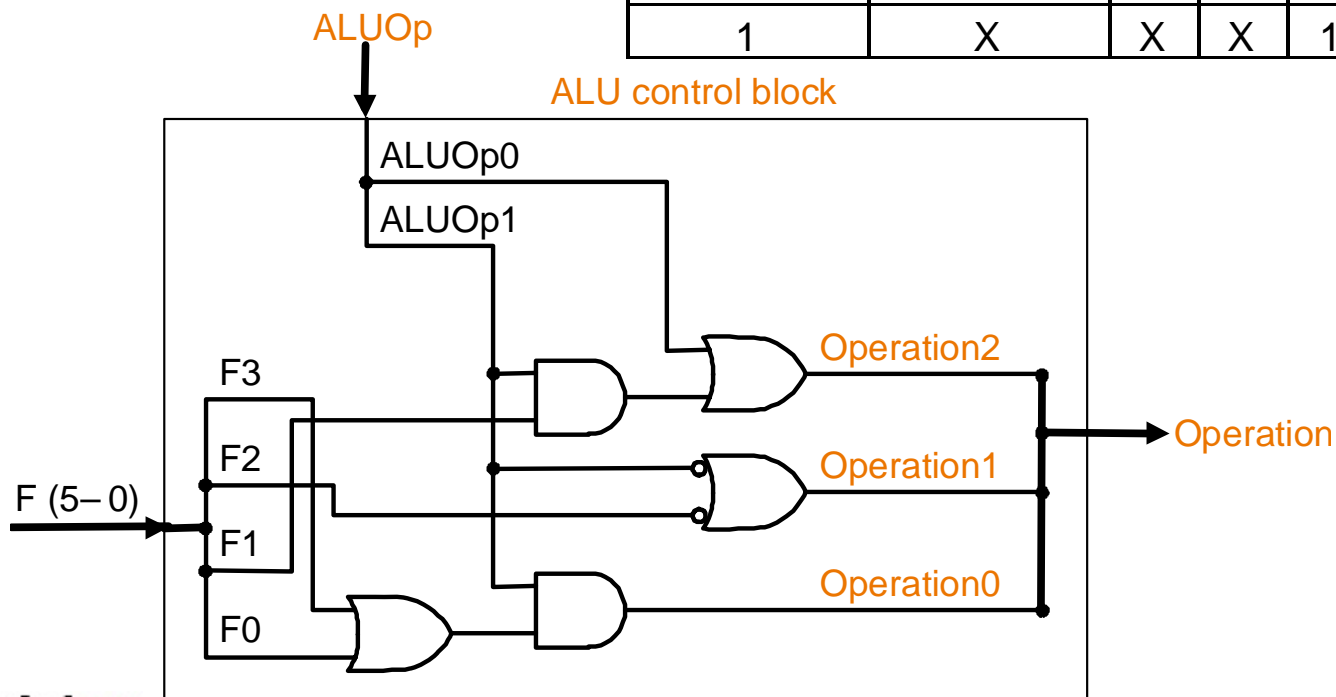
Instruction		Instruction	Funct	Desired	ALU Control
Opcode	ALUOp	Operation	Field	ALU Action	Operation
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100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

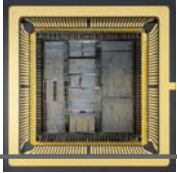


ALU Control - Truth Table & Implementation

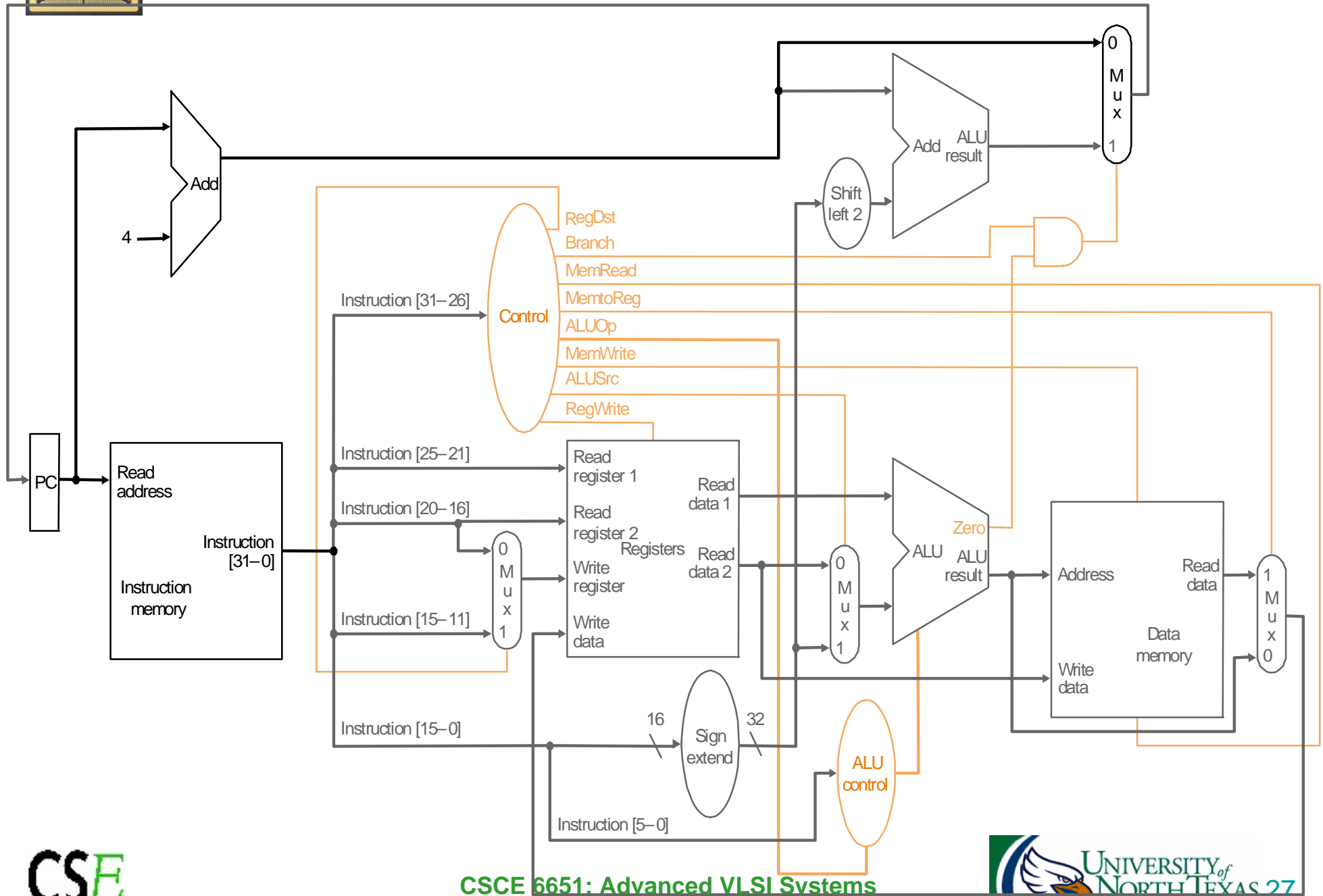
Describe it using a truth table (can turn into gates):

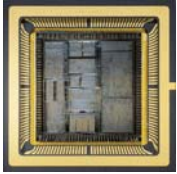
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111





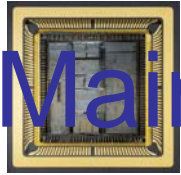
Designing the Main Control





Control Signals and their Effects

Signal Name	Effect When deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write 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 second register file output (Read data 2)	The second ALU operand is the sign-extended lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of $PC + 4$.	The PC is replaced by the output of the adder 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 comes from the ALU	The value fed to the register Write data input comes from the data memory.



Main Control: Truth Table & Implementation

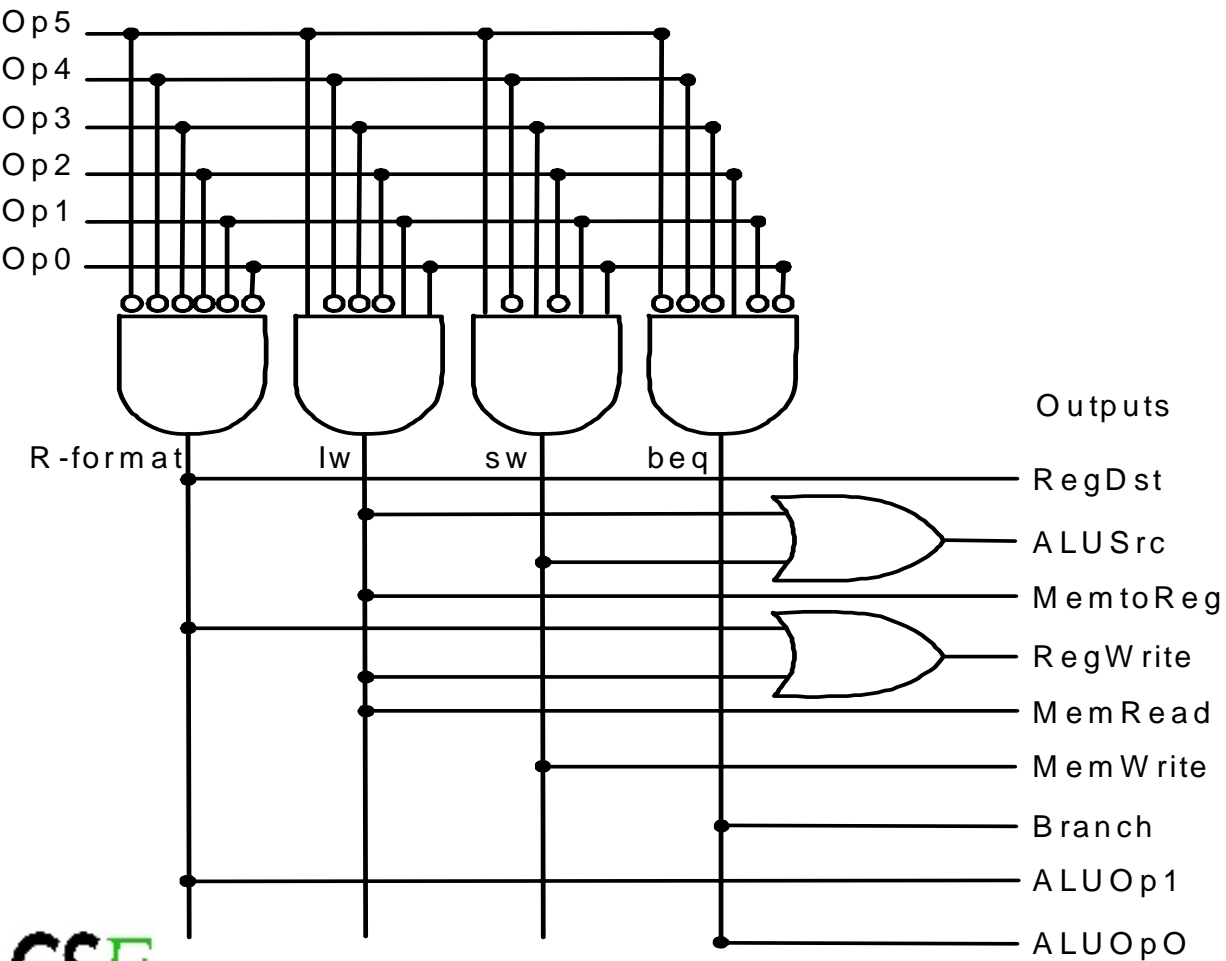
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

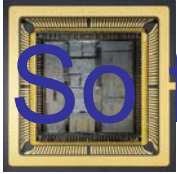
Inputs

Op5
Op4
Op3
Op2
Op1
Op0

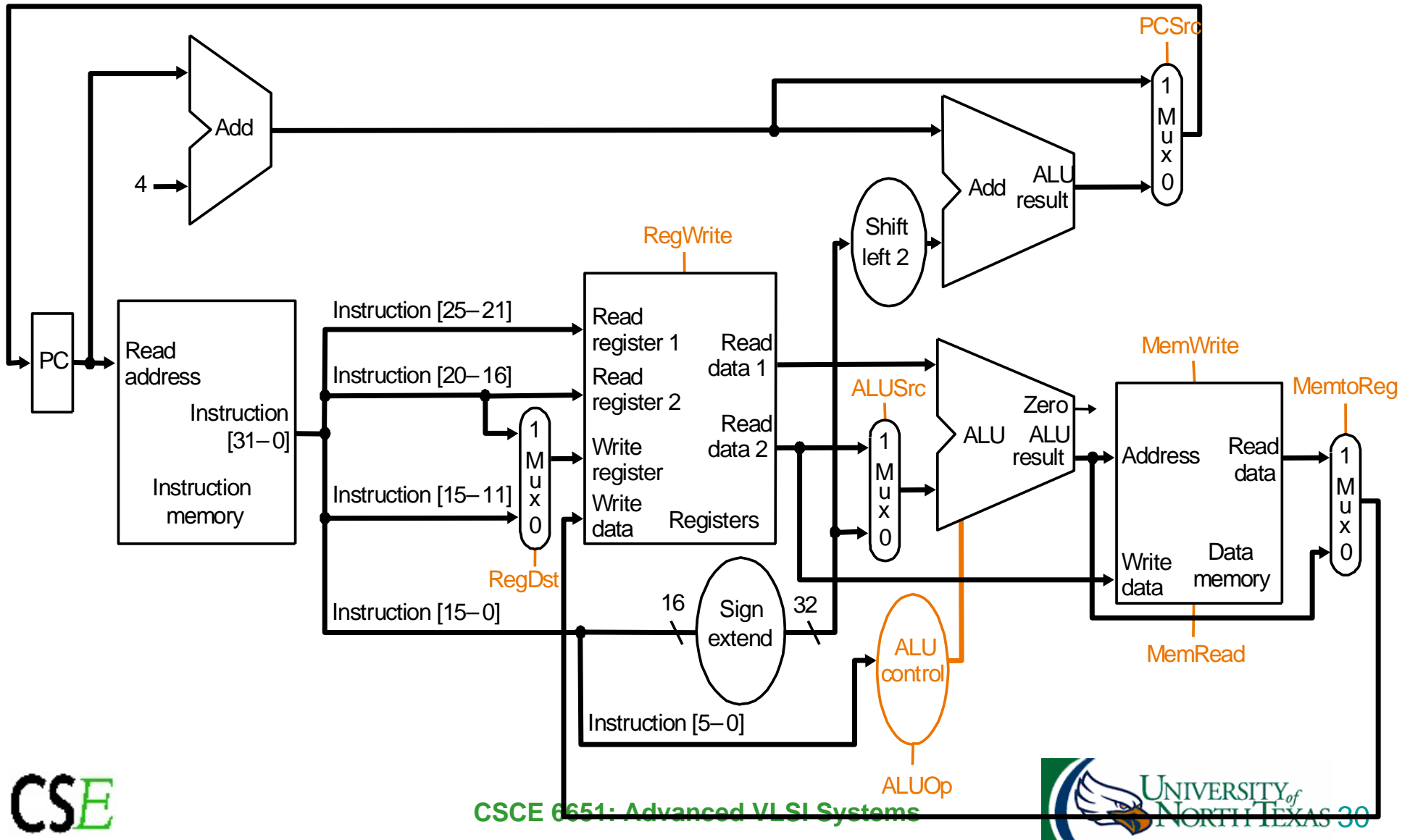
Outputs

RegDst
ALUSrc
MemtoReg
RegWrite
MemRead
MemWrite
Branch
ALUOp1
ALUOp0





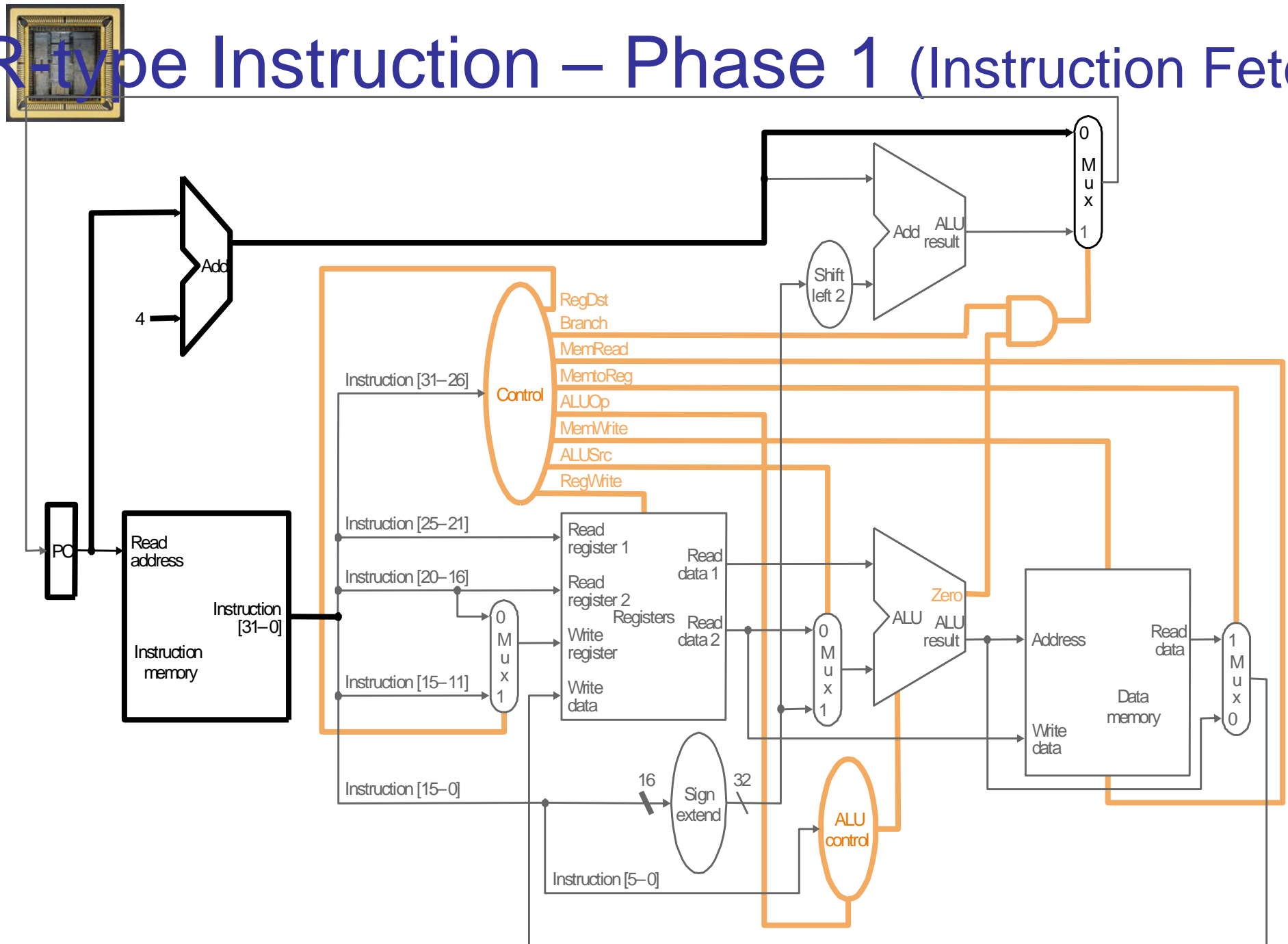
- Calculate cycle time assuming negligible delays except:
 - memory (2ns), ALU and adders (2ns), register file access (1ns)



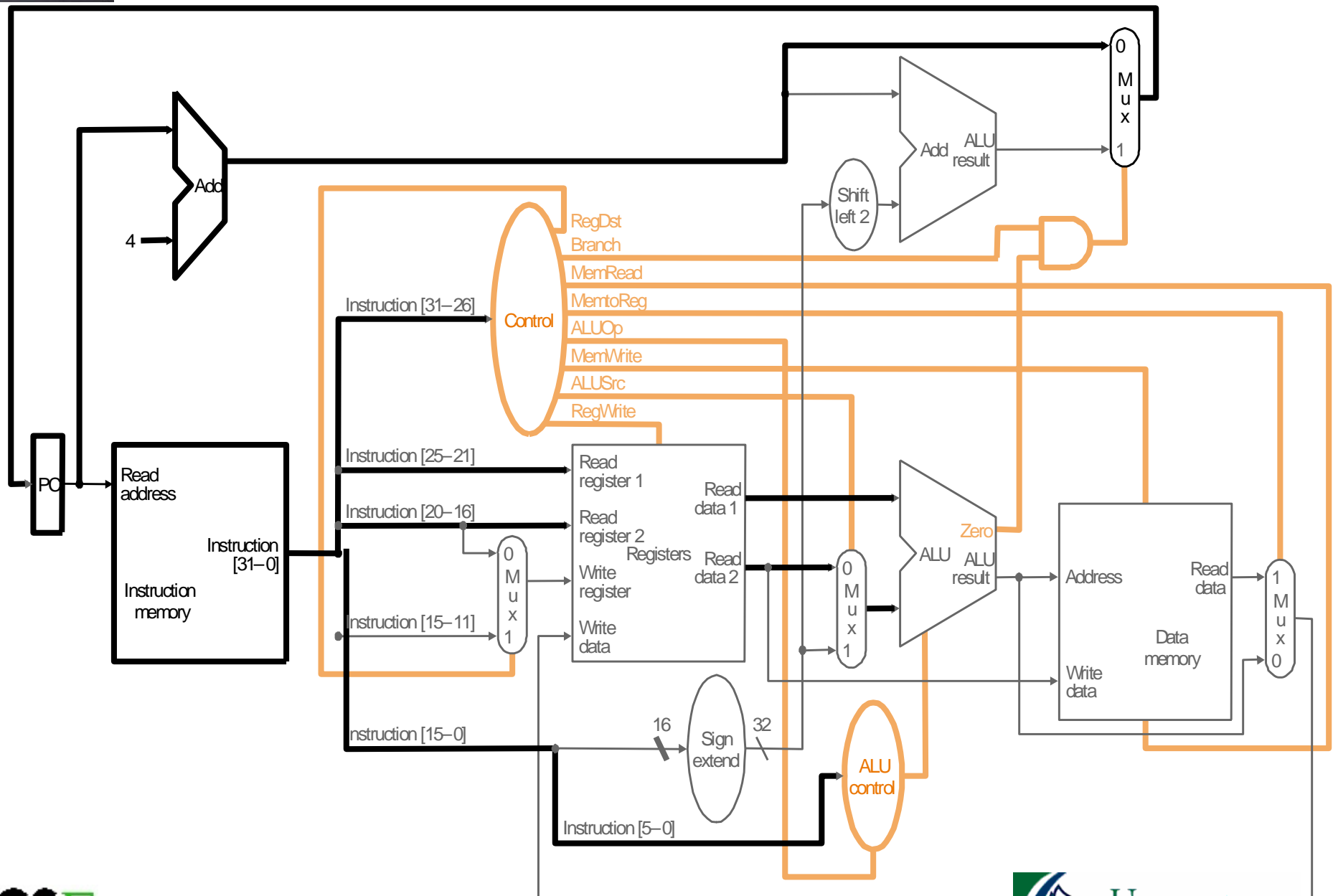
How does the single cycle datapath works?

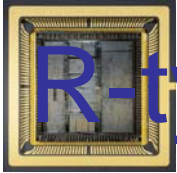
- Let us understand this by highlighting the portions of the datapath when an R-type instruction is executed
- For an R-type instruction we go through the following phases:
 - Phase 1: Instruction Fetch
 - Phase 2: Register File Read
 - Phase 3: ALU execution
 - Phase 4: Write the Result into the Register 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)

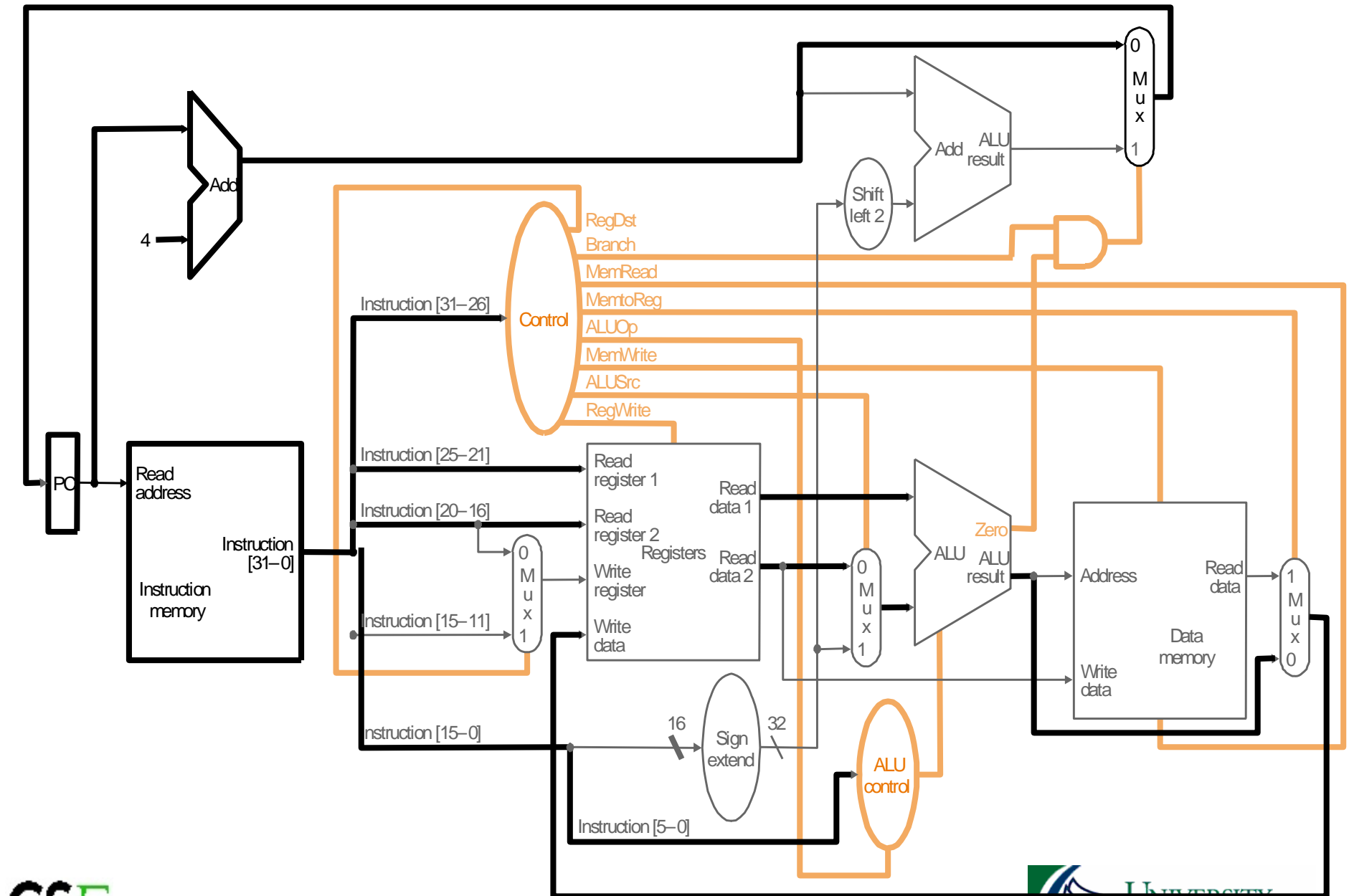


R-type Instruction – Phase 2 (Register Read)

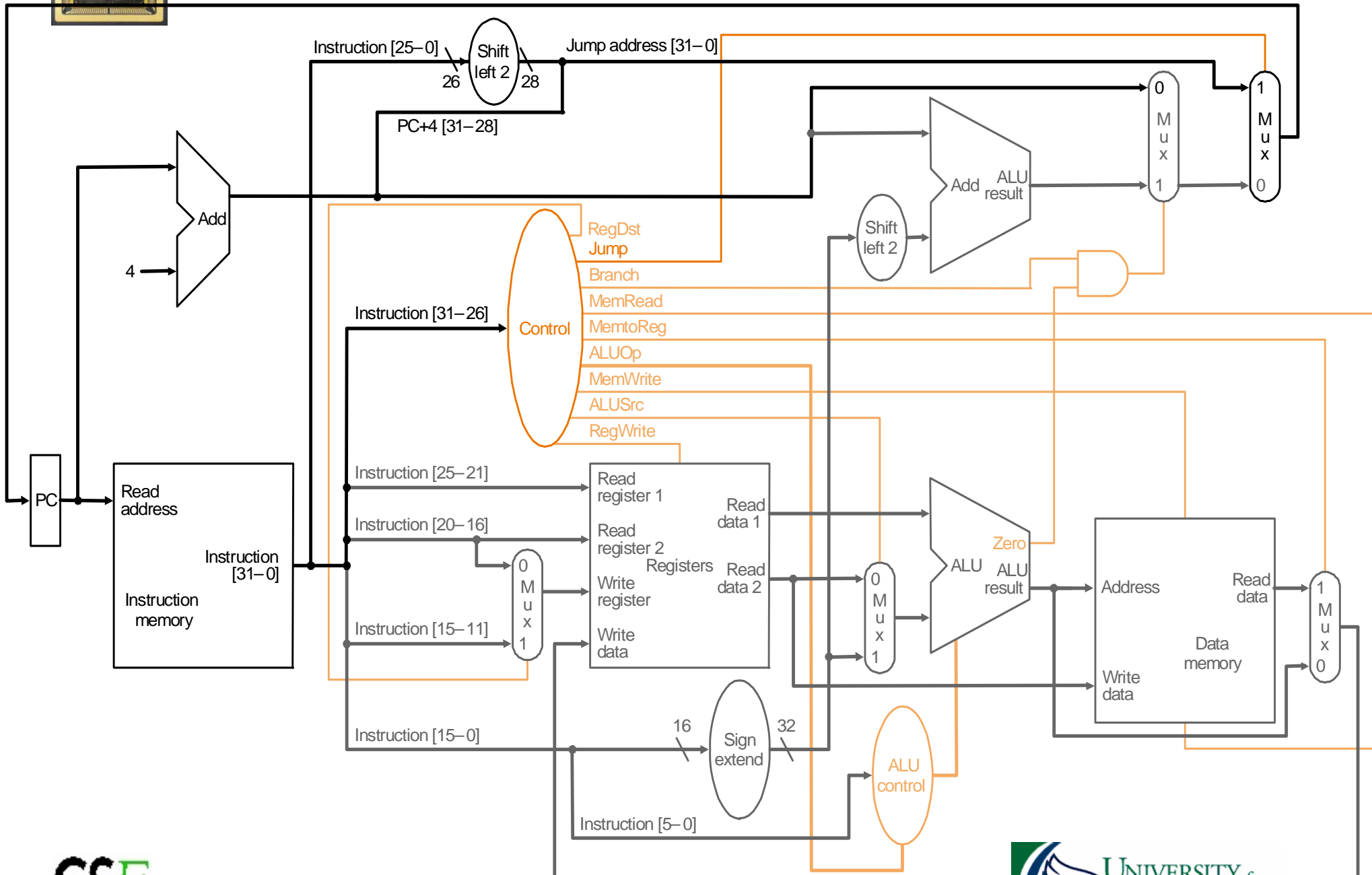


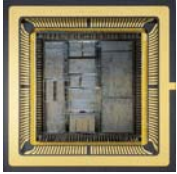


R-type Instruction – Phase 4 (Write the Result)



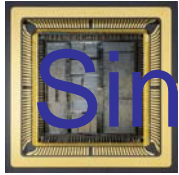
How do we handle jump?





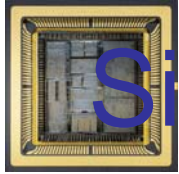
Where we are headed

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



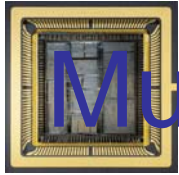
Single Cycle Implementation: Summary

- 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
- Suggestion: STARE, STARE, STARE at the single cycle datapath to familiarize yourself
- Given the single cycle datapath, you should be able to “highlight” active portions of the datapath for any given instruction (just as we did for an R-type instruction in the class)

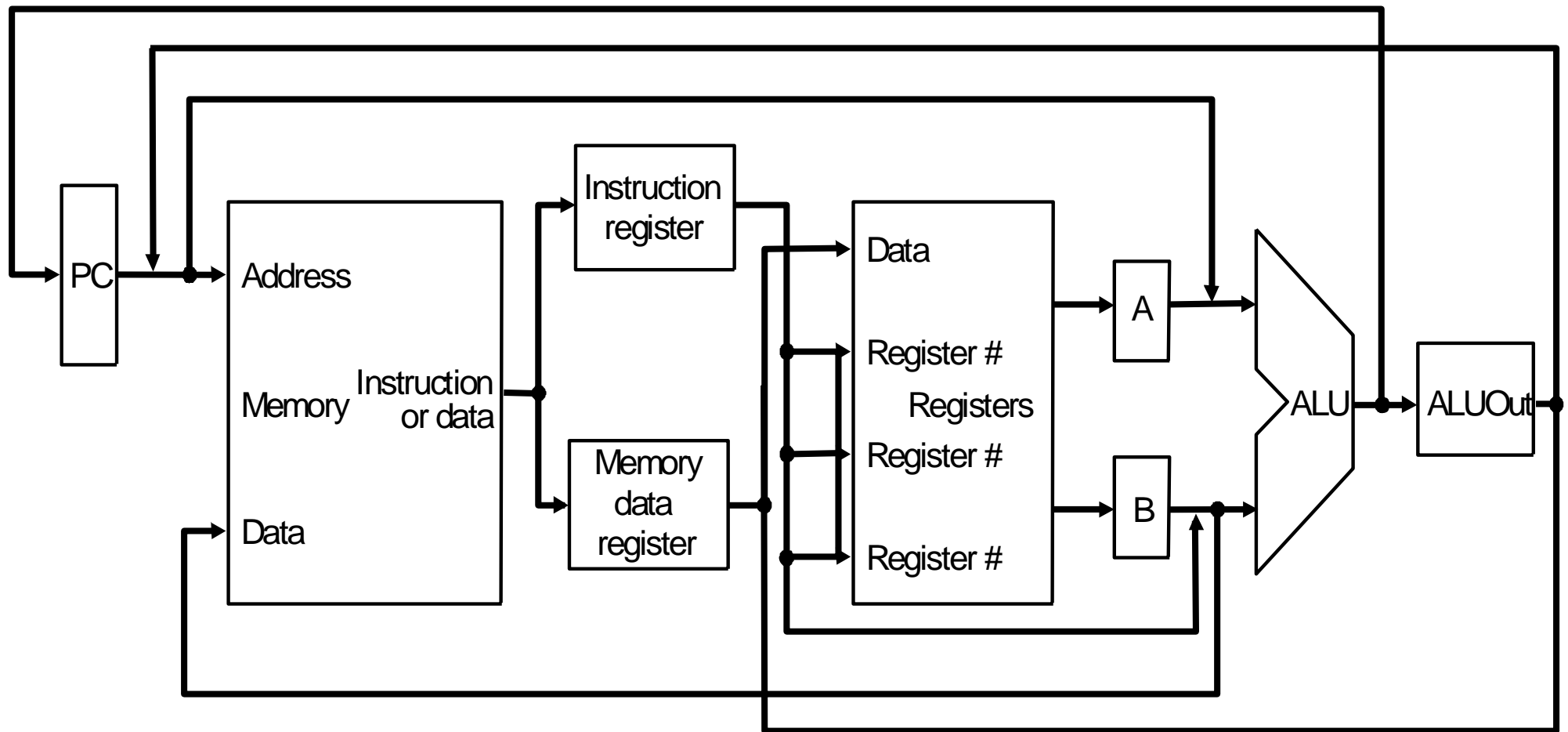


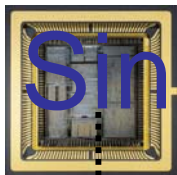
Single Cycle Implementation - Recap

- 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:

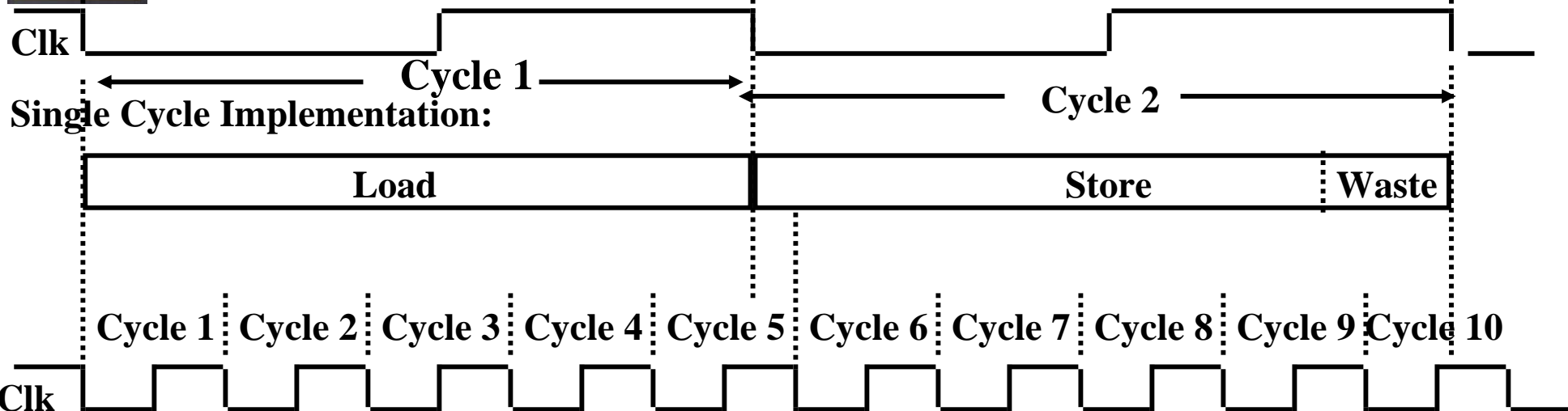


Multicycle Approach – High Level View





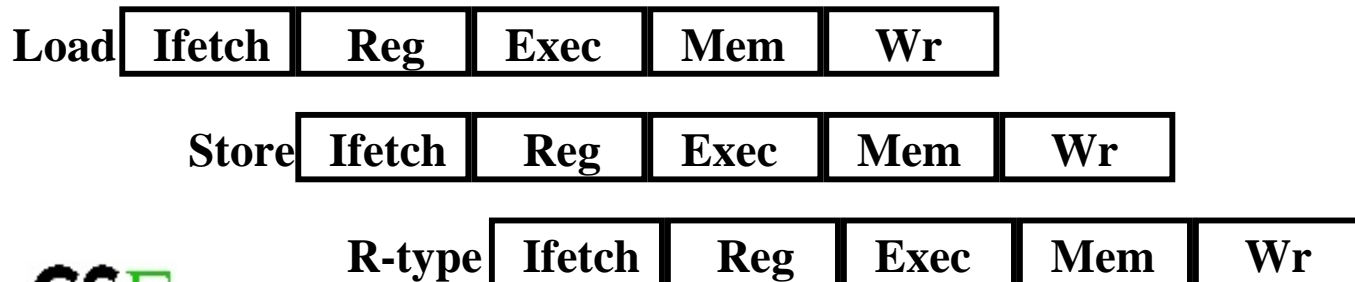
Single Cycle, Multiple Cycle, vs. Pipeline

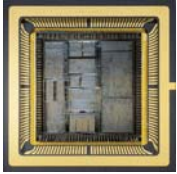


Multiple Cycle Implementation:

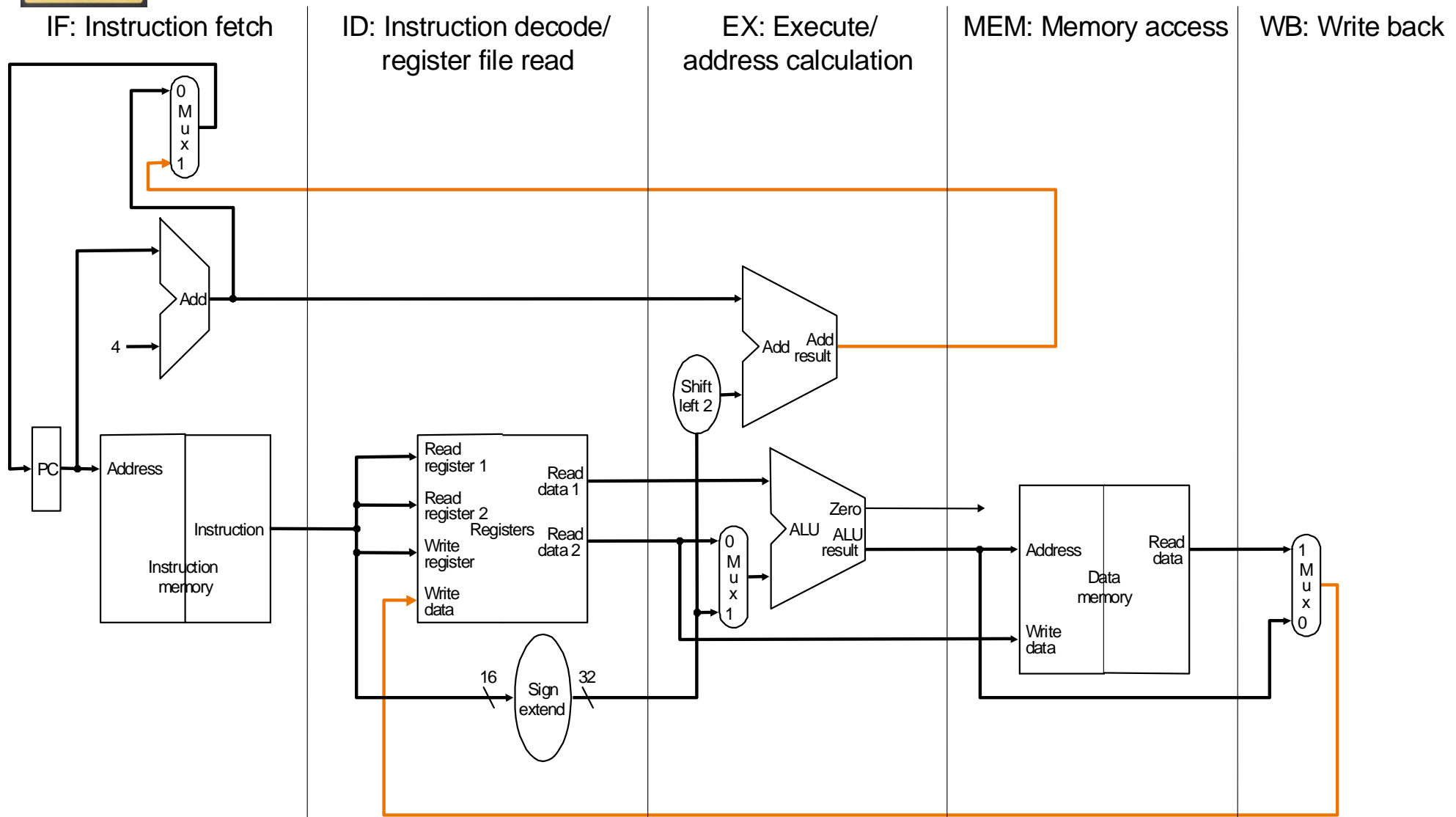


Pipeline Implementation:

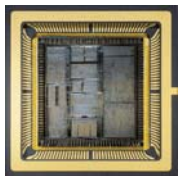




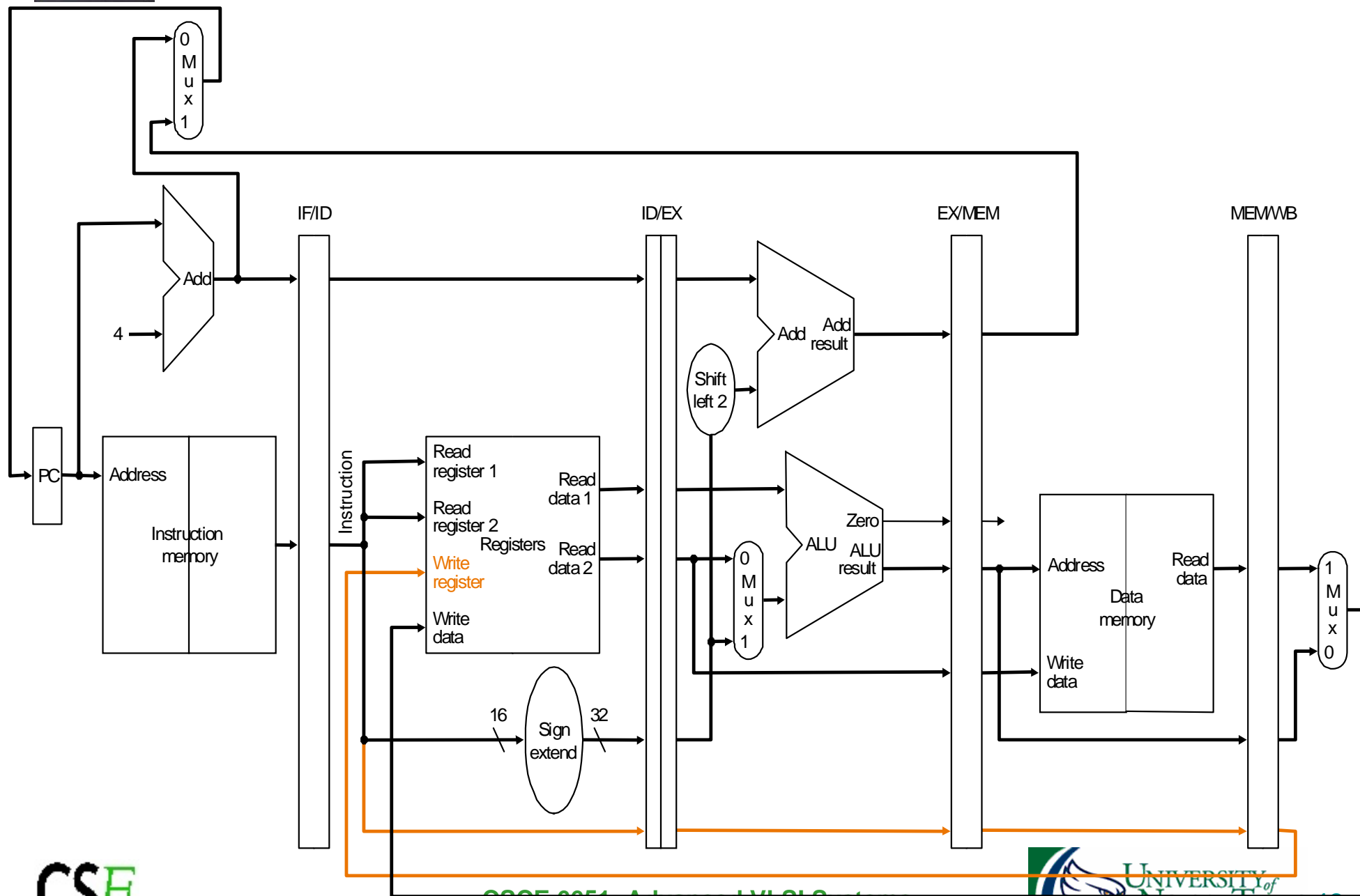
Basic Idea

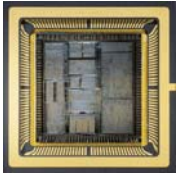


What do we need to add to actually split the datapath into stages?

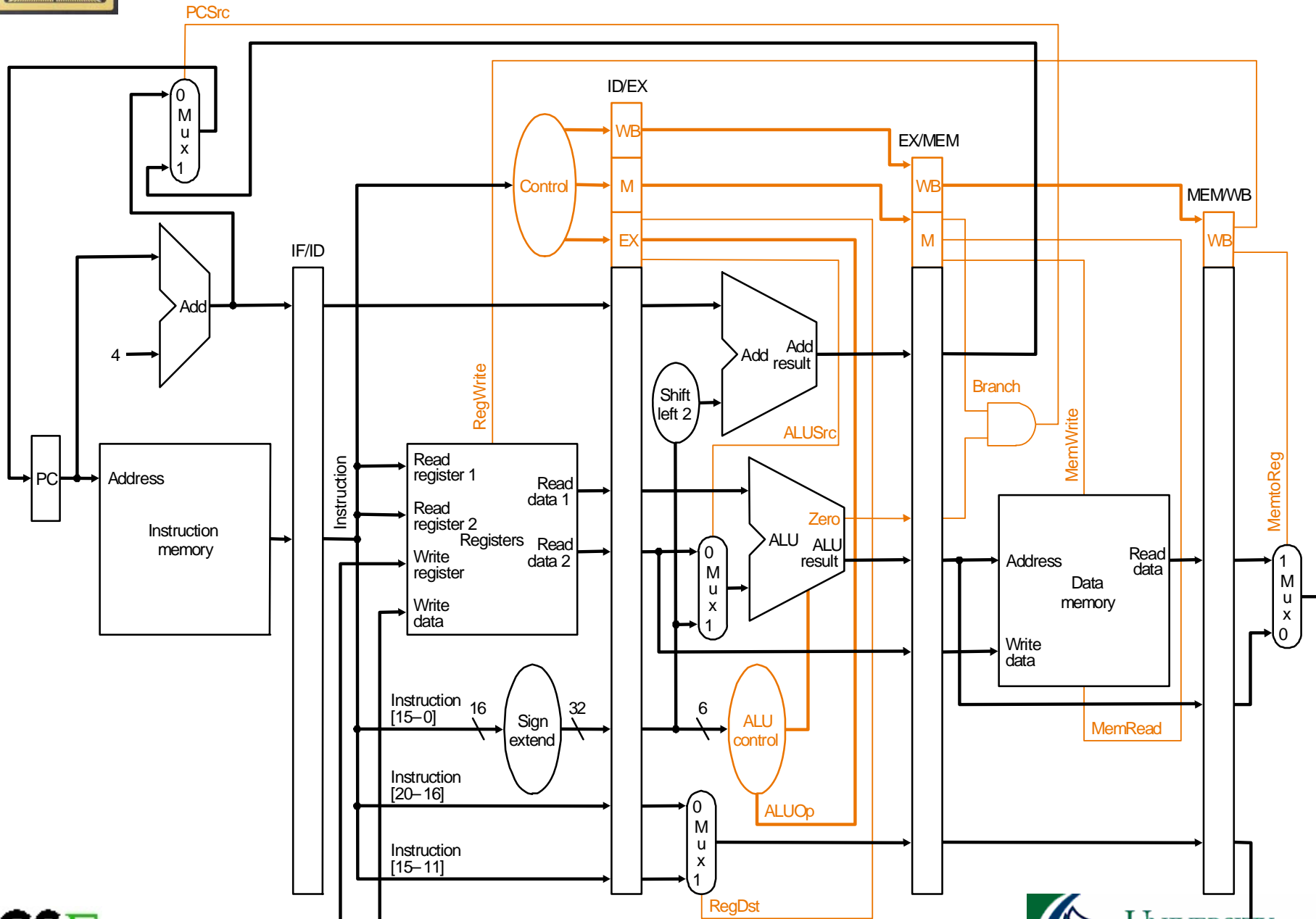


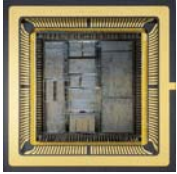
Corrected Datapath





Datapath with Control





Pipelining Summary

- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to “**fill**” pipeline and time to “**drain**” it reduces speedup
- Three types of pipeline hazards: structural, data, and control/branch
- Stalling helps any kind of hazard
- Data hazard solutions: Stalling, Data forwarding, Hazard detection
- Control or Branch Hazard solutions: Stalling, Branch Prediction, Delayed Branching