COMP 273
Introduction to Computer Systems

COMP 273

Micro Architecture

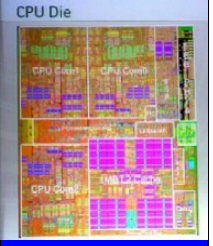
Part 2

The Pipeline CPU

Prof. Joseph Vybihal

Announcements

- Assignment #2 posted
- Midterm exam
  - Feb 12, 2015
  - In class
  - 1.5 hours



COMP 273
Introduction to Computer Systems

Readings

- Computer Organization & Design
  - Chapter 4
  - Appendix D
- Check out the resources on the course web site.
-

At Home

- If you made up two machine language instructions, how would you wire the control unit?
- Soul Of A New Machine

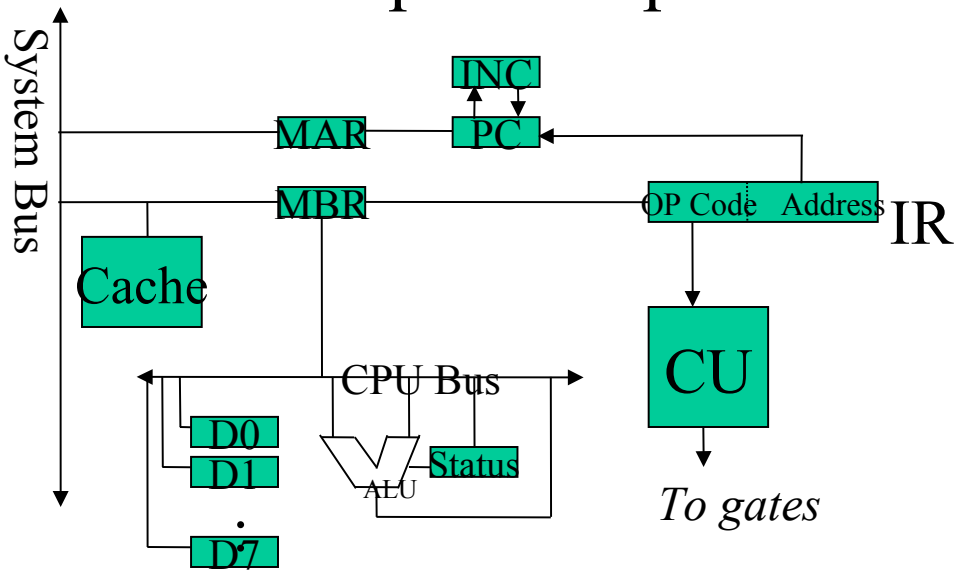
COMP 273
Introduction to Computer Systems

Part 1

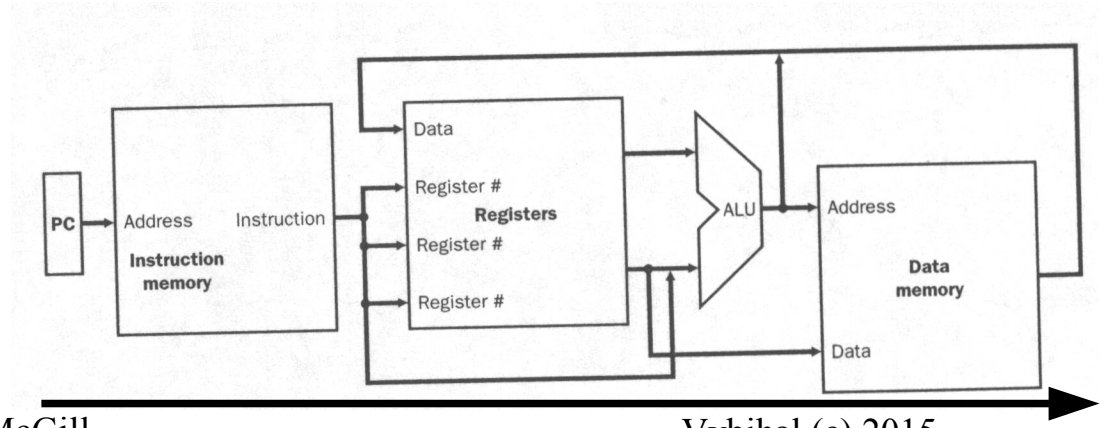
The Pipeline CPU

Pipeline as Optimized Architecture

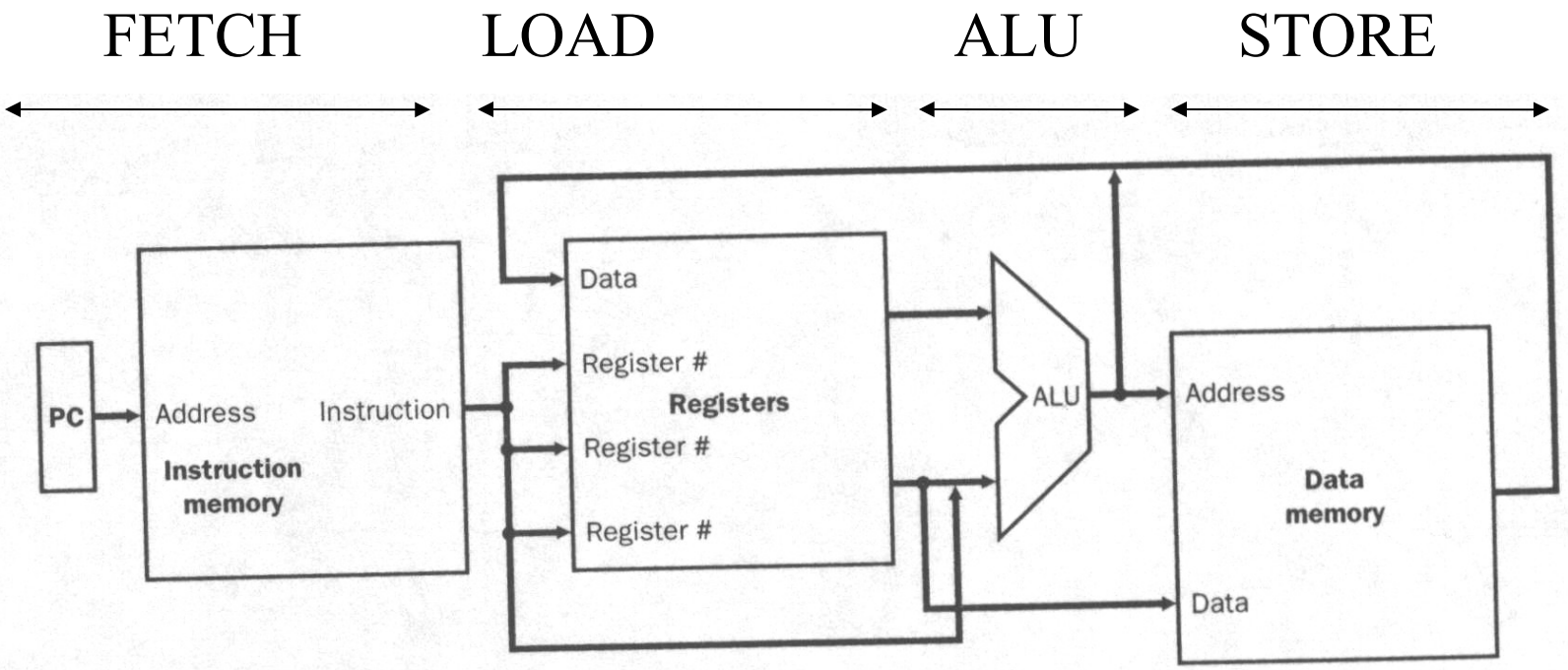
- Classic CPU Architecture vs. Pipeline Optimized



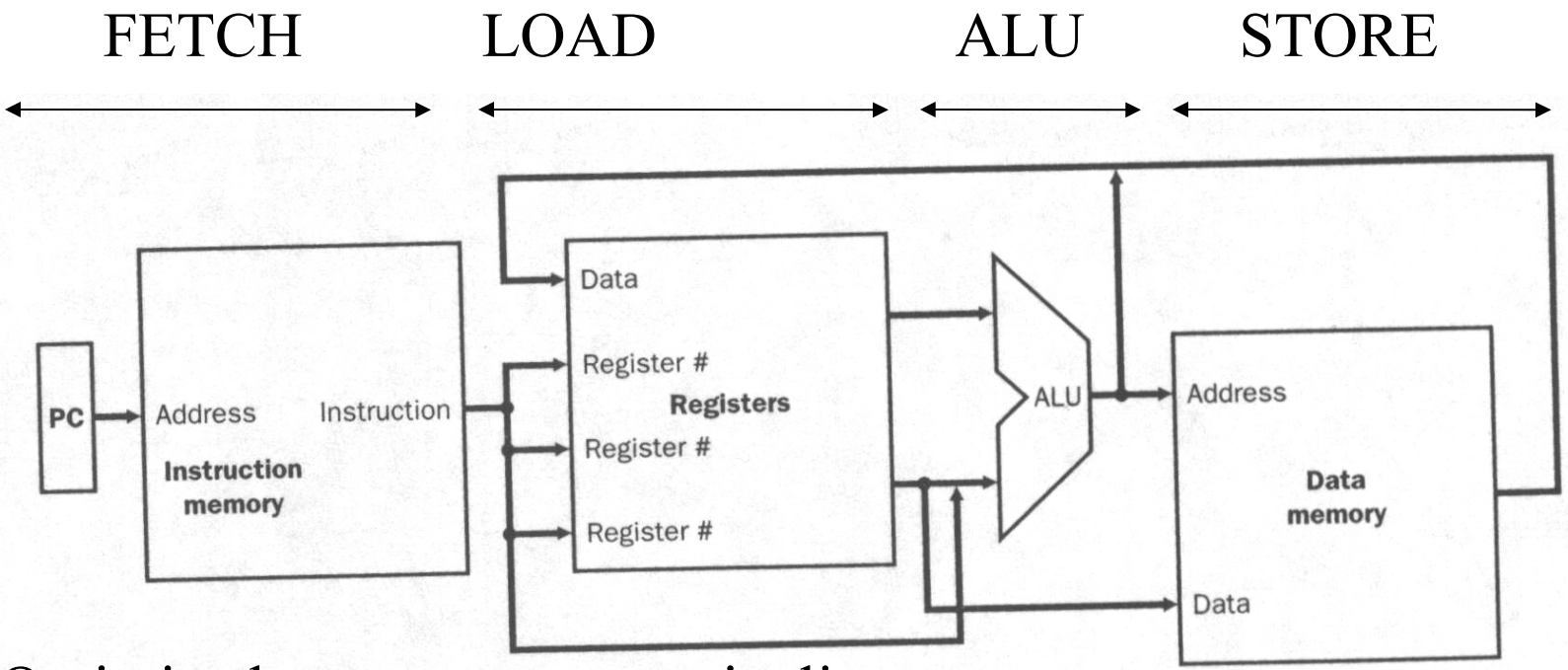
- Unused circuits
- Clock tick sharing



Pipeline Architecture



Pipeline Architecture

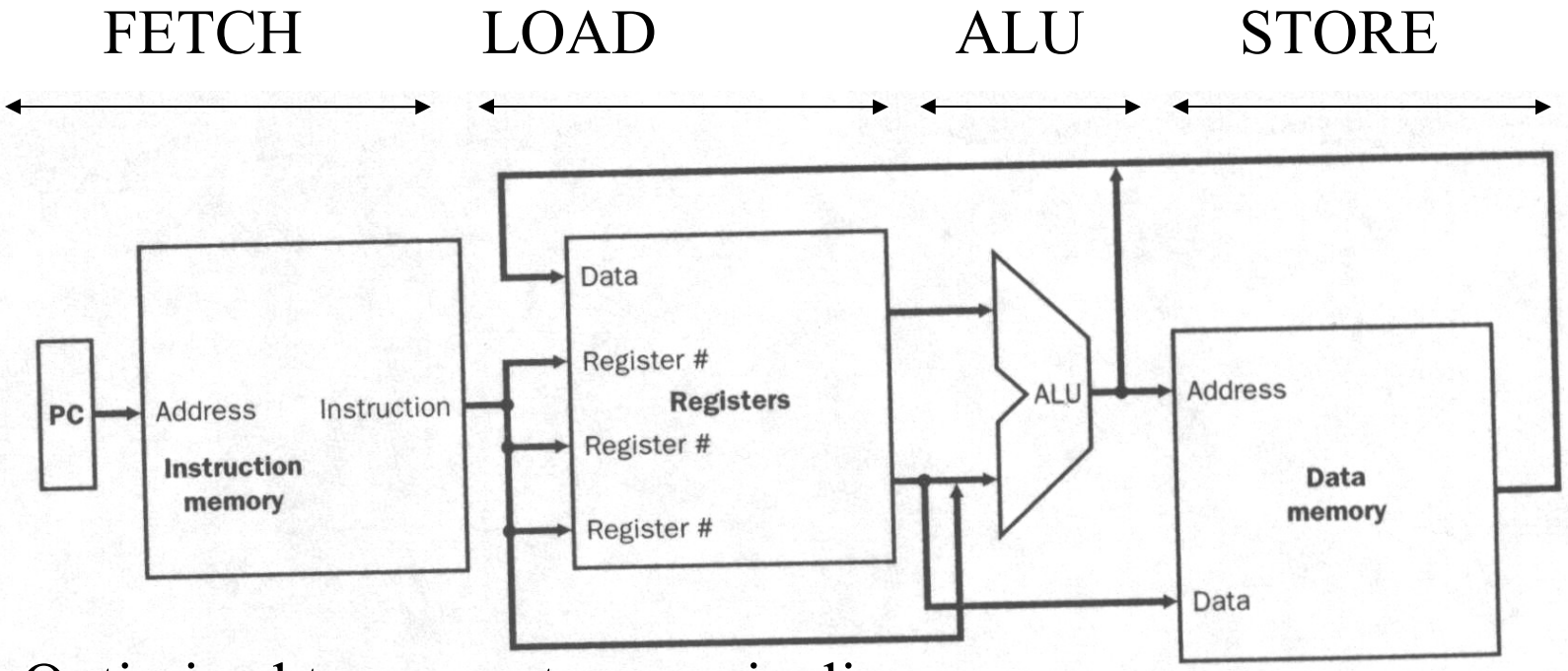


Optimized to compute as a pipeline:

- As we are doing a FETCH of the next instruction…
- We can also do a LOAD of the current instruction
- Actually, 4 instructions can be executed in stages at the "same time"

Pipeline Architecture

ADD Y,5,2

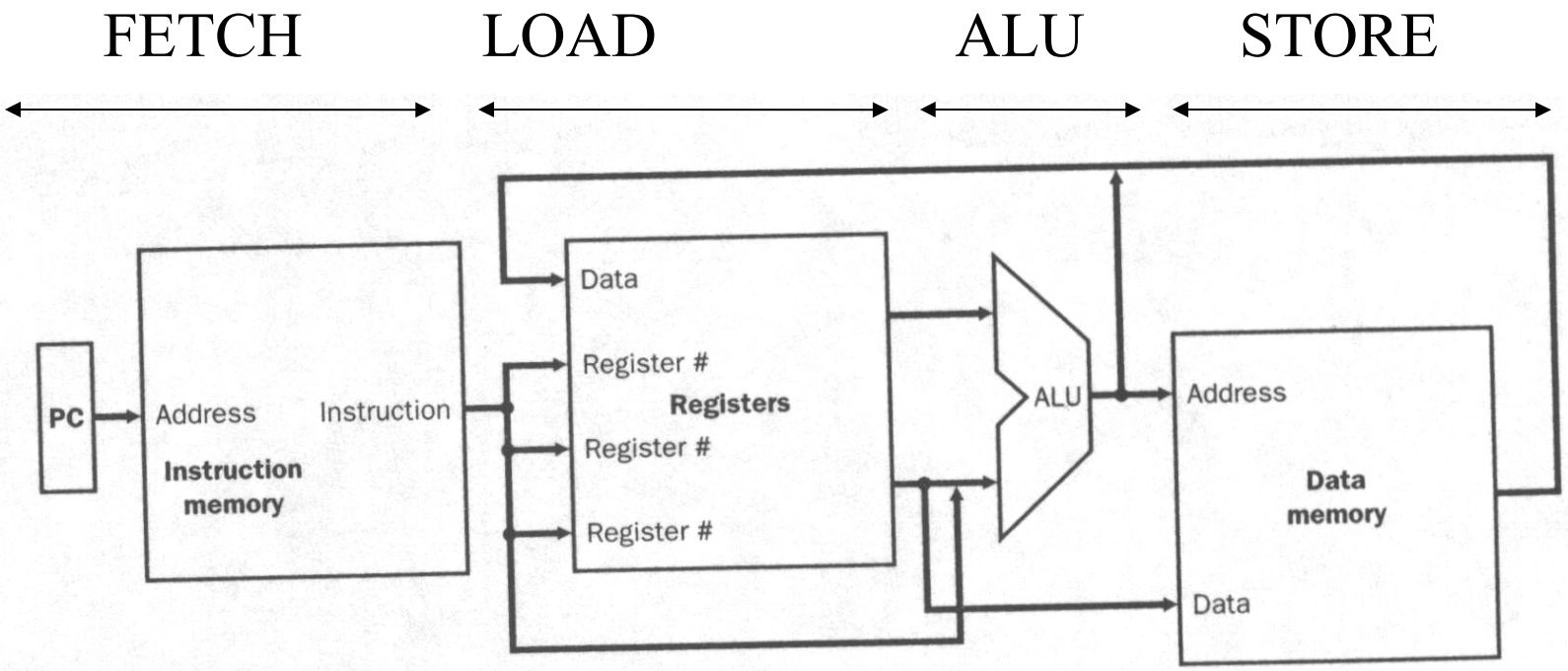


Optimized to compute as a pipeline:

- As we are doing a FETCH of the next instruction…
- We can also do a LOAD of the current instruction
- Actually, 4 instructions can be executed in stages at the same time

Pipeline Architecture

LOAD R0,X ADD Y,**5,2**



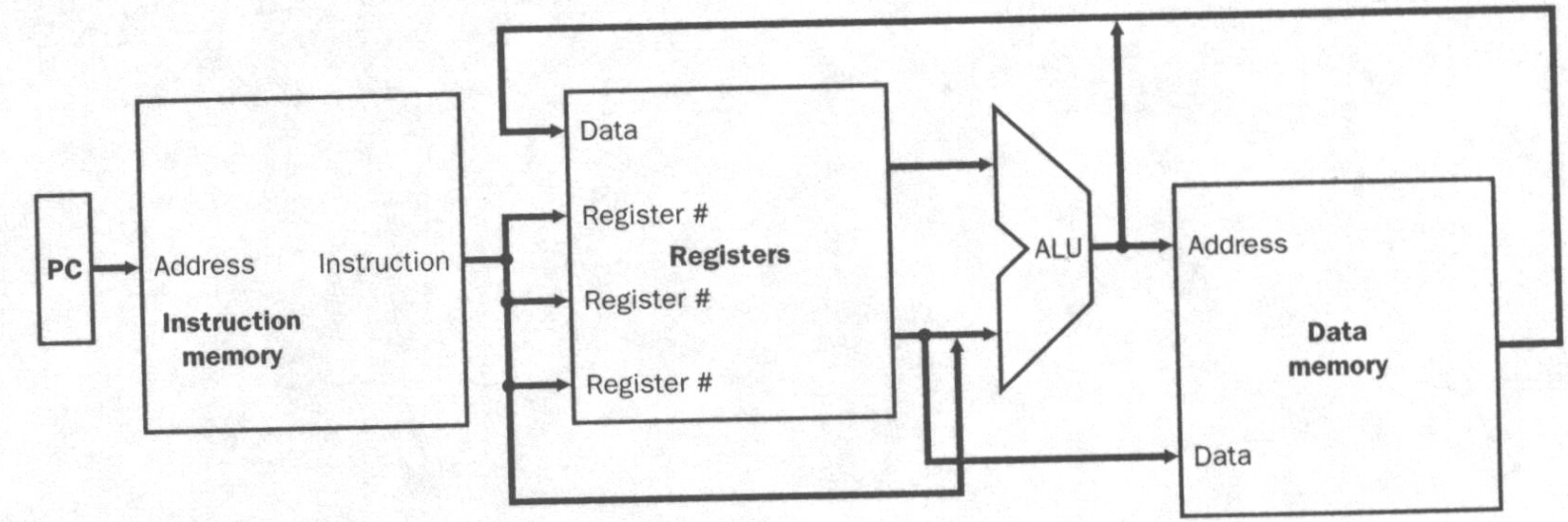
Optimized to compute as a pipeline:

- As we are doing a FETCH of the next instruction…
- We can also do a LOAD of the current instruction
- Actually, 4 instructions can be executed in stages at the same time

Pipeline Architecture

BREQ R0,Z LOAD R0,**X** **ADD** Y,5,2

FETCH LOAD ALU STORE



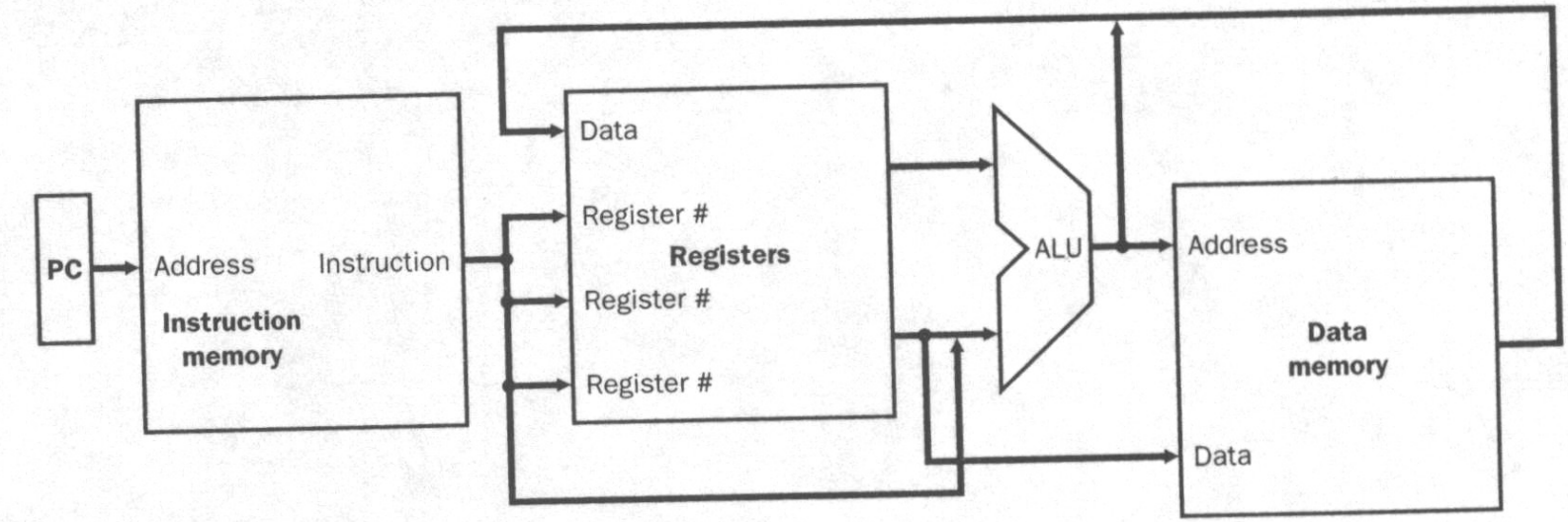
Optimized to compute as a pipeline:

- As we are doing a FETCH of the next instruction…
- We can also do a LOAD of the current instruction
- Actually, 4 instructions can be executed in stages at the same time

Pipeline Architecture

BREQ **R0**,Z LOAD **R0,X** ADD **Y**,5,2

FETCH LOAD ALU STORE

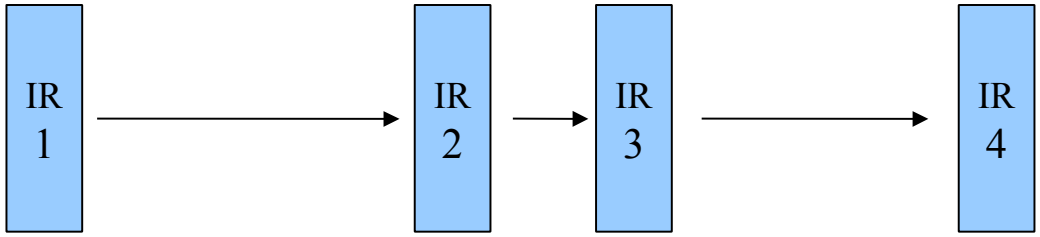


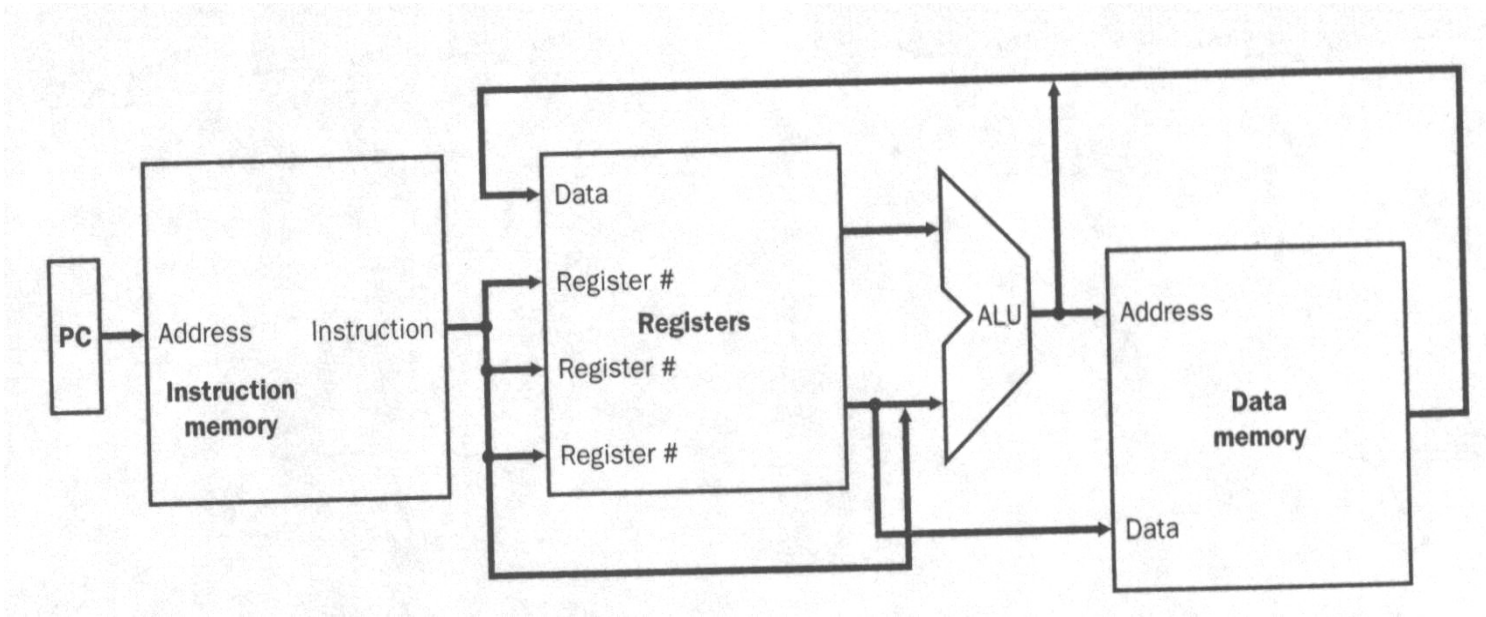
Optimized to compute as a pipeline:

- As we are doing a FETCH of the next instruction…
- We can also do a LOAD of the current instruction
- Actually, 4 instructions can be executed in stages at the same time

Pipeline Architecture

Gated Control of instruction registers



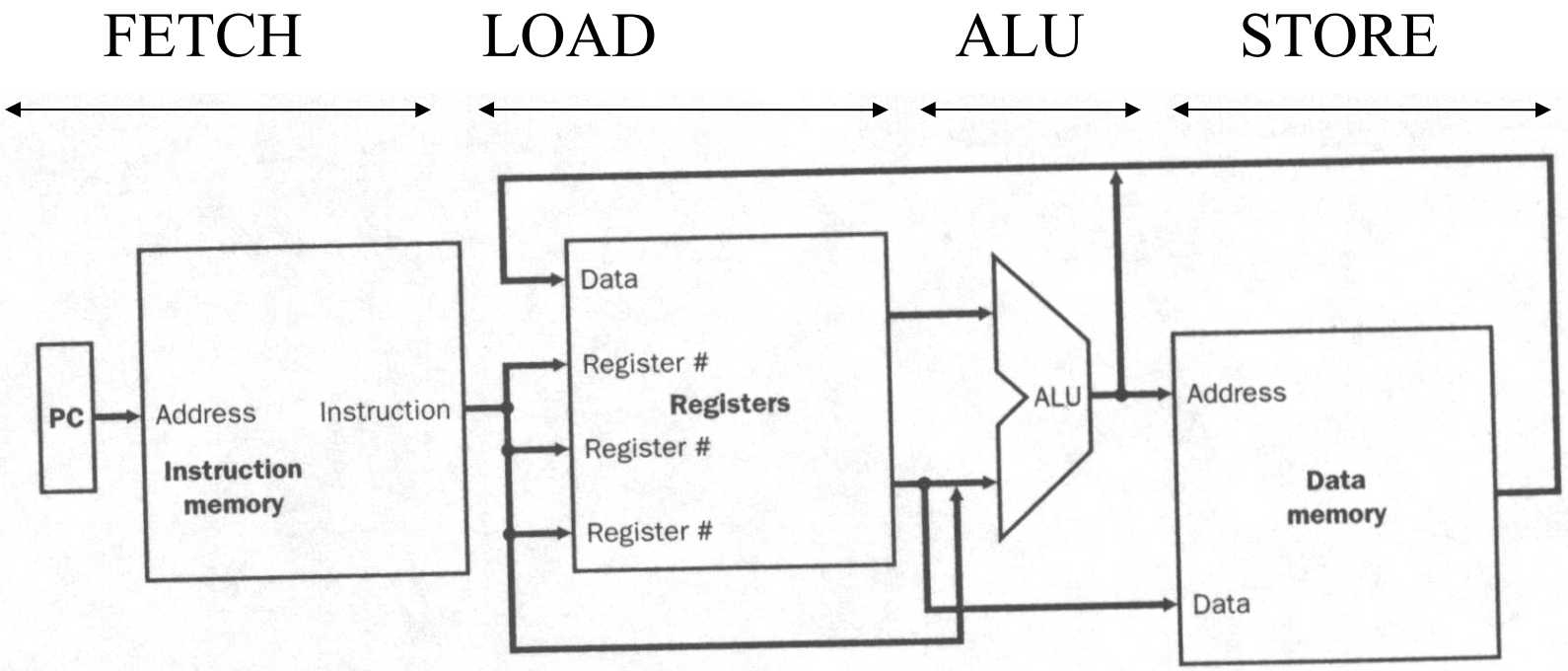


Linearly optimized for assembly line execution of instruction

Pipeline Architecture

Cache:

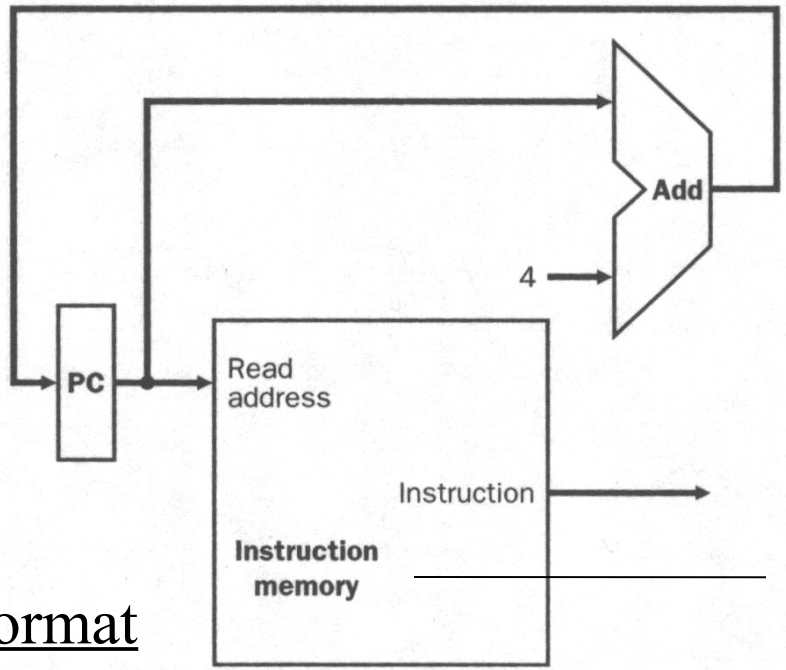
- Two caches! - memory map
- Code / Load prediction – dumb vs. Smart vs. "AI"



Question

- What does all this mean for the IR and CU?
  - No longer 1:1 with IR and PC
  - One long CU or many mini CU s?
- The pipeline needs to be gated into CU controlled stages/steps

FETCH Portion of CPU



+4 since 32-bit instr.

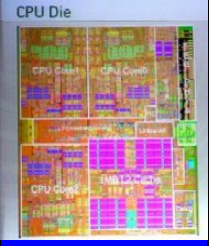
More than 1 format exists

Notice that code is separated from data

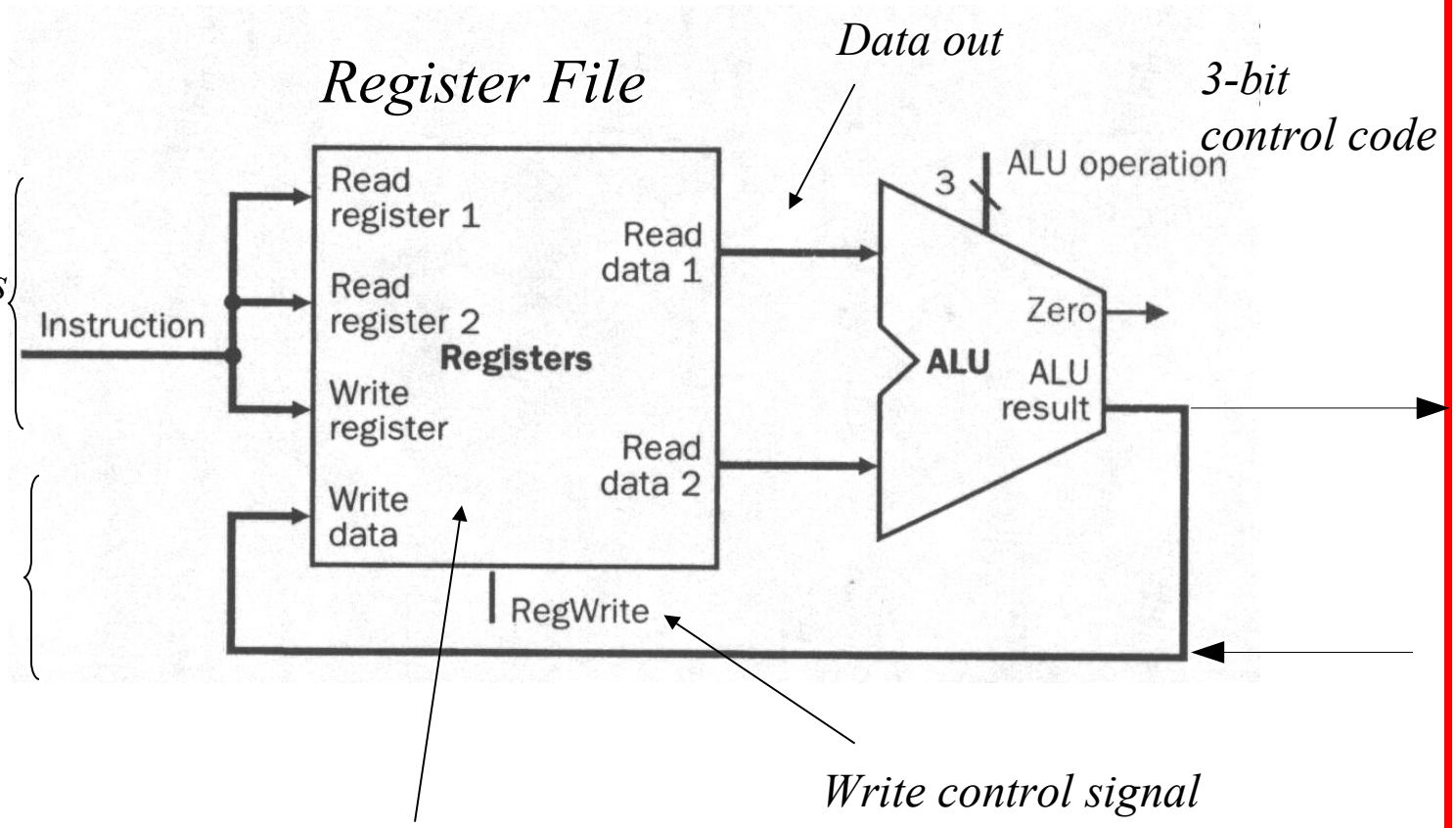
Instruction R-Format

OP = Op-code
RS = register source
RT = second register source
RD = register destination
SHAMT = shift amount (jump)
FUNCT = Function-code (i.e. sub-op-code)

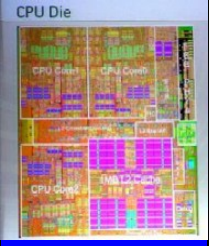
GATES?



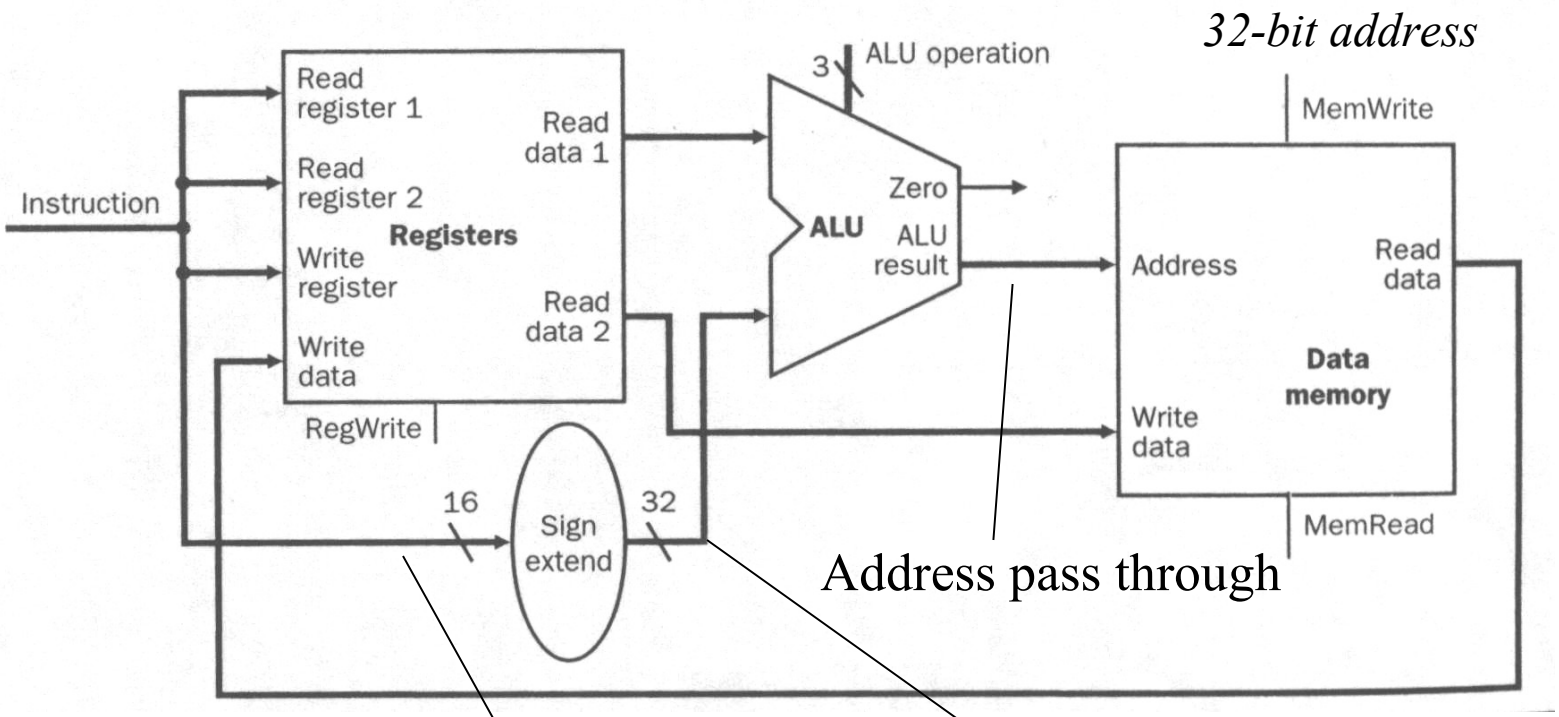
LOAD Portion of CPU



PARTIAL VIEW



Full View of LOAD & ALU



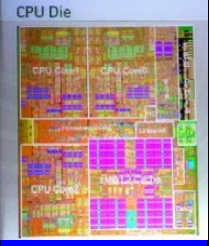
Full view includes data from memory:

- Data memory unit (the data cache portion)
- Sign-extension unit

Wired bit increase

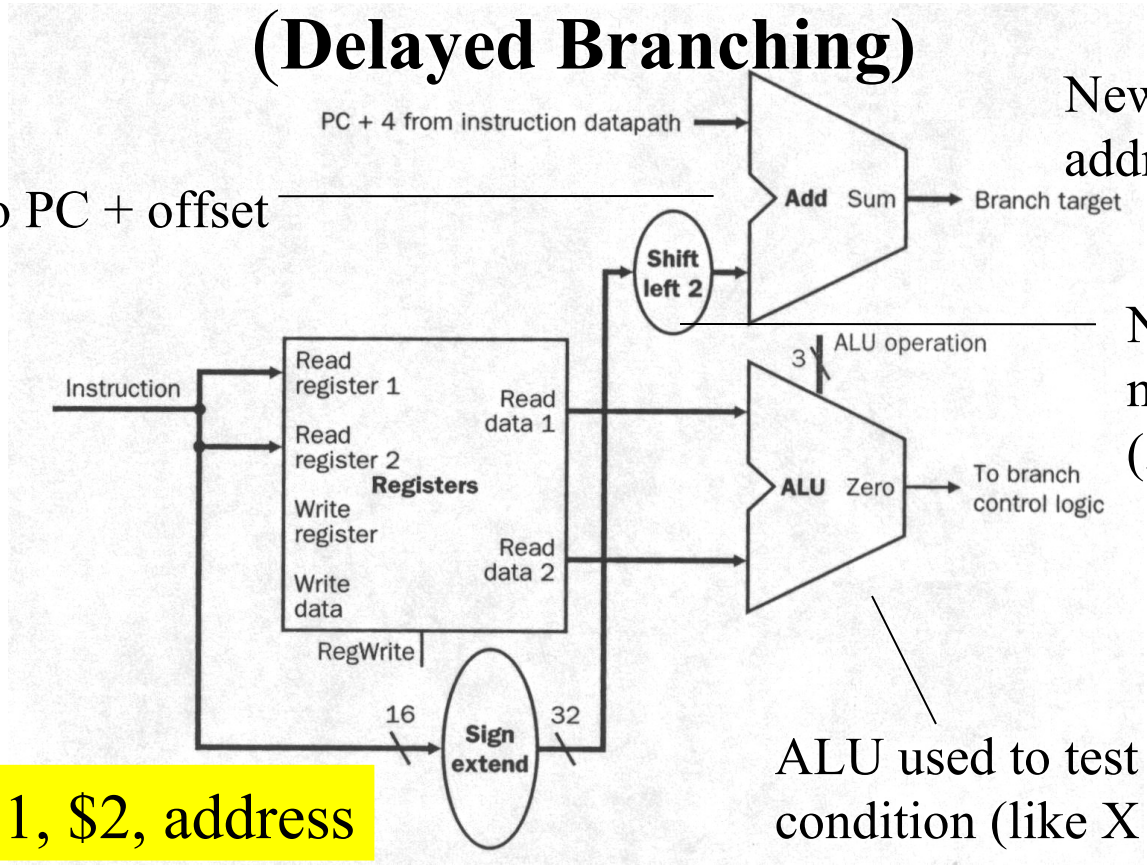
Short constant stored in instruction

LW $3, 8($2)



ALU Portion of CPU

(Delayed Branching)



Branch to PC + offset

New destination address

New address multiple of 4 (32-bit offset)

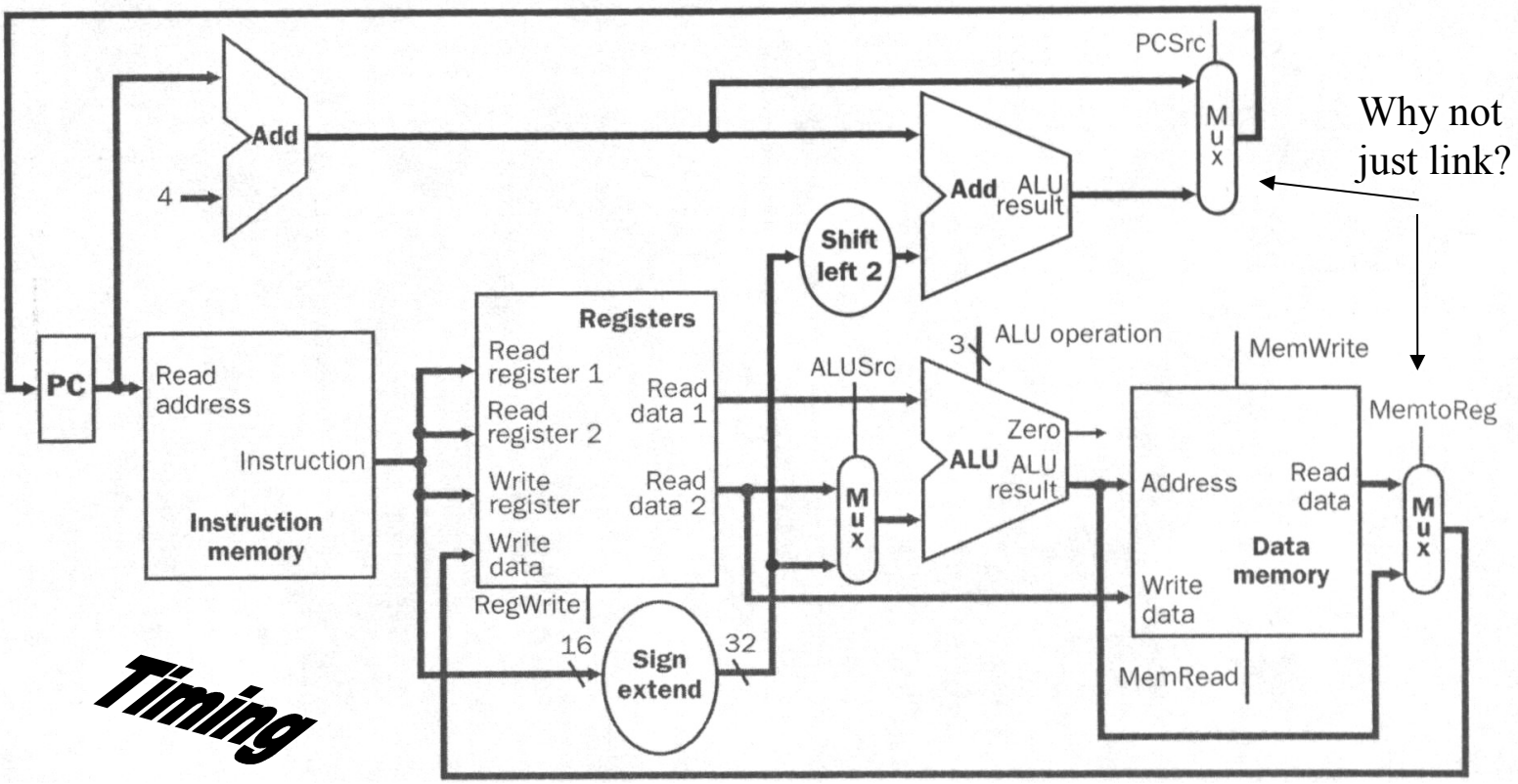
ALU used to test branch condition (like X > Y)

BEQ $1, $2, address

Since pipeline has already evaluated the next instruction regardless of the branch result the actual execution of a branch causes the pipeline to reset and discard the preceding staged executed instructions

Summary Layout #1

Single Clock Cycle: Load / Store / ALU / Branch



Many different instruction classes can be executed at once.
Dark lines show how a branch activates certain pathways.

The ALU

"Everything" passes through the ALU

- Some instructions don't need the ALU

Multi-Purpose ALU

- Depending on instruction class
  - Load/Store: compute memory address
  - R-type: AND, OR, Sub, Add, set-on-less-than
  - BEQ: uses subtraction
- Managed by the 3-bit ALU operation lines

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
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

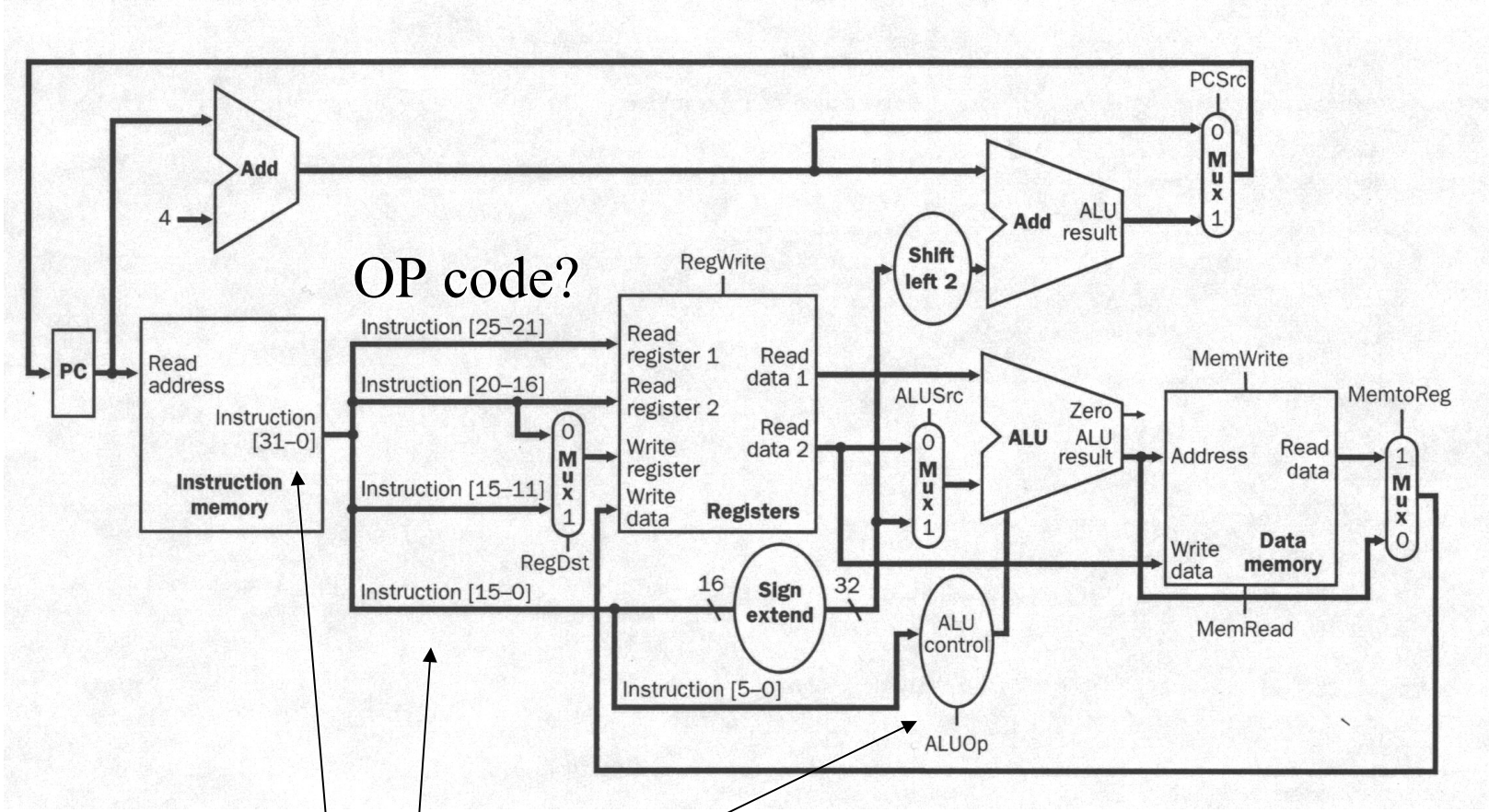
ALU Control Bits

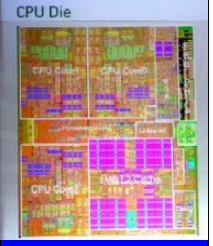
CU IR ALU

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

Summary Layout #2

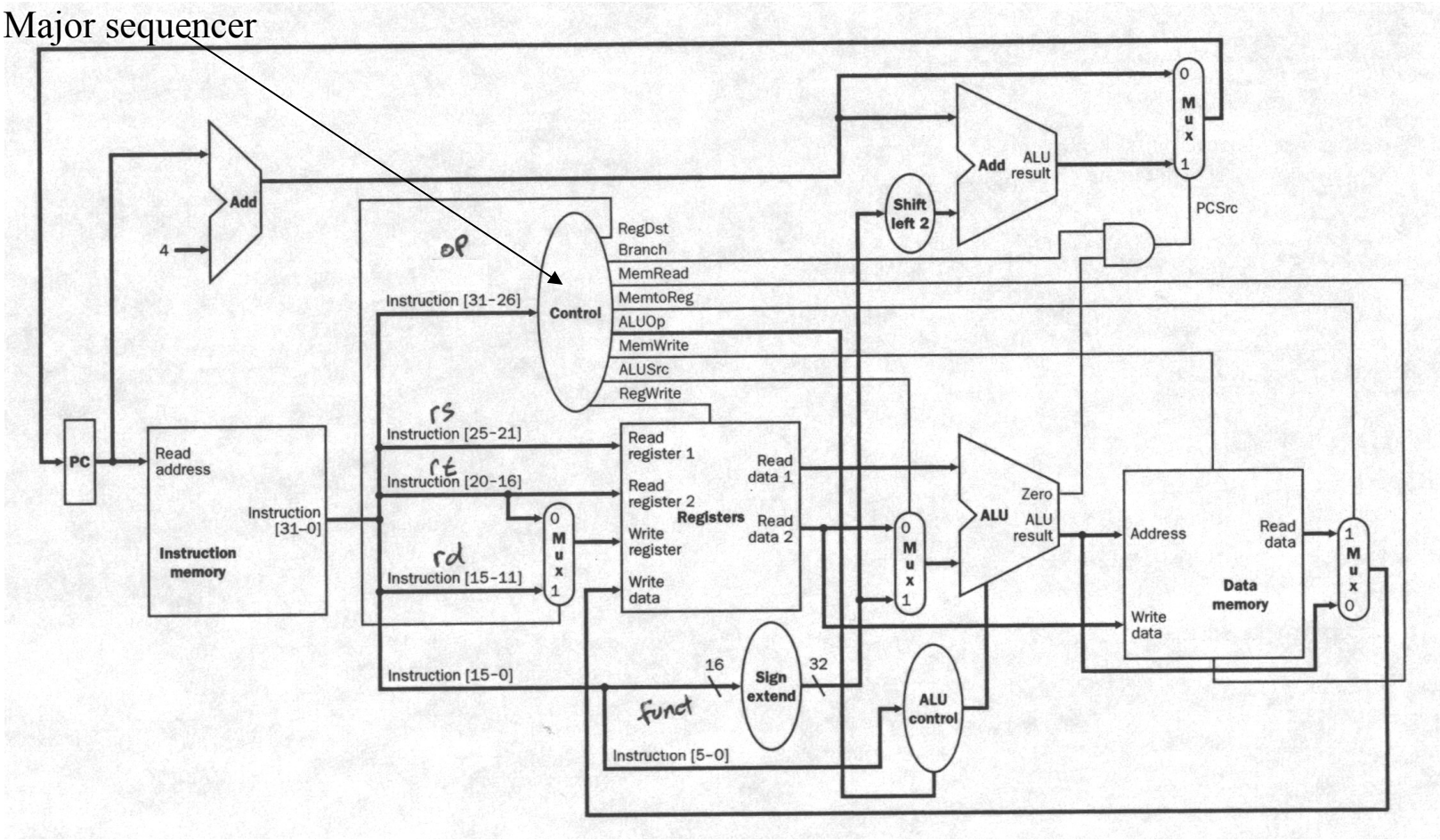
With bit layouts identified





Summary Layout #3
With Control Unit Circuitry (internal)

Major sequencer

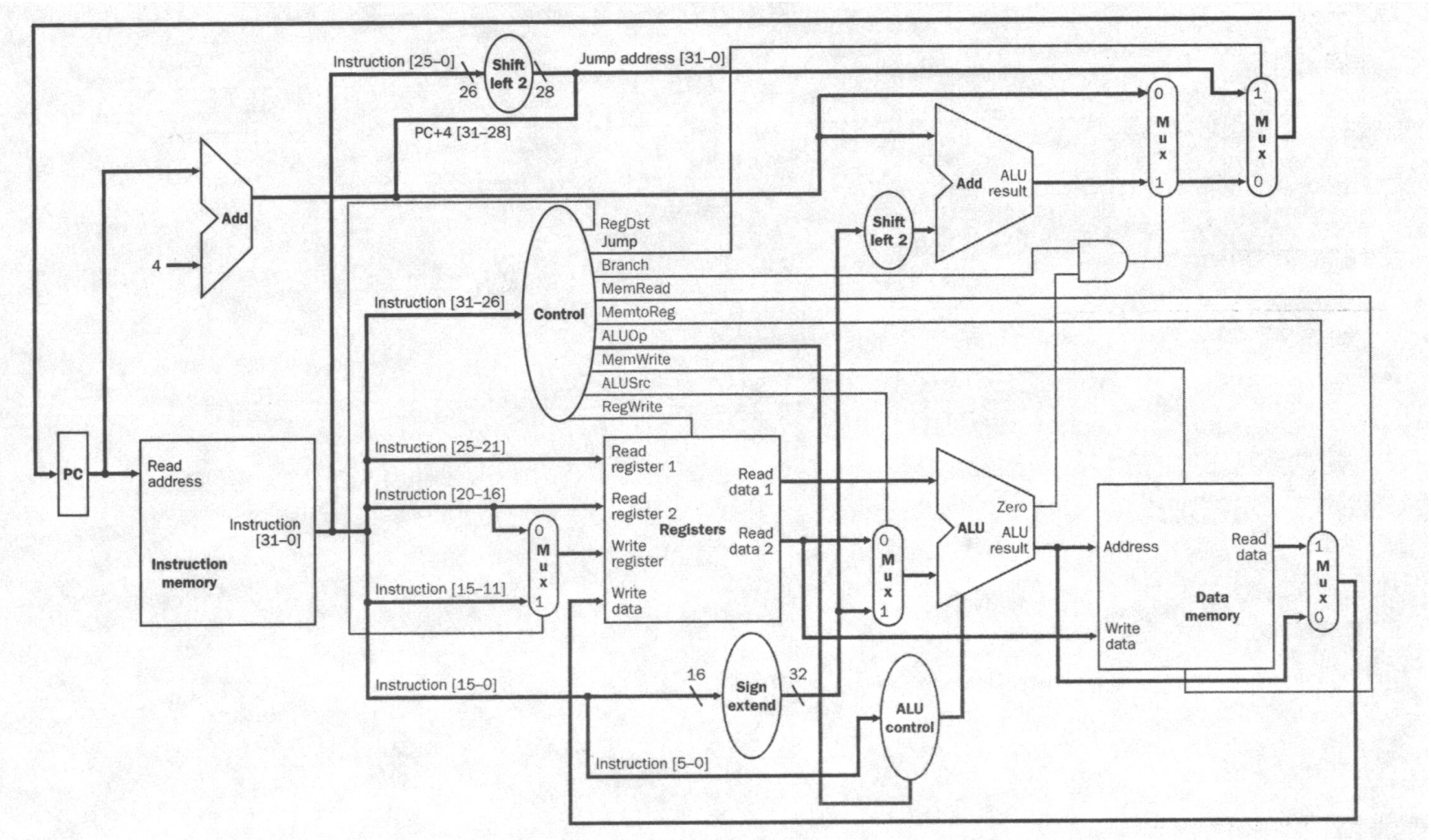


ADD R1, R2, R3
BEQ R1, R2, LABEL
LW R1, LABEL

IR | Opcode | Function

Final Summary Layout

With Jump Instruction



JMP LABEL

Part 2

Example Control Sequence Execution

MIPS 4 Instruction Classes

(add,sub,and, or, slt)

Op-Code

0	rs	rt	rd	shamt	funct
31–26	25–21	20–16	15–11	10–6	5–0

Bit positions

a. R-type instruction

load store

Loc = base + address

35 or 43	rs	rt	address
31–26	25–21	20–16	15–0

Bit positions

b. Load or store instruction

base

43 – source

35 - dest

breq

4	rs	rt	address
31–26	25–21	20–16	15–0

Bit positions

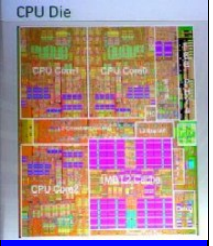
c. Branch instruction

Equal?

Loc = sign ext, shift 2, add to PC

d. Jump Instruction	2	address
	31 – 26	25 - 0

- Notice syntax
- Think about interaction with control unit



Effect on Control Unit

Output Control Lines

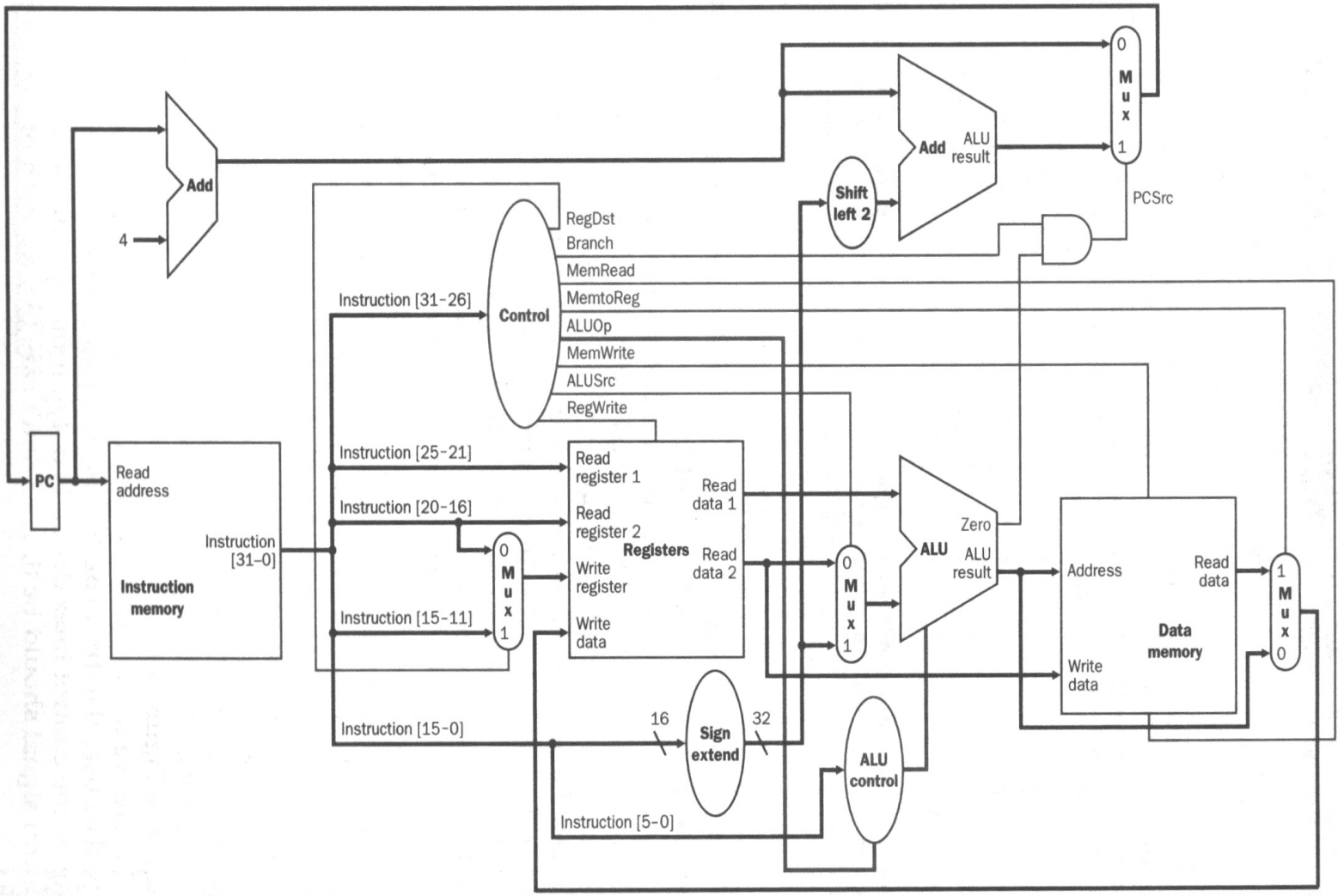
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Input Op-code

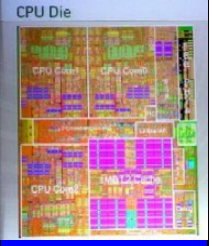
Name	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-format	0_{ten}	0	0	0	0	0	0
lw	35_{ten}	1	0	0	0	1	1
sw	43_{ten}	1	0	1	0	1	1
beq	4_{ten}	0	0	0	1	0	0

R-Type Instruction Processing

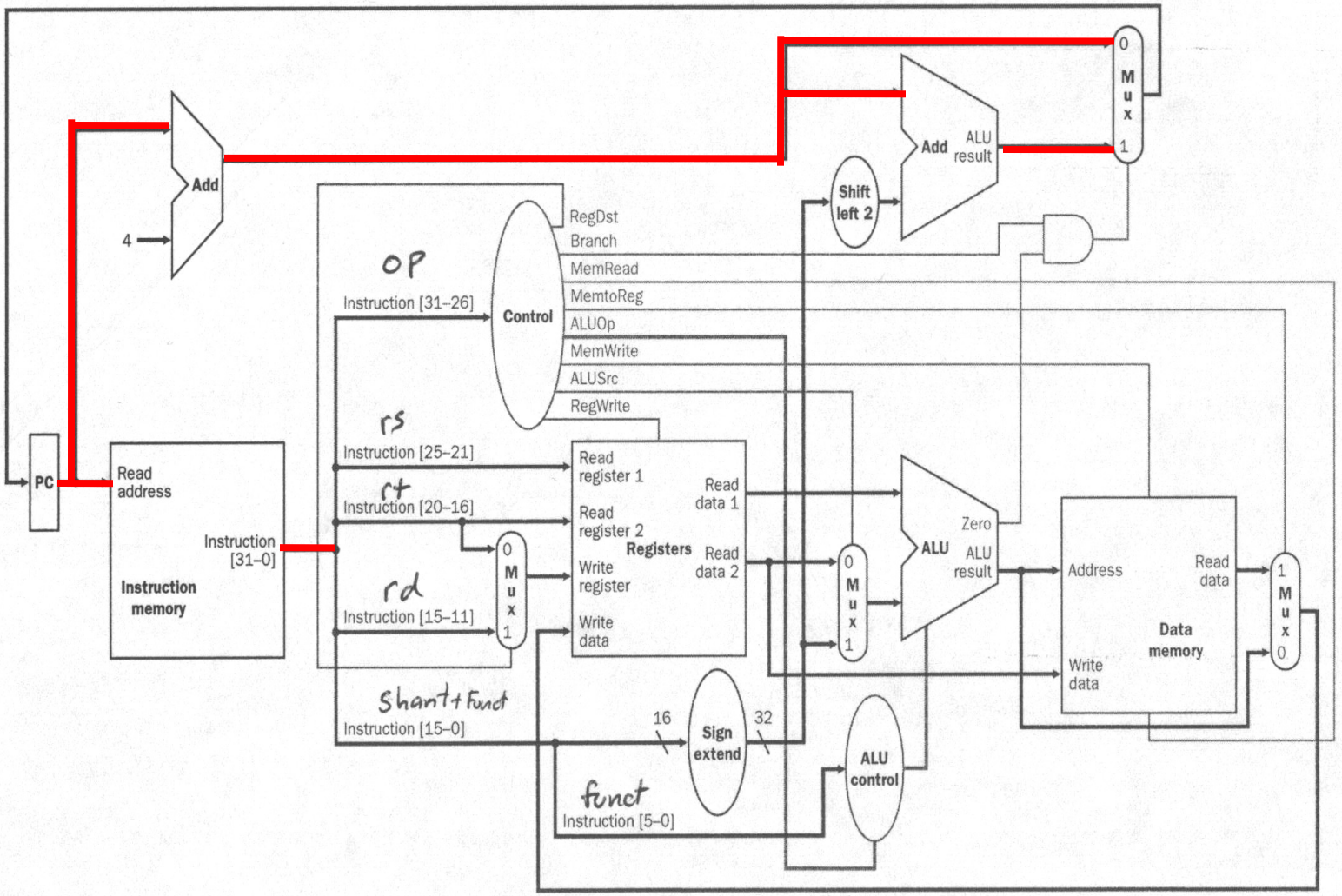
Inactive MIPS CPU



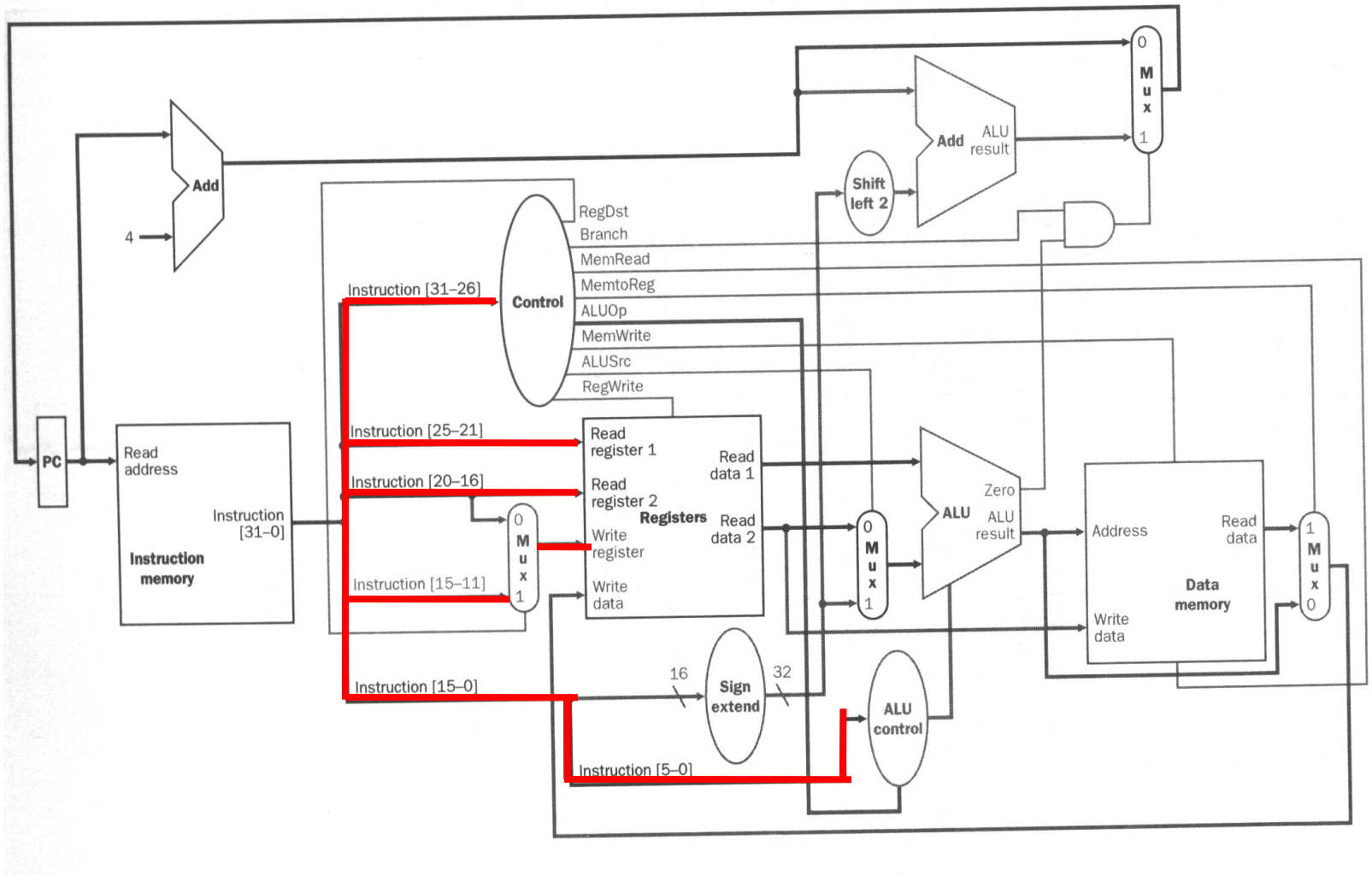
Ordering the control unit (a counter)

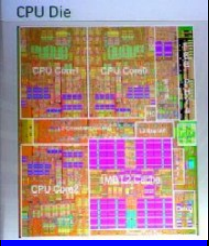


R-Type Fetch Datapath



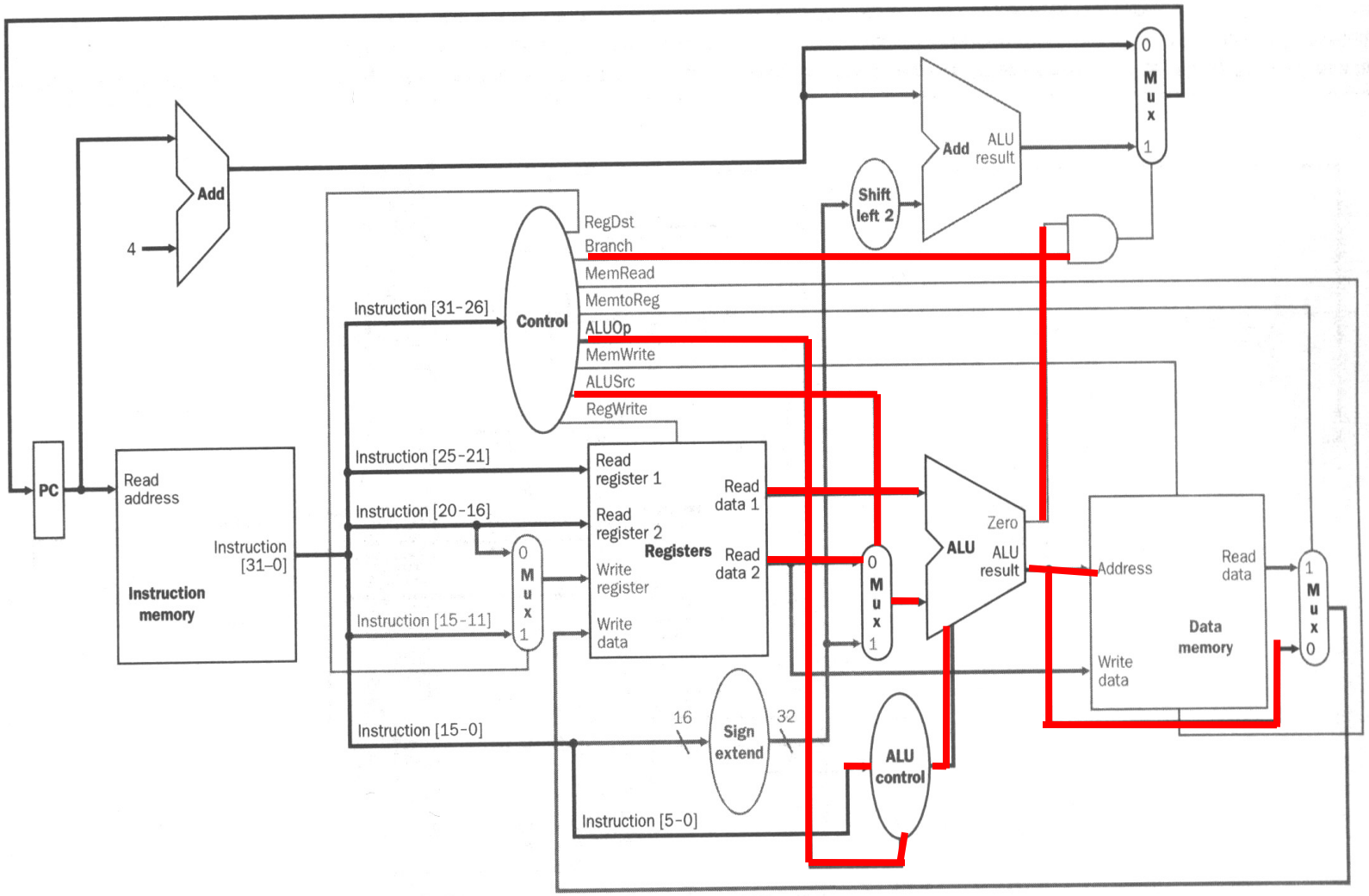
R-Type Load Register Phase

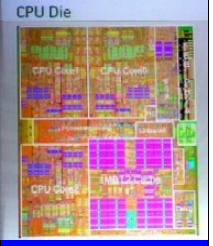




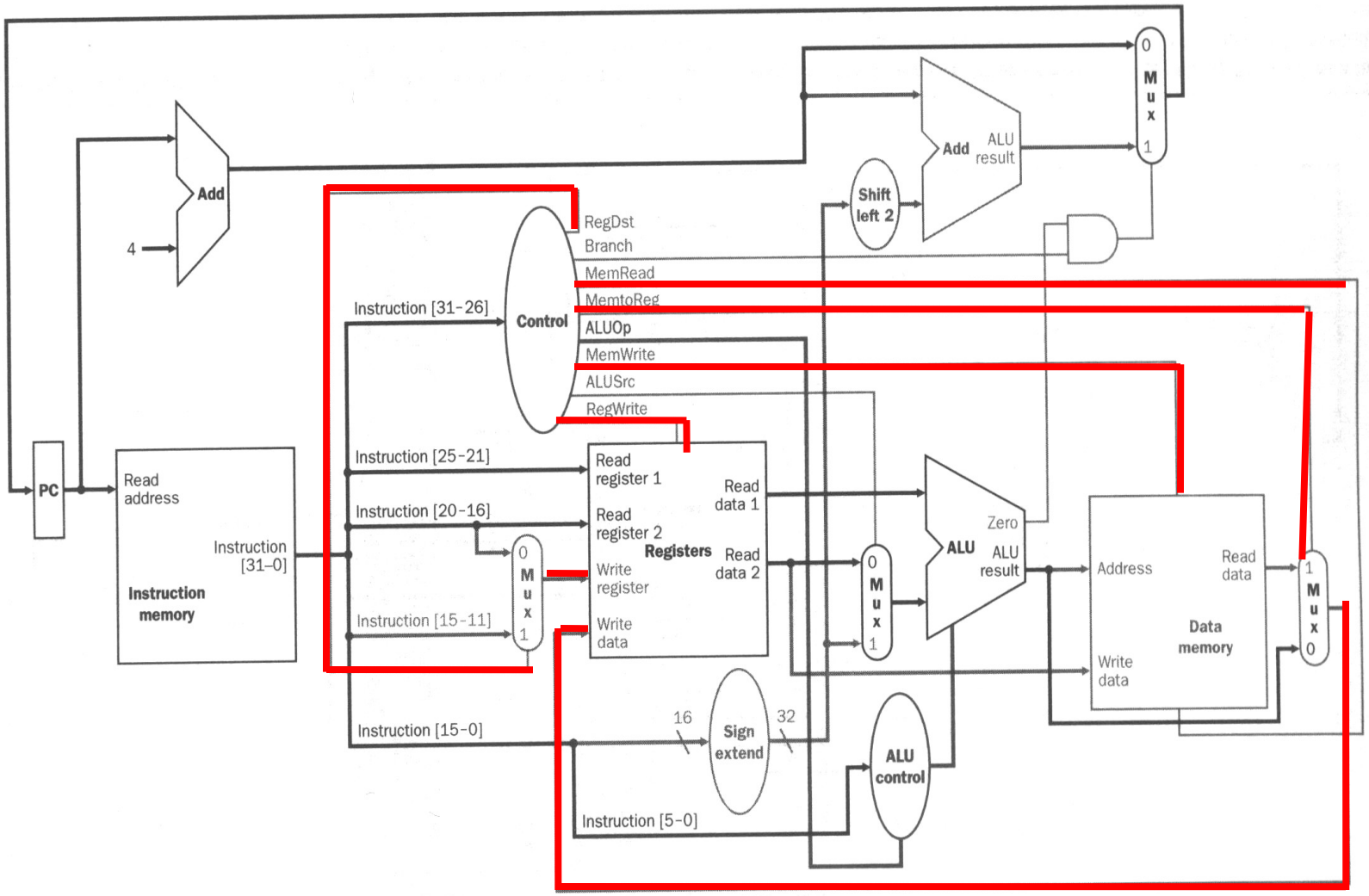
R-Type ALU Phase

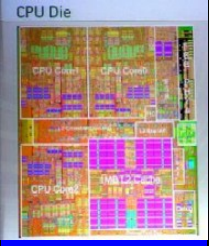
?





R-Type Store Phase





Part 3

The MIPS CPU Statistics

(This is the CPU we will use to program)

R2000 – 210000 MIPS

- Specifications: RISC ~ reduced instruction set
  - 5 stage pipeline
    - Fetch instruction, load registers, ALU, cache, store
  - Cache (n-byte guess load)
  - 32-bit instructions
    - constant size to facilitate pipeline
    - 3 operand instructions
  - 32 general purpose registers
  - Minimum support for:
    - Status code register (uses general purpose register)
    - Stacks (only SP register – Stack Pointer register)
    - Subroutines ($31 register is return address)
    - Interrupts (pre-defined memory location jump)
    - Exceptions (similar facility as with Interrupts)

MIPS Design

- Easy modern architecture to program under
- CPU design philosophies:
  - CISC (Complex Instruction Set Computing)
    - Intel x86 and Motorola 680X0
    - Many powerful instructions (like: load n int to array)
      - Single instruction that can do many things
      - Instruction power = m clock ticks, s.t. m very large
      - None or minimum pipeline capabilities (complexity)
  - RISC (Reduced Instruction Set Computing)
    - MIPS (R2000, R10000), (PDP11)
    - Few and simple instructions (like: load R1 with 5)
      - Instruction = n clock ticks, s.t. n is = 1 or close to that
      - Pipelines exist and can be optimized
- MIPS uses the RISC philosophy

Functional Units of MIPS CPU

- Two internal CPU constructions:
  - Combinational Constructions
    - Given the same input produces the same output
    - For example the ALU
  - State Constructions
    - For example the Cache or a Register
    - Their contents depend on what is currently there
- Therefore, two kinds of instructions:
  - Combinational (like add, subtract)
  - State (like move, load)
- Pipeline optimized architecture