CMPE 200

COMPUTER ARCHITECTURE

Lecture 3 – The Processor – Single Clock

Bo Yu
Computer Engineering Department
SJSU

Adapted from Computer Organization and Design, 5th Edition, 4th edition, Patterson and Hennessy, MK and Computer Architecture – A Quantitative Approach, 4th edition, Patterson and Hennessy, MK

Lecture 2 Key Concepts Review

Last Lecture

- **ISA - Language of the Computer:**
  - **Instructions: Language of the computer**
  - **Different computer architectures**
  - **MIPS Architecture**

Lecture 2 Key Concepts Review

- MIPS Instruction Set
  - o a RISC instruction set architecture
  - o Interface between software and its hardware
  - o a computer's assembly language
  - o Current version MIPS32/64 release 6, MIPS32 is used as example throughout the book
  - o Instructions are encoded in binary (machine code)
- MIPS has a 32 x 32-bit register file (GPR)
  - o Use for frequently accessed data, faster than memory
  - o Numbered 0 to 31
  - o 32-bit data called a "word"
  - o MIPS instructions are encoded as 32-bit instruction words, all instructions a single size
  - o 3 types of MIPS instruction set format: R-type, I-type, J-type
  - o MIPS is big endian (most significant byte at least address of a word
- Memory Operands
  - o Main memory: Arrays, structures, dynamic data
  - o Load (memory to register), store (register to memory)
  - o Memory is byte (8-bit) addressed, address must be a multiple of 4 for words aligned in memory
- Binary signed and unsigned number
  - o Two's complement representation: leading 0s mean positive, leading 1s mean negative
  - o 32bits (MSB-sign bit): $(d31x-2^{31})+(d30x2^{30})+(d29x2^{29})+\ldots+(d1x2^{1})+(d0x2^{0})$
- Design Principle 1: Simplicity favors regularity
  - o Regularity makes implementation simpler
  - o Simplicity enables higher performance at lower cost
- Design Principle 2: Smaller is faster
  - o c.f. main memory: millions of locations
- Design Principle 3: Good design demands good compromises
  - o Keep all instructions the same length, different formats for different instructions
- Design Principle 4: Making the common case fast

Lecture 2 Key Concepts Review

- MIPS 32 General Purpose Registers (GPR)

Register Number	Conventional Name	Usage
$0	$zero	Hard-wired 0
$1	$at	Reserved by assembler to handle large constants
$2-$3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions, not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary data, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved by kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

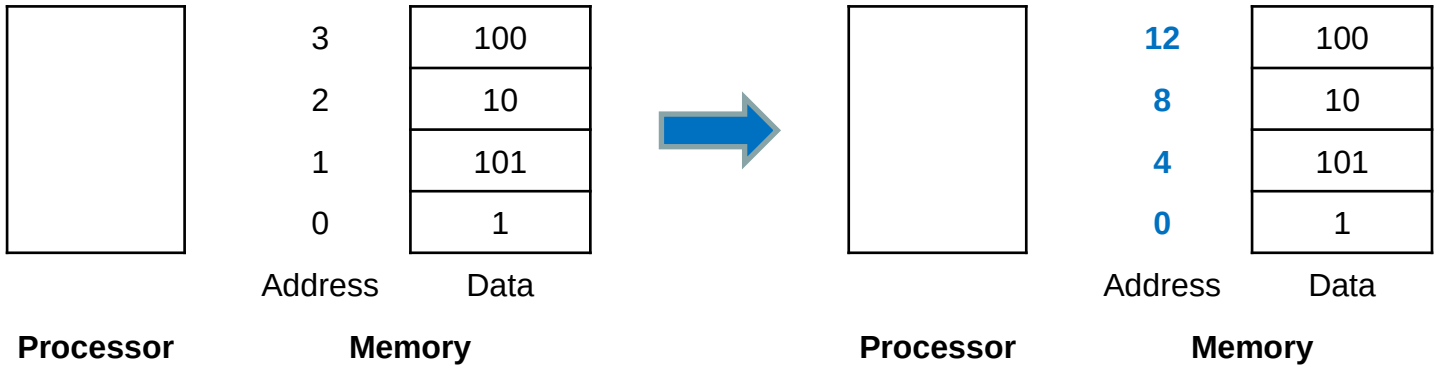
- MIPS Instruction Format

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

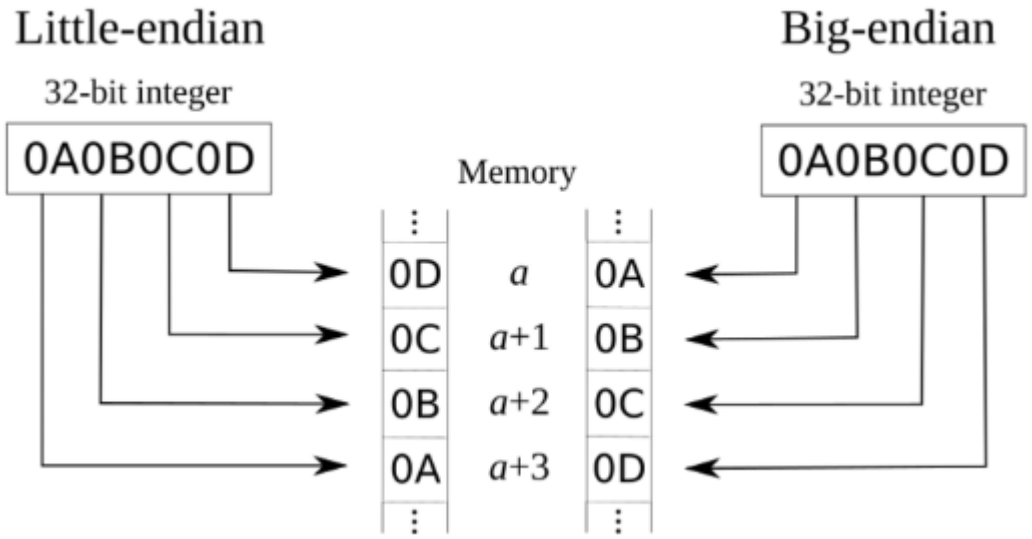
- MIPS Instruction Classes and Assembly Language

Instruction class	MIPS examples	HLL correspondence
Arithmetic	add, sub, addi	Arithmetic operations
Data transfer	lw, sw, lh, lhu, sh, lb, lbu, sb	References to data structure, etc.
Logical	And, or, nor, andi, ori, sll, srl	Logical operations
Conditional branch	Beq, bne, slt, sltu, slti, sltiu	If statements and loops
Unconditional jump	J, jr, jal	Procedure calls, returns, case/switch statements

Word address must be a multiple of 4



Big Endian vs. Little Endian



Lecture 2 Key Concepts Review

- MIPS instruction encoding

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

- MIPS machine language

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Lecture 2 Key Concepts Review

- MIPS R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

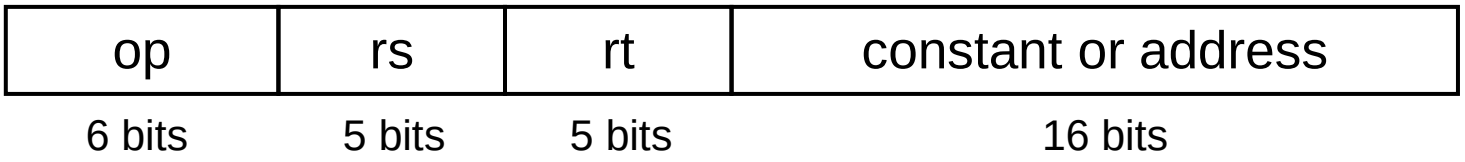
```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

Lecture 2 Key Concepts Review

- MIPS I-format Instructions



  - Immediate arithmetic and load/store instructions
    - rt: destination or source register number
    - Constant: -2^{15} to $+2^{15} - 1$
    - Address: offset added to base address in rs
- Example of translating MIPS Assembly Language into Machine Language:

  A[300] = h + A[300]; $t1 has base address of array A, $s2 corresponds to h

  The above assignment statement is compiled into:

Assembly Language ⇨

```
lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

Machine Language Instructions ⇨

Op	rs	rt	rd	address/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Machine Binary Code ⇨

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Binary signed and unsigned number

- Representation of Signed numbers
  - Sign and magnitude: Add a separate sign represented in a single bit 
  - Two's Complement Representation (every computer uses today)
    - leading 0s mean positive, leading 1s mean negative
  - 32bits (MSB-sign bit): $(d31x\text{-}2^{31})+(d30x2^{30})+(d29x2^{29})+\ldots+(d1x2^{1})+(d0x2^{0})$
- Signed numbers

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
……
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten} = (2^{31}\text{-}3)$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten} = (2^{31}\text{-}2)$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten} = (2^{31}\text{-}1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten} = (-2^{31})$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten} = (2^{31}+1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten} = (2^{31}+2)$
……
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

- Signed Negation
  - Complement and add 1
    - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

  - Example: negate +2
    - +2 = 0000 0000 … 0010_2
    - –2 = 1111 1111 … 1101_2 + 1
      = 1111 1111 … 1110_2

- Hexadecimal
  - Base 16
    - Compact representation of bit strings
    - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

  - Example: eca8 6420
    - 1110 1100 1010 1000 0110 0100 0010 0000

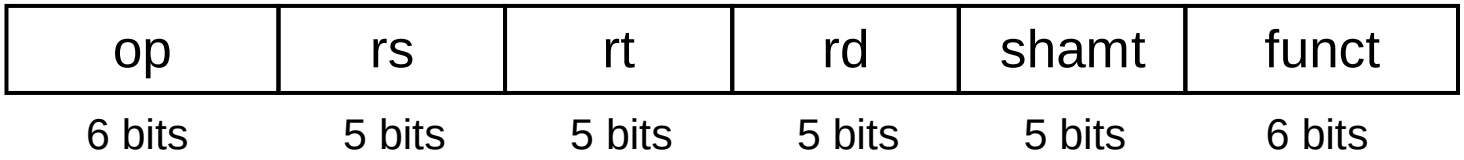
Lecture 2 Key Concepts Review

- Logical Operations
  - Instructions for bitwise manipulation
  - Useful for extracting and inserting groups of bits in a word

Logical Operations	C Operators	Java Operators	MIPS Instructions
Shift left logical	<<	<<	sll
Shift right logical	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOR	~	~	nor

Lecture 2 Key Concepts Review

- Shift Operations



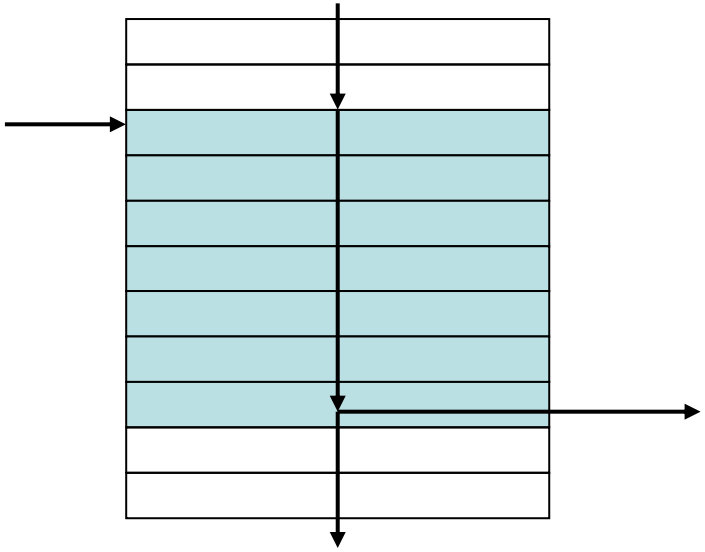
  - shamt: how many positions to shift
  - Shift left logical
    - Shift left and fill with 0 bits
    - sll by i bits multiplies by 2^i
  - Shift right logical
    - Shift right and fill with 0 bits
    - srl by i bits divides by 2^i (unsigned only)

Lecture 2 Key Concepts Review

- Conditional Operations
  - Branch to a labeled instruction if a condition is true
    - Otherwise, continue sequentially
  - beq rs, rt, L1
    - if (rs == rt) branch to instruction labeled L1;
  - bne rs, rt, L1
    - if (rs != rt) branch to instruction labeled L1;
  - j L1
    - unconditional jump to instruction labeled L1

Lecture 2 Key Concepts Review

- Basic Blocks
  - C code:
  - A basic block is a sequence of instructions with
    - No embedded branches (except at end)
    - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Lecture 2 Key Concepts Review

- More Conditional Operations
  - Set result to 1 if a condition is true
    - Otherwise, set to 0
  - slt rd, rs, rt
    - if (rs < rt) rd = 1; else rd = 0;
  - slti rt, rs, constant
    - if (rs < constant) rt = 1; else rt = 0;
  - Use in combination with beq, bne

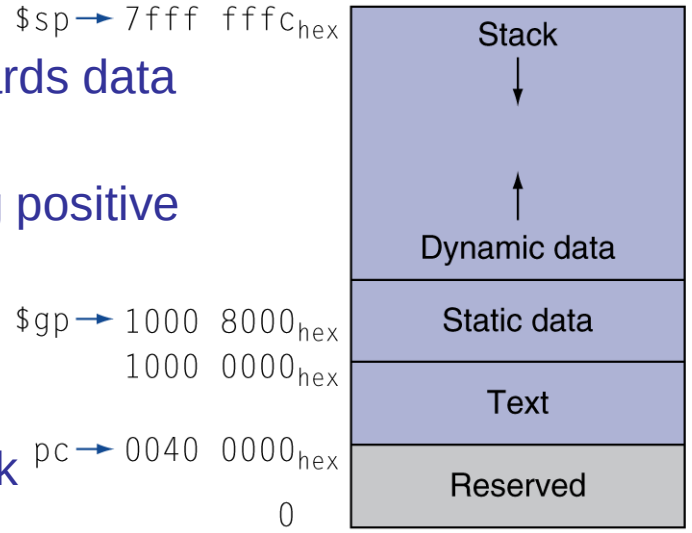
```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #   branch to L
```

Lecture 2 Key Concepts Review

- Signed vs. Unsigned Comparison
  - Signed comparison: slt, slti
  - Unsigned comparison: sltu, sltui
  - Example
    - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
    - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
    - slt $t0, $s0, $s1 # signed
      - –1 < +1 ⇒ $t0 = 1
    - sltu $t0, $s0, $s1 # unsigned
      - +4,294,967,295 > +1 ⇒ $t0 = 0

Lecture 2 Key Concepts Review

- Memory Layout
  - Text: program code
  - Static data: global variables
    - e.g., static variables in C, constant arrays and strings
    - $gp initialized to address allowing ±offsets into this segment
    - $sp → starts at 7fff fffc, grows down towards data segment
    - $gp → starts at 1000 8000, moves using positive and negative 16-bit offset
    - Text Segment: MIPS machine code
    - Static data: starts at 1000 0000
    - Dynamic data: heap grows towards stack
  - Dynamic data: heap
    - E.g., malloc in C, new in Java
  - Stack: automatic storage



Lecture 2 Key Concepts Review

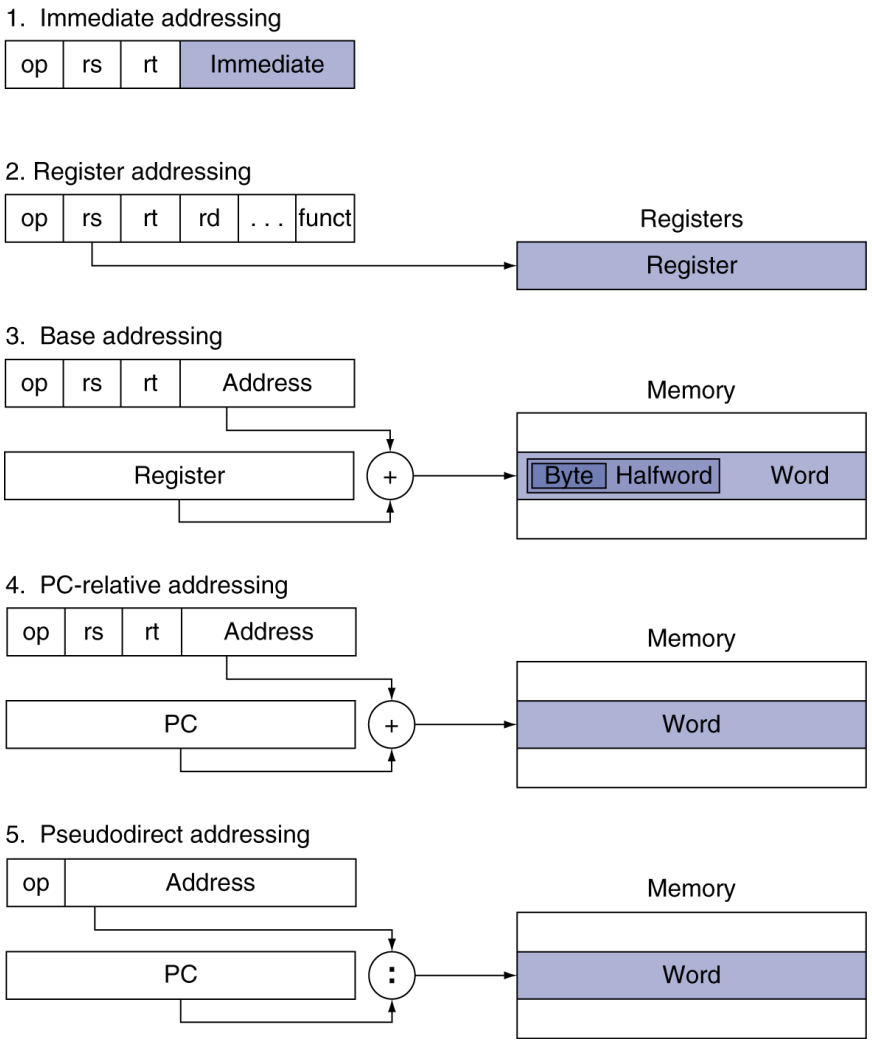
- Target Addressing Example
  - Loop code from earlier example
    - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

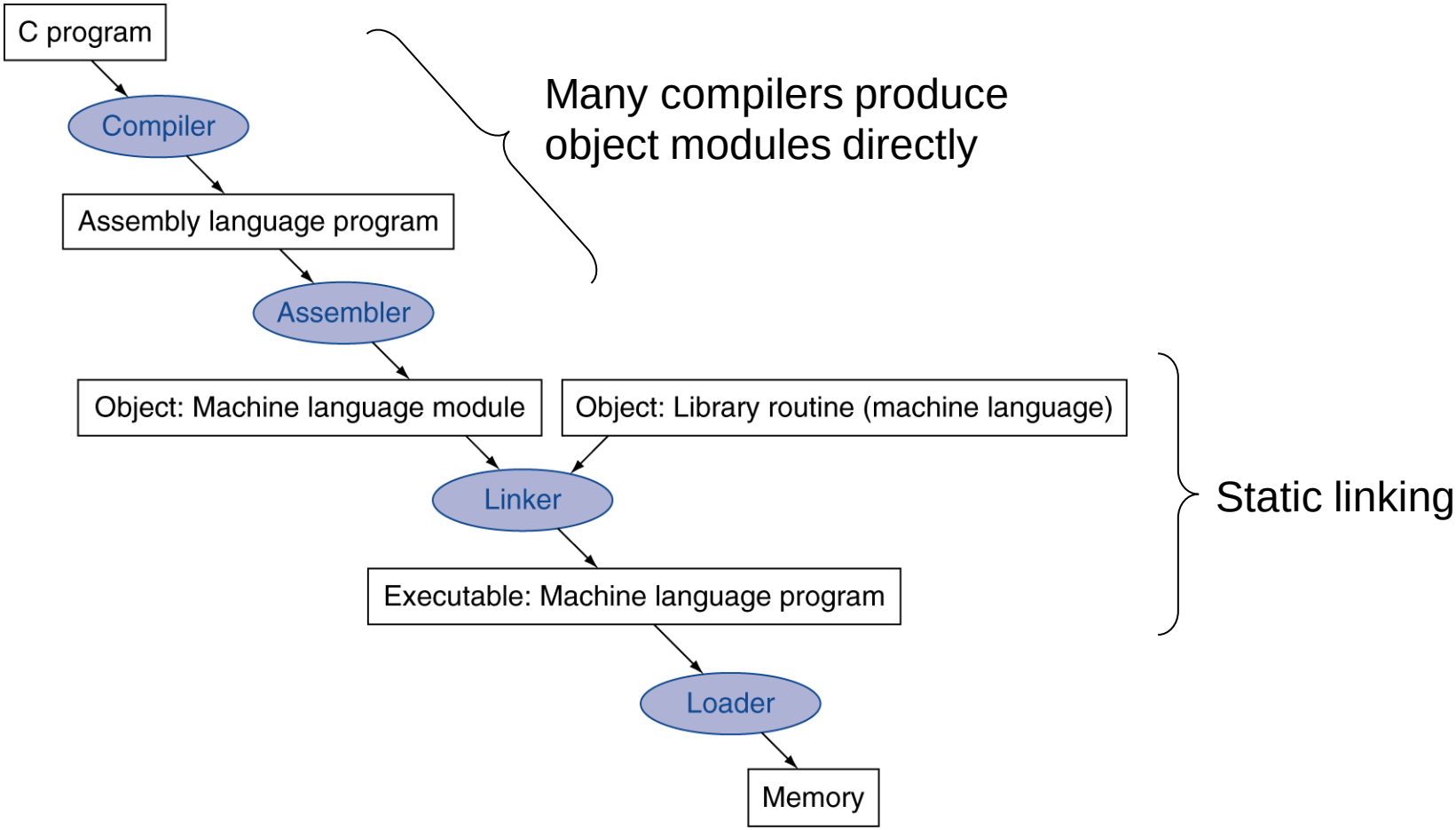
80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Lecture 2 Key Concepts Review

Addressing Mode Summary



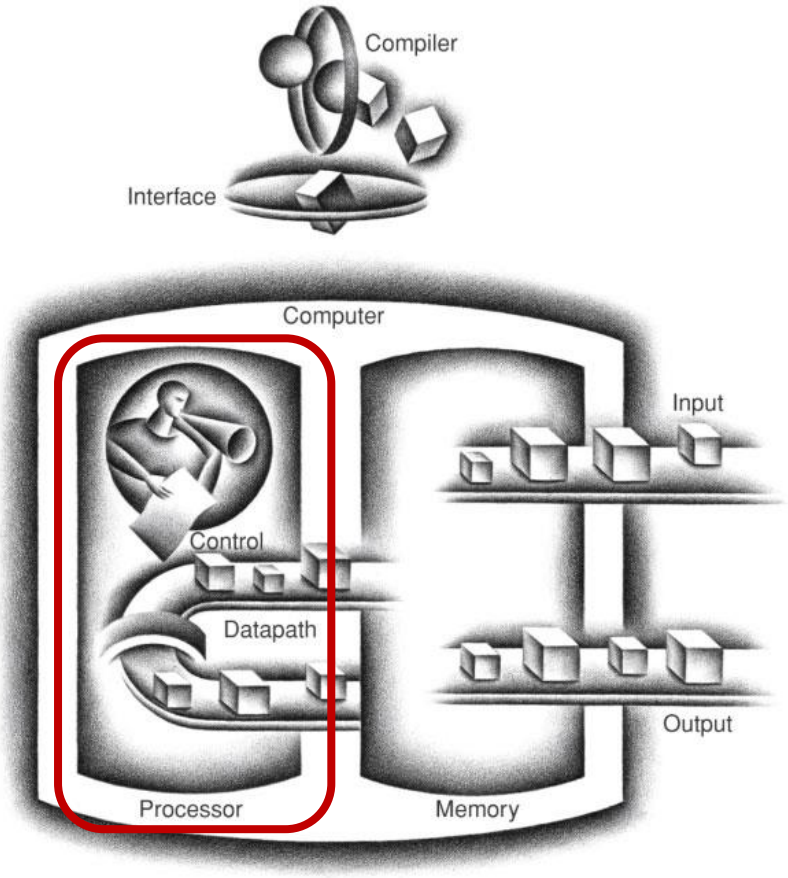
- Four steps transforming a C program into a program running on a computer



Today's Lecture (CH4.1-4.4):

- Datapath Design
  - **Single clock**
  - Pipelining - later
  - Hazards - later
  - Parallelism – ILP - later

- Control Unit Design
  - **Control Unit**
  - **Jumps**
  - CALL/RETURN - later
  - Exceptions - later



Two MIPS Implementations

- CPU performance factors
  - Instruction count: Determined by ISA and compiler
  - CPI and Cycle time: Determined by CPU hardware

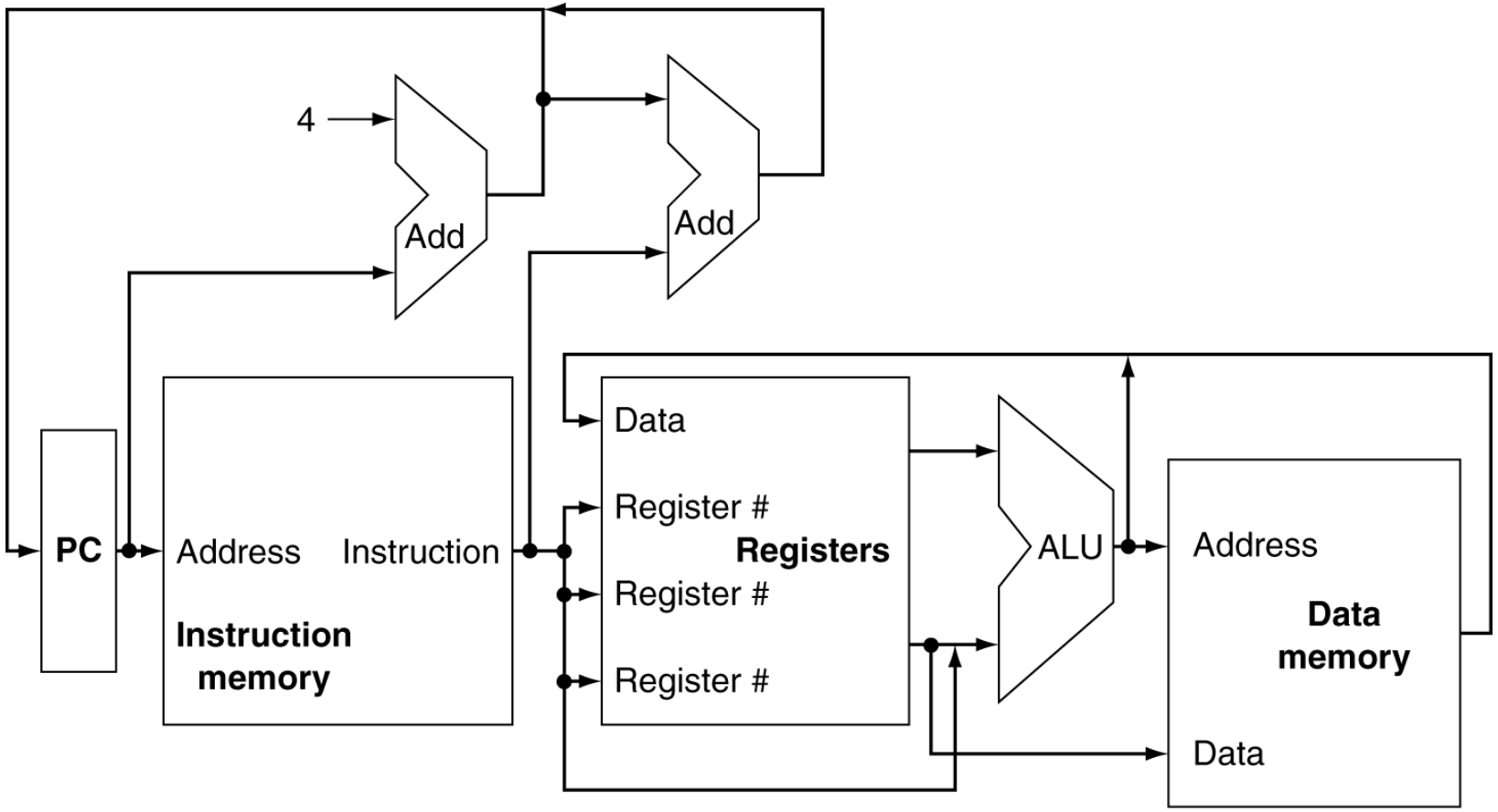
- Simple MIPS Implementation
  - Single Clock Implementation
  - Fixed size instructions
  - Uses a subset of core MIPS instruction set
    - Memory reference: lw, sw
    - Arithmetic/logical: add, sub, and, or, slt
    - Control transfer: beq, j

- Pipelined MIPS Implementation

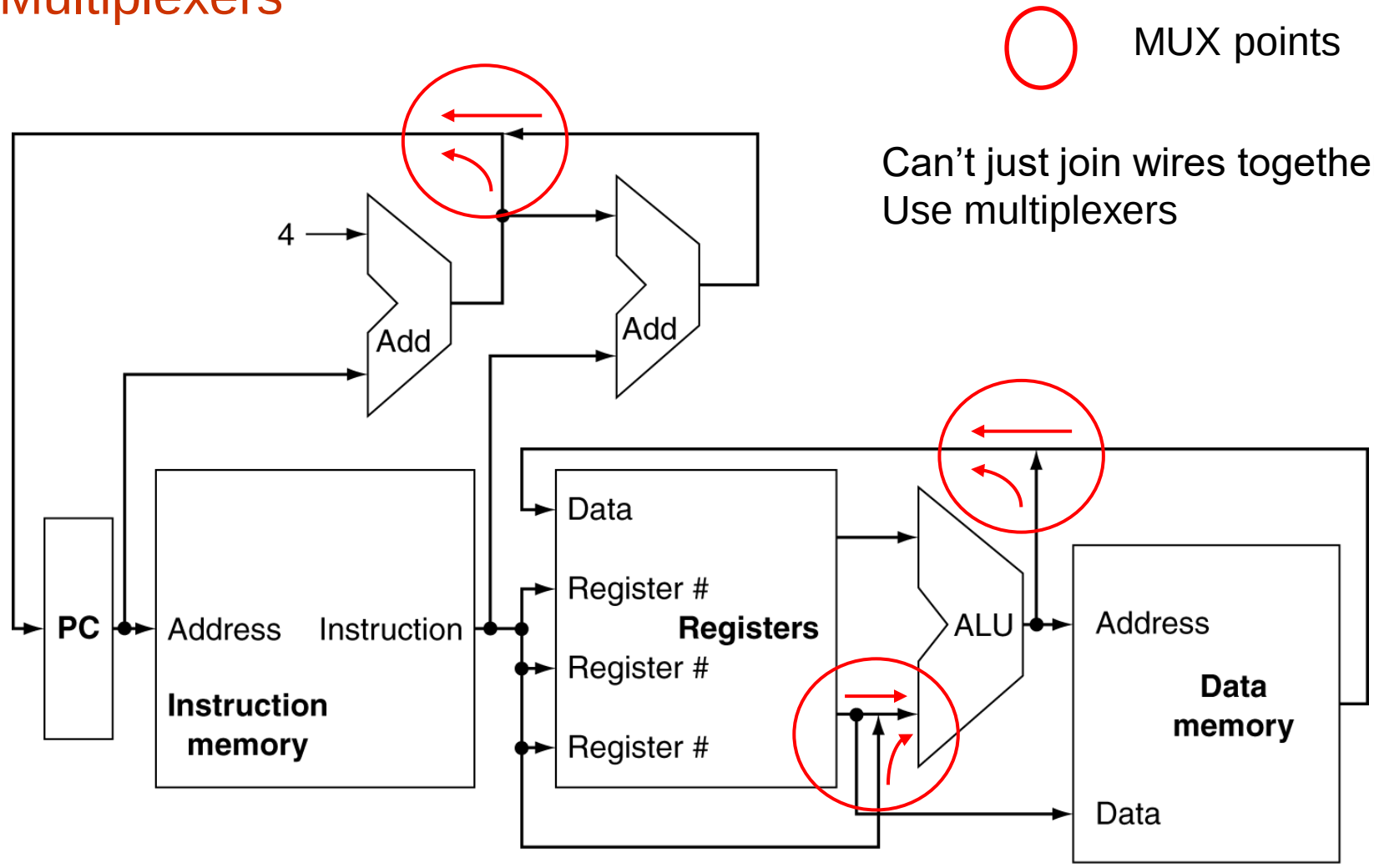
Instruction Execution

- Instruction Fetch
  - PC => instruction memory => Instruction register
- Operand Read
  - Register Numbers => Register File
  - Read Registers
- Execute
  - ALU to calculate
    - Results for arithmetic operations
    - Data memory address for Load/Store
    - Branch target address (alternative implementation?)
- Access data memory for load/store
- Next Instruction
  - PC + target address for jumps or PC + 4

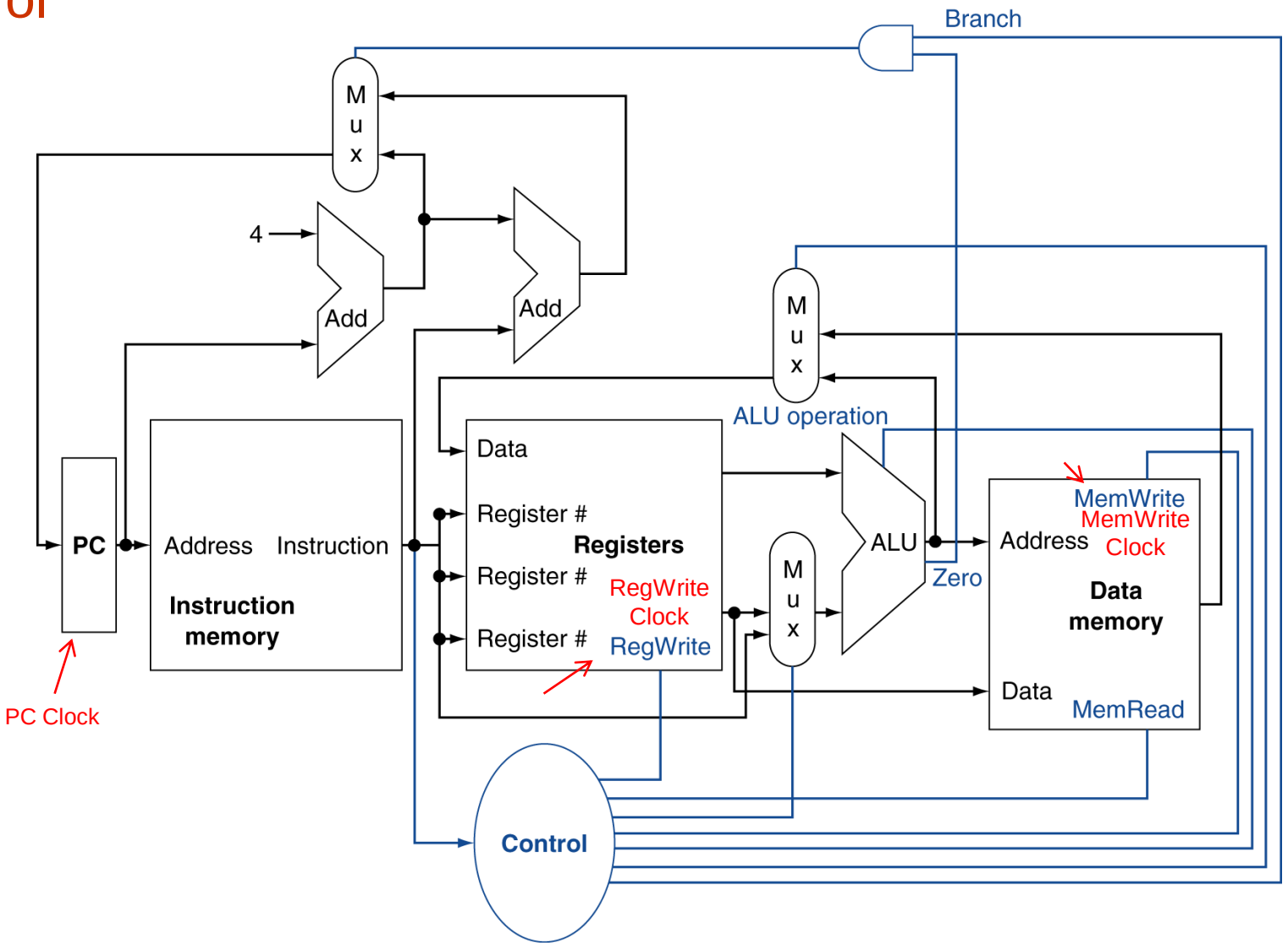
CPU Overview



Multiplexers



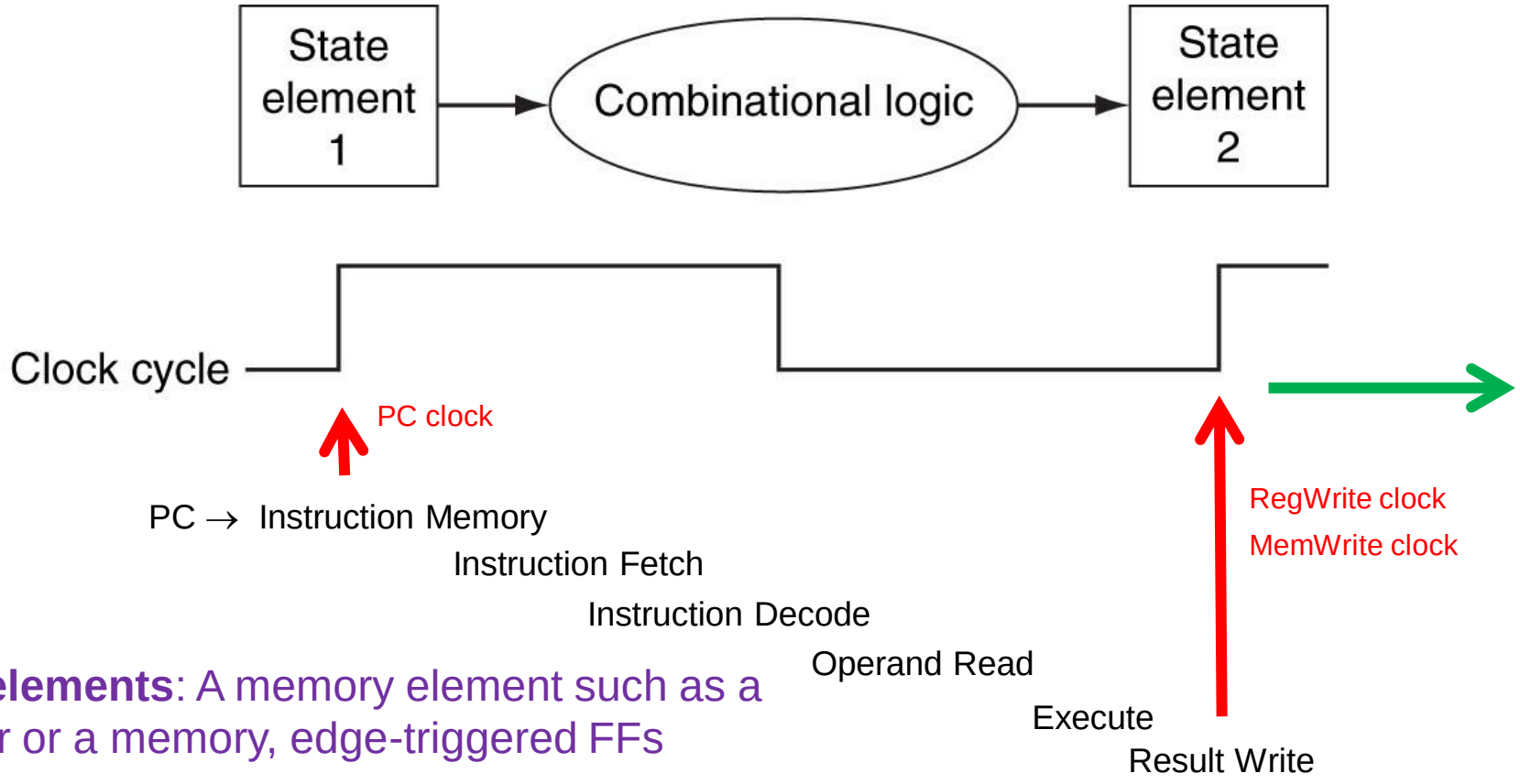
- Control


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
Clock
RegWrite
M
u
x
ALU
Zero
MemWrite
MemWrite
Clock
Address
Data memory
Data
MemRead
PC Clock
Control

How it works – Single-Clock System

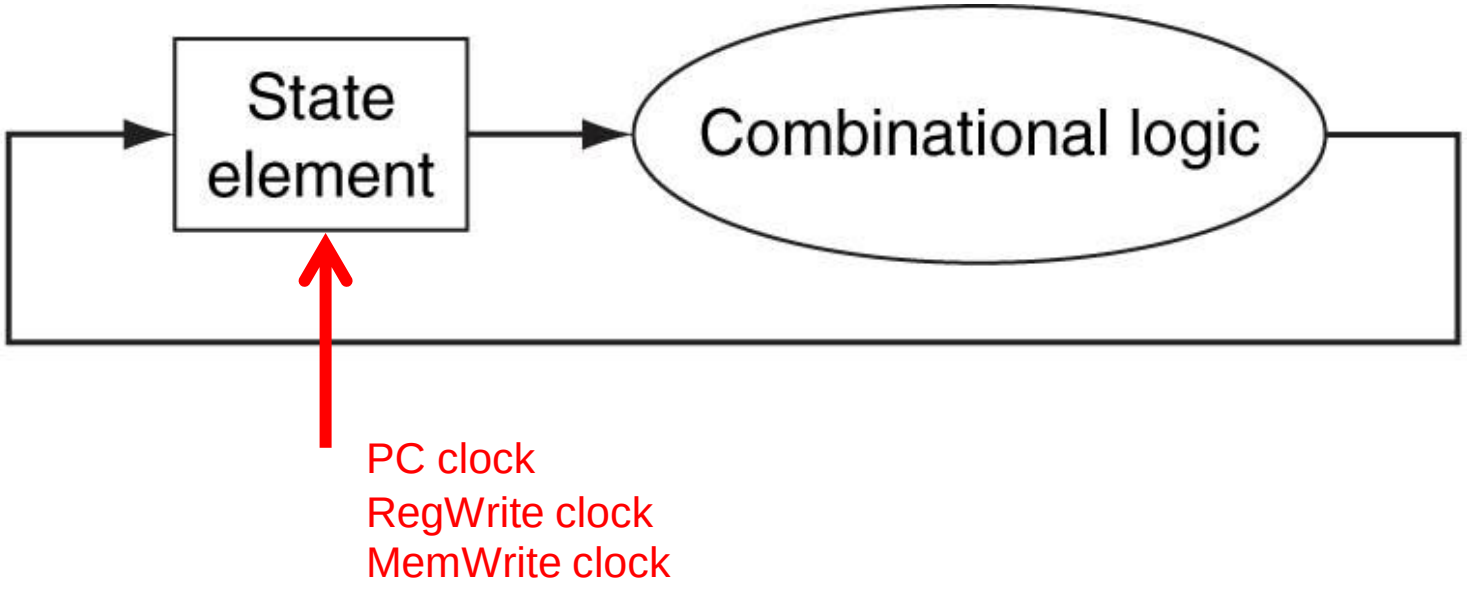
- **Clocking**

Combinational logic: An operational element such as an AND gate or an ALU



State elements: A memory element such as a register or a memory, edge-triggered FFs

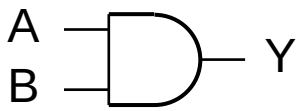
- How it works – Single-Clock System
  - **Clocking – continuous operation**
  - **Combinational logic transforms data during clock cycles**
    - Between clock edges
    - Input from state elements, output to state element
    - Longest delay determines clock period



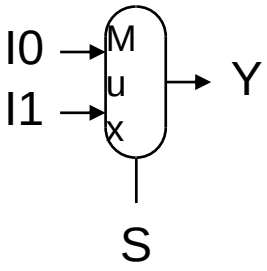
Logic Design Basics

- **Information encoded in binary**
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- **Combinational element**
  - Operate on data
  - Output is a function of input
- **State (sequential) elements**
  - Store information

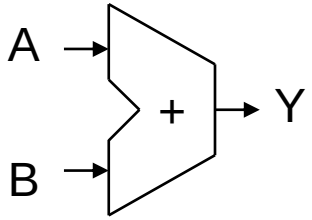
- Combinational Elements
  - AND-gate
    - Y = A & B



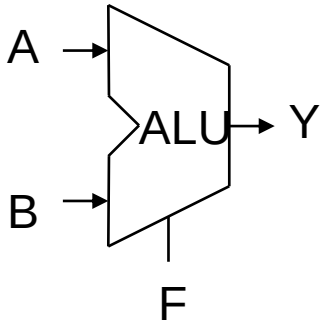
  - Multiplexer
    - Y = S ? I1 : I0



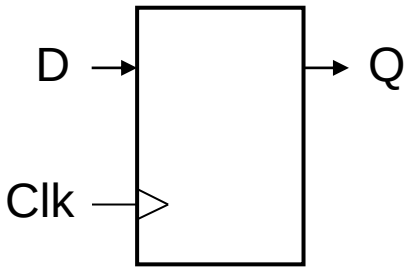
  - Adder
    - Y = A + B

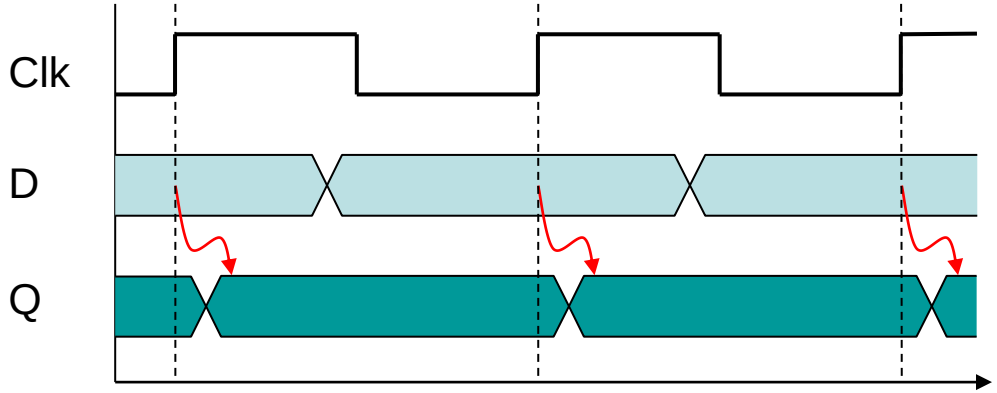


  - Arithmetic/Logic Unit
    - Y = F(A, B)

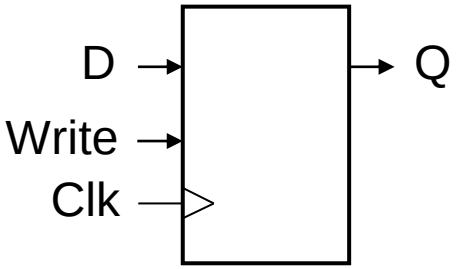


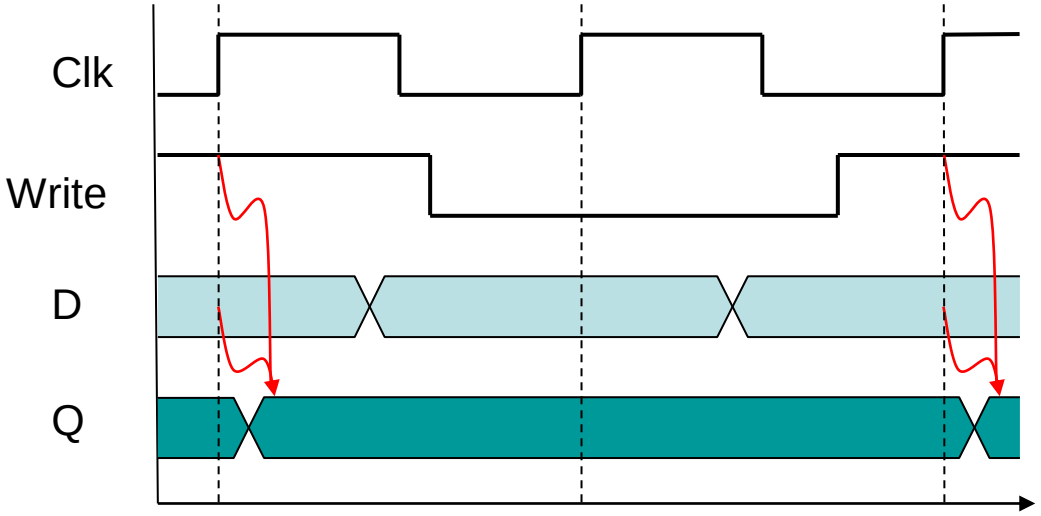
- Sequential Elements
  - **Register: stores data in a circuit**
    - Uses a clock signal to determine when to update the stored value
    - Edge-triggered: update when Clk changes from 0 to 1





- Sequential Elements
  - **Register with write control**
    - Only updates on clock edge when write control input is 1
    - Used when stored value is required later



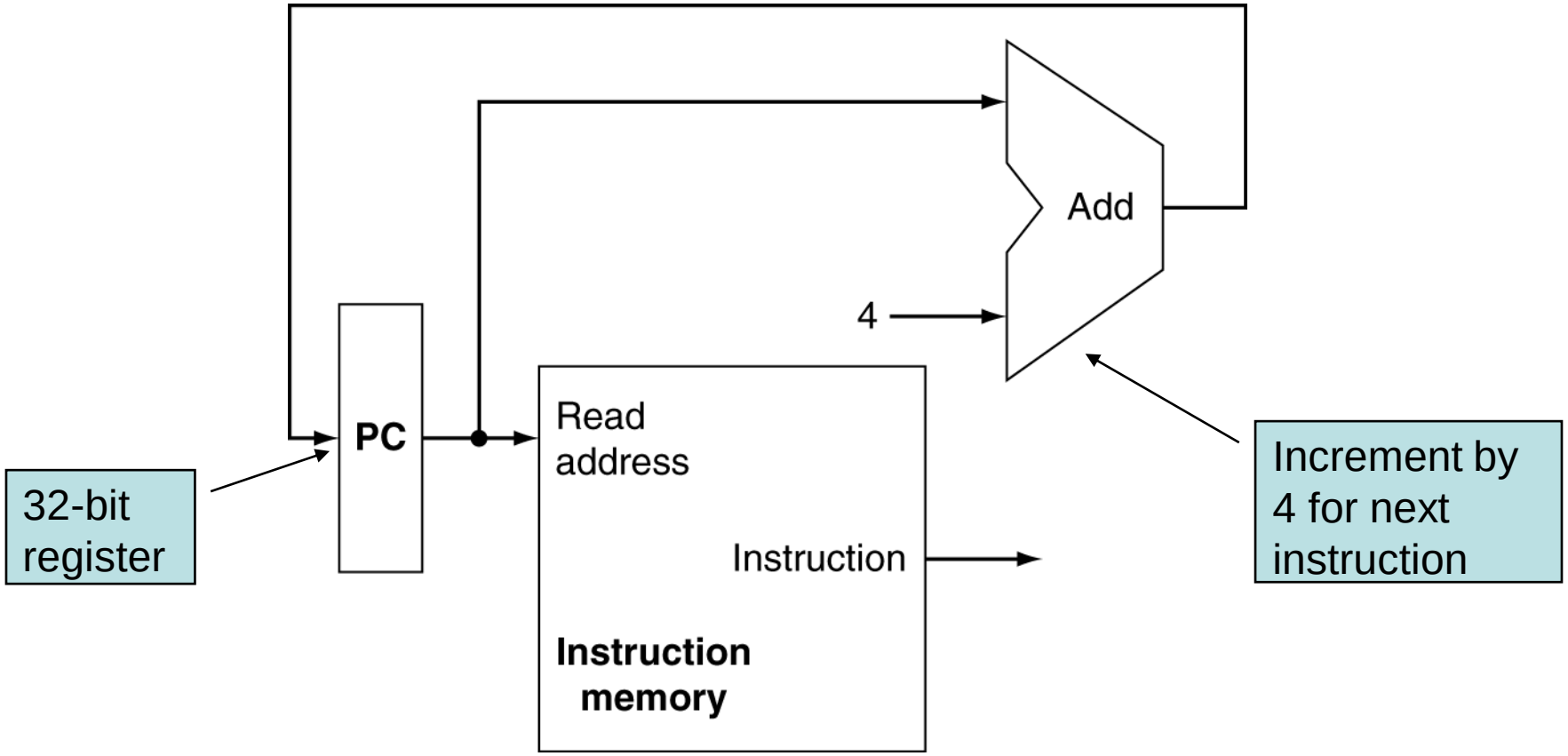


Building a Datapath

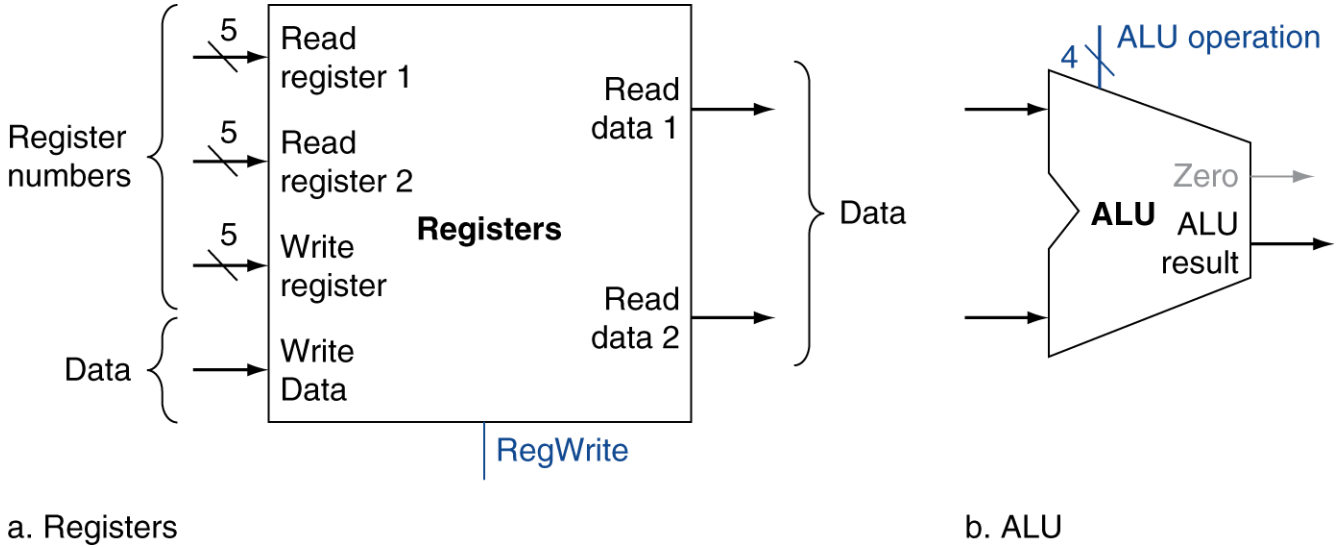
- **Datapath**
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …

- **We will build a MIPS datapath incrementally**
  - Refining the overview design

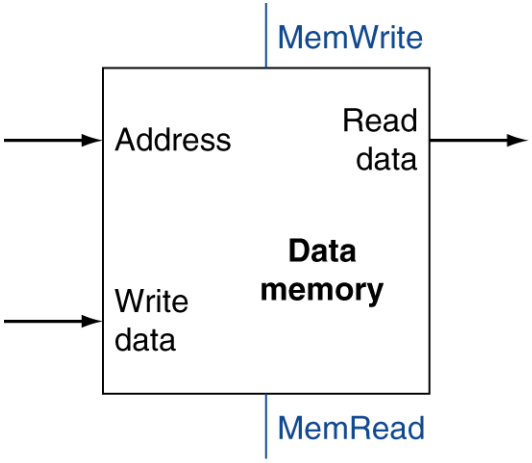
- Data Path
  - **Instruction Fetch**



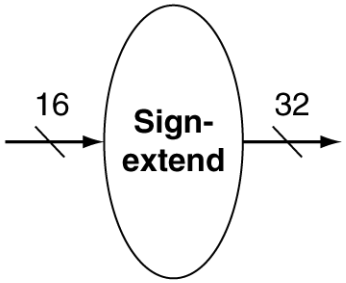
- R-Format Instructions
  - **Read two register operands**
  - **Perform arithmetic/logical operation**
  - **Write register result**



- Load/Store (I-Format) Instructions
  - **Read register operands**
  - **Calculate address using 16-bit offset**
    - **Use ALU, but sign-extend offset**
  - **Load: Read memory and update register**
  - **Store: Write register value to memory**



a. Data memory unit



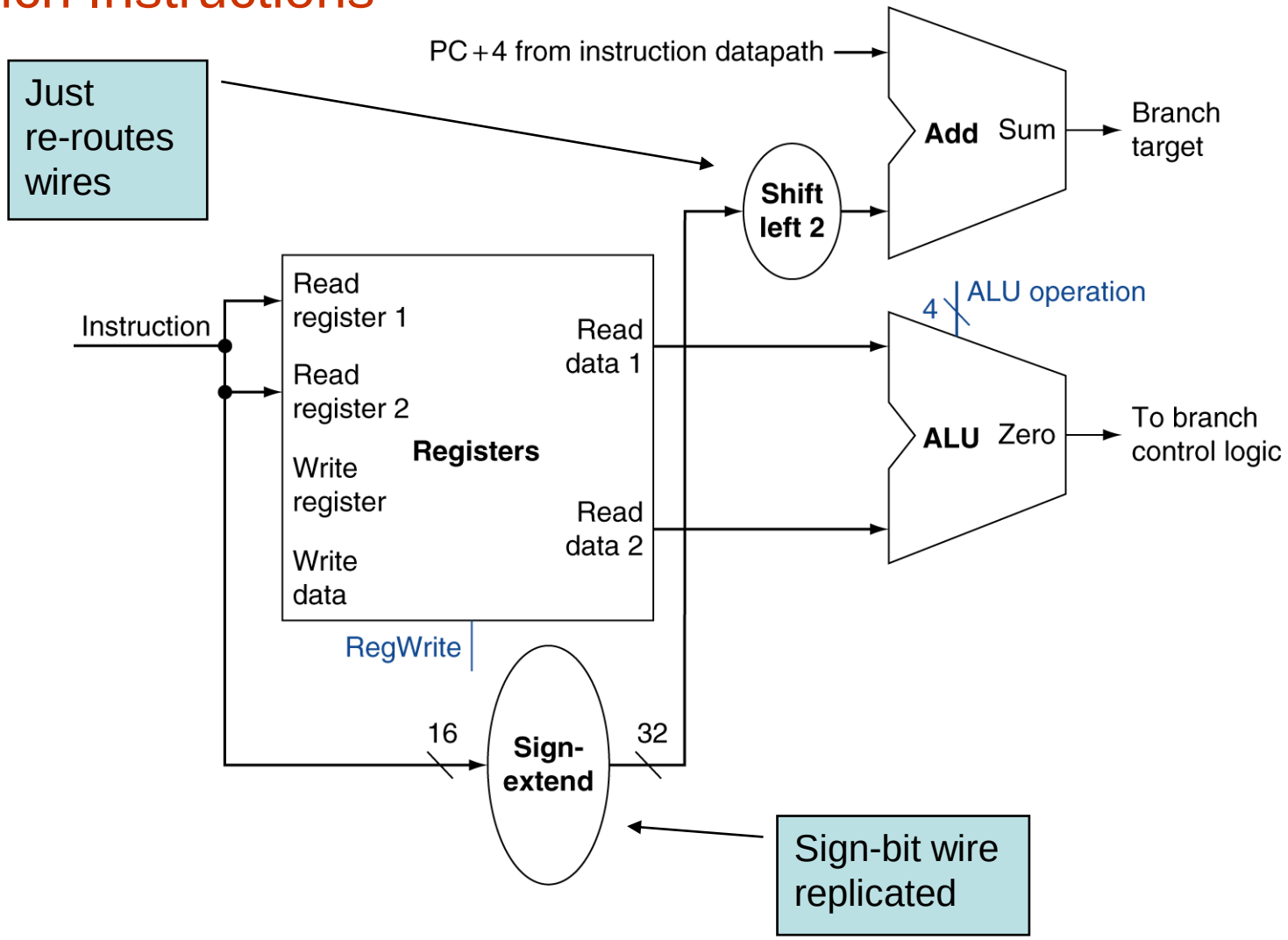
b. Sign extension unit

- Branch Instructions
  - **Read register operands**
  - **Compare operands**
    - **Perform ALU op as required for condition testing**
    - **Use ALU, subtract and check Zero output**
  - **Calculate target address based on the condition**
    - **Sign-extend displacement**
    - **Shift left 2 places (word displacement)**
    - **Add to PC + 4**
      - ❖Already calculated by instruction fetch

Examples (with RTL descriptions):

BEQ If [R(rs)==R(rt) then PC ← PC + signExt(Imm16) || 00
else PC ← PC + 4

- Branch Instructions



- Composing the Elements
  - **First-cut data path does an instruction in one clock cycle**
    - **Each datapath element can only do one function at a time**
    - **Hence, we need separate instruction and data memories**
  - **Use multiplexers where alternate data sources are used for different instructions**

R-Type and I-Type Datapath

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

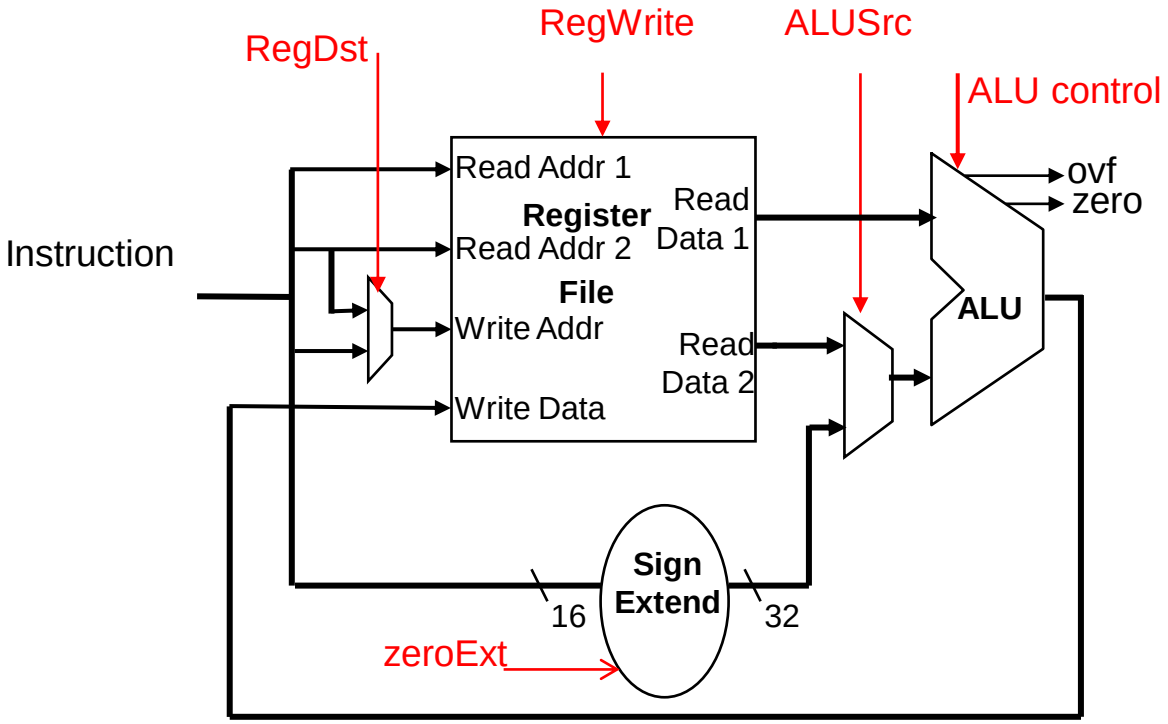
Examples (with RTL description):

ADDU R(rd) ← R(rs) + R(rt)

ADDI R(rt) ← R(rs) + signExt(Imm16)

OR R(rd) ← R(rs) .OR. R(rt)

ORi R(rt) ← R(rs) + zeroExt(Imm16)



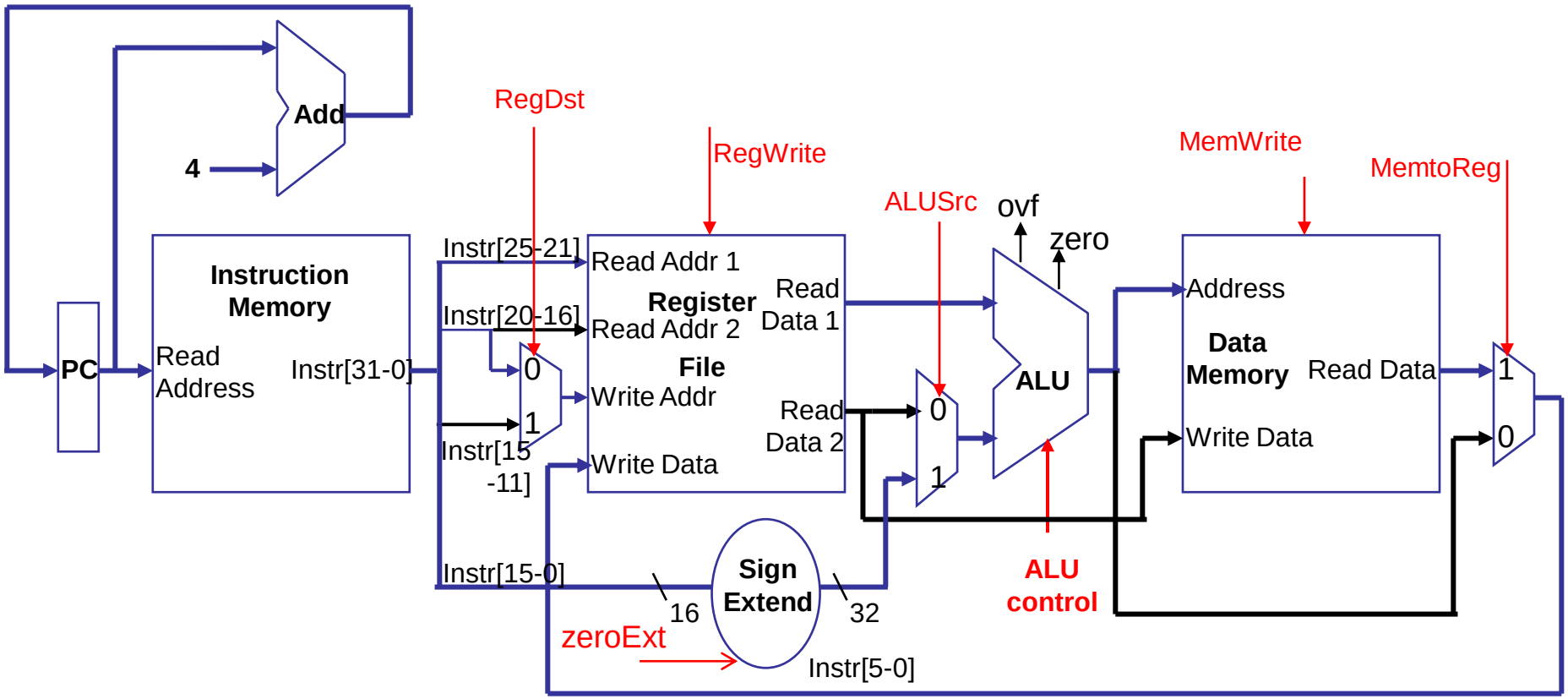
R-Type/Load/Store Datapath

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extent offset
- Load: Read memory, and update register
- Store: write register value to memory

Examples (withRTL desriptions):

LD R(rt) ← Mem[R(rs) + signExt(Imm16)]

ST Mem[R(rs) + signExt(Imm16)] ← R(rt)



- Polling: MemtoReg on LW
  - Which of the following is correct for a load instruction?

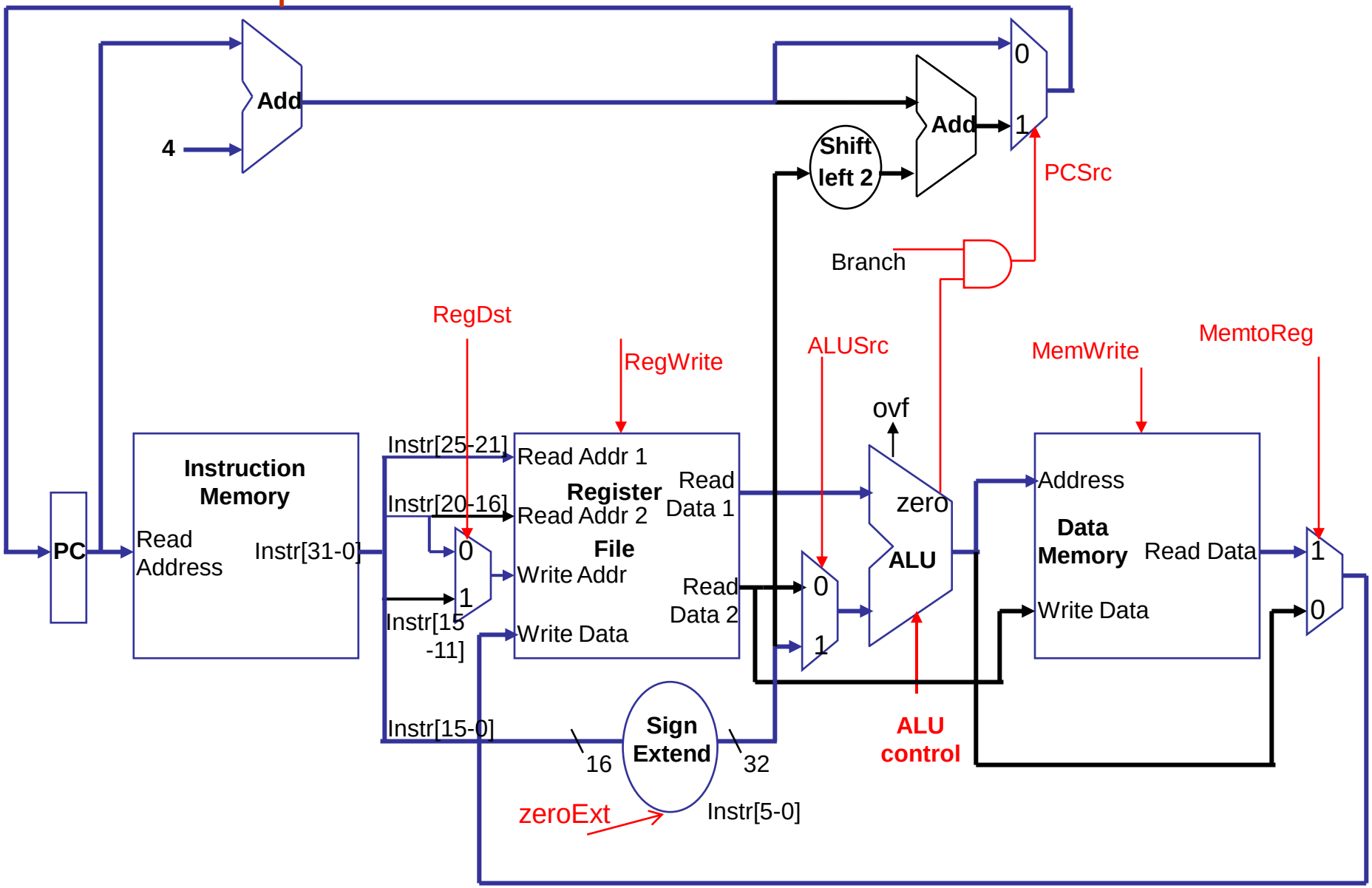
a) MemtoReg should be set to cause the data from memory to be sent to the register file.

b) MemtoReg should be set to cause the correct register destination to be sent to the register file.

c) We don not care about the setting of MemtoReg for loads.

Lecture 3: The Processor

Full Datapath



ALU Control

- **ALU used for**
  - **Load/Store: F = add (to compute the address)**
  - **Branch: F = subtract**
  - **R-type: F depends on funct field**

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

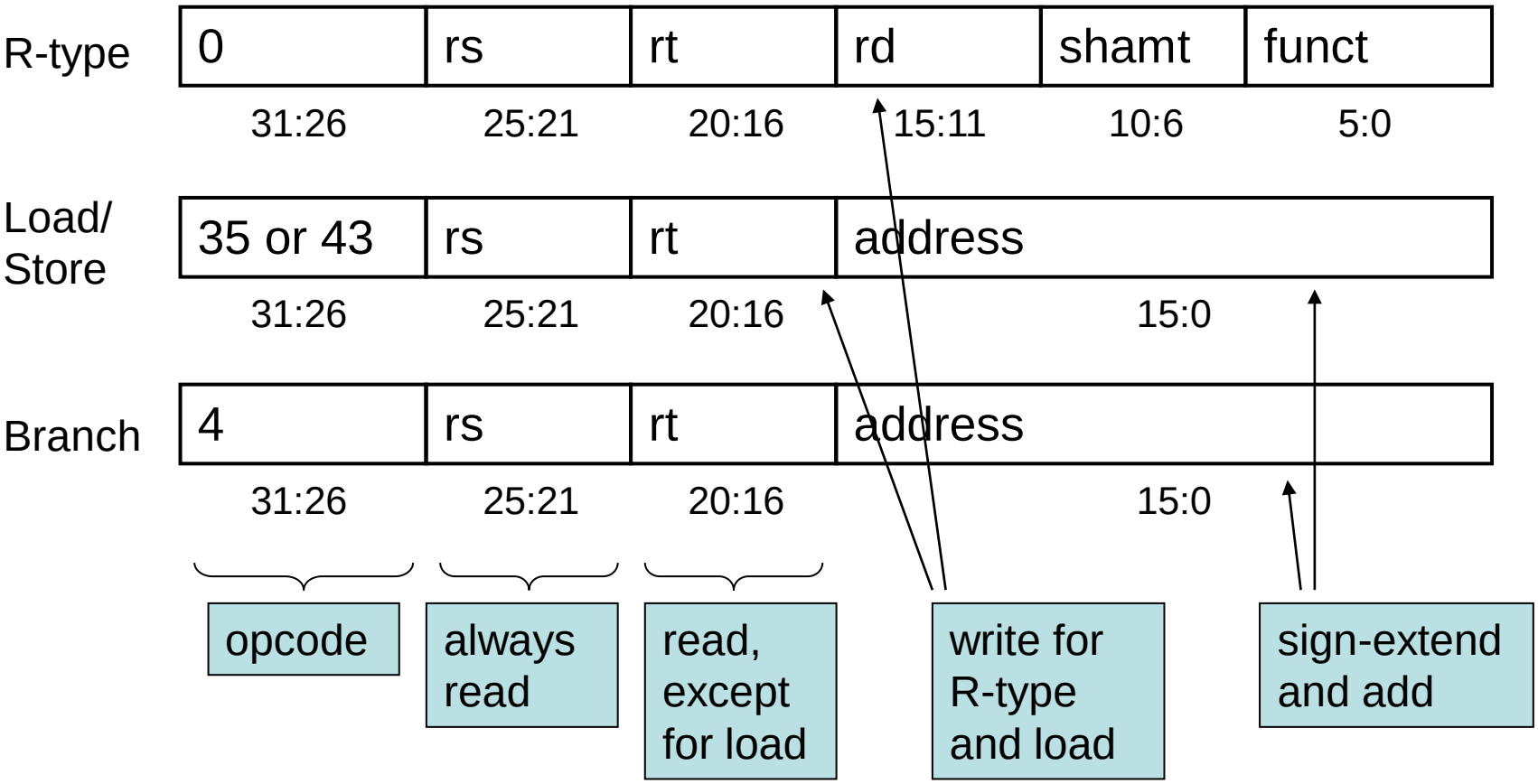
ALU Control

- **Assume 2-bit ALUOp derived from opcode**
  - **Combinational logic derives ALU control**

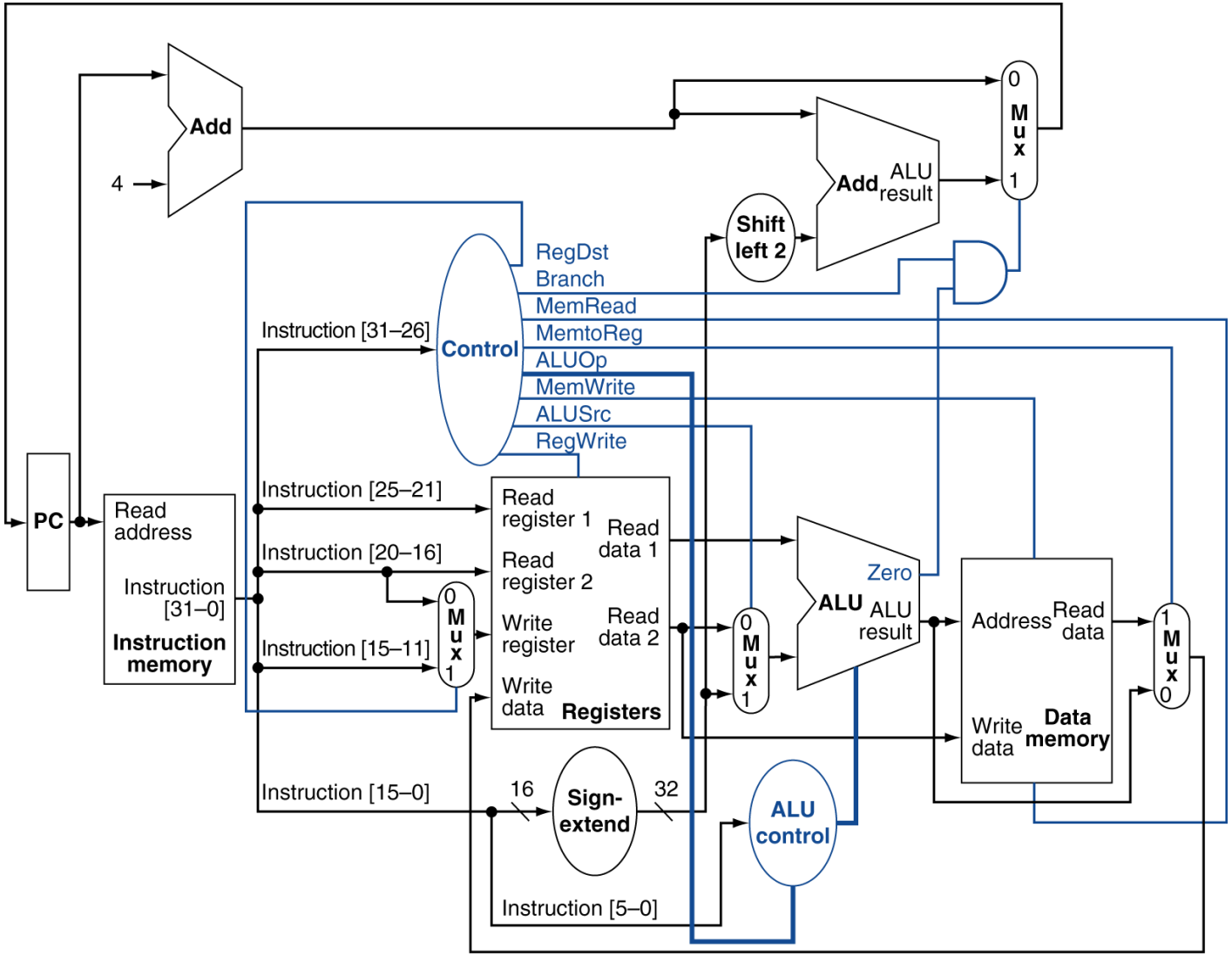
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

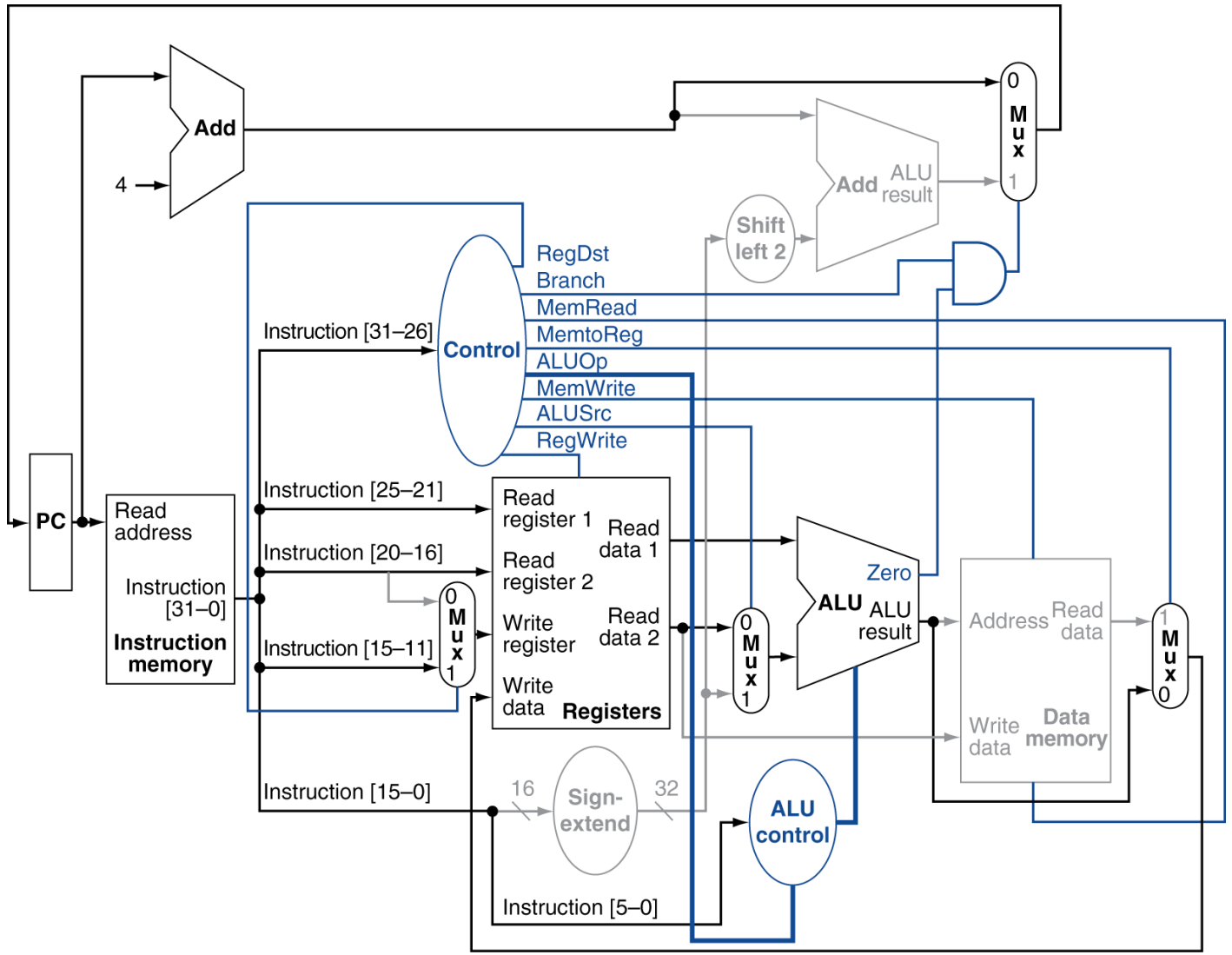
- **Control signals derived from instruction**



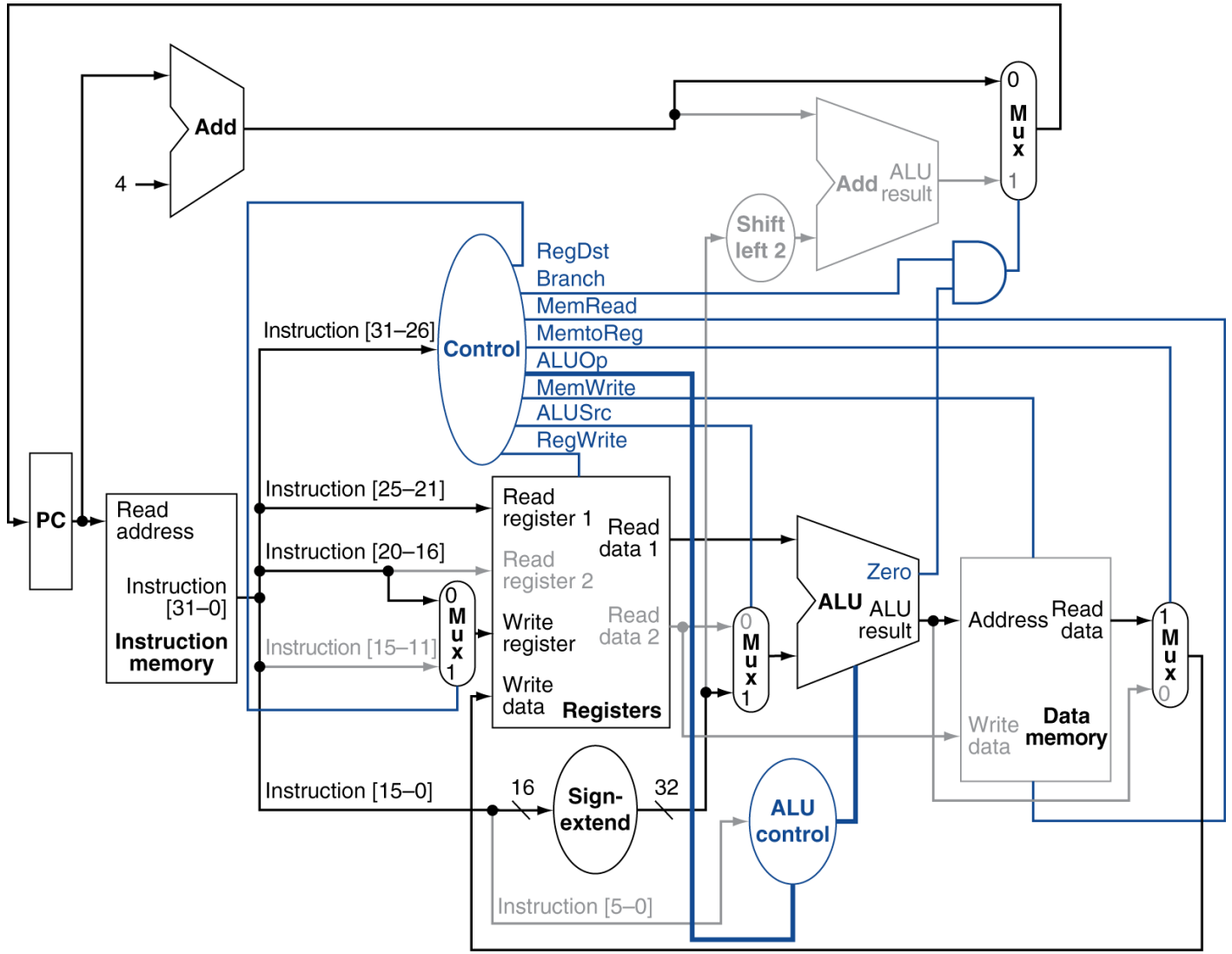
Datapath with Control



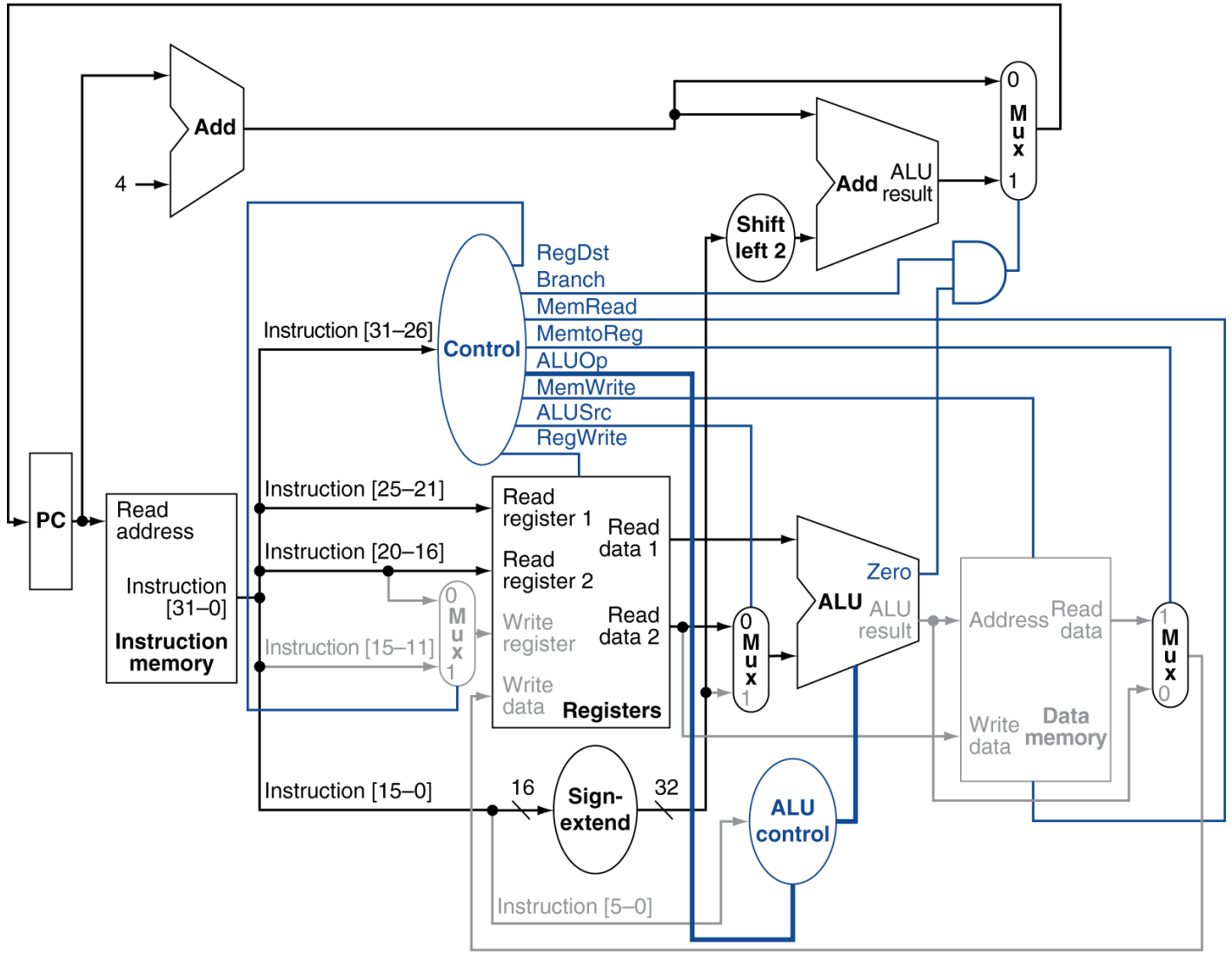
R-Type Instruction



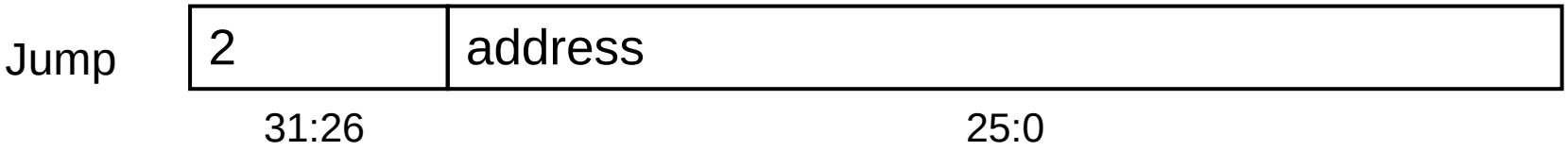
Load Instruction



Branch-on-Equal Instruction

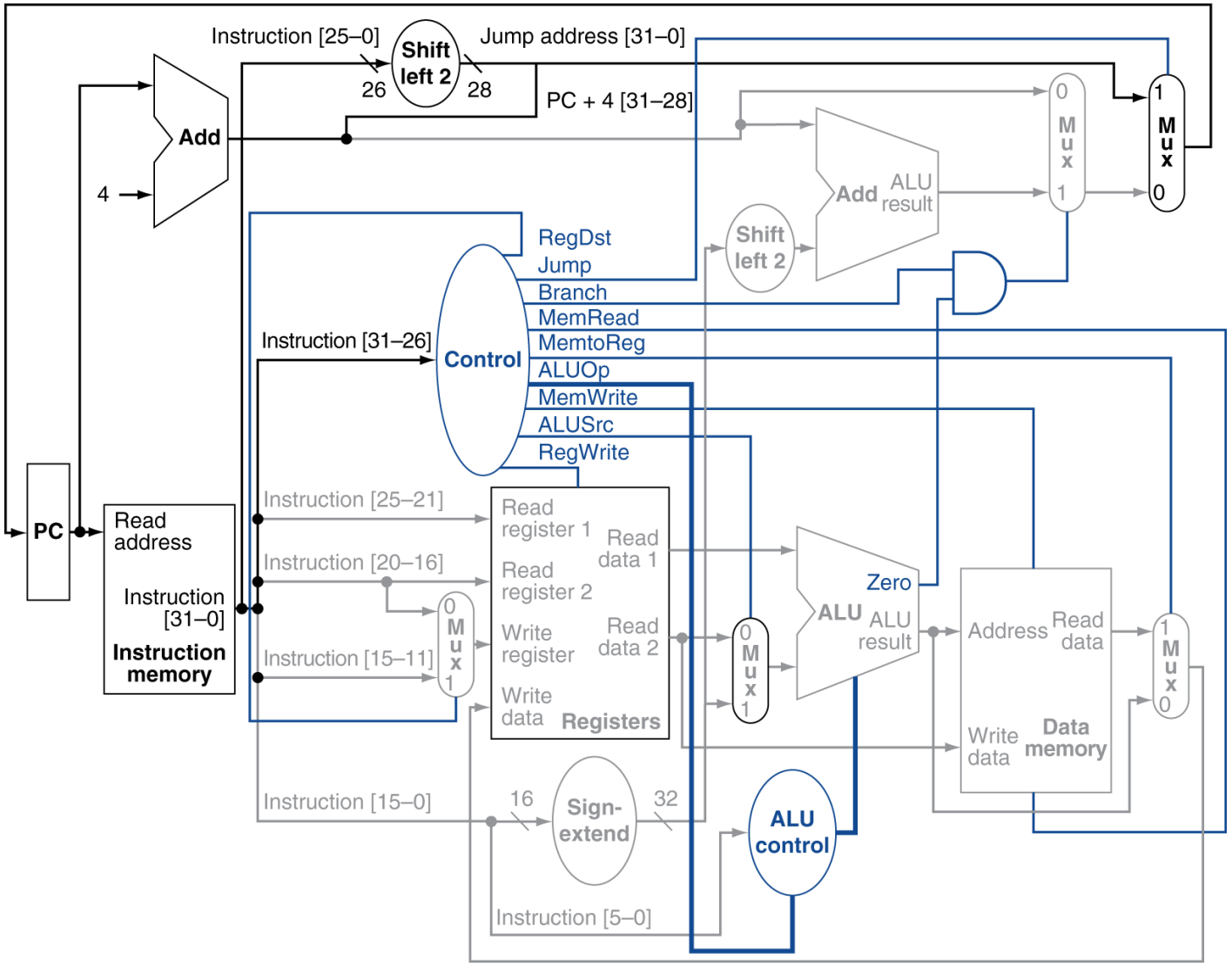


J-Type Instruction



- **Jump uses word address**
- **Update PC with concatenation of**
  - Top 4 bits of current PC+4 (bits 31:28)
  - 26-bit jump address
  - 00
- **Need an extra control signal decoded from opcode**

Datapath with Jumps Added

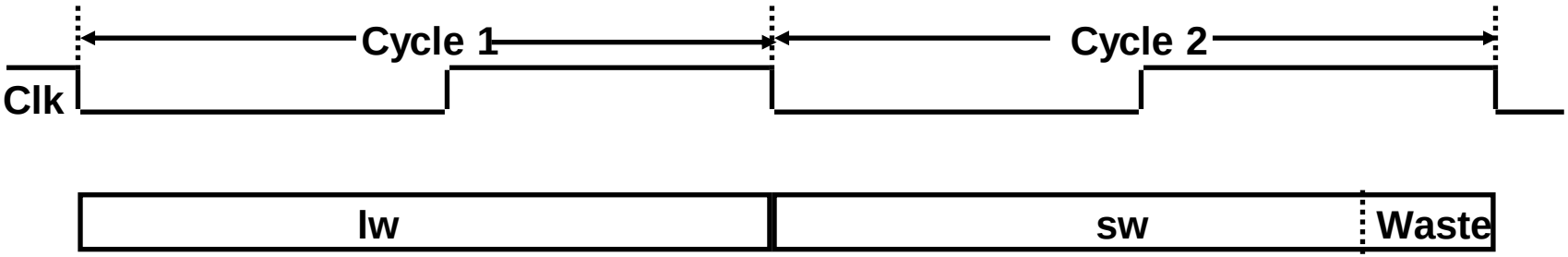


- Finalizing Control Circuit Design
  - **Combinational Network**

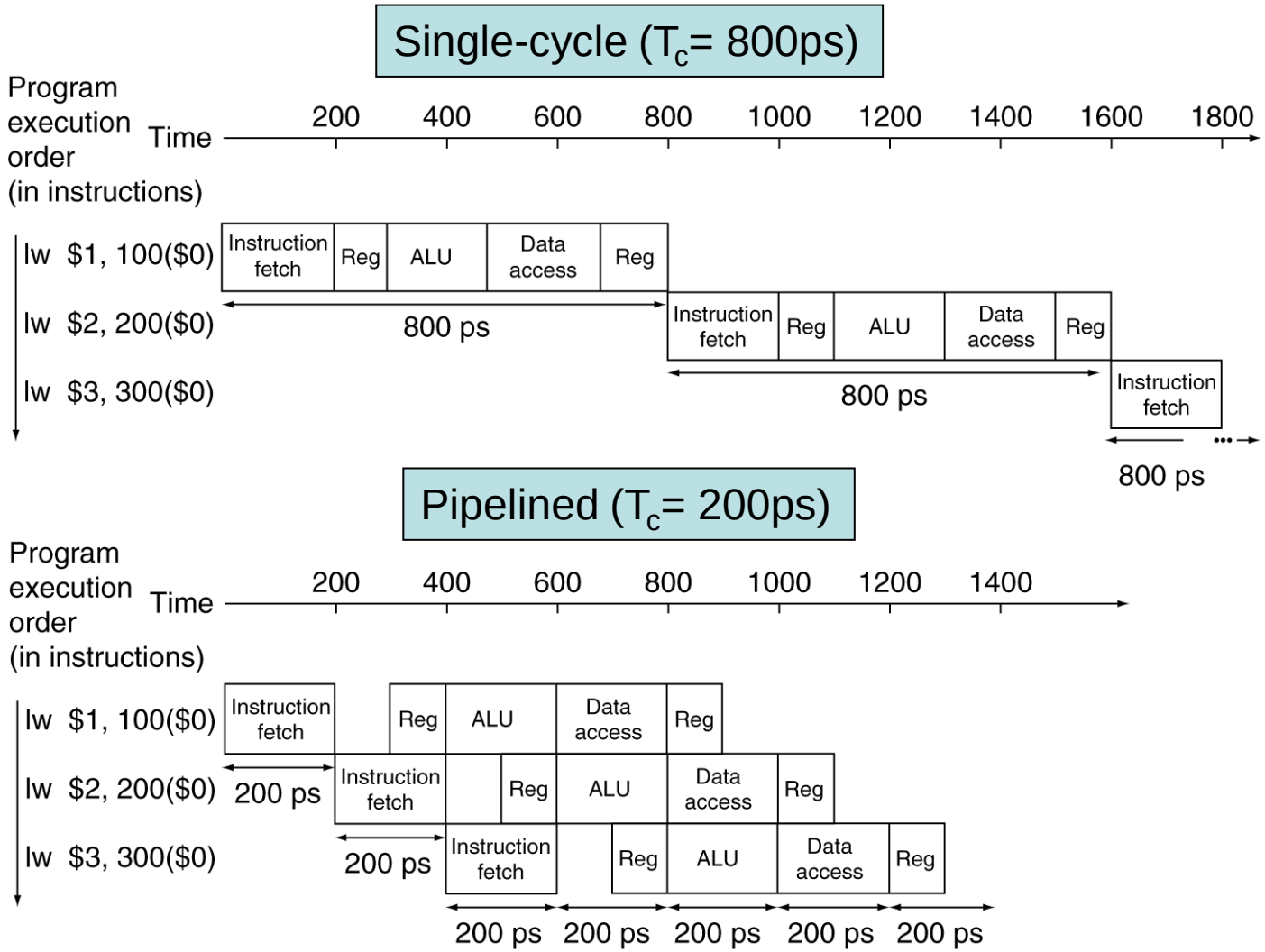
	Inputs				Outputs								
Instruction	Opcode	funct	shamt	ALU func	ALUContrl	RegDst	Branch	JUMP	MemToReg	MemWrite	ALUsrc	RegWrite	zeroExt
LW	xxxxx			ADD	0010	0	0	0	1	0	1	1	0
SW	yyyyyy			ADD	0010	x	0	0	x	1	1	0	0
BEQ	zzzzz			SUB	0110	x	1	0	x	0	0	0	0
R-type	add	100000		ADD	0010	1	0	0	0	0	0	1	0
	sub	100010		SUB	0110	1	0	0	0	0	0	1	0
	and	100100		AND	0000	1	0	0	0	0	0	1	0
	or	100101		OR	0001	1	0	0	0	0	0	1	0
	shift	xxxx	xxxx	SHIFT		1	0	0	0	0	0	1	0
	ori					1	0	0	0	0	0	1	1
Jump					x	x	x	1	x	0	x	0	x

Performance Issues

- Single Clock
  - Functional Block have different delays
  - Single clock designs use the clock inefficiently – the clock must be timed to accommodate the slowest instruction
  - Longest delay determines clock period
    - √ Critical path: load instruction
    - √ Instruction memory → register file → ALU → data memory → register file
  - Violates design principle: Making the common case fast
  - We will improve performance by pipelining



- Solution


Single-cycle (T_c= 800ps)
Program execution order (in instructions)
Time
200 400 600 800 1000 1200 1400 1600 1800
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
800 ps
800 ps
800 ps
Pipelined (T_c= 200ps)
Program execution order (in instructions)
Time
200 400 600 800 1000 1200 1400
200 ps
200 ps
200 ps 200 ps 200 ps 200 ps 200 ps

- Next lecture
  - Multicycle datapath - Pipelining