Chapter4-1

The Processor: Datapath and Control (Single-cycle implementation)

臺大電機系

吳安宇教授

2025/03/17 v2

outline

- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

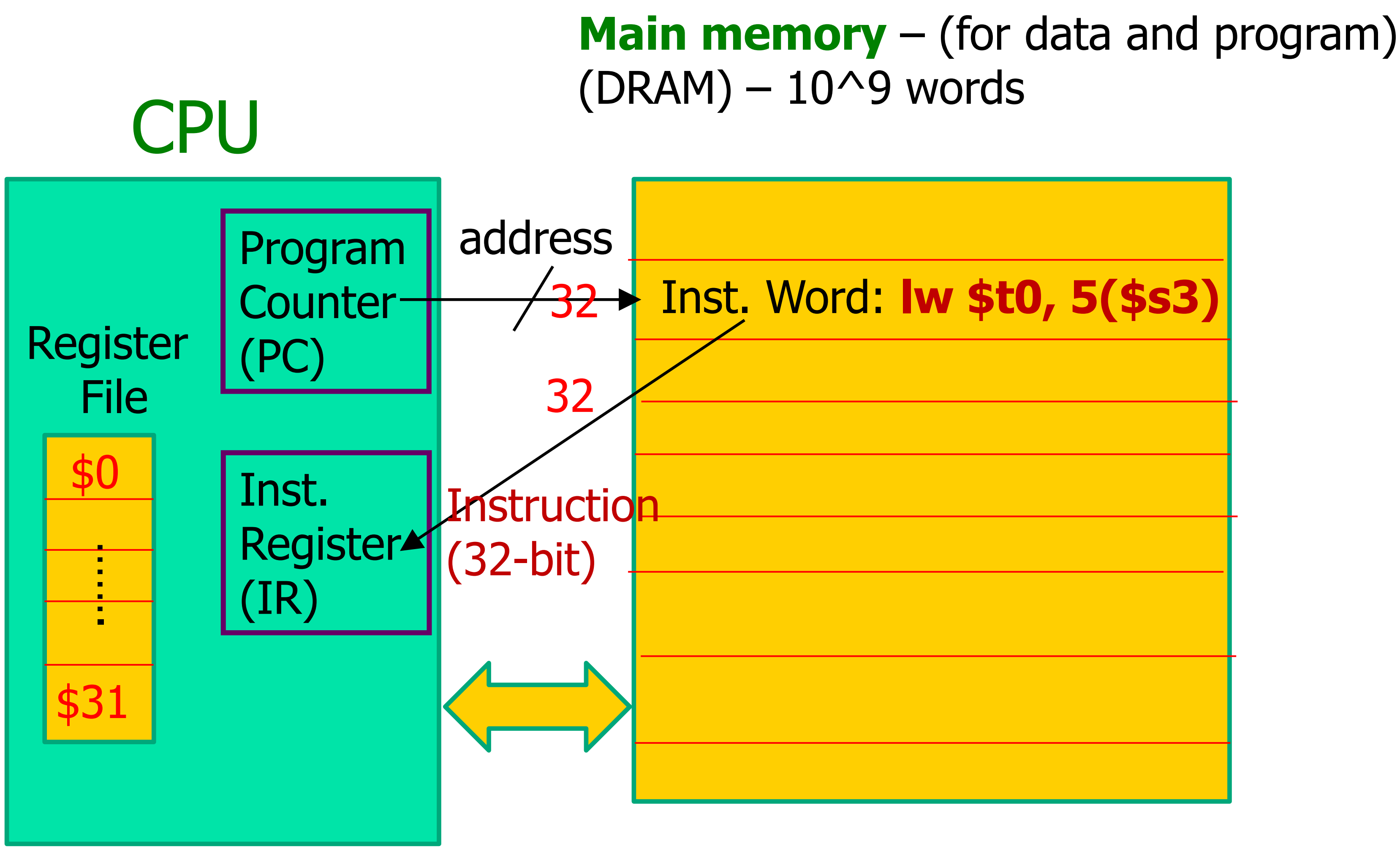
Introduction

- Show key issues in creating datapaths and designing controls.

- Design and implement the MIPS instructions including:
  (1) memory-reference instructions: lw, sw
  (2) arithmetic-logical instructions: add, sub, and, or, slt
  (3) branch instructions: beq, j

- Guideline in hardware implementation:
  (1) *Make the common case fast*
  (2) *Simplicity favors regularity*

Addressing mode of Inst. Fetch (IF)



Overview of the implementation (1/3)

- For every instruction, the first two steps are the **same**:

  A. Fetch: Send the Program Counter (PC) to the memory that contains the code (Instruction Fetch)

  B. Read registers: Use fields of the instructions to select the registers to read.

  - Load/Store : Read **one** register (I-type)
  - Others : Read **two** registers (R-type)

  - ***lw $s1, 200($s2)***
  - ***add $t0, $s1, $s2***

Overview of the implementation (2/3)

C. **Common actions** for three instruction types:

(all instructions use **ALU** after reading registers)

(1) Memory-reference instructions:

use ALU to calculate "effective address"

e.g., lw $t0 offset($s5) → compute ***offset + $s5***

(2) Arithmetic-logical instructions:

use ALU for opcode execution → ***add, sub, or, and***

(3) Branch instructions:

use ALU for comparison – beq $s1, $s2, offset

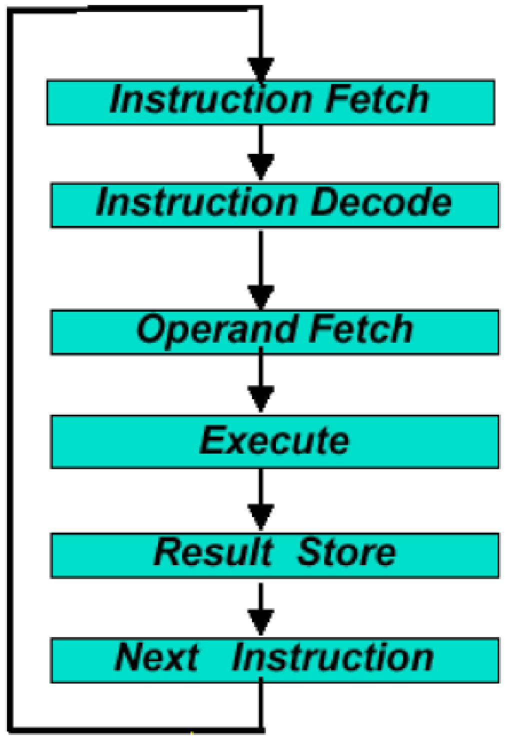
→ ***$s1-$s2, and check zero bit of the results***

Overview of the implementation (3/3)

D. After using ALU (different):

1) *Memory-reference instructions:* need to access the memory containing the data to complete a "load" operation, *or* "store" a word to that memory location.
2) *Arithmetic-logical instructions:* write the result of the ALU back into a destination register.
3) *Branch instructions:* need to change the next instruction address based on the comparison (i.e., change the value of Program Counter, PC)

Typical Instruction Execution



IF: access instruction memory

ID: identify opcode, generate control signals

OF: read register(s), route immediate field

EX: perform operation; generate address, access data memory

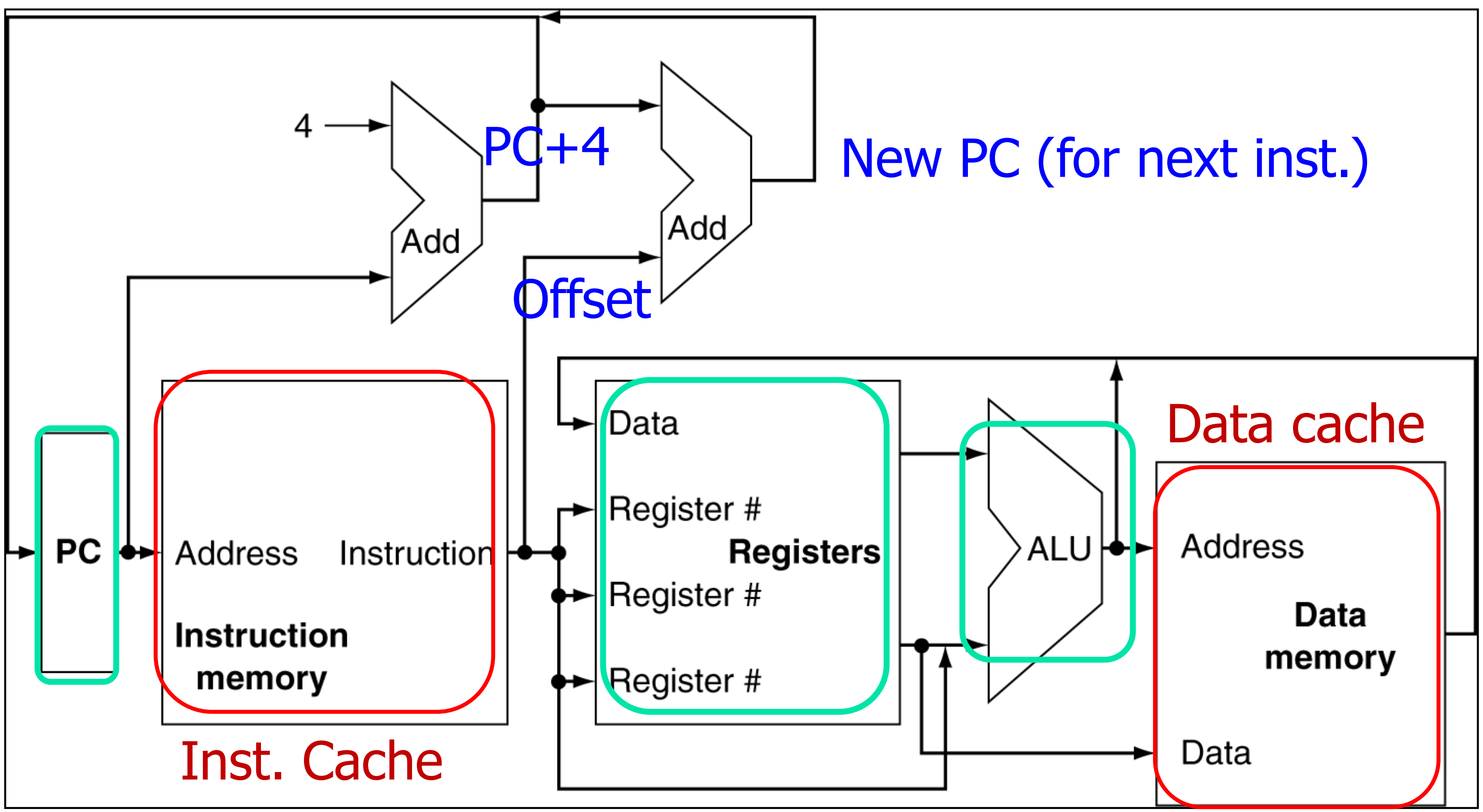
RS: write register, write memory

NI: increment PC or update PC with new target

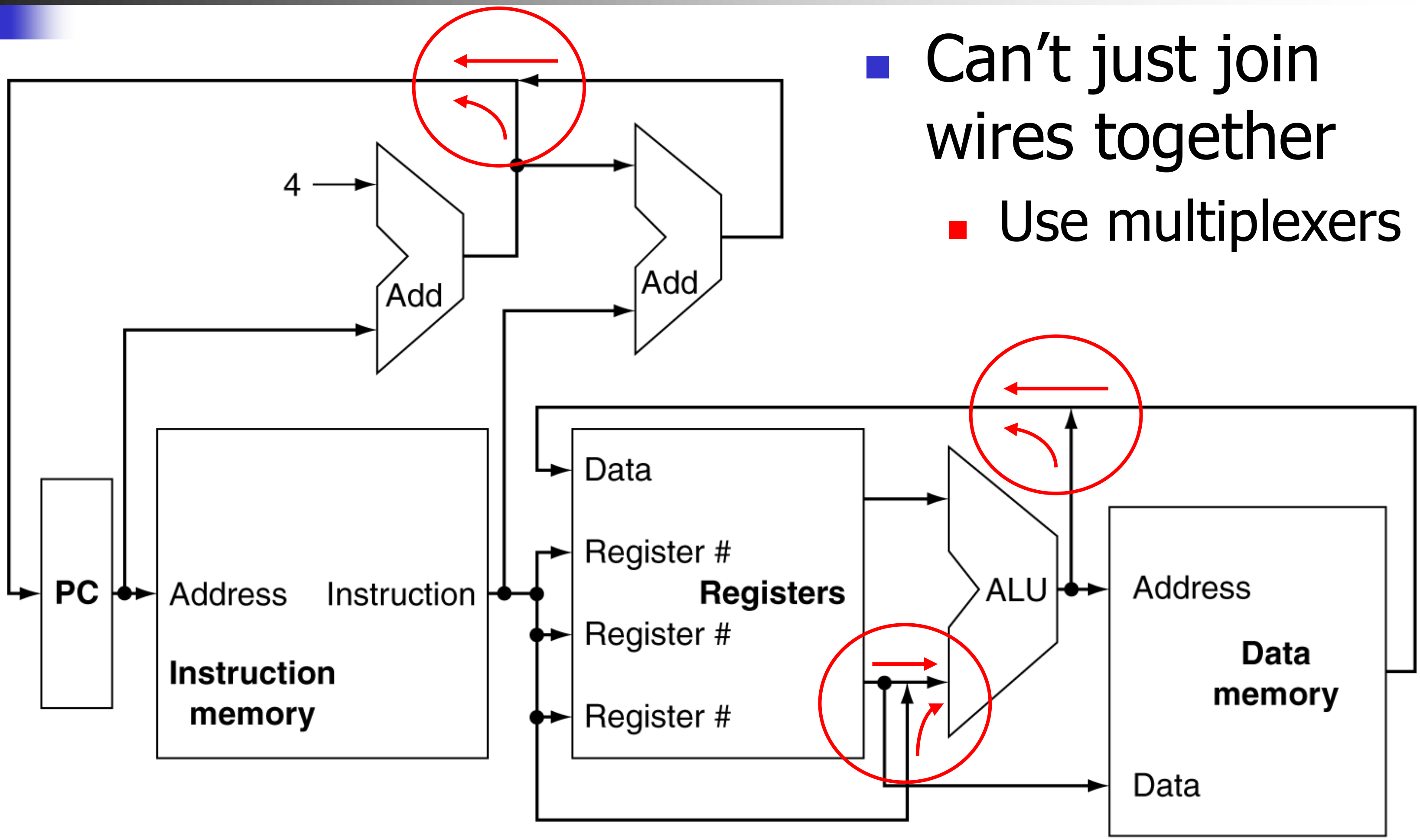
Note that each step does not necessarily correspond to a clock cycle. These only describe the basic flow of instruction execution. The details vary with instruction type.

Abstract View of MIPS CPU Implementation (1/3)

- An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

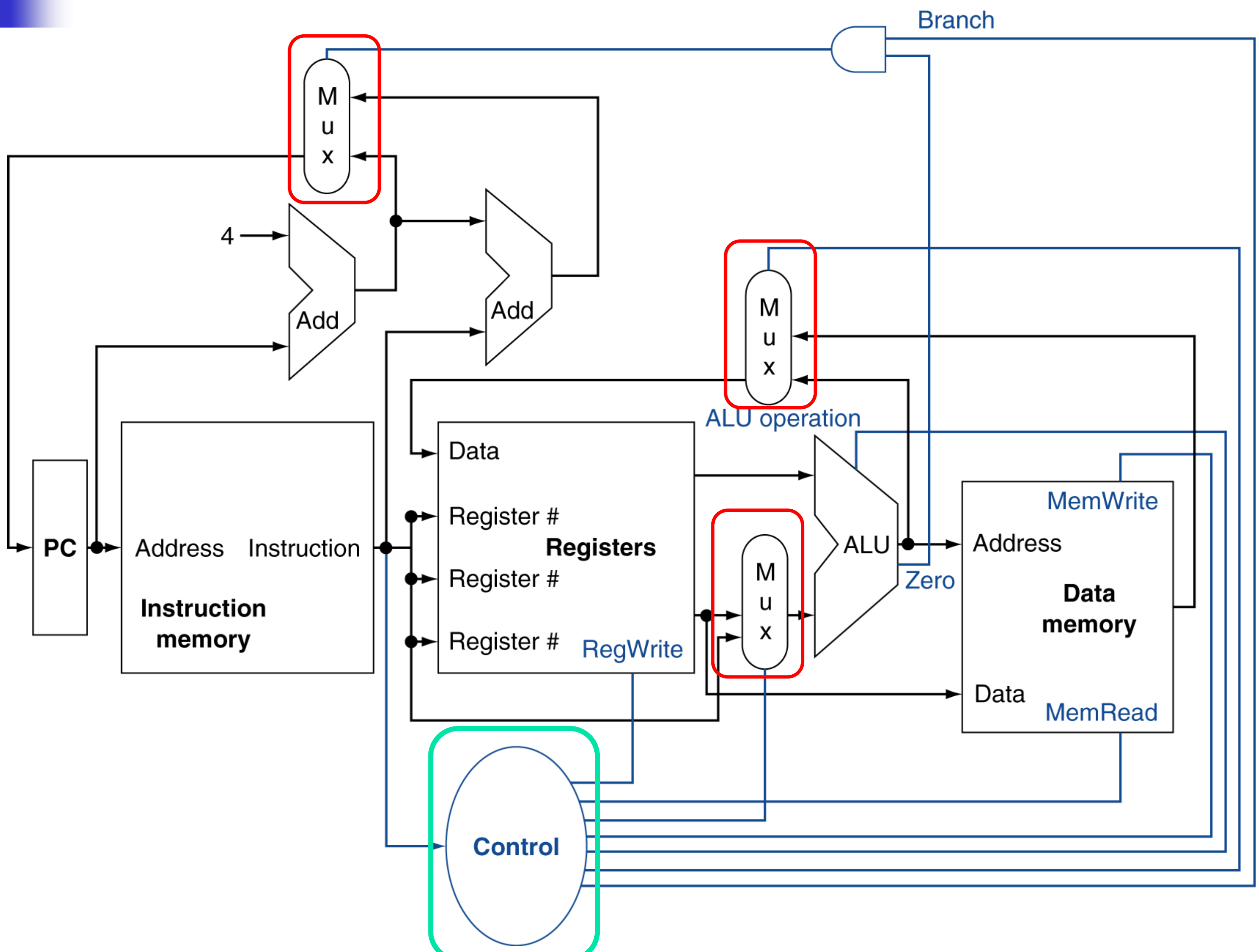


Multiplexers (2/3)



- Can't just join wires together
  - Use multiplexers

Control Signals (3/3)

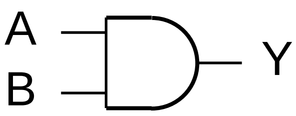


Outline

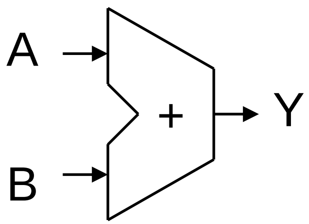
- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

Combinational Elements

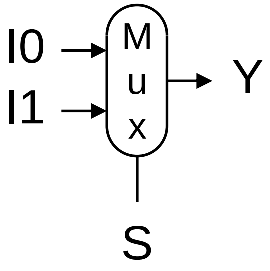
- AND-gate
  - Y = A & B



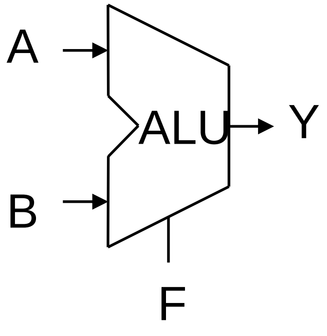
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

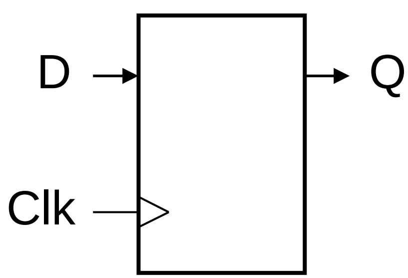


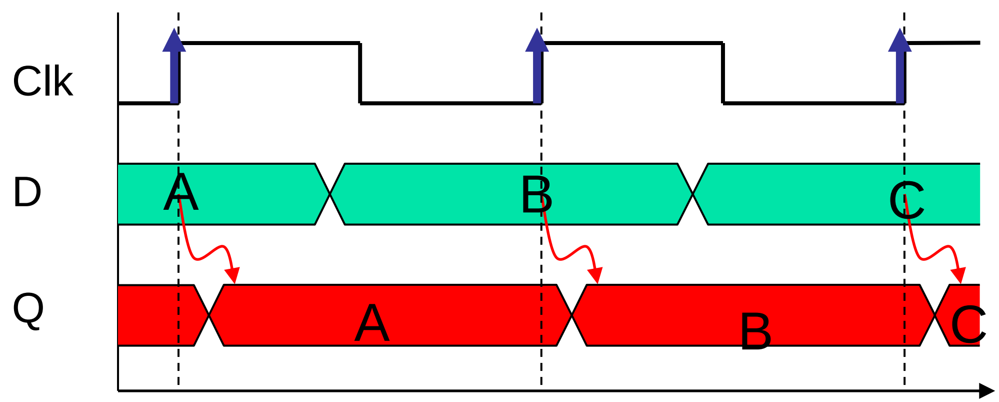
- Arithmetic/Logic Unit (ALU)
  - Y = F(A, B)



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

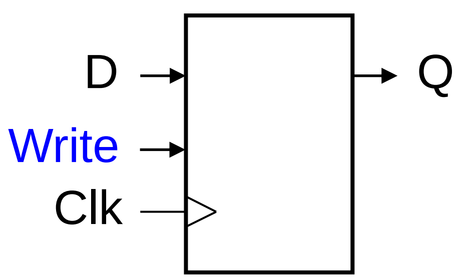


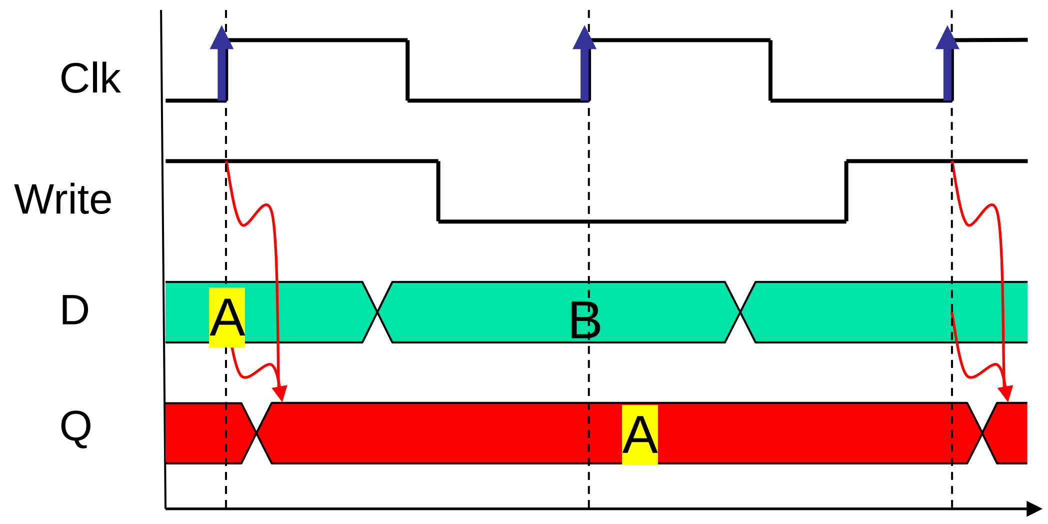


Q value: stable for **one cycle** for our operations

Sequential Elements

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

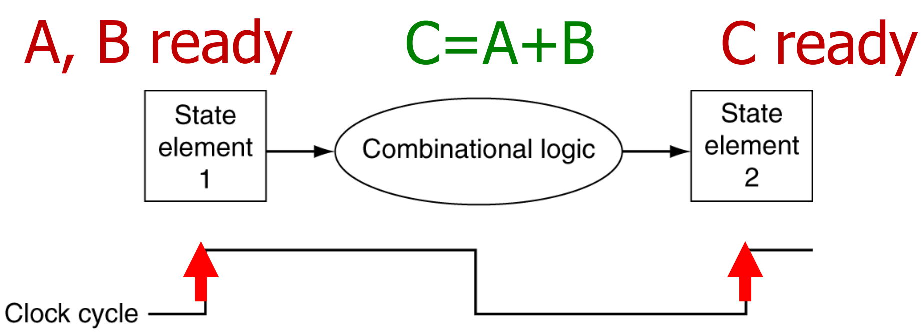


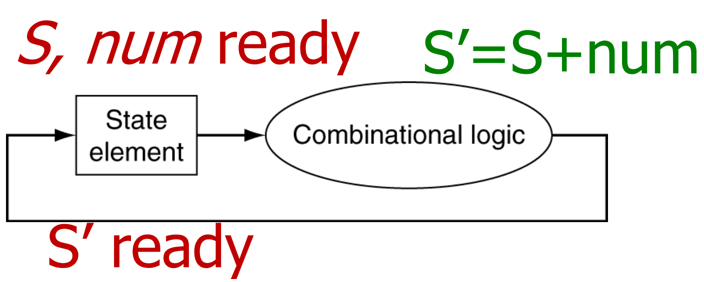


Clocking Methodology

- Combinational logic transforms data during clock cycles
  - An **Edge-triggered** methodology
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

  - Typical execution:
    - Read contents of some state elements,
    - Send values through some combinational logic
    - Write results to one or more state elements



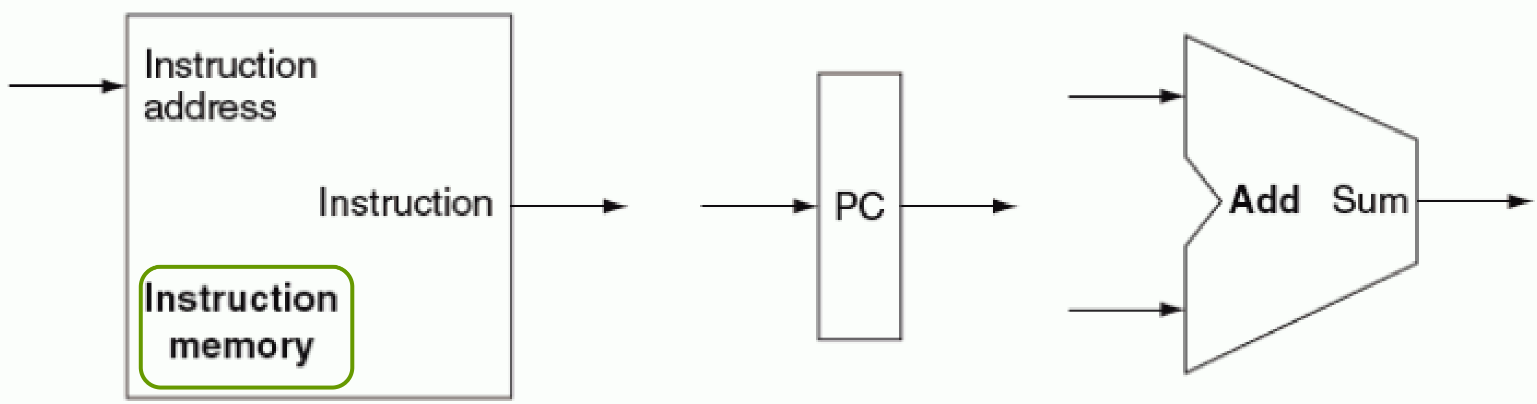


Outline

- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

Building a Datapath

- Basic elements for “access” instructions:
  - (a) Instruction Memory (**IM**) unit
  - (b) Program Counter (**PC**): increase by 4 each time
  - (c) Adder: to perform “increase by 4”

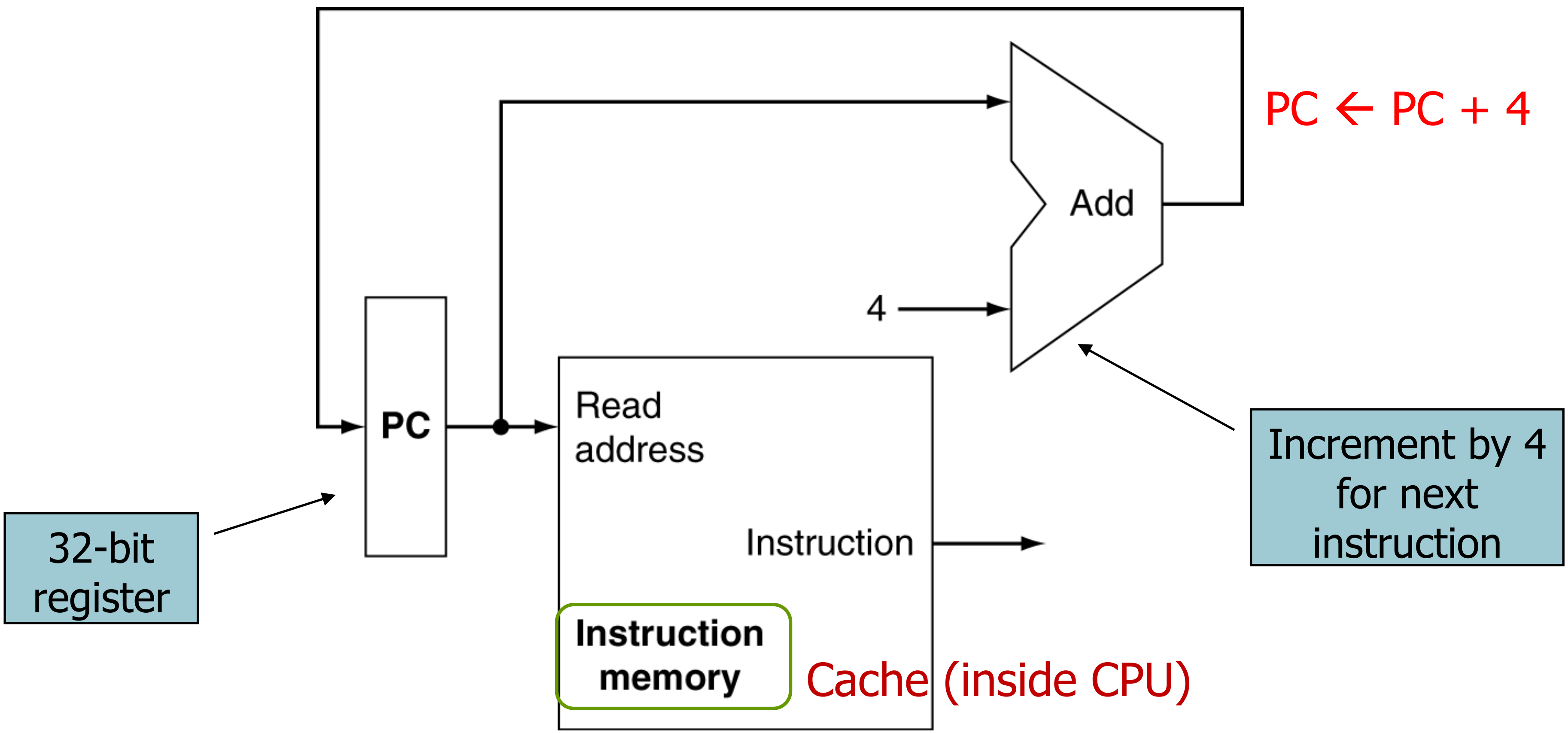


a. Instruction memory　　b. Program counter　　c. Adder

Cache (inside CPU)

Building a Datapath for PC

- A portion of datapath used for fetching instructions and incrementing the program counter (Inst. Fetch, IF)

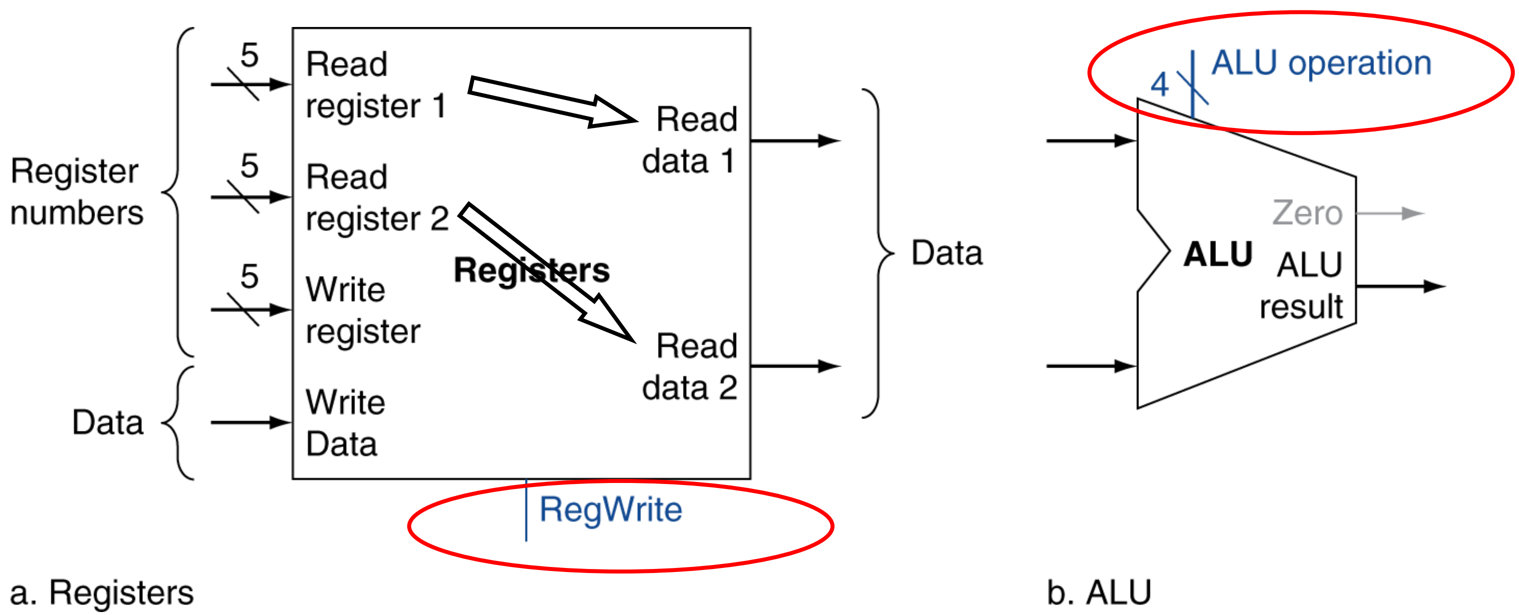


Building a Datapath for R-type

- Basic operations for **R-type** instructions:
  - Function:
    1. Read two registers
    2. Perform an ALU operation on the contents of registers
    3. Write the result back into the destination register
  - Read operation:
    1. Input to the “register file” to specify the indices of the TWO registers to be read.
    2. Two outputs of the register contents.
  - Write operation:
    1. An input to the “register file” to specify the index of the register to be written.
    2. An input (from ALU output) to supply the data to be written into the specified register.

Building a Datapath

- Elements which we need:
  - (a) **Register file**: a collection of registers in which any register can be read or written by specifying the index of the register in the file.
  - (b) **ALU (32 bits):** operate on the values read from the registers.



Review of Instruction Format

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

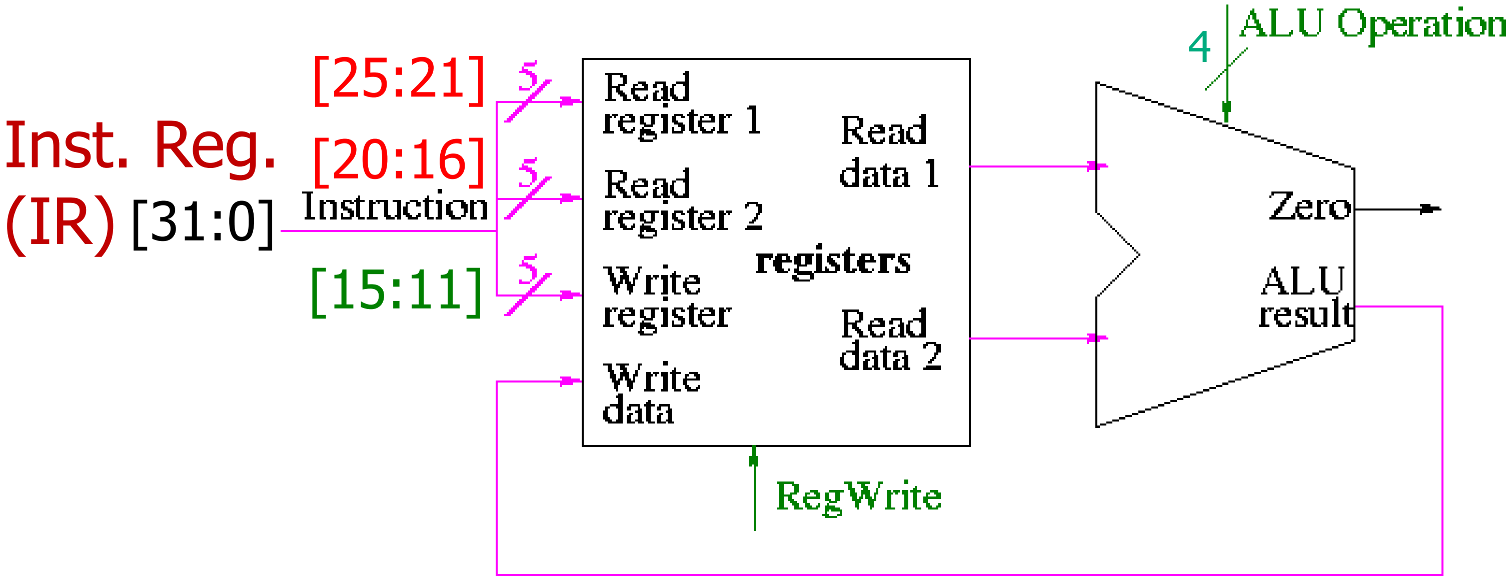
b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

FIGURE 4.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.

Datapath for R-type instructions



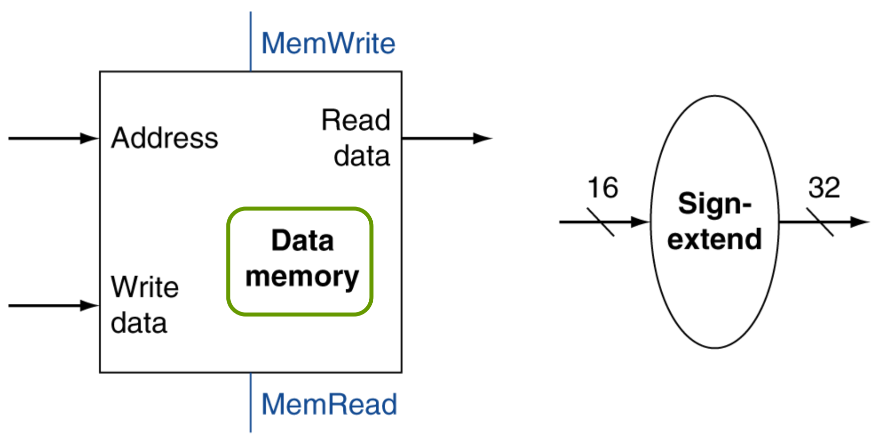
add $rd, $rs, $rt

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
		[25:21]	[20:16]	[15:11]		

Building a Datapath for I-type

- Basic elements for load/store instructions:
  1. Data memory unit: read/write data
  2. Sign-extend unit: sign-extend the 16-bit offset field in the instruction to a 32-bit signed value.
  3. Register file
  4. ALU (add “reg” + “offset” to computer the mem address)

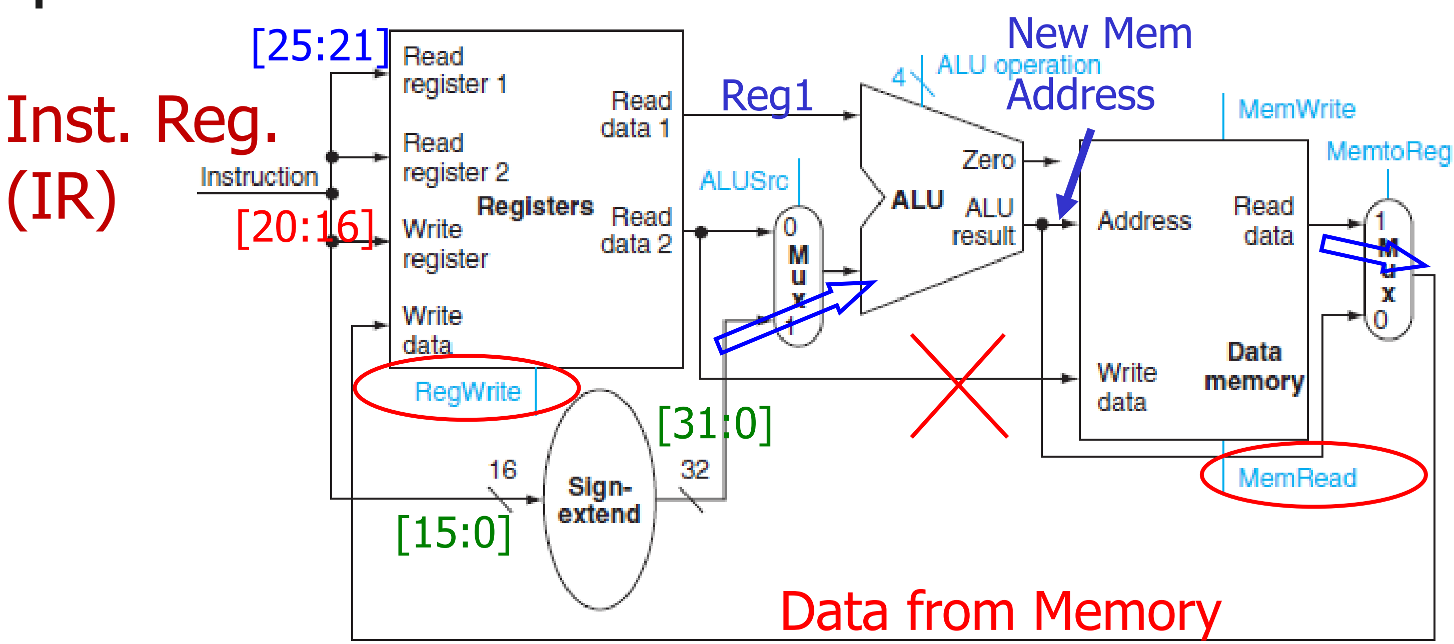
SRAM-based Data cache (inside CPU)



a. Data memory unit b. Sign extension unit

-- (3) & (4) are just shown as the previous slide.

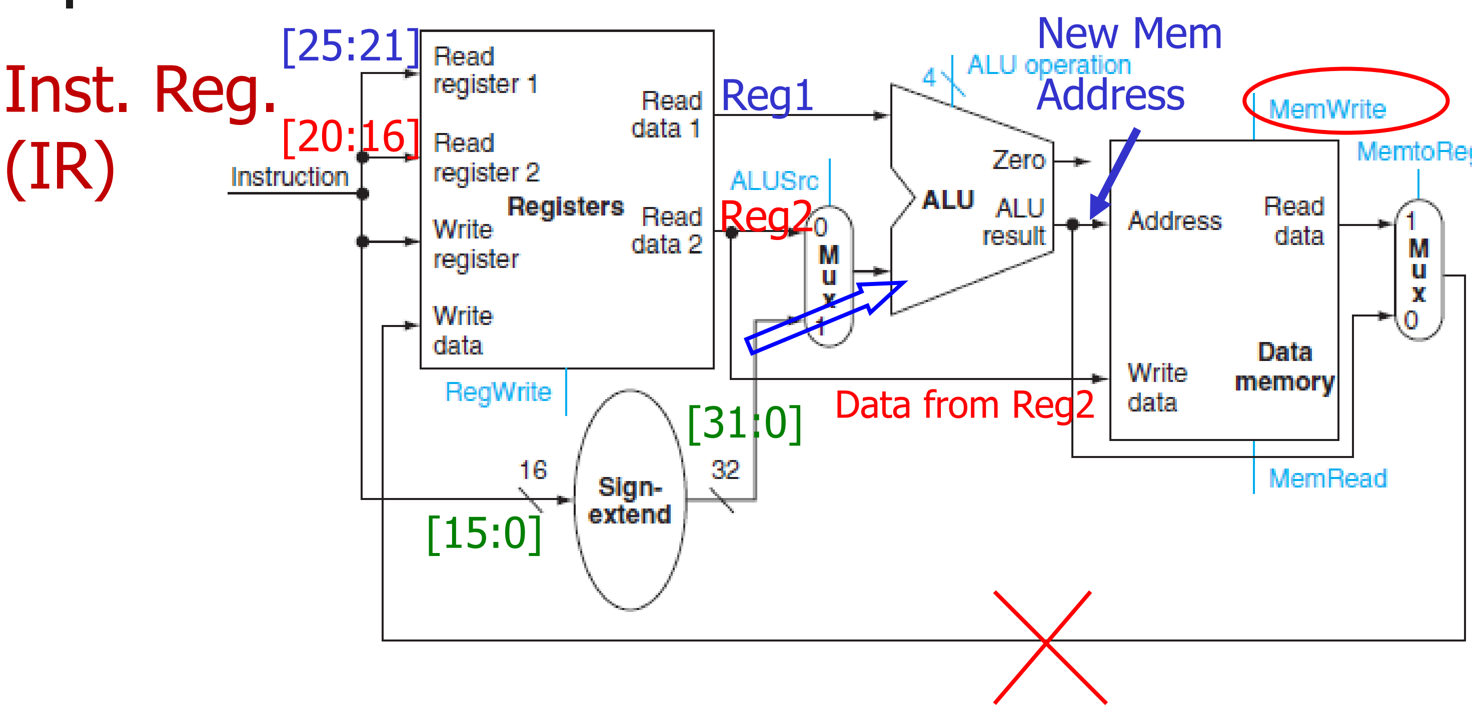
Datapath for **lw** instructions



lw $rt, 100($rs)

I:	op	rs	rt	address / immediate
		[25:21]	[20:16]	[15:0]

Datapath for **sw** instructions



sw $rt, 100($rs)

I:	op	rs	rt	address / immediate
		[25:21]	[20:16]	[15:0]

Branch Instructions

- (ex) beq $t1, $t2, offset

  \# if ($t1==$t2)

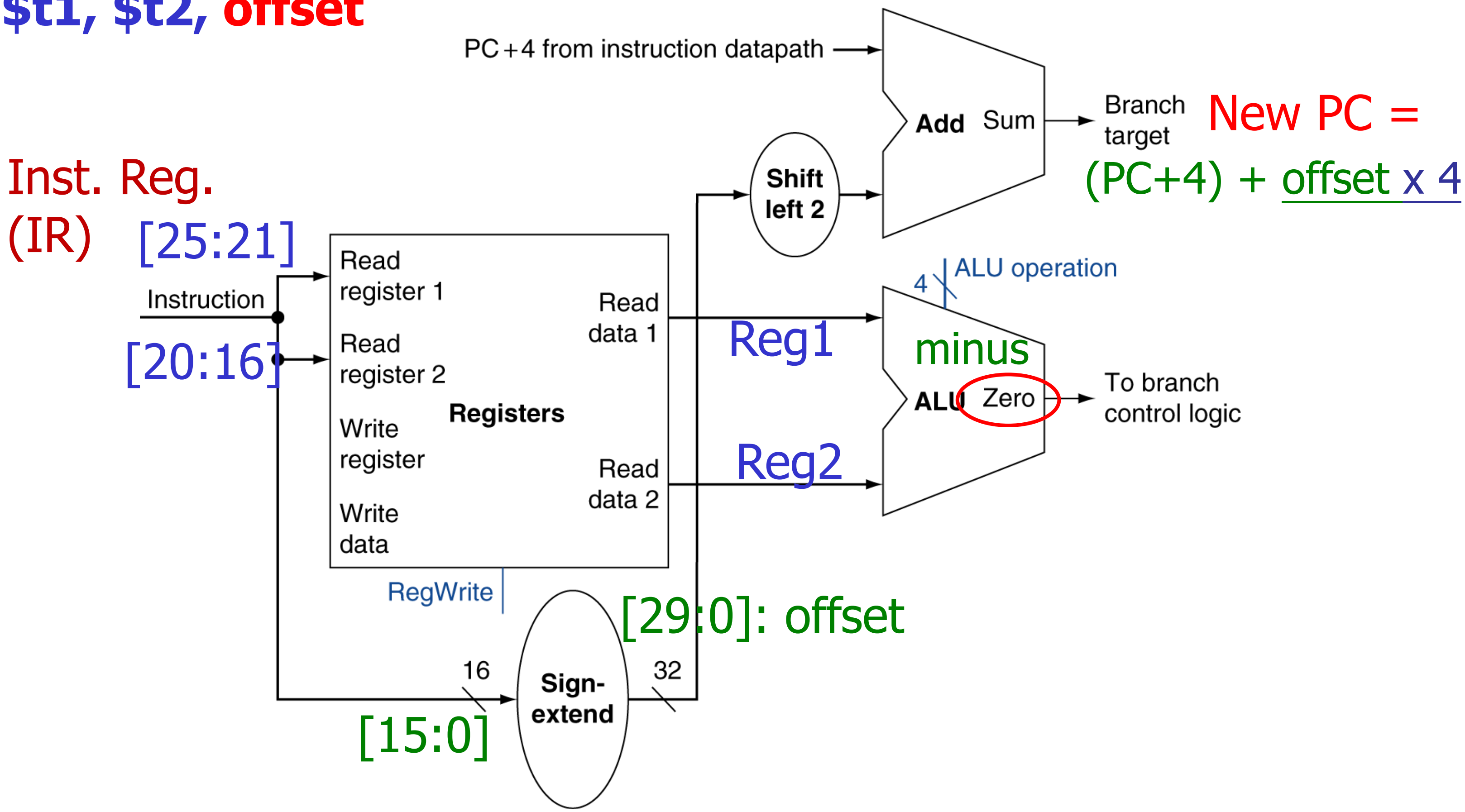
  goto (PC+4 + **offset**) // PC ← (PC+4)+offset

  else

  execute next instruction // PC ← PC+4
- Note:
  1. The offset field is shifted left 2 bits so that it's a "word offset".
  2. Branch is taken (taken branch): when the condition is true, the branch target address becomes the new PC.
  3. Branch isn't taken (untaken branch): the incremented PC (PC+4) replaces the current PC, just as for normal instruction.
- Operations:
  1. Compute the branch target address.
  2. Compare the contents of the two registers.

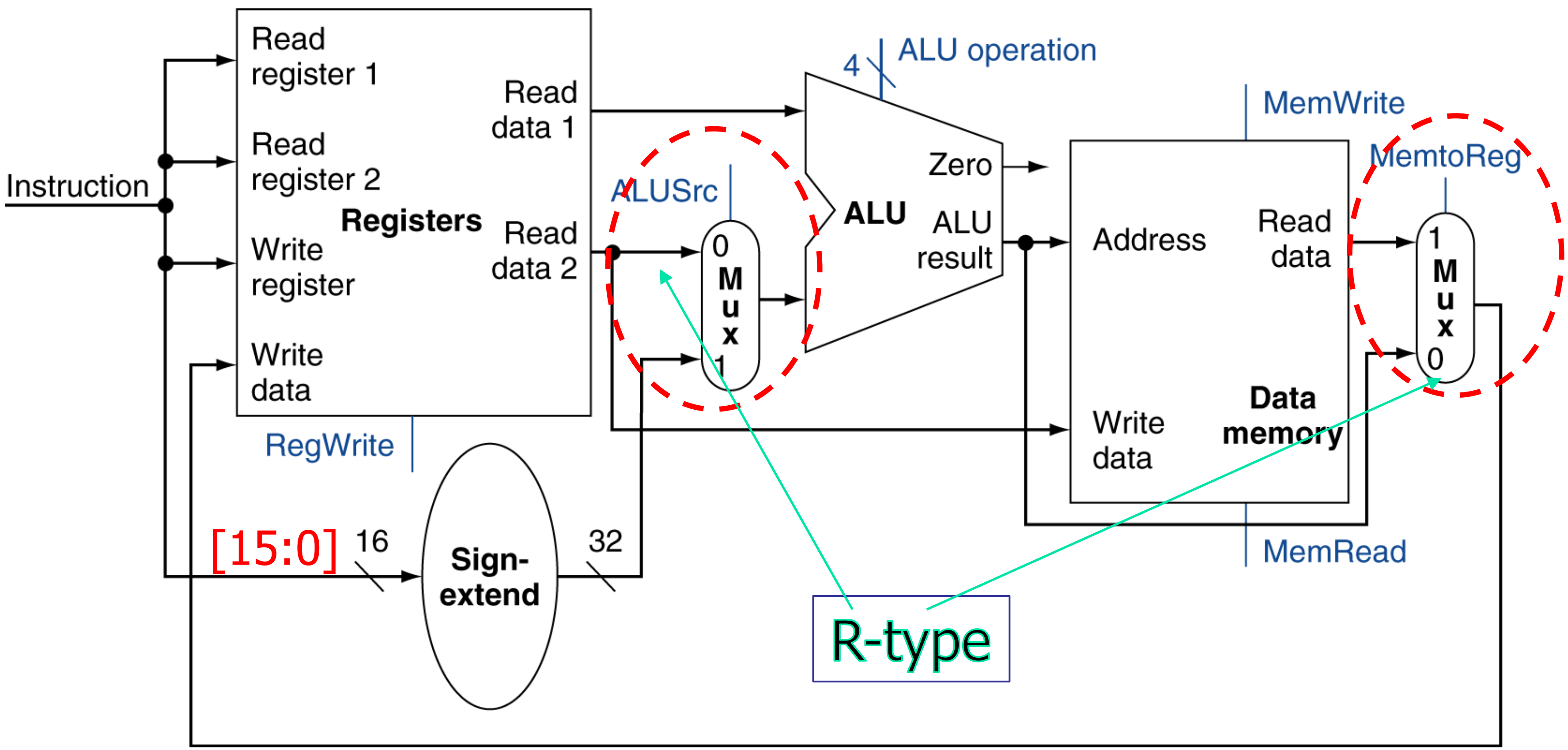
Datapath for "beq" Instructions

- beq $t1, $t2, offset

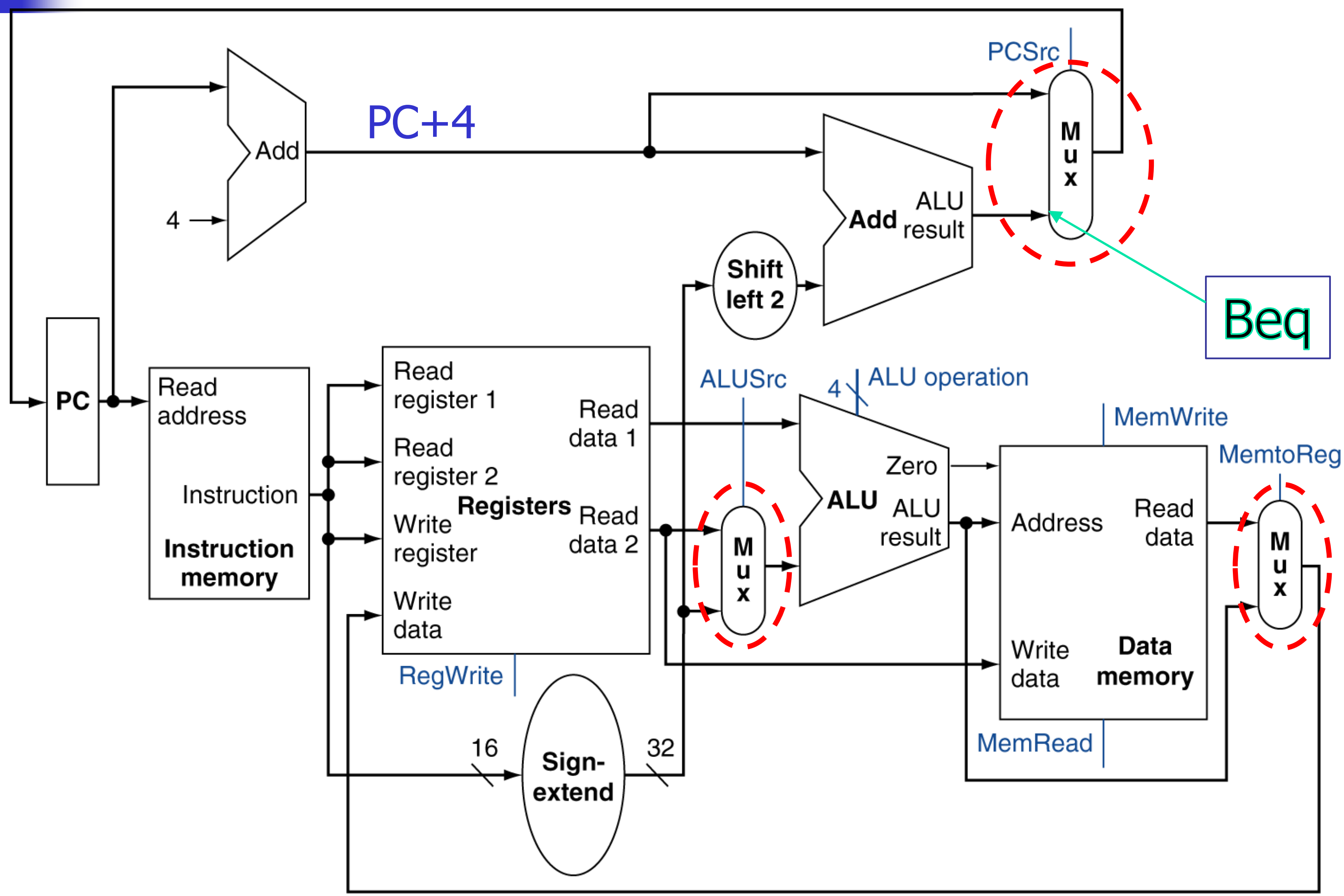


I:	op	rs	rt	address / immediate
		[25:21]	[20:16]	[15:0]

Datapath for both Memory and R-type Instructions



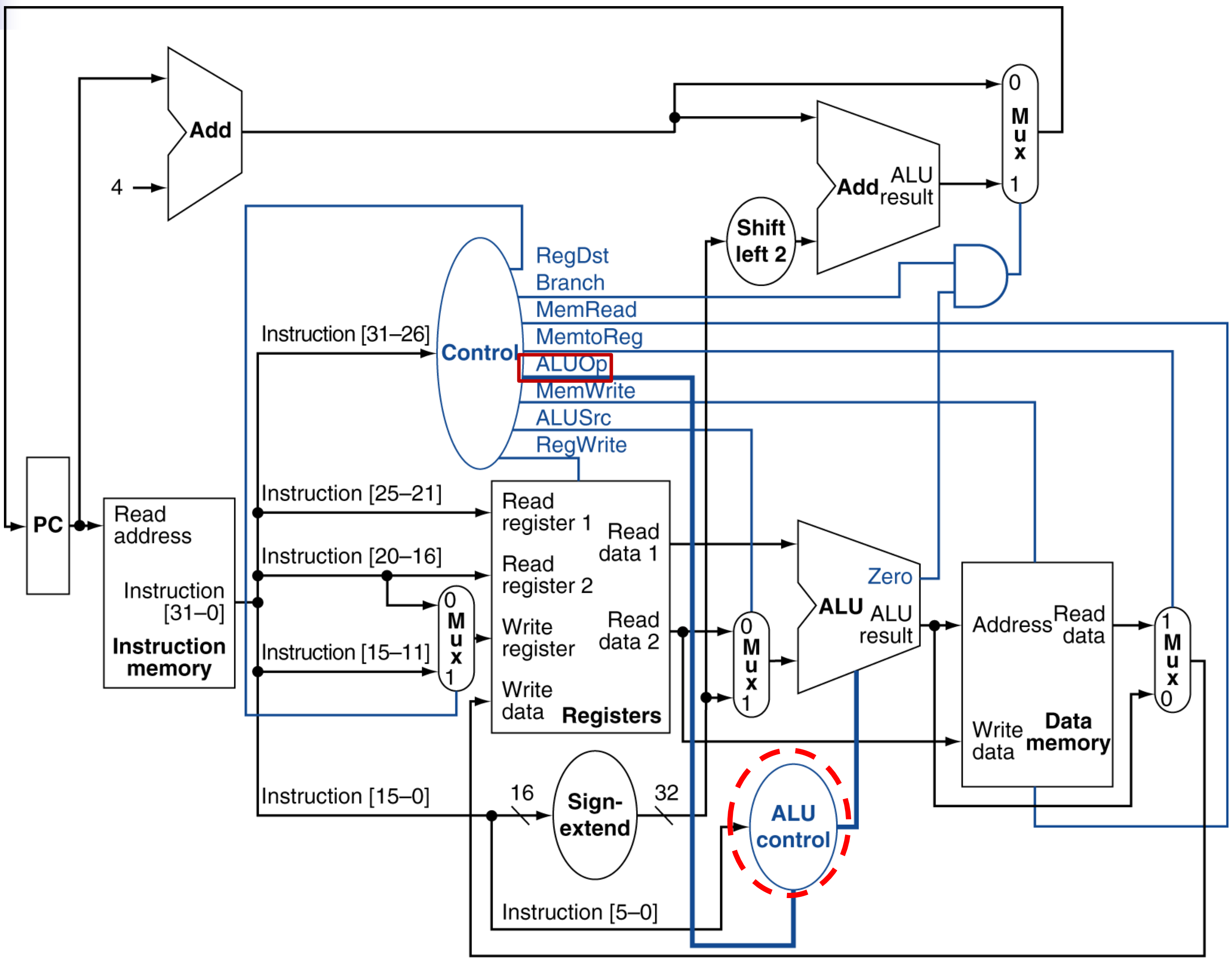
Simple Datapath for All three types of Instructions



Outline

- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

Basic Datapath with Control Signals



Design of ALU control unit

- Depending on the instruction type, the ALU will perform
  - lw/sw: compute the memory address by addition
  - R-type (add, sub, AND,OR, slt): depending on the value of the 6-bit function field
  - Branch (beq): subtraction for comparison ($r1-$r2)

- ALU control signals:

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

ALU control for each type of instruction

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

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW		store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type		subtract	100010	subtract	0110
R-type		AND	100100	and	0000
R-type		OR	100101	or	0001
R-type		set on less than	101010	set on less than	0111

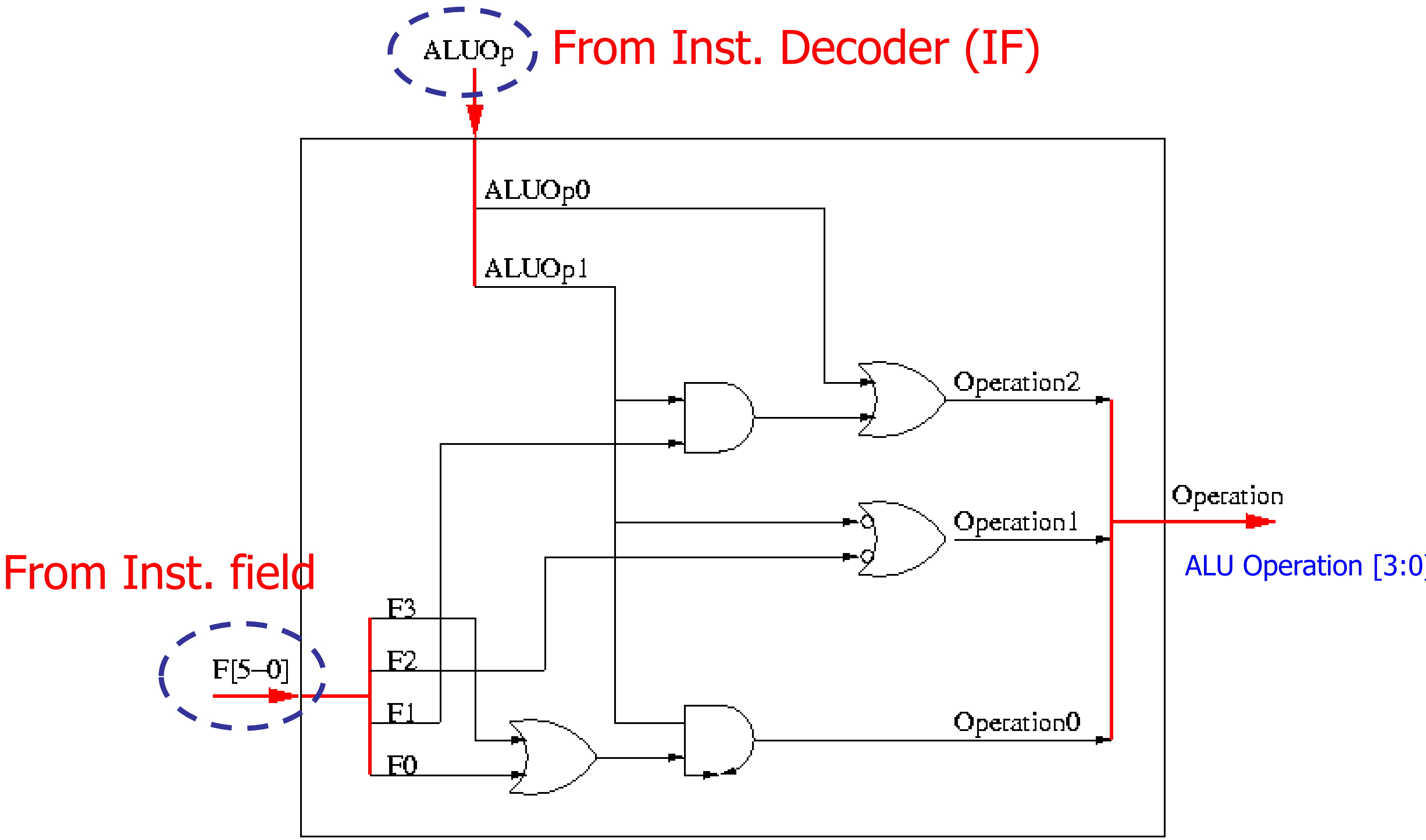
A Simple Implementation Scheme

- The truth table for the three ALU control bits (called Operation)

ALUOp (2bits)		Funct field (6bits)						ALU Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	1	0	0	0	0	0	0010
1	0	1	0	0	0	1	0	0110
1	0	1	0	0	1	0	0	0000
1	0	1	0	0	1	0	1	0001
1	0	1	0	1	0	1	0	0111

Simplify the ALU Control Design

- ALU control logic (overall)



Designing the main control unit

Review of Instruction Format

- The two instruction classes
  - R-type:

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

  - I-type:

31–26	25–21	20–16	15–0
op	rs	rt	immediate/offset
6 bits	5 bits	5 bits	16 bits

- Observations:
  - op field: opcode (bit[31:26], which is **called Op[5:0].**
  - The two registers to be read are specified by **rs & rt** (for R-type, beq).
  - Base register (for lw, sw) is **rs**
  - 16-bit offset (for lw, sw, beq) is bit[15:0] (also immediate values)
  - The destination register is in one of the two places:
    - lw : rt, bit[20:16]
    - R-type : rd, bit[15:11]

The Main Control Unit

- Control signals derived from instruction

R-type

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

Load/ Store

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

Branch

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

$\underbrace{\text{31:26}}_{\text{opcode}}$ $\underbrace{\text{25:21}}_{\text{always read}}$ $\underbrace{\text{20:16}}_{\text{read, except for load (lw)}}$

opcode

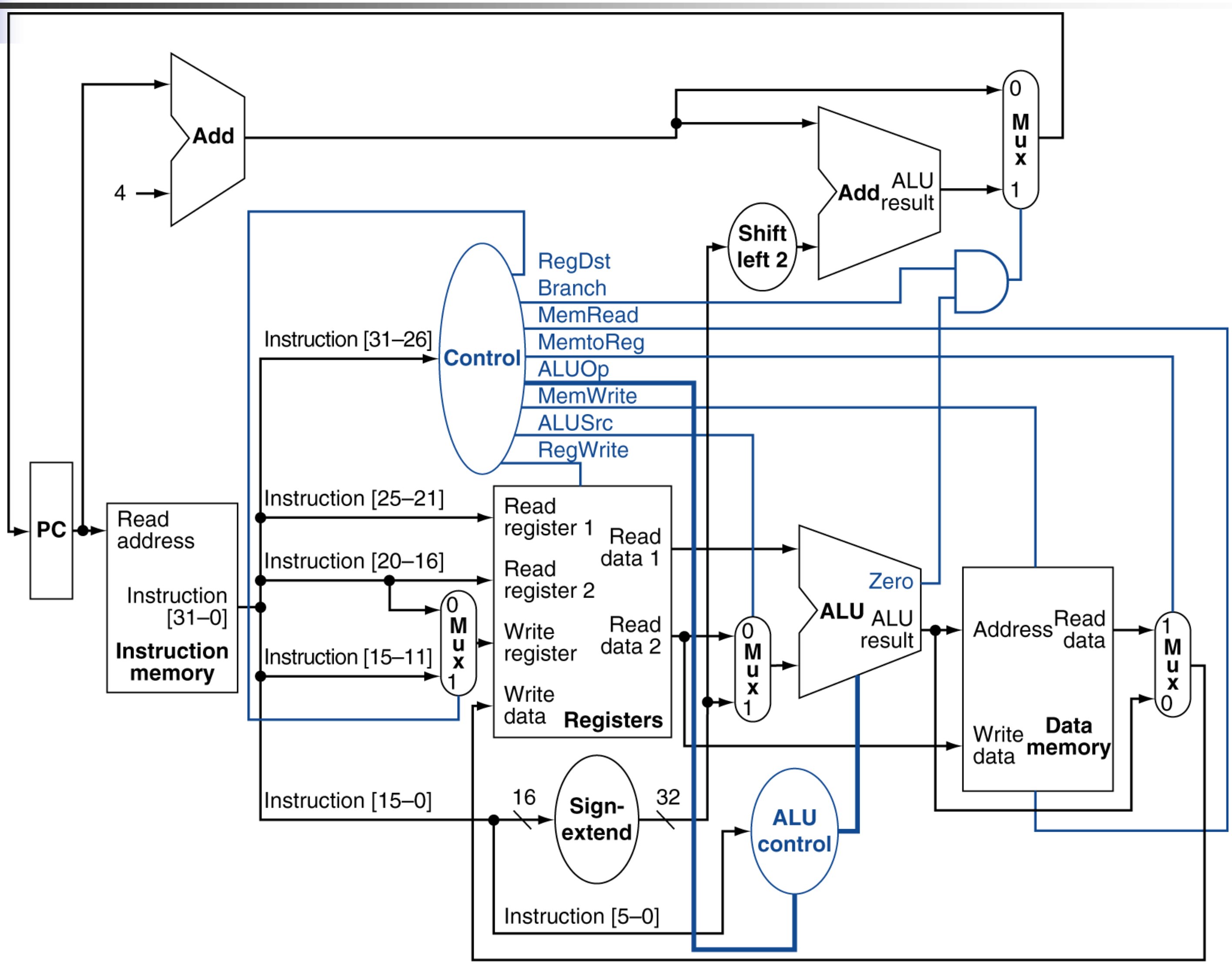
always read

read, except for load **(lw)**

write for R-type and load **(lw)**

sign-extend and add

Datapath With Control Signals



Effect of the 7 control signals

Signal name	Effect when deasserted(0)	Effect when asserted(1)
RegDst	The register destination number for the Write register comes from the rf field (bits20-16).	The register destination number for the Write register comes from the rd field (bits15-11).
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-extend, 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.

Operation for R-type instruction

- The 4 steps of the operation for R-type instruction

  - **R-type:**

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

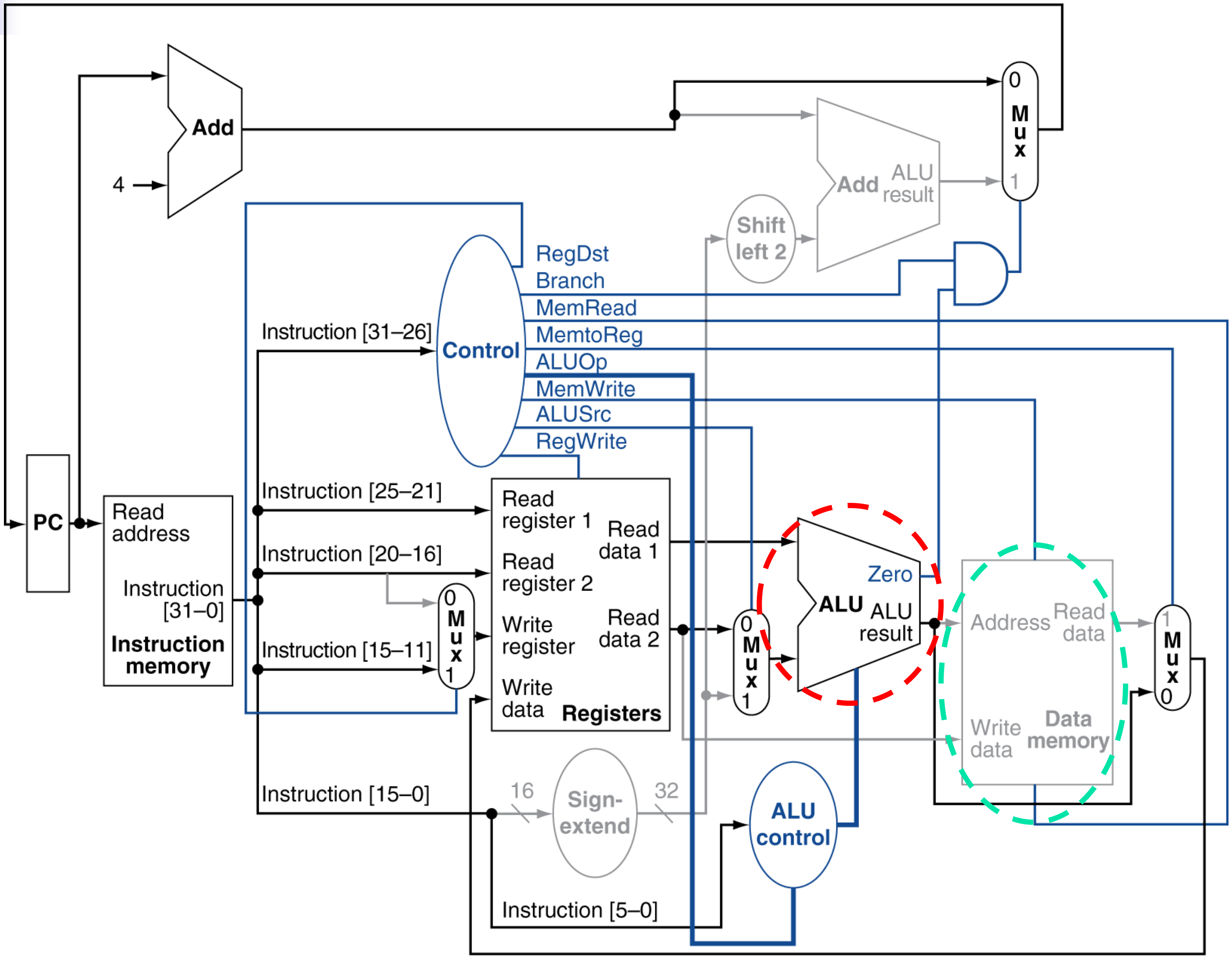
add $t1, $t2, $t3

- **Fetch** instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)

- **Read** registers (Reg1=Reg[rs], Reg2=Reg[rt])
- Run the **ALU** operation (Result = Reg1 ALUop Reg2)
- **Store** the result into Register File (Reg[rd] = Result)

Datapath for R-Type Instruction



Operation for “load” instruction

- The 5 steps of the operation for “load” instruction
  - I-type:

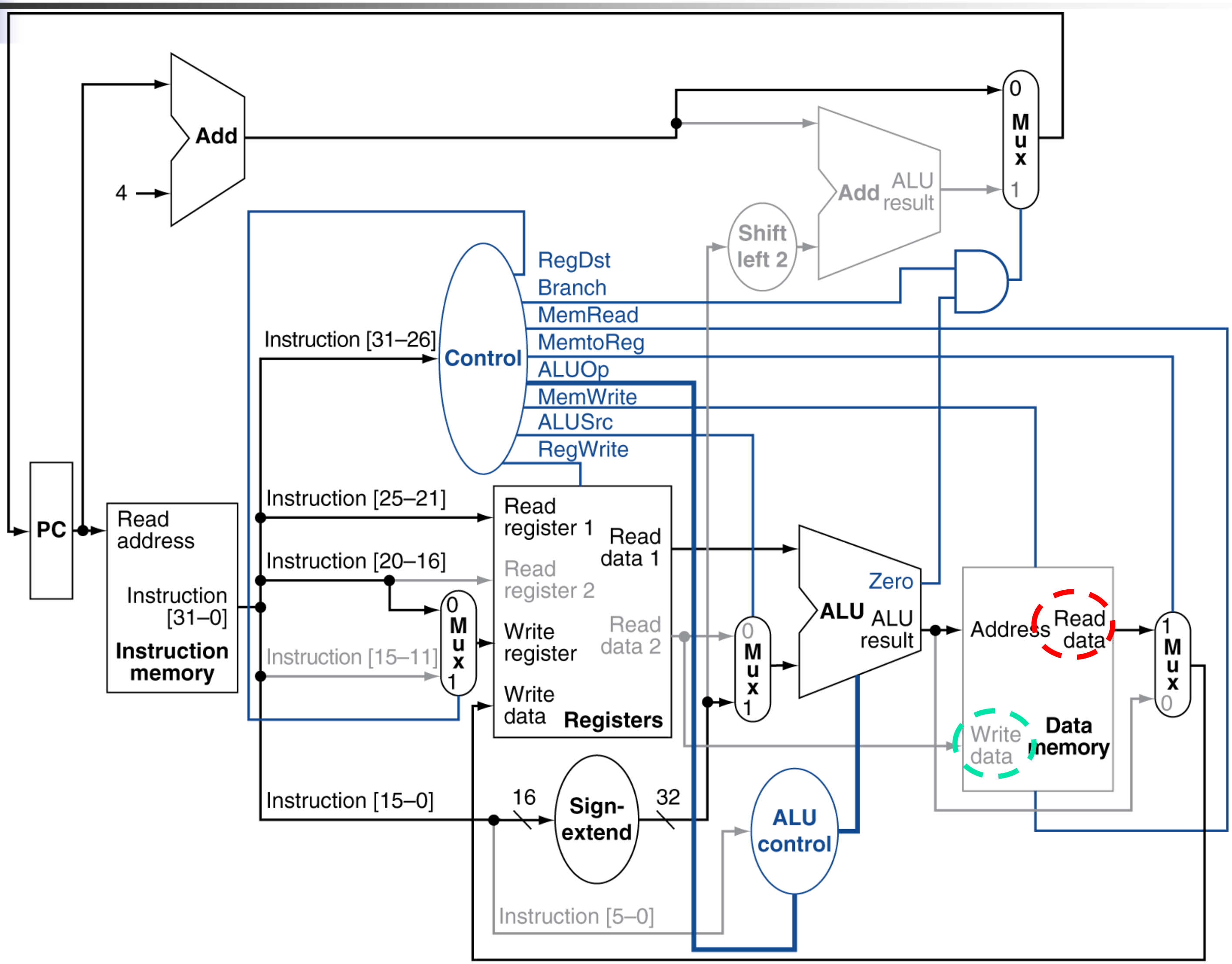
31 26	21	16	0
op	rs	rt	immediate/offset
6 bits	5 bits	5 bits	16 bits

lw $t1, offset($t2)

- **Fetch** instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)
- **Read** registers (temp = Reg[rs] , only one register is read)
- **Address computing** (Result = temp + sign-extend(Instr[15-0]))
- **Load** data from memory (Data = Memory[Result])
- **Store** data into Register File (Reg[rt] = Data)

Datapath for Load Instruction



Operation for “store” instruction

- The 4 steps of the operation for “store” instruction
  - I-type:

31–26	25–21	20–16	15–0
op	rs	rt	immediate/offset
6 bits	5 bits	5 bits	16 bits

sw $t1, offset($t2)

- **Fetch** instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)
- **Read** two registers (Reg1=Reg[rs], Reg2=Reg[rt])
- **Address computing** (Result = Reg1 + sign-extend(Instr[15-0]))
- **Store data** into memory (Memory[Result] = Reg2)

Operation for “beq” instruction

- The 3 steps of the operation for “beq” instruction
  - I-type:

31–26	25–21	20–16	15–0
op	rs	rt	immediate /offset
6 bits	5 bits	5 bits	16 bits

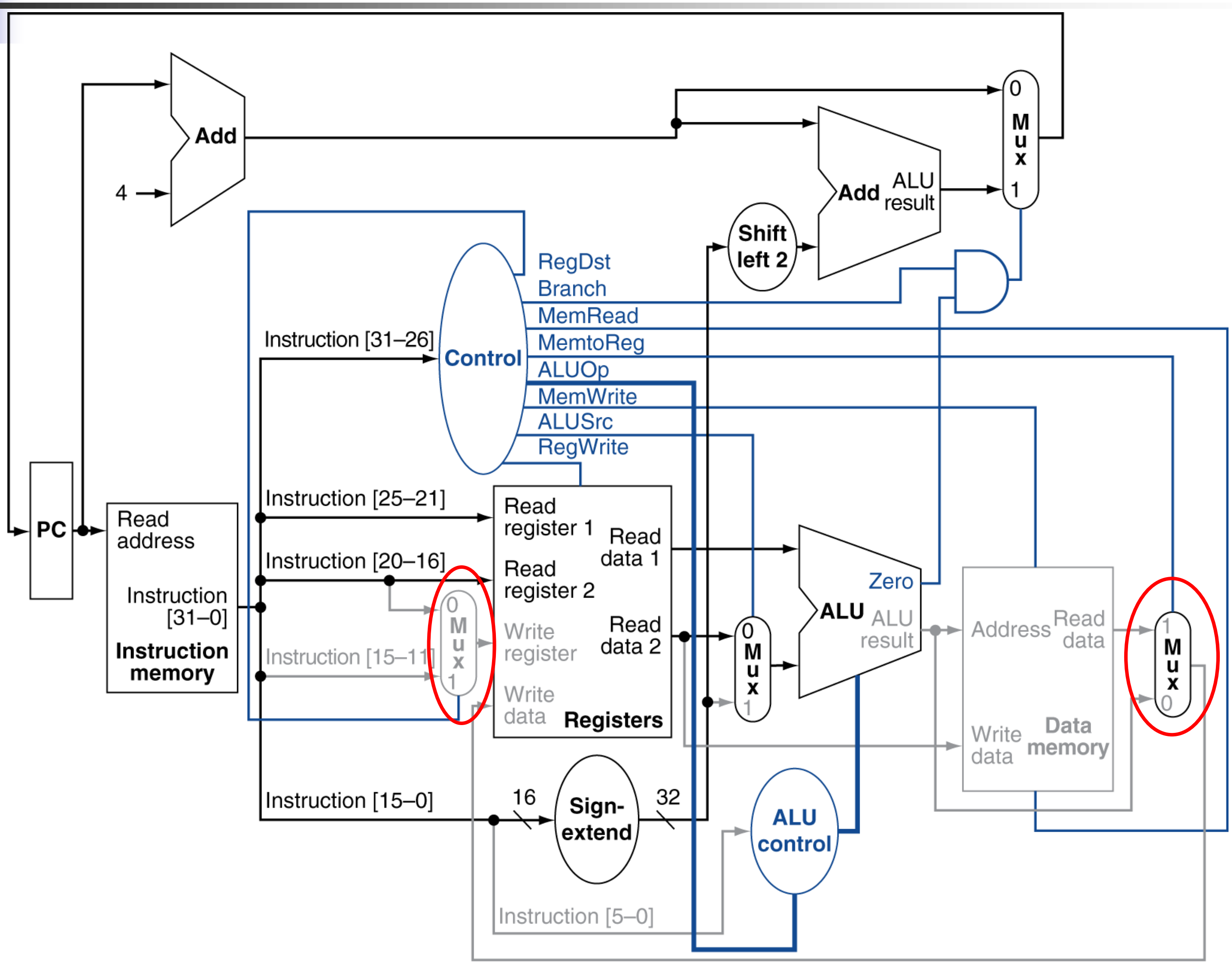
beq $t1, $t2, offset

- **Fetch** instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)
- **Read** two registers (Reg1=Reg[rs], Reg2=Reg[rt])
  - Compute **branch target address** (Result = PC + (sign-extend (Instr[15-0] << 2)))
  - Run the ALU operation (Result = Reg1 minus Reg2)
- **Observe “zero”** to branch or not

  (If zero==1, then PC = Result. Otherwise, PC unchanged)

Branch-on-Equal Instruction



Control Unit Design

- The setting of the control lines is completed by the “opcode” field (op[5:0]) of the instruction.

Op<5-0>	Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
000000	R-format	1	0	0	1	0	0	0	1	0
100011	lw	0	1	1	1	1	0	0	0	0
101011	sw	X	1	X	0	0	1	0	0	0
000100	beq	X	0	X	0	0	0	1	0	1

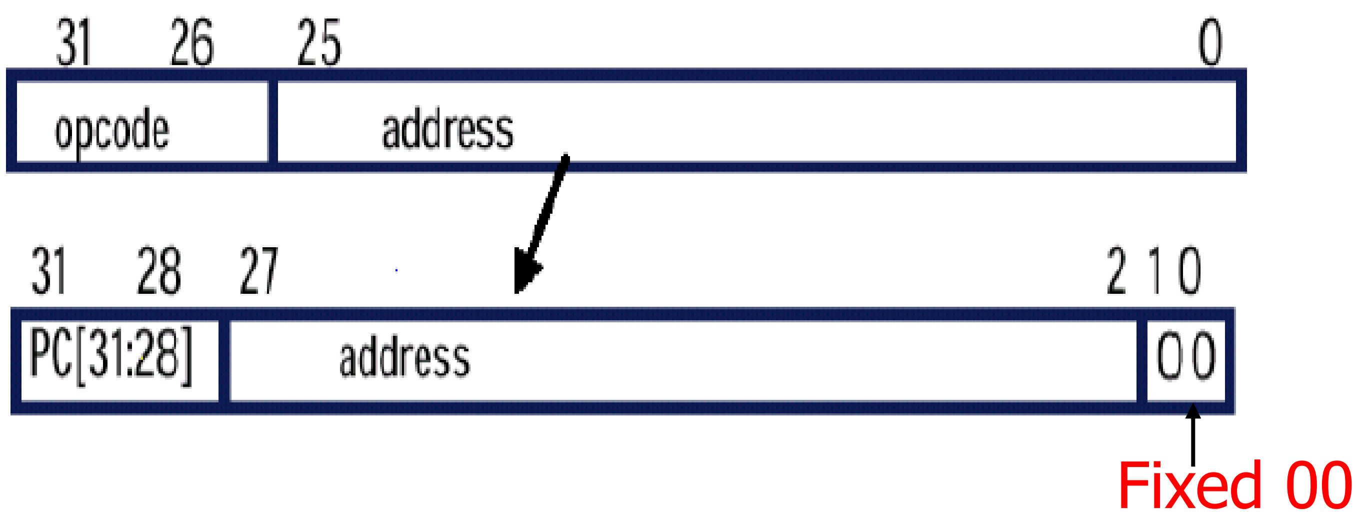
Note this table can be further simplified. (e.g. Branch is the same as ALUop0)

Finalizing the control signals

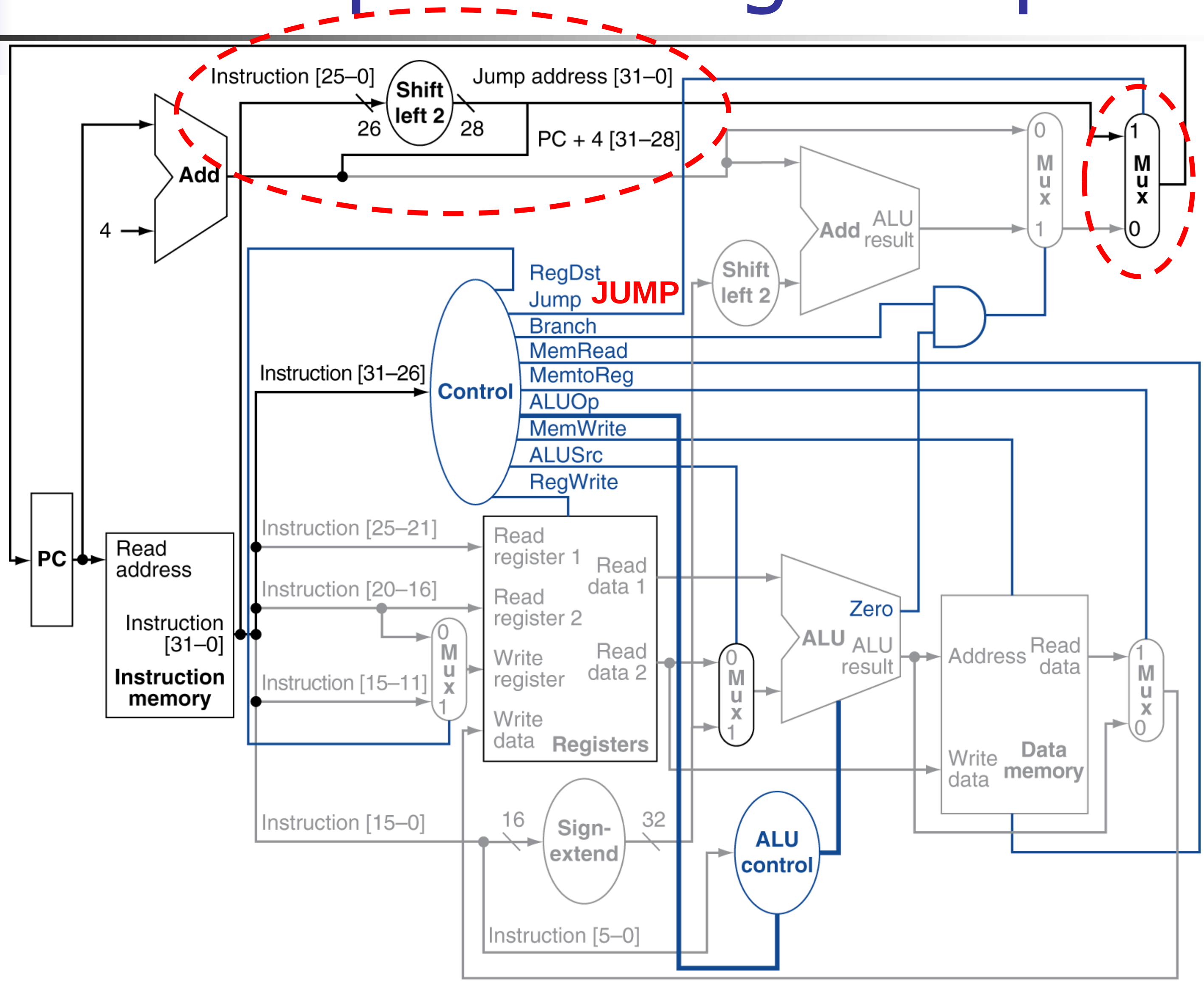
Input/output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Datapath for "Jump"

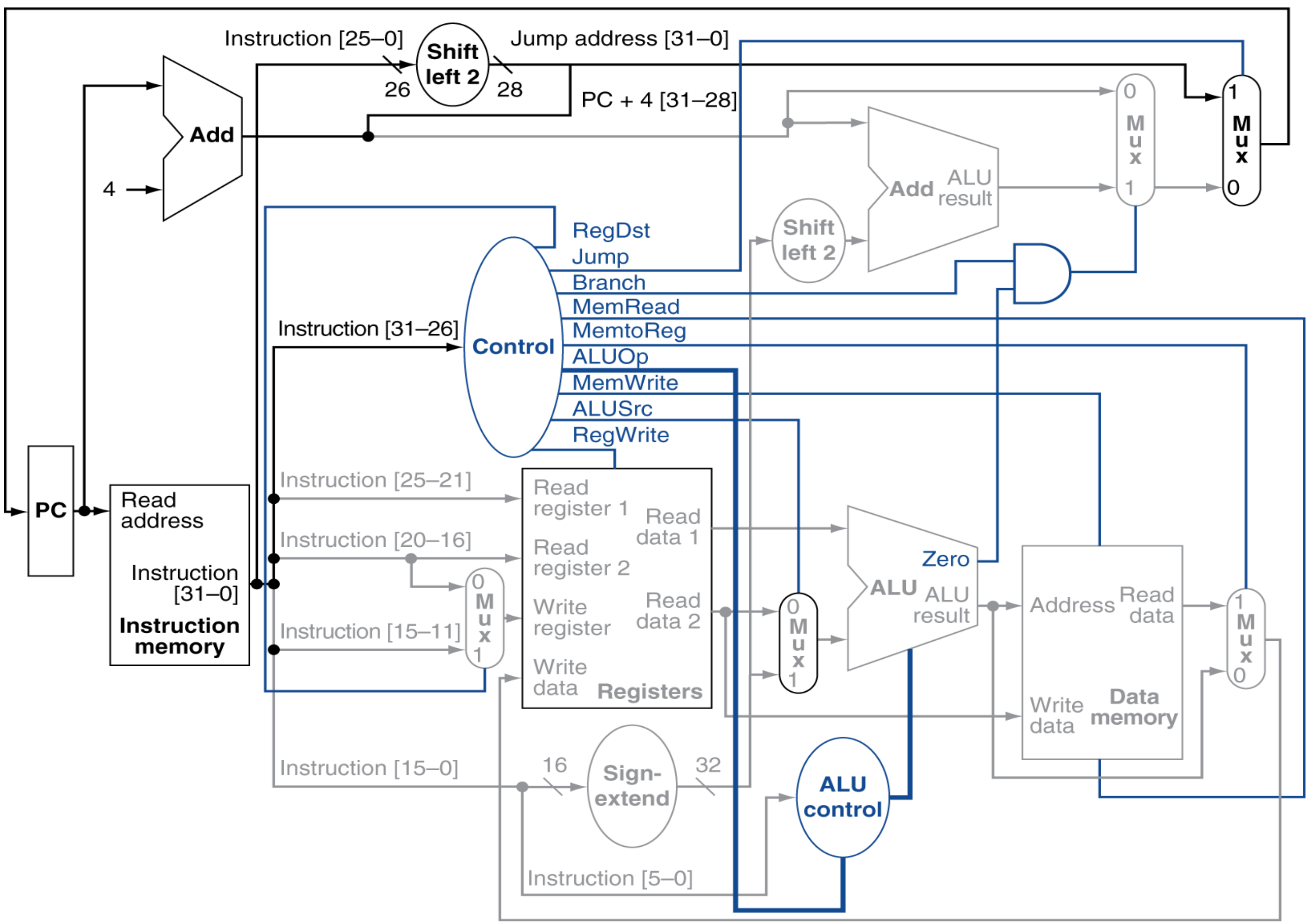
- "Jump" operation: (opcode = 000010)
  - Replace a portion of the PC(bit 27-0) with the lower 26 bits of the instruction shifted left by 2 bits.
  - The shift operation is accomplished by simple concatenating "00" to the jump offest.



Implementing "Jumps"



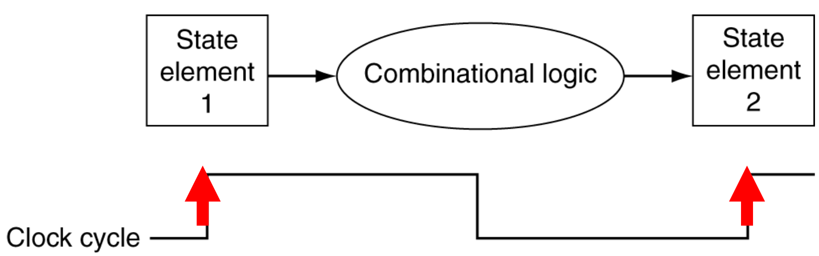
Single-cycle MIPS Implementation with 4 Types of Instructions (R, I, Branch, J)

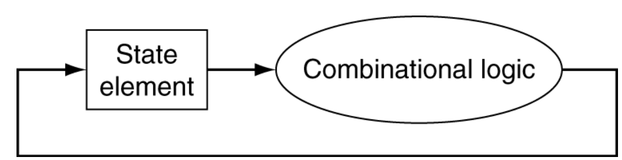


Clocking (recall)

- Combinational logic transforms data during clock cycles
  - An **Edge-triggered** methodology
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

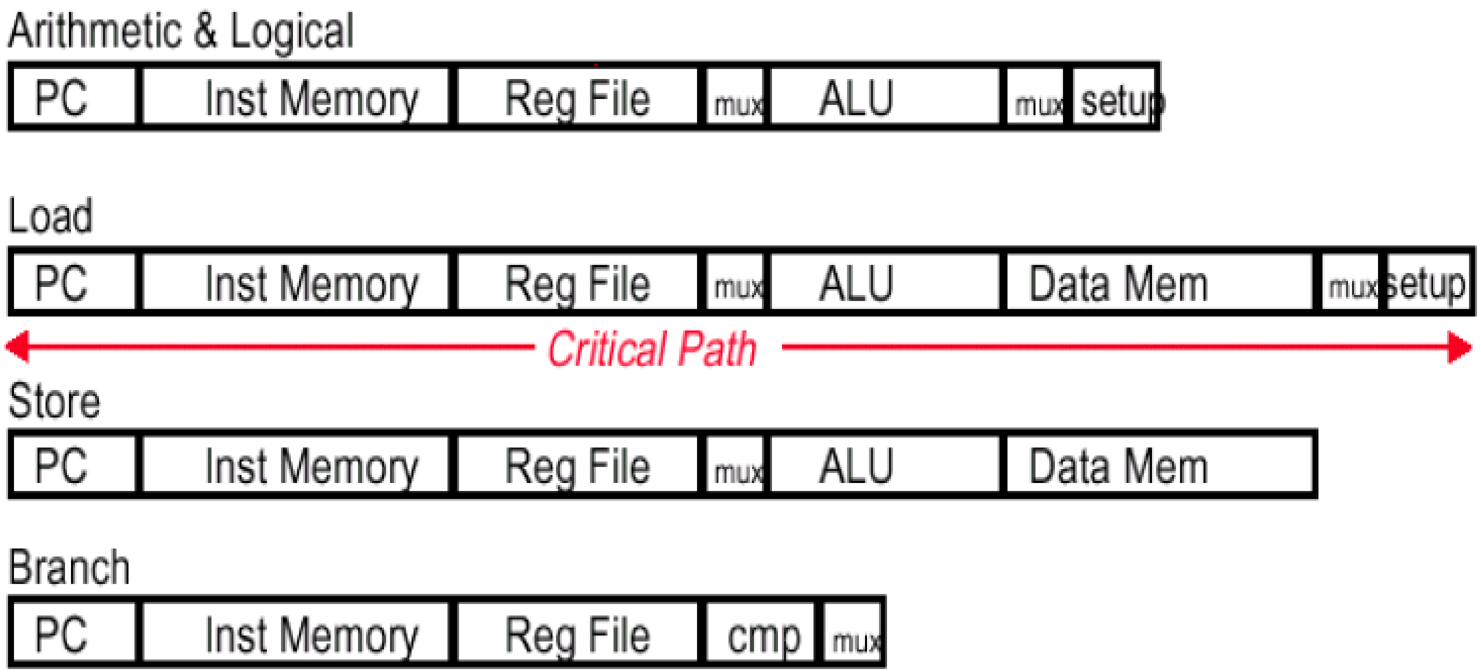
  - Typical execution:
    - Read contents of some state elements,
    - Send values through some combinational logic
    - Write results to one or more state elements





Single-cycle implementation

- Why a single-cycle implementation isn't used today?



  - Long cycle time for each instruction (load takes longest time)
  - All instructions take as much time as the slowest one

Performance of single-cycle implementation

- Example:
  - Assumption:
    - Memory units: 200 ps
    - ALU and adders: 100 ps
    - Register file (read / write): 50 ps
    - Multiplexers, control unit, PC accesses, sign extension unit, and wires have no delay.
    - The **instruction mix**: 25% loads, 10% stores, 45% ALU instructions, 15% branches, 5% jumps.

  - Problem: which one would be faster and by how much?

    (1) Fixed clock cycle (based on critical path)

    (2) Variable-length clock cycle (each instr. has its own clock period)

Performance of single-cycle implementation

- Answer:
  - The critical path for the different instruction classes:

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

  - Compute the require length for each instruction class:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Performance of single-cycle implementation

- Calculation equations:
  - CPU execution time = instruction count * CPI * clock cycle time
  - Assume CPI=1, CPU execution time = instruction count * clock cycle time
- Calculate CPU execution time :

  (1) Fixed clock cycle : 600 ps

  (2) Variable-length clock cycle (based on instruction mix) :

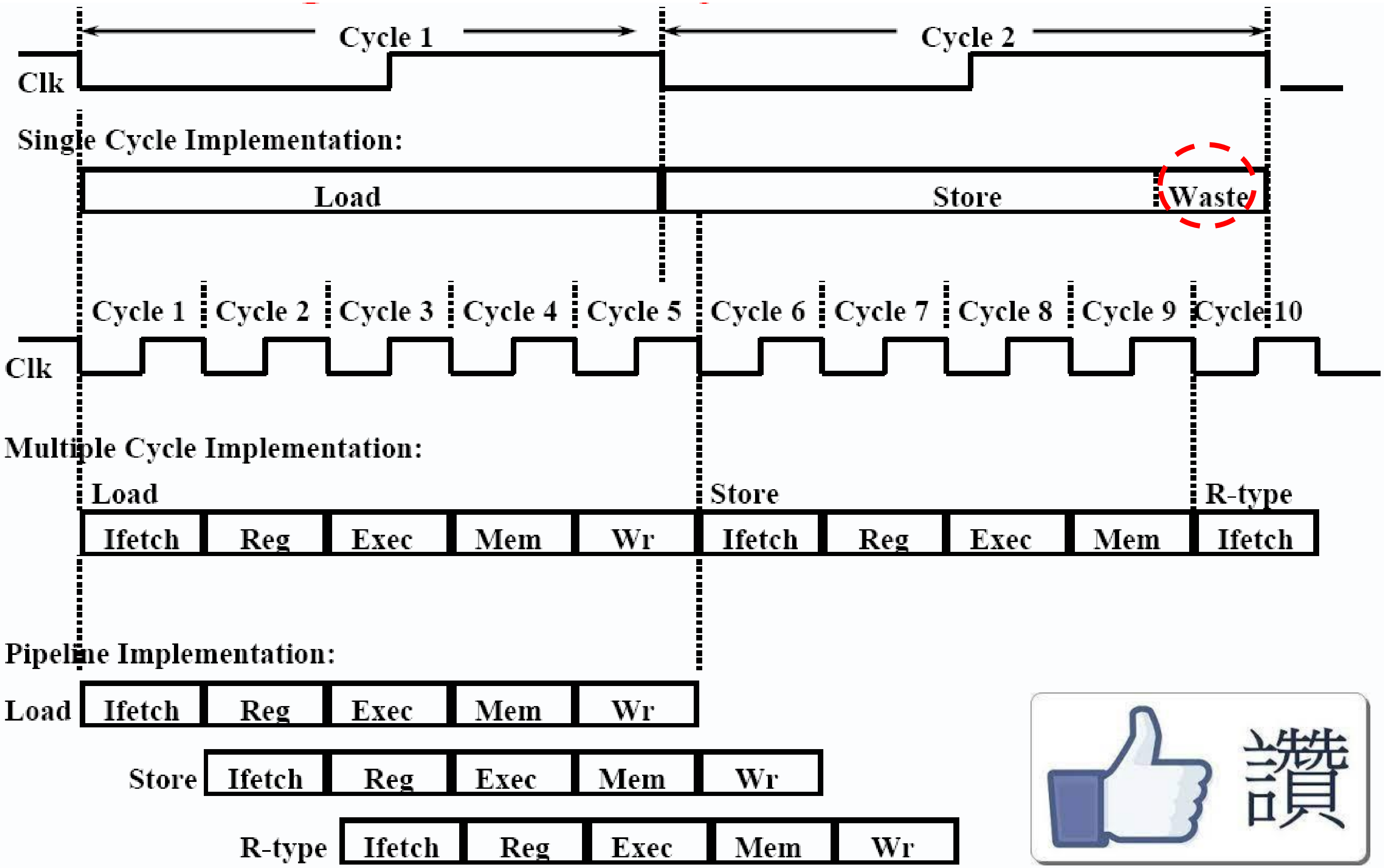
  600*25% + 550*10% + 400*45% + 350*15% + 200* 5%
  = 447.5 ps

  -- The one with variable-length clock cycle is faster.

- Performance ratio:

$$\frac{\text{CPU clock cycle (fixed)}}{\text{CPU clock cycle (variable)}} = \frac{600}{447.5} = 1.34$$

Single-, Multi-cycle, vs. Pipeline



Interrupt and Exception

Interrupt and Exception (Chap.4.9)

- Interrupts were initially created to handle unexpected events like **arithmetic overflow** and to **signal requests for service from I/O devices.**
- Some events generated internally or externally:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Interrupt and Exception (Chap.4.9)

- ***Exception:*** any unexpected change in control flow without distinguishing whether the cause is internal or external
- ***Interrupt:*** only when the event is externally caused
- We will only discuss how to handle ***an undefined instruction*** or an ***arithmetic overflow*** in this chapter.

- How exceptions are handled:
  - Save the address of the offending instruction in the ***Exception Program Counter (EPC)*** and transfer control to the operating system at some specified address.
  - Take some predefined action in response to an overflow, or stop the execution of the program and report an error (Execute ***Interrupt Service Routine, ISR***)
  - Terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.

Handling Exceptions in MIPS

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)

1. Save PC of offending (or interrupted) instruction
    - In MIPS: Exception Program Counter (EPC)
2. Save indication of the problem
    - In MIPS: **Cause register**
    - We'll assume 1-bit
        - 0 for undefined opcode, 1 for overflow
3. Jump to handler at 8000 00180

Vectored Interrupts

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode (addr.): C000 0000
  - Overflow (address): C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required

- If restartable
  - Take corrective action
  - use **EPC** to return to program
- Otherwise
  - **Terminate** program
  - Report error using **EPC, cause,** …

Interrupt Registers

Two main methods used to communicate the reason for an exception:

1. **Cause register: A status register** which holds a field that indicates the reason for the exception (used in MIPS architecture)
2. **Vectored interrupt:** An interrupt for which the address to which control is transferred is determined by the cause of the exception.

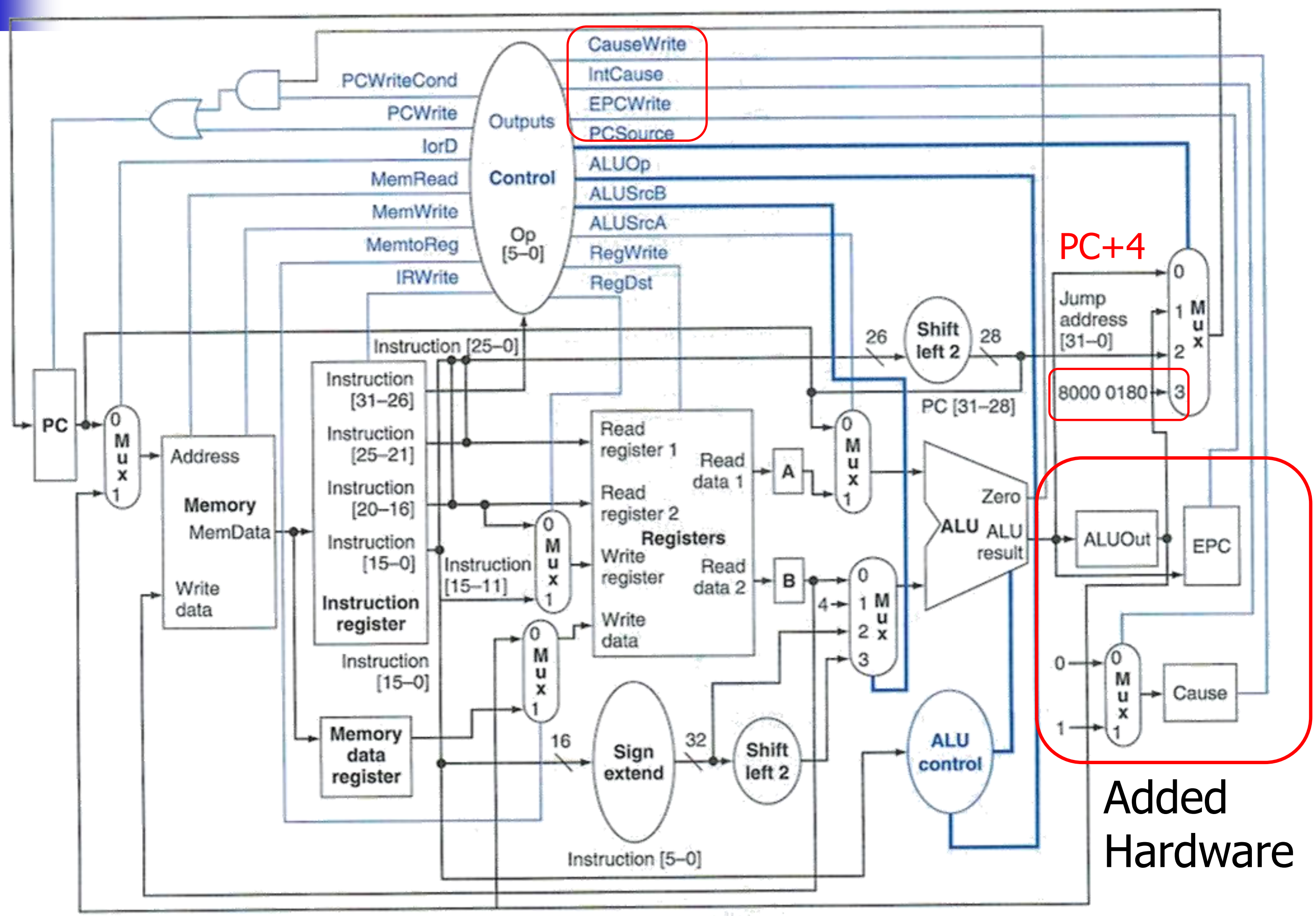
Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

- The operating system knows the reason for the exception by the address at which it is initiated.
- The address are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence.

Support Exception in MIPS CPU

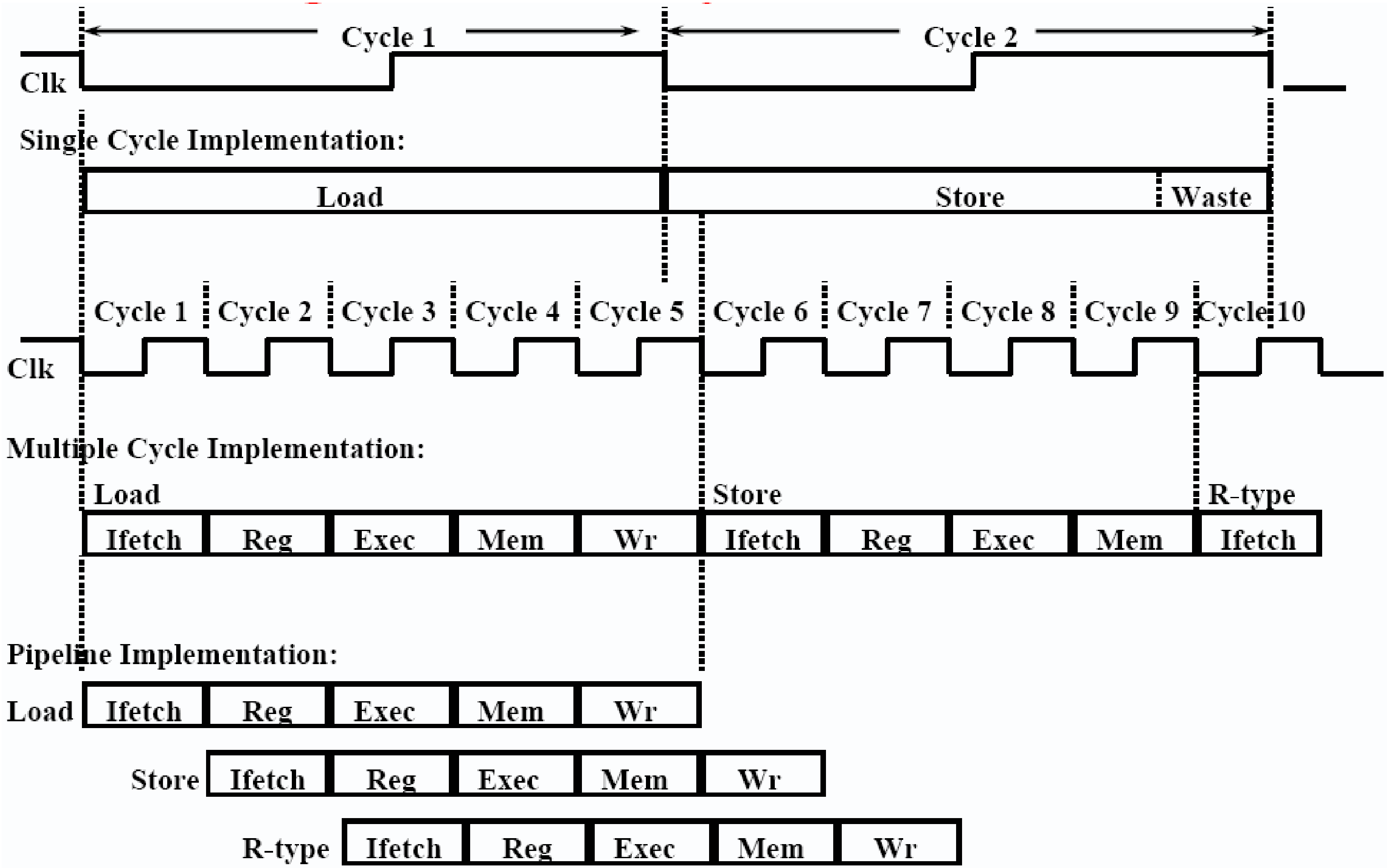
- For MIPS exception system
  - Two additional registers to the datapath:
    1. **EPC (exception program counter):** A 32-bit register used to hold the address of the affected instruction.
    2. **Cause:** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits.
  - 3 Additional control signals:
    - *EPCWrite* (update the problem instruction address)
    - *CauseWrite* (update the Cause number)
    - *IntCause*
  - Change the 3-way multiplexor (controlled by PCSouse) to a 4-way multiplexor, with additional input wired to the constant value **8000 0180**$_{\text{hex}}$ → The Operating system entry point for exception handling subroutines (for arithmetic overflow)

Support Exception in Multi-Cycle MIPS CPU



Added Hardware

Single-, Multi-cycle, vs. Pipeline



Summary

- Single-cycle and multi-cycle RISC CPU designs are introduced.
- Can add new instructions is easy (*e.g.,* add **jr, jal, addi, subi, slt, slti, sltui**) by adding/changing datapath units and control signals.
- Good for complex Verilog programming (e.g., Digital System Design (DSD) Course)
- Will be enhanced by **pipelined structure.**