

# Single Cycle Processor Design

M S Bhat

Dept. of E&C,  
NITK Suratkal

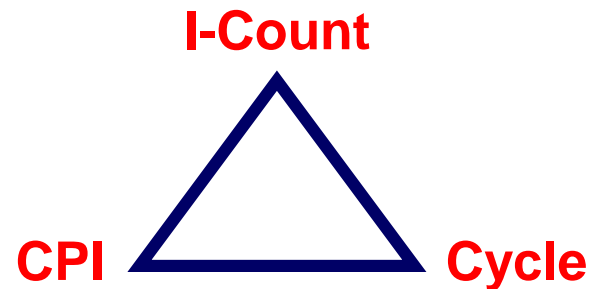
# Outline

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

# The Performance Perspective

## ❖ Recall, performance is determined by:

- ✧ Instruction count
- ✧ Clock cycles per instruction (CPI)
- ✧ Clock cycle time



## ❖ Processor design will affect

- ✧ Clock cycles per instruction
- ✧ Clock cycle time

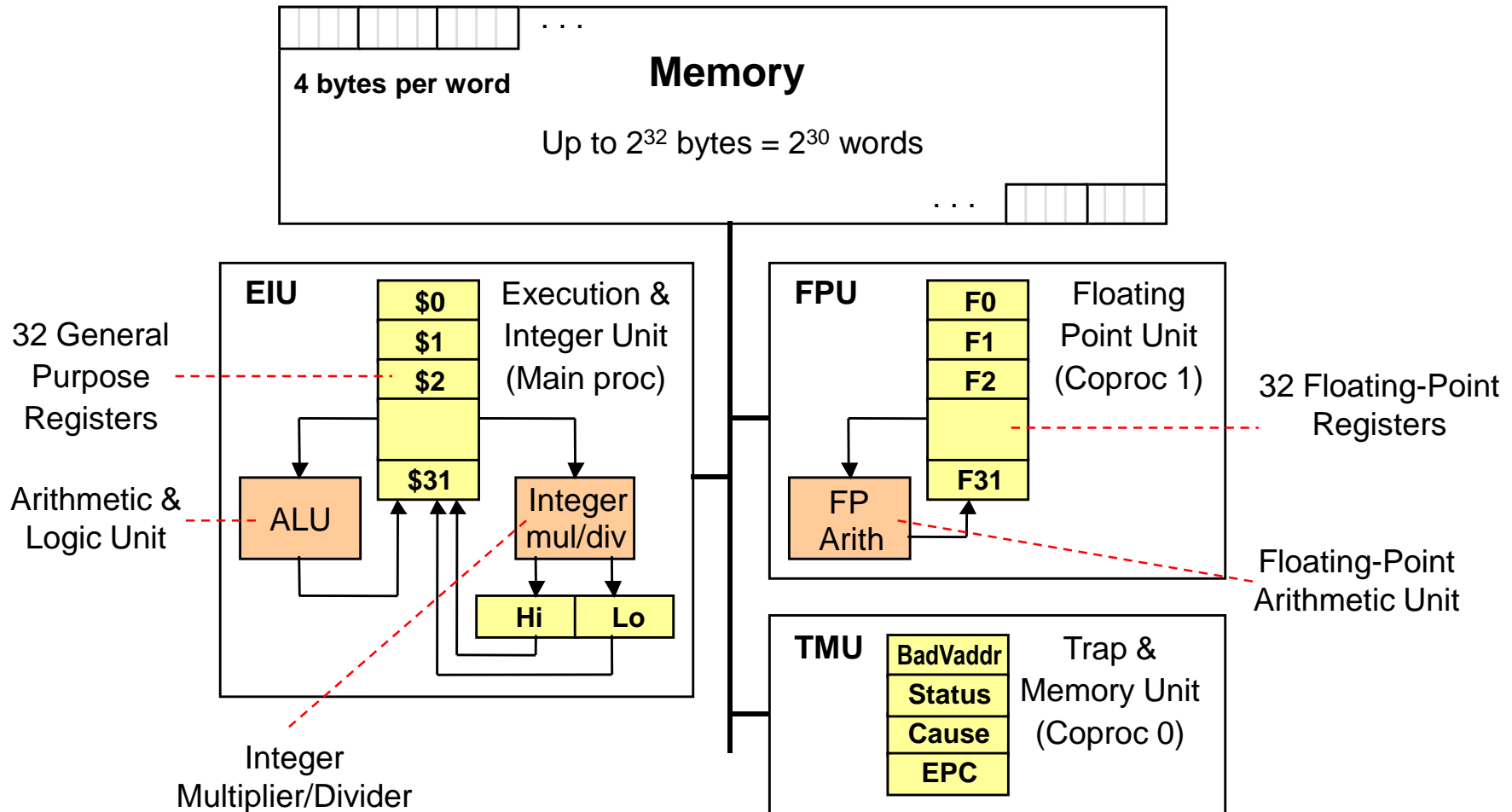
## ❖ Single cycle datapath and control design:

- ✧ Advantage: One clock cycle per instruction
- ✧ Disadvantage: long cycle time

# Designing a Processor: Step-by-Step

- ❖ Analyze instruction set => **datapath requirements**
  - ✧ The meaning of each instruction is given by the **register transfers**
  - ✧ Datapath must include storage elements for ISA registers
  - ✧ Datapath must support each register transfer
- ❖ Select **datapath components** and **clocking methodology**
- ❖ Assemble **datapath** meeting the requirements
- ❖ Analyze implementation of **each instruction**
  - ✧ Determine the setting of **control signals** for register transfer
- ❖ Assemble the **control logic**

# Overview of the MIPS Architecture



# MIPS General-Purpose Registers

## ❖ 32 General Purpose Registers (GPRs)

- ✧ Assembler uses the dollar notation to name registers
  - \$0 is register 0, \$1 is register 1, ..., and \$31 is register 31
- ✧ All registers are 32-bit wide in MIPS32
- ✧ Register \$0 is always zero
  - Any value written to \$0 is discarded

## ❖ Software conventions

- ✧ There are many registers (32)
- ✧ Software defines names to all registers
  - To standardize their use in programs
- ✧ Example: \$8 - \$15 are called \$t0 - \$t7
  - Used for **temporary** values

|              |             |
|--------------|-------------|
| \$0 = \$zero | \$16 = \$s0 |
| \$1 = \$at   | \$17 = \$s1 |
| \$2 = \$v0   | \$18 = \$s2 |
| \$3 = \$v1   | \$19 = \$s3 |
| \$4 = \$a0   | \$20 = \$s4 |
| \$5 = \$a1   | \$21 = \$s5 |
| \$6 = \$a2   | \$22 = \$s6 |
| \$7 = \$a3   | \$23 = \$s7 |
| \$8 = \$t0   | \$24 = \$t8 |
| \$9 = \$t1   | \$25 = \$t9 |
| \$10 = \$t2  | \$26 = \$k0 |
| \$11 = \$t3  | \$27 = \$k1 |
| \$12 = \$t4  | \$28 = \$gp |
| \$13 = \$t5  | \$29 = \$sp |
| \$14 = \$t6  | \$30 = \$fp |
| \$15 = \$t7  | \$31 = \$ra |

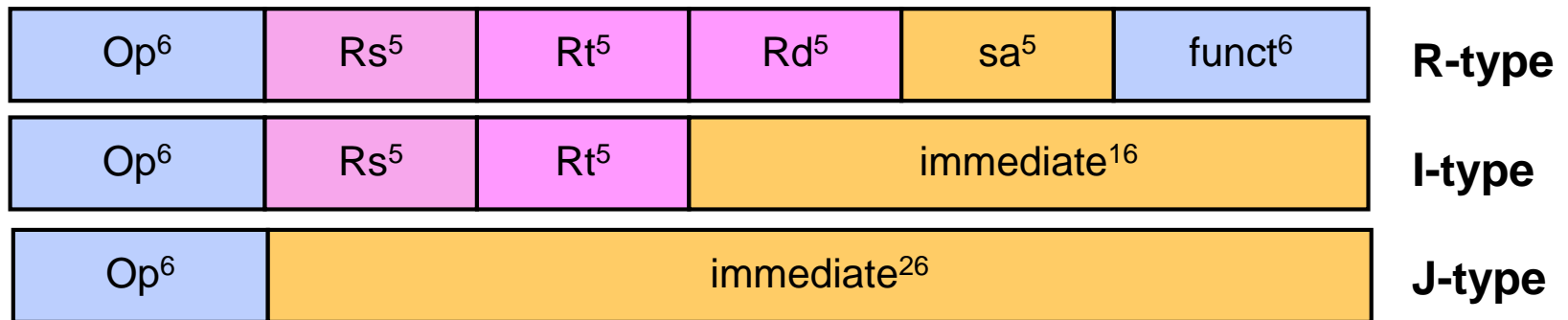
# MIPS Register Conventions

- ❖ Assembler can refer to registers by name or by number
  - ✧ It is easier to remember registers by name
  - ✧ Assembler converts register name to its corresponding number

| Name               | Register           | Usage  |
|--------------------|--------------------|--|
| <b>\$zero</b>      | <b>\$0</b>         | Always 0 (forced by hardware)                  |
| <b>\$at</b>        | <b>\$1</b>         | Reserved for assembler use                     |
| <b>\$v0 – \$v1</b> | <b>\$2 – \$3</b>   | Result values of a function                    |
| <b>\$a0 – \$a3</b> | <b>\$4 – \$7</b>   | Arguments of a function                        |
| <b>\$t0 – \$t7</b> | <b>\$8 – \$15</b>  | Temporary Values                               |
| <b>\$s0 – \$s7</b> | <b>\$16 – \$23</b> | Saved registers (preserved across call)        |
| <b>\$t8 – \$t9</b> | <b>\$24 – \$25</b> | More temporaries                               |
| <b>\$k0 – \$k1</b> | <b>\$26 – \$27</b> | Reserved for OS kernel                         |
| <b>\$gp</b>        | <b>\$28</b>        | Global pointer (points to global data)         |
| <b>\$sp</b>        | <b>\$29</b>        | Stack pointer (points to top of stack)         |
| <b>\$fp</b>        | <b>\$30</b>        | Frame pointer (points to stack frame)          |
| <b>\$ra</b>        | <b>\$31</b>        | Return address (used by jal for function call) |

# Review of MIPS Instruction Formats

- ❖ All instructions are **32-bit wide**
- ❖ Three instruction formats: **R-type**, **I-type**, and **J-type**



- ✧ Op<sup>6</sup>: 6-bit opcode of the instruction
- ✧ Rs<sup>5</sup>, Rt<sup>5</sup>, Rd<sup>5</sup>: 5-bit source and destination register numbers
- ✧ sa<sup>5</sup>: 5-bit **shift amount** used by shift instructions
- ✧ funct<sup>6</sup>: 6-bit **function** field for R-type instructions
- ✧ immediate<sup>16</sup>: 16-bit immediate value or address offset
- ✧ immediate<sup>26</sup>: 26-bit target address of the jump instruction



# MIPS Subset of Instructions

- ❖ Only a subset of the MIPS instructions are considered
  - ✧ ALU instructions (R-type): **add, sub, and, or, xor, slt**
  - ✧ Immediate instructions (I-type): **addi, slti, andi, ori, xori**
  - ✧ Load and Store (I-type): **lw, sw**
  - ✧ Branch (I-type): **beq, bne**
  - ✧ Jump (J-type): **j**
- ❖ This subset does not include all the integer instructions
- ❖ But sufficient to illustrate design of datapath and control
- ❖ Concepts used to implement the MIPS subset are used to construct a broad spectrum of computers

# Details of the MIPS Subset

| Instruction |                   | Meaning          | Format     |           |        |           |   |      |
|-------------|-------------------|------------------|------------|-----------|--------|-----------|---|------|
| add         | rd, rs, rt        | addition         | $op^6 = 0$ | $rs^5$    | $rt^5$ | $rd^5$    | 0 | 0x20 |
| sub         | rd, rs, rt        | subtraction      | $op^6 = 0$ | $rs^5$    | $rt^5$ | $rd^5$    | 0 | 0x22 |
| and         | rd, rs, rt        | bitwise and      | $op^6 = 0$ | $rs^5$    | $rt^5$ | $rd^5$    | 0 | 0x24 |
| or          | rd, rs, rt        | bitwise or       | $op^6 = 0$ | $rs^5$    | $rt^5$ | $rd^5$    | 0 | 0x25 |
| xor         | rd, rs, rt        | exclusive or     | $op^6 = 0$ | $rs^5$    | $rt^5$ | $rd^5$    | 0 | 0x26 |
| slt         | rd, rs, rt        | set on less than | $op^6 = 0$ | $rs^5$    | $rt^5$ | $rd^5$    | 0 | 0x2a |
| addi        | rt, rs, $im^{16}$ | add immediate    | 0x08       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| slti        | rt, rs, $im^{16}$ | slt immediate    | 0x0a       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| andi        | rt, rs, $im^{16}$ | and immediate    | 0x0c       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| ori         | rt, rs, $im^{16}$ | or immediate     | 0x0d       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| xori        | rt, $im^{16}$     | xor immediate    | 0x0e       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| lw          | rt, $im^{16}(rs)$ | load word        | 0x23       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| sw          | rt, $im^{16}(rs)$ | store word       | 0x2b       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| beq         | rs, rt, $im^{16}$ | branch if equal  | 0x04       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| bne         | rs, rt, $im^{16}$ | branch not equal | 0x05       | $rs^5$    | $rt^5$ | $im^{16}$ |   |      |
| j           | $im^{26}$         | jump             | 0x02       | $im^{26}$ |        |           |   |      |

# Register Transfer Level (RTL)

- ❖ RTL is a description of data flow between registers
- ❖ RTL gives a **meaning** to the instructions
- ❖ All instructions are fetched from memory at address PC

## Instruction      RTL Description

|     |   |                                      |
|-----|---|--------------------------------------|
| ADD | $\text{Reg(Rd)} \leftarrow \text{Reg(Rs)} + \text{Reg(Rt)};$  | $\text{PC} \leftarrow \text{PC} + 4$ |
| SUB | $\text{Reg(Rd)} \leftarrow \text{Reg(Rs)} - \text{Reg(Rt)};$  | $\text{PC} \leftarrow \text{PC} + 4$ |
| ORI | $\text{Reg(Rt)} \leftarrow \text{Reg(Rs)} \mid \text{zero\_ext(Im16)};$   | $\text{PC} \leftarrow \text{PC} + 4$ |
| LW  | $\text{Reg(Rt)} \leftarrow \text{MEM}[\text{Reg(Rs)} + \text{sign\_ext(Im16)}];$  | $\text{PC} \leftarrow \text{PC} + 4$ |
| SW  | $\text{MEM}[\text{Reg(Rs)} + \text{sign\_ext(Im16)}] \leftarrow \text{Reg(Rt)};$  | $\text{PC} \leftarrow \text{PC} + 4$ |
| BEQ | if ( $\text{Reg(Rs)} == \text{Reg(Rt)}$ )<br>$\text{PC} \leftarrow \text{PC} + 4 + 4 \times \text{sign\_extend(Im16)}$<br>else $\text{PC} \leftarrow \text{PC} + 4$ |                                      |

# Instructions are Executed in Steps

- ❖ **R-type**
  - Fetch instruction:  $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$
  - Fetch operands:  $\text{data1} \leftarrow \text{Reg}(\text{Rs}), \text{data2} \leftarrow \text{Reg}(\text{Rt})$
  - Execute operation:  $\text{ALU\_result} \leftarrow \text{func}(\text{data1}, \text{data2})$
  - Write ALU result:  $\text{Reg}(\text{Rd}) \leftarrow \text{ALU\_result}$
  - Next PC address:  $\text{PC} \leftarrow \text{PC} + 4$
- ❖ **I-type**
  - Fetch instruction:  $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$
  - Fetch operands:  $\text{data1} \leftarrow \text{Reg}(\text{Rs}), \text{data2} \leftarrow \text{Extend}(\text{imm16})$
  - Execute operation:  $\text{ALU\_result} \leftarrow \text{op}(\text{data1}, \text{data2})$
  - Write ALU result:  $\text{Reg}(\text{Rt}) \leftarrow \text{ALU\_result}$
  - Next PC address:  $\text{PC} \leftarrow \text{PC} + 4$
- ❖ **BEQ**
  - Fetch instruction:  $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$
  - Fetch operands:  $\text{data1} \leftarrow \text{Reg}(\text{Rs}), \text{data2} \leftarrow \text{Reg}(\text{Rt})$
  - Equality:  $\text{zero} \leftarrow \text{subtract}(\text{data1}, \text{data2})$
  - Branch:  $\text{if (zero) } \text{PC} \leftarrow \text{PC} + 4 + 4 \times \text{sign\_ext}(\text{imm16})$   
 $\text{else } \text{PC} \leftarrow \text{PC} + 4$

# Instruction Execution - cont'd

## ❖ LW

|                      |   |
|----------------------|---|
| Fetch instruction:   | $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$                       |
| Fetch base register: | $\text{base} \leftarrow \text{Reg}(\text{Rs})$                              |
| Calculate address:   | $\text{address} \leftarrow \text{base} + \text{sign\_extend}(\text{imm16})$ |
| Read memory:         | $\text{data} \leftarrow \text{MEM}[\text{address}]$                         |
| Write register Rt:   | $\text{Reg}(\text{Rt}) \leftarrow \text{data}$                              |
| Next PC address:     | $\text{PC} \leftarrow \text{PC} + 4$  |

## ❖ SW

|                    |  |
|--------------------|--|
| Fetch instruction: | $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$  |
| Fetch registers:   | $\text{base} \leftarrow \text{Reg}(\text{Rs}), \text{data} \leftarrow \text{Reg}(\text{Rt})$ |
| Calculate address: | $\text{address} \leftarrow \text{base} + \text{sign\_extend}(\text{imm16})$                  |
| Write memory:      | $\text{MEM}[\text{address}] \leftarrow \text{data}$  |
| Next PC address:   | $\text{PC} \leftarrow \text{PC} + 4$   |

## ❖ Jump

|                    |   |
|--------------------|---|
| Fetch instruction: | $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$           |
| Target PC address: | $\text{target} \leftarrow \text{PC}[31:28], \text{Imm26}, '00'$ |
| Jump:              | $\text{PC} \leftarrow \text{target}$                            |

concatenation

# Requirements of the Instruction Set

## ❖ Memory

- ❖ **Instruction memory** where instructions are stored
- ❖ **Data memory** where data is stored

## ❖ Registers

- ❖ **32 × 32-bit general purpose registers**, R0 is always zero
- ❖ Read source register Rs
- ❖ Read source register Rt
- ❖ Write destination register Rt or Rd

## ❖ Program counter **PC register** and **Adder** to increment PC

## ❖ Sign and Zero **extender** for immediate constant

## ❖ **ALU** for executing instructions

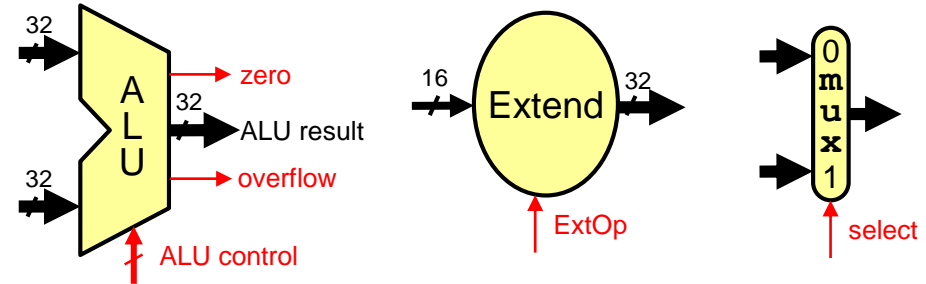
# Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

# Components of the Datapath

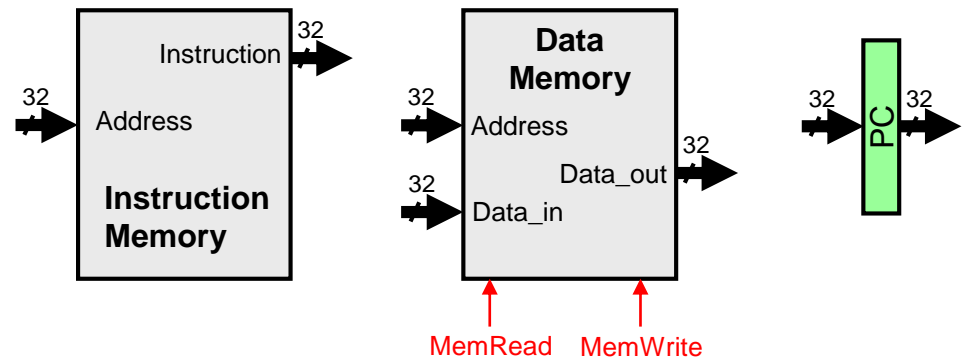
## ❖ Combinational Elements

- ✧ ALU, Adder
- ✧ Immediate extender
- ✧ Multiplexers



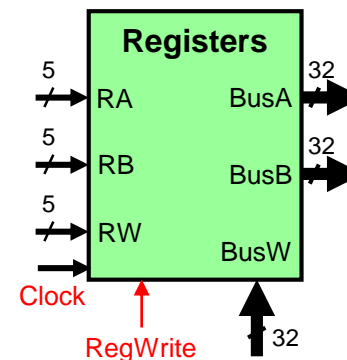
## ❖ Storage Elements

- ✧ Instruction memory
- ✧ Data memory
- ✧ PC register
- ✧ Register file



## ❖ Clocking methodology

- ✧ Timing of reads and writes





# Register Element

## ❖ Register

- ✧ Similar to the D-type Flip-Flop

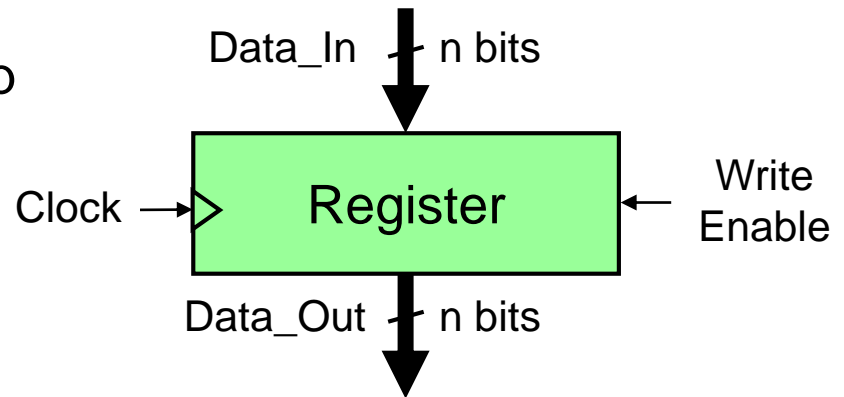
## ❖ n-bit input and output

## ❖ Write Enable:

- ✧ Enable / disable writing of register
- ✧ Negated (0): Data\_Out will not change
- ✧ Asserted (1): Data\_Out will become Data\_In **after clock edge**

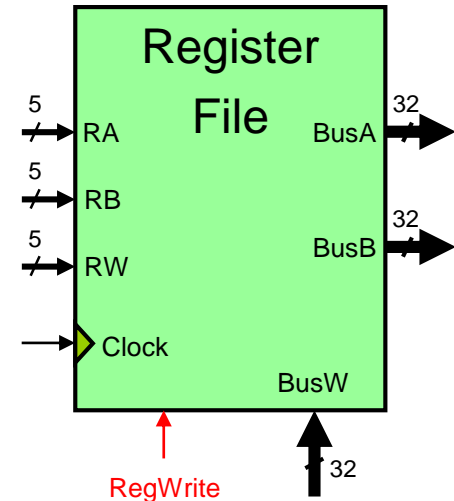
## ❖ Edge triggered Clocking

- ✧ Register output is modified at **clock edge**



# MIPS Register File

- ❖ Register File consists of  $32 \times 32$ -bit registers
  - ✧ **BusA** and **BusB**: 32-bit output busses for reading 2 registers
  - ✧ **BusW**: 32-bit input bus for writing a register when **RegWrite** is 1
  - ✧ Two registers read and one written in a cycle
- ❖ Registers are selected by:
  - ✧ **RA** selects register to be **read** on **BusA**
  - ✧ **RB** selects register to be **read** on **BusB**
  - ✧ **RW** selects the register to be **written**
- ❖ Clock input
  - ✧ The clock input is **used ONLY during write** operation
  - ✧ During read, register file behaves as a **combinational logic** block
    - RA or RB valid  $\Rightarrow$  BusA or BusB valid after **access time**



# Tri-State Buffers

- ❖ Allow multiple sources to drive a single bus

- ❖ Two Inputs:

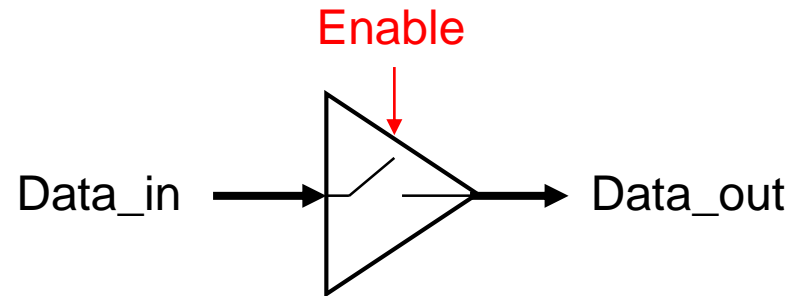
  - ✧ Data signal (data\_in)

  - ✧ Output enable

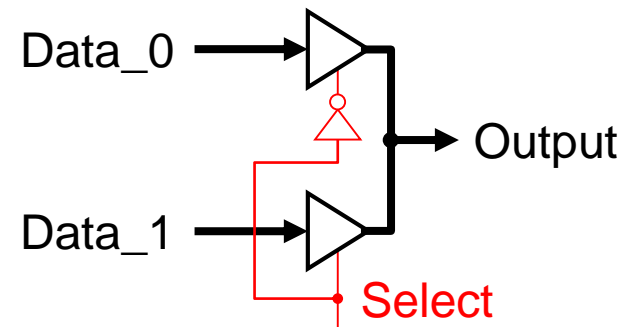
- ❖ One Output (data\_out):

  - ✧ If (Enable) Data\_out = Data\_in

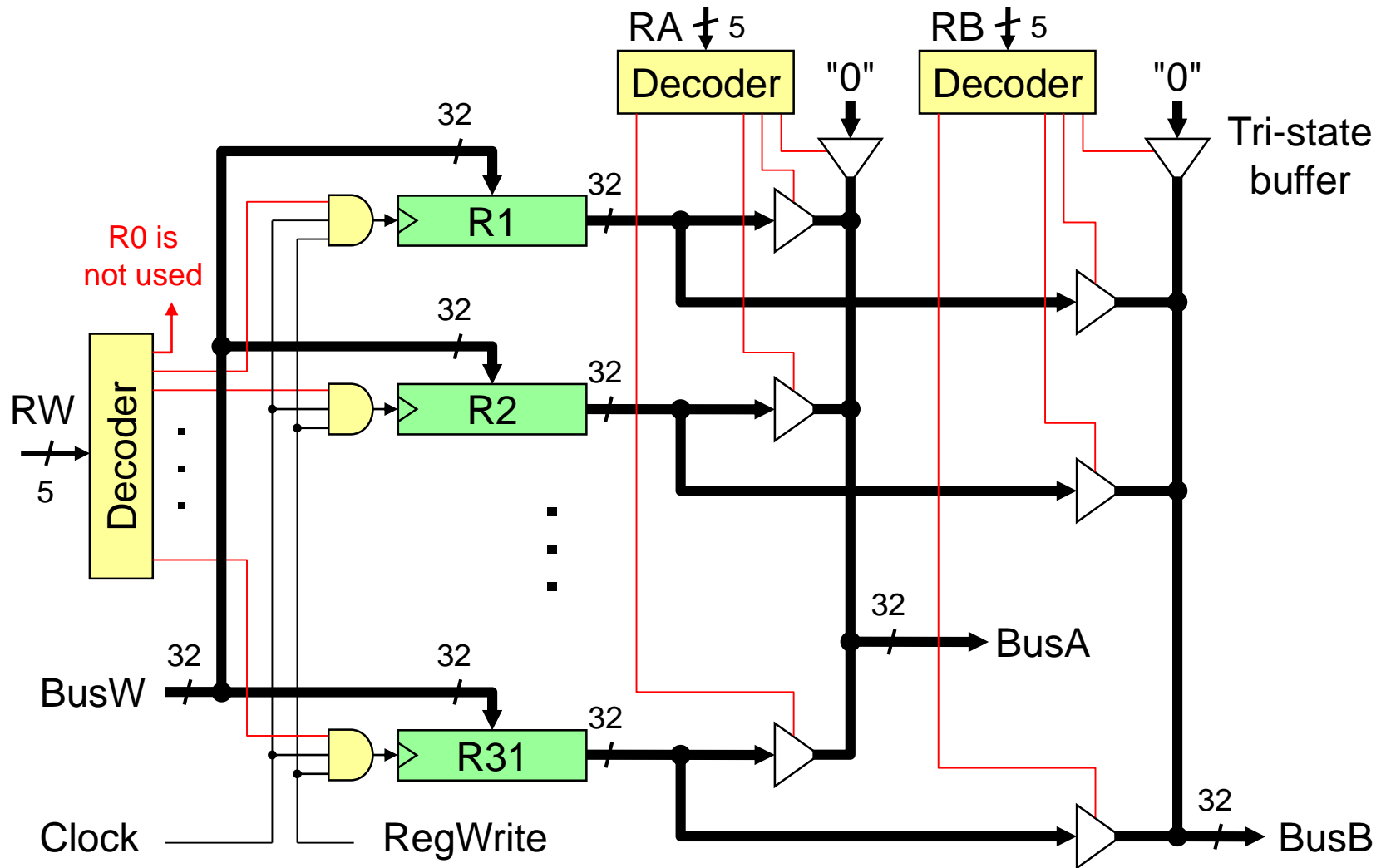
  - else Data\_out = High Impedance state (output is disconnected)



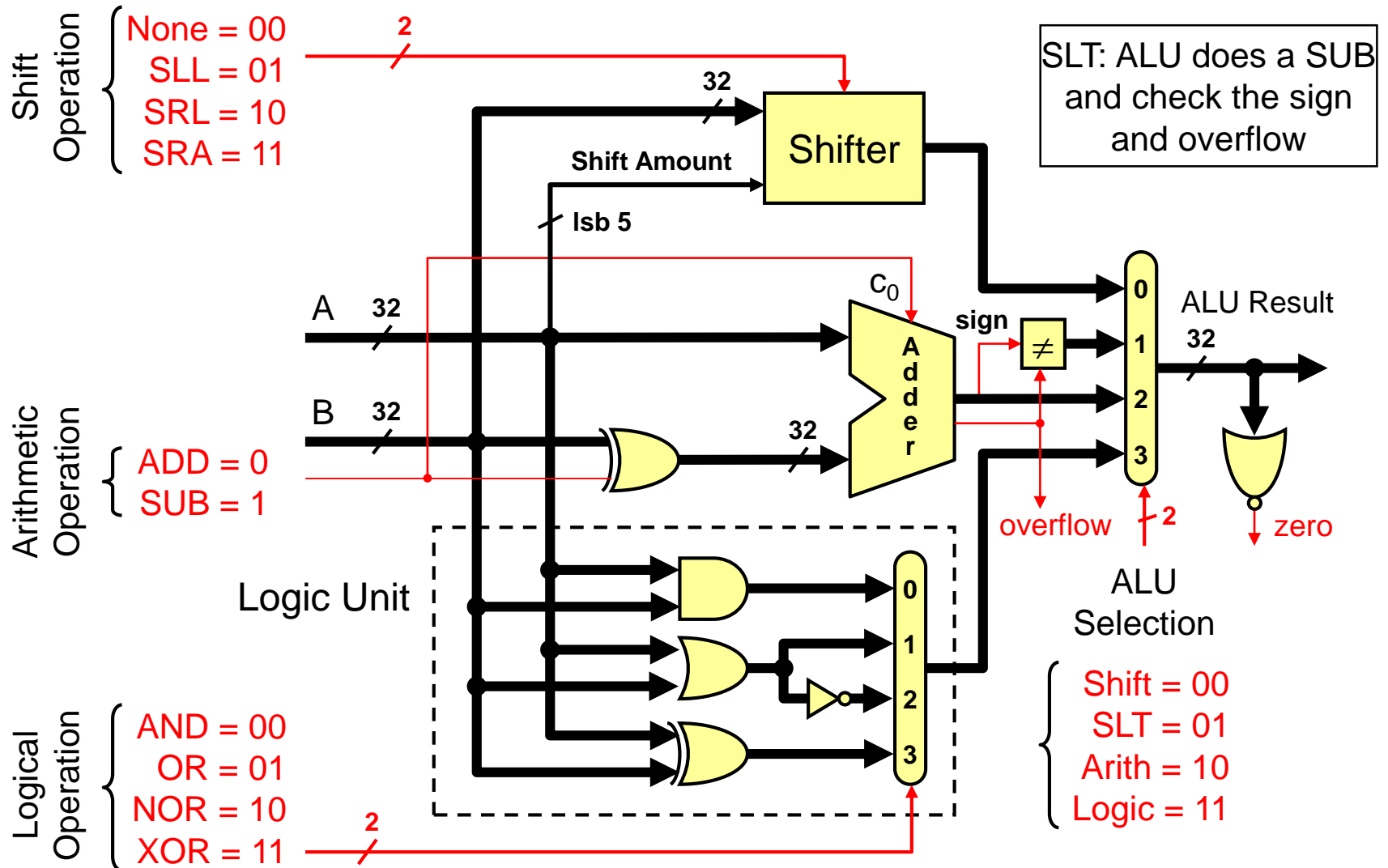
- ❖ Tri-state buffers can be used to build multiplexors



# Details of the Register File



# Building a Multifunction ALU



# Instruction and Data Memories

## ❖ Instruction memory needs only read access

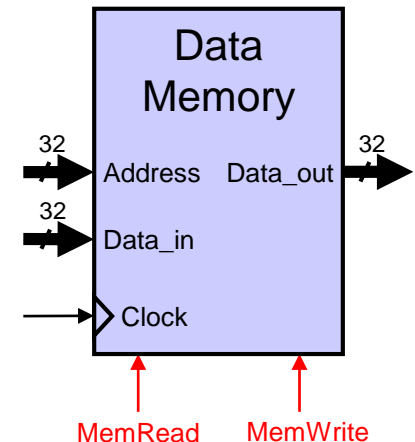
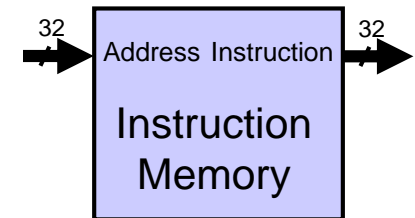
- ✧ Because datapath does not write instructions
- ✧ Behaves as combinational logic for read
- ✧ **Address** selects **Instruction** after **access time**

## ❖ Data Memory is used for load and store

- ✧ **MemRead**: enables output on **Data\_out**
  - **Address** selects the word to put on **Data\_out**
- ✧ **MemWrite**: enables writing of **Data\_in**
  - **Address** selects the memory word to be written
  - The **Clock** synchronizes the write operation

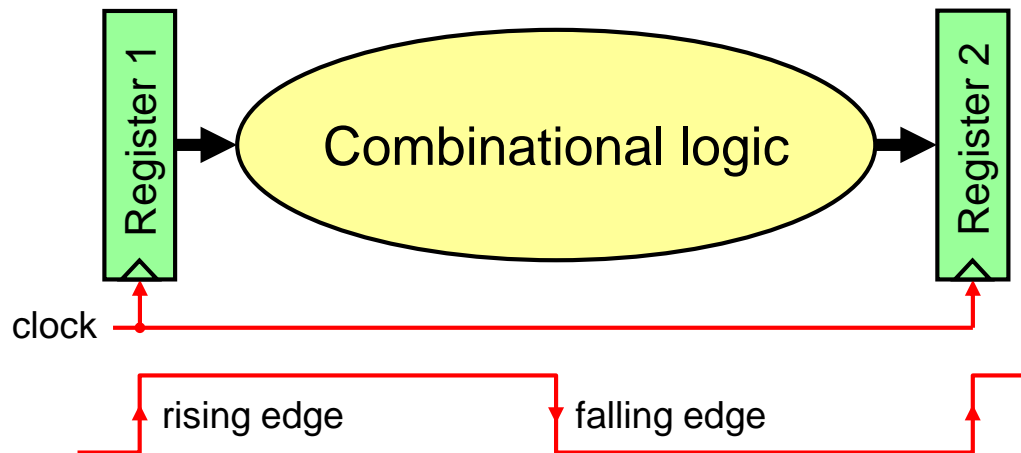
## ❖ Separate instruction and data memories

- ✧ Later, we will replace them with **caches**



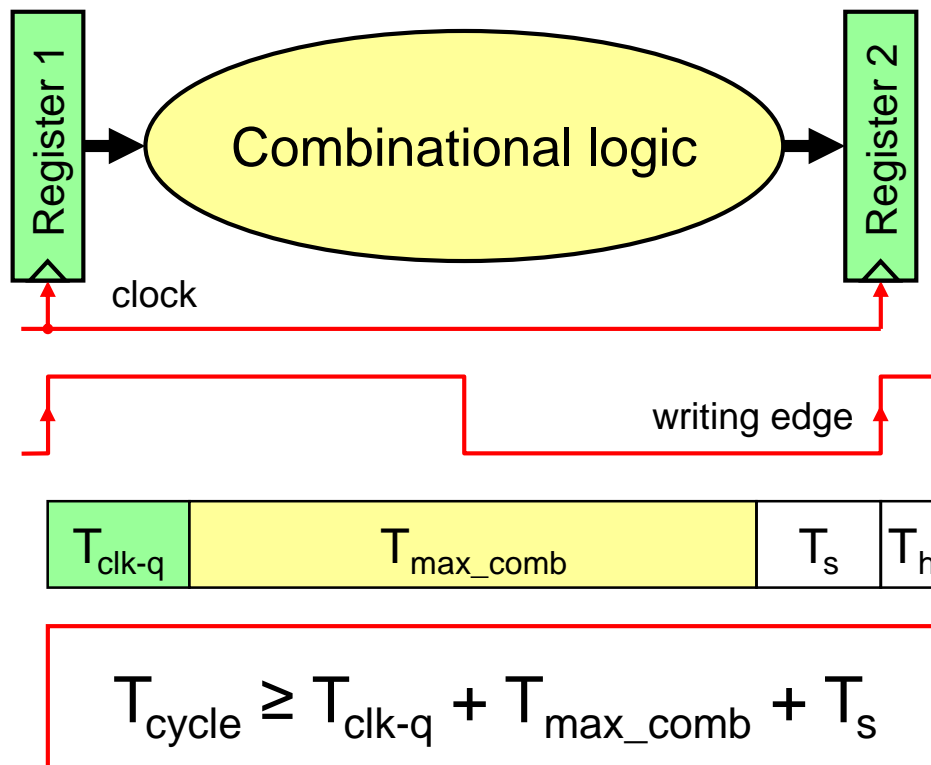
# Clocking Methodology

- ❖ Clocks are needed in a sequential logic to decide when a state element (register) should be updated
- ❖ To ensure correctness, a **clocking methodology** defines when data can be written and read
- ❖ We assume **edge-triggered clocking**
- ❖ All state changes occur on the **same clock edge**
- ❖ Data must be **valid** and **stable** before arrival of clock edge
- ❖ Edge-triggered clocking allows a register to be read and written during same clock cycle



# Determining the Clock Cycle

- ❖ With edge-triggered clocking, the clock cycle must be long enough to accommodate the path from one register through the combinational logic to another register



- ❖  $T_{\text{clk-q}}$ : clock to output delay through register
- ❖  $T_{\text{max\_comb}}$ : longest delay through combinational logic
- ❖  $T_s$ : setup time that input to a register must be stable before arrival of clock edge
- ❖  $T_h$ : hold time that input to a register must hold after arrival of clock edge



# Clock Skew

- ❖ Clock skew arises because the clock signal uses different paths with slightly different delays to reach state elements
- ❖ Clock skew is the difference in absolute time between when two storage elements see a clock edge
- ❖ With clock skew, the clock cycle time will be changed

$$T_{\text{cycle}} \geq T_{\text{clk-q}} + T_{\text{max\_comb}} + T_{\text{setup}} \pm T_{\text{skew}}$$

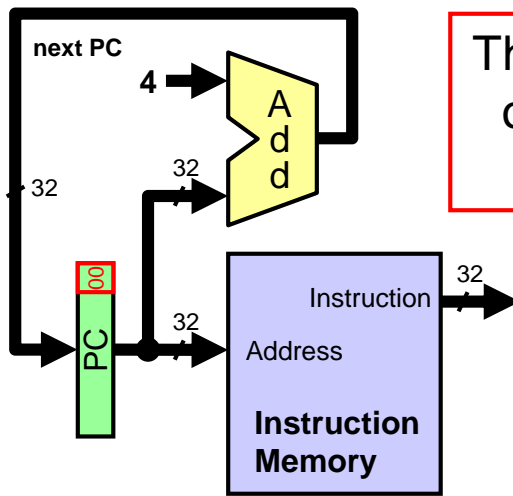
- ❖ Clock skew is reduced by balancing the clock delays

# Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

# Instruction Fetching Datapath

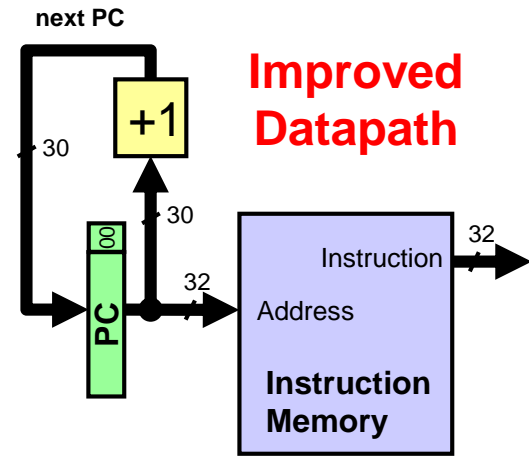
- ❖ Assembling the datapath from its components
- ❖ For instruction fetching, we need ...
  - ✧ Program Counter (PC) register
  - ✧ Instruction Memory
  - ✧ Adder for incrementing PC



The least significant 2 bits of the PC are '00' since PC is a multiple of 4

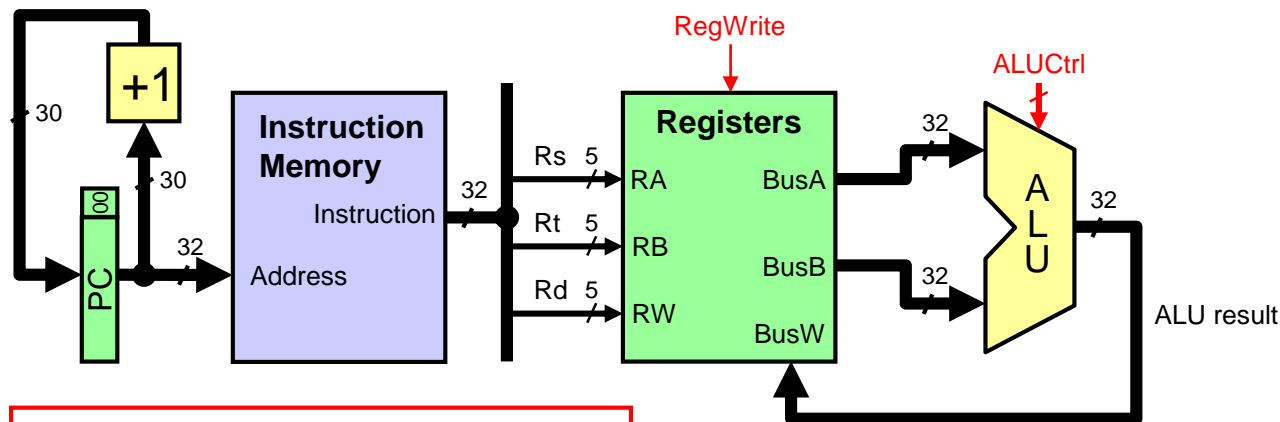
Datapath does not handle branch or jump instructions

Improved datapath increments **upper 30 bits** of PC by 1



**Improved Datapath**

# Datapath for R-type Instructions



RA & RB come from the instruction's Rs & Rt fields

ALU inputs come from BusA & BusB

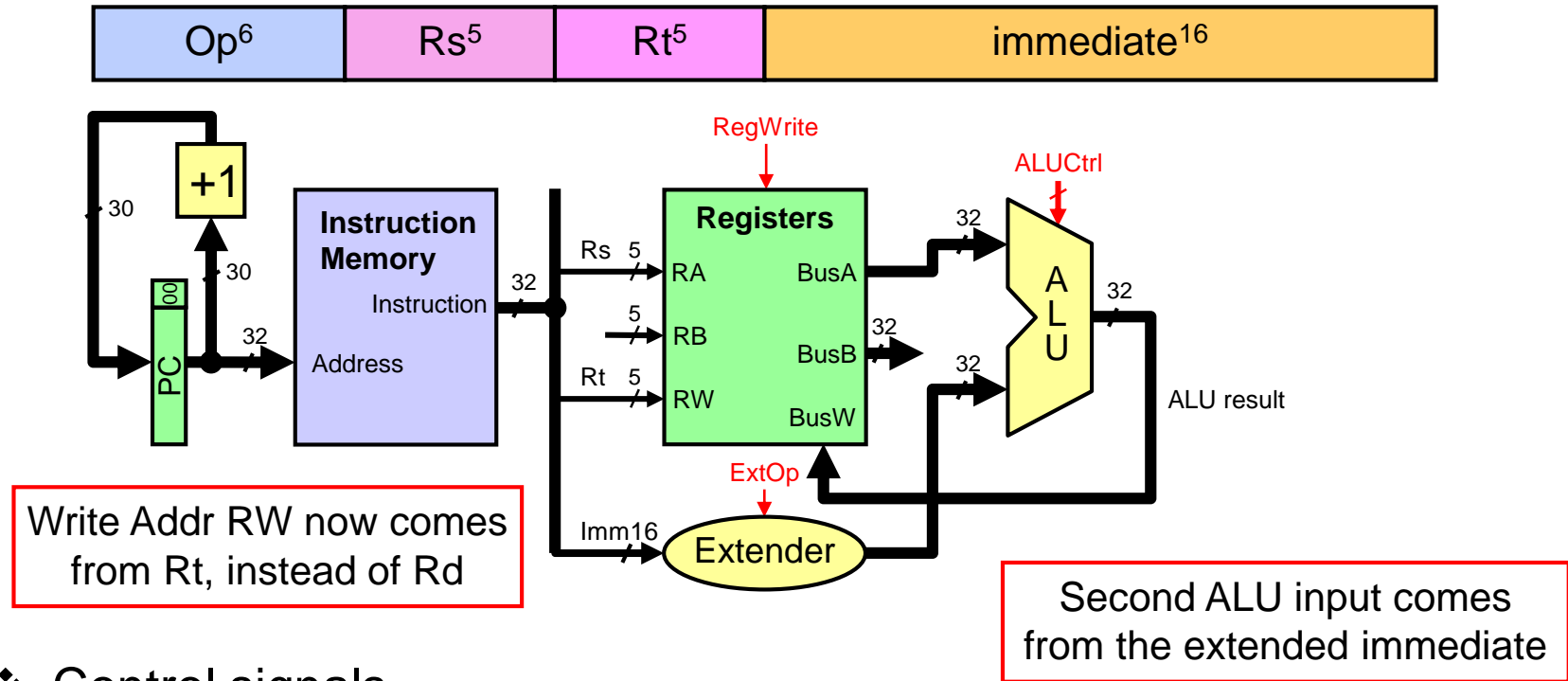
RW comes from the Rd field

ALU result is connected to BusW

## ❖ Control signals

- ❖ **ALU Ctrl** is derived from the **funct** field because Op = 0 for R-type
- ❖ **RegWrite** is used to enable the writing of the ALU result

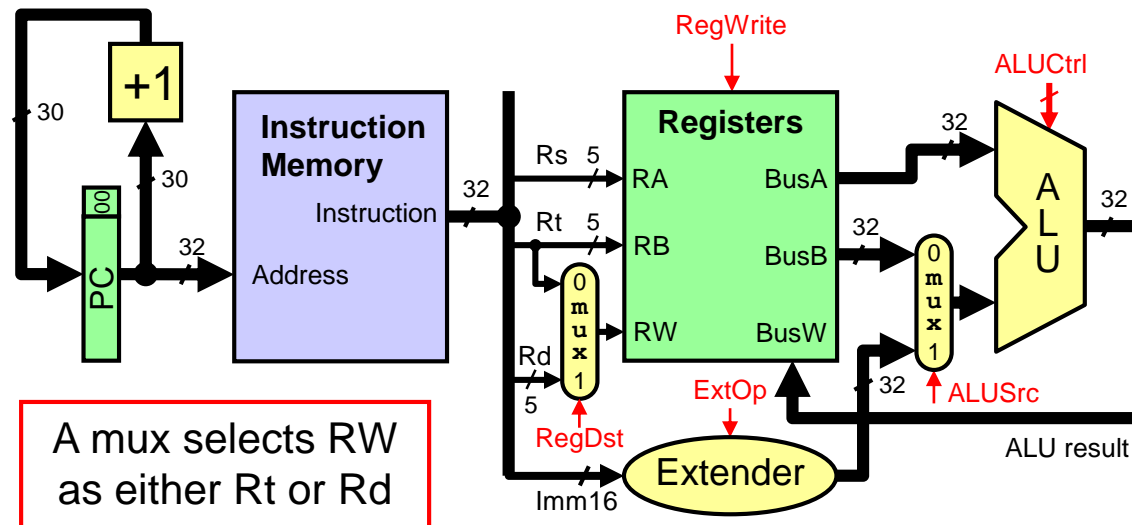
# Datapath for I-type ALU Instructions



## ❖ Control signals

- ❖ **ALU Ctrl** is derived from the **Op** field
- ❖ **RegWrite** is used to enable the writing of the **ALU result**
- ❖ **ExtOp** is used to control the extension of the 16-bit immediate

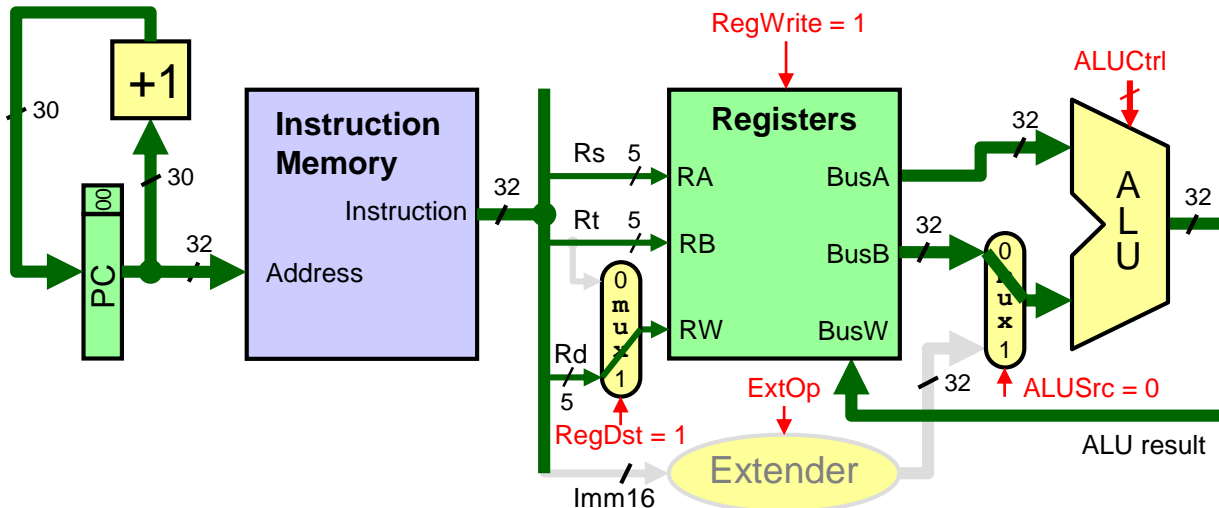
# Combining R-type & I-type Datapaths



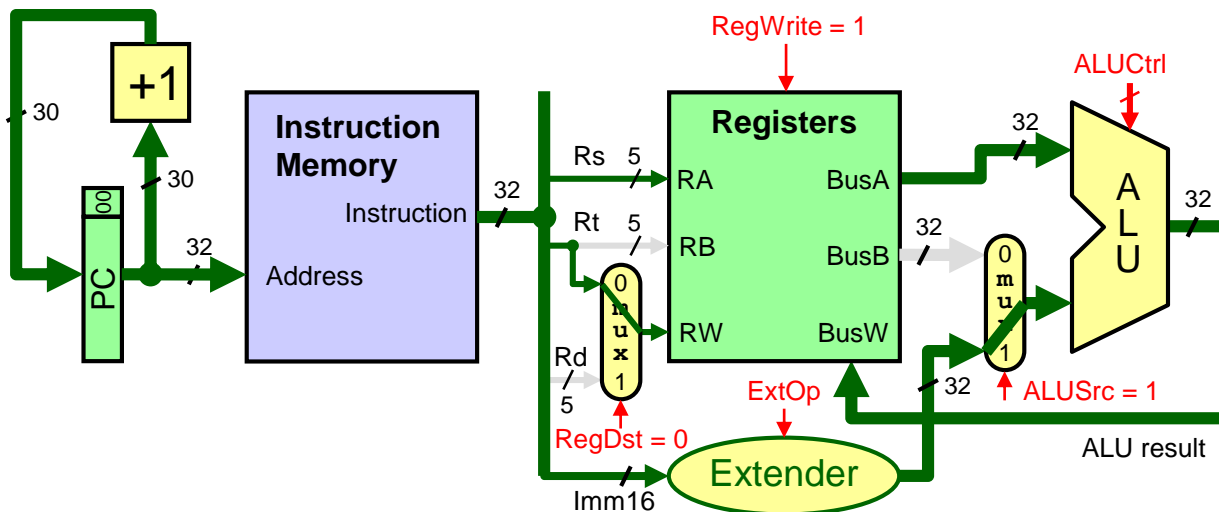
## ❖ Control signals

- ❖ **ALUCtrl** is derived from either the **Op** or the **funct** field
- ❖ **RegWrite** enables the writing of the **ALU result**
- ❖ **ExtOp** controls the extension of the 16-bit immediate
- ❖ **RegDst** selects the register destination as either **Rt** or **Rd**
- ❖ **ALUSrc** selects the 2<sup>nd</sup> ALU source as **BusB** or **extended immediate**

# Controlling ALU Instructions



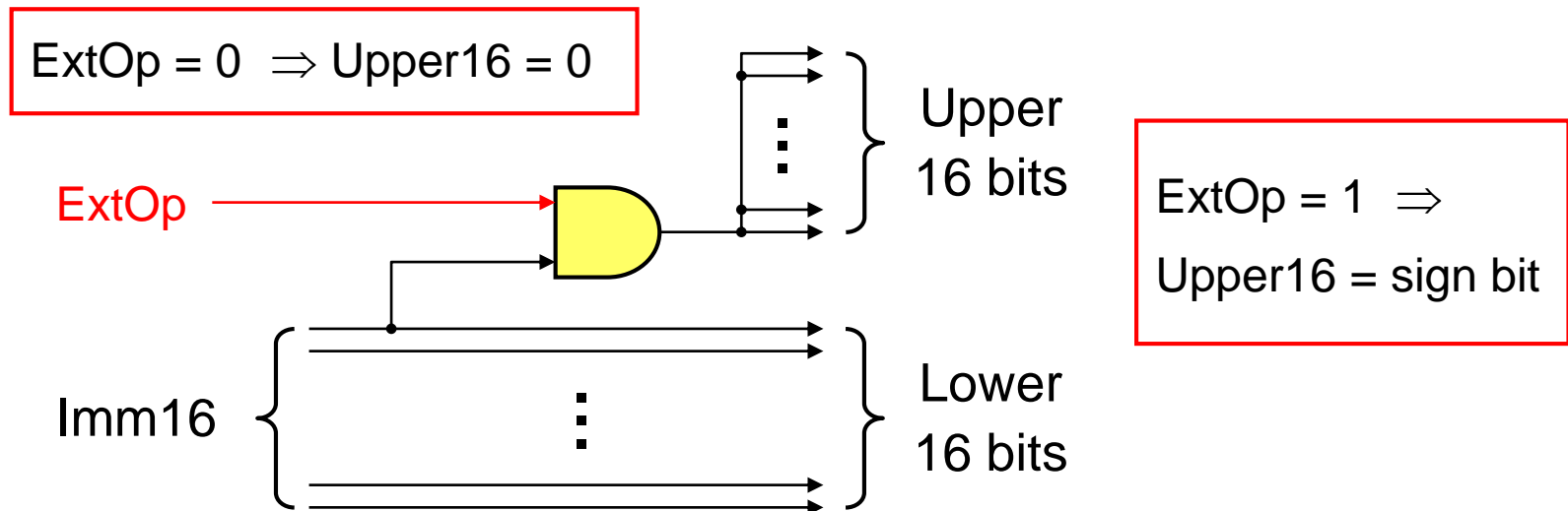
For R-type ALU instructions, **RegDst** is '1' to select Rd on RW and **ALUSrc** is '0' to select BusB as second ALU input. The active part of datapath is shown in **green**



For I-type ALU instructions, **RegDst** is '0' to select Rt on RW and **ALUSrc** is '1' to select Extended immediate as second ALU input. The active part of datapath is shown in **green**

# Details of the Extender

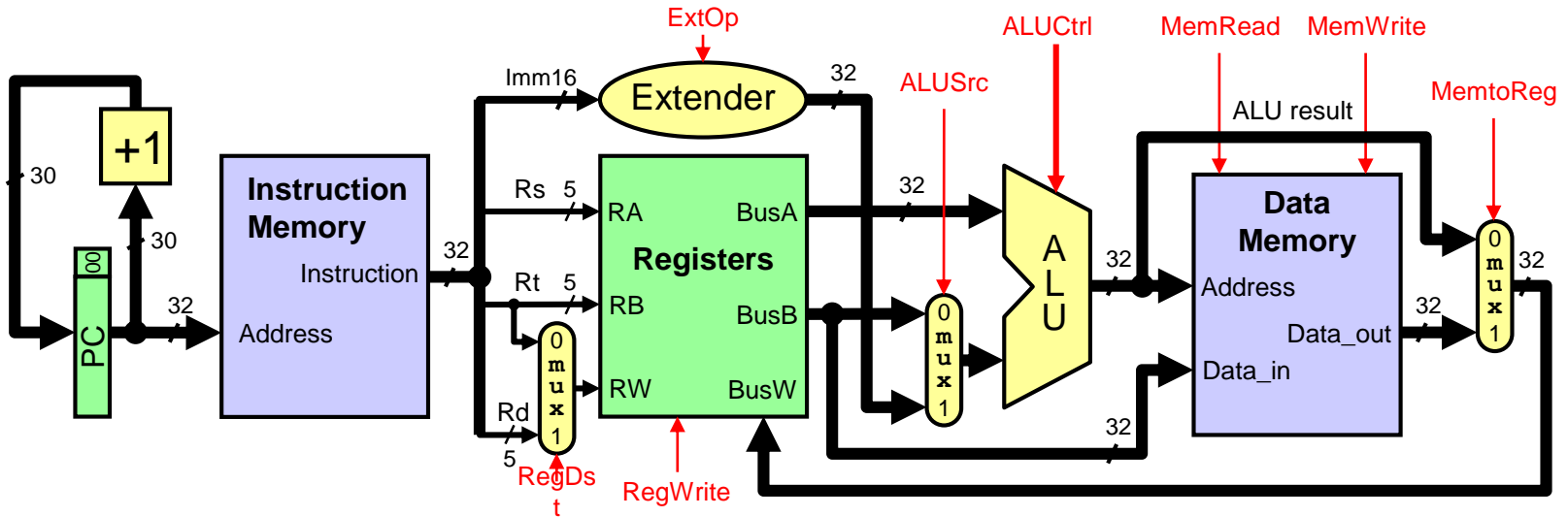
- ❖ Two types of extensions
  - ✧ Zero-extension for unsigned constants
  - ✧ Sign-extension for signed constants
- ❖ Control signal **ExtOp** indicates type of extension
- ❖ Extender Implementation: wiring and one AND gate





# Adding Data Memory to Datapath

- ❖ A **data memory** is added for **load** and **store** instructions



ALU calculates data memory address

A 3<sup>rd</sup> mux selects data on BusW as either ALU result or memory data\_out

- ❖ Additional Control signals

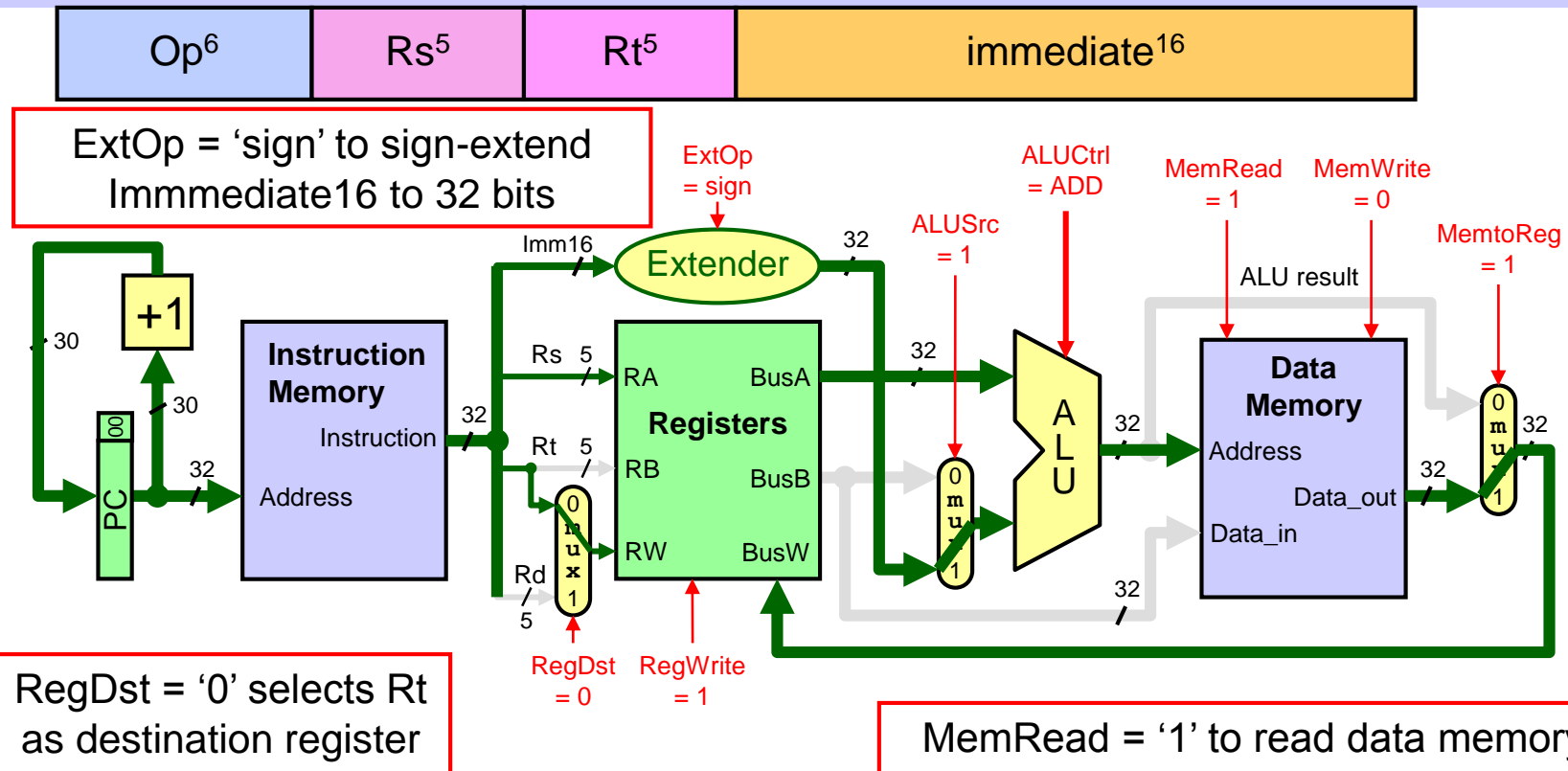
✧ **MemRead** for load instructions

✧ **MemWrite** for store instructions

✧ MemtoReg selects data on BusW as ALU result or Memory Data\_out

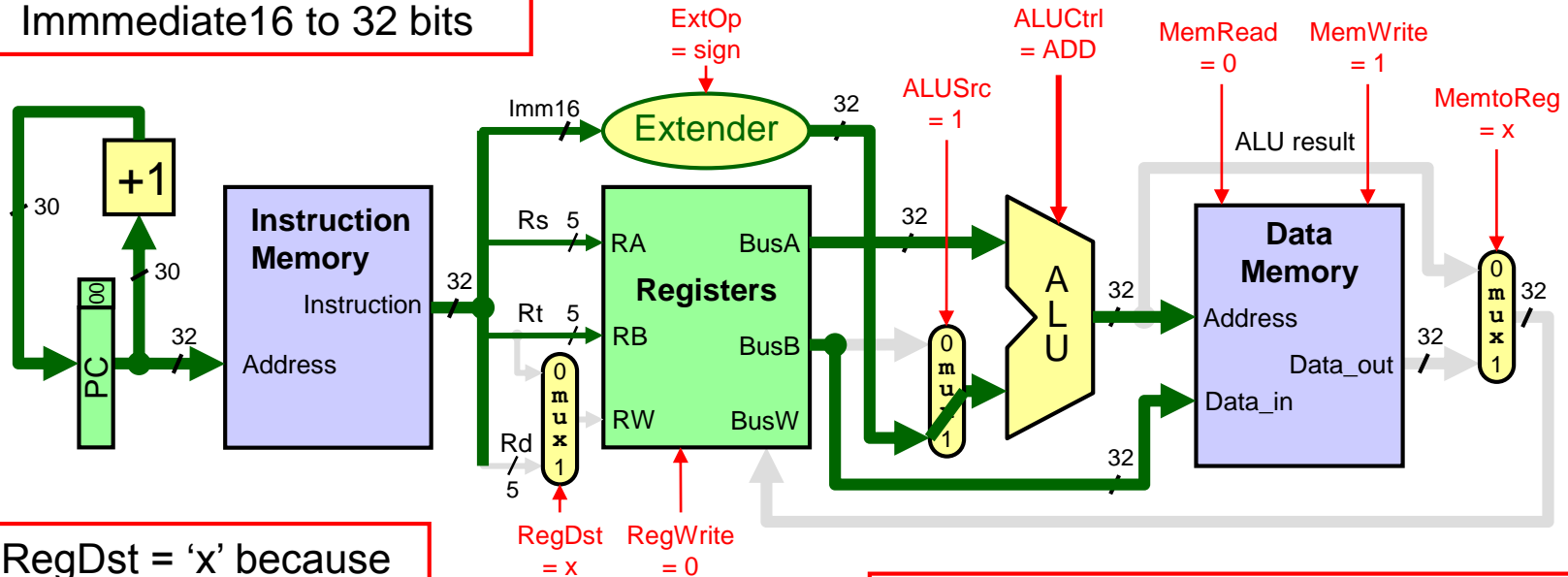
BusB is connected to Data\_in of Data Memory for store instructions

# Controlling the Execution of Load



# Controlling the Execution of Store

ExtOp = 'sign' to sign-extend  
Immediate16 to 32 bits



RegDst = 'x' because  
no destination register

MemWrite = '1' to write data memory

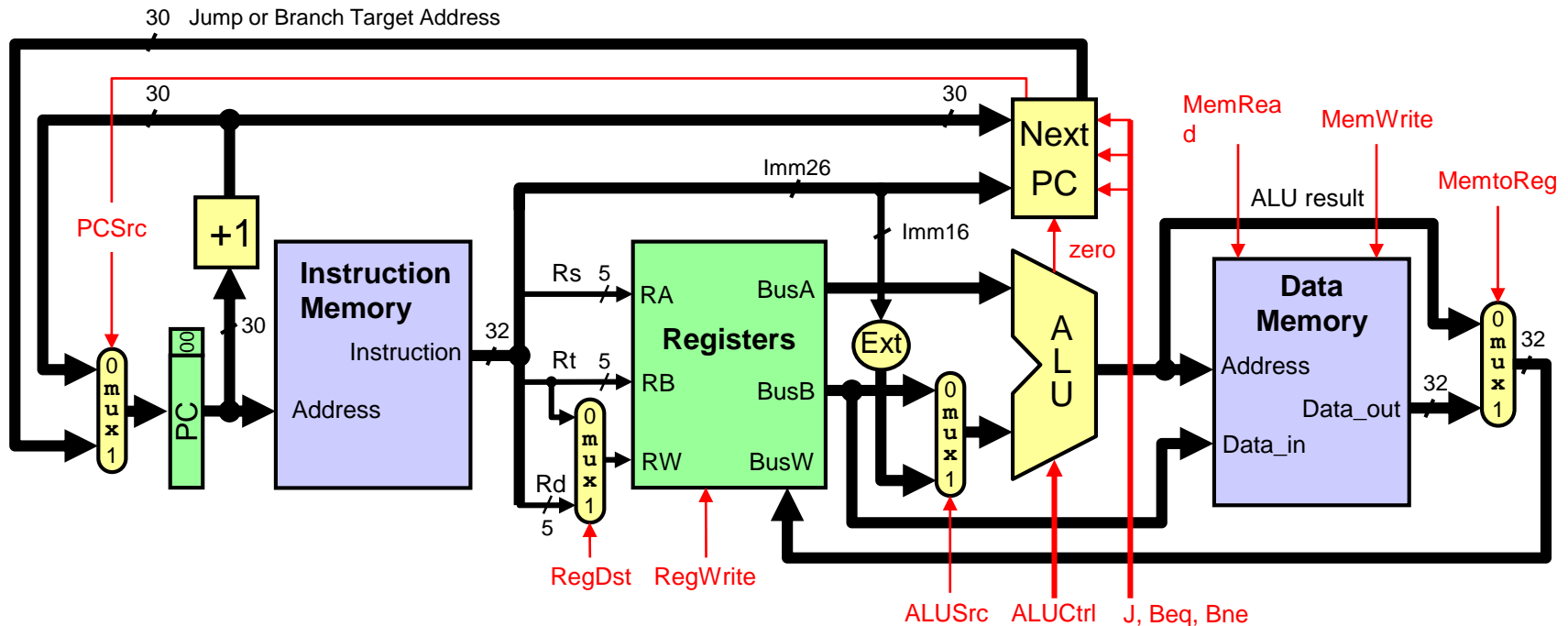
ALUSrc = '1' to select the extended  
immediate as second ALU input

MemtoReg = 'x' because we don't  
care what data is placed on BusW

ALU Ctrl = 'ADD' to calculate data memory  
address as  $\text{Reg}(\text{Rs}) + \text{sign-extend}(\text{Imm16})$

RegWrite = '0' because no register is  
written by the store instruction

# Adding Jump and Branch to Datapath



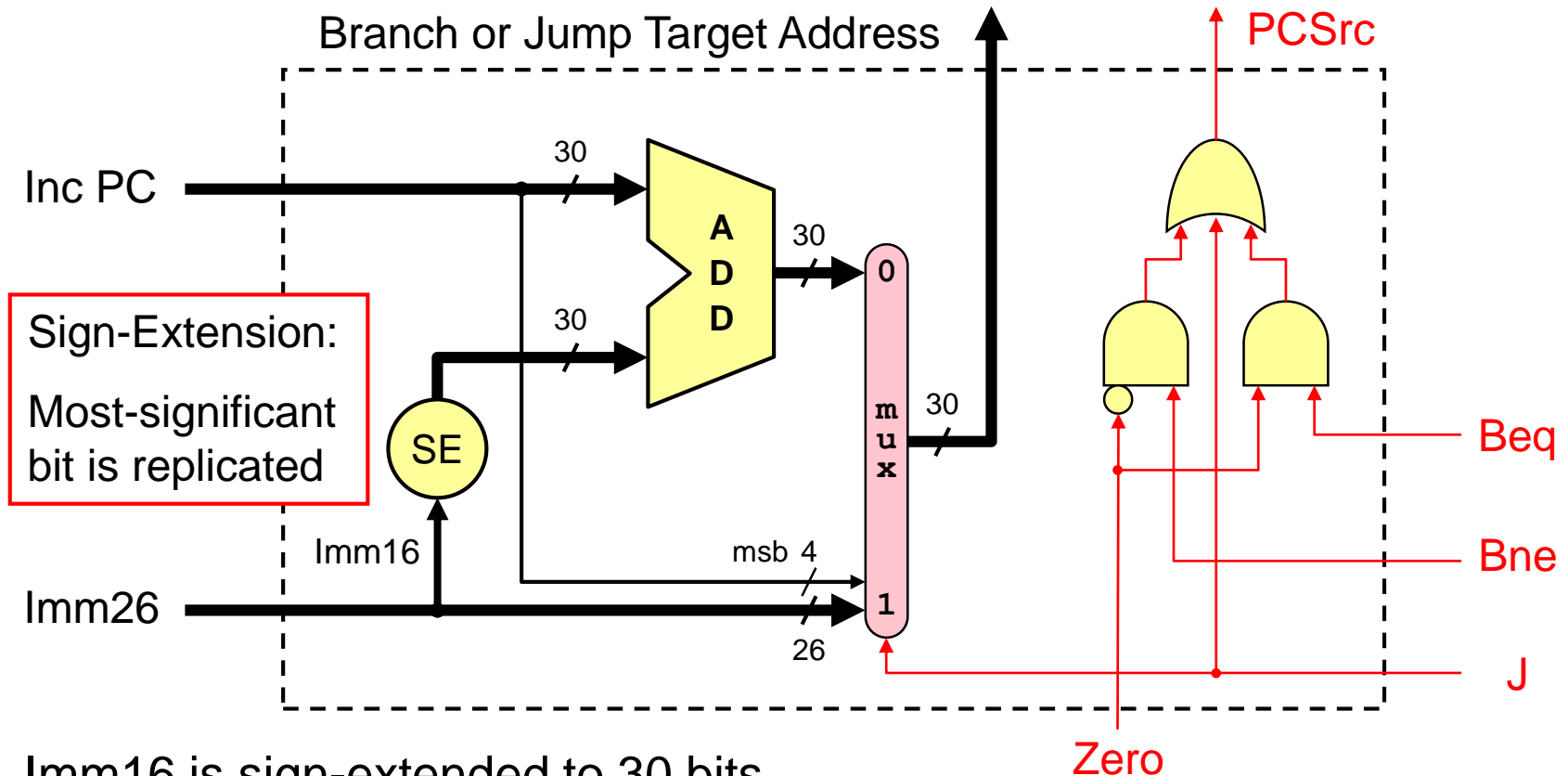
## ❖ Additional Control Signals

- ❖ **J, Beq, Bne** for jump and branch instructions
- ❖ **Zero** condition of the ALU is examined
- ❖ **PCSrc = 1** for Jump & taken Branch

Next PC computes jump or branch target instruction address

For Branch, ALU does a subtraction

# Details of Next PC

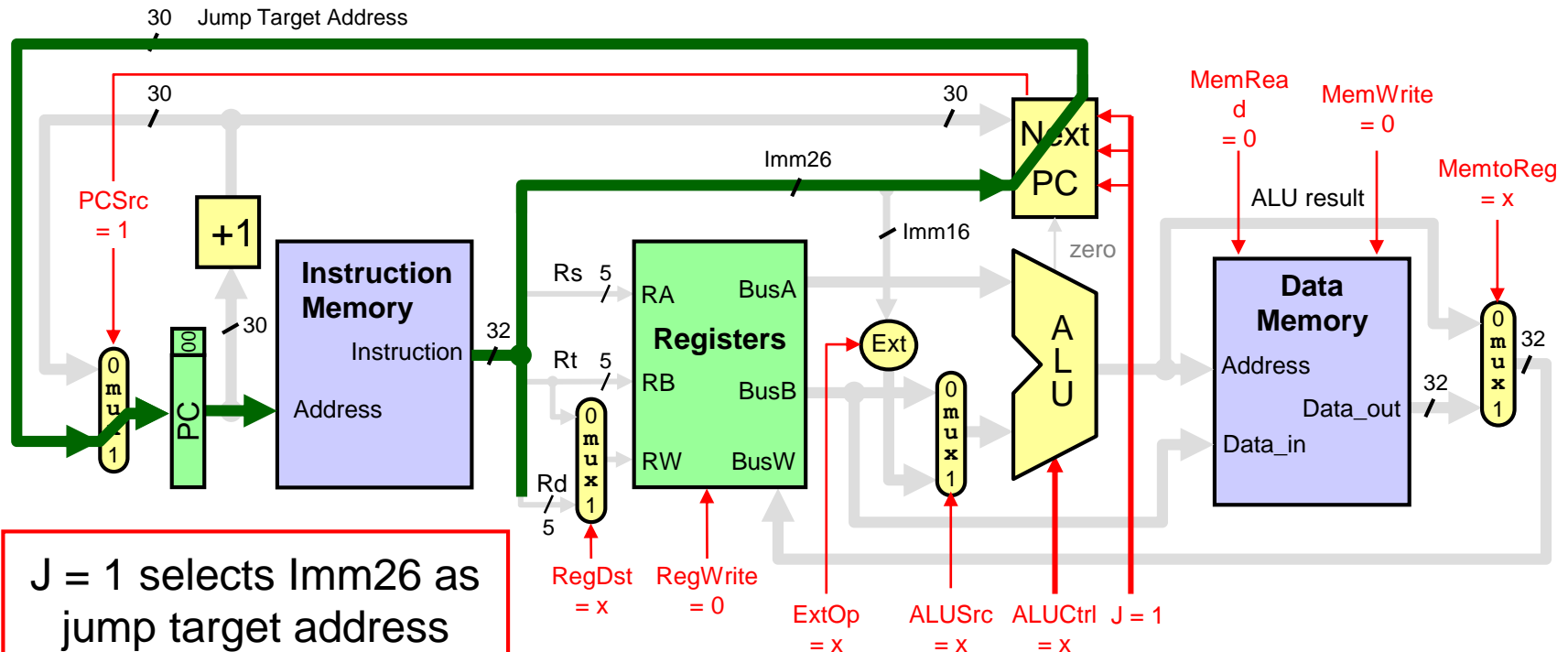


Imm16 is sign-extended to 30 bits

Jump target address: upper 4 bits of PC are concatenated with Imm26

$$PCSrc = J + (Beq \cdot Zero) + (Bne \cdot \overline{Zero})$$

# Controlling the Execution of Jump



J = 1 selects Imm26 as jump target address

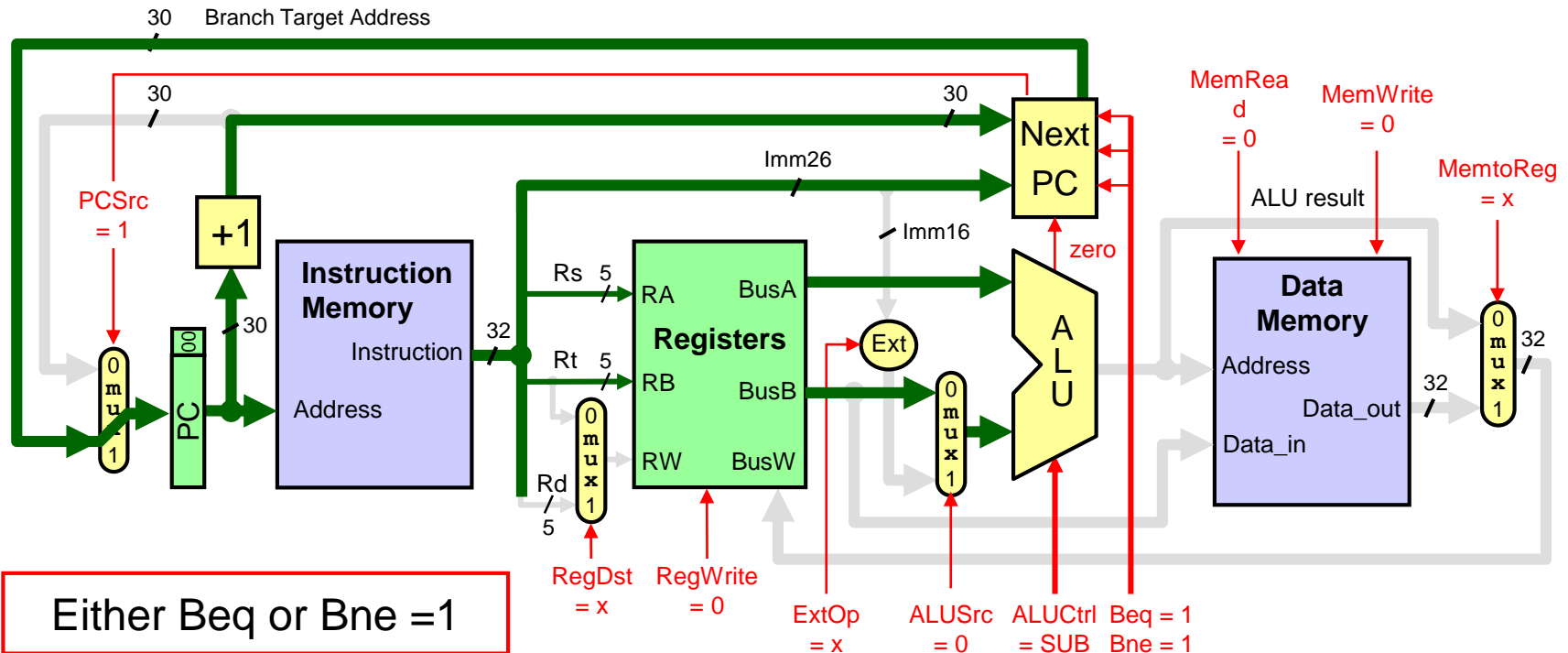
Upper 4 bits are from the incremented PC

$PCSrc = 1$  to select jump target address

MemRead, MemWrite & RegWrite are 0

We don't care about RegDst, ExtOp, ALUSrc, ALUCtrl, and MemtoReg

# Controlling the Execution of Branch



Next PC outputs branch target address

ALUSrc = '0' (2<sup>nd</sup> ALU input is BusB)  
ALUCtrl = 'SUB' produces zero flag

Next PC logic determines PCSrc according to zero flag

MemRead = MemWrite = RegWrite = 0

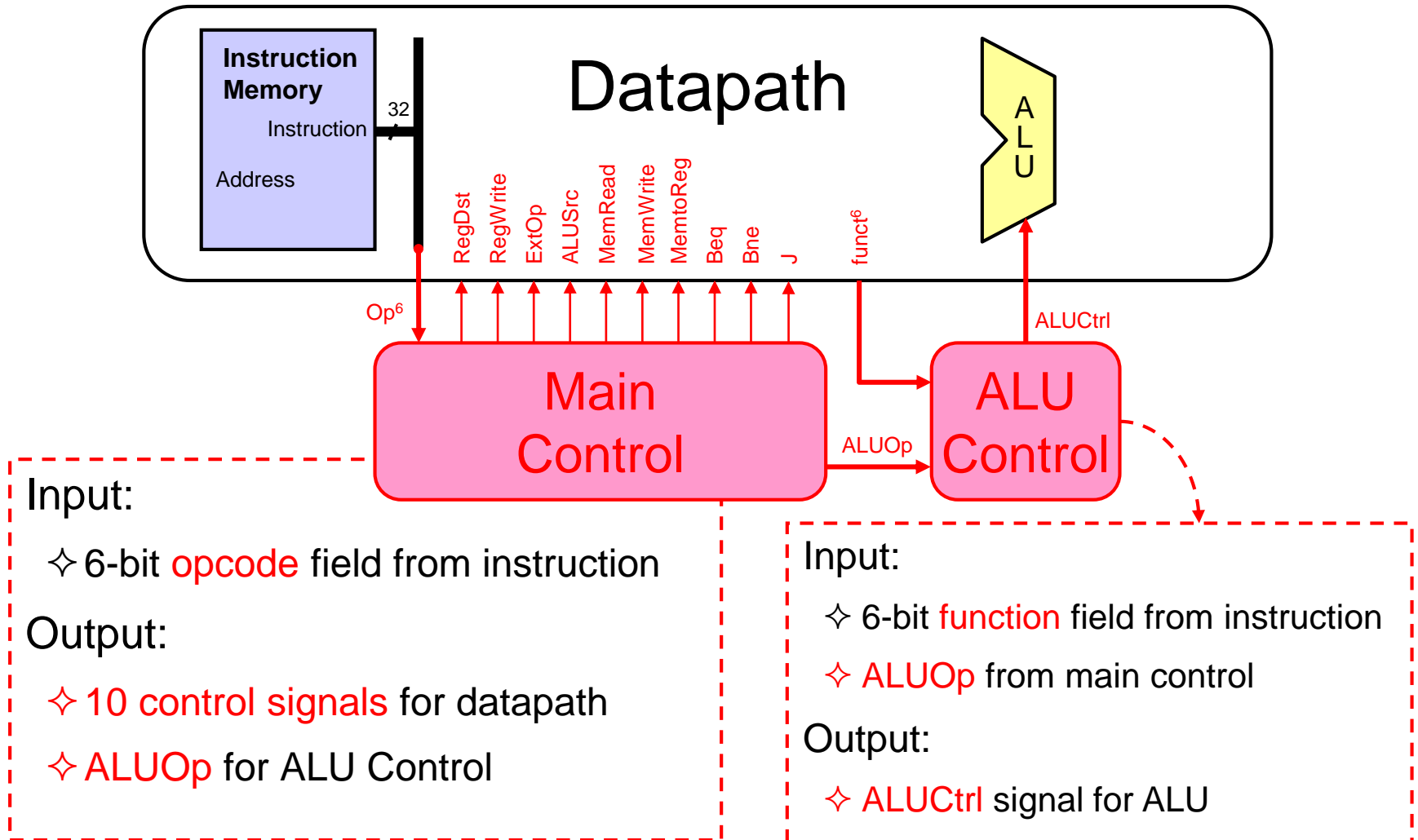
RegDst = ExtOp = MemtoReg = x

# Next . . .

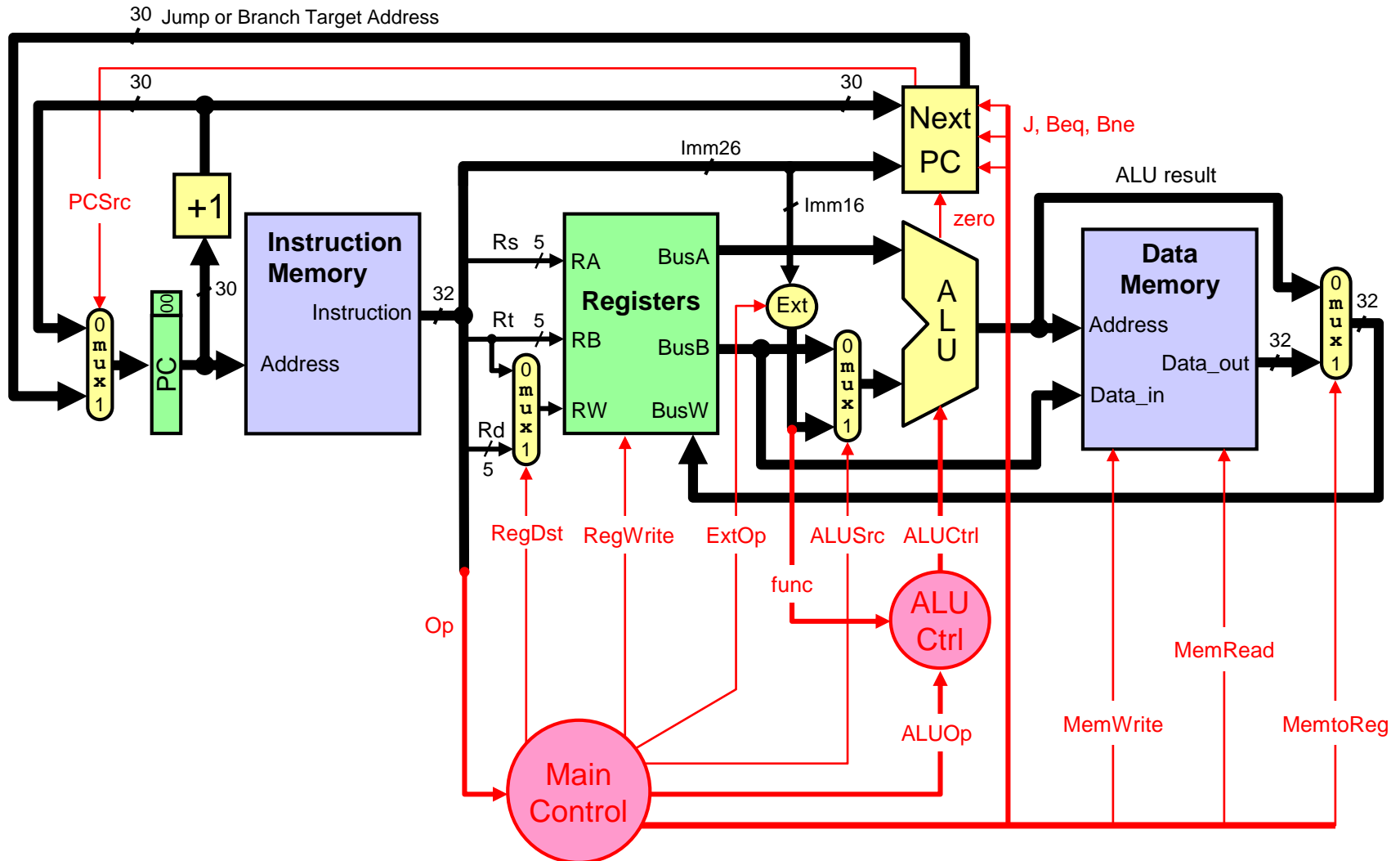
- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design



# Main Control and ALU Control



# Single-Cycle Datapath + Control



# Main Control Signals

| Signal   | Effect when '0'   | Effect when '1'  |
|----------|---|--|
| RegDst   | Destination register = Rt   | Destination register = Rd  |
| RegWrite | None  | Destination register is written with the data value on BusW                          |
| ExtOp    | 16-bit immediate is zero-extended   | 16-bit immediate is sign-extended  |
| ALUSrc   | Second ALU operand comes from the second register file output (BusB)          | Second ALU operand comes from the extended 16-bit immediate                          |
| MemRead  | None  | Data memory is read<br>$\text{Data\_out} \leftarrow \text{Memory}[\text{address}]$   |
| MemWrite | None  | Data memory is written<br>$\text{Memory}[\text{address}] \leftarrow \text{Data\_in}$ |
| MemtoReg | BusW = ALU result   | BusW = Data_out from Memory  |
| Beq, Bne | $\text{PC} \leftarrow \text{PC} + 4$  | $\text{PC} \leftarrow \text{Branch target address}$<br>If branch is taken            |
| J        | $\text{PC} \leftarrow \text{PC} + 4$  | $\text{PC} \leftarrow \text{Jump target address}$                                    |
| ALUOp    | This multi-bit signal specifies the ALU operation as a function of the opcode |  |

# Main Control Signal Values

| Op     | Reg Dst | Reg Write | Ext Op | ALU Src | ALU Op | Beq | Bne | J | Mem Read | Mem Write | Mem toReg |
|--------|---------|-----------|--------|---------|--------|-----|-----|---|----------|-----------|-----------|
| R-type | 1 = Rd  | 1         | x      | 0=BusB  | R-type | 0   | 0   | 0 | 0        | 0         | 0         |
| addi   | 0 = Rt  | 1         | 1=sign | 1=Imm   | ADD    | 0   | 0   | 0 | 0        | 0         | 0         |
| slti   | 0 = Rt  | 1         | 1=sign | 1=Imm   | SLT    | 0   | 0   | 0 | 0        | 0         | 0         |
| andi   | 0 = Rt  | 1         | 0=zero | 1=Imm   | AND    | 0   | 0   | 0 | 0        | 0         | 0         |
| ori    | 0 = Rt  | 1         | 0=zero | 1=Imm   | OR     | 0   | 0   | 0 | 0        | 0         | 0         |
| xori   | 0 = Rt  | 1         | 0=zero | 1=Imm   | XOR    | 0   | 0   | 0 | 0        | 0         | 0         |
| lw     | 0 = Rt  | 1         | 1=sign | 1=Imm   | ADD    | 0   | 0   | 0 | 1        | 0         | 1         |
| sw     | x       | 0         | 1=sign | 1=Imm   | ADD    | 0   | 0   | 0 | 0        | 1         | x         |
| beq    | x       | 0         | x      | 0=BusB  | SUB    | 1   | 0   | 0 | 0        | 0         | x         |
| bne    | x       | 0         | x      | 0=BusB  | SUB    | 0   | 1   | 0 | 0        | 0         | x         |
| j      | x       | 0         | x      | x       | x      | 0   | 0   | 1 | 0        | 0         | x         |

❖ X is a don't care (can be 0 or 1), used to minimize logic

# Logic Equations for Control Signals

RegDst  $\leq$  R-type

RegWrite  $\leq$  (sw + beq + bne + j)

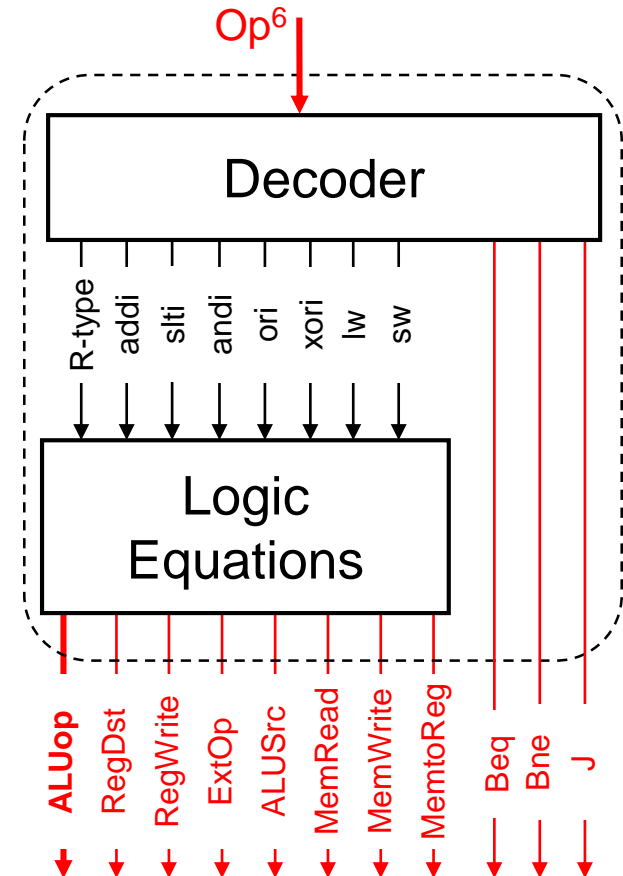
ExtOp  $\leq$  (andi + ori + xori)

ALUSrc  $\leq$  (R-type + beq + bne)

MemRead  $\leq$  lw

MemWrite  $\leq$  sw

MemtoReg  $\leq$  lw



# ALU Control Truth Table

| Op <sup>6</sup> | ALU Control |                    |         | 4-bit Encoding |
|-----------------|-------------|--------------------|---------|----------------|
|                 | ALUOp       | funct <sup>6</sup> | ALUCtrl |                |
| R-type          | R-type      | add                | ADD     | 0000           |
| R-type          | R-type      | sub                | SUB     | 0010           |
| R-type          | R-type      | and                | AND     | 0100           |
| R-type          | R-type      | or                 | OR      | 0101           |
| R-type          | R-type      | xor                | XOR     | 0110           |
| R-type          | R-type      | slt                | SLT     | 1010           |
| addi            | ADD         | x                  | ADD     | 0000           |
| slti            | SLT         | x                  | SLT     | 1010           |
| andi            | AND         | x                  | AND     | 0100           |
| ori             | OR          | x                  | OR      | 0101           |
| xori            | XOR         | x                  | XOR     | 0110           |
| lw              | ADD         | x                  | ADD     | 0000           |
| sw              | ADD         | x                  | ADD     | 0000           |
| beq             | SUB         | x                  | SUB     | 0010           |
| bne             | SUB         | x                  | SUB     | 0010           |
| j               | x           | x                  | x       | x              |

The 4-bit encoding for ALUctrl is chosen here to be equal to the last 4 bits of the function field

Other binary encodings are also possible. The idea is to choose a binary encoding that will minimize the logic for ALU Control

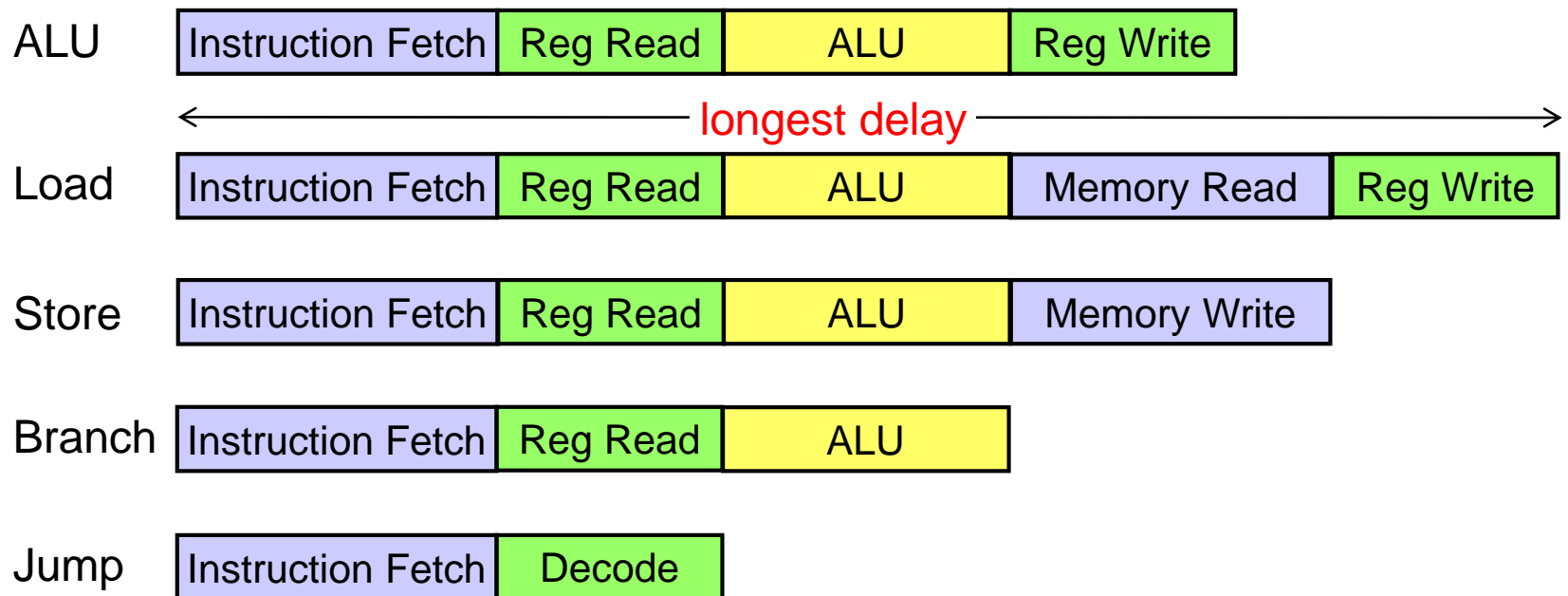
# Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ The Main Controller and ALU Controller
- ❖ Drawback of the single-cycle processor design

# Drawbacks of Single Cycle Processor

## ❖ Long cycle time

✧ All instructions take as much time as the **slowest**



## ❖ Alternative Solution: **Multicycle** implementation

✧ Break down instruction execution into multiple cycles



# Multicycle Implementation

## ❖ Break instruction execution into **five steps**

- ✧ Instruction fetch
- ✧ Instruction decode and register read
- ✧ Execution, memory address calculation, or branch completion
- ✧ Memory access or ALU instruction completion
- ✧ Load instruction completion

## ❖ **One step = One clock cycle** (clock cycle is reduced)

- ✧ First 2 steps are the same for all instructions

| Instruction | # cycles | Instruction | # cycles |
|-------------|----------|-------------|----------|
| ALU & Store | 4        | Branch      | 3        |
| Load        | 5        | Jump        | 2        |

# Performance Example

- ❖ Assume the following instruction mix:
  - ✧ 40% ALU, 20% Loads, 10% stores, 20% branches, & 10% jumps
- ❖ Assume the following operation times for components:
  - ✧ Instruction and data memories: 200 ps
  - ✧ ALU and adders: 180 ps
  - ✧ Decode and Register file access (read or write): 150 ps
  - ✧ Ignore the delays in PC, mux, extender, and wires
- ❖ Which of the following would be faster and by how much?
  - ✧ Single-cycle implementation for all instructions
  - ✧ Multicycle implementation optimized for every class of instructions

# Solution

| Instruction Class | Instruction Memory | Register Read | ALU Operation        | Data Memory | Register Write | Total  |
|-------------------|--------------------|---------------|----------------------|-------------|----------------|--------|
| ALU               | 200                | 150           | 180                  |             | 150            | 680 ps |
| Load              | 200                | 150           | 180                  | 200         | 150            | 880 ps |
| Store             | 200                | 150           | 180                  | 200         |                | 730 ps |
| Branch            | 200                | 150           | 180                  |             |                | 530 ps |
| Jump              | 200                | 150 ←         | decode and update PC |             |                | 350 ps |

❖ For fixed single-cycle implementation:

✧ Clock cycle = 880 ps determined by longest delay (load instruction)

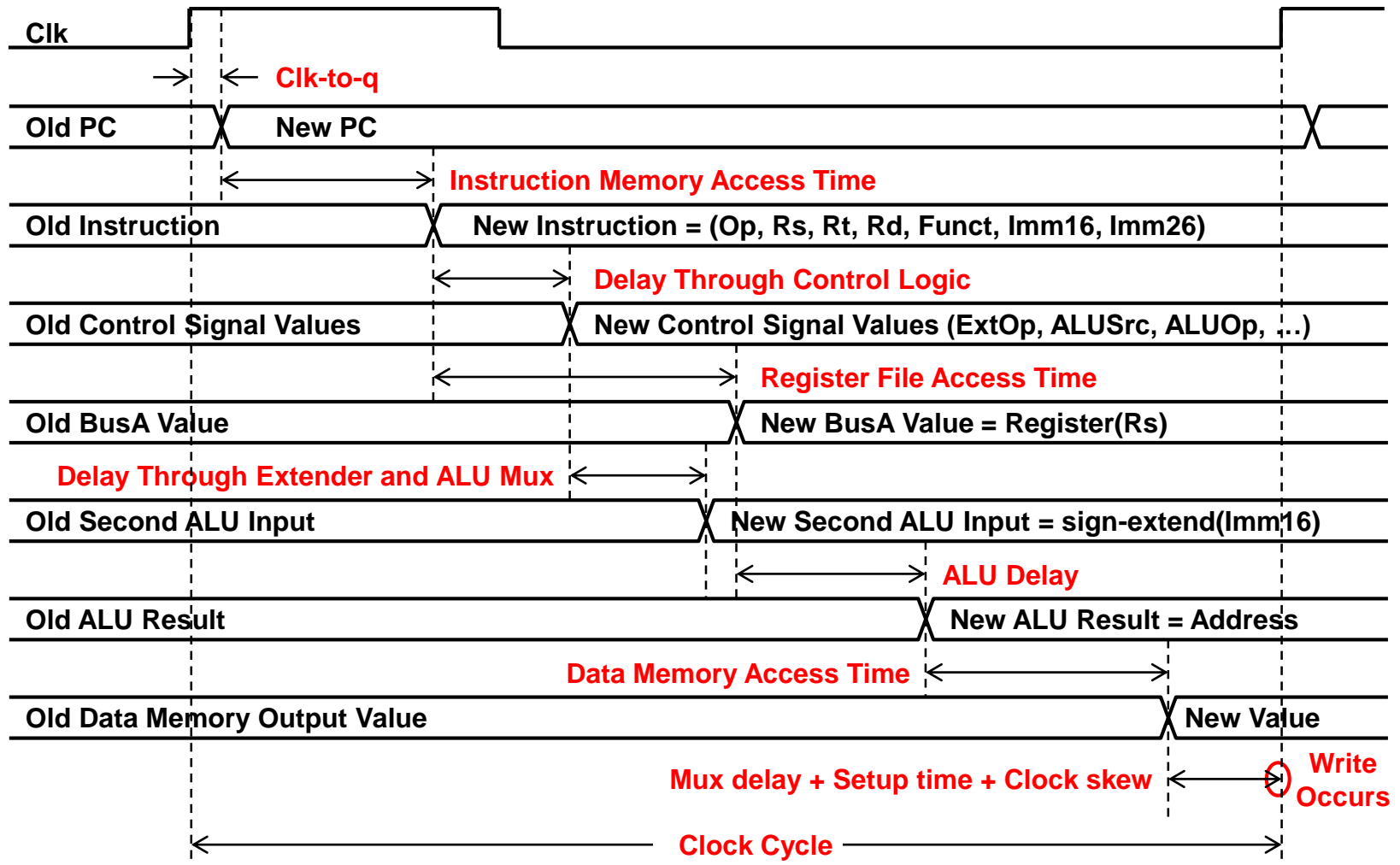
❖ For multi-cycle implementation:

✧ Clock cycle =  $\max(200, 150, 180) = 200$  ps (maximum delay at any step)

✧ Average CPI =  $0.4 \times 4 + 0.2 \times 5 + 0.1 \times 4 + 0.2 \times 3 + 0.1 \times 2 = 3.8$

❖ Speedup =  $880 \text{ ps} / (3.8 \times 200 \text{ ps}) = 880 / 760 = 1.16$

# Worst Case Timing (Load Instruction)



# Worst Case Timing - Cont'd

- ❖ Long cycle time: must be long enough for **Load** operation
  - PC's Clk-to-Q
  - + Instruction Memory's Access Time
  - + Maximum of (
    - Register File's Access Time,
    - Delay through control logic + extender + ALU mux)
  - + ALU to Perform a 32-bit Add
  - + Data Memory Access Time
  - + Delay through MemtoReg Mux
  - + Setup Time for Register File Write + Clock Skew
- ❖ Cycle time is **longer than needed** for other instructions
  - ✧ Therefore, single cycle processor design is not used in practice

# Summary

## ❖ 5 steps to design a processor

- ✧ Analyze instruction set => datapath requirements
- ✧ Select datapath components & establish clocking methodology
- ✧ Assemble datapath meeting the requirements
- ✧ Analyze implementation of each instruction to determine control signals
- ✧ Assemble the control logic

## ❖ MIPS makes Control easier

- ✧ Instructions are of same size
- ✧ Source registers always in same place
- ✧ Immediates are of same size and same location
- ✧ Operations are always on registers/immediates

## ❖ Single cycle datapath => CPI=1, but Long Clock Cycle