

# Single Cycle Processor Design

COE 301

Computer Organization

Dr. Muhamed Mudawar

College of Computer Sciences and Engineering  
King Fahd University of Petroleum and Minerals

# Presentation Outline

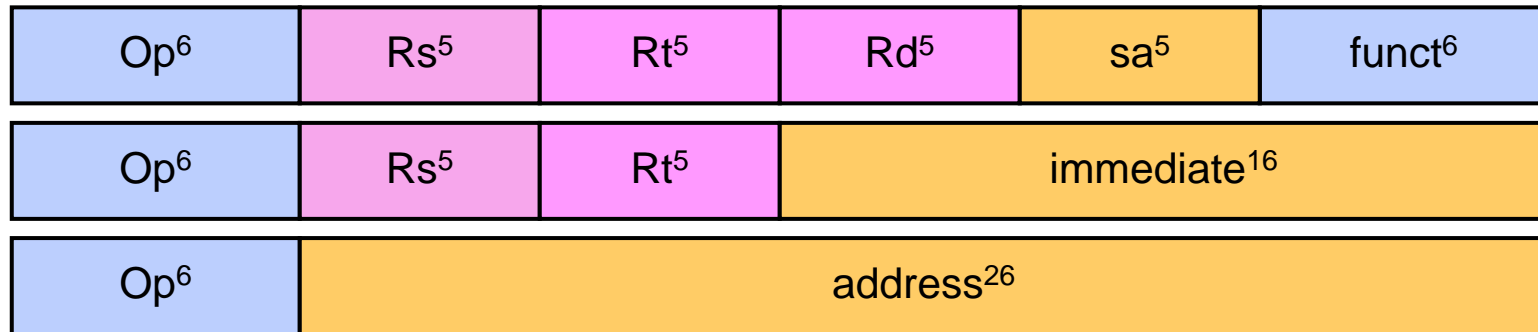
- ❖ **Designing a Processor: Step-by-Step**
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ Main, ALU, and PC Control

# Designing a Processor: Step-by-Step

1. 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
2. Select **datapath components** and **clocking methodology**
3. Assemble **datapath** meeting the requirements
4. Analyze implementation of **each instruction**
  - ✧ Determine the setting of **control signals** for register transfer
5. Assemble the **control logic**

# 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 constant or PC-relative offset
- ✧ address<sup>26</sup>: 26-bit target address of the jump instruction

# MIPS Subset of Instructions

- ❖ Only a subset of the MIPS instructions is 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 <sup>6</sup> = 0	rs <sup>5</sup>	rt <sup>5</sup>	rd <sup>5</sup>	0	0x20
sub	rd, rs, rt	subtraction	op <sup>6</sup> = 0	rs <sup>5</sup>	rt <sup>5</sup>	rd <sup>5</sup>	0	0x22
and	rd, rs, rt	bitwise and	op <sup>6</sup> = 0	rs <sup>5</sup>	rt <sup>5</sup>	rd <sup>5</sup>	0	0x24
or	rd, rs, rt	bitwise or	op <sup>6</sup> = 0	rs <sup>5</sup>	rt <sup>5</sup>	rd <sup>5</sup>	0	0x25
xor	rd, rs, rt	exclusive or	op <sup>6</sup> = 0	rs <sup>5</sup>	rt <sup>5</sup>	rd <sup>5</sup>	0	0x26
slt	rd, rs, rt	set on less than	op <sup>6</sup> = 0	rs <sup>5</sup>	rt <sup>5</sup>	rd <sup>5</sup>	0	0x2a
addi	rt, rs, imm <sup>16</sup>	add immediate	0x08	rs <sup>5</sup>	rt <sup>5</sup>	imm <sup>16</sup>		
slti	rt, rs, imm <sup>16</sup>	slt immediate	0x0a	rs <sup>5</sup>	rt <sup>5</sup>	imm <sup>16</sup>		
andi	rt, rs, imm <sup>16</sup>	and immediate	0x0c	rs <sup>5</sup>	rt <sup>5</sup>	imm <sup>16</sup>		
ori	rt, rs, imm <sup>16</sup>	or immediate	0x0d	rs <sup>5</sup>	rt <sup>5</sup>	imm <sup>16</sup>		
xori	rt, imm <sup>16</sup>	xor immediate	0x0e	rs <sup>5</sup>	rt <sup>5</sup>	imm <sup>16</sup>		
lw	rt, imm <sup>16</sup> (rs)	load word	0x23	rs <sup>5</sup>	rt <sup>5</sup>	imm <sup>16</sup>		
sw	rt, imm <sup>16</sup> (rs)	store word	0x2b	rs <sup>5</sup>	rt <sup>5</sup>	imm <sup>16</sup>		
beq	rs, rt, offset <sup>16</sup>	branch if equal	0x04	rs <sup>5</sup>	rt <sup>5</sup>	offset <sup>16</sup>		
bne	rs, rt, offset <sup>16</sup>	branch not equal	0x05	rs <sup>5</sup>	rt <sup>5</sup>	offset <sup>16</sup>		
j	address <sup>26</sup>	jump	0x02	address <sup>26</sup>				

# 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**       $\text{Reg}(\text{rd}) \leftarrow \text{Reg}(\text{rs}) + \text{Reg}(\text{rt});$        $\text{PC} \leftarrow \text{PC} + 4$

**SUB**       $\text{Reg}(\text{rd}) \leftarrow \text{Reg}(\text{rs}) - \text{Reg}(\text{rt});$        $\text{PC} \leftarrow \text{PC} + 4$

**ORI**       $\text{Reg}(\text{rt}) \leftarrow \text{Reg}(\text{rs}) | \text{zero\_ext}(\text{imm}^{16});$        $\text{PC} \leftarrow \text{PC} + 4$

**LW**       $\text{Reg}(\text{rt}) \leftarrow \text{MEM}[\text{Reg}(\text{rs}) + \text{sign\_ext}(\text{imm}^{16})];$        $\text{PC} \leftarrow \text{PC} + 4$

**SW**       $\text{MEM}[\text{Reg}(\text{rs}) + \text{sign\_ext}(\text{imm}^{16})] \leftarrow \text{Reg}(\text{rt});$        $\text{PC} \leftarrow \text{PC} + 4$

**BEQ**      if ( $\text{Reg}(\text{rs}) == \text{Reg}(\text{rt})$ )  
                 $\text{PC} \leftarrow \text{PC} + 4 + 4 \times \text{sign\_ext}(\text{offset}^{16})$   
    else  $\text{PC} \leftarrow \text{PC} + 4$

# Instruction Fetch/Execute

- ❖ **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:  $\text{data1} \leftarrow \text{Reg}(\text{rs}), \text{data2} \leftarrow \text{Extend}(\text{imm}^{16})$
  - 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:
    - if (zero)  $\text{PC} \leftarrow \text{PC} + 4 + 4 \times \text{sign\_ext}(\text{offset}^{16})$
    - else  $\text{PC} \leftarrow \text{PC} + 4$



# Instruction Fetch/Execute - cont'd

## ❖ LW

Fetch instruction:  $\text{Instruction} \leftarrow \text{MEM}[\text{PC}]$   
Fetch base register:  $\text{base} \leftarrow \text{Reg}(\text{rs})$   
Calculate address:  $\text{address} \leftarrow \text{base} + \text{sign\_extend}(\text{imm}^{16})$   
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:  $\text{address} \leftarrow \text{base} + \text{sign\_extend}(\text{imm}^{16})$   
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:  $\text{target} \leftarrow \text{PC}[31:28] \parallel \text{address}^{26} \parallel \text{'00'}$   
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

- ❖ **31 × 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

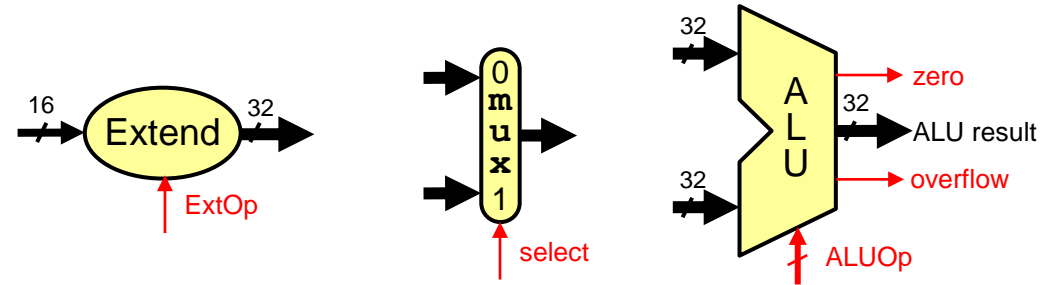
# Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ **Datapath Components and Clocking**
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ Main, ALU, and PC Control

# Components of the Datapath

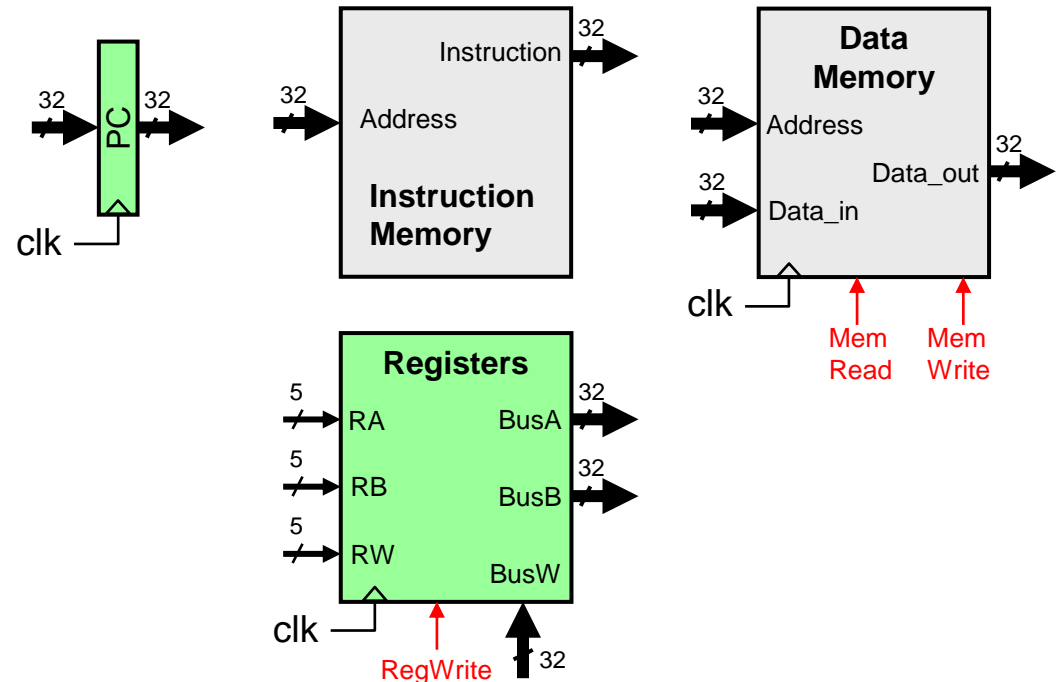
## ❖ Combinational Elements

- ❖ ALU, Adder
- ❖ Immediate extender
- ❖ Multiplexers



## ❖ Storage Elements

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



## ❖ Clocking methodology

- ❖ Timing of writes

# Register Element

## ❖ Register

- ❖ Similar to the D-type Flip-Flop

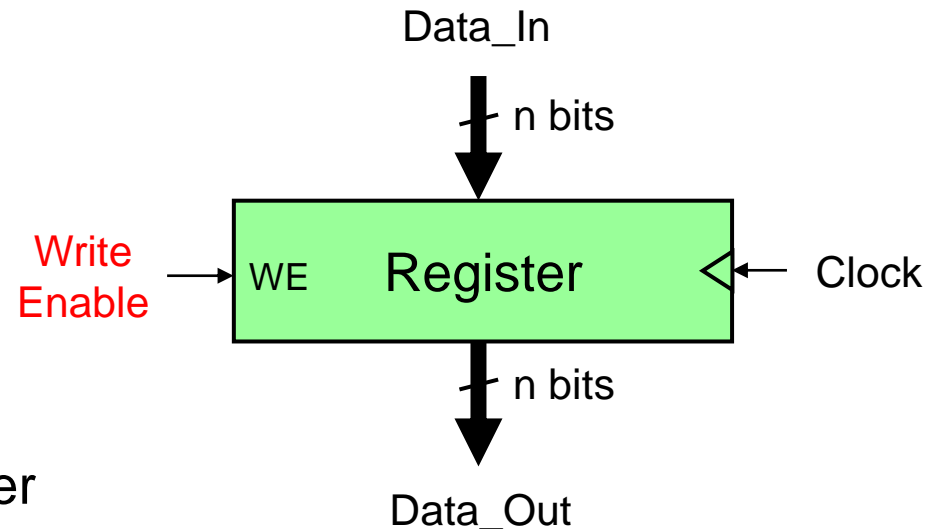
## ❖ n-bit input and output

## ❖ Write Enable (WE):

- ❖ 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

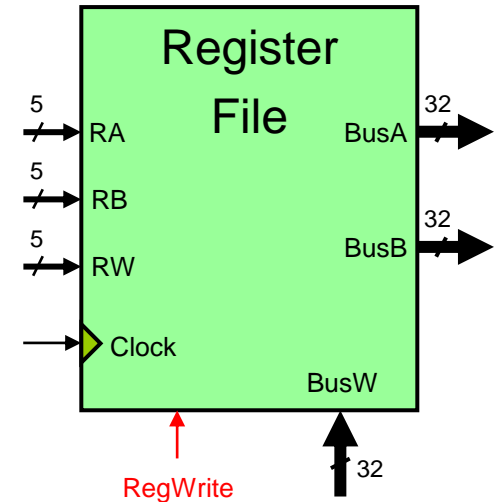
- ❖ Register File consists of  $31 \times 32$ -bit registers
  - ❖ **BusA** and **BusB**: 32-bit output busses for reading 2 registers
  - ❖ **BusW**: 32-bit input bus for writing a register when **RegWrite** is 1
  - ❖ Two registers read and one written in a cycle

- ❖ Registers are selected by:

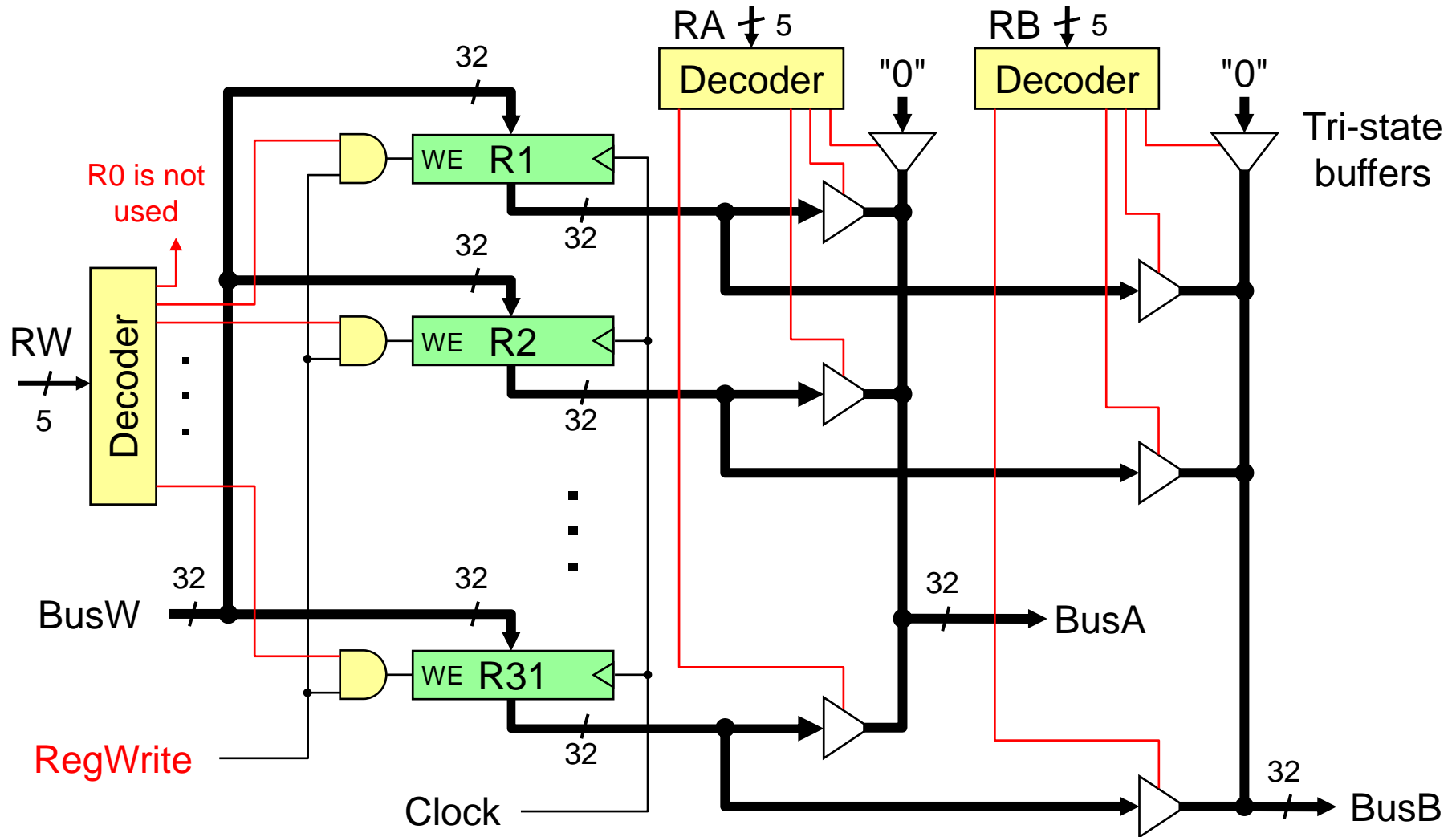
- ❖ **RA** selects register to be **read** on **BusA**
- ❖ **RB** selects register to be **read** on **BusB**
- ❖ **RW** selects the register to be **written**

- ❖ 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**



# Details of the Register File



# Tri-State Buffers

- ❖ Allow multiple sources to drive a single bus

- ❖ Two Inputs:

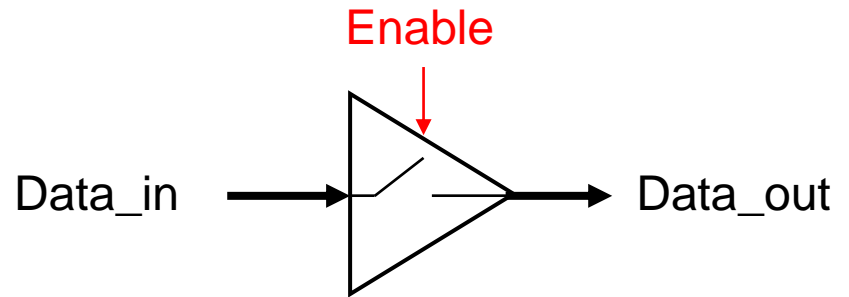
  - ❖ Data\_in

  - ❖ **Enable** (to enable output)

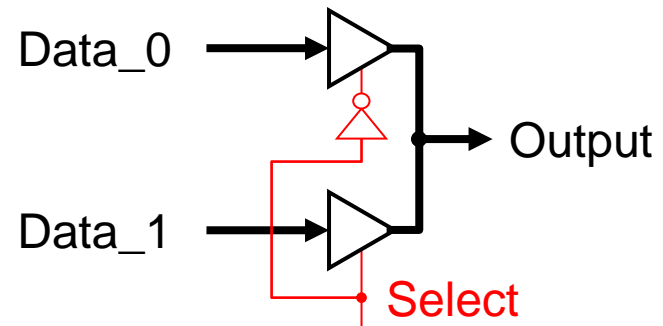
- ❖ One Output: Data\_out

  - ❖ If (**Enable**) Data\_out = Data\_in

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

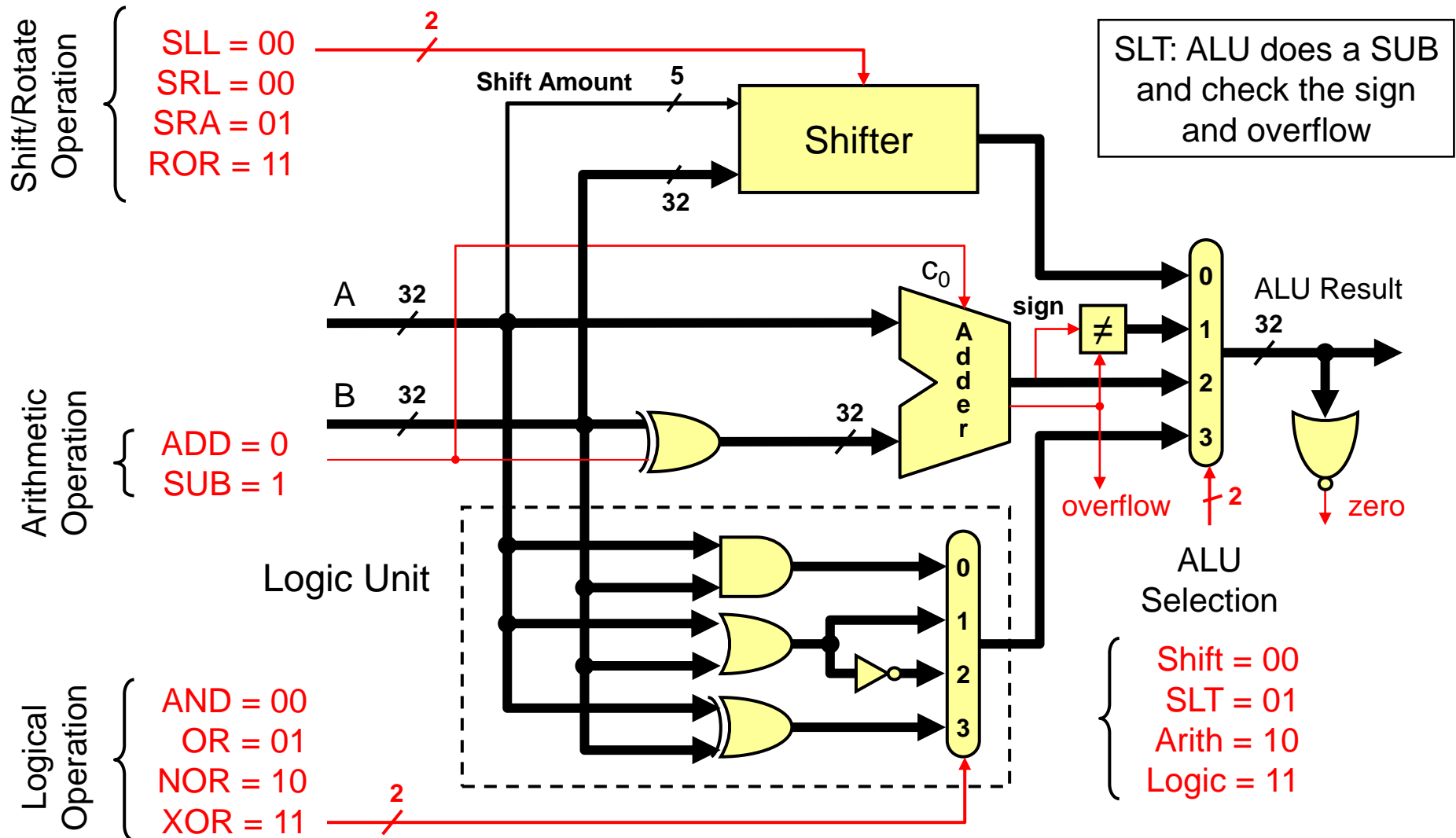


- ❖ Tri-state buffers can be used to build multiplexors



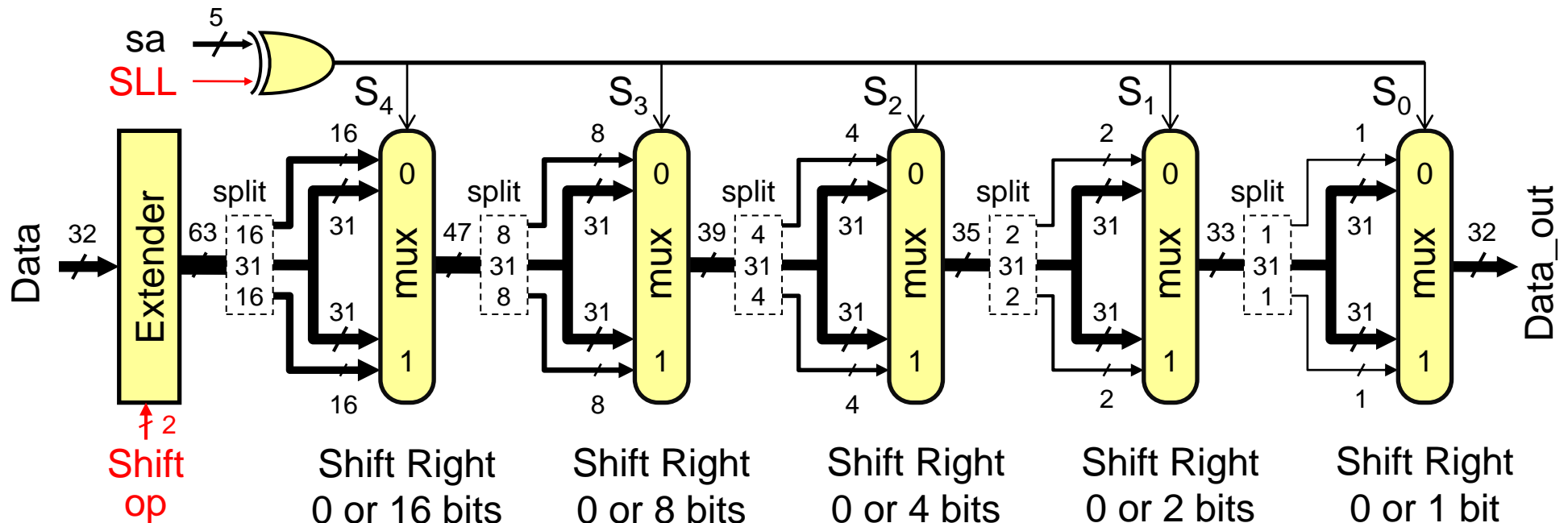


# Building a Multifunction ALU



# Details of the Shifter

- ❖ Implemented with multiplexers and wiring
- ❖ Shift Operation can be: **SLL**, **SRL**, **SRA**, or **ROR**
- ❖ Input Data is extended to 63 bits according to **Shift Op**
- ❖ The 63 bits are shifted right according to  $S_4S_3S_2S_1S_0$



# Details of the Shifter - cont'd

- ❖ Input data is extended from 32 to 63 bits as follows:
  - ✧ If shift op = SRL then  $\text{ext\_data}[62:0] = 0^{31} \parallel \text{data}[31:0]$
  - ✧ If shift op = SRA then  $\text{ext\_data}[62:0] = \text{data}[31]^{31} \parallel \text{data}[31:0]$
  - ✧ If shift op = ROR then  $\text{ext\_data}[62:0] = \text{data}[30:0] \parallel \text{data}[31:0]$
  - ✧ If shift op = SLL then  $\text{ext\_data}[62:0] = \text{data}[31:0] \parallel 0^{31}$
- ❖ For SRL, the 32-bit input data is zero-extended to 63 bits
- ❖ For SRA, the 32-bit input data is sign-extended to 63 bits
- ❖ For ROR, 31-bit extension = lower 31 bits of data
- ❖ Then, shift right according to the shift amount
- ❖ As the extended data is shifted right, the upper bits will be: 0 (SRL), sign-bit (SRA), or lower bits of data (ROR)

# Implementing Shift Left Logical

- ❖ The wiring of the above shifter dictates a right shift
- ❖ However, we can convert a left shift into a right shift
- ❖ For SLL, 31 zeros are appended to the right of data
  - ✧ To shift left by 0 is equivalent to shifting right by 31
  - ✧ To shift left by 1 is equivalent to shifting right by 30
  - ✧ To shift left by 31 is equivalent to shifting right by 0
  - ✧ Therefore, for SLL use the **1's complement** of the shift amount
- ❖ ROL is equivalent to ROR if we use  $(32 - \text{rotate amount})$
- ❖ ROL by 10 bits is equivalent to ROR by  $(32-10) = 22$  bits
- ❖ Therefore, software can convert ROL to ROR

# Instruction and Data Memories

## ❖ Instruction memory needs only provide read access

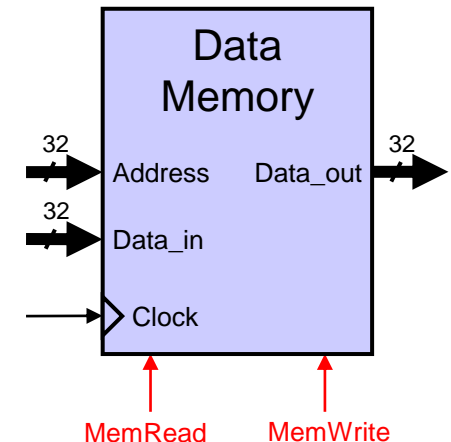
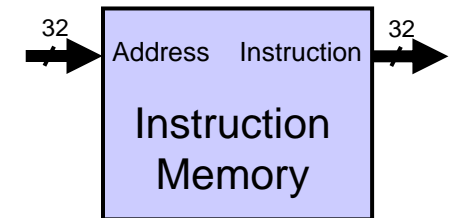
- ✧ Because datapath does not write instructions
- ✧ Behaves as combinational logic for read
- ✧ **Address** selects **Instruction** after **access time**

## ❖ 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

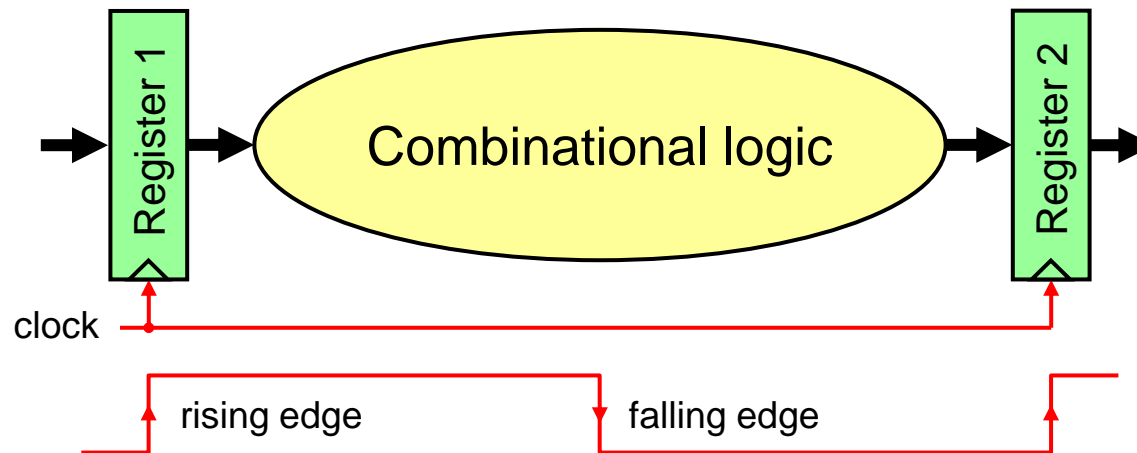
## ❖ Separate instruction and data memories

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



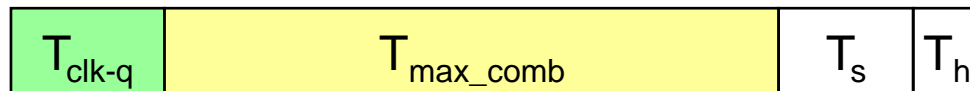
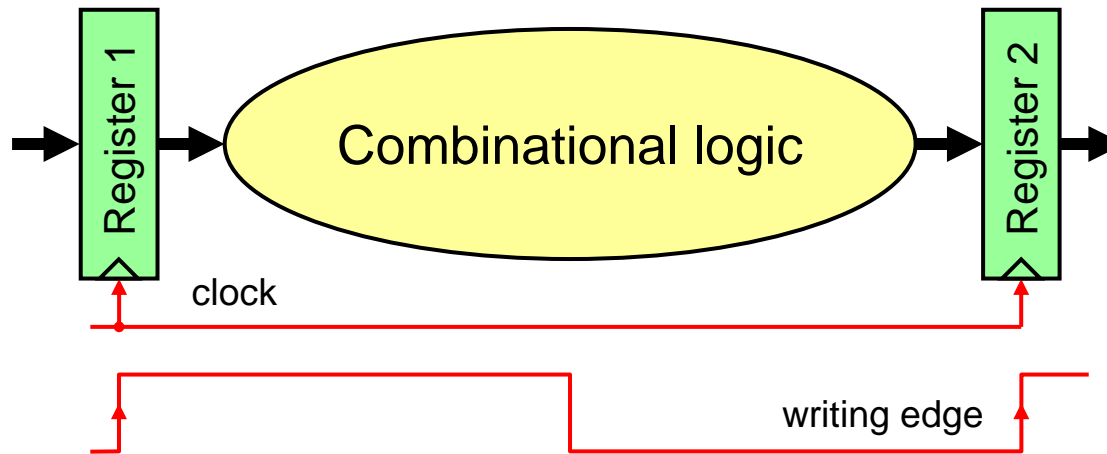
# Clocking Methodology

- ❖ 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



$$T_{\text{cycle}} \geq T_{\text{clk-q}} + T_{\text{max\_comb}} + T_s$$

- ❖  $T_{\text{clk-q}}$ : 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
- ❖ Hold time ( $T_h$ ) is normally satisfied since  $T_{\text{clk-q}} > T_h$

# 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 a clock skew, the clock cycle time is increased

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

- ❖ Clock skew is reduced by balancing the clock delays

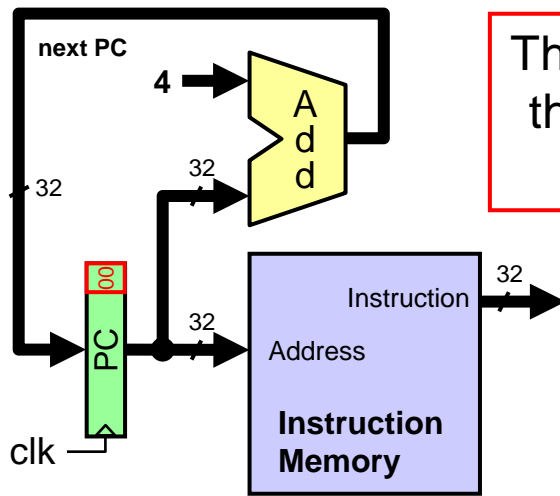


# Next ...

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath**
- ❖ Controlling the Execution of Instructions**
- ❖ Main, ALU, and PC Control

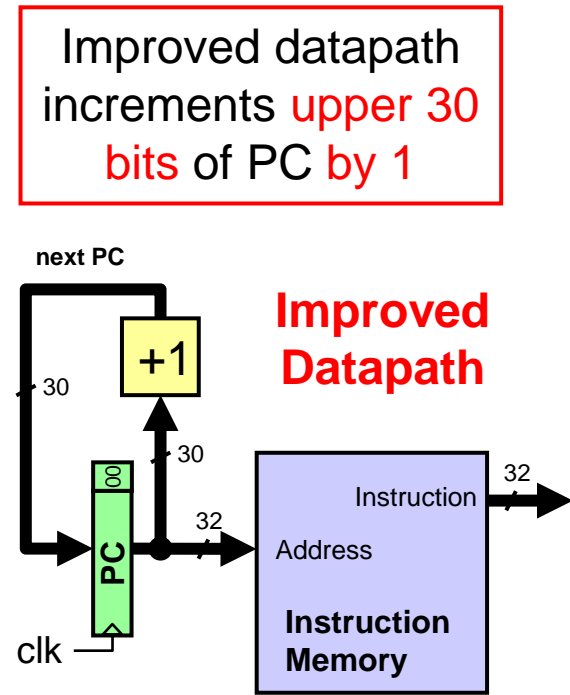
# Instruction Fetching Datapath

- ❖ We can now assemble 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

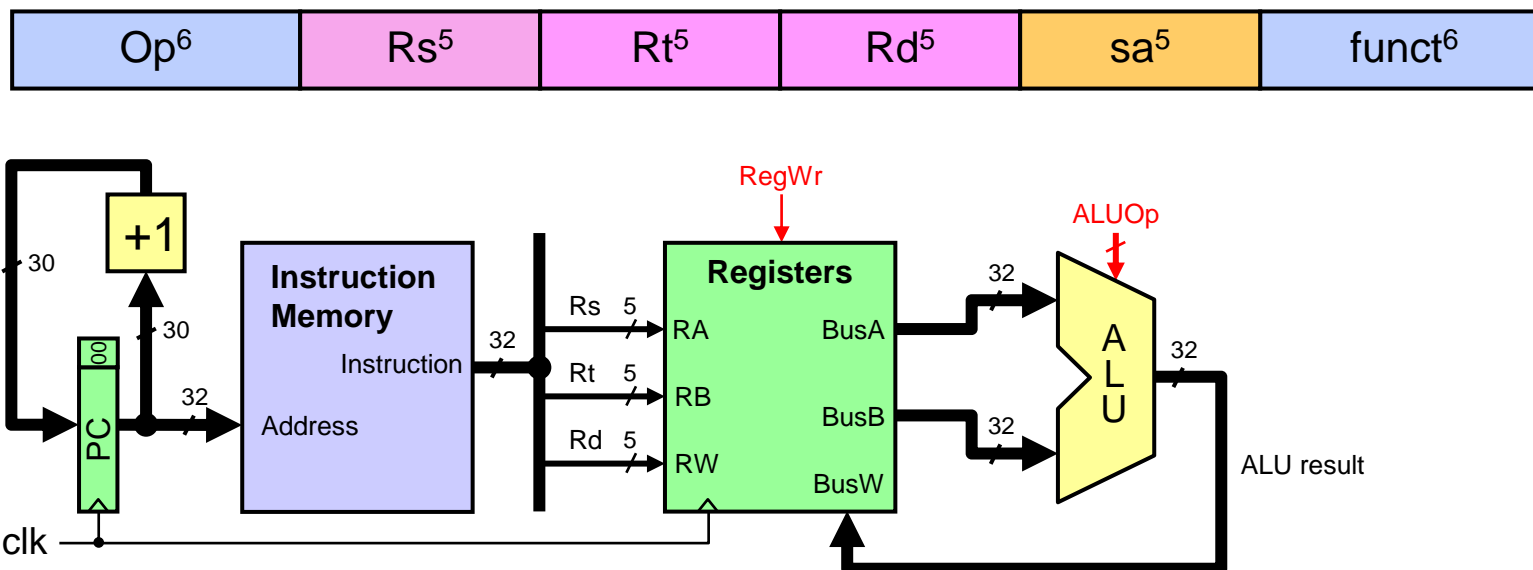
Datapath does not handle branch or jump instructions



Improved datapath increments upper 30 bits of PC by 1

Improved Datapath

# Datapath for R-type Instructions



$Rs$  and  $Rt$  fields select two registers to read.  $Rd$  field selects register to write

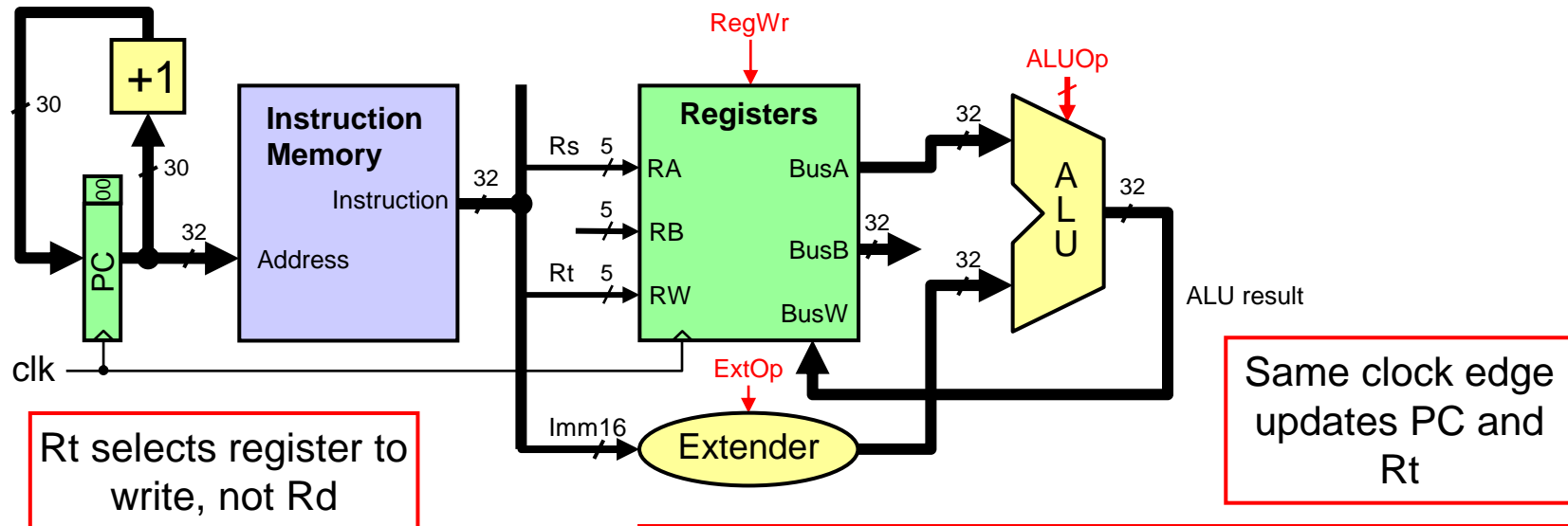
BusA & BusB provide data input to ALU. ALU result is connected to BusW

Same clock updates PC and  $Rd$  register

## ❖ Control signals

- ❖  $ALUOp$  is the ALU operation as defined in the  $funct$  field for R-type
- ❖  $RegWr$  is used to enable the writing of the ALU result

# Datapath for I-type ALU Instructions



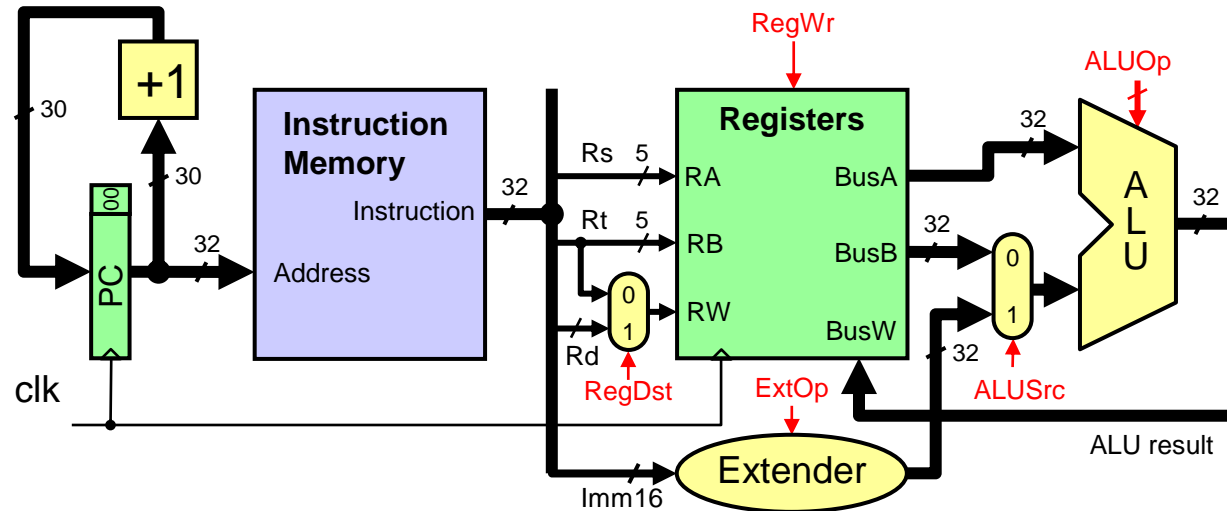
Rt selects register to write, not Rd

Second ALU input comes from the extended immediate. RB and BusB are not used

## ❖ Control signals

- ❖ ALUOp is derived from the Op field for I-type instructions
- ❖ RegWr 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



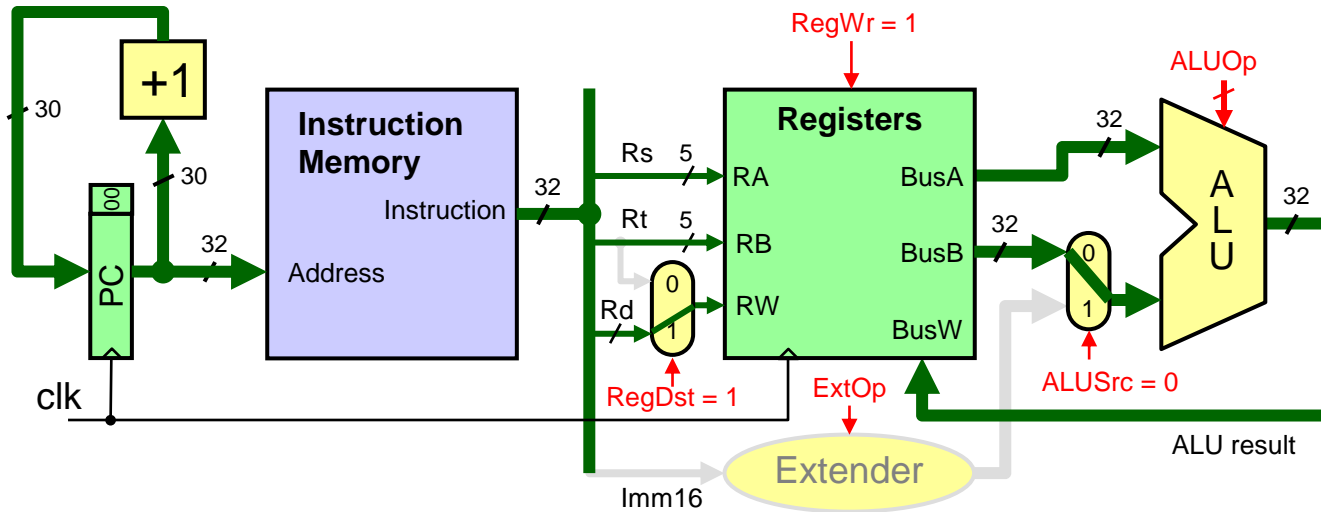
A mux selects RW as either Rt or Rd

Another mux selects 2<sup>nd</sup> ALU input as either data on BusB or the extended immediate

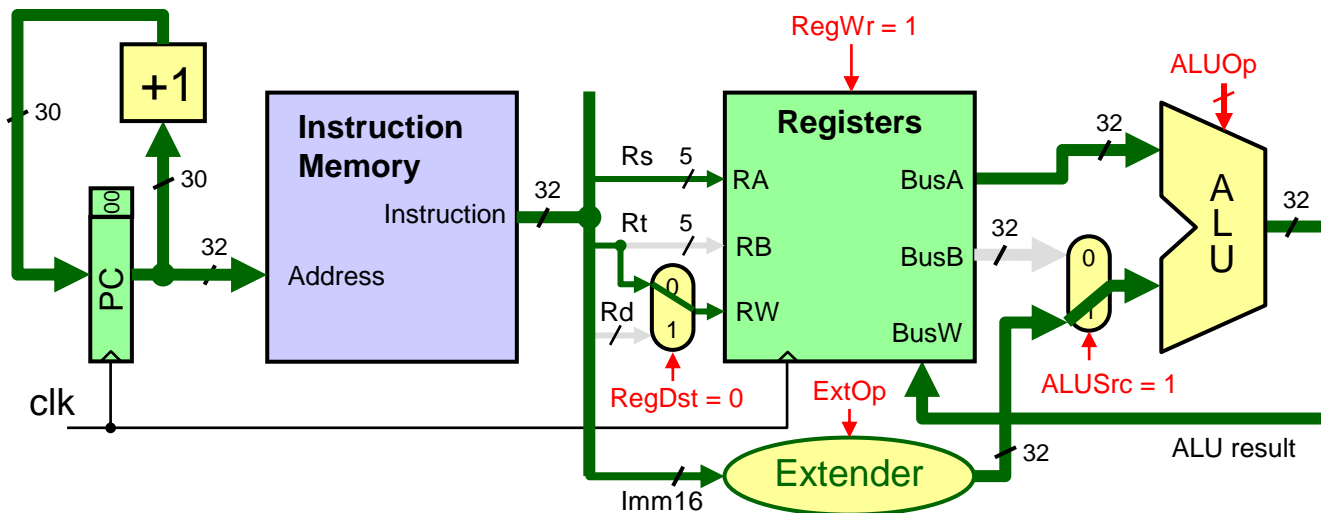
## ❖ Control signals

- ❖ **ALUOp** is derived from either the **Op** or the **funct** field
- ❖ **RegWr** 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



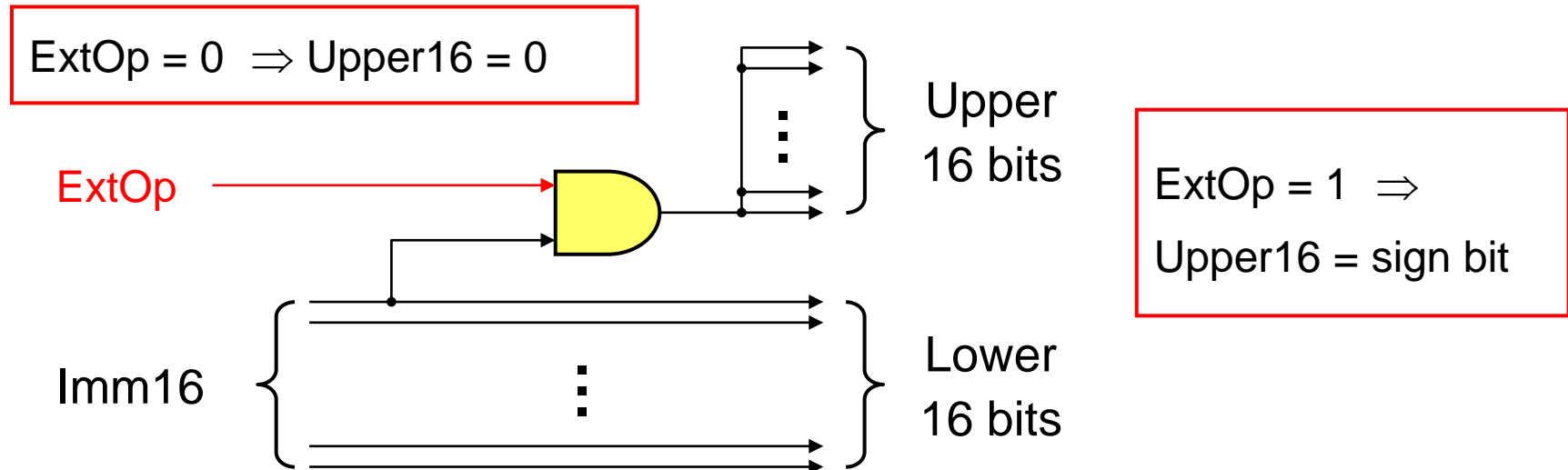
For R-type ALU instructions, **RegDst** is '1' to select Rd on RW and **ALUSrc** is '0' to select BusB as second ALU input. The active part of datapath is shown in **green**



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**

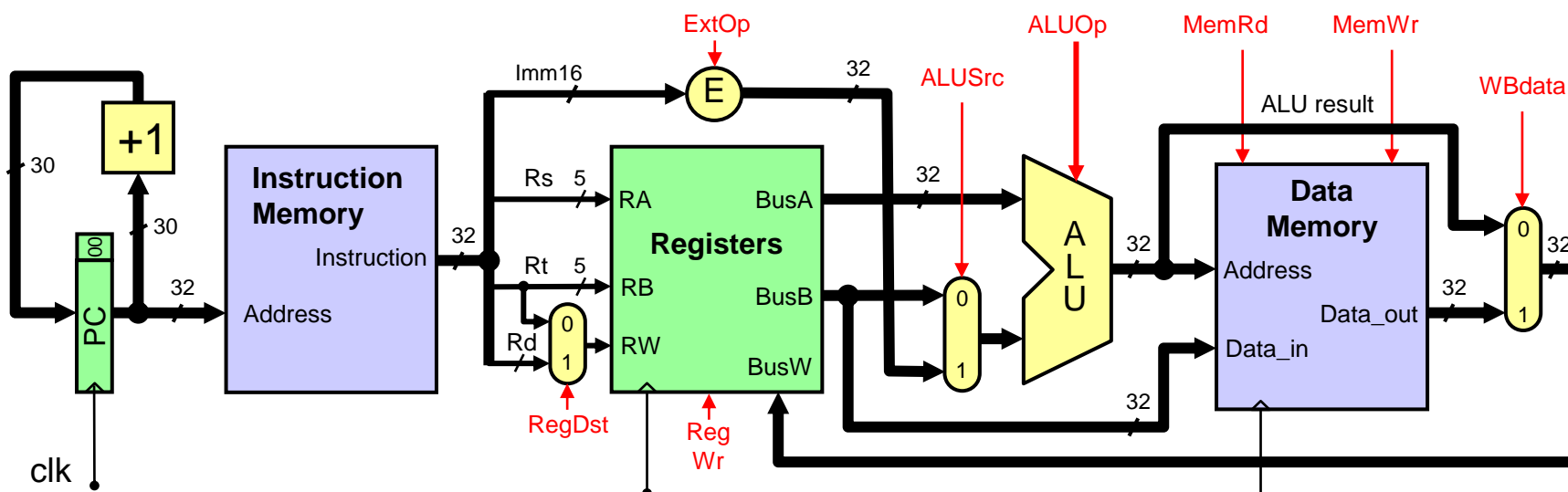
# Details of the Extender

- ❖ 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

- ❖ **MemRd** for load instructions

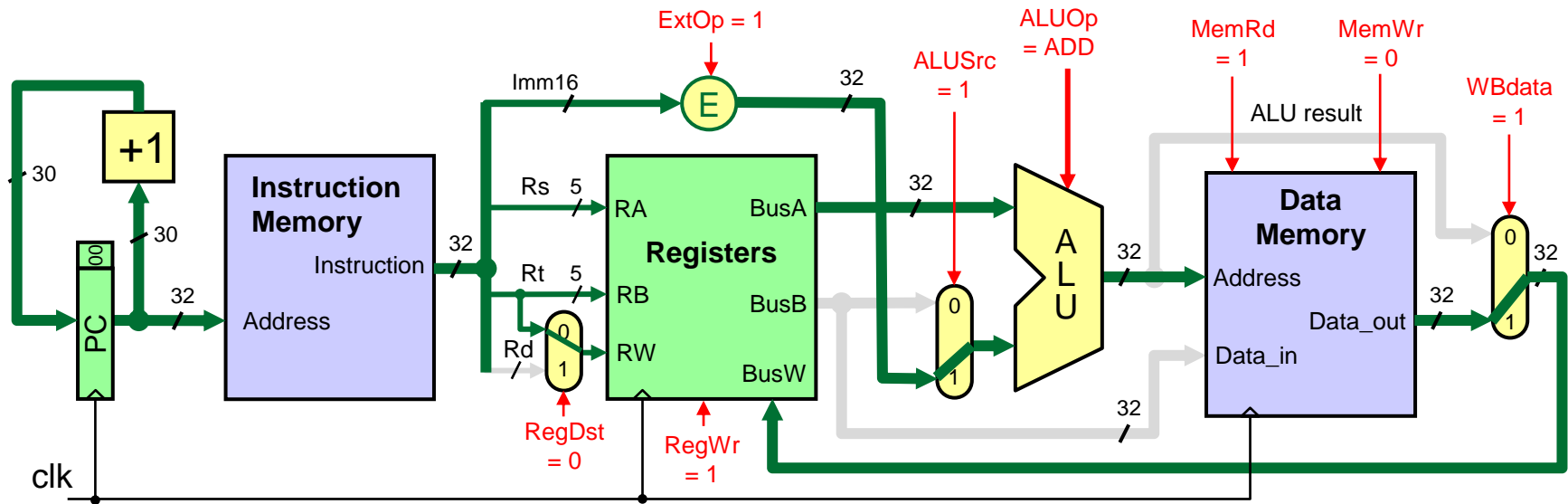
- ❖ **MemWr** for store instructions

- ❖ **WBdata** 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



RegDst = '0' selects Rt as destination register

RegWr = '1' to enable writing of register file

ExtOp = 1 to sign-extend Immediate16 to 32 bits

ALUSrc = '1' selects extended immediate as second ALU input

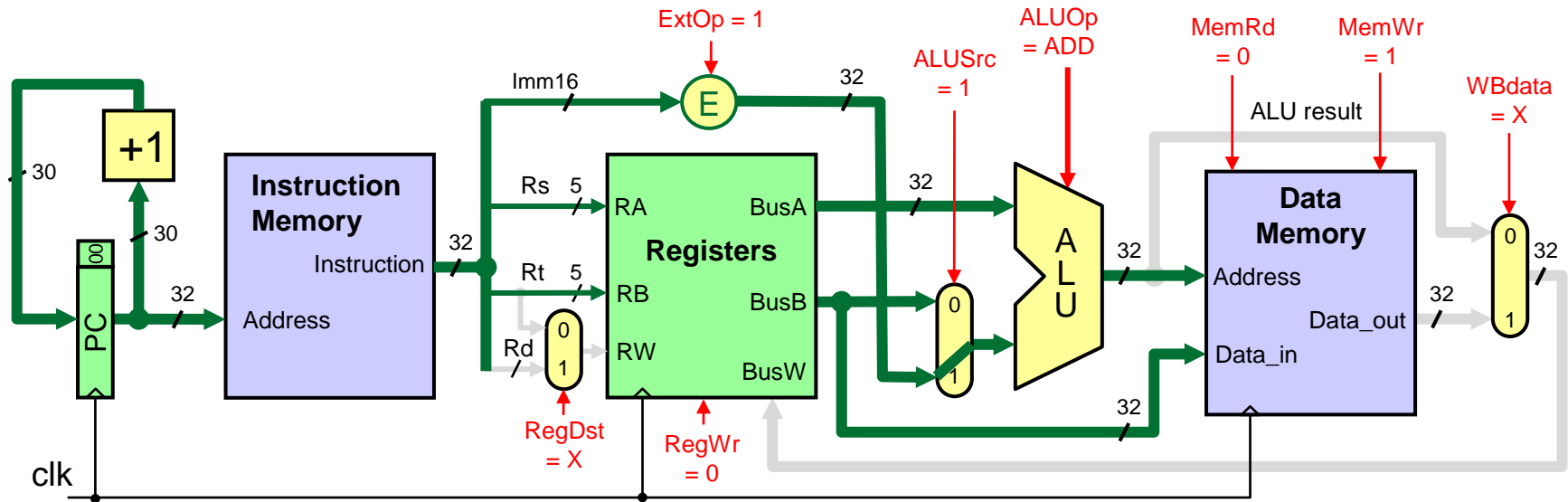
ALUOp = 'ADD' to calculate data memory address as  $\text{Reg}(\text{Rs}) + \text{sign-extend}(\text{Imm16})$

MemRd = '1' to read data memory

WBdata = '1' places the data read from memory on BusW

Clock edge updates PC and Register Rt

# Controlling the Execution of Store



RegDst = 'X' because no register is written

RegWr = '0' to disable writing of register file

ExtOp = 1 to sign-extend Immediate16 to 32 bits

ALUSrc = '1' selects extended immediate as second ALU input

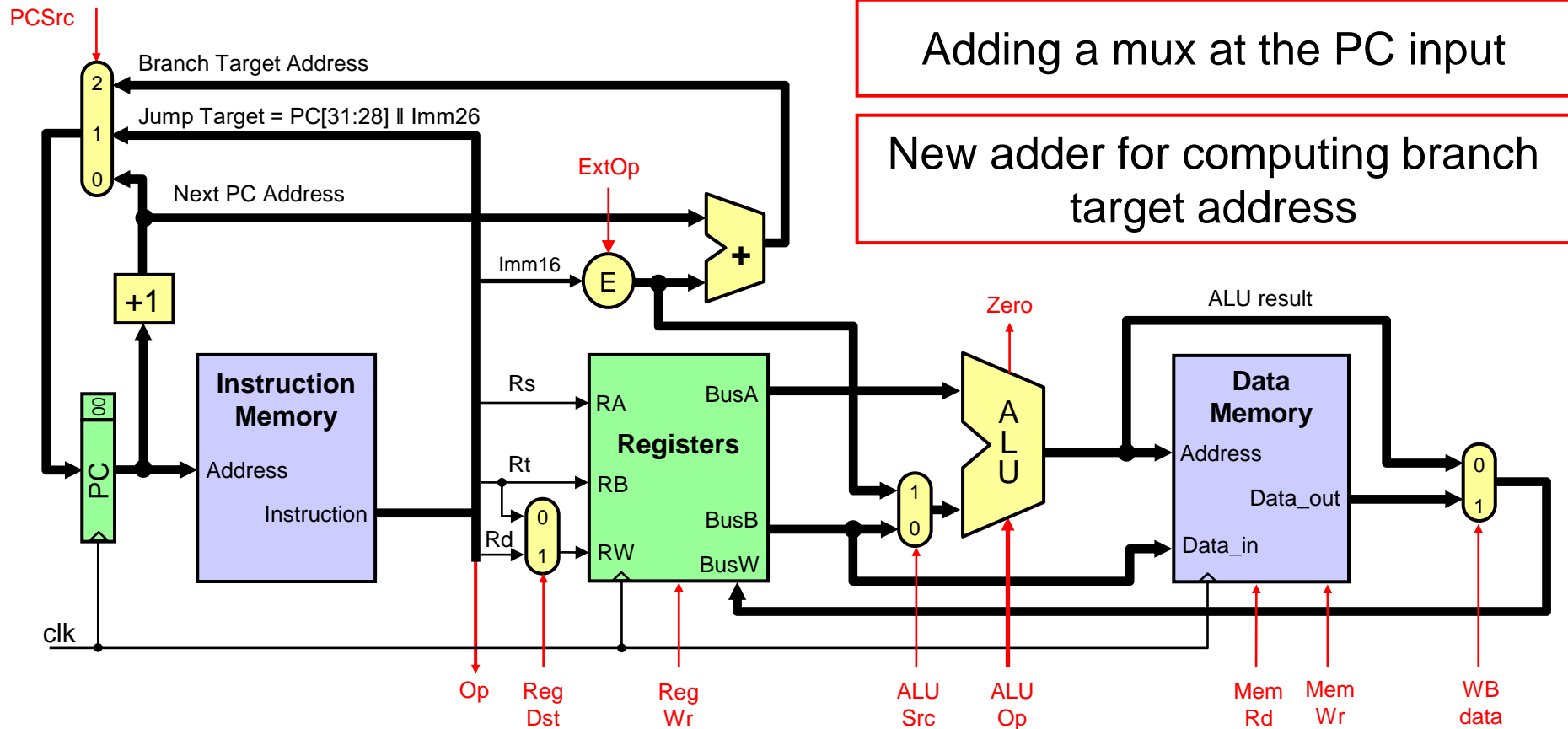
ALUOp = 'ADD' to calculate data memory address as  $\text{Reg}(Rs) + \text{sign-extend}(\text{Imm16})$

MemWr = '1' to write data memory

WBdata = 'X' because don't care what data is put on BusW

Clock edge updates PC and Data Memory

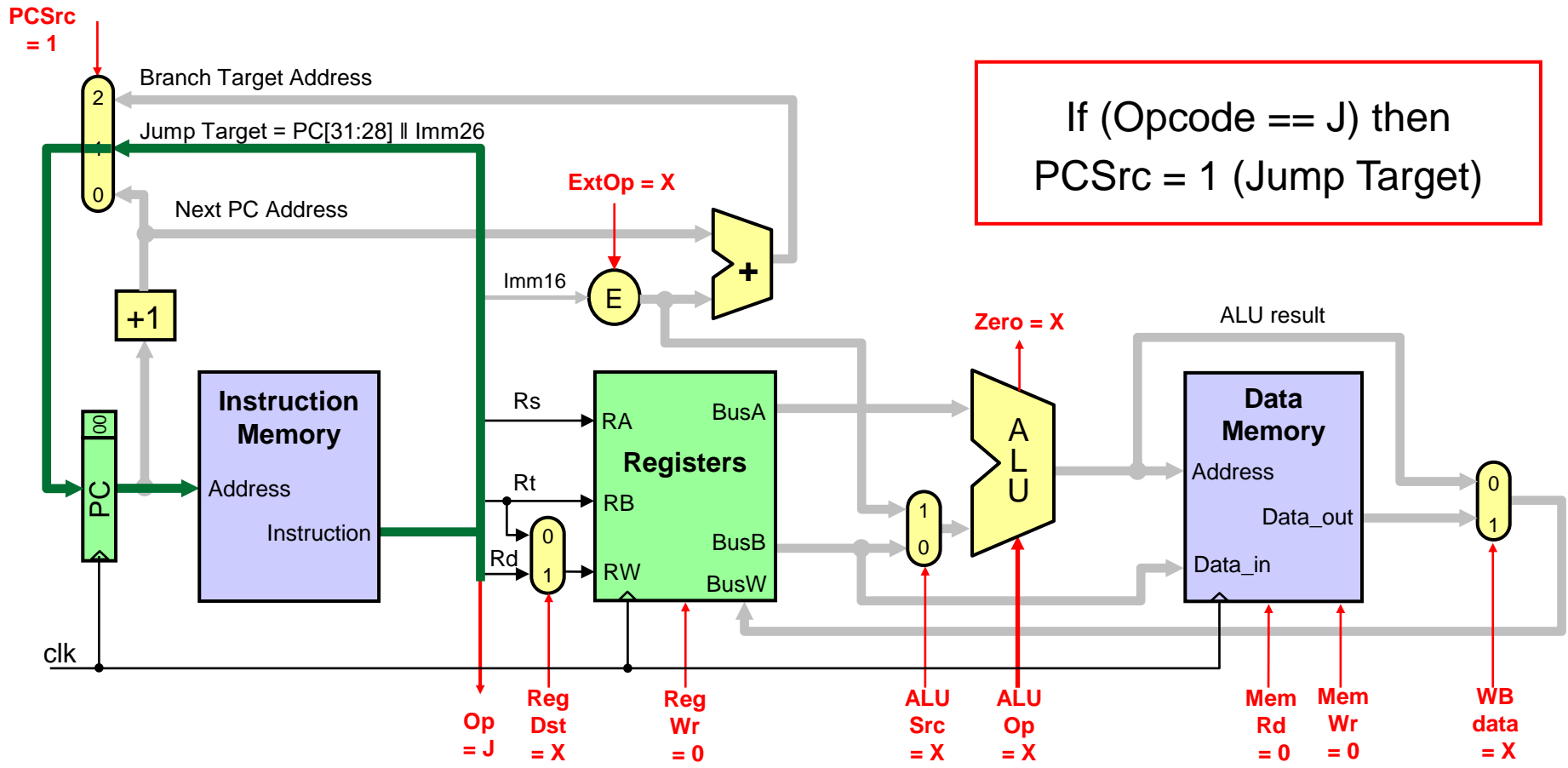
# Adding Jump and Branch to Datapath



## ❖ Additional Control Signals

- ❖ **PCSrc** for PC control: **1** for a jump and **2** for a taken branch
- ❖ **Zero** flag for branch control: whether branch is taken or not

# Controlling the Execution of a Jump

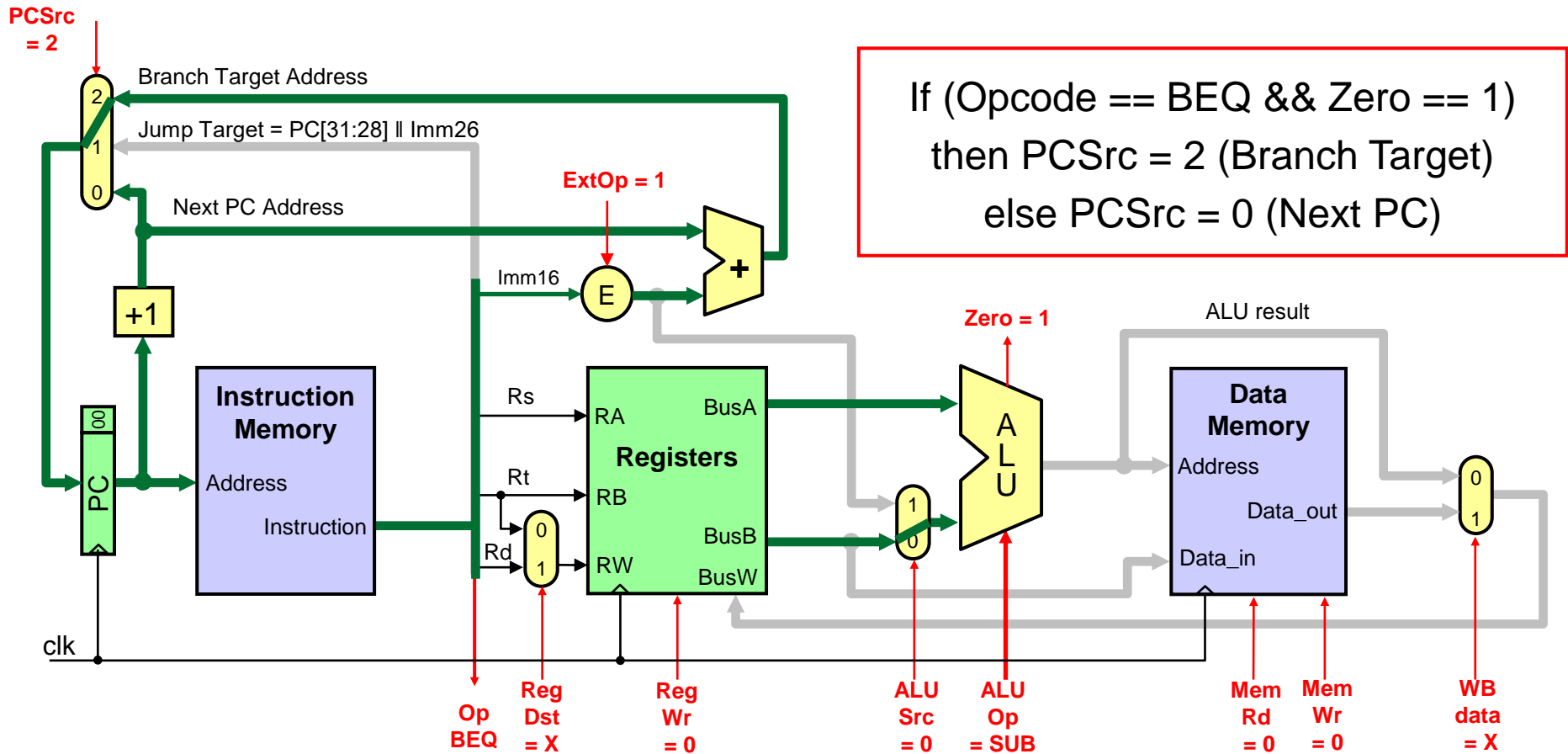


If (Opcode == J) then  
PCSrc = 1 (Jump Target)

MemRd = MemWr = RegWr = 0, Don't care about other control signals

Clock edge updates PC register only

# Controlling the Execution of a Branch



If (Opcode == BEQ && Zero == 1)  
then PCSrc = 2 (Branch Target)  
else PCSrc = 0 (Next PC)

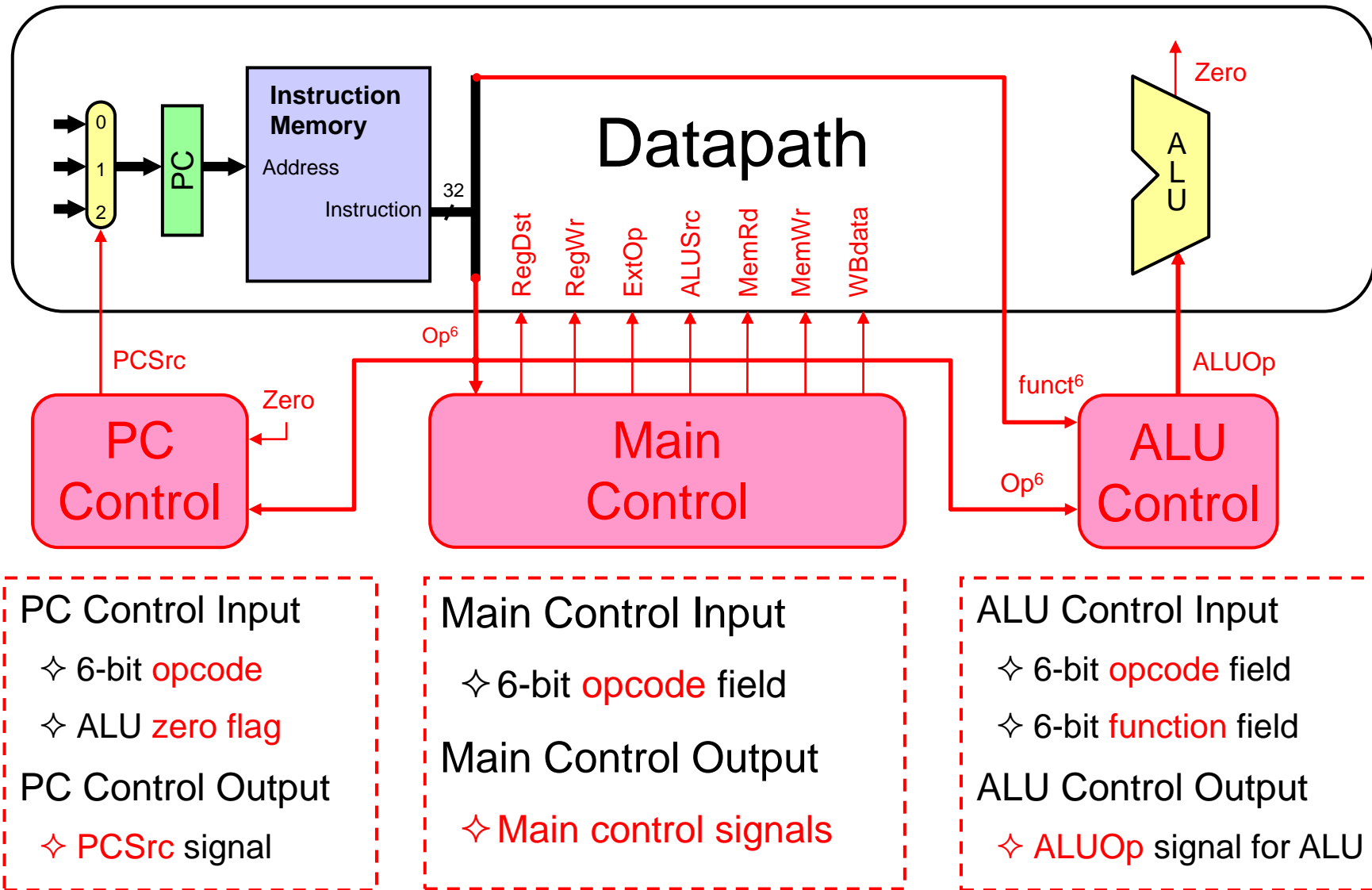
ALUSrc = 0, ALUOp = SUB, ExtOp = 1, MemRd = MemWr = RegWr = 0

Clock edge updates PC register only

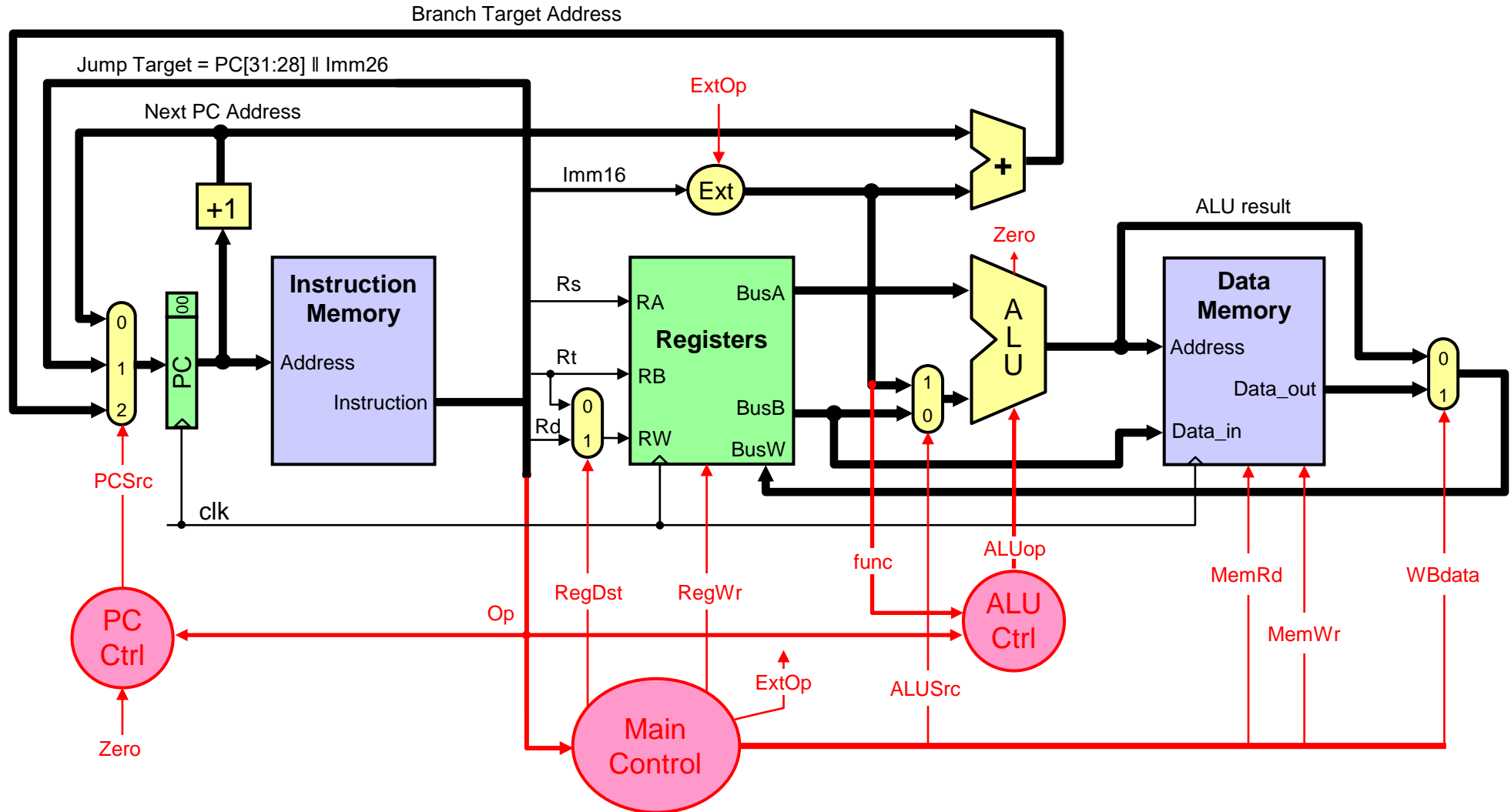
# Next . . .

- ❖ Designing a Processor: Step-by-Step
- ❖ Datapath Components and Clocking
- ❖ Assembling an Adequate Datapath
- ❖ Controlling the Execution of Instructions
- ❖ **Main, ALU, and PC Control**

# Main, ALU, and PC Control



# Single-Cycle Datapath + Control





# Main Control Signals

Signal	Effect when '0'	Effect when '1'
RegDst	Destination register = Rt	Destination register = Rd
RegWr	No register is written	Destination register (Rt or Rd) is written with the data on BusW
ExtOp	16-bit immediate is zero-extended	16-bit immediate is sign-extended
ALUSrc	Second ALU operand is the value of register Rt that appears on BusB	Second ALU operand is the value of the extended 16-bit immediate
MemRd	Data memory is NOT read	Data memory is read Data_out $\leftarrow$ Memory[address]
MemWr	Data Memory is NOT written	Data memory is written Memory[address] $\leftarrow$ Data_in
WBdata	BusW = ALU result	BusW = Data_out from Memory

# Main Control Truth Table

Op	RegDst	RegWr	ExtOp	ALUSrc	MemRd	MemWr	WBdata
R-type	1 = Rd	1	X	0 = BusB	0	0	0 = ALU
ADDI	0 = Rt	1	1 = sign	1 = Imm	0	0	0 = ALU
SLTI	0 = Rt	1	1 = sign	1 = Imm	0	0	0 = ALU
ANDI	0 = Rt	1	0 = zero	1 = Imm	0	0	0 = ALU
ORI	0 = Rt	1	0 = zero	1 = Imm	0	0	0 = ALU
XORI	0 = Rt	1	0 = zero	1 = Imm	0	0	0 = ALU
LW	0 = Rt	1	1 = sign	1 = Imm	1	0	1 = Mem
SW	X	0	1 = sign	1 = Imm	0	1	X
BEQ	X	0	1 = sign	0 = BusB	0	0	X
BNE	X	0	1 = sign	0 = BusB	0	0	X
J	X	0	X	X	0	0	X

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

# Logic Equations for Main Control Signals

RegDst = R-type

RegWrite =  $(SW + BEQ + BNE + J)$

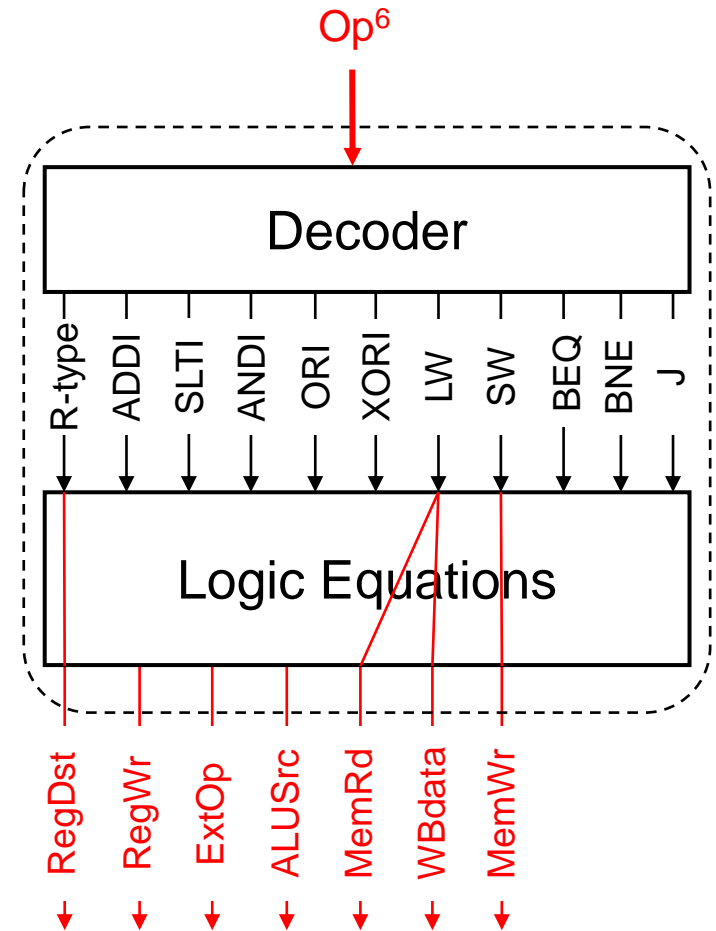
ExtOp =  $(ANDI + ORI + XORI)$

ALUSrc =  $(R\text{-type} + BEQ + BNE)$

MemRd = LW

MemWr = SW

WBdata = LW



# ALU Control Truth Table

Op	funct	ALUOp	4-bit Code
R-type	AND	AND	11 00
R-type	OR	OR	11 01
R-type	XOR	XOR	11 10
R-type	ADD	ADD	10 00
R-type	SUB	SUB	10 10
R-type	SLT	SLT	01 10
ADDI	X	ADD	10 00
SLTI	X	SLT	01 10
ANDI	X	AND	11 00
ORI	X	OR	11 01
XORI	X	XOR	11 10
LW	X	ADD	10 00
SW	X	ADD	10 00
BEQ	X	SUB	10 10
BNE	X	SUB	10 10
J	X	X	X

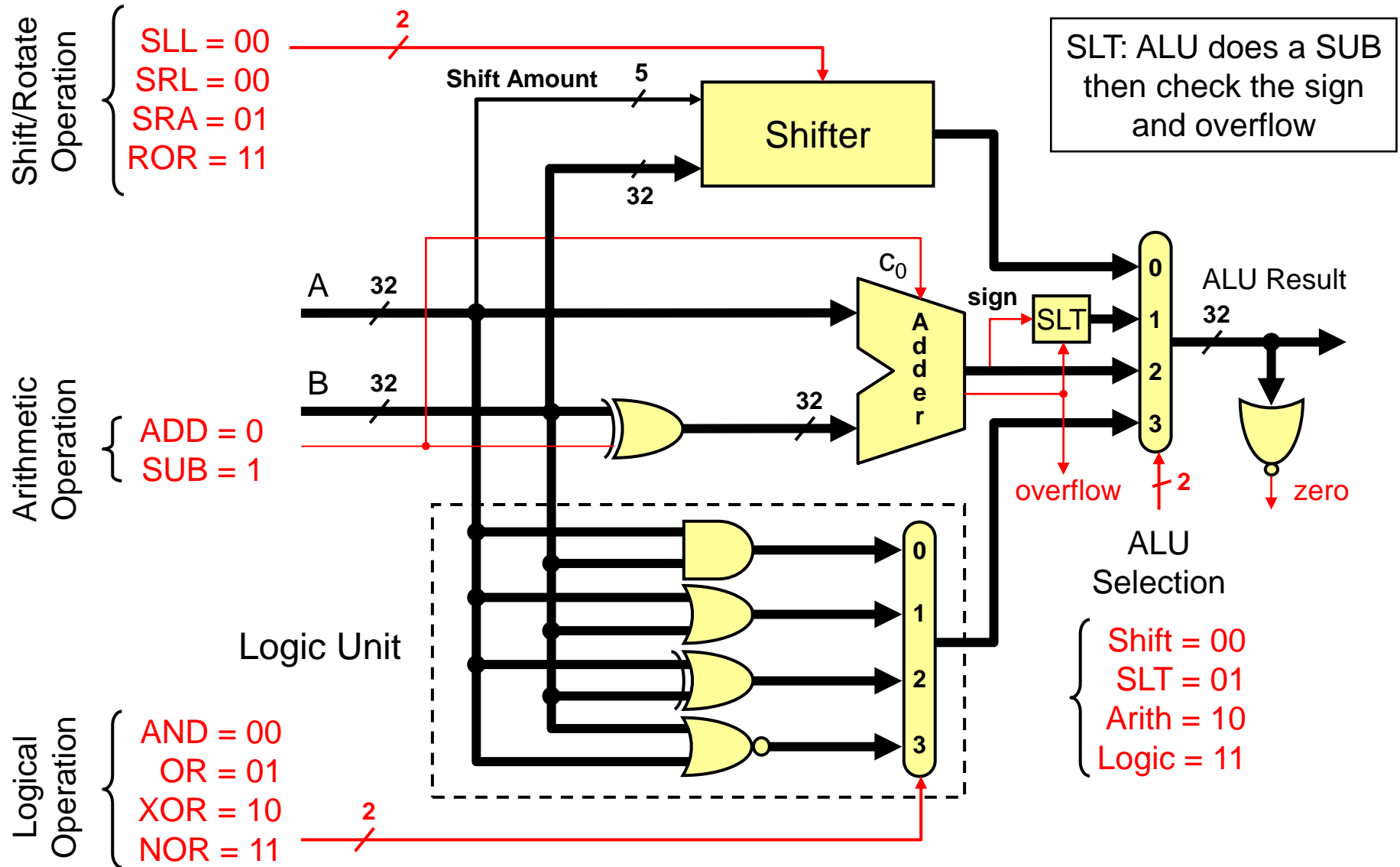
The 4-bit codes define the binary ALU operations.

The 4-bit codes match the ALU implementation.

Upper 2 bits = ALU selection  
Lower 2 bits = Logic or Arith op.

The 4-bit codes can be derived easily from the opcode and function code.

# Multifunction ALU



# PC Control Truth Table

Op	Zero flag	PCSrc
R-type	X	0 = Increment PC
J	X	1 = Jump Target Address
BEQ	0	0 = Increment PC
BEQ	1	2 = Branch Target Address
BNE	0	2 = Branch Target Address
BNE	1	0 = Increment PC
Other than Jump or Branch	X	0 = Increment PC

The ALU Zero flag is used by BEQ and BNE instructions

# PC Control Logic

❖ The PC control logic can be described as follows:

```
if (Op == J) PCSrc = 1;
```

```
else if ((Op == BEQ && Zero == 1) ||
```

```
        (Op == BNE && Zero == 0)) PCSrc = 2;
```

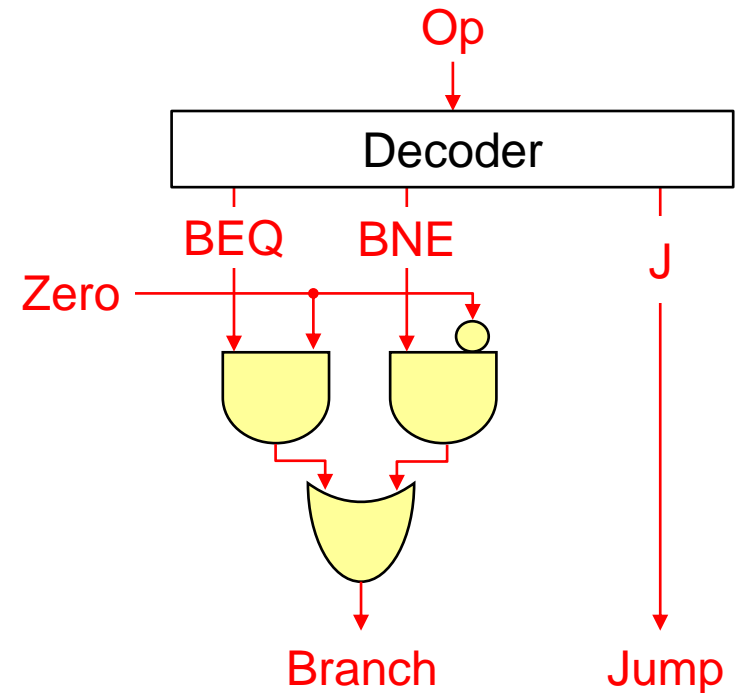
```
else PCSrc = 0;
```

Branch = (BEQ . Zero) + (BNE .  $\overline{\text{Zero}}$ )

Branch = 1, Jump = 0 → PCSrc = 2

Branch = 0, Jump = 1 → PCSrc = 1

Branch = 0, Jump = 0 → PCSrc = 0



# 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 the same size
- ✧ Source registers always in the same place
- ✧ Immediate constants are of same size and same location
- ✧ Operations are always on registers/immediates