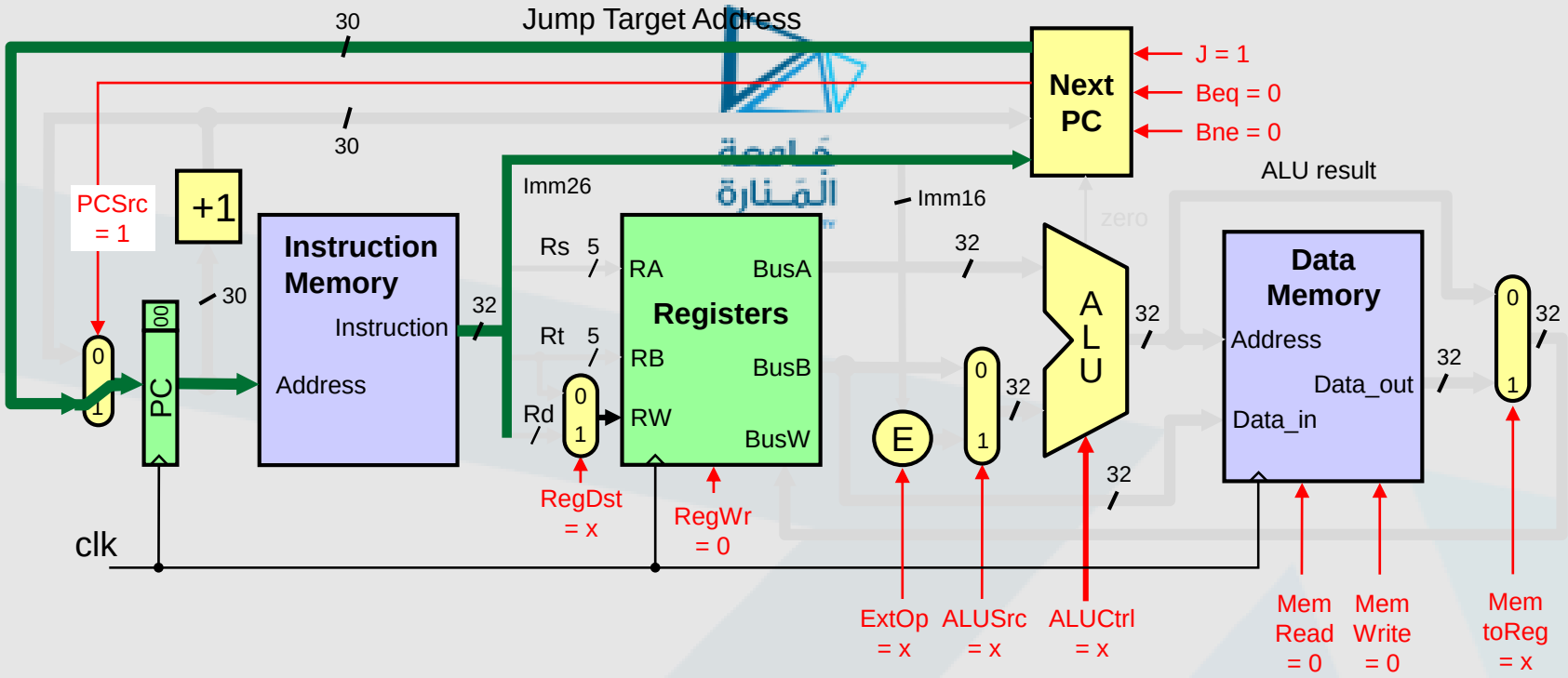


A Simplified MIPS Implementation

Processor Data path and Control

Computer Architecture

Lectures 7-8

Mechatronics Engineering

Assistant Professor: Isam M. Asaad

Presentation 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

Introduction

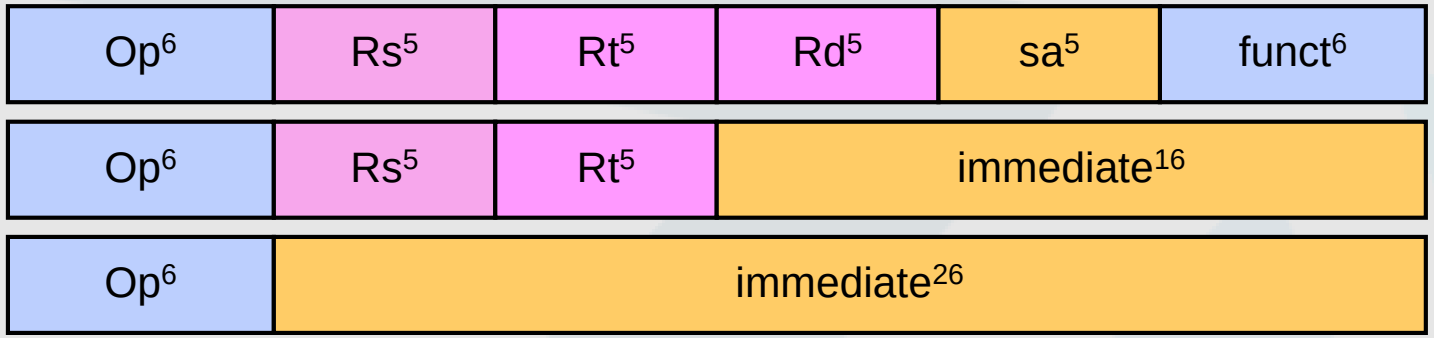
- The Processor
  - Also called the Central processor unit (CPU)
  - The active part of the computer
  - Contains
    - The Datapath
      - The component of the processor that performs (arithmetic) operations
    - Control
      - The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program

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

Review of MIPS Instruction Formats

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



- Op^6: 6-bit opcode of the instruction
- Rs^5, Rt^5, Rd^5: 5-bit source and destination register numbers
- sa^5: 5-bit shift amount used by shift instructions
- $funct^6$: 6-bit function field for R-type instructions
- $immediate^{16}$: 16-bit immediate value or address offset
- $immediate^{26}$: 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

Register Transfer Level (RTL)

- RTL is a description of data flow between registers
- All instructions are fetched from memory at address PC (Program Counter)

Instruction	RTL Description	
ADD	Reg(Rd) ← Reg(Rs) + Reg(Rt);	PC ← PC + 4
SUB	Reg(Rd) ← Reg(Rs) – Reg(Rt);	PC ← PC + 4
ORI	Reg(Rt) ← Reg(Rs) | zero_ext(Im16);	PC ← PC + 4
LW	Reg(Rt) ← MEM[Reg(Rs) + sign_ext(Im16)];	PC ← PC + 4
SW	MEM[Reg(Rs) + sign_ext(Im16)] ← Reg(Rt);	PC ← PC + 4
BEQ	if (Reg(Rs) == Reg(Rt))	PC ← PC + 4 + 4 × sign_extend(Im16)
	else	PC ← PC + 4

Instructions are Executed in Steps

❖ R-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
	Execute operation:	ALU_result ← func(data1, data2)
	Write ALU result:	Reg(Rd) ← ALU_result
	Next PC address:	PC ← PC + 4
❖ I-type	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Extend(imm16)
	Execute operation:	ALU_result ← op(data1, data2)
	Write ALU result:	Reg(Rt) ← ALU_result
	Next PC address:	PC ← PC + 4
❖ BEQ	Fetch instruction:	Instruction ← MEM[PC]
	Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
	Equality:	zero ← subtract(data1, data2)
	Branch:	if (zero) PC ← PC + 4 + 4×sign_ext(imm16)
		else PC ← PC + 4

Instruction Execution – cont'd

- **LW**

Fetch instruction:	Instruction ← MEM[PC]
Fetch base register:	base ← Reg(Rs)
Calculate address:	address ← base + sign_extend(imm16)
Read memory:	data ← MEM[address]
Write register Rt:	Reg(Rt) ← data
Next PC address:	PC ← PC + 4

- **SW**

Fetch instruction:	Instruction ← MEM[PC]
Fetch registers:	base ← Reg(Rs), data ← Reg(Rt)
Calculate address:	address ← base + sign_extend(imm16)
Write memory:	MEM[address] ← data
Next PC address:	PC ← PC + 4

- **Jump**

Fetch instruction:	Instruction ← MEM[PC]
Target PC address:	target ← PC[31:28] || Imm26 || '00'
Jump:	PC ← 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
- The Main Controller and ALU Controller
- Drawback of the single-cycle processor design

The Datapath

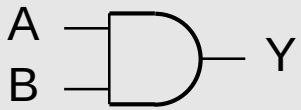
- **Datapath element**
  - A unit used to operate on or hold data within a processor
- **Types of datapath elements**
  - Elements that operate on data values (Combinational Logic)
    - The output depend only on the current inputs
  - Elements that contain state (state elements or sequential elements)
    - The outputs depend on both the inputs and the internal state
    - Has some internal storage
    - A state element has at least two inputs and one output
      - Input: The data value to be written into the element
      - Input: The clock, which determines when the data value is written
      - Output: provides the value that was written in an earlier clock cycle
        - A state element can be read at any time
    - Examples
      - D-type flip-flop
      - Memories
      - Registers

Logic Design Basics

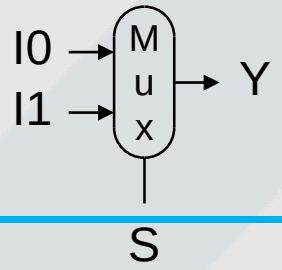
- **Information encoded in binary**
  - **Low voltage = 0, High voltage = 1**
  - **One wire per bit**
  - **Multi-bit data encoded on multi-wire buses**
- **Combinational element**
  - **Operate on data**
  - **Output is a function of input**
- **State (sequential) elements**
  - **Store information**

Combinational Elements

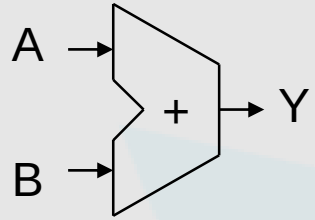
- AND-gate
  - Y = A & B



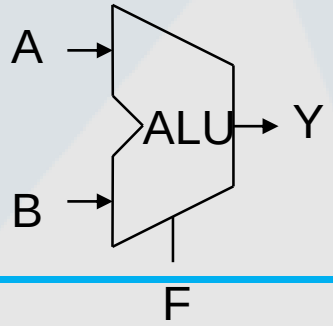
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

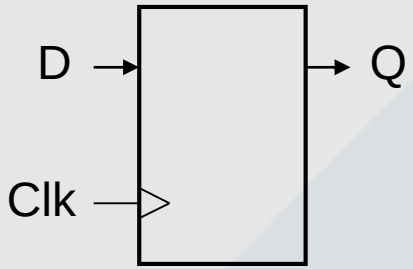


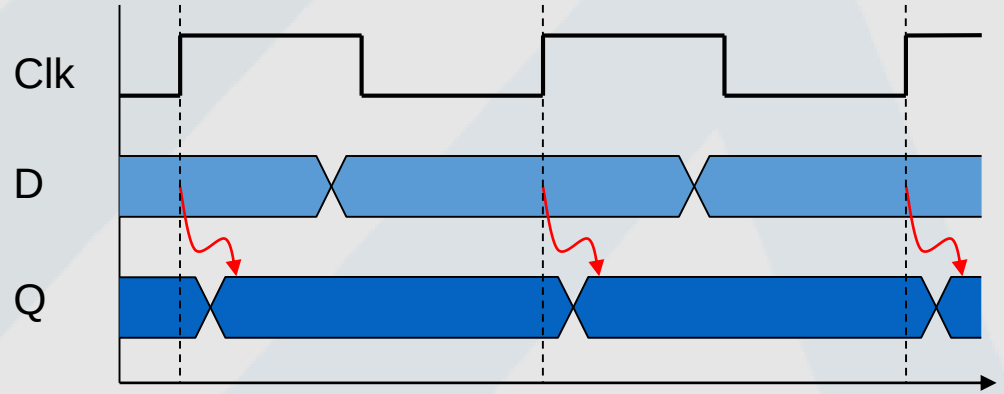
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

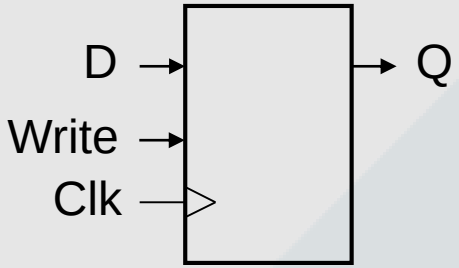
- **Register: stores data in a circuit**
  - **Uses a clock signal to determine when to update the stored value**
  - **Edge-triggered: update when Clk changes from 0 to 1**

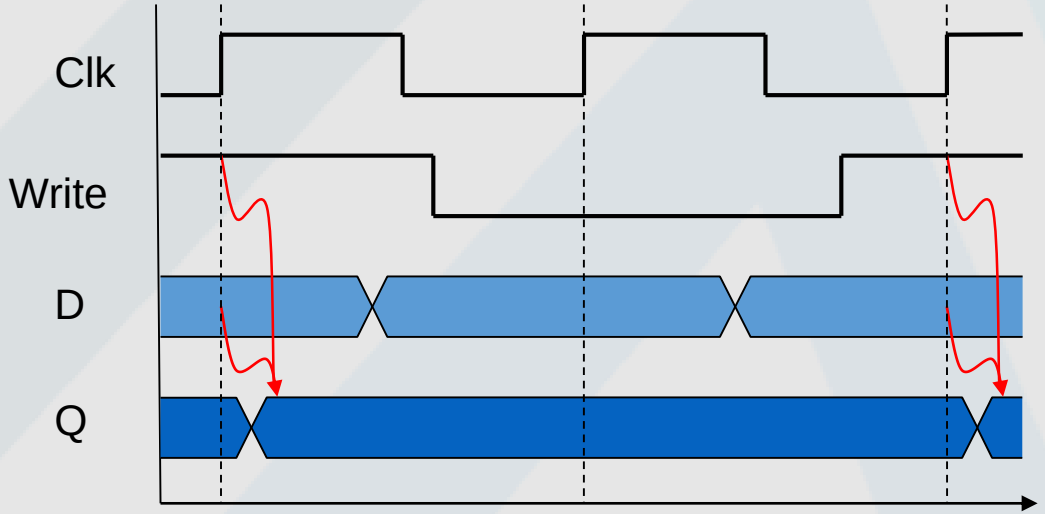




Sequential Elements

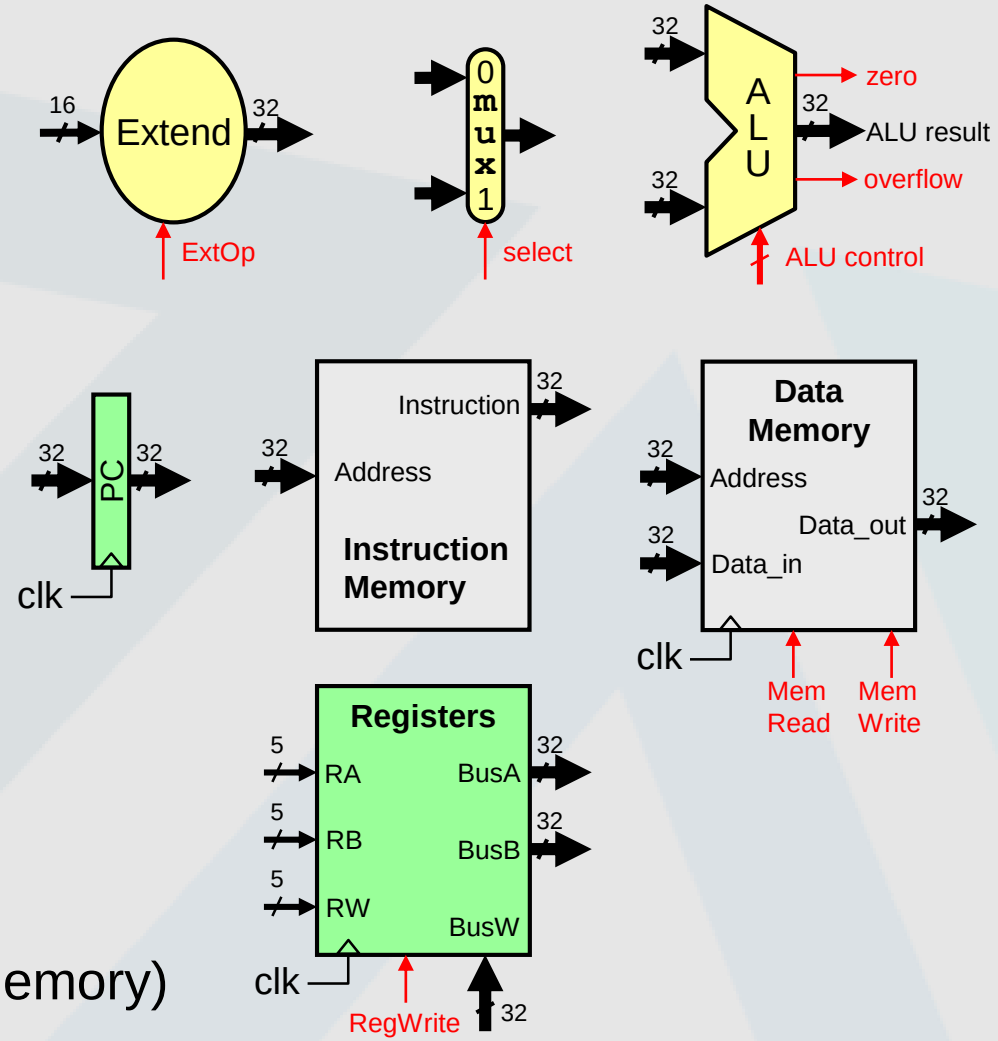
- **Register with write control**
  - **Only updates on clock edge when write control input is 1**
  - **Used when stored value is required later**





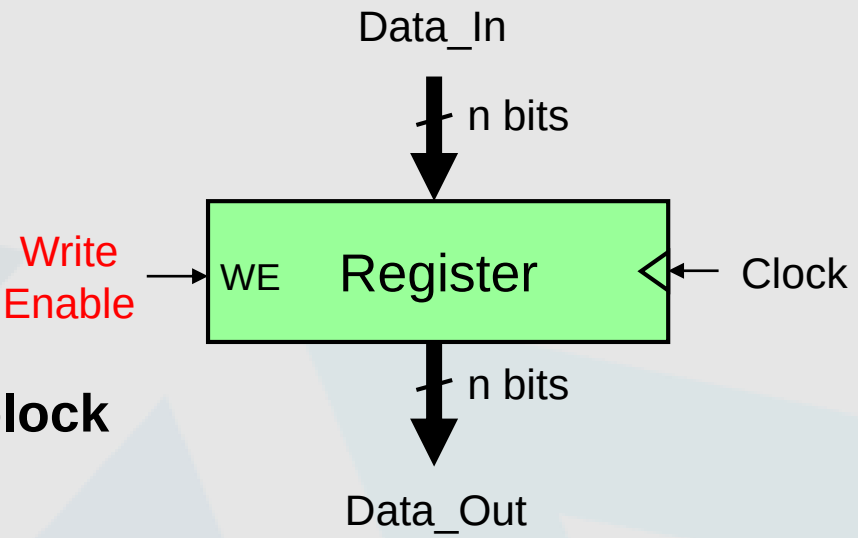
Components of the MIPS Datapath

- Combinational Elements
  - ALU, Adder
  - Immediate extender
  - Multiplexers
- Storage Elements
  - Instruction memory
  - Data memory
  - PC register
  - Register file
- Clocking methodology
  - Timing of writes (to registers/memory)



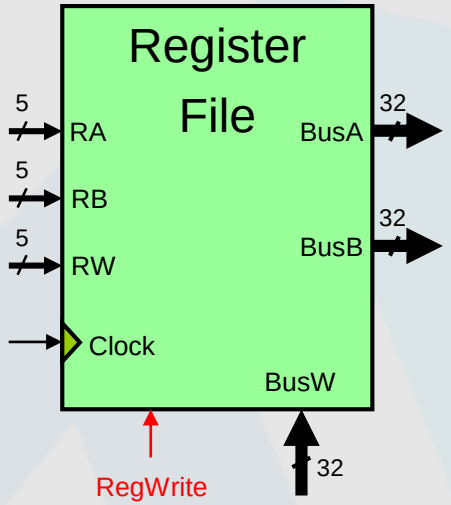
Components of the MIPS Datapath
Register Element

- Register
  - Similar to the D-type Flip-Flop
- n-bit input and output
- **During read, register behaves as a combinational logic block**
- 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 32 × 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 => BusA or BusB valid after access time



Components of the MIPS Datapath
Details of the Register File Read/Write

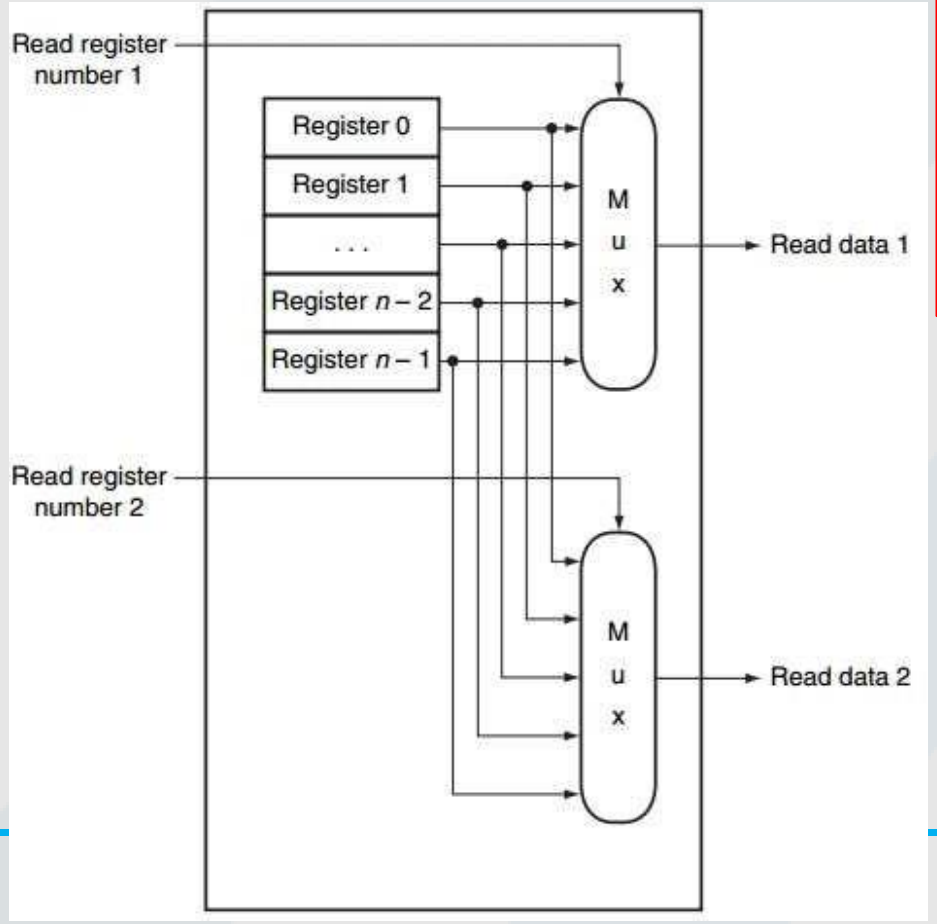
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

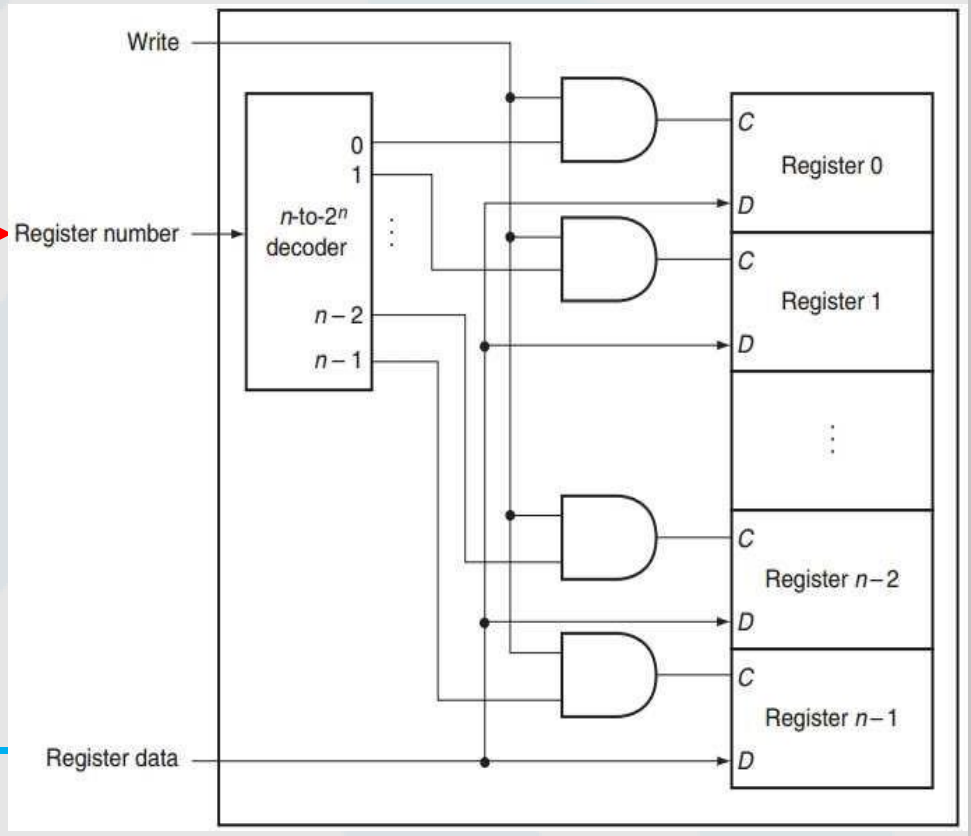
To **write** a data word to reg. file we need:

- two inputs: one with reg. # to be written and one to supply data
- Write enable

For data word to be **read** from reg. we need:

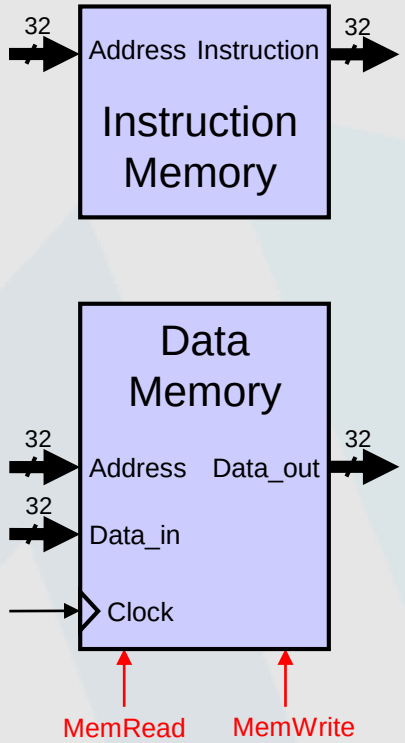
- input to reg. file with reg. # to be read
- output from reg. file carrying that reg. data





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

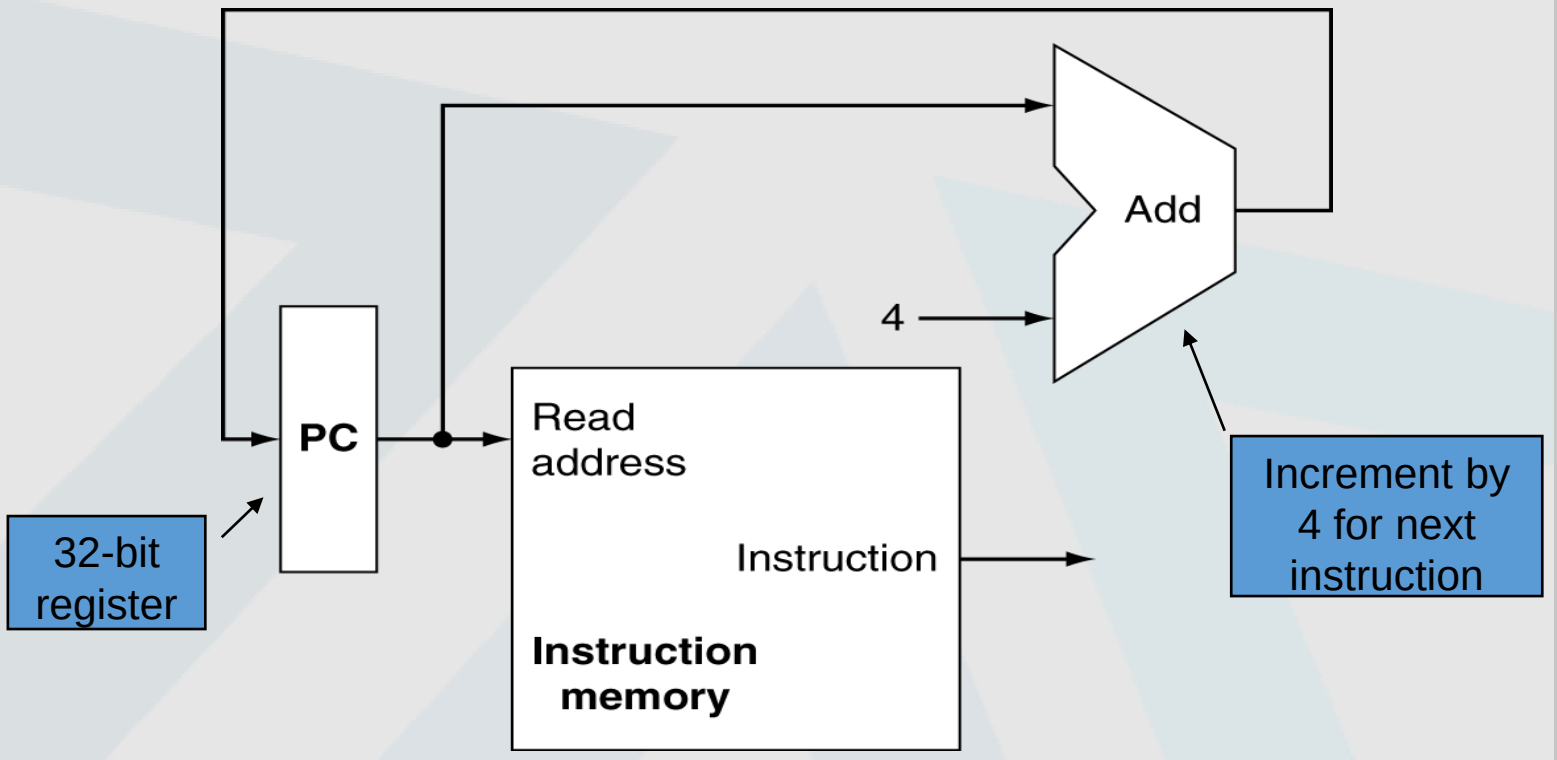


Next . . .

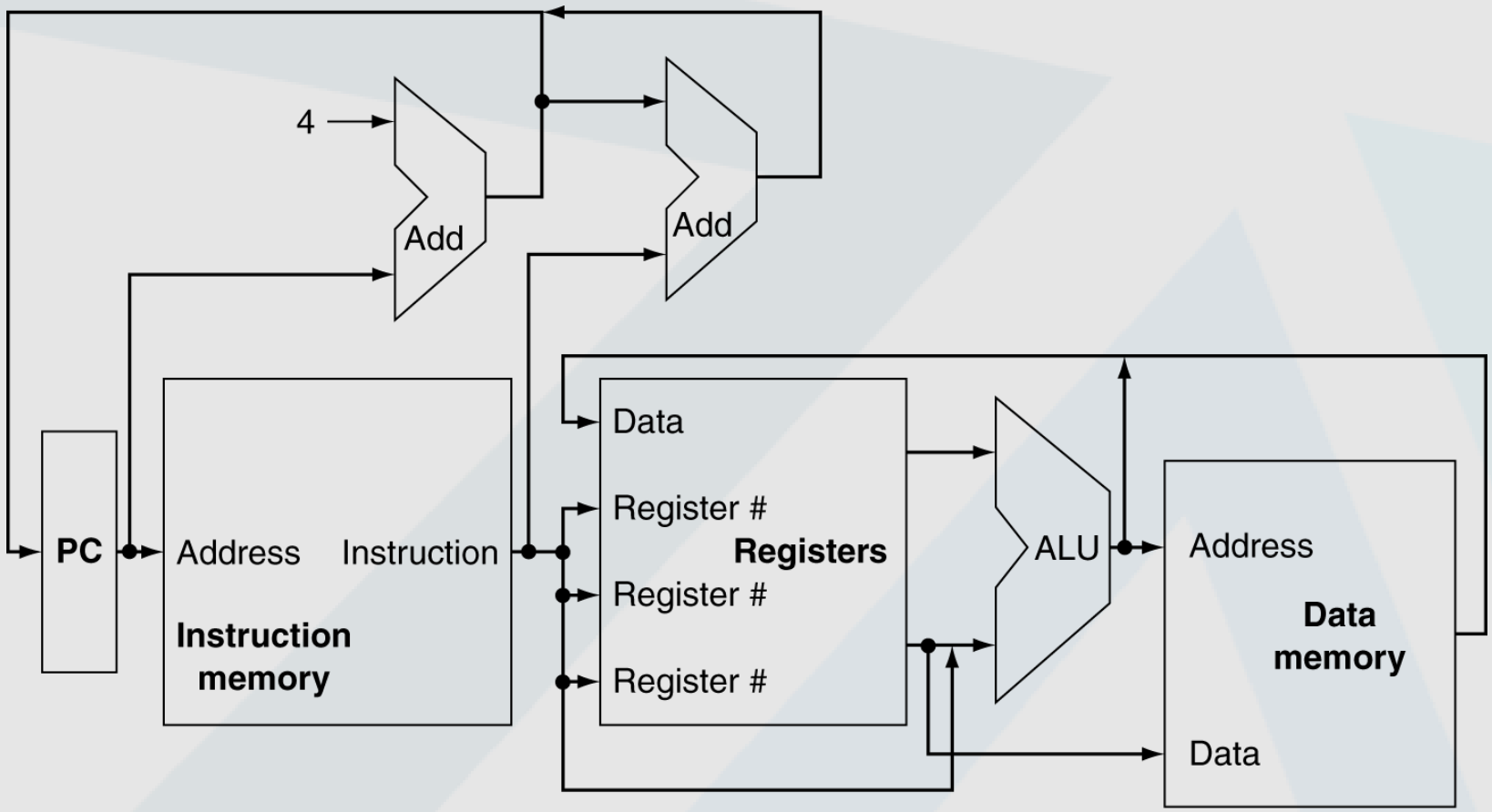
- Designing a Processor: Step-by-Step
- Datapath Components and Clocking
- Assembling an Adequate Datapath
- Controlling the Execution of Instructions
- The Main Controller and ALU Controller
- Drawback of the single-cycle processor design

Fetching Instructions

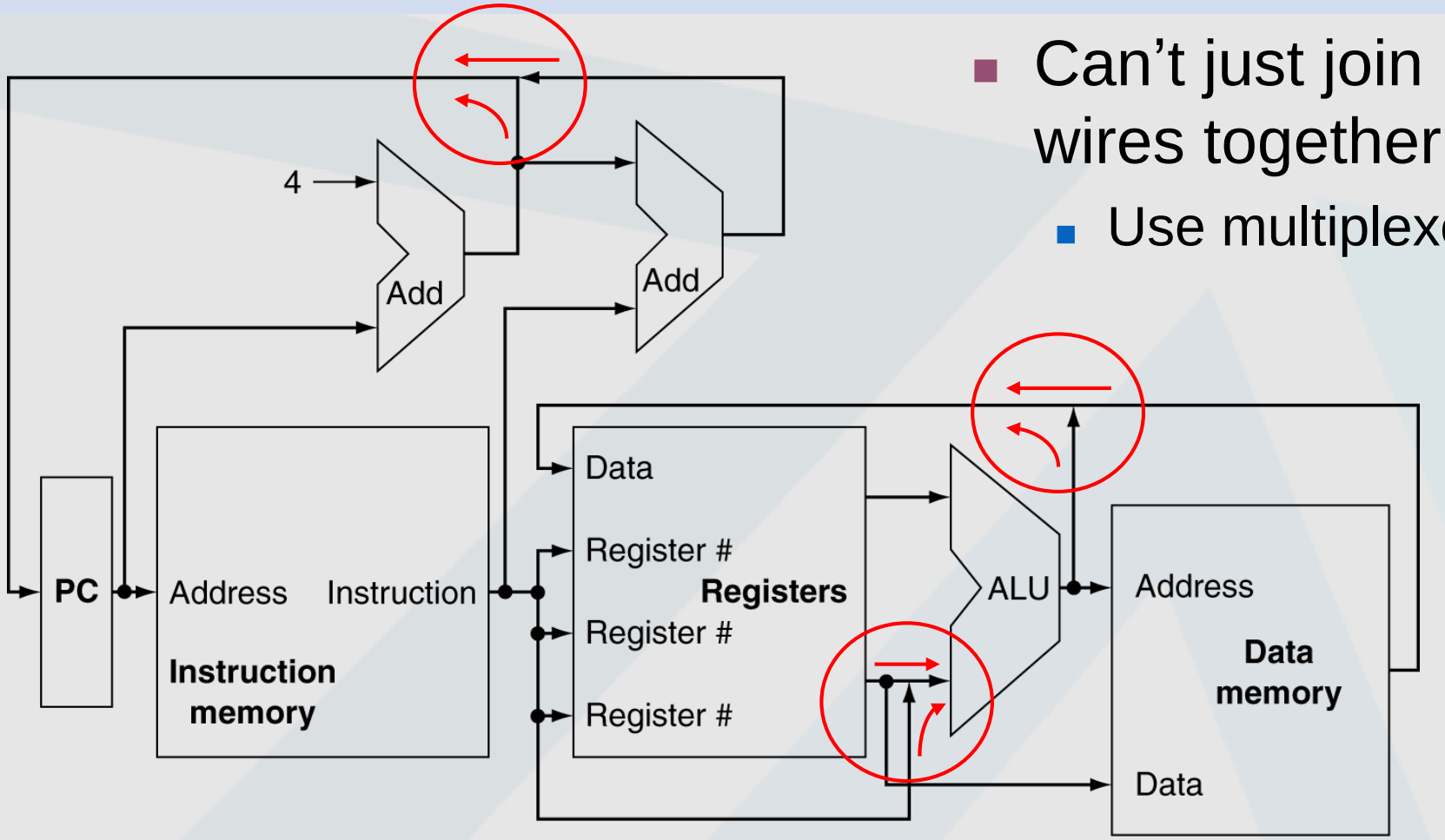
- **Fetching Instructions**
  - **Send the program counter (PC) to the memory that contains the code**
  - **Fetch the instruction from that memory**
- **Program counter PC (state element)**
  - **A register containing the address of the instruction in the program being executed**
  - **32-bit register**
  - **Written at the end of every clock cycle**
    - **Does not need a write control signal**



CPU Overview
Multiple data sources

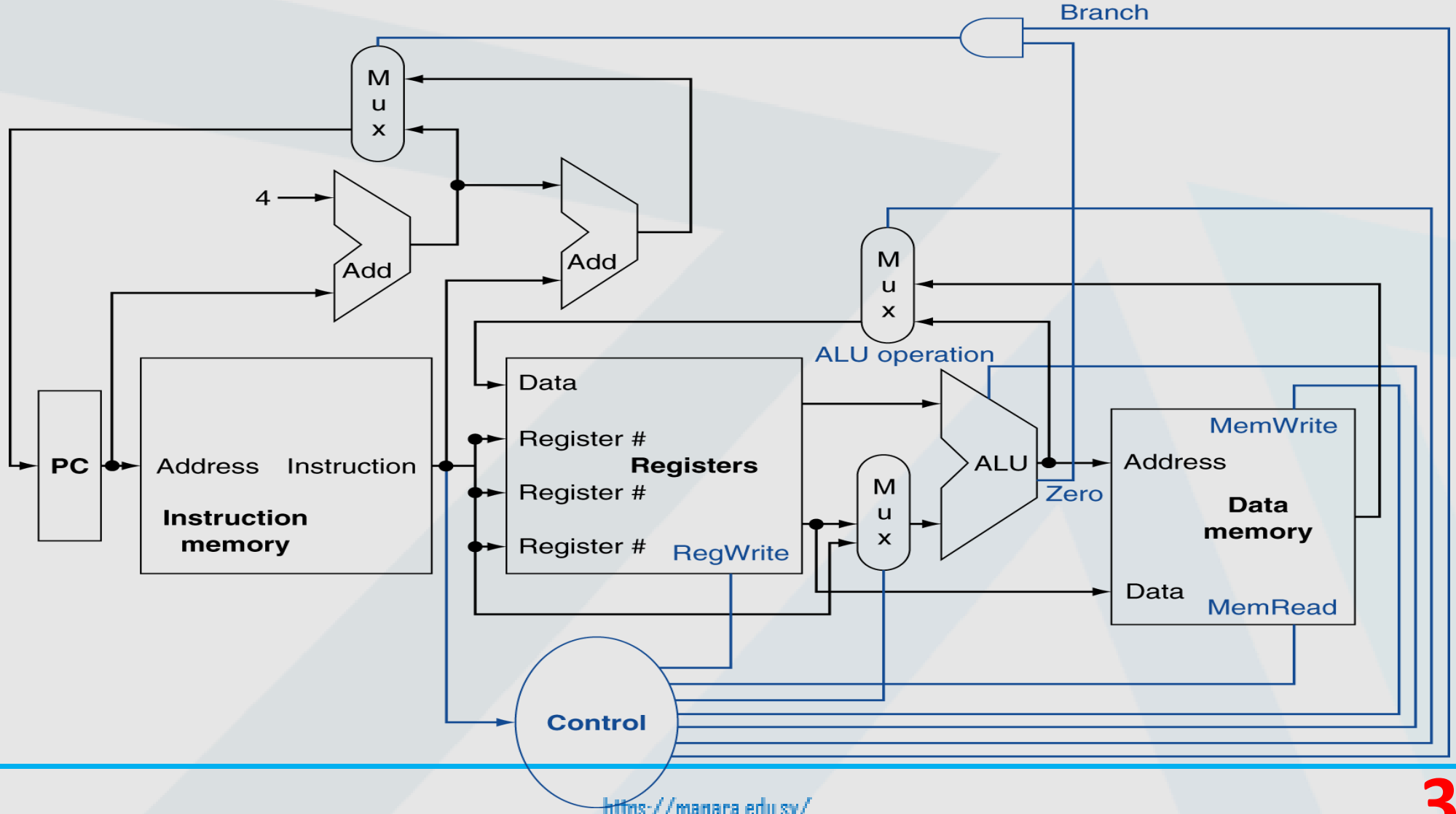


CPU Overview (Multiple data sources) Multiplexers



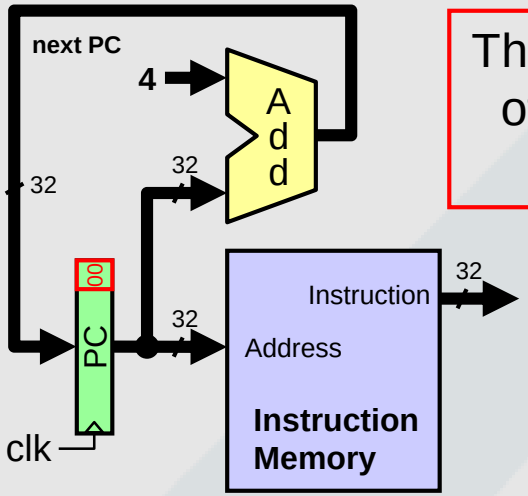
- Can't just join wires together
  - Use multiplexers

CPU Overview (Multiple data sources) Multiplexers + Control



Instruction Fetching Datapath

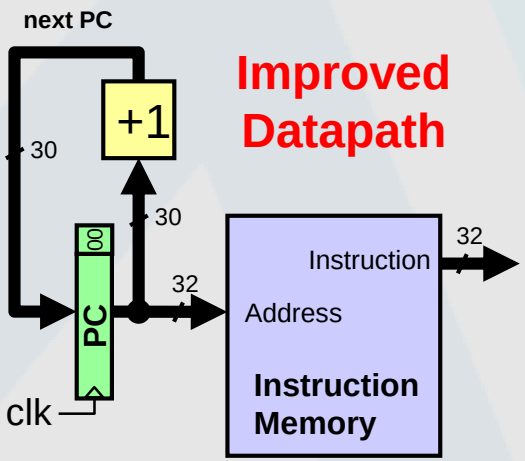
- We can now assemble the datapath from its components
- For instruction fetching, we need …
  - Program Counter (PC) register
  - Instruction Memory
  - Adder for incrementing PC



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

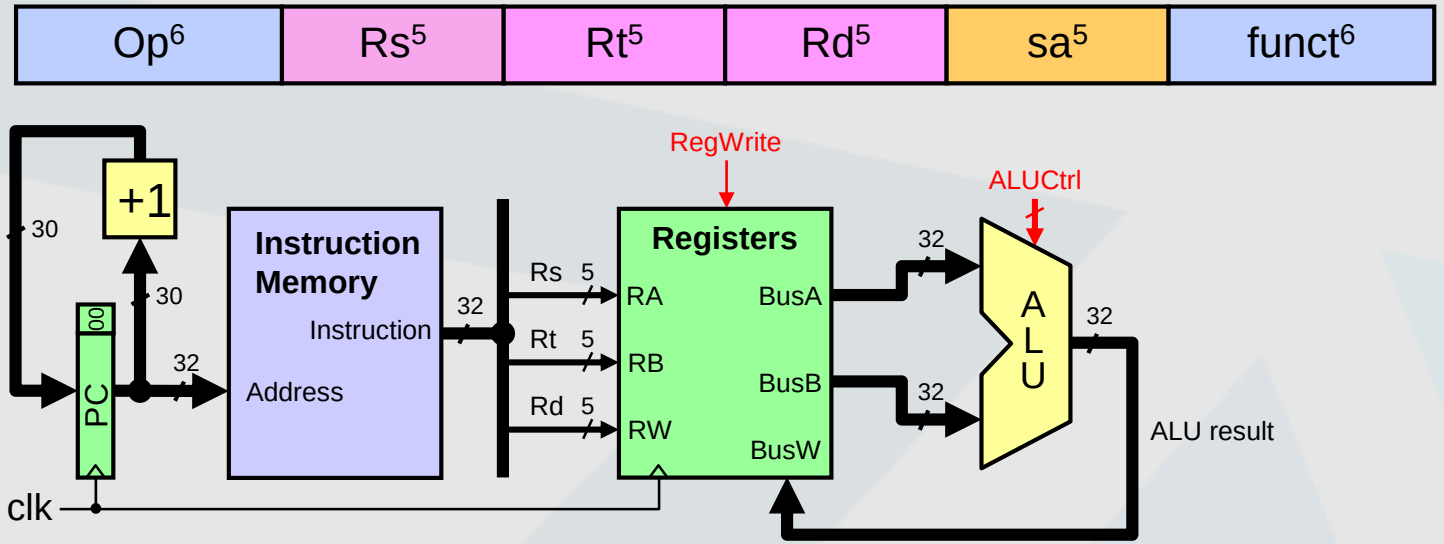
Datapath does not handle branch or jump instructions

Improved datapath increments upper 30 bits of PC by 1



Improved Datapath

Datapath for R-type Instructions



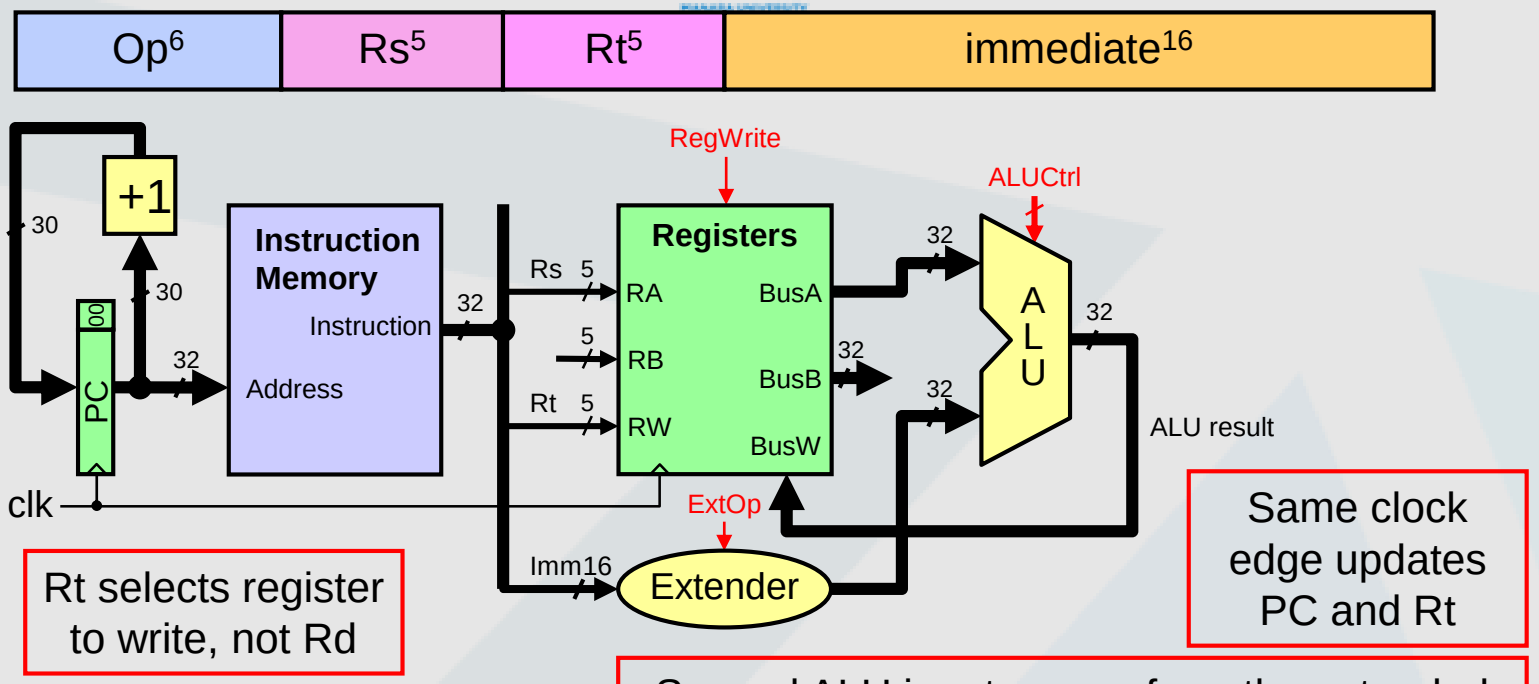
Rs and Rt fields select two registers to read. Rd field selects register to write

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

Same clock updates PC and Rd register

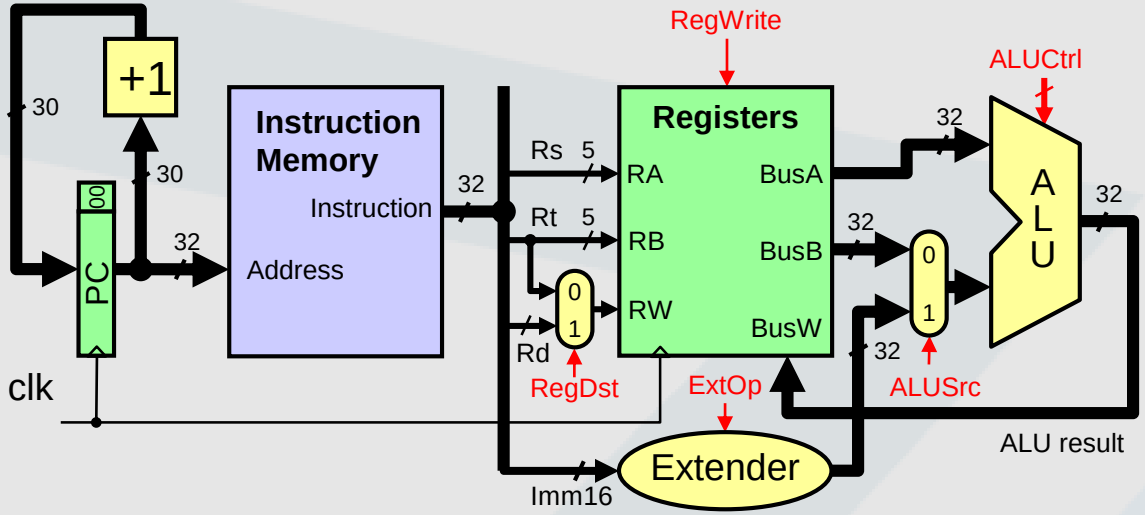
- Control signals
  - ALUCtrl is derived from the funct field because Op = 0 for R-type
  - RegWrite is used to enable the writing of the ALU result

Datapath for I-type ALU Instructions



- Control signals
  - ALUCtrl 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

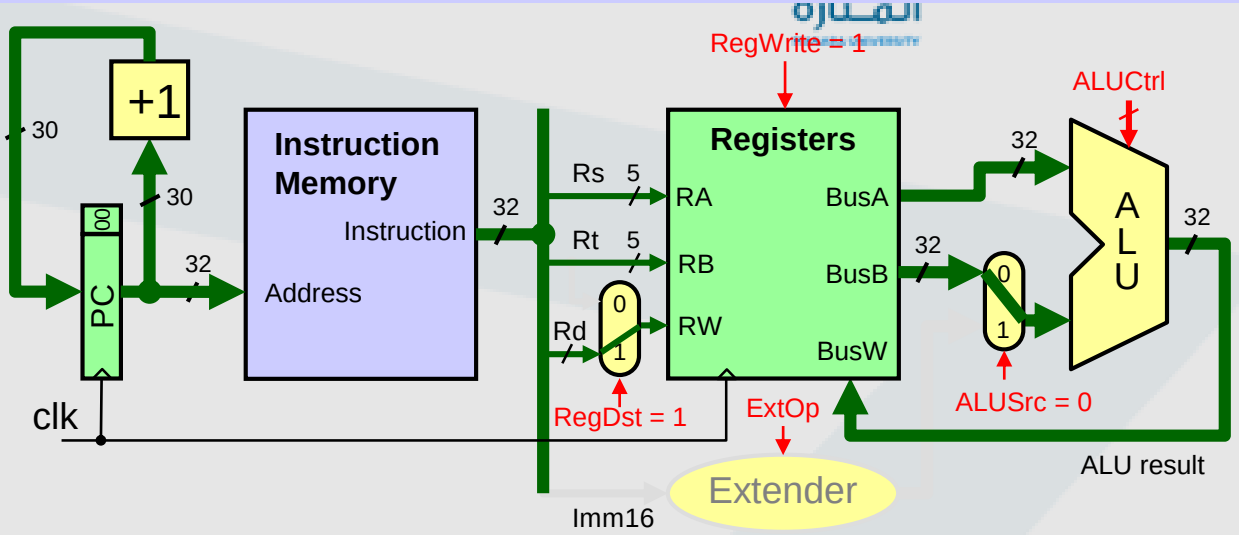


A mux selects RW as either Rt or Rd

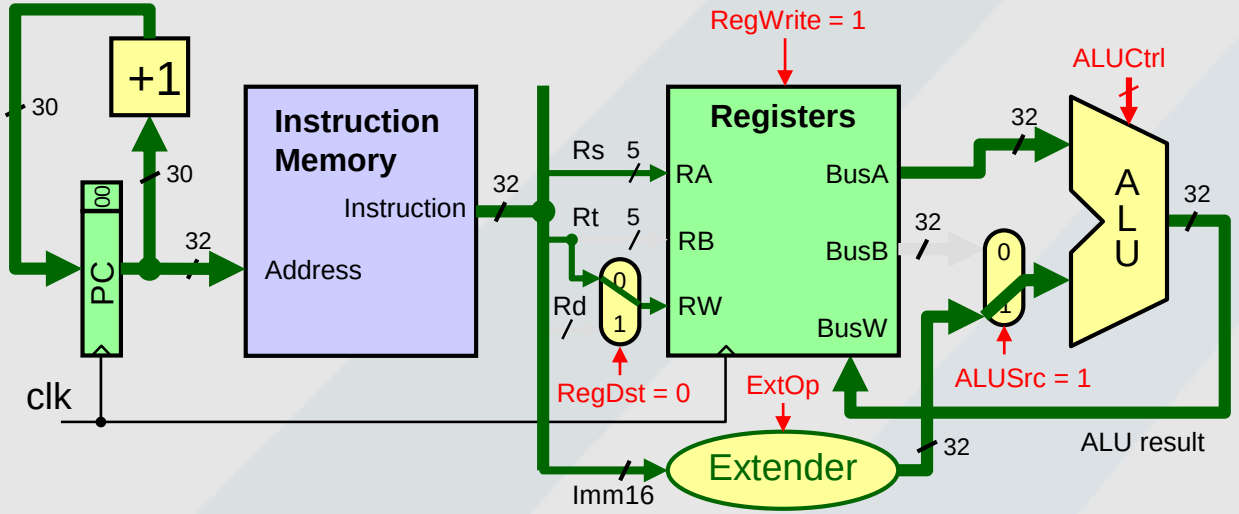
Another mux selects 2nd ALU input as either data on BusB or the extended immediate

- 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 2nd 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**

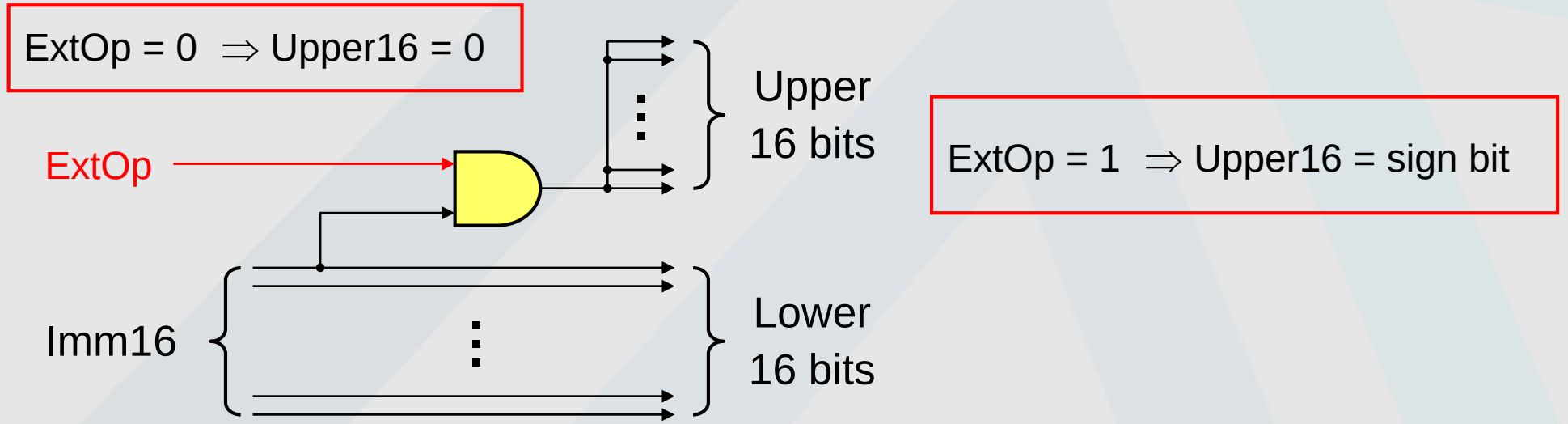


For I-type ALU instructions, RegDst is '0' to select Rt on RW and ALUSrc is '1' to select Extended immediate as second ALU input. The active part of datapath is shown in **green**

Load/Store instructions are not discussed here.

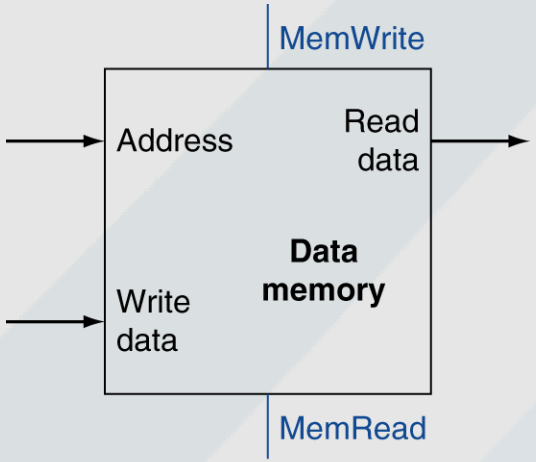
Details of the Extender

- 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

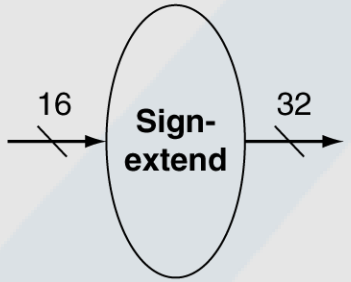


Load/Store Instructions

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



a. Data memory unit



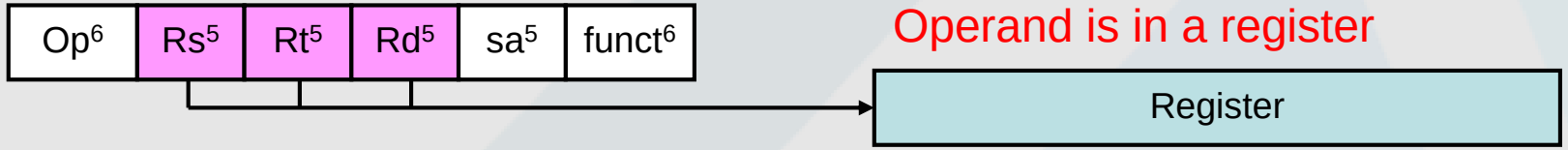
b. Sign extension unit

Addressing Modes

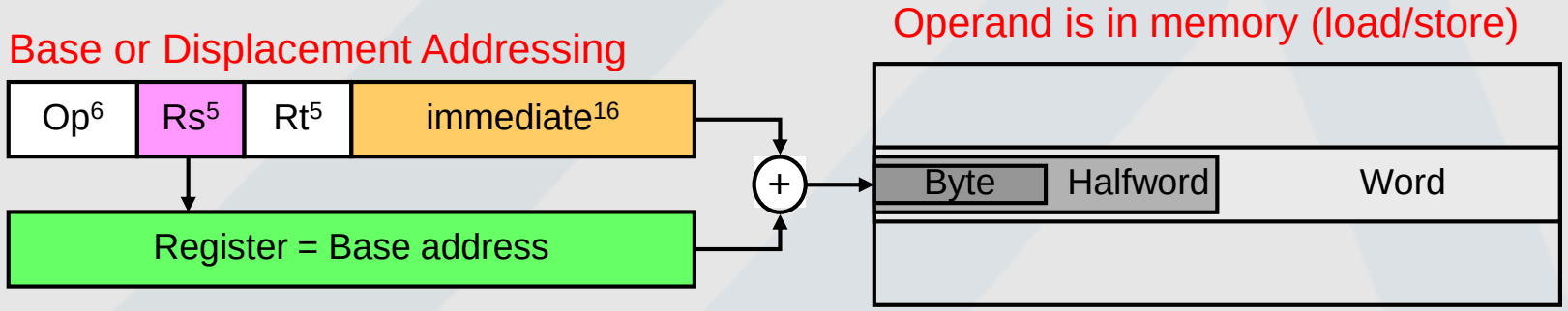
- Where are the operands?
- How memory addresses are computed?

Immediate Addressing

Register Addressing



Base or Displacement Addressing



Building a Datapath: `lw` and `sw`

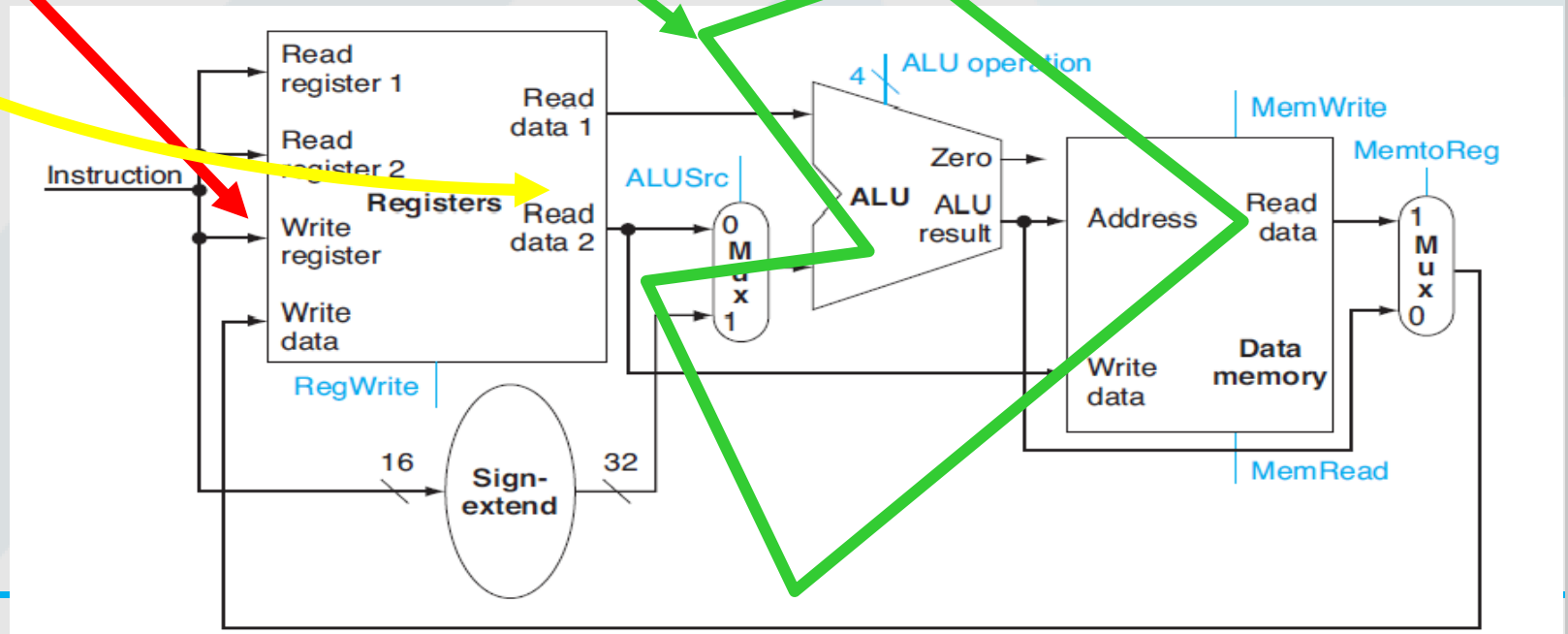
- Recall `lw` and `sw`
  - `lw  $t1, offset_value($t2)`
  - `sw  $t1, offset_value($t2)`

Load: writes to `$t1` (Need data memory)
Store: reads from `$t1`

Both instructions compute memory address
`offset_value + $t2`
`offset_value`: 16-bit signed value

(Need: Register file, ALU and Sign Extend Unit)

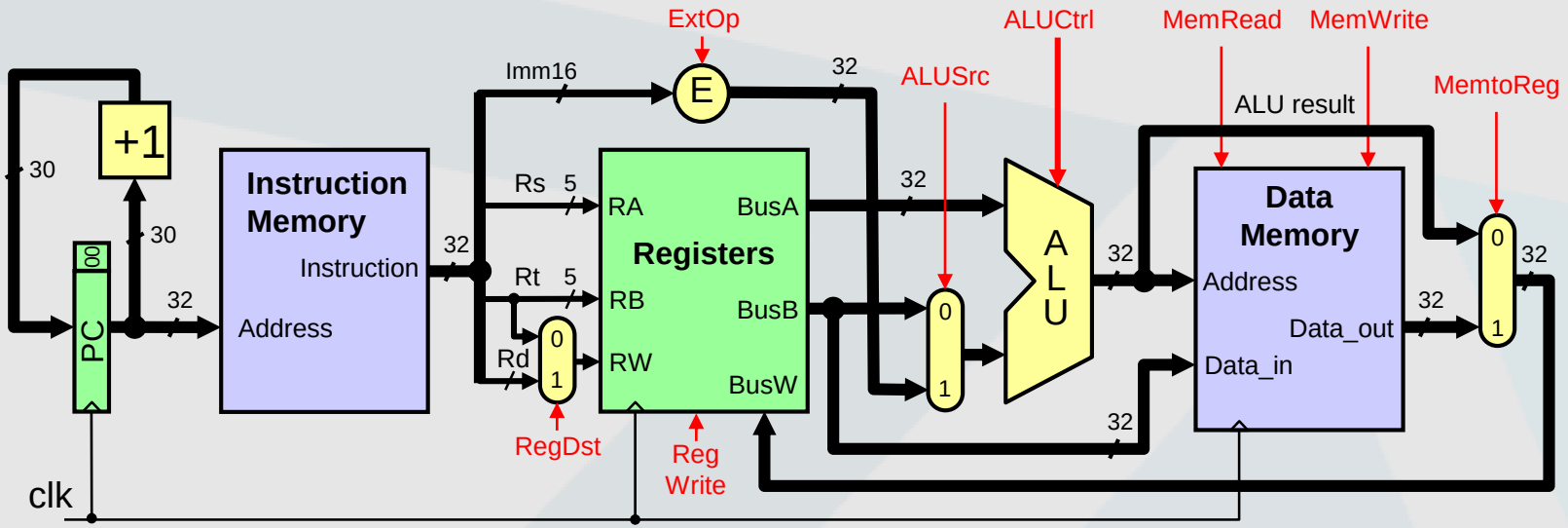
- The Multiplexor on the second ALU Input
  - Used to determine the source of the second ALU input
    - Either from the registers
      - for an arithmetic-logical instruction OR a branch
    - Or from the offset field of the instruction
      - for a load or store



The datapath for the memory instructions and the R-type instructions.

Adding Data Memory to Datapath

- A data memory is added for load and store instructions



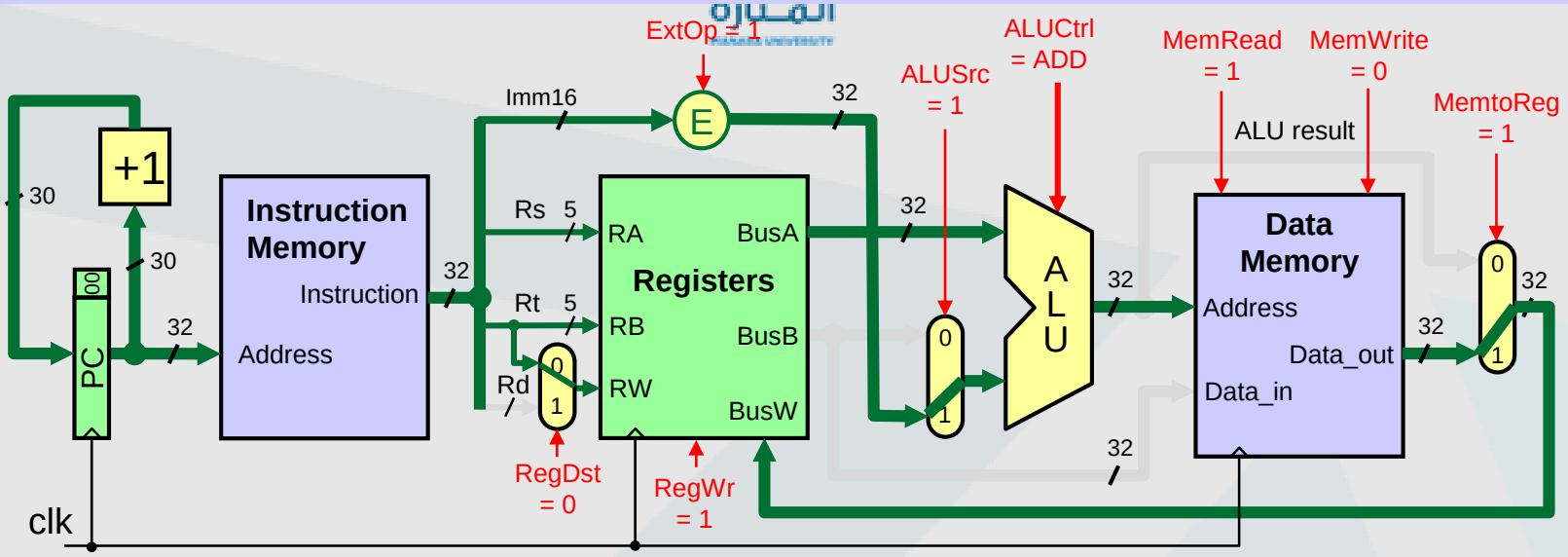
ALU calculates data memory address

A 3rd 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



RegDst = ‘0’ selects Rt as destination register

RegWrite = ‘1’ to enable writing of register file

ExtOp = 1 to sign-extend Immmediate16 to 32 bits

ALUSrc = ‘1’ selects extended immediate as second ALU input

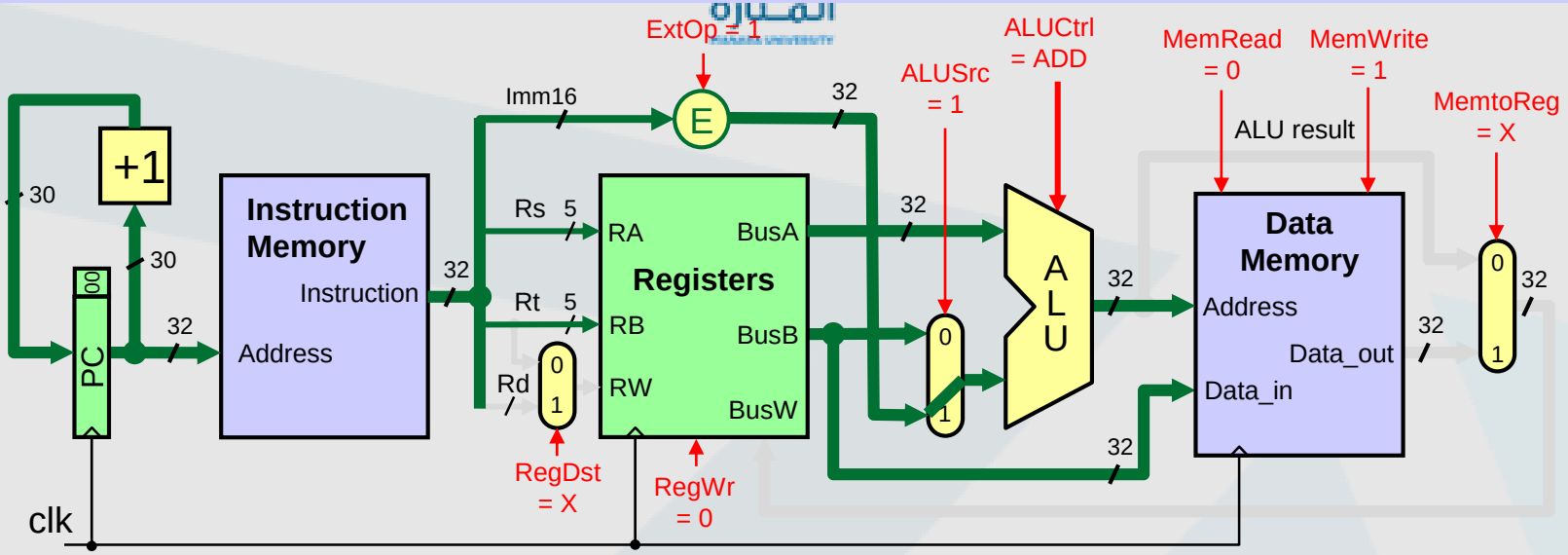
ALUCtrl = ‘ADD’ to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemRead = ‘1’ to read data memory

MemtoReg = ‘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

RegWrite = ‘0’ to disable writing of register file

ExtOp = 1 to sign-extend Immmediate16 to 32 bits

ALUSrc = ‘1’ selects extended immediate as second ALU input

ALUCtrl = ‘ADD’ to calculate data memory address as Reg(Rs) + sign-extend(Imm16)

MemWrite = ‘1’ to write data memory

MemtoReg = ‘X’ because don’t care what data is put on BusW

Clock edge updates PC and Data Memory

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement for [offset x 4])
  - Add to PC + 4
    - Already calculated by instruction fetch

Building a Datapath: beq

- `beq  $t1, $t2, offset`

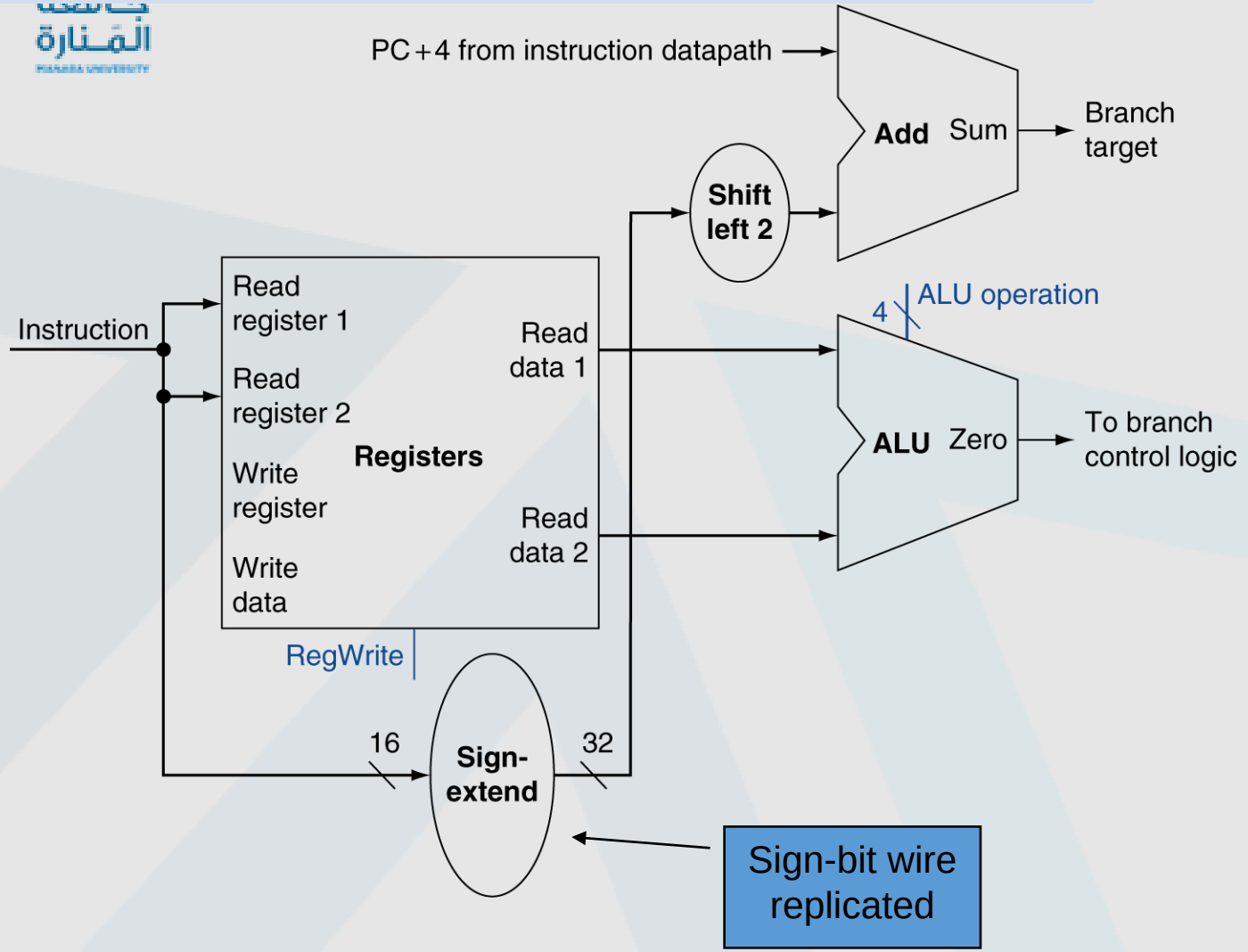
1) Branch Target Address Calculation:

`(PC + 4)+(offset×4)`

2) Compares between contents of $t1 and $t2

Need:

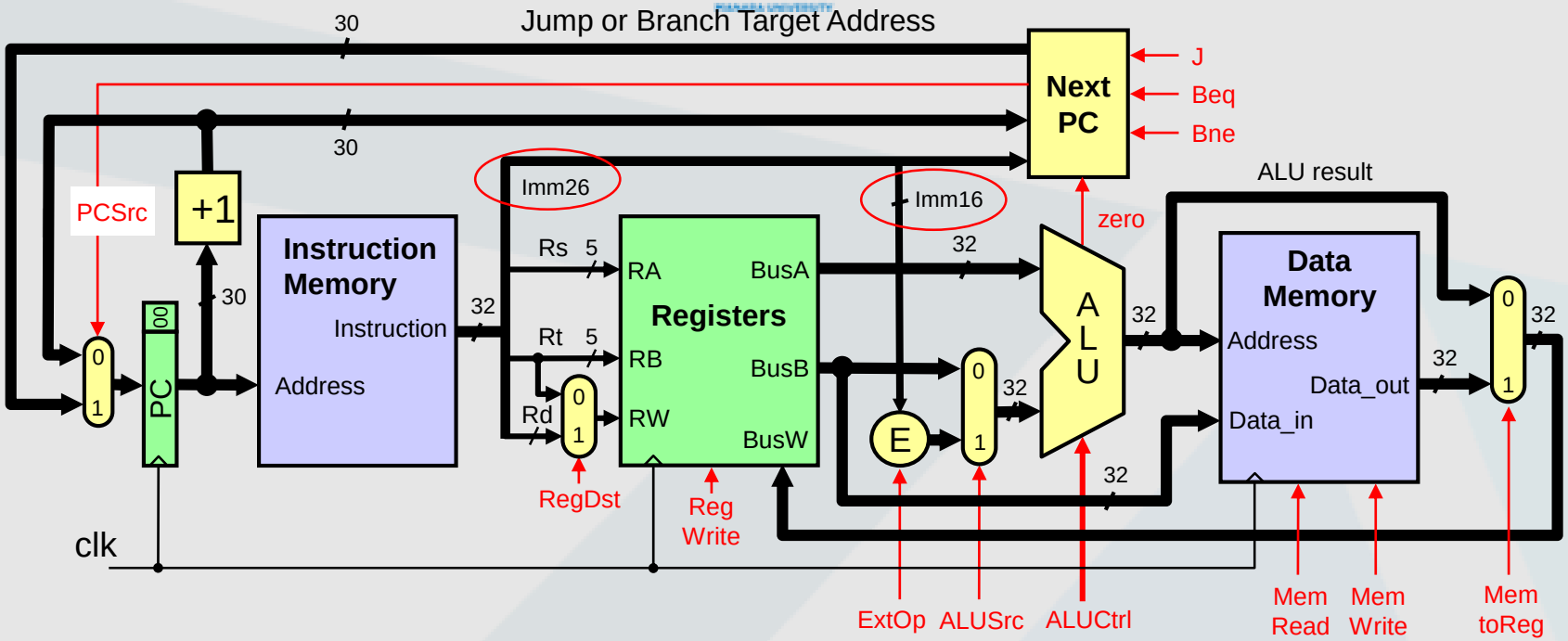
- Sign Extend
- Adder
- PC
- Register file
- ALU



$If\ \$t1 = \$t2, we\ say: Branch\ taken$

$If\ \$t1 \neq \$t2, we\ say: Branch\ not\ taken$

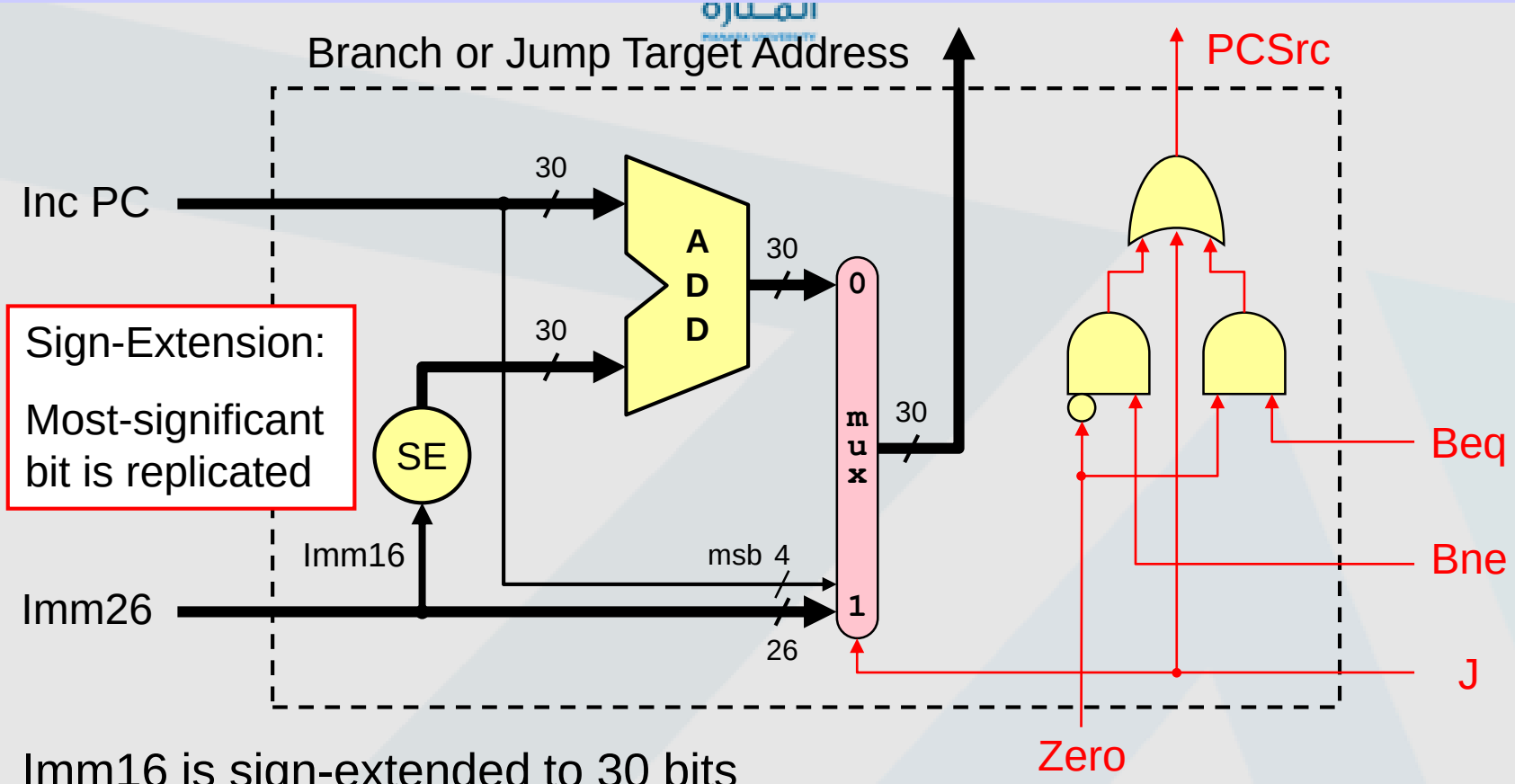
Adding Jump and Branch to Datapath



- Additional Control Signals
  - J, Beq, Bne for jump and branch instructions
  - Zero flag of the ALU is examined
  - PCSrc = 1 for jump & taken branch

Next PC logic computes jump or branch target instruction address

Details of Next PC

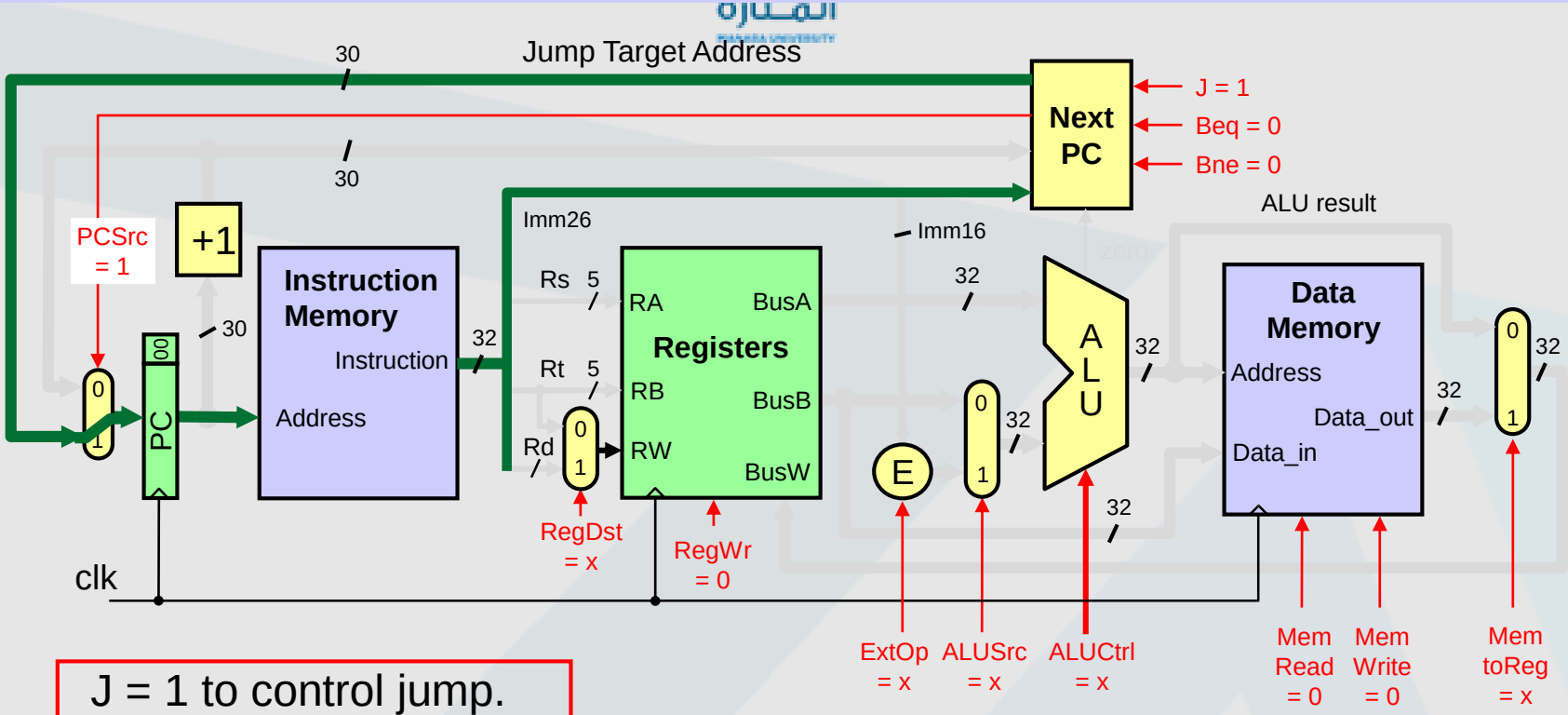


Imm16 is sign-extended to 30 bits

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

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

Controlling the Execution of Jump



J = 1 to control jump.
Next PC outputs Jump Target Address

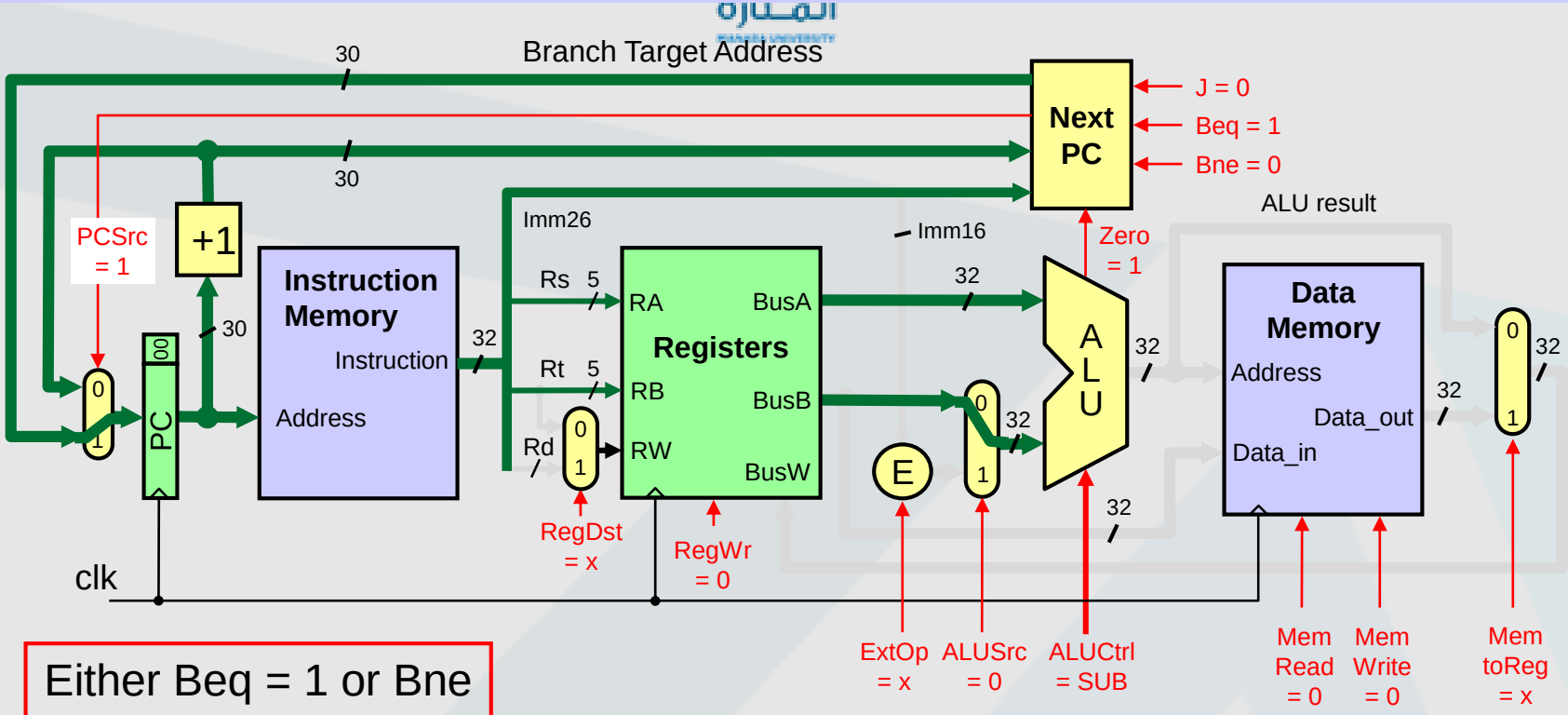
PCSrc=1

MemRead, MemWrite, and RegWrite are 0

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

Clock edge updates PC register only

Controlling the Execution of Branch



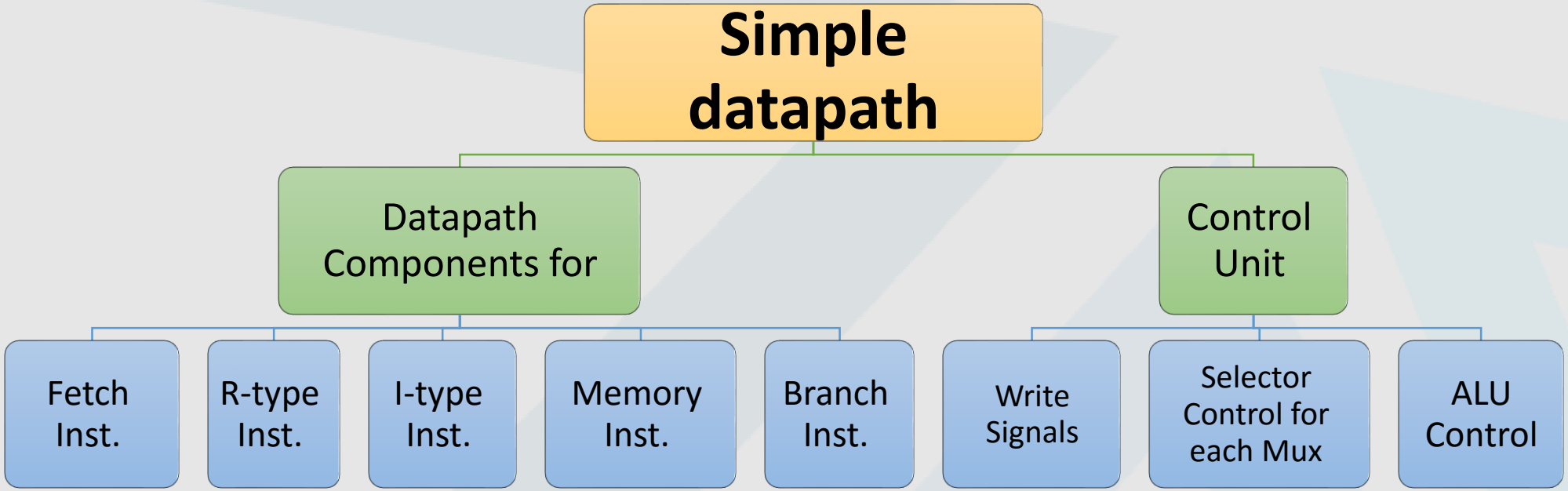
Either Beq = 1 or Bne depending on opcode
ALUSrc = 0 to select value on BusB
ALUCtrl = SUB to generate Zero Flag

Next PC outputs branch target address PCSrc = 1 if branch is taken
RegWrite, MemRead, and MemWrite are 0
Clock edge updates PC register only

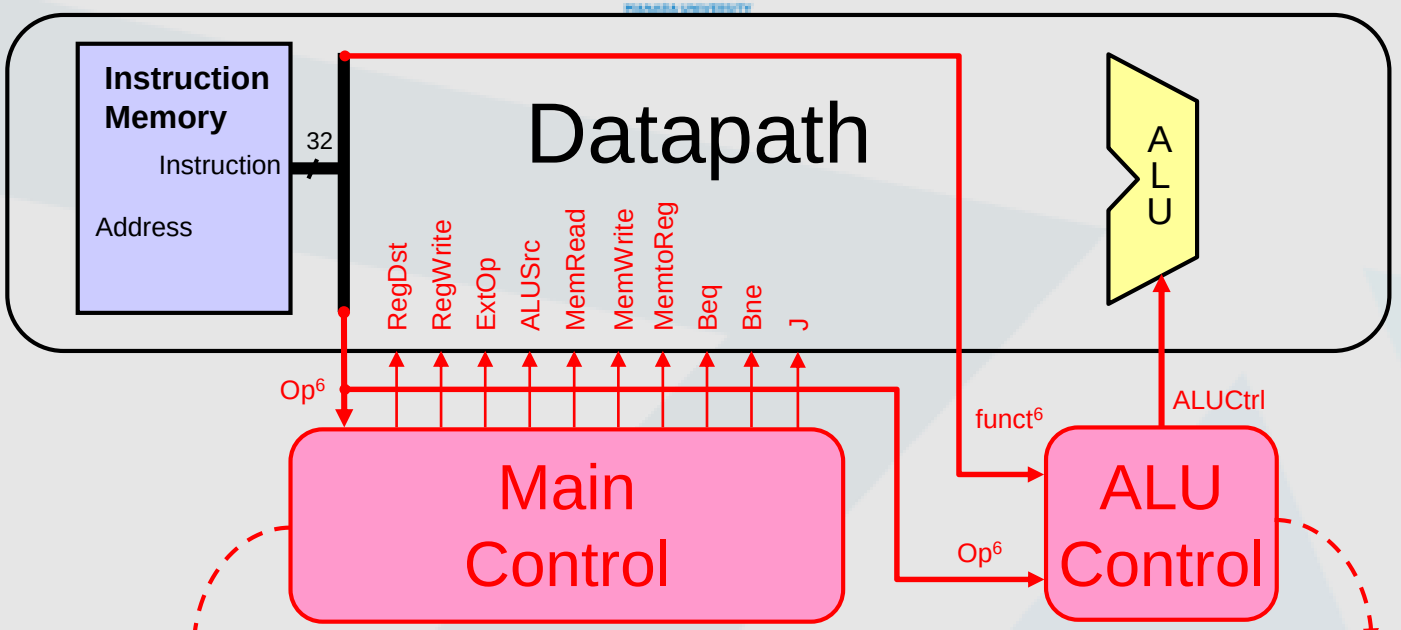
Next . . .

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

Creating a simple datapath



Main Control and ALU Control



Main Control Input:

- 6-bit opcode field from instruction

Main Control Output:

- 10 control signals for the Datapath

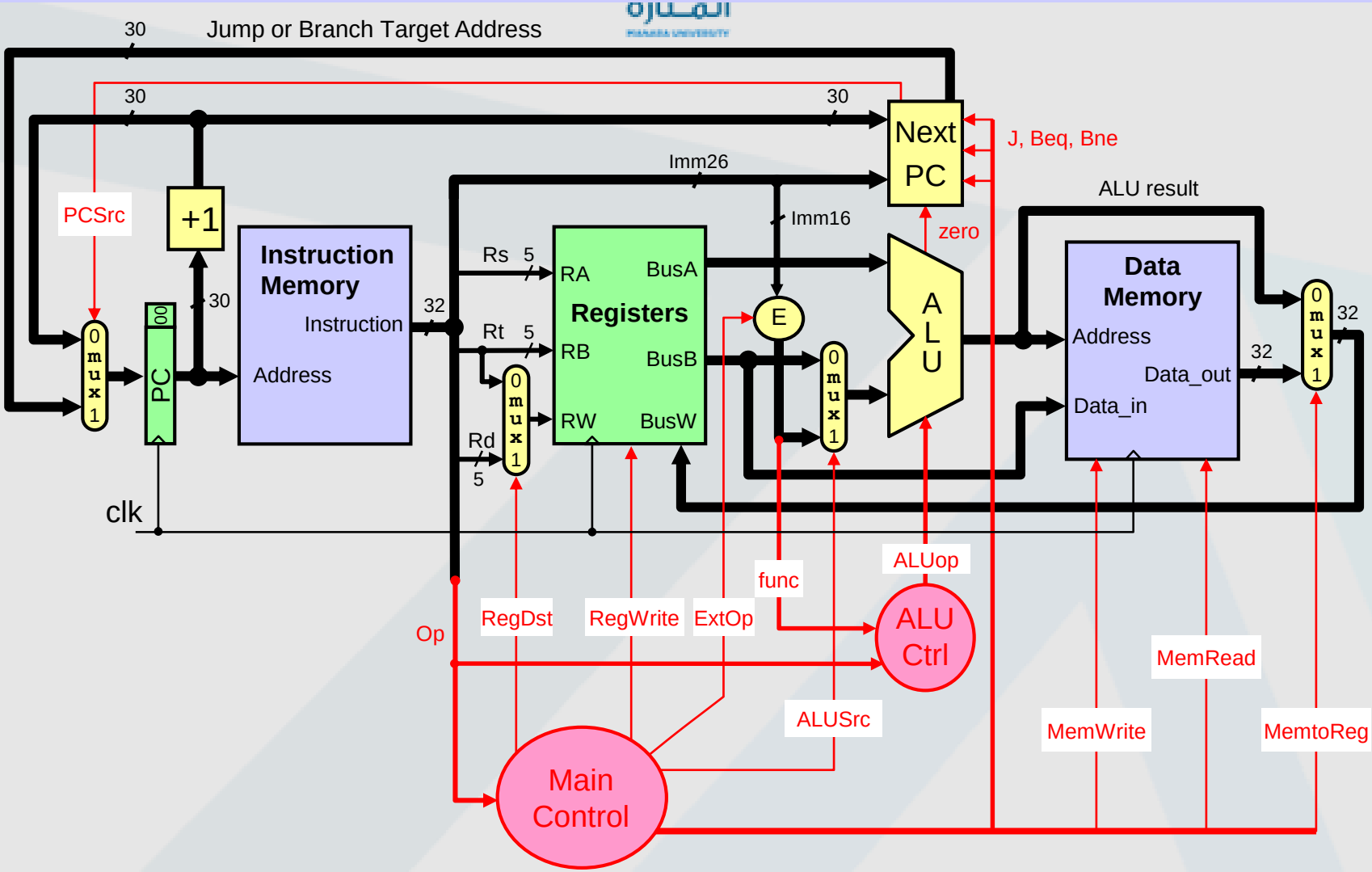
ALU Control Input:

- 6-bit opcode field from instruction
- 6-bit function field from instruction

ALU Control Output:

- ALUCtrl signal for ALU

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 Data_out ← Memory[address]
MemWrite	None	Data memory is written Memory[address] ← Data_in
MemtoReg	BusW = ALU result	BusW = Data_out from Memory
Beq, Bne	PC ← PC + 4	PC ← Branch target address If branch is taken
J	PC ← PC + 4	PC ← Jump target address

Main Control Signal Values

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

RegWrite = $\overline{(sw + beq + bne + j)}$

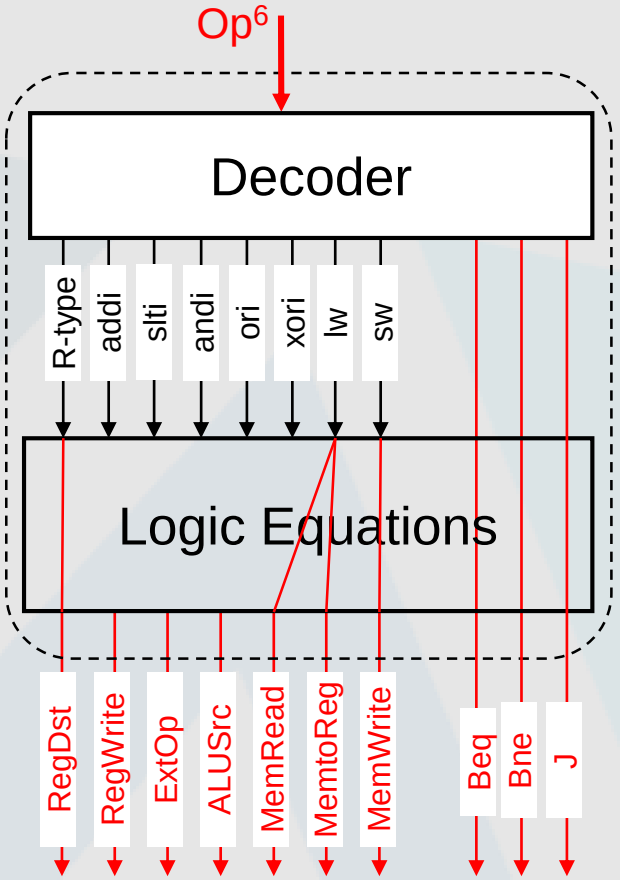
ExtOp = $\overline{(andi + ori + xori)}$

ALUSrc = $\overline{(R\text{-}type + beq + bne)}$

MemRead = lw

MemtoReg = lw

MemWrite = sw



ALU Control Truth Table

Input		Output	4-bit
Op^6	$funct^6$	ALUCtrl	Encoding
R-type	add	ADD	0000
R-type	sub	SUB	0010
R-type	and	AND	0100
R-type	or	OR	0101
R-type	xor	XOR	0110
R-type	slt	SLT	1010
addi	x	ADD	0000
slti	x	SLT	1010
andi	x	AND	0100
ori	x	OR	0101
xori	x	XOR	0110
lw	x	ADD	0000
sw	x	ADD	0000
beq	x	SUB	0010
bne	x	SUB	0010
j	x	x	x

The 4-bit ALUCtrl is encoded according to the ALU implementation

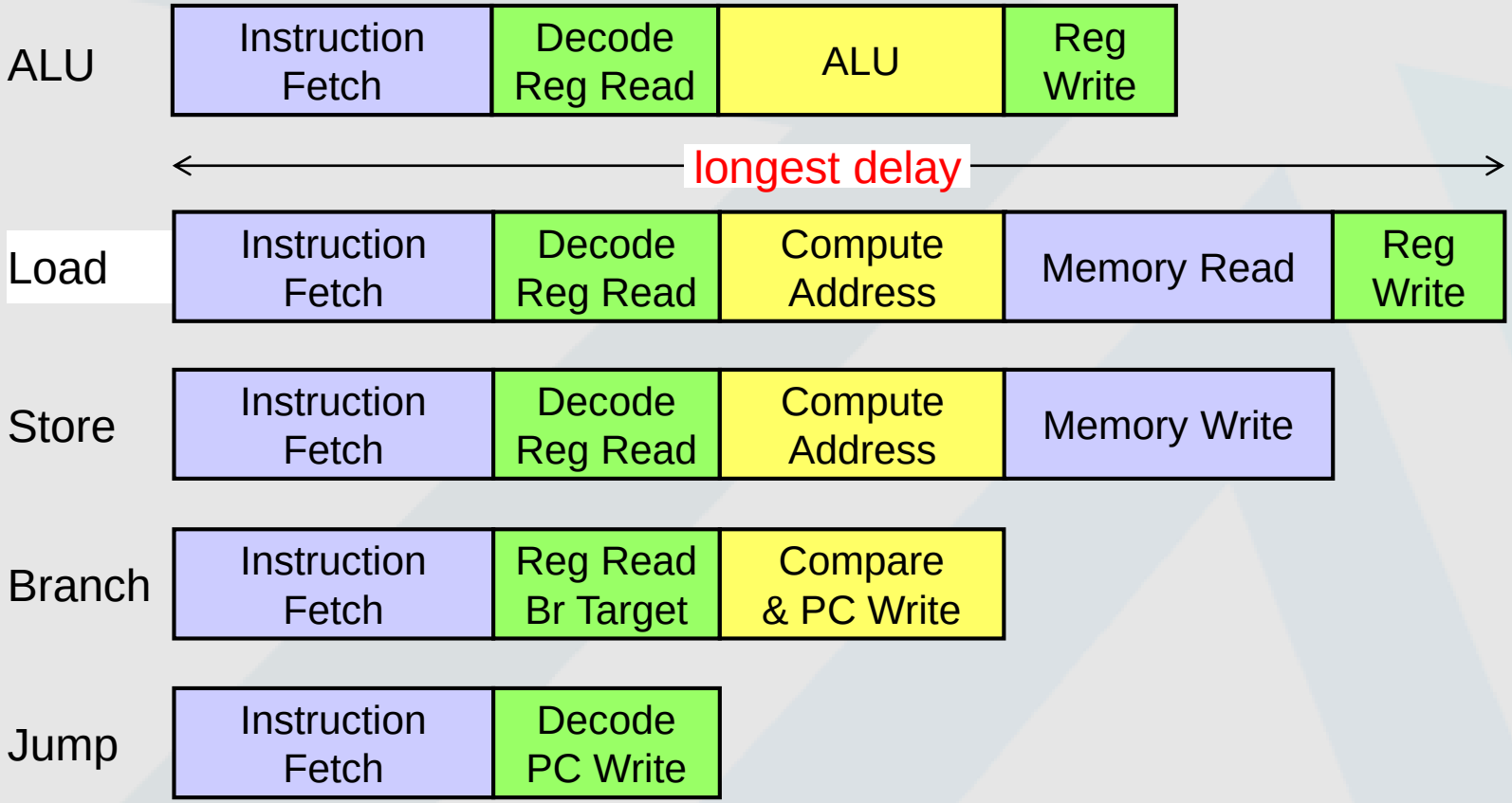
Other ALU control encodings are also possible. The idea is to choose a binary encoding that will simplify the logic

Next . . .

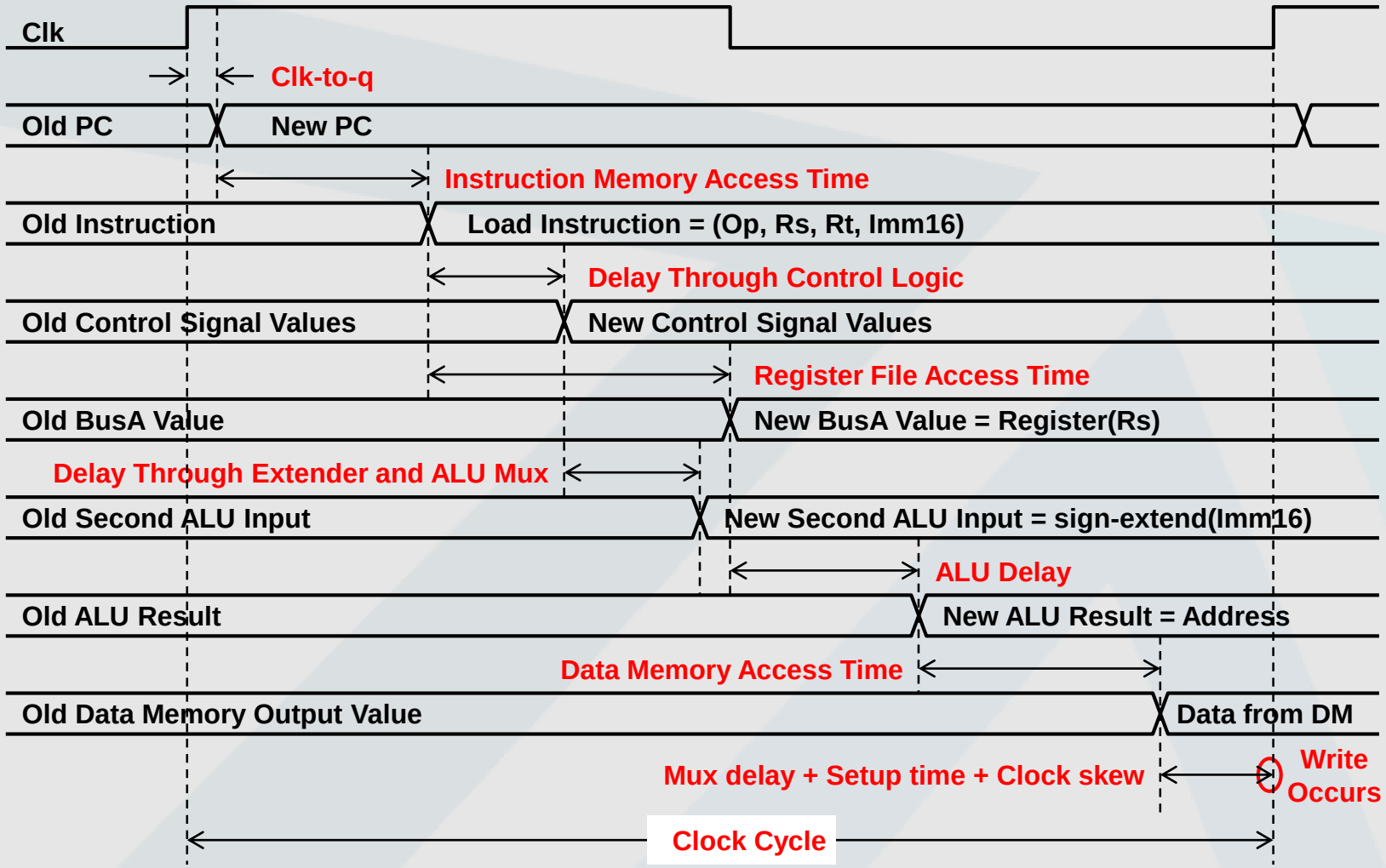
- Designing a Processor: Step-by-Step
- Datapath Components and Clocking
- Assembling an Adequate Datapath
- Controlling the Execution of Instructions
- The Main Controller and ALU Controller
- Drawback of the single-cycle processor design

Drawbacks of Single Cycle Processor

- Long cycle time
  - All instructions take as much time as the slowest instruction



Timing of a Load Instruction



Worst Case Timing – Cont'd

- Long cycle time: long enough for Slowest instruction

  PC Clk-to-Q delay

  + Instruction Memory Access Time

  + Maximum of (

    Register File 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

Alternative: Multicycle Implementation

- Break instruction execution into five steps
  - Instruction fetch
  - Instruction decode, register read, target address for jump/branch
  - Execution, memory address calculation, or branch outcome
  - Memory access or ALU instruction completion
  - Load instruction completion
- One clock cycle per step (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 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
- Assume the following instruction mix:
  - 40% ALU, 20% Loads, 10% stores, 20% branches, & 10% jumps

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 ← Compare and write PC			530 ps
Jump	200	150 ← Decode and write 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 = 0.4×4 + 0.2×5 + 0.1×4+ 0.2×3 + 0.1×2 = 3.8
- Speedup = 880 ps / (3.8 × 200 ps) = 880 / 760 = 1.16

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

References

- David A. Patterson, John L. Hennessy, “Computer Organization and Design MIPS Edition: The Hardware/Software Interface”, Morgan Kaufmann, 2020.
- “Single Cycle Processor Design” slides, COE 308 – KFUPM – Prof. Muhamed Mudawar.