National Tsing Hua University

COMPUTER ARCHITECTURE

Designing a Single-Cycle Processor

國立清華大學資訊工程學系
黃婷婷教授

Outline

- Introduction to designing a processor
- Analyzing the instruction set（step 1）
- Building the datapath（steps 2 and 3）
- A single-cycle implementation
- Control for the single-cycle CPU（steps 4 and 5）
  - Control of CPU operations
  - ALU controller
  - Main controller
- Adding *jump* instruction

Outline

- Introduction to designing a processor
- Analyzing the instruction set（step 1）
- Building the datapath（steps 2 and 3）
- A single-cycle implementation
- Control for the single-cycle CPU（steps 4 and 5）
  - Control of CPU operations
  - ALU controller
  - Main controller
- Adding *jump* instruction

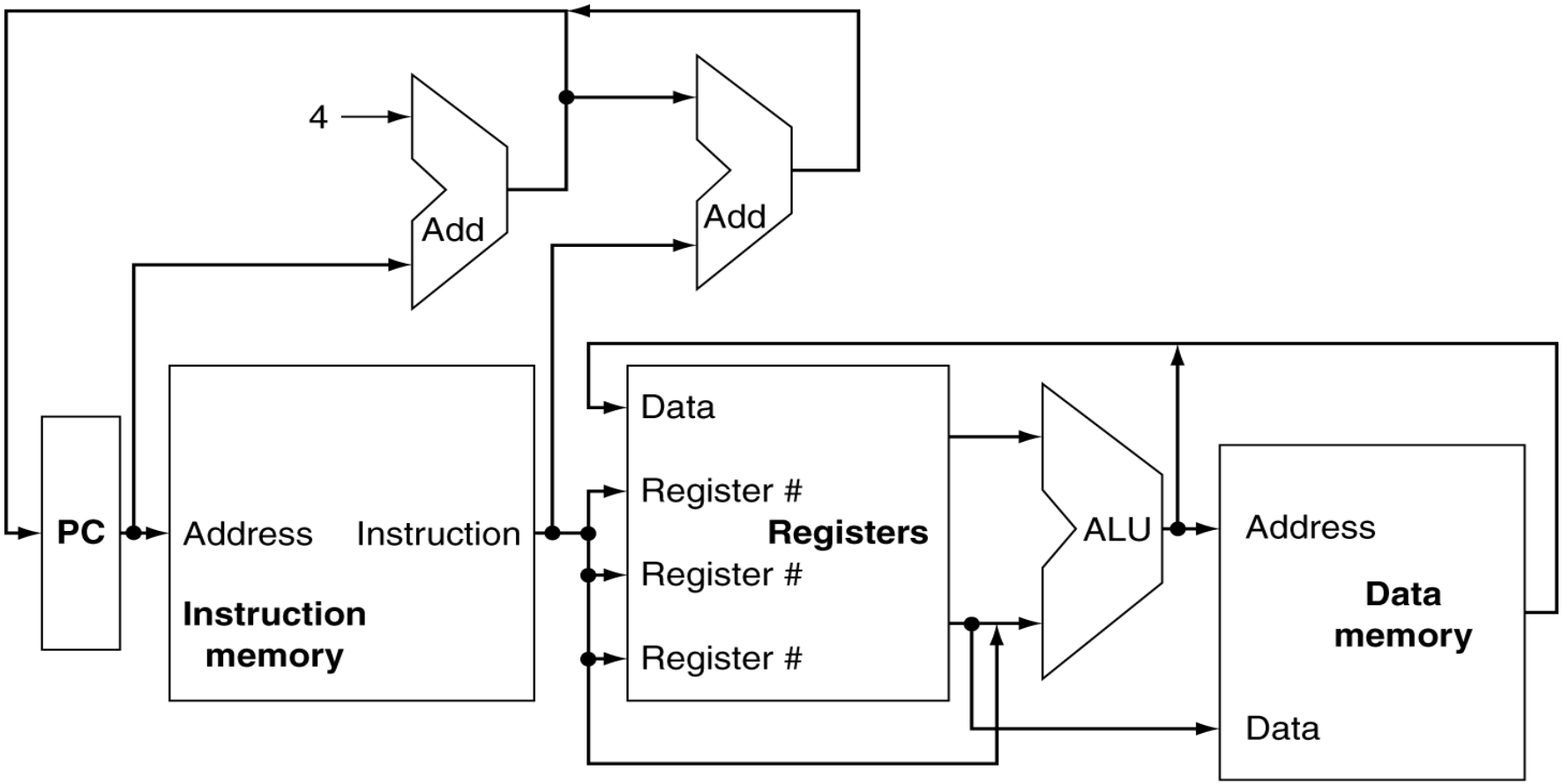
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: add, sub, and, `or`, `slt`
  - Control transfer: beq, `j`

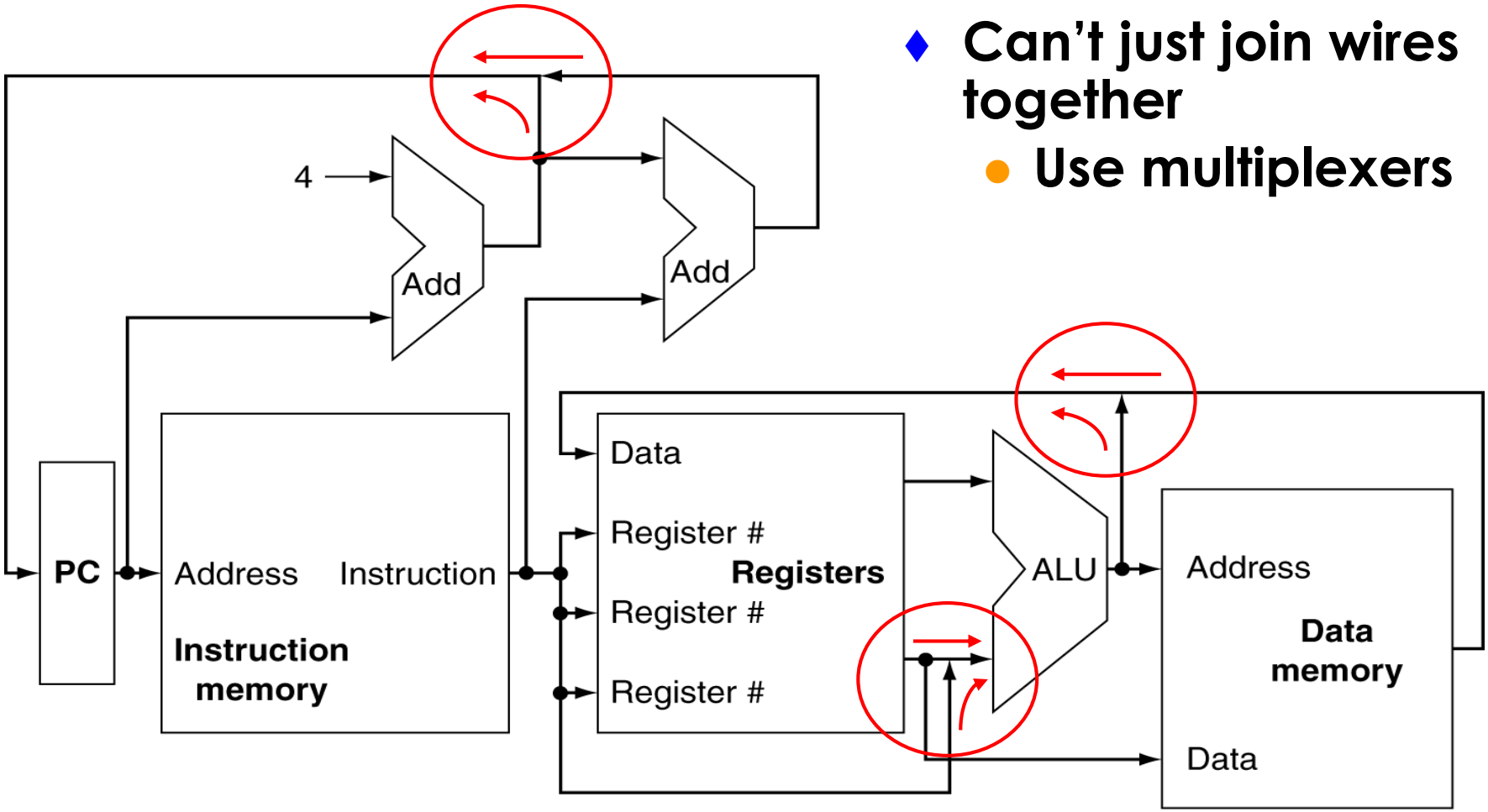
Instruction Execution

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch equality, target address
  - Access data memory for load/store
  - PC ← target address or PC + 4

CPU Overview

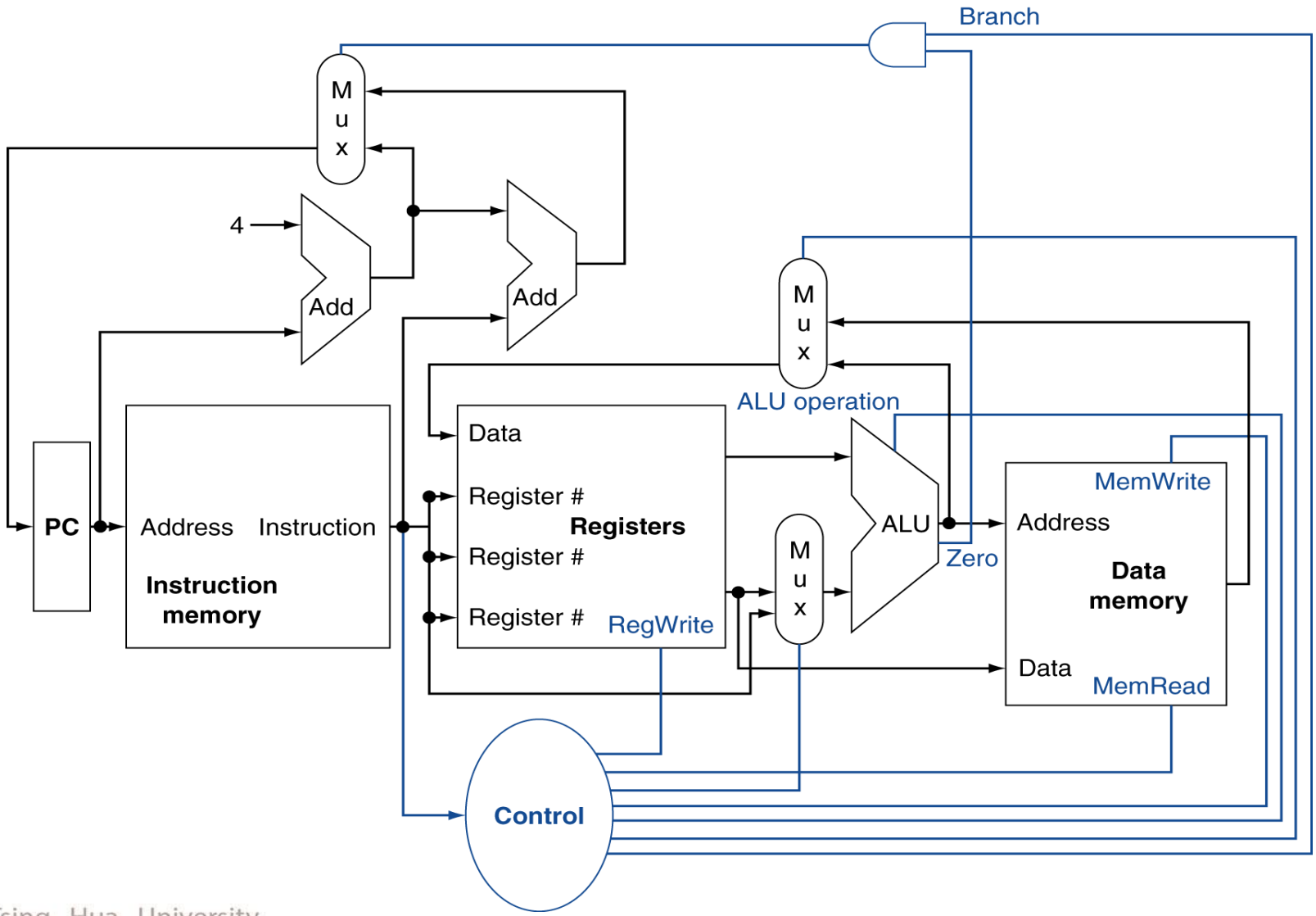


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

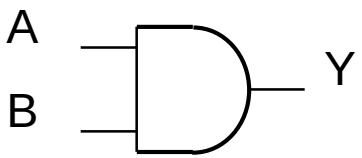

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

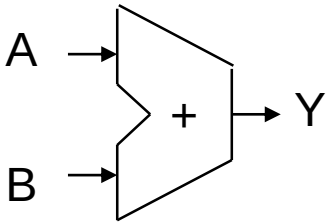
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

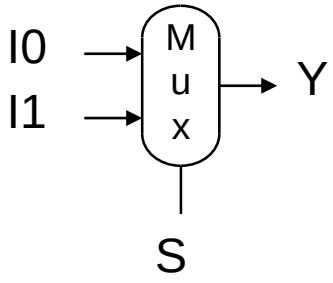
- **AND-gate**
  - **Y = A & B**



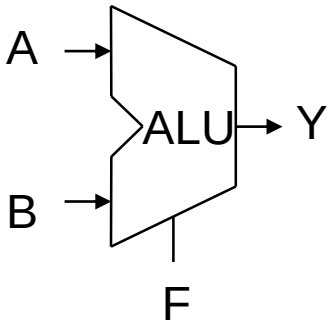
- **Adder**
  - **Y = A + B**



- **Multiplexer**
  - **Y = S ? I1 : I0**

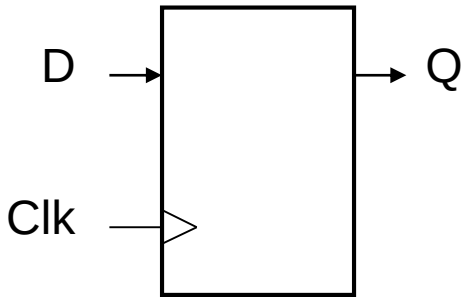


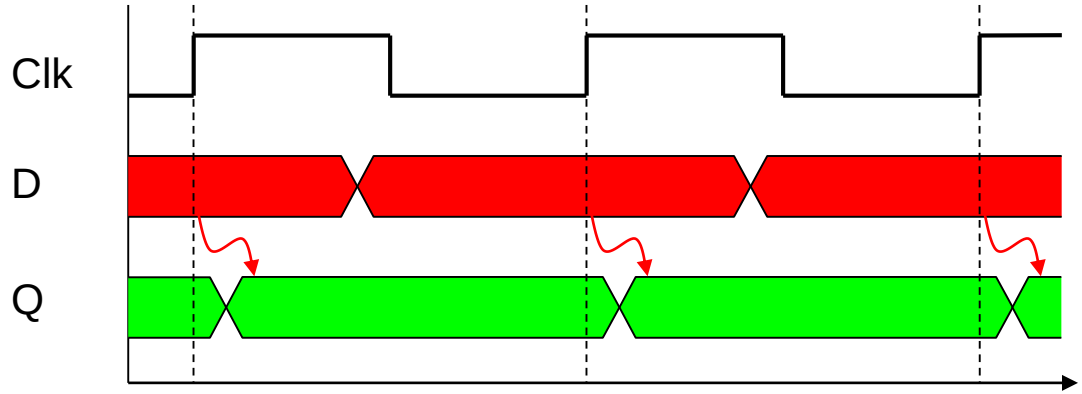
- **Arithmetic/Logic Unit**
  - **Y = F(A, B)**



Sequential Elements

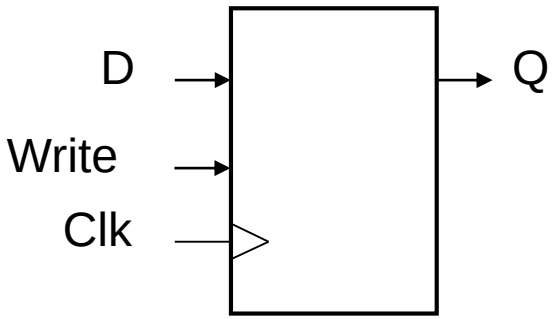
- **Register: stores data in a circuit**
  - **Uses a clock signal to determine when to update the stored value**
  - **Edge-triggered: update when Clk changes from 0 to 1**

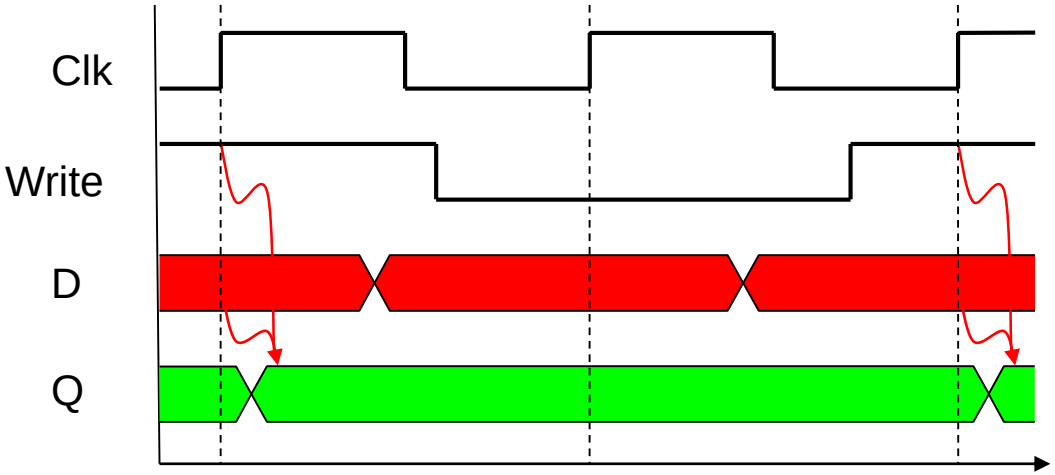




Sequential Elements

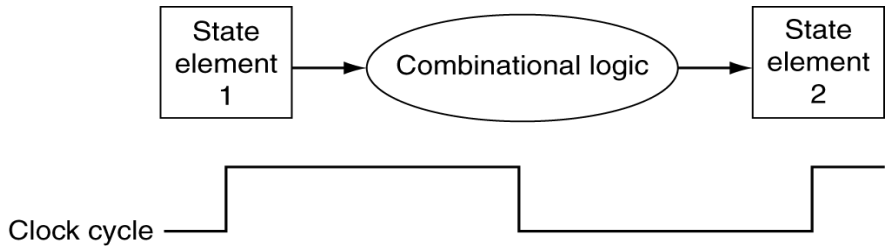
- **Register with write control**
  - **Only updates on clock edge when write control input is 1**
  - **Used when stored value is required later**

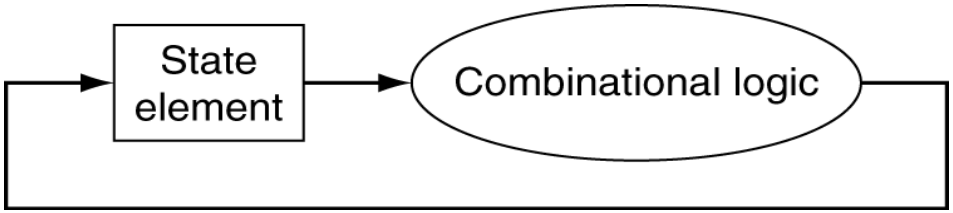




Clocking Methodology

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





How to Design a Processor?

1. Analyze instruction set (datapath requirements)
   - The meaning of each instruction is given by the *register transfers*
   - Datapath must include storage element
   - Datapath must support each register transfer
2. Select set of datapath components and establish clocking methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points effecting register transfer
5. Assemble the control logic

Outline

- Introduction to designing a processor
- Analyzing the instruction set (step 1)
- Building the datapath (steps 2 and 3)
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

Step 1: Analyze Instruction Set

- All MIPS instructions are 32 bits long with 3 formats:
  - R-type:

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

  - I-type:

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

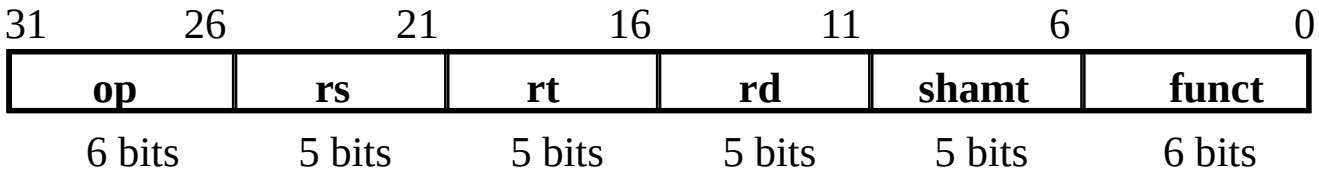
  - J-type:

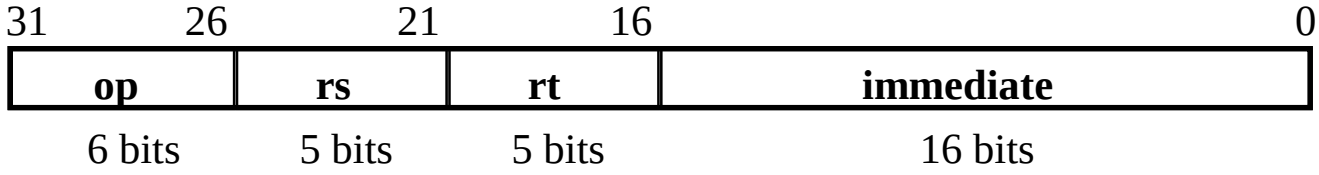
31–26	25–0
op	target address
6 bits	26 bits

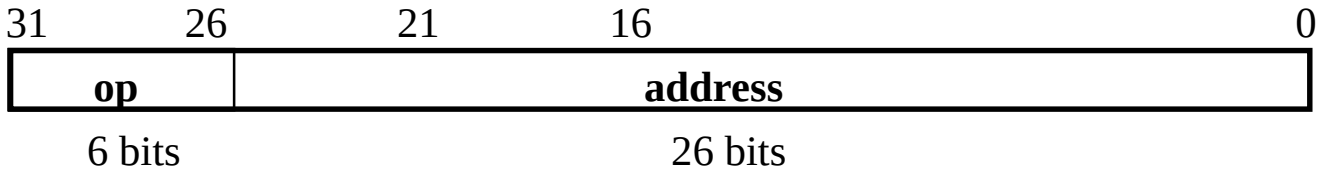
- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: source and destination register
  - shamt: shift amount
  - funct: selects variant of the “op” field
  - address / immediate
  - target address: target address of jump

Our Example: A MIPS Subset

- **R-Type:**
  - add rd, rs, rt
  - sub rd, rs, rt
  - and rd, rs, rt
  - or rd, rs, rt
  - slt rd, rs, rt
- **Load/Store:**
  - lw rt,rs,imm16
  - sw rt,rs,imm16
- **Imm operand:**
  - addi rt,rs,imm16
- **Branch:**
  - beq rs,rt,imm16
- **Jump:**
  - j target







Logical Register Transfers

- RTL gives the <u>meaning</u> of the instructions
- All start by fetching the instruction, read registers, then use ALU => simplicity and regularity help

```
MEM[ PC ] = op | rs | rt | rd | shamt | funct
      or      = op | rs | rt |   Imm16
      or      = op | Imm26 (added at the end)
```

Inst	Register transfers
ADD	R[rd] <- R[rs] + R[rt]; PC <- PC + 4
SUB	R[rd] <- R[rs] - R[rt]; PC <- PC + 4
LOAD	R[rt] <- MEM[R[rs] + sign_ext(Imm16)]; PC <- PC + 4
STORE	MEM[R[rs] + sign_ext(Imm16)] <-R[rt]; PC <- PC + 4
ADDI	R[rt] <- R[rs] + sign_ext(Imm16)]; PC <- PC + 4
BEQ	if (R[rs] == R[rt]) then PC <- PC + 4 + sign_ext(Imm16)] || 00 else PC <- PC + 4

Requirements of Instruction Set

After checking the register transfers, we can see that datapath needs the followings:

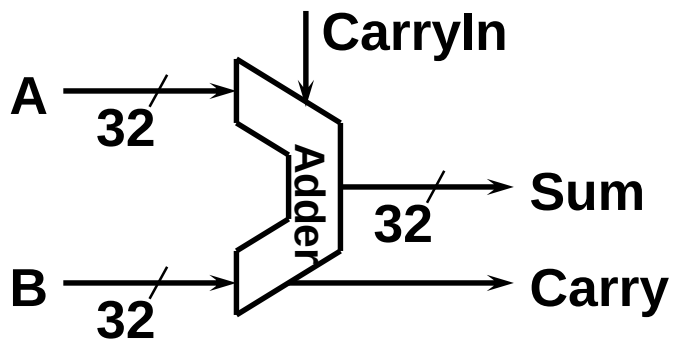
- Memory
  - store instructions and data
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Extender for zero- or sign-extension
- Add and sub register or extended immediate (ALU)
- Add 4 or extended immediate to PC

Outline

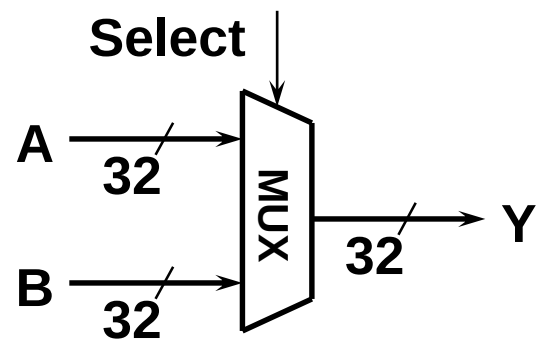
- Introduction to designing a processor
- Analyzing the instruction set (step 1)
- Building the datapath (steps 2, 3)
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller
- Adding *jump* instruction

Step 2a: Combinational Components for Datapath

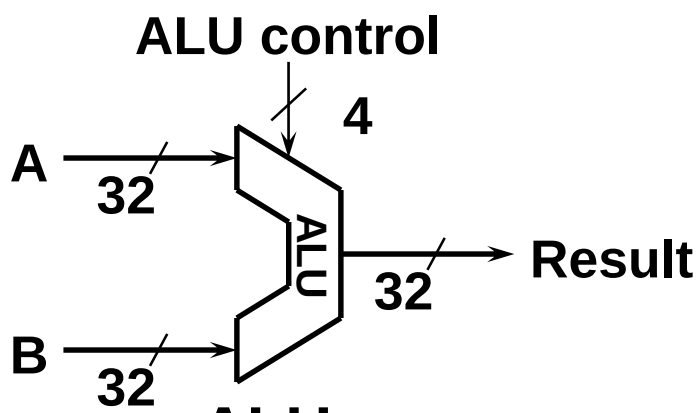
- **Basic building blocks of combinational logic elements :**



Adder



MUX

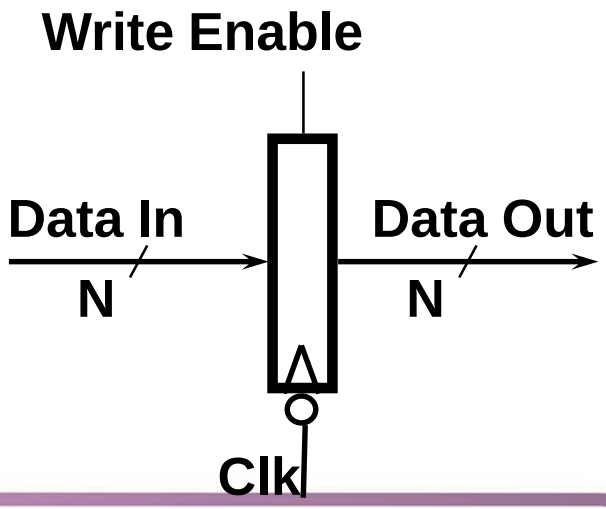


ALU

Step 2b: Sequential Components for Datapath

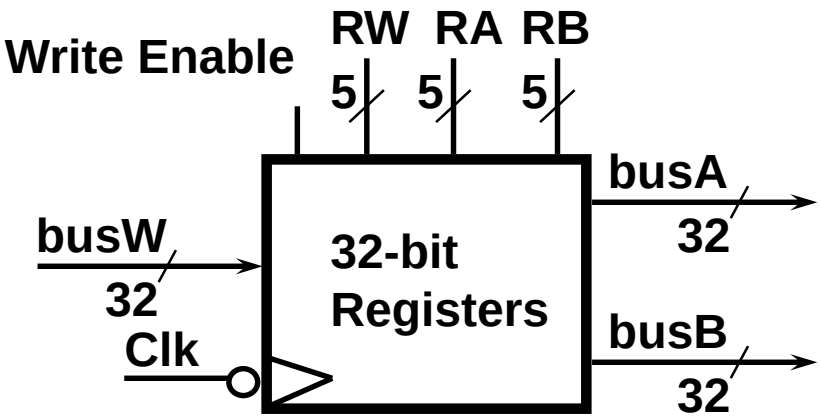
Storage elements:

- **Register:**
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - negated (0): Data Out will not change
    - asserted (1): Data Out will become Data In



Storage Element: Register File

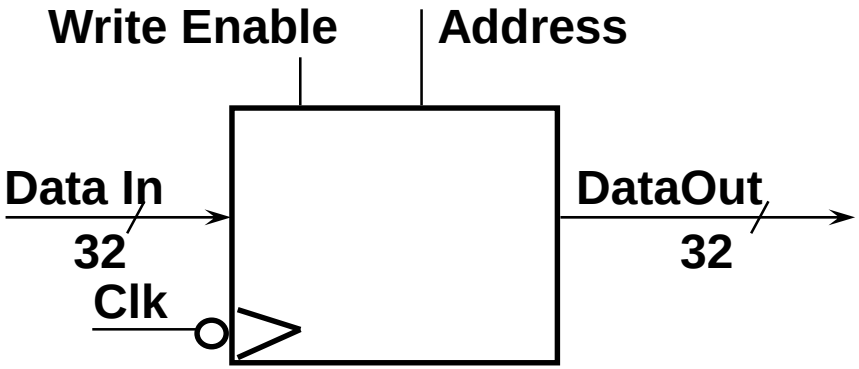
- Consists of 32 registers:
  - Appendix B.8
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW



- Register is selected by:
  - RA selects the register to put on busA (data)
  - RB selects the register to put on busB (data)
  - RW selects the register to be written via busW (data) when Write Enable is 1
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read, behaves as a combinational circuit

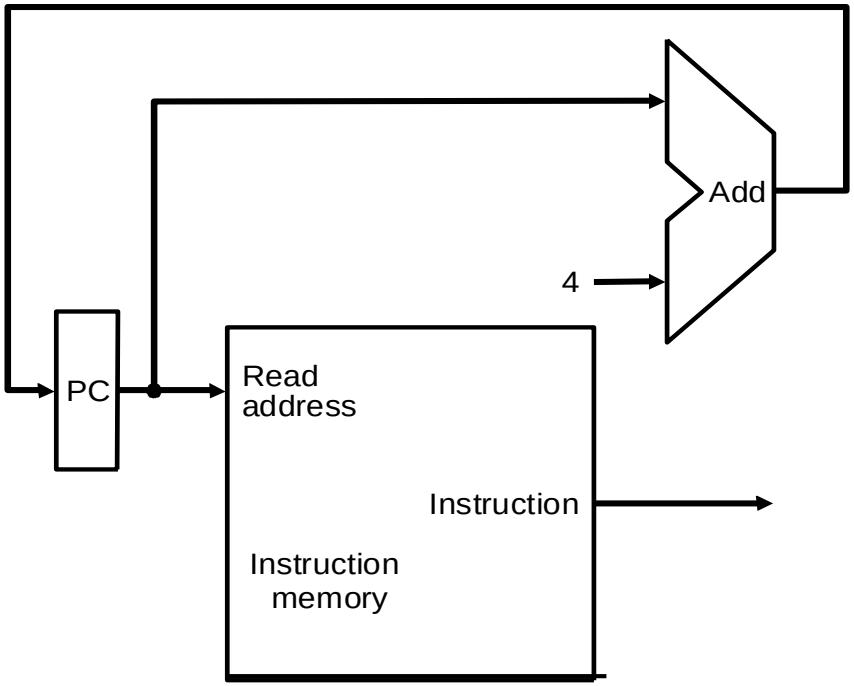
Storage Element: Memory

- Memory (idealized)
  - Appendix B.8
  - One input bus: Data In
  - One output bus: Data Out
- Word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after access time
    - No need for read control



Step 3a: Datapath Assembly

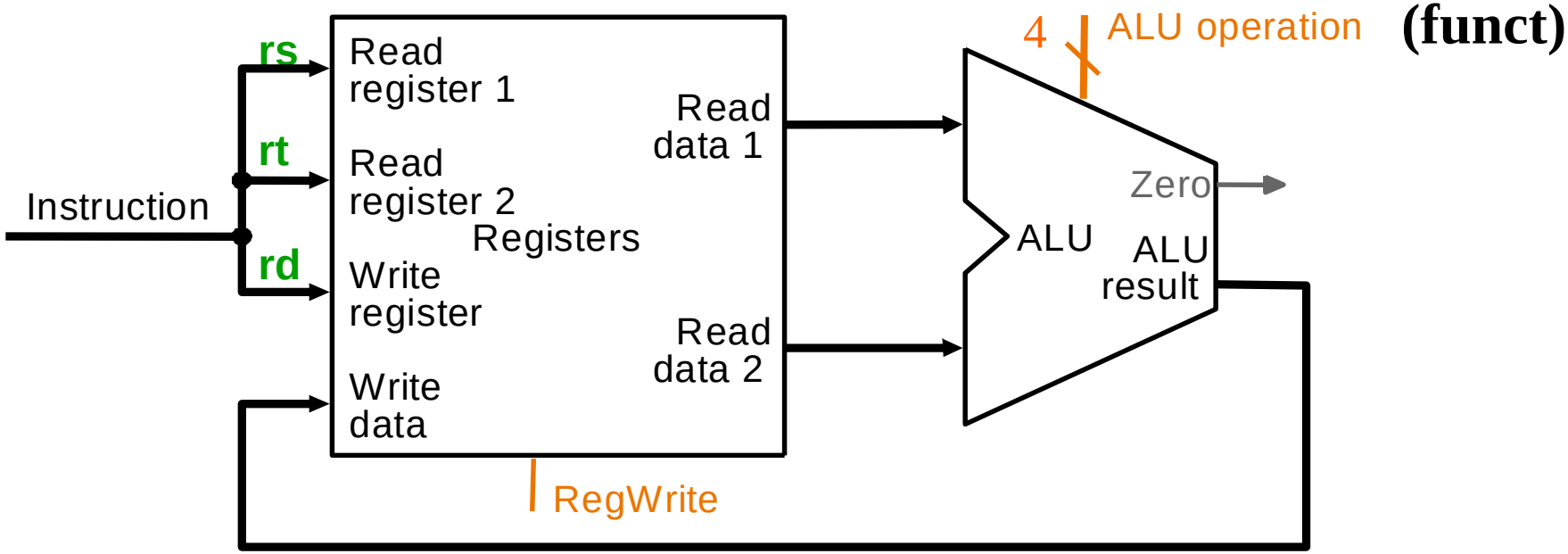
- **Instruction fetch unit: common operations**
  - **Fetch the instruction: mem[PC]**
  - **Update the program counter:**
    - **Sequential code: PC <- PC + 4**
    - **Branch and Jump: PC <- "Something else"**



Step 3b: Add and Subtract

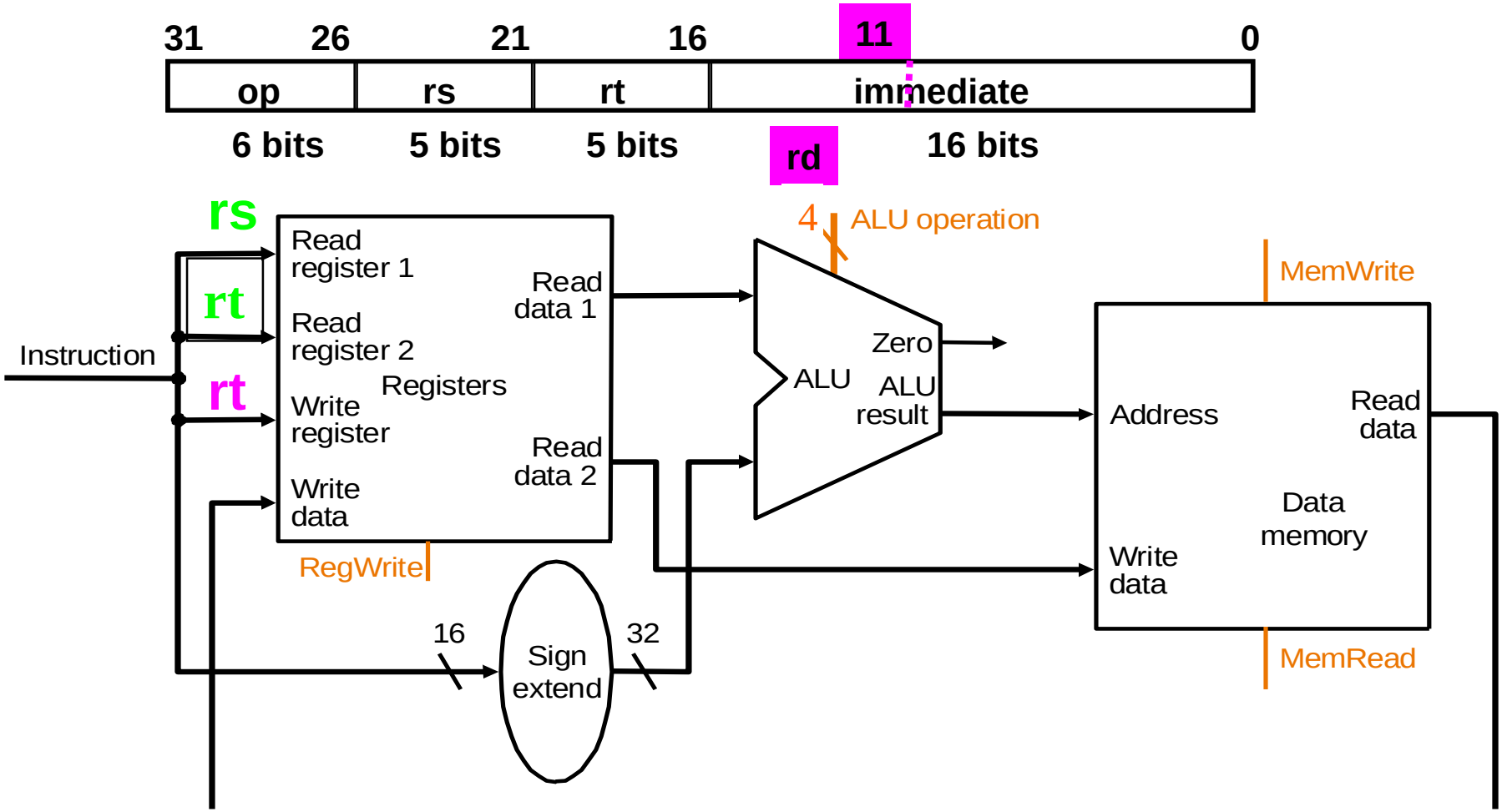
- R[rd] <- R[rs] op R[rt] Ex: add rd, rs, rt
  - Ra, Rb, Rw come from inst.'s rs, rt, and rd fields
  - ALU and RegWrite: control logic after decode

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

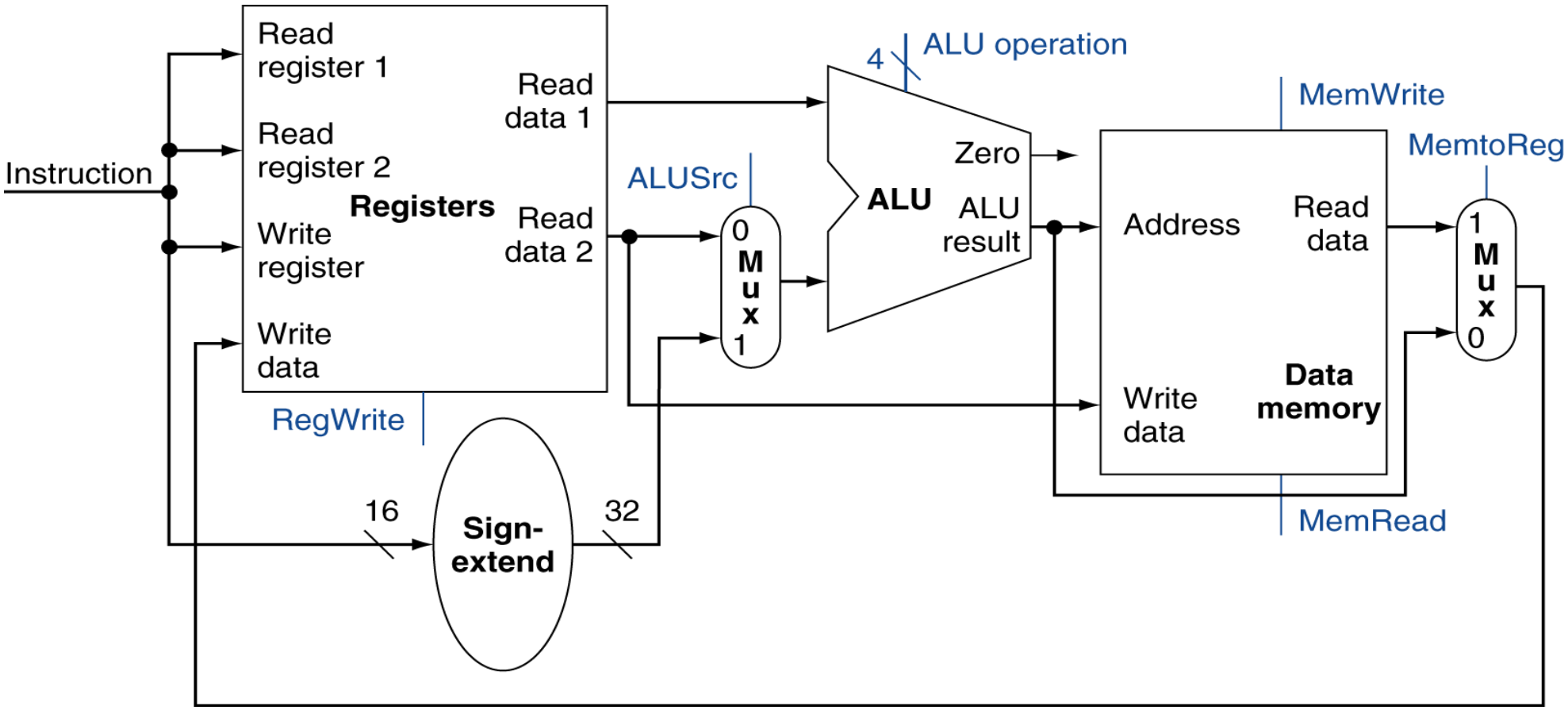


Step 3c: Store/Load Operations

- R[rt]<-Mem[R[rs]+SignExt[imm16]] Ex: lw rt,rs,imm16



R-Type/Load/Store Datapath

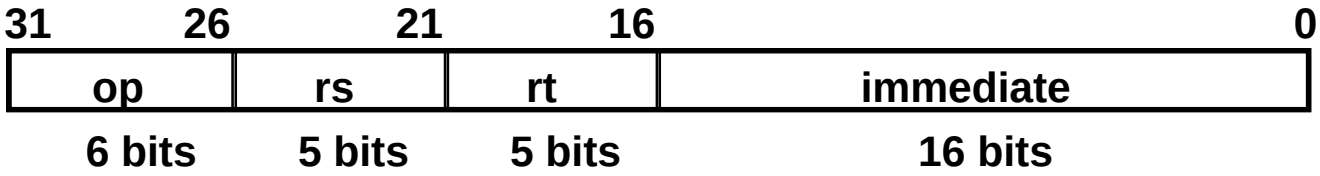


Step 3d: Branch Operations

- **beq rs, rt, imm16**

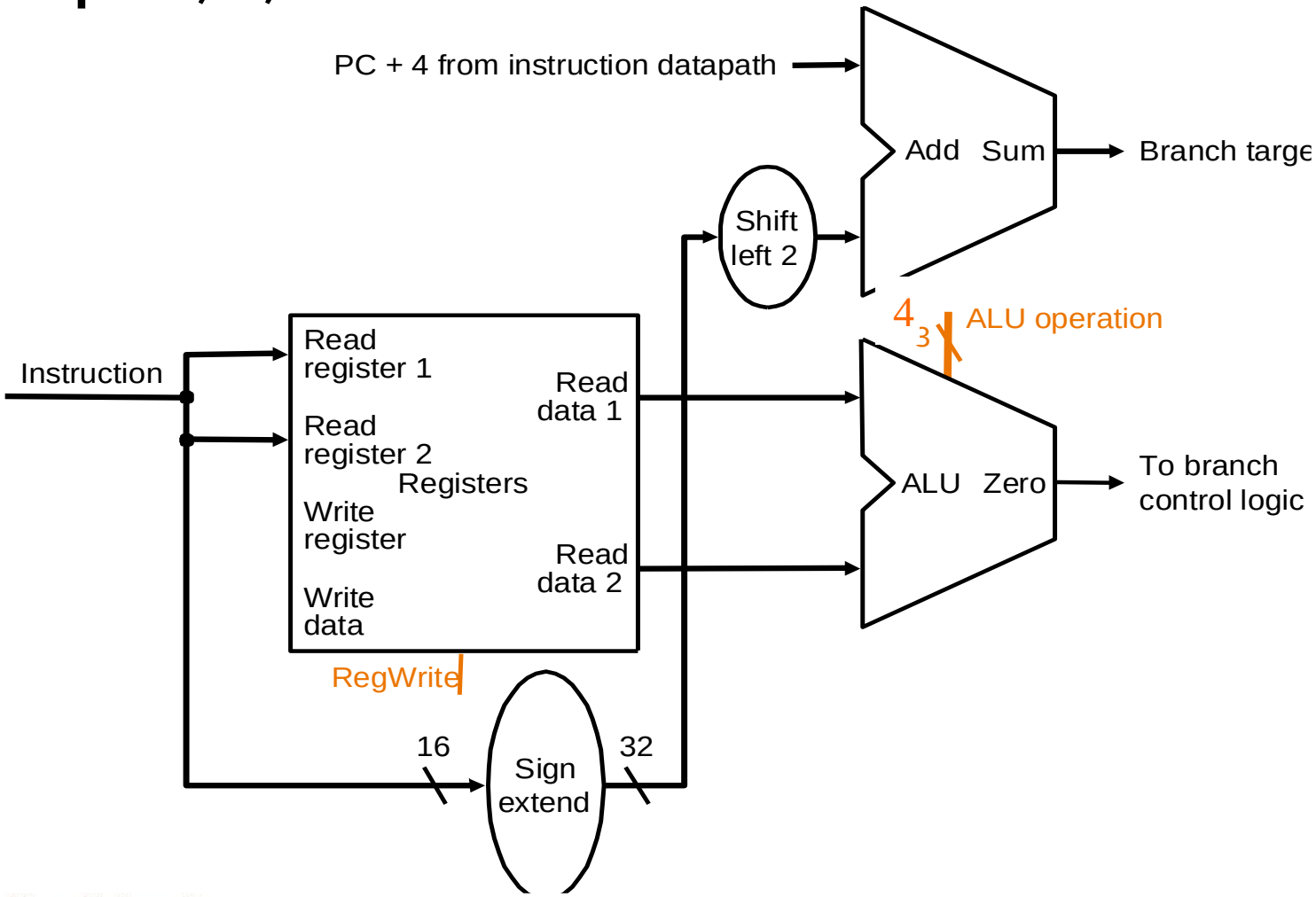
mem[PC]	Fetch inst. from memory
Equal <- R[rs] == R[rt]	Calculate branch condition
if (COND == 0)	Calculate next inst. address

```
    PC  <-  PC + 4 + ( SignExt(imm16) x 4 )
else
    PC  <-  PC + 4
```



Datapath for Branch Operations

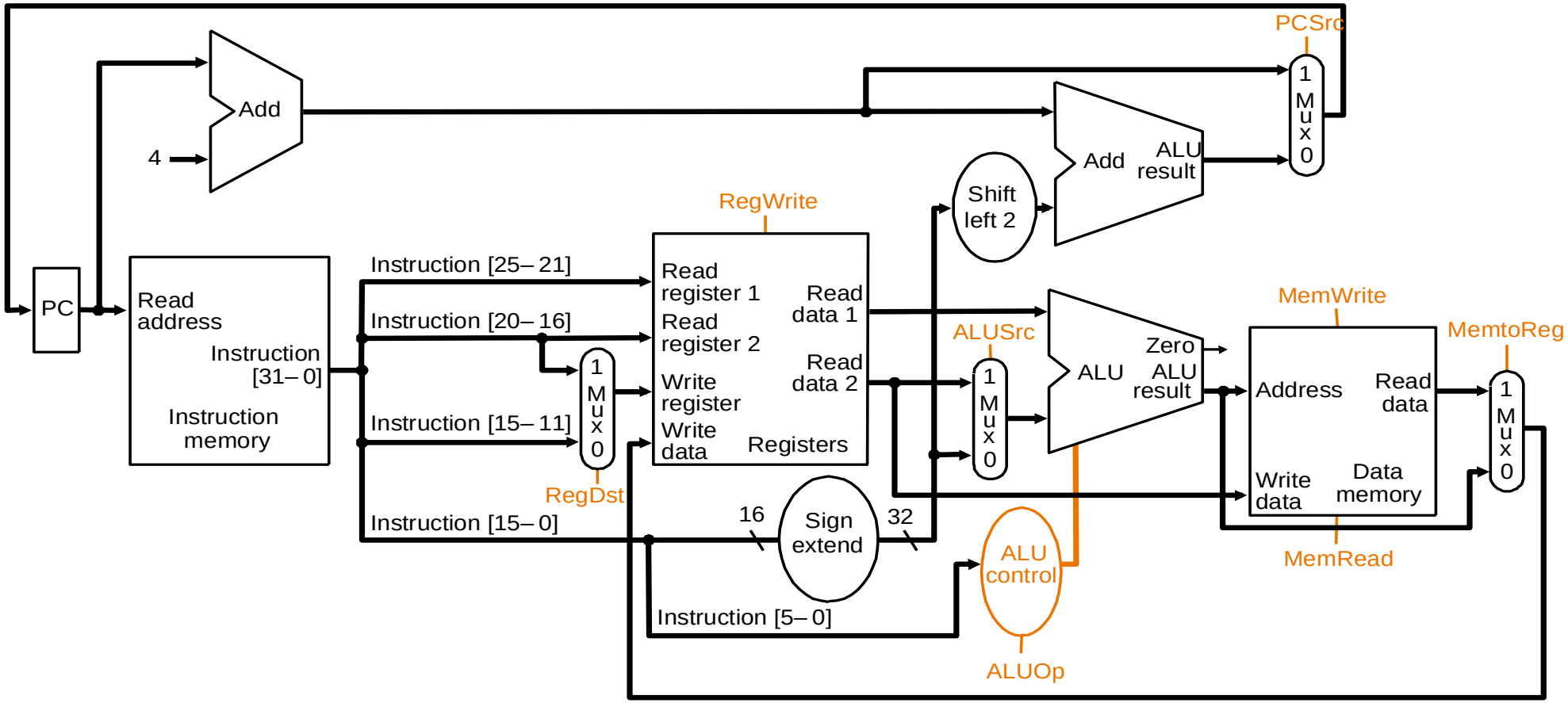
- **beq rs, rt, imm16**



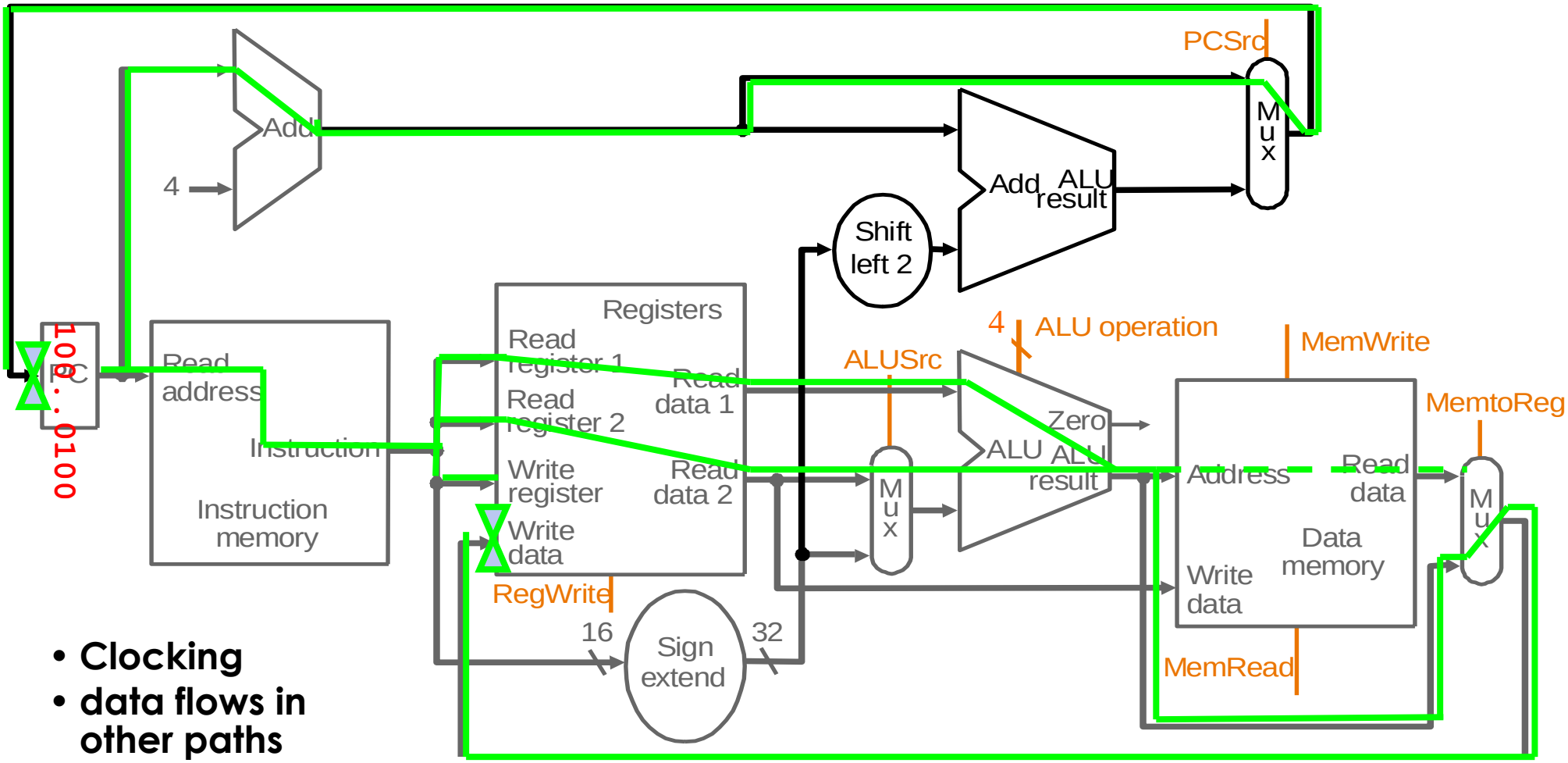
Outline

- Introduction to designing a processor
- Analyzing the instruction set
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller

A Single Cycle Datapath



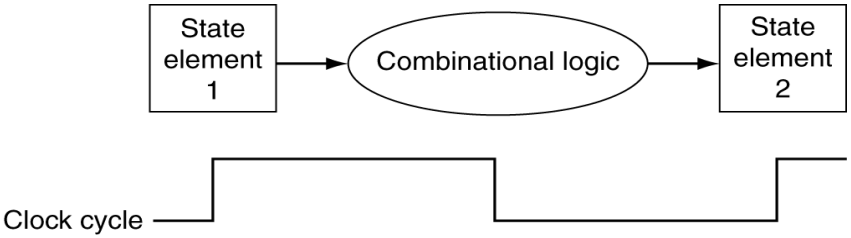
Data Flow during add

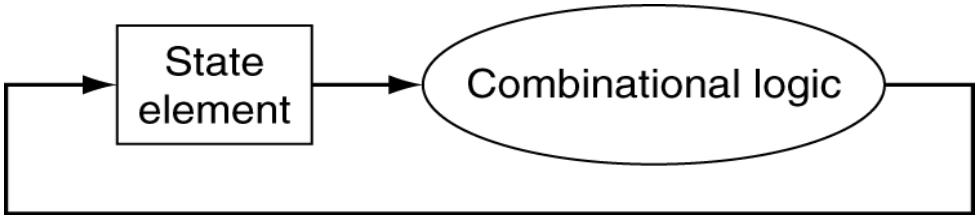


- **Clocking**
- **data flows in other paths**

Clocking Methodology

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

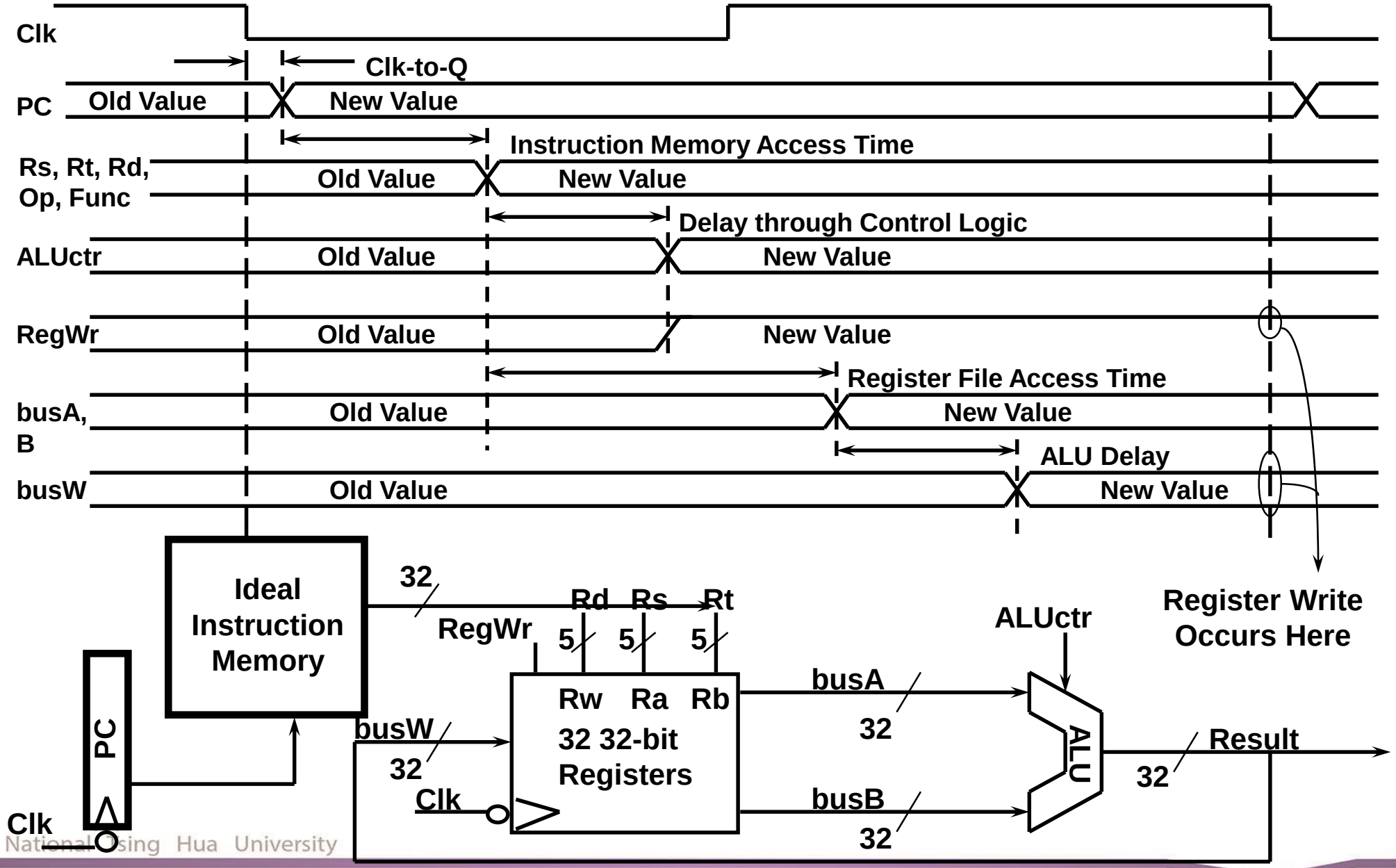




Clocking Methodology

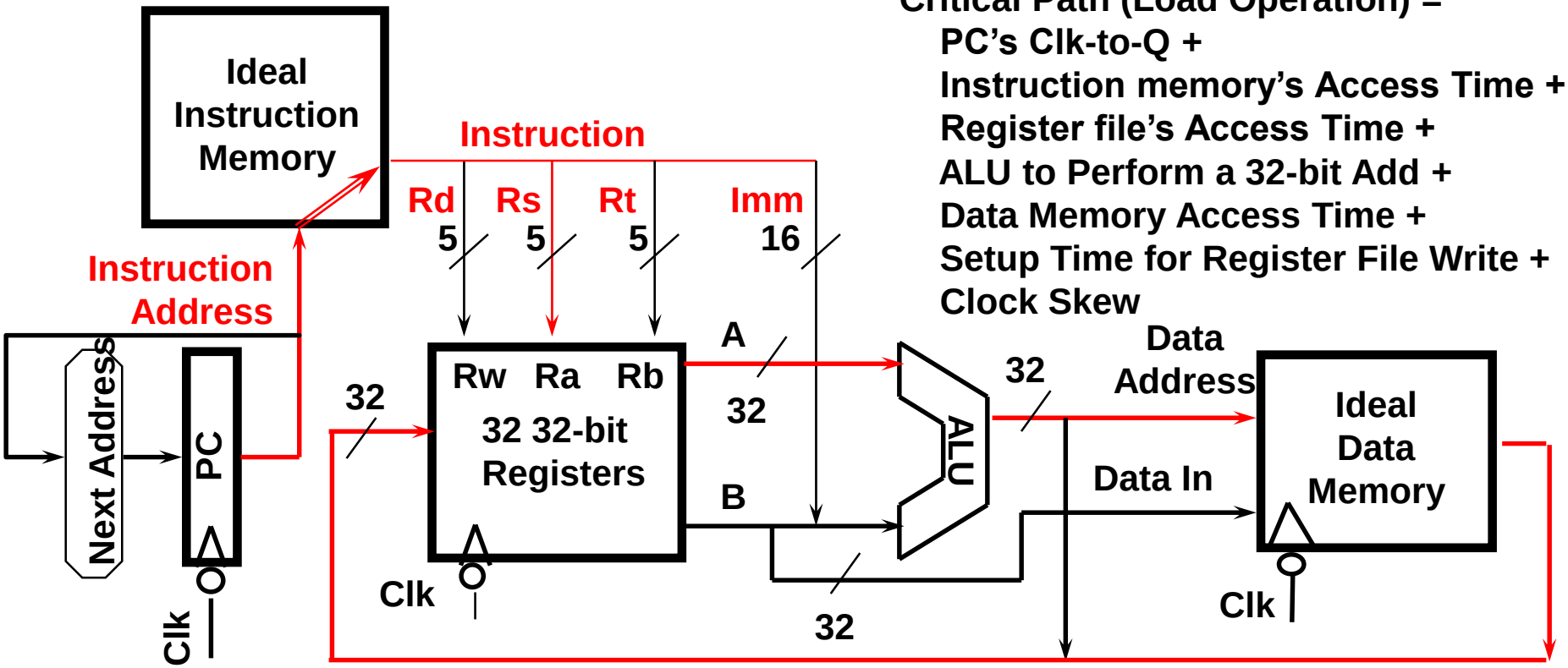
- Define when signals are read and written
- Assume edge-triggered:
  - Values in storage (state) elements updated only on a clock edge
    => clock edge should arrive only after input signals stable
  - Any combinational circuit must have inputs from and outputs to storage elements
  - *Clock cycle*: time for signals to propagate from one storage element, through combinational circuit, to reach the second storage element
  - A register can be read, its value propagated through some combinational circuit, new value is written back to the same register, all in same cycle => no feedback within a single cycle

Register-Register Timing



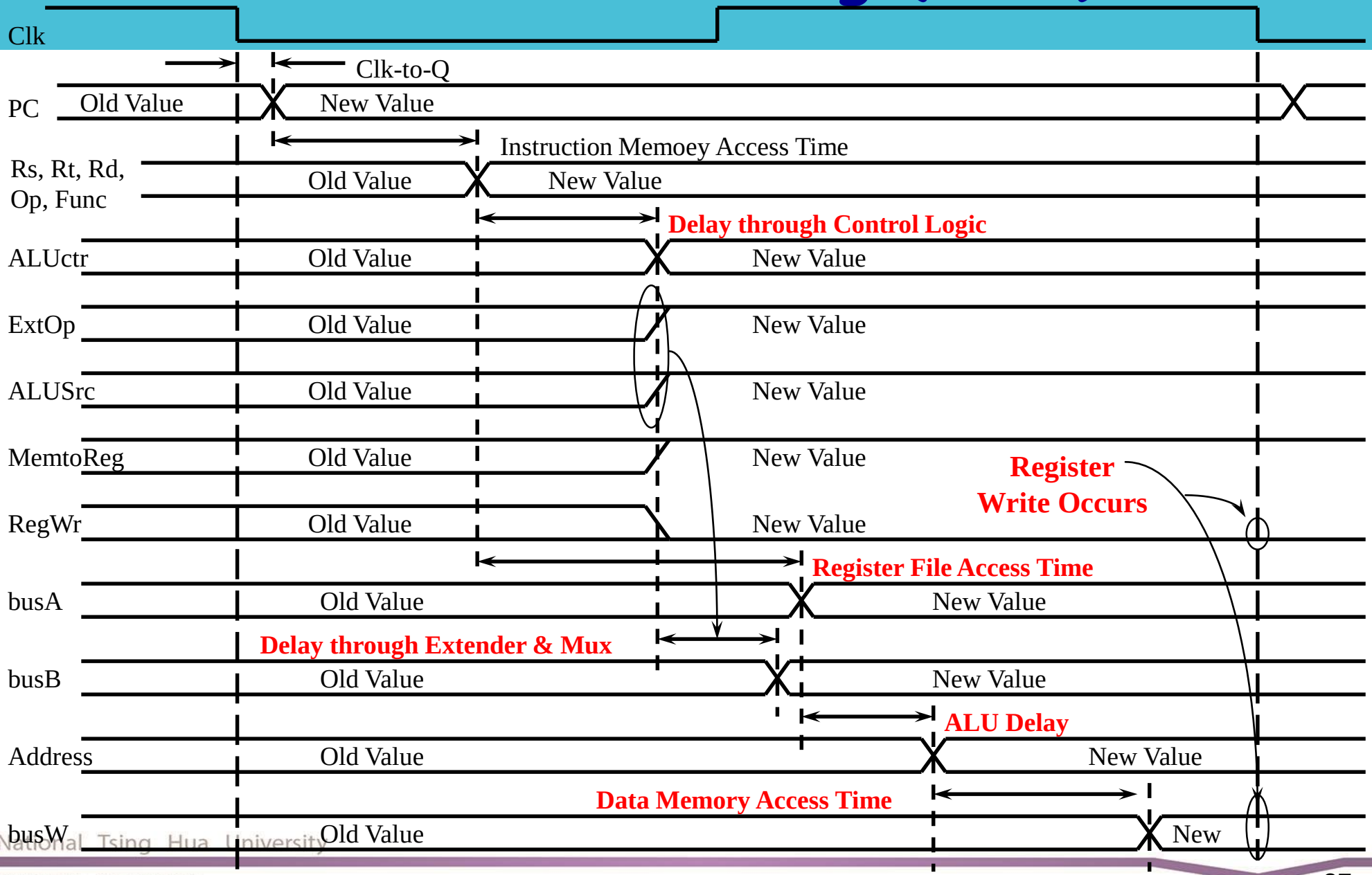
The Critical Path

- **Register file and ideal memory:**
  - **During read, behave as combinational logic:**
    - **Address valid => Output valid after access time**



Critical Path (Load Operation) =
PC's Clk-to-Q +
Instruction memory's Access Time +
Register file's Access Time +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

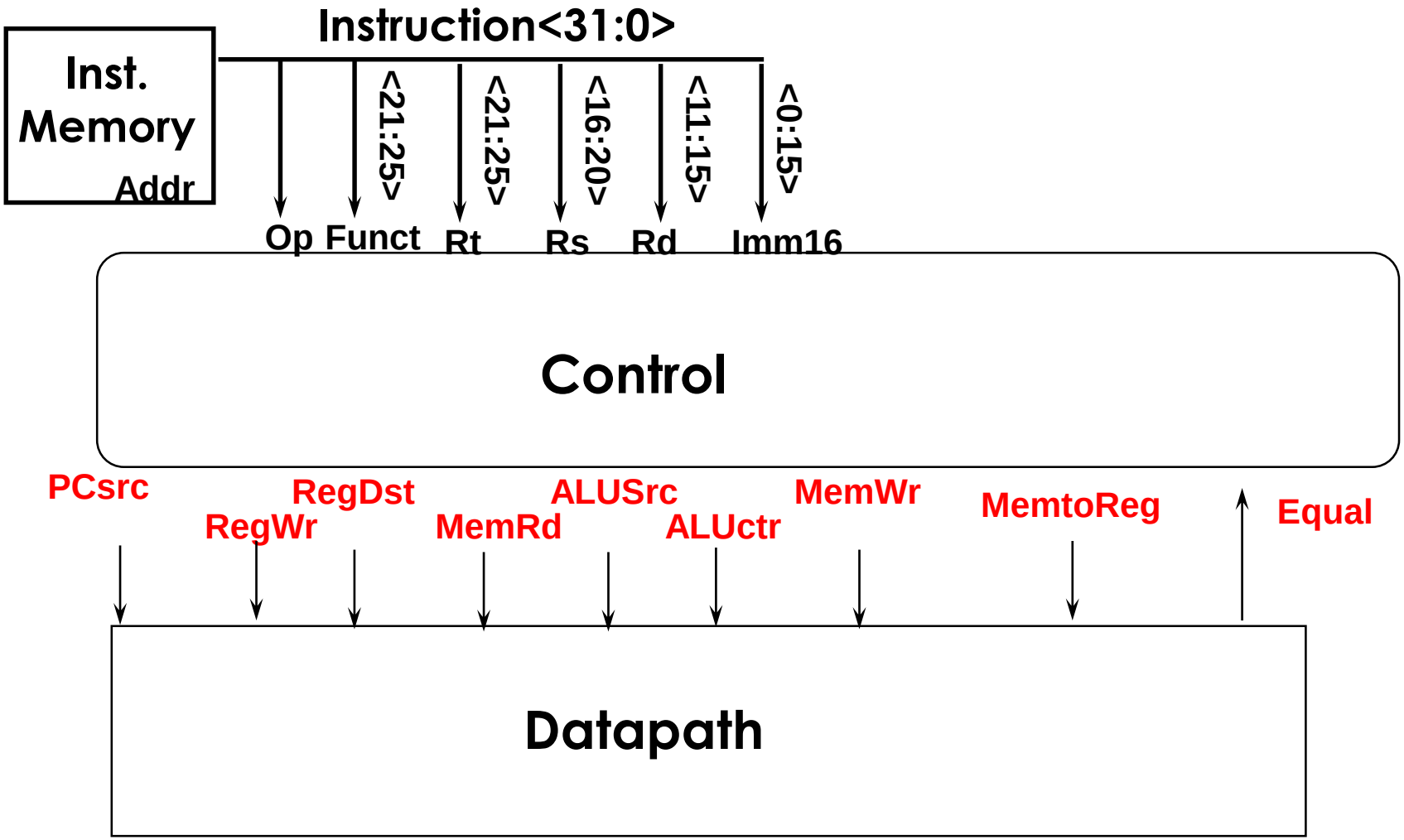
Worst Case Timing (Load)



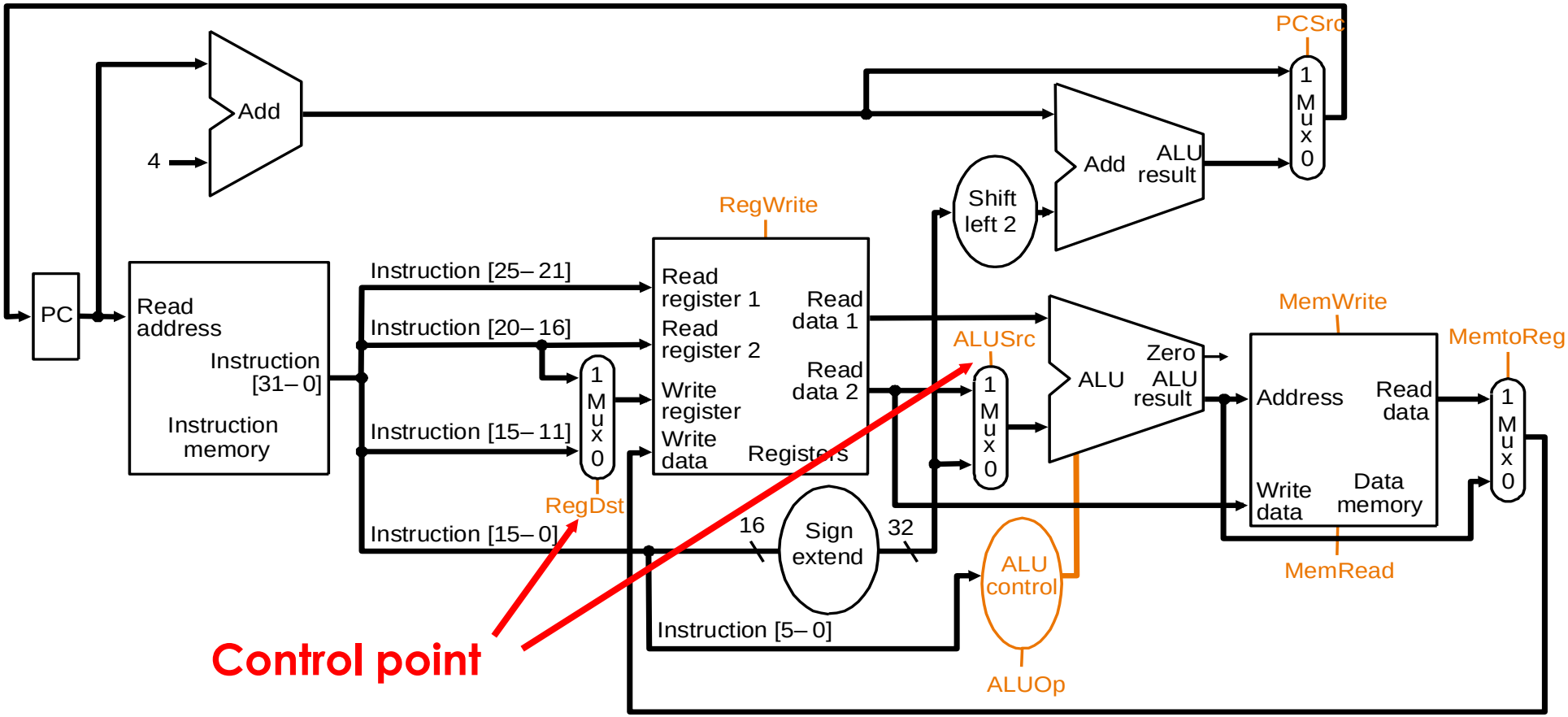
Outline

- Introduction to designing a processor
- Analyzing the instruction set
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations (step 4)
  - ALU controller (step 5a)
  - Main controller (step 5b)
- Adding *jump* instruction

Step 4: Control Points and Signals



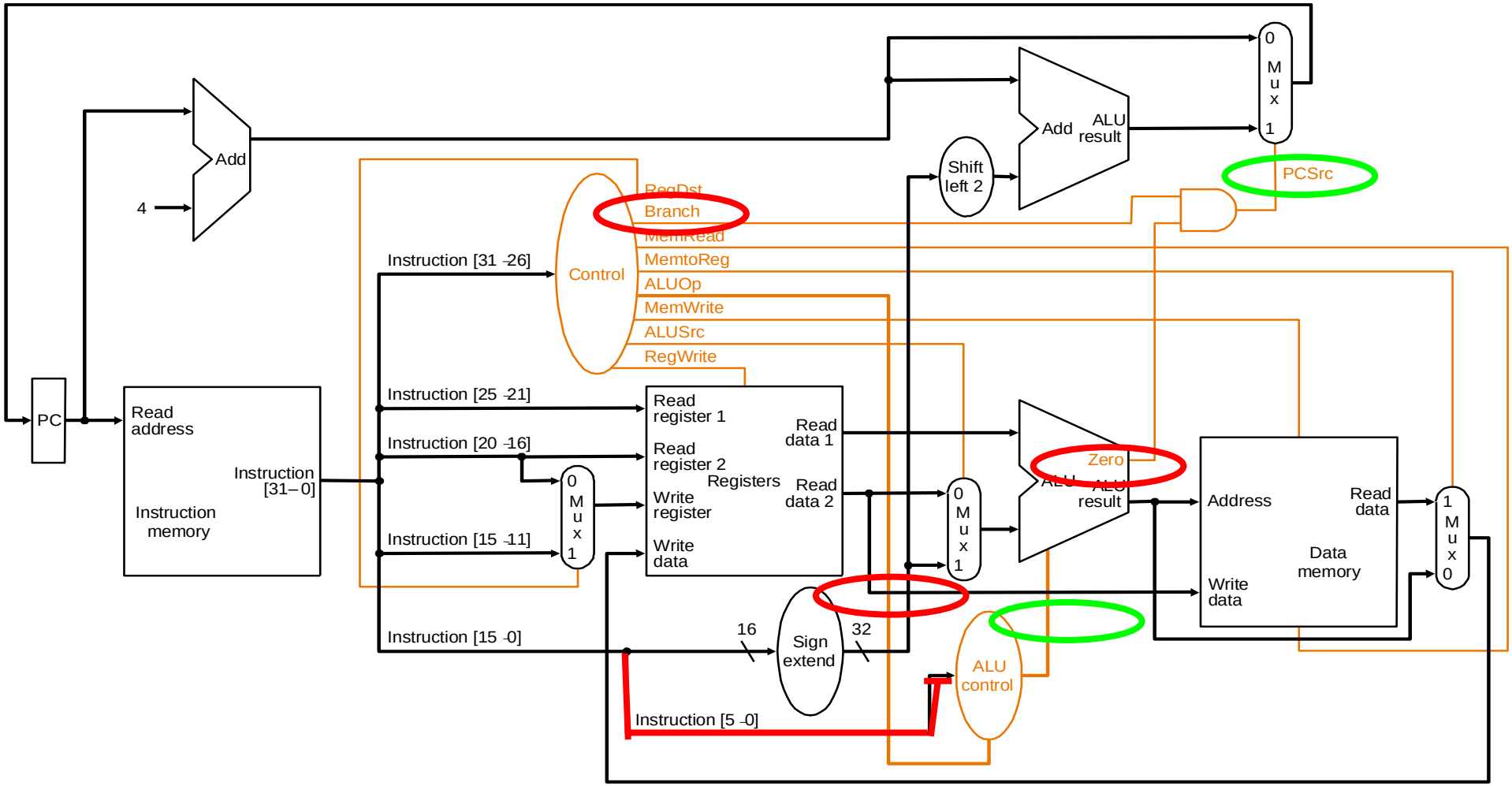
Datapath with Mux and Control



Designing Main Control

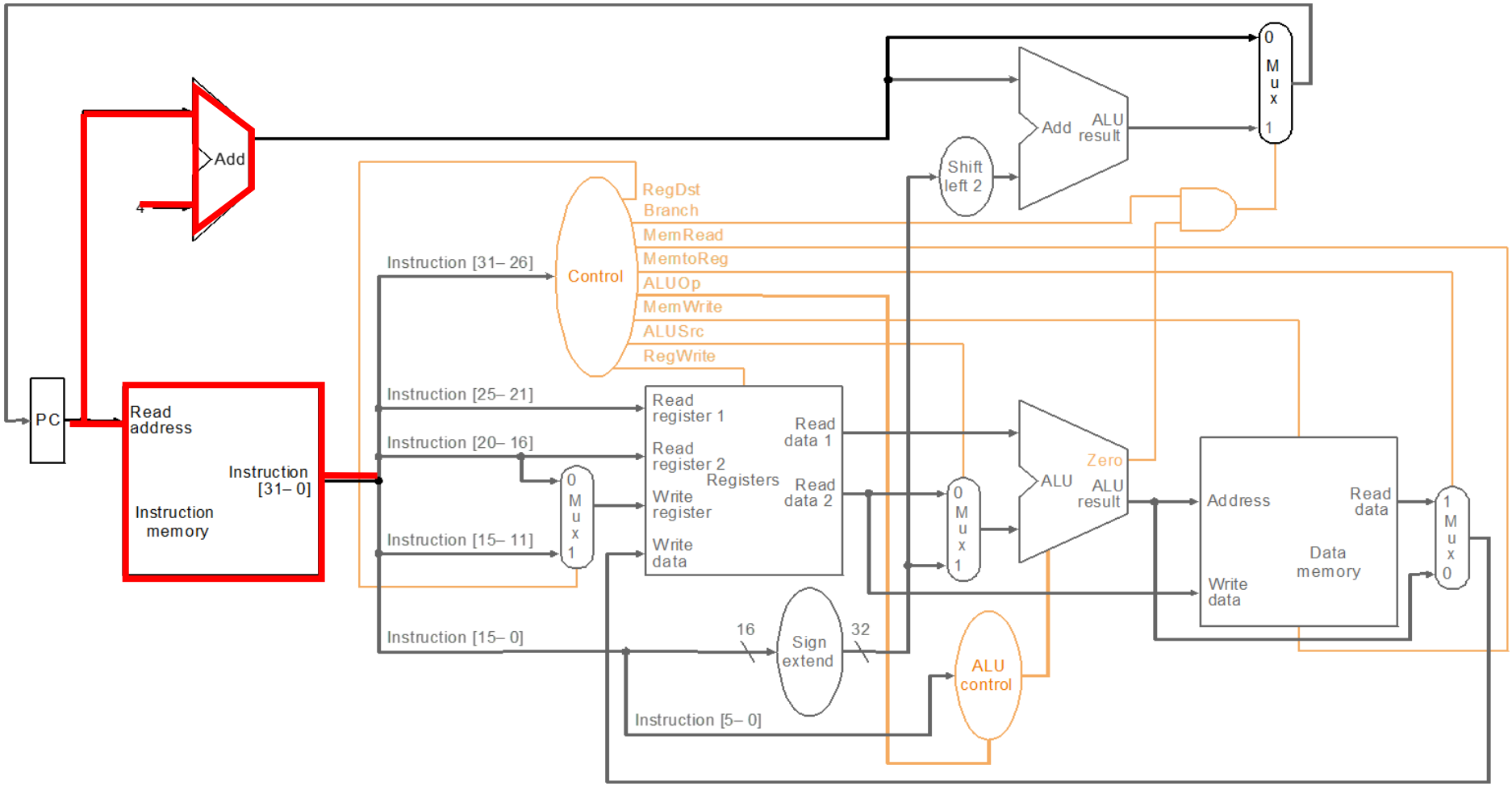
- **Some observations:**
  - **opcode (Op[5-0]) is always in bits 31-26**

Datapath with Control Unit



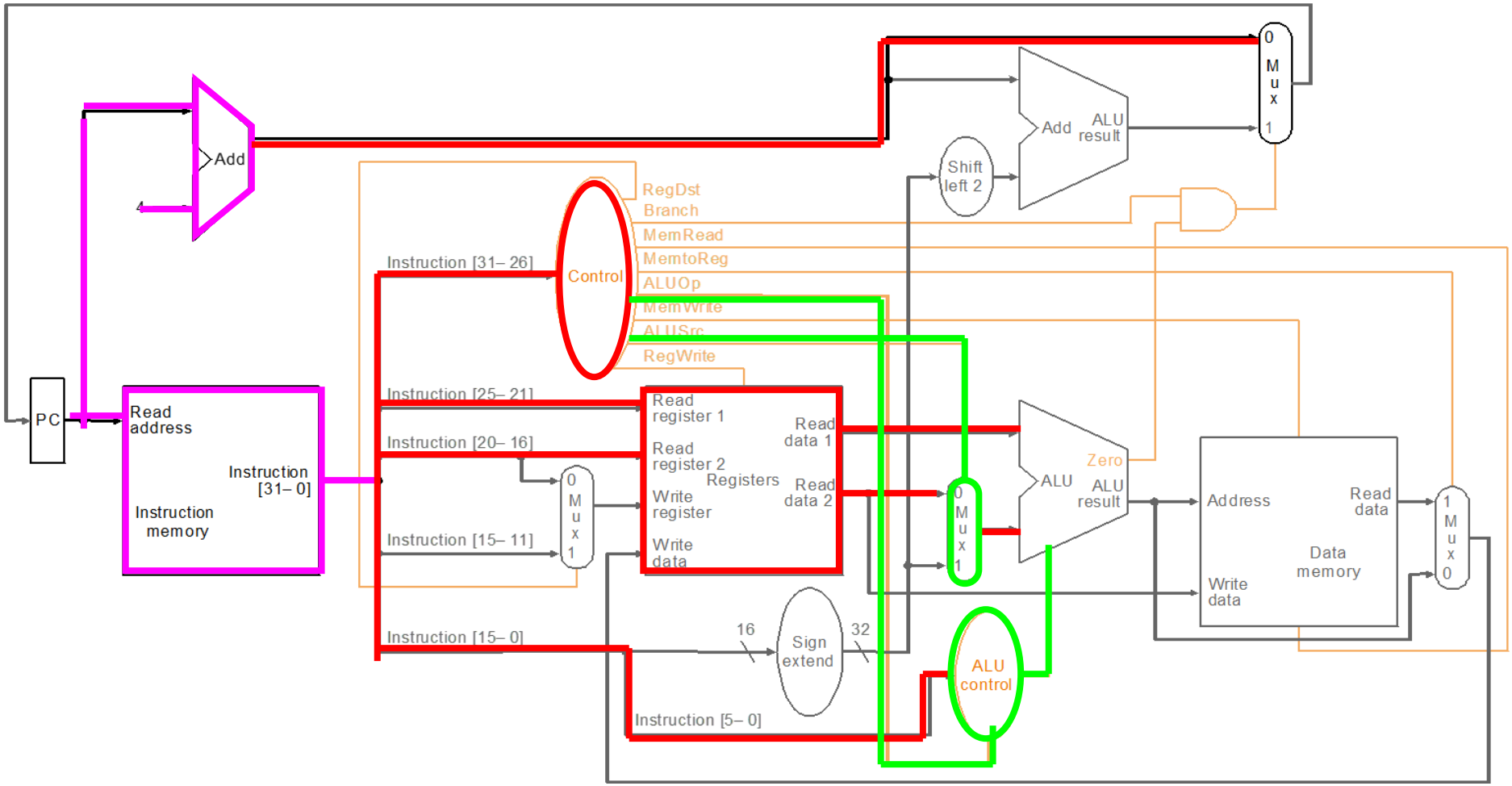
Instruction Fetch at Start of Add

- instruction <- mem[PC]; PC + 4



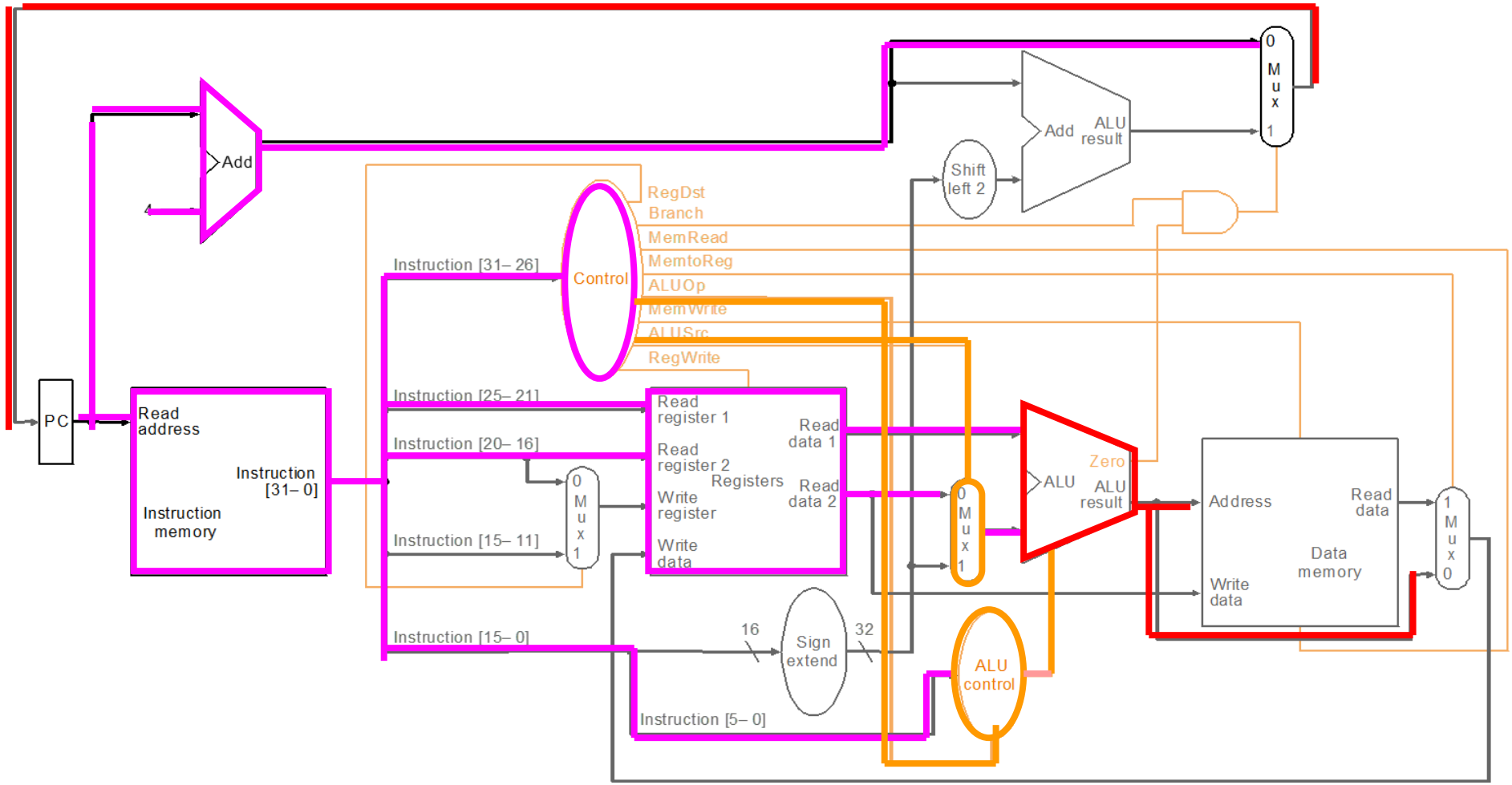
Instruction Decode of Add

- **Fetch the two operands and decode instruction:**



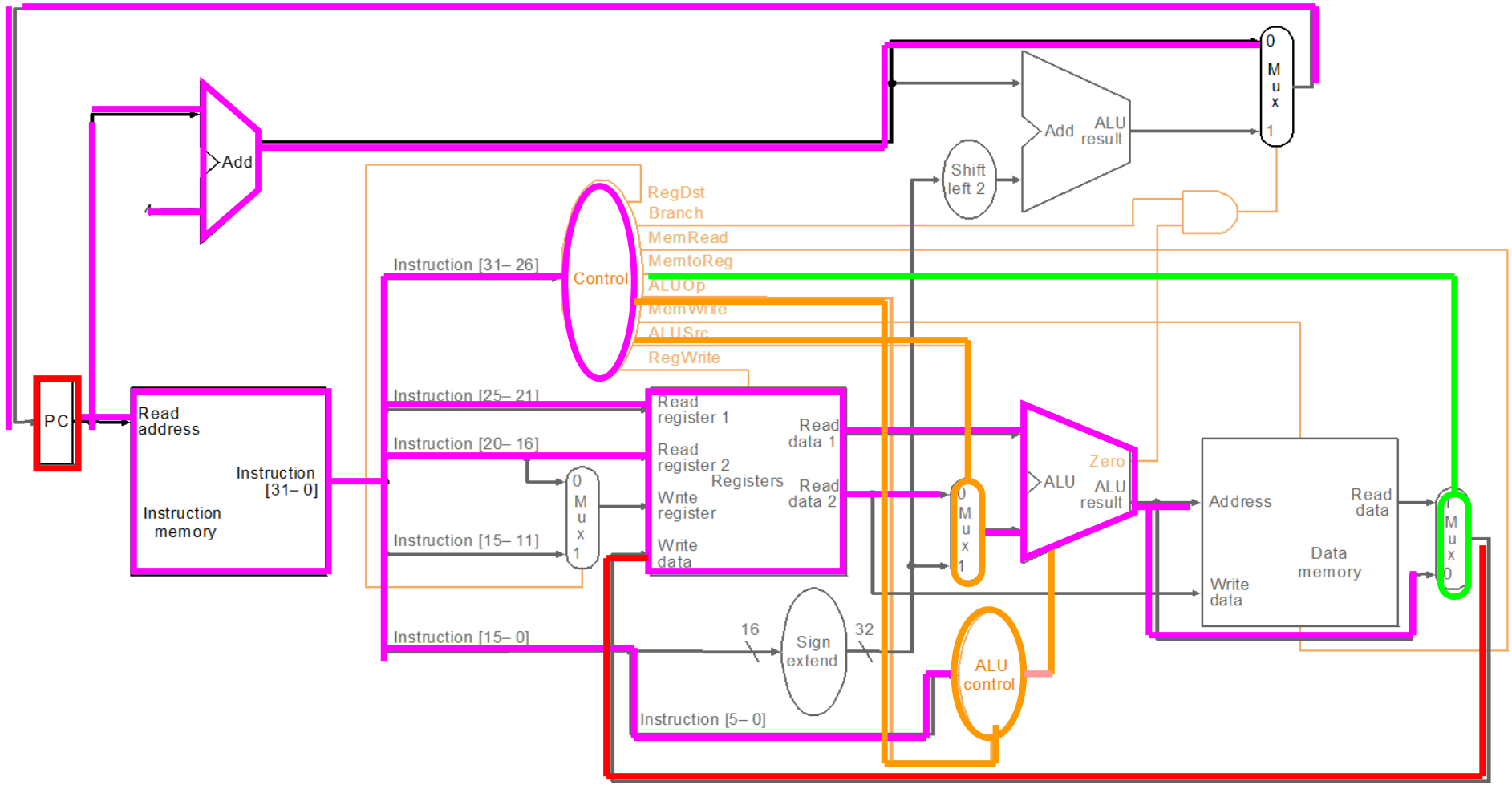
ALU Operation during Add

- R[rs] + R[rt]



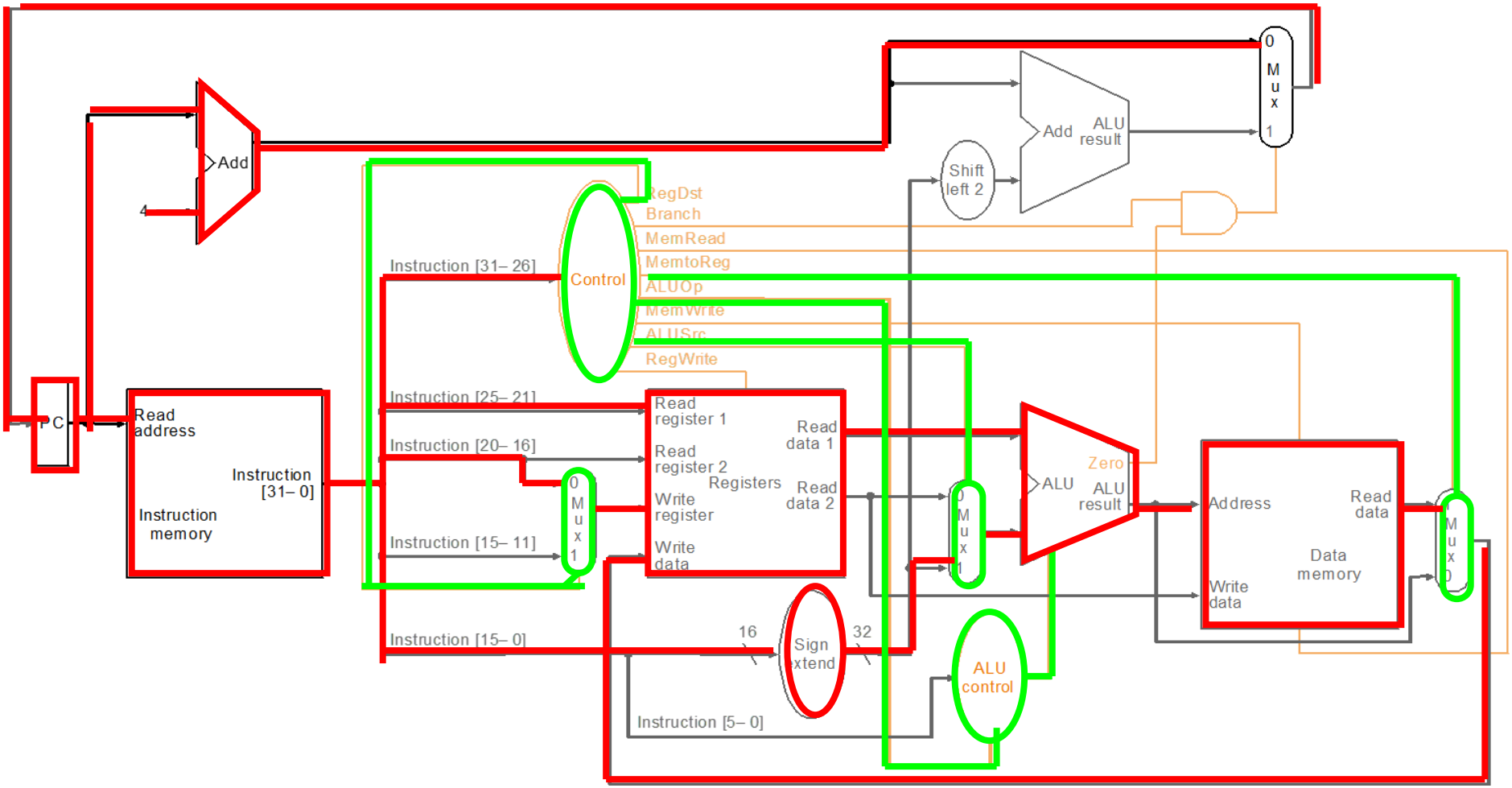
Write Back at the End of Add

- R[rd] <- ALU; PC <- PC + 4



Datapath Operation for lw

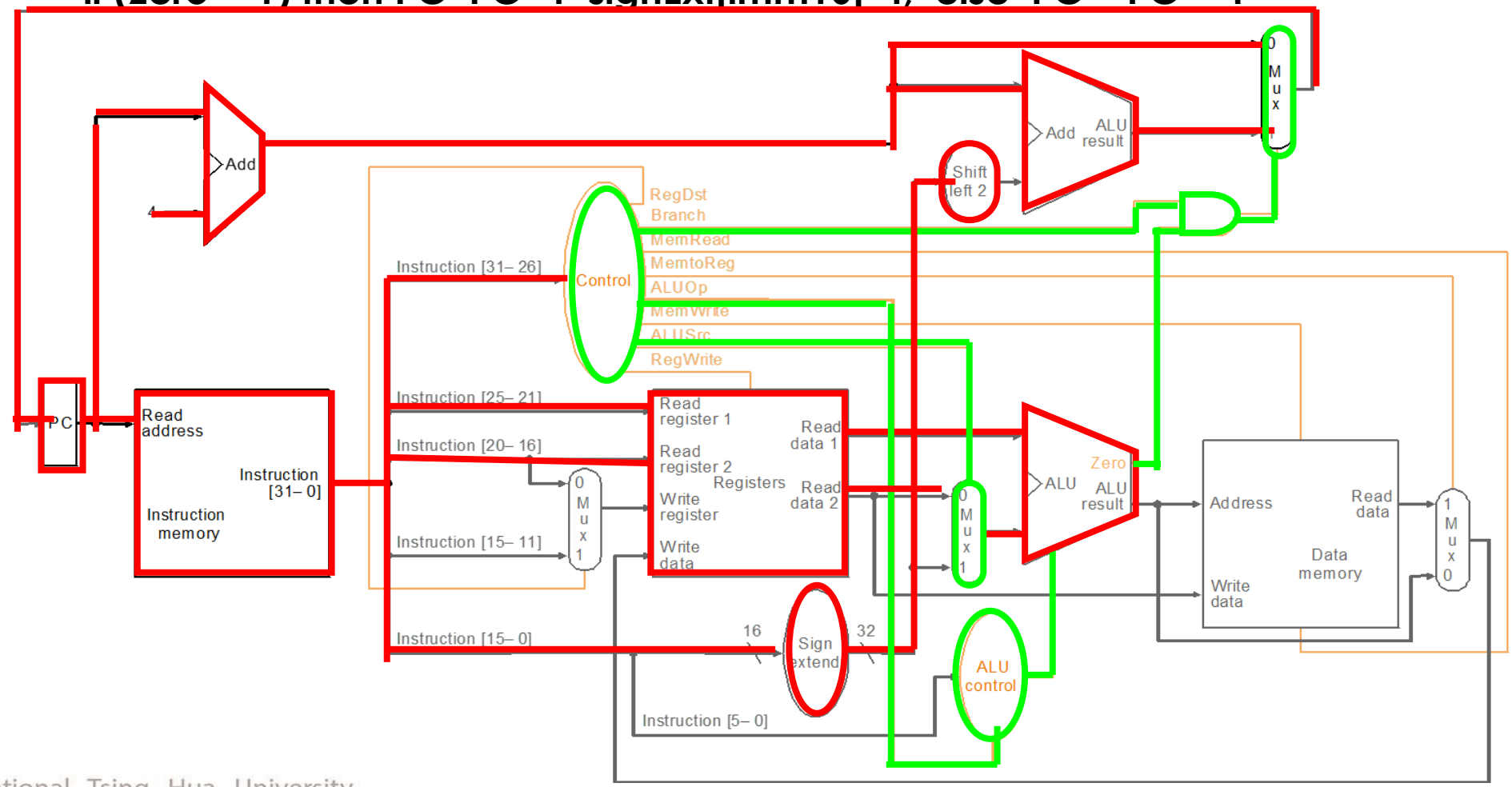
- R[rt] <- Memory {R[rs] + SignExt[imm16]}



Datapath Operation for beq

if (R[rs]-R[rt]==0) then Zero<-1 else Zero<-0

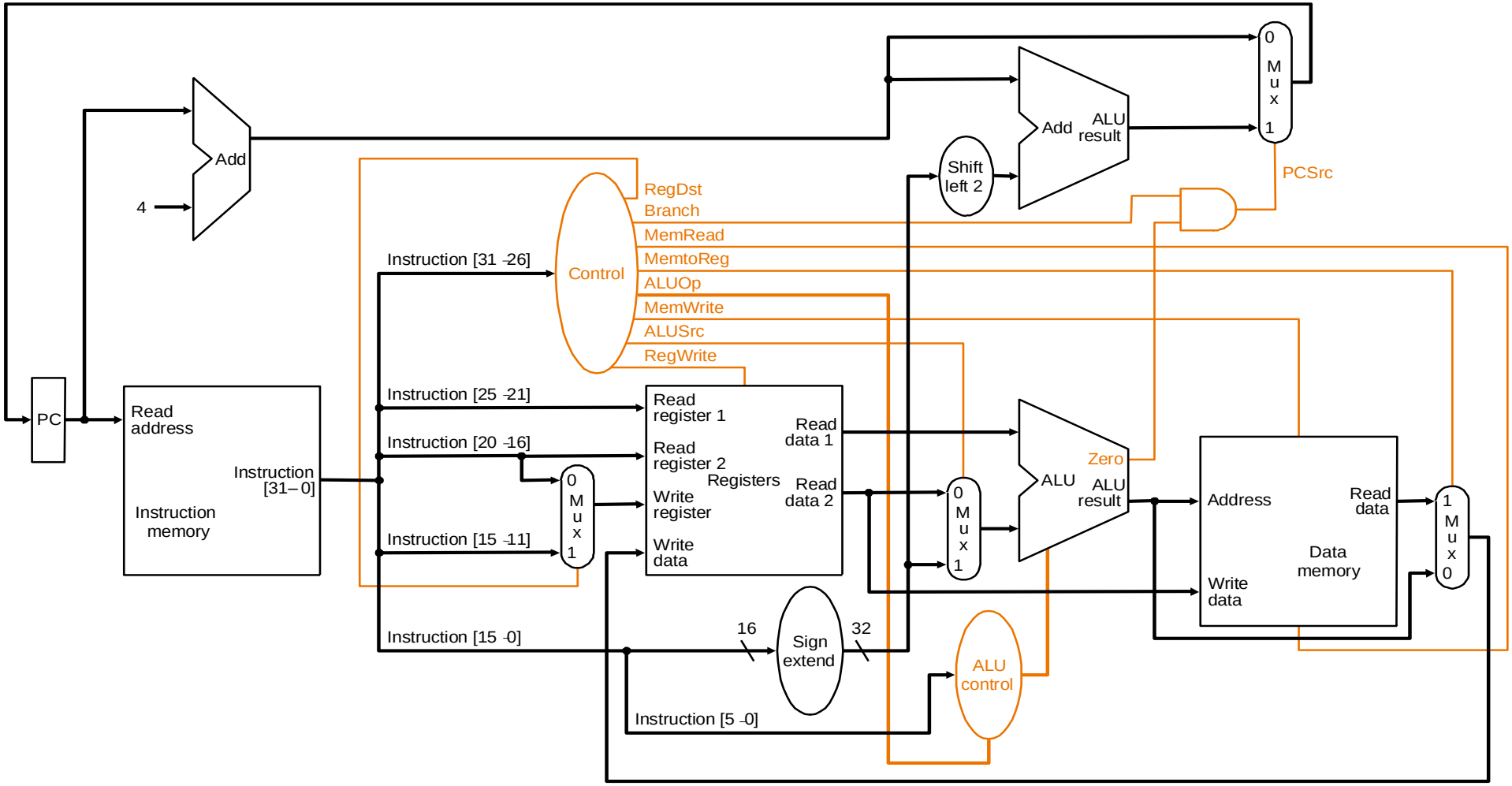
if (Zero==1) then PC=PC+4+signExt[imm16]*4; else PC = PC + 4



Outline

- Designing a processor
- Analyzing the instruction set
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations (step 4)
  - ALU controller (step 5a)
  - Main controller (step 5b)
- Adding *jump* instruction

Datapath with Control Unit

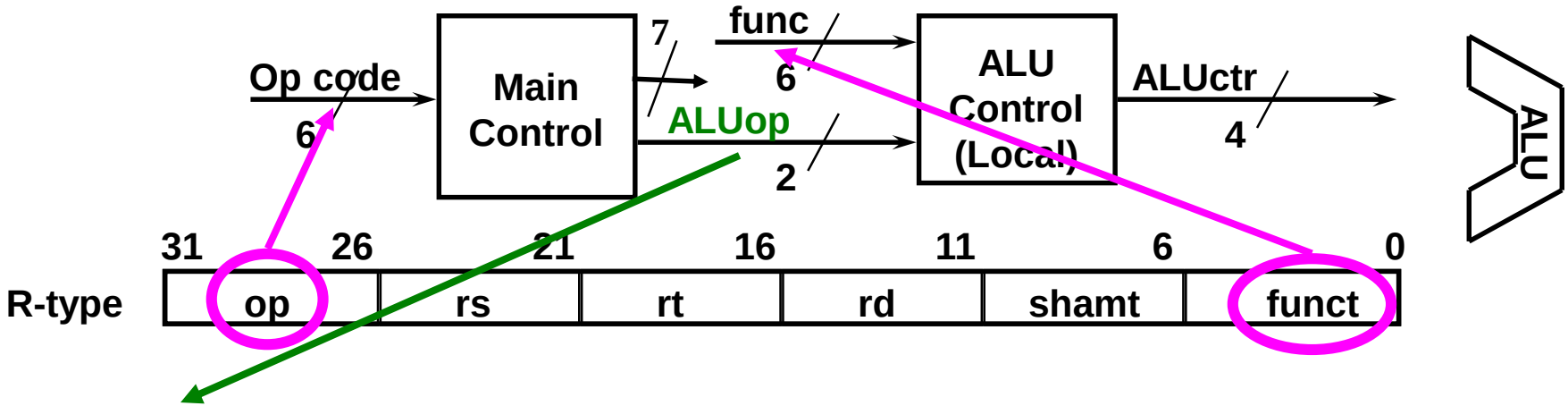


Step 5a: ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

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

Our Plan for the Controller



- **ALUop is 2-bit wide to represent:**
  - load/store requiring the ALU to perform add (00)
  - beq requiring the ALU to perform sub (01)
  - "R-type" need to reference `func` field (10)

	R-type	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Add	Add	Subtract	xxx
ALUop<1:0>	10	00	00	01	xxx

ALU Control

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

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

Logic Equation for ALUctr

ALUop		func						ALUctr			
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>
0	0	x	x	x	x	x	x	0	0	1	0
x	1	x	x	x	x	x	x	0	1	1	0
1	x	x	x	0	0	0	0	0	0	1	0
1	x	x	x	0	0	1	0	0	1	1	0
1	x	x	x	0	1	0	0	0	0	0	0
1	x	x	x	0	1	0	1	0	0	0	1
1	x	x	x	1	0	1	0	0	1	1	1

Logic Equation for ALUctr2

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
x	1	x	x	x	x	x	x	1
1	x	x	x	0	0	1	0	1
1	x	x	x	1	0	1	0	1

This makes func<3> a don't care

$$ALUctr2 = ALUop0 + ALUop1 \cdot func2' \cdot func1 \cdot func0'$$

Logic Equation for ALUctr1

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	x	x	x	x	x	x	1
x	1	x	x	x	x	x	x	1
1	x	x	x	0	0	0	0	1
1	x	x	x	0	0	1	0	1
1	x	x	x	1	0	1	0	1

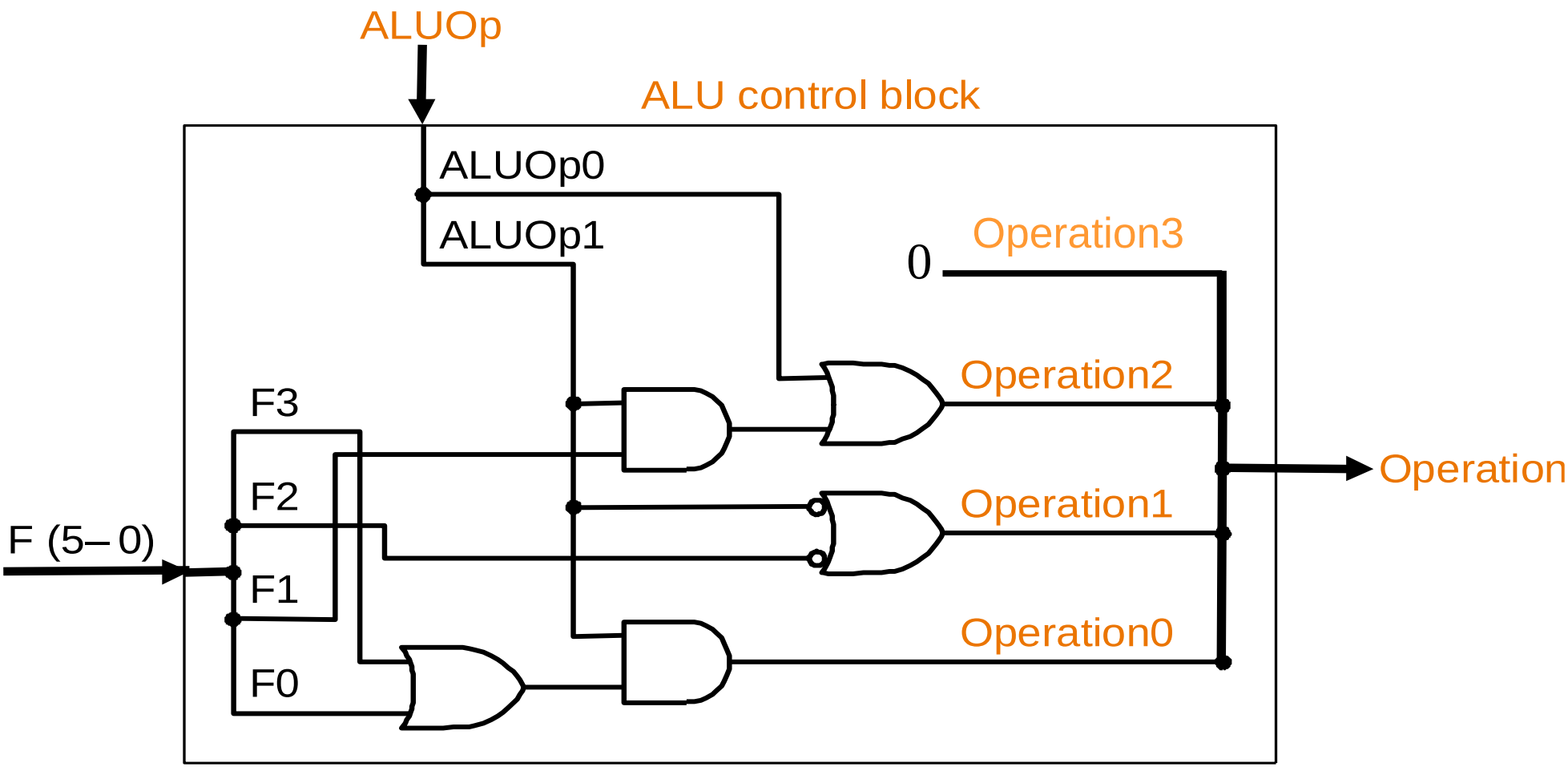
ALUctr1 = ALUop1'
+ ALUop1·func2'·func0'

Logic Equation for ALUctr0

ALUop		func						
bit<1>	bit<0>	bit<5>	bit<4>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
1	x	x	x	0	1	0	1	1
1	x	x	x	1	0	1	0	1

ALUctr0 = ALUop1
·func3'·func2·func1'·func0
+ ALUop1·func3
·func2'·func1·func0'

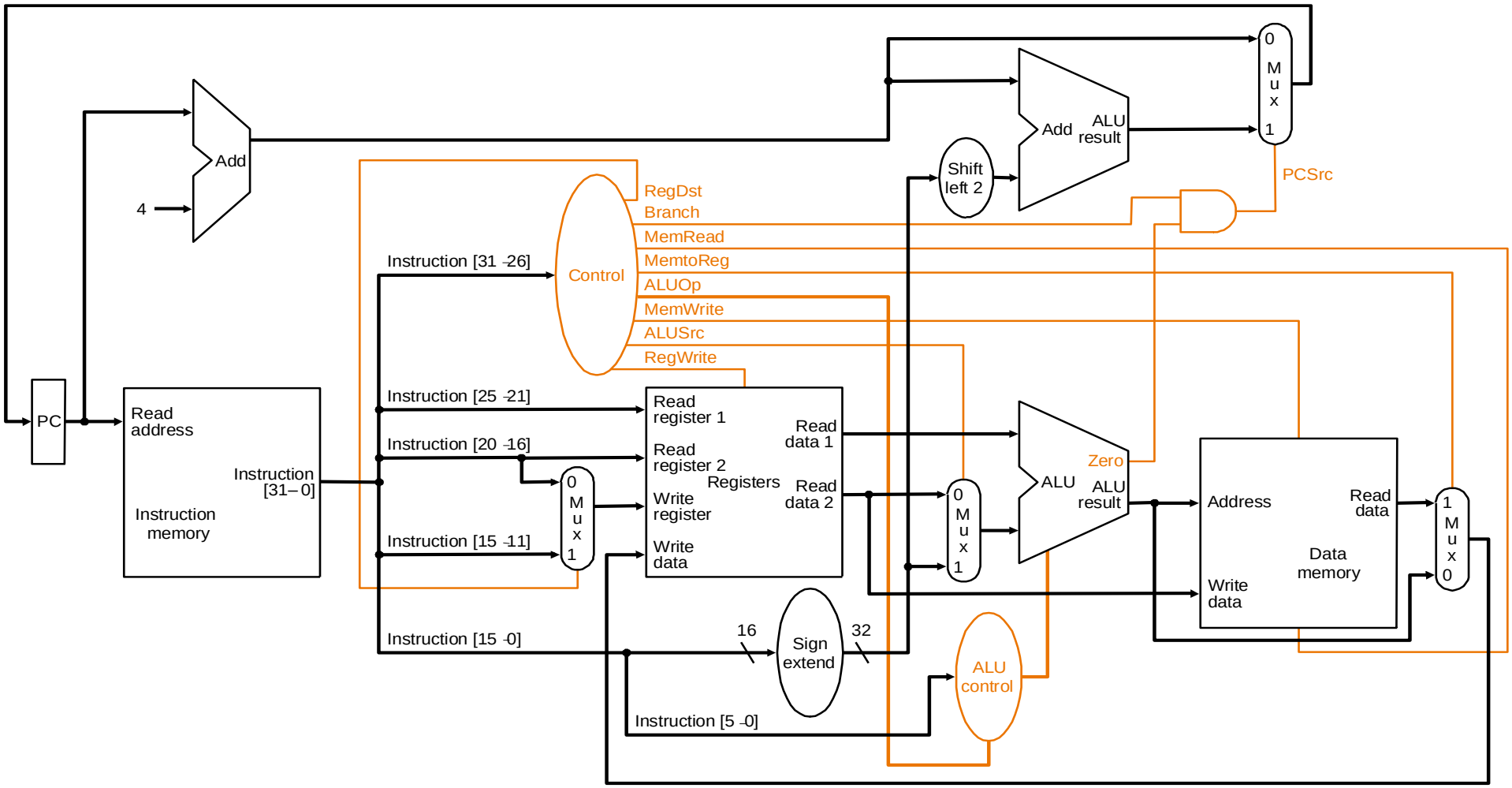
The Resultant ALU Control Block



Outline

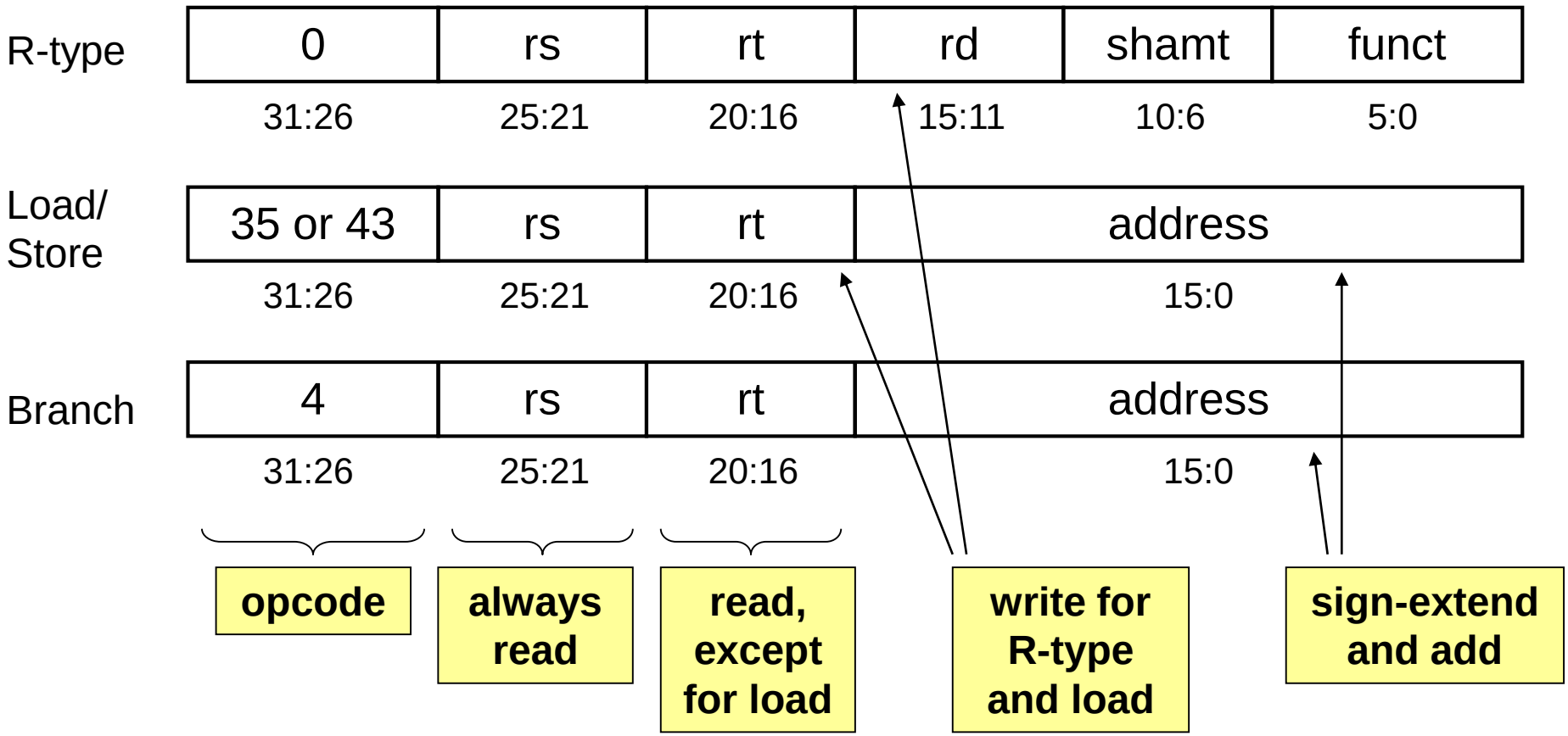
- Introduction to designing a processor
- Analyzing the instruction set
- Building the datapath
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller (step 5b)
- Adding *jump* instruction

Datapath with Control Unit



Step 5b: The Main Control Unit

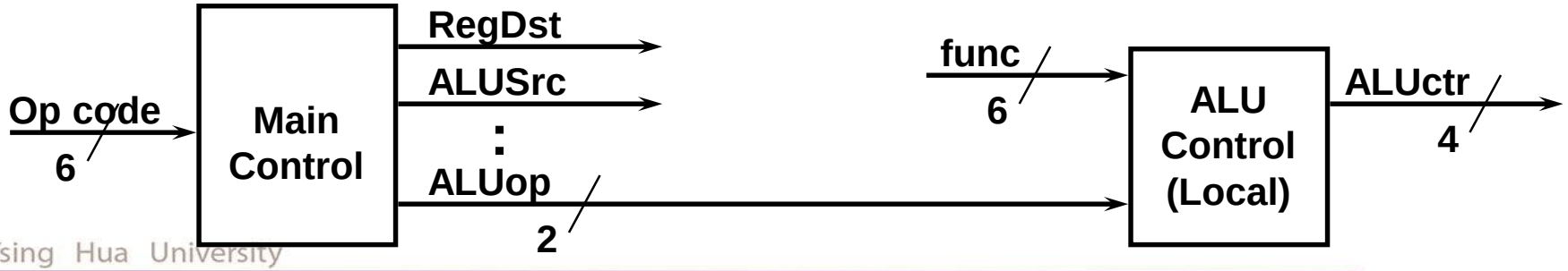
- **Control signals derived from instruction**



Truth Table of Control Signals (6 inputs and 9 outputs)

See Appendix A

	func	10 0000	10 0010	**We Don't Care :-)**		
	op	00 0000	00 0000	10 0011	10 1011	00 0100
		add	**sub**	**lw**	**sw**	**beq**
RegDst		1	1	0	x	x
ALUSrc		0	0	1	1	0
MemtoReg		0	0	1	x	x
RegWrite		1	1	1	0	0
MemRead		0	0	1	0	0
MemWrite		0	0	0	1	0
Branch		0	0	0	0	1
ALUop1		1	1	0	0	0
ALUop0		0	0	0	0	1



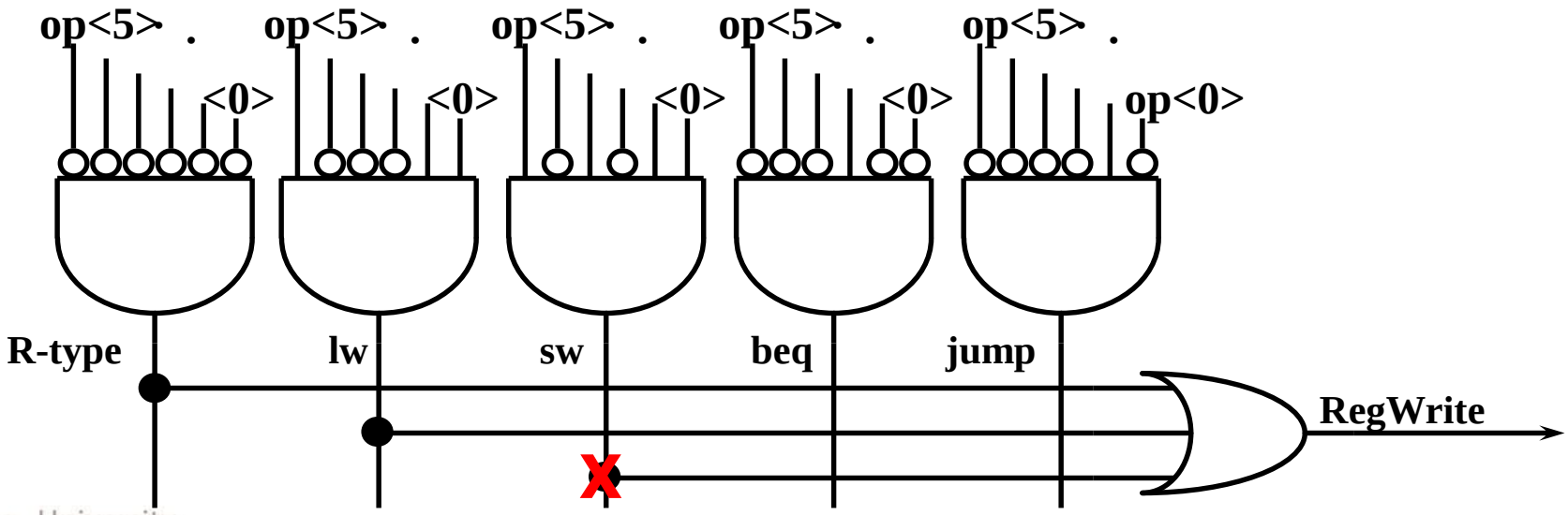
Truth Table for RegWrite

Op code	00 0000	10 0011	10 1011	00 0100
	R-type	lw	sw	beq
RegWrite	1	1	0	0

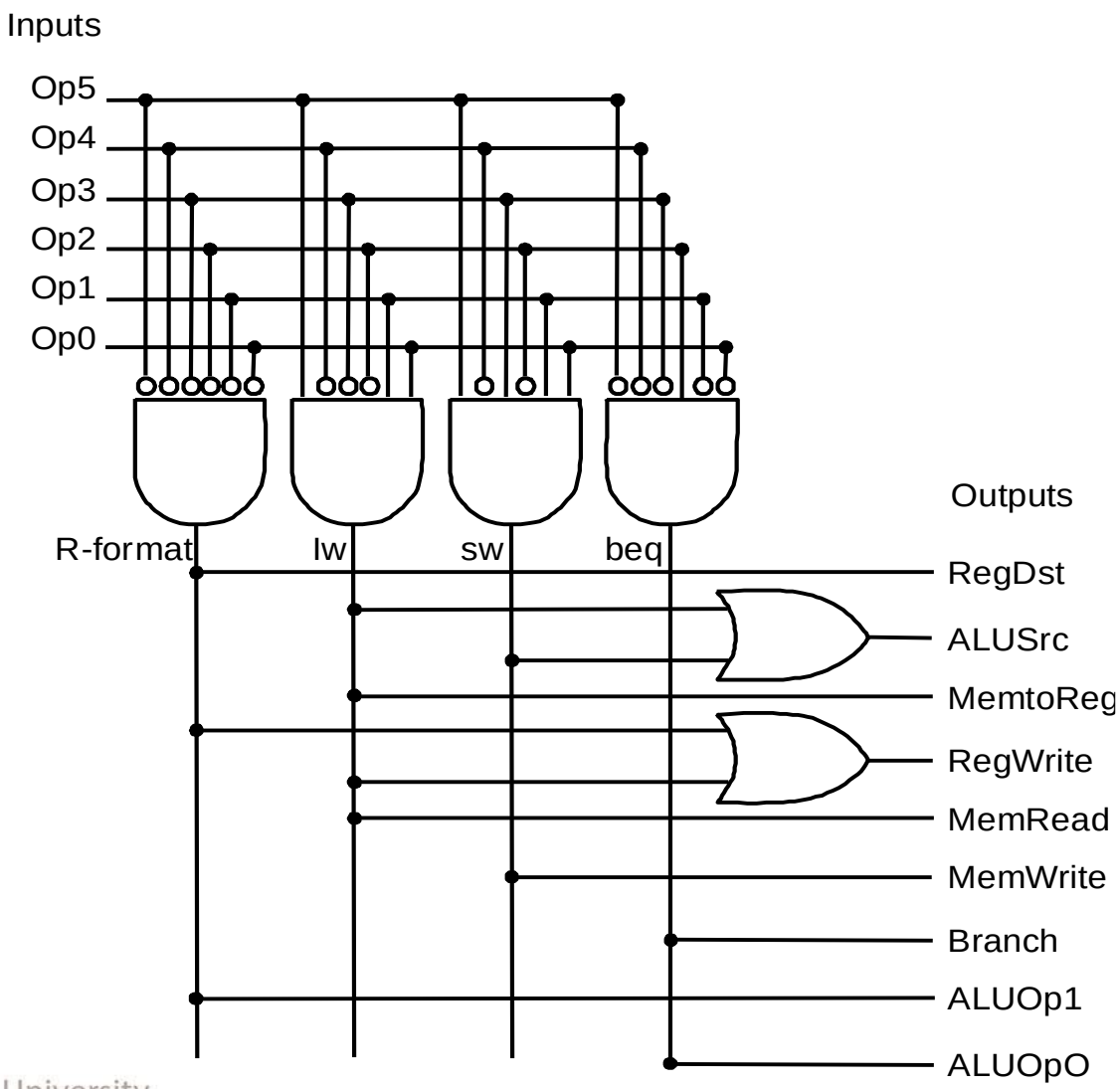
RegWrite = R-type + lw

= op5'·op4'·op3'·op2'·op1'·op0' (R-type)

+ op5·op4'·op3'·op2'·op1· op0 (lw)



PLA Implementing Main Control



Outline

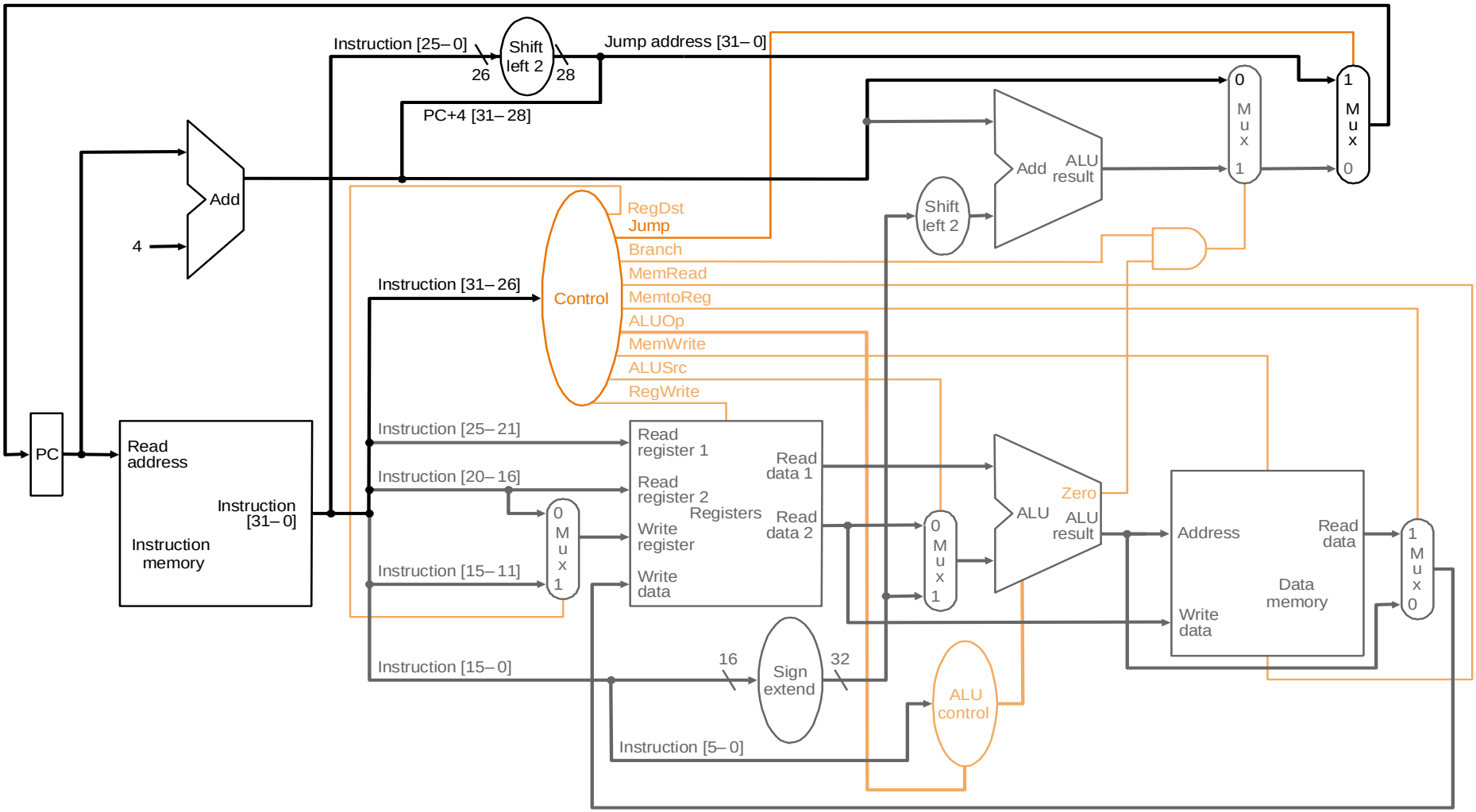
- Introduction to designing a processor
- Analyzing the instruction set (step 1)
- Building the datapath (steps 2, 3)
- A single-cycle implementation
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU controller
  - Main controller
- Adding *jump* instruction

Implementing Jumps

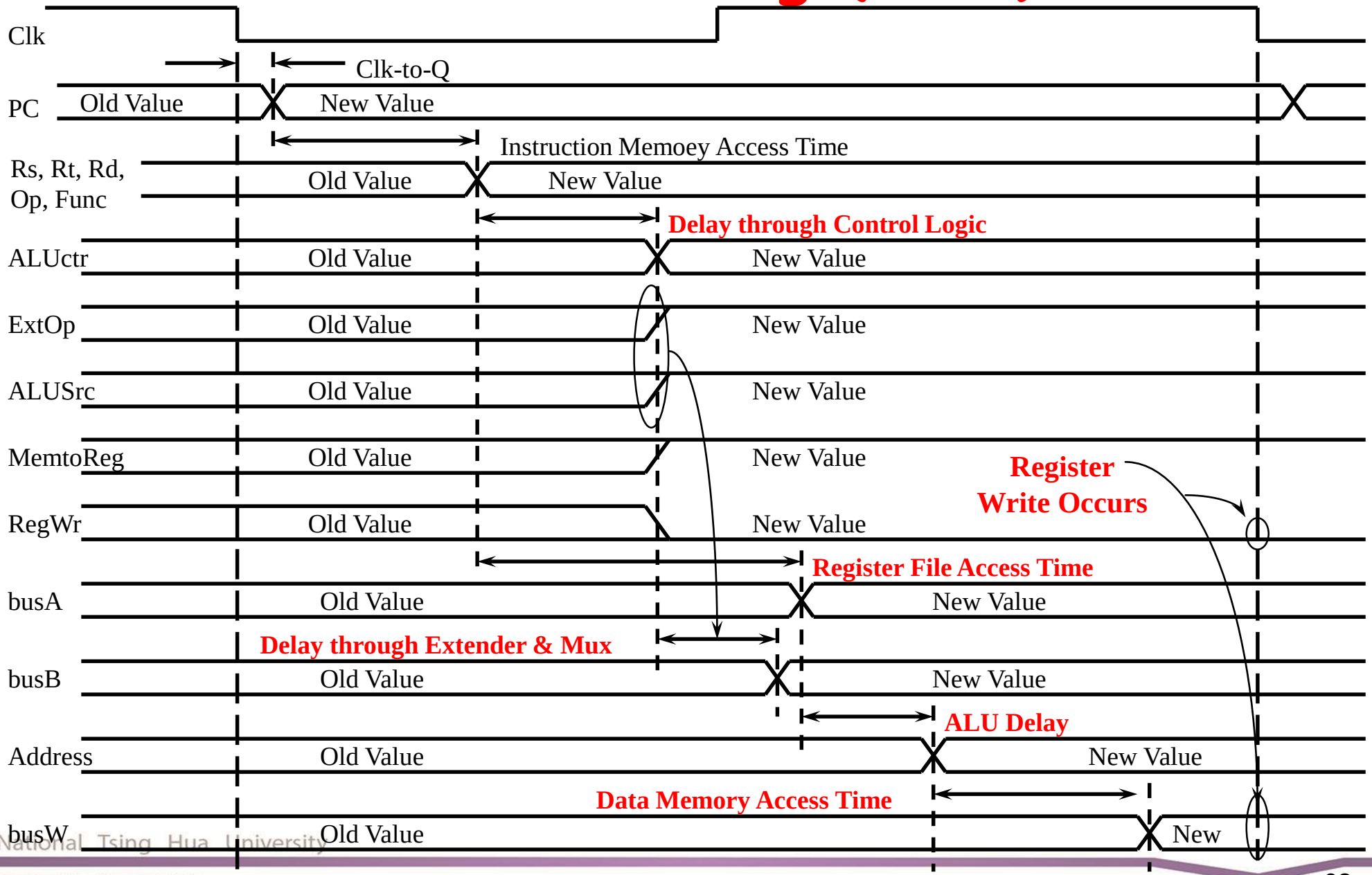
Jump	2	address
	31:26	25:0

- **Jump uses word address**
- **Update PC with concatenation of**
  - **Top 4 bits of old PC**
  - **26-bit jump address**
  - **00**
- **Need an extra control signal decoded from opcode**

Putting it Altogether (+ jump instruction)



Worst Case Timing (Load)



Drawback of Single-Cycle Design

- **Long cycle time:**
  - **Cycle time must be long enough for the load instruction:**

    PC's Clock -to-Q +
    Instruction Memory Access Time +
    Register File Access Time +
    ALU Delay (address calculation) +
    Data Memory Access Time +
    Register File Setup Time +
    Clock Skew
- **Cycle time for load is much longer than needed for all other instructions**

Summary

- Single cycle datapath => CPI=1, Clock cycle time long
- MIPS makes control easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates