

COMPUTER SYSTEMS ORGANIZATION

Acknowledgment: Almost all of these slides are based on Dave Patterson's CS152 Lecture Slides at UC, Berkeley

Single Cycle CPU Design -- Spring 2012 -- IIIT-H -- Suresh Purini

MIPS CPU Instructions

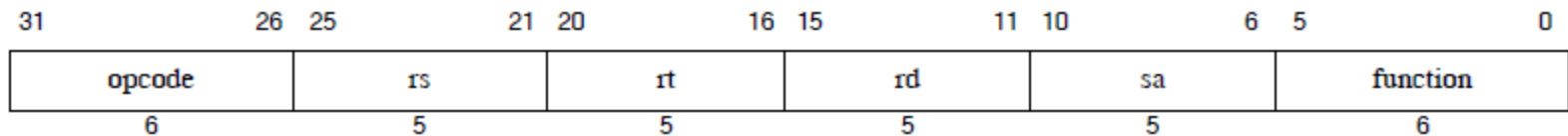


Three Types of CPU Instructions

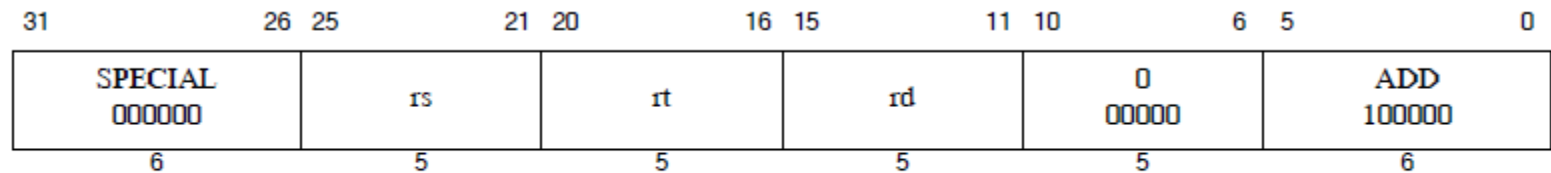
- R-type
- I-Type
- J-Type

R-Type Instructions

□ R-type Instruction Format



R-Type Instructions: ADD

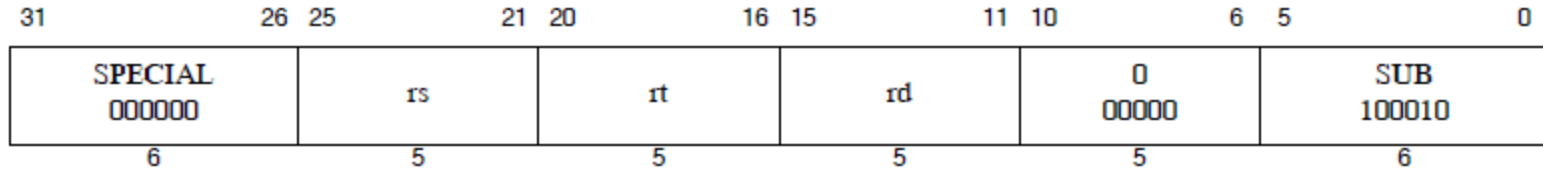


Format: ADD rd, rs, rt

MIPS32

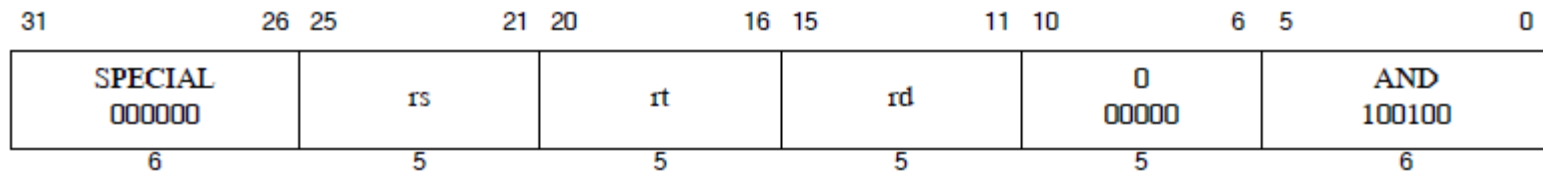
- **Format:** ADD rd, rs, rt
- $R[rd] = R[rs] + R[rt]$
- 32-bit 2's Complement Addition
- Destination register will not be modified if integer overflow exceptions occurs.

R-Type Instructions: SUB



- **Format:** SUB rd, rs, rt
- $R[rd] = R[rs] - R[rt]$
- 32-bit signed subtraction
- Destination register will not be modified if integer overflow exceptions occurs.

R-Type Instructions: AND



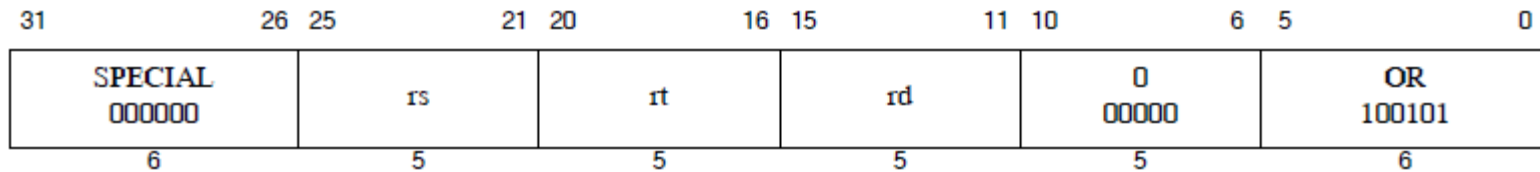
Format: AND rd, rs, rt

MIPS32

□ Format: AND rd, rs, rt

□ $R[rd] = R[rs] \& R[rt]$

R-Type Instructions: OR



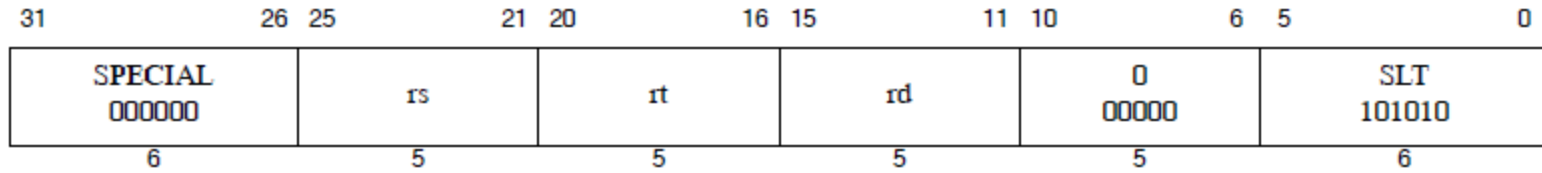
Format: OR rd, rs, rt

MIPS32

□ Format: OR rd, rs, rt

□ $R[rd] = R[rs] \mid R[rt]$

R-Type Instructions: SLT



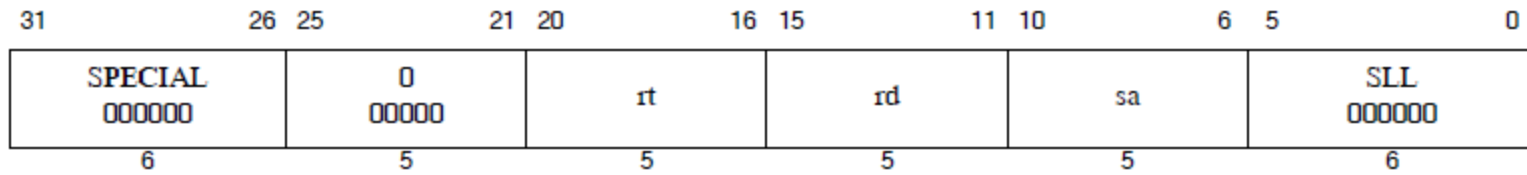
Format: SLT rd, rs, rt

MIPS32

- **Format:** SLT rd, rs, rt
- $R[rd] = R[rs] < R[rt] ? 1:0$
- Signed comparision

There are many other R-type instructions like ADDU, NOR, XOR etc.

R-Type Instructions: SLL



Format: SLL rd, rt, sa

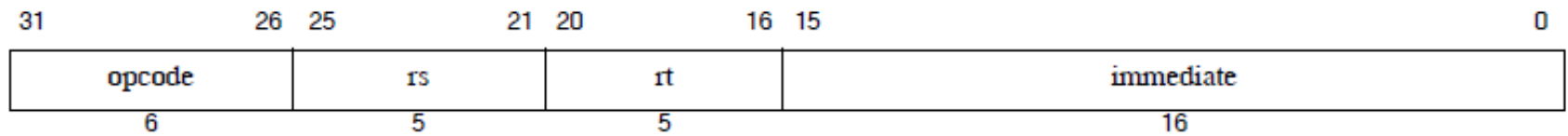
MIPS32

□ Format: SLL rd, rt, sa

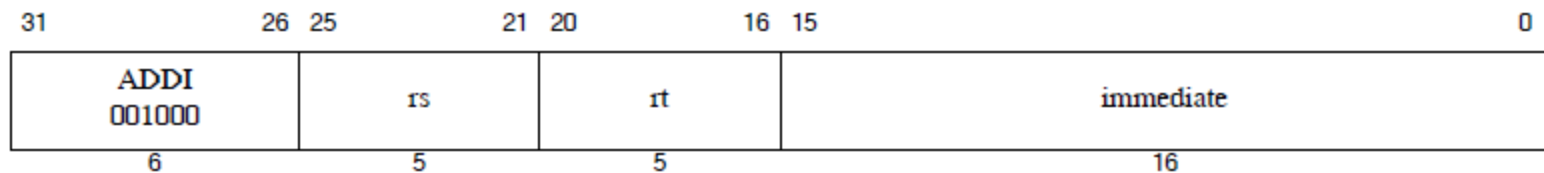
□ $R[rd] = R[rt] \ll sa$

Note: In our processor design we do not implement shift instructions.

I-type Instructions



I-type Instructions

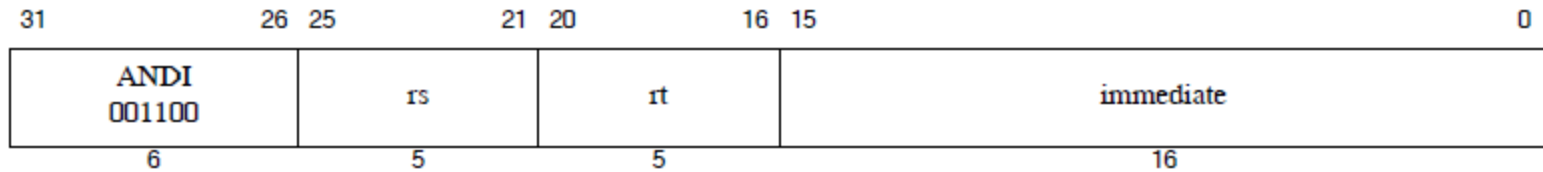


Format: `ADDI rt, rs, immediate`

MIPS32

- **Format:** `ADDI rt, rs, immediate`
- $R[rt] = R[rs] + \text{sign_extend}(\text{immediate})$
- immediate is 16-bit signed immediate
- 32-bit 2's complement addition
- Destination register will not be updated if integer overflow exception occurs

I-type Instructions

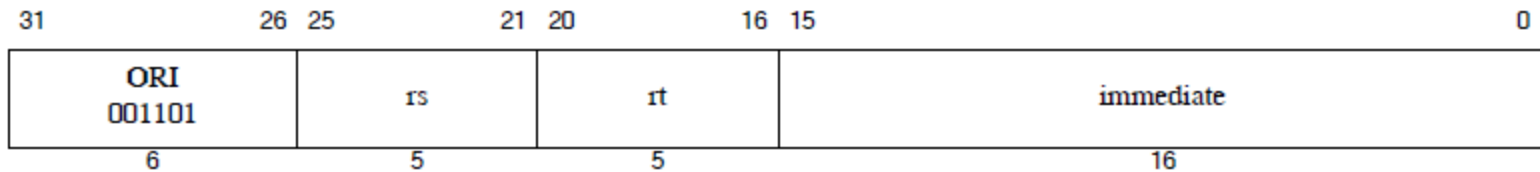


Format: ANDI rt, rs, immediate

MIPS32

- **Format:** ANDI rt, rs, immediate
- $R[rt] = R[rs] \& \text{zero_extend}(\text{immediate})$

I-type Instructions: ORI

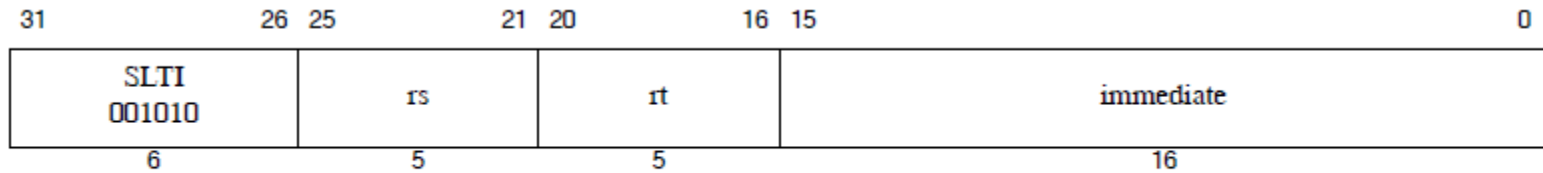


Format: ORI rt, rs, immediate

MIPS32

- **Format:** ORI rt, rs, immediate
- $R[rt] = R[rs] \mid \text{zero_extend}(\text{immediate})$

I-type Instructions: SLTI

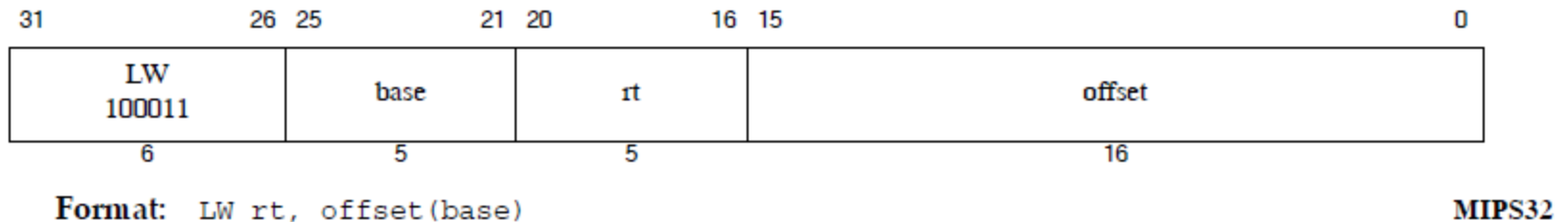


Format: SLTI rt, rs, immediate

MIPS32

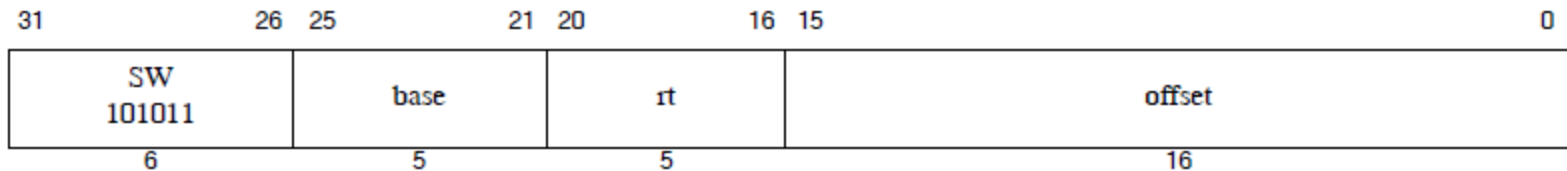
- **Format:** SLTI rt, rs, immediate
- $R[rt] = R[rs] < \text{sign_extend}(\text{immediate}) ? 1:0$

I-type Instructions: LW



- **Format:** LW rt, offset(base)
- $vaddr = \text{sign_extend}(\text{offset}) + R[\text{base}]$
- $R[\text{rt}] = \text{Mem}[vaddr]$
- If vaddr is now word-aligned, an exception will be raised.

I-type Instructions: SW

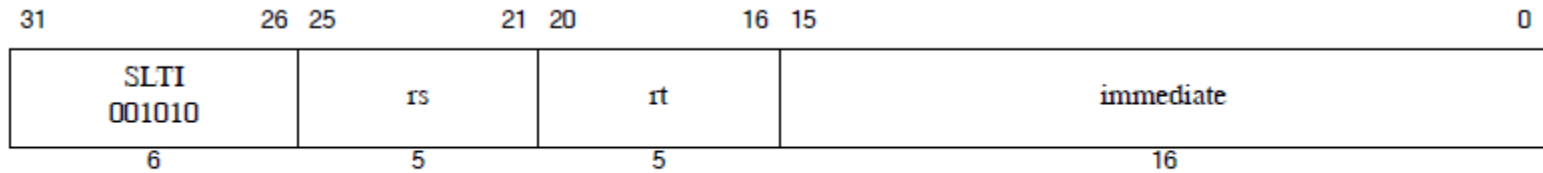


Format: SW rt, offset(base)

MIPS32

- **Format:** SW rt, offset(base)
- $vaddr = \text{sign_extend}(\text{offset}) + R[\text{base}]$
- $\text{Mem}[vaddr] = R[\text{rt}]$
- If vaddr is now word-aligned, an exception will be raised.

I-type Instructions: SLTI

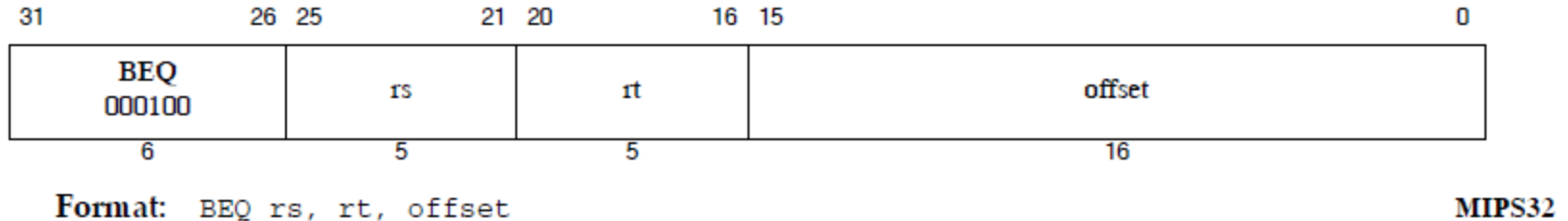


Format: SLTI rt, rs, immediate

MIPS32

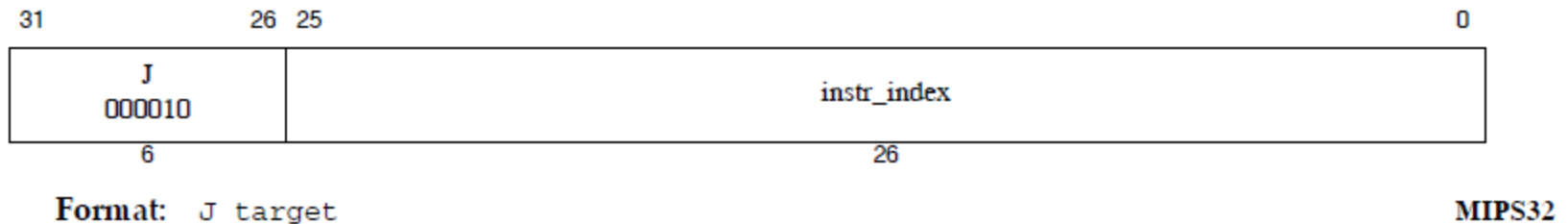
- **Format:** SLTI rt, rs, immediate
- $R[rt] = R[rs] < \text{sign_extend}(\text{immediate}) ? 1:0$

I-Type Instructions: BEQ



- **Format:** BEQ rs, rt, offset
- If $R[rs] == R[rt]$ then
 - ▣ $PC = \text{addr_of_branch} + 4 + \text{sign_extend}(\text{offset} \ll 2)$

MIPS J-Type Instructions



- **Format:** J target

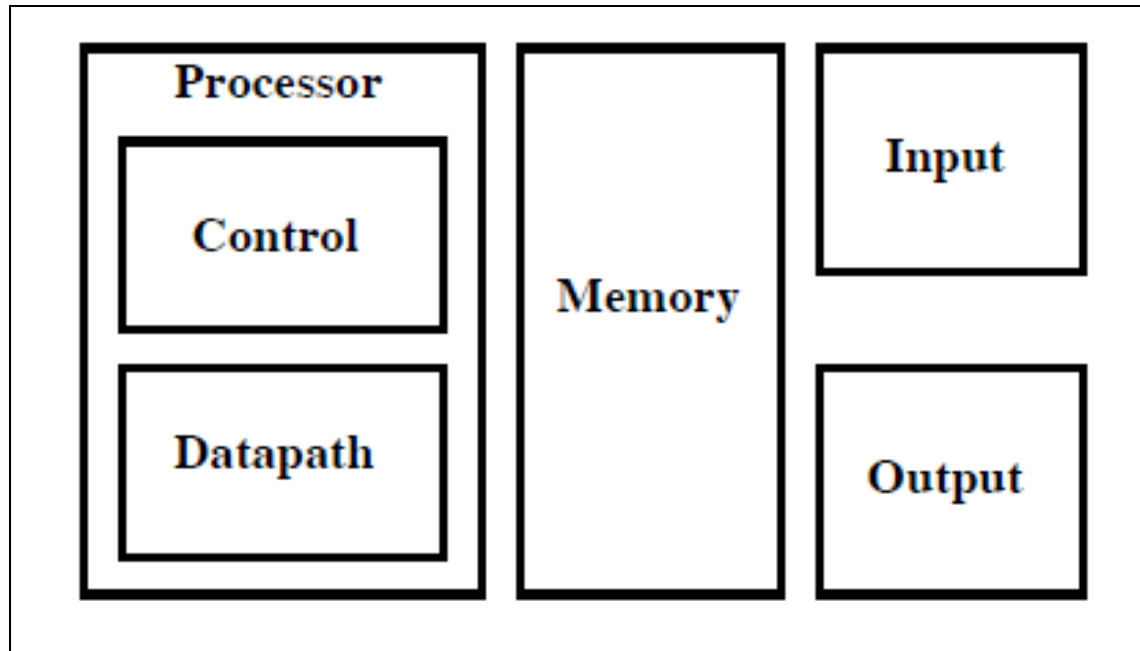
- **Target Address**

- ▣ **Lower 28 bits:** instr_index || 00

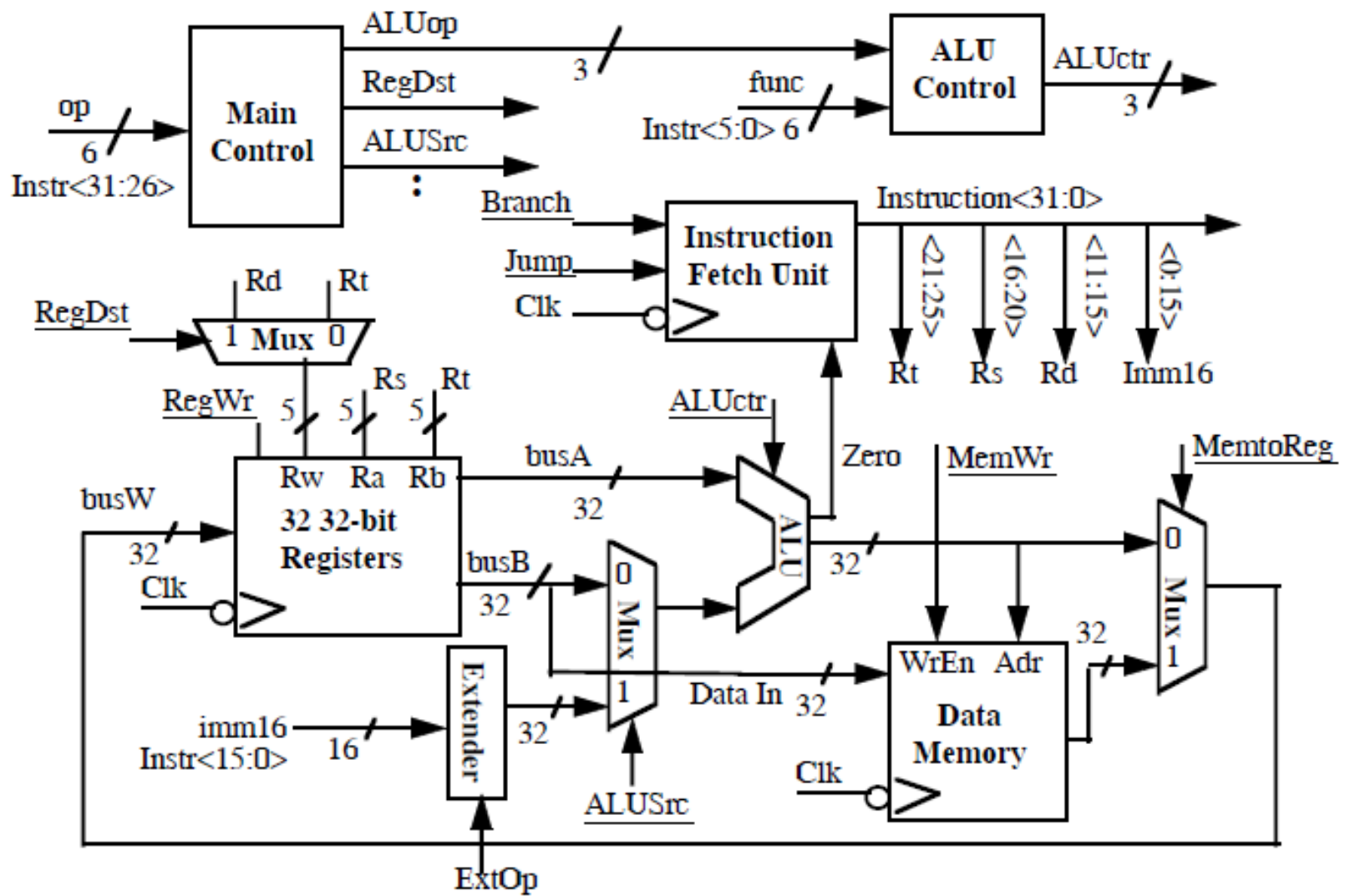
- ▣ **Upper Four Bits:** Bits 31, 30, 29, 28 of the address of the Jump Instruction.

The Big Picture: Where are We Now?

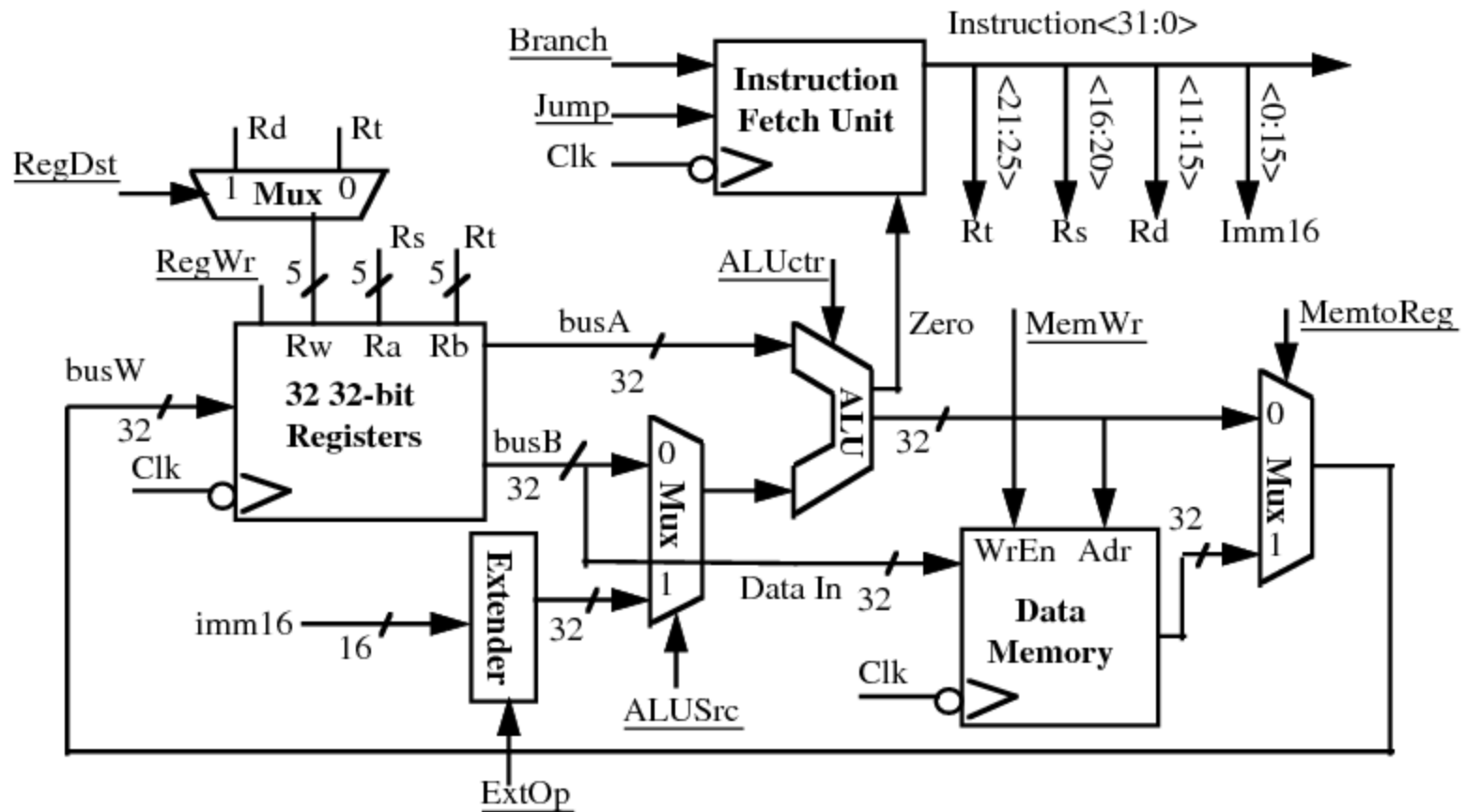
□ Five Classic Components of a Computer



MIPS Processor: Control Path + Data Path



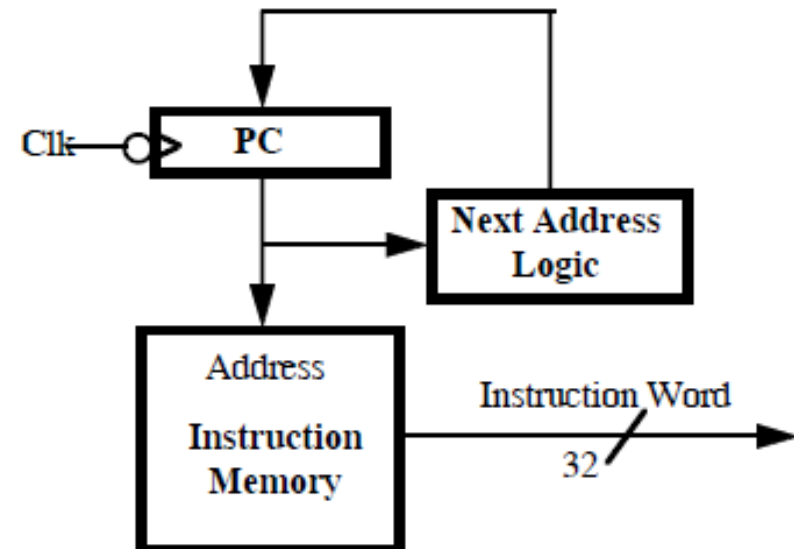
Data Path



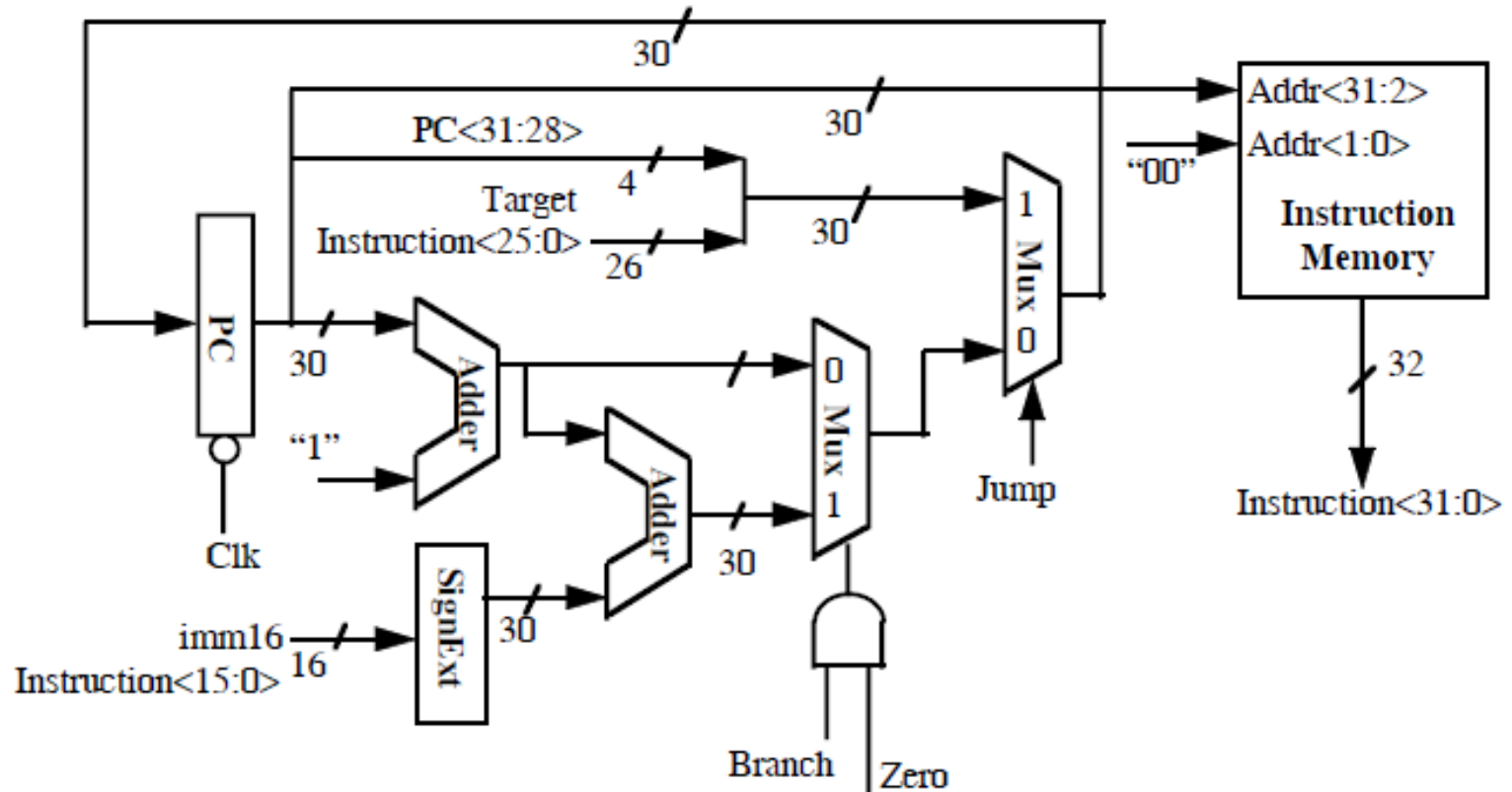
Overview of Instruction Fetch Unit

At a falling clock edge what happens:

- PC gets updated at the falling clock edge
- Fetch the Instruction from the address pointed to by PC
- Pass the PC through the next address logic
- Next value of the PC
 - ▣ Sequential Code
 - $\text{nextPC} = \text{PC} + 4$
 - ▣ Branch and Jump
 - $\text{nextPC} = \text{"something else"}$



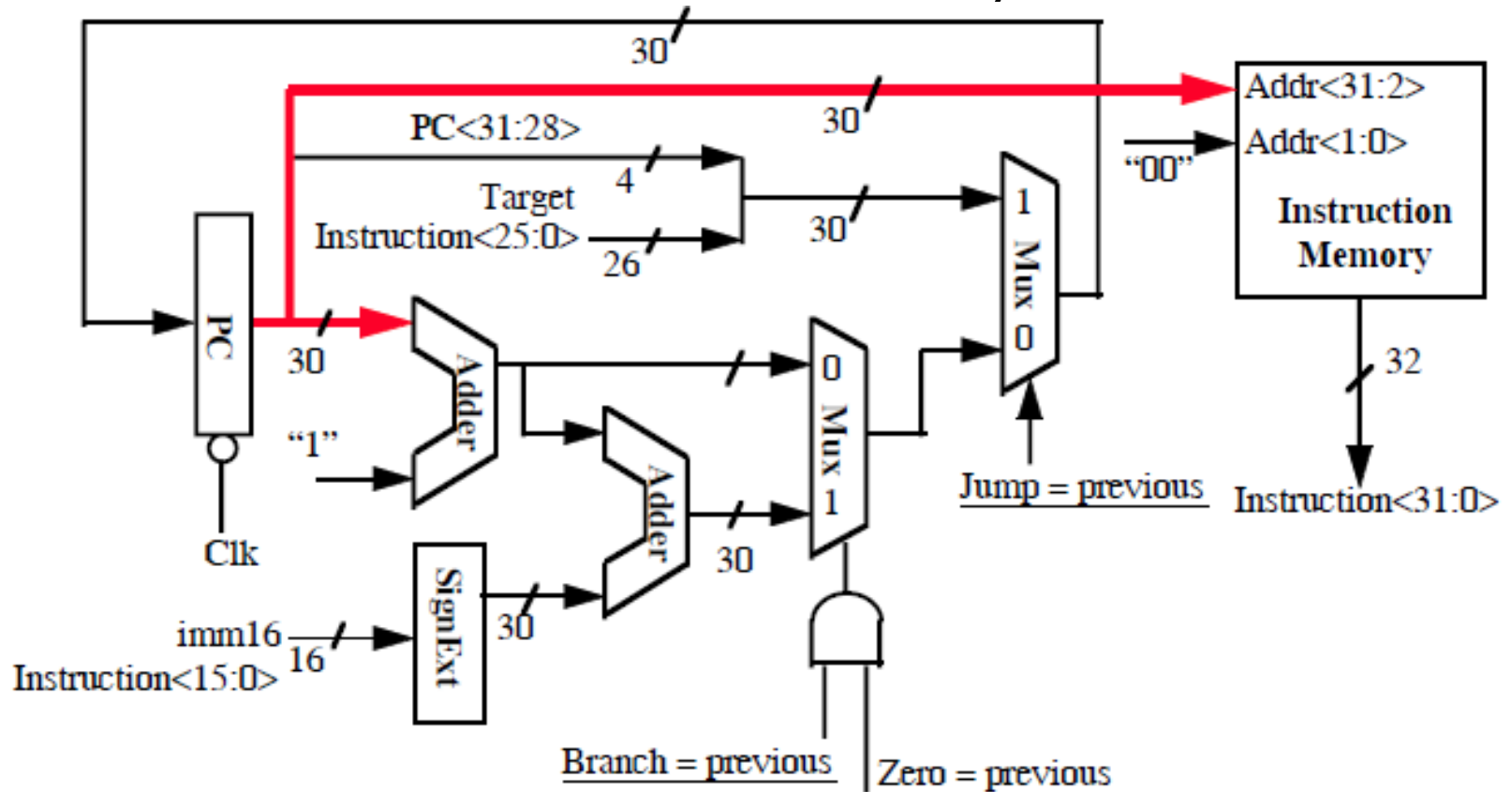
Instruction Fetch Unit



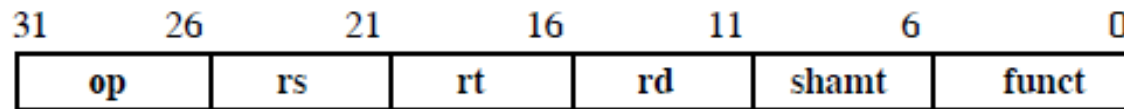
Instruction Fetch Unit at the Beginning of Add / Subtract

The following two steps are the same for all the instructions.

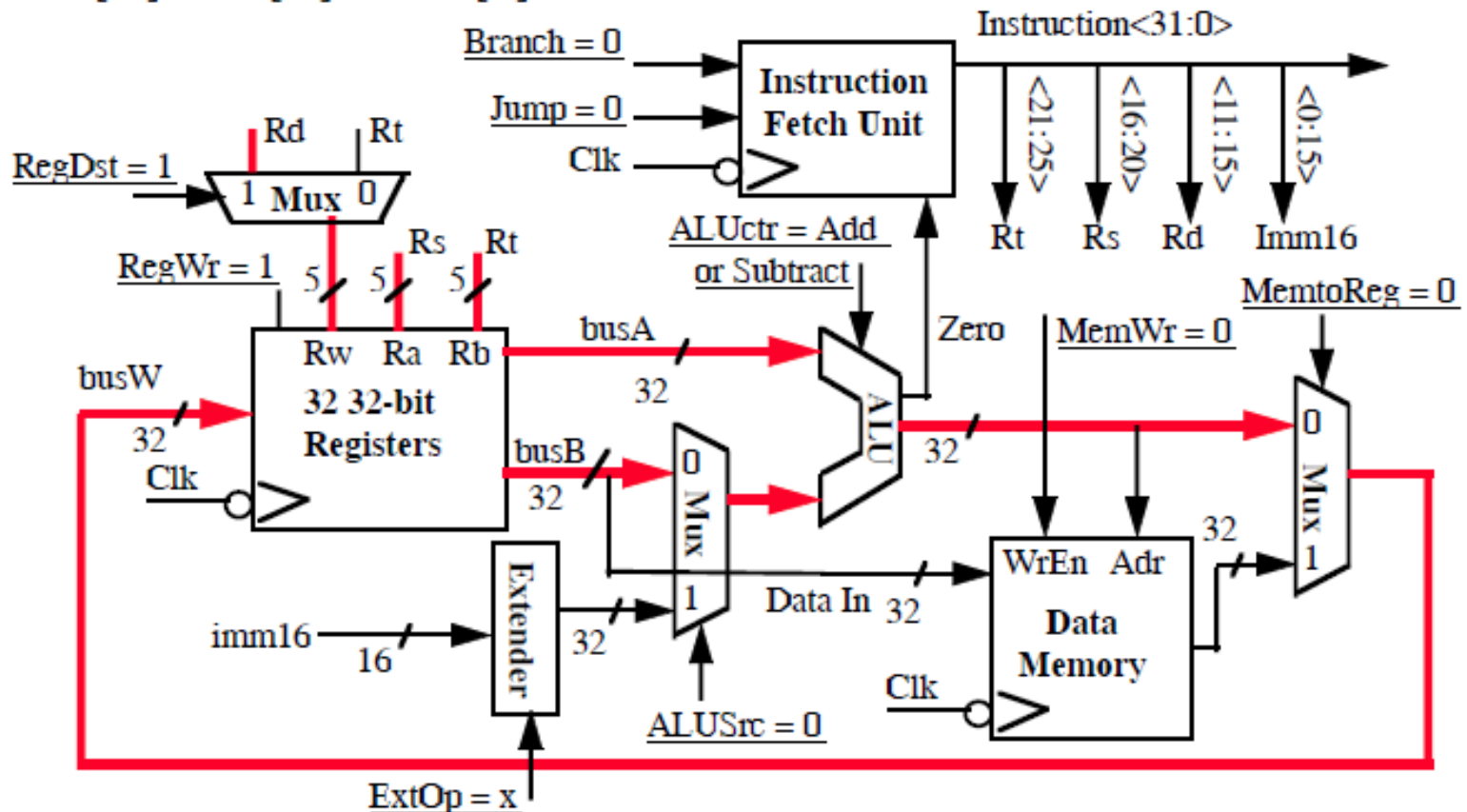
1. $PC = \text{nextPC}$
2. Fetch the instruction from Instruction memory: $\text{Instruction} = \text{mem}[PC]$



The Single Cycle Datapath during Add and Subtract

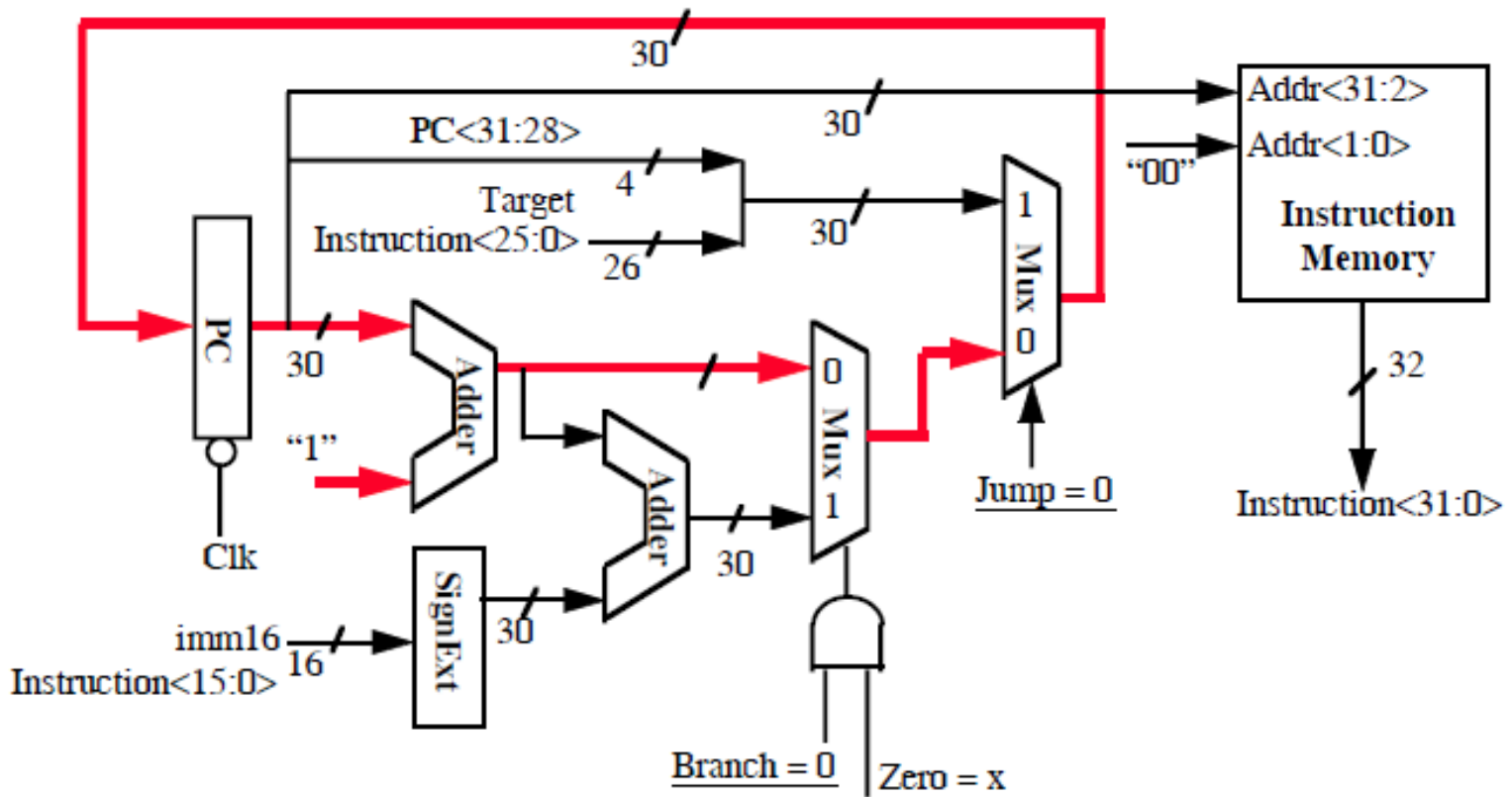


◦ $R[rd] \leftarrow R[rs] + / - R[rt]$

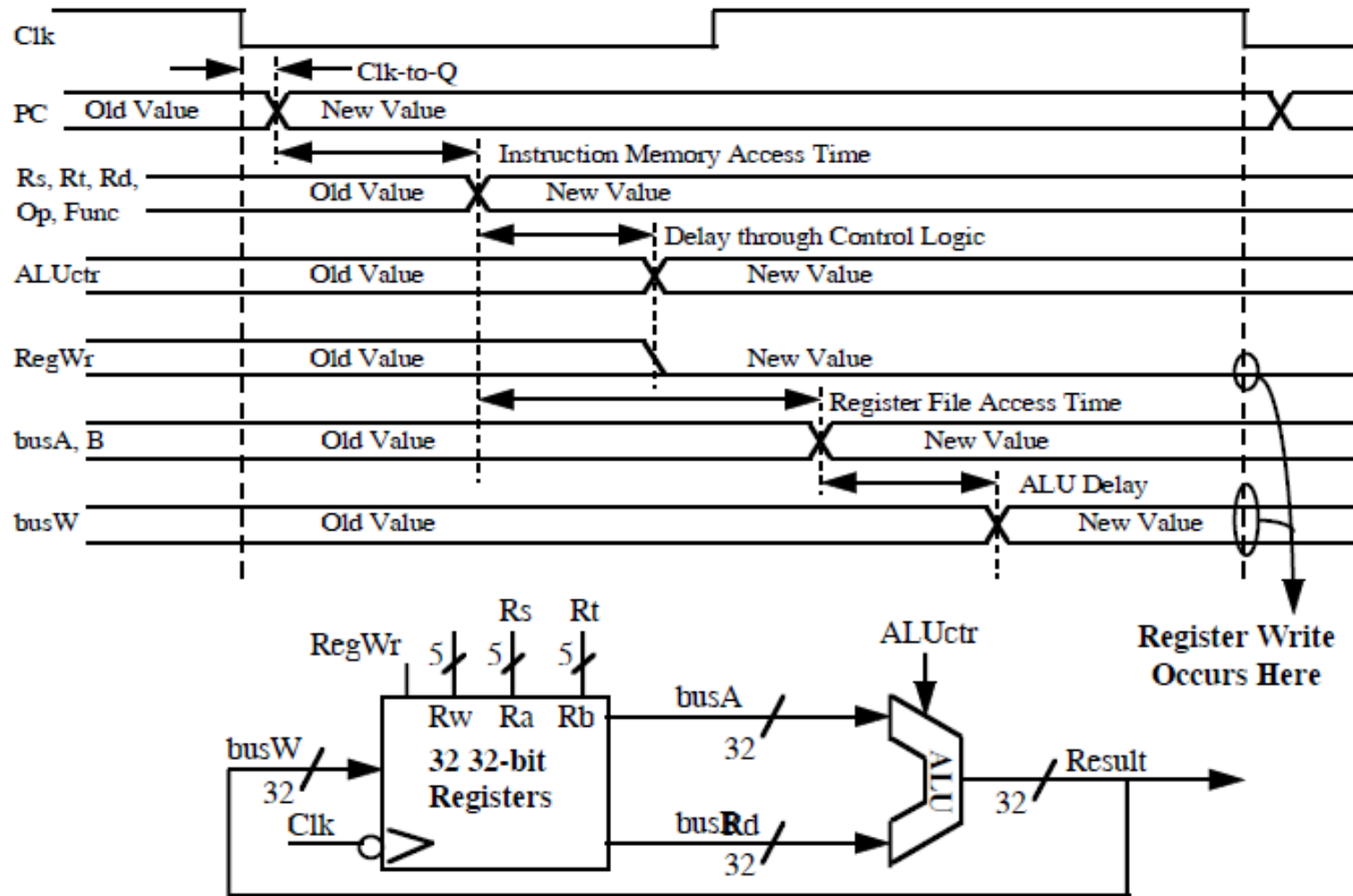


Instruction Fetch Unit at the End of Add and Subtract

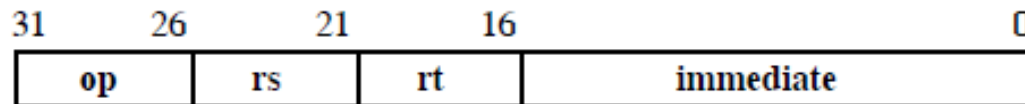
- $PC = PC + 4$
 - ▣ This is the same for all instructions except: Branch and Jump



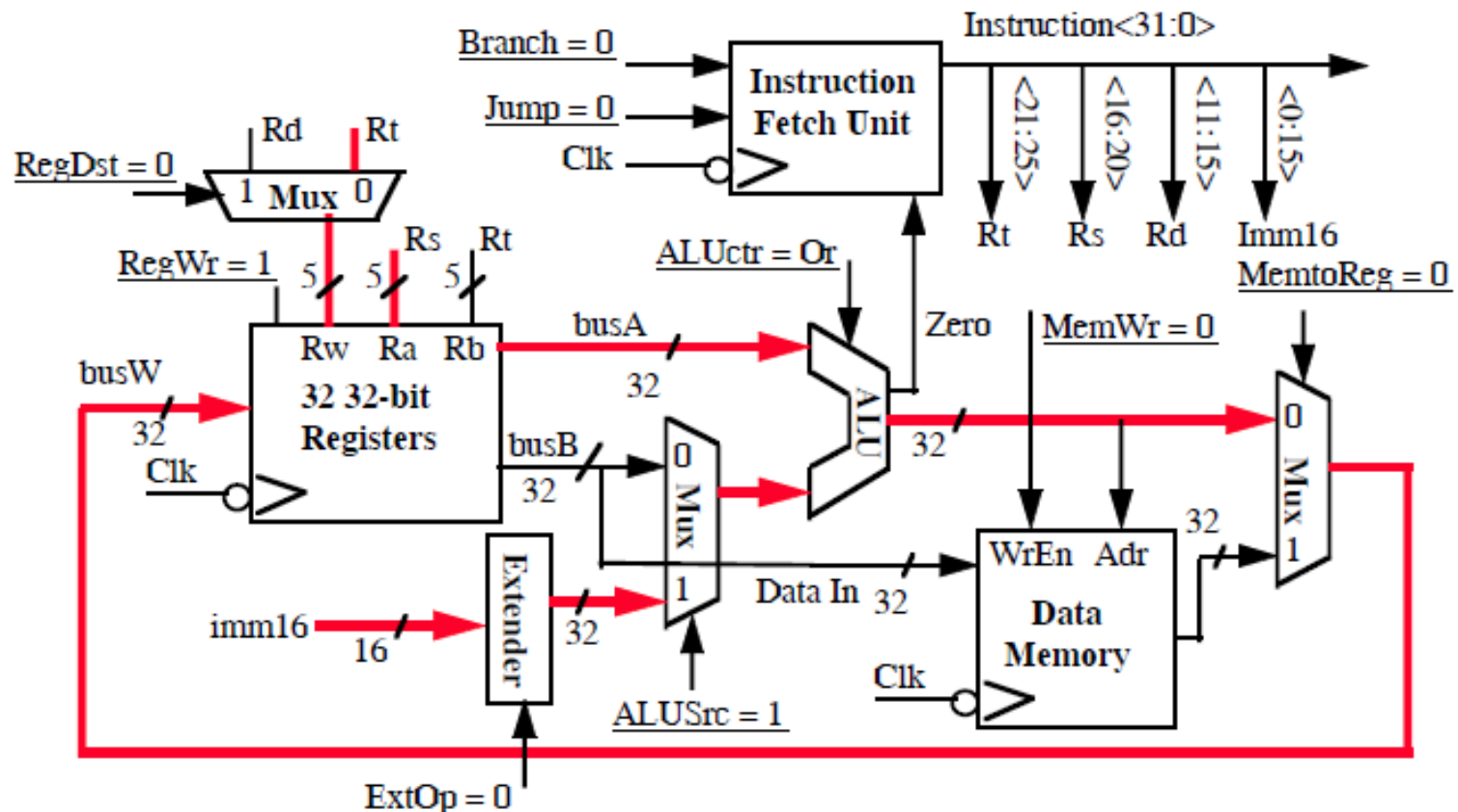
Register – Register Timing



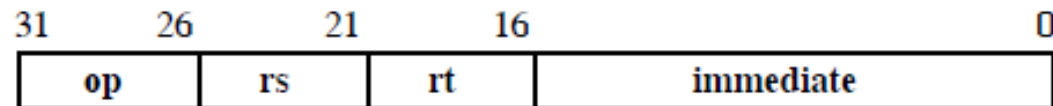
The Single Cycle Datapath during Or Immediate



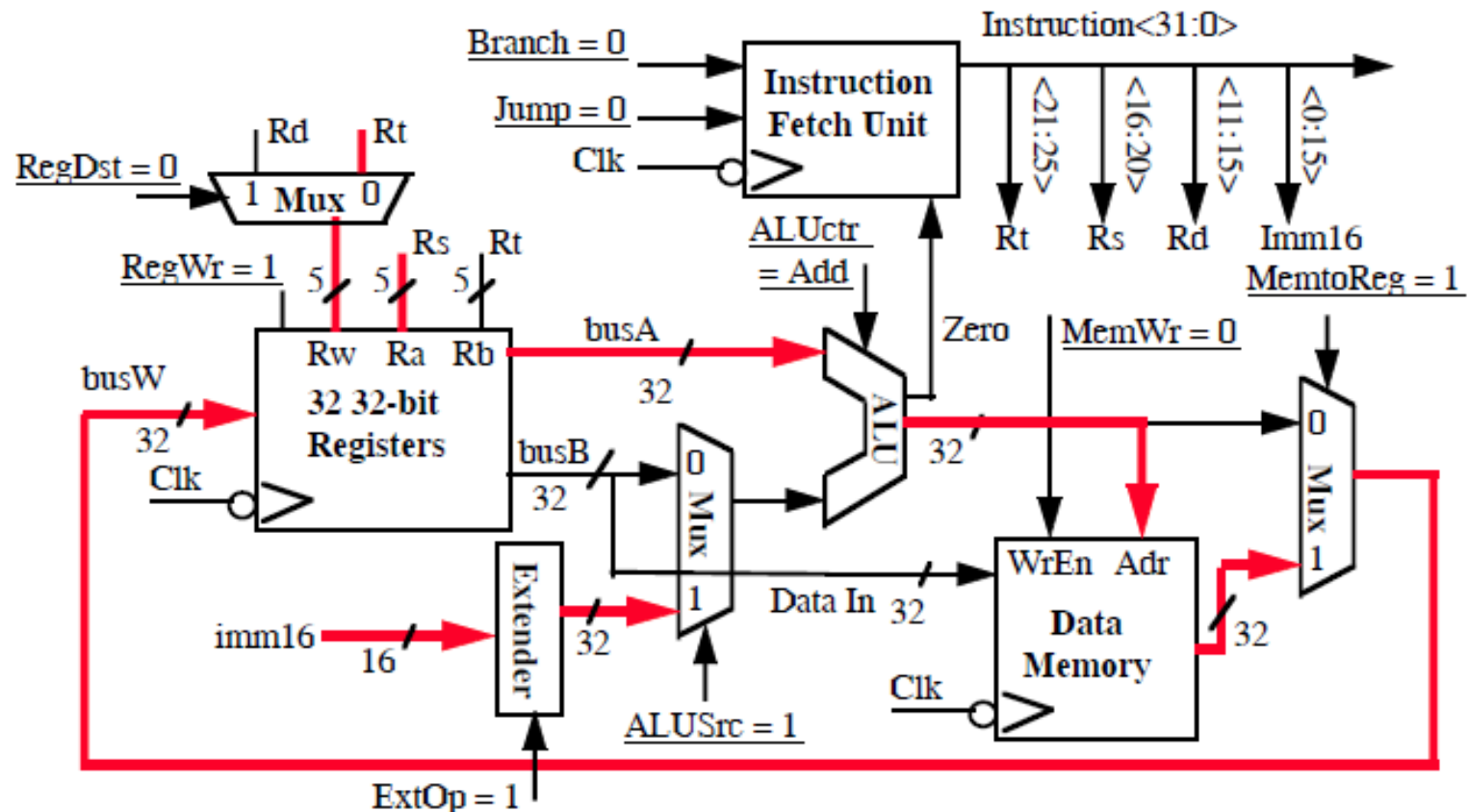
◦ $R[rt] \leftarrow R[rs] \text{ or } \text{ZeroExt}[Imm16]$



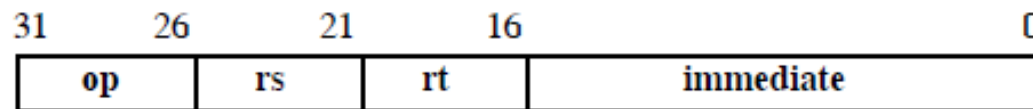
The Single Cycle Datapath during Load



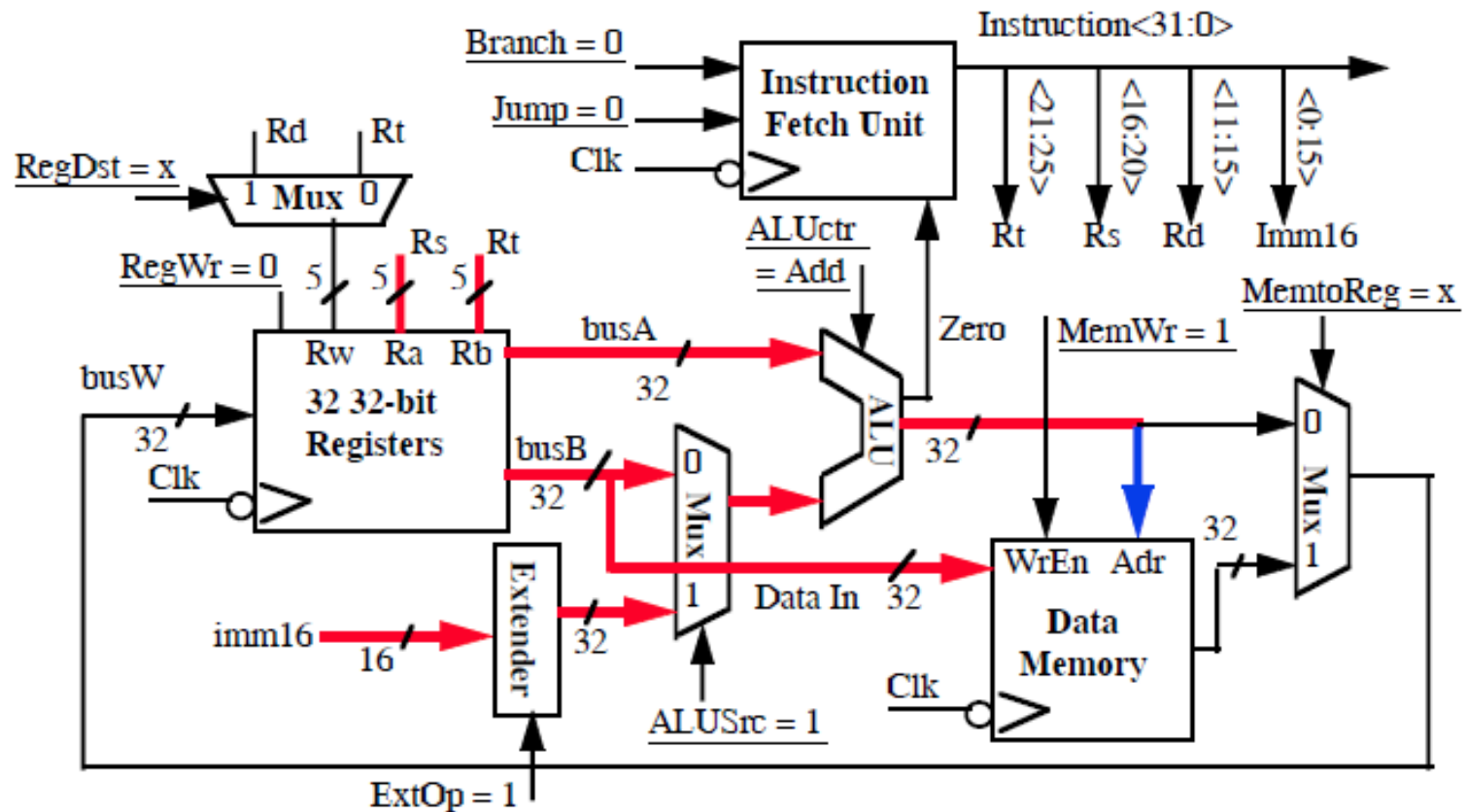
◦ $R[rt] \leftarrow \text{Data Memory } \{R[rs] + \text{SignExt}[imm16]\}$



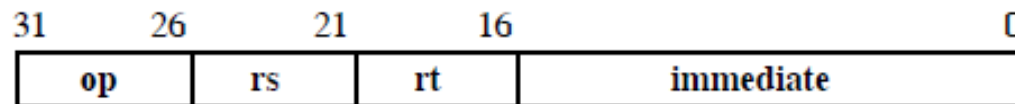
The Single Cycle Datapath during Store



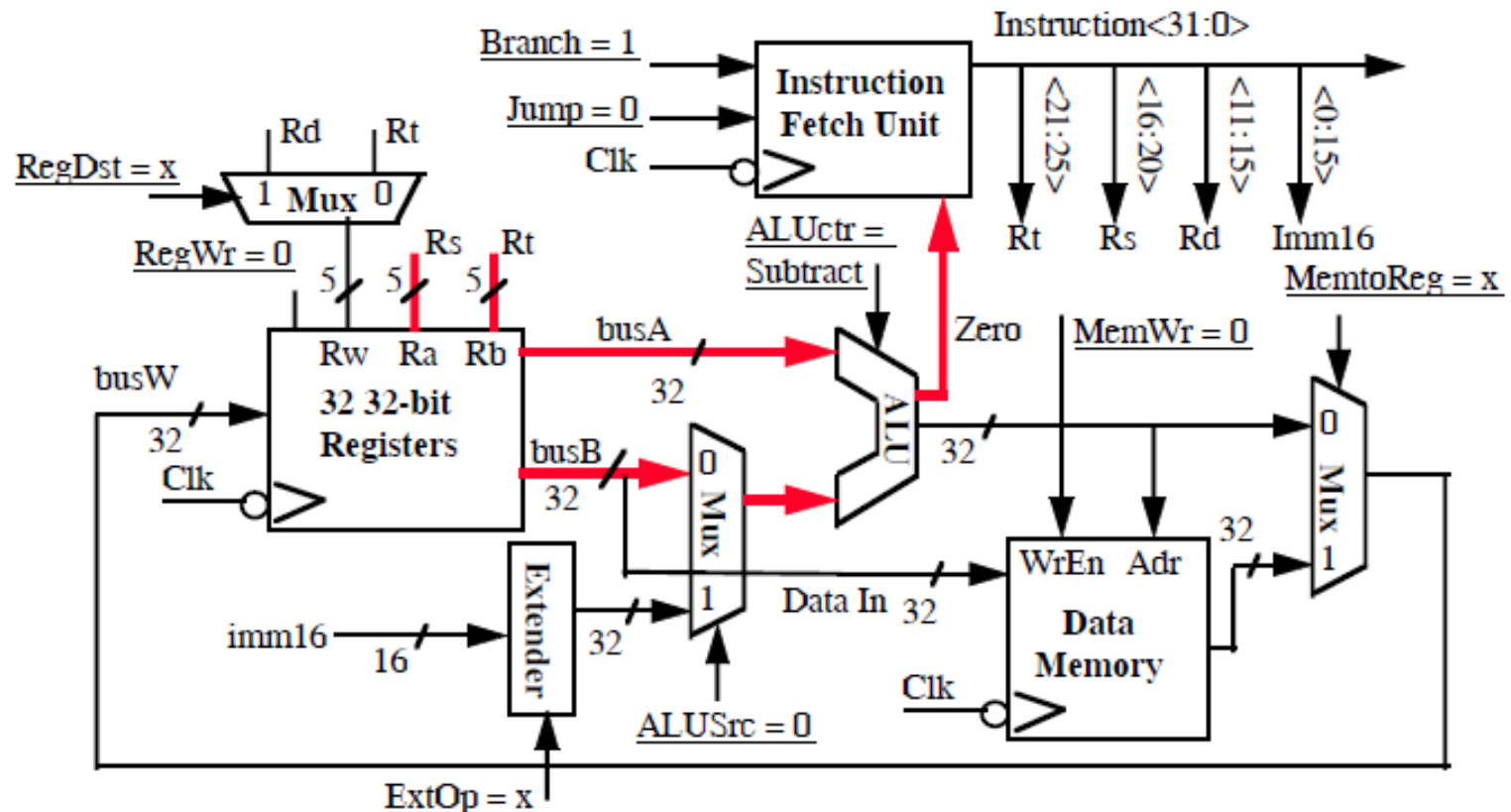
◦ Data Memory $\{R[rs] + \text{SignExt}[imm16]\} \leftarrow R[rt]$



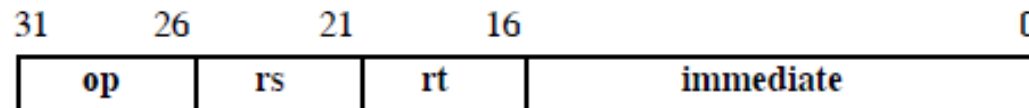
The Single Cycle Datapath during Branch



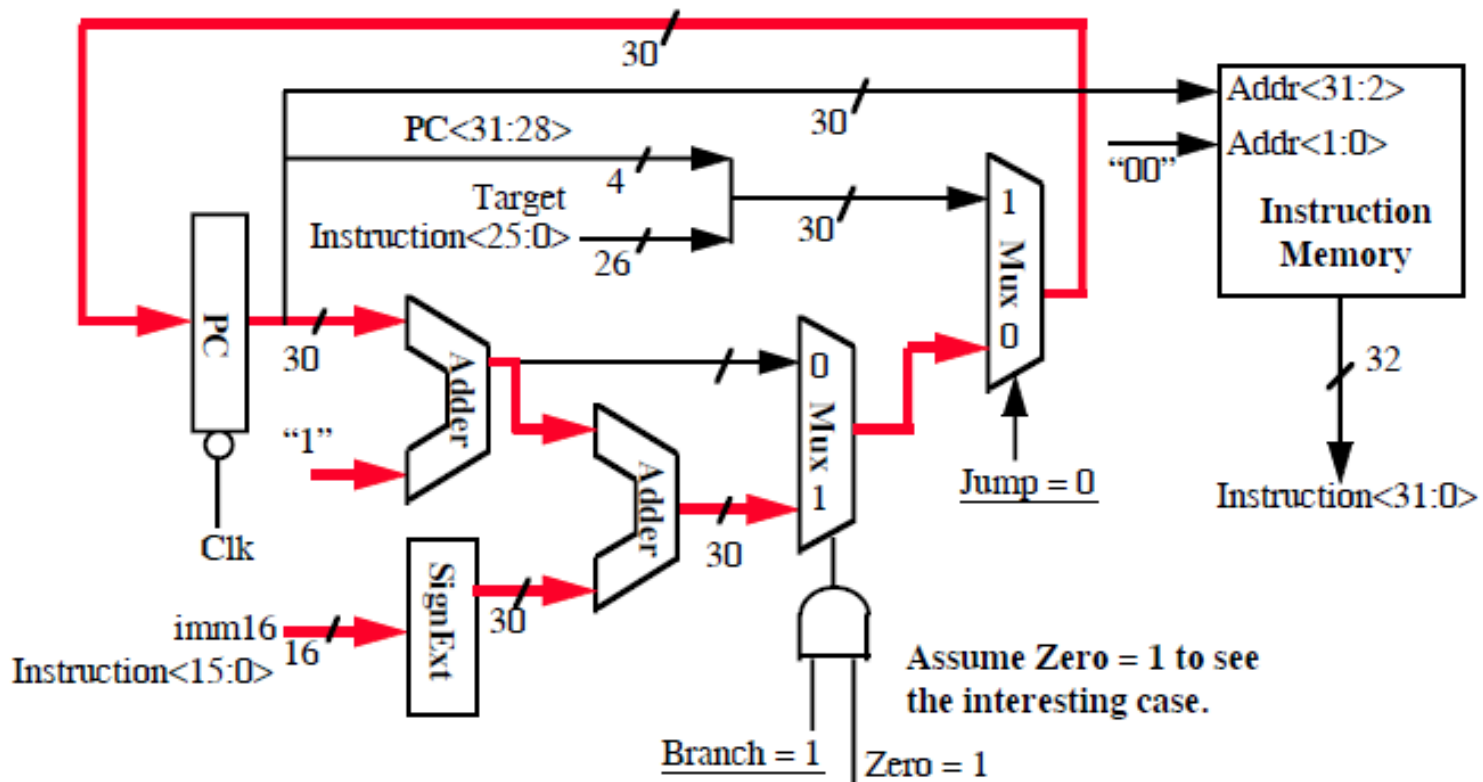
◦ if ($R[rs] - R[rt] == 0$) then Zero $\leftarrow 1$; else Zero $\leftarrow 0$



Instruction Fetch Unit at the End of Branch

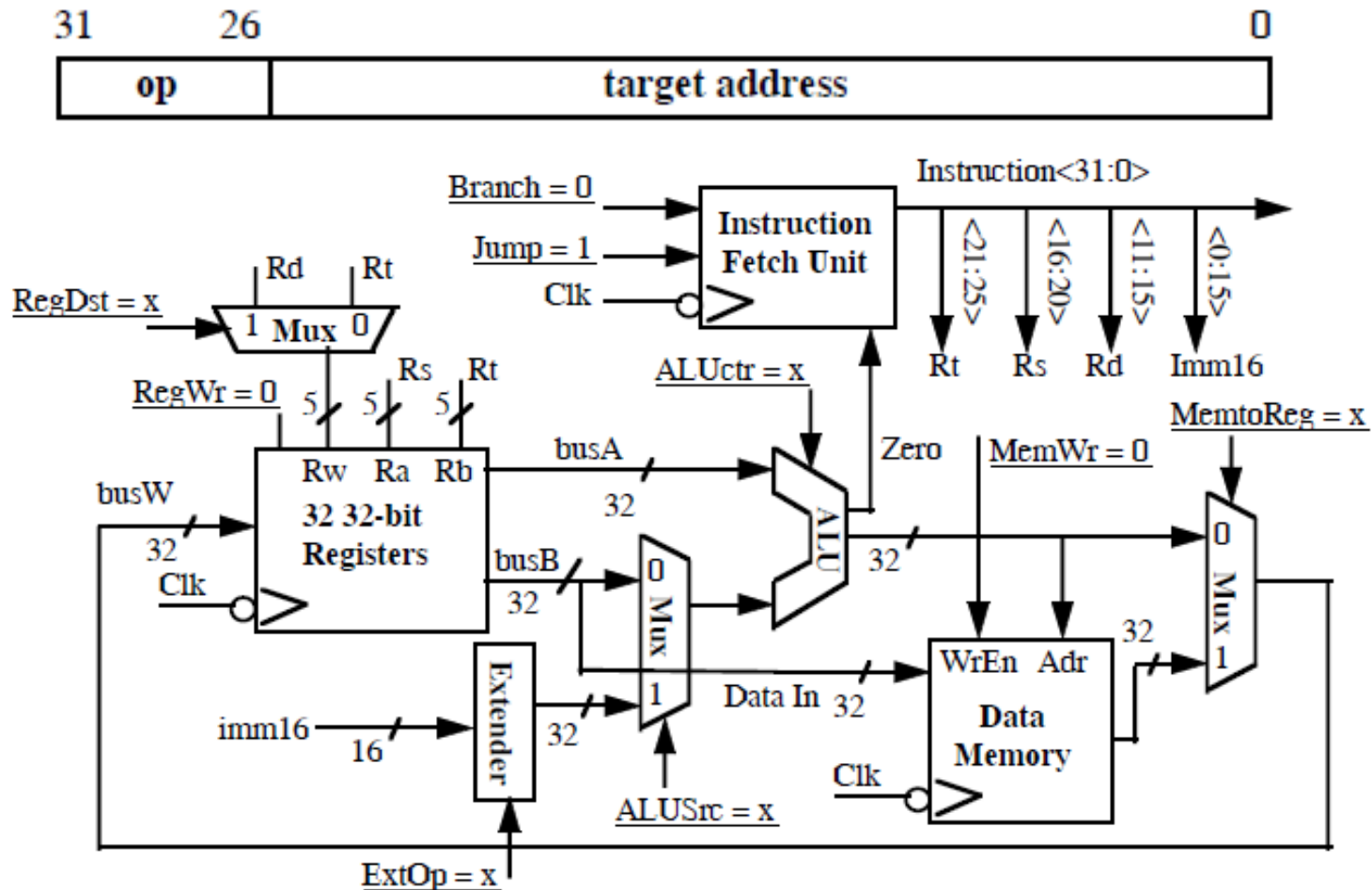


° if (Zero == 1) then $PC = PC + 4 + \text{SignExt}[\text{imm16}] * 4$; else $PC = PC + 4$

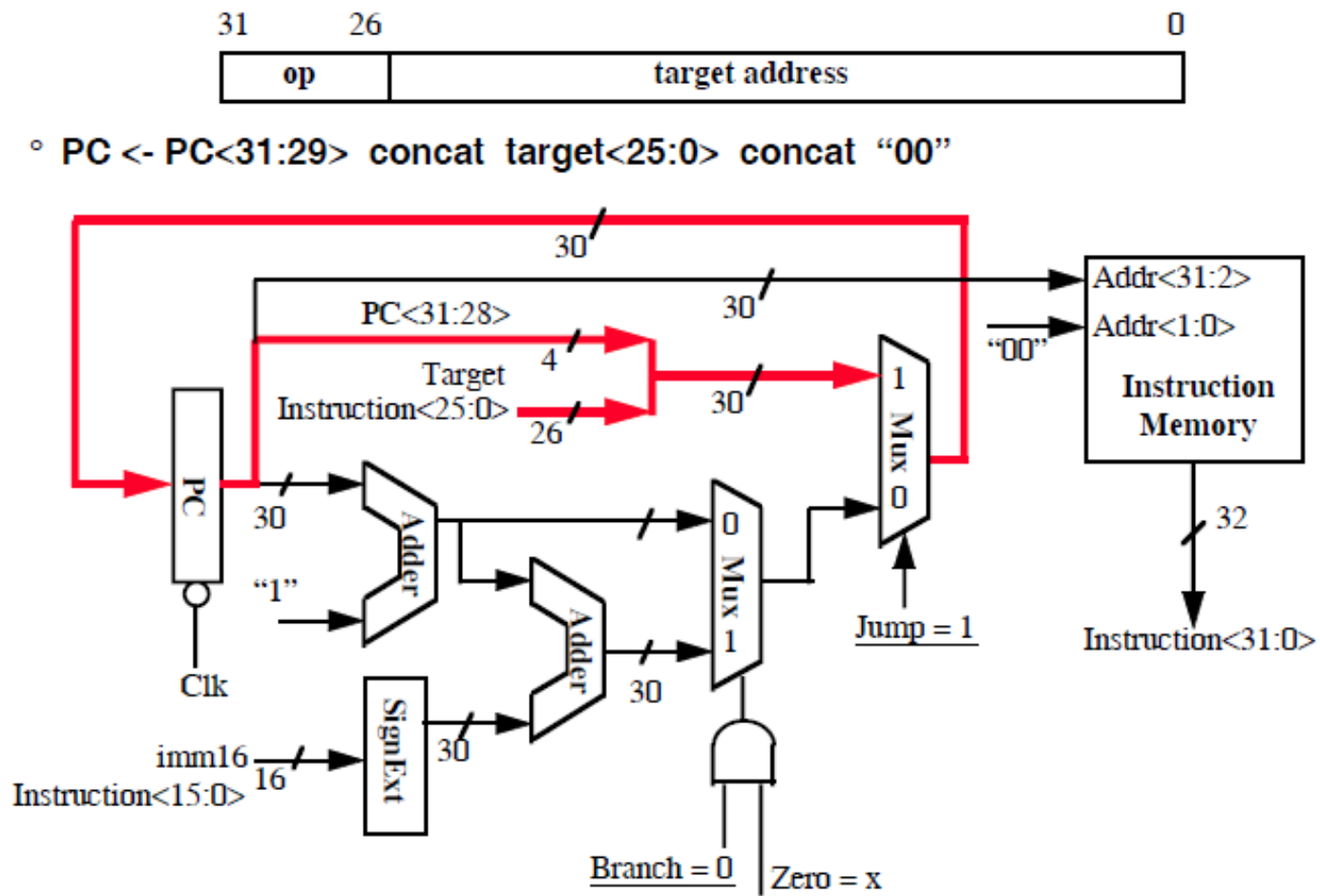


The Single Cycle Datapath during Jump

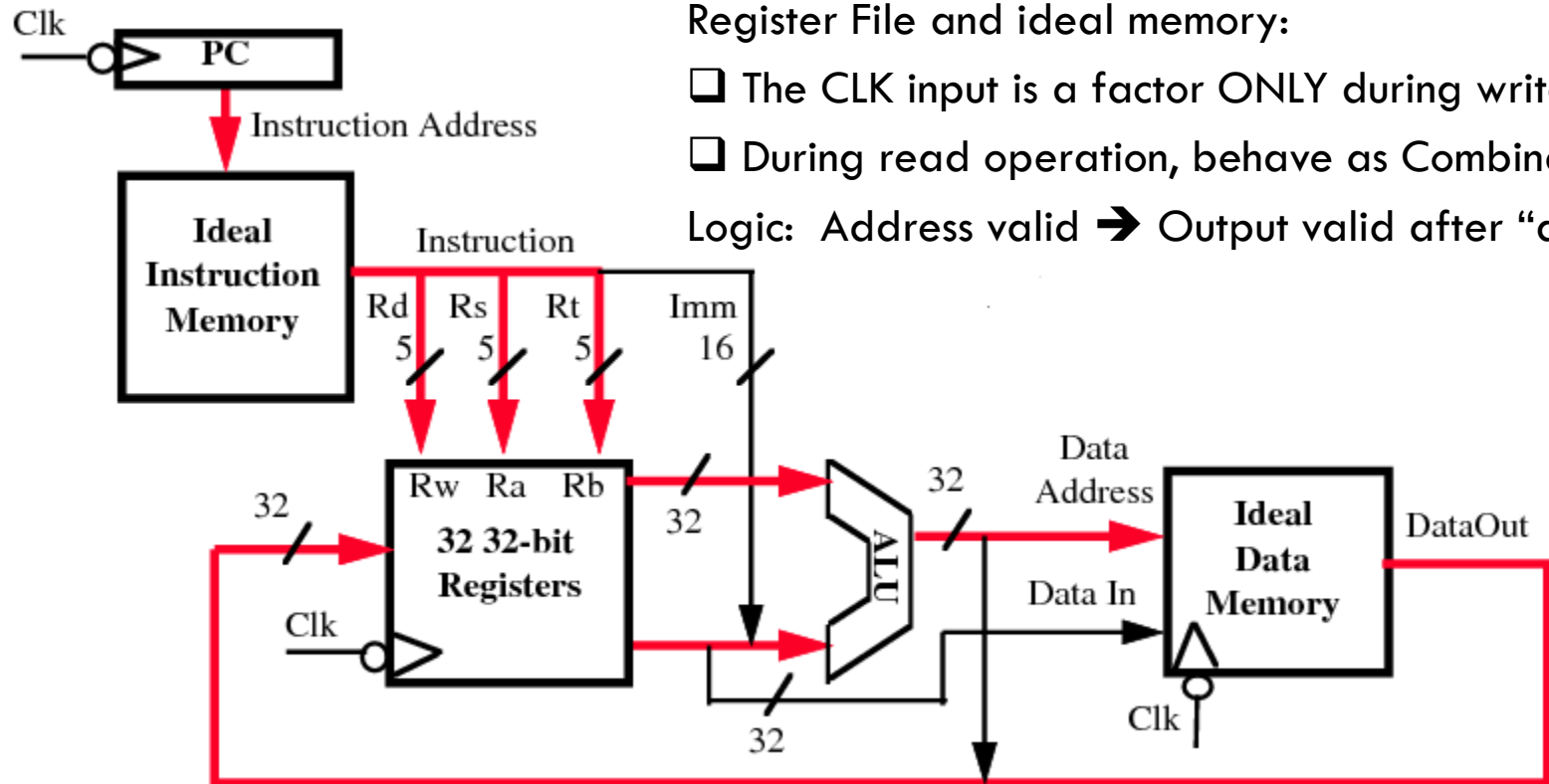
- Nothing to do! Make sure control signals are set correctly!



Instruction Fetch Unit at the End of Jump



An Abstract View of the Critical Path



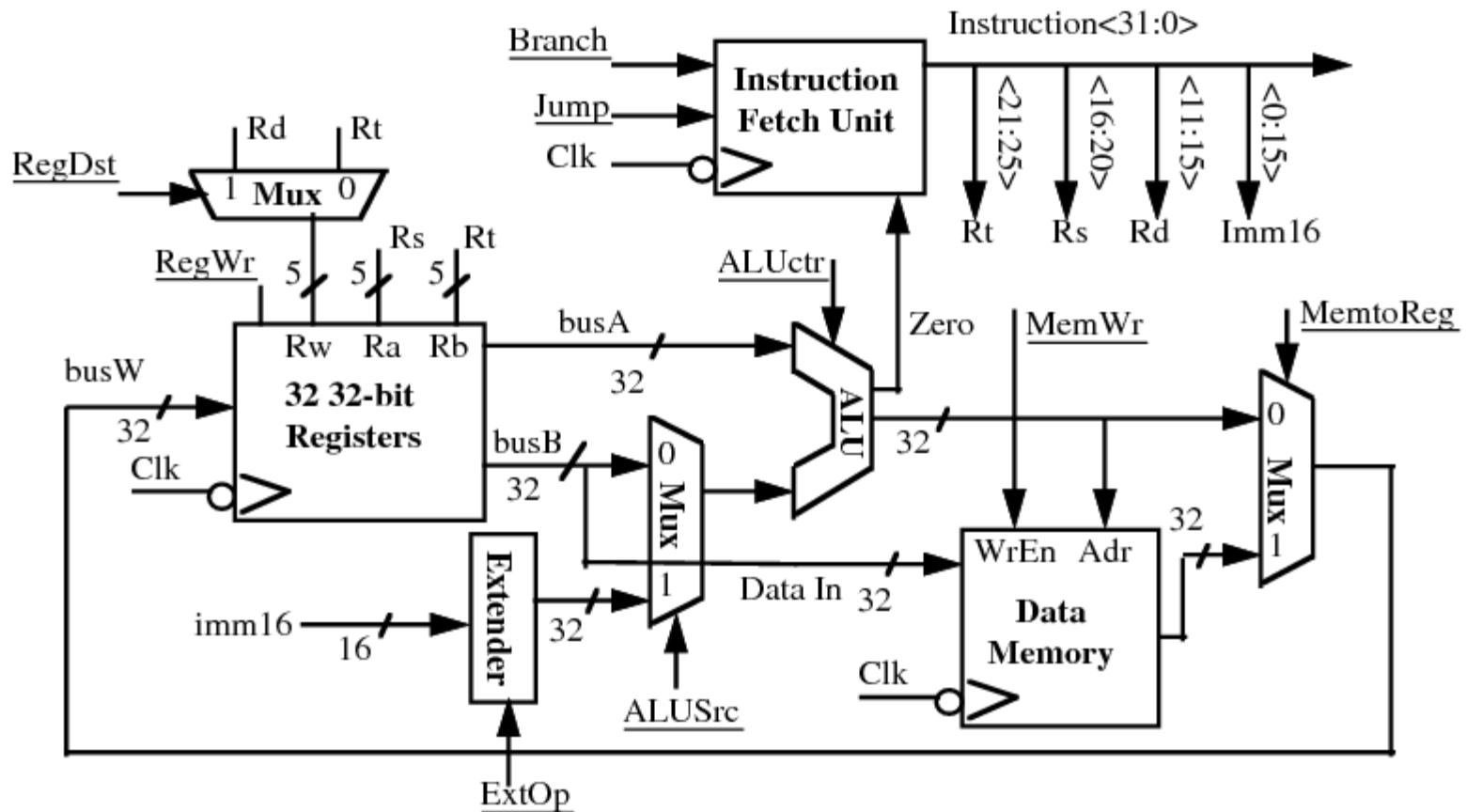
Register File and ideal memory:

- ❑ The CLK input is a factor ONLY during write operation
- ❑ During read operation, behave as Combinational Logic: Address valid → Output valid after “access time”

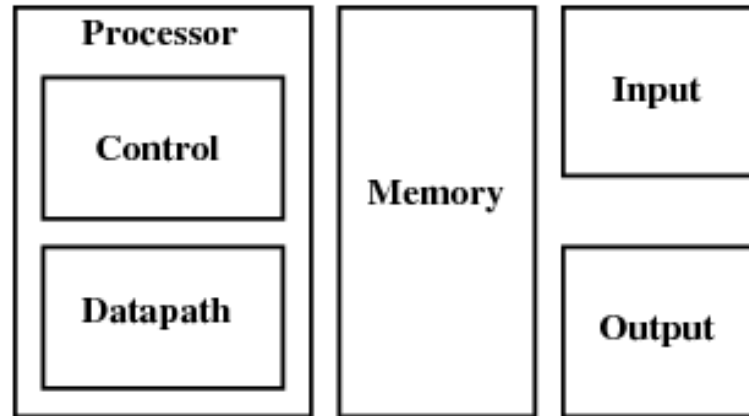
Critical Path (Load Operation) = PC's Clk-to-Q + Instruction Memory's Access Time + Register File's Access Time + ALU to Perform 32-bit Add + Data Memory Access Time + Setup Time for Register File Write

Putting it all together: A Single Cycle Datapath

We have everything except control signals (underline)

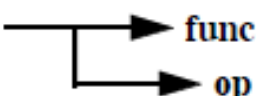


The Big Picture: Where are we Now?



- ❑ The Five Classic Components of a Computer
- ❑ **Next Topic:** Control Path Design

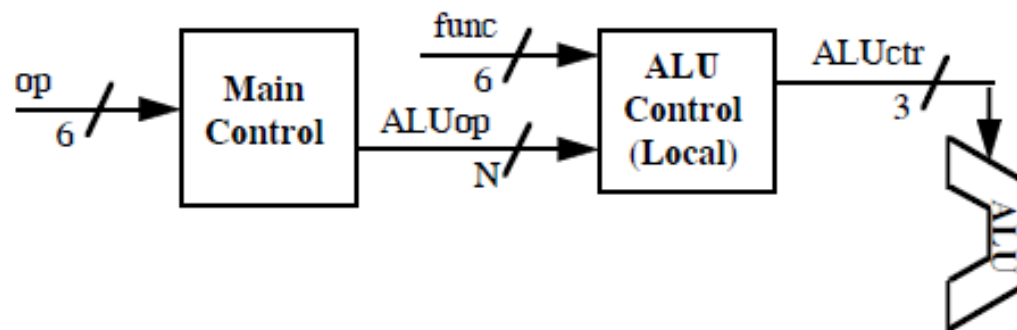
A Summary of Control Signals

	func	10 0000	10 0010	We Don't Care :-)				
	op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
		add	sub	ori	lw	sw	beq	jump
RegDst		1	0	0	0	x	x	x
ALUSrc		0	0	1	1	1	0	x
MemtoReg		0	0	0	1	x	x	x
RegWrite		1	1	1	1	0	0	0
MemWrite		0	0	0	0	1	0	0
Branch		0	0	0	0	0	1	0
Jump		0	0	0	0	0	0	1
ExtOp		x	x	0	1	1	x	x
ALUctr<2:0>		Add	Subtract	Or	Add	Add	Subtract	xxx

	31	26	21	16	11	6	0	
R-type	op	rs	rt	rd	shamt	funct		add, sub
I-type	op	rs	rt	immediate				ori, lw, sw, beq
J-type	op	target address						jump

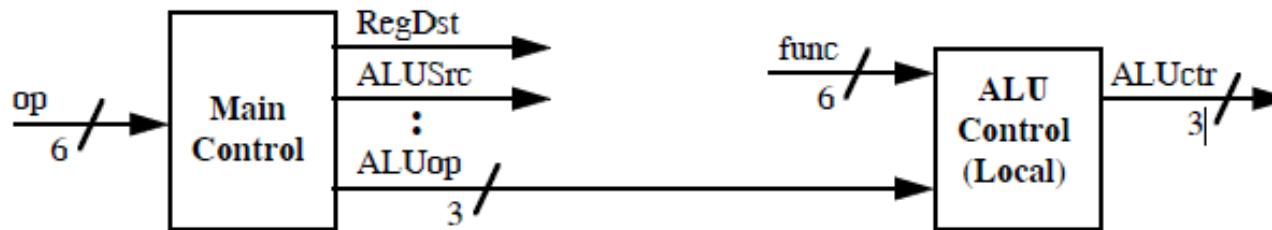
The Concept of Local Decoding

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



Key Idea: Two levels of Control logic.

The “Truth Table” for the Main Control

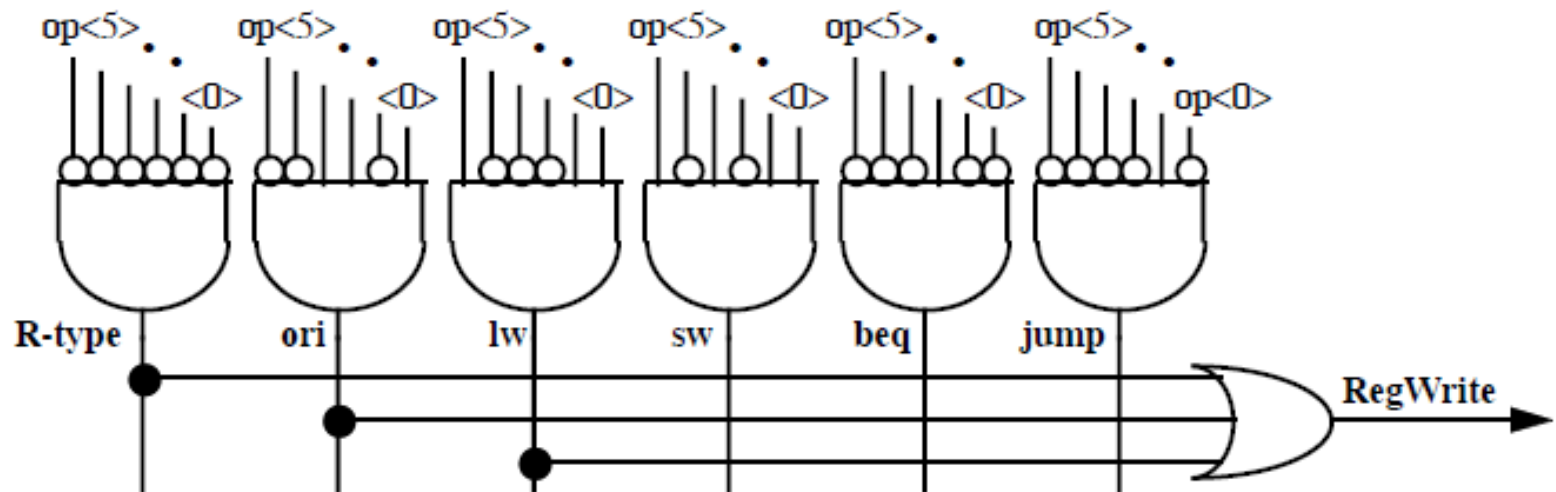


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

Question: Can you write the truth table for the ALU control keeping in mind the ALU we designed in the class?

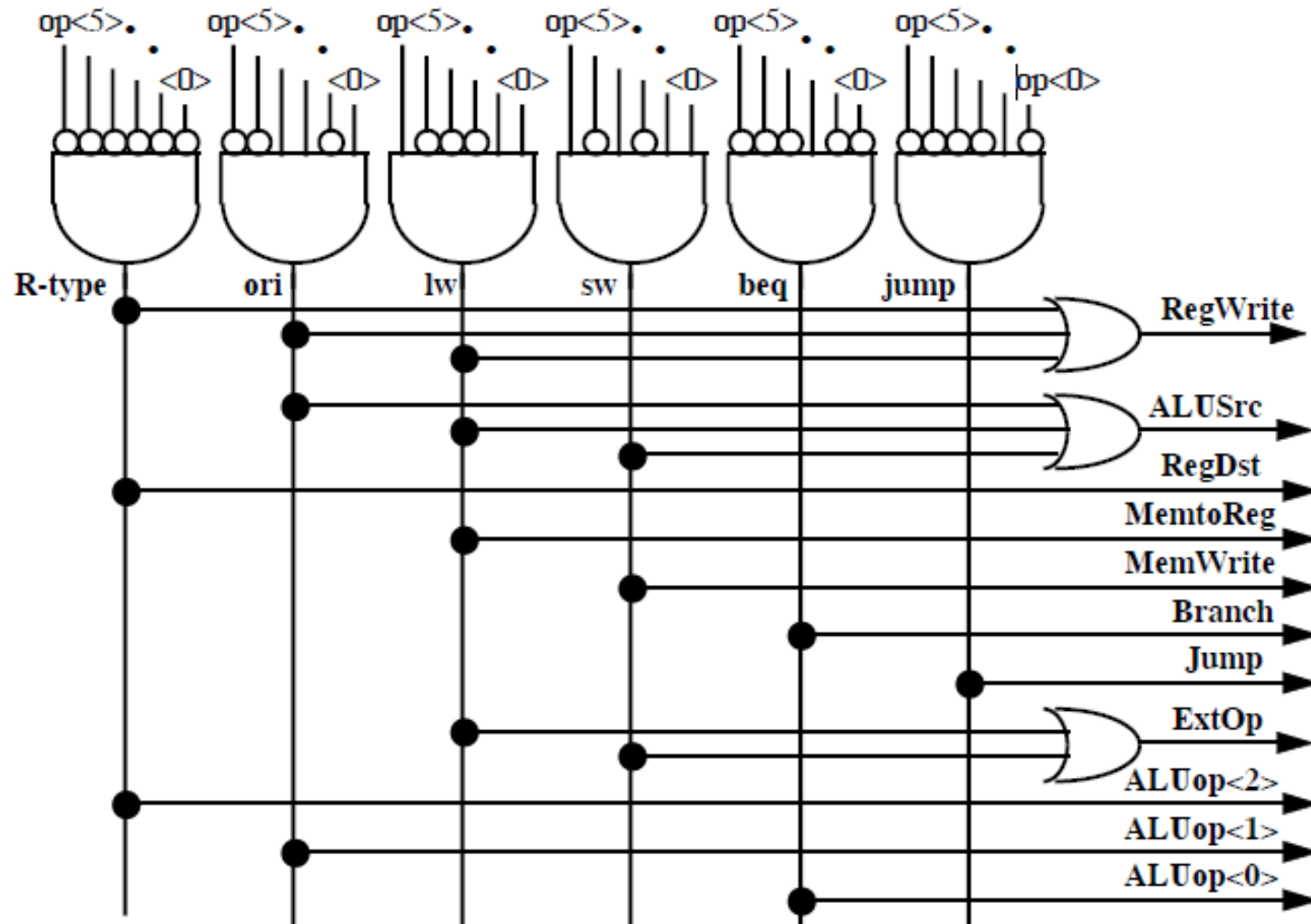
The “Truth Table” for RegWrite

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	x	x	x

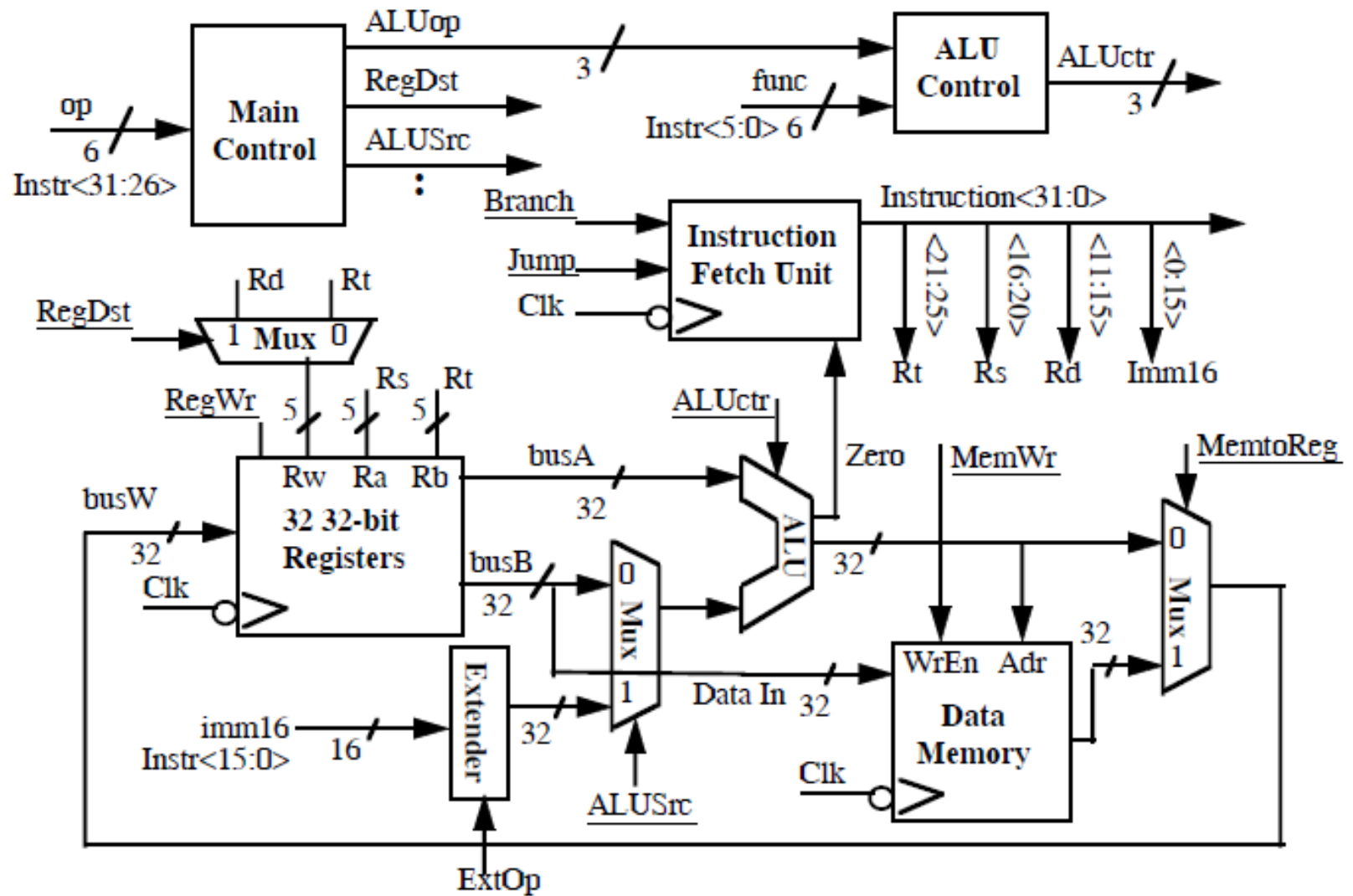


PLA Implementation of Main Control

Hmm! What is PLA?



Putting it All Together: A Single Cycle Processor



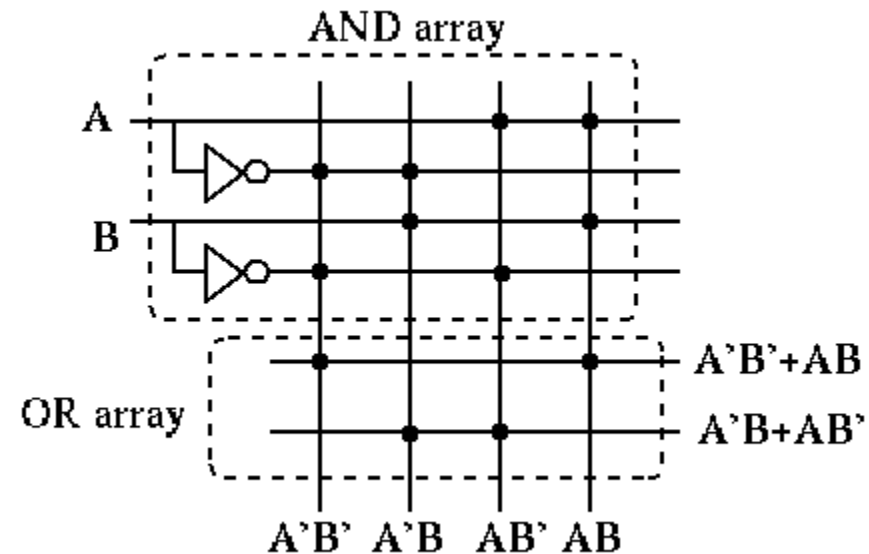
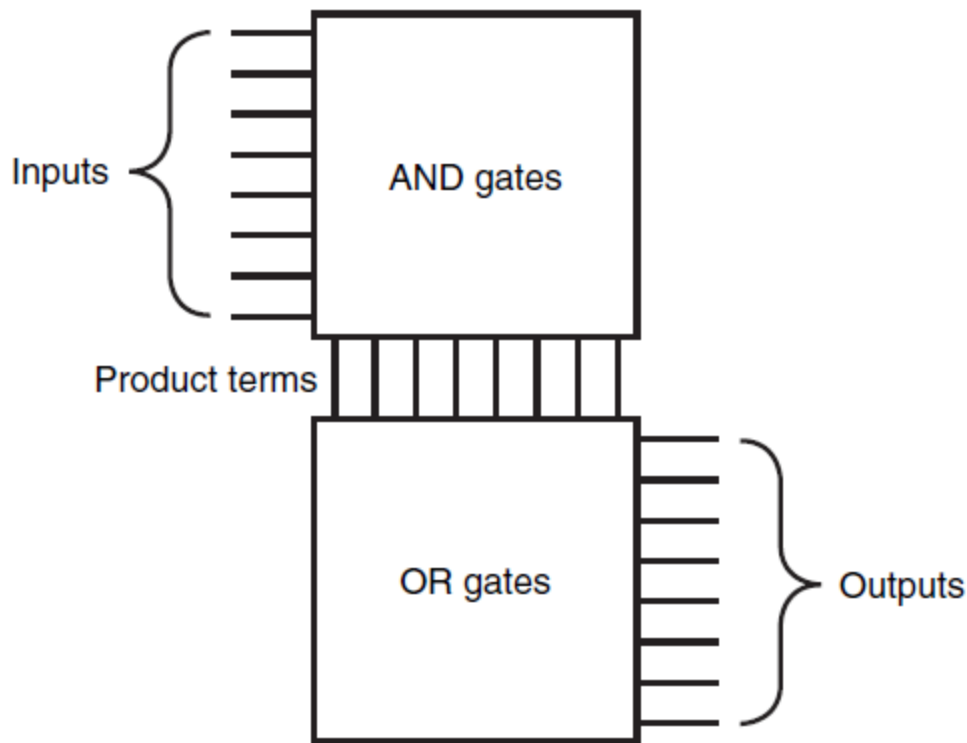
Drawback of this Single Cycle Processor

- ❑ 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
- ❑ Cycle time is much longer than needed for all other instructions

We are assuming
Clock Skew is zero

Programmable Logic Arrays

- PLAs can be used to realize combinational circuits



PLAs

Inputs

A
B
C

