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

COMPUTER SYSTEMS ORGANIZATION

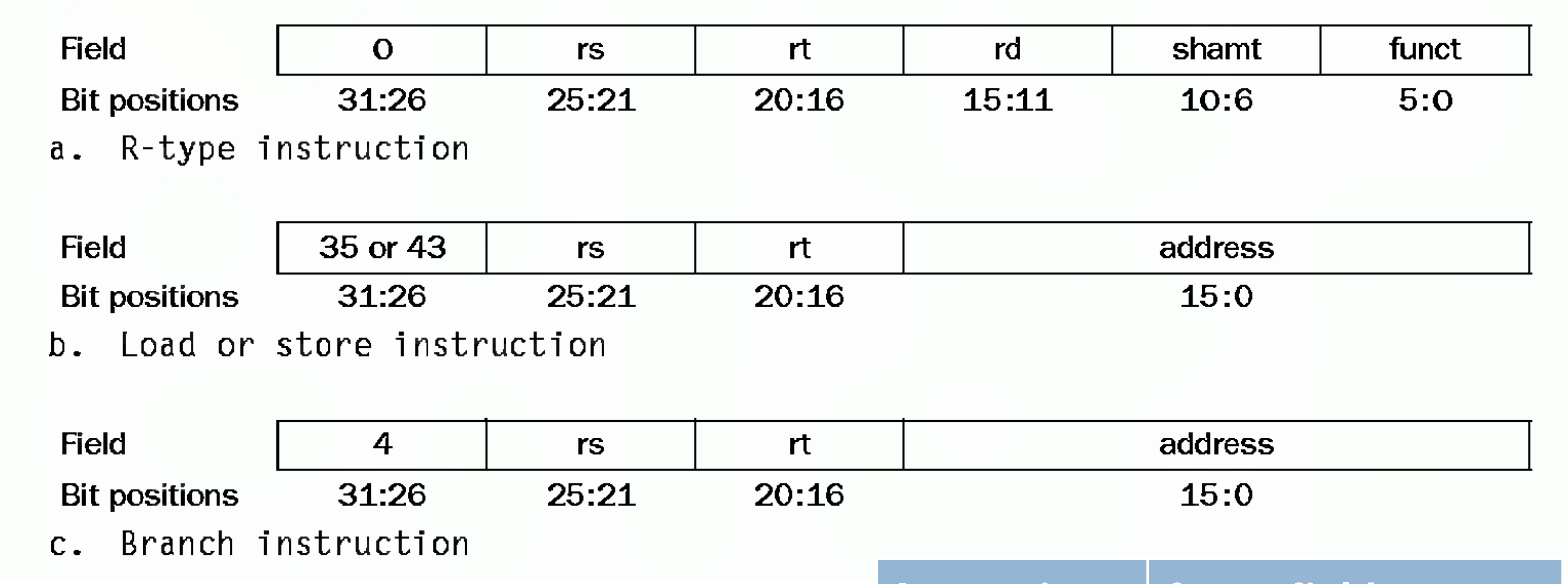
Single Cycle CPU Design -- Spring 2010 -- IIIT-H -- Suresh Purini

Hardware Realization of MIPS ISA

Goal: Design a processor which can execute the following 8 instructions?

Instructions	Functionality
lw r_1 , 24(r_2)	r_1 = mem[r_2 + 24]
sw r_1 , 24(r_2)	mem[r_2 + 24] = r_1
add r_1, r_2 , r_3	$r_1 = r_2 + r_3$
sub r_1, r_2 , r_3	$r_1 = r_2 - r_3$
and r_1, r_2 , r_3	$r_1 = r_2$ & r_3
or r_1, r_2 , r_3	$r_1 = r_2 \mid r_3$
slt r_1, r_2 , r_3	if $r_2 < r_3$ then $r_1 = 1$ else $r_1 = 0$
beq r_1, r_2, 25	if $r_1 == r_2$ then pc = pc + 4 + 100

MIPS Instruction Format



Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

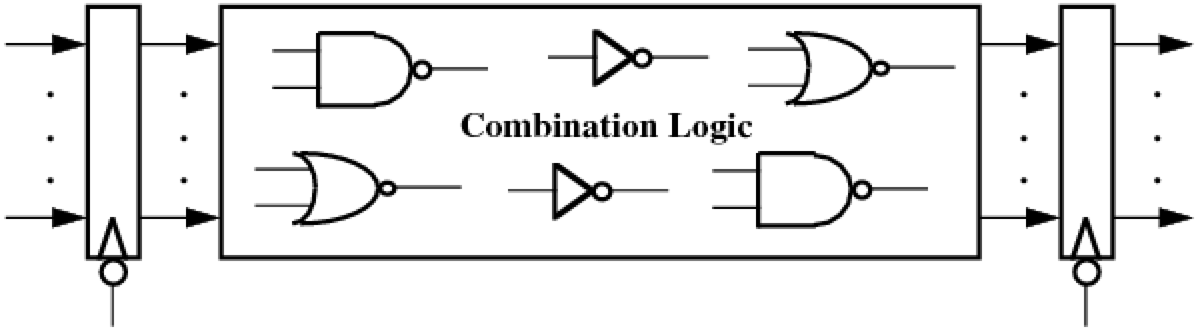
b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

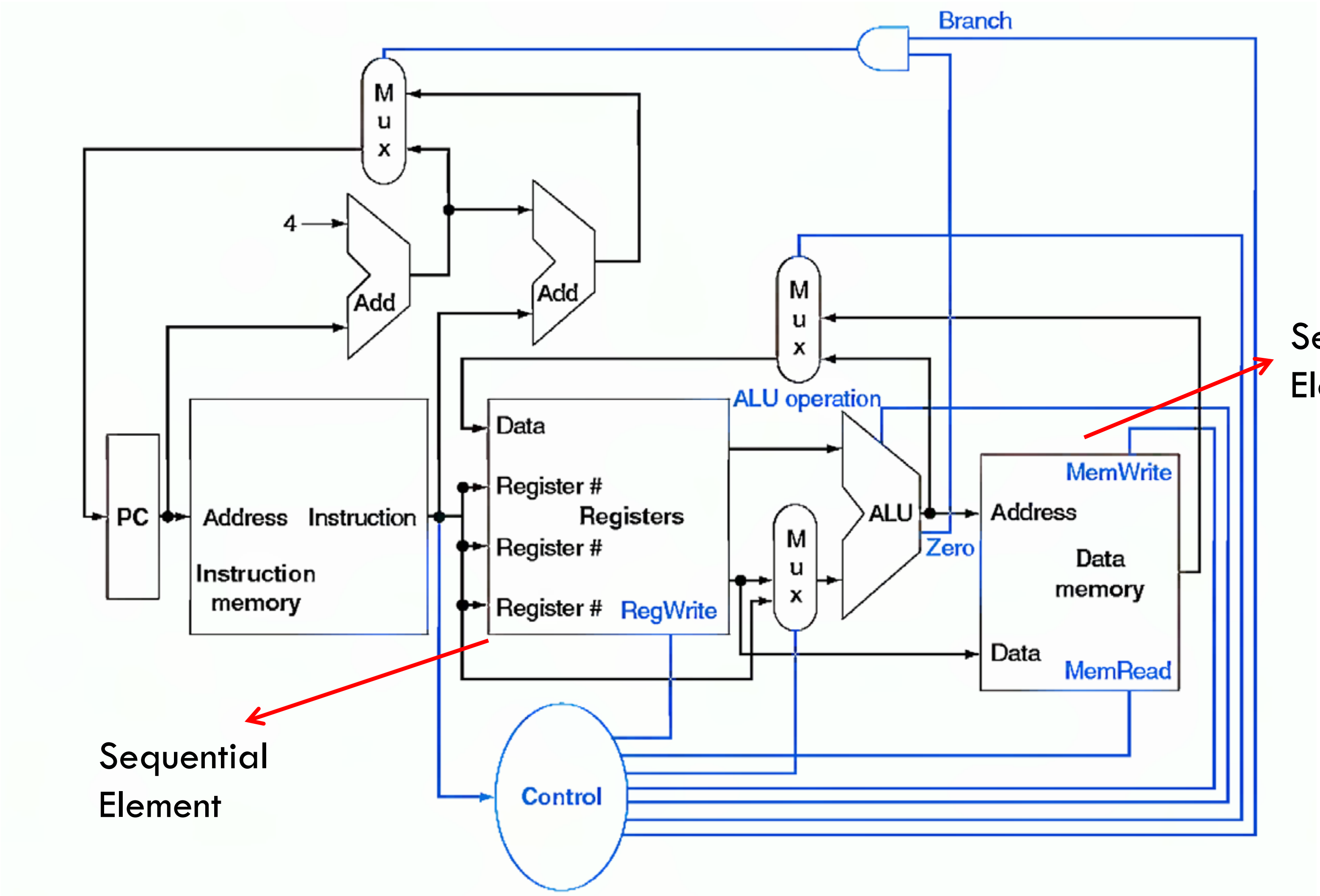
Instruction	funct field
add	100000 (32)
sub	100010 (34)
and	100100 (36)
or	100101 (37)
nor	100111 (39)
slt	101010 (42)

Sequential and Combinational Circuits



- What’s the difference between sequential and combinational circuits?

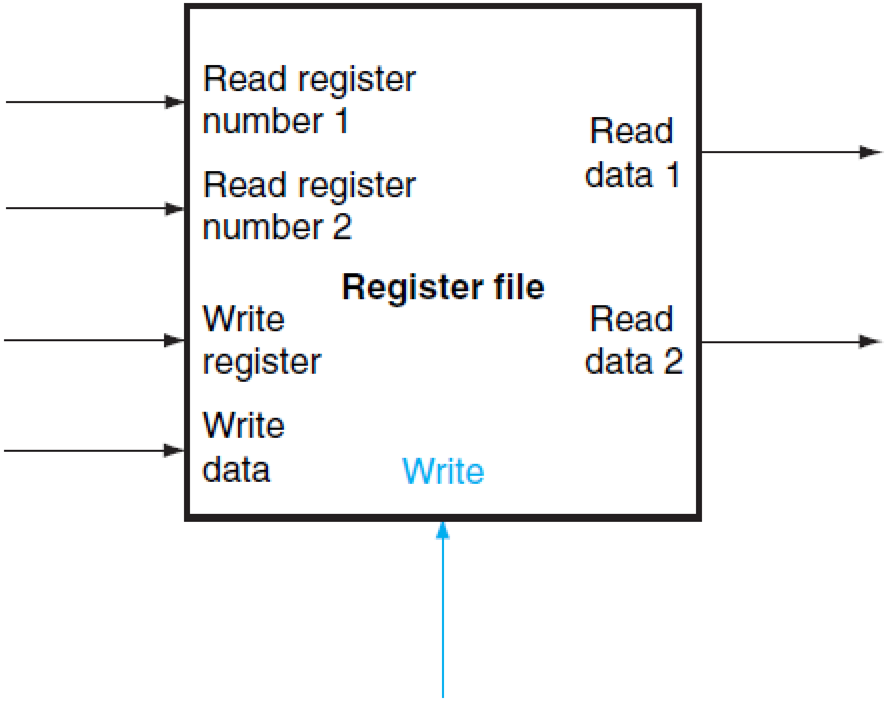
Abstract Design of MIPS Processor



Sequential Element

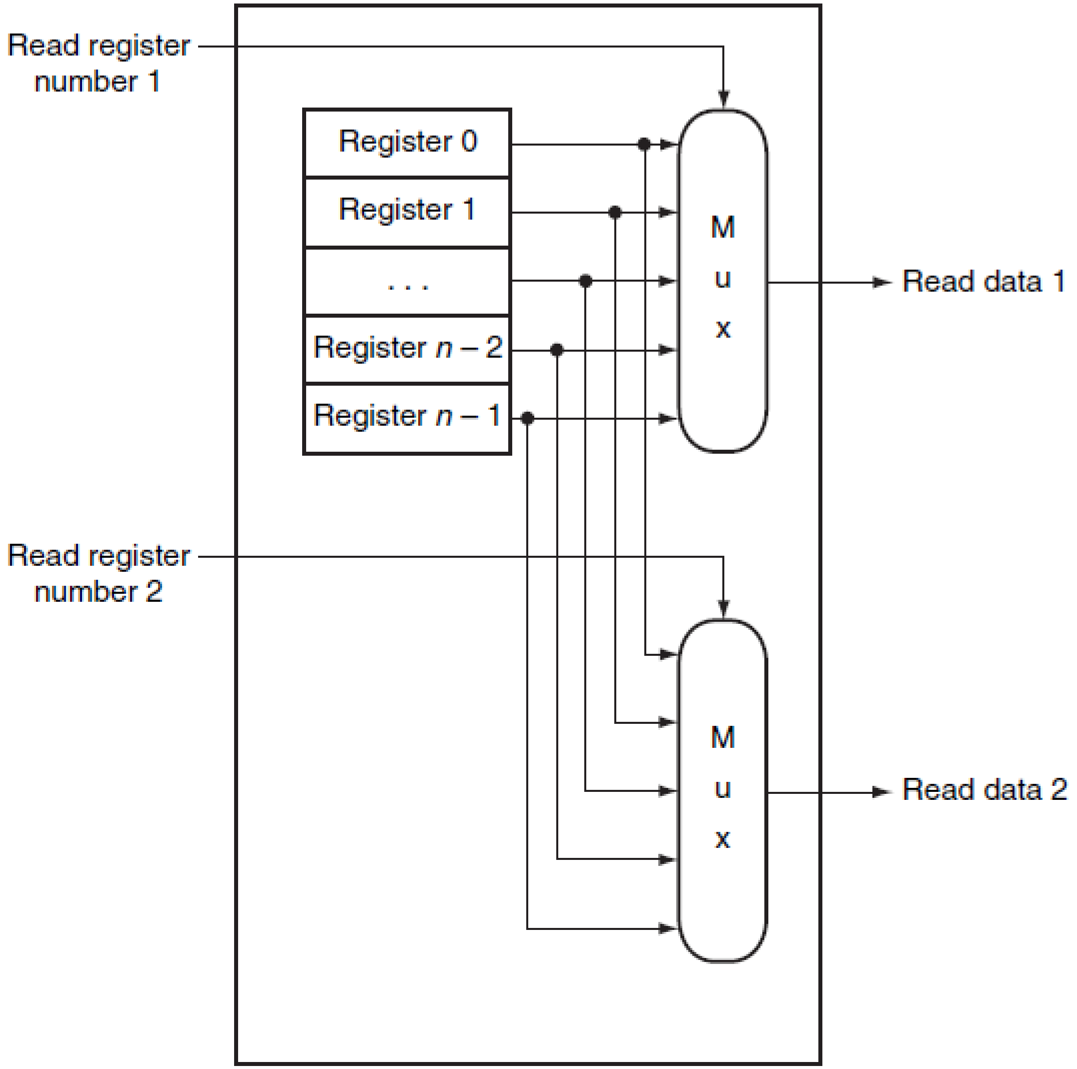
Sequential Element

Register File

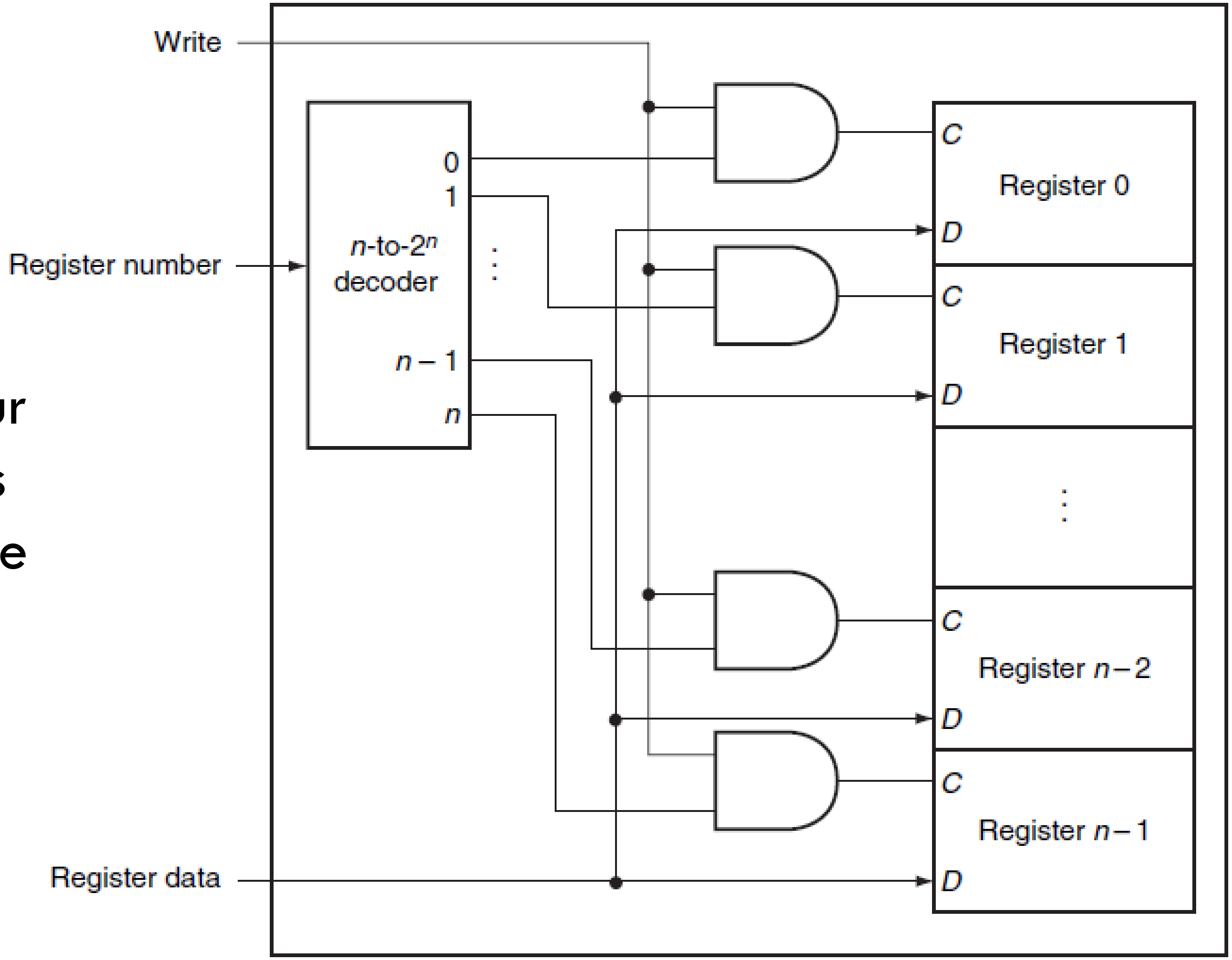


- Register file is driven by a clock not shown in this picture
- Clock has effect only when we are writing to the register file

Register File

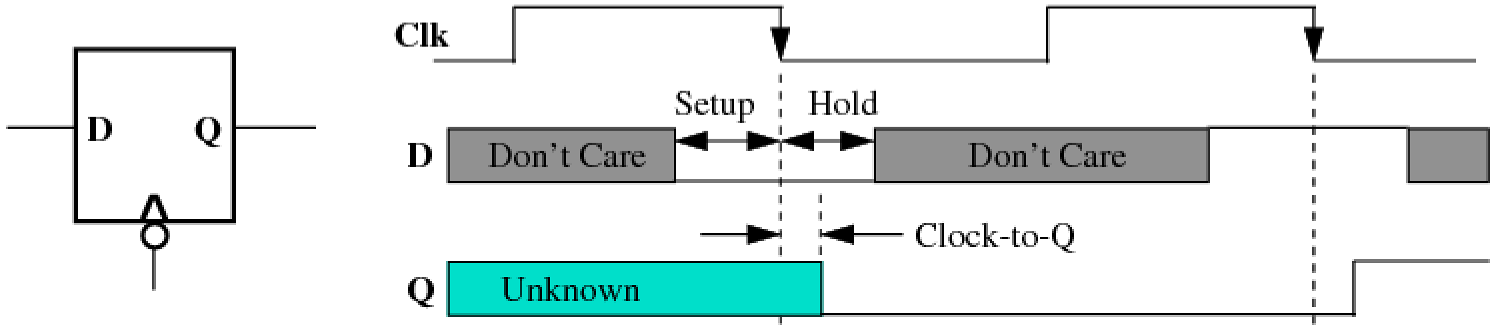


Register File



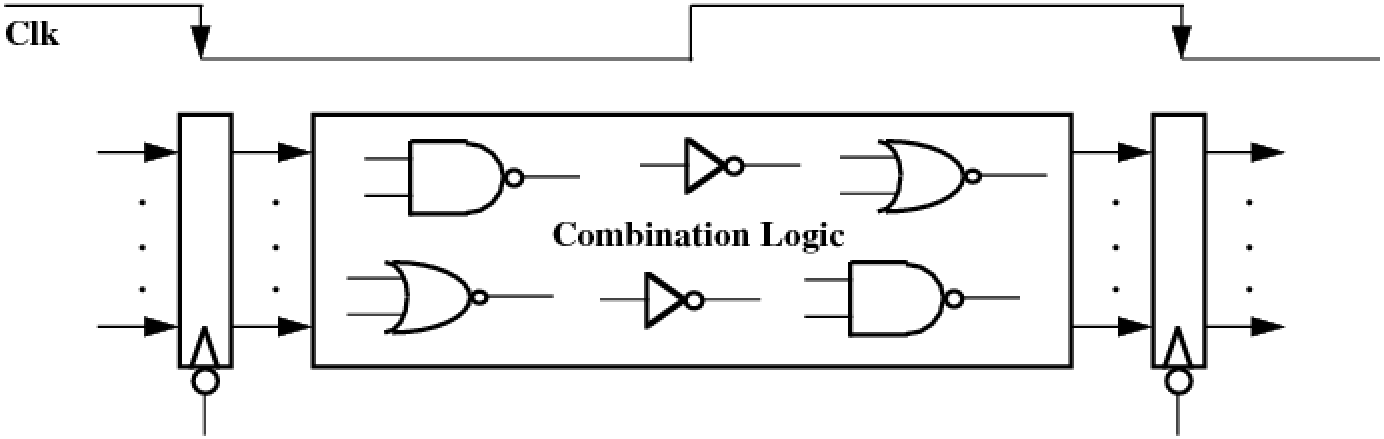
Convention: All our storage elements are negative edge triggered.

Storage Element Timing Model – Negative Edge Triggered D-Flip Flop



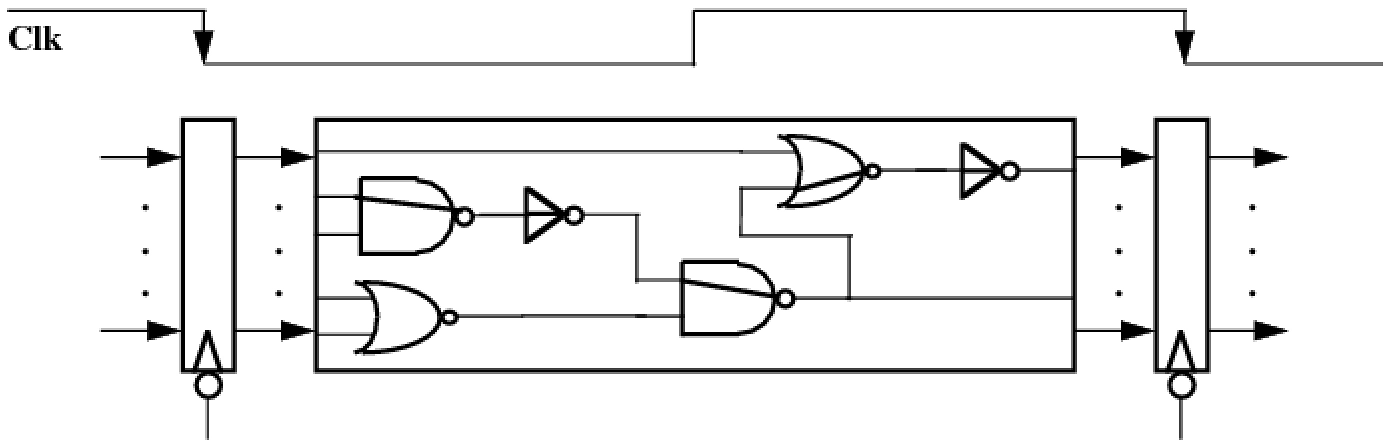
- Setup Time: Input must be stable BEFORE the trigger clock edge.
- Hold Time: Input must be stable AFTER the trigger clock edge.
- Clock-to-Q time: Output cannot change instantaneously at the trigger clock edge.
  - Similar to delay in logic gates.

Clocking Methodology



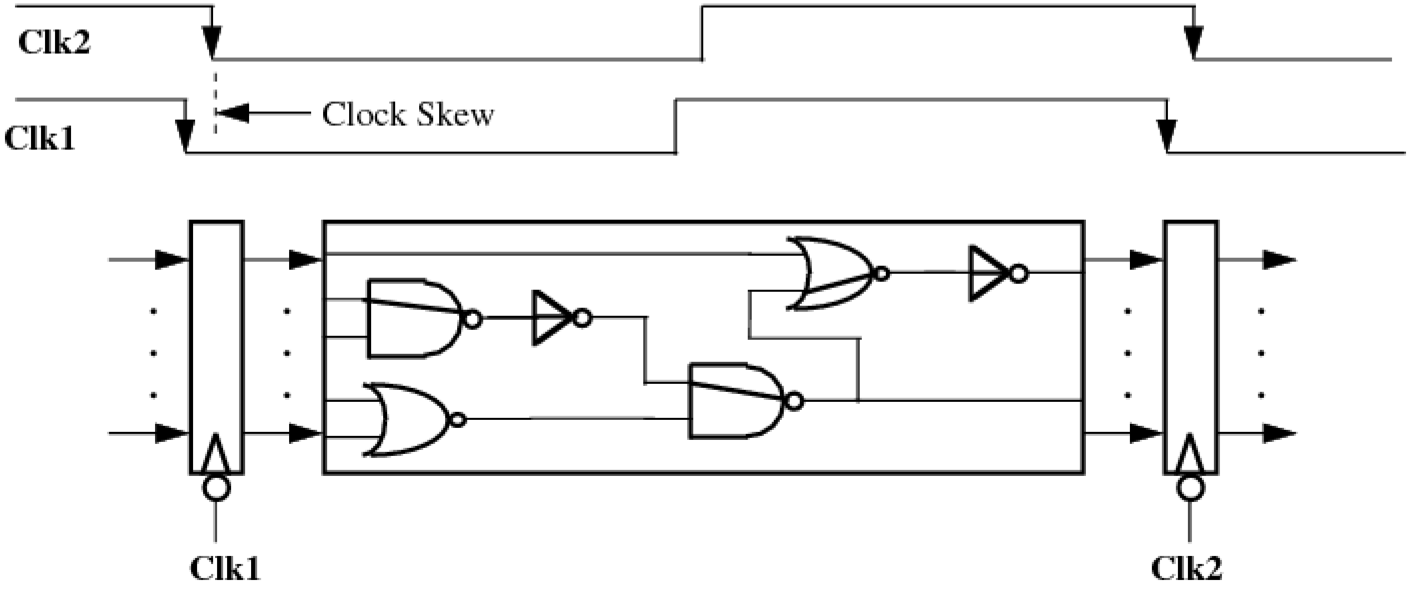
- All storage elements are clocked by the same clock edge
- The combination logic block's:
  - Inputs are updated at each clock tick
  - All outputs MUST be stable before the next clock tick

Critical Path and Cycle Time

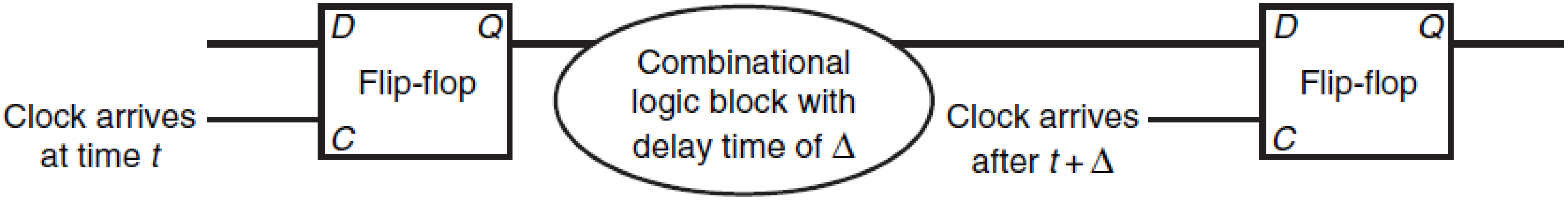


- Critical Path: Slowest path between any two storage devices
- Cycle time is a function of critical path
- More specifically, the cycle time must be greater than:
  - Clock-to-Q + Longest Path through the Combinational Logic + Setup Time

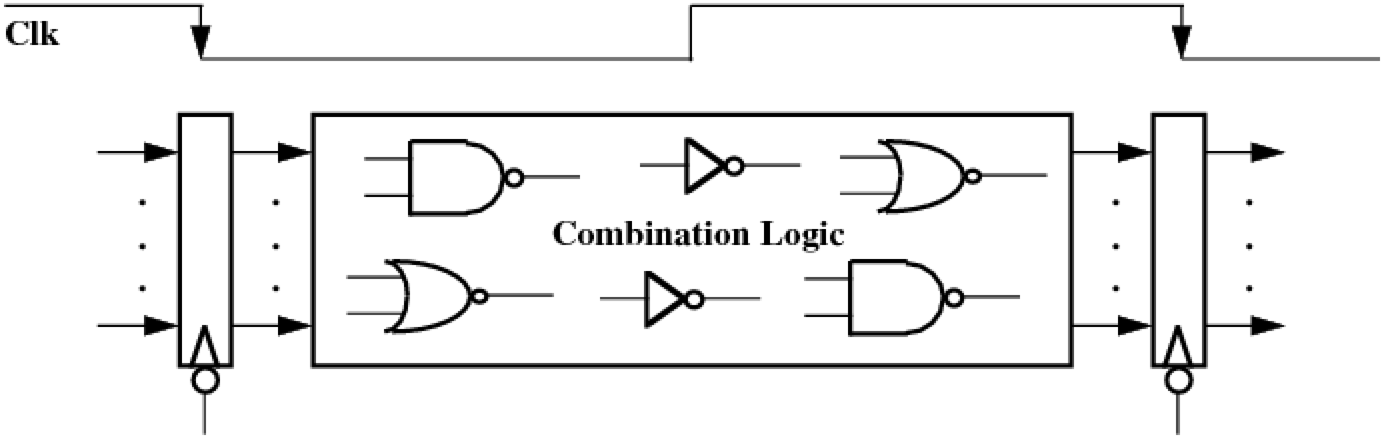
Clock Skew’s Effect on Cycle Time



- ❑How to take care of Clock Skew?
- ❑We shall assume there is not Clock Skew.

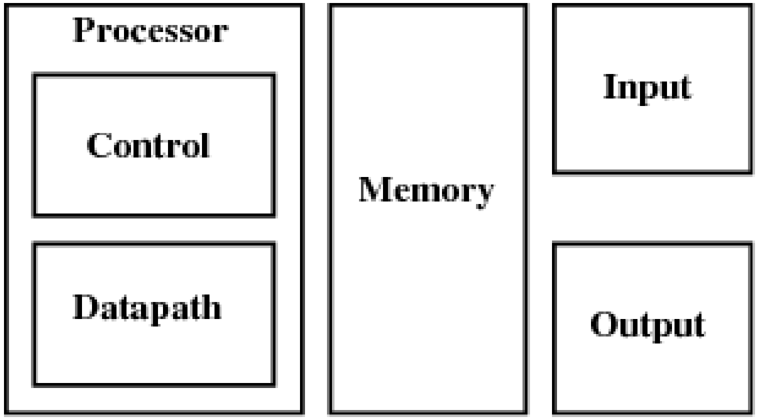


How to Avoid Hold Time Violation?



- □ Hold time requirement:
  - Input to register must NOT change immediately after the clock tick.
- □ CLK-to-Q + Shortest Delay Path must be greater than Hold Time

The Big Picture: Where are we Now?

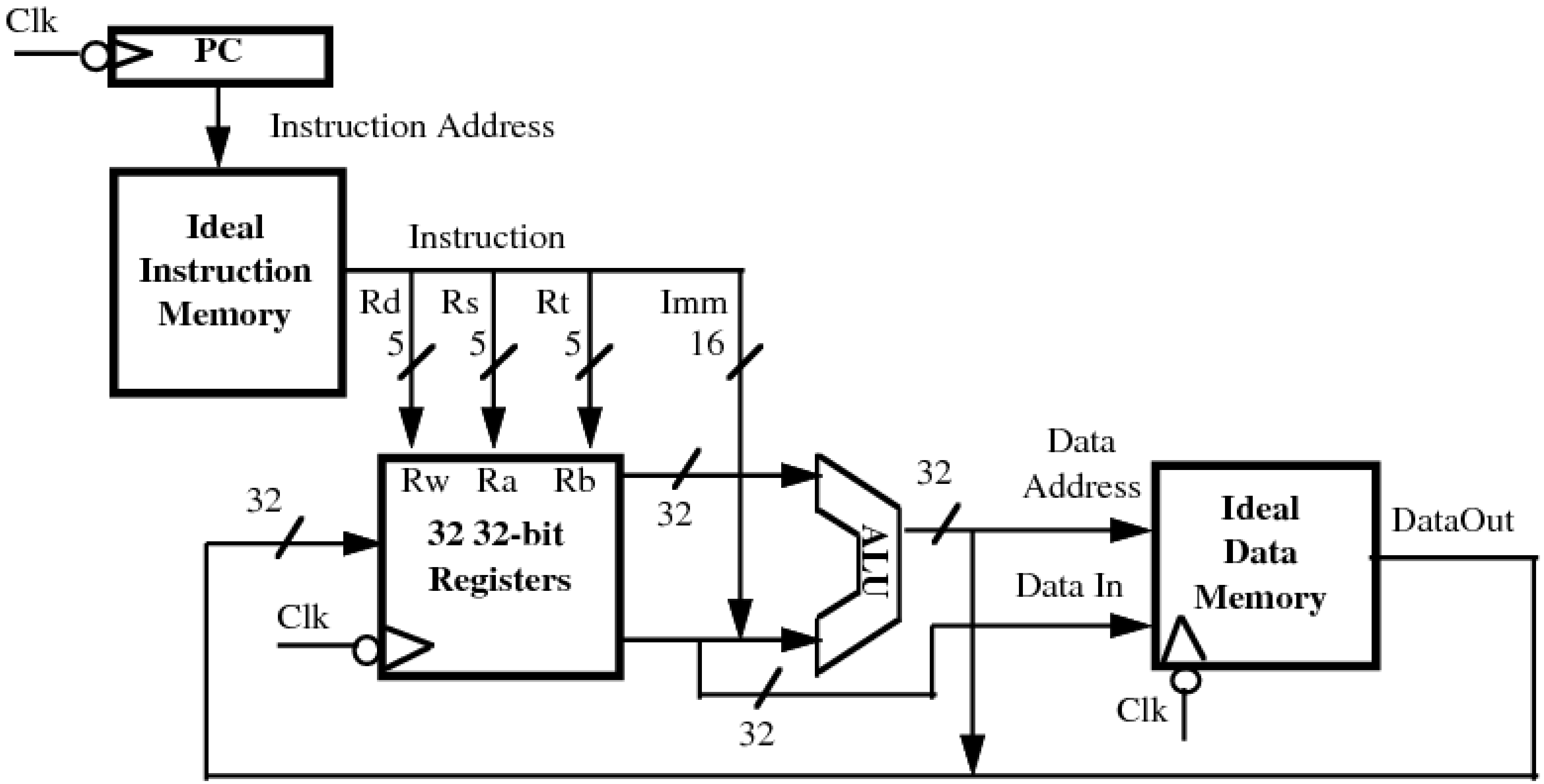


- The Five Classic Components of a Computer
- We shall start with the Data Path Design

The Big Picture: Performance Perspective

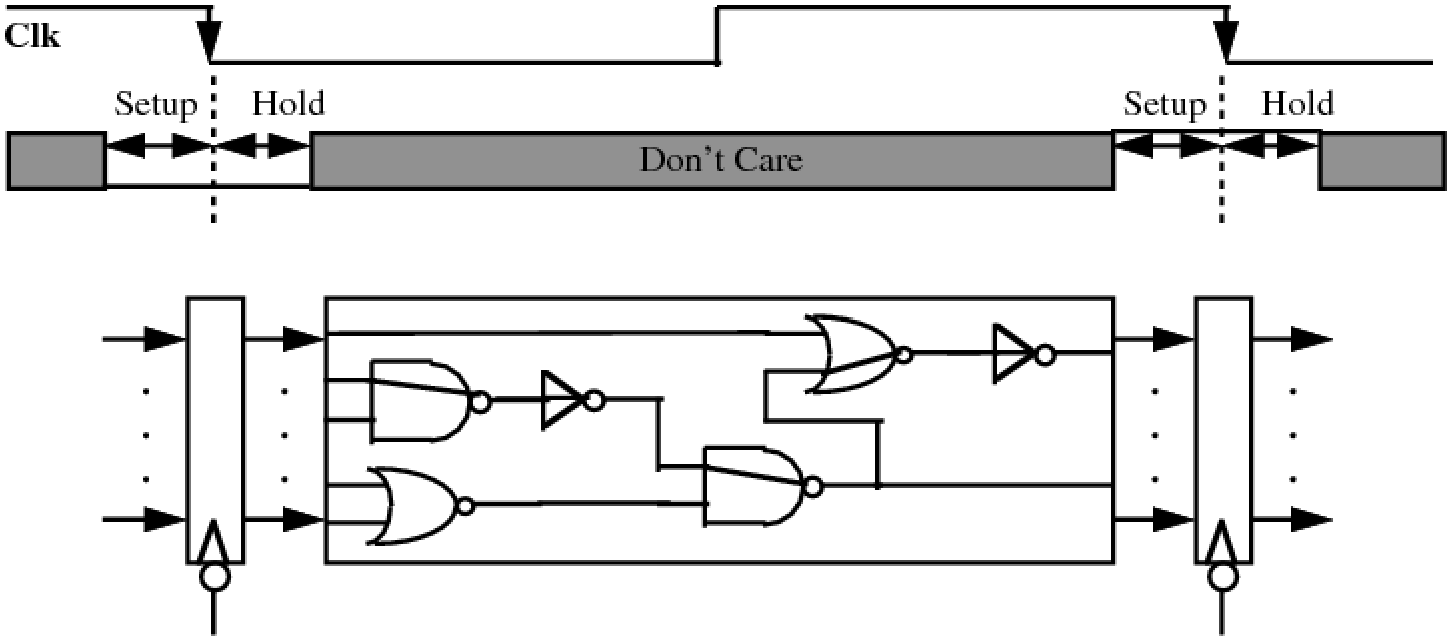
- Performance of a machine was determined by
  - Instruction Count
  - Clock cycle Time
  - Clock cycles per instruction
- Processor Design (data path and control) will determine
  - Clock cycle time
  - Clock cycles per instruction
- We shall first design a Single Cycle Processor
  - Advantage: One clock cycle per instruction
  - Disadvantage: Long cycle time

An Abstract View of Implementation



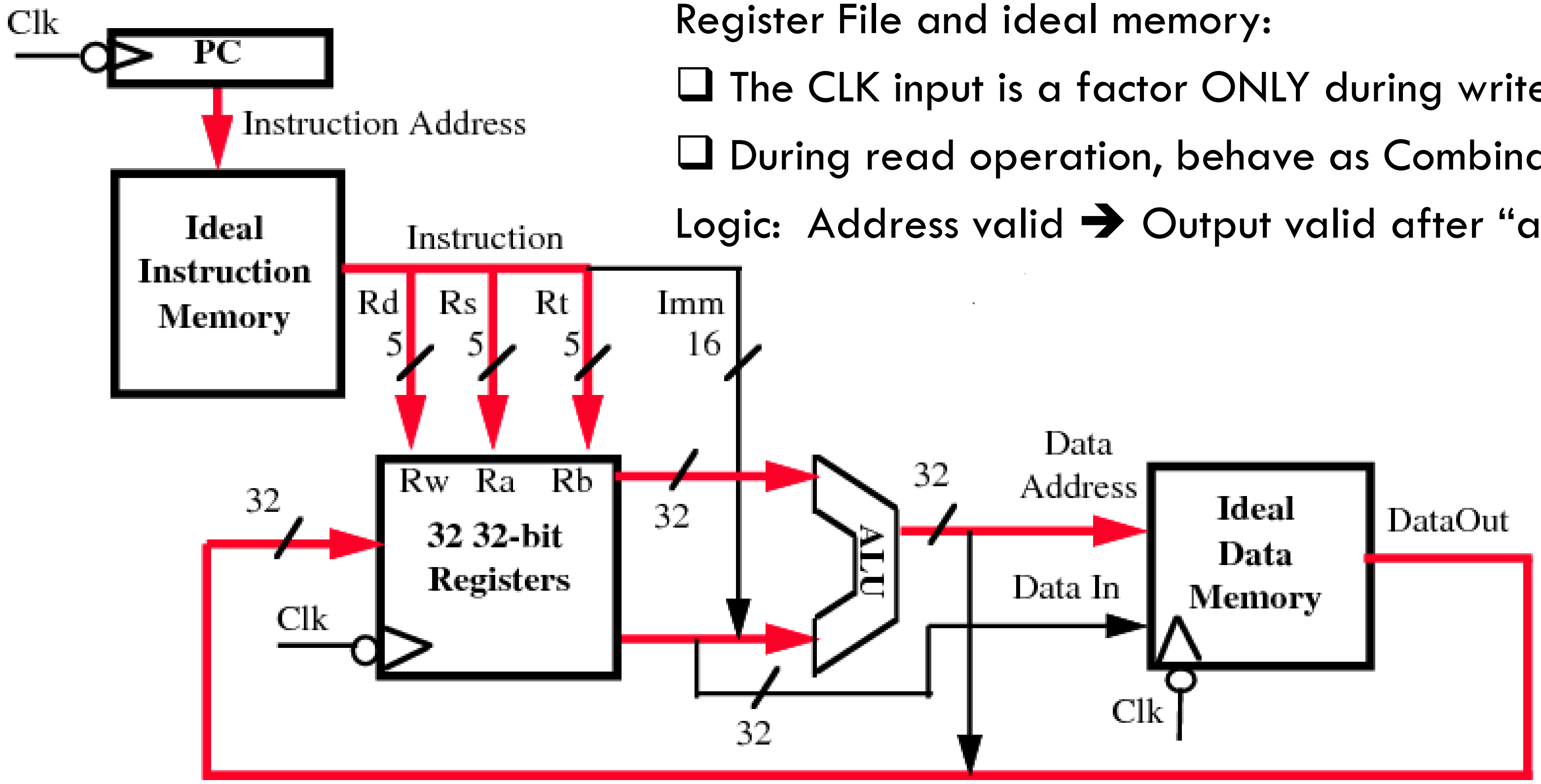
Assumption: Clock Skew is zero

Clocking Methodology



- All storage elements are clocked by the same clock edge
- Cycle time = CLK-to-Q + Longest Delay Path + Setup Time
- (CLK-to-Q + Shortest Delay Path) > Hold Time

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"

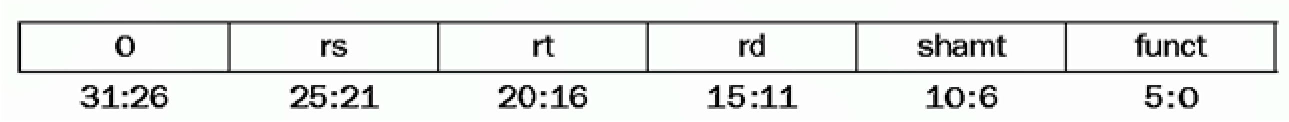
Critical Path (Load Operation) = PC's Clk-to-Q + Instruction Memory's Access Time + Register File's Access Time + ALU to Perform 32-bit Add + Data Memory Access Time + Setup Time for Register File Write

The Steps of Designing a Processor

- Instruction Set Architecture => Register Transfer Language
- Register Transfer Language =>
  - Datapath components
  - Datapath interconnect
- Datapath components => Control Signals
- Control signals => Control logic

RTL: The ADD Instruction

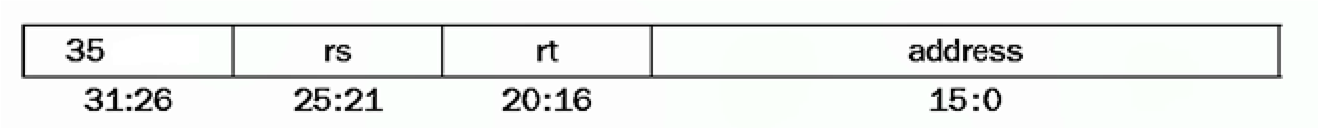
For ADD funct = 100000



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

- add rd, rs, rt
  - mem[pc] Fetch the instruction from the memory
  - R[rd] = R[rs] + R[rt] The ADD operation
  - PC = PC + 4 Calculate the next instruction’s address

RTL: The LOAD Instruction

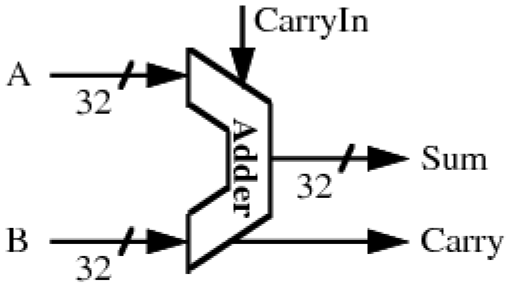


35	rs	rt	address
31:26	25:21	20:16	15:0

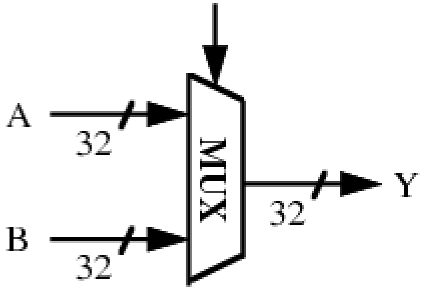
- lw rd, rs, imm16
  - mem[pc] Fetch the instruction from the memory
  - Addr = R[rs] + SignExt32(imm16) Calculate memory address
  - R[rt] = Mem[Addr] Load the data into the register
  - PC = PC + 4 Calculate the next instruction's address

Combinational Logic Elements

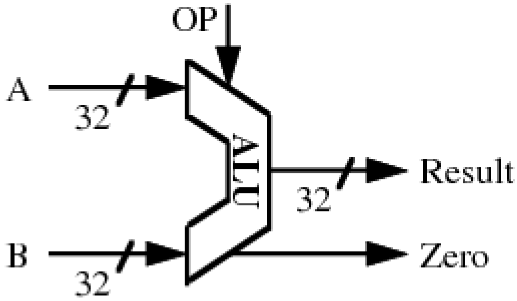
- **Adder**



- **MUX**

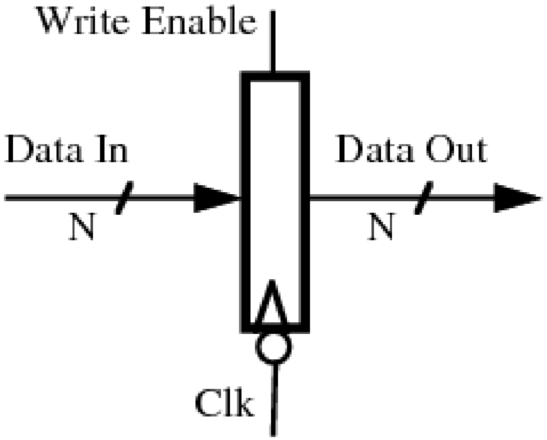


- **ALU**



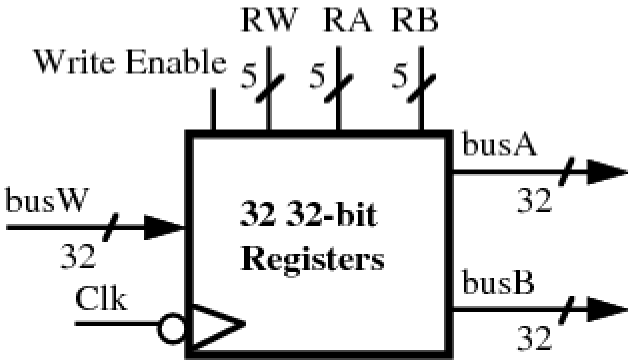
Sequential Element: Register

- Register
  - Similar to D Flip Flop except
    - N bit input and output
    - Write Enable input
  - Write Enable
    - 0: Data out will not change
    - 1: Data out will become Data In



Storage Element: Register File

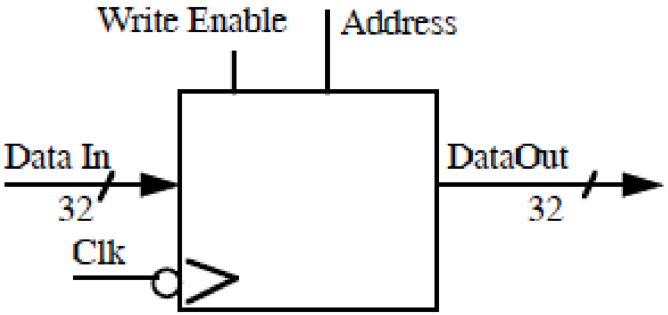
- Register File consists of 32 registers
  - Two 32-bit output busses: busA and busB
  - one 32-bit input bus: busW
- Register is selected by
  - RA selects the register to put on busA
  - RB selects the register to put on busB
  - RW selects the register to be written via busW when Write Enable is 1



- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block
    - RA or RB valid => busA or busB valid after access time

Storage Element: Memory

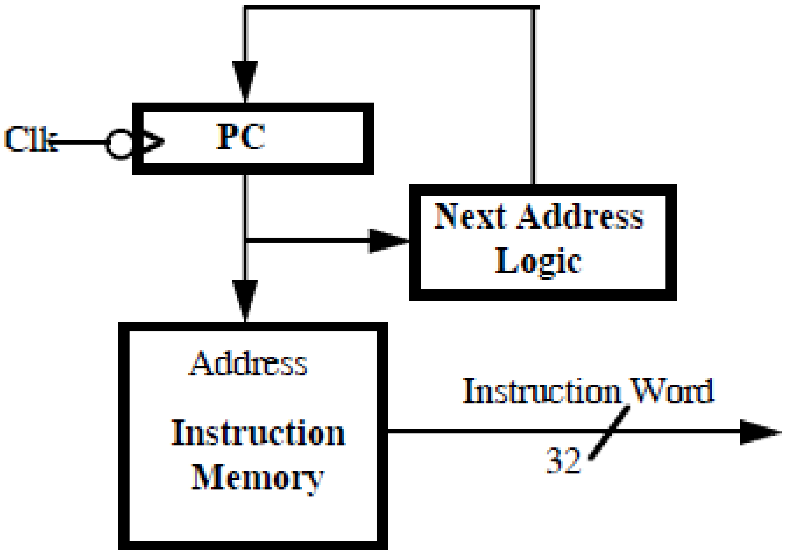
- Memory
  - One input bus: Data In
  - One output bus: Data Out



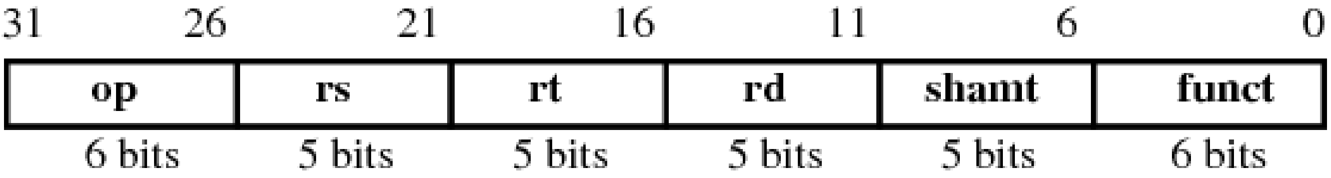
- Memory word is selected by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => Data Out valid after "access time."

Overview of Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC = PC + 4
    - Branch and Jump PC = “something else”



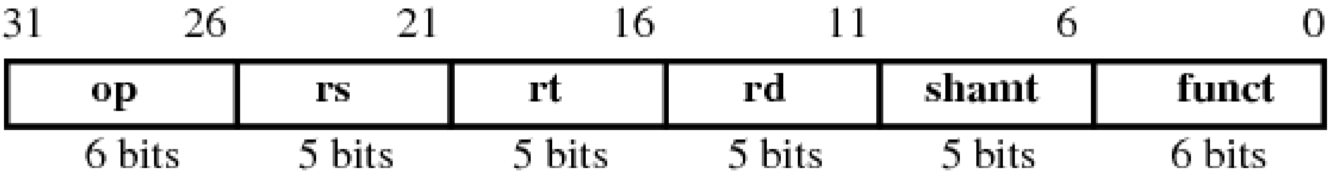
RTL: The ADD Instruction



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

- add rd, rs, rt
  - mem[pc] Fetch the instruction from the memory
  - R[rd] = R[rs] + R[rt] The ADD operation
  - PC = PC + 4 Calculate the next instruction’s address

RTL: The SUB Instruction



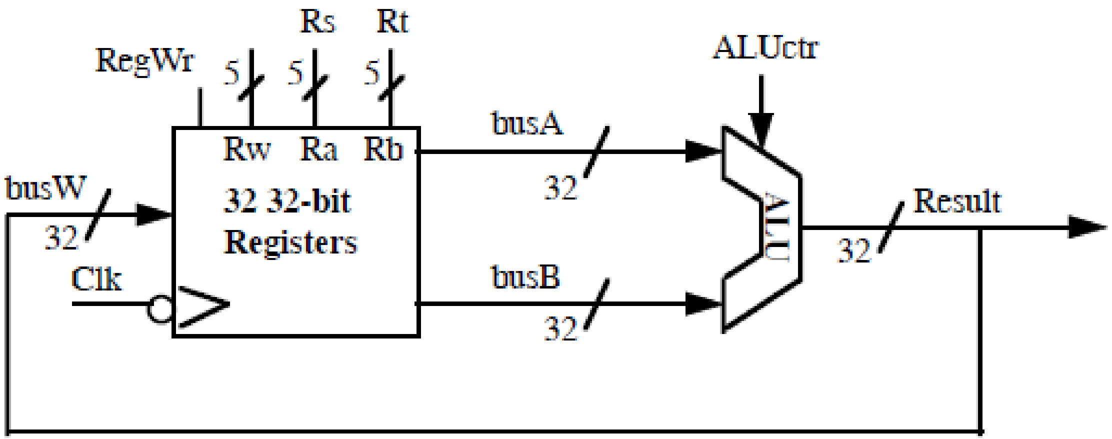
31 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- add rd, rs, rt
  - mem[pc]   Fetch the instruction from the memory
  - R[rd] = R[rs] – R[rt]   The ADD operation
  - PC = PC + 4   Calculate the next instruction's address

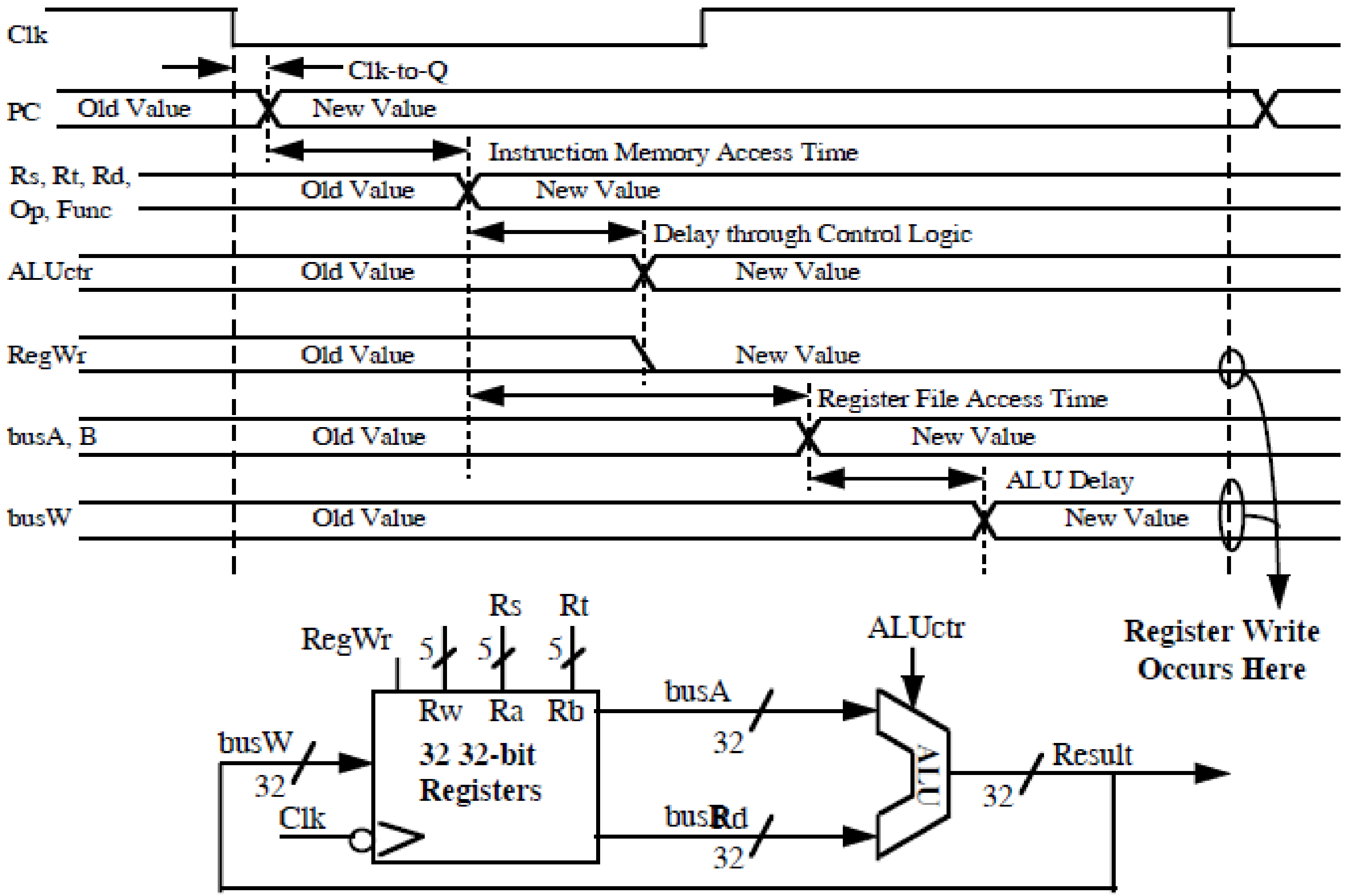
Datapath for Register-Register Operations

- R[rd] = R[rs] op R[rt] Example: add rd, rs, rt
  - Ra, Rb, and Rw comes from instruction's rs, rt, and rd fields
  - ALUctr and RegWr: control logic after decoding the instruction

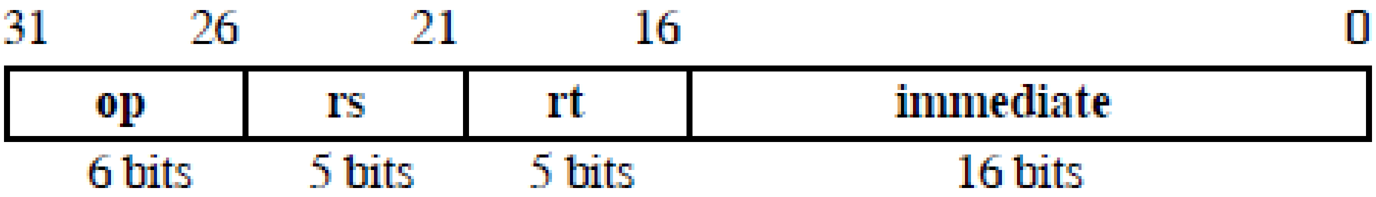
31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



Register – Register Timing



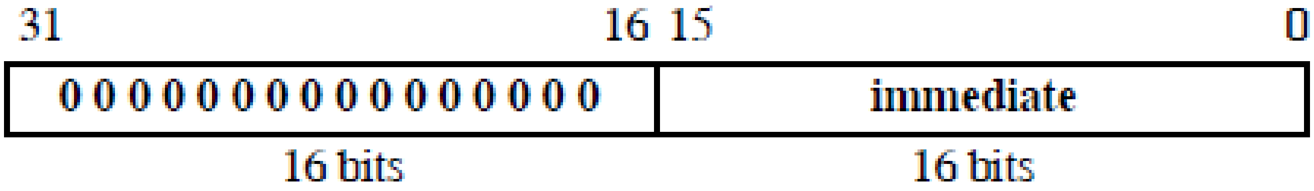
RTL: The OR Immediate Instruction



31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- ori rt, rs, imm16
  - mem[PC] Fetch the instruction from memory
- R[rt] = R[rs] or ZeroExt(imm16) The OR operation
- PC = PC + 4 Calculate the next instruction's address



31 16 15 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	immediate
16 bits	16 bits

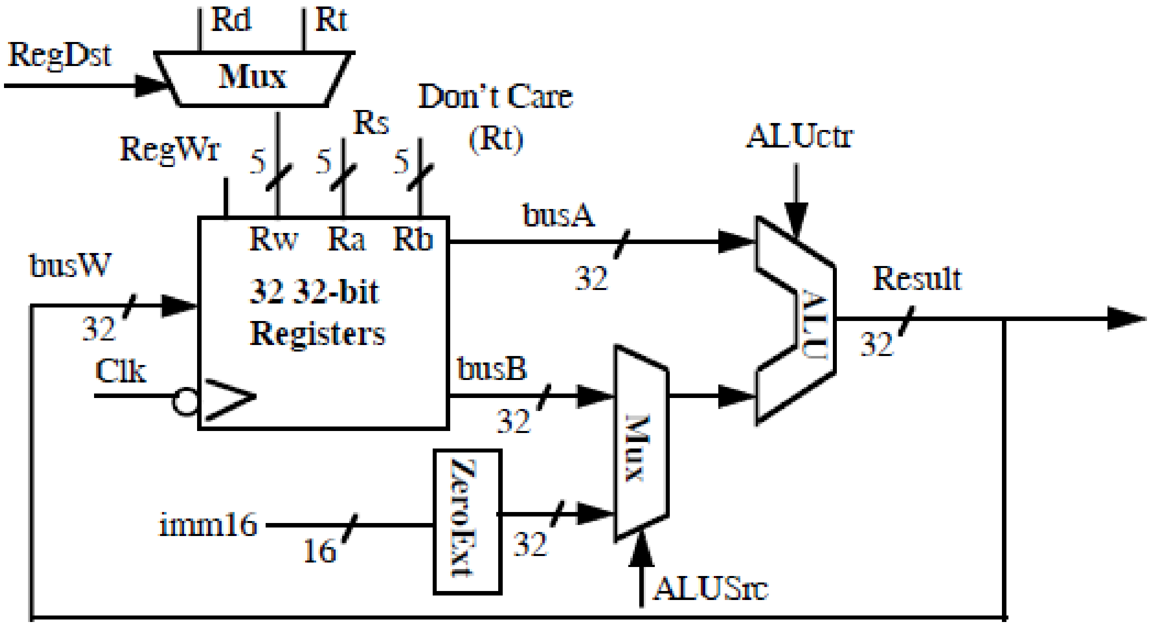
Datapath for Logical Operations with Immediate

- R[rt] = R[rs] op ZeroExt[imm16]] Example: ori rt, rs, imm16

31 26 21 16 11 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

rd



RTL: The LOAD Instruction

31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- lw rd, rs, imm16
  - mem[pc] Fetch the instruction from the memory
  - Addr = R[rs] + SignExt(imm16) Calculate memory address
  - R[rt] = Mem[Addr] Load the data into the register
  - PC = PC + 4 Calculate the next instruction's address

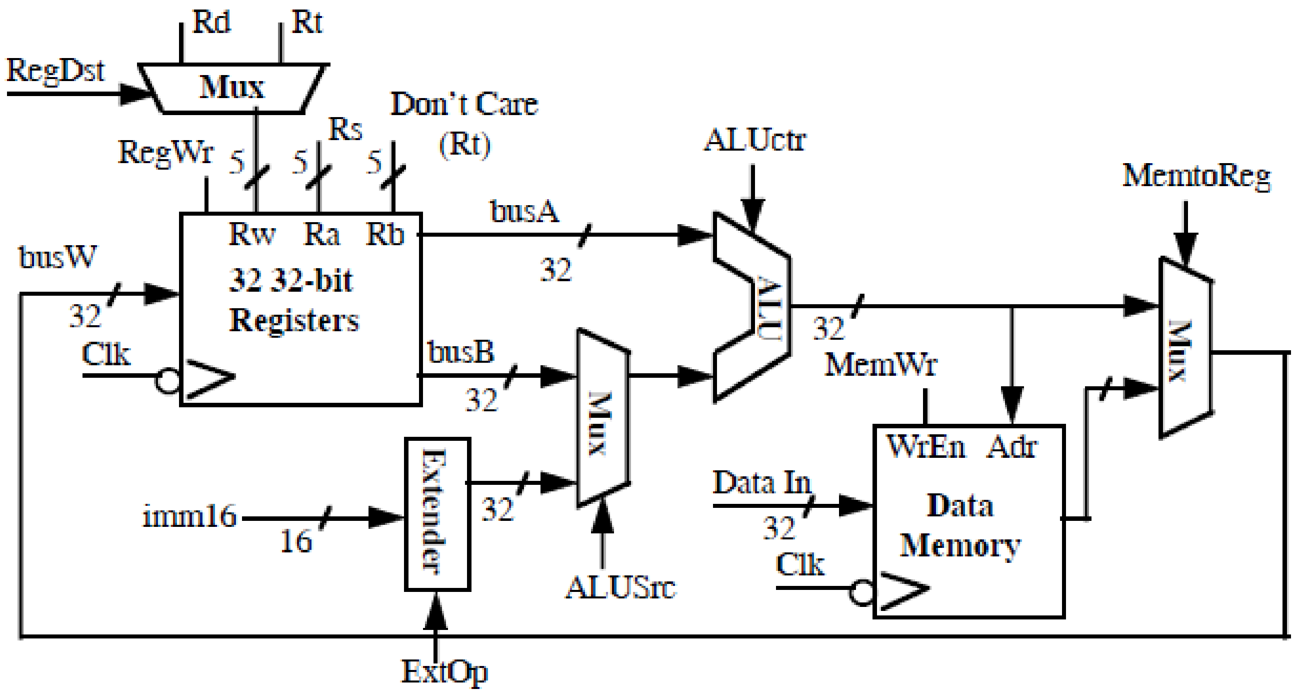
31 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 immediate
16 bits	16 bits

31 16	15 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 immediate
16 bits	16 bits

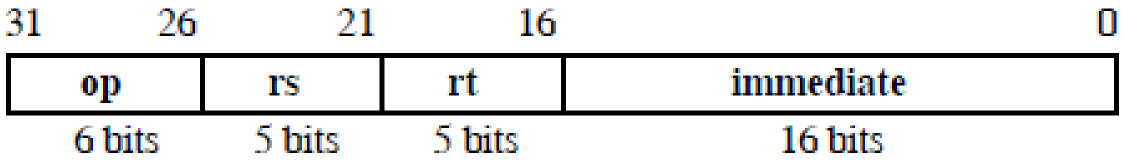
Datapath for Load Operations

- R[rt] = Mem[R[rs] + SignExt[imm16]] Example: lw rt, rs, imm16

31 26	21	16	11 0
op	rs	rt	immediate
6 bits	5 bits	5 bits	rd 16 bits



RTL: The Store Instruction

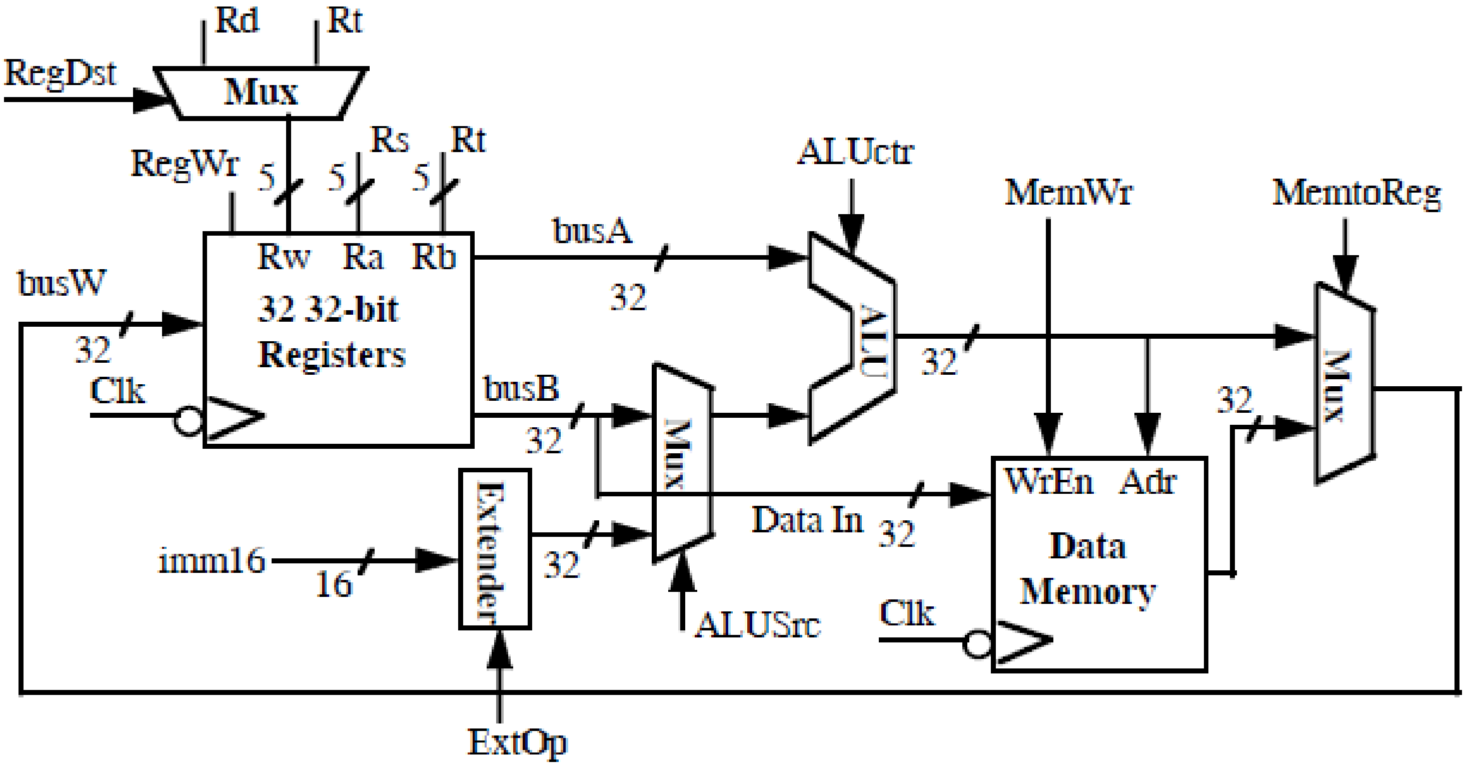


31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

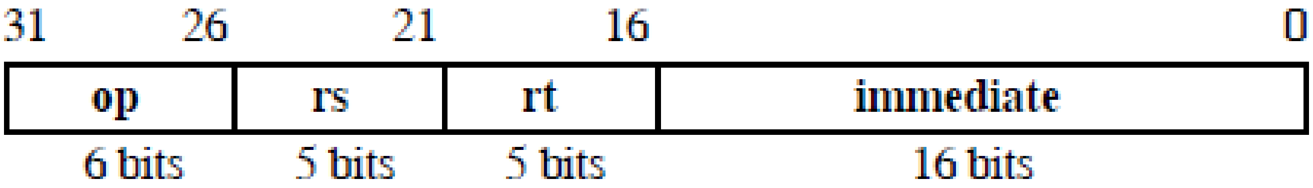
- sw rt, rs, imm16
  - mem[PC] Fetch the instruction from memory
  - Addr = R[rs] + SignExt(imm16) Calculate the memory address
  - Mem[Addr] = R[rt] Store the register into memory
  - PC = PC + 4 Calculate the next instruction’s address

Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16] = R[rt]] Example: sw rt, rs, imm16



RTL: Branch Instruction



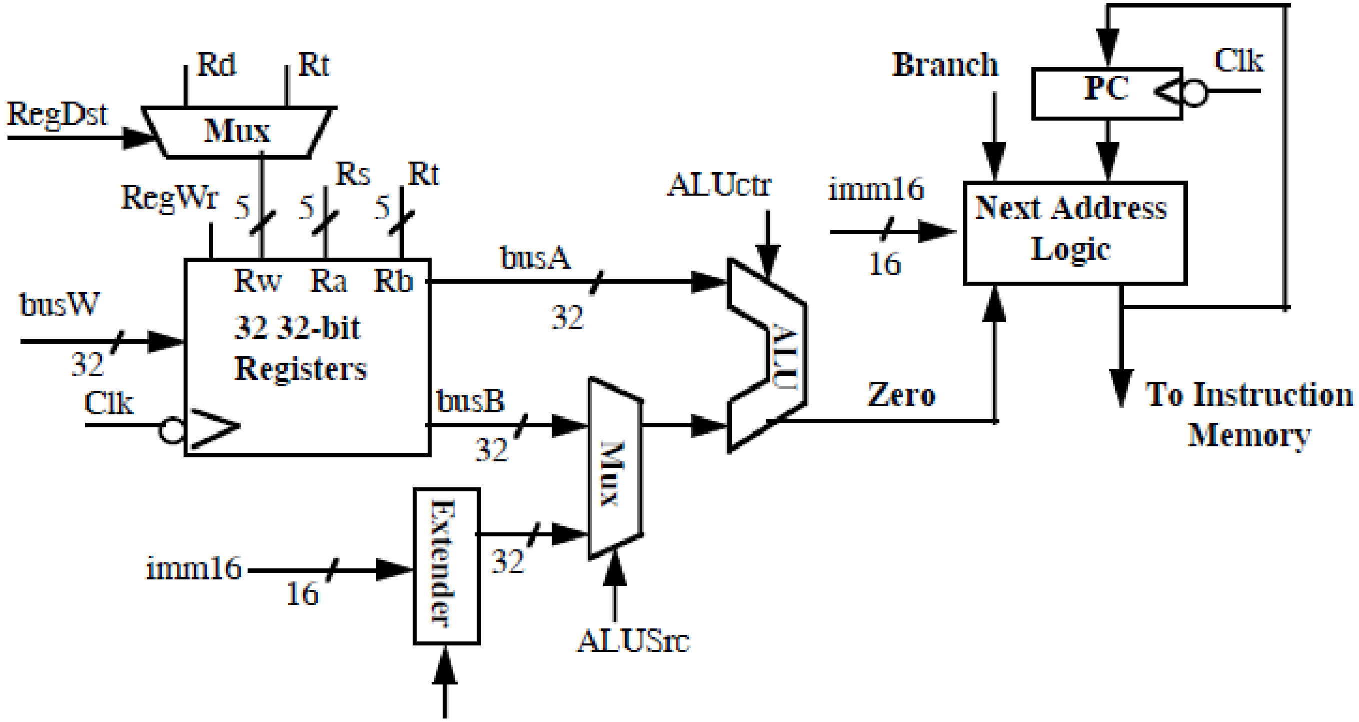
31 26	21	16	0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- beq rs, rt, imm16
  - mem[PC] Fetch the instruction from memory
  - Cond = R[rs] - R[rt] Calculate the branch condition
  - if (Cond == 0) Calculate the next instruction's address
    - PC = PC + 4 + (SignExt(imm16) x 4)
  - else
    - PC = PC + 4

Datapath for Branch Operations

- beq rs, rt, imm16 We need to compare Rs and Rt!

31	26	21	16	0
op	rs	rt	immediate	
6 bits	5 bits	5 bits	16 bits	



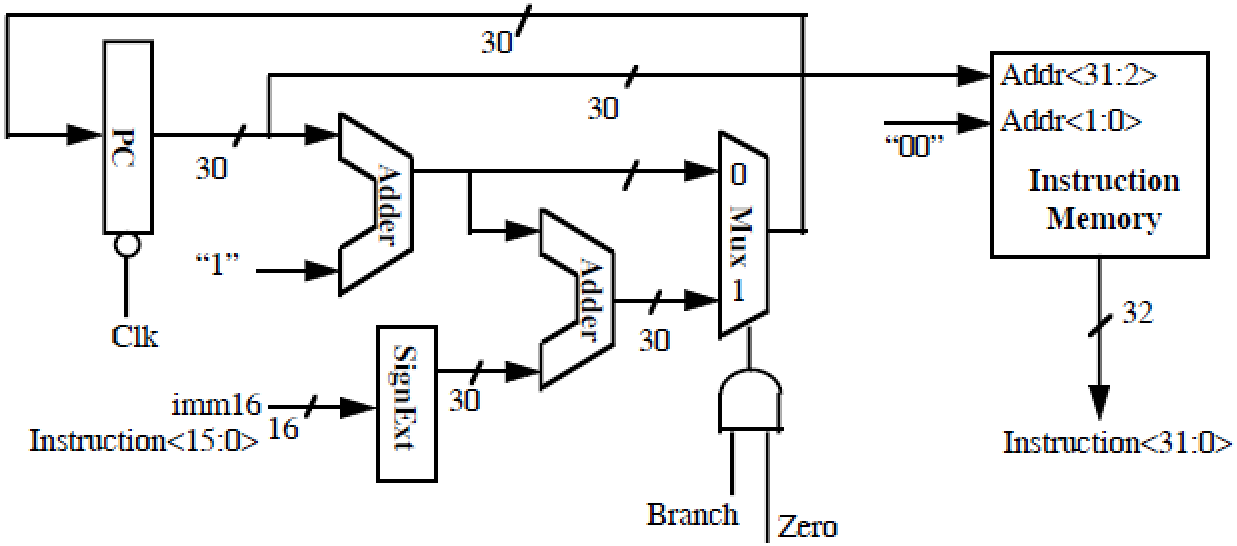
Binary Arithmetic for Next Address

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: PC<31:0> = PC<31:0> + 4
  - Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt30[Imm16] * 4
- However:
  - The 2 LSBs of the 32-bit PC are always zeros (why?)
  - There is no reason to have hardware to keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - In either case: Instruction Memory Address = PC<31:2> concat “00”

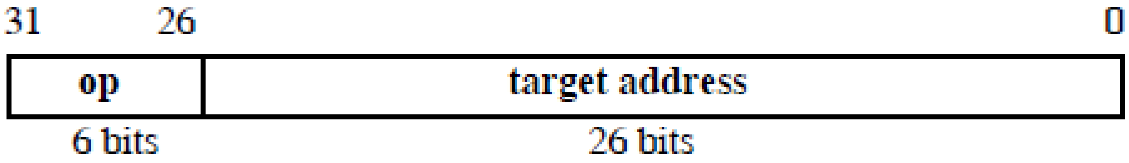
Next Address Logic

Using a 30-bit PC:

- Sequential operation: PC<31:2> = PC<31:2> + 1
- Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
- In either case: Instruction Memory Address = PC<31:2> concat “00”



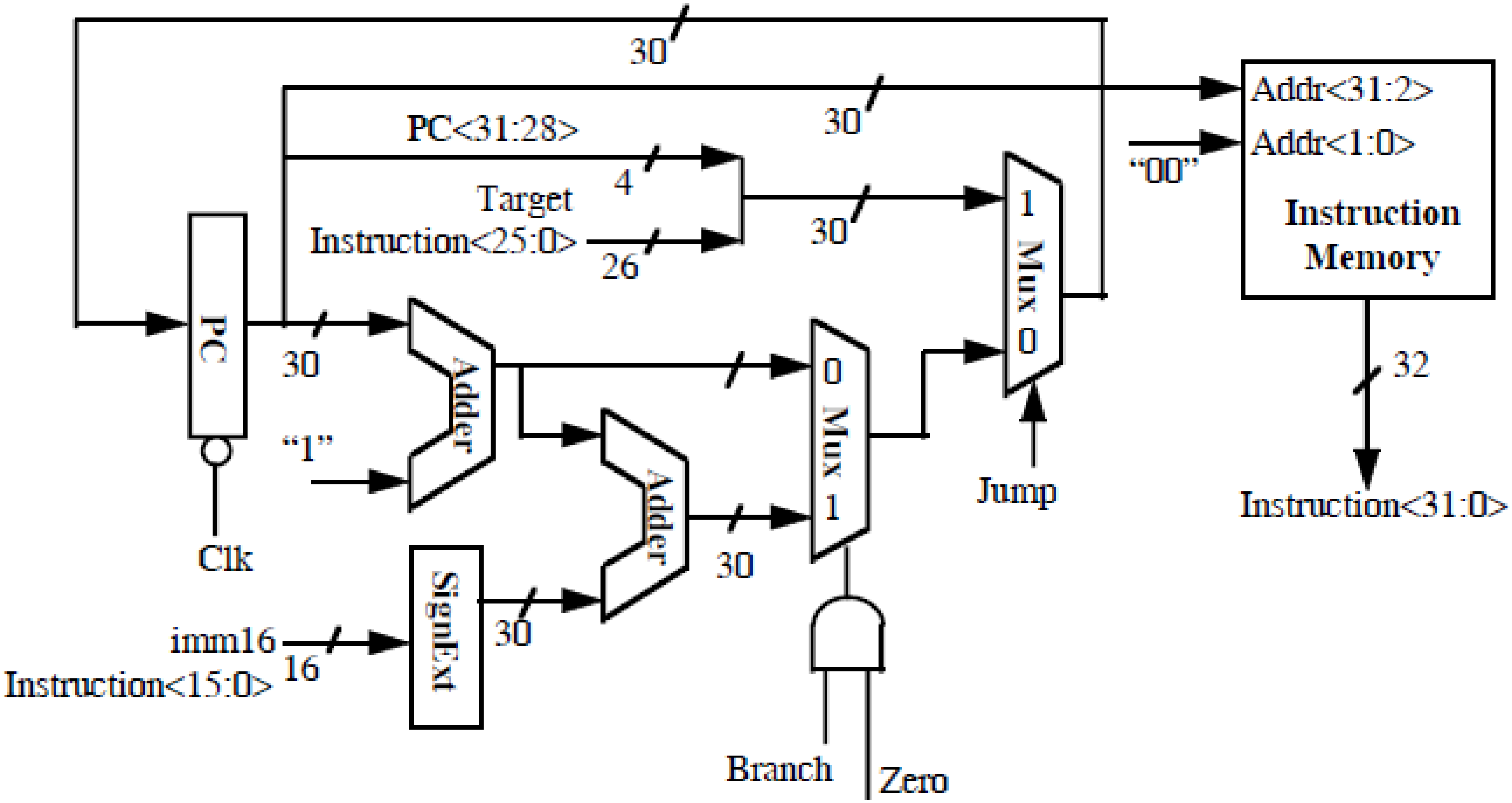
RTL: The Jump Instruction



- j target
  - mem[PC] Fetch the instruction from memory
  - PC<31:2> = PC<31:28> concat target<25:0>

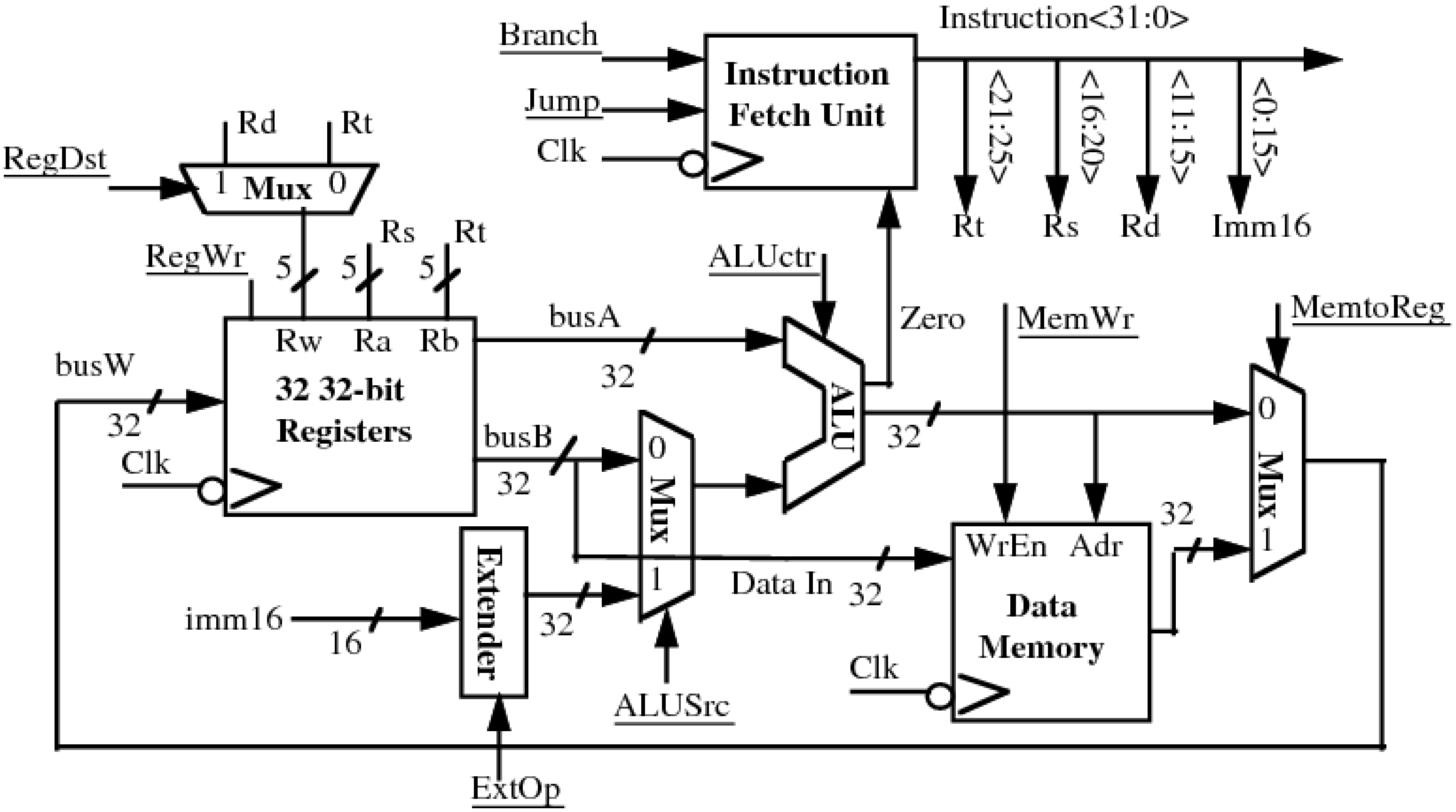
    (Calculate the next instruction's address)

Final Instruction Instruction Fetch Unit

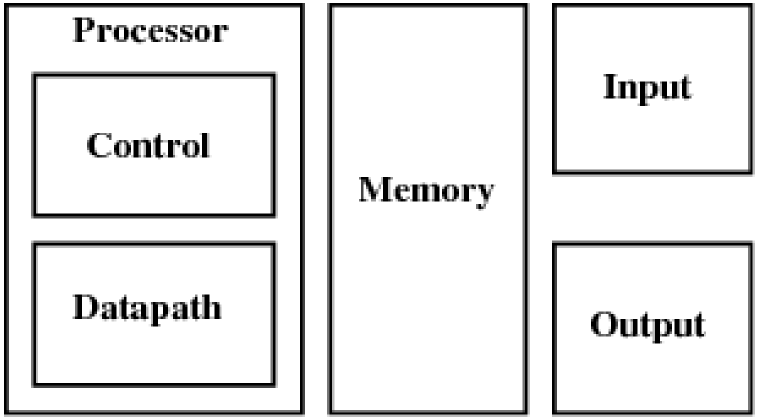


Putting it all together: A Single Cycle Datapath

We have everything except control signals (underline)



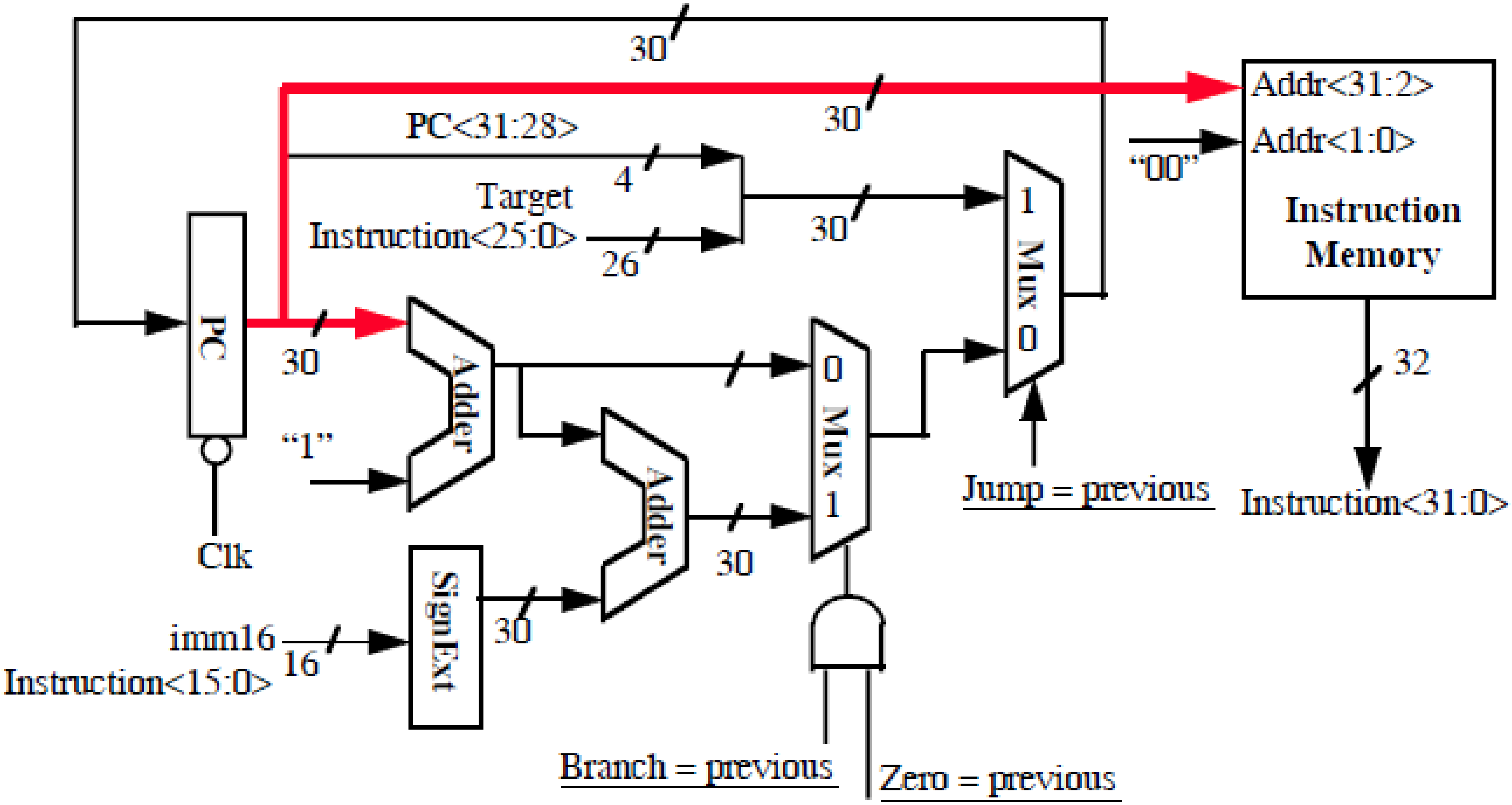
The Big Picture: Where are we Now?



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

Instruction Fetch Unit at the Beginning of Add / Subtract

- Fetch the instruction from Instruction memory: Instruction = mem[PC]
  - This is the same for all instructions

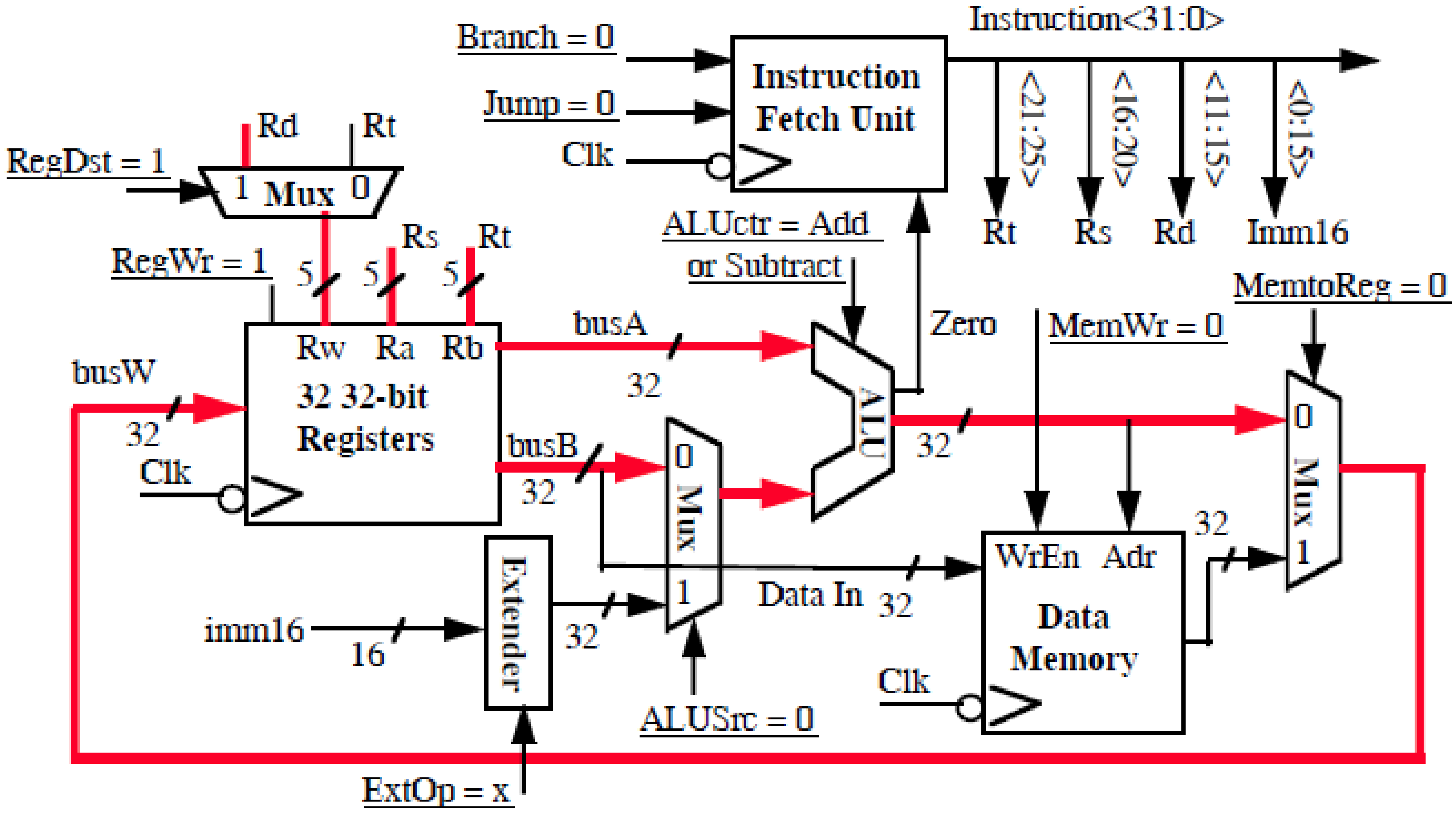


The Single Cycle Datapath during Add and Subtract

31 26 21 16 11 6 0

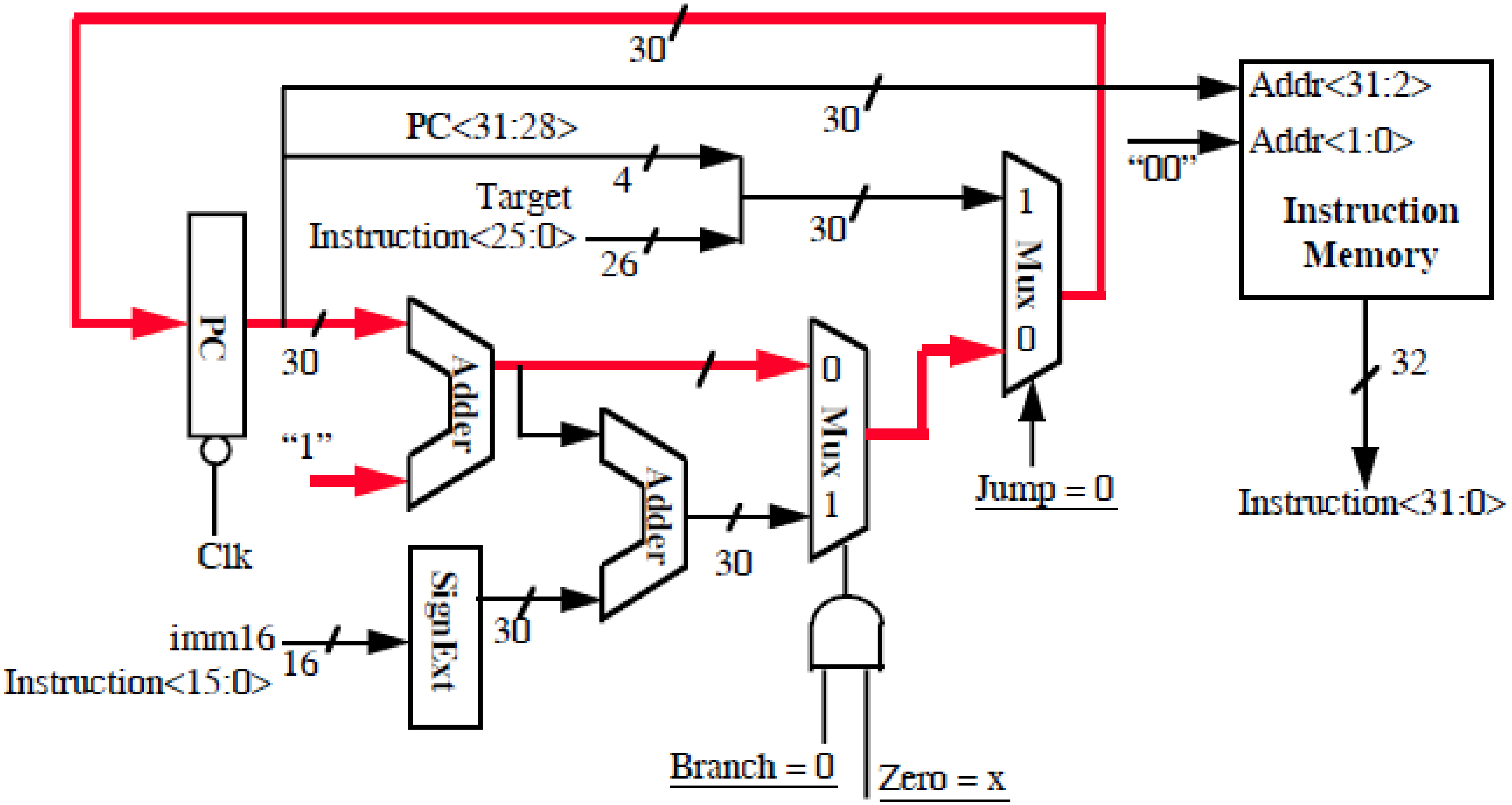
op	rs	rt	rd	shamt	funct

- R[rd] <- 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

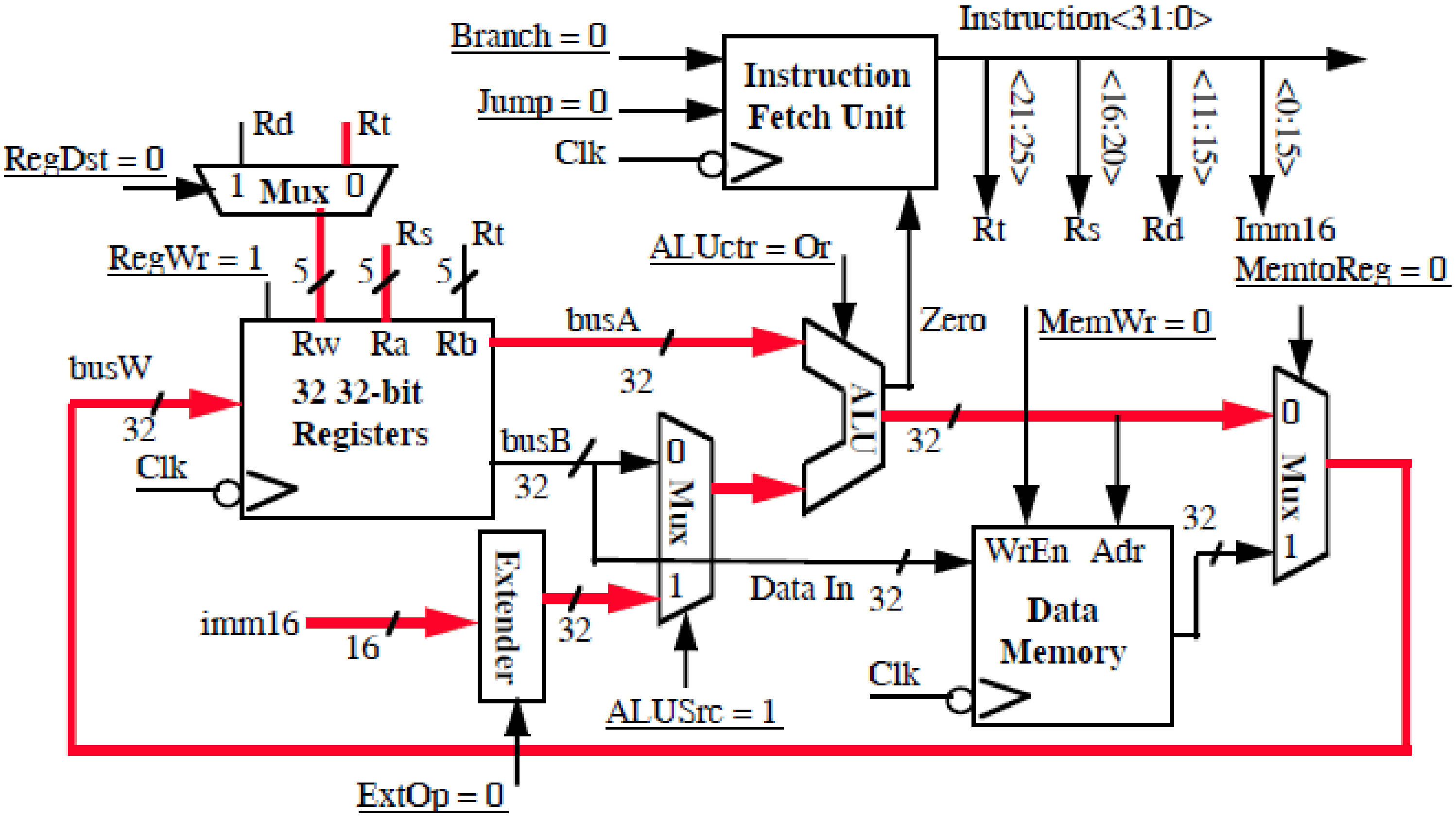


The Single Cycle Datapath during Or Immediate

31 26 21 16 0

op	rs	rt	immediate

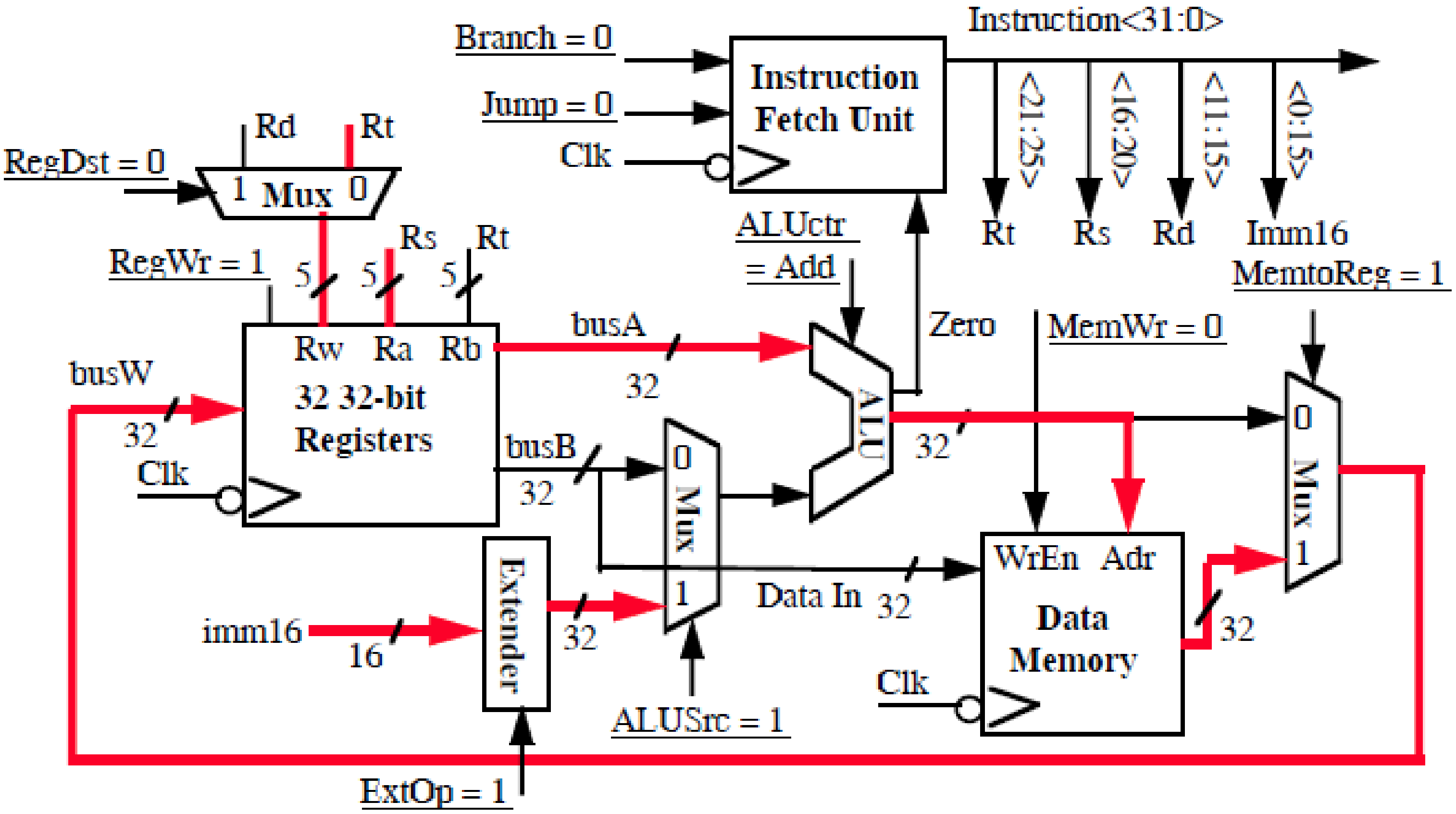
- R[rt] <- R[rs] or ZeroExt[Imm16]



The Single Cycle Datapath during Load

31	26	21	16 0
op	rs	rt	immediate

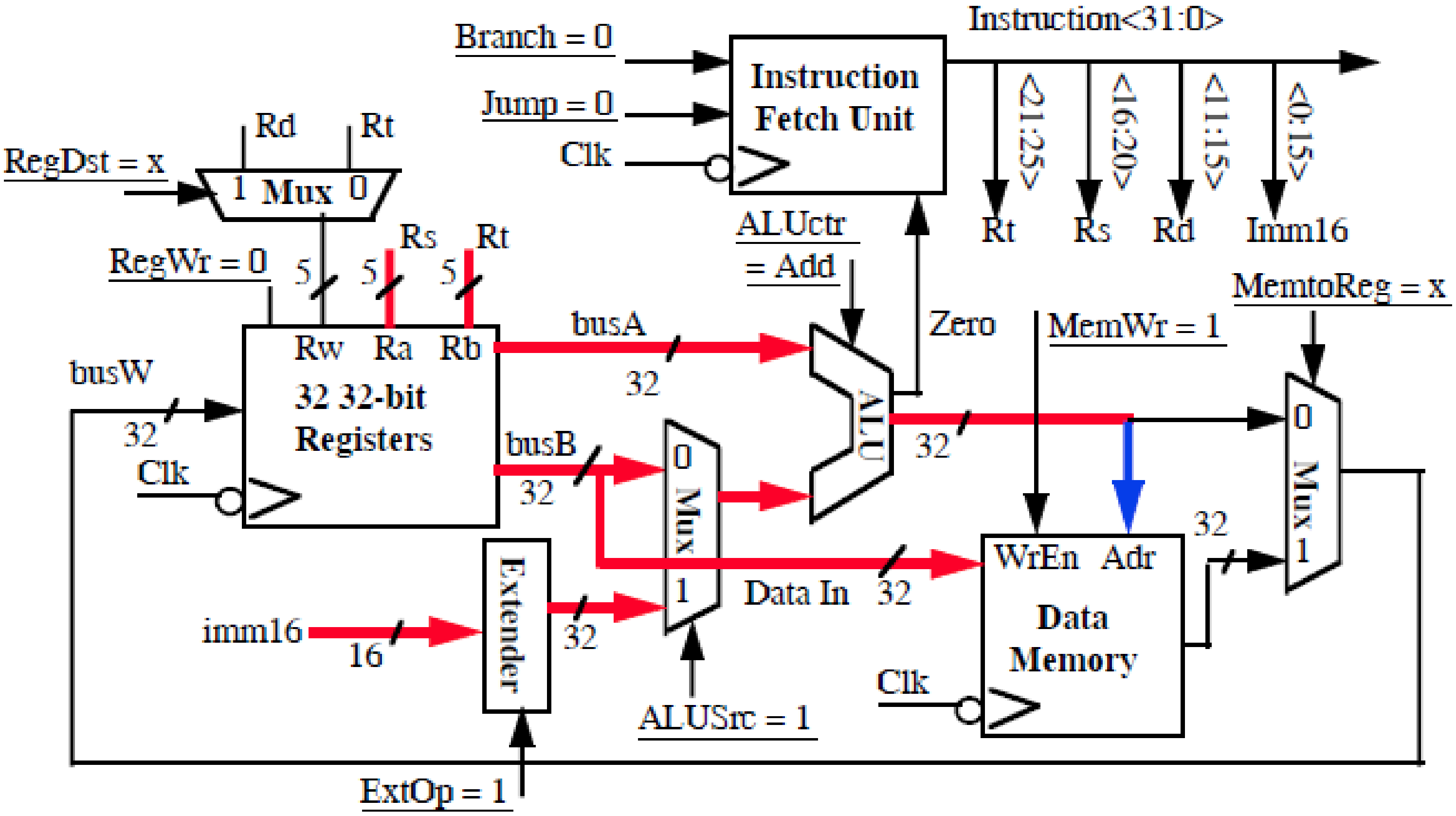
- R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



The Single Cycle Datapath during Store

31	26	21	16	0
op	rs	rt	immediate	

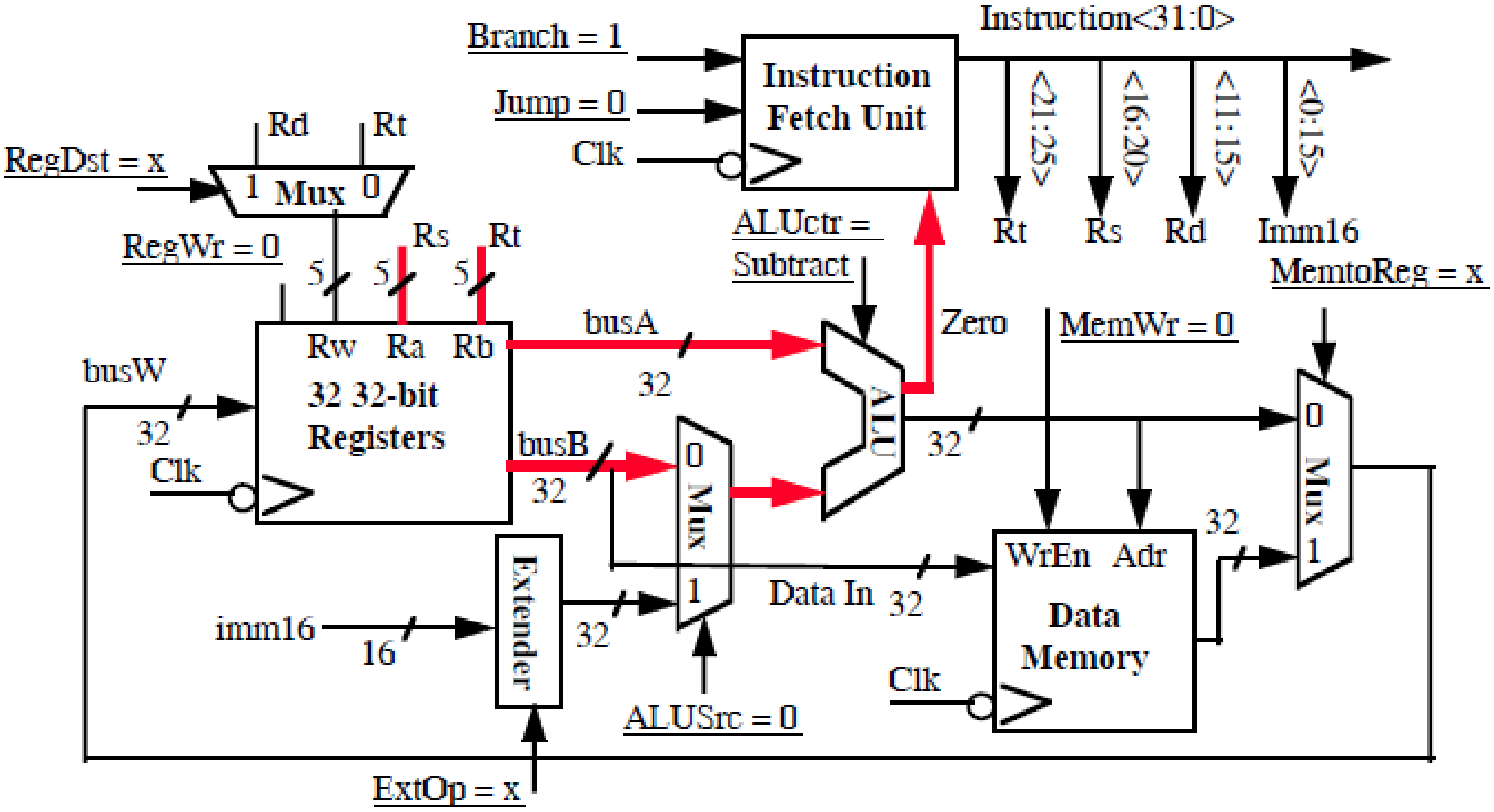
- Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



The Single Cycle Datapath during Branch

31	26	21	16	0
op	rs	rt	immediate	

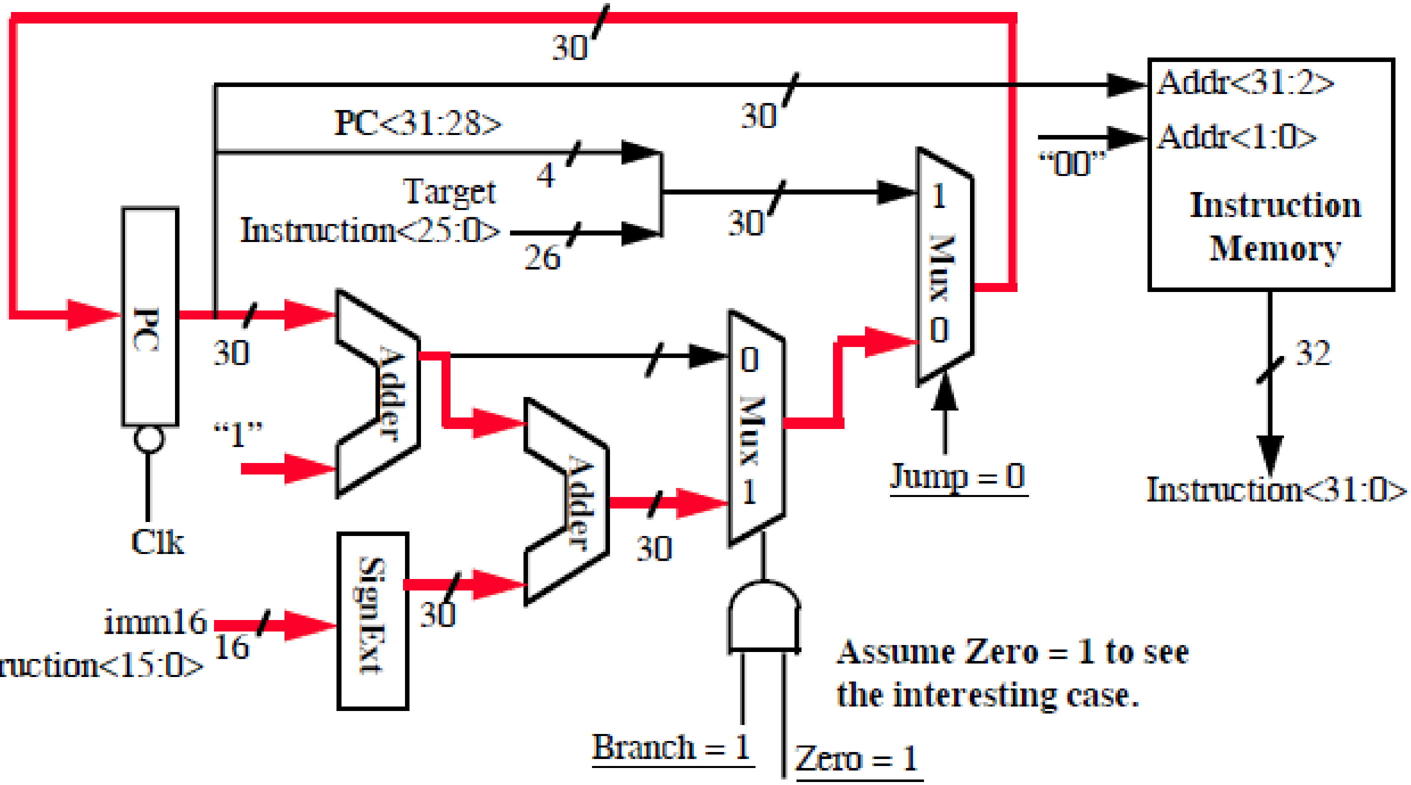
- if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



Instruction Fetch Unit at the End of Branch

31	26	21	16	0
op	rs	rt	immediate	

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

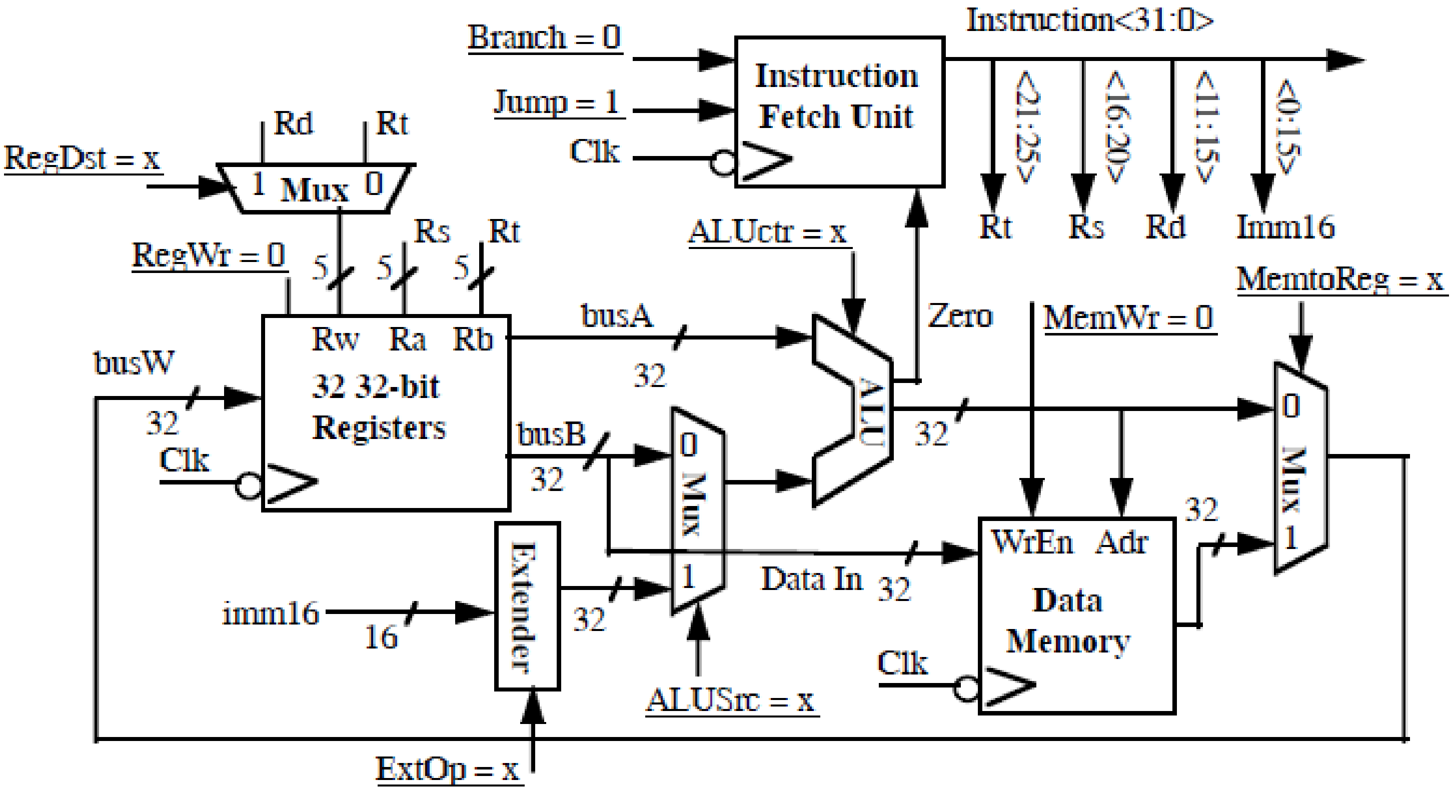


Assume Zero = 1 to see the interesting case.

The Single Cycle Datapath during Jump

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

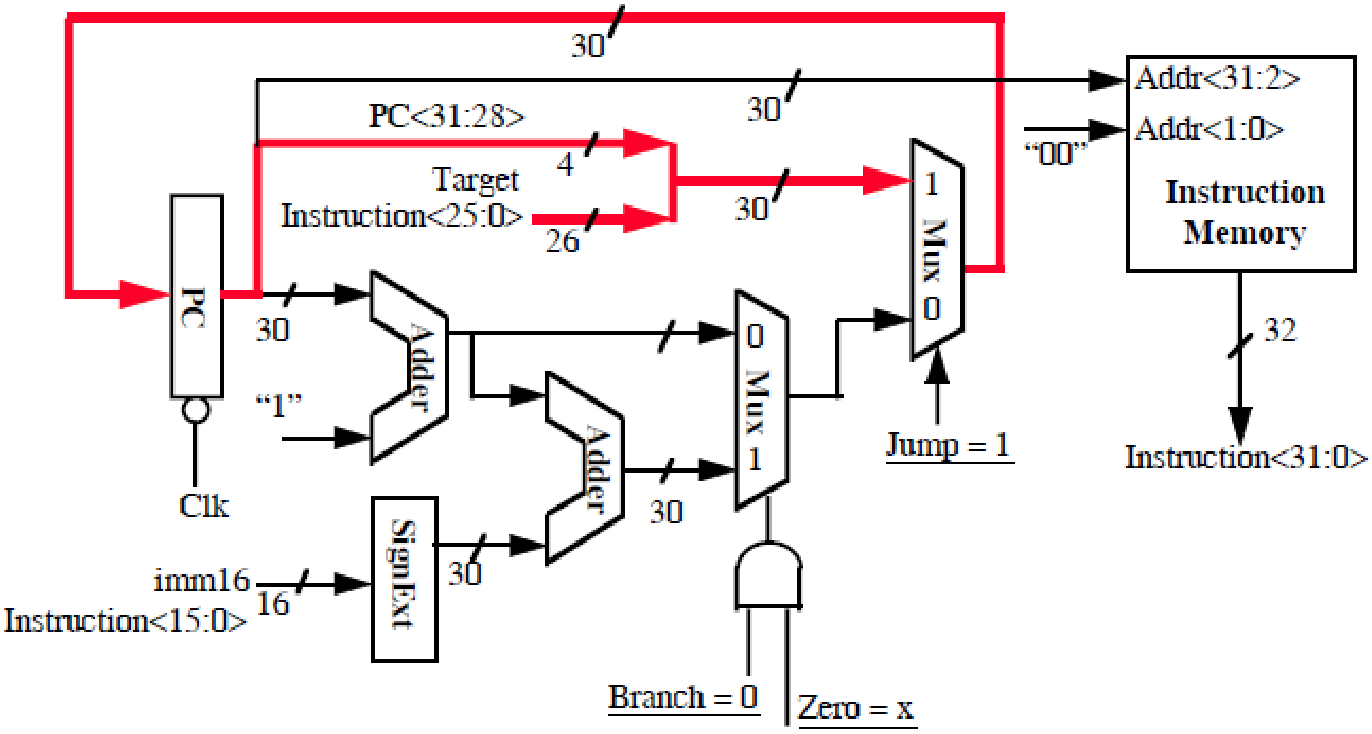
31 26	0
op	target address



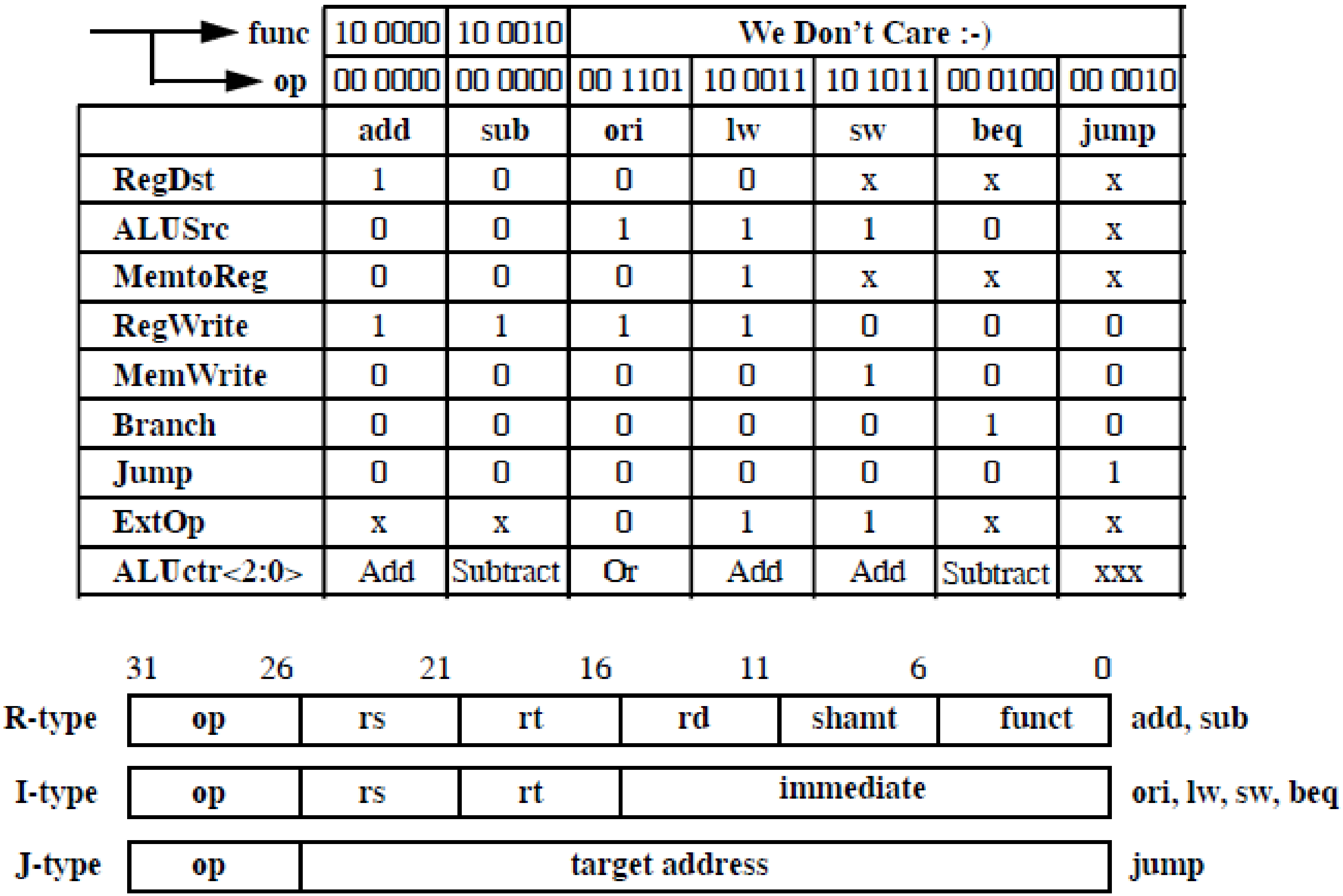
Instruction Fetch Unit at the End of Jump

31 26	0
op	target address

- PC <- PC<31:29> concat target<25:0> concat "00"



A Summary of Control Signals

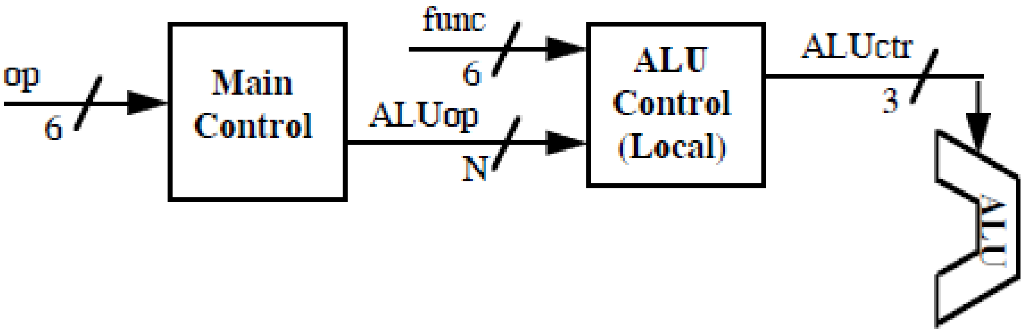


func	10 0000	10 0010	We Don't Care :-)				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	0	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
Branch	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x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

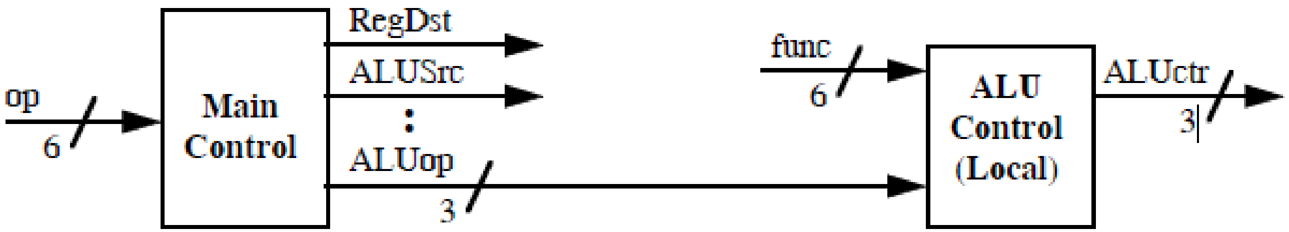
The Concept of Local Decoding

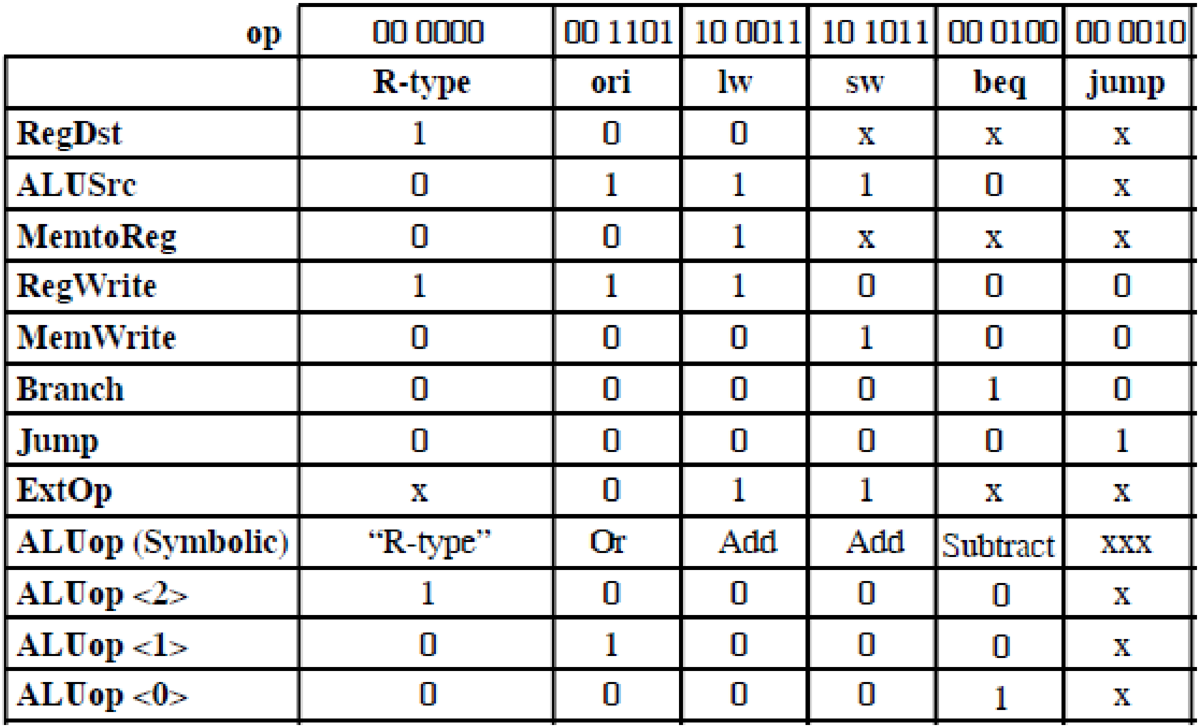
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



The “Truth Table” for the Main Control

Key Idea: Two levels of Control logic.

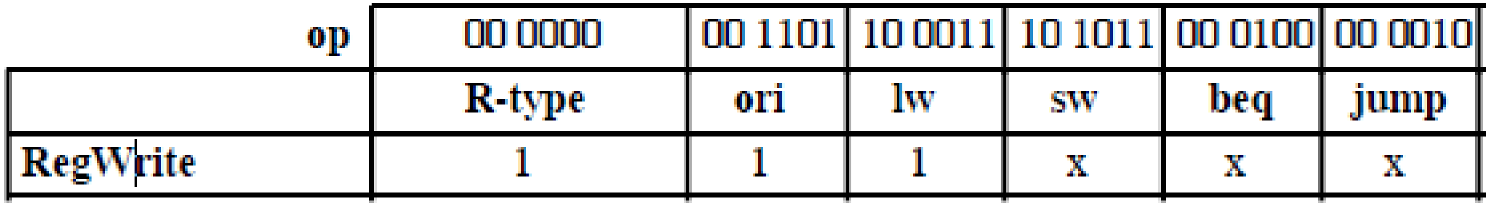




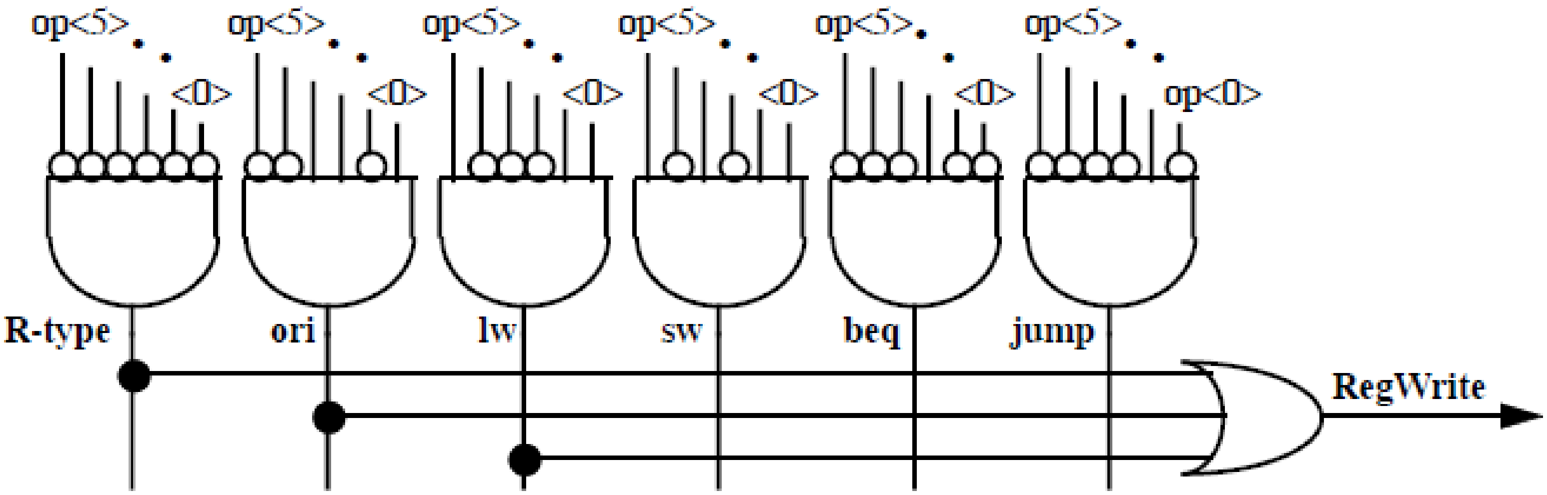
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0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

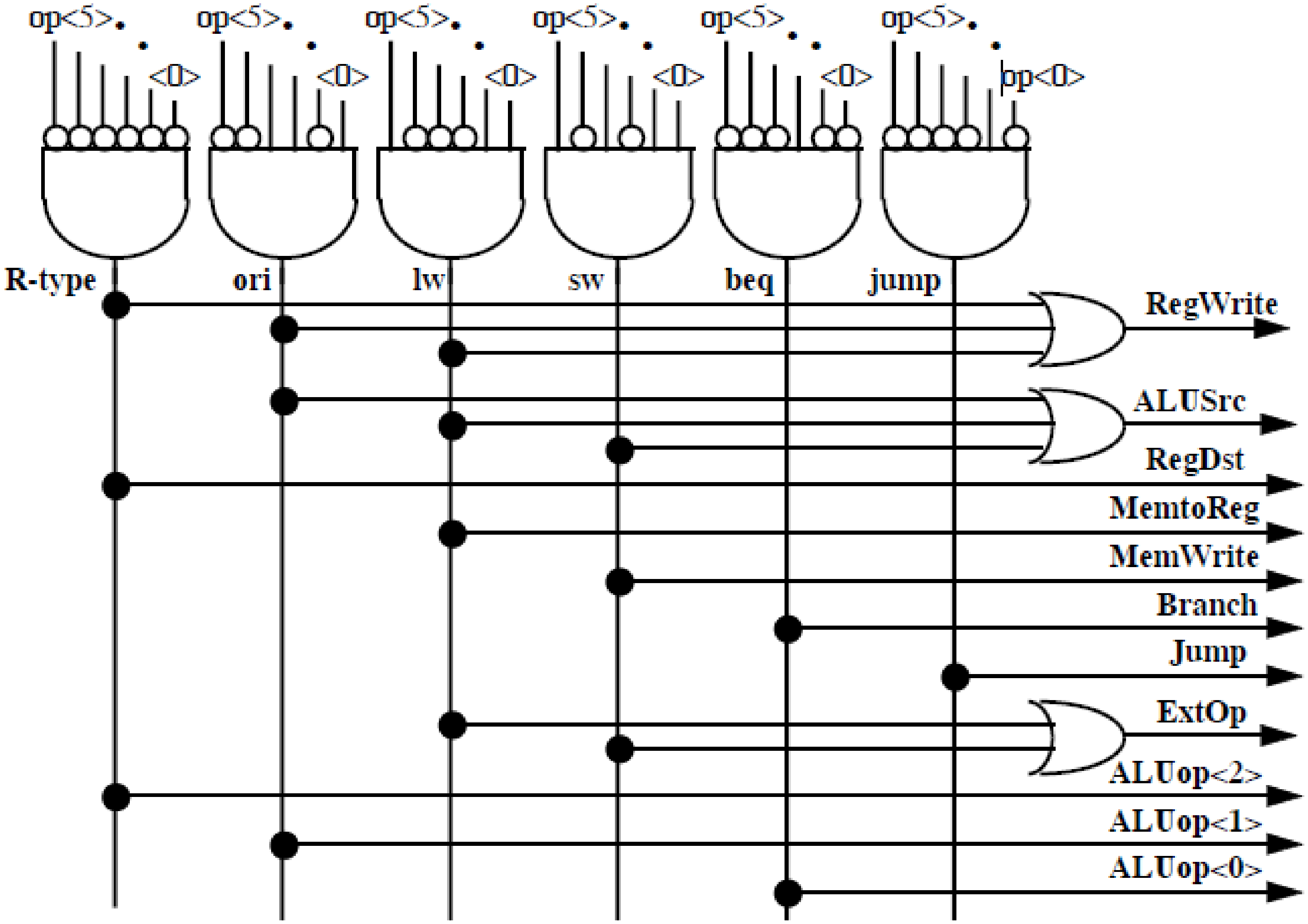


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	x	x	x

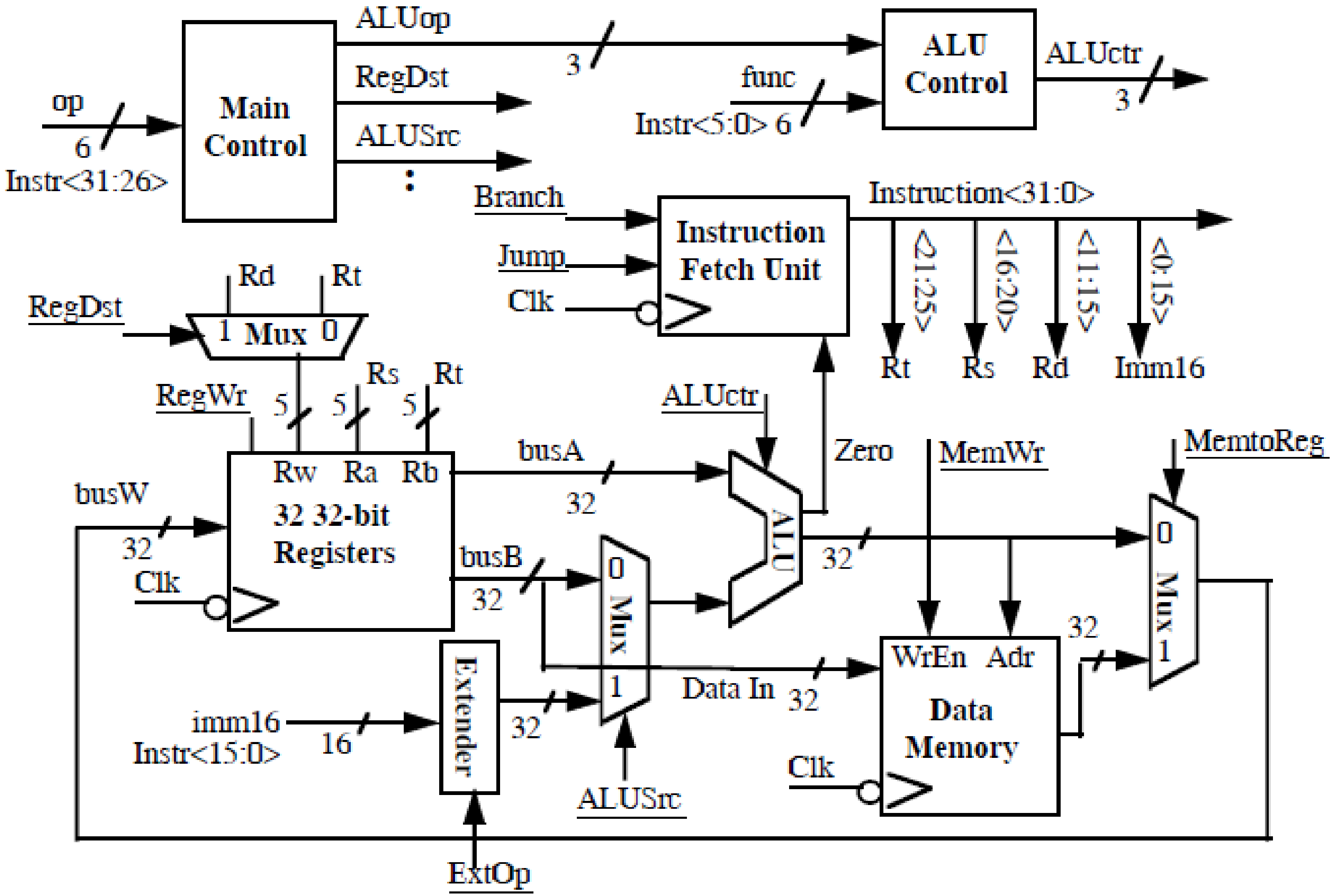


PLA Implementation of Main Control

Hmm! What is PLA?



Putting it All Together: A Single Cycle Processor



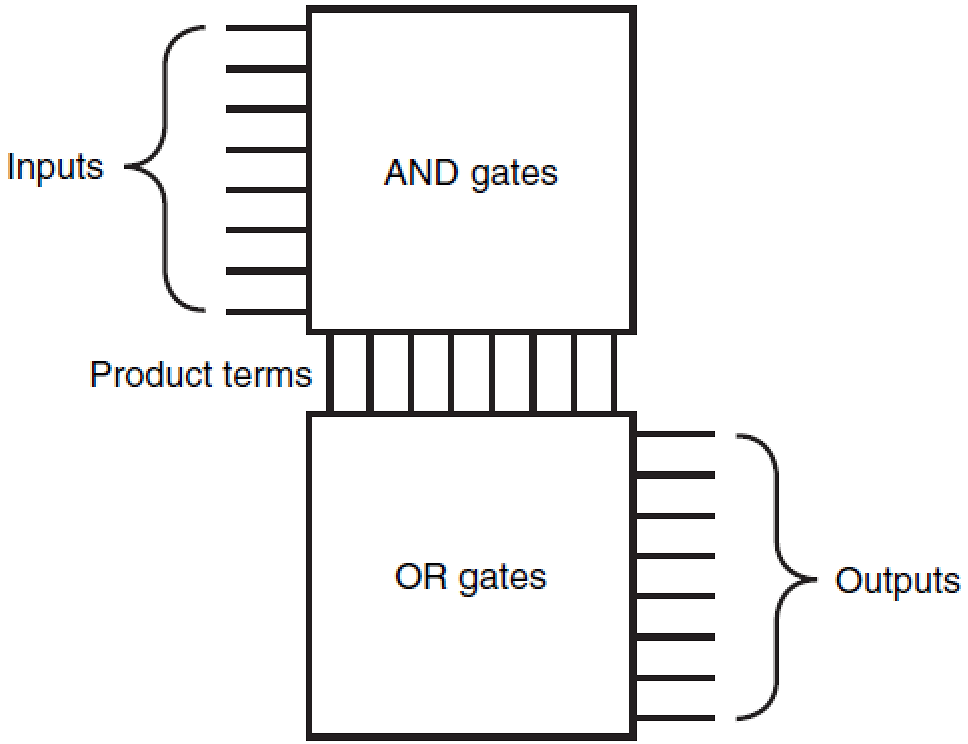
Drawback of this Single Cycle Processor

- 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

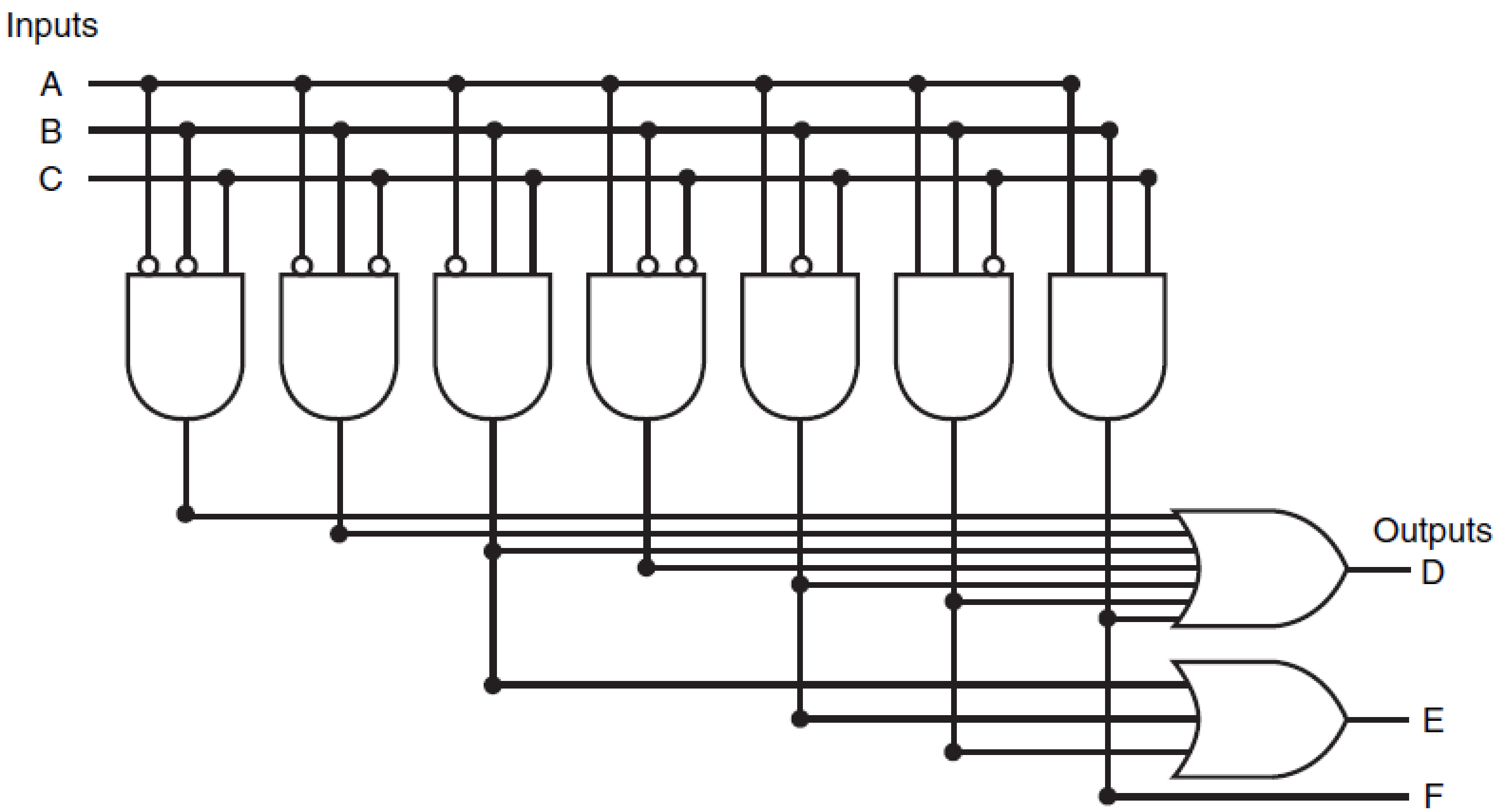
We are assuming Clock Skew is zero

Programmable Logic Arrays

- PLAs can be used to realize combinational circuits



PLAs


Inputs
A
B
C
Outputs
D
E
F