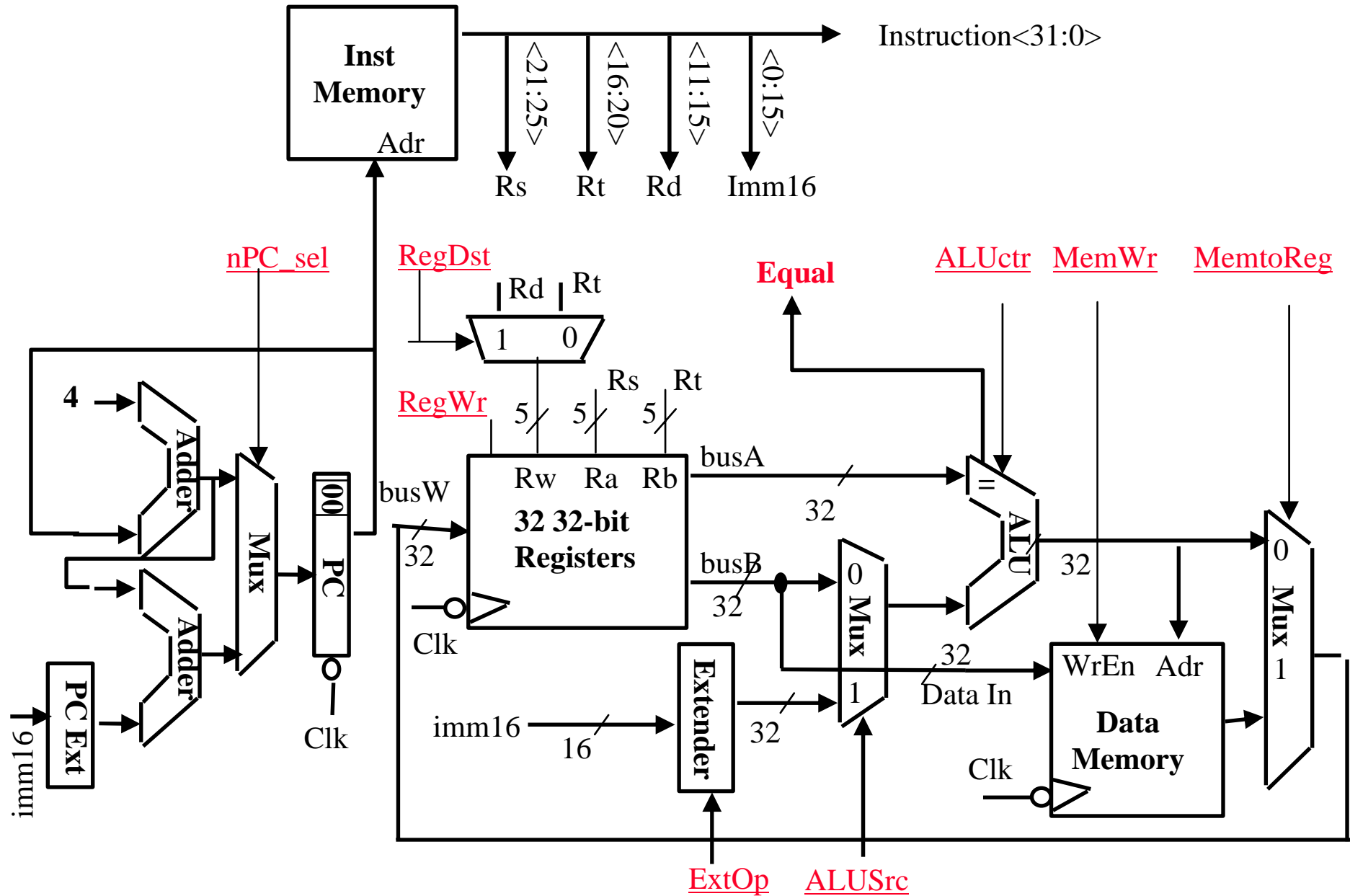


Processor Design - 2

Computer Architecture

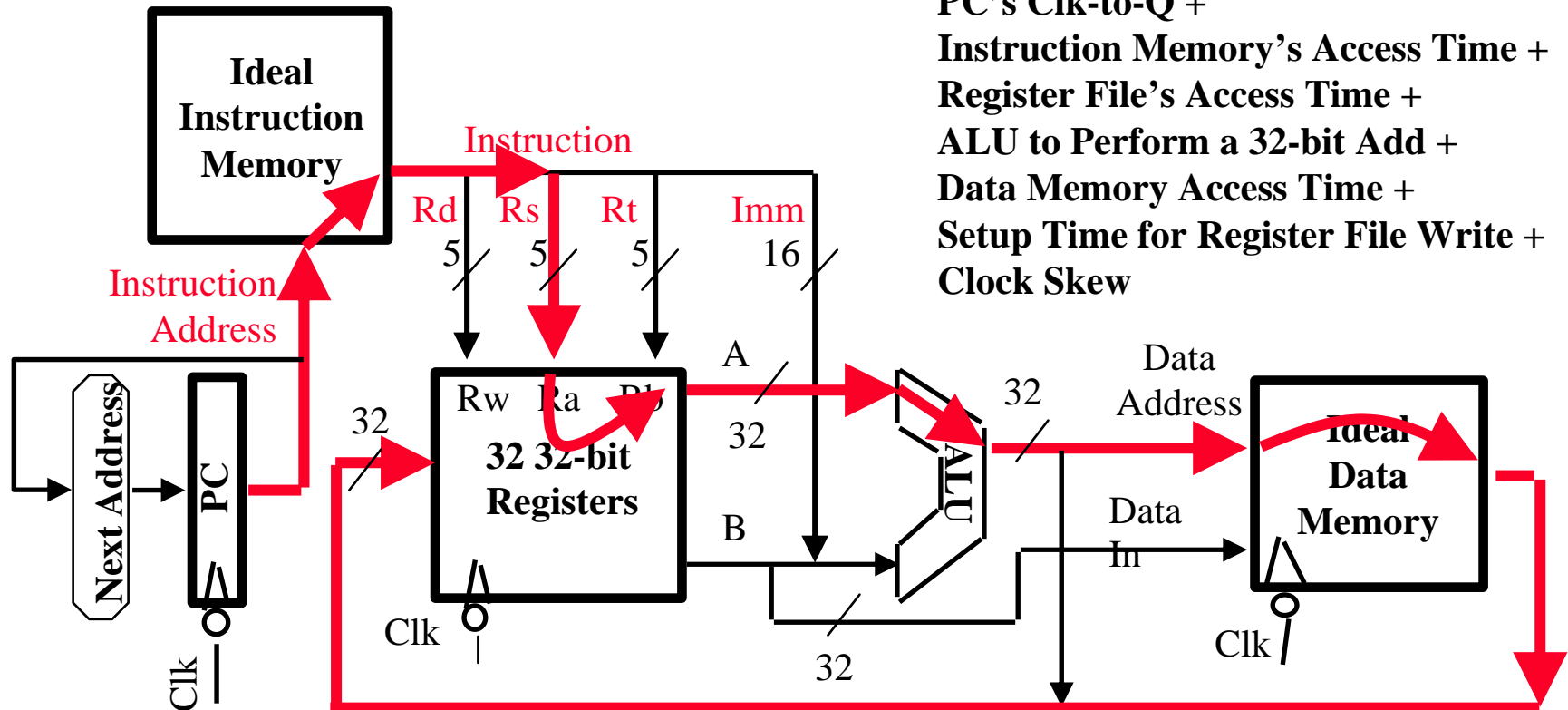
Summary: A Single Cycle Datapath



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 a 32-bit Add +

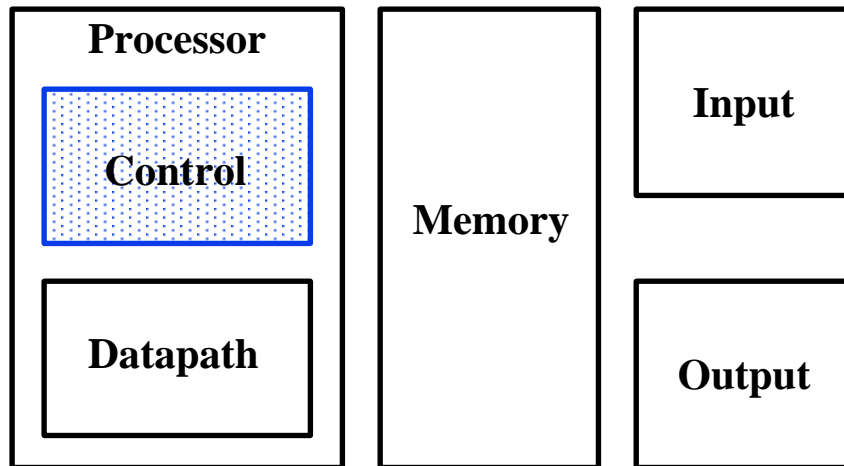
Data Memory Access Time +

Setup Time for Register File Write +

Clock Skew

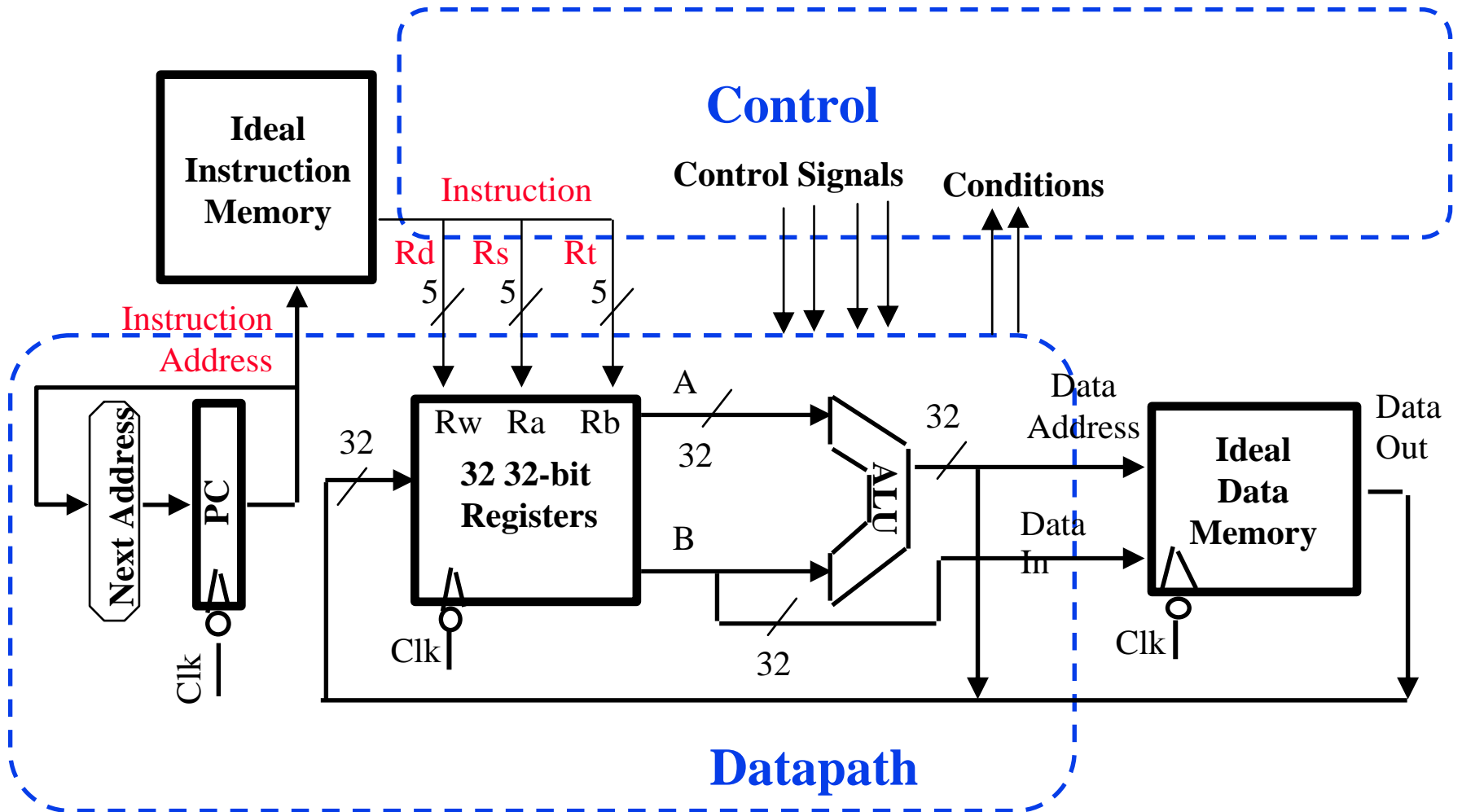
The Big Picture: Where are We Now?

- **The Five Classic Components of a Computer**



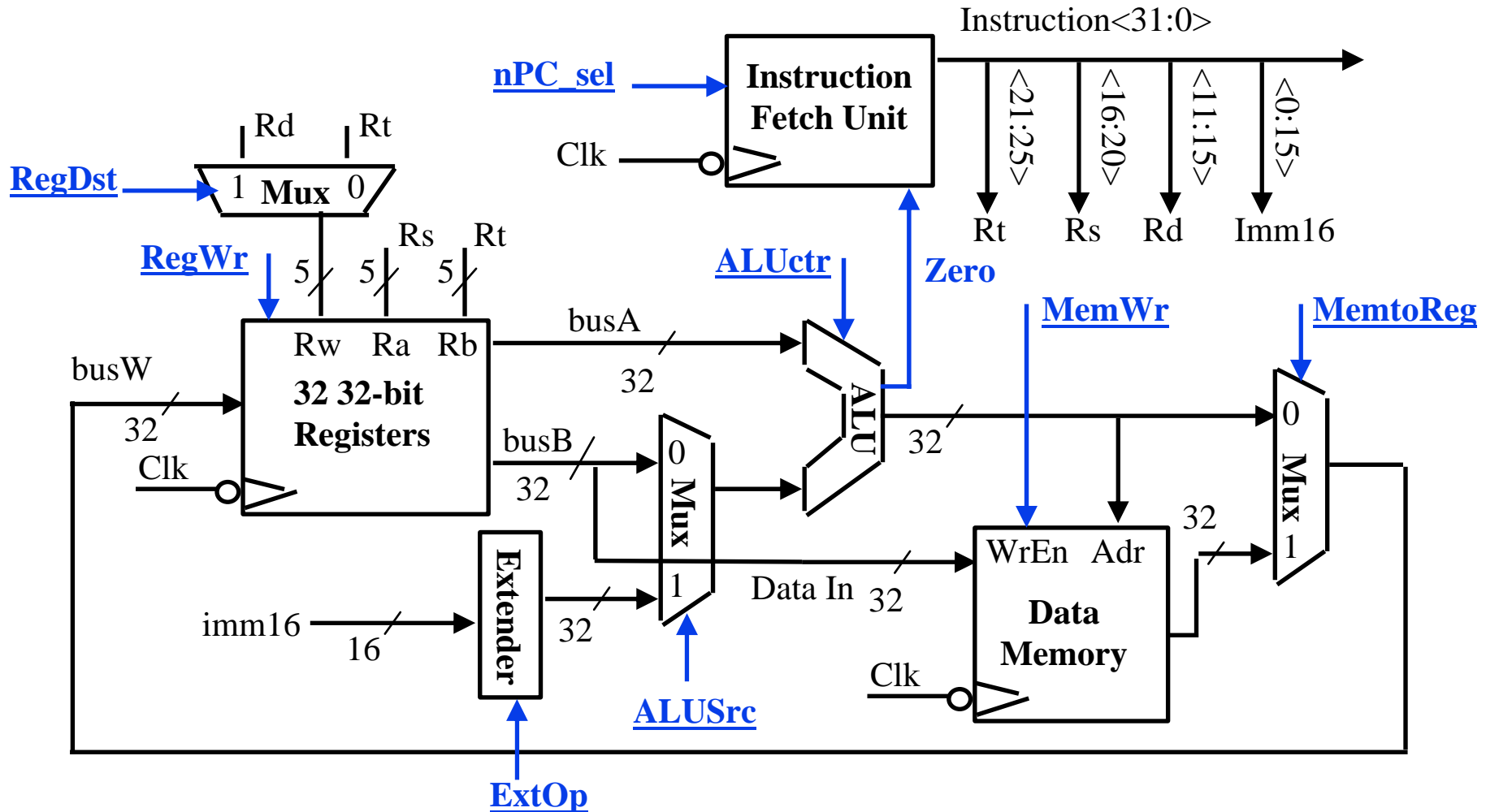
- **Next Topic: Designing the Control for the Single Cycle Datapath**

An Abstract View of the Implementation



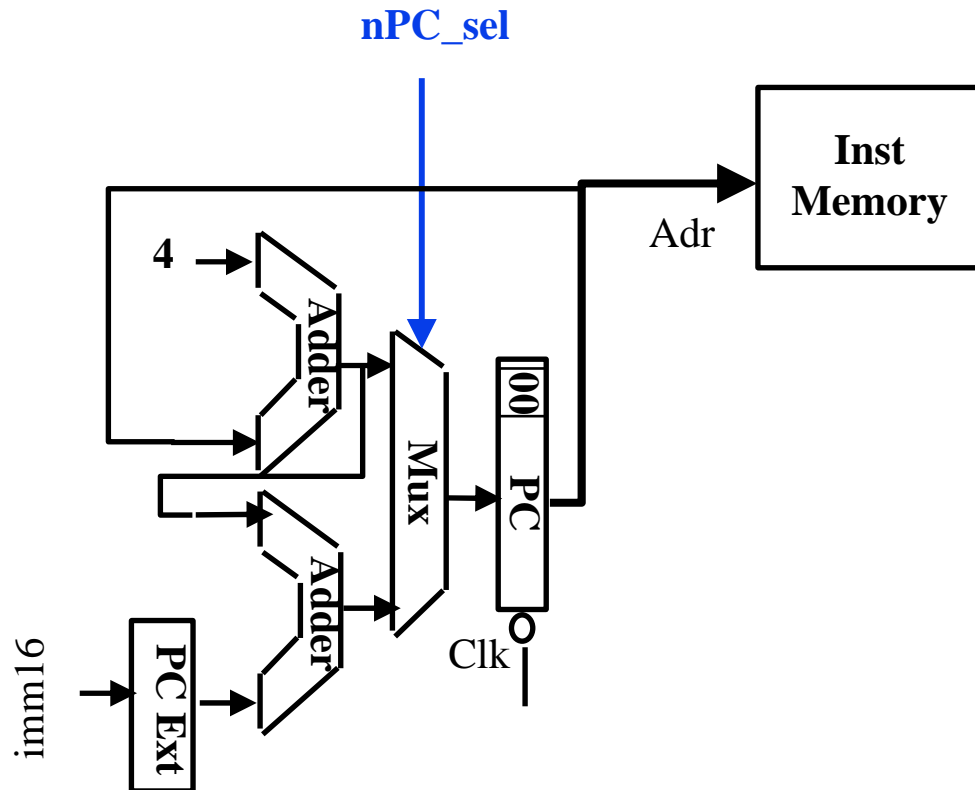
Recap: A Single Cycle Datapath

- Rs, Rt, Rd and Imed16 hardwired into datapath from Fetch Unit
- We have everything except control signals (underline)
 - Today's lecture will show you how to generate the control signals



Recap: Meaning of the Control Signals

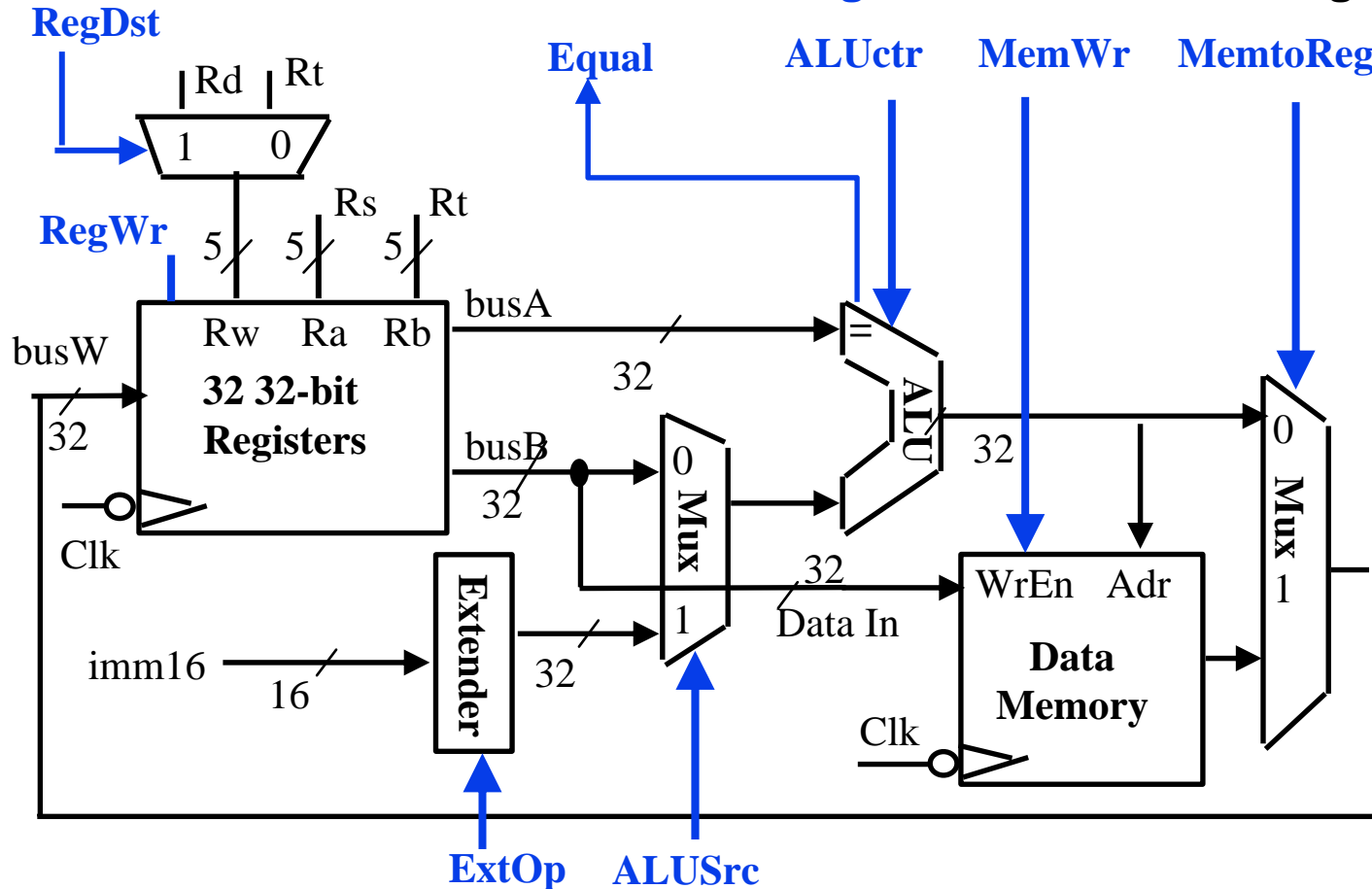
- **nPC_sel:** 0 \Rightarrow PC \leftarrow PC + 4
 1 \Rightarrow PC \leftarrow PC + 4 + SignExt(Im16) || 00
- Later in lecture: higher-level connection between mux and branch cond



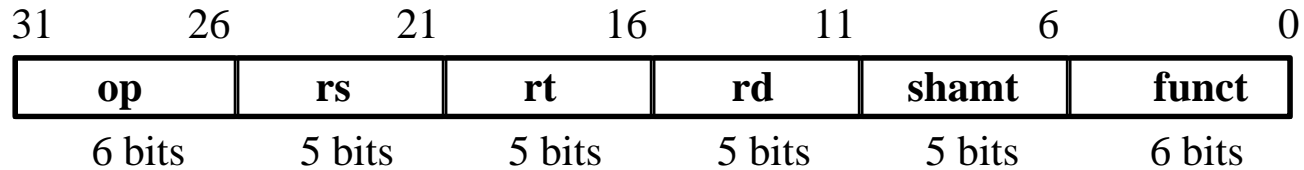
Recap: Meaning of the Control Signals

- **ExtOp:** “zero”, “sign”
- **ALUSrc:** 0 \Rightarrow regB; 1 \Rightarrow immedi
- **ALUctr:** “add”, “sub”, “or”

- **MemWr:** 1 \Rightarrow write memory
- **MemtoReg:** 0 \Rightarrow ALU; 1 \Rightarrow Mem
- **RegDst:** 0 \Rightarrow “rt”; 1 \Rightarrow “rd”
- **RegWr:** 1 \Rightarrow write register



The add Instruction



• **add rd, rs, rt**

mem[PC]

Fetch the instruction from memory

$R[rd] \leftarrow R[rs] + R[rt]$

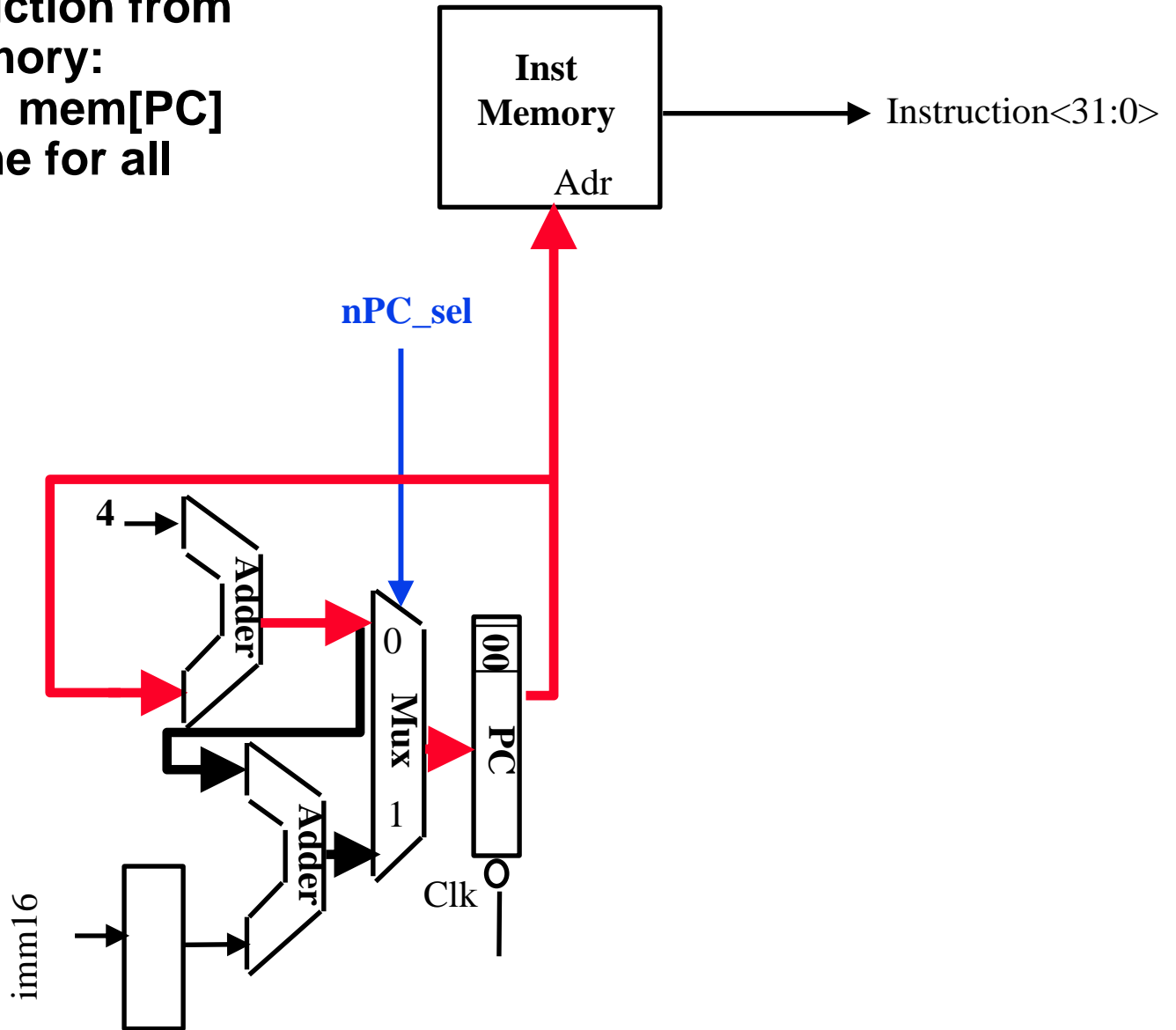
The actual operation

$PC \leftarrow PC + 4$

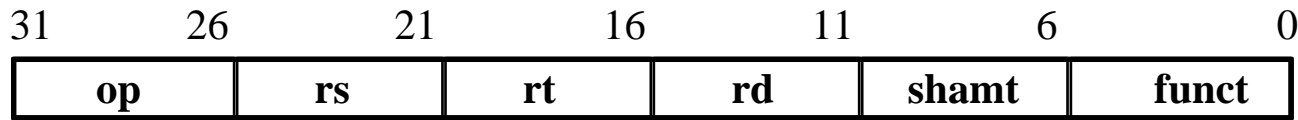
Calculate the next instruction's address

Fetch Unit at the Beginning of add

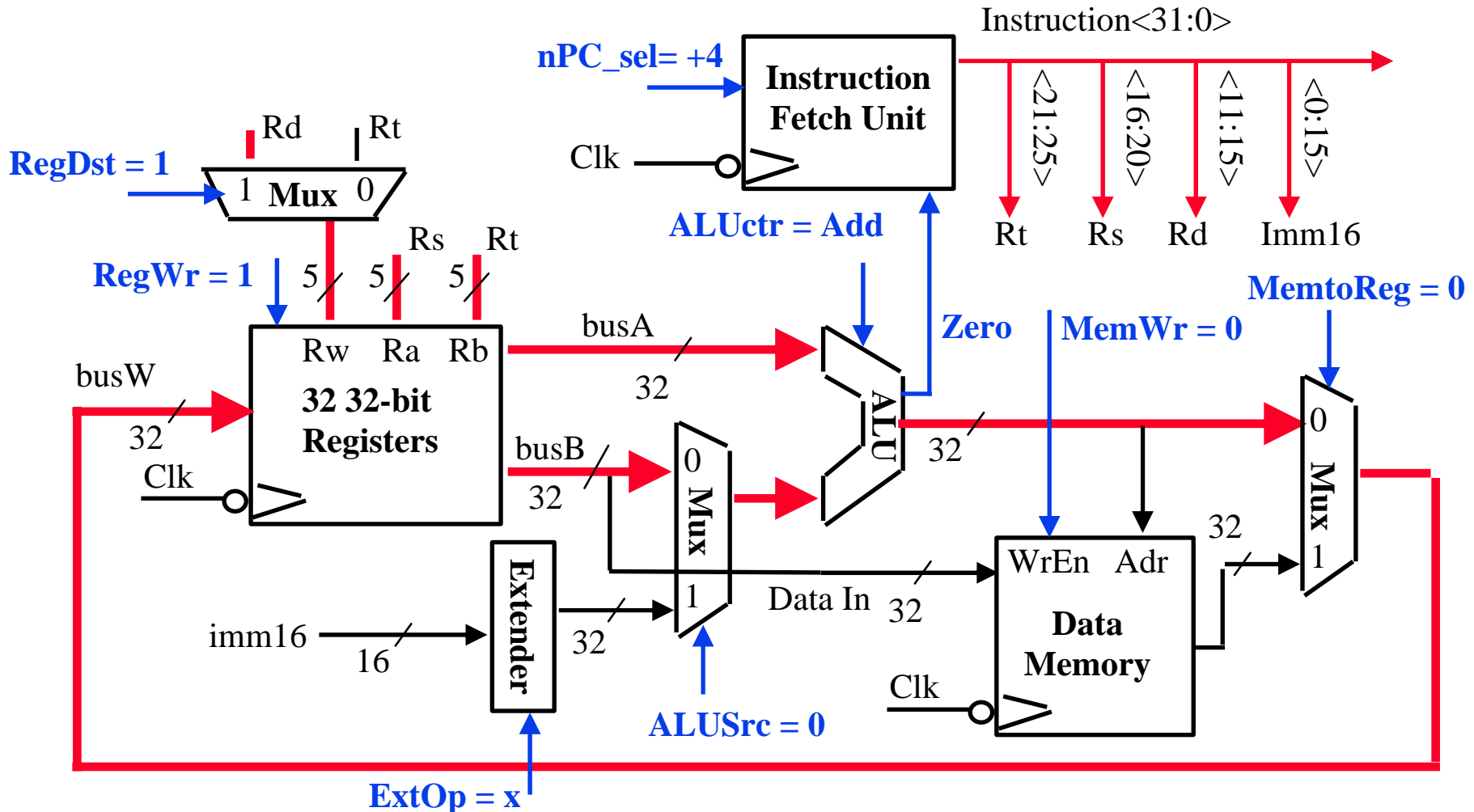
- Fetch the instruction from Instruction memory:
 $\text{Instruction} \leftarrow \text{mem}[\text{PC}]$
(This is the same for all instructions)



The Single Cycle Datapath during add

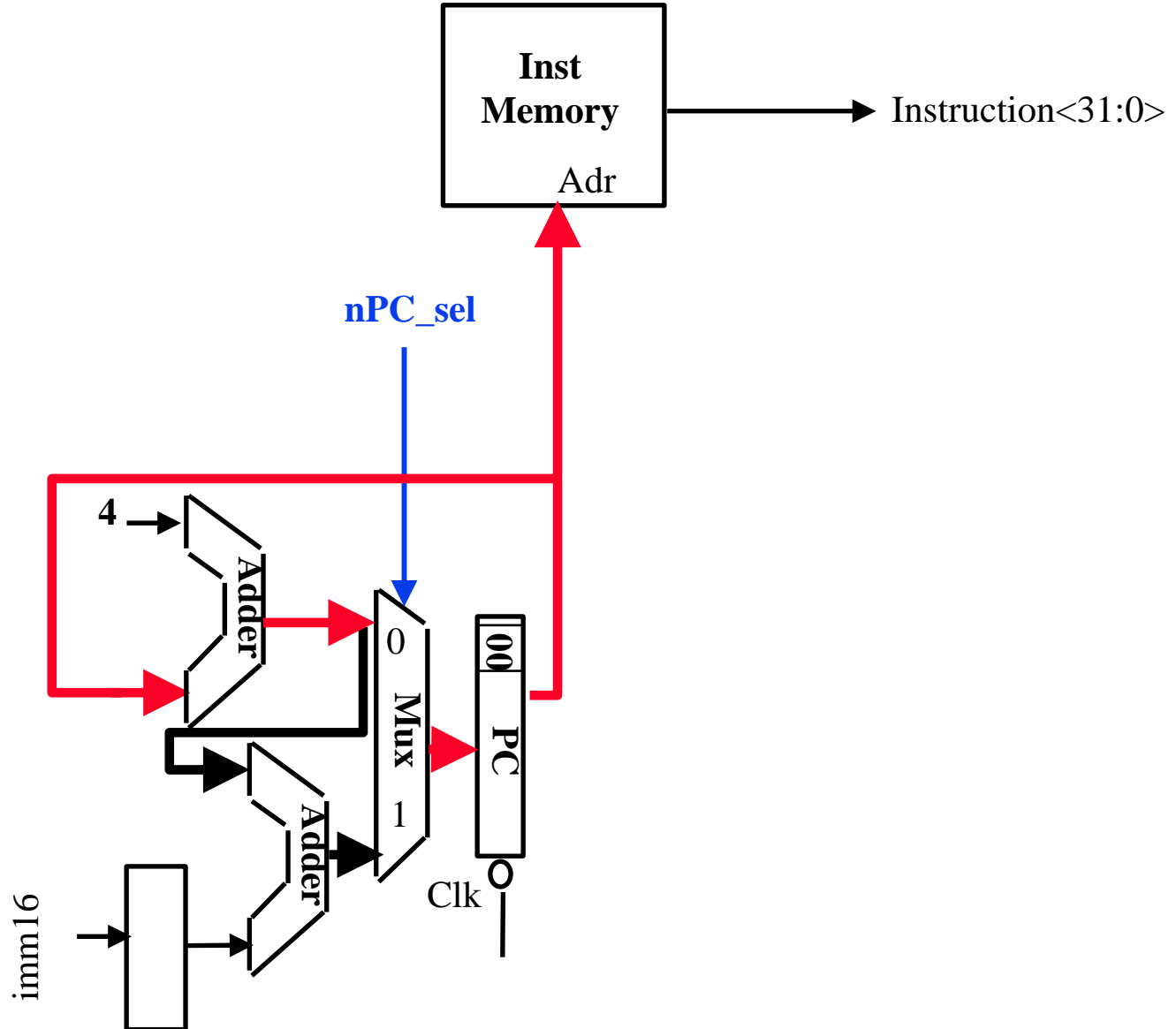


• $R[rd] \leftarrow R[rs] + R[rt]$

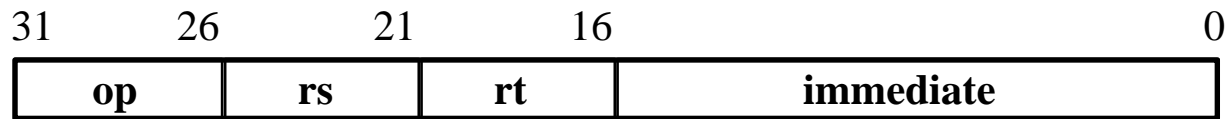


Instruction Fetch Unit at the End of add

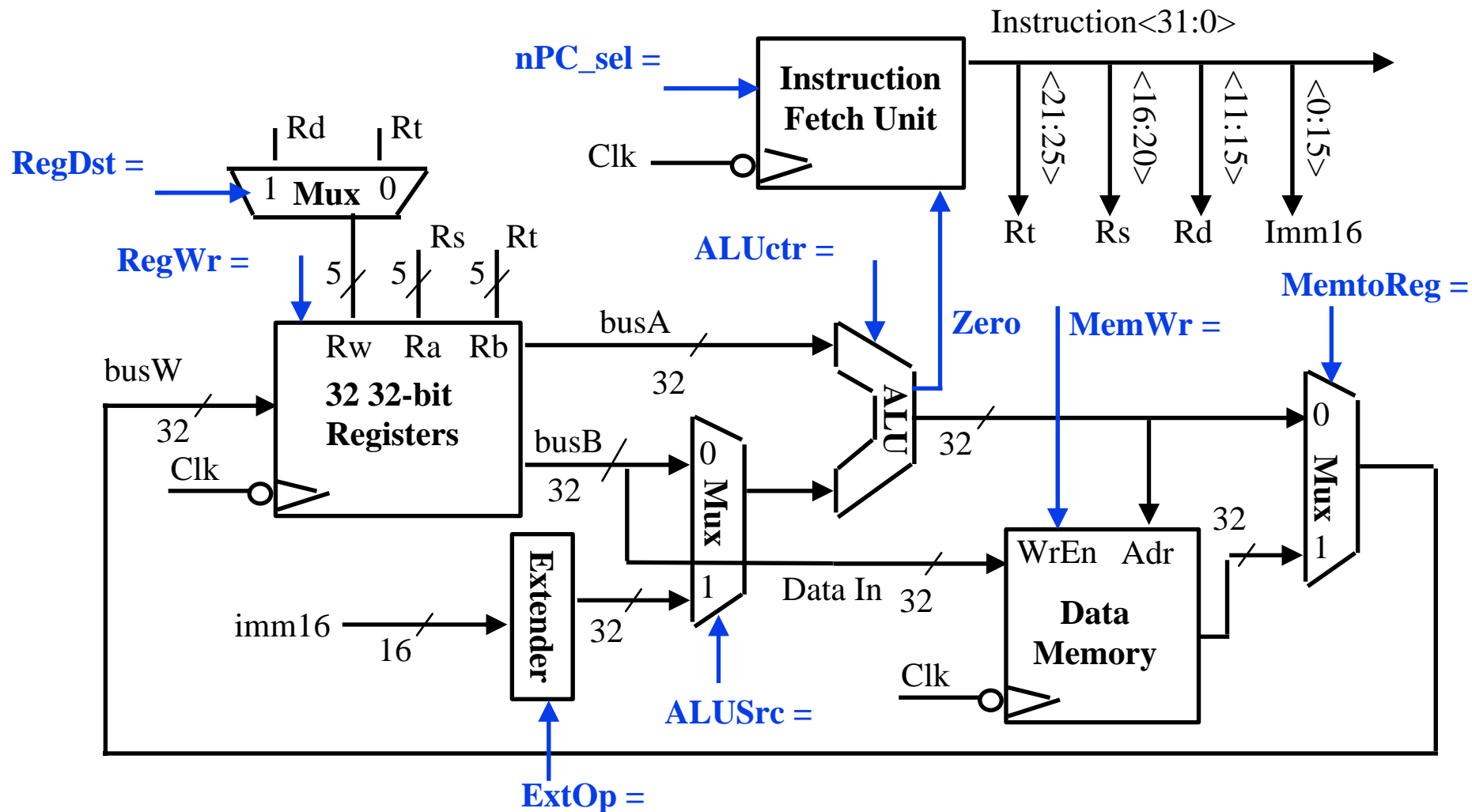
- $PC \leftarrow PC + 4$
 - This is the same for all instructions except: Branch and Jump



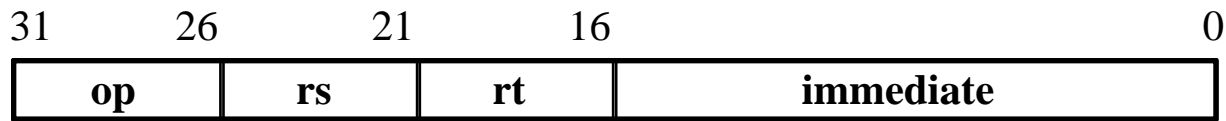
The Single Cycle Datapath during Or Immediate



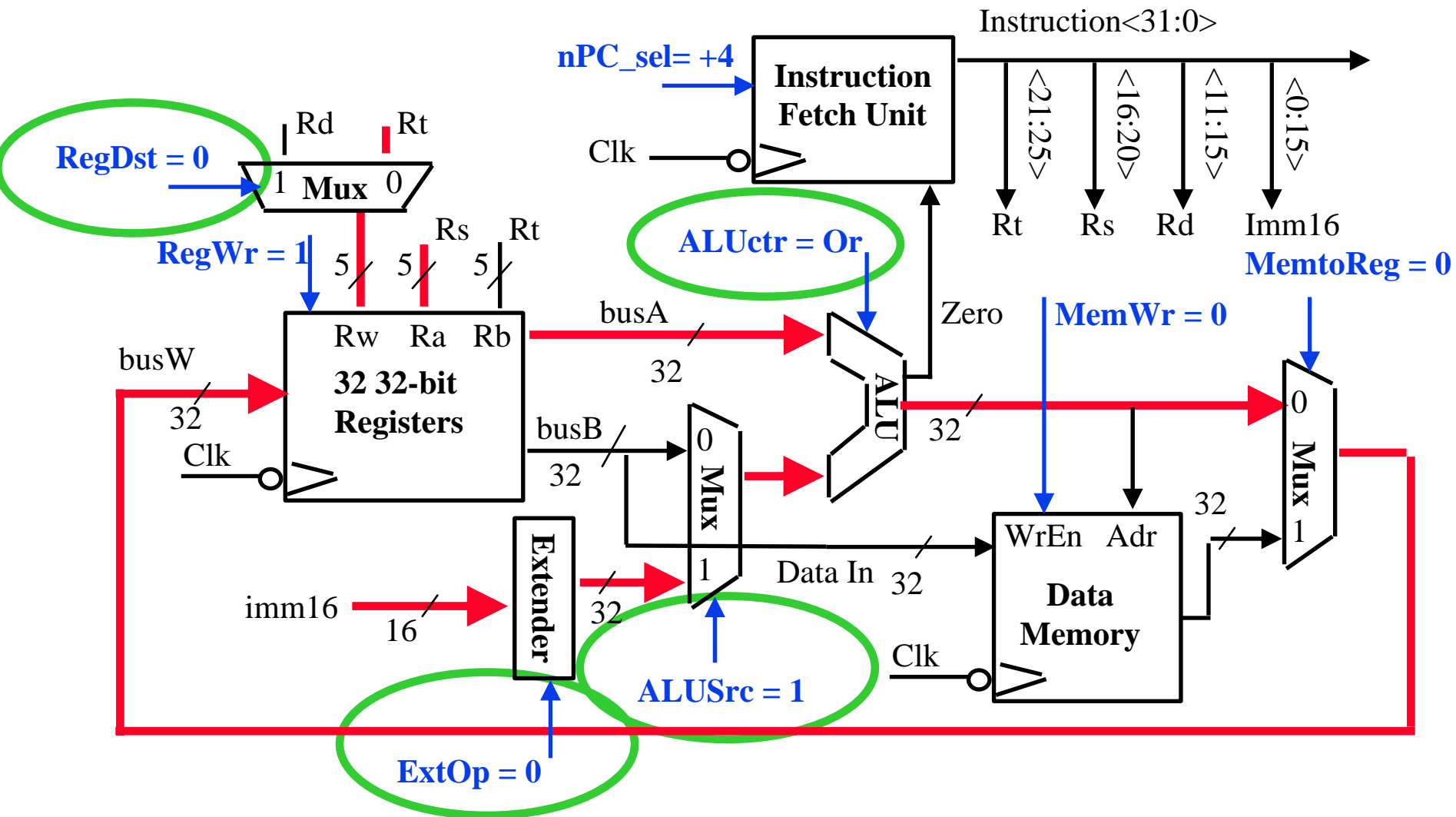
- $R[rt] \leftarrow R[rs] \text{ or } \text{ZeroExt}(\text{Imm16})$



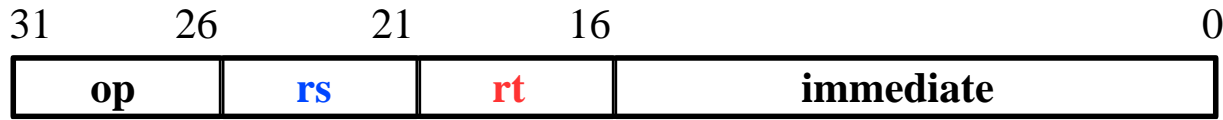
The Single Cycle Datapath during Or Immediate



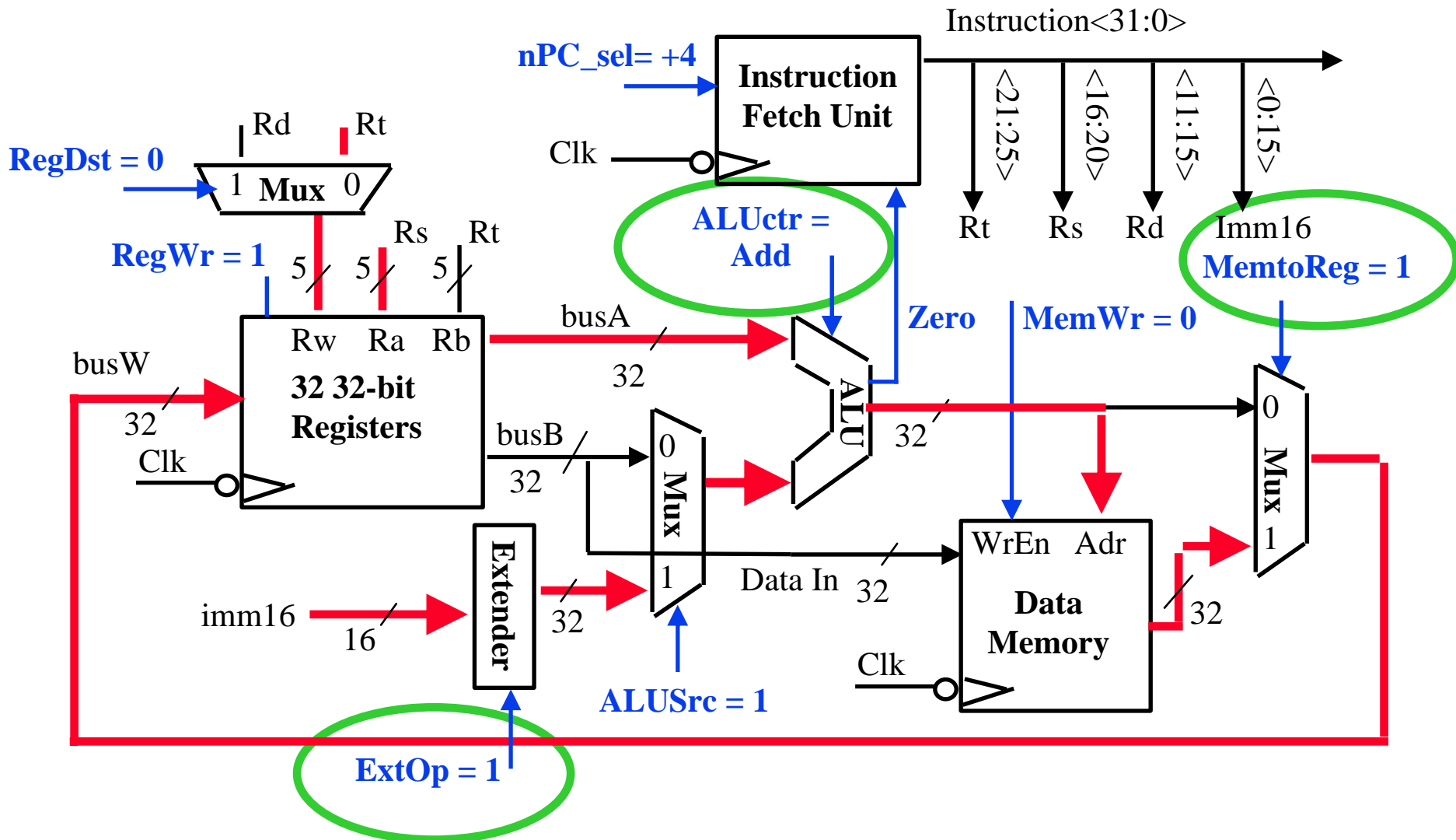
• $R[rt] \leftarrow R[rs] \text{ or } \text{ZeroExt}(\text{Imm16})$



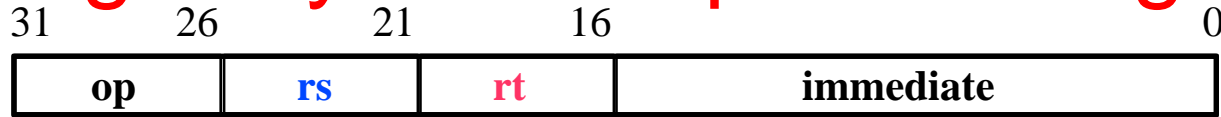
The Single Cycle Datapath during Load



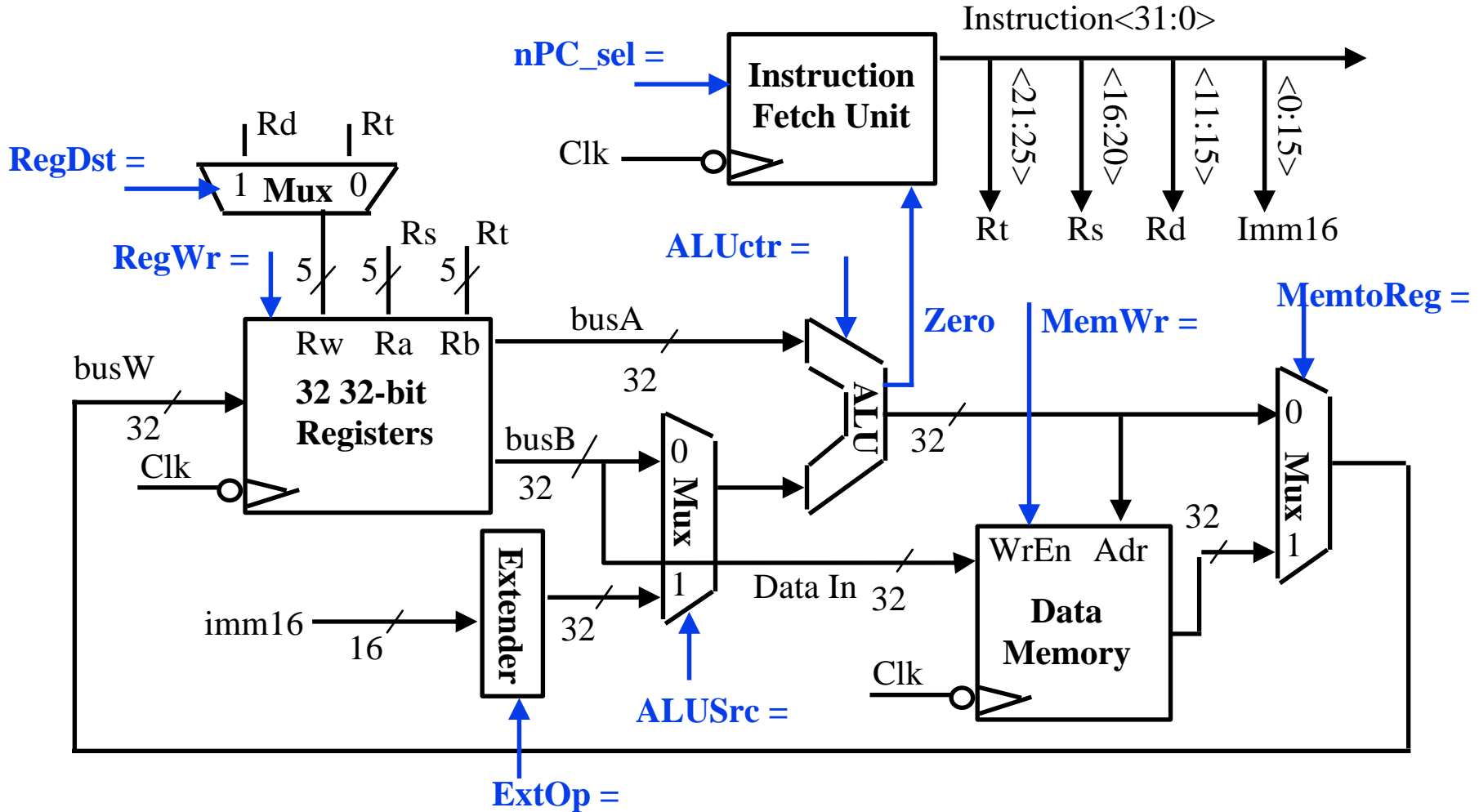
• $R[rt] \leftarrow \text{Data Memory} \{R[rs] + \text{SignExt}[imm16]\}$



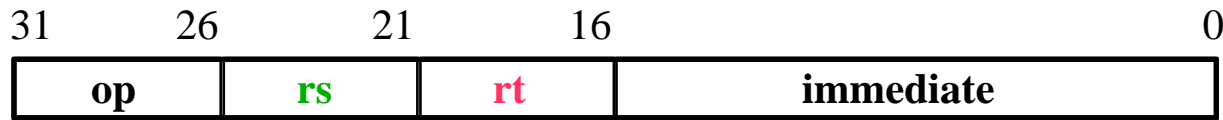
The Single Cycle Datapath during Store



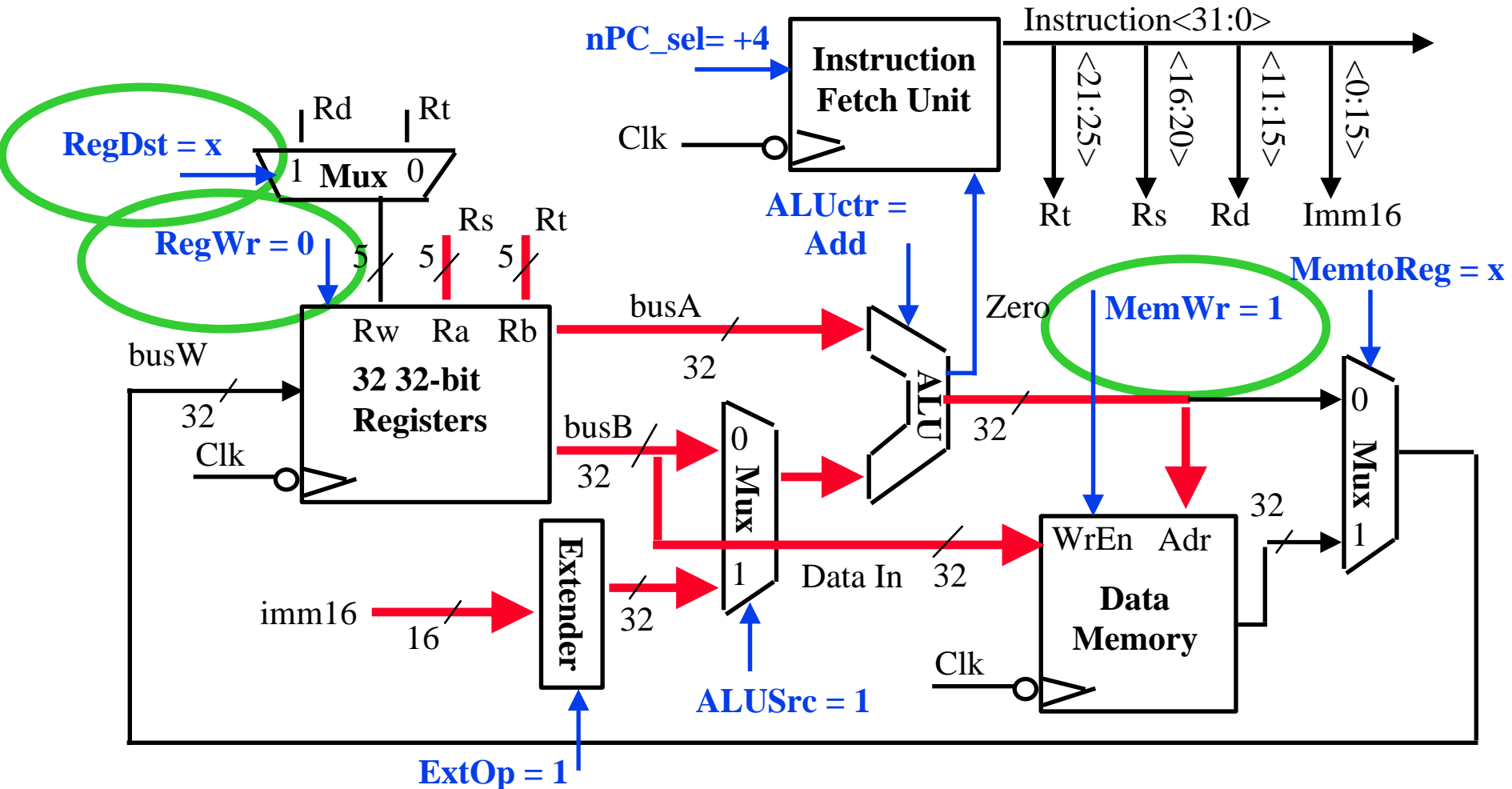
- Data Memory $\{R[rs] + \text{SignExt}[imm16]\} \leftarrow R[rt]$



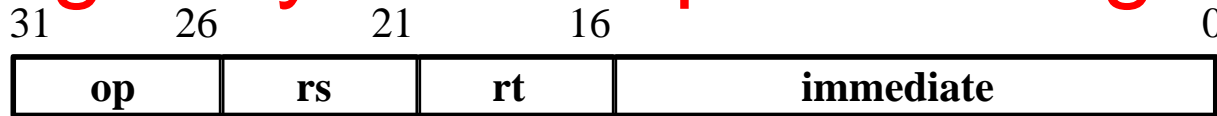
The Single Cycle Datapath during Store



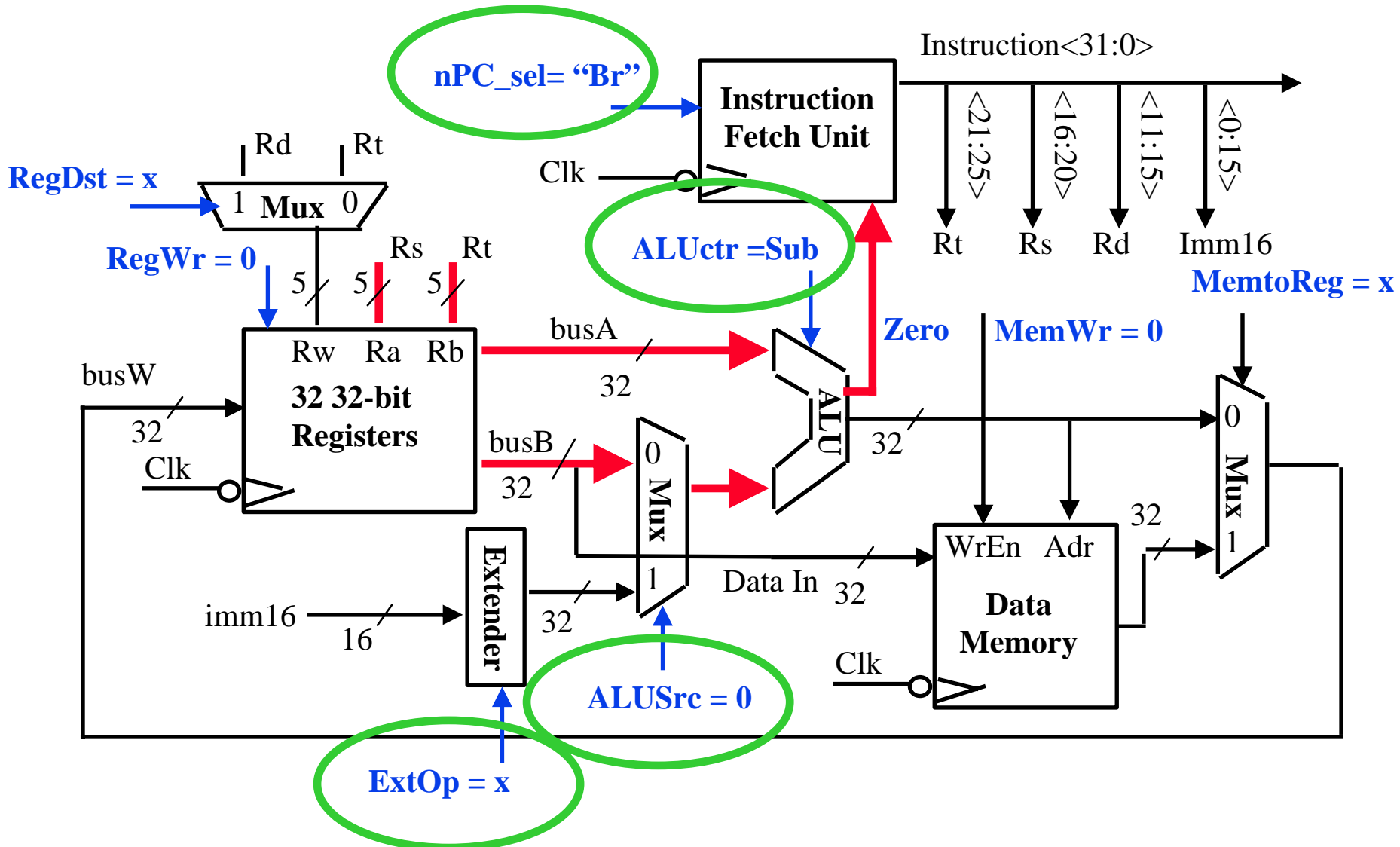
• Data Memory $\{R[rs] + \text{SignExt}[imm16]\} \leftarrow R[rt]$



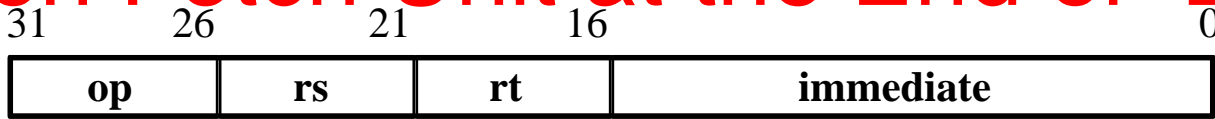
The Single Cycle Datapath during Branch



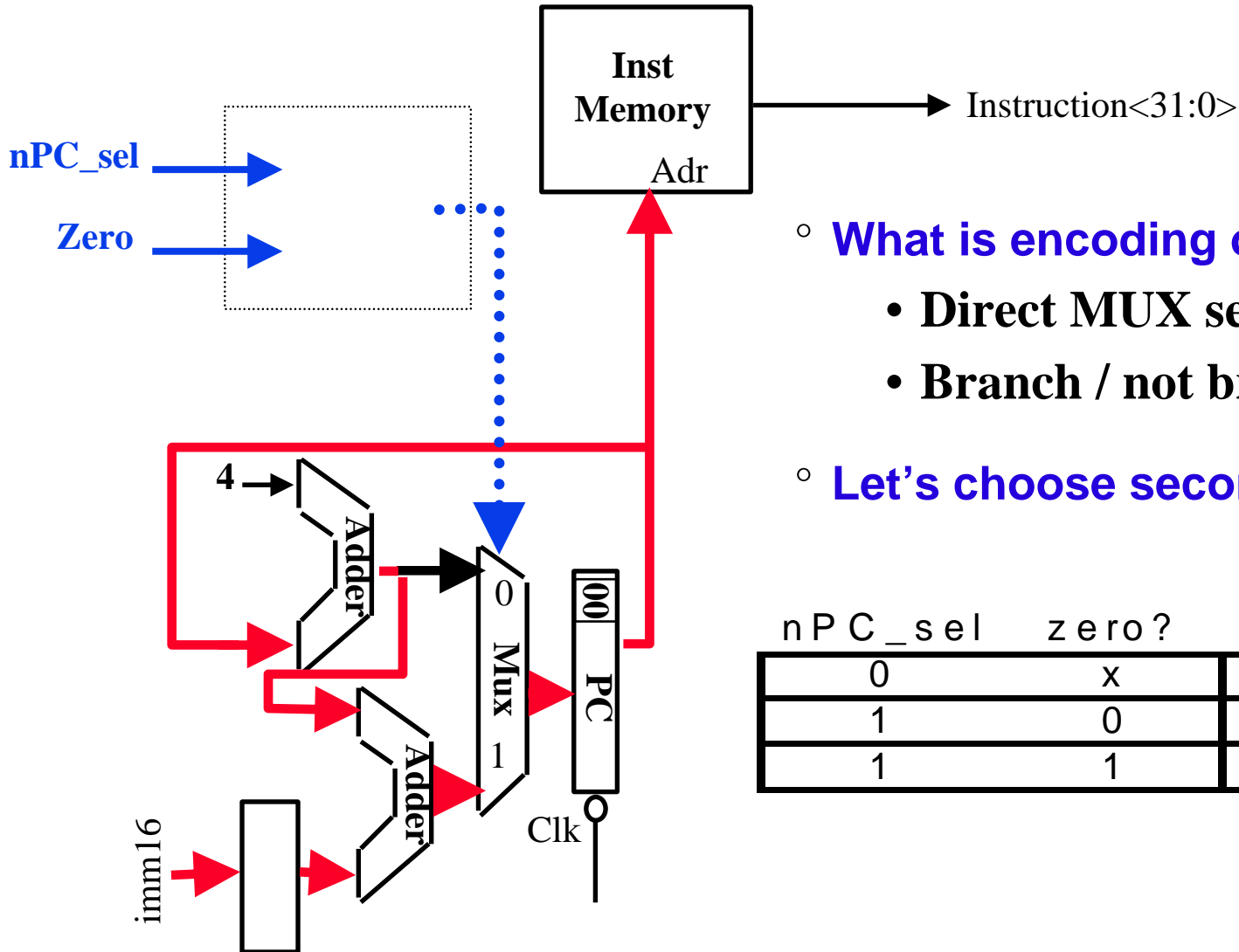
- if (R[rs] - R[rt] == 0) then Zero ← 1; else Zero ← 0



Instruction Fetch Unit at the End of Branch



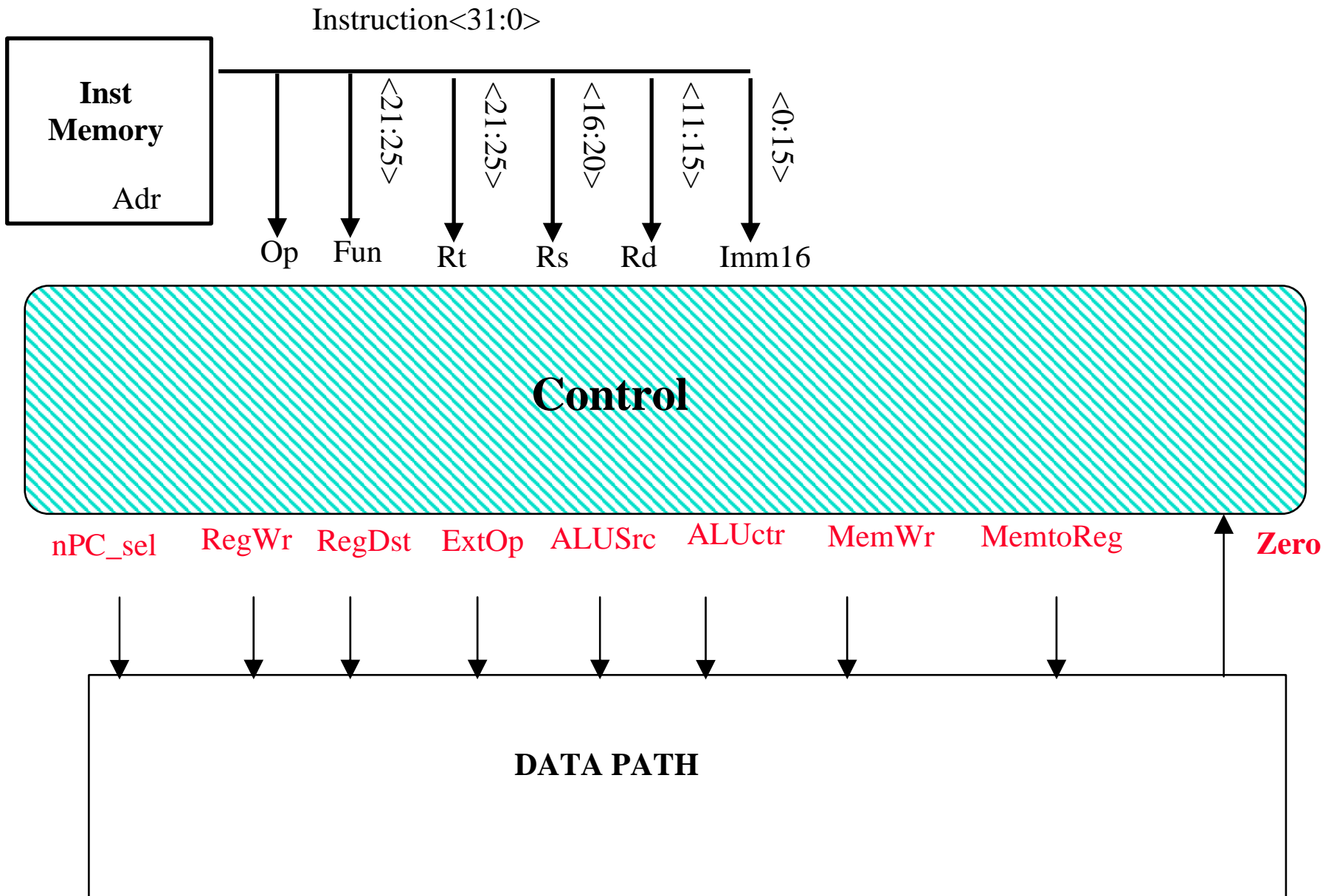
- if (Zero == 1) then $PC = PC + 4 + \text{SignExt}(\text{imm16}) \times 4$; else $PC = PC + 4$



- What is encoding of nPC_sel?
 - Direct MUX select?
 - Branch / not branch
- Let's choose second option

nPC_sel	zero?	MUX
0	x	0
1	0	0
1	1	1

Step 4: Given Datapath: RTL \rightarrow Control



A Summary of Control Signals

inst Register Transfer

ADD $R[rd] \leftarrow R[rs] + R[rt];$ $PC \leftarrow PC + 4$

$ALUsrc = \text{RegB}, ALUctr = \text{“add”}, \text{RegDst} = rd, \text{RegWr}, nPC_sel = \text{“+4”}$

SUB $R[rd] \leftarrow R[rs] - R[rt];$ $PC \leftarrow PC + 4$

$ALUsrc = \text{RegB}, ALUctr = \text{“sub”}, \text{RegDst} = rd, \text{RegWr}, nPC_sel = \text{“+4”}$

ORi $R[rt] \leftarrow R[rs] + \text{zero_ext}(\text{Imm16});$ $PC \leftarrow PC + 4$

$ALUsrc = \text{Im}, \text{Extop} = \text{“Z”}, ALUctr = \text{“or”}, \text{RegDst} = rt, \text{RegWr}, nPC_sel = \text{“+4”}$

LOAD $R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})];$ $PC \leftarrow PC + 4$

$ALUsrc = \text{Im}, \text{Extop} = \text{“Sn”}, ALUctr = \text{“add”},$

$\text{MemtoReg}, \text{RegDst} = rt, \text{RegWr},$ $nPC_sel = \text{“+4”}$

STORE $\text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})] \leftarrow R[rs];$ $PC \leftarrow PC + 4$

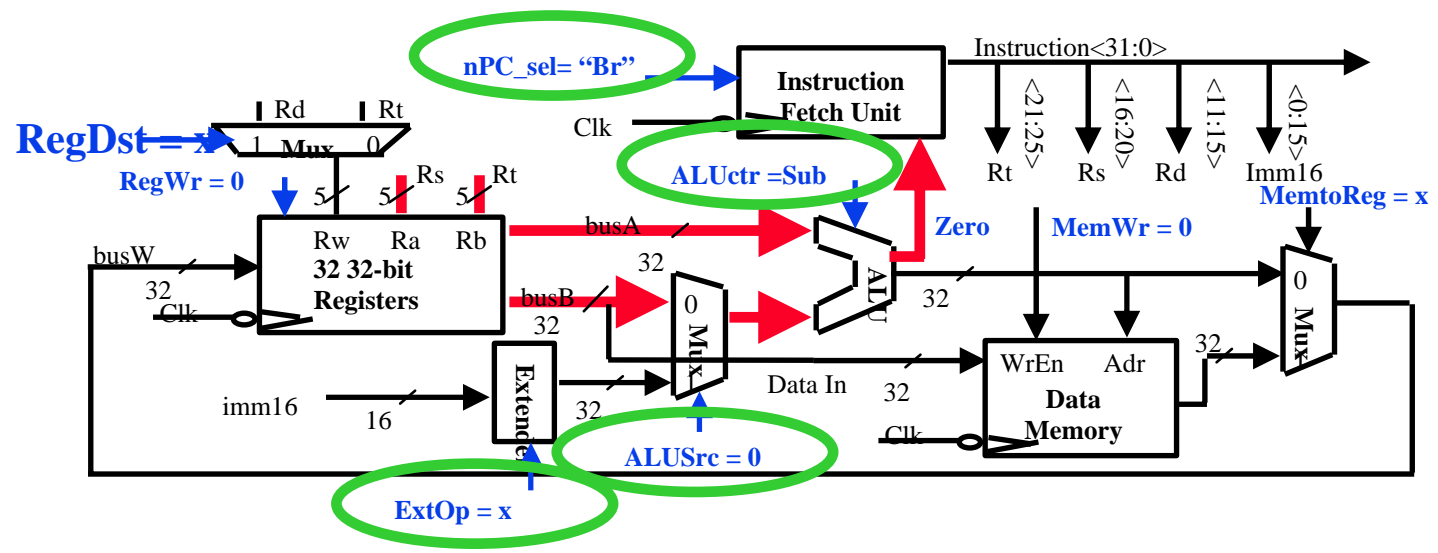
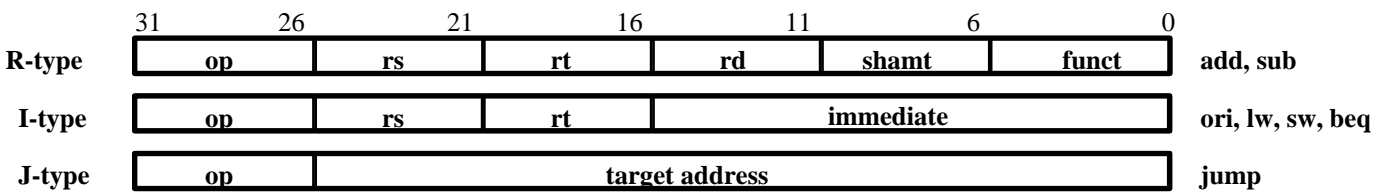
$ALUsrc = \text{Im}, \text{Extop} = \text{“Sn”}, ALUctr = \text{“add”}, \text{MemWr}, nPC_sel = \text{“+4”}$

BEQ $\text{if} (R[rs] == R[rt]) \text{ then } PC \leftarrow PC + \text{sign_ext}(\text{Imm16}) \parallel 00 \text{ else } PC \leftarrow PC + 4$

$nPC_sel = \text{“Br”}, ALUctr = \text{“sub”}$

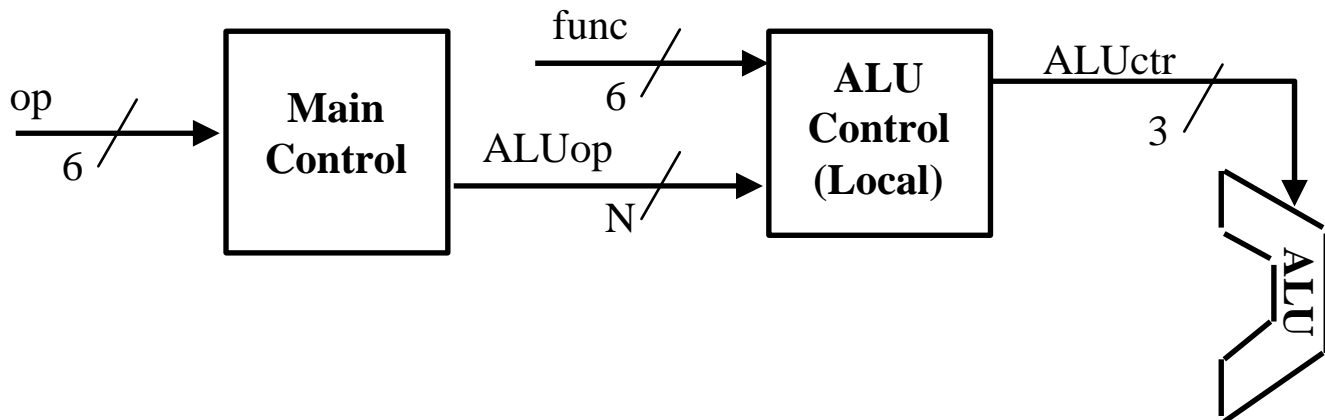
See Appendix A

	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	1	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
nPCsel		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

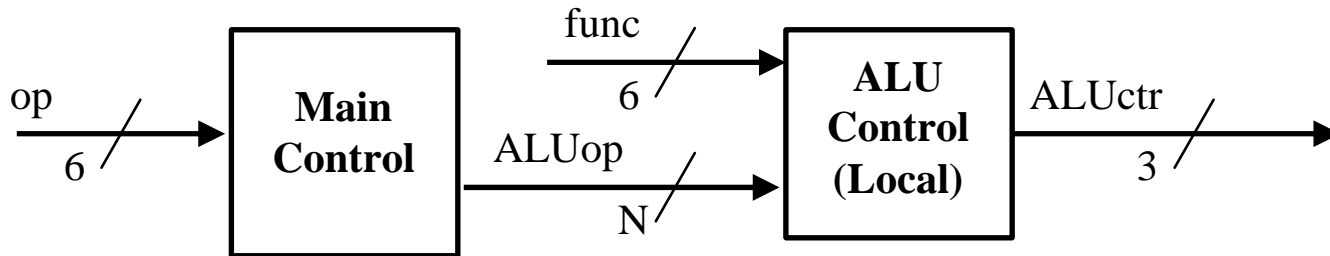


The Concept of Local Decoding

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



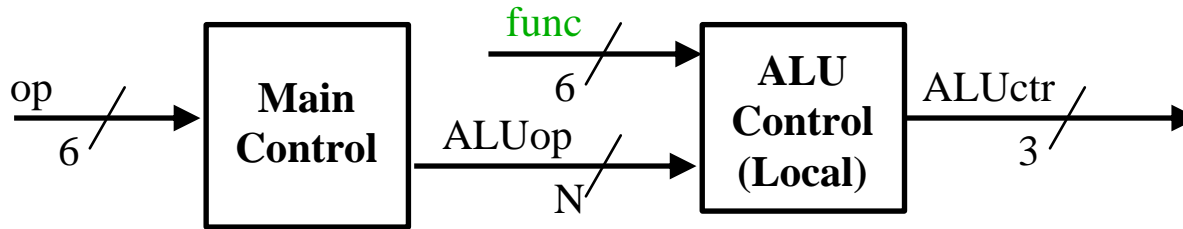
The Encoding of ALUop



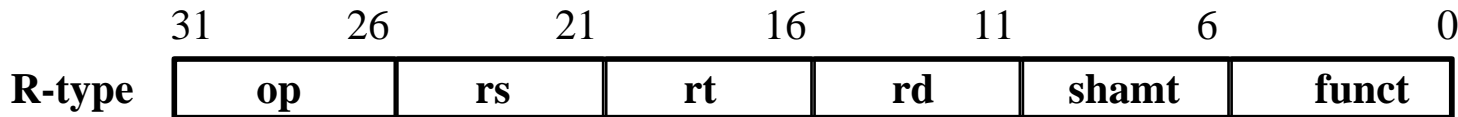
- **In this exercise, ALUop has to be 2 bits wide to represent:**
 - “R-type” instructions (1)
 - “I-type” instructions that require the ALU to perform:
 - (2) Or, (3) Add, and (4) Subtract
- **To implement the full MIPS ISA, ALUop has to be 3 bits to represent:**
 - “R-type” instructions (1)
 - “I-type” instructions that require the ALU to perform:
 - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

The Decoding of the “func” Field

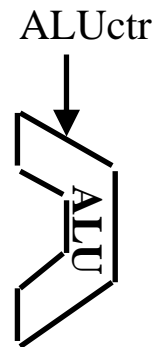


	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx



P. 286 text:

funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than



ALUctr<2:0>	ALU Operation
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

The Truth Table for ALUctr

ALUop (Symbolic)	R-type	ori	lw	sw	beq
	"R-type"	Or	Add	Add	Subtract
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01

func<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

The Logic Equation for ALUctr<2>

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

This makes func<3> a don't care

- $ALUctr<2> = !ALUop<2> \& ALUop<0> + ALUop<2> \& !func<2> \& func<1> \& !func<0>$

The Logic Equation for ALUctr<1>

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

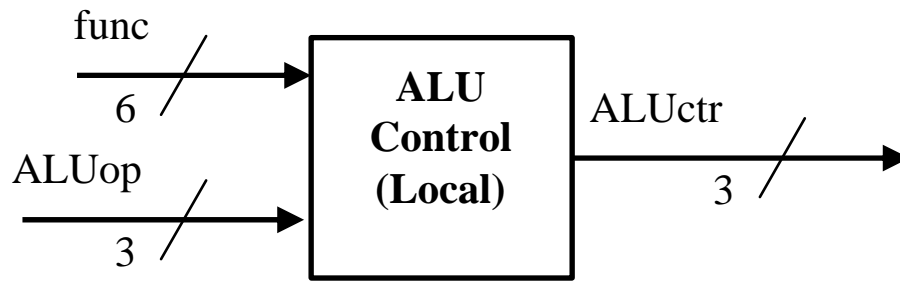
- $ALUctr<1> = !ALUop<2> \& !ALUop<0> + ALUop<2> \& !func<2> \& !func<0>$

The Logic Equation for ALUctr<0>

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

- $ALUctr<0> = !ALUop<2> \& ALUop<0>$
+ $ALUop<2> \& !func<3> \& func<2> \& !func<1> \& func<0>$
+ $ALUop<2> \& func<3> \& !func<2> \& func<1> \& !func<0>$

The ALU Control Block



- $ALUctr<2> = !ALUop<2> \ \& \ ALUop<0> \ +$
 $ALUop<2> \ \& \ !func<2> \ \& \ func<1> \ \& \ !func<0>$
- $ALUctr<1> = !ALUop<2> \ \& \ !ALUop<0> \ +$
 $ALUop<2> \ \& \ !func<2> \ \& \ !func<0>$
- $ALUctr<0> = !ALUop<2> \ \& \ ALUop<0>$
 $+ \ ALUop<2> \ \& \ !func<3> \ \& \ func<2>$
 $\ \& \ !func<1> \ \& \ func<0>$
 $+ \ ALUop<2> \ \& \ func<3> \ \& \ !func<2>$
 $\ \& \ func<1> \ \& \ !func<0>$

Step 5: Logic for Each Control Signal

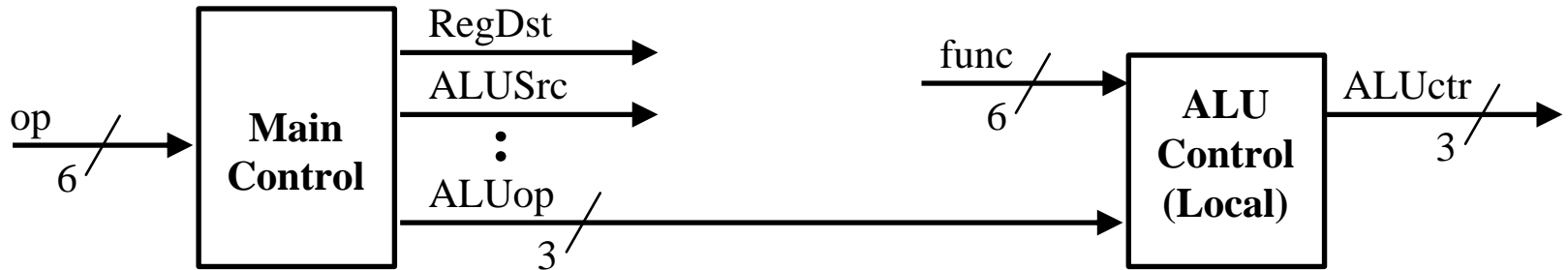
- **nPC_sel** <= if (OP == BEQ) then “Br” else “+4”
- **ALUsrc** <= if (OP == “Rtype”) then “regB” else “immed”
- **ALUctr** <= if (OP == “Rtype”) then funct
 elseif (OP == ORi) then “OR”
 elseif (OP == BEQ) then “sub”
 else “add”

- **ExtOp** <= _____
- **MemWr** <= _____
- **MemtoReg** <= _____
- **RegWr:** <= _____
- **RegDst:** <= _____

Step 5: Logic for each control signal

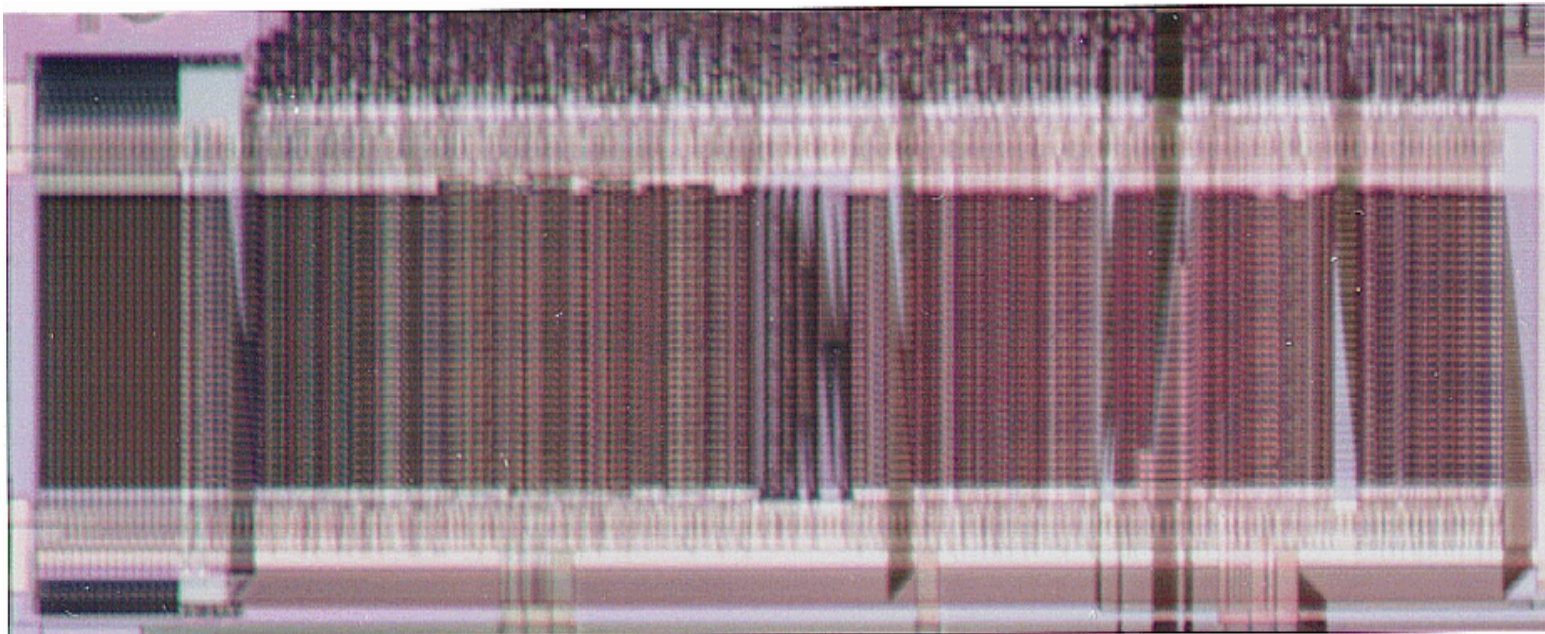
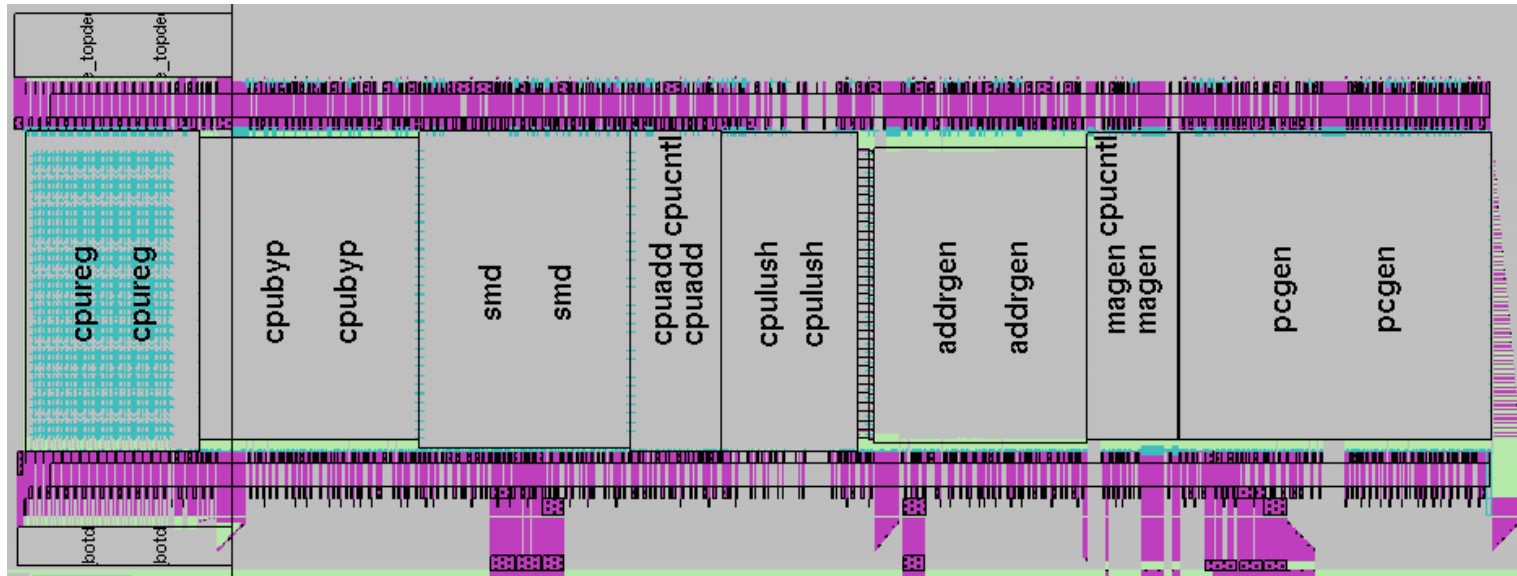
- **nPC_sel** \leq if (OP == BEQ) then “Br” else “+4”
- **ALUsrc** \leq if (OP == “Rtype”) then “regB” else “immed”
- **ALUctr** \leq if (OP == “Rtype”) then funct
 elseif (OP == ORi) then “OR”
 elseif (OP == BEQ) then “sub”
 else “add”
- **ExtOp** \leq if (OP == ORi) then “zero” else “sign”
- **MemWr** \leq (OP == Store)
- **MemtoReg** \leq (OP == Load)
- **RegWr:** \leq if ((OP == Store) || (OP == BEQ)) then 0 else 1
- **RegDst:** \leq if ((OP == Load) || (OP == ORi)) then 0 else 1

The “Truth Table” for the Main Control

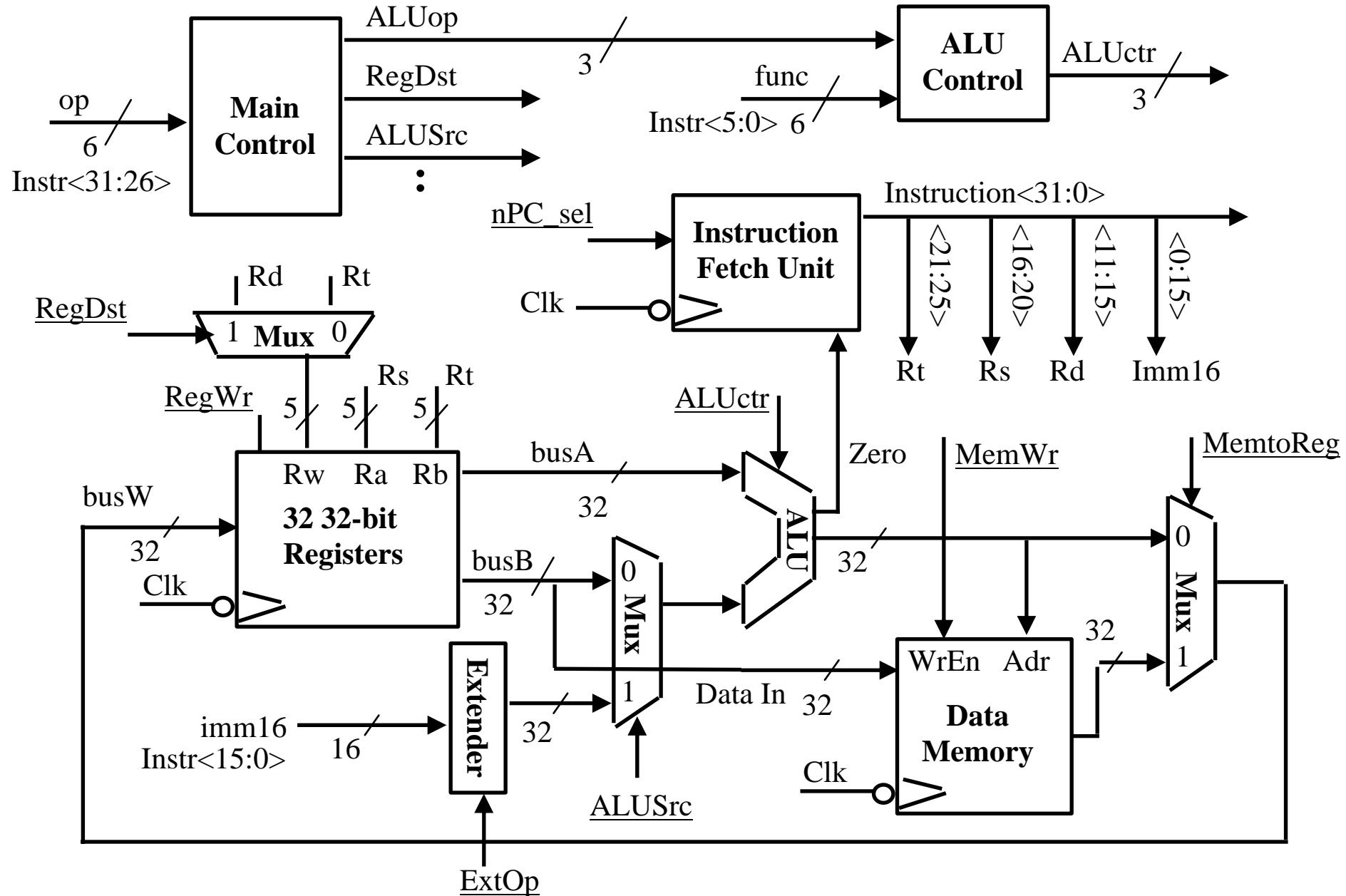


op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
<u>RegWrite</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>
MemWrite	0	0	0	1	0	0
nPC_sel	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUOp (Symbolic)	“R-type”	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

A Real MIPS Datapath (CNS T0)



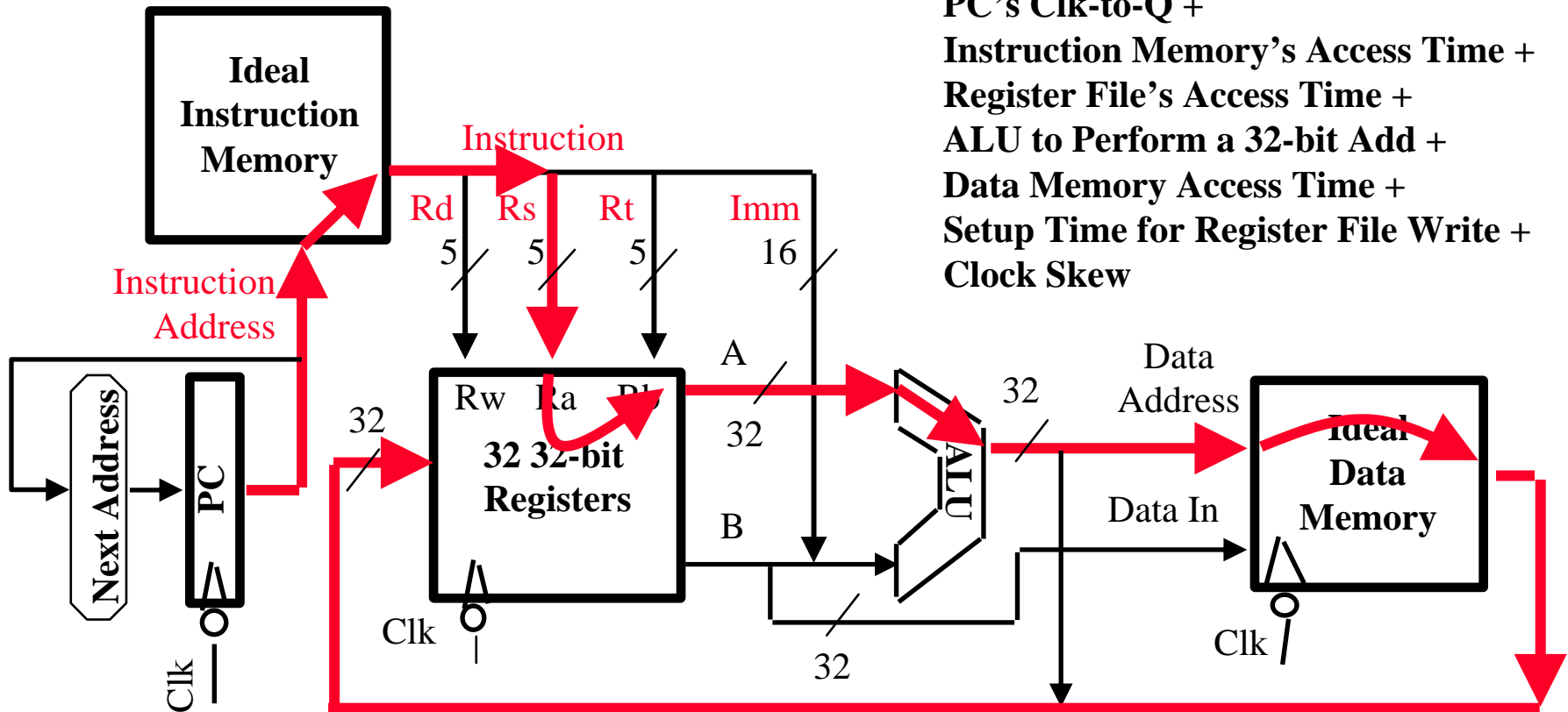
Summary: A Single Cycle Processor



Recap: An Abstract View of the Critical Path (Load)

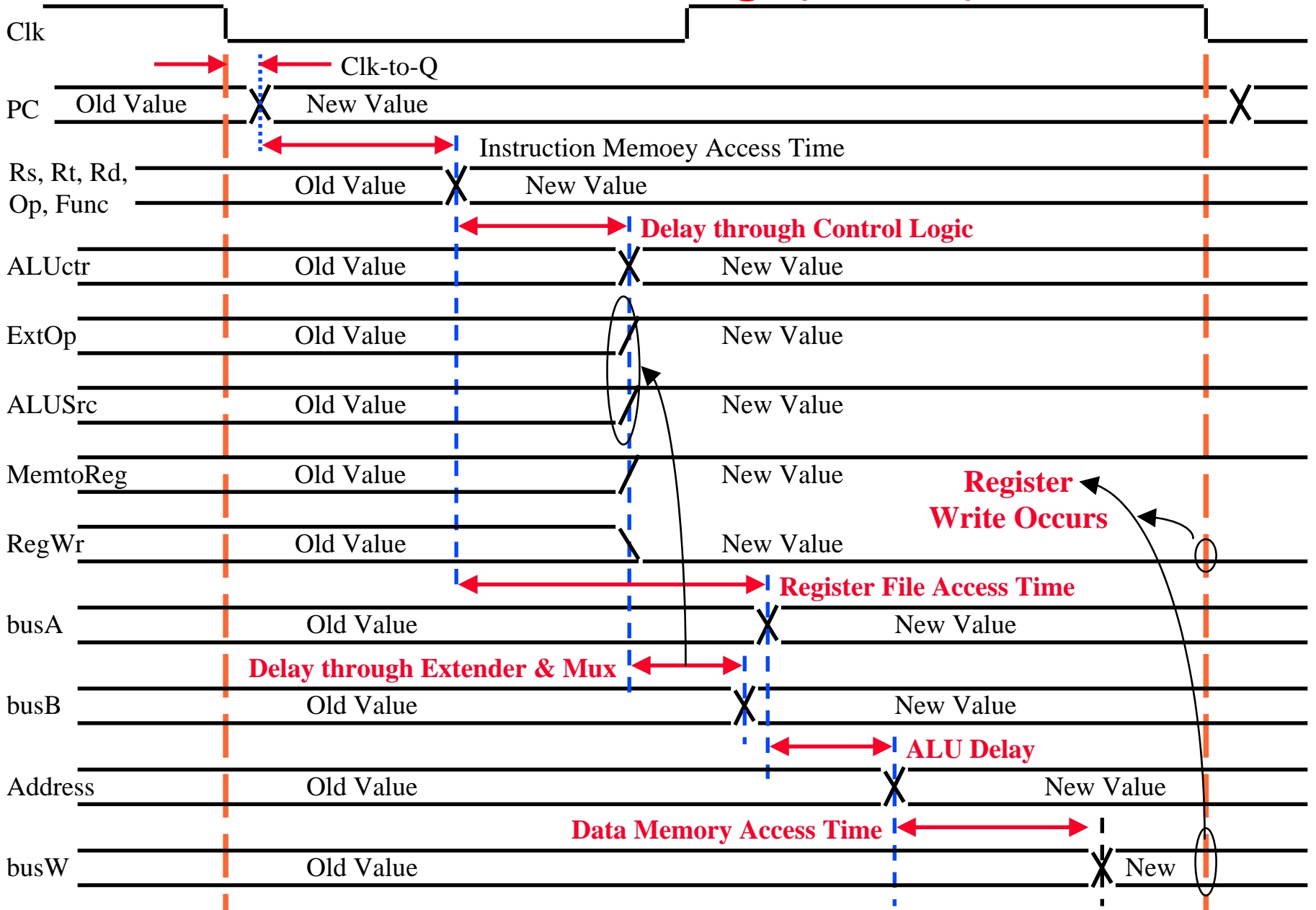
- **Register file and ideal memory:**

- 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 a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

Worst Case Timing (Load)



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 +

Clock Skew

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

Summary

- **Single cycle datapath: CPI = 1, CCT → long**
- **5 steps to design a processor**
 - 1. Analyze instruction set => datapath requirements
 - 2. Select set of datapath components & establish clock methodology
 - 3. Assemble datapath meeting the requirements
 - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
 - 5. Assemble the control logic

◦ **Control is the hard part**

◦ **MIPS makes control easier**

- Instructions same size
- Source registers always in same place
- Immediates same size, location
- Operations always on registers/immediates

