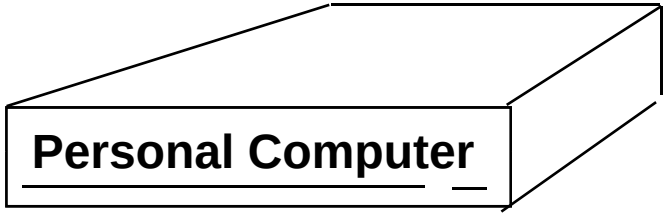
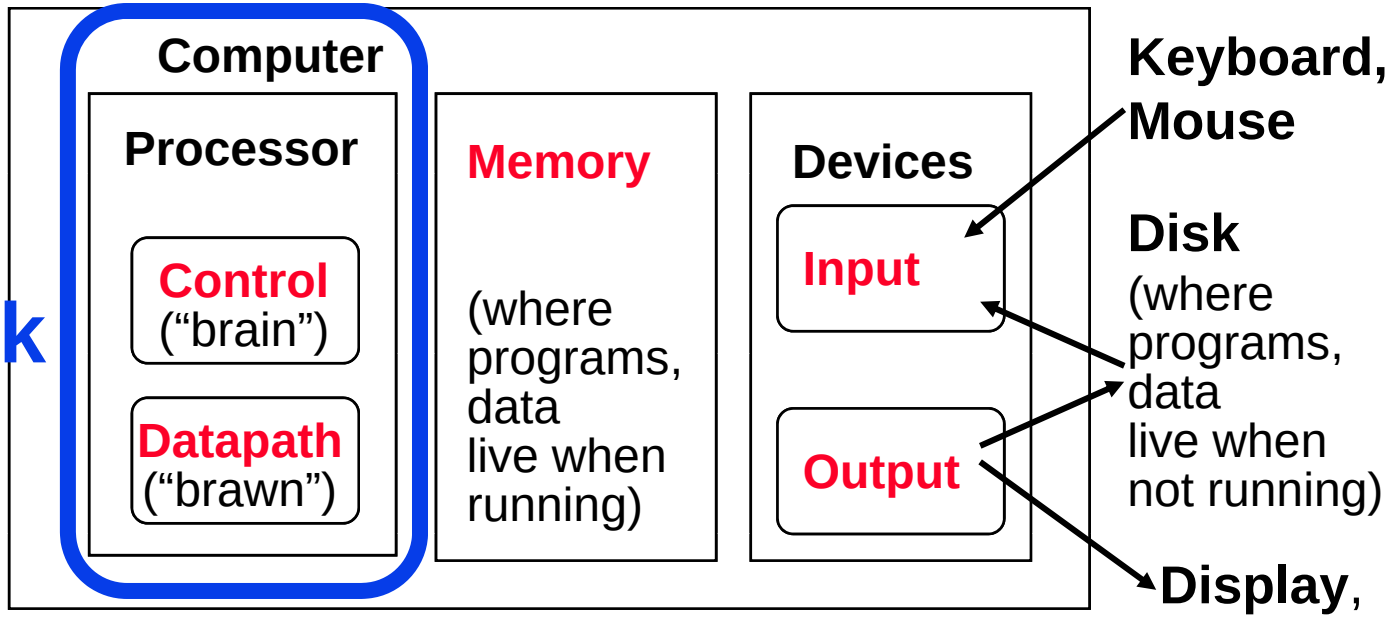
Lecture #17: CPU Design II – Control

2005-07-19

Anatomy: 5 components of any Computer



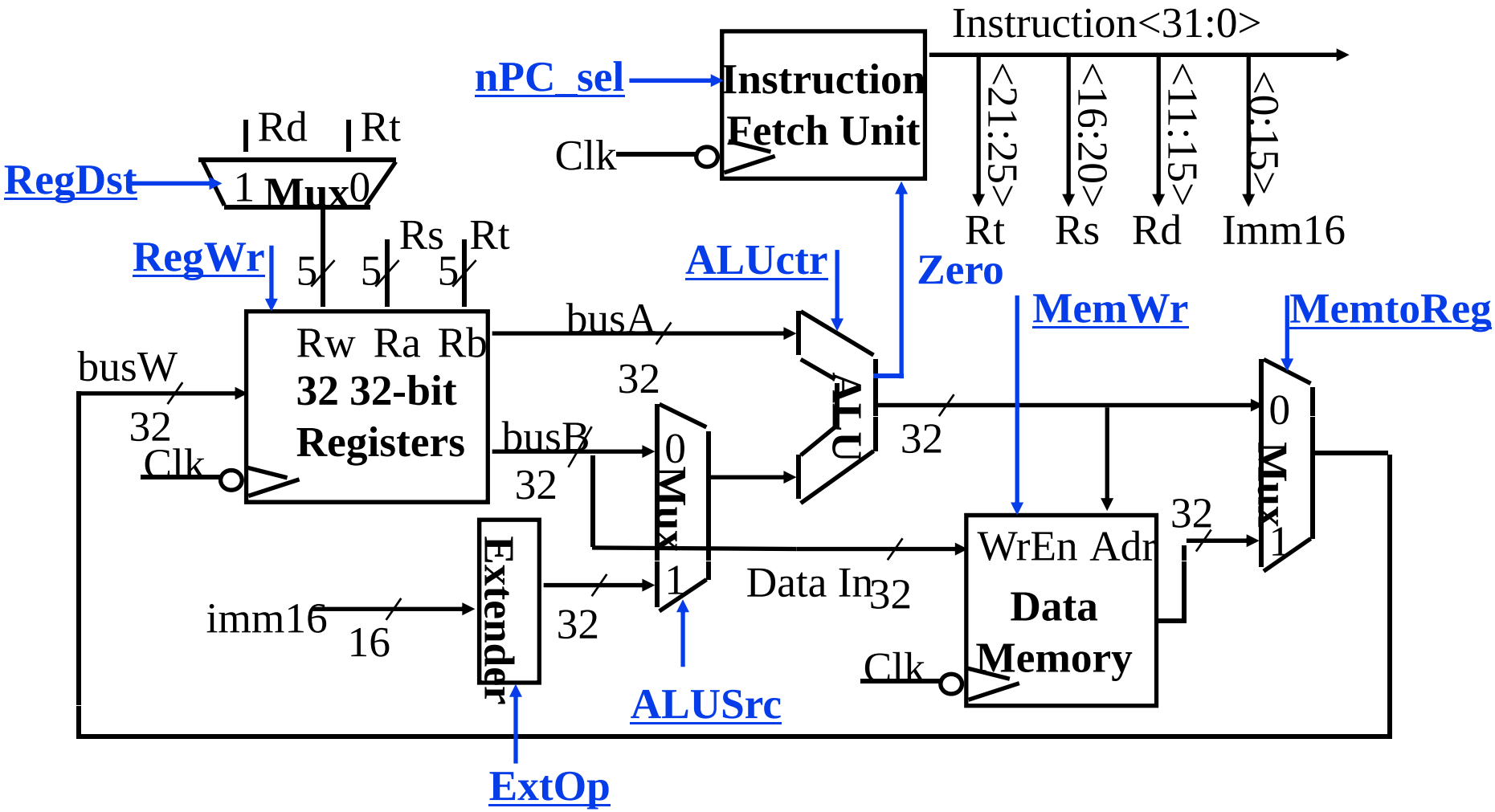
This week



☹brawn: muscle

Review: A Single Cycle Datapath

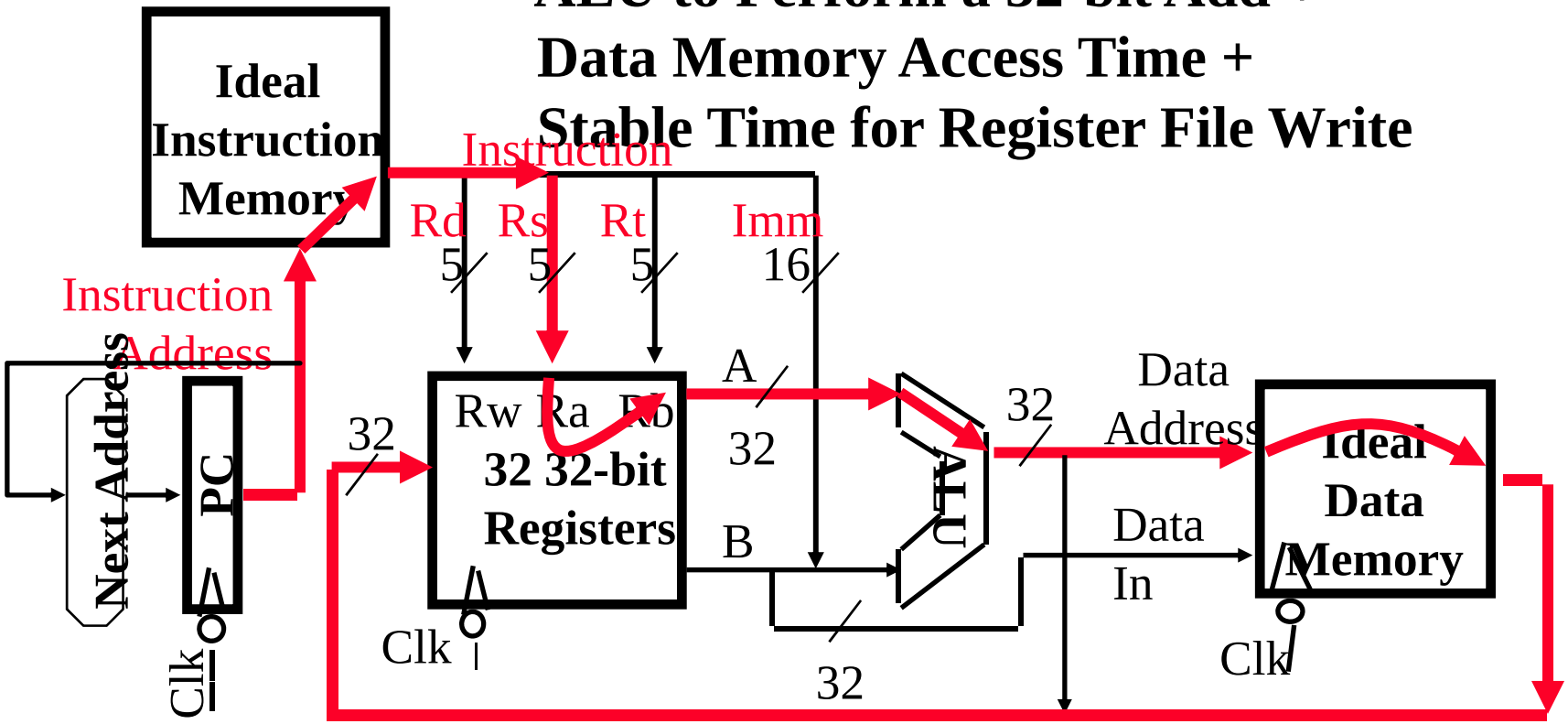
- Rs, Rt, Rd, Imed16 connected to datapath
- We have everything except control signals



An Abstract View of the Critical Path

- **This affects how much you can overclock your PC!**

Critical Path (Load Operation) =
Delay clock through PC (FFs) +
Instruction Memory's Access Time +
Register File's Access Time, +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Stable Time for Register File Write

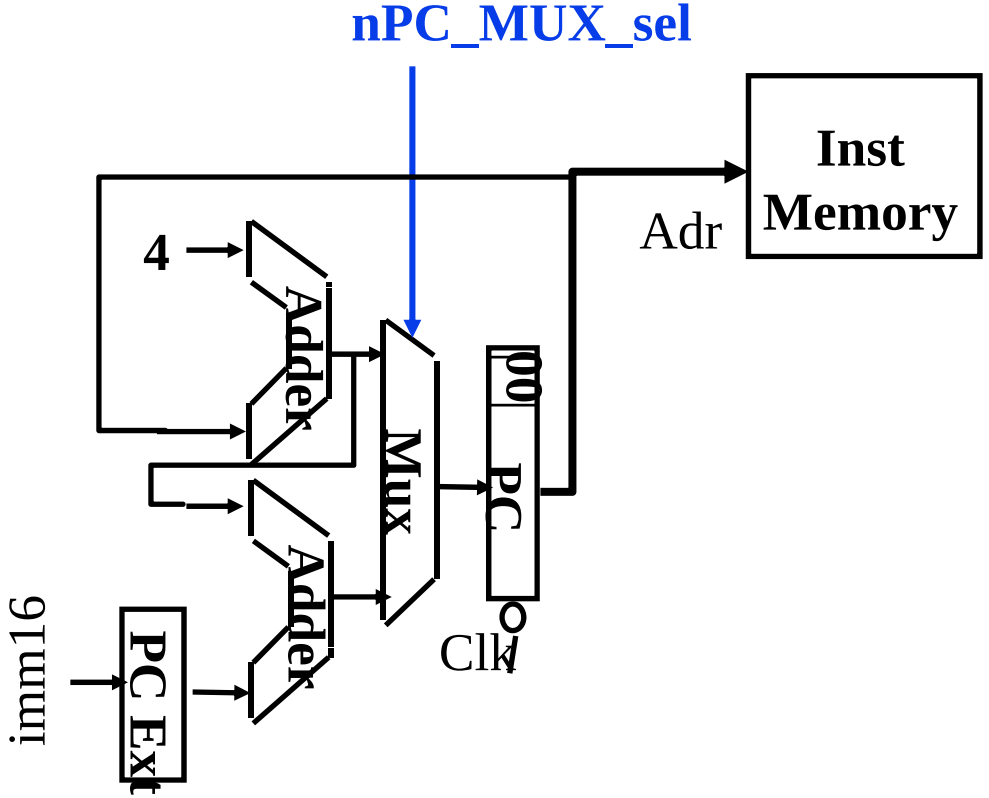


Recap: Meaning of the Control Signals

- **nPC_MUX_sel:** **0 ⇒ PC <– PC + 4**
  1 ⇒ PC <– PC + 4 + {SignExt(Imm16) , 00 }

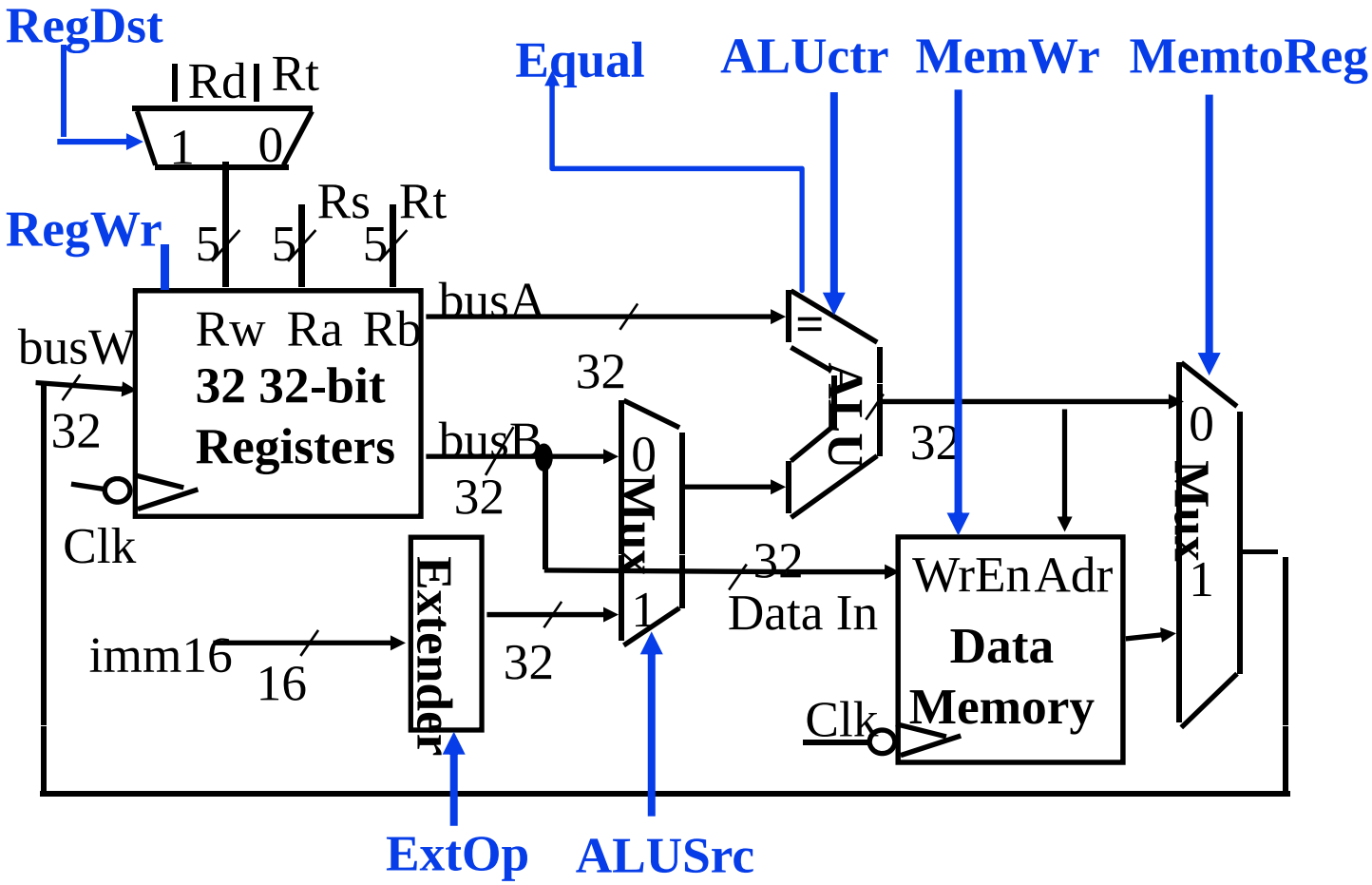
  "n"=next

- **Later in lecture: higher-level connection between mux and branch cond**

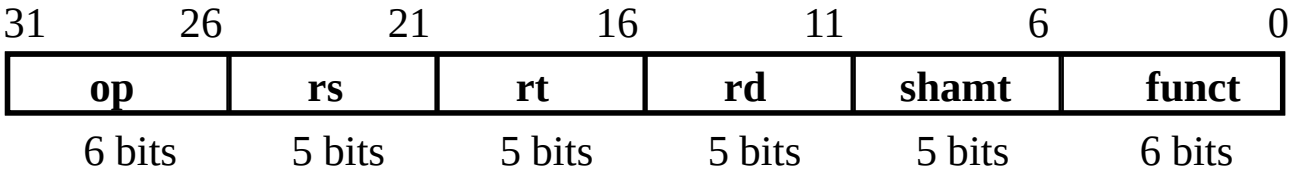


Recap: Meaning of the Control Signals

- **ExtOp:** "zero", "sign"
  Zero or sign extended
- **ALUsrc:** 0 ⇒ regB;
  1 ⇒ immed
- **ALUctr:** "add", "sub", "or"
- **MemWr:** 1 ⇒ write memory
- **MemtoReg:** 0 ⇒ ALU; 1 ⇒ Mem
- **RegDst:** 0 ⇒ "rt"; 1 ⇒ "rd"
- **RegWr:** 1 ⇒ write register



RTL: The Add Instruction

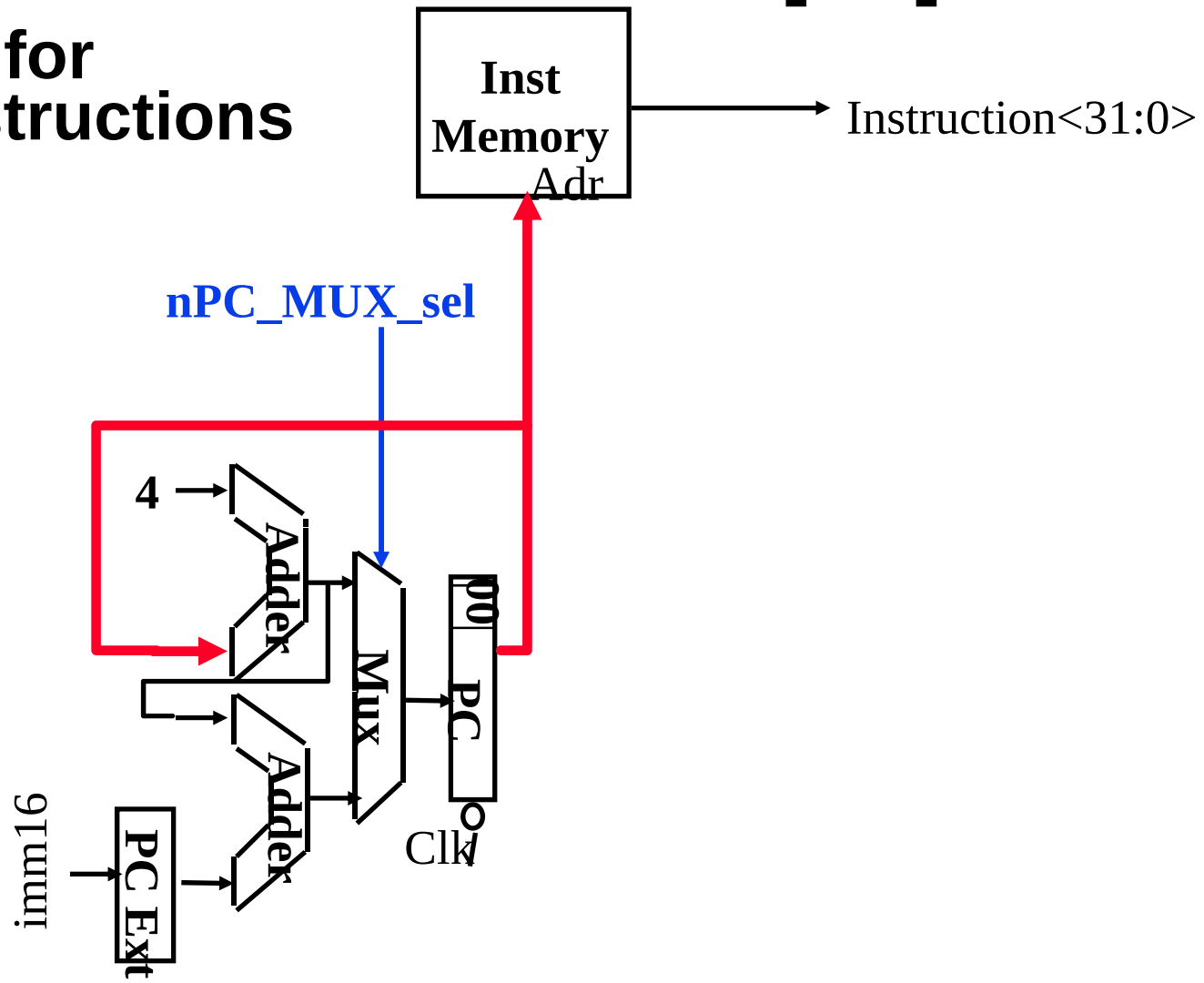



```
add rd, rs, rt
```

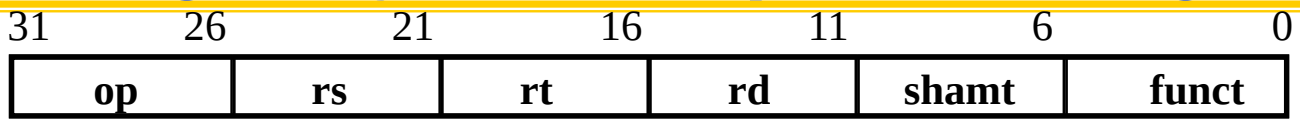
- **MEM[PC]** **Fetch the instruction from memory**
- **R[rd] = R[rs] + R[rt]** **The actual operation**
- **PC = PC + 4** **Calculate the next instruction's address**

Instruction Fetch Unit at the Beginning of Add

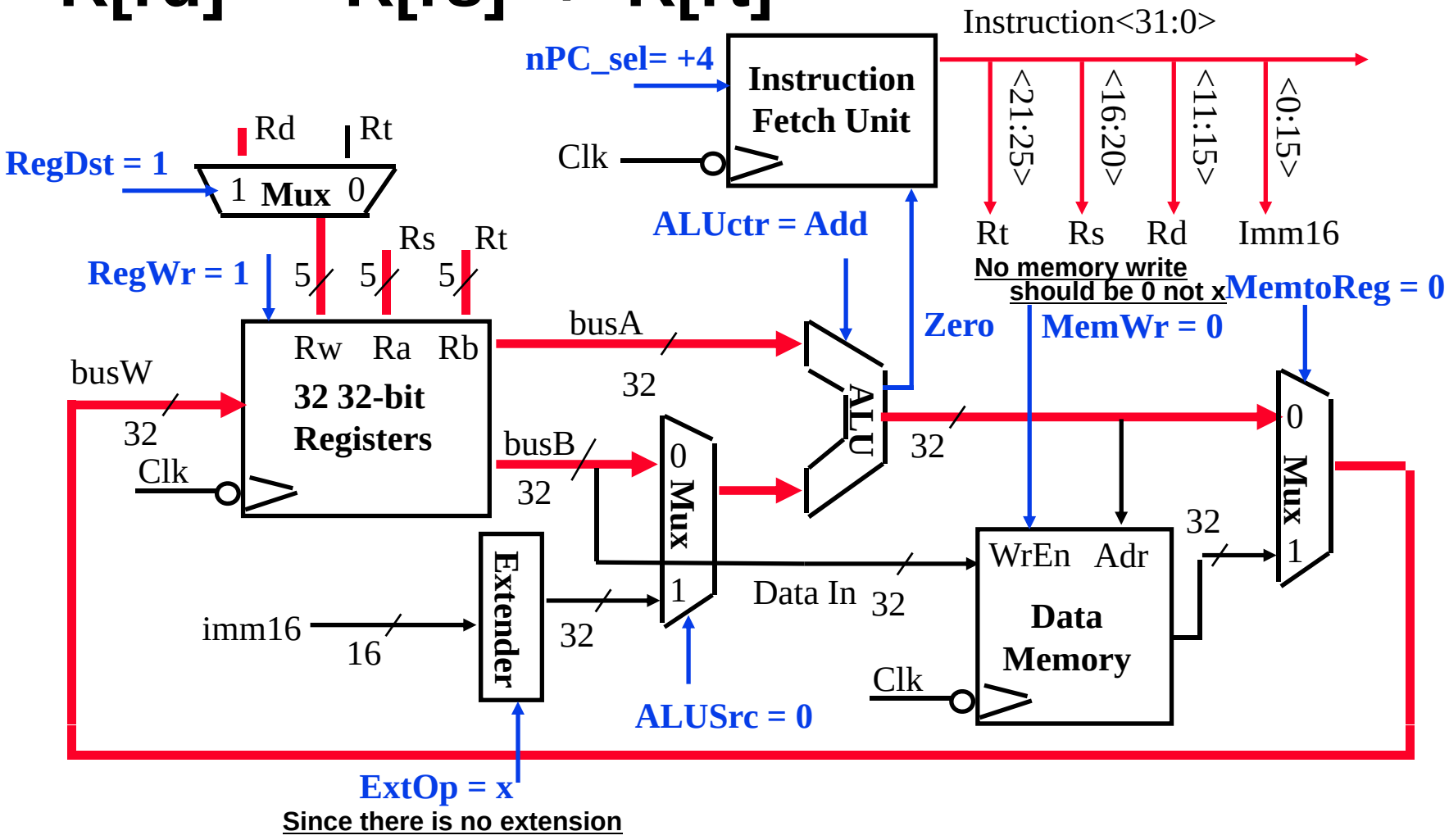
- **Fetch the instruction from Instruction memory: Instruction = MEM[PC]**
  - **same for all instructions**



The Single Cycle Datapath during Add

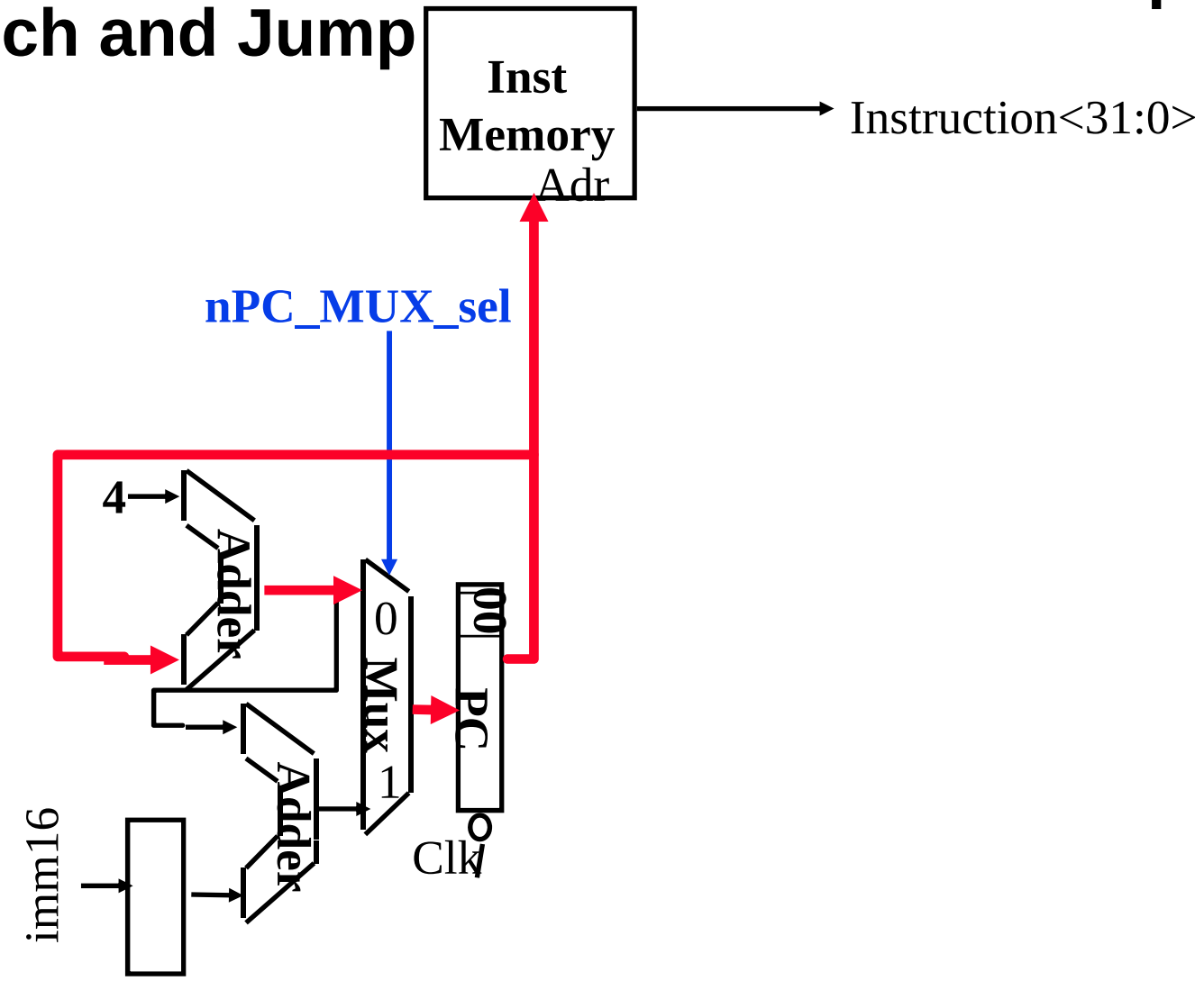


- R[rd] = R[rs] + R[rt]

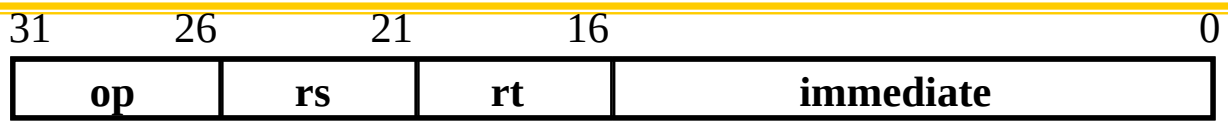


Instruction Fetch Unit at the End of Add

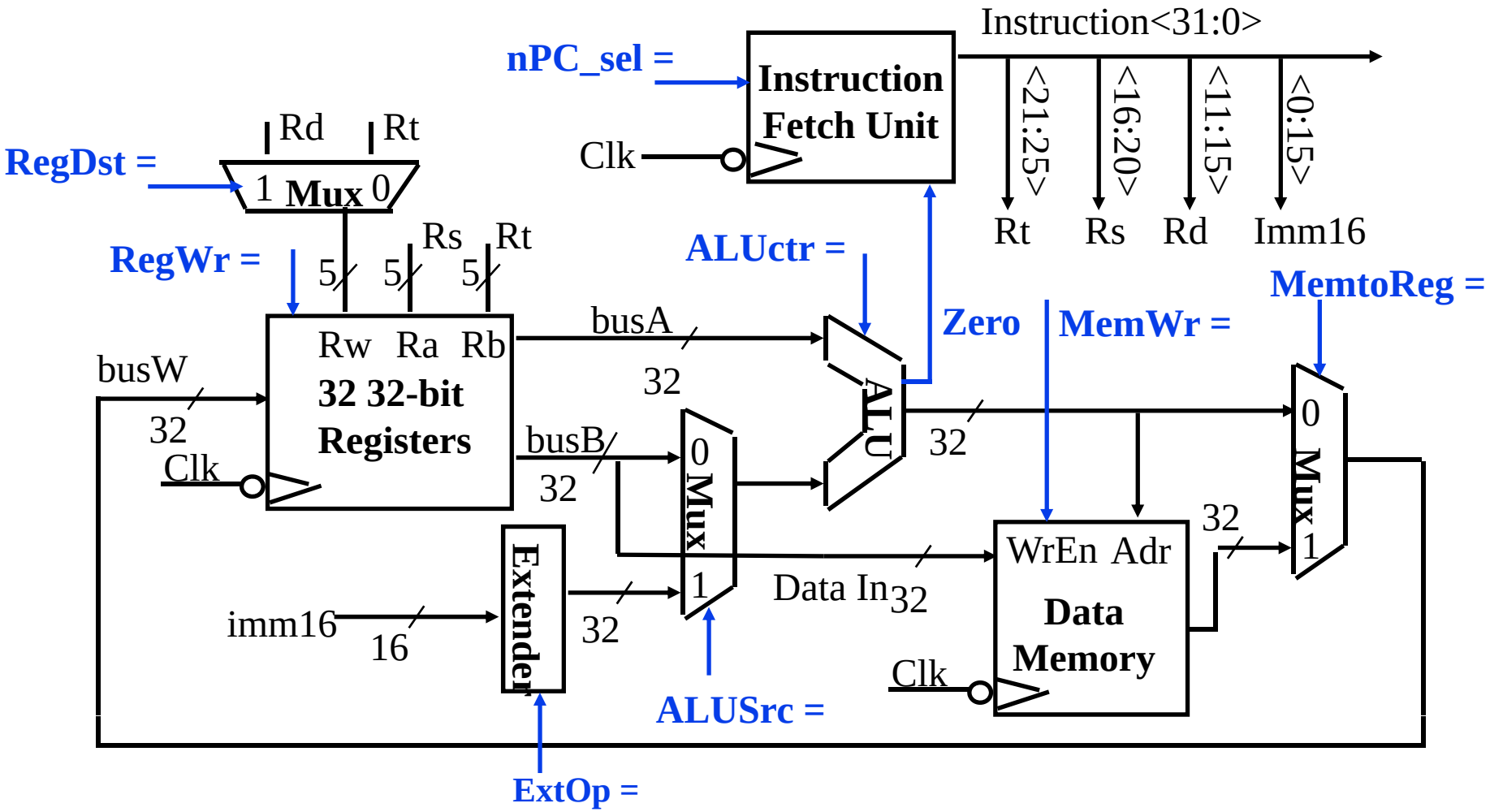
- PC = PC + 4
  - This is the same for all instructions except: Branch and Jump



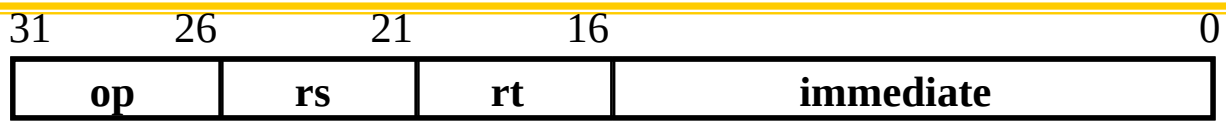
Single Cycle Datapath during Or Immediate?



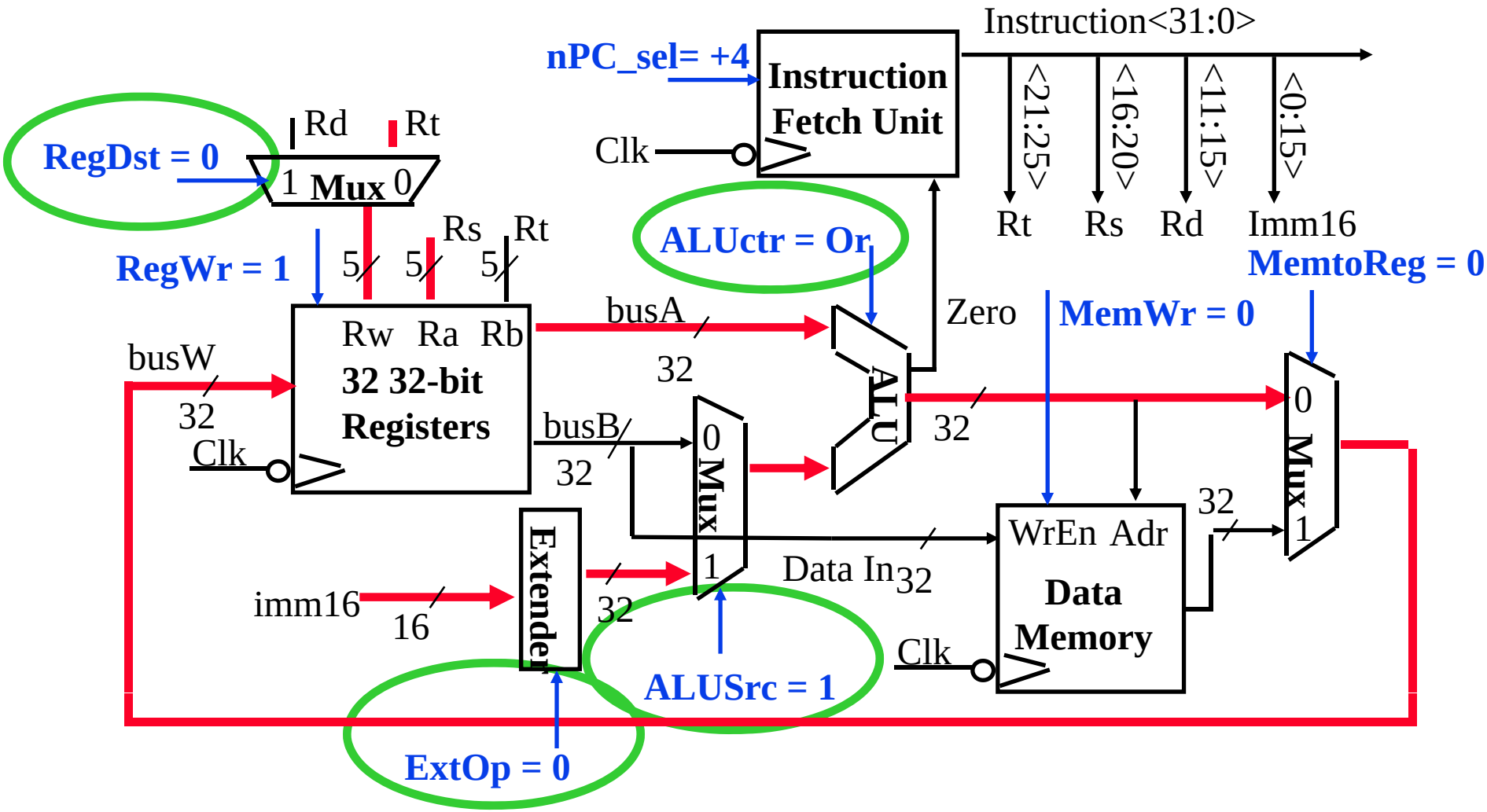
- **R[rt] = R[rs] OR ZeroExt[Imm16]**



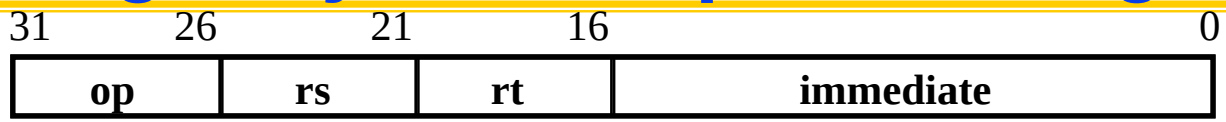
Single Cycle Datapath during Or Immediate?



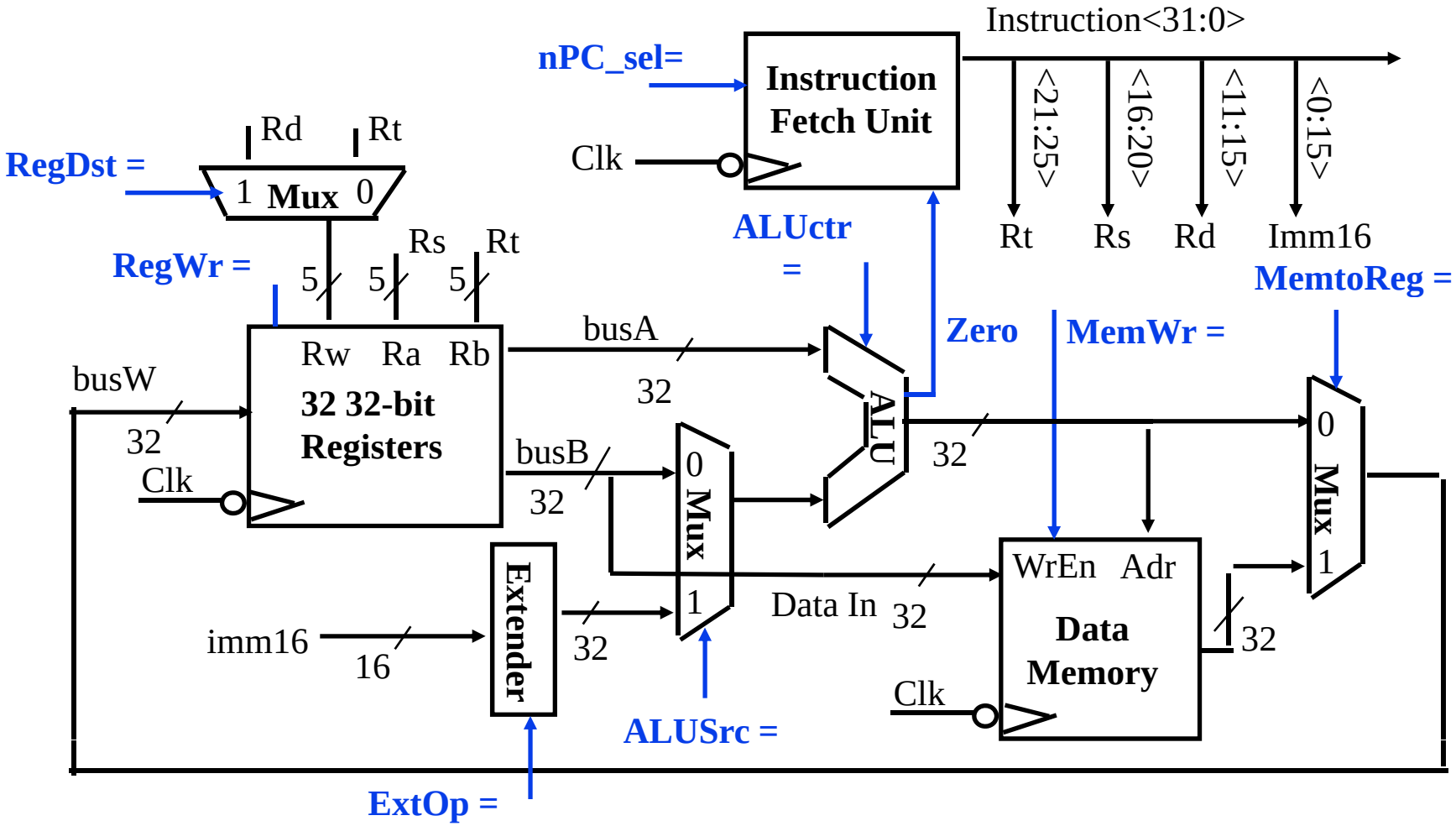
- **R[rt] = R[rs] OR ZeroExt[Imm16]**



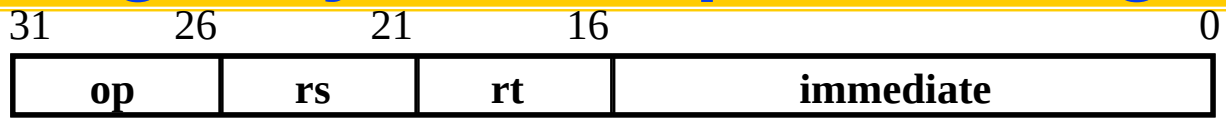
The Single Cycle Datapath during Load?



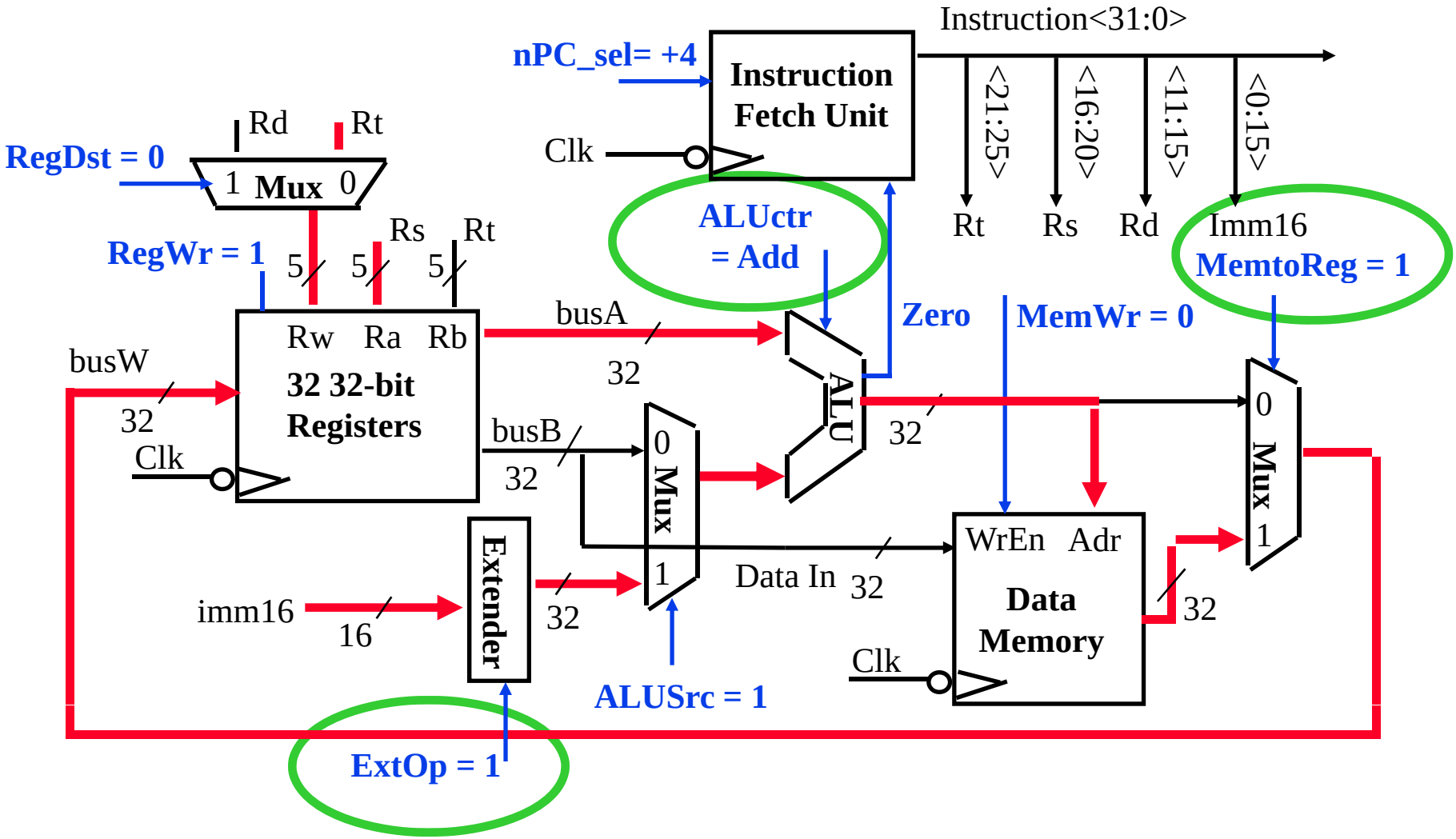
- R[rt] = Data Memory {R[rs] + SignExt[imm16]}



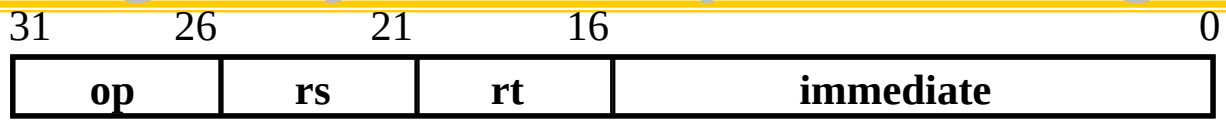
The Single Cycle Datapath during Load



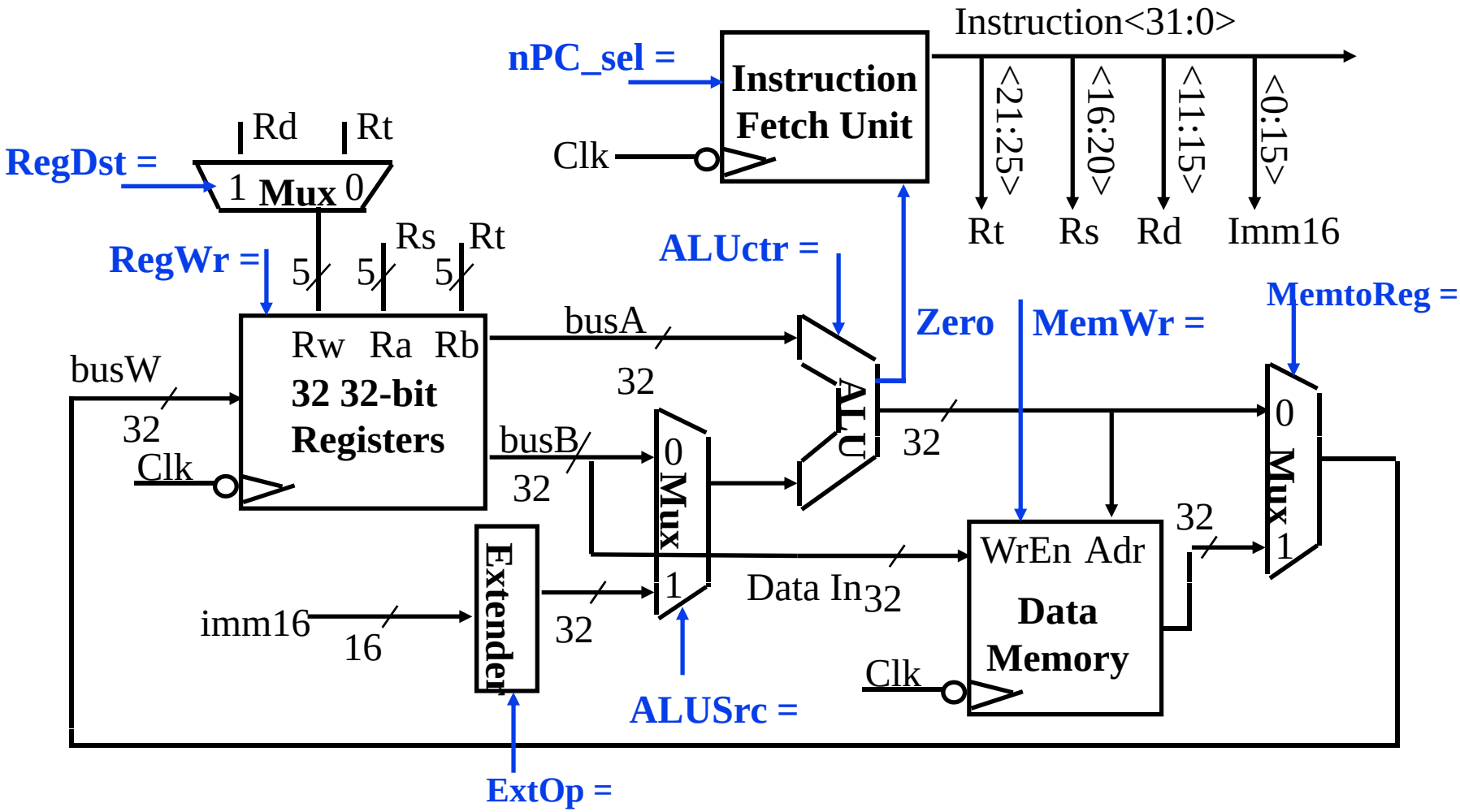
- R[rt] = Data Memory {R[rs] + SignExt[imm16]}



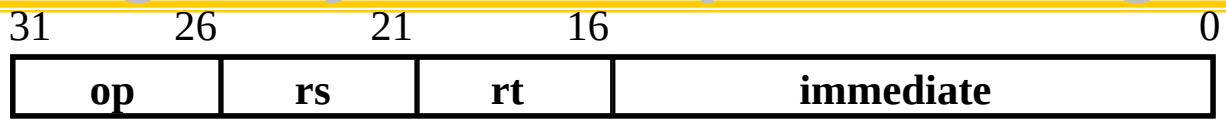
The Single Cycle Datapath during Store?



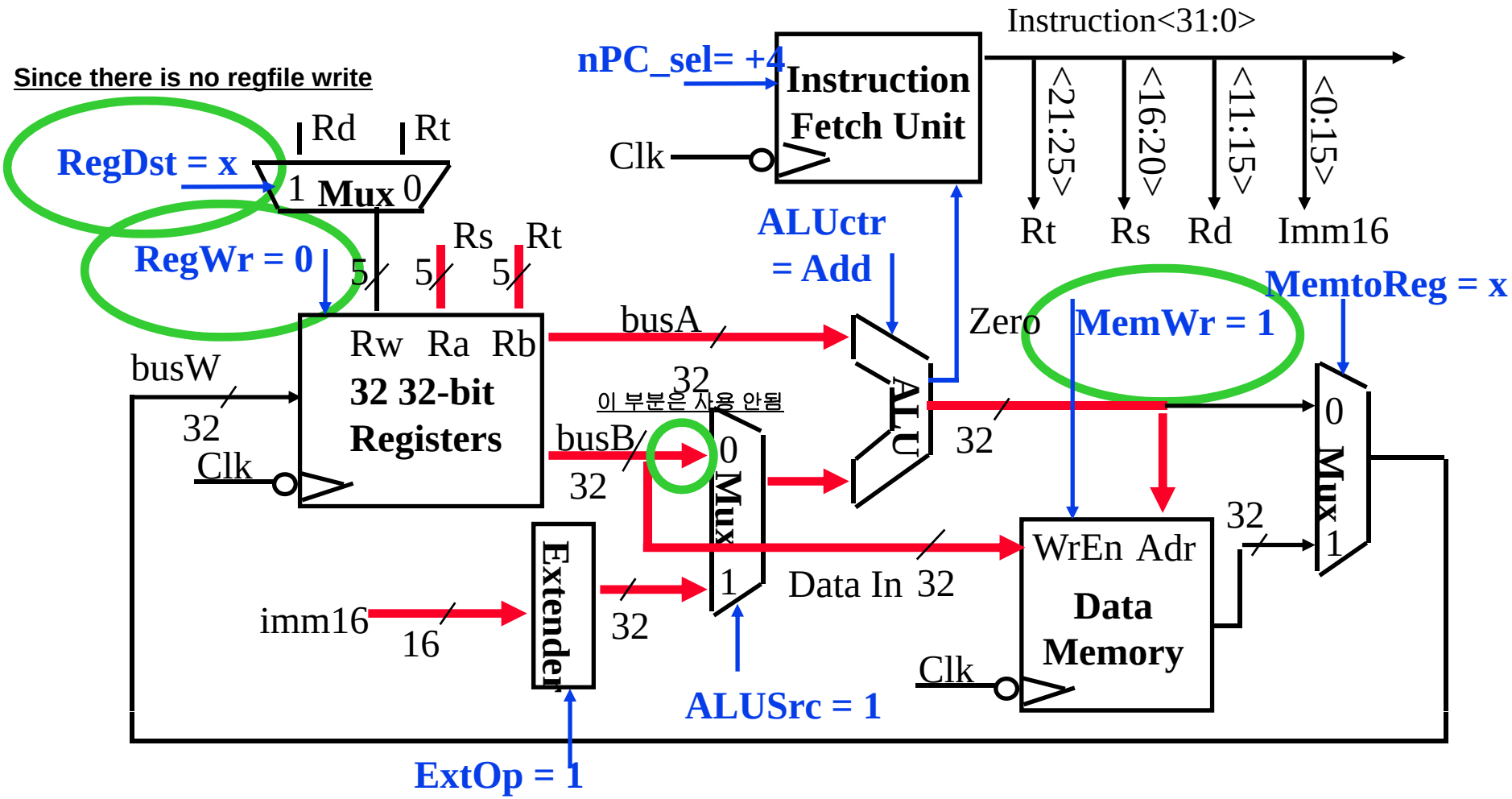
- **Data Memory {R[rs] + SignExt[imm16]} = R[rt]**



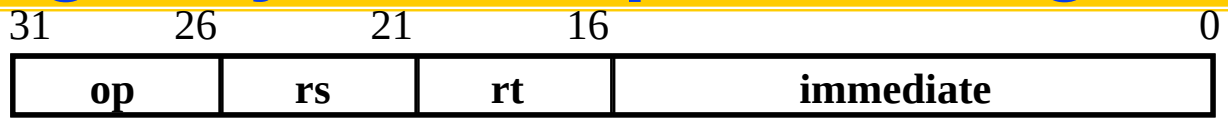
The Single Cycle Datapath during Store



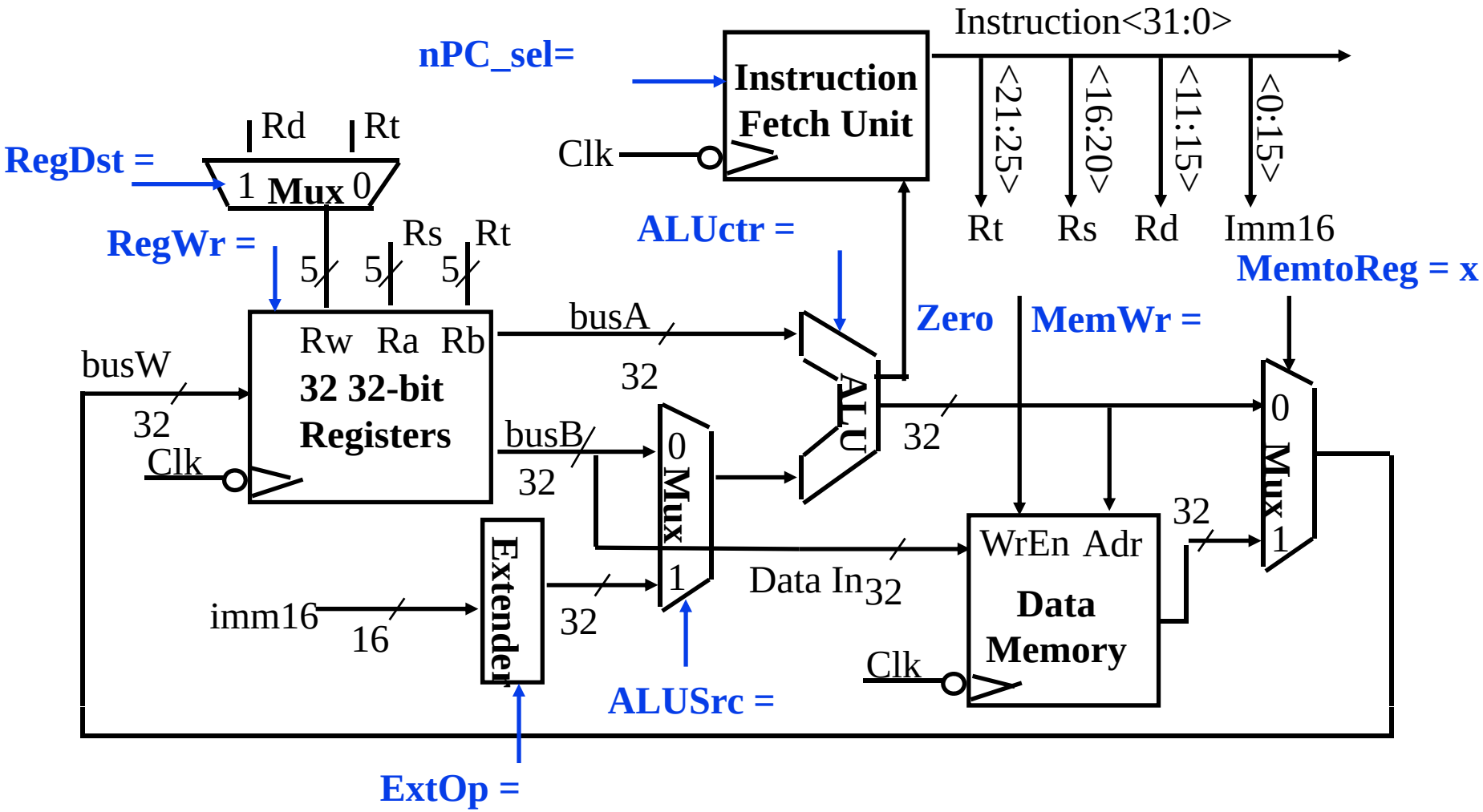
- **Data Memory {R[rs] + SignExt[imm16]} = R[rt]**



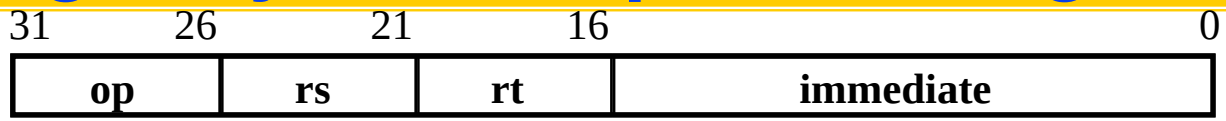
The Single Cycle Datapath during Branch?



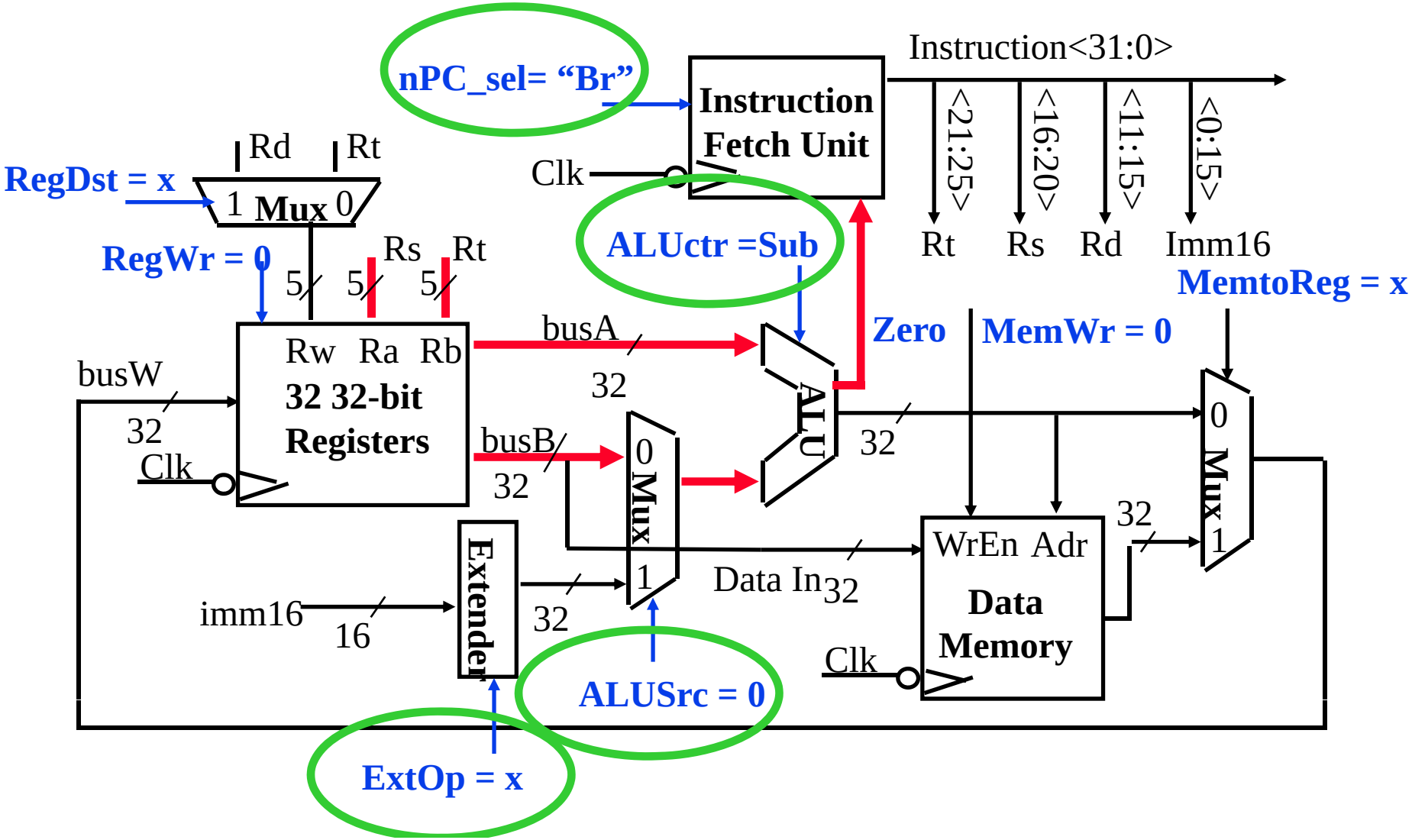
- if (R[rs] - R[rt] == 0) then Zero = 1 ; else Zero = 0



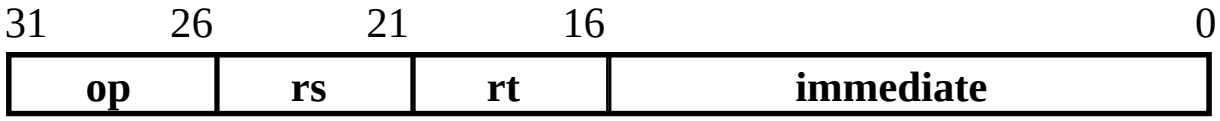
The Single Cycle Datapath during Branch



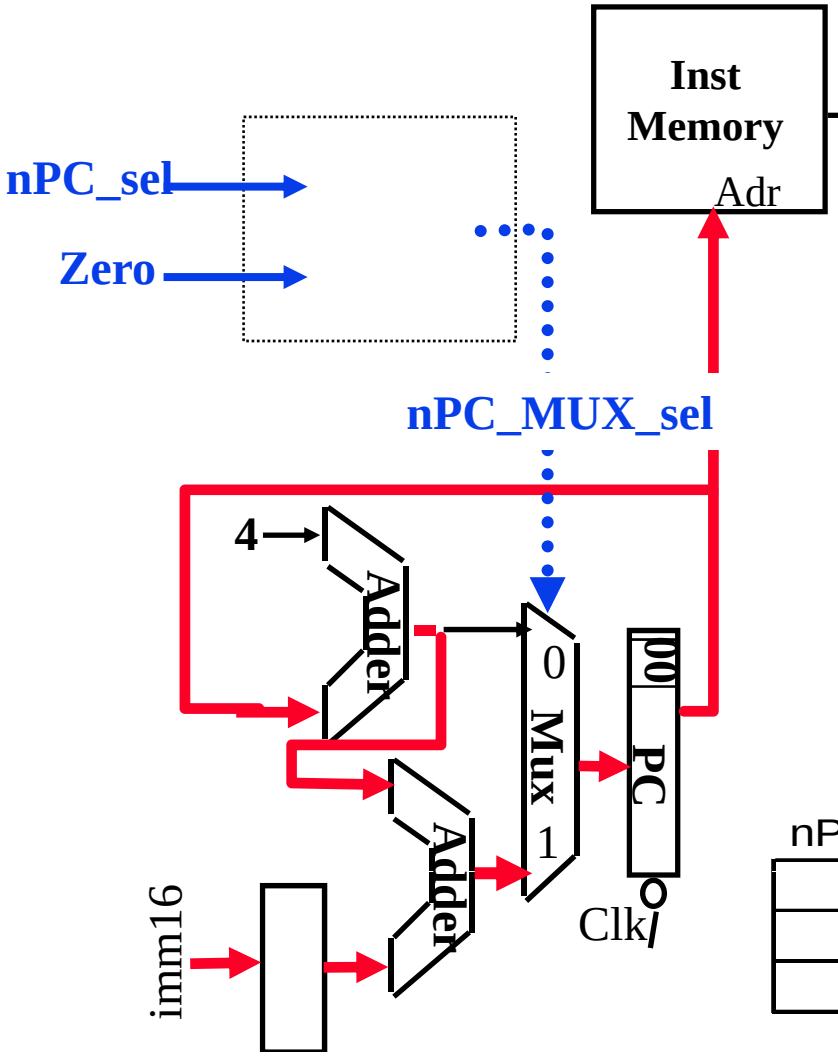
- if (R[rs] - R[rt] == 0) then Zero = 1 ; else Zero = 0



Instruction Fetch Unit at the End of Branch



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



- **What is encoding of nPC_sel?**
  - **Direct MUX select?**
  - **Branch / not branch**
- **Let's pick 2nd option**

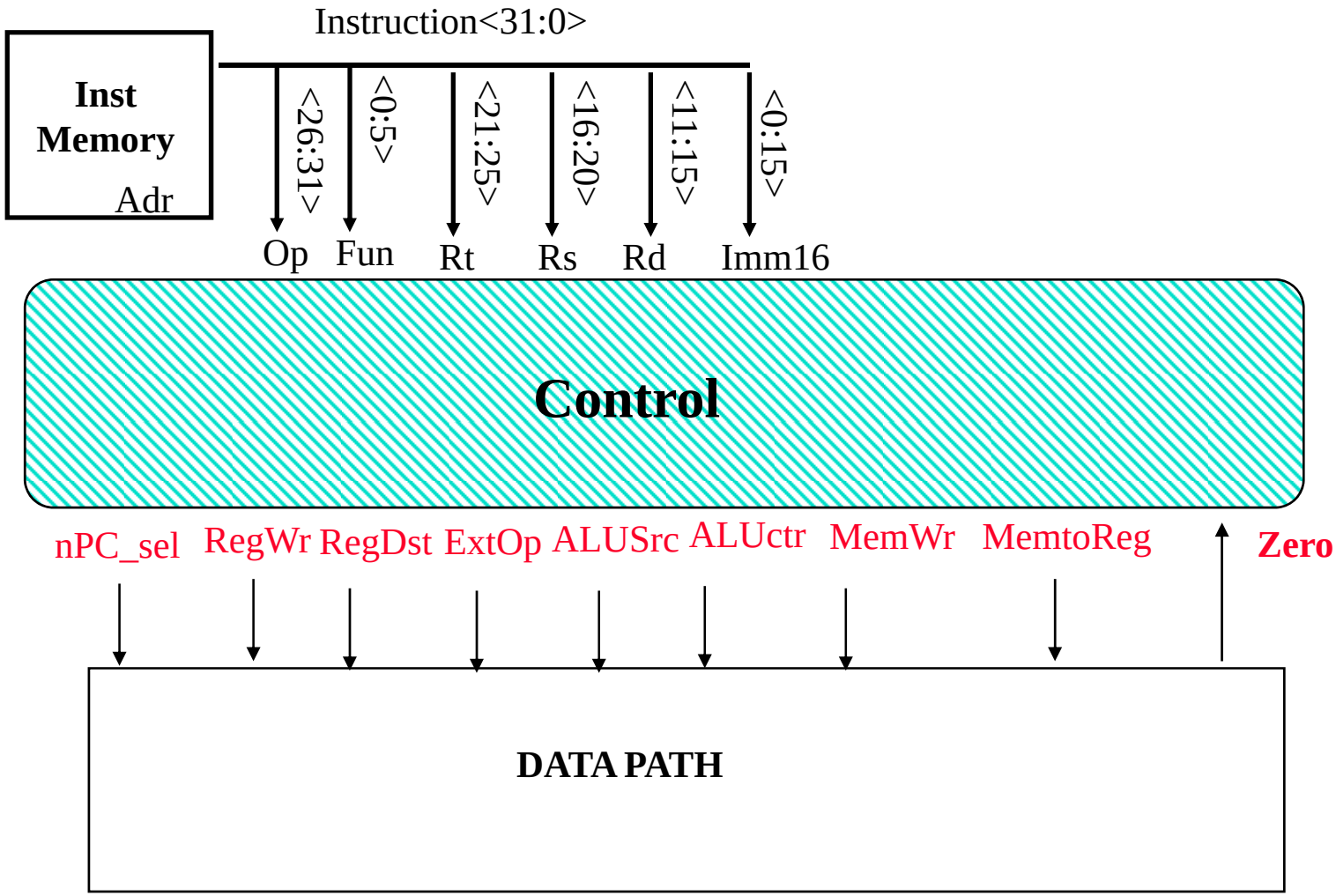
nPC_MUX_sel

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

Q: What logic gate?

AND gate

Step 4: Given Datapath: RTL -> Control



A Summary of the Control Signals (1/2)

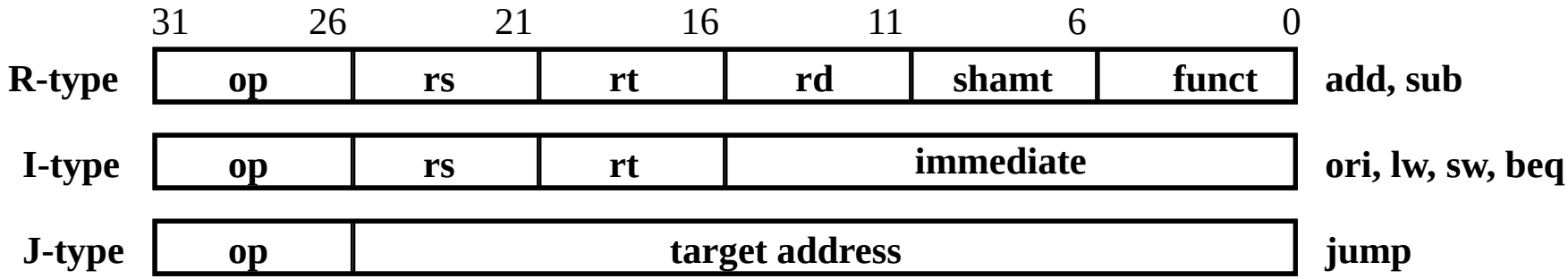
inst	Register Transfer	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”	
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, nPC_sel = “+4”	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, nPC_sel =“+4”	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr,	nPC_sel = “+4”
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, nPC_sel = “+4”	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = “Br”, ALUctr = “sub”	

A Summary of the Control Signals (2/2)

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

Column by column 으로 볼것



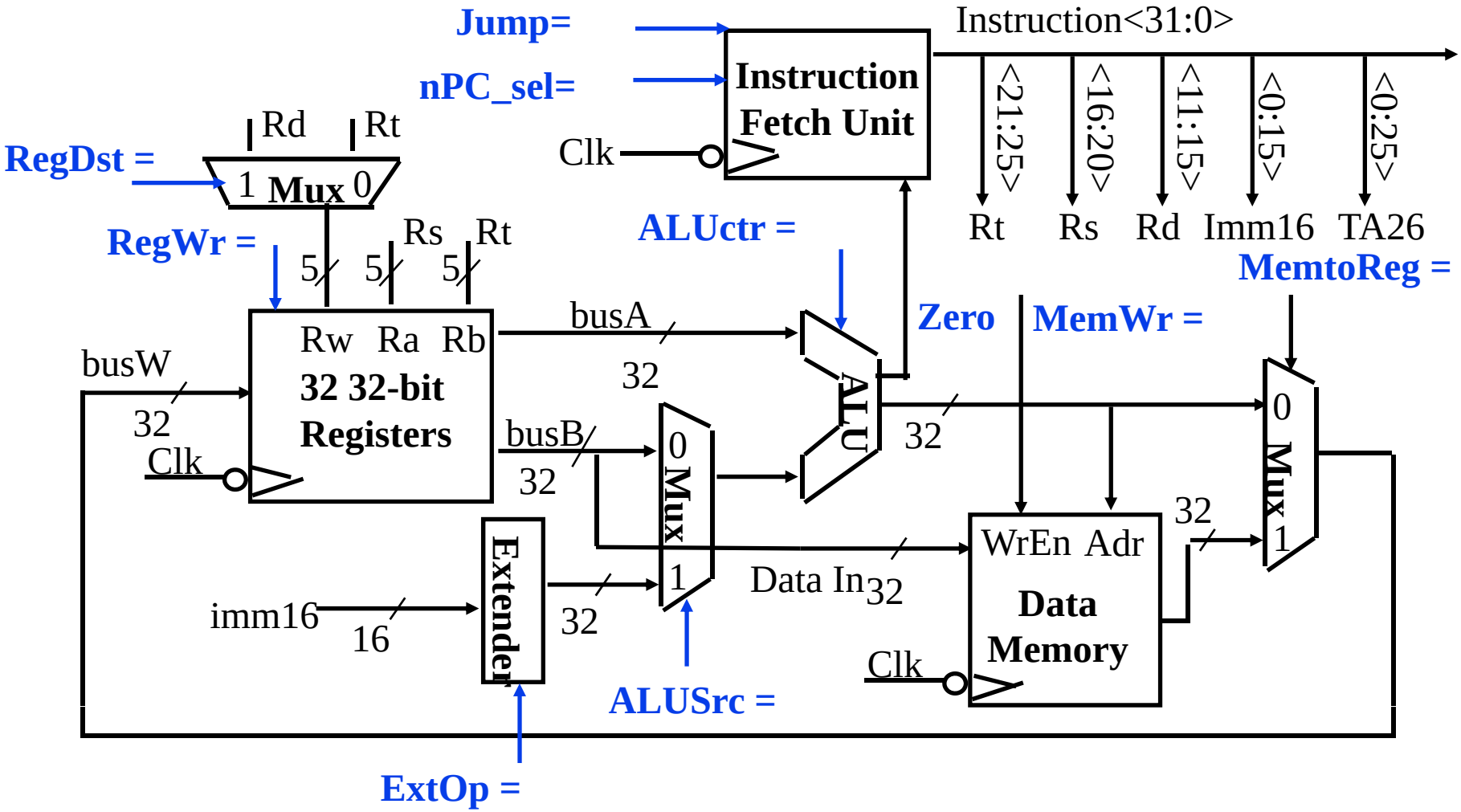
Administrivia

- **Project 2 Due Sunday**
- **HW 6 out tomorrow**
- **Slight shakeup of the schedule starting tomorrow (we're only doing one Control lecture)**
  - **This allows us to have the second midterm before the drop deadline, if that would be preferable**

☹이후 self study

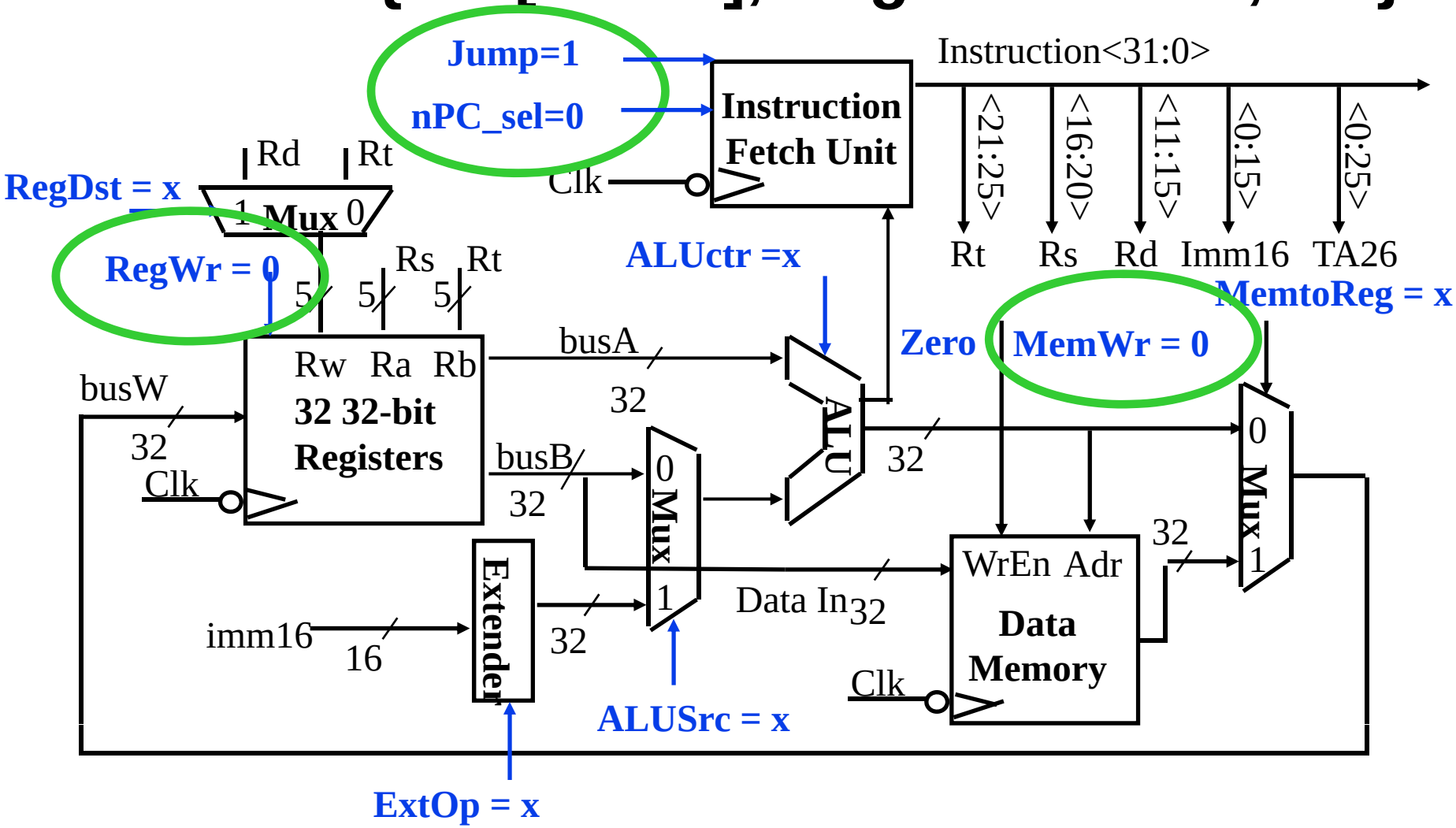
The Single Cycle Datapath during Jump

- New PC = { PC[31..28], target address, 00 }



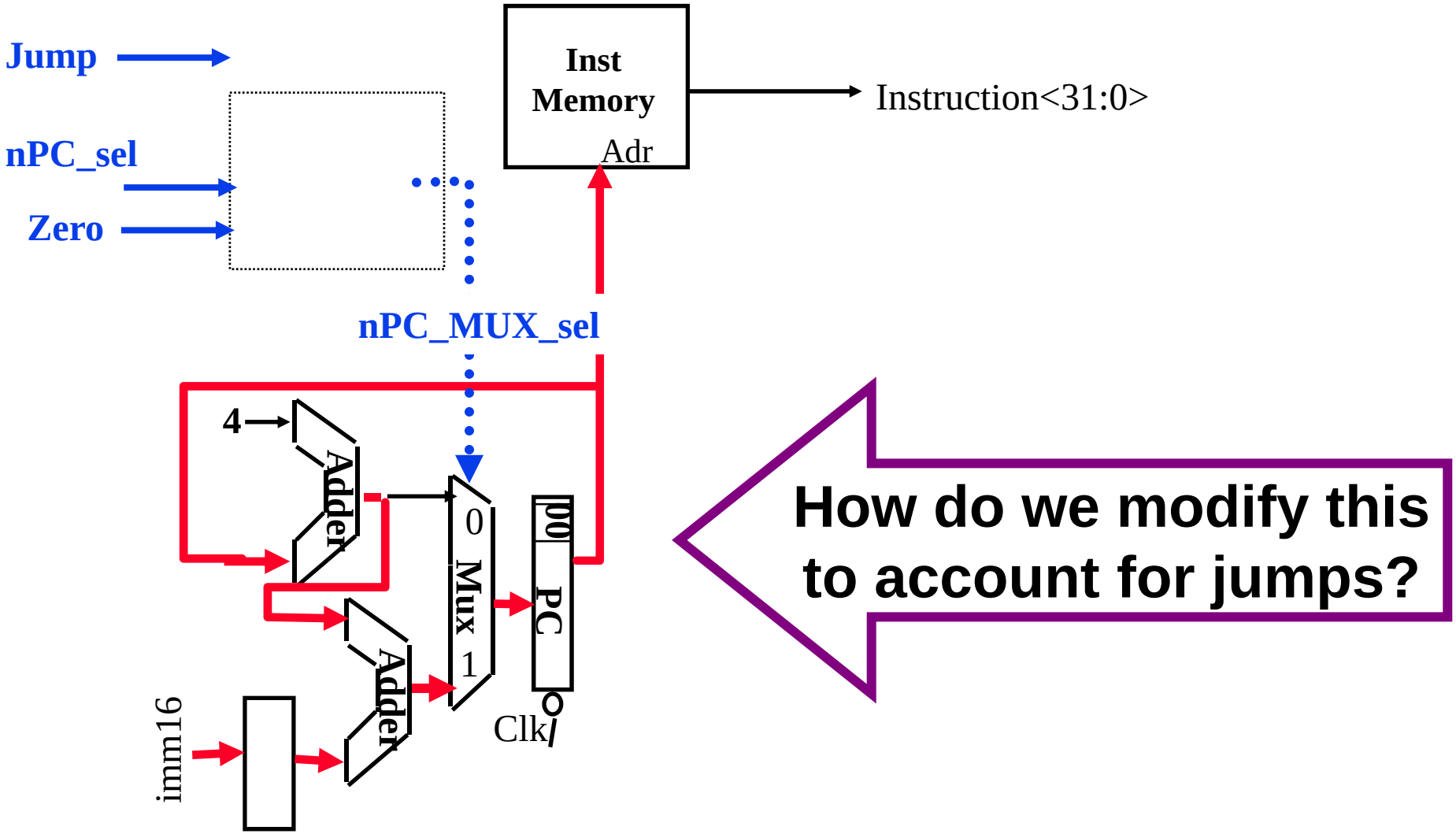
The Single Cycle Datapath during Jump

- New PC = { PC[31..28], target address, 00 }



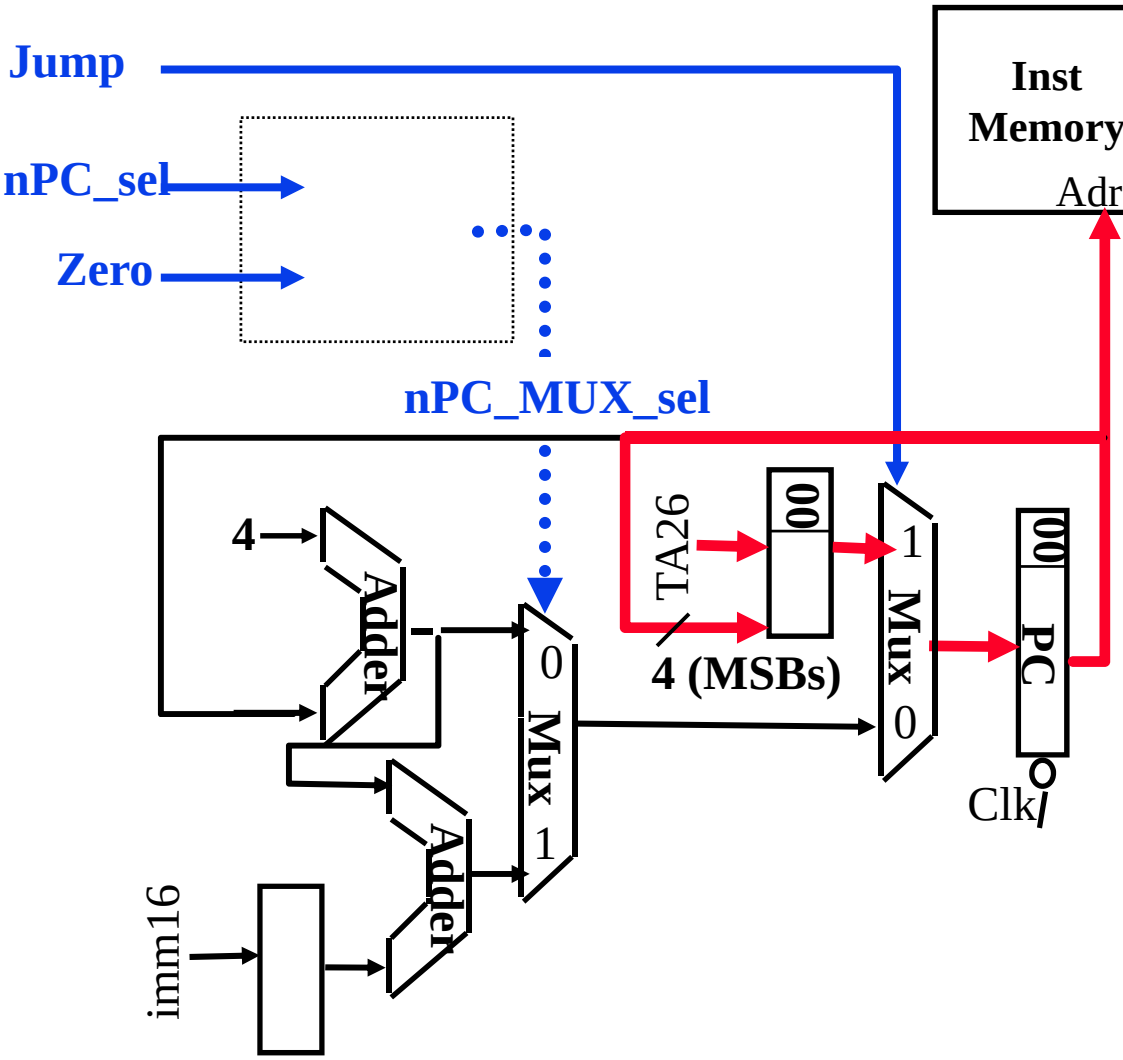
Instruction Fetch Unit at the End of Jump

- New PC = { PC[31..28], target address, 00 }



Instruction Fetch Unit at the End of Jump

- New PC = { PC[31..28], target address, 00 }

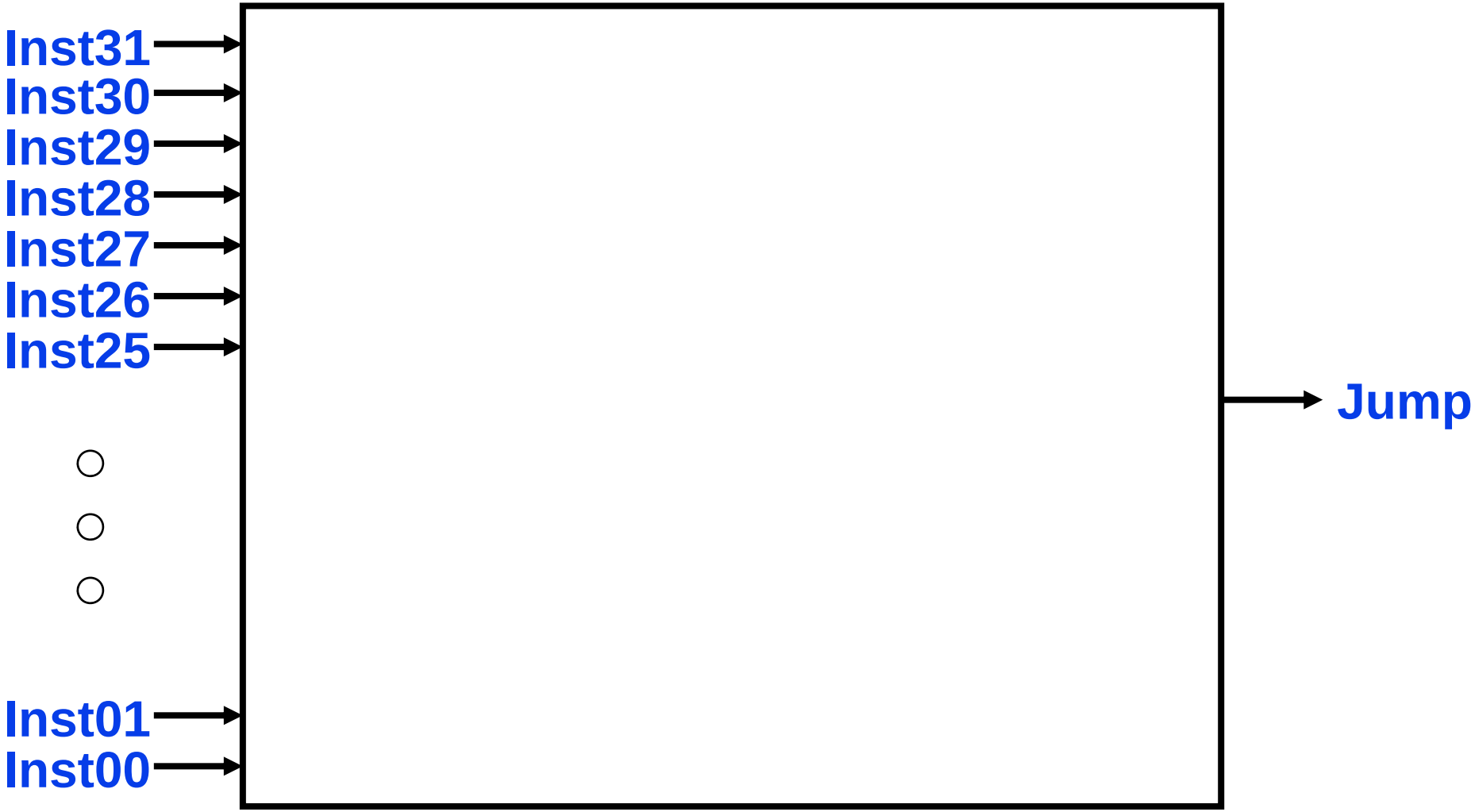


Query

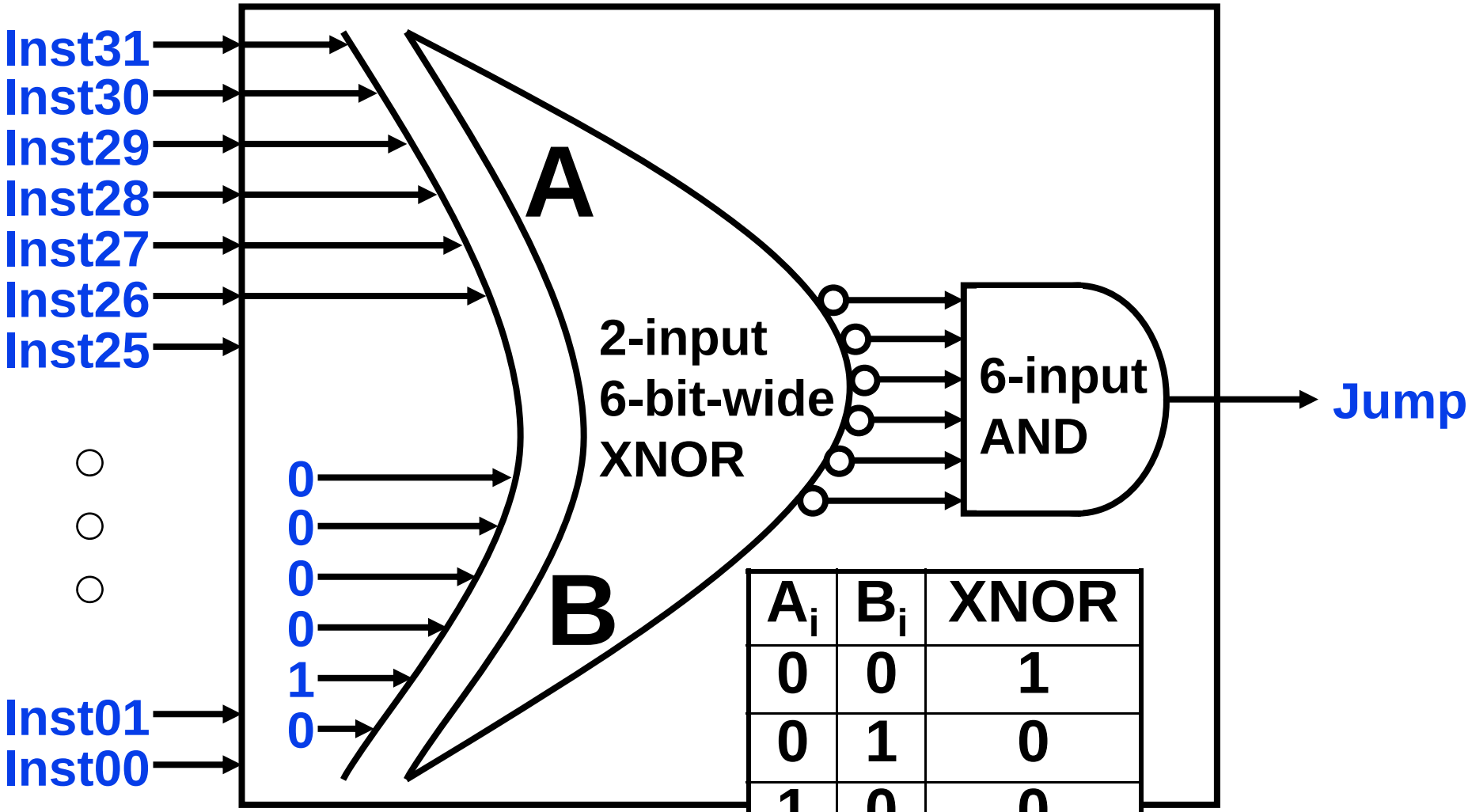
- Can Zero still get asserted?
- Does nPC_sel need to be 0?
  - If not, what?

All depends on designer

Build CL to implement Jump on paper now

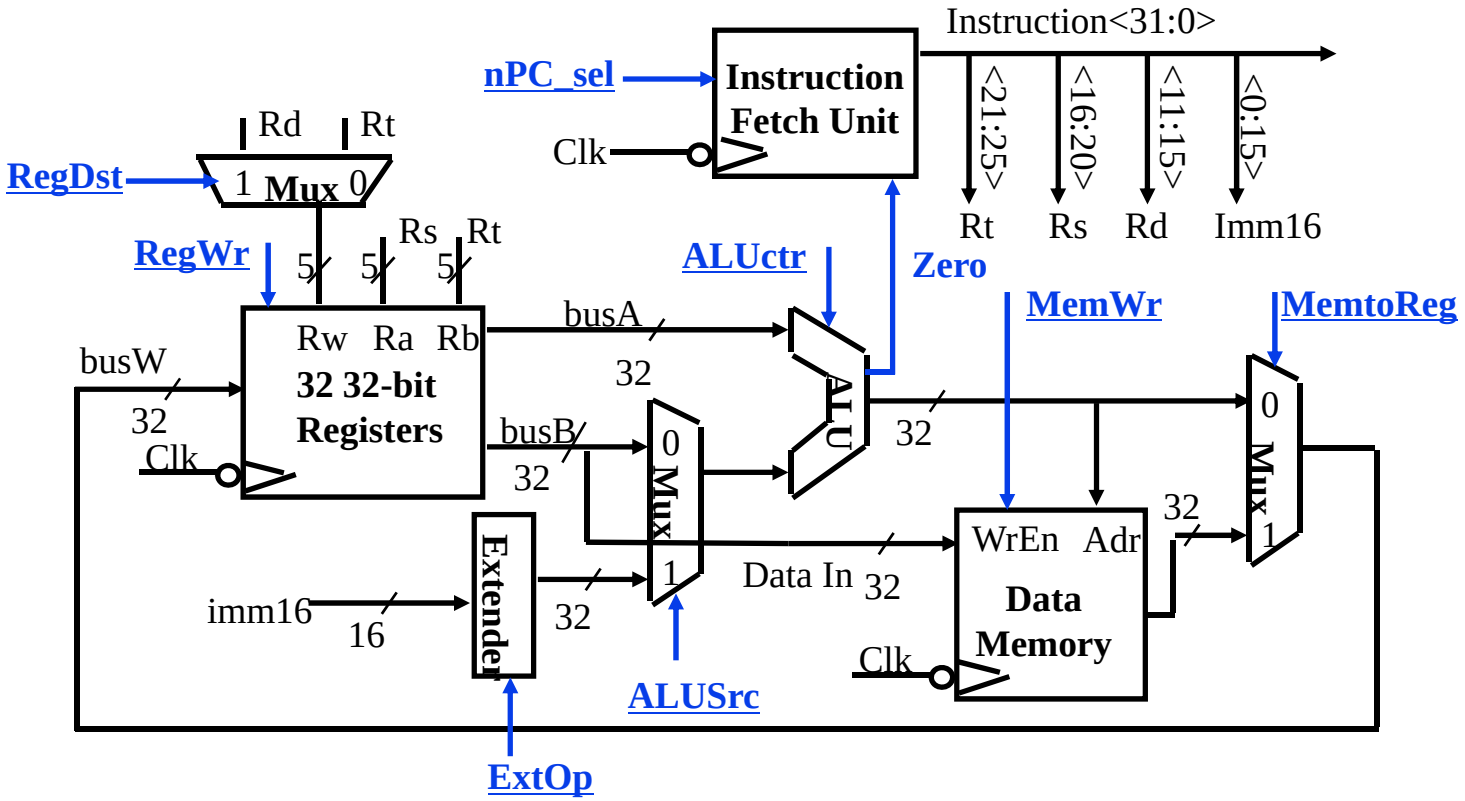


Build CL to implement Jump on paper now



A_i	B_i	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

Peer Instruction



A. MemToReg='x' & ALUctr='sub'. SUB or BEQ?

B. ALUctr='add'. Which 1 signal is different for all 3 of: ADD, LW, & SW? RegDst or ExtOp?

C. "Don't Care" signals are useful because we can simplify our Boolean equations?

And in Conclusion... Single cycle control

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