CSE 140
Computer Architecture

Lecture 3 – Single Cycle CPU

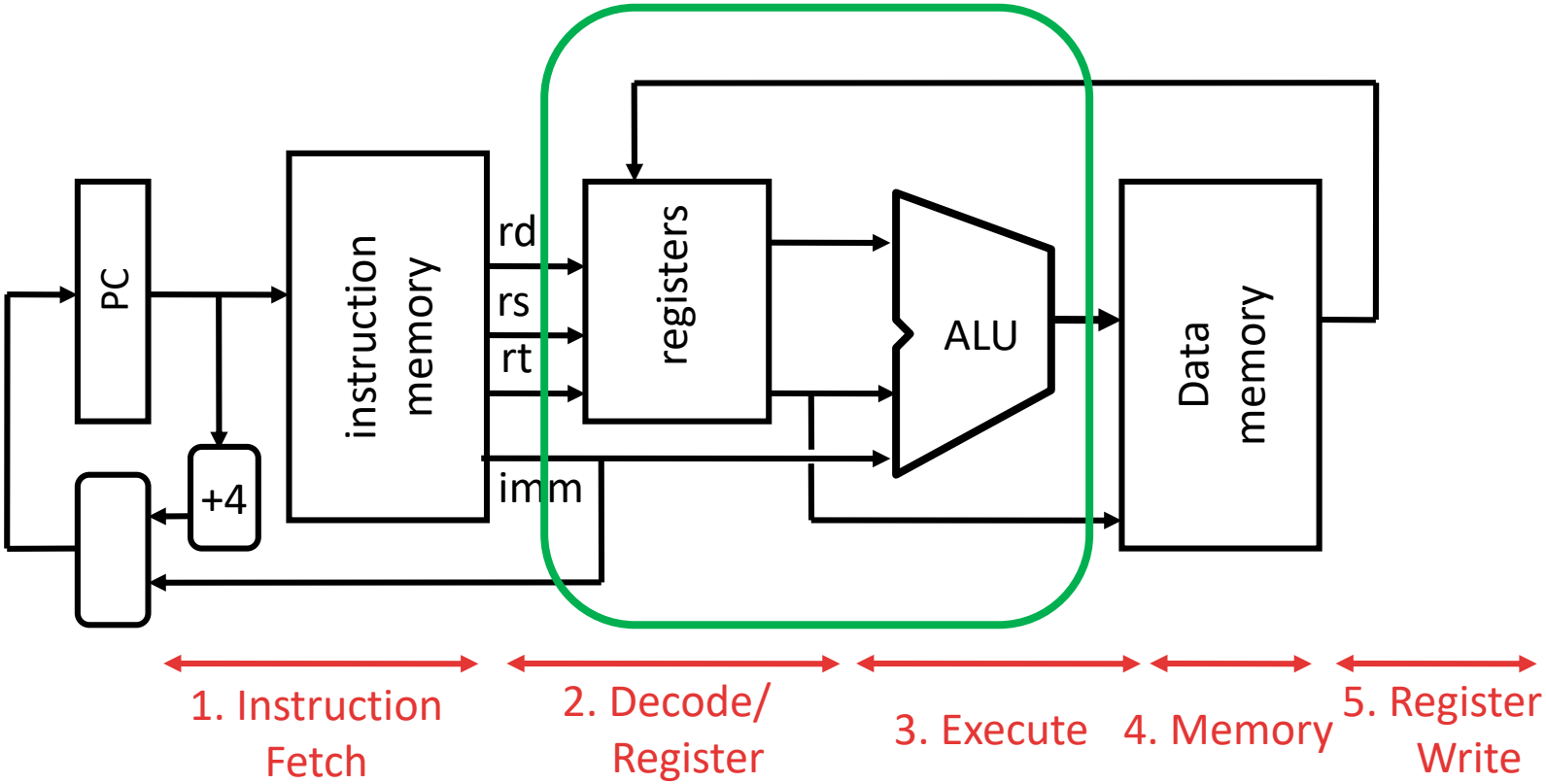
Announcement

- Lab #1 starts next week (9/9)
  - Due in one week
- Reading assignment #2
  - Chapter 2.5, 4.5 – 4.8
    - Do all Participation Activities in each section
    - Access through CatCourses
    - Due Thursday (9/12) at 11:59pm
  - Review CSE 31 materials (available at CatCourses)
    - Assembly language and machine code: Ch. 2.1-2.8, 2.9-2.10, 2.13
  - Quick review:
    - https://classroom.udacity.com/courses/ud219

Review

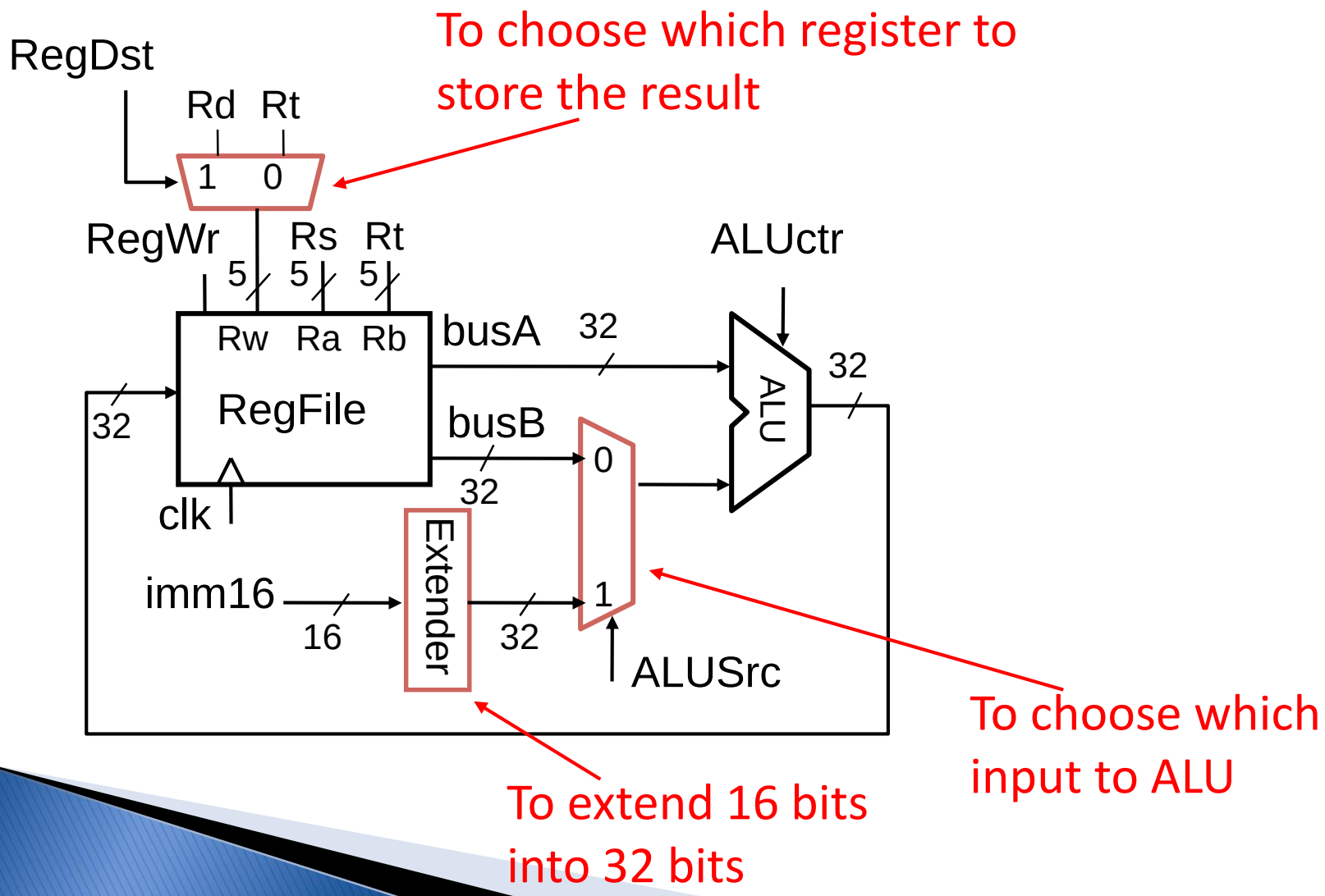
- CPU design involves Datapath, Control
  - Datapath in MIPS involves 5 CPU stages
    1. Instruction Fetch
    2. Instruction Decode & Register Read
    3. ALU (Execute)
    4. Memory
    5. Register Write

Generic Steps of Datapath



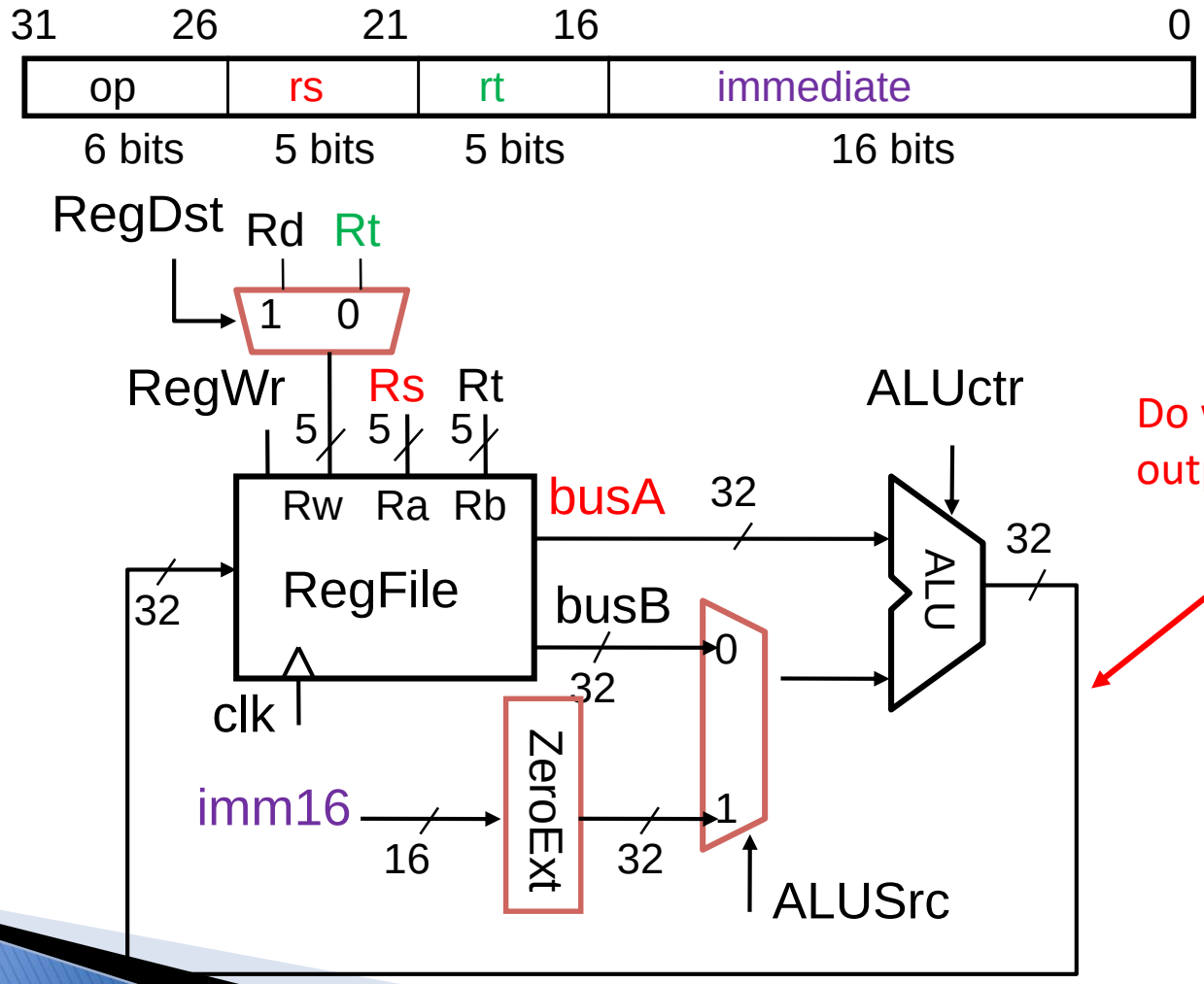
How do we handle the different register usage between r-type and i-type instructions?

A zoomed in version of RegFile and ALU



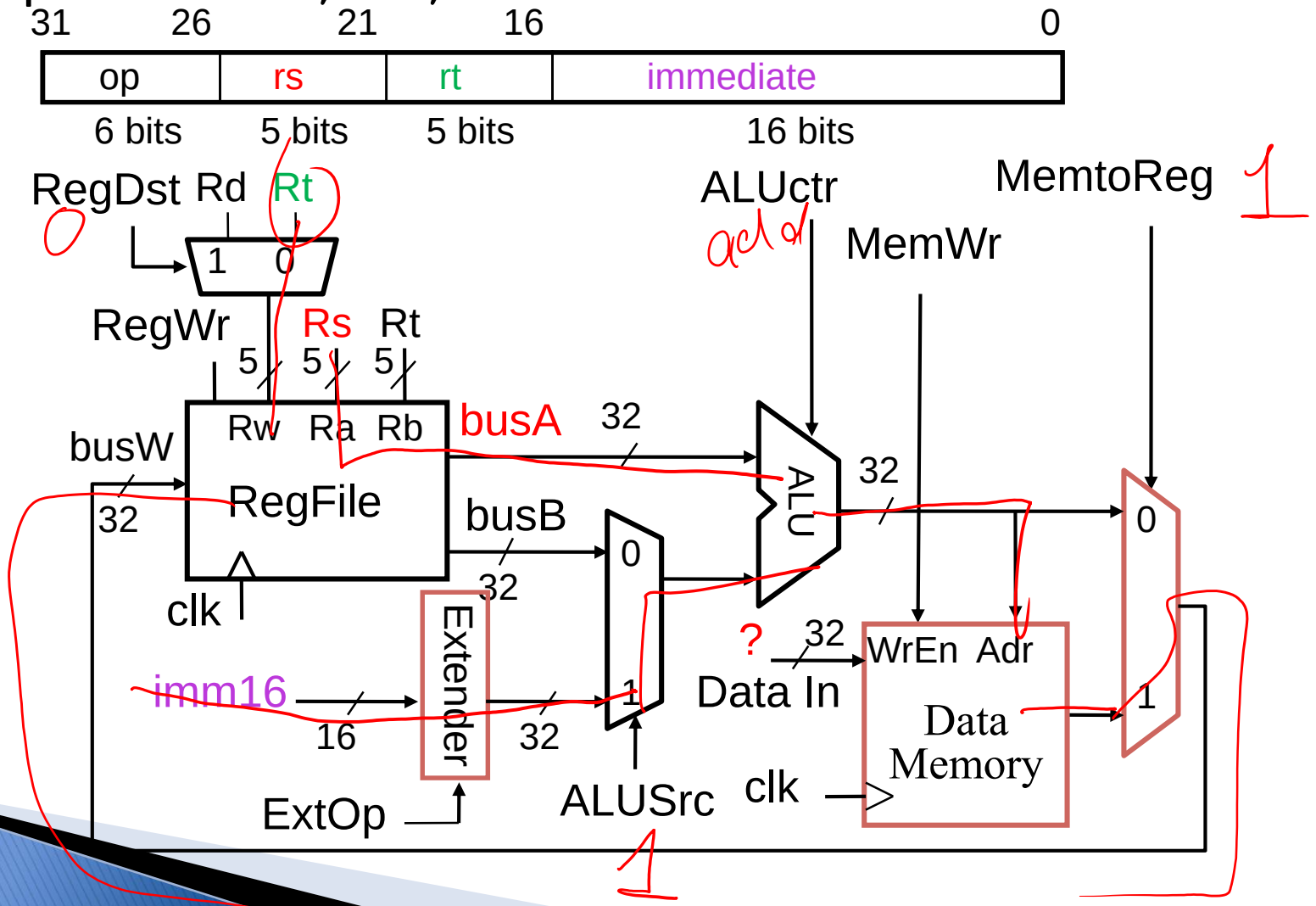
Load Memory

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



Load Memory

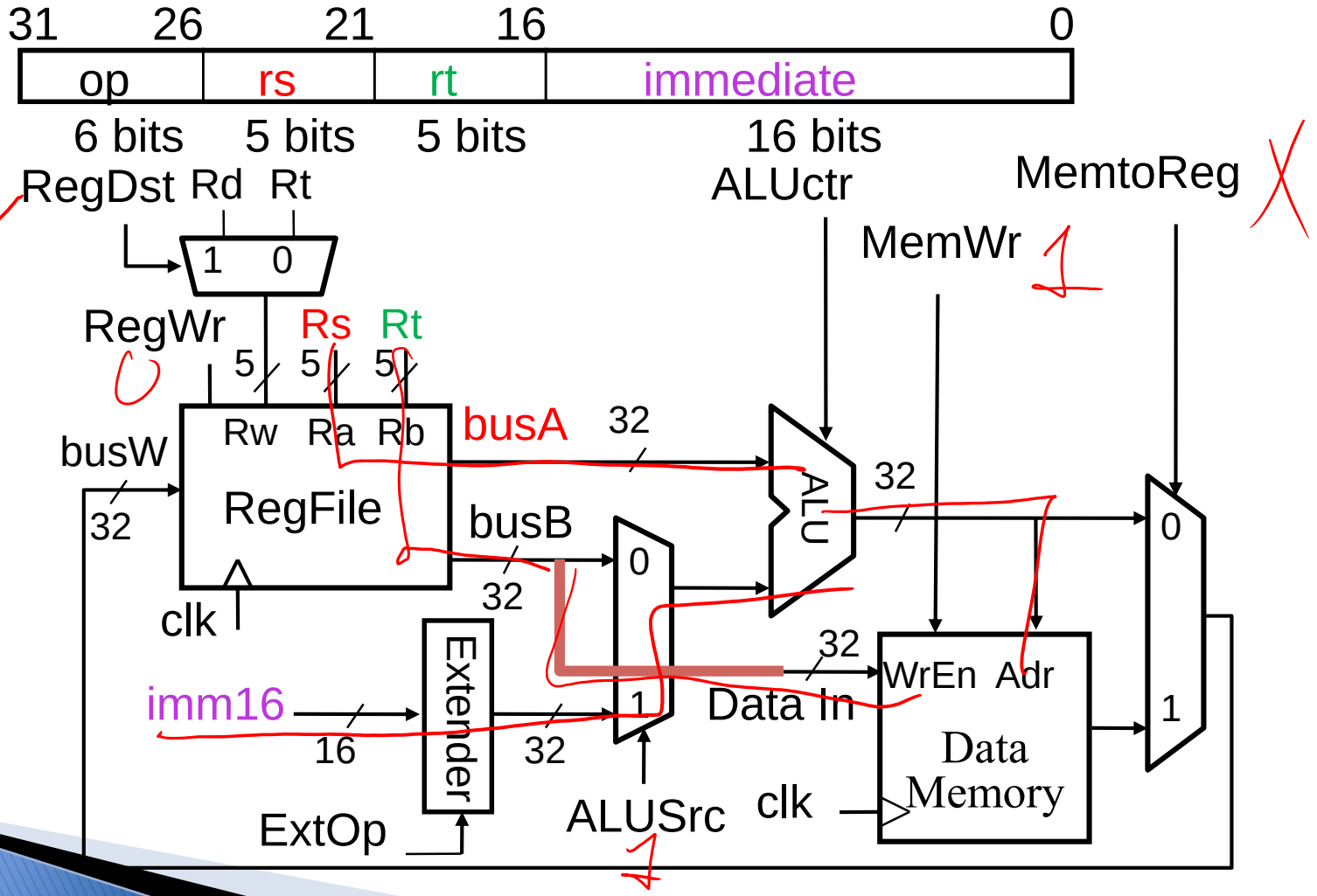
- R[rt] = Mem[R[rs] + SignExt[imm16]]
- Example: `lw rt,rs,imm16`



Store Memory

- Mem[R[rs] + SignExt[imm16]] = R[rt]

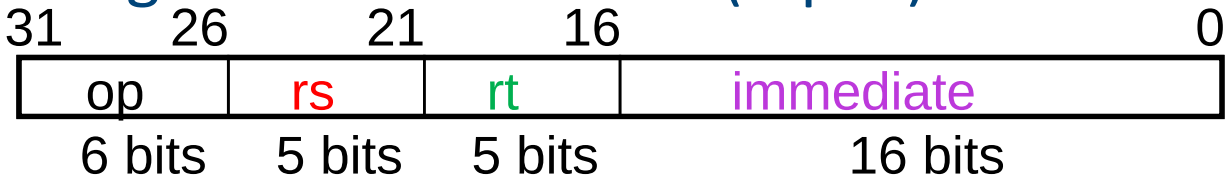
  Ex.: `sw rt, rs, imm16`

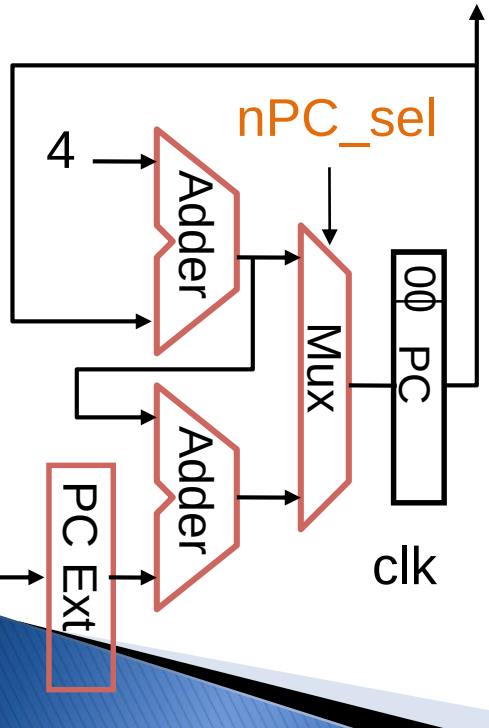


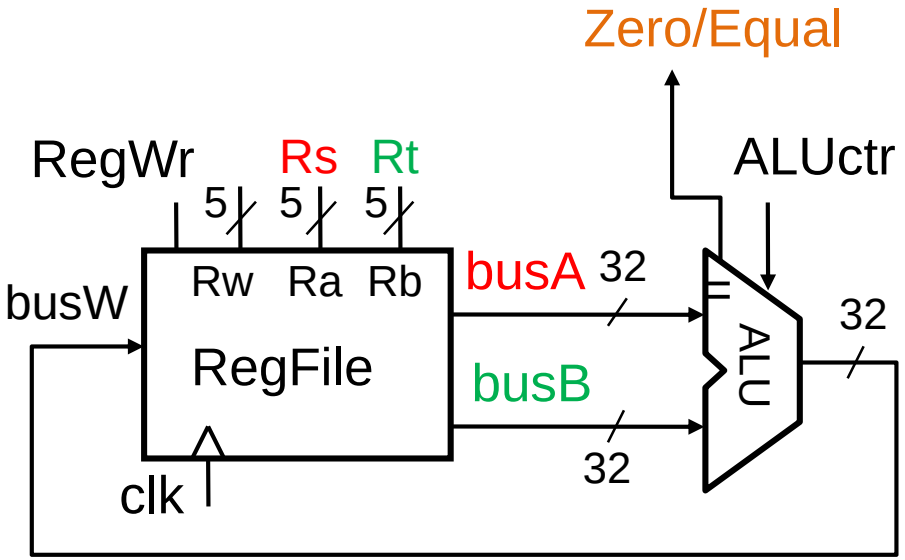
Datapath for Branch Operations

- beq rs, rt, imm16

  Datapath generates condition (equal)

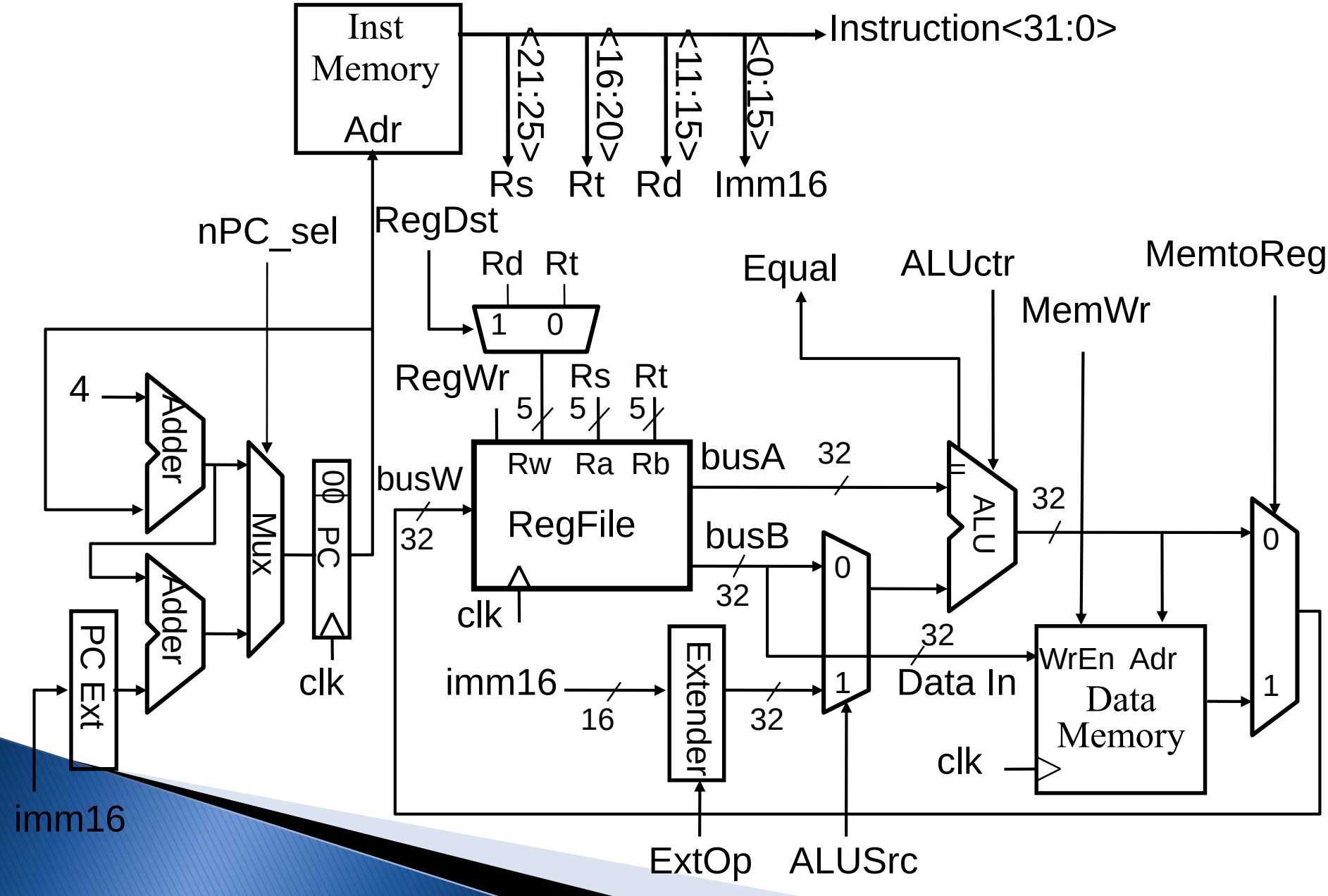




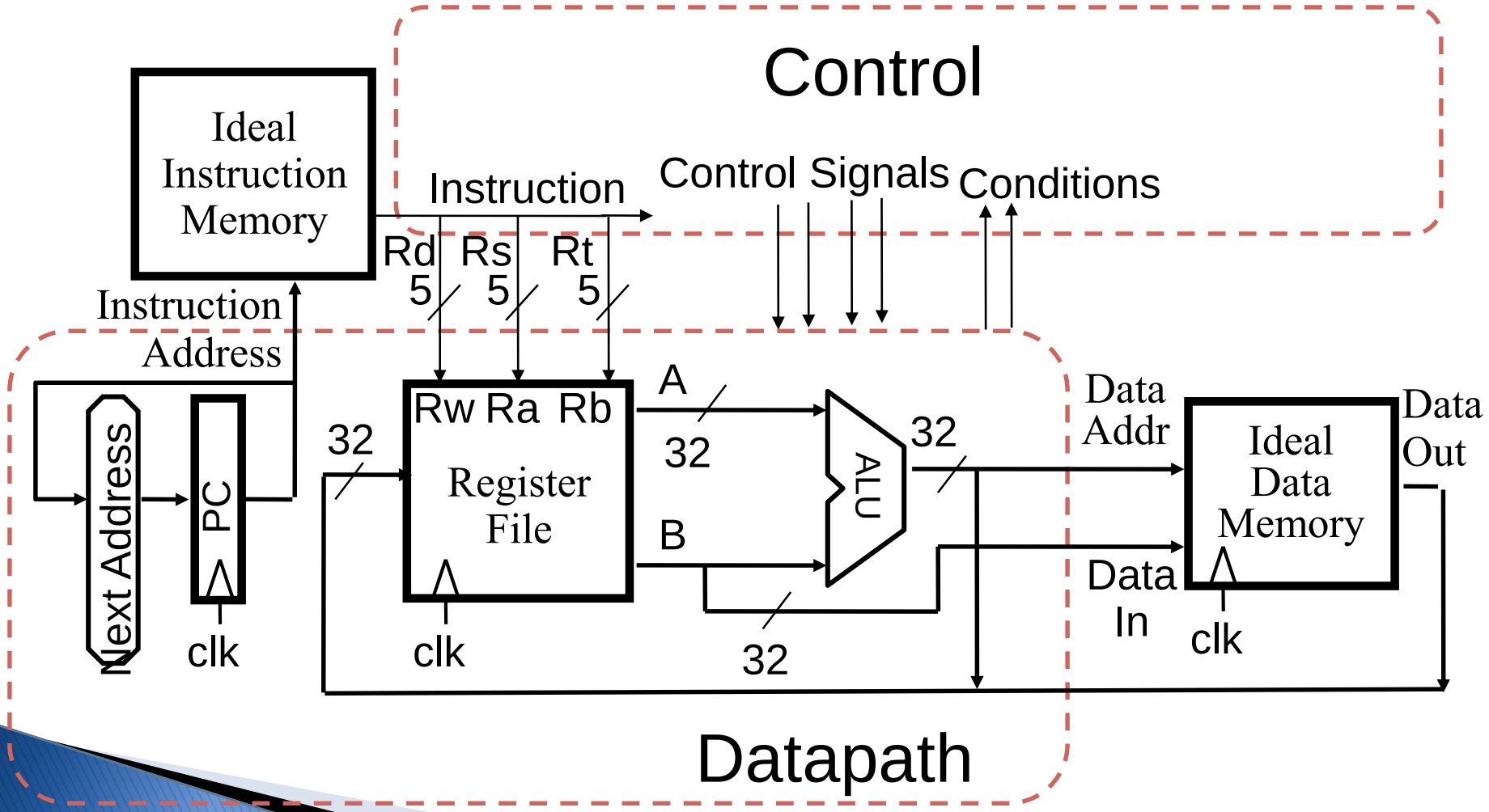


Already have mux, adder, need equal compare (sub?) to update for PC

Single Cycle Datapath

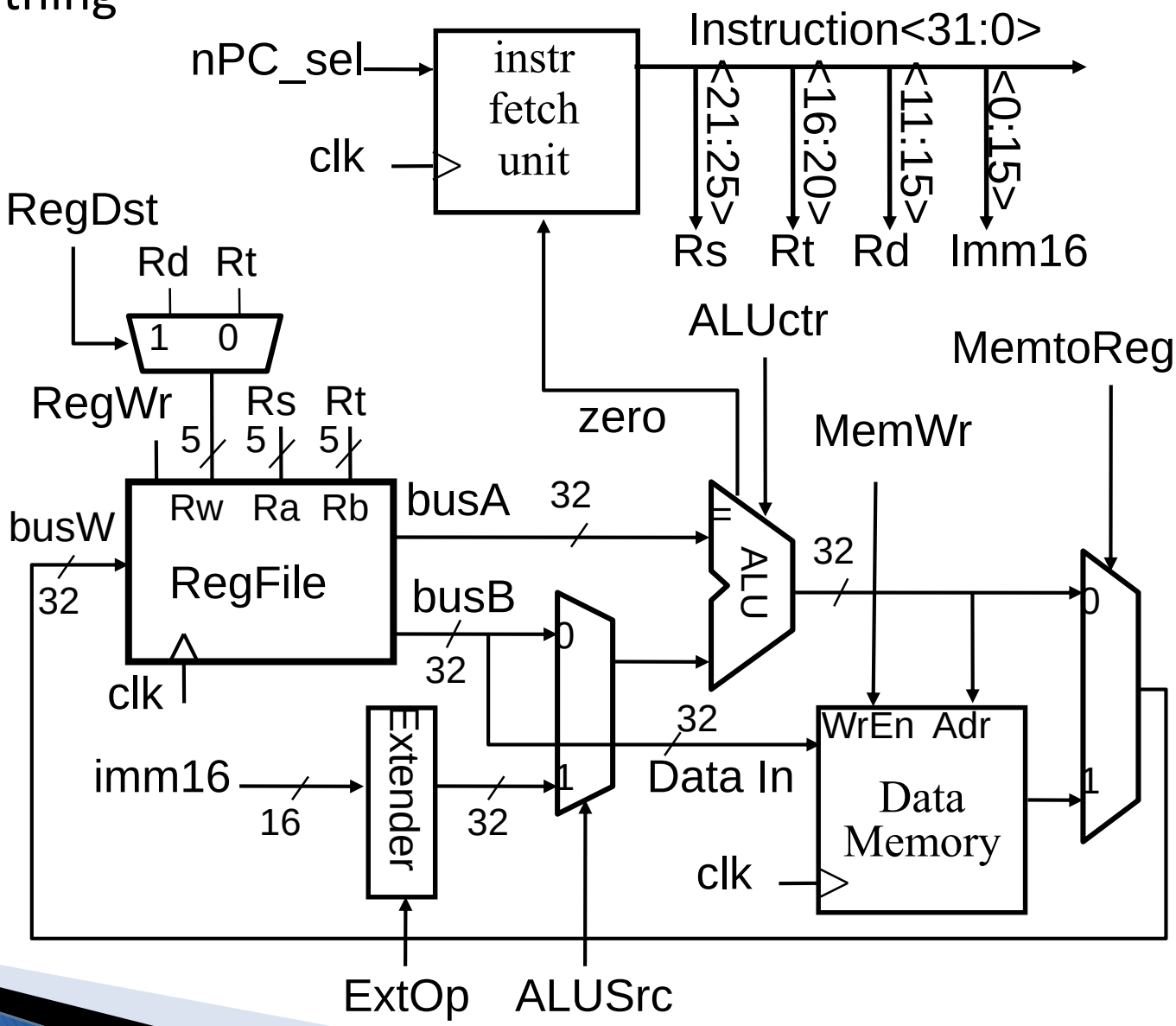


Abstract View of the Implementation



A Single Cycle Datapath

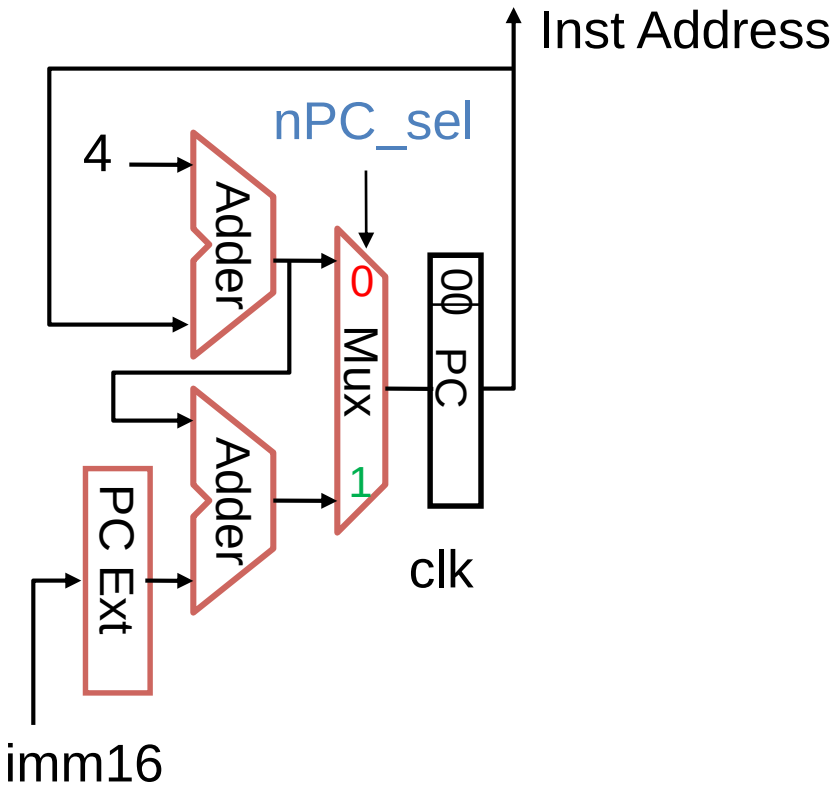
- We have everything except control signals



Meaning of the Control Signals

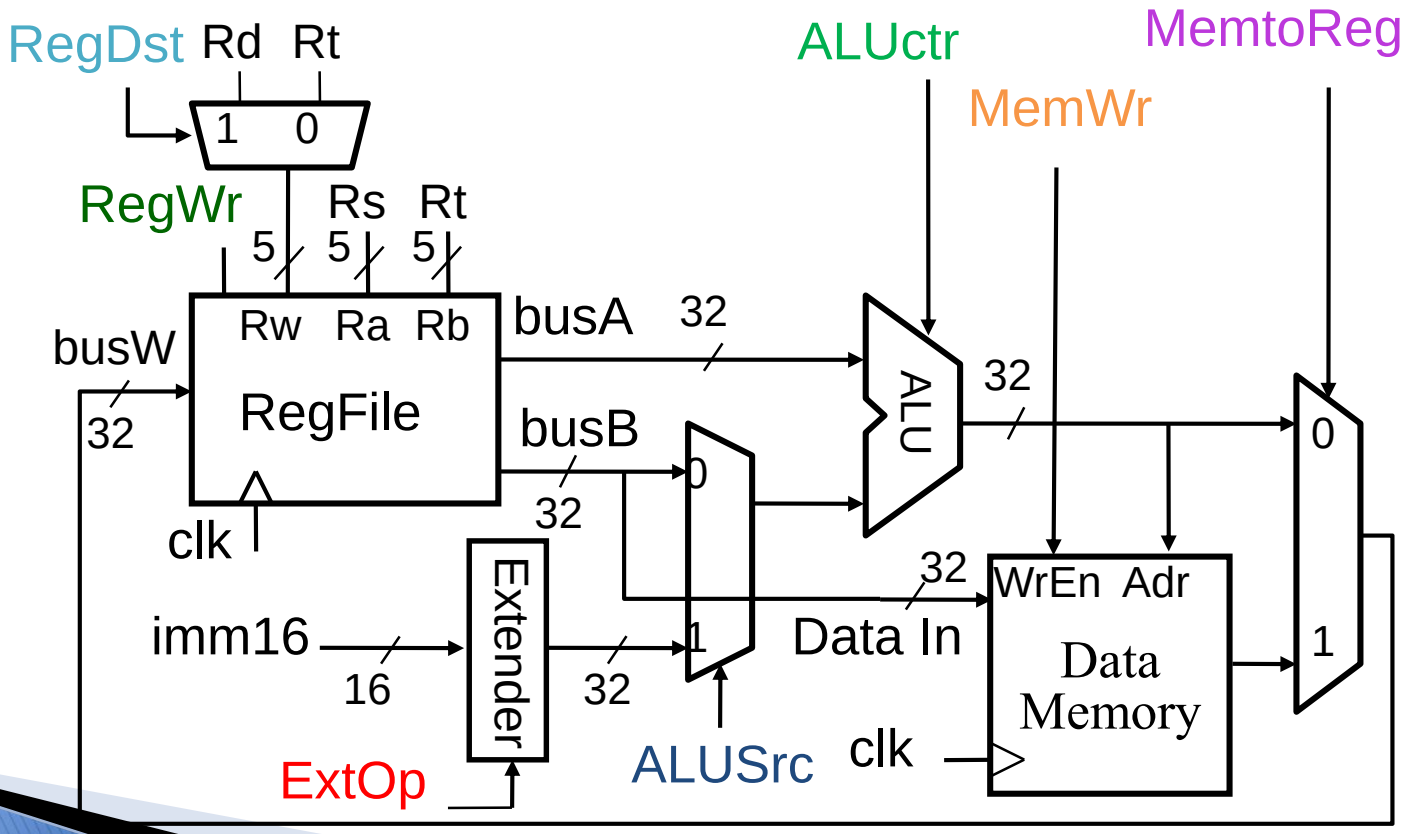
- nPC_sel: "+4": 0 ⇒ PC <– PC + 4
  "br": 1 ⇒ PC <– PC + 4 + {SignExt(Im16) , 00 }

"n"=next

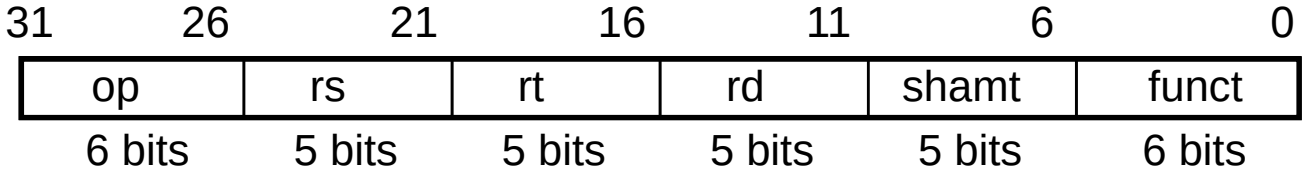


Meaning of the Control Signals

- ExtOp: "zero", "sign"
- 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



The Add Instruction



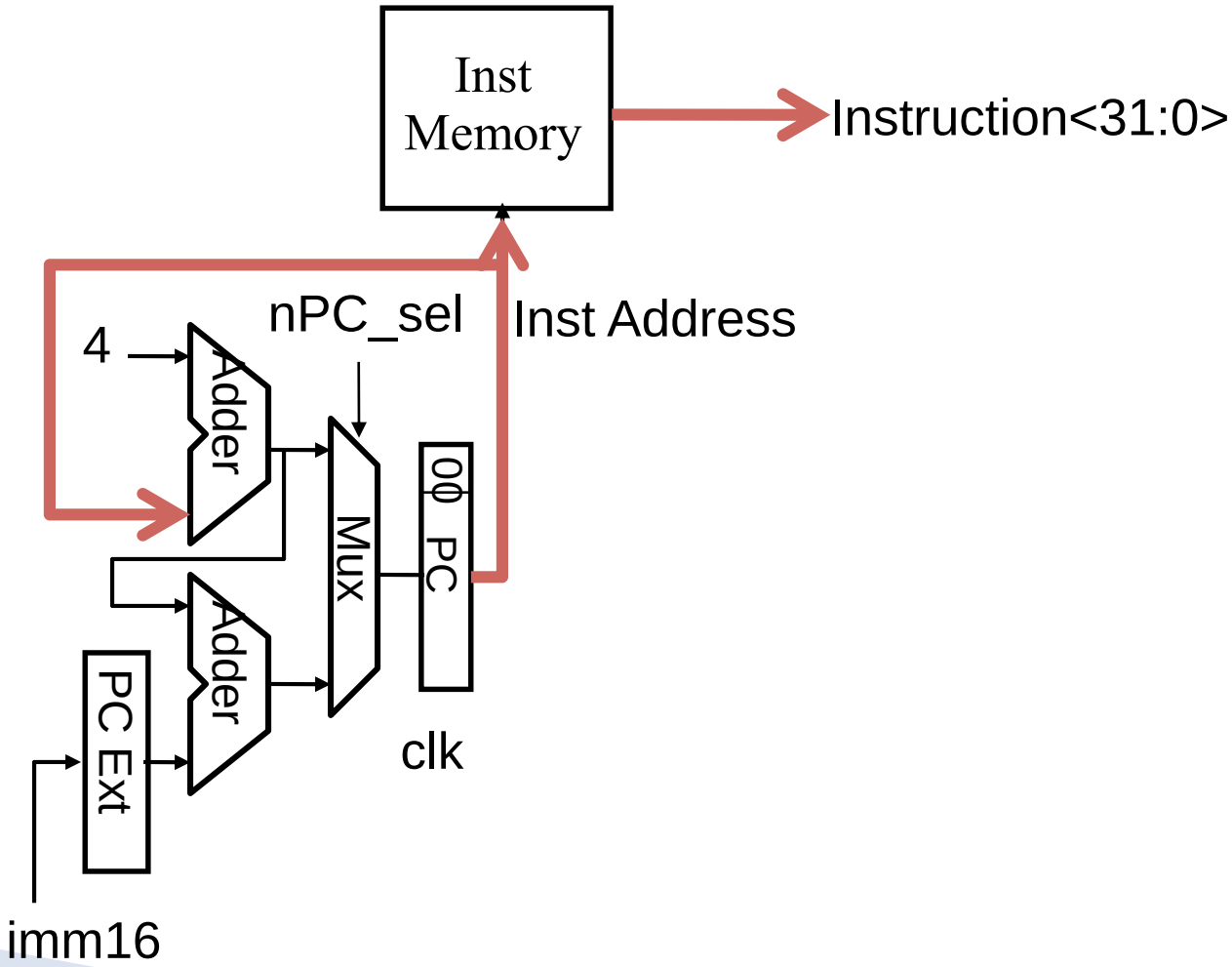

```
add rd, rs, rt
```

- 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

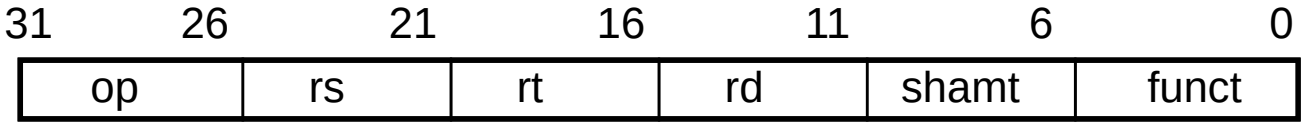
Instruction Fetch Unit start of Add

- Fetch the instruction from Instruction memory:

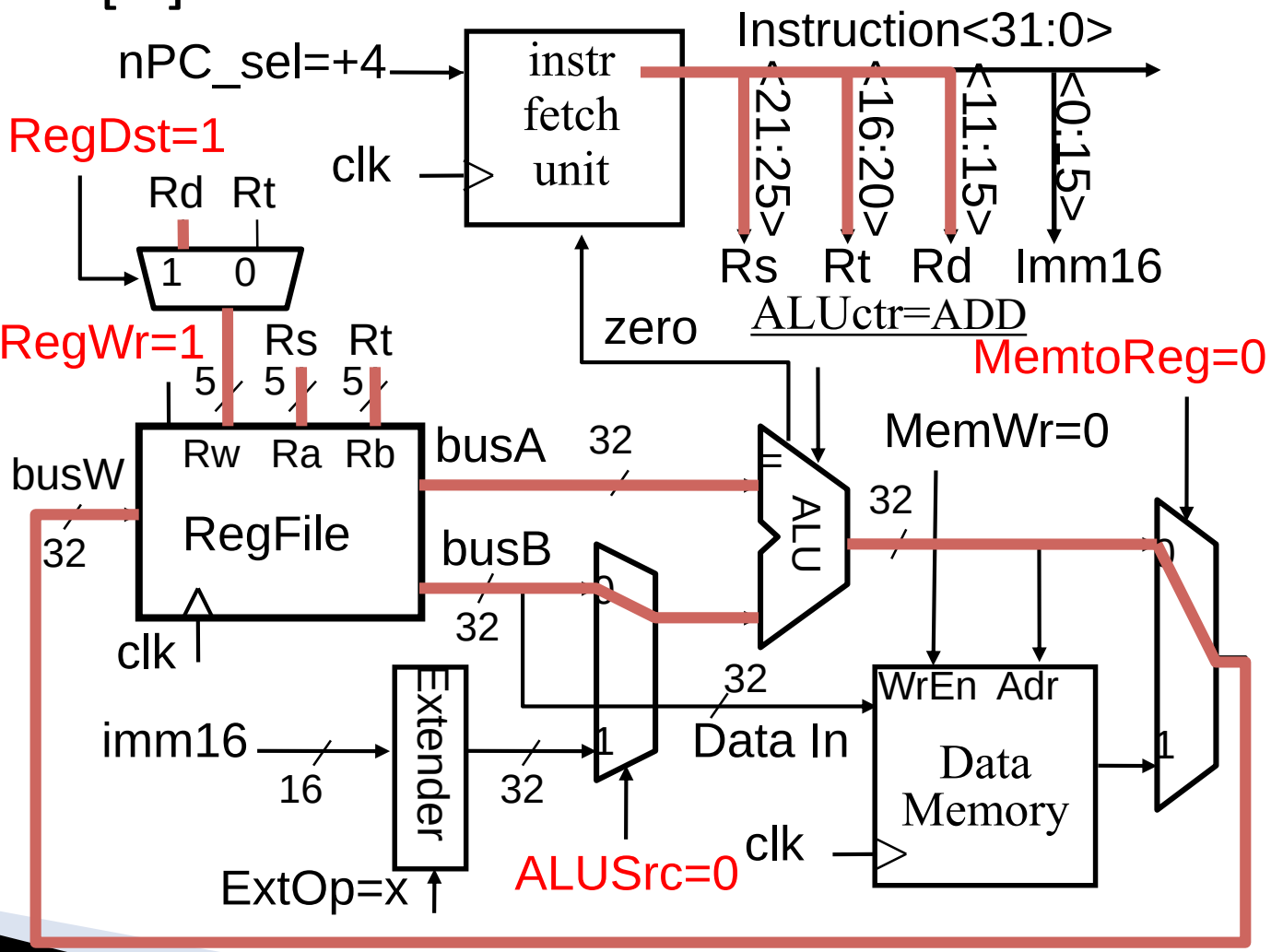
  Instruction = MEM[PC]
  - same for all instructions



The Single Cycle Datapath during Add

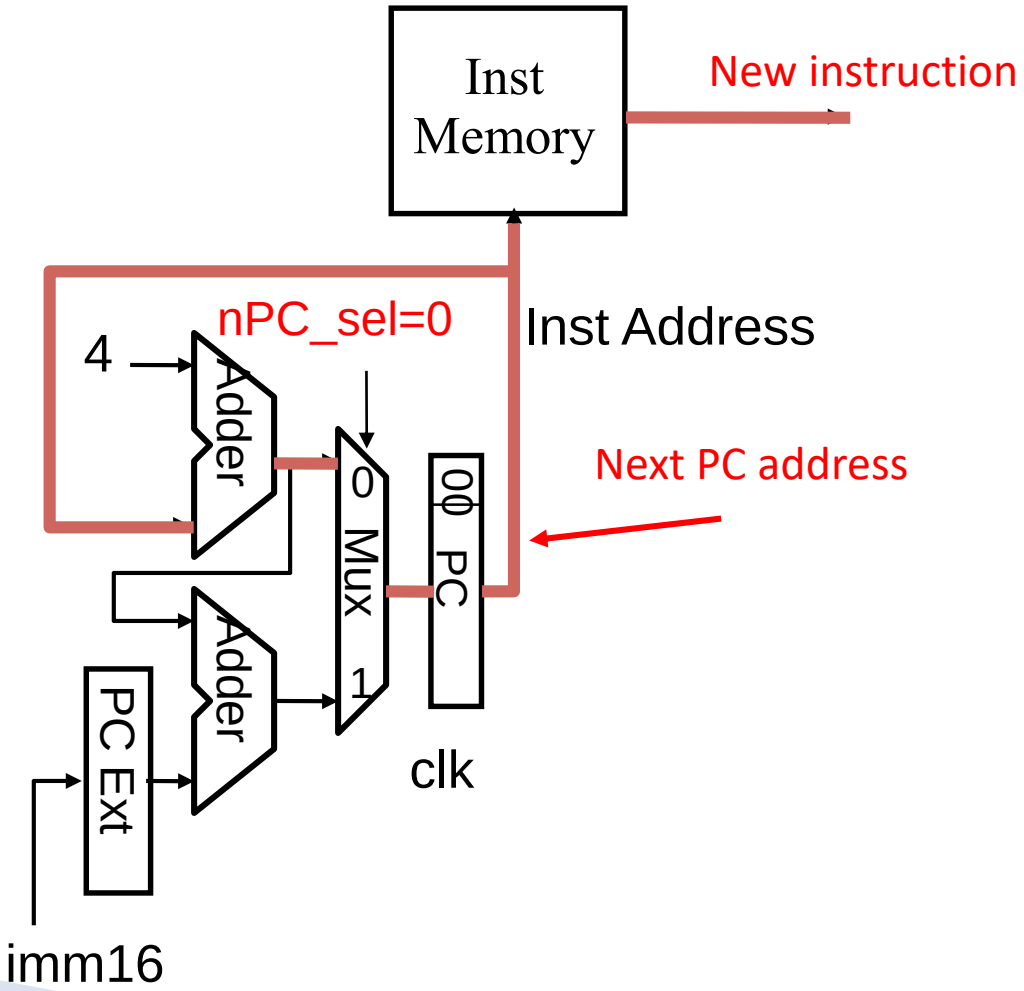


R[rd] = R[rs] + R[rt]

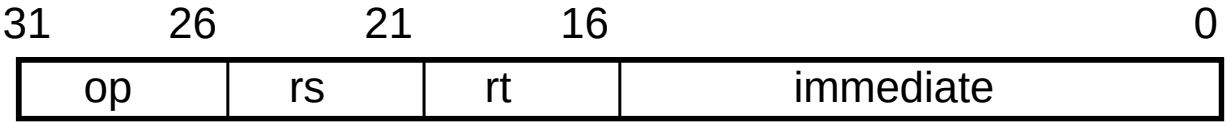


Instruction Fetch Unit end of Add

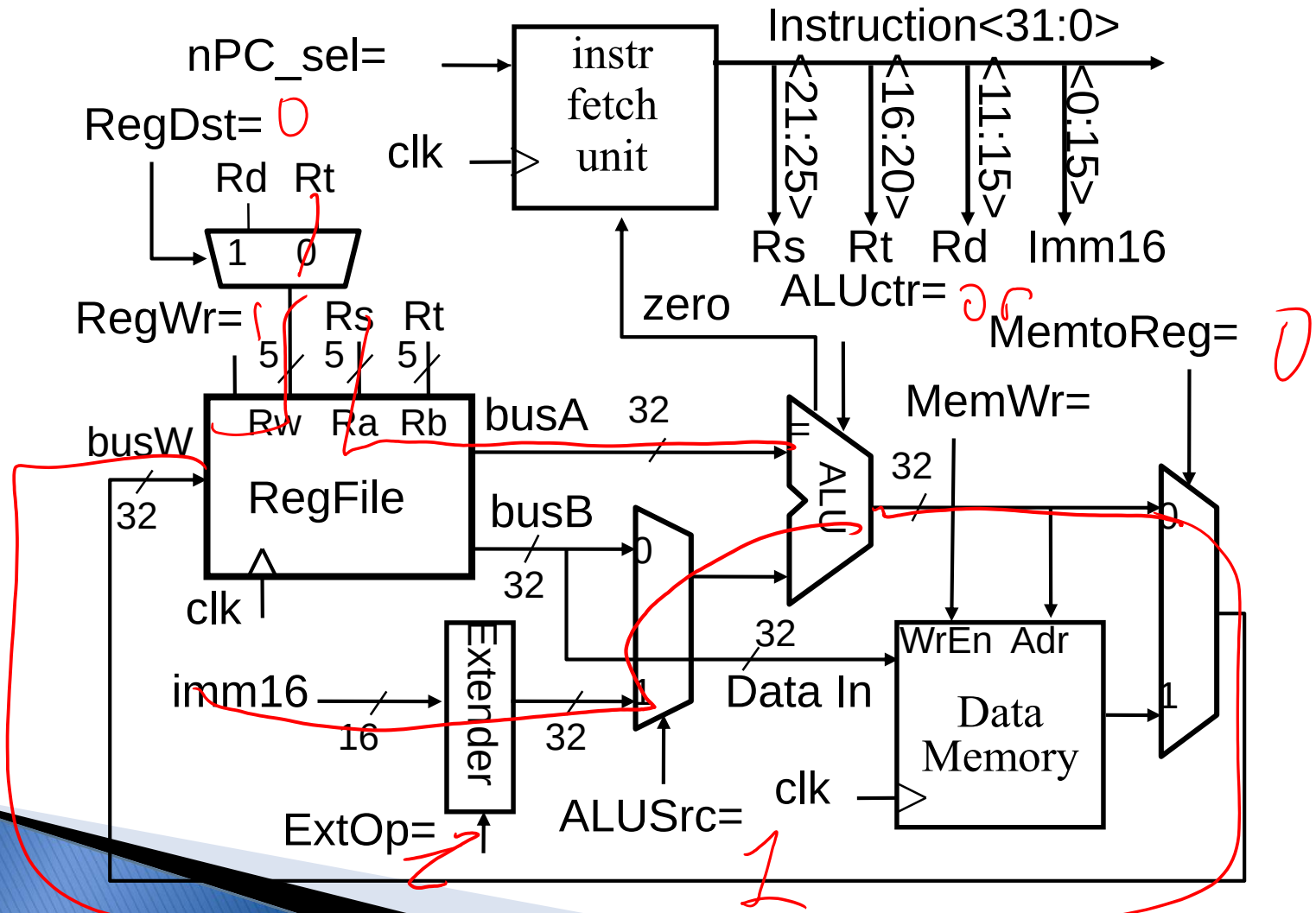
- PC = PC + 4
  - This is the same for all instructions except: Branch and Jump



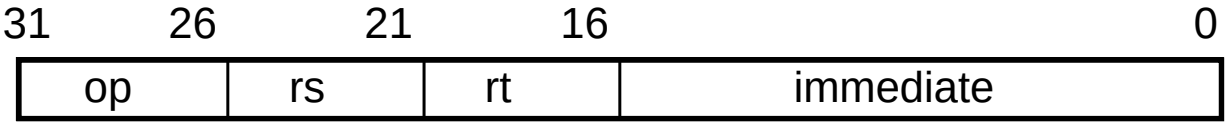
Single Cycle Datapath for Ori



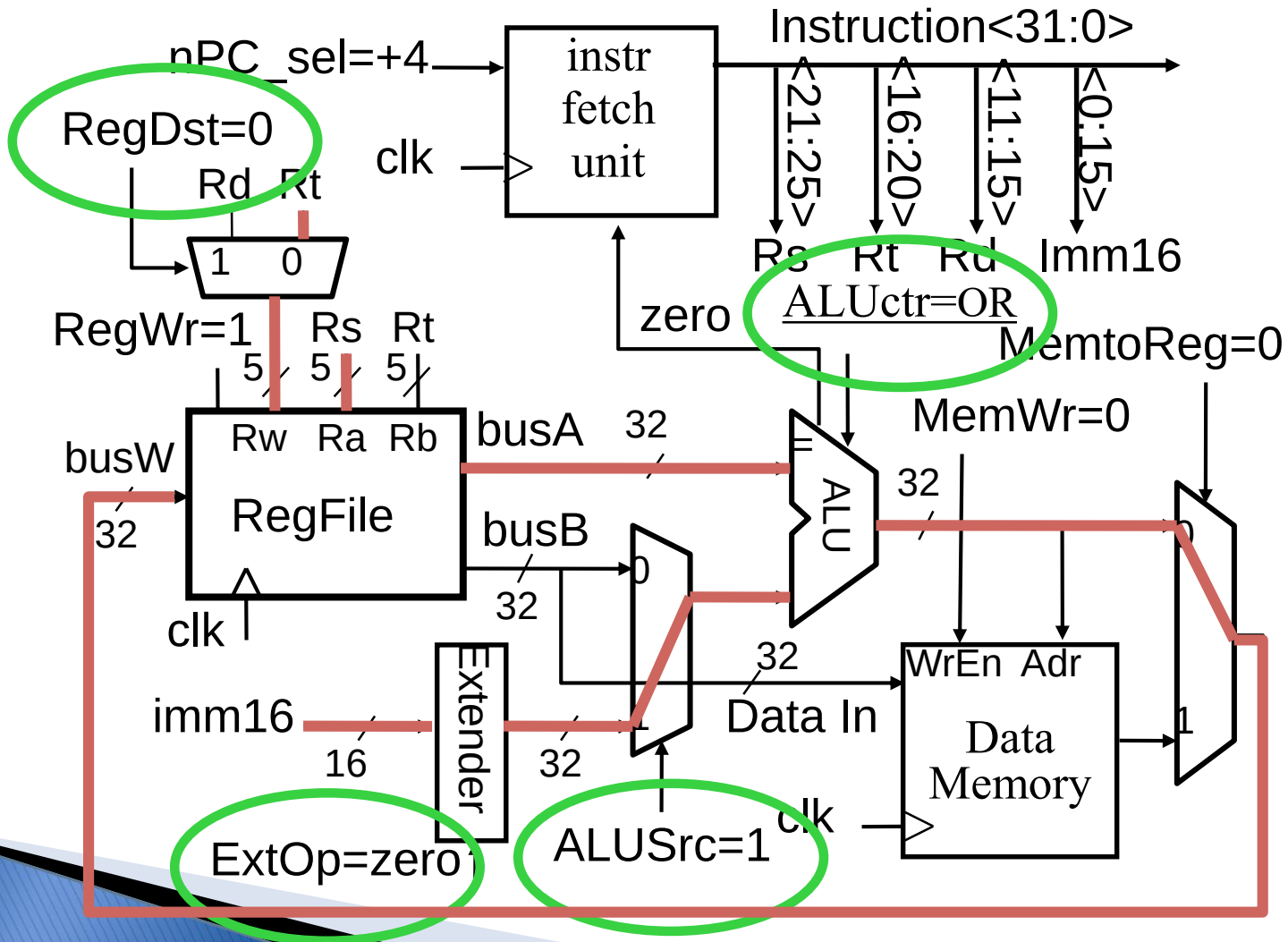
- R[rt] = R[rs] OR ZeroExt[Imm16]



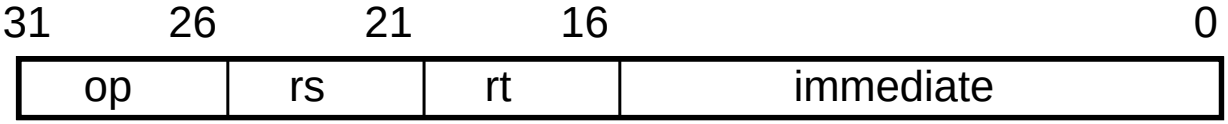
Single Cycle Datapath for Ori



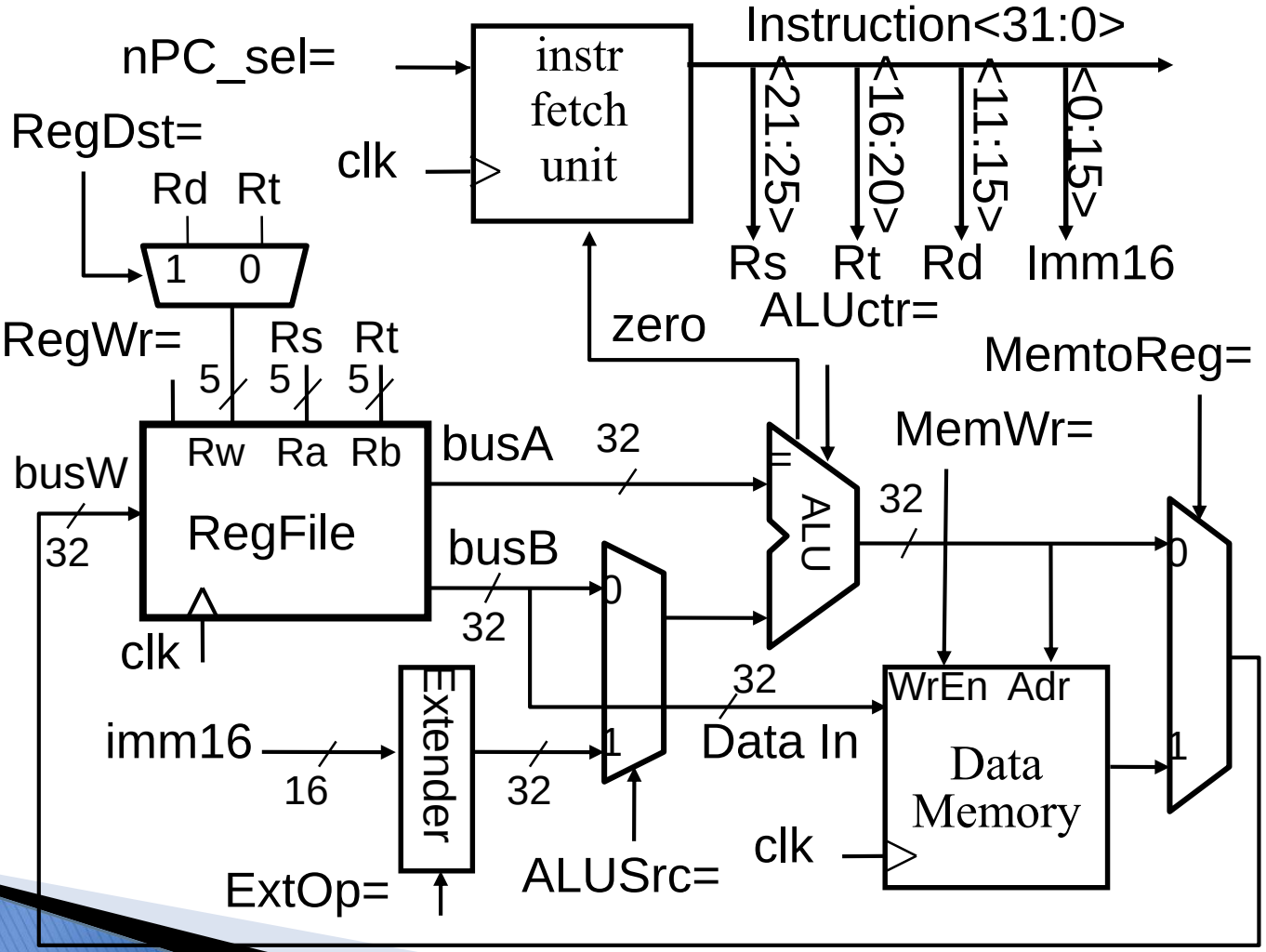
- R[rt] = R[rs] OR ZeroExt[Imm16]



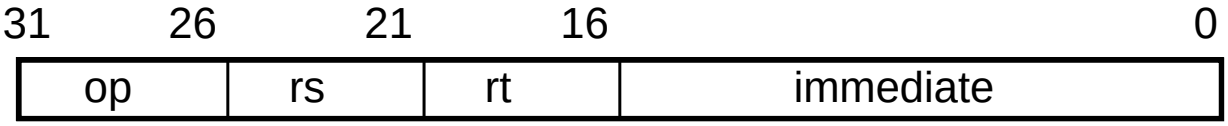
Single Cycle Datapath for LW



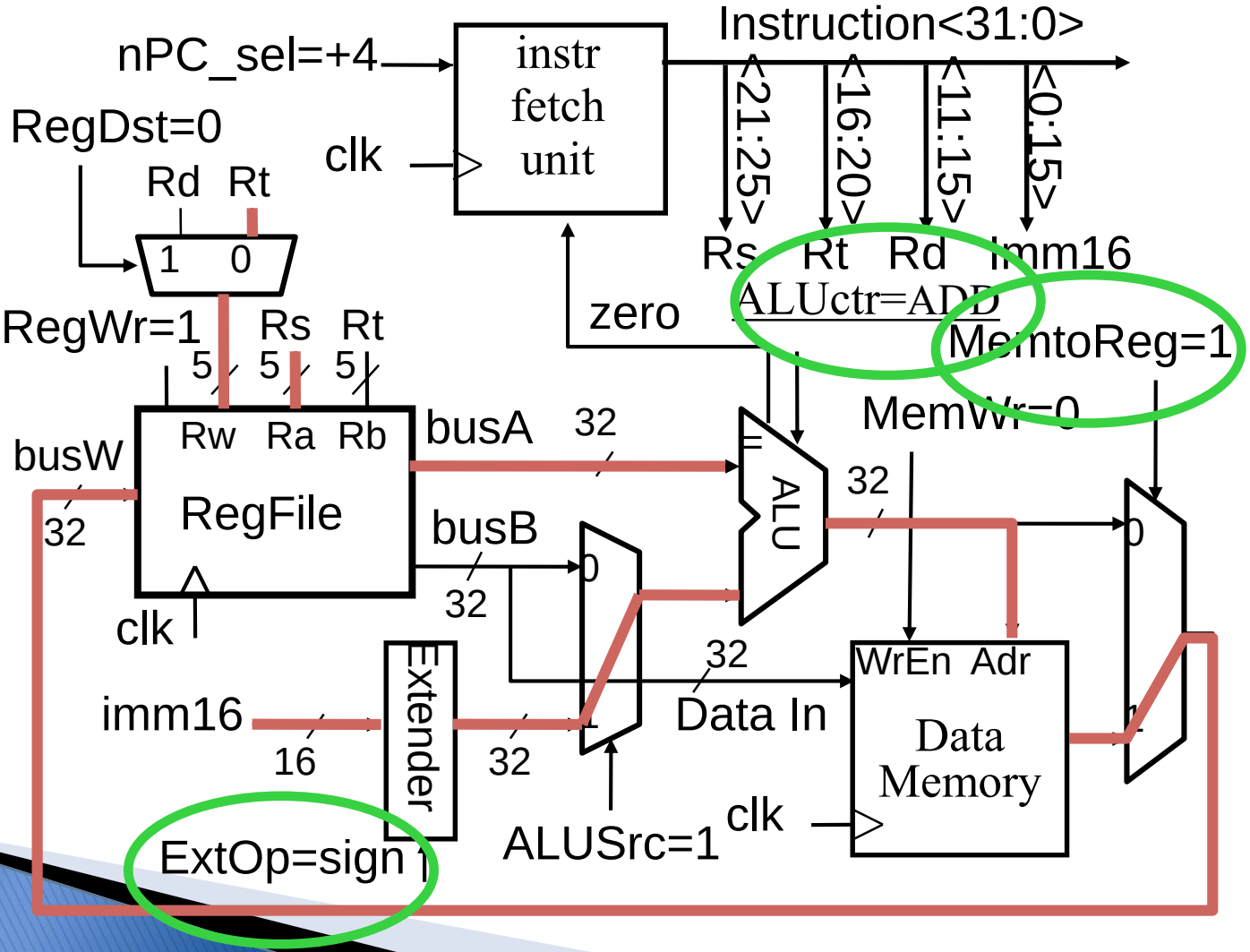
- R[rt] = Data Memory {R[rs] + SignExt[imm16]}



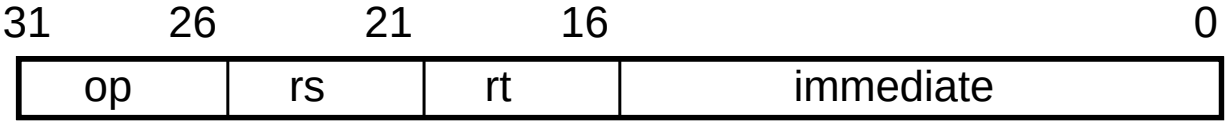
Single Cycle Datapath for LW



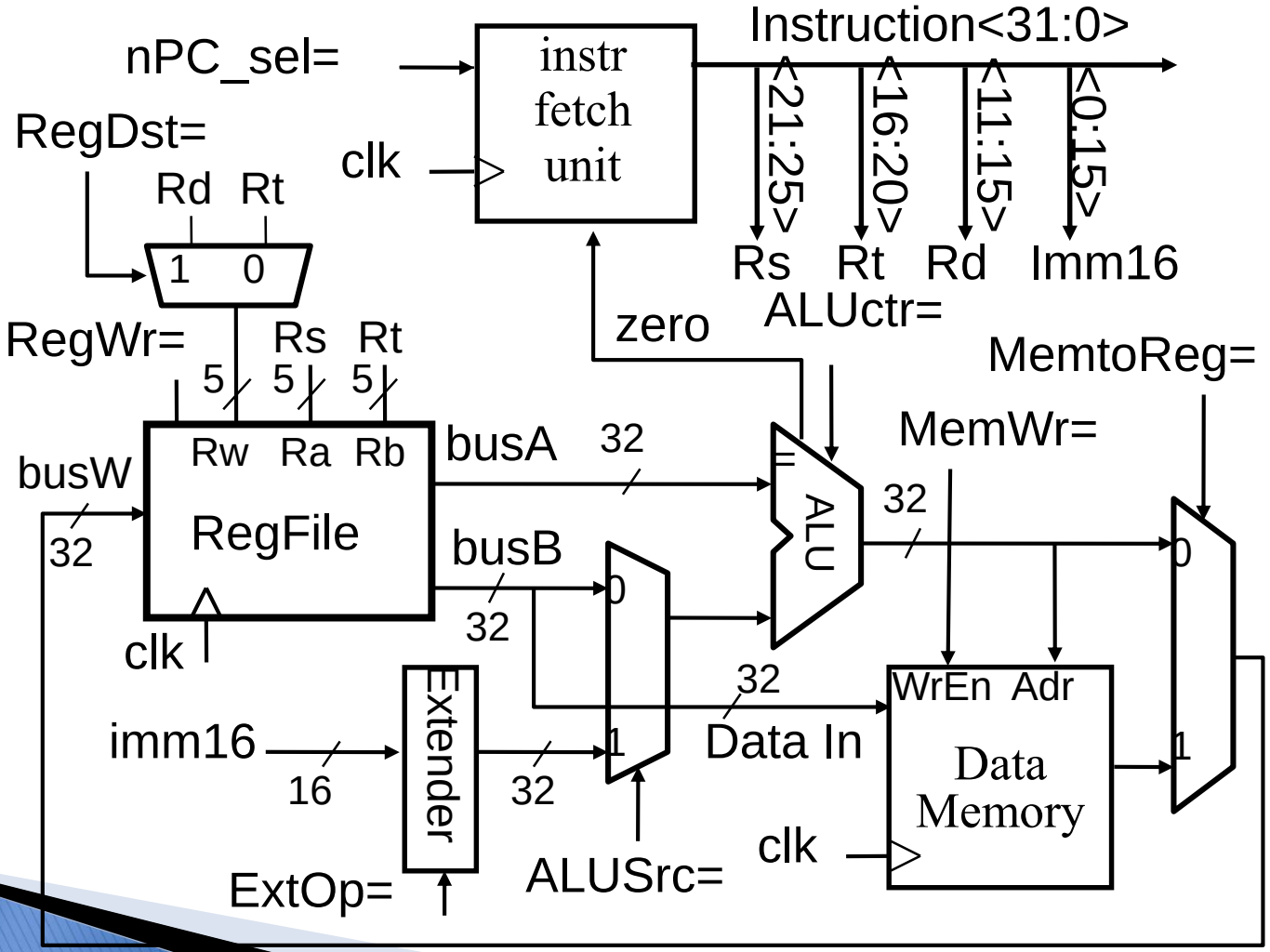
- R[rt] = Data Memory {R[rs] + SignExt[imm16]}



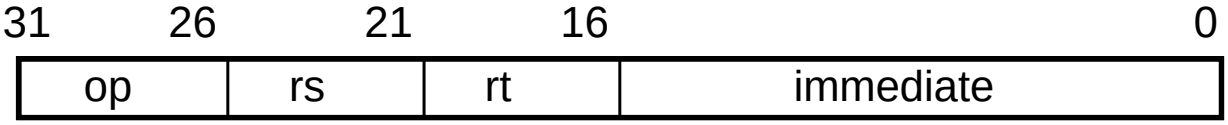
Single Cycle Datapath for SW



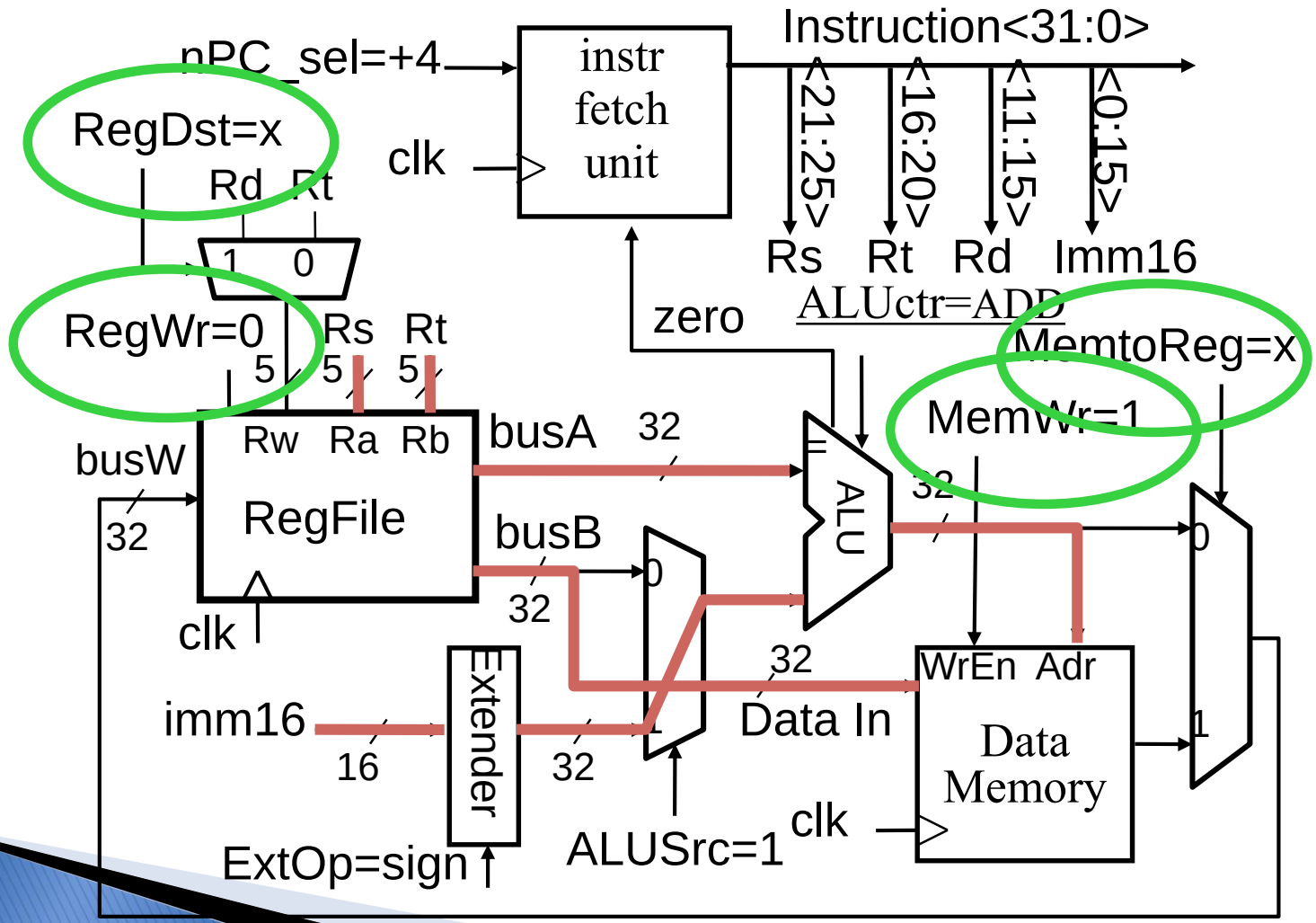
- Data Memory {R[rs] + SignExt[imm16]} = R[rt]



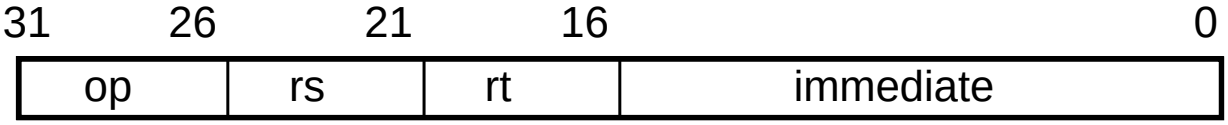
Single Cycle Datapath for SW



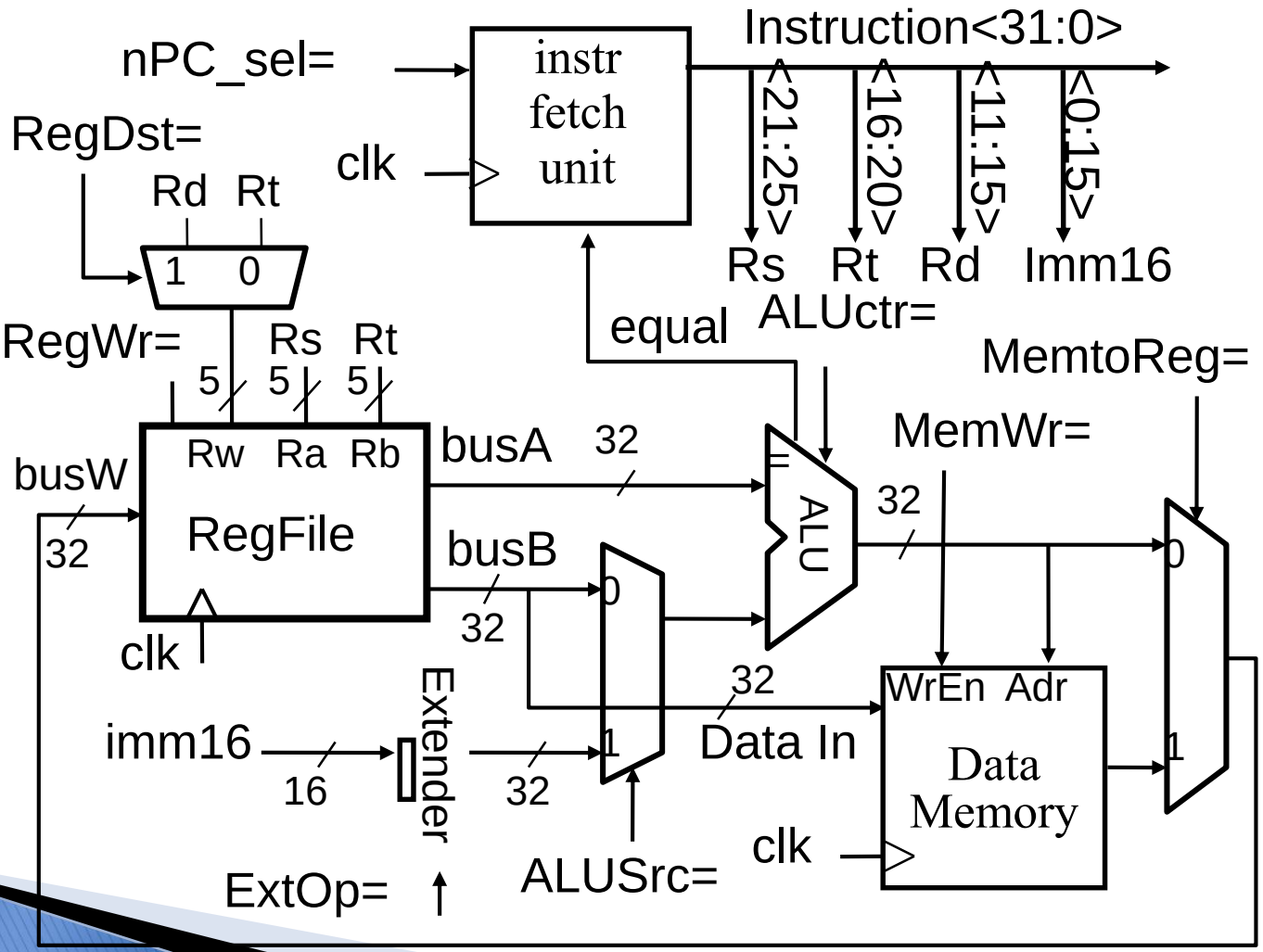
- Data Memory {R[rs] + SignExt[imm16]} = R[rt]



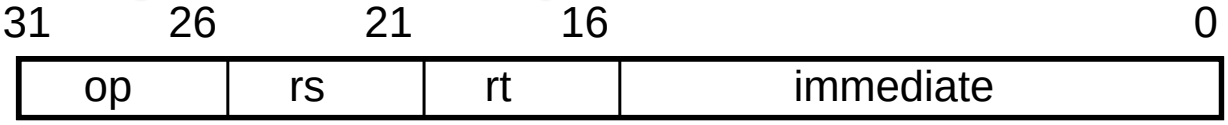
Single Cycle Datapath for Branch



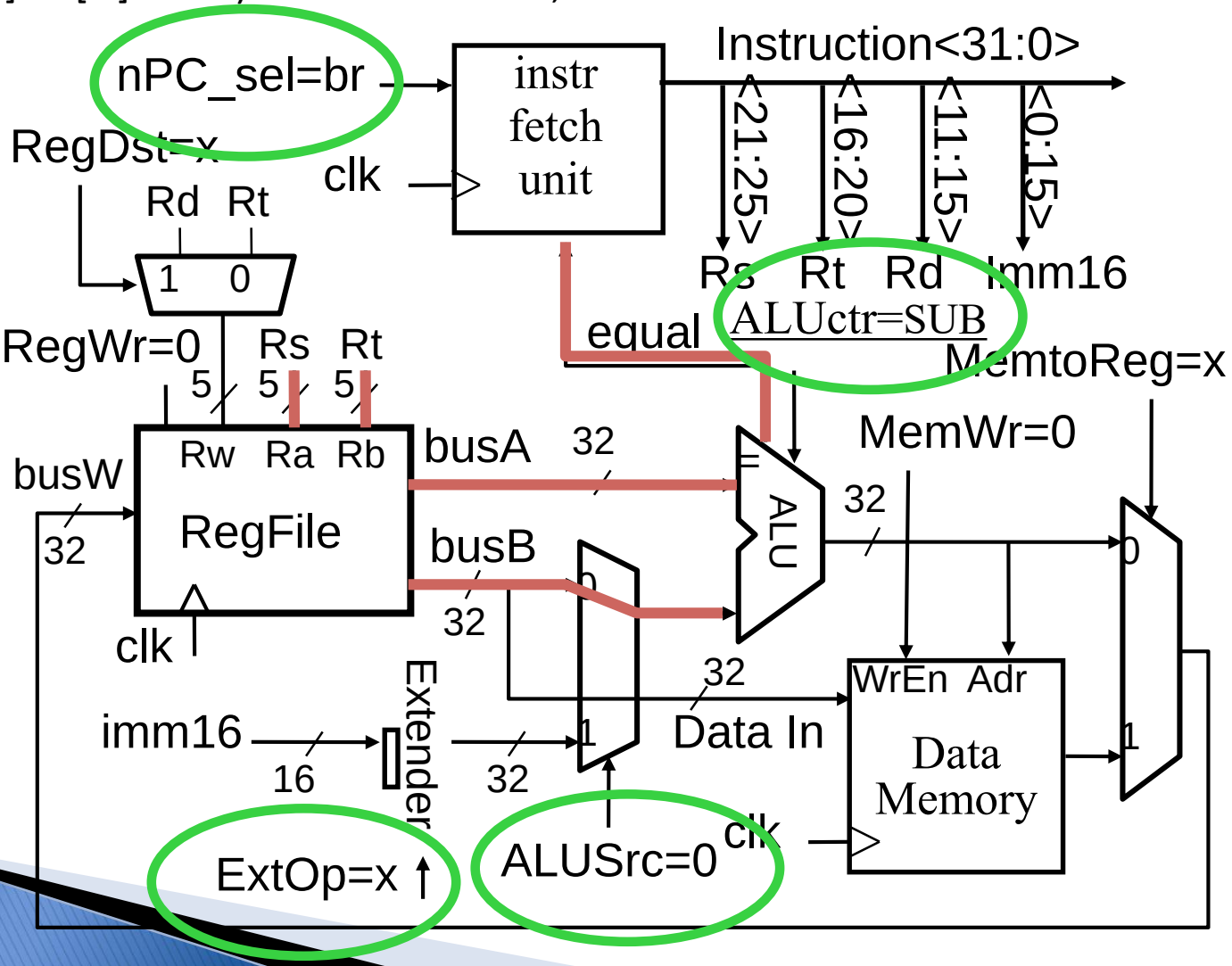
- if (R[rs] - R[rt] == 0) then Zero = 1 ; else Zero = 0



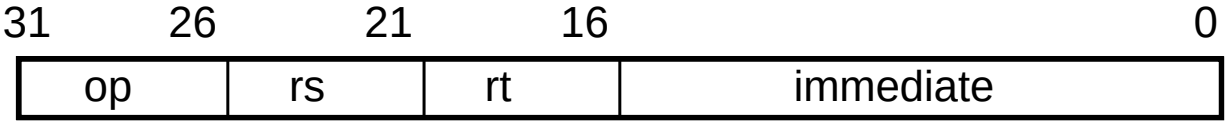
Single Cycle Datapath for Branch



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

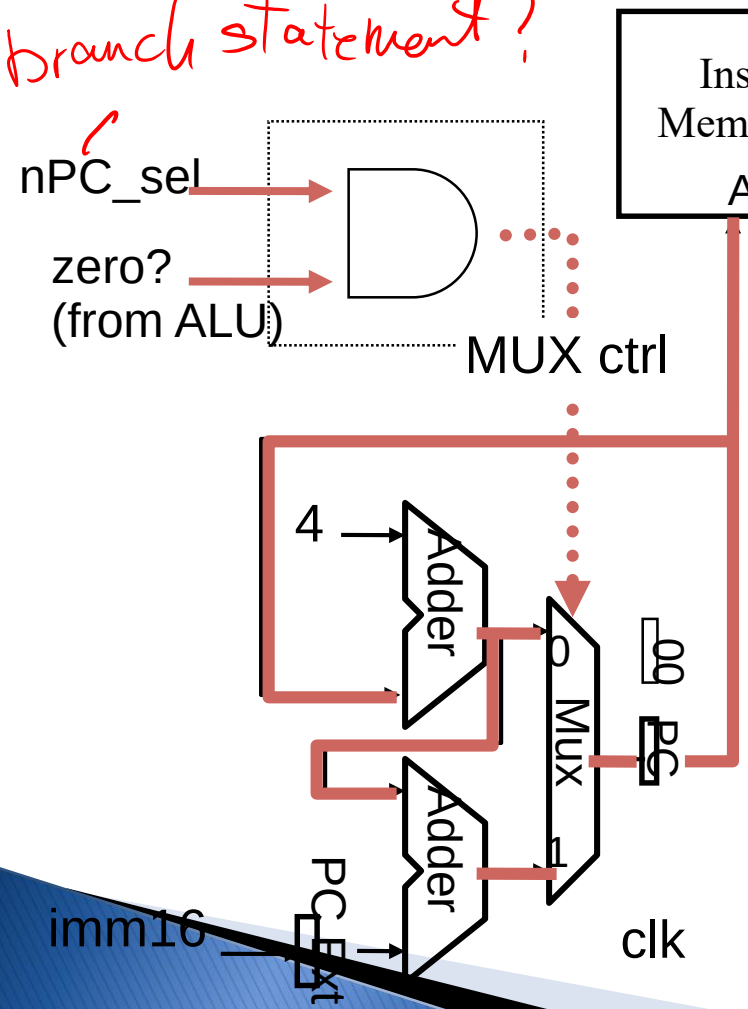


Instruction Fetch Unit end of Branch



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

branch statement?

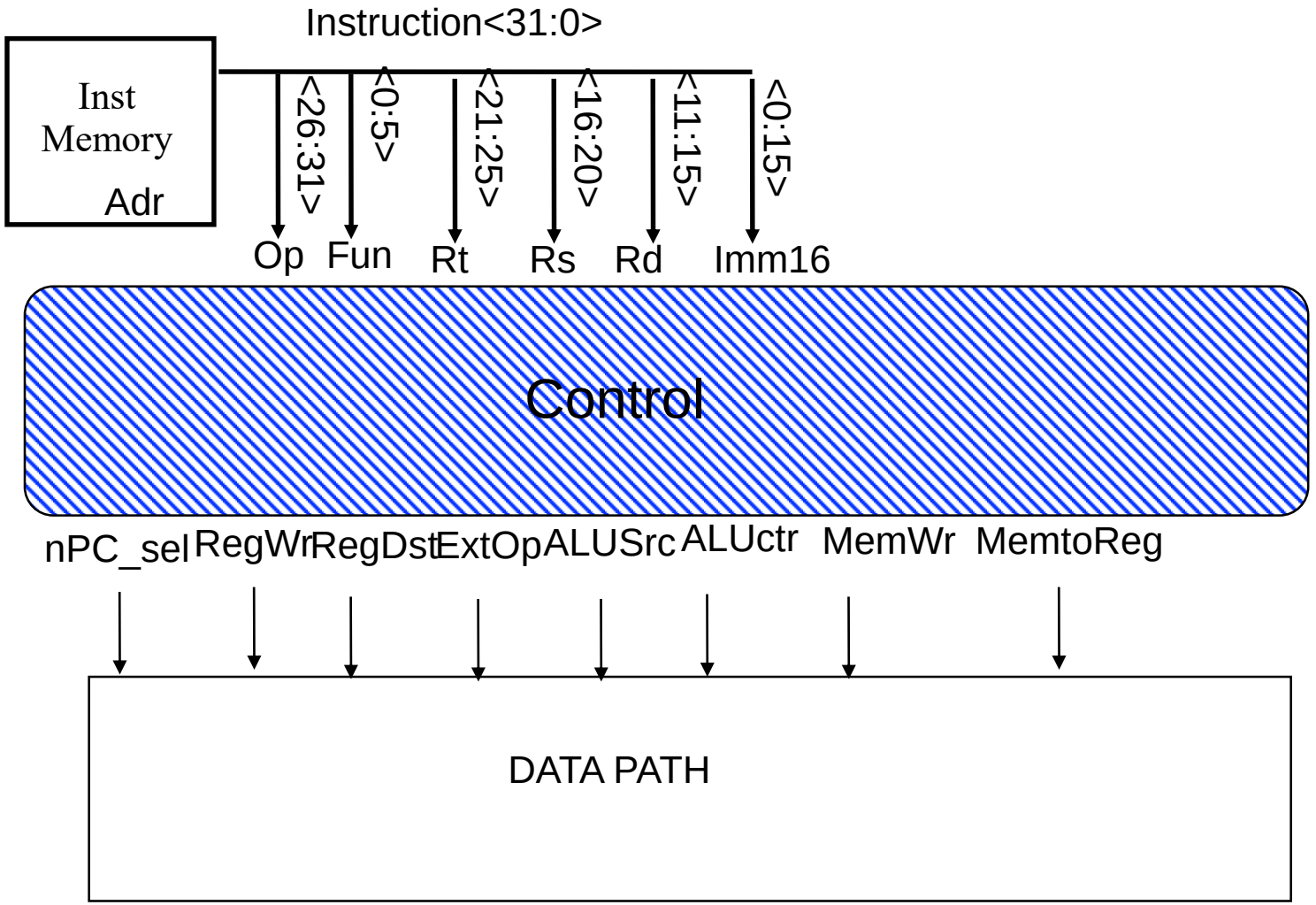


- What is encoding of nPC_sel?
  - **Direct MUX select?**
  - **Branch inst. / not branch**
- Let's pick 2nd option

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

Q: What logic gate?

Control Logic



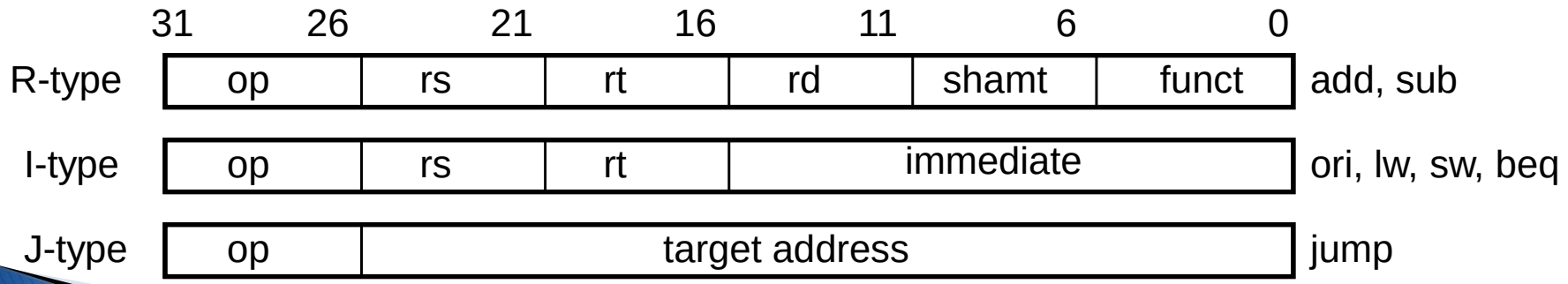
Control Signals (1/2)

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"
lw	R[rt] ← MEM[R[rs] + sign_ext(Imm16)]; PC ← PC + 4
	ALUsrc = Im, Extop = "sn", ALUctr = "ADD", MemtoReg, RegDst = rt, RegWr, nPC_sel = "+4"
sw	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"

Control Signals (2/2)

See Appendix A → func / op

	add	sub	ori	lw	sw	beq	jump
	func: 10 0000	func: 10 0010	We Don't Care :-)				
	op: 00 0000	op: 00 0000	op: 00 1101	op: 10 0011	op: 10 1011	op: 00 0100	op: 00 0010
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	?
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	x



Boolean Expressions for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq (assume ALUctr is 00 ADD, 01: SUB, 10: OR)
ALUctr[1] = or

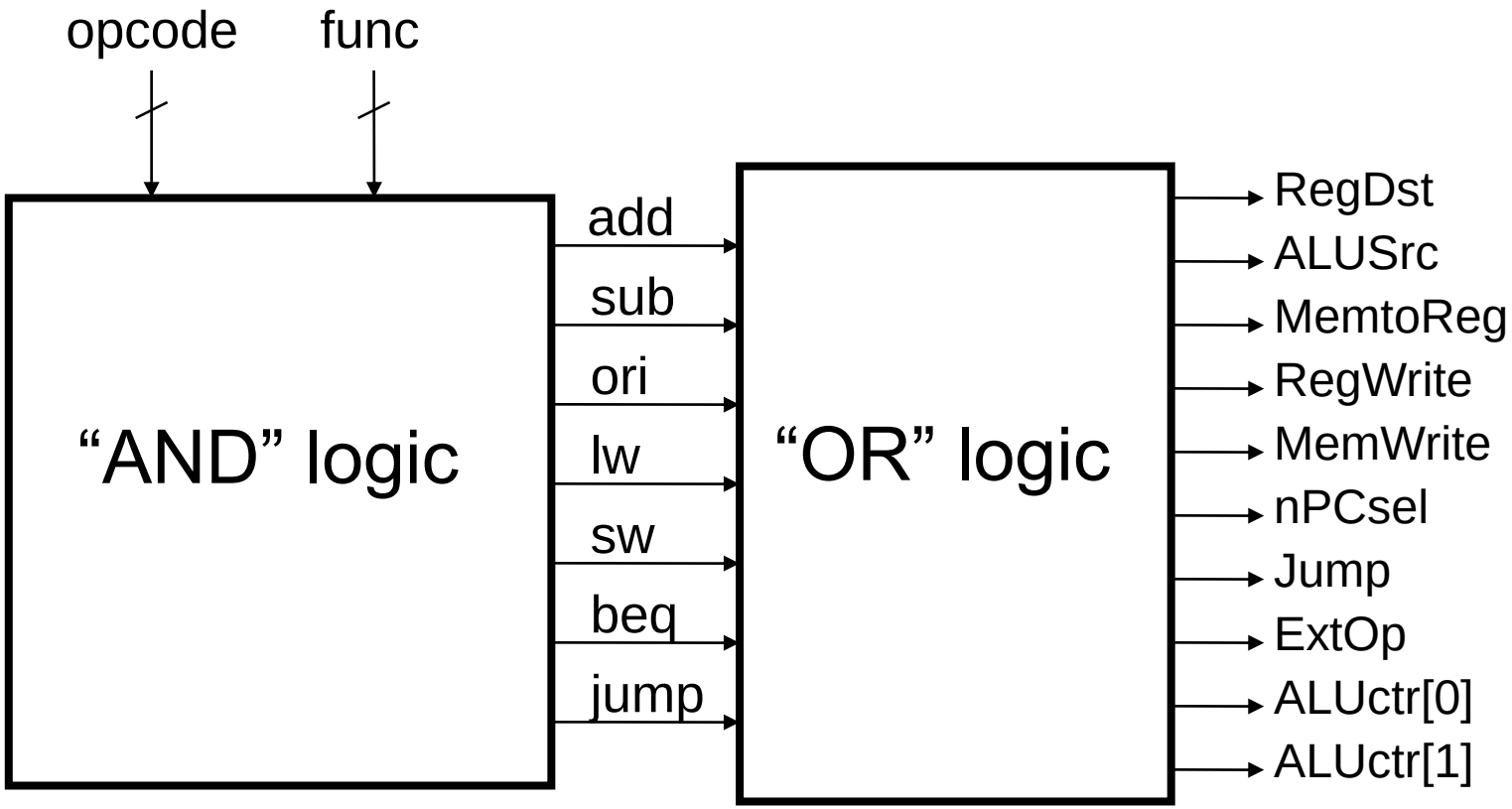
where,

rtype = $\sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet \sim op_1 \bullet \sim op_0$,
ori = $\sim op_5 \bullet \sim op_4 \bullet op_3 \bullet op_2 \bullet \sim op_1 \bullet op_0$
lw = $op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0$
sw = $op_5 \bullet \sim op_4 \bullet op_3 \bullet \sim op_2 \bullet op_1 \bullet op_0$
beq = $\sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet op_2 \bullet \sim op_1 \bullet \sim op_0$
jump = $\sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet op_1 \bullet \sim op_0$

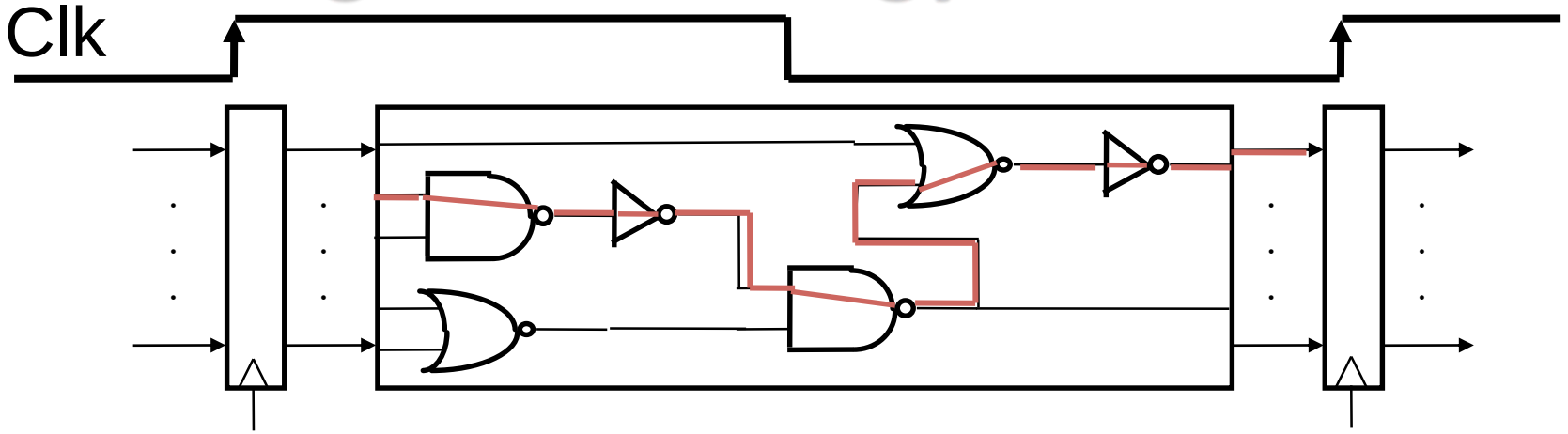
How do we implement this in gates?

add = rtype $\bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet \sim func_1 \bullet \sim func_0$
sub = rtype $\bullet func_5 \bullet \sim func_4 \bullet \sim func_3 \bullet \sim func_2 \bullet func_1 \bullet \sim func_0$

Controller Implementation



Clocking Methodology

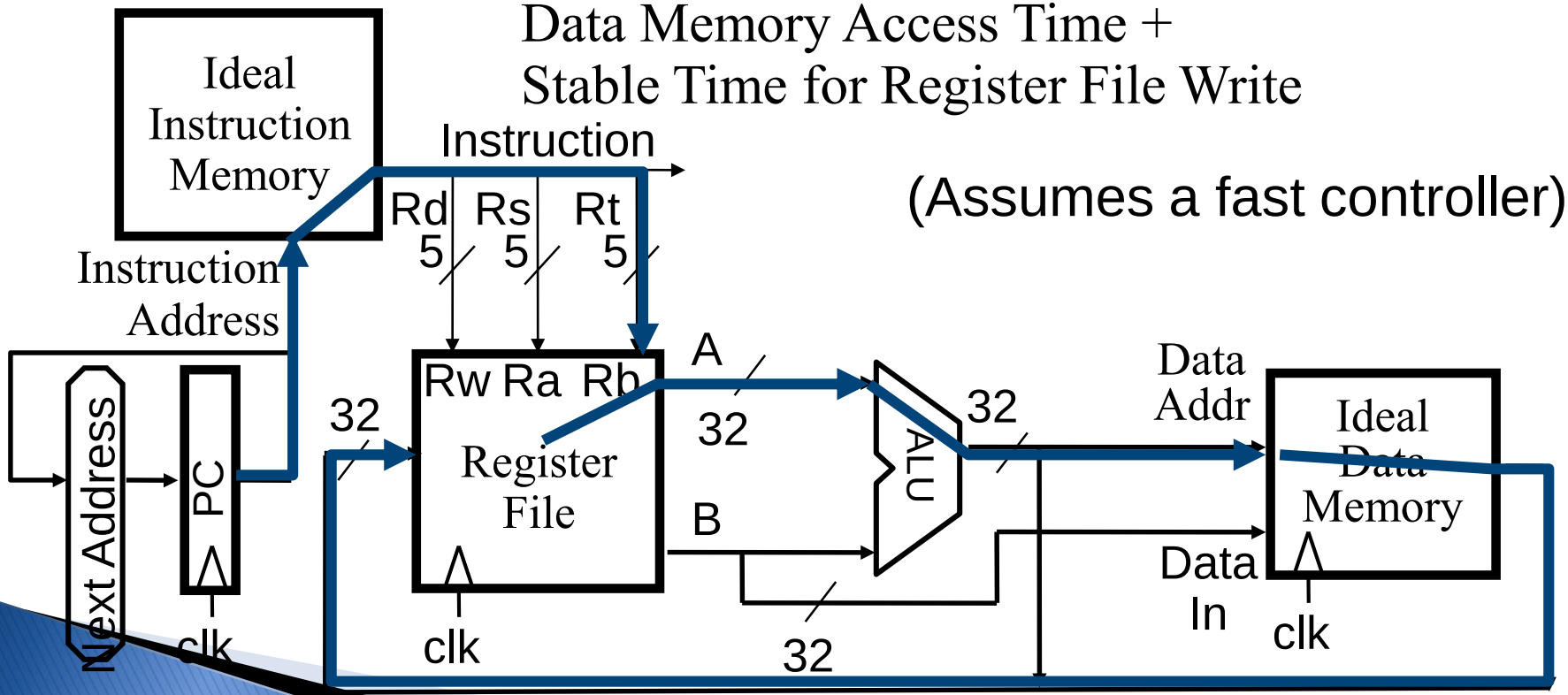


- Storage elements clocked by same edge
- Being physical devices, flip-flops (FF) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF (set-up time), and we have the usual clock-to-Q delay
- "Critical path" (longest path through logic) determines length of clock period

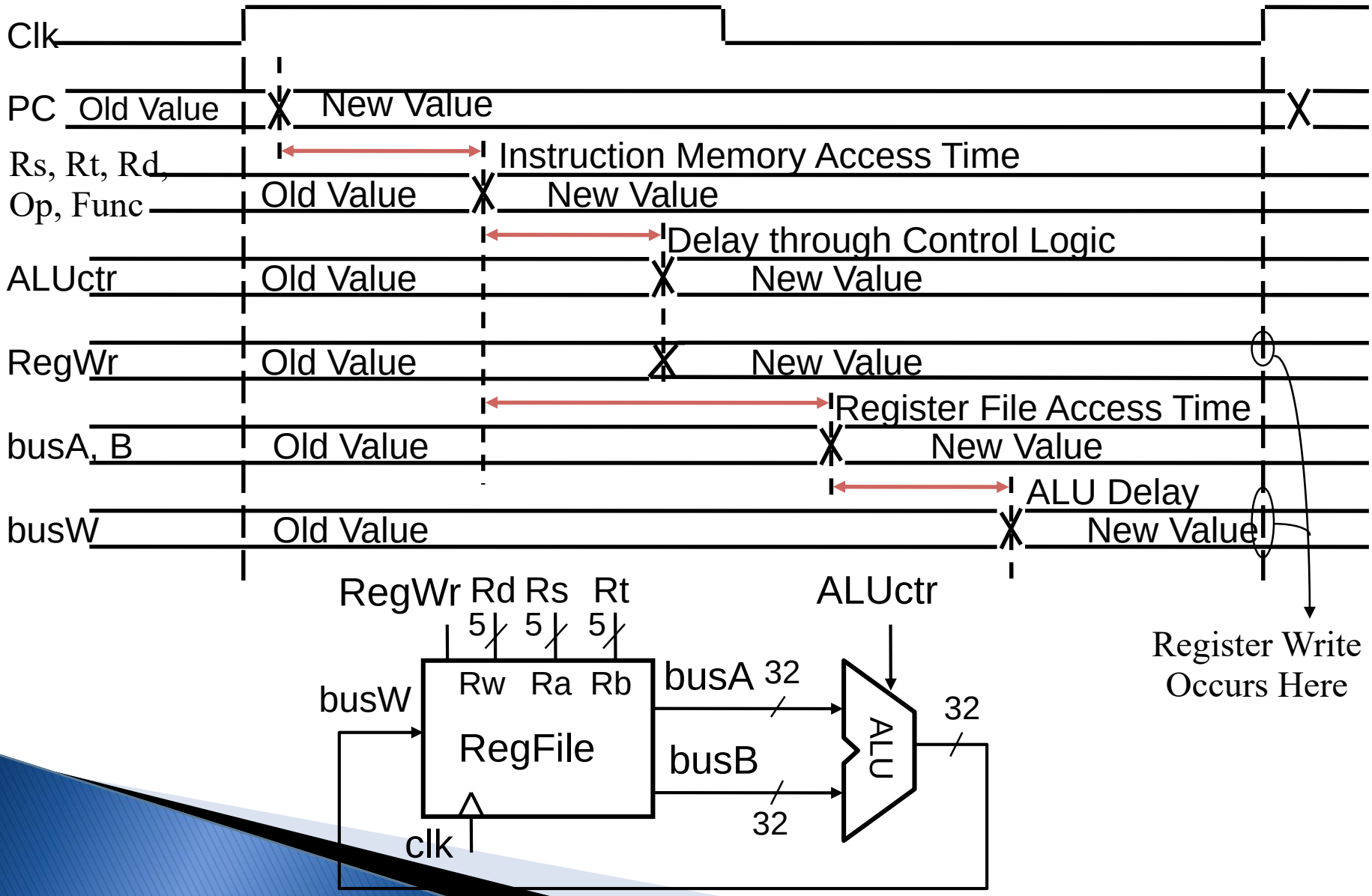
An Abstract View of the Critical Path

Critical Path (Load Instruction) =
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

(Assumes a fast controller)



Register-Register Timing: Cycle



Summary: Single-cycle Processor

- 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**
    - **Formulate Logic Equations**
    - **Design Circuits**