

CS 110
Computer Architecture
Single-Cycle CPU
Datapath & Control

Instructor:
Sören Schwertfeger

<http://shtech.org/courses/ca/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Processor Design: 5 steps

Step 1: Analyze instruction set to determine datapath requirements

- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

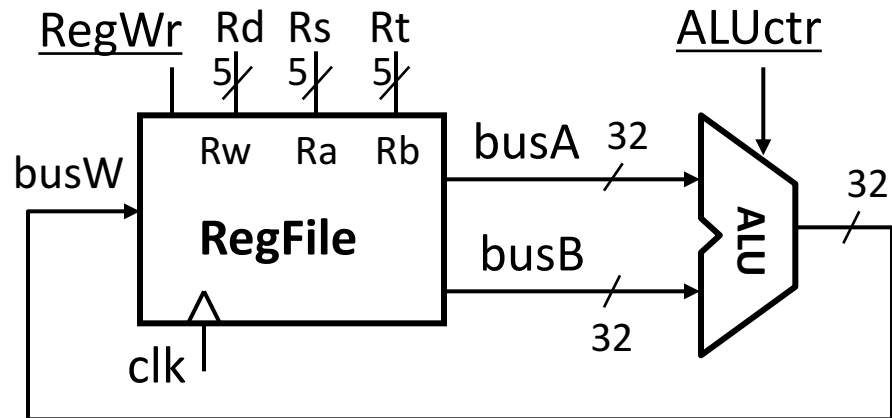
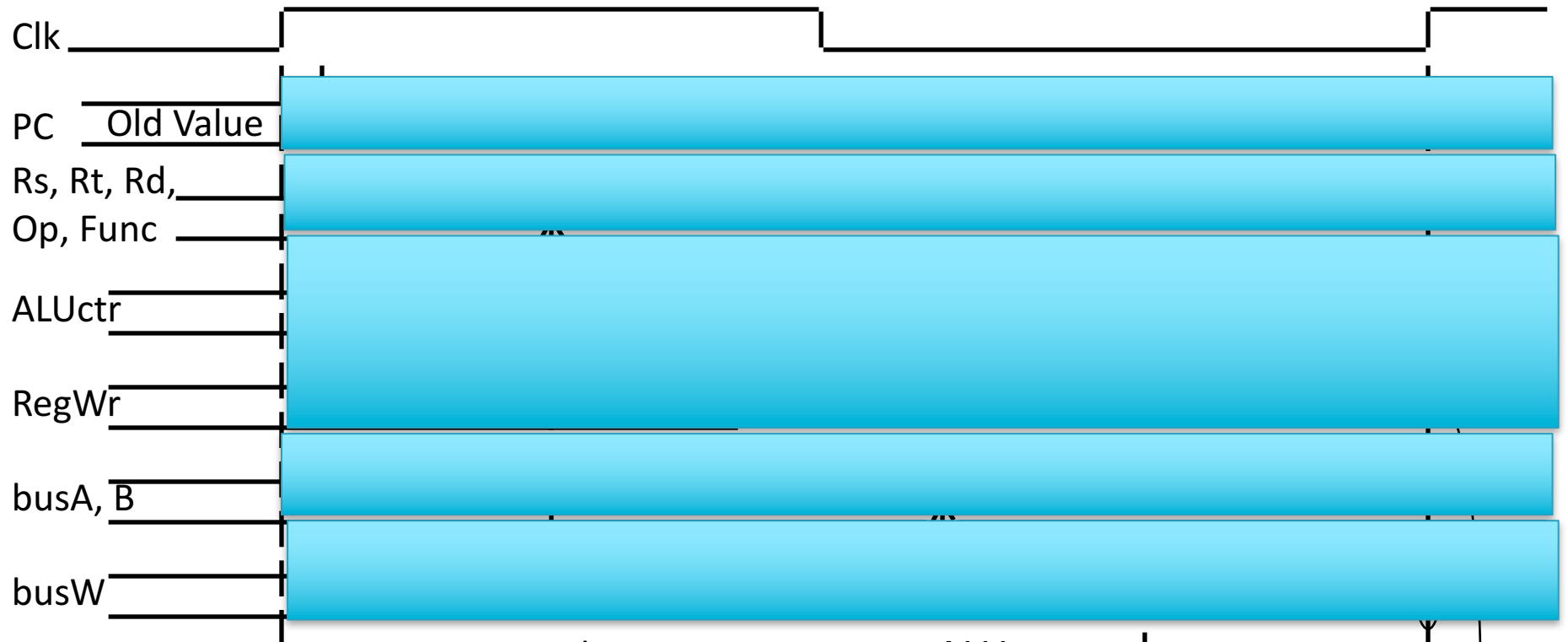
Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

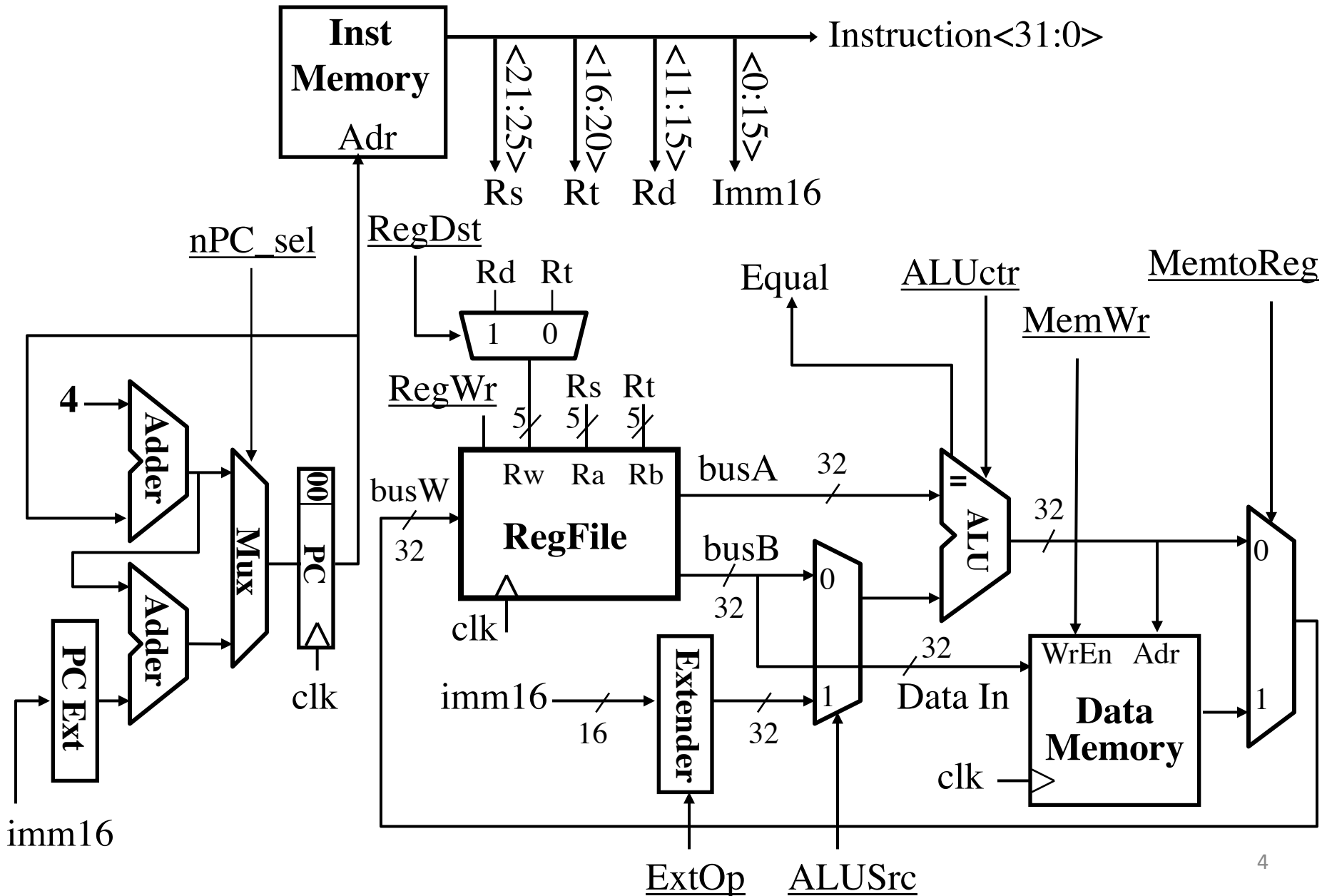
Step 5: Assemble the control logic

Register-Register Timing: One Complete Cycle (Add/Sub)



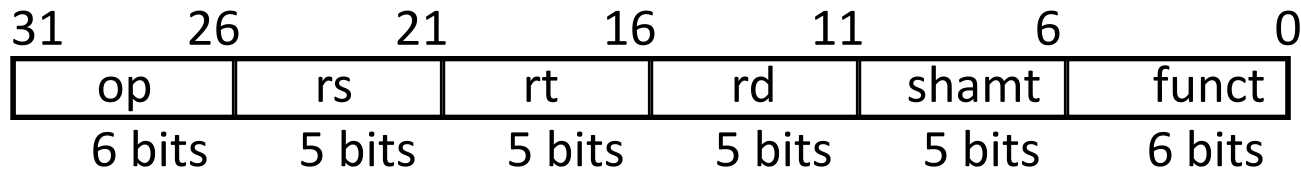
Register Write
Occurs Here

A Single Cycle Datapath

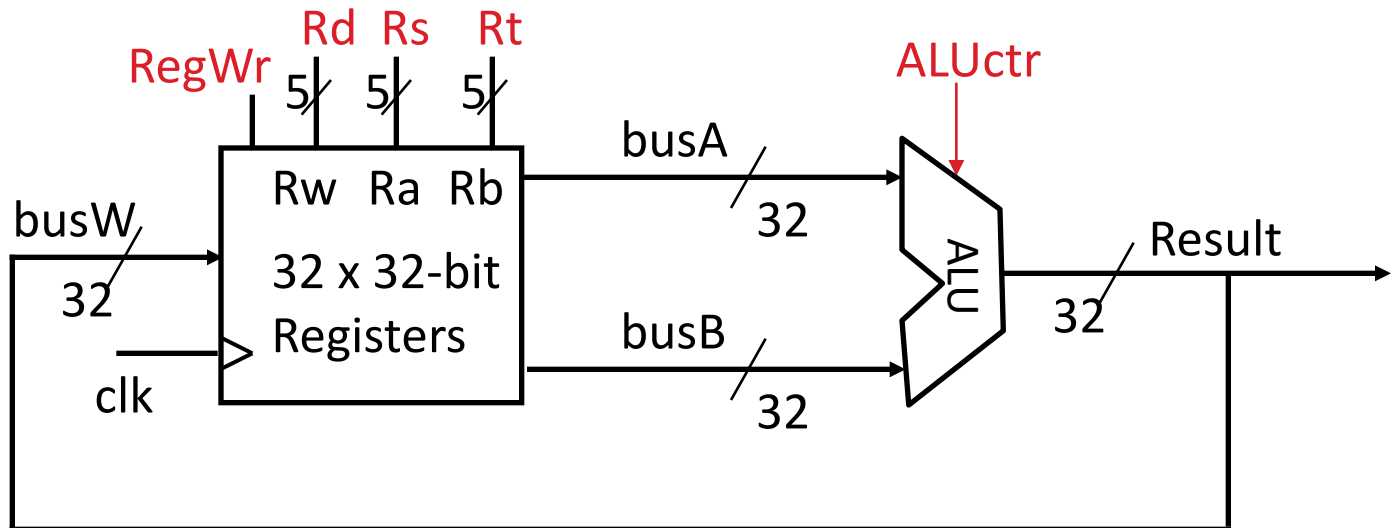


Step 3b: Add & Subtract

- $R[rd] = R[rs] \text{ op } R[rt]$ (`addu rd,rs,rt`)
 - Ra, Rb, and Rw come from instruction's Rs, Rt, and Rd fields



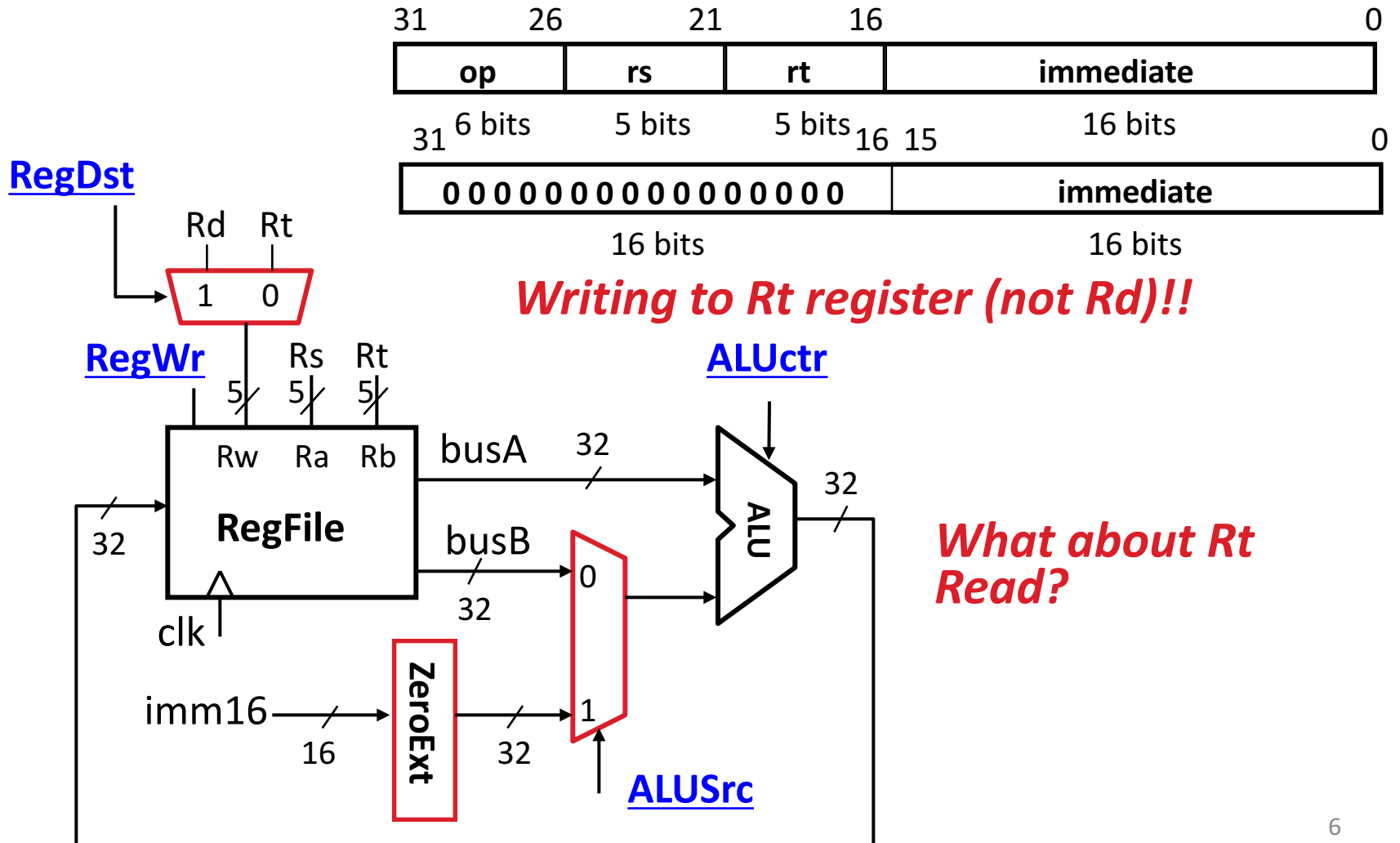
- **ALUctr** and **RegWr**: control logic after decoding the instruction



- ... Already defined the register file & ALU

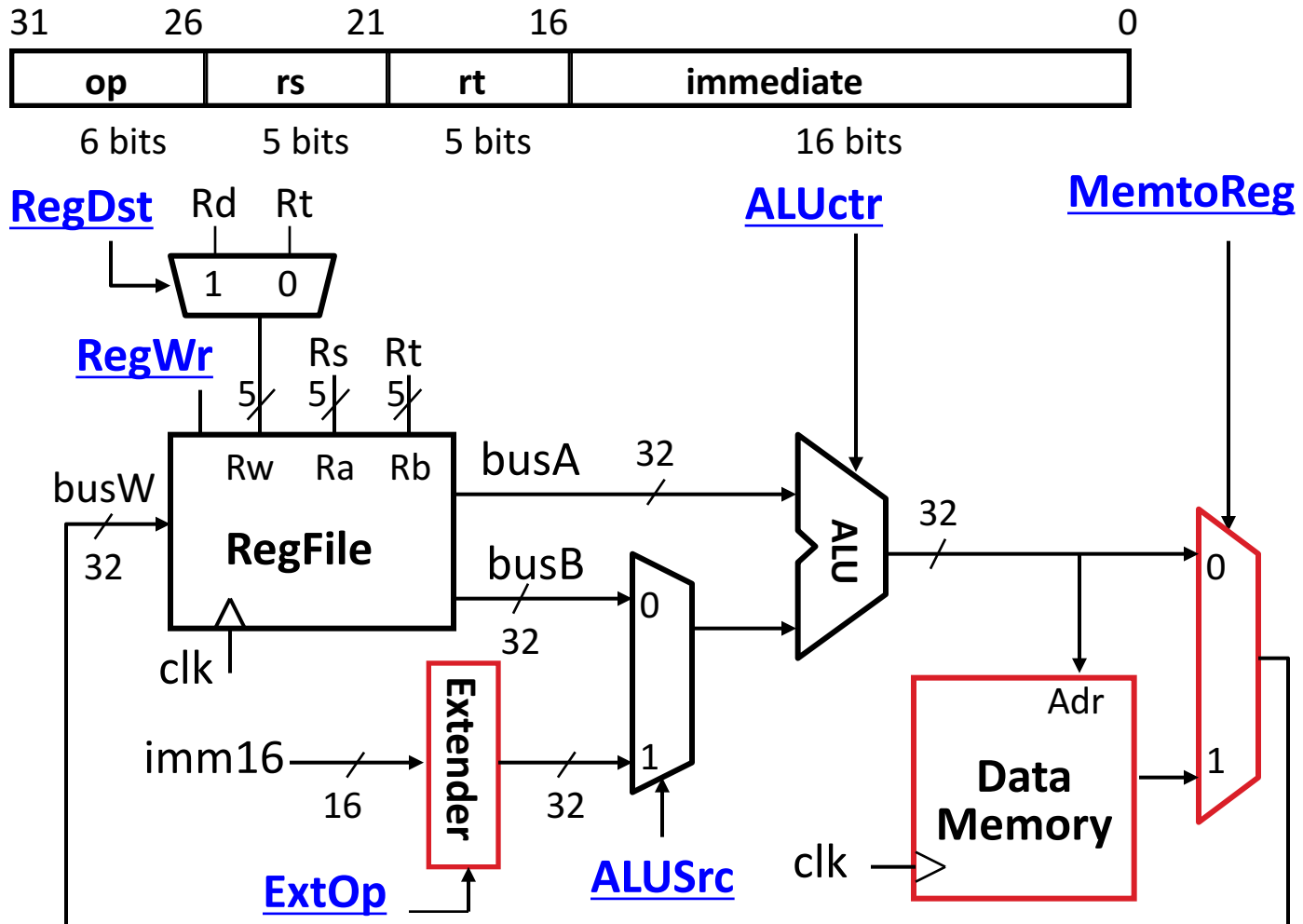
3c: Logical Op (or) with Immediate

- $R[rt] = R[rs] \text{ op ZeroExt}[imm16]$



3d: Load Operations

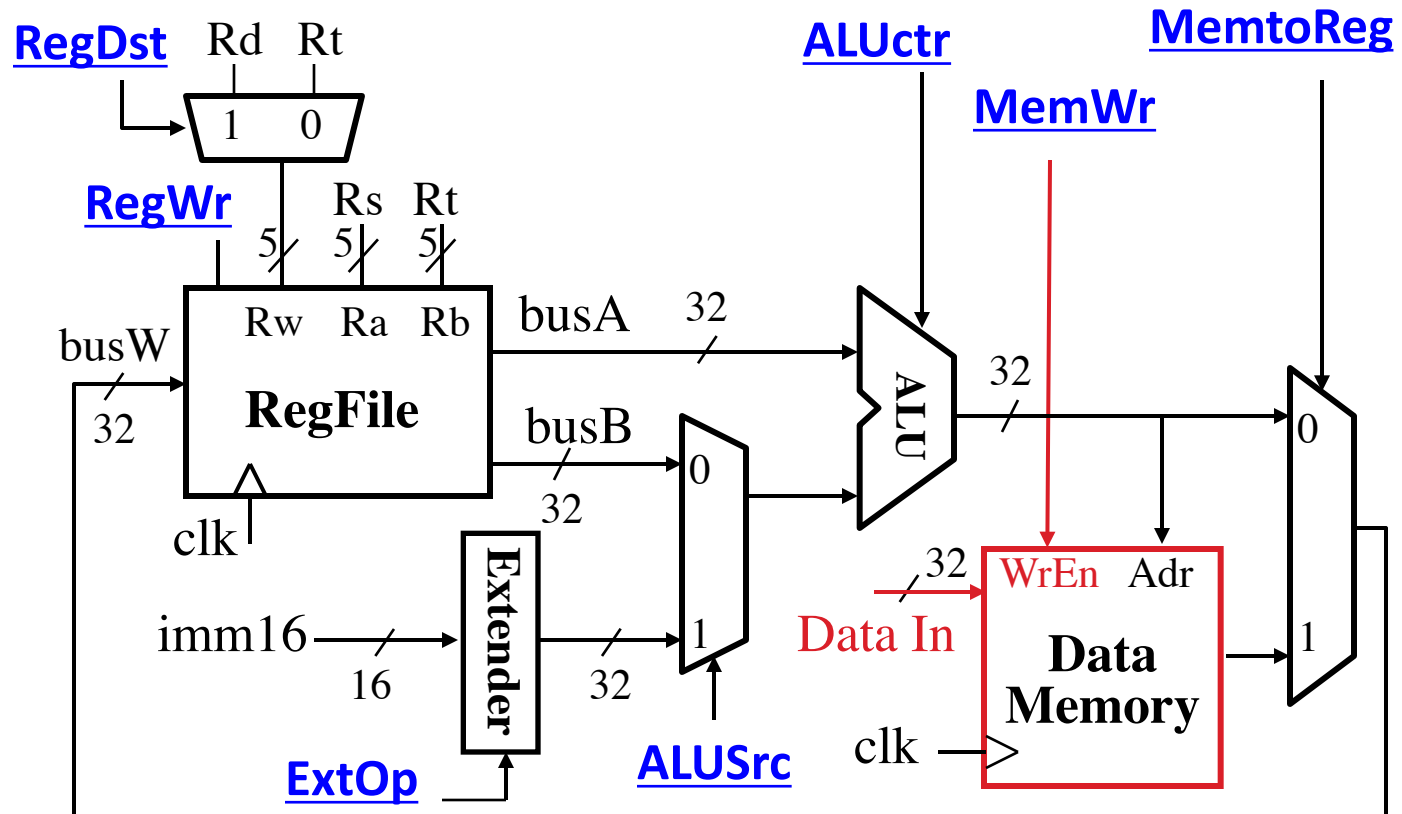
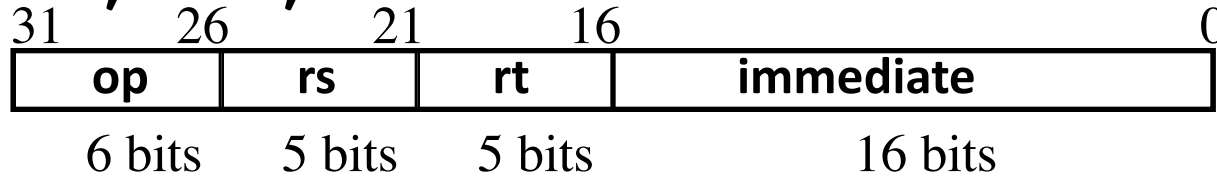
- $R[rt] = Mem[R[rs] + SignExt[imm16]]$
Example: `lw rt, rs, imm16`



3e: Store Operations

- $\text{Mem}[R[\text{rs}] + \text{SignExt}[\text{imm16}]] = R[\text{rt}]$

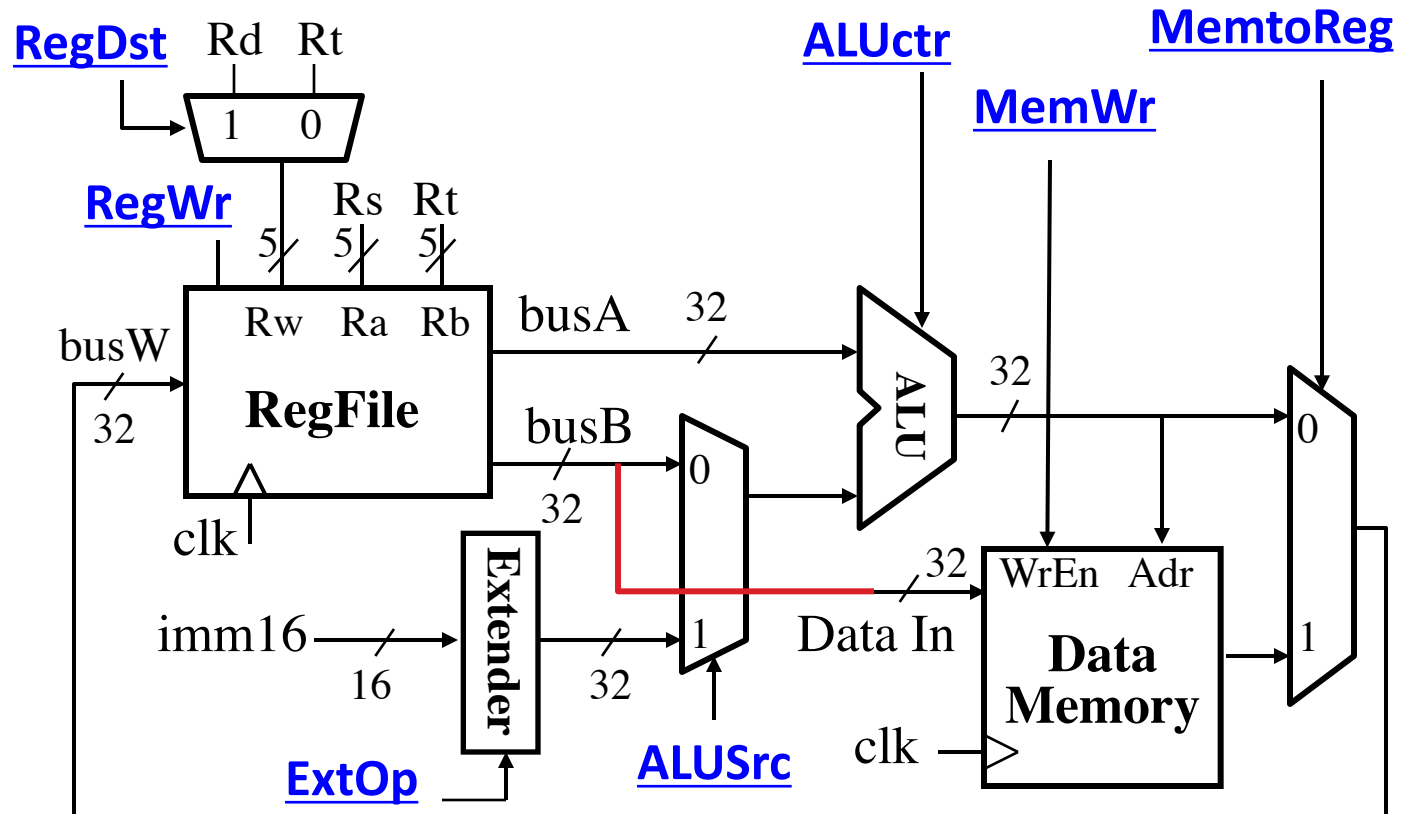
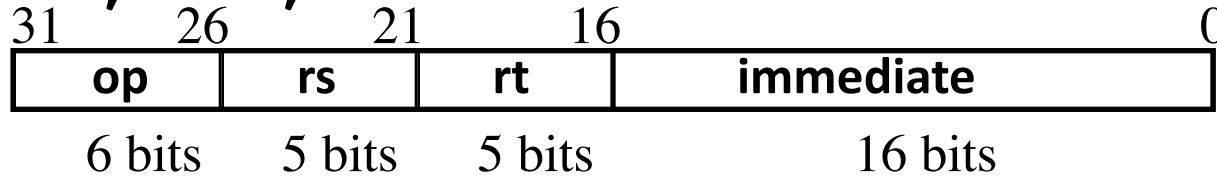
Ex.: `sw rt, rs, imm16`



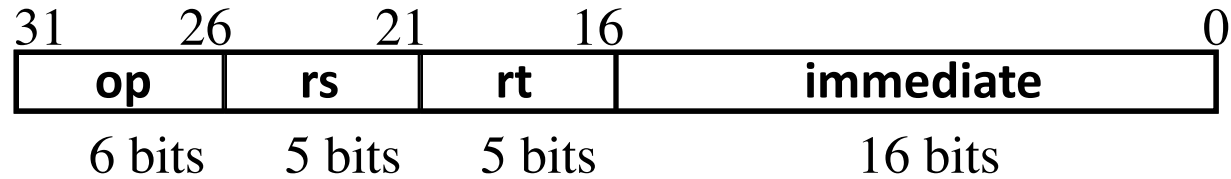
3e: Store Operations

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

Ex.: sw rt, rs, imm16



3f: The Branch Instruction



`beq rs, rt, imm16`

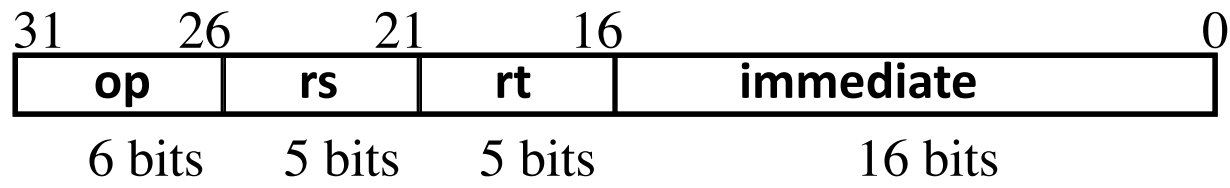
- `mem[PC]` Fetch the instruction from memory
- Equal = $(R[rs] == R[rt])$ Calculate branch condition
- if (Equal) Calculate the next instruction's address
 - $PC = PC + 4 + (\text{SignExt}(\text{imm16}) \times 4)$

else

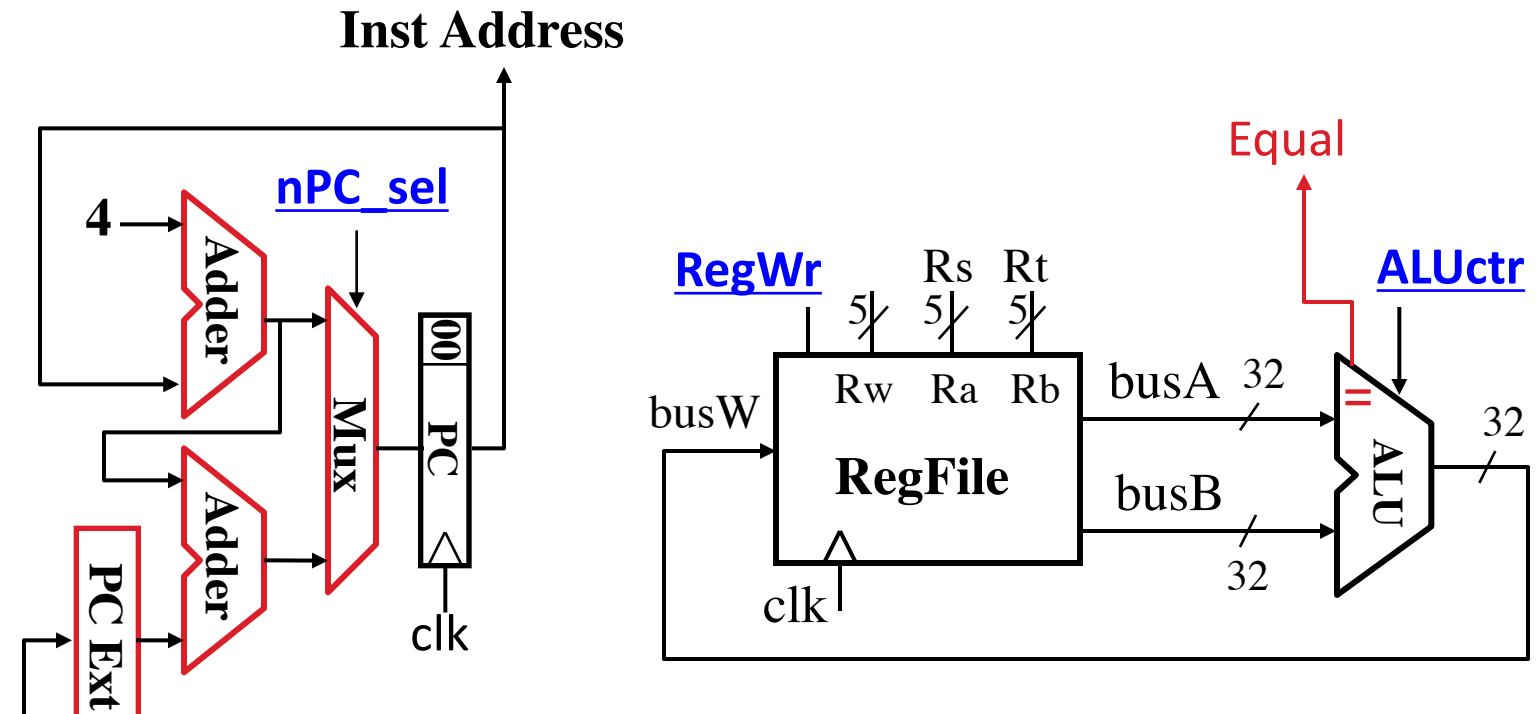
- $PC = PC + 4$

Datapath for Branch Operations

beq rs, rt, imm16

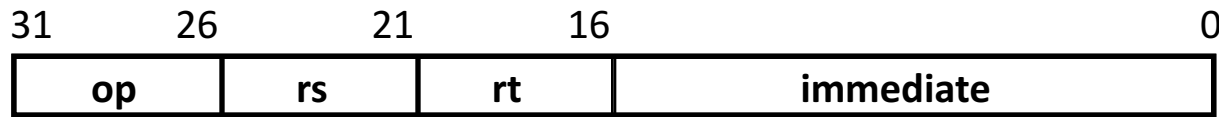


Datapath generates condition (Equal)

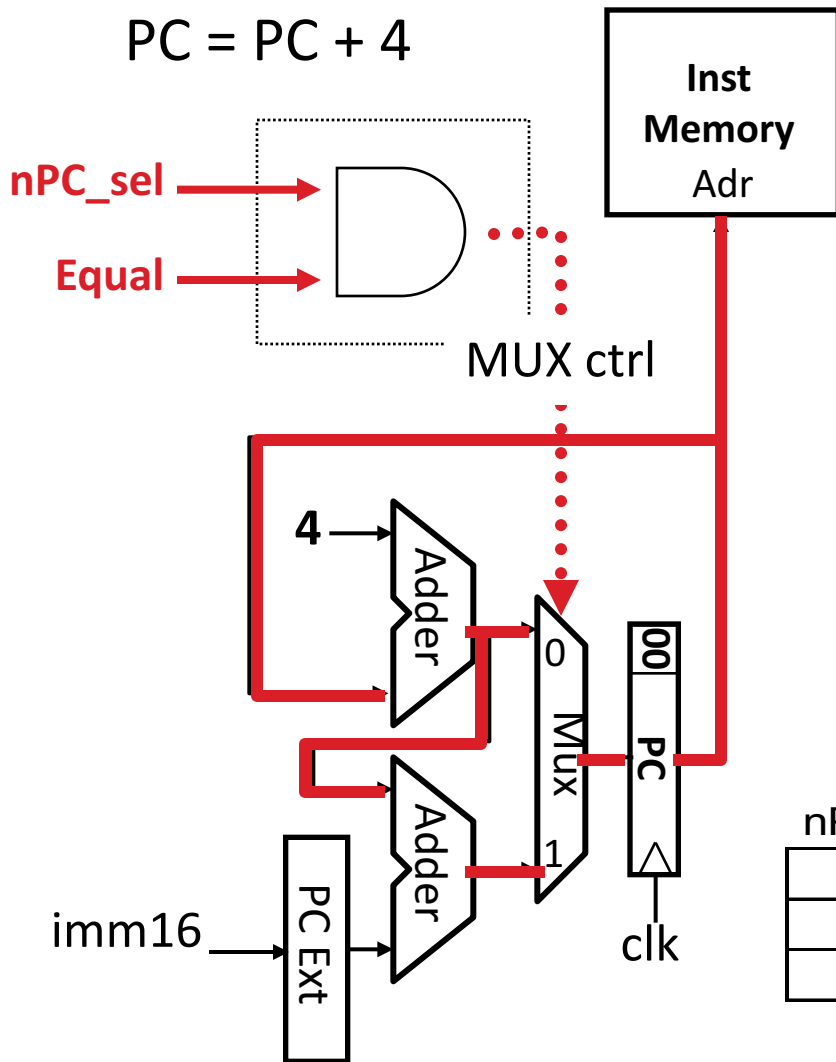


Already have mux, adder, need special sign extender for PC, need equal compare (sub?)

Instruction Fetch Unit including Branch



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



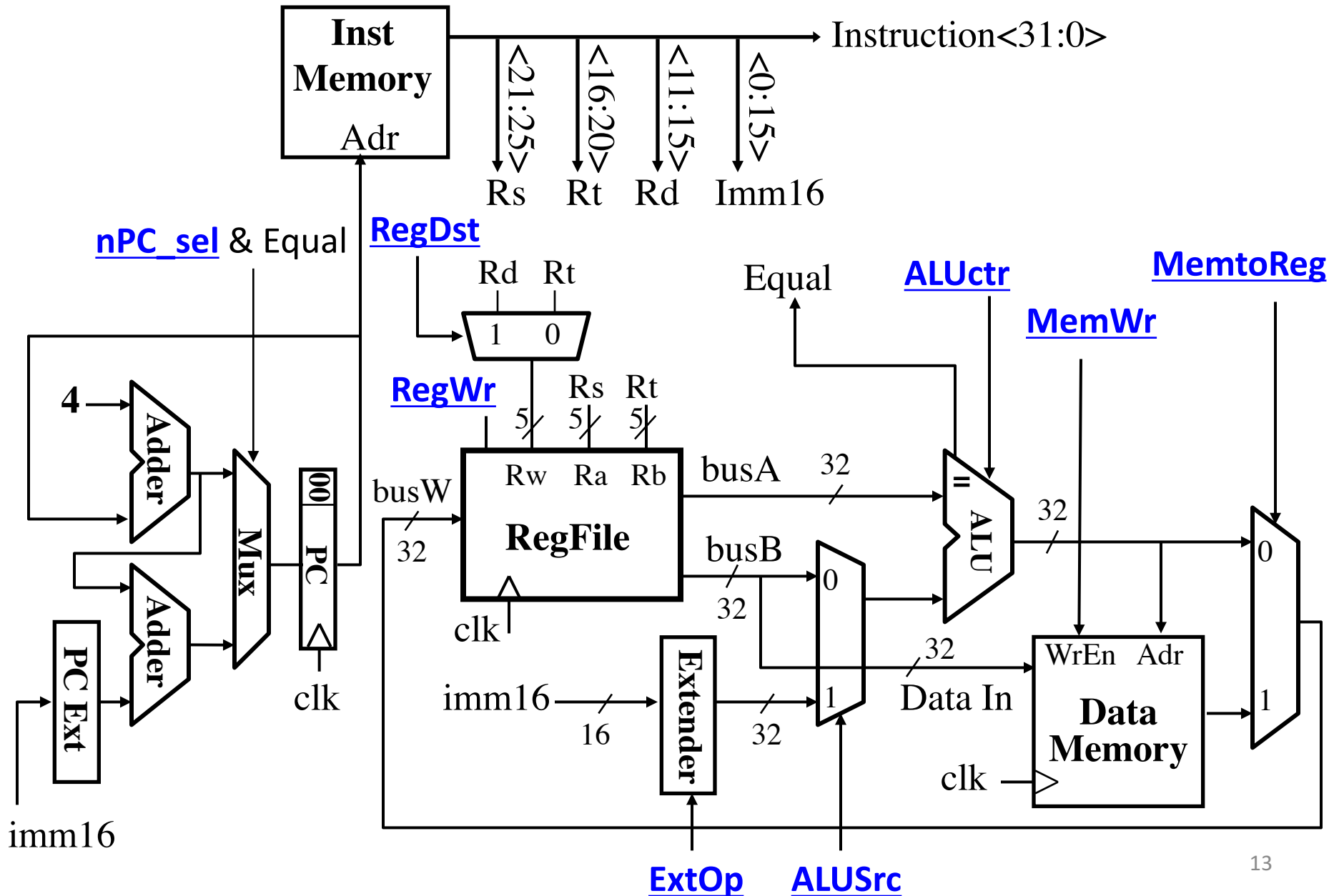
- How to encode nPC_sel?
 - Direct MUX select?
 - Branch inst. / not branch inst.
- Let's pick 2nd option

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

Q: What logic gate?



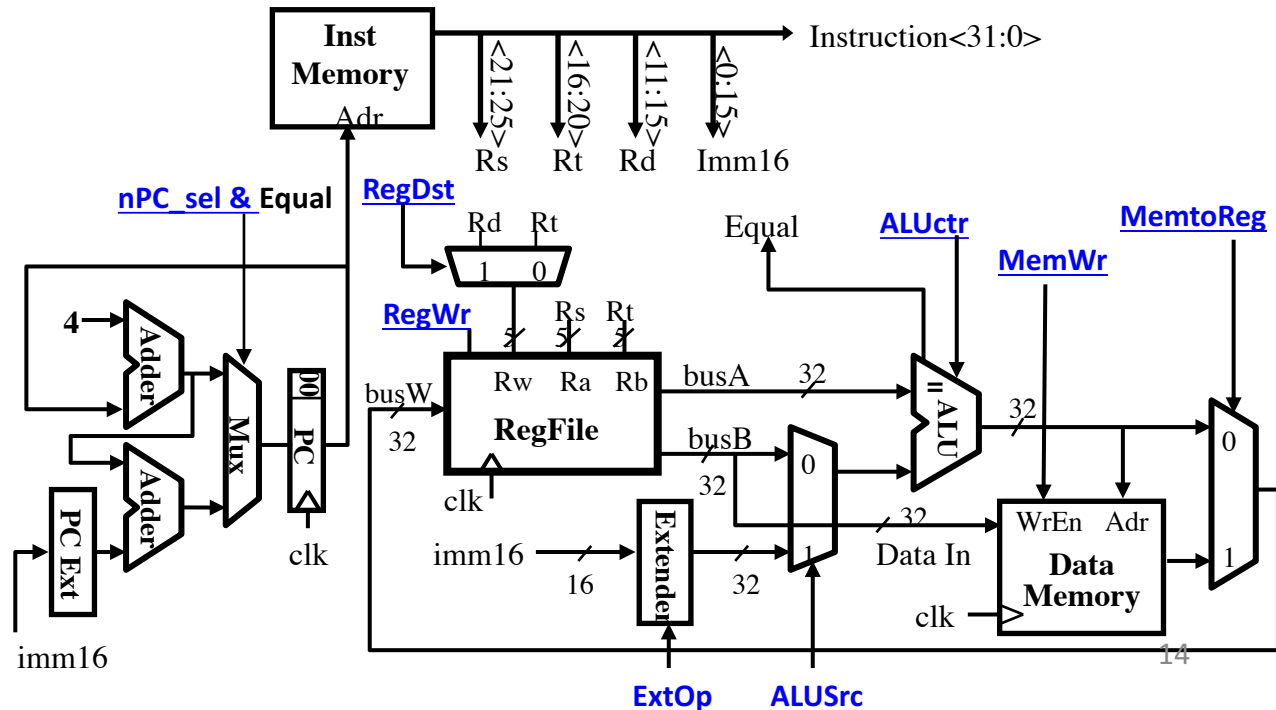
Putting it All Together: A Single Cycle Datapath



Question

What new instruction would need no new datapath hardware?

- A: branch if reg==immediate
- B: add two registers and branch if result zero
- C: store with auto-increment of base address:
 - sw rt, rs, offset // rs incremented by offset after store
- D: shift left logical by two bits



Processor Design: 5 steps

Step 1: Analyze instruction set to determine datapath requirements

- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

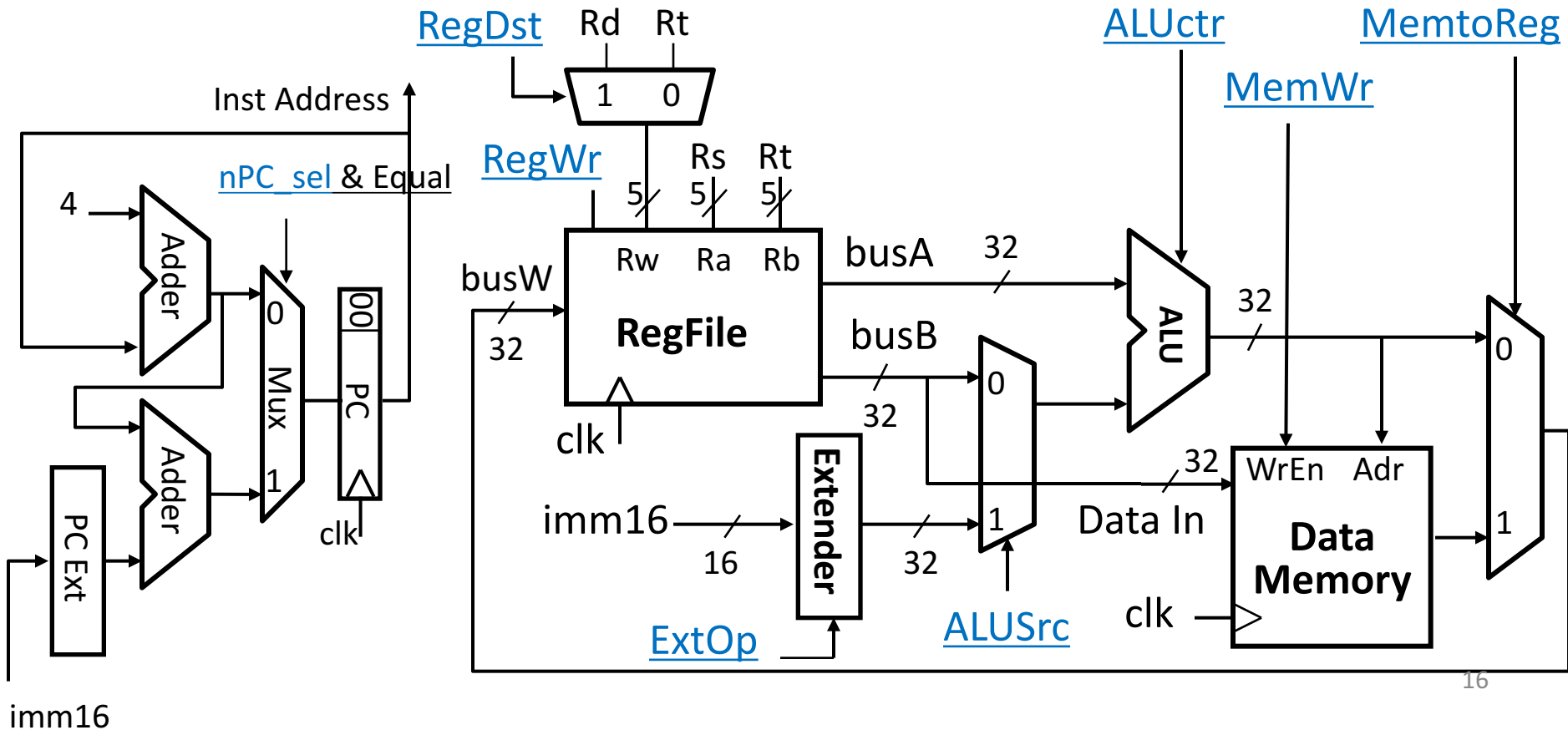
Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

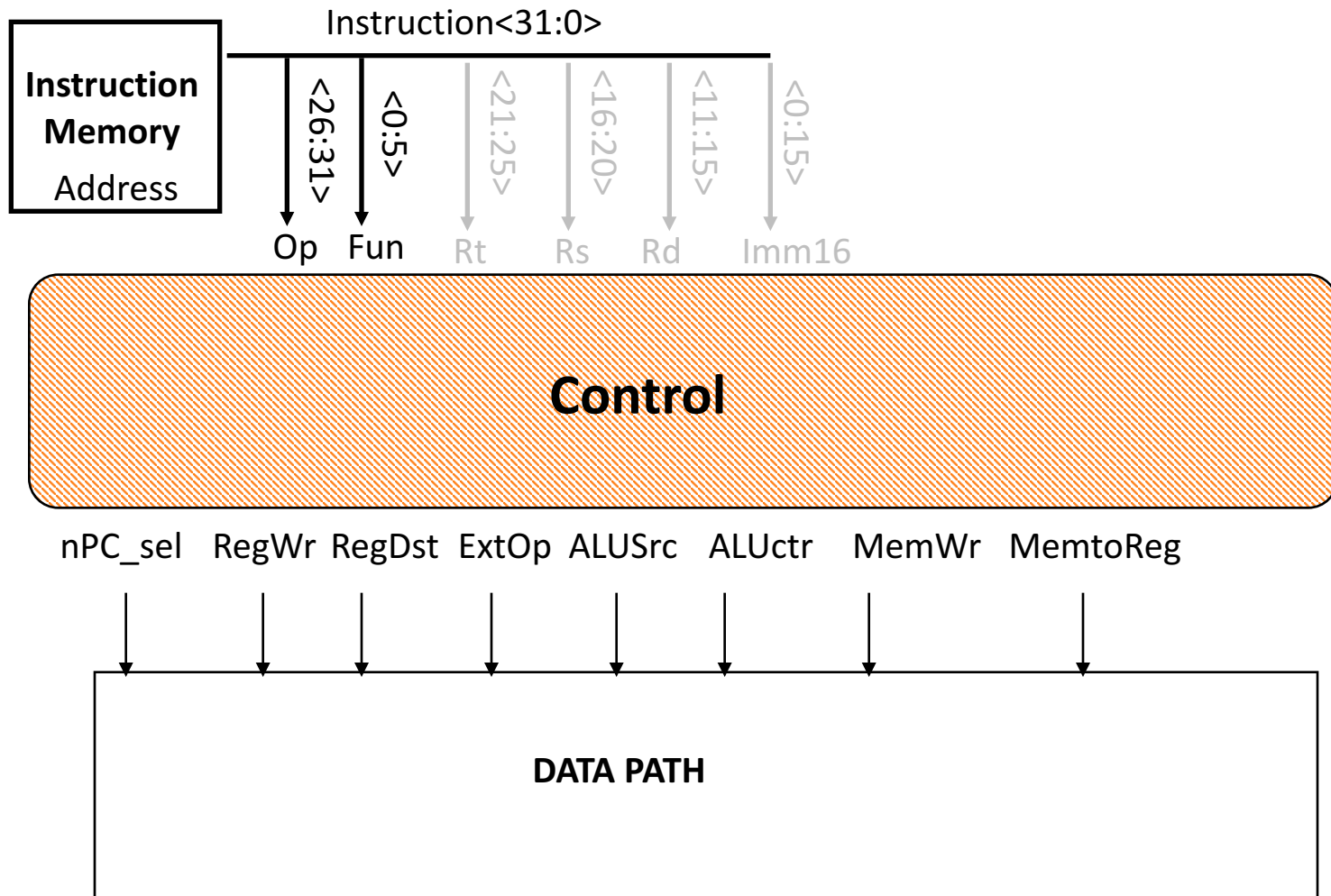
Step 5: Assemble the control logic

Datapath Control Signals

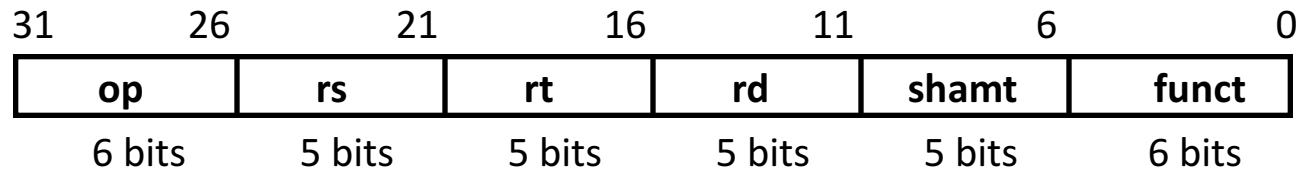
- ExtOp: “zero”, “sign”
- ALUsrc: 0 => regB; 1 => immed
- ALUctr: “ADD”, “SUB”, “OR”
- nPC_sel: 1 => branch
- MemWr: 1 => write memory
- MemtoReg: 0 => ALU; 1 => Mem
- RegDst: 0 => “rt”; 1 => “rd”
- RegWr: 1 => write register



Given Datapath: RTL \rightarrow Control



RTL: The Add Instruction



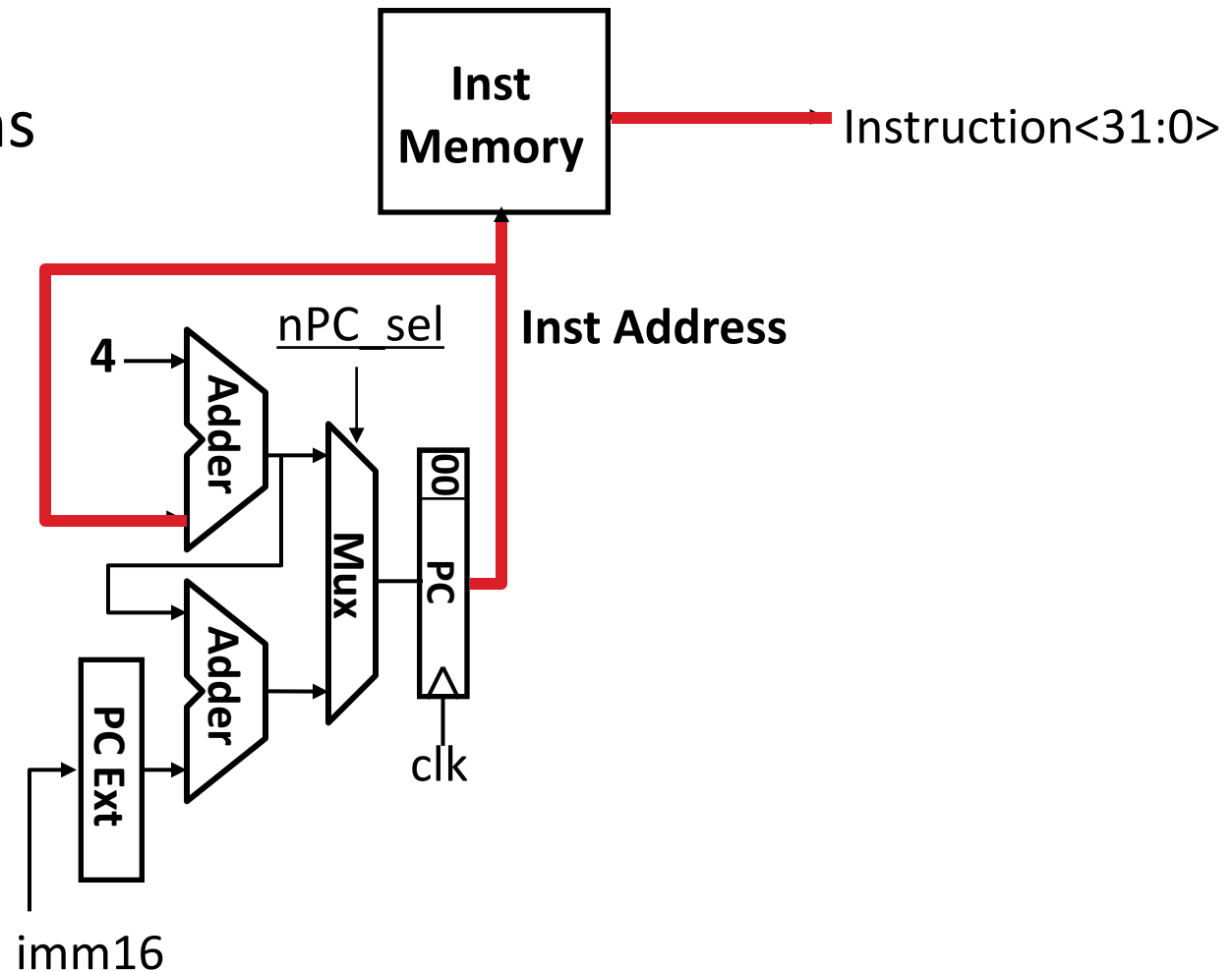
`add rd, rs, rt`

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

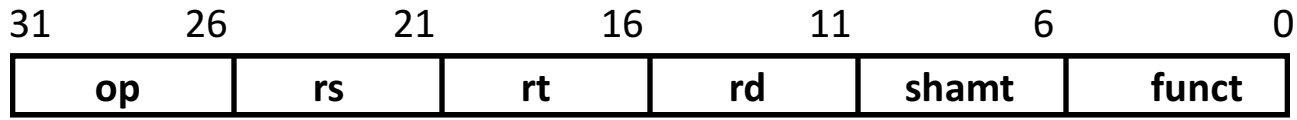
Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory: $\text{Instruction} = \text{MEM}[\text{PC}]$

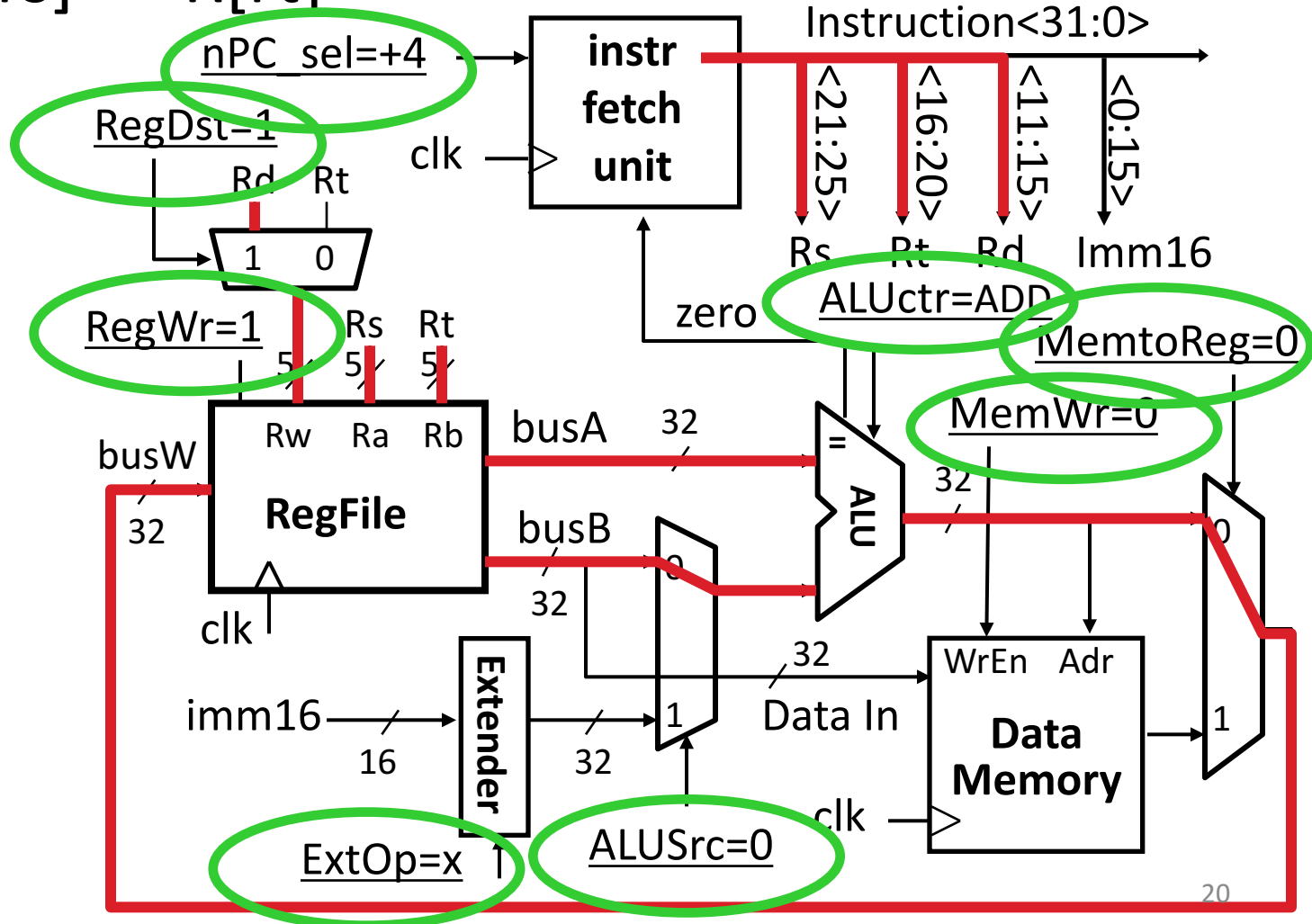
– same for all instructions



Single Cycle Datapath during Add

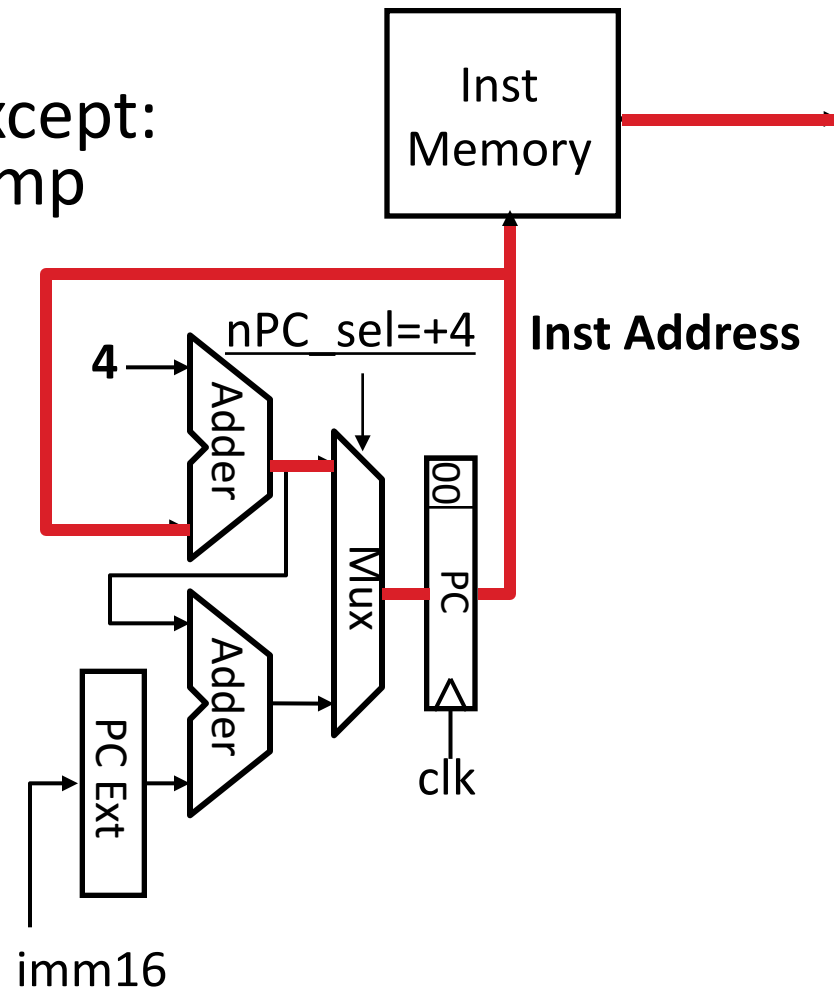


$$R[rd] = R[rs] + R[rt]$$



Instruction Fetch Unit at End of Add

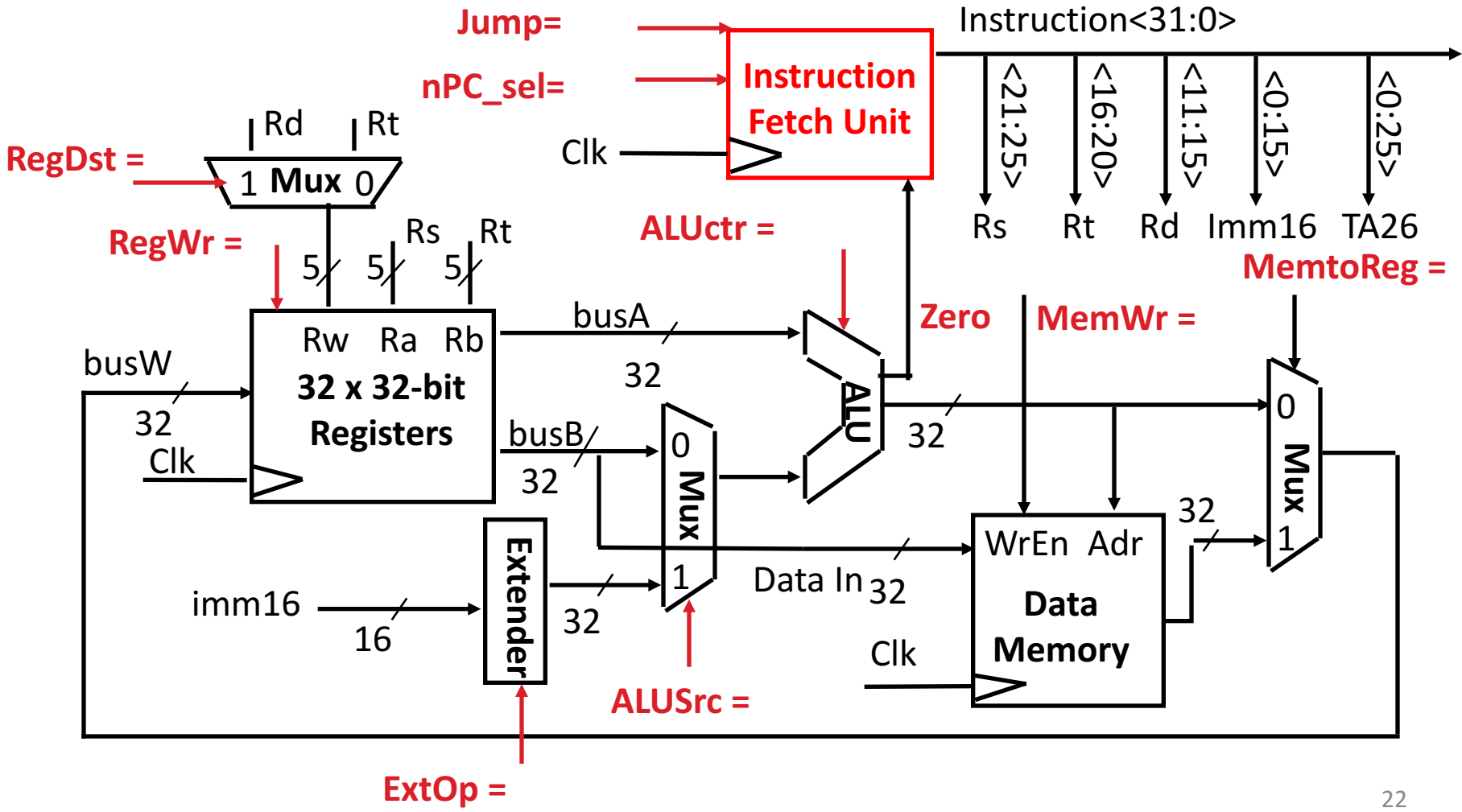
- $PC = PC + 4$
 - Same for all instructions except:
Branch and Jump



Single Cycle Datapath during Jump



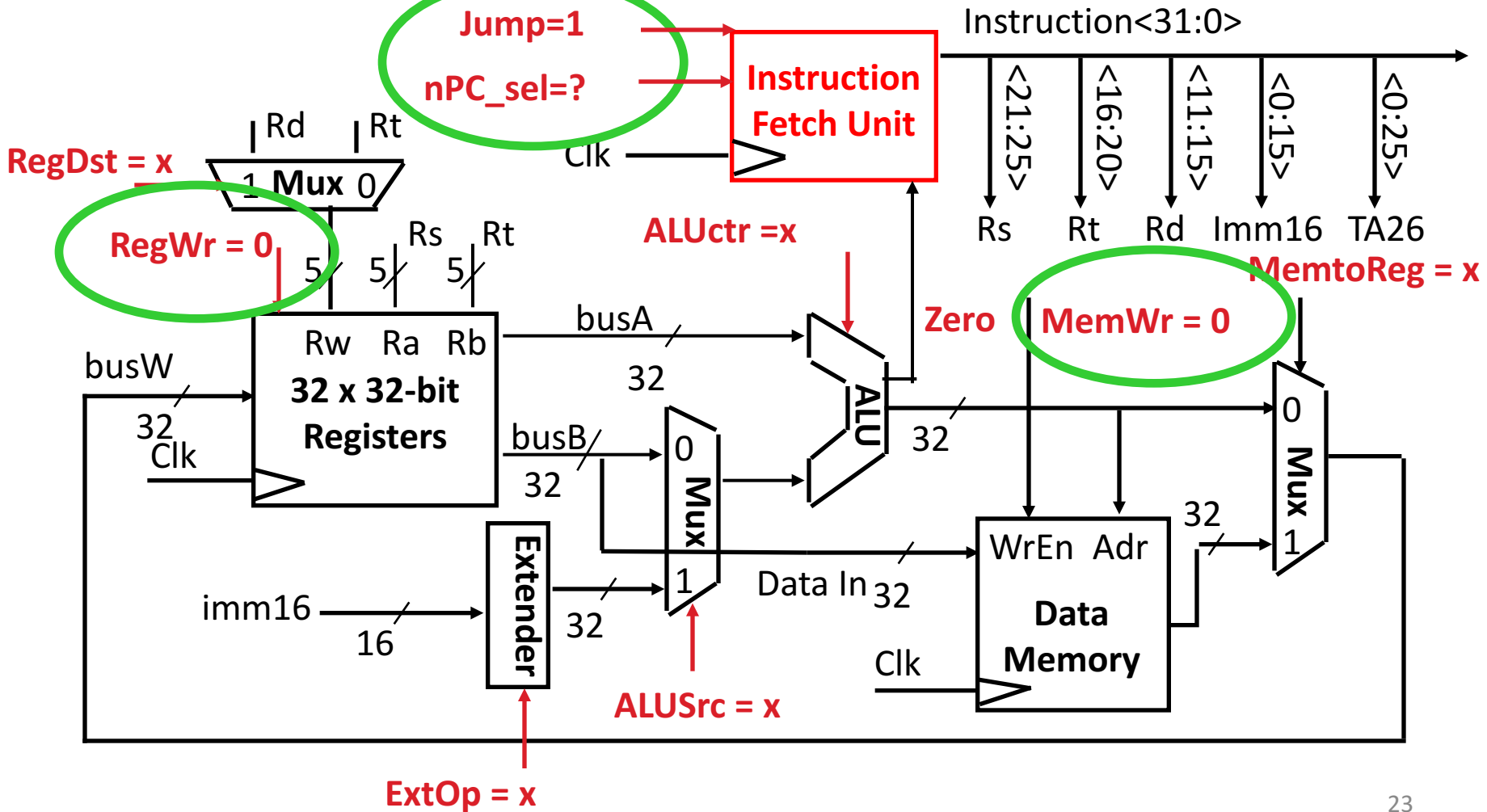
- New PC = { PC[31..28], target address, 00 }



Single Cycle Datapath during Jump



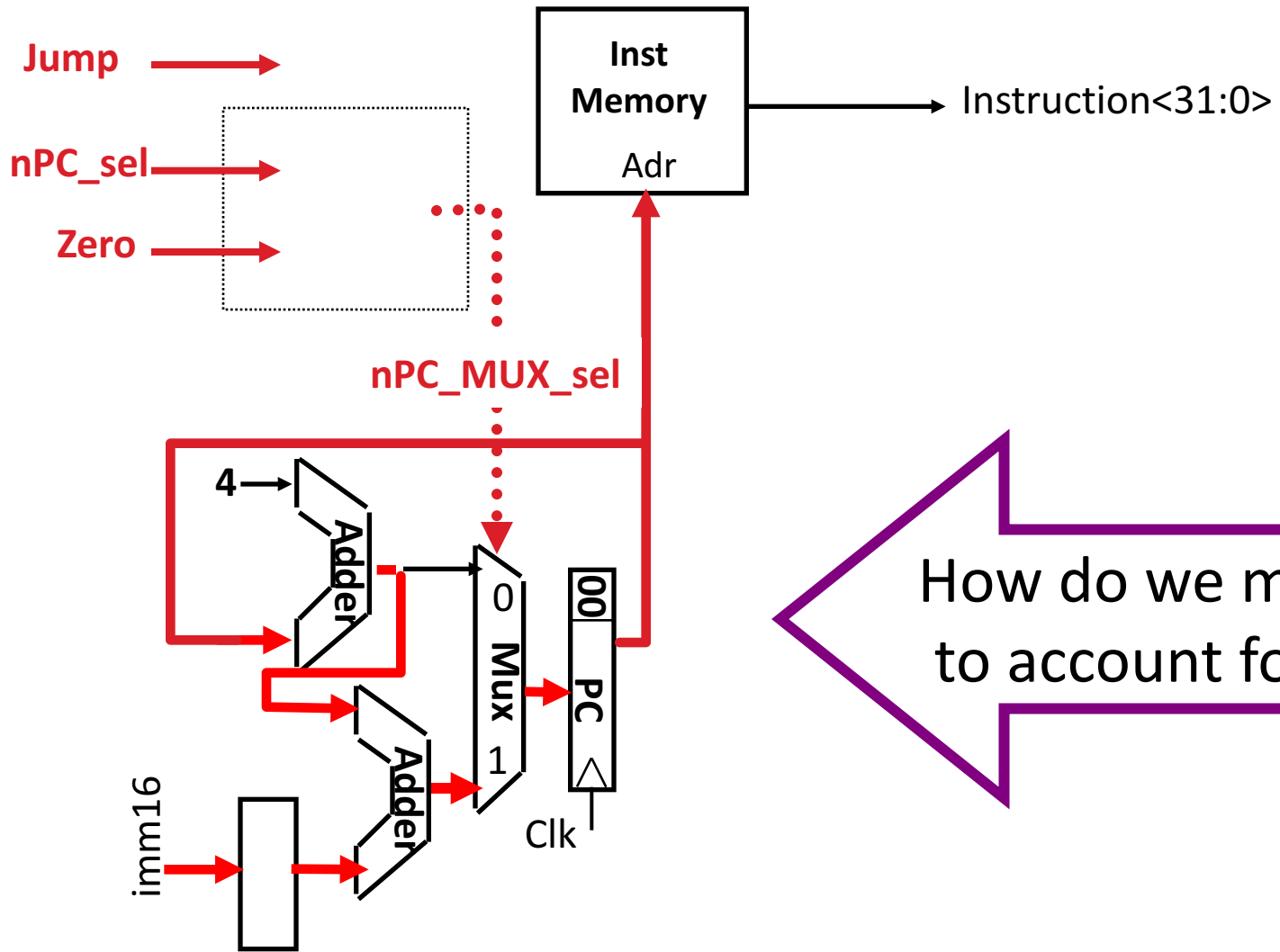
- New PC = { PC[31..28], target address, 00 }



Instruction Fetch Unit at the End of Jump



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



How do we modify this to account for jumps?

Question

Which of the following is TRUE?

- A. The clock can have a shorter period for instructions that don't use memory
- B. The ALU is used to set PC to PC+4 when necessary
- C. Worst-delay path in Instruction Fetch unit is Add+mux delay
- D. The CPU's control needs only *opcode* to determine the next PC value to select
- E. `npc_sel` affects the next PC address on a *jump*

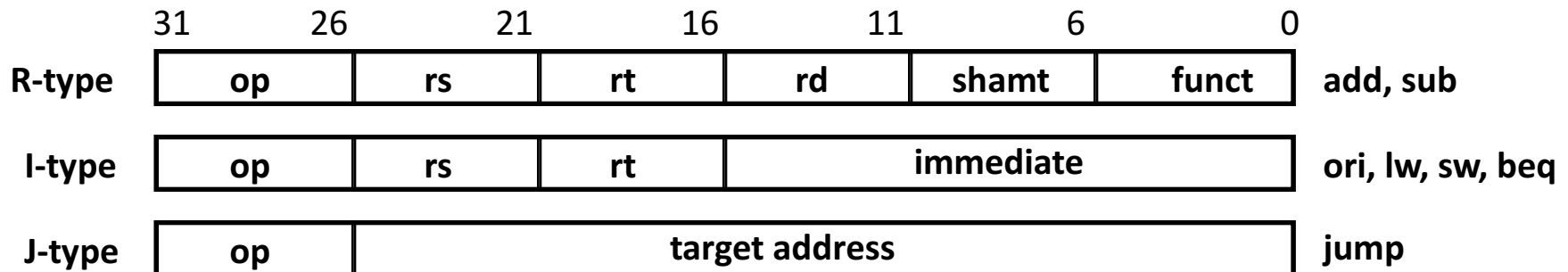
Summary of the Control Signals (1/2)

```
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"
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"
```

Summary of the Control Signals (2/2)

See Appendix A → **func**
 See Appendix A → **op**

	10 0000	10 0010	We Don't Care :-)				
	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	?
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	ADD	0000 00ss ssst tttt dddd d 000 00	10 0000
ALUSrc	= ori + lw + sw	SUB	0000 00ss ssst tttt dddd d 000 00	10 0010
MemtoReg	= lw	ORI	0011 01ss ssst tttt iiiiiiii iiiiiiii	
RegWrite	= add + sub + ori + lw	LW	1000 11ss ssst tttt iiiiiiii iiiiiiii	
MemWrite	= sw	SW	1010 11ss ssst tttt iiiiiiii iiiiiiii	
nPCsel	= beq	BEQ	0001 00ss ssst tttt iiiiiiii iiiiiiii	
Jump	= jump	JUMP	0000 10ii iiiiiiii iiiiiiii iiiiiiii	
ExtOp	= lw + sw			
ALUctr[0]	= sub + beq	(assume ALUctr is 00 ADD, 01 SUB, 10 OR)		
ALUctr[1]	= or			

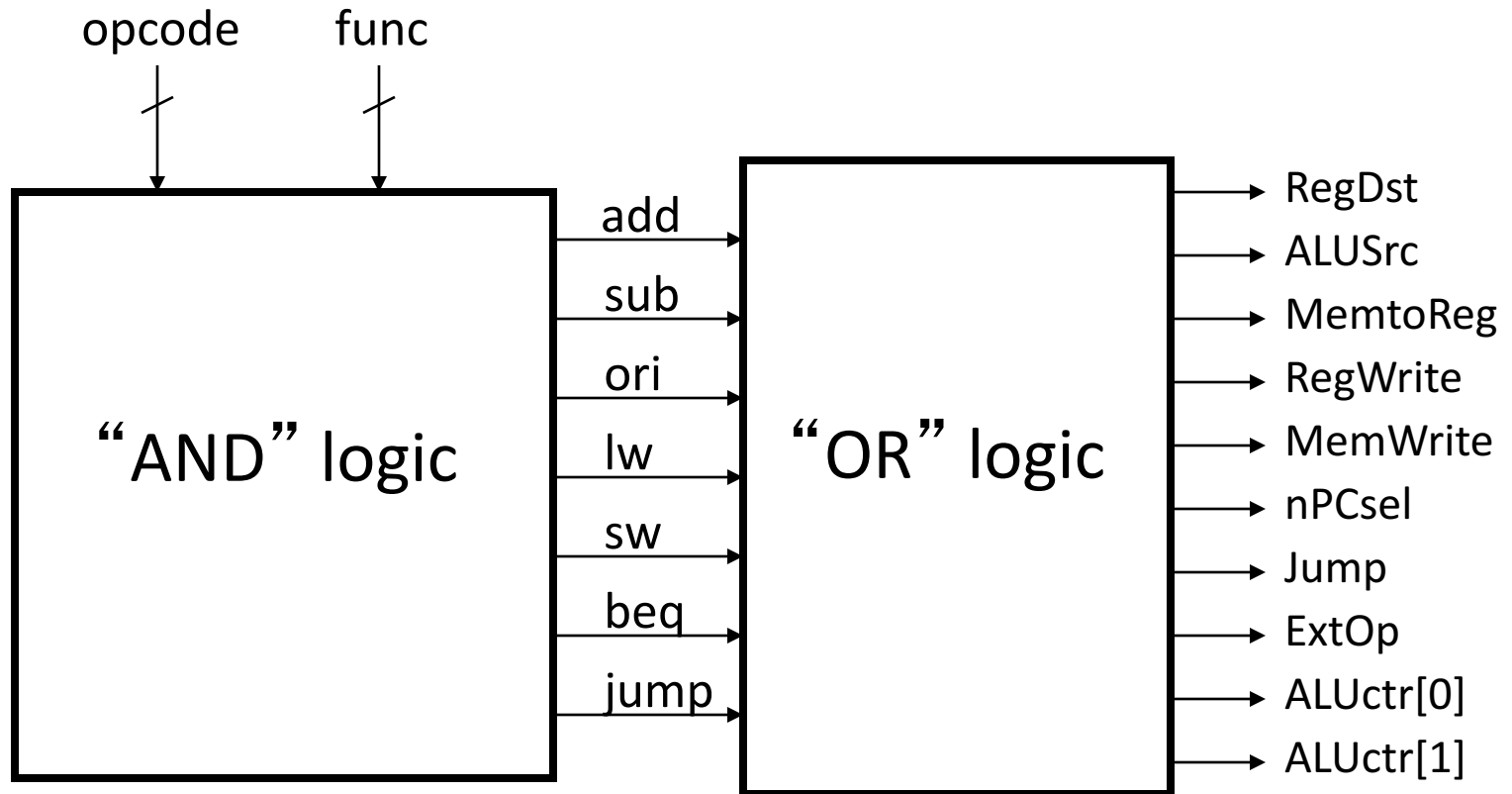
Where:

$$\begin{aligned} \text{rtype} &= \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \sim\text{op}_2 \cdot \sim\text{op}_1 \cdot \sim\text{op}_0 \\ \text{ori} &= \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \text{op}_3 \cdot \text{op}_2 \cdot \sim\text{op}_1 \cdot \text{op}_0 \\ \text{lw} &= \text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \sim\text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\ \text{sw} &= \text{op}_5 \cdot \sim\text{op}_4 \cdot \text{op}_3 \cdot \sim\text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\ \text{beq} &= \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \text{op}_2 \cdot \sim\text{op}_1 \cdot \sim\text{op}_0 \\ \text{jump} &= \sim\text{op}_5 \cdot \sim\text{op}_4 \cdot \sim\text{op}_3 \cdot \sim\text{op}_2 \cdot \text{op}_1 \cdot \sim\text{op}_0 \end{aligned}$$

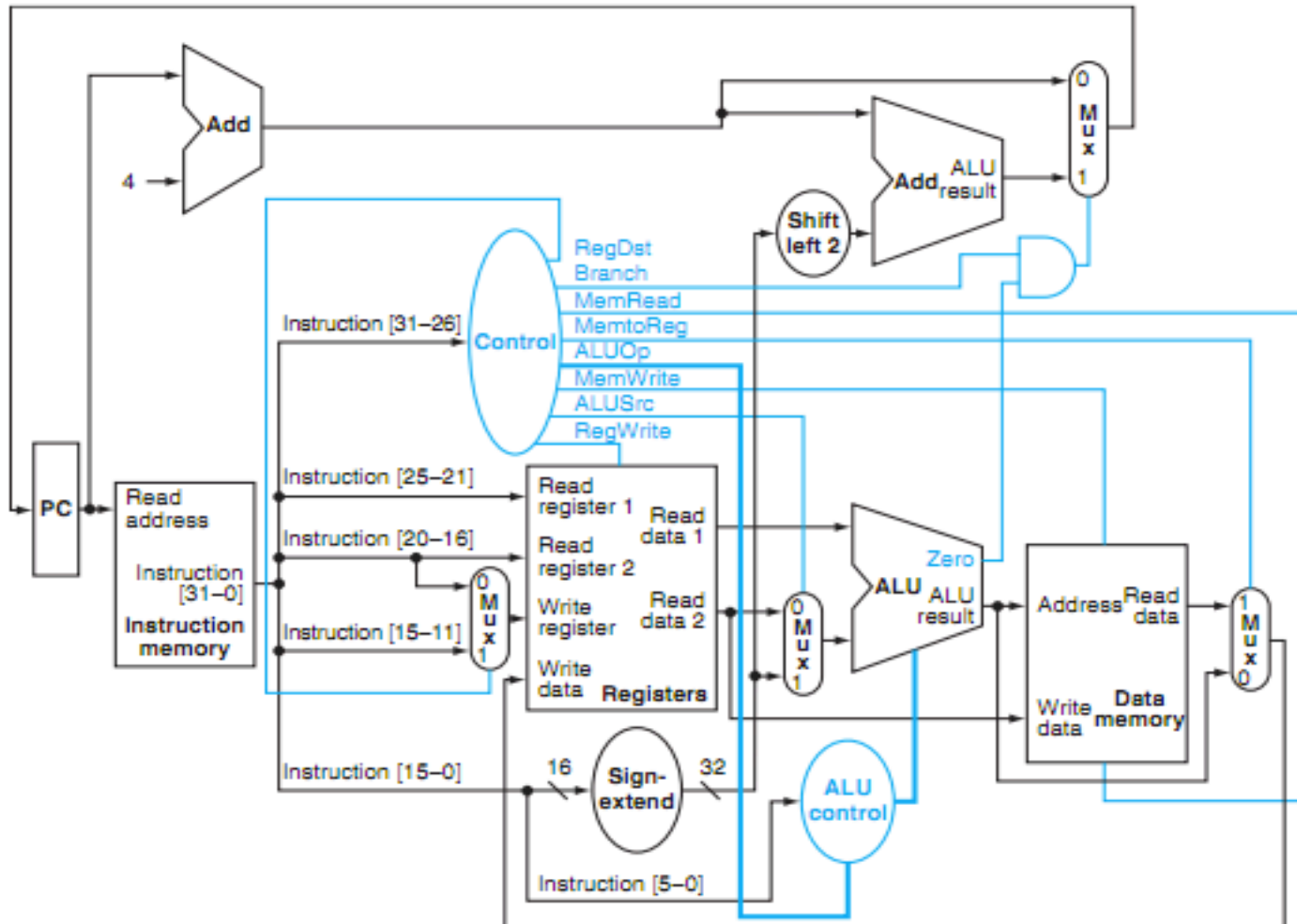
$$\begin{aligned} \text{add} &= \text{rtype} \cdot \text{func}_5 \cdot \sim\text{func}_4 \cdot \sim\text{func}_3 \cdot \sim\text{func}_2 \cdot \sim\text{func}_1 \cdot \sim\text{func}_0 \\ \text{sub} &= \text{rtype} \cdot \text{func}_5 \cdot \sim\text{func}_4 \cdot \sim\text{func}_3 \cdot \sim\text{func}_2 \cdot \text{func}_1 \cdot \sim\text{func}_0 \end{aligned}$$

How do we
implement this in
gates?

Controller Implementation



P&H Figure 4.17



Summary: Single-cycle Processor

- Five steps to design a processor:

1. Analyze instruction set → datapath requirements

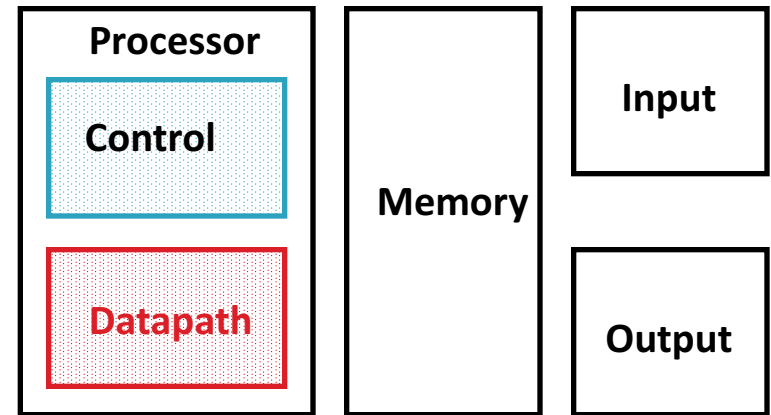
2. Select set of datapath components & establish clock methodology

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



Levels of Representation/Interpretation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

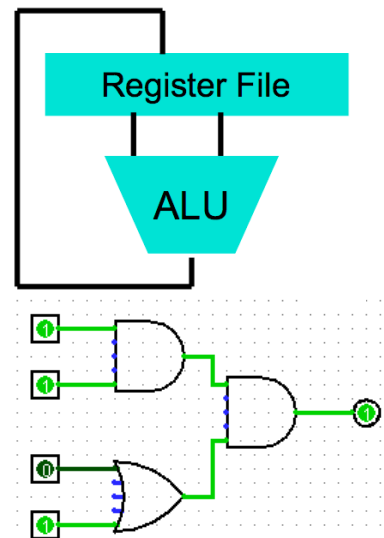
Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



No More Magic!

