CS 61C: Great Ideas in Computer Architecture (Machine Structures) Single-Cycle CPU *Datapath & Control Part 2*

Instructors:

Krste Asanovic & Vladimir Stojanovic

http://inst.eecs.berkeley.edu/~cs61c/

Review: 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

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

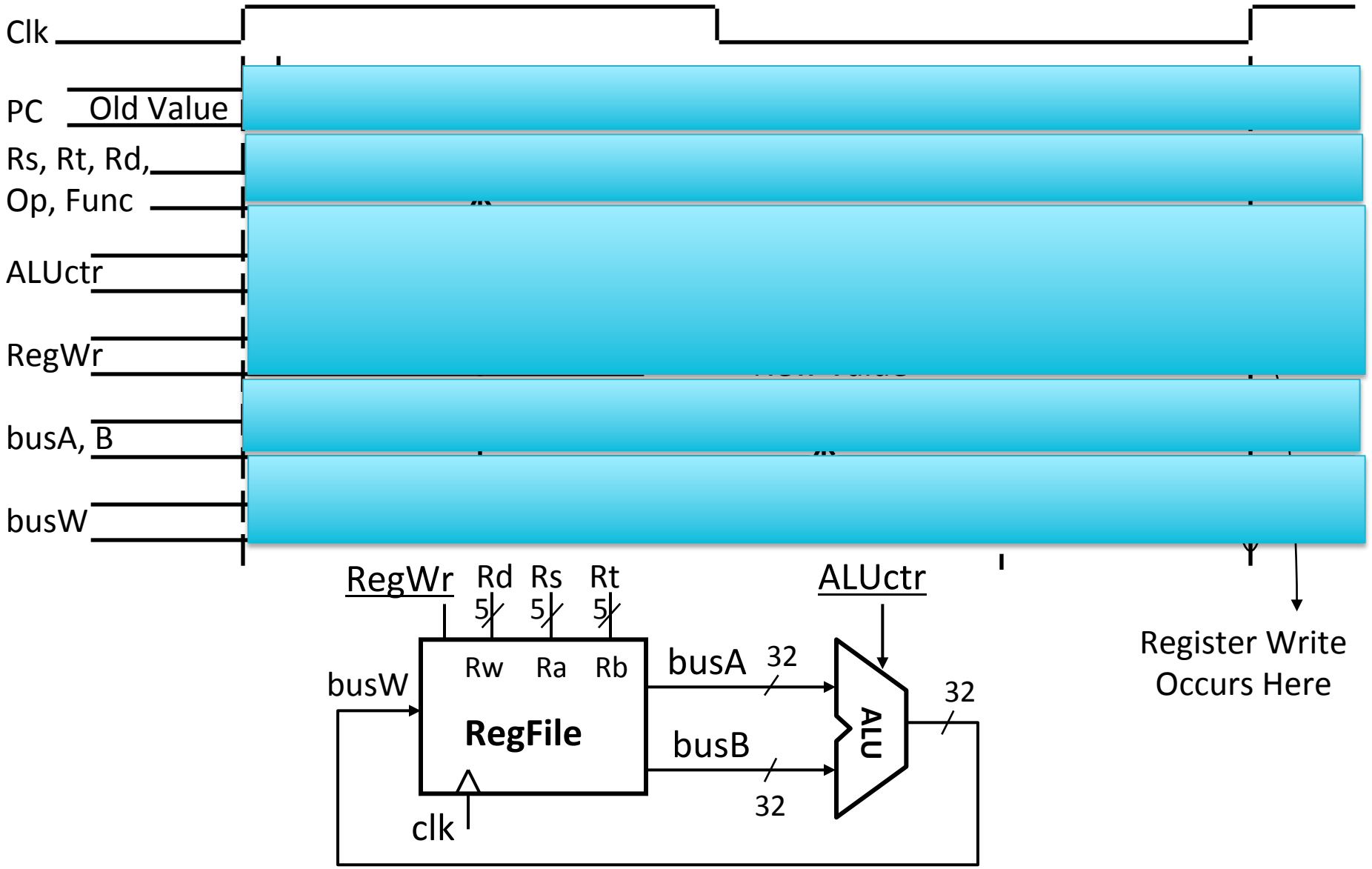
Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

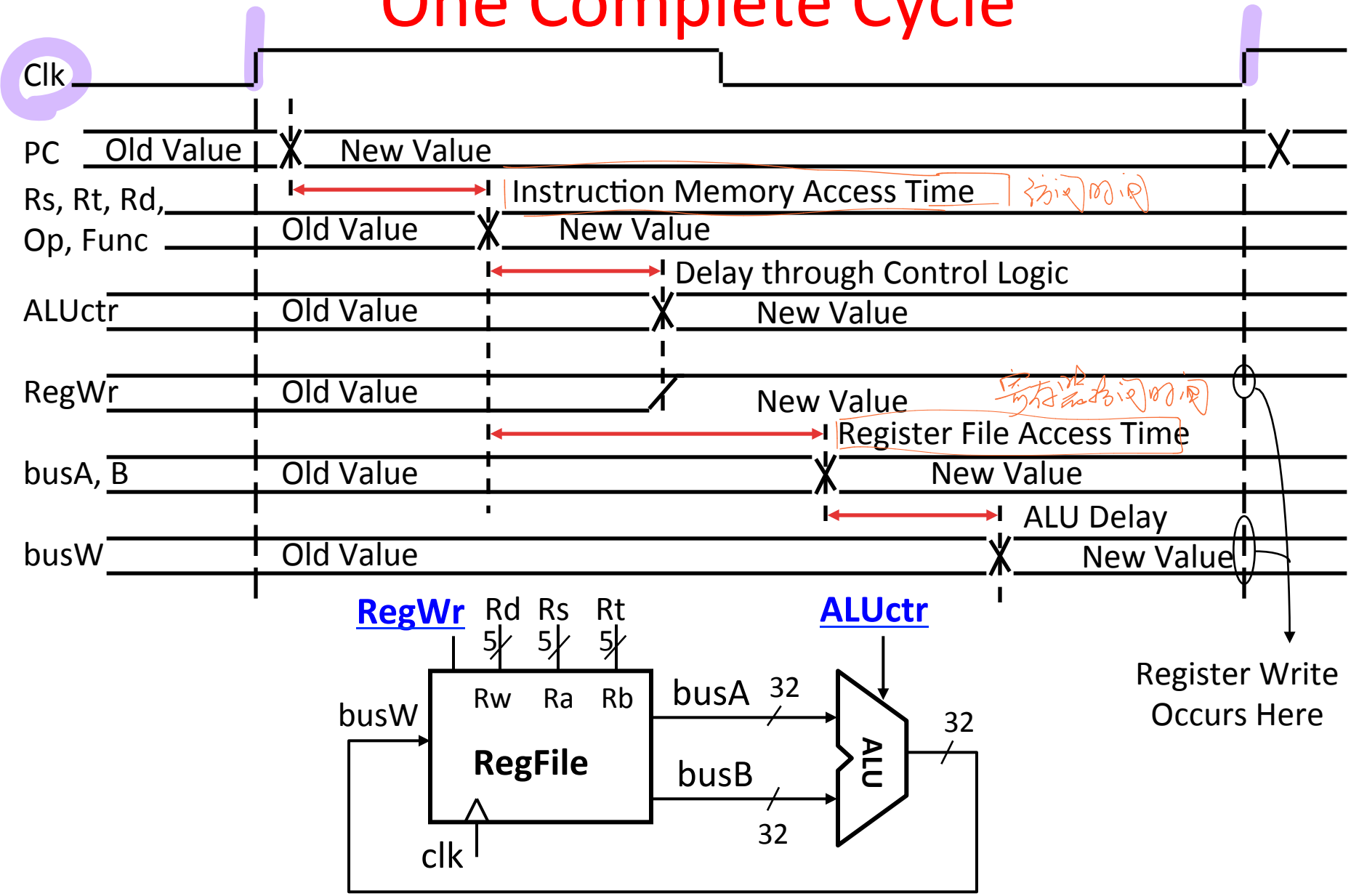
Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic

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

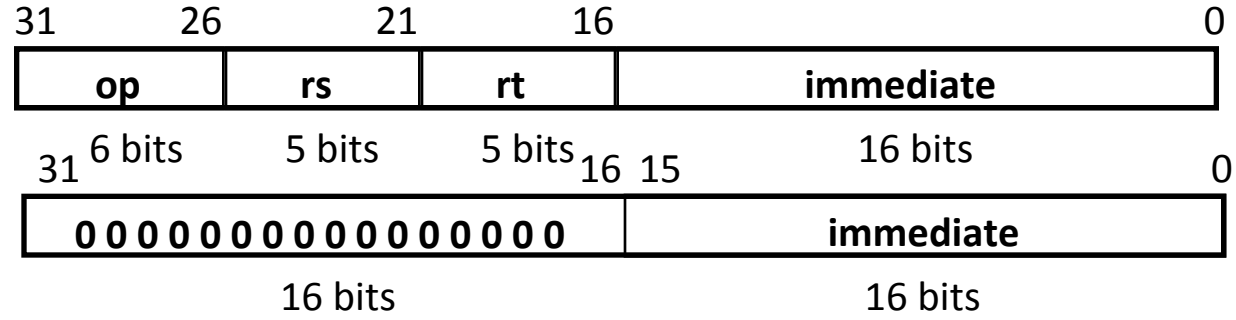


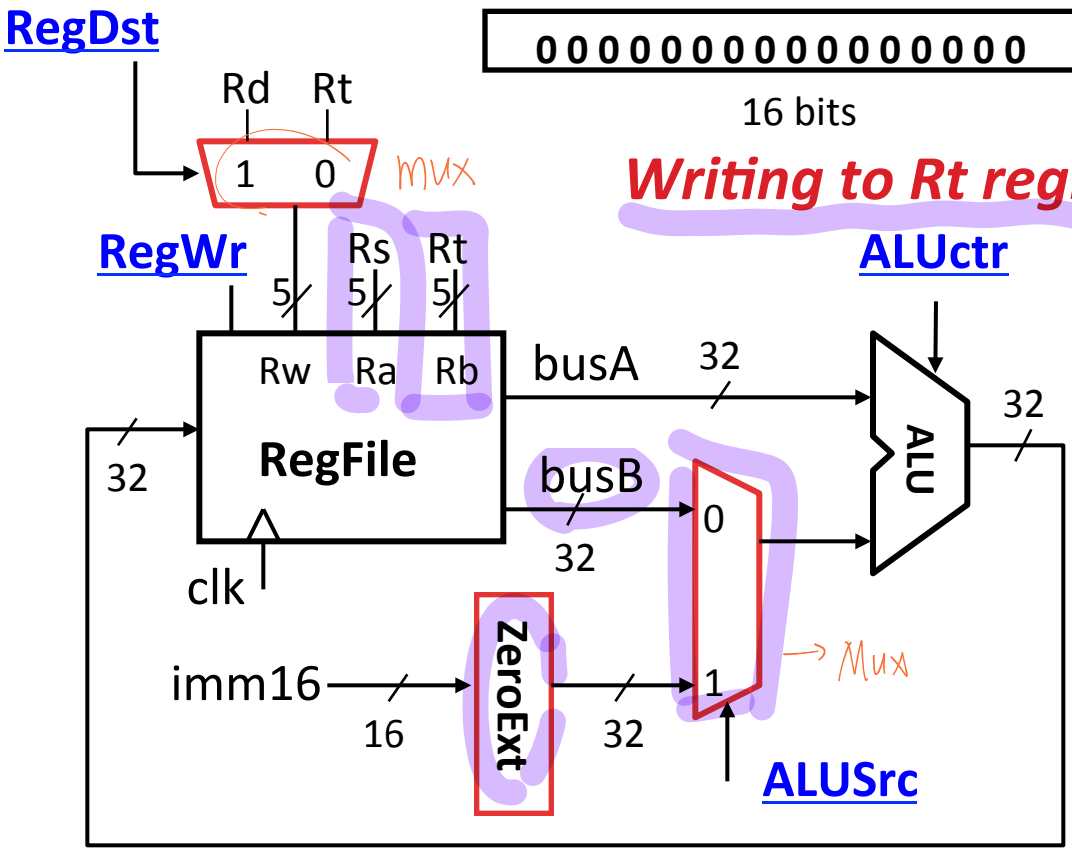
Register-Register Timing: One Complete Cycle



3c: Logical Op (or) with Immediate

- R[rt] = R[rs] op ZeroExt[imm16]



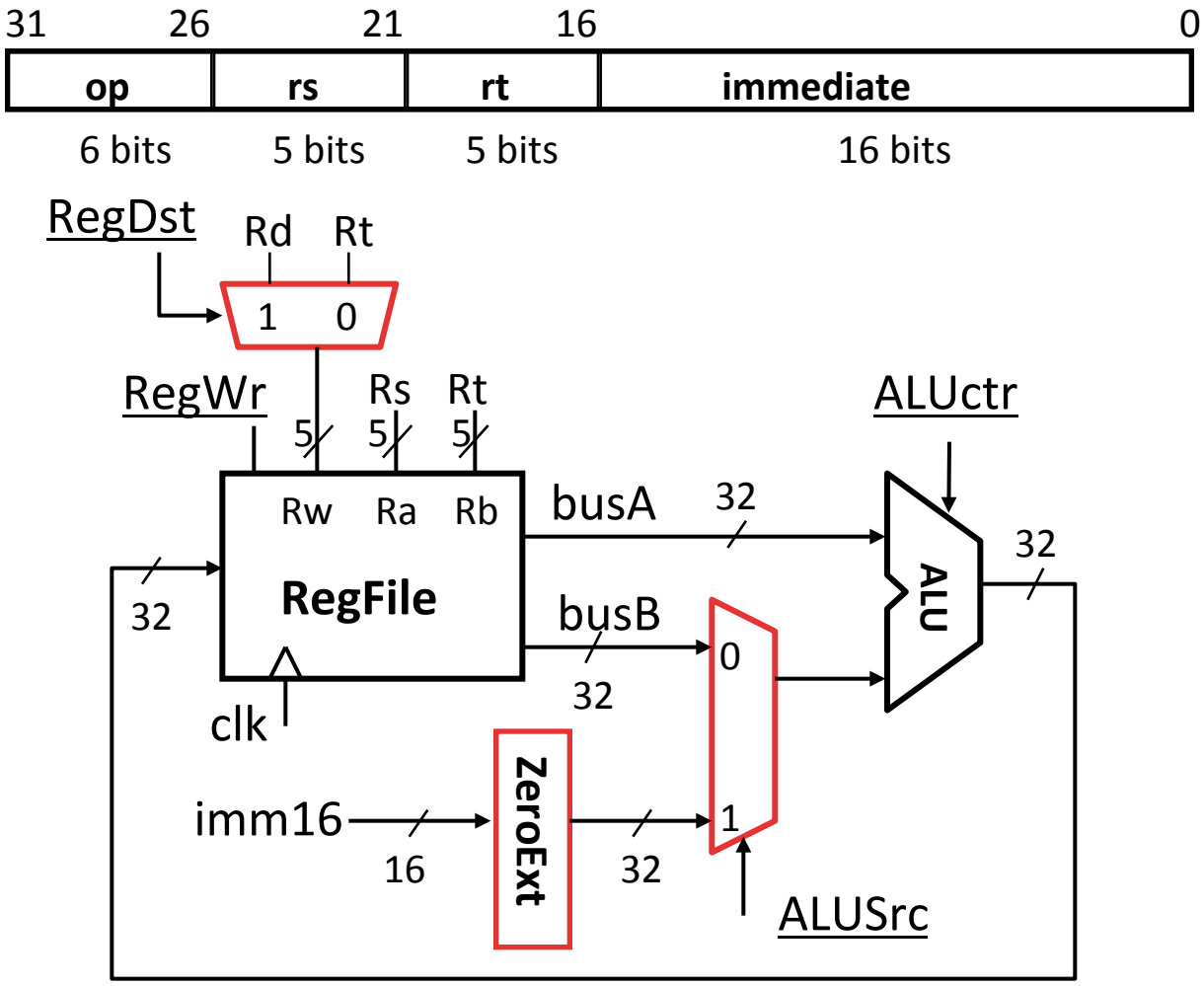


Writing to Rt register (not Rd)!!

What about Rt Read?

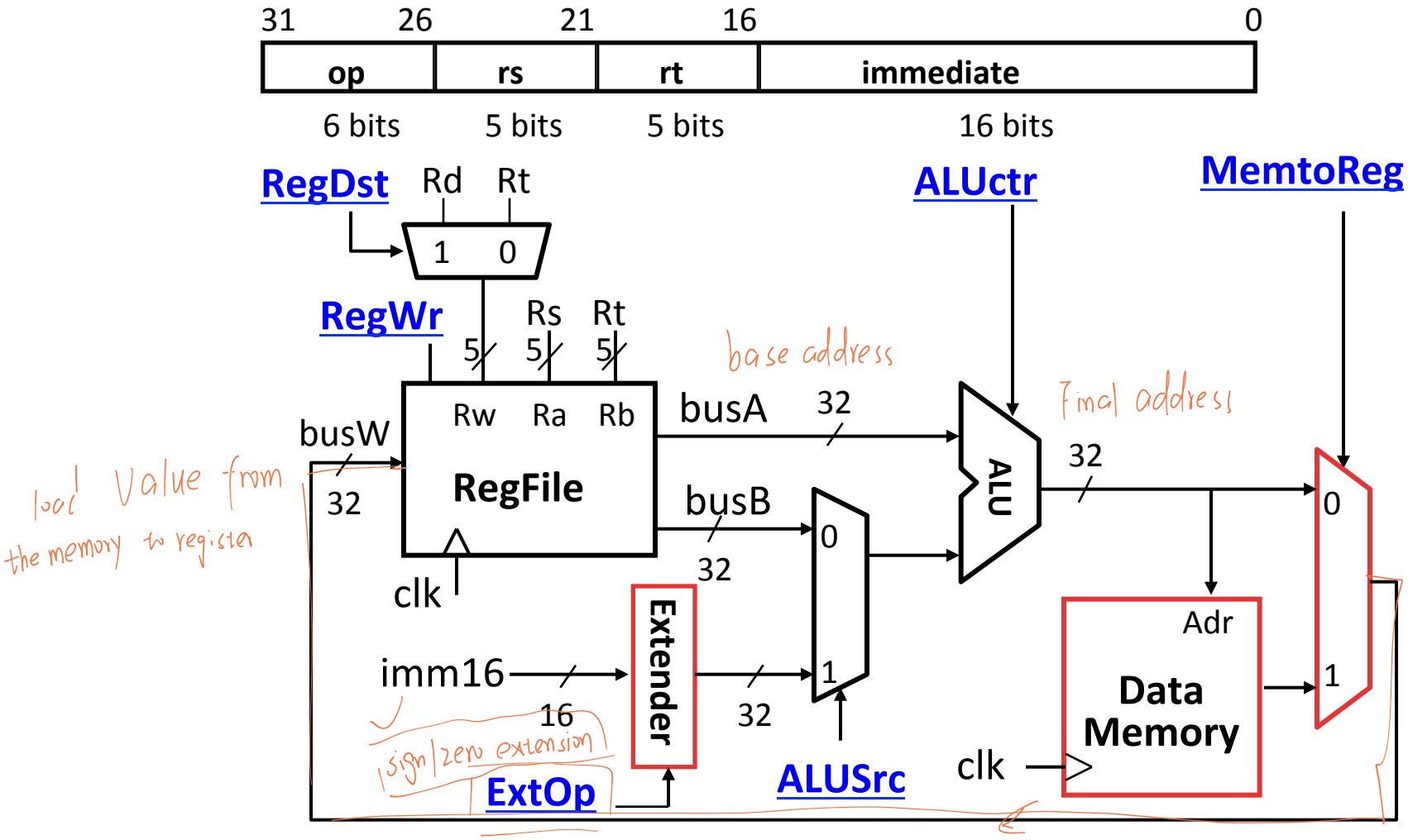
3d: Load Operations

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



3d: Load Operations

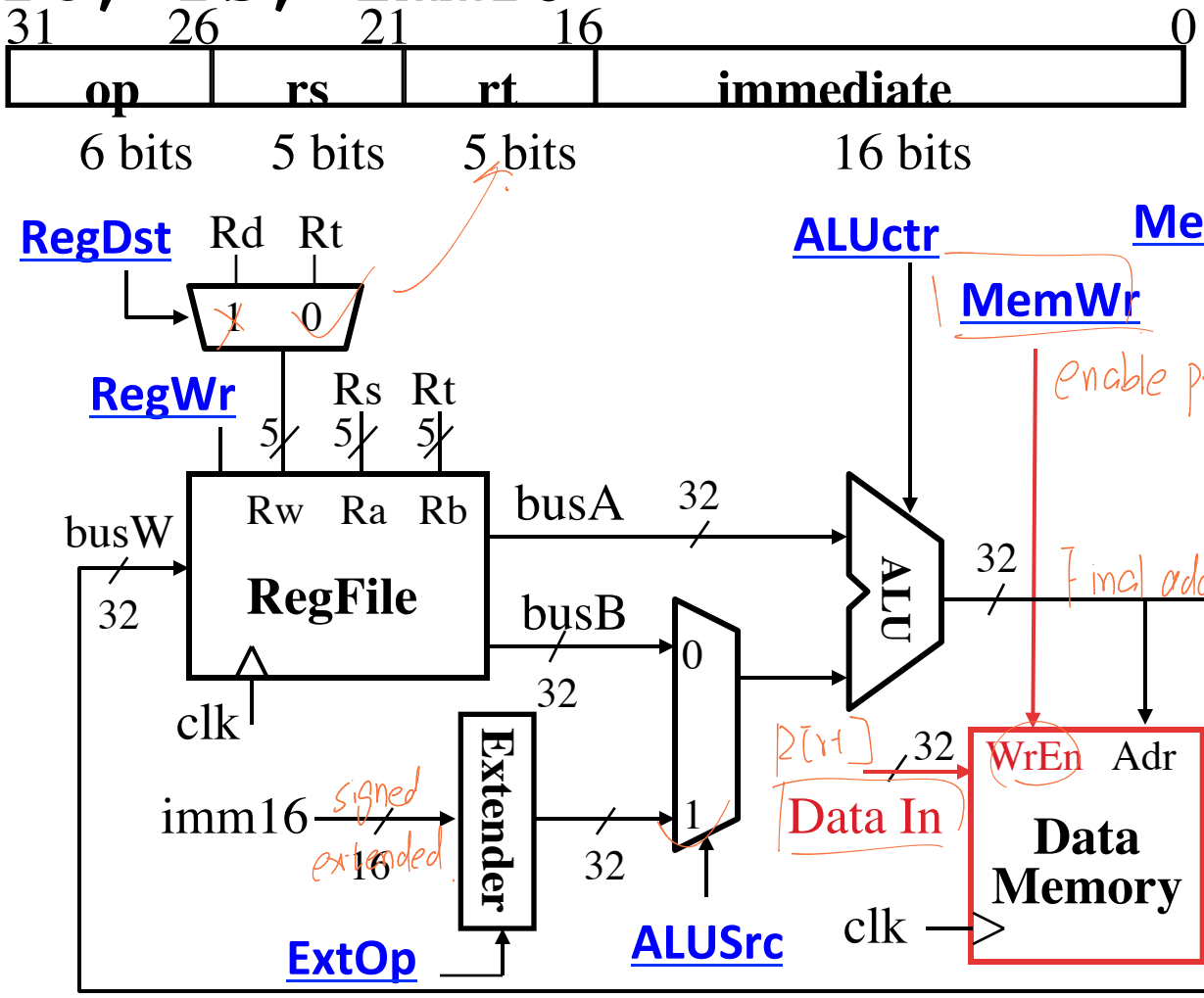
- R[rt] = Mem[R[rs] + SignExt[imm16]]
  Example: `lw rt,rs,imm16`



3e: Store Operations

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

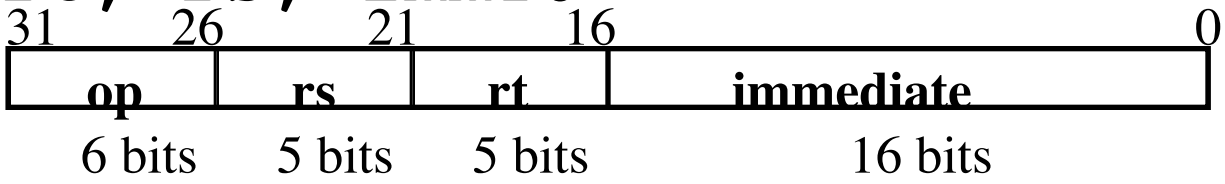
  Ex.: `sw rt, rs, imm16`

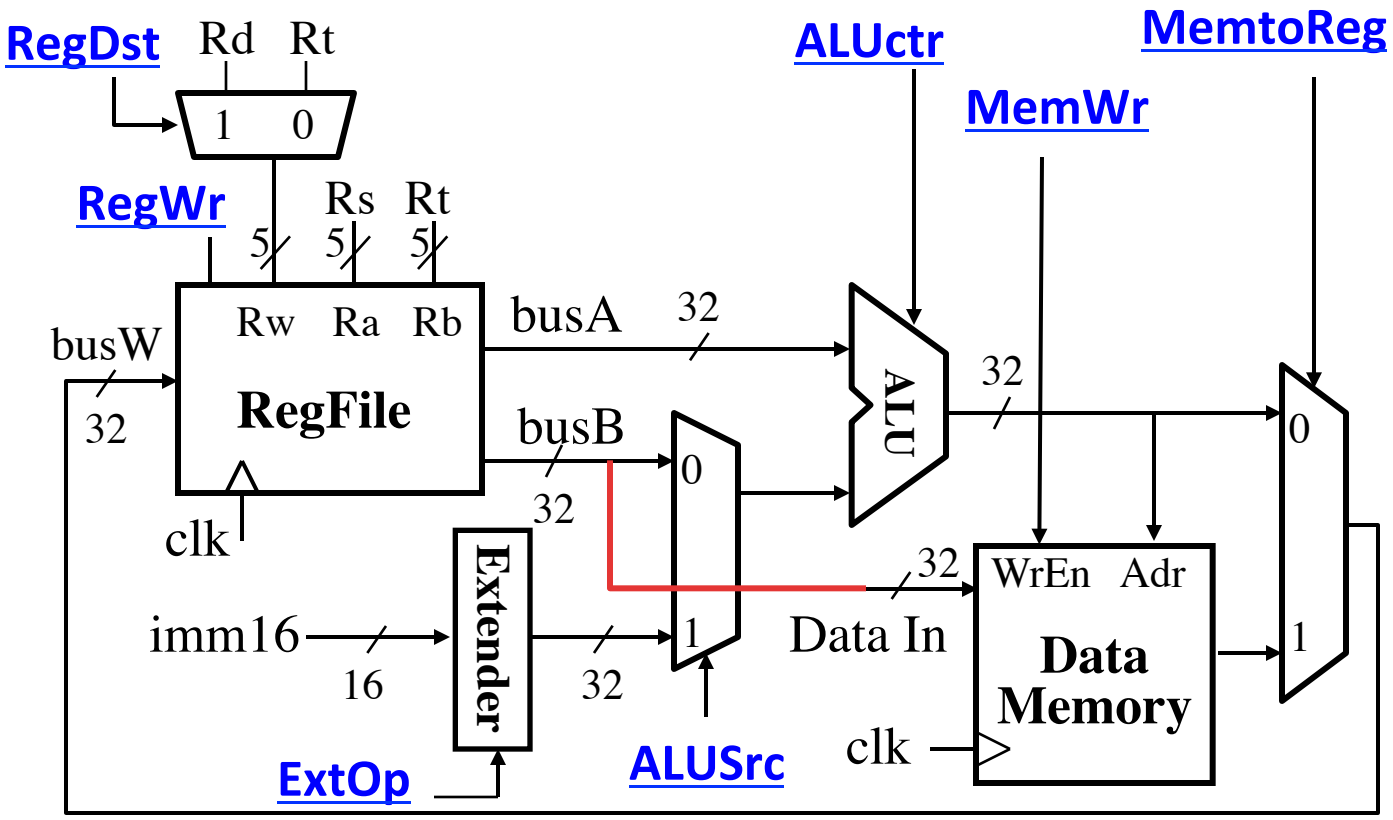


3e: Store Operations

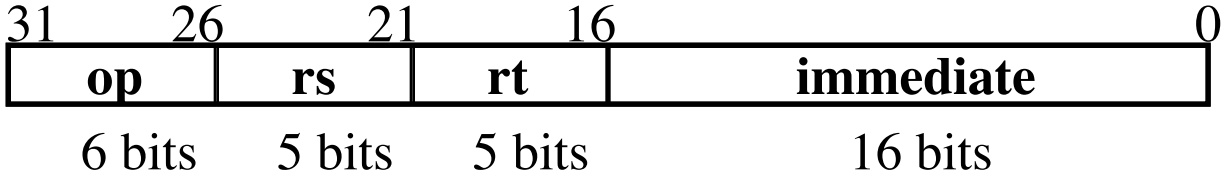
- Mem[R[rs] + SignExt[imm16]] = R[rt]

  Ex.: `sw rt, rs, imm16`





3f: The Branch Instruction



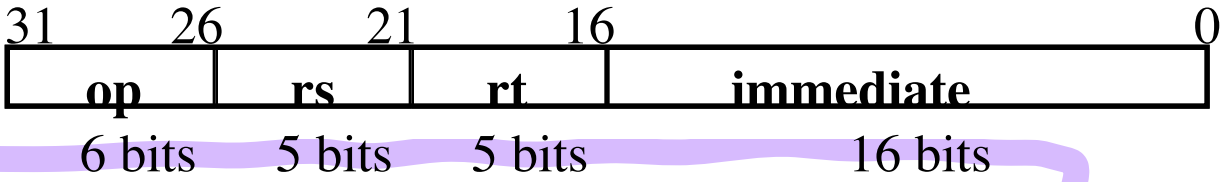
`beq rs, rt, imm16`

- 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 + (SignExt(imm16) x 4)

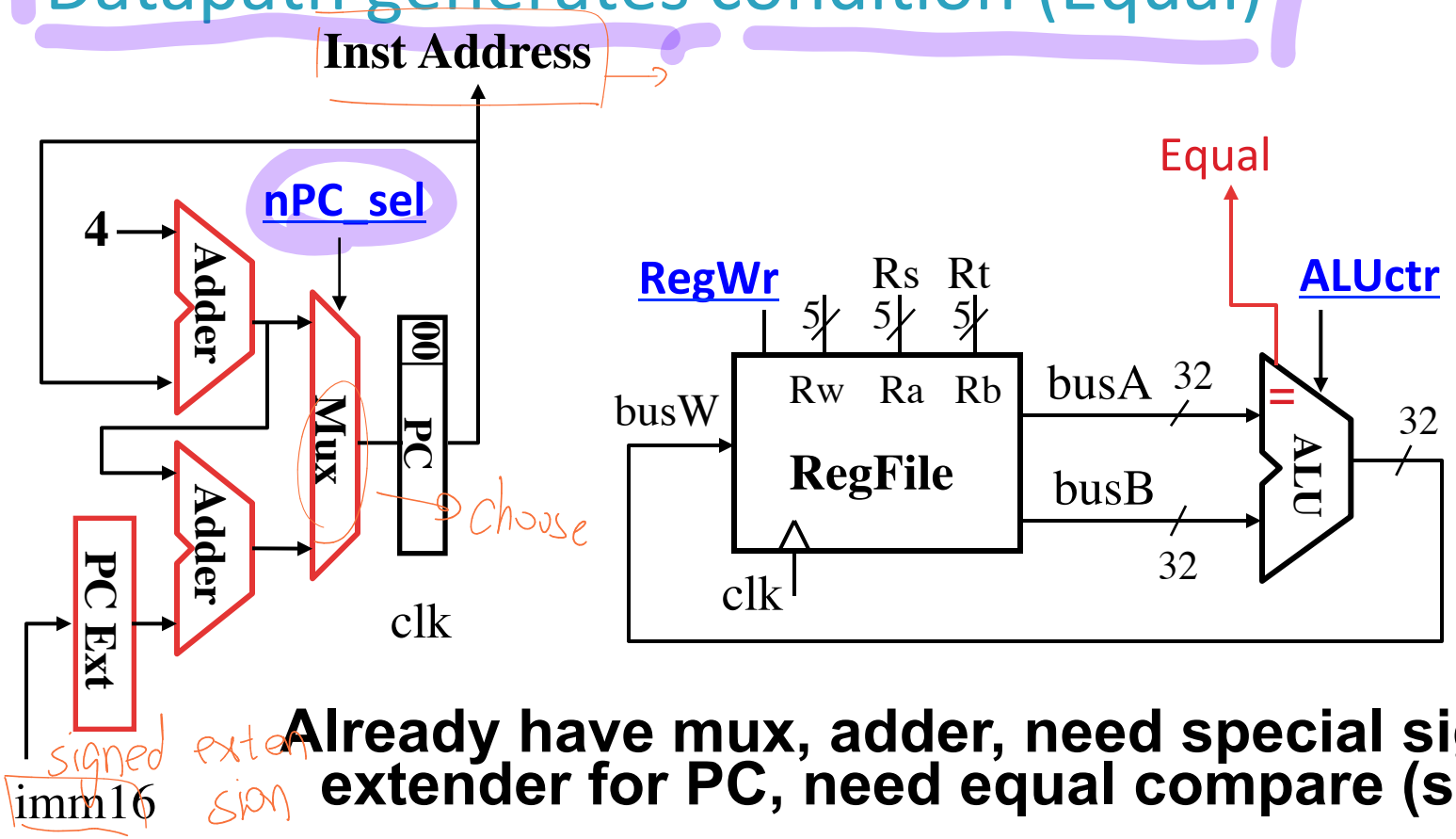
  else
  - PC = PC + 4

Datapath for Branch Operations

beq rs, rt, imm16

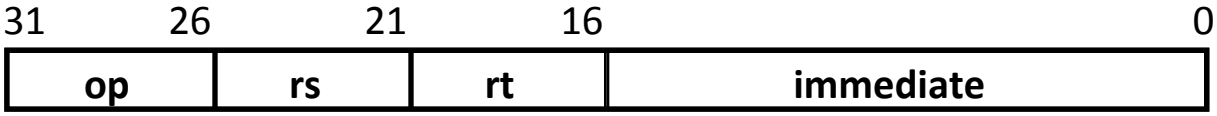


Datapath generates condition (Equal)

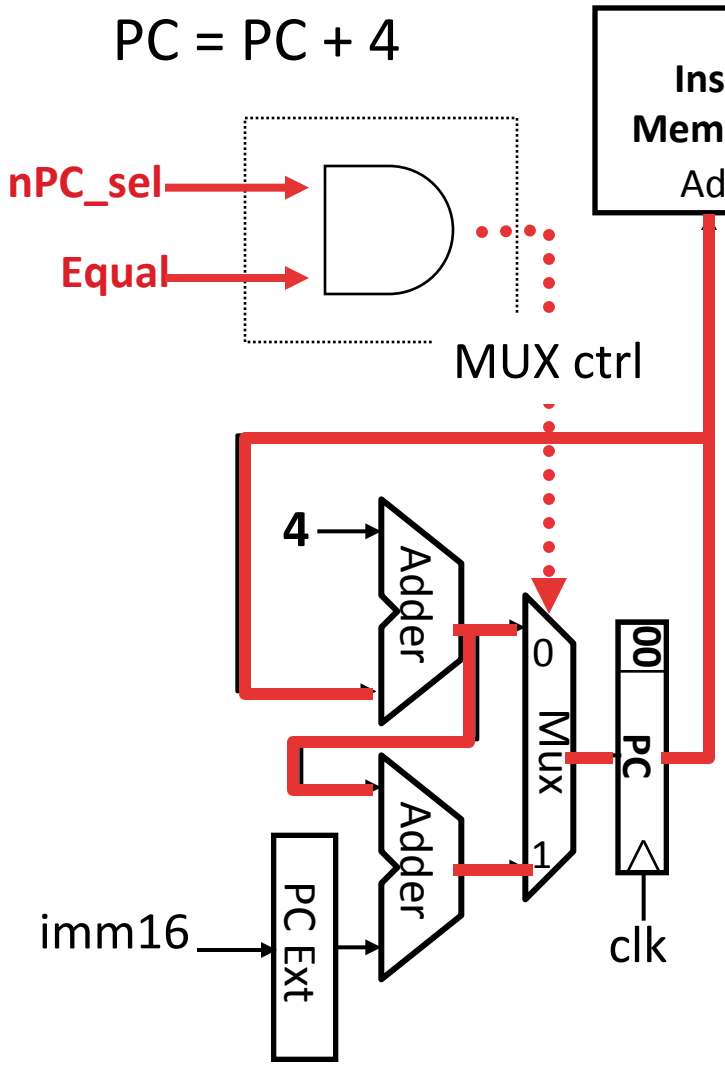


Already have mux, adder, need special sign extender for PC, need equal compare (sub?)

Instruction Fetch Unit including Branch



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

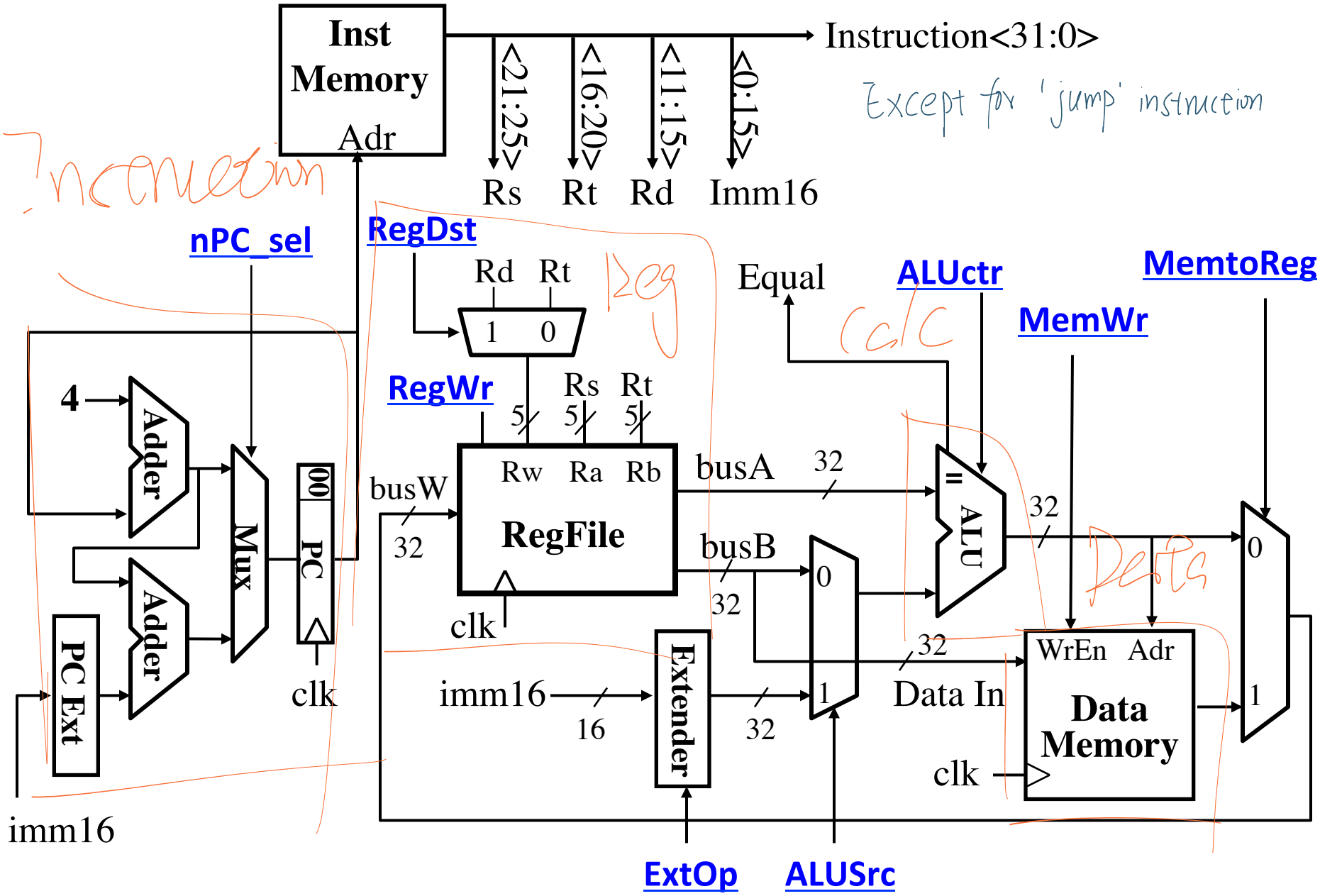


- How to encode nPC_sel?
  - Direct MUX select?
  - Branch inst. / not branch inst.
- Let's pick 2nd option

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

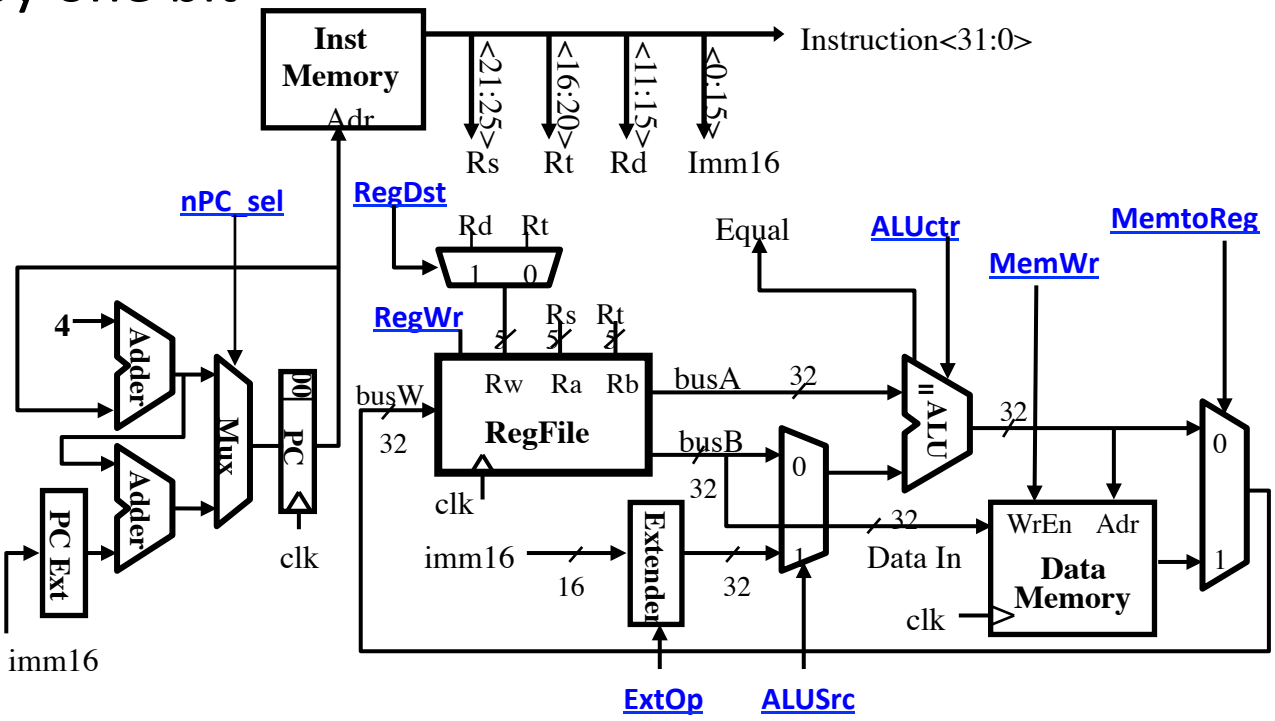


Clickers/Peer Instruction

What new (pseudo)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 one bit

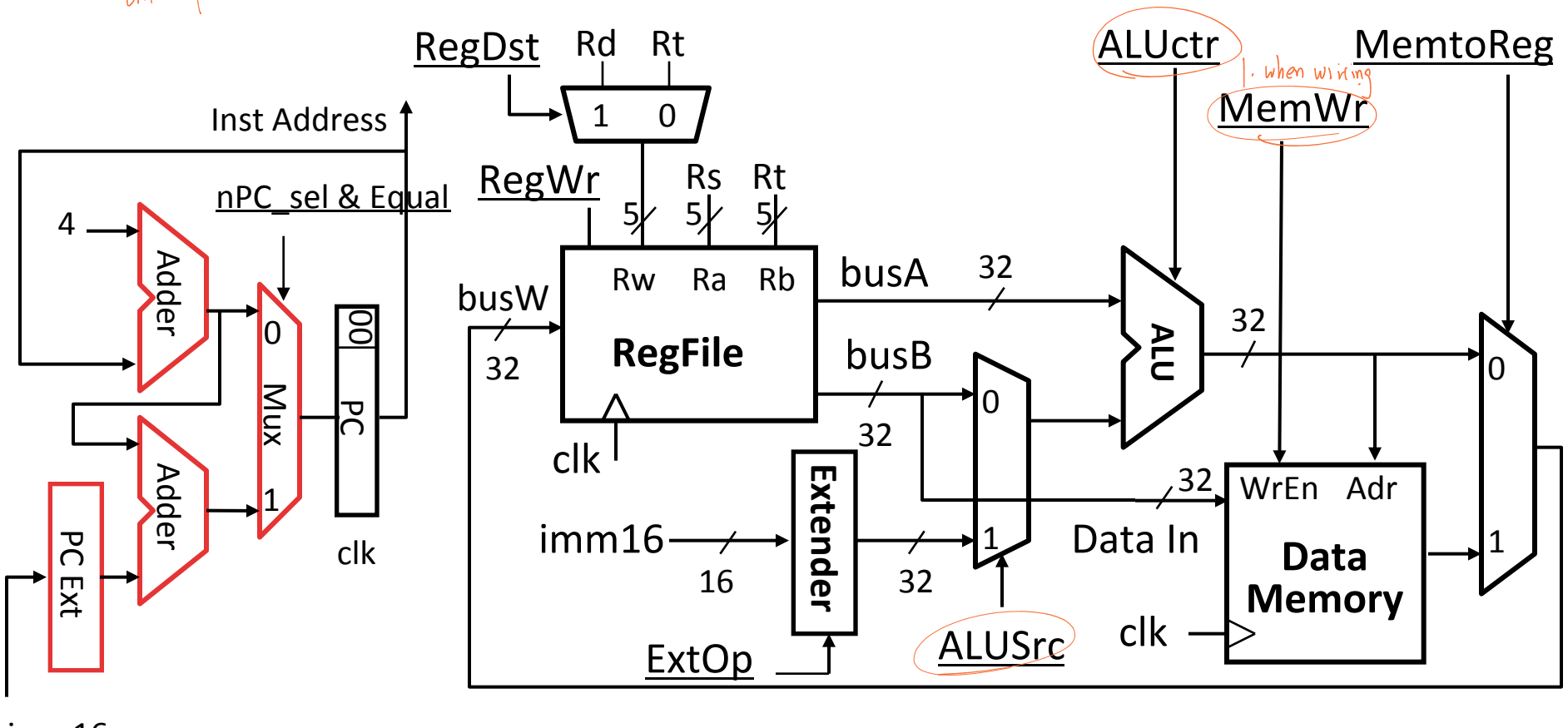
Closest / ALU have the capability to finish comparison.



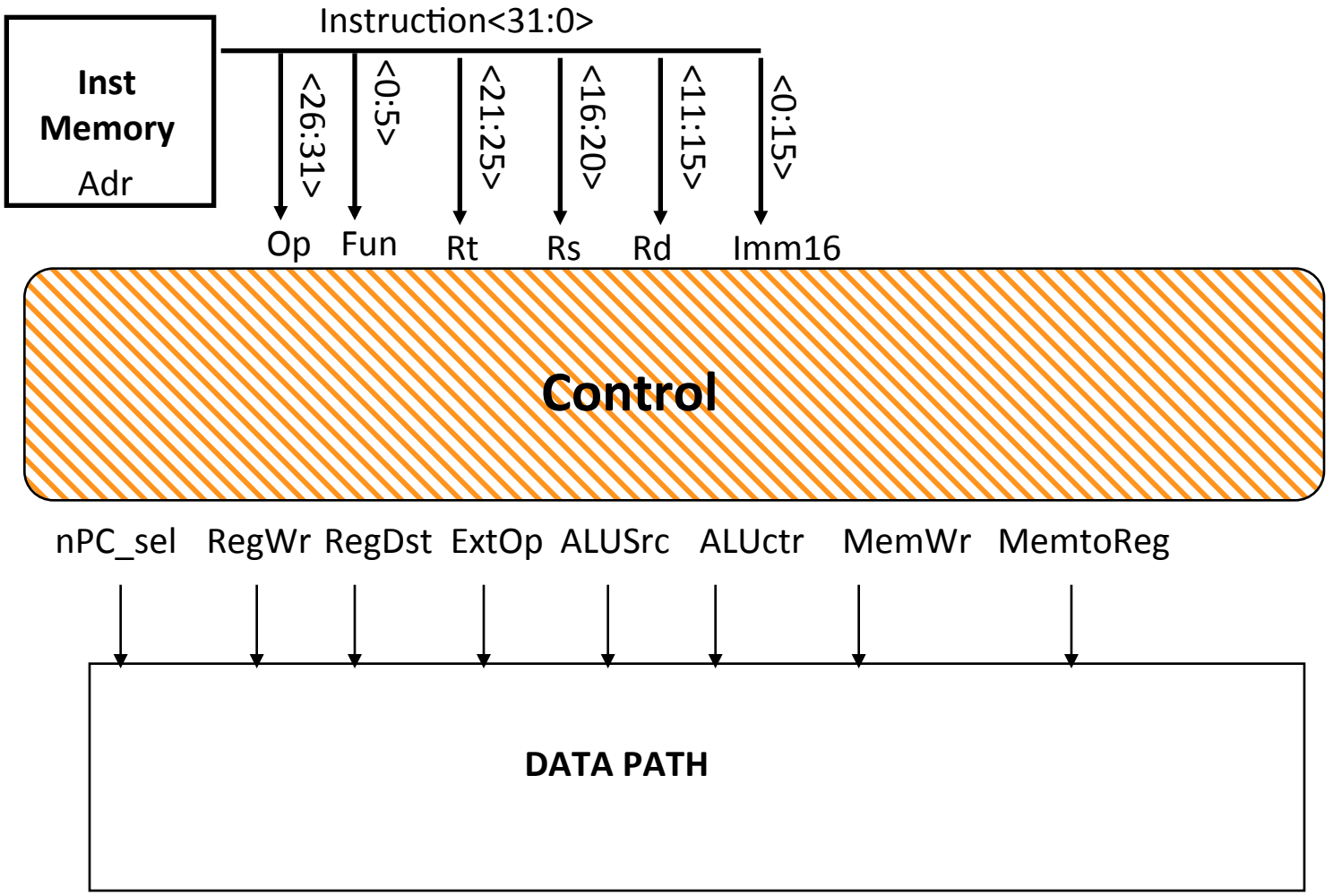
Administrivia

Datapath Control Signals

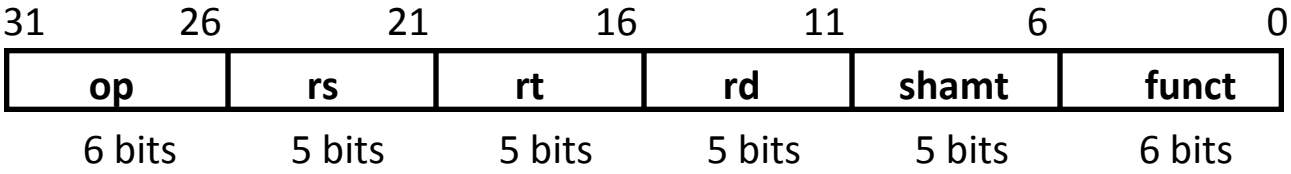
- ExtOp: "zero", "sign"
- ALUsrc: 0 ⇒ regB; 1 ⇒ immed
- ALUctr: "ADD", "SUB", "OR"
- MemWr: 1 ⇒ write memory
- MemtoReg: 0 ⇒ ALU; 1 ⇒ Mem
- RegDst: 0 ⇒ "rt"; 1 ⇒ "rd"
- RegWr: 1 ⇒ write register



Given Datapath: RTL → Control



RTL: The Add Instruction

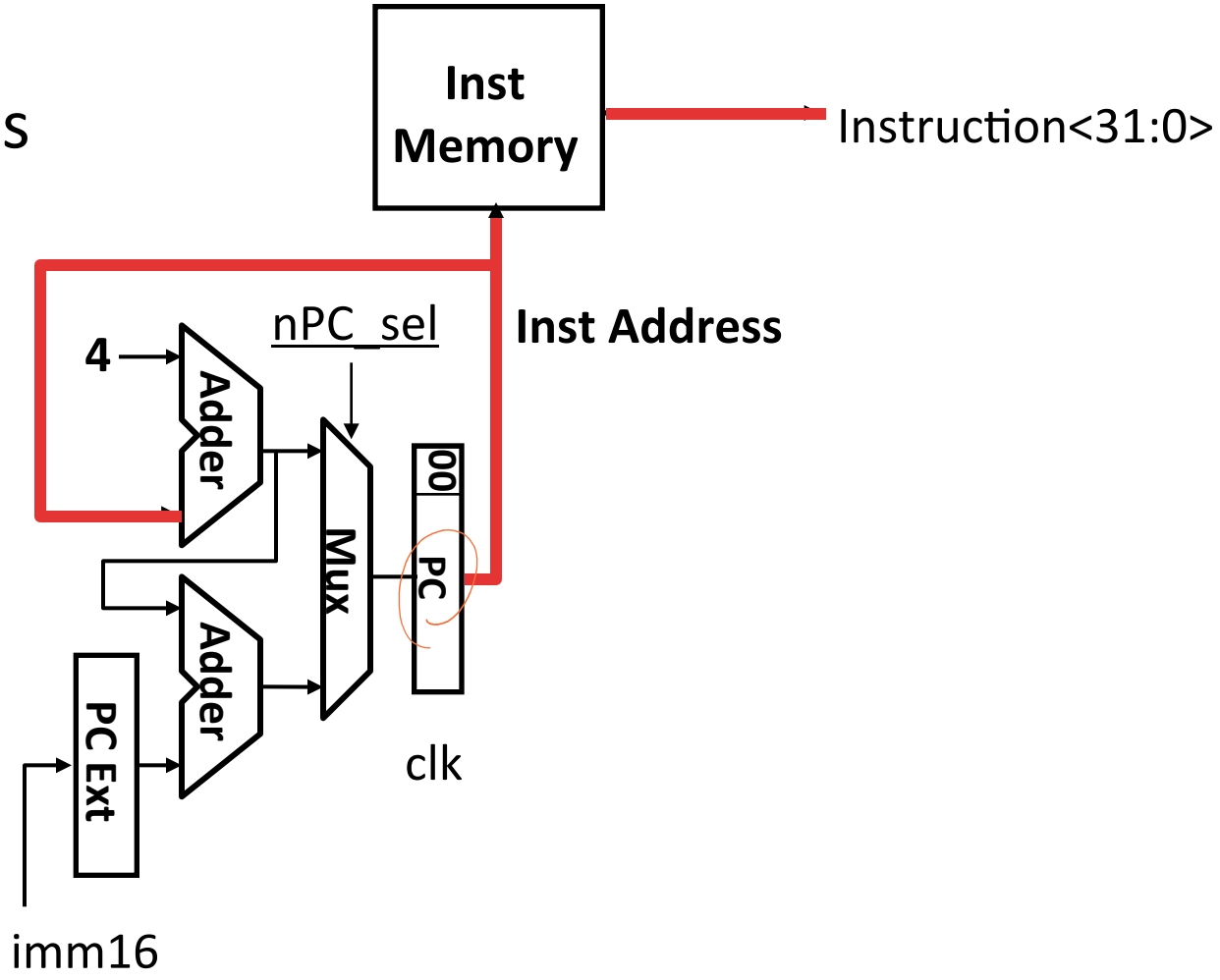


`add rd, rs, rt`

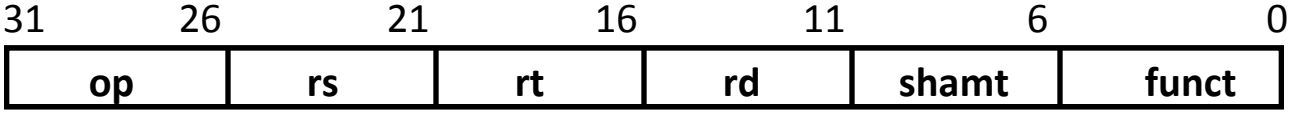
- MEM[PC] Fetch the instruction from memory
- R[rd] = R[rs] + R[rt] The actual operation
- PC = PC + 4 Calculate the next instruction's address

Instruction Fetch Unit at the Beginning of Add

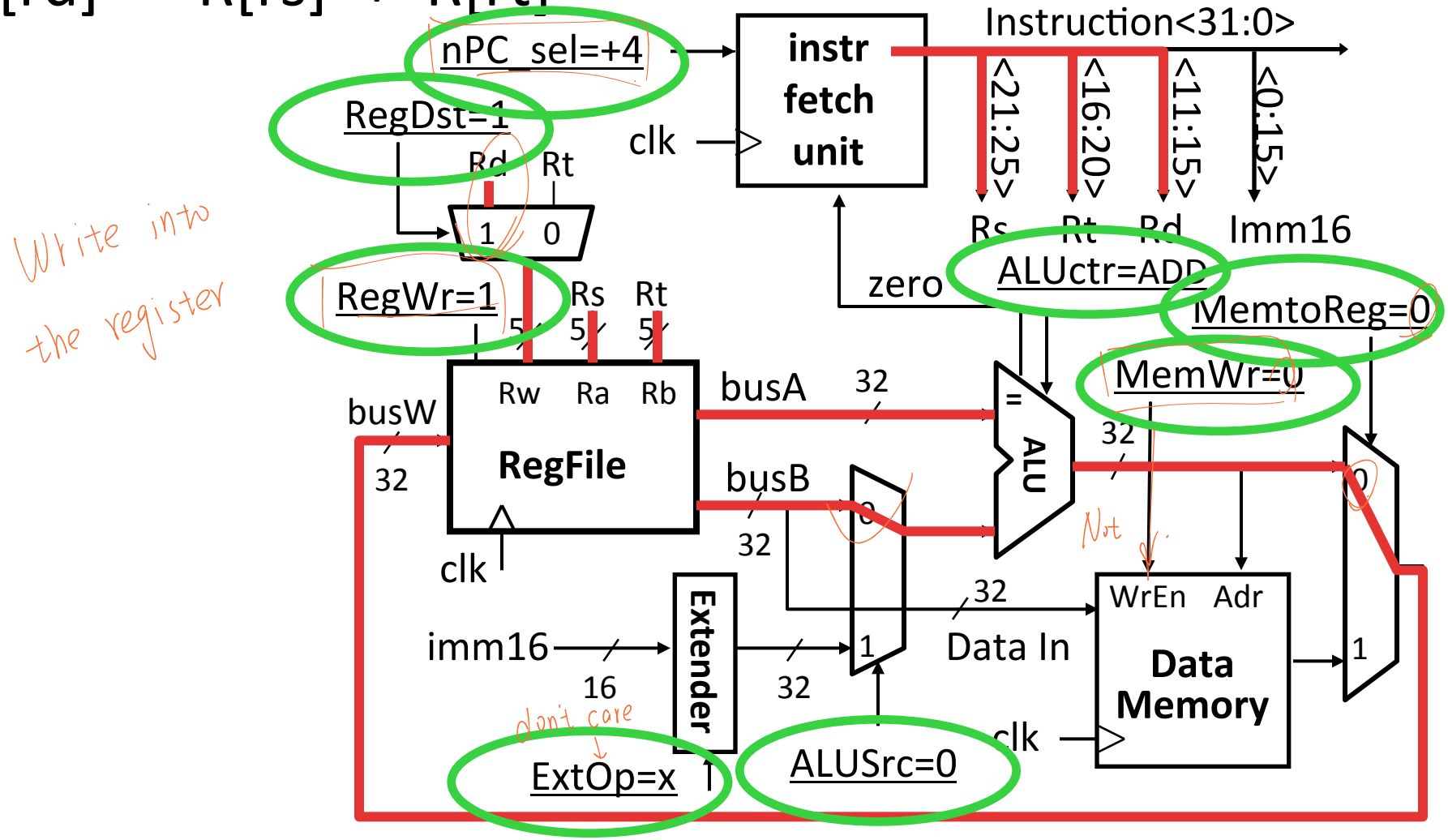
- Fetch the instruction from Instruction memory: Instruction = MEM[PC]
  - same for all instructions



Single Cycle Datapath during Add

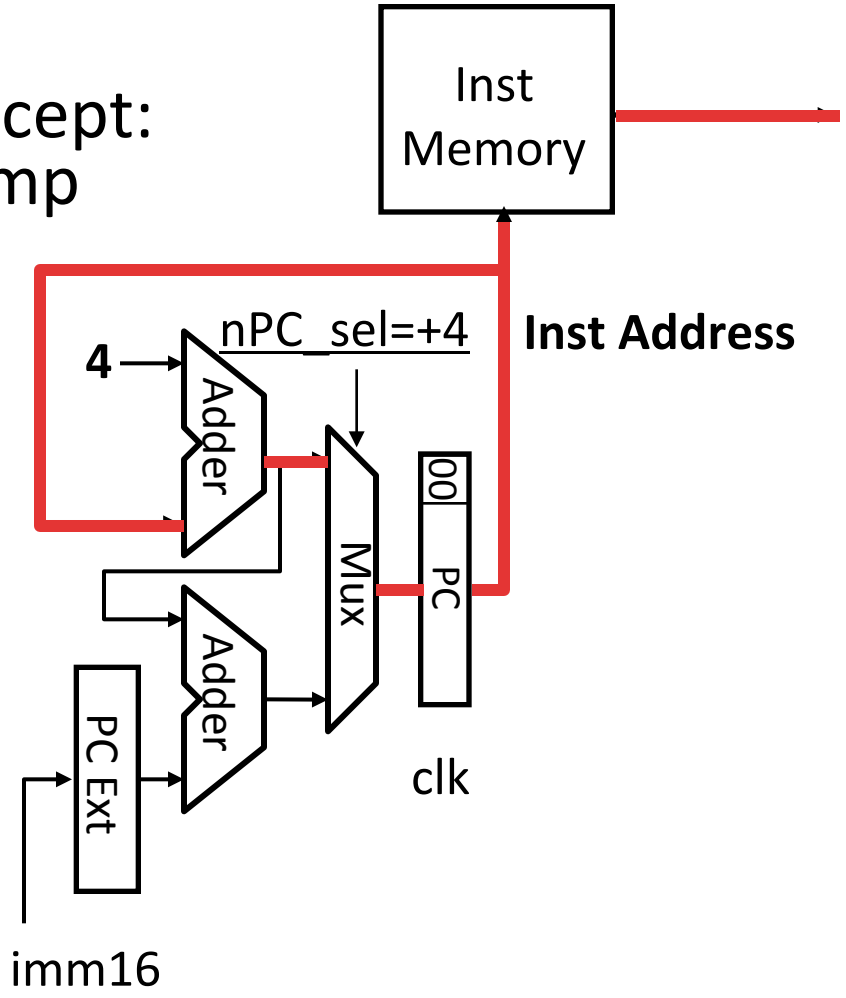


R[rd] = R[rs] + R[rt]



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 }

Jump=
nPC_sel=
Instruction Fetch Unit
Clk
Instruction<31:0>
<21:25>
<16:20>
<11:15>
<0:15>
<0:25>
Rt
Rs
Rd
Imm16
TA26
RegDst =
Rd
Rt
1 Mux 0
RegWr =
5
5
5
Rs
Rt
ALUctr =
Zero
MemWr =
MemtoReg =
busA
32
Rw Ra Rb
32 x 32-bit Registers
busW
32
Clk
busB
32
0 Mux 1
ALU
32
0 Mux 1
32
WrEn Adr
Data In
32
Data Memory
Clk
imm16
16
Extender
32
ALUSrc =
ExtOp =

Single Cycle Datapath during Jump

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

Nothing will be created

Jump=1

nPC_sel=?

Instruction Fetch Unit

Clk

Instruction<31:0>

<21:25>

<16:20>

<11:15>

<0:15>

<0:25>

Rt Rs Rd Imm16 TA26

RegDst = x

Rd Rt

1 Mux 0

RegWr = 0

5 5 5

Rs Rt

ALUctr =x

MemtoReg = x

Zero

MemWr = 0

busA

32

Rw Ra Rb

32 x 32-bit Registers

busW

32

Clk

busB

32

0 Mux 1

ALU

32

0 Mux 1

32

WrEn Adr

Data In 32

Data Memory

Clk

imm16

16

Extender

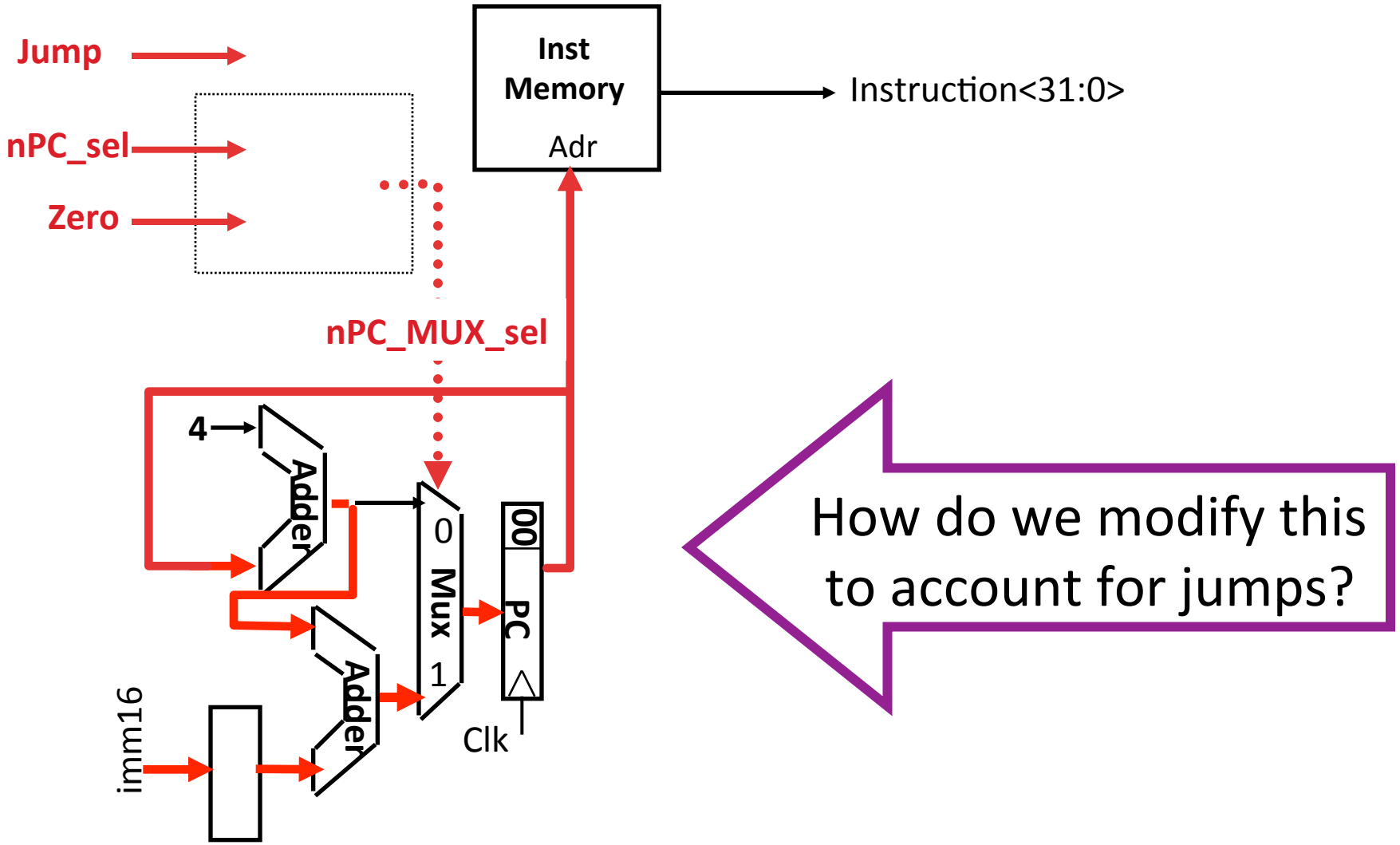
32

ALUSrc = x

ExtOp = x

Instruction Fetch Unit at the End of Jump

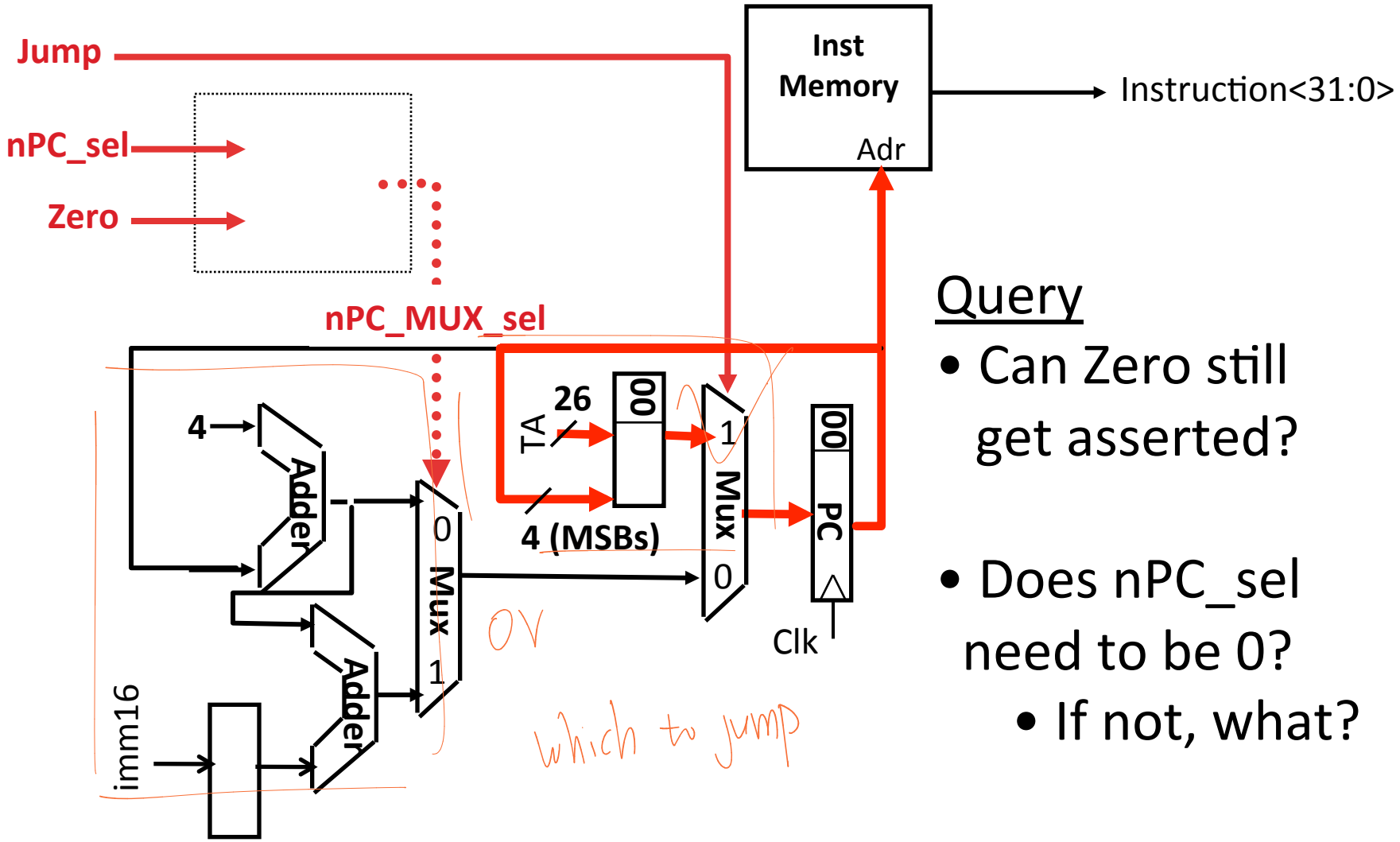
- New PC = { PC[31..28], target address, 00 }



How do we modify this to account for jumps?

Instruction Fetch Unit at the End of Jump

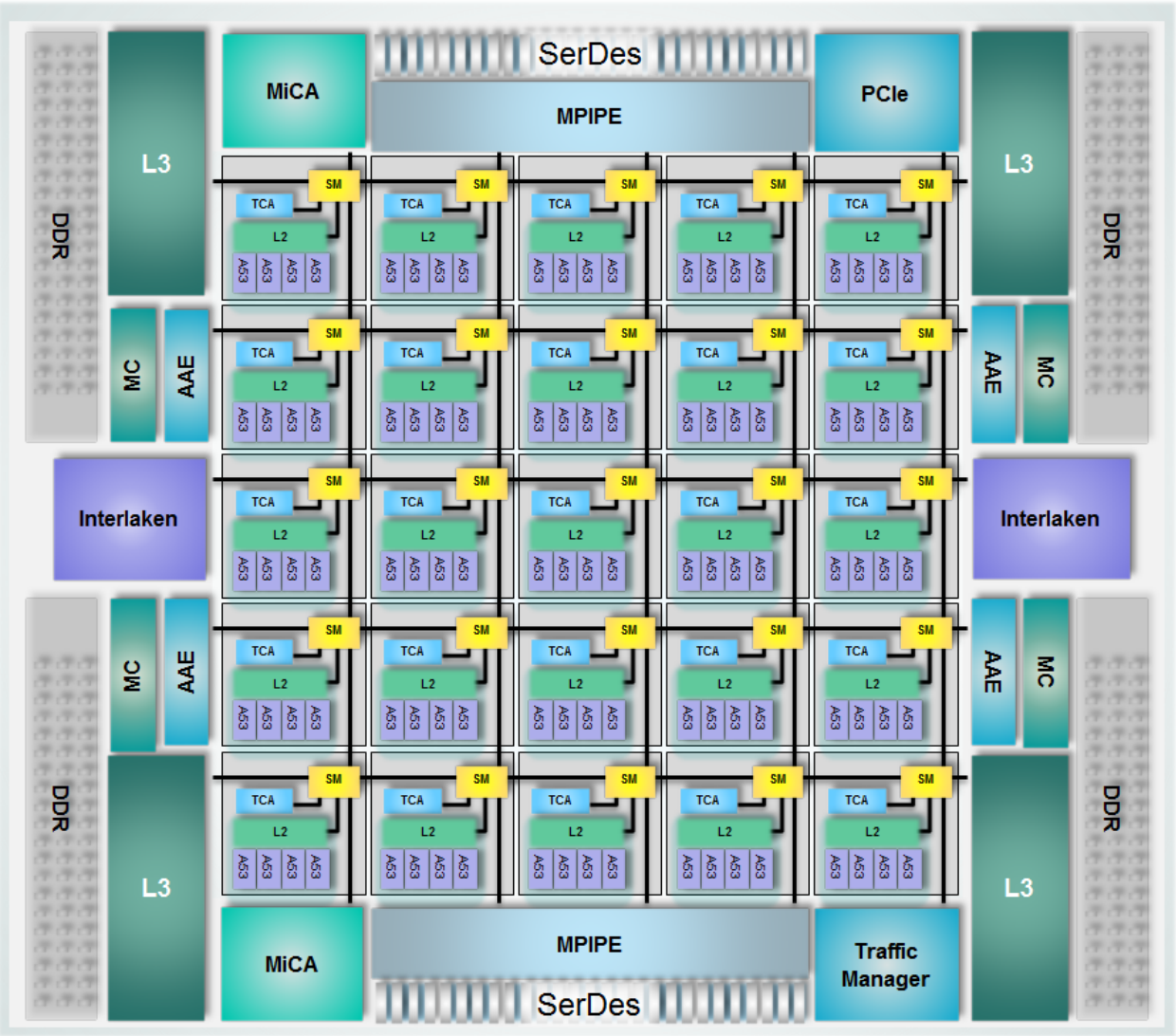
- New PC = { PC[31..28], target address, 00 }



Query

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

In The News: Tile-Mx100
100 64-bit ARM cores on one chip

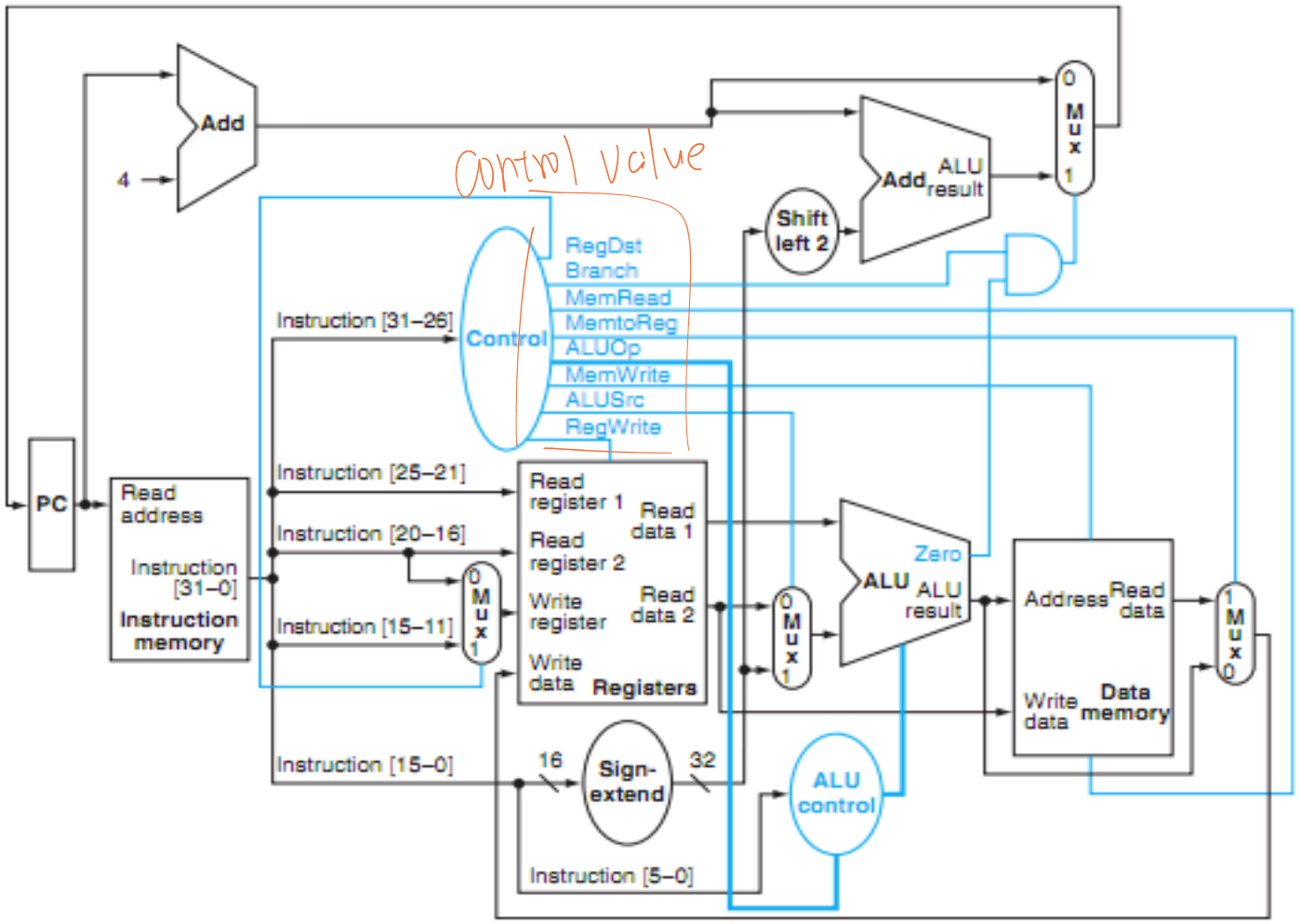


EZChip (bought Tilera)

100 64-bit ARM Cortex A53

- Dual-issue, in-order

P&H Figure 4.17



Summary of the Control Signals (1/2)

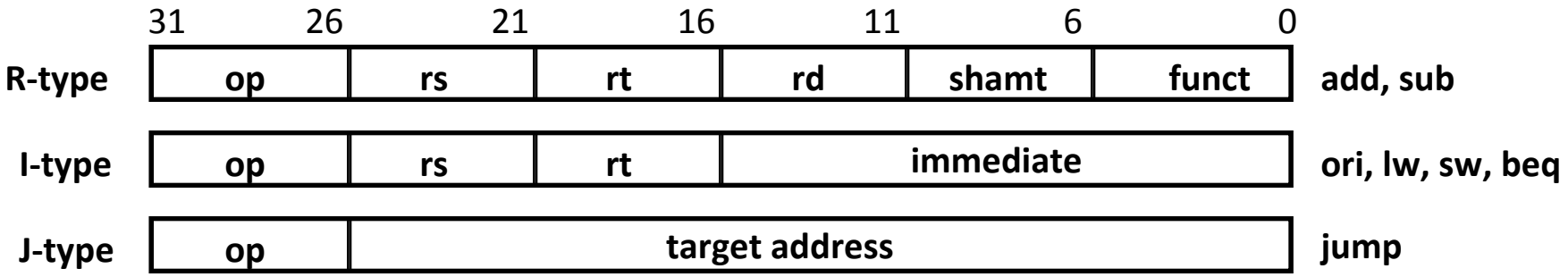
inst	Register Transfer
add	R[rd] ← R[rs] + R[rt]; PC ← PC + 4
	ALUsrc=RegB, ALUctr="ADD", RegDst=rd, RegWr, nPC_sel="+4"
sub	R[rd] ← R[rs] – R[rt]; PC ← PC + 4
	ALUsrc=RegB, ALUctr="SUB", RegDst=rd, RegWr, nPC_sel="+4"
ori	R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4
	ALUsrc=Im, Extop="Z", ALUctr="OR", 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)

Truth table

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	?
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
```

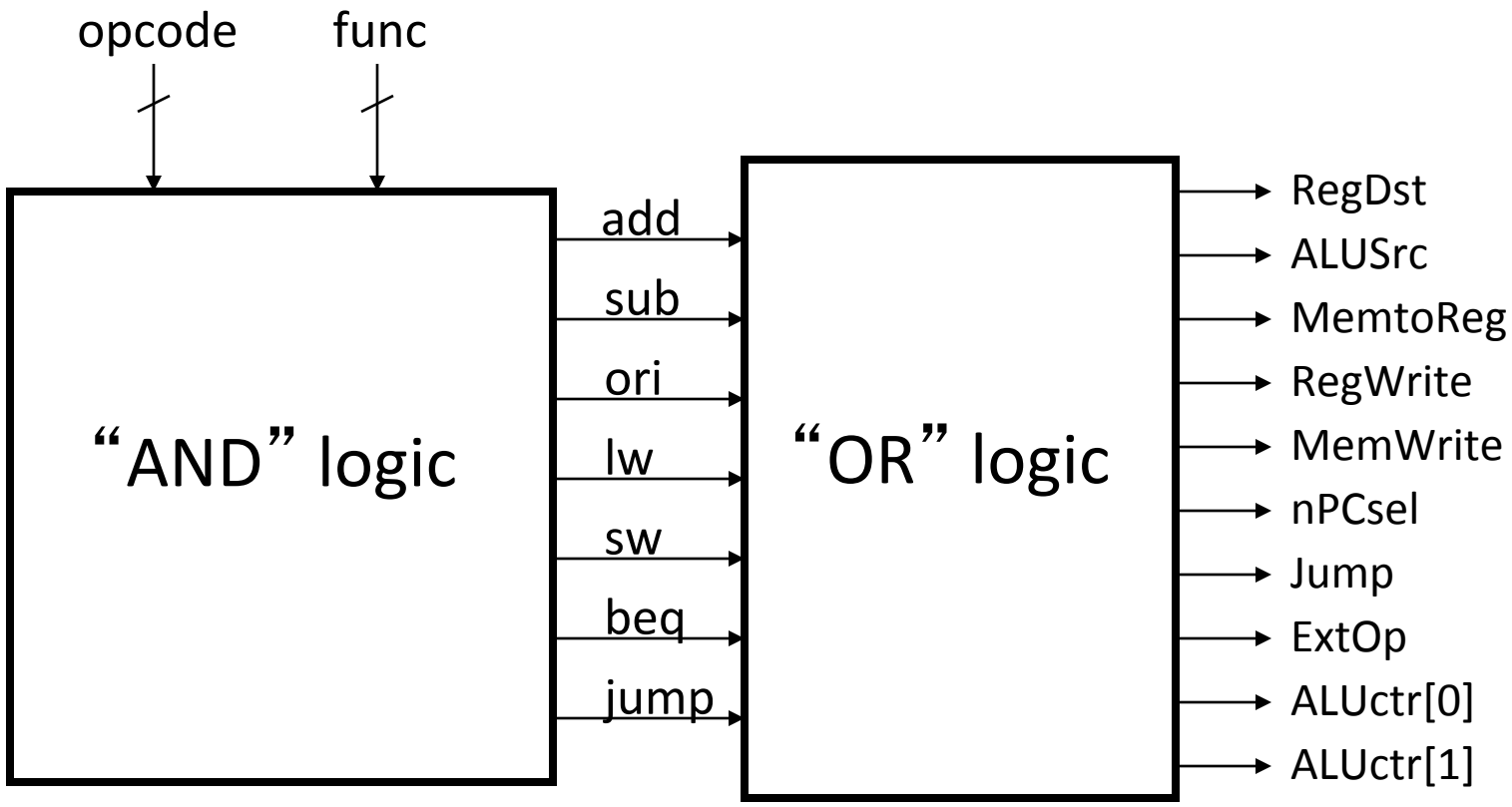
Where:

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

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

How do we implement this in gates?

Controller Implementation



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