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

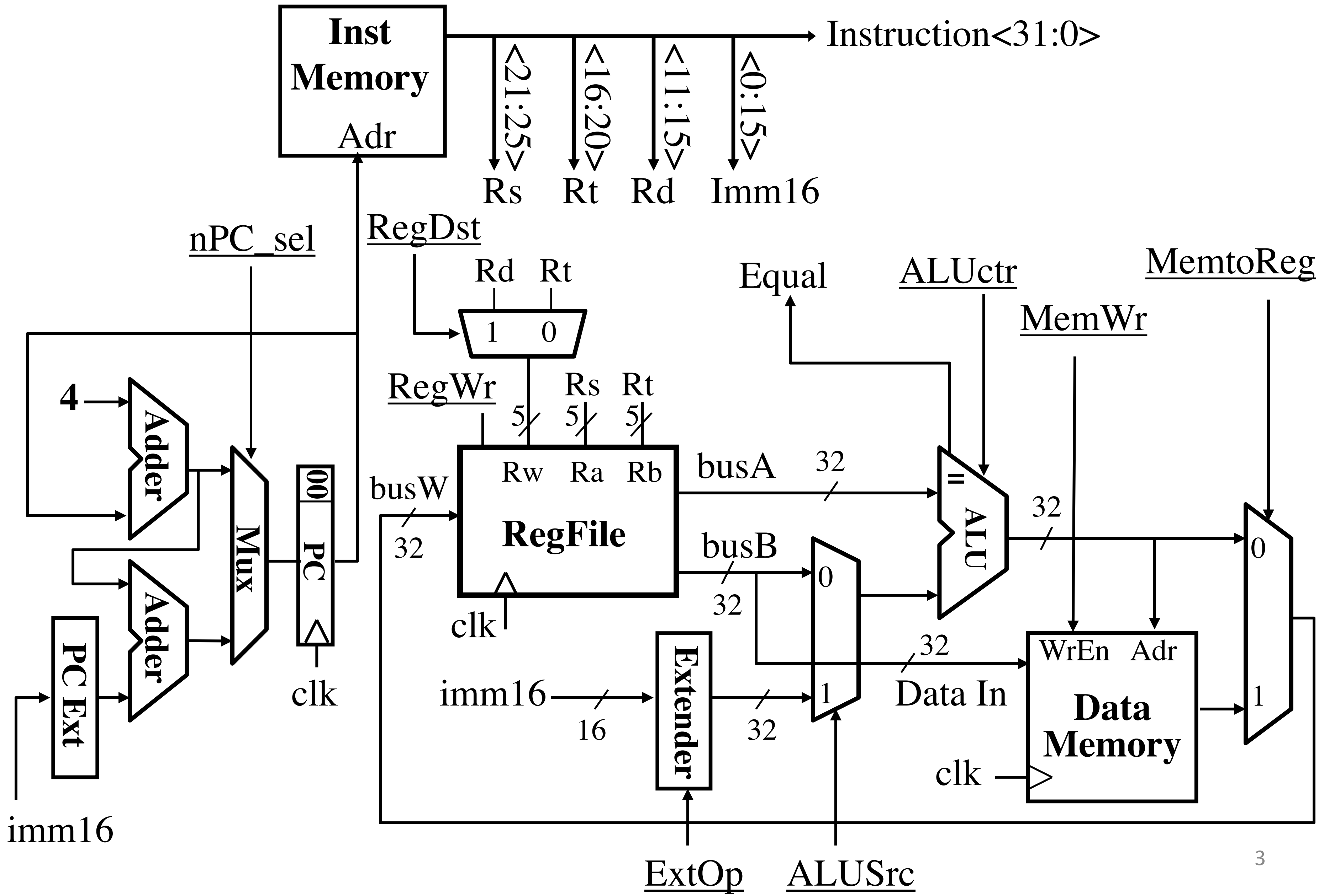
Step 2: Select set of datapath components & establish clock methodology

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

A Single Cycle Datapath

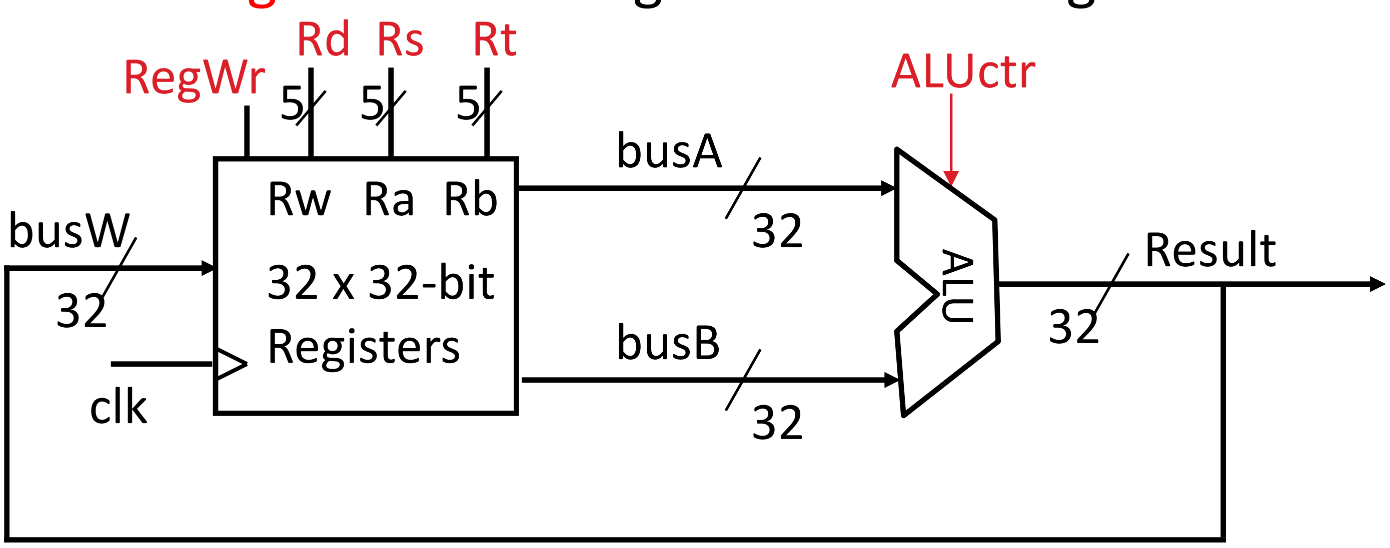


Step 3b: Add & Subtract

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

31 — 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

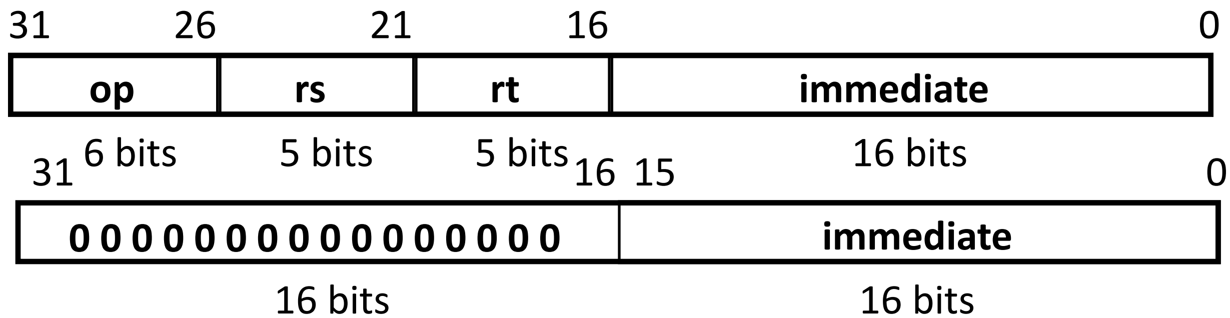
  - ALUctr and RegWr: control logic after decoding the instruction



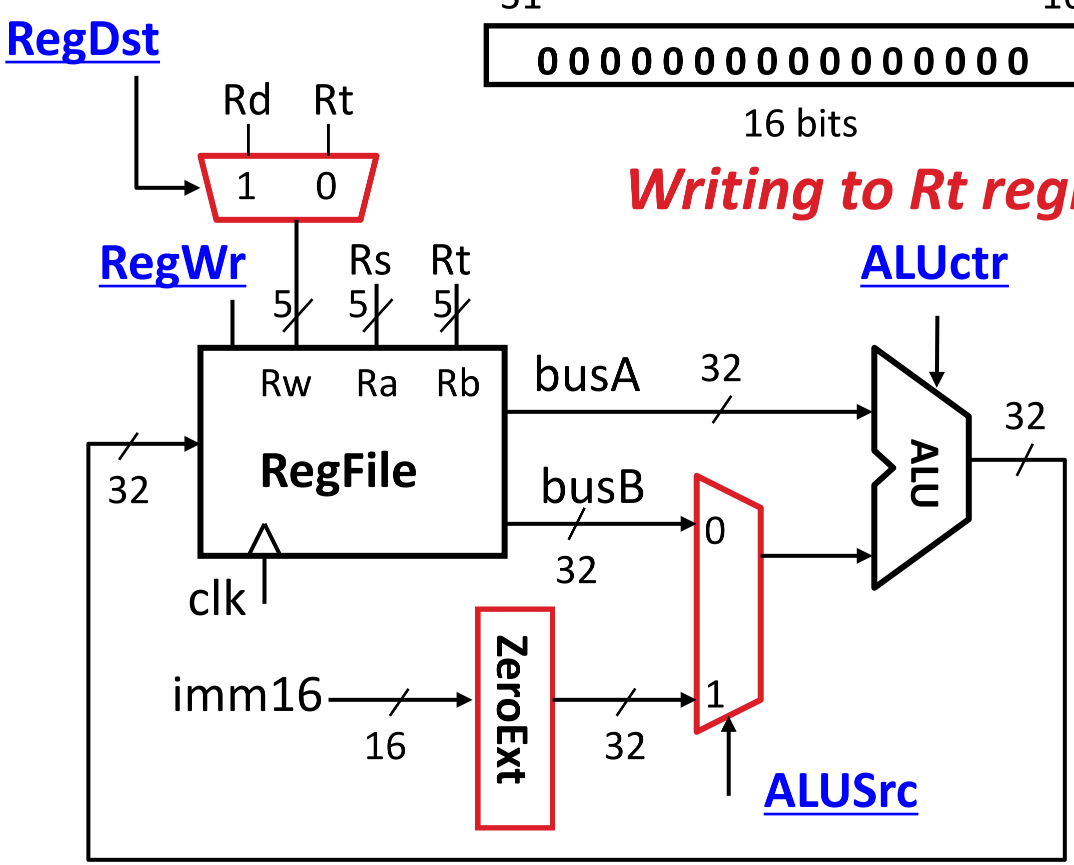
- … Already defined the register file & ALU

3c: Logical Op (or) with Immediate

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



Writing to Rt register (not Rd)!!

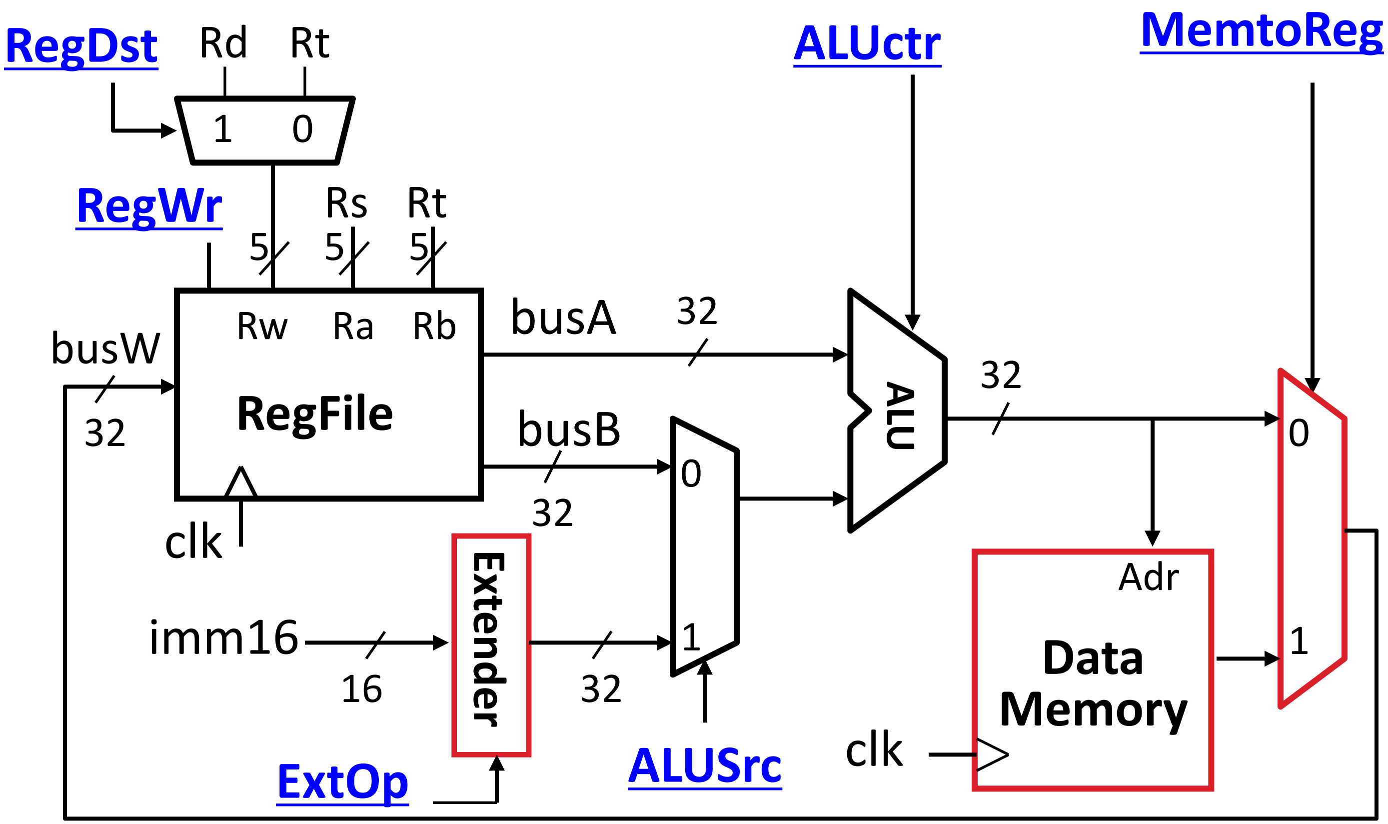


What about Rt Read?

3d: Load Operations

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

31 op 26	rs 21	rt 16	immediate 0
6 bits	5 bits	5 bits	16 bits

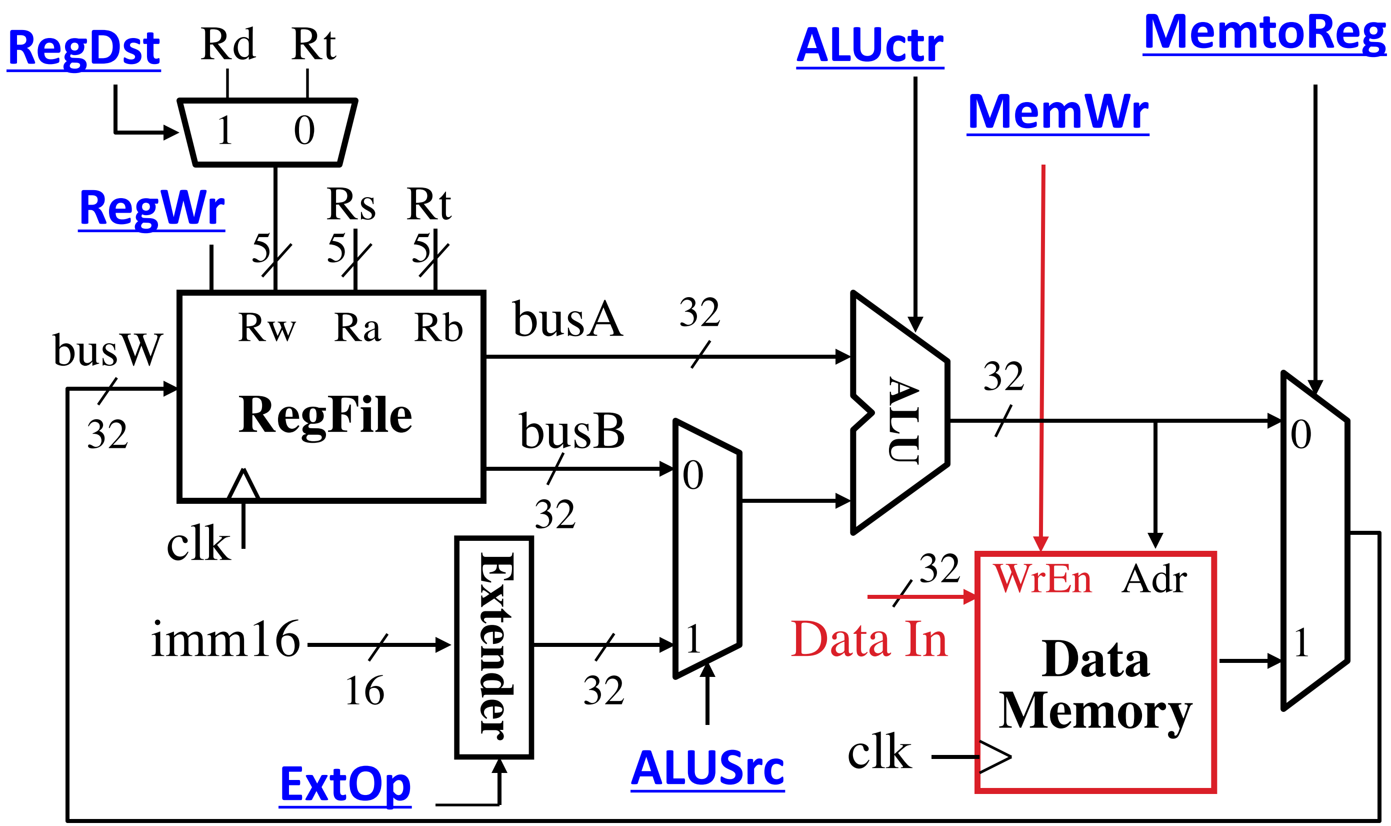


3e: Store Operations

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

  Ex.: `sw rt, rs, imm16`

31–26	25–21	20–16	15–0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

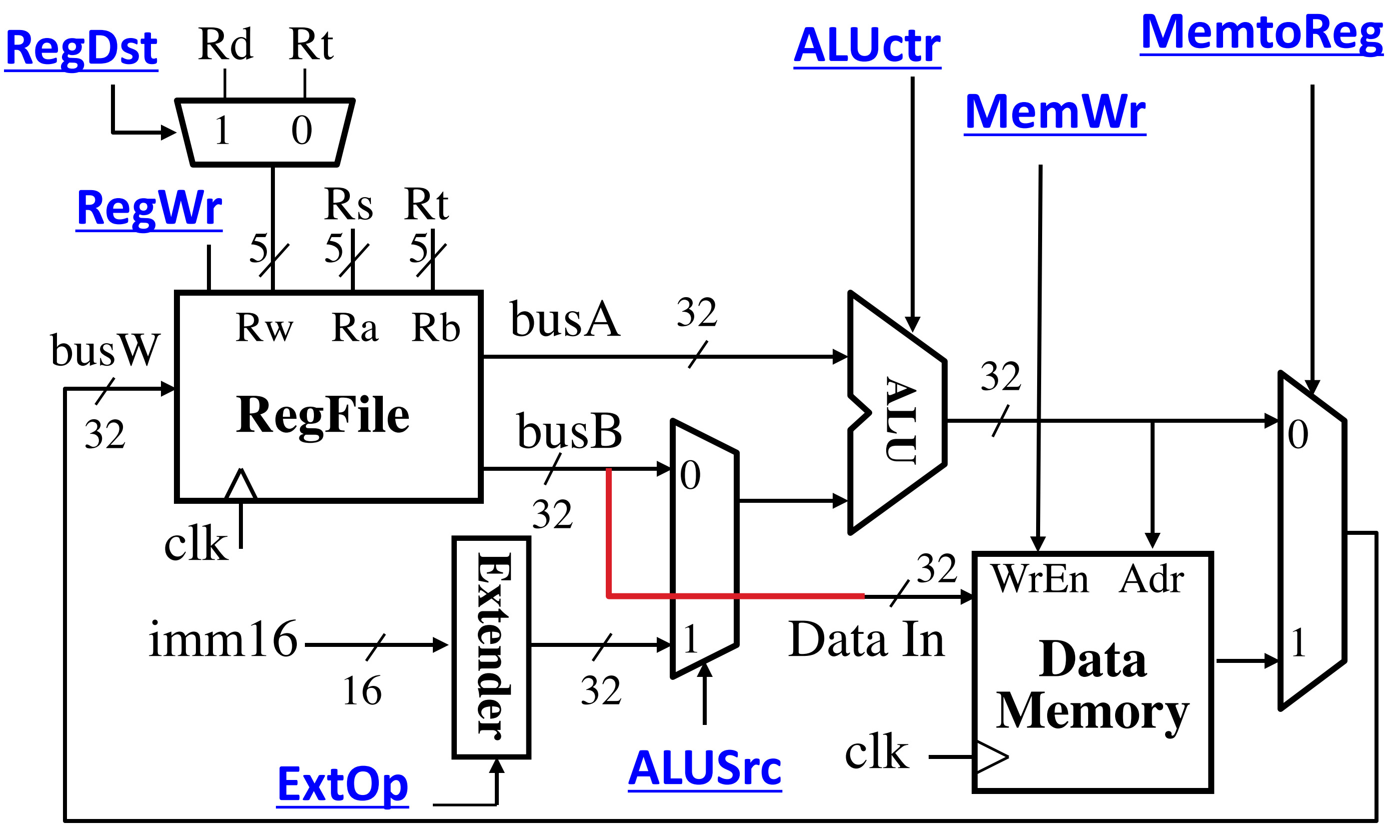


3e: Store Operations

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

  Ex.: `sw rt, rs, imm16`

31–26	26–21	21–16	16–0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits



3f: The Branch Instruction

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

`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

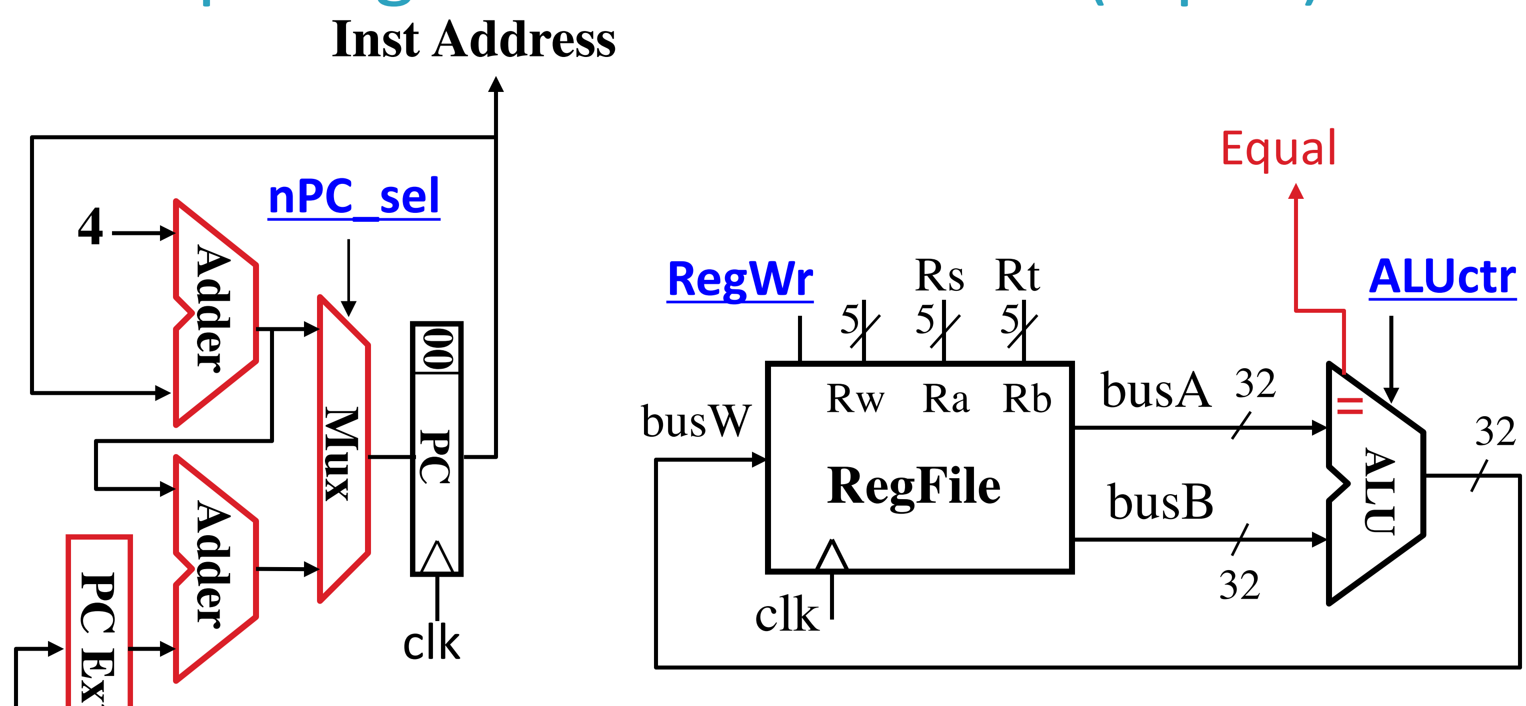
- PC = PC + 4

Datapath for Branch Operations

beq rs, rt, imm16

31–26	25–21	20–16	15–0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

Datapath generates condition (Equal)

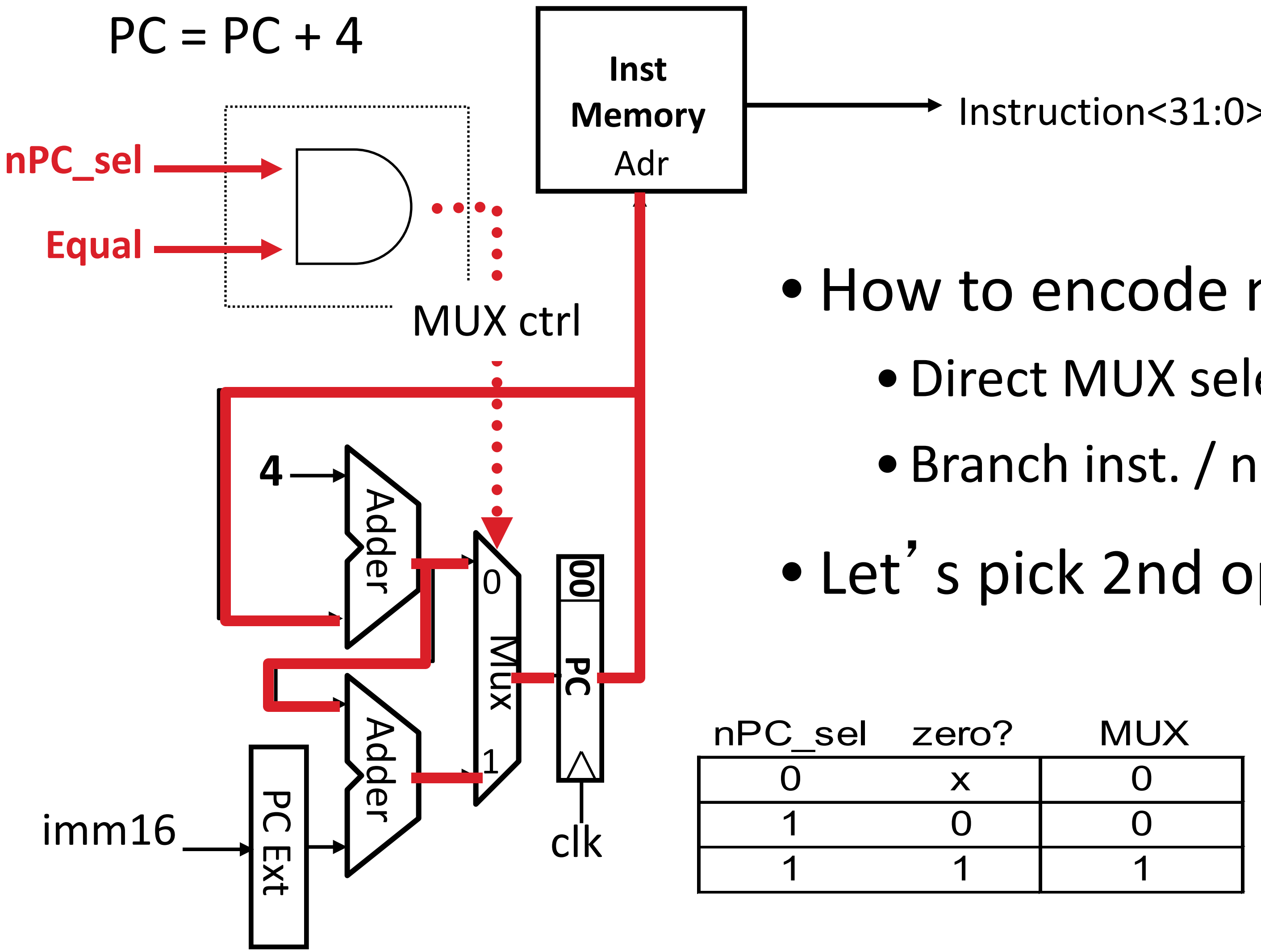


Already have mux, adder, need special sign extender for PC, need equal compare (sub?)

Instruction Fetch Unit including Branch

31	26	21	16	0
op	**rs**	**rt**	**immediate**	

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

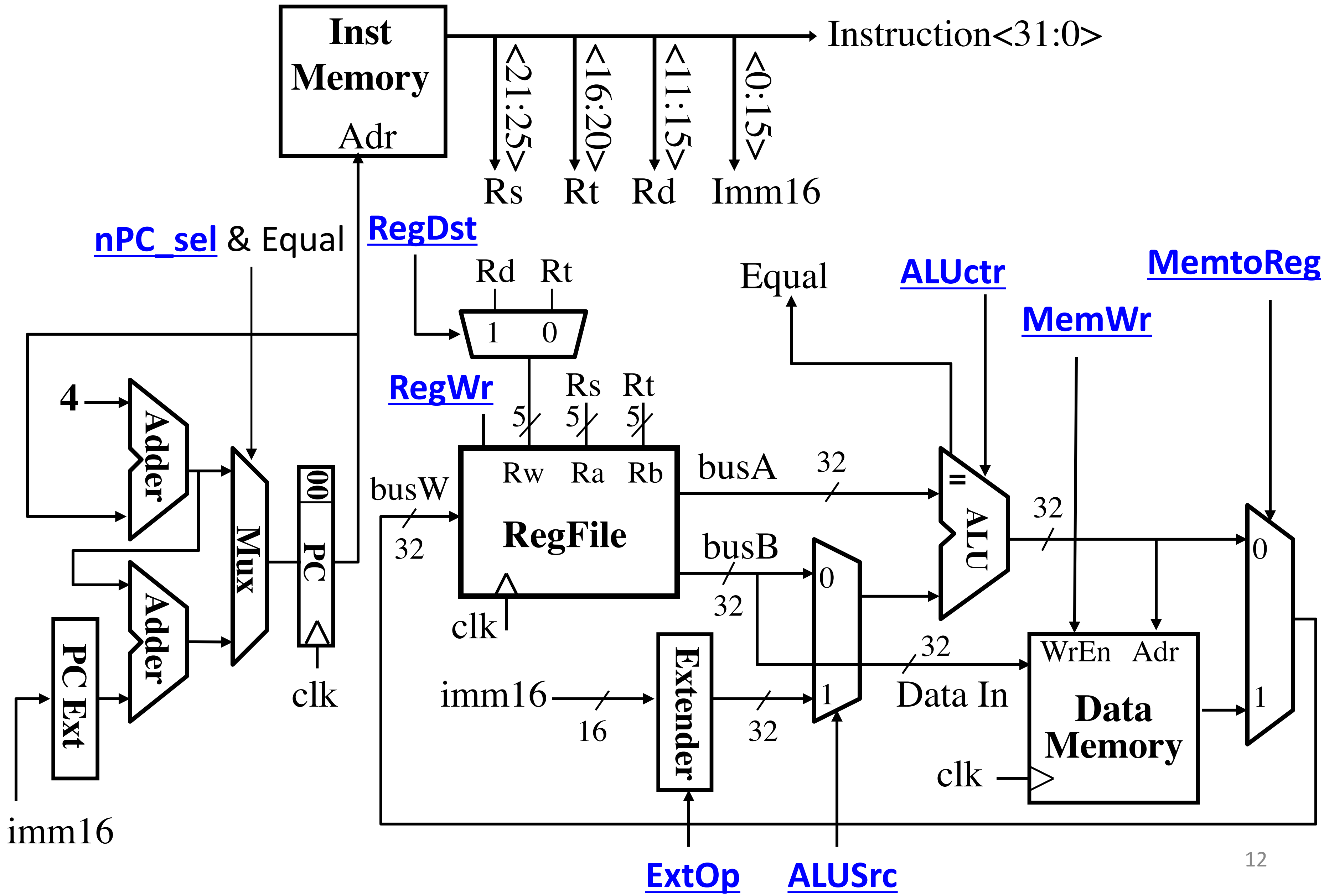


- How to encode nPC_sel?
  - Direct MUX select?
  - Branch inst. / not branch inst.
- Let's pick 2nd option

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

Q: What logic gate?

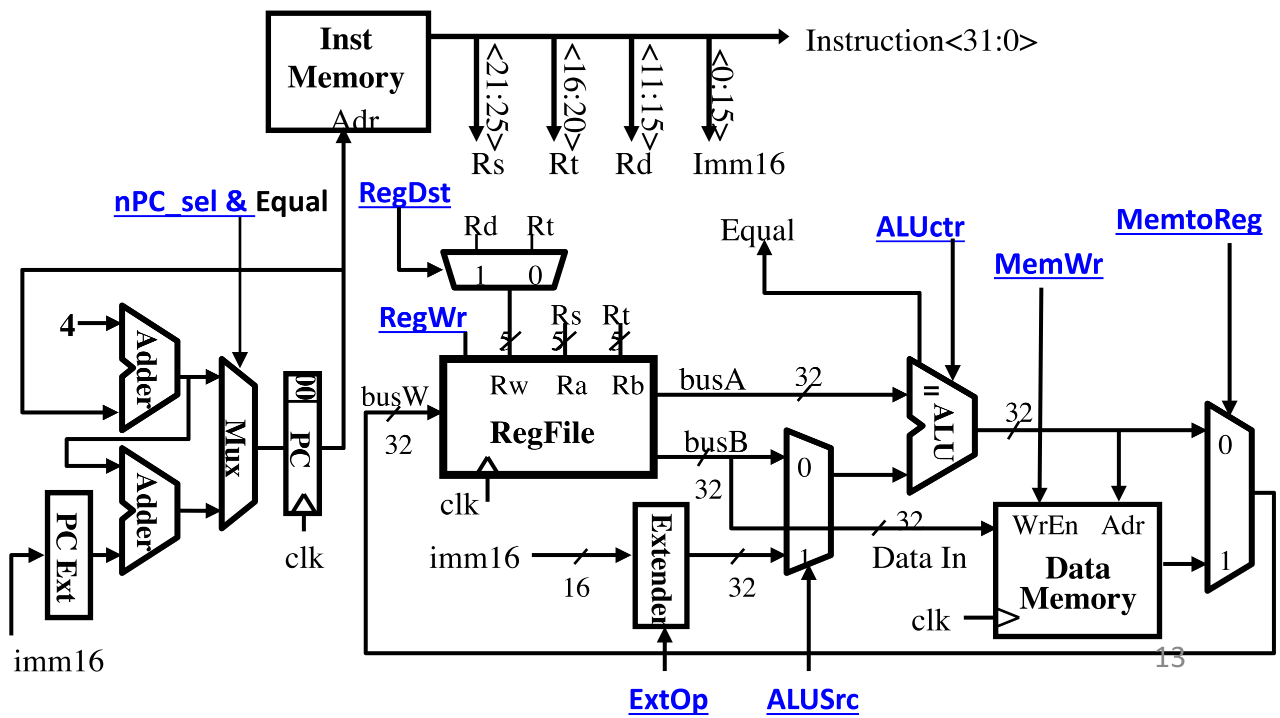
Putting it All Together:A Single Cycle Datapath



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



Admin

- Updated the score for HW 1 on gradebot for those that didn't properly comment their code.
- We will also check Project 1.1 and Project 1.2 for proper comments!

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

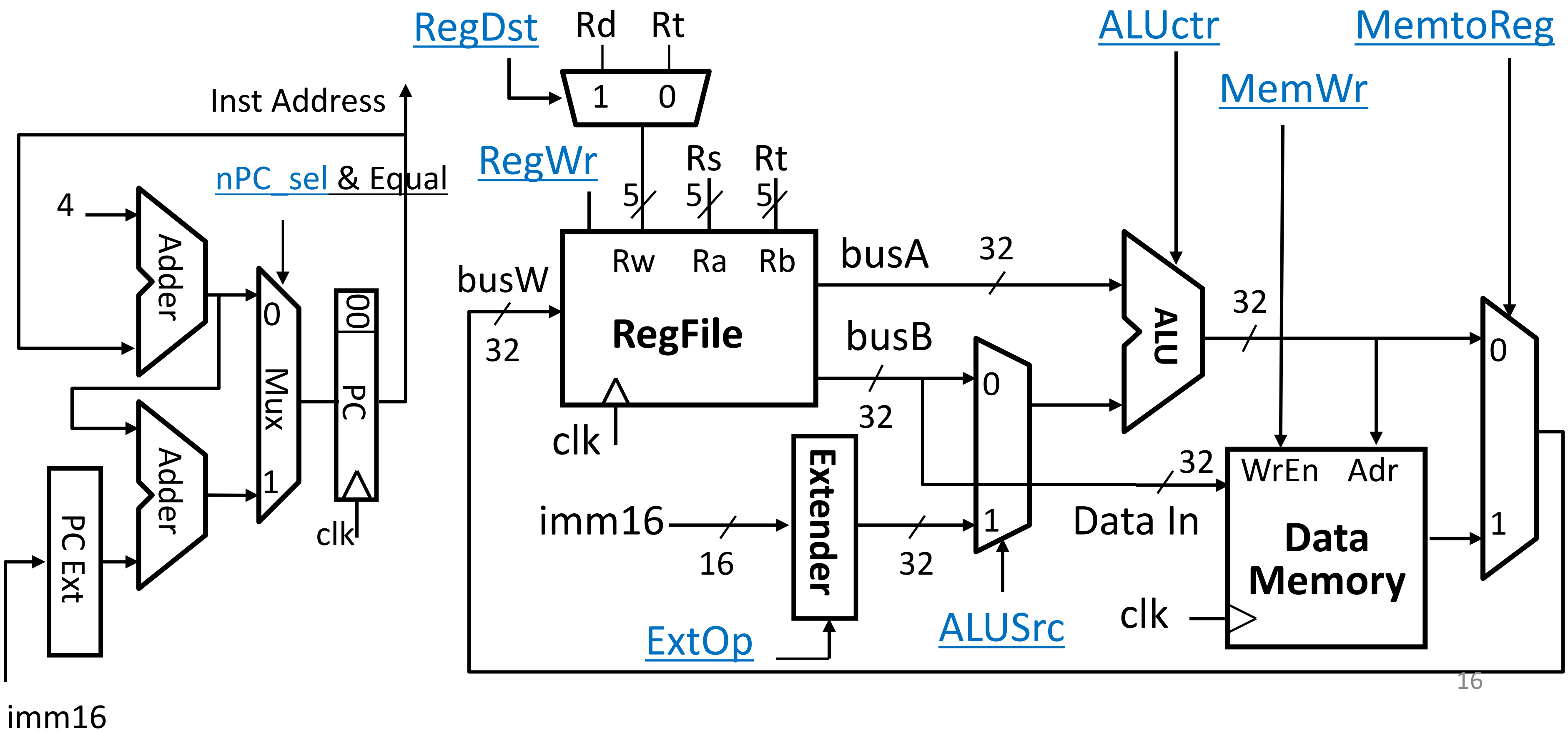
Step 3: Assemble datapath components that meet the requirements

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

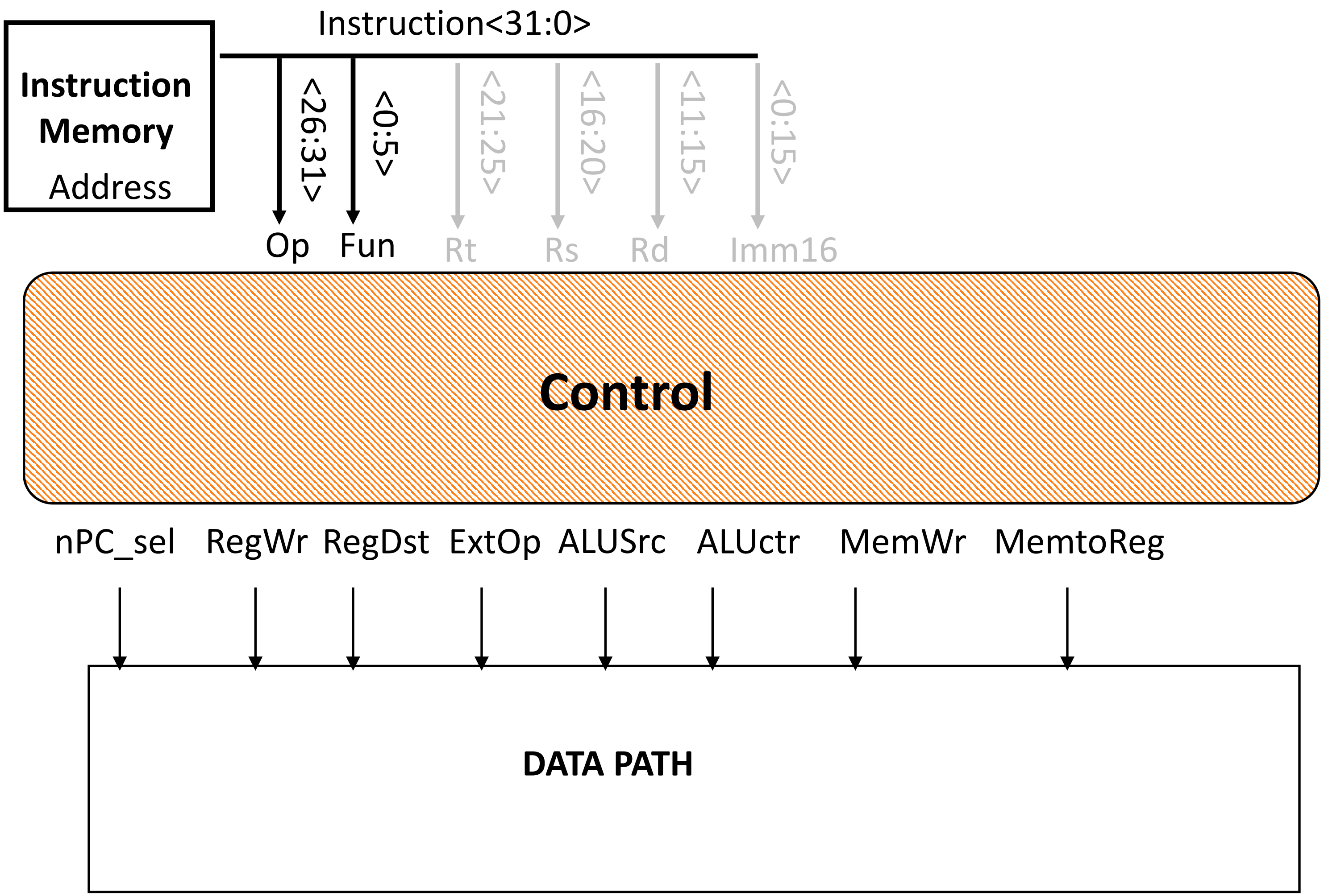
Step 5: Assemble the control logic

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 → Control



RTL: The Add Instruction

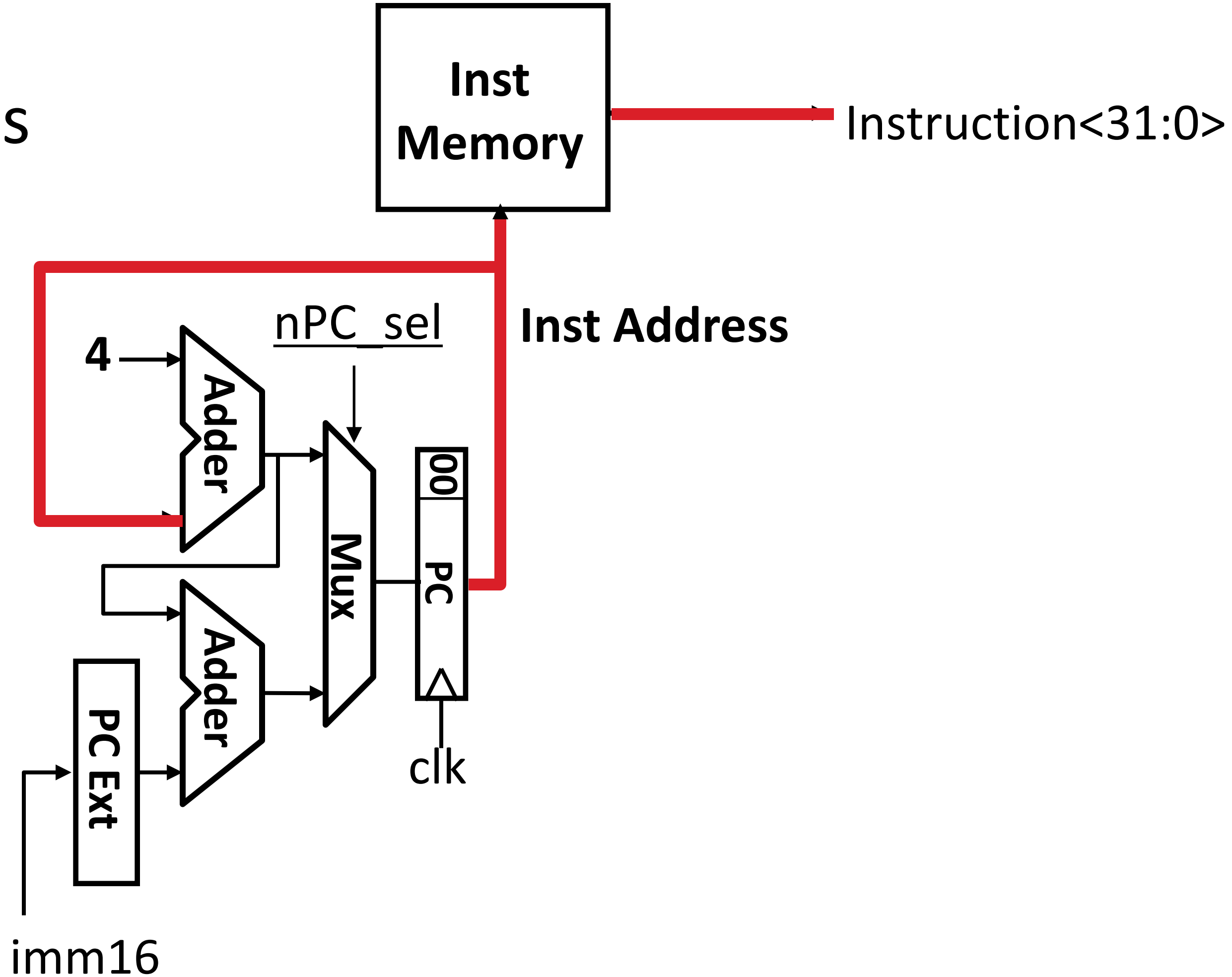
31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

`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`

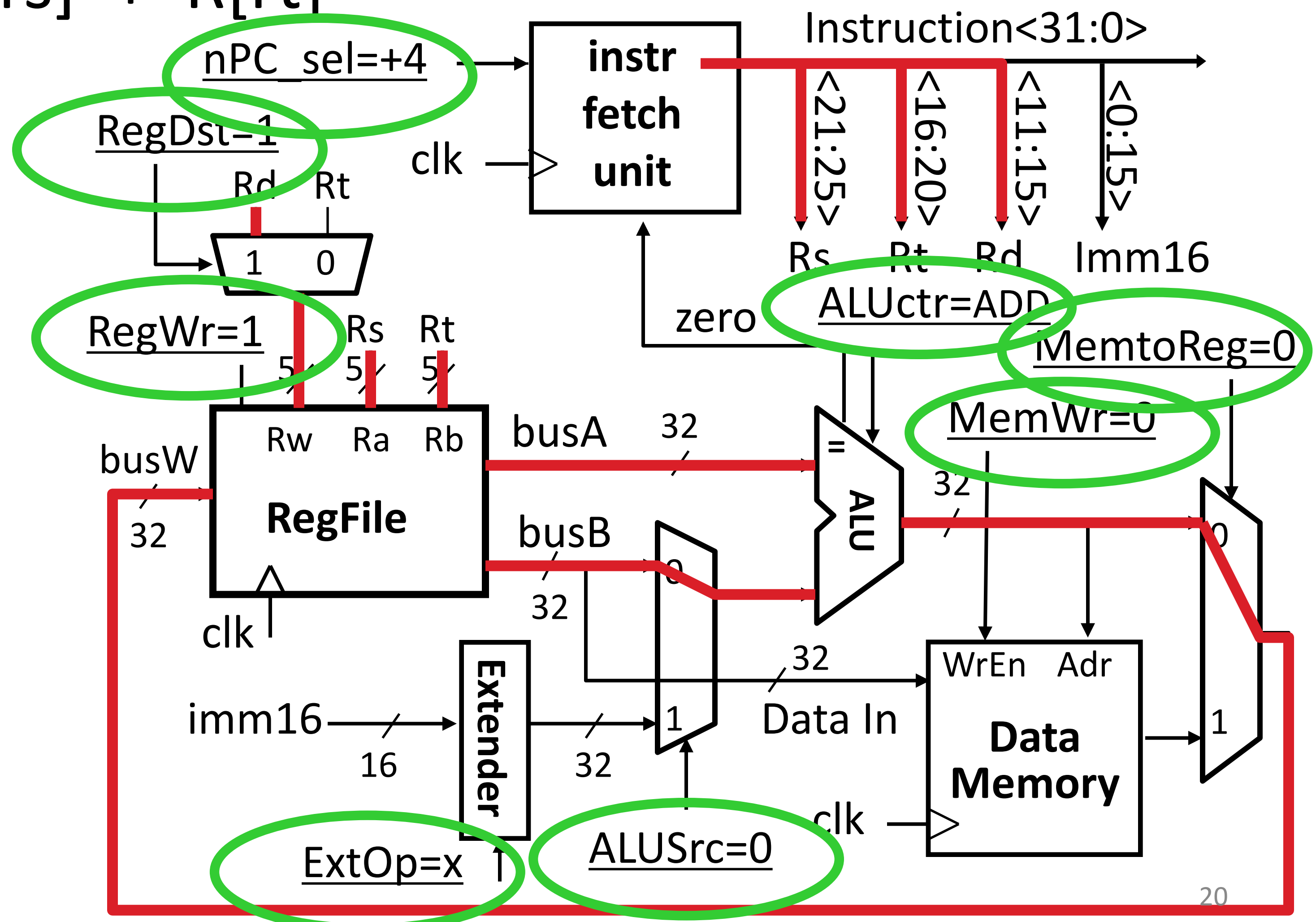
- Fetch the instruction from Instruction memory: Instruction = MEM[PC]
  - same for all instructions



Single Cycle Datapath during Add

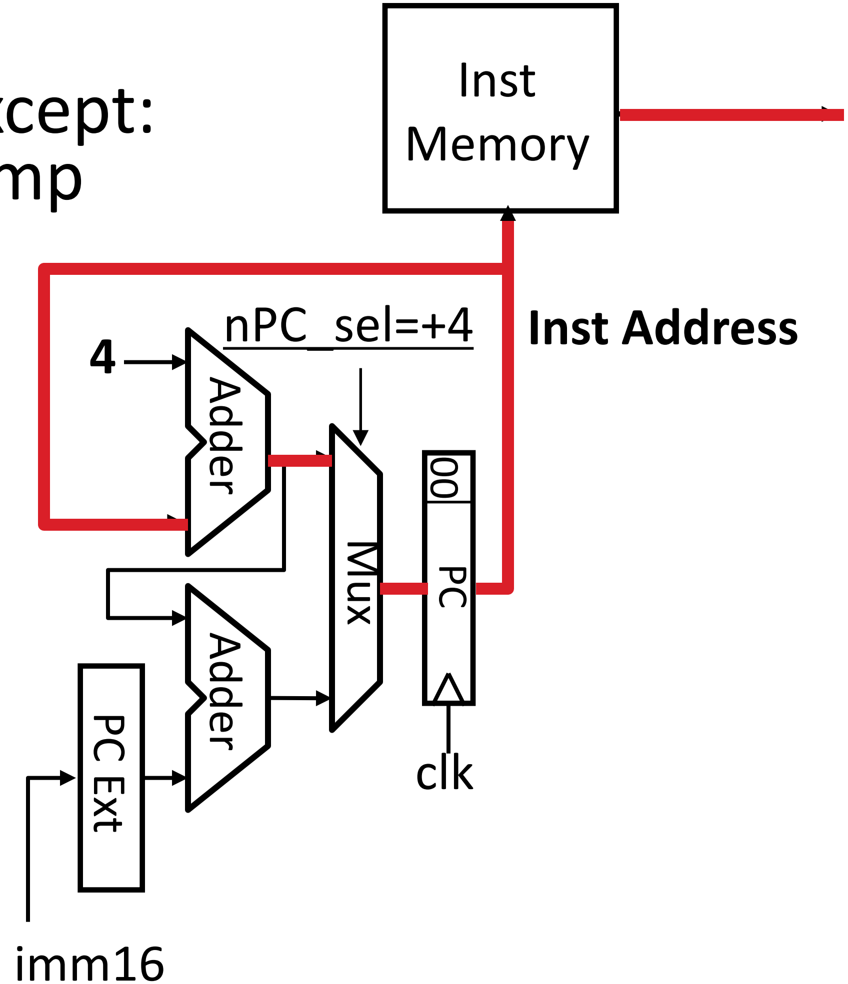
31	26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**	

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



Instruction Fetch Unit at End of `Add`

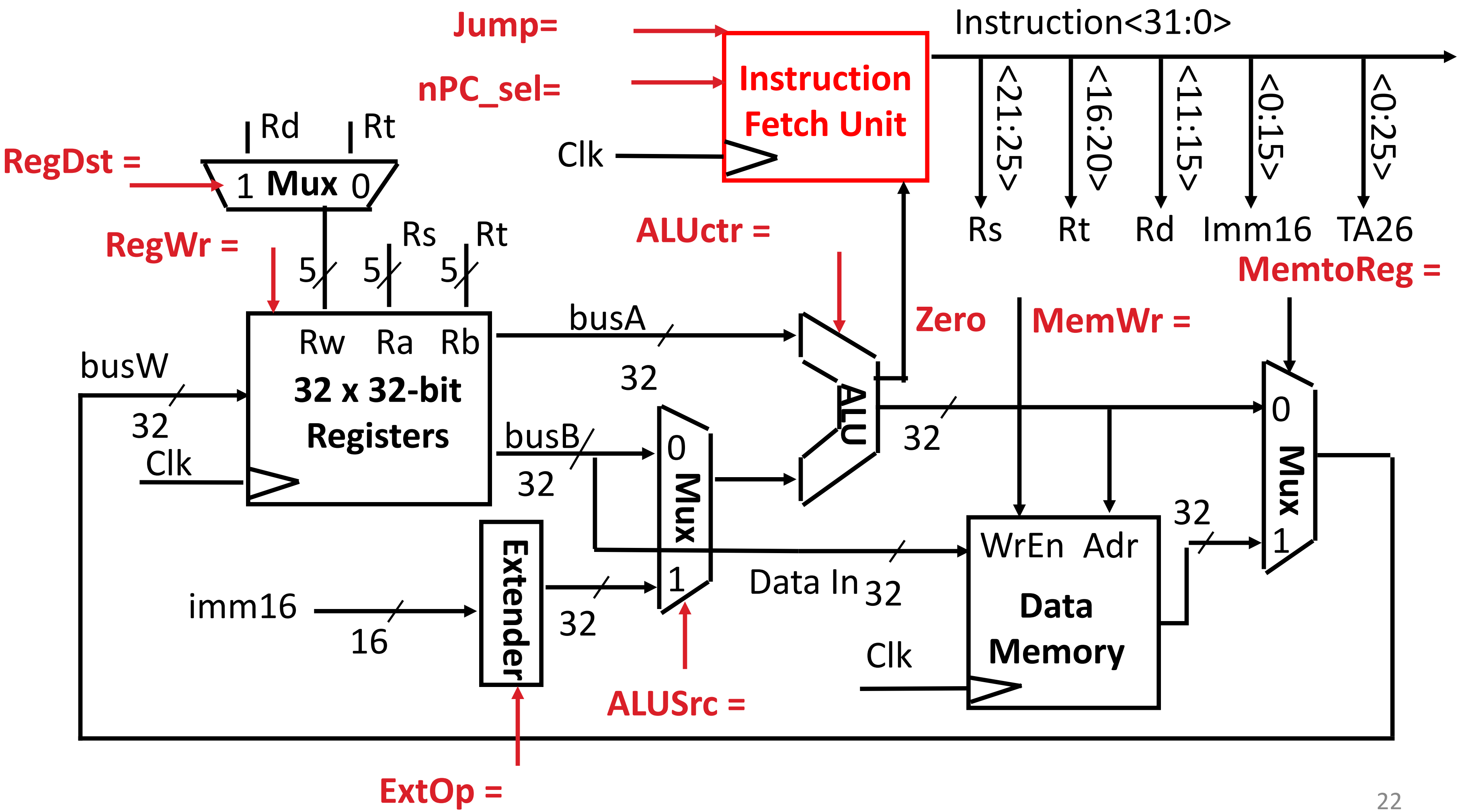
- PC = PC + 4
  - Same for all instructions except: Branch and Jump



Single Cycle Datapath during Jump

	31 … 26	25 … 0	
J-type	**op**	**target address**	**jump**

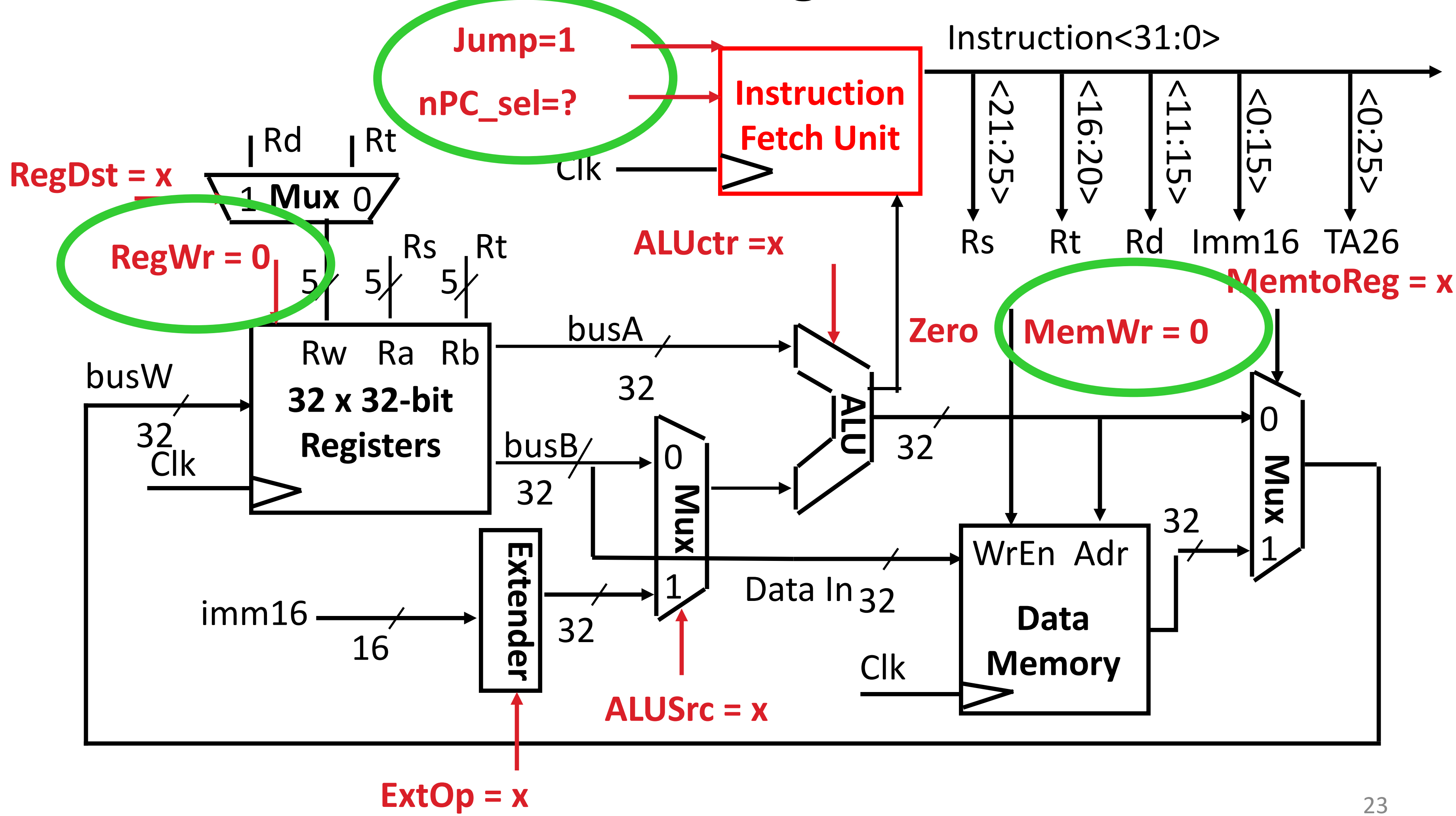
- New PC = { PC[31..28], target address, 00 }



Single Cycle Datapath during Jump

	31 26	25 0	
J-type	op	target address	jump

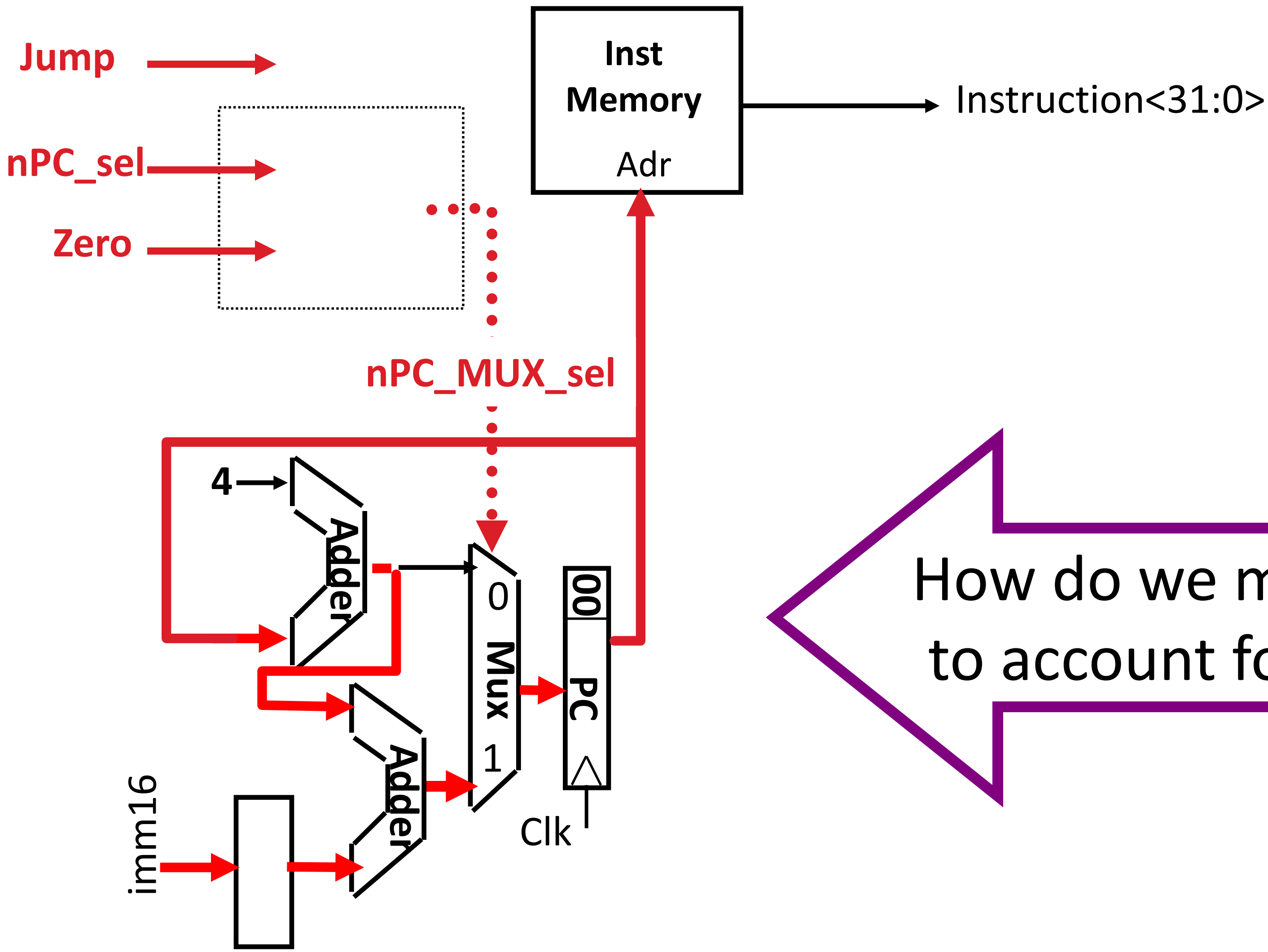
- New PC = { PC[31..28], target address, 00 }



Instruction Fetch Unit at the End of Jump

	31 26	25 0	
J-type	op	target address	jump

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

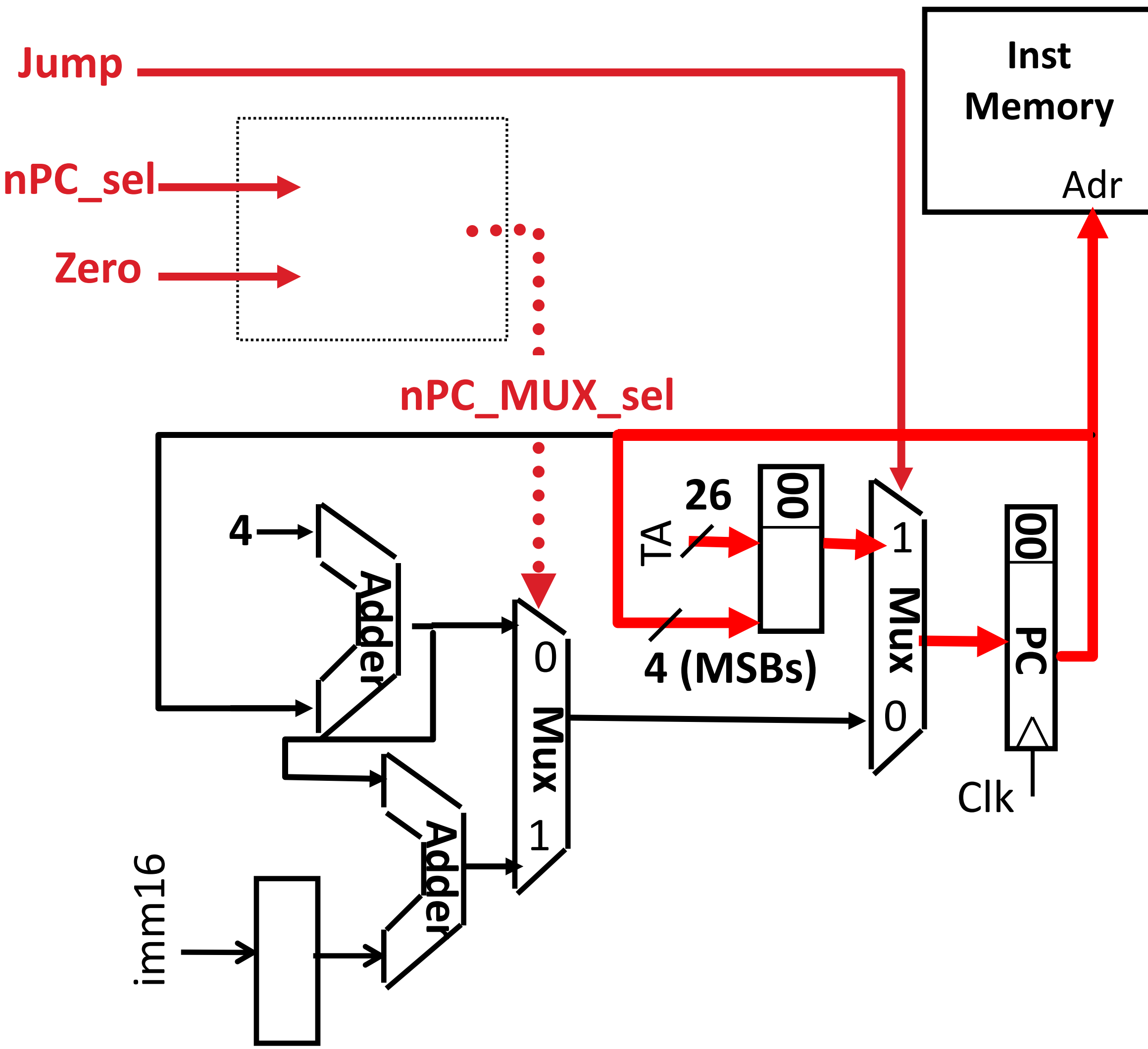


How do we modify this to account for jumps?

Instruction Fetch Unit at the End of Jump

J-type	31 … 26	25 … 0	
	op	target address	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?

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

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

	31 – 26	21	16	11	6	0	
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**	**add, sub**
I-type	**op**	**rs**	**rt**	**immediate**			**ori, lw, sw, beq**
J-type	**op**	**target address**					**jump**

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
```

ADD	0000 00ss ssst tttt dddd d 000 00 **10 0000**
SUB	0000 00ss ssst tttt dddd d 000 00 **10 0010**
ORI	0011 01ss ssst tttt iiii iiii iiii iiii
LW	1000 11ss ssst tttt iiii iiii iiii iiii
SW	1010 11ss ssst tttt iiii iiii iiii iiii
BEQ	0001 00ss ssst tttt iiii iiii iiii iiii
JUMP	0000 10ii iiii iiii iiii iiii iiii iiii

Where:

$rtype = \sim op_5 \cdot \sim op_4 \cdot \sim op_3 \cdot \sim op_2 \cdot \sim op_1 \cdot \sim op_0$

$ori = \sim op_5 \cdot \sim op_4 \cdot op_3 \cdot op_2 \cdot \sim op_1 \cdot op_0$

$lw = op_5 \cdot \sim op_4 \cdot \sim op_3 \cdot \sim op_2 \cdot op_1 \cdot op_0$

$sw = op_5 \cdot \sim op_4 \cdot op_3 \cdot \sim op_2 \cdot op_1 \cdot op_0$

$beq = \sim op_5 \cdot \sim op_4 \cdot \sim op_3 \cdot op_2 \cdot \sim op_1 \cdot \sim op_0$

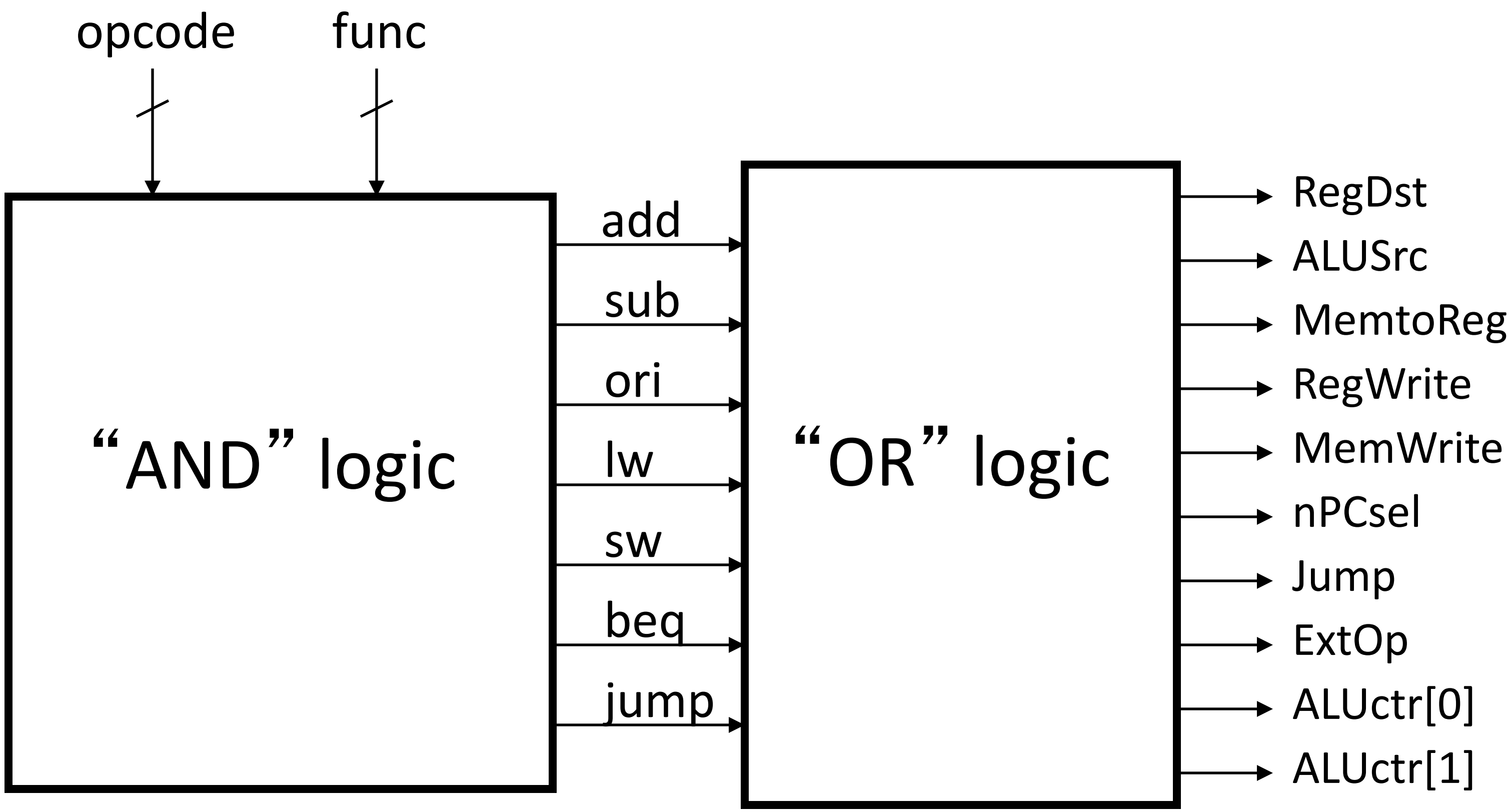
$jump = \sim op_5 \cdot \sim op_4 \cdot \sim op_3 \cdot \sim op_2 \cdot op_1 \cdot \sim op_0$

How do we implement this in gates?

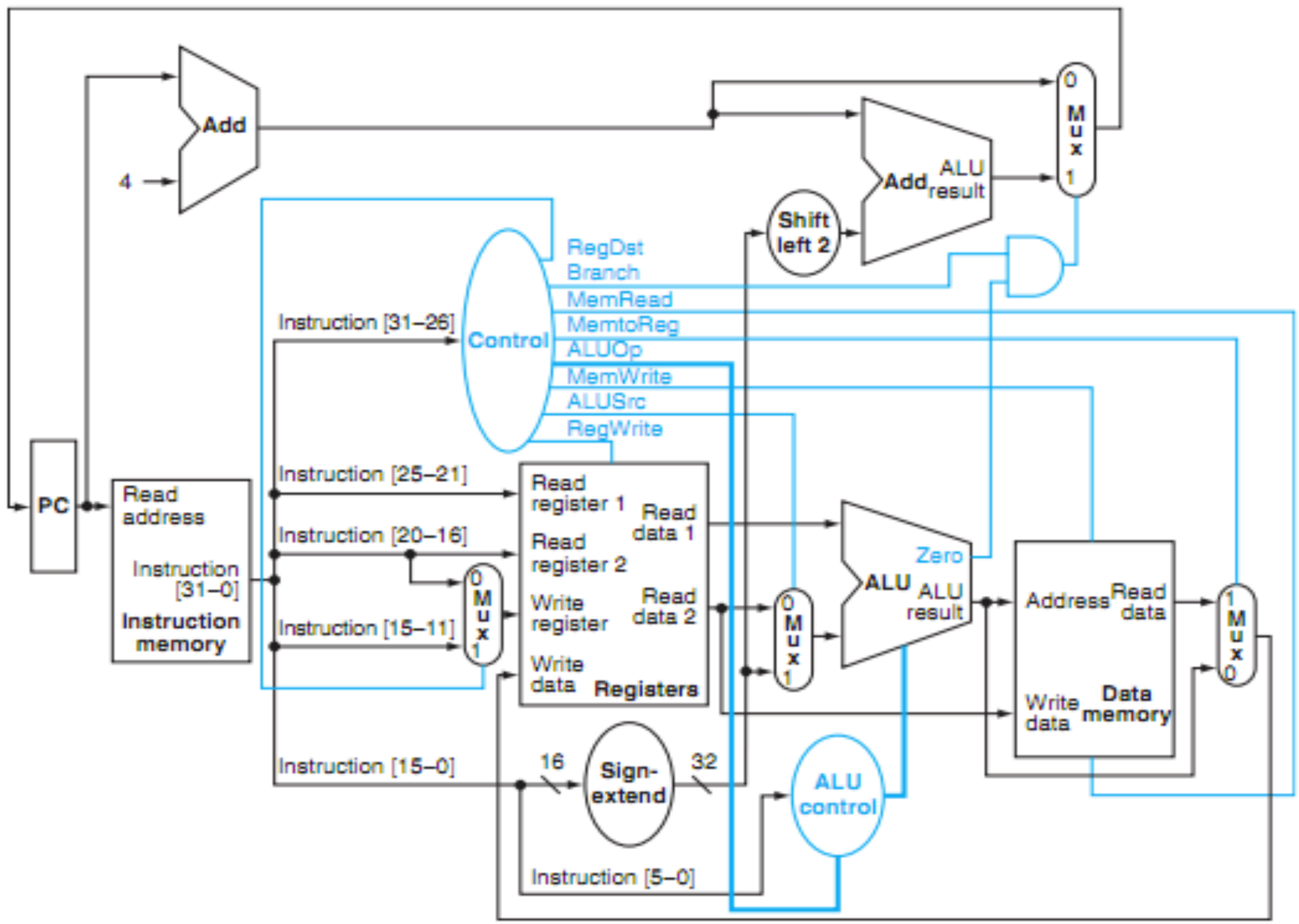
$add = rtype \cdot func_5 \cdot \sim func_4 \cdot \sim func_3 \cdot \sim func_2 \cdot \sim func_1 \cdot \sim func_0$

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

Controller Implementation

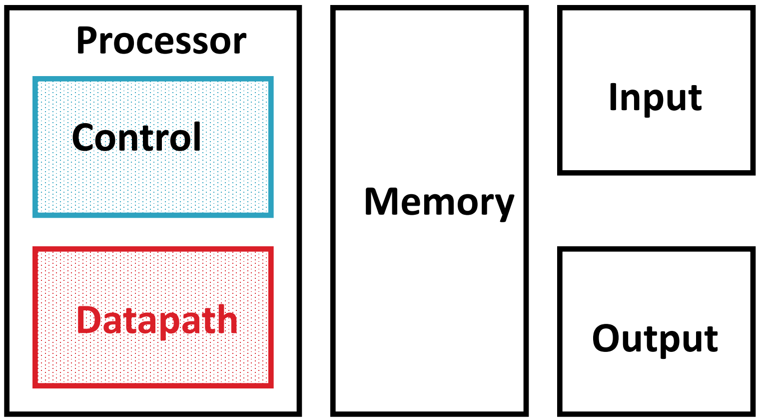


P&H Figure 4.17



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

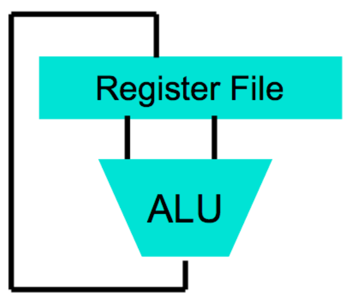
Logic Circuit Description (Circuit Schematic Diagrams)

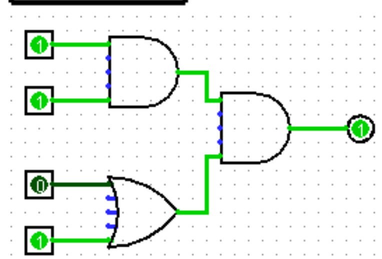
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
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!