CS 61C:
Great Ideas in Computer Architecture
Single-Cycle CPU
Datapath & Control
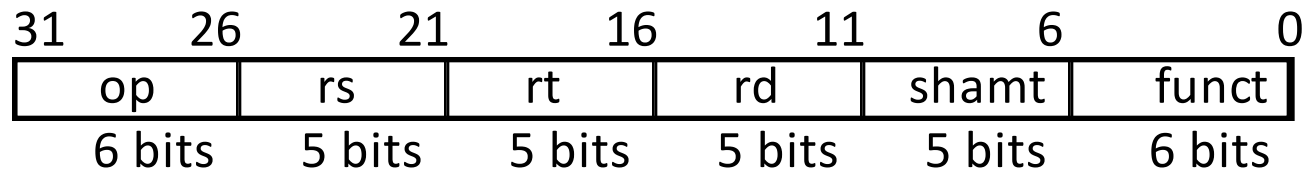
Instructors:

John Wawrzynek & Vladimir Stojanovic
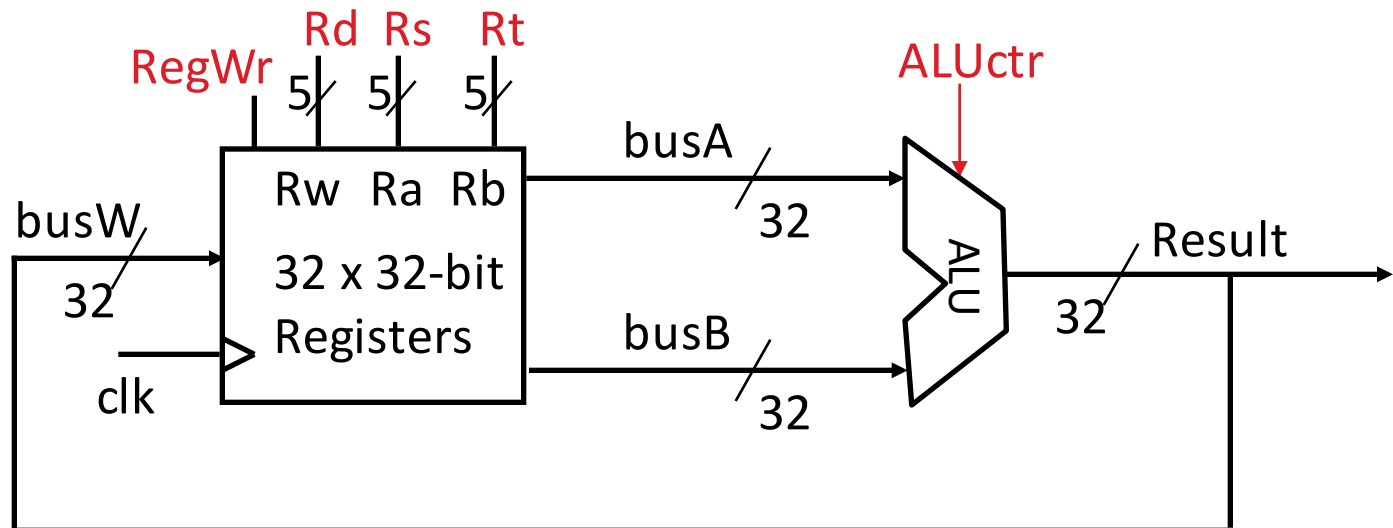
http://inst.eecs.Berkeley.edu/~cs61c/fa15

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

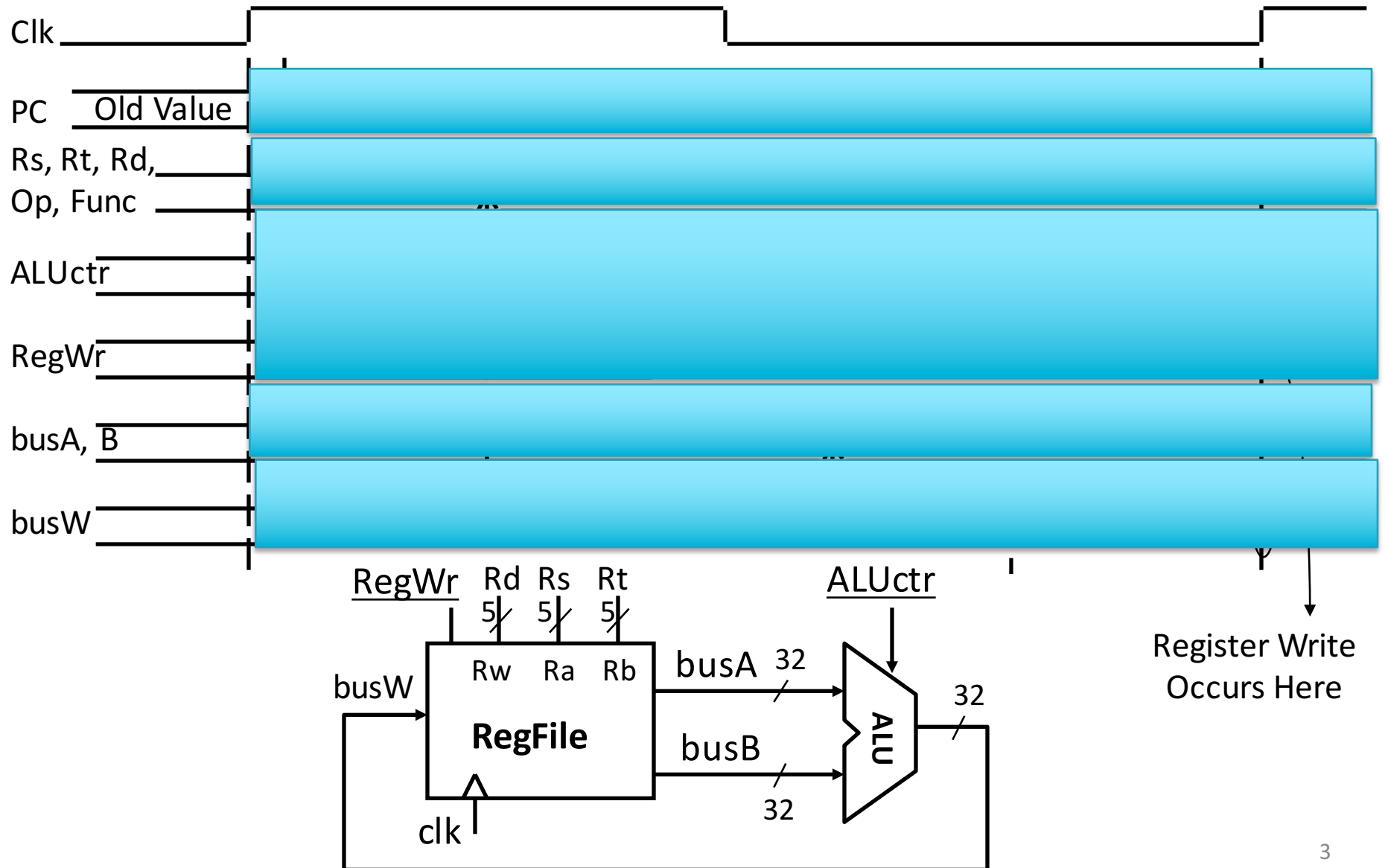| 31 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - ALUctr and RegWr: control logic after decoding the instruction

- ... Already defined the register file & ALU

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

Clk

PC   Old Value

Rs, Rt, Rd,

Op, Func

ALUctr

RegWr

busA, B

busW

RegWr Rd Rs Rt
        5   5   5

busW   Rw   Ra   Rb   busA 32

       RegFile   32

                          busB

       clk   32

ALUctr

ALU
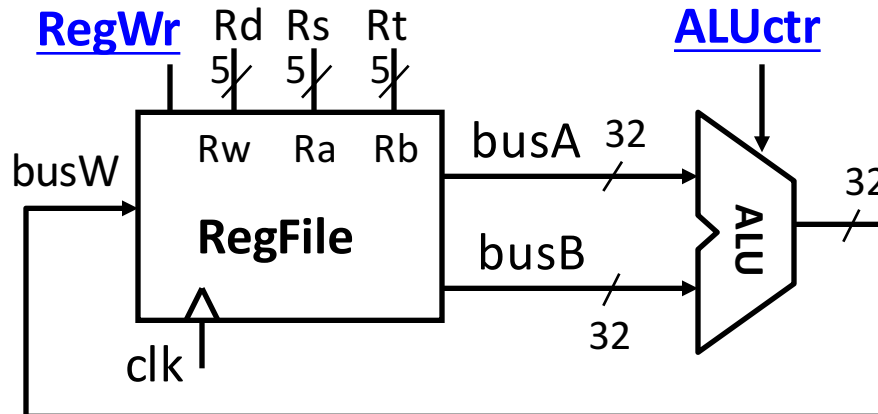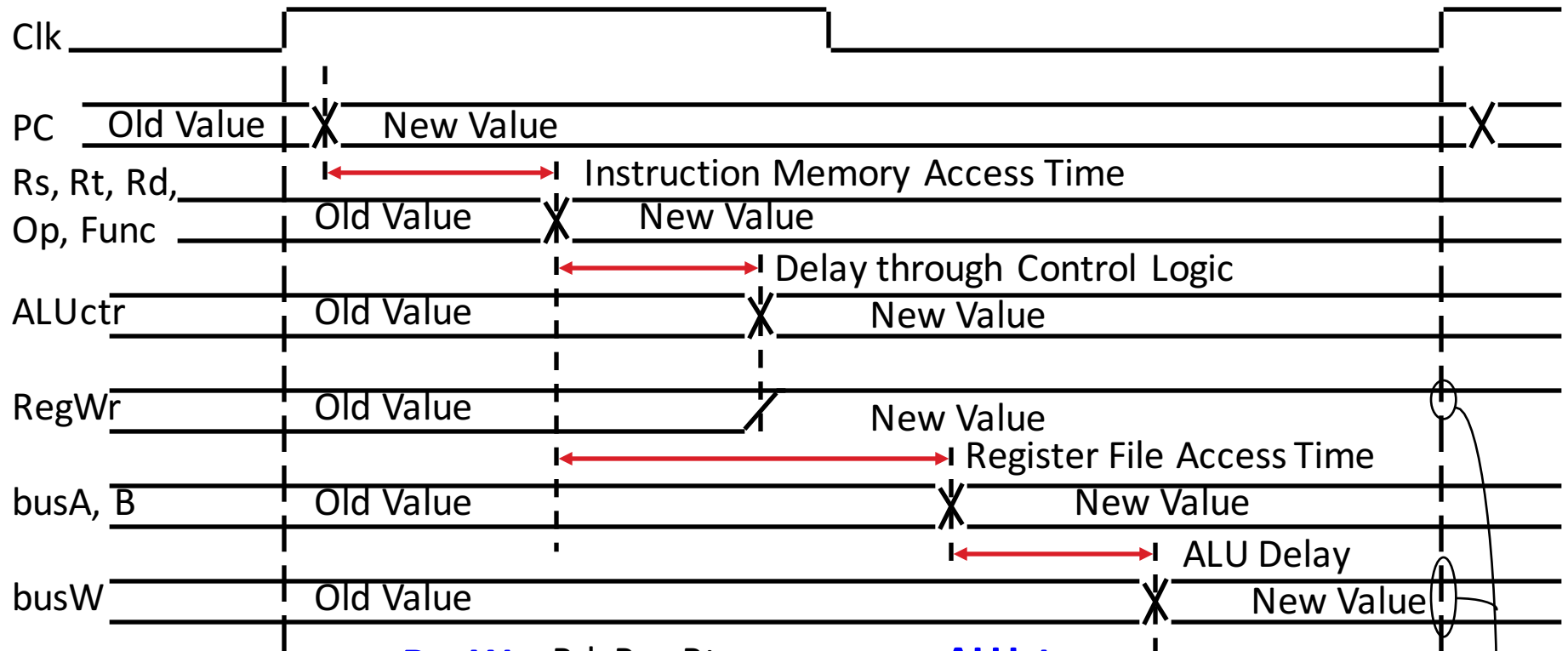
Register Write
Occurs Here

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?
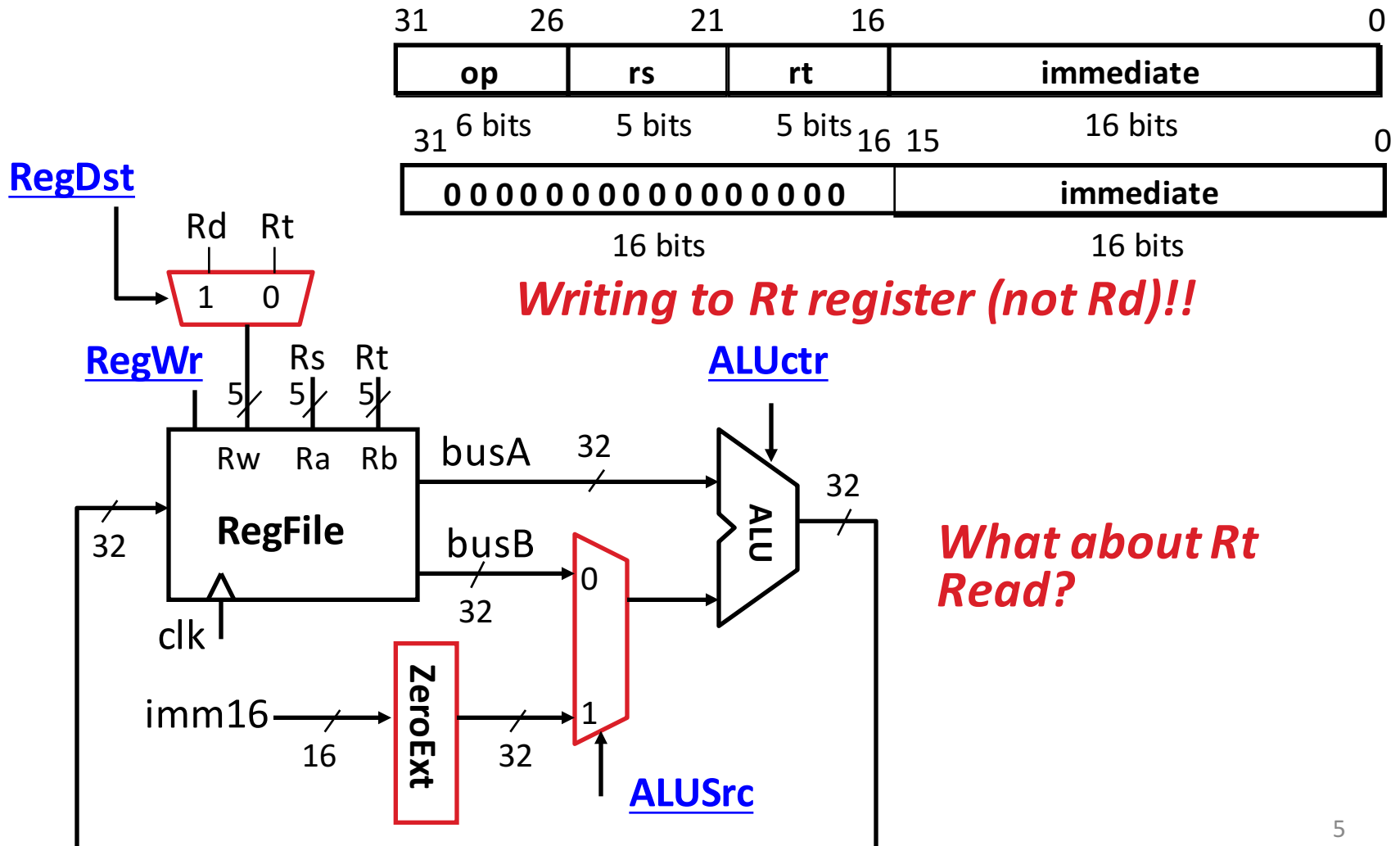
3d: Load Operations

- R[rt] = Mem[R[rs] + SignExt[imm16]]
  Example: `lw rt,rs,imm16`
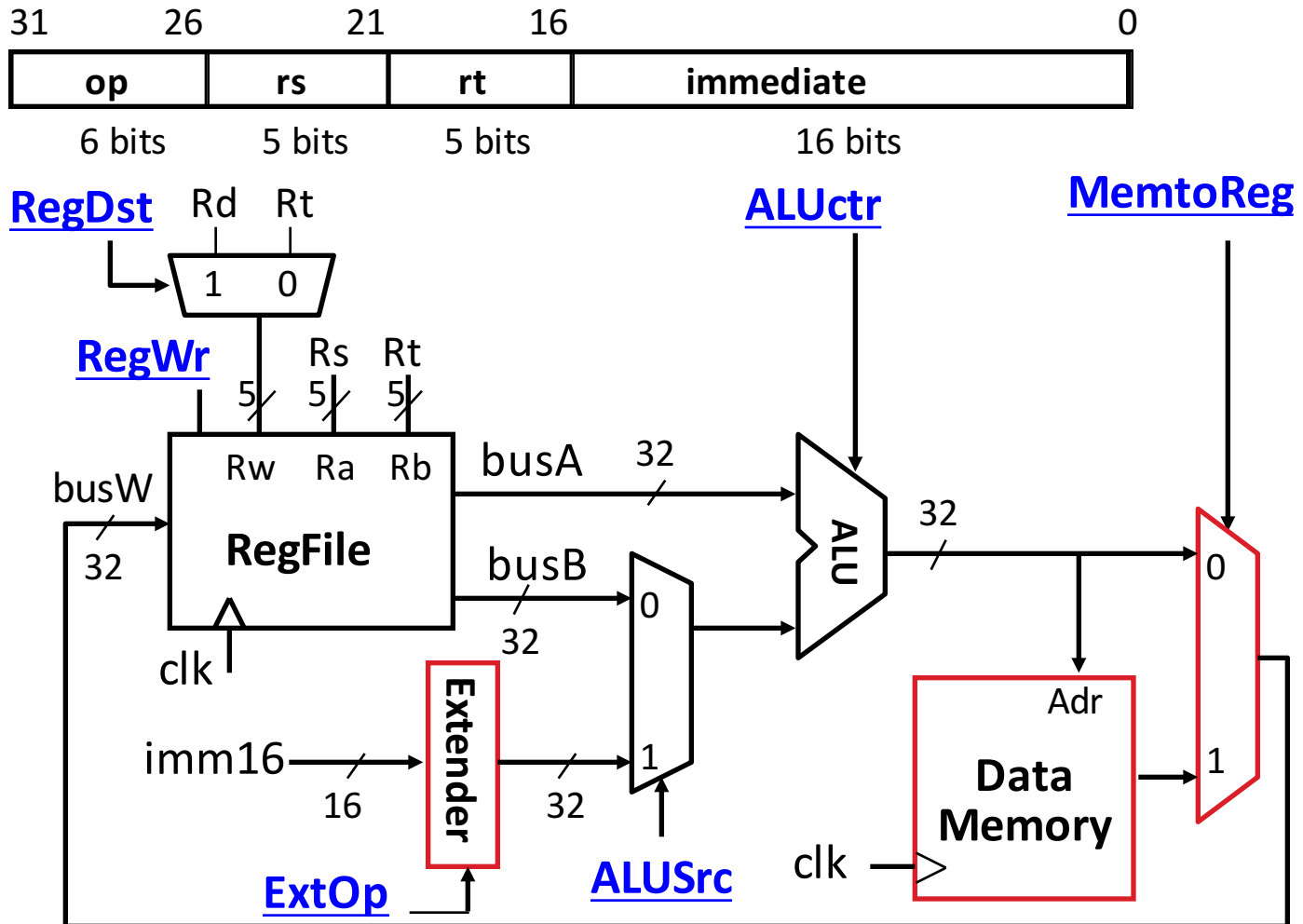
3e: Store Operations

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

Ex.: `sw rt, rs, imm16`

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

31   26   21   16   0

3e: Store Operations

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

Ex.: `sw rt, rs, imm16`

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

31   26   21   16   0
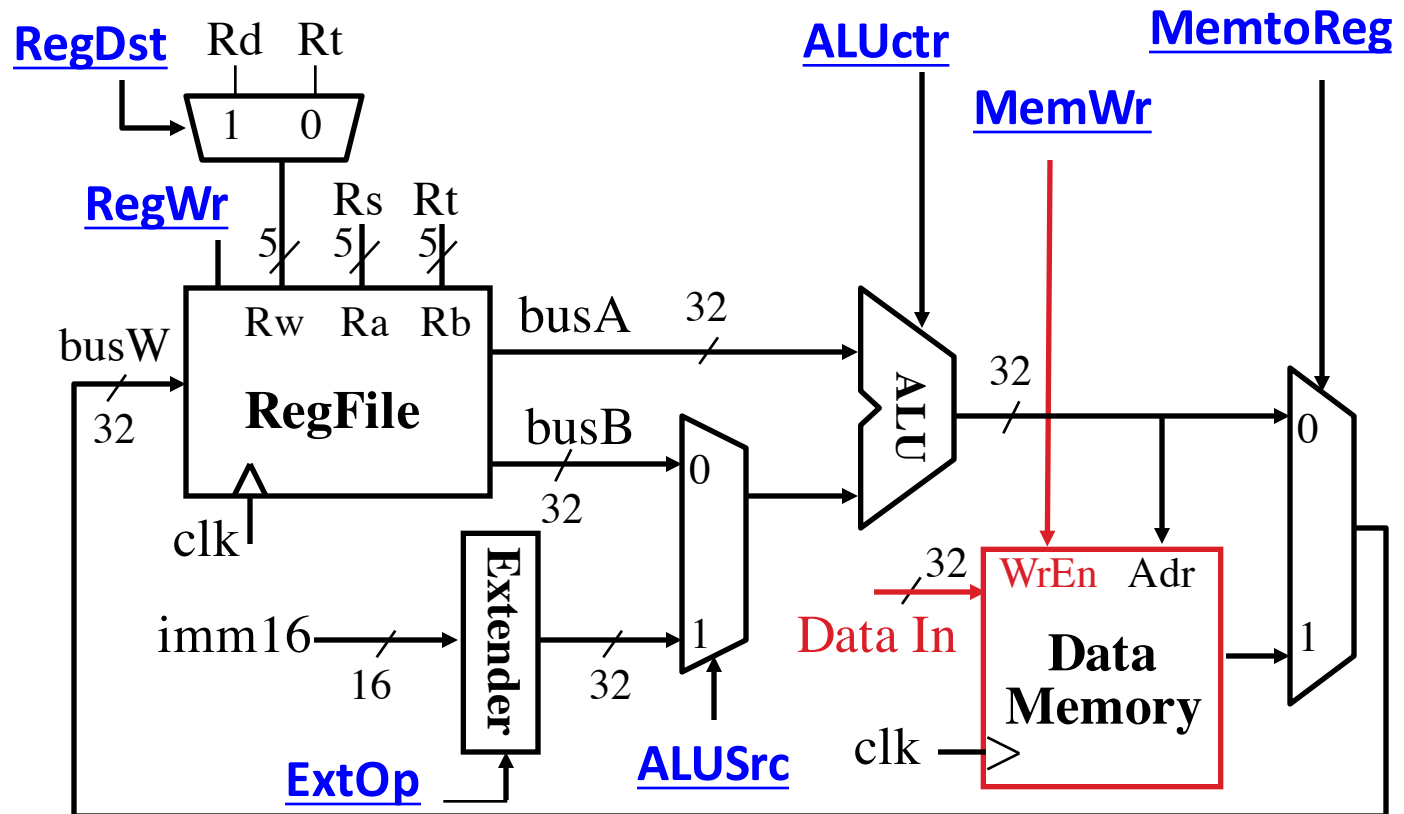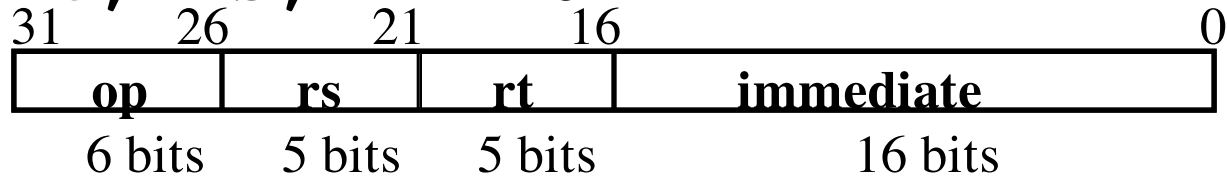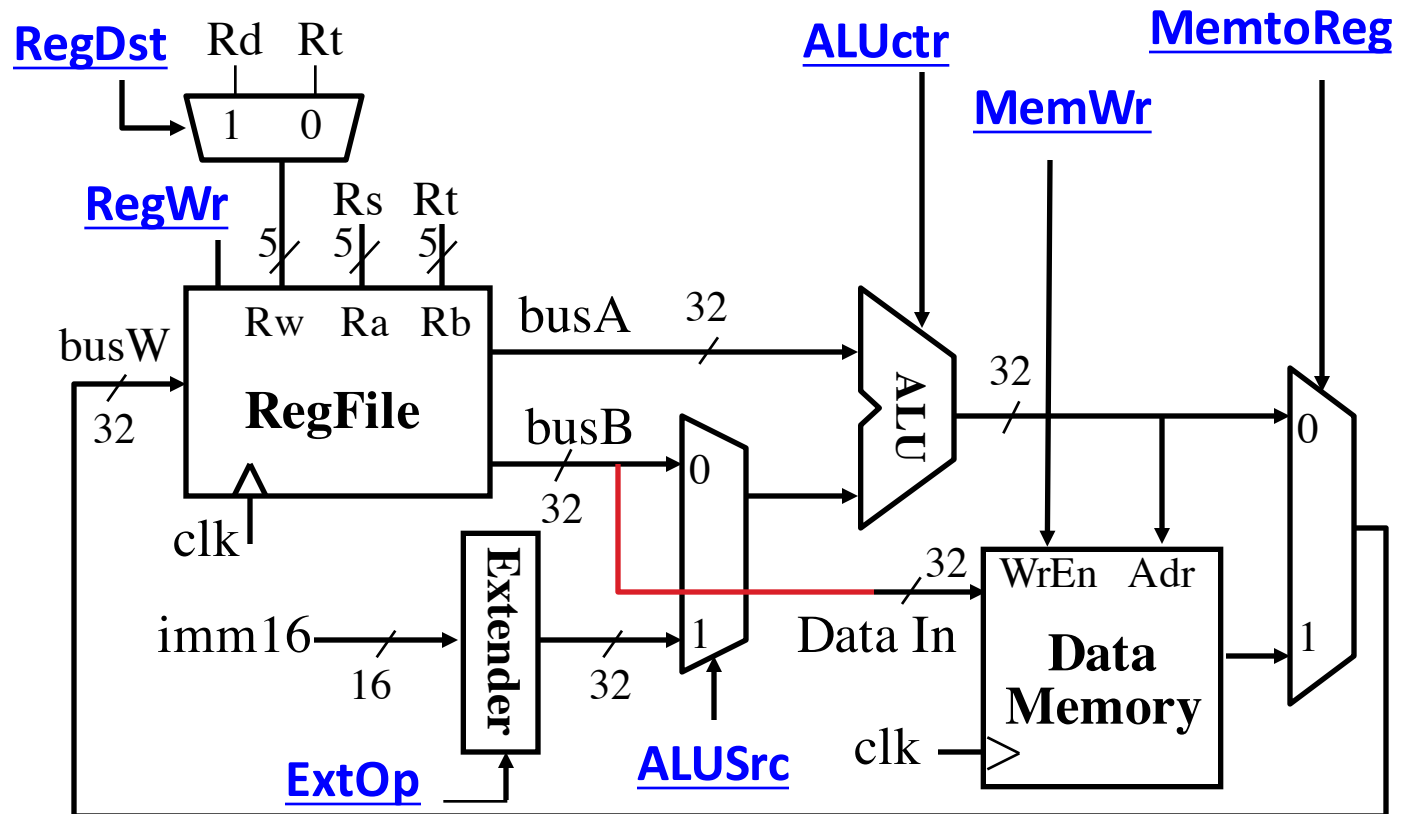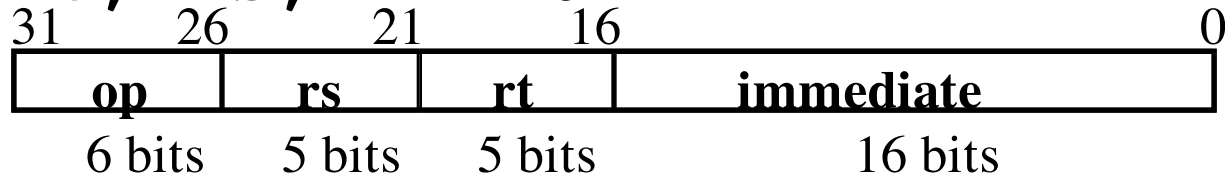
3f: The Branch Instruction

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

6 bits   5 bits   5 bits   16 bits
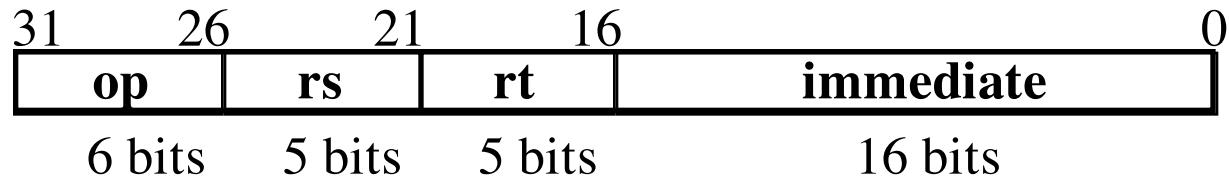
`beq rs, rt, imm16`

– 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

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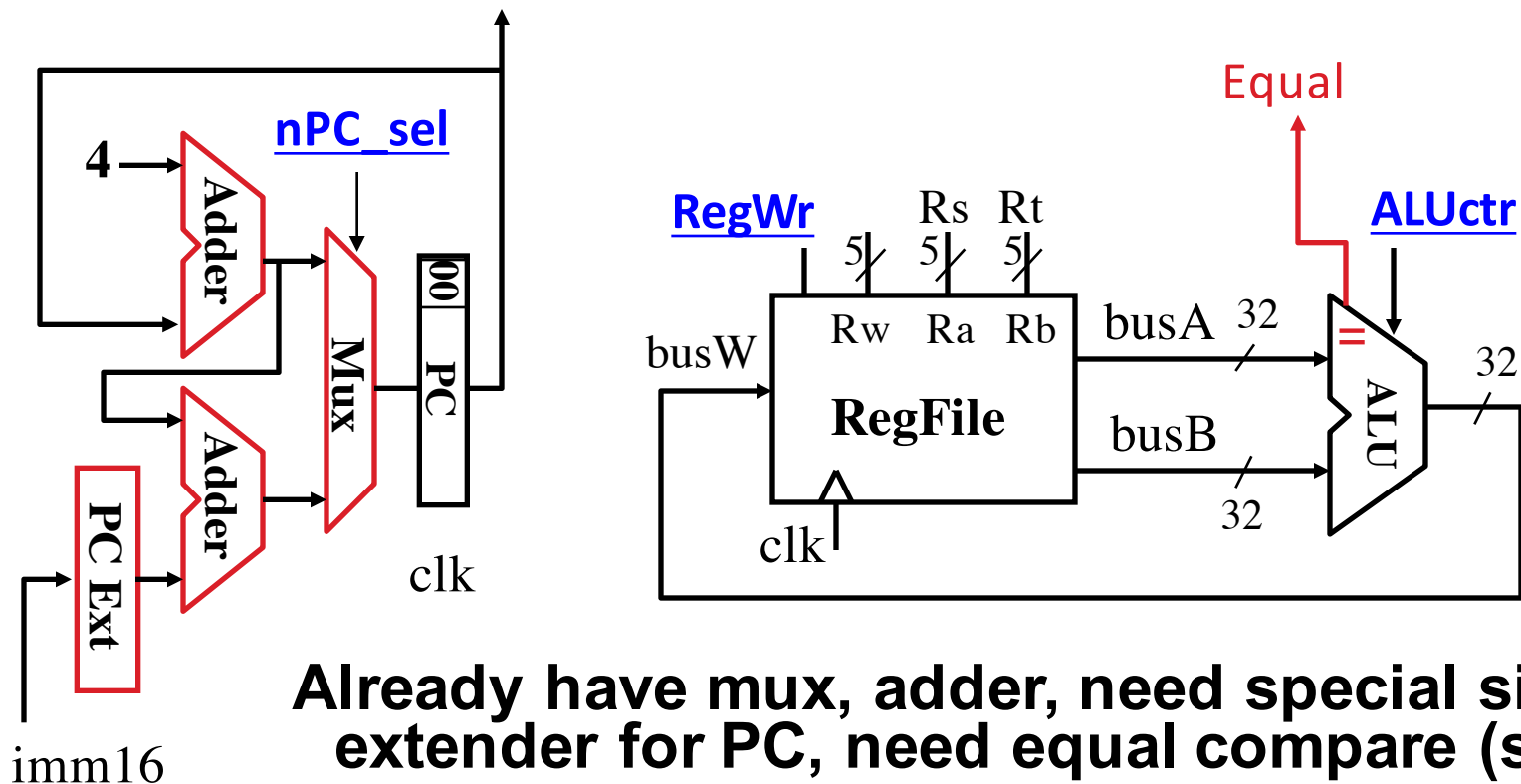
Already have mux, adder, need special sign extender for PC, need equal compare (sub?)

Instruction Fetch Unit including Branch

31	26	21	16	0

op	rs	rt	immediate

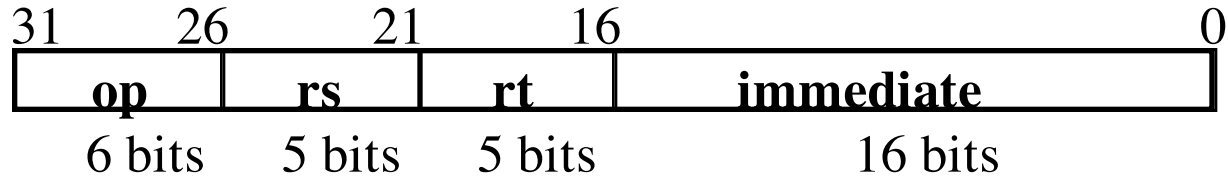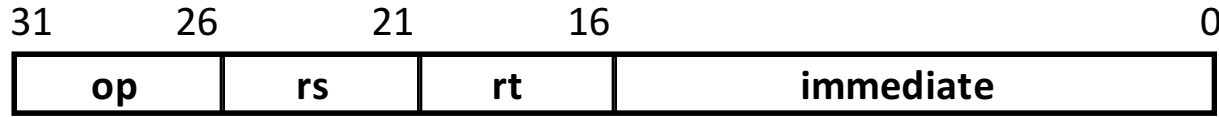- if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4 ; else PC = PC + 4

nPC_sel

Equal

MUX ctrl

Inst Memory Adr

Instruction<31:0>

4 → Adder

Mux 0 / 1

00 PC

imm16 → PC Ext → Adder

clk
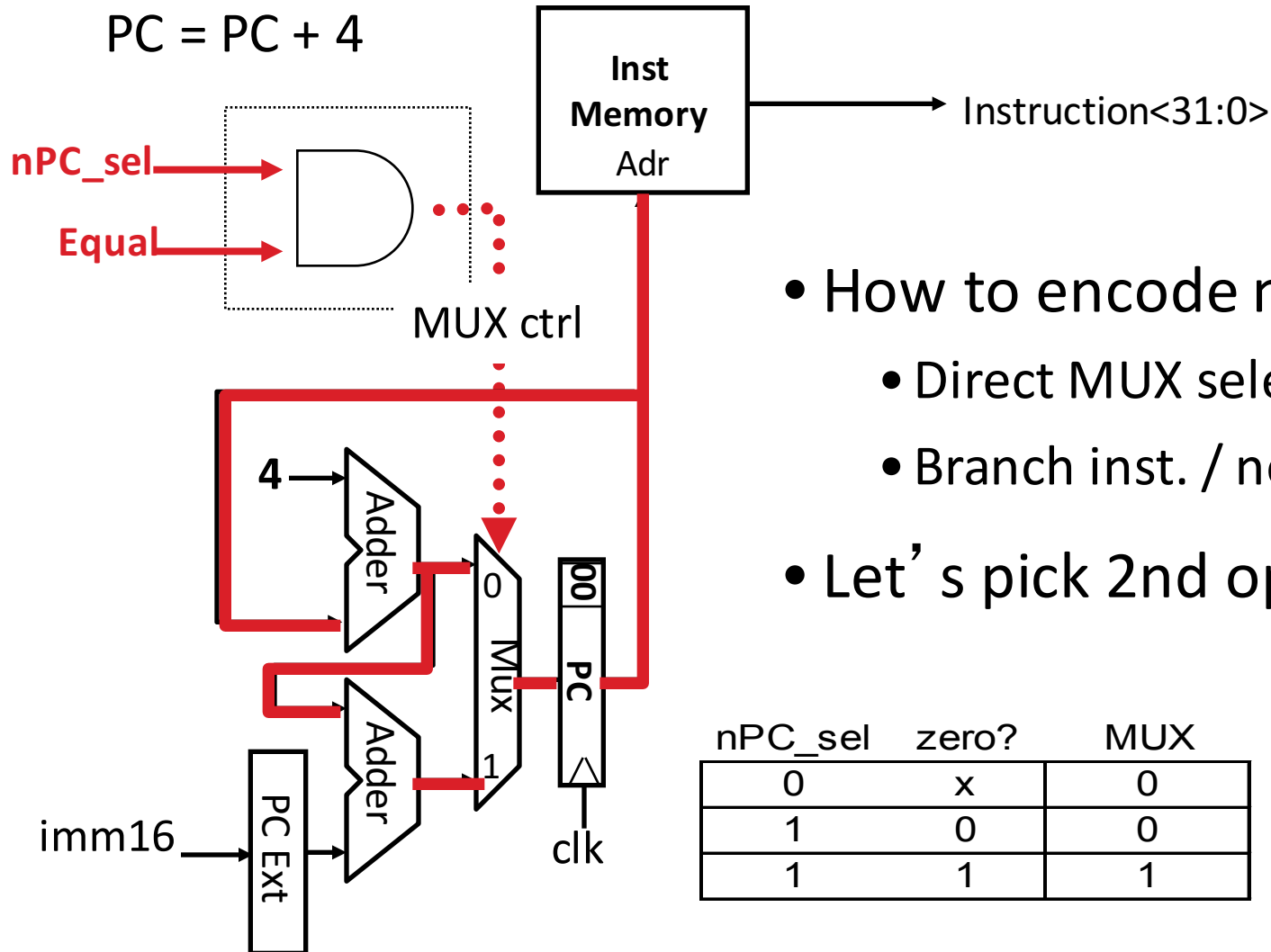
- How to encode nPC_sel?
  - Direct MUX select?
  - Branch inst. / not branch inst.

- Let's pick 2nd option

Q: What logic gate?

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

11

Putting it All Together: A Single Cycle Datapath
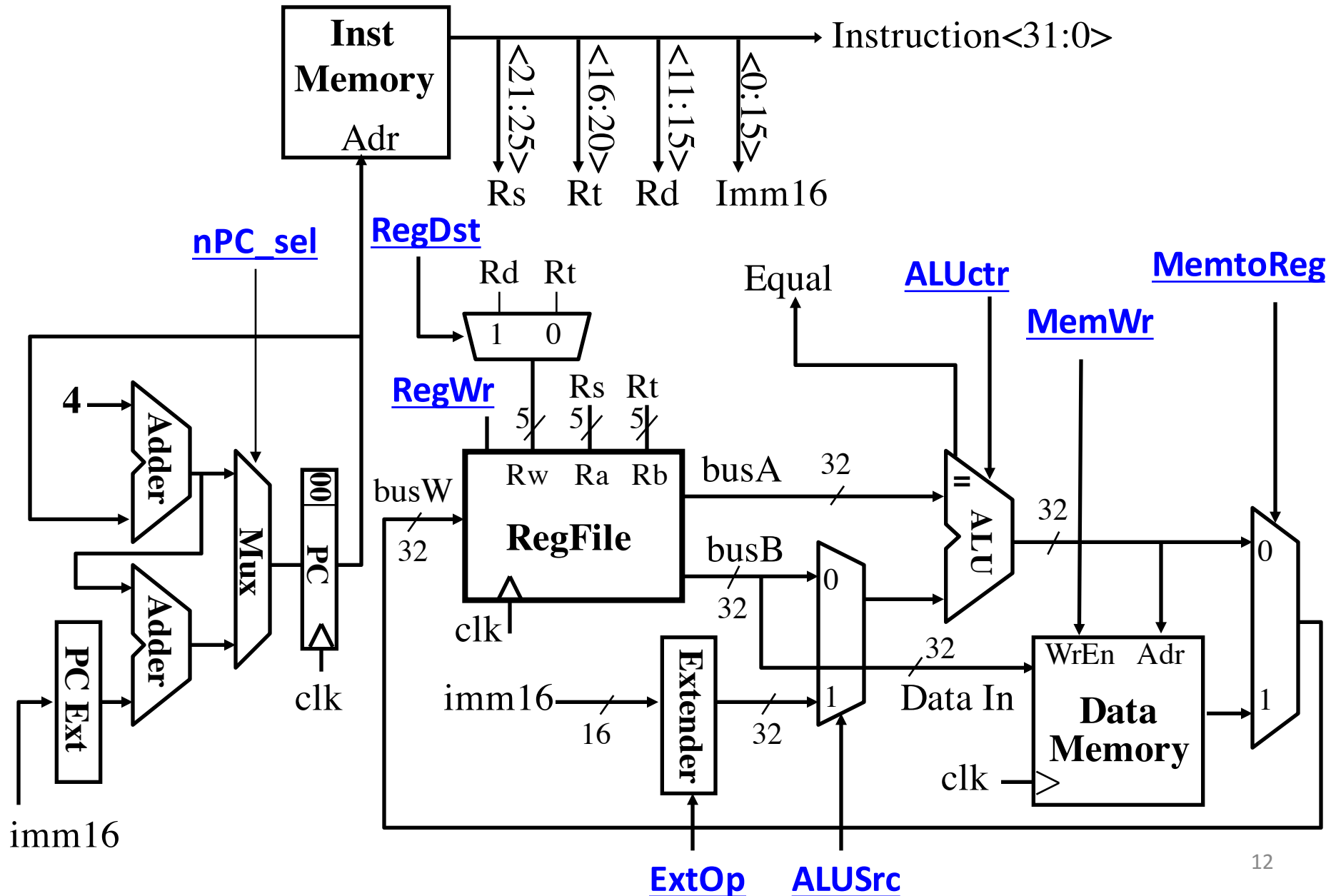
Clickers/Peer Instruction

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 one bit
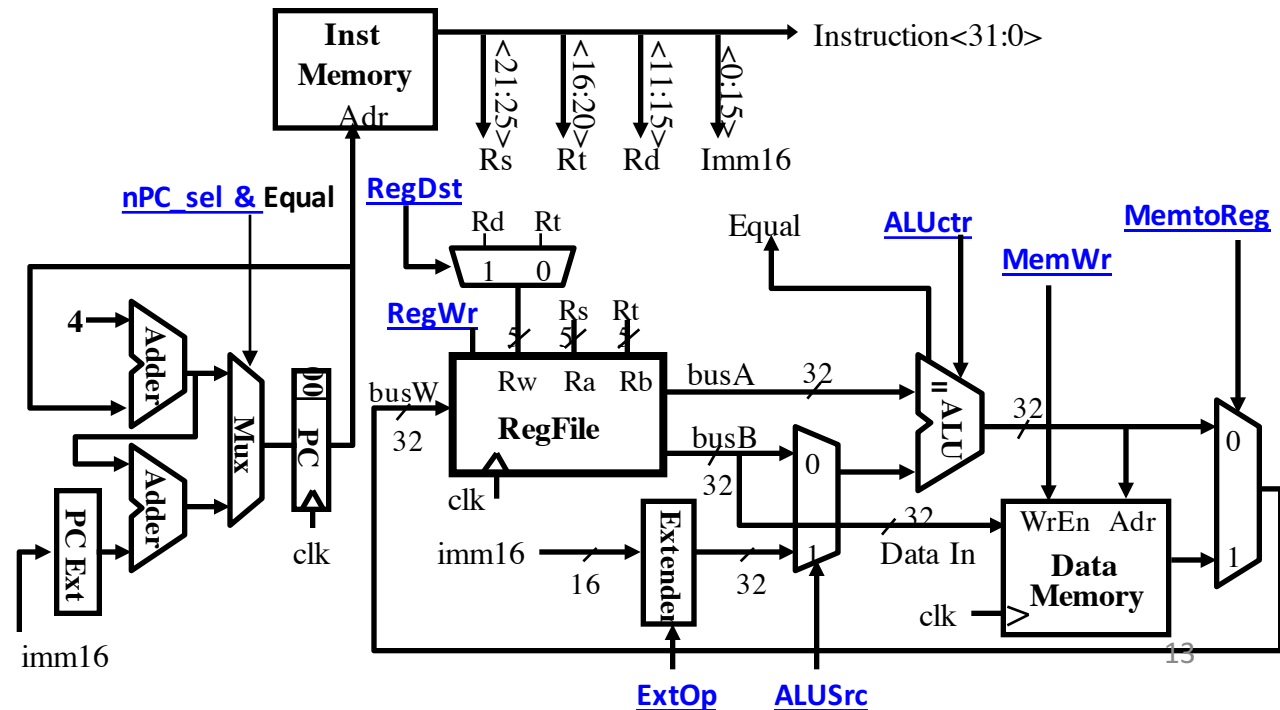
Administrivia

- Midterm
  - A: Super Hard

  …

  - E: Too Easy

- Project 2-2 due 10/11
  - A: Haven't started
  - B: Done with Step 1
  - C: Done with Step 2
  - D: Done with Step 3
  - E: Finished

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: $0 \Rightarrow$ regB;
  $1 \Rightarrow$ immed
- ALUctr: "ADD", "SUB", "OR"

- MemWr: $1 \Rightarrow$ write memory
- MemtoReg: $0 \Rightarrow$ ALU; $1 \Rightarrow$ Mem
- RegDst: $0 \Rightarrow$ "rt"; $1 \Rightarrow$ "rd"
- RegWr: $1 \Rightarrow$ write register
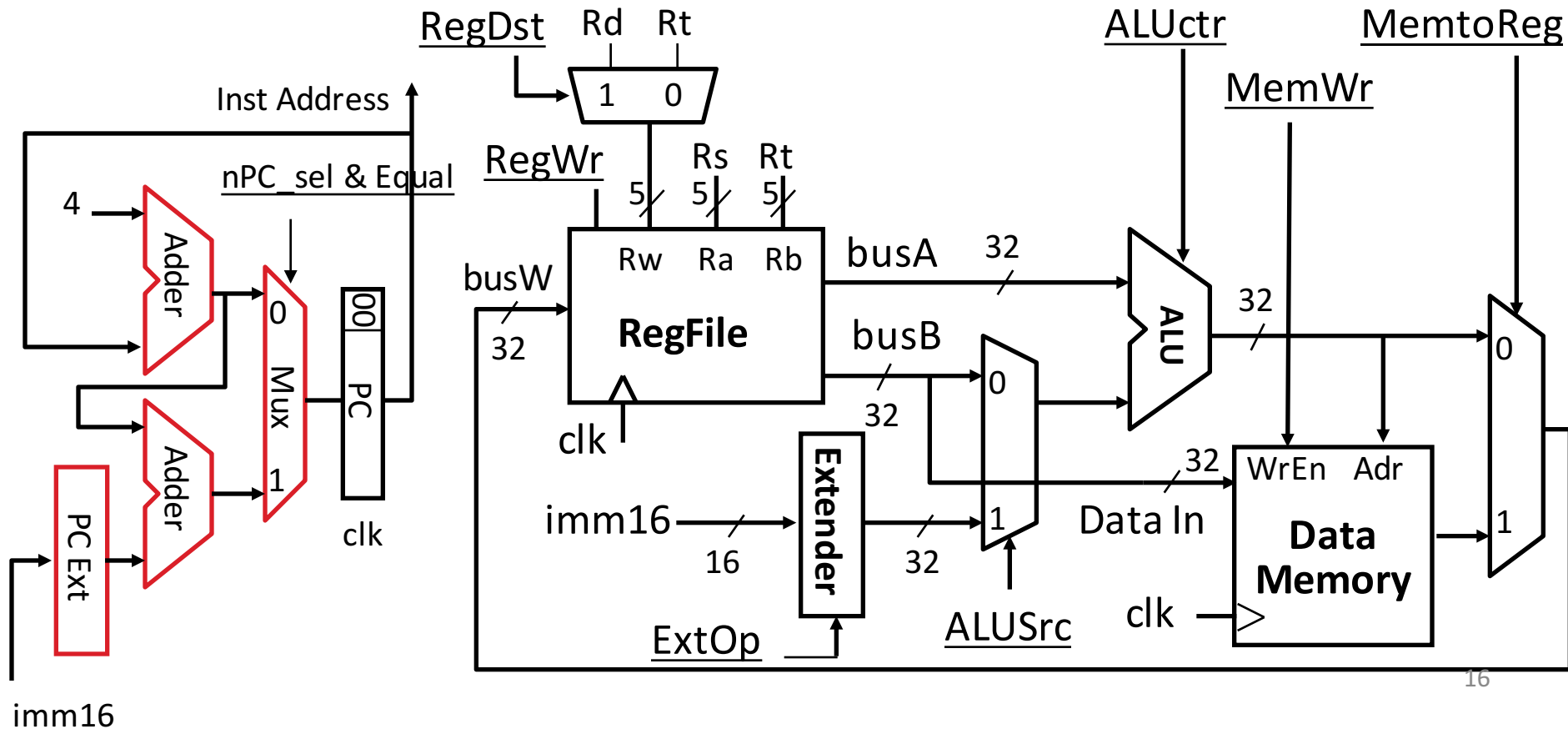
Given Datapath: RTL → Control

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
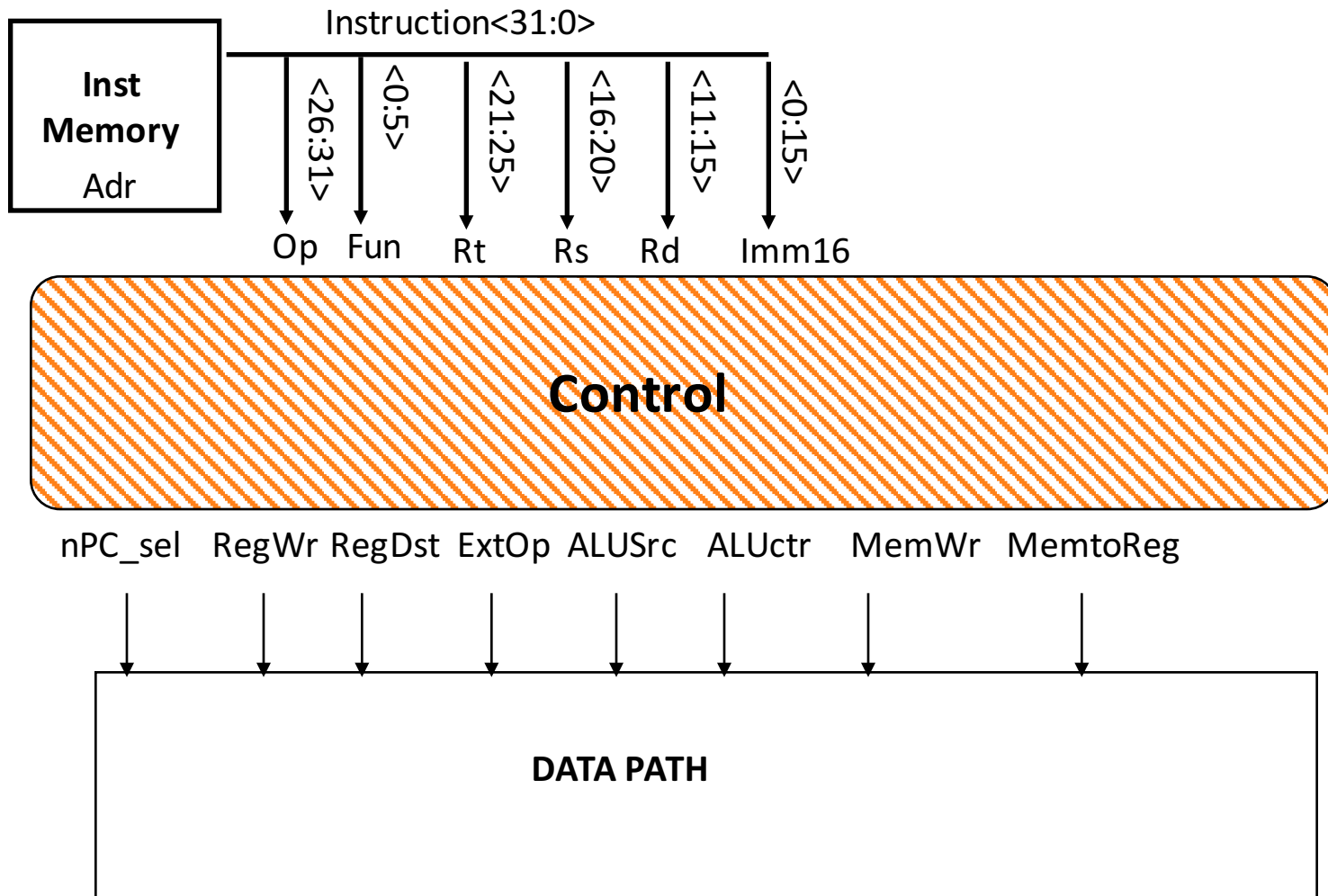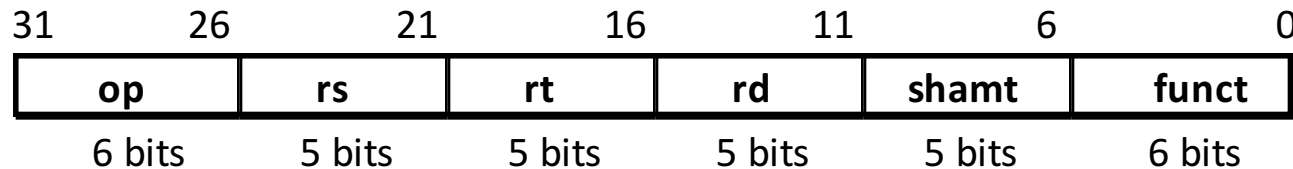
add rd, rs, rt

– 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

Inst Memory

Instruction<31:0>

nPC_sel

Inst Address

4

Adder

Mux

00

PC

Adder

PC Ext

clk

imm16
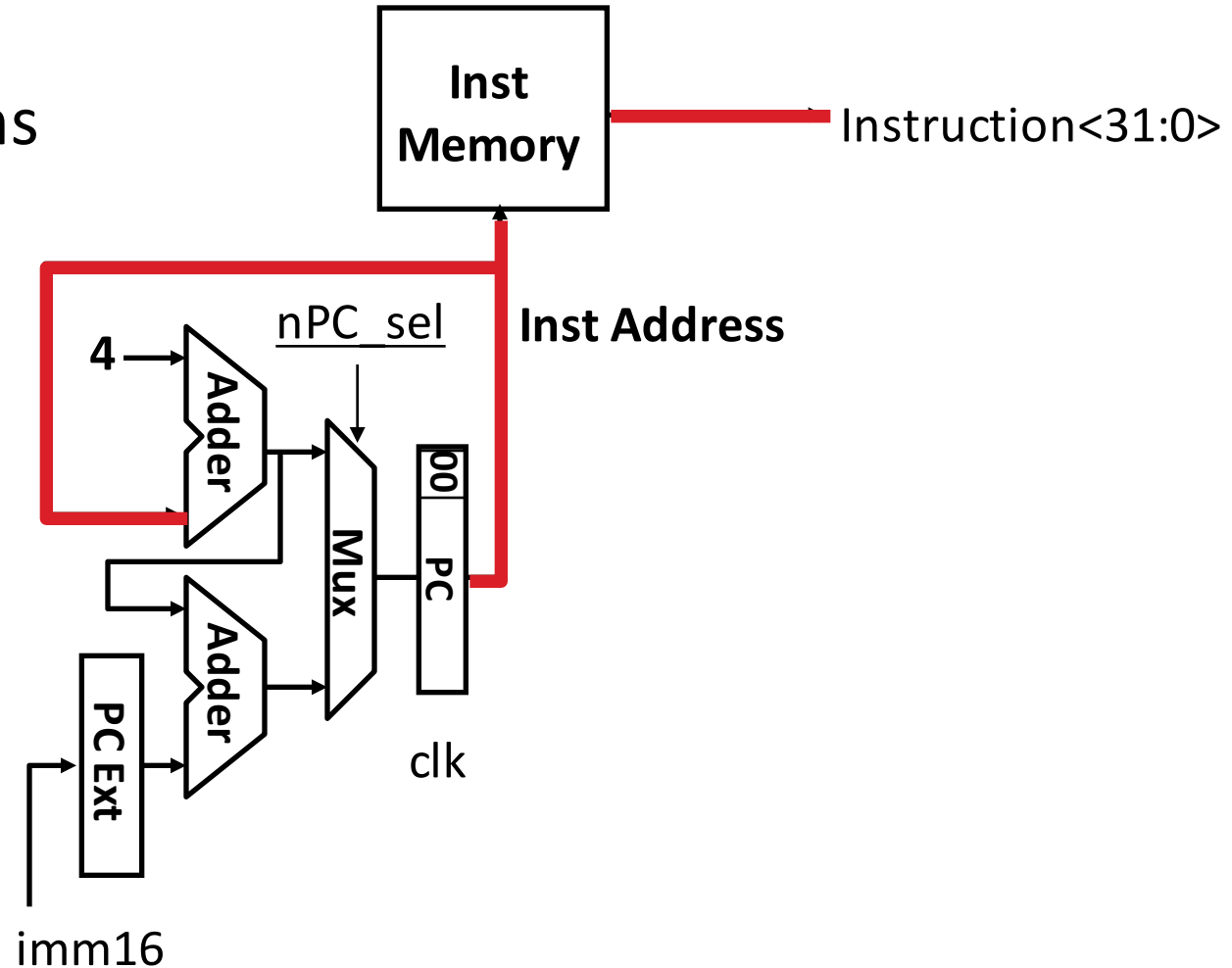
Single Cycle Datapath during Add
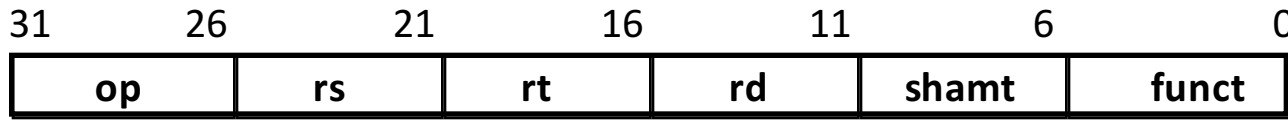
R[rd] = R[rs] + R[rt]
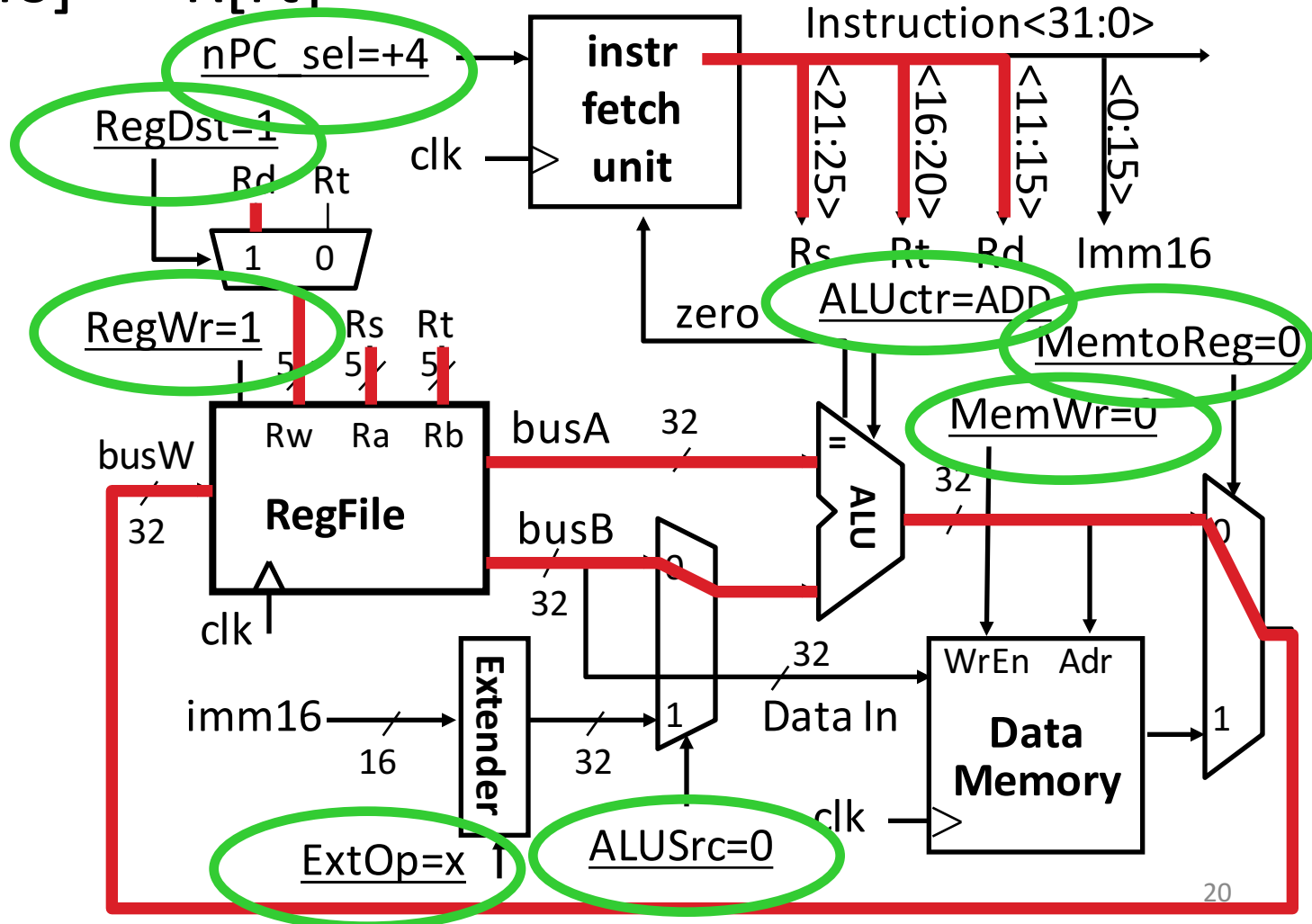
Instruction Fetch Unit at End of Add

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

Inst Memory

nPC_sel=+4

Inst Address

4

Adder

Mux

PC

00

Adder

PC Ext

clk

imm16

Single Cycle Datapath during Jump

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

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

Jump=

nPC_sel=

RegDst =

RegWr =

Instruction Fetch Unit

Clk

Instruction<31:0>

<21:25> <16:20> <11:15> <0:15> <0:25>

Rt Rs Rd Imm16 TA26

MemtoReg =

ALUctr =

Zero MemWr =

RegDst =

ALUSrc =

ExtOp =

22

Single Cycle Datapath during Jump

J-type:

31 26	25 0
op	**target address**	jump

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

Jump=1

nPC_sel=?

Instruction Fetch Unit

Instruction<31:0>

Clk

<21:25> <16:20> <11:15> <0:15> <0:25>

Rt   Rs   Rd   Imm16   TA26

RegDst = x

Rd   Rt

1 **Mux** 0

RegWr = 0

Rs   Rt

5   5   5

ALUctr =x

MemtoReg = x

busW

32
Clk

Rw   Ra   Rb

32 x 32-bit Registers

busA

32

busB

32

0

Mux

1

ALU

Zero

MemWr = 0

32

0

Mux

1

imm16

16

Extender

32

Data In 32

WrEn Adr

Data Memory

32

Clk

ALUSrc = x

ExtOp = x

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 }

Jump →

nPC_sel →

Zero →

Inst Memory → Instruction<31:0>

Adr

nPC_MUX_sel

4→ Adder

0 Mux 1

00 PC

Clk

imm16

Adder

Adder

How do we modify this to account for jumps?

24

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 }

Jump → **Inst Memory** → Instruction<31:0>

nPC_sel →

Zero →

nPC_MUX_sel

Adr

4 →

Adder

Mux 0 / 1

imm16

Adder

TA **26** **00**

4 (MSBs)

Mux 1 / 0

PC **00**

Clk

Query

- Can Zero still get asserted?

- Does nPC_sel need to be 0?
  - If not, what?

25
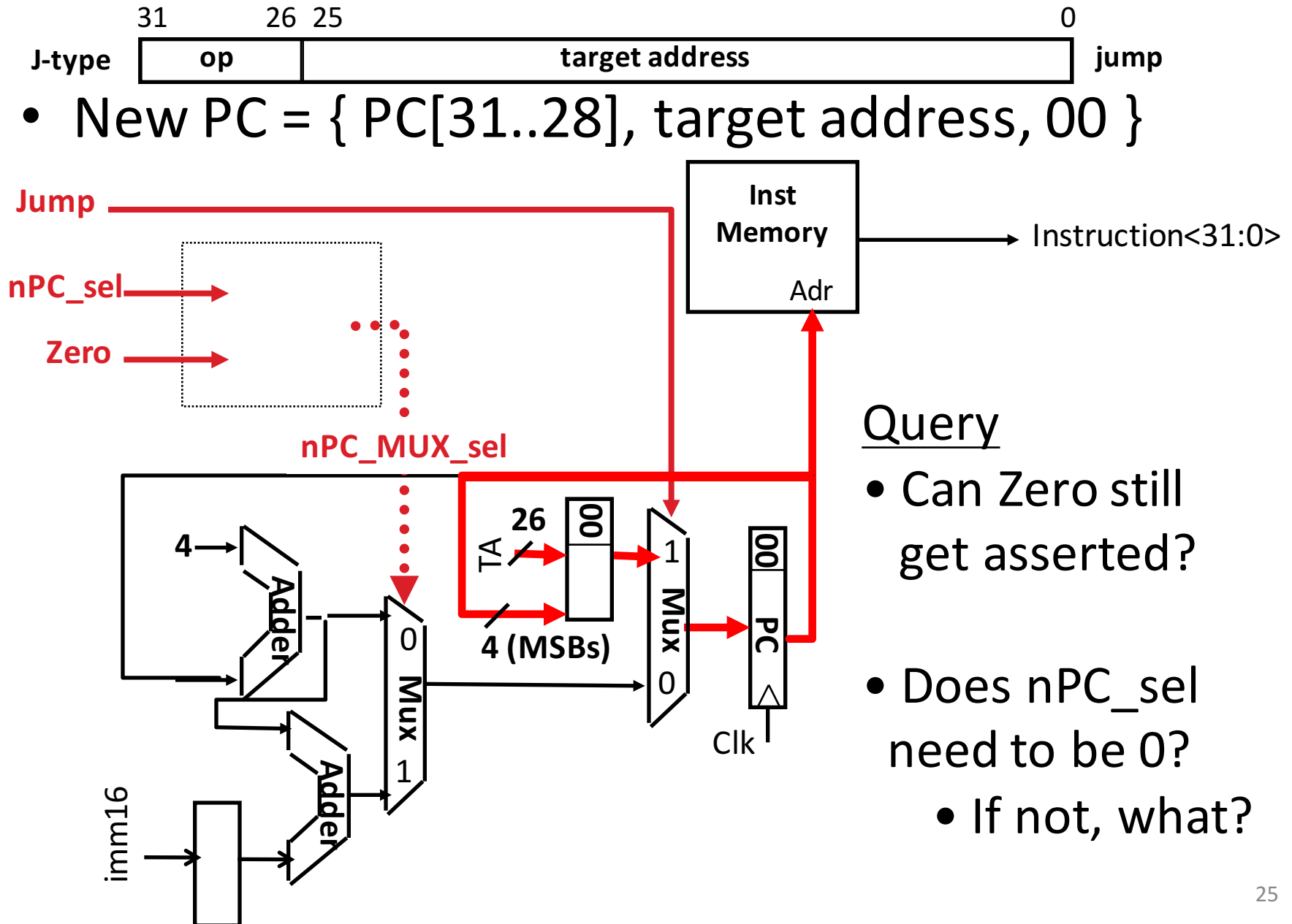
Clicker Question

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*

In The News: Tile-Mx100
100 64-bit ARM cores on one chip
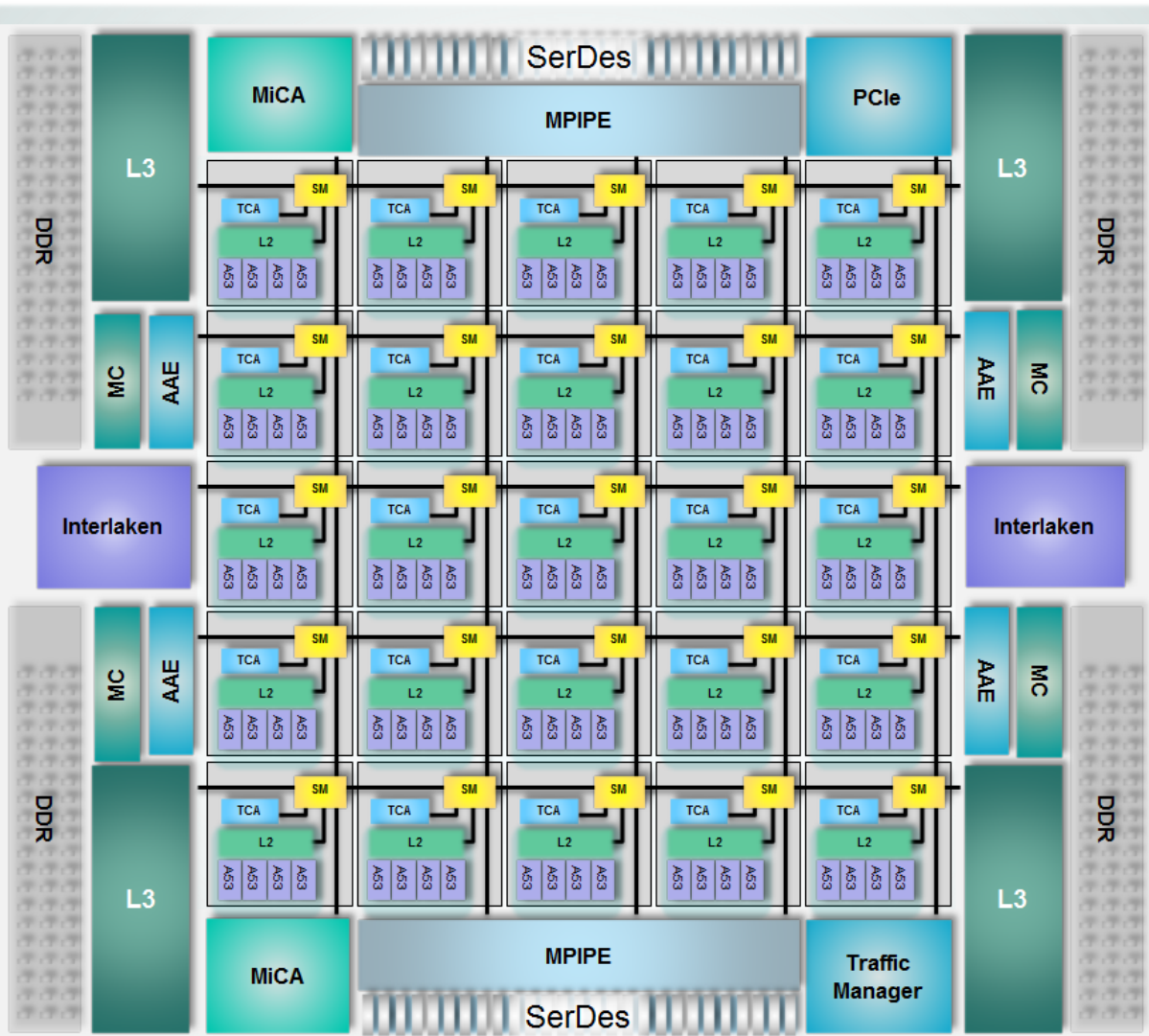
EZChip (bought Tilera)

100 64-bit ARM Cortex A53
- Dual-issue, in-order

27

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 ⟶ func	10 0000	10 0010	**We Don't Care :-)**				
Appendix A ⟶ 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

	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
```
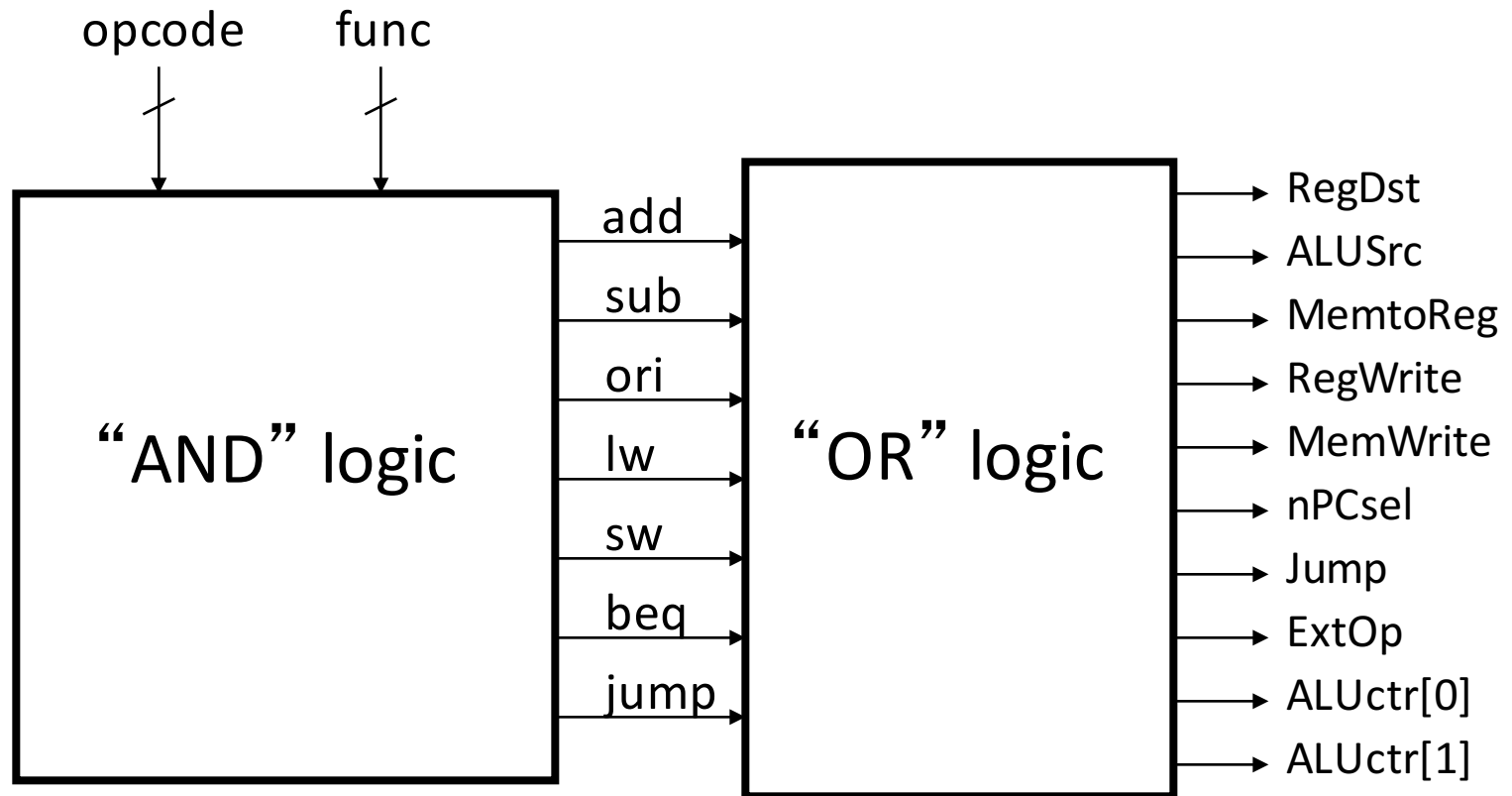
Where:

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

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

opcode   func

"AND" logic

add
sub
ori
lw
sw
beq
jump

"OR" logic

RegDst
ALUSrc
MemtoReg
RegWrite
MemWrite
nPCsel
Jump
ExtOp
ALUctr[0]
ALUctr[1]

P&H Figure 4.17
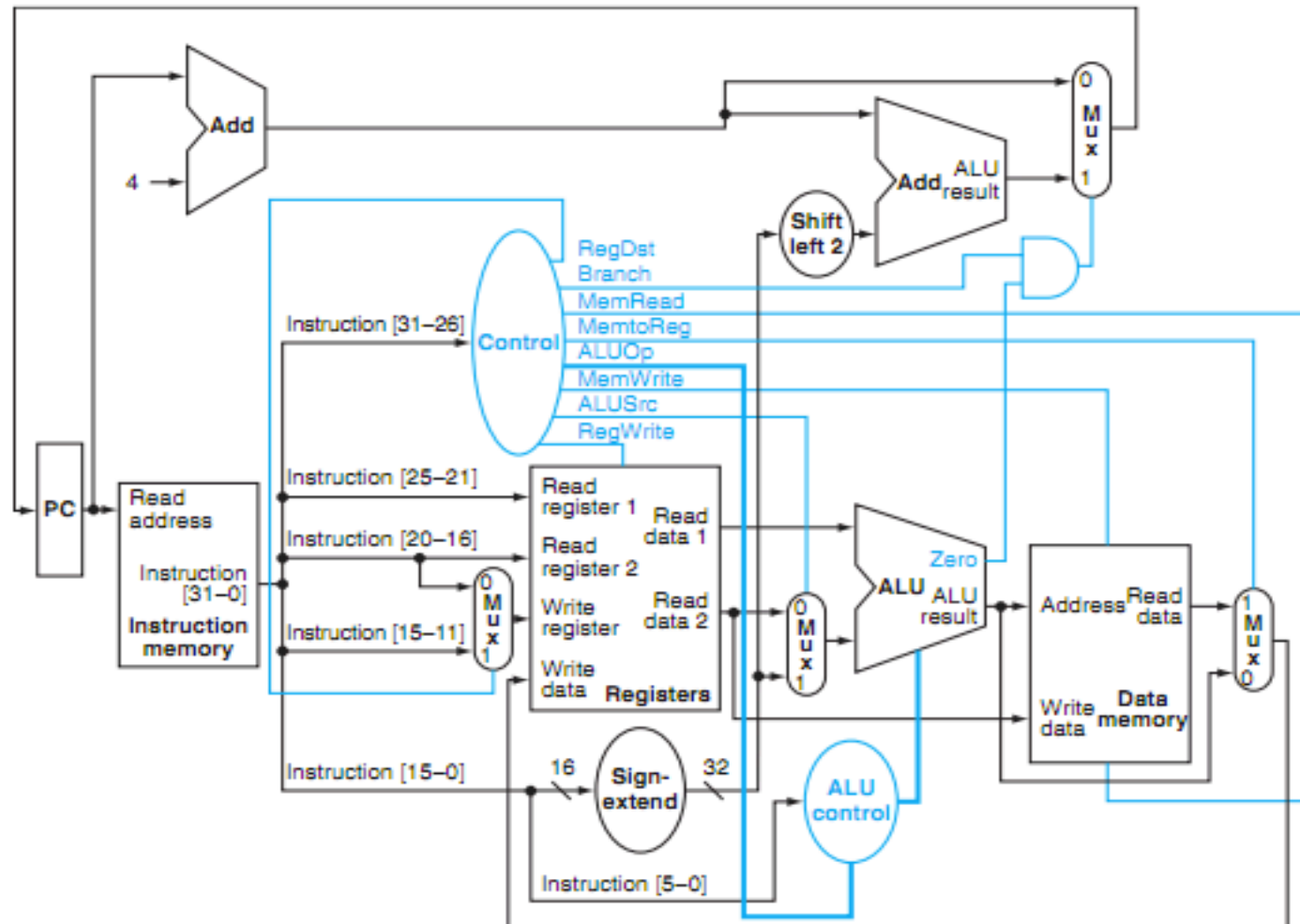
Summary: Single-cycle Processor

- 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
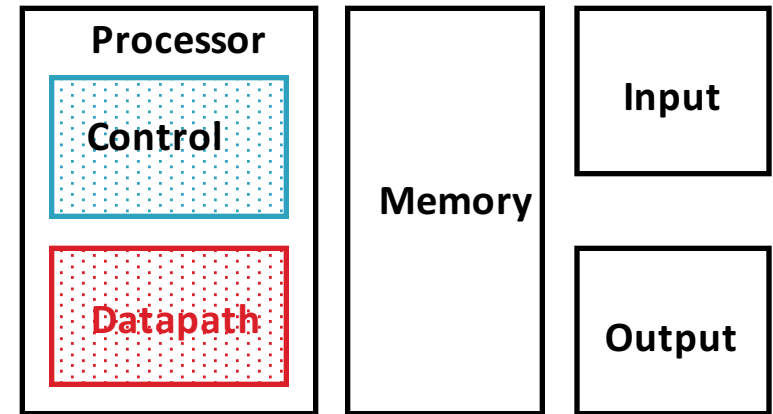
Levels of Representation/Interpretation

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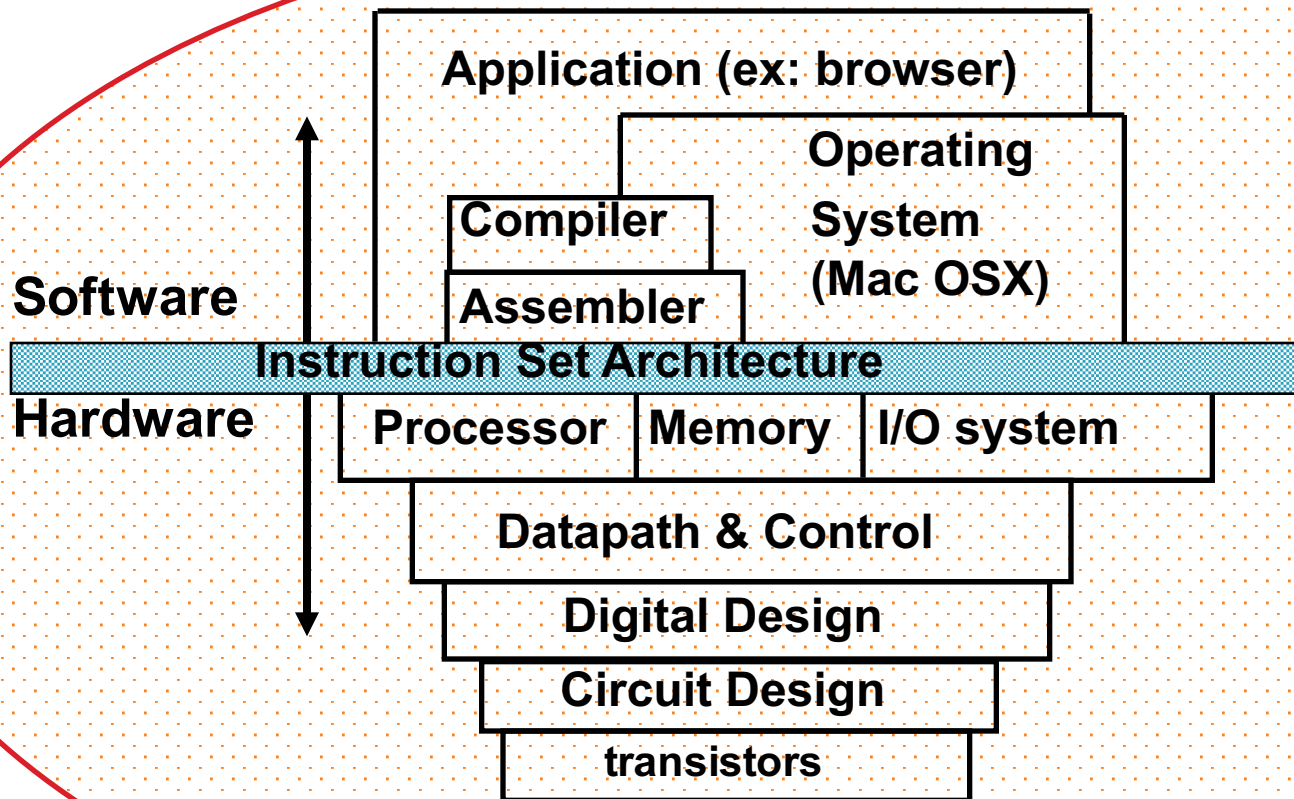
```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

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
```
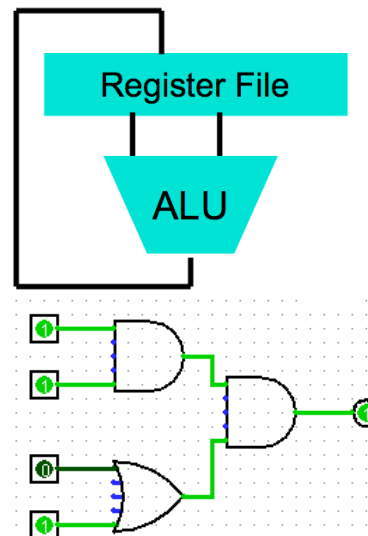
No More Magic!

CS61A
CS61B
CS61C ✔
CS61C ✔
CS61C ✔
CS61C ✔
CS61C ←
CS61C ✔
EE40
Phys 7B