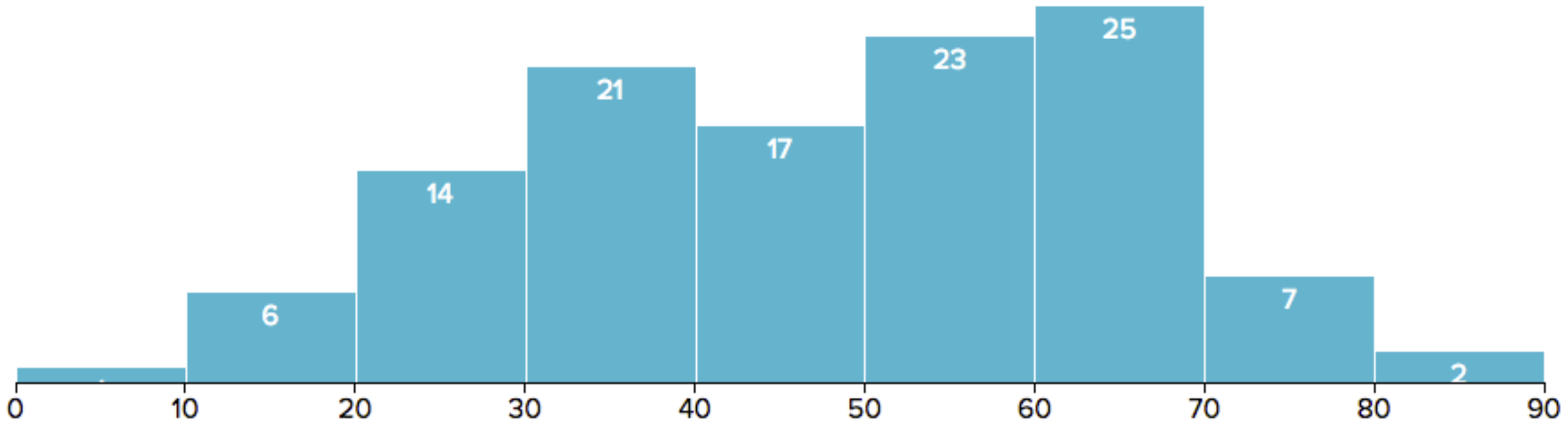
CS 61C: Great Ideas in Computer Architecture

Lecture 12: *Single-Cycle CPU, Datapath & Control Part 2*

Instructor: Sagar Karandikar
sagark@eecs.berkeley.edu

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

Midterm 1 Results



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
7.5	49.25	85.5	47.53	17.43

- You may submit regrade requests by Wednesday @ 23:59:59
- Solutions posted on Piazza

If you didn't do as well as you'd hoped

- You can still get an A with the clobber
  - 3 days preceding the final: there are no assignments, labs and discussions are OH
- Lots of resources to help:
  - 12 hours of OH/week
    - Go earlier in the week for conceptual questions (or come to mine)
  - Guerrilla section every Thursday (goes over exam-style problems)
  - You may go to multiple discussions (good to hear things from multiple perspectives)

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

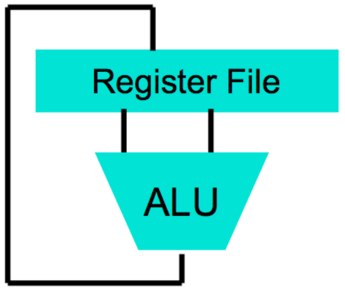
Logic Circuit Description (Circuit Schematic Diagrams)

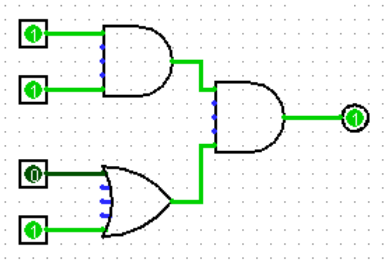
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

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

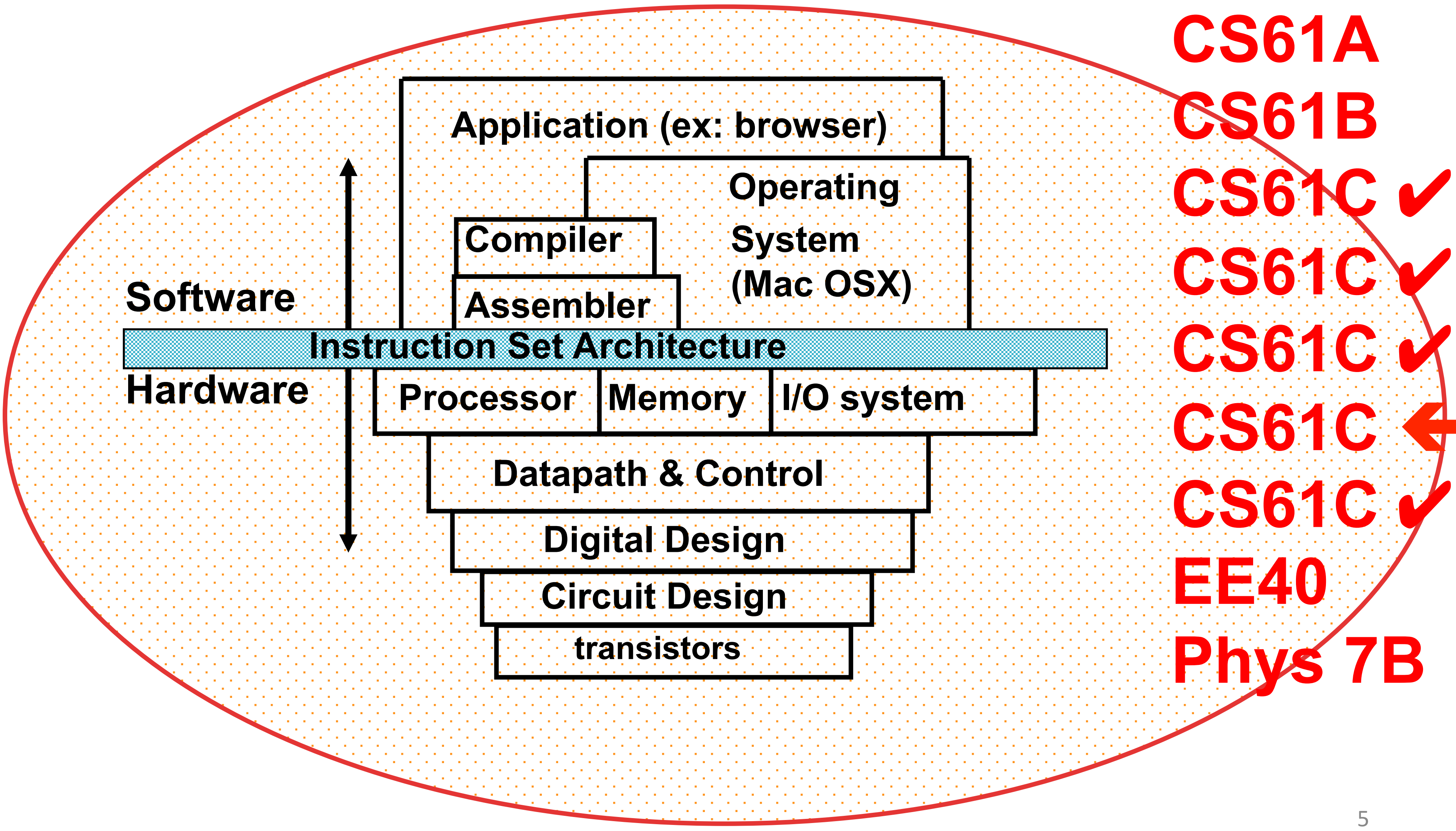
Anything can be represented as a *number*, i.e., data or instructions

```
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!



Last time: Processor Design: 3 of 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

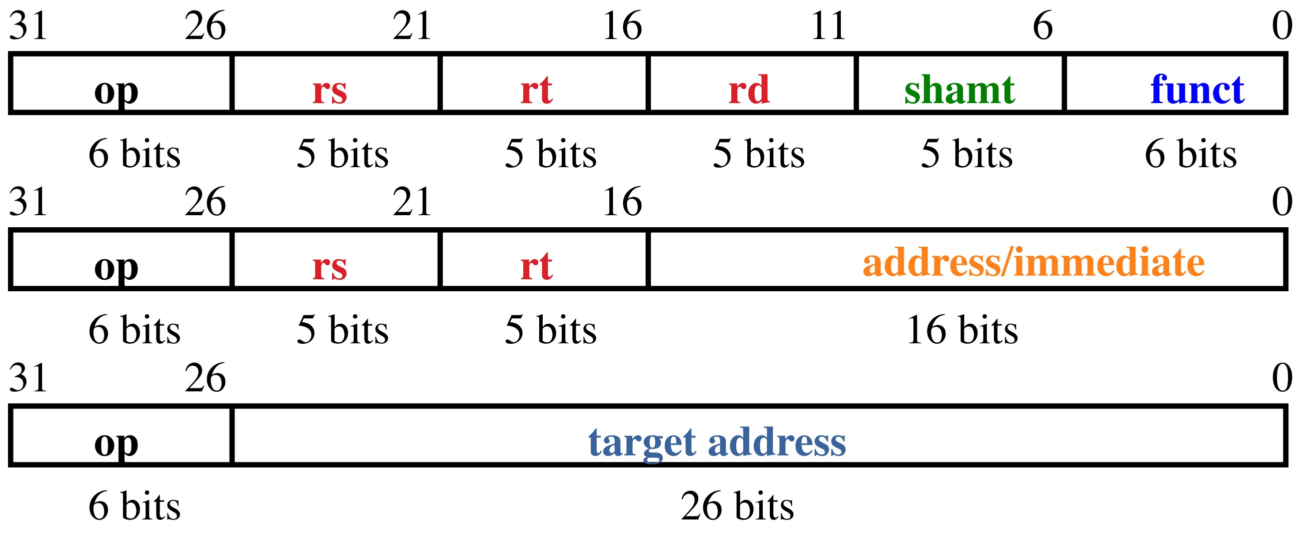
Step 3: Assemble datapath components that meet the requirements

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

Step 5: Assemble the control logic

Step 1: The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. 3 formats:
  - R-type
  - I-type
  - J-type



- The different fields are:
  - op: operation ("opcode") of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the "op" field
  - address / immediate: address offset or immediate value
  - target address: target address of jump instruction

Step 1: Register Transfer Level (RTL)

- Colloquially called “Register Transfer Language”
- RTL gives the meaning of the instructions
- All start by fetching the instruction itself

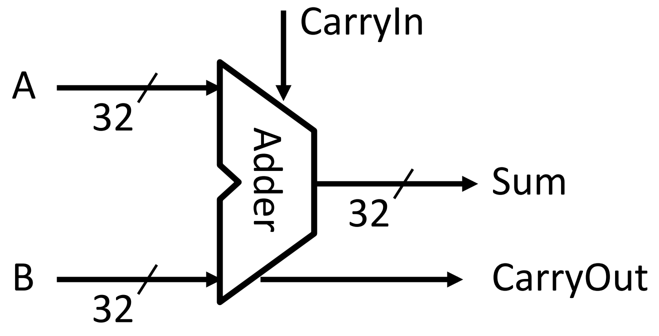
```
{op , rs , rt , rd , shamt , funct} ← MEM[ PC ]

{op , rs , rt ,    Imm16} ← MEM[ PC ]

Inst   Register Transfers

ADDU   R[rd] ← R[rs] + R[rt]; PC ← PC + 4

SUBU   R[rd] ← R[rs] – R[rt]; PC ← PC + 4

ORI    R[rt] ← R[rs] | zero_ext(Imm16); PC ← PC + 4

LOAD   R[rt] ← MEM[ R[rs] + sign_ext(Imm16)]; PC ← PC + 4

STORE  MEM[ R[rs] + sign_ext(Imm16) ] ← R[rt]; PC ← PC + 4

BEQ    if ( R[rs] == R[rt] )
           PC ← PC + 4 + {sign_ext(Imm16), 2’b00}
       else PC ← PC + 4
```

Step 1: Requirements of the Instruction Set

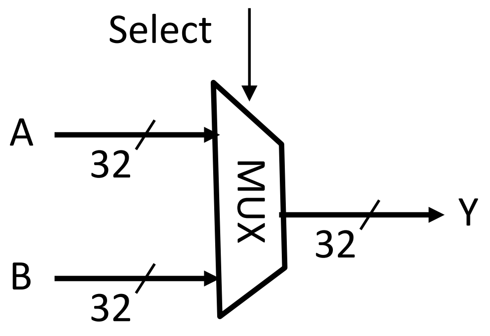
- Memory (MEM)
  - Instructions & data (will use one for each)
- Registers (R: 32, 32-bit wide registers)
  - Read RS
  - Read RT
  - Write RT or RD
- Program Counter (PC)
- Extender (sign/zero extend)
- Add/Sub/OR/etc unit for operation on register(s) or extended immediate (ALU)
- Add 4 (+ maybe extended immediate) to PC
- Compare registers?

Step 2: Components of the Datapath

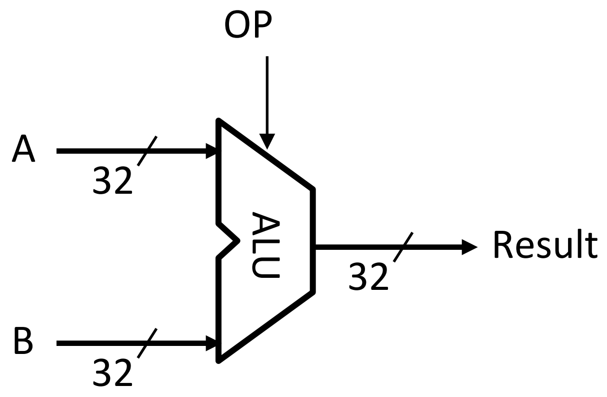
- Combinational Elements
- Storage Elements + Clocking Methodology
- Building Blocks



Adder



Multiplexer



ALU

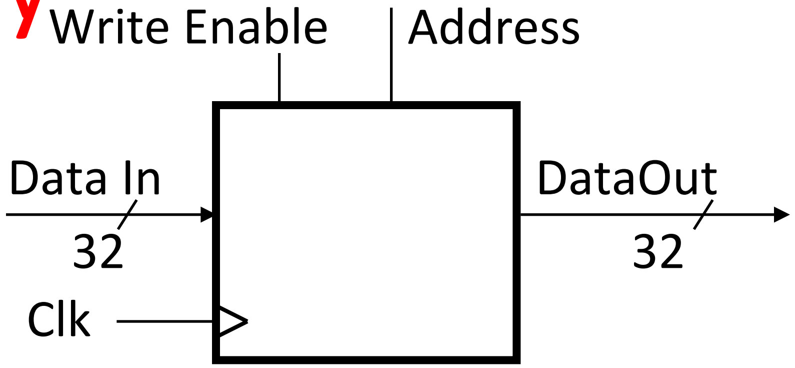
Step 2: ALU Needs for MIPS-lite + Rest of MIPS

- Addition, subtraction, logical OR, ==:

```
ADDU  R[rd] = R[rs] + R[rt]; ...
SUBU  R[rd] = R[rs] – R[rt]; ...
ORI   R[rt] = R[rs] | zero_ext(Imm16)...
BEQ   if ( R[rs] == R[rt] )...
```

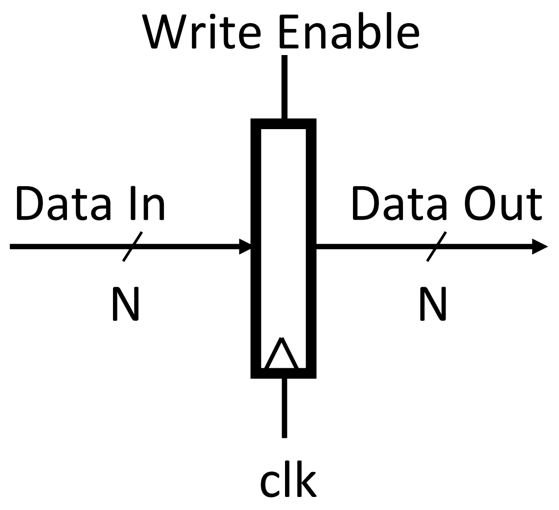
- Test to see if output == 0 for any ALU operation gives == test. How?
- P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)
- ALU follows Chapter 5

Step 2: Storage Element: Idealized Memory



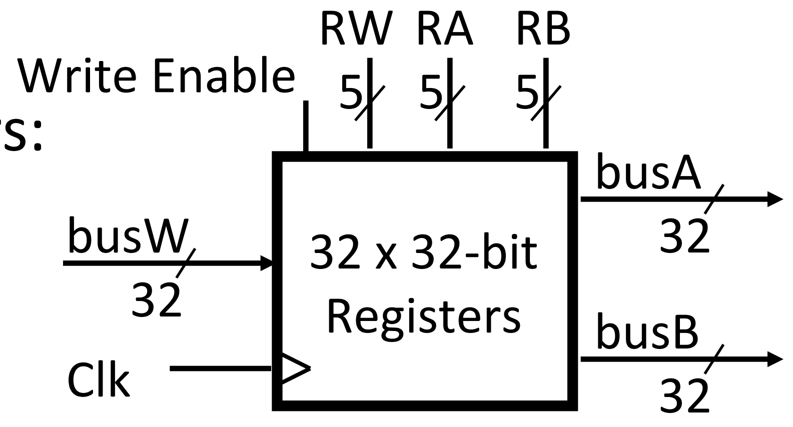
- "Magic" Memory
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - For Read: Address selects the word to put on Data Out
  - For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid $\Rightarrow$ Data Out valid after "access time"

Step 2: Storage Element: Register (Building Block)



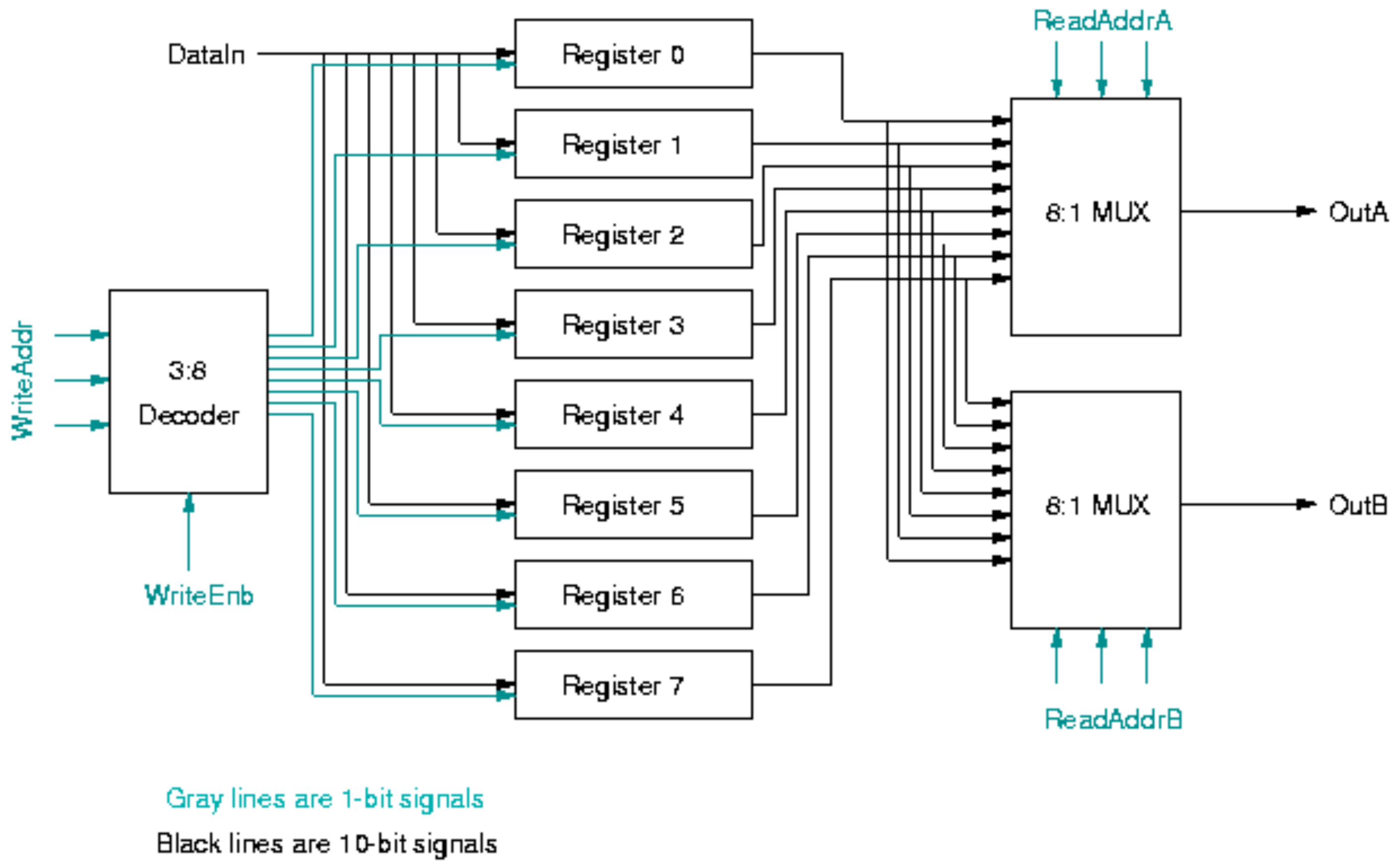
- Similar to D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - Negated (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on positive edge of clock

Step 2: Storage Element: Register File



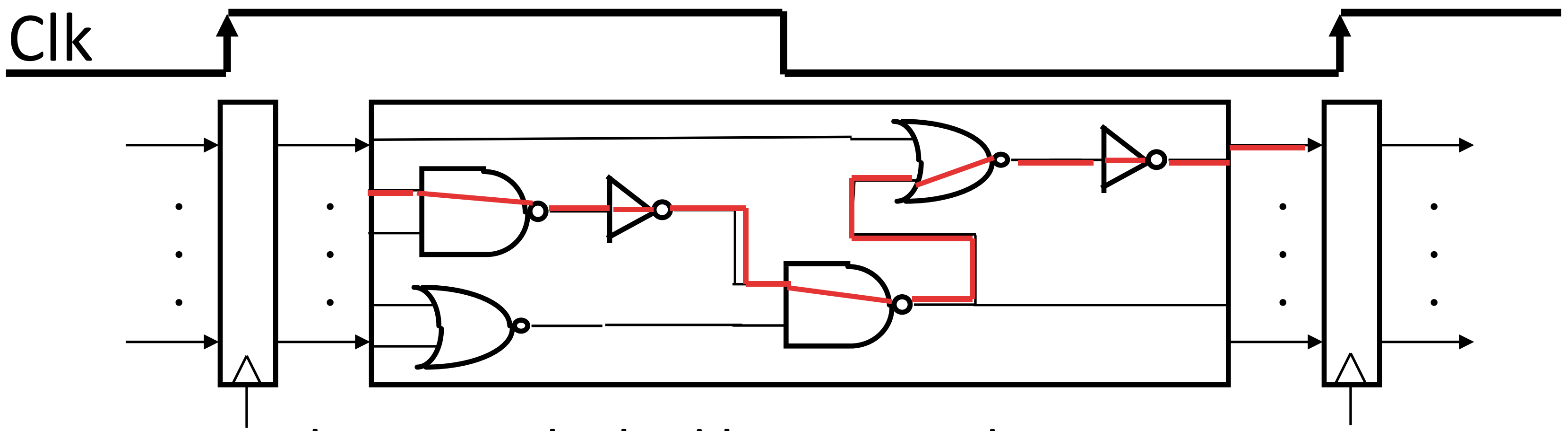
- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid $\Rightarrow$ busA or busB valid after "access time."

Example: RegFile with 8 Registers



https://jindongpu.files.wordpress.com/2012/03/register_file.gif

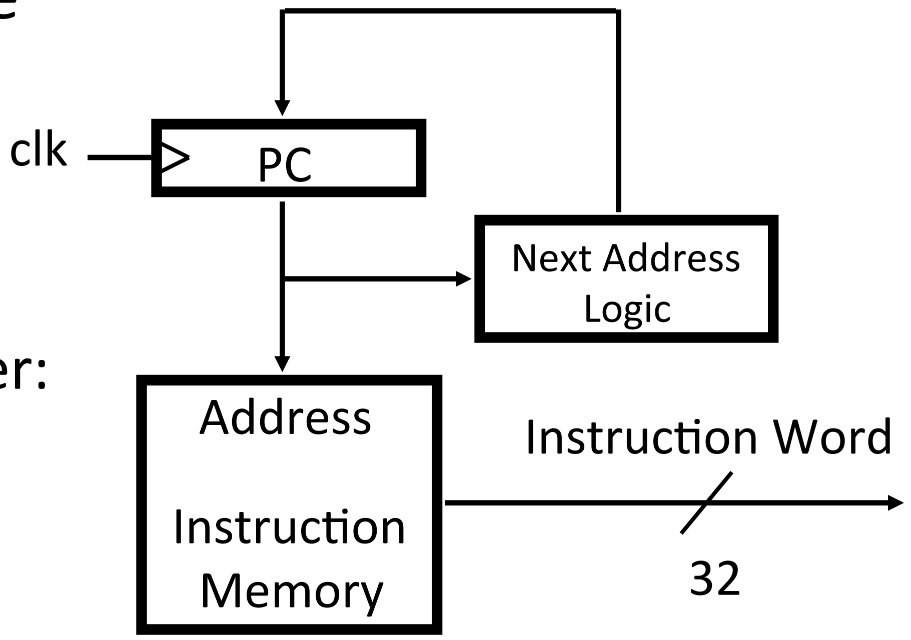
Step 2: Clocking Methodology



- Storage elements clocked by same edge
- Flip-flops (FFs) 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

Step 3a: Instruction Fetch Unit

- Register Transfer Requirements ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation
- Common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC ← PC + 4
    - Branch and Jump: PC ← "something else"

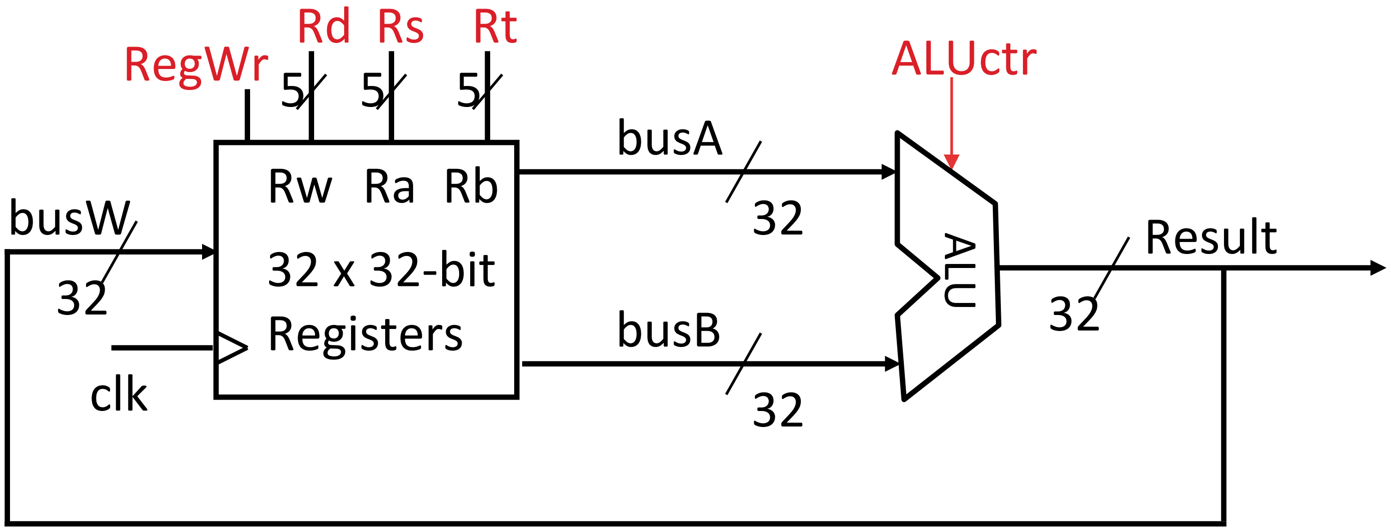


Step 3b: Add & Subtract

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

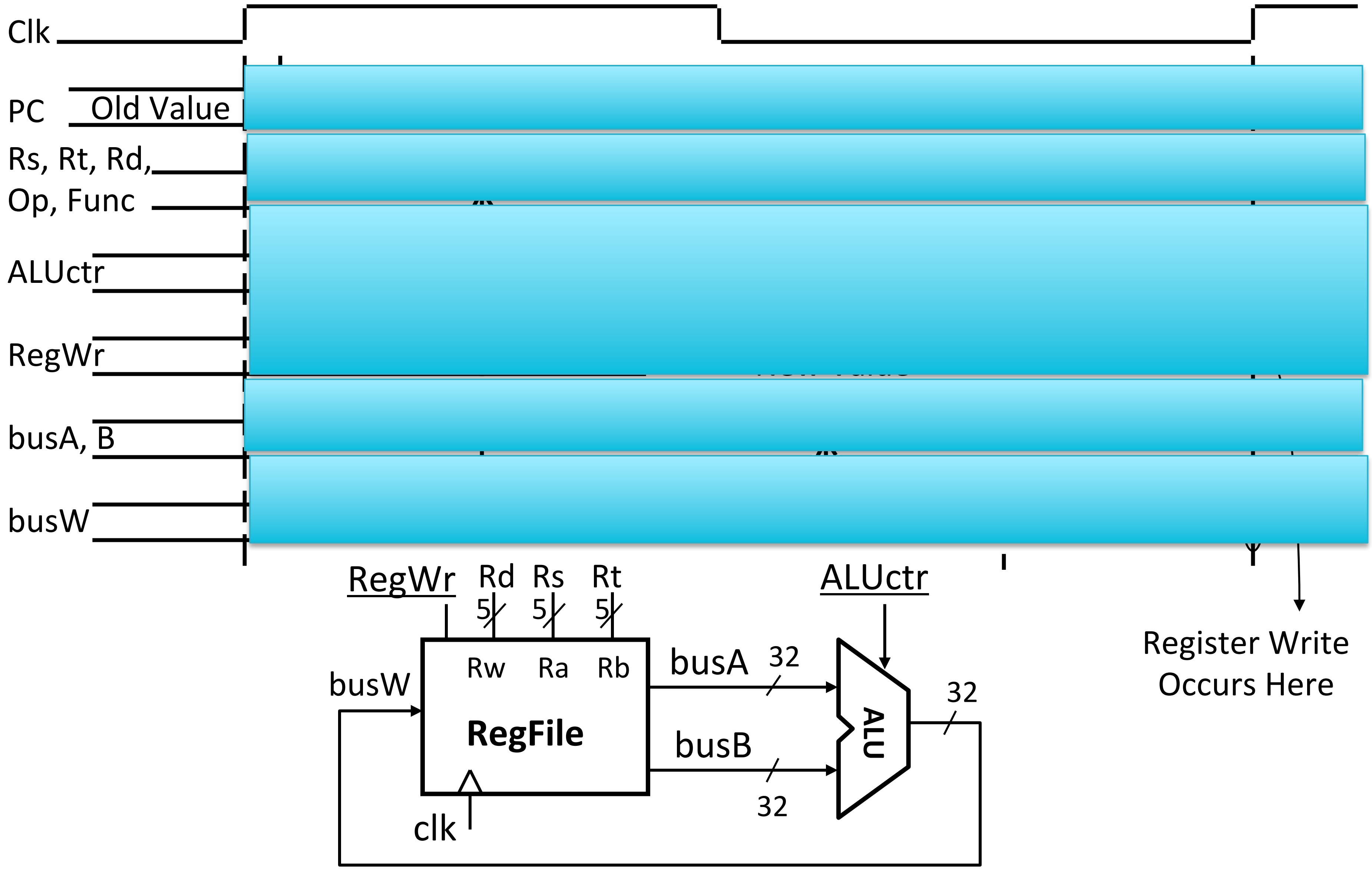
31 26	21	16	11	6	0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - ALUctr and RegWr: control logic after decoding the instruction



- … Already defined the register file & ALU

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



Peer Instruction

1. We should use the main ALU to compute PC=PC+4 in order to save some gates
2. The ALU is inactive for memory reads (loads) or writes (stores).

	1	2
A	F	F
B	F	T
C	T	F
D	T	T

Administrivia

- HW3 Out
  - Covers SDS topics from last week
- Proj 2-2 out
  - Make sure you test your code on hive machines, that's where we'll grade them

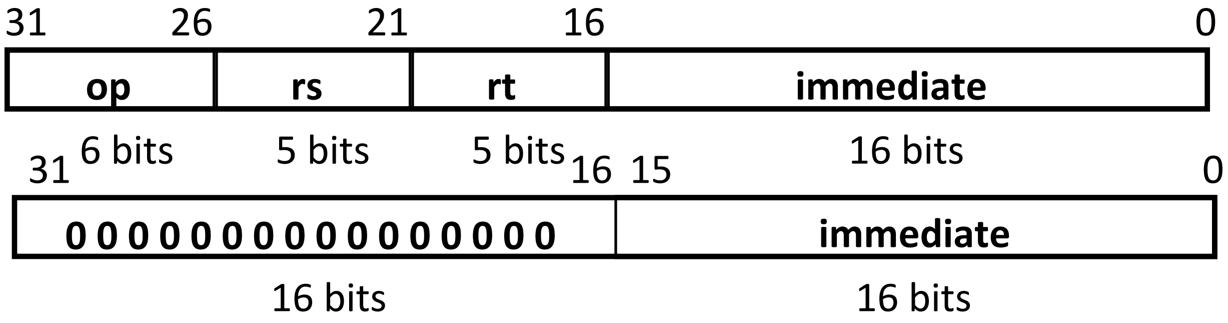
Break

End of the review

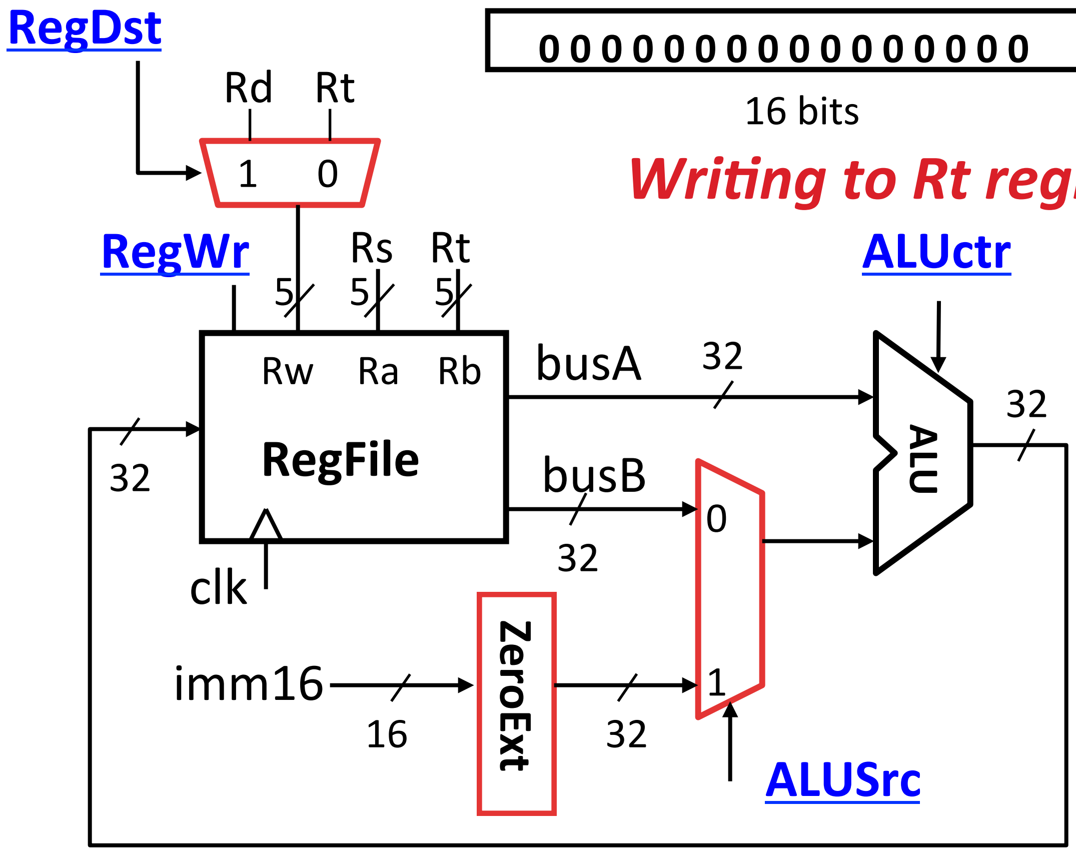
- Now let’s finish steps 3, 4, and 5
- By the end of today, you’ll know how an entire computer works!

3c: Logical Op (or) with Immediate

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



Writing to Rt register (not Rd)!!

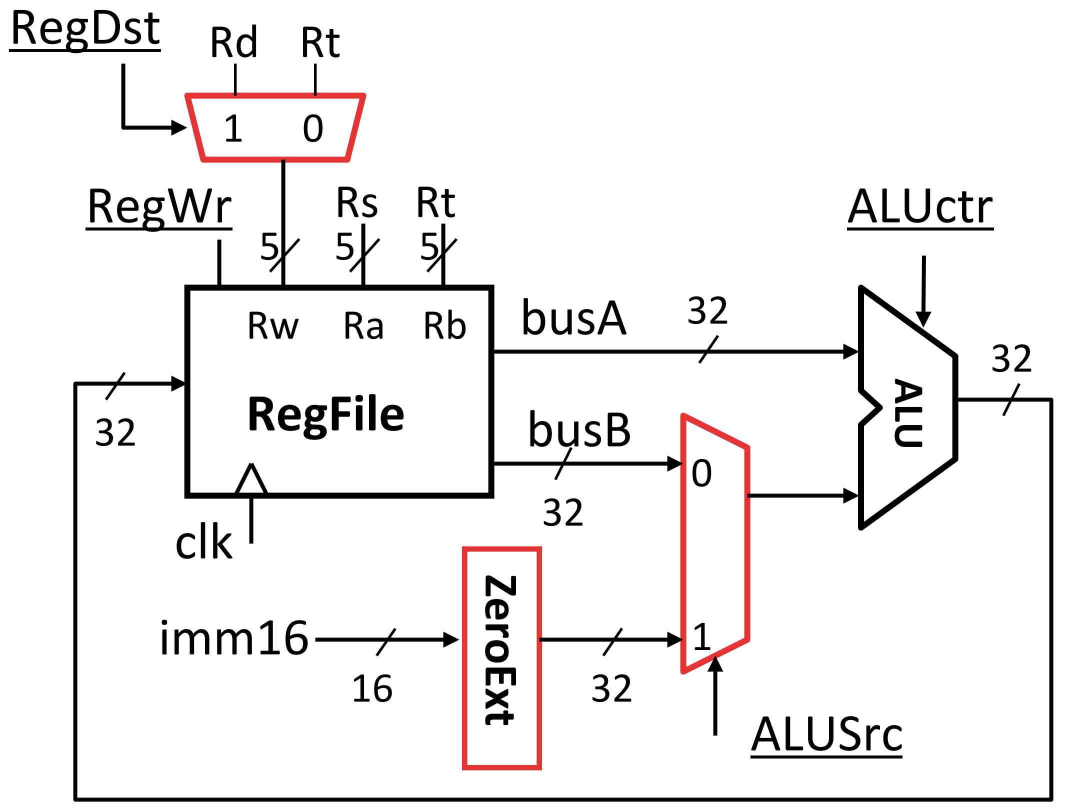


What about Rt Read?

3d: Load Operations

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

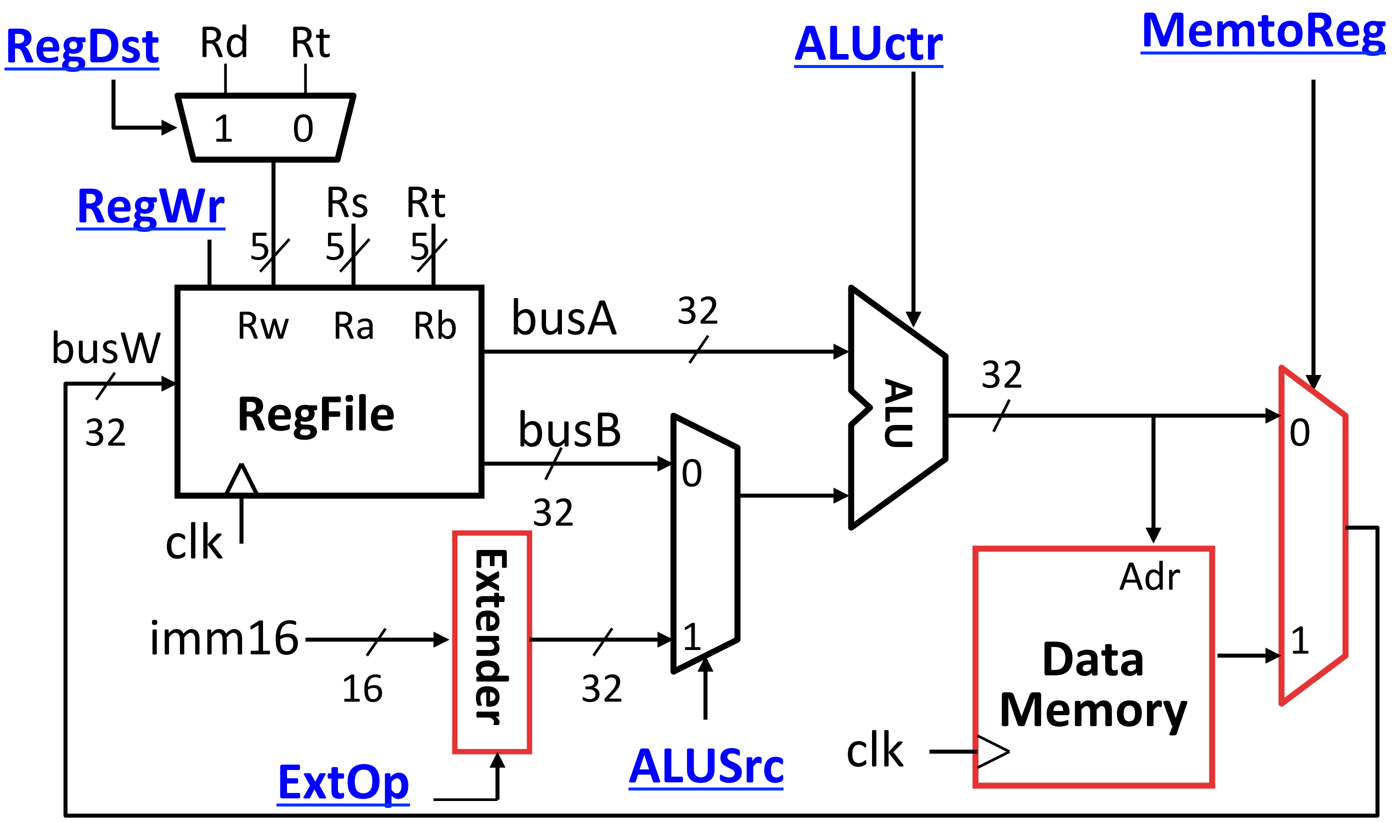
31 – 26	25 – 21	20 – 16	15 – 0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits



3d: Load Operations

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

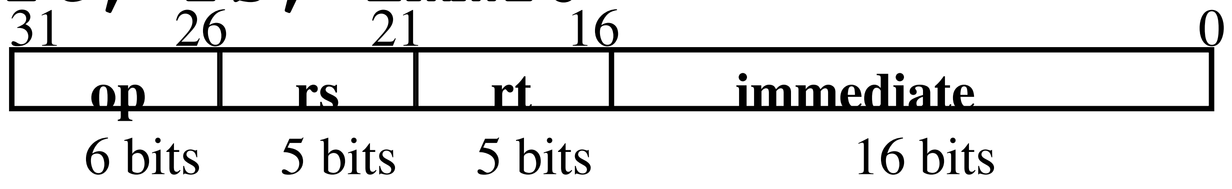
31 26	26 21	21 16	16 0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

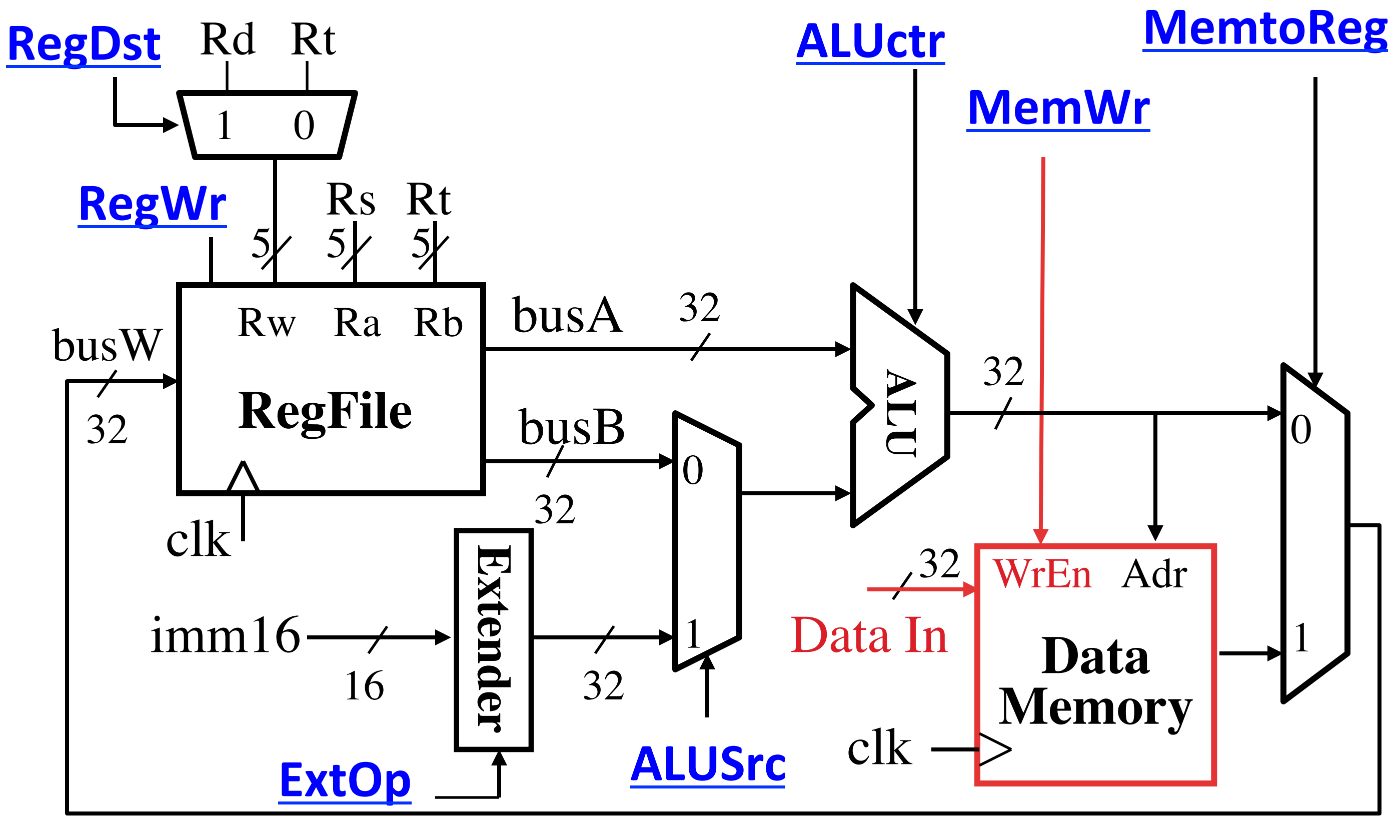


3e: Store Operations

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

  Ex.: `sw rt, rs, imm16`

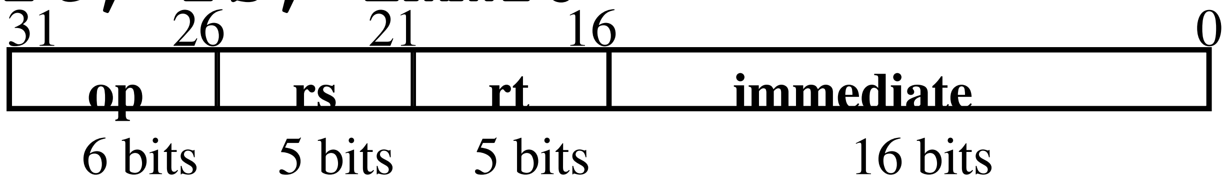


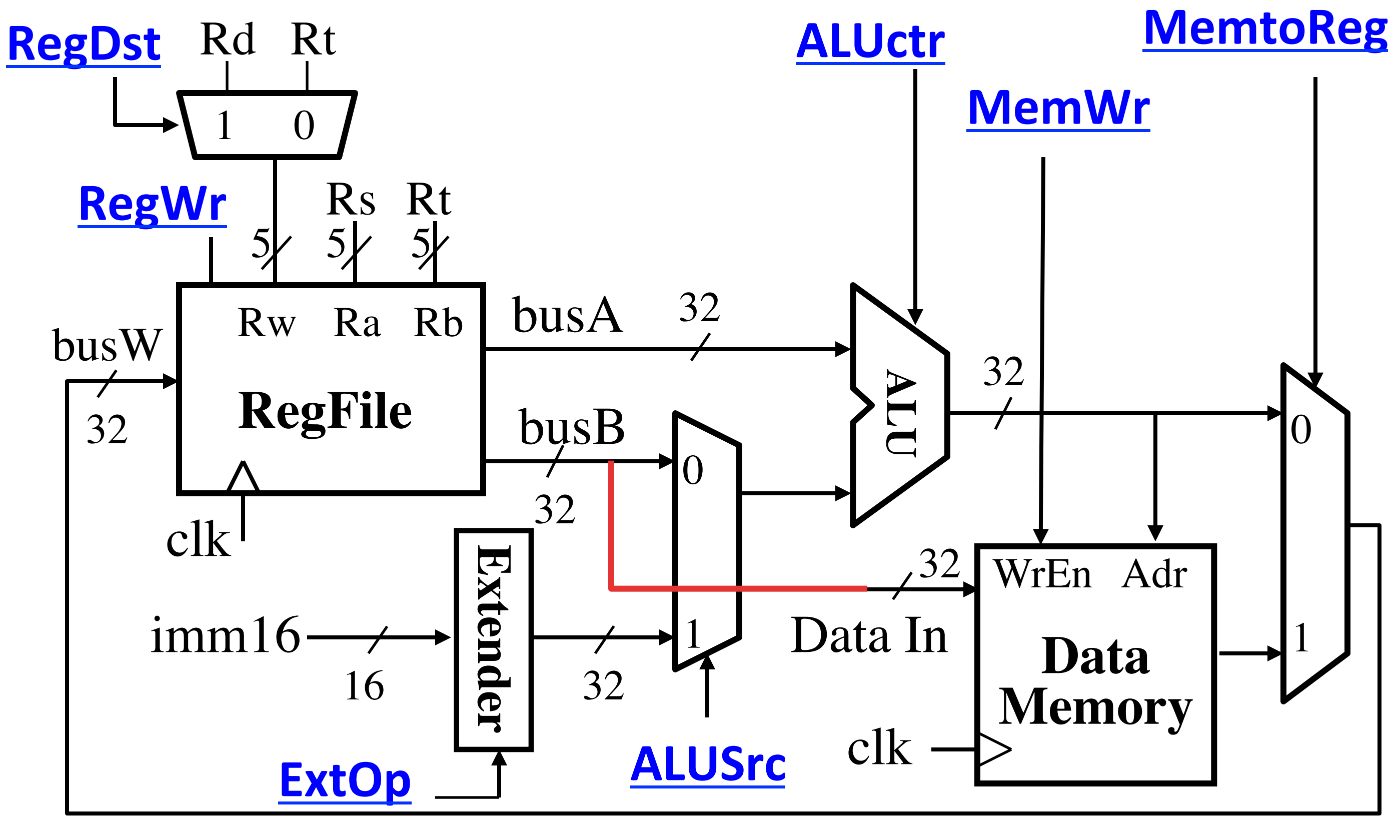


3e: Store Operations

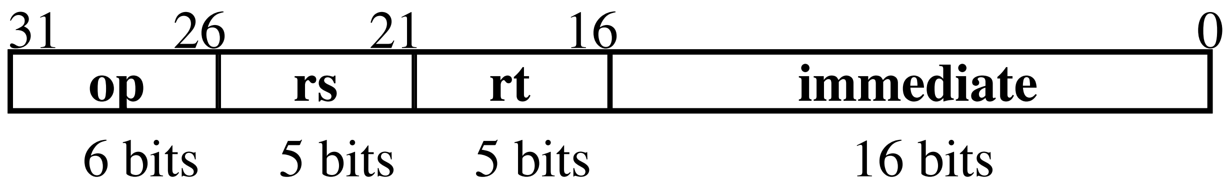
- Mem[R[rs] + SignExt[imm16]] = R[rt]

  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

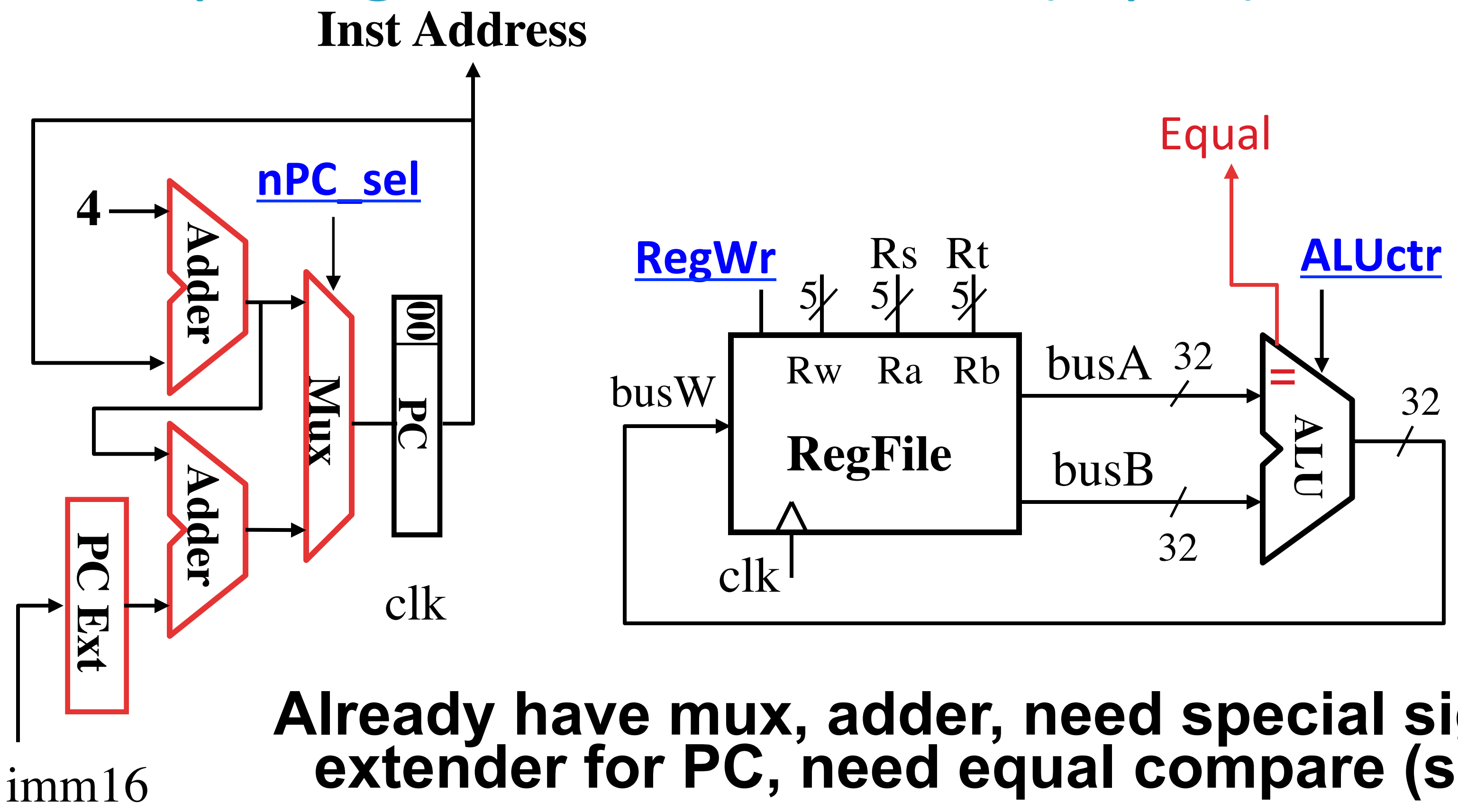
- PC = PC + 4

Datapath for Branch Operations

beq   rs, rt, imm16

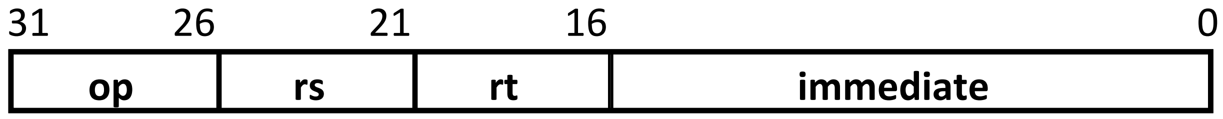
31 … 26	26 … 21	21 … 16	16 … 0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

Datapath generates condition (Equal)

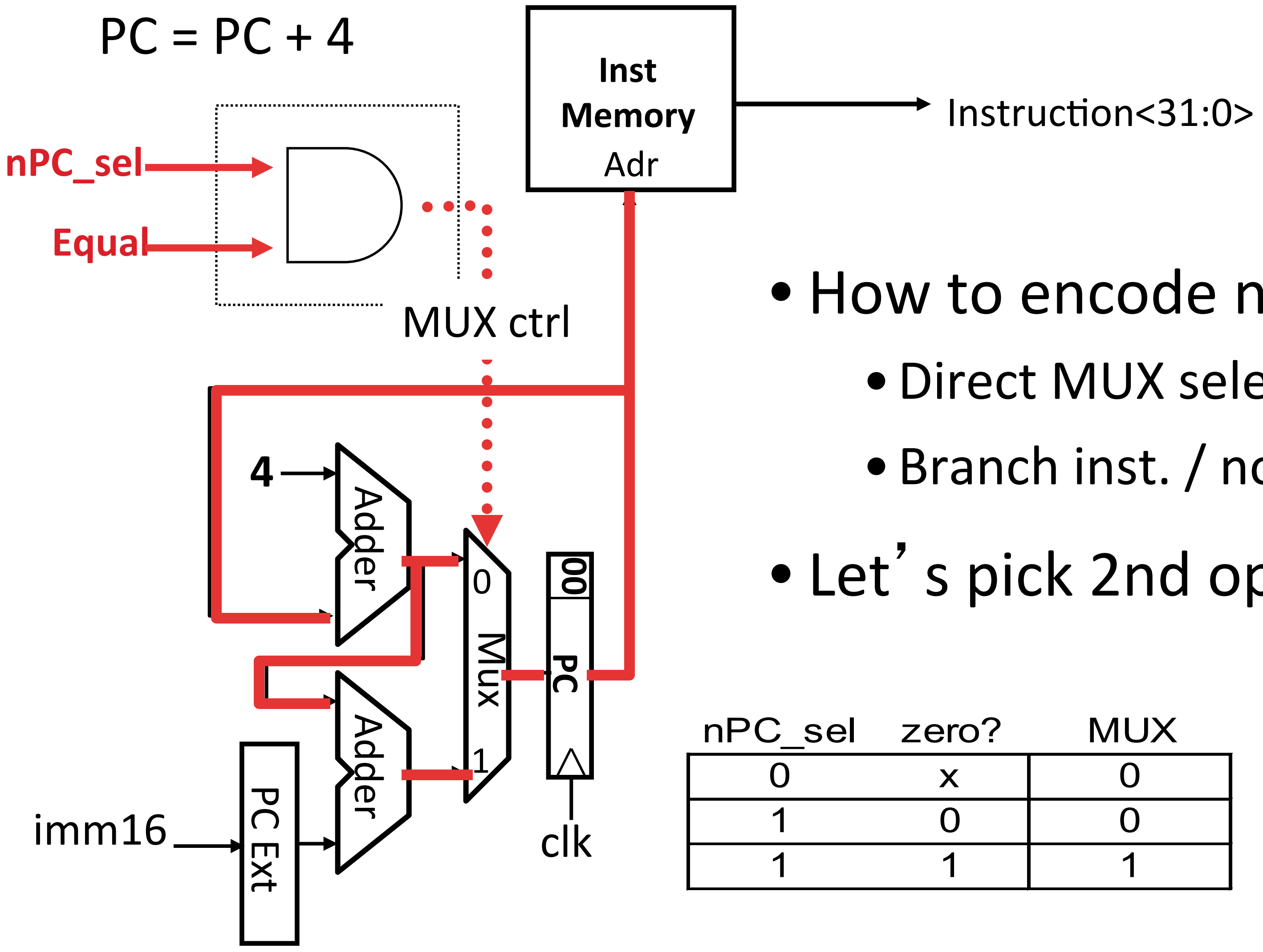


Already have mux, adder, need special sign extender for PC, need equal compare (sub?)

Instruction Fetch Unit including Branch



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

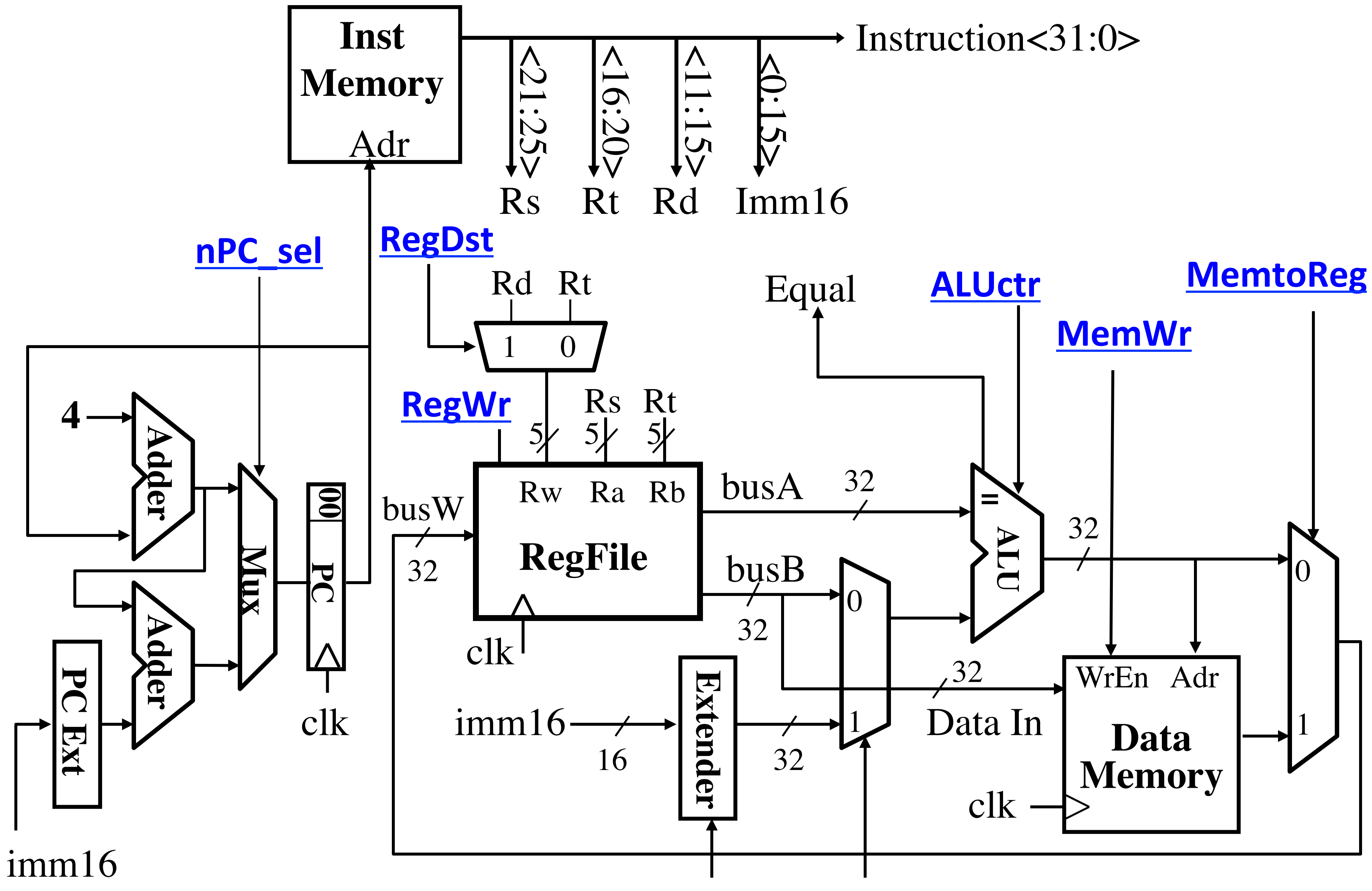


- How to encode nPC_sel?
  - Direct MUX select?
  - Branch inst. / not branch inst.
- Let's pick 2nd option

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



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

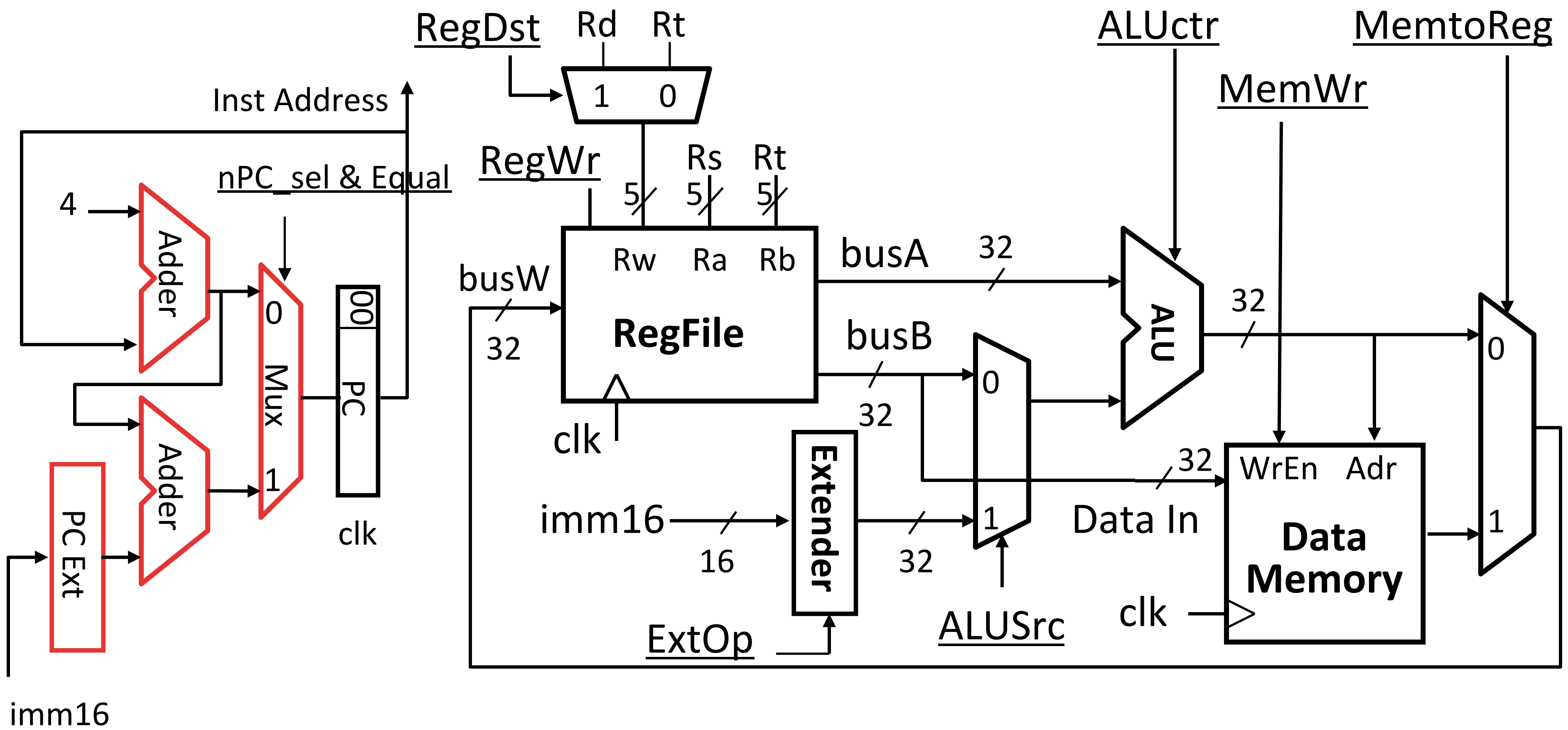
Step 3: Assemble datapath components that meet the requirements

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

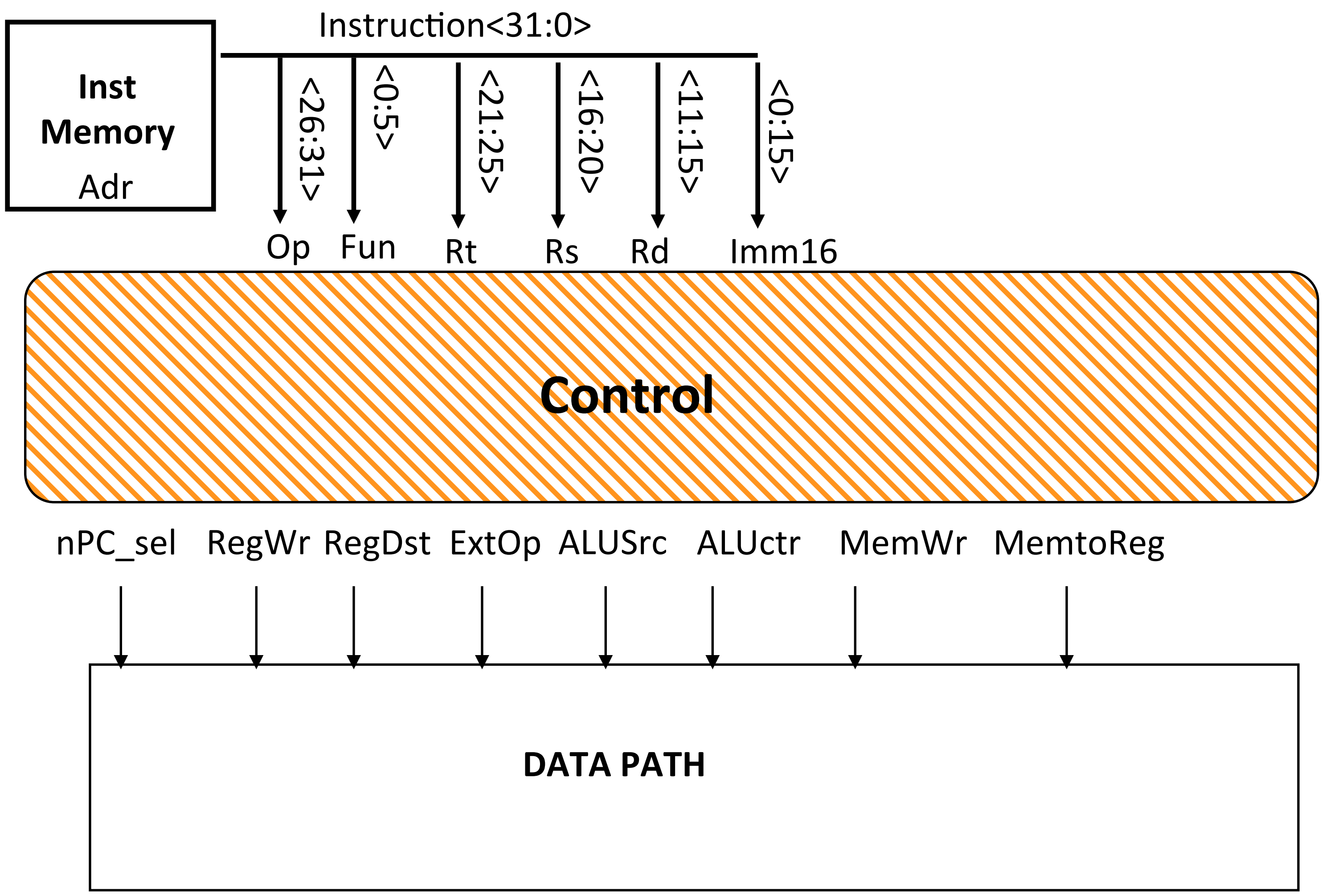
Step 5: Assemble the control logic

Datapath Control Signals

• ExtOp:	"zero", "sign"
• ALUsrc:	0 ⇒ regB; 1 ⇒ immed
• ALUctr:	"ADD", "SUB", "OR"
• 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

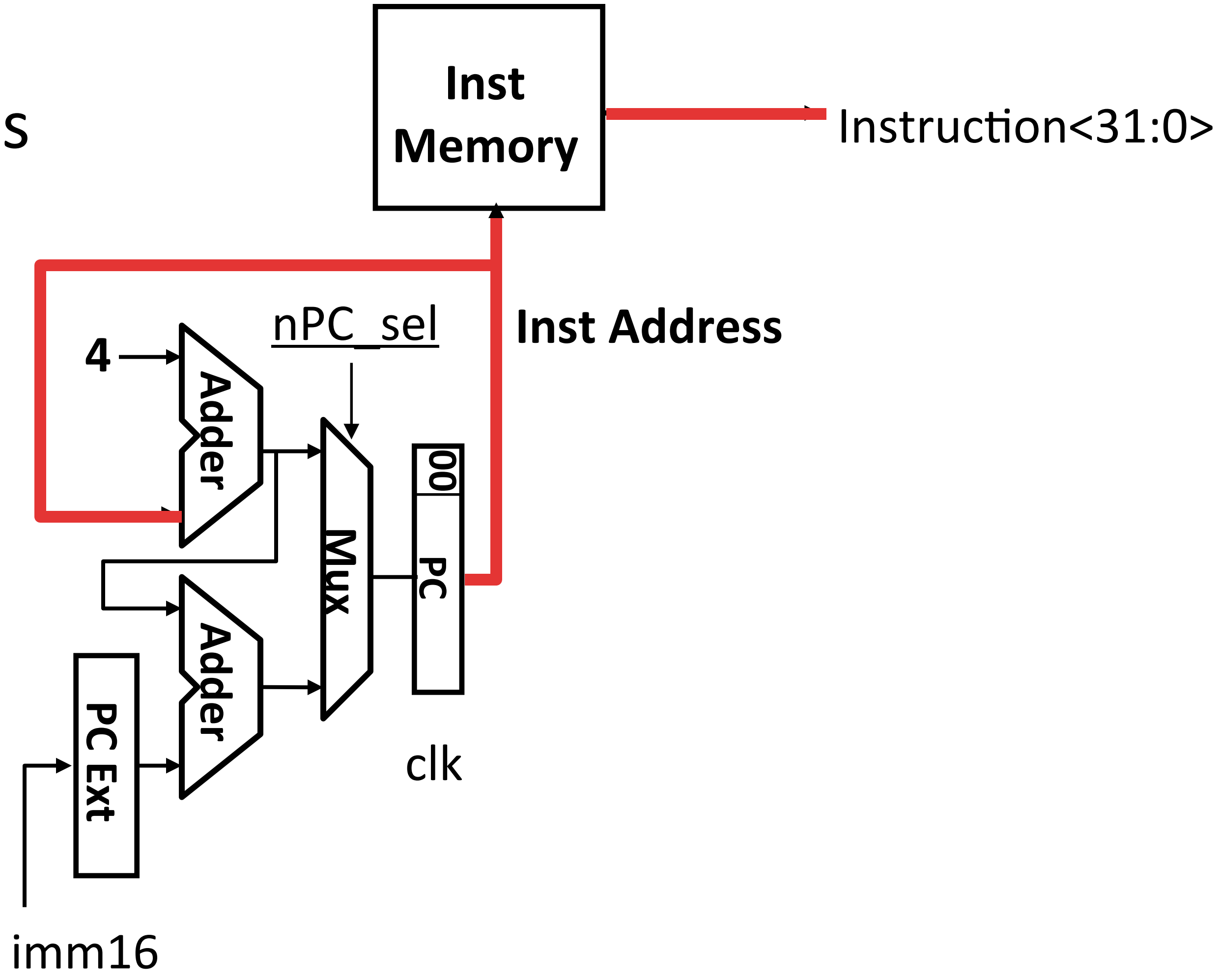
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

`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`

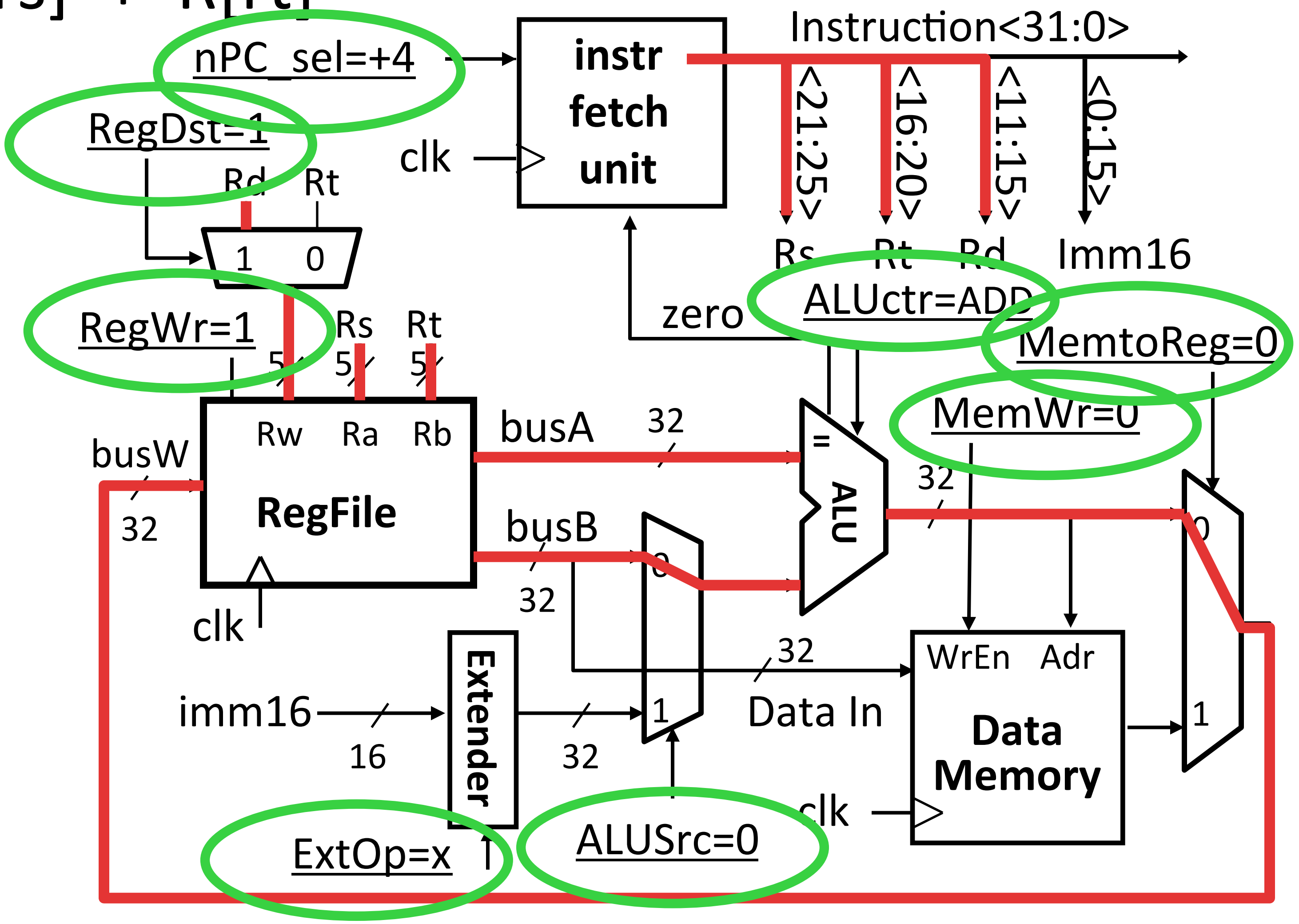
- Fetch the instruction from Instruction memory: Instruction = MEM[PC]
  - same for all instructions



Single Cycle Datapath during Add

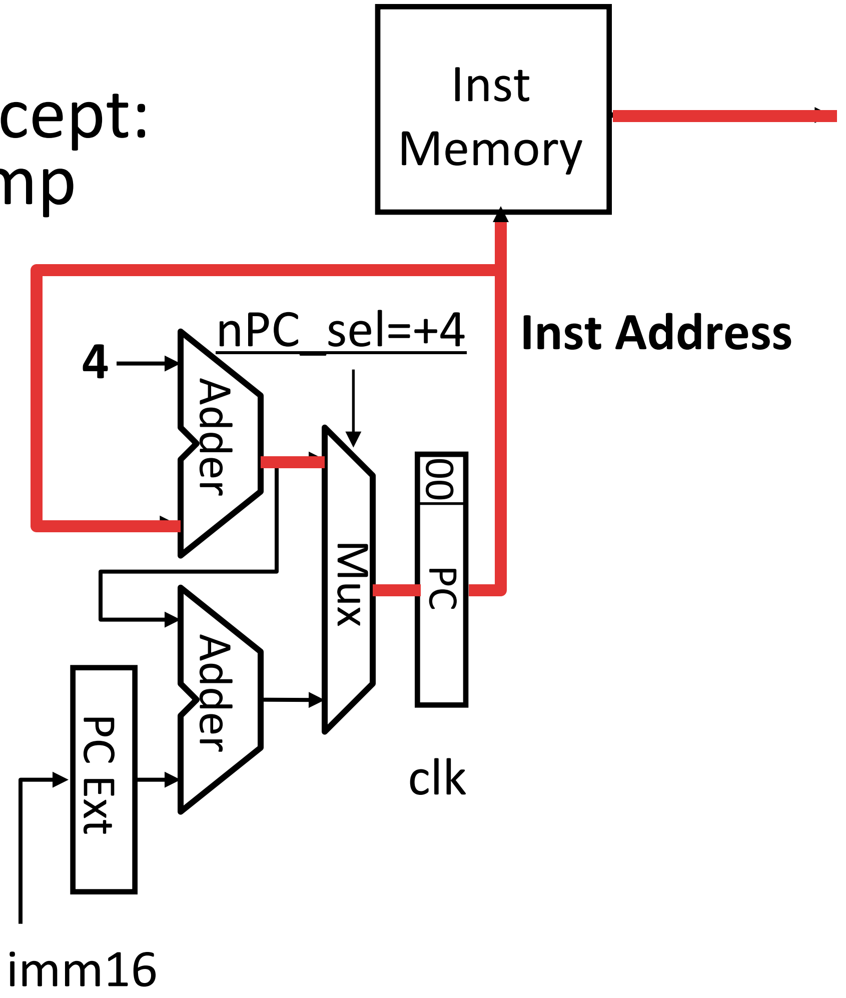
31	26	21	16	11	6	0
op	rs	rt	rd	shamt	funct	

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



Instruction Fetch Unit at End of `Add`

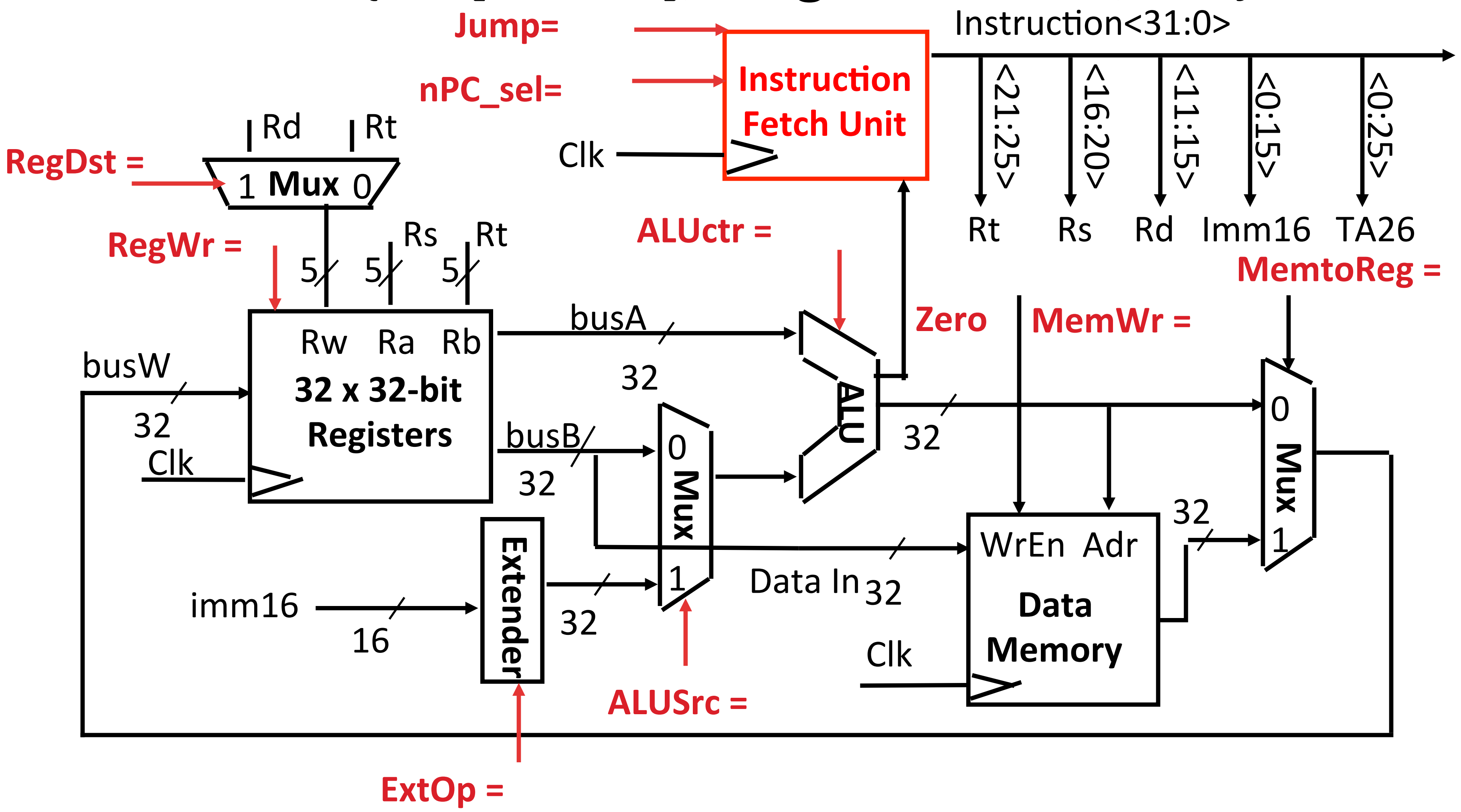
- PC = PC + 4
  - Same for all instructions except: Branch and Jump



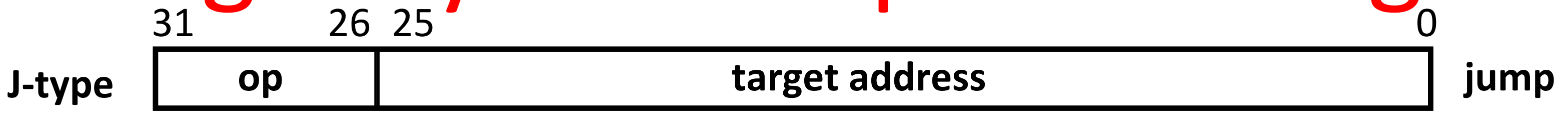
Single Cycle Datapath during Jump

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

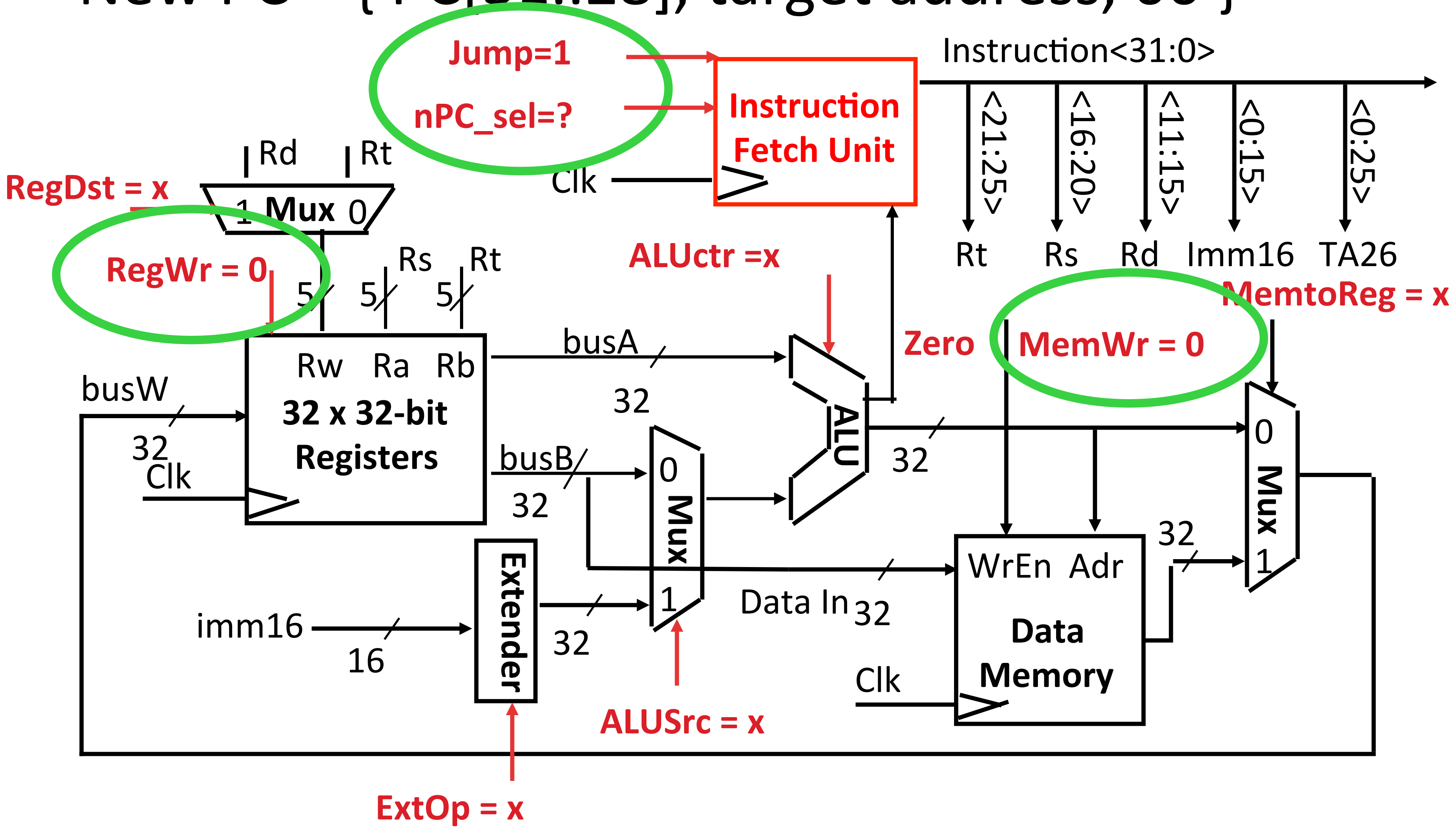
- New PC = { PC[31..28], target address, 00 }



Single Cycle Datapath during Jump



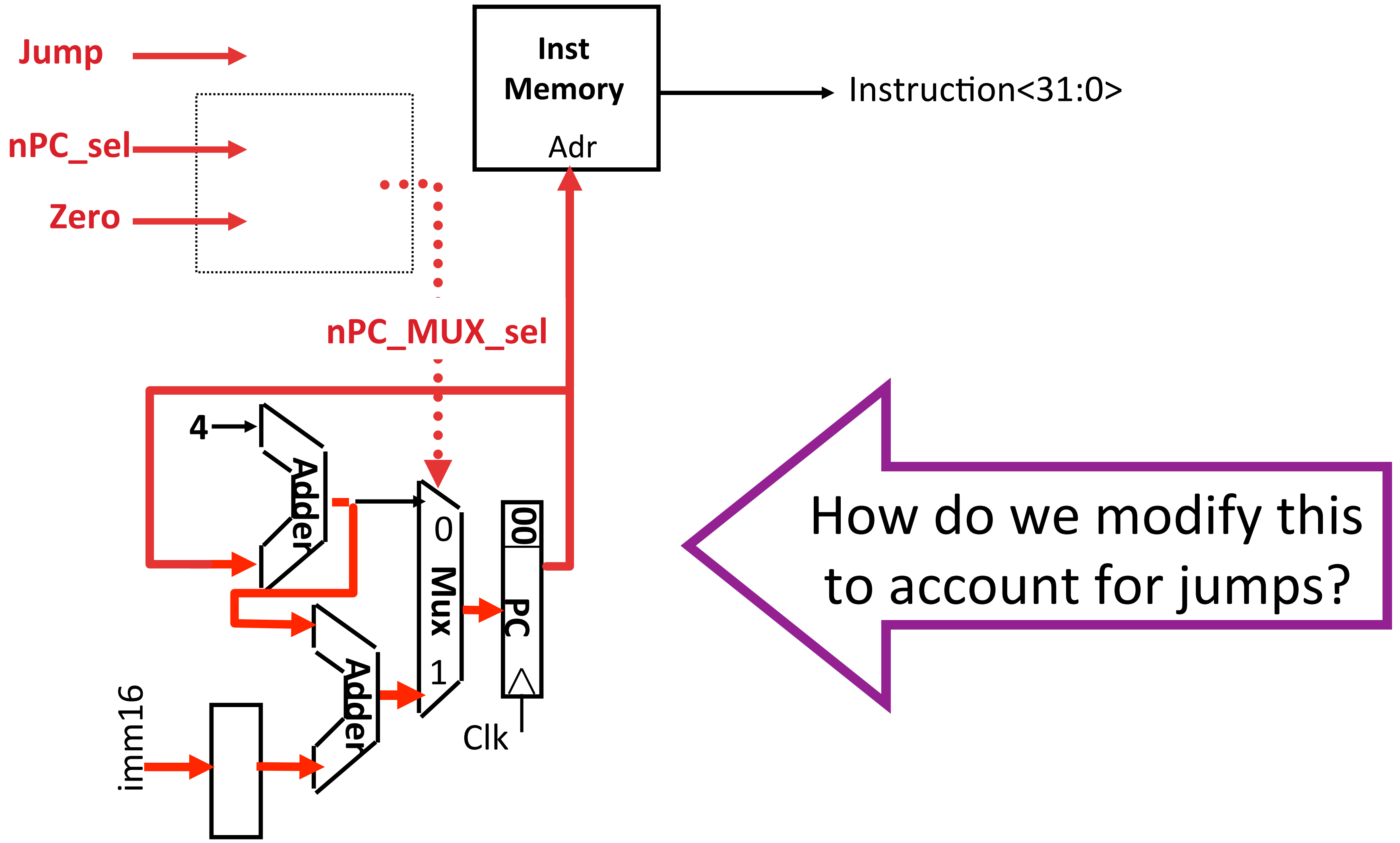
- New PC = { PC[31..28], target address, 00 }



Instruction Fetch Unit at the End of Jump

	31 26	25 0	
J-type	op	target address	jump

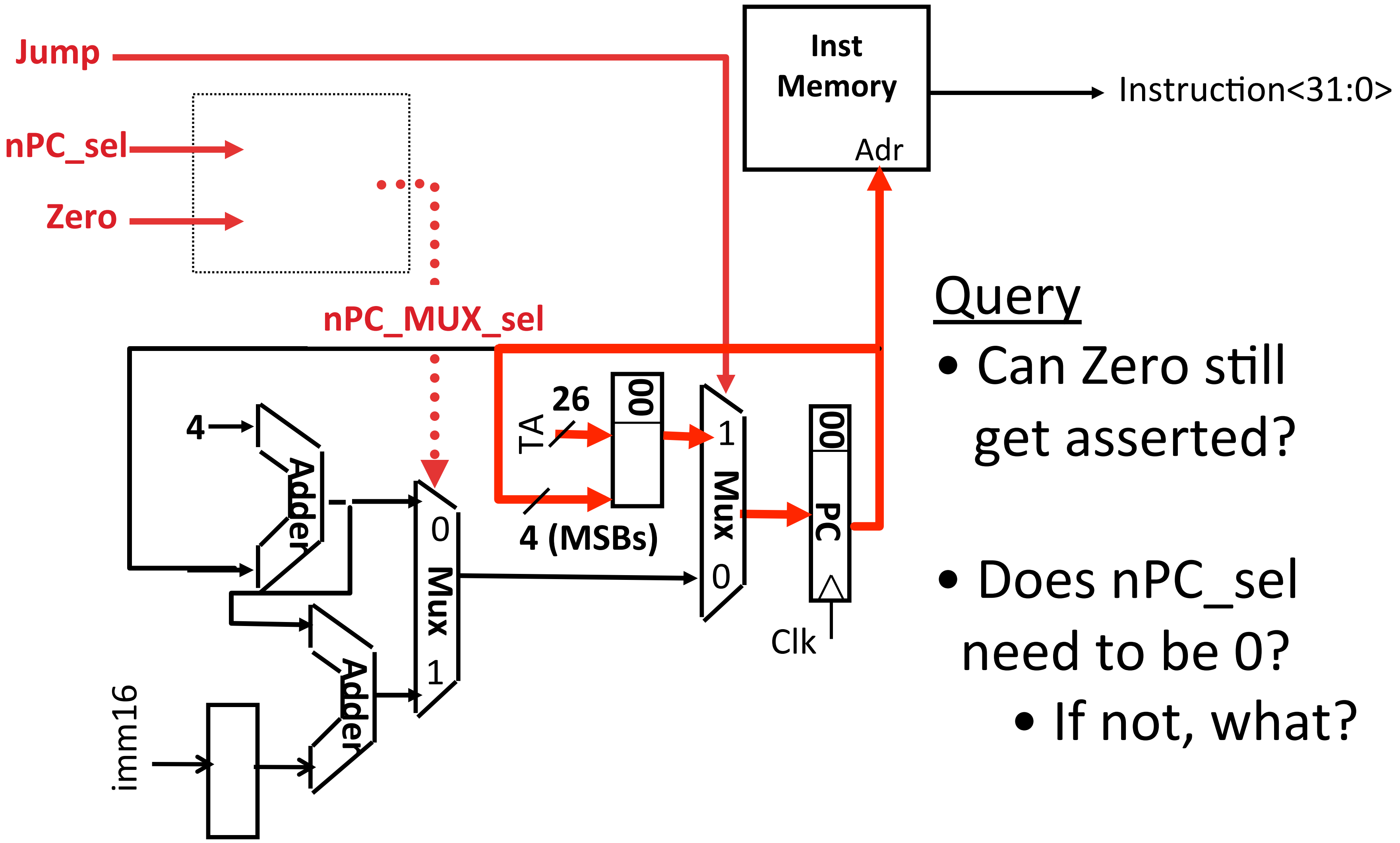
- New PC = { PC[31..28], target address, 00 }



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 }

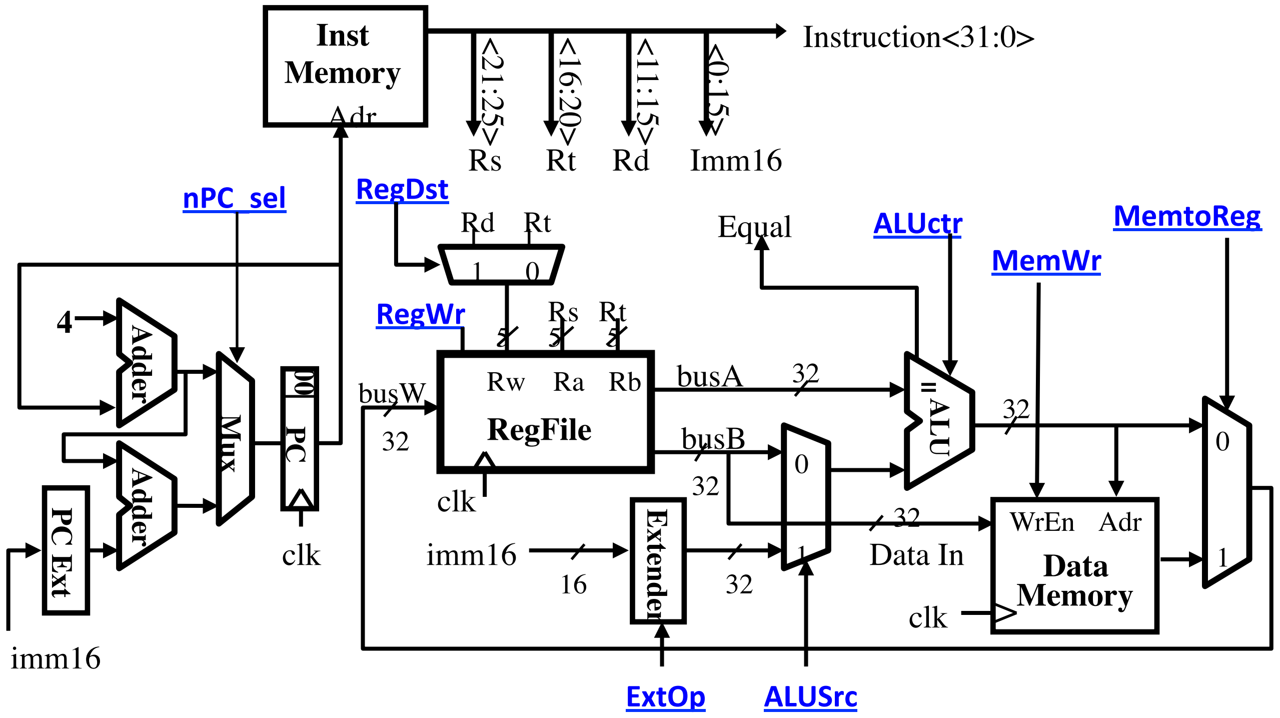


Query
- Can Zero still get asserted?
- Does nPC_sel need to be 0?
    - If not, what?

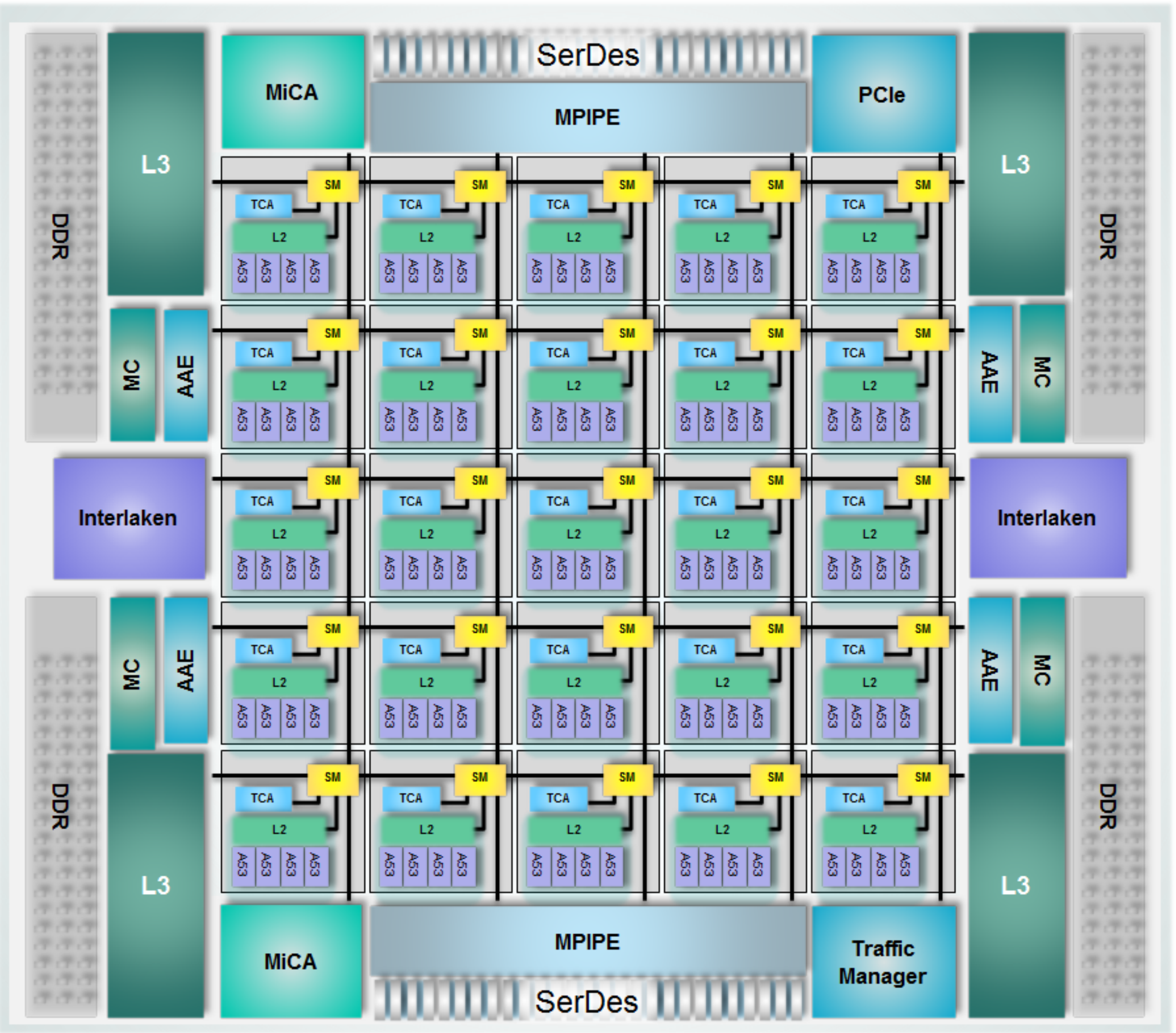
Clickers/Peer Instruction

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



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



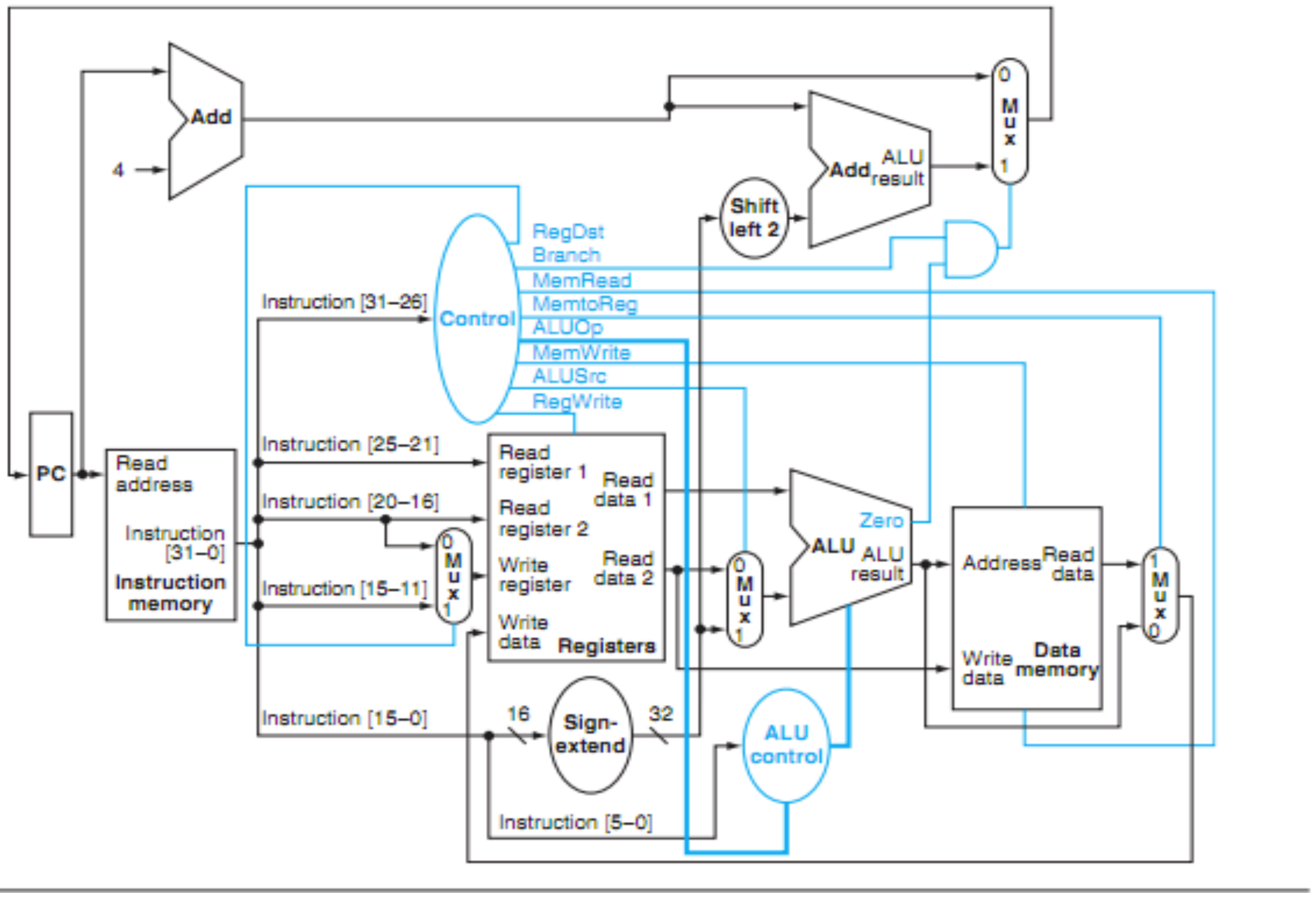
EZChip (bought Tilera)

100 64-bit ARM Cortex A53

- Dual-issue, in-order

Break

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)

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

	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
```

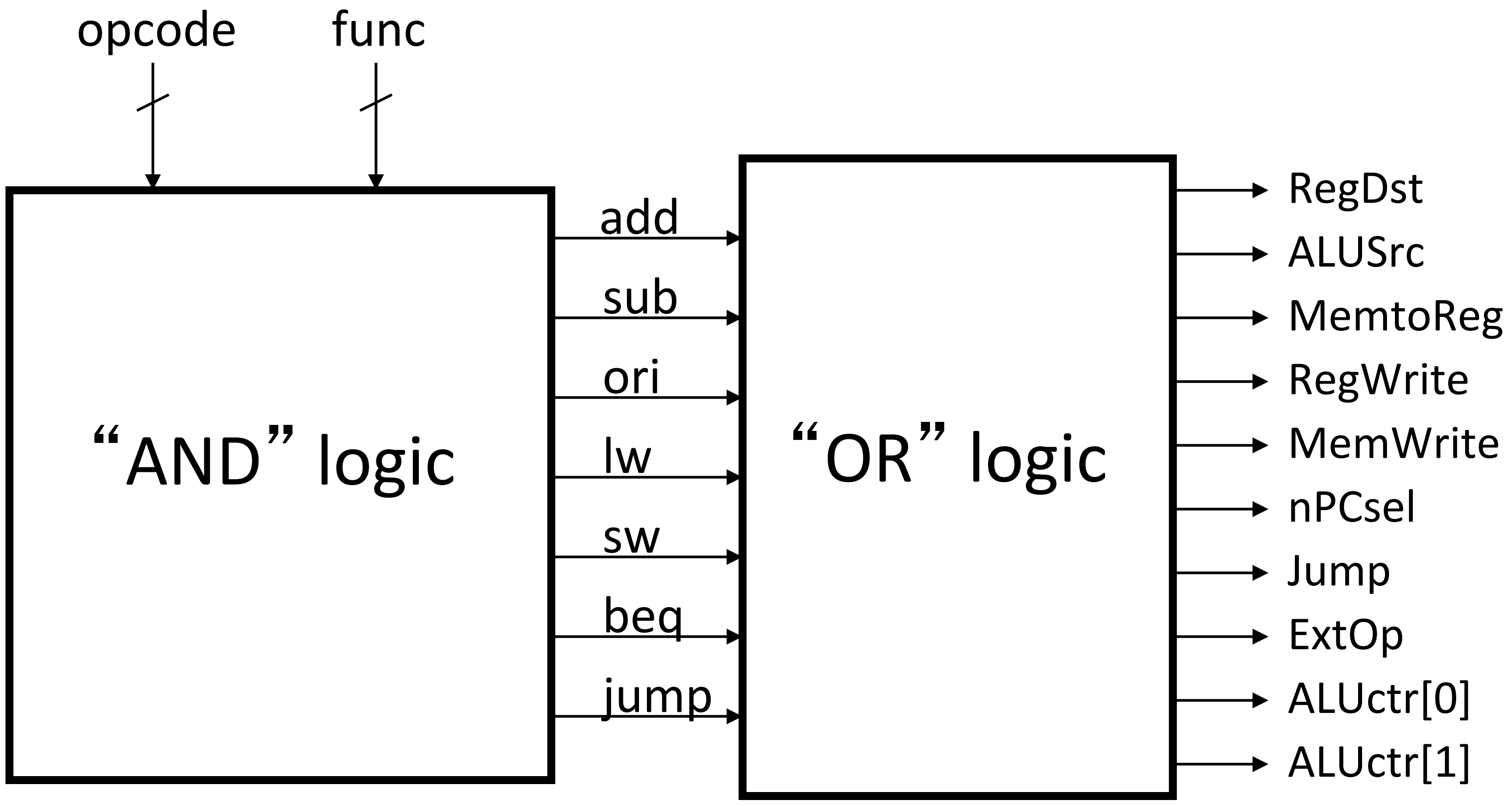
Where:

rtype = $\sim op_5 \bullet \sim op_4 \bullet \sim op_3 \bullet \sim op_2 \bullet \sim op_1 \bullet \sim op_0$,
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



Clicker Question

Which of the following is TRUE?

A. The CPU’s control needs only opcode/funct to determine the next PC value to select

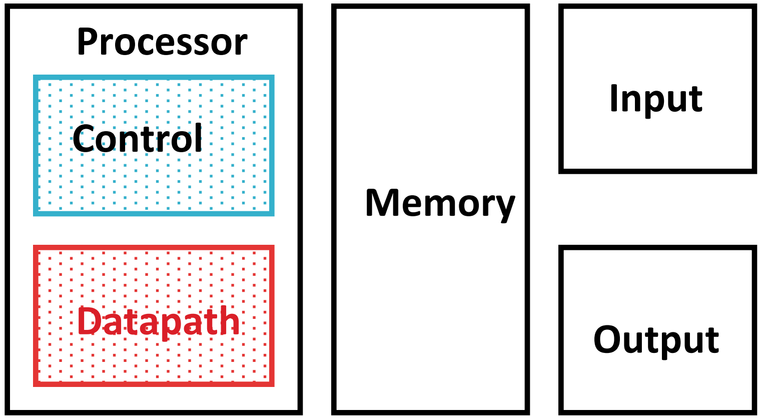
B. The clock can have a shorter period for instructions that don’t use memory

C. The CPU requires a separate instruction memory and data memory

D. The ALU is used to set PC to PC+4 when necessary

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


Processor
Control
Datapath
Memory
Input
Output