

CSE 140

Computer Architecture

Lecture 2 – Digital Logic in Datapath



Announcement

- ▶ Lab #1 starts next week (9/9)
- ▶ HW #1 out this Friday
 - Due Monday (9/16)
- ▶ Reading assignment
 - Chapter 2.12, 4.1 – 4.3
 - Do all **Participation Activities** in each section
 - Access through CatCourses
 - Due Thursday (9/5) at 11:59pm
 - Review CSE 31 materials (available at CatCourses)
 - Assembly language and machine code: Ch. 2.1-2.8, 2.9-2.10, 2.13
 - Quick review:
 - <https://classroom.udacity.com/courses/ud219>

Truth Tables

- Describe the behavior of the following logic gate

INPUTS			OUTPUTS		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

D=1 if at least 1 input is true
E=1 if exactly 2 inputs are true
F=1 only if all 3 inputs are true

- Two types of logic blocks
 - Combinational logic: depends only on inputs, with no memory
 - Sequential logic: depends on the inputs and memory (state)

Boolean Algebra

- ▶ Logic functions with logical operators
 - OR operator: $+$, AND operator: $*$, NOT operator: \bar{A}
- ▶ Properties:
 - Identity law: $A+0=A$, and $A*1=A$
 - Zero and One laws: $A+1=1$, and $A*0=0$
 - Inverse laws: $A+\bar{A}=1$, and $A*\bar{A}=0$
 - Commutative laws: $A+B=B+A$, and $A*B=B*A$
 - Associate laws: $A+(B+C)=(A+B)+C$, and $A*(B*C)=(A*B)*C$
 - Distributive laws: $A*(B+C)=(A*B)+(A*C)$, and $A+(B*C)=(A+B)*(A+C)$
 - DeMorgan's laws: $\overline{A \cup B} = \bar{A} \cap \bar{B}$, $\overline{A \cap B} = \bar{A} \cup \bar{B}$

Logic Equations

INPUTS			OUTPUTS		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

D=1 if at least 1 input is true
E=1 if exactly 2 inputs are true
F=1 only if all 3 inputs are true

$$D = A + B + C$$

$$F = A * B * C$$

$$E = ((A * B) + (A * C) + (B * C)) * \overline{(A * B * C)}$$

$$E = (A * B * \overline{C}) + (A * C * \overline{B}) + (B * C * \overline{A})$$

Canonical forms

Gates

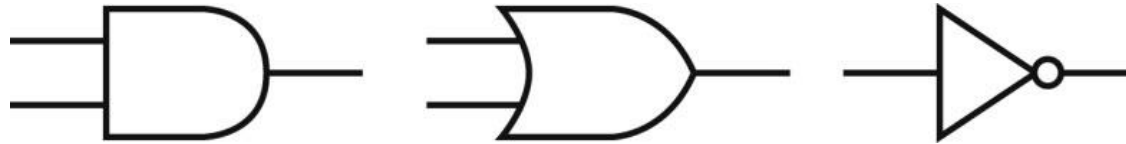


FIGURE B.2.1 Standard drawing for an AND gate, OR gate, and an inverter, shown from left to right. The signals to the left of each symbol are the inputs, while the output appears on the right. The AND and OR gates both have two inputs. Inverters have a single input.

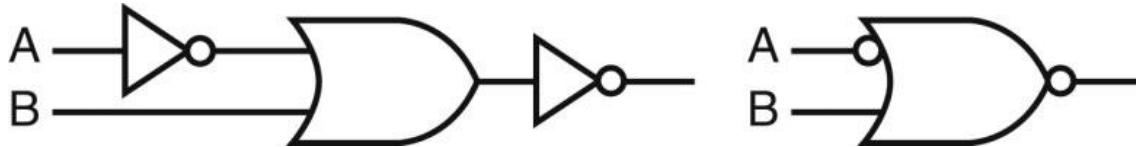


FIGURE B.2.2 Logic gate implementation of using explicit inverters on the left and bubbled inputs and outputs on the right. This logic function can be simplified to or in Verilog, $A \& \sim B$.

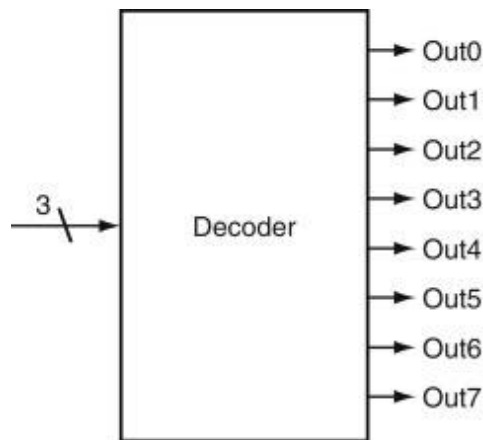
Assert: a signal that is true

Deassert: a signal that is false

- ▶ Any logic function can be constructed using AND, OR and NOT gates
- ▶ Any logic function can be constructed using NOR and NAD gates

Combinational Logic

► 3-to-8 decoder



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

FIGURE B.3.1 A 3-bit decoder has 3 inputs, called 12, 11, and 10, and $2^3 = 8$ outputs, called Out0 to Out7. Only the output corresponding to the binary value of the input is true, as shown in the truth table. The label 3 on the input to the decoder says that the input signal is 3 bits wide.

Is it the same as Mux?

Multiplexors

▶ 2-to-1 MUX

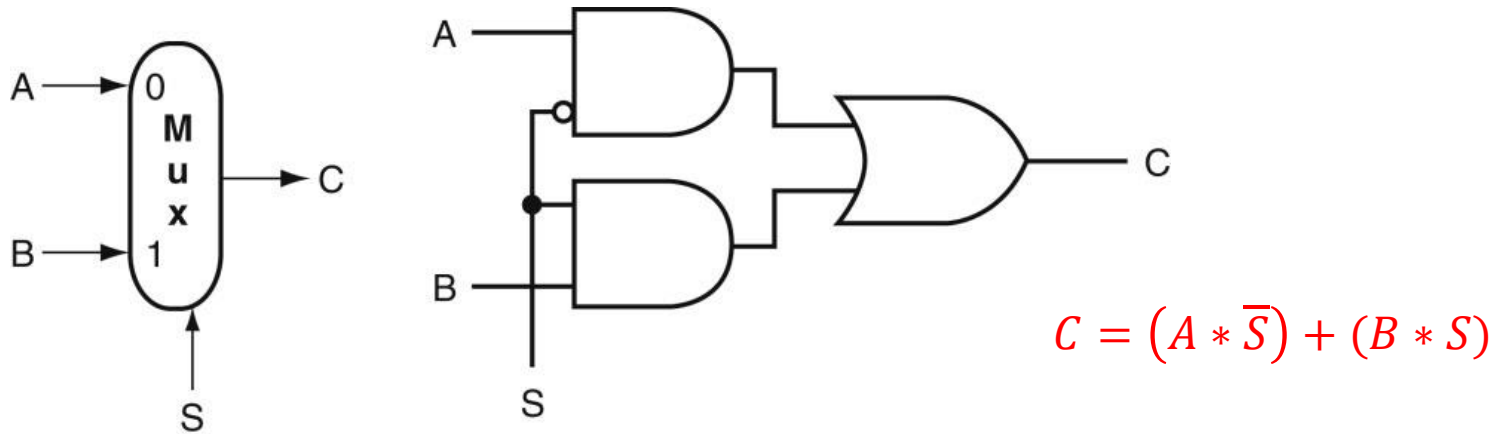
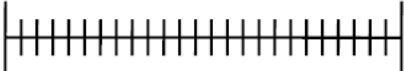


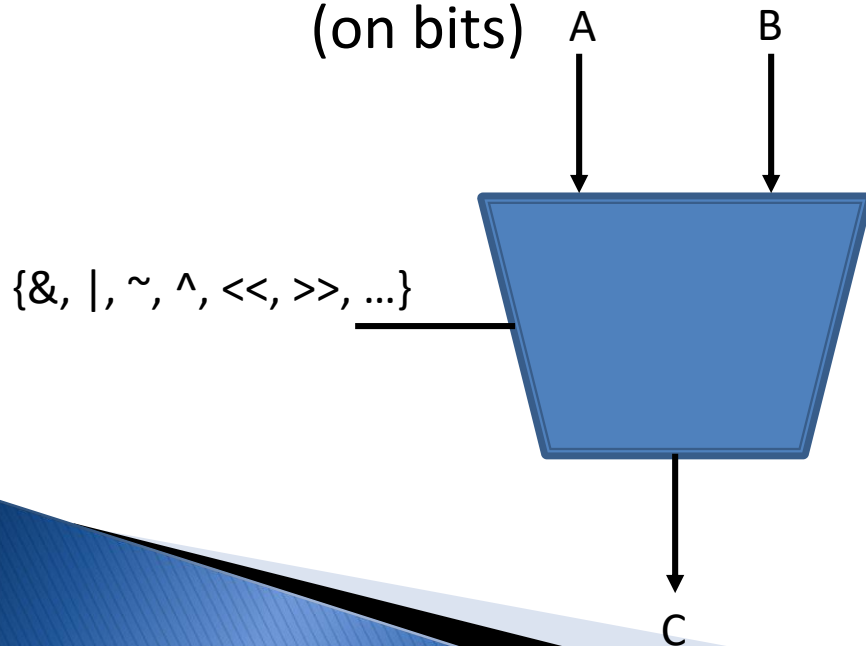
FIGURE B.3.2 A two-input multiplexor on the left and its implementation with gates on the right. The multiplexor has two data inputs (A and B), which are labeled 0 and 1, and one selector input (S), as well as an output C . Implementing multiplexors in Verilog requires a little more work, especially when they are wider than two inputs. We show how to do this beginning on page B-23.

- For n data inputs, need $\log_2(n)$ selector

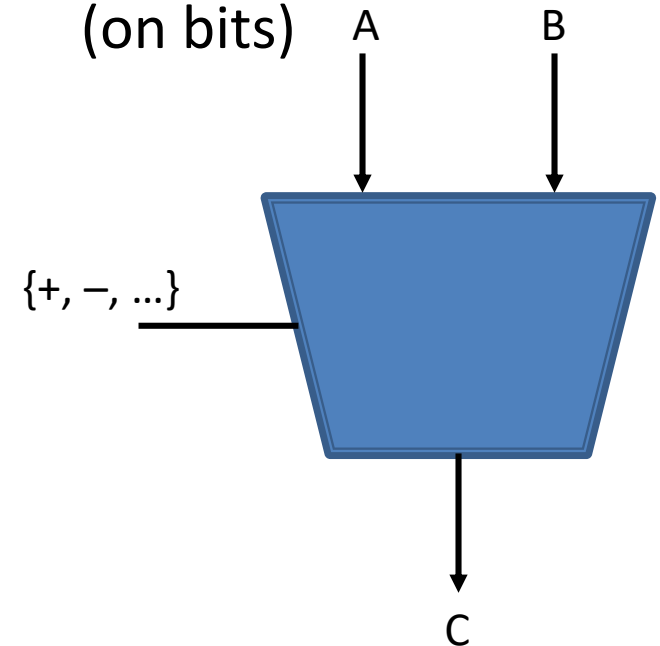
Operations (on the representation)

- ▶ Computers provide direct hardware support for manipulating certain basic objects
- ▶ Word of bits: 

Logical Operations
(on bits)



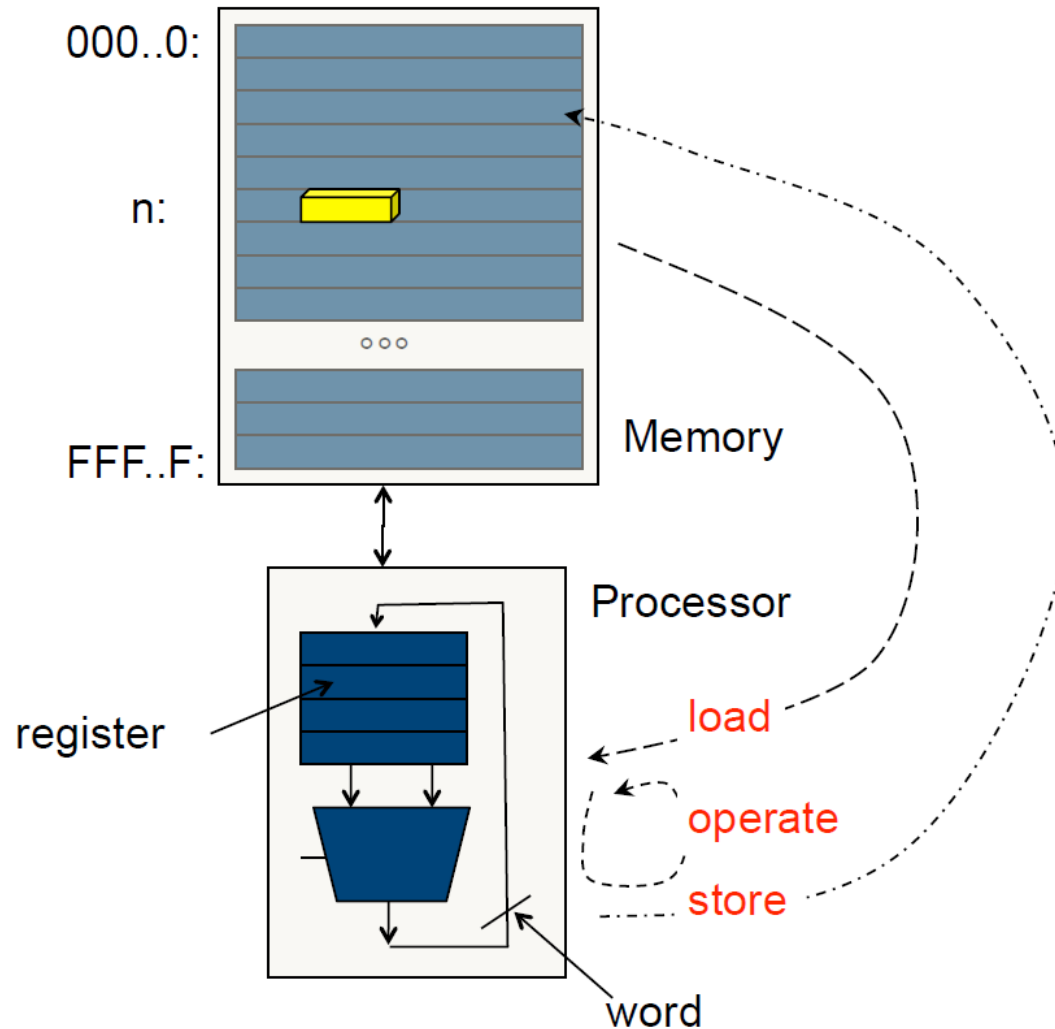
Arithmetic Operations
(on bits)



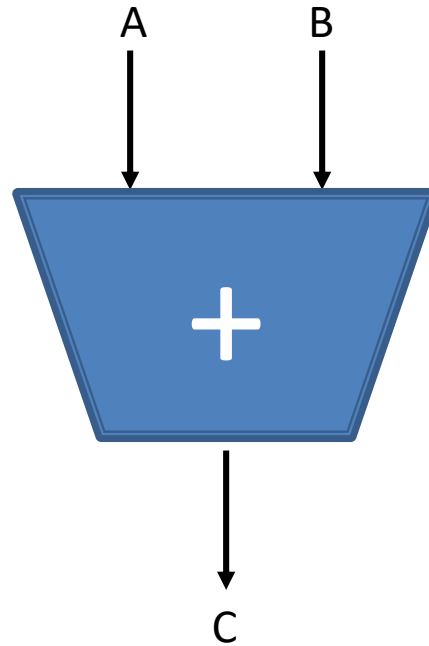
Manipulating bits

- ▶ We can work with what the object represents
 - $x = x + y;$
 - $Z = x * 2;$
 - $Q = x / 2;$
 - `putchar(c);`
 - `if (A && !B) { ... }`
- ▶ Or with its underlying representation (bitwise)
 - $Z = x \ll 1;$
 - $Q = x \gg 1;$
 - $D = x \& 1;$
 - $E = x \& 0xFFFFFFFF;$
 - $G = x \& \sim 1$
 - $M = (y \& 0x74) \gg 3$

Where do Objects live and work?



ALU - Add

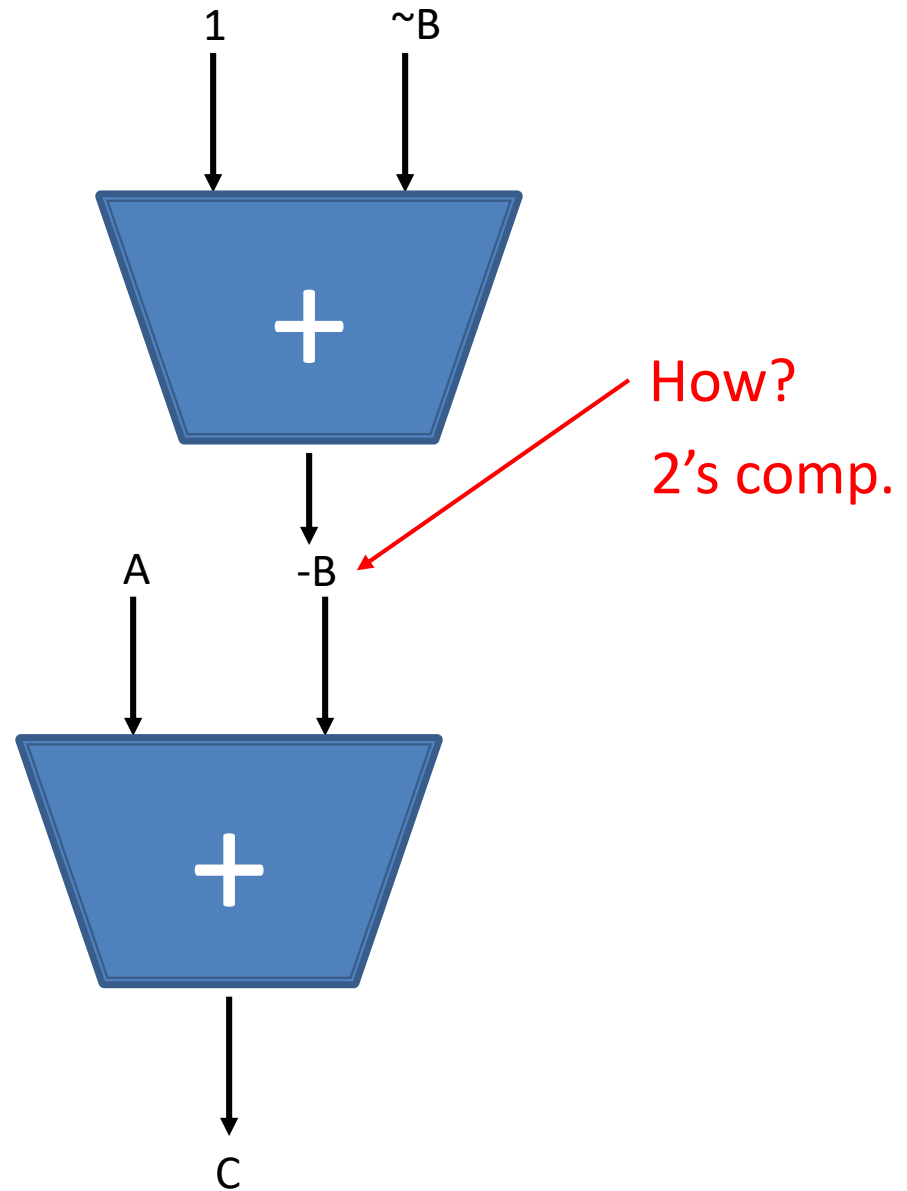


Subtract?

$$C = A - B$$

Can we do it without
introducing a SUB operator?

How do you get -B?

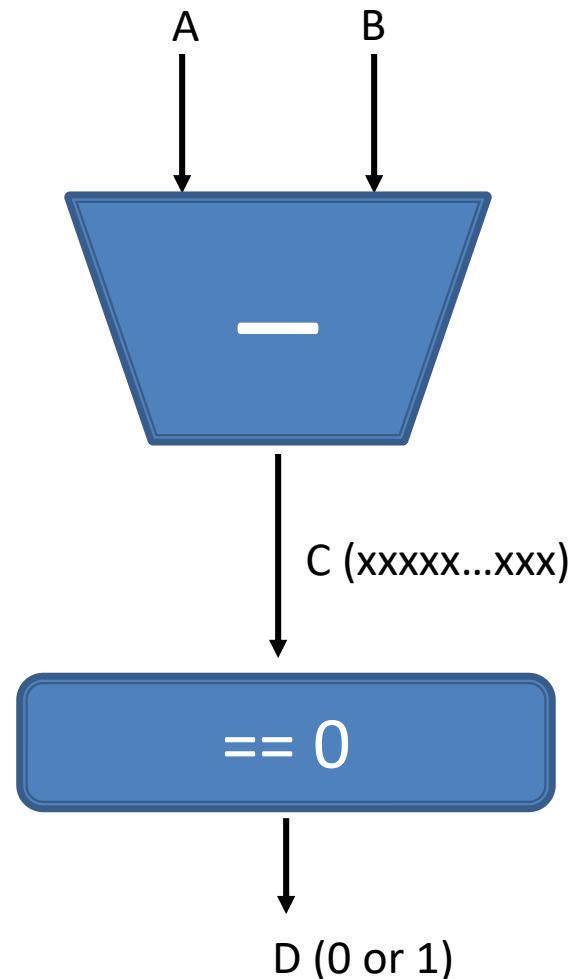


Equal to 0? ($B == 0$)

- ▶ Bit 0 is 0
 - ▶ Bit 1 is 0
 - ▶ Bit 2 is 0
 - ▶ ...
 - ▶ Bit $n-1$ is 0
 - ▶ What is the logic?
-
- ▶ $\sim(B0 \mid B1 \mid B2 \mid \dots \mid B_{n-1})$

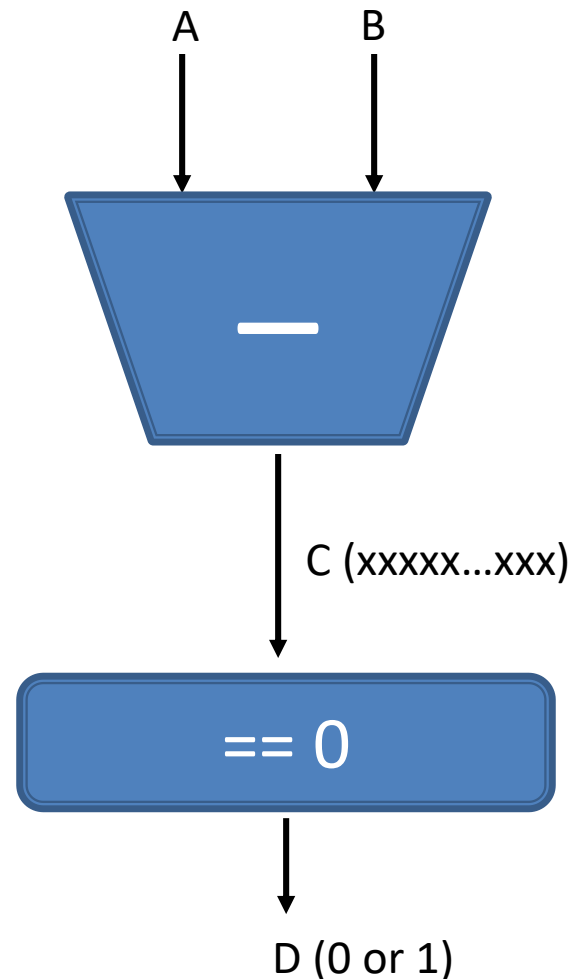
Quiz: How to Compare A to B?

Depends on the values of C_{n-1}
(left-most bit of C) and D!

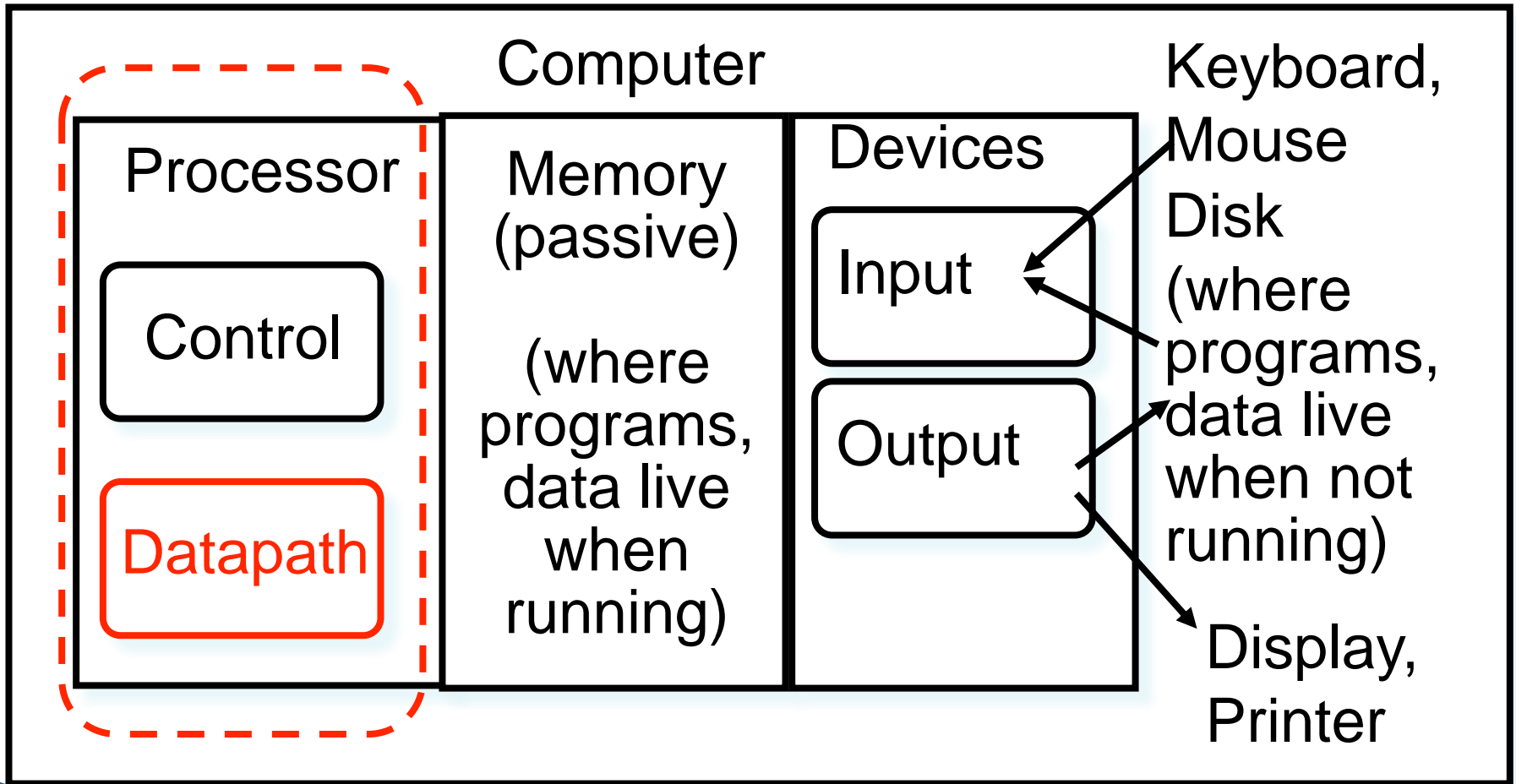


Answer: How to Compare A to B?

- ▶ Less than :
 - C_{n-1}
- ▶ Less than or equal :
 - $D \mid C_{n-1}$
- ▶ Equal :
 - D
- ▶ Not Equal:
 - $\sim D$
- ▶ Greater than :
 - $\sim D \ \& \ \sim C_{n-1}$
- ▶ Greater than or equal :
 - $\sim C_{n-1}$



Five Components of a Computer



The CPU

- ▶ **Processor (CPU)**: the active part of the computer, which does all the work (data manipulation and decision-making)
 - **Datapath**: portion of the processor which contains hardware necessary to perform operations required by the processor (how data is transformed)
 - **Control**: portion of the processor (also in hardware) which tells the datapath what needs to be done (where data go)

Stages of the Datapath

- ▶ Problem: a single, atomic block which “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient
- ▶ Solution: break up the process of “executing an instruction” into **stages**, and then connect the stages to create the whole **datapath**
 - smaller stages are easier to design
 - easy to optimize (change) one stage without touching the others

Stages of the Datapath (1/5)

- ▶ There is a wide variety of MIPS instructions: so what general steps do they have in common?
- ▶ Stage 1: **Instruction Fetch**
 - No matter what the instruction type is, the 32-bit instruction word must first be fetched from memory (the cache-memory hierarchy)
 - Also, this is where we *Increment PC (program counter)* (that is, $PC = PC + 4$, to point to the next instruction: byte addressing so + 4)

Stages of the Datapath (2/5)

► Stage 2: Instruction Decode

- Upon fetching the instruction, we next gather data from the fields (decode all necessary instruction data)
- First, read the `opcode` to determine instruction type and field lengths
- Second, read in data from all necessary registers
 - for `add`, read two registers (R-format)
 - for `addi`, read one register (I-format)
 - for `jal`, no reads necessary (J-format)

Stages of the Datapath (3/5)

► Stage 3: ALU (Arithmetic-Logic Unit)

- The real work of most instructions is done here: arithmetic (+, -, *, /), shifting, logic (&, |), comparisons (`slt`)
- What about loads and stores?
 - `lw $t0, 40($t1)`
 - the address we are accessing in memory = the value in `$t1` PLUS the value 40
 - so we do this addition in this stage

Stages of the Datapath (4/5)

► Stage 4: Memory Access

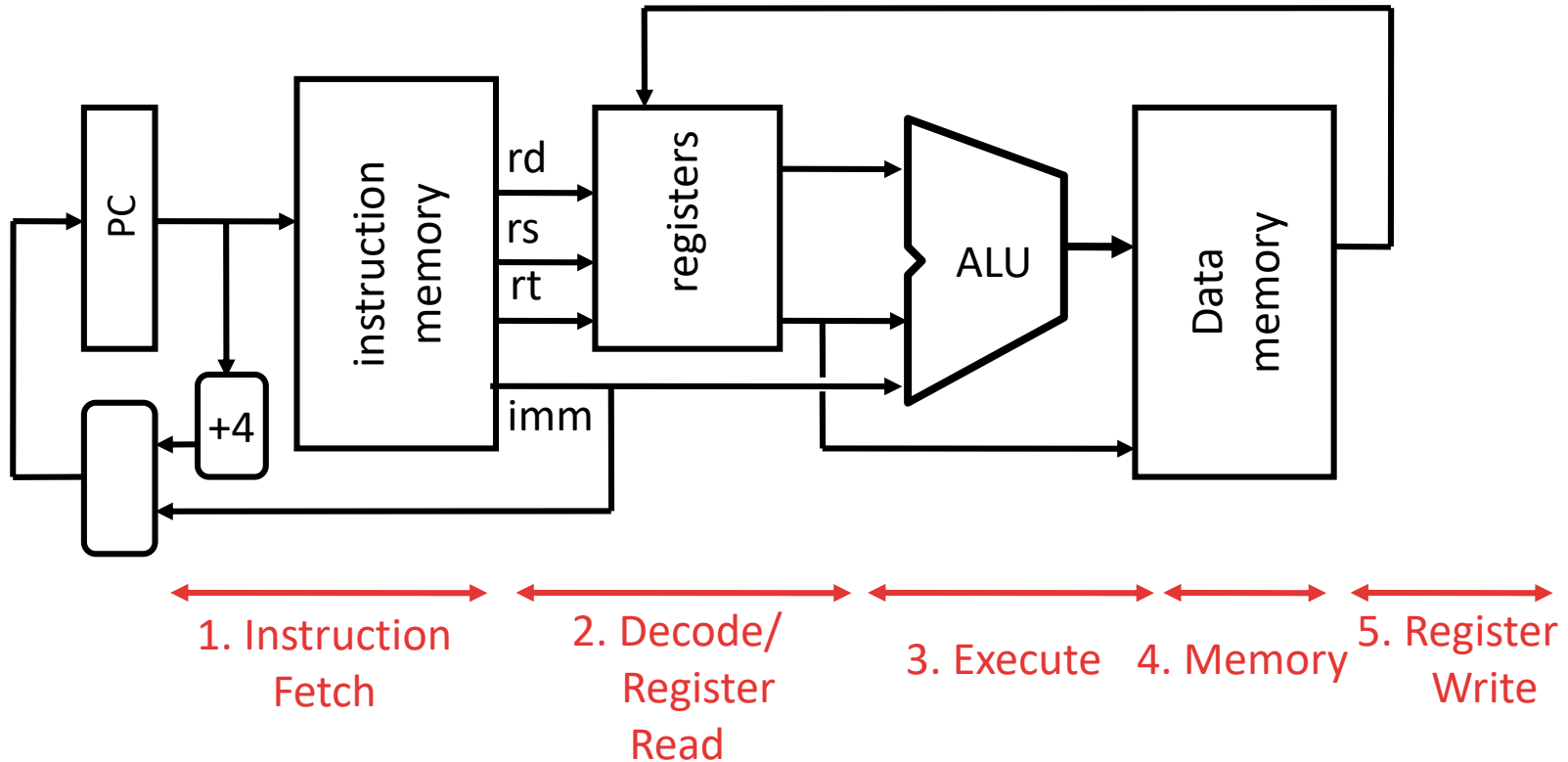
- Actually only the load and store instructions do anything during this stage; the others remain idle during this stage or skip it all together
- Since these instructions have a unique step, we need this extra stage to account for them
- as a result of the cache system, this stage is expected to be fast

Stages of the Datapath (5/5)

► Stage 5: Register Write

- Most instructions write the result of some computation into a register
- Examples: arithmetic, logical, shifts, loads, slt
- What about stores, branches, jumps?
 - don't write anything into a register at the end
 - these remain idle during this fifth stage or skip it all together

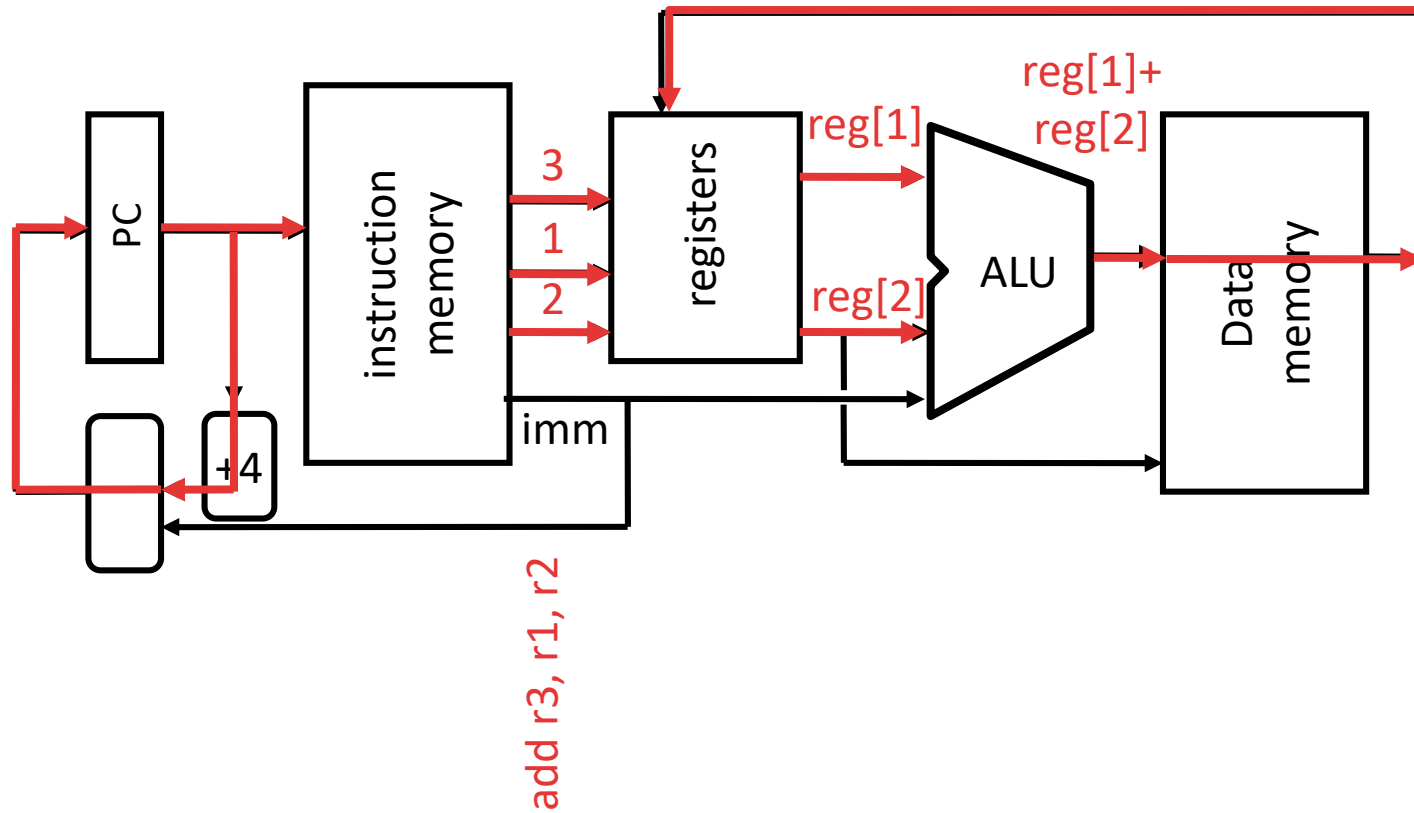
Generic Steps of Datapath



Datapath Walkthroughs (1/3)

- ▶ `add $r3,$r1,$r2 # r3 = r1+r2`
 - Stage 1: fetch this instruction, inc. PC
 - Stage 2: decode to find it's an `add`, then read registers `$r1` and `$r2`
 - Stage 3: add the two values retrieved in Stage 2
 - Stage 4: idle (nothing to write to memory)
 - Stage 5: write result of Stage 3 into register `$r3`

Example: add Instruction

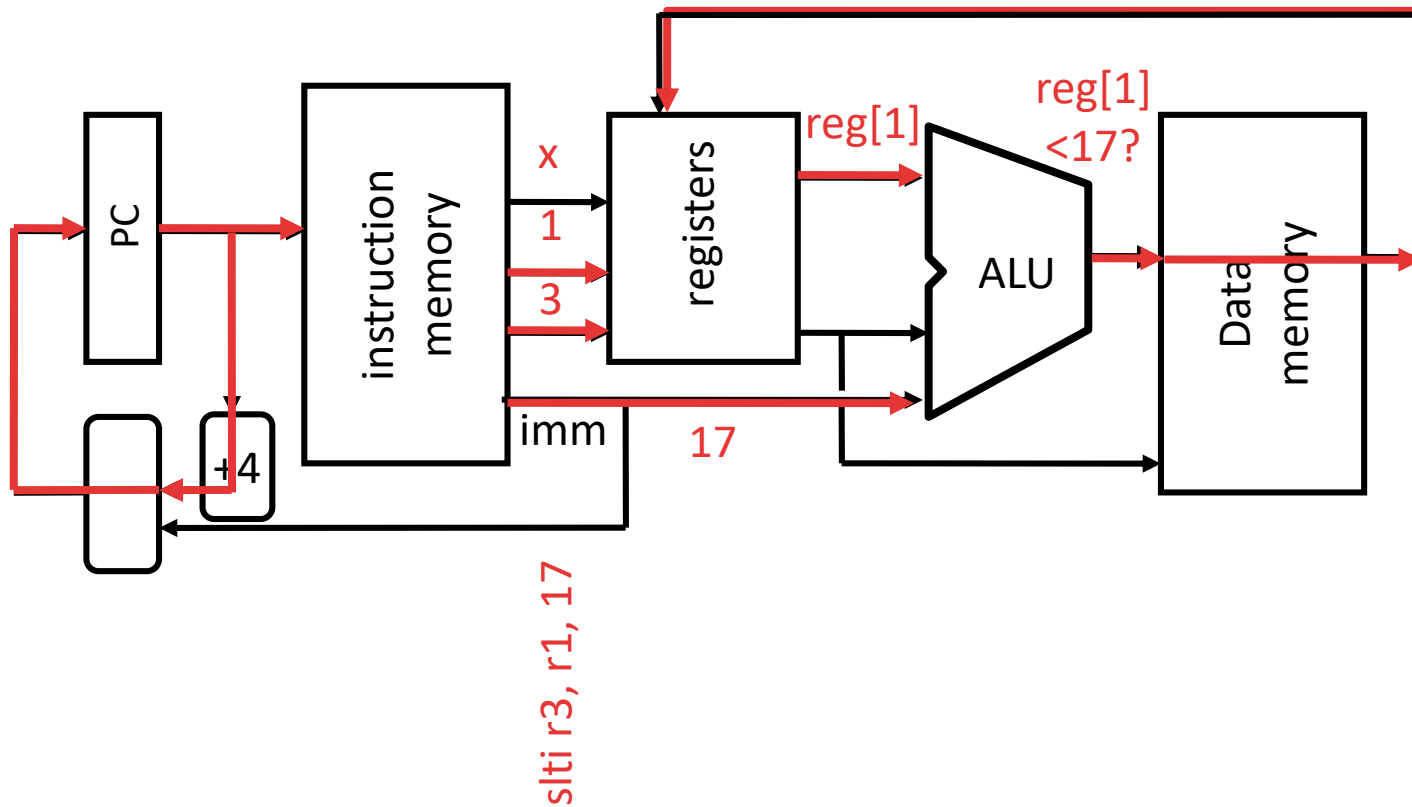


Datapath Walkthroughs (2/3)

▶ `slti $r3, $r1, 17`

- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it's an `slti`, then read register `$r1`
- Stage 3: compare value retrieved in Stage 2 with the integer 17
- Stage 4: idle
- Stage 5: write the result of Stage 3 in register `$r3`

Example: `slti` Instruction

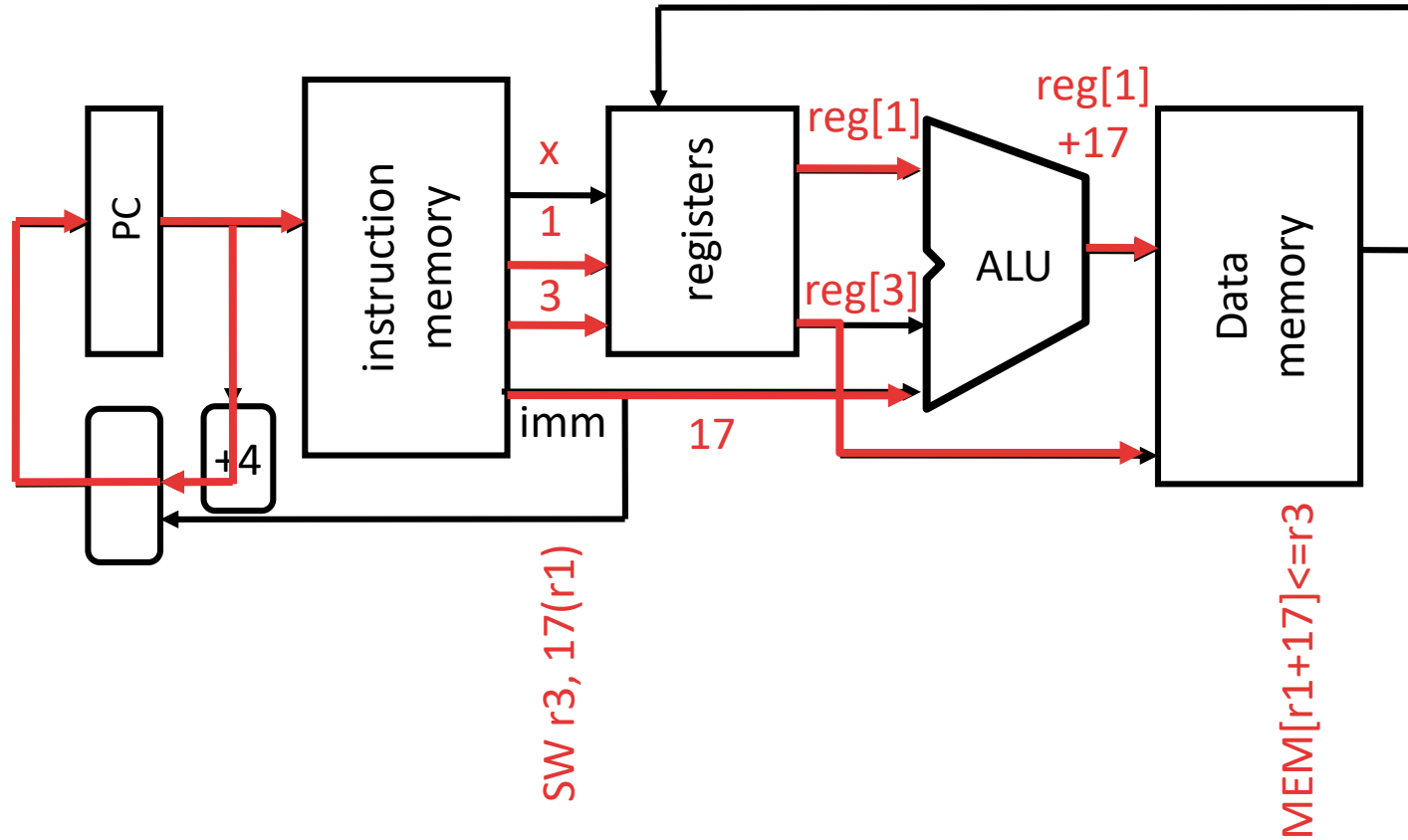


Datapath Walkthroughs (3/3)

► `sw $r3, 17($r1)`

- Stage 1: fetch this instruction, inc. PC
- Stage 2: decode to find it's a `sw`, then read registers `$r1` and `$r3`
- Stage 3: add 17 to value in register `$r1` (retrieved in Stage 2)
- Stage 4: write value from register `$r3` (retrieved in Stage 2) into memory address computed in Stage 3
- Stage 5: idle (nothing to write into a register)

Example: *sw* Instruction



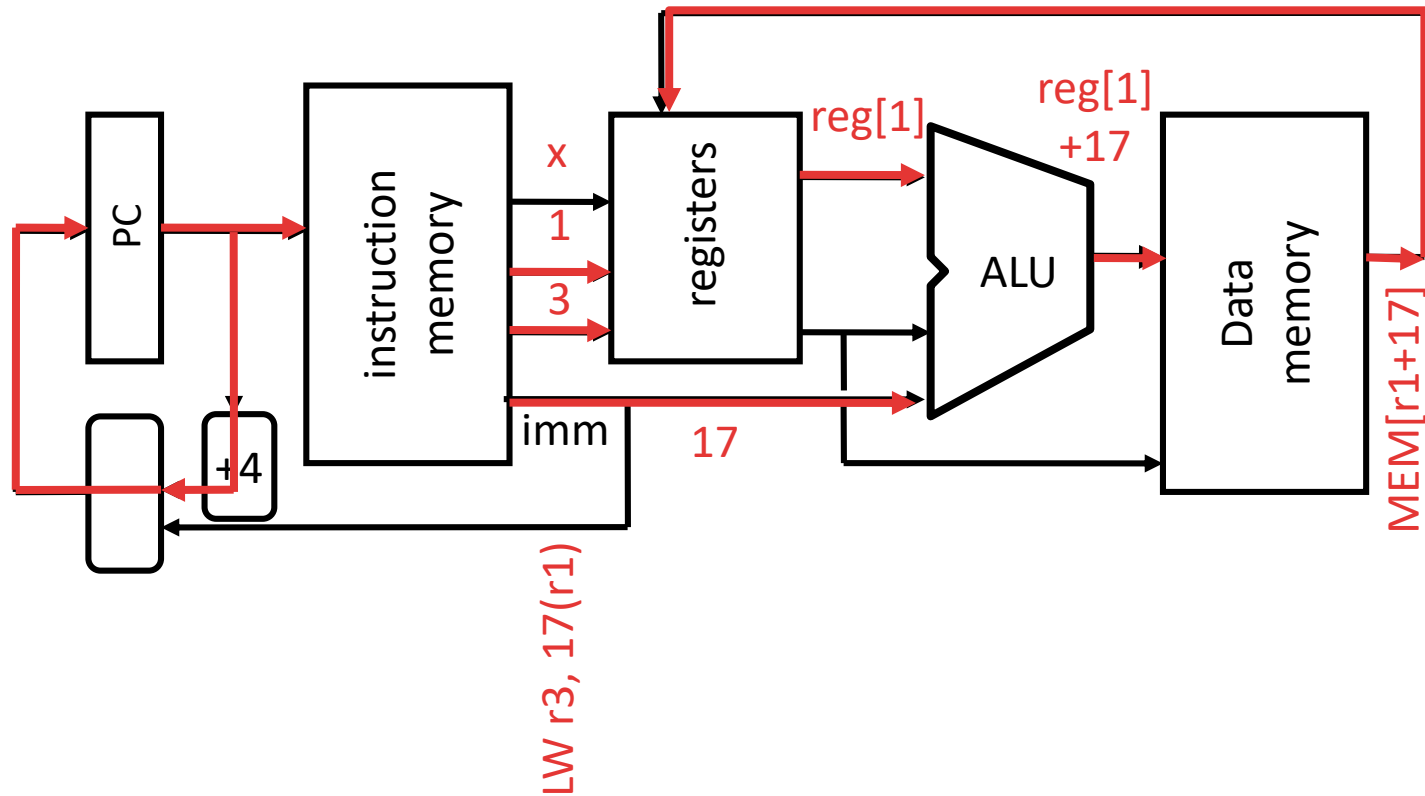
Why Five Stages? (1/2)

- ▶ Could we have a different number of stages?
 - Yes, and other architectures do (Intel Core-i has 14)
- ▶ So why does MIPS have five if instructions tend to idle for at least one stage?
 - The five stages are the union of all the operations needed by all the instructions.
 - There is one instruction that uses all five stages:
 - the **load**

Why Five Stages? (2/2)

- ▶ `lw $r3, 17($r1)`
 - Stage 1: fetch this instruction, inc. PC
 - Stage 2: decode to find it's a `lw`, then read register `$r1`
 - Stage 3: add 17 to value in register `$r1` (retrieved in Stage 2)
 - Stage 4: read value from memory address compute in Stage 3
 - Stage 5: write value found in Stage 4 into register `$r3`

Example: 1w Instruction



Datapath Summary

- Datapath based on data transfers required to perform instructions
- Controller causes the right transfers to happen

