Processor Review

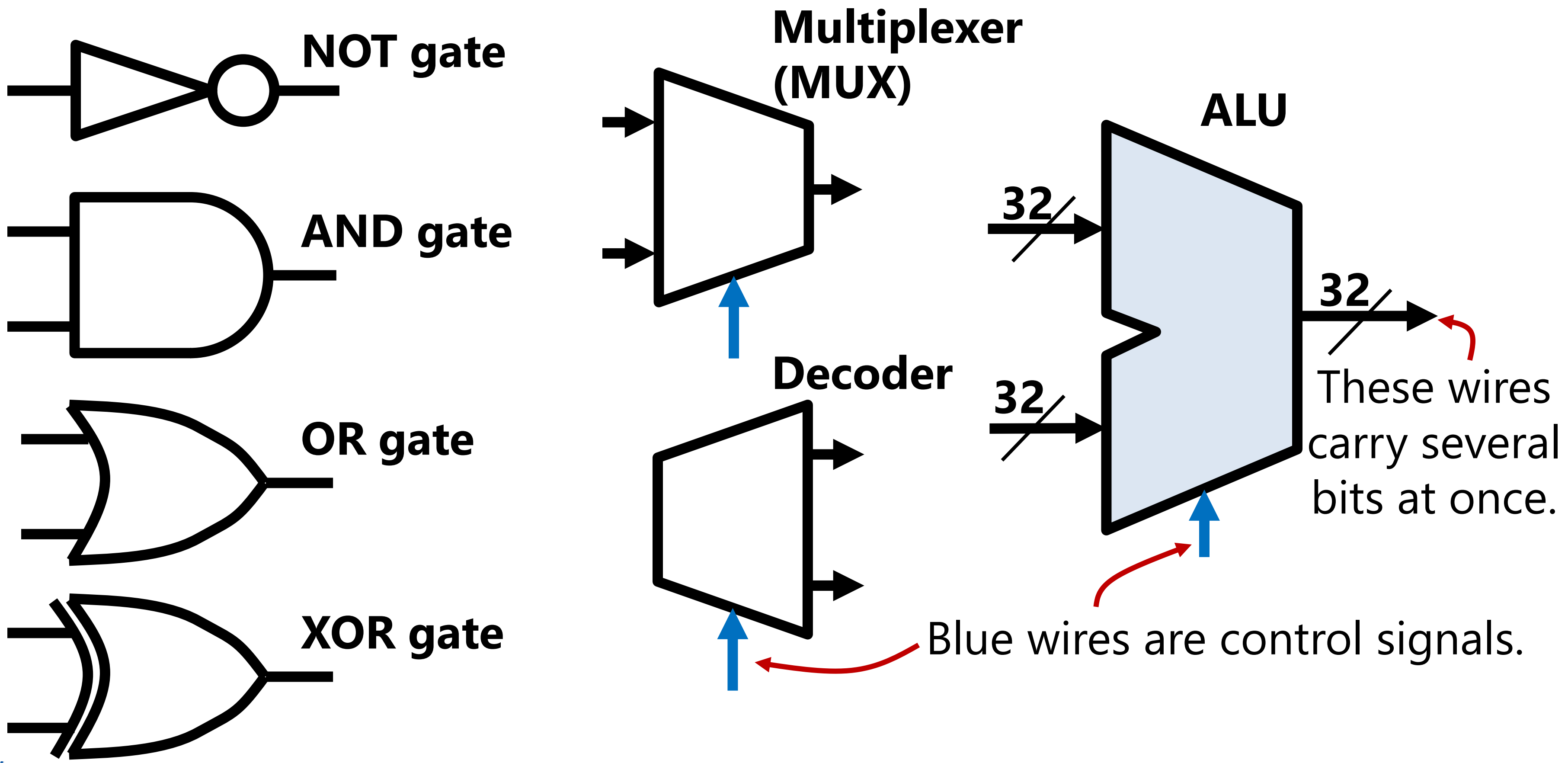
CS 1541
Wonsun Ahn

Clocking Review

Stuff you learned in CS 447

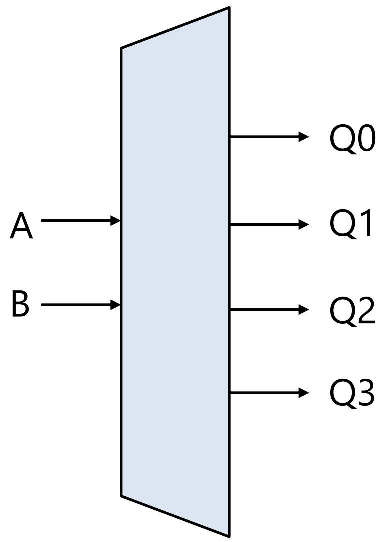
Logic components

- Do you remember what all these do?


NOT gate
AND gate
OR gate
XOR gate
Multiplexer (MUX)
Decoder
ALU
32
32
32
These wires carry several bits at once.
Blue wires are control signals.

Uses of a Decoder

- Translates a set of input signals to a bunch of output signals.
  - E.g. a binary decoder:



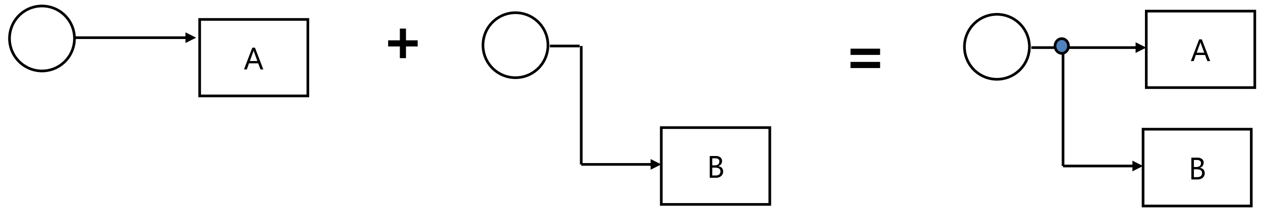
Truth Table for Decoder

A	B	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

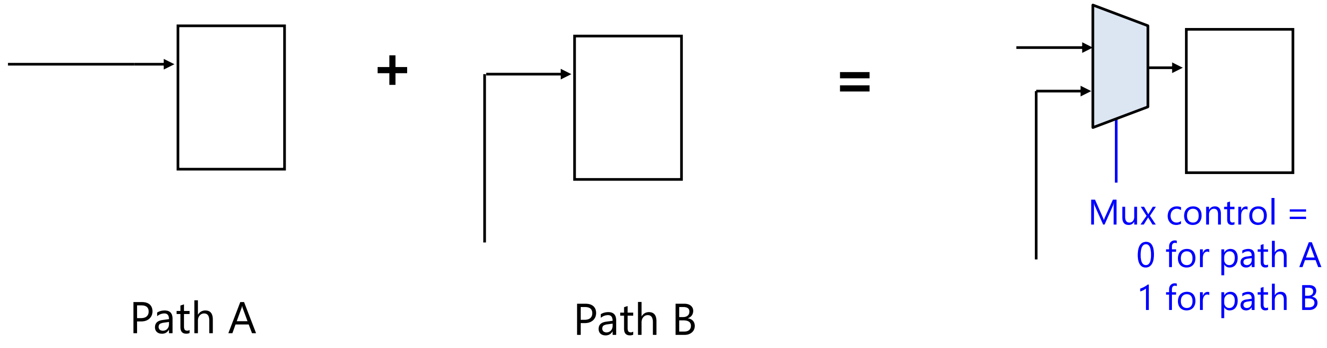
  - You can come up with any truth table and make a decoder for it!

Uses of a Multiplexer

- No problem in fanning out one signal to two points

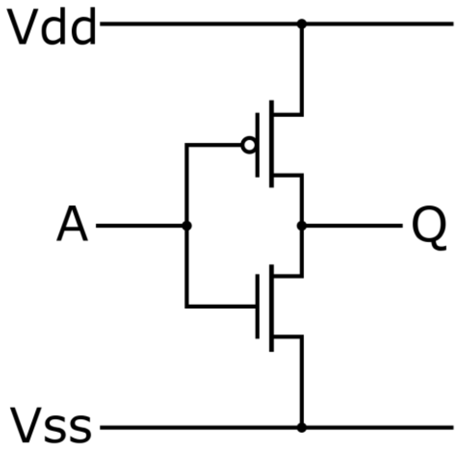


- Cannot connect two signals to one point
  - Must use a multiplexer to *select* between the two

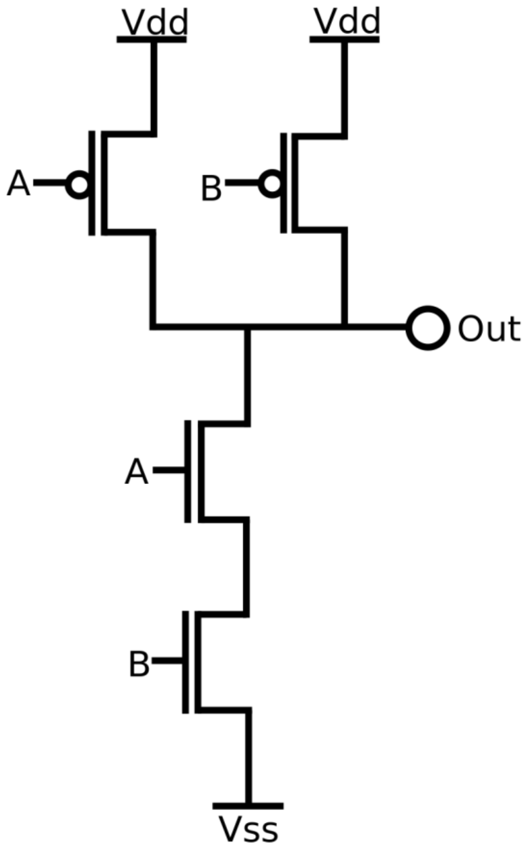


Gates are made of transistors (of course)

- NOT gate

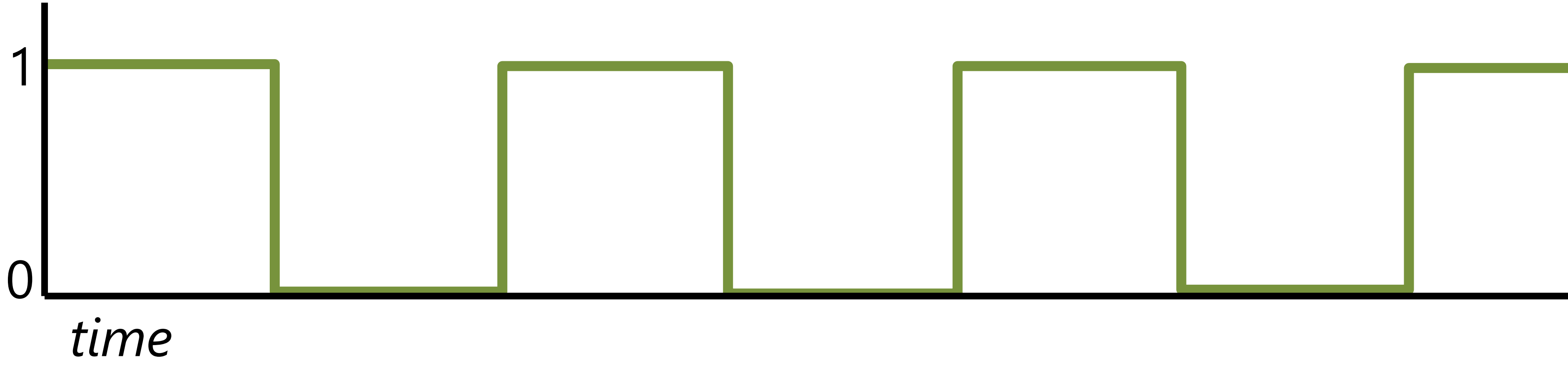


- NAND gate



The clock signal

- The clock is a signal that alternates regularly between 0 and 1:

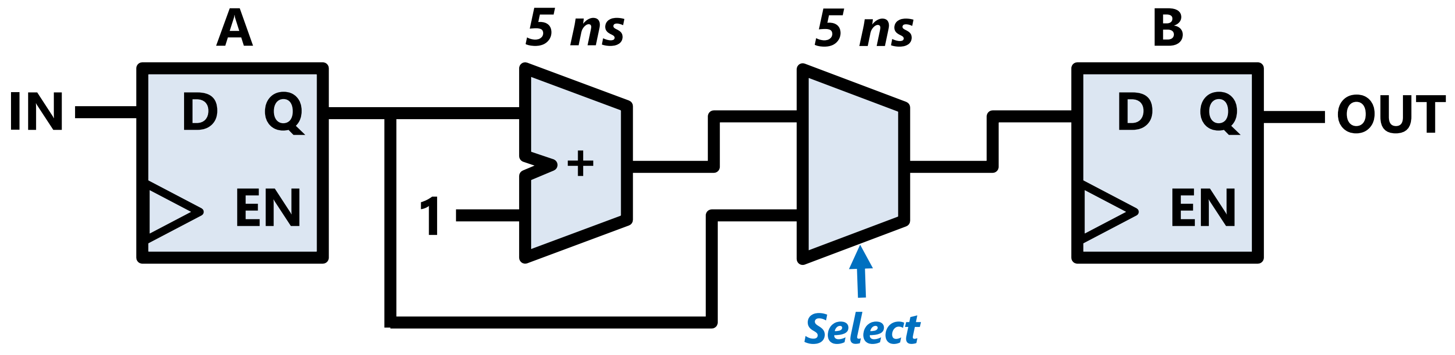


- Bits are latched on to registers and flip-flops on rising edges

- In between rising edges, bits propagate through the logic circuit
    - Composed of ALUs, muxes, decoders, etc.
    - ***Propagation delay***: amount of time it takes from input to output

Critical Path

- ***Critical path***: path in a circuit that has longest propagation delay
  - Determines the overall clock speed.



  - The ALU and the multiplexer both have a 5 ns delay
- How fast can we clock this circuit?
  - Is it 1 / 5 ns (5×10^{-9}s) = 200 MHz?
  - Or is it 1 / 10 ns (10×10^{-9}s) = 100 MHz? ✓

MIPS Review

Stuff you learned in CS 447

The MIPS ISA - Registers

- MIPS has 32 32-bit registers, with the following usage conventions
  - But really, all are general purpose registers (nothing special about them)

Name	Register number	Usage
`$zero`	0	the constant value 0 (can't be written)
`$at`	1	assembler temporary
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	function arguments
`$t0-$t7`	8-15	unsaved temporaries
`$s0-$s7`	16-23	saved temporaries (like program variables)
`$t8-$t9`	24-25	more unsaved temporaries
`$k0-$k1`	26-27	reserved for OS kernel
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

The MIPS ISA - Memory

- MIPS is a ***RISC (reduced instruction set computer)*** architecture
- It is also a ***load-store*** architecture
  - **All** memory accesses performed by load and store instructions
- Memory is a giant array of 2^{32} bytes

Addr	Data
0	0x3F
1	0x00
2	0x2A
3	0x08
4	0x47
5	0xF4
6	0x26
7	0xB9
...	...

Addr	Data
0	0x3F00
2	0x2A08
4	0x47F4
6	0x26B9
...	...

Addr	Data
0	0x3F002A08
4	0x47F426B9
...	...

- The same memory viewed as bytes, 16-bit halfwords, and 32-bit words (using big-endian)
- All addresses are ***aligned*** (multiples of data size)

The MIPS ISA - Memory

- Loads move data *from* memory *into* the registers.

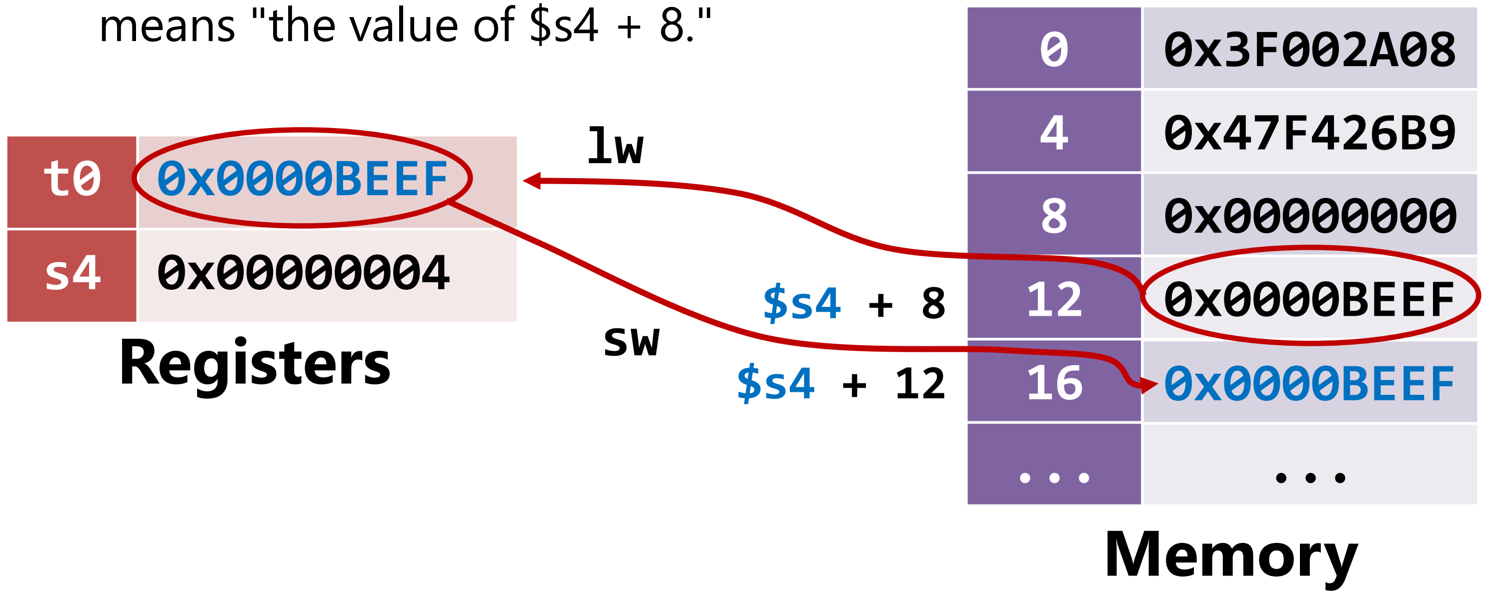
$\texttt{lw \$t0, } \underbrace{\texttt{8(\$s4)}}$

This is the address, and it means "the value of $s4 + 8."

- Stores move data *from* the registers *into* memory.

`sw $t0, 12($s4)`

$t0 is the SOURCE!



The MIPS ISA – Flow control

- Jump and branch instructions change the flow of execution.

```
_top:
    # ....
    # lots o' code
    # ....
    j _top
```

```
    li   $s0, 10
_loop:
    # ....
    addi $s0, $s0, -1
    bne  $s0, $zero, _loop
    jr   $ra
```

j : jumps *unconditionally*

- jumps to `_top`

bne : jumps *conditionally*

If `$s0` != `$zero`, jumps to `_loop`

If `$s0` == `$zero`, continues to `jr   $ra`

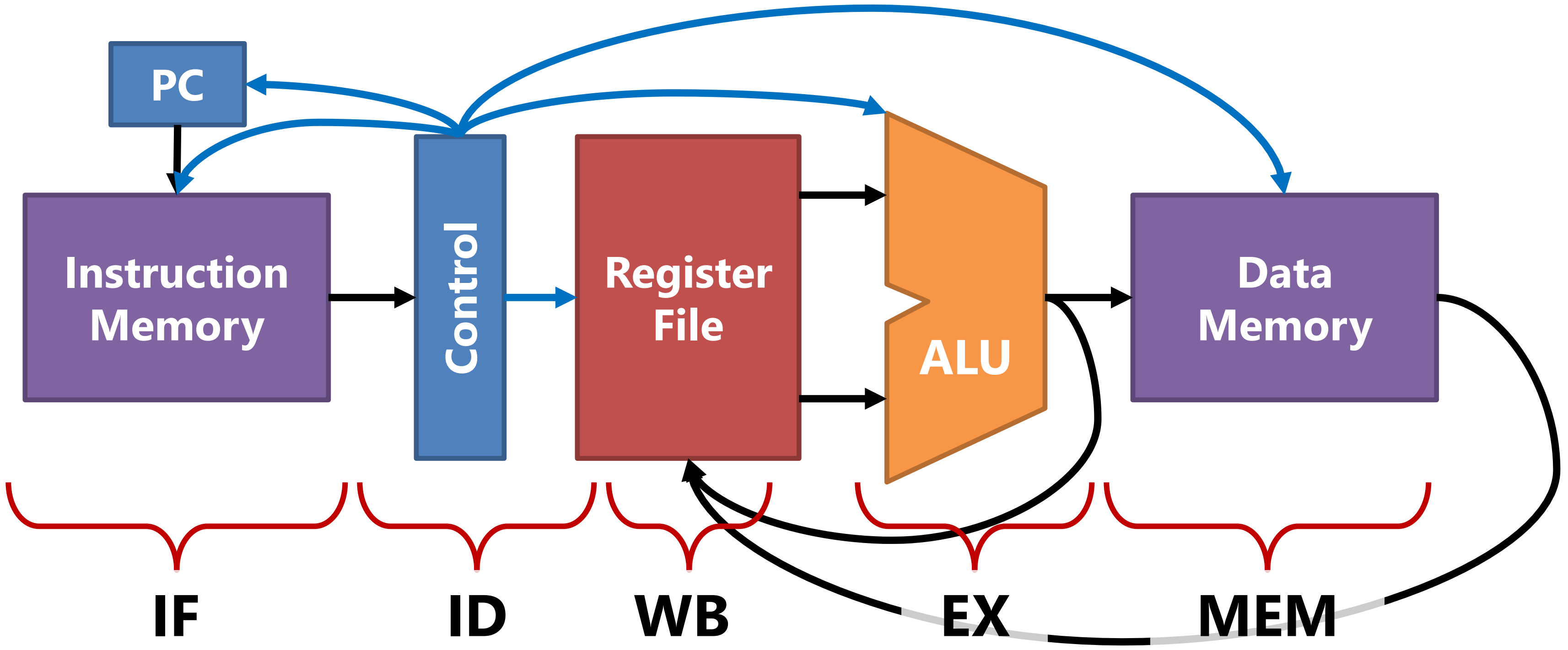
Phases of instruction execution

- In most architectures, there are five phases:
  1. **IF** (Instruction Fetch) – get next instruction from memory
  2. **ID** (Instruction Decode) – figure out what instruction it is
  3. **EX** (Execute – ALU) – do any arithmetic
  4. **MEM** (Memory) – read or write data from/to memory
  5. **WB** (Register Writeback) – write any results to the registers

- Sometimes these phases are chopped into smaller stages

University of Pittsburgh

A simple single-cycle implementation



- An instruction goes through IF/ID/EX/MEM/WB in one cycle

"Minimal MIPS"

It’s a “subset” of MIPS

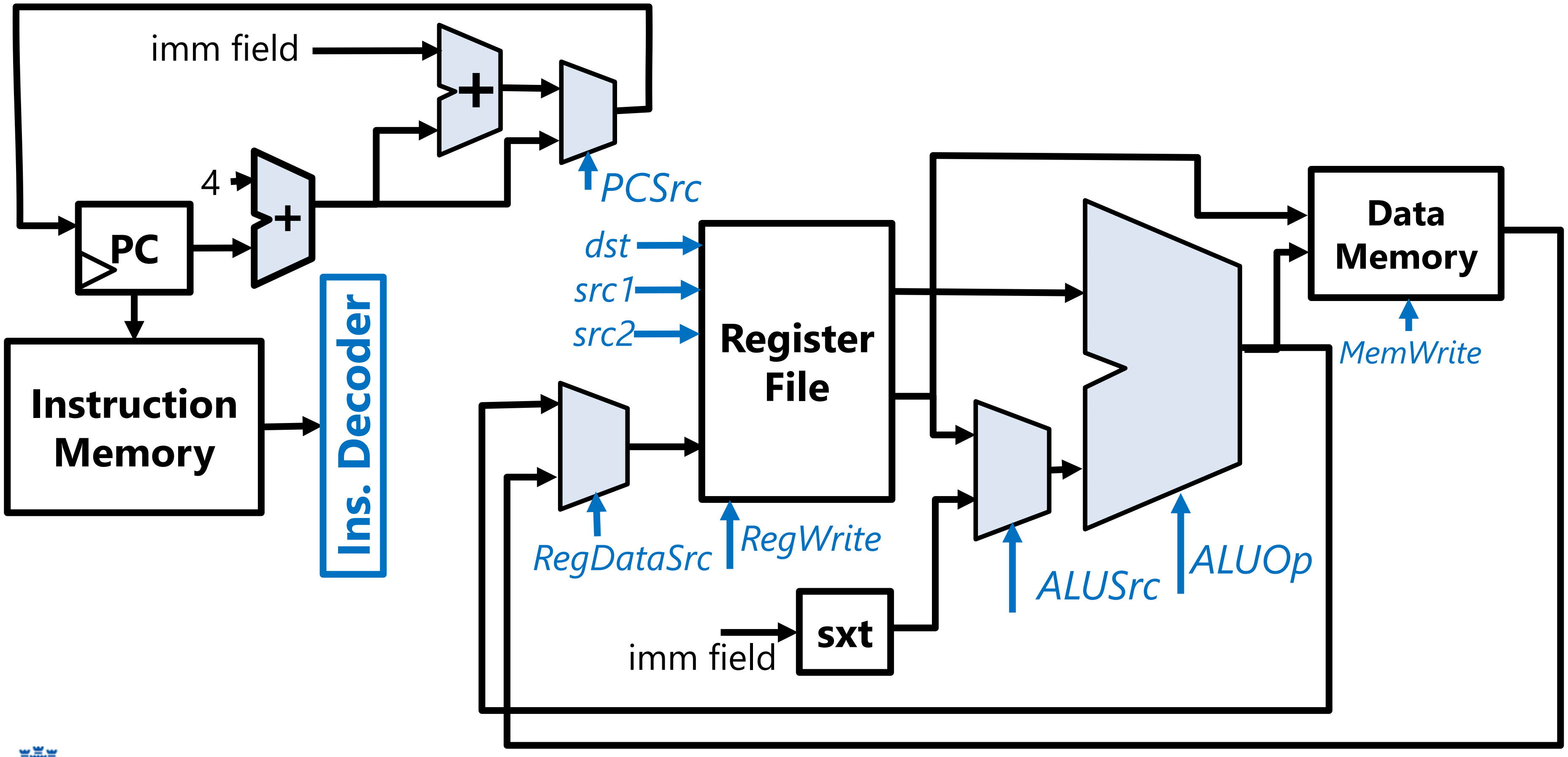
- For pedagogical (teaching) purposes

- Contains only a minimal number of instructions:
  - **lw, sw, add, sub, and, or, slt, beq,** and **j**
  - Other instructions in MIPS are variations on these anyway

- Let's review the Minimal MIPS CPU focusing on the control signals
  - Again, these control signals are decoded from the instruction

The Minimal MIPS single-cycle CPU

- A more detailed view of the 5-phase implementation

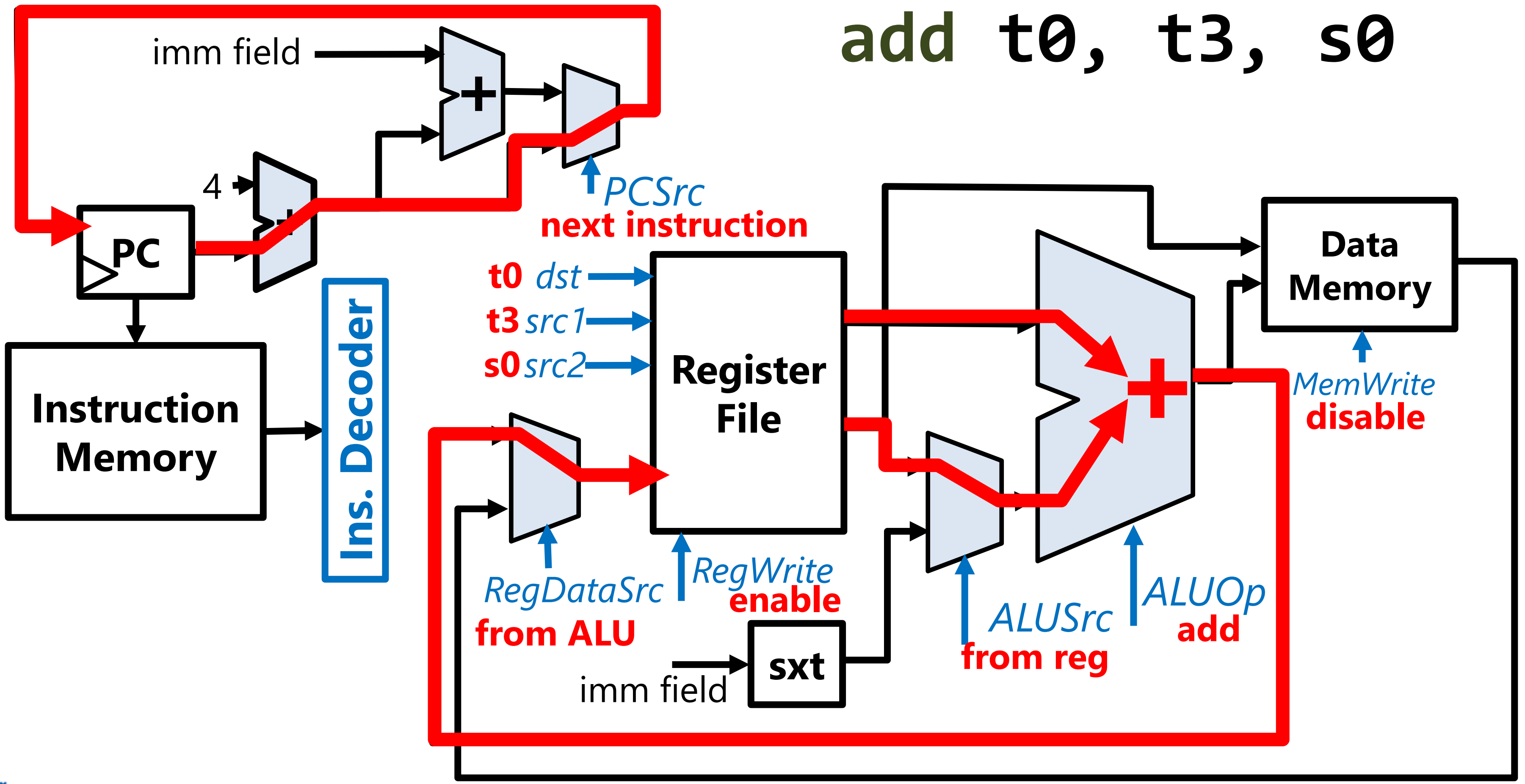


Control signals

- Registers
  - **RegDataSrc**: controls source of a register write (ALU / memory)
  - **RegWrite**: enables a write to the register file
  - **src1, src2, dst**: the register number for each respective operand
- ALU
  - **ALUSrc**: whether second operand of ALU is a register / immediate
  - **ALUOp**: controls what the ALU will do (add, sub, and, or etc)
- Memory
  - **MemWrite**: enables a write to data memory
- PC
  - **PCSrc**: controls source of next PC (PC + 4 / PC + 4 + imm)

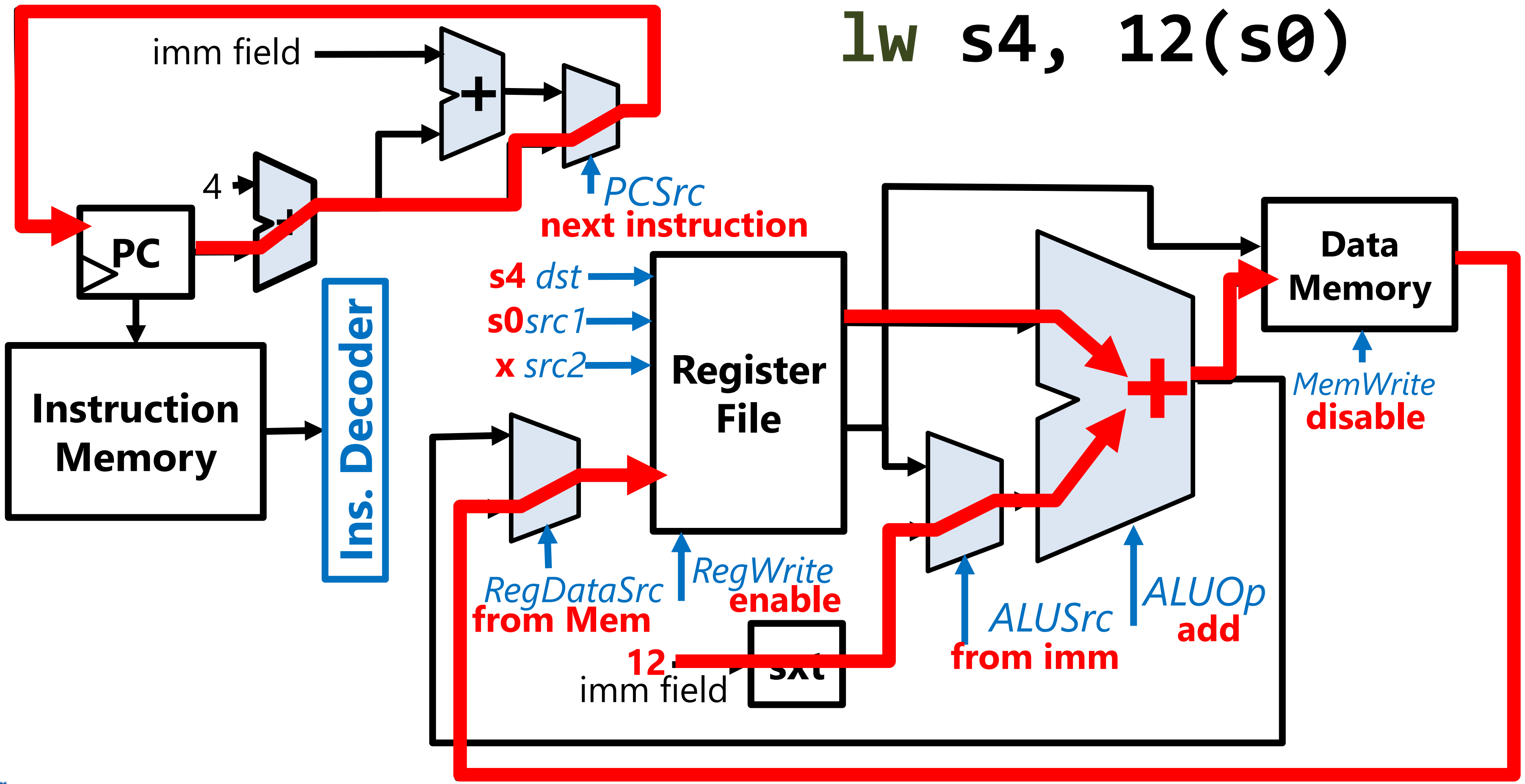
→ All these signals are decoded from the instruction!

How an **add/sub/and/or/slt** work



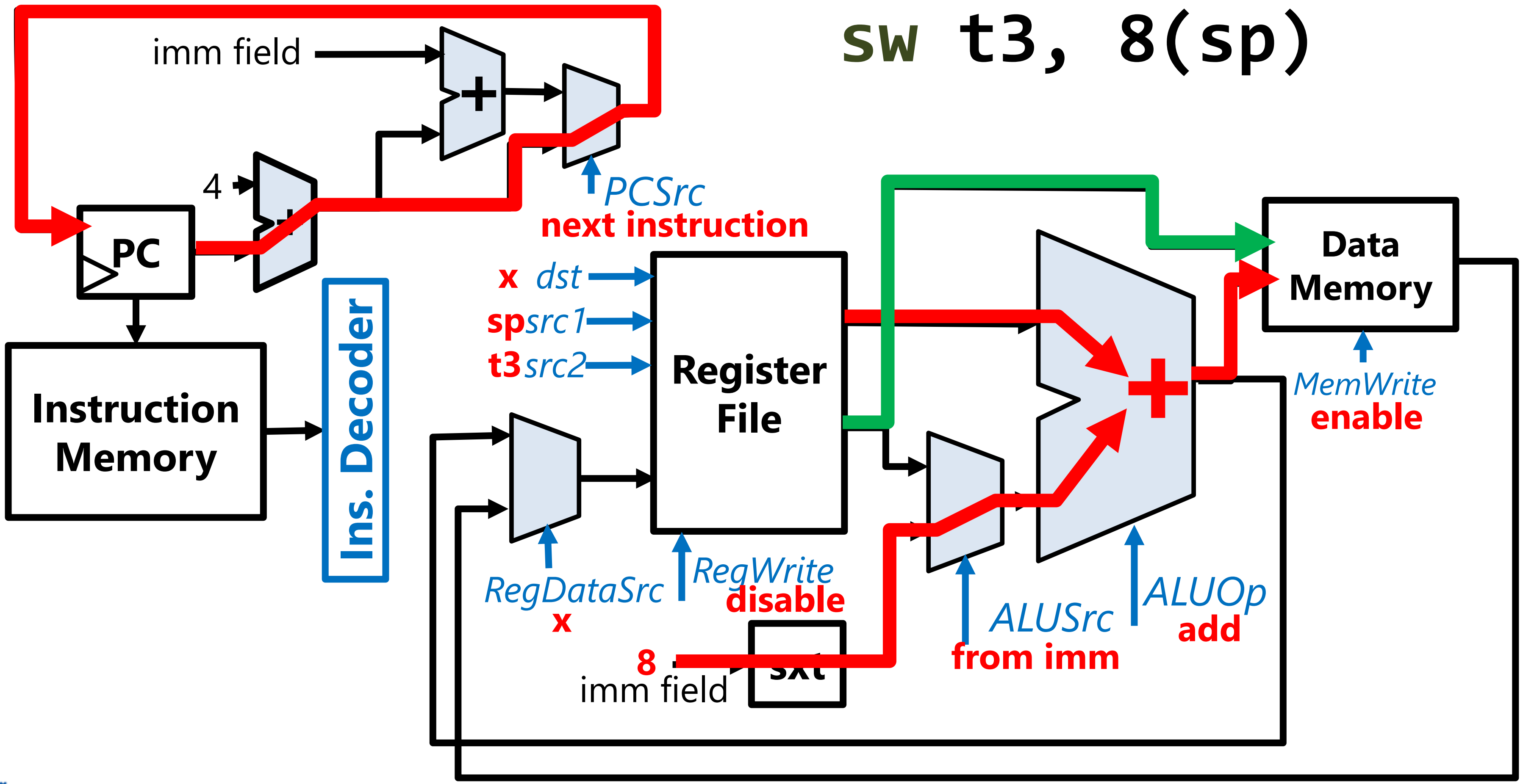
How an **lw** works

`lw s4, 12(s0)`



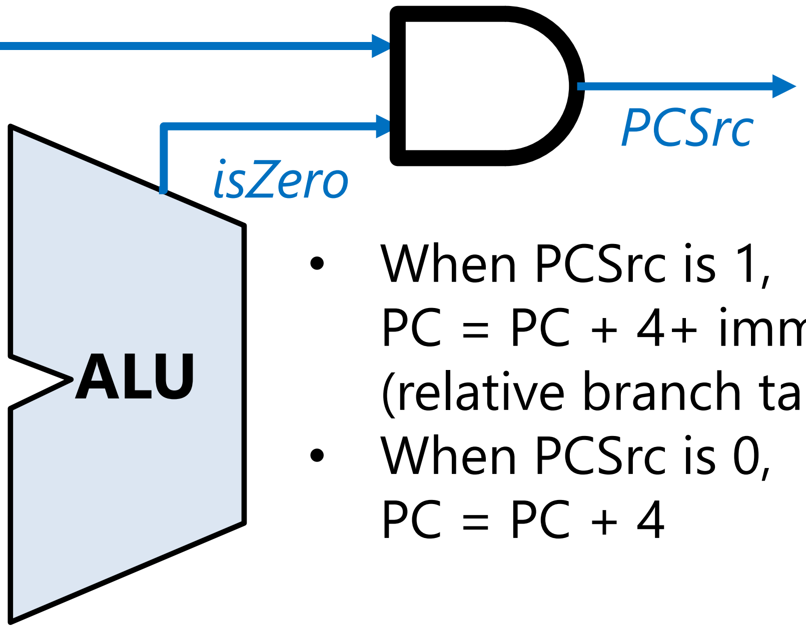
How an **sw** works

`sw t3, 8(sp)`



What about **beq**?

- Compares numbers by subtracting and see if result is 0
  - If result is 0, we set PCSrc to use the branch target.
  - Otherwise, we set PCSrc to PC + 4.



- Instruction decoder outputs isBEQ
  - 1: When instruction is beq
  - 0: When instruction not beq

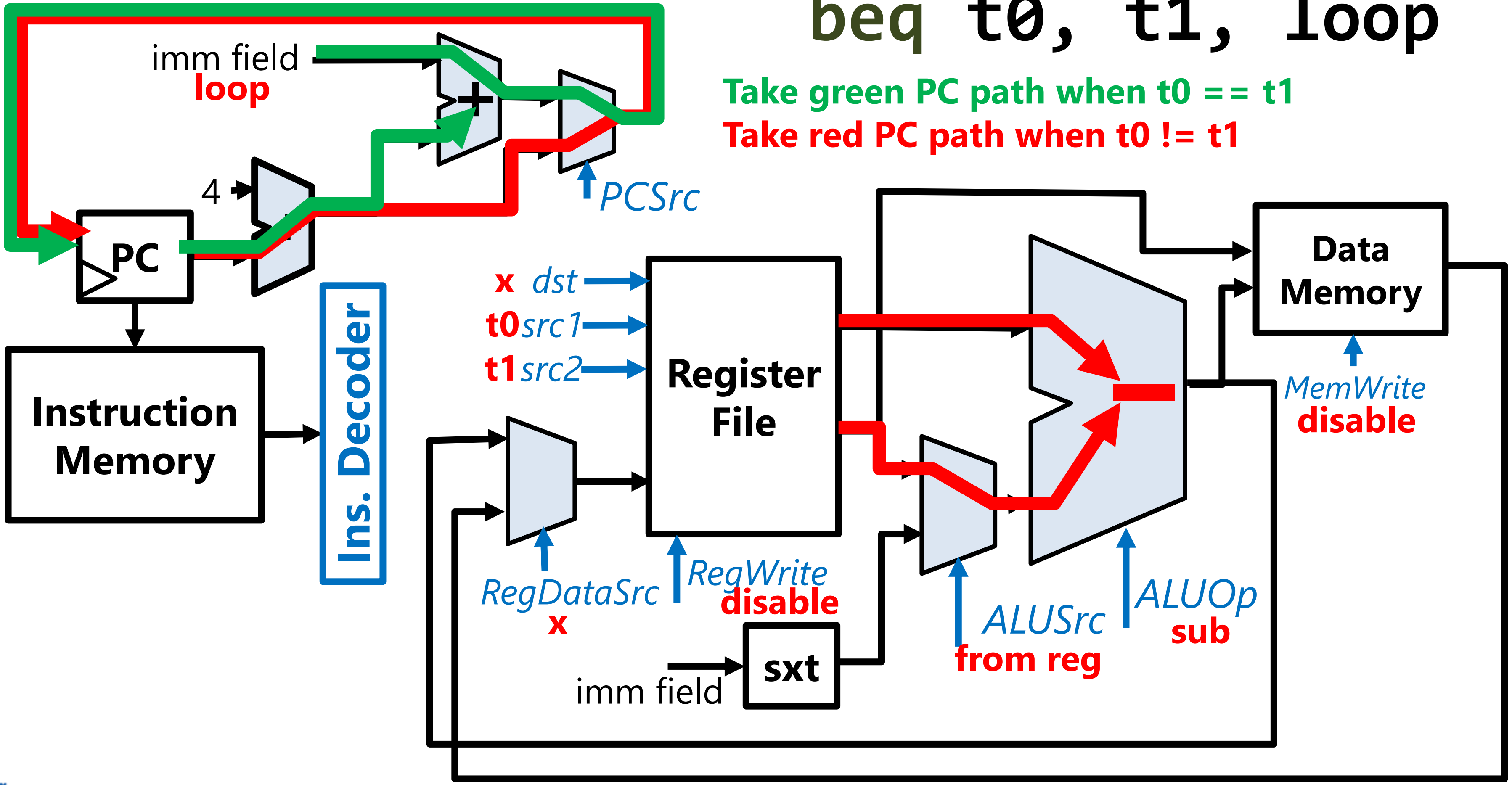
- When PCSrc is 1, PC = PC + 4+ imm (relative branch target)
- When PCSrc is 0, PC = PC + 4

How a **beq** works

`beq t0, t1, loop`

Take green PC path when t0 == t1

Take red PC path when t0 != t1

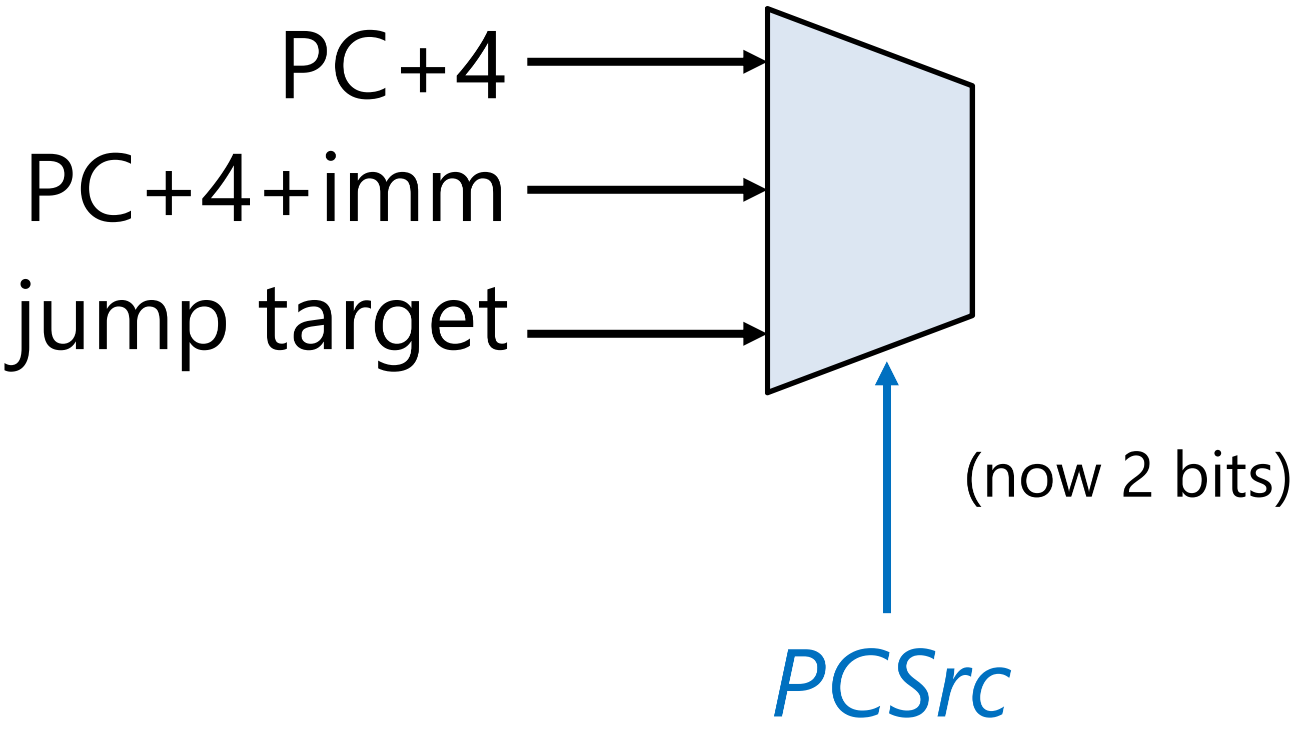


University of Pittsburgh

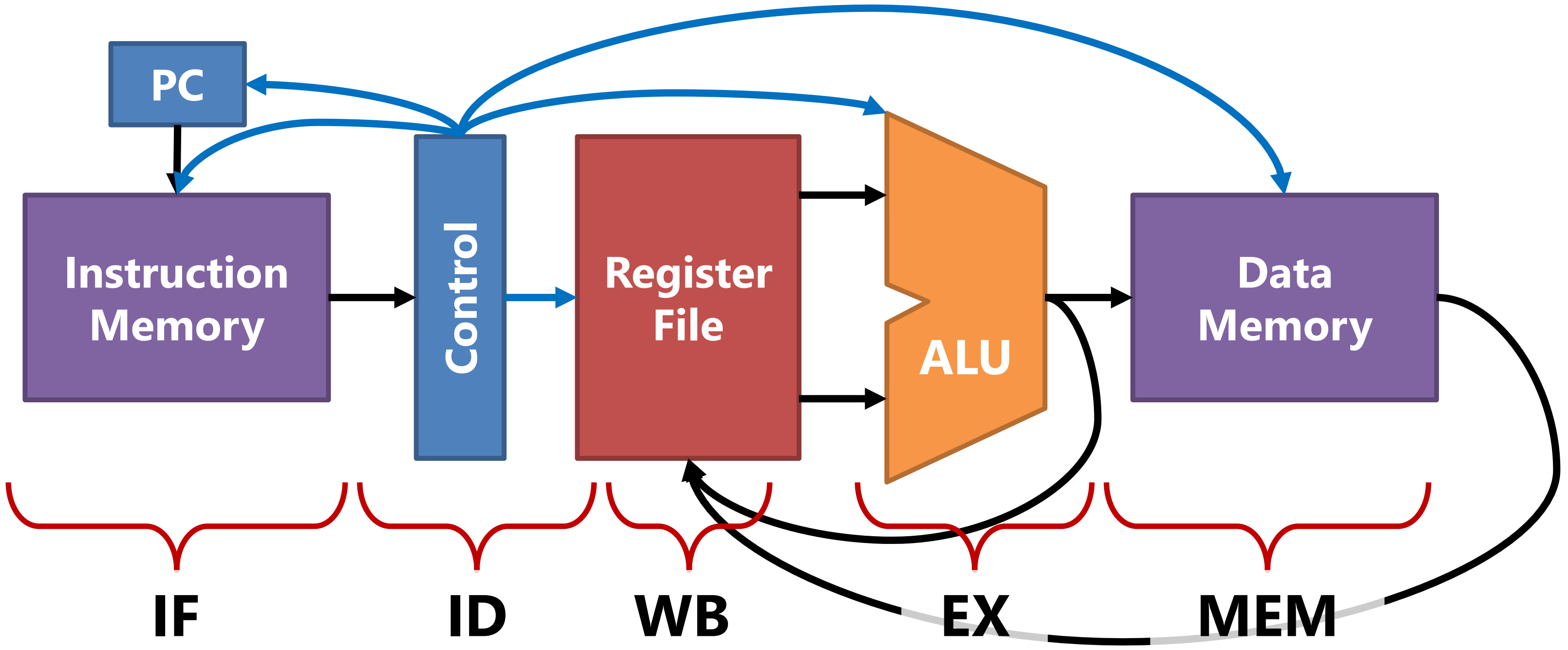
What about **j**?

- We have to add another input to the PCSrc mux.

```
j    top
```



A Single-cycle Implementation is not Optimal



- Why? Since the ***longest*** critical path must be chosen for cycle time
  - And there is a wide variation among different instructions

A Single-cycle Implementation is not Optimal

- In our CPU, the **lw** instruction has the longest critical path
  - Must go through all 5 stages: IF/ID/EX/MEM/WB
  - Whereas **add** goes through just 4 stages: IF/ID/EX/WB

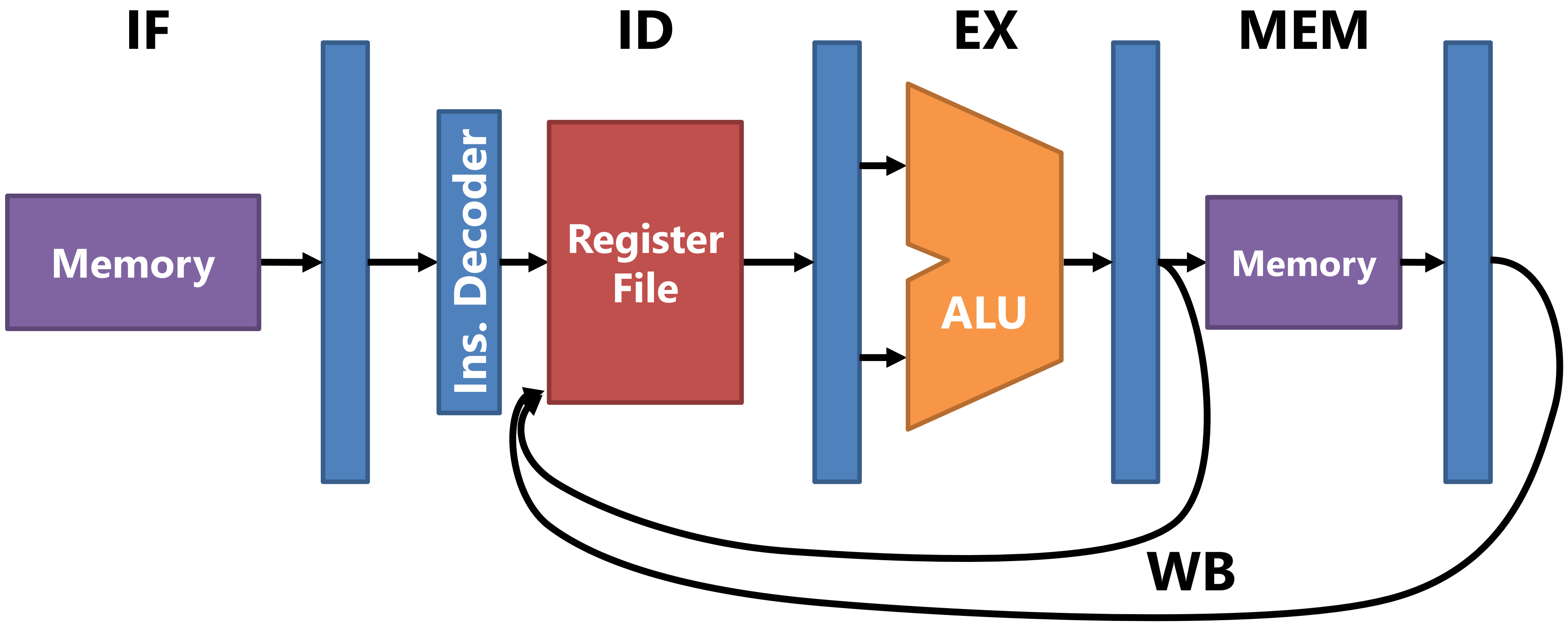
- If each phase takes *1 ns* each, cycle time must be *5 ns*:
  - Because it needs to be able to handle **lw**, which takes *5 ns*
  - **add** also takes *5 ns* when it could have been done in *4 ns*

Q) If **lw** is 1% and **add** is 99% of instruction mix,
what is the average instruction execution time?

A) Still *5 ns*! Even if **lw** is only 1% of instructions!

A Multi-cycle Implementation

- It takes one cycle for each phase through the use of internal latches



A Multi-cycle Implementation is Faster!

- Now each instruction takes different number of cycles to complete
  - **lw** takes 5 cycles: IF/ID/EX/MEM/WB
  - **add** takes 4 cycles: IF/ID/EX/WB

- If each phase takes *1 ns* as before:
  - **lw** takes *5 ns* and **add** takes *4 ns*

Q) If **lw** is 1% and **add** is 99% of instruction mix,
what is the average instruction execution time?

A) 0.01 * *5 ns* + 0.99 * *4 ns* = *4.01 ns* (25% faster than single cycle)

** Caveat: delay due to the added latches not shown, but net win*

And we can do even better!

- Did you notice?
  - When an instruction is on a particular phase (e.g. IF) …
  - … other phases (ID/EX/MEM/WB) are not doing any work!

- Our CPU is getting chronically ***underutilized***!
  - If CPU is a factory, 80% (4/5) of the workers are idling!

- Car factories create an assembly line to solve this problem
  - No need to wait until a car is finished before starting on next one
  - Our CPU is going to use a ***pipeline*** (similar concept)