Lecture-5 (Pipelining)
CS422-Spring 2018

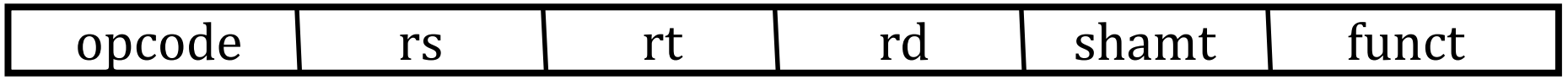
Biswa@CSE-IITK

Before That: Single Cycle Design

An R-format single Cycle CPU

Syntax: ADD $8 $9 $10

Semantics: $8 = $9 + $10

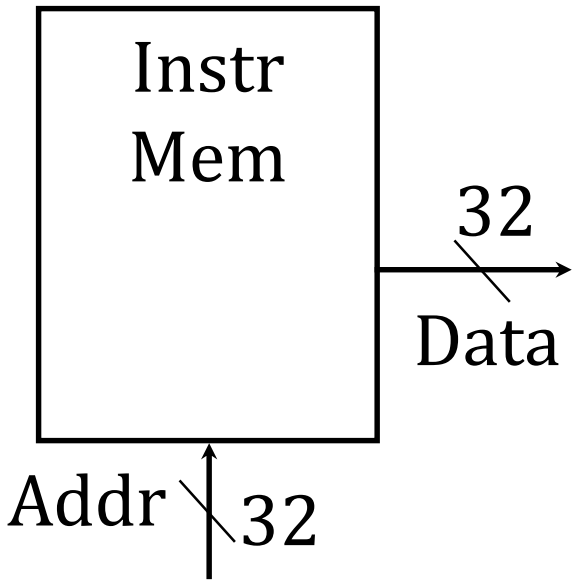


Sample program:
ADD $8 $9 $10
SUB $4 $8 $3
AND $9 $8 $4
…

How registers get their initial values are not of concern to us right now.

- No branches or jumps: machine only runs straight line code.
- No loads or stores: machine has no use for data memory, only instruction memory.

Instruction Memory

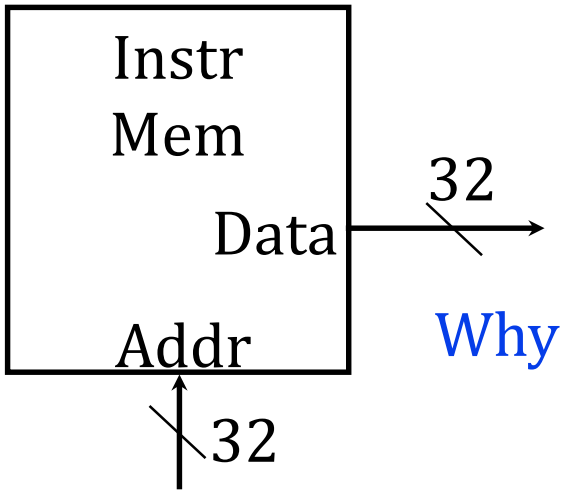


Reads are combinational: Put a stable address on input, a short time later data appears on output.

Not concerned about how programs are loaded into this memory.

Related to separate instruction and data caches in "real" designs.

Let's Fetch It



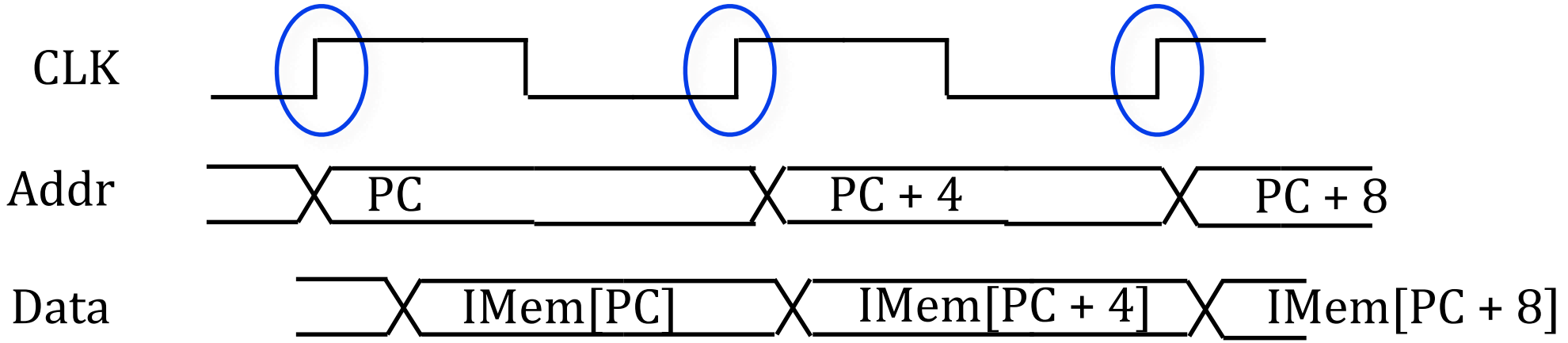
Fetching straight-line MIPS instructions requires a machine that generates this timing diagram:

Why increment every cycle?

Why +4 and not +1?

Straight-line code.

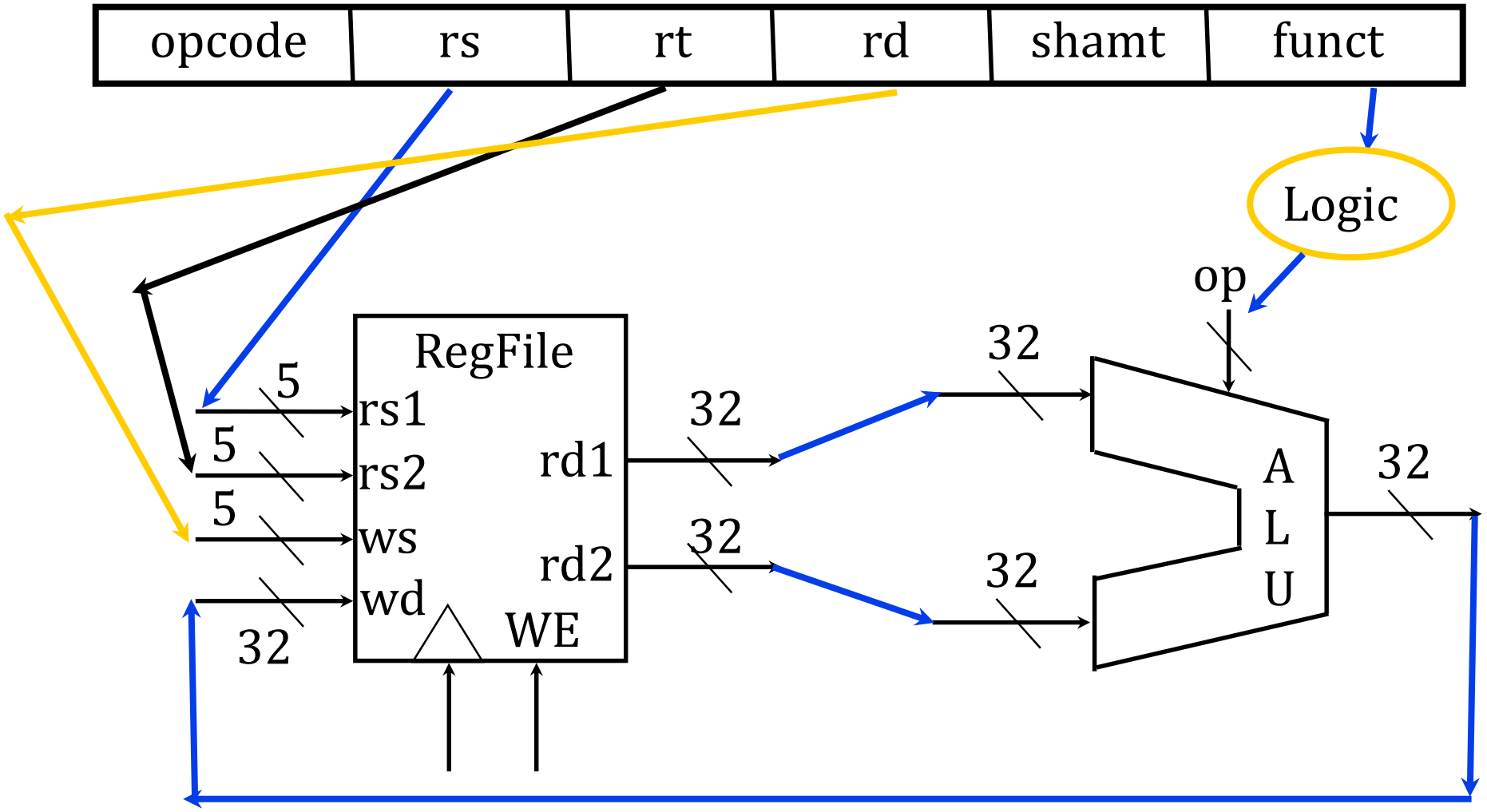
32-bit instructions.



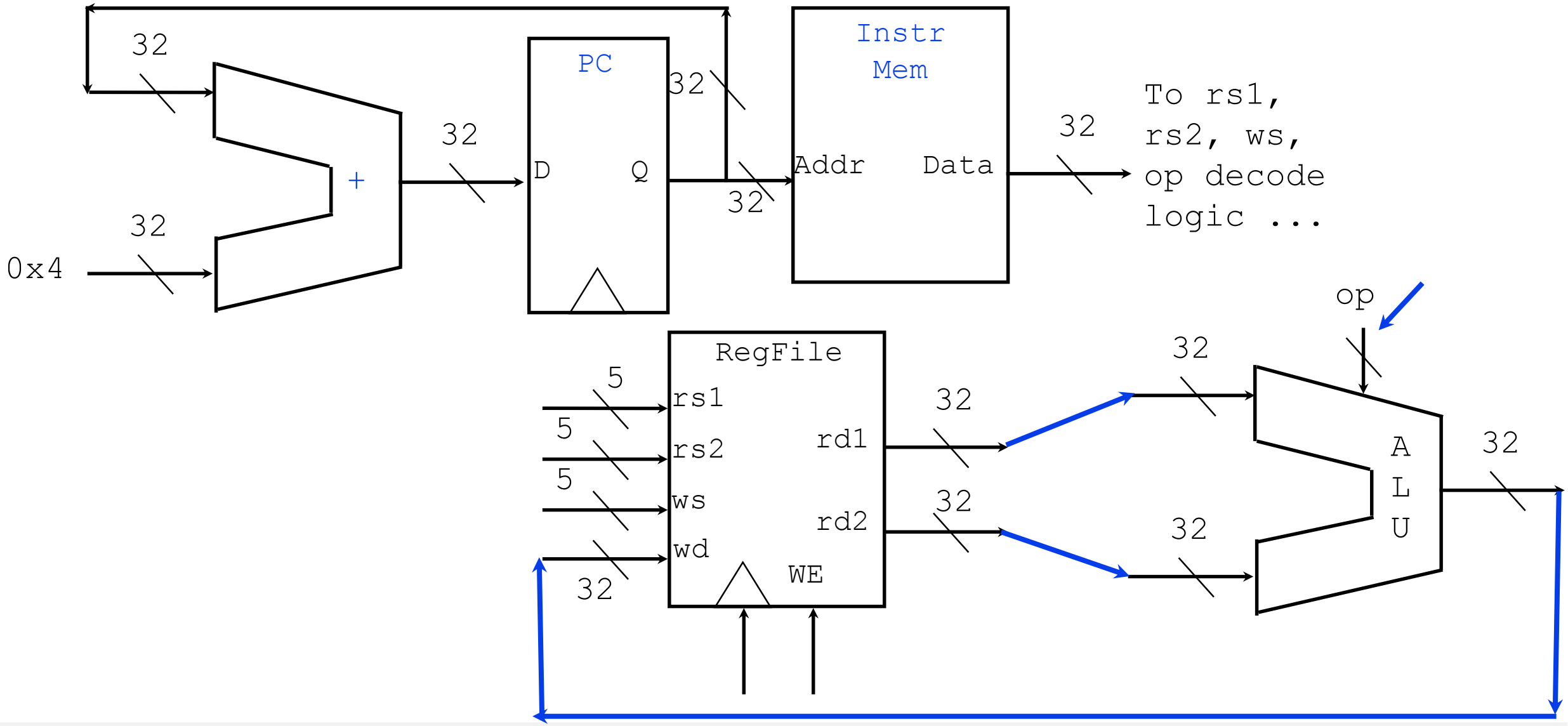
PC == Program Counter, points to next instruction.

Decode & Execute

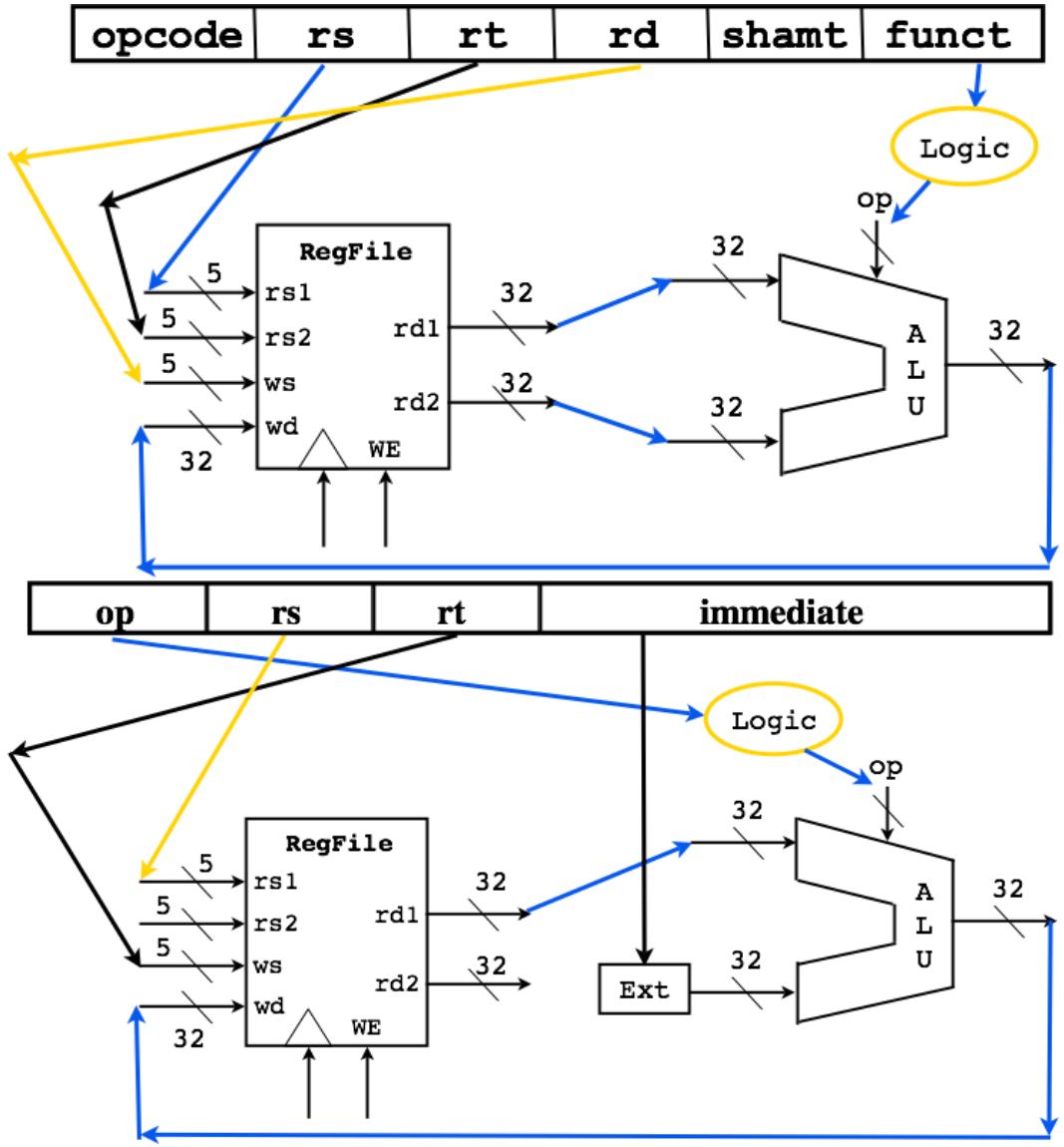
Decode fields to get : ADD $8 $9 $10



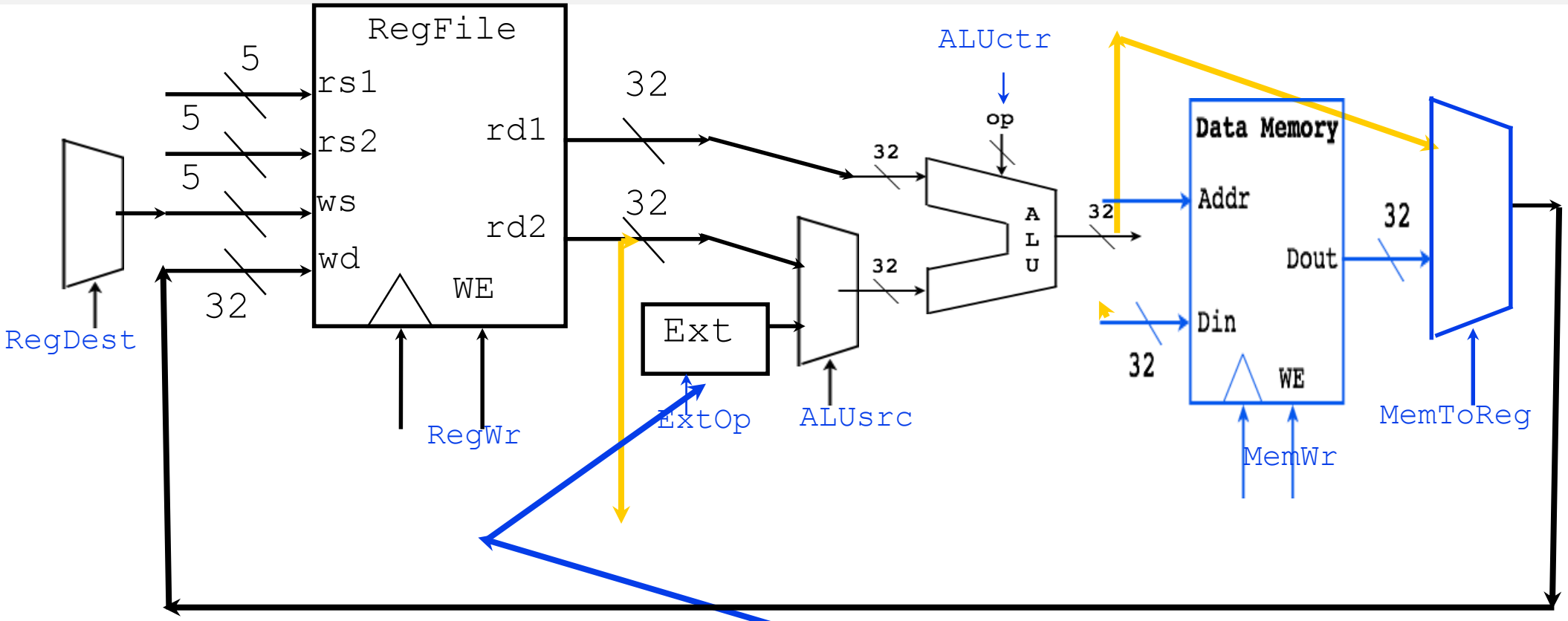
Let's Put It Together



R TO I Format


opcode
rs
rt
rd
shamt
funct
Logic
op
RegFile
rs1
rs2
ws
wd
WE
rd1
rd2
5
32
A
L
U
op
rs
rt
immediate
Logic
Ext

Adding Data Memory

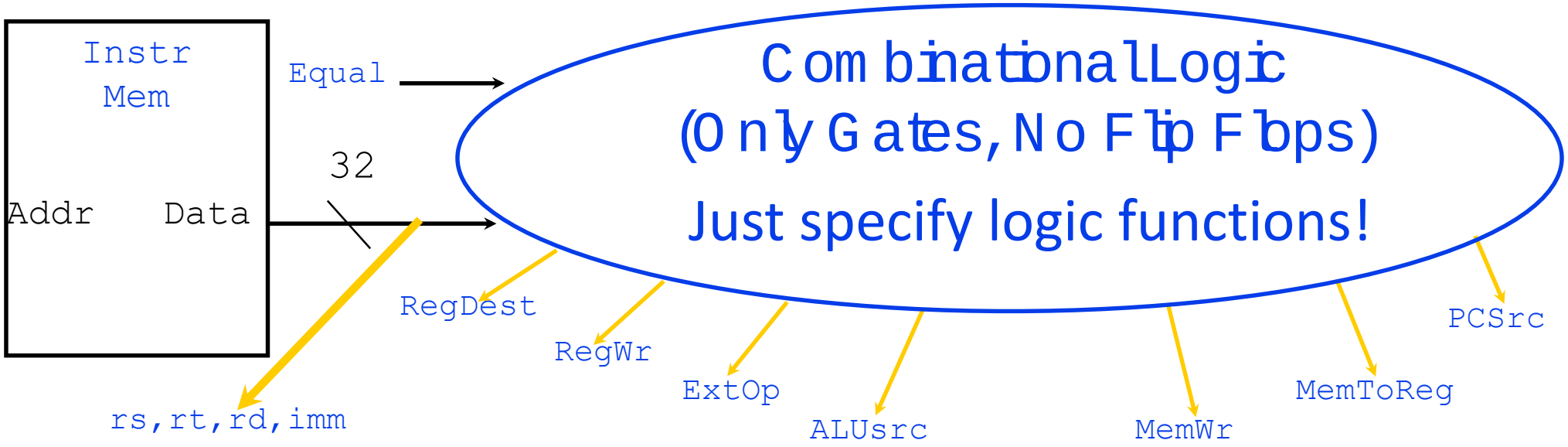


Syntax: LW $1, 32($2)

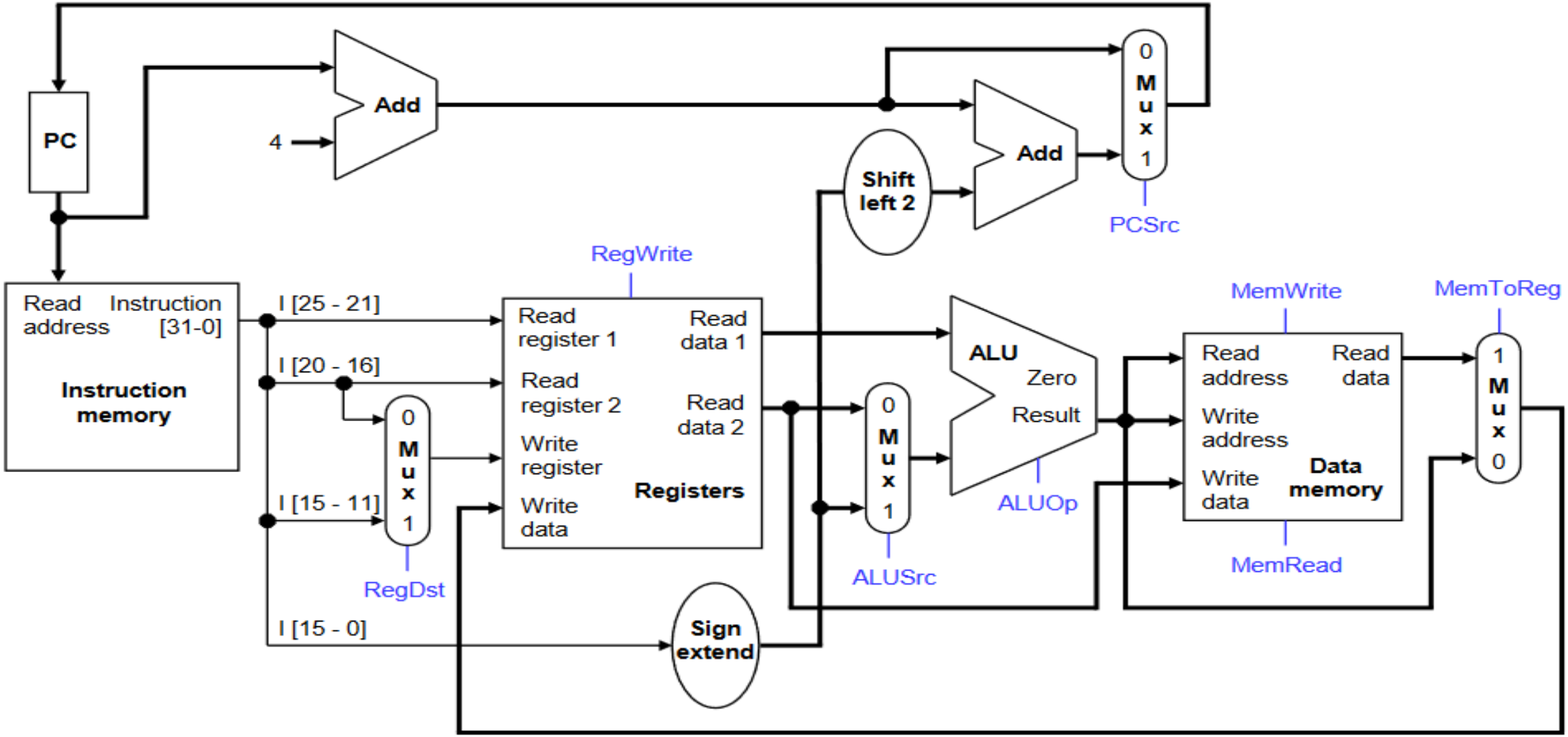
op	rs	rt	immediate

Action: $1 = M[$2 + 32]

What is Single Cycle Here?



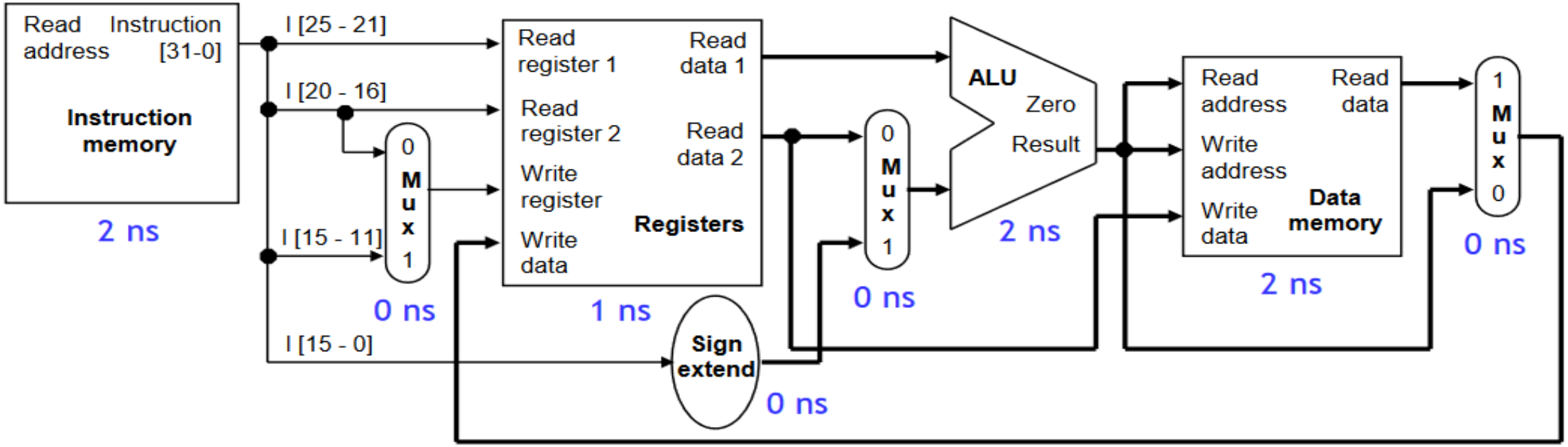
Let's Understand This



Implications?

- For example, lw $t0, -4($sp) needs 8ns, assuming the delays shown here.

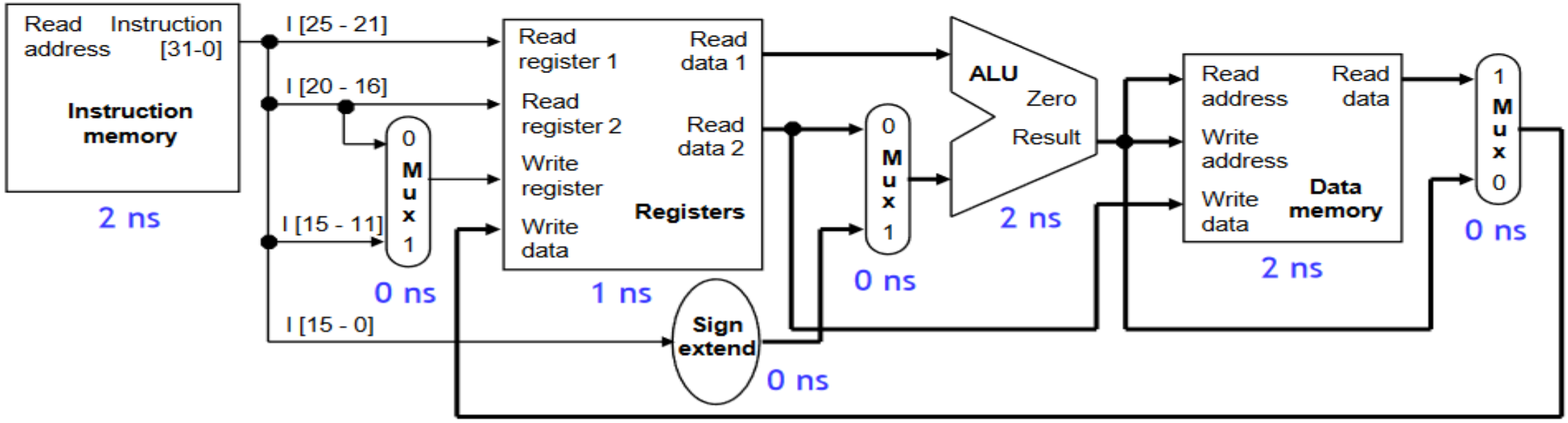
reading the instruction memory	2ns	8ns
reading the base register $sp	1ns	
computing memory address $sp-4	2ns	
reading the data memory	2ns	
storing data back to $t0	1ns	



Implications?

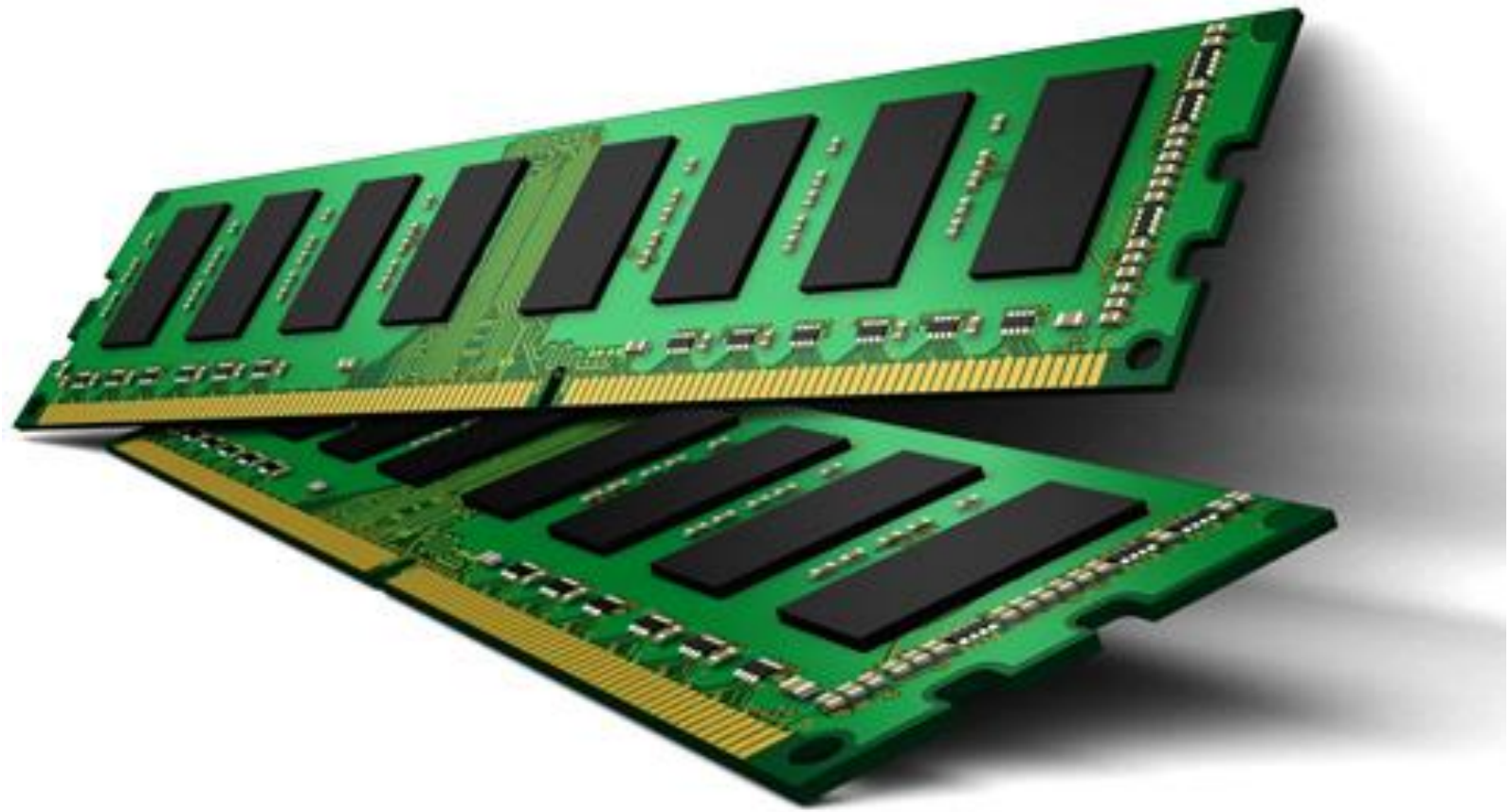
- For example, the instruction add $s4, $t1, $t2 really needs just 6ns.

reading the instruction memory	2 ns	6 ns
reading registers $t1 and $t2	1 ns	
computing $t1 + $t2	2 ns	
storing the result into $s0	1 ns	



So 8ns enough?

It's the Memory Stupid (~ 50 ns) – Oh NO!!



So, frequency of ~ 10MHz

But Confucius says "Make the common case first" ☺

World of Latency, Throughput, Parallelism

- Latency
  - time it takes to complete one instance

- Throughput
  - number of computations done per unit time

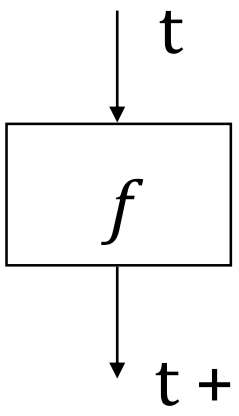
Pipelining: Goal

- Goal: Increase machine *throughput* by making better use of available hardware resources
- Pipelining increases throughput at the cost of latency
  - Hopefully not too high of a cost

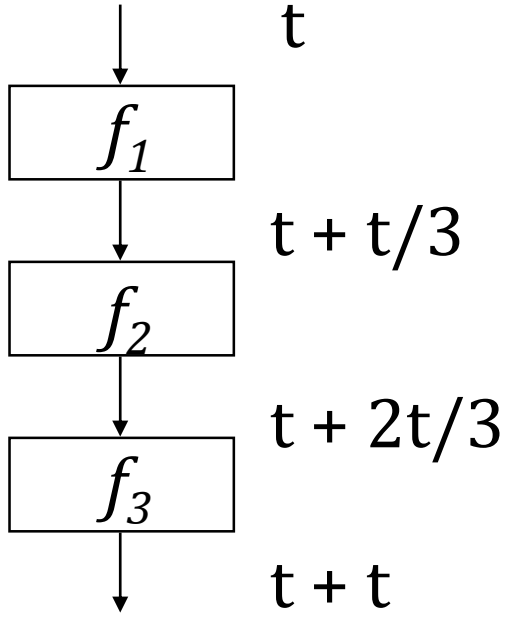
- Assumptions
  - Idle hardware resources exist
    - parallelism!
  - Work available
    - parallelism!

How?

- Partition hardware function into sub-functions so overlap can occur
- Ideally each sub-function same time

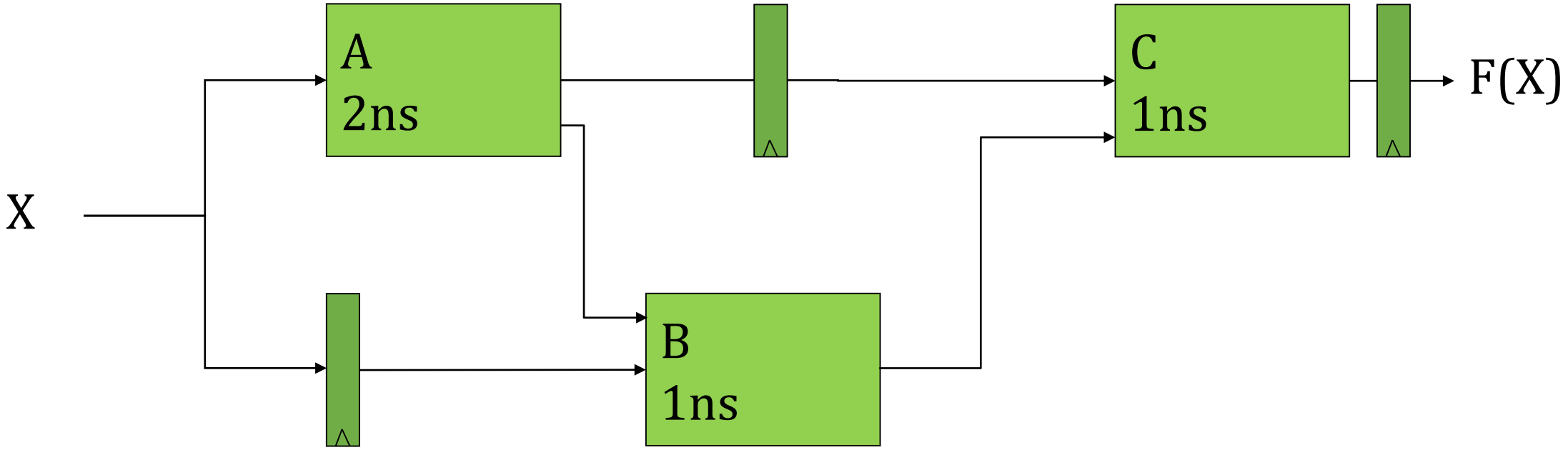


Issue op every t time units



Issue op every t/3 time units (may be slightly more than t/3-- why?)

Malformed Pipeline

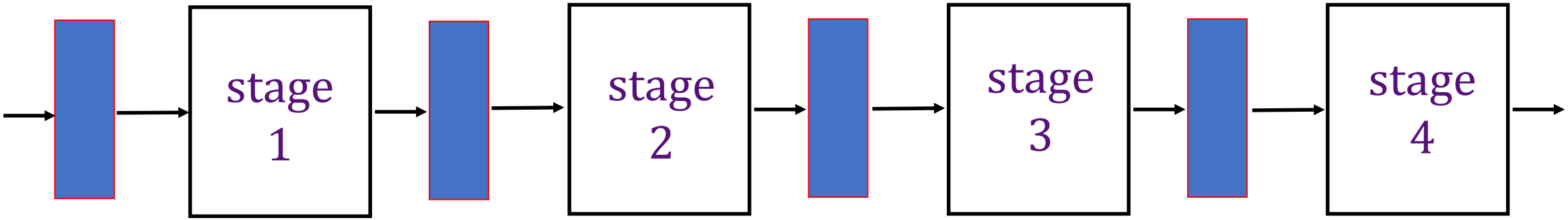


How many stage pipeline is this?
What is the problem?

Ideal Pipeline

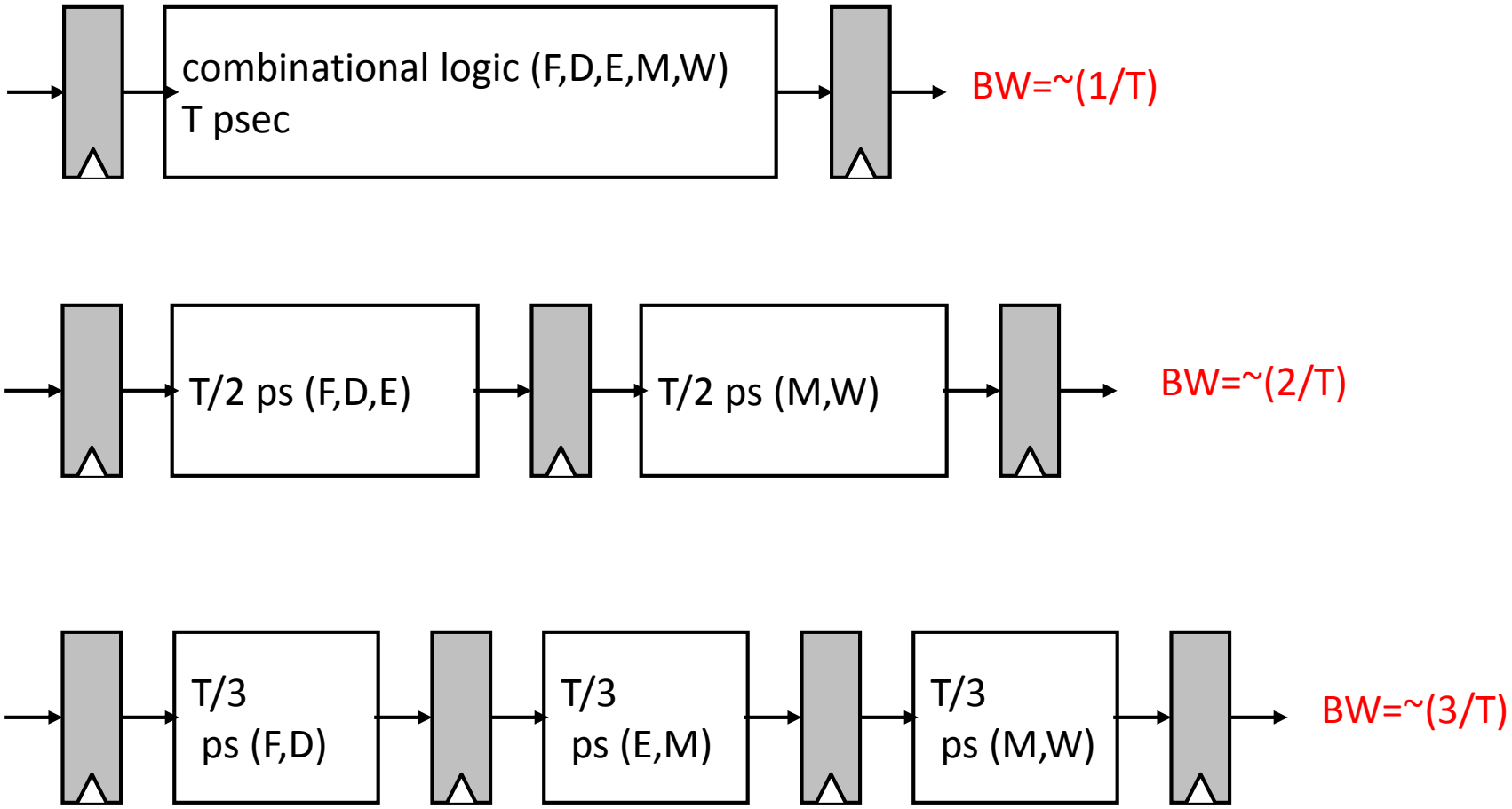
- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry

Ideal Pipeline



- All objects go through the same stages
- No sharing of resources between any two stages
- Propagation delay through all pipeline stages is equal
- The scheduling of an object entering the pipeline is not affected by the objects in other stages

Ideal Pipeline



More Realistic One

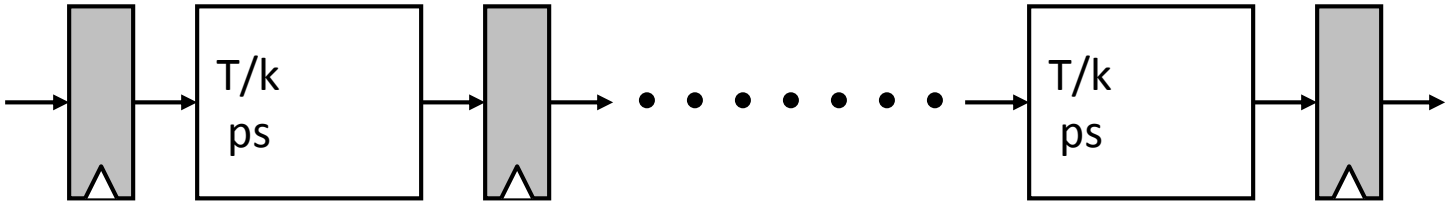
- Nonpipelined version with delay T

  BW = 1/(T+S) where S = latch delay

- k-stage pipelined version

  $BW_{k\text{-stage}} = 1 / (T/k + S)$

  $BW_{max} = 1 / (1 \text{ gate delay} + S)$



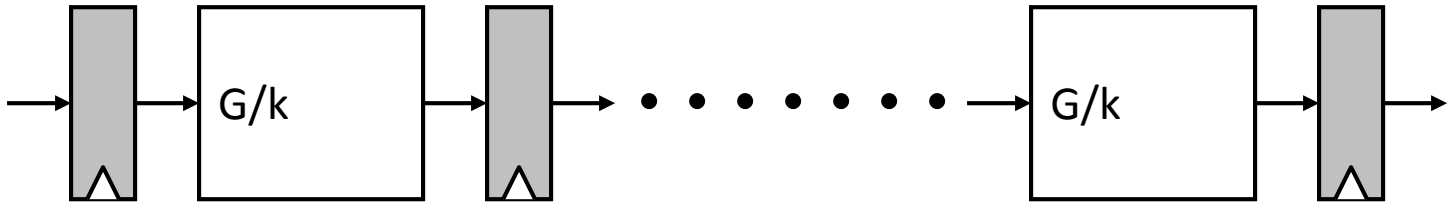
More Realistic One

- Nonpipelined version with combinational cost G

  Cost = G+L where L = latch cost

- k-stage pipelined version

  $Cost_{k\text{-}stage} = G + Lk$



World Is Not The Ideal One

- Identical operations ... NOT!

  ⇒ different instructions do not need all stages

  - Forcing different instructions to go through the same multi-function pipe

  → external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!

  ⇒ difficult to balance the different pipeline stages

  - Not all pipeline stages do the same amount of work

  → internal fragmentation (some pipe stages are too-fast but take the same clock cycle time)

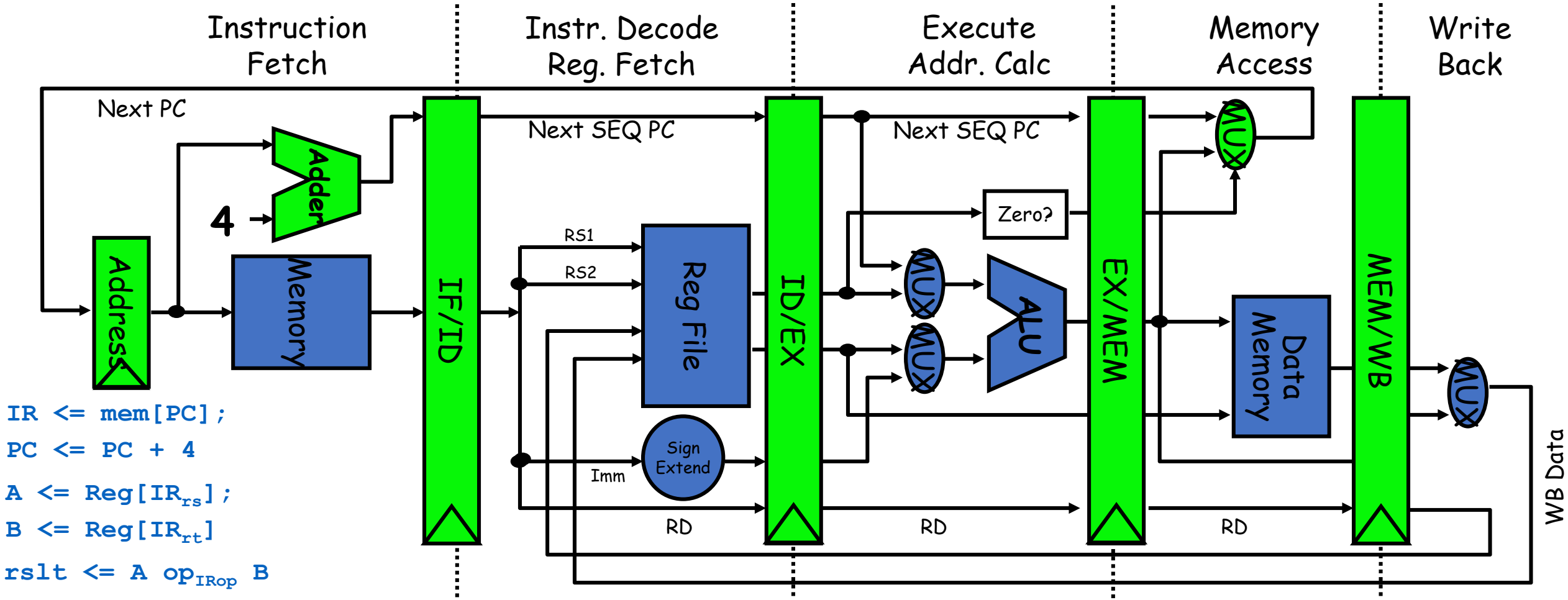
- Independent operations ... NOT!

  ⇒ instructions are not independent of each other

  - Need to detect and resolve inter-instruction dependencies to ensure the pipeline operates correctly → Pipeline is not always moving (it stalls)

Let’s Digress a Bit: Latch and Flip-flop (Piazza: +1)

Simple 5-stage Pipeline




```
IR <= mem[PC];
PC <= PC + 4
A <= Reg[IR_rs];
B <= Reg[IR_rt]
rslt <= A op_IRop B
WB <= rslt
Reg[IR_rd] <= WB
```

- Data stationary control
  - local decode for each instruction phase / pipeline stage

Visualise It