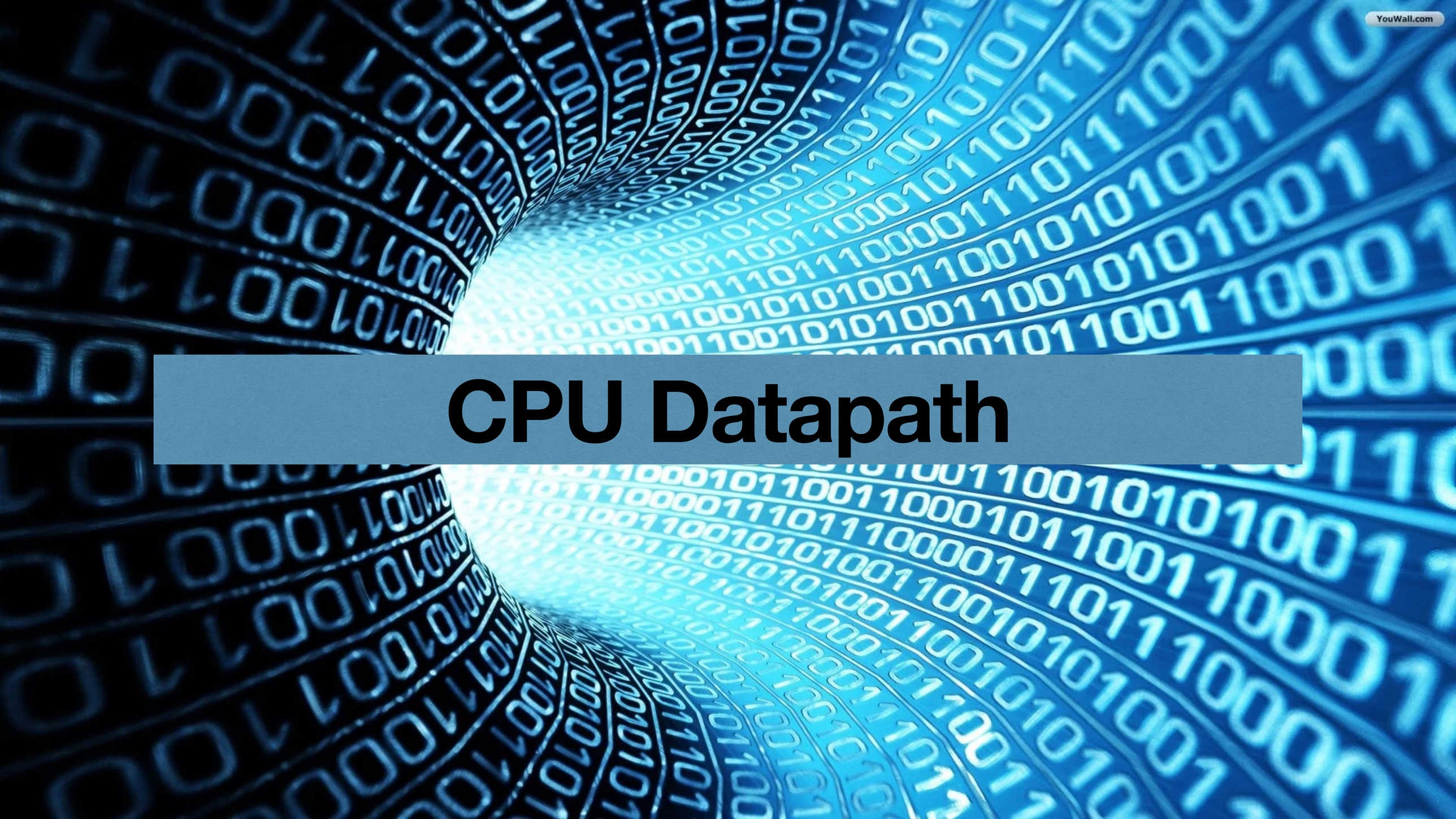
# Digital Logic Design + Computer Architecture

**Sayandeep Saha**

**Assistant Professor**
**Department of Computer Science and Engineering**
**Indian Institute of Technology Bombay**




# CPU Datapath

# What is “Datapath” for a Processor?

- Same as the datapath control split we studied for GCD processor.
  - **Datapath processes the data**
  - **Controller tells how to process the data**
- But here it’s for processing the instructions in an ISA
- A datapath that can process all the instructions in an ISA


Image generated by ChatGPT

# Datapath Elements

- We shall begin with the simplest case
  - Every instruction finishes in **one clock cycle**
  - The clock frequency is determined by the instruction finishes last
- Remember, again how we constructed the GCD datapath
- Let’s first identity what are the components needed

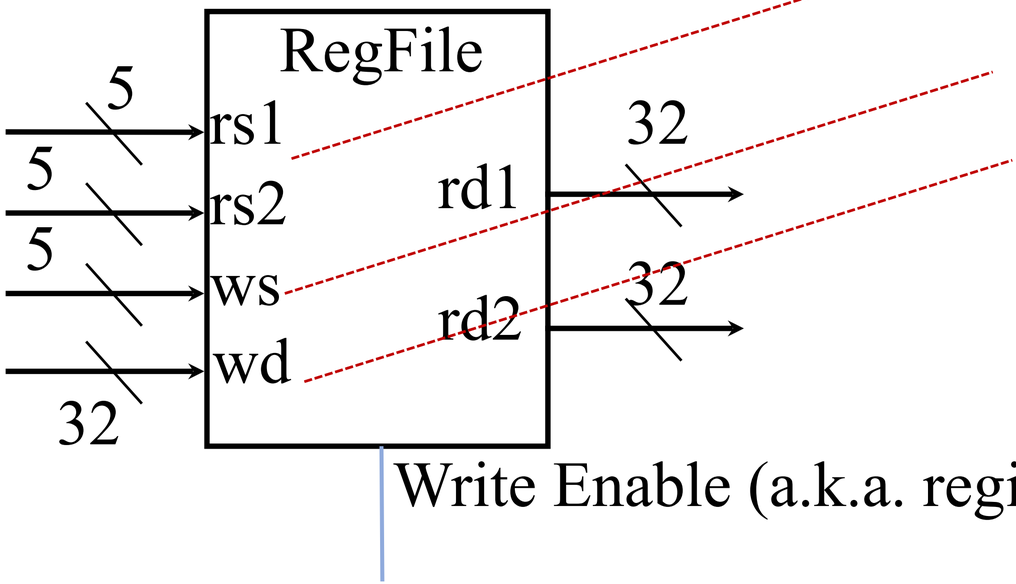

Image generated by ChatGPT

# Datapath Elements

- Remember, again how we constructed the GCD datapath
- Let’s first identity what are the components needed


Image generated by ChatGPT

# Datapath Elements: Register File

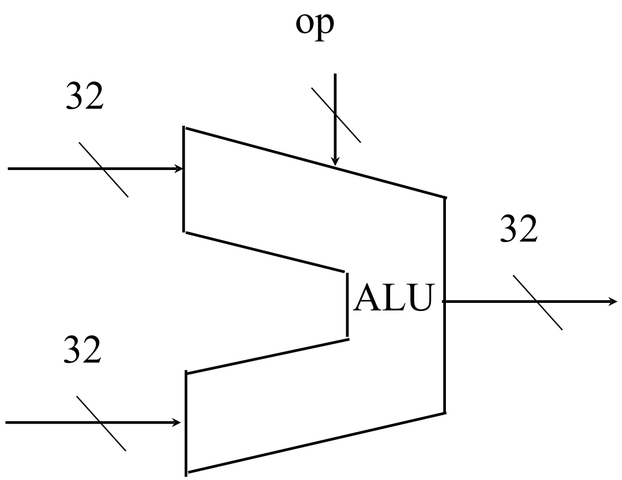



rs1 and rs2: Read register 1 and 2

ws: Write register

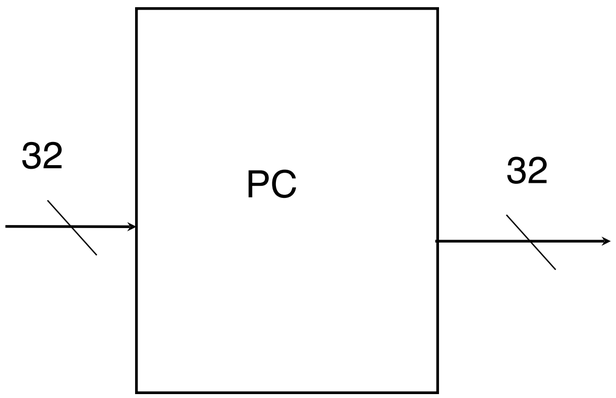
wd: Write data

# Datapath Elements: The ALU



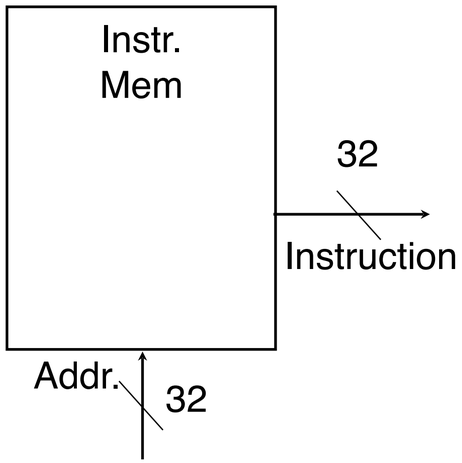

- All arithmetic and logical operation happens here
- Operation selection
- E.g.,
  - Op = 0: Y = A+B
  - OP = 1: Y = A- B
  - OP = 2; Y = A*B

# Datapath Elements: Program Counter




- Remember PC register??
- It always points to the next instruction to be executed…
- But where is the next instruction???

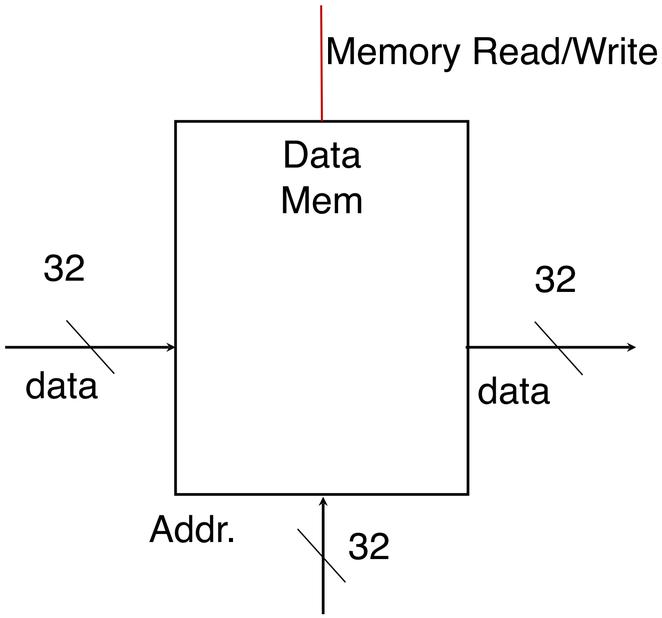
# Datapath Elements: Instruction Memory




Remember: No writes to instruction memory

Not concerned about how programs are loaded into this memory.

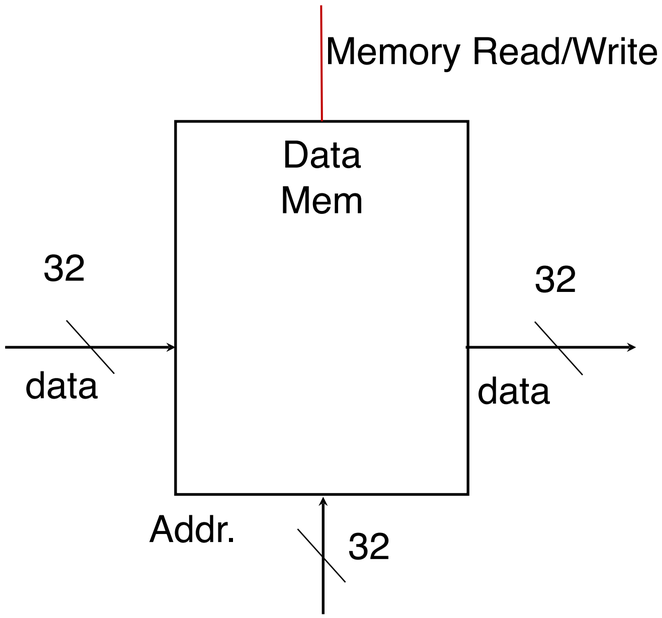
# Datapath Elements: Data Memory




Why data and instruction memory and not one memory?

- Recall Harvard vs. Von Neuman
- Not so simple matter, will discuss later.

# Datapath Elements: Data Memory




Why data and instruction memory and not one memory?

- Recall Harvard vs. Von Neuman
- Not so simple matter, will discuss later.

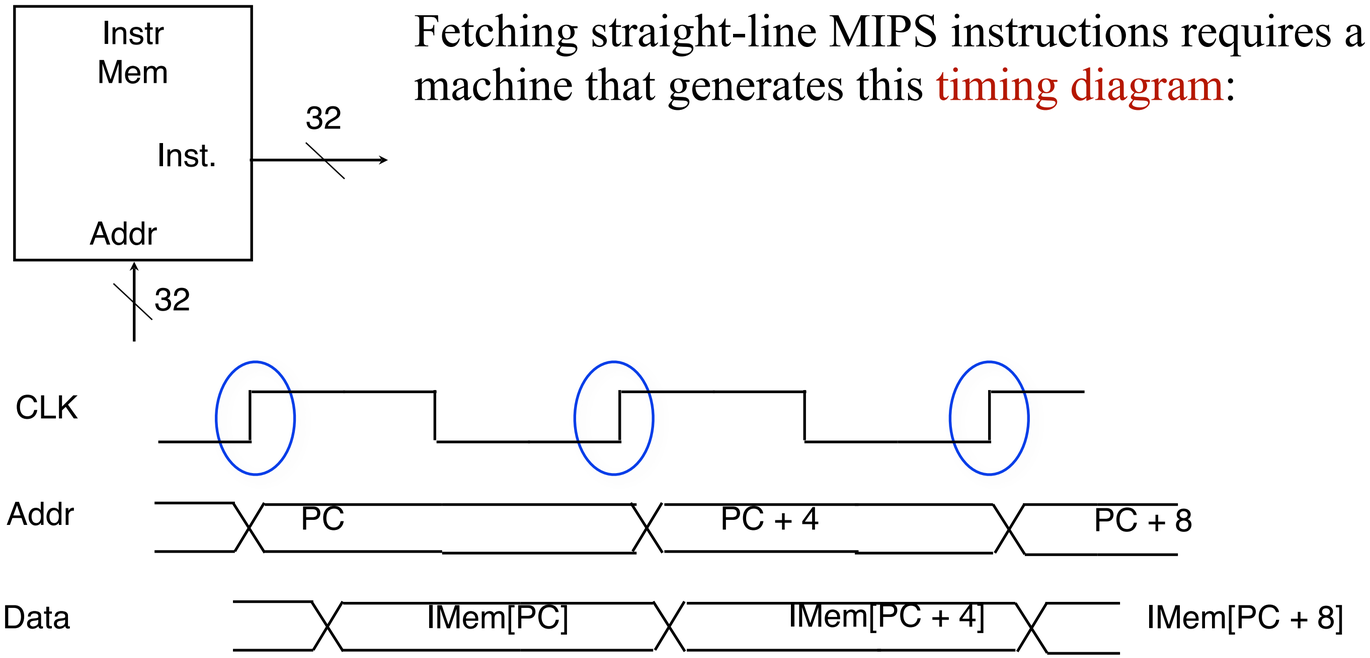
# Datapath Elements: Buses

- Same as your public transport
- Transfer data and instructions
- Separate buses for address and data

# Datapath Elements: Buses

- Same as your public transport
- Transfer data and instructions
- Separate buses for address and data
  - Why???

# Datapath Elements: Timing Diagram

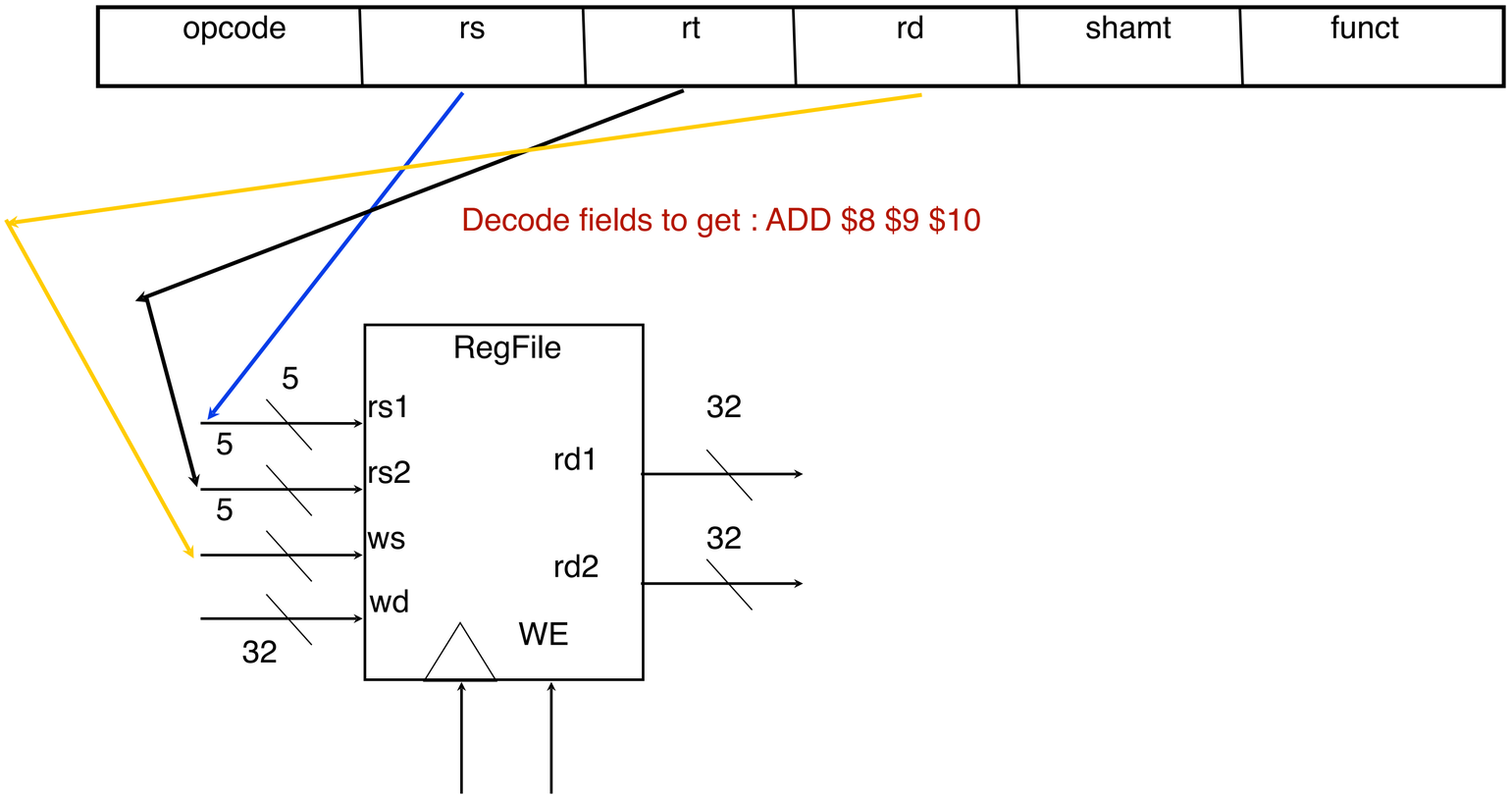



Fetching straight-line MIPS instructions requires a machine that generates this timing diagram:

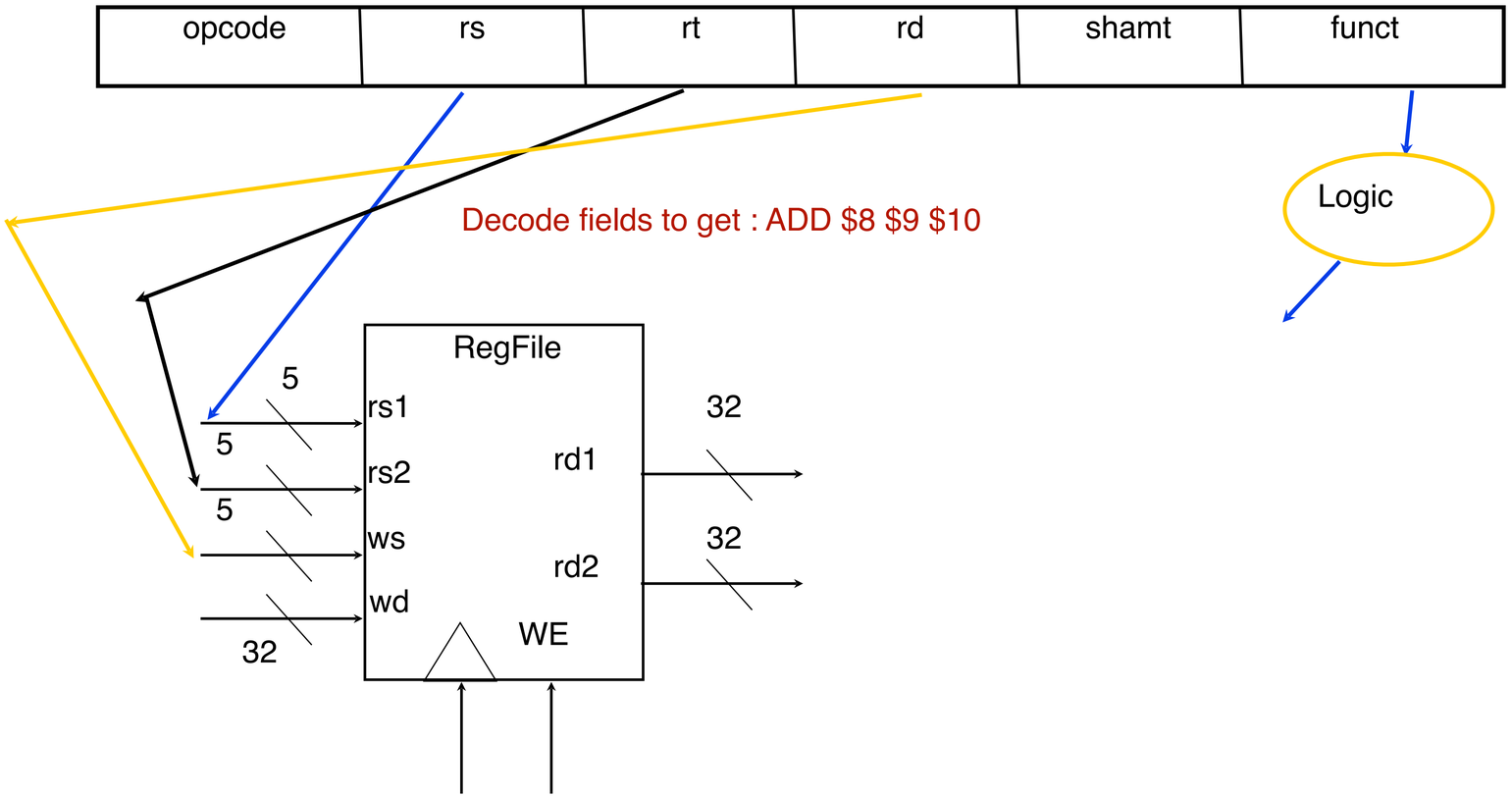
- Every clock cycle we process one instruction (super simple)
- Sending the PC and getting the instruction from memory is called **Instruction Fetch**

PC == Program Counter, points to next instruction.

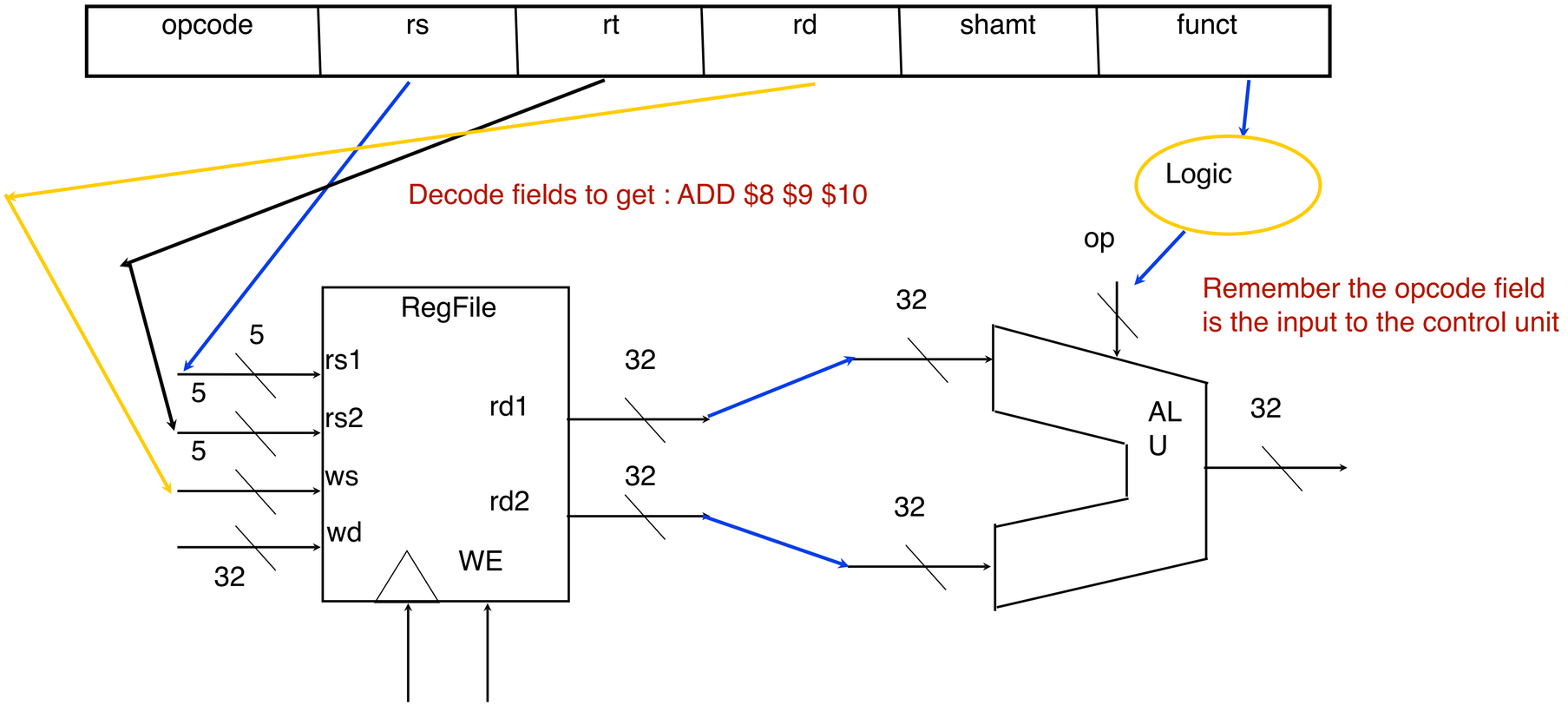
# Datapath Elements: Decoding Instructions



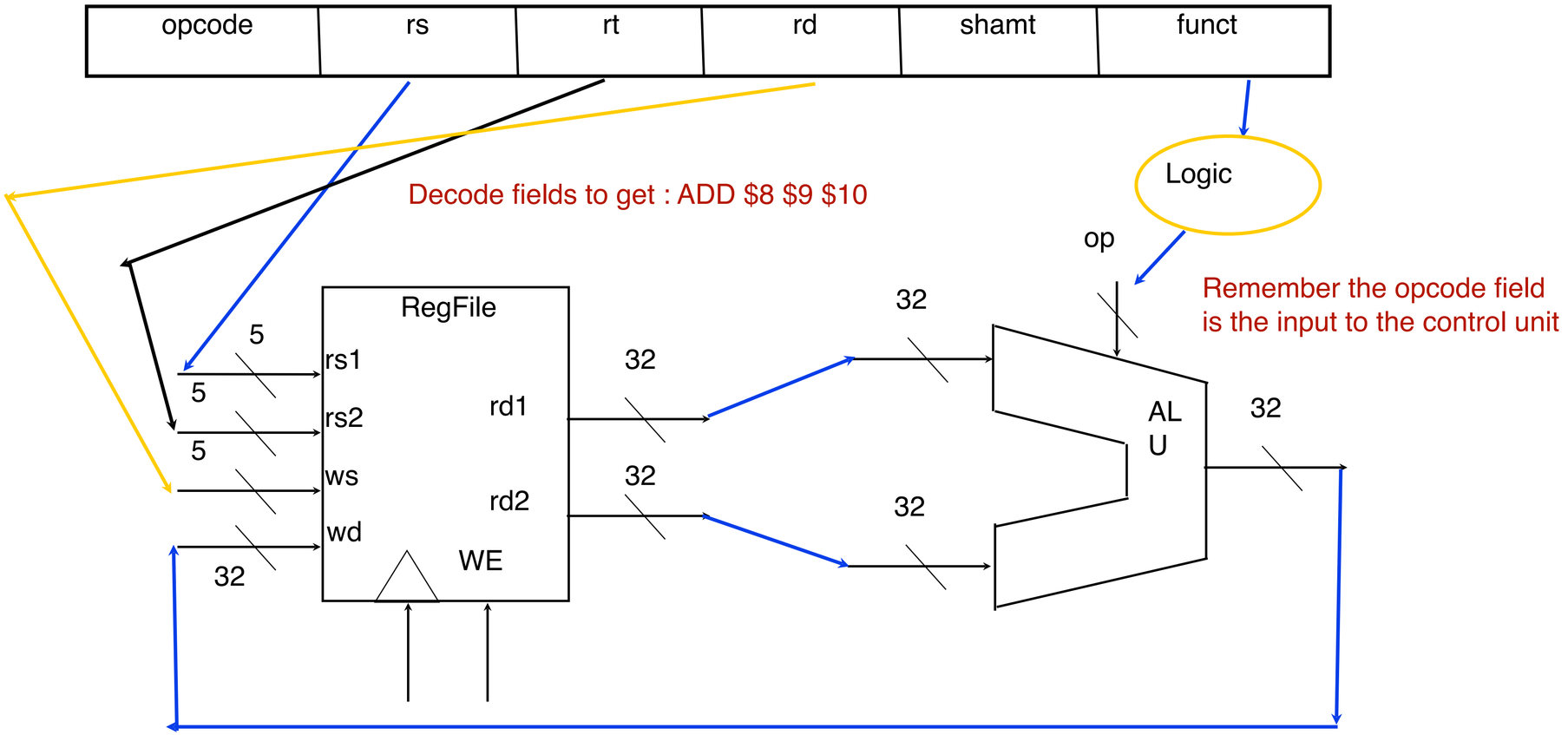
# Datapath Elements: Decoding Instructions



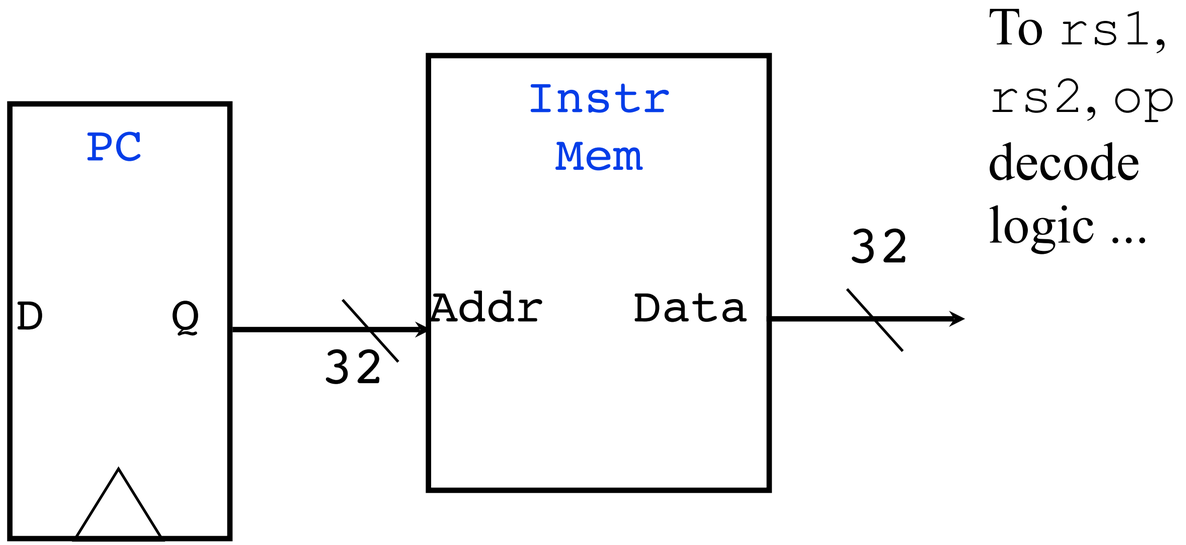
# Datapath Elements: Executing Instructions



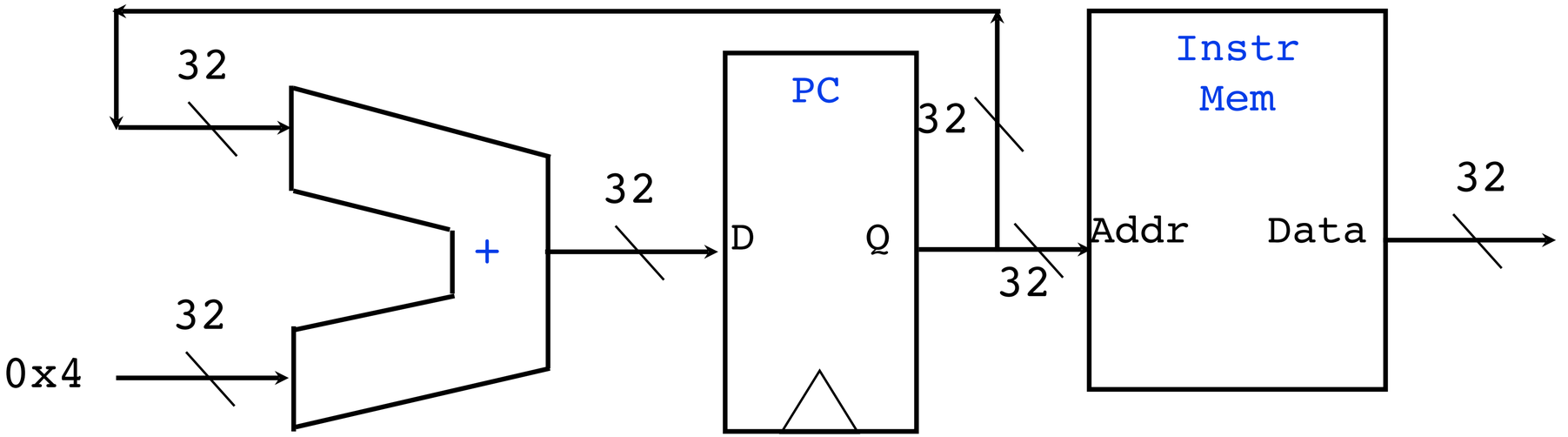
# Datapath Elements: Executing Instructions



# Datapath For Instruction Fetch

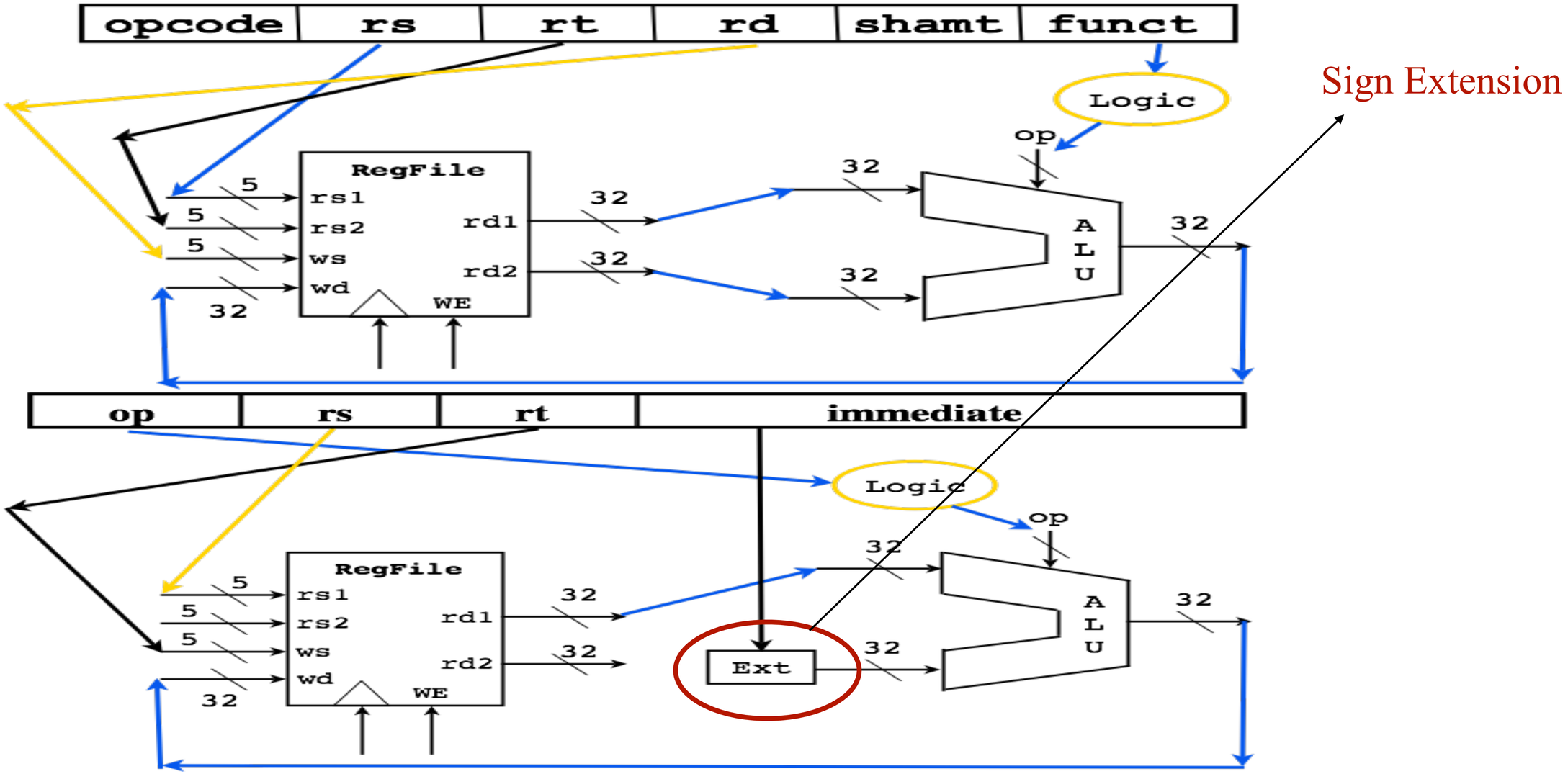


# Datapath For Instruction Fetch

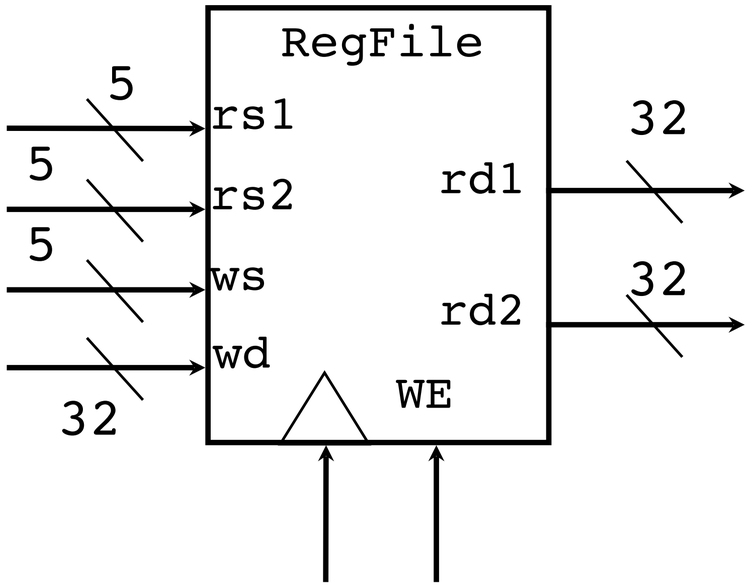


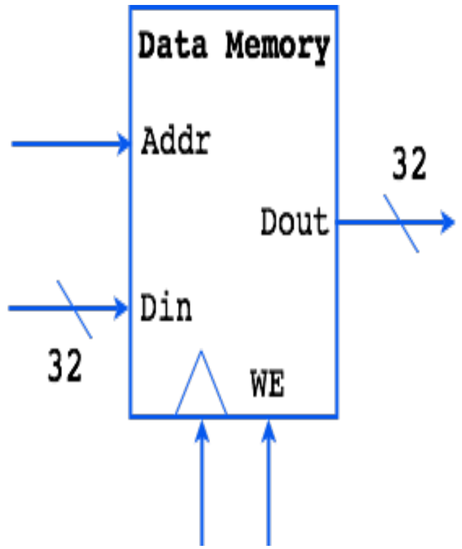

To `rs1`, `rs2`, `op` decode logic ...

# What about I format?



# Loads from Memory

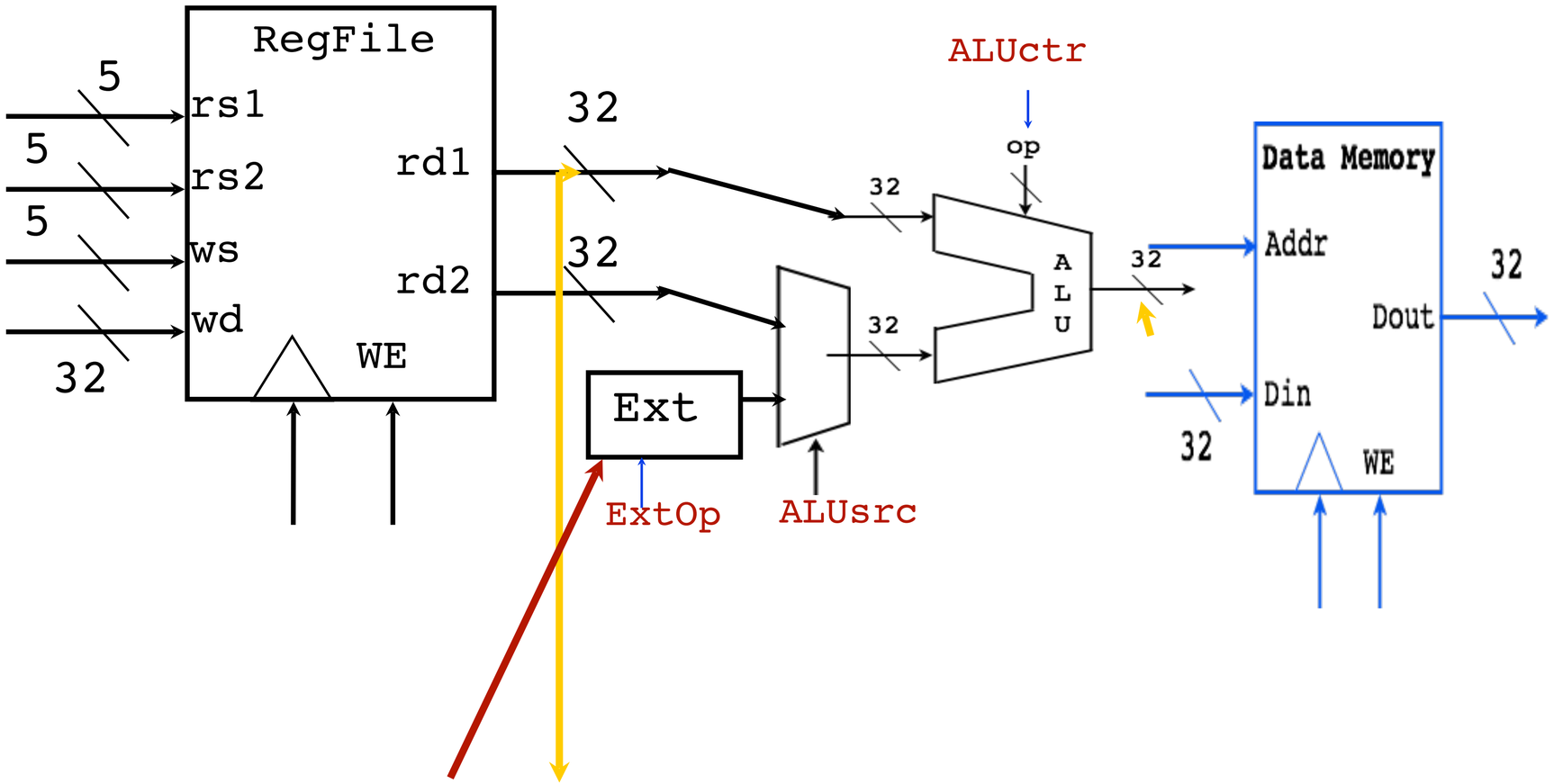



Syntax: LW $1, 32($2)

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

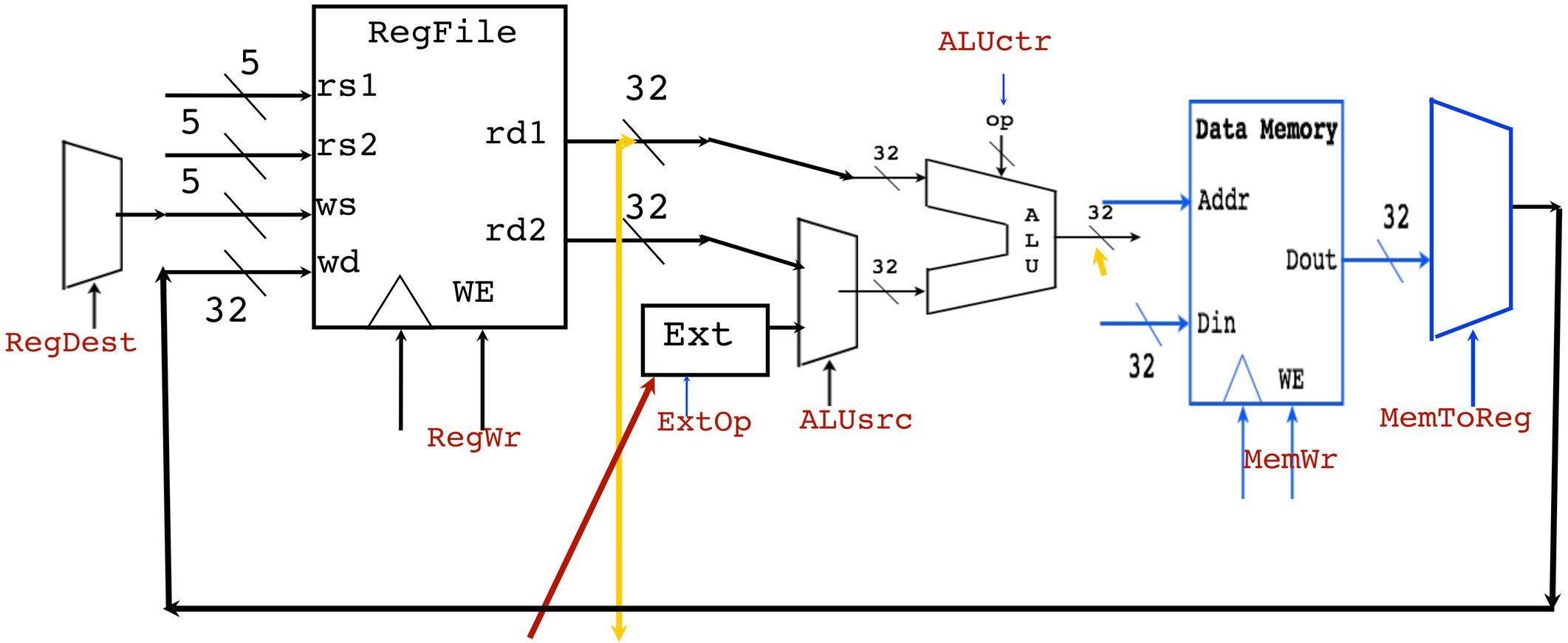
# Loads from Memory




Syntax: LW $1, 32($2)

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

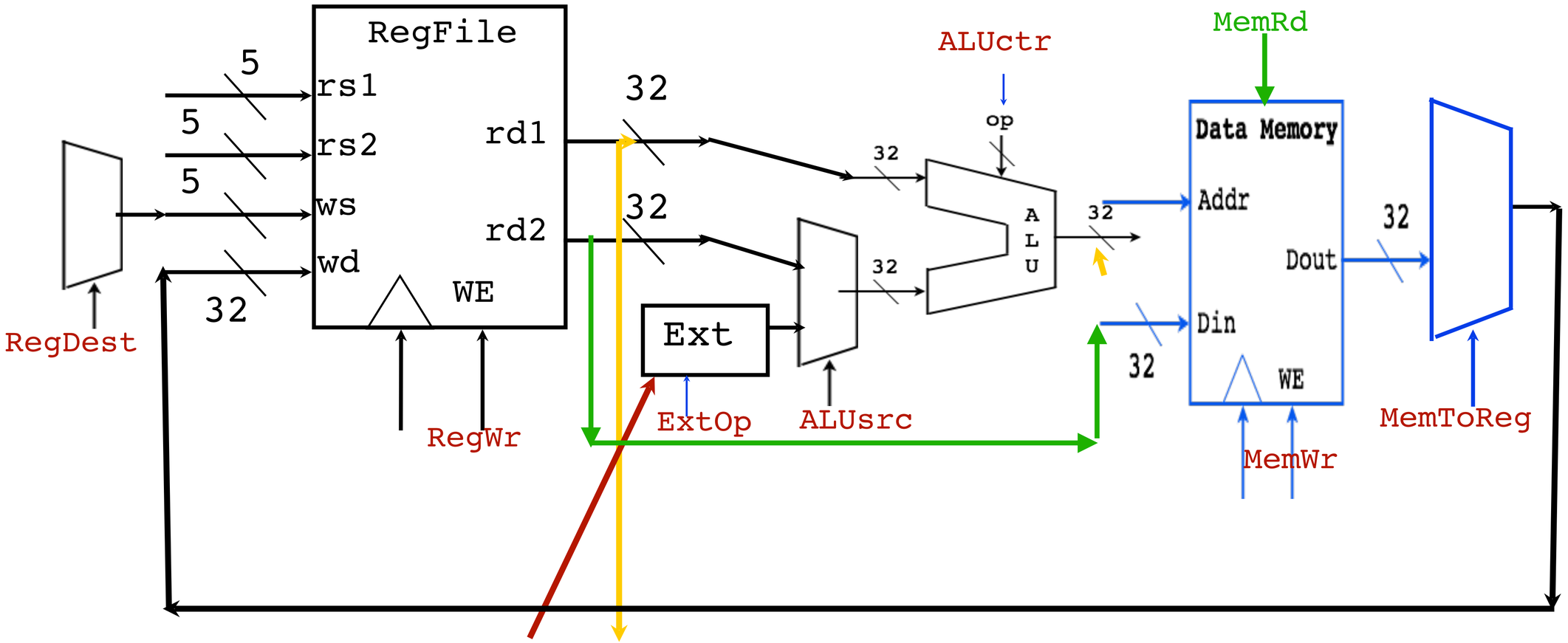
# Loads from Memory




Syntax: LW $1, 32($2)

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

# Stores to Memory (with Load Datapath)

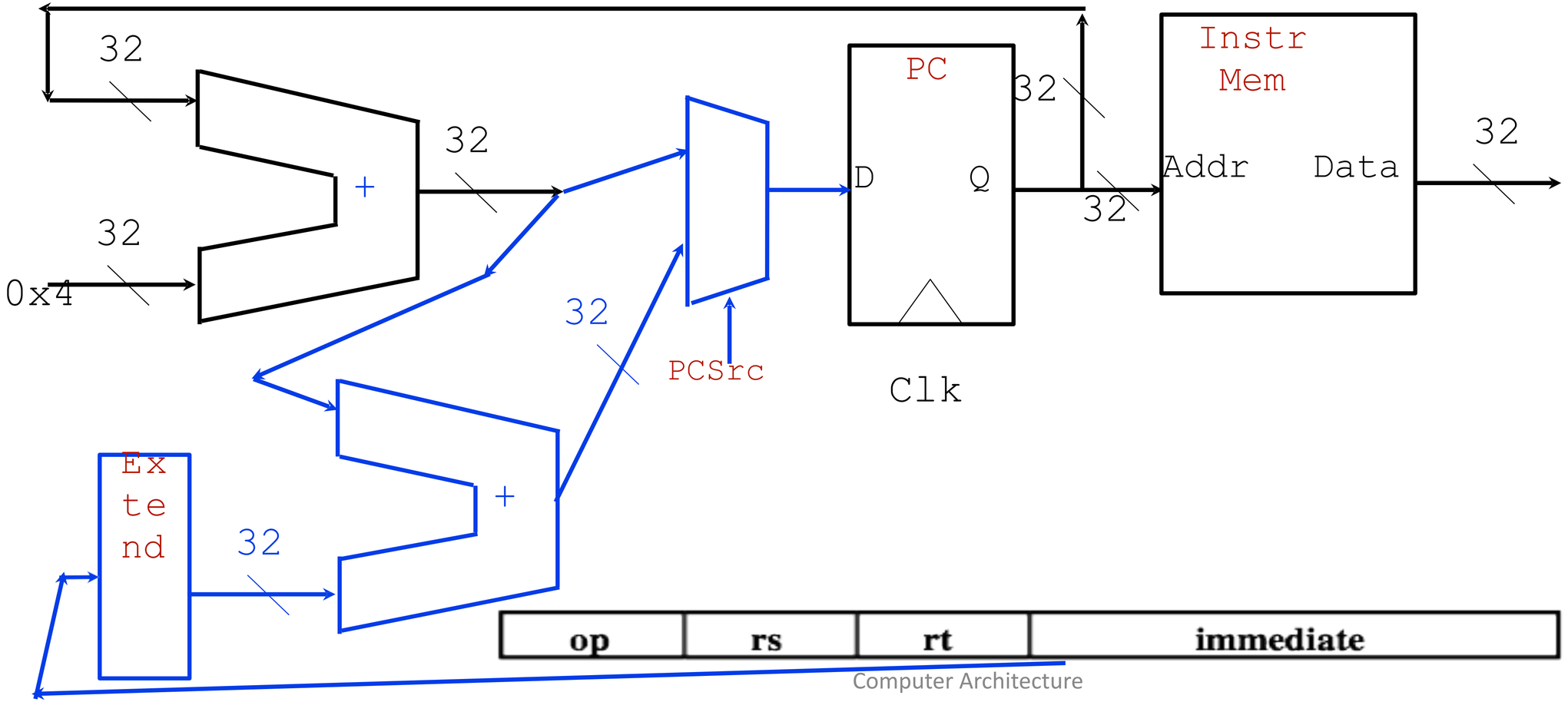



Syntax: SW $1, 32($2)

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

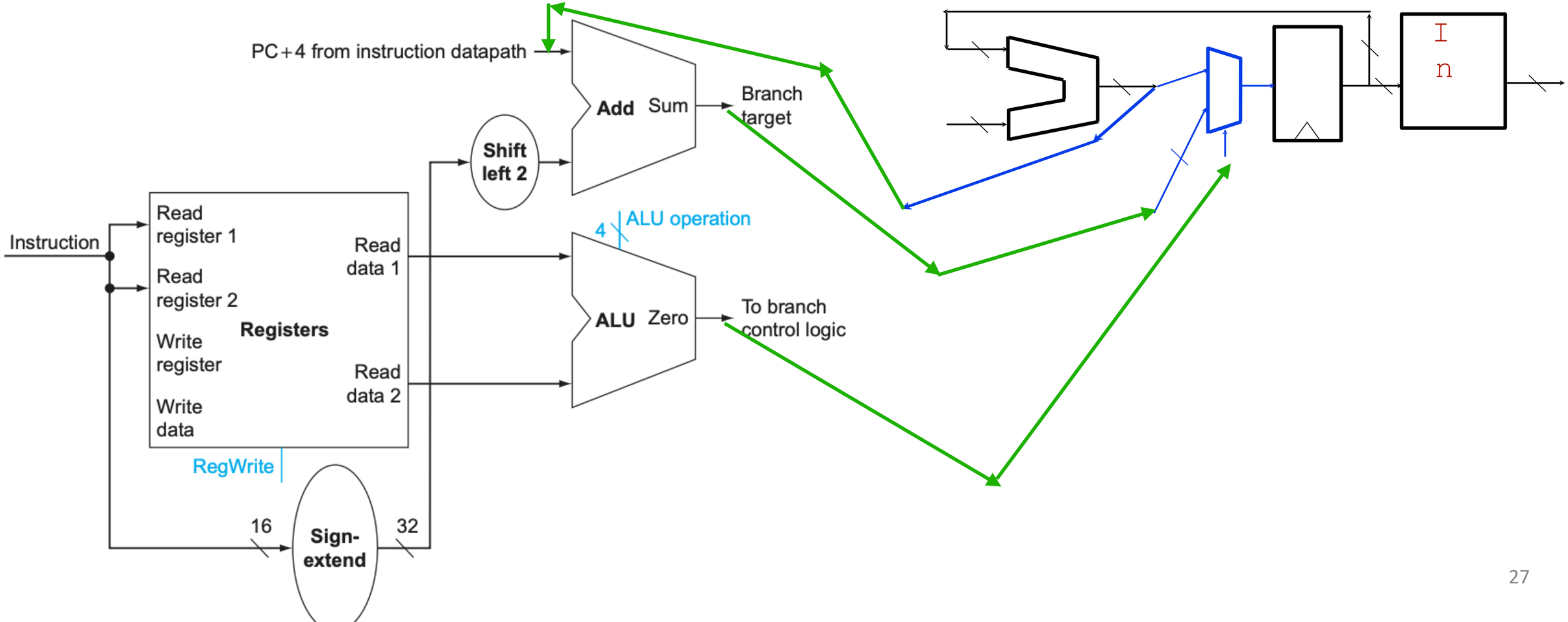
# Branch Instructions

```
Syntax: BEQ $1, $2, 12
Action: If ($1 != $2), PC = PC + 4
Action: If ($1 == $2), PC = PC + 4 + 48
```

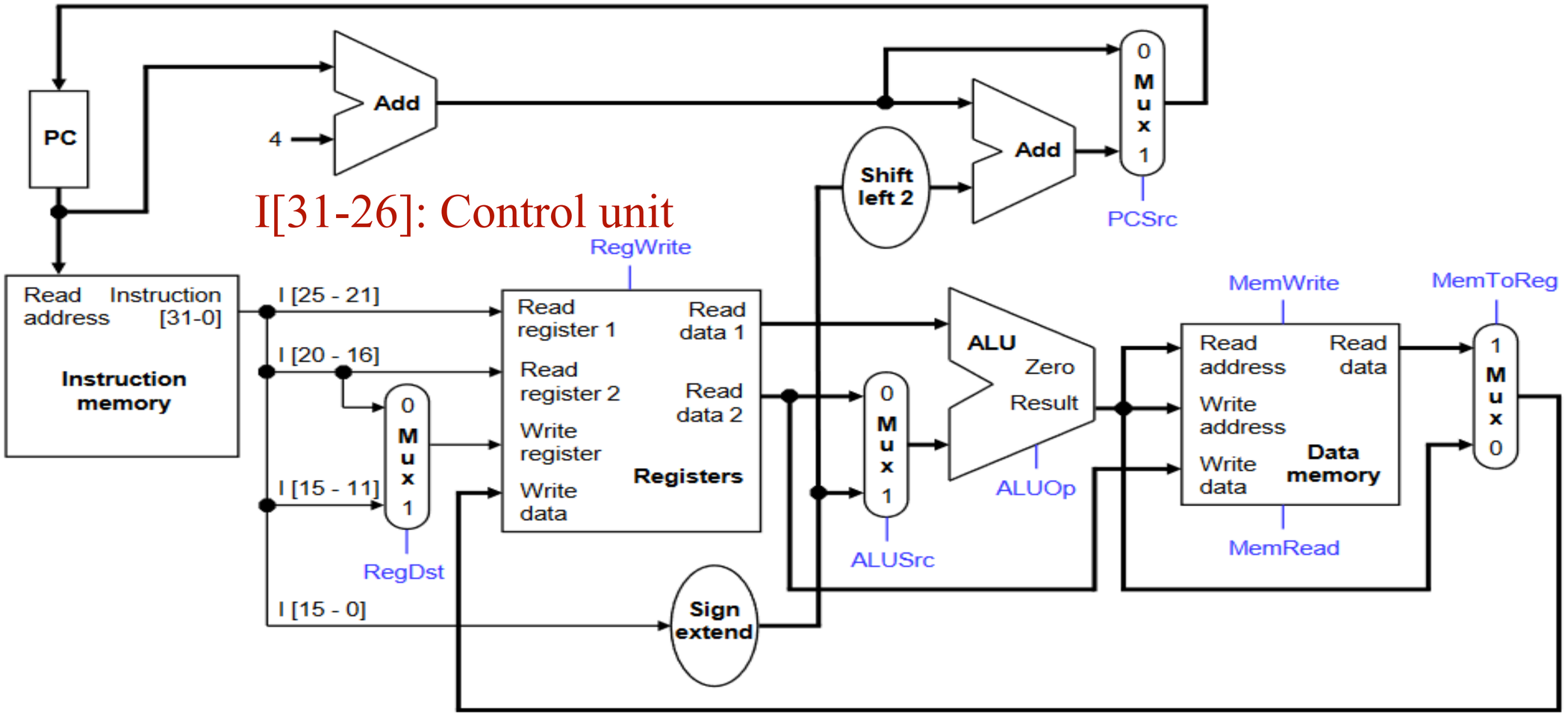


# Branch Instructions

```
Syntax: BEQ $1, $2, 12
Action: If ($1 != $2), PC = PC + 4
Action: If ($1 == $2), PC = PC + 4 + 48
```

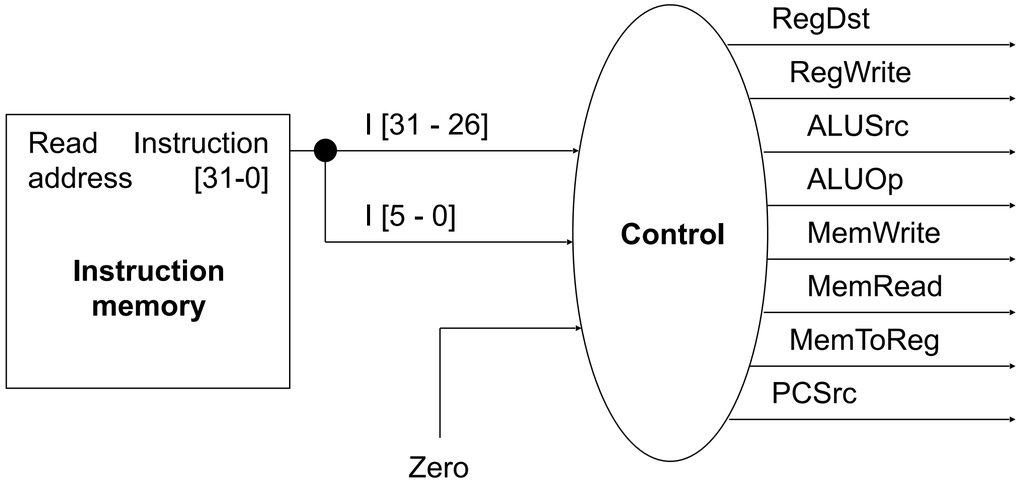


# The Complete Picture



# Control Signals So far

- MemRead
- MemWrite
- RegWrite
- MemtoReg
- RegDst
- ALUop, ALUSrc
- PCSrc



# In Detail

- MemRead: Read from memory when assert
- MemWrite: Write into the memory when assert
- RegWrite: Reg. on Write register updated with the input, on assert
- MemtoReg: On assert, memory to register, on deassert, ALU to register
- RegDst: On assert, use rd field, on deassert use rt field
- ALUSrc: On assert, lower 16 bits of an inst., on deassert from the second register
- PCSrc: On assert, branch target, deassert, PC+4

# Control Signal Table

| Operation | RegDst | RegWrite | ALUSrc | ALUOp | MemWrite | MemRead | MemToReg |
|---|---|---|---|---|---|---|---|
| add | 1 | 1 | 0 | 010 | 0 | 0 | 0 |
| sub | 1 | 1 | 0 | 110 | 0 | 0 | 0 |
| and | 1 | 1 | 0 | 000 | 0 | 0 | 0 |
| or | 1 | 1 | 0 | 001 | 0 | 0 | 0 |
| slt | 1 | 1 | 0 | 111 | 0 | 0 | 0 |
| lw | 0 | 1 | 1 | 010 | 0 | 1 | 1 |
| sw | X | 0 | 1 | 010 | 1 | 0 | X |
| beq | X | 0 | 0 | 110 | 0 | 0 | X |

# Why not single cycle?

- The longest possible datapath is the clock cycle time.

What does it mean?

# Why not single cycle?

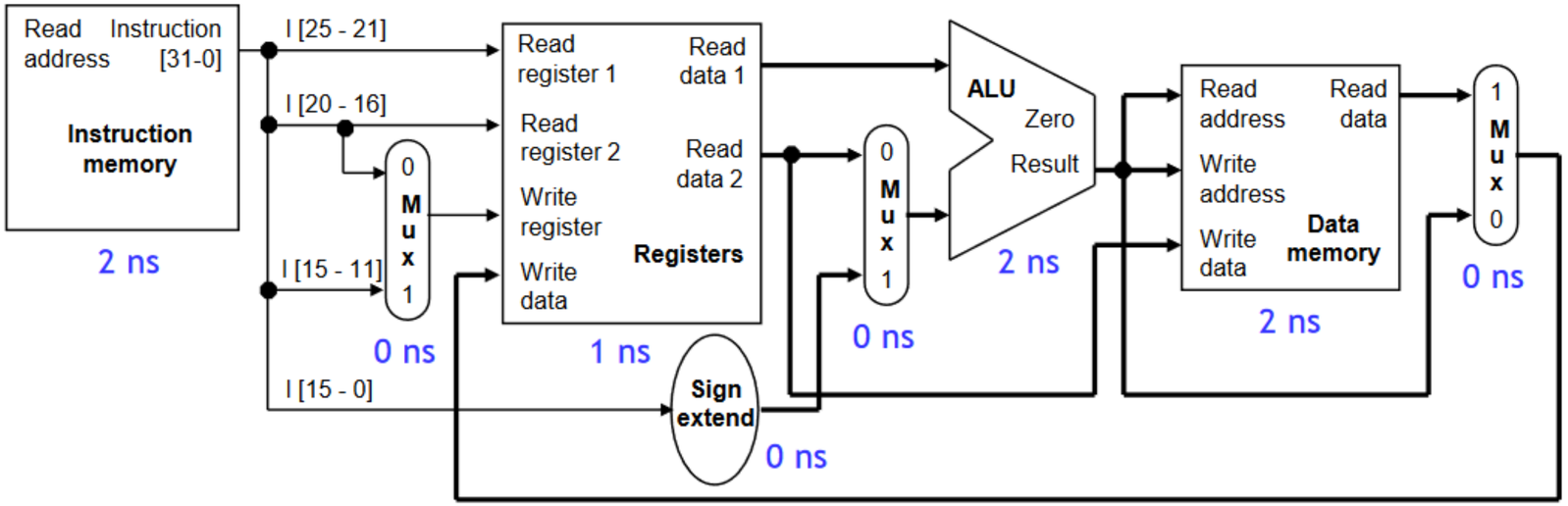
- For example, lw $t0, -4($sp) needs 8ns, assuming the delays shown here.

| | | |
|---|---|---|
| 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 | |

one clock cycle: 8ns

Processor frequency: 125MHz

Cycle per Instruction (CPI): 1




An add instruction: no need of 8ns

# Why not single cycle?

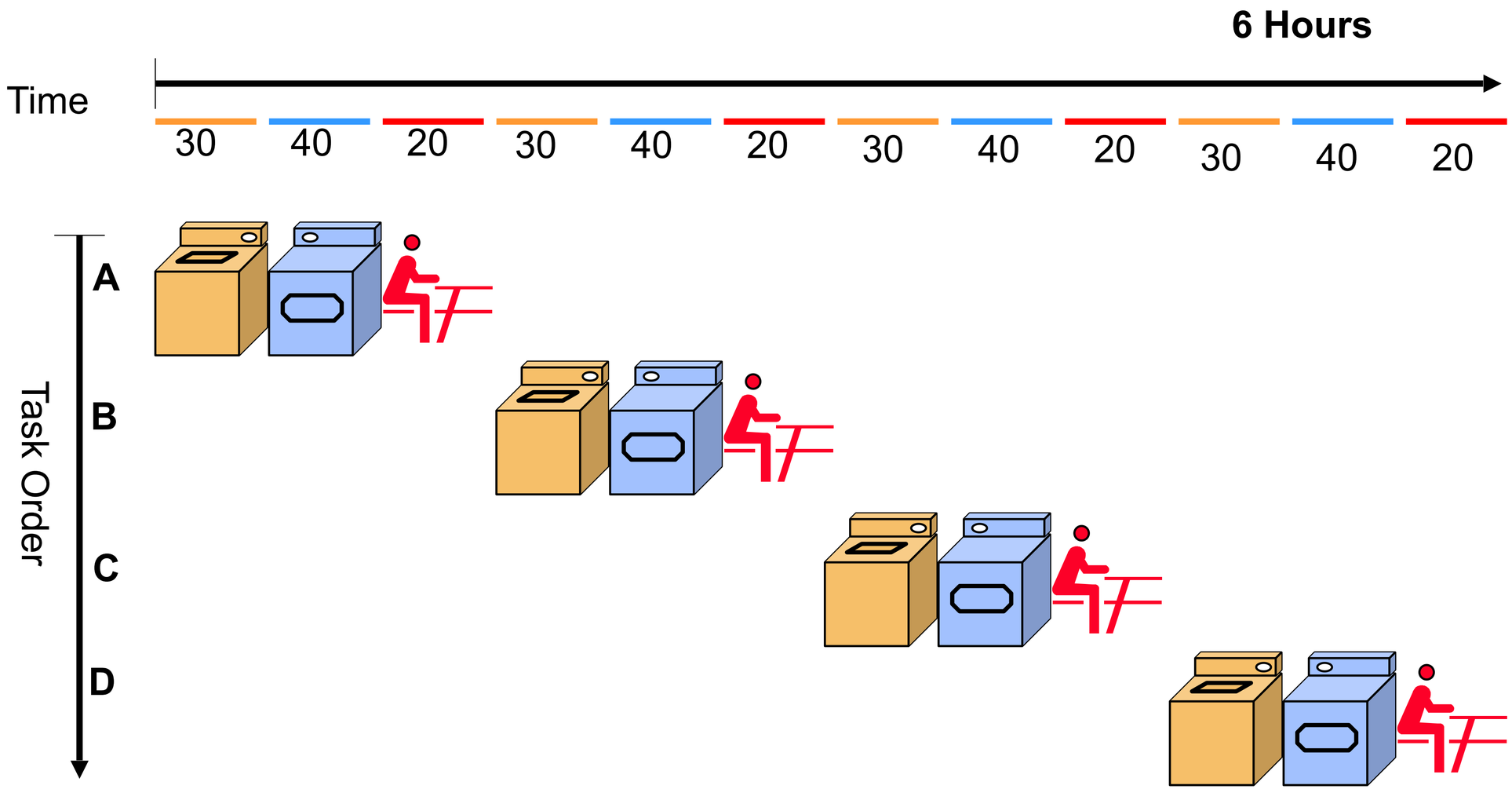
- The longest possible datapath is the clock cycle time.

Violating *common case fast* — Basic principle of computer architecture.

# Now what should we do??

# Why not single cycle?

**A, B, C, D wants to wash cloths**



# Single Vs. Multi-cycle: Where is the Gain?

Single cycle (Worst case)

Everyone takes 90 mins  Full package = 90 minutes

Multi cycle (average case kinda)

One person: 20 to 90 mins  avg = 53.33 minutes

# Single to Multi Cycle

- Measuring Performance:

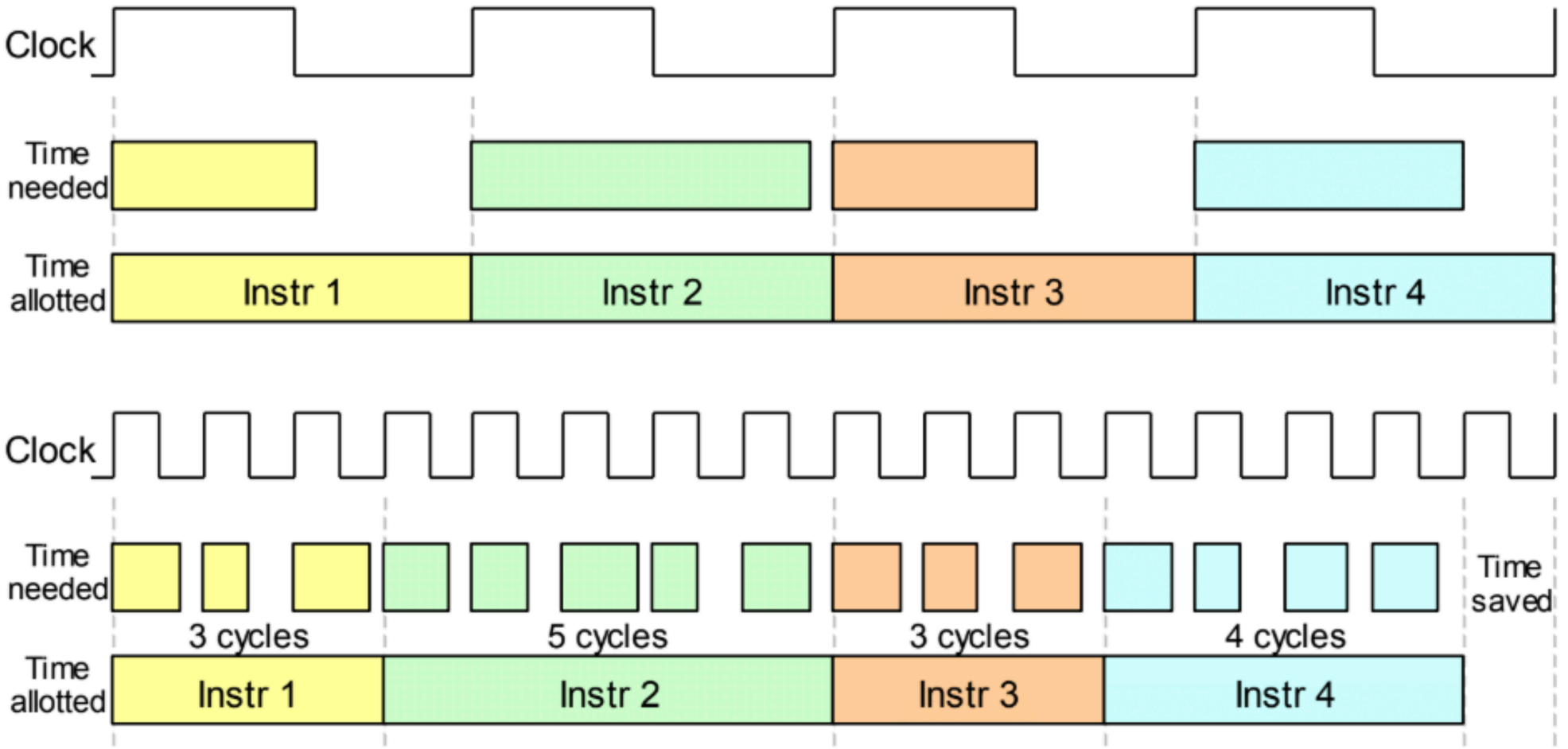
$$Execution\ Time = \#Instructios \times Cycles\ Per\ Instruction \times Clock\ Cycle\ Time$$

$$Speedup = \frac{Execution\ Time_{old}}{Execution\ Time_{new}}$$

  - Cycles Per Instruction (CPI) = 1 for single cycle processor
    - Very good, can't be better in simple scenarios
    - But still it's problematic — Why?
      - The clock cycle time is bounded by the length of the critical path of the datapath — here it is the load instruction
      - In simple words, the overall computation time will be poor!!!

  - How can we improve?
    - Multi-cycle datapath: Improve clock frequency — but poor CPI — not a great option

# Single to Multi Cycle

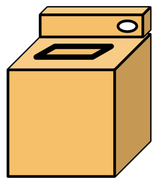


# Can We Have Both?

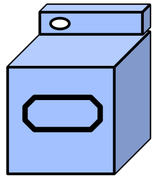
Faster clock rate and also CPI=1?

# Basic Intuition

**A, B, C, D wants to wash cloths**

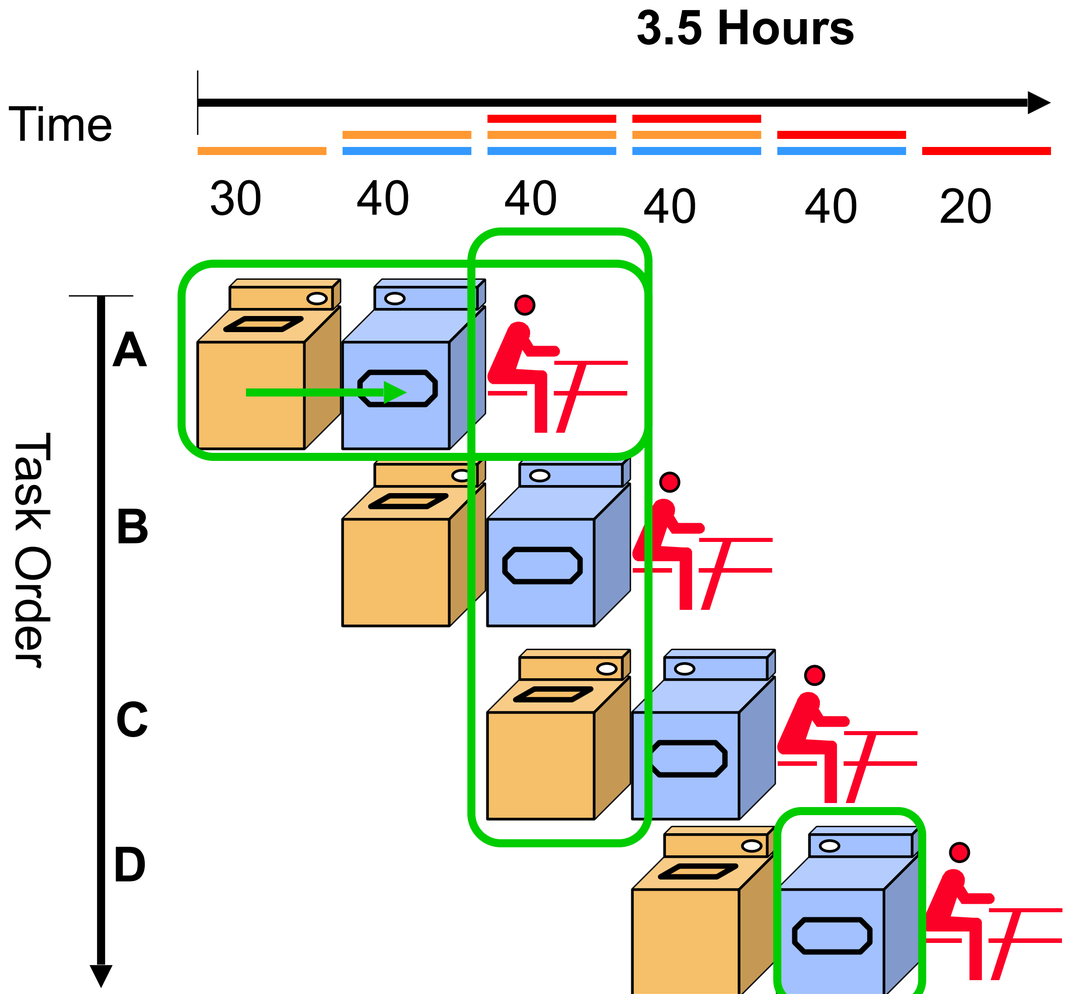


washer
30 mins



dryer
40 mins

folder
20 mins




Main Observations

- Different computation stages which can have overlapped tasks
  - This is the point of speedup!!!
- The timing of each computation stage is determined by the most time consuming task
- **This is called pipelining**
- **Tricky**:
  - Each person still needs 90 mins.
  - But considering all A, B, C, D, the average execution time improves a lot.

# Latency and Bandwidth (throughput)

- Latency
  - time it takes to complete one instance

- Throughput
  - number of computations done per unit time

**Following slides are adapted (with modifications) from Biswa's slides**

# Let’s Summarize

Single cycle:  CPI: 1 , Cycle time: long

Multi cycle: CPI: >1, Cycle time: short

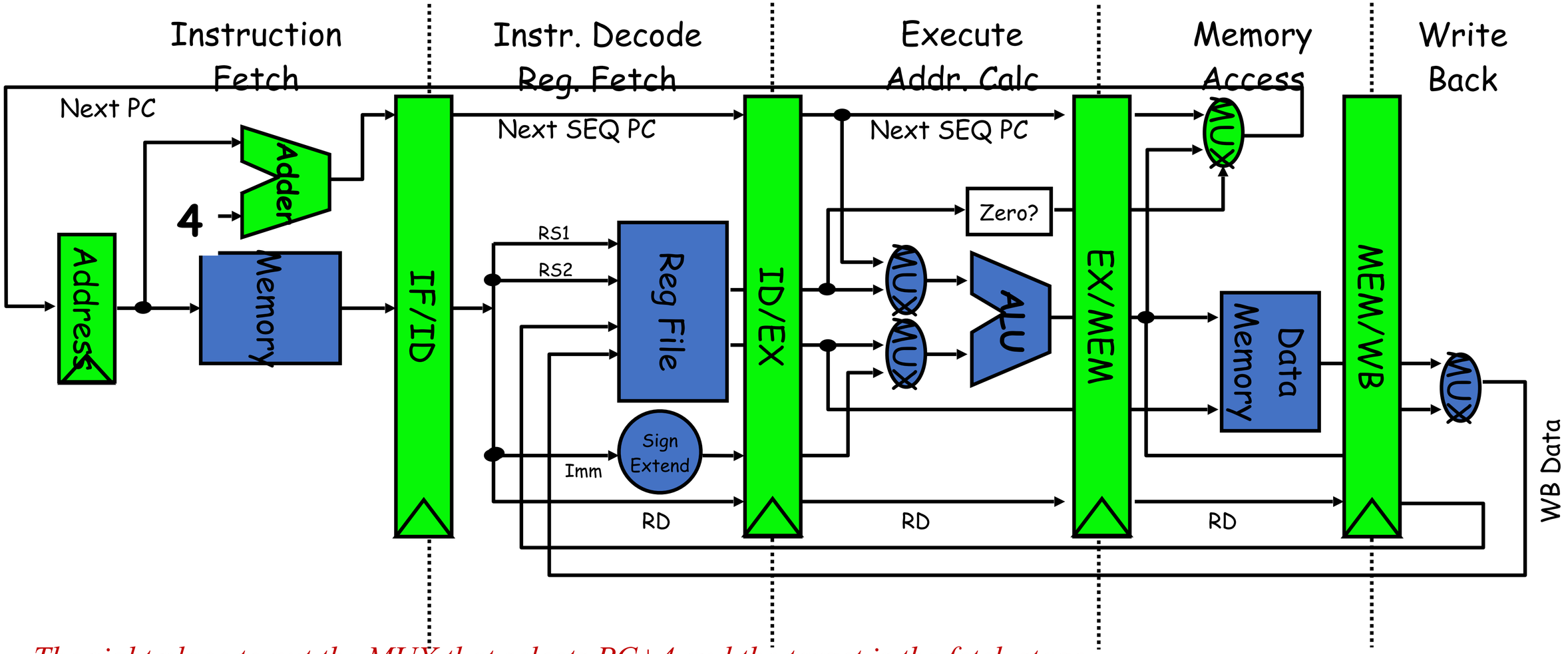
Pipelined: CPI: 1, Cycle time: short  (improves throughput but not latency)

# Pipelining and Richard Feynman

https://www.youtube.com/watch?v=9miKIWIYi4w

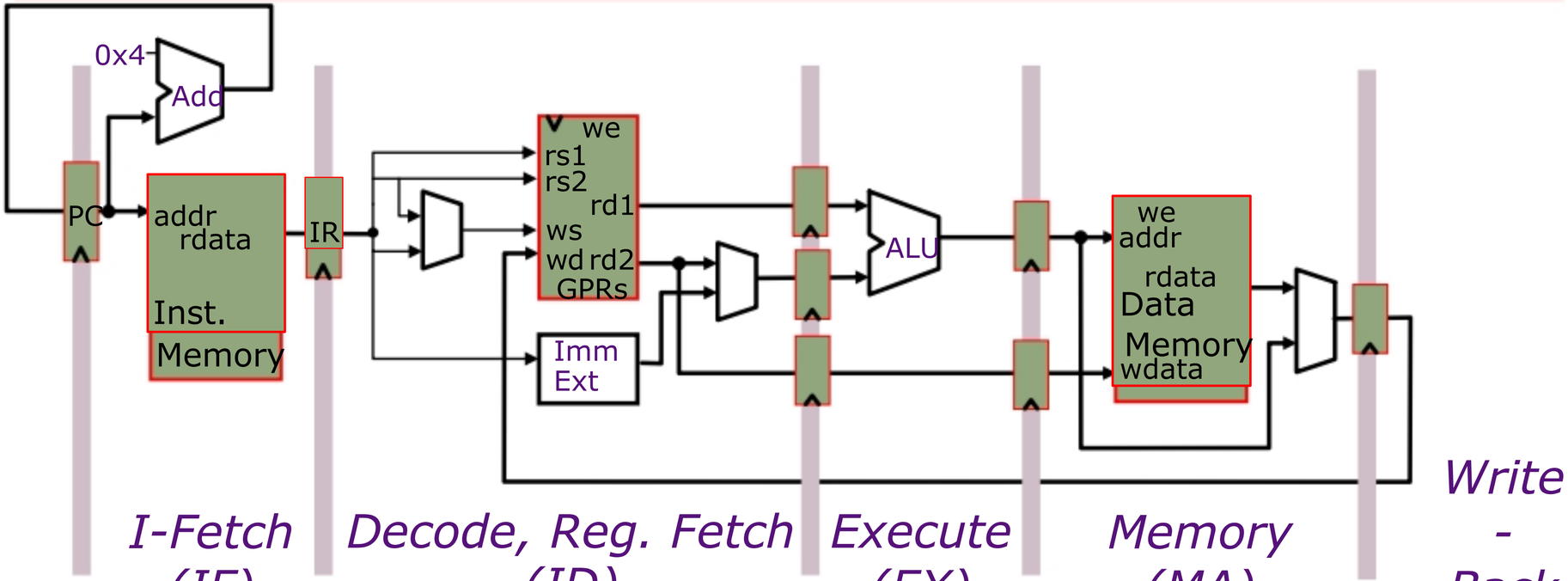
Jump to 1:25

# Vanilla 5-stage pipeline




*The right place to put the MUX that selects PC+4 and the target is the fetch stage.*
*The slide shows a vanilla 5-stage pipeline if we just take a single cycle datapath and divide it into five stages.*

# Resource Utilization

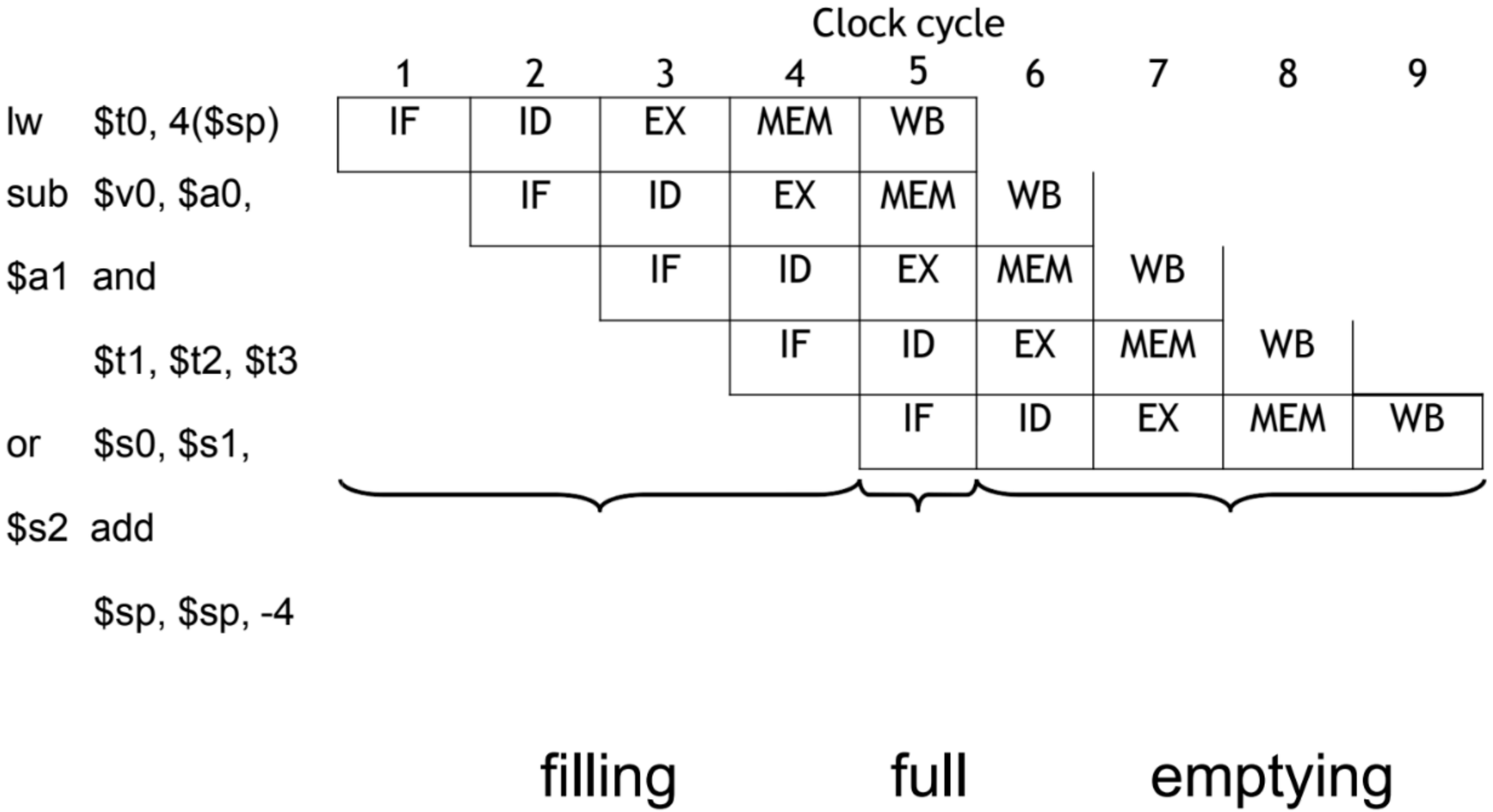



*I-Fetch (IF)* | *Decode, Reg. Fetch (ID)* | *Execute (EX)* | *Memory (MA)* | *Write - Back (WB)*

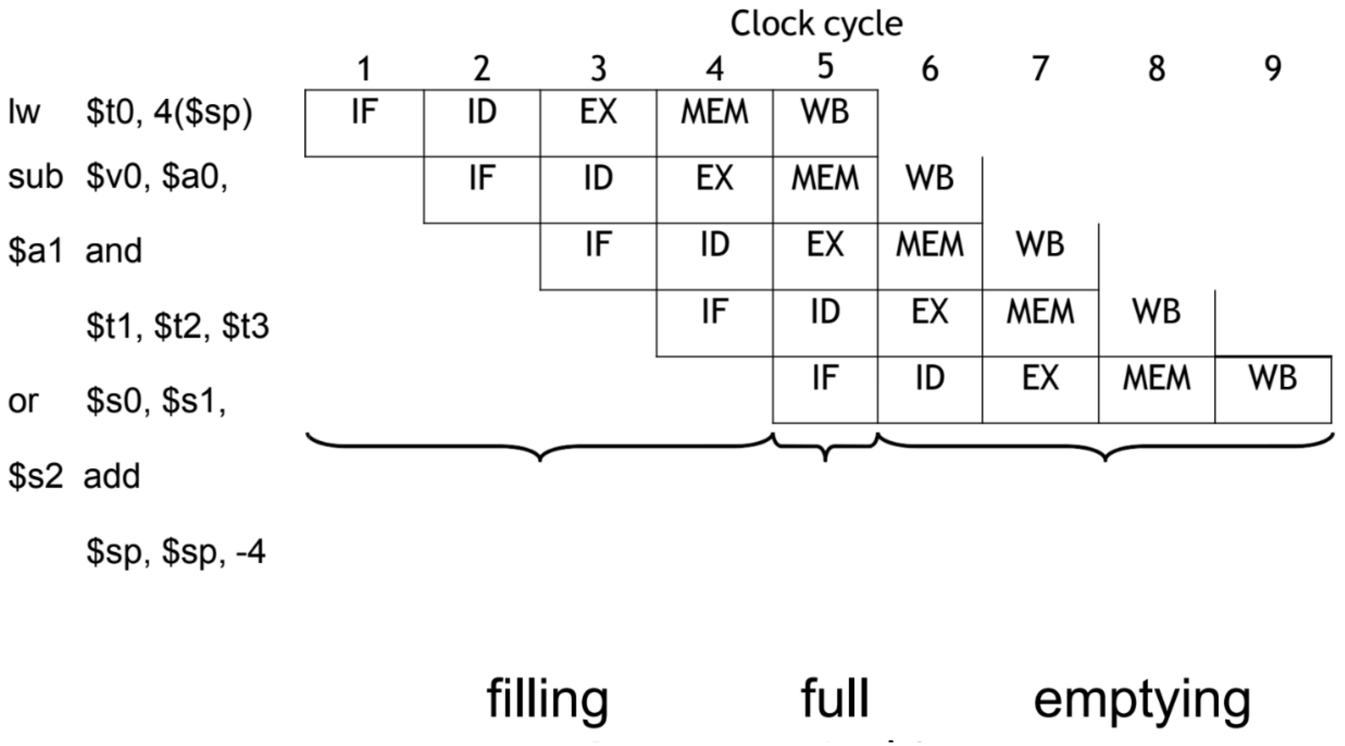
*Resources*

| *time* | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | . . . . |
|---|---|---|---|---|---|---|---|---|---|
| *IF* | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | | | | |
| *ID* | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | | | |
| *EX* | | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | | |
| *MA* | | | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | |
| *WB* | | | | | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ |

# Visualizing Pipeline



# Visualizing Pipeline: Execution Time




For a k-stage pipeline executing N instructions

first instruction: K cycles

Next N-1 instructions: N-1 cycles, total = K + (N-1) cycles

# Pipelined versus Single cycle CPU design

| Instruction | Ifetch | Decode | Execute | Memory | Writeback | Total time |
|---|---|---|---|---|---|---|
| LOAD | 200ns | 100 | 200 | 200 | 100 | 800ns |
| STORE | 200 | 100 | 200 | 200 | | 700ns |
| ADD | 200 | 100 | 200 | | 100 | 600ns |
| BRANCH | 200 | 100 | 200 | | | 500ns |

Total latency in single cycle CPU: 3200 ns

Total latency in pipelined CPU (200ns clock cycle):
1000ns (1st instruction) + 3 X 200 ns (for next three) = 1600 ns

# What's the big deal

Speedup = 3200ns/1600ns = 2X

What if we have a billion instructions?

Single cycle = 1 billion * 800ns = 800 seconds

Pipelined = 1000ns + (1 billion -1) * 200ns ~ 200 seconds

Speedup = 4X ☺

# Let’s include latch latency too

Inter-stage latch = 10ns

New clock cycle time in the pipelined design = 210ns

First instruction will get completed by 1040ns (five stages X 200 ns + four inter-stage latches * 10ns)

New Speedup = 800s/210s ~ 3.8X

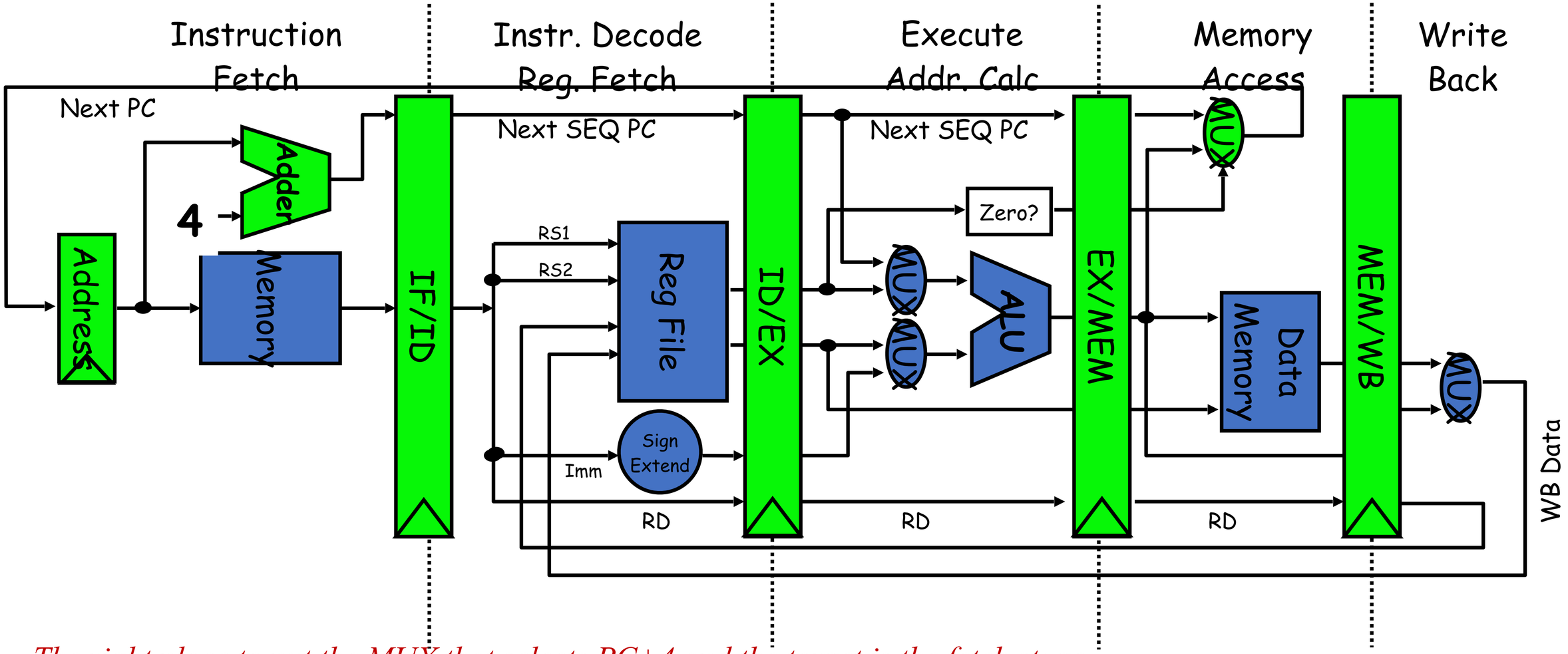
# How to Divide the Datapath?

Suppose memory is significantly slower than other stages. For example, suppose

$t_{IM}$ = 10 units
$t_{DM}$ = 10 units
$t_{ALU}$ = 5 units
$t_{RF}$ = 1 unit
$t_{RW}$ = 1 unit

Since the slowest stage determines the clock, it may be possible to combine some stages without any loss of performance

# Vanilla 5-stage pipeline

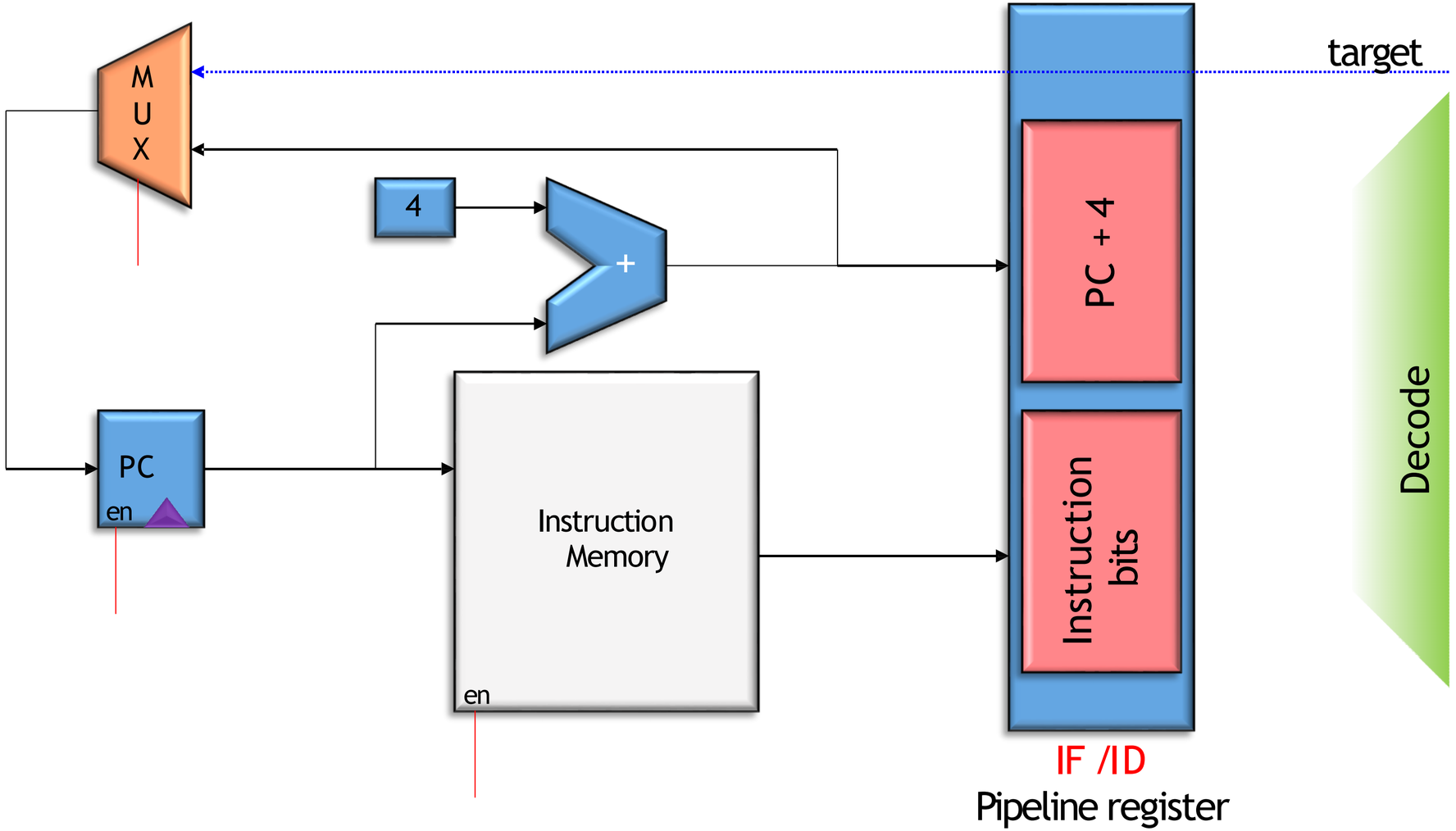



*The right place to put the MUX that selects PC+4 and the target is the fetch stage.*
*The slide shows a vanilla 5-stage pipeline if we just take a single cycle datapath and divide it into five stages.*

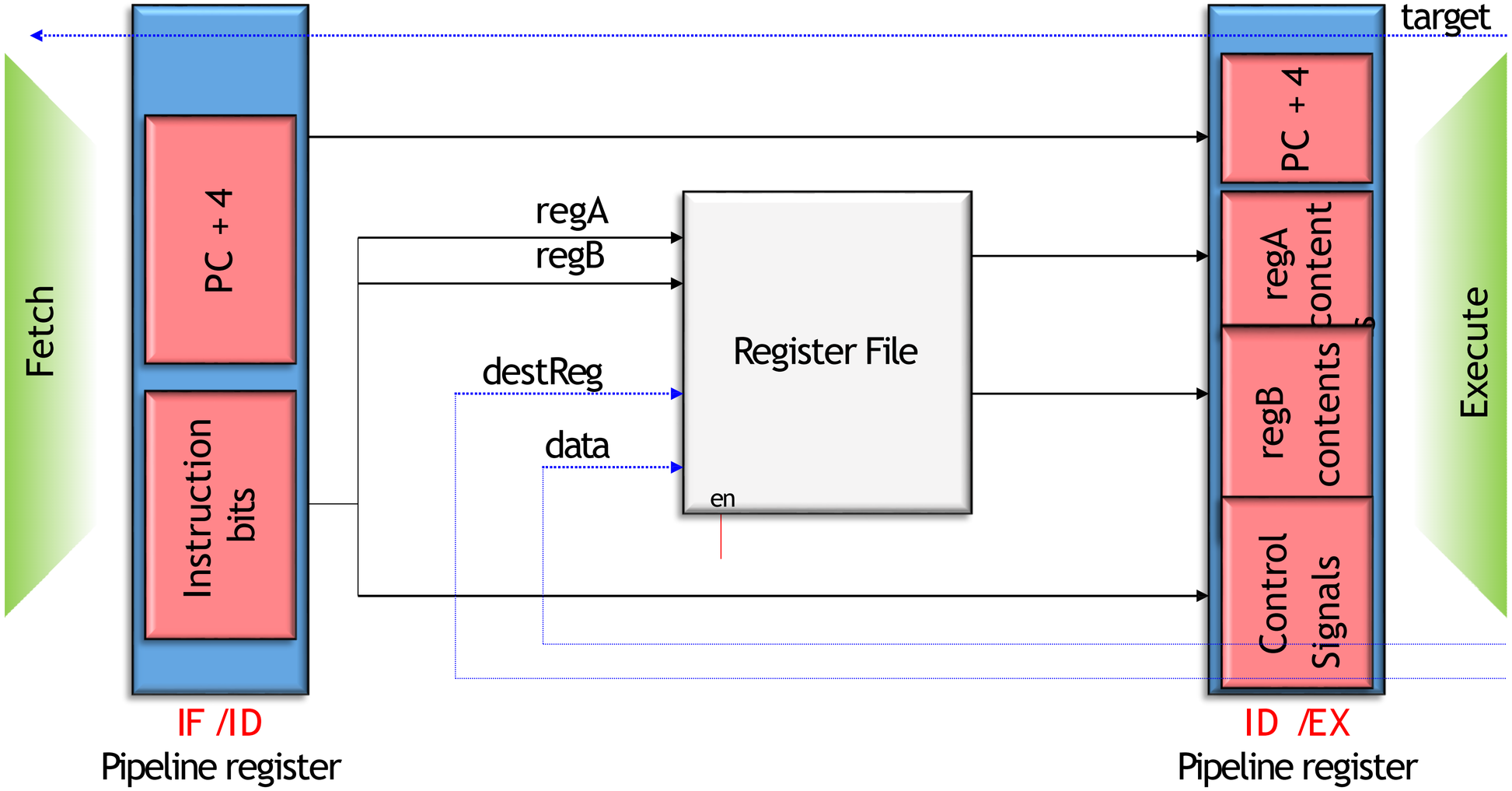
# #Stages and Speedup

| Assumptions | Unpipelined $t_C$ | Pipelined $t_C$ | Speedup |
|---|---|---|---|
| 1. $t_{IM} = t_{DM} = 10$, $t_{ALU} = 5$, $t_{RF} = t_{RW} = 1$, 4-stage pipeline | 27 | 10 | 2.7 |
| 2. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$, 4-stage pipeline | 25 | 10 | 2.5 |
| 3. $t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW} = 5$, 5-stage pipeline | 25 | 5 | 5.0 |

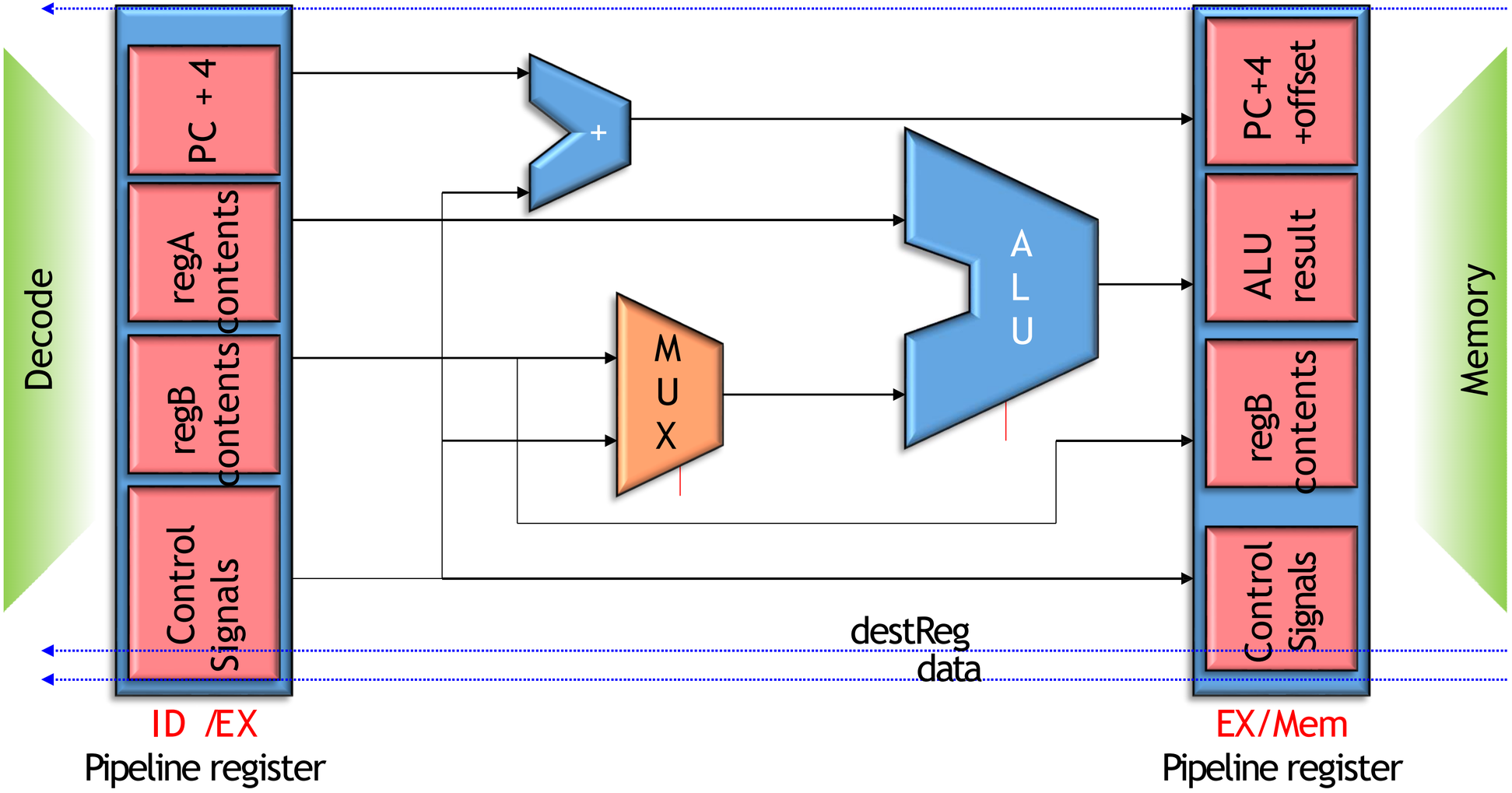
# Stage-1: Fetch



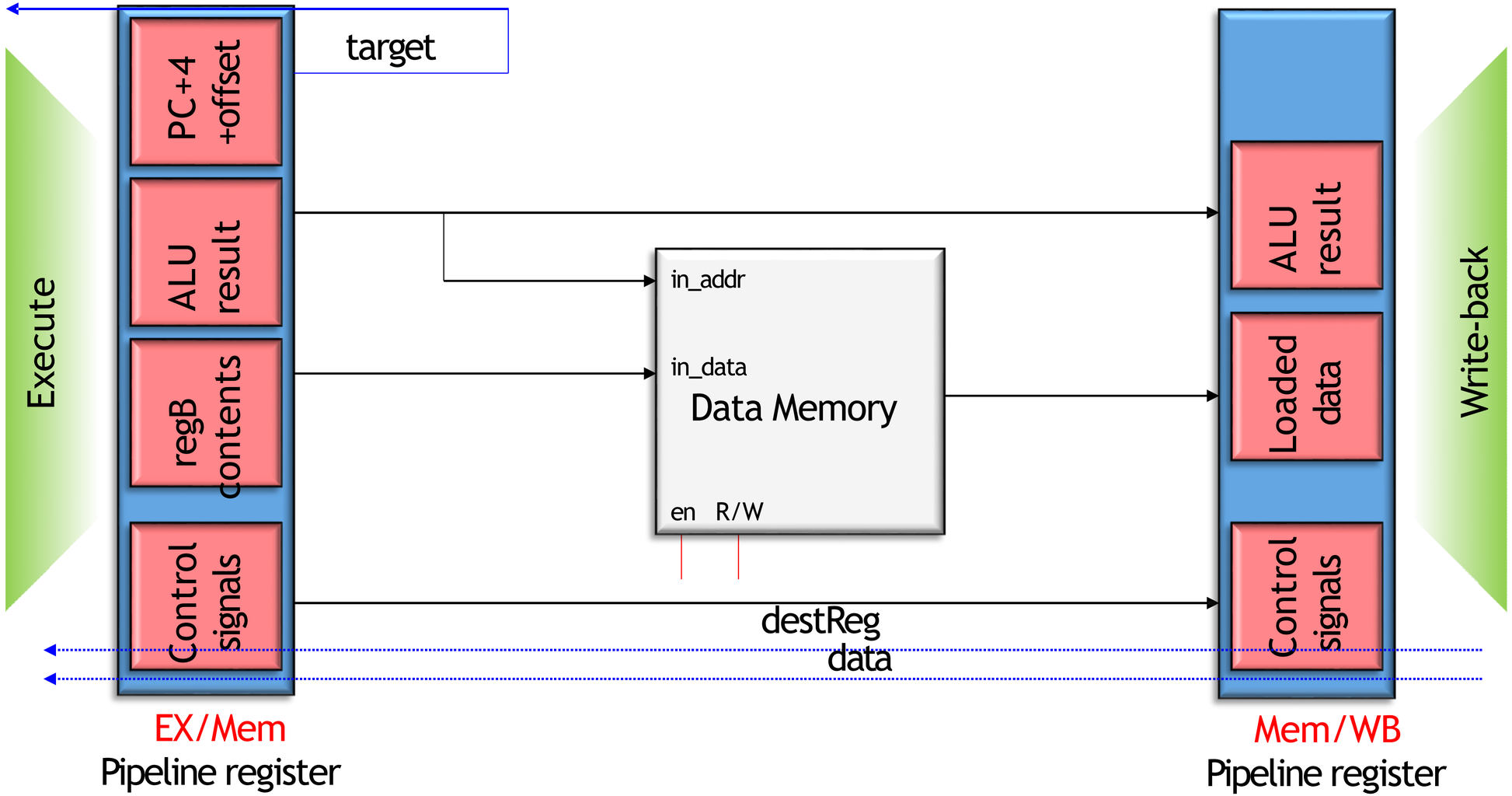
# Stage 2: Decode



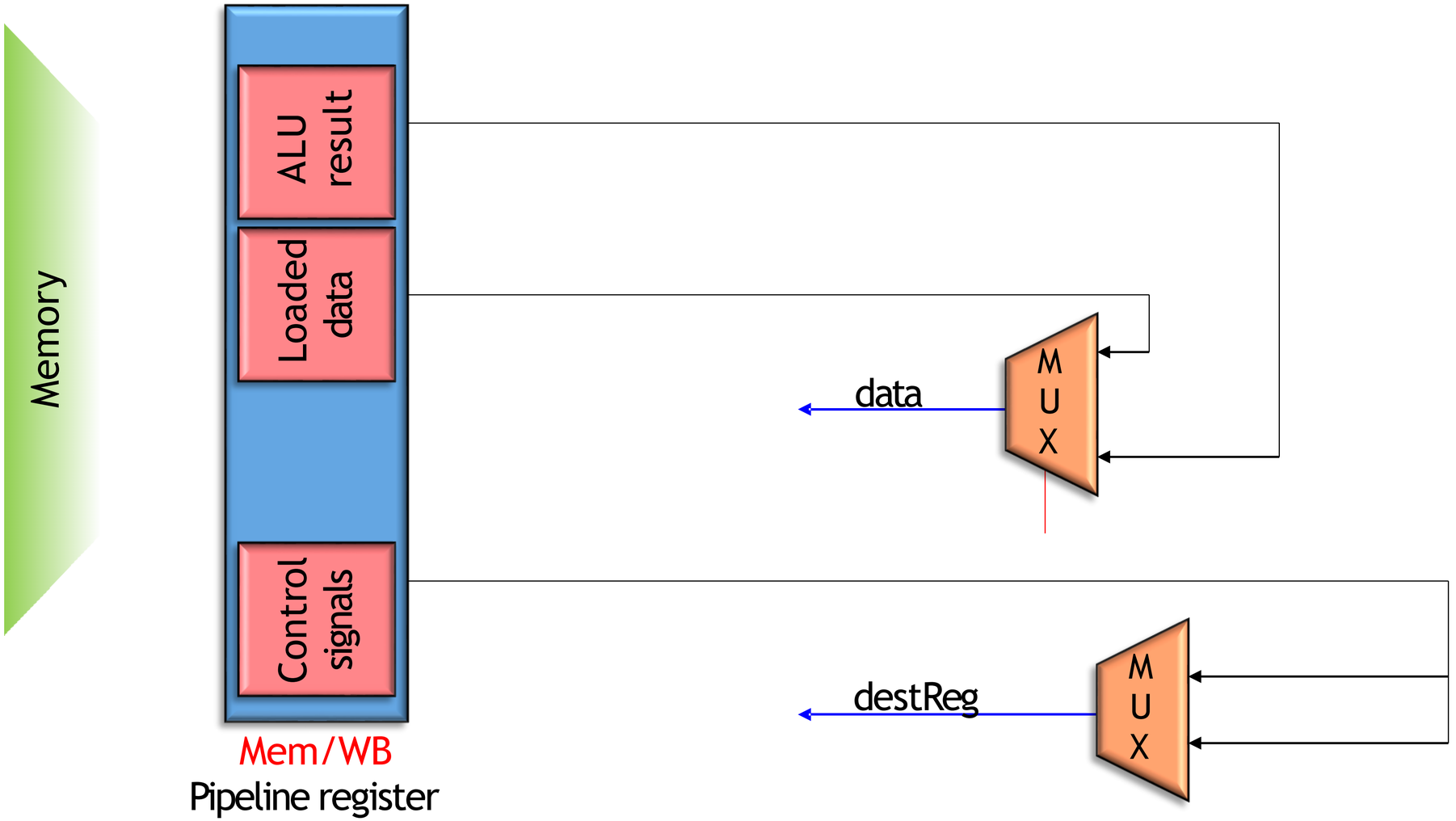
# Stage 3: Execute



# Stage 4: Memory Stage



# Stage 5: Write-back



# The Complete Picture