
EECS 470

Lecture 1

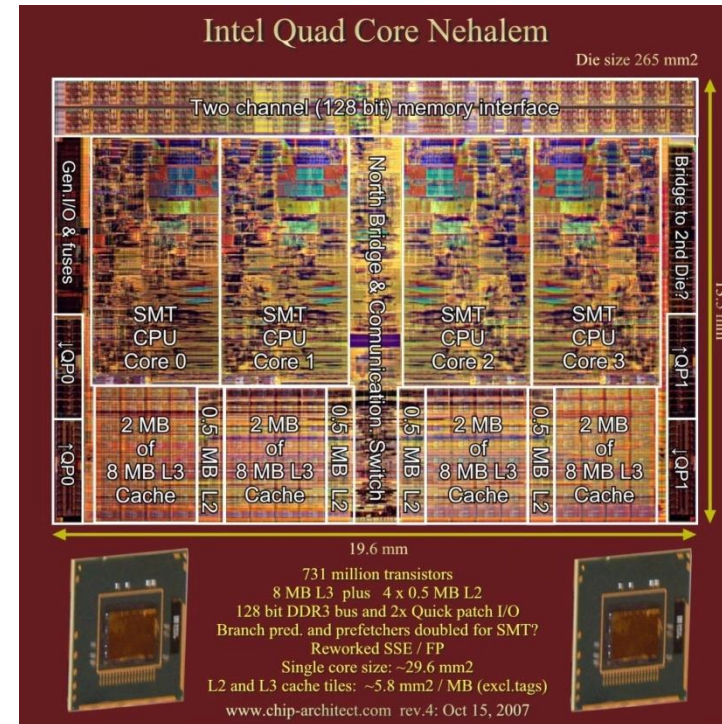
Computer Architecture
Winter 2024



*Slides developed in part by Profs. Brehob, Austin, Falsafi,
Hill, Hoe, Lipasti, Shen, Smith, Sohi, Tyson, Vijaykumar,
Wenisch and probably others...*

What Is Computer Architecture?

“The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct from the organization of the dataflow and controls, the logic design, and the physical implementation.”



Gene Amdahl, IBM Journal of R&D, April 1964

Architecture as used here...

- We use a wider definition
 - The stuff seen by the programmer
 - Instructions, registers, programming model, etc.
 - Micro-architecture
 - How the architecture is implemented.

Teaching Staff

Instructor

- Dr. Mark Brehob, brehob
 - Office hours
 - Mondays 10:30-noon (4632 Beyster)
 - Sunday 5:30-6:45pm (remote/Zoom)
 - I'll also be around after class for 10-15 minutes most days.

GSI/IAs

- Bradley Schulz – GSI
- Ian Wrzesinski – GSI
- Mustafa Miyaziwala – IA

Their office hours will be posted shortly.

Lecture Today

- Class intro (30 minutes)
 - Class goals
 - Your grade
 - Work expected
- Start review
 - Fundamental concepts
 - Pipelines
 - Hazards and Dependencies (time allowing)
 - Performance measurement (as reference only)

Class goals

- Provide a high-level understanding of many of the relevant issues in *modern* computer architecture
 - Dynamic Out-of-order processing
 - Static (compiler based) out-of-order processing
 - Memory hierarchy issues and improvements
 - Multi-processor issues
 - Power and reliability issues
- Provide a *low-level* understanding of the most important parts of modern computer architecture.
 - Details about all the functions a given component will need to perform
 - How difficult certain tasks, such as caching, really are in hardware

**Class/HW
& Reading**

**Project
& Reading**

Communication

- Website: <http://www.eecs.umich.edu/courses/eecs470/>

EECS 470: Computer Architecture

Winter 2024

Under Construction

[Home / announcements](#) | [Course overview](#) | [Staff and hours](#) | [Piazza](#)
[References / Notes / Handouts](#) | [Homework/Labs/Projects](#) | [Exams](#) | [Schedule](#) | [Gradescope](#)

- Piazza
 - piazza.com/umich/fall2021/eecs470
- Gradescope:
 - <https://piazza.com/umich/winter2024/eecs470>
 - Entry Code: **470rules**
- Lab
 - Attend your assigned section
 - You *need* to go!
 - It does meet this week.

Course outline – near term

EECS 470 Winter 2024 schedule—note that it is likely we won't follow this exactly

Day	Date	Topic	Reading	Homework	Verilog & Project
Thursday	1/11	Introduction and Pipeline review	A.1-A.2		
Friday	1/12	<i>Inlab1: Tools Tutorial</i>			
Tuesday	1/16	In-order hazards, dependencies, start of OoO	A.2-A.4		
Thursday	1/18	Dependencies, rename, reorder	3.1, 3.4-3.5	HW1 due	
Friday	1/19	<i>Inlab2: Writing testbenches</i>			
Tuesday	1/23	Tomasulo's Algorithm	3.4-3.5		P1 due

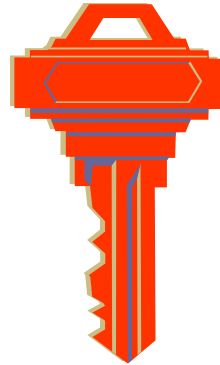
HW1 and P1 will be posted tonight

Grading

- Grade weights are:
 - Midterm – 22%
 - Final Exam – 22%
 - Homework/Quiz – 10%
 - 5 homeworks, 1 quiz, all equal. Drop the lowest grade.
 - Verilog Programming assignments – 8%
 - 3 assignments worth 1%, 2%, and 5%
 - In lab assignments – 3%
 - Project – 35%
 - Open-ended (group) processor design in Verilog
 - Grades can vary quite a bit, so is a distinguisher.

Hours (and hours) of work

- Attend class & lab
 - You *really* need to go to lab!
 - **5 hours/week**
- Read book & handouts – **2-3 hours/week**
- Do homework
 - 5 assignments at ~4 hours each/15 weeks
 - **1-2 hours/week**
- 3 Programming assignments
 - 6, 6, 20 hours each = 32 hours/15 weeks
 - **~2 hours/week**
- Verilog Project
 - ~200 hours/15 weeks = **13 hours/week**
- General studying – **~ 1 hour/week**
- Total:
 - **~25 hours/week, some students claim more...**



"Key" concepts



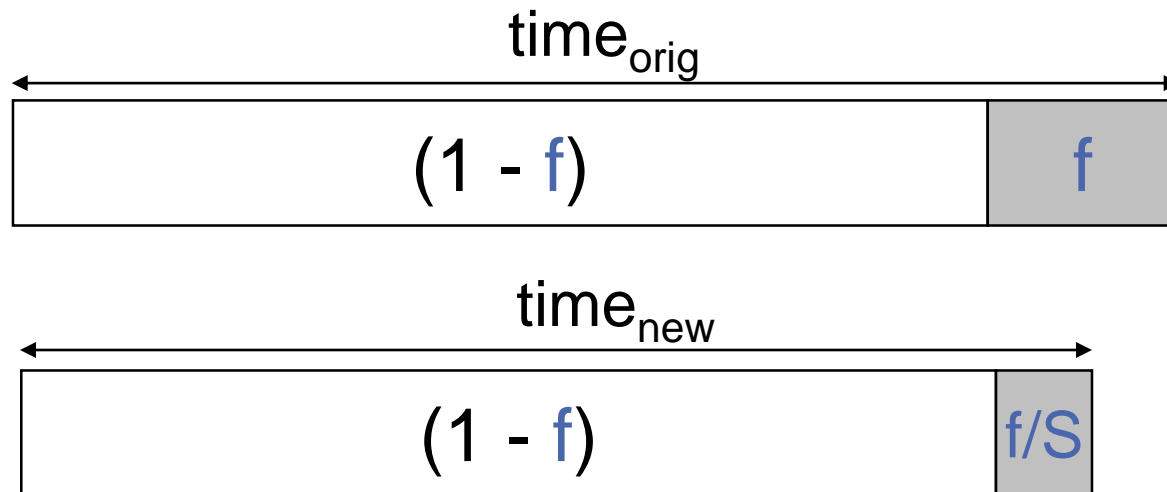
Amdahl's Law

Speedup = $\text{time}_{\text{without enhancement}} / \text{time}_{\text{with enhancement}}$

Suppose an enhancement speeds up a fraction f of a task by a factor of S

$$\text{time}_{\text{new}} = \text{time}_{\text{orig}} \cdot ((1-f) + f/S)$$

$$S_{\text{overall}} = 1 / ((1-f) + f/S)$$





Parallelism: Work and Critical Path

Parallelism - the amount of independent sub-tasks available

Work= T_1 - time to complete a computation on a sequential system

Critical Path= T_∞ - time to complete the same computation on an infinitely-parallel system

Average Parallelism

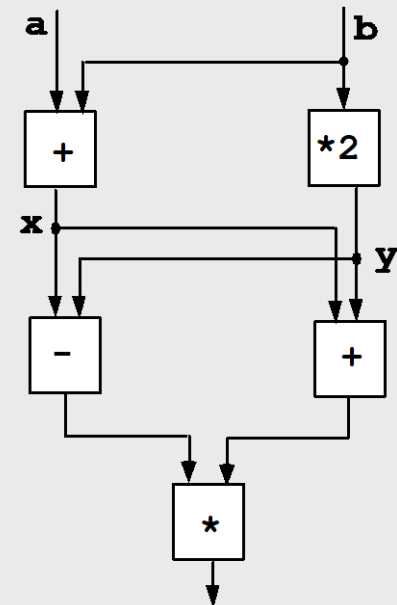
$$P_{\text{avg}} = T_1 / T_\infty$$

For a p wide system

$$T_p \geq \max\{ T_1/p, T_\infty \}$$

$$P_{\text{avg}} \gg p \Rightarrow T_p \approx T_1/p$$

```
x = a + b;  
y = b * 2  
z = (x - y) * (x + y)
```





Locality Principle

One's recent past is a good indication of near future.

- **Temporal Locality:** If you looked something up, it is very likely that you will look it up again soon
- **Spatial Locality:** If you looked something up, it is very likely you will look up something nearby next

Locality == Patterns == Predictability

Converse:

Anti-locality : If you haven't done something for a very long time, it is very likely you won't do it in the near future either



Memoization

Dual of temporal locality but for computation

If something is expensive to compute, you might want to remember the answer for a while, just in case you will need the same answer again

Why does memoization work??

Examples

- Trace caches



Amortization

overhead cost : one-time cost to set something up

per-unit cost : cost for per unit of operation

$$\text{total cost} = \text{overhead} + \text{per-unit cost} \times N$$

It is often okay to have a high overhead cost if the cost can be distributed over a large number of units

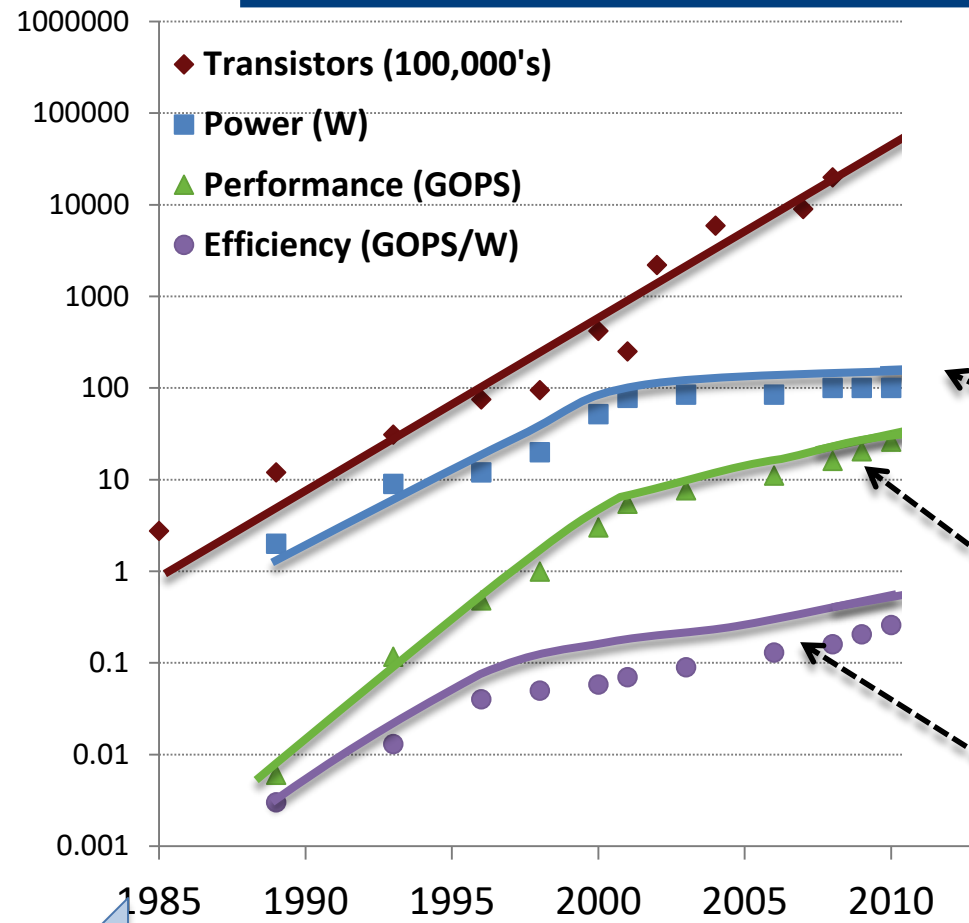
⇒ lower the *average cost*

$$\text{average cost} = \text{total cost} / N$$

$$= (\text{overhead} / N) + \text{per-unit cost}$$

Trends in computer architecture

A Paradigm Shift In Computing



IEEE Computer—April 2001
T. Mudge

Limits on heat extraction

Stagnates performance growth

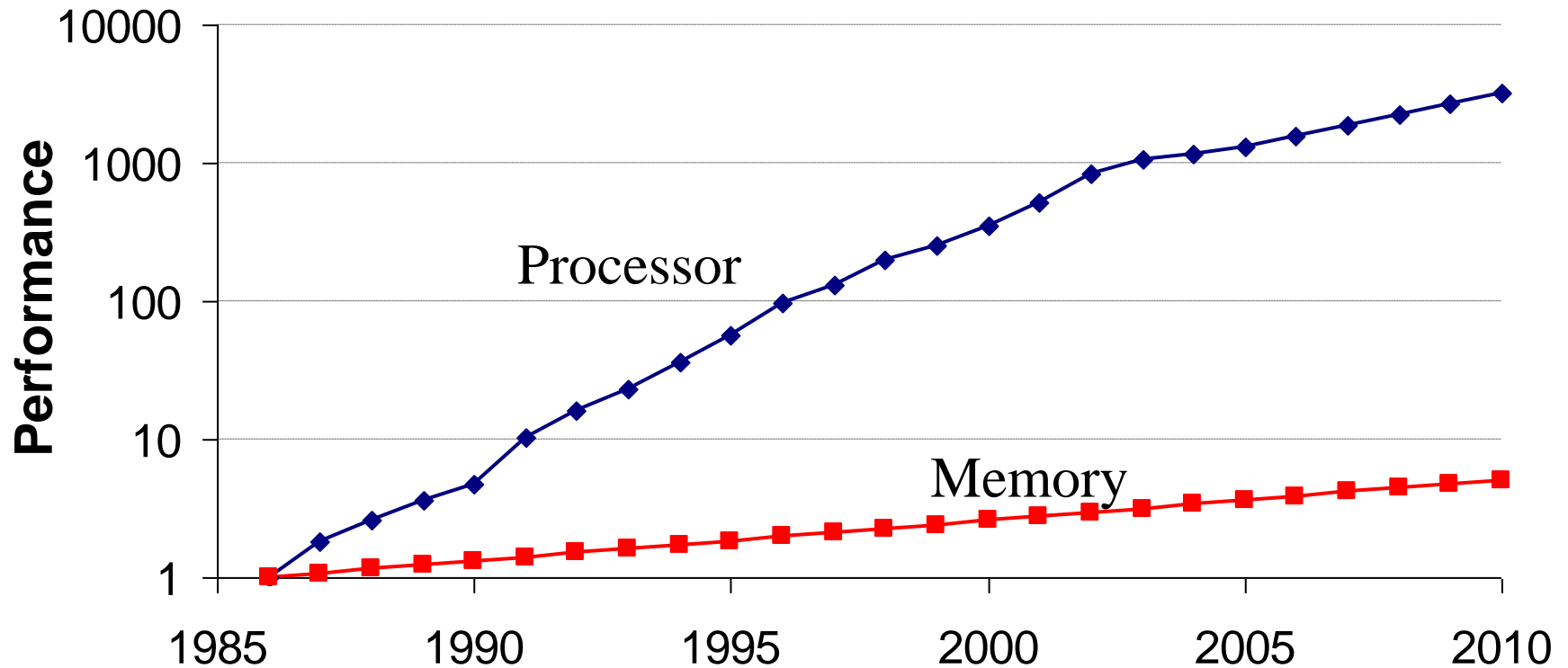
Limits on energy-efficiency of operations

Era of High Performance Computing

Era of Energy-Efficient Computing

c. 2000

The Memory Wall



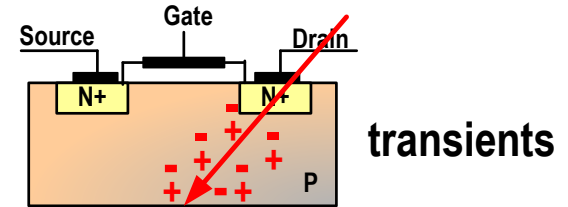
Source: Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed.

Today: 1 mem access \approx 500 arithmetic ops

Reliability Wall

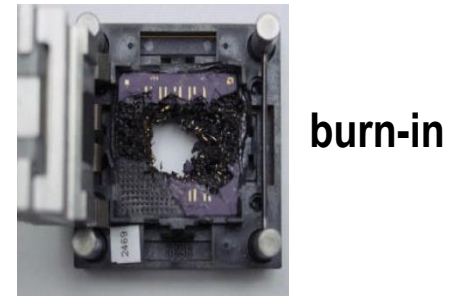
Transient faults

- E.g, high-energy particle strikes



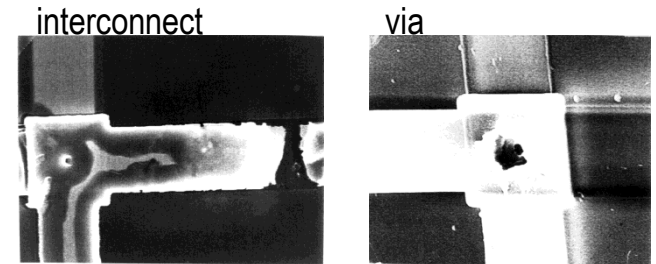
Manufacturing faults

- E.g., broken connections



Wearout faults

- E.g., Electromigration



Device variability

(not all transistors created equal)

Review of basic pipelining

Review of basic pipelining

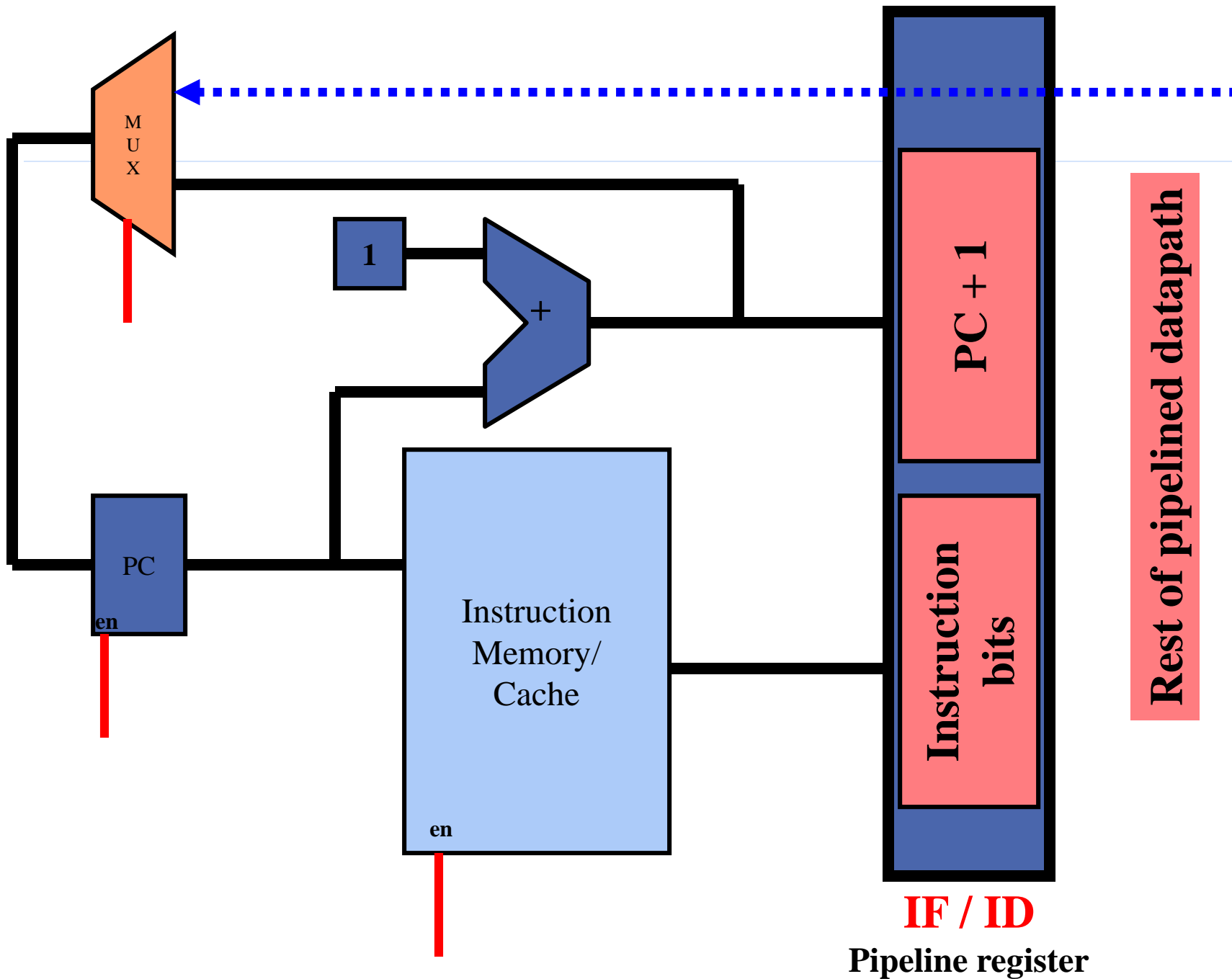
- 5 stage “RISC” load-store architecture
 - About as simple as things get
 1. Instruction fetch:
 - get instruction from memory/cache
 2. Instruction decode:
 - translate opcode into control signals and read regs
 3. Execute:
 - perform ALU operation
 4. Memory:
 - Access memory if load/store
 5. Writeback/retire:
 - update register file

Pipelined implementation

- Break the execution of the instruction into cycles (5 in this case).
- Design a separate datapath **stage** for the execution performed during each cycle.
- Build **pipeline registers** to communicate between the stages.

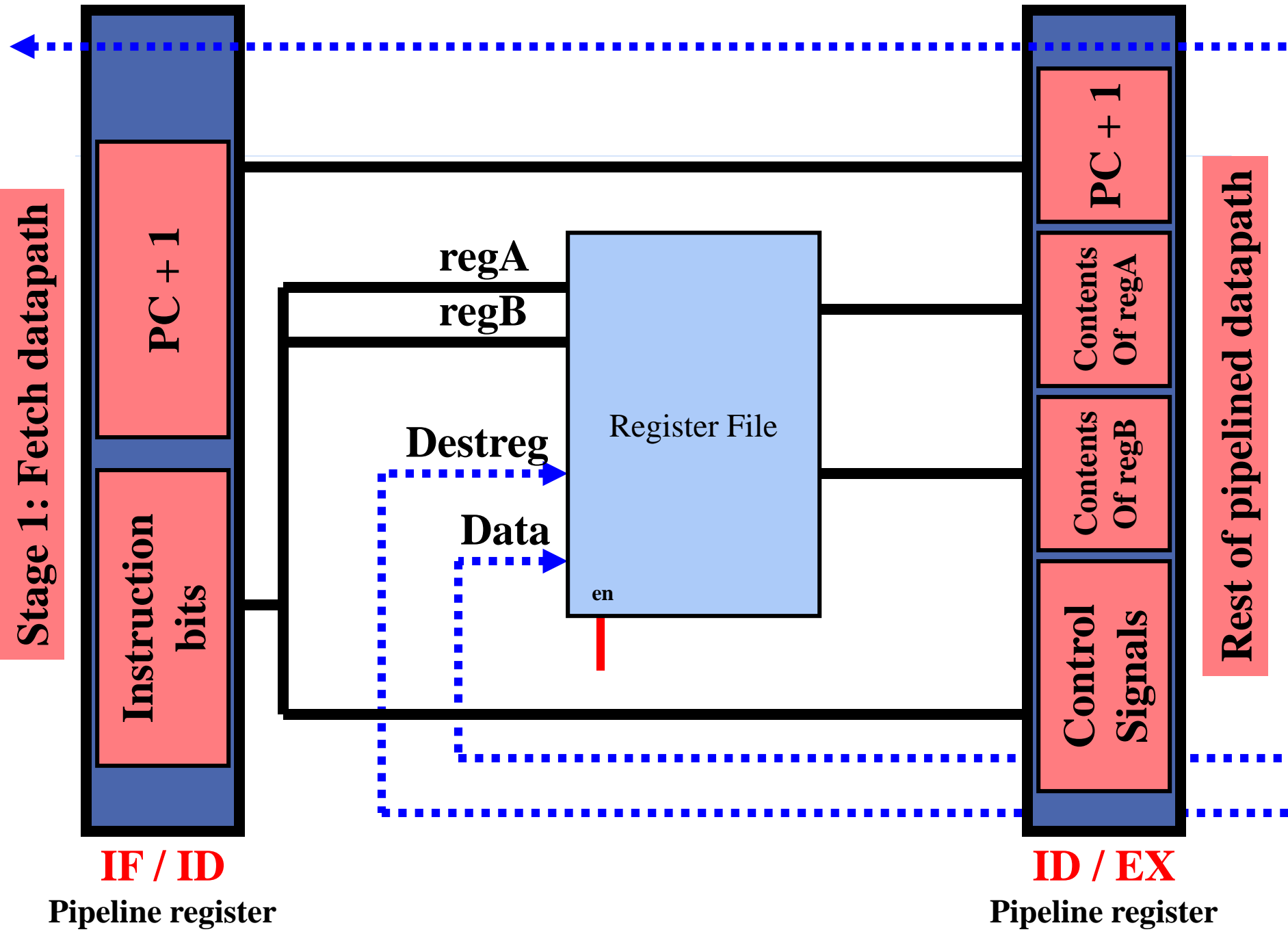
Stage 1: Fetch

- Design a datapath that can fetch an instruction from memory every cycle.
 - Use PC to index memory to read instruction
 - Increment the PC (assume no branches for now)
- Write everything needed to complete execution to the **pipeline register (IF/ID)**
 - The next **stage** will read this pipeline register.
 - Note that pipeline register must be edge triggered



Stage 2: Decode

- Design a datapath that reads the IF/ID pipeline register, decodes instruction and reads register file (specified by regA and regB of instruction bits).
 - Decode can be easy, just pass on the opcode and let later stages figure out their own control signals for the instruction.
- Write everything needed to complete execution to the pipeline register (**ID/EX**)
 - Pass on the offset field and both destination register specifiers (or simply pass on the whole instruction!).
 - Including PC+1 even though decode didn't use it.



Stage 1: Fetch datapath

IF / ID

Pipeline register

Rest of pipelined datapath

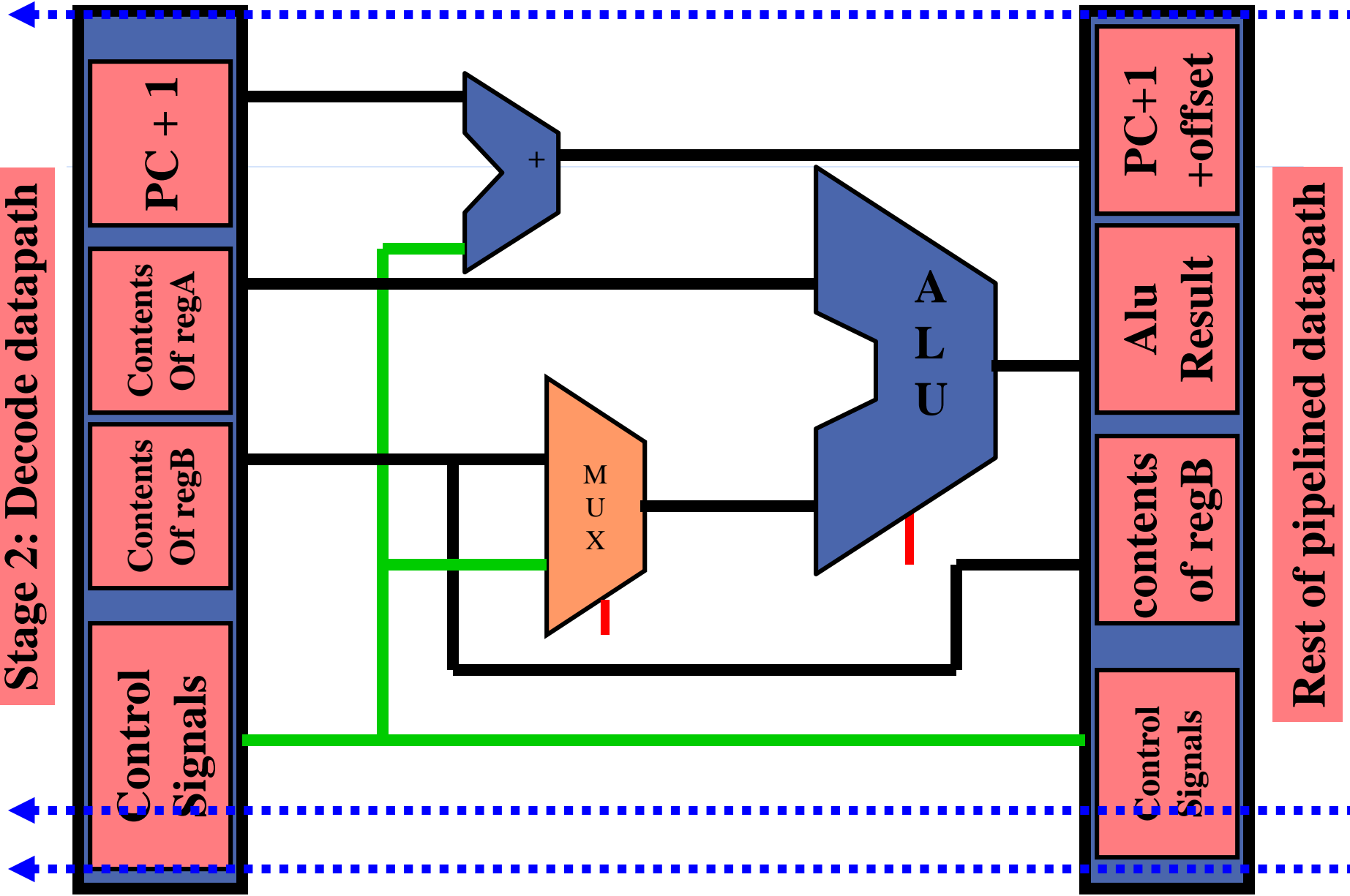
ID / EX

Pipeline register

Stage 3: Execute

- Design a datapath that performs the proper ALU operation for the instruction specified and the values present in the ID/EX pipeline register.
 - The inputs are the contents of regA and either the contents of RegB or the offset field on the instruction.
 - Also, calculate $PC+1+offset$ in case this is a branch.
- Write everything needed to complete execution to the pipeline register (**EX/Mem**)
 - ALU result, contents of regB and $PC+1+offset$
 - Instruction bits for opcode and destReg specifiers

Stage 2: Decode datapath



ID / EX

Pipeline register

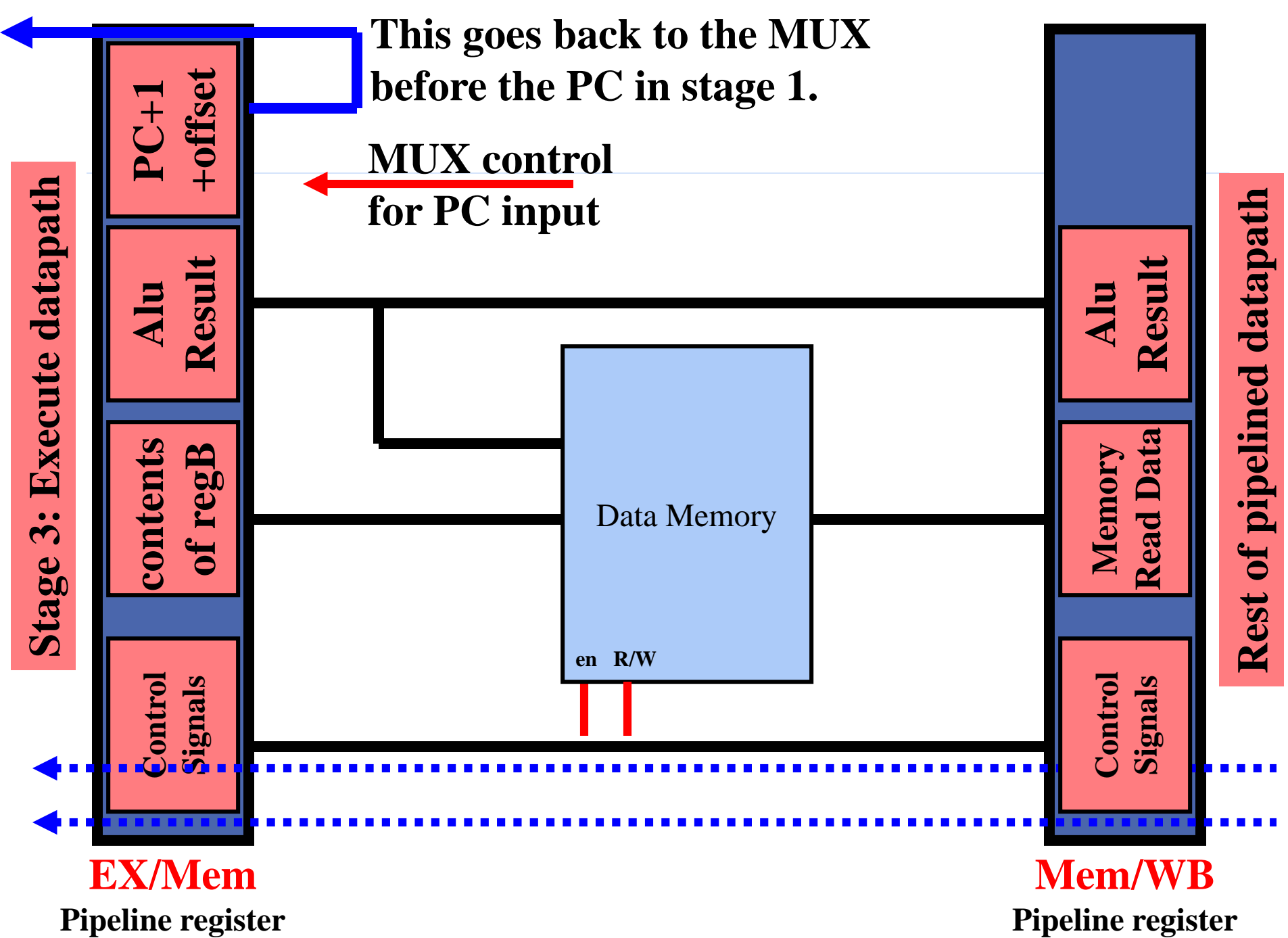
EX/Mem

Pipeline register

Rest of pipelined datapath

Stage 4: Memory Operation

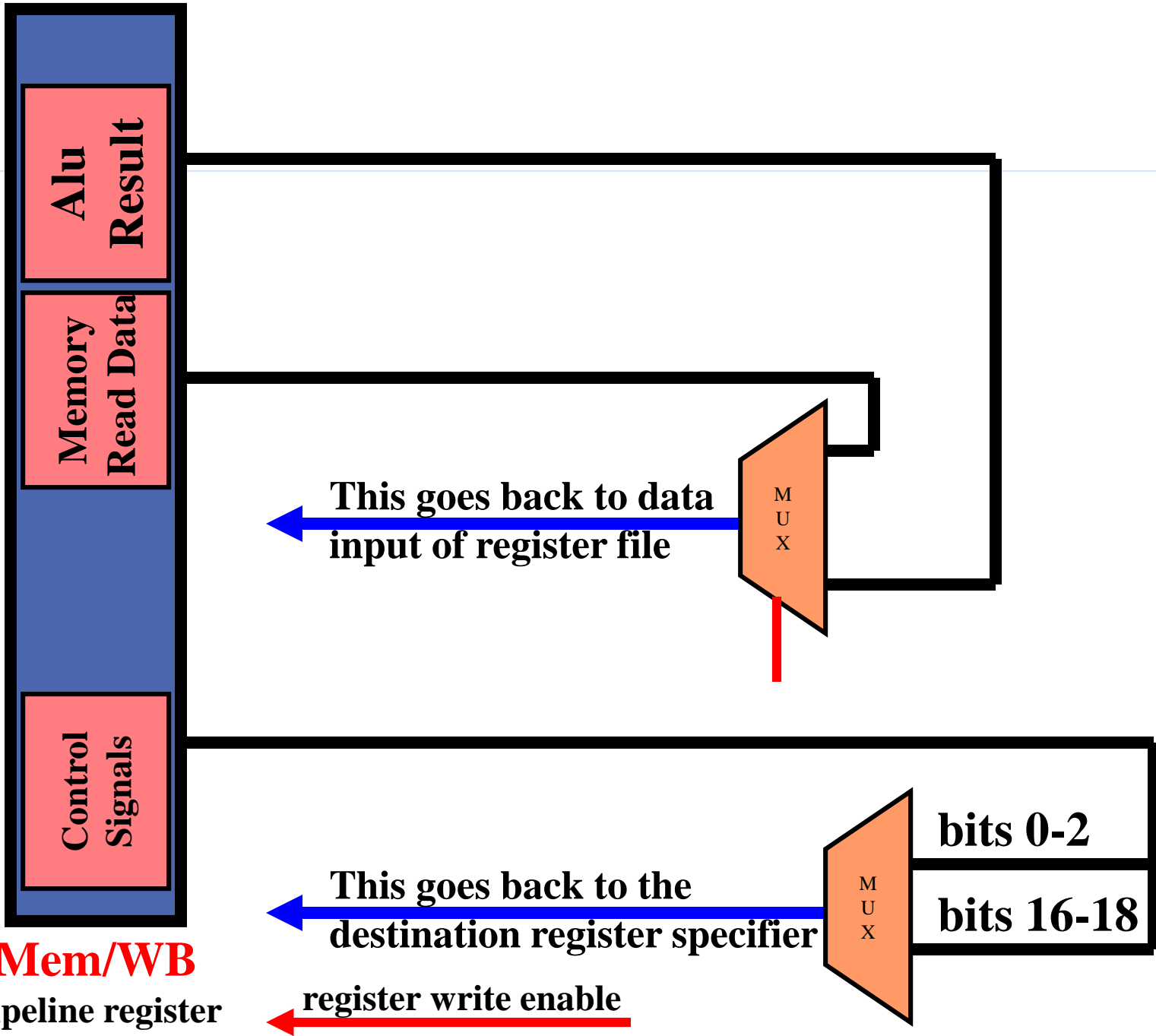
- Design a datapath that performs the proper memory operation for the instruction specified and the values present in the EX/Mem pipeline register.
 - ALU result contains address for **ld** and **st** instructions.
 - Opcode bits control memory R/W and enable signals.
- Write everything needed to complete execution to the pipeline register (**Mem/WB**)
 - ALU result and MemData
 - Instruction bits for opcode and destReg specifiers



Stage 5: Write back

- Design a datapath that completes the execution of this instruction, writing to the register file if required.
 - Write MemData to destReg for ld instruction
 - Write ALU result to destReg for add or nand instructions.
 - Opcode bits also control register write enable signal.

Stage 4: Memory datapath



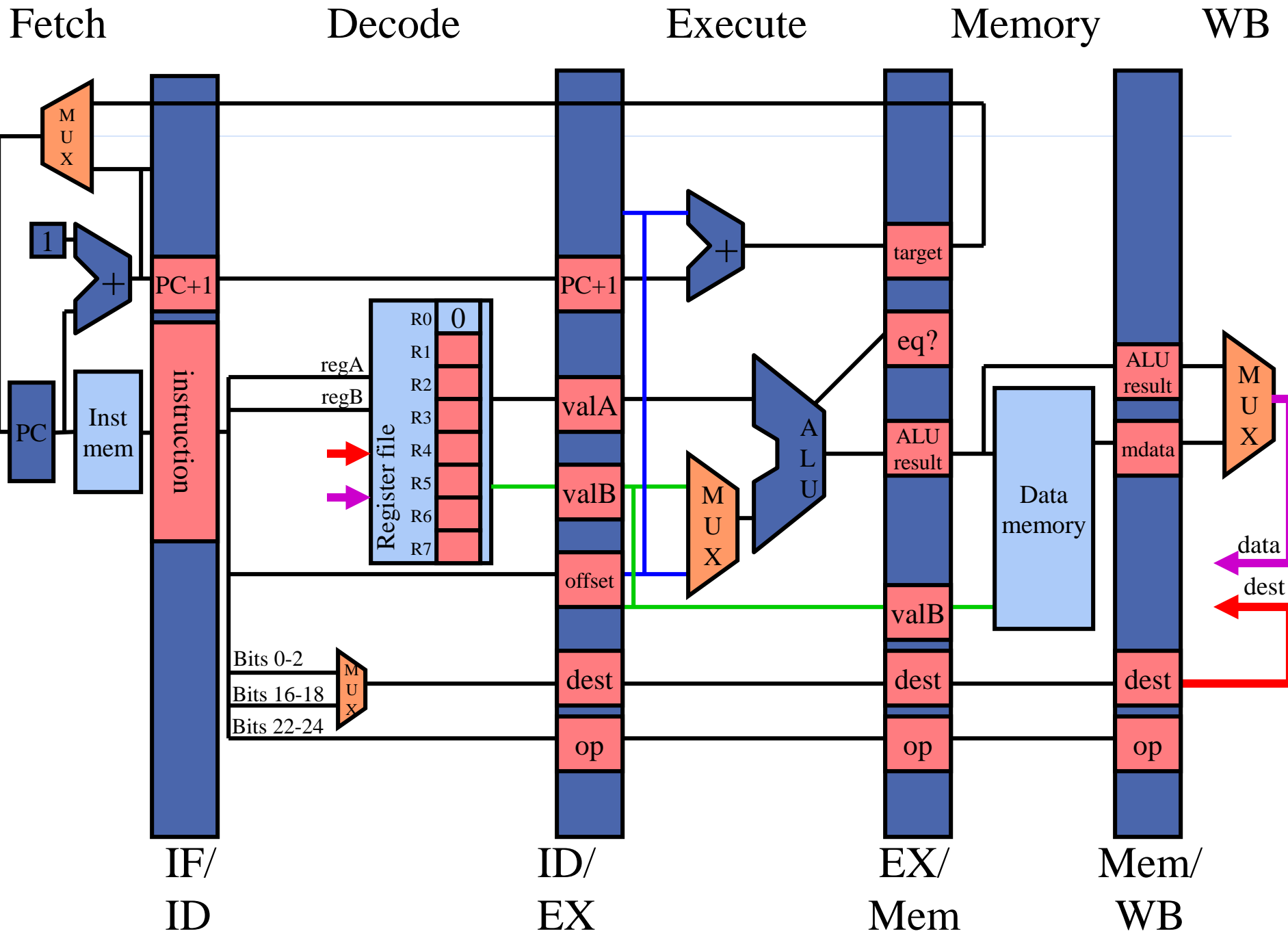
Mem/WB

Pipeline register

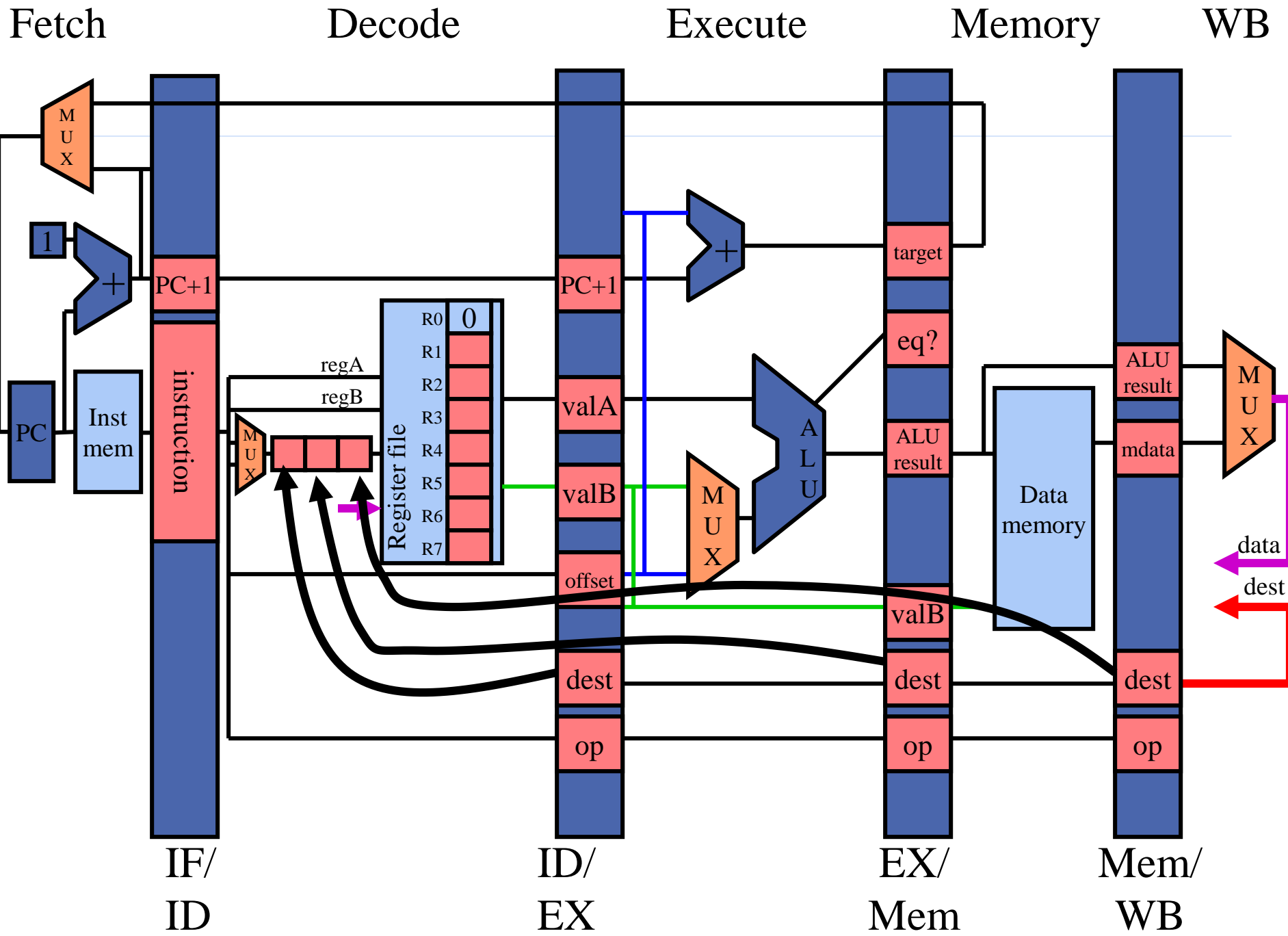
What can go wrong?

- **Data hazards:** since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if it is about to be written.
- **Control hazards:** A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- **Exceptions:** How do you handle exceptions in a pipelined processor with 5 instructions in flight?

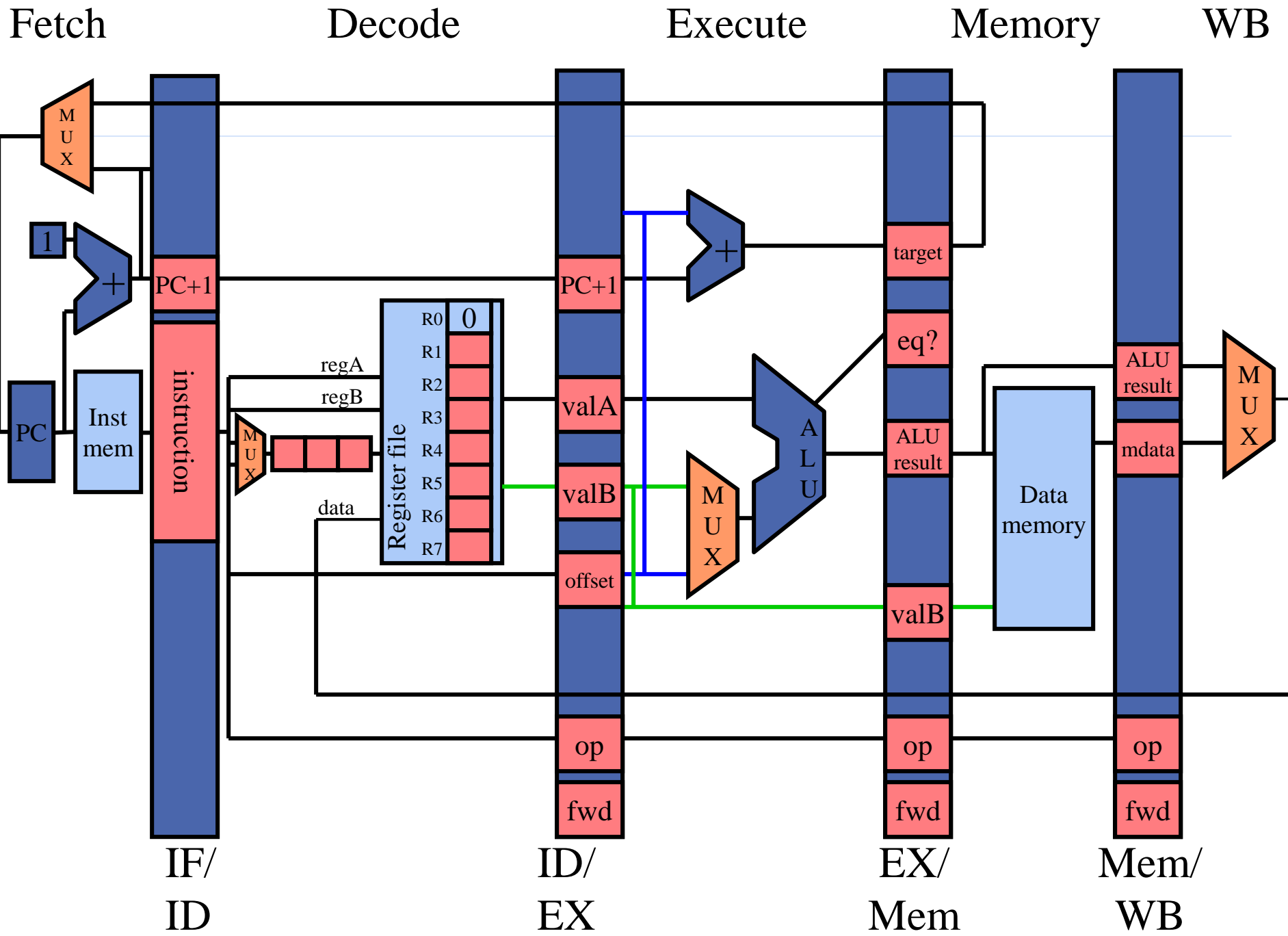
Basic Pipelining



Basic Pipelining



Basic Pipelining



Measuring performance

This will likely not be covered in class but you need to read it.

Metrics of Performance

Time (latency)

- ❑ elapsed time vs. processor time

Rate (bandwidth or throughput)

- ❑ performance = rate = work per time

Distinction is sometimes blurred

- ❑ consider batched vs. interactive processing
- ❑ consider overlapped vs. non-overlapped processing
- ❑ may require conflicting optimizations

What is the most popular metric of performance?

The "Iron Law" of Processor Performance

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

Architecture --> Implementation --> Realization

Compiler Designer

Processor Designer

Chip Designer

MIPS

MIPS - Millions of Instructions per Second

$$\text{MIPS} = \frac{\text{\# of instructions}}{\text{benchmark}} \times \frac{\text{benchmark}}{\text{total run time}}$$

1,000,000

When comparing two machines (A, B) with the same **instruction set**, MIPS is a fair comparison (*sometimes...*)

But, MIPS can be a “*meaningless indicator of performance...*”

- instruction sets are not equivalent
- different programs use a different instruction mix
- instruction count is not a reliable indicator of work
 - ❑ some optimizations add instructions
 - ❑ instructions have varying work

MIPS (Cont.)

Example:

- ❑ Machine A has a special instruction for performing square root
 - It takes 100 cycles to execute
- ❑ Machine B doesn't have the special instruction
 - must perform square root in software using simple instructions
 - e.g, Add, Mult, Shift each take 1 cycle to execute

- ❑ Machine A: $1/100$ MIPS = 0.01 MIPS
- ❑ Machine B: 1 MIPS

MFLOPS

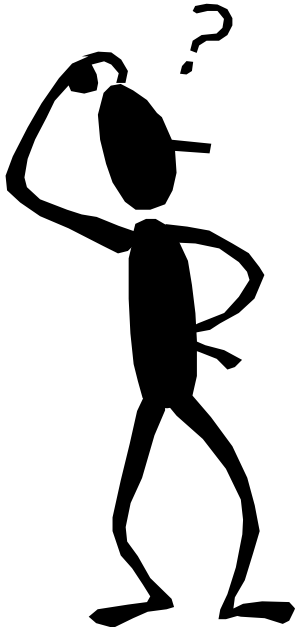
$$\text{MFLOPS} = (\text{FP ops/program}) \times (\text{program/time}) \times 10^{-6}$$

Popular in scientific computing

- ❑ There was a time when FP ops were much much slower than regular instructions (i.e., off-chip, sequential execution)

Not great for “predicting” performance because it

- ❑ ignores other instructions (e.g., load/store)
- ❑ not all FP ops are created equally
- ❑ depends on how FP-intensive program is



Beware of “peak” MFLOPS!

Comparing Performance

Often, we want to compare the performance of different machines or different programs. Why?

- ❑ help architects understand which is “better”
- ❑ give marketing a “silver bullet” for the press release
- ❑ help customers understand why they should buy <my machine>

Performance vs. Execution Time

Often, we use the phrase “*X is faster than Y*”

- Means the response time or execution time is lower on X than on Y
- Mathematically, “X is N times faster than Y” means

$$\frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = N$$

$$\frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = N = \frac{1/\text{Performance}_Y}{1/\text{Performance}_X} = \frac{\text{Performance}_X}{\text{Performance}_Y}$$

Performance and Execution time are *reciprocals*

- Increasing performance decreases execution time

By Definition

Machine A is n times faster than machine B iff
 $\text{perf}(A)/\text{perf}(B) = \text{time}(B)/\text{time}(A) = n$

Machine A is $x\%$ faster than machine B iff
 $\text{perf}(A)/\text{perf}(B) = \text{time}(B)/\text{time}(A) = 1 + x/100$

E.g., A 10s, B 15s

$15/10 = 1.5 \rightarrow$ A is 1.5 times faster than B

$15/10 = 1 + 50/100 \rightarrow$ A is 50% faster than B

Remember to compare “performance”

Let's Try a Simpler Example

Two machines timed on two benchmarks

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

- ❑ How much faster is Machine A than Machine B?
- ❑ Attempt 1: **ratio of run times, normalized to Machine A times**

program1: $4/2$

program2 : $8/12$

- ❑ Machine A ran 2 times faster on program 1, $2/3$ times faster on program 2
- ❑ On **average**, Machine A is **$(2 + 2/3) / 2 = 4/3$** times faster than Machine B

It turns this “**averaging**” stuff can fool us; watch...

Example (con't)

Two machines timed on two benchmarks

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

- ❑ How much faster is Machine A than B?
- ❑ Attempt 2: **ratio of run times, normalized to Machine B times**

program 1: 2/4 program 2 : 12/8

- ❑ Machine A ran program 1 in $1/2$ the time and program 2 in $3/2$ the time
- ❑ On **average, $(1/2 + 3/2) / 2 = 1$**
- ❑ Put another way, Machine A is **1.0 times faster** than Machine B

Example (con't)

Two machines timed on two benchmarks

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds	4 seconds
Program 2	12 seconds	8 seconds

- ❑ How much faster is Machine A than B?
- ❑ Attempt 3: **ratio of run times, aggregate (total sum) times, norm. to A**
 - Machine A took 14 seconds for both programs
 - Machine B took 12 seconds for both programs
 - Therefore, Machine A takes $14/12$ of the time of Machine B
 - Put another way, Machine A is **6/7 faster** than Machine B

Which is Right?

Question:

- How can we get **three different answers**?

Solution

- Because, while they are all **reasonable** calculations...

...each answers a *different* question

We need to be more precise in understanding and posing these performance & metric questions

Using a Weighted Sum (or weighted average)

	<u>Machine A</u>	<u>Machine B</u>
Program 1	2 seconds (20%)	4 seconds (20%)
Program 2	12 seconds (80%)	8 seconds (80%)
Total	10 seconds	7.2 seconds

- Allows us to determine relative performance $10/7.2 = 1.38$
--> Machine B is 1.38 times faster than Machine A

Summary

Performance is important to measure

- ❑ For architects comparing different deep mechanisms
- ❑ For developers of software trying to optimize code, applications
- ❑ For users, trying to decide which machine to use, or to buy

Performance metrics are subtle

- ❑ **Easy to mess up the “machine A is XXX times faster than machine B” numerical performance comparison**
- ❑ You need to know exactly what you are measuring: time, rate, throughput, CPI, cycles, etc
- ❑ You need to know how combining these to give aggregate numbers does different kinds of “distortions” to the individual numbers
- ❑ No metric is perfect, so lots of emphasis on standard benchmarks today