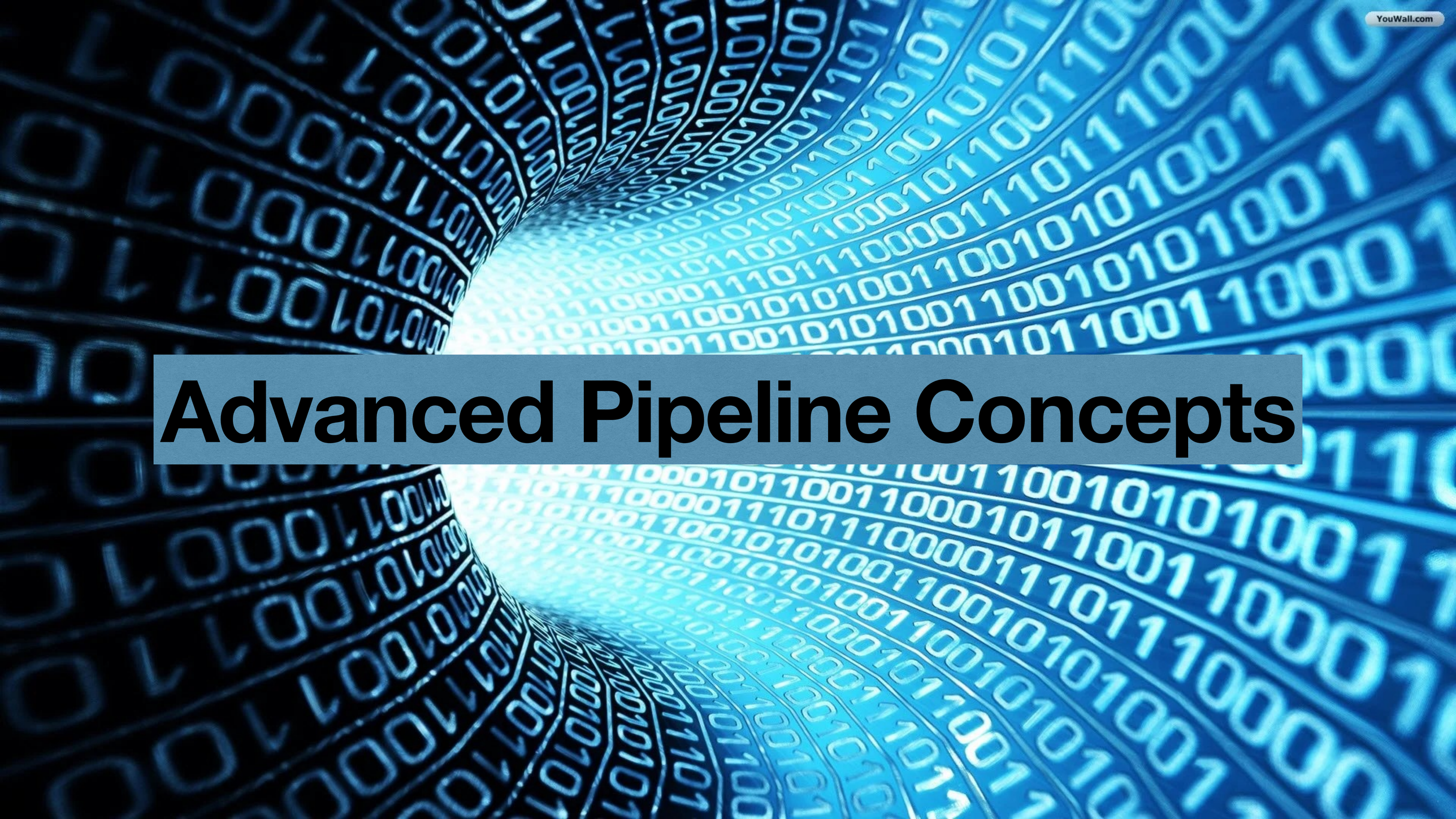
Digital Logic Design + Computer Architecture

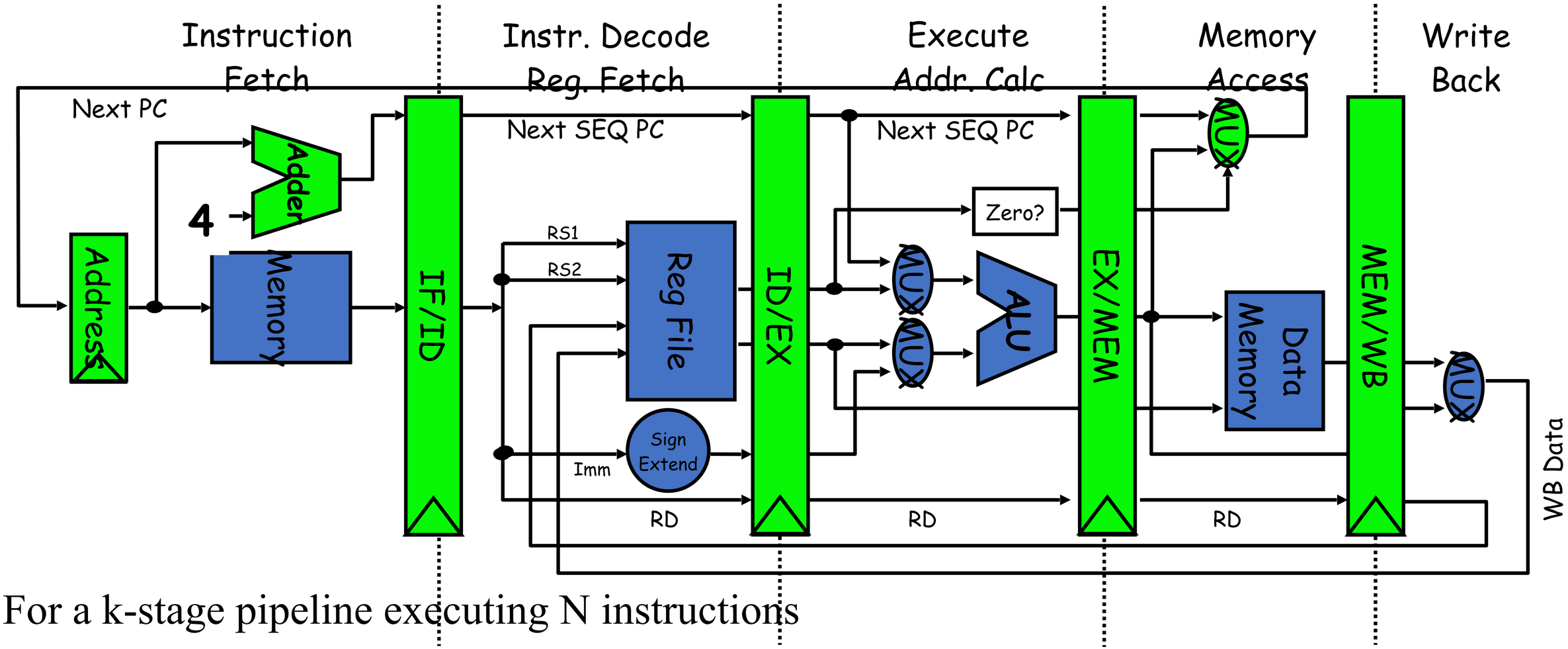
Sayandeep Saha

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



Advanced Pipeline Concepts

Pipeline Recap…



For a k-stage pipeline executing N instructions

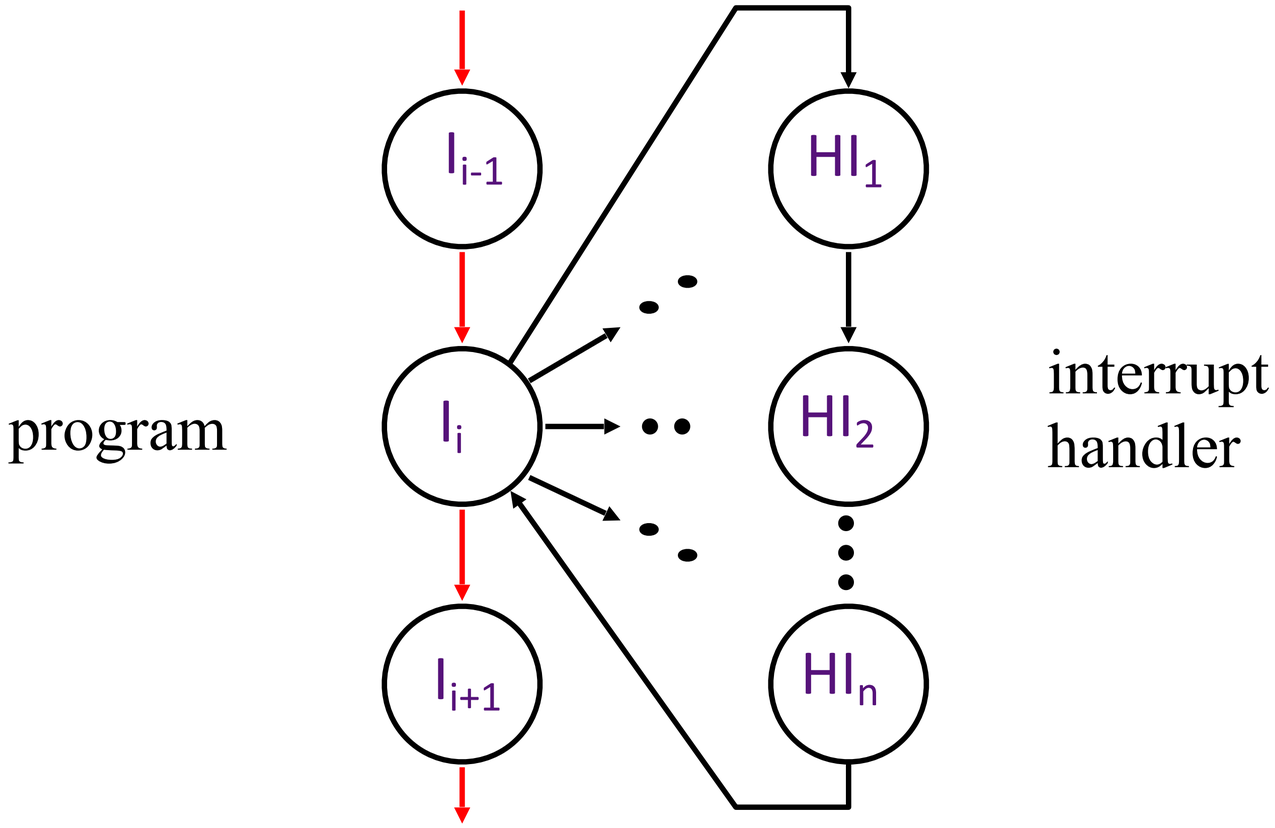
first instruction: k cycles

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

Exception/Interrupt

An unscheduled event that disrupts program (instructions) in action.

Interrupt Handling

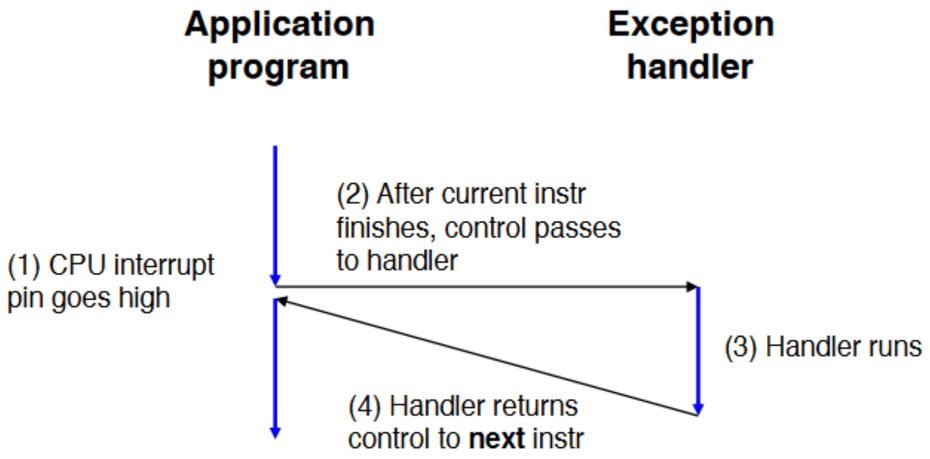


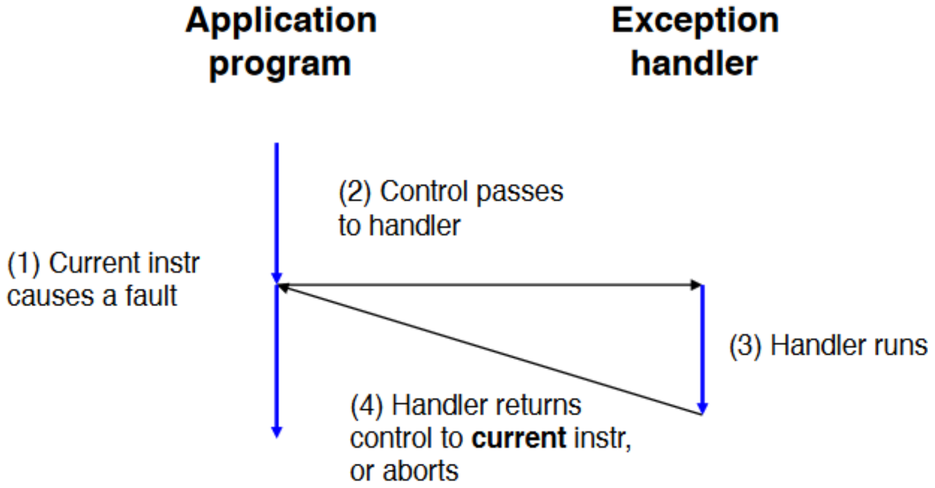
An *external or internal event* that needs to be processed. The event is usually unexpected or rare from program's point of view.

Types of Interrupts

- Asynchronous: an *external event*
  - input/output device service-request
  - timer expiration
  - power disruptions, hardware failure
- Synchronous: an *internal event (a.k.a. traps or exceptions)*
  - undefined opcode, privileged instruction
  - arithmetic overflow, FPU exception, misaligned memory access
  - *virtual memory exceptions:* page faults, TLB misses, protection violations
  - system calls, e.g., jumps into kernel

Interrupt and Exception





Interrupt Handler

Exception program counter (EPC): address of the offending instruction,

Saves EPC before enabling interrupts to allow nested interrupts

Need to mask further interrupts at least until EPC can be saved

Need to read a *status register* that indicates the cause of the interrupt

Handshake between processor and the OS

Processor:

stops the offending instruction,

makes sure all prior instructions complete,

flushes all the future instructions (in the pipeline)

Sets a register to show the cause

Saves EPC

Disables further interrupts

Jumps to pre-decided address (cause register or vectored)

Handshake between processor and the OS

OS:

Looks at the cause of the exception

Interrupt handler saves the GPRs

Handles the interrupt/exception

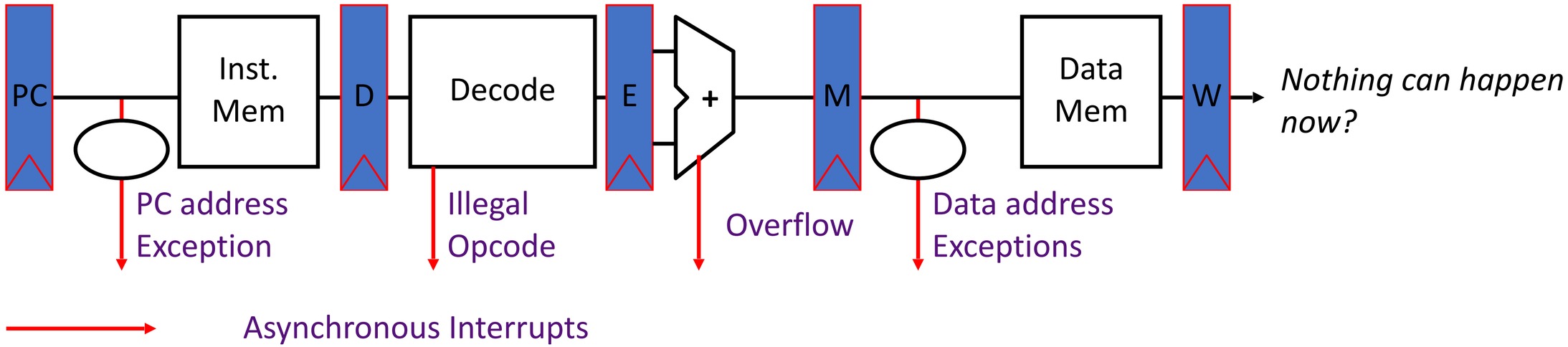
Calls RFE

Contd.

Uses a special indirect jump instruction RFE (*return-from-exception*) which

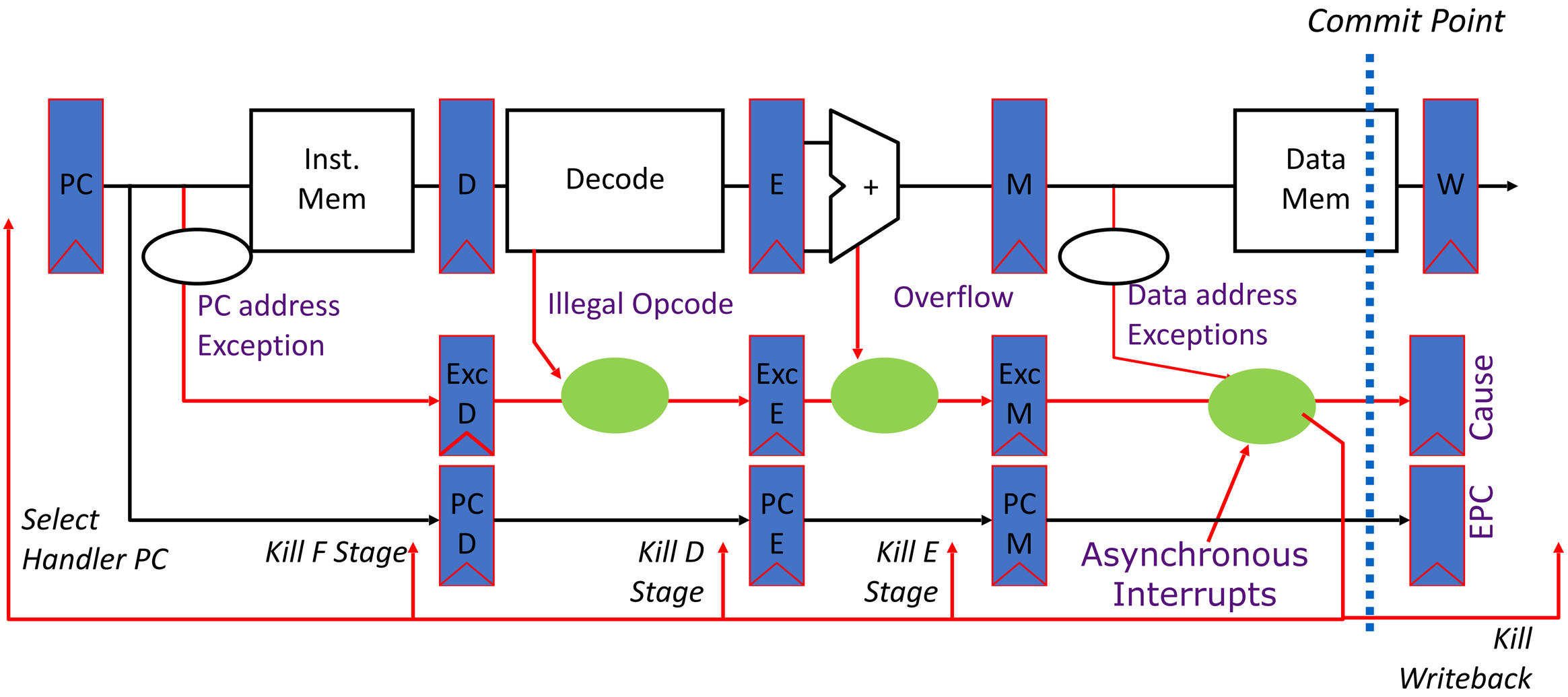
- enables interrupts
- restores the processor to the user mode

Exception handling and Pipelining



- When do we stop the pipeline for *precise* interrupts or exceptions?
- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?

Exception handling and Pipelining



Contd.

- Hold exception flags in pipeline until commit point for instructions that will be killed
- Exceptions in earlier pipe stages override later exceptions *for a given instruction*
- If exception at commit: update cause and EPC registers, kill all stages, inject handler PC into fetch stage

SuperScalar: More Instructions Per Cycle (IPC)

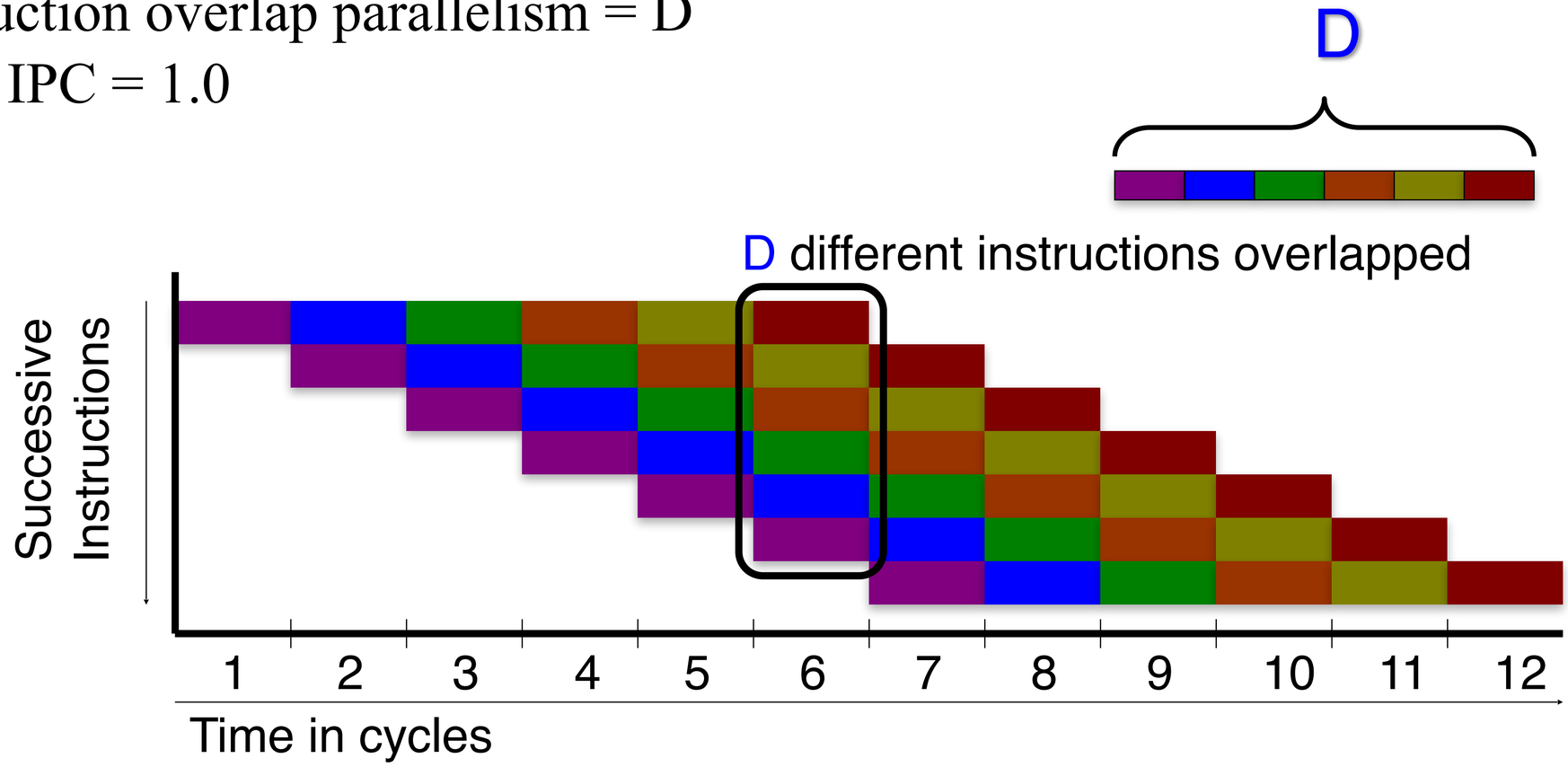
Beyond Scalar

- Scalar pipeline limited to CPI ≥ 1.0
  - Can never run more than 1 insn per cycle

- "Superscalar" can achieve CPI ≤ 1.0 (i.e., IPC ≥ 1.0)
  - *Superscalar* means executing multiple insns in parallel

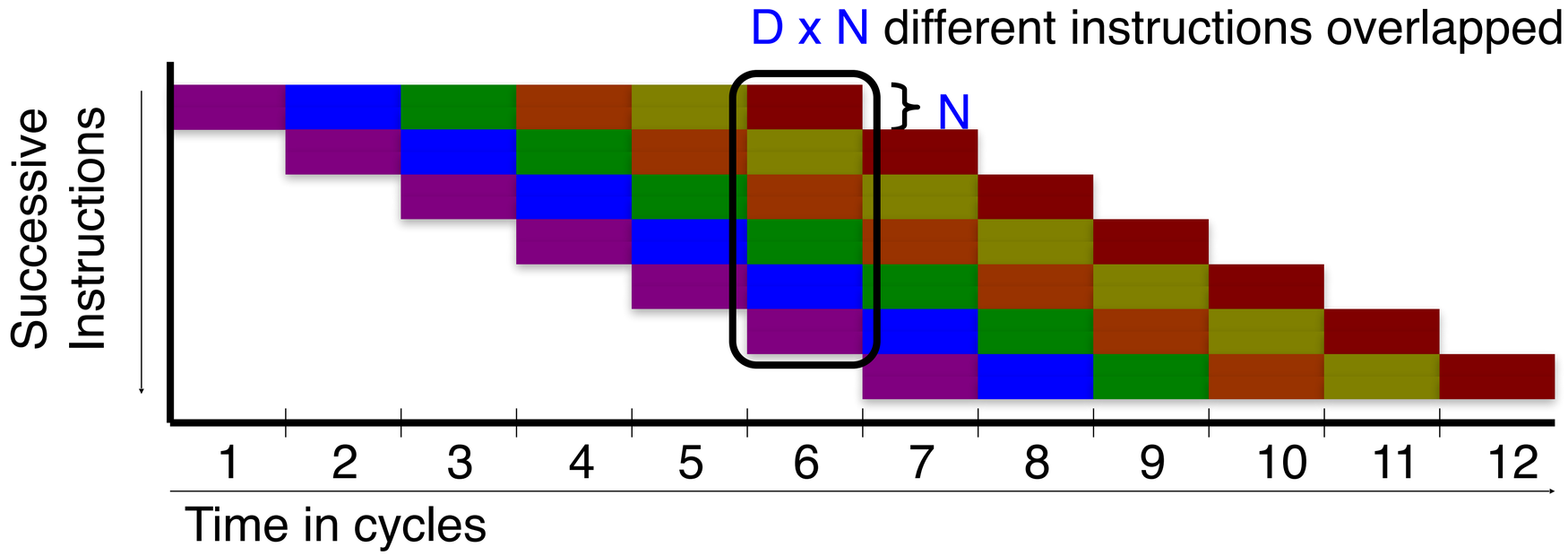
Instruction Level Parallelism (ILP)

- Scalar pipeline (baseline)
  - Instruction overlap parallelism = D
  - Peak IPC = 1.0



Superscalar Processor

- Superscalar (pipelined) Execution
  - Instruction parallelism = D x N
  - Peak IPC = N per cycle



What is the deal?

We get an IPC boost if the number of instructions fetched in one cycle are independent ☺

Complicates datapaths, multi-ported structures, complicates exception handling

Out of order (O3) processor: Pursuit of even higher IPC

Out-of-order follows data-flow order

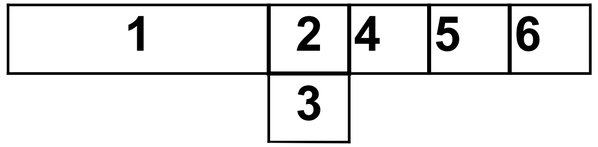
Example:

```
(1) r1 ← r4 / r7
(2) r8 ← r1 + r2
(3) r5 ← r5 + 1
(4) r6 ← r6 - r3
(5) r4 ← r5 + r6
(6) r7 ← r8 * r4
```

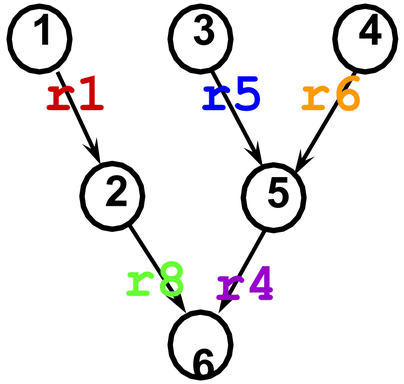
/* assume division takes 20 cycles */

In-order execution

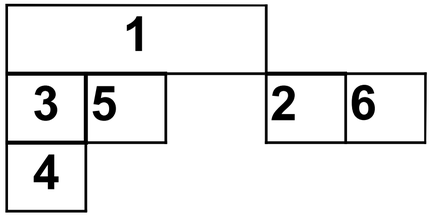
In-order (2-way superscalar)



Data Flow Graph



Out-of-order execution

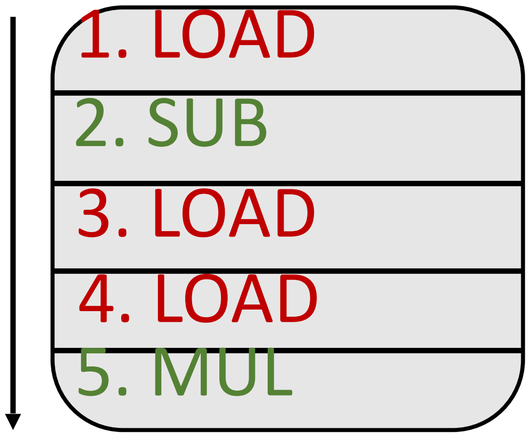


O3

Two or more instructions can execute in any order if they have no dependences (RAW, WAW, WAR)

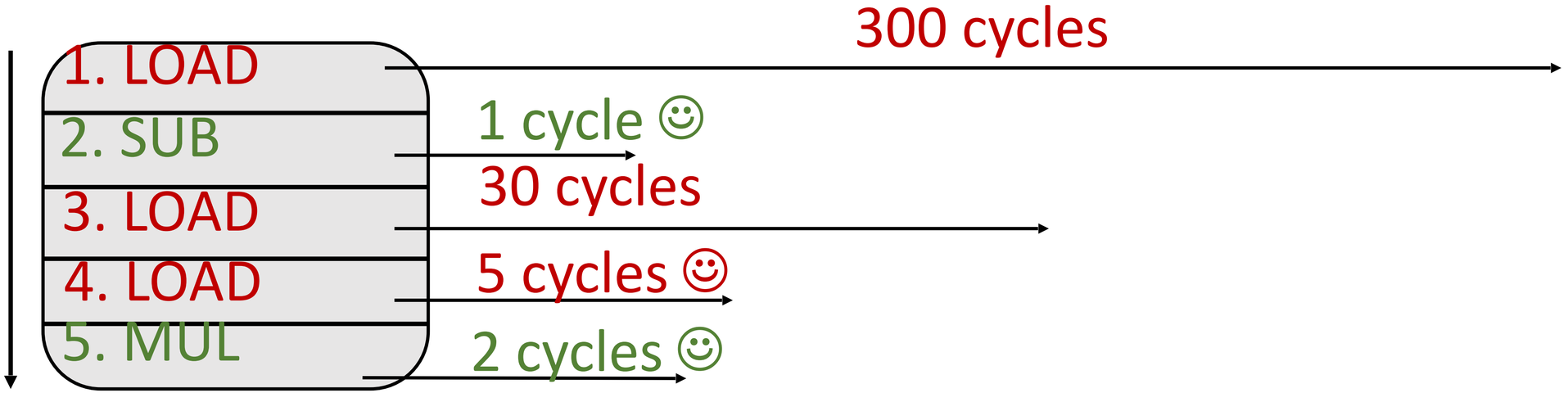
Completely orthogonal to superscalar/pipelining

O3 + Superscalar



In-order Instruction Fetch
(Multiple fetch in one cycle)

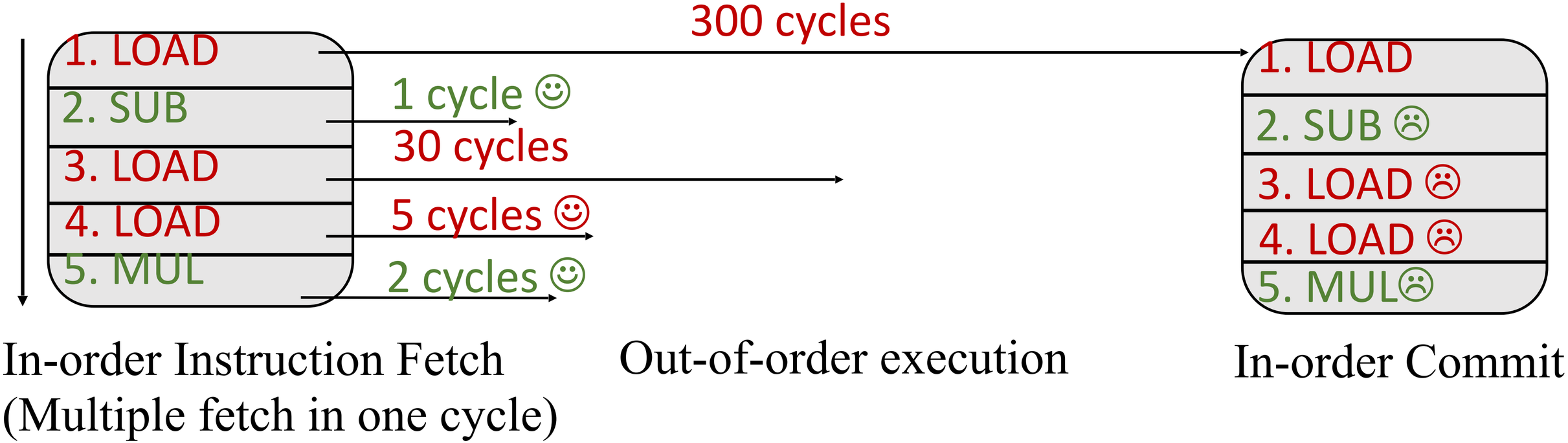
O3 + Superscalar



In-order Instruction Fetch
(Multiple fetch in one cycle)

Out-of-order execution

O3 + Superscalar



O3

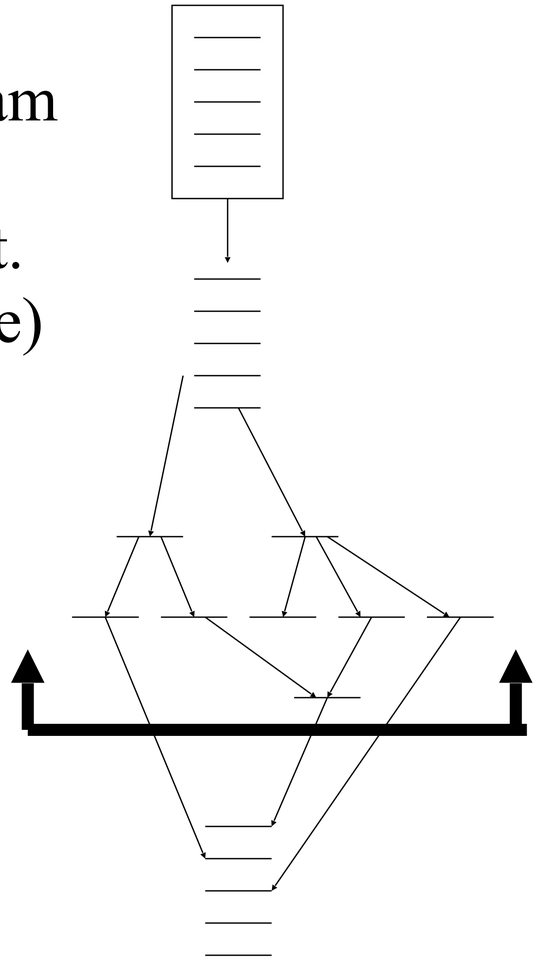
Program Form

Static program

dynamic inst.
Stream (trace)

execution
window

completed
instructions



Processing Phase

Dispatch/ dependences

inst. Issue

inst execution

inst. Reorder &
commit

The notion of Commit

After commit, the results of a committed instruction is visible to the programmer

and

the order at which instructions are fetched is also visible.

Why we need in-order commit?

Think about exceptions and precise exceptions

We should know till when we are done as per the programmer's view.

Performance: Time (Iron Law)

Time/Program =

Instructions/program X cycles/instruction X Time/cycle

Source code	ISA	microarch.
Compiler	microarch.	technology
ISA		

Example

Program p = one billion instructions

Processor takes one cycle per instruction

Processor clock is 4 GHz

CPU time = 10^{9} instructions X 1 cycle/instruction X 1/4 ns

= 0.25 second (4X faster)

Example

Program p = one billion instructions

Processor processes 10 instructions in one cycle

Processor clock is 4 GHz

CPU time = 10^{9} instructions X 0.10 cycle/instruction X 1/4 ns

= 0.025 second (40X faster)

Performance

- *Latency* (execution/response time): time to finish one task. It is additive (Performance = 1/latency)
- *Throughput* (bandwidth): number of tasks/unit time. It is not additive

Latency vs bandwidth

Latency vs Bandwidth, How they affect each other?

Latency helps bandwidth but not vice versa.

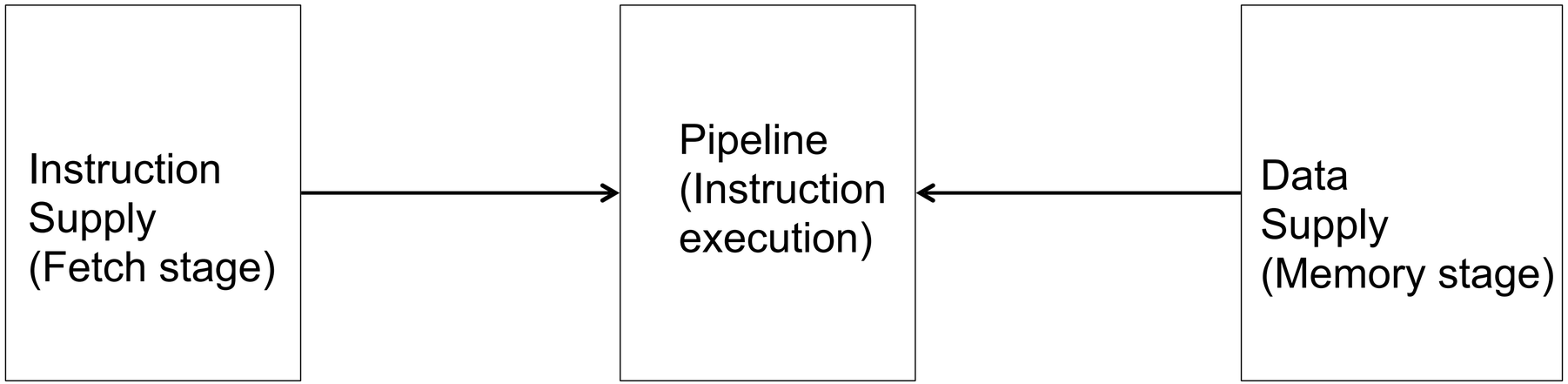
DRAM latency ↓ More # Accesses ~DRAM Bandwidth ↑

Bandwidth usually hurts latency

Queues - Bandwidth ↑ Increases latency ↓

Down the Memory Lane

The Ideal World

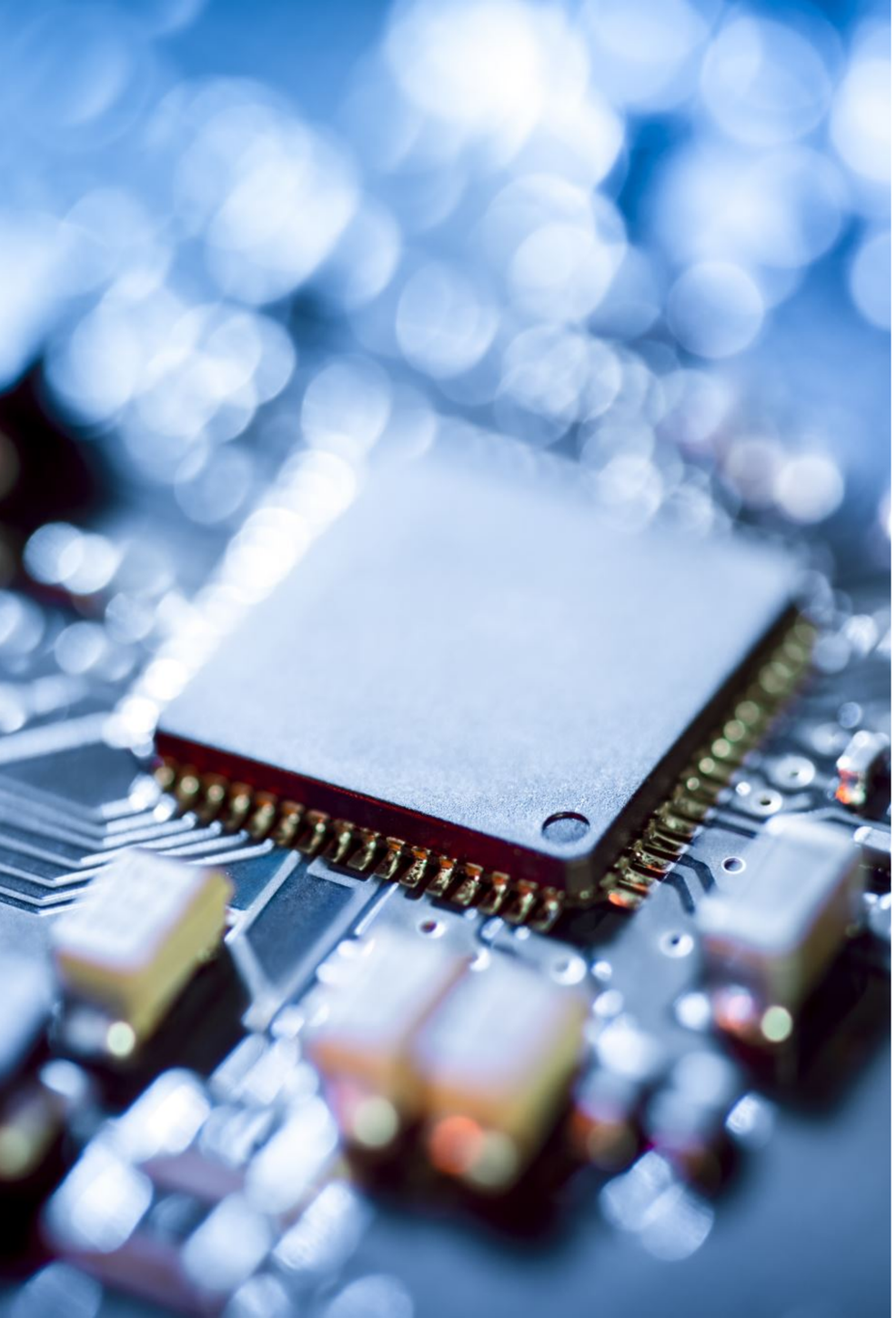


- Zero-cycle latency
- Infinite capacity
- Perfect control flow

- Zero-cycle latency
- Infinite capacity
- Infinite bandwidth

World of Memory Hierarchy: Why?

Before that What is memory?

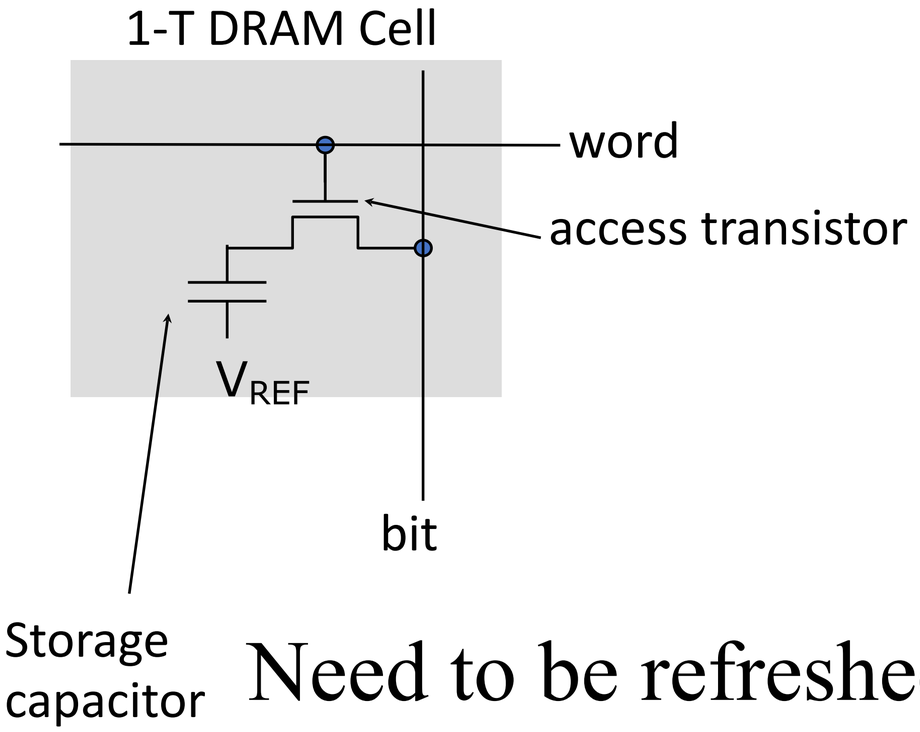


Semiconductor Memory

- Semiconductor memory began to be competitive in early 1970s
  - Intel formed to exploit market for semiconductor memory
  - Early semiconductor memory was Static RAM (SRAM). SRAM cell internals similar to a latch (cross-coupled inverters).

- First commercial Dynamic RAM (DRAM) was Intel 1103
  - 1Kbit of storage on single chip
  - charge on a capacitor used to hold value

One transistor DRAM



Denser

Value kept in one cell is as per the charge in the capacitor

Need to be refreshed periodically to maintain the charge

So dynamic RAM (DRAM)

DRAM

- Dynamic random-access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor

- Capacitor leaks
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed

SRAMs (6T to 8T)

Static RAMs. No need of refresh as no capacitor.

Faster access (no capacitor)

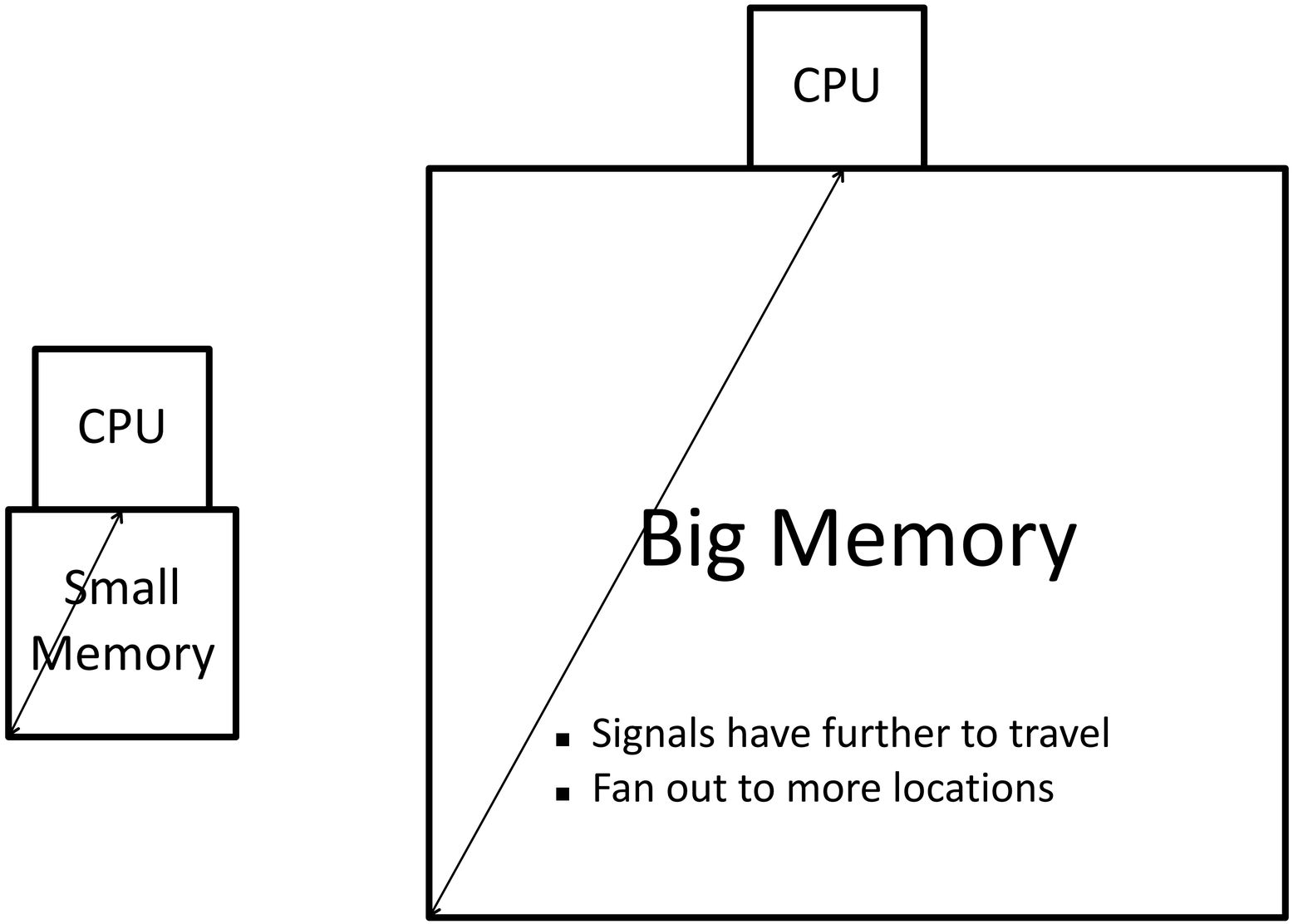
Density low as 6T: 1 bit compared to 1T: 1bit in DRAM

Costly than DRAM

The Problem

- Bigger is slower
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is more expensive (dollars and chip area)
  - SRAM, < 1000$ per GB
  - DRAM, < 20$ per GB
  - Hard Disk < 0.01$ per GB
  - These sample values scale with time
- Other technologies have their place as well
  - Flash memory, NVRAM, MRAM etc

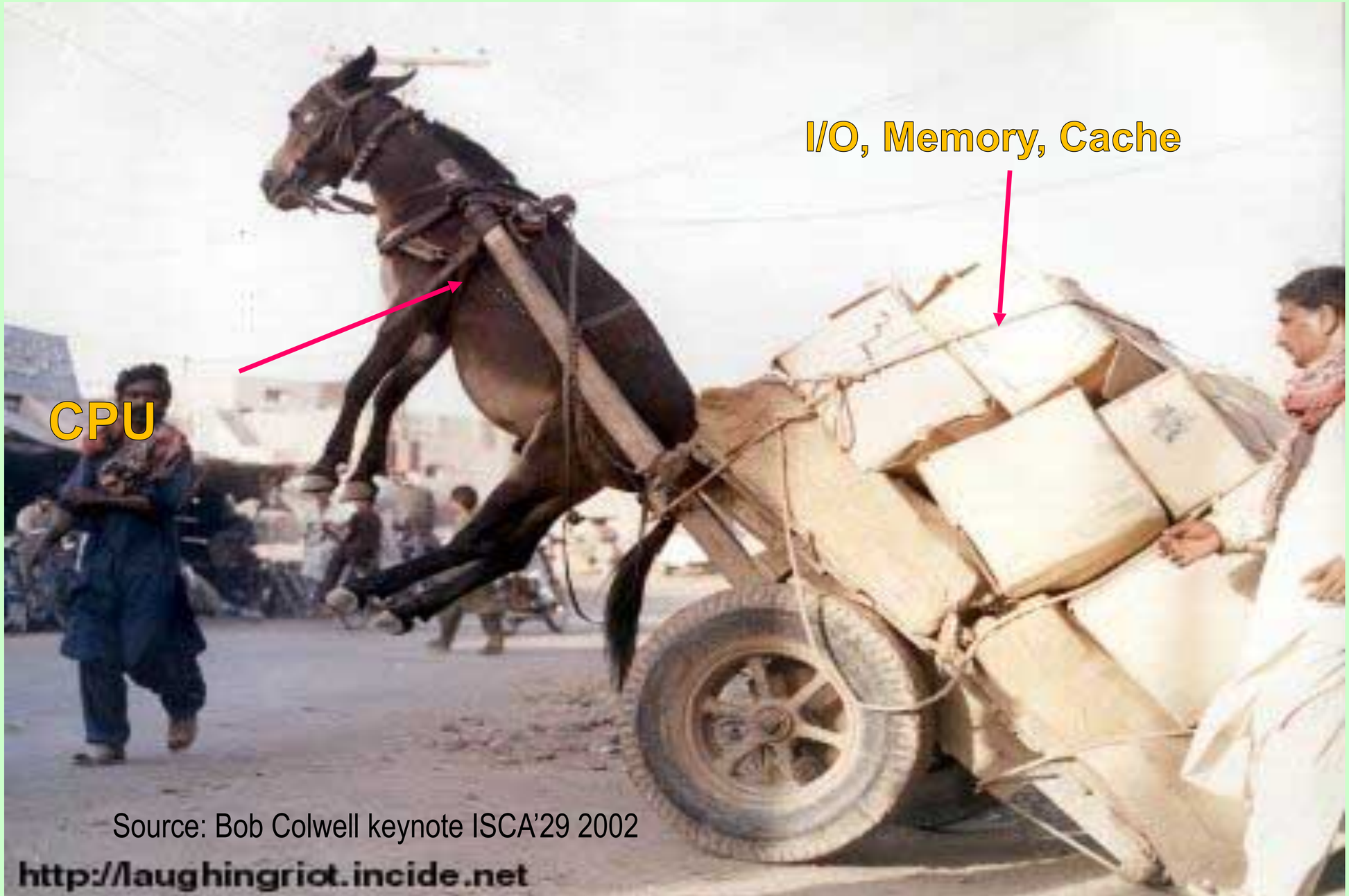
Size affects latency



An Unbalanced System

- Even a sophisticated processor may perform well below an ordinary processor:
  - Unless supported by matching performance by the memory system.
  - Unfortunately the gap is widening.
- Over the next few classes:
  - Study how memory system performance has been enhanced through various innovations and optimizations.

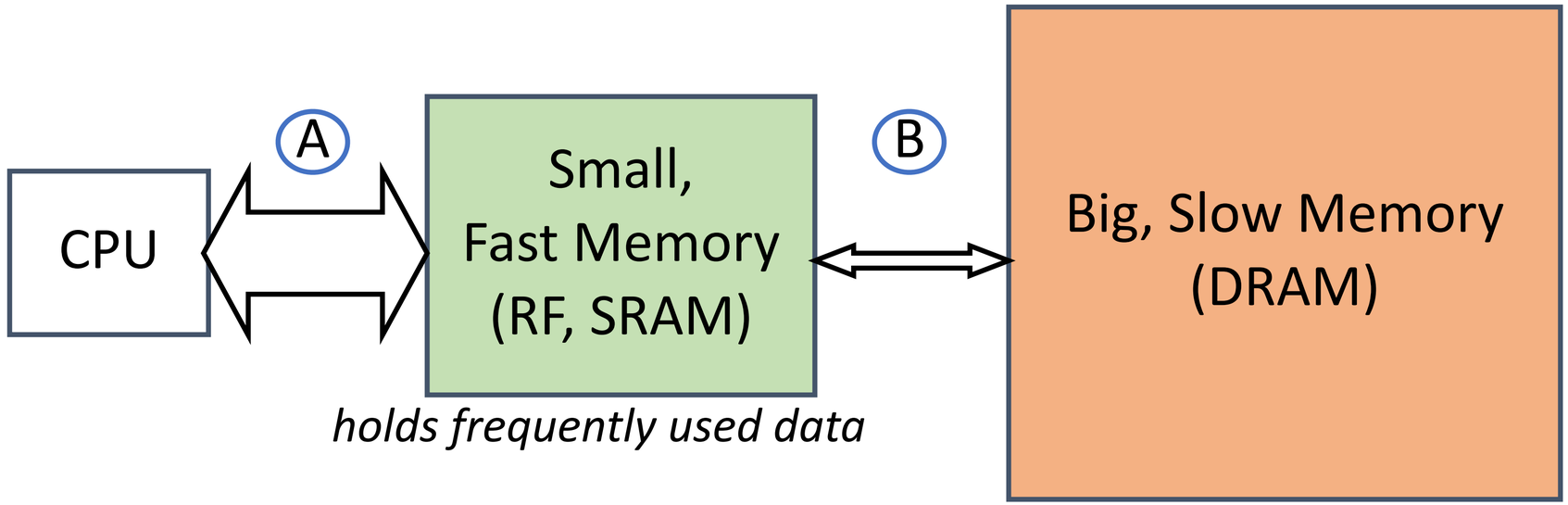
An Unbalanced System



Why Memory Hierarchy

- We want both fast and large
- But we cannot achieve both with a single level of memory
- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

Welcome to Memory Hierarchy



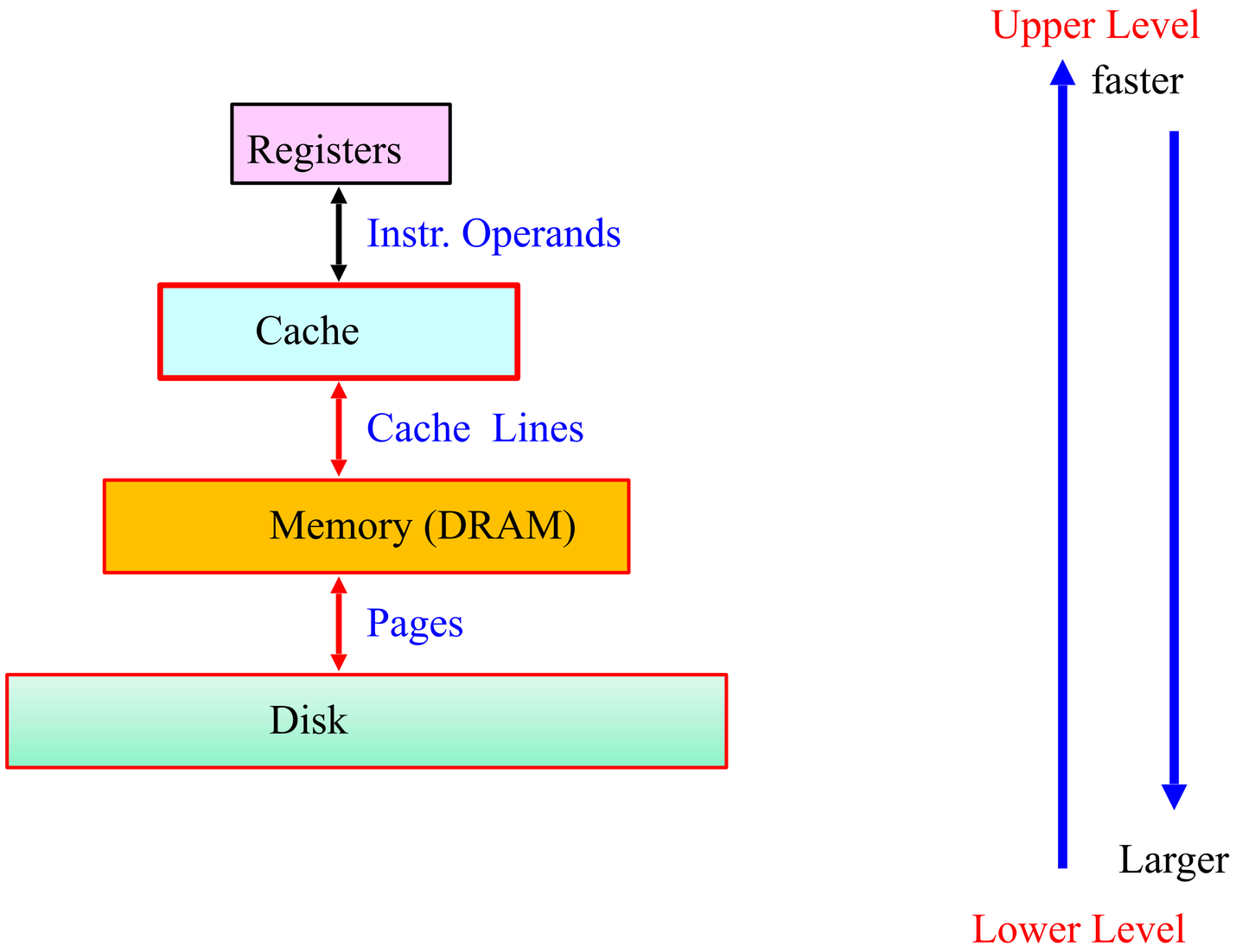
- *capacity*: Register << SRAM (Cache) << DRAM
- *latency*: Register << SRAM (Cache) << DRAM
- *bandwidth:* on-chip >> off-chip

On a data access:

if data $\in$ fast memory $\Rightarrow$ low latency access *Cache*

if data $\notin$ fast memory $\Rightarrow$ high latency access *DRAM*

Levels of the Memory Hierarchy



World with no caches

North pole ☹

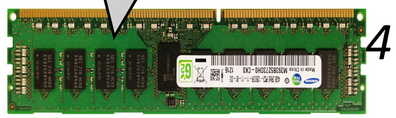
Core

32-bit Address

Data

200 to 300 cycles

Costly DRAM accesses☹



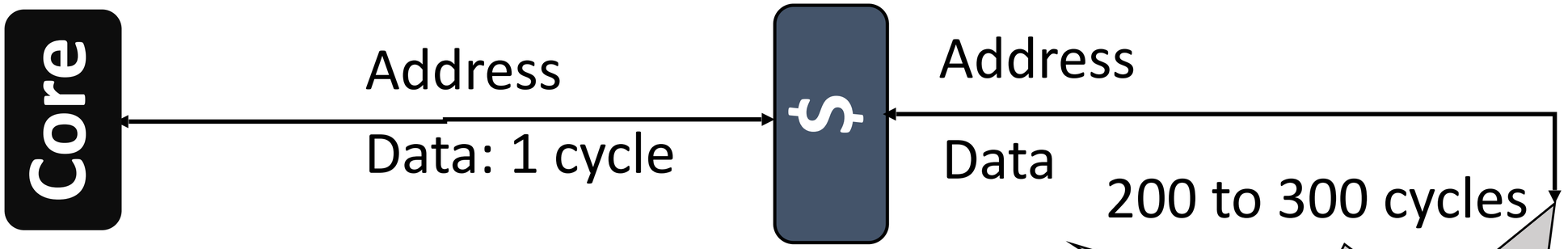
4 GB DRAM

South pole ☹

Minimizing costly DRAM accesses is critical for performance

Cache with latency

North pole ☺

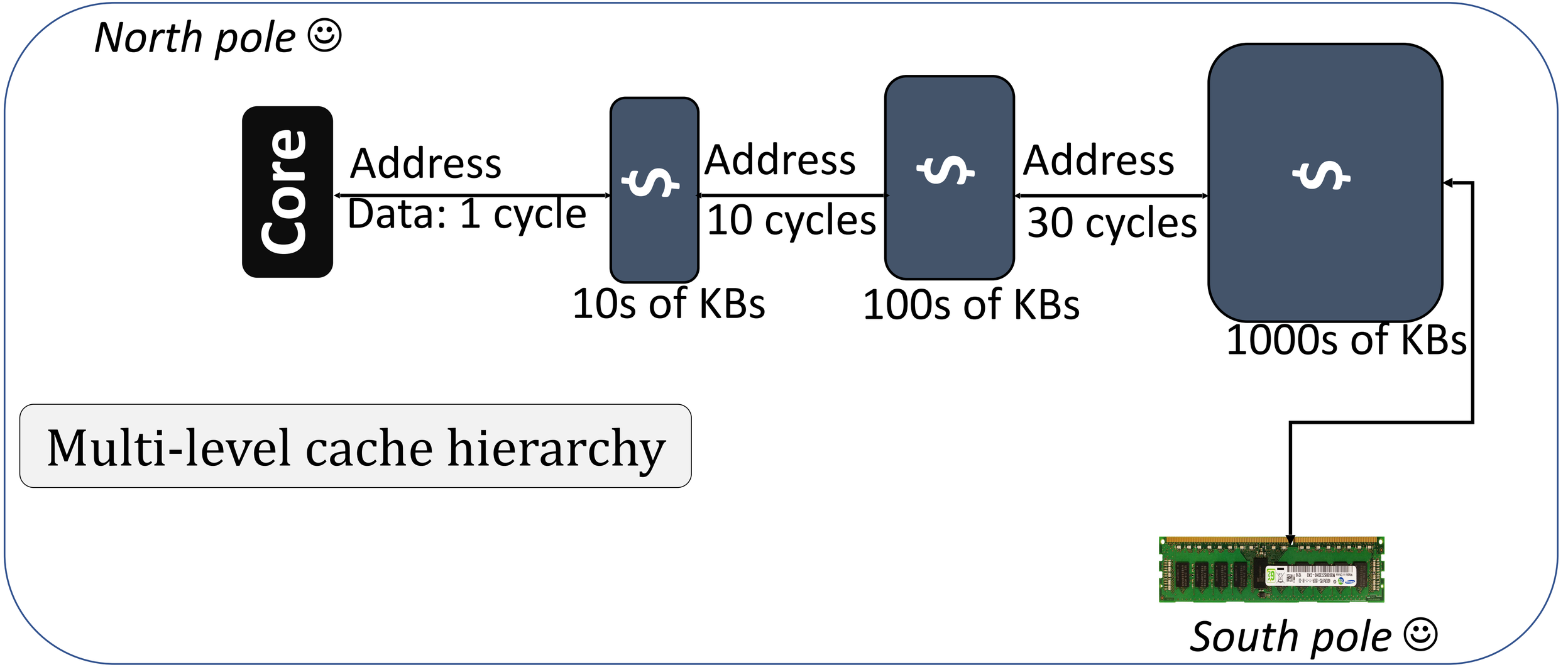


Costly DRAM accesses☹

32 to 64KB $ will be available in one to four cycles ☹

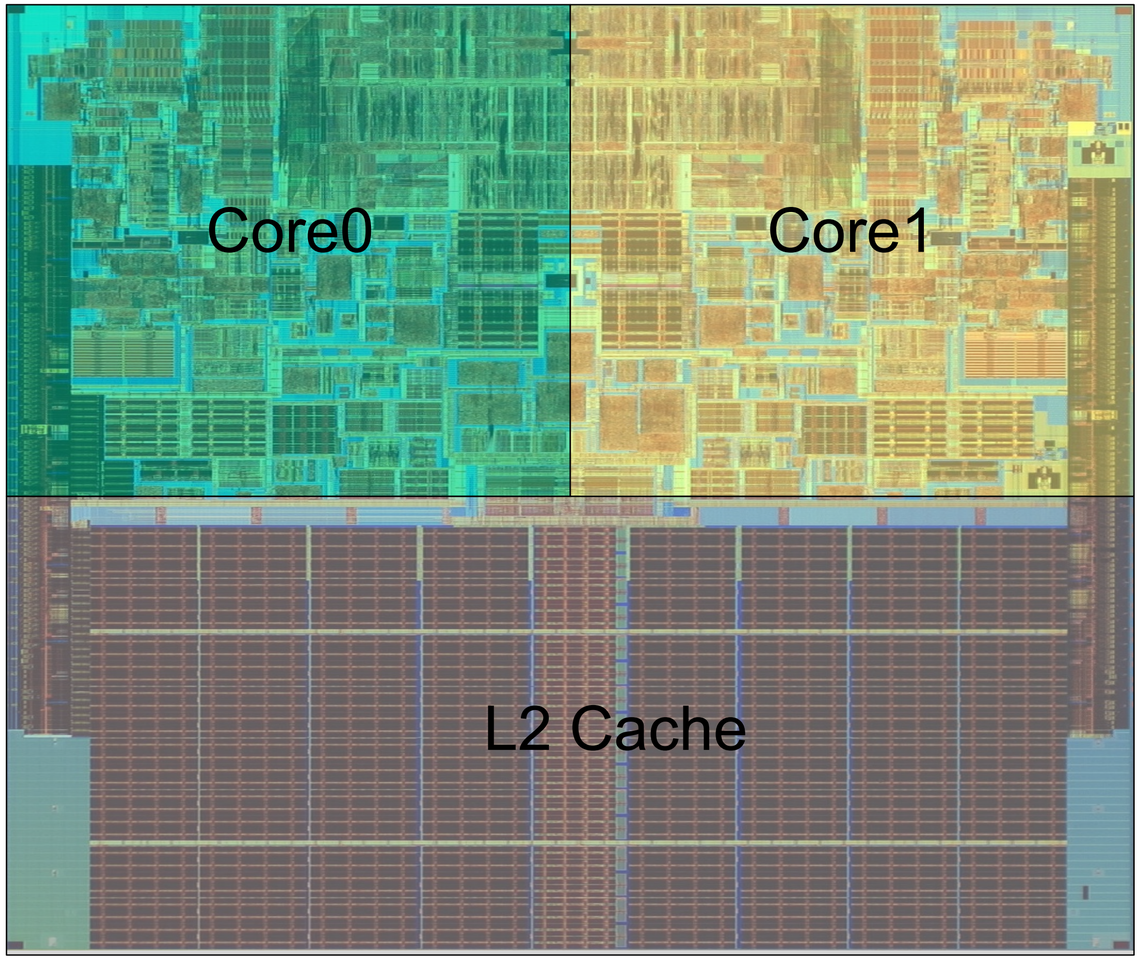
South pole ☺

Cache hierarchy with latency



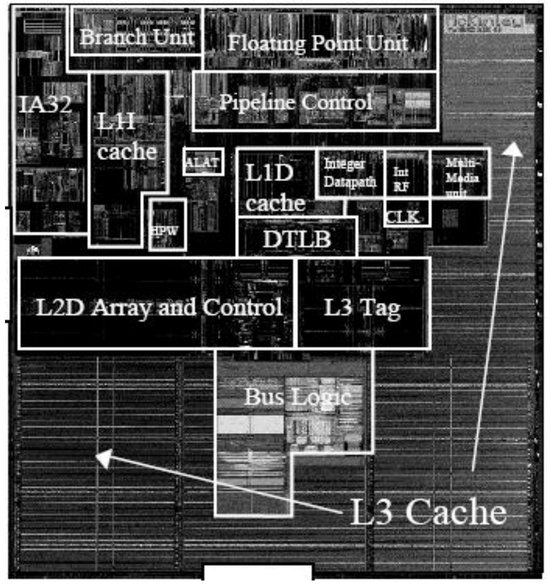
Case Study:Intel Core2 Duo

L1	32 KB, 8-Way, 64 Byte/ Line, LRU, WB 3 Cycle Latency
L2	4.0 MB, 16-Way, 64 Byte/Line, LRU, WB 14 Cycle Latency



Source: http://www.sandpile.org

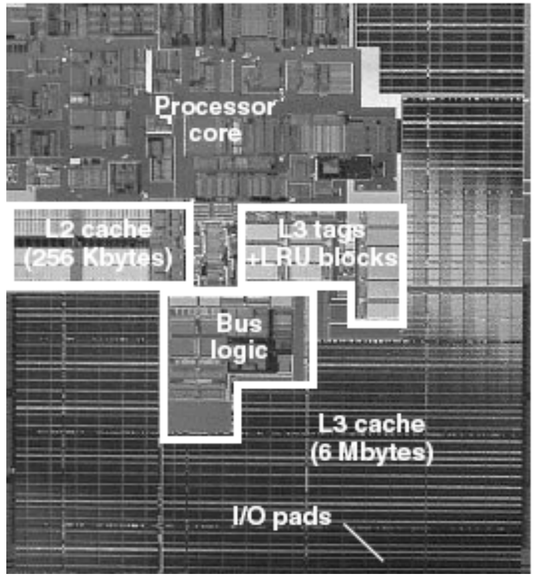
Case study: Intel Itanium 2



3MB
Version
180nm
421 mm^2

2002

6MB
Version
130nm
374 mm^2



Attribute	L1 instruction cache	L1 data cache	L2	L3
Size	16 Kbytes	16 Kbytes	256 Kbytes	6 Mbytes
Line size (bytes)	64	64	128	128
No. of ways	4	4	8	24
Replacement policy	Least recently used	Not recently used	Not recently used	Not recently used
Latency (no. of cycles)	1	Integer: 1 Floating-point: NA	Integer: 5 Floating-point: 6	14
Write policy	NA	Write through	Write back	Write back
Bandwidth (Gbytes/s)				
Read	48	24	48	48
Write	NA	24	48	48

Book

P & H, Chapter 4