Evolution of processor architectures: growing complexity of CPUs and its impact on the software landscape
*Lecture **1***

Basic concepts in computer architectures

Paweł Szostek

CERN

Inverted CERN School of Computing, 23-24 February 2015

Content of this presentation

I **will talk** about:

- high-level model of a computation and computer
- CPU-memory interaction
- Caches
- Pipeline
- Hazards

I will **not** talk about:

- OS-related stuff (e.g. virtual memory)
- faults
- data representation in memory
- high level languages' requirements for CPU architectures (e.g. stack)

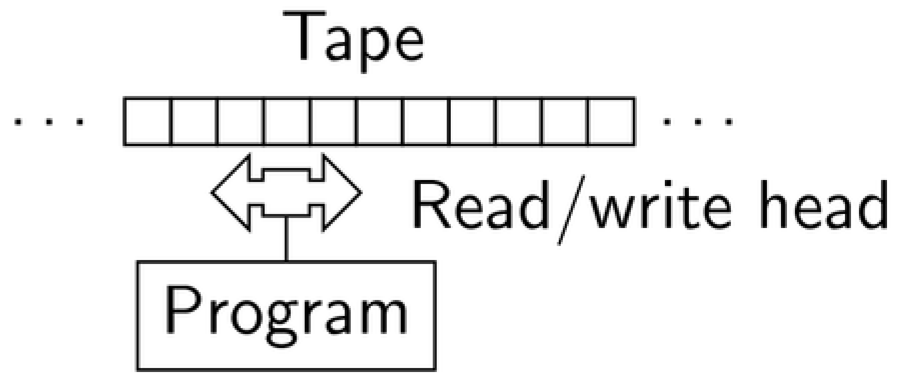
Please interrupt and ask questions at every point of the presentation!

- This talk **does not** aim to explain complete computer architecture!

CERN School of Computing

Turing machine

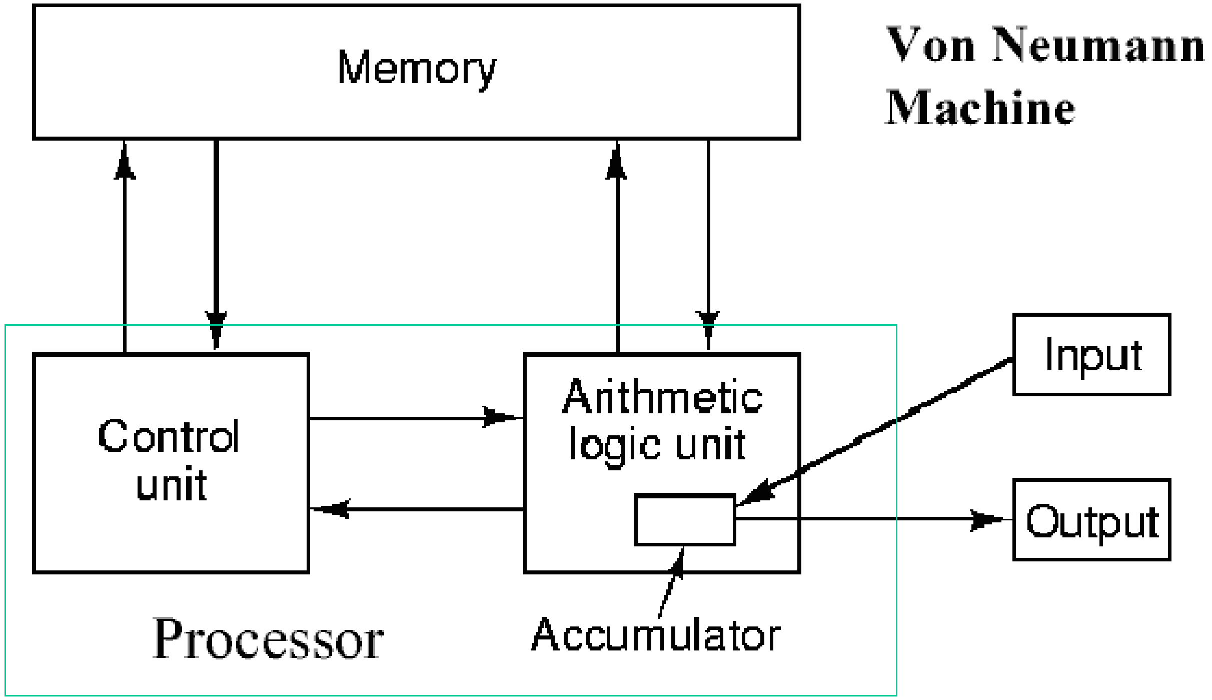
- a theoretical model of a computing apparatus
- it can read and write symbols to an infinite tape
- program is a state machine with outputs and transitions being a function of current state and input
- if a real machine can compute something, Turing machine can as well



source: decodescience.com

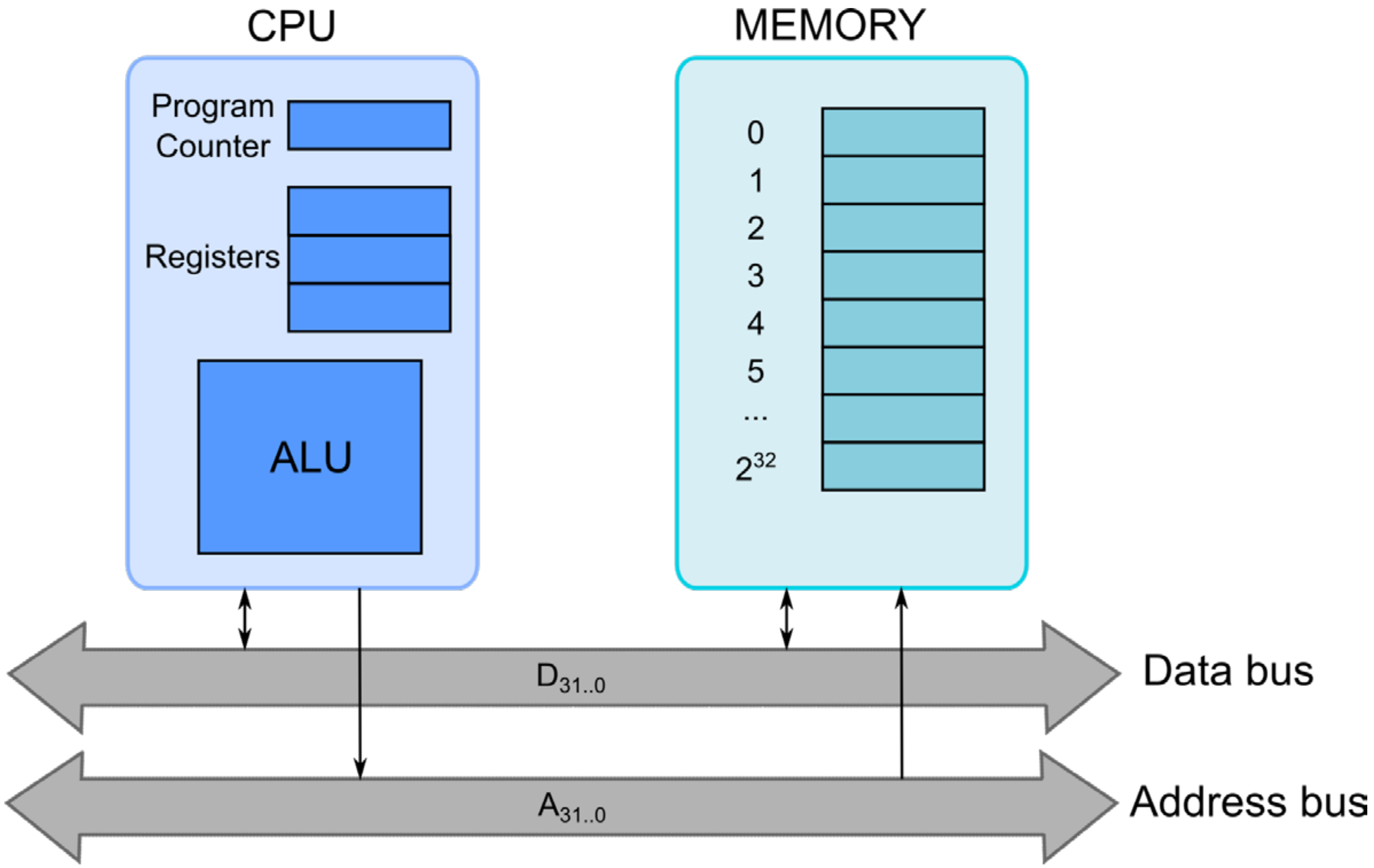
von Neumann machine

- realistic model of a processing machine from 1945…
- still valid in 2015



source: www.spring-alpha.org

Model of a simple computer



How many memory cells can be addressed with 32 bits?

Model of a simple computer (II)

- **What do we have inside?**
  - Program Counter – register that keeps track of the currently executed instruction
  - Registers – in-processor super-fast memory for keeping instruction operands and bookkeeping
  - ALU – Arithmetic-logical unit – building block able to perform arithmetic (e.g. add, divide, subtract, multiply) and logical operations (e.g. shifts, comparisons)
  - data and address buses, to ensure communication between CPU and the memory
  - large and slow main memory for keeping data and code

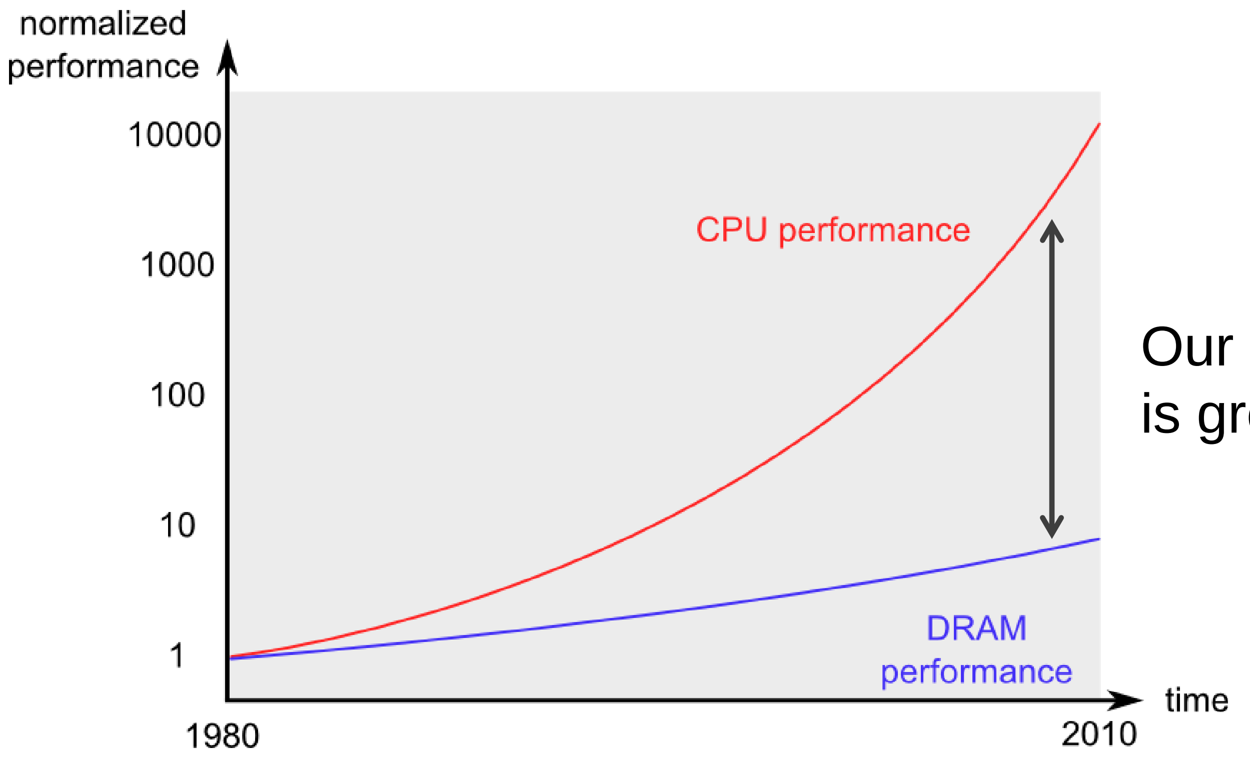
source: matryoshka.biz

Basic concepts in computer architectures: part 2

MEMORY SUBSYSTEM

What changed since 70's?

- Back in 1970s memory used to be faster than CPUs
- Nowadays, memory compared to CPUs is incredibly slow

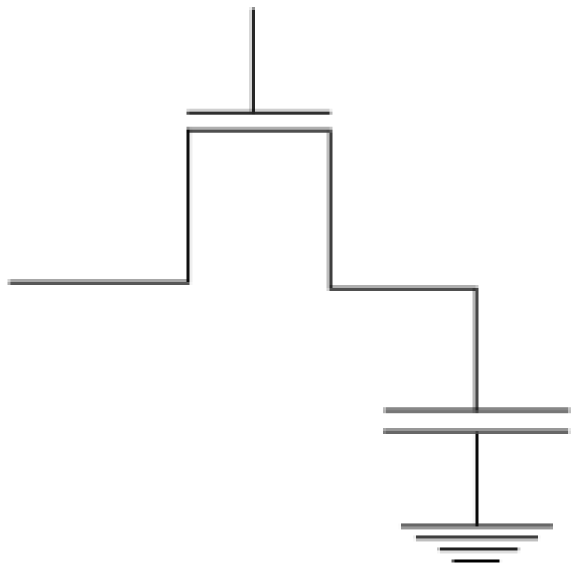


Our problem is growing

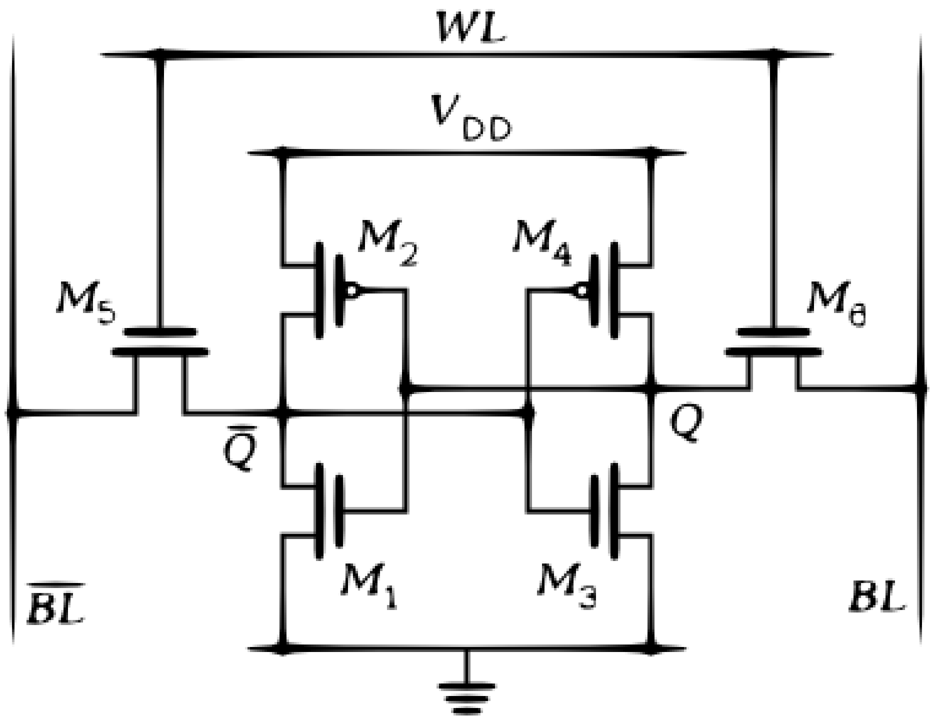
CERN School of Computing

Interlude: memory != memory

- There are different memory technologies



DRAM
(Dynamic RAM)



source: wikipedia.org

SRAM
(Static RAM)

Interlude: memory != memory

DRAM

- needs to be refreshed periodically (leaks!)
- charging and discharging a capacitor takes time
- DRAM cell output must be amplified
- space-efficient

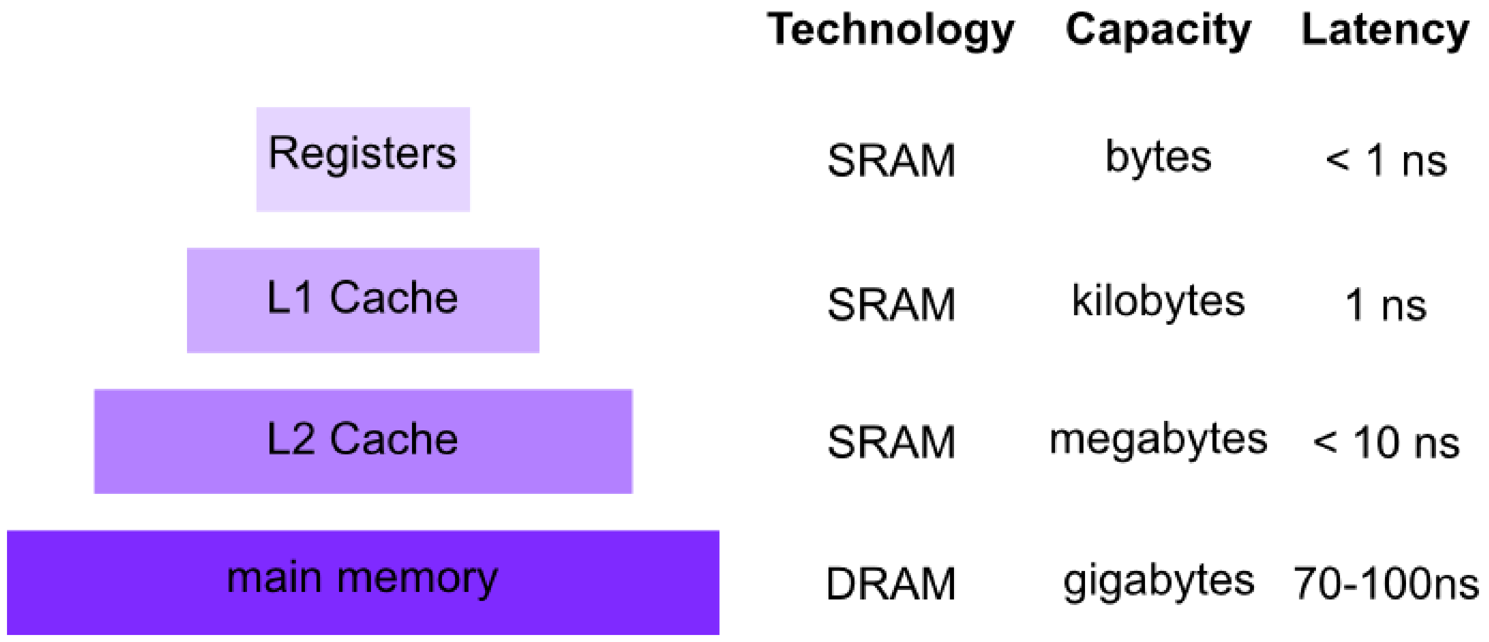
SRAM

- transistors actively drive output lines
- almost instantaneous state change
- more expensive than DRAM
- space-consuming

source: Jens Teubner "Data Processing on Modern Hardware"

Let’s speed up the memory

- Problem: fast memory is expensive
- Solution: introduce memory hierarchy, with a fast memory on the top and slow on the bottom

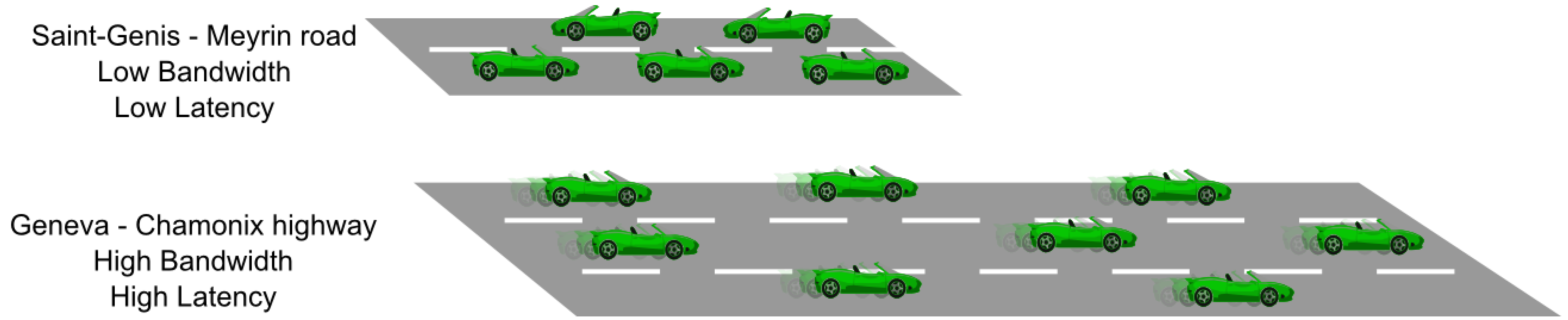


	Technology	Capacity	Latency
Registers	SRAM	bytes	< 1 ns
L1 Cache	SRAM	kilobytes	1 ns
L2 Cache	SRAM	megabytes	< 10 ns
main memory	DRAM	gigabytes	70-100ns

source: Jens Teubner “Data processing on modern hardware”

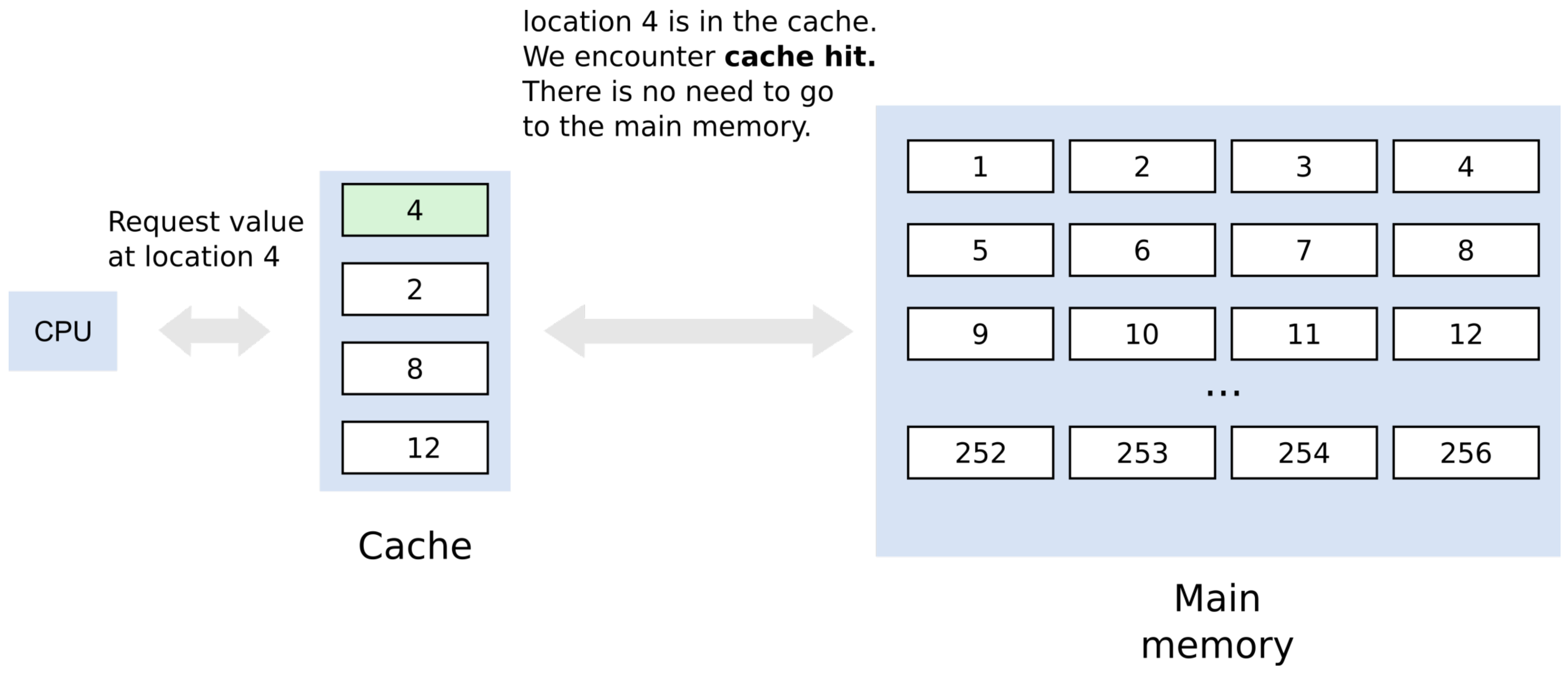
Interlude: latency vs. bandwidth

- latency and bandwidth are two orthogonal dimensions of performance
  - latency = how fast can we finish a job
  - bandwidth = how much can we push at a time



Can you think of a data link with incredibly high bandwidth, but huge latency?

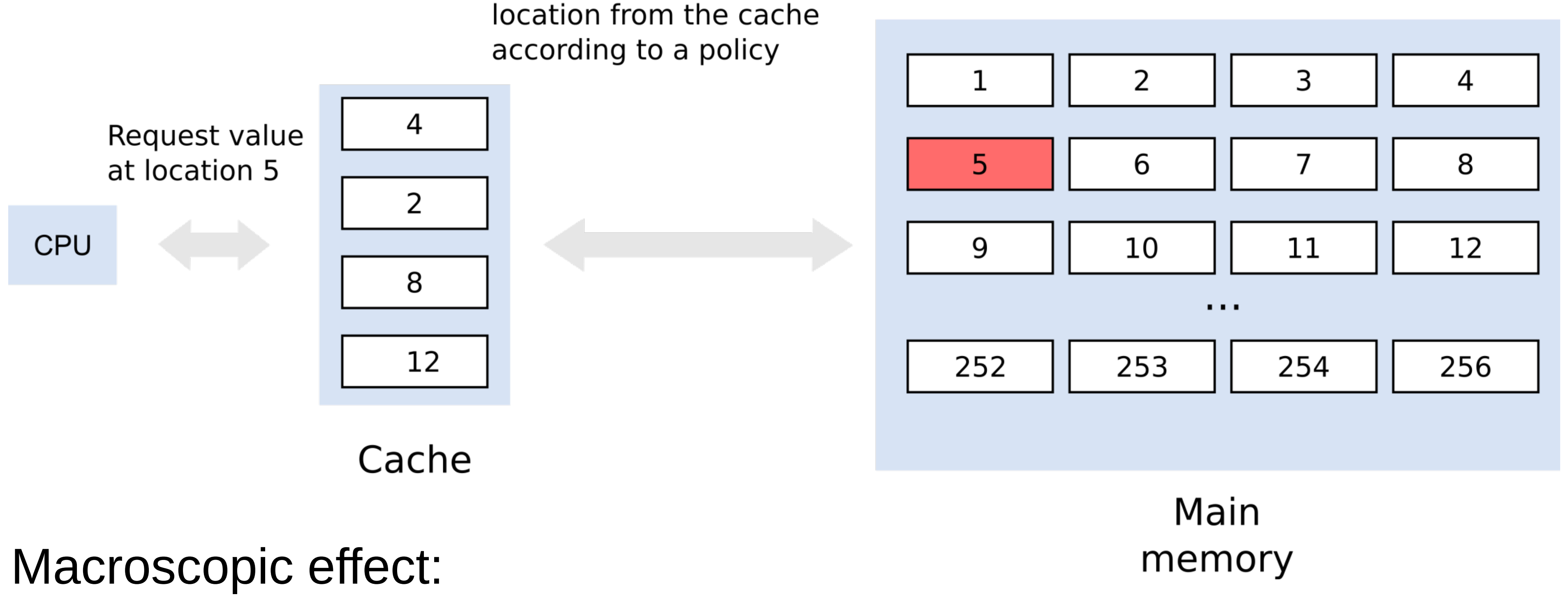
Interlude: cache hit and miss



source: Markus Pueschel “How to write fast numerical code”

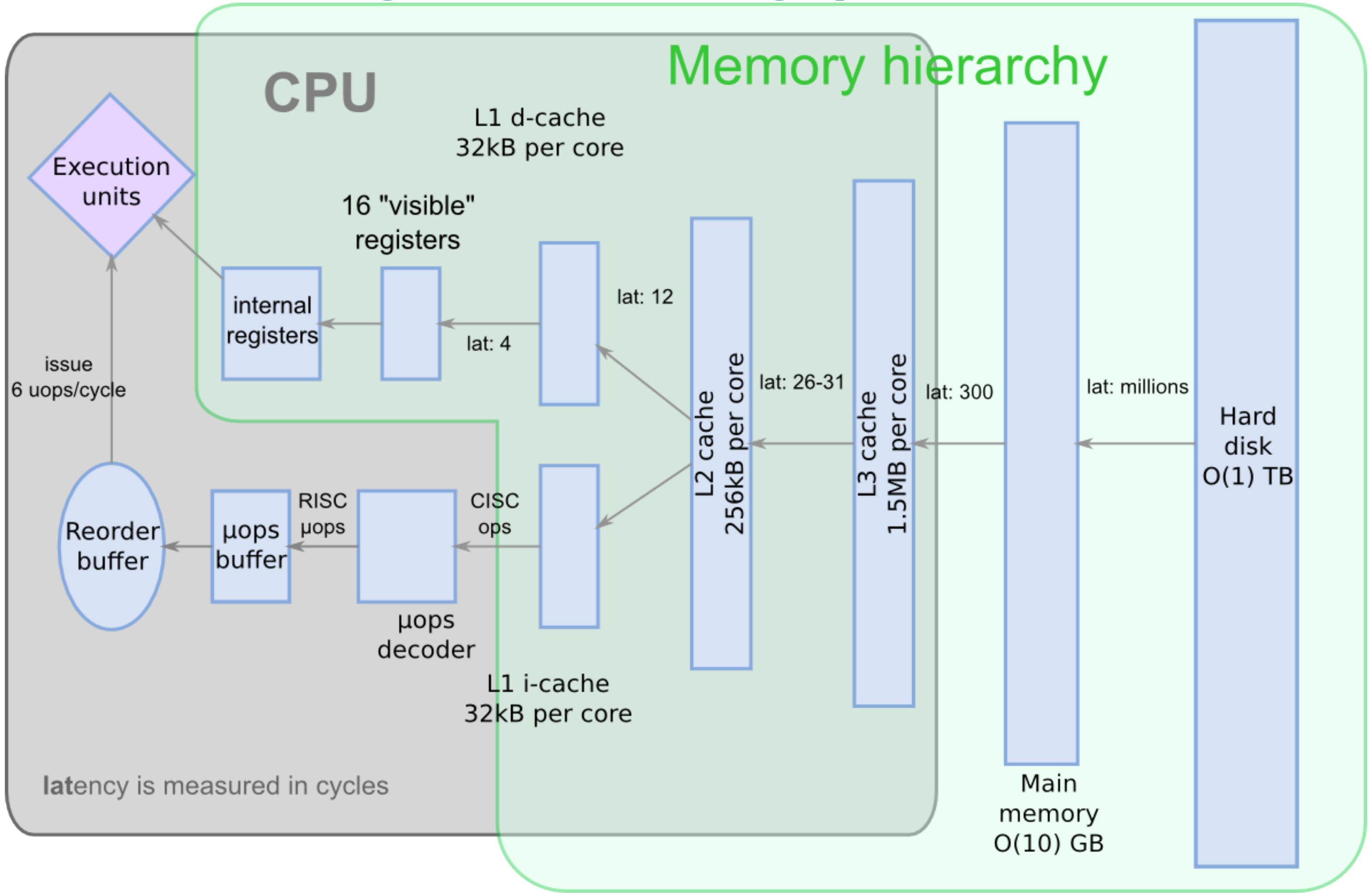
Interlude: cache hit and miss

Location 5 is not in the cache.
We encounter a **cache miss**.
There is a need to **fetch** location 5 from the main memory and **evict** another location from the cache according to a policy



Macroscopic effect:
your code goes terribly slow

Food for thought: the big picture



There is a divergence from the von Neumann model. Where?

source: Markus Pueschel "How to write fast numerical code"

source: ok3.org

Basic concepts in computer architectures: part 3

PIPELINE

Memory latency consequences

- Theoretical peak memory bandwidth: the maximum amount of data that can be read in a time unit.

$$bandwidth_{peak} = 2 \times channels \times bus\,width \times frequency$$

- therefore for a real memory we get:

$$bandwidth_{peak} = 2 \times 2 \times \frac{384}{8}(bytes) \times 3GHz = 288GB/s$$

  - 288GB is equivalent to 36G doubles

- NVIDIA Tesla Kepler K40 throughput = 1400GFLOPS (double)

- To achieve peak performance a program need to perform 1400/36 ≈ 39 operations on every double fetched from the main memory

How to improve utilization of the data fetched from RAM?

source: Vincenzo Innocente

CERN School of Computing

How does CPU see my code?

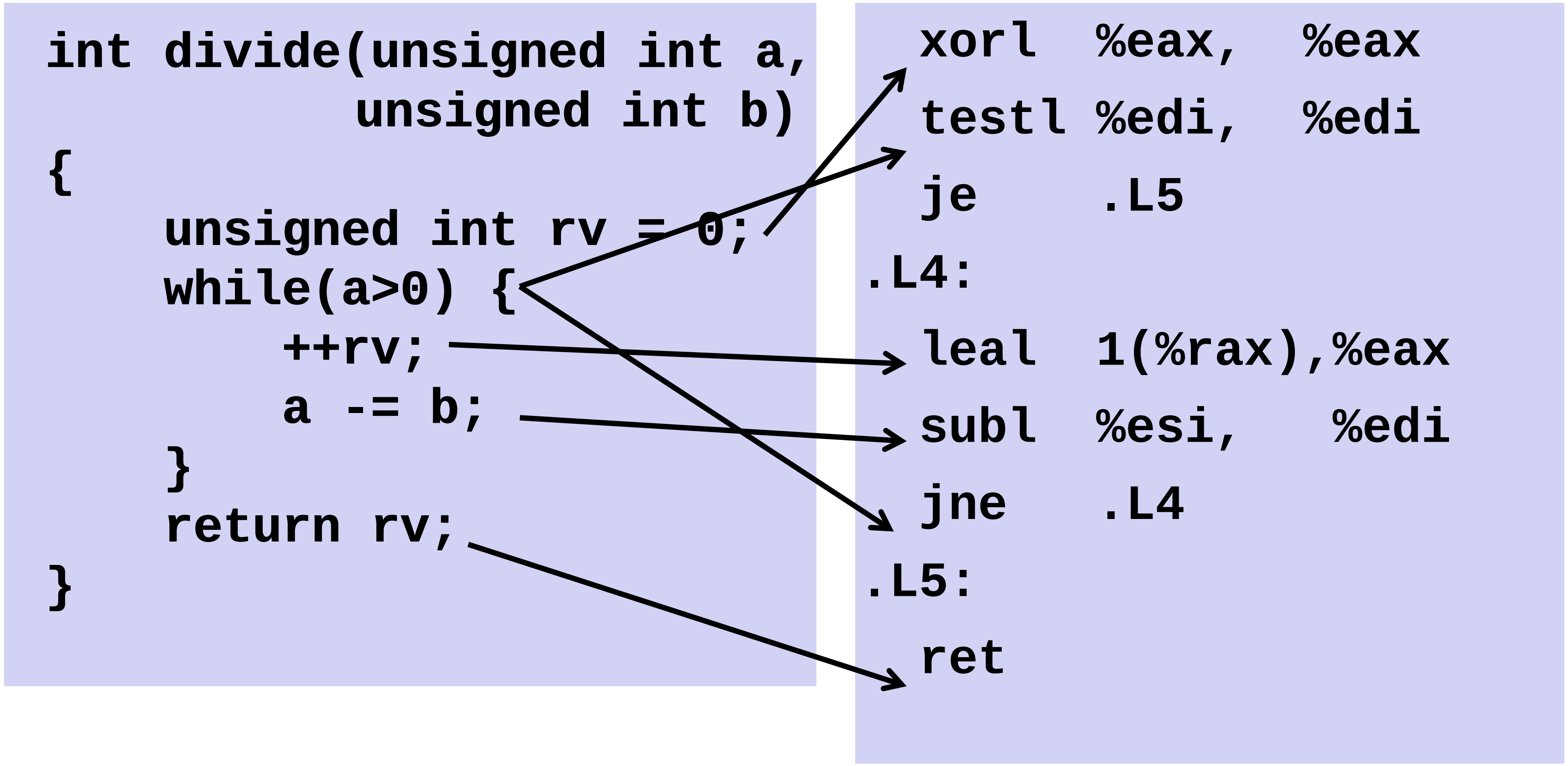
C code

```
int divide(unsigned int a,
           unsigned int b)
{
    unsigned int rv = 0;
    while(a>0) {
        ++rv;
        a -= b;
    }
    return rv;
}
```

Assembly code

```
  xorl  %eax,  %eax
  testl %edi,  %edi
  je    .L5
.L4:
  leal  1(%rax),%eax
  subl  %esi,   %edi
  jne   .L4
.L5:
  ret
```

How does CPU see my code? (II)



C code

```
int divide(unsigned int a,
           unsigned int b)
{
    unsigned int rv = 0;
    while(a>0) {
        ++rv;
        a -= b;
    }
    return rv;
}
```

Assembly code

```
  xorl  %eax,   %eax
  testl %edi,   %edi
  je    .L5
.L4:
  leal  1(%rax),%eax
  subl  %esi,   %edi
  jne   .L4
.L5:
  ret
```

Interlude: RISC vs. CISC

- CISC = Complex Instruction Set Computing
  RISC = Reduced Instruction Set Computing
- Two major directions in CPU design
- To see how it works let's **multiply two numbers** from the memory

CISC

```
mul $(0x123), $(0x456)
```

RISC

```
load RA, $(0x123)
load RB, $(0x456)
mul RA, RB
store $(0x123), RA
```

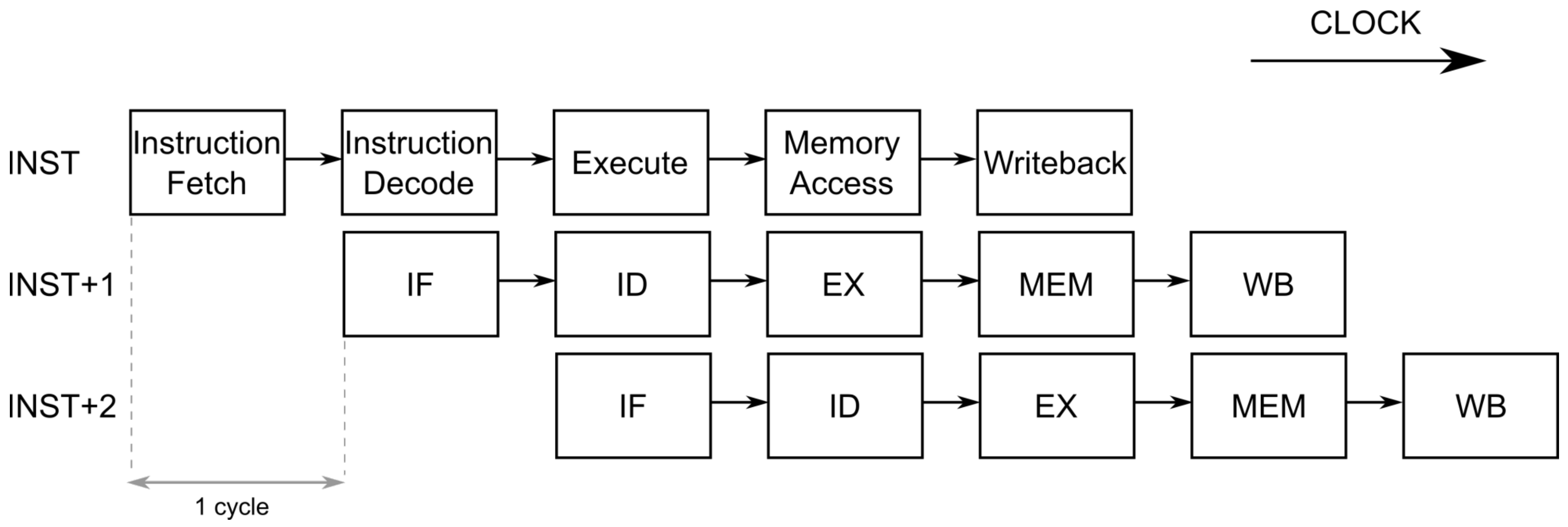
CERN School of Computing

Interlude: RISC vs. CISC (II)

CISC	RISC
Emphasis on hardware	Emphasis on software
Multi-clock complex instructions translated to simpler operations	Single-clock reduced instructions executed directly*
Memory-to-memory operations	Register-to-register operations <u>only</u>
Smaller machine code size	Larger machine code size
Any instruction can reference memory	Only LOAD and STORE references memory
Applications: desktop and server machines	Applications: low power designs (cell phones, tablets)

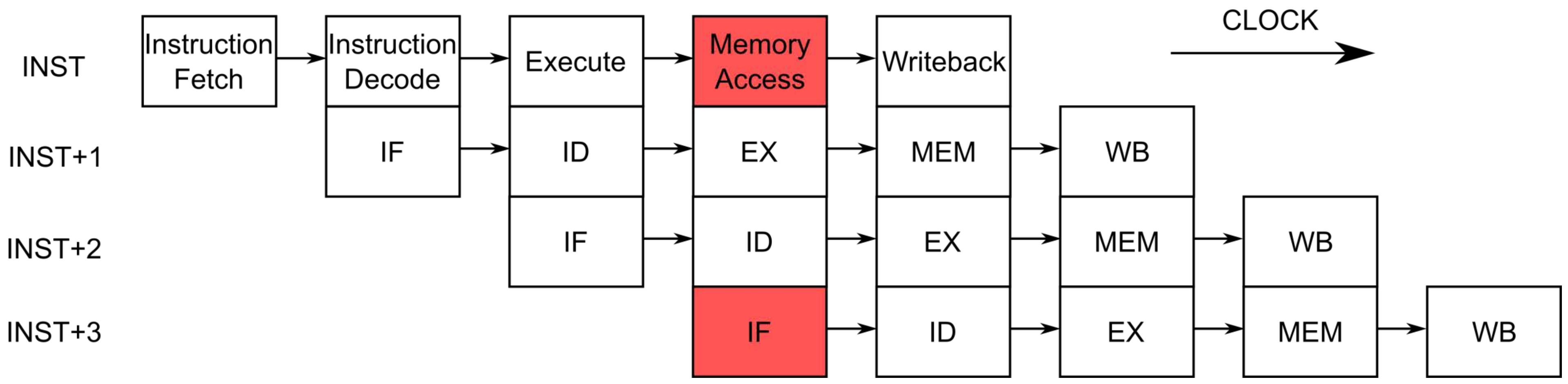
Pipelining

- Execution of a single instruction takes a couple of cycles (5 on the drawing below) and is split into simpler operations
- **Pipelining** is a technique for instruction-level parallelism. In a perfect case *n*-stages pipeline improves throughput by factor of *n*.
- Pipeline performance can be hindered by an “unfortunate” instruction flow called **hazard (structural, data, control)**



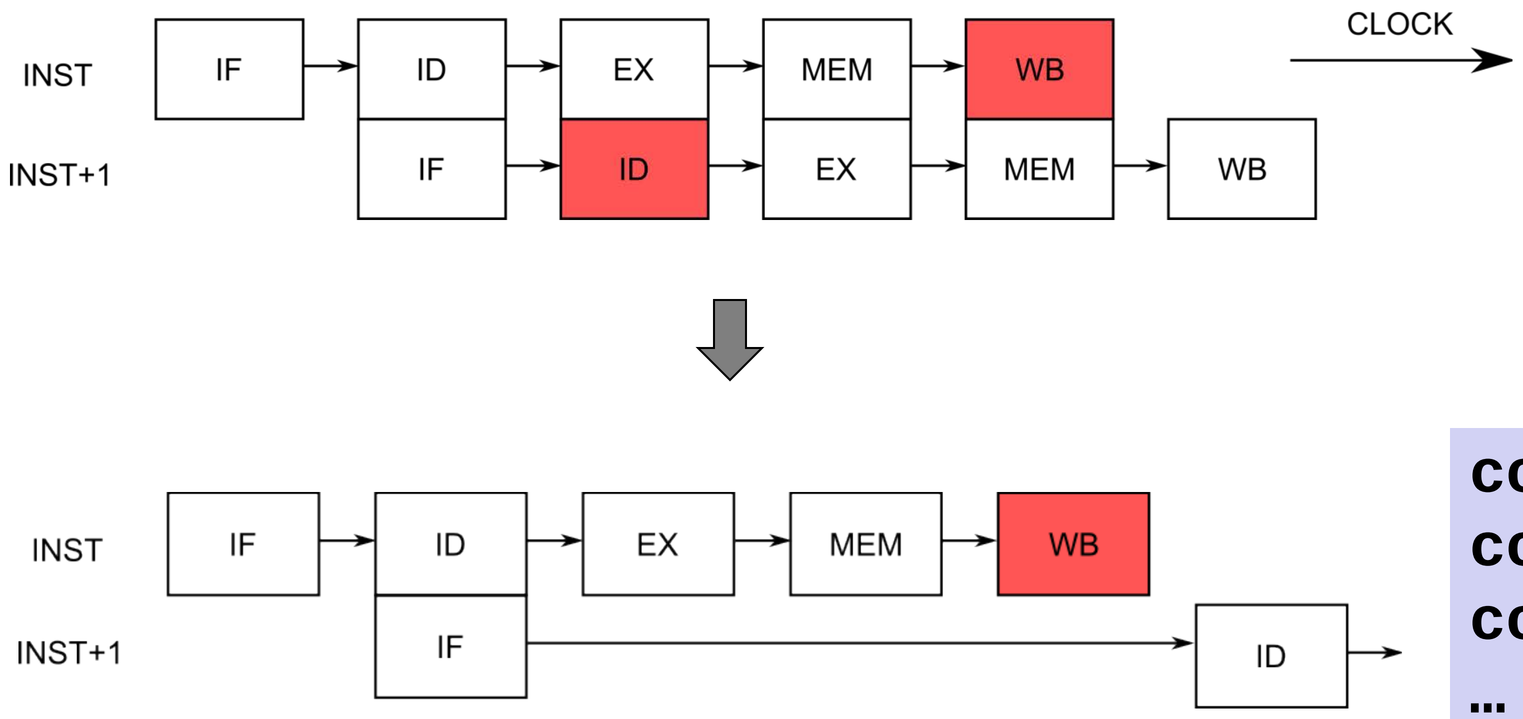
Hazard #1: Structural Hazard

Structural hazard occurs when the same hardware resources are needed by different instructions in the same time.
Example: memory write and **instruction fetch** executed by the same hardware.



Hazard #2: Data Hazard

Data hazards occur when result of an instruction is not ready when it is accessed by a later instruction.

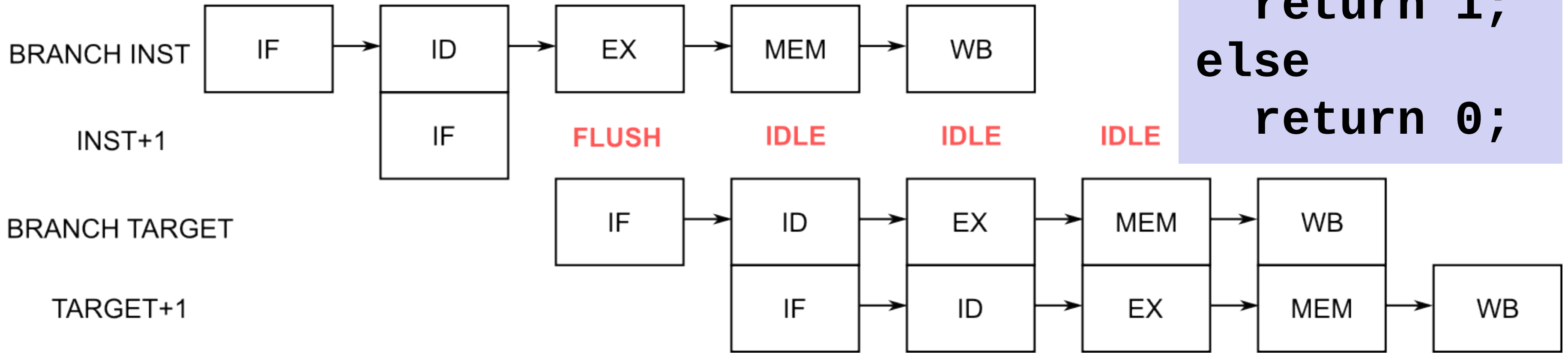


```
counter+=1;
counter+=1;
counter+=1;
…
```

Hazard #3: Control Hazard

- **Pipeline cannot be filled before it is known which path to follow**
  - in modern CPUs, a path is executed speculatively by predicting if a branch will be taken

```
a=b+c;
if (a%2)
  return 1;
else
  return 0;
```



Can you think of a way to optimize this code?

Speculative execution

- **Problem**: branch can be taken or not
- **Solution**: **execute speculatively**
  - guess branch target → start executing at guessed position
  - execute branch → verify guess afterwards
- Branch prediction: guessing the branch
  - one of the non-trivial problems in computer architectures
  - dedicated hardware to keep track of branch targets,
  - if the guess is correct, there is no **penalty** at all
  - if the guess is incorrect, CPU needs to **flush** the pipeline

Execution hazards - example

- An array of 10’000 integers from a uniform [0,9999] distribution
- We change condition selectivity by changing threshold
- Straight way to a **control hazard**
- **IF** statement prevents vectorization

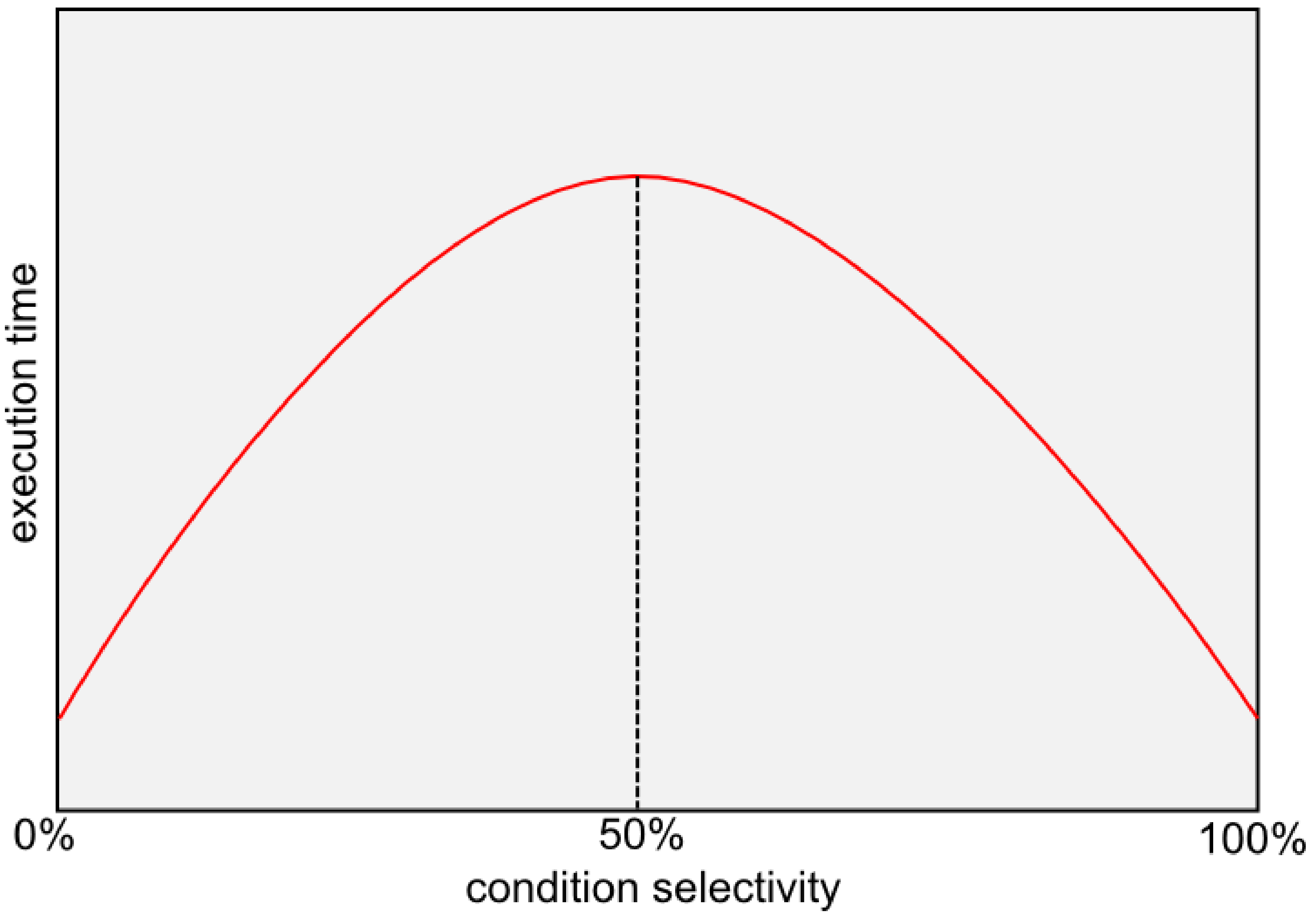
```
int array[10000];
const int THRESHOLD = 5000;
int count = 0;

for(int i=0; i<10000; ++i)
  array = (int)(random(0,1)*10000);
for(int i=0; i<10000; ++i)
  if(array[i] > THRESHOLD)
    ++count;
```

$\}$ Initialization will be skipped on next slid

source: Jens Teubner “Data processing on Modern Hardware”

Execution hazards – example (II)



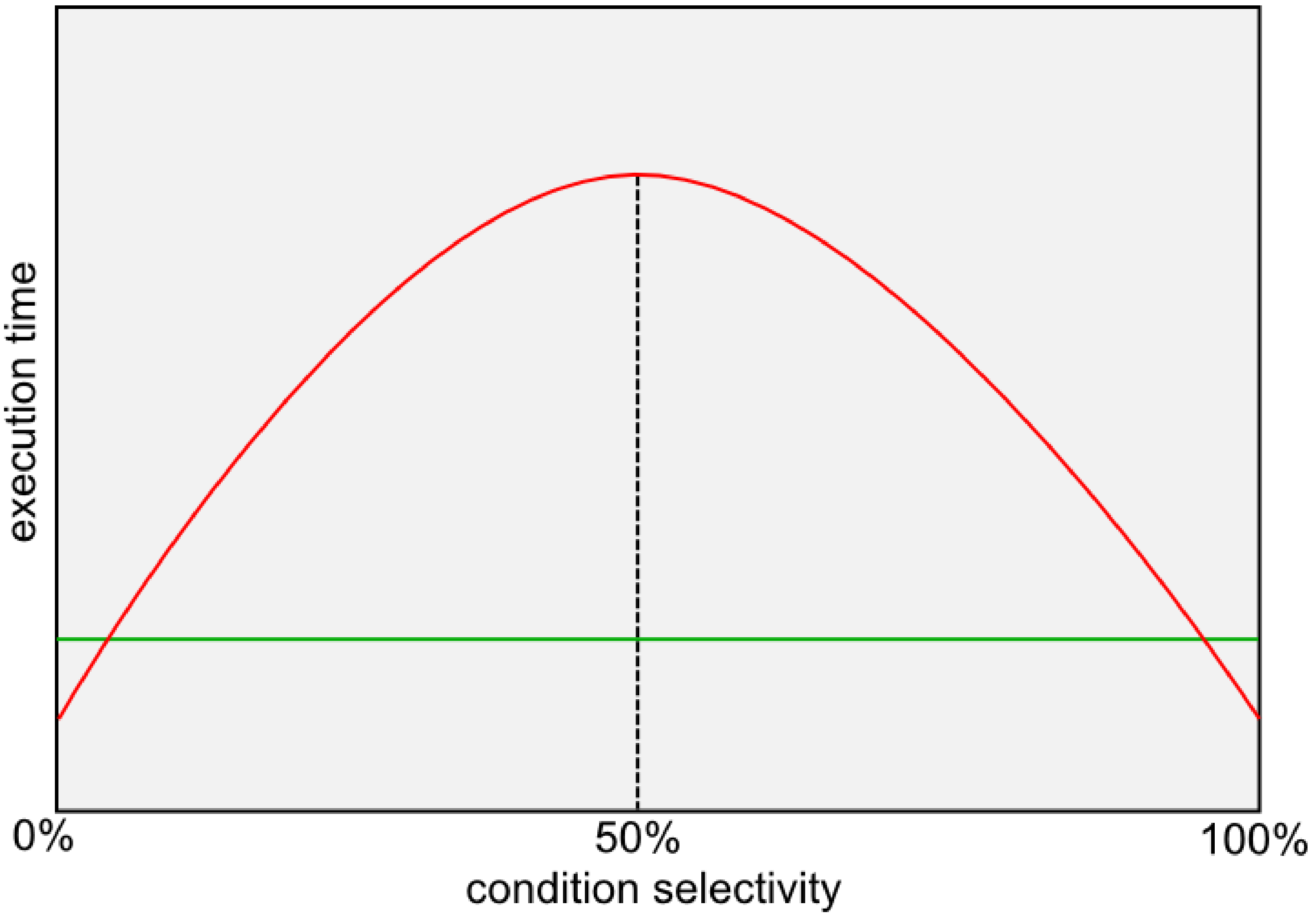
Execution hazards – example (III)

- we change control flow into data flow
- cost: we need to do an addition at every iteration
- gains:
  - no more mispredicted branches,
  - the loop can be vectorized

```
for(int i=0; i<10000; ++i)
  if(array[i] > THRESHOLD)
     ++count;
```

```
for(int i=0; i<10000; ++i)
  count += (array[i] > THRESHOLD);
```

Execution hazards – example(IV)



Execution hazards – example (V)

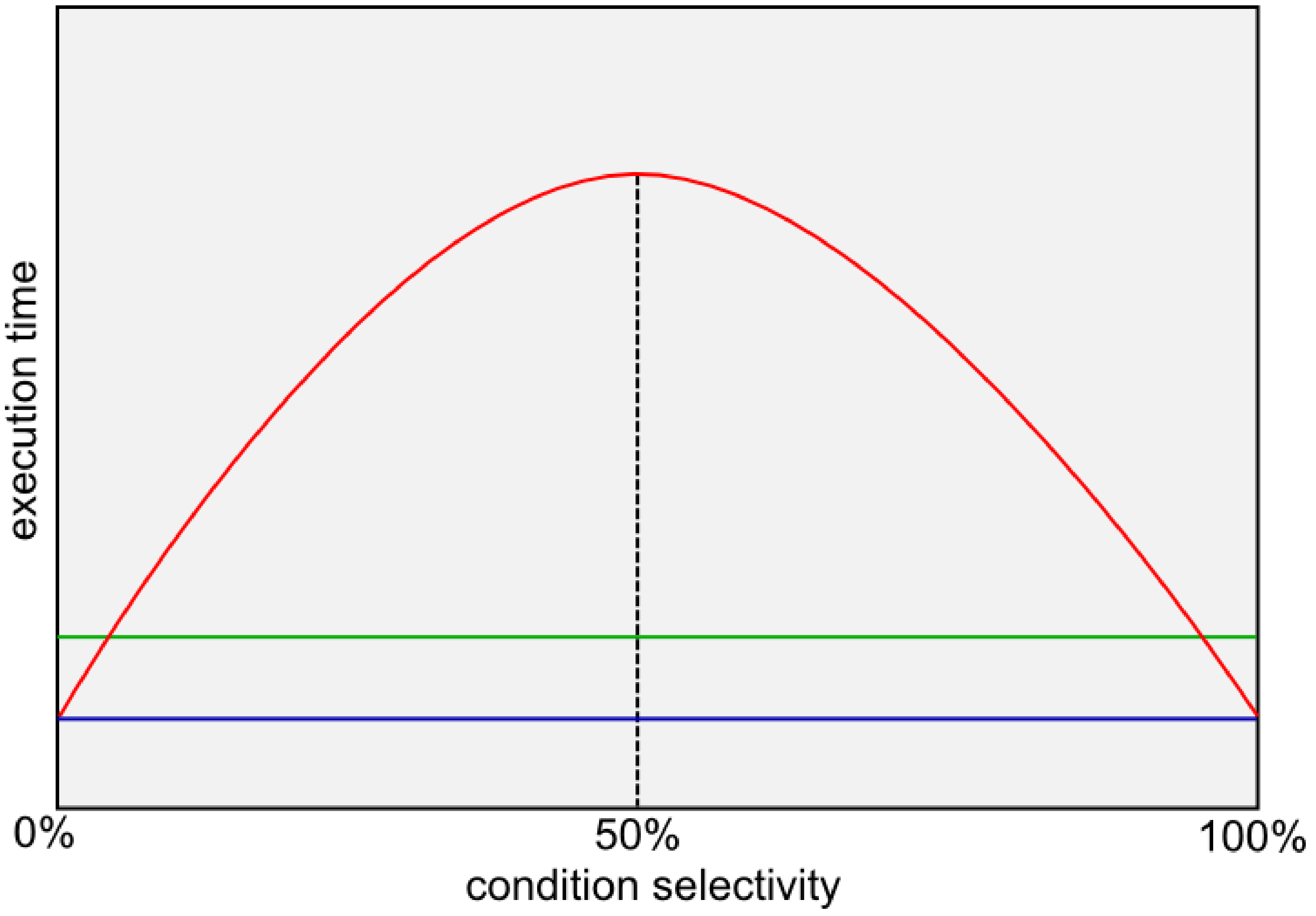
- There is still a data hazard
  - In each iteration we need the value of **count** from the previous iteration
  - We can divide the array in two parts and interleave addition to two counters

```
for(int i=0; i<10000; ++i)
  count += (array[i] > THRESHOLD);
```

$\Downarrow$

```
for(int i=0; i<5000; ++i) {
  count1 += (array[i] > THRESHOLD);
  count2 += (array[i+5000] > THRESHOLD);
}
count1 += count2;
```

Execution hazards – example (VI)



Takeaway messages

1. We stick to a model which is 70 years old.
2. Main blocking factor in heavy computation is memory. Caches speed-up things, but as long as the problem fits into them.
3. Big classes are not a problem of space – they are problem of speed.
4. Complex instructions are translated to simpler ones.
5. “Unfortunate” instruction stream can inhibit performance.
6. Most readable code is probably **not the most efficient.**