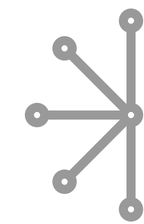
Technische Universität Braunschweig



Institut für Betriebssysteme und Rechnerverbund
Algorithmik

Algorithm Engineering
Lecture 2: CPU Architecture 1

Why Hardware Matters

Why Hardware Matters

Why should **you** care?

Why Hardware Matters

Why should **you** care?

- Constant factors matter in practice (even well below the $60000\times$ speedup)

Technische Universität Braunschweig

Why Hardware Matters

Why should **you** care?

- Constant factors matter in practice (even well below the $60000\times$ speedup)
- **Sometimes:** enormous constant factors hidden in O-Notation

Why Hardware Matters

Why should **you** care?

- Constant factors matter in practice (even well below the $60000\times$ speedup)
- **Sometimes:** enormous constant factors hidden in O-Notation
- **Sometimes:** enormous constant factor from misunderstanding modern CPUs

Why Hardware Matters

Why should **you** care?

- Constant factors matter in practice (even well below the $60000\times$ speedup)
- **Sometimes:** enormous constant factors hidden in O-Notation
- **Sometimes:** enormous constant factor from misunderstanding modern CPUs
- A greedy python heuristic could be slower than a well-optimized exact C++ algorithm (at least on the instances that you/your boss/your client are interested in)

Why Hardware Matters

Why should **you** care?

- Constant factors matter in practice (even well below the $60000\times$ speedup)
- **Sometimes:** enormous constant factors hidden in O-Notation
- **Sometimes:** enormous constant factor from misunderstanding modern CPUs
- A greedy python heuristic could be slower than a well-optimized exact C++ algorithm (at least on the instances that you/your boss/your client are interested in)

Here: No vendor-/hardware-specific details, but a basic (generalizable) understanding.

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n) {
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n)
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

→

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n) {
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n)
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n)
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n)
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n)
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```cpp
std::int64_t sum(const std::int64_t *x, std::size_t n)
    std::int64_t res = 0;
    for(std::size_t i = 0; i < n; ++i) {
        res += x[i];
    }
    return res;
}
```

```asm
# sum(x in rdi, n in rsi):
        xor     eax, eax
        test    rsi, rsi
        je      .loop_end
        xor     ecx, ecx
.loop_begin:
        add     rax, qword ptr [rdi + 8*rcx]
        inc     rcx
        cmp     rsi, rcx
        jne     .loop_begin
.loop_end:
        ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```cpp
1  std::int64_t sum(const std::int64_t *x, std::size_t n) {
2      std::int64_t res = 0;
3      for(std::size_t i = 0; i < n; ++i) {
4          res += x[i];
5      }
6      return res;
7  }
```

```asm
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```
1  std::int64_t sum(const std::int64_t *x, std::size_t n)
2      std::int64_t res = 0;
3      for(std::size_t i = 0; i < n; ++i) {
4          res += x[i];
5      }
6      return res;
7  }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
→10        jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```
1  std::int64_t sum(const std::int64_t *x, std::size_t n)
2      std::int64_t res = 0;
3      for(std::size_t i = 0; i < n; ++i) {
4          res += x[i];
5      }
6      return res;
7  }
```

→

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```cpp
1  std::int64_t sum(const std::int64_t *x, std::size_t n) {
2      std::int64_t res = 0;
3      for(std::size_t i = 0; i < n; ++i) {
4          res += x[i];
5      }
6      return res;
7  }
```

```asm
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```
1  std::int64_t sum(const std::int64_t *x, std::size_t n)
2      std::int64_t res = 0;
3      for(std::size_t i = 0; i < n; ++i) {
4          res += x[i];
5      }
6      return res;
7  }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

How does a CPU execute your code?

```cpp
1 std::int64_t sum(const std::int64_t *x, std::size_t n)
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```asm
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n)
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

How does a CPU execute your code?

```
1  std::int64_t sum(const std::int64_t *x, std::size_t n)
2      std::int64_t res = 0;
3      for(std::size_t i = 0; i < n; ++i) {
4          res += x[i];
5      }
6      return res;
7  }
```

```
1  # sum(x in rdi, n in rsi):
2          xor     eax, eax
3          test    rsi, rsi
4          je      .loop_end
5          xor     ecx, ecx
6  .loop_begin:
7          add     rax, qword ptr [rdi + 8*rcx]
8          inc     rcx
9          cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
12         ret
```

Architectural State: Memory content, register content, instruction pointer, flags…

Architectural Model: Correct but also Wrong

How does a CPU execute your code?

```
1 std::int64_t sum(const std::int64_t *x, std::size_t n) {
2     std::int64_t res = 0;
3     for(std::size_t i = 0; i < n; ++i) {
4         res += x[i];
5     }
6     return res;
7 }
```

```
 1 # sum(x in rdi, n in rsi):
 2         xor     eax, eax
 3         test    rsi, rsi
 4         je      .loop_end
 5         xor     ecx, ecx
 6 .loop_begin:
 7         add     rax, qword ptr [rdi + 8*rcx]
 8         inc     rcx
 9         cmp     rsi, rcx
10         jne     .loop_begin
11 .loop_end:
→12        ret
```

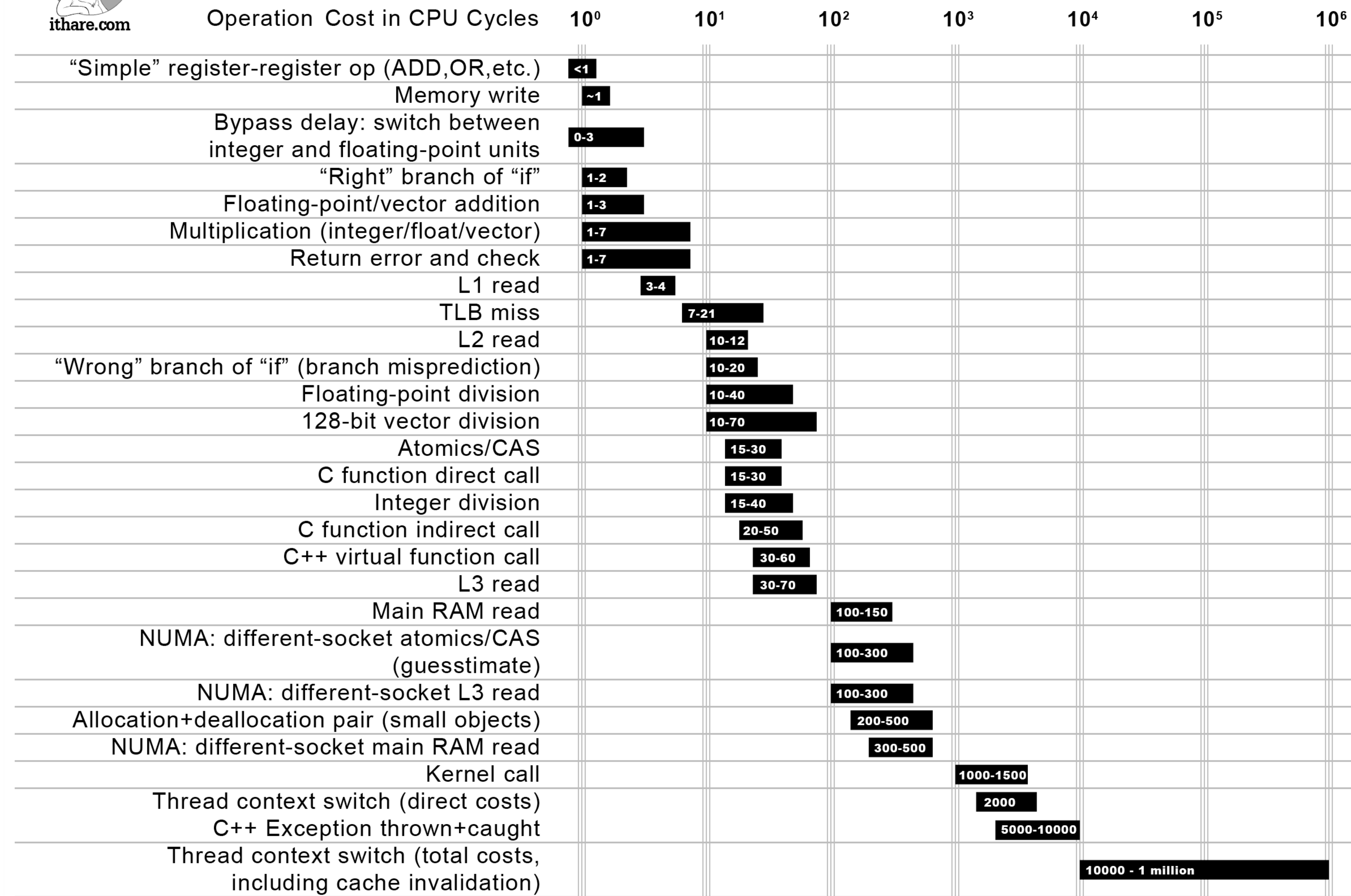
Architectural State: Memory content, register content, instruction pointer, flags…

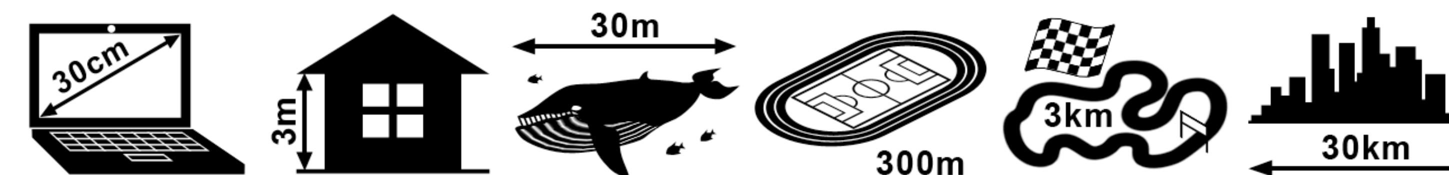
Compiler Explorer: `https://godbolt.org/`

How does a CPU

```
1  std::int64_t sum(const
2      std::int64_t res =
3      for(std::size_t i =
4          res += x[i];
5      }
6      return res;
7  }
```

```
tr [rdi + 8*rcx]
```





Architectural Sta… on pointer, flags…

Compiler Explorer: `https://godbolt.org/`

Pipeline Model

Pipeline Model

Like an assembly line in a factory:

Pipeline Model

Like an assembly line in a factory:

- New instructions can be fetched while others are decoded/executed/written back

Pipeline Model

Like an assembly line in a factory:

- New instructions can be fetched while others are decoded/executed/written back
- Each stage of the pipeline handles a different instruction simultaneously

Pipeline Model

Like an assembly line in a factory:

- New instructions can be fetched while others are decoded/executed/written back
- Each stage of the pipeline handles a different instruction simultaneously
- **General idea:** avoid waiting as much as possible!

Pipeline Model

Like an assembly line in a factory:

- New instructions can be fetched while others are decoded/executed/written back
- Each stage of the pipeline handles a different instruction simultaneously
- **General idea:** avoid waiting as much as possible!
- Example with a 5-stage pipeline (stages: Fetch, Decode, Execute, Memory, Write Back)

Pipeline Model

Like an assembly line in a factory:

- New instructions can be fetched while others are decoded/executed/written back
- Each stage of the pipeline handles a different instruction simultaneously
- **General idea:** avoid waiting as much as possible!
- Example with a 5-stage pipeline (stages: Fetch, Decode, Execute, Memory, Write Back)

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
Instruction x	IF	ID	EXE	MEM	WB				
Instruction x+1		IF	ID	EXE	MEM	WB			
Instruction x+2			IF	ID	EXE	MEM	WB		
Instruction x+3				IF	ID	EXE	MEM	WB	
Instruction x+4					IF	ID	EXE	MEM	WB

In real CPUs

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

Modern CPUs take this to the extreme:

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

Modern CPUs take this to the extreme:

- **Superscalar:** Independent instructions can run simultaneously if resources permit

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

Modern CPUs take this to the extreme:

- **Superscalar:** Independent instructions can run simultaneously if resources permit
- **More stages:** 10-25+ finer-grained stages

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

Modern CPUs take this to the extreme:

- **Superscalar:** Independent instructions can run simultaneously if resources permit
- **More stages:** 10-25+ finer-grained stages
- **Out-of-Order:** Later independent instructions can ‘overtake' earlier instructions; *retirement* happens in order.

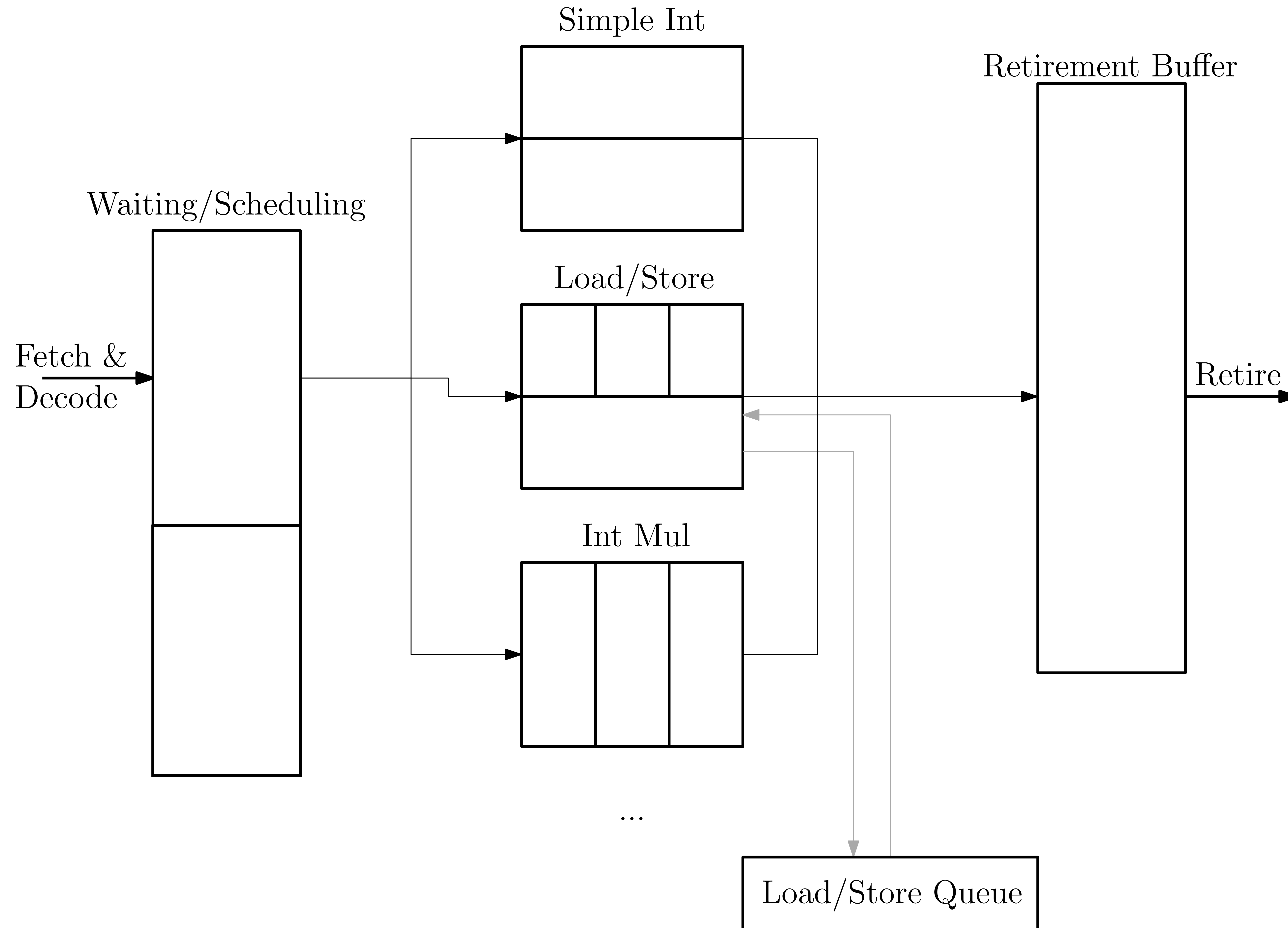
In real CPUs

As-if rule (c ... of
the program, ...

Essentially, a ... ution.

Modern CP...

- **Superscal**...
- **More stag**...
- **Out-of-Or**...
  retirement



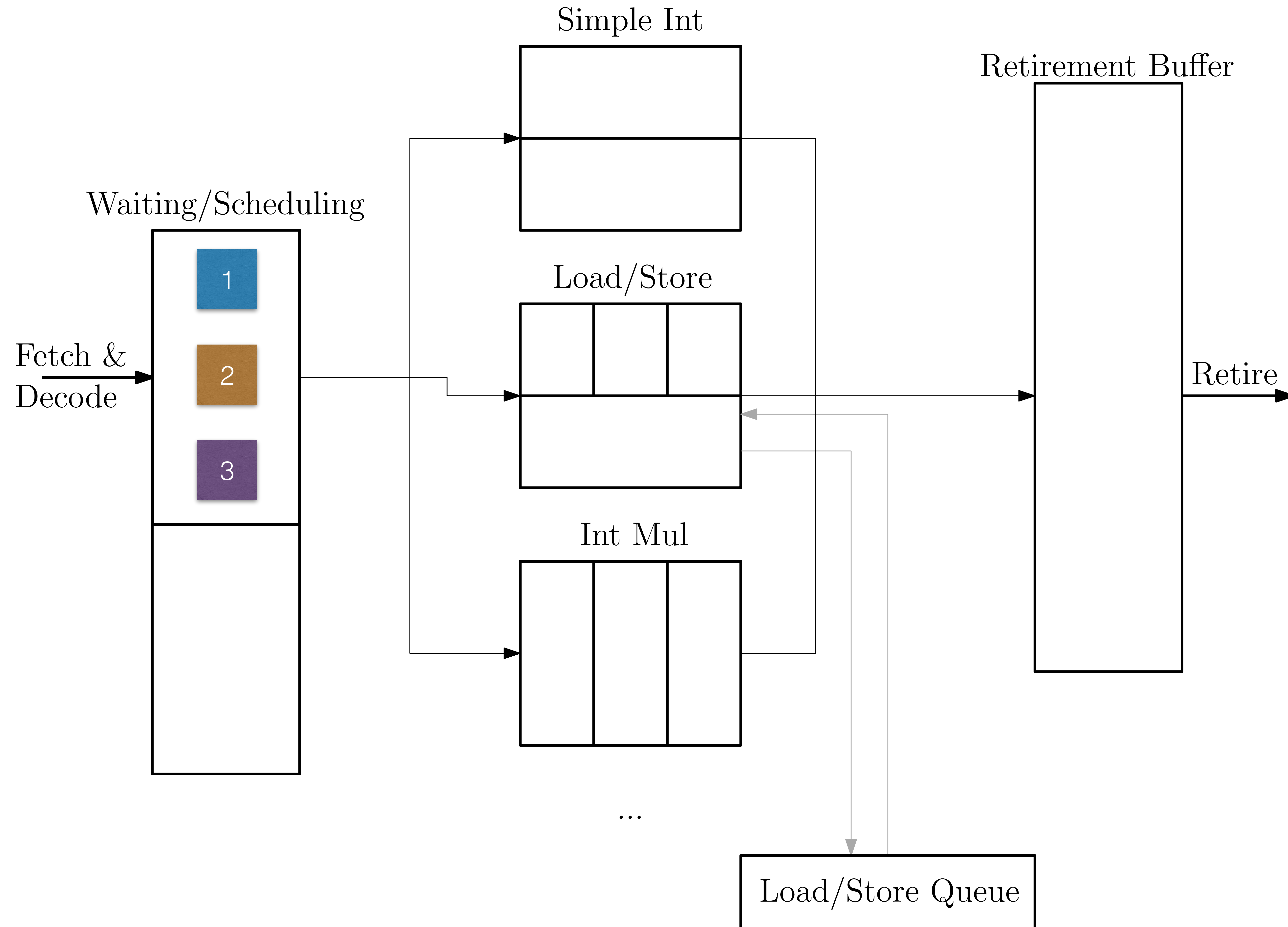
In real CPUs

As-if rule (c [...] of
the program,

Essentially, a [...] ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Ord**
  retirement

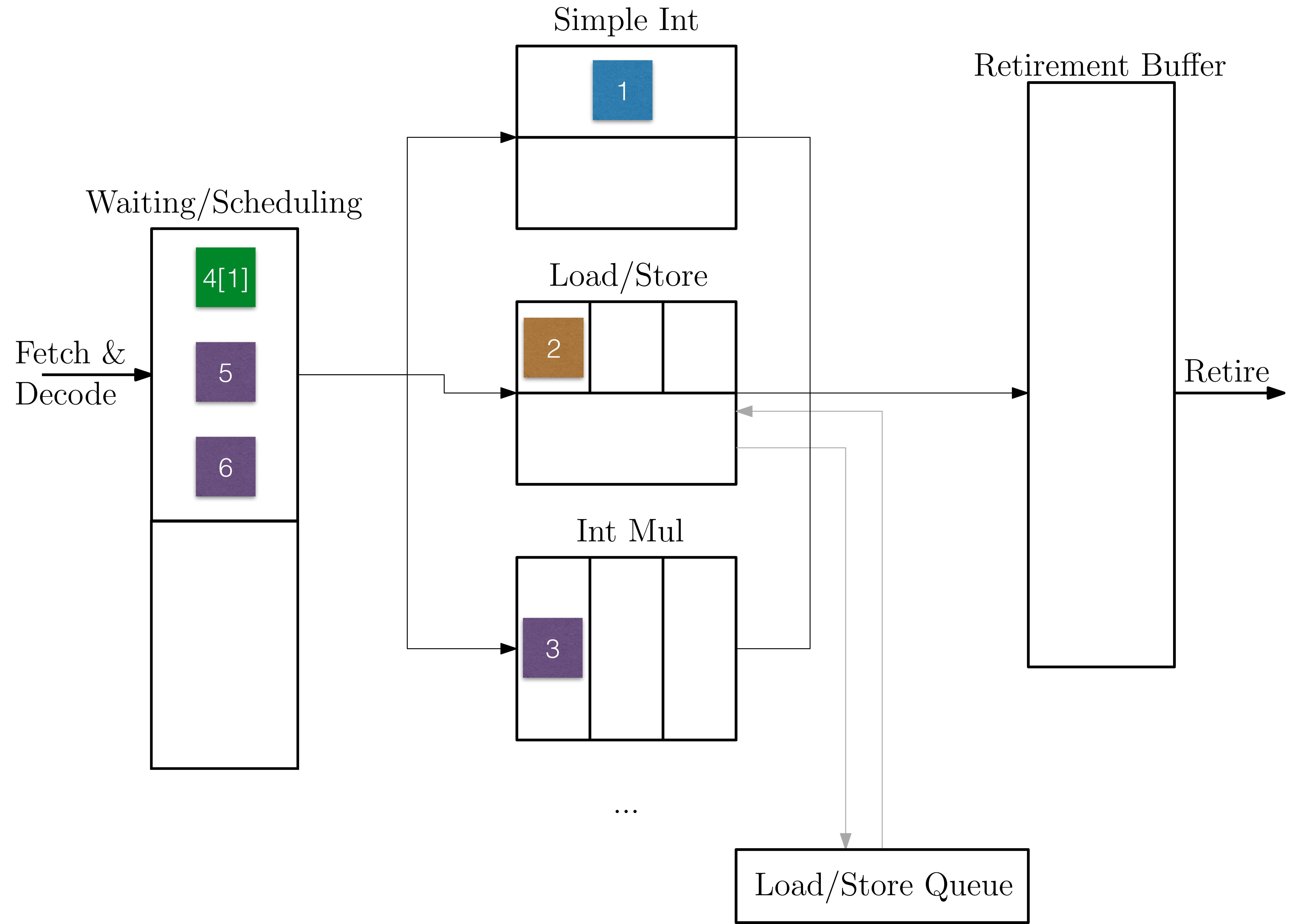


As-if rule (c of
the program,

Essentially, a ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Ord** *retirement*



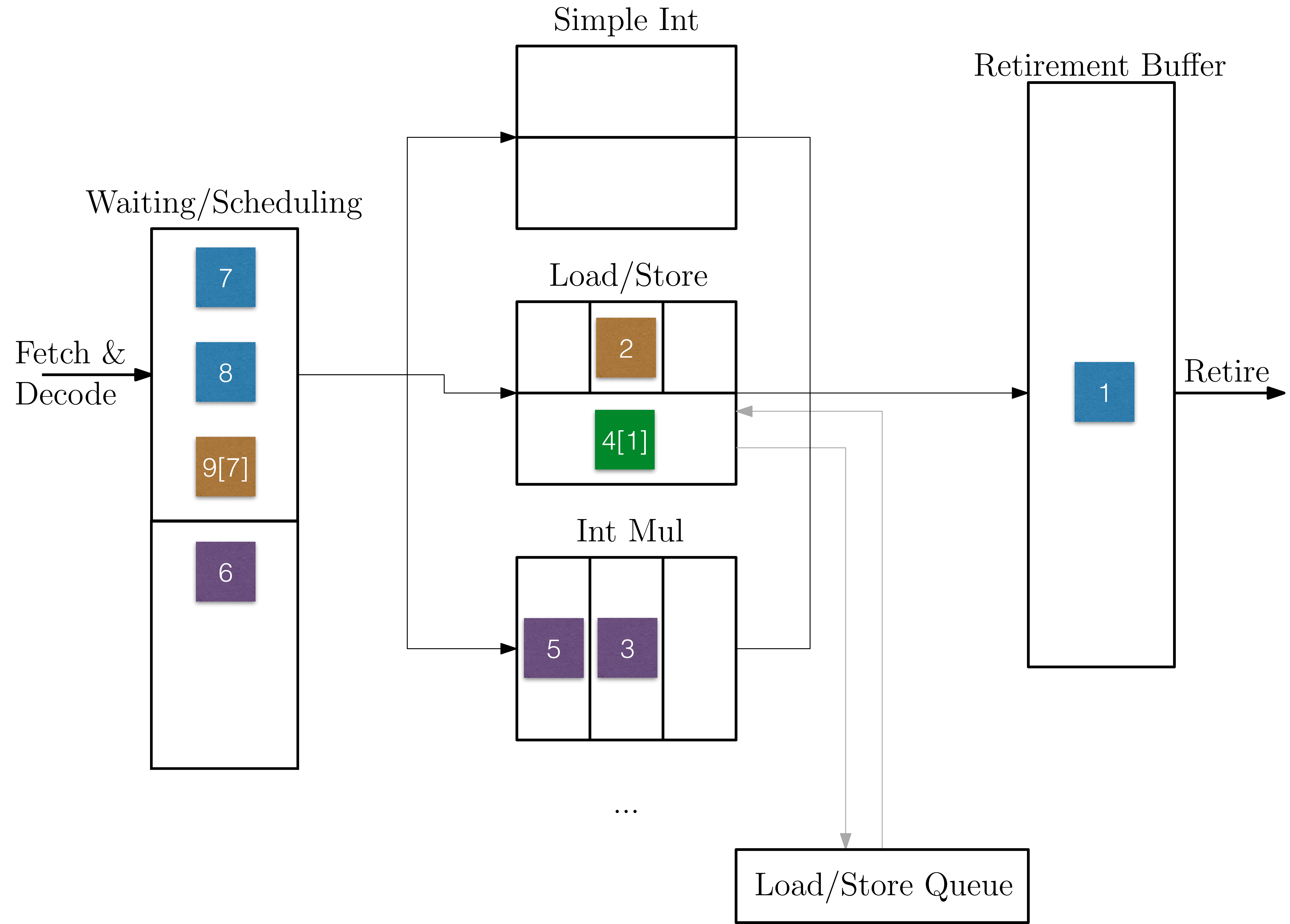
In real CPUs

As-if rule (c ... of
the program, ...

Essentially, a ... ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Or**
  retirement



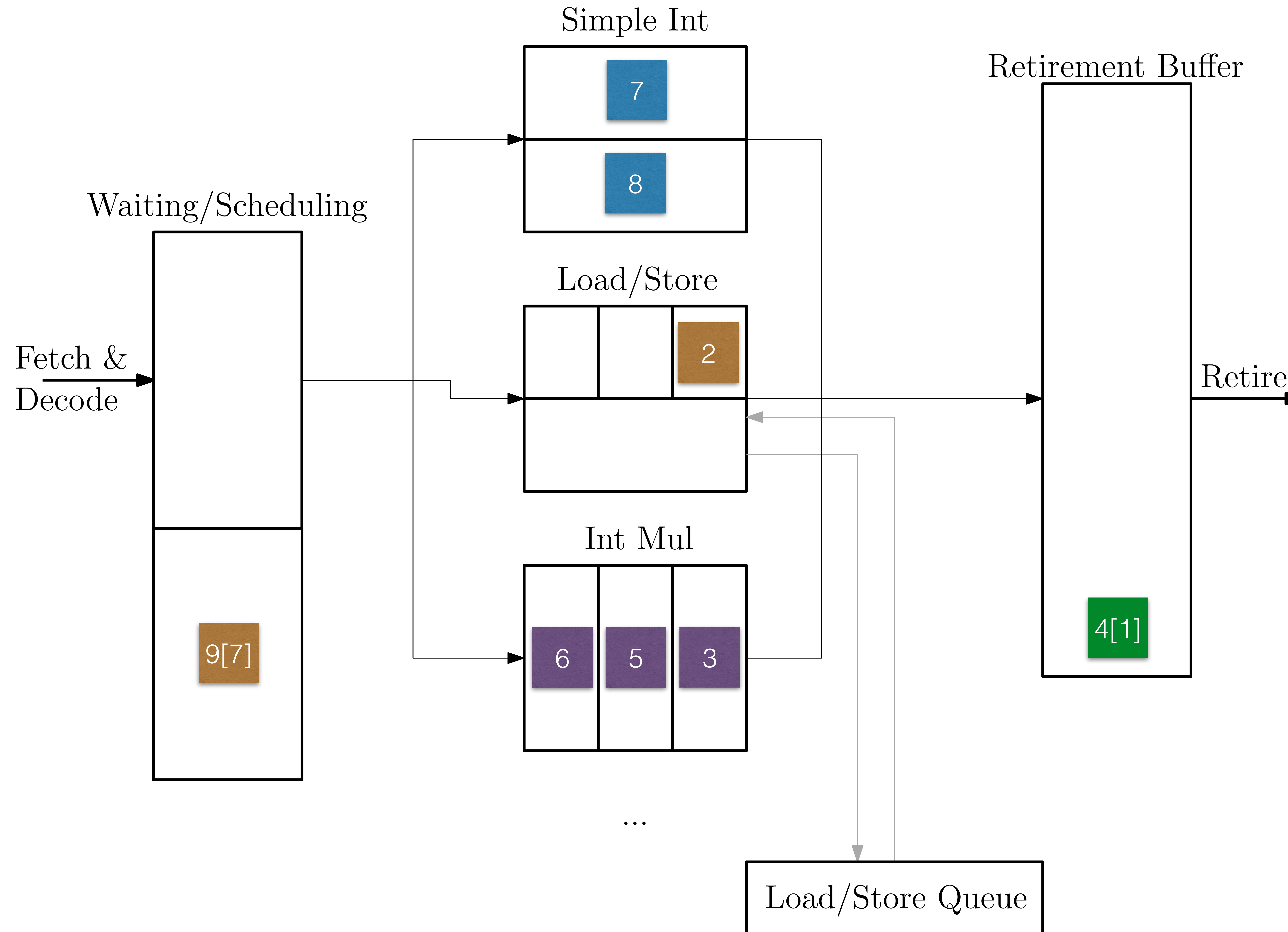
In real CPUs

As-if rule (c... ... of
the program, ...

Essentially, a... ...ution.

Modern CPU...

- **Superscal**...
- **More stag**...
- **Out-of-Or**... *retirement* ...



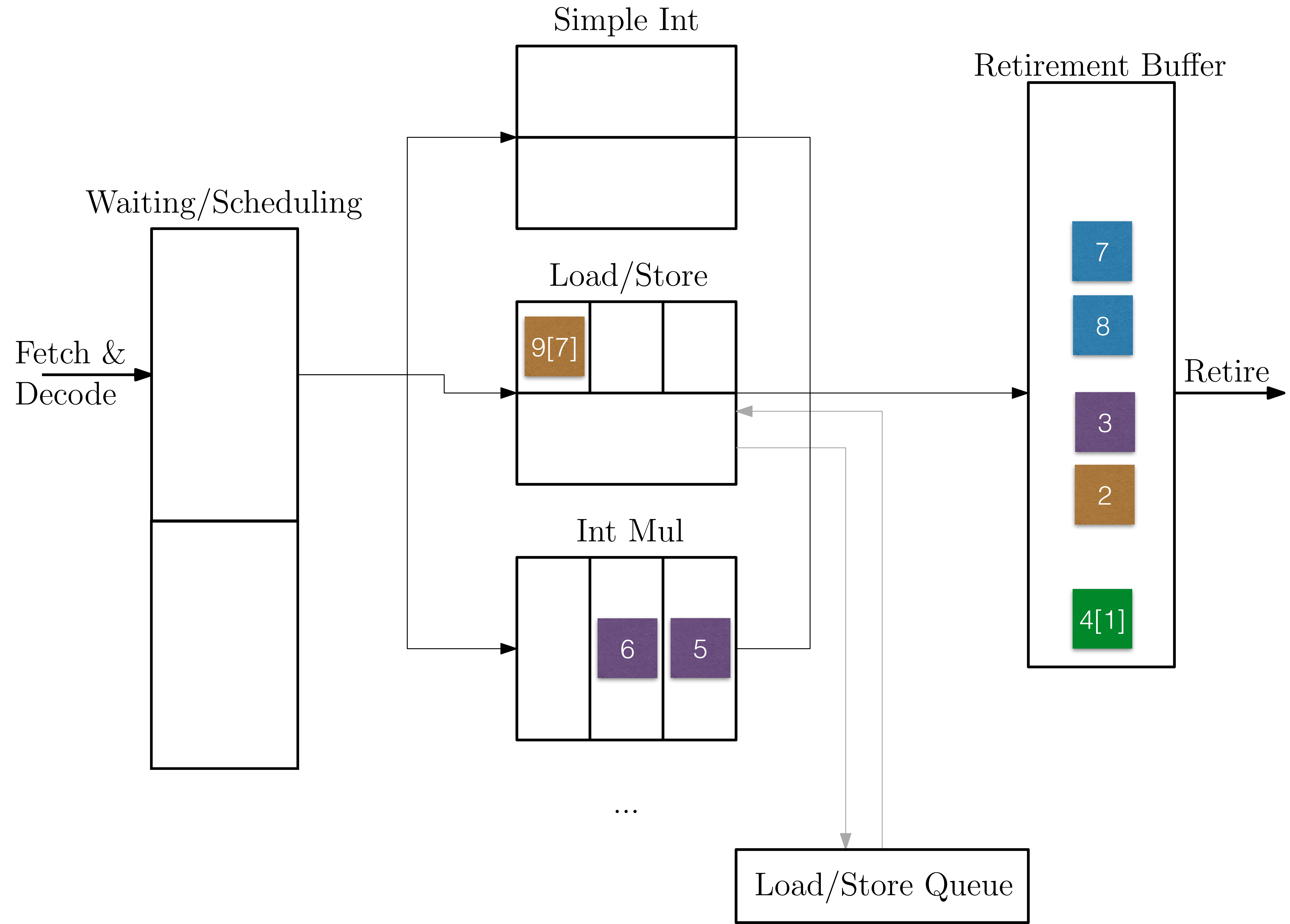
In real CPUs

As-if rule (c … of
the program, …

Essentially, a … ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Or**
  retirement



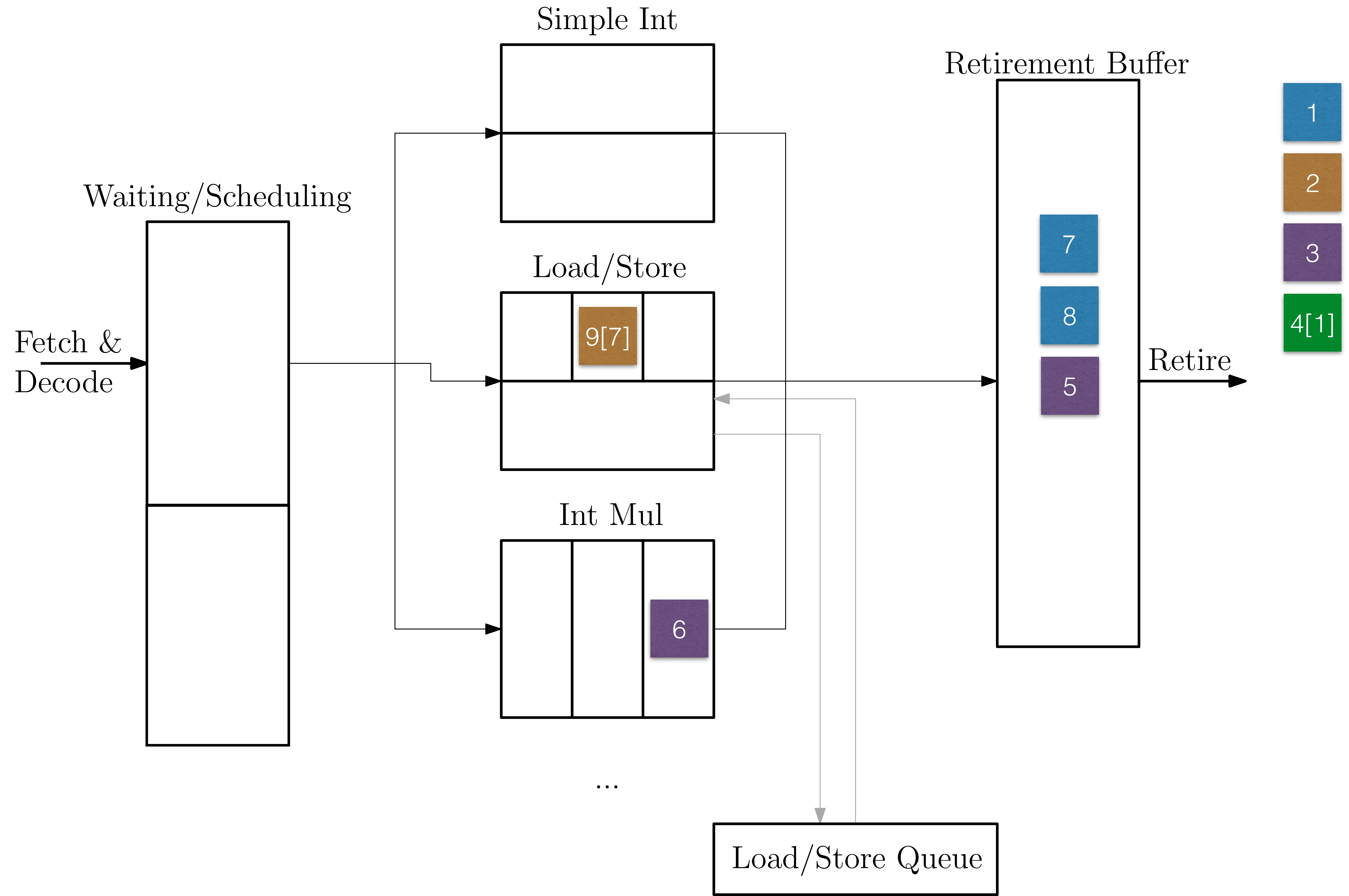
In real CPUs

As-if rule (c … of
the program, …

Essentially, a … ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Ord** … *retirement*



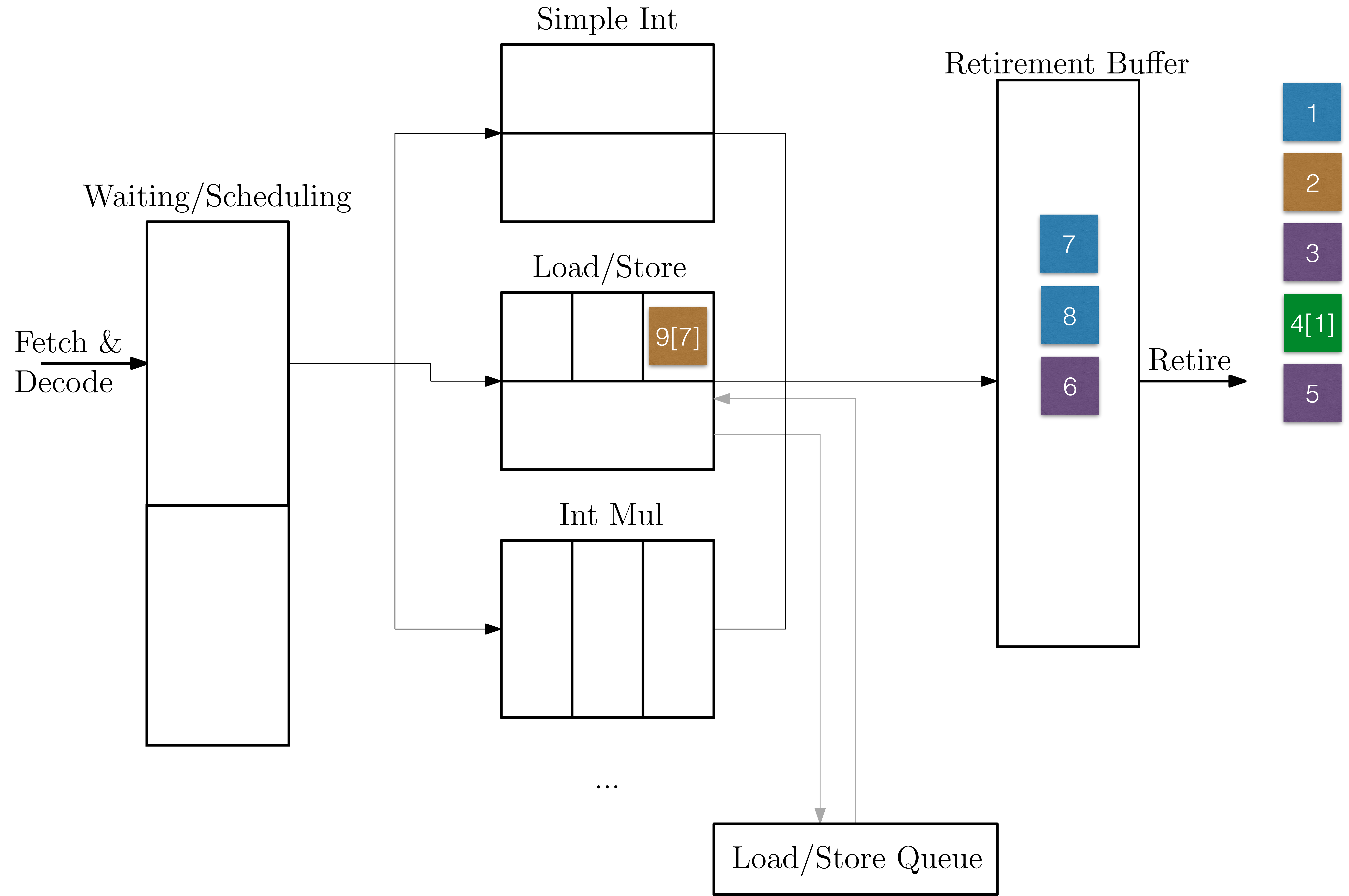
In real CPUs

As-if rule (c ... of
the program, ...

Essentially, a ... ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Or** *retirement*



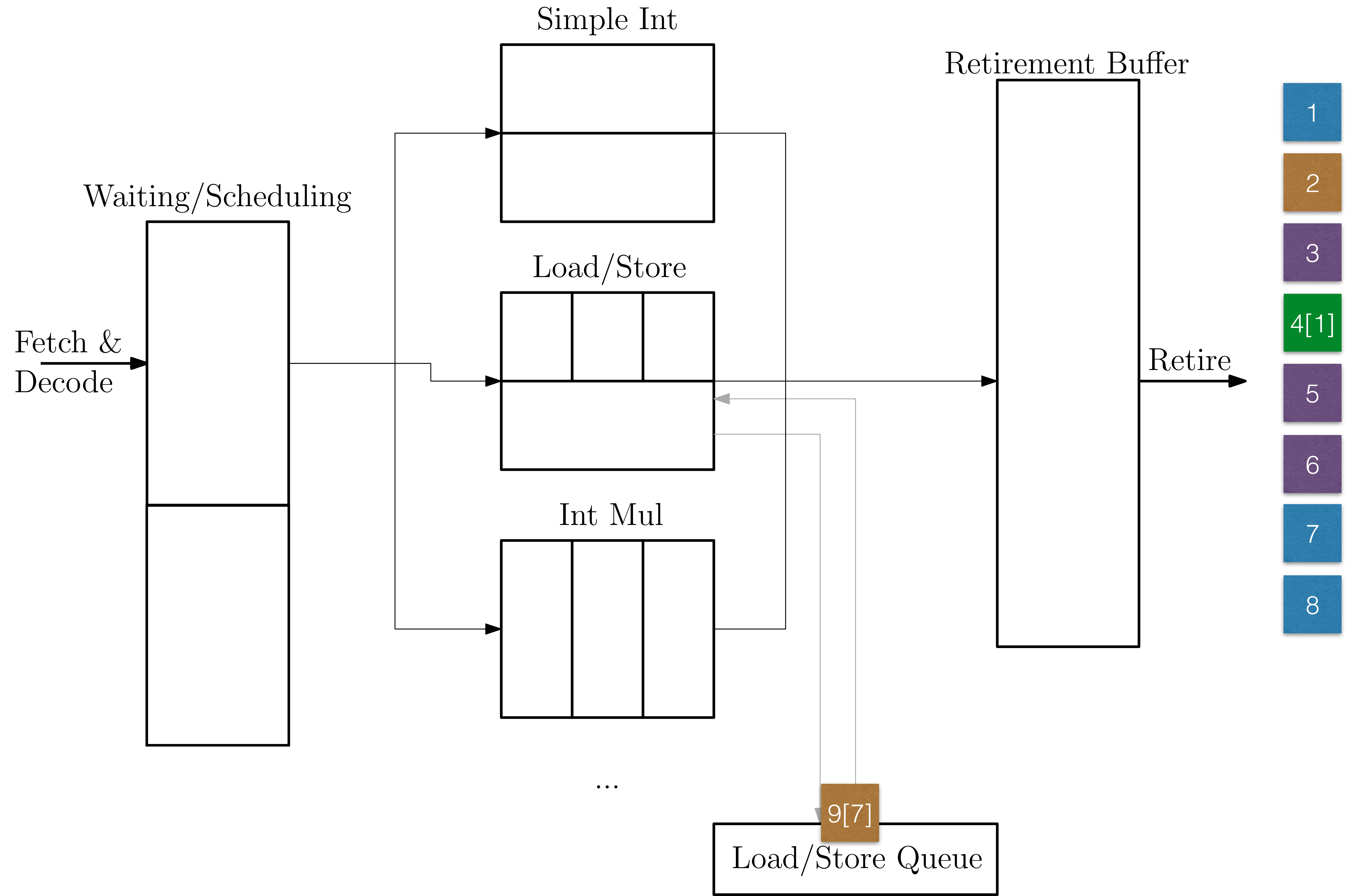
In real CPUs

As-if rule (c   of
the program,

Essentially, a   ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Or**
  retirement



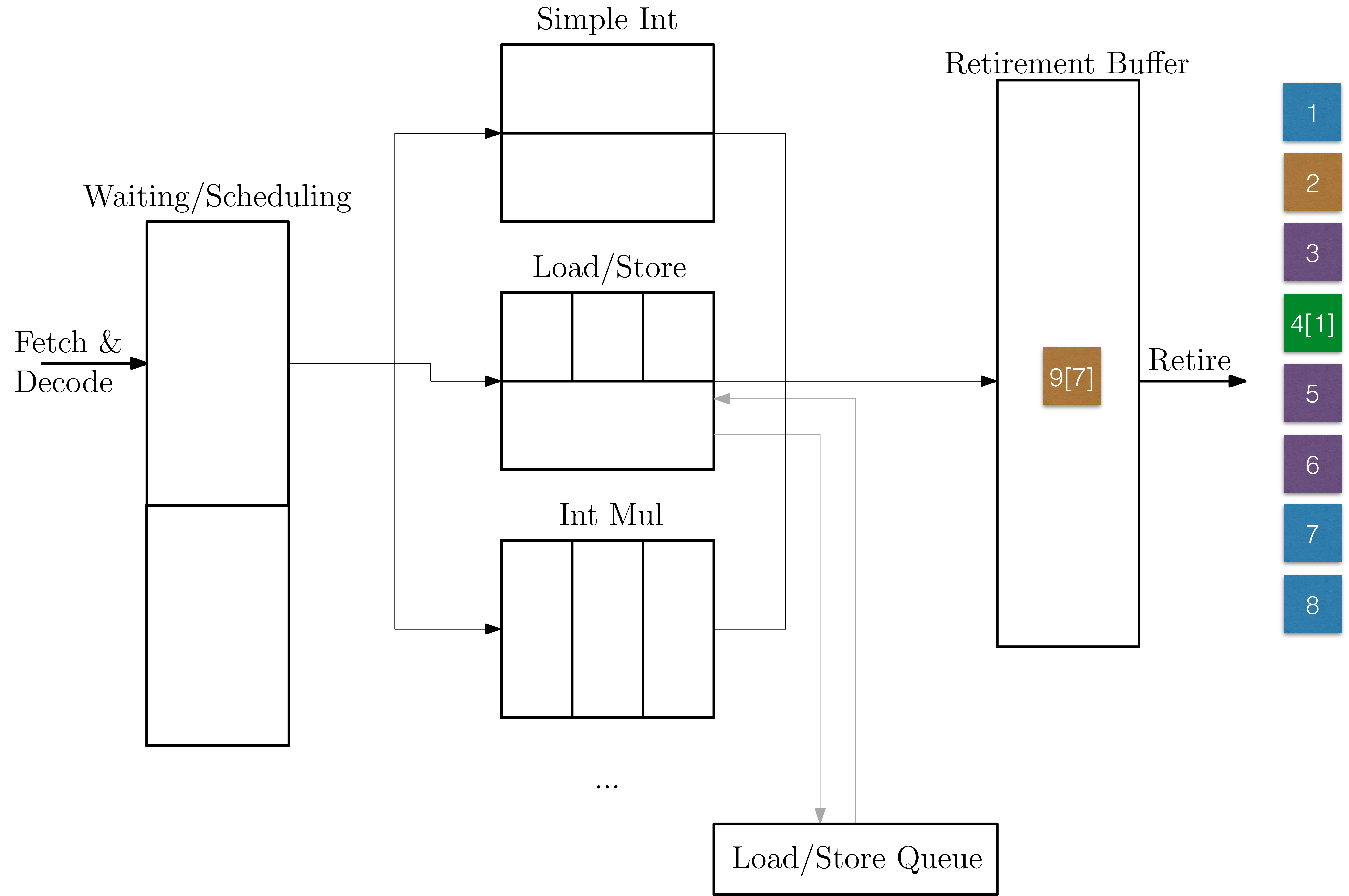
In real CPUs

As-if rule (c… of
the program, …

Essentially, a… …ution.

Modern CPU…

- **Superscal**…
- **More stag**…
- **Out-of-Or**…
  retirement



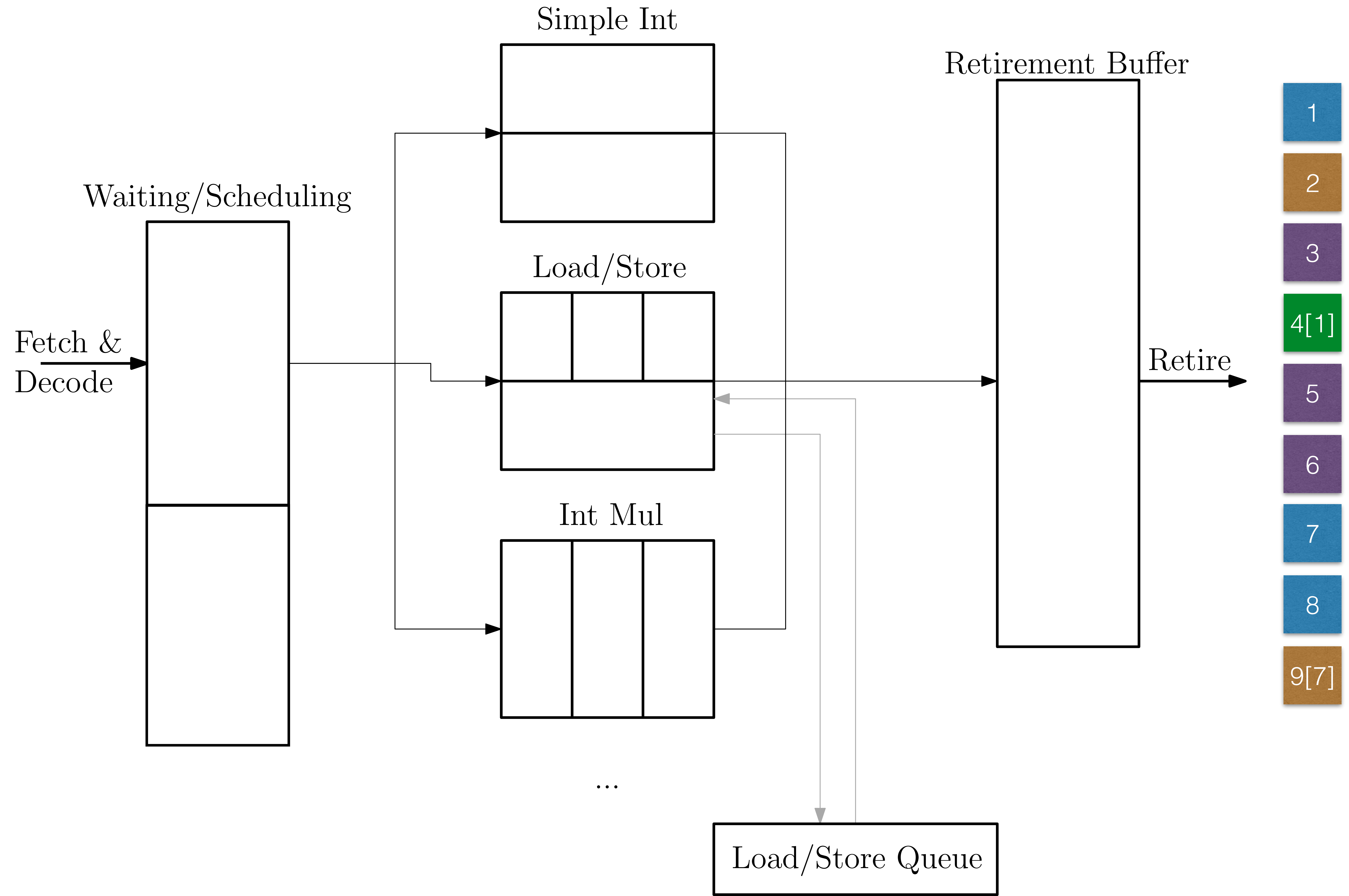
In real CPUs

As-if rule (c ... of
the program, ...

Essentially, a ... ution.

Modern CPU

- **Superscal**
- **More stag**
- **Out-of-Or** *retirement*



In real CPUs

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

Modern CPUs take this to the extreme:

- **Superscalar:** Independent instructions can run simultaneously if resources permit
- **More stages:** 10-25+ finer-grained stages
- **Out-of-Order:** Later independent instructions can ‘overtake' earlier instructions; *retirement* happens in order.

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

Modern CPUs take this to the extreme:

- **Superscalar:** Independent instructions can run simultaneously if resources permit
- **More stages:** 10-25+ finer-grained stages
- **Out-of-Order:** Later independent instructions can ‘overtake' earlier instructions; *retirement* happens in order.
- **Speculation:** Some instructions that are internally executed never retire.

In real CPUs

As-if rule (compilers): if some optimization does not change the *observable behavior* of the program, the compiler is allowed to apply it.

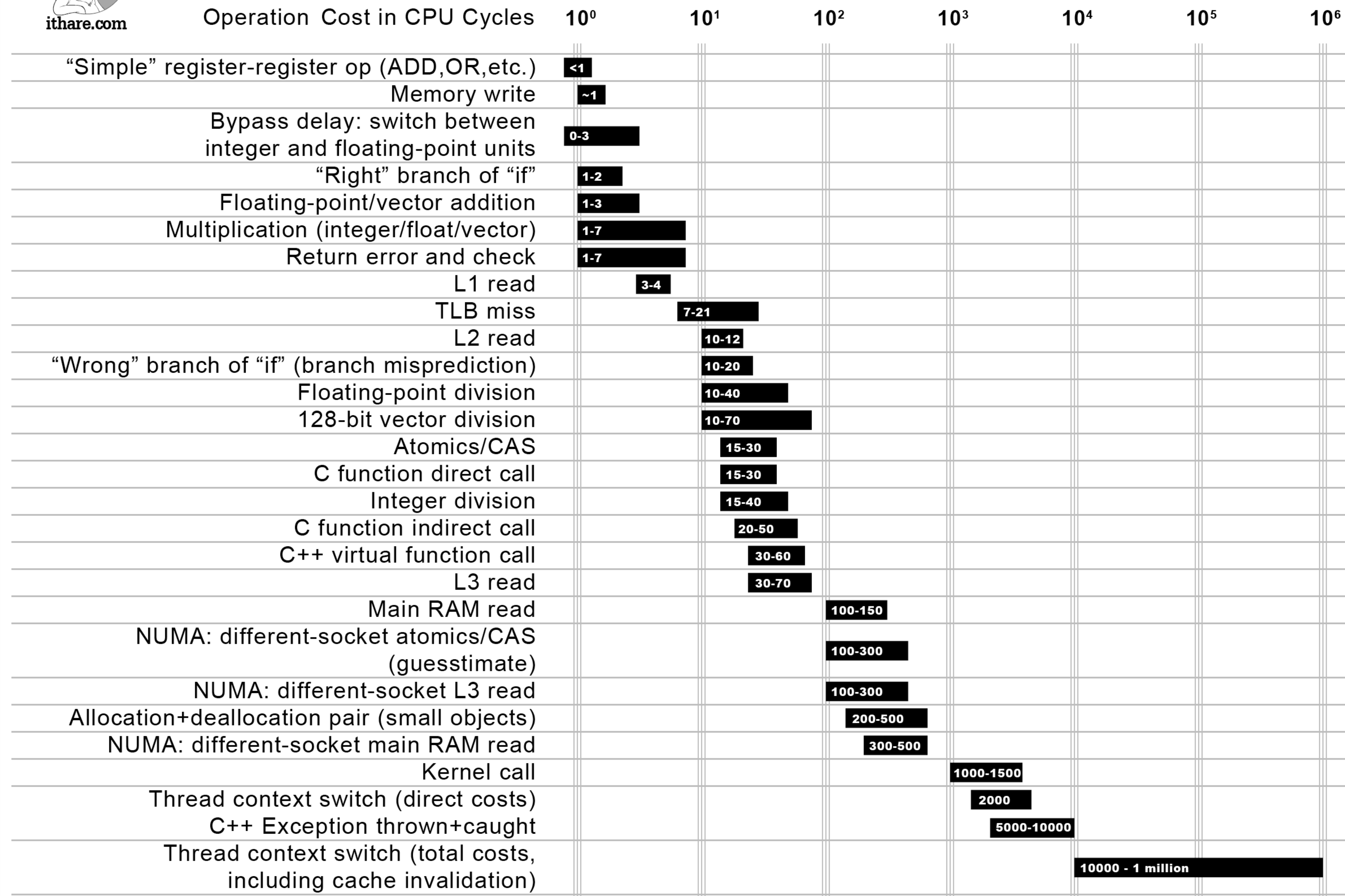
Essentially, a similar rule applies to CPUs w.r.t. the architectural model vs. actual execution.

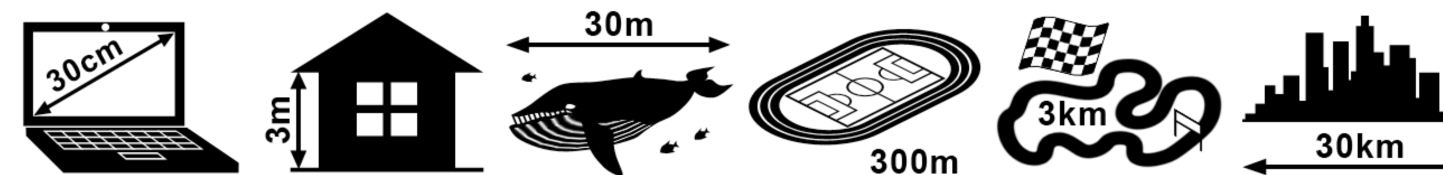
Modern CPUs take this to the extreme:

- **Superscalar:** Independent instructions can run simultaneously if resources permit
- **More stages:** 10-25+ finer-grained stages
- **Out-of-Order:** Later independent instructions can 'overtake' earlier instructions; *retirement* happens in order.
- **Speculation:** Some instructions that are internally executed never retire.

There can be **hundreds of instructions** in flight at any time!

Costs





Latency vs. Throughput

Latency vs. Throughput

Latency:

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.

Technische Universität Braunschweig

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- After scheduling instruction A, how long until a dependent instruction B can start?

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- After scheduling instruction A, how long until a dependent instruction B can start?
- Pipelining has little effect on latency.

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- After scheduling instruction A, how long until a dependent instruction B can start?
- Pipelining has little effect on latency.
- **But:** Pipelining can hide the cost of high-latency instructions!

Technische Universität Braunschweig

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- After scheduling instruction A, how long until a dependent instruction B can start?
- Pipelining has little effect on latency.
- **But:** Pipelining can hide the cost of high-latency instructions!

Throughput:

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- After scheduling instruction A, how long until a dependent instruction B can start?
- Pipelining has little effect on latency.
- **But:** Pipelining can hide the cost of high-latency instructions!

Throughput:

- How many instructions can be completed per unit of time?

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- After scheduling instruction A, how long until a dependent instruction B can start?
- Pipelining has little effect on latency.
- **But:** Pipelining can hide the cost of high-latency instructions!

Throughput:

- How many instructions can be completed per unit of time?
- Example: `add reg, reg` has a 1/4 cycle (reciprocal) throughput on modern x86 CPUs.

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- After scheduling instruction A, how long until a dependent instruction B can start?
- Pipelining has little effect on latency.
- **But:** Pipelining can hide the cost of high-latency instructions!

Throughput:

- How many instructions can be completed per unit of time?
- Example: `add reg, reg` has a 1/4 cycle (reciprocal) throughput on modern x86 CPUs.
- Throughput is affected by pipelining!

Latency vs. Throughput

Latency:

- How long does an instruction take until it completes?
- Example: `add reg, reg` has a 1 cycle latency on typical x86 CPUs.
- …until a dependent instruction B can start?
- …later
- …l per
- Example: `add reg, reg` has a 1/4 cycle (reciprocal) throughput on modern x86 CPUs.
- Throughput is affected by pipelining!

Integer instructions

Instruction	Operands	Ops	Latency	Reciprocal throughput	Execution pipes	Notes, instruction set
Move instructions						
MOV	r8/16,r8/16	1	1	0.2		
MOV	r8h,r8	1	1	0.5		
MOV	r32,r32	1	0	0.17		renaming
MOV	r64,r64	1	0	0.17		renaming
MOV	r8/16,i	1	1	0.2		
MOV	r32/64,i	1	1	0.17		
MOV	r8/16,m	1	1	0.5		
MOV	r32/64,m	1	0-4	0.25		may mirror
MOV	m8/16,r	1		0.5		
MOV	m32/64,r	1	0-4	0.5		may mirror
MOV	m,i	1		0.5		
MOVNTI	m,r	1	~6	1		
MOVZX, MOVSX	r16,r8	1	1	0.5		
MOVZX, MOVSX	r32/64,r	1	1	0.17		
MOVZX, MOVSX	r16,m8	1		0.5		
MOVZX, MOVSX	r32/64,m	1	4	0.25		

3.12 Branch prediction in AMD Bulldozer, Piledriver, Steamroller, and Excavator

The AMD Bulldozer has a new branch prediction design which has no connection to the code cache, unlike previous models. The prediction mechanism is described as a hybrid with a local predictor and a global predictor. Most probably, the branch predictor is based on *perceptrons*. A perceptron is similar to a neuron, and it learns by tracking correlations in the branch history. Unlike the adaptive two-level predictor, the perceptron predictor can learn very long branch patterns.

The branch target buffer (BTB) has two levels, according to AMD's software optimization guide. The level-1 BTB is organized as a set-associative cache with 128 sets of 4 ways = 512 entries. The level-2 BTB has1024 sets of 5 ways = 5120 entries in the Bulldozer and Piledriver, and probably 10240 entries in the Steamroller.

The BTB and predictor is shared between the two cores of a compute unit.

https://www.agner.org/optimize/instruction_tables.pdf

Threats to pipelining

Threats to pipelining

So what threatens the flow of our pipeline, i.e., efficient use of our CPU's resources?

Threats to pipelining

So what threatens the flow of our pipeline, i.e., efficient use of our CPU’s resources?

There are multiple threats; the two most important ones are:

Threats to pipelining

So what threatens the flow of our pipeline, i.e., efficient use of our CPU's resources?

There are multiple threats; the two most important ones are:

- Branching

Threats to pipelining

So what threatens the flow of our pipeline, i.e., efficient use of our CPU's resources?

There are multiple threats; the two most important ones are:

- Branching
- High latency instructions (in particular: cache misses to RAM)

Branch prediction

```
1 if (condition) {
2     foo();
3 } else {
4     bar();
5 }
```

```
1 while(condition) {
2     foo();
3 }
```

Branch prediction

```
1 if (condition) {
2     foo();
3 } else {
4     bar();
5 }
```

```
1 while(condition) {
2     foo();
3 }
```

Which instruction is next depends on the condition (conditional jump)!

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
BRANCH (a<b)	IF	ID	EXE	MEM	WB				
CALL foo				IF	ID	EXE	MEM	WB	
// Instr from foo					IF	ID	EXE	MEM	WB

Branch prediction

```
1 if (condition) {
2     foo();
3 } else {
4     bar();
5 }
```

```
1 while(condition) {
2     foo();
3 }
```

Which instruction is next depends on the condition (conditional jump)!

Idea: predict if jump is taken; until that is resolved, **speculatively** execute instructions.

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
BRANCH (a<b)	IF	ID	EXE	MEM	WB				
CALL foo				IF	ID	EXE	MEM	WB	
// Instr from foo					IF	ID	EXE	MEM	WB

Instruction	Clock cycle								
	1	2	3	4	5	6	7	8	9
BRANCH (a<b)	IF	ID	EXE	MEM	WB				
CALL foo		IF*	ID*	EXE	MEM	WB			
// Instr from foo			IF*	ID	EXE	MEM	WB		

Branch prediction

```
1 if (condition) {
2     foo();
3 } else {
4     bar();
5 }
```

```
1 while(condition) {
2     foo();
3 }
```

Which instruction is next depends on the condition (conditional jump)!

Idea: predict if jump is taken; until that is resolved, **speculatively** execute instructions.

Branch prediction

```
1 if (condition) {
2     foo();
3 } else {
4     bar();
5 }
```

```
1 while(condition) {
2     foo();
3 }
```

Which instruction is next depends on the condition (conditional jump)!

Idea: predict if jump is taken; until that is resolved, **speculatively** execute instructions.

Prediction correct: great, we saved time by not waiting!

Branch prediction

```
1 if (condition) {
2     foo();
3 } else {
4     bar();
5 }
```

```
1 while(condition) {
2     foo();
3 }
```

Which instruction is next depends on the condition (conditional jump)!

Idea: predict if jump is taken; until that is resolved, **speculatively** execute instructions.

Prediction correct: great, we saved time by not waiting!

Otherwise: branch misprediction, throw away intermediate results.

Branch prediction

```
1 if (condition) {
2     foo();
3 } else {
4     bar();
5 }
```

```
1 while(condition) {
2     foo();
3 }
```

Which instruction is next depends on the condition (conditional jump)!

Idea: predict if jump is taken; until that is resolved, **speculatively** execute instructions.

Prediction correct: great, we saved time by not waiting!

Otherwise: branch misprediction, throw away intermediate results.

Performance cost: depends on latency of conditions; on average ~10-20 cycles

Quiz

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

Quiz

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

Quiz

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

Quiz

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

Quiz

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

Quiz question: compiling (unoptimized), what time impact does `DO_SORT` have?

Quiz

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

Quiz question: compiling (unoptimized), what time impact does `DO_SORT` have?

```
# on this laptop (N=2**20, REPETITIONS=1000)
❯ ./quiz1
Done in 10.9049 s
Time per repetition: 0.0109049 s
❯ ./quiz1 sorted
Done in 4.81467 s
Time per repetition: 0.00481467 s
```

```
#   sorted
15744145903   cycles
65867123322   instructions    # 4.18 IPC
16909340627   branches
     536423   branch-misses
# unsorted
35285663593   cycles
65878007797   instructions    # 1.87 IPC
16897962034   branches
  523835843   branch-misses
```

```
1  std::vector<int> data(N);
2  for (int i = 0; i < N; i++)
3      data[i] = std::rand() % 256;  // random values in [0, 255]
4
5  if (DO_SORT)
6      std::ranges::sort(data);
7
8  for (int rep = 0; rep < REPETITIONS; rep++) {
9      for (int i = 0; i < N; i++) {
10         if (data[i] >= 128)
11             sum += data[i];
12     }
13 }
```

Quiz question: compiling (unoptimized), what time impact does `DO_SORT` have?

```
# on this laptop (N=2**20, REPETITIONS=1000)
❯ ./quiz1
Done in 10.9049 s
Time per repetition: 0.0109049 s
❯ ./quiz1 sorted
Done in 4.81467 s
Time per repetition: 0.00481467 s
```

```
#   sorted
15744145903   cycles
65867123322   instructions    # 4.18 IPC
16909340627   branches
     536423   branch-misses
# unsorted
35285663593   cycles
65878007797   instructions    # 1.87 IPC
16897962034   branches
  523835843   branch-misses
```

Per miss:
37.43 cycles
11.54 ns

Branch prediction

How does branch prediction work?

How does branch prediction work?

- **Temporal:** if the branch was taken the last 100 times, predict it will be taken again. Also, recognize patterns such as N-T-N-T or T-T-T-N-T-T-T-N.

How does branch prediction work?

- **Temporal:** if the branch was taken the last 100 times, predict it will be taken again. Also, recognize patterns such as N-T-N-T or T-T-T-N-T-T-T-N.

- **Spatial:** infer from the behavior of nearby branches that were recently handled; particularly important for predicting addresses of indirect branches.

How does branch prediction work?

- **Temporal:** if the branch was taken the last 100 times, predict it will be taken again. Also, recognize patterns such as N-T-N-T or T-T-T-N-T-T-T-N.

- **Spatial:** infer from the behavior of nearby branches that were recently handled; particularly important for predicting addresses of indirect branches.

- Complex heuristics exist (e.g., for handling nested loops)

How does branch prediction work?

- **Temporal:** if the branch was taken the last 100 times, predict it will be taken again. Also, recognize patterns such as N-T-N-T or T-T-T-N-T-T-T-N.
- **Spatial:** infer from the behavior of nearby branches that were recently handled; particularly important for predicting addresses of indirect branches.
- Complex heuristics exist (e.g., for handling nested loops)
- Modern branch predictors achieve >95% accuracy on most typical code.

Branch prediction

How does branch prediction work?

- **Temporal:** if the branch was taken the last 100 times, predict it will be taken again. Also, recognize patterns such as N-T-N-T or T-T-T-N-T-T-T-N.
- **Spatial:** infer from the behavior of nearby branches that were recently handled; particularly important for predicting addresses of indirect branches.
- Complex heuristics exist (e.g., for handling nested loops)
- Modern branch predictors achieve >95% accuracy on most typical code.

Hard to predict branches:
(semi-)random branches taken ~50% of the time, patterns that are too long

Branch prediction

What can we do?

What can we do?

- **Never ever measure running time compiling without optimization!**

What can we do?

- **Never ever measure running time compiling without optimization!**

```
# sorted
 253790507   cycles
1236932506   instructions   # 4.87 IPC
  87424599   branches
    523166   branch-misses
  0.105917   seconds wall time
# unsorted
 226426424   cycles
1165094196   instructions   # 5.15 IPC
  77164816   branches
     26114   branch-misses
  0.098894   seconds wall time
```

What can we do?

- **Never ever measure running time compiling without optimization!**
- If you need debug info (say for profiling), use a release-with-debug-info build.

What can we do?

- **Never ever measure running time compiling without optimization!**
- If you need debug info (say for profiling), use a release-with-debug-info build.
- Measure (see tutorial) before worrying too much (not only about branch mispredictions)!

What can we do?

- **Never ever measure running time compiling without optimization!**
- If you need debug info (say for profiling), use a release-with-debug-info build.
- Measure (see tutorial) before worrying too much (not only about branch mispredictions)!

But no, seriously, what can we do? How did the compiler fix it?

What can we do?

- **Never ever measure running time compiling without optimization!**
- If you need debug info (say for profiling), use a release-with-debug-info build.
- Measure (see tutorial) before worrying too much (not only about branch mispredictions)!

But no, seriously, what can we do? How did the compiler fix it?

- **Branchless** (and vectorized) code.

What can we do?

- **Never ever measure running time compiling without optimization!**
- If you need debug info (say for profiling), use a release-with-debug-info build.
- Measure (see tutorial) before worrying too much (not only about branch mispredictions)!

But no, seriously, what can we do? How did the compiler fix it?

- **Branchless** (and vectorized) code.
- Replace branching by other methods, such as conditional move instructions

Branch prediction

What can we do?

- **Never ever measure running time compiling without optimization!**
- If you need debug info (say for profiling), use a release-with-debug-info build.
- Measure (see tutorial) before worrying too much (not only about branch mispredictions)!

But no, seriously, what can we do? How did the compiler fix it?

- **Branchless** (and vectorized) code.
- Replace branching by other methods, such as conditional move instructions
- Other alternative: restructure the code in other ways (reorder iterations, sorting, …)

Branchless code — Trade-offs

Why not make everything branchless?

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap

Branchless code — Trade-offs

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)

Branchless code — Trade-offs

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)
- Control dependency → data dependency

Branchless code — Trade-offs

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)
- Control dependency → data dependency
- Other branchless code also often involves more work

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)
- Control dependency → data dependency
- Other branchless code also often involves more work
- So it most often helps only if:

Branchless code — Trade-offs

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)
- Control dependency → data dependency
- Other branchless code also often involves more work
- So it most often helps only if:
  - Both paths are cheap

Branchless code — Trade-offs

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)
- Control dependency → data dependency
- Other branchless code also often involves more work
- So it most often helps only if:
  - Both paths are cheap
  - The branch is hard to predict

Branchless code — Trade-offs

Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)
- Control dependency → data dependency
- Other branchless code also often involves more work
- So it most often helps only if:
  - Both paths are cheap
  - The branch is hard to predict
  - Or if the branch prevents other optimizations

Branchless code — Trade-offs

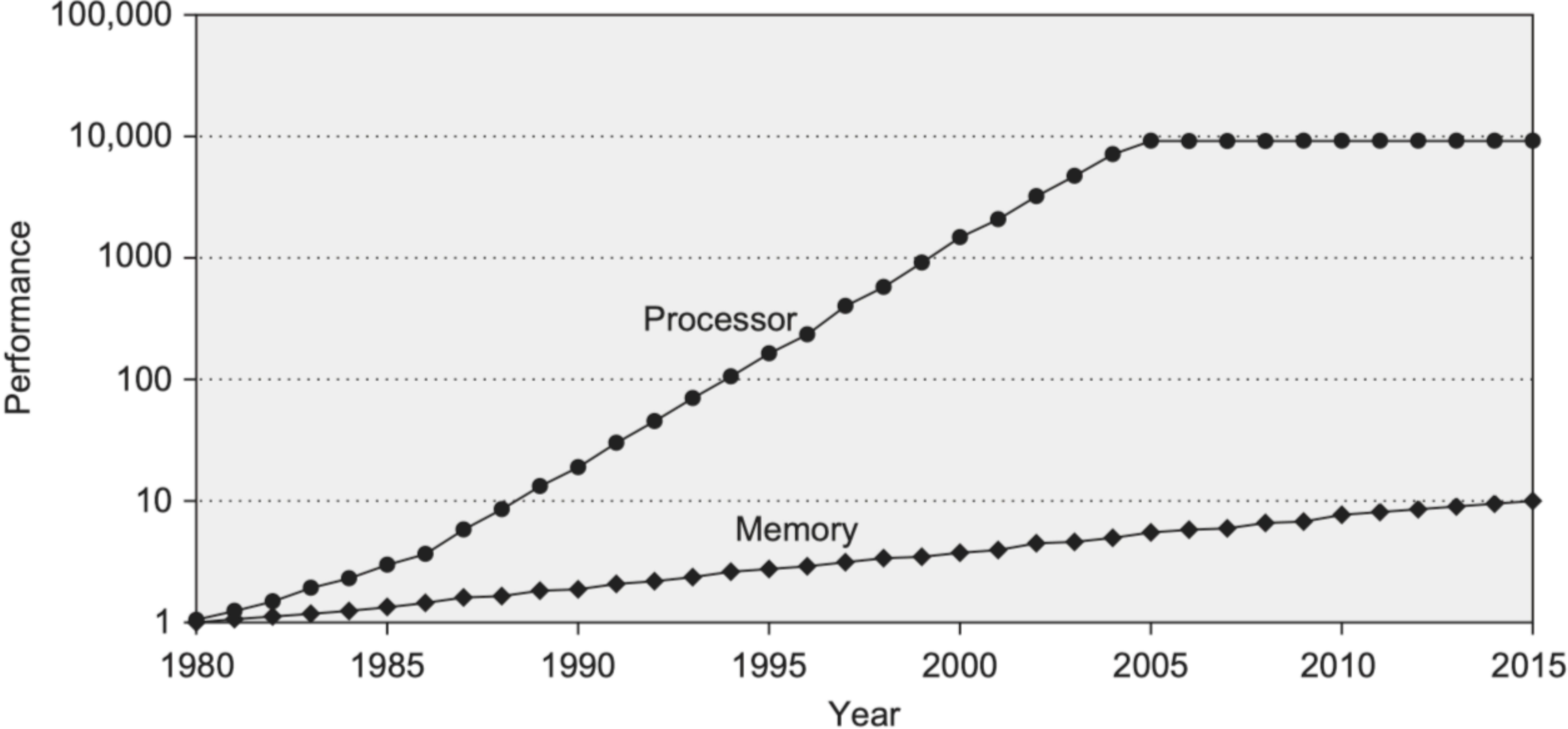
Why not make everything branchless?

- Well-predicted conditional jumps are really cheap
- Conditional moves compute both paths (can be significantly slower)
- Control dependency → data dependency
- Other branchless code also often involves more work
- So it most often helps only if:
  - Both paths are cheap
  - The branch is hard to predict
  - Or if the branch prevents other optimizations
- **Measure first** — then do a single change and measure before and after.

The other elephant in the room

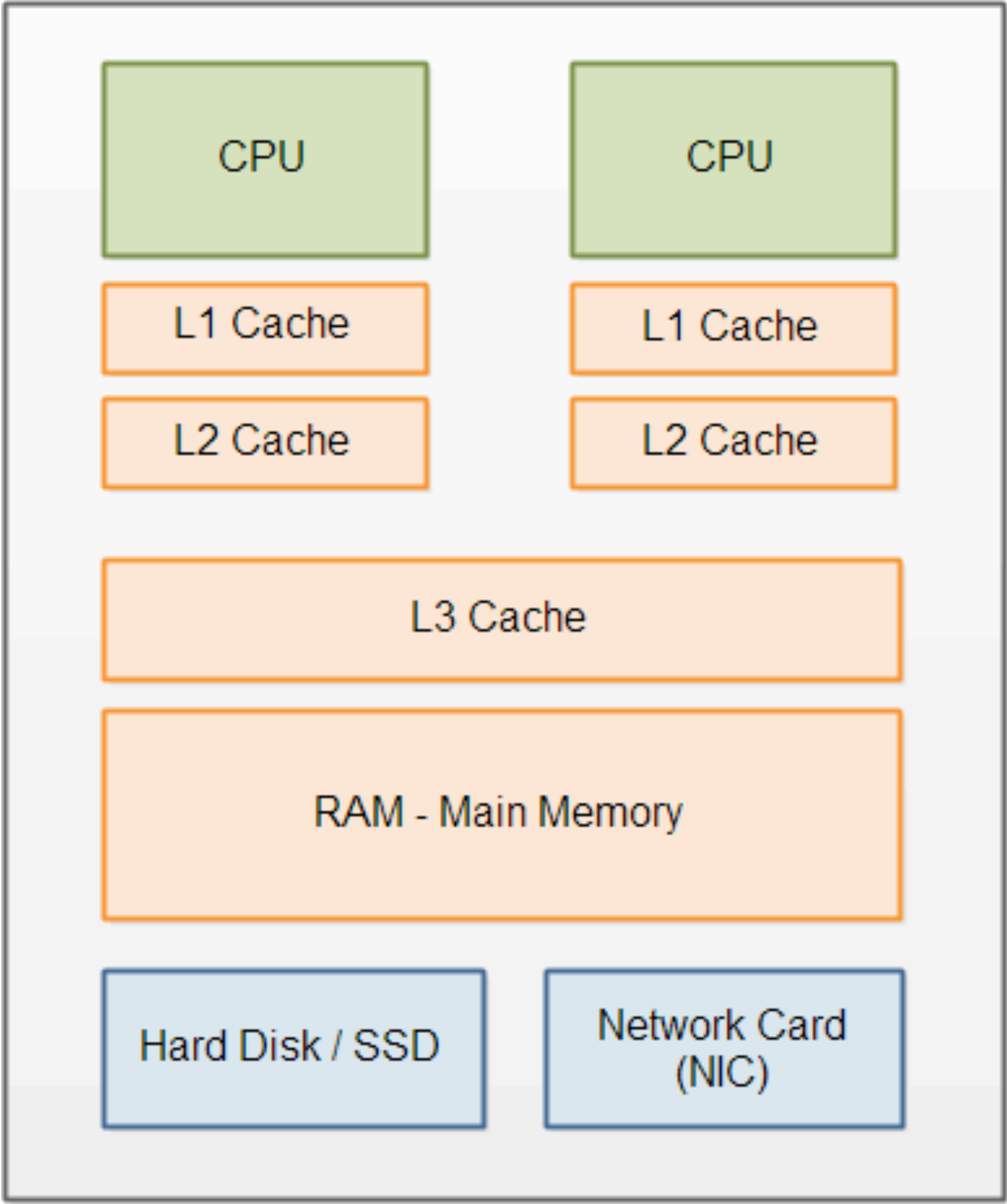
CPU speed vs RAM speed

CPU speed vs RAM speed



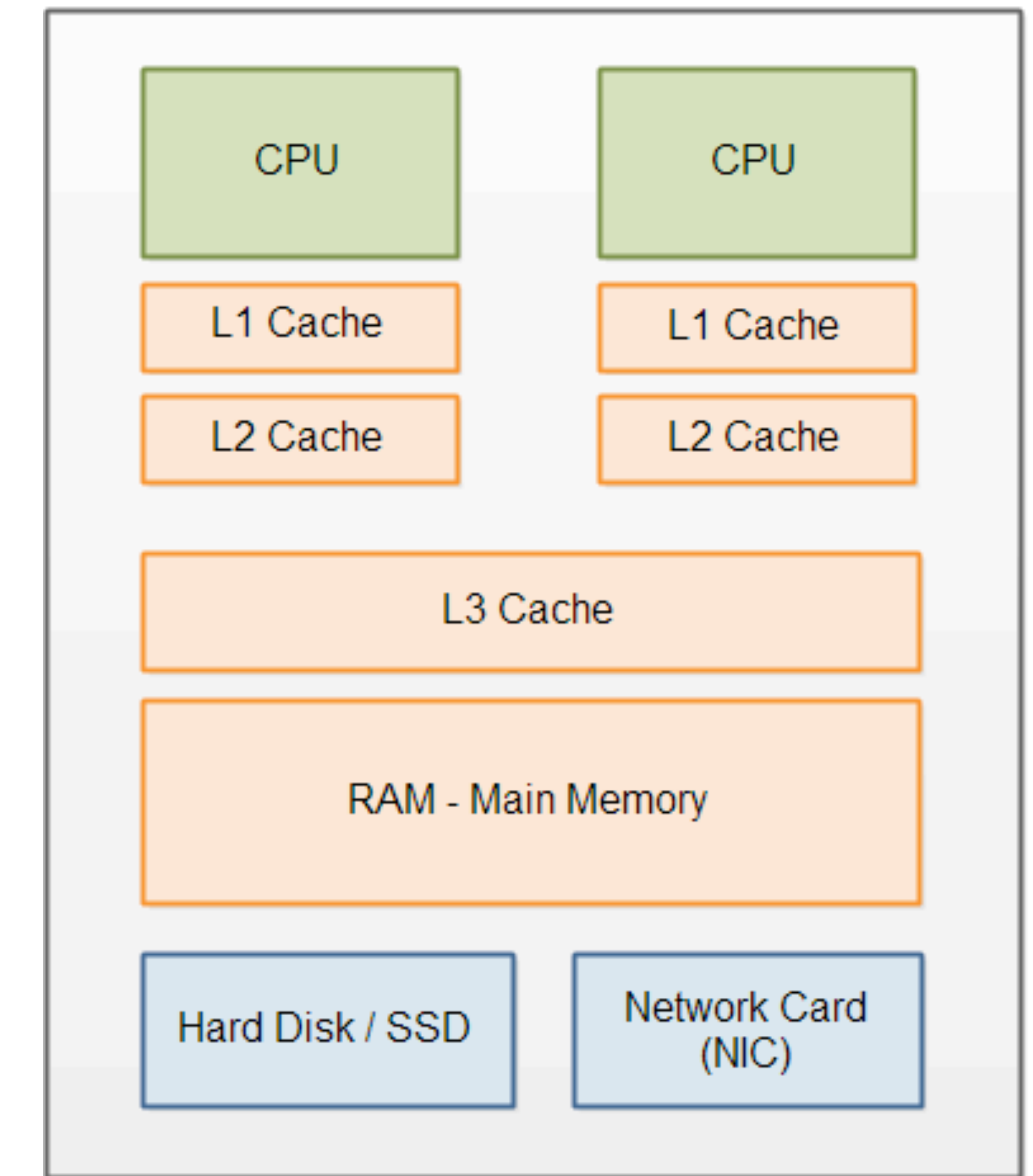
CPU speed vs RAM speed

100,000
10,000
1000
100
10
1
Performance
Processor
Memory
1980 1985 1990 1995 2000 2005 2010 2015
Year



CPU speed vs RAM speed

Performance
100,000
10,000
1000
100
10
1
Processor
Memory
1980 1985 1990 1995 2000 2005 2010 2015
Year



Memory access: nowadays has hundreds of cycles of latency.

Keeping the CPU busy: caches with much lower latency.

Cache Levels

Level	Size range	Latency	Throughput/Bandwidth	What fits
L1	32 kB-256 kB per core	~4 cycles (~1 ns)	~2+1 lines/cycle ~1 TB/second	A few small arrays
L2	256 kB-2 MB per core	~12 cycles (~3 ns)	~1 TB/second	Moderate data structures
L3	8-192 MB shared	~40 cycles (~10 ns)	~400 GB/second	Large data set
DRAM	8-256 GB	~200 cycles (~50 ns)	~64 GB/second per channel	Hopefully, the rest of your data

Cache Levels

Level	Size range	Latency	Throughput/Bandwidth	What fits
L1	32 kB-256 kB per core	~4 cycles (~1 ns)	~2+1 lines/cycle ~1 TB/second	A few small arrays
L2	256 kB-2 MB per core	~12 cycles (~3 ns)	~1 TB/second	Moderate data structures
L3	8-192 MB shared	~40 cycles (~10 ns)	~400 GB/second	Large data set
DRAM	8-256 GB	~200 cycles (~50 ns)	~64 GB/second per channel	Hopefully, the rest of your data

Data transfer: in units of cache lines (usually 64 Bytes)

So: loading one `int` (`a[0]`) from main memory actually fetches 16 `int`s (`a[0]..a[15]`).

How can caches work at all?

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program’s memory access pattern look to help cache hits?

As with branches, **predictability** is key!

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

- **Spatial locality:** accessing `x` means you will likely access things close to `x`

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

- **Spatial locality:** accessing `x` means you will likely access things close to `x`
- **Temporal locality:** accessing `x` means you will likely access `x` again soon

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

- **Spatial locality:** accessing `x` means you will likely access things close to `x`
- **Temporal locality:** accessing `x` means you will likely access `x` again soon

How are these used?

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

- **Spatial locality:** accessing `x` means you will likely access things close to `x`
- **Temporal locality:** accessing `x` means you will likely access `x` again soon

How are these used?

- Least recently used (LRU)-like eviction policies (temporal locality)

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

- **Spatial locality:** accessing `x` means you will likely access things close to `x`
- **Temporal locality:** accessing `x` means you will likely access `x` again soon

How are these used?

- Least recently used (LRU)-like eviction policies (temporal locality)
- Loading whole cache lines brings close-by data into cache (spatial locality)

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

- **Spatial locality:** accessing `x` means you will likely access things close to `x`
- **Temporal locality:** accessing `x` means you will likely access `x` again soon

How are these used?

- Least recently used (LRU)-like eviction policies (temporal locality)
- Loading whole cache lines brings close-by data into cache (spatial locality)
- Other predictable patterns, e.g., linear forward/backward scans (hardware prefetcher)

How can caches work at all?

Tiny caches: how can they not constantly run into cache misses?

How can a program's memory access pattern look to help cache hits?

As with branches, **predictability** is key!

Two properties of access patterns satisfied by most programs:

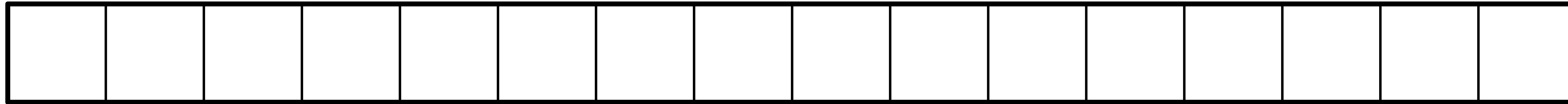
- **Spatial locality:** accessing x means you will likely access things close to x
- **Temporal locality:** accessing x means you will likely access x again soon

How are these used?

```
1 // Prefetcher works: sequential → next cache line is pre-fetched
2 for (int i = 0; i < N; i++)  sum += a[i];
3
4 // Prefetcher helpless: each pointer leads somewhere unpredictable
5 for (Node* p = head; p; p = p->next)  sum += p->val;
```

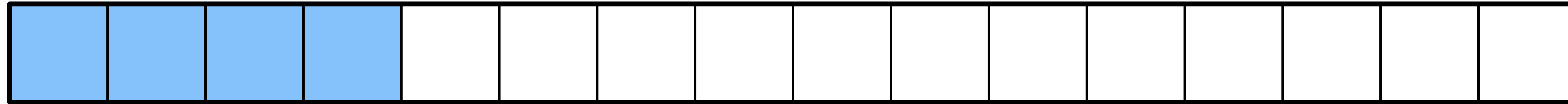
- Other predictable patterns, e.g., linear forward/backward scans (hardware prefetcher)

A $N \times N$ integer matrix in memory ($N = 4$), row major layout:



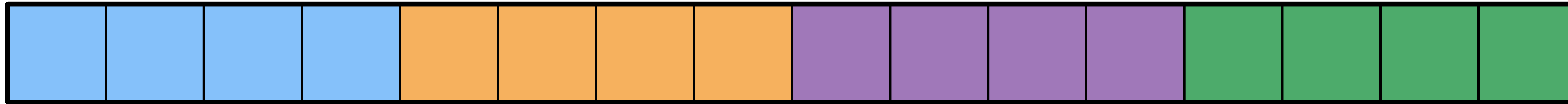
Key Experiment: Matrix Traversal

A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

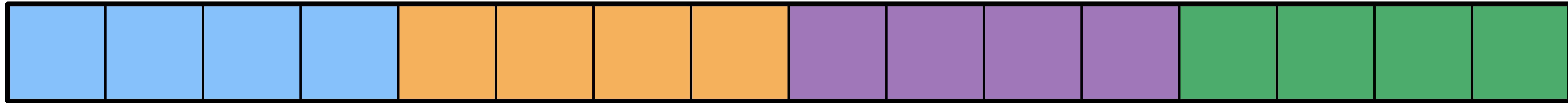


Key Experiment: Matrix Traversal

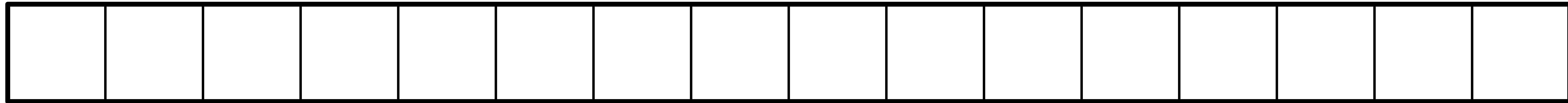
A $N \times N$ integer matrix in memory ($N = 4$), row major layout:



A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

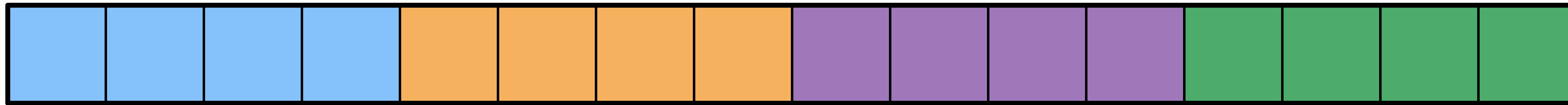


Matching (row-major) traversal:

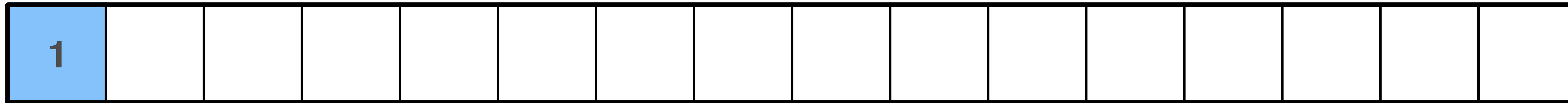


Key Experiment: Matrix Traversal

A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

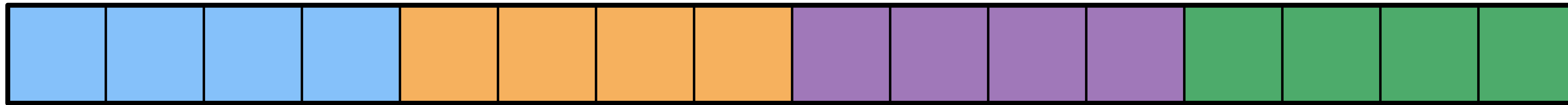


Matching (row-major) traversal:

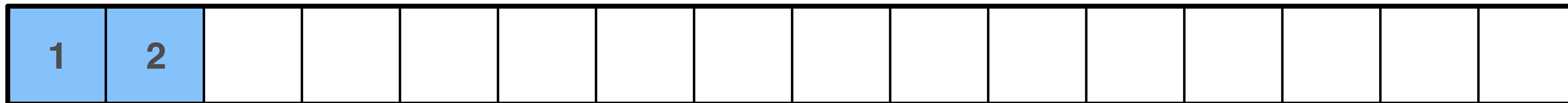


Key Experiment: Matrix Traversal

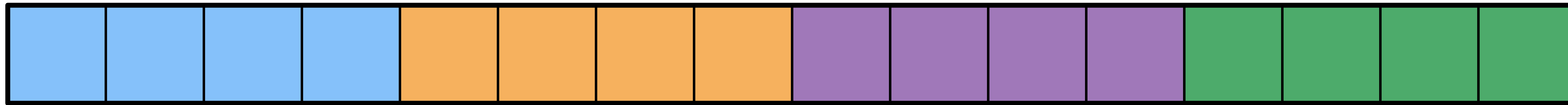
A $N \times N$ integer matrix in memory ($N = 4$), row major layout:



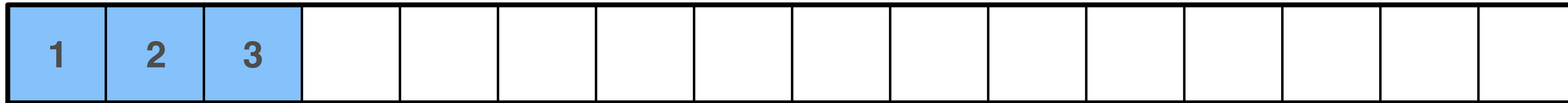
Matching (row-major) traversal:



A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

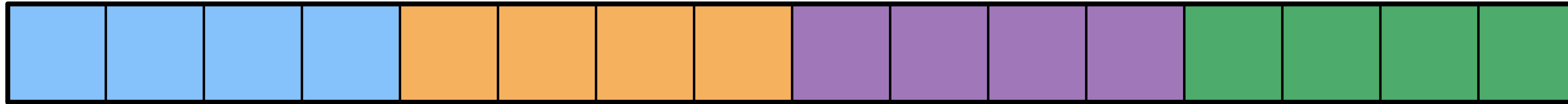


Matching (row-major) traversal:

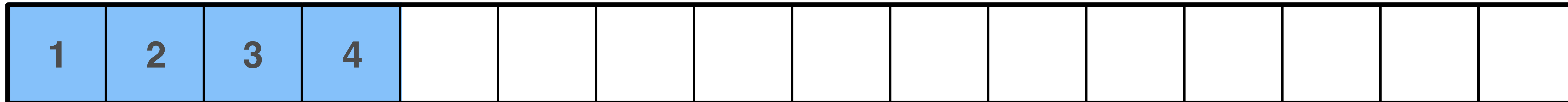


Key Experiment: Matrix Traversal

A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

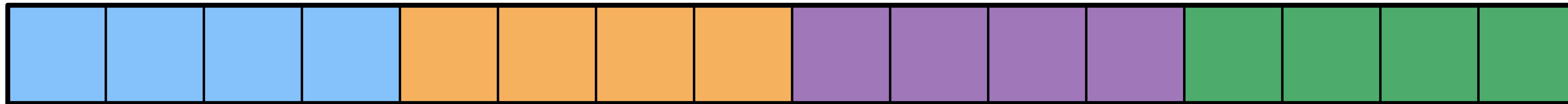


Matching (row-major) traversal:



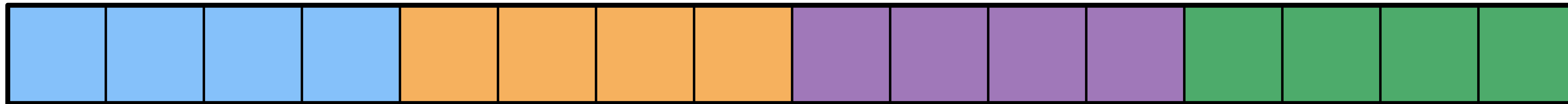
Key Experiment: Matrix Traversal

A $N \times N$ integer matrix in memory ($N = 4$), row major layout:



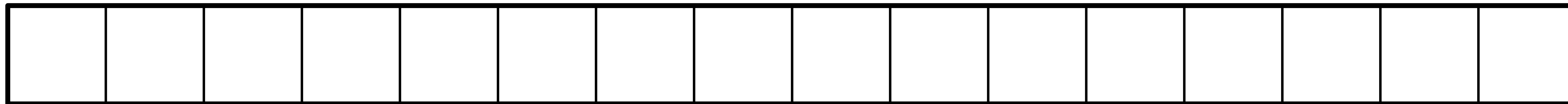
Matching (row-major) traversal:

A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

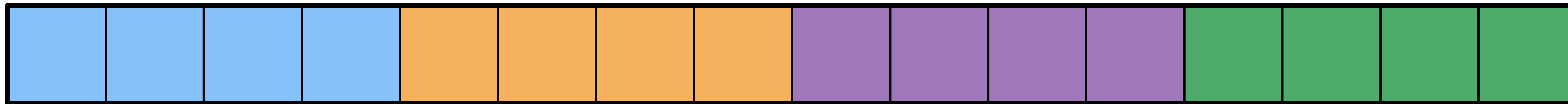


Matching (row-major) traversal:

Mismatched (column-major) traversal:

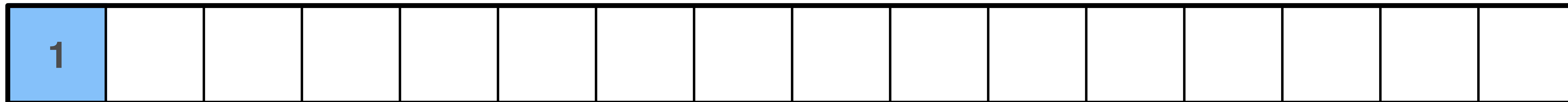


A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

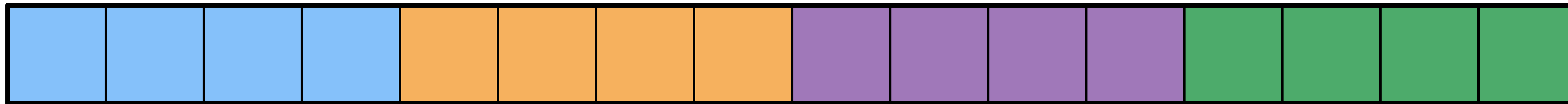


Matching (row-major) traversal:

Mismatched (column-major) traversal:

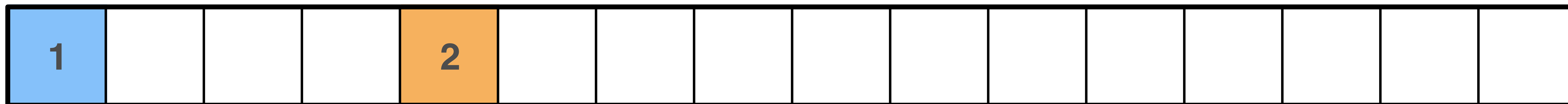


A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

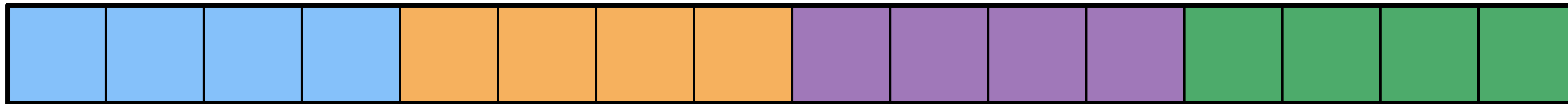


Matching (row-major) traversal:

Mismatched (column-major) traversal:

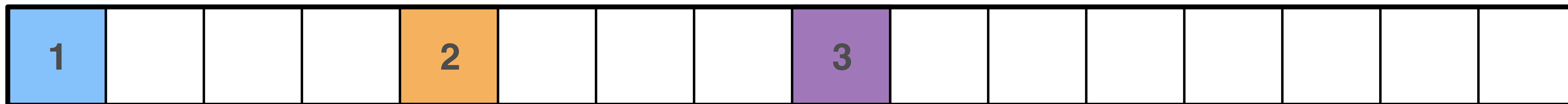


A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

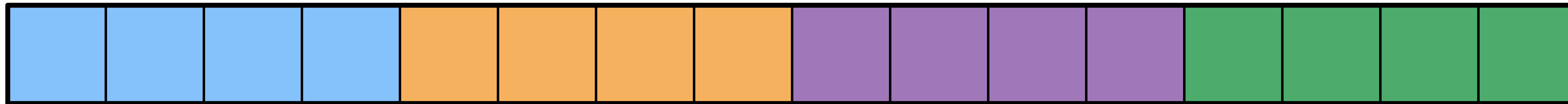


Matching (row-major) traversal:

Mismatched (column-major) traversal:

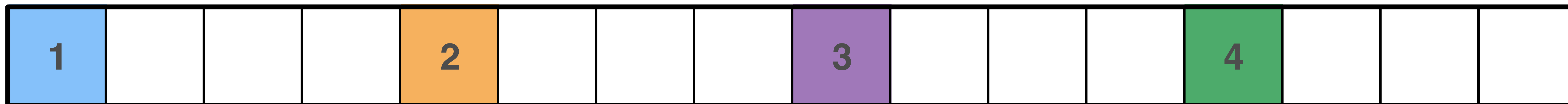


A $N \times N$ integer matrix in memory ($N = 4$), row major layout:



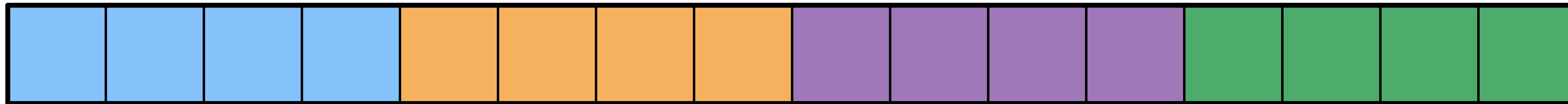
Matching (row-major) traversal:

Mismatched (column-major) traversal:



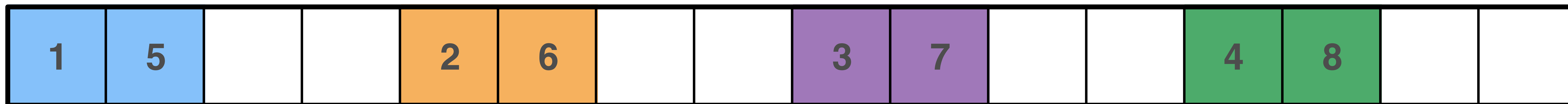
Key Experiment: Matrix Traversal

A $N \times N$ integer matrix in memory ($N = 4$), row major layout:

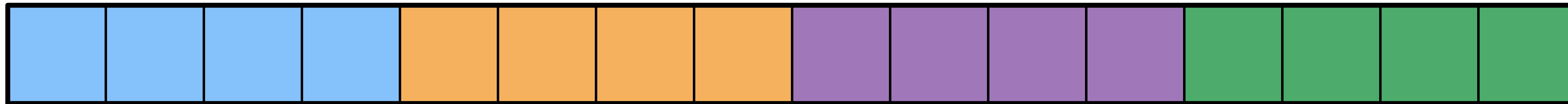


Matching (row-major) traversal:

Mismatched (column-major) traversal:



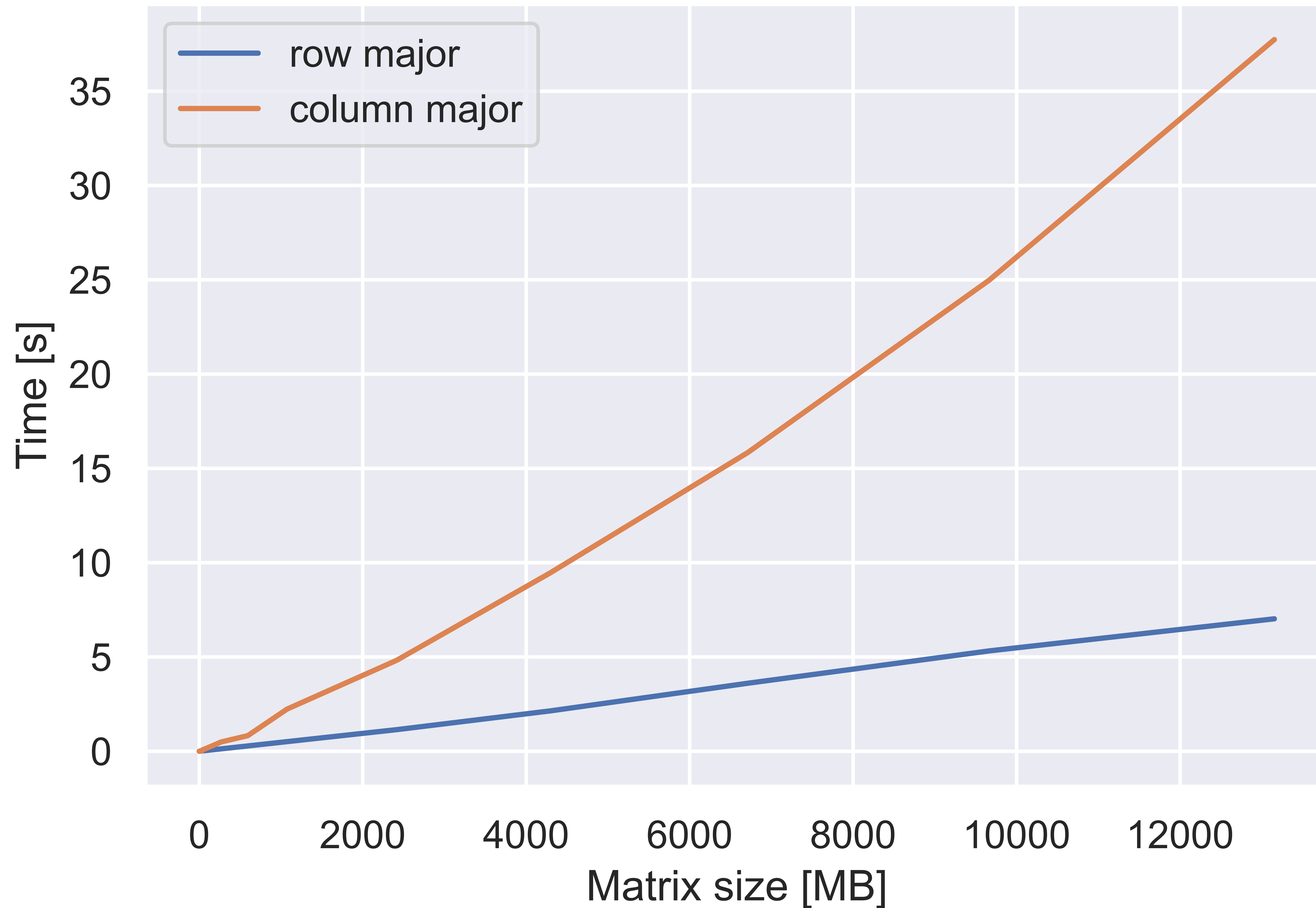
A $N \times N$ integer matrix in memory ($N = 4$), row major layout:



Matching (row-major) traversal:

Mismatched (column-major) traversal:

Key Experiment: Matrix Writing



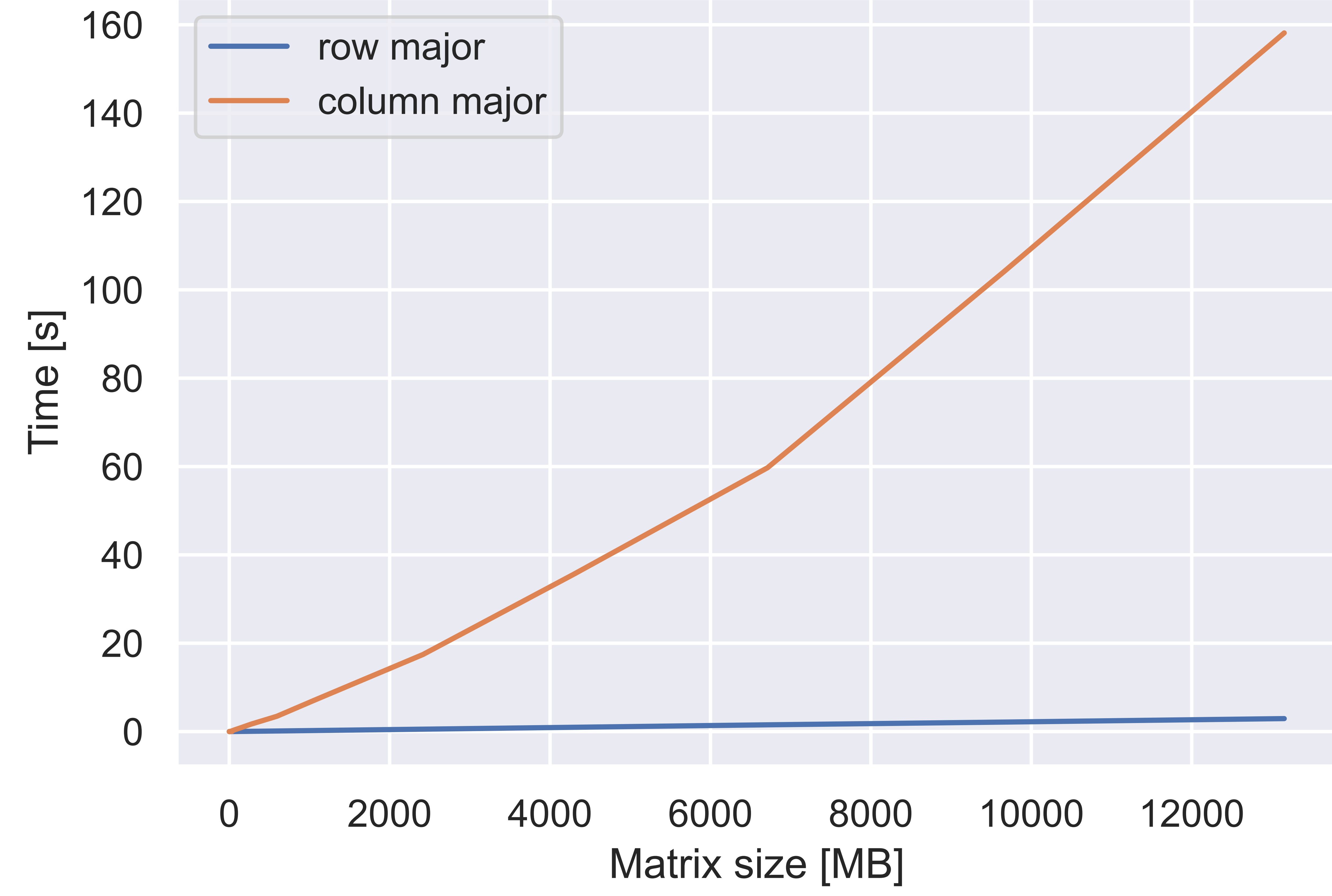
Column major, N = 16384:

```
14375469415 cycles
 4786912494 instructions
       0.33 ins/cycle
 3126991309 cache refs
  253004603 cache misses
       8.09 % miss rate
```

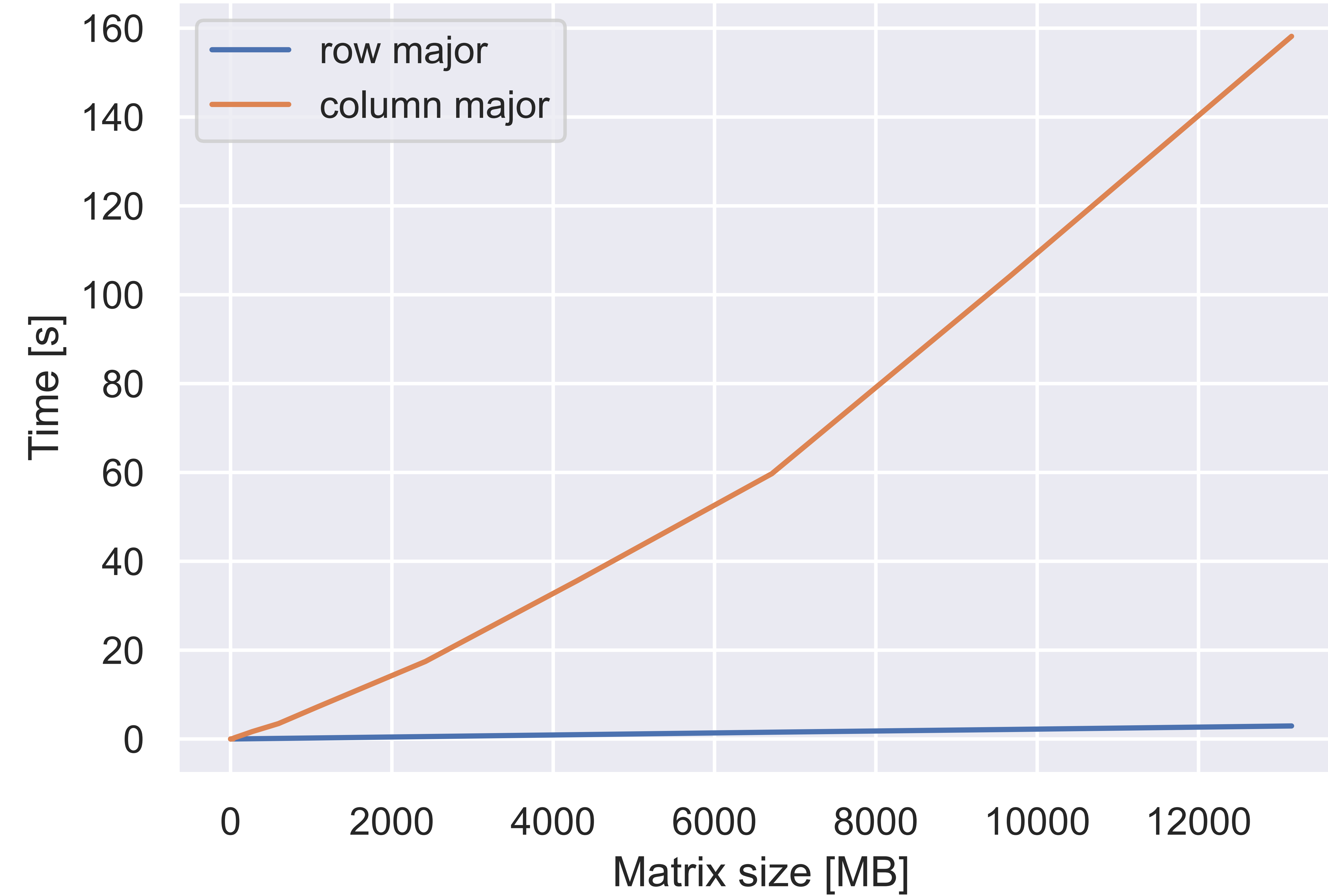
Row major, N = 16384:

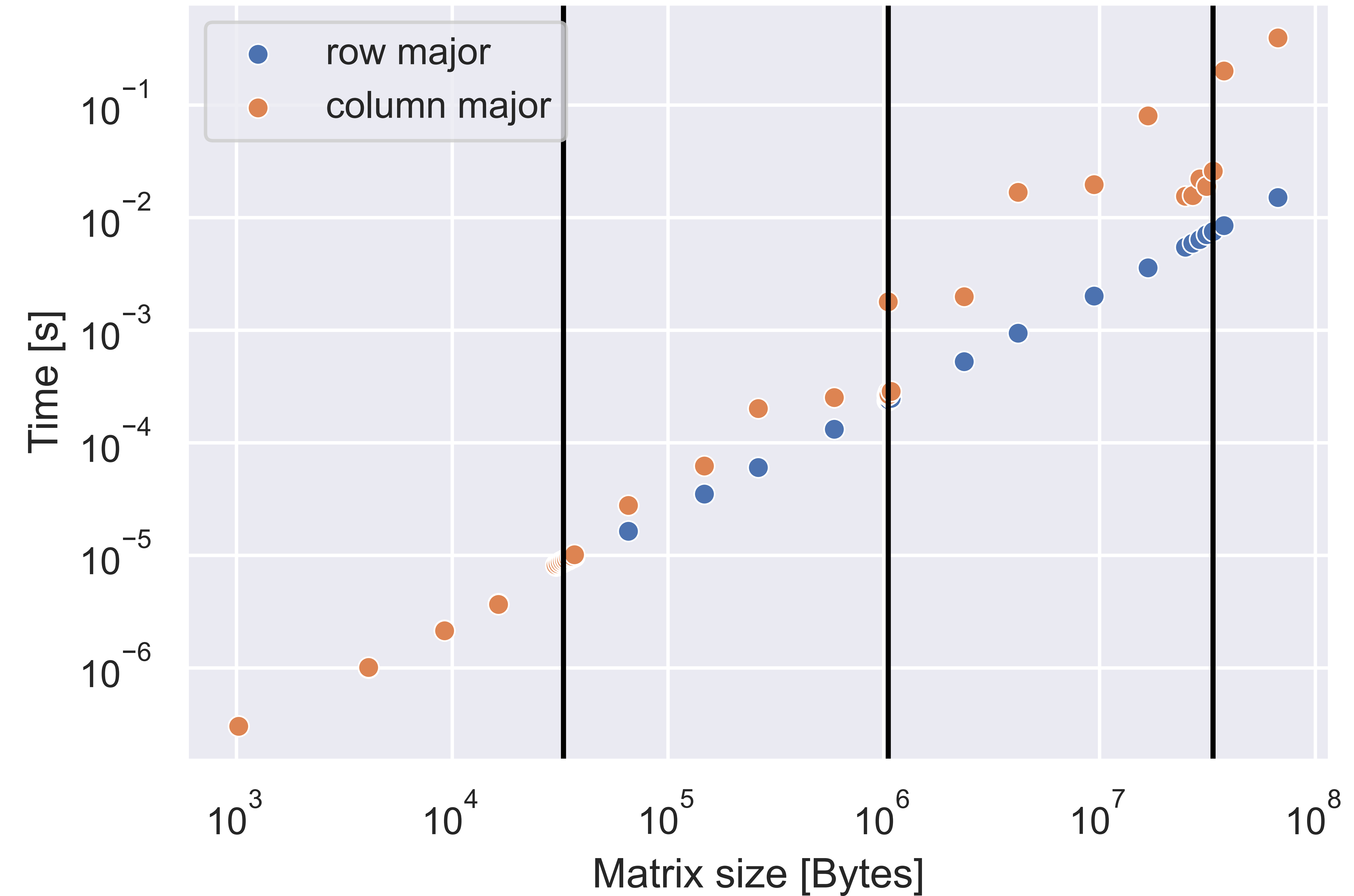
```
 3936675885 cycles
 4781854861 instructions
       1.21 ins/cycle
  154502459 cache refs
    2625165 cache misses
        1.7 % miss rate
```

Key Experiment: Matrix Reading



Key Experiment: Matrix Reading





Matrix Multiplication

Matrix Multiplication

Question: so what should you do for dense matrix multiplication?

Question: so what should you do for dense matrix multiplication?

- Create a transposed copy of the matrix that you access awkwardly before multiplying!

Matrix Multiplication

Question: so what should you do for dense matrix multiplication?

- Create a transposed copy of the matrix that you access awkwardly before multiplying!
- Or (of course): do not reimplement the wheel.

Striding Access

Striding:

Striding Access

Striding:

- Accessing `a[0], a[1], a[2],` …: stride of 4 bytes:

Striding:

- Accessing `a[0], a[1], a[2], …`: stride of 4 bytes:
  - accesses and uses every one of the 16 `int`s in each loaded cache line

Striding Access

Striding:

- Accessing `a[0], a[1], a[2],` ...: stride of 4 bytes:
  - accesses and uses every one of the 16 `int`s in each loaded cache line
  - no writing of partially updated cache lines

Striding Access

Striding:

- Accessing `a[0], a[1], a[2],` ...: stride of 4 bytes:
  - accesses and uses every one of the 16 `int`s in each loaded cache line
  - no writing of partially updated cache lines
  - prefetcher easily recognizes access pattern

Striding:

- Accessing `a[0], a[1], a[2],` …: stride of 4 bytes:
  - accesses and uses every one of the 16 `int`s in each loaded cache line
  - no writing of partially updated cache lines
  - prefetcher easily recognizes access pattern
- Accessing `a[i], a[N + i], a[2*N + i],` …: stride of `4*N` bytes

Striding Access

Striding:

- Accessing `a[0], a[1], a[2],` …: stride of 4 bytes:
  - accesses and uses every one of the 16 `int`s in each loaded cache line
  - no writing of partially updated cache lines
  - prefetcher easily recognizes access pattern
- Accessing `a[i], a[N + i], a[2*N + i],` …: stride of `4*N` bytes
  - accesses only one `int` in each cache line (6 % of each loaded cache line)

Striding Access

Striding:

- Accessing `a[0], a[1], a[2],` …: stride of 4 bytes:
  - accesses and uses every one of the 16 `int`s in each loaded cache line
  - no writing of partially updated cache lines
  - prefetcher easily recognizes access pattern
- Accessing `a[i], a[N + i], a[2*N + i],` …: stride of `4*N` bytes
  - accesses only one `int` in each cache line (6 % of each loaded cache line)
  - large distance between accesses: prefetcher cannot act on access pattern

Common Data Structures

Data Structure and Cache Efficiency:

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`):

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`):

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good

Common Data Structures

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`):

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`):

Common Data Structures

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better

Common Data Structures

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`):

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps:

Common Data Structures

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`):

Common Data Structures

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`): similar or worse than linked lists

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`): similar or worse than linked lists
- Binary search on vector:

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`): similar or worse than linked lists
- Binary search on vector: usually much better

Common Data Structures

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`): similar or worse than linked lists
- Binary search on vector: usually much better
- Double-ended queues (`std::deque`):

Common Data Structures

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`): similar or worse than linked lists
- Binary search on vector: usually much better
- Double-ended queues (`std::deque`): implementation-dependent

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`): similar or worse than linked lists
- Binary search on vector: usually much better
- Double-ended queues (`std::deque`): implementation-dependent
- Dynamic ring buffers:

Data Structure and Cache Efficiency:

- Dynamic array (`std::vector`): as good as it gets
- Linked list (`std::list`): allocator dependent, pointer chasing, usually not good
- Chaining hash maps (`std::unordered_map`): similar problems as linked lists
- Open-addressing hash maps (`boost::unordered_flat_map`): usually better
- Bitsets (`std::vector<bool>`): can be better depending on density and operations
- Array-backed binary heaps: usually decent
- Binary trees (`std::map`): similar or worse than linked lists
- Binary search on vector: usually much better
- Double-ended queues (`std::deque`): implementation-dependent
- Dynamic ring buffers: usually better

Structure of Arrays vs. Array of Structures

Striding:

Structure of Arrays vs. Array of Structures

Striding:

- At least the first of these issues also happens when iterating arrays of structures

Structure of Arrays vs. Array of Structures

Striding:

- At least the first of these issues also happens when iterating arrays of structures
- Unless we use most of the parts of the structure

Structure of Arrays vs. Array of Structures

Striding:

- At least the first of these issues also happens when iterating arrays of structures
- Unless we use most of the parts of the structure
- Fundamental issue in many situations:
  - Array of structures (AoS) is much cleaner from a software engineering perspective
  - Structure of arrays (SoA) can be more performant for several reasons

```
1  struct Particle {
2      float pos[2];
3      float speed[2];
4      float color[4];
5      float shinyness;
6      float rotation;
7      float age;
8      float springiness;
9      float density;
10     float area;
11     float thickness;
12     // ...
13 };
14 std::vector<Particle> particles;
```

```
1  struct Particles {
2      std::vector<std::array<float,2>> positions;
3      std::vector<std::array<float,2>> speeds;
4      std::vector<std::array<float,4>> colors;
5      std::vector<float> shinyness;
6      std::vector<float> rotation;
7      std::vector<float> age;
8      std::vector<float> springiness;
9      std::vector<float> density;
10     std::vector<float> area;
11     std::vector<float> thickness;
12     // ...
13 };
14 Particles particles;
```

What can the compiler do?

What can the compiler do?

Mostly low-level tricks:

What can the compiler do?

Mostly low-level tricks:

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, ...): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, ...): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅
- Changing loop order: sometimes ✅

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅
- Changing loop order: sometimes ✅
- Completely optimize away your microbenchmark: ✅ (dead code elimination)

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅
- Changing loop order: sometimes ✅
- Completely optimize away your microbenchmark: ✅ (dead code elimination)

But: your mileage may vary (depending on compiler, inlining site, moon phase, …)

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅
- Changing loop order: sometimes ✅
- Completely optimize away your microbenchmark: ✅ (dead code elimination)

But: your mileage may vary (depending on compiler, inlining site, moon phase, …)

The compiler cannot (usually):

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅
- Changing loop order: sometimes ✅
- Completely optimize away your microbenchmark: ✅ (dead code elimination)

But: your mileage may vary (depending on compiler, inlining site, moon phase, …)

The compiler cannot (usually):

- Change data structures (e.g., `vector` to `list`)

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅
- Changing loop order: sometimes ✅
- Completely optimize away your microbenchmark: ✅ (dead code elimination)

But: your mileage may vary (depending on compiler, inlining site, moon phase, …)

The compiler cannot (usually):

- Change data structures (e.g., `vector` to `list`)
- Change memory layout (e.g., AoS to SoA)

What can the compiler do?

Mostly low-level tricks:

- Branchless code (simple cases like `min`, `max`, `abs`, …): ✅
- Strength reduction (e.g., div-by-constant to mul+shift): ✅
- Common subexpression elimination, hoisting, constant folding, inlining: ✅
- Auto-vectorizing (simple to medium-complexity) loops: ✅
- Changing loop order: sometimes ✅
- Completely optimize away your microbenchmark: ✅ (dead code elimination)

But: your mileage may vary (depending on compiler, inlining site, moon phase, …)

The compiler cannot (usually):

- Change data structures (e.g., `vector` to `list`)
- Change memory layout (e.g., AoS to SoA)
- Restructure/change algorithms

Summary

CPUs are ridiculously complex to avoid waiting

Summary

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

Summary

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

- CPU attempts branch prediction; predictable branches are very cheap

Summary

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

- CPU attempts branch prediction; predictable branches are very cheap
- Random/unpredictable branches in hot loops: massive slowdown

Summary

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

- CPU attempts branch prediction; predictable branches are very cheap
- Random/unpredictable branches in hot loops: massive slowdown
- **Don’t overdo it:** measure first — most branches are fine!

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

- CPU attempts branch prediction; predictable branches are very cheap
- Random/unpredictable branches in hot loops: massive slowdown
- **Don’t overdo it:** measure first — most branches are fine!

Memory access:

Summary

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

- CPU attempts branch prediction; predictable branches are very cheap
- Random/unpredictable branches in hot loops: massive slowdown
- **Don’t overdo it:** measure first — most branches are fine!

Memory access:

- Main memory is slow, caches are fast

Summary

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

- CPU attempts branch prediction; predictable branches are very cheap
- Random/unpredictable branches in hot loops: massive slowdown
- **Don’t overdo it:** measure first — most branches are fine!

Memory access:

- Main memory is slow, caches are fast
- Predictable, ideally sequential memory access pattern, avoiding striding

Summary

CPUs are ridiculously complex to avoid waiting

The **two main issues** you should be aware of: **branching** and **memory access**

Branching:

- CPU attempts branch prediction; predictable branches are very cheap
- Random/unpredictable branches in hot loops: massive slowdown
- **Don’t overdo it:** measure first — most branches are fine!

Memory access:

- Main memory is slow, caches are fast
- Predictable, ideally sequential memory access pattern, avoiding striding
- Know & choose the right data structures & memory layout

Optimization Hierarchy

Optimization Hierarchy

Correctness comes first! Then:

- choose the right algorithm, then
- choose the right data structures and memory layout, then
- measure performance, help branch prediction, etc.

Reading

Optional further reading for a deeper dive:

- Bakhvalov, Denis. 2024.
  Performance Analysis and Tuning on Modern CPUs. 2nd (in progress).
- Drepper, Ulrich. 2007. What Every Programmer Should Know about Memory.
- Fog, Agner. 2024. Optimizing Software in C++:
  An Optimization Guide for Windows, Linux, and Mac Platforms.
- Hennessy, John L., and David A. Patterson. 2019. Computer Architecture: A Quantitative Approach. 6th ed. Morgan Kaufmann.
- Kocher, Paul, Jann Horn, Anders Fogh, et al. 2019. “Spectre Attacks: Exploiting Speculative Execution.” IEEE Symposium on Security and Privacy (s&p).
- Seznec, André. 2011. “A New Case for the TAGE Branch Predictor.” Proc. Of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).