Performance

We are here

- Part 1: How computers works
  - Boolean logic, integers
  - Instructions
  - Memory
- Part 2: Software development
  - Compiling (clang, make, …)
  - Code management (regex, git)
- Part 3: Correctness
  - ABIs, Specifications
  - Static & dynamic analysis, debugging
- Part 4: Performance
  - CPU pipelines, caches ⟵ TODAY
  - Parallel computation

Performance

What is performance?

We want computers to perform required actions while minimizing their use of resources:

- Time
- Power use (mobile, servers)
- Network use (mobile)
- Memory use (peak and average RAM usage)
- Storage (solid state drive / hard disk drive)

We care about those resources in proportion to their cost (financial, environmental, …)

How do we achieve good performance?

High level

1. Pick a good algorithm
   - specifically, an algorithm with low computational complexity:
     $O(\log(n))$ better than $O(n^2)$ better than $O(n^6)$ better than $O(2^n)$
   - we can hope for great improvements, $100\times$ faster, $1000\times$, etc.
     $\rightarrow$ first thing to try!

2. Pick an algorithm that is fast on the computers you use and implement it well
   - this course!
   - smaller improvements: from a few percent to $50\times$ faster

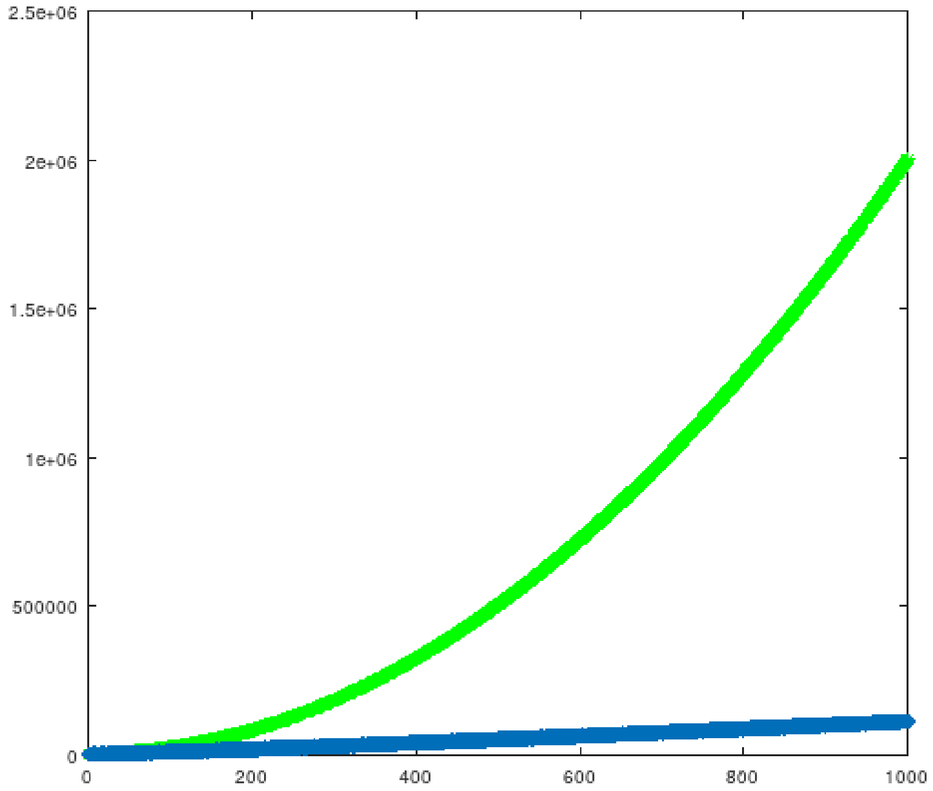
Low level

3. Translate the implementation into efficient instructions (compilers do that well)

What hides inside the big-O?

Let us compare two algorithms:

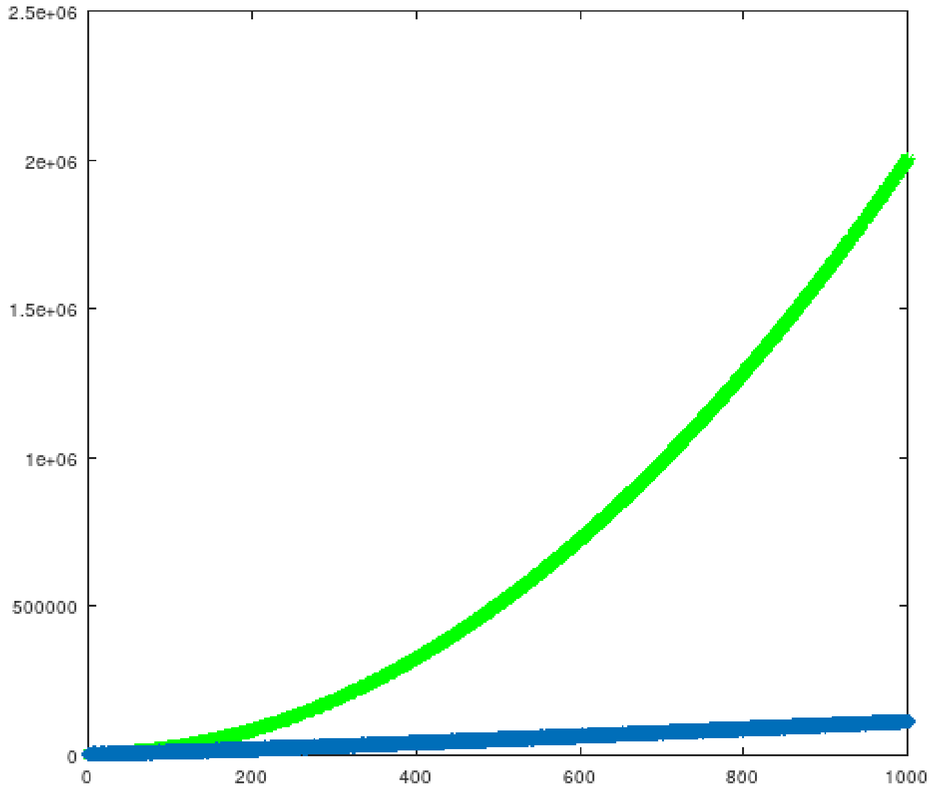
- Algorithm A has complexity $O(n^2)$
- Algorithm B has complexity $O(n \log(n))$



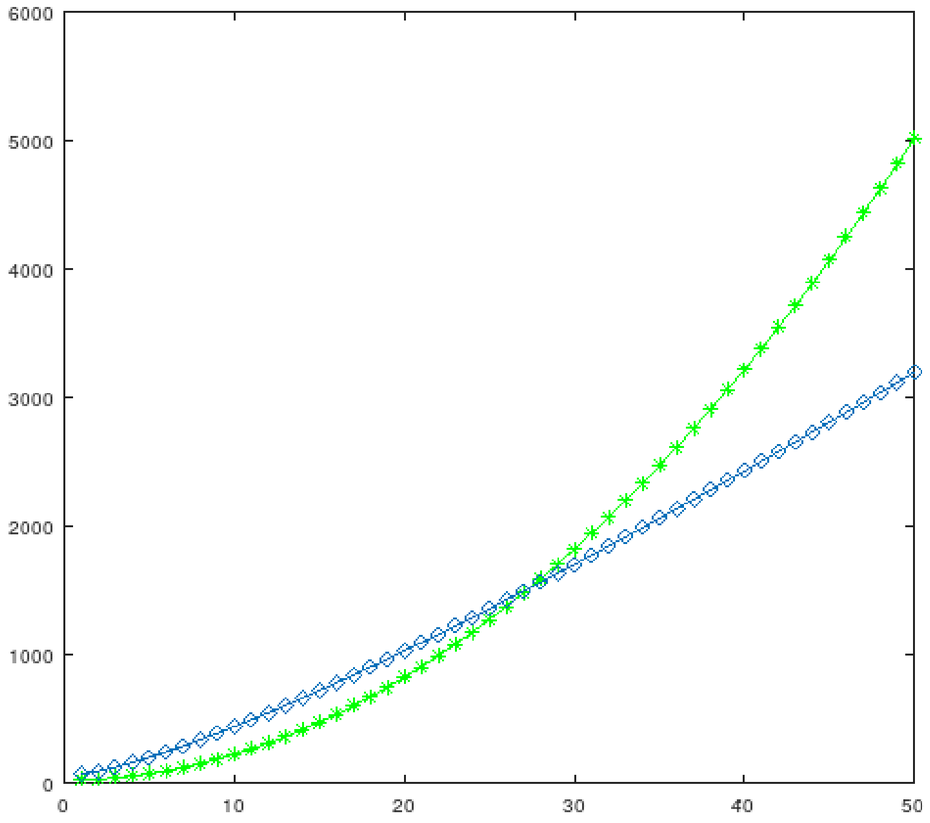
Algorithm A	Algorithm B
$O(n^2)$	$O(n \log(n))$

Specifically,

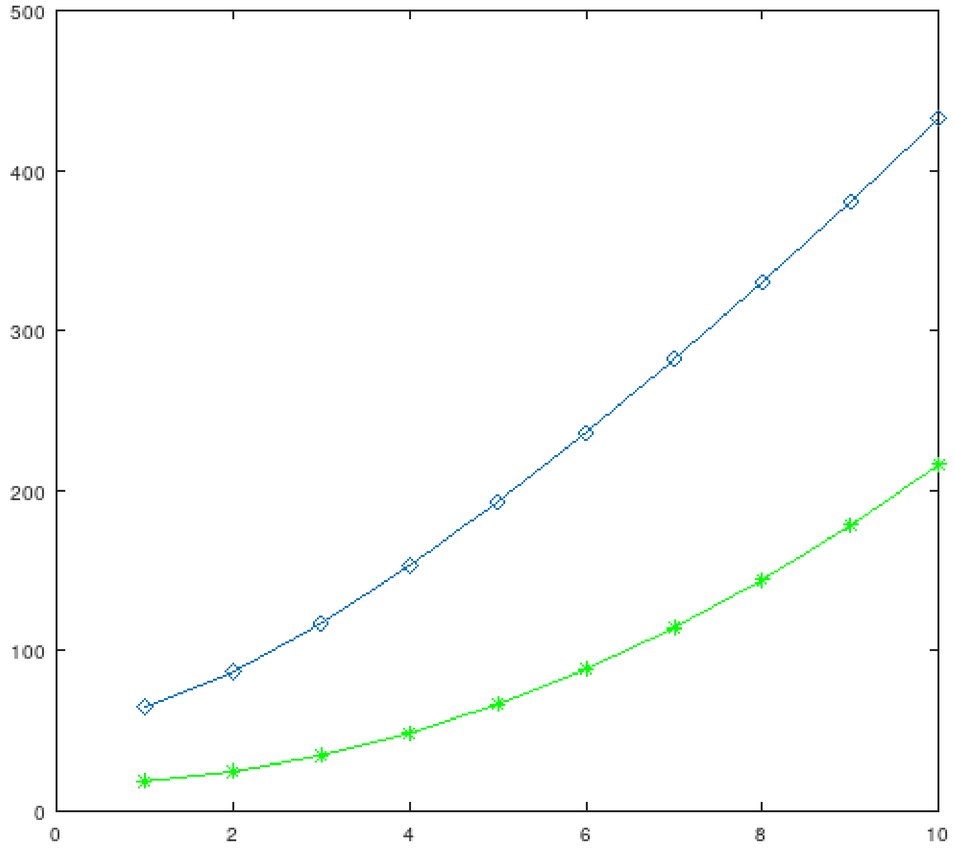
- Algorithm A performs $2n^2 + 16$ operations
- Algorithm B performs $16n \log(n) + 64$ operations



Algorithm A	Algorithm B
$2n^2 + 16$	$16n\log(n) + 64$



Algorithm A	Algorithm B
$2n^2 + 16$	$16n\log(n) + 64$

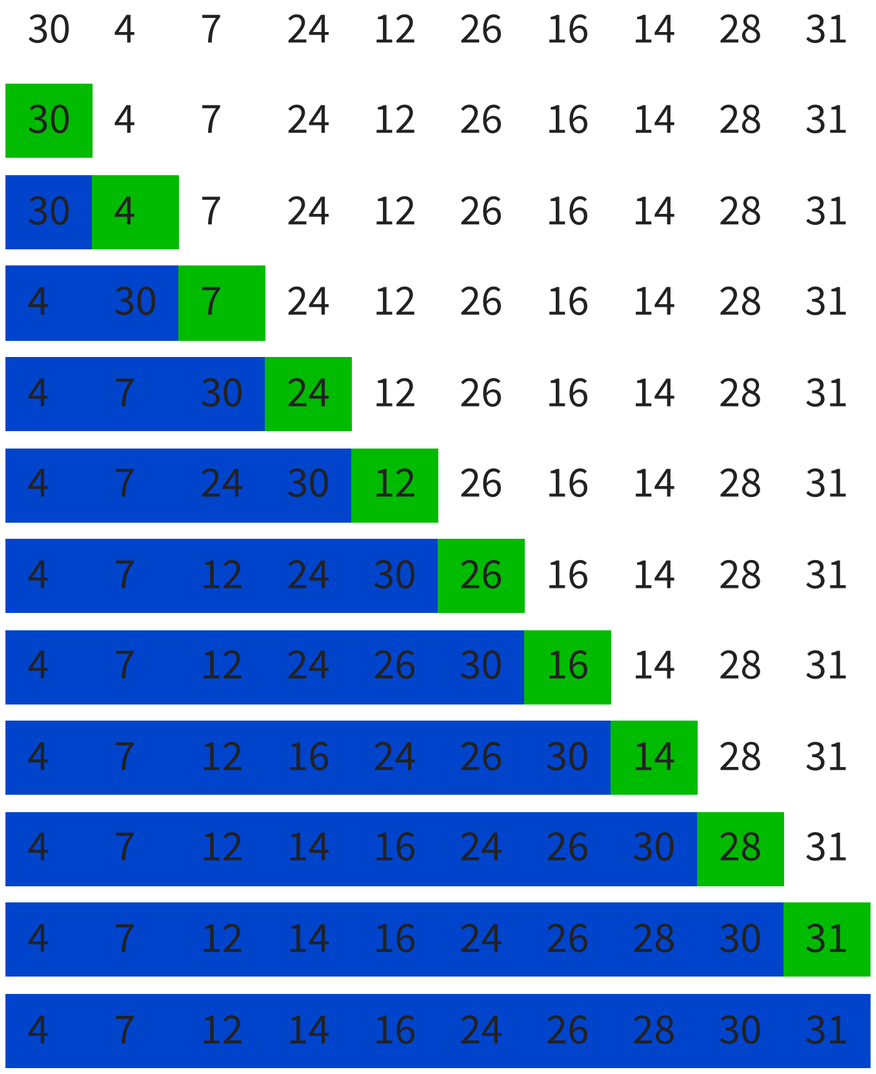


Algorithm A	Algorithm B
$2n^2 + 16$	$16n\log(n) + 64$

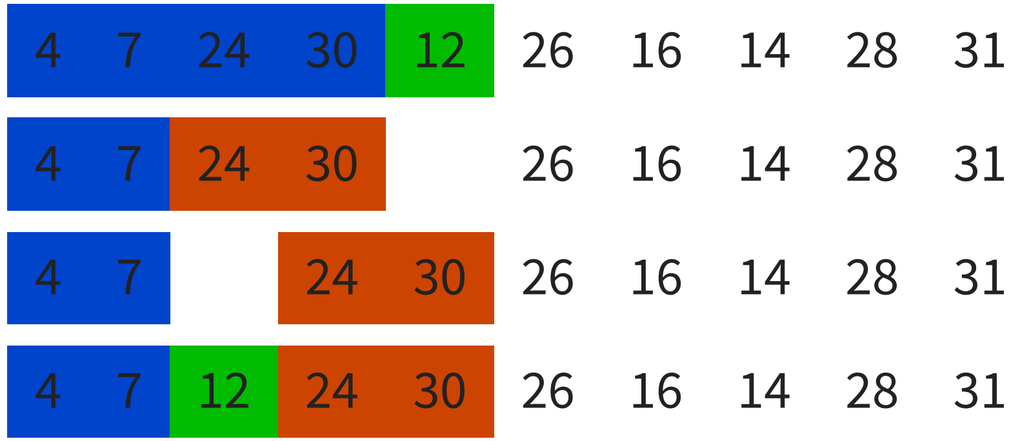
Which algorithm do we choose?

- Algorithm A is an implementation of insertion sort
- Algorithm B is an implementation of merge sort

Insertion sort



Insertion operation

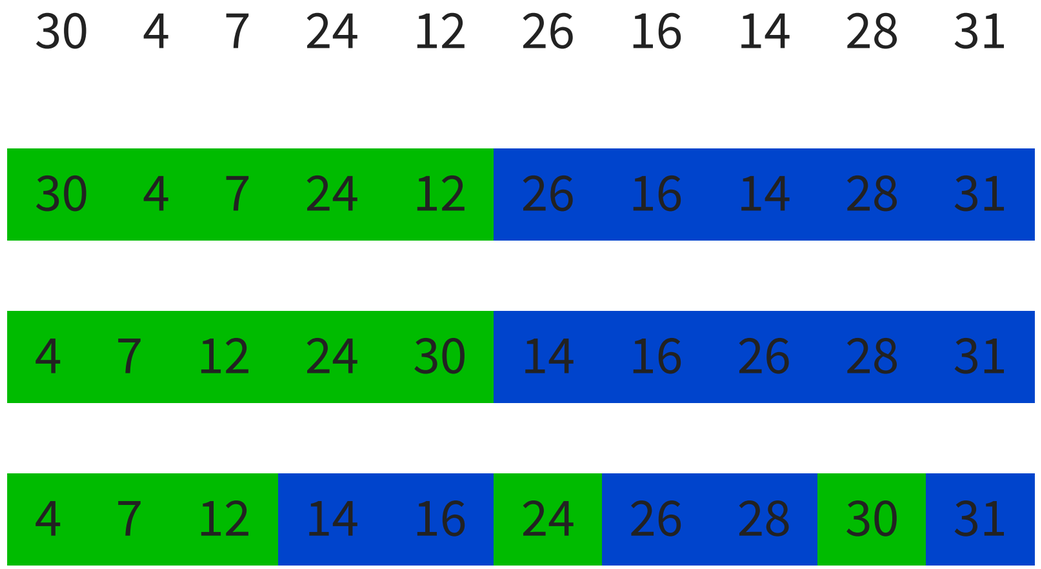


Each insertion has complexity $O(n)$.

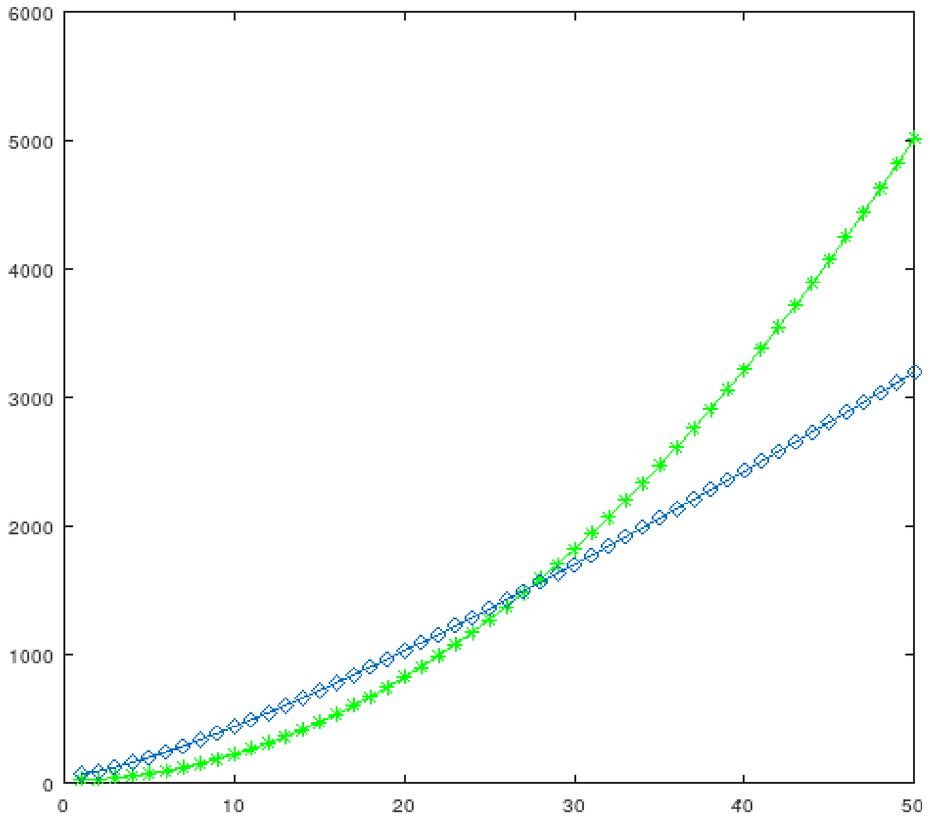
We have n insertions $\Rightarrow$ total complexity $O(n^2)$.

```python
def insertion_sort(a):
    r = list(a)

    for i in range(len(r)):
        piv = r[i]

        j = i
        while j > 0 and r[j - 1] > piv:
            r[j] = r[j - 1]
            j -= 1

        r[j] = piv

    return r
```

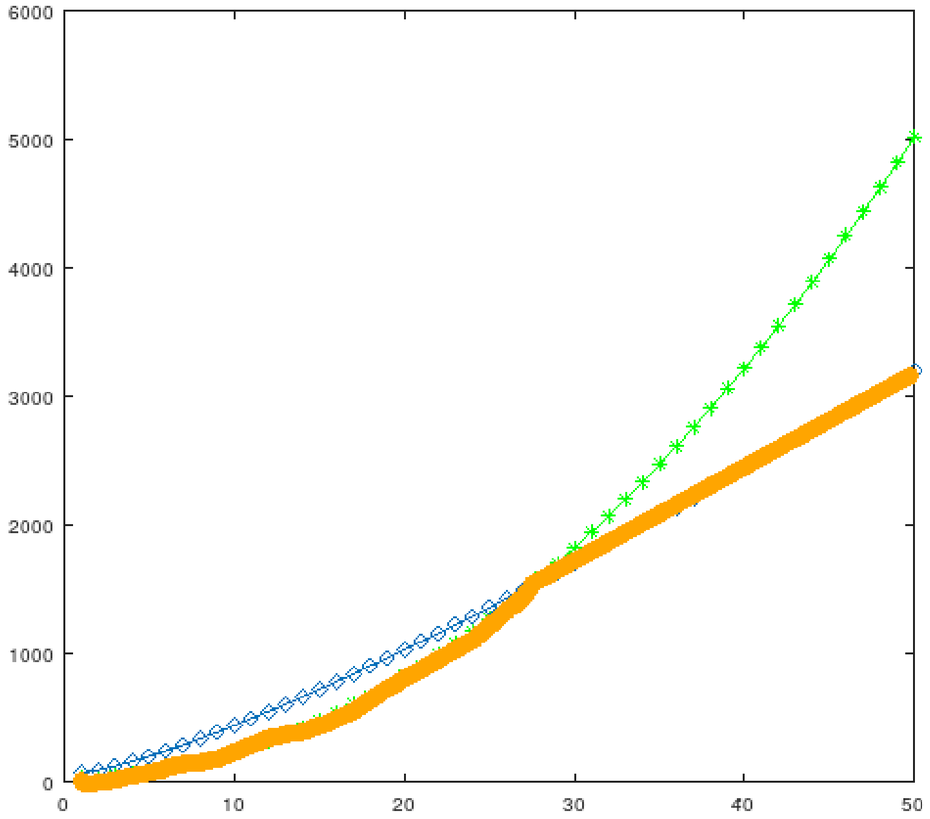
Merge sort example



```python
def merge_sort(a):

    # single-element list
    if len(a) <= 1:
        return a

    # split list, sort each part
    pivot = len(a) // 2

    part1 = merge_sort(a[:pivot])
    part2 = merge_sort(a[pivot:])

    # merge parts
    r = []
    i1 = 0
    i2 = 0

    while i1 < len(part1) or i2 < len(part2):
        if i1 < len(part1) and (i2 == len(part2) or part1[i1] <= part2[i2]):
            r.append(part1[i1])
            i1 += 1
        else:
            r.append(part2[i2])
            i2 += 1

    return r
```



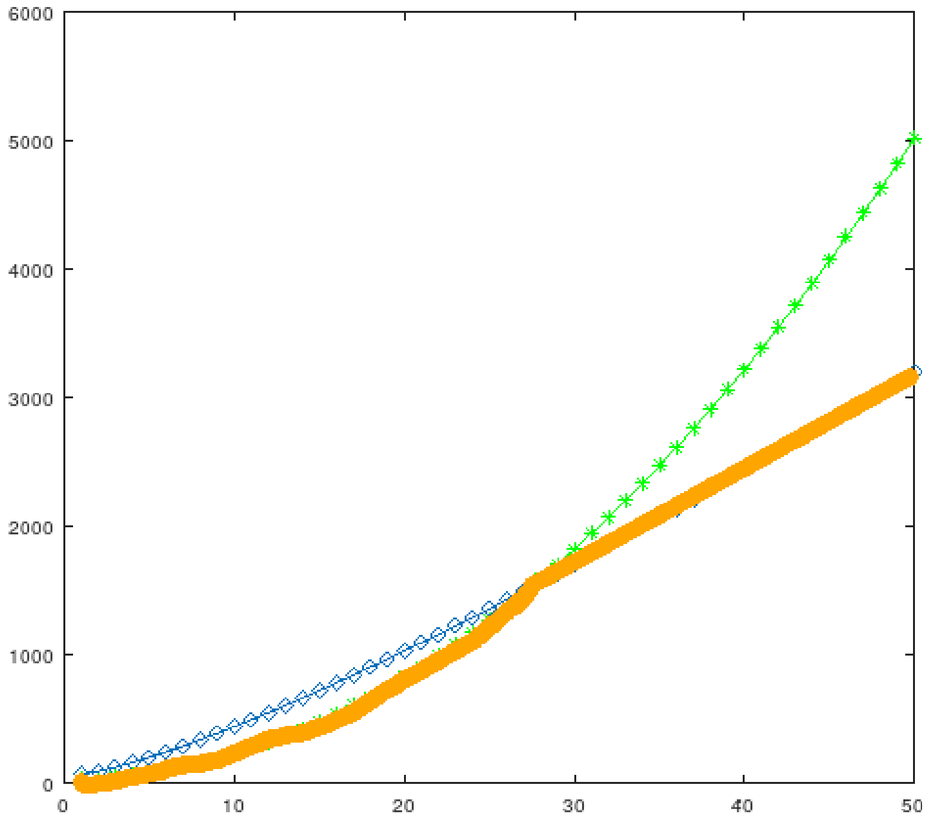
Insertion sort	Merge sort
$2n^2 + 16$	$16n \log(n) + 64$



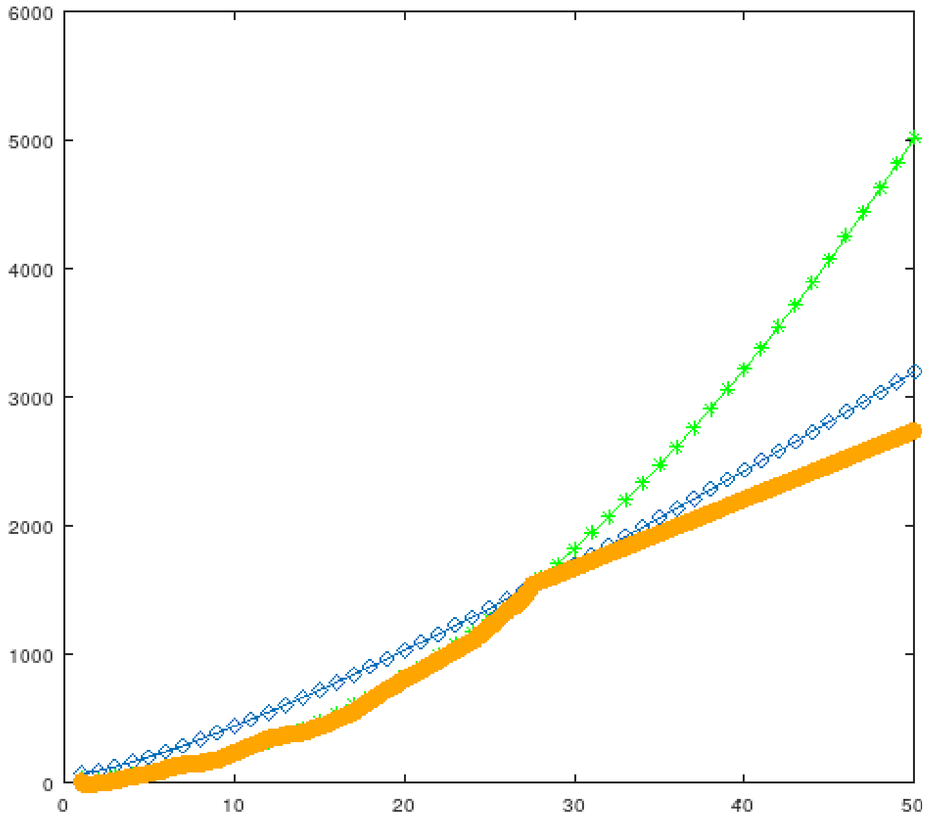
$\longrightarrow$ we should pick the right algorithm for each value of n

```python
def merge_sort(a):

    # single-element list
    if len(a) <= 1:
        return a

    # split list, sort each part
    pivot = len(a) // 2

    part1 = merge_sort(a[:pivot])
    part2 = merge_sort(a[pivot:])

    # merge parts
    r = []
    i1 = 0
    i2 = 0

    while i1 < len(part1) or i2 < len(part2):
        if i1 < len(part1) and (i2 == len(part2) or part1[i1] <= part2[i2]):
            r.append(part1[i1])
            i1 += 1
        else:
            r.append(part2[i2])
            i2 += 1

    return r
```

```
def merge_sort(a):

    # short list
    if (len(a) <= 28):
        return insertion_sort(a)

    # split list, sort each part
    pivot = len(a) // 2

    part1 = merge_sort(a[:pivot])
    part2 = merge_sort(a[pivot:])

    # merge parts
    r = []
    i1 = 0
    i2 = 0

    while i1 < len(part1) or i2 < len(part2):
        if i1 < len(part1) and (i2 == len(part2) or part1[i1] <= part2[i2]):
            r.append(part1[i1])
            i1 += 1
        else:
            r.append(part2[i2])
            i2 += 1

    return r
```



We could pick the right algorithm for each value of n…



Here, we can do even better: we can combine algorithms

This is how sort algorithms are implemented in practice

CPU pipelines

Back to instructions

Instruction decoding:

```
48 2b 06              sub    rax, QWORD PTR [rsi]
48 0f af 06           imul   rax, QWORD PTR [rsi]
48 0f af 02           imul   rax, QWORD PTR [rdx]
```

From `48 0f af 06`, the CPU needs to understand:

- that it must perform a multiplication (as opposed to, say, a subtraction)
- that one term is the value of a 64-bit register, `rax`
- that the other term comes from memory: the 64-bit value pointed to by `rsi`

What happens after instruction decoding?

```
48 0f af 06          imul   rax, QWORD PTR [rsi]
```

- the CPU has Boolean circuitry to compute multiplications
- it must ensure that one of the two inputs of the multiplier is `rax`
- the CPU has Boolean circuitry to access memory
- it must ensure that the input of the memory circuitry is `rsi`
- it sets the second input of the multiplier to the output of the memory circuitry
- it stores the output of the multiplier back to `rax`

It is no longer possible to do all this in a single cycle

(e.g. at 4 GHz, i.e. 4 billion cycles per second, so in 0.25 ns)

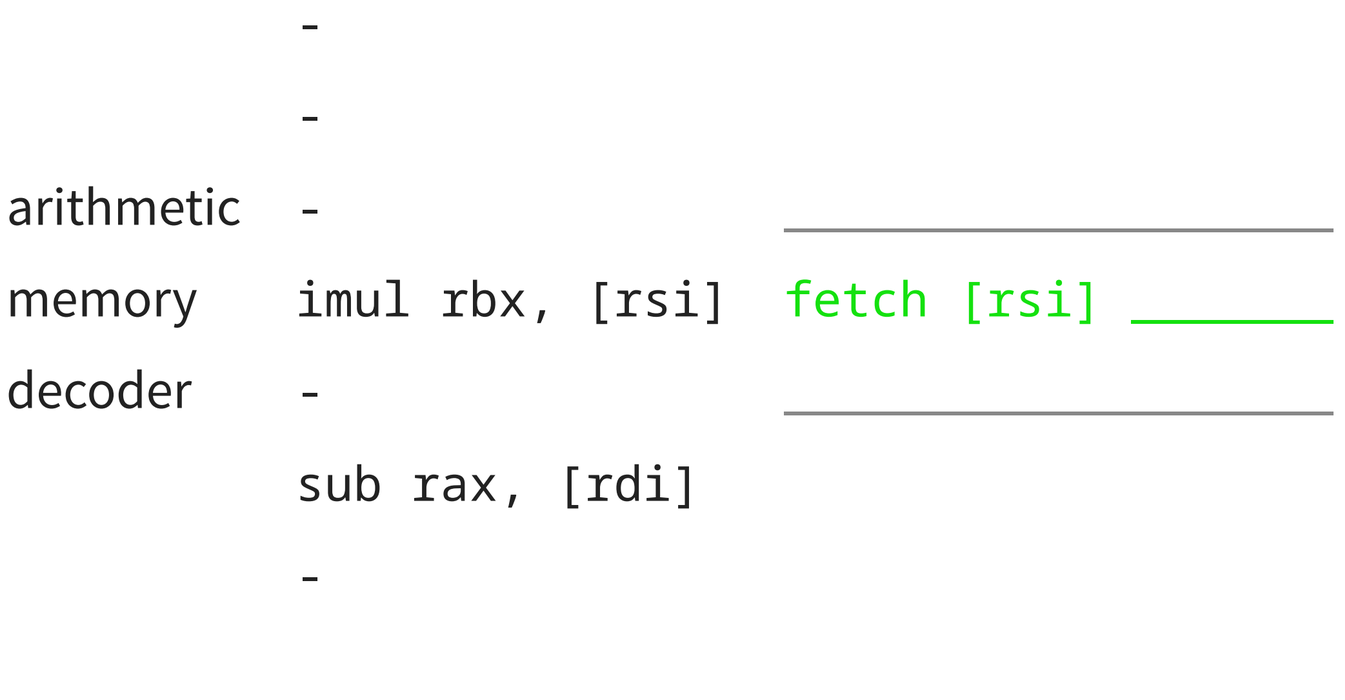
Take an imaginary computer in which it takes:

- one cycle to decode an instruction
- one cycle to fetch data from memory
- one cycle to perform arithmetic

	-	
	-	
arithmetic	-	
memory	-	
decoder	-	

```
imul rbx, [rsi]
sub rax, [rdi]
```

```
                -
                -
arithmetic      -                    ____________________
memory          -                    ____________________
decoder         imul rbx, [rsi]      decode "imul" _____
                sub rax, [rdi]
                -
```


-
-
arithmetic -
memory imul rbx, [rsi]
fetch [rsi]
decoder -
sub rax, [rdi]
-

```
                -
                -
arithmetic      imul rbx, [rsi]     compute rbx * [rsi]
memory          -                   ____________________
decoder         -                   ____________________
                sub rax, [rdi]
                -
```

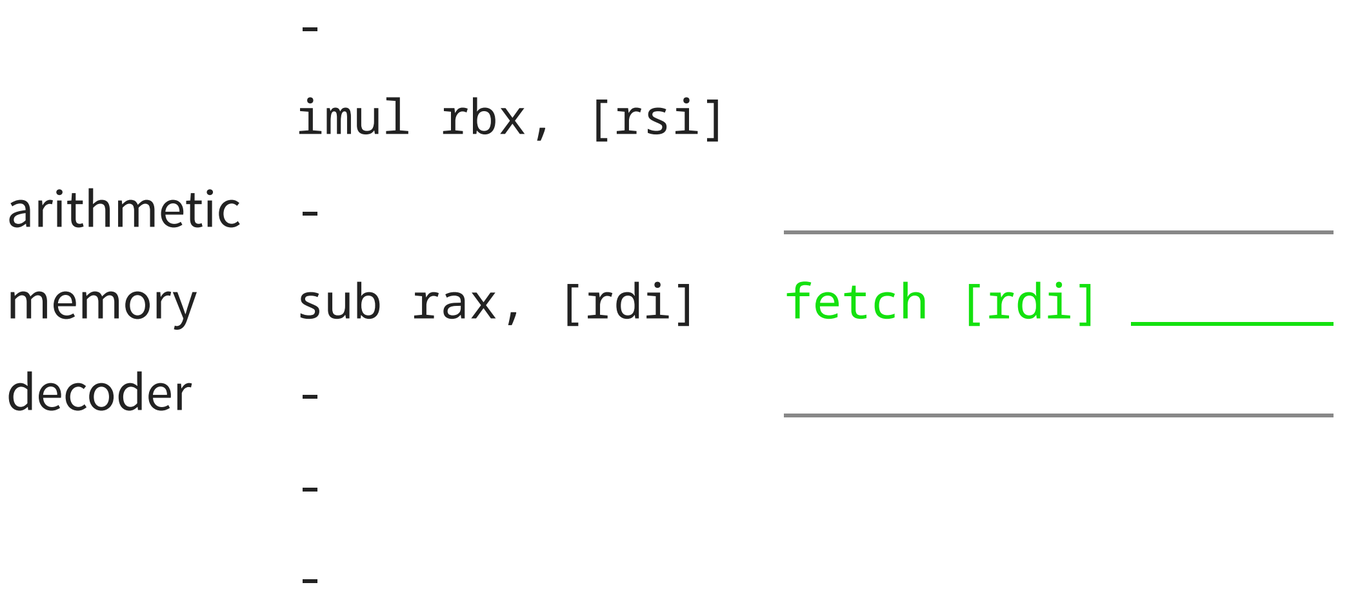
```
-
imul rbx, [rsi]
-
-
sub rax, [rdi]
-
```

arithmetic

memory

decoder

	-	
	imul rbx, [rsi]	
arithmetic	-	_____
memory	-	_____
decoder	sub rax, [rdi]	decode "sub" _____
	-	
	-	


-
imul rbx, [rsi]
arithmetic
-
memory
sub rax, [rdi]
fetch [rdi]
decoder
-
-

	-	
	imul rbx, [rsi]	
arithmetic	sub rax, [rdi]	compute rax - [rdi]
memory	-	_____
decoder	-	_____
	-	
	-	

```
            imul rbx, [rsi]
            sub rax, [rdi]
```

arithmetic	-	
memory	-	
decoder	-	
	-	
	-	

Each instruction takes 3 cycles

However, in this model,

- while the memory circuitry is busy fetching `QWORD PTR [rsi]`, the multiplier is idle
- while the multiplier computes the result, the memory is idle
- during instruction decoding, everything else is idle

We can exploit this!

Pipelined execution

	-	
	-	
	-	
arithmetic	-	(idle)_____
memory	-	(idle)_____
decoder	-	(idle)_____
	-	

```
add rcx, [rbp]
imul rbx, [rsi]
sub rax, [rdi]
```

Pipelined execution

	-	
	-	
	-	
arithmetic	-	(idle)_____
memory	-	(idle)_____
decoder	-	(idle)_____
	`add rcx, [rbp]`	
	`imul rbx, [rsi]`	
	`sub rax, [rdi]`	
	-	

Pipelined execution

	-	
	-	
	-	
arithmetic	-	(idle)_____
memory	-	(idle)_____
decoder	add rcx, [rbp]	decode "add" _____
	imul rbx, [rsi]	
	sub rax, [rdi]	
	-	
	-	

Pipelined execution

	-	
	-	
	-	
arithmetic	-	(idle)_____
memory	add rcx, [rbp]	fetch [rbp] _____
decoder	imul rbx, [rsi]	decode "imul" _____
	sub rax, [rdi]	
	-	
	-	
	-	

Pipelined execution

```
                -
                -
                -
arithmetic      add rcx, [rbp]    compute rcx + [rbp] _
memory          imul rbx, [rsi]   fetch [rsi] __________
decoder         sub rax, [rdi]    decode "sub" _________
                -
                -
                -
                -
```

Pipelined execution

	-	
	-	
	add rcx, [rbp]	
arithmetic	imul rbx, [rsi]	compute rbx + [rsi] _
memory	sub rax, [rdi]	fetch [rdi] _____
decoder	-	(idle) _____
	-	
	-	
	-	
	-	

Pipelined execution

	-	
	add rcx, [rbp]	
	imul rbx, [rsi]	
arithmetic	sub rax, [rdi]	compute rax + [rdi] _
memory	-	(idle) _____
decoder	-	(idle) _____
	-	
	-	
	-	
	-	

Pipelined execution

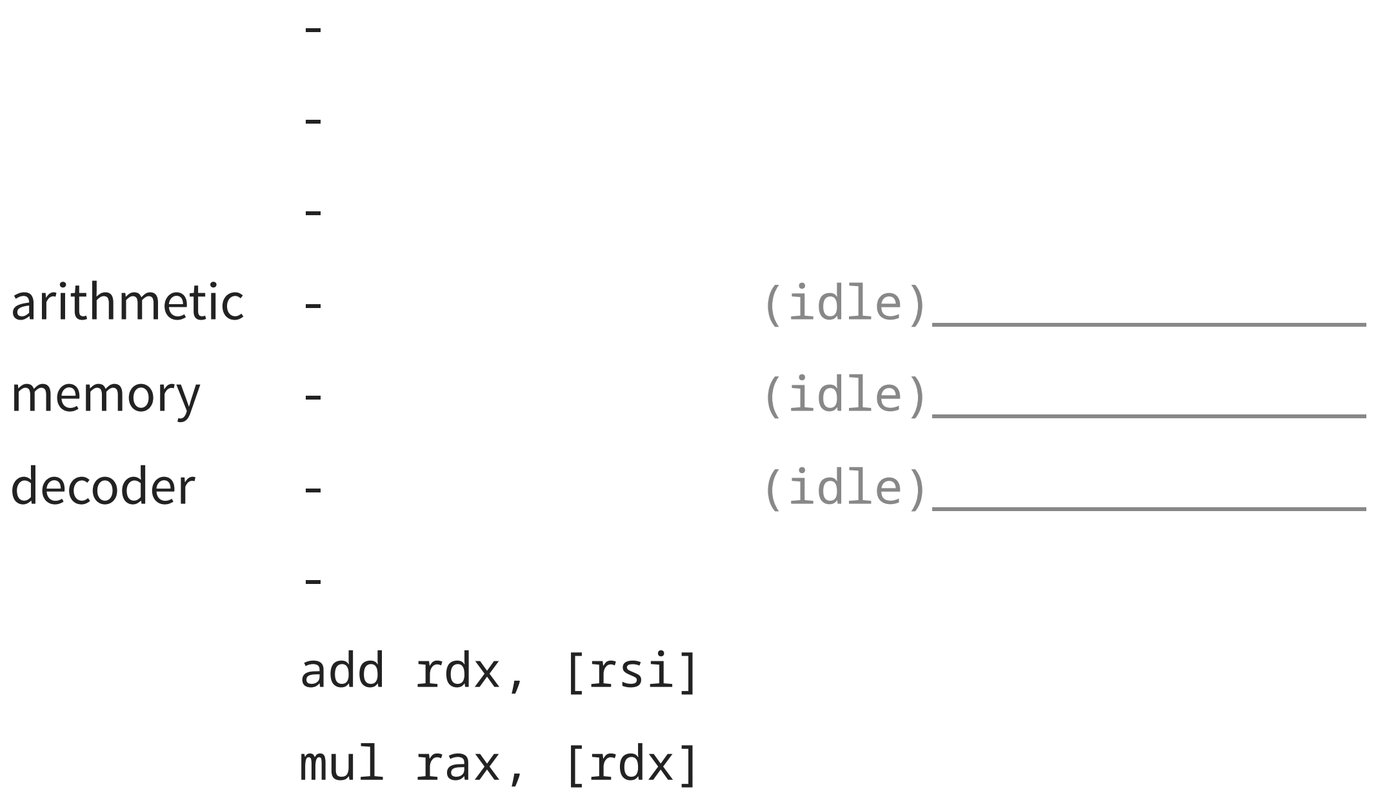
```
                add rcx, [rbp]
                imul rbx, [rsi]
                sub rax, [rdi]
arithmetic      -                   (idle) ________________
memory          -                   (idle) ________________
decoder         -                   (idle) ________________
                -
                -
                -
                -
```

Throughput vs. latency

- Latency:
  - Executing each instruction still takes 3 cycles!

- Throughput:
  - But on average, we execute up to 1 instruction per cycle.

Data dependencies



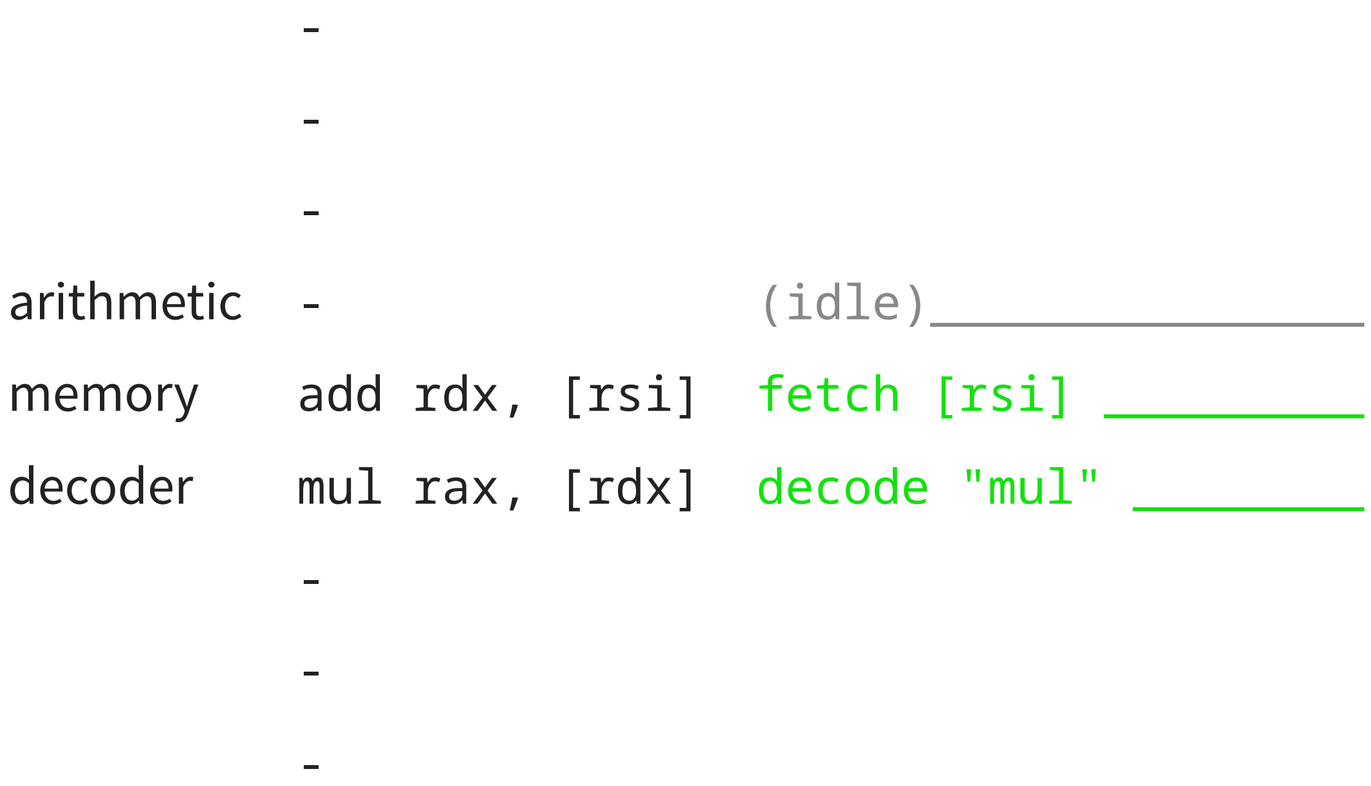
Data dependencies

```
                -
                -
                -
arithmetic      -                (idle)_________________
memory          -                (idle)_________________
decoder         -                (idle)_________________
                add rdx, [rsi]
                mul rax, [rdx]
                -
```

Data dependencies

```
                -
                -
                -
arithmetic      -                (idle)_________________
memory          -                (idle)_________________
decoder         add rdx, [rsi]   decode "add" _________
                mul rax, [rdx]
                -
                -
```

Data dependencies



Data dependencies

	-	
	-	
	-	
arithmetic	`add rdx, [rsi]`	add [rsi] to rdx ____
memory	`mul rax, [rdx]`	(rdx not ready) _____
decoder	-	(idle) _____
	-	
	-	
	-	

Data dependencies

	-	
	-	
	add rdx, [rsi]	
arithmetic	-	(idle) _____
memory	mul rax, [rdx]	fetch [rdx] _____
decoder	-	(idle) _____
	-	
	-	
	-	

Data dependencies

```
                -
                add rdx, [rsi]
                -
arithmetic      mul rax, [rdx]   compute rax * [rdx] _
memory          -                (idle) _____________
decoder         -                (idle) _____________
                -
                -
                -
```

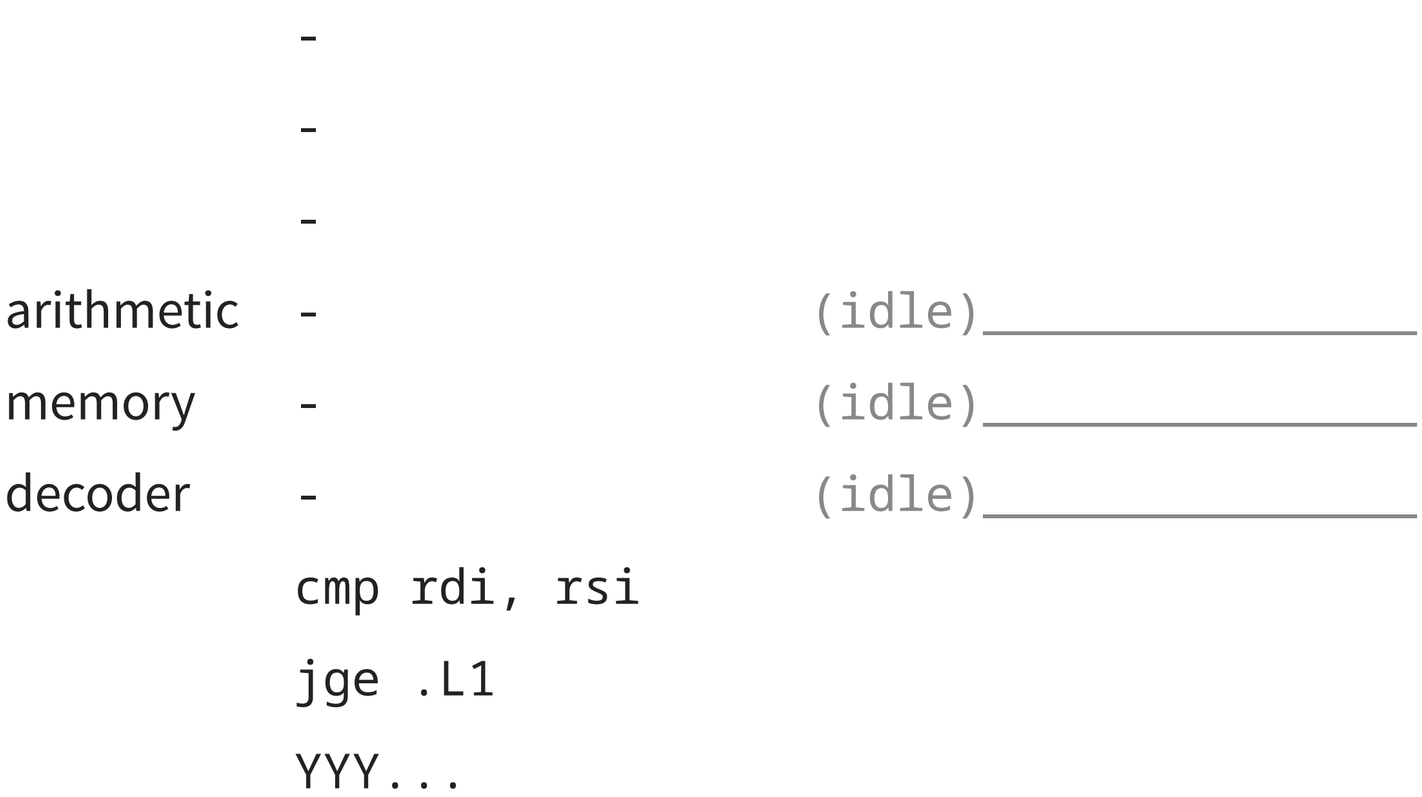
Data dependencies

```
                add rdx, [rsi]
                -
                mul rax, [rdx]
arithmetic      -               (idle)_________________
memory          -               (idle) _______________
decoder         -               (idle) _______________
                -
                -
                -
```

Conditional branching

```
if (a < b) {
    YYY...
}
ZZZ...
```

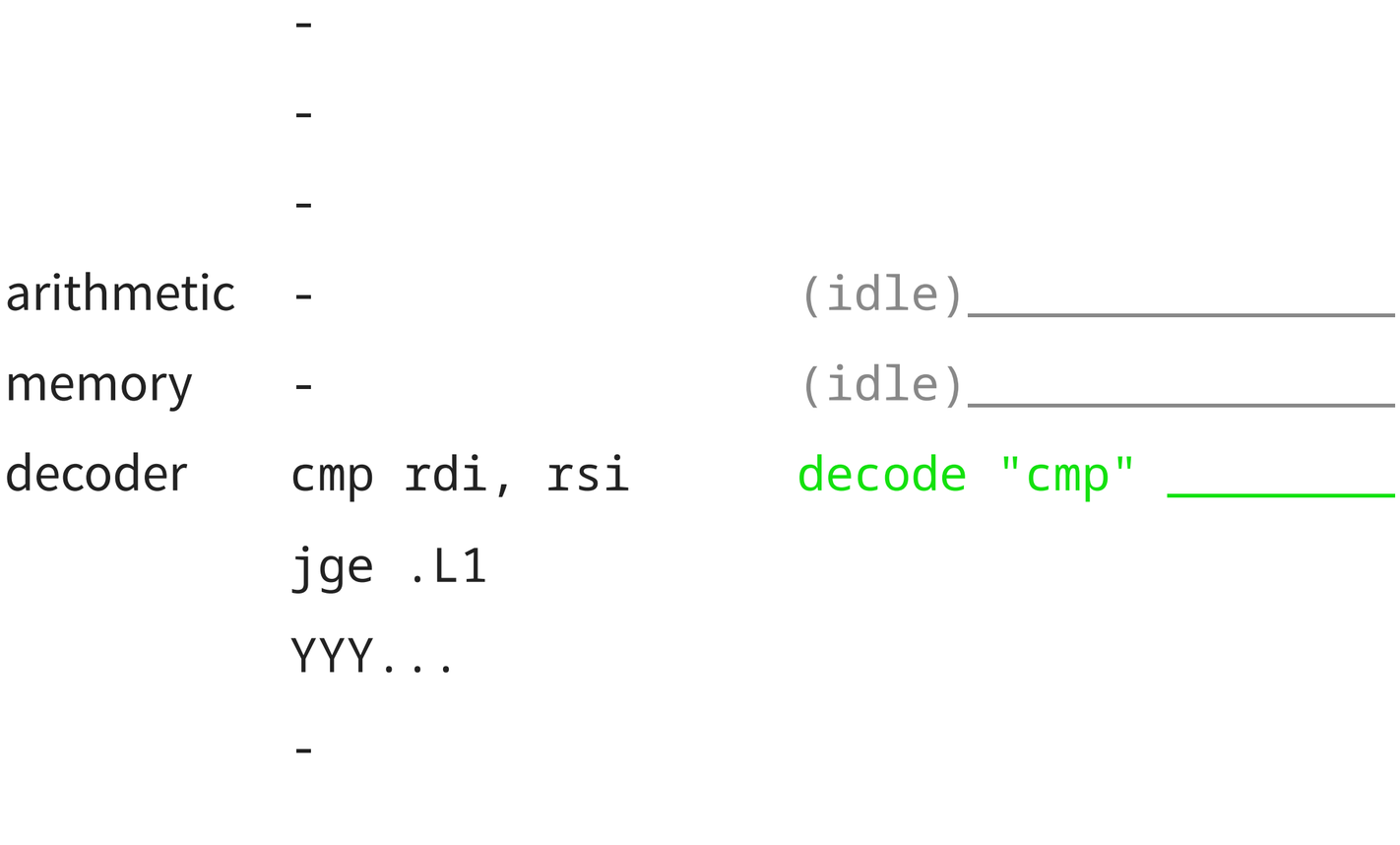
```
        cmp     rdi, rsi
        jge     .L1
        YYY...
.L1:
        ZZZ...
```



Conditional branching

```
if (a < b) {
    YYY...
}
ZZZ...
```

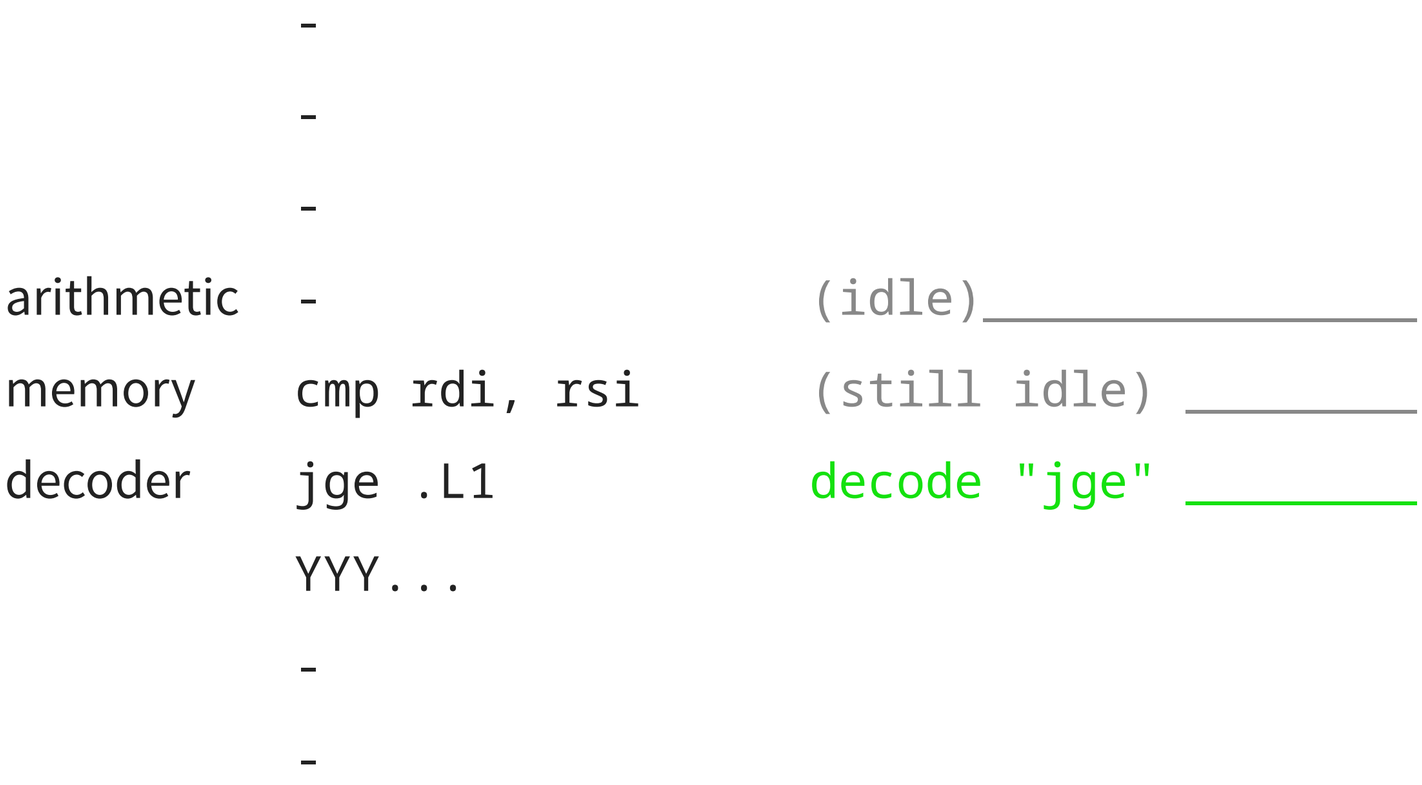
```
        cmp     rdi, rsi
        jge     .L1
        YYY...
.L1:
        ZZZ...
```



Conditional branching

```
if (a < b) {
    YYY...
}
ZZZ...
```

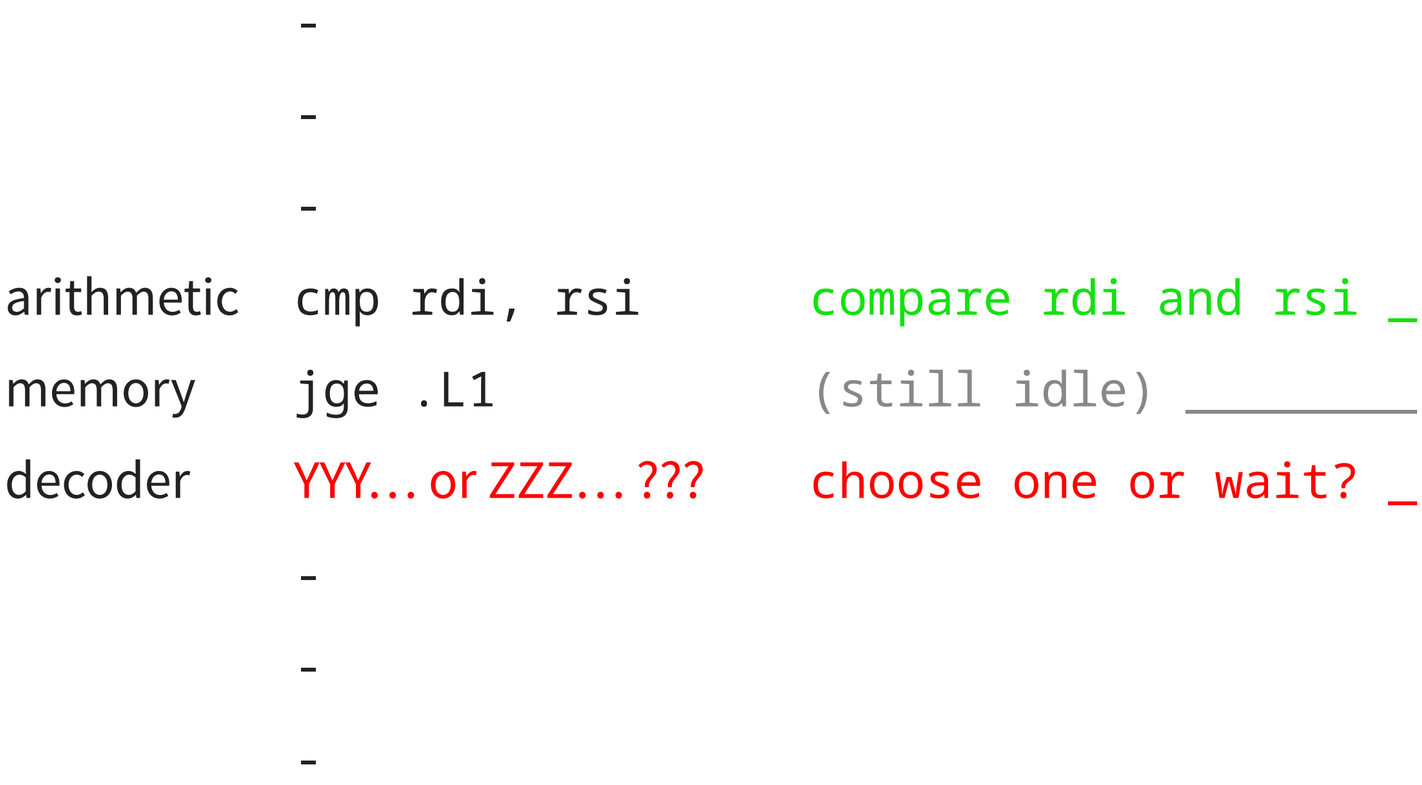
```
        cmp     rdi, rsi
        jge     .L1
        YYY...
.L1:
        ZZZ...
```



Conditional branching

```
if (a < b) {
    YYY...
}
ZZZ...
```

```
        cmp     rdi, rsi
        jge     .L1
        YYY...
.L1:
        ZZZ...
```

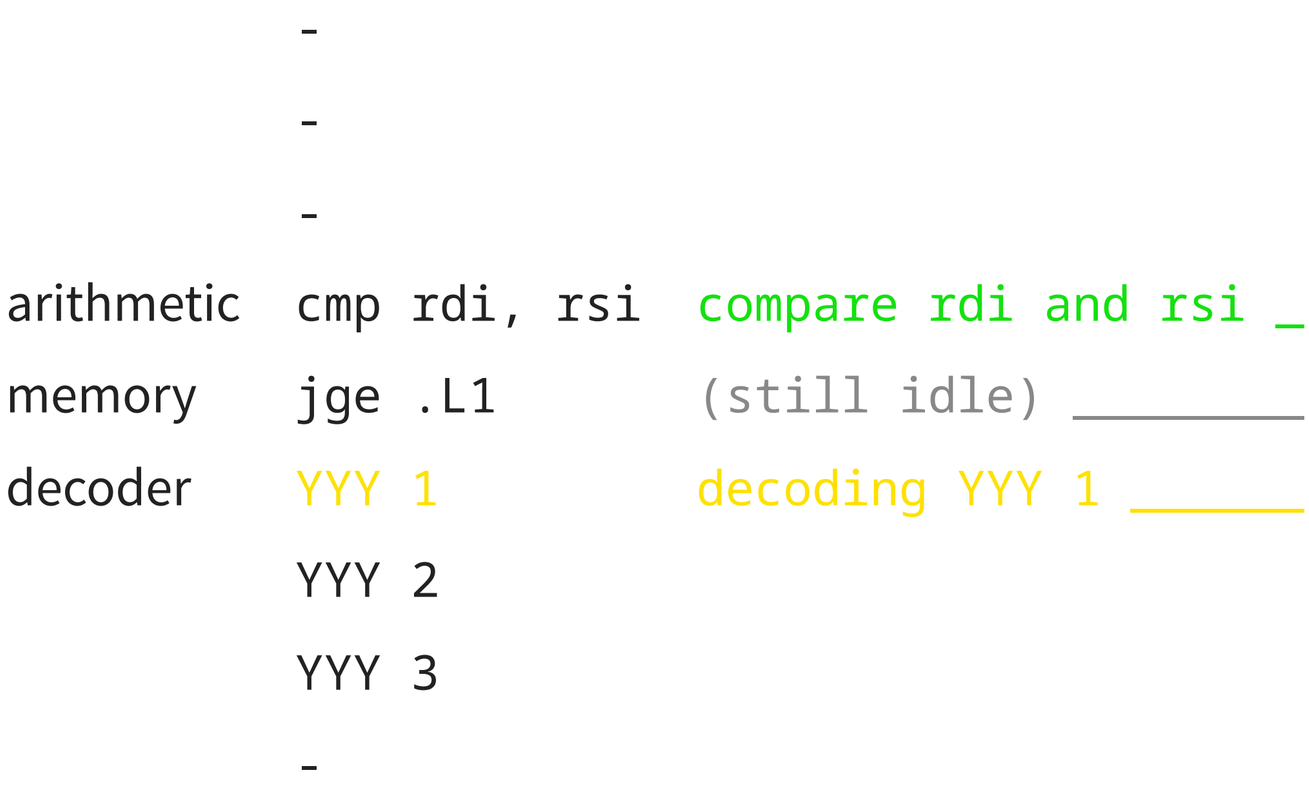


Branch prediction

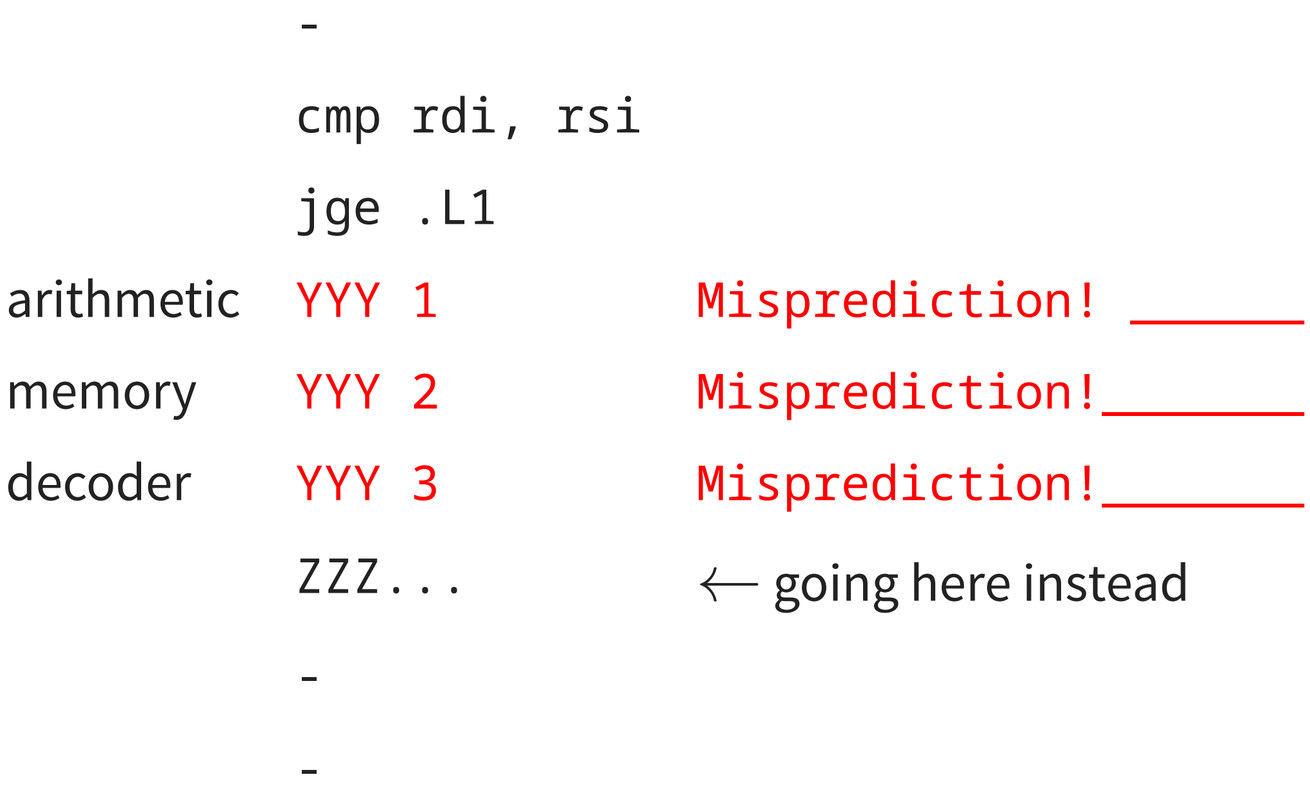
- in practice, the CPU will try to predict which branch will be taken

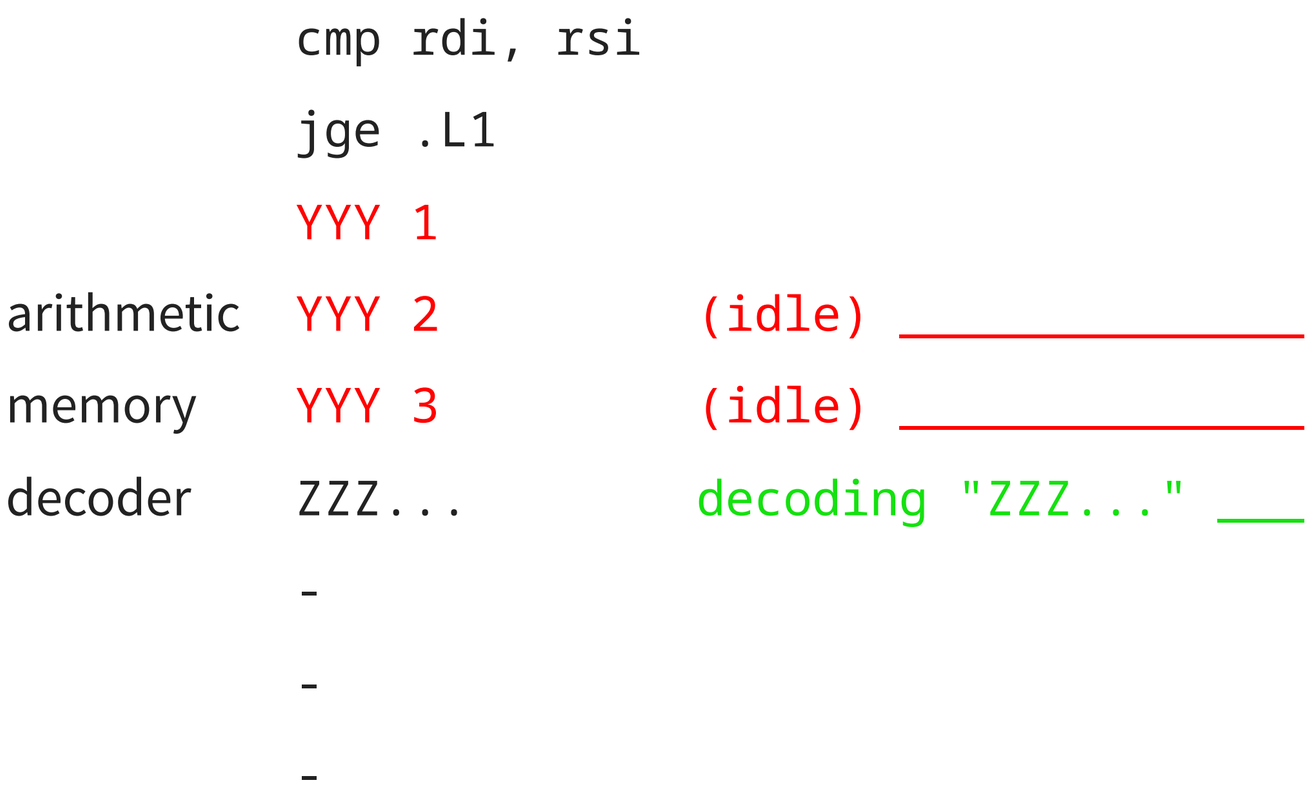
  (based on past choices at that specific instruction)

- and speculatively choose that branch


-
-
-
arithmetic cmp rdi, rsi compare rdi and rsi _
memory jge .L1 (still idle) _____
decoder YYY 1 decoding YYY 1 _____
YYY 2
YYY 3
-

	-	
	-	
	cmp rdi, rsi	
arithmetic	jge .L1	decide taken branch _
memory	YYY 1	memory op for YYY 1 _
decoder	YYY 2	decoding YYY 2 _____
	YYY 3	
	-	
	-	


-
cmp rdi, rsi
jge .L1
arithmetic
YYY 1
Misprediction!
memory
YYY 2
Misprediction!
decoder
YYY 3
Misprediction!
ZZZ...
← going here instead
-
-


cmp rdi, rsi
jge .L1
YYY 1
arithmetic
YYY 2
(idle)
memory
YYY 3
(idle)
decoder
ZZZ...
decoding "ZZZ..."
-
-
-

In practice

- When all goes perfect, processors can actually execute more than one instruction per cycle
- Modern processor pipelines have between 5 and 40 stages
- At each stage, there are multiple circuitry blocks

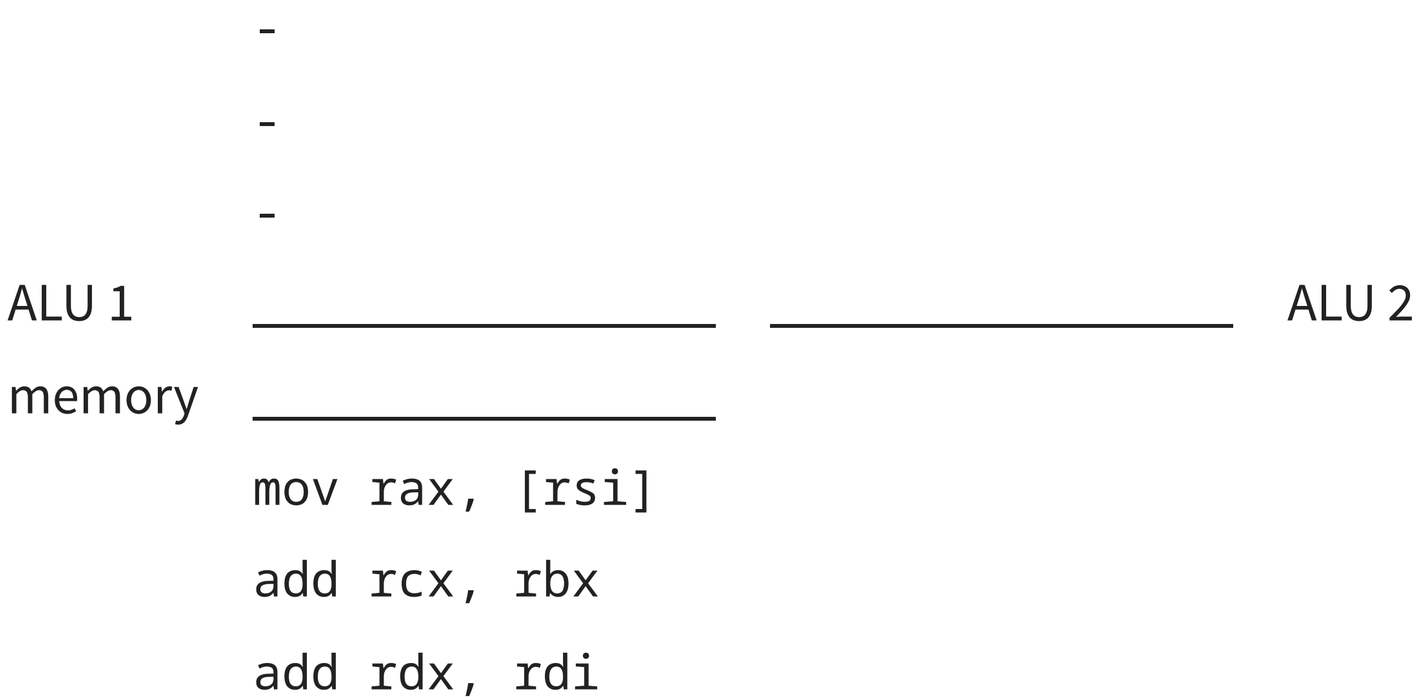
  (decoders, arithmetic and logic unit (ALU) “ports”, etc.)
- Branch mispredict penalty is typically $\geq$ 10 cycles
- Main memory latency is 50–200 cycles

How do we write good code?

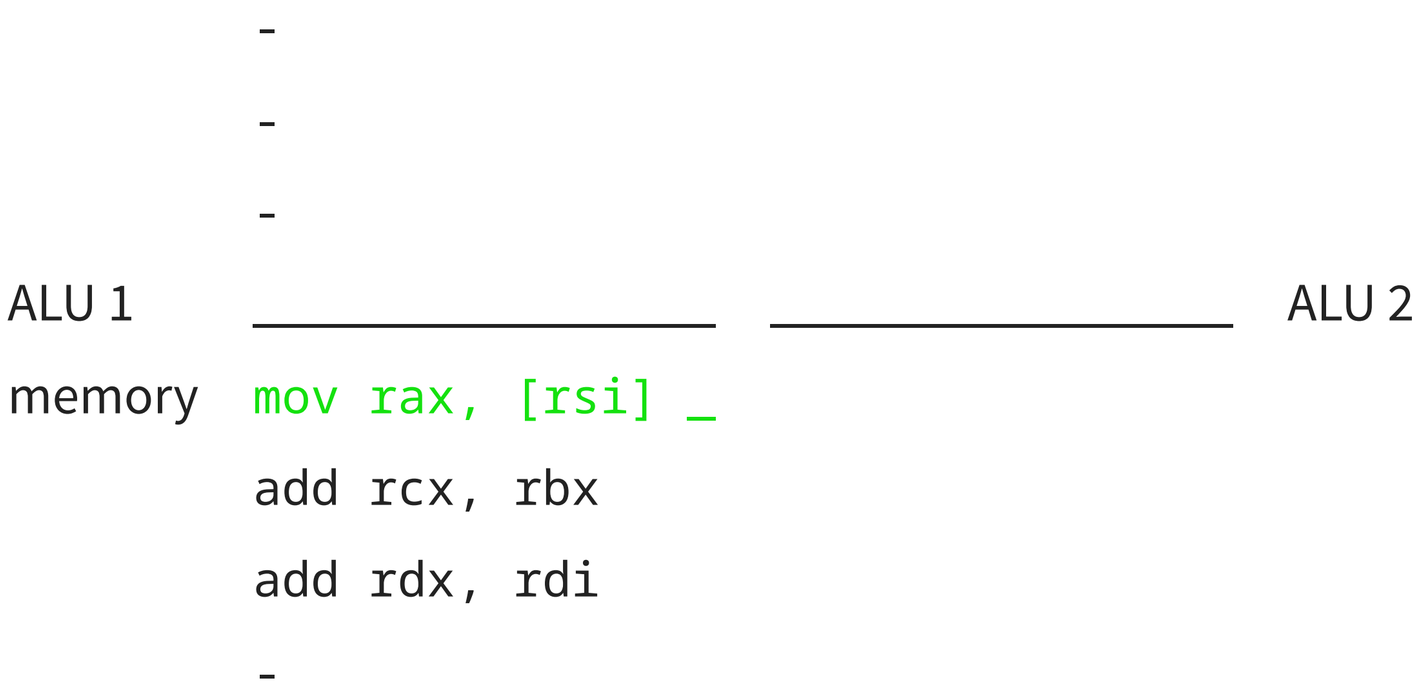
- These parameters vary widely from CPU to CPU
- Specific characteristics are often not public
- It is almost impossible to predict the number of cycles a given set of instructions will take

  (in the presence of branches and memory accesses)
- $\Rightarrow$ Qualitatively: we try to understand the phenomena at play
- $\Rightarrow$ Quantitatively: We measure at runtime

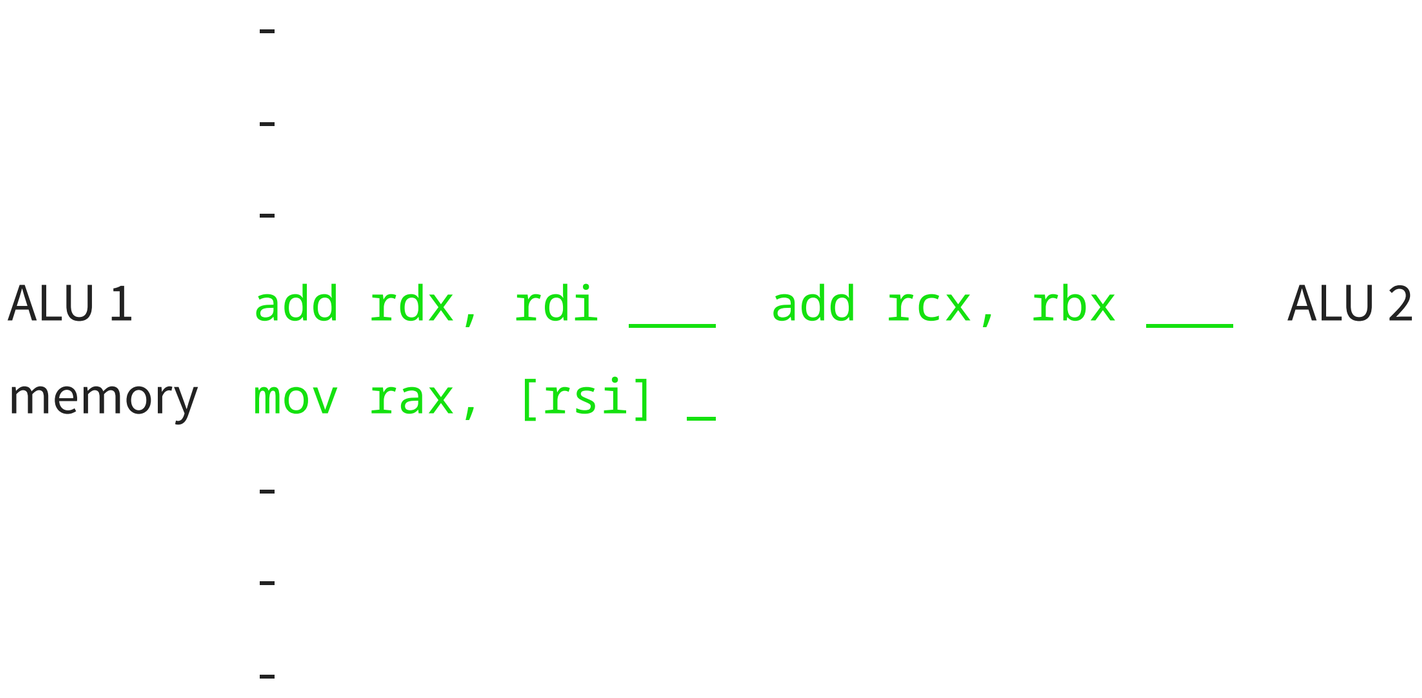
Out-of-order execution



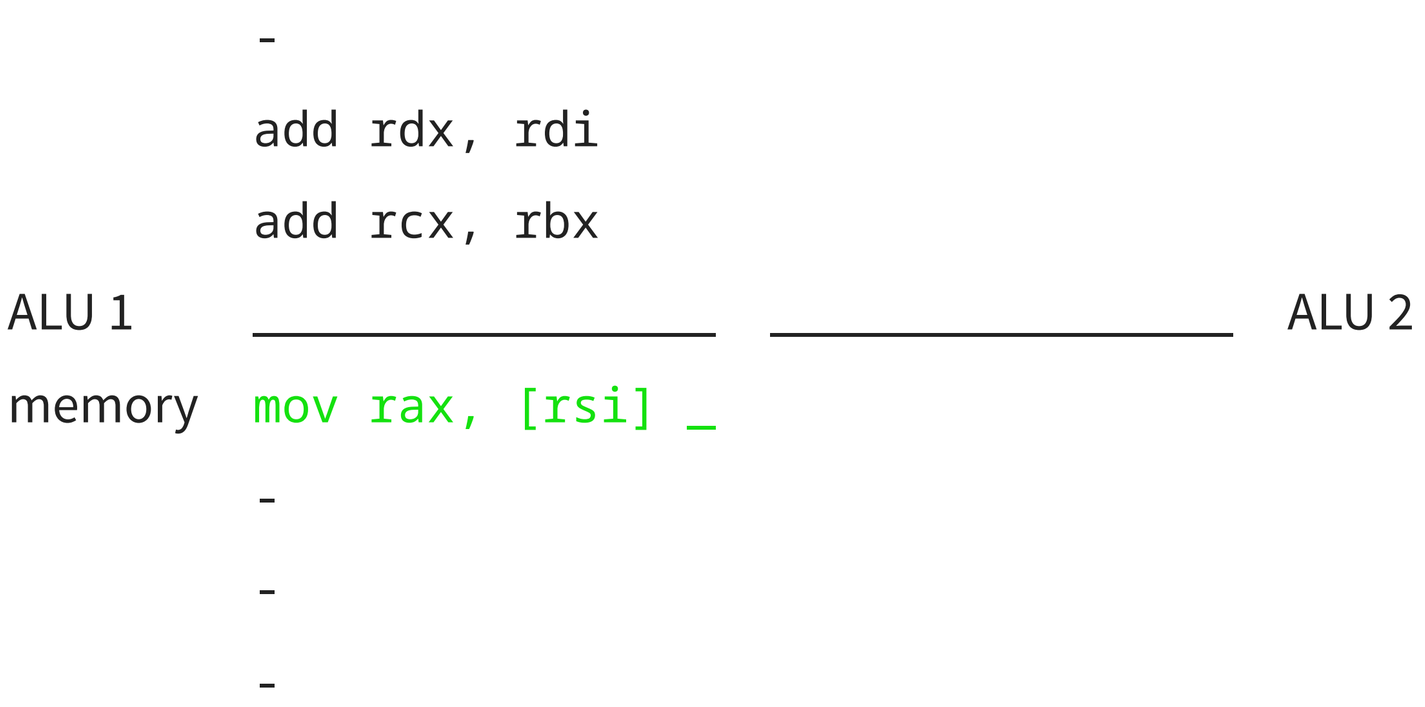
Out-of-order execution



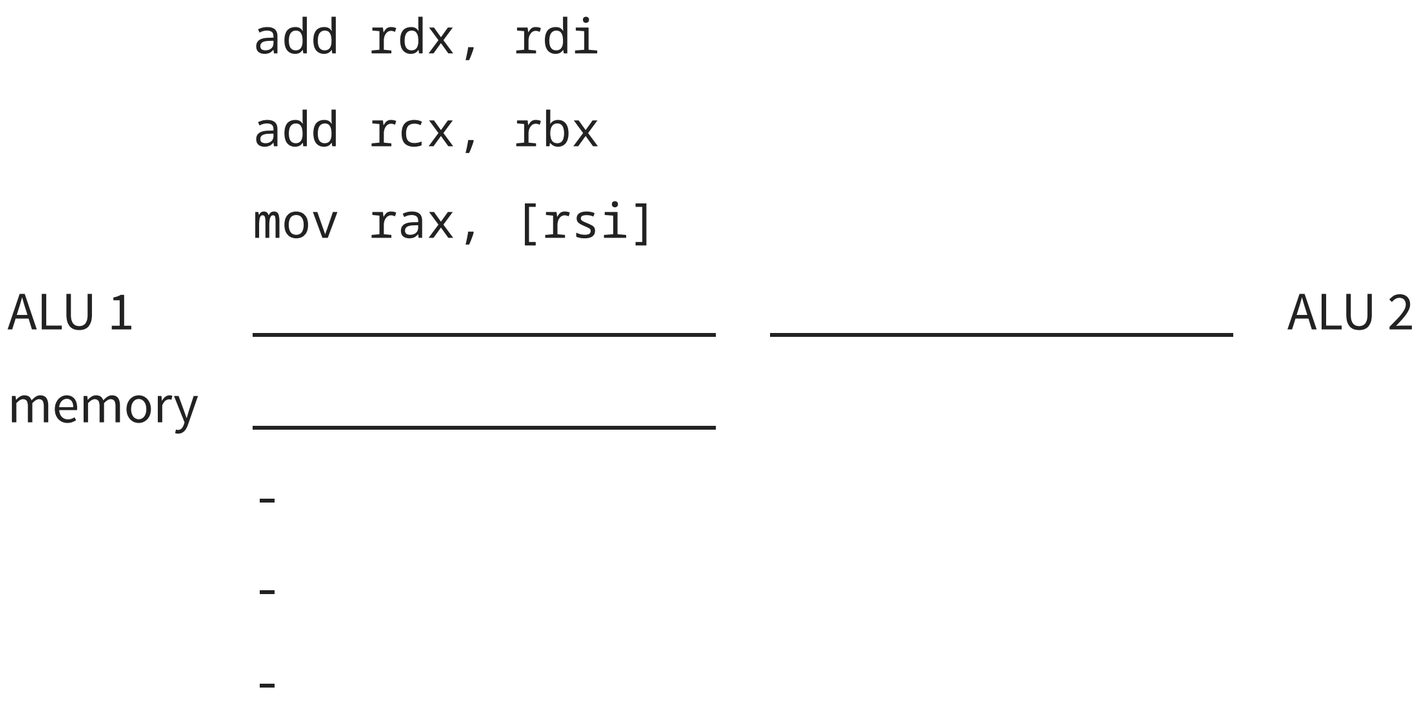
Out-of-order execution



Out-of-order execution

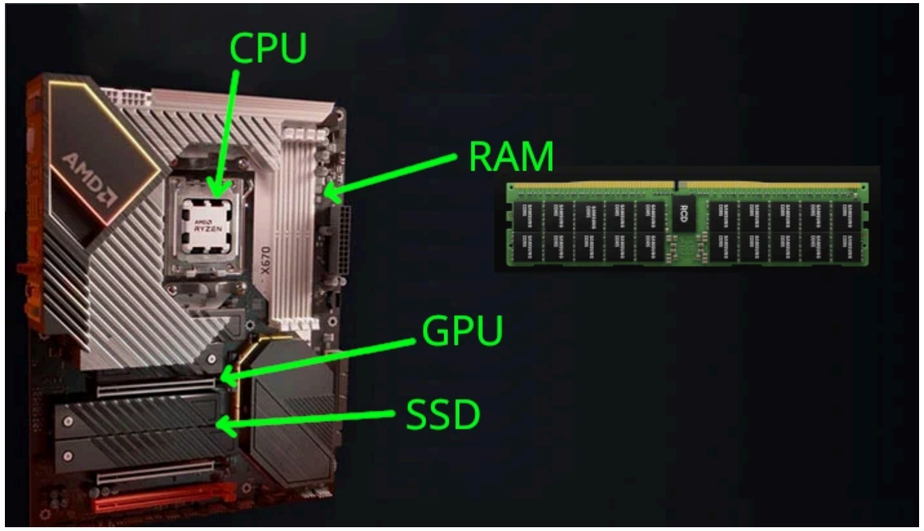


Out-of-order execution



Memory

Access to memory (“random access memory” or RAM) is slow (50–100 cycles)



On desktop & server computers, RAM is typically on distinct integrated circuit (IC) packages, physically centimeters away from the CPU.

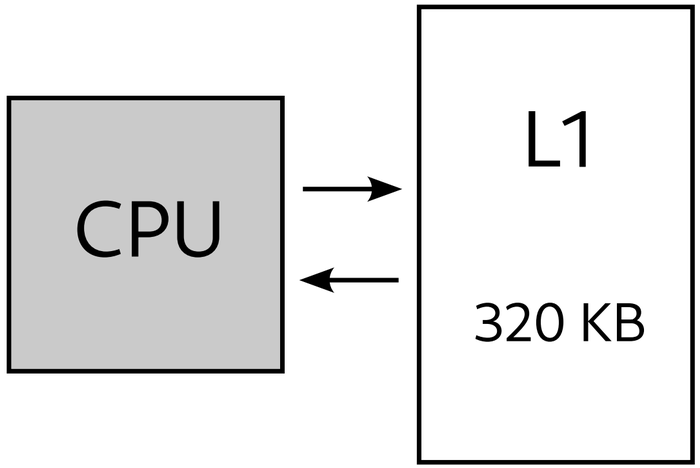
Solution: caching

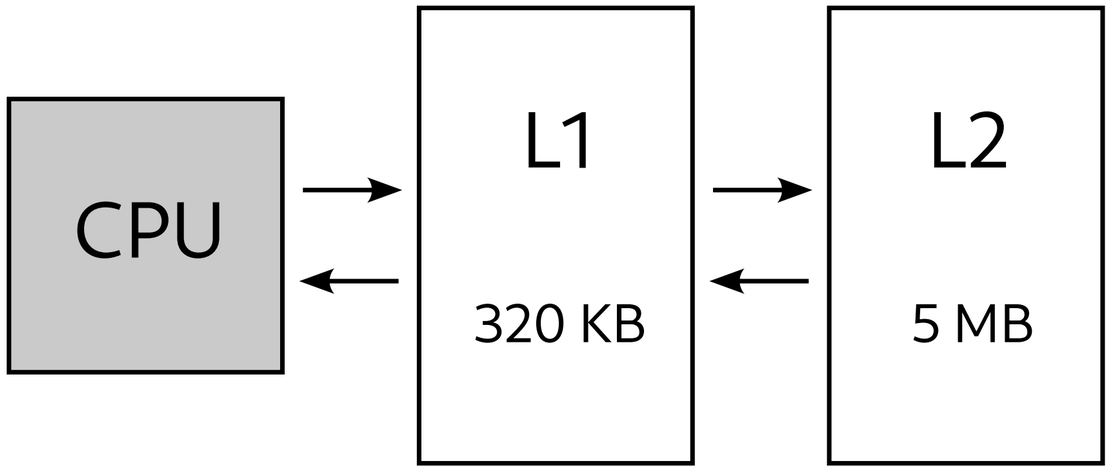
Caching

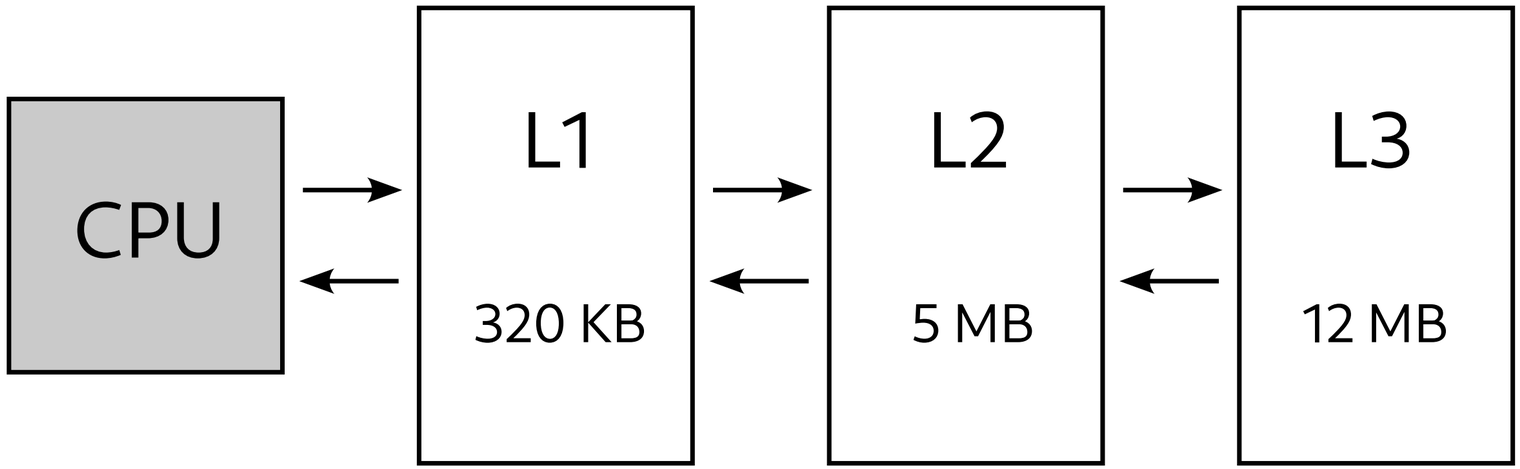
Level 1 (“L1”) cache:

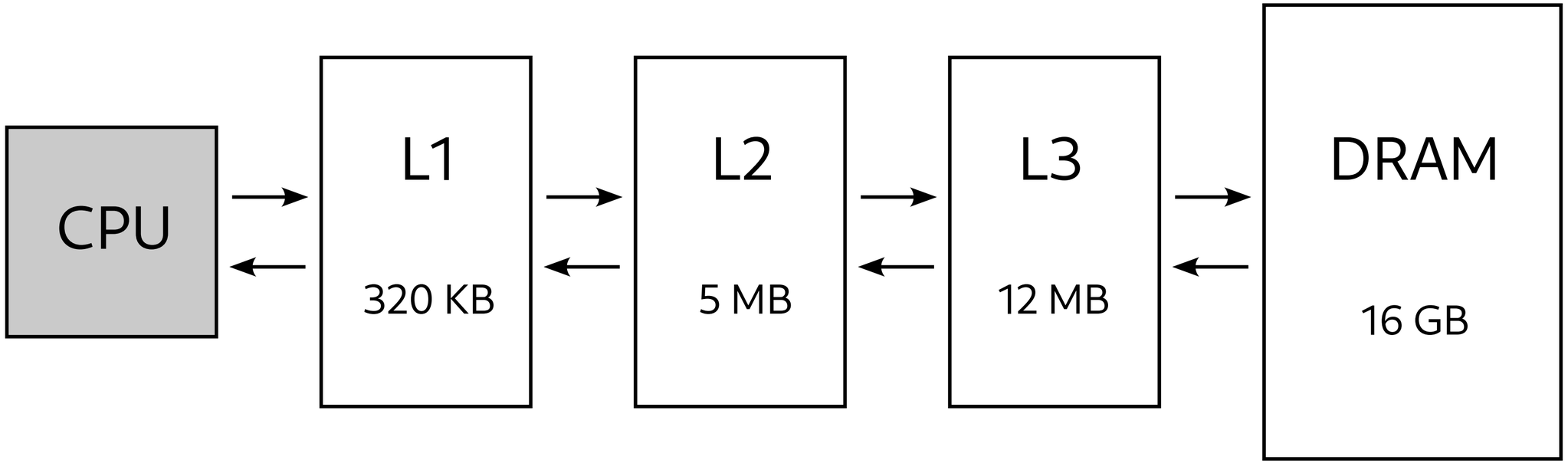
- The CPU contains a small amount of extremely fast memory
- This memory requires many of logic gates on-package
- But it has very low latency (0–10 cycles)
- The CPU contains logic to decide which part of the main memory gets stored in its L1 cache
- This continuously changes over time

- Level 2 (“L2”) cache:
  - slower than L1
  - but requires fewer logic gates, so we can have more
- Level 3 (“L3”) cache:
  - slower than L2 (~50 cycles)
  - but requires fewer logic gates, so we can have more


CPU
L1
320 KB


CPU
L1
320 KB
L2
5 MB


CPU
L1
320 KB
L2
5 MB
L3
12 MB


CPU
L1
320 KB
L2
5 MB
L3
12 MB
DRAM
16 GB

Typical configuration

- memory transits in units of one cache line
  - 64 bytes on x86_64
  - 128 bytes on M1–M6 Macs
- there is no concept of locality beyond cache lines

  if two bytes are in different cache lines, it is irrelevant how close their addresses are
- every memory access is performed through L1 cache
- when all cache entries are full, we need to overwrite one
  - $\longrightarrow$ cache eviction policies e.g. least-recently used (LRU)
- pipelined CPUs have a memory prefetcher (speculatively fills caches in advance)

Zen 4 Cache

	L1I cache	L1D cache	L2 cache	L3 cache
Cache size	32kB	32kB	1MB	xxx
Associativity	8 way	8 way	8 way	xxx
Cache line size	64 b	64 b	64 b	64 b

How do we write good code?

- Again, cache operation varies widely from CPU to CPU
- It is almost impossible to predict how it will behave with complex instruction streams
- $\Rightarrow$ Qualitatively: we try to understand how caches work
- $\Rightarrow$ Quantitatively: We measure at runtime

Memory caches

Decimal example

Consider a CPU with 10000 bytes of memory , with hardware addresses 0 to 9999.

It also has 10 cache lines, of 10 bytes each (L1 only, no L2 or L3).

What parts of the main memory are also stored into the fast cache, and how?

We read one byte at hardware address 1123

The hardware address 1123 is decomposed into three parts:

tag	block index	block offset
11	2	3

Our 10 cache lines look like:

(block index)	tag	10 bytes of data
0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
1	06	e4 ae 2d f3 06 39 83 f8 d1 13
2	51	00 01 02 03 04 05 06 07 08 09
3	42	56 5e c1 3b 7d 29 ca 41 3b 77
4	13	36 ea 28 92 67 a9 67 a3 27 12
5	08	fa f8 83 9f 71 3a 5e 51 37 41
6	99	5a ba 06 50 a9 52 ab 8d f4 a0
7	71	51 8f 74 59 9a e3 3e 4d fa 96
8	55	44 84 47 68 b0 68 b8 f6 0e f6
9	20	82 9f a6 f6 09 01 6e 70 ce 9d

We read one byte at hardware address 1123

The hardware address 1123 is decomposed into three parts:

tag	block index	block offset
11	→ 2	3

Our 10 cache lines look like:

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	51	00 01 02 03 04 05 06 07 08 09
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

tag mismatch ($11 \neq 51$) → **Cache miss!**

We read one byte at hardware address 1123 (alternate situation)

The hardware address 1123 is decomposed into three parts:

tag	block index	block offset
11	2	3

Our 10 cache lines look like:

(block index)	tag	10 bytes of data
0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
1	06	e4 ae 2d f3 06 39 83 f8 d1 13
2	**11**	f0 f1 f2 f3 f4 f5 f6 f7 f8 f9
3	42	56 5e c1 3b 7d 29 ca 41 3b 77
4	13	36 ea 28 92 67 a9 67 a3 27 12
5	08	fa f8 83 9f 71 3a 5e 51 37 41
6	99	5a ba 06 50 a9 52 ab 8d f4 a0
7	71	51 8f 74 59 9a e3 3e 4d fa 96
8	55	44 84 47 68 b0 68 b8 f6 0e f6
9	20	82 9f a6 f6 09 01 6e 70 ce 9d

We read one byte at hardware address 1123 (alternate situation)

The hardware address 1123 is decomposed into three parts:

tag	block index	block offset
11	→ 2	3

Our 10 cache lines look like:

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	**11**	f0 f1 f2 f3 f4 f5 f6 f7 f8 f9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

tag match (11 = 11) → **Cache hit!**

We read one byte at hardware address 1123 (alternate situation)

The hardware address 1123 is decomposed into three parts:

tag	block index	block offset
11	→ 2	3

Our 10 cache lines look like:

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	11	f0 f1 f2→f3←f4 f5 f6 f7 f8 f9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

tag match (11 = 11) → **Cache hit!**

We write the byte ab at hardware address 1123

The hardware address 1123 is decomposed into three parts:

tag	block index	block offset
11	→ 2	3

Our 10 cache lines look like:

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	55	00 01 02 03 04 05 06 07 08 09
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

tag mismatch ($11 \neq 55$) → **Cache miss!**

We write the byte ab at hardware address 1123

The hardware address 1123 is decomposed into three parts:

tag	block index	block offset
11	→ 2	3

Our 10 cache lines look like:

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	11	f0 f1 f2 f3 f4 f5 f6 f7 f8 f9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

Cache miss → We fetch cache line 1120 from RAM.

We write the byte ab at hardware address 1123

The hardware address 123 is decomposed into three parts:

tag	block index	block offset
11	→ 2	3

Our 10 cache lines look like:

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	11	f0 f1 f2→ab←f4 f5 f6 f7 f8 f9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

We write ab at 1123.

Problematic access pattern for 1-way associative caches

```
char *ptr = (char *)1123;

for (int i = 0; i < 80; i++) {
    ptr[i * 100] = ptr[(i + 1) * 100];
}
```

Read address 1223

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	12	b0 b1 b2→b3←b4 b5 b6 b7 b8 b9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

Problematic access pattern for 1-way associative caches

```
char *ptr = (char *)1123;

for (int i = 0; i < 80; i++) {
    ptr[i * 100] = ptr[(i + 1) * 100];
}
```

Write address 1123

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	11	a0 a1 a2→b3←a4 a5 a6 a7 a8 a9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

Problematic access pattern for 1-way associative caches

```
char *ptr = (char *)1123;

for (int i = 0; i < 80; i++) {
    ptr[i * 100] = ptr[(i + 1) * 100];
}
```

Read address 1323

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	13	c0 c1 c2→c3←c4 c5 c6 c7 c8 c9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

Problematic access pattern for 1-way associative caches

```
char *ptr = (char *)1123;

for (int i = 0; i < 80; i++) {
    ptr[i * 100] = ptr[(i + 1) * 100];
}
```

Write address 1223

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	12	b0 b1 b2→c3←b4 b5 b6 b7 b8 b9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

Problematic access pattern for 1-way associative caches

```
char *ptr = (char *)1123;

for (int i = 0; i < 80; i++) {
    ptr[i * 100] = ptr[(i + 1) * 100];
}
```

Read address 1423

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	14	d0 d1 d2→d3←d4 d5 d6 d7 d8 d9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

Problematic access pattern for 1-way associative caches

```
char *ptr = (char *)1123;

for (int i = 0; i < 80; i++) {
    ptr[i * 100] = ptr[(i + 1) * 100];
}
```

Write address 1323

	(block index)	tag	10 bytes of data
	0	82	77 79 ee 3c 36 21 c4 6e 1a 3f
	1	06	e4 ae 2d f3 06 39 83 f8 d1 13
→	2	13	c0 c1 c2→d3←c4 c5 c6 c7 c8 c9
	3	42	56 5e c1 3b 7d 29 ca 41 3b 77
	4	13	36 ea 28 92 67 a9 67 a3 27 12
	5	08	fa f8 83 9f 71 3a 5e 51 37 41
	6	99	5a ba 06 50 a9 52 ab 8d f4 a0
	7	71	51 8f 74 59 9a e3 3e 4d fa 96
	8	55	44 84 47 68 b0 68 b8 f6 0e f6
	9	20	82 9f a6 f6 09 01 6e 70 ce 9d

Problematic access pattern for 1-way associative caches

Because our block index is always the same (2),
we are only ever using a single cache line

Solution: n-way associative cache

Example 2-way associative cache:

tag	block index	block offset
11	2	3

(block index)	tag 0	10 bytes of data for tag 0	tag 1	10 bytes of data for tag 1
0	82	77 79 ee 3c 36 21 c4 6e 1a 3f	26	70 6a b2 d0 17 a1 f2 bc 6f ef
1	06	e4 ae 2d f3 06 39 83 f8 d1 13	17	31 19 b0 f7 36 f9 49 44 13 7c
2	13	b0 b1 b2 b3 b4 b5 b6 b7 b8 b9	49	a3 16 58 c9 30 ac 4e 9b 37 fa
3	42	56 5e c1 3b 7d 29 ca 41 3b 77	51	ab 23 35 34 d0 7d e6 4d 3a 95
4	13	36 ea 28 92 67 a9 67 a3 27 12	22	1b c7 c7 90 58 e7 8e 78 1d 26
5	08	fa f8 83 9f 71 3a 5e 51 37 41	28	45 8c 2a 20 bc 9e 5e 02 9a 7e
6	99	5a ba 06 50 a9 52 ab 8d f4 a0	57	c1 a3 f2 7e 40 82 55 13 56 92
7	71	51 8f 74 59 9a e3 3e 4d fa 96	69	67 ee ee 1a 28 da 53 f4 36 93
8	55	44 84 47 68 b0 68 b8 f6 0e f6	26	3a 7f 64 f2 11 48 5c 80 6b fc
9	20	82 9f a6 f6 09 01 6e 70 ce 9d	03	23 b3 01 33 f3 e5 52 3d 2b 0f

Real example

Zen 4 Cache

	L1I cache	L1D cache	L2 cache	L3 cache
Cache size	32kB	32kB	1MB	xxx
Associativity	8 way	8 way	8 way	xxx
Cache line size	64 b	64 b	64 b	64 b

L1D cache (per core):

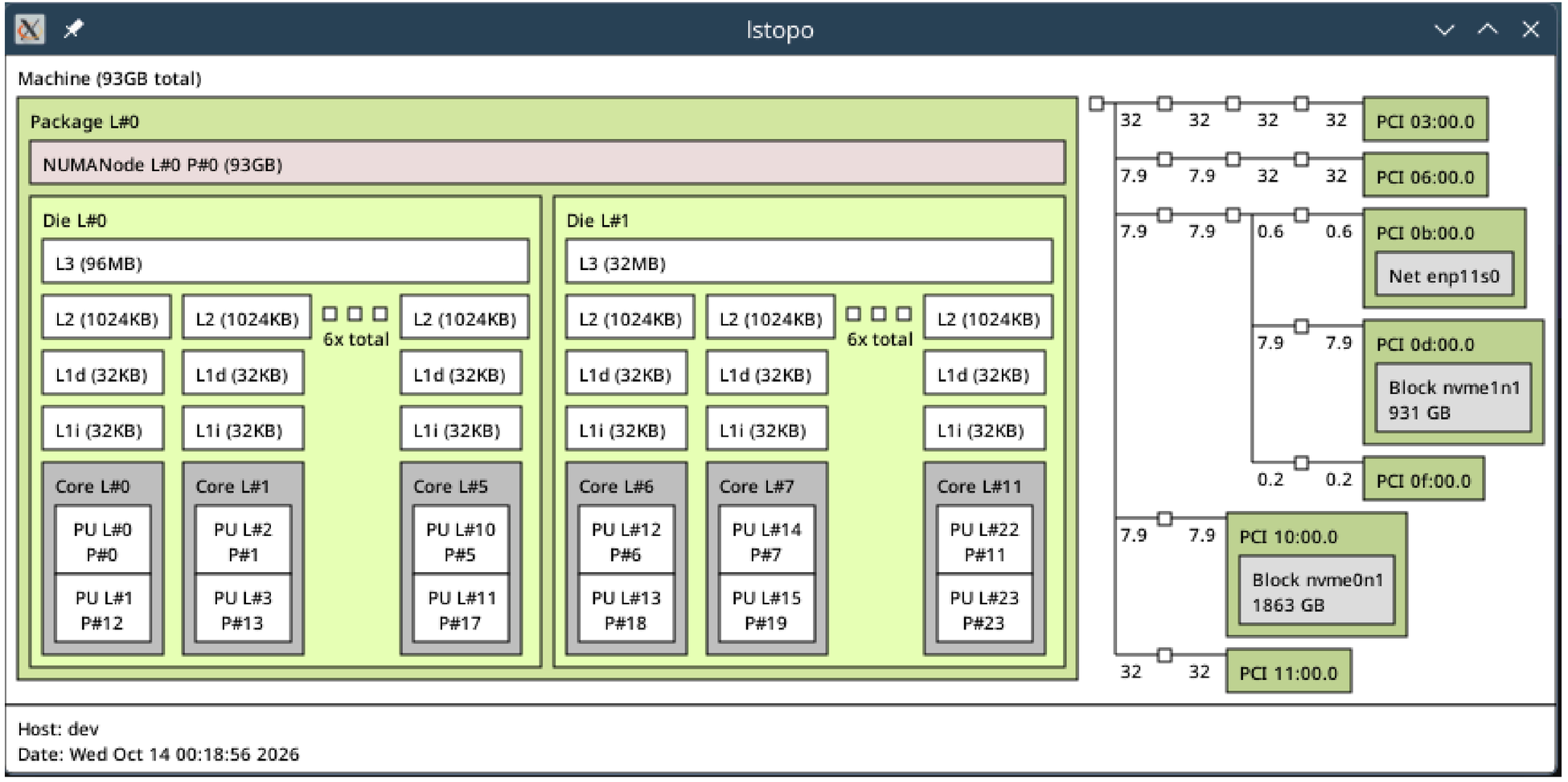
tag	block index	block offset
up to 52 bits	6 bits: 0..63	6 bits: 0..63

64 caches lines × 8 way associative × 64 bytes = 32768 total bytes

Real example

Zen 4 Cache

	L1I cache	L1D cache	L2 cache	L3 cache
Cache size	32kB	32kB	1MB	xxx
Associativity	8 way	8 way	8 way	xxx
Cache line size	64 b	64 b	64 b	64 b


lstopo
Machine (93GB total)
Package L#0
NUMANode L#0 P#0 (93GB)
Die L#0
L3 (96MB)
L2 (1024KB)
L2 (1024KB)
6x total
L2 (1024KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#0
PU L#0 P#0
PU L#1 P#12
Core L#1
PU L#2 P#1
PU L#3 P#13
Core L#5
PU L#10 P#5
PU L#11 P#17
Die L#1
L3 (32MB)
L2 (1024KB)
L2 (1024KB)
6x total
L2 (1024KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core L#6
PU L#12 P#6
PU L#13 P#18
Core L#7
PU L#14 P#7
PU L#15 P#19
Core L#11
PU L#22 P#11
PU L#23 P#23
32
32
32
32
PCI 03:00.0
7.9
7.9
32
32
PCI 06:00.0
7.9
7.9
0.6
0.6
PCI 0b:00.0
Net enp11s0
7.9
7.9
PCI 0d:00.0
Block nvme1n1
931 GB
0.2
0.2
PCI 0f:00.0
7.9
7.9
PCI 10:00.0
Block nvme0n1
1863 GB
32
32
PCI 11:00.0
Host: dev
Date: Wed Oct 14 00:18:56 2026

More caches and pipelines

TLB

- recall that memory is virtualized
- a virtual address $\longrightarrow$ hardware address translation is necessary for every memory access
- the translation uses a page table
- the page table is stored in memory
- hence at least two effective memory accesses per memory fetch instruction?

Solution: part of the page table is **cached** in the CPU

the “Translation Lookaside Buffer” (TLB)

Caches and pipelines are used at various levels to hide access latency

	typical latency	
instruction	~0.25 ns	0.25 ns
RAM	~100 ns	100 ns
solid state drive (SSD)	~0.2 ms	200,000 ns
hard disk drive (HDD)	~2 ms	2,000,000 ns
wired ethernet (round-trip)	~1 ms	1,000,000 ns
wifi latency (round-trip)	~5 ms	5,000,000 ns
same-city internet (round-trip)	~5 ms	5,000,000 ns
same-continent internet (round-trip)	~25 ms	25,000,000 ns
transatlantic internet (round-trip)	~100 ms	100,000,000 ns

Examples of caches

- SSDs have internal RAM caches (typically 0-4 GB)
- the operating system caches files in memory
- large content providers (Google, Amazon, Netflix, Cloudflare) have caches all over the world

Examples of pipelines

- Storage devices:
  - SSDs typically access data in “pages” of **4096 bytes**
  - **0.2ms** SSD latency would imply a max speed of 20 MB / s
  - instead SSDs routinely read and write 500 MB / s
- Networks:
  - Network packets are typically **1500 bytes**
  - **10ms** WiFi latency would imply 150 KB / s
  - instead most WiFi networks do at least 10,000 KB / s
- Browsers:
  - Google Chrome maintains up to **6** connections per domain