TDT4200 Parallel Programming

Bart van Blokland

Lecture 2

Two things before we start..

First lab session:

This Wednesday

12:00 – 14:00+, ITS-015

Compiling a program with one source file:

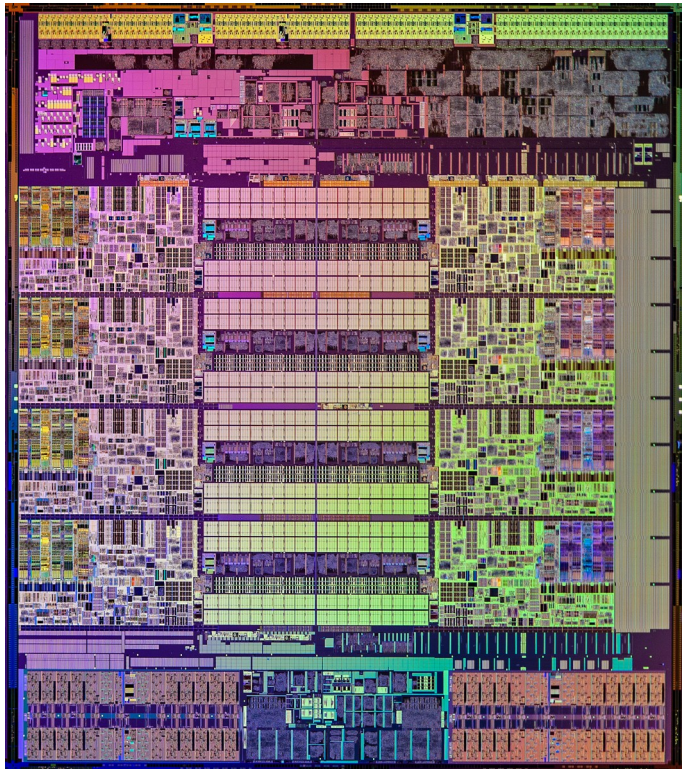
```
g++ sourceFile.cpp -o executable
```

Enabling compiler optimisations:

```
g++ sourceFile.cpp -o executable -O3 -march=native
```

Last Time:

Parallel Programming – What and Why?



Performance

Running code in parallel is only one solution..

Today and next week:

Single-threaded performance

Software profiling:
Where DID the time go??

“ Premature optimization is the root of all evil. ”

- Donald Knuth

“ Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. ”

- Donald Knuth

Measure

Why measure?

Execution time is not always obvious.

Which one is faster?

A

```
long sumOfIntegers(int limit) {
    return (limit * (limit + 1)) / 2;
}
```

B

```
long sumOfIntegers(int limit) {
    long sum = 0;
    for(int i = 0; i < limit; i++) {
        sum += i;
    }
    return sum;
}
```

Which one is faster?

A

```
const long count = 50000;
long sum = 0;

for(long i = 0; i < count * count; i++) {
    sum += i;
}

std::cout << sum << std::endl;
```

B

```
const long count = 5000;
long sum = 0;

for(long i = 0; i < count * count; i++) {
    sum += i;
}

std::cout << sum << std::endl;
```

Which one is faster?

A

```
const long count = 50000;
long sum = 0;

for(long i = 0; i < count * count; i++) {
    sum += i;
}

std::cout << sum << std::endl;
```

B

```
const long count = 5000;
long sum = 0;

for(long i = 0; i < count * count; i++) {
    sum += i;
}

std::cout << sum << std::endl;
```

A: 3.258 s

B: 0.044 s

Which one is faster?

A

```
const unsigned long count = 500000000;
long long sum = 0;

for(long i = 0; i < count; i++) {
    sum += i / (count - i);
}

std::cout << sum << std::endl;
```

B

```
const unsigned long count = 500000000;
long long sum = 0;

for(long i = 0; i < count; i++) {
    sum += i * (count - i);
}

std::cout << sum << std::endl;
```

Which one is faster?

A

```
const unsigned long count = 500000000;
long long sum = 0;

for(long i = 0; i < count; i++) {
    sum += i / (count - i);
}

std::cout << sum << std::endl;
```

A: 3.159 s

B: 1.073 s

B

```
const unsigned long count = 500000000;
long long sum = 0;

for(long i = 0; i < count; i++) {
    sum += i * (count - i);
}

std::cout << sum << std::endl;
```

Which one is faster?

A

```
long sum2D(int* someArray, int size) {
    long sum = 0;
    for(int row = 0; row < size; row++) {
        for(int col = 0; col < size; col++) {
            sum += someArray[col * size + row];
        }
    }
    return sum;
}
```

B

```
long sum2D(int* someArray, int size) {
    long sum = 0;
    for(int col = 0; col < size; col++) {
        for(int row = 0; row < size; row++) {
            sum += someArray[col * size + row];
        }
    }
    return sum;
}
```

Which one is faster?

A

```
long sum2D(int* someArray, int size) {
    long sum = 0;
    for(int row = 0; row < size; row++) {
        for(int col = 0; col < size; col++) {
            sum += someArray[col * size + row];
        }
    }
    return sum;
}
```

B

```
long sum2D(int* someArray, int size) {
    long sum = 0;
    for(int col = 0; col < size; col++) {
        for(int row = 0; row < size; row++) {
            sum += someArray[col * size + row];
        }
    }
    return sum;
}
```

Size = 25000

A: 10.767 s

B: 0.352 s

(That's more than a 30x difference)

What caused these differences?

Know Your Hardware

Instruction	**Latency (CPU cycles)**
Addition	1
Subtraction	1
Bit shift	1
Multiplication	3
Division	10
Modulo	10
Float sine / cosine	53 - 105

← 01101001 >> 1
= 00110100

Which one is faster?

A

```
const unsigned long count = 500000000;
long long sum = 0;

for(long i = 0; i < count; i++) {
    sum += i / (count - i);
}

std::cout << sum << std::endl;
```

A: 3.159 s

B: 1.073 s

B

```
const unsigned long count = 500000000;
long long sum = 0;

for(long i = 0; i < count; i++) {
    sum += i * (count - i);
}

std::cout << sum << std::endl;
```

Most code on a CPU runs fine regardless.

Compilers do local optimisations for you

(example: a / 2 → a >> 1)

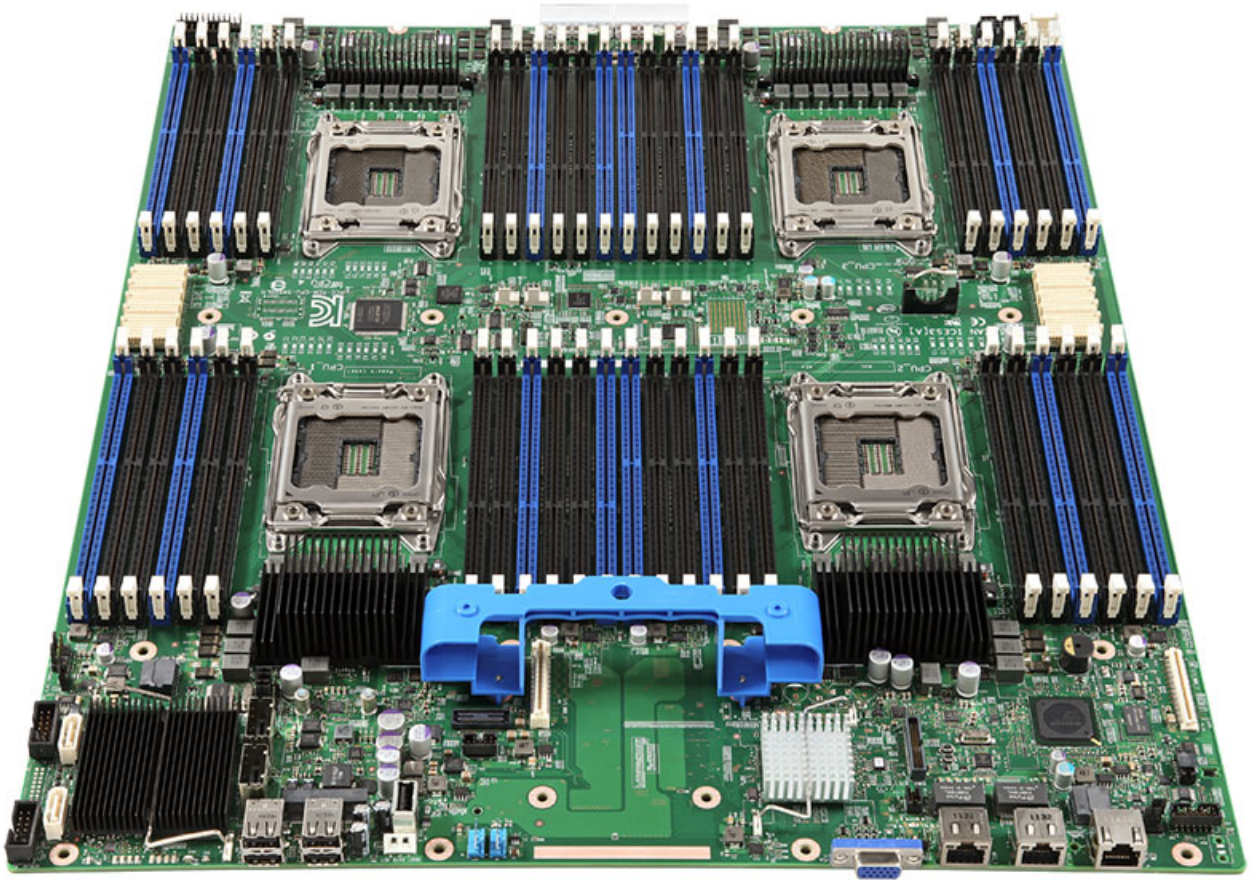
Three Amazing Hardware Features™

Amazing Hardware Feature™ #1:

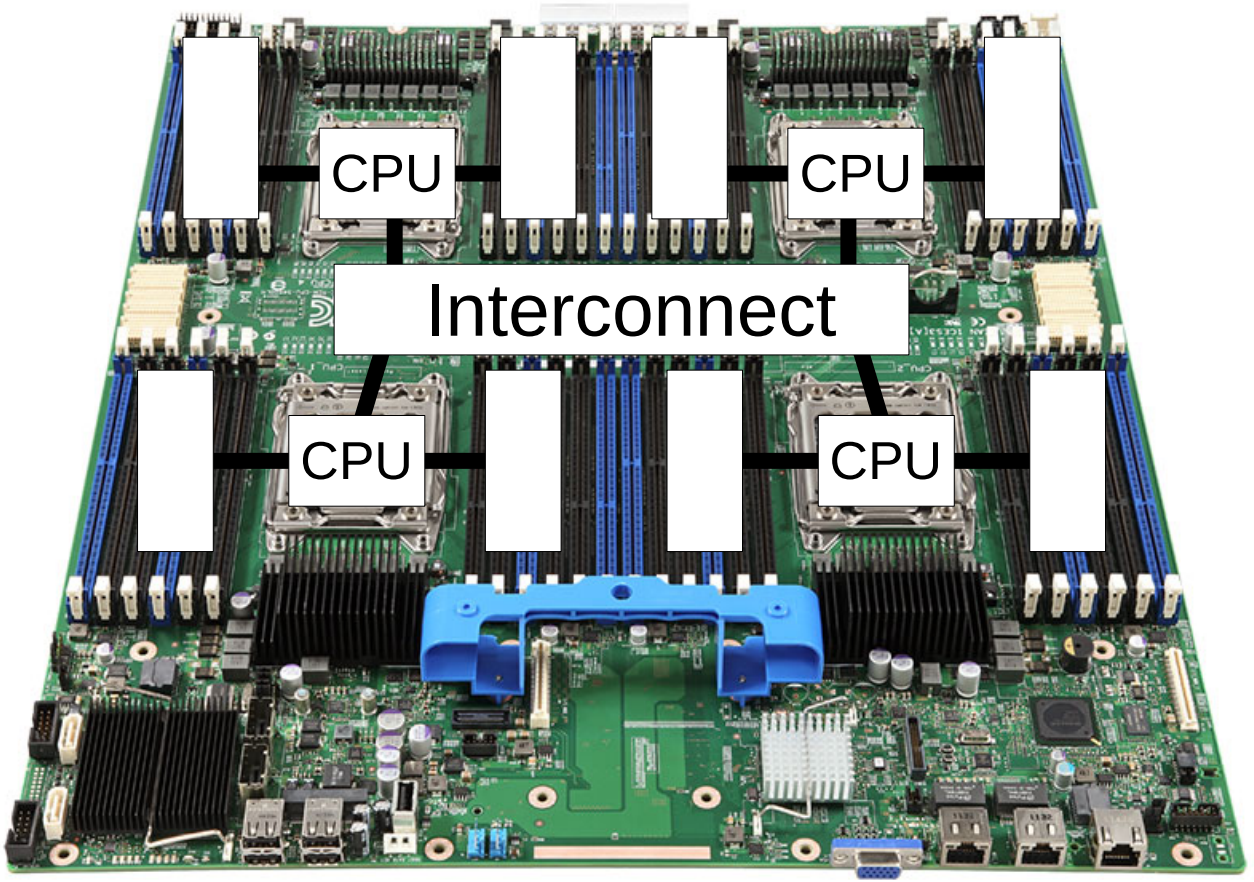
NUMA

Uniform Memory Access (UMA)

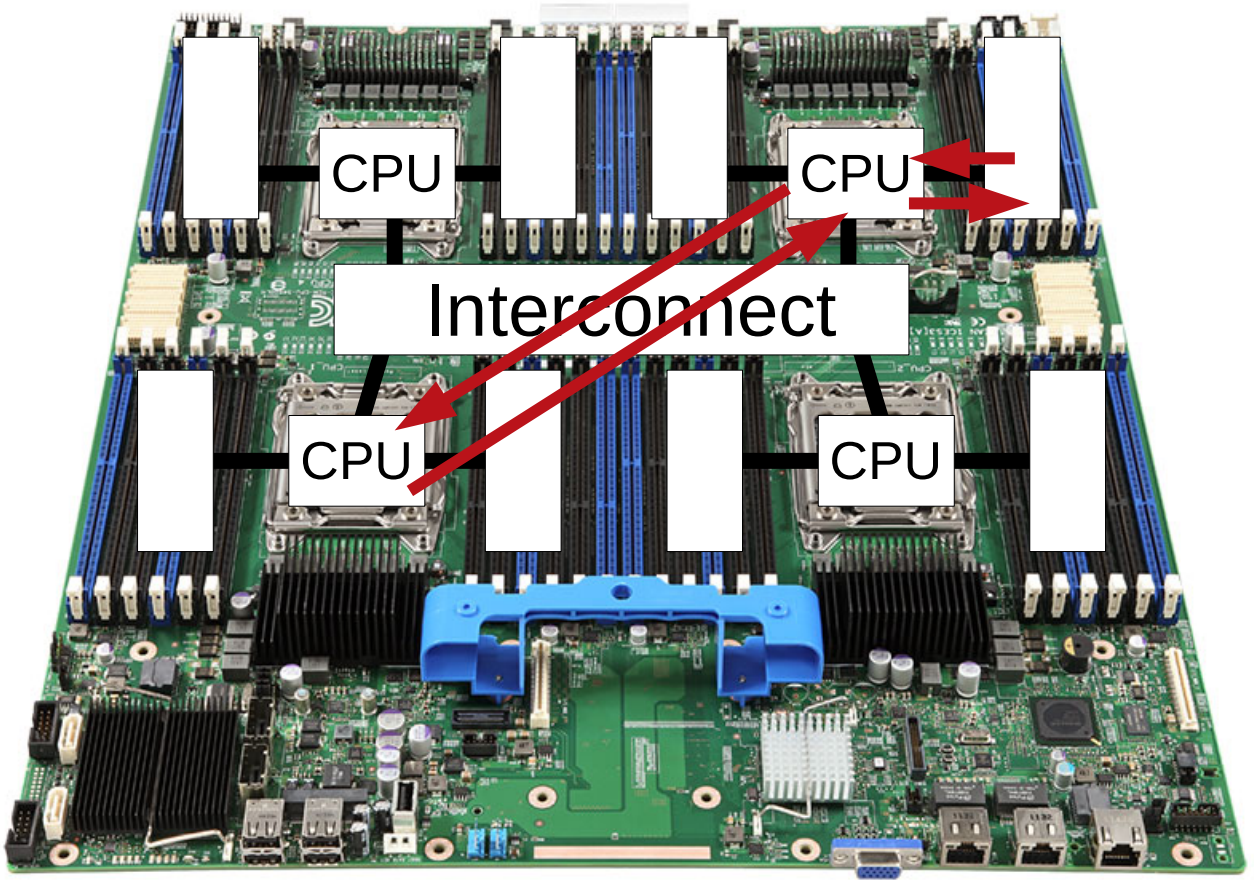
Non-Uniform Memory Access (NUMA)



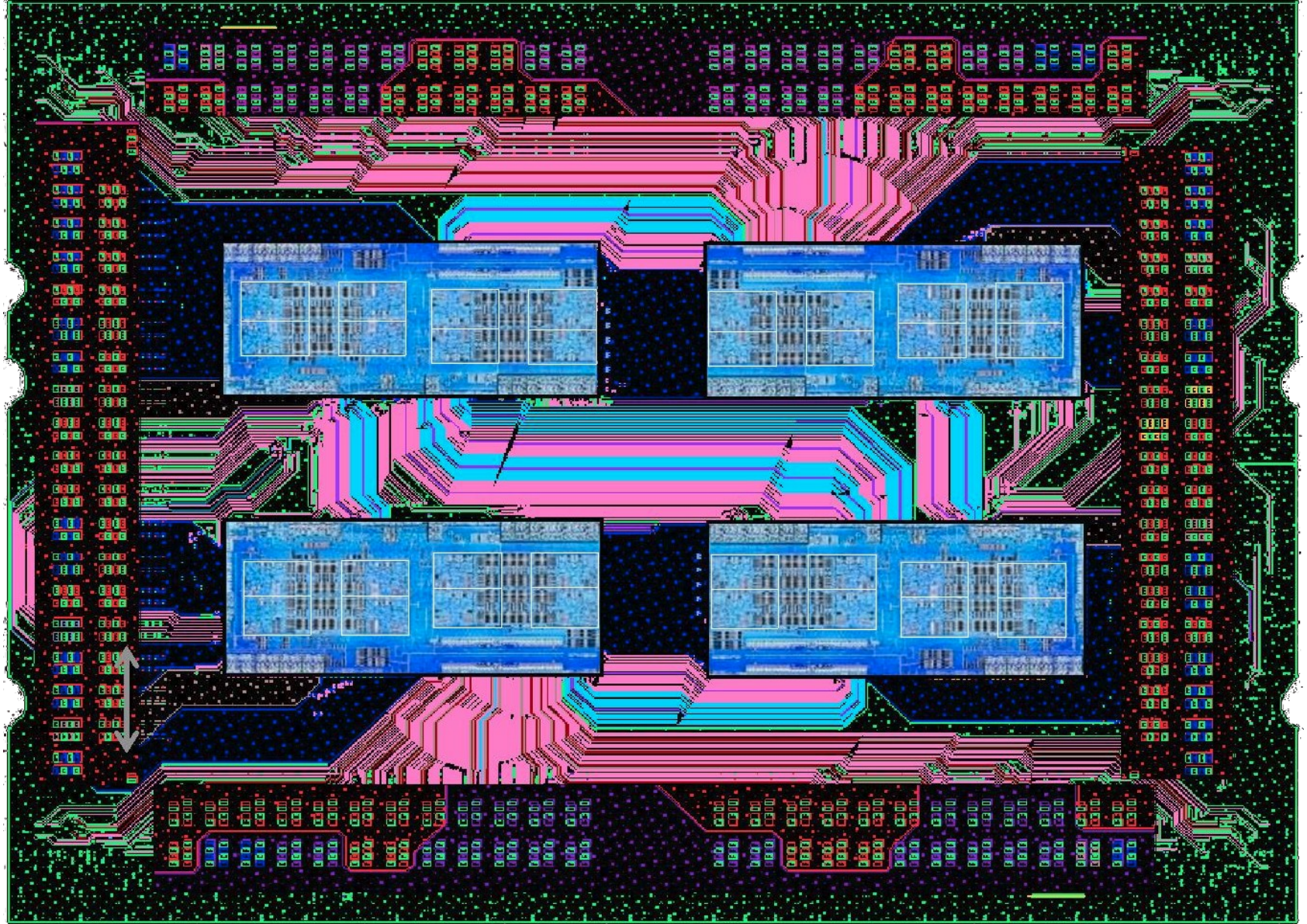
Non-Uniform Memory Access (NUMA)

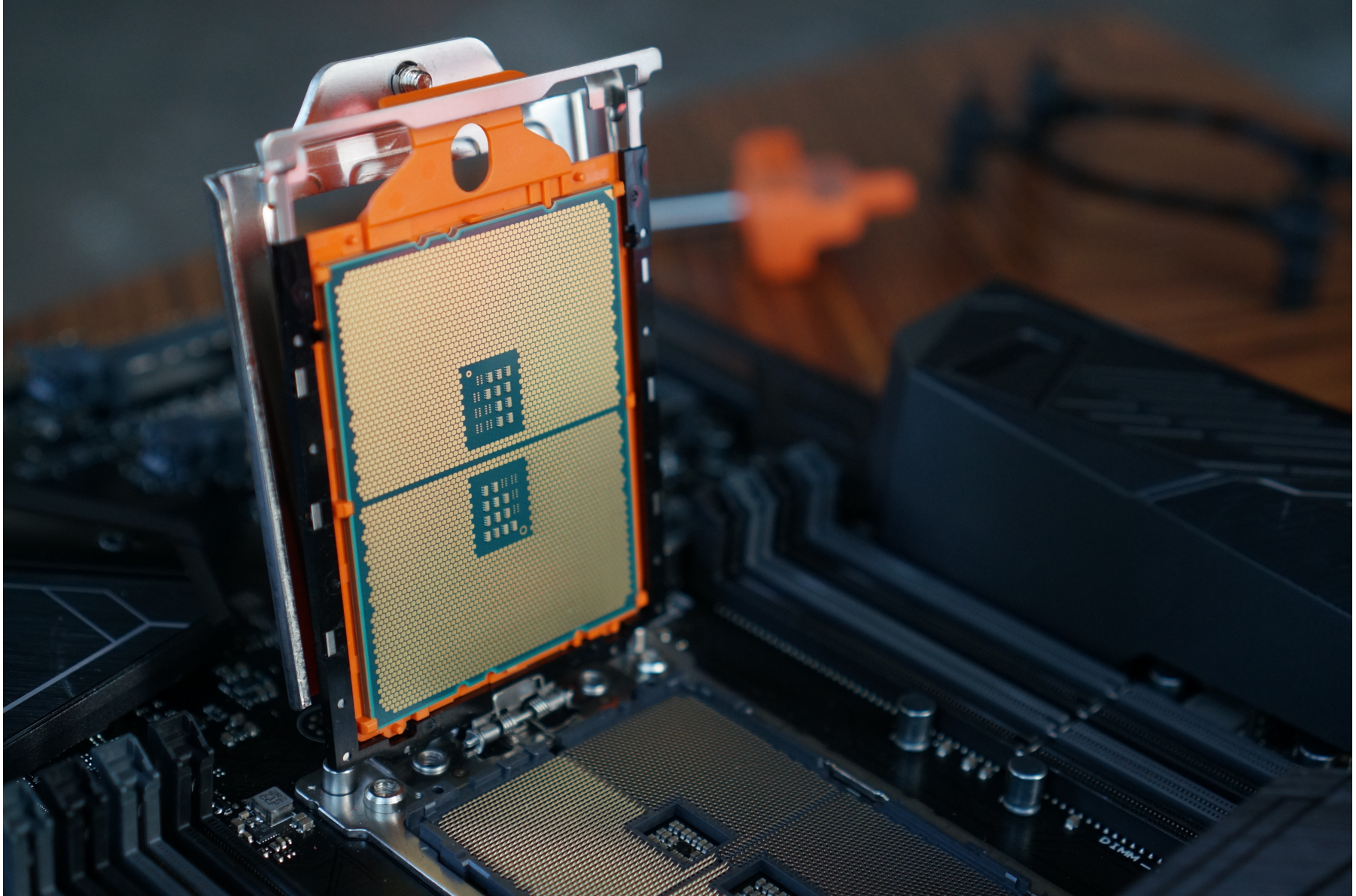


Non-Uniform Memory Access (NUMA)




AMD
RYZEN
THREADRIPPER





Amazing Hardware Feature™ #2:

Instruction Level Parallelism

Pipelining

Clock Cycle / Instruction	1	2	3	4	5	6	7	8	9	10	11	12
1	Fetch	Deco	Exec	Mem	Write							
2		Fetch	Deco	Exec	Mem	Write						
3			Fetch	Deco	Exec	Mem	Write					
4				Fetch	Deco	Exec	Mem	Write				
5					Fetch	Deco	Exec	Mem	Write			
6						Fetch	Deco	Exec	Mem	Write		
7							Fetch	Deco	Exec	Mem	Write	
8								Fetch	Deco	Exec	Mem	Write
9									Fetch	Deco	Exec	Mem

Pipelining

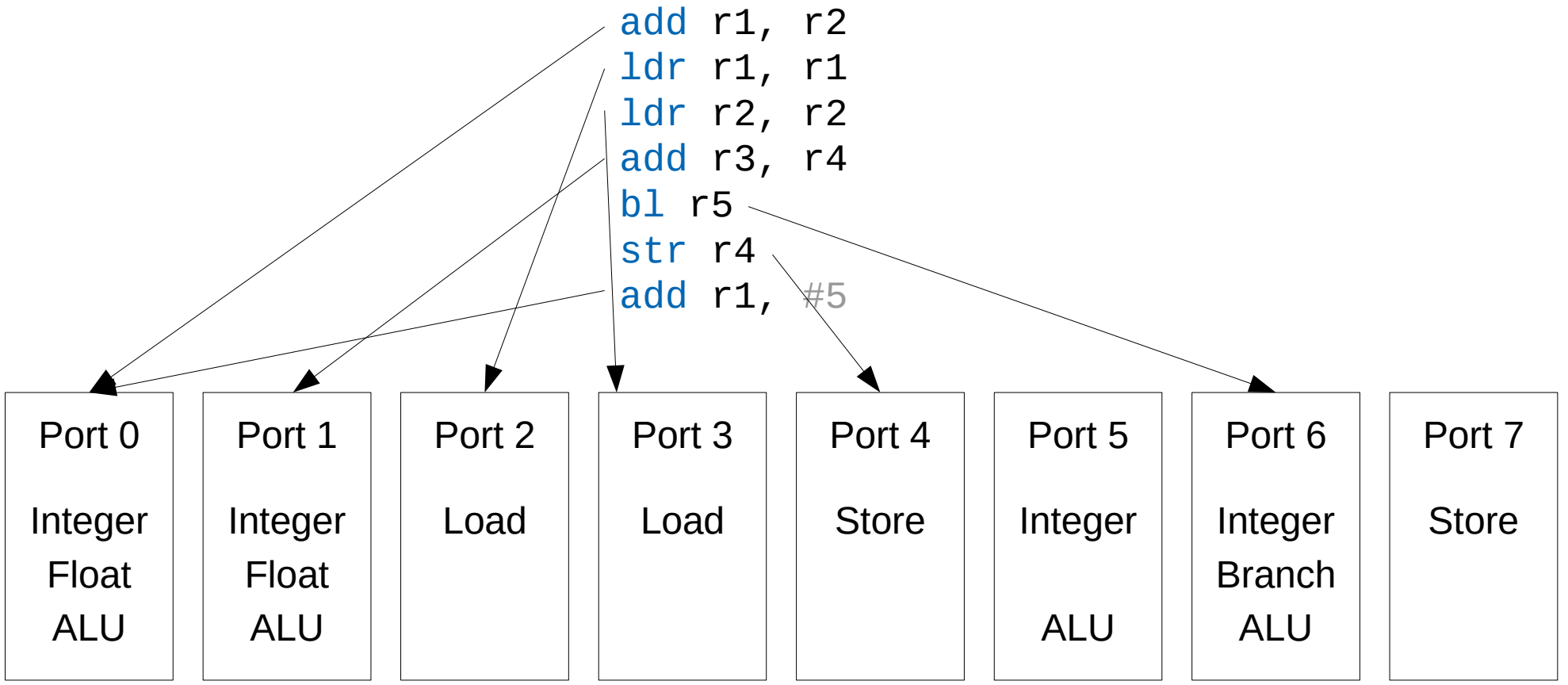
Clock Cycle / Instruction	1	2	3	4	5	6	7	8	9	10	11	12
1	Fetch	Deco	Exec	Mem	Write							
2		Fetch	Deco	Exec	Mem	Write						
3			Fetch	Deco	Exec	Mem	Write					
4				Fetch	Deco	Exec	Mem	Write				
5					Fetch	Deco	Exec	Mem	Write			
6						Fetch	Deco	Exec	Mem	Write		
7							Fetch	Deco	Exec	Mem	Write	
8								Fetch	Deco	Exec	Mem	Write
9									Fetch	Deco	Exec	Mem

Pipelining

Clock Cycle / Instruction	1	2	3	4	5	6	7	8	9	10	11	12
1	Fetch	Deco	Exec	Mem	Write							
2		Fetch	Deco	Exec	Mem	Write						
3			Fetch	Deco	Exec	Mem	Write					
4				Fetch	Deco	Exec	Mem	Write				
5					Fetch	Deco	Exec	Mem	Write			
6						Fetch	Deco	Exec	Mem	Write		
7							Fetch	Deco	Exec	Mem	Write	
8												Write
9											Exec	Mem

Some processors can have as many as 30 stages!
Modern desktop ones are in the range of 14-19

Multiple Issue



Amazing Hardware Feature™ #2.5:

Speculative and Out of Order Execution

Amazing Hard 

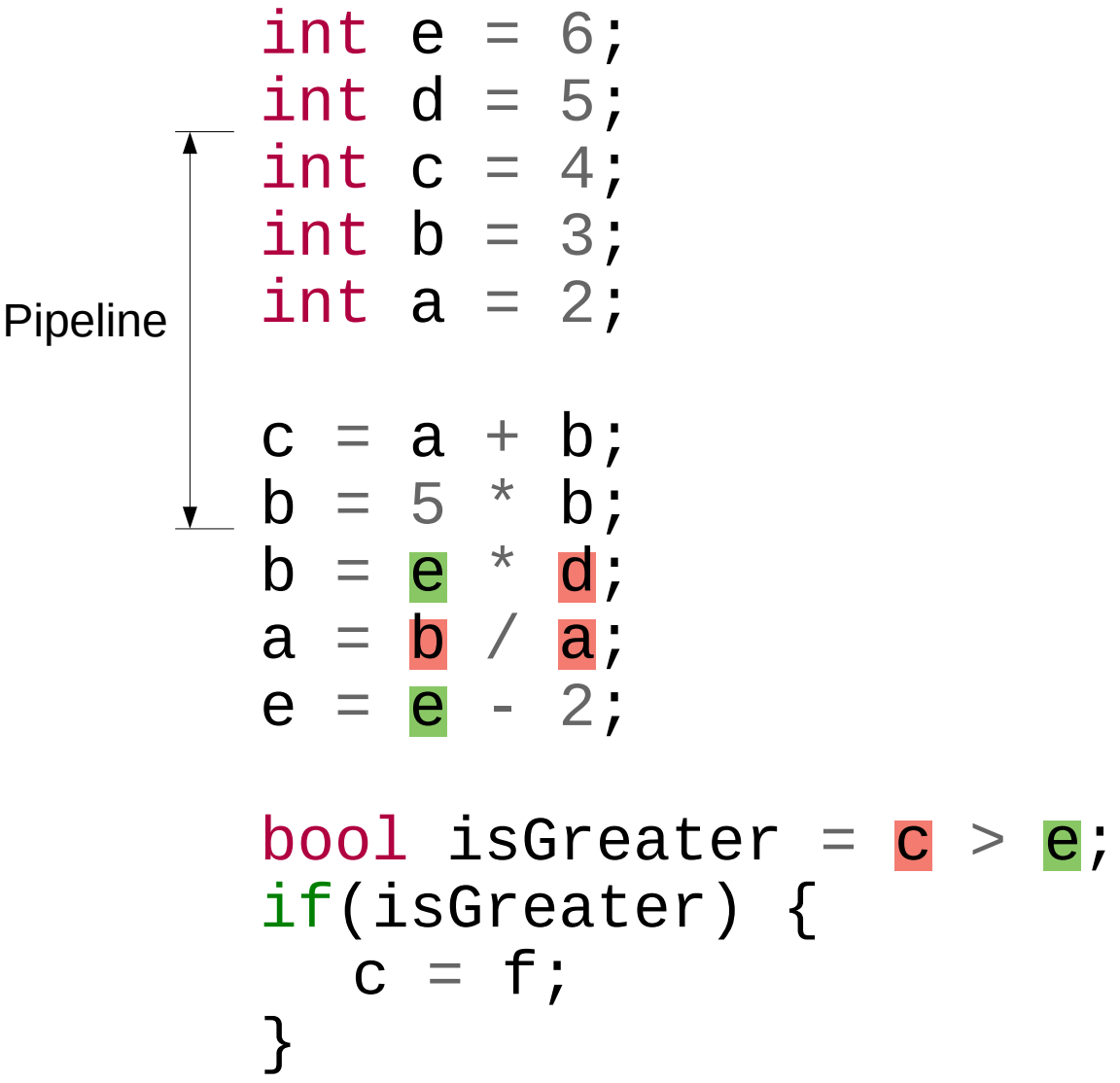
e™ #2.5:

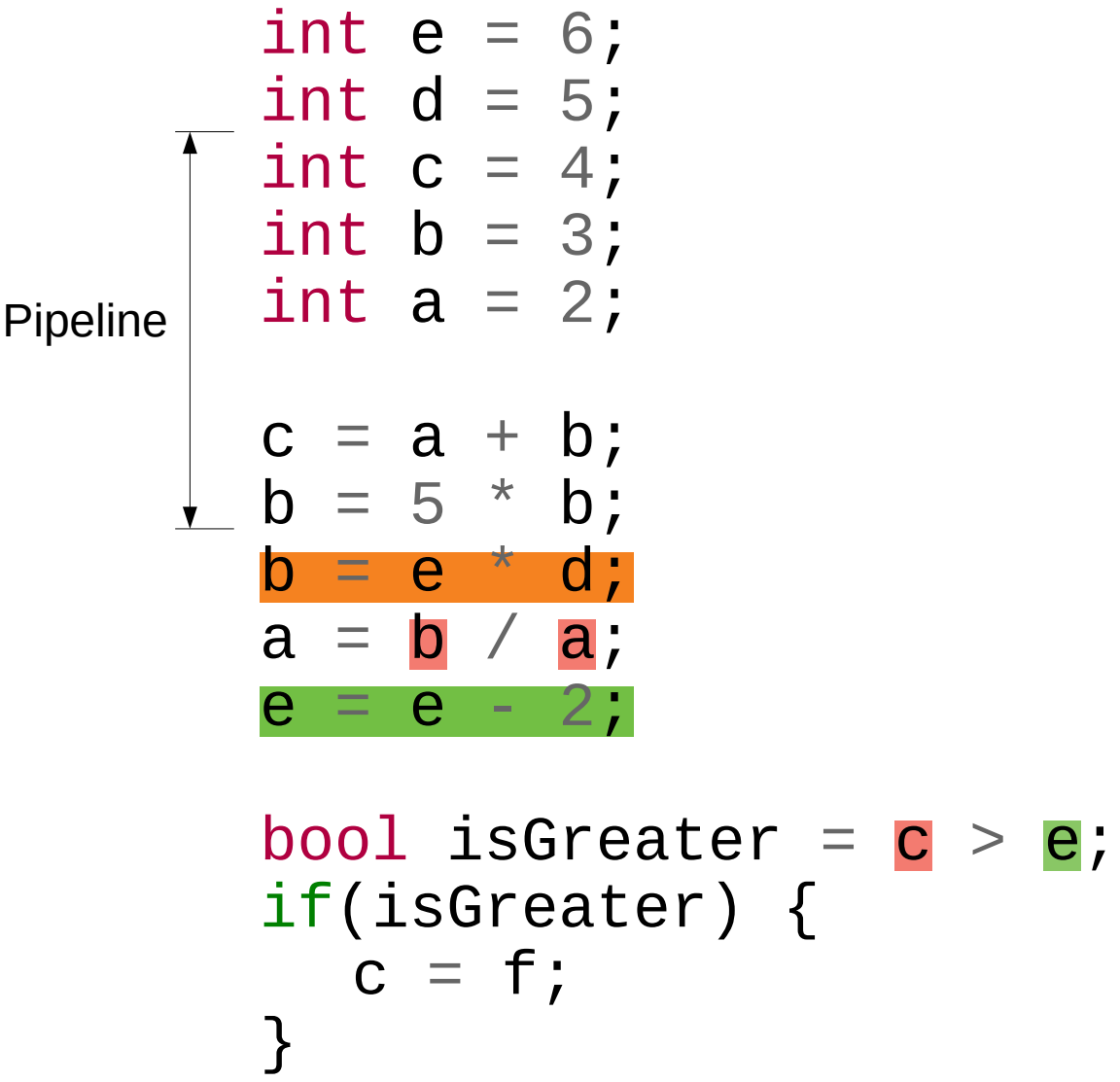
Speculative and Execution

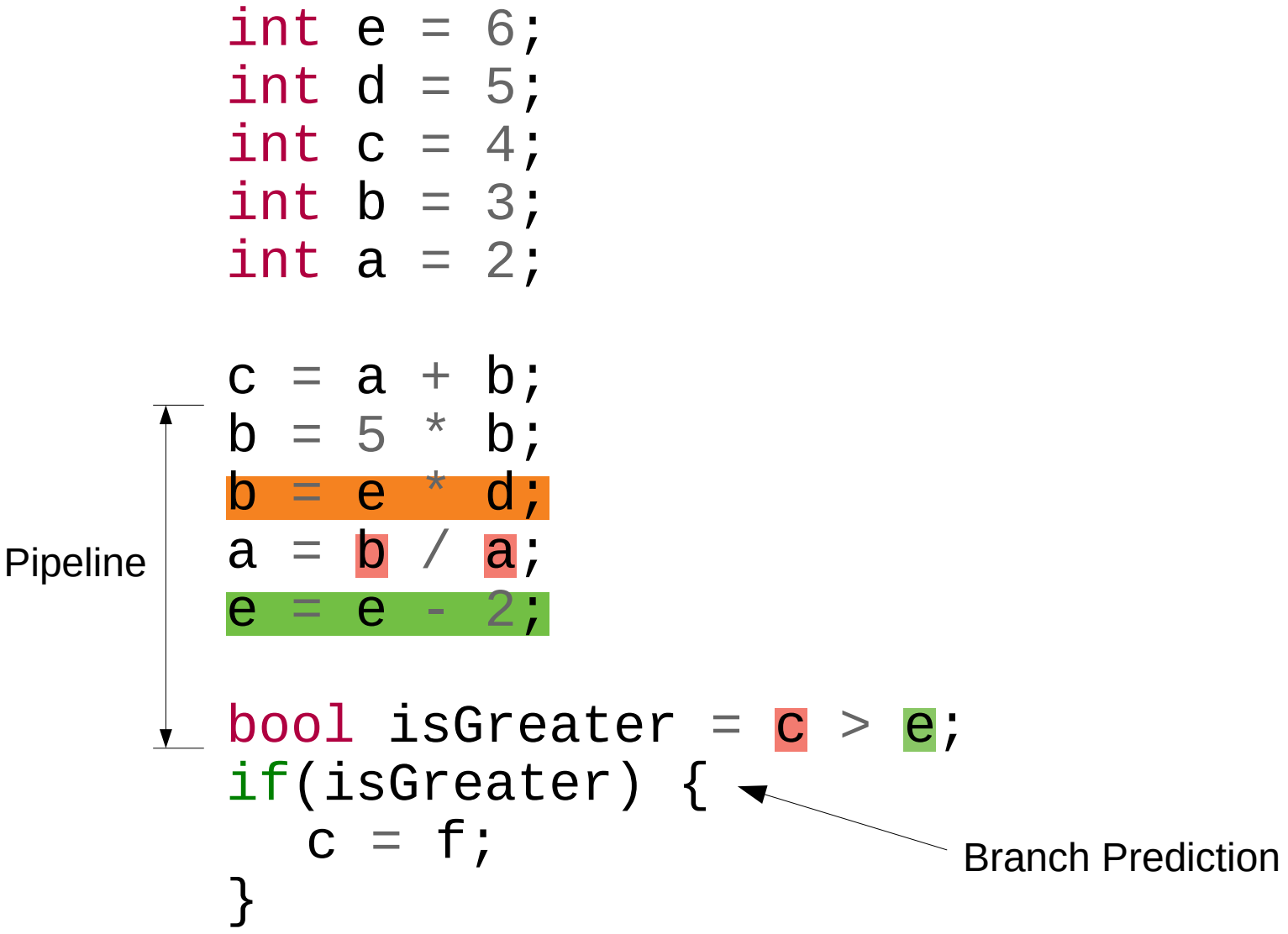
```
int e = 6;
int d = 5;
int c = 4;
int b = 3;
int a = 2;

c = a + b;
b = 5 * b;
b = e * d;
a = b / a;
e = d - 2;

bool isGreater = c > e;
if(isGreater) {
   c = f;
}
```

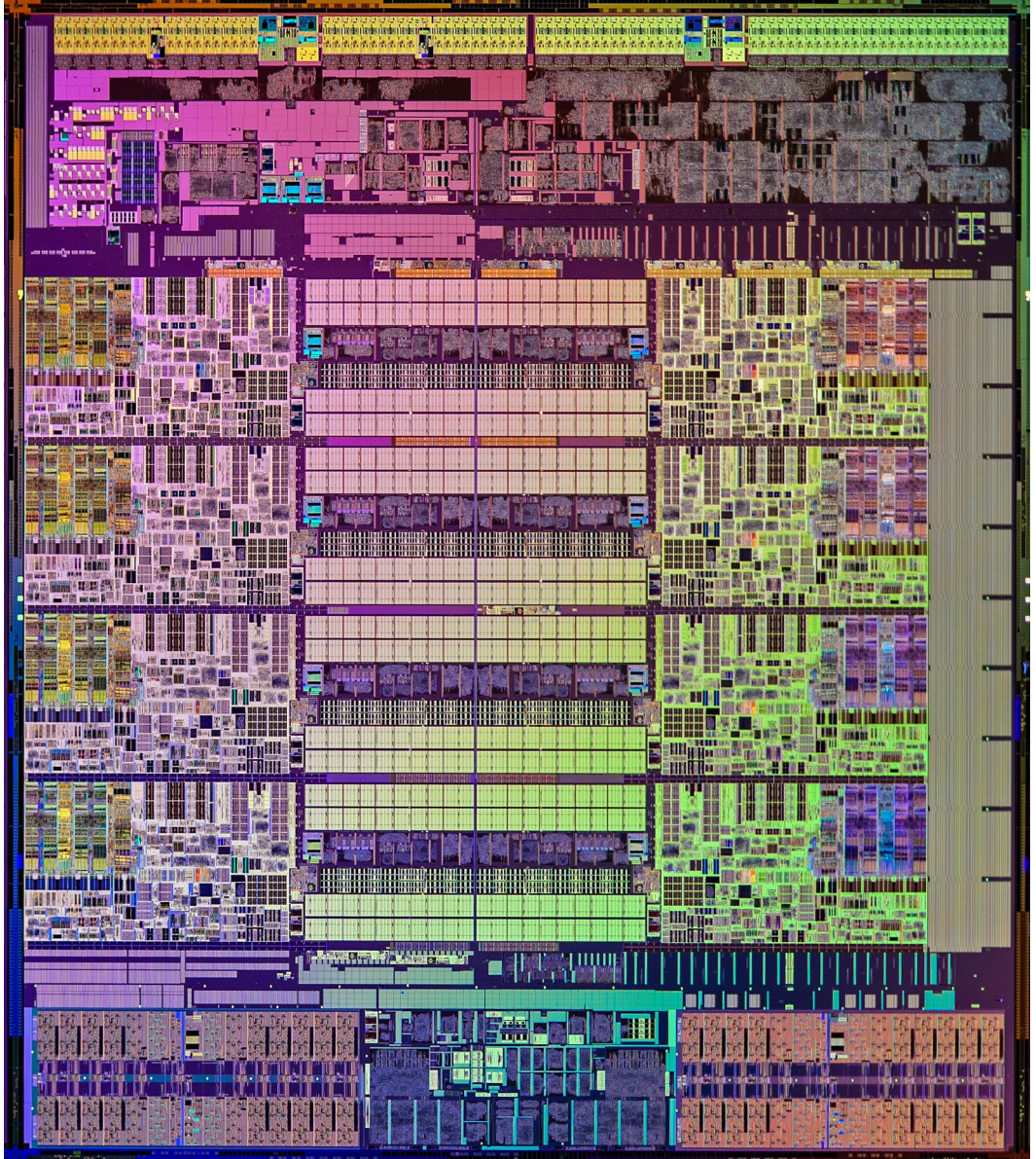

Pipeline
int e = 6;
int d = 5;
int c = 4;
int b = 3;
int a = 2;
c = a + b;
b = 5 * b;
b = e * d;
a = b / a;
e = e - 2;
bool isGreater = c > e;
if(isGreater) {
c = f;
}


Pipeline
int e = 6;
int d = 5;
int c = 4;
int b = 3;
int a = 2;
c = a + b;
b = 5 * b;
b = e * d;
a = b / a;
e = e - 2;
bool isGreater = c > e;
if(isGreater) {
c = f;
}


int e = 6;
int d = 5;
int c = 4;
int b = 3;
int a = 2;
c = a + b;
b = 5 * b;
b = e * d;
a = b / a;
e = e - 2;
bool isGreater = c > e;
if(isGreater) {
c = f;
}
Pipeline
Branch Prediction

Amazing Hardware Feature™ #3:

CPU Cache



Which one is faster?

A

```
long sum2D(int* someArray, int size) {
    long sum = 0;
    for(int row = 0; row < size; row++) {
        for(int col = 0; col < size; col++) {
            sum += someArray[col * size + row];
        }
    }
    return sum;
}
```

B

```
long sum2D(int* someArray, int size) {
    long sum = 0;
    for(int col = 0; col < size; col++) {
        for(int row = 0; row < size; row++) {
            sum += someArray[col * size + row];
        }
    }
    return sum;
}
```

Size = 25000

A: 10.767 s

B: 0.352 s

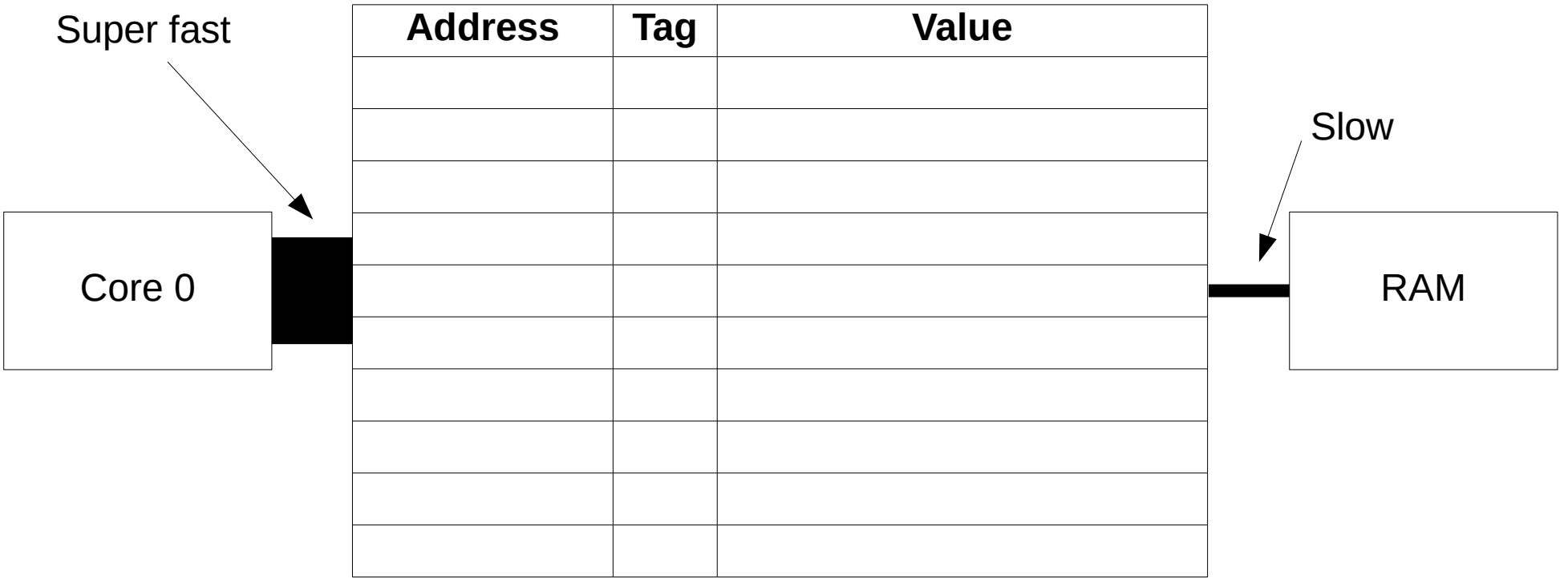
(That's more than a 30x difference)

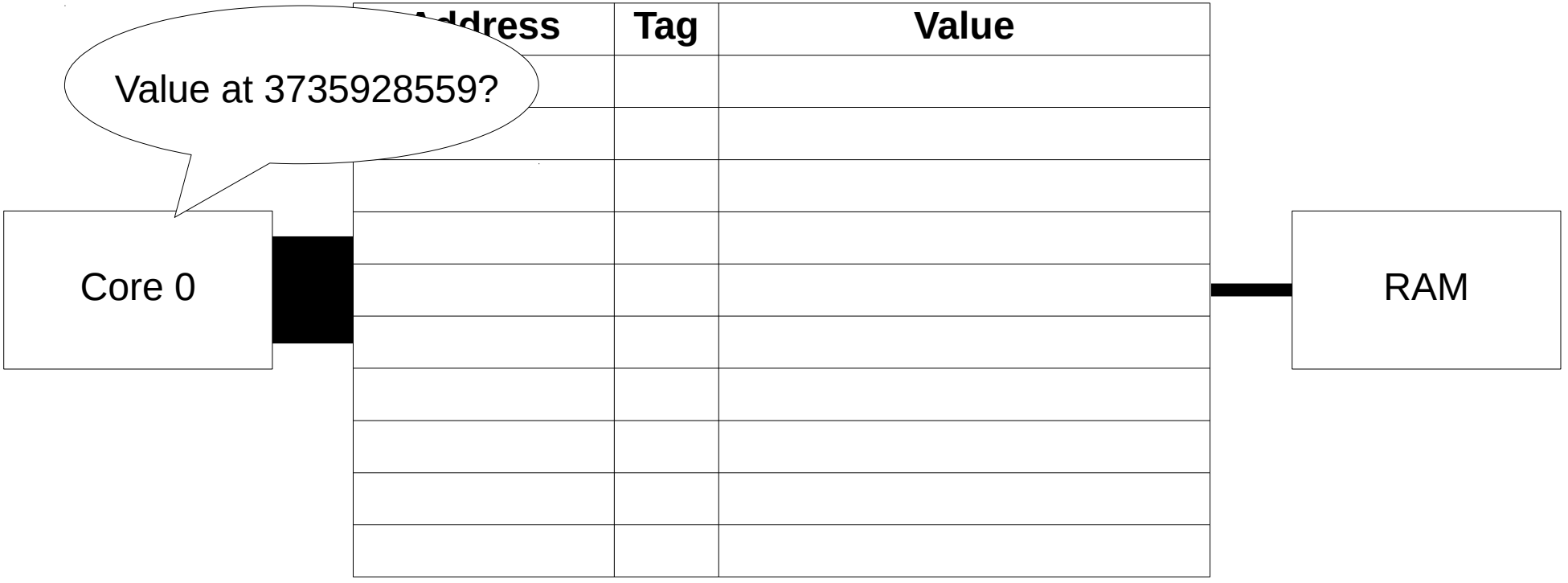
Cache works on two principles:

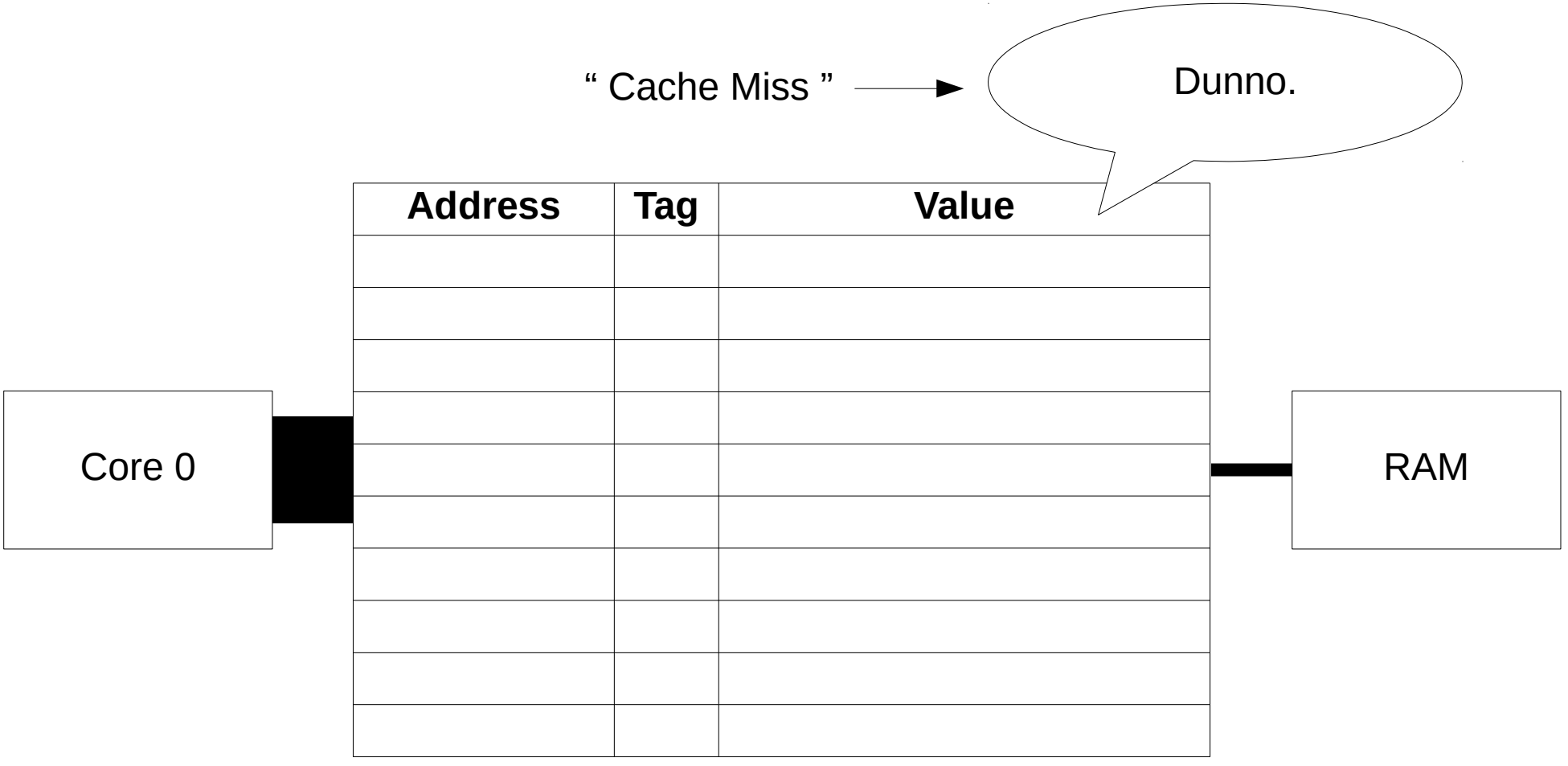
1. Spatial Locality

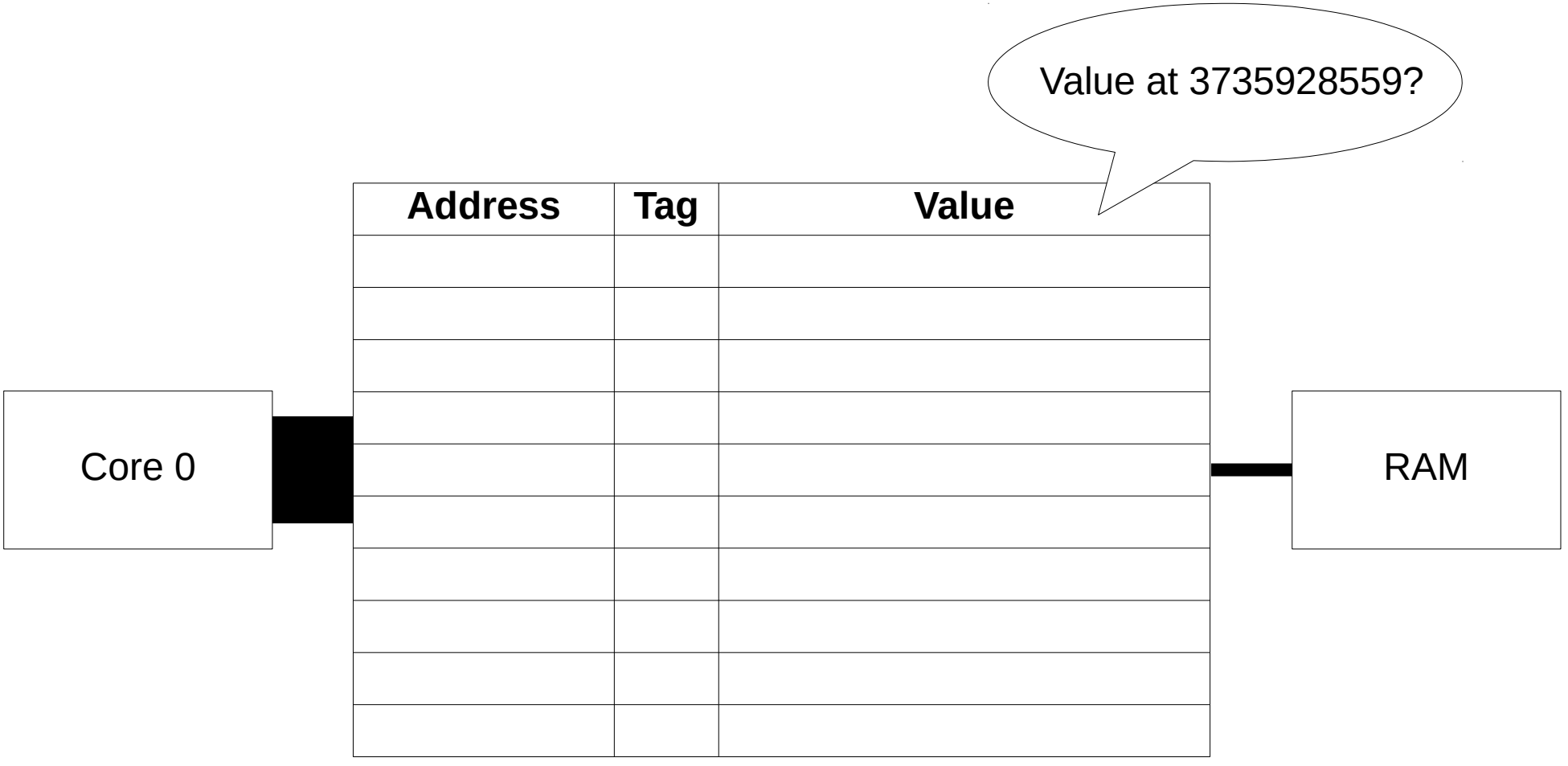
2. Temporal Locality

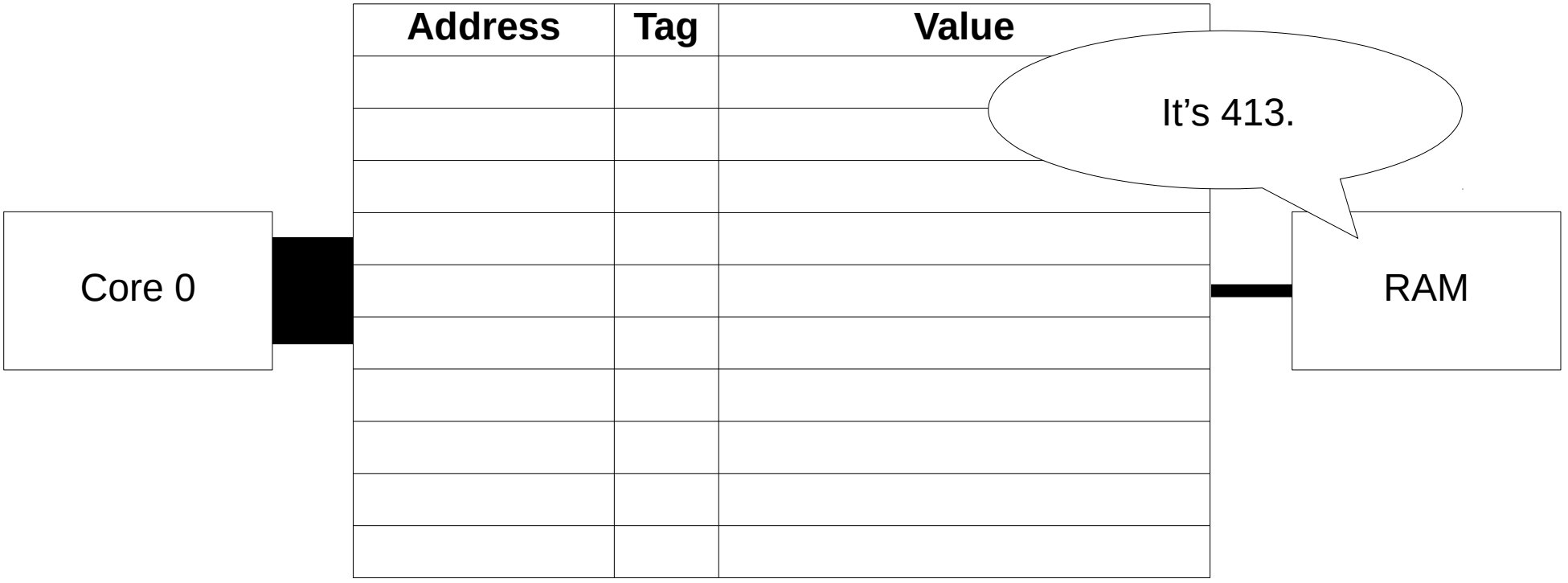
```
for(int i = 0; i < count; i++) {
   sum += someArray[i];
}
```


Super fast
Address
Tag
Value
Slow
Core 0
RAM


Value at 3735928559?
Core 0
Address
Tag
Value
RAM


“ Cache Miss ”
Dunno.
Address
Tag
Value
Core 0
RAM


Value at 3735928559?
Address
Tag
Value
Core 0
RAM

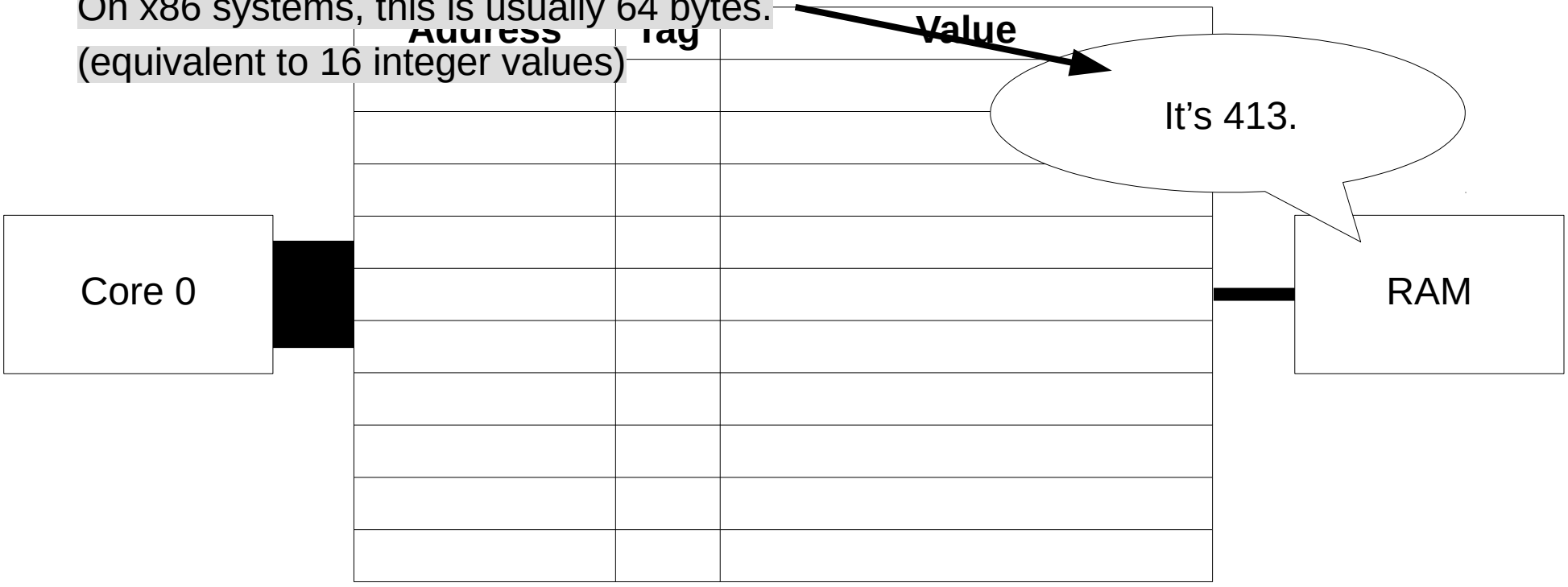

Address
Tag
Value
It’s 413.
Core 0
RAM

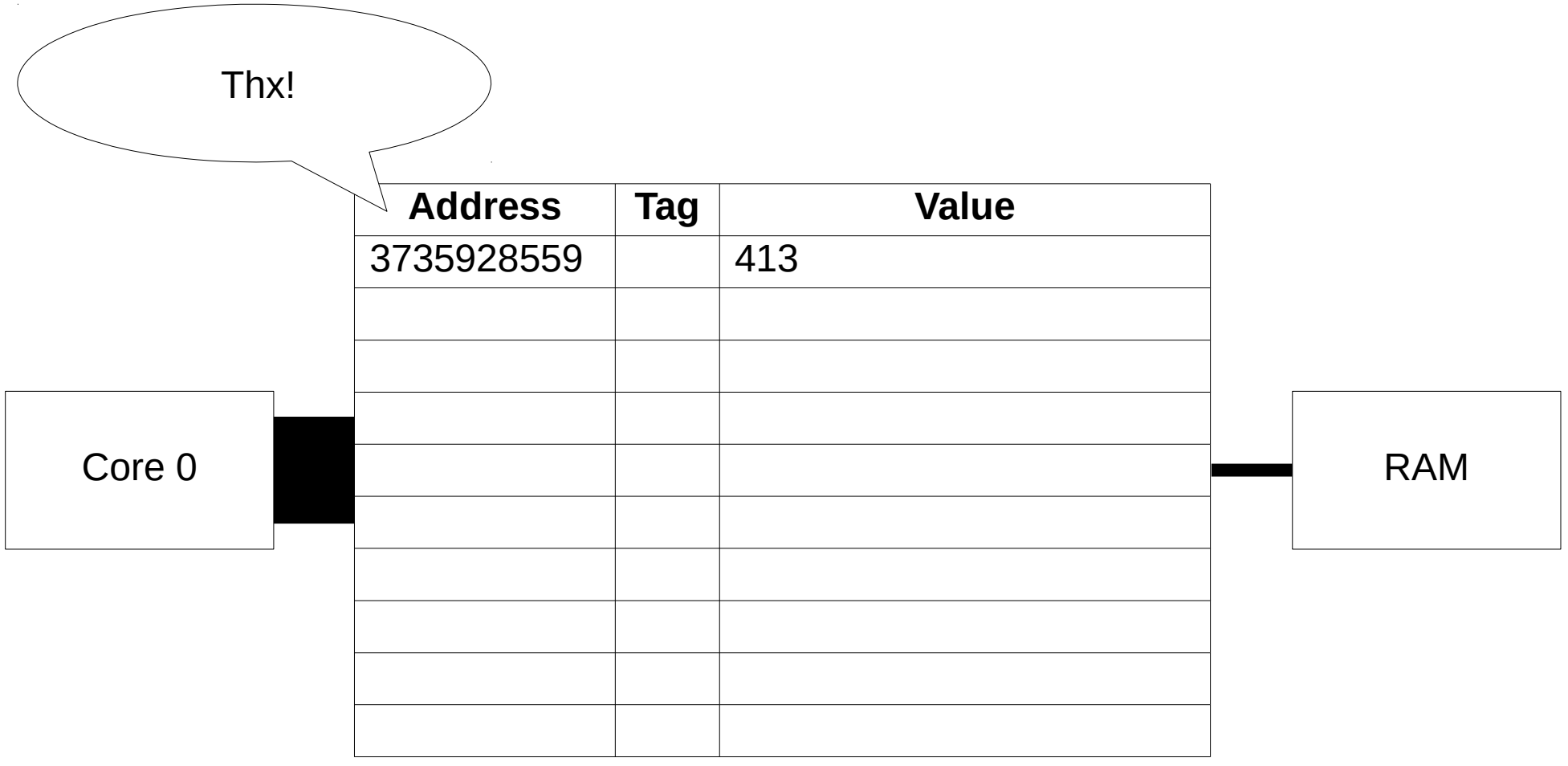
!! SIMPLIFICATION ALERT !!

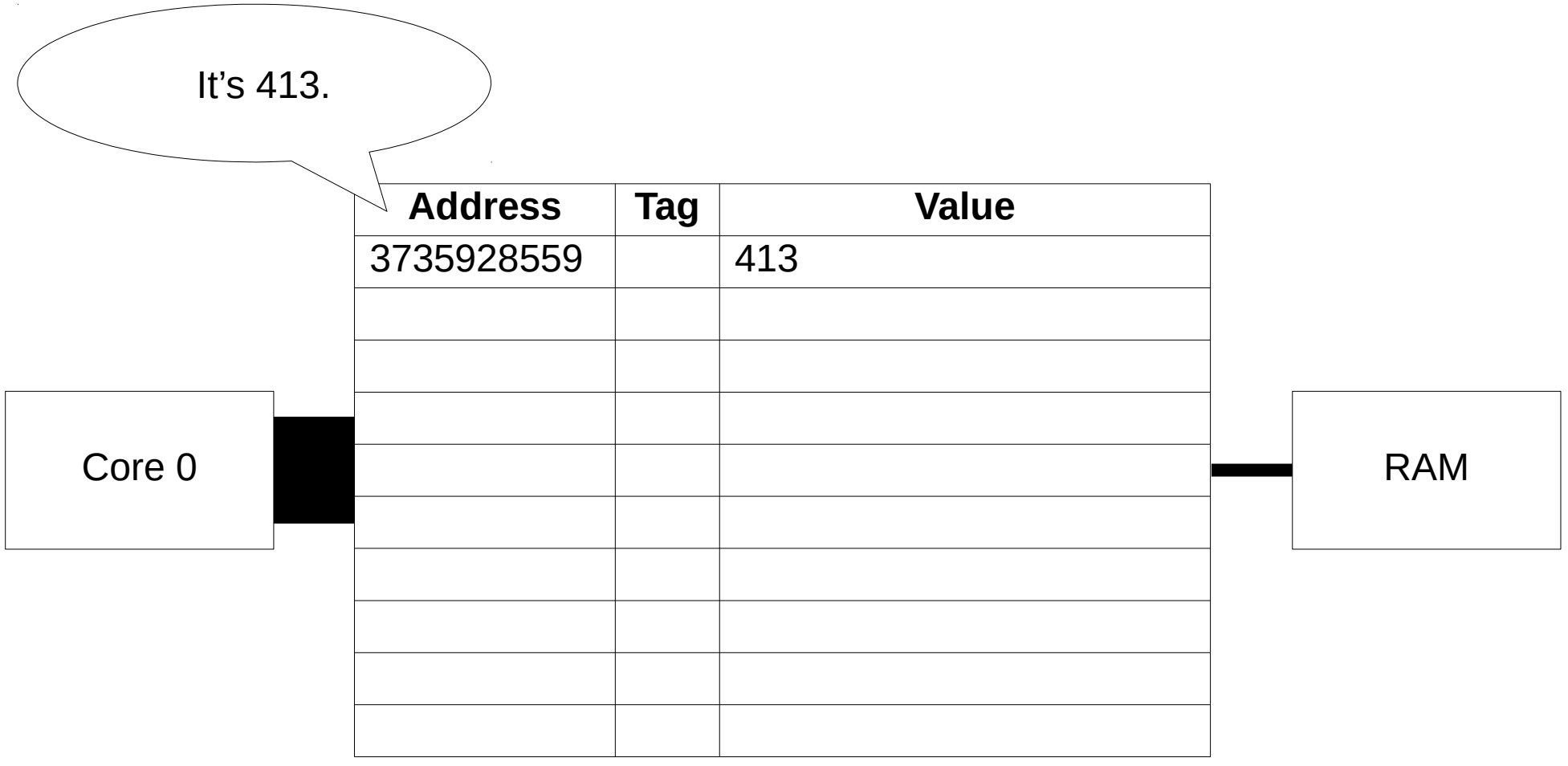
For modern processors, this is not just a single value, but a so-called “ Cache Line ”.

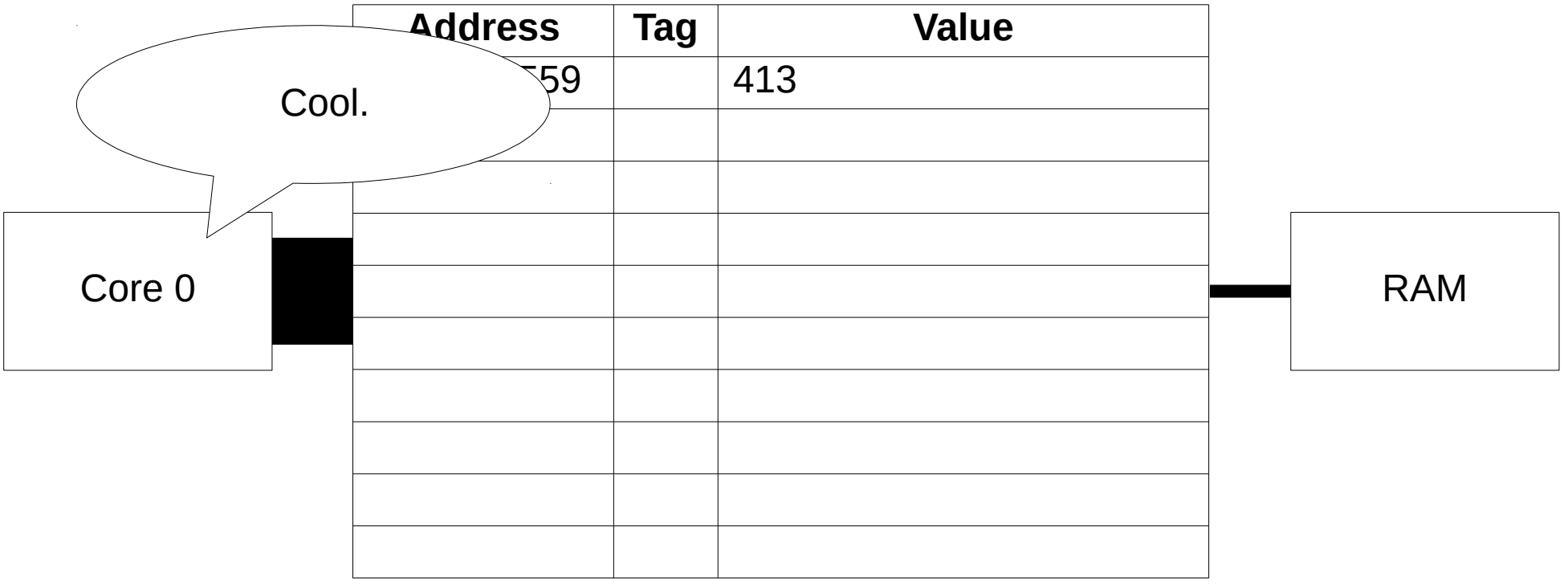
On x86 systems, this is usually 64 bytes.

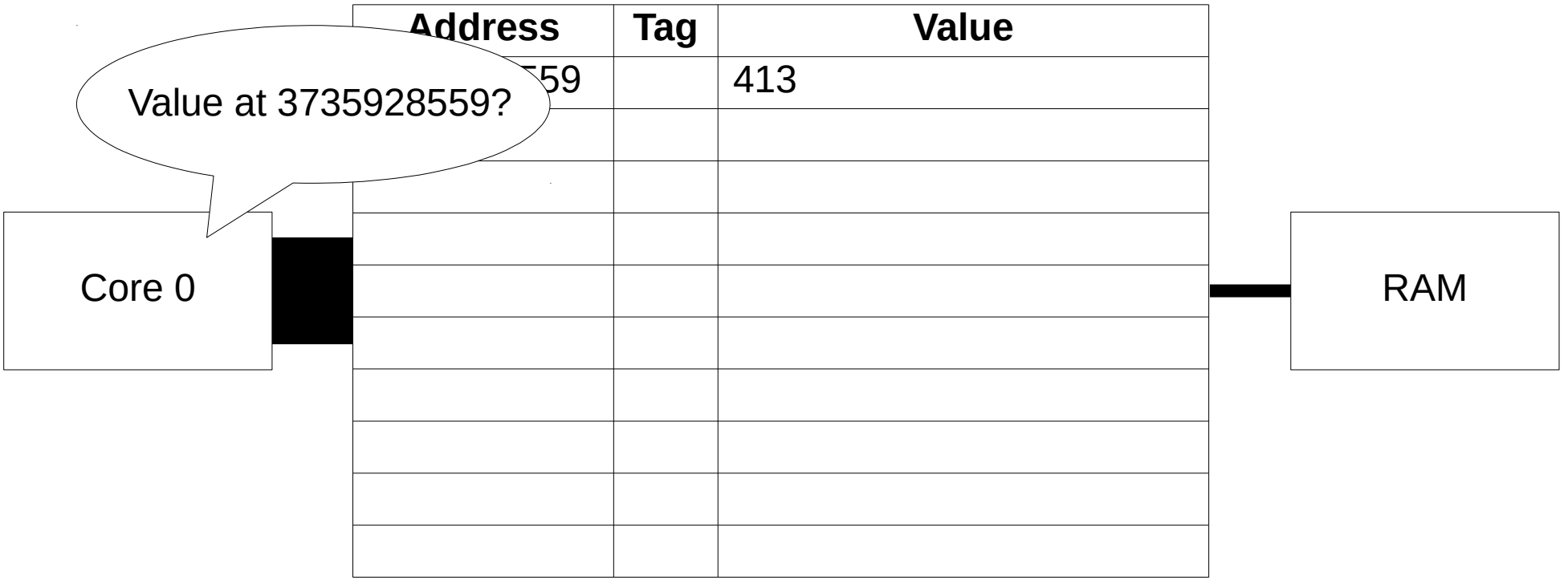
(equivalent to 16 integer values)

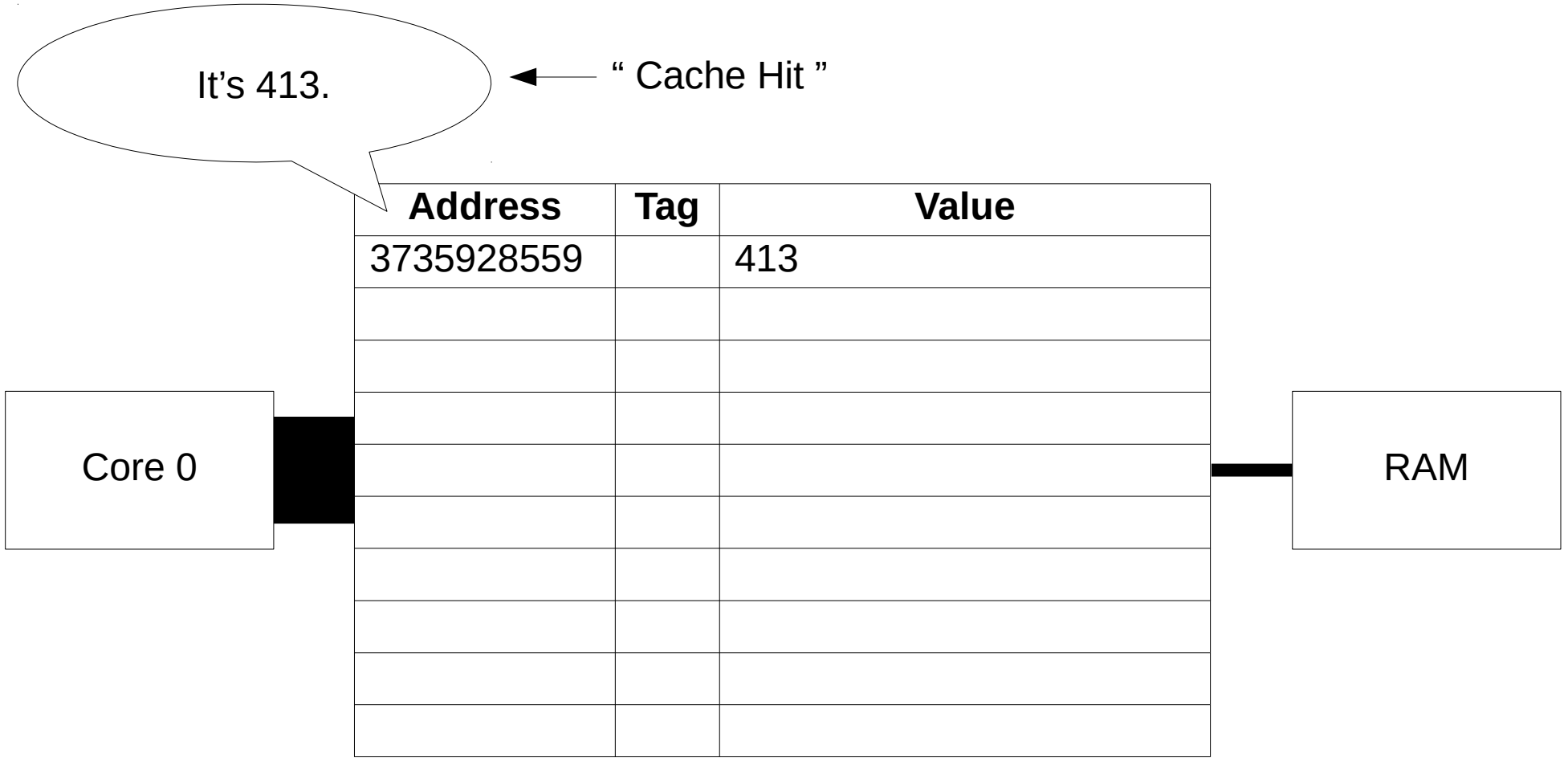


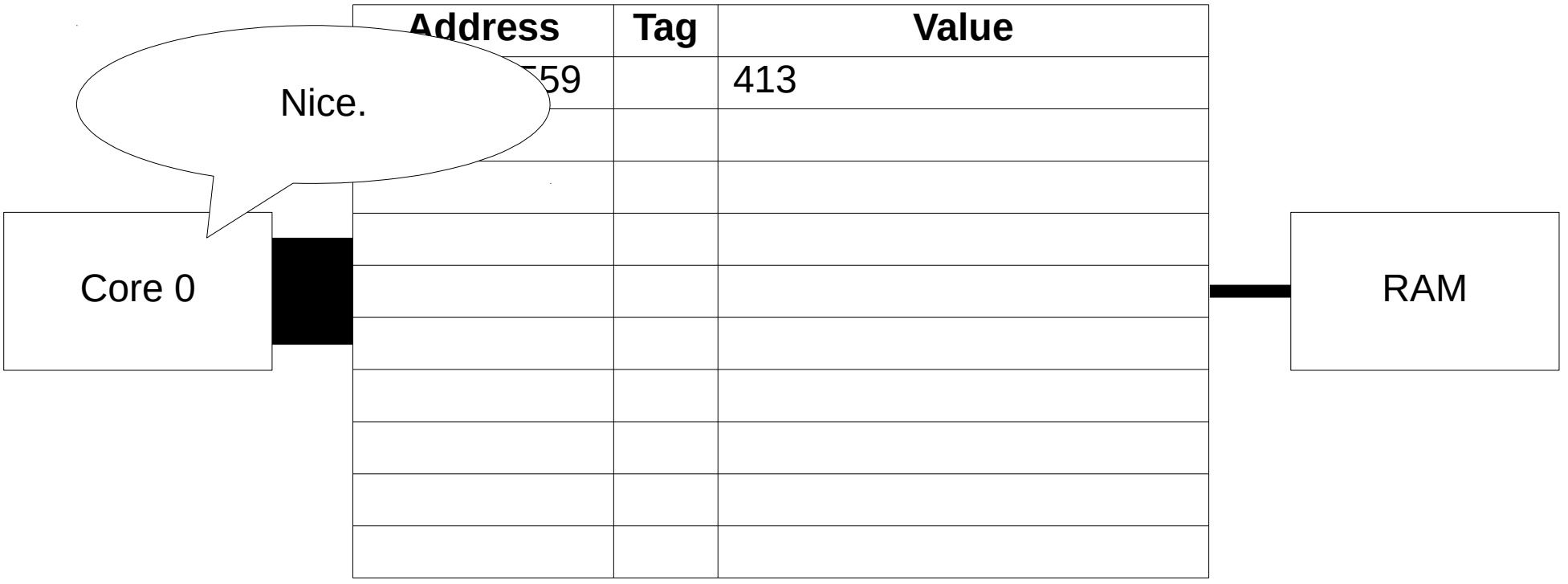

Thx!
Address
Tag
Value
3735928559
413
Core 0
RAM

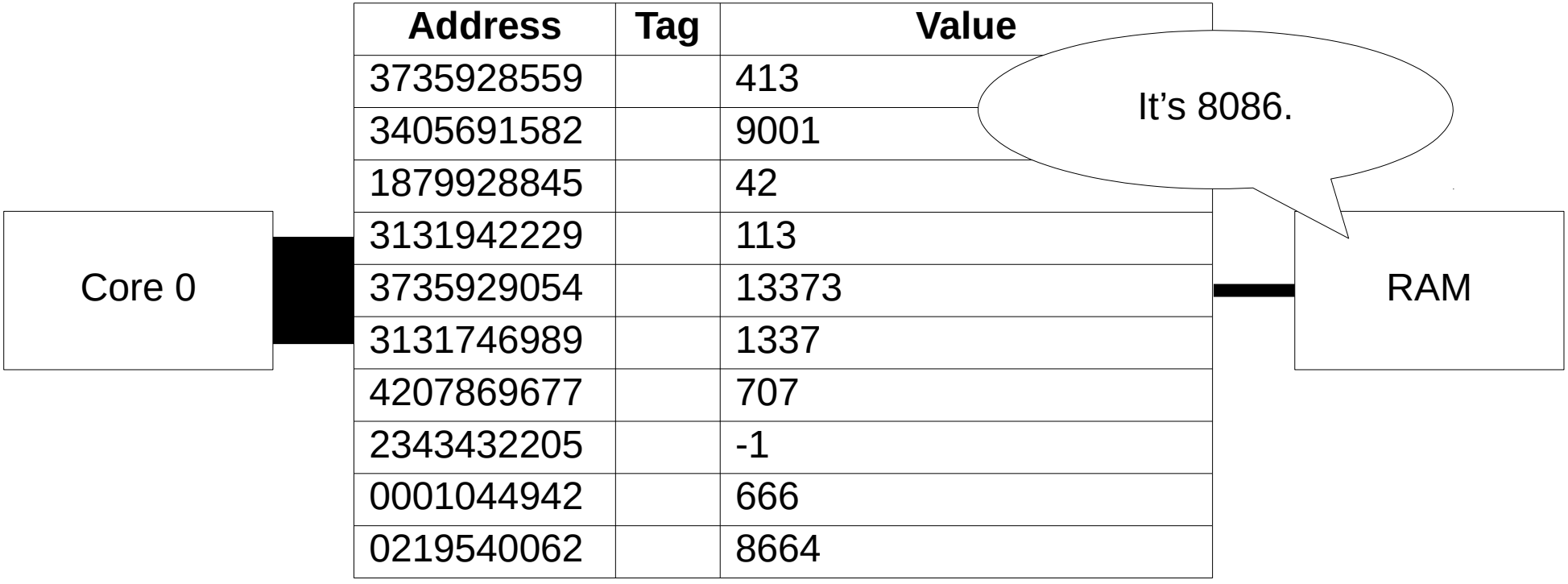

It’s 413.
Address
Tag
Value
3735928559
413
Core 0
RAM

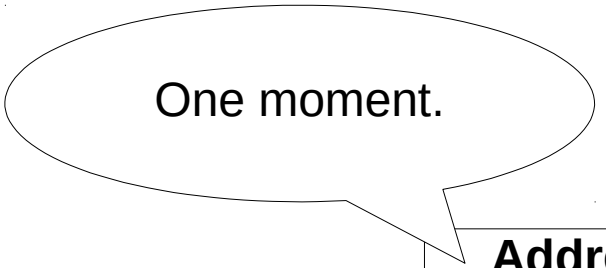

Cool.
Core 0
Address
Tag
Value
559
413
RAM


Value at 3735928559?
Core 0
Address
Tag
Value
559
413
RAM


It’s 413.
“ Cache Hit ”
Address
Tag
Value
3735928559
413
Core 0
RAM


Address
Tag
Value
559
413
Nice.
Core 0
RAM

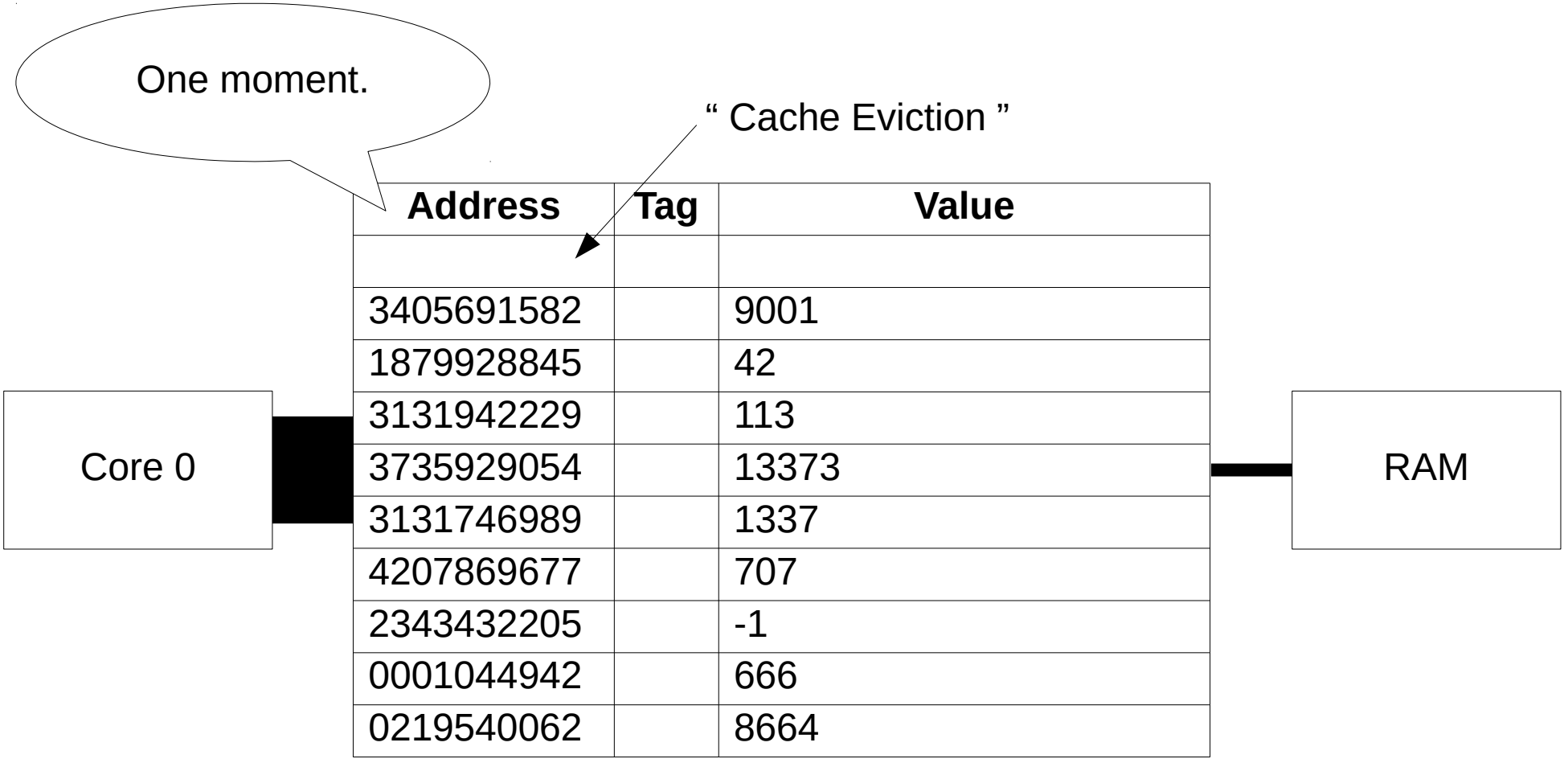

Address
Tag
Value
3735928559
413
3405691582
9001
1879928845
42
3131942229
113
3735929054
13373
3131746989
1337
4207869677
707
2343432205
-1
0001044942
666
0219540062
8664
It’s 8086.
Core 0
RAM

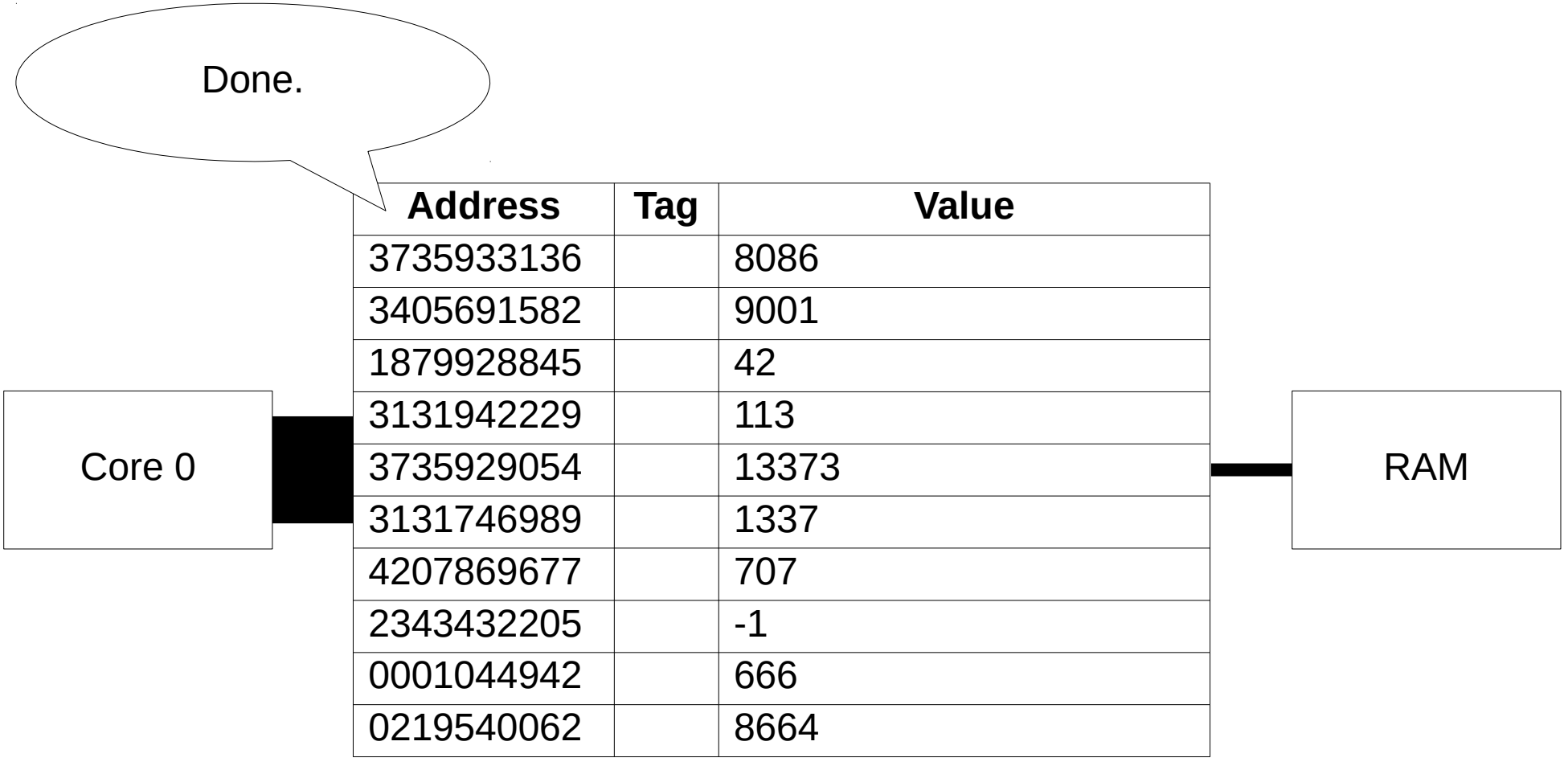


Core 0

Address	Tag	Value
3735928559		413
3405691582		9001
1879928845		42
3131942229		113
3735929054		13373
3131746989		1337
4207869677		707
2343432205		-1
0001044942		666
0219540062		8664

RAM


One moment.
“ Cache Eviction ”
Address
Tag
Value
3405691582
9001
1879928845
42
3131942229
113
3735929054
13373
3131746989
1337
4207869677
707
2343432205
-1
0001044942
666
0219540062
8664
Core 0
RAM


Done.
Address
Tag
Value
3735933136
8086
3405691582
9001
1879928845
42
3131942229
113
3735929054
13373
3131746989
1337
4207869677
707
2343432205
-1
0001044942
666
0219540062
8664
Core 0
RAM

Caches on modern processors tend to be n-way set associative

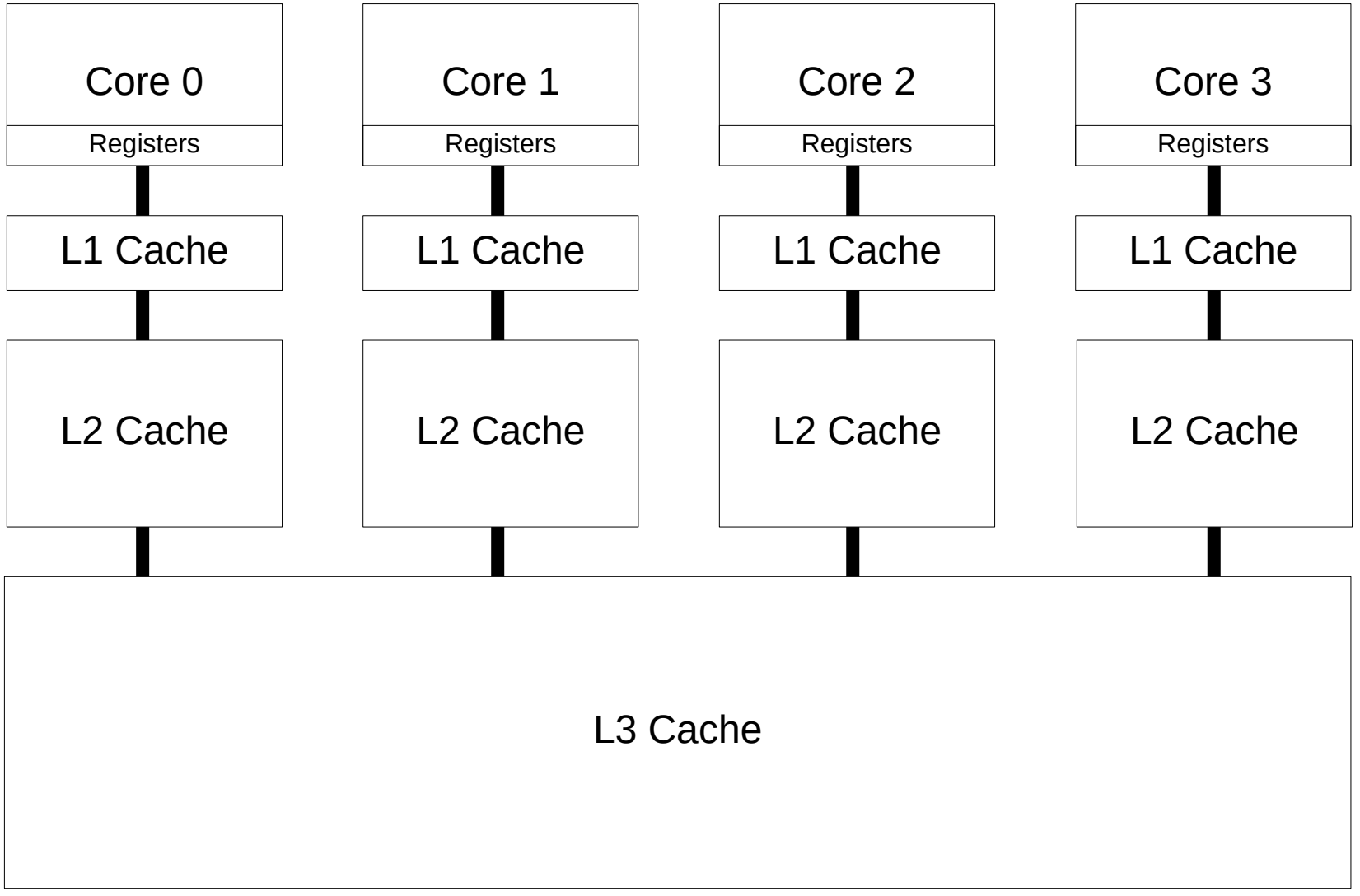
Set 0

Address	Tag	Value
3405691582		9001
3735929054		13373
0001044942		666
0219540062		8664

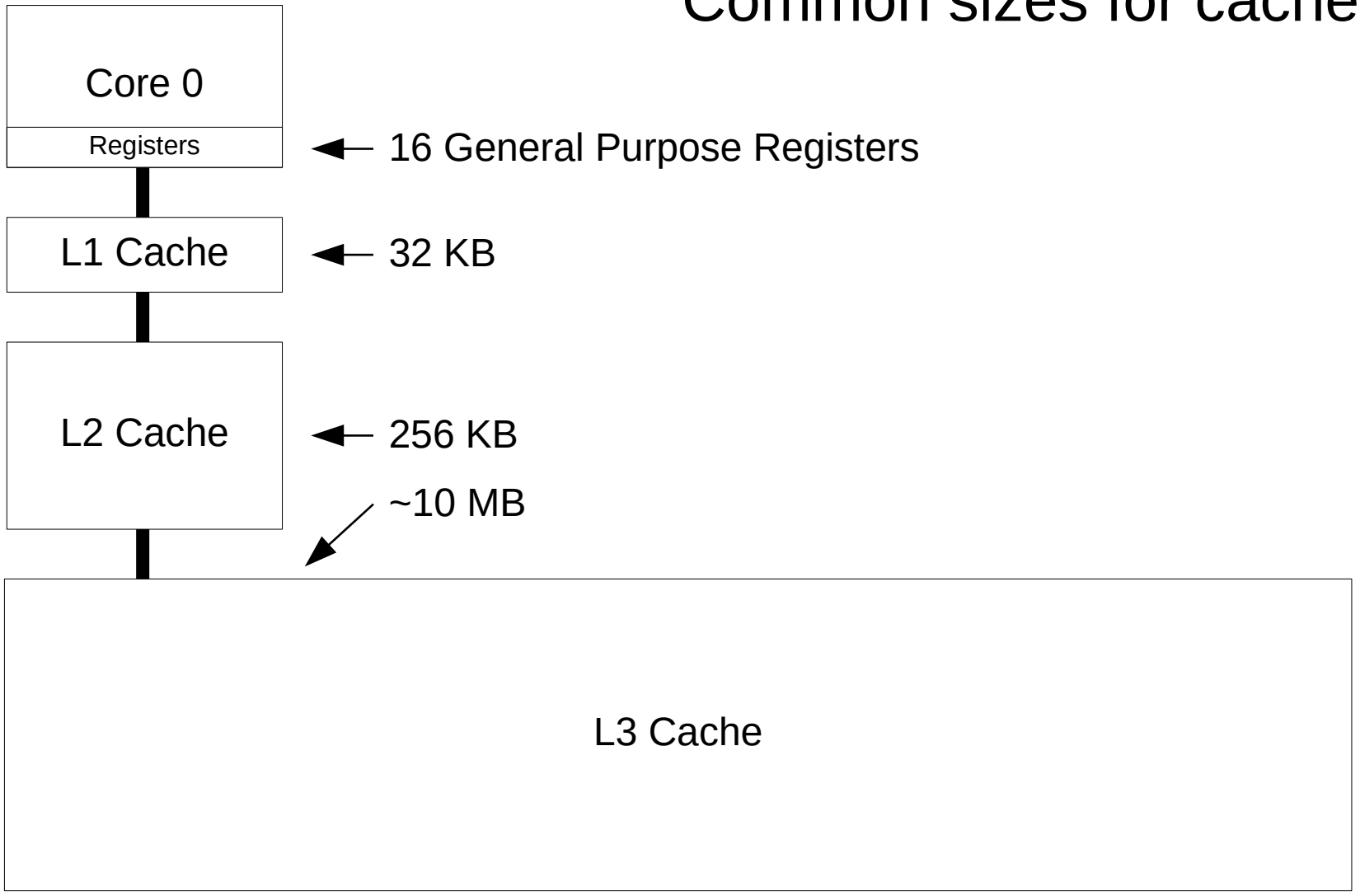
Set 1

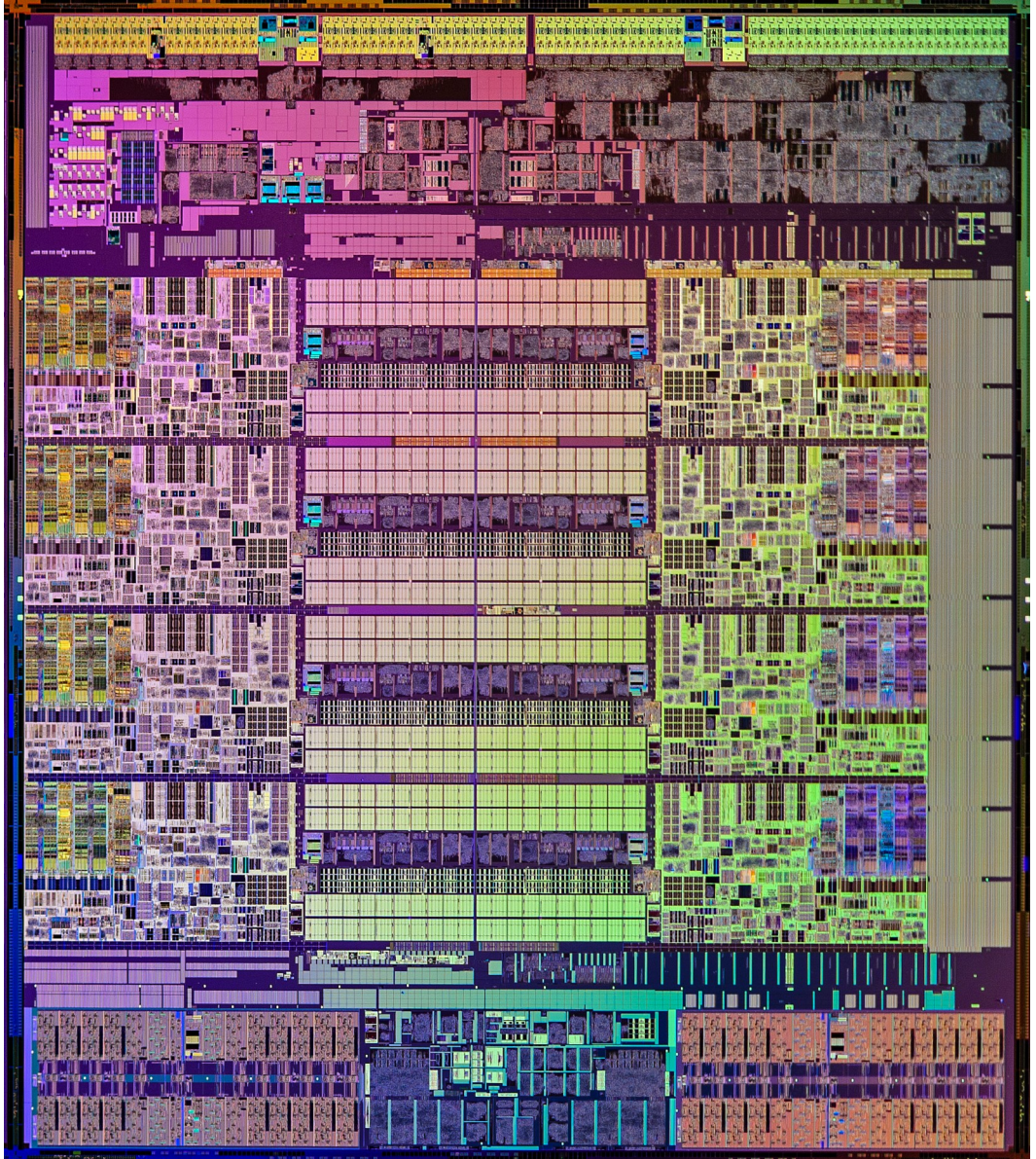
Address	Tag	Value
3735933136		8086
1879928845		42
3131942229		113
3131746989		1337
4207869677		707
2343432205		-1

Two sets means 2-way set associative


Core 0
Registers
L1 Cache
L2 Cache
Core 1
Registers
L1 Cache
L2 Cache
Core 2
Registers
L1 Cache
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
L3 Cache

Common sizes for cache levels





Measuring Execution Time

$$Speedup = \frac{t_{slow}}{t_{fast}}$$

```cpp
#include <chrono>

auto start = std::chrono::high_resolution_clock::now();
slowFunction();
auto end = std::chrono::high_resolution_clock::now();

auto time =
    std::chrono::duration_cast<std::chrono::milliseconds>(end – start).count();

std::cout << "Execution took " << time << " ms" << std::endl;
```

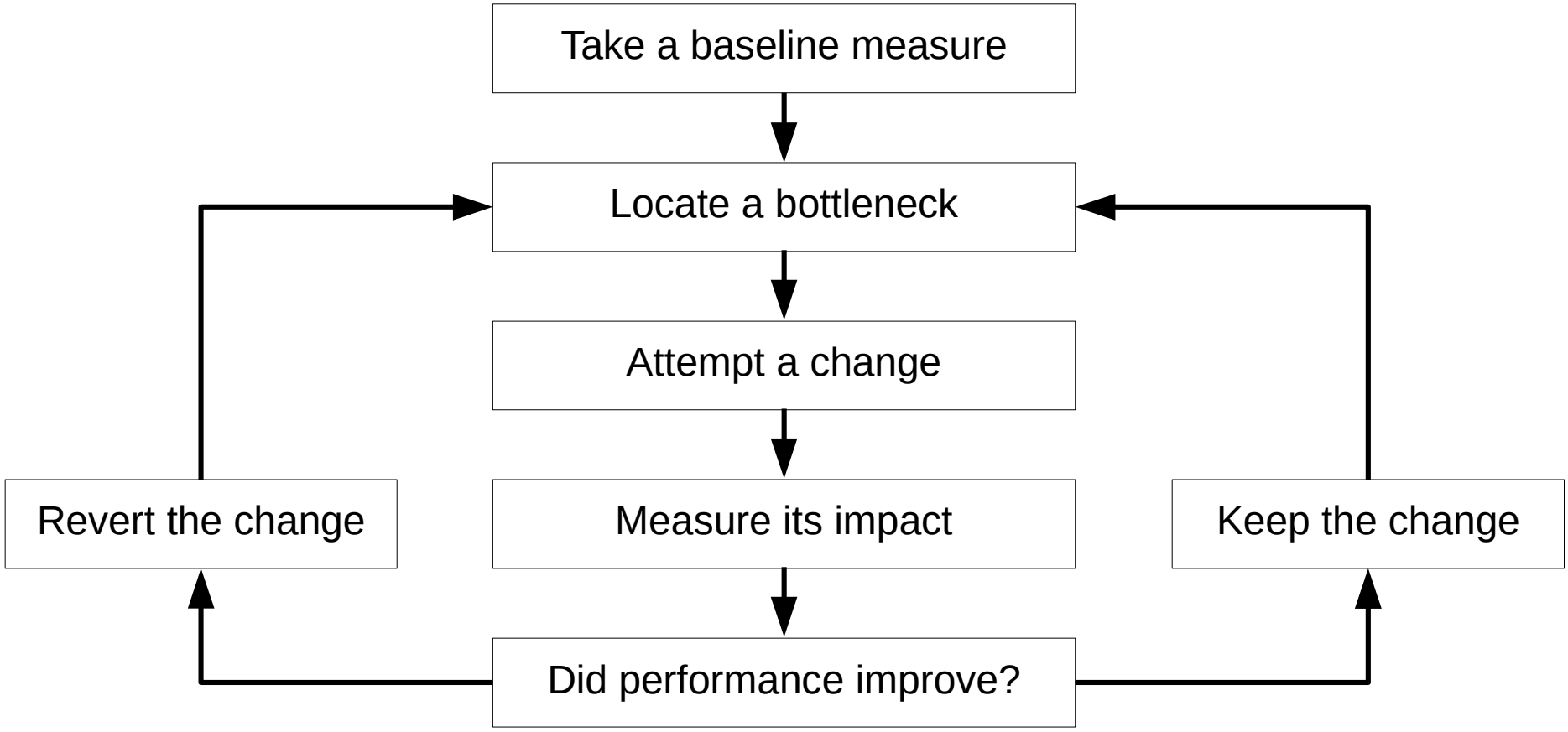
The 90 / 10 rule

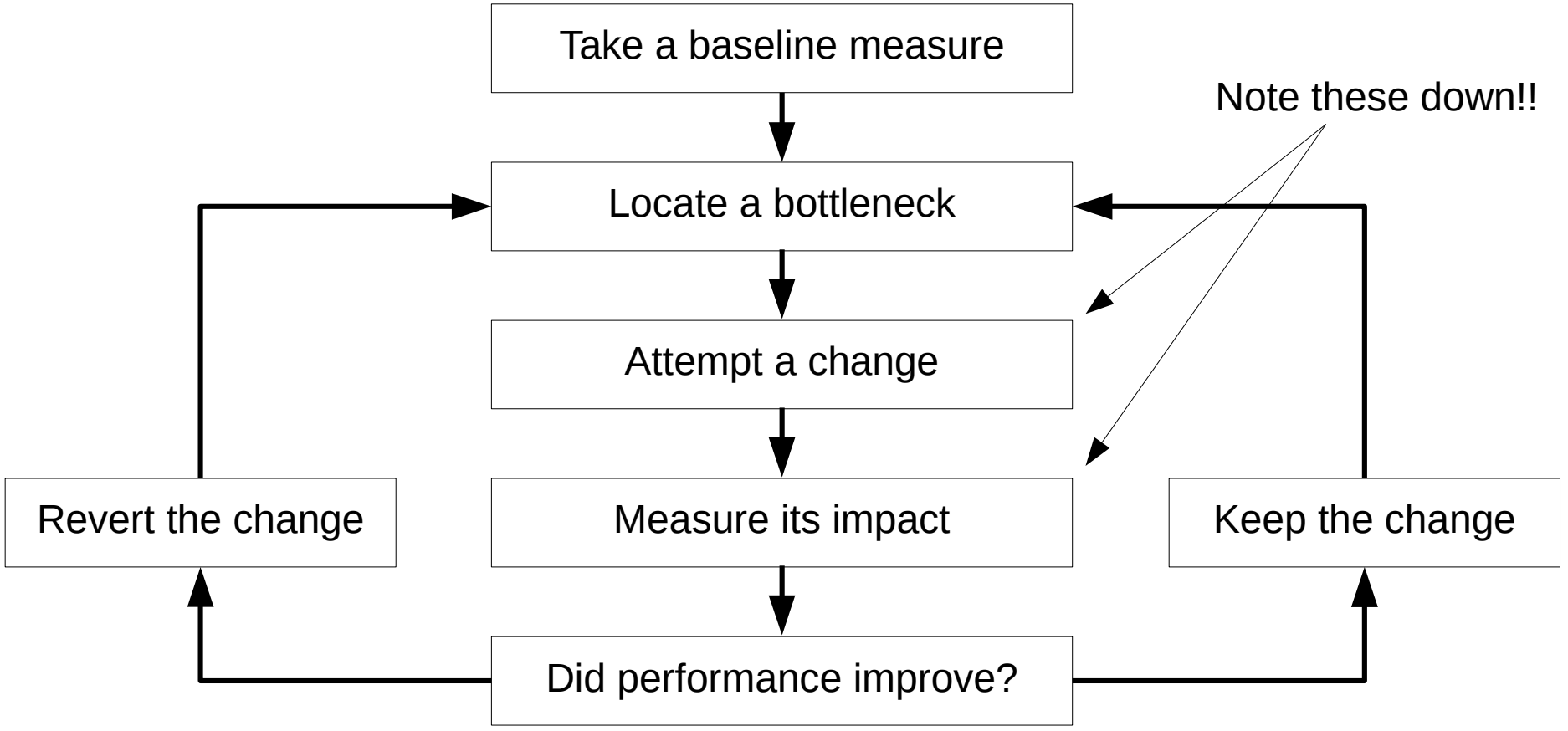
Usually a “ Bottleneck ”

Bottleneck: Section of a program where an executing thread spends most of its time

Latency != Throughput

The Best™ Optimisation Process


Take a baseline measure
Locate a bottleneck
Attempt a change
Measure its impact
Did performance improve?
Revert the change
Keep the change


Take a baseline measure
Note these down!!
Locate a bottleneck
Attempt a change
Revert the change
Measure its impact
Keep the change
Did performance improve?

Keep a Lab Journal!

Common approaches to implement profilers:

Sampling

Code instrumentation

Sampling Profiler:

Interrupts program frequently

Records where interrupt happened

Code Instrumentation:

Code is compiled with profiling instructions

Instructions create profiling results table

How not to measure runtimes

for dummies

```
auto start = time_now();
int sum = 3 + 5;
auto end = time_now();
std::cout << "Time taken: " << (end - start) << std::endl;
```

```
auto start = time_now();

long sum = 0;

for(int i = 0; i < 50000000; i++) {
    sum += i;
    std::cout << sum << std::endl;
}

auto end = time_now();

std::cout << "Time taken: " << (end - start) << std::endl;
```

```
long count = computeRandomLong();

auto start = time_now();
long sum = 0;

for(long i = 0; i < count; i++) {
    sum += i;
}

auto end = time_now();

std::cout << "Time taken: " << (end - start) << std::endl;
```

```
auto start = time_now();

for(int i = 0; i < 50000000; i++) {
   expensiveFunctionTheProgramRarelyCalls();
}

auto end = time_now();

std::cout << "Time taken: " << (end - start) << std::endl;
```

```
auto start = time_now();

readFile("tinyTextFile.txt");

auto end = time_now();

std::cout << "Time taken: " << (end - start) << std::endl;
```

```
auto start = time_now();

cheapFunction();

auto end = time_now();

for(int i = 0; i < 500000; i++) {
   expensiveFunction();
}

std::cout << "Time taken: " << (end - start) << std::endl;
```

Demonstration of valgrind + kcachegrind

Measure

Know your hardware

Keep a Lab Journal