You must be logged in PollEv to get credit

Participation

The “participation” segment of your grade has three main components:

- 4% for Lecture attendance, as measured by occasional Poll Everywhere polls. Starting on Oct. 20th, you need to be **logged into an account assoicated with your Cornell NetID** for your Poll Everywhere participation to count. This is the only way for us to reliably identify who participated.
- 4% for lab attendance, as recorded by the lab’s instructors.
- 2% for surveys:
    - The introduction survey (on Gradescope) in the first week of class.
    - The mid-semester feedback survey.
    - The semester-end course evaluation.

We know that life happens, so you can miss up to 3 lab sections and 5 lectures without penalty.

CS3410: Computer Systems and Organization

LEC15: Caches (Vol. I)

Professor Giulia Guidi
Monday, October 20, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

Final Exam

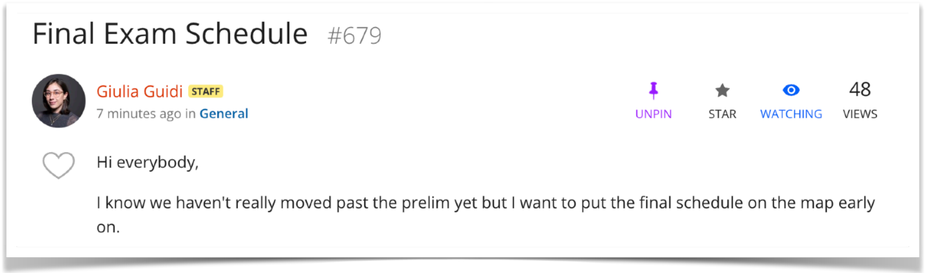
The **regular final** is on **Saturday, December 13, 7-9 PM**

The **early final** is on **Saturday, December 13, 4:30-6:30 PM**

The **make-up final** is on **Friday, Decemeber 12, 9-11 AM**

Check conflict early and let us know **by December 1** which exam you plan to take—**no** other make-up will be scheduled.

There's **no** weight transfer for the final or make-up final.

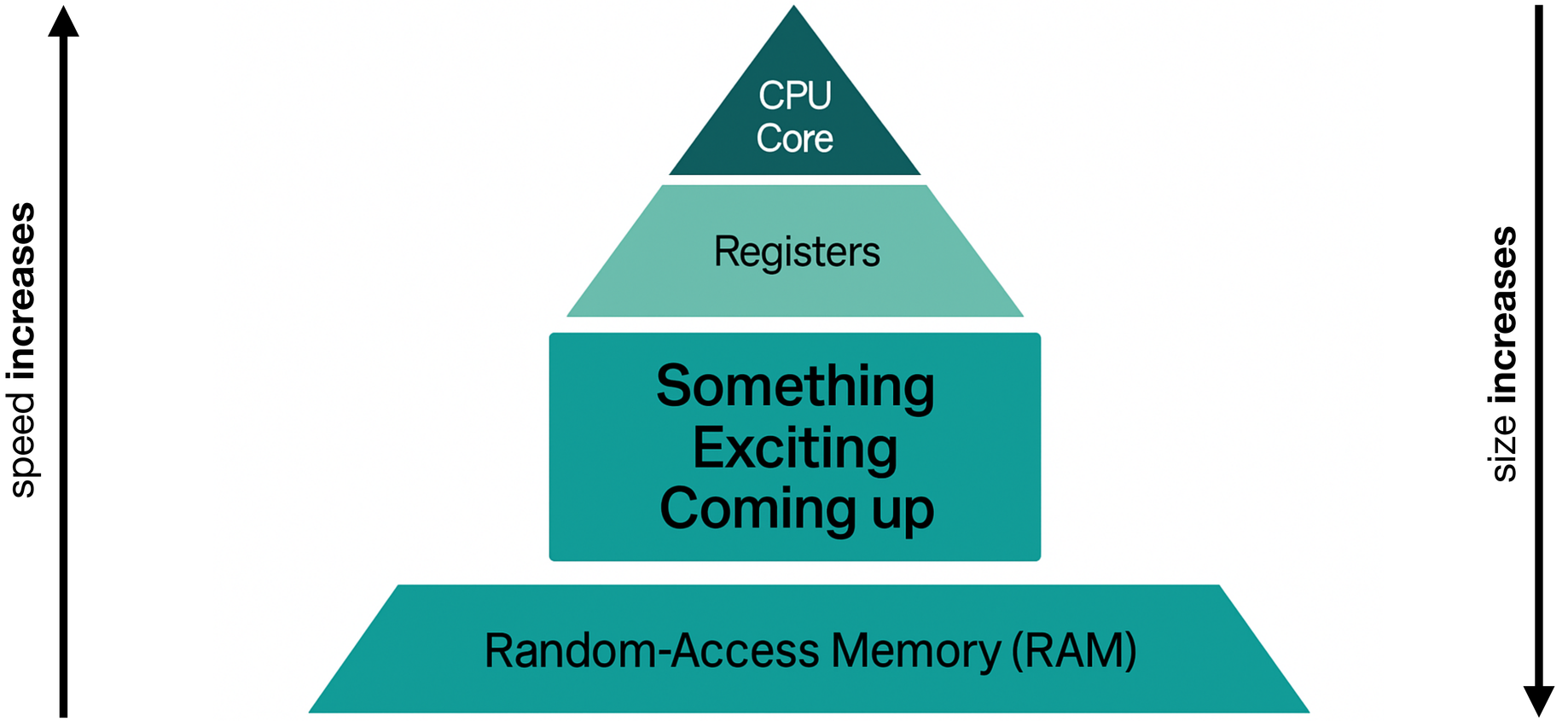


Plan for Today

- Introduction to Caches
- Direct-mapped Caches

On to caches (that “something great” mentioned earlier)

Principle of Locality and Memory Hierarchy



Register versus Memory

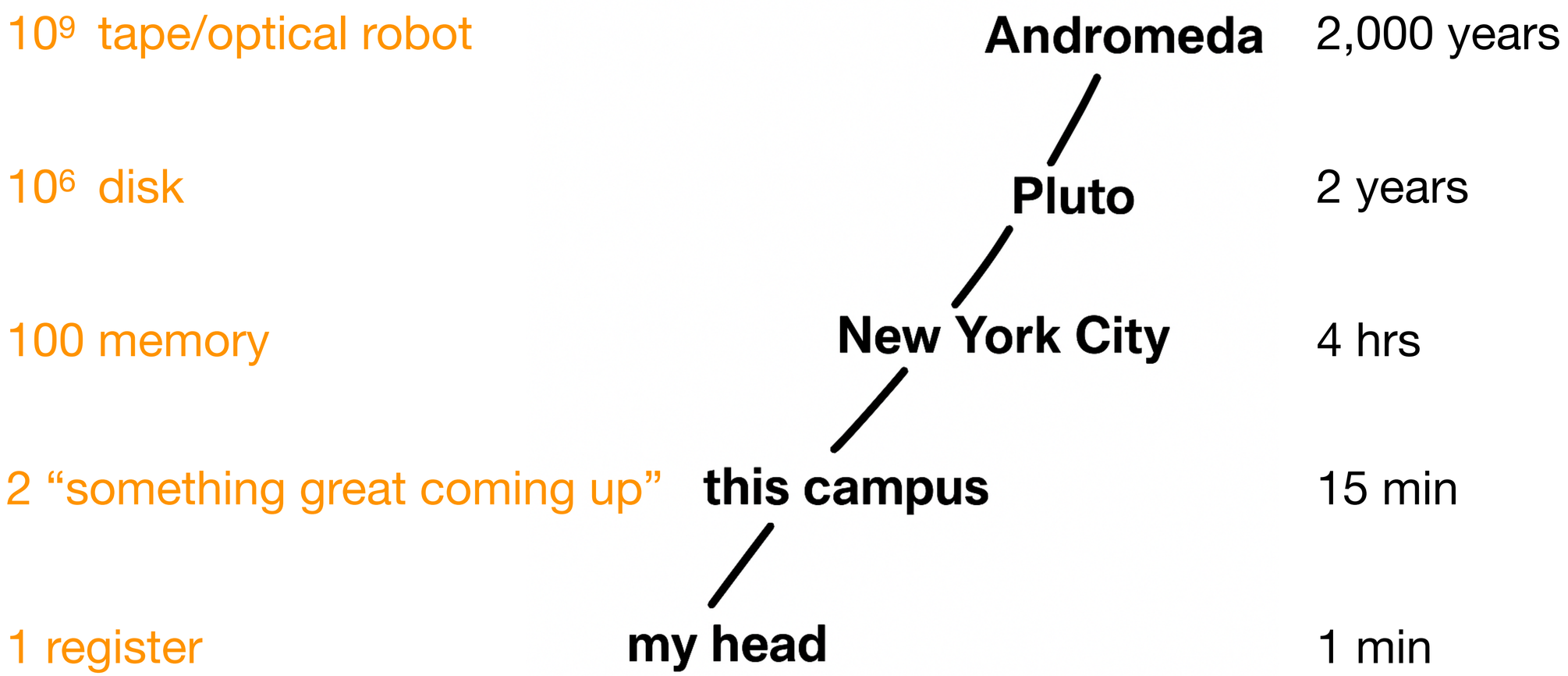
Given that:

- Registers: 256 bytes is RV64 or 128 bytes if RV32
- DRAM (data memory): billions of bytes (2-96 GB on a typical laptop)

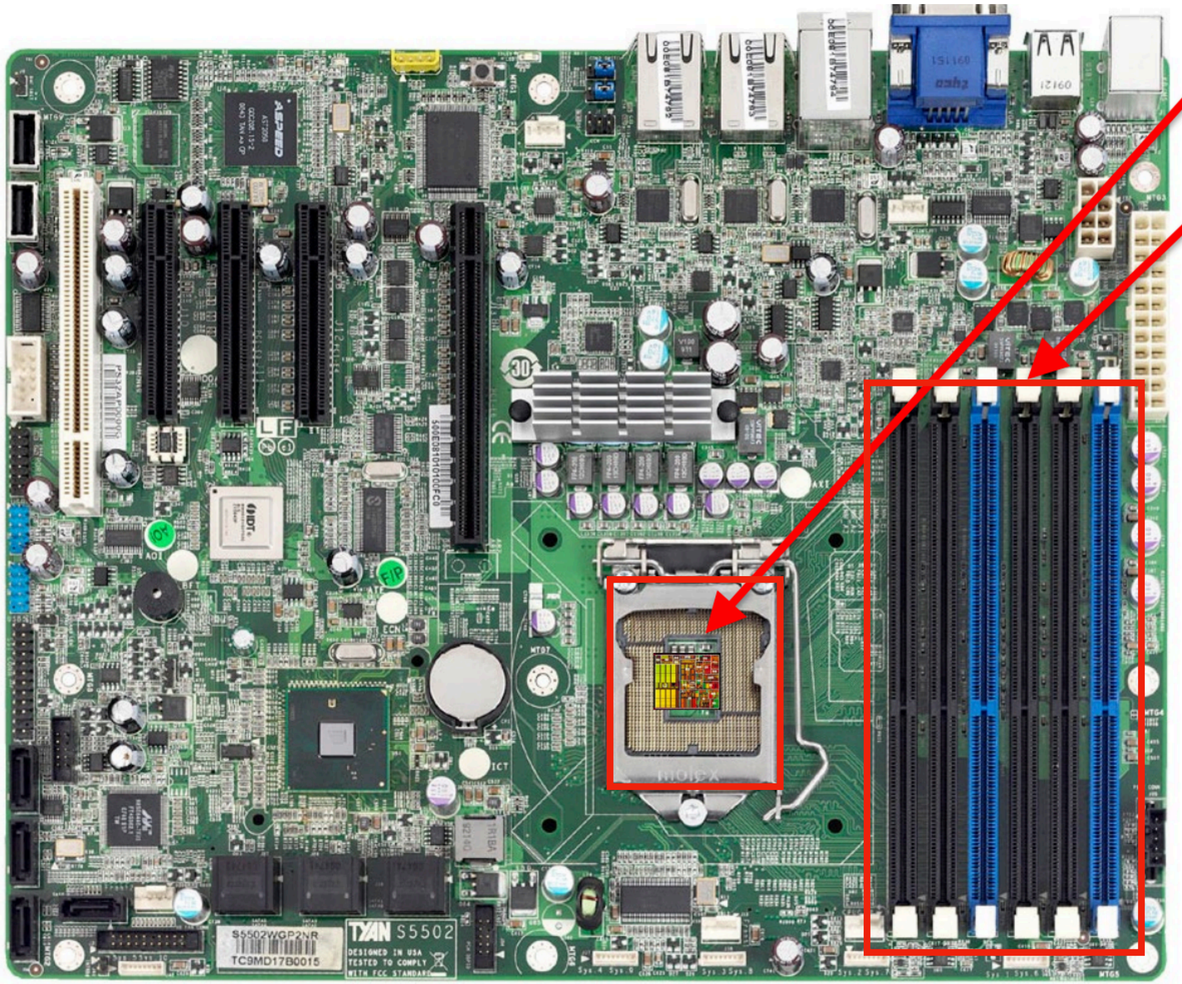
Physics dictates that **smaller is faster**

Registers are **50-500 times faster** than DRAM (one access latency, tens of ns)!

Register versus Memory



The problem is that memory is far from CPU



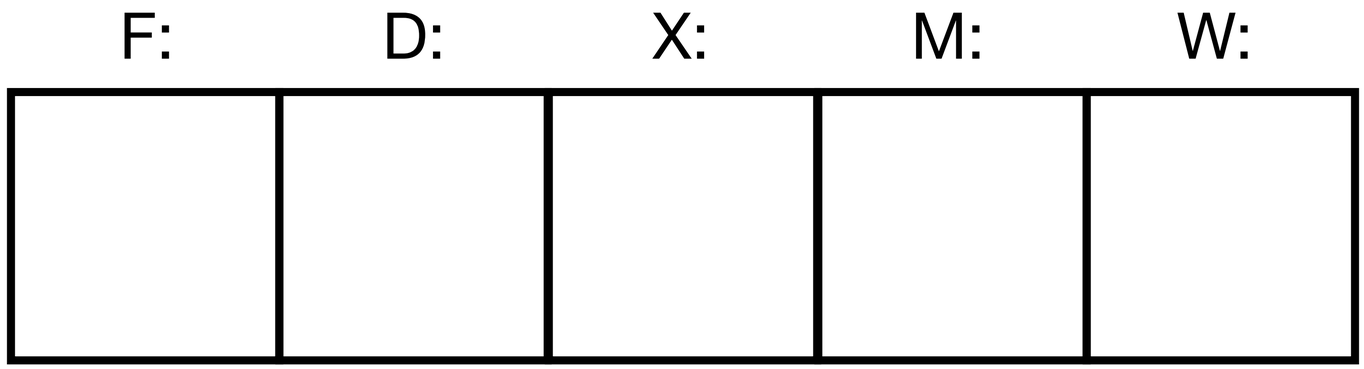
CPU, registers, ALU, etc.

DRAM (main data memory)

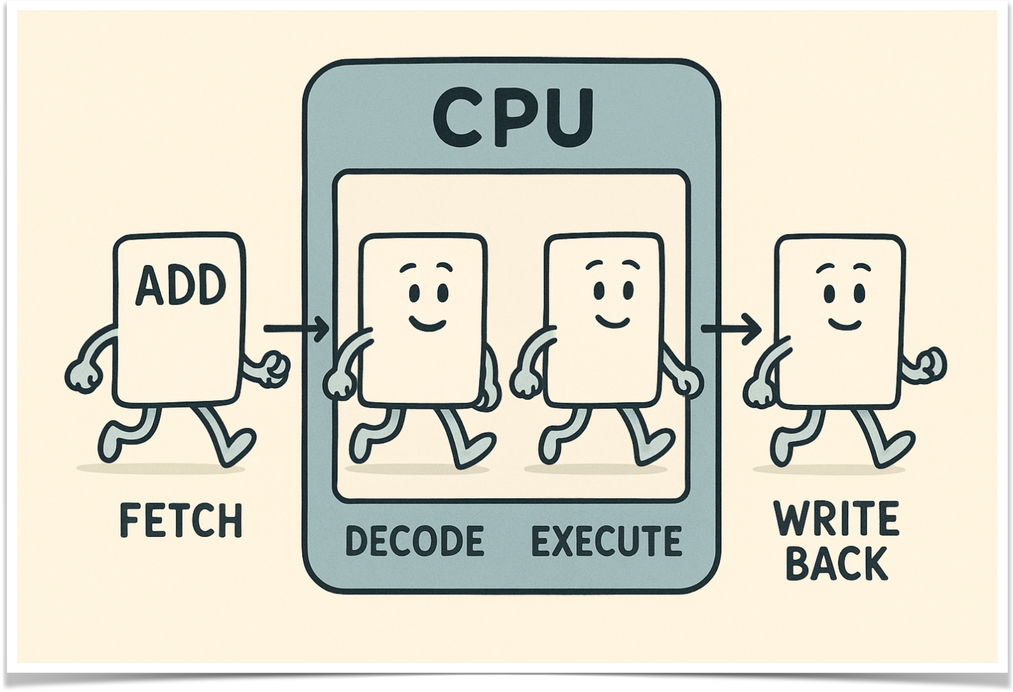
- It’s far away
- It’s big
- It’s slow

SandyBridge Motherboard, 2011 http://news.softpedia.com

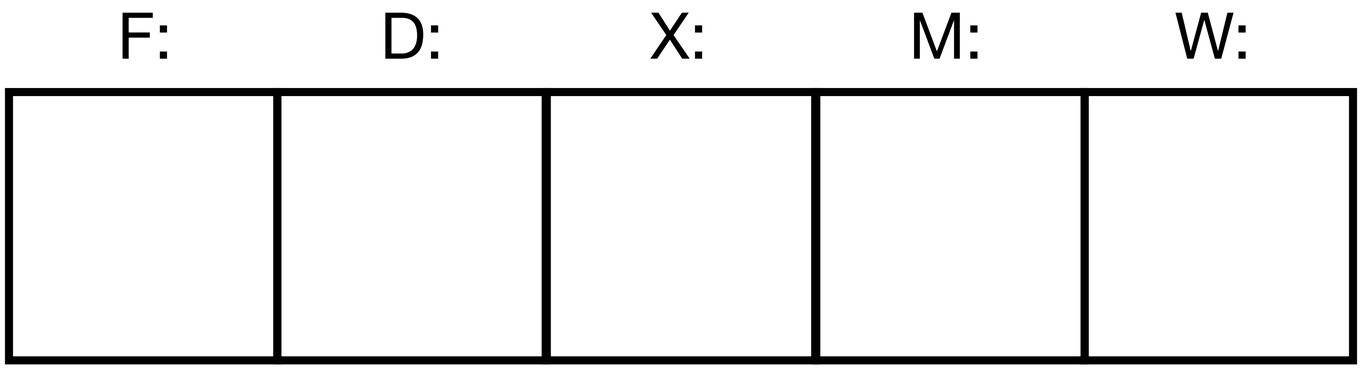
The need for speed

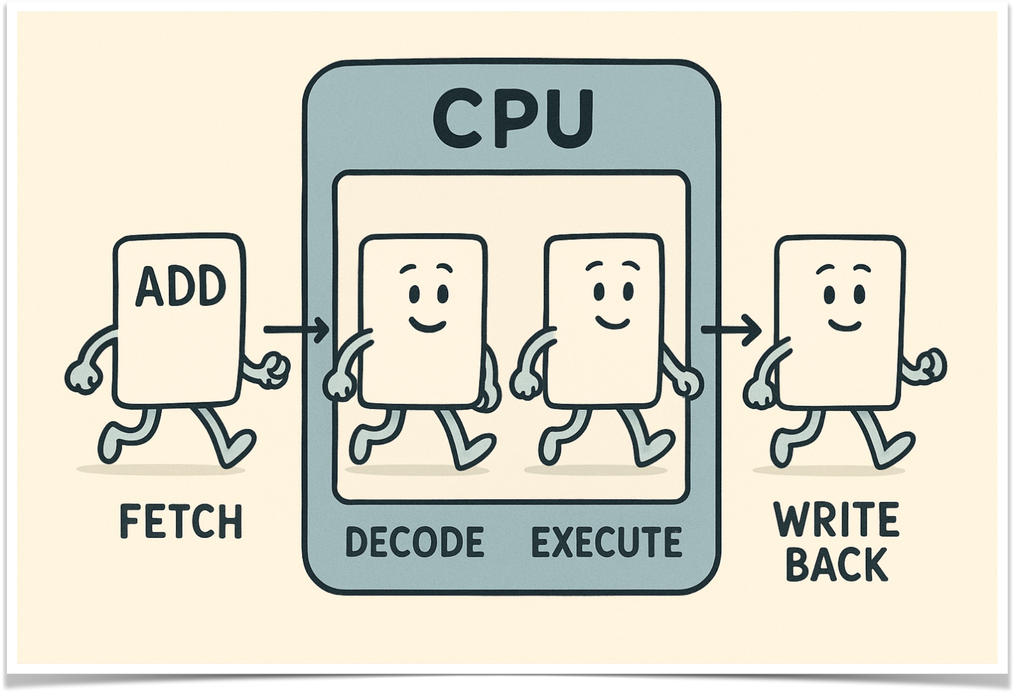


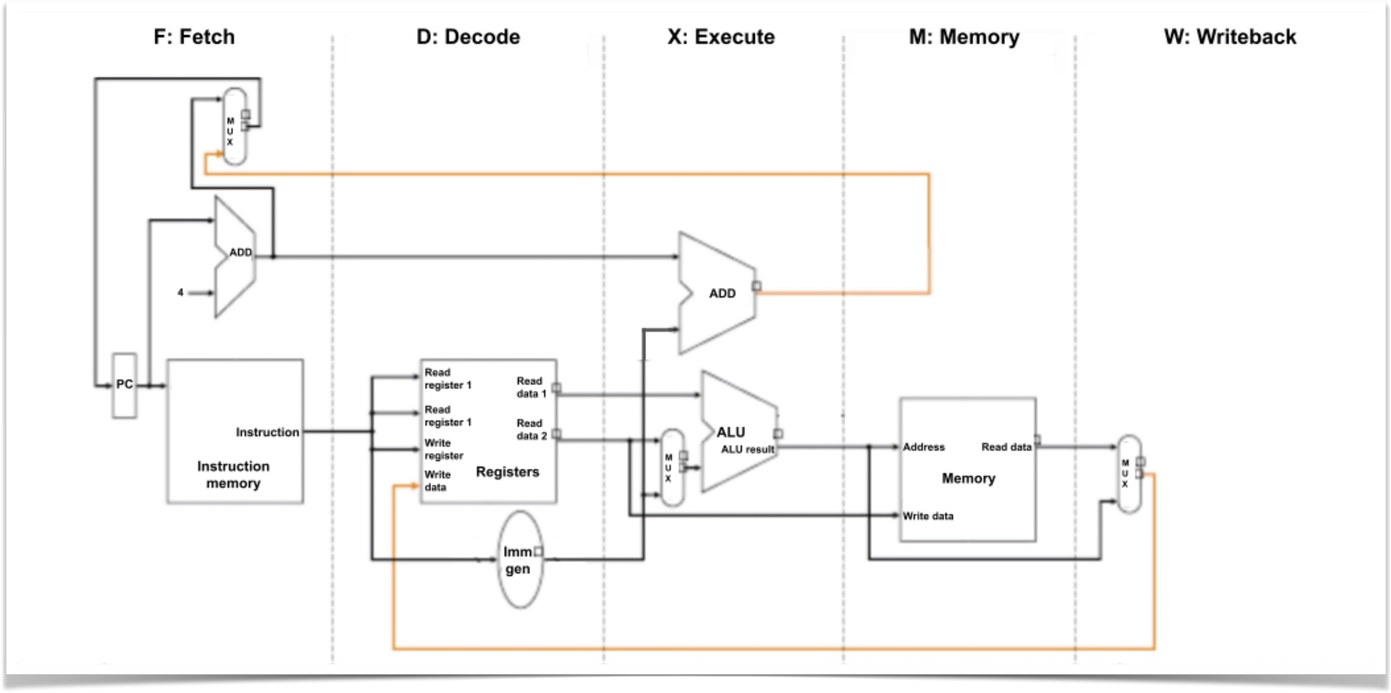
CPU stages of an instruction



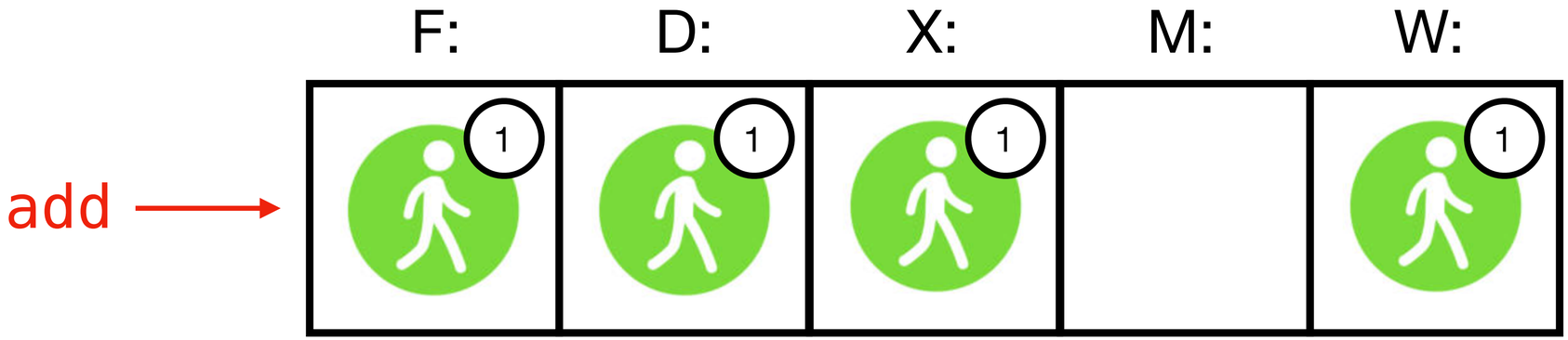
The need for speed





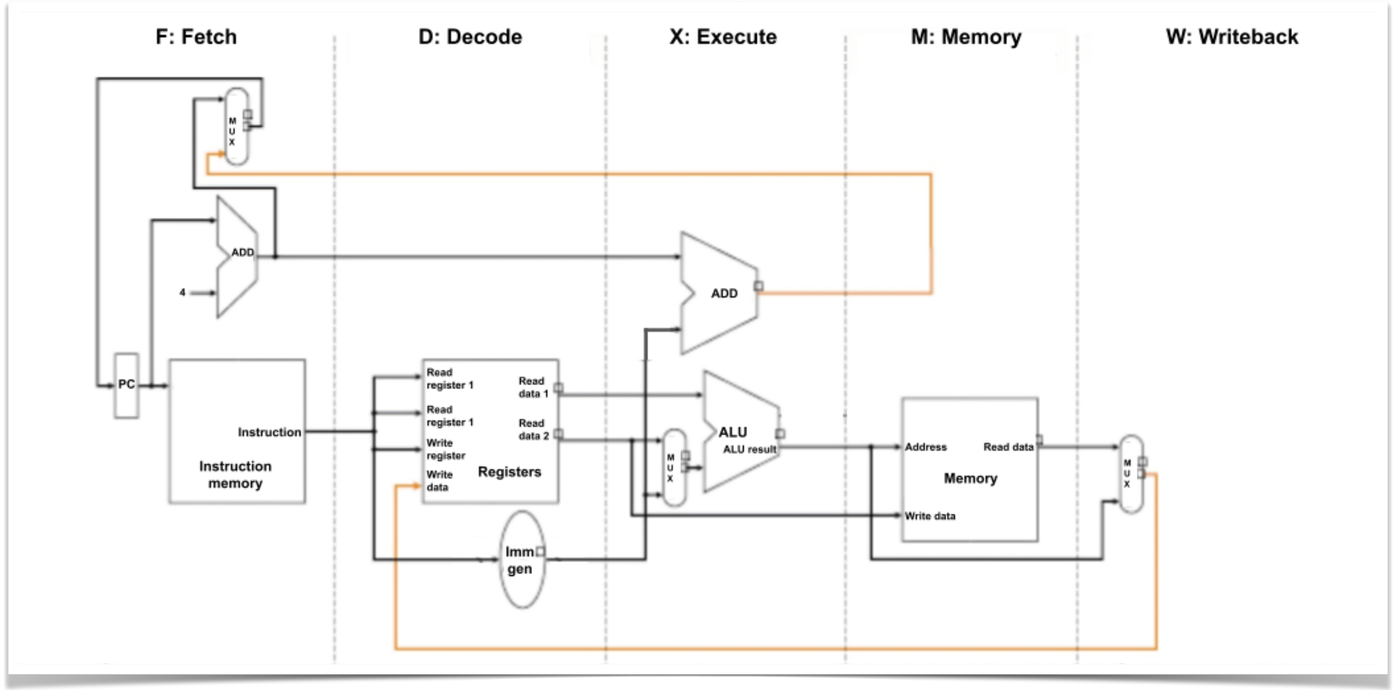


The need for speed



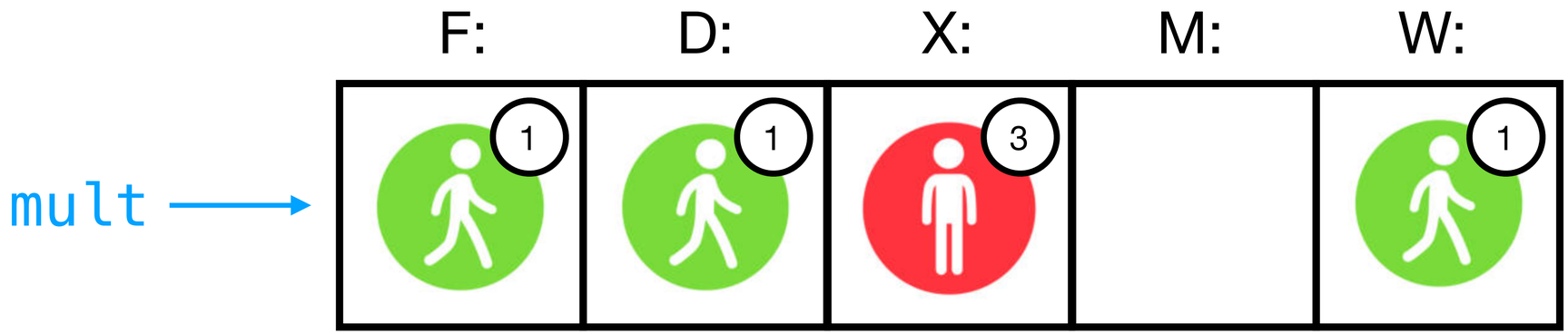
The instruction speeds:

- `add`, `sub`, `shift`: 1 cycle
- `mult`: 3 cycles
- `load`, `store`: 100 cycles[1]



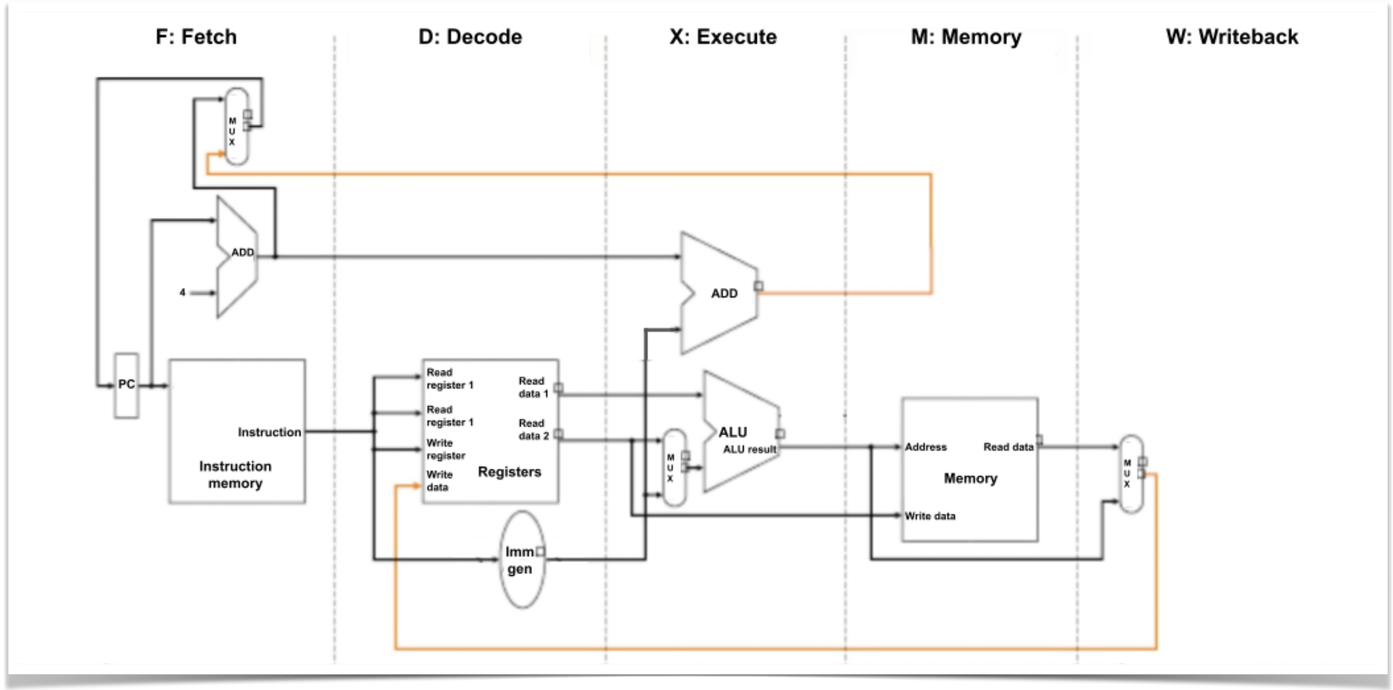
[1]To main memory (DRAM)

The need for speed



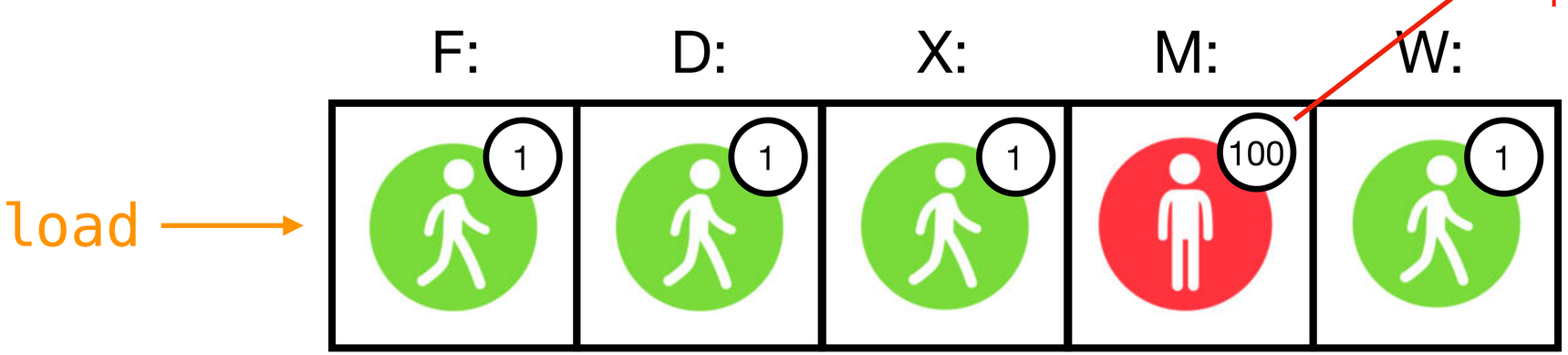
The instruction speeds:

- `add`, `sub`, `shift`: 1 cycle
- `mult`: 3 cycles
- `load`, `store`: 100 cycles[1]



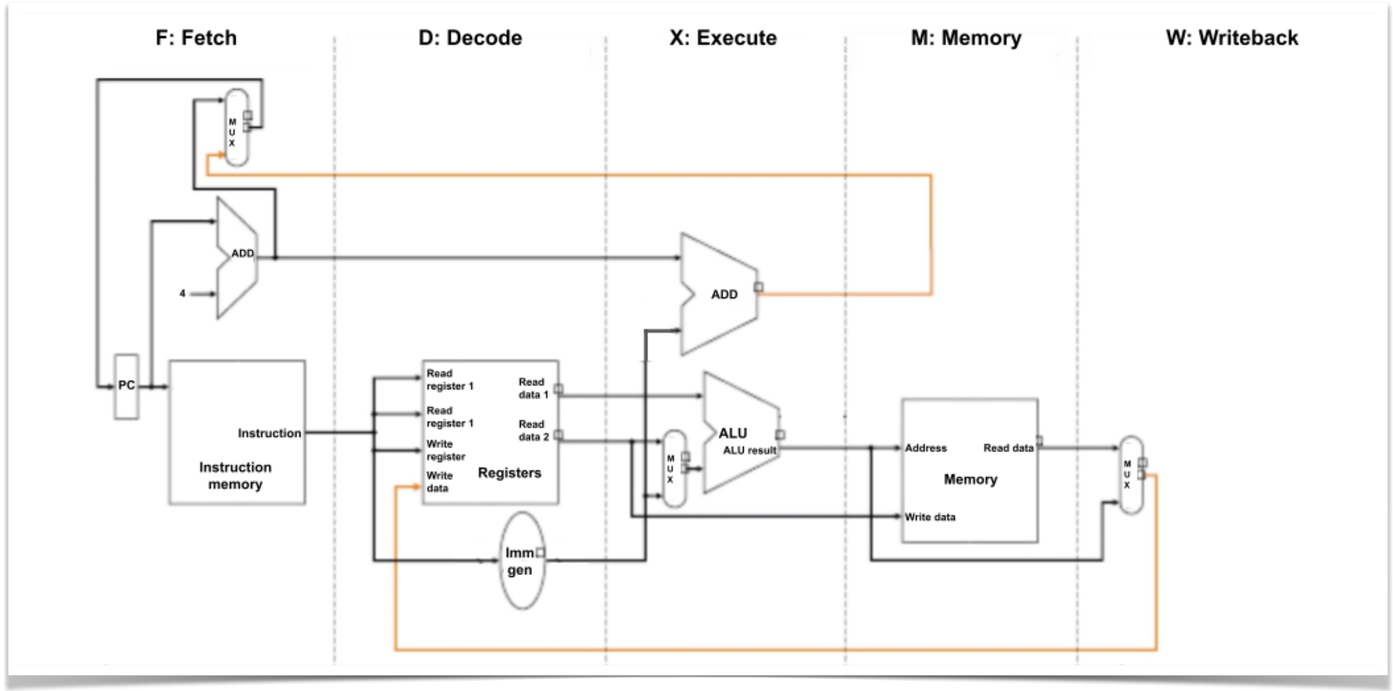
[1]To main memory (DRAM)

The need for speed



The instruction speeds:

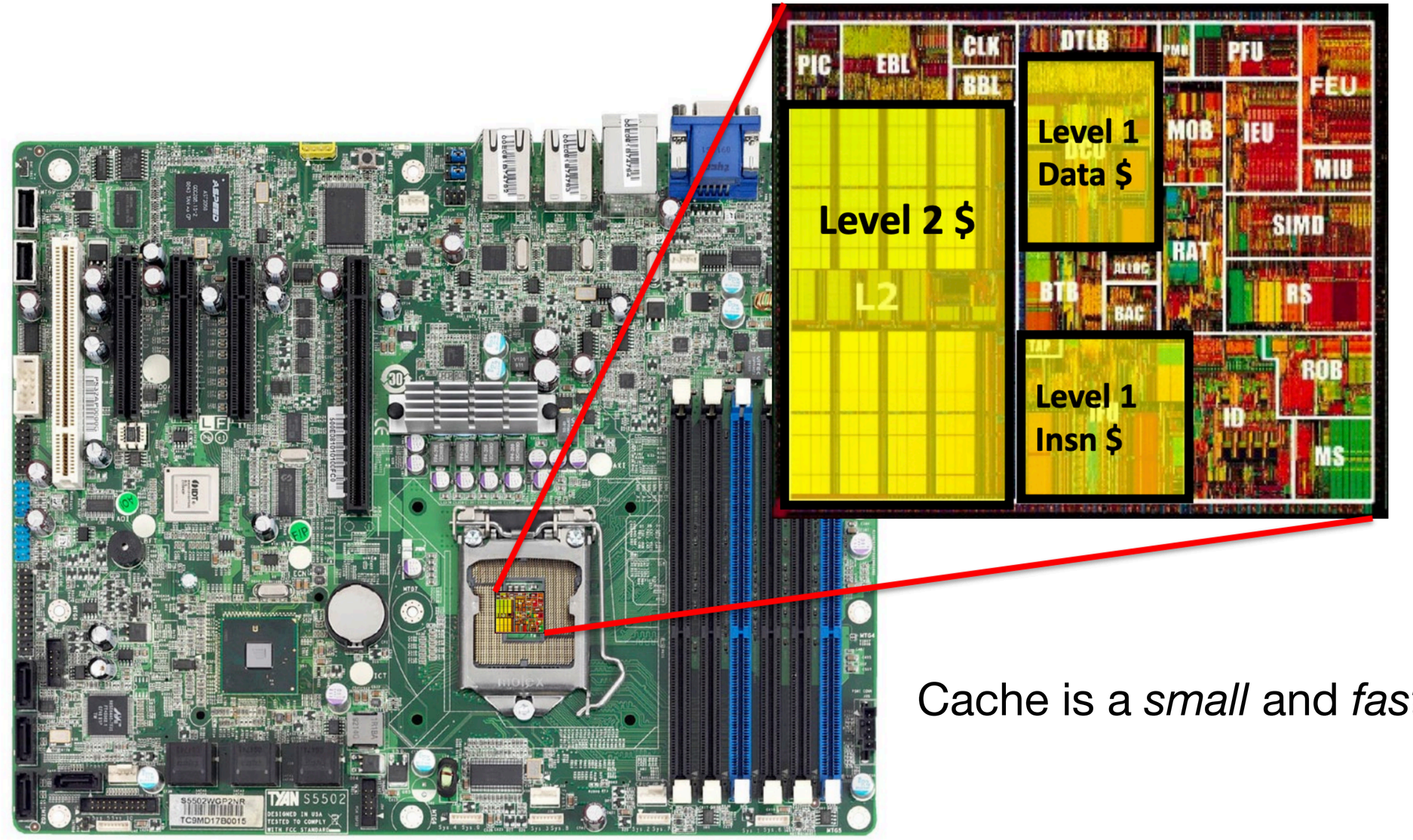
- `add`, `sub`, `shift`: 1 cycle
- `mult`: 3 cycles
- `load`, `store`: 100 cycles[1]



[1]To main memory (DRAM)

So how can I address this bottleneck?

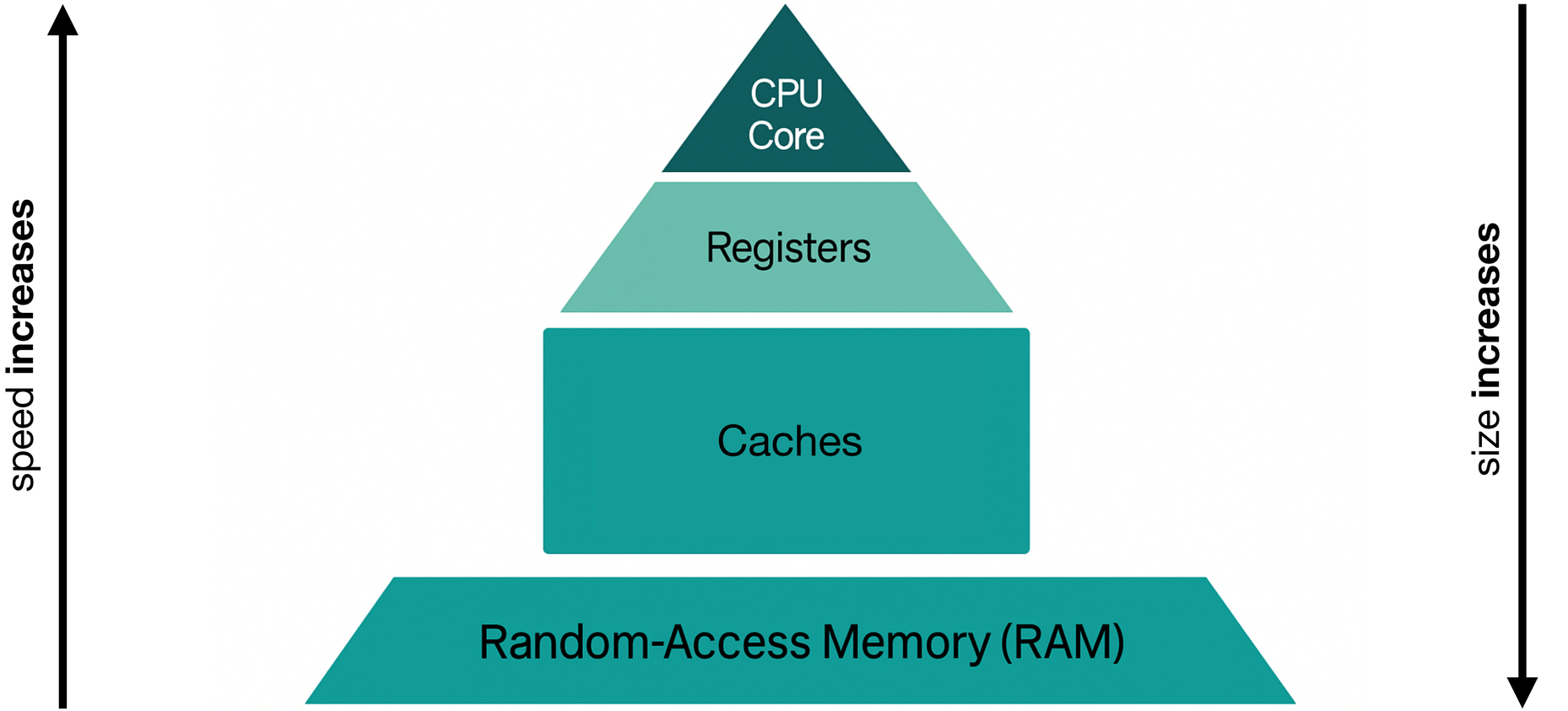
Caches!

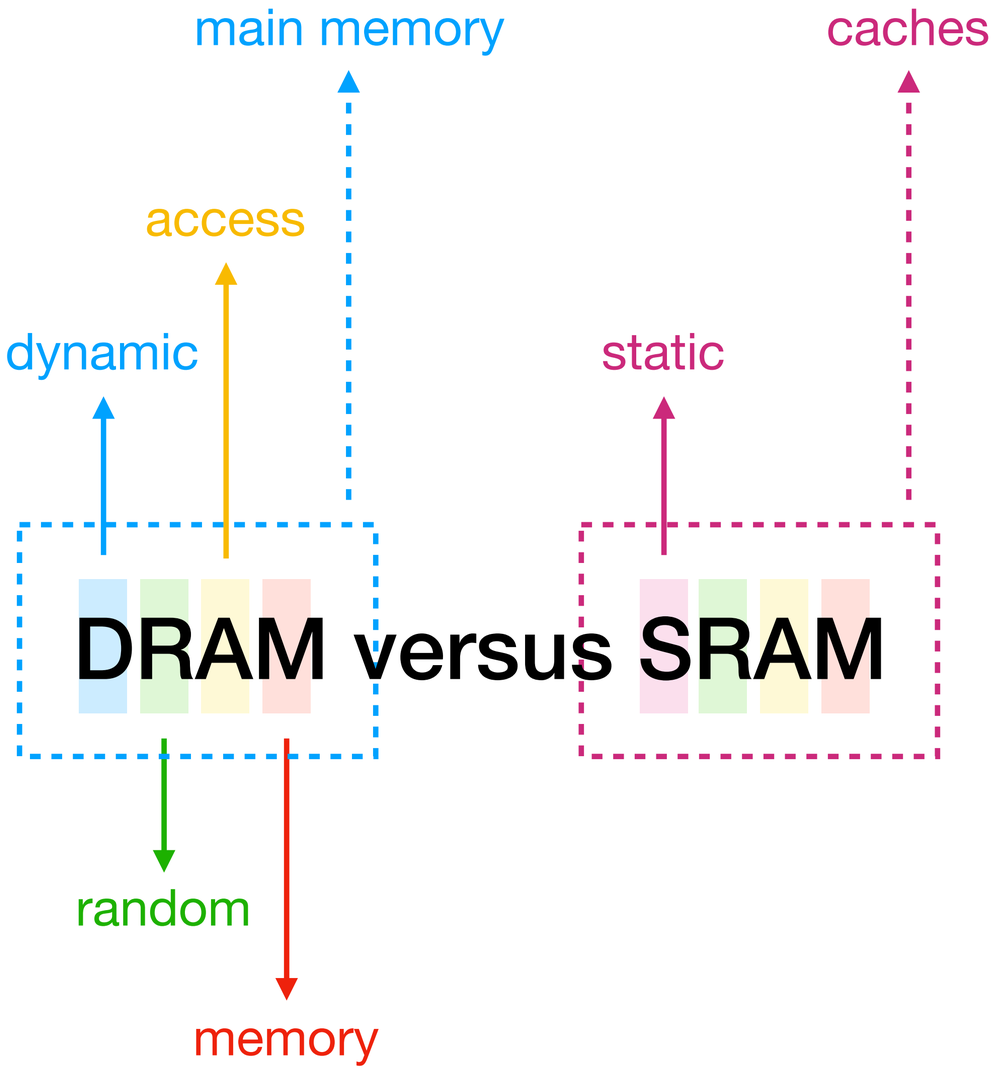


Cache is a *small* and *fast* memory **on chip**

SandyBridge Motherboard, 2011 http://news.softpedia.com

Principle of Locality and Memory Hierarchy




main memory
caches
access
dynamic
static
DRAM versus SRAM
random
memory

Main Memory is DRAM

Dynamic Random Access Memory:

- Latency to access first word: ~10ns (~30-40 processor cycles), each successive (0.5ns – 1ns)
- Each access brings 64 bits
- $3/GiB

  GiB (Gibibyte) = 1,073,741,824 bytes (2^{30}); **binary**, used by operating systems like Windows/Linux
  GB (Gigabyte) = 1,000,000,000 bytes (10^{9}); **decimal**, used by storage manufacturers

Main Memory is DRAM

Dynamic Random Access Memory:

- Latency to access first word: ~10ns (~30-40 processor cycles), each successive (0.5ns – 1ns)
- Each access brings 64 bits
- $3/GiB

  GiB (Gibibyte) = 1,073,741,824 bytes (2^{30}); **binary**, used by operating systems like Windows/Linux
  GB (Gigabyte) = 1,000,000,000 bytes (10^{9}); **decimal**, used by storage manufacturers

Data is **not** permanent**:**

- Each bit of data in DRAM is **stored in a capacitor** on a transistor
  - Capacitors leak charge over time and the stored bits fade, so the memory must be constantly refreshed → That's why it's called **dynamic**

Main Memory is DRAM

Dynamic Random Access Memory:

- Latency to access first word: ~10ns (~30-40 processor cycles), each successive (0.5ns – 1ns)
- Each access brings 64 bits
- $3/GiB

  GiB (Gibibyte) = 1,073,741,824 bytes (2^{30}); **binary**, used by operating systems like Windows/Linux
  GB (Gigabyte) = 1,000,000,000 bytes (10^{9}); **decimal**, used by storage manufacturers

Data is **not** permanent**:**

- Each bit of data in DRAM is **stored in a capacitor** on a transistor
  - Capacitors leak charge over time and the stored bits fade, so the memory must be constantly refreshed → That's why it's called **dynamic**
- It is **volatile**, meaning all data disappears when power is removed

Cache is SRAM

Static Random Access Memory:

- It’s faster (0.5 ns), more expensive, and allow lower density
- Unlike DRAM, it **stores each bit using a small latch** made of multiple transistors (typically 6), so it doesn’t need refreshing → That’s why it’s called **static**
- It is **volatile**, meaning all data disappears when power is removed

DRAM versus SRAM

Feature	DRAM (Dynamic RAM)	SRAM (Static RAM)
Data storage	Uses **1 transistor + 1 capacitor** per bit	Uses a **latch with 6 transistors** per bit
Refresh needed?	**Yes**, must be refreshed periodically	**No** refresh required
Speed	**Slower** than SRAM	**Fast**
Cost	**Cheaper** (fewer transistors)	**Expensive** (more transistors)
Density	**High density** (more bits per chip)	**Low density** (takes more space)
Power usage	**Lower idle power**, but refresh consumes energy	**Higher idle power**, but efficient during access
Volatility	**Volatile** (data lost without power)	**Volatile** (data lost without power)
Typical use	**Main memory**	**CPU caches (L1/L2/L3)**
Access time	~10–100 ns (nanoseconds)	~1–2 ns (nanoseconds)
Cost per bit	**Low**	**High**

Storage / Disk / Secondary Memory

They are attached as a **peripheral I/O device: non-volatile**

- **Solid-State Device (SSD)**
  - Time access: 40-100μs (~100k processor cycles)
  - $0.05-0.5/GB
  - Usually flash memory
- **Hard-Disk Drive (HDD)**
  - Time access: < 5-10ms (10-20M processor cycles)
  - $0.01-0.1/GB
  - Usually mechanical

Poll

You must be logged in PollEv to get credit

Why does DRAM require refreshes?

PollEv.com /gguidi

Or send **gguidi** to **22333**

Poll

You must be logged in PollEv to get credit

Which memory has higher density (more bits per chip area)?

PollEv.com /gguidi

Or send **gguidi** to **22333**

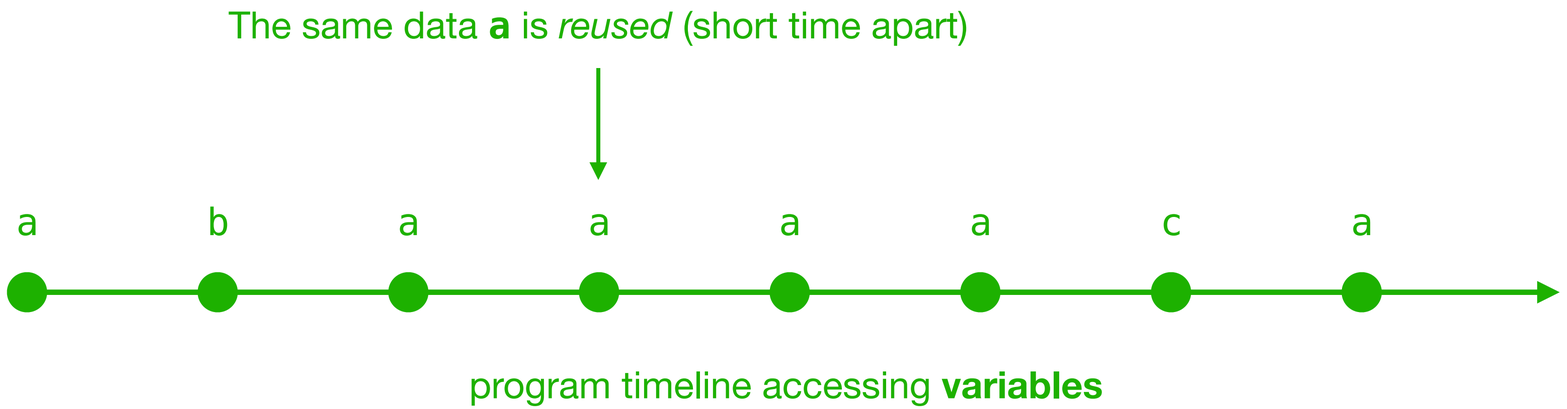
Locality, Locality, Locality

Locality Locality Locality

2 main categories!

If you ask for something, you're likely to ask for:

- The same piece of data again soon **temporal locality**

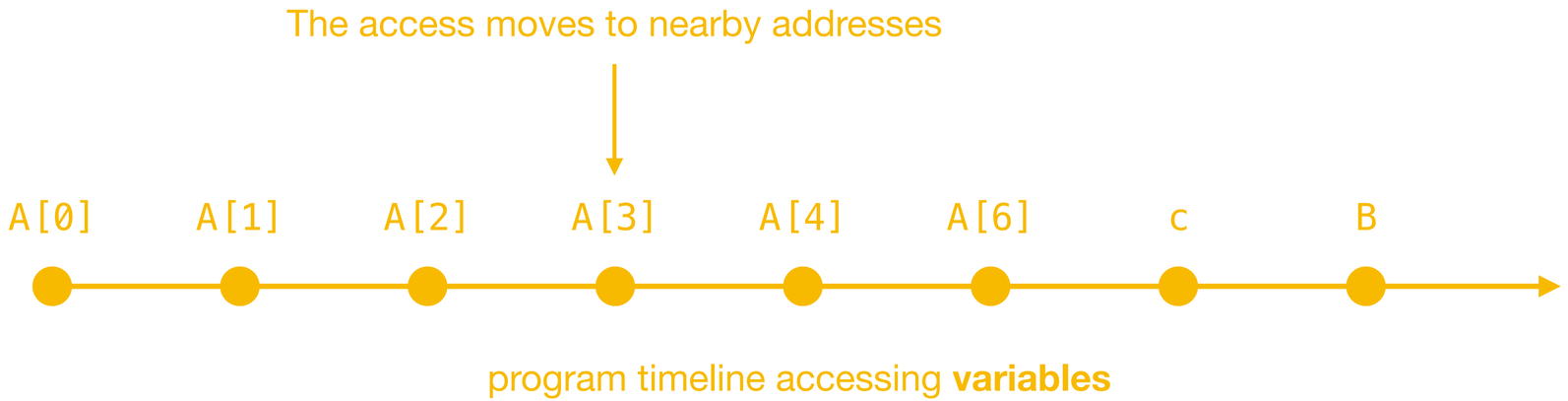


Locality Locality Locality

2 main categories!

If you ask for something, you're likely to ask for:

- Piece of data that's near the previous piece of data **spatial locality**



Poll

You must be logged in PollEv to get credit

Choose the variable that has good **spatial locality** and the one that has good **temporal locality:**

```
1 total = 0;
2 for (i = 1; i < n; i++)
{
3     n--;
4     total += a[i];
}
5 return total;
```

PollEv.com /gguidi

Or send **gguidi** to **22333**

Locality (Errata Corrige)

Choose the variable that has good **spatial locality** and the one that has good **temporal locality:**

```
1 total = 0;
2 for (i = 1; i < n; i++)
{
3     n--;
4     total += a[i];
}
5 return total;
```

Variable	Temporal locality	Spatial locality
`total`	✅ High	❌ Low
`a[i]`	❌ Low	✅ High
`n`	✅ High	❌ Low
`i`	✅ High	❌ Low

`i`, `n`, and `total` all exhibit **temporal locality** because they are accessed every iteration of the loop, while `a[i]` instead shows **spatial locality** as we move sequentially through the array

Locality

C code:

```
total = 0;
for (i = 1; i < n; i++)
{
    total += a[i];
}
return total;
```

Locality

C code:

```
total = 0;
for (i = 1; i < n; i++)
{
    total += a[i];
}
return total;
```

`total` has good temporal locality because it will be asked for again and again

The accesses to `a[i]` have good spatial locality because after asking for `a[i]` the code soon will ask for the data right after it in memory `a[i+1]`

Locality

C code:

```
int total = 0;
int temp;
int a[1000];

for (int i = 0; i < 1000; i++) {
    temp = i * 3;
    total += a[i * 10];
}
// 100 lines of code and temp is never used again
return total;
```

Variable	Temporal locality	Spatial locality
`total`	✅ High	❌ Low
`a[i * 10]`	❌ Low	❌ Low
`temp`	❌ Low	❌ Low
`i`	✅ High	❌ Low

`temp` has **bad** temporal locality because it is never reused beyond the loop, and the array has bad spatial locality because elements are accessed with a large stride instead of sequentially

Locality

C code:

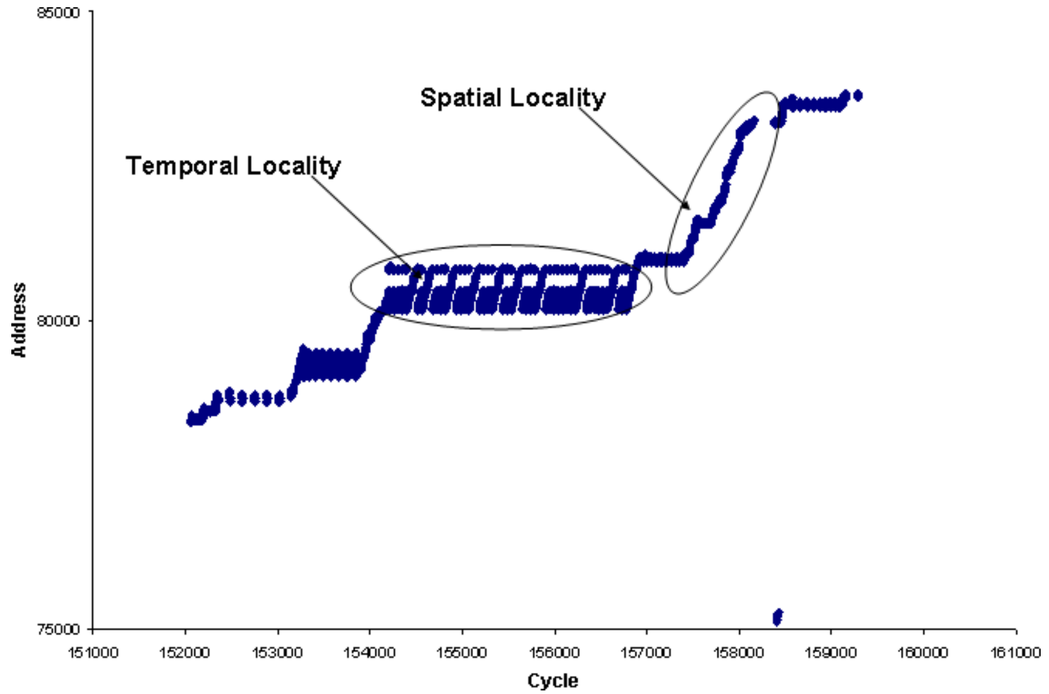
```
int total = 0;
int temp;
int a[1000];

for (int i = 0; i < 1000; i++) {
    temp = i * 3;
    total += a[i * 10];
}
// 100 lines of code and temp is never used again
return total;
```

Variable	Temporal locality	Spatial locality
`total`	✅ High	❌ Low
`a[i * 10]`	❌ Low	❌ Low
`temp`	❌ Low	❌ Low
`i`	✅ High	❌ Low

In the loop, `temp = i * 3` does have good temporal locality *as a variable access b*ut the **value** stored in `temp` has **bad** temporal locality

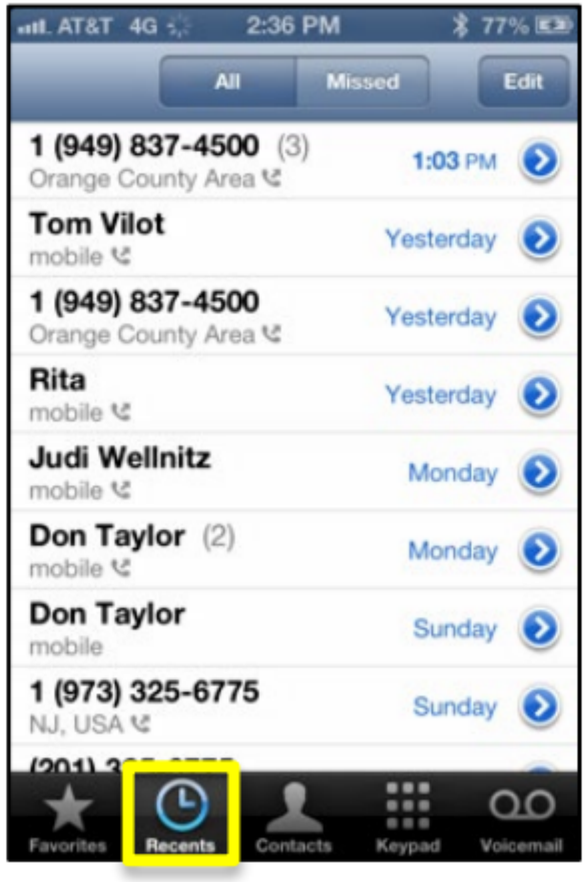
Locality

Locality is *not just about how often a variable appears*, but **about how the value is reused** over time or space **relative to the rest of the program**

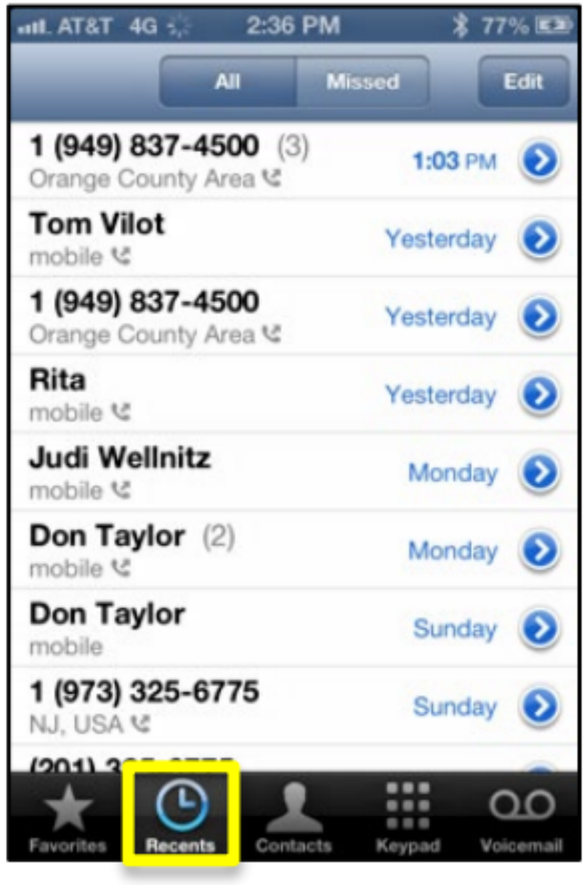


https://stackoverflow.com/questions/7638932/what-is-locality-of-reference

Your Life is Full of Locality



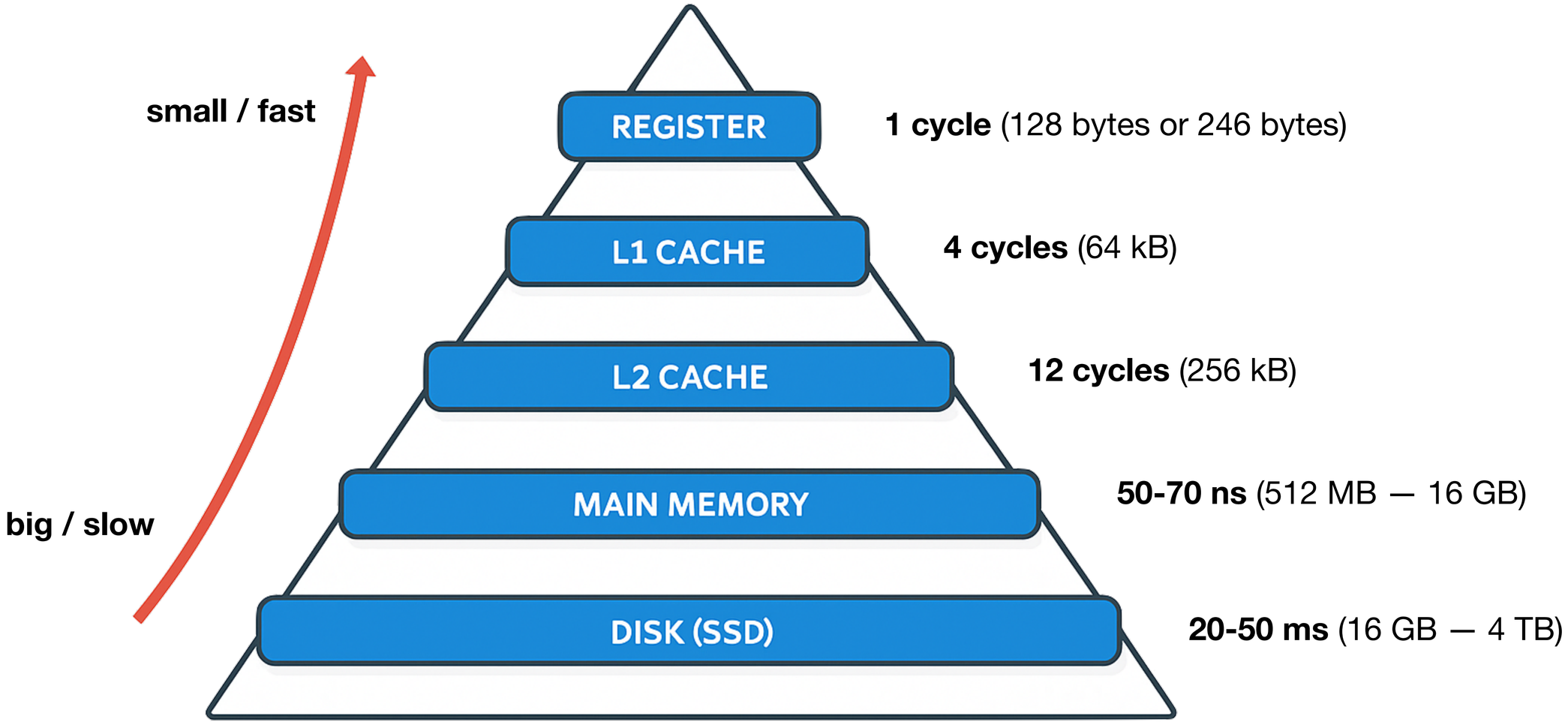
Your Life is Full of Locality



Please take **one minute** to think about an example of **locality** in everyday life and **one minute** to discuss that with your neighbor

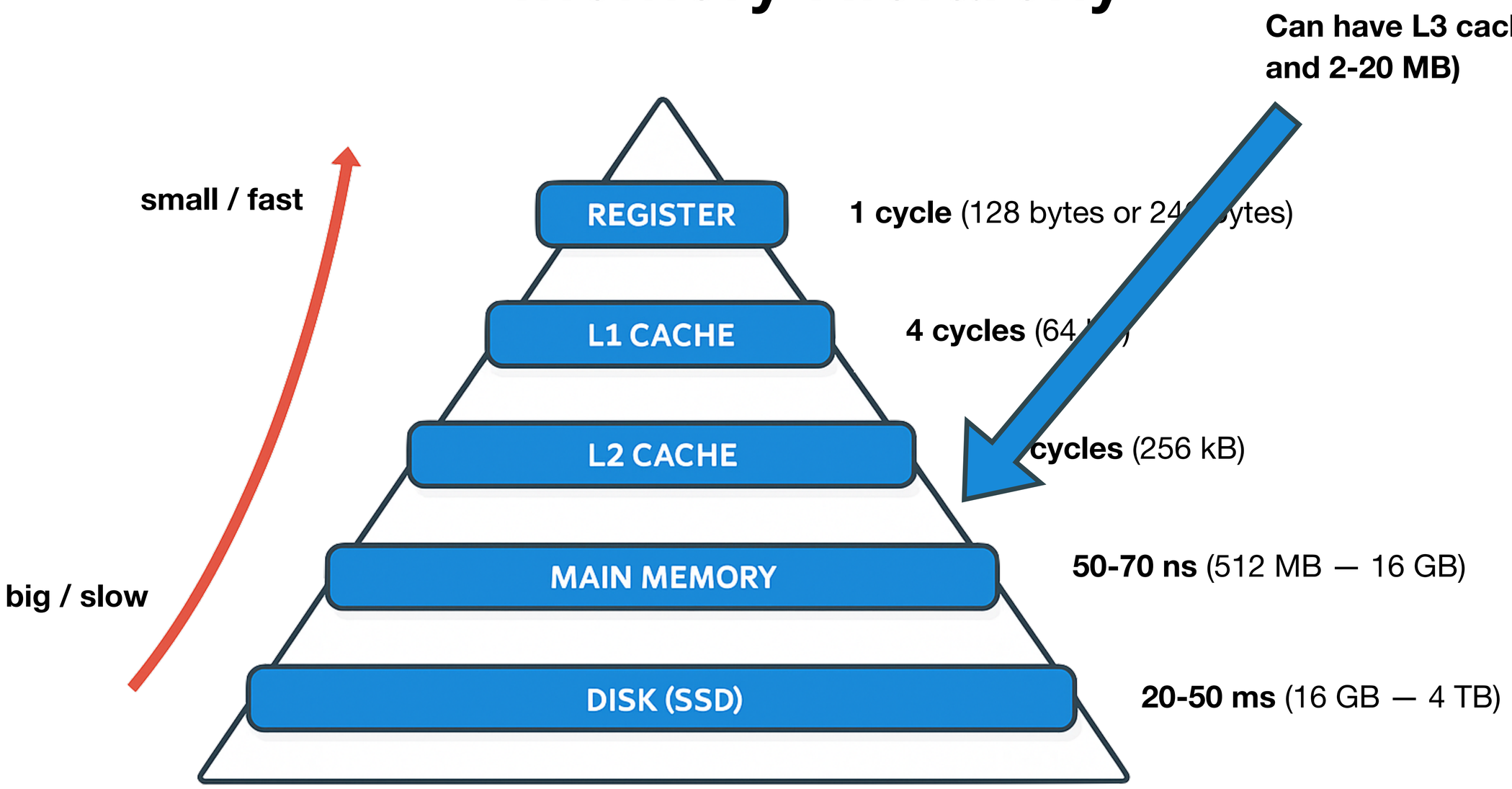
Back to the memory hierarchy

Memory Hierarchy



Intel Haswell Processor, 2013

Memory Hierarchy



Intel Haswell Processor, 2013

Terminology

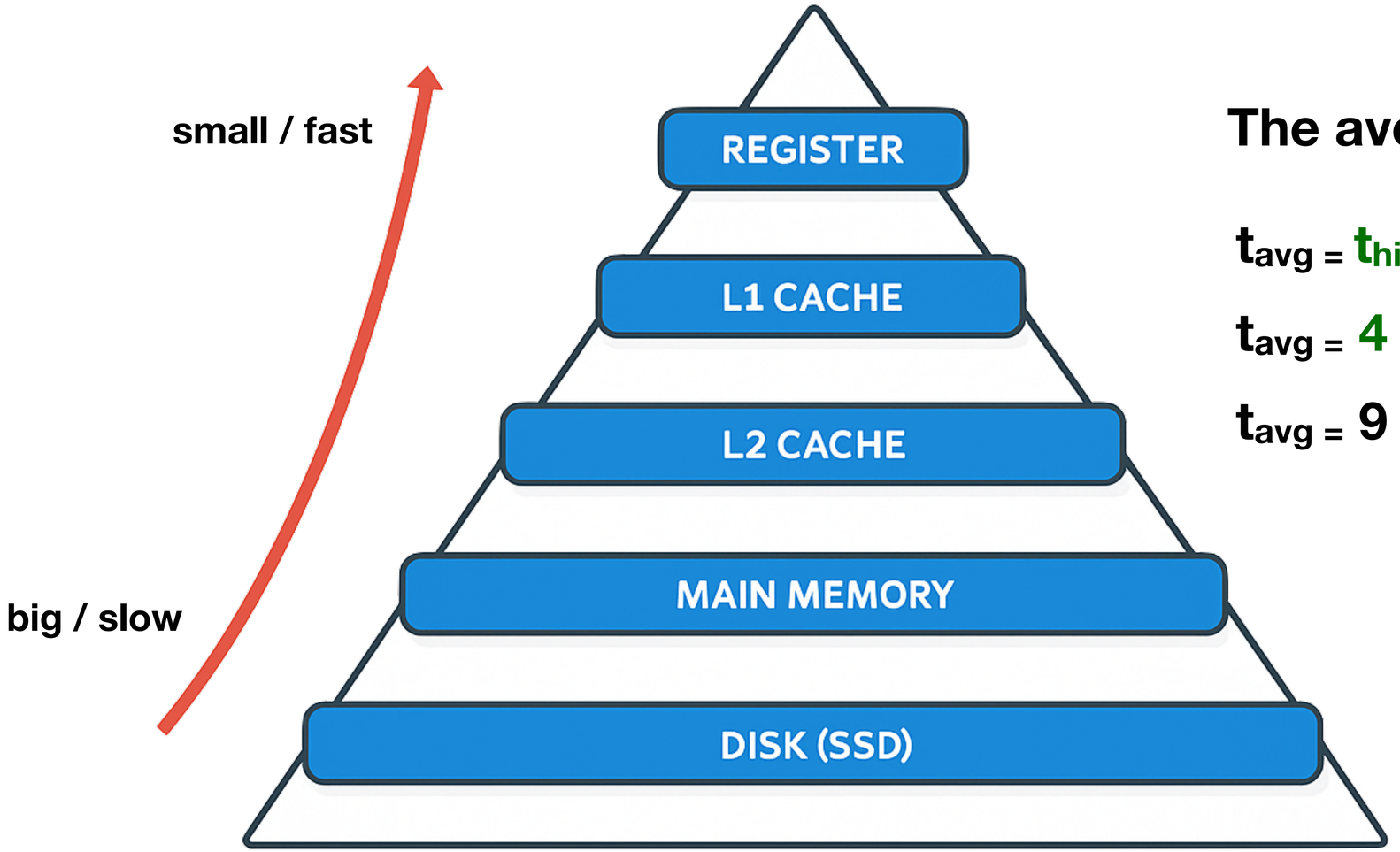
Cache hit

- Data is in the cache
- $\mathbf{t_{hit}}$ = time to access data in the cache
- **Hit Rate** (%) = # cache hits / # cache accesses

Cache miss

- Data is **not** in the cache
- $\mathbf{t_{miss}}$ = time to access and retrieve data
- **Miss Rate** (%) = # cache misses / # cache accesses

Memory Hierarchy



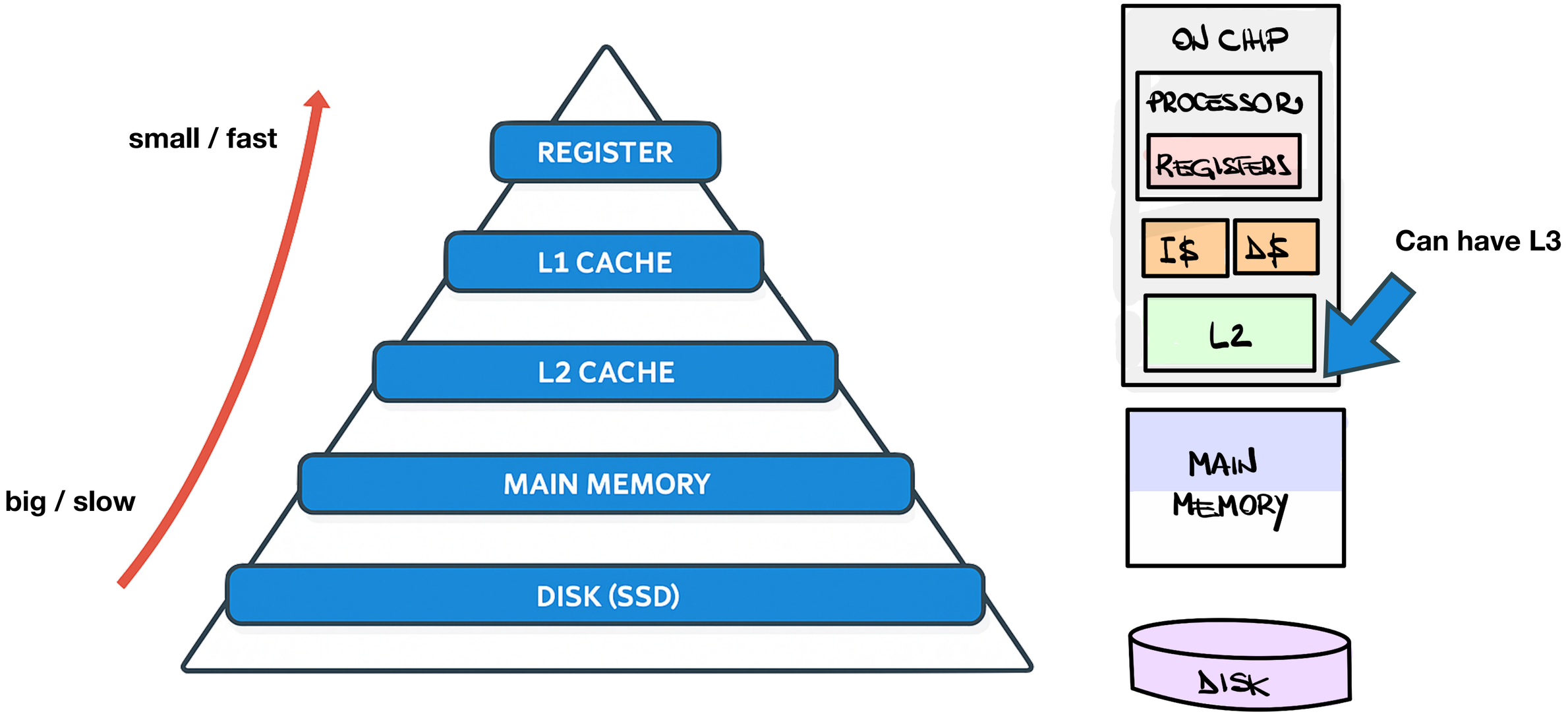
The average access time t_{avg}:

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

$t_{avg} = 4 + 5\% * 100$

$t_{avg} = 9$ cycles

Single-Core Memory Hierarchy



On to cache design—let's start with direct mapped cache

16 Byte Memory

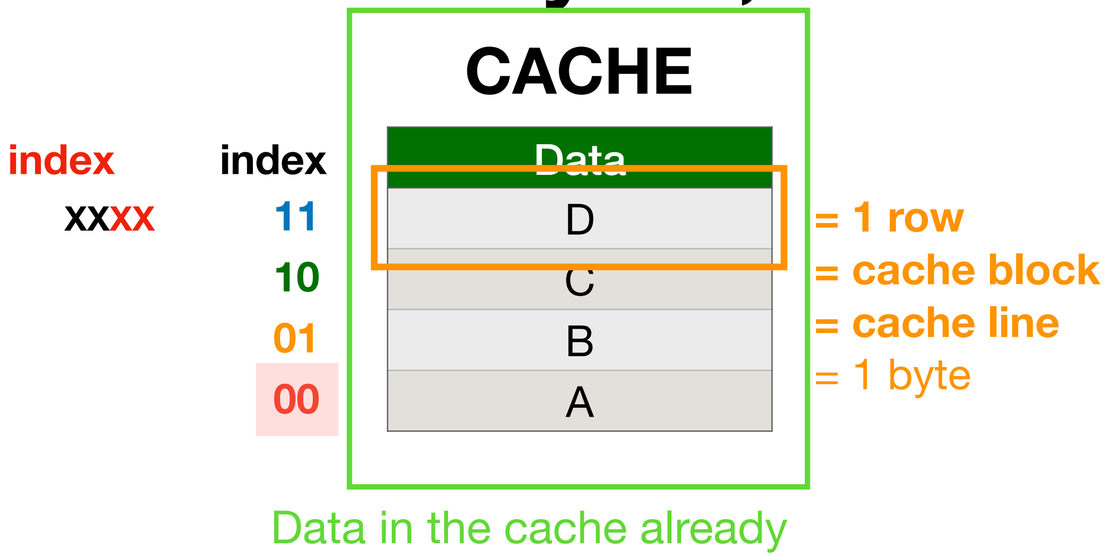
`load 1100 ⟶ r1`

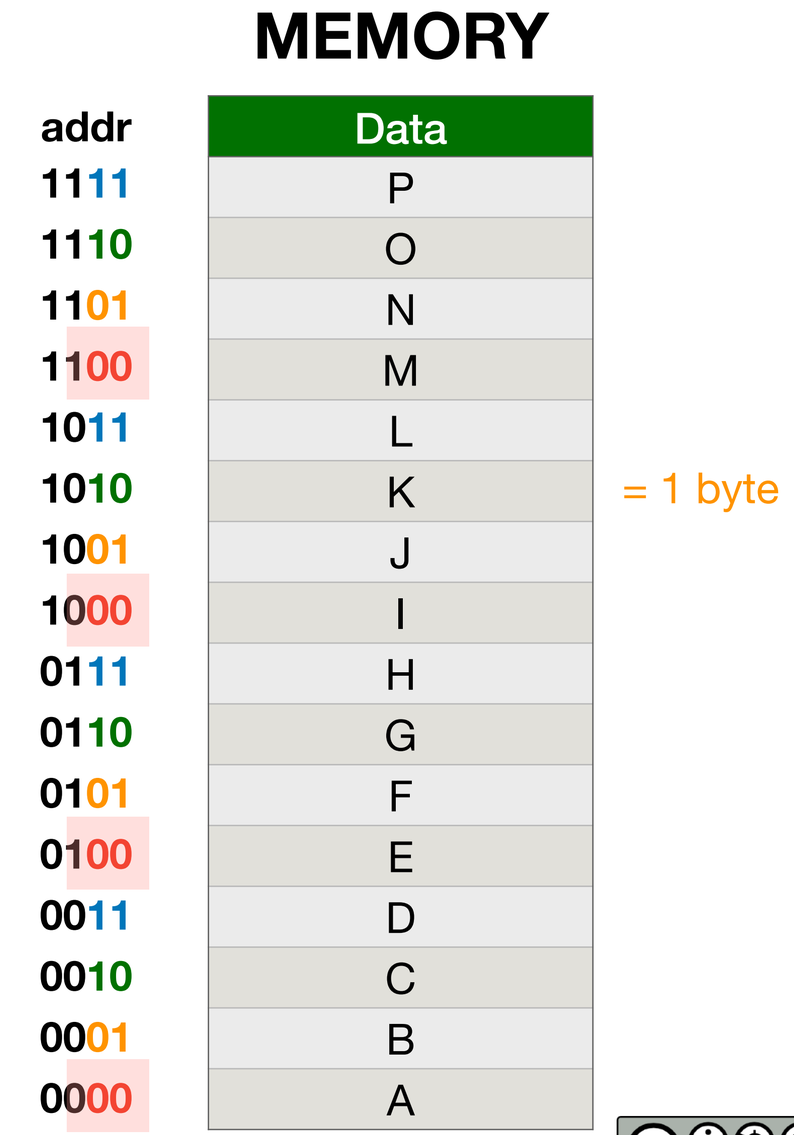
- **Byte-addressable** memory
- In this example, **4-bit address**

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

4 Bytes, Directed Mapped Cache





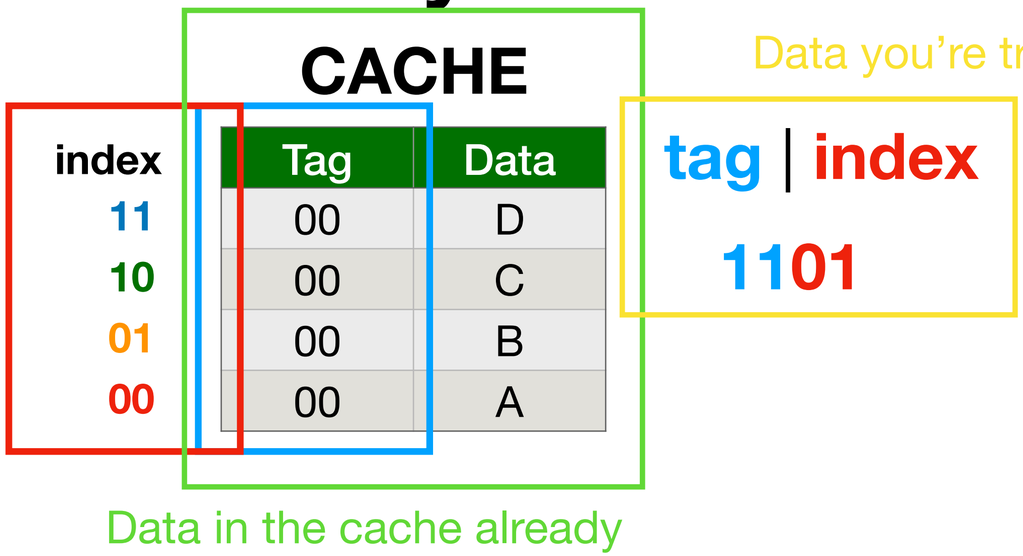
Direct mapped:

Each memory address has **exactly one possible location in the cache** where it can go—this makes lookups **very fast**

It's indexed with LSB: = least significant bit

Good **spatial** locality

4-Byte Directed Mapped Cache



Tag: minimalist label/address

```
address = tag + index
```

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

4-Byte Directed Mapped Cache

CACHE

index	V	Tag	Data
11	0	00	D
10	0	00	C
01	0	00	B
00	0	00	A

tag | index

1101

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

One last tweak: valid bit

The valid bit is a **single bit** in each cache line that indicates whether the data stored in that line is valid or not

Valid Bit = 1: The data in the cache line is **valid** and can be used

Valid Bit = 0: The data in the cache line is **invalid**, meaning it doesn't contain useful information and **cannot produce a cache hit**

the cache line is essentially considered **empty**

The access algorithm

CACHE

index	V	Tag	Data
11	0	00	D
10	0	00	C
01	0	00	B
00	0	00	A

Data in the cache already

Data you're trying to load in the register

tag | index

1101

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

1. Split the address between **tag** `t` and **index** `i`
2. Check the entry `i`
3. Is it valid? If no, cache miss!
4. If the bit is valid, then is it the tag `t`? If no, cache miss!
5. Otherwise, cache hit!

The access algorithm

CACHE

index	V	Tag	Data
11	0	00	D
10	0	00	C
01	0	00	B
00	0	00	A

Data in the cache already

Data you're trying to load in the register

tag | index

1101

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

On cache miss, **fill** the cache:

1. Get data **d** from the main memory
2. Is entry valid? If so, **evict**
3. Then, set valid bit = 1, tag = **t**, data = **d** in entry **i**