



# Core C++ 2025

19 Oct. 2025 :: Tel-Aviv

# Order! Order!

Shar-yashuv Giat  
Ofek Shochat

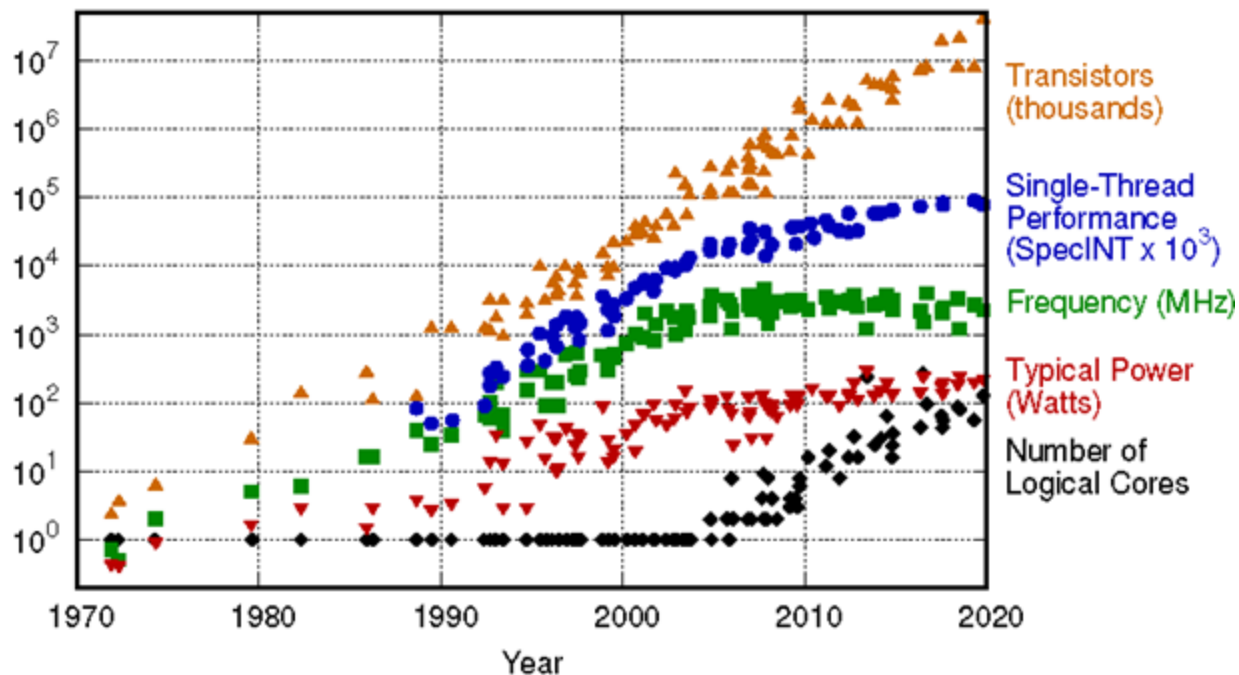
#whoami



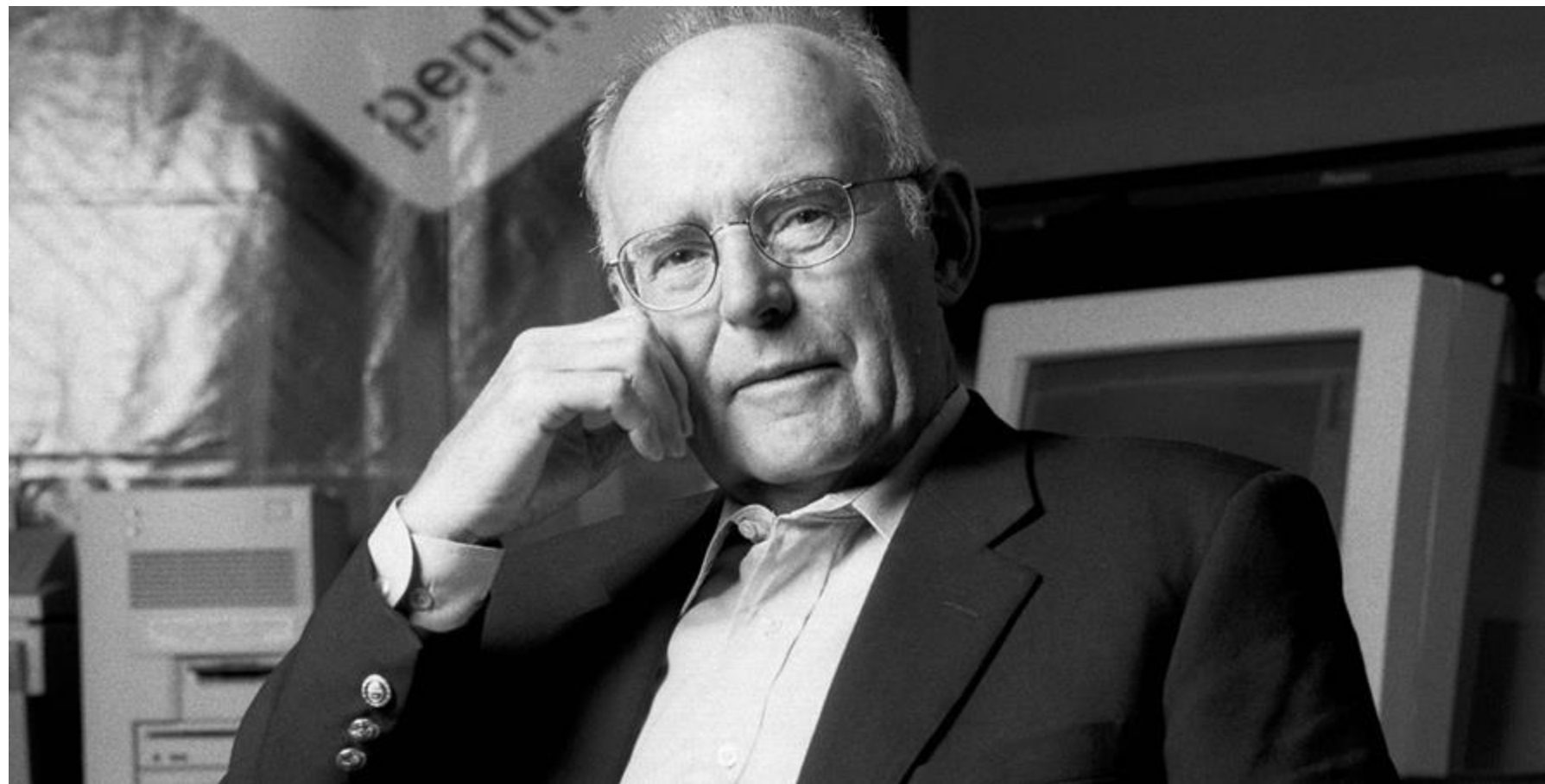
#whoami



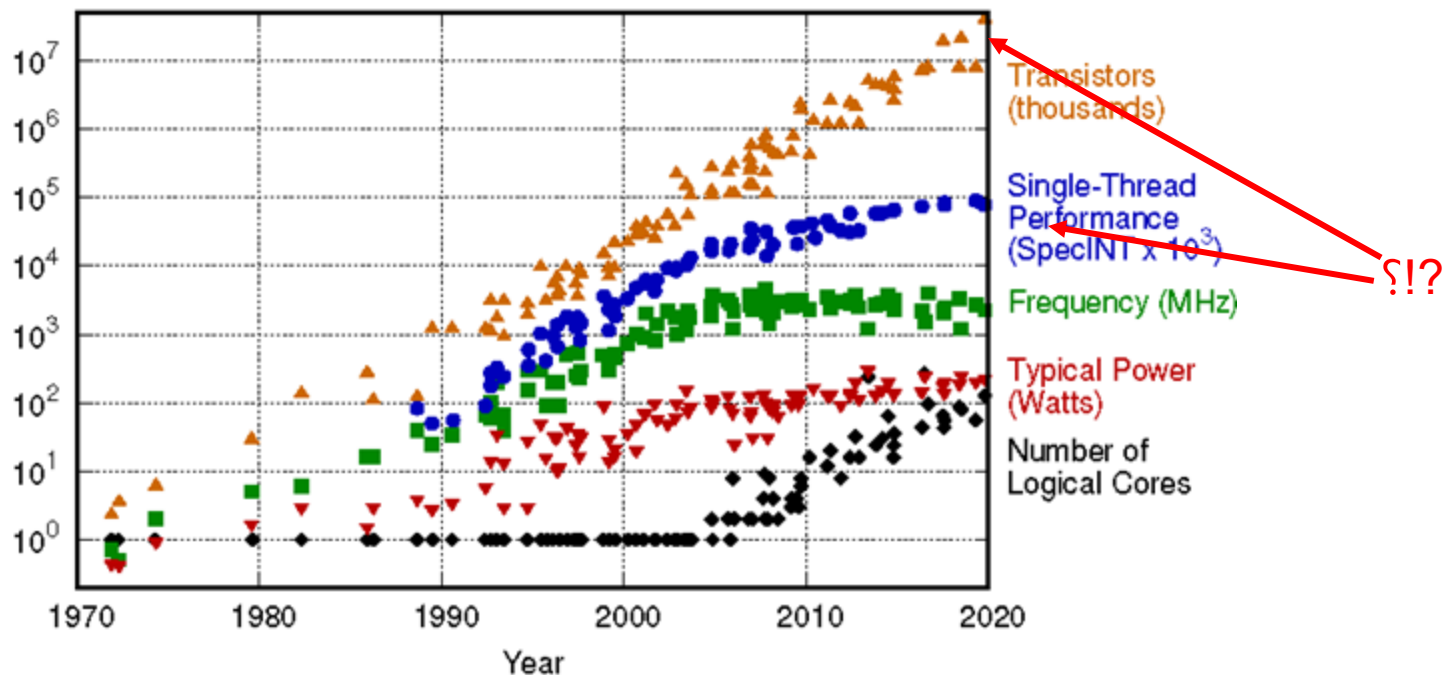
## 48 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2019 by K. Rupp

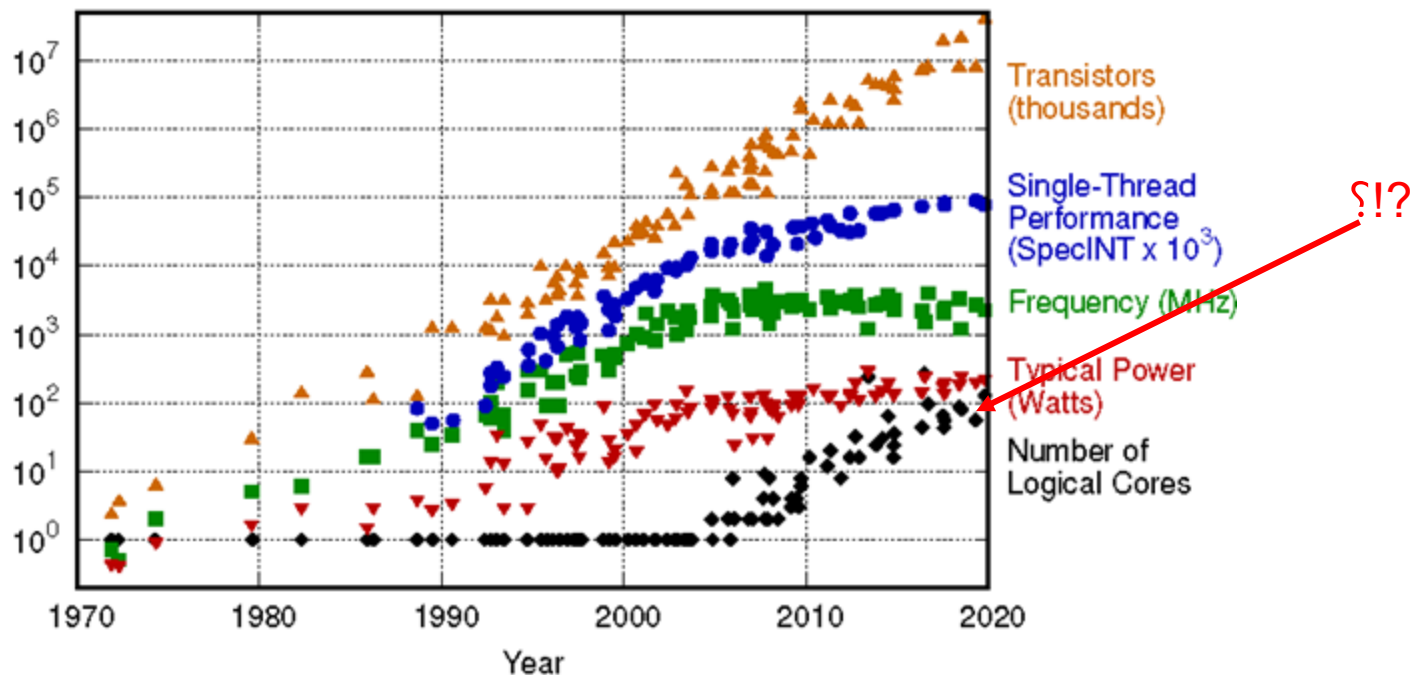


## 48 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2019 by K. Rupp

48 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2019 by K. Rupp

# Atomics very briefly

Atomics are shared variables which all threads can modify safely.



# Atomics very briefly

Atomics are shared variables which all threads can modify safely.

load                    `T std::atomic::load();`

store                   `void std::atomic::store(T desired);`

(rmw:)

exchange               `T std::atomic::exchange(T desired);`

cmpxchg                `bool std::atomic::compare_exchange_*(T& expected, T  
desired);`

fetch\_and\_\*   `T std::atomic::fetch_add(T arg);`

# What's in it for us?

Often have better scalability for simple operations.

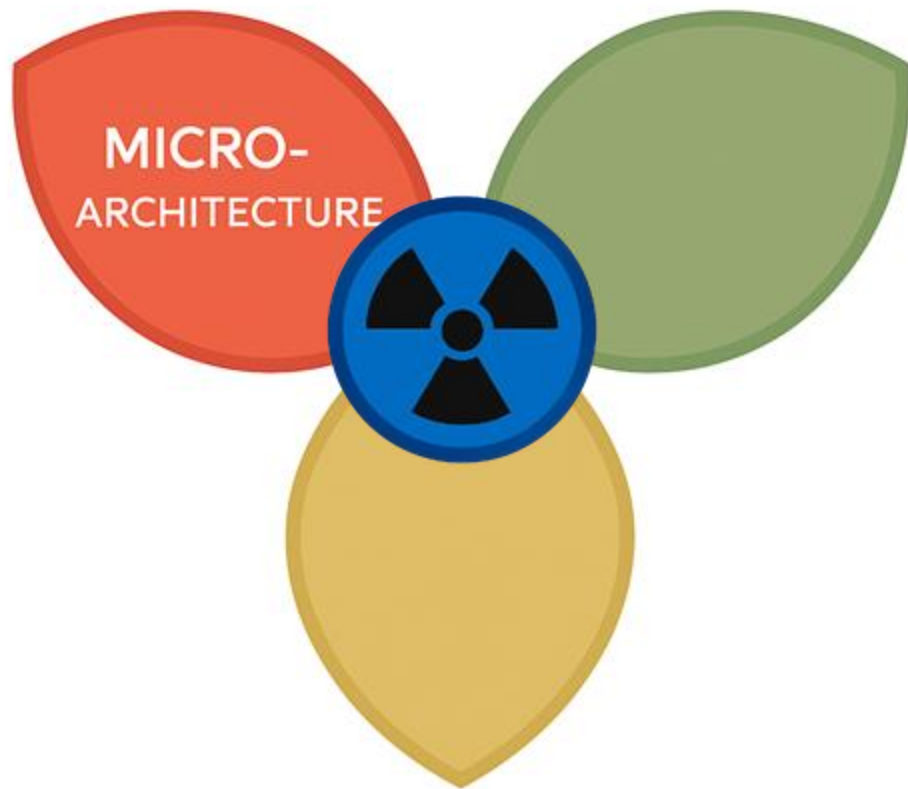
```
Benchmark 1: locked_ring_buffer_spisc.exe
Time (mean ± σ):      5.078 s ± 1.027 s    [User: 8.799 s, System: 0.113 s]
Range (min ... max):  4.058 s ... 7.700 s    10 runs

Benchmark 2: lockless_ring_buffer_spisc.exe
Time (mean ± σ):      2.354 s ± 0.185 s    [User: 4.105 s, System: 0.077 s]
Range (min ... max):  1.930 s ... 2.495 s    10 runs

Summary
lockless_ring_buffer_spisc.exe ran
2.16 ± 0.47 times faster than locked_ring_buffer_spisc.exe
```

(win11 i7-12700h 32GB 3200mhz ram)

I'm all ears...



# Our Baby Memory Model

1. Loads and stores are totally ordered.

# Our Baby Memory Model

1. Loads and stores are totally ordered. Well, **everything** is in program order.

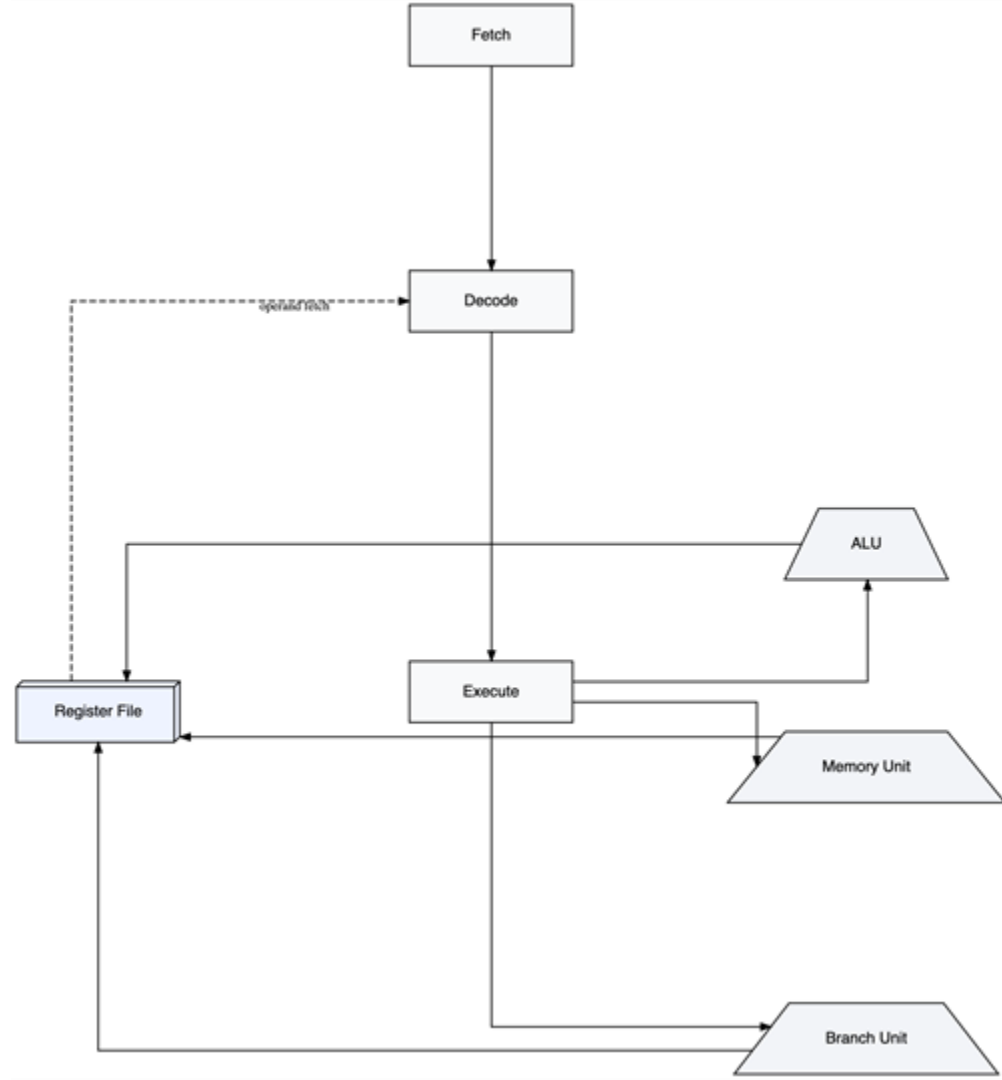
# Our Baby Memory Model

1. Loads and stores are totally ordered. Well, **everything** is in program order.
2. Nothing!

# But hey it's slow

When ordering is enforced:

- Some units will just stall
- Memory will be practically locked

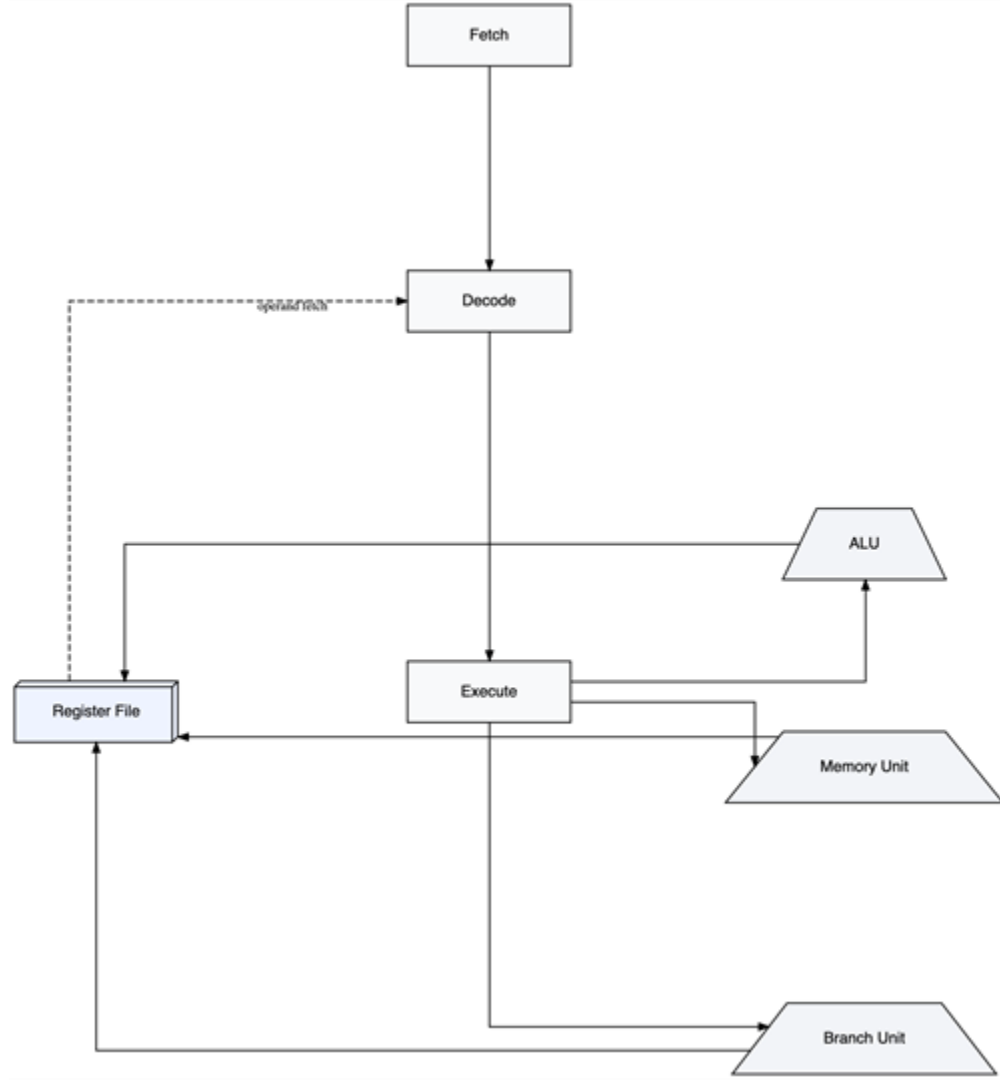




# But hey it's slow

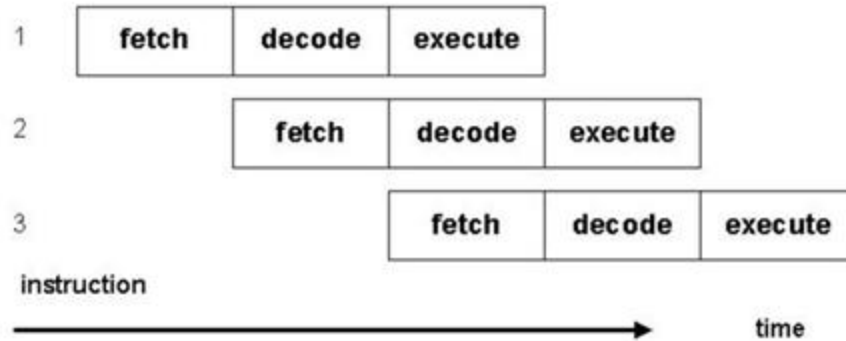
When ordering is enforced:

- **some compute units will just stall**
- Memory will be practically locked



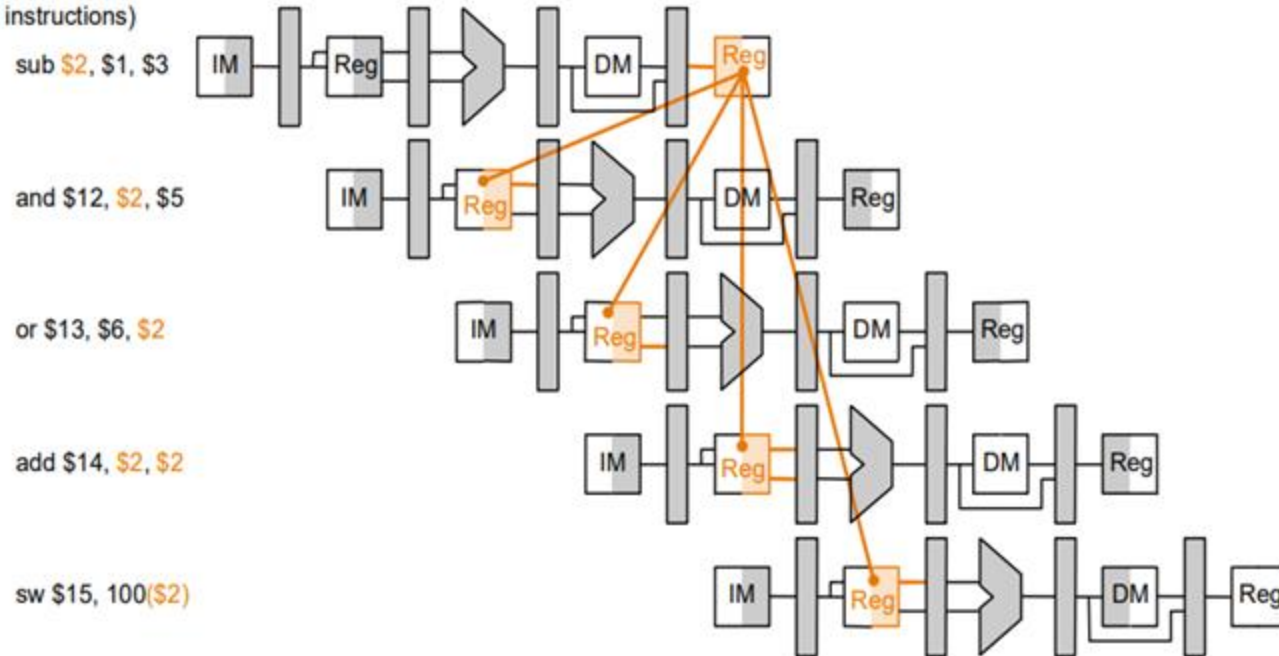
# Pipelining

1. *Fetch*
2. *Decode*
3. *Execute*

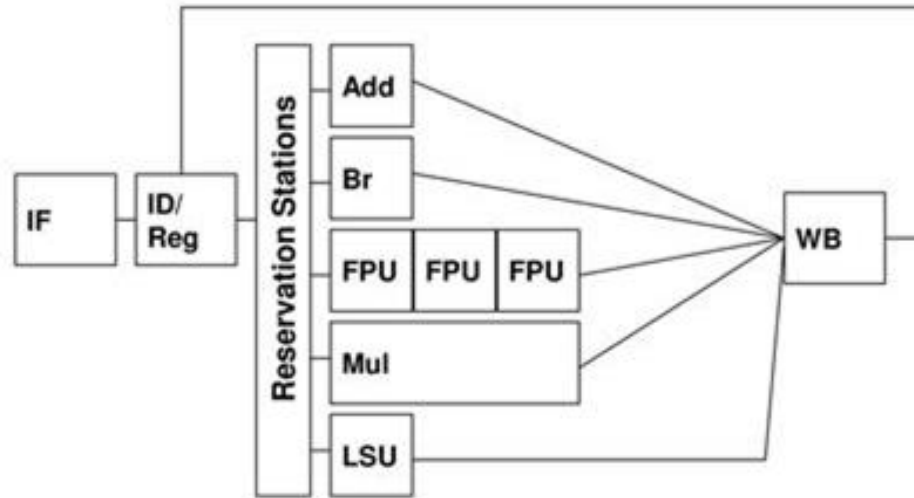


# OoOE

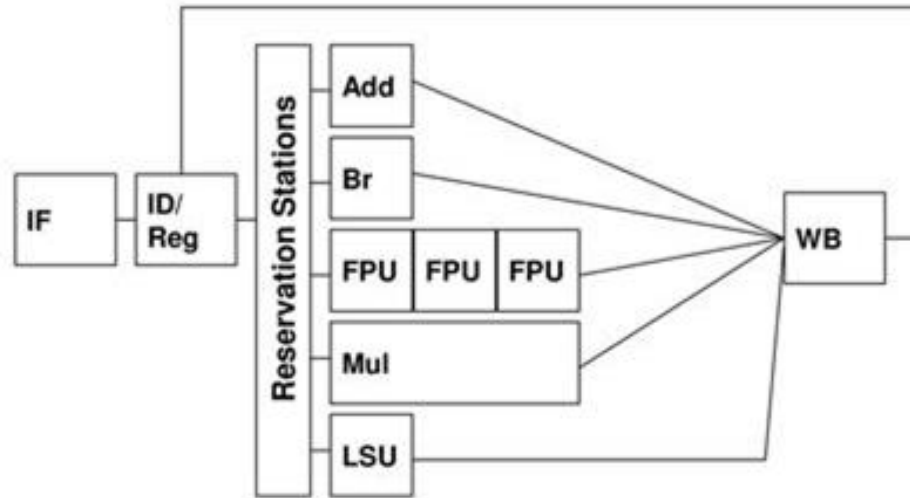
Program  
execution  
order  
(in instructions)



# OoOE



# OoOE



“Modern Intel and AMD architectures use variable depth pipelines ranging between 14 and 19 stages on 22-32 nm lithographic processes.”  
(2014... I was still munching crayons!)

# What does it mean really?

We do want for stuff to look like they're ordered. (We are used to think sequentially!)

# What does it mean really?

We do want for stuff to look like they're ordered. (We are used to think sequentially!)

The CPU promises us a few things:

Stores and loads to the same locations will not be reordered.


Earlier loads will not reorder with newer stores.

# What does it mean really?

...

Stores and loads to the same locations will not be reordered.

This allows Store-Load reordering if they are not to the same location:



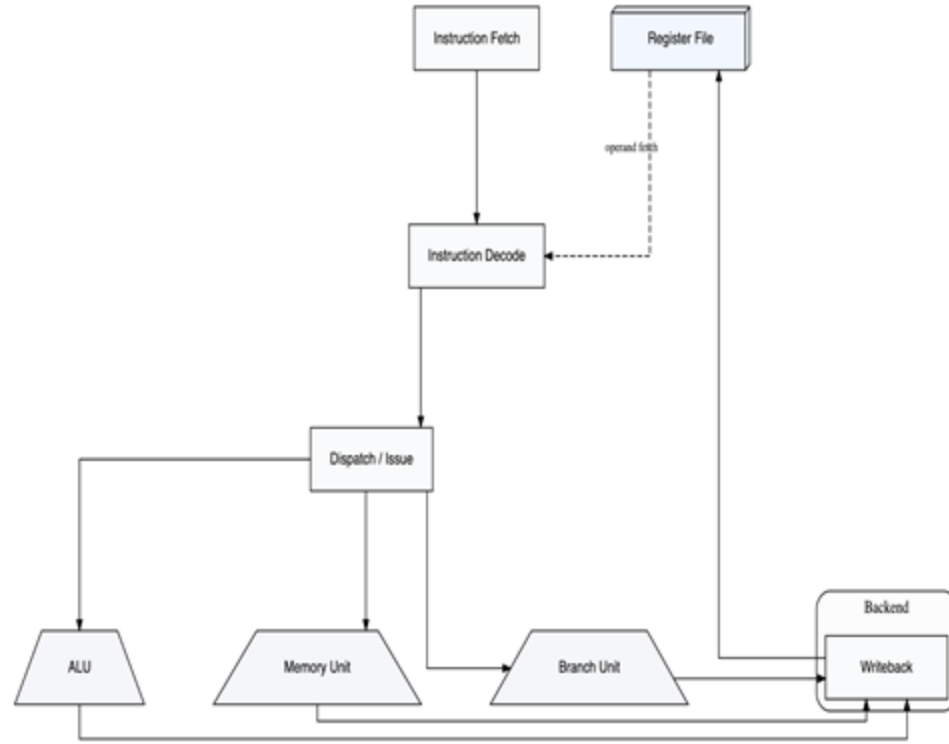
store [y] = 1  
load [x]



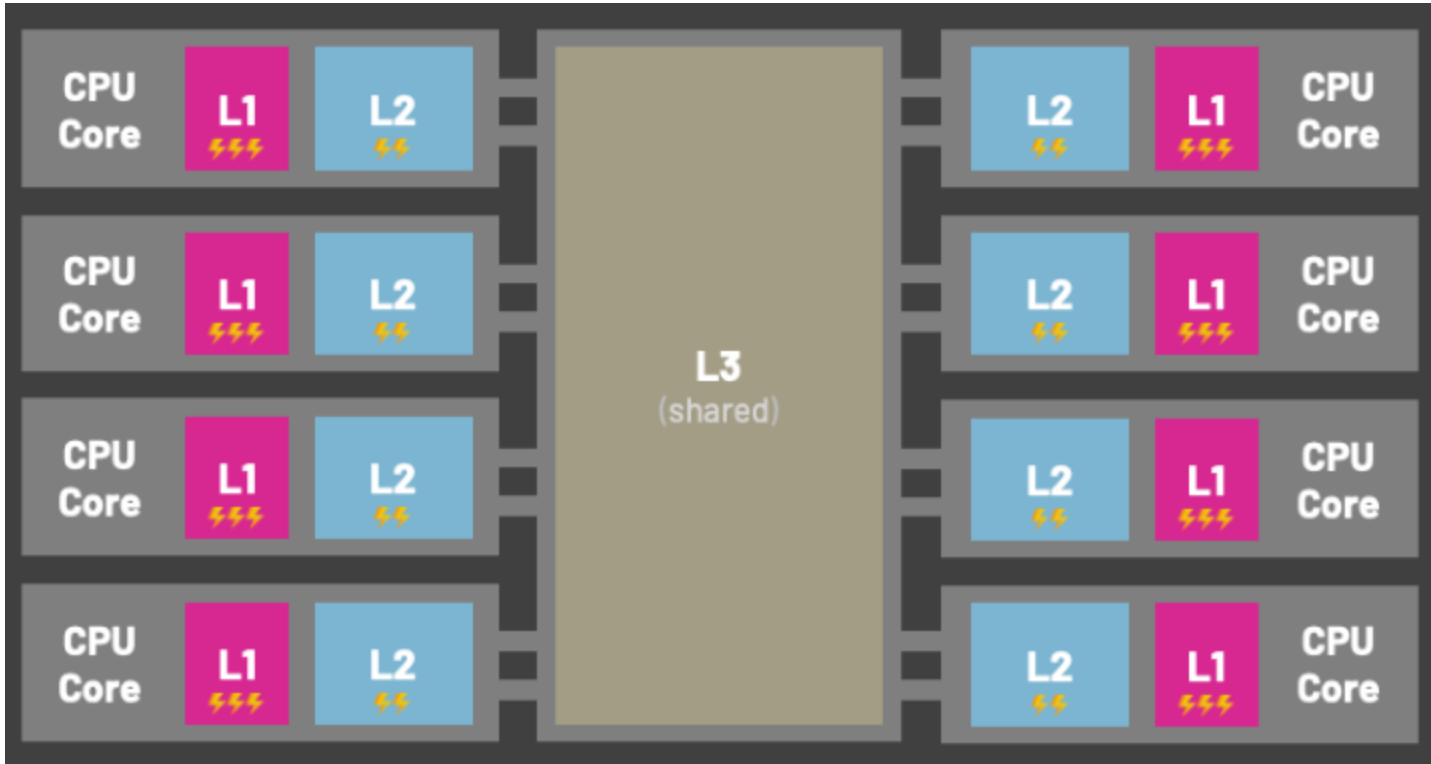
# But hey it's slow

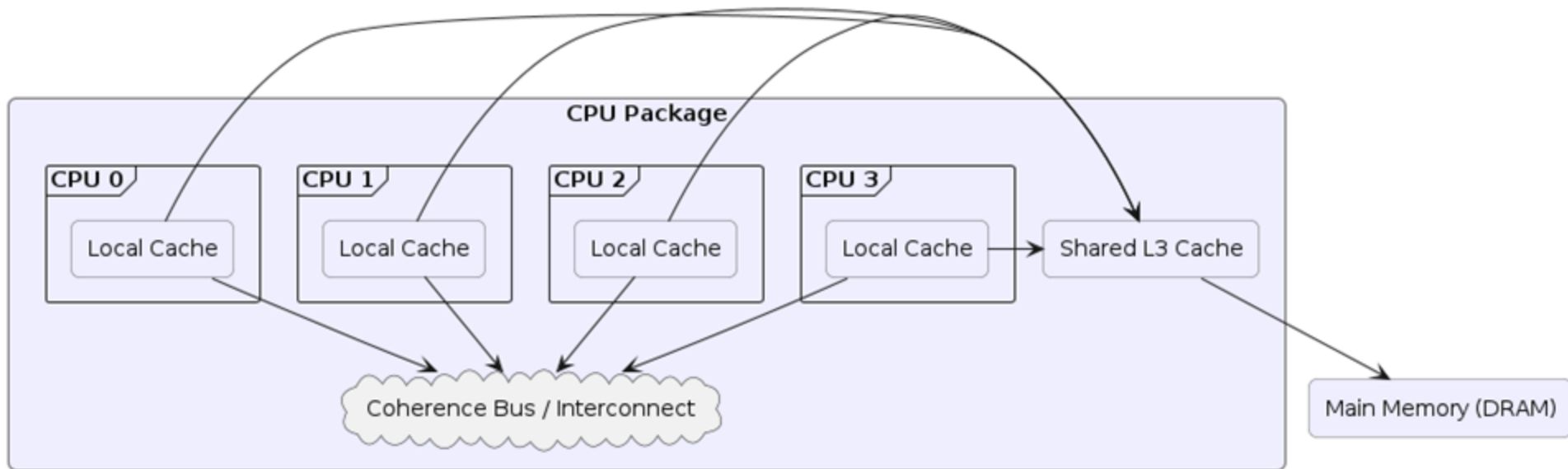
When ordering is enforced:

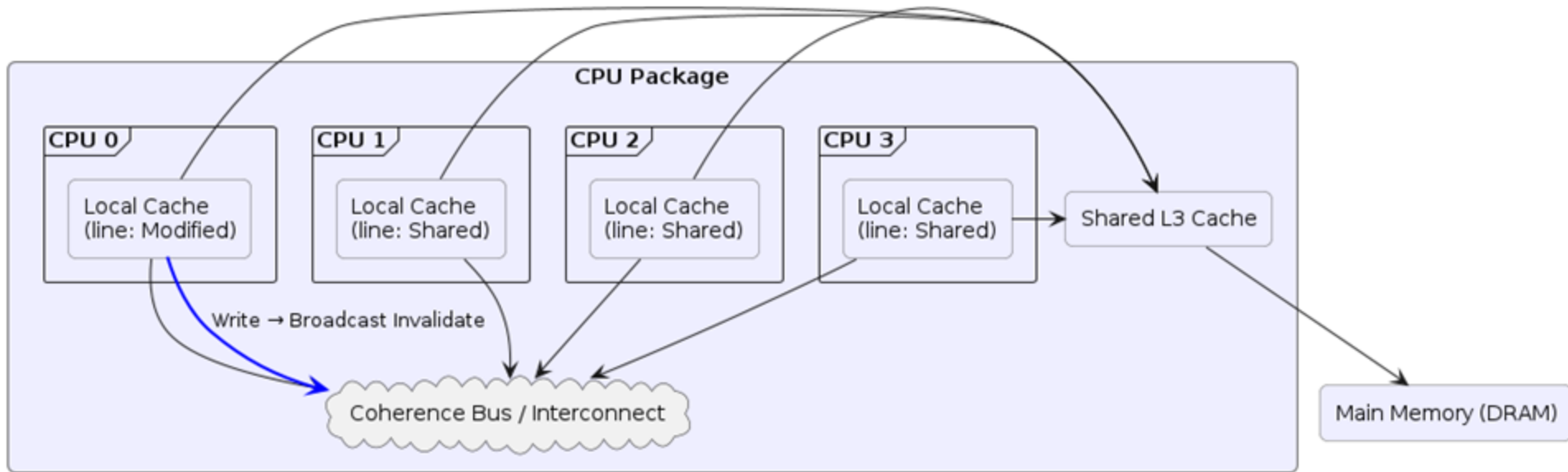
- some compute units will just stall
- **Memory will be practically locked**

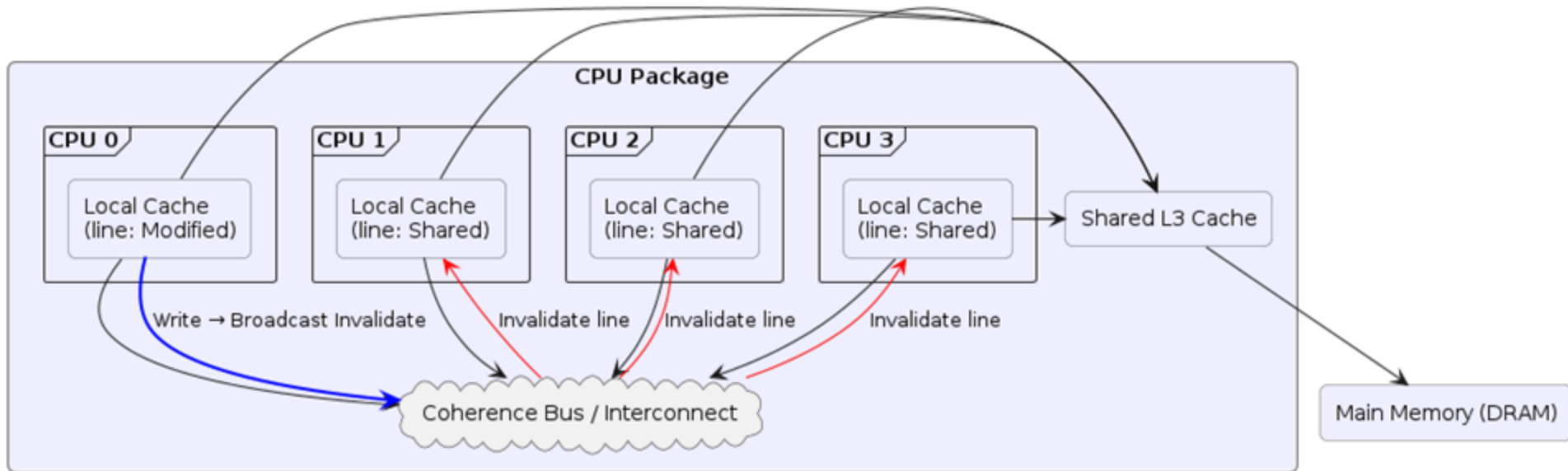


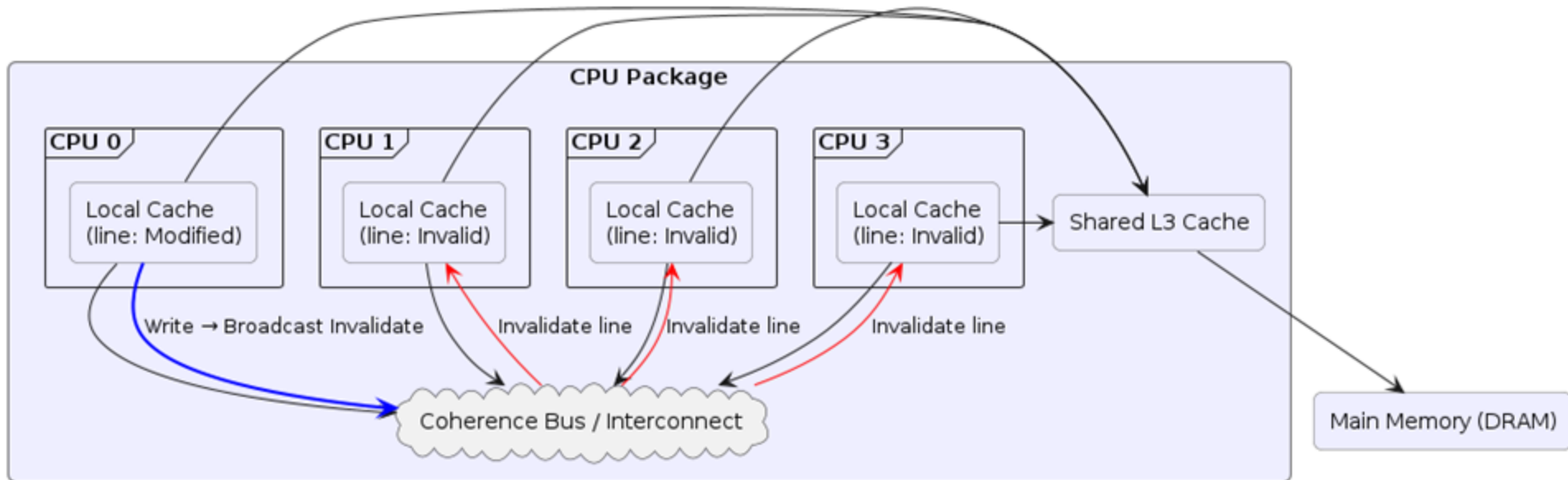
# Cache







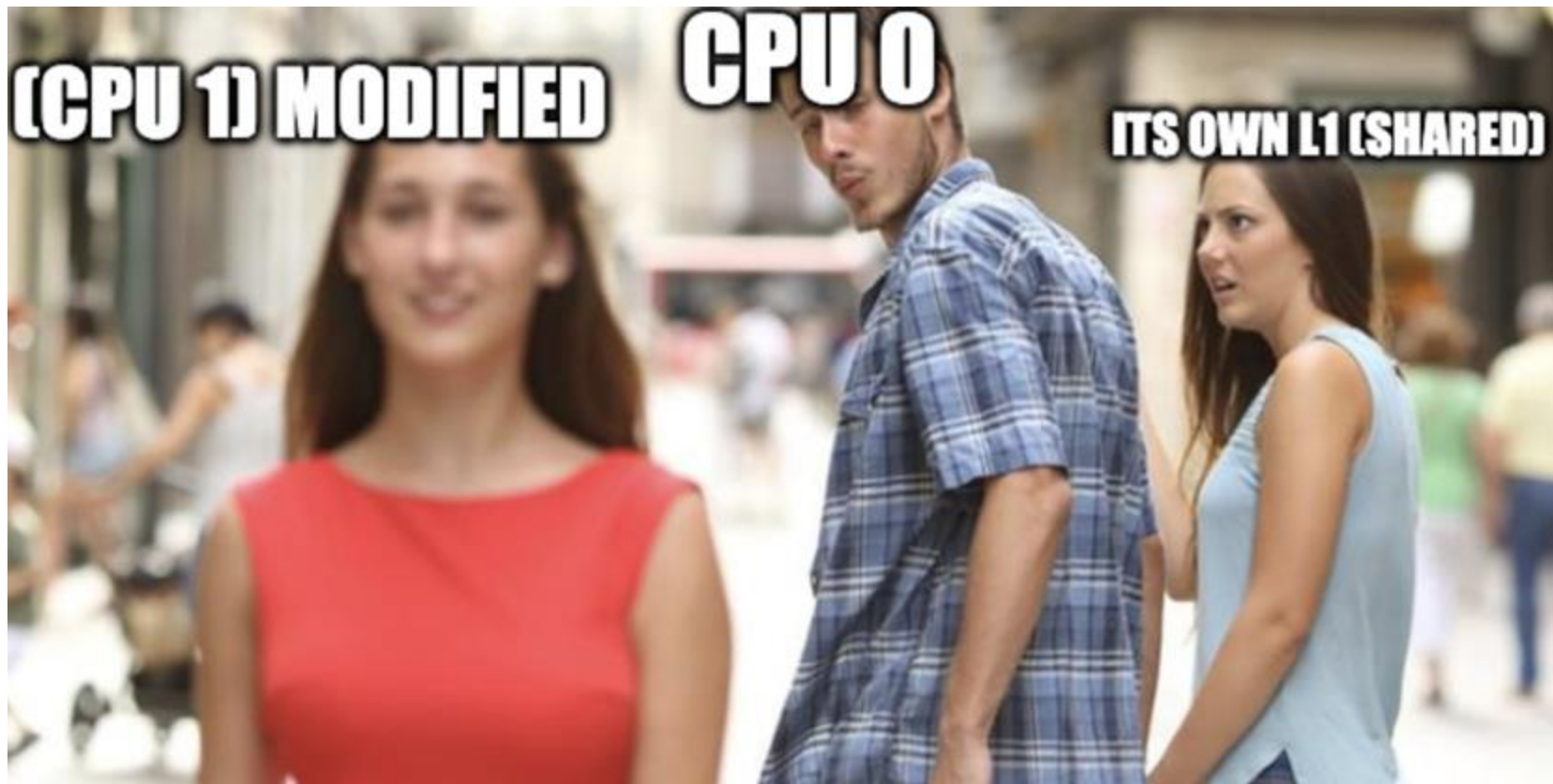




**(CPU 1) MODIFIED**

**CPU 0**

**ITS OWN L1 (SHARED)**



# Store buffer

(empty)
(empty)
(empty)

--- Step 0 ---

-> %EAX = add %EBX  
    %EAX = mul %EBX  
    store %EAX -> @X

%EAX = 2

%EBX = 3

Memory:

X = 0



# Store buffer

(empty)
(empty)
(empty)

--- Step 1 ---

%EAX = add %EBX  
-> %EAX = mul %EBX  
store %EAX -> @X

%EAX = 5

%EBX = 3

Memory:

X = 0

# Store buffer

(empty)
(empty)
(empty)

--- Step 2 ---

%EAX = add %EBX  
%EAX = mul %EBX  
-> store %EAX -> @X

%EAX = 15

%EBX = 3

Memory:

X = 0

# Store buffer

@X=15
(empty)
(empty)

--- Step 3 ---

%EAX = add %EBX  
%EAX = mul %EBX  
store %EAX -> @X

%EAX = 15

%EBX = 3

Memory:  
X = 0

# Store buffer

@X=15
(empty)
(empty)

--- Step ... ---

%EAX = add %EBX  
%EAX = mul %EBX  
store %EAX -> @X

...

Memory:  
X = 0

%EAX = 15

%EBX = 3

# store->load forwarding

@X=15
(empty)
(empty)

--- Step ...+1 ---

%EAX = add %EBX  
%EAX = mul %EBX  
store %EAX -> @X

...  
-> load %ebx <- @X

Memory:  
X = 0

%EAX = 15

%EBX = 3

# Store buffer implications

Now because we log our speculative stores, we can reorder:

store [x] = 1

load [y]

...

Thread 1

Thread 2

store [x] = 1  
load [y]

store [y] = 1  
load [x]

...

Thread 1

Thread 2

store [x] = 1  
load [y] ; 0

store [y] = 1  
load [x] ; ...Also 0??



## Another example

T1:

Cmpxchg word mem = 0x0000  
- > 0x0101

T3:

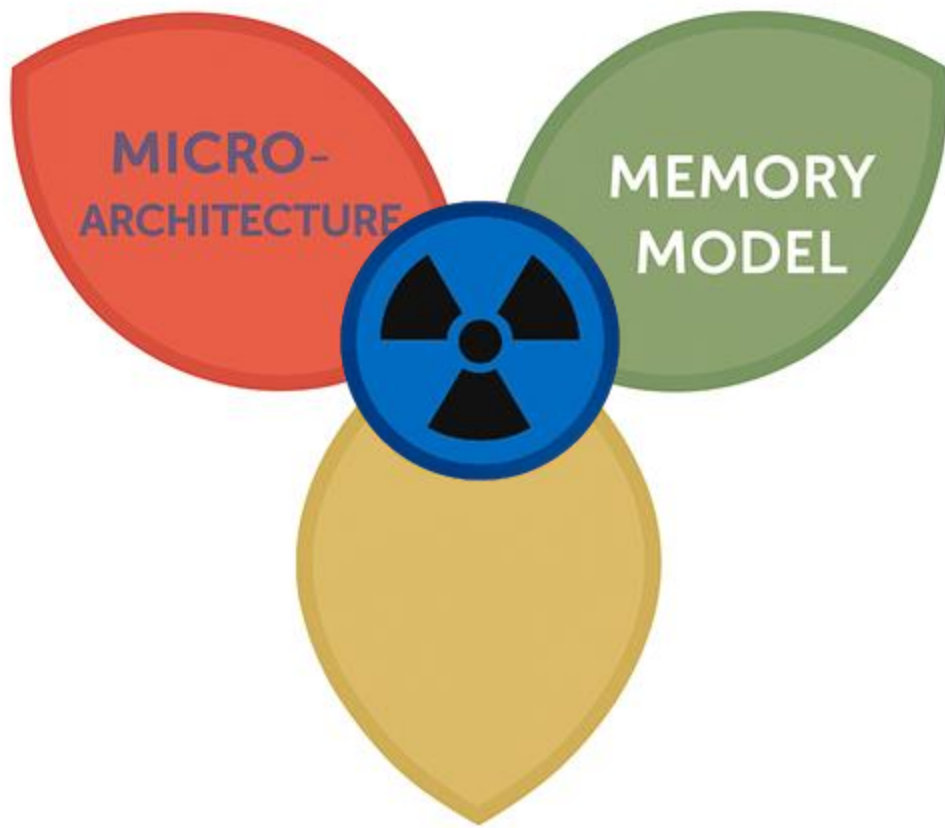
Write byte mem = 0x03

Read mem to reg

T2:

Cmpxchg word mem 0x0101 -  
> 0x0202

The cache coherency can get  
0x0203 but reg in T3 can be 0x0103  
We got value that was never in the  
cache.



# Simplified C++ memory model

- We can't have two regular operations on the same variable when at least one of them is write.
- Atomic operations make sure a single operation will work as intended.
- If we want some sequence of operations to work one after another we need synchronization.

# Relaxed

Do not impose any Synchronization.

Need to be very careful with them as they can be unintuitive.

Within a single thread, relaxed atomics cannot appear to execute out of program order.

# Relaxed

## Example:

// Thread 1:

```
r1 = y.load(std::memory_order_relaxed); // A
```

```
x.store(r1, std::memory_order_relaxed); // B
```

// Thread 2:

```
r2 = x.load(std::memory_order_relaxed); // C
```

```
y.store(42, std::memory_order_relaxed); // D
```

Allowed to be `r1 == r2 == 42`

# Relaxed since C++14

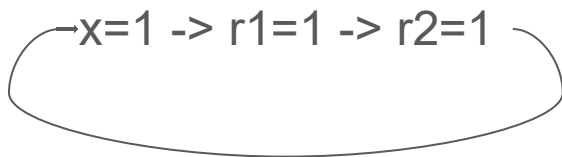
Circular reasoning is disallowed.

// Thread 1

```
r1 = y.load(std::memory_order_relaxed);  
if (r1 == 1) x.store(1, std::memory_order_relaxed);
```

// Thread 2

```
r2 = x.load(std::memory_order_relaxed);  
if (r2 == 1) y.store(1, std::memory_order_relaxed);
```



# Release and acquire

If the acquire sees the release store then:

- The release happens before the acquire

- Everything that happened before the release the acquire sees

Only matching\* release and acquire threads will synchronize

## The problem with release and acquire

Thread 1

```
x = 1;
```

Thread 2

```
y = 1;
```

Thread 3

```
if( x==1 && y= 0 )  
    print( "x first" );
```

Thread 4

```
if( y==1 && x=0 )  
    print( "y first" );
```



# sequential consistency

The very simplistic model, everything is total ordered



# So why should we even bother?

Benchmark 1: `lockless_ring_buffer_spisc.exe`

Time (mean  $\pm \sigma$ ): 678.6 ms  $\pm$  122.7 ms

Range (min ... max): 285.5 ms ... 970.2 ms

[User: 1289.1 ms, System: 40.5 ms]

100 runs

Benchmark 2: `fine_grained_lockless_ring_buffer_spisc.exe`

Time (mean  $\pm \sigma$ ): 158.6 ms  $\pm$  44.1 ms

Range (min ... max): 90.5 ms ... 349.4 ms

[User: 259.1 ms, System: 35.6 ms]

100 runs

## Summary

`fine_grained_lockless_ring_buffer_spisc.exe` ran

4.28  $\pm$  1.42 times faster than `lockless_ring_buffer_spisc.exe`

# So how does the cpu do it?

Intel:

All rmw are Seq CST, store and loads are Release and Acquire.

armv7:

Has only memory barriers and LL SC.

armv8:

Has special instruction for acquire and release. Seq CST = acq\_rel

# Small note: LL/SC

LL *r*, [*addr*] → loads the value at *addr* into *r* and sets a *reservation* on that address.

...

SC [*addr*], *r2* → attempts to store *r2* into *addr*.

(Succeeds only if the reservation is still valid.)

Some common patterns

# Lightweight synchronization

```
void producer() {  
    data = 42;  
    ready.store(true, std::memory_order_release);  
}  
  
void consumer() {  
    while (!ready.load(std::memory_order_acquire));  
    std::cout << data; // guaranteed to see 42  
}
```

# Locks

```
class SpinLock {  
    std::atomic_flag locked{};  
public:  
    void lock() {  
        while (locked.test_and_set(std::memory_order_acquire));  
    }  
    void unlock() {  
        locked.clear(std::memory_order_release);  
    }  
};
```

\*don't write locks like that, goto Igor and ben lecture to learn how to

# CAS loops

1. Read a shared memory location  $V$  (the expected value).
2. Compute a new value  $V'$  based on the current state.
3. Attempt to atomically set the memory location to  $V'$  only if it still equals the expected value.
4. If the CAS fails (another thread changed the memory), repeat from step 1.



# CAS loops

1. Read a shared memory location  $V$  (the expected value).
2. Compute a new value  $V'$  based on the current state.
3. Attempt to atomically set the memory location to  $V'$  only if it still equals the expected value.
4. If the CAS fails (another thread changed the memory), repeat from step 1.

Dynamic value based on state -> CAS

# CAS loops

```
new_node = new Node(val, head);  
while (!head.compare_exchange_weak(new_node->next, new_node));
```

Both `fetch_and_*` and `exchange` can be implemented with simple cas loop.

This is the way the new `fetch_min/max` are implemented in x86

# Takeaways

- We can make stuff go faster!

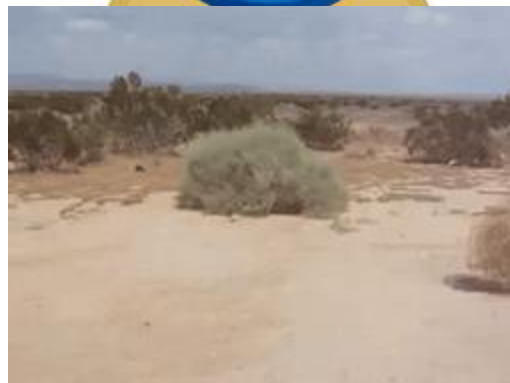
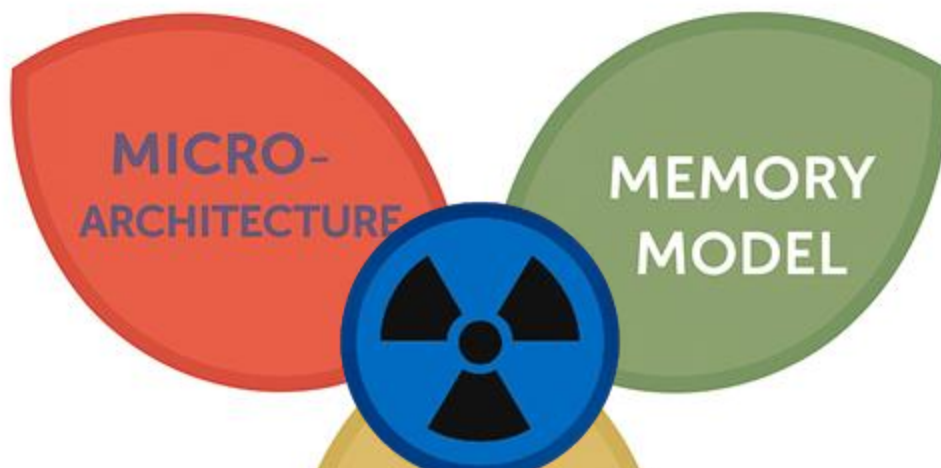
# Takeaways

- We can make stuff go faster!
- Micro-architecture is interesting! ...And sometimes useful.

END

# END

But we have time...



## Bonus 1 - sat solver

SAT (Boolean satisfiability) is a central problem in theoretical CS academic. It asks whether a formula of AND of OR-clauses of boolean variables have a solution.

It is the first problem proven to be NP-complete.



# Example

Given this code to get mutual exclusion

```
std::atomic<int> x = 0, y = 0;

void thread() {
    int tid = std::this_thread::get_tid();
    while (1) {
        x = tid;
        if (y && y != tid) continue;
        y = tid;
        if (x && x != tid) continue;
        /* critical section */
    }
}
```

# solution

<https://colab.research.google.com/drive/1Ep8dU4dF1GKCW7qlfLys4OcNWYEgcAXO?usp=sharing>

# Pros and cons

## Pro:

- Finds concrete counterexamples.

- There is multiple encoders(CBMC,KLEE) and solvers(z3).

## Cons:

- Can blow up for large depths.

- Can make mistakes encoding the constraints.

- In the end it can't really solve just say that in some number of steps there isn't bug.

## Bonus 2 - CAS and the aba problem

```
void LL::push_front(int val)
{
    auto new_node = new Node{val, this->head};
    while (!this->head.compare_exchange_weak(new_node->next, new_node));
}

auto LL::pop_front()
{
    Node* ret_ptr;
    Node* next_ptr;
    do {
        ret_ptr = head.load();
        next_ptr = ret_ptr->next;
    } while (!this->head.compare_exchange_weak(ret_ptr, next_ptr));
    delete ret_ptr;
}
```

## Bonus 3 - volatile vs atomics

Volatile are just saying reads and writes will not be optimized.

Atomic variable on the other hand can be optimized but do impose synchronization.