Inside a Modern CPU

With Cliff Click

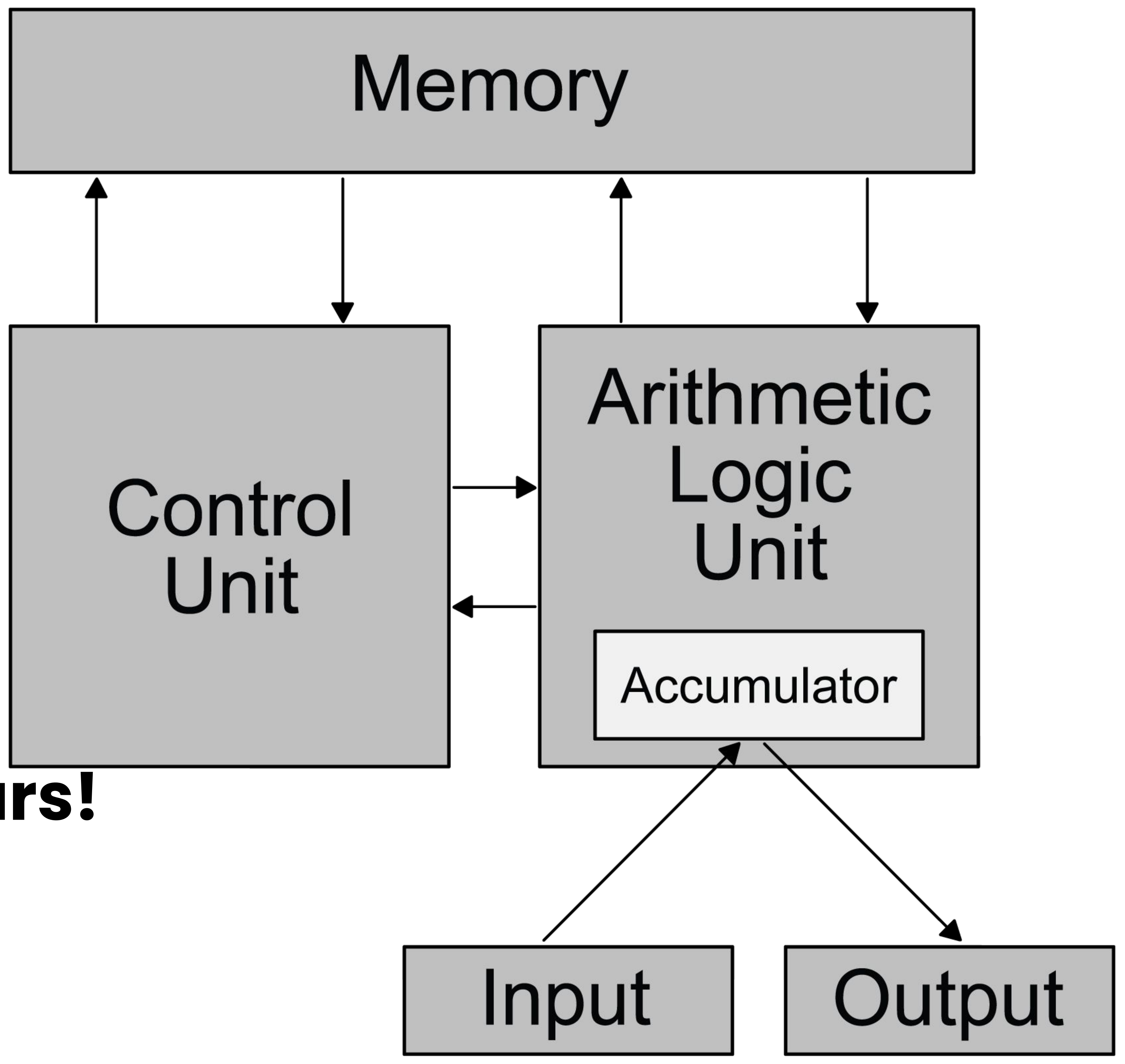
The Von Neumann Machine Architecture

What I was taught in school

Sequential Execution: One Instruction at a time

Great model for learning...

But it's not how computers have worked for 20 years!



Processor Trends

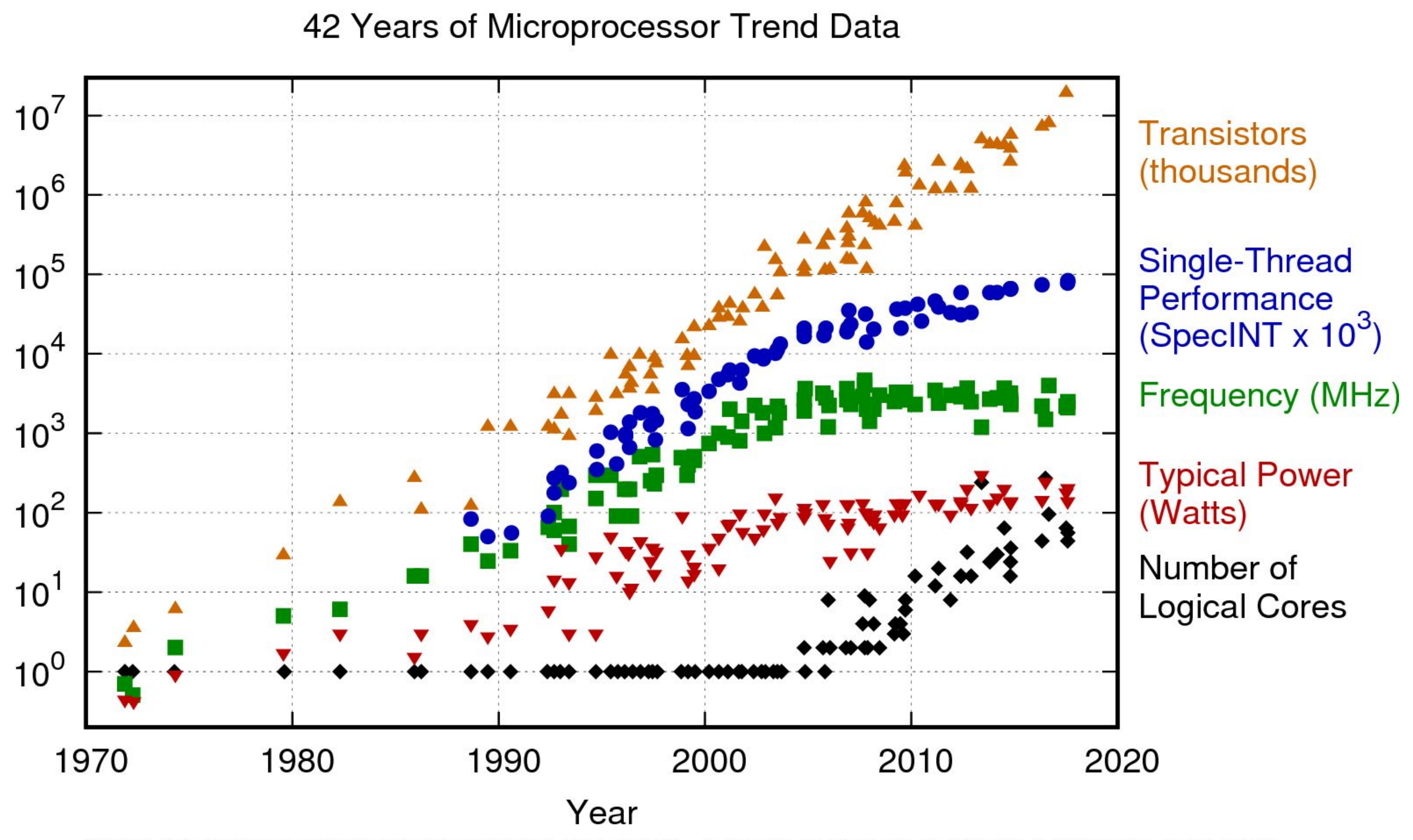
Clock frequency flat for 20 yrs

Still getting more transistors

Core-counts rising rapidly

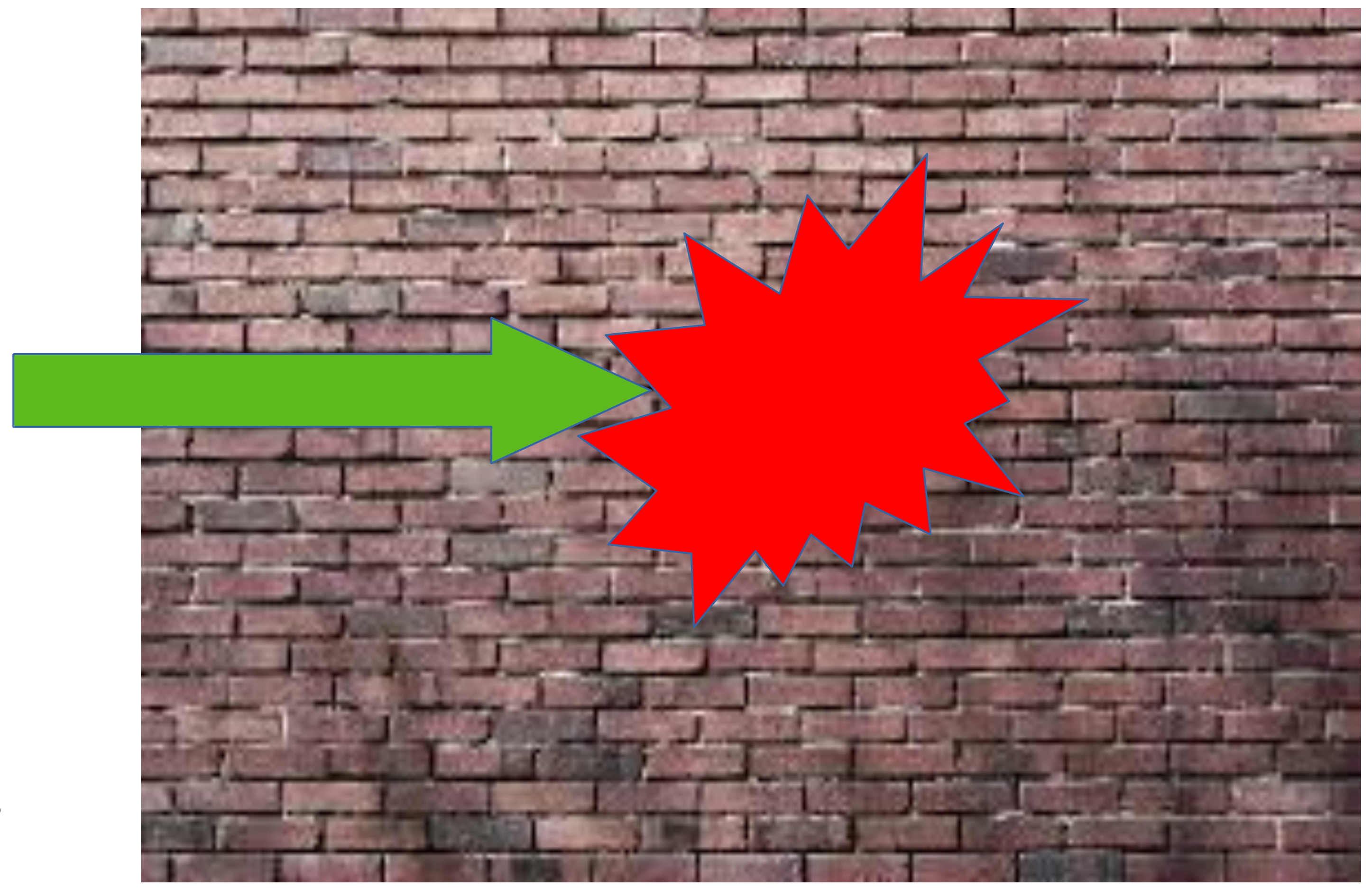
Performance rising slowly -

Why aren't computers getting faster?



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 2010-2017 by K. Rupp

Hitting A Wall...



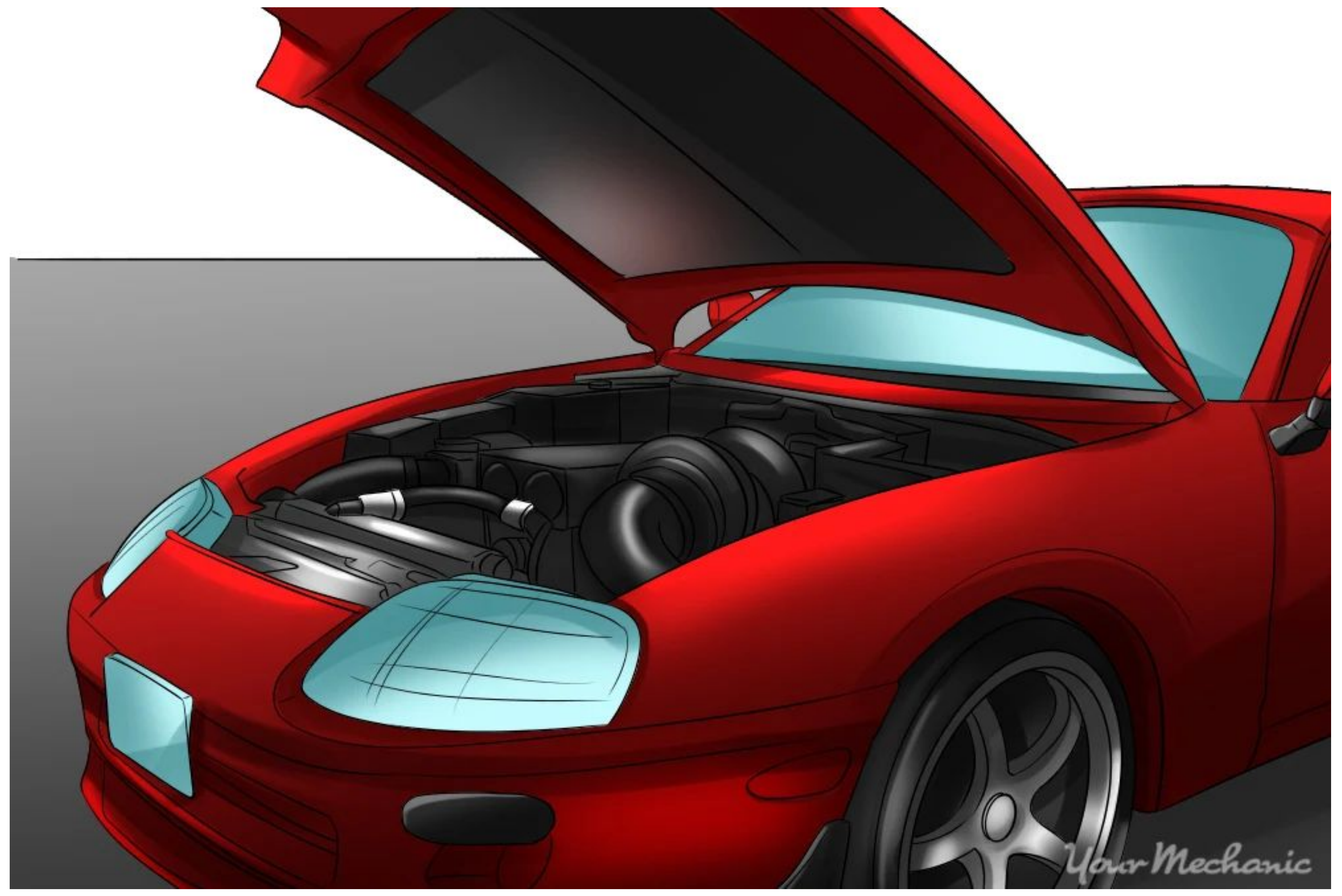
- **Power Wall**
  - Higher freq → more power → more heat → chip melts!
- **ILP (Instruction Level Parallelism) Wall**
  - Hitting limits in caching, branch prediction, speculative execution
- **Memory Wall**
  - Memory performance has lagged CPU performance
  - Program performance now dominated by cache misses
- **Speed of light**
  - Takes more than a clock cycle for a signal to cross a big chip

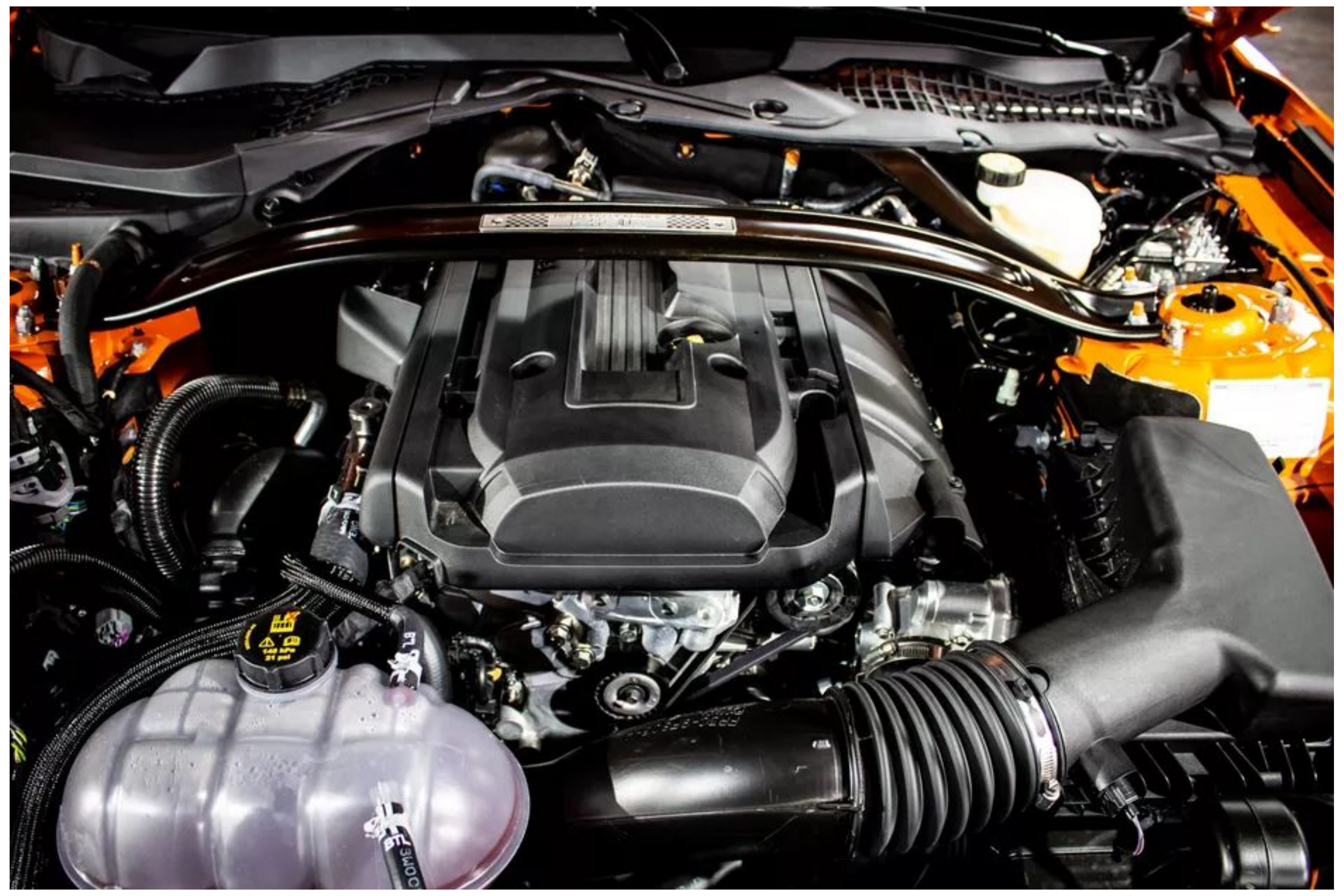
Instruction Level Parallelism

Lets look under the hood of a modern processor

Faster CPUs at the same clock rate:

- Pipelining
- Caching, Hit-Under-Miss
- Branch Prediction
- Speculative Execution
- Wide-Issue
- Out-Of-Order
- Register Renaming
- Prefetching





PIPELINING

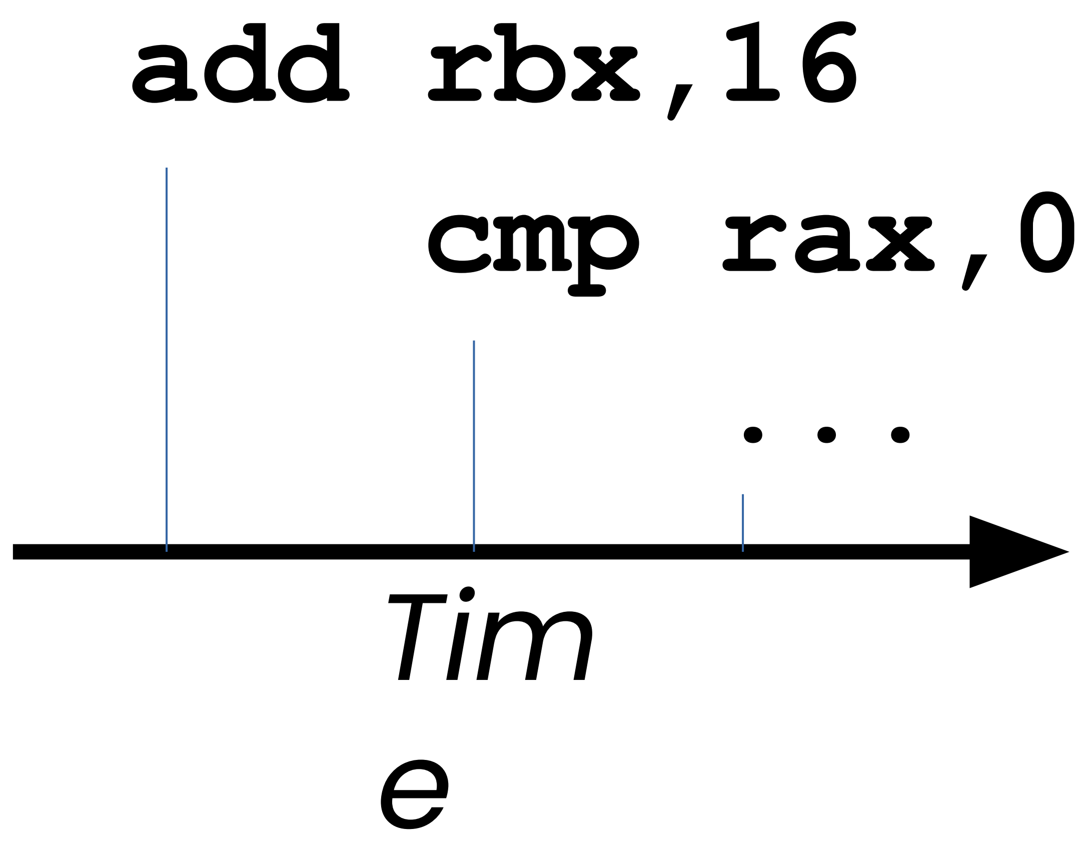
```
add   rbx,16      // add 16 to rbx
cmp   rax,0       // compare rax
...
```

On early machines, these would be e.g. 4 clks each

Pipelining allows them to appear as 1 clk

- And allows a higher clock rate
- Execution is parallelized in the pipe

Found on all modern CPUs



PIPELINING

Pipelining improves *throughput* not *latency*

- The deeper the pipe, the higher the (theoretical) effective frequency

All instructions take many cycles

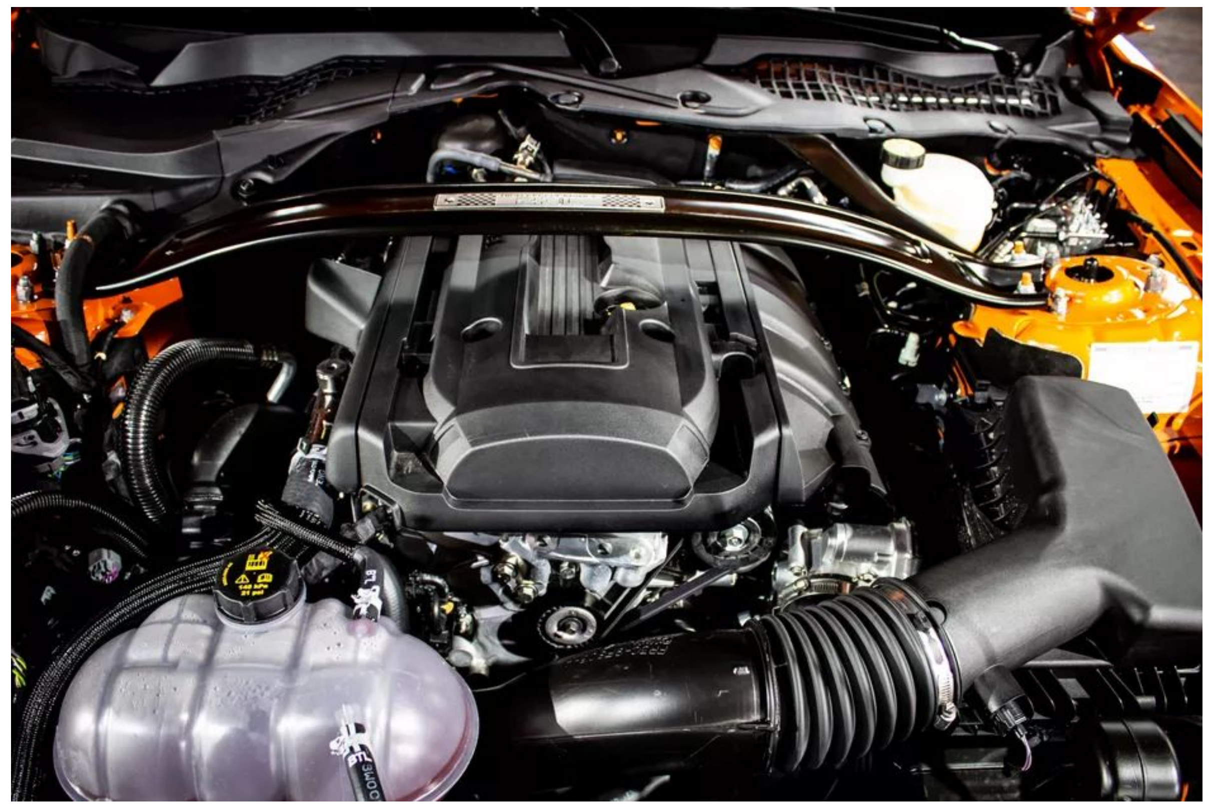
- From 10 to 30, depending on CPU & Instruction

Pipelining can fail! Lack of resources or conflicts

- Pipe maybe stalls a few clocks, or
- Pipe might have to *flush* & reload

LOADS & CACHES

```
ld  rax⇐[rbx+16]   // load @ rbx+16
```



Loads read from *cache* before *memory*

- A *cache hit* might be 2 or 3 clock cycles
- A *miss* might take 200 or 300 clock cycles to read from memory
- Many cache levels & variations

Loaded value might not be available for a long time!

- Notice **100x** variation in time!

What happens on a miss?

- Simplist CPUs, eg GPUs, stall until load is ready

LOADS & CACHES – A Miss

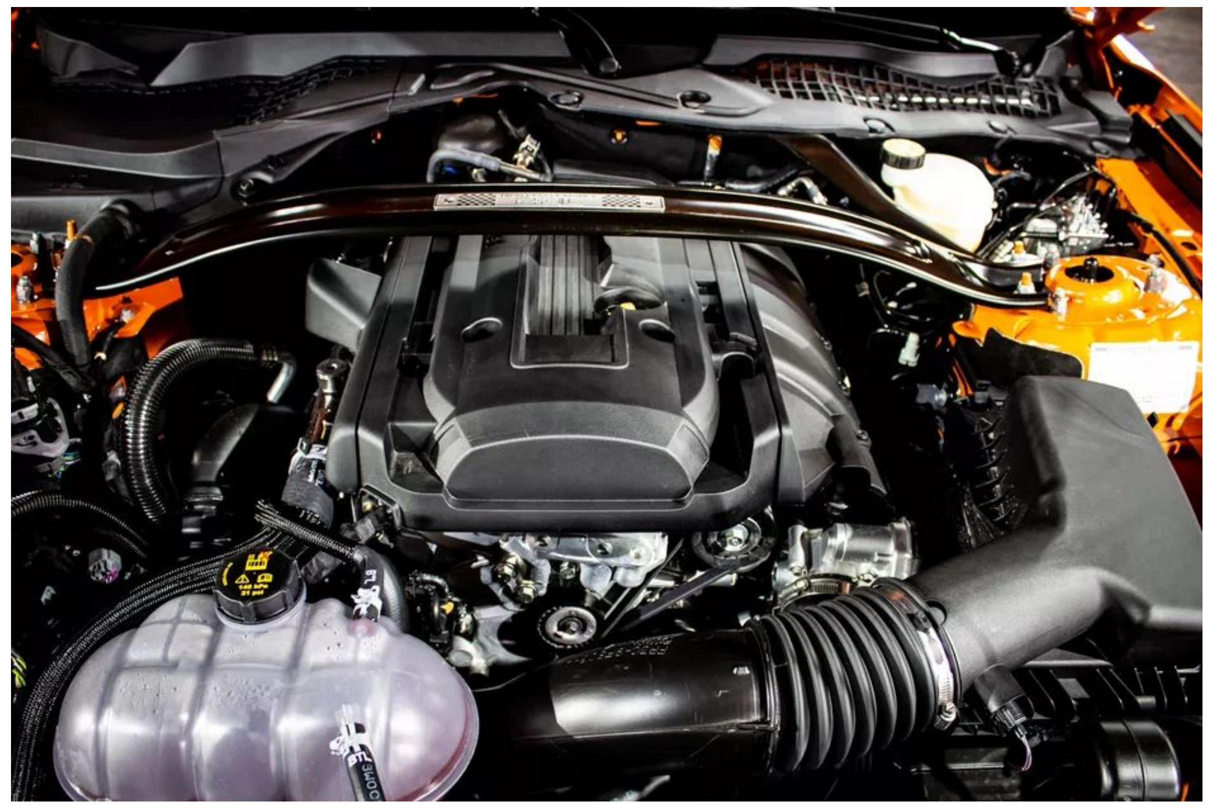
```
ld  rax, [rbx+16]   // load @ rbx+16
...
cmp rax,0           // test loaded value
```

Maybe execute in the *miss shadow*, until rax is used

- True dependence blocks – typically after a few clocks
- Load/Store resource tied up
- Fairly common; found on even embedded CPUs

Better: keep executing until the NEXT *cache miss*

Goal: run two cache misses in parallel, twice the throughput



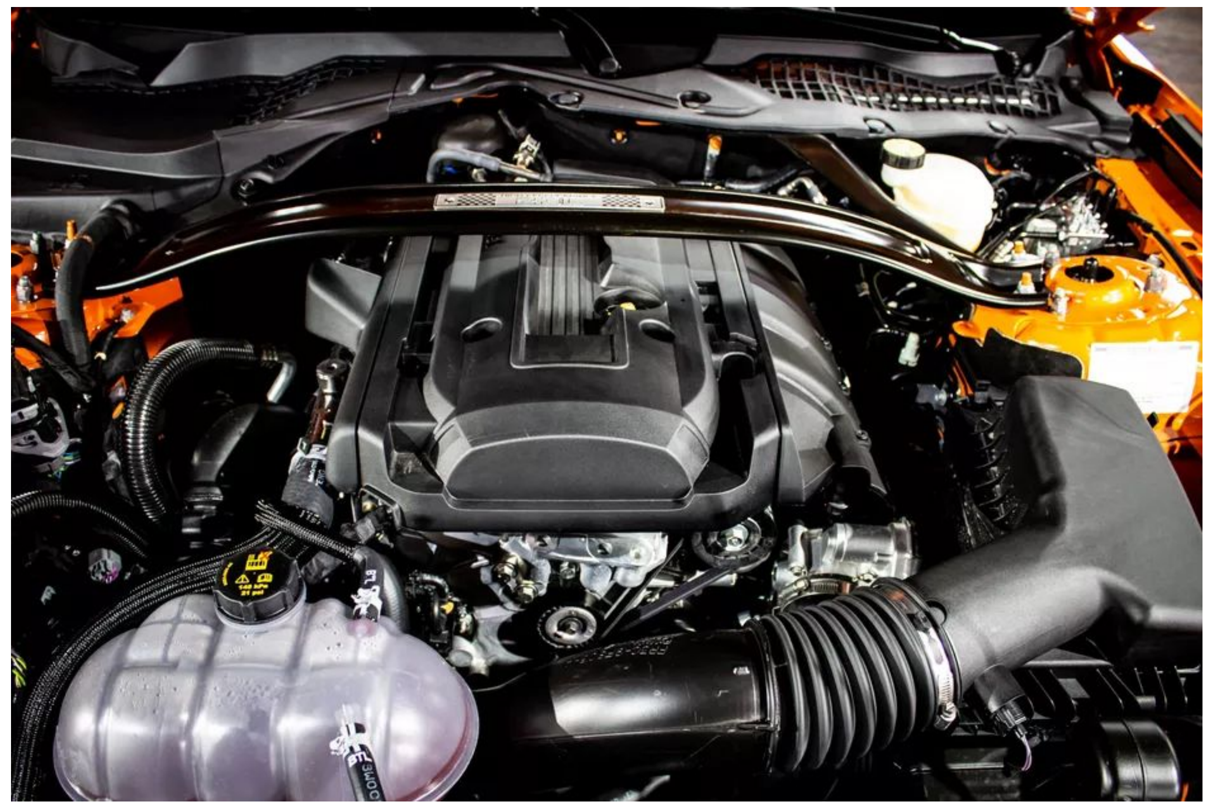
BRANCH PREDICTION

```
ld  rax⇐[rbx+16]   // load @ rbx+16
…
cmp rax,0          // test loaded value
jeq null_chk       // jump fail handling
st  [rbx-16]⇐rcx   // … speculative...
```

Test results often **not available** at branch!

So *predict* (guess) and run on:

- If wrong, pay *mispredict penalty* to clean up
- If right, makes forward progress
- Right > 95% of the time!



WIDE ISSUE

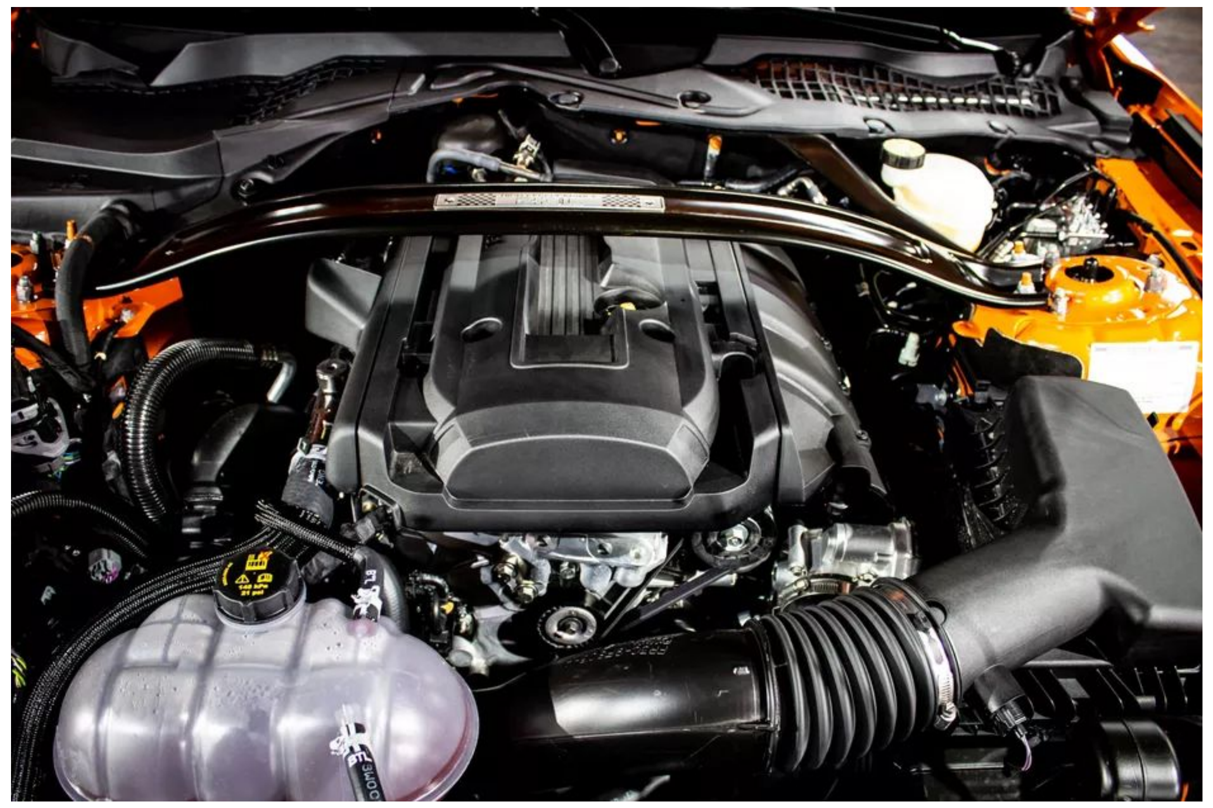
```
add   rbx,16      // add 16 to rbx
cmp   rax,0       // compare rax
...
```

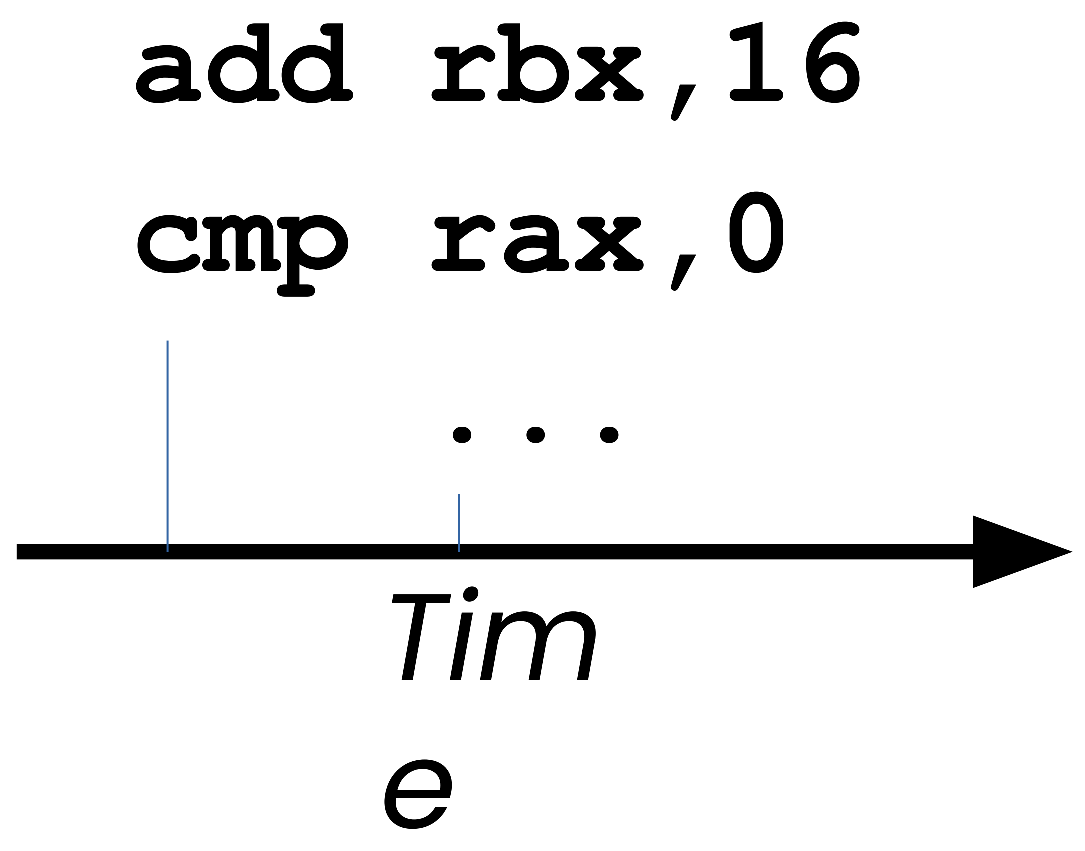
Do both instructions at once!

- No register or resource conflicts
- Twice as fast, when it works

Found on all modern CPUs

- Requires quadratic resources to check conflicts
- Rapidly diminishing returns





REGISTER RENAMING, SPECULATION, OUT-OF-ORDER EXECUTION

Register renaming, branch prediction, speculation, O-O-O are all *synergistic*

- Extra speculative state in extra registers
- Correct prediction: rename extras as "real" registers
- Mis-predict recovery: toss extra registers
- Like rolling back a broken transaction

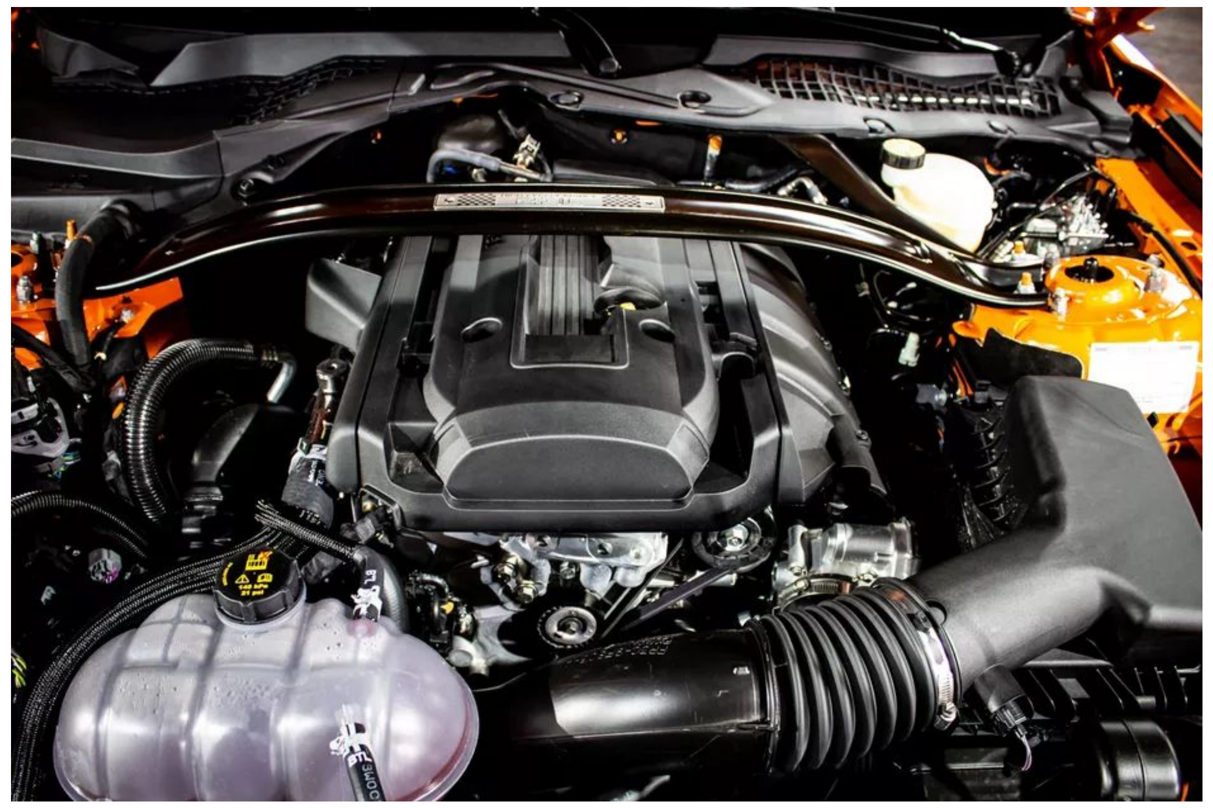
Can execute past branches, stores, everything

- Old Goal: Just Run Faster
- New Goal: Run until can start the **next** cache miss

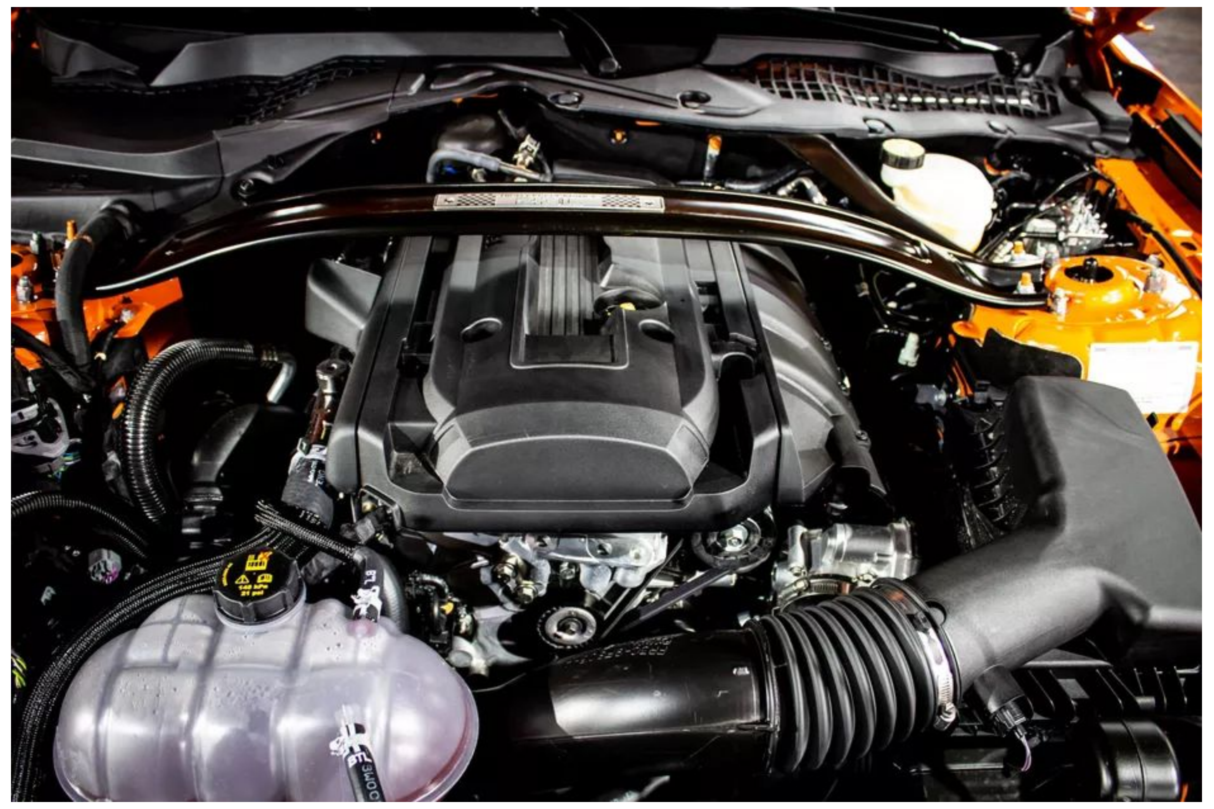
Goal: divide that 300 clock memory stall by parallel miss count!

A BIGGER EXAMPLE

```
ld   rax⇦[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
st   [rbx-16]⇦rcx
ld   rcx⇦[rdx+0]
ld   rax⇦[rax+8]
...
```



A BIGGER EXAMPLE

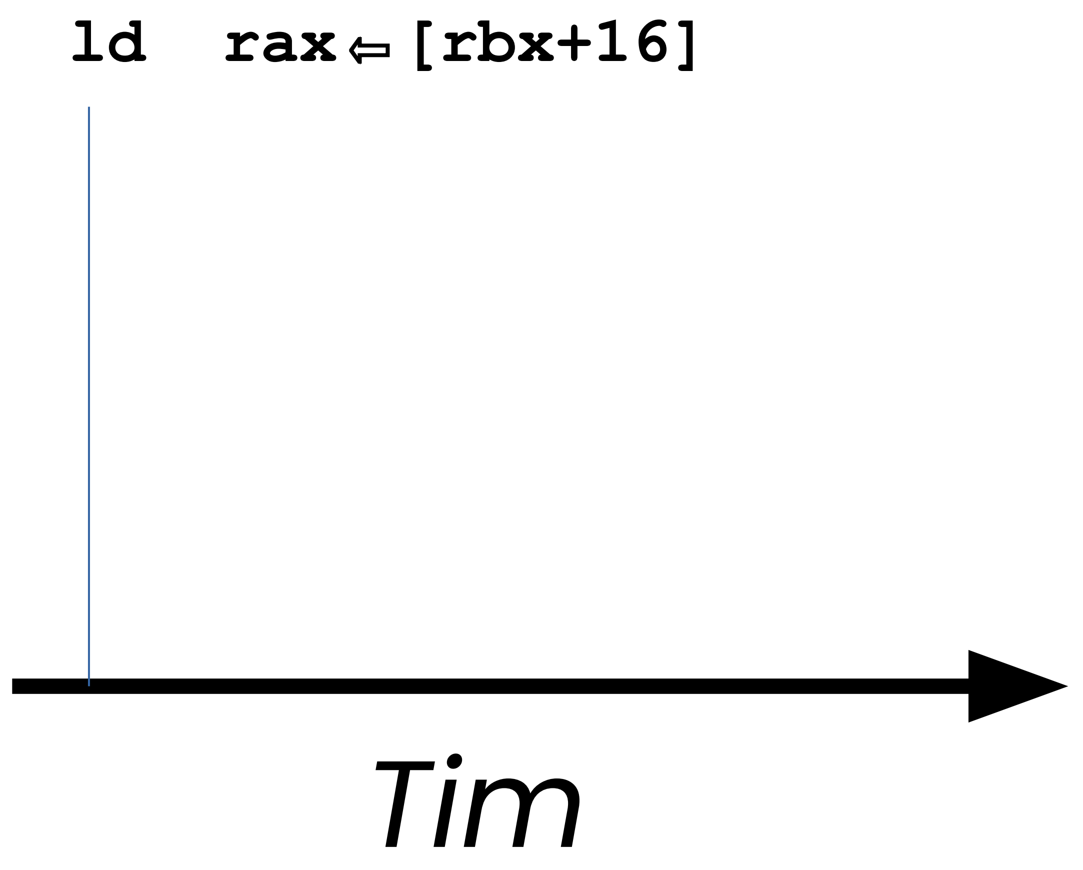


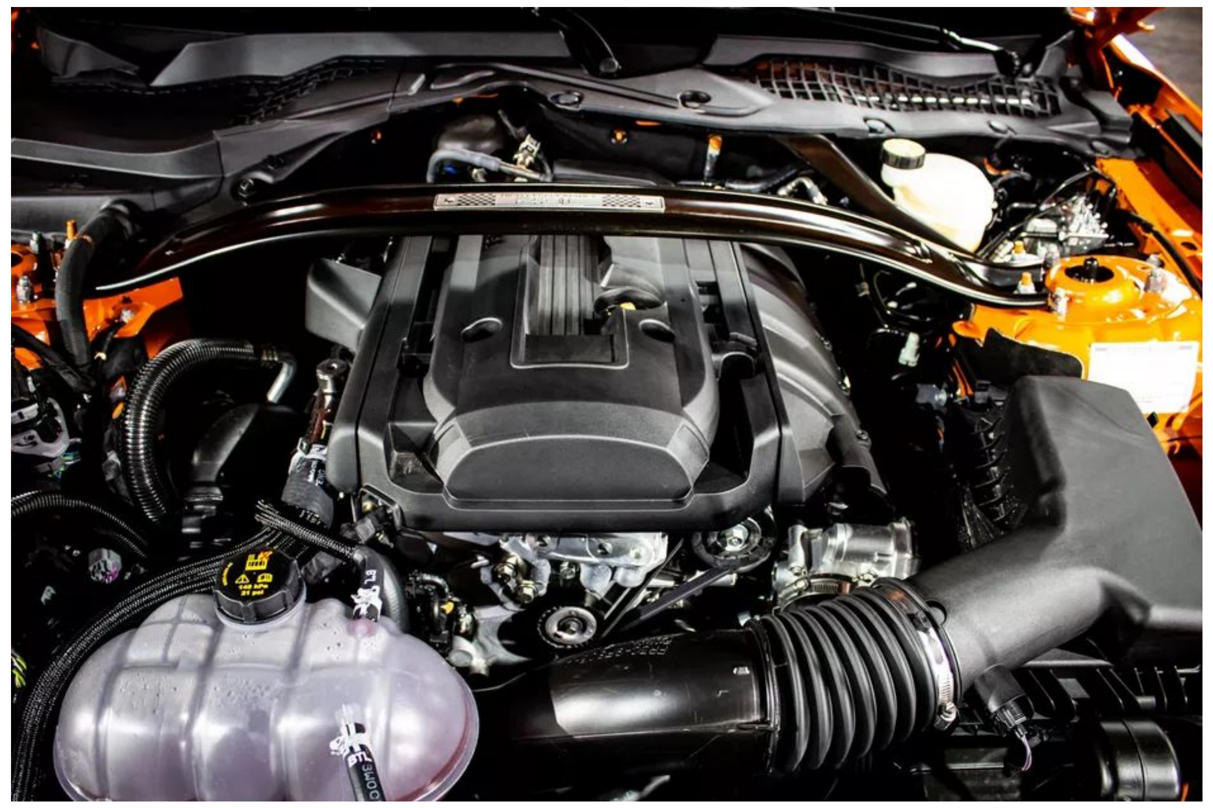
```
ld   rax⇐[rbx+16]
add  rbx,16
cmp  rax,0
jeq  null_chk
st   [rbx-16]⇐rcx
ld   rcx⇐[rdx+0]
ld   rax⇐[rax+8]
...
```

Load rax from memory.

Assume a *cache miss*! 300 clocks until ready.

Load starts, and execution continues…





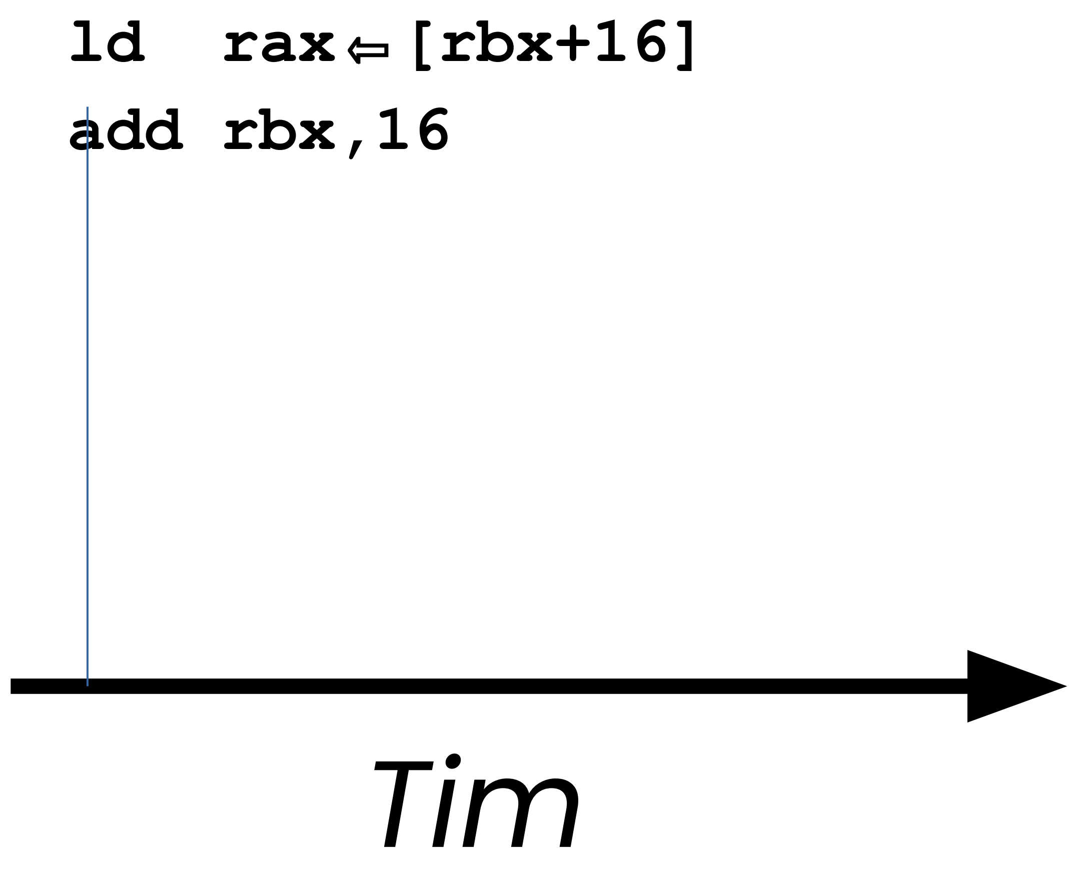
A BIGGER EXAMPLE

```
ld   rax⇦[rbx+16]
add  rbx,16
cmp  rax,0
jeq  null_chk
st   [rbx-16]⇦rcx
ld   rcx⇦[rdx+0]
ld   rax⇦[rax+8]
...
```

Wide issue: runs in parallel

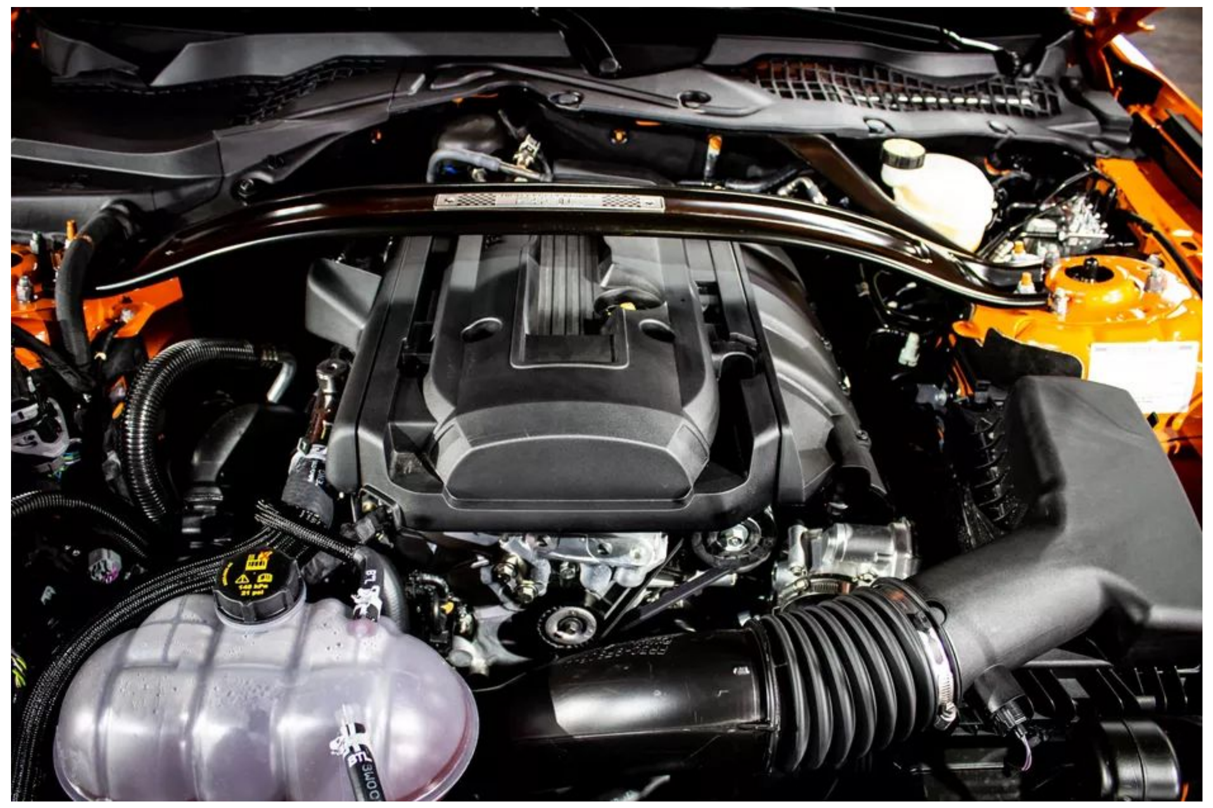
Writes rbx, which is read by prior op

Register renaming allows overlap



```
ld   rax⇦[rbx+16]
add  rbx,16
```

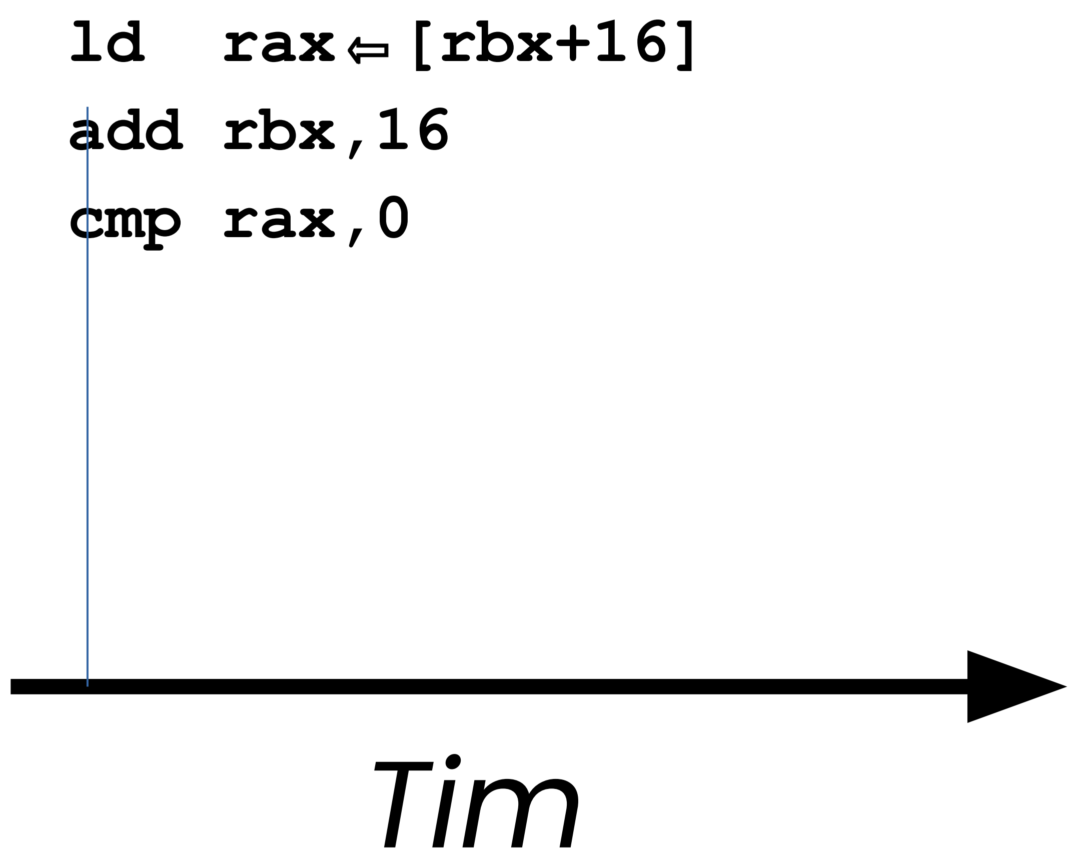
A BIGGER EXAMPLE



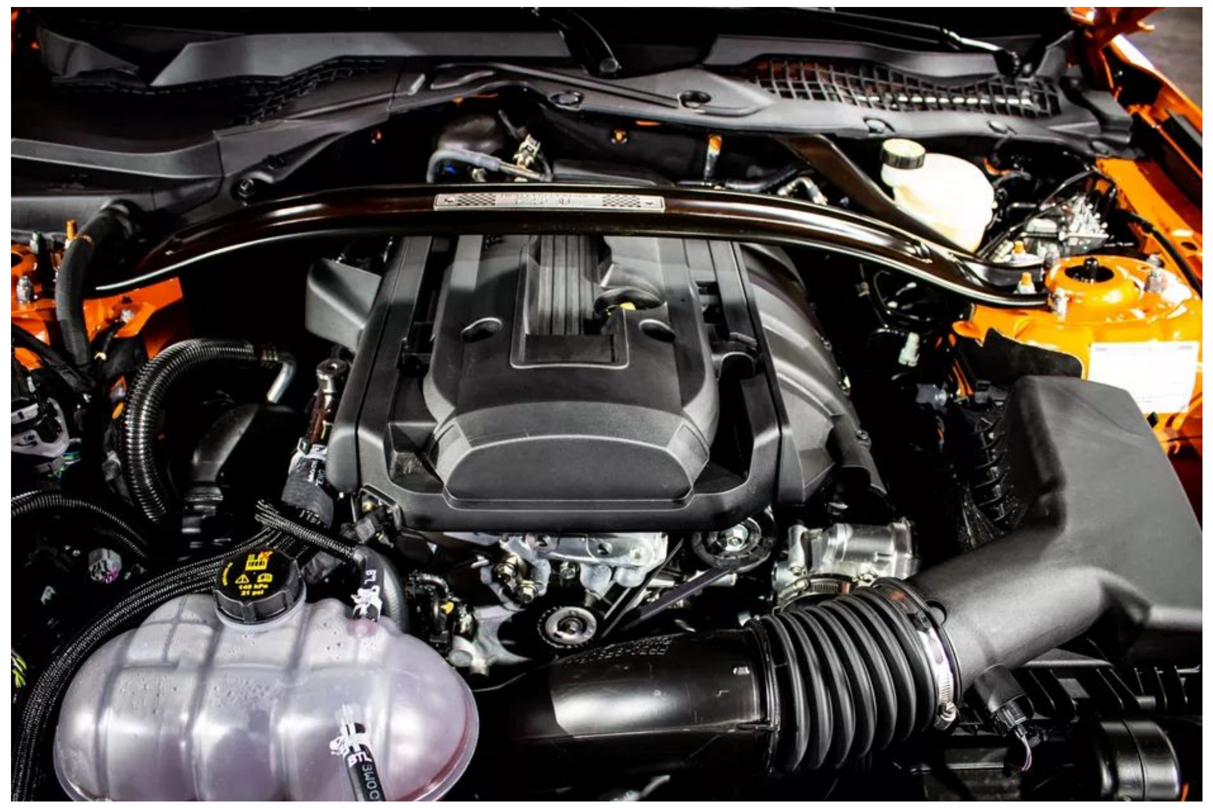
```
ld  rax⇐[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
st  [rbx-16]⇐rcx
ld  rcx⇐[rdx+0]
ld  rax⇐[rax+8]
...
```

rax not ready, so cannot compute *flags*

Queues up behind load



A BIGGER EXAMPLE

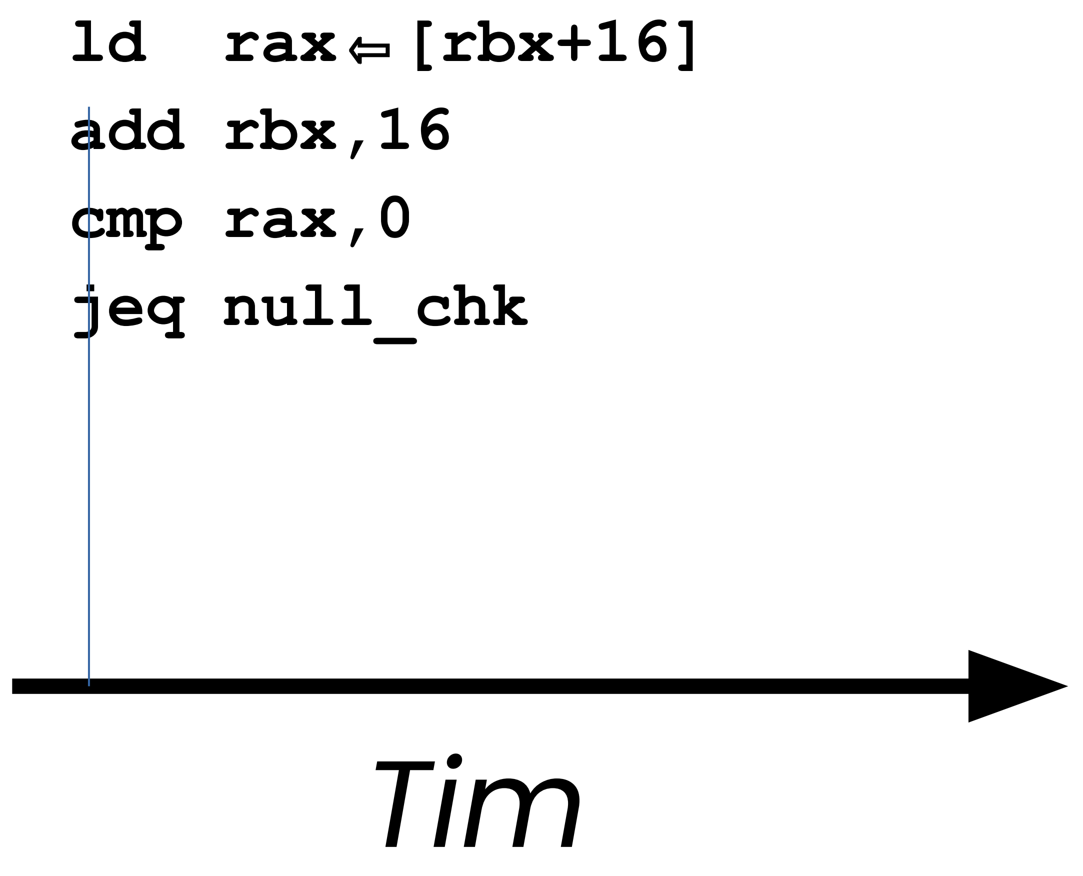


```
ld   rax⇐[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
st   [rbx-16]⇐rcx
ld   rcx⇐[rdx+0]
ld   rax⇐[rax+8]
...
```

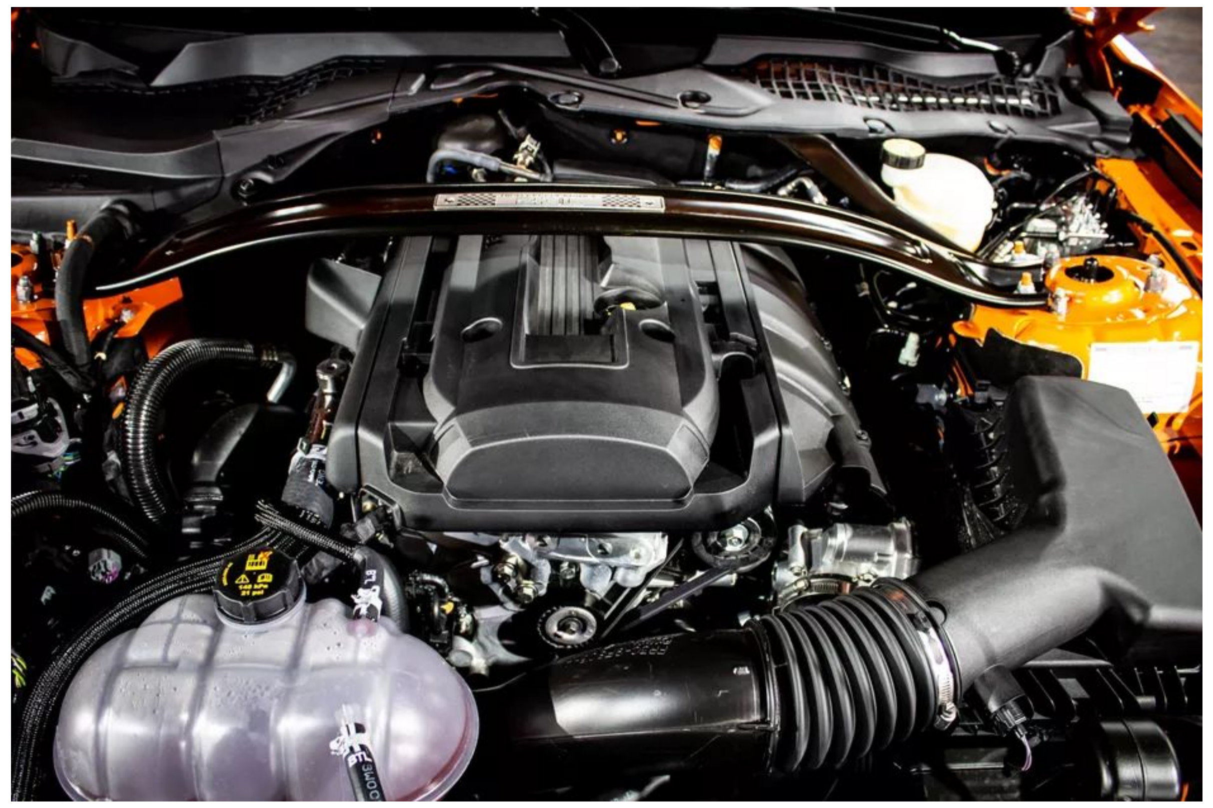
flags not ready

Branch predictor predicts not-taken

Limit of 4-wide issue



```
ld   rax⇐[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
```



A BIGGER EXAMPLE

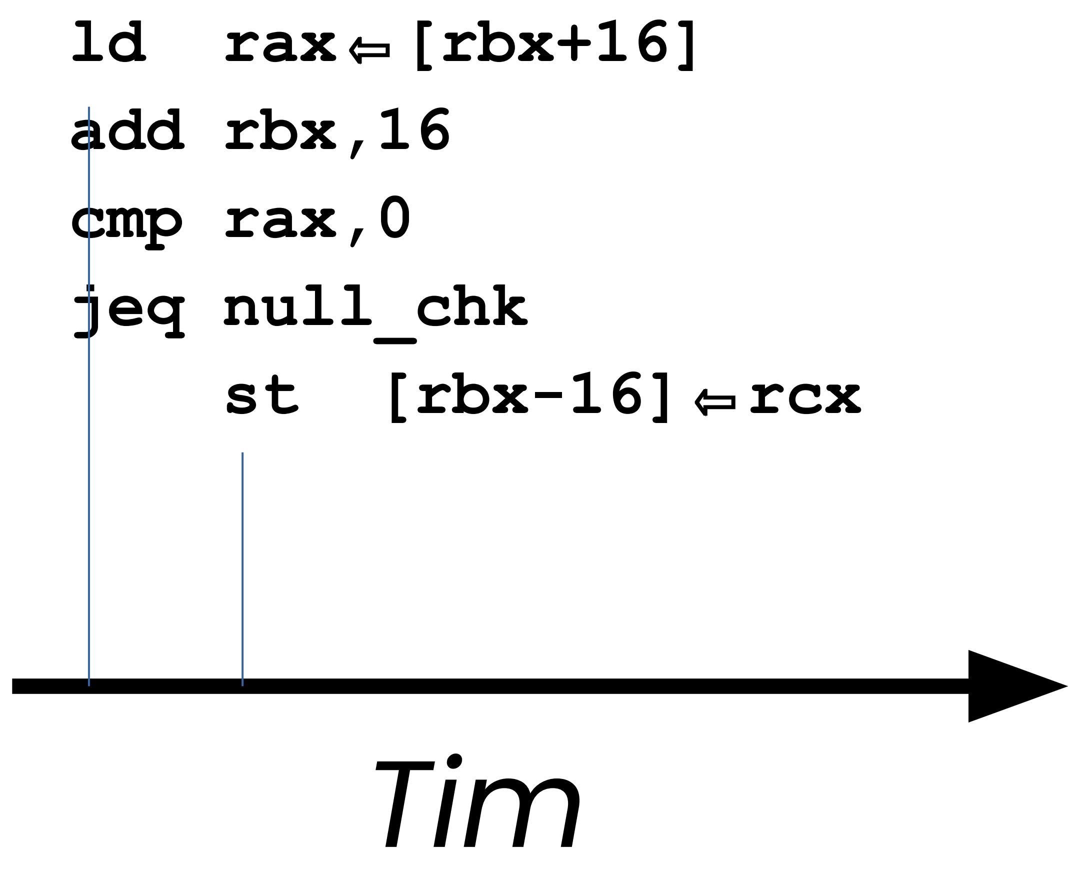
```
ld   rax⇐[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
st   [rbx-16]⇐rcx
ld   rcx⇐[rdx+0]
ld   rax⇐[rax+8]
...
```

Store is speculative

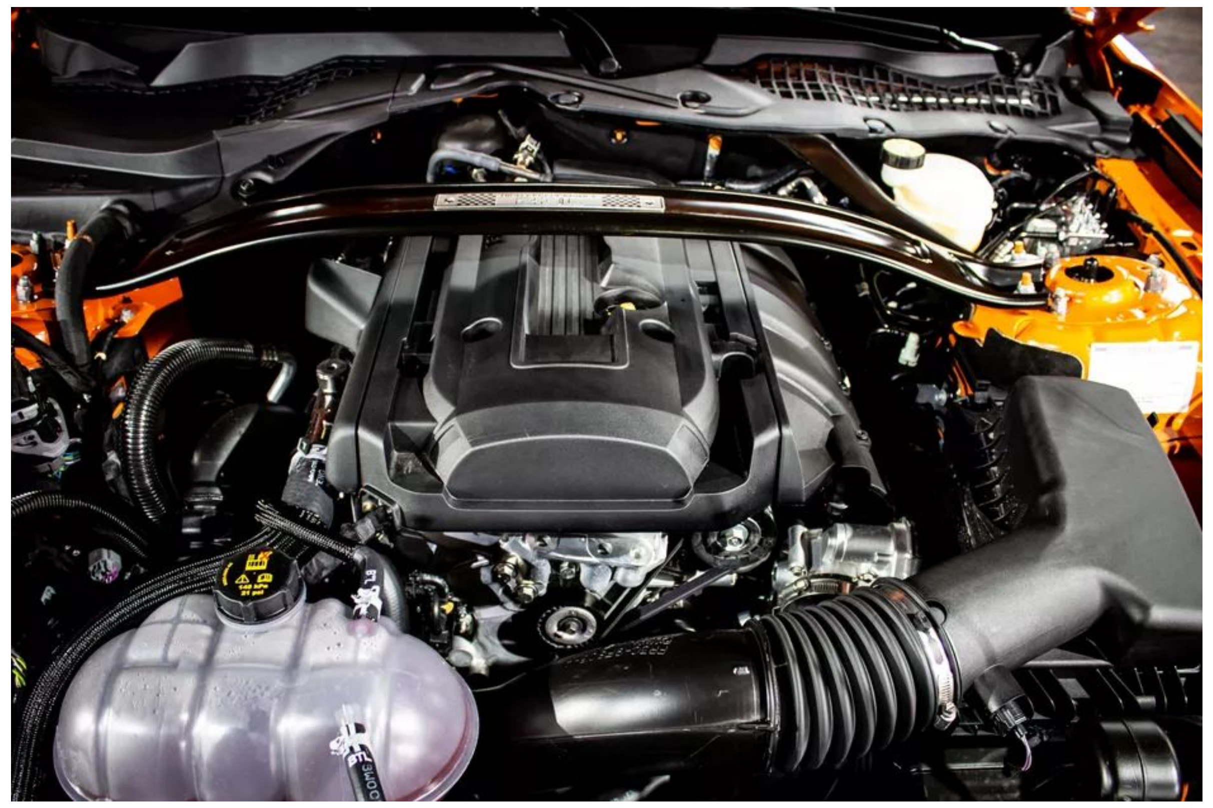
Kept in store buffer

Also, rbx might be null (fault)

L/S unit busy again



```
ld   rax⇐[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
     st   [rbx-16]⇐rcx
```



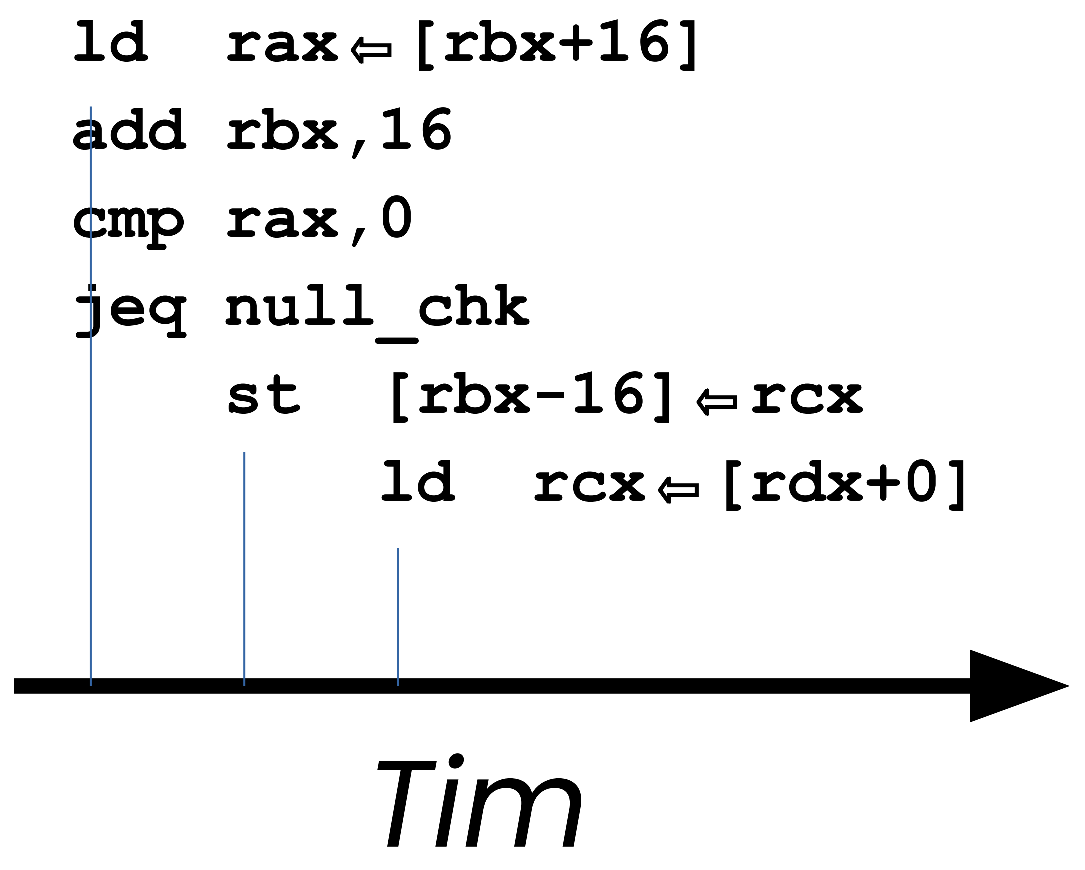
A BIGGER EXAMPLE

```
ld  rax⇦[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
st  [rbx-16]⇦rcx
ld  rcx⇦[rdx+0]
ld  rax⇦[rax+8]
...
```

Unrelated cache miss!

Misses now overlap

L/S unit busy again



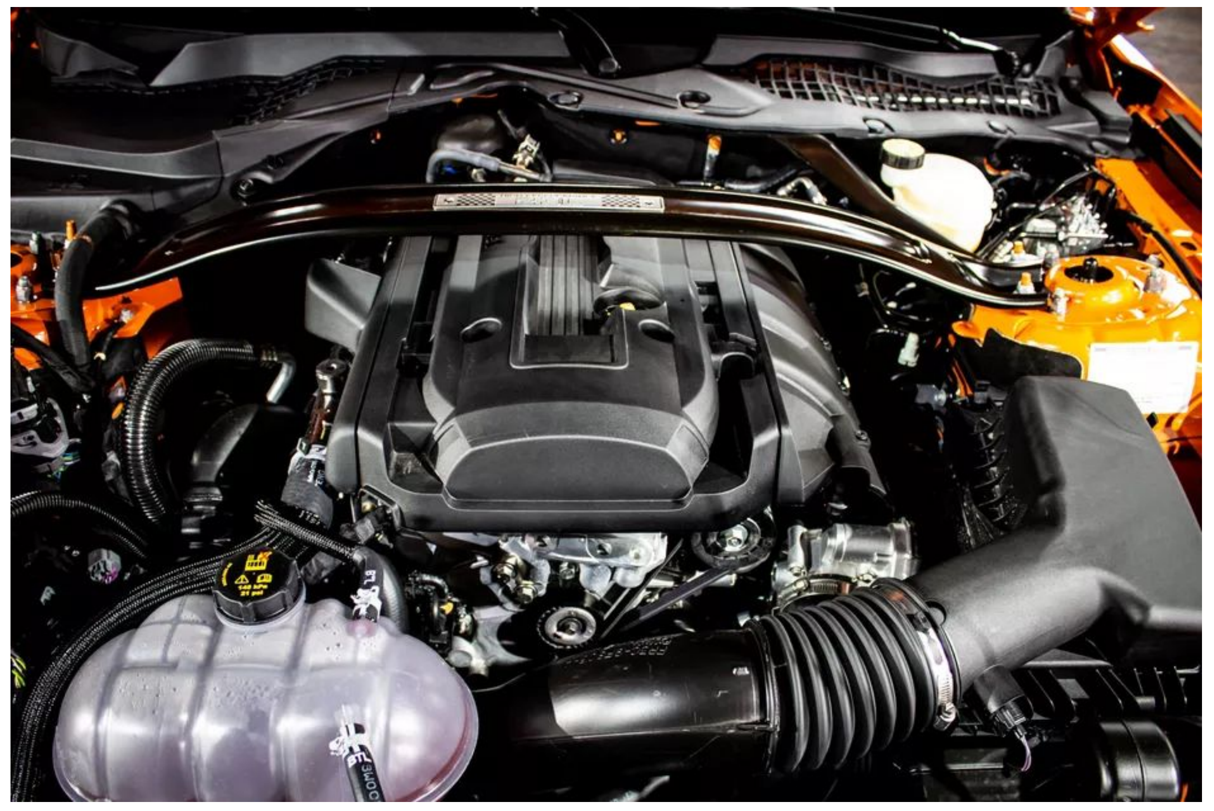
```
ld  rax⇦[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
    st  [rbx-16]⇦rcx
        ld  rcx⇦[rdx+0]
```

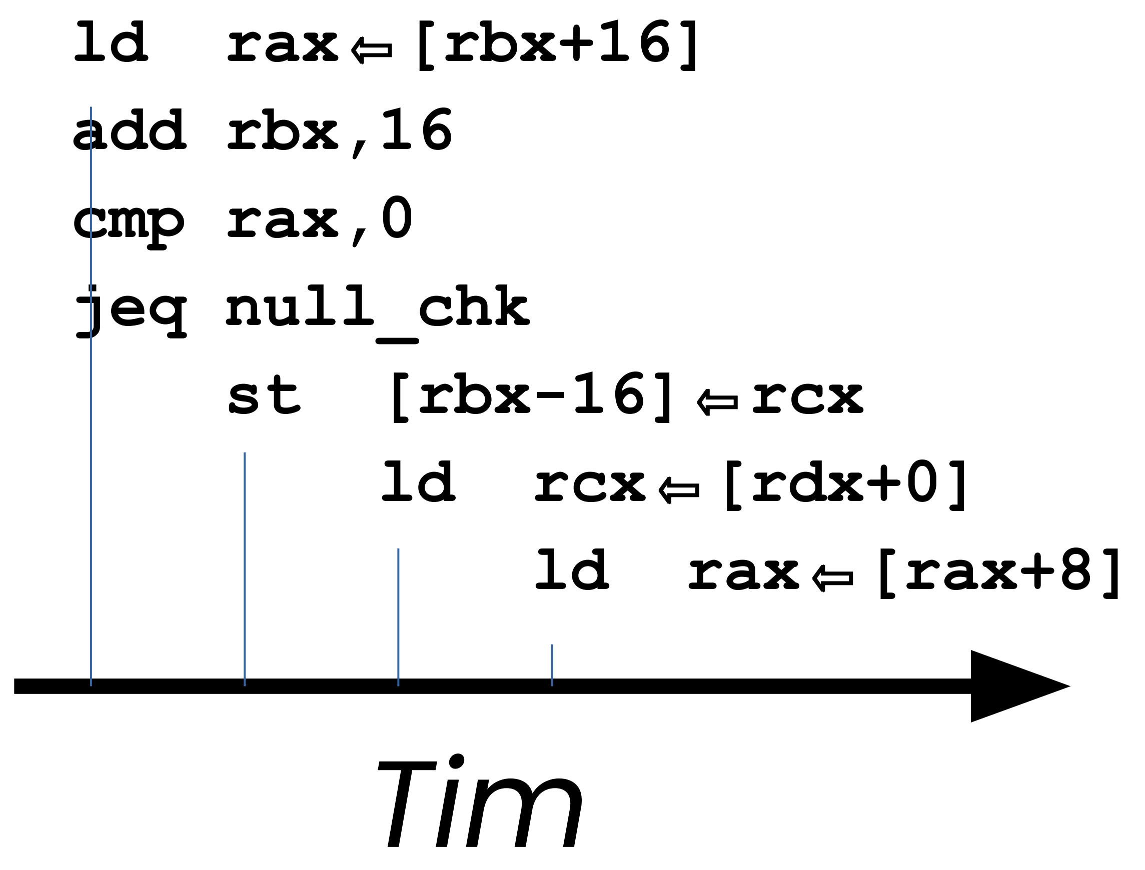
A BIGGER EXAMPLE

```
ld   rax⇐[rbx+16]
add  rbx,16
cmp  rax,0
jeq  null_chk
st   [rbx-16]⇐rcx
ld   rcx⇐[rdx+0]
ld   rax⇐[rax+8]
...
```

rax still not ready

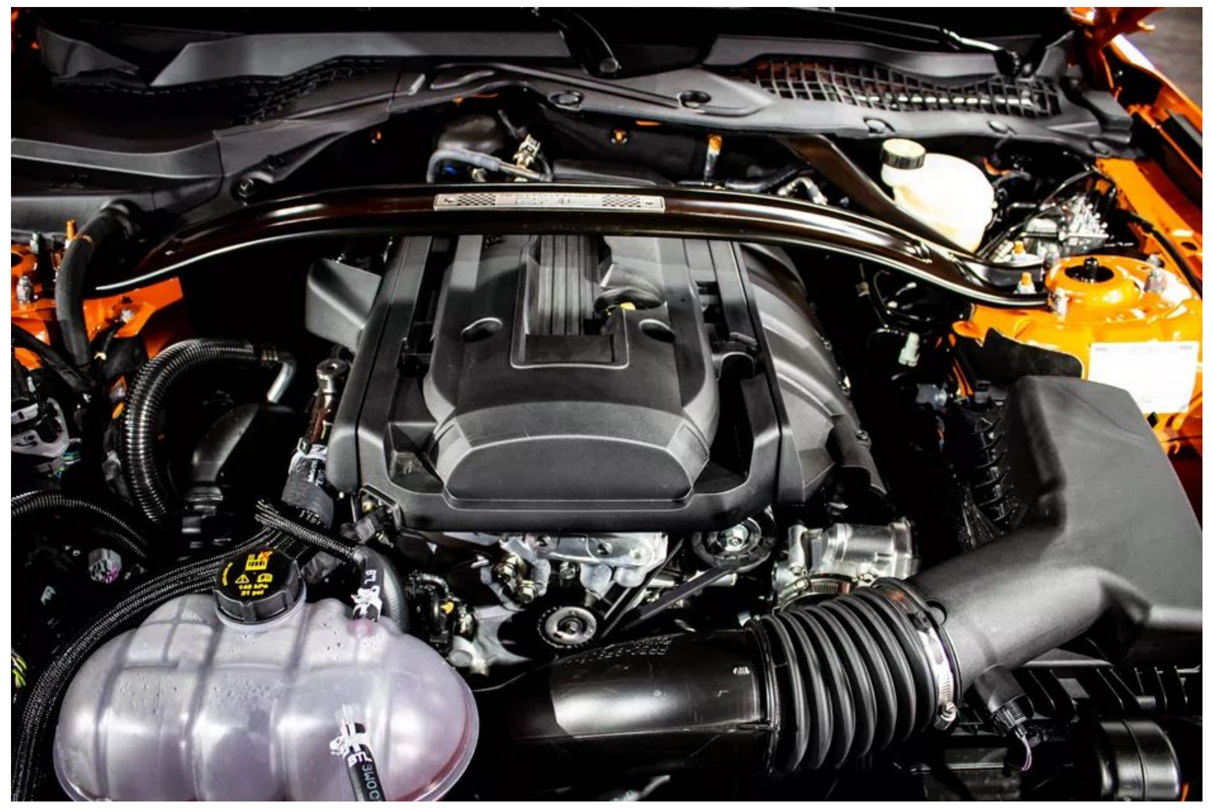
Load cannot start




```
ld   rax⇐[rbx+16]
add  rbx,16
cmp  rax,0
jeq  null_chk
     st   [rbx-16]⇐rcx
          ld   rcx⇐[rdx+0]
               ld   rax⇐[rax+8]
```

A BIGGER EXAMPLE

```
ld   rax⇦[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
st   [rbx-16]⇦rcx
ld   rcx⇦[rdx+0]
ld   rax⇦[rax+8]
...
```



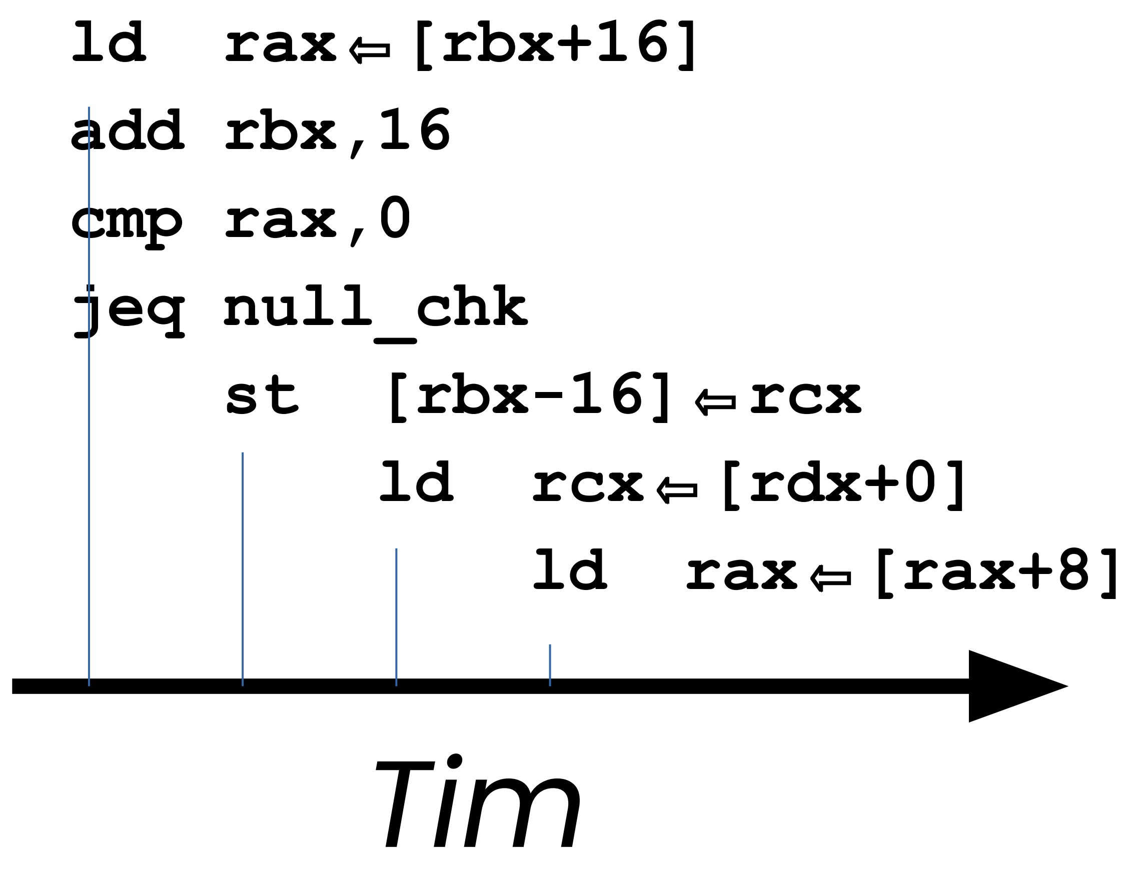
In 4 clocks started 7 ops

And 2 cache misses

Misses return in clocks 300 and 302

So 7 ops in 302 cycles

Misses totally dominate



```
ld   rax⇦[rbx+16]
add rbx,16
cmp rax,0
jeq null_chk
    st   [rbx-16]⇦rcx
         ld   rcx⇦[rdx+0]
              ld   rax⇦[rax+8]
```

WHAT JUST HAPPENED?

ILP is mined out – minor future improvements
- Deep pipelining, wider issue, more functional units,
giant re-order buffers, 100’s of instructions in flight
- Super complex, but at the limits

CPU Chips are already **95% cache**, more cache is not helping
- Miss rates are really low
- But a miss costs 300clocks x 4issue= ~1200 instructions
- So a 5% miss rate dominates

Time to rethink our performance model!

THINK DATA NOT CODE

Indirection layers cost more than you think:
- Each round-trip to memory costs more than 100’s of instructions

Data copies cost double:
- Each copy takes precious cache space
- And requires a round-trip to memory

Network byte buffer → JSON String → DOM
- If you must copy, make sure the cost pays off
- Copy-then-use-once rarely worth it

Data Structure choices matter more than most code!

MEMORY IS EXPENSIVE AND EVERYTHING ELSE IS CHEAP

Out-of-Memory-Bandwidth does not show in most profilers
- And yet removing data copies can lead to **integer factor speedups**
- Needs hardware performance counters to spot the problem

GC **new** allocation **always** misses in cache
- And thus uses lots of memory bandwidth
- Sometimes object pooling wins over GC

Performance problems are almost always solved with a holistic view

Start by looking at your **data structure choices!**