

High Performance From Understanding The Low Levels

Dr. Cliff Click

rocketrealtime.com

cliffc@acm.org

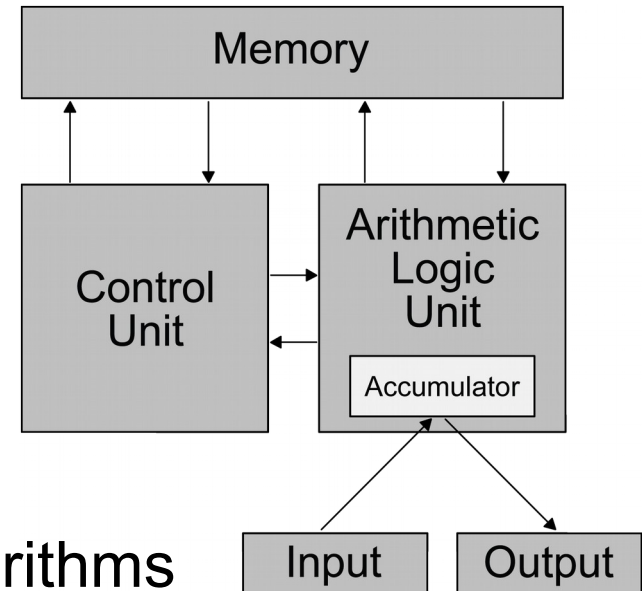
cliffc.org/blog

Agenda

- Introduction
- **Serial Performance: Its All About Memory**
 - The Quest for Single-Core Speed is Over
 - Example 1: Indirection: LinkedList vs ArrayList
 - Example 2: Big Data vs Bandwidth
- Parallel Memory Behavior:
 - A Data Race
 - Specter and Meltdown
 - Java Memory Model
- Q&A & Closing Comments

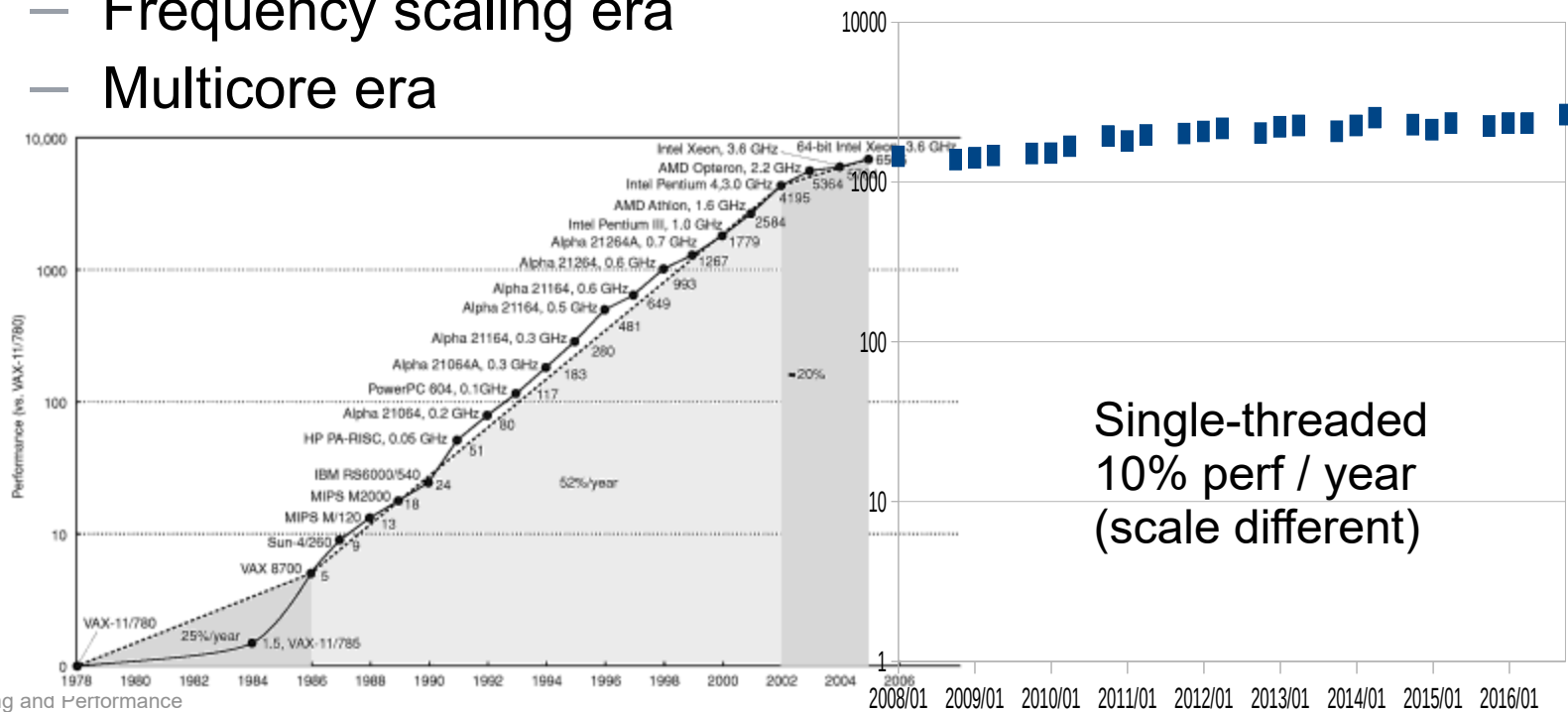
The Von Neumann Machine Architecture

- Characteristics of the von Neumann architecture include
 - Program is stored in memory
 - Memory is shared between program and data
 - *Sequential execution model*
- This is a great model for designing algorithms
 - But it's not how computers really work today!
 - At one point this described real computers
 - Now it is a useful abstraction for computation
 - Like all abstractions, we should understand its limitations



CPU Performance

- Graph shows CPU performance over time
 - Log scale, normalized to VAX-11/780 performance
- Can divide graph into three distinct phases
 - CISC era
 - Frequency scaling era
 - Multicore era



Hitting the wall

- Serial performance hit the wall a decade ago
 - Power Wall
 - Higher freq → more power → more heat → chip melts!
 - ILP Wall
 - Hitting limits in 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 signal to propagate across a complex CPU!

The Multicore era, 2002- ?

- Clock rates have been basically flat for 15 years
 - Getting more expensive to build faster processors
 - Instead we put more cores on a chip
- Moore's law: **more** cores, but **not faster** cores
 - Core counts still increasing rapidly
- Challenges for programmers
 - How are we going to use those cores?
 - Adjusting our mental performance models?

Agenda

- Introduction
- Serial Memory Performance:
 - **The Quest for Single-Core Speed is Over**
 - Example 1: Indirection: LinkedList vs ArrayList
 - Example 2: Big Data vs Bandwidth
- Parallel Memory Behavior:
 - A Data Race
 - Specter and Meltdown
 - Java Memory Model
- Q&A & Closing Comments



The Quest for ILP

- ILP = Instruction Level Parallelism
- Faster CPUs at the same clock rate
 - Pipelining
 - Branch Prediction
 - Wide-issue
 - Speculative execution
 - Out-Of-Order (O-O-O) execution
 - Hit-Under-Miss cache, no-lockup cache
 - Prefetching



The Quest for ILP: Pipelining

rocketrealtime.com

- Internally, each instruction has multiple stages
 - Many of which must be done sequentially
 - Fetching the instruction from memory
 - Also identifying the end of the instruction (update PC)
 - Decoding the instruction
 - Fetching needed operands (memory or register)
 - Performing the operation (e.g., addition)
 - Writing the result somewhere (memory or register)
 - Each instruction takes more than one clock cycle
 - But stages of different instructions can overlap
 - While decoding instruction N, fetch instruction N+1

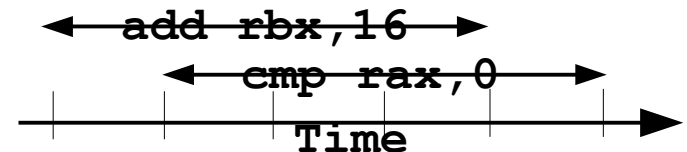


Pipelining

```
add    rbx, 16
cmp     rax, 0
```

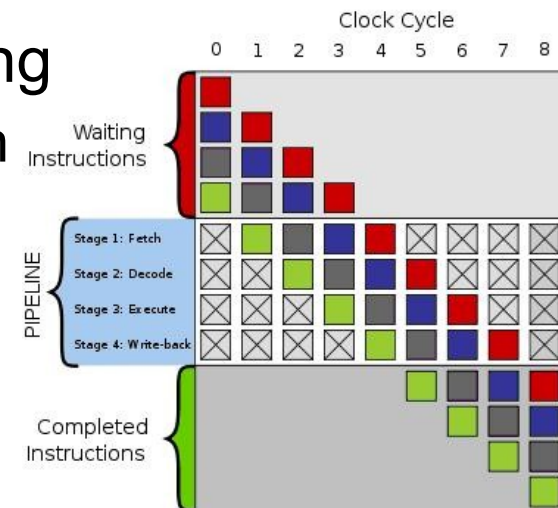
add 16 to register RBX
then compare RAX to 0

- On early machines, these ops would be e.g. 4 clks
- *Pipelining* allows them to appear as 1 clk
 - And allows a much higher clock rate
 - Much of the execution is parallelized in the *pipe*
- Found on all modern CPUs



Pipelining

- Pipelining improves throughput, but not latency
 - The deeper the pipeline, the higher the (theoretical) multiplier for effective CPI
- "Single cycle execution" is a misnomer
 - All instructions take multiple cycles end-to-end
 - Pipelining can reduce CPI to 1 (in theory)
- RISC ISAs are designed for easier pipelining
 - Instruction size is uniform, simplifying fetch
 - No memory-to-memory ops
 - Some ops not pipelined (e.g. div)



Pipelining hazards

- Pipelining attempts to impose parallelism on sequential control flows
- This may fail to work if:
 - There are conflicts over CPU resources
 - There are data conflicts between instructions
 - Instruction fetch is not able to identify the next PC
 - For example, because of branches
- Hazards can cause pipeline *stalls*
 - In the worst case, a branch could cause a complete pipeline *flush*



Loads & Caches

```
ld    rax ← [rbx+16]
```

Loads RAX from memory

- Loads read from cache, then memory
 - Cache hitting loads take 2-3 clks
 - Cache misses to memory take 200-300 clks
 - Can be many cache levels; lots of variation in clks
- Key theme: value in RAX might not be available for a long long time
- Simplest CPUs *stall* execution until value is ready
 - e.g. Typical GPU

Loads & Caches

```
ld    rax ← [rbx+16]
```

```
...
```

```
cmp   rax, 0
```

RAX still not available

- Commonly, execution continues until RAX is used
 - Allows useful work in the miss “shadow”
- True data-dependence stalls in-order execution
- Also Load/Store Unit resources are tied up
- Fairly common
 - Many embedded CPUs, Azul, Sparc, Power

Branch Prediction

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

No RAX yet, so no flags
Branch not resolved
...speculative execution

- Flags not available so branch *predicts*
 - Execution past branch is *speculative*
 - If wrong, pay *mispredict penalty* to clean up mess
 - If right, execution does not stall
 - Right > 95% of time

Multiple issue

- Modern CPUs are designed to issue multiple instructions on each clock cycle
 - Called *multiple-issue* or *superscalar execution*
 - Offers possibility for $CPI < 1$
 - Subject to all the same constraints (data contention, branch misprediction)
 - Requires even more speculative execution

Dual-Issue or Wide-Issue

```
add    rbx, 16  
cmp    rax, 0
```

add 16 to register RBX
then compare RAX to 0

- Can be *dual-issued* or *wide-issued*
 - Same 1 clk for both ops
 - Must read & write unrelated registers
 - Or not use 2 of the same resource
- Dual issue is a common CPU feature
 - Not found on simplest embedded cpus

Register Renaming, Speculation, O-O-O

- Register renaming, branch prediction, speculation, O-O-O are all *synergistic*
 - Speculative state kept in extra renamed registers
 - On mis-predict, toss renamed registers
 - Revert to original register contents, still hanging around
 - Like rolling back a transaction
 - On correct-predict, rename the extra registers
 - As the “real” registers
- Allows more execution past cache misses
 - Old goal: just run more instructions
 - New goal: run until can start the *next* cache miss

X86 O-O-O Dispatch Example

rocketrealtime.com

```
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]
```

X86 O-O-O Dispatch 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 cache miss -
300 cycles to load
Instruction starts and
dispatch continues...

Clock 0 – instruction 0

X86 O-O-O Dispatch 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]
```

Next op writes **RBX** -
which is read by prior op
Register-renaming allows
parallel dispatch

Clock 0 – instruction 1

X86 O-O-O Dispatch 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 available yet -
cannot compute **flags**
Queues up behind load

Clock 0 – instruction 2

X86 O-O-O Dispatch Example

rocketrealtime.com

```
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 still not ready

branch prediction -

speculates not-taken

Limit of 4-wide dispatch -
next op starts new clock

Clock 0 – instruction 3

X86 O-O-O Dispatch Example

rocketrealtime.com

```
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
Result kept in store buffer
Also RBX might be null
L/S used, no more mem
ops this cycle

Clock 1 – instruction 4

X86 O-O-O Dispatch 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

Clock 2 – instruction 5

X86 O-O-O Dispatch Example

rocketrealtime.com

```
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 till
1st load returns

Clock 3 – instruction 6

X86 O-O-O Dispatch Summary

rocketrealtime.com

```
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 clks started 7 ops
- And 2 cache misses
- Misses return in cycle 300 and 302.
- So 7 ops in 302 cycles
- Misses totally dominate performance

The Quest for ILP

- X86: a Grand Effort to mine *dynamic* ILP
 - Incremental addition of performance hacks
- Deep pipelining, ever wider-issue, parallel dispatch, giant re-order buffers, lots of functional units, 128 instructions “in flight”, etc
- Limited by cache misses and branch mispredict
 - Both miss rates are really low now
 - But a miss costs 100-1000 instruction issue slots
 - So a ~5% miss rate dominates performance



How did this turn out?

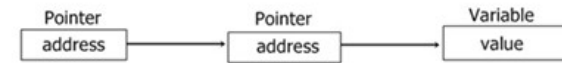
- ILP is mined out
 - As CPUs get more complicated, more transistors are thrown at dealing with the hazards of ILP
 - Like speculative execution
 - Instead of providing more computational power
 - Moore's law gives us a growing transistor budget
 - But we spend more and more on ILP hazards
- Contrast to GPUs
 - Zillions of simple cores
 - But only works well on narrow problem domain

Agenda

- Introduction
- Serial Memory Performance:
 - The Quest for Single-Core Speed is Over
 - **Example 1: Indirection: LinkedList vs ArrayList**
 - Example 2: Big Data vs Bandwidth
- Parallel Memory Behavior:
 - A Data Race
 - Specter and Meltdown
 - Java Memory Model
- Q&A & Closing Comments



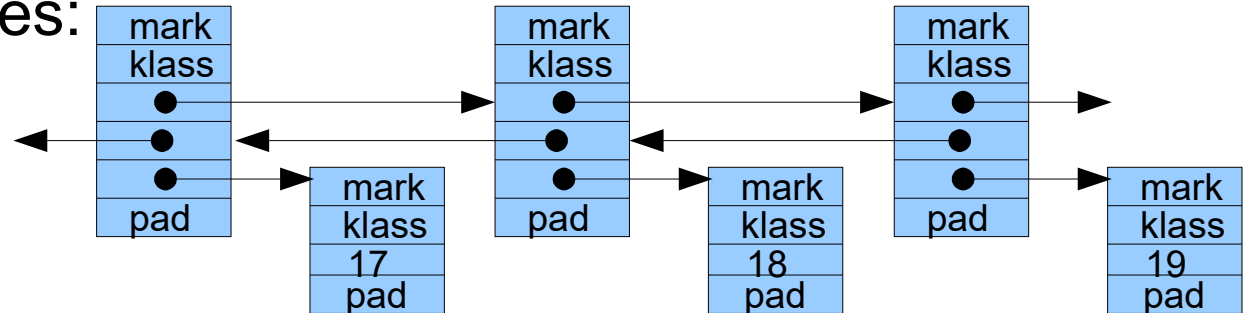
Single Indirection



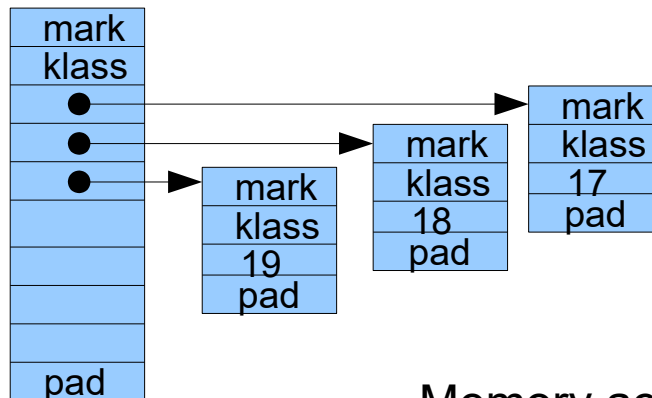
Multiple Indirection

LinkedList vs ArrayList

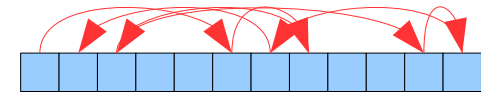
- LinkedList Nodes:



- ArrayList:

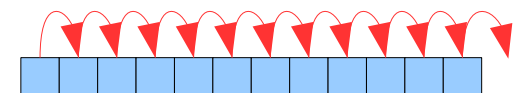


Memory access order:



memory →

Memory access order:



memory →

LinkedList vs ArrayList

- For “normal” operations on collections of size:
 - < 100 – ArrayList is 1.5x faster
 - < 1000 – ArrayList is 3x faster
 - < 10000 – ArrayList is 4.5x faster
 - > L3 cache – ArrayList is 2.5x faster
- Why?
 - LinkedList requires a pointer indirection
 - And a dependent load... see last section
 - And more memory: you run out of cache faster

DEMO: Example Lists.java and ListPrim.java

The Tension with Indirection

- “All problems in computer science can be solved by **another** level of indirection” - David Wheeler
- “All performance problems in computer science can be solved by **removing** a level of indirection” - Unknown???
- Indirection is a key programming concept
 - We use it every day
- Indirection directly “fights” the laws of chip physics
- Which we mostly Don’t Care until the problem gets big
- And then we need a Data Structure Change

Data Has Weight

- Changing a common data structure is one of the harder restructuring tasks... be it DB tables or JSON or Arrays
- Think Before Selecting:
 - What are the common use-cases? Common ops?
 - Mostly write/archive? Mostly read? Caching? Bulk? Lots-of-small? Few Big? Rate-of-change?
 - Whats the typical size? Max size? Typical load spike?
 - Access: Frequent and fine-grained? Bulk & compressed?
- IMHO: Choosing the right Data Structure carries more long-term benefits than getting the initial code right...

Agenda

- Introduction
- Serial Memory Performance:
 - The Quest for Single-Core Speed is Over
 - Example 1: Indirection: LinkedList vs ArrayList
 - **Example 2: Big Data vs Bandwidth**
- Parallel Memory Behavior:
 - A Data Race
 - Specter and Meltdown
 - Java Memory Model
- Q&A & Closing Comments



Big Data and Bandwidth

- Like caches, you only got so much Bandwidth
 - Use it wisely!
- Here “Big Data” means “much much bigger than cache”
- Touching the data means reading it through the mem bus
- WRITING it means sending it back out the mem bus, and READING it again later
 - And many times the WRITES are hidden...

DEMO: Example Bandwidth.java

Big Data and Passes

rocketrealtime.com

- Big Data is NOT in Cache
- Every Pass reads it **all** from slow Memory
- Math & Analytics are much much cheaper
 - Rollups & summaries & “BI” math are tiny – fit in cache
- Design Point: Fewer “Bigger” Passes
 - Expensive to “get” the Data in the CPU
 - Do as much as you can with Data once you got it!
 - e.g. Loop-Fusion, Stream-Fusion, 1-pass algorithms
 - “Drink the Big Data” through the tiny straw only once!



Cost of Allocation Per-Big-Data

- What else is in that pipe?
 - `readline()` a **new String** plus **split**
- “**new**”: New memory, not in cache!
 - Read memory, then overwrite in cache
 - Write used objects back out
 - “readline” is `new String()`, `char[]`, `byte`→`char` plus headers
 - String: $2 \text{ hdrs} + \text{ptr} + \text{hash} + \text{pad} = 32$
 - `char[]`: $2 \text{ hdrs} + \text{len} = 20$
- “**split**”: Same again, but with a lot more headers
 - ~65 Strings plus a `String[]`: $(65 \times 52) + (20 + 65 \times 8)$
- 14.5:1 ratio of “junk” to “data”

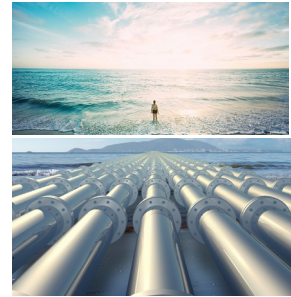
abc x1

newnew x2
oldold x2

headers x.085

headers x6.45

newnew x2
oldold x2



Big Data and “new”

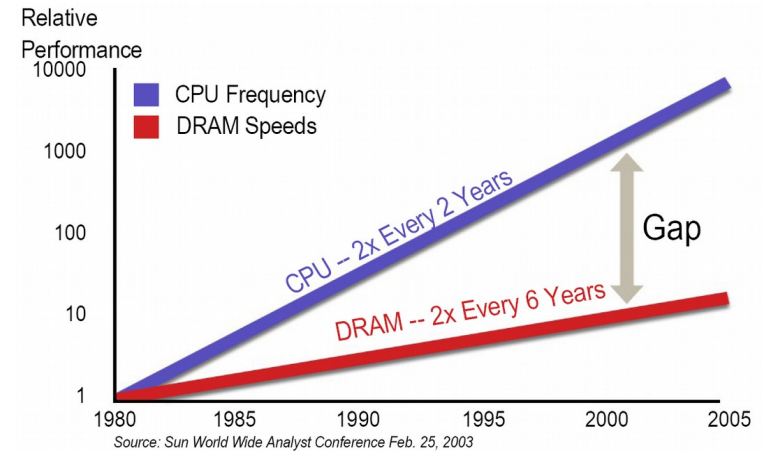
- Doing a “new” for every “X” when “X” is Big
 - Here we copied the data from nio ByteBuffer to String
 - String is local, dead, cheap GC’d... BUT
 - Costs memory bandwidth to read the dead String, and write the dead String: 2 bytes read and written, per byte
 - Then did a String split()
 - Which copied it all again (minus the “,”)
 - Plus String headers; probably another 6bytes per byte
- End result: for every byte read, an additional 14.5 bytes
 - 1/14th bandwidth for Big Data, 13/14th for junk Strings
 - Remove junk allocation gets an easy 5x win

Agenda

- Introduction
- Serial Memory Performance:
 - The Quest for Single-Core Speed is Over
 - Example 1: Indirection: LinkedList vs ArrayList
 - Example 2: Big Data vs Bandwidth
- **Parallel Memory Behavior:**
 - A Data Race
 - Specter and Meltdown
 - Java Memory Model
- Q&A & Closing Comments

Memory subsystem performance

- Chart shows speedup in CPU vs memory
 - Exponentially widening gap
- In older CPUs, memory access was only slightly slower than register fetch
- Today, fetching from main memory takes several hundred clock cycles
 - Modern CPUs use sophisticated multilevel memory caches
 - And cache misses still dominate performance



Types of memory

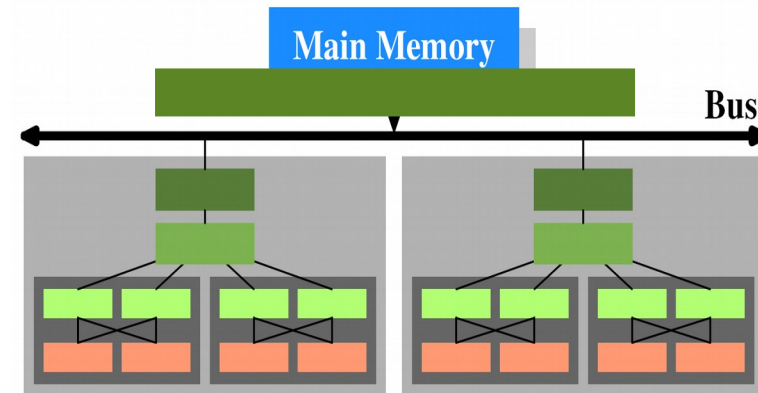
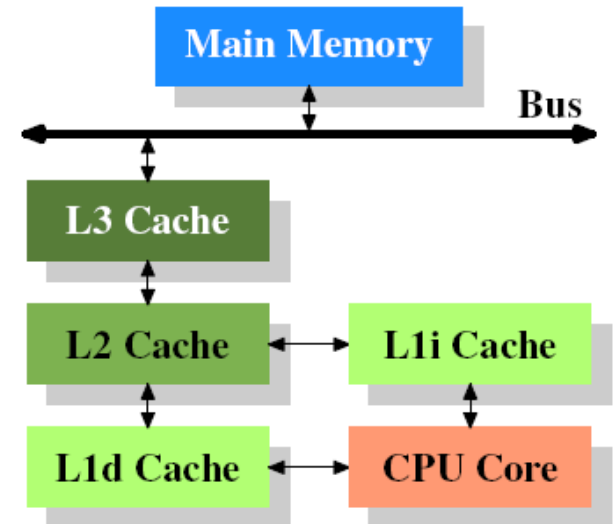
- Static RAM (SRAM) – fast but expensive
 - Six transistors per bit
- Dynamic RAM (DRAM) – cheap but slow
 - One transistor + one capacitor per bit
 - Improvements in DRAM (DDR, DDR2, DDR4, etc) improve bandwidth but latency not so much
 - More improvements in power & density than speed

Caching

- Adding small amounts of faster SRAM can really improve memory performance
 - Caching works because programs exhibit both code and data locality (in both time & space)
 - Typically have separate instruction and data caches
 - Code and data each have their own locality
- Moves the data closer to the CPU
 - Speed of light counts!
 - Major component of memory latency is wire delay

Caching

- As the CPU-memory speed gap widens, need more cache layers
 - Relative access speeds
 - Register: <1 clk
 - L1: ~3 clks
 - L2: ~15 clks
 - Main memory: ~300 clks
- On multicore systems, lowest cache layer is shared
 - But not all caches visible to all cores



Caching

- With high memory latency, ILP doesn't help
 - In the old days, loads were cheap and multiplies / FP ops were expensive
 - Now, multiplies are cheap but loads expensive!
- With a large gap between CPU and memory speed, cache misses dominate performance
- **Memory is the new disk!**

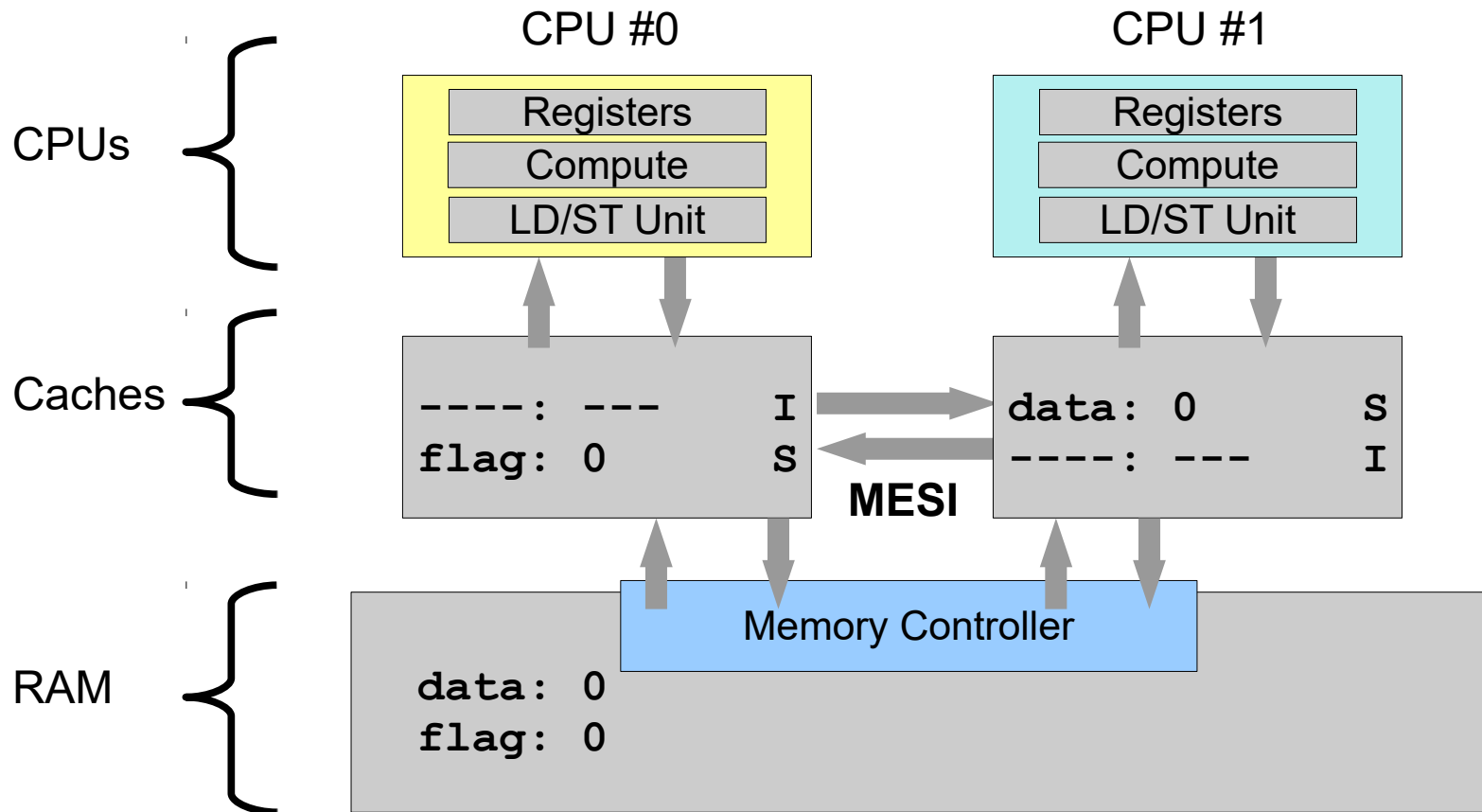
In search of faster memory access

- To make memory access cheaper
 - Relax coherency constraints
 - Improves throughput, not latency
 - Is this theme sounding familiar yet?
- More complex programming model
 - Must use synchronization to identify shared data
- Weird things can happen
 - Stale reads
 - Order of execution is
relative to the observing CPU (thread)

Agenda

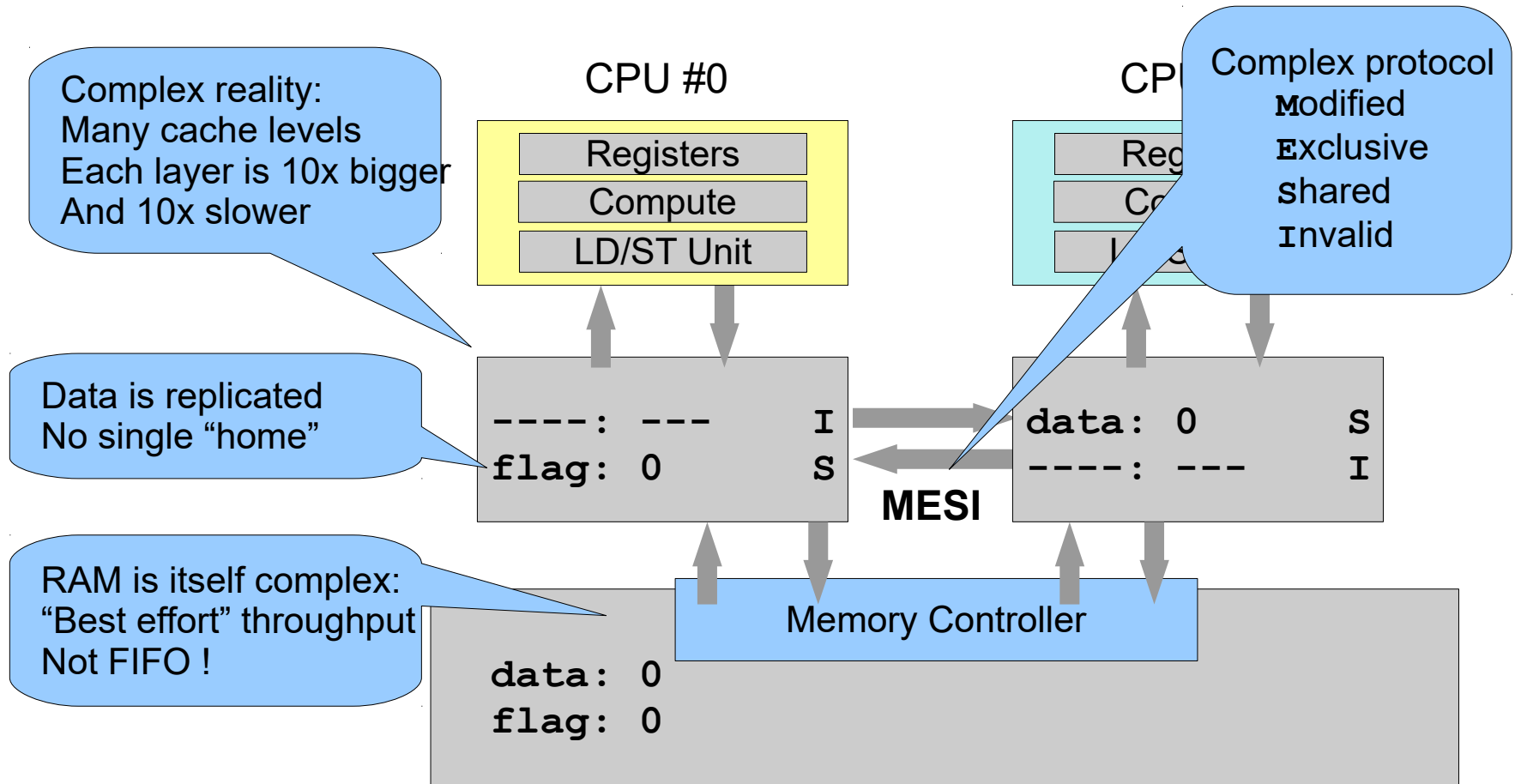
- Introduction
- Serial Memory Performance:
 - The Quest for Single-Core Speed is Over
 - Example 1: Indirection: LinkedList vs ArrayList
 - Example 2: Big Data vs Bandwidth
- Parallel Memory Behavior:
 - **A Data Race**
 - Specter and Meltdown
 - Java Memory Model
- Q&A & Closing Comments

Real Chips Reorder Stuff



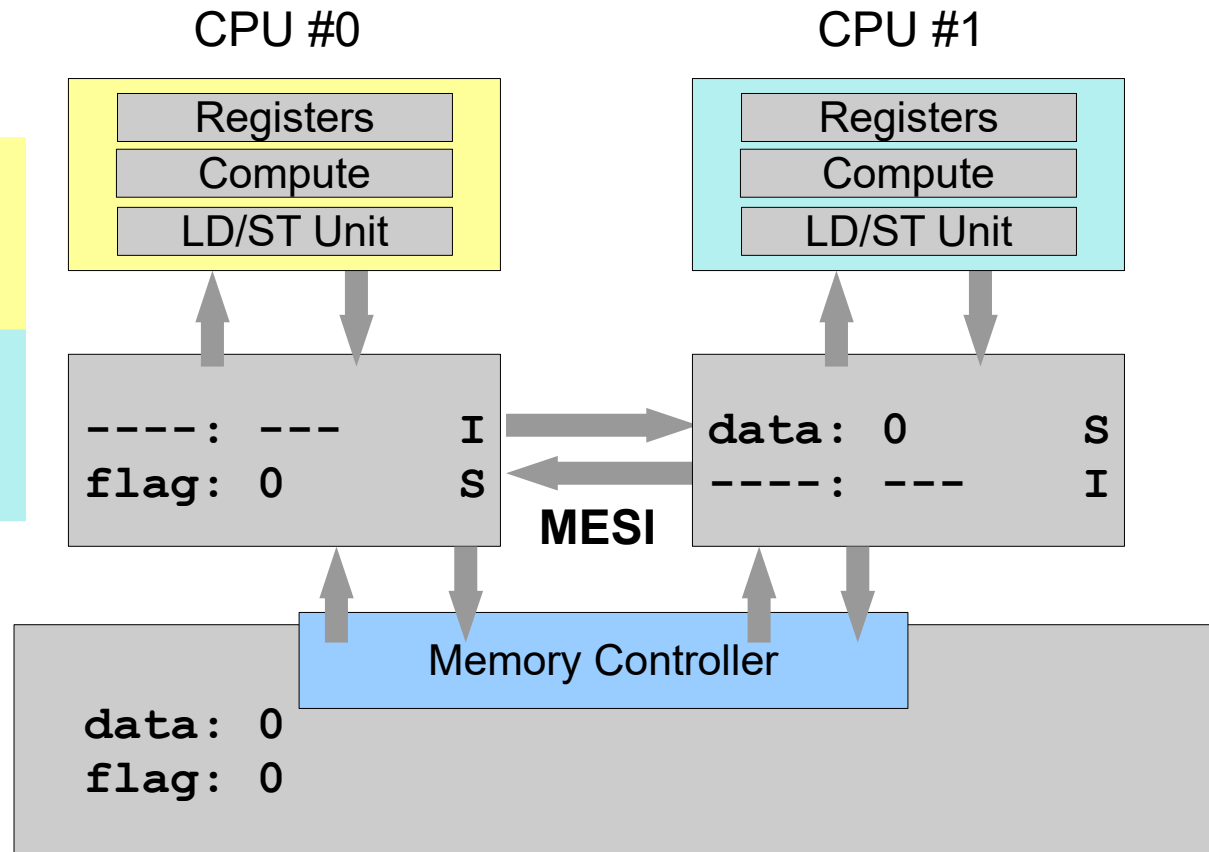
Real Chips Reorder Stuff

rocketrealtime.com



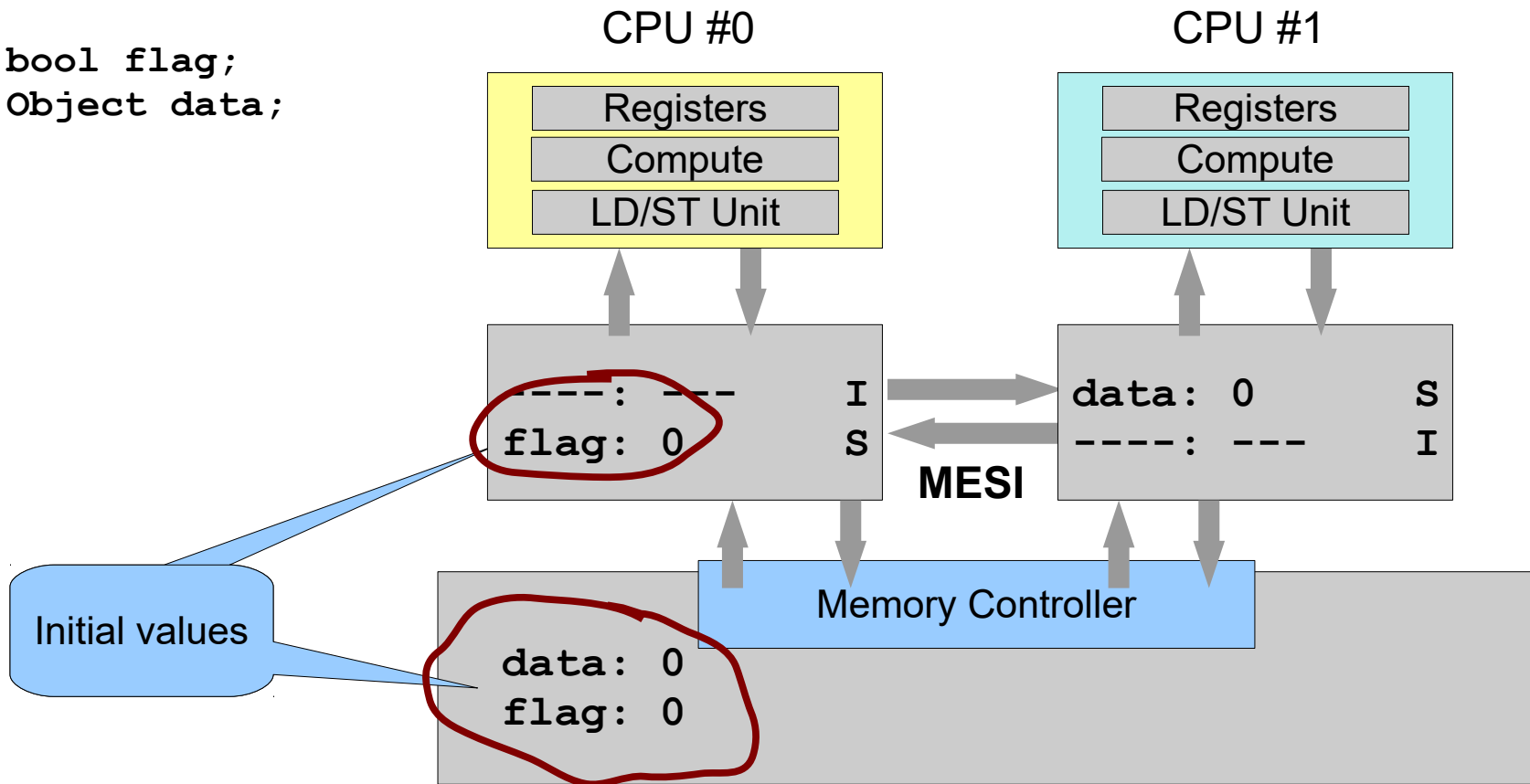
Real Chips Reorder Stuff

```
bool flag;  
Object data;  
init() {  
    data = ...;  
    flag = true;  
}  
Object read() {  
    if( !flag ) ...;  
    return data;  
}
```

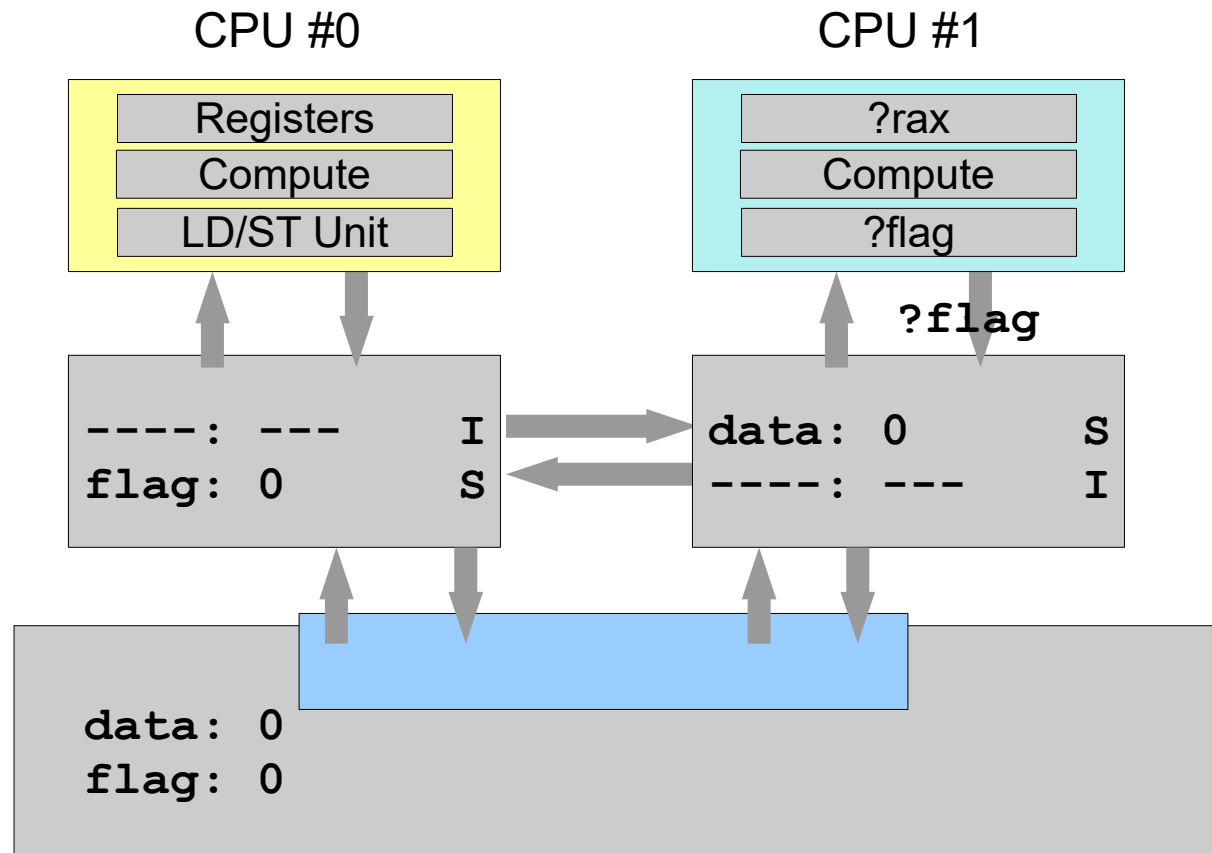
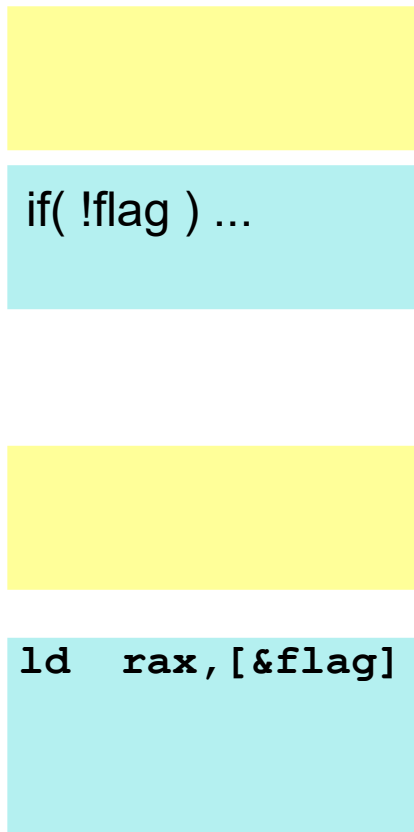


Real Chips Reorder Stuff

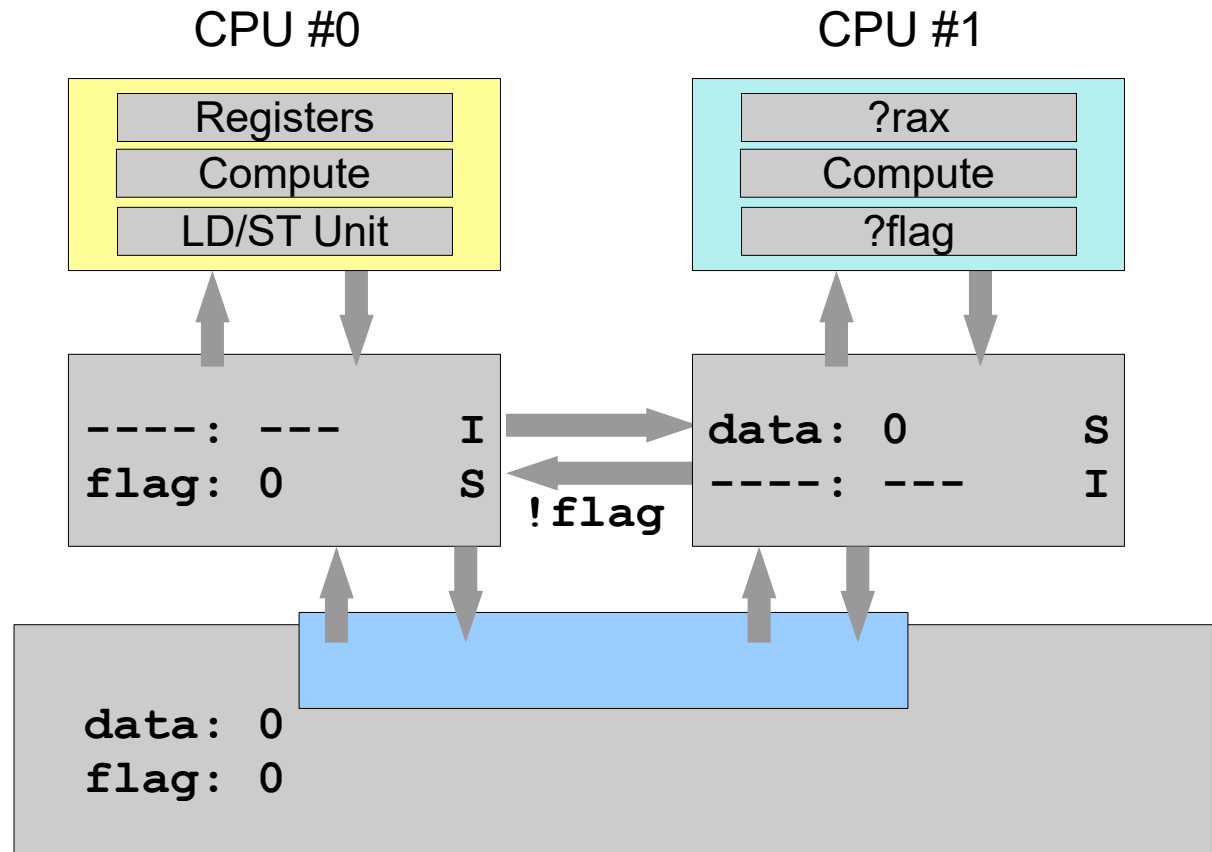
```
bool flag;  
Object data;
```



Real Chips Reorder Stuff



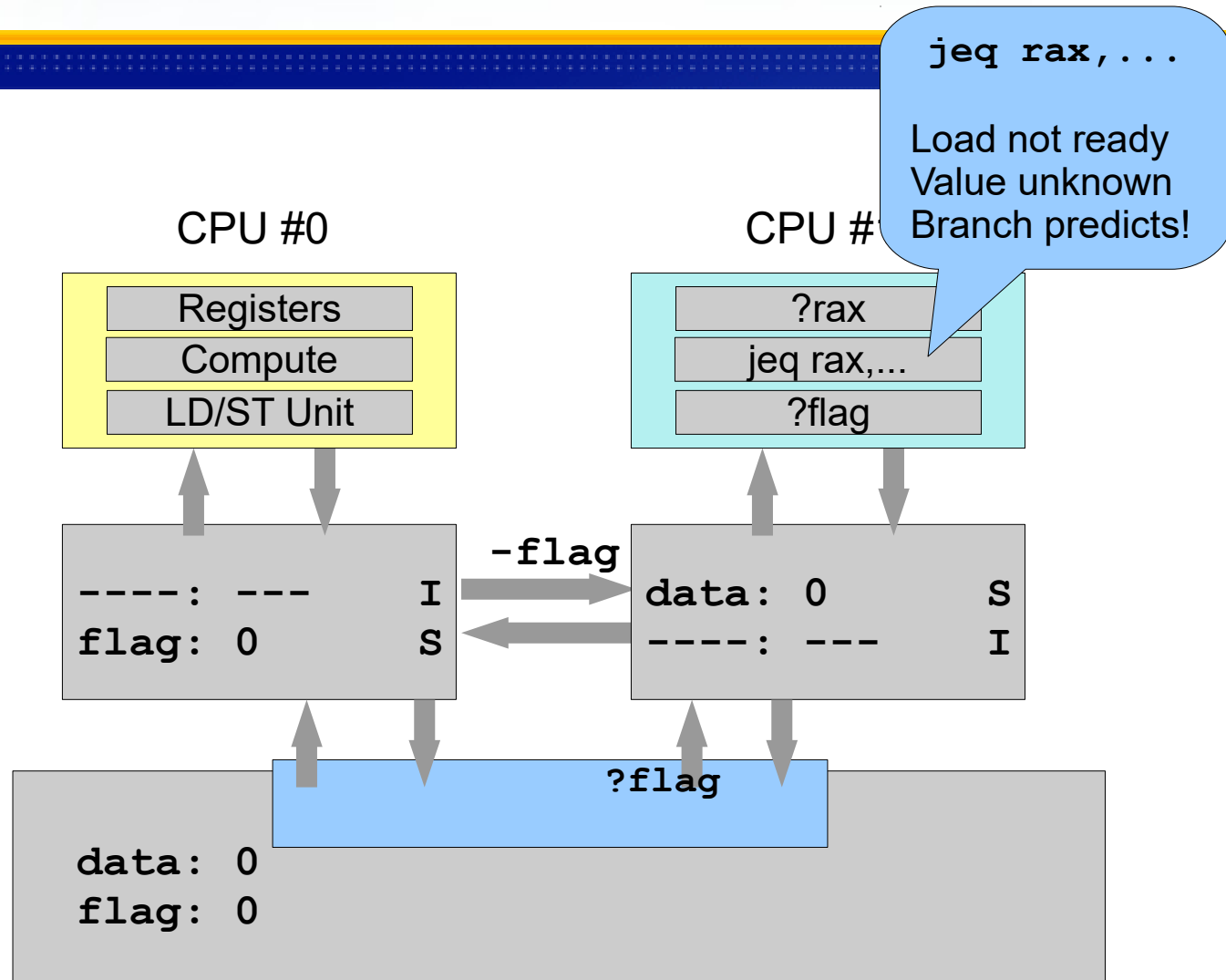
Real Chips Reorder Stuff



Real Chips Reorder Stuff

if(!flag) ...

~~ld rax, [&flag]~~
jeq rax, ...



Real Chips Reorder Stuff

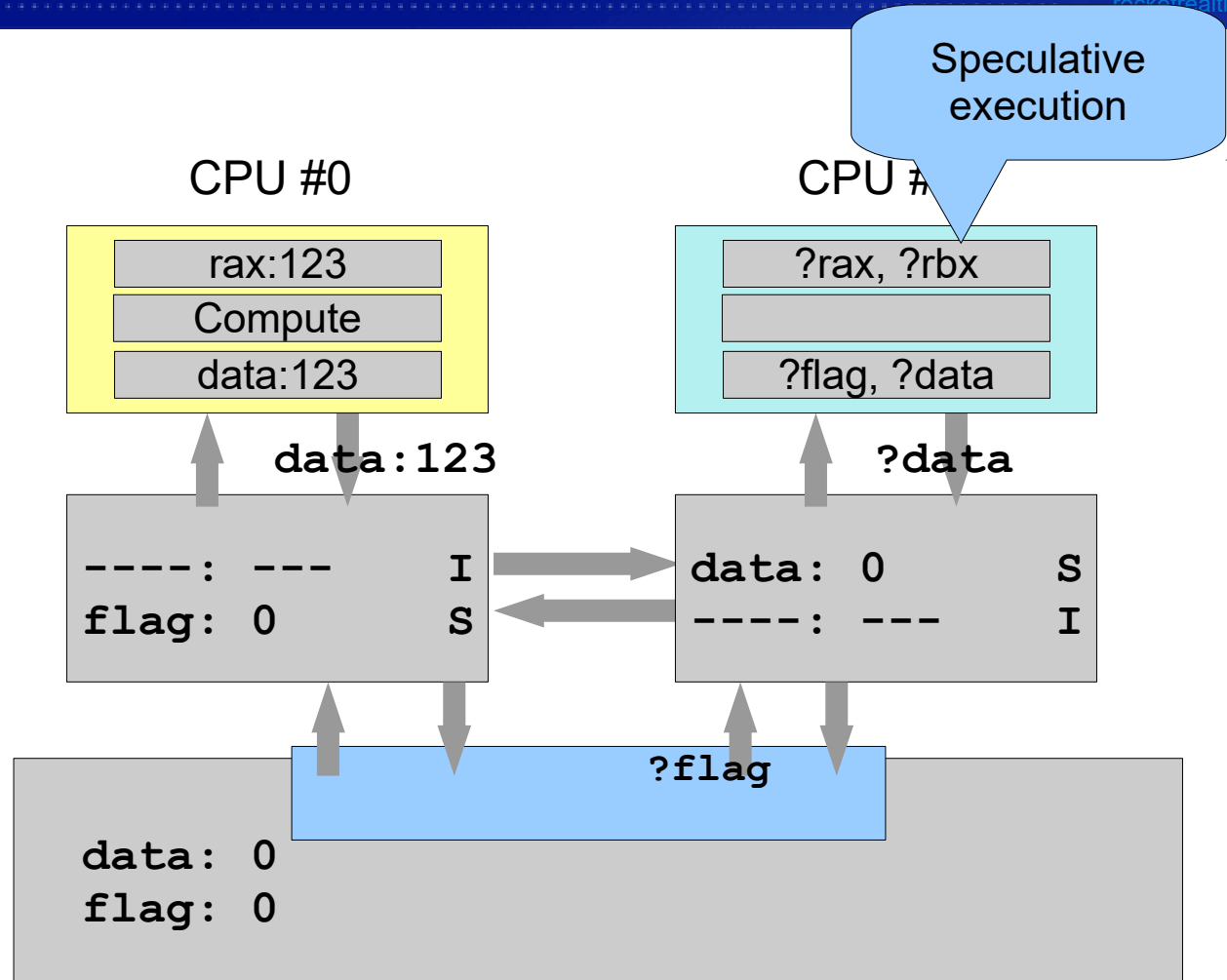
rocketrealtime.com

```
data = ...;
```

```
if( !flag ) ...  
return data;
```

```
mov rax,123  
st  [&data],rax
```

```
ld rax,[&flag]  
jeq rax,...  
ld  rbx,[&data]
```



Real Chips Reorder Stuff

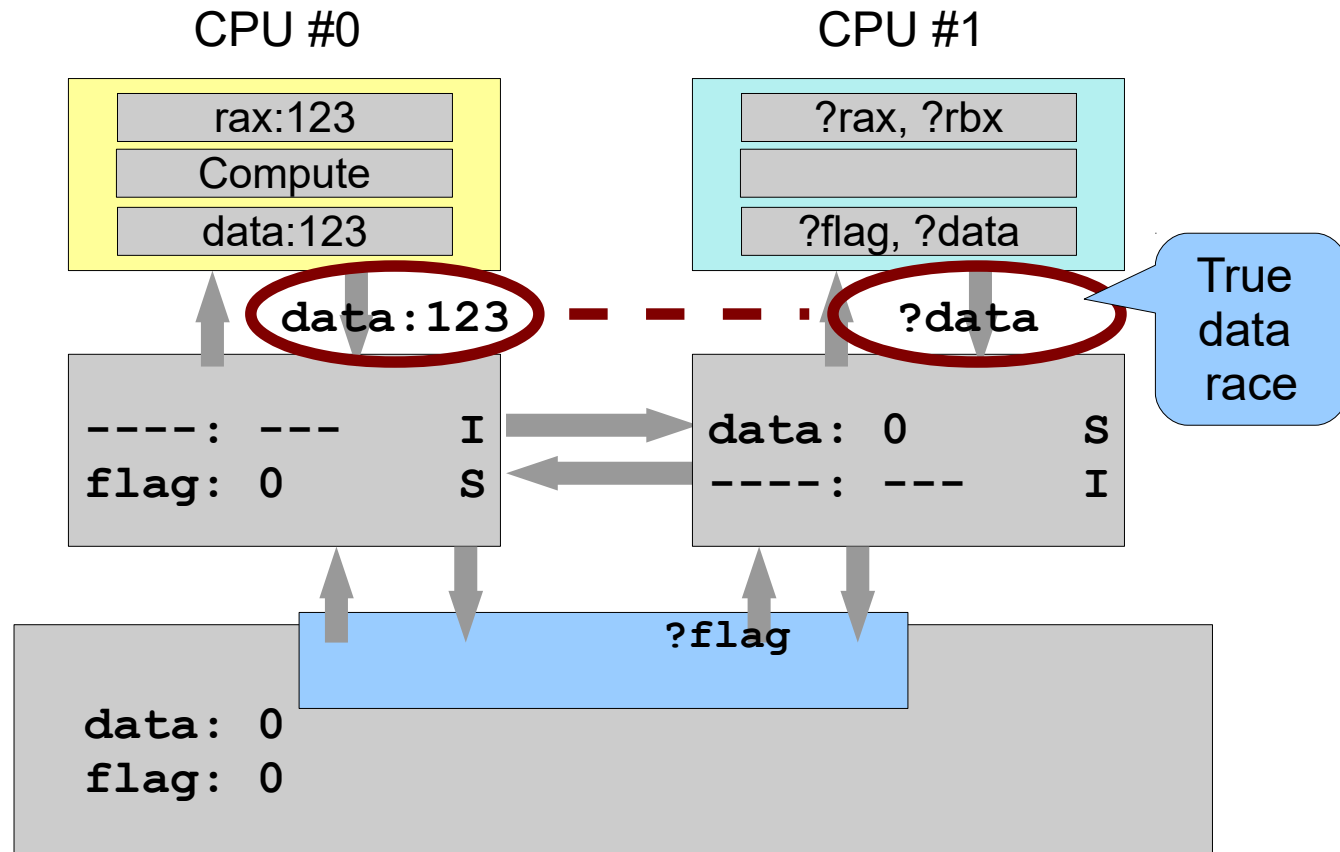
rocketrealtime.com

```
data = ...;
```

```
if( !flag ) ...  
return data;
```

```
mov rax,123  
st  [&data],rax
```

```
ld rax,&flag  
jeq rax,...  
ld  rbx,&data
```



Real Chips Reorder Stuff

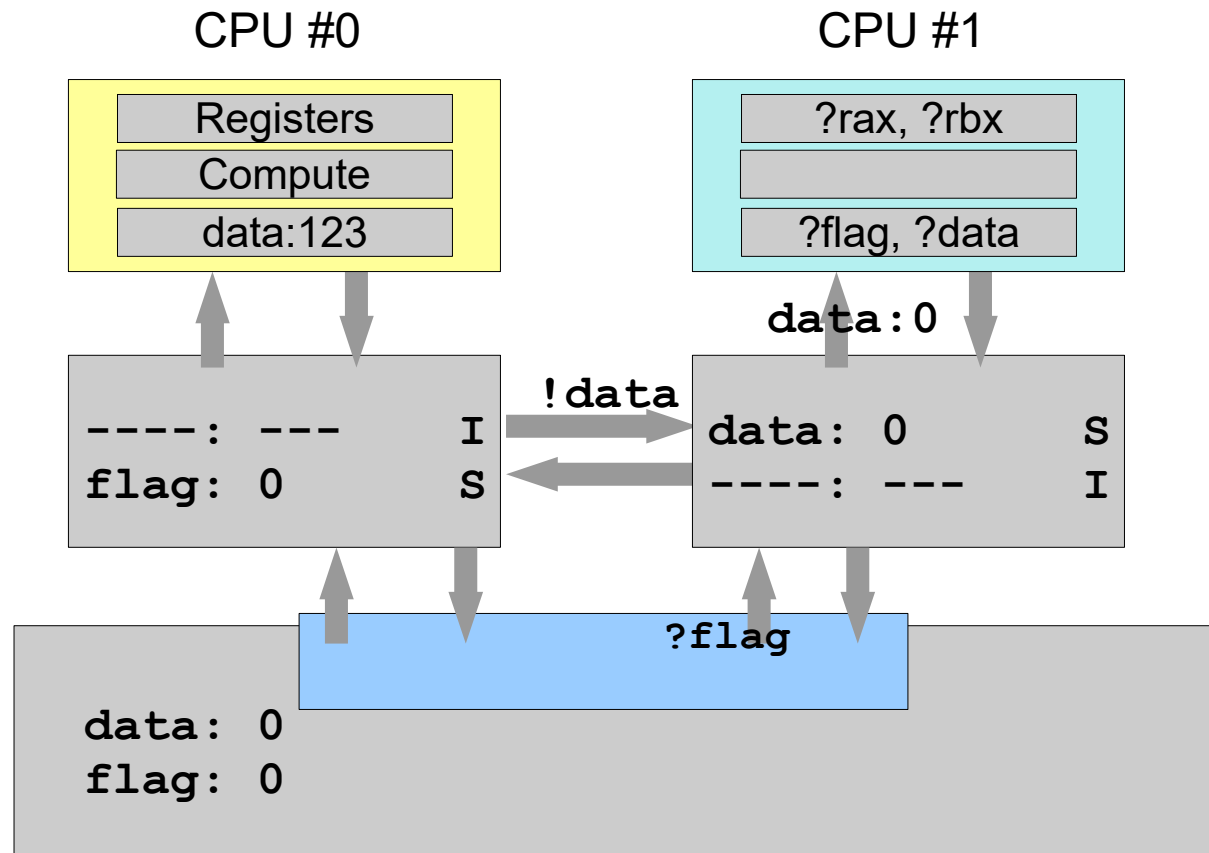
rocketrealtime.com

```
data = ...;
```

```
if( !flag ) ...  
return data;
```

```
mov rax, 123  
st [&data], rax
```

```
ld rax, [&flag]  
jeq rax, ...  
ld rbx, [&data]
```



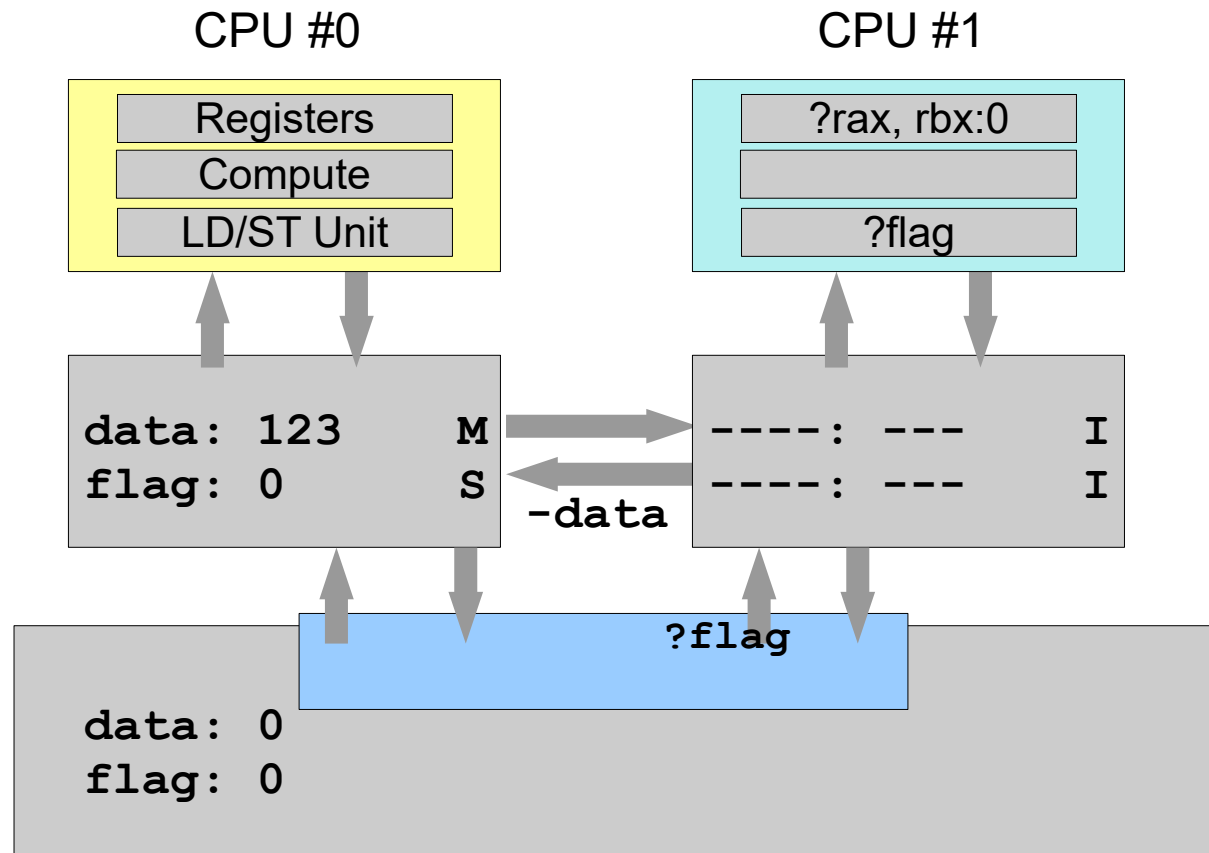
Real Chips Reorder Stuff

```
data = ...;
```

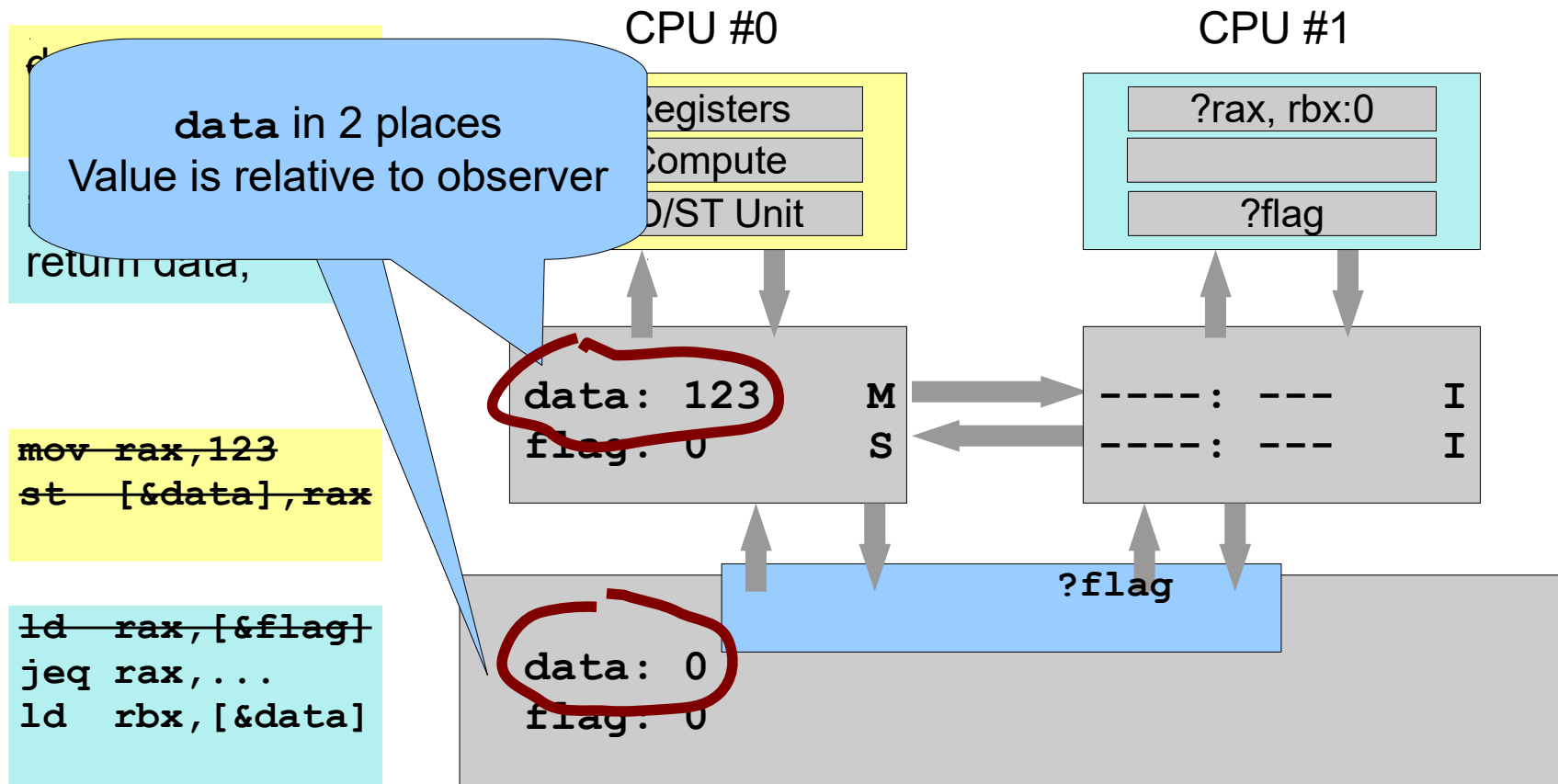
```
if( !flag ) ...  
return data;
```

```
mov rax, 123  
st [&data], rax
```

```
ld rax, [&flag]  
jeq rax, ...  
ld rbx, [&data]
```



Real Chips Reorder Stuff



Real Chips Reorder Stuff

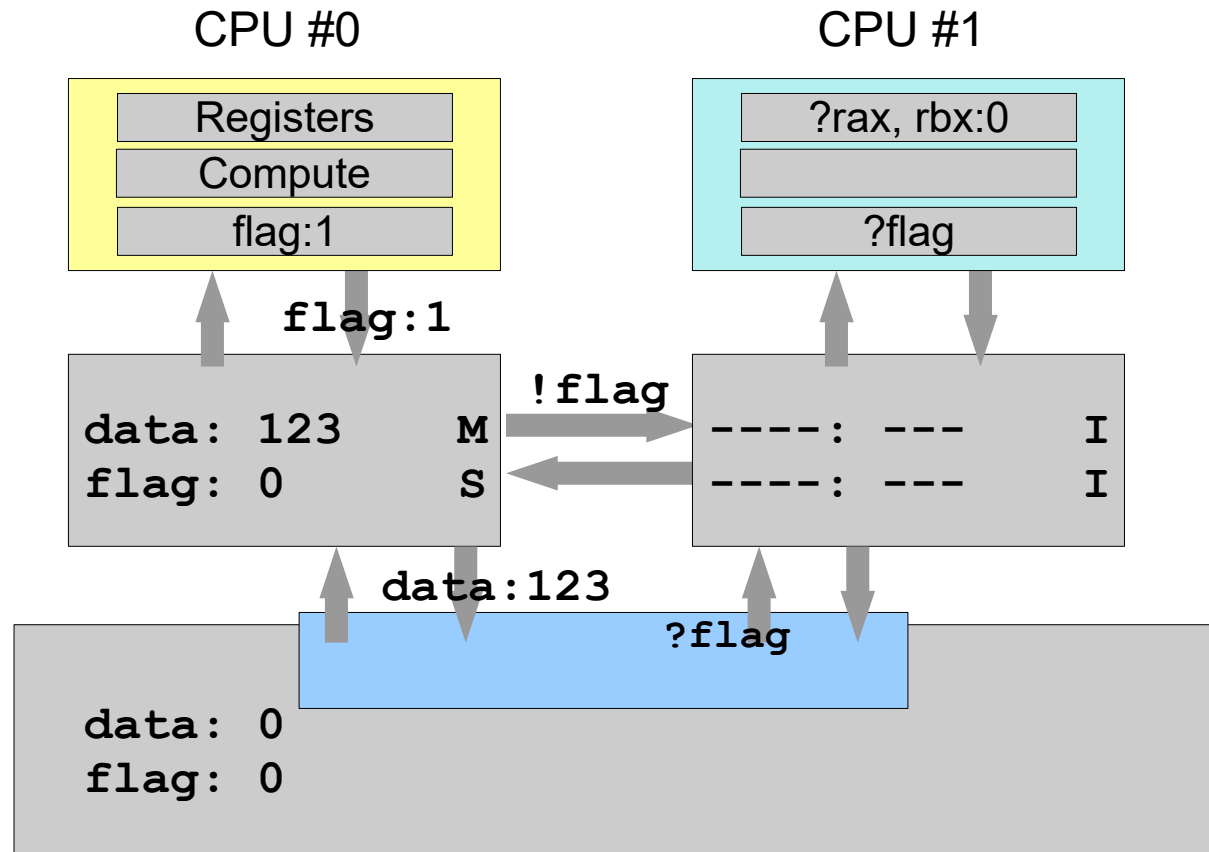
rocketrealtime.com

```
data = ...;  
flag=true;
```

```
if( !flag ) ...  
return data;
```

```
mov rax,123  
st [&data],rax  
st  [&flag],1
```

```
ld rax,[&flag]  
jeq rax,...  
ld  rbx,[&data]
```



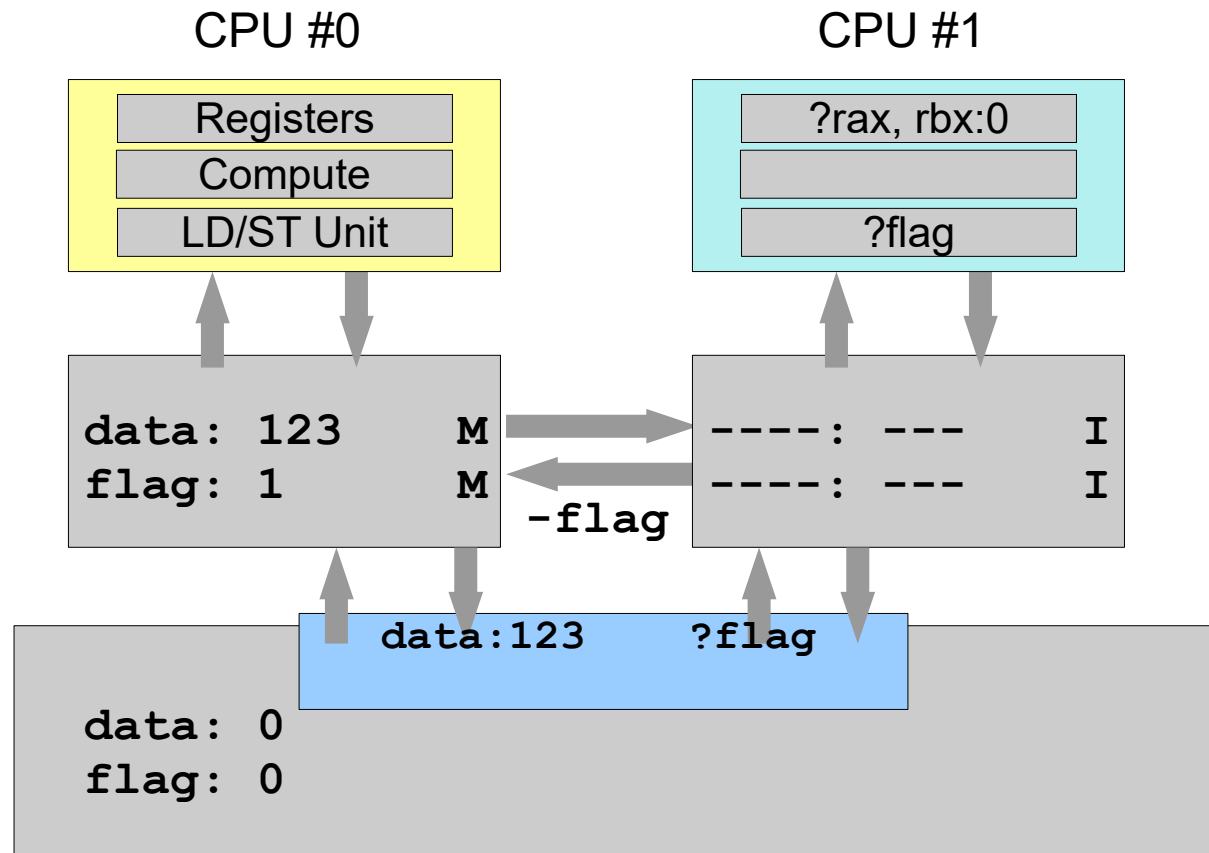
Real Chips Reorder Stuff

```
data = ...;  
flag=true;
```

```
if( !flag ) ...  
return data;
```

```
mov rax, 123  
st [&data], rax  
st [&flag], 1
```

```
ld rax, [&flag]
jeq rax, ...
ld rbx, [&data]
```



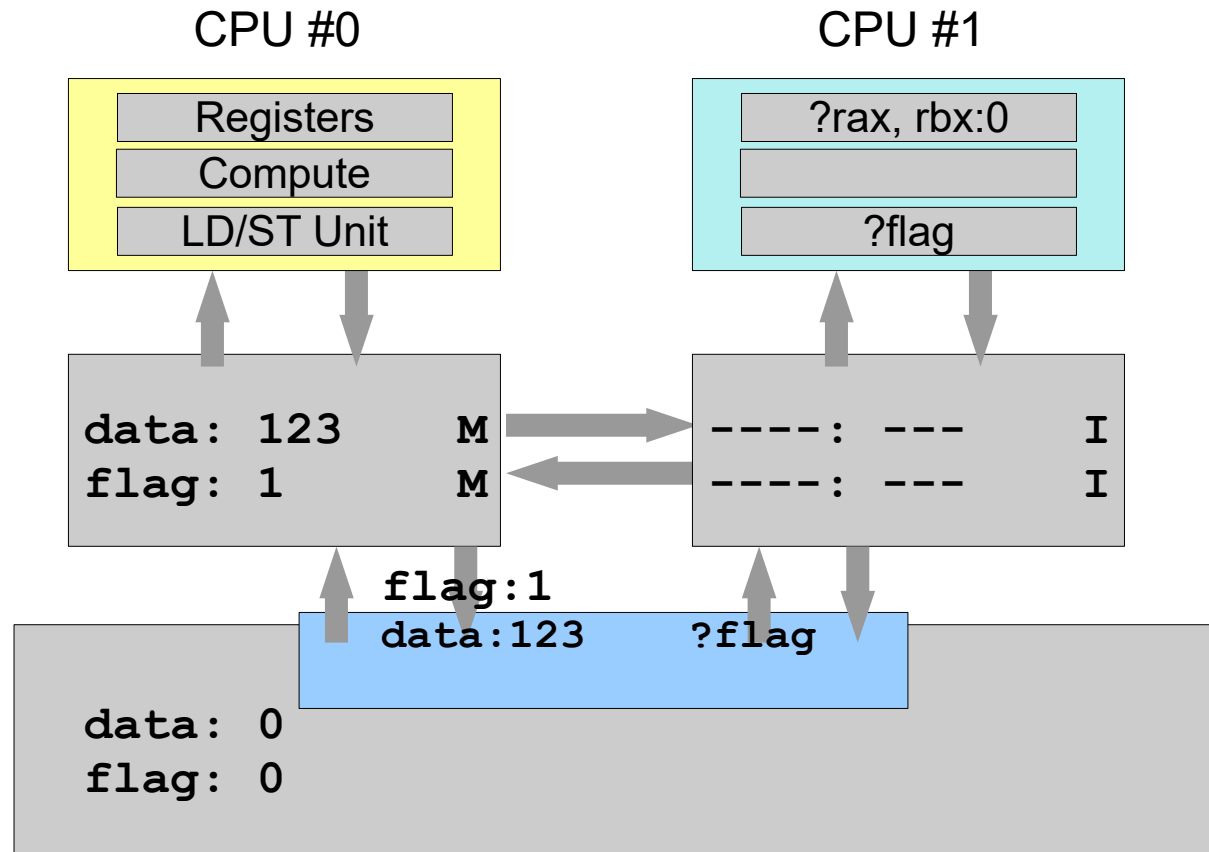
Real Chips Reorder Stuff

```
data = ...;  
flag=true;
```

```
if( !flag ) ...  
return data;
```

```
mov rax,123  
st [&data],rax  
st [&flag],1
```

```
ld rax,[&flag]  
jeq rax,...  
ld rbx,[&data]
```



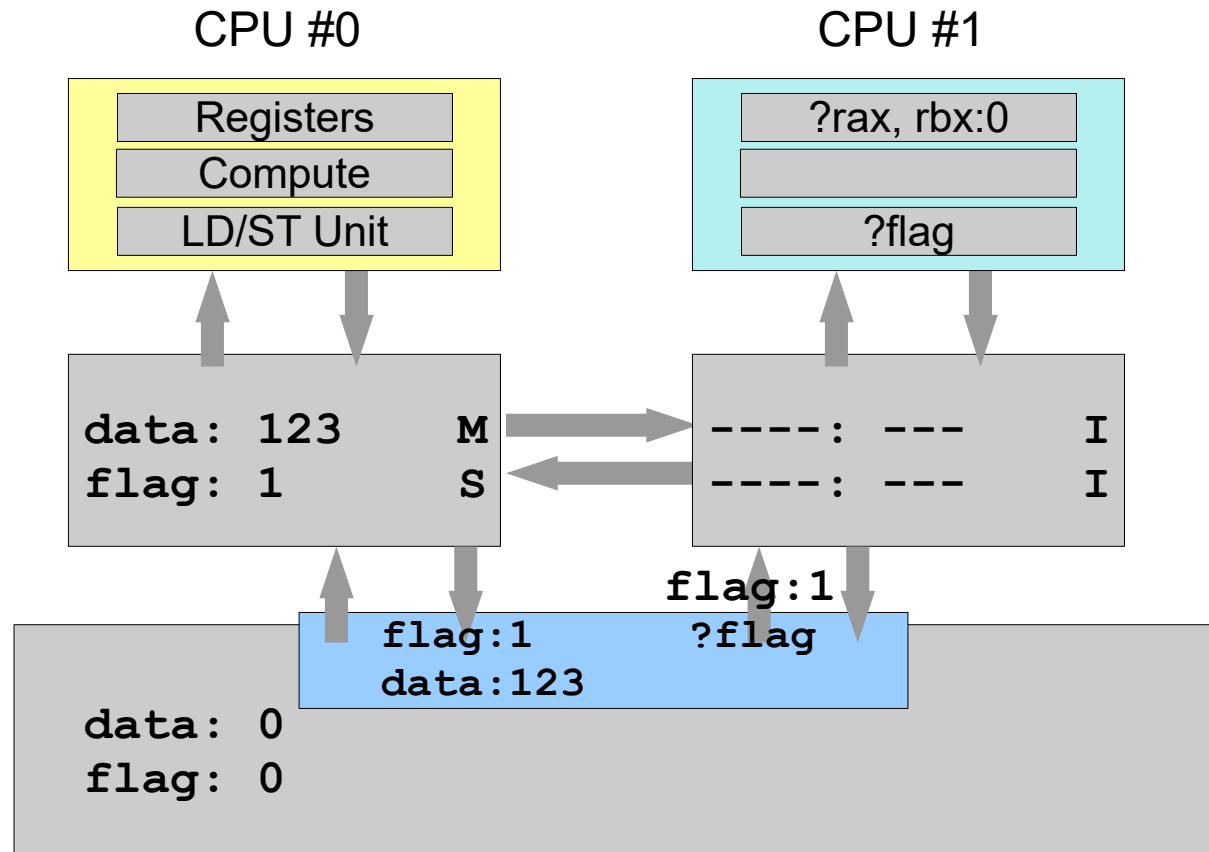
Real Chips Reorder Stuff

```
data = ...;  
flag=true;
```

```
if( !flag ) ...  
return data;
```

```
mov rax,123  
st [&data],rax  
st [&flag],1
```

```
ld rax,[&flag]  
jeq rax,...  
ld rbx,[&data]
```



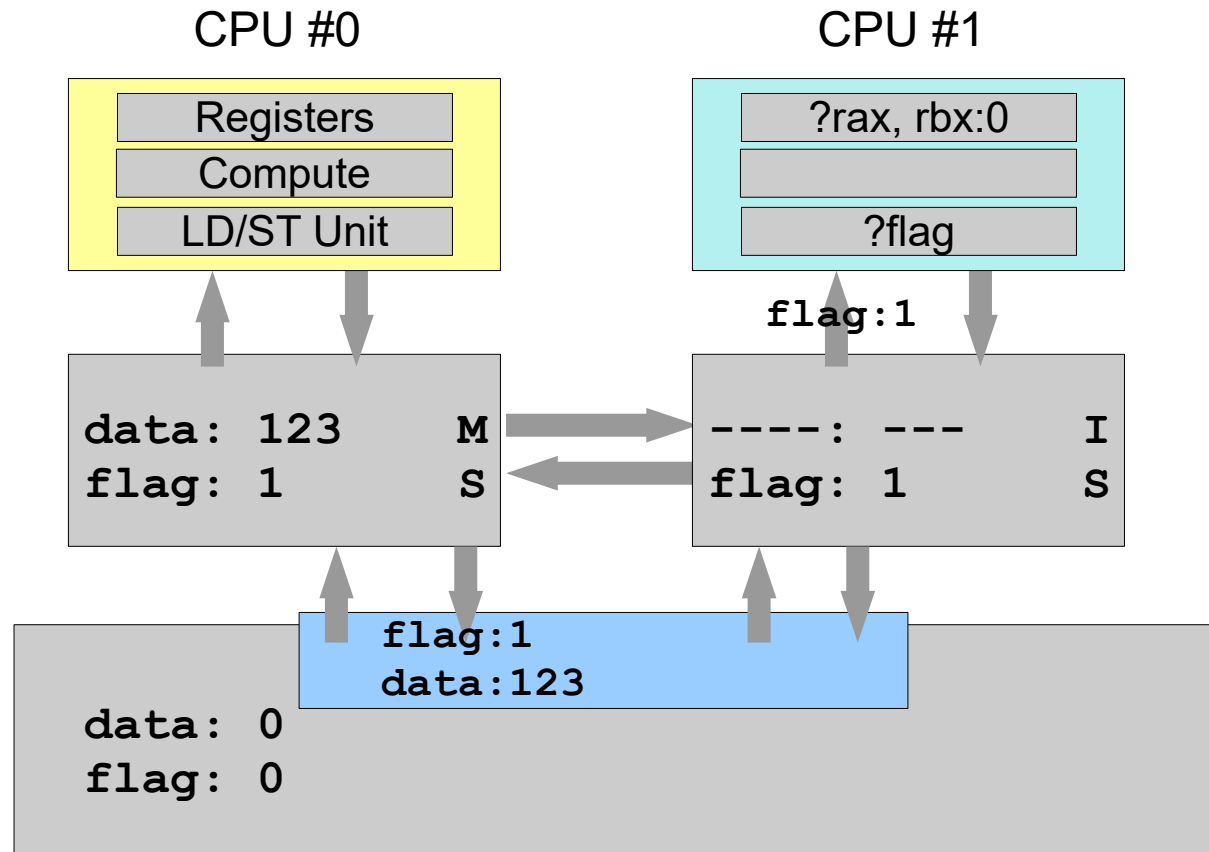
Real Chips Reorder Stuff

```
data = ...;  
flag=true;
```

```
if( !flag ) ...  
return data;
```

```
mov rax,123  
st [&data],rax  
st [&flag],1
```

```
ld rax,[&flag]  
jeq rax,...  
ld rbx,[&data]
```



Real Chips Reorder Stuff

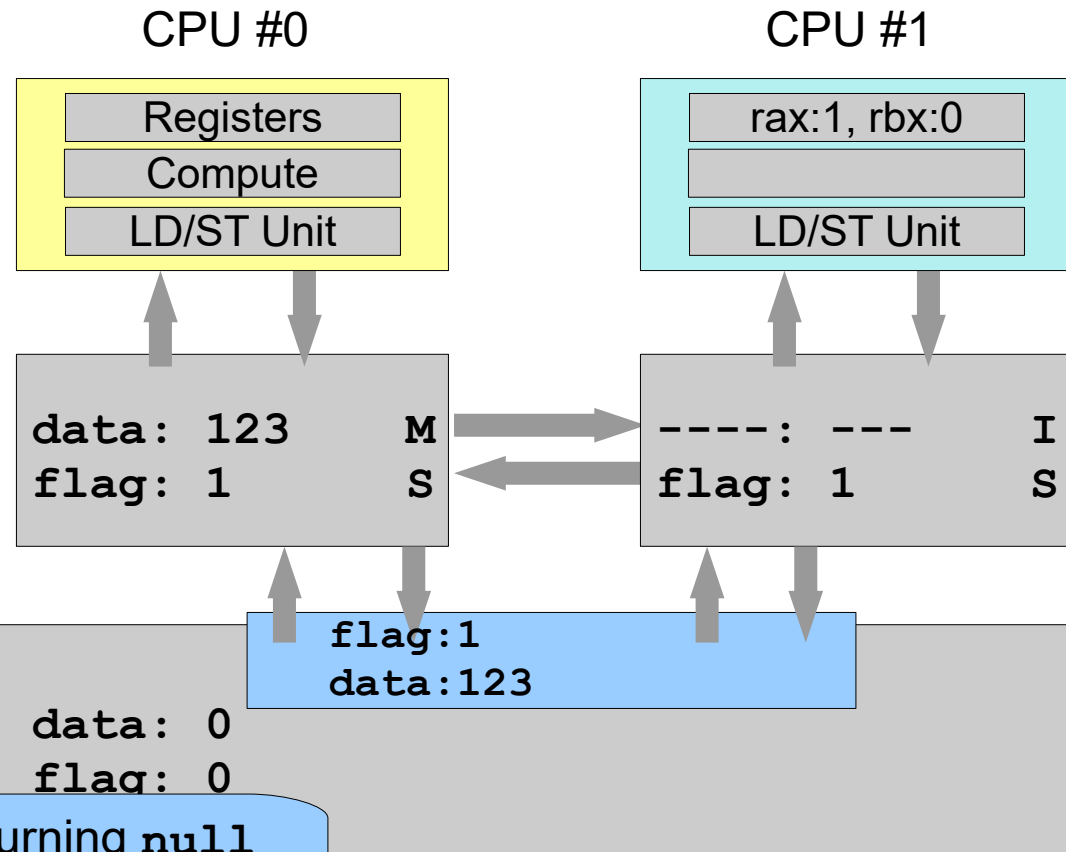
```
data = ...;  
flag=true;
```

```
if(!flag) ...  
return data;
```

```
mov rax,123  
st [&data],rax  
st [&flag],1
```

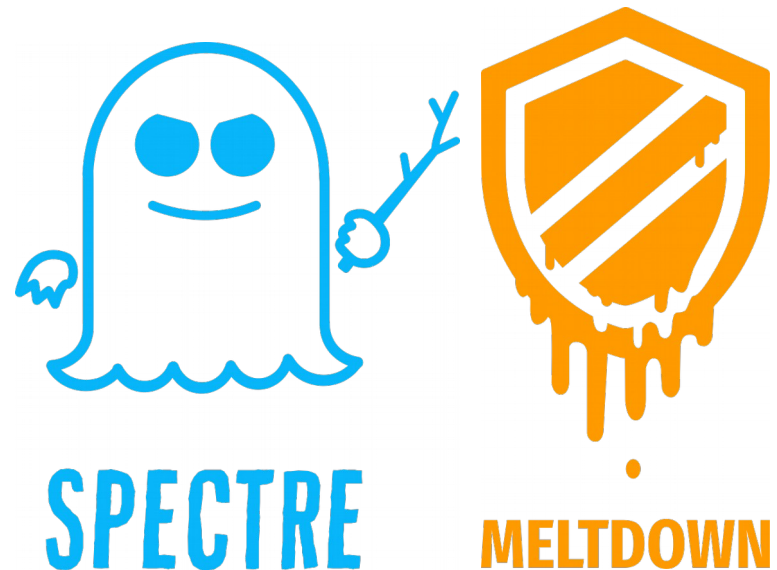
```
ld rax,&flag  
jeq rax,...  
ld rbx,&data  
ret rbx
```

Returning null
Crash-n-burn time!



Agenda

- Introduction
- Serial Memory Performance:
 - The Quest for Single-Core Speed is Over
 - Example 1: Indirection: LinkedList vs ArrayList
 - Example 2: Big Data vs Bandwidth
- Parallel Memory Behavior:
 - A Data Race
 - **Specter and Meltdown**
 - Java Memory Model
- Q&A & Closing Comments



Specter & Meltdown

- Both use CPU speculation
 - Speculation changes the non-architectural state
 - e.g. state of caches, branch prediction, BTB
- Speculation loads secret data into cache
- Secret data still not available to attacker process
 - Security works as expected
- Attacker reads victim cache via timing
 - *Side-channel timing attack*

Victim's L1 Cache

address	value
0x123	0x456
&key	keybytes
0x234	0x567
0x345	0x678

Specter

- Attacker finds certain code in victim's memory:

```
if (x < ary.length)
    y = buf[ary[x] * 256];
```

- **ary** is **byte []**; attacker knows both **ary** and **buf**
- Attacker controls **x** but **x** is range-checked
- Goal: read byte at any address '*key*' using speculation
 - Such as secret crypto byte *k*
- Pick **x = &key - ary**; e.g. **ary[x] == &key**

Specter: Train Branch Prediction

- Want range-check predicted to pass:

```
if (x < ary.length)
    y = buf[ary[x] * 256];
```

- It normally does. Use code normally 'enough' times.
 - 'enough' varies by CPU, but e.g. 100 times should work
- During attack, **ary.length** will miss in cache and CPU will speculate next instruction
 - With carefully selected **ary[x] == key**

branch address	Prediction
&(x<len)	false
0x123	false
0x456	true

Specter: Prepare caches

- L1 cache is e.g. 64kb with 2048 32b lines.
 - Suffices to read 2048 times from 64kb array
 - L2 is larger, but flushed with same strategy
- Desired end result:
 - Cache does NOT hold **ary.length** nor any of **buf**.
- Load 'key' into cache, e.g. ask process to use crypto key
 - Note: 'k' still not available to attacker
 - But IS in cache
 - On the value side, not the address side

address	value
junk	0
junk+32	0
junk+64	0
junk+96	0

address	value
junk	0
key	k
junk+64	0
junk+96	0

Specter: Run the Attack

- Load `ary.length` misses in cache

```
ld4    Rlen, [Rary+8]
```

address	value
junk	0
key	<i>k</i>
junk+64	0
junk+96	0

Specter: Run the Attack

- Load `ary.length` misses in cache
- Range check unknown, pending

<code>ld4</code>	<code>Rlen, [Rary+8]</code>
------------------	-----------------------------

<code>cmp</code>	<code>Rx, Rlen</code>
------------------	-----------------------

address	value
junk	0
key	<i>k</i>
junk+64	0
junk+96	0

Specter: Run the Attack

- Load `ary.length` misses in cache
- Range check unknown, pending
- Branch speculates

<code>ld4</code>	<code>Rlen, [Rary+8]</code>
<code>cmp</code>	<code>Rx, Rlen</code>
<code>ja</code>	<code>fail_check</code>

address	value
junk	0
key	<i>k</i>
junk+64	0
junk+96	0

Specter: Run the Attack

- Load `ary.length` misses in cache
- Range check unknown, pending
- Branch speculates
- Following load of '*k*' hits in cache

ld4	Rlen, [Rary+8]
cmp	Rx, Rlen
ja	fail_check
ld1	Rk, [Rary+Rx]
mul	Rtmp, Rk*256

address	value
junk	0
key	<i>k</i>
junk+64	0
junk+96	0

Specter: Run the Attack

- Load `ary.length` misses in cache
- Range check unknown, pending
- Branch speculates
- Following load of '*k*' hits in cache
- Dependent load changes cache
- Note: value of '*k*' now in cache address

ld4	Rlen, [Rary+8]
cmp	Rx, Rlen
ja	fail_check
ld1	Rk, [Rary+Rx]
mul	Rtmp, Rk*256
ld8	Ry, [Rtmp+Rbuf]

address	value
junk	0
key	<i>k</i>
junk+64	0
<i>k</i> *256+buf	0x1234

Specter: Read out secret 'k'

- Time cache loads
 - `for i=0 to 255`
 - `rdtsc // read fast&accurate counter`
 - `ld Ra,[buf+i*256]`
 - `Diff rdtsc // check speed of load`
 - Will be fast for `i=='k'` and slow for other `i`
- We now have 'k'
- Repeat for other bytes of 'key'
 - ...and we now have entire crypto key
- Actually, fast enough to read much of process

Agenda

- Introduction
- Serial Memory Performance:
 - The Quest for Single-Core Speed is Over
 - Example 1: Indirection: LinkedList vs ArrayList
 - Example 2: Big Data vs Bandwidth
- Parallel Memory Behavior:
 - A Data Race
 - Specter and Meltdown
 - **Java Memory Model**
- Q&A & Closing Comments



Java Memory Model

- It's Big, It's Complicated, It's Important...
- <http://g.oswego.edu/dl/jmm/cookbook.html>
<http://www.cl.cam.ac.uk/~pes20/weakmemory/index.html>
<https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf>
Sections 17.4, 17.5
- Ok, I'm assuming you are at least familiar
- Data Race: Requires Two (Threads) To Tango
 - What reorderings ARE possible?
- Mental shortcut: Happens-Before Ordering
 - What "Happens Before" what?


Reordering Memory Ops

T1	T2
<code>_data = stuff</code>	
<code>_init = true</code>	
	<code>r1 = _init</code> <code>if(!r1) ...</code>
	<code>r2 = _data</code>

- Writing 2 fields
- Can T2 see stale `_data`?
- Yes!

Reordering Memory Ops

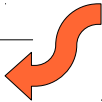
rocketrealtime.com

T1	T2
<code>_data = stuff</code> <code>_init = true</code>	<code>r2 = _data</code>  <code>r1 = _init</code> <code>if(!r1) ...</code>

- Writing 2 fields
- Can T2 see stale `_data`?
- Yes!
- **Compiler can reorder**
 - Standard faire for -O
- Java: make `_init` volatile

Reordering Memory Ops

rocketrealtime.com


T1	T2
<pre>_data = stuff _init = true</pre>	<pre>r1 = _init if(!r1) ... // predict r2 = _data ...r1 true ...so keep r2</pre> 

- Writing 2 fields
- Can T2 see stale `_data`?
- Yes!
- **Hardware can reorder**
- Load `_init` misses cache
- Predict `r1==true`
- Speculatively load `_data` early, hits cache
- `_init` comes back true
- Keep speculative `_data`

Reordering Memory Ops

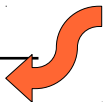
T1

T2



```
_data = stuff  
_membar  
_init = true
```


```
r1 = _init  
if( !r1 ) ...  
_membar  
r2 = _data
```



- Writing 2 fields
- Can T2 see stale `_data`?
- No!
- Need store-side ordering
- Need load-side ordering
- Included in Java volatile

Reordering Memory Ops

rocketrealtime.com

T1	T2
<pre>_data = stuff _membar—— _init = true</pre>	<pre>r1 = _init if(!r1) ... // predict r2 = _data</pre>  <pre>...r1 true ...so keep r2</pre>

- Writing 2 fields
- Can T2 see stale `_data`?
- Yes!
- **Missing load-ordering**
- Read of `_init` misses
- Predict branch
- Fetch `_data` early
- Confirm good branch

Reordering Memory Ops

rocketrealtime.com

T1

`_data = stuff`

`_init = true`


T2

`r1 = _init`

`if(!r1) ...`

`—membar—`

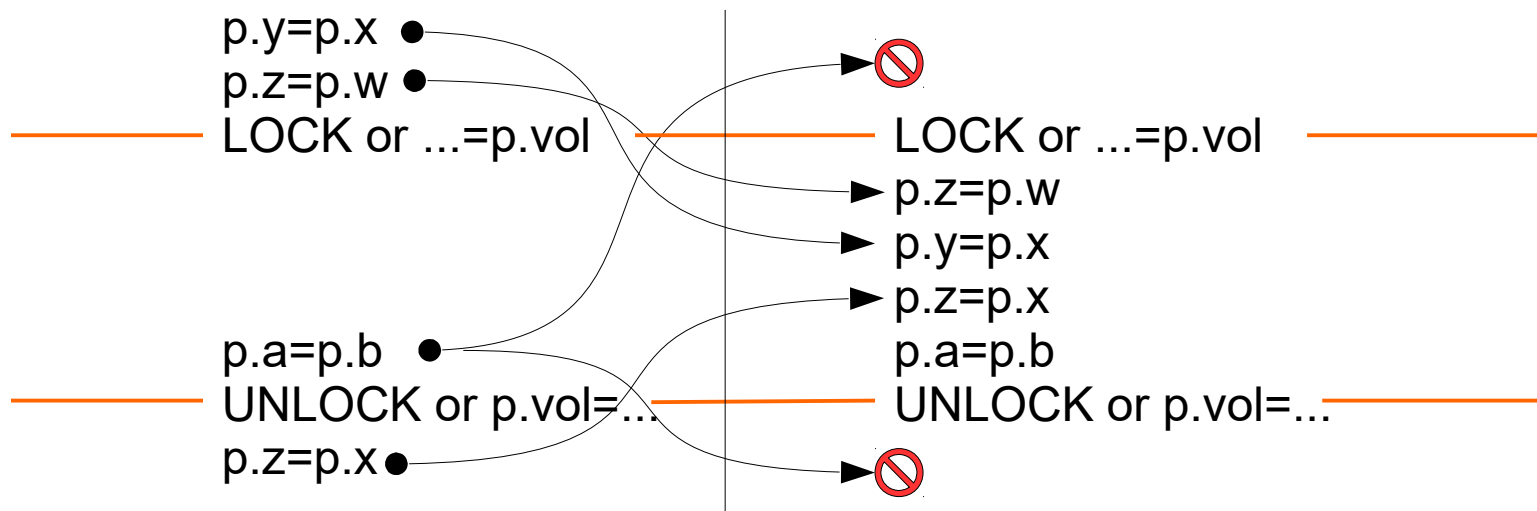
`r2 = _data`

 ...stuff
actually
...visible

- Writing 2 fields
- Can T2 see stale `_data`?
- Yes!
- **Missing store ordering**
- Write of `_data` misses
- Write of `_init` hits cache
- T2 reads `_init`
- T2 reads stale `_data`

Normal Loads and Stores Reorder

- Normal loads & stores reorder:
 - Freely reorder with each other
 - Done by the JIT and some hardware (not X86)
- “Roach Motel:” Can move into “locked” regions, but not out
 - Includes volatile-load as a “lock”
 - Includes volatile-store as a “unlock”



Chasing Happens Before

- Reach for HB when designing concurrent algorithms
- A “HB” B if
 - B uses A’s result (e.g. load/branch-if-true)
 - A lock owner changed threads
 - Or a vol-load/vol-store
- HB is fairly intuitive
 - Lay out “train track” of 2 threads
 - Bounce HB edges across them
 - Slide tracks or events up & down
 - Try to get “bad” things to slide past each other

Agenda

rocketrealtime.com

- Introduction
- Serial Memory Performance:
 - The Quest for Single-Core Speed is Over
 - Example 1: Indirection: LinkedList vs ArrayList
 - Example 2: Big Data vs Bandwidth
- Parallel Memory Behavior:
 - A Data Race
 - Specter and Meltdown
 - Java Memory Model
- **Q&A & Closing Comments**

Managing performance

- Dominant operations
 - 1985: page faults
 - Locality is critical
 - 1995: instructions executed
 - Multiplies are expensive, loads are cheap
 - Locality not so important
 - 2005: cache misses
 - Multiplies are cheap, loads are expensive!
 - Locality is critical again!
 - 2015: same speed cores, but more of them
- We need to update our mental performance models as the hardware evolves

Think Data, Not Code

- In the old days, we could count instructions
 - Because instruction time was predictable
- Today, performance is dominated by patterns of memory access
 - Cache misses dominate – memory is the new disk
 - VMs are very good at eliminating the cost of code abstraction, but not yet at data indirection
- Multiple data indirections may mean *multiple cache misses*
 - ***That extra layer of indirection hurts!***

Think Data, Not Code

rocketrealtime.com

- Remember when buffer-copy was bad?
 - (hint: 80's OS classes, zero-copy network stacks)
- Now it's Protobuf → JSON → DOM → SQL → ...
- Each conversion passes all data thru cache
- Don't bother converting unless you must!
- If you convert for speed (e.g. JSON → DOM)
 - Then must recoup loss with repeated DOM use
 - A 1-use conversion is nearly always a loser
- “Out of Cache” is hard to spot in most profilers
 - Just looks like everything is slow

Share & mutate less

- Shared data == OK
- Mutable data == OK
- Shared + mutable data = **DANGER**
 - Requires synchronization
 - Error-prone, has costs
 - More likely to generate cache contention
 - Multiple CPUs can share a cache line if all are readers
- Immutability also tends to make for more robust code

Summary

- CPUs give the illusion of simplicity
- But are ***really complex*** under the hood
 - There are lots of parts moving in parallel
 - The performance model has changed
 - Heroic efforts to speed things up are mined out
- Performance analysis is not an armchair game
 - Unless you profile (deeply) you just don't know
 - Premature optimization is the root of much evil

High Performance From Understanding The Low Levels

Dr. Cliff Click

rocket.realtime.school@gmail.com

cliffc@acm.org

cliffc.org/blog