Concurrency: The Hardware Perspective

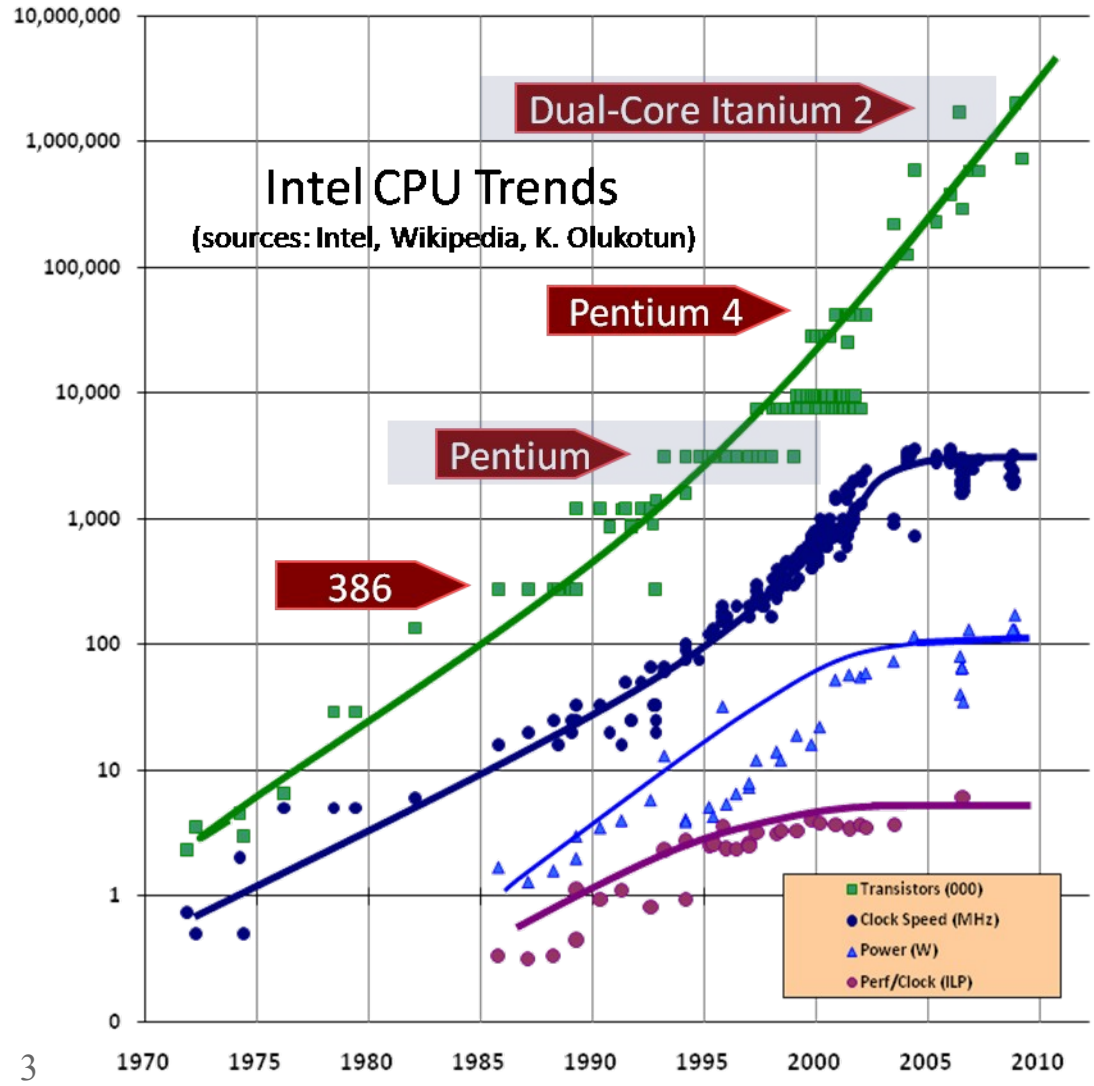
Or, how did we find ourselves in this mess?

Brian Goetz
Sun Microsystems
brian@briangoetz.com

Agenda

- **Introduction**
- The memory wall
- The Quest for ILP
- Multicore and CMT
- Changing our Behavior

The Future is Parallel – Get Used To It



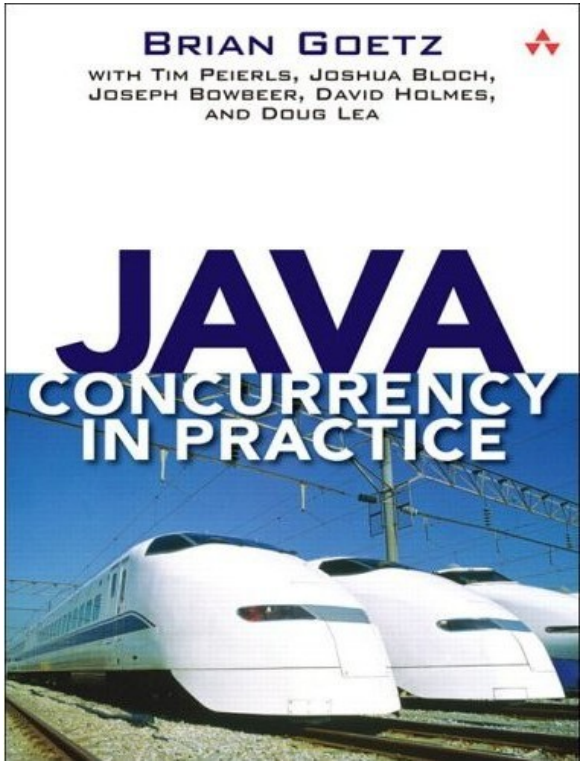
- Obligatory "free lunch is over" graph
  - Shows trends in Moore's Law, CPU speed, power, and ILP over 40 years
  - Graph courtesy of Herb Sutter
- This talk will focus on the trends that are pushing us to an increasingly parallel world
  - Whether we like it or not

The Role of Threads

- A long time ago...
  - Multiway systems were rare
  - Threads were used primarily for asynchrony
  - Concurrent programming was the realm of wizards
- Several boiled frogs later...
  - Every system is concurrent today
  - Every app is concurrent today
    - Concurrency often injected silently via frameworks
  - But programming with locks and threads still requires wizardry

Shameless Plug

- Programming with threads and locks requires wizardry...
  - Wizardry Instruction Manual available...



The Role of Threads

- Threads were good enough when we could limit concurrency it to small corners of the program
  - Could be crafted by wizards
- Threads have evolved for exposing hardware parallelism
  - Necessary, but increases complexity and risk
  - We didn't really notice this until the water was hot!
- The water is only going to get hotter

Hardware Trends

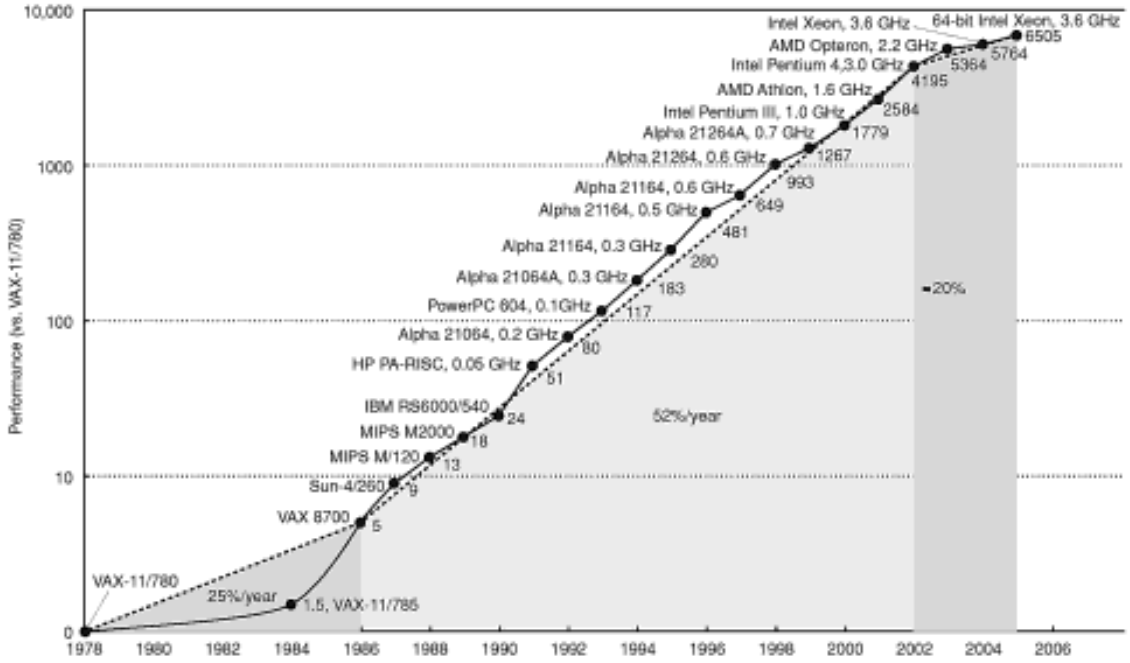
- For years, CPU designers focused on increasing sequential performance
  - Higher clock frequency
  - Instruction-level parallelism (ILP)
- These factors created the "free lunch" environment we got used to
  - But we've hit the wall on all of these
- Going forward CPU designers will focus on parallelism for increasing throughput
  - Optimizing for computing bandwidth over latency

Hitting the wall

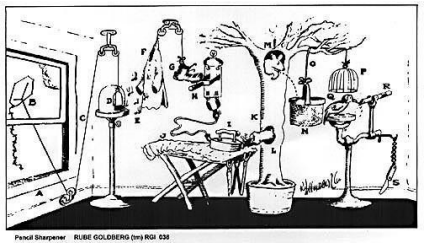
- Serial performance has hit the wall
  - Power Wall
    - Higher freq → more power → more heat → chip melts!
  - Speed of light
    - Takes more than a clock cycle for signal to propagate across a complex CPU!
  - Memory Wall
    - Memory performance has lagged CPU performance
    - Program performance now dominated by cache misses
  - ILP Wall
    - Hitting limits of practicality in branch prediction, speculative execution, multiple issue

CPU Archeology

- Three main periods in CPU history
  - CISC era
  - RISC era
  - Multicore era



CISC systems



- CISC ISAs were designed to be *used by humans*
  - Canonical example: VAX
    - Orthogonal instruction set
      - Any instruction could be used with any data type and any combination of addressing modes
    - Exotic primitives for functionality that would today live in libraries
      - Packed character arithmetic, string pattern matching, polynomial evaluation
    - Lots of addressing modes
      - Multiple levels of indirection possible in a single instruction
        - Convenient to program, hard to pipeline!
      - Example: ADDL3 4(R1)[R2], @8(R1), R3

CISC systems

- CPI (cycles per instruction) for CISC chips varied
  - 4-10 was typical (but highly predictable!)
  - Program performance was basically:
    N*page faults + instructions executed
    - Memory was expensive
      - So performance was generally dominated by page faults
    - Both page fault count and instruction count were easy to measure
    - *Managing code and data locality was key to good performance*
- CISC systems were hard to scale up
  - Outrun by the dumber but more agile RISC chips!

RISC systems

- RISC == Reduced Instruction Set Computing
  - Design reboot – CPUs had gotten too complicated
  - RISC has a highly simplified instruction set
    - No memory-memory ops, simpler addressing modes
    - No exotic ops
    - Fixed width instructions
    - Designed for easy pipelining
  - Simplified architecture was easier to scale
  - Cost: ISA not practical for programming by hand
    - Required more sophisticated compilers

The Next Reboot

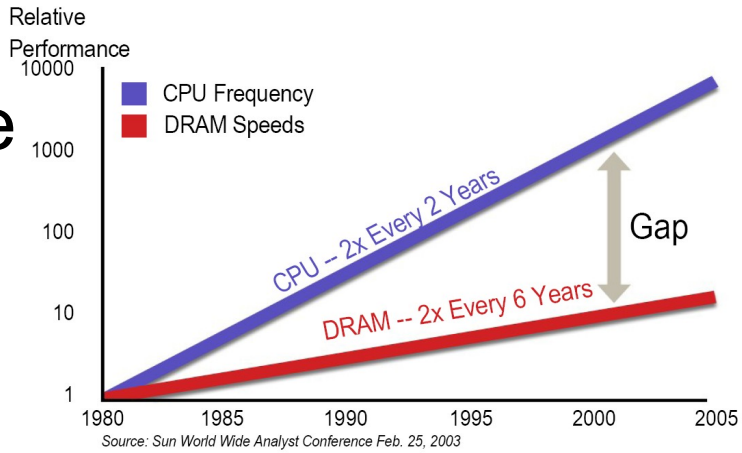
- Just as RISC rebooted CISC, we're in the midst of another reboot
  - Clock rates have been basically flat for 5 years
  - RISC chips got complicated again!
    - Wacky speculative out-of-order execution
    - Deep pipelines
    - Heroic attempts to work around memory latency
      - 80% of chip real estate devoted to cache
      - Primary driver for wacky OOO techniques is memory latency
- Solution: more cores – but slower, simpler cores
  - How are we going to use those cores?
  - How must we change the way we program?

Agenda

- Introduction
- **The Memory Wall**
- The Quest for ILP
- Multicore and CMT
- Changing our Behavior

Memory subsystem performance

- Memory and CPU performance both growing exponentially
  - But with different bases
  - Exponentially widening gap



- In older CPUs, memory access was only slightly slower than register fetch
  - Today, fetching from main memory could take several hundred clock cycles
- Drives need for sophisticated multilevel memory caches
  - And cache misses still dominate performance

Types of memory

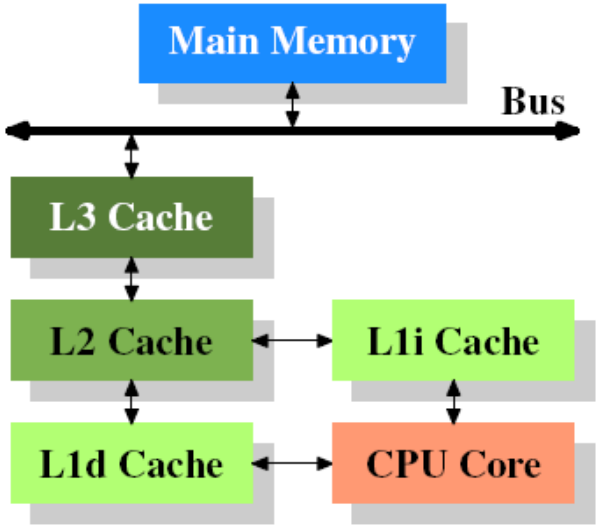
- Static RAM (SRAM) – fast but expensive
  - Six transistors per bit
- Dynamic RAM (DRAM) – cheap but slow
  - One transistor + one capacitor per bit
  - Improvements in DRAM (DDR, DDR2, etc) generally improve bandwidth but not latency
    - Stated clock rate on memory understates latency
      - 800MHz FSB is really 200MHz with four transfers per cycle
      - DRAM protocol has many wait states

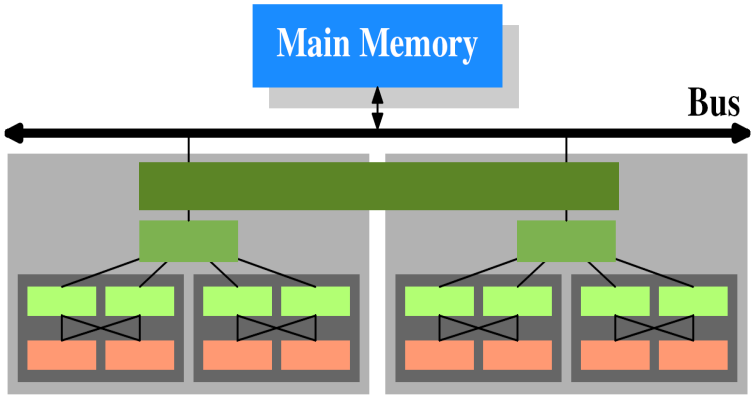
Caching

- 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
- Cost: more of our transistor budget used for cache

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: ~200 clks
- On multicore systems, lowest cache layer is shared
  - But not all caches visible to all cores





Caching

- Trends in memory latency turn traditional performance models upside down
  - 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!**
- Speeding up the CPU doesn't help if it spends all its time waiting for data

In search of faster memory access

- To make memory access cheaper
  - Relax coherency constraints
    - Exposes inherent parallelism and nondeterminism
  - Improves throughput, not latency
- 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
- The Memory Wall
- **The Quest for ILP**
- Multicore and CMT
- Changing our Behavior

The Quest for ILP

- ILP = Instruction Level Parallelism
  - Attempts to exploit inherent parallelism *transparently*
- Goal: faster CPUs at the same clock rate, via
  - Pipelining
  - Branch Prediction
  - Speculative execution
  - Multiple-issue
  - Out-Of-Order (O-O-O) execution
  - And much more...

The Quest for ILP: Pipelining

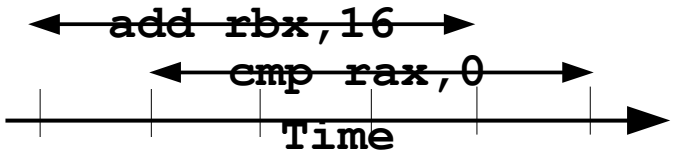
- Internally, each instruction has multiple stages
  - Many of which must be done sequentially
    - Fetching the instruction from memory
      - Also identifying the end of the instruction (update PC)
    - Decoding the instruction
    - Fetching needed operands (memory or register)
    - Performing the operation (e.g., addition)
    - Writing the result somewhere (memory or register)
  - Which means that executing an instruction from start to end 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, some FP ops)

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
  - But how long is not transparent

Loads & Caches

```
ld    rax⇦[rbx+16]
...
cmp   rax,0
```

RAX still not available

- Simplest CPUs *stall* execution until value is ready
  - e.g. Typical GPU
- 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

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 CPU resource
- Common CPU feature

Speculative Execution

- Speculative execution, branch prediction, wide issue, and out-of-order execution are synergistic
- Keep speculative state 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
    - Like committing a transaction

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]
```

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

```
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

```
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

```
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

```
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

- Originally, these tricks were intended to just run more instructions at a time
  - When run time was roughly equivalent to instruction count
- As memory latency dominates, the goal changes
  - Allow more execution past cache misses
    - So we can start the *next* cache miss earlier
    - Run multiple cache misses in parallels

The Quest for ILP

- Itanium: a Billion-$$$ Effort to mine *static* ILP
- Theory: Big Gains possible on “infinite” machines
  - Machines w/infinite registers, infinite cache-misses, infinite speculation, etc
- Practice: Not much gain w/huge effort
  - Instruction encoding an issue
  - Limits of compiler knowledge
    - e.g. memory aliasing even with whole-program opt
  - Works well on scientific apps
  - Not so well on desktop & server apps

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 pretty 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
  - Latency is killing us
- Contrast to GPUs – zillions of simple cores
  - But only works well on narrow problem domain

Agenda

- Introduction
- The Memory Wall
- The Quest for ILP
- **Multicore and CMT**
- Changing our Behavior

Multicore and CMT

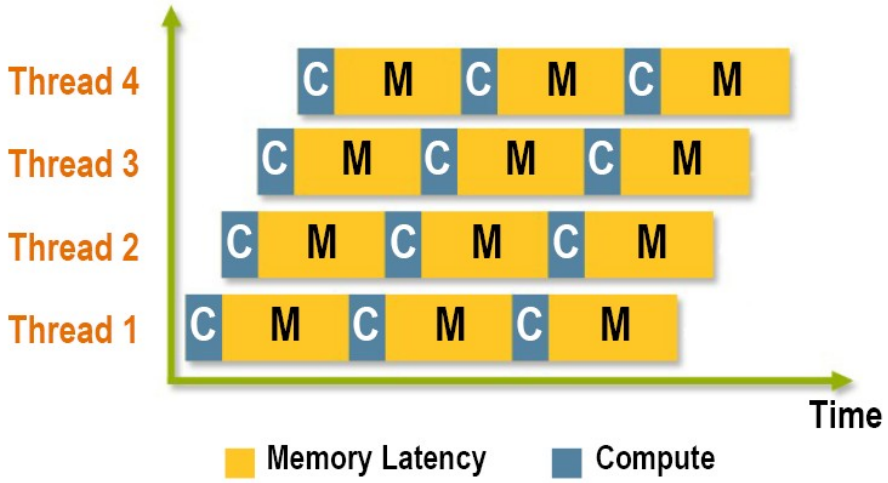
- Moore's law has been generous to us
  - But the tax rate for sequential performance is ever-increasing
    - Spending lots of transistors on speculative out-of-order execution, cache coherency protocols, and cache
    - And hitting diminishing returns
- Time to reboot RISC
  - RISC started simple, but got complicated
  - Spend that budget on more slower, simpler cores
    - Lower tax rate == more useful work per transistor

Multicore and CMT

- More useful work per transistor sounds good
  - But … must use those cores effectively
- One pesky problem: *concurrency is hard*
  - We're still figuring that one out...
  - This is the primary reason we don't have 100 core chips on every desktop
    - Can easily fit 100 Pentium III cores on an Itanium die
      - Pentium III had ~10M transistors
      - Dual-core Itanium has 1.7B
    - But the lack of parallel software is the gating factor

Chip Multi-Threading (CMT)

- CMT embraces memory latency and let multiple threads share computation pipelines
  - Every cycle, an instruction from a different thread is executed
    - Improves throughput, not latency
    - Can achieve very high pipeline utilization
  - Niagara supports 4 threads per core
  - Niagara II supports 8

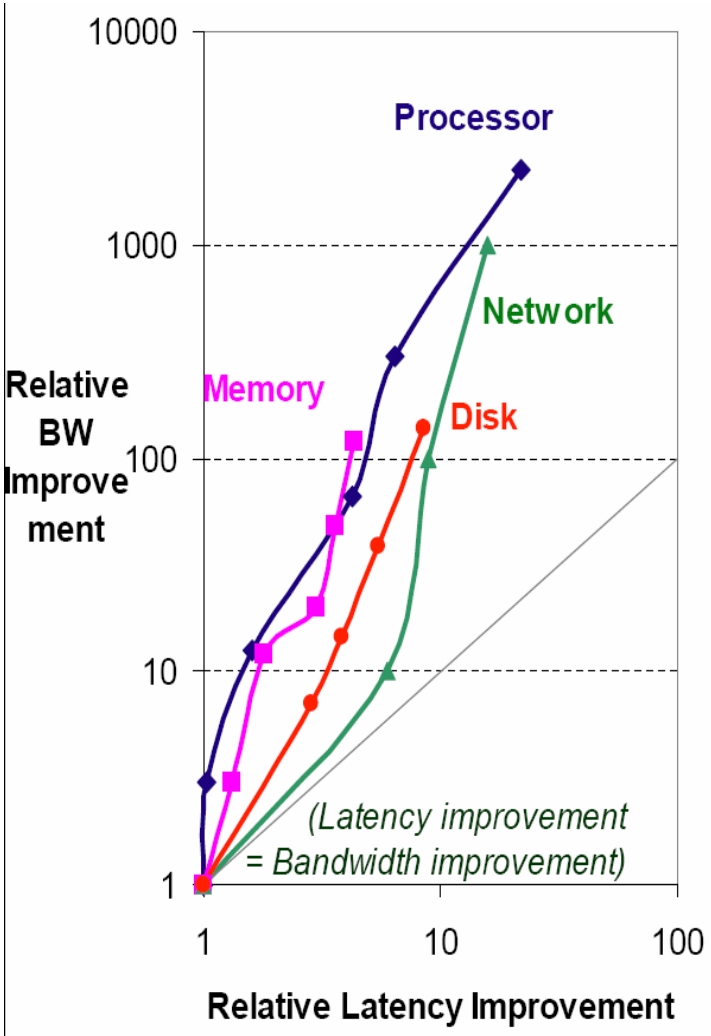


Latency is the Enemy

- Much of the complexity of OOO CPUs comes from the need to work around memory latency
- The cure for latency is … concurrency
  - Use threads to hide latency inherent in asynchronous IO
  - ILP hides memory latency by finding other work to do (possibly speculatively)
  - Lots Of Simple Cores hides memory latency by ensuring there's always a thread ready to run
- Concurrency hides latency!

Latency is Everywhere

- Easier to improve bandwidth than latency
  - True for CPU, memory, disk, networks
- Moore's Law is generous to bandwidth, not to latency
  - More/faster transistors helps bandwidth
  - Can't outrace speed of light



New metrics, new tools

- With CMT, speedup depends on memory usage characteristics!
  - Code with lots of misses may see linear speedup
    - (until you run out of bandwidth)
  - Code with no misses may see none
- CPU utilization is often a misleading metric
- Need tools for measuring pipeline utilization, cache
  - Such as corestat for Sparc
- Out-of-cache is hard to spot in most profilers
  - Just looks like all code is slow...

Agenda

- Introduction
- The Quest for ILP
- Memory Subsystem Performance
- Multicore and CMT
- **Changing our behavior**

Think Data, Not Code

- 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

- Twenty years ago, we had to worry about locality to avoid paging
  - Ten years ago, we forgot about that
  - Locality is back!
- Data locality should be a first-class design concern for high-performance software

Exploiting Parallelism

- Keeping lots of cores busy requires more flexible decomposition of programs
  - Fork-join / map-reduce
  - Specify decomposition strategy independently of execution topology
    - Let the runtime distribute across cores / nodes based on availability
- Consider message-passing approaches over direct coupling

Share less, mutate less

- Shared data == OK
- Mutable data == OK
- Shared + mutable data = EVIL
  - More likely to generate cache contention
    - Multiple CPUs can share a cache line if all are readers
  - Requires synchronization
    - Error-prone, has costs
- Bonus: exploiting immutability also tends to make for more robust code
  - Tastes great, less filling!

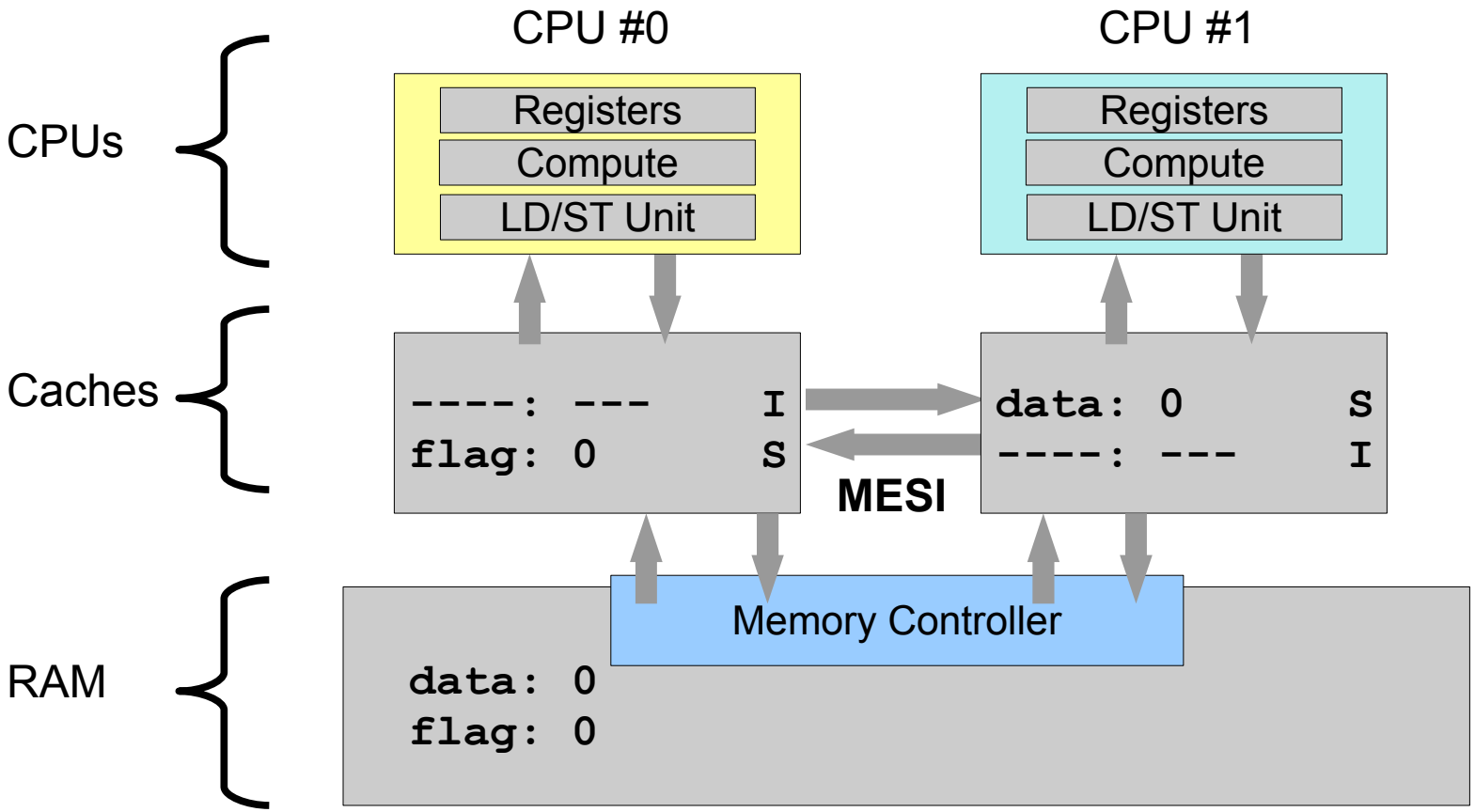
Summary

- CPUs give the illusion of simplicity
- But have grown ***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
- Pendulum swinging back towards more, simpler cores
  - Primary challenge: keeping those cores busy

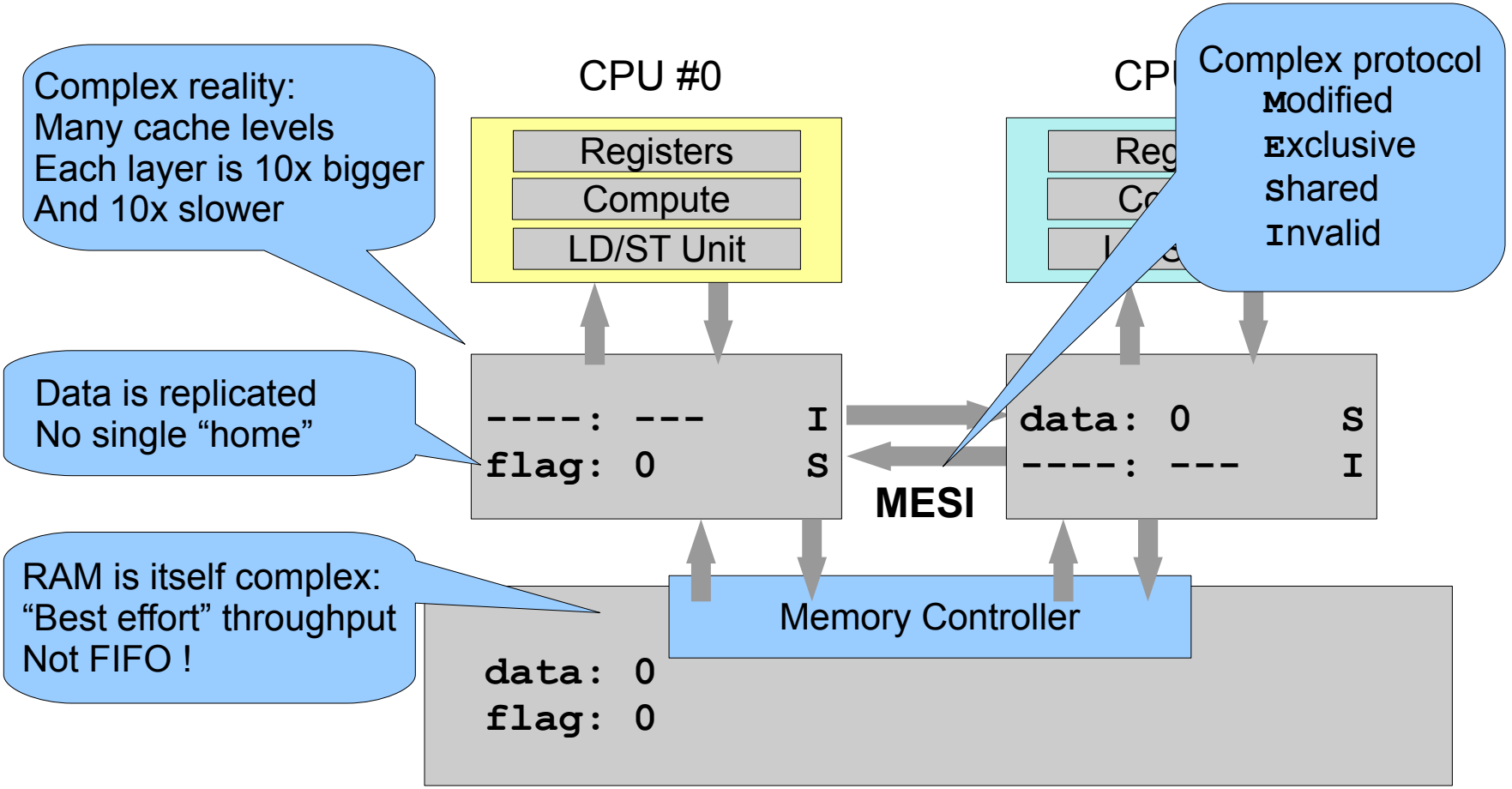
For more information

- *Computer Architecture: A Quantitative Approach*
  - Hennesey and Patterson
- *What Every Programmer Should Know About Memory*
  - Ulrich Drepper

Real Chips Reorder Stuff

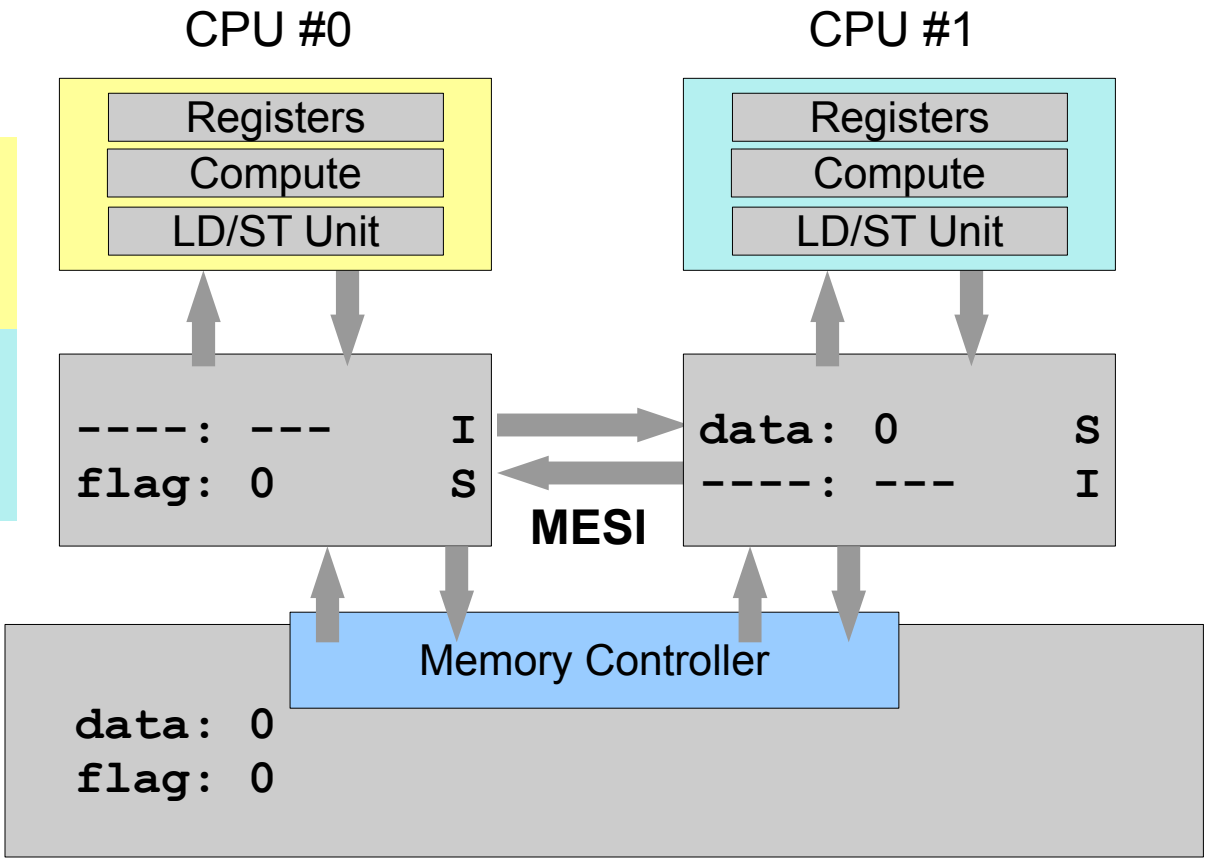


Real Chips Reorder Stuff



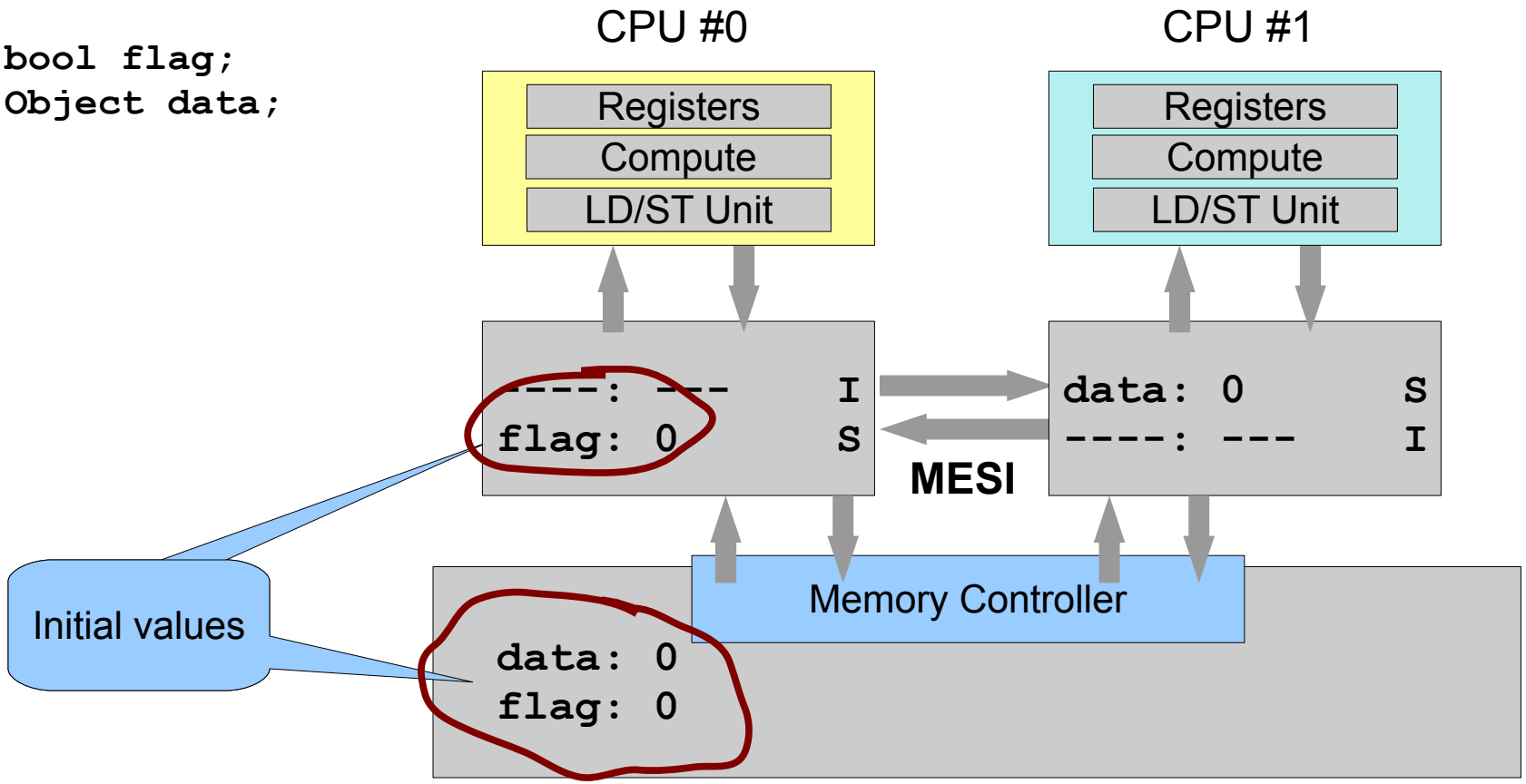
Real Chips Reorder Stuff

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



Real Chips Reorder Stuff

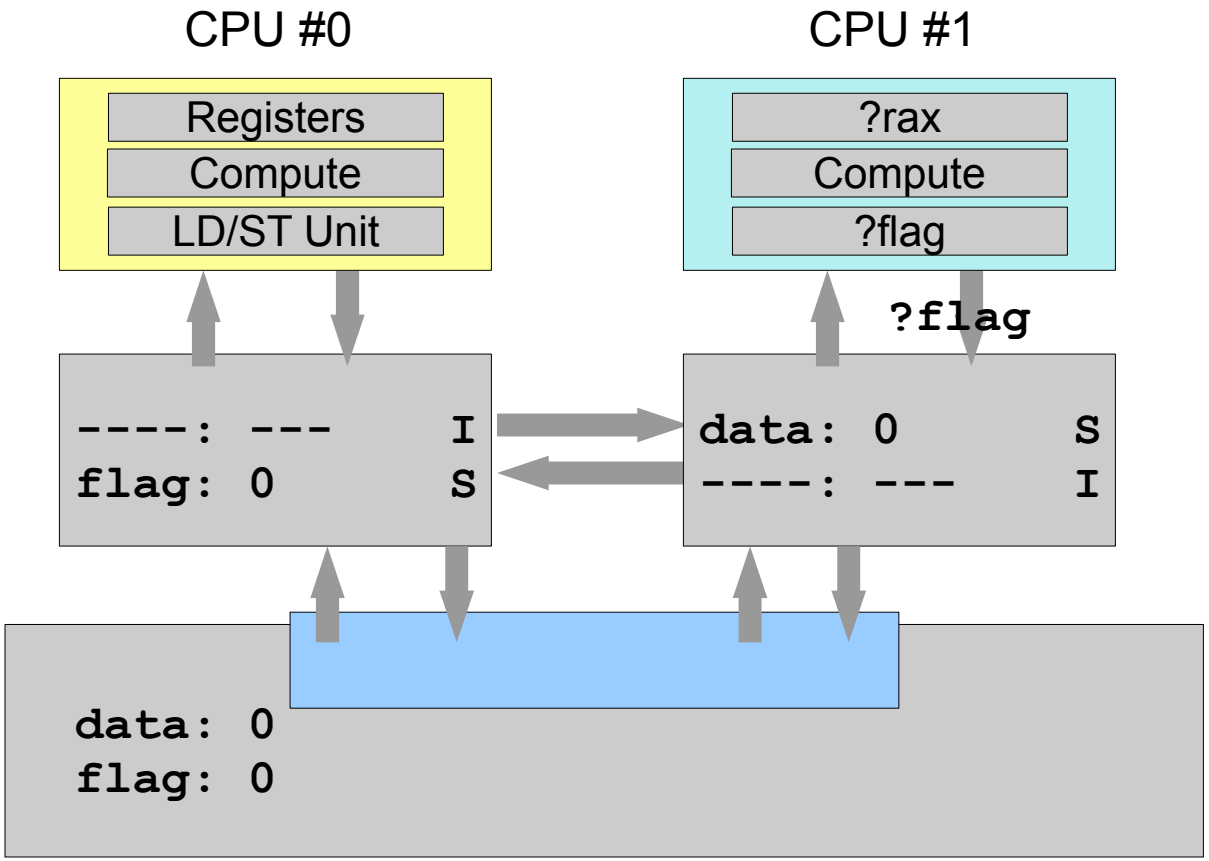
```
bool flag;
Object data;
```



Real Chips Reorder Stuff

```
if( !flag ) ...
```

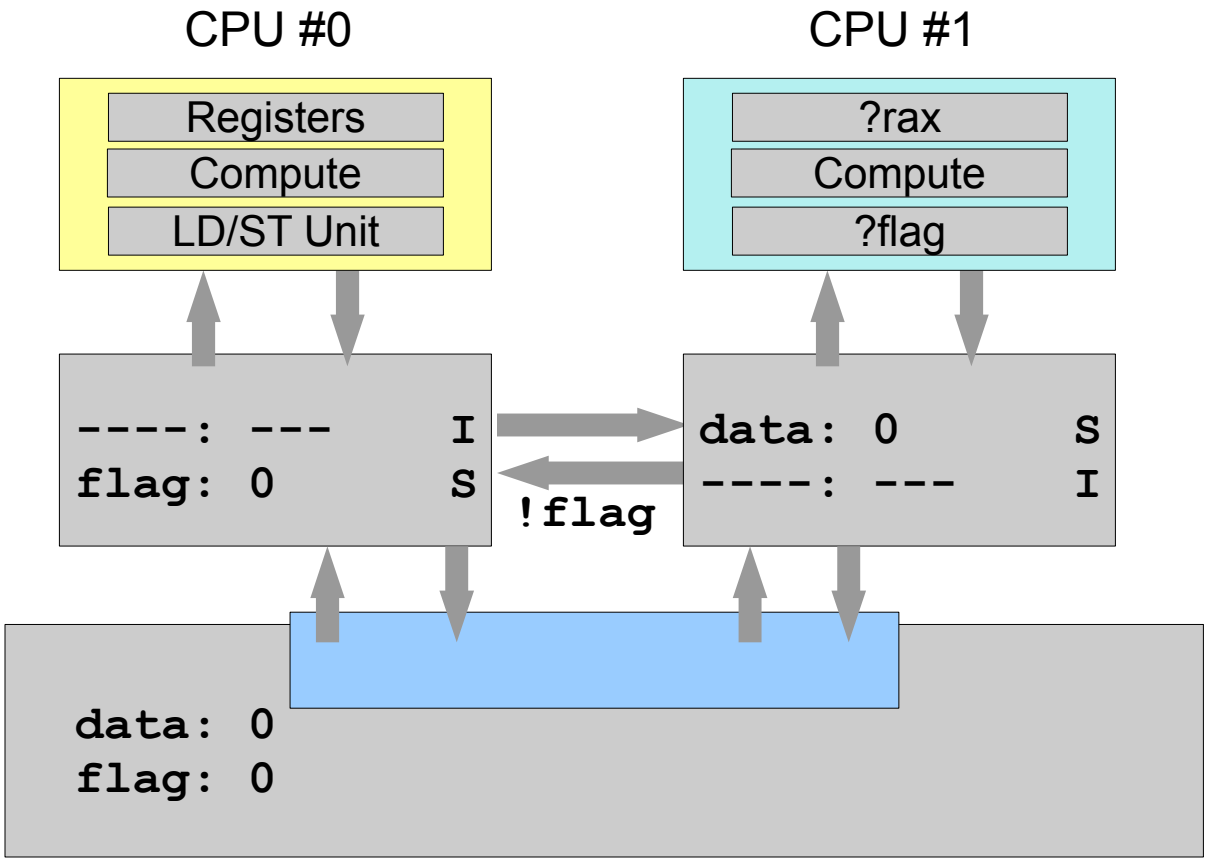
```
ld   rax,[&flag]
```



Real Chips Reorder Stuff

if(!~~flag~~) ...

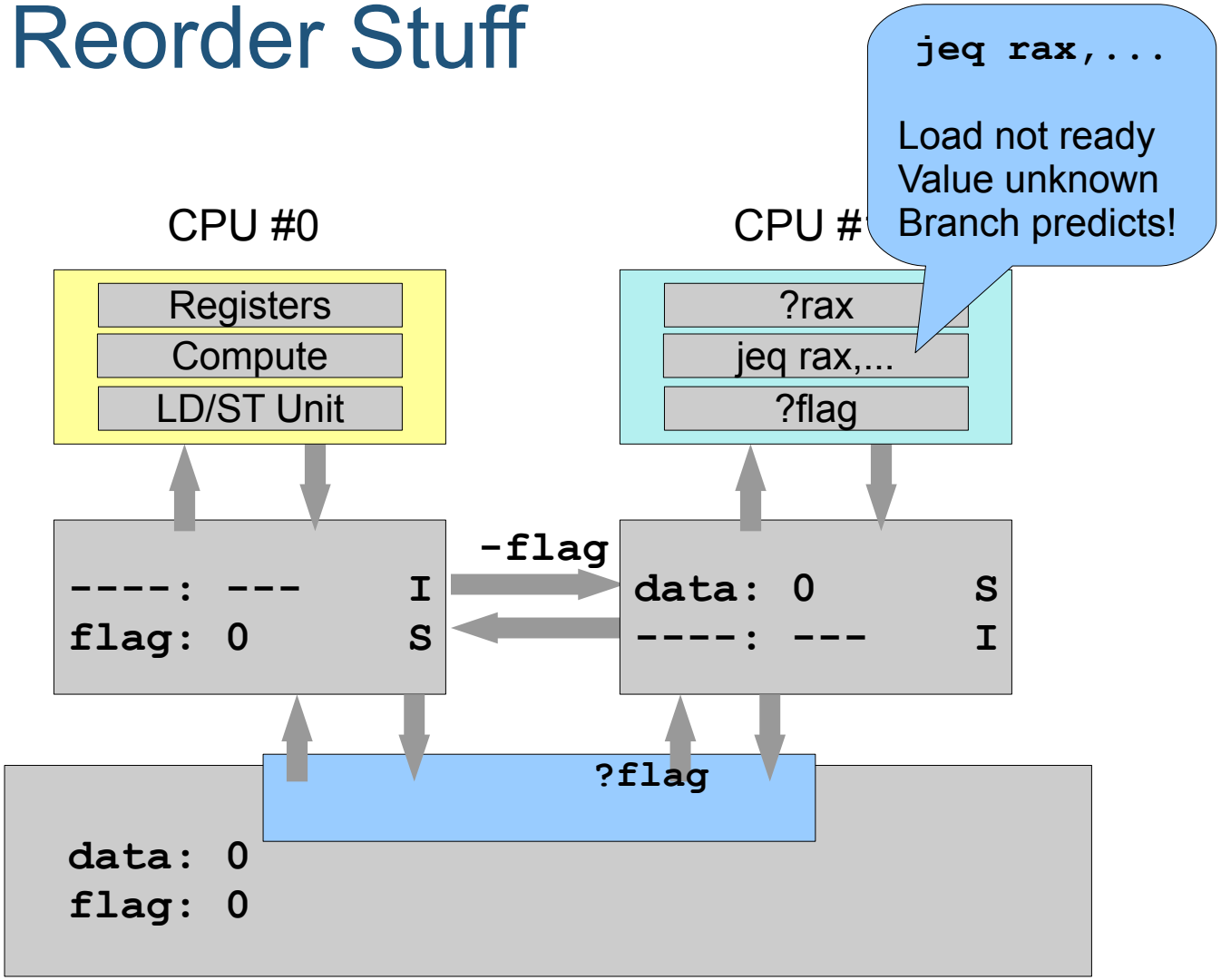
~~ld rax,[&flag]~~



Real Chips Reorder Stuff

if(!~~flag~~) ...

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



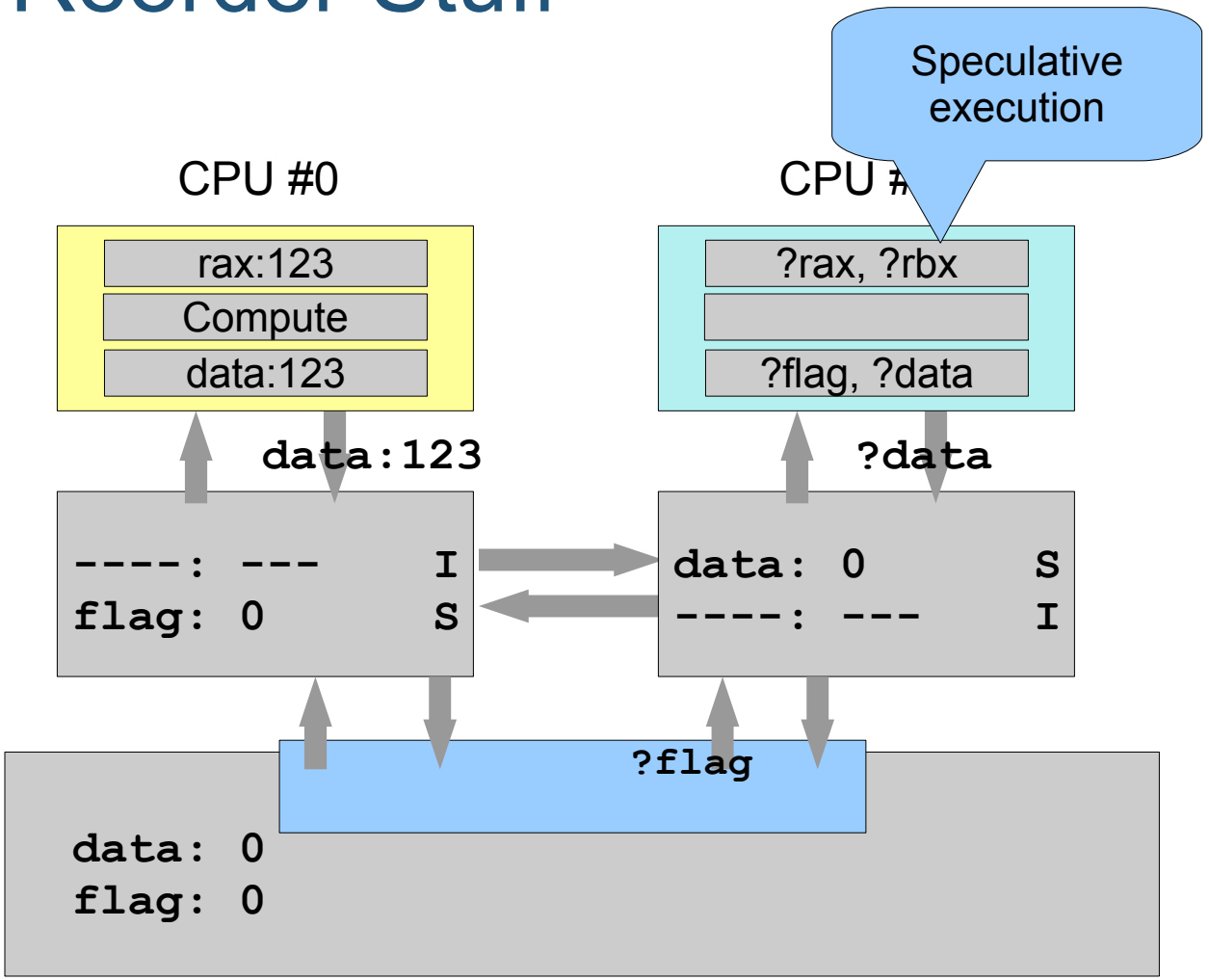
Real Chips Reorder Stuff

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

```
if( !flag ) …
return data;
```

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

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



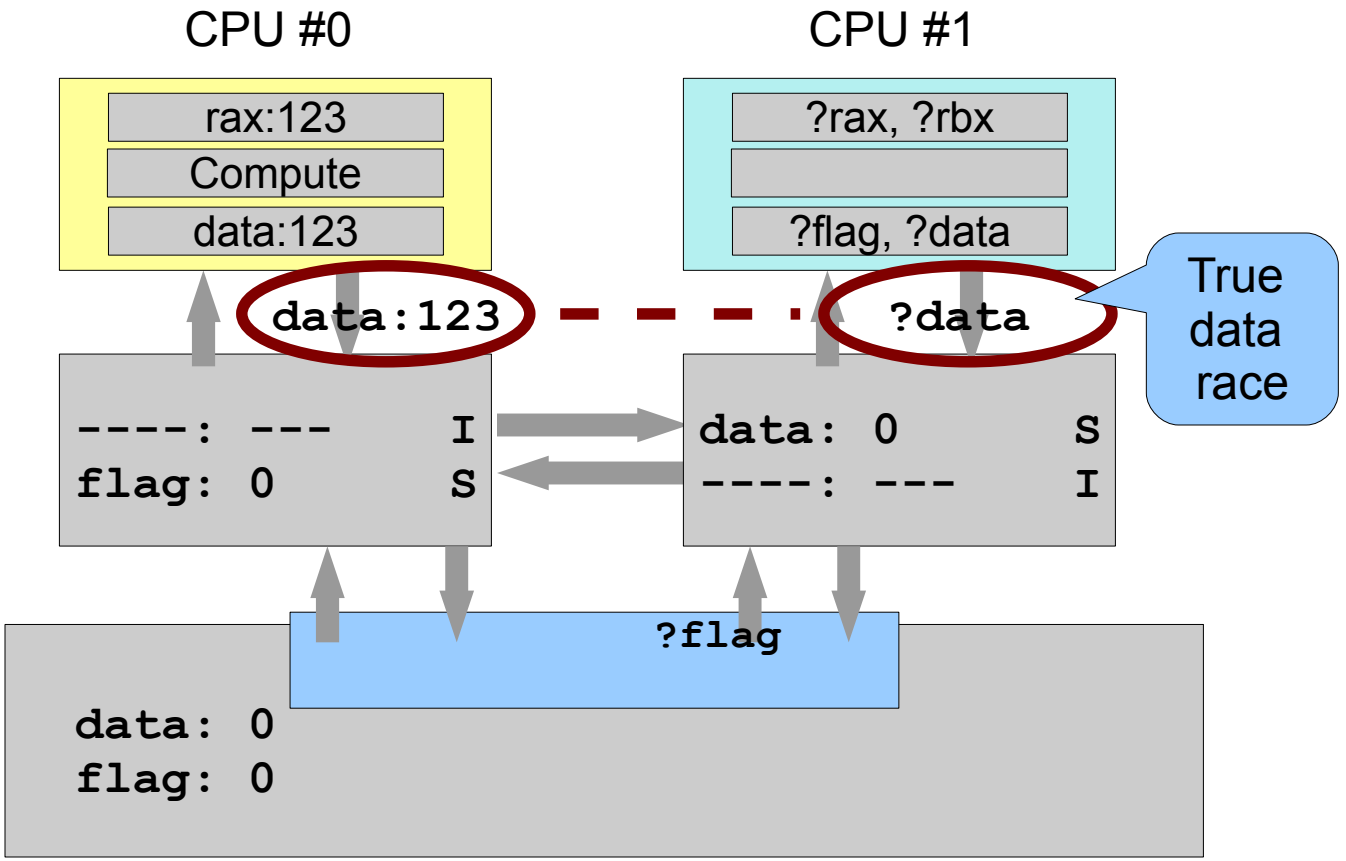
Real Chips Reorder Stuff

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

```
if( !flag ) …
return data;
```

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

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



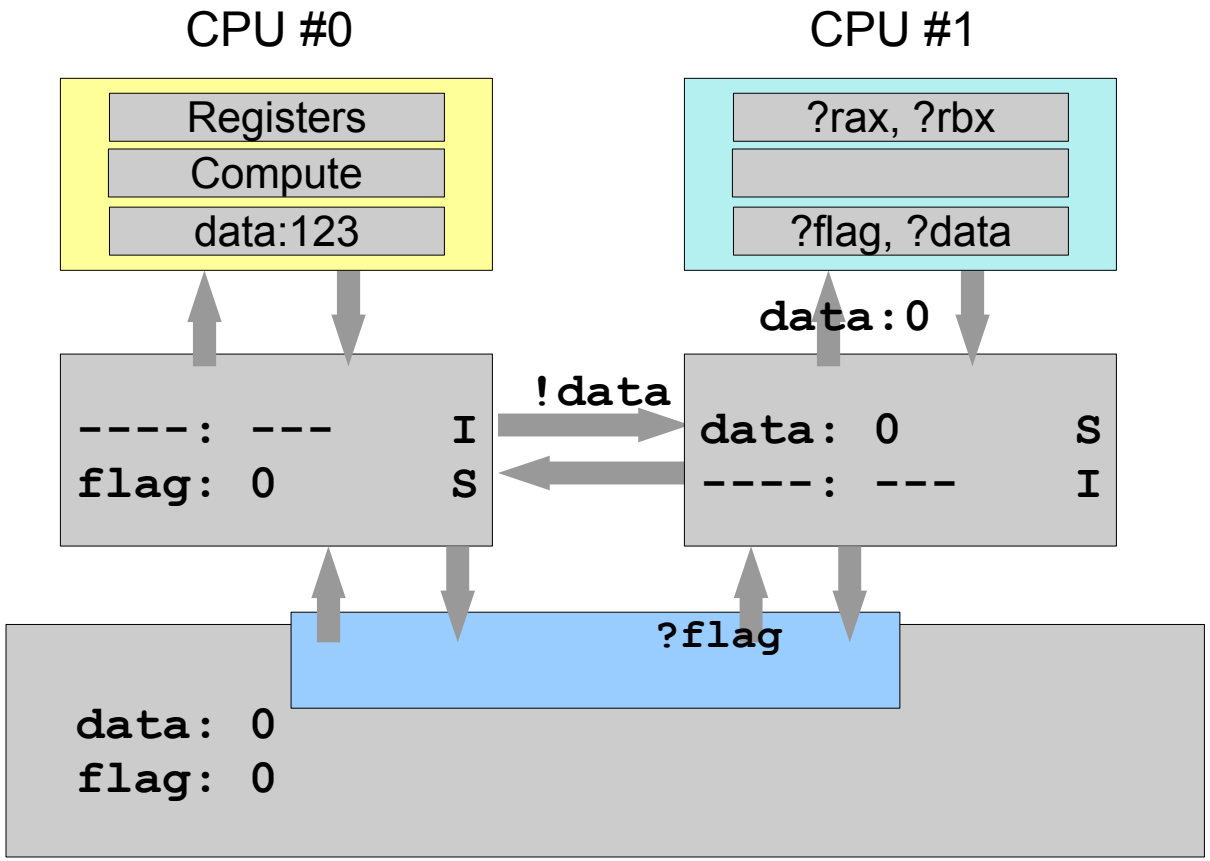
Real Chips Reorder Stuff

~~data = ...;~~

if(!~~flag~~) …
return data;

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

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



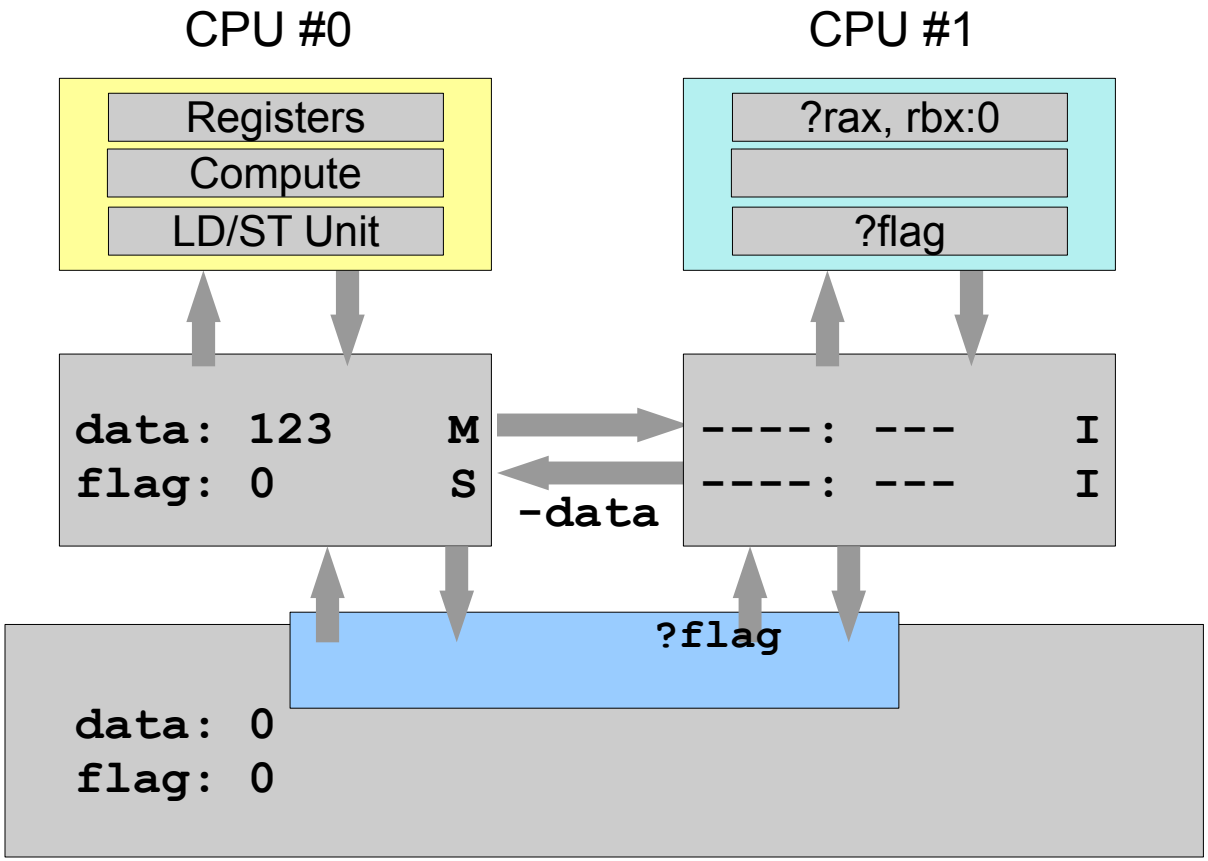
Real Chips Reorder Stuff

~~data = ...;~~

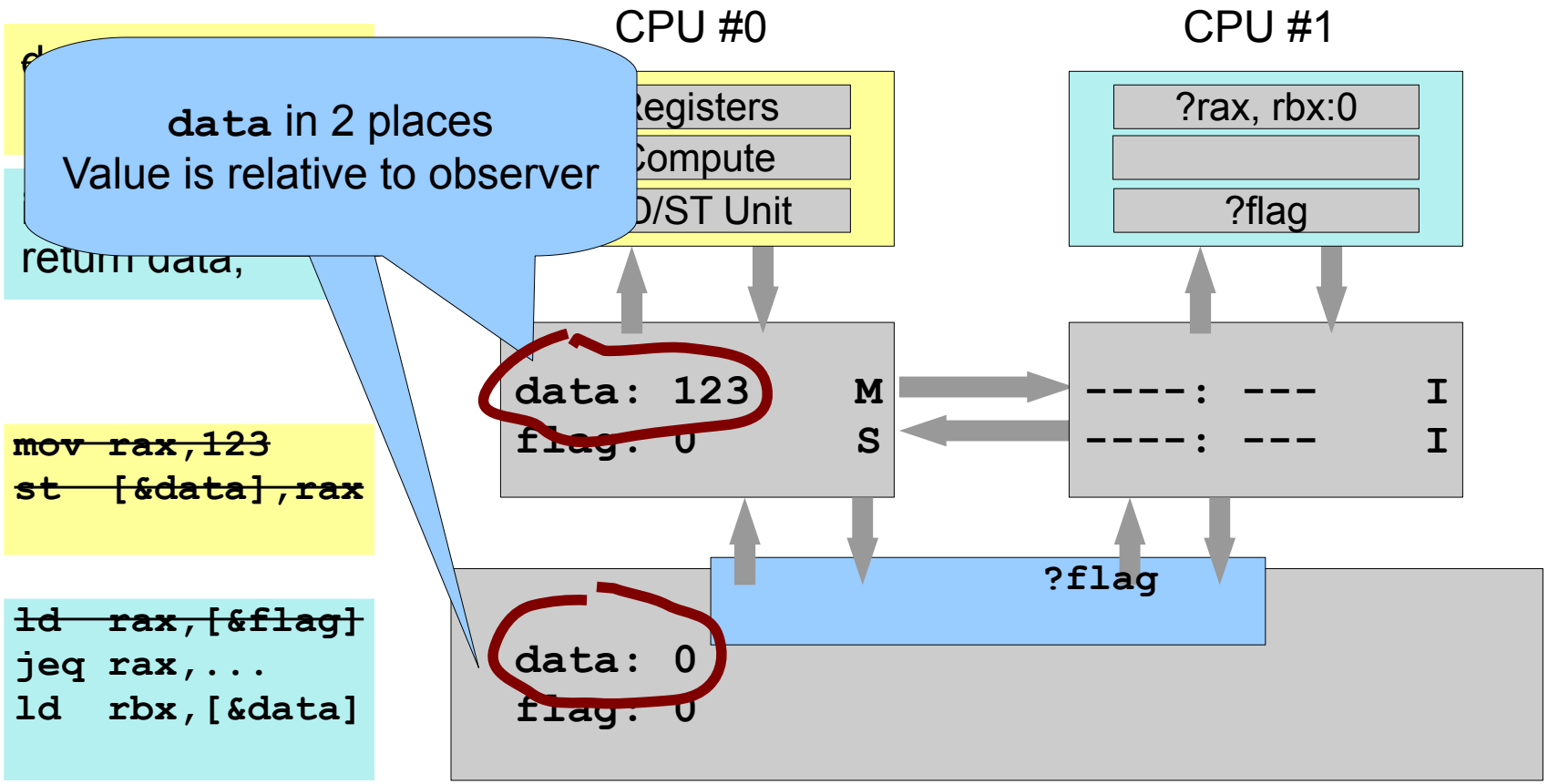
if(!~~flag~~) …
return data;

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

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



Real Chips Reorder Stuff



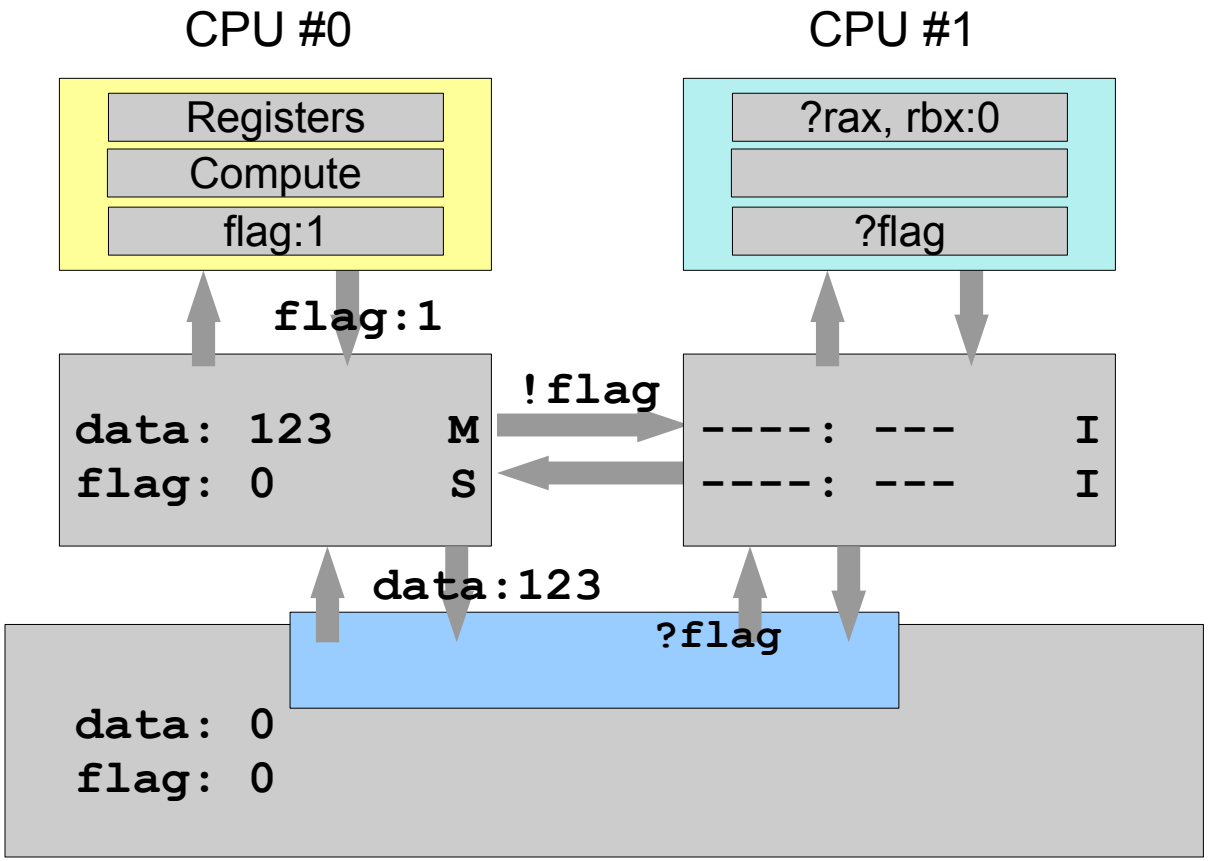
Real Chips Reorder Stuff

~~data = ...;~~
flag=true;

if(!~~flag~~) …
return data;

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

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



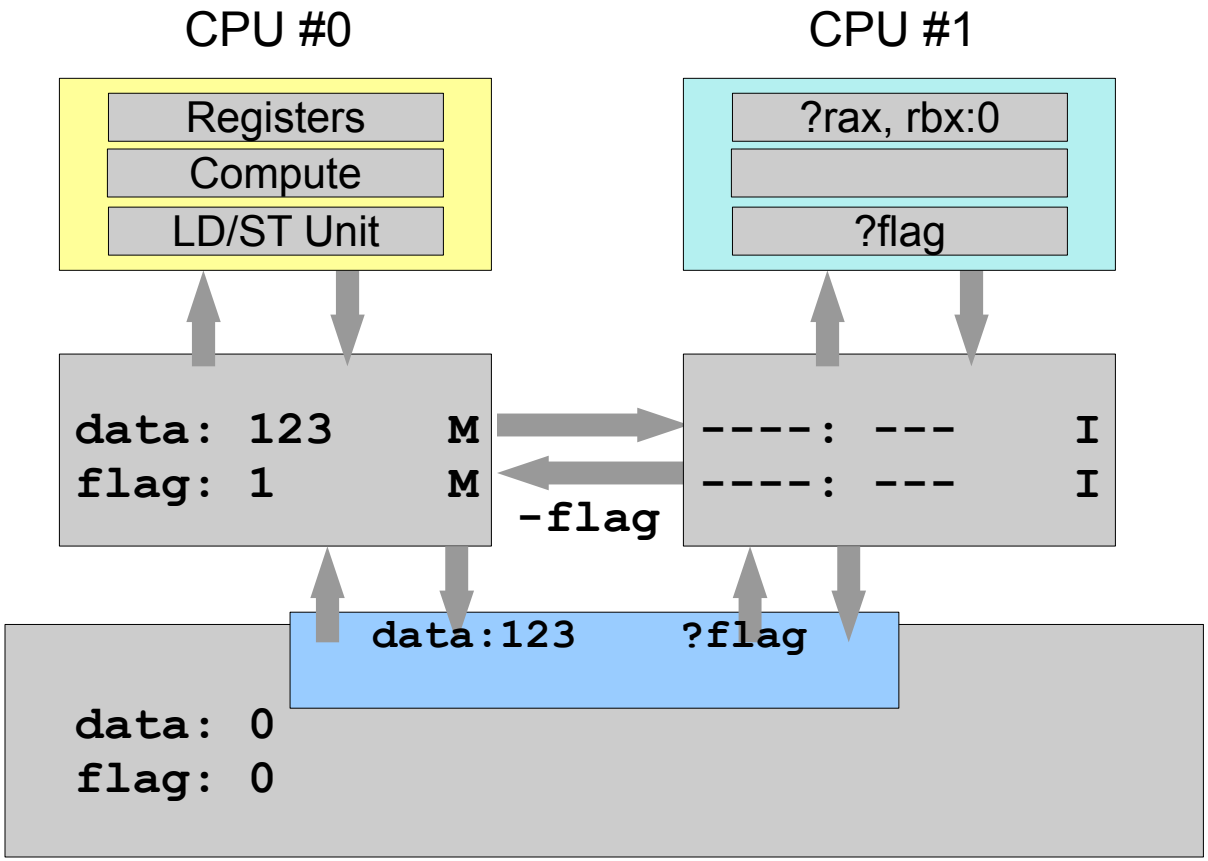
Real Chips Reorder Stuff

~~data = ...;~~
~~flag=true;~~

if(!~~flag~~) …
return data;

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

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



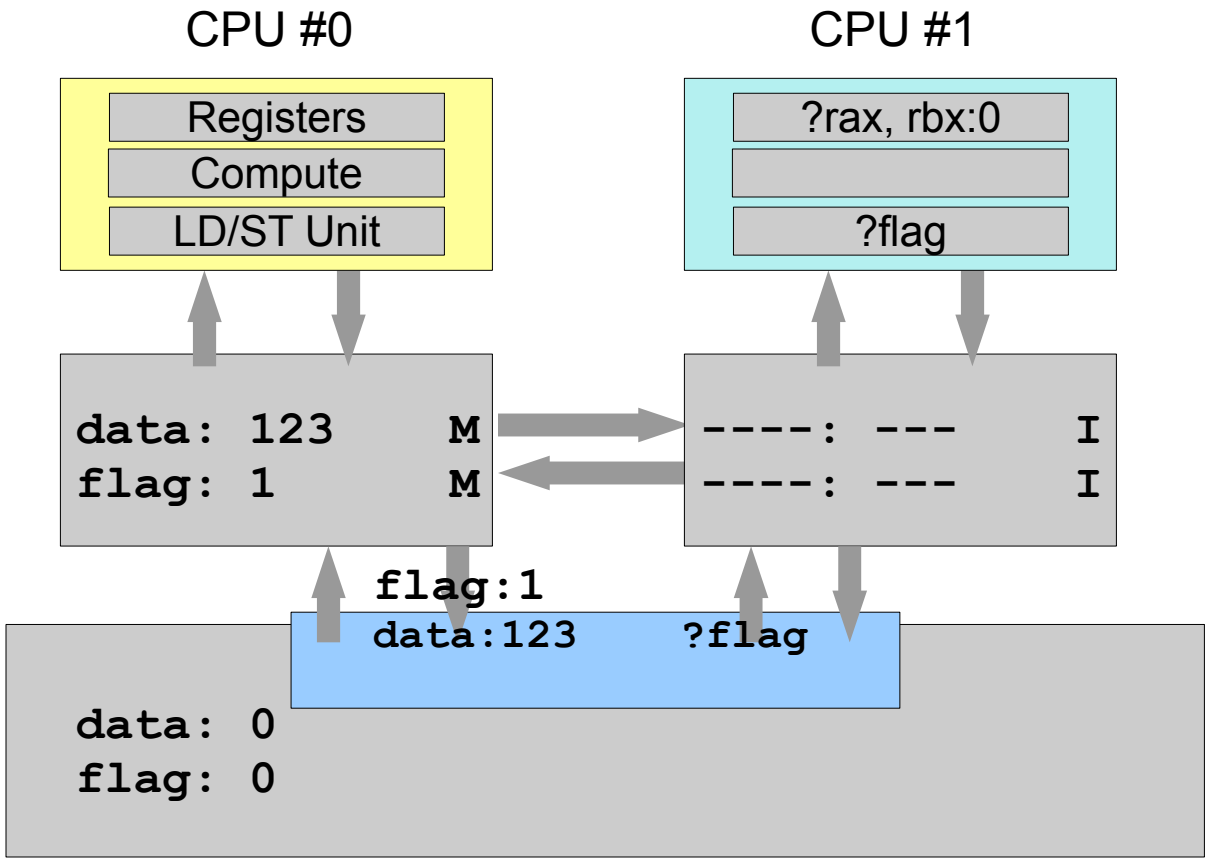
Real Chips Reorder Stuff

~~data = ...;~~
~~flag=true;~~

if(!~~flag~~) …
return data;

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

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



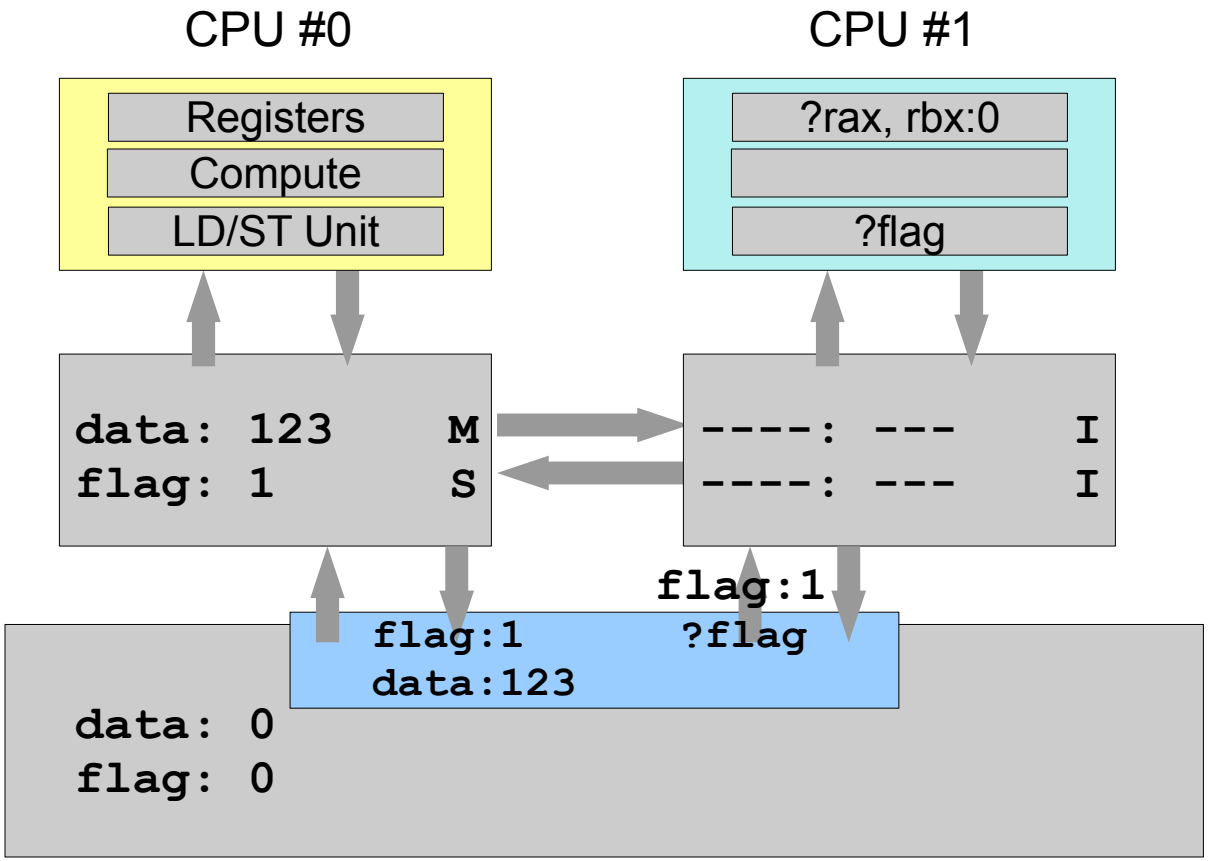
Real Chips Reorder Stuff

~~data = ...;~~
~~flag=true;~~

if(!~~flag~~) …
return data;

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

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



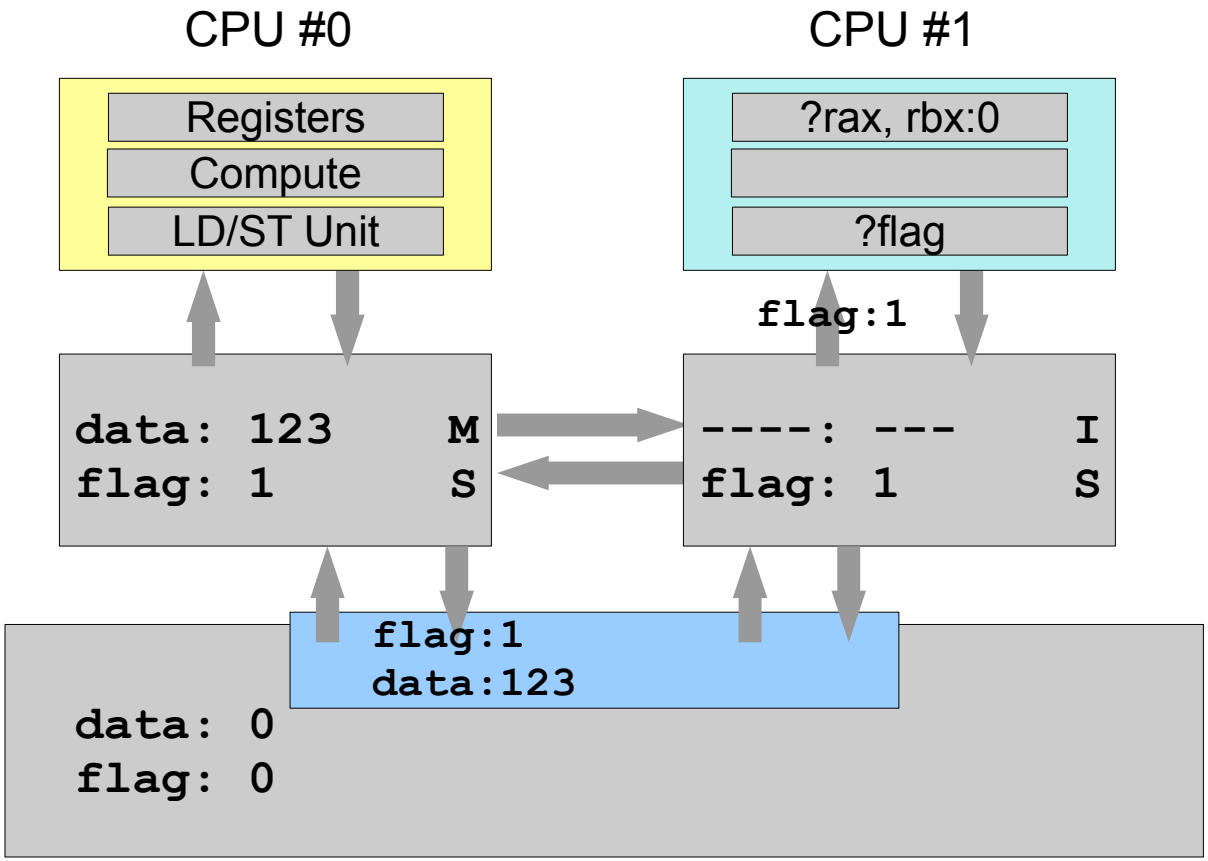
Real Chips Reorder Stuff

~~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
```